

Chester Ismay, Albert Y. Kim, and Arturo Valdivia

Statistical Inference via Data Science: A ModernDive into R and the Tidyverse

Chester: To Randy, who first got me interested in teaching R and has been a great mentor, teacher, and friend. I am grateful for the opportunities you have given me to teach and learn from others and your continued support of my work.

Albert: To Ginna: Thanks for tolerating my playing of “Nothing In This World Will Ever Break My Heart Again” on repeat while I finished this book. I love you.

Contents

Foreword	xv
Preface	xvii
About the authors	xxix
1 Getting Started with Data in R	1
1.1 What are R and RStudio?	1
1.1.1 Installing R and RStudio	2
1.1.2 Using R via RStudio	2
1.2 How do I code in R?	4
1.2.1 Basic programming concepts and terminology	4
1.2.2 Errors, warnings, and messages	5
1.2.3 Tips on learning to code	7
1.3 What are R packages?	7
1.3.1 Package installation	9
1.3.2 Package loading	10
1.3.3 Package use	11
1.4 Explore your first datasets	11
1.4.1 <code>nycflights23</code> package	12
1.4.2 <code>flights</code> data frame	13
1.4.3 Exploring data frames	14
1.4.4 Identification and measurement variables	17
1.4.5 Help files	18
1.5 Conclusion	19
1.5.1 Additional resources	19
1.5.2 What's to come?	19

I Data Science with tidyverse	21
2 Data Visualization	23
2.1 The grammar of graphics	23
2.1.1 Components of the grammar	24
2.1.2 Gapminder data	24
2.1.3 Other components	26
2.1.4 ggplot2 package	26
2.2 Five named graphs - the 5NG	27
2.3 5NG#1: Scatterplots	27
2.3.1 Scatterplots via <code>geom_point</code>	28
2.3.2 Overplotting	30
2.3.3 Summary	34
2.4 5NG#2: Linegraphs	34
2.4.1 Linegraphs via <code>geom_line</code>	35
2.4.2 Summary	37
2.5 5NG#3: Histograms	37
2.5.1 Histograms via <code>geom_histogram</code>	38
2.5.2 Adjusting the bins	40
2.5.3 Summary	42
2.6 Facets	42
2.7 5NG#4: Boxplots	44
2.7.1 Boxplots via <code>geom_boxplot</code>	47
2.7.2 Summary	49
2.8 5NG#5: Barplots	50
2.8.1 Barplots via <code>geom_bar</code> or <code>geom_col</code>	51
2.8.2 Must avoid pie charts!	53
2.8.3 Two categorical variables	55
2.8.4 Summary	59
2.9 Conclusion	59
2.9.1 Summary table	59
2.9.2 Function argument specification	60

<i>Contents</i>	vii
2.9.3 Additional resources	61
2.9.4 What's to come	61
3 Data Wrangling	63
3.1 The pipe operator: <code> ></code>	64
3.2 <code>filter</code> rows	66
3.3 <code>summarize</code> variables	69
3.4 <code>group_by</code> rows	72
3.5 <code>mutate</code> existing variables	79
3.6 <code>arrange</code> and sort rows	83
3.7 <code>join</code> data frames	85
3.7.1 Matching “key” variable names	86
3.7.2 Different “key” variable names	87
3.7.3 Multiple “key” variables	88
3.7.4 Normal forms	89
3.8 Other verbs	90
3.8.1 <code>select</code> variables	90
3.8.2 <code>relocate</code> variables	92
3.8.3 <code>rename</code> variables	92
3.8.4 <code>top_n</code> values of a variable	92
3.9 Conclusion	93
3.9.1 Summary table	93
3.9.2 Additional resources	95
3.9.3 What's to come?	96
4 Data Importing and “Tidy” Data	97
4.1 Importing data	98
4.1.1 Using the console	98
4.1.2 Using RStudio’s interface	99
4.2 “Tidy” data	100
4.2.1 Definition of “tidy” data	103
4.2.2 Converting to “tidy” data	105

4.2.3 <code>nycflights23</code> package	110
4.3 Case study: Democracy in Guatemala	110
4.4 <code>tidyverse</code> package	113
4.5 Conclusion	114
4.5.1 Additional resources	114
4.5.2 What's to come?	115
II Statistical Modeling with <code>moderndive</code>	117
5 Simple Linear Regression	119
5.1 One numerical explanatory variable	121
5.1.1 Exploratory data analysis	121
5.1.2 Simple linear regression	129
5.1.3 Observed/fitted values and residuals	132
5.2 One categorical explanatory variable	135
5.2.1 Exploratory data analysis	136
5.2.2 Linear regression	143
5.2.3 Observed/fitted values and residuals	146
5.3 Related topics	148
5.3.1 Correlation is not necessarily causation	148
5.3.2 Best-fitting line	151
5.3.3 <code>get_regression_x()</code> functions	155
5.4 Conclusion	156
5.4.1 Additional resources	156
5.4.2 What's to come?	157
6 Multiple Regression	159
6.1 One numerical and one categorical explanatory variable	159
6.1.1 Exploratory data analysis	160
6.1.2 Model with interactions	163
6.1.3 A model without interactions	168
6.1.4 Observed responses, fitted values and residuals	170
6.2 Two numerical explanatory variables	172

<i>Contents</i>	ix
6.2.1 Exploratory data analysis	172
6.2.2 Multiple regression with two numerical regressors	178
6.2.3 Observed/fitted values and residuals	180
6.3 Conclusion	180
6.3.1 Additional resources	180
6.3.2 What's to come?	181
III Statistical Inference with <code>infer</code>	183
7 Sampling	185
7.1 Sampling bowl activity	185
7.1.1 What proportion of this bowl's balls are red?	186
7.1.2 Using the shovel once	186
7.1.3 Using the shovel 33 times	188
7.1.4 What did we just do?	191
7.2 Virtual sampling	192
7.2.1 Using the virtual shovel once	192
7.2.2 Using the virtual shovel 33 times	196
7.2.3 Using the virtual shovel 1000 times	199
7.2.4 Using different shovels	202
7.3 Sampling framework	206
7.3.1 Terminology and notation	206
7.3.2 Statistical definitions	211
7.3.3 The moral of the story	214
7.4 Case study: Polls	217
7.5 Central Limit Theorem	221
7.6 Conclusion	223
7.6.1 Sampling scenarios	223
7.6.2 Theory-based standard-errors	224
7.6.3 Additional resources	225
7.6.4 What's to come?	225

8 Bootstrapping and Confidence Intervals	227
8.1 Pennies activity	229
8.1.1 What is the average year on US pennies in 2019?	229
8.1.2 Resampling once	233
8.1.3 Resampling 35 times	238
8.1.4 What did we just do?	240
8.2 Computer simulation of resampling	241
8.2.1 Virtually resampling once	241
8.2.2 Virtually resampling 35 times	243
8.2.3 Virtually resampling 1000 times	245
8.3 Understanding confidence intervals	248
8.3.1 Percentile method	249
8.3.2 Standard error method	250
8.4 Constructing confidence intervals	252
8.4.1 Original workflow	252
8.4.2 <code>infer</code> package workflow	253
8.4.3 Percentile method with <code>infer</code>	261
8.4.4 Standard error method with <code>infer</code>	262
8.5 Interpreting confidence intervals	264
8.5.1 Did the net capture the fish?	265
8.5.2 Precise and shorthand interpretation	274
8.5.3 Width of confidence intervals	275
8.6 Case study: Is yawning contagious?	278
8.6.1 <i>Mythbusters</i> study data	278
8.6.2 Sampling scenario	279
8.6.3 Constructing the confidence interval	281
8.6.4 Interpreting the confidence interval	288
8.7 Conclusion	289
8.7.1 Comparing bootstrap and sampling distributions	289
8.7.2 Theory-based confidence intervals	294
8.7.3 Additional resources	298
8.7.4 What's to come?	298

9 Hypothesis Testing	299
9.1 Music popularity activity	300
9.1.1 Is metal music more popular than deep house music?	300
9.1.2 Shuffling once	303
9.1.3 What did we just do?	307
9.2 Understanding hypothesis tests	308
9.3 Conducting hypothesis tests	310
9.3.1 <code>infer</code> package workflow	312
9.3.2 Comparison with confidence intervals	319
9.3.3 “There is only one test”	321
9.4 Interpreting hypothesis tests	322
9.4.1 Two possible outcomes	323
9.4.2 Types of errors	324
9.4.3 How do we choose alpha?	325
9.5 Case study: Are action or romance movies rated higher?	327
9.5.1 IMDb ratings data	327
9.5.2 Sampling scenario	330
9.5.3 Conducting the hypothesis test	331
9.6 Conclusion	336
9.6.1 Theory-based hypothesis tests	336
9.6.2 When inference is not needed	344
9.6.3 Problems with p-values	347
9.6.4 Additional resources	347
9.6.5 What’s to come	348
10 Inference for Regression	349
10.1 Regression refresher	349
10.1.1 Teaching evaluations analysis	350
10.1.2 Sampling scenario	352
10.2 Interpreting regression tables	353
10.2.1 Standard error	354
10.2.2 Test statistic	354

10.2.3 p-value	355
10.2.4 Confidence interval	356
10.2.5 How does R compute the table?	358
10.3 Conditions for inference for regression	358
10.3.1 Residuals refresher	359
10.3.2 Linearity of relationship	360
10.3.3 Independence of residuals	361
10.3.4 Normality of residuals	362
10.3.5 Equality of variance	364
10.3.6 What's the conclusion?	366
10.4 Simulation-based inference for regression	368
10.4.1 Confidence interval for slope	368
10.4.2 Hypothesis test for slope	372
10.5 Conclusion	375
10.5.1 Theory-based inference for regression	375
10.5.2 Summary of statistical inference	376
10.5.3 Additional resources	377
10.5.4 What's to come	377
IV Conclusion	379
11 Tell Your Story with Data	381
11.1 Review	381
11.2 Case study: Seattle house prices	384
11.2.1 Exploratory data analysis: Part I	384
11.2.2 Exploratory data analysis: Part II	391
11.2.3 Regression modeling	394
11.2.4 Making predictions	396
11.3 Case study: Effective data storytelling	398
11.3.1 Bechdel test for Hollywood gender representation	398
11.3.2 US Births in 1999	399
11.3.3 Scripts of R code	402

<i>Contents</i>	xiii
Appendix A Statistical Background	405
A.1 Basic statistical terms	405
A.1.1 Mean	405
A.1.2 Median	405
A.1.3 Standard deviation and variance	405
A.1.4 Five-number summary	406
A.1.5 Distribution	406
A.1.6 Outliers	406
A.2 Normal distribution	406
A.3 log ₁₀ transformations	409
Appendix B Versions of R Packages Used	411
Bibliography	413
Index	415

This work by Chester Ismay¹, Albert Y. Kim², and Arturo Valdivia is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

¹<https://chester.rbind.io/>

²<https://rudeboybert.rbind.io/>

Foreword

These are exciting times in statistics and data science education. (I am predicting this statement will continue to be true regardless of whether you are reading this foreword in 2020 or 2050.) But (isn't there always a but?), as a statistics educator, it can also feel a bit overwhelming to stay on top of all the new statistical, technological, and pedagogical innovations. I find myself constantly asking, "Am I teaching my students the correct content, with the relevant software, and in the most effective way?" Before I make all of us feel lost at sea, let me point out how great a life raft I have found in *ModernDive*. In a sea of intro stats textbooks, *ModernDive* floats to the top of my list, and let me tell you why. (Note my use of *ModernDive* here refers to the book in its shortened title version. This also matches up nicely with the neat hex sticker³ Drs. Ismay and Kim created for the cover of *ModernDive*, too.)

Why I ❤️ ModernDive

- * Provides students experience with the whole data analysis ↗ line.
- * Incorporates contemporary, user-friendly R packages directly into the text.
- * Emphasizes models that prepare students for our multivariate 🌎.

My favorite aspect of *ModernDive*, if I must pick a favorite, is that students gain experience with the whole data analysis pipeline (see Figure 2). In particular, *ModernDive* is one of the few intro stats textbooks that teaches students how to wrangle data. And, while data cleaning may not be as groovy as model building, it's often a prerequisite step! The world is full of messy data and *ModernDive* equips students to transform their data via the `dplyr` package.

Speaking of `dplyr`, students of *ModernDive* are exposed to the `tidyverse` suite of R packages. Designed with a common structure, `tidyverse` functions are written to be

³https://moderndive.com/images/logos/hex_blue_text.png

easy to learn and use. And, since most intro stats students are programming newbies, *ModernDive* carefully walks the students through each new function it presents and provides frequent reinforcement through the many *Learning checks* dispersed throughout the chapters.

Overall, *ModernDive* includes wise choices for the placement of topics. Starting with data visualization, *ModernDive* gets students building `ggplot2` graphs early on and then continues to reinforce important concepts graphically throughout the book. After moving through data wrangling and data importing, modeling plays a prominent role, with two chapters devoted to building regression models and a later chapter on inference for regression. Lastly, statistical inference is presented through a computational lens with calculations done via the `infer` package.

I first met Drs. Ismay and Kim while attending their workshop at the 2017 US Conference on Teaching Statistics⁴. They pushed us as participants to put data first and to use computers, instead of math, as the engine for statistical inference. That experience helped me modernize my own intro stats course and introduced me to two really forward-thinking statistics and data science educators. It has been exciting to see *ModernDive* develop and grow into such a wonderful, timely textbook. I hope you have decided to dive on in!

Kelly S. McConville, Harvard University

⁴<https://www.causeweb.org/cause/uscdots/uscdots17/workshop/3>

Preface



**Help! I'm completely new to coding and I need to learn R and RStudio!
What do I do?**

If you're asking yourself this question, then you've come to the right place! Start with the "Introduction for students" section.

- *Are you an instructor hoping to use this book in your courses? We recommend reading the "Introduction for students" section first. Then, read the "Introduction for instructors" section for more information on how to teach with this book.*
 - *Are you looking to connect with and contribute to ModernDive? Then, read the "Connect and contribute" section for information on how.*
 - *Are you curious about the publishing of this book? Then, read the "About this book" section for more information on the open-source technology, in particular R Markdown and the bookdown package.*
-

Introduction for students

This book assumes no prerequisites: no algebra, no calculus, and no prior programming/coding experience. This is intended to be a gentle introduction to the practice of analyzing data and answering questions using data the way data scientists, statisticians, data journalists, and other researchers would.

We present a map of your upcoming journey in Figure 1.



FIGURE 1: *ModernDive* flowchart.

You'll first get started with data in Chapter 1 where you'll learn about the difference between R and RStudio, start coding in R, install and load your first R packages, and explore your first dataset: all domestic departure flights from a New York City airport in 2023. Then you'll cover the following three portions of this book (Parts 2 and 4 are combined into a single portion):

1. Data science with `tidyverse`. You'll assemble your data science toolbox using `tidyverse` packages. In particular, you'll
 - Ch.2: Visualize data using the `ggplot2` package.
 - Ch.3: Wrangle data using the `dplyr` package.
 - Ch.4: Learn about the concept of “tidy” data as a standardized data input and output format for all packages in the `tidyverse`. Furthermore, you'll learn how to import spreadsheet files into R using the `readr` package.
2. Statistical/Data modeling with `moderndive`. Using these data science tools and helper functions from the `moderndive` package, you'll fit your first data models. In particular, you'll
 - Ch.5: Discover basic regression models with only one explanatory variable.

- Ch.6: Examine multiple regression models with more than one explanatory variable.
3. Statistical inference with `infer`. Once again using your newly acquired data science tools, you'll unpack statistical inference using the `infer` package. In particular, you'll:
- Ch.7: Learn about the role that sampling variability plays in statistical inference and the role that sample size plays in this sampling variability.
 - Ch.8: Construct confidence intervals using bootstrapping.
 - Ch.9: Conduct hypothesis tests using permutation.
4. Statistical/Data modeling with `moderndive` (revisited): Armed with your understanding of statistical inference, you'll revisit and review the models you've constructed in Ch.5 and Ch.6. In particular, you'll:
- Ch.10: Interpret confidence intervals and hypothesis tests in a regression setting.

We'll end with a discussion on what it means to "tell your story with data" in Chapter 11 by presenting example case studies.⁵

What we hope you will learn from this book

We hope that by the end of this book, you'll have learned how to:

1. Use R and the `tidyverse` suite of R *packages* for data science.
2. Fit your first *models* to data, using a method known as *linear regression*.
3. Perform *statistical inference* using *sampling*, *confidence intervals*, and *hypothesis tests*.
4. *Tell your story with data* using these tools.

What do we mean by data stories? We mean any analysis involving data that engages the reader in answering questions with careful visuals and thoughtful discussion. Further discussions on data stories can be found in the blog post "Tell a Meaningful Story With Data."⁶

Over the course of this book, you will develop your "data science toolbox," equipping yourself with tools such as data visualization, data formatting, data wrangling, and statistical/data modeling using regression.

⁵Note that you'll see different versions of the word "ModernDive" in this book: (1) `moderndive` refers to the R package. (2) *ModernDive* is an abbreviated version of *Statistical Inference via Data Science: A ModernDive into R and the Tidyverse*. It's essentially a nickname we gave the book. (3) *ModernDive* (without italics) corresponds to both the book and the corresponding R package together as an entity.

⁶<https://www.thinkwithgoogle.com/marketing-resources/data-measurement/tell-meaningful-stories-with-data/>

In particular, this book will lean heavily on data visualization. In today’s world, we are bombarded with graphics that attempt to convey ideas. We will explore what makes a good graphic and what the standard ways are used to convey relationships within data. In general, we’ll use visualization as a way of building almost all of the ideas in this book.

To impart the statistical lessons of this book, we have intentionally minimized the number of mathematical formulas used. Instead, you’ll develop a conceptual understanding of statistics using data visualization and computer simulations. We hope this is a more intuitive experience than the way statistics has traditionally been taught in the past and how it is commonly perceived.

Finally, you’ll learn the importance of literate programming. By this we mean you’ll learn how to write code that is useful not just for a computer to execute, but also for readers to understand exactly what your analysis is doing and how you did it. This is part of a greater effort to encourage reproducible research (see the “Reproducible research” subsection in this Preface for more details). Hal Abelson coined the phrase that we will follow throughout this book:

Programs must be written for people to read, and only incidentally for machines to execute.

We understand that there may be challenging moments as you learn to program. Both of us continue to struggle and find ourselves often using web searches to find answers and reach out to colleagues for help. In the long run though, we all can solve problems faster and more elegantly via programming. We wrote this book as our way to help you get started and you should know that there is a huge community of R users that are happy to help everyone along as well. This community exists in particular on the internet on various forums and websites such as stackoverflow.com⁷.

Data/science pipeline

You may think of statistics as just being a bunch of numbers. We commonly hear the phrase “statistician” when listening to broadcasts of sporting events. Statistics (in particular, data analysis), in addition to describing numbers like with baseball batting averages, plays a vital role in all of the sciences.

You’ll commonly hear the phrase “statistically significant” thrown around in the media. You’ll see articles that say, “Science now shows that chocolate is good for you.” Underpinning these claims is data analysis. By the end of this book, you’ll be able to

⁷<https://stackoverflow.com/>

better understand whether these claims should be trusted or whether we should be wary. Inside data analysis are many sub-fields that we will discuss throughout this book (though not necessarily in this order):

- data collection
- data wrangling
- data visualization
- statistical modeling
- inference
- correlation and regression
- interpretation of results
- data communication/storytelling

These sub-fields are summarized in what Garrett Grolemund and Hadley Wickham have previously termed the “data/science pipeline”⁸ in Figure 2.



FIGURE 2: Data/science pipeline.

We will begin by digging into the grey **Understand** portion of the cycle with data visualization, then with a discussion on what is meant by tidy data and data wrangling, and then conclude by talking about interpreting and discussing the results of our models via **Communication**. These steps are vital to any statistical analysis. But, why should you care about statistics?

There’s a reason that many fields require a statistics course. Scientific knowledge grows through an understanding of statistical significance and data analysis. You needn’t be intimidated by statistics. It’s not the beast that it used to be and, paired with computation, you’ll see how reproducible research in the sciences particularly increases scientific knowledge.

⁸<http://r4ds.had.co.nz/explore-intro.html>

Reproducible research

The most important tool is the *mindset*, when starting, that the end product will be reproducible. – Keith Baggerly

Another goal of this book is to help readers understand the importance of reproducible analyses. The hope is to get readers into the habit of making their analyses reproducible from the very beginning. This means we'll be trying to help you build new habits. This will take practice and be difficult at times. You'll see just why it is so important for you to keep track of your code and document it well to help yourself later and any potential collaborators as well.

Copying and pasting results from one program into a word processor is not an ideal way to conduct efficient and effective scientific research. It's much more important for time to be spent on data collection and data analysis and not on copying and pasting plots back and forth across a variety of programs.

In traditional analyses, if an error was made with the original data, we'd need to step through the entire process again: recreate the plots and copy-and-paste all of the new plots and our statistical analysis into our document. This is error prone and a frustrating use of time. We want to help you to get away from this tedious activity so that we can spend more time doing science.

We are talking about *computational* reproducibility. – Yihui Xie

Reproducibility means a lot of things in terms of different scientific fields. Are experiments conducted in a way that another researcher could follow the steps and get similar results? In this book, we will focus on what is known as **computational reproducibility**. This refers to being able to pass all of one's data analysis, datasets, and conclusions to someone else and have them get exactly the same results on their machine. This allows for time to be spent interpreting results and considering assumptions instead of the more error prone way of starting from scratch or following a list of steps that may be different from machine to machine.

Final note for students

At this point, if you are interested in instructor perspectives on this book, ways to contribute and collaborate, or the technical details of this book's construction and

publishing, then continue with the rest of the chapter. Otherwise, let's get started with R and RStudio in Chapter 1!

Introduction for instructors

Resources

Here are some resources to help you use *ModernDive*:

1. We've included review questions posed as *Learning checks*. You can find all the solutions to all *Learning checks* in Appendix D of the online version of the book at <https://moderndive.com/D-appendixD.html>.
2. Dr. Jenny Smetzer and Albert Y. Kim have written a series of labs and problem sets. You can find them at <https://moderndive.com/labs>.
3. You can see the webpages for two courses that use *ModernDive*:
 - Smith College “SDS192 Introduction to Data Science”: <https://rudeboybert.github.io/SDS192/>.
 - Smith College “SDS220 Introduction to Probability and Statistics”: <https://rudeboybert.github.io/SDS220/>.

Why did we write this book?

This book is inspired by

- *Mathematical Statistics with Resampling and R* ([Chihara and Hesterberg, 2011](#))
- *OpenIntro: Intro Stat with Randomization and Simulation* ([Diez et al., 2014](#))
- *R for Data Science* ([Grolemund and Wickham, 2017](#))

The first book, designed for upper-level undergraduates and graduate students, provides an excellent resource on how to use resampling to impart statistical concepts like sampling distributions using computation instead of large-sample approximations and other mathematical formulas. The last two books are free options for learning about introductory statistics and data science, providing an alternative to the many traditionally expensive introductory statistics textbooks.

When looking over the introductory statistics textbooks that currently exist, we found there wasn't one that incorporated many newly developed R packages directly into the text, in particular the many packages included in the `tidyverse`⁹ set of packages,

⁹<http://tidyverse.org/>

such as `ggplot2`, `dplyr`, `tidyverse`, and `readr` that will be the focus of this book’s first part on “Data Science with `tidyverse`.”

Additionally, there wasn’t an open-source and easily reproducible textbook available that exposed new learners to all four of the learning goals we listed in the “Introduction for students” subsection. We wanted to write a book that could develop theory via computational techniques and help novices master the R language in doing so.

Who is this book for?

This book is intended for instructors of traditional introductory statistics classes using RStudio, who would like to inject more data science topics into their syllabus. RStudio can be used in either the server version or the desktop version. (This is discussed further in Subsection 1.1.1.) We assume that students taking the class will have no prior algebra, no calculus, nor programming/coding experience.

Here are some principles and beliefs we kept in mind while writing this text. If you agree with them, this is the book for you.

1. Blur the lines between lecture and lab

- With increased availability and accessibility of laptops and open-source non-proprietary statistical software, the strict dichotomy between lab and lecture can be loosened.
- It’s much harder for students to understand the importance of using software if they only use it once a week or less. They forget the syntax in much the same way someone learning a foreign language forgets the grammar rules. Frequent reinforcement is key.

2. Focus on the entire data/science research pipeline

- We believe that the entirety of Grolemund and Wickham’s data/science pipeline¹⁰ as seen in Figure 2 should be taught.
- We heed George Cobb’s call to “minimize prerequisites to research”¹¹: students should be answering questions with data as soon as possible.

3. It’s all about the data

- We leverage R packages for rich, real, and realistic datasets that at the same time are easy-to-load into R, such as the `nycflights23` and `fivethirtyeight` packages.
- We believe that data visualization is a “gateway drug” for statistics¹² and that the grammar of graphics as implemented in the `ggplot2` package is the best way to impart such lessons. However, we often hear: “You can’t teach `ggplot2` for data visualization in intro stats!” We,

¹⁰<http://r4ds.had.co.nz/introduction.html>

¹¹<https://arxiv.org/abs/1507.05346>

¹²<http://escholarship.org/uc/item/84v3774z>

like David Robinson¹³, are much more optimistic and have found our students have been largely successful in learning it.

- `dplyr` has made data wrangling much more accessible¹⁴ to novices, and hence much more interesting datasets can be explored.

4. Use simulation/resampling to introduce statistical inference, not probability/mathematical formulas

- Instead of using formulas, large-sample approximations, and probability tables, we teach statistical concepts using simulation-based inference.
- This allows for a de-emphasis of traditional probability topics, freeing up room in the syllabus for other topics. Bridges to these mathematical concepts are given as well to help with relation of these traditional topics with more modern approaches.

5. Don't fence off students from the computation pool, throw them in!

- Computing skills are essential to working with data in the 21st century. Given this fact, we feel that to shield students from computing is to ultimately do them a disservice.
- We are not teaching a course on coding/programming per se, but rather just enough of the computational and algorithmic thinking necessary for data analysis.

6. Complete reproducibility and customizability

- We are frustrated when textbooks give examples, but not the source code and the data itself. We give you the source code for all examples as well as the whole book! While we have made choices to occasionally hide the code that produces more complicated figures, reviewing the book's GitHub repository will provide you with all the code (see below).
- Ultimately the best textbook is one you've written yourself. You know best your audience, their background, and their priorities. You know best your own style and the types of examples and problems you like best. Customization is the ultimate end. We encourage you to take what we've provided and make it work for your own needs. For more about how to make this book your own, see "About this book" later in this Preface.

¹³http://varianceexplained.org/r/teach_ggplot2_to_beginners/

¹⁴<http://chance.amstat.org/2015/04/setting-the-stage/>

Connect and contribute

If you would like to connect with ModernDive, check out the following links:

- If you would like to receive periodic updates about ModernDive (roughly every 6 months), please sign up for our mailing list¹⁵.
- Contact Albert at albert.ys.kim@gmail.com, Chester at chester.ismay@gmail.com, and Arturo at arturo.valdivia@gmail.com.
- We're on X (formerly Twitter) at <https://x.com/ModernDive>.

If you would like to contribute to *ModernDive*, there are many ways! We would love your help and feedback to make this book as great as possible! For example, if you find any errors, typos, or areas for improvement, then please email us or post an issue on our GitHub issues¹⁶ page. If you are familiar with GitHub and would like to contribute, see the “About this book” section.

Acknowledgements

The authors would like to thank Nina Sonneborn¹⁷, Dr. Alison Hill¹⁸, Kristin Bott¹⁹, Dr. Jenny Smetzer, Prof. Katherine Kinnaird²⁰, and the participants of our 2017²¹ and 2019²² USCOTS workshops for their feedback and suggestions. We'd also like to thank Dr. Andrew Heiss²³ for contributing nearly all of Subsection 1.2.3 on “Errors, warnings, and messages,” Evgeni Chasnovski²⁴ for creating the new `geom_parallel_slopes()` extension to the `ggplot2` package for plotting parallel slopes models, and Smith College Statistical & Data Sciences students Starry Zhou²⁵ and Marium Tapal²⁶ for their many edits to the book. A special thanks goes to Dr. Jude Weinstein-Jones, co-founder of The Learning Scientists²⁷, for their extensive feedback.

¹⁵<http://eepurl.com/cBkItf>

¹⁶https://github.com/moderndive/moderndive_book/issues

¹⁷<https://github.com/nsonneborn>

¹⁸<https://alison.rbind.io/>

¹⁹<https://twitter.com/rhobott?lang=en>

²⁰<https://www.smith.edu/academics/faculty/katherine-kinnaird>

²¹<https://www.causeweb.org/cause/uscots/uscots17/workshop/3>

²²<https://www.causeweb.org/cause/uscots/uscots19/workshop/4>

²³<https://twitter.com/andrewheiss>

²⁴<https://github.com/echasnovski>

²⁵<https://github.com/Starryz>

²⁶<https://github.com/mariumtapal>

²⁷<https://www.learningscientists.org>

We were both honored to have Dr. Kelly S. McConville²⁸ write the **Foreword** of the book. Dr. McConville is a pioneer in statistics education and was a source of great inspiration to both of us as we continued to update the book to get it to its current form. Thanks additionally to the continued contributions by members of the community²⁹ to the book on GitHub and to the many individuals that have recommended this book to others. We are so very appreciative of all of you!

Lastly, a very special shout out to any student who has ever taken a class with us at either Pacific University, Reed College, Middlebury College, Amherst College, or Smith College. We couldn't have made this book without you!

About this book

This book was written using RStudio's bookdown³⁰ package by Yihui Xie ([Xie, 2024](#)). This package simplifies the publishing of books by having all content written in R Markdown³¹. The bookdown/R Markdown source code for all versions of ModernDive is available on GitHub:

- **Latest online version** The most up-to-date release:
 - Version 2.0.0 released on June 30, 2024 (source code³²)
 - Available at <https://moderndive.com/>
- **Print edition** The CRC Press print edition³³ of *ModernDive* corresponds to Version 1.1.0 (with some typos fixed).
- **Previous online versions** Older versions that may be out of date:
 - Version 1.0.0³⁴ released on November 25, 2019 (source code³⁵)
 - Version 0.6.1³⁶ released on August 28, 2019 (source code³⁷)
 - Version 0.6.0³⁸ released on August 7, 2019 (source code³⁹)
 - Version 0.5.0⁴⁰ released on February 24, 2019 (source code⁴¹)

²⁸<https://mcconville.rbind.io/>

²⁹https://github.com/moderndive/ModernDive_book/graphs/contributors

³⁰<https://bookdown.org/>

³¹http://rmarkdown.rstudio.com/html_document_format.html

³²https://github.com/moderndive/moderndive_book/releases/tag/v2.0.0

³³<https://www.routledge.com/Statistical-Inference-via-Data-Science-A-ModernDive-into-R-and-the-Tidyverse/Ismay-Kim/p/book/9780367409821>

³⁴[previous_versions/v1.0.0/index.html](https://github.com/moderndive/ModernDive_book/releases/tag/v1.0.0)

³⁵https://github.com/moderndive/ModernDive_book/releases/tag/v1.0.0

³⁶[previous_versions/v0.6.1/index.html](https://github.com/moderndive/ModernDive_book/releases/tag/v0.6.1)

³⁷https://github.com/moderndive/ModernDive_book/releases/tag/v0.6.1

³⁸[previous_versions/v0.6.0/index.html](https://github.com/moderndive/ModernDive_book/releases/tag/v0.6.0)

³⁹https://github.com/moderndive/moderndive_book/releases/tag/v0.6.0

⁴⁰[previous_versions/v0.5.0/index.html](https://github.com/moderndive/moderndive_book/releases/tag/v0.5.0)

⁴¹https://github.com/moderndive/moderndive_book/releases/tag/v0.5.0

- Version 0.4.0⁴² released on July 21, 2018 (source code⁴³)
- Version 0.3.0⁴⁴ released on February 3, 2018 (source code⁴⁵)
- Version 0.2.0⁴⁶ released on August 2, 2017 (source code⁴⁷)
- Version 0.1.3⁴⁸ released on February 9, 2017 (source code⁴⁹)
- Version 0.1.2⁵⁰ released on January 22, 2017 (source code⁵¹)

Could this be a new paradigm for textbooks? Instead of the traditional model of textbook companies publishing updated *editions* of the textbook every few years, we apply a software design influenced model of publishing more easily updated *versions*. We can then leverage open-source communities of instructors and developers for ideas, tools, resources, and feedback. As such, we welcome your GitHub pull requests.

Finally, since this book is under a Creative Commons Attribution - NonCommercial - ShareAlike 4.0 license⁵², feel free to modify the book as you wish for your own non-commercial needs, but please list the authors at the top of `index.Rmd` as: “Chester Ismay, Albert Y. Kim, Arturo Valdivia, and YOU!”

⁴²[previous_versions/v0.4.0/index.html](#)

⁴³https://github.com/moderndive/moderndive_book/releases/tag/v0.4.0

⁴⁴[previous_versions/v0.3.0/index.html](#)

⁴⁵https://github.com/moderndive/moderndive_book/releases/tag/v0.3.0

⁴⁶[previous_versions/v0.2.0/index.html](#)

⁴⁷https://github.com/moderndive/moderndive_book/releases/tag/v0.2.0

⁴⁸[previous_versions/v0.1.3/index.html](#)

⁴⁹https://github.com/moderndive/moderndive_book/releases/tag/v0.1.3

⁵⁰[previous_versions/v0.1.2/index.html](#)

⁵¹https://github.com/moderndive/moderndive_book/releases/tag/v0.1.2

⁵²<https://creativecommons.org/licenses/by-nc-sa/4.0/>

About the authors

Chester Ismay



Albert Y. Kim



Arturo Valdivia

Chester Ismay is Vice President of Data and Automation at MATE Seminars and is a freelance data science consultant. He also teaches in the Center for Executive and Professional Education at Portland State University. He completed his PhD in statistics from Arizona State University in 2013. He has previously worked in a variety of roles including as an actuary at Scottsdale Insurance Company (now Nationwide E&S/Specialty) and at Ripon College, Reed College, and Pacific University. He has experience working in online education and was previously a Data Science Evangelist at DataRobot, where he led data science, machine learning, and data engineering in-person and virtual workshops for DataRobot University. In addition to his work for *ModernDive*, he also contributed as initial developer of the `infer`⁵³ R package and is author and maintainer of the `thesisdown`⁵⁴ R package.

- Email: chester.ismay@gmail.com
- Webpage: <https://chester.rbind.io/>
- X (formerly Twitter): old_man_chester⁵⁵
- GitHub: <https://github.com/ismayc>

Albert Y. Kim is an Associate Professor of Statistical & Data Sciences at Smith College in Northampton, MA, USA. He completed his PhD in statistics at the University of Washington in 2011. Previously he worked in the Search Ads Metrics Team

⁵³<https://cran.r-project.org/package=infer>

⁵⁴<https://github.com/ismayc/thesisdown>

⁵⁵https://x.com/old_man_chester

at Google Inc. as well as at Reed, Middlebury, and Amherst Colleges. In addition to his work for *ModernDive*, he is a co-author of the `resampleddata`⁵⁶ and `SpatialEpi`⁵⁷ R packages.

- Email: albert.ys.kim@gmail.com
- Webpage: <https://rudeboybert.rbind.io/>
- X (formerly Twitter): [rudeboybert⁵⁸](https://twitter.com/rudeboybert)
- GitHub: <https://github.com/rudeboybert>

Both Drs. Ismay and Kim, along with Jennifer Chun⁵⁹, are co-authors of the `fivethirtyeight`⁶⁰ package of code and datasets published by the data journalism website FiveThirtyEight.com⁶¹.

⁵⁶<https://cran.r-project.org/package=resampleddata>

⁵⁷<https://cran.r-project.org/package=SpatialEpi>

⁵⁸<https://x.com/rudeboybert>

⁵⁹<https://github.com/jchunn>

⁶⁰<https://fivethirtyeight-r.netlify.app/>

⁶¹<https://fivethirtyeight.com/>

1

Getting Started with Data in R

Before we can start exploring data in R, there are some key concepts to understand first:

1. What are R and RStudio?
2. How do I code in R?
3. What are R packages?

We'll introduce these concepts in the upcoming Sections 1.1-1.3. If you are already somewhat familiar with these concepts, feel free to skip to Section 1.4 where we'll introduce our first dataset: all domestic flights departing one of the three main New York City (NYC) airports in 2023. This is a dataset we will explore in depth for much of the rest of this book.

1.1 What are R and RStudio?

Throughout this book, we will assume that you are using R via RStudio. First time users often confuse the two. At its simplest, R is like a car's engine while RStudio is like a car's dashboard as illustrated in Figure 1.1.

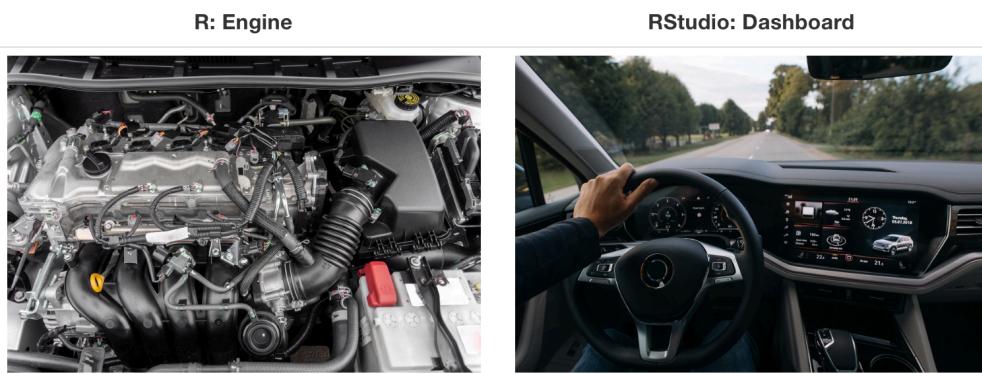


FIGURE 1.1: Analogy of difference between R and RStudio.

More precisely, R is a programming language that runs computations, while RStudio is an *integrated development environment (IDE)* that provides an interface by adding many convenient features and tools. So just as the way of having access to a speedometer, rearview mirrors, and a navigation system makes driving much easier, using RStudio's interface makes using R much easier as well.

1.1.1 Installing R and RStudio

Note about RStudio Server or RStudio Cloud: If your instructor has provided you with a link and access to RStudio Server or RStudio Cloud, then you can skip this section. We do recommend after a few months of working on RStudio Server/Cloud that you return to these instructions to install this software on your own computer though.

You will first need to download and install both R and RStudio (Desktop version) on your computer. It is important that you install R first and then install RStudio.

1. **You must do this first:** Download and install R by going to <https://cloud.r-project.org/>.
 - If you are a Windows user: Click on “Download R for Windows”, then click on “base”, then click on the Download link.
 - If you are macOS user: Click on “Download R for macOS”, then under “Latest release:” click on R-X.X.X.pkg, where R-X.X.X is the version number. For example, the latest version of R as of May 24, 2024 was R-4.4.0.
 - If you are a Linux user: Click on “Download R for Linux” and choose your distribution for more information on installing R for your setup.
2. **You must do this second:** Download and install RStudio at <https://www.rstudio.com/products/rstudio/download/>.
 - Scroll down to “Installers for Supported Platforms” near the bottom of the page.
 - Click on the download link corresponding to your computer’s operating system.

1.1.2 Using R via RStudio

Recall our car analogy from earlier. Much as we don’t drive a car by interacting directly with the engine but rather by interacting with elements on the car’s dashboard, we won’t be using R directly but rather we will use RStudio’s interface. After

you install R and RStudio on your computer, you'll have two new *programs* (also called *applications*) you can open. We'll always work in RStudio and not in the R application. Figure 1.2 shows what icon you should be clicking on your computer.

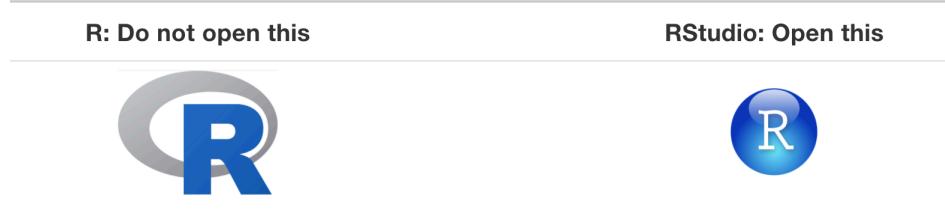


FIGURE 1.2: Icons of R versus RStudio on your computer.

After you open RStudio, you should see something similar to Figure 1.3. (Note that slight differences might exist if the RStudio interface is updated to not be this by default.)

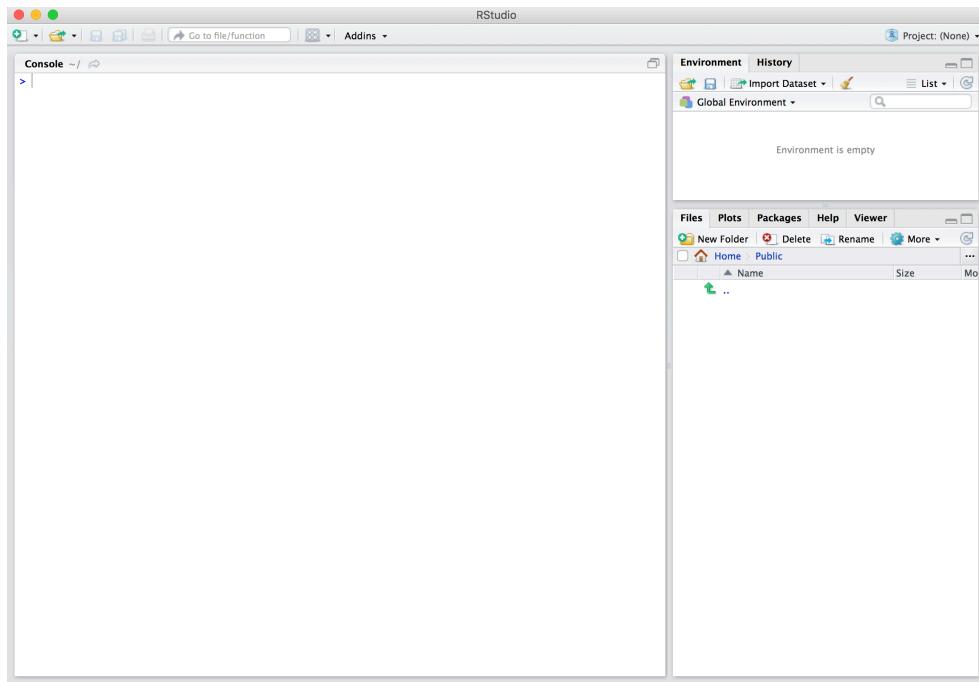


FIGURE 1.3: RStudio interface to R.

Note the three *panes* which are three panels dividing the screen: the *console pane*, the *files pane*, and the *environment pane*. Over the course of this chapter, you'll come to learn what purpose each of these panes serves.

1.2 How do I code in R?

Now that you're set up with R and RStudio, you are probably asking yourself, "OK. Now how do I use R?". The first thing to note is that unlike other statistical software programs like Excel, SPSS, or Minitab that provide point-and-click¹ interfaces, R is an interpreted language². This means you have to type in commands written in *R code*. In other words, you have to code/program in R. Note that we'll use the terms "coding" and "programming" interchangeably in this book.

While it is not required to be a seasoned coder/computer programmer to use R, there is still a set of basic programming concepts that new R users need to understand. Consequently, while this book is not a book on programming, you will still learn just enough of these basic programming concepts needed to explore and analyze data effectively.

1.2.1 Basic programming concepts and terminology

We now introduce some basic programming concepts and terminology. Instead of asking you to memorize all these concepts and terminology right now, we'll guide you so that you'll "learn by doing." To help you learn, we will always use a different font to distinguish regular text from `computer_code`. The best way to master these topics is, in our opinions, through deliberate practice³ with R and lots of repetition.

- Basics:
 - *Console pane*: where you enter in commands.
 - *Running code*: the act of telling R to perform an act by giving it commands in the console.
 - *Objects*: where values are saved in R. We'll show you how to *assign* values to objects and how to display the contents of objects.
 - *Data types*: integers, doubles/numerics, logicals, and characters. Integers are values like -1, 0, 2, 4092. Doubles or numerics are a larger set of values containing both the integers but also fractions and decimal values like -24.932 and 0.8. Logicals are either `TRUE` or `FALSE` while characters are text such as "cabbage", "Hamilton", "The Wire is the greatest TV show ever", and "This ramen is delicious." Note that characters are often denoted with the quotation marks around them.
- *Vectors*: a series of values. These are created using the `c()` function, where `c()` stands for "combine" or "concatenate." For example, `c(6, 11, 13, 31, 90, 92)` creates a six element series of positive integer values .

¹https://en.wikipedia.org/wiki/Point_and_click

²https://en.wikipedia.org/wiki/Interpreted_language

³<https://jamesclear.com/deliberate-practice-theory>

- *Factors*: *categorical data* are commonly represented in R as factors. Categorical data can also be represented as *strings*. We'll study this difference as we progress through the book.
- *Data frames*: rectangular spreadsheets. They are representations of datasets in R where the rows correspond to *observations* and the columns correspond to *variables* that describe the observations. We'll cover data frames later in Section 1.4.
- *Conditionals*:
 - Testing for equality in R using `==` (and not `=`, which is typically used for assignment). For example, `2 + 1 == 3` compares `2 + 1` to `3` and is correct R code, while `2 + 1 = 3` will return an error.
 - Boolean algebra: `TRUE/FALSE` statements and mathematical operators such as `<` (less than), `<=` (less than or equal), and `!=` (not equal to). For example, `4 + 2 >= 3` will return `TRUE`, but `3 + 5 <= 1` will return `FALSE`.
 - Logical operators: `&` representing “and” as well as `|` representing “or.” For example, `(2 + 1 == 3) & (2 + 1 == 4)` returns `FALSE` since both clauses are not `TRUE` (only the first clause is `TRUE`). On the other hand, `(2 + 1 == 3) | (2 + 1 == 4)` returns `TRUE` since at least one of the two clauses is `TRUE`.
- *Functions*, also called *commands*: Functions perform tasks in R. They take in inputs called *arguments* and return outputs. You can either manually specify a function's arguments or use the function's *default values*.
 - For example, the function `seq()` in R generates a sequence of numbers. If you just run `seq()` it will return the value `1`. That doesn't seem very useful! This is because the default arguments are set as `seq(from = 1, to = 1)`. Thus, if you don't pass in different values for `from` and `to` to change this behavior, R just assumes all you want is the number `1`. You can change the argument values by updating the values after the `=` sign. If we try out `seq(from = 2, to = 5)` we get the result `2 3 4 5` that we might expect.
 - We'll work with functions a lot throughout this book and you'll get lots of practice in understanding their behaviors. To further assist you in understanding when a function is mentioned in the book, we'll also include the `()` after them as we did with `seq()` above.

This list is by no means an exhaustive list of all the programming concepts and terminology needed to become a savvy R user; such a list would be so large it wouldn't be very useful, especially for novices. Rather, we feel this is a minimally viable list of programming concepts and terminology you need to know before getting started. We feel that you can learn the rest as you go. Remember that your mastery of all of these concepts and terminology will build as you practice more and more.

1.2.2 Errors, warnings, and messages

One thing that intimidates new R and RStudio users is how it reports *errors*, *warnings*, and *messages*. R reports errors, warnings, and messages in a glaring red font,

which makes it seem like it is scolding you. However, seeing red text in the console is not always bad.

R will show red text in the console pane in three different situations:

- **Errors:** When the red text is a legitimate error, it will be prefaced with “Error in...” and will try to explain what went wrong. Generally when there’s an error, the code will not run. For example, we’ll see in Subsection 1.3.3 if you see `Error in ggplot(...)` : could not find function “`ggplot`”, it means that the `ggplot()` function is not accessible because the package that contains the function (`ggplot2`) was not loaded with `library(ggplot2)`. Thus you cannot use the `ggplot()` function without the `ggplot2` package being loaded first.
- **Warnings:** When the red text is a warning, it will be prefaced with “Warning:” and R will try to explain why there’s a warning. Generally your code will still work, but with some caveats. For example, you will see in Chapter 2 if you create a scatterplot based on a dataset where two of the rows of data have missing entries that would be needed to create points in the scatterplot, you will see this warning: `Warning: Removed 2 rows containing missing values (geom_point)`. R will still produce the scatterplot with all the remaining non-missing values, but it is warning you that two of the points aren’t there.
- **Messages:** When the red text doesn’t start with either “Error” or “Warning”, it’s *just a friendly message*. You’ll see these messages when you load *R packages* in the upcoming Subsection 1.3.2 or when you read data saved in spreadsheet files with the `read_csv()` function as you’ll see in Chapter 4. These are helpful diagnostic messages and they don’t stop your code from working. Additionally, you’ll see these messages when you install packages too using `install.packages()` as discussed in Subsection 1.3.1.

Remember, when you see red text in the console, *don’t panic*. It doesn’t necessarily mean anything is wrong. Rather:

- If the text starts with “Error”, figure out what’s causing it. Think of errors as a red traffic light: something is wrong!
- If the text starts with “Warning”, figure out if it’s something to worry about. For instance, if you get a warning about missing values in a scatterplot and you know there are missing values, you’re fine. If that’s surprising, look at your data and see what’s missing. Think of warnings as a yellow traffic light: everything is working fine, but watch out/pay attention.
- Otherwise, the text is just a message. Read it, wave back at R, and thank it for talking to you. Think of messages as a green traffic light: everything is working fine and keep on going!

1.2.3 Tips on learning to code

Learning to code/program is quite similar to learning a foreign language. It can be daunting and frustrating at first. Such frustrations are common and it is normal to feel discouraged as you learn. However, just as with learning a foreign language, if you put in the effort and are not afraid to make mistakes, anybody can learn and improve.

Here are a few useful tips to keep in mind as you learn to program:

- **Remember that computers are not actually that smart:** You may think your computer or smartphone is “smart,” but really people spent a lot of time and energy designing them to appear “smart.” In reality, you have to tell a computer everything it needs to do. Furthermore, the instructions you give your computer can’t have any mistakes in them, nor can they be ambiguous in any way.
- **Take the “copy, paste, and tweak” approach:** Especially when you learn your first programming language or you need to understand particularly complicated code, it is often much easier to take existing code that you know works and modify it to suit your ends. This is as opposed to trying to type out the code from scratch. We call this the “*copy, paste, and tweak*” approach. So early on, we suggest not trying to write code from memory, but rather take existing examples we have provided you, then copy, paste, and tweak them to suit your goals. After you start feeling more confident, you can slowly move away from this approach and write code from scratch. Think of the “copy, paste, and tweak” approach as training wheels for a child learning to ride a bike. After getting comfortable, they won’t need them anymore.
- **The best way to learn to code is by doing:** Rather than learning to code for its own sake, we find that learning to code goes much smoother when you have a goal in mind or when you are working on a particular project, like analyzing data that you are interested in and that is important to you.
- **Practice is key:** Just as the only method to improve your foreign language skills is through lots of practice and speaking, the only method to improving your coding skills is through lots of practice. Don’t worry, however, we’ll give you plenty of opportunities to do so!

1.3 What are R packages?

Another point of confusion with many new R users is the idea of an R package. R packages extend the functionality of R by providing additional functions, data, and documentation. They are written by a worldwide community of R users and can be downloaded for free from the internet.

For example, among the many packages we will use in this book are the `ggplot2` package (Wickham et al., 2024a) for data visualization in Chapter 2, the `dplyr` package (Wickham et al., 2023) for data wrangling in Chapter 3, the `moderndive` package (Kim and Ismay, 2024) that accompanies this book, and the `infer` package (Bray et al., 2024) for “tidy” and transparent statistical inference in Chapters 8, 9, and 10.

A good analogy for R packages is they are like apps you can download onto a mobile phone:

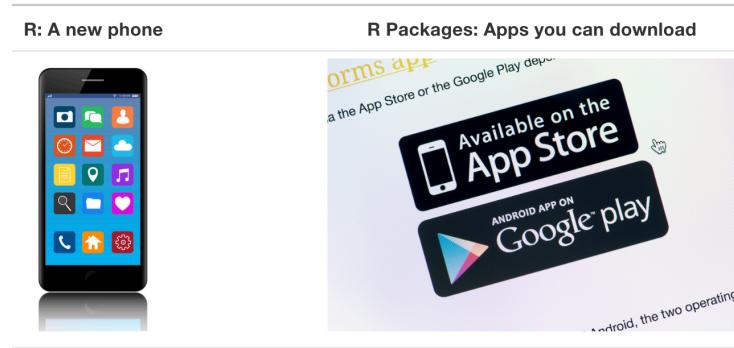


FIGURE 1.4: Analogy of R versus R packages.

So R is like a new mobile phone: while it has a certain amount of features when you use it for the first time, it doesn’t have everything. R packages are like the apps you can download onto your phone from Apple’s App Store or Android’s Google Play.

Let’s continue this analogy by considering the Instagram app for editing and sharing pictures. Say you have purchased a new phone and you would like to share a photo you have just taken with friends on Instagram. You need to:

1. *Install the app:* Since your phone is new and does not include the Instagram app, you need to download the app from either the App Store or Google Play. You do this once and you’re set for the time being. You might need to do this again in the future when there is an update to the app.
2. *Open the app:* After you’ve installed Instagram, you need to open it.

Once Instagram is open on your phone, you can then proceed to share your photo with your friends and family. The process is very similar for using an R package. You need to:

1. *Install the package:* This is like installing an app on your phone. Most packages are not installed by default when you install R and RStudio. Thus if you want to use a package for the first time, you need to install it first. Once you’ve installed a package, you likely won’t install it again unless you want to update it to a newer version.

2. “Load” the package: “Loading” a package is like opening an app on your phone. Packages are not “loaded” by default when you start RStudio on your computer; you need to “load” each package you want to use every time you start RStudio.

Let’s perform these two steps for the `ggplot2` package for data visualization.

1.3.1 Package installation

Note about RStudio Server or RStudio Cloud: If your instructor has provided you with a link and access to RStudio Server or RStudio Cloud, you might not need to install packages, as they might be preinstalled for you by your instructor. That being said, it is still a good idea to know this process for later on when you are not using RStudio Server or Cloud, but rather RStudio Desktop on your own computer.

There are two ways to install an R package: an easy way and a more advanced way. Let’s install the `ggplot2` package the easy way first as shown in Figure 1.5. In the Files pane of RStudio:

- a) Click on the “Packages” tab.
- b) Click on “Install” next to Update.
- c) Type the name of the package under “Packages (separate multiple with space or comma):” In this case, type `ggplot2`.
- d) Click “Install.”

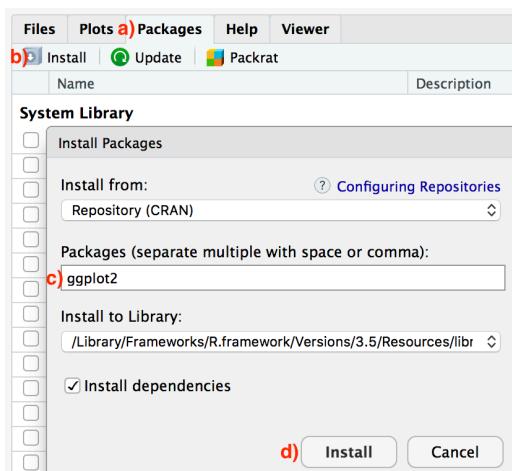


FIGURE 1.5: Installing packages in R the easy way.

An alternative but slightly less convenient way to install a package is by typing `install.packages("ggplot2")` in the console pane of RStudio and pressing Return/Enter on your keyboard. Note you must include the quotation marks around the name of the package.

Much like an app on your phone, you only have to install a package once. However, if you want to update a previously installed package to a newer version, you need to reinstall it by repeating the earlier steps.

Learning check

(LC1.1) Repeat the earlier installation steps, but for the `dplyr`, `nycflights23`, and `knitr` packages. This will install the earlier mentioned `dplyr` package for data wrangling, the `nycflights23` package containing data on all domestic flights leaving a NYC airport in 2023, and the `knitr` package for generating easy-to-read tables in R. We'll use these packages in the next section.

Note that if you'd like your output on your computer to match up exactly with the output presented throughout the book, you may want to use the exact versions of the packages that we used. You can find a full listing of these packages and their versions in Appendix B. This likely won't be relevant for novices, but we included it for reproducibility reasons.

1.3.2 Package loading

Recall that after you've installed a package, you need to "load it." In other words, you need to "open it." We do this by using the `library()` command.

For example, to load the `ggplot2` package, run the following code in the console pane. What do we mean by "run the following code"? Either type or copy-and-paste the following code into the console pane and then hit the Enter key.

```
library(ggplot2)
```

If after running the earlier code, a blinking cursor returns next to the > "prompt" sign, it means you were successful and the `ggplot2` package is now loaded and ready to use. If, however, you get a red "error message" that reads ...

```
Error in library(ggplot2) : there is no package called 'ggplot2'
```

... it means that you didn't successfully install it. This is an example of an “error message” we discussed in Subsection 1.2.2. If you get this error message, go back to Subsection 1.3.1 on R package installation and make sure to install the `ggplot2` package before proceeding.

Learning check

(LC1.2) “Load” the `dplyr`, `nycflights23`, and `knitr` packages as well by repeating the earlier steps.

1.3.3 Package use

One very common mistake new R users make when wanting to use particular packages is they forget to “load” them first by using the `library()` command we just saw. Remember: *you have to load each package you want to use every time you start RStudio*. If you don’t first “load” a package, but attempt to use one of its features, you’ll see an error message similar to:

```
Error: could not find function
```

This is a different error message than the one you just saw on a package not having been installed yet. R is telling you that you are trying to use a function in a package that has not yet been “loaded.” R doesn’t know where to find the function you are using. Almost all new users forget to do this when starting out, and it is a little annoying to get used to doing it. However, you’ll remember with practice and after some time it will become second nature for you.

1.4 Explore your first datasets

Let’s put everything we’ve learned so far into practice and start exploring some real data! Data comes to us in a variety of formats, from pictures to text to numbers. Throughout this book, we’ll focus on datasets that are saved in “spreadsheet”-type format. This is probably the most common way data are collected and saved in many fields. Remember from Subsection 1.2.1 that these “spreadsheet”-type datasets are

called *data frames* in R. We'll focus on working with data saved as data frames throughout this book.

Let's first load all the packages needed for this chapter, assuming you've already installed them. Read Section 1.3 for information on how to install and load R packages if you haven't already.

```
library(nycflights23)
library(dplyr)
library(knitr)
```

At the beginning of all subsequent chapters in this book, we'll always have a list of packages that you should have installed and loaded in order to work with that chapter's R code.

1.4.1 nycflights23 package

Many of us have flown on airplanes or know someone who has. Air travel has become an ever-present aspect of many people's lives. If you look at the Departures flight information board at an airport, you will frequently see that some flights are delayed for a variety of reasons. Are there ways that we can understand the reasons that cause flight delays?

We'd all like to arrive at our destinations on time whenever possible. (Unless you secretly love hanging out at airports. If you are one of these people, pretend for a moment that you are very much anticipating being at your final destination.) Throughout this book, we're going to analyze data related to all domestic flights departing from one of New York City's three main airports in 2023: Newark Liberty International (EWR), John F. Kennedy International (JFK), and LaGuardia Airport (LGA). We'll access this data using the `nycflights23` R package, which contains five datasets saved in five data frames:

- `flights`: Information on all flights.
- `airlines`: A table matching airline names and their two-letter International Air Transport Association (IATA) airline codes (also known as carrier codes) for 14 airline companies. For example, "DL" is the two-letter code for Delta.
- `planes`: Information about each of the 4,840 physical aircraft used.
- `weather`: Hourly meteorological data for each of the three NYC airports. This data frame has 26,204 rows, roughly corresponding to the $365 \times 24 \times 3 = 26,280$ possible hourly measurements one can observe at three locations over the course of a year.
- `airports`: Names, codes, and locations of the 1,251 domestic destinations.

The `nycflights23` package is an updated version of the classic `nycflights13` R package⁴. `nycflights23` was authored by ModernDive co-author Chester Ismay using the

⁴<https://nycflights13.tidyverse.org/>

anyflights R package⁵ developed by Simon Couch. Simon granted permission to the ModernDive team to use the data in the `anyflights` package to create `nycflights23` and submit the package to CRAN.

1.4.2 flights data frame

We'll begin by exploring the `flights` data frame and get an idea of its structure. Run the following code in your console, either by typing it or by cutting-and-pasting it. It displays the contents of the `flights` data frame in your console. Note that depending on the size of your monitor, the output may vary slightly.

```
flights
```

```
# A tibble: 435,352 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>     <int>          <int>     <dbl>    <int>
1 2023     1     1       1            2038      203     328
2 2023     1     1      18            2300       78     228
3 2023     1     1      31            2344       47     500
4 2023     1     1      33            2140      173     238
5 2023     1     1      36            2048      228     223
6 2023     1     1     503            500        3     808
7 2023     1     1     520            510       10     948
8 2023     1     1     524            530       -6     645
9 2023     1     1     537            520       17     926
10 2023    1     1     547            545        2     845
# i 435,342 more rows
# i 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#   time_hour <dttm>
```

Let's unpack this output:

- A `tibble`: 435,352 x 19: A `tibble` is a specific kind of data frame in R. This particular data frame has
 - 435,352 rows corresponding to different *observations*. Here, each observation is a flight.
 - 19 columns corresponding to 19 *variables* describing each observation.

⁵<https://anyflights.netlify.app/>

- `year`, `month`, `day`, `dep_time`, `sched_dep_time`, `dep_delay`, and `arr_time` are the different columns, in other words, the different variables of this dataset.
- We then have a preview of the first 10 rows of observations corresponding to the first 10 flights. R is only showing the first 10 rows, because if it showed all 435,352 rows, it would overwhelm your screen.
- ... with 435,342 more rows` and 11 more variables: indicating to us that 435,342 more rows of data and 11 more variables could not fit in this screen.

Unfortunately, this output does not allow us to explore the data very well, but it does give a nice preview. Let's look at some different ways to explore data frames.

1.4.3 Exploring data frames

There are many ways to get a feel for the data contained in a data frame such as `flights`. We present three functions that take as their “argument” (their input) the data frame in question. We also include a fourth method for exploring one particular column of a data frame:

1. Using the `View()` function, which brings up RStudio’s built-in data viewer.
2. Using the `glimpse()` function, which is included in the `dplyr` package.
3. Using the `kable()` function, which is included in the `knitr` package.
4. Using the `$` “extraction operator,” which is used to view a single variable/column in a data frame.

1. `View()`:

Run `View(flights)` in your console in RStudio, either by typing it or cutting-and-pasting it into the console pane. Explore this data frame in the resulting pop up viewer. You should get into the habit of viewing any data frames you encounter. Note the uppercase `V` in `View()`. R is case-sensitive, so you’ll get an error message if you run `view(flights)` instead of `View(flights)`.

Learning check

(LC1.3) What does any *ONE* row in this `flights` dataset refer to?

- A. Data on an airline
- B. Data on a flight
- C. Data on an airport
- D. Data on multiple flights

By running `View(flights)`, we can explore the different *variables* listed in the columns. Observe that there are many different types of variables. Some of the variables like `distance`, `day`, and `arr_delay` are what we will call *quantitative* variables. These variables are numerical in nature. Other variables here are *categorical*.

Note that if you look in the leftmost column of the `View(flights)` output, you will see a column of numbers. These are the row numbers of the dataset. If you glance across a row with the same number, say row 5, you can get an idea of what each row is representing. This will allow you to identify what object is being described in a given row by taking note of the values of the columns in that specific row. This is often called the *observational unit*. The observational unit in this example is an individual flight departing from New York City in 2023. You can identify the observational unit by determining what “thing” is being measured or described by each of the variables. We’ll talk more about observational units in Subsection 1.4.4 on *identification* and *measurement* variables.

2. `glimpse()`:

The second way we’ll cover to explore a data frame is using the `glimpse()` function included in the `dplyr` package. Thus, you can only use the `glimpse()` function after you’ve loaded the `dplyr` package by running `library(dplyr)`. This function provides us with an alternative perspective for exploring a data frame than the `View()` function:

```
glimpse(flights)
```

```
Rows: 435,352
Columns: 19
$ year      <int> 2023, 2023, 2023, 2023, 2023, 2023, 2023, 2023, 20~
$ month     <int> 1, 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, 1, ~
$ dep_time   <int> 1, 18, 31, 33, 36, 503, 520, 524, 537, 547, 549, 5~
$ sched_dep_time <int> 2038, 2300, 2344, 2140, 2048, 500, 510, 530, 520, ~
$ dep_delay   <dbl> 203, 78, 47, 173, 228, 3, 10, -6, 17, 2, -10, -9, ~
$ arr_time    <int> 328, 228, 500, 238, 223, 808, 948, 645, 926, 845, ~
$ sched_arr_time <int> 3, 135, 426, 2352, 2252, 815, 949, 710, 818, 852, ~
$ arr_delay   <dbl> 205, 53, 34, 166, 211, -7, -1, -25, 68, -7, 4, -13~
$ carrier     <chr> "UA", "DL", "B6", "B6", "UA", "AA", "B6", "AA", "U~
$ flight      <int> 628, 393, 371, 1053, 219, 499, 996, 981, 206, 225, ~
$ tailnum     <chr> "N25201", "N830DN", "N807JB", "N265JB", "N17730", ~
$ origin      <chr> "EWR", "JFK", "JFK", "JFK", "EWR", "EWR", "JFK", "J~
$ dest        <chr> "SMF", "ATL", "BQN", "CHS", "DTW", "MIA", "BQN", "B~
```

```
$ air_time      <dbl> 367, 108, 190, 108, 80, 154, 192, 119, 258, 157, 1~  
$ distance     <dbl> 2500, 760, 1576, 636, 488, 1085, 1576, 719, 1400, ~  
$ hour         <dbl> 20, 23, 23, 21, 20, 5, 5, 5, 5, 5, 6, 5, 6, 6, ~  
$ minute        <dbl> 38, 0, 44, 40, 48, 0, 10, 30, 20, 45, 59, 0, 59, 0~  
$ time_hour    <dttm> 2023-01-01 20:00:00, 2023-01-01 23:00:00, 2023-01~
```

Observe that `glimpse()` will give you the first few entries of each variable in a row after the variable name. In addition, the *data type* (see Subsection 1.2.1) of the variable is given immediately after each variable’s name inside `< >`. Here, `int` and `dbl` refer to “integer” and “double”, which are computer coding terminology for quantitative/numerical variables. “Doubles” take up twice the size to store on a computer compared to integers.

In contrast, `chr` refers to “character”, which is computer terminology for text data. In most forms, text data, such as the `carrier` or `origin` of a flight, are categorical variables. The `time_hour` variable is another data type: `dttm`. These types of variables represent date and time combinations. However, we won’t work with dates and times in this book; we leave this topic for other data science books like *Data Science: A First Introduction* by Tiffany-Anne Timbers, Melissa Lee, and Trevor Campbell⁶ or *R for Data Science*⁷ (Gromlund and Wickham, 2017).

Learning check

(LC1.4) What are some other examples in this dataset of *categorical* variables? What makes them different than *quantitative* variables?

3. `kable()`:

The final way to explore the entirety of a data frame is using the `kable()` function from the `knitr` package. Let’s explore the different carrier codes for all the airlines in our dataset two ways. Run both of these lines of code in the console:

```
airlines  
kable(airlines)
```

⁶<https://datasciencebook.ca/>

⁷<https://r4ds.had.co.nz/dates-and-times.html>

At first glance, it may not appear that there is much difference in the outputs. However, when using tools for producing reproducible reports such as R Markdown⁸, the latter code produces output that is much more legible and reader-friendly. You'll see us use this reader-friendly style in many places in the book when we want to print a data frame as a nice table.

4. \$ operator

Lastly, the \$ operator allows us to extract and then explore a single variable within a data frame. For example, run the following in your console:

```
airlines$name
```

We used the \$ operator to extract only the `name` variable and return it as a vector of length 16. We'll only be occasionally exploring data frames using the \$ operator, instead favoring the `View()` and `glimpse()` functions.

1.4.4 Identification and measurement variables

There is a subtle difference between the kinds of variables that you will encounter in data frames. There are *identification variables* and *measurement variables*. For example, let's explore the `airports` data frame by showing the output of `glimpse(airports)`:

```
glimpse(airports)
```

```
Rows: 1,251
Columns: 8
$ faa    <chr> "AAF", "AAP", "ABE", "ABI", "ABL", "ABQ", "ABR", "ABY", "AC~
$ name   <chr> "Apalachicola Regional Airport", "Andrau Airpark", "Lehigh ~
$ lat    <dbl> 29.7, 29.7, 40.7, 32.4, 67.1, 35.0, 45.4, 31.5, 41.3, 31.6, ~
$ lon    <dbl> -85.0, -95.6, -75.4, -99.7, -157.9, -106.6, -98.4, -84.2, --~
$ alt    <dbl> 20, 79, 393, 1791, 334, 5355, 1302, 197, 47, 516, 221, 75, ~
$ tz     <dbl> -5, -6, -5, -6, -9, -7, -6, -5, -5, -6, -8, -5, -10, -6, -9~
$ dst    <chr> "A", ~
$ tzone  <chr> "America/New_York", "America/Chicago", "America/New_York", ~
```

The variables `faa` and `name` are what we will call *identification variables*, variables that uniquely identify each observational unit. In this case, the identification variables uniquely identify airports. Such variables are mainly used in practice to uniquely identify each row in a data frame. `faa` gives the unique code provided by the FAA

⁸<http://rmarkdown.rstudio.com/lesson-1.html>

for that airport, while the `name` variable gives the longer official name of the airport. The remaining variables (`lat`, `lon`, `alt`, `tz`, `dst`, `tzone`) are often called *measurement* or *characteristic* variables: variables that describe properties of each observational unit. For example, `lat` and `long` describe the latitude and longitude of each airport.

Furthermore, sometimes a single variable might not be enough to uniquely identify each observational unit: combinations of variables might be needed. While it is not an absolute rule, for organizational purposes it is considered good practice to have your identification variables in the leftmost columns of your data frame.

Learning check

(LC1.5) What properties of each airport do the variables `lat`, `lon`, `alt`, `tz`, `dst`, and `tzone` describe in the `airports` data frame? Take your best guess.

(LC1.6) Provide the names of variables in a data frame with at least three variables where one of them is an identification variable and the other two are not. Further, create your own tidy data frame that matches these conditions.

1.4.5 Help files

Another nice feature of R are help files, which provide documentation for various functions and datasets. You can bring up help files by adding a `?` before the name of a function or data frame and then run this in the console. You will then be presented with a page showing the corresponding documentation if it exists. For example, let's look at the help file for the `flights` data frame.

```
?flights
```

The help file should pop up in the Help pane of RStudio. If you have questions about a function or data frame included in an R package, you should get in the habit of consulting the help file right away.

Learning check

(LC1.7) Look at the help file for the `airports` data frame. Revise your earlier guesses about what the variables `lat`, `lon`, `alt`, `tz`, `dst`, and `tzone` each describe.

1.5 Conclusion

We've given you what we feel is a minimally viable set of tools to explore data in R. Does this chapter contain everything you need to know? Absolutely not. To try to include everything in this chapter would make the chapter so large it wouldn't be useful! As we said earlier, the best way to add to your toolbox is to get into RStudio and run and write code as much as possible.

1.5.1 Additional resources

Solutions to all *Learning checks* can be found online in Appendix D⁹.

If you are new to the world of coding, R, and RStudio and feel you could benefit from a more detailed introduction, we suggest you check out the short book, *Getting Used to R, RStudio, and R Markdown*¹⁰ (Ismay and Kennedy, 2016). It includes screencast recordings that you can follow along and pause as you learn. This book also contains an introduction to R Markdown, a tool used for reproducible research in R.

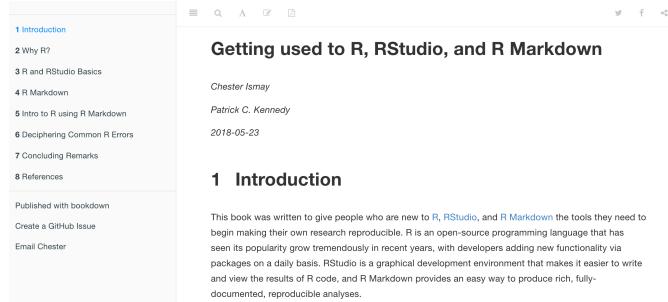


FIGURE 1.6: Preview of *Getting Used to R, RStudio, and R Markdown*.

1.5.2 What's to come?

We're now going to start the “Data Science with tidyverse” portion of this book in Chapter 2 as shown in Figure 1.7 with what we feel is the most important tool in a data scientist’s toolbox: data visualization. We’ll continue to explore the data included in the `moderndive` and `nycflights23` packages using the `ggplot2` package for

⁹<https://moderndive.com/D-appendixD.html>

¹⁰<https://rbasics.netlify.app/>

data visualization. You'll see that data visualization is a powerful tool to add to your toolbox for data exploration that provides additional insight to what the `View()` and `glimpse()` functions can provide.

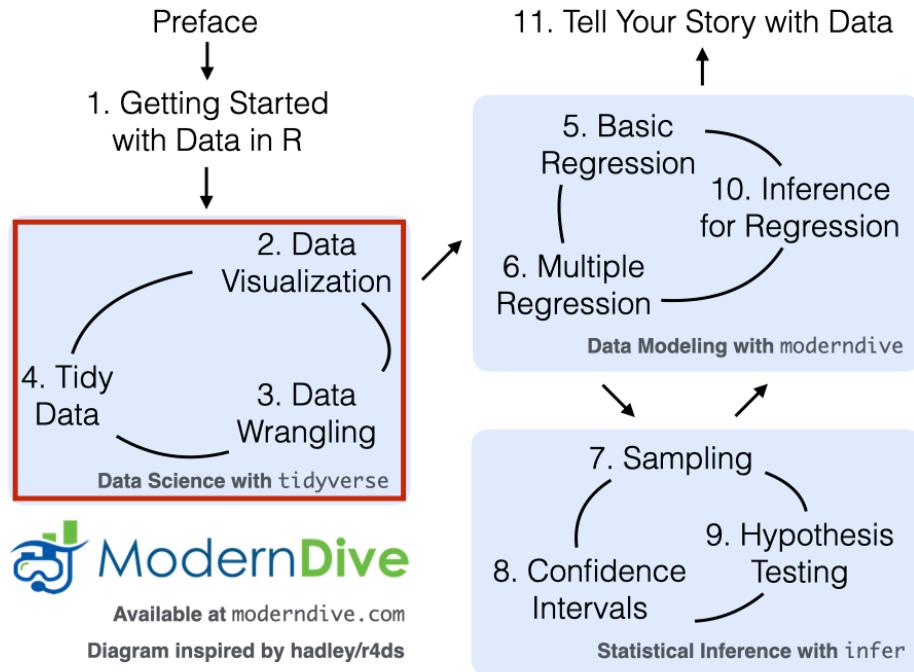


FIGURE 1.7: *ModernDive* flowchart - on to Part I!

Part I

Data Science with tidyverse

2

Data Visualization

We begin the development of your data science toolbox with data visualization. By visualizing data, we gain valuable insights we couldn't initially obtain from just looking at the raw data values. We'll use the `ggplot2` package, as it provides an easy way to customize your plots. `ggplot2` is rooted in the data visualization theory known as *the grammar of graphics* ([Wilkinson, 2005](#)), developed by Leland Wilkinson.

At their most basic, graphics/plots/charts (we use these terms interchangeably in this book) provide a nice way to explore the patterns in data, such as the presence of *outliers*, *distributions* of individual variables, and *relationships* between groups of variables. Graphics are designed to emphasize the findings and insights you want your audience to understand. This does, however, require a balancing act. On the one hand, you want to highlight as many interesting findings as possible. On the other hand, you don't want to include so much information that it overwhelms your audience.

As we will see, plots also help us to identify patterns and outliers in our data. We'll see that a common extension of these ideas is to compare the *distribution* of one numerical variable, such as what are the center and spread of the values, as we go across the levels of a different categorical variable.

Needed packages

Let's load all the packages needed for this chapter (this assumes you've already installed them). Read Section [1.3](#) for information on how to install and load R packages.

```
library(nycflights23)
library(ggplot2)
library(moderndive)
```

2.1 The grammar of graphics

We start with a discussion of a theoretical framework for data visualization known as “the grammar of graphics.” This framework serves as the foundation for the `ggplot2`

package which we'll use extensively in this chapter. Think of how we construct and form sentences in English by combining different elements, like nouns, verbs, articles, subjects, objects, etc. We can't just combine these elements in any arbitrary order; we must do so following a set of rules known as a linguistic grammar. Similarly to a linguistic grammar, "the grammar of graphics" defines a set of rules for constructing *statistical graphics* by combining different types of *layers*. This grammar was created by Leland Wilkinson ([Wilkinson, 2005](#)) and has been implemented in a variety of data visualization software platforms like R, but also Plotly¹ and Tableau².

2.1.1 Components of the grammar

In short, the grammar tells us that:

A statistical graphic is a mapping of data variables to aesthetic attributes of geometric objects.

Specifically, we can break a graphic into the following three essential components:

1. `data`: the dataset containing the variables of interest.
2. `geom`: the geometric object in question. This refers to the type of object we can observe in a plot. For example: points, lines, and bars.
3. `aes`: aesthetic attributes of the geometric object. For example, x/y position, color, shape, and size. Aesthetic attributes are *mapped* to variables in the dataset.

You might be wondering why we wrote the terms `data`, `geom`, and `aes` in a computer code type font. We'll see very shortly that we'll specify the elements of the grammar in R using these terms. However, let's first break down the grammar with an example.

2.1.2 Gapminder data

In February 2006, a Swedish physician and data advocate named Hans Rosling gave a TED talk titled "The best stats you've ever seen"³ where he presented global economic, health, and development data from the website [gapminder.org](#)⁴. For example, for data on 142 countries in 2007, let's consider only a few countries in Table 2.1 as a peek into the data.

¹<https://plot.ly/>

²<https://www.tableau.com/>

³https://www.ted.com/talks/hans_rosling_shows_the_best_stats_you_ve_ever_seen

⁴http://www.gapminder.org/tools/#_locale_id=en;&chart-type=bubbles

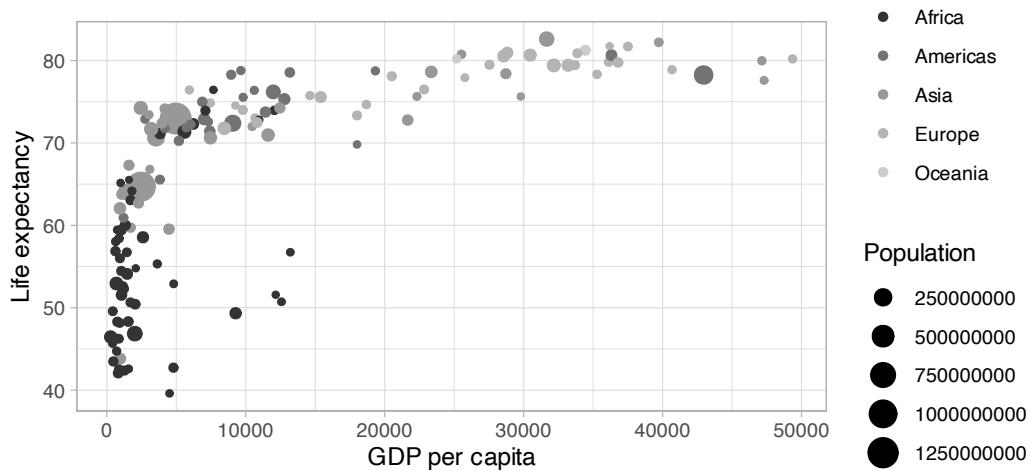
TABLE 2.1: Gapminder 2007 Data: First 3 of 142 countries

Country	Continent	Life Expectancy	Population	GDP per Capita
Afghanistan	Asia	43.8	31889923	975
Albania	Europe	76.4	3600523	5937
Algeria	Africa	72.3	33333216	6223

Each row in this table corresponds to a country in 2007. For each row, we have 5 columns:

1. **Country:** Name of country.
2. **Continent:** Which of the five continents the country is part of. Note that “Americas” includes countries in both North and South America and that Antarctica is excluded.
3. **Life Expectancy:** Life expectancy in years.
4. **Population:** Number of people living in the country.
5. **GDP per Capita:** Gross domestic product (in US dollars).

Now consider Figure 2.1, which plots this for all 142 of the data’s countries.

**FIGURE 2.1:** Life expectancy over GDP per capita in 2007.

Let’s view this plot through the grammar of graphics:

1. The data variable **GDP per Capita** gets mapped to the x-position aesthetic of the points.
2. The data variable **Life Expectancy** gets mapped to the y-position aesthetic of the points.
3. The data variable **Population** gets mapped to the size aesthetic of the points.
4. The data variable **Continent** gets mapped to the color aesthetic of the points.

We'll see shortly that `data` corresponds to the particular data frame where our data is saved and that "data variables" correspond to particular columns in the data frame. Furthermore, the type of geometric object considered in this plot are points. That being said, while in this example we are considering points, graphics are not limited to just points. We can also use lines, bars, and other geometric objects.

Let's summarize the three essential components of the grammar in Table 2.2.

TABLE 2.2: Summary of the grammar of graphics for this plot

data variable	aes	geom
GDP per Capita	x	point
Life Expectancy	y	point
Population	size	point
Continent	color	point

2.1.3 Other components

There are other components of the grammar of graphics we can control as well. As you start to delve deeper into the grammar of graphics, you'll start to encounter these topics more frequently. In this book, we'll keep things simple and only work with these two additional components:

- faceting breaks up a plot into several plots split by the values of another variable (Section 2.6)
- position adjustments for barplots (Section 2.8)

Other more complex components like scales and coordinate systems are left for a more advanced text such as *R for Data Science*⁵ (Grolmund and Wickham, 2017). Generally speaking, the grammar of graphics allows for a high degree of customization of plots and also a consistent framework for easily updating and modifying them.

2.1.4 ggplot2 package

In this book, we will use the `ggplot2` package for data visualization, which is an implementation of the grammar of graphics for R (Wickham et al., 2024a). As we noted earlier, a lot of the previous section was written in a computer code type font. This is because the various components of the grammar of graphics are specified in the `ggplot()` function included in the `ggplot2` package. For the purposes of this book, we'll always provide the `ggplot()` function with the following arguments (i.e., inputs) at a minimum:

⁵<http://r4ds.had.co.nz/data-visualisation.html#aesthetic-mappings>

- The data frame where the variables exist: the `data` argument.
- The mapping of the variables to aesthetic attributes: the `mapping` argument which specifies the aesthetic attributes involved.

After we've specified these components, we then add *layers* to the plot using the `+` sign. The most essential layer to add to a plot is the layer that specifies which type of geometric object we want the plot to involve: points, lines, bars, and others. Other layers we can add to a plot include the plot title, axes labels, visual themes for the plots, and facets (which we'll see in Section 2.6).

Let's now put the theory of the grammar of graphics into practice.

2.2 Five named graphs - the 5NG

In order to keep things simple in this book, we will only focus on five different types of graphics, each with a commonly given name. We term these “five named graphs” or in abbreviated form, the **5NG**:

1. scatterplots
2. linegraphs
3. histograms
4. boxplots
5. barplots

We'll also present some variations of these plots, but with this basic repertoire of five graphics in your toolbox, you can visualize a wide array of different variable types. Note that certain plots are only appropriate for categorical variables, while others are only appropriate for numerical variables.

2.3 5NG#1: Scatterplots

The simplest of the 5NG are *scatterplots*, also called *bivariate plots*. They allow you to visualize the *relationship* between two numerical variables. While you may already be familiar with scatterplots, let's view them through the lens of the grammar of graphics we presented in Section 2.1. Specifically, we will visualize the relationship between the following two numerical variables in the `envoy_flights` data frame included in the `moderndive` package:

1. `dep_delay`: departure delay on the horizontal “x” axis and

2. `arr_delay`: arrival delay on the vertical “y” axis

for Envoy Airlines flights leaving NYC in 2023. In other words, `envoy_flights` does not consist of *all* flights that left NYC in 2023, but rather only those flights where `carrier` is `MQ` (which is Envoy Airlines’ carrier code).

Learning check

(LC2.1) Take a look at both the `flights` data frame from the `nycflights23` package and the `envoy_flights` data frame from the `moderndive` package by running `View(flights)` and `View(envoy_flights)`. In what respect do these data frames differ? For example, think about the number of rows in each dataset.

2.3.1 Scatterplots via `geom_point`

Let’s now go over the code that will create the desired scatterplot, while keeping in mind the grammar of graphics framework we introduced in Section 2.1. Let’s take a look at the code and break it down piece-by-piece.

Note: The printed version of this book uses `theme_light()` instead of the default `theme_grey()` for the plots created with `ggplot2` throughout the book. Bars and points are also converted to greyscale using `scale_color_grey()` and `scale_fill_grey()`. This helps with readability of the plots in the printed copy. As you follow along and run the code yourself, your plots will have a grey background instead of the white background in the printed book. Also, your plots will have colors beyond the greyscale versions provided in this printing.

```
ggplot(data = envoy_flights, mapping = aes(x = dep_delay, y = arr_delay)) +  
  geom_point()
```

Within the `ggplot()` function, we specify two of the components of the grammar of graphics as arguments (i.e., inputs):

1. The data as the `envoy_flights` data frame via `data = envoy_flights`.
2. The aesthetic `mapping` by setting `mapping = aes(x = dep_delay, y = arr_delay)`. Specifically, the variable `dep_delay` maps to the `x` position aesthetic, while the variable `arr_delay` maps to the `y` position.

We then add a layer to the `ggplot()` function call using the `+` sign. The added layer in question specifies the third component of the grammar: the geometric object. In this case, the geometric object is set to be points by specifying `geom_point()`. After running these two lines of code in your console, you'll notice two outputs: a warning message and the graphic shown in Figure 2.2.

```
Warning: Removed 3 rows containing missing values or values outside the scale range
(`geom_point()`).
```

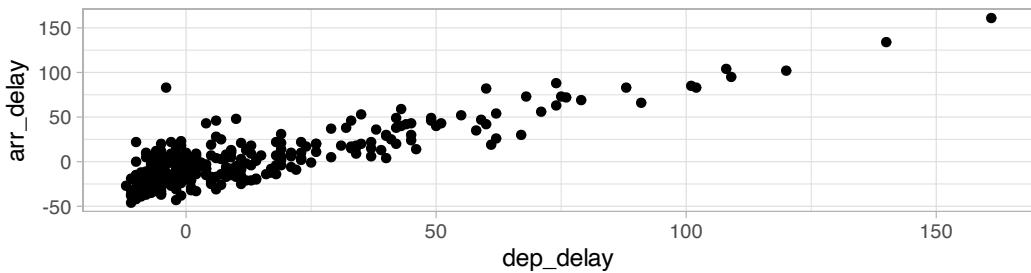


FIGURE 2.2: Arrival delays versus departure delays for Envoy Air flights from NYC in 2023.

Let's first unpack the graphic in Figure 2.2. Observe that a *positive relationship* exists between `dep_delay` and `arr_delay`: as departure delays increase, arrival delays tend to also increase. Observe also the large mass of points clustered near $(0, 0)$, the point indicating flights that neither departed nor arrived late.

Let's turn our attention to the warning message. R is alerting us to the fact that three rows were ignored due to them being missing. For these three rows, either the value for `dep_delay` or `arr_delay` or both were missing (recorded in R as `NA`), and thus these rows were ignored in our plot.

Before we continue, let's make a few more observations about this code that created the scatterplot. Note that the `+` sign comes at the end of lines, and not at the beginning. You'll get an error in R if you put it at the beginning of a line. When adding layers to a plot, you are encouraged to start a new line after the `+` (by pressing the Return/Enter button on your keyboard) so that the code for each layer is on a new line. As we add more and more layers to plots, you'll see this will greatly improve the legibility of your code.

To stress the importance of adding the layer specifying the geometric object, consider Figure 2.3 where no layers are added. Because the geometric object was not specified, we have a blank plot which is not very useful!

```
ggplot(data = envoy_flights, mapping = aes(x = dep_delay, y = arr_delay))
```

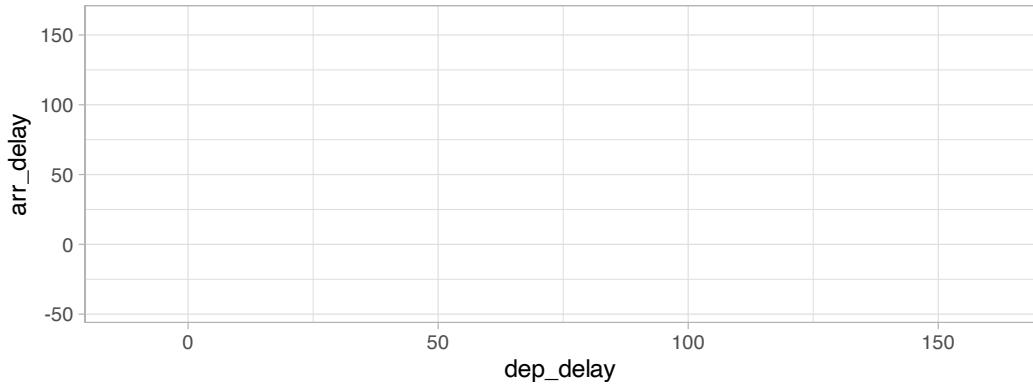


FIGURE 2.3: A plot with no layers.

Learning check

(LC2.2) What are some practical reasons why `dep_delay` and `arr_delay` have a positive relationship?

(LC2.3) What variables in the `weather` data frame would you expect to have a negative correlation (i.e., a negative relationship) with `dep_delay`? Why? Remember that we are focusing on numerical variables here. Hint: Explore the `weather` dataset by using the `View()` function.

(LC2.4) Why do you believe there is a cluster of points near (0, 0)? What does (0, 0) correspond to in terms of the Envoy Air flights?

(LC2.5) What are some other features of the plot that stand out to you?

(LC2.6) Create a new scatterplot using different variables in the `envoy_flights` data frame by modifying the example given.

2.3.2 Overplotting

The large mass of points near (0, 0) in Figure 2.2 can cause some confusion since it is hard to tell the true number of points that are plotted. This is the result of a phenomenon called *overplotting*. As one may guess, this corresponds to points being plotted on top of each other over and over again. When overplotting occurs, it is

difficult to know the number of points being plotted. There are two methods to address the issue of overplotting. Either by

1. Adjusting the transparency of the points or
2. Adding a little random “jitter”, or random “nudges”, to each of the points.

Method 1: Changing the transparency

The first way of addressing overplotting is to change the transparency-opacity of the points by setting the `alpha` argument in `geom_point()`. We can change the `alpha` argument to be any value between `0` and `1`, where `0` sets the points to be 100% transparent and `1` sets the points to be 100% opaque. By default, `alpha` is set to `1`. In other words, if we don’t explicitly set an `alpha` value, R will use `alpha = 1`.

Note how the following code is identical to the code in Section 2.3 that created the scatterplot with overplotting, but with `alpha = 0.2` added to the `geom_point()` function:

```
ggplot(data = envoy_flights, mapping = aes(x = dep_delay, y = arr_delay)) +
  geom_point(alpha = 0.2)
```

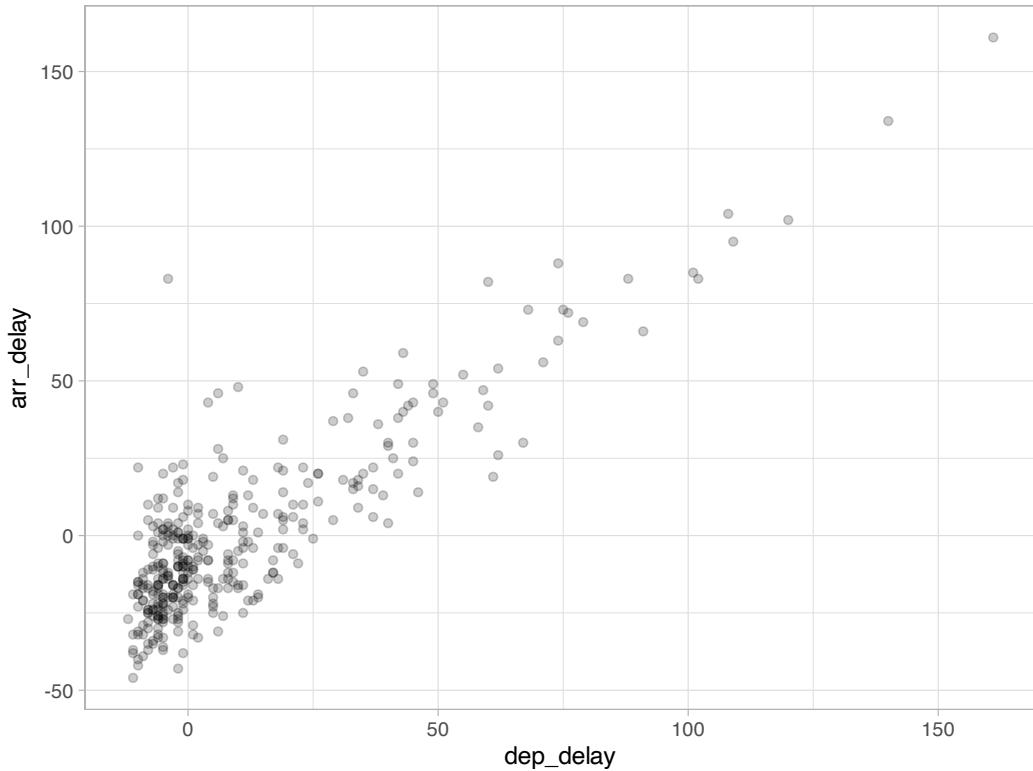


FIGURE 2.4: Arrival vs. departure delays scatterplot with $\text{alpha} = 0.2$.

The key feature to note in Figure 2.4 is that the transparency of the points is cumulative: areas with a high-degree of overplotting are darker, whereas areas with a lower degree are less dark. Note furthermore that there is no `aes()` surrounding `alpha = 0.2`. This is because we are not mapping a variable to an aesthetic attribute, but rather merely changing the default setting of `alpha`. In fact, you'll receive an error if you try to change the second line to read `geom_point(aes(alpha = 0.2))`.

Method 2: Jittering the points

The second way of addressing overplotting is by *jittering* all the points. This means giving each point a small “nudge” in a random direction. You can think of “jittering” as shaking the points around a bit on the plot. Let's illustrate using a simple example first. Say we have a data frame with 4 identical rows of x and y values: (0,0), (0,0), (0,0), and (0,0). In Figure 2.5, we present both the regular scatterplot of these 4 points (on the left) and its jittered counterpart (on the right).

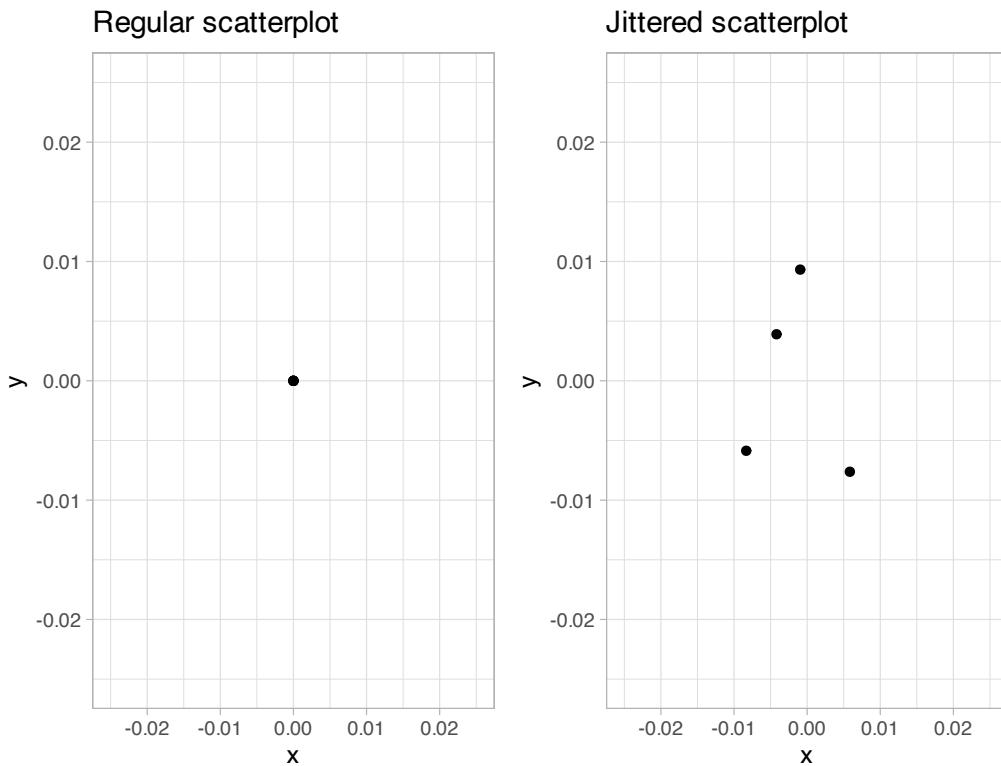


FIGURE 2.5: Regular and jittered scatterplot.

In the left-hand regular scatterplot, observe that the 4 points are superimposed on top of each other. While we know there are 4 values being plotted, this fact might not be apparent to others. In the right-hand jittered scatterplot, it is now plainly evident that this plot involves four points since each point is given a random “nudge.”

Keep in mind, however, that jittering is strictly a visualization tool; even after creating a jittered scatterplot, the original values saved in the data frame remain unchanged.

To create a jittered scatterplot, instead of using `geom_point()`, we use `geom_jitter()`. Observe how the following code is very similar to the code that created the scatterplot with overplotting in Subsection 2.3.1, but with `geom_point()` replaced with `geom_jitter()`.

```
ggplot(data = envoy_flights, mapping = aes(x = dep_delay, y = arr_delay)) +
  geom_jitter(width = 30, height = 30)
```

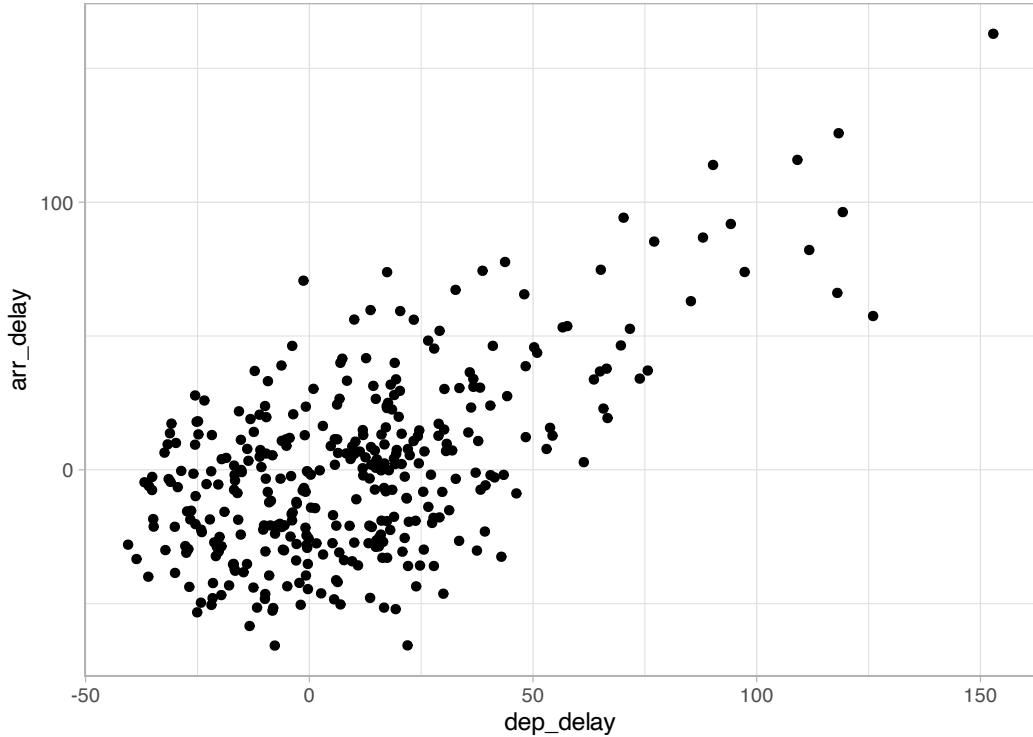


FIGURE 2.6: Arrival versus departure delays jittered scatterplot.

In order to specify how much jitter to add, we adjusted the `width` and `height` arguments to `geom_jitter()`. This corresponds to how hard you'd like to shake the plot in horizontal x-axis units and vertical y-axis units, respectively. In this case, both axes are in minutes. How much jitter should we add using the `width` and `height` arguments? On the one hand, it is important to add just enough jitter to break any overlap in points, but on the other hand, not so much that we completely alter the original pattern in points.

As can be seen in the resulting Figure 2.6, in this case jittering doesn't really provide much new insight. In this particular case, it can be argued that changing the transparency of the points by setting `alpha` proved more effective. When would it be better to use a jittered scatterplot? When would it be better to alter the points' transparency? There is no single right answer that applies to all situations. You need to make a subjective choice and own that choice. At the very least when confronted with overplotting, however, we suggest you make both types of plots and see which one better emphasizes the point you are trying to make.

Learning check

(LC2.7) Why is setting the `alpha` argument value useful with scatterplots? What further information does it give you that a regular scatterplot cannot?

(LC2.8) After viewing Figure 2.4, give an approximate range of arrival delays and departure delays that occur most frequently. How has that region changed compared to when you observed the same plot without `alpha = 0.2` set in Figure 2.2?

2.3.3 Summary

Scatterplots display the relationship between two numerical variables. They are among the most commonly used plots because they can provide an immediate way to see the trend in one numerical variable versus another. However, if you try to create a scatterplot where either one of the two variables is not numerical, you might get strange results. Be careful!

With medium to large datasets, you may need to play around with the different modifications to scatterplots we saw such as changing the transparency-opacity of the points or by jittering the points. This tweaking is often a fun part of data visualization, since you'll have the chance to see different relationships emerge as you tinker with your plots.

2.4 5NG#2: Linegraphs

The next of the five named graphs are linegraphs. Linegraphs show the relationship between two numerical variables when the variable on the x-axis, also called the *explanatory* variable, is of a sequential nature. In other words, there is an inherent ordering to the variable.

The most common examples of linegraphs have some notion of time on the x-axis: hours, days, weeks, years, etc. Since time is sequential, we connect consecutive observations of the variable on the y-axis with a line. Linegraphs that have some notion of time on the x-axis are also called *time series* plots. Let's illustrate linegraphs using another dataset in the `nycflights23` package: the `weather` data frame.

Let's explore the `weather` data frame from the `nycflights23` package by running `View(weather)` and `glimpse(weather)`. Furthermore let's read the associated help file by running `?weather` to bring up the help file.

Observe that there is a variable called `temp` of hourly wind speed recordings in miles per hour at weather stations near all three major airports in New York City: Newark (`origin` code `EWR`), John F. Kennedy International (`JFK`), and LaGuardia (`LGA`).

However, instead of considering hourly wind speeds for all days in 2023 for all three airports, for simplicity let's only consider hourly wind speeds at Newark airport for the first 15 days in January. This data is accessible in the `early_january_2023_weather` data frame included in the `moderndive` package. In other words, `early_january_2023_weather` contains hourly weather observations for `origin` equal to `EWR` (Newark's airport code), `month` equal to 1, and `day` less than or equal to 15.

Learning check

(LC2.9) Take a look at both the `weather` data frame from the `nycflights23` package and the `early_january_2023_weather` data frame from the `moderndive` package by running `View(weather)` and `View(early_january_2023_weather)`. In what respect do these data frames differ?

(LC2.10) `View()` the `flights` data frame again. Why does the `time_hour` variable uniquely identify the hour of the measurement, whereas the `hour` variable does not?

2.4.1 Linegraphs via `geom_line`

Let's create a time series plot of the hourly wind speeds saved in the `early_january_2023_weather` data frame by using `geom_line()` to create a linegraph, instead of using `geom_point()` like we used previously to create scatterplots:

```
ggplot(data = early_january_2023_weather,  
       mapping = aes(x = time_hour, y = wind_speed)) +  
       geom_line()
```

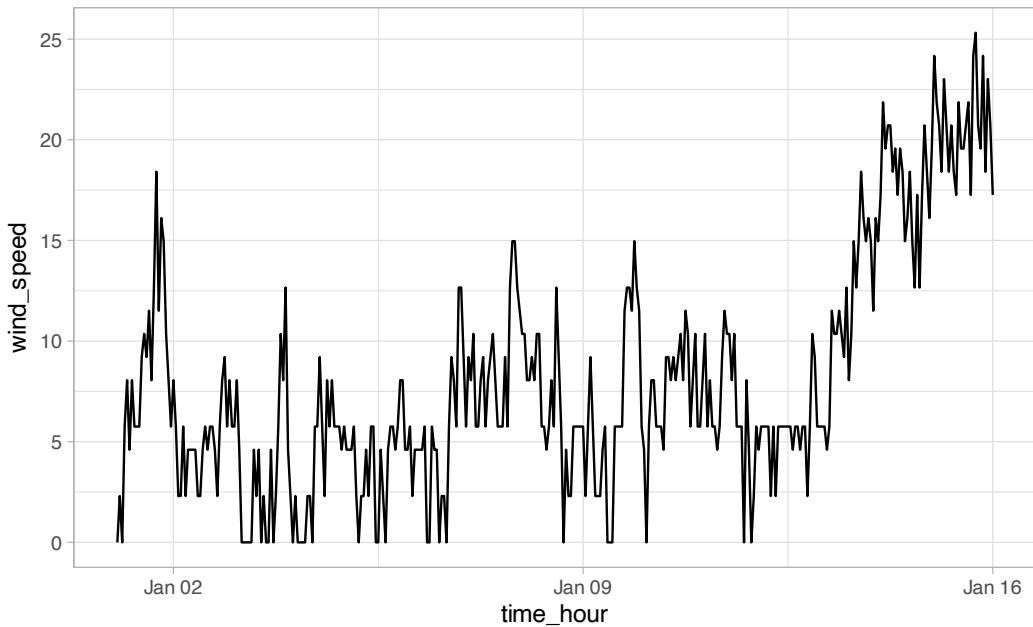


FIGURE 2.7: Hourly wind speed in Newark for January 1-15, 2023.

Much as with the `ggplot()` code that created the scatterplot of departure and arrival delays for Envoy Air flights in Figure 2.2, let's break down this code piece-by-piece in terms of the grammar of graphics:

Within the `ggplot()` function call, we specify two of the components of the grammar of graphics as arguments:

1. The data to be the `early_january_2023_weather` data frame by setting `data = early_january_2023_weather`.
2. The aesthetic mapping by setting `mapping = aes(x = time_hour, y = temp)`. Specifically, the variable `time_hour` maps to the `x` position aesthetic, while the variable `wind_speed` maps to the `y` position aesthetic.

We add a layer to the `ggplot()` function call using the `+` sign. The layer in question specifies the third component of the grammar: the geometric object in question. In this case, the geometric object is a line set by specifying `geom_line()`.

Learning check

(LC2.11) Why should linegraphs be avoided when there is not a clear ordering of the horizontal axis?

(LC2.12) Why are linegraphs frequently used when time is the explanatory variable on the x-axis?

(LC2.13) Plot a time series of a variable other than `wind_speed` for Newark Airport in the first 15 days of January 2023. Try to select a variable that doesn't have a lot of missing (`NA`) values.

2.4.2 Summary

Linegraphs, just like scatterplots, display the relationship between two numerical variables. However, it is preferred to use linegraphs over scatterplots when the variable on the x-axis (i.e., the explanatory variable) has an inherent ordering, such as some notion of time.

2.5 5NG#3: Histograms

Let's consider the `wind_speed` variable in the `weather` data frame once again, but unlike with the linegraphs in Section 2.4, let's say we don't care about its relationship with time, but rather we only care about how the values of `wind_speed` *distribute*. In other words:

1. What are the smallest and largest values?
2. What is the “center” or “most typical” value?
3. How do the values spread out?
4. What are frequent and infrequent values?

One way to visualize this *distribution* of this single variable `wind_speed` is to plot them on a horizontal line as we do in Figure 2.8:

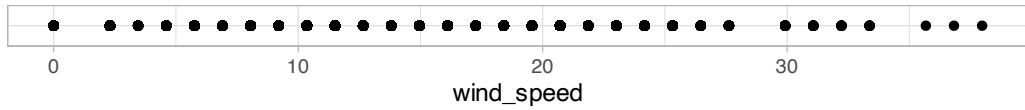


FIGURE 2.8: Plot of hourly wind speed recordings from NYC in 2023.

This gives us a general idea of how the values of `wind_speed` distribute: observe that wind speeds vary from around 0 miles per hour (0 kilometers per hour) up to 38 miles per hour (approximately 61 kilometers per hour). There appear to be more recorded wind speeds between 0 and 20 miles per hour (mph) than outside this range. However, because of the high degree of overplotting in the points, it's hard to get a sense of exactly how many values are between, say, 10 mph and 15 mph.

What is commonly produced instead of Figure 2.8 is known as a *histogram*. A histogram is a plot that visualizes the *distribution* of a numerical value as follows:

1. We first cut up the x-axis into a series of *bins*, where each bin represents a range of values.
2. For each bin, we count the number of observations that fall in the range corresponding to that bin.
3. Then for each bin, we draw a bar whose height marks the corresponding count.

Let's drill-down on an example of a histogram, shown in Figure 2.9.

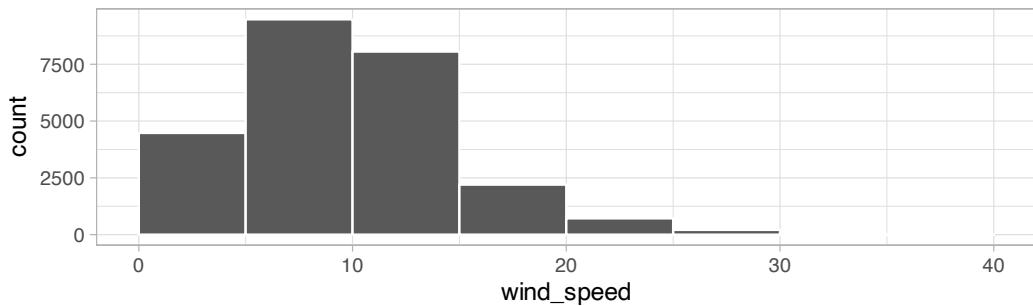


FIGURE 2.9: Example histogram.

Let's focus only on wind speeds between 10 mph and 25 mph for now. Observe that there are three bins of equal width between 10 mph and 25 mph. Thus we have three bins of width 5 mph each: one bin for the 10-15 mph range, another bin for the 15-20 mph range, and another bin for the 20-25 mph range. Since:

1. The bin for the 10-15 mph range has a height of around 8000. In other words, around 8000 of the hourly wind speed recordings are between 10 mph and 15 mph.
2. The bin for the 15-20 mph range has a height of around 2400. In other words, around 2400 of the hourly wind speed recordings are between 15 mph and 20 mph.
3. The bin for the 20-25 mph range has a height of around 700. In other words, around 700 of the hourly wind speed recordings are between 20 mph and 25 mph.

All eight bins spanning 0 mph to 40 mph on the x-axis have this interpretation.

2.5.1 Histograms via `geom_histogram`

Let's now present the `ggplot()` code to plot your first histogram! Unlike with scatterplots and linegraphs, there is now only one variable being mapped in `aes()`: the

single numerical variable `wind_speed`. The y-aesthetic of a histogram, the count of the observations in each bin, gets computed for you automatically. Furthermore, the geometric object layer is now a `geom_histogram()`. After running the following code, you'll see the histogram in Figure 2.10 as well as warning messages. We'll discuss the warning messages first.

```
ggplot(data = weather, mapping = aes(x = wind_speed)) +
  geom_histogram()
```

`'stat_bin()'` using `'bins = 30'`. Pick better value with `'binwidth'`.

Warning: Removed 1033 rows containing non-finite outside the scale range
(`'stat_bin()'`).

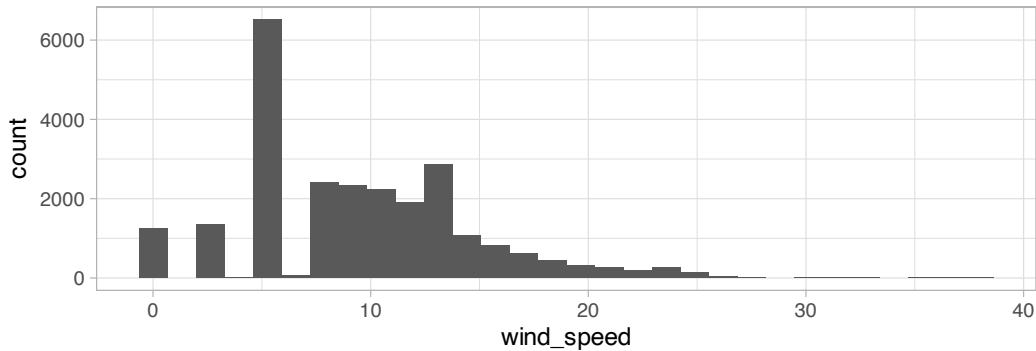


FIGURE 2.10: Histogram of hourly wind speeds at three NYC airports.

The first message is telling us that the histogram was constructed using `bins = 30` for 30 equally spaced bins. This is known in computer programming as a default value; unless you override this default number of bins with a number you specify, R will choose 30 by default. We'll see in the next section how to change the number of bins to another value than the default.

The second message is telling us something similar to the warning message we received when we ran the code to create a scatterplot of departure and arrival delays for Envoy Air flights in Figure 2.2: that because some rows have missing `NA` value for `wind_speed`, they were omitted from the histogram. R is just giving us a friendly heads up that this was the case.

Now let's unpack the resulting histogram in Figure 2.10. Observe that values above 30 mph are rather rare. However, because of the large number of bins, it's hard to get a sense for which range of wind speeds is spanned by each bin; everything is one giant amorphous blob. So let's add white vertical borders demarcating the bins by

adding a `color = "white"` argument to `geom_histogram()` and ignore the warning about setting the number of bins to a better value:

```
ggplot(data = weather, mapping = aes(x = wind_speed)) +
  geom_histogram(color = "white")
```

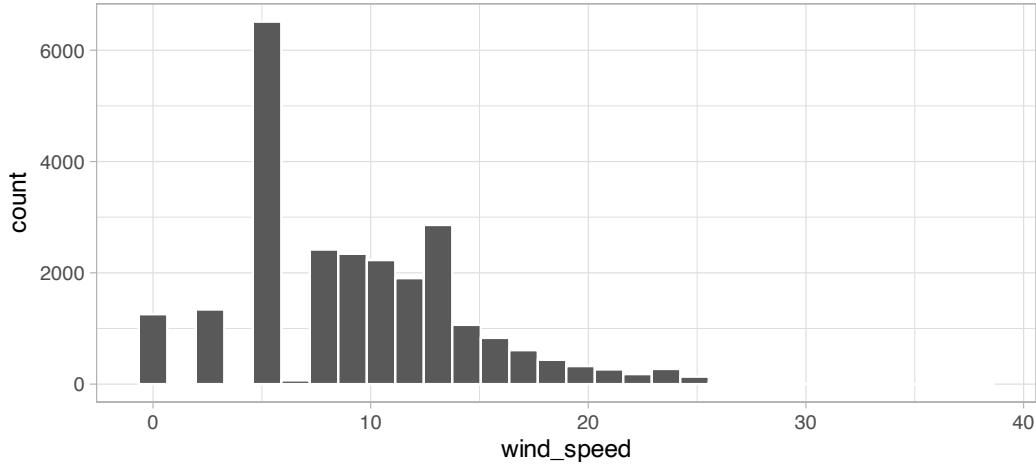


FIGURE 2.11: Histogram of hourly wind speeds at three NYC airports with white borders.

We now have an easier time associating ranges of wind speeds to each of the bins in Figure 2.11. We can also vary the color of the bars by setting the `fill` argument. For example, you can set the bin colors to be “blue steel” by setting `fill = "steelblue"`:

```
ggplot(data = weather, mapping = aes(x = wind_speed)) +
  geom_histogram(color = "white", fill = "steelblue")
```

If you’re curious, run `colors()` to see all 657 possible choice of colors in R!

2.5.2 Adjusting the bins

Observe in Figure 2.11 that in the 10-20 mph range there appear to be roughly 8 bins. Thus each bin has width 10 divided by 8, or 1.125 mph, which is not a very easily interpretable range to work with. Let’s improve this by adjusting the number of bins in our histogram in one of two ways:

1. By adjusting the number of bins via the `bins` argument to `geom_histogram()`.

2. By adjusting the width of the bins via the `binwidth` argument to `geom_histogram()`.

Using the first method, we have the power to specify how many bins we would like to cut the x-axis up in. As mentioned in the previous section, the default number of bins is 20. We can override this default, to say 20 bins, as follows:

```
ggplot(data = weather, mapping = aes(x = wind_speed)) +
  geom_histogram(bins = 20, color = "white")
```

Using the second method, instead of specifying the number of bins, we specify the width of the bins by using the `binwidth` argument in the `geom_histogram()` layer. For example, let's set the width of each bin to be five mph.

```
ggplot(data = weather, mapping = aes(x = wind_speed)) +
  geom_histogram(binwidth = 5, color = "white")
```

We compare both resulting histograms side-by-side in Figure 2.12.

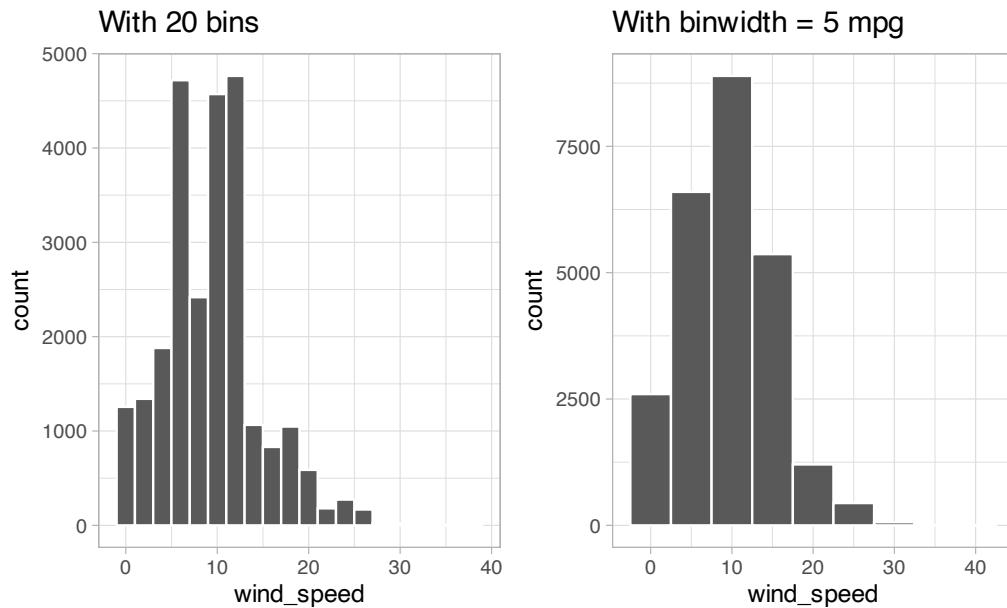


FIGURE 2.12: Setting histogram bins in two ways.

Learning check

(LC2.14) What does changing the number of bins from 30 to 20 tell us about the distribution of wind speeds?

(LC2.15) Would you classify the distribution of wind speeds as symmetric or skewed in one direction or another?

(LC2.16) What would you guess is the “center” value in this distribution? Why did you make that choice?

(LC2.17) Is this data spread out greatly from the center or is it close? Why?

2.5.3 Summary

Histograms, unlike scatterplots and linegraphs, present information on only a single numerical variable. Specifically, they are visualizations of the distribution of the numerical variable in question.

2.6 Facets

Before continuing with the next of the 5NG, let’s briefly introduce a new concept called *faceting*. Faceting is used when we’d like to split a particular visualization by the values of another variable. This will create multiple copies of the same type of plot with matching x and y axes, but whose content will differ.

For example, suppose we were interested in looking at how the histogram of hourly wind speed recordings at the three NYC airports we saw in Figure 2.9 differed in each month. We could “split” this histogram by the 12 possible months in a given year. In other words, we would plot histograms of `wind_speed` for each `month` separately. We do this by adding `facet_wrap(~ month)` layer. Note the ~ is a “tilde” and can generally be found on the key next to the “1” key on US keyboards. The tilde is required and you’ll receive the error `Error in as.quoted(facets) : object 'month' not found` if you don’t include it here.

```
ggplot(data = weather, mapping = aes(x = wind_speed)) +  
  geom_histogram(binwidth = 5, color = "white") +  
  facet_wrap(~ month)
```

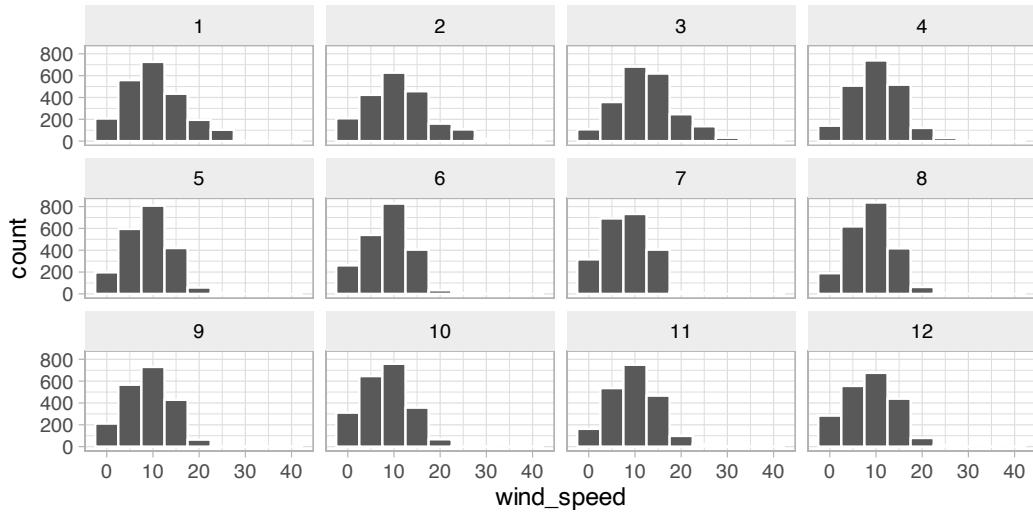


FIGURE 2.13: Faceted histogram of hourly wind speeds by month.

We can also specify the number of rows and columns in the grid by using the `nrow` and `ncol` arguments inside of `facet_wrap()`. For example, say we would like our faceted histogram to have 4 rows instead of 3. We simply add an `nrow = 4` argument to `facet_wrap(~ month)`.

```
ggplot(data = weather, mapping = aes(x = wind_speed)) +
  geom_histogram(binwidth = 5, color = "white") +
  facet_wrap(~ month, nrow = 4)
```

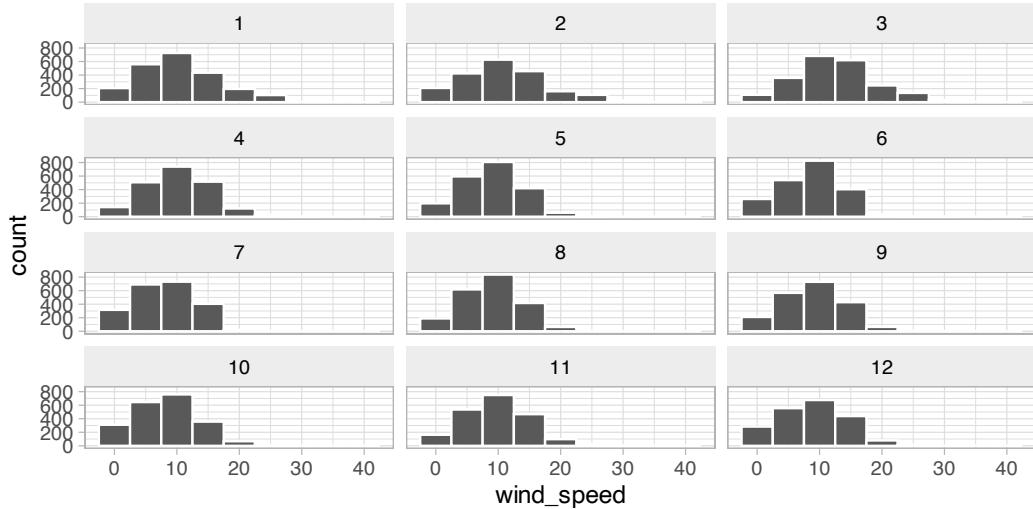


FIGURE 2.14: Faceted histogram with 4 instead of 3 rows.

Observe in both Figures 2.13 and 2.14 the majority of wind speed observations for all months are clustered between 0 and 20 mph, with very few observations exceeding 30 mph. The histograms show a similar shape across months, with most distributions having a similar largest count and a few larger speed outliers, indicating that lower wind speeds are more common than higher wind speeds.

Learning check

(LC2.18) What other things do you notice about this faceted plot? How does a faceted plot help us see relationships between two variables?

(LC2.19) What do the numbers 1-12 correspond to in the plot? What about 10, 20, and 30?

(LC2.20) For which types of datasets would faceted plots not work well in comparing relationships between variables? Give an example describing the nature of these variables and other important characteristics.

(LC2.21) Does the `wind_speed` variable in the `weather` dataset have a lot of variability? Why do you say that?

2.7 5NG#4: Boxplots

While faceted histograms are one type of visualization used to compare the distribution of a numerical variable split by the values of another variable, another type of visualization that achieves this same goal is a *side-by-side boxplot*. A boxplot is constructed from the information provided in the *five-number summary* of a numerical variable (see Appendix A.1).

To keep things simple for now, let's only consider the 2057 recorded hourly wind speed recordings for the month of April, each represented as a jittered point in Figure 2.15.

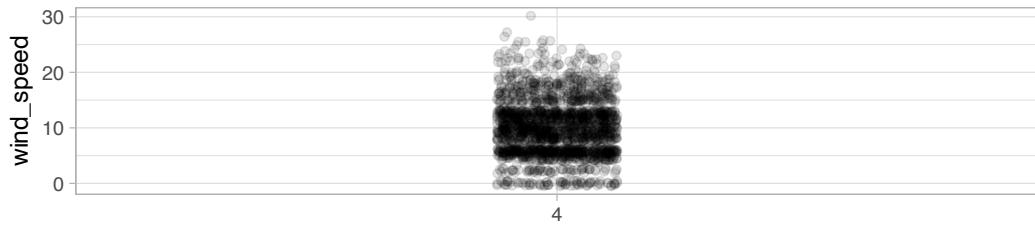
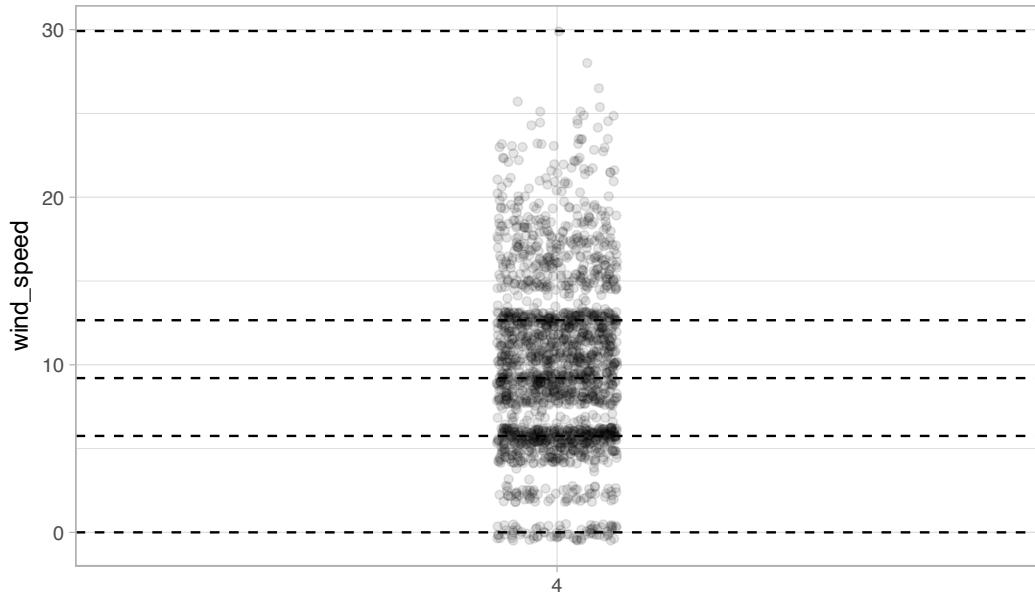


FIGURE 2.15: April wind speeds represented as jittered points.

These 2057 observations have the following *five-number summary*:

1. Minimum: 0 mph
2. First quartile (25th percentile): 5.8 mph
3. Median (second quartile, 50th percentile): 9.2 mph
4. Third quartile (75th percentile): 12.7 mph
5. Maximum: 29.92 mph

In the leftmost plot of Figure 2.16, let's mark these 5 values with dashed horizontal lines on top of the 2057 points. In the middle plot of Figure 2.16 let's add the *boxplot*. In the rightmost plot of Figure 2.16, let's remove the points and the dashed horizontal lines for clarity's sake.



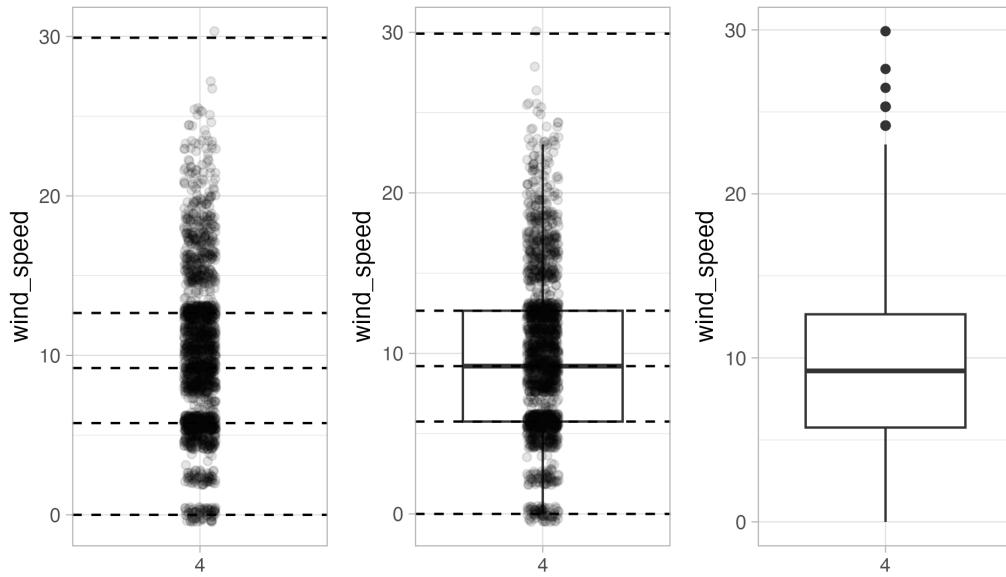


FIGURE 2.16: Building up a boxplot of April wind speeds.

What the boxplot does is visually summarize the 2057 points by cutting the wind speed recordings into *quartiles* at the dashed lines, where each quartile contains roughly $2057 \div 4 \approx 514$ observations. Thus

1. 25% of points fall below the bottom edge of the box, which is the first quartile of 5.8 mph. In other words, 25% of observations were below 5.8 mph.
2. 25% of points fall between the bottom edge of the box and the solid middle line, which is the median of 9.2 mph. Thus, 25% of observations were between 5.8 mph and 9.2 mph and 50% of observations were below 9.2 mph.
3. 25% of points fall between the solid middle line and the top edge of the box, which is the third quartile of 12.7 mph. It follows that 25% of observations were between 9.2 mph and 12.7 mph and 75% of observations were below 12.7 mph.
4. 25% of points fall above the top edge of the box. In other words, 25% of observations were above 12.7 mph.
5. The middle 50% of points lie within the *interquartile range (IQR)* between the first and third quartile. Thus, the IQR for this example is $12.7 - 5.8 = 6.905$ mph. The interquartile range is a measure of a numerical variable's *spread*.

Furthermore, in the rightmost plot of Figure 2.16, we see the *whiskers* of the boxplot. The whiskers stick out from either end of the box all the way to the minimum and maximum observed wind speeds of 0 mph and 29.92 mph, respectively. However, the

whiskers don't always extend to the smallest and largest observed values as they do here. They in fact extend no more than $1.5 \times$ the interquartile range from either end of the box, in this case of the April wind speeds, no more than 1.5×6.905 mph = 10.357 mph from either end of the box. Any observed values outside this range get marked with points called *outliers*, which are marked here, and we'll discuss further in the next section.

2.7.1 Boxplots via `geom_boxplot`

Let's now create a side-by-side boxplot of hourly wind speeds split by the 12 months as we did previously with the faceted histograms. We do this by mapping the `month` variable to the x-position aesthetic, the `wind_speed` variable to the y-position aesthetic, and by adding a `geom_boxplot()` layer:

```
ggplot(data = weather, mapping = aes(x = month, y = wind_speed)) +
  geom_boxplot()
```

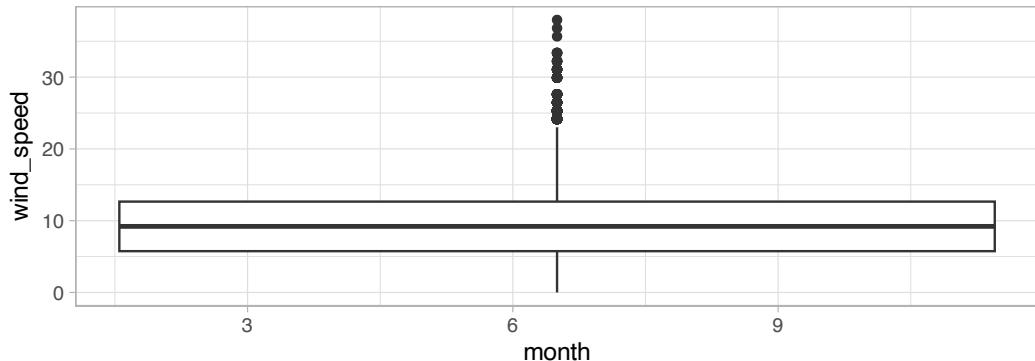


FIGURE 2.17: Invalid boxplot specification.

Warning message:

1: Continuous x aesthetic -- did you forget `aes(group=...)`?

Observe in Figure 2.17 that this plot does not provide information about wind speed separated by month. The first warning message clues us in as to why. It is telling us that we have a "continuous", or numerical variable, on the x-position aesthetic. Boxplots, however, require a categorical variable to be mapped to the x-position aesthetic.

We can convert the numerical variable `month` into a `factor` categorical variable by using the `factor()` function. So after applying `factor(month)`, month goes from having numerical values just the 1, 2, ..., and 12 to having an associated ordering. With this

ordering, `ggplot()` now knows how to work with this variable to produce the needed plot.

```
ggplot(data = weather, mapping = aes(x = factor(month), y = wind_speed)) +
  geom_boxplot()
```

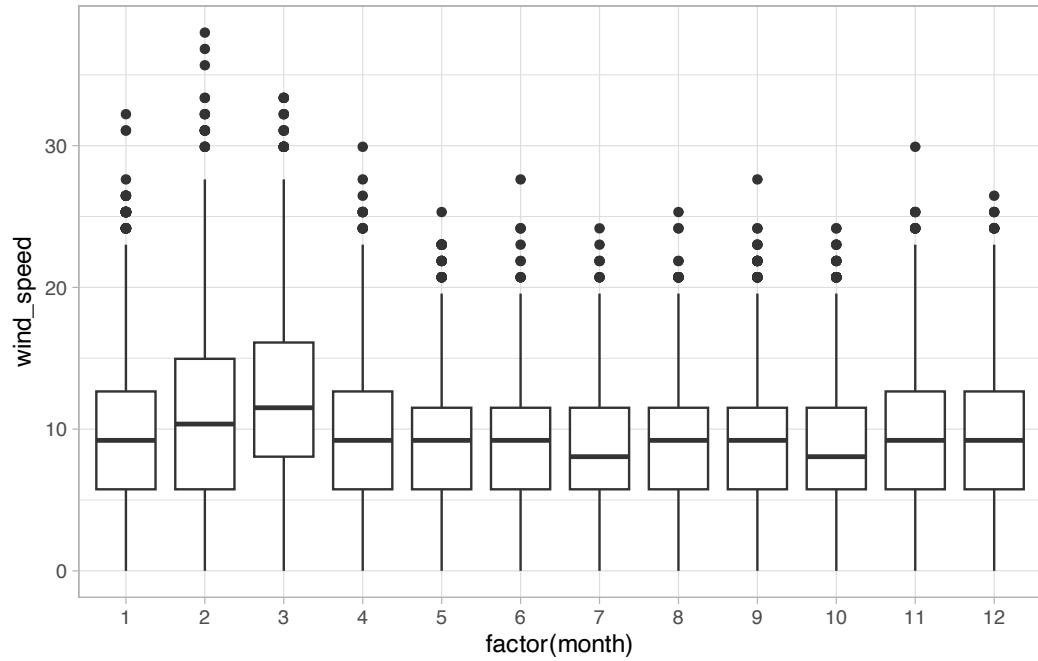


FIGURE 2.18: Side-by-side boxplot of wind speed split by month.

The resulting Figure 2.18 shows 12 separate “box and whiskers” plots similar to the rightmost plot of Figure 2.16 of only April wind speeds. Thus the different boxplots are shown “side-by-side.”

- The “box” portions of the visualization represent the 1st quartile, the median (the 2nd quartile), and the 3rd quartile.
- The height of each box (the value of the 3rd quartile minus the value of the 1st quartile) is the interquartile range (IQR). It is a measure of the spread of the middle 50% of values, with longer boxes indicating more variability.
- The “whisker” portions of these plots extend out from the bottoms and tops of the boxes and represent points less than the 25th percentile and greater than the 75th percentiles, respectively. They’re set to extend out no more than $1.5 \times IQR$ units away from either end of the boxes. We say “no more than” because the ends of the whiskers have to correspond to observed wind speeds. The length of these whiskers show how the data outside the middle 50% of values vary, with longer whiskers indicating more variability.

- The dots representing values falling outside the whiskers are called *outliers*. These can be thought of as anomalous (“out-of-the-ordinary”) values.

It is important to keep in mind that the definition of an outlier is somewhat arbitrary and not absolute. In this case, they are defined by the length of the whiskers, which are no more than $1.5 \times IQR$ units long for each boxplot. Looking at this side-by-side plot we can see that the months of February and March have higher median wind speeds as evidenced by the higher solid lines in the middle of the boxes. We can easily compare wind speeds across months by drawing imaginary horizontal lines across the plot. Furthermore, the heights of the 12 boxes as quantified by the interquartile ranges are informative too; they tell us about variability, or spread, of wind speeds recorded in a given month.

Learning check

(LC2.22) What do the dots at the top of the plot for January correspond to? Explain what might have occurred in January to produce these points.

(LC2.23) Which months seem to have the highest variability in wind speed? What reasons can you give for this?

(LC2.24) We looked at the distribution of the numerical variable `wind_speed` split by the numerical variable `month` that we converted using the `factor()` function in order to make a side-by-side boxplot. Why would a boxplot of `wind_speed` split by the numerical variable `pressure` similarly converted to a categorical variable using the `factor()` not be informative?

(LC2.25) Boxplots provide a simple way to identify outliers. Why may outliers be easier to identify when looking at a boxplot instead of a faceted histogram?

2.7.2 Summary

Side-by-side boxplots provide us with a way to compare the distribution of a numerical variable across multiple values of another variable. One can see where the median falls across the different groups by comparing the solid lines in the center of the boxes.

To study the spread of a numerical variable within one of the boxes, look at both the length of the box and also how far the whiskers extend from either end of the box. Outliers are even more easily identified when looking at a boxplot than when looking at a histogram as they are marked with distinct points.

2.8 5NG#5: Barplots

Both histograms and boxplots are tools to visualize the distribution of numerical variables. Another commonly desired task is to visualize the distribution of a categorical variable. This is a simpler task, as we are simply counting different categories within a categorical variable, also known as the *levels* of the categorical variable. Often the best way to visualize these different counts, also known as *frequencies*, is with barplots (also called barcharts).

One complication, however, is how your data is represented. Is the categorical variable of interest “pre-counted” or not? For example, run the following code that manually creates two data frames representing a collection of fruit: 3 apples and 2 oranges.

```
fruits <- tibble(
  fruit = c("apple", "apple", "orange", "apple", "orange")
)
fruits_counted <- tibble(
  fruit = c("apple", "orange"),
  number = c(3, 2)
)
```

We see both the `fruits` and `fruits_counted` data frames represent the same collection of fruit. Whereas `fruits` just lists the fruit individually...

```
# A tibble: 5 x 1
  fruit
  <chr>
1 apple
2 apple
3 orange
4 apple
5 orange
```

... `fruits_counted` has a variable `count` which represent the “pre-counted” values of each fruit.

```
# A tibble: 2 x 2
  fruit   number
  <chr>   <dbl>
1 apple     3
2 orange    2
```

Depending on how your categorical data is represented, you’ll need to add a different geometric layer type to your `ggplot()` to create a barplot, as we now explore.

2.8.1 Barplots via `geom_bar` or `geom_col`

Let's generate barplots using these two different representations of the same basket of fruit: 3 apples and 2 oranges. Using the `fruits` data frame where all 5 fruits are listed individually in 5 rows, we map the `fruit` variable to the x-position aesthetic and add a `geom_bar()` layer:

```
ggplot(data = fruits, mapping = aes(x = fruit)) +
  geom_bar()
```

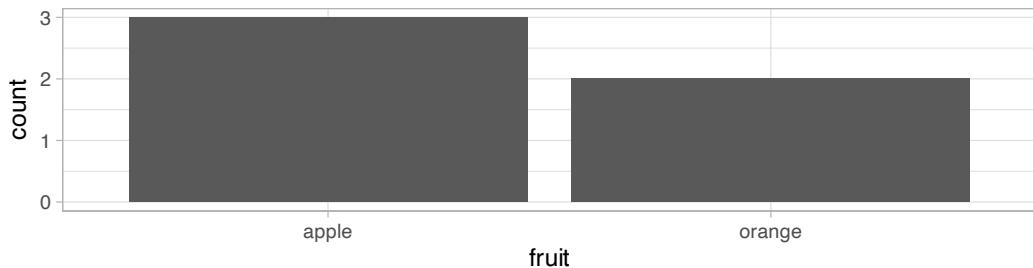


FIGURE 2.19: Barplot when counts are not pre-counted.

However, using the `fruits_counted` data frame where the fruits have been “pre-counted”, we once again map the `fruit` variable to the x-position aesthetic, but here we also map the `count` variable to the y-position aesthetic, and add a `geom_col()` layer instead.

```
ggplot(data = fruits_counted, mapping = aes(x = fruit, y = number)) +
  geom_col()
```

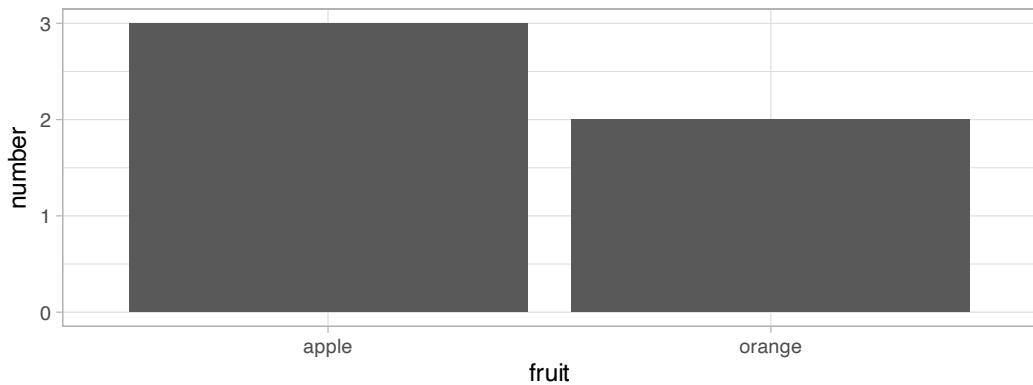


FIGURE 2.20: Barplot when counts are pre-counted.

Compare the barplots in Figures 2.19 and 2.20. They are identical because they reflect counts of the same five fruits. However, depending on how our categorical data is represented, either “pre-counted” or not, we must add a different `geom` layer. When the categorical variable whose distribution you want to visualize

- Is *not* pre-counted in your data frame, we use `geom_bar()`.
- Is pre-counted in your data frame, we use `geom_col()` with the y-position aesthetic mapped to the variable that has the counts.

Let’s now go back to the `flights` data frame in the `nycflights23` package and visualize the distribution of the categorical variable `carrier`. In other words, let’s visualize the number of domestic flights out of New York City each airline company flew in 2023. Recall from Subsection 1.4.3 when you first explored the `flights` data frame, you saw that each row corresponds to a flight. In other words, the `flights` data frame is more like the `fruits` data frame than the `fruits_counted` data frame because the flights have not been pre-counted by `carrier`. Thus we should use `geom_bar()` instead of `geom_col()` to create a barplot. Much like a `geom_histogram()`, there is only one variable in the `aes()` aesthetic mapping: the variable `carrier` gets mapped to the x-position. As a difference though, histograms have bars that touch whereas bar graphs have white space between the bars going from left to right.

```
ggplot(data = flights, mapping = aes(x = carrier)) +
  geom_bar()
```

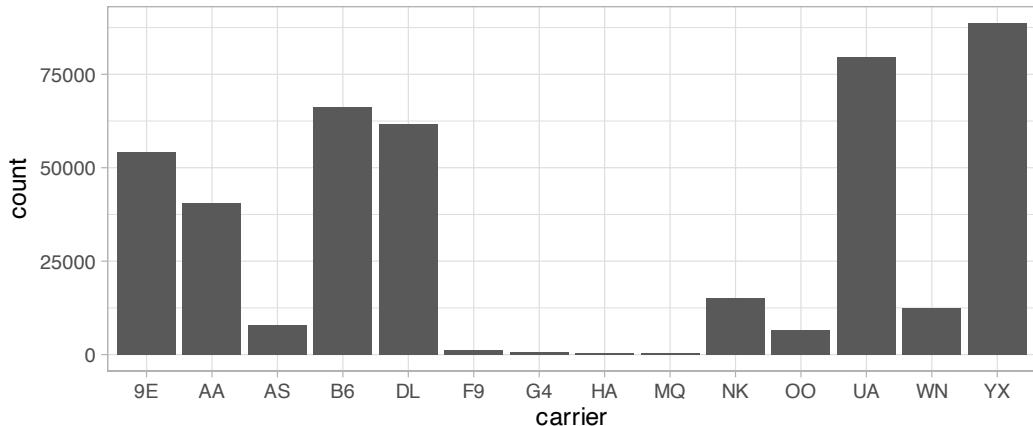


FIGURE 2.21: Number of flights departing NYC in 2023 by airline using `geom_bar()`.

Observe in Figure 2.21 that Republic Airline (YX), United Airlines (UA), and JetBlue Airways (B6) had the most flights depart NYC in 2023. If you don’t know which

airlines correspond to which carrier codes, then run `View(airlines)` to see a directory of airlines. For example, AA is American Airlines Inc. Alternatively, say you had a data frame where the number of flights for each carrier was pre-counted as in Table 2.3.

TABLE 2.3: Number of flights pre-counted for each carrier

carrier	number
9E	54141
AA	40525
AS	7843
B6	66169
DL	61562
F9	1286
G4	671
HA	366
MQ	357
NK	15189
OO	6432
UA	79641
WN	12385
YX	88785

In order to create a barplot visualizing the distribution of the categorical variable `carrier` in this case, we would now use `geom_col()` instead of `geom_bar()`, with an additional `y = number` in the aesthetic mapping on top of the `x = carrier`. The resulting barplot would be identical to Figure 2.21.

Learning check

(LC2.26) Why are histograms inappropriate for categorical variables?

(LC2.27) What is the difference between histograms and barplots?

(LC2.28) How many Alaska Air flights departed NYC in 2023?

(LC2.29) What was the 7th highest airline for departed flights from NYC in 2023? How could we better present the table to get this answer quickly?

2.8.2 Must avoid pie charts!

One of the most common plots used to visualize the distribution of categorical data is the pie chart. While they may seem harmless enough, pie charts actually present a

problem in that humans are unable to judge angles well. As Naomi Robbins describes in her book, *Creating More Effective Graphs* (Robbins, 2013), we overestimate angles greater than 90 degrees and we underestimate angles less than 90 degrees. In other words, it is difficult for us to determine the relative size of one piece of the pie compared to another.

Let's examine the same data used in our previous barplot of the number of flights departing NYC by airline in Figure 2.21, but this time we will use a pie chart in Figure 2.22. Try to answer the following questions:

- How much smaller is the portion of the pie for Hawaiian Airlines Inc. (HA) compared to US Airways (US)?
- What is the third largest carrier in terms of departing flights?
- How many carriers have fewer flights than Delta Air Lines Inc. (DL)?

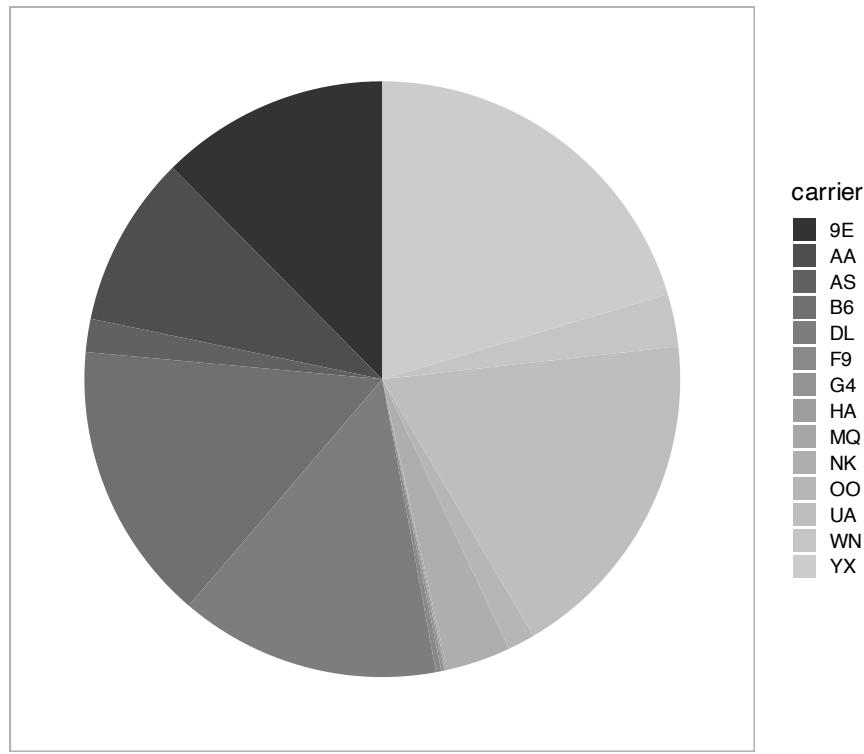


FIGURE 2.22: The dreaded pie chart.

While it is quite difficult to answer these questions when looking at the pie chart in Figure 2.22, we can much more easily answer these questions using the barchart in Figure 2.21. This is true since barplots present the information in a way such that comparisons between categories can be made with single horizontal lines, whereas pie charts present the information in a way such that comparisons must be made by comparing angles.

Learning check

(LC2.30) Why should pie charts be avoided and replaced by barplots?

(LC2.31) Why do you think people continue to use pie charts?

2.8.3 Two categorical variables

Barplots are a very common way to visualize the frequency of different categories, or levels, of a single categorical variable. Another use of barplots is to visualize the *joint* distribution of two categorical variables at the same time. Let's examine the *joint* distribution of outgoing domestic flights from NYC by `carrier` as well as `origin`, in other words, the number of flights for each `carrier` and `origin` combination. This corresponds to the number of American Airlines flights from `JFK`, the number of American Airlines flights from `LGA`, the number of American Airlines flights from `EWR`, the number of Endeavor Air flights from `JFK`, and so on. Recall the `ggplot()` code that created the barplot of `carrier` frequency in Figure 2.21:

```
ggplot(data = flights, mapping = aes(x = carrier)) +  
  geom_bar()
```

We can now map the additional variable `origin` by adding a `fill = origin` inside the `aes()` aesthetic mapping.

```
ggplot(data = flights, mapping = aes(x = carrier, fill = origin)) +  
  geom_bar()
```

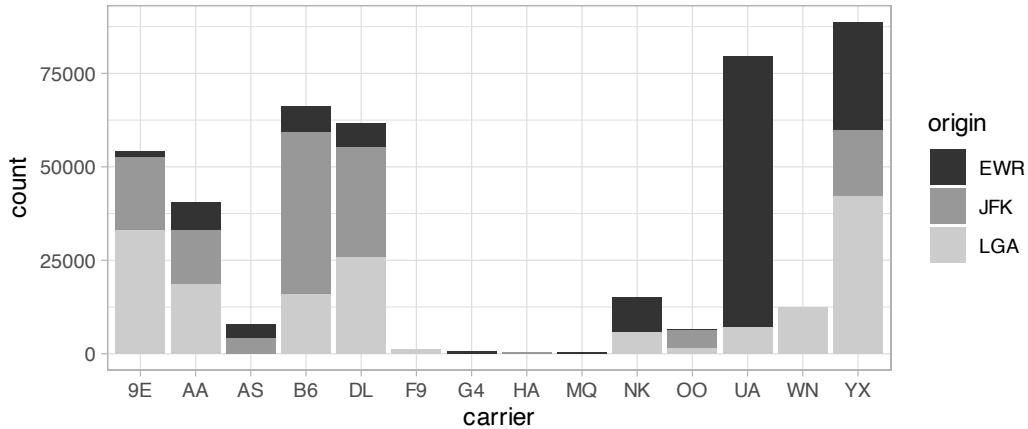


FIGURE 2.23: Stacked barplot of flight amount by carrier and origin.

Figure 2.23 is an example of a *stacked barplot*. While simple to make, in certain aspects it is not ideal. For example, it is difficult to compare the heights of the different colors between the bars, corresponding to comparing the number of flights from each `origin` airport between the carriers.

Before we continue, let's address some common points of confusion among new R users. First, the `fill` aesthetic corresponds to the color used to fill the bars, while the `color` aesthetic corresponds to the color of the outline of the bars. This is identical to how we added color to our histogram in Subsection 2.5.1: we set the outline of the bars to white by setting `color = "white"` and the colors of the bars to blue steel by setting `fill = "steelblue"`. Observe in Figure 2.24 that mapping `origin` to `color` and not `fill` yields grey bars with different colored outlines.

```
ggplot(data = flights, mapping = aes(x = carrier, color = origin)) +
  geom_bar()
```

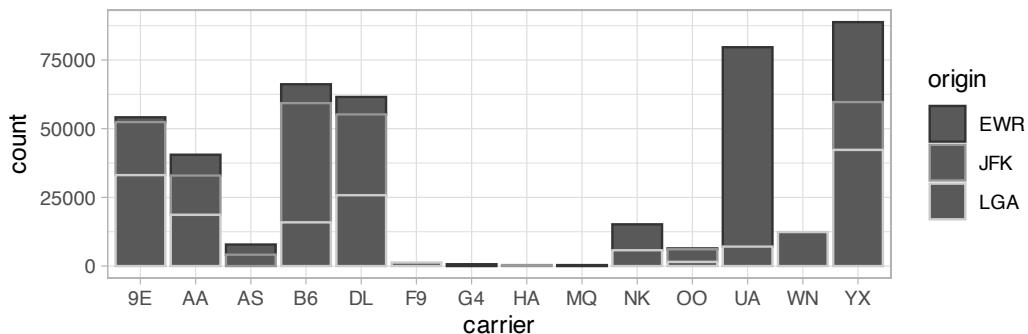


FIGURE 2.24: Stacked barplot with color aesthetic used instead of fill.

Second, note that `fill` is another aesthetic mapping much like `x`-position; thus we were careful to include it within the parentheses of the `aes()` mapping. The following code, where the `fill` aesthetic is specified outside the `aes()` mapping will yield an error. This is a fairly common error that new `ggplot` users make:

```
ggplot(data = flights, mapping = aes(x = carrier), fill = origin) +
  geom_bar()
```

An alternative to stacked barplots are *side-by-side barplots*, also known as *dodged barplots*, as seen in Figure 2.25. The code to create a side-by-side barplot is identical to the code to create a stacked barplot, but with a `position = "dodge"` argument added to `geom_bar()`. In other words, we are overriding the default barplot type, which is a *stacked* barplot, and specifying it to be a side-by-side barplot instead.

```
ggplot(data = flights, mapping = aes(x = carrier, fill = origin)) +
  geom_bar(position = "dodge")
```

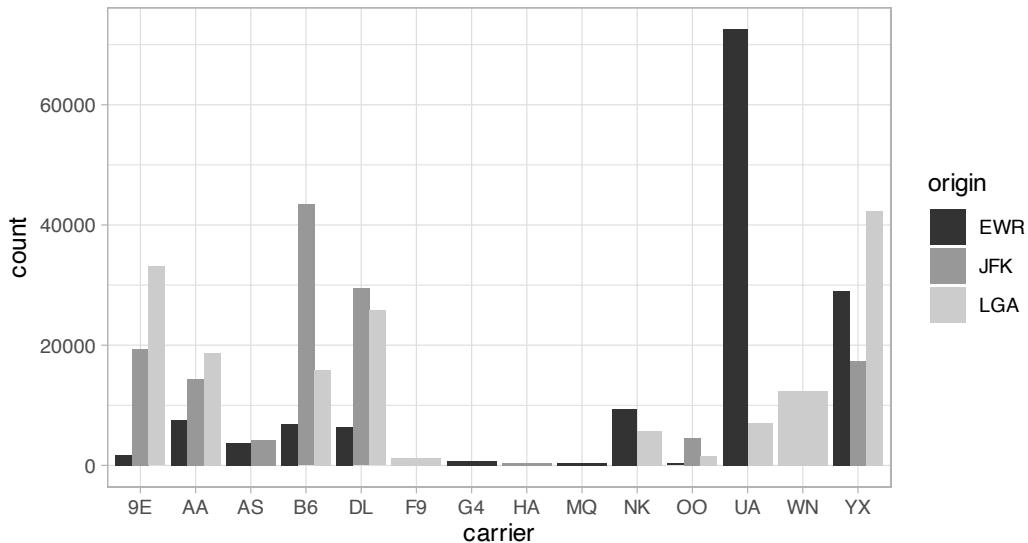


FIGURE 2.25: Side-by-side barplot comparing number of flights by carrier and origin.

Lastly, another type of barplot is a *faceted barplot*. Recall in Section 2.6 we visualized the distribution of hourly wind speeds at the 3 NYC airports *split* by month using facets. We apply the same principle to our barplot visualizing the frequency of `carrier` split by `origin`: instead of mapping `origin` to `fill` we include it as the variable to create small multiples of the plot across the levels of `origin`.

```
ggplot(data = flights, mapping = aes(x = carrier)) +
  geom_bar() +
  facet_wrap(~ origin, ncol = 1)
```

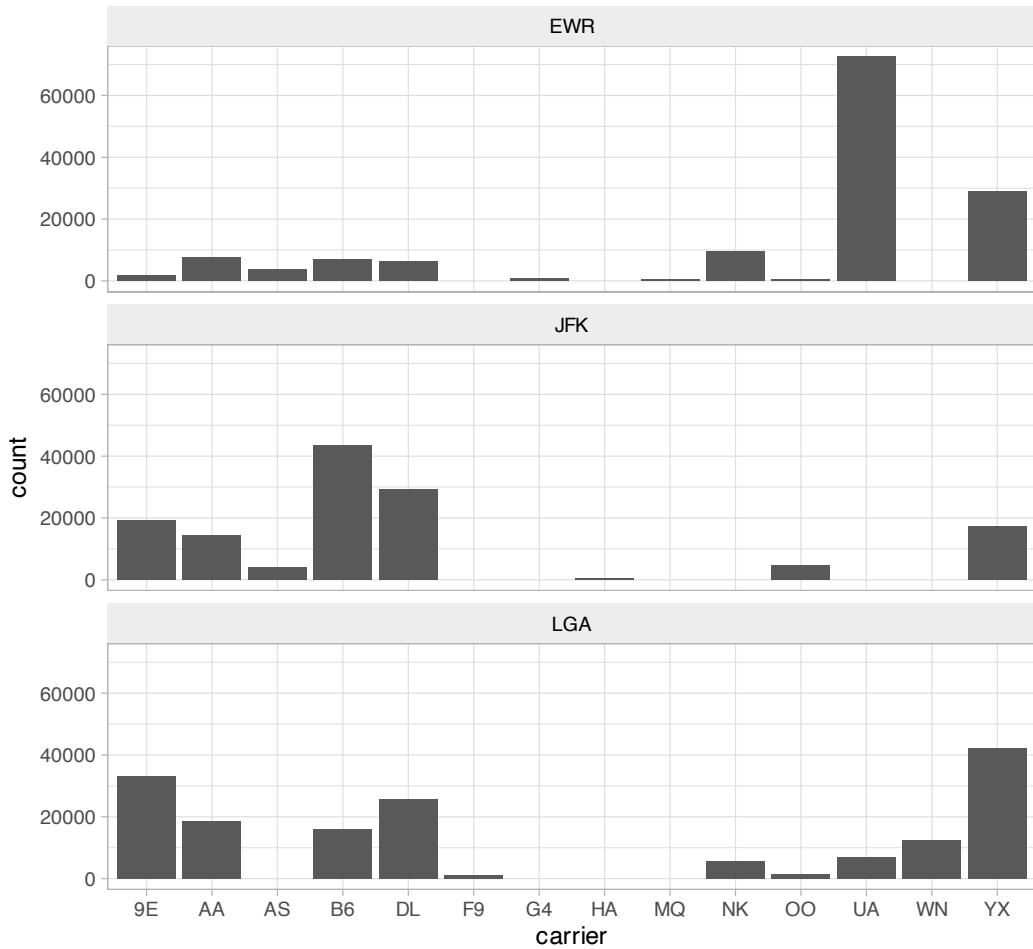


FIGURE 2.26: Faceted barplot comparing the number of flights by carrier and origin.

Learning check

(LC2.32) What kinds of questions are not easily answered by looking at Figure 2.23?

(LC2.33) What can you say, if anything, about the relationship between airline and airport in NYC in 2023 in regards to the number of departing flights?

(LC2.34) Why might the side-by-side barplot be preferable to a stacked barplot in this case?

(LC2.35) What are the disadvantages of using a dodged barplot, in general?

(LC2.36) Why is the faceted barplot preferred to the side-by-side and stacked barplots in this case?

(LC2.37) What information about the different carriers at different airports is more easily seen in the faceted barplot?

2.8.4 Summary

Barplots are a common way of displaying the distribution of a categorical variable, or in other words the frequency with which the different categories (also called *levels*) occur. They are easy to understand and make it easy to make comparisons across levels. Furthermore, when trying to visualize the relationship of two categorical variables, you have many options: stacked barplots, side-by-side barplots, and faceted barplots. Depending on what aspect of the relationship you are trying to emphasize, you will need to make a choice between these three types of barplots and own that choice.

2.9 Conclusion

2.9.1 Summary table

Let's recap all five of the five named graphs (5NG) in Table 2.4 summarizing their differences. Using these 5NG, you'll be able to visualize the distributions and relationships of variables contained in a wide array of datasets. This will be even more the case as we start to map more variables to more of each geometric object's aesthetic attribute options, further unlocking the awesome power of the `ggplot2` package.

TABLE 2.4: Summary of Five Named Graphs

Named graph	Shows	Geometric object	Notes
1 Scatterplot	Relationship between 2 numerical variables	geom_point()	
2 Linegraph	Relationship between 2 numerical variables	geom_line()	Used when there is a sequential order to x-variable, e.g., time
3 Histogram	Distribution of 1 numerical variable	geom_histogram()	Faceted histograms show the distribution of 1 numerical variable split by the values of another variable
4 Boxplot	Distribution of 1 numerical variable split by the values of another variable	geom_boxplot()	
5 Barplot	Distribution of 1 categorical variable	geom_bar() when counts are not pre-counted, geom_col() when counts are pre-counted	Stacked, side-by-side, and faceted barplots show the joint distribution of 2 categorical variables

2.9.2 Function argument specification

Let's go over some important points about specifying the arguments (i.e., inputs) to functions. Run the following two segments of code:

```
# Segment 1:
ggplot(data = flights, mapping = aes(x = carrier)) +
  geom_bar()

# Segment 2:
ggplot(flights, aes(x = carrier)) +
  geom_bar()
```

You'll notice that both code segments create the same barplot, even though in the second segment we omitted the `data =` and `mapping =` code argument names. This is because the `ggplot()` function by default assumes that the `data` argument comes first and the `mapping` argument comes second. As long as you specify the data frame in

question first and the `aes()` mapping second, you can omit the explicit statement of the argument names `data =` and `mapping =`.

Going forward for the rest of this book, all `ggplot()` code will be like the second segment: with the `data =` and `mapping =` explicit naming of the argument omitted with the default ordering of arguments respected. We'll do this for brevity's sake; it's common to see this style when reviewing other R users' code.

2.9.3 Additional resources

Solutions to all *Learning checks* can be found online in Appendix D⁶.

An R script file of all R code used in this chapter is available at <https://www.moderndive.com/scripts/02-visualization.R>.

If you want to further unlock the power of the `ggplot2` package for data visualization, we suggest that you check out RStudio's "Data Visualization with `ggplot2`" cheatsheet. This cheatsheet summarizes much more than what we've discussed in this chapter. In particular, it presents many more than the 5 geometric objects we covered in this chapter while providing quick and easy to read visual descriptions. For all the geometric objects, it also lists all the possible aesthetic attributes one can tweak. In the current version of RStudio in mid 2024, you can access this cheatsheet by going to the RStudio Menu Bar -> Help -> Cheatsheets -> "Data Visualization with `ggplot2`." Alternatively, you can preview the cheat sheet by going to the `ggplot2` Github page with this link⁷.

2.9.4 What's to come

Recall in Figure 2.2 in Section 2.3 we visualized the relationship between departure delay and arrival delay for Envoy Air flights only, rather than *all* flights. This data is saved in the `envoy_flights` data frame from the `moderndive` package.

In reality, the `envoy_flights` data frame is merely a subset of the `flights` data frame from the `nycflights23` package consisting of *all* flights that left NYC in 2023. We created `envoy_flights` using the following code that uses the `dplyr` package for data wrangling:

```
library(dplyr)

envoy_flights <- flights |>
  filter(carrier == "MQ")
```

⁶<https://moderndive.com/D-appendixD.html>

⁷<https://github.com/rstudio/cheatsheets/blob/main/data-visualization-2.1.pdf>

```
ggplot(data = envoy_flights, mapping = aes(x = dep_delay, y = arr_delay)) +
  geom_point()
```

This code takes the `flights` data frame and `filter()` it to only return the 357 rows where `carrier` is equal to "MQ", Envoy Air's carrier code. (Recall from Section 1.2 that testing for equality is specified with `==` and not `=`.) The code then cycles back to save the output in a new data frame called `envoy_flights` using the `<- assignment` operator.

Similarly, recall in Figure 2.7 in Section 2.4 we visualized hourly wind speed recordings at Newark airport only for the first 15 days of January 2023. This data is saved in the `early_january_2023_weather` data frame from the `moderndive` package.

In reality, the `early_january_2023_weather` data frame is merely a subset of the `weather` data frame from the `nycflights23` package consisting of *all* hourly weather observations in 2023 for *all* three NYC airports. We created `early_january_2023_weather` using the following `dplyr` code:

```
early_january_2023_weather <- weather |>
  filter(origin == "EWR" & month == 1 & day <= 15)

ggplot(data = early_january_2023_weather, mapping = aes(x = time_hour, y = temp)) +
  geom_line()
```

This code pares down the `weather` data frame to a new data frame `early_january_2023_weather` consisting of hourly wind speed recordings only for `origin == "EWR"`, `month == 1`, and `day` less than or equal to 15.

These two code segments are a preview of Chapter 3 on data wrangling using the `dplyr` package. Data wrangling is the process of transforming and modifying existing data with the intent of making it more appropriate for analysis purposes. For example, these two code segments used the `filter()` function to create new data frames (`envoy_flights` and `early_january_2023_weather`) by choosing only a subset of rows of existing data frames (`flights` and `weather`). In the next chapter, we'll formally introduce the `filter()` and other data wrangling functions as well as the *pipe operator* `|>` which allows you to combine multiple data wrangling actions into a single sequential *chain* of actions. On to Chapter 3 on data wrangling!

3

Data Wrangling

So far in our journey, we've seen how to look at data saved in data frames using the `glimpse()` and `view()` functions in Chapter 1, and how to create data visualizations using the `ggplot2` package in Chapter 2. In particular we studied what we term the “five named graphs” (5NG):

1. scatterplots via `geom_point()`
2. linegraphs via `geom_line()`
3. boxplots via `geom_boxplot()`
4. histograms via `geom_histogram()`
5. barplots via `geom_bar()` or `geom_col()`

We created these visualizations using the grammar of graphics, which maps variables in a data frame to the aesthetic attributes of one of the 5 geometric objects. We can also control other aesthetic attributes of the geometric objects such as the size and color as seen in the Gapminder data example in Figure 2.1.

In this chapter, we'll introduce a series of functions from the `dplyr` package for data wrangling that will allow you to take a data frame and “wrangle” it (transform it) to suit your needs. Such functions include:

1. `filter()` a data frame's existing rows to only pick out a subset of them. For example, the `alaska_flights` data frame.
2. `summarize()` one or more of its columns/variables with a *summary statistic*. Examples of summary statistics include the median and interquartile range of temperatures as we saw in Section 2.7 on boxplots.
3. `group_by()` its rows. In other words, assign different rows to be part of the same *group*. We can then combine `group_by()` with `summarize()` to report summary statistics for each group *separately*. For example, say you don't want a single overall average departure delay `dep_delay` for all three `origin` airports combined, but rather three separate average departure delays, one computed for each of the three `origin` airports.
4. `mutate()` its existing columns/variables to create new ones. For example, convert hourly temperature recordings from degrees Fahrenheit to degrees Celsius.
5. `arrange()` its rows. For example, sort the rows of `weather` in ascending or descending order of `temp`.

6. `join()` it with another data frame by matching along a “key” variable. In other words, merge these two data frames together.

Notice how we used `computer_code` font to describe the actions we want to take on our data frames. This is because the `dplyr` package for data wrangling has intuitively verb-named functions that are easy to remember.

There is a further benefit to learning to use the `dplyr` package for data wrangling: its similarity to the database querying language SQL¹ (pronounced “sequel” or spelled out as “S”, “Q”, “L”). SQL (which stands for “Structured Query Language”) is used to manage large databases quickly and efficiently and is widely used by many institutions with a lot of data. While SQL is a topic left for a book or a course on database management, keep in mind that once you learn `dplyr`, you can learn SQL easily. We’ll talk more about their similarities in Subsection 3.7.4.

Needed packages

Let’s load all the packages needed for this chapter (this assumes you’ve already installed them). If needed, read Section 1.3 for information on how to install and load R packages.

```
library(dplyr)
library(ggplot2)
library(nycflights23)
```

3.1 The pipe operator: `|>`

Before we start data wrangling, let’s first introduce a nifty tool that has been a part of R since May 2021: the native pipe operator `|>`. The pipe operator allows us to combine multiple operations in R into a single sequential *chain* of actions. In modern R, the native pipe operator `|>` is now the default for chaining functions, replacing the previously common tidyverse pipe (`%>%`) that was loaded with the `dplyr` package. Introduced in R 4.1.0 in May 2021, `|>` offers a more intuitive and readable syntax for data wrangling and other tasks, eliminating the need for additional package dependencies. You’ll still often see R code using `%>%` in older scripts or searches online, but we’ll use `|>` in this book. The tidyverse pipe still works, so don’t worry if you see it in other code.

¹<https://en.wikipedia.org/wiki/SQL>

Let's start with a hypothetical example. Say you would like to perform a hypothetical sequence of operations on a hypothetical data frame x using hypothetical functions $f()$, $g()$, and $h()$:

1. Take x *then*
2. Use x as an input to a function $f()$ *then*
3. Use the output of $f(x)$ as an input to a function $g()$ *then*
4. Use the output of $g(f(x))$ as an input to a function $h()$

One way to achieve this sequence of operations is by using nesting parentheses as follows:

```
h(g(f(x)))
```

This code isn't so hard to read since we are applying only three functions: $f()$, then $g()$, then $h()$ and each of the functions is short in its name. Further, each of these functions also only has one argument. However, you can imagine that this will get progressively harder to read as the number of functions applied in your sequence increases and the arguments in each function increase as well. This is where the pipe operator $|>$ comes in handy. $|>$ takes the output of one function and then "pipes" it to be the input of the next function. Furthermore, a helpful trick is to read $|>$ as "then" or "and then." For example, you can obtain the same output as the hypothetical sequence of functions as follows:

```
x |>
  f() |>
  g() |>
  h()
```

You would read this sequence as:

1. Take x *then*
2. Use this output as the input to the next function $f()$ *then*
3. Use this output as the input to the next function $g()$ *then*
4. Use this output as the input to the next function $h()$

So while both approaches achieve the same goal, the latter is much more human-readable because you can clearly read the sequence of operations line-by-line. But what are the hypothetical x , $f()$, $g()$, and $h()$? Throughout this chapter on data wrangling:

1. The starting value x will be a data frame. For example, the `flights` data frame we explored in Section 1.4.

2. The sequence of functions, here `f()`, `g()`, and `h()`, will mostly be a sequence of any number of the six data wrangling verb-named functions we listed in the introduction to this chapter. For example, the `filter(carrier == "MQ")` function and argument specified we previewed earlier.
3. The result will be the transformed/modified data frame that you want. In our example, we'll save the result in a new data frame by using the `<-` assignment operator with the name `alaska_flights` via `alaska_flights <-`.

```
envoy_flights <- flights |>
  filter(carrier == "AS")
```

Much like when adding layers to a `ggplot()` using the `+` sign, you form a single *chain* of data wrangling operations by combining verb-named functions into a single sequence using the pipe operator `|>`. Furthermore, much like how the `+` sign has to come at the end of lines when constructing plots, the pipe operator `|>` has to come at the end of lines as well.

Keep in mind, there are many more advanced data wrangling functions than just the six listed in the introduction to this chapter; you'll see some examples of these in Section 3.8. However, just with these six verb-named functions you'll be able to perform a broad array of data wrangling tasks for the rest of this book.

3.2 filter rows

Subset Observations (Rows)



FIGURE 3.1: Diagram of `filter()` rows operation.

The `filter()` function here works much like the “Filter” option in Microsoft Excel; it allows you to specify criteria about the values of a variable in your dataset and then filters out only the rows that match that criteria.

We begin by focusing only on flights from New York City to Phoenix, Arizona. The `dest` destination code (or airport code) for Phoenix, Arizona is “`PHX`”. Run the

following and look at the results in RStudio's spreadsheet viewer to ensure that only flights heading to Portland are chosen here:

```
phoenix_flights <- flights |>
  filter(dest == "PHX")
View(phoenix_flights)
```

Note the order of the code. First, take the `flights` data frame `flights` *then* `filter()` the data frame so that only those where the `dest` equals "PHX" are included. We test for equality using the double equal sign `==` and not a single equal sign `=`. In other words `filter(dest = "PHX")` will yield an error. This is a convention across many programming languages. If you are new to coding, you'll probably forget to use the double equal sign `==` a few times before you get the hang of it.

You can use other operators beyond just the `==` operator that tests for equality:

- `>` corresponds to “greater than”
- `<` corresponds to “less than”
- `>=` corresponds to “greater than or equal to”
- `<=` corresponds to “less than or equal to”
- `!=` corresponds to “not equal to.” The `!` is used in many programming languages to indicate “not.”

Furthermore, you can combine multiple criteria using operators that make comparisons:

- `|` corresponds to “or”
- `&` corresponds to “and”

To see many of these in action, let's filter `flights` for all rows that departed from JFK *and* were heading to Burlington, Vermont ("BTV") or Seattle, Washington ("SEA") *and* departed in the months of October, November, or December. Run the following:

```
btv_sea_flights_fall <- flights |>
  filter(origin == "JFK" & (dest == "BTV" | dest == "SEA") & month >= 10)
View(btv_sea_flights_fall)
```

Note that even though colloquially speaking one might say “all flights leaving Burlington, Vermont *and* Seattle, Washington,” in terms of computer operations, we really mean “all flights leaving Burlington, Vermont *or* leaving Seattle, Washington.” For a given row in the data, `dest` can be "BTV", or "SEA", or something else, but not both

"BTV" and "SEA" at the same time. Furthermore, note the careful use of parentheses around `dest == "BTV" | dest == "SEA"`.

We can often skip the use of `&` and just separate our conditions with a comma. The previous code will return the identical output `btv_sea_flights_fall` as the following code:

```
btv_sea_flights_fall <- flights |>
  filter(origin == "JFK", (dest == "BTV" | dest == "SEA"), month >= 10)
View(btv_sea_flights_fall)
```

Let's present another example that uses the `!` "not" operator to pick rows that *don't* match a criteria. As mentioned earlier, the `!` can be read as "not." Here we are filtering rows corresponding to flights that didn't go to Burlington, VT or Seattle, WA.

```
not_BTV_SEA <- flights |>
  filter(!(dest == "BTV" | dest == "SEA"))
View(not_BTV_SEA)
```

Again, note the careful use of parentheses around the `(dest == "BTV" | dest == "SEA")`. If we didn't use parentheses as follows:

```
flights |> filter(!dest == "BTV" | dest == "SEA")
```

We would be returning all flights not headed to "BTV" *or* those headed to "SEA", which is an entirely different resulting data frame.

Now say we have a larger number of airports we want to filter for, say "SEA", "SFO", "PHX", "BTV", and "BDL". We could continue to use the `|` (*or*) operator.

```
many_airports <- flights |>
  filter(dest == "SEA" | dest == "SFO" | dest == "PHX" |
        dest == "BTV" | dest == "BDL")
```

As we progressively include more airports, this will get unwieldy to write. A slightly shorter approach uses the `%in%` operator along with the `c()` function. Recall from Subsection 1.2.1 that the `c()` function "combines" or "concatenates" values into a single *vector* of values.

```
many_airports <- flights |>
  filter(dest %in% c("SEA", "SFO", "PHX", "BTV", "BDL"))
View(many_airports)
```

What this code is doing is filtering `flights` for all flights where `dest` is in the vector of airports `c("BTV", "SEA", "PHX", "SFO", "BDL")`. Both outputs of `many_airports` are the same, but as you can see the latter takes much less energy to code. The `%in%` operator is useful for looking for matches commonly in one vector/variable compared to another.

As a final note, we recommend that `filter()` should often be among the first verbs you consider applying to your data. This cleans your dataset to only those rows you care about, or put differently, it narrows down the scope of your data frame to just the observations you care about.

Learning check

(LC3.1) What's another way of using the “not” operator `!` to filter only the rows that are not going to Burlington, VT nor Seattle, WA in the `flights` data frame? Test this out using the previous code.

3.3 summarize variables

The next common task when working with data frames is to compute *summary statistics*. Summary statistics are single numerical values that summarize a large number of values. Commonly known examples of summary statistics include the mean (also called the average) and the median (the middle value). Other examples of summary statistics that might not immediately come to mind include the *sum*, the smallest value also called the *minimum*, the largest value also called the *maximum*, and the *standard deviation*. See Appendix A.1 for a glossary of such summary statistics.

Let's calculate two summary statistics of the `wind_speed` temperature variable in the `weather` data frame: the mean and standard deviation (recall from Section 1.4 that the `weather` data frame is included in the `nycflights23` package). To compute these summary statistics, we need the `mean()` and `sd()` *summary functions* in R. Summary functions in R take in many values and return a single value, as illustrated in Figure 3.2.



FIGURE 3.2: Diagram illustrating a summary function in R.

More precisely, we'll use the `mean()` and `sd()` summary functions within the `summarize()` function from the `dplyr` package. Note you can also use the British English spelling of `summarise()`. As shown in Figure 3.3, the `summarize()` function takes in a data frame and returns a data frame with only one row corresponding to the summary statistics.

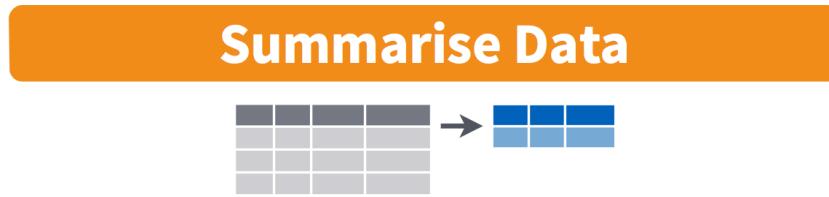


FIGURE 3.3: Diagram of summarize() rows.

We'll save the results in a new data frame called `summary_windspeed` that will have two columns/variables: the `mean` and the `std_dev`:

```
summary_windspeed <- weather |>
  summarize(mean = mean(wind_speed), std_dev = sd(wind_speed))
summary_windspeed
```

```
# A tibble: 1 × 2
  mean std_dev
  <dbl>   <dbl>
1     NA      NA
```

Why are the values returned `NA`? `NA` is how R encodes *missing values* where `NA` indicates “not available” or “not applicable.” If a value for a particular row and a particular column does not exist, `NA` is stored instead. Values can be missing for many reasons. Perhaps the data was collected but someone forgot to enter it? Perhaps the data was not collected at all because it was too difficult to do so? Perhaps there was an erroneous value that someone entered that has been corrected to read as missing? You’ll often encounter issues with missing values when working with real data.

Going back to our `summary_windspeed` output, by default any time you try to calculate a summary statistic of a variable that has one or more `NA` missing values in R, `NA` is returned. To work around this fact, you can set the `na.rm` argument to `TRUE`, where `rm` is short for “remove”; this will ignore any `NA` missing values and only return the summary value for all non-missing values.

The code that follows computes the mean and standard deviation of all non-missing values of `temp`:

```
summary_windspeed <- weather |>
  summarize(mean = mean(wind_speed, na.rm = TRUE),
            std_dev = sd(wind_speed, na.rm = TRUE))
summary_windspeed
```

```
# A tibble: 1 x 2
  mean std_dev
  <dbl>   <dbl>
1 9.44    5.26
```

Notice how the `na.rm = TRUE` are used as arguments to the `mean()` and `sd()` summary functions individually, and not to the `summarize()` function.

However, one needs to be cautious whenever ignoring missing values as we’ve just done. In the upcoming *Learning checks* questions, we’ll consider the possible ramifications of blindly sweeping rows with missing values “under the rug.” This is in fact why the `na.rm` argument to any summary statistic function in R is set to `FALSE` by default. In other words, R does not ignore rows with missing values by default. R is alerting you to the presence of missing data and you should be mindful of this missingness and any potential causes of this missingness throughout your analysis.

What are other summary functions we can use inside the `summarize()` verb to compute summary statistics? As seen in the diagram in Figure 3.2, you can use any function in R that takes many values and returns just one. Here are just a few:

- `mean()`: the average
- `sd()`: the standard deviation, which is a measure of spread
- `min()` and `max()`: the minimum and maximum values, respectively
- `IQR()`: interquartile range
- `sum()`: the total amount when adding multiple numbers
- `n()`: a count of the number of rows in each group. This particular summary function will make more sense when `group_by()` is covered in Section 3.4.

Learning check

(LC3.2) Say a doctor is studying the effect of smoking on lung cancer for a large number of patients who have records measured at five-year intervals. She notices that a large number of patients have missing data points because the patient has died, so she chooses to ignore these patients in her analysis. What is wrong with this doctor's approach?

(LC3.3) Modify the earlier `summarize()` function code that creates the `summary_windspeed` data frame to also use the `n()` summary function: `summarize(..., count = n())`. What does the returned value correspond to?

(LC3.4) Why doesn't the following code work? Run the code line-by-line instead of all at once, and then look at the data. In other words, select and then run `summary_windspeed <- weather |> summarize(mean = mean(wind_speed, na.rm = TRUE))` first.

```
summary_windspeed <- weather |>
  summarize(mean = mean(wind_speed, na.rm = TRUE)) |>
  summarize(std_dev = sd(wind_speed, na.rm = TRUE))
```

3.4 group_by rows



FIGURE 3.4: Diagram of `group_by()` and `summarize()`.

We can modify our code above to look at the average wind speed and its spread instead of wind speed too, keeping the `na.rm = TRUE` set just in case any missing values are stored in the `temp` column:

```
summary_temp <- weather |>
  summarize(mean = mean(wind_speed, na.rm = TRUE),
            std_dev = sd(wind_speed, na.rm = TRUE))
summary_temp
```

```
# A tibble: 1 x 2
  mean std_dev
  <dbl>   <dbl>
1 9.44    5.26
```

Say instead of a single mean wind speed for the whole year, we would like 12 mean temperatures, one for each of the 12 months separately. In other words, we would like to compute the mean wind speed split by month. We can do this by “grouping” temperature observations by the values of another variable, in this case by the 12 values of the variable `month`:

```
summary_monthly_windspeed <- weather |>
  group_by(month) |>
  summarize(mean = mean(wind_speed, na.rm = TRUE),
            std_dev = sd(wind_speed, na.rm = TRUE))
summary_monthly_windspeed
```

```
# A tibble: 12 x 3
  month  mean std_dev
  <int> <dbl>   <dbl>
1     1 10.3    6.01
2     2 10.9    6.57
3     3 12.3    6.33
4     4 10.0    5.03
5     5  8.89   4.46
6     6  8.53   4.43
7     7  7.98   4.35
8     8  8.85   4.34
9     9  8.92   4.66
10    10  8.23   4.69
11    11  9.50   4.84
12    12  8.77   5.02
```

This code is identical to the previous code that created `summary_windspeed`, but with an extra `group_by(month)` added before the `summarize()`. Grouping the `weather` dataset by `month` and then applying the `summarize()` functions yields a data frame that displays the mean and standard deviation wind speed split by the 12 months of the year.

It is important to note that the `group_by()` function doesn't change data frames by itself. Rather it changes the *meta-data*, or data about the data, specifically the grouping structure. It is only after we apply the `summarize()` function that the data frame changes.

As another example, let's consider the `diamonds` data frame included in the `ggplot2` package. Run this code:

```
diamonds
```

```
# A tibble: 53,940 x 10
  carat cut      color clarity depth table price     x     y     z
  <dbl> <ord>    <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
1 0.23 Ideal    E     SI2     61.5   55   326   3.95  3.98  2.43
2 0.21 Premium  E     SI1     59.8   61   326   3.89  3.84  2.31
3 0.23 Good     E     VS1     56.9   65   327   4.05  4.07  2.31
4 0.29 Premium  I     VS2     62.4   58   334   4.2    4.23  2.63
5 0.31 Good     J     SI2     63.3   58   335   4.34  4.35  2.75
6 0.24 Very Good J    VVS2    62.8   57   336   3.94  3.96  2.48
7 0.24 Very Good I    VVS1    62.3   57   336   3.95  3.98  2.47
8 0.26 Very Good H    SI1     61.9   55   337   4.07  4.11  2.53
9 0.22 Fair      E     VS2     65.1   61   337   3.87  3.78  2.49
10 0.23 Very Good H   VS1     59.4   61   338    4     4.05  2.39
# i 53,930 more rows
```

Observe that the first line of the output reads `# A tibble: 53,940 x 10`. This is an example of meta-data, in this case the number of observations/rows and variables/columns in `diamonds`. The actual data itself are the subsequent table of values. Now let's pipe the `diamonds` data frame into `group_by(cut)`:

```
diamonds |>
  group_by(cut)
```

```
# A tibble: 53,940 x 10
# Groups:   cut [5]
  carat cut      color clarity depth table price     x     y     z
  <dbl> <ord>    <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
1 0.23 Ideal    E     SI2     61.5   55   326   3.95  3.98  2.43
2 0.21 Premium  E     SI1     59.8   61   326   3.89  3.84  2.31
3 0.23 Good     E     VS1     56.9   65   327   4.05  4.07  2.31
4 0.29 Premium  I     VS2     62.4   58   334   4.2    4.23  2.63
5 0.31 Good     J     SI2     63.3   58   335   4.34  4.35  2.75
```

```

6 0.24 Very Good J    VVS2      62.8    57    336  3.94  3.96  2.48
7 0.24 Very Good I    VVS1      62.3    57    336  3.95  3.98  2.47
8 0.26 Very Good H    SI1       61.9    55    337  4.07  4.11  2.53
9 0.22 Fair     E    VS2       65.1    61    337  3.87  3.78  2.49
10 0.23 Very Good H   VS1       59.4    61   338   4    4.05  2.39
# i 53,930 more rows

```

Observe that now there is additional meta-data: # Groups: cut [5] indicating that the grouping structure meta-data has been set based on the 5 possible levels of the categorical variable cut: "Fair", "Good", "Very Good", "Premium", and "Ideal". On the other hand, observe that the data has not changed: it is still a table of $53,940 \times 10$ values.

Only by combining a `group_by()` with another data wrangling operation, in this case `summarize()`, will the data actually be transformed.

```

diamonds |>
  group_by(cut) |>
  summarize(avg_price = mean(price))

```

```

# A tibble: 5 x 2
  cut      avg_price
  <ord>     <dbl>
1 Fair        4359.
2 Good        3929.
3 Very Good  3982.
4 Premium     4584.
5 Ideal       3458.

```

If you would like to remove this grouping structure meta-data, we can pipe the resulting data frame into the `ungroup()` function:

```

diamonds |>
  group_by(cut) |>
  ungroup()

```

```

# A tibble: 53,940 x 10
  carat cut      color clarity depth table price     x     y     z
  <dbl> <ord>    <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
1 0.23 Ideal    E     SI2      61.5    55    326  3.95  3.98  2.43
2 0.21 Premium  E     SI1      59.8    61    326  3.89  3.84  2.31

```

```

3 0.23 Good     E   VS1    56.9   65   327  4.05  4.07  2.31
4 0.29 Premium  I   VS2    62.4   58   334  4.2   4.23  2.63
5 0.31 Good     J   SI2    63.3   58   335  4.34  4.35  2.75
6 0.24 Very Good J   VVS2   62.8   57   336  3.94  3.96  2.48
7 0.24 Very Good I   VVS1   62.3   57   336  3.95  3.98  2.47
8 0.26 Very Good H   SI1    61.9   55   337  4.07  4.11  2.53
9 0.22 Fair     E   VS2    65.1   61   337  3.87  3.78  2.49
10 0.23 Very Good H  VS1    59.4   61   338  4     4.05  2.39
# i 53,930 more rows

```

Observe how the # Groups: cut [5] meta-data is no longer present.

Let's now revisit the `n()` counting summary function we briefly introduced previously. Recall that the `n()` function counts rows. This is opposed to the `sum()` summary function that returns the sum of a numerical variable. For example, suppose we'd like to count how many flights departed each of the three airports in New York City:

```

by_origin <- flights |>
  group_by(origin) |>
  summarize(count = n())
by_origin

```

```

# A tibble: 3 x 2
  origin  count
  <chr>   <int>
1 EWR     138578
2 JFK     133048
3 LGA     163726

```

We see that LaGuardia ("LGA") had the most flights departing in 2023 followed by Newark ("EWR") and lastly by "JFK". Note there is a subtle but important difference between `sum()` and `n()`; while `sum()` returns the sum of a numerical variable, `n()` returns a count of the number of rows/observations.

Grouping by more than one variable

You are not limited to grouping by one variable. Say you want to know the number of flights leaving each of the three New York City airports *for each month*. We can also group by a second variable `month` using `group_by(origin, month)`:

```

by_origin_monthly <- flights |>
  group_by(origin, month) |>
  summarize(count = n())

```

```
`summarise()` has grouped output by 'origin'. You can override using the  
.groups` argument.
```

```
by_origin_monthly
```

```
# A tibble: 36 x 3  
# Groups:   origin [3]  
  origin month count  
  <chr>  <int> <int>  
1 EWR      1 11623  
2 EWR      2 10991  
3 EWR      3 12593  
4 EWR      4 12022  
5 EWR      5 12371  
6 EWR      6 11339  
7 EWR      7 11646  
8 EWR      8 11561  
9 EWR      9 11373  
10 EWR     10 11805  
# i 26 more rows
```

Note that an additional message appears here specifying the grouping done. The `.groups` argument to `summarize()` has four options: `drop_last`, `drop`, `keep`, and `rowwise`:

- `drop_last` drops the last grouping variable,
- `drop` drops all grouping variables,
- `keep` keeps all grouping variables, and
- `rowwise` turns each row into a group.

In most circumstances that you'll encounter, the default is `drop_last` which drops the last grouping variable. The message is informing us that the default behavior is to drop the last grouping variable, which in this case is `month`.

Observe that there are 36 rows to `by_origin_monthly` because there are 12 months for 3 airports (`EWR`, `JFK`, and `LGA`). Why do we `group_by(origin, month)` and not `group_by(origin)` and then `group_by(month)`? Let's investigate:

```
by_origin_monthly_incorrect <- flights |>  
  group_by(origin) |>  
  group_by(month) |>  
  summarise(count = n())  
by_origin_monthly_incorrect
```

```
# A tibble: 12 x 2
  month count
  <int> <int>
1     1 36020
2     2 34761
3     3 39514
4     4 37476
5     5 38710
6     6 35921
7     7 36211
8     8 36765
9     9 35505
10   10 36586
11   11 34521
12   12 33362
```

What happened here is that the second `group_by(month)` overwrote the grouping structure meta-data of the earlier `group_by(origin)`, so that in the end we are only grouping by `month`. The lesson here is if you want to `group_by()` two or more variables, you should include all the variables at the same time in the same `group_by()` adding a comma between the variable names.

Learning check

(LC3.5) Recall from Chapter 2 when we looked at wind speeds by months in NYC. What does the standard deviation column in the `summary_monthly_temp` data frame tell us about temperatures in NYC throughout the year?

(LC3.6) What code would be required to get the mean and standard deviation wind speed for each day in 2023 for NYC?

(LC3.7) Recreate `by_monthly_origin`, but instead of grouping via `group_by(origin, month)`, group variables in a different order `group_by(month, origin)`. What differs in the resulting dataset?

(LC3.8) How could we identify how many flights left each of the three airports for each `carrier`?

(LC3.9) How does the `filter()` operation differ from a `group_by()` followed by a `summarize()`?

3.5 mutate existing variables

Make New Variables

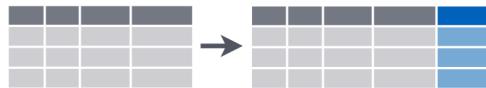


FIGURE 3.5: Diagram of `mutate()` columns.

Another common transformation of data is to create/compute new variables based on existing ones. For example, say you are more comfortable thinking of temperature in degrees Celsius ($^{\circ}\text{C}$) instead of degrees Fahrenheit ($^{\circ}\text{F}$). The formula to convert temperatures from $^{\circ}\text{F}$ to $^{\circ}\text{C}$ is

$$\text{temp in C} = \frac{\text{temp in F} - 32}{1.8}$$

We can apply this formula to the `temp` variable using the `mutate()` function from the `dplyr` package, which takes existing variables and mutates them to create new ones.

```
weather <- weather |>
  mutate(temp_in_C = (temp - 32) / 1.8)
```

In this code, we `mutate()` the `weather` data frame by creating a new variable `temp_in_C` = `(temp - 32) / 1.8` and then *overwrite* the original `weather` data frame. Why did we overwrite the data frame `weather`, instead of assigning the result to a new data frame like `weather_new`? As a rough rule of thumb, as long as you are not losing original information that you might need later, it's acceptable practice to overwrite existing data frames with updated ones, as we did here. On the other hand, why did we not overwrite the variable `temp`, but instead created a new variable called `temp_in_C`? Because if we did this, we would have erased the original information contained in `temp` of temperatures in Fahrenheit that may still be valuable to us.

Let's now compute monthly average temperatures in both $^{\circ}\text{F}$ and $^{\circ}\text{C}$ using the `group_by()` and `summarize()` code we saw in Section 3.4:

```
summary_monthly_temp <- weather |>
  group_by(month) |>
  summarize(mean_temp_in_F = mean(temp, na.rm = TRUE),
```

```
mean_temp_in_C = mean(temp_in_C, na.rm = TRUE))
summary_monthly_temp
```

```
# A tibble: 12 x 3
  month mean_temp_in_F mean_temp_in_C
  <int>      <dbl>      <dbl>
1     1       35.7       2.04
2     2       34.5       1.39
3     3       45.0       7.24
4     4       54.6      12.6
5     5       53.6      12.0
6     6       69.2      20.6
7     7       78.4      25.8
8     8       72.8      22.7
9     9       64.7      18.1
10    10      64.2      17.9
11    11      47.5      8.64
12    12      45.9      7.72
```

Let's consider another example. Passengers are often frustrated when their flight departs late, but aren't as annoyed if, in the end, pilots can make up some time during the flight. This is known in the airline industry as *gain*, and we will create this variable using the `mutate()` function:

```
flights <- flights |>
  mutate(gain = dep_delay - arr_delay)
```

Let's take a look at only the `dep_delay`, `arr_delay`, and the resulting `gain` variables for the first 5 rows in our updated `flights` data frame in Table 3.1.

TABLE 3.1: First five rows of departure/arrival delay and gain variables

dep_delay	arr_delay	gain
203	205	-2
78	53	25
47	34	13
173	166	7
228	211	17

The flight in the first row departed 203 minutes late but arrived 205 minutes late, so its “gained time in the air” is a gain of -2 minutes, hence its `gain` is $203 - 205 = -2$, which is a loss of 2 minutes. On the other hand, the flight in the third row departed

late (`dep_delay` of 47) but arrived 34 minutes late (`arr_delay` of 34), so its “gained time in the air” is $47 - 34 = 13$ minutes, hence its `gain` is 13.

Let’s look at some summary statistics of the `gain` variable by considering multiple summary functions at once in the same `summarize()` code:

```
gain_summary <- flights |>
  summarize(
    min = min(gain, na.rm = TRUE),
    q1 = quantile(gain, 0.25, na.rm = TRUE),
    median = quantile(gain, 0.5, na.rm = TRUE),
    q3 = quantile(gain, 0.75, na.rm = TRUE),
    max = max(gain, na.rm = TRUE),
    mean = mean(gain, na.rm = TRUE),
    sd = sd(gain, na.rm = TRUE),
    missing = sum(is.na(gain))
  )
gain_summary
```

```
# A tibble: 1 x 8
  min     q1 median     q3   max   mean     sd missing
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>   <int>
1 -321     1     11    20    101   9.35  18.4    12534
```

We see for example that the median gain is 11 minutes, while the largest is +101 minutes and the largest negative gain (or loss) at -321 minutes! However, this code would take some time to type out in practice. We’ll see later on in Subsection 5.1.1 that there is a much more succinct way to compute a variety of common summary statistics: using the `skim()` function from the `skimr` package.

Recall from Section 2.5 that since `gain` is a numerical variable, we can visualize its distribution using a histogram.

```
ggplot(data = flights, mapping = aes(x = gain)) +
  geom_histogram(color = "white", bins = 20)
```

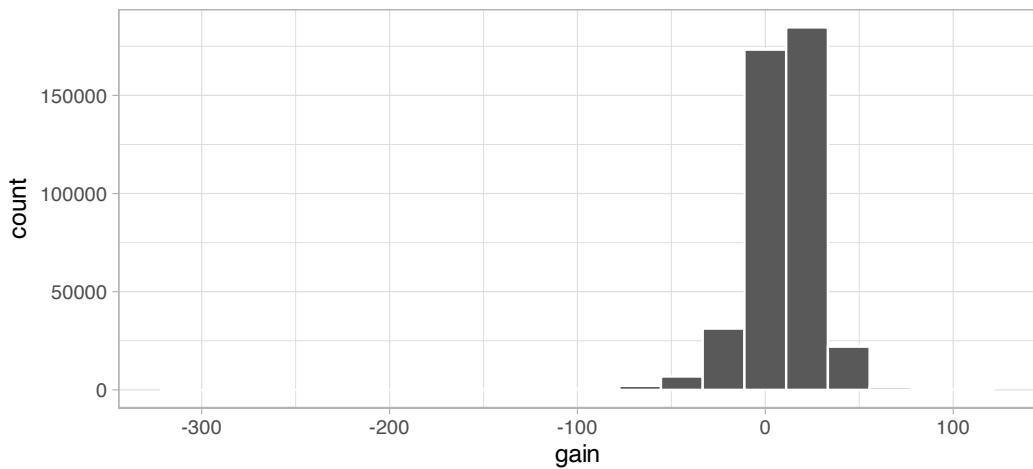


FIGURE 3.6: Histogram of gain variable.

The resulting histogram in Figure 3.6 provides additional perspective on the `gain` variable than the summary statistics we computed earlier. For example, note that most values of `gain` are right around 0.

To close out our discussion on the `mutate()` function to create new variables, note that we can create multiple new variables at once in the same `mutate()` code. Furthermore, within the same `mutate()` code we can refer to new variables we just created. As an example, consider the `mutate()` code Hadley Wickham and Garrett Grolemund show in Chapter 5 of *R for Data Science* (Grolemund and Wickham, 2017):

```
flights <- flights |>
  mutate(
    gain = dep_delay - arr_delay,
    hours = air_time / 60,
    gain_per_hour = gain / hours
  )
```

Learning check

(LC3.10) What do positive values of the `gain` variable in `flights` correspond to? What about negative values? And what about a zero value?

(LC3.11) Could we create the `dep_delay` and `arr_delay` columns by simply subtracting `dep_time` from `sched_dep_time` and similarly for arrivals? Try the code out and explain any differences between the result and what actually appears in `flights`.

(LC3.12) What can we say about the distribution of gain? Describe it in a few sentences using the plot and the gain_summary data frame values.

3.6 arrange and sort rows

One of the most commonly performed data wrangling tasks is to sort a data frame's rows in the alphanumeric order of one of the variables. The `dplyr` package's `arrange()` function allows us to sort/reorder a data frame's rows according to the values of the specified variable.

Suppose we are interested in determining the most frequent destination airports for all domestic flights departing from New York City in 2023:

```
freq_dest <- flights |>
  group_by(dest) |>
  summarize(num_flights = n())
freq_dest
```

```
# A tibble: 118 x 2
  dest    num_flights
  <chr>      <int>
1 ABQ        228
2 ACK        916
3 AGS         20
4 ALB       1581
5 ANC         95
6 ATL       17570
7 AUS        4848
8 AVL        1617
9 AVP         145
10 BDL        701
# i 108 more rows
```

Observe that by default the rows of the resulting `freq_dest` data frame are sorted in alphabetical order of `dest`. Say instead we would like to see the same data, but sorted from the most to the least number of flights (`num_flights`) instead:

```
freq_dest |>
  arrange(num_flights)
```

```
# A tibble: 118 x 2
  dest   num_flights
  <chr>     <int>
1 LEX          1
2 AGS         20
3 OGG         20
4 SBN         24
5 HDN         28
6 PNS         71
7 MTJ         77
8 ANC         95
9 VPS        109
10 AVP        145
# i 108 more rows
```

This is, however, the opposite of what we want. The rows are sorted with the least frequent destination airports displayed first. This is because `arrange()` always returns rows sorted in ascending order by default. To switch the ordering to be in “descending” order instead, we use the `desc()` function as so:

```
freq_dest |>
  arrange(desc(num_flights))
```

```
# A tibble: 118 x 2
  dest   num_flights
  <chr>     <int>
1 BOS        19036
2 ORD        18200
3 MCO        17756
4 ATL        17570
5 MIA        16076
6 LAX        15968
7 FLL        14239
8 CLT        12866
9 DFW        11675
10 SFO       11651
# i 108 more rows
```

3.7 join data frames

Another common data transformation task is “joining” or “merging” two different datasets. For example, in the `flights` data frame, the variable `carrier` lists the carrier code for the different flights. While the corresponding airline names for "UA" and "AA" might be somewhat easy to guess (United and American Airlines), what airlines have codes "VX", "HA", and "B6"? This information is provided in a separate data frame `airlines`.

```
View(airlines)
```

We see that in `airlines`, `carrier` is the carrier code, while `name` is the full name of the airline company. Using this table, we can see that "G4", "HA", and "B6" correspond to Allegiant Air, Hawaiian Airlines, and JetBlue, respectively. However, wouldn't it be nice to have all this information in a single data frame instead of two separate data frames? We can do this by “joining” the `flights` and `airlines` data frames.

Note that the values in the variable `carrier` in the `flights` data frame match the values in the variable `carrier` in the `airlines` data frame. In this case, we can use the variable `carrier` as a *key variable* to match the rows of the two data frames. Key variables are almost always *identification variables* that uniquely identify the observational units as we saw in Subsection 1.4.4. This ensures that rows in both data frames are appropriately matched during the join. Hadley and Garrett ([Grolemund and Wickham, 2017](#)) created the diagram shown in Figure 3.7 to help us understand how the different data frames in the `nycflights23` package are linked by various key variables:

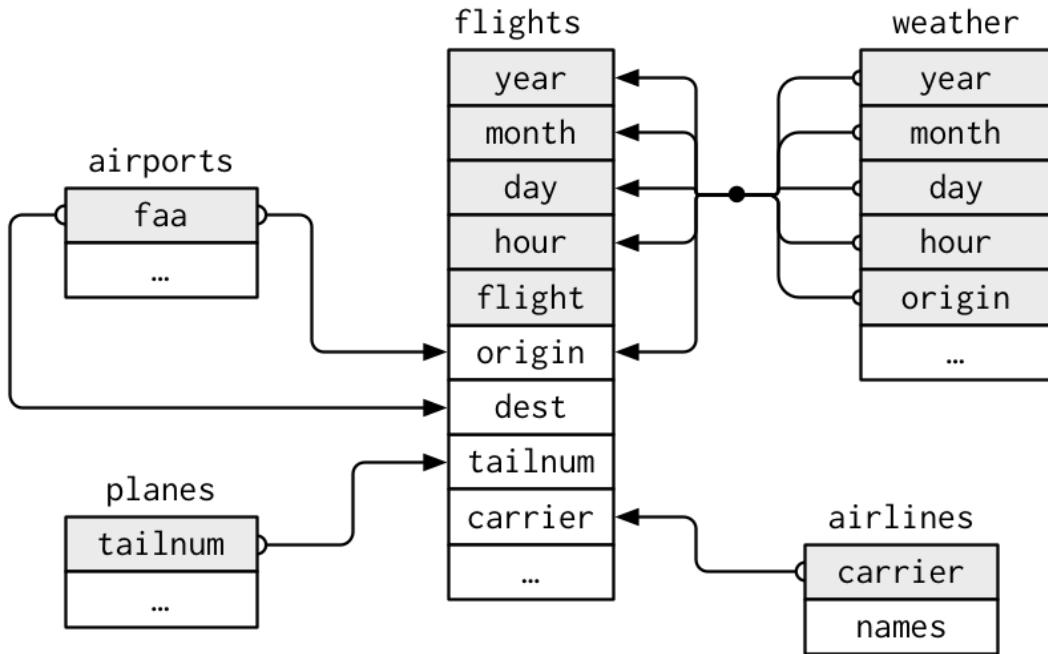


FIGURE 3.7: Data relationships in `nycflights` from *R for Data Science*.

3.7.1 Matching “key” variable names

In both the `flights` and `airlines` data frames, the key variable we want to join/merge/match the rows by has the same name: `carrier`. Let’s use the `inner_join()` function to join the two data frames, where the rows will be matched by the variable `carrier`, and then compare the resulting data frames:

```

flights_joined <- flights |>
  inner_join(airlines, by = "carrier")
View(flights)
View(flights_joined)
  
```

Observe that the `flights` and `flights_joined` data frames are identical except that `flights_joined` has an additional variable `name`. The values of `name` correspond to the airline companies’ names as indicated in the `airlines` data frame.

A visual representation of the `inner_join()` is shown in Figure 3.8 (Grollemund and Wickham, 2017). There are other types of joins available (such as `left_join()`, `right_join()`, `outer_join()`, and `anti_join()`), but the `inner_join()` will solve nearly all of the problems you’ll encounter in this book.

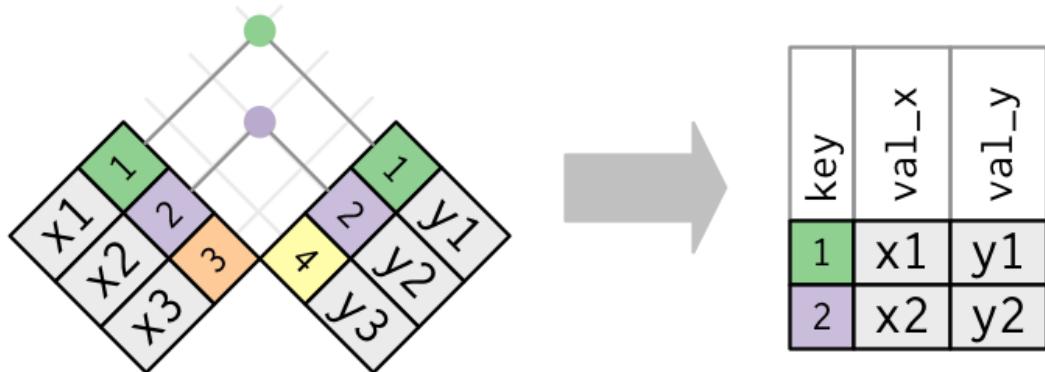


FIGURE 3.8: Diagram of inner join from *R for Data Science*.

3.7.2 Different “key” variable names

Say instead you are interested in the destinations of all domestic flights departing NYC in 2023, and you ask yourself questions like: “What cities are these airports in?”, or “Is “ORD” Orlando?”, or “Where is “FLL”?“.

The `airports` data frame contains the airport codes for each airport:

```
View(airports)
```

However, if you look at both the `airports` and `flights` data frames, you’ll find that the airport codes are in variables that have different names. In `airports` the airport code is in `faa`, whereas in `flights` the airport codes are in `origin` and `dest`. This fact is further highlighted in the visual representation of the relationships between these data frames in Figure 3.7.

In order to join these two data frames by airport code, our `inner_join()` operation will use the `by = c("dest" = "faa")` argument with modified code syntax allowing us to join two data frames where the key variable has a different name:

```
flights_with_airport_names <- flights |>
  inner_join(airports, by = c("dest" = "faa"))
View(flights_with_airport_names)
```

Let’s construct the chain of pipe operators `|>` that computes the number of flights from NYC to each destination, but also includes information about each destination airport:

```

named_dests <- flights |>
  group_by(dest) |>
  summarize(num_flights = n()) |>
  arrange(desc(num_flights)) |>
  inner_join(airports, by = c("dest" = "faa")) |>
  rename(airport_name = name)
named_dests

# A tibble: 114 x 9
  dest num_flights airport_name      lat   lon   alt   tz dst tzone
  <chr>     <int> <chr>           <dbl> <dbl> <dbl> <dbl> <chr> <chr>
1 BOS        19036 General Edward L~  42.4 -71.0    20    -5 A Amer~
2 ORD        18200 Chicago O'Hare I~  42.0 -87.9   672    -6 A Amer~
3 MCO        17756 Orlando Internat~  28.4 -81.3    96    -5 A Amer~
4 ATL        17570 Hartsfield Jacks~  33.6 -84.4   1026   -5 A Amer~
5 MIA        16076 Miami Internatio~  25.8 -80.3     8    -5 A Amer~
6 LAX        15968 Los Angeles Inte~  33.9 -118.    125    -8 A Amer~
7 FLL        14239 Fort Lauderdale ~  26.1 -80.2     9    -5 A Amer~
8 CLT        12866 Charlotte Dougl~  35.2 -80.9   748    -5 A Amer~
9 DFW        11675 Dallas Fort Wort~  32.9 -97.0   607    -6 A Amer~
10 SFO       11651 San Francisco In~  37.6 -122.    13    -8 A Amer~

# i 104 more rows

```

In case you didn't know, "ORD" is the airport code of Chicago O'Hare airport and "FLL" is the main airport in Fort Lauderdale, Florida, which can be seen in the `airport_name` variable.

3.7.3 Multiple “key” variables

Say instead we want to join two data frames by *multiple key variables*. For example, in Figure 3.7, we see that in order to join the `flights` and `weather` data frames, we need more than one key variable: `year`, `month`, `day`, `hour`, and `origin`. This is because the combination of these 5 variables act to uniquely identify each observational unit in the `weather` data frame: hourly weather recordings at each of the 3 NYC airports.

We achieve this by specifying a *vector* of key variables to join by using the `c()` function. Recall from Subsection 1.2.1 that `c()` is short for “combine” or “concatenate.”

```

flights_weather_joined <- flights |>
  inner_join(weather, by = c("year", "month", "day", "hour", "origin"))
View(flights_weather_joined)

```

Learning check

(LC3.13) Looking at Figure 3.7, when joining `flights` and `weather` (or, in other words, matching the hourly weather values with each flight), why do we need to join by all of `year`, `month`, `day`, `hour`, and `origin`, and not just `hour`?

(LC3.14) What surprises you about the top 10 destinations from NYC in 2023?

3.7.4 Normal forms

The data frames included in the `nycflights23` package are in a form that minimizes redundancy of data. For example, the `flights` data frame only saves the `carrier` code of the airline company; it does not include the actual name of the airline. For example, the first row of `flights` has `carrier` equal to `UA`, but it does not include the airline name of “United Air Lines Inc.”

The names of the airline companies are included in the `name` variable of the `airlines` data frame. In order to have the airline company name included in `flights`, we could join these two data frames as follows:

```
joined_flights <- flights |>
  inner_join(airlines, by = "carrier")
View(joined_flights)
```

We are capable of performing this join because each of the data frames have *keys* in common to relate one to another: the `carrier` variable in both the `flights` and `airlines` data frames. The *key* variable(s) that we base our joins on are often *identification variables* as we mentioned previously.

This is an important property of what’s known as *normal forms* of data. The process of decomposing data frames into less redundant tables without losing information is called *normalization*. More information is available on Wikipedia².

Both `dplyr` and SQL³ we mentioned in the introduction of this chapter use such *normal forms*. Given that they share such commonalities, once you learn either of these two tools, you can learn the other very easily.

²https://en.wikipedia.org/wiki/Database_normalization

³<https://en.wikipedia.org/wiki/SQL>

Learning check

(LC3.15) What are some advantages of data in normal forms? What are some disadvantages?

3.8 Other verbs

Here are some other useful data wrangling verbs:

- `select()` only a subset of variables/columns.
- `relocate()` variables/columns to a new position.
- `rename()` variables/columns to have new names.
- Return only the `top_n()` values of a variable.

3.8.1 `select` variables

Subset Variables (Columns)



FIGURE 3.9: Diagram of `select()` columns.

We've seen that the `flights` data frame in the `nycflights23` package contains 19 different variables. You can identify the names of these 19 variables by running the `glimpse()` function from the `dplyr` package:

```
glimpse(flights)
```

However, say you only need two of these 19 variables, say `carrier` and `flight`. You can `select()` these two variables:

```
flights |>  
  select(carrier, flight)
```

This function makes it easier to explore large datasets since it allows us to limit the scope to only those variables we care most about. For example, if we `select()` only a smaller number of variables as is shown in Figure 3.9, it will make viewing the dataset in RStudio’s spreadsheet viewer more digestible.

Let’s say instead you want to drop, or de-select, certain variables. For example, consider the variable `year` in the `flights` data frame. This variable isn’t quite a “variable” because it is always `2023` and hence doesn’t change. Say you want to remove this variable from the data frame. We can deselect `year` by using the `-` sign:

```
flights_no_year <- flights |> select(-year)
```

Another way of selecting columns/variables is by specifying a range of columns:

```
flight_arr_times <- flights |> select(month:day, arr_time:sched_arr_time)  
flight_arr_times
```

This will `select()` all columns between `month` and `day`, as well as between `arr_time` and `sched_arr_time`, and drop the rest.

The helper functions `starts_with()`, `ends_with()`, and `contains()` can be used to select variables/columns that match those conditions. As examples,

```
flights |> select(starts_with("a"))  
flights |> select(ends_with("delay"))  
flights |> select(contains("time"))
```

Lastly, the `select()` function can also be used to reorder columns when used with the `everything()` helper function. For example, suppose we want the `hour`, `minute`, and `time_hour` variables to appear immediately after the `year`, `month`, and `day` variables, while not discarding the rest of the variables. In the following code, `everything()` will pick up all remaining variables:

```
flights_reordered <- flights |>
  select(year, month, day, hour, minute, time_hour, everything())
glimpse(flights_reordered)
```

3.8.2 relocate variables

Another (usually shorter) way to reorder variables is by using the `relocate()` function. This function allows you to move variables to a new position in the data frame. For example, if we want to move the `hour`, `minute`, and `time_hour` variables to appear immediately after the `year`, `month`, and `day` variables, we can use the following code:

```
flights_relocate <- flights |>
  relocate(hour, minute, time_hour, .after = day)
glimpse(flights_relocate)
```

3.8.3 rename variables

One more useful function is `rename()`, which as you may have guessed changes the name of variables. Suppose we want to only focus on `dep_time` and `arr_time` and change `dep_time` and `arr_time` to be `departure_time` and `arrival_time` instead in the `flights_time` data frame:

```
flights_time_new <- flights |>
  select(dep_time, arr_time) |>
  rename(departure_time = dep_time, arrival_time = arr_time)
glimpse(flights_time_new)
```

Note that in this case we used a single `=` sign within the `rename()`. For example, `departure_time = dep_time` renames the `dep_time` variable to have the new name `departure_time`. This is because we are not testing for equality like we would using `==`. Instead we want to assign a new variable `departure_time` to have the same values as `dep_time` and then delete the variable `dep_time`. Note that new `dplyr` users often forget that the new variable name comes before the equal sign.

3.8.4 top_n values of a variable

We can also return the top `n` values of a variable using the `top_n()` function. For example, we can return a data frame of the top 10 destination airports using the example from Subsection 3.7.2. Observe that we set the number of values to return

to `n = 10` and `wt = num_flights` to indicate that we want the rows corresponding to the top 10 values of `num_flights`. See the help file for `top_n()` by running `?top_n` for more information.

```
named_dests |> top_n(n = 10, wt = num_flights)
```

Let's further `arrange()` these results in descending order of `num_flights`:

```
named_dests |>  
  top_n(n = 10, wt = num_flights) |>  
  arrange(desc(num_flights))
```

Learning check

(LC3.16) What are some ways to select all three of the `dest`, `air_time`, and `distance` variables from `flights`? Give the code showing how to do this in at least three different ways.

(LC3.17) How could one use `starts_with()`, `ends_with()`, and `contains()` to select columns from the `flights` data frame? Provide three different examples in total: one for `starts_with()`, one for `ends_with()`, and one for `contains()`.

(LC3.18) Why might we want to use the `select()` function on a data frame?

(LC3.19) Create a new data frame that shows the top 5 airports with the largest arrival delays from NYC in 2023.

3.9 Conclusion

3.9.1 Summary table

Let's recap our data wrangling verbs in Table 3.2. Using these verbs and the pipe `|>` operator from Section 3.1, you'll be able to write easily legible code to perform almost all the data wrangling and data transformation necessary for the rest of this book.

TABLE 3.2: Summary of data wrangling verbs

Verb	Data wrangling operation
filter()	Pick out a subset of rows
summarize()	Summarize many values to one using a summary statistic function like mean(), median(), etc.
group_by()	Add grouping structure to rows in data frame. Note this does not change values in data frame, rather only the meta-data
mutate()	Create new variables by mutating existing ones
arrange()	Arrange rows of a data variable in ascending (default) or descending order
inner_join()	Join/merge two data frames, matching rows by a key variable

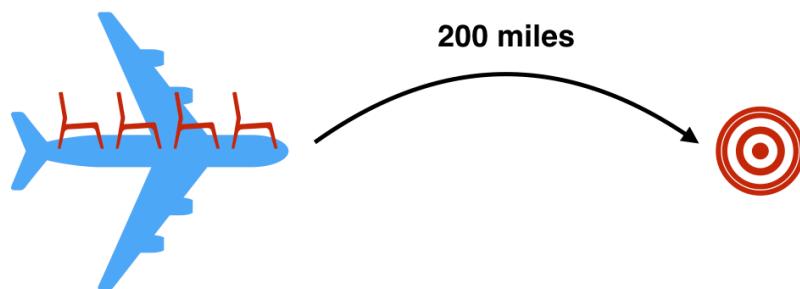
Learning check

(LC3.20) Let's now put your newly acquired data wrangling skills to the test!

An airline industry measure of a passenger airline's capacity is the available seat miles⁴, which is equal to the number of seats available multiplied by the number of miles or kilometers flown summed over all flights.

For example, let's consider the scenario in Figure 3.10. Since the airplane has 4 seats and it travels 200 miles, the available seat miles are $4 \times 200 = 800$.

Measure of airline capacity: Available Seat Miles



$$\begin{aligned} \text{Available Seat Miles} &= 4 \text{ seats} \times 200 \text{ miles} \\ &= 800 \text{ seat miles} \end{aligned}$$

FIGURE 3.10: Example of available seat miles for one flight.

⁴https://en.wikipedia.org/wiki/Available_seat_miles

Extending this idea, let's say an airline had 2 flights using a plane with 10 seats that flew 500 miles and 3 flights using a plane with 20 seats that flew 1000 miles, the available seat miles would be $2 \times 10 \times 500 + 3 \times 20 \times 1000 = 70,000$ seat miles.

Using the datasets included in the `nycflights23` package, compute the available seat miles for each airline sorted in descending order. After completing all the necessary data wrangling steps, the resulting data frame should have 16 rows (one for each airline) and 2 columns (airline name and available seat miles). Here are some hints:

1. **Crucial:** Unless you are very confident in what you are doing, it is worthwhile not starting to code right away. Rather, first sketch out on paper all the necessary data wrangling steps not using exact code, but rather high-level *pseudocode* that is informal yet detailed enough to articulate what you are doing. This way you won't confuse *what* you are trying to do (the algorithm) with *how* you are going to do it (writing `dplyr` code).
2. Take a close look at all the datasets using the `View()` function: `flights`, `weather`, `planes`, `airports`, and `airlines` to identify which variables are necessary to compute available seat miles.
3. Figure 3.7 showing how the various datasets can be joined will also be useful.
4. Consider the data wrangling verbs in Table 3.2 as your toolbox!

3.9.2 Additional resources

Solutions to all *Learning checks* can be found online in Appendix D⁵.

An R script file of all R code used in this chapter is available at <https://www.moderndive.com/scripts/03-wrangling.R>.

In the online Appendix C⁶, we provide a page of data wrangling “tips and tricks” consisting of the most common data wrangling questions we’ve encountered in student projects (shout out to Dr. Jenny Smetzer⁷ for her work setting this up!):

- Dealing with missing values
- Reordering bars in a barplot
- Showing money on an axis
- Changing values inside cells
- Converting a numerical variable to a categorical one
- Computing proportions
- Dealing with %, commas, and \$

⁵<https://moderndive.com/D-appendixD.html>

⁶<https://moderndive.com/C-appendixC.html>

⁷<https://www.scsparkscience.org/fellow/jennifer-smetzer/>

However to provide a tips and tricks page covering all possible data wrangling questions would be too long to be useful! If you want to further unlock the power of the `dplyr` package for data wrangling, we suggest that you check out RStudio’s “Data Transformation with `dplyr`” cheatsheet. This cheatsheet summarizes much more than what we’ve discussed in this chapter, in particular more intermediate level and advanced data wrangling functions, while providing quick and easy-to-read visual descriptions. In fact, many of the diagrams illustrating data wrangling operations in this chapter, such as Figure 3.1 on `filter()`, originate from this cheatsheet.

In the current version of RStudio in late 2019, you can access this cheatsheet by going to the RStudio Menu Bar -> Help -> Cheatsheets -> “Data Transformation with `dplyr`.”

On top of the data wrangling verbs and examples we presented in this section, if you’d like to see more examples of using the `dplyr` package for data wrangling, check out Chapter 5⁸ of *R for Data Science* (Grollemund and Wickham, 2017).

3.9.3 What’s to come?

So far in this book, we’ve explored, visualized, and wrangled data saved in data frames. These data frames were saved in a spreadsheet-like format: in a rectangular shape with a certain number of rows corresponding to observations and a certain number of columns corresponding to variables describing these observations.

We’ll see in the upcoming Chapter 4 that there are actually two ways to represent data in spreadsheet-type rectangular format: (1) “wide” format and (2) “tall/narrow” format. The tall/narrow format is also known as “tidy” format in R user circles. While the distinction between “tidy” and non-“tidy” formatted data is subtle, it has immense implications for our data science work. This is because almost all the packages used in this book, including the `ggplot2` package for data visualization and the `dplyr` package for data wrangling, all assume that all data frames are in “tidy” format.

Furthermore, up until now we’ve only explored, visualized, and wrangled data saved within R packages. But what if you want to analyze data that you have saved in a Microsoft Excel, a Google Sheets, or a “Comma-Separated Values” (CSV) file? In Section 4.1, we’ll show you how to import this data into R using the `readr` package.

⁸<http://r4ds.had.co.nz/transform.html>

4

Data Importing and “Tidy” Data

In Subsection 1.2.1, we introduced the concept of a data frame in R: a rectangular spreadsheet-like representation of data where the rows correspond to observations and the columns correspond to variables describing each observation. In Section 1.4, we started exploring our first data frame: the `flights` data frame included in the `nycflights23` package. In Chapter 2, we created visualizations based on the data included in `flights` and other data frames such as `weather`. In Chapter 3, we learned how to take existing data frames and transform/modify them to suit our ends.

In this final chapter of the “Data Science with `tidyverse`” portion of the book, we extend some of these ideas by discussing a type of data formatting called “tidy” data. You will see that having data stored in “tidy” format is about more than just what the everyday definition of the term “tidy” might suggest: having your data “neatly organized.” Instead, we define the term “tidy” as it’s used by data scientists who use R, outlining a set of rules by which data is saved.

Knowledge of this type of data formatting was not necessary for our treatment of data visualization in Chapter 2 and data wrangling in Chapter 3. This is because all the data used were already in “tidy” format. In this chapter, we’ll now see that this format is essential to using the tools we covered up until now. Furthermore, it will also be useful for all subsequent chapters in this book when we cover regression and statistical inference. First, however, we’ll show you how to import spreadsheet data in R.

Needed packages

Let’s load all the packages needed for this chapter (this assumes you’ve already installed them). If needed, read Section 1.3 for information on how to install and load R packages.

```
library(dplyr)
library(ggplot2)
library(readr)
library(tidyr)
library(nycflights23)
library(fivethirtyeight)
```

4.1 Importing data

Up to this point, we’ve almost entirely used data stored inside of an R package. Say instead you have your own data saved on your computer or somewhere online. How can you analyze this data in R? Spreadsheet data is often saved in one of the following three formats:

First, a *Comma Separated Values* .csv file. You can think of a .csv file as a bare-bones spreadsheet where:

- Each line in the file corresponds to one row of data/one observation.
- Values for each line are separated with commas. In other words, the values of different variables are separated by commas in each row.
- The first line is often, but not always, a *header* row indicating the names of the columns/variables.

Second, an Excel .xlsx spreadsheet file. This format is based on Microsoft’s proprietary Excel software. As opposed to bare-bones .csv files, .xlsx Excel files contain a lot of meta-data (data about data). Recall we saw a previous example of meta-data in Section 3.4 when adding “group structure” meta-data to a data frame by using the `group_by()` verb. Some examples of Excel spreadsheet meta-data include the use of bold and italic fonts, colored cells, different column widths, and formula macros.

Third, a Google Sheets¹ file, which is a “cloud” or online-based way to work with a spreadsheet. Google Sheets allows you to download your data in both comma separated values .csv and Excel .xlsx formats. One way to import Google Sheets data in R is to go to the Google Sheets menu bar -> File -> Download as -> Select “Microsoft Excel” or “Comma-separated values” and then load that data into R. A more advanced way to import Google Sheets data in R is by using the `googlesheets4`² package, a method we leave to a more advanced data science book.

We’ll cover two methods for importing .csv and .xlsx spreadsheet data in R: one using the console and the other using RStudio’s graphical user interface, abbreviated as “GUI.”

4.1.1 Using the console

First, let’s import a Comma Separated Values .csv file that exists on the internet. The .csv file `dem_score.csv` contains ratings of the level of democracy in different countries spanning 1952 to 1992 and is accessible at https://moderndive.com/data/dem_score.csv. Let’s use the `read_csv()` function from the `readr` (Wickham et al., 2024b) package to read it off the web, import it into R, and save it in a data frame called `dem_score`.

¹<https://www.google.com/sheets/about/>

²<https://googlesheets4.tidyverse.org/>

```
library(readr)
dem_score <- read_csv("https://moderndive.com/data/dem_score.csv")
dem_score
```

```
# A tibble: 96 x 10
  country `1952` `1957` `1962` `1967` `1972` `1977` `1982` `1987` `1992`
  <chr>   <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 Albania     -9     -9     -9     -9     -9     -9     -9     -9      5
2 Argentina    -9     -1     -1     -9     -9     -9     -8      8      7
3 Armenia      -9     -7     -7     -7     -7     -7     -7     -7      7
4 Australia     10     10     10     10     10     10     10     10     10
5 Austria       10     10     10     10     10     10     10     10     10
6 Azerbaijan   -9     -7     -7     -7     -7     -7     -7     -7      1
7 Belarus       -9     -7     -7     -7     -7     -7     -7     -7      7
8 Belgium       10     10     10     10     10     10     10     10     10
9 Bhutan        -10    -10    -10    -10    -10    -10    -10    -10    -10
10 Bolivia      -4     -3     -3     -4     -7     -7      8      9      9
# i 86 more rows
```

In this `dem_score` data frame, the minimum value of `-10` corresponds to a highly autocratic nation, whereas a value of `10` corresponds to a highly democratic nation. Note also that backticks surround the different variable names. Variable names in R by default are not allowed to start with a number nor include spaces, but we can get around this fact by surrounding the column name with backticks. We'll revisit the `dem_score` data frame in a case study in the upcoming Section 4.3.

Note that the `read_csv()` function included in the `readr` package is different than the `read.csv()` function that comes installed with R. While the difference in the names might seem trivial (an `_` instead of a `.`), the `read_csv()` function is, in our opinion, easier to use since it can more easily read data off the web and generally imports data at a much faster speed. Furthermore, the `read_csv()` function included in the `readr` saves data frames as `tibbles` by default.

4.1.2 Using RStudio's interface

Let's read in the exact same data, but this time from an Excel file saved on your computer. Furthermore, we'll do this using RStudio's graphical interface instead of running `read_csv()` in the console. First, download the Excel file `dem_score.xlsx` by going to https://moderndive.com/data/dem_score.xlsx, then

1. Go to the Files pane of RStudio.
2. Navigate to the directory (i.e., folder on your computer) where the downloaded `dem_score.xlsx` Excel file is saved. For example, this might be in your Downloads folder.

3. Click on `dem_score.xlsx`.
4. Click “Import Dataset...”

At this point, you should see a screen pop-up like in Figure 4.1. After clicking on the “Import” button on the bottom right of Figure 4.1, RStudio will save this spreadsheet’s data in a data frame called `dem_score` and display its contents in the spreadsheet viewer.

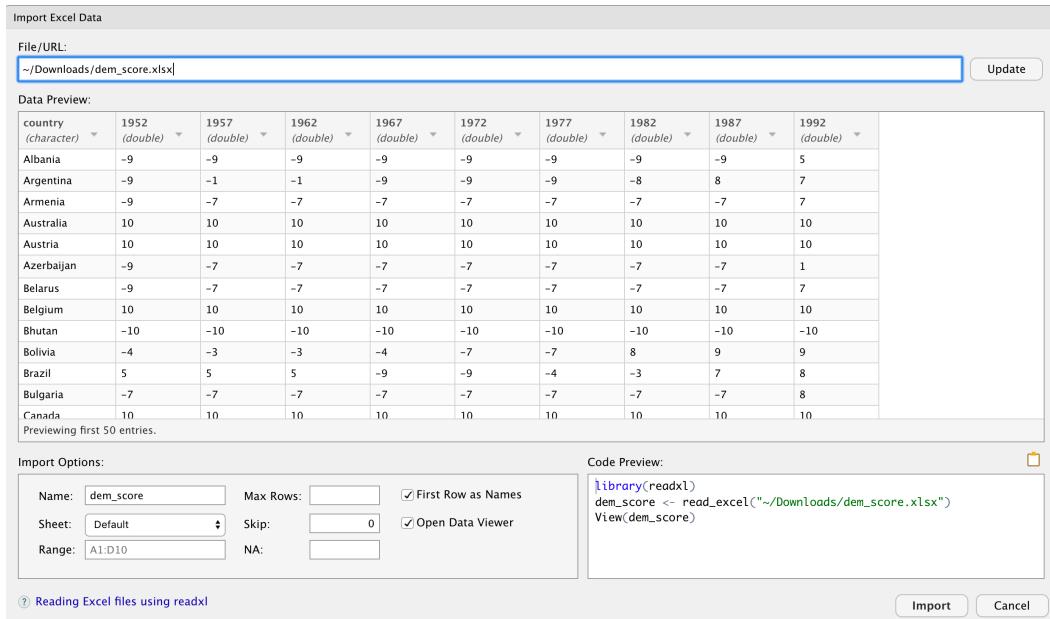


FIGURE 4.1: Importing an Excel file to R.

Furthermore, note the “Code Preview” block in the bottom right of Figure 4.1. You can copy and paste this code to reload your data again later programmatically, instead of repeating this manual point-and-click process.

4.2 “Tidy” data

Let’s now switch gears and learn about the concept of “tidy” data format with a motivating example from the `fivethirtyeight` package. The `fivethirtyeight` package (Kim et al., 2021) provides access to the datasets used in many articles published by the data journalism website, FiveThirtyEight.com³. For a complete list of all 128 datasets included in the `fivethirtyeight` package, check out the package webpage by going to: <https://fivethirtyeight-r.netlify.app/articles/fivethirtyeight.html>.

³<https://fivethirtyeight.com/>

Let’s focus our attention on the `drinks` data frame and look at its first 5 rows:

```
# A tibble: 5 × 5
  country    beer_servings spirit_servings wine_servings
  <chr>        <int>          <int>          <int>
1 Afghanistan      0            0            0
2 Albania         89           132           54
3 Algeria          25            0            14
4 Andorra          245           138           312
5 Angola           217           57            45
# i 1 more variable: total_litres_of_pure_alcohol <dbl>
```

After reading the help file by running `?drinks`, you’ll see that `drinks` is a data frame containing results from a survey of the average number of servings of beer, spirits, and wine consumed in 193 countries. This data was originally reported on FiveThirtyEight.com in Mona Chalabi’s article: “Dear Mona Followup: Where Do People Drink The Most Beer, Wine And Spirits?”⁴.

Let’s apply some of the data wrangling verbs we learned in Chapter 3 on the `drinks` data frame:

1. `filter()` the `drinks` data frame to only consider 4 countries: the United States, China, Italy, and Saudi Arabia, *then*
2. `select()` all columns except `total_litres_of_pure_alcohol` by using the `-` sign, *then*
3. `rename()` the variables `beer_servings`, `spirit_servings`, and `wine_servings` to `beer`, `spirit`, and `wine`, respectively.

and save the resulting data frame in `drinks_smaller`:

```
drinks_smaller <- drinks |>
  filter(country %in% c("USA", "China", "Italy", "Saudi Arabia")) |>
  select(-total_litres_of_pure_alcohol) |>
  rename(beer = beer_servings, spirit = spirit_servings, wine = wine_servings)
drinks_smaller
```

```
# A tibble: 4 × 4
  country      beer   spirit   wine
  <chr>       <int>   <int>   <int>
1 China         79     192      8
2 Italy         85     42      237
```

⁴<https://fivethirtyeight.com/features/dear-mona-followup-where-do-people-drink-the-most-beer-wine-and-spirits/>

3 Saudi Arabia	0	5	0
4 USA	249	158	84

Let’s now ask ourselves a question: “Using the `drinks_smaller` data frame, how would we create the side-by-side barplot in Figure 4.2?”. Recall we saw barplots displaying two categorical variables in Subsection 2.8.3.

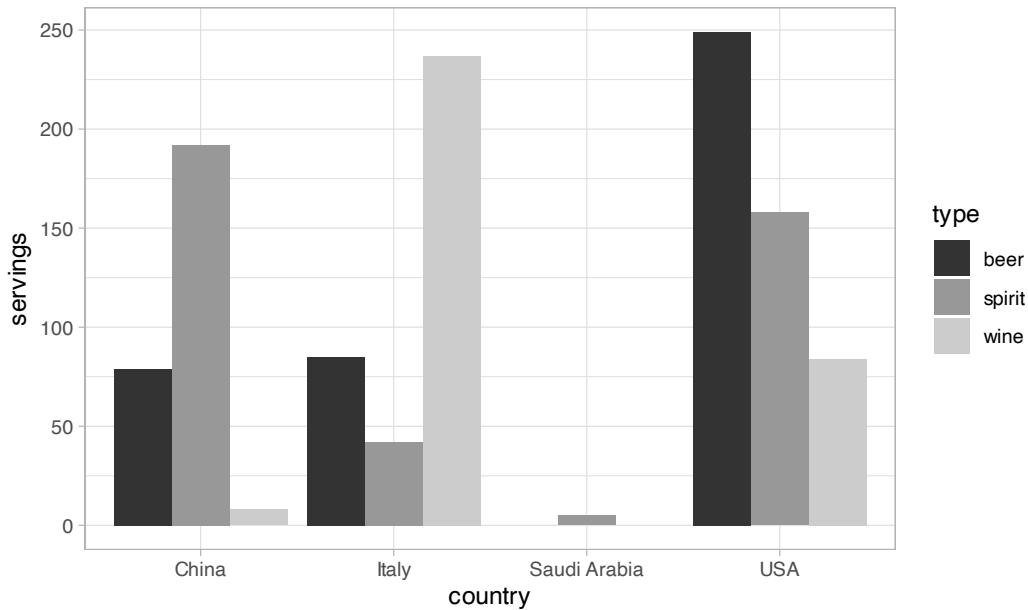


FIGURE 4.2: Comparing alcohol consumption in 4 countries.

Let’s break down the grammar of graphics we introduced in Section 2.1:

1. The categorical variable `country` with four levels (China, Italy, Saudi Arabia, USA) would have to be mapped to the x-position of the bars.
2. The numerical variable `servings` would have to be mapped to the y-position of the bars (the height of the bars).
3. The categorical variable `type` with three levels (beer, spirit, wine) would have to be mapped to the fill color of the bars.

Observe that `drinks_smaller` has three separate variables `beer`, `spirit`, and `wine`. In order to use the `ggplot()` function to recreate the barplot in Figure 4.2 however, we need a *single variable* `type` with three possible values: `beer`, `spirit`, and `wine`. We could then map this `type` variable to the `fill` aesthetic of our plot. In other words, to recreate the barplot in Figure 4.2, our data frame would have to look like this:

`drinks_smaller_tidy`

```
# A tibble: 12 x 3
  country     type   servings
  <chr>      <chr>    <int>
1 China       beer      79
2 Italy        beer      85
3 Saudi Arabia beer       0
4 USA          beer     249
5 China        spirit    192
6 Italy         spirit     42
7 Saudi Arabia spirit      5
8 USA           spirit    158
9 China         wine      8
10 Italy        wine     237
11 Saudi Arabia wine       0
12 USA          wine     84
```

Observe that while `drinks_smaller` and `drinks_smaller_tidy` are both rectangular in shape and contain the same 12 numerical values (3 alcohol types by 4 countries), they are formatted differently. `drinks_smaller` is formatted in what's known as "wide"⁵ format, whereas `drinks_smaller_tidy` is formatted in what's known as "long/narrow"⁶ format.

In the context of doing data science in R, long/narrow format is also known as "tidy" format. In order to use the `ggplot2` and `dplyr` packages for data visualization and data wrangling, your input data frames *must* be in "tidy" format. Thus, all non-"tidy" data must be converted to "tidy" format first. Before we convert non-"tidy" data frames like `drinks_smaller` to "tidy" data frames like `drinks_smaller_tidy`, let's define "tidy" data.

4.2.1 Definition of “tidy” data

You have surely heard the word "tidy" in your life:

- “Tidy up your room!”
- “Write your homework in a tidy way so it is easier to provide feedback.”
- Marie Kondo's best-selling book, *The Life-Changing Magic of Tidying Up: The Japanese Art of Decluttering and Organizing*⁷, and Netflix TV series *Tidying Up with Marie Kondo*⁸.

⁵https://en.wikipedia.org/wiki/Wide_and_narrow_data

⁶https://en.wikipedia.org/wiki/Wide_and_narrow_data#Narrow

⁷<https://www.powells.com/book/-9781607747307>

⁸<https://www.netflix.com/title/80209379>

- “I am not by any stretch of the imagination a tidy person, and the piles of unread books on the coffee table and by my bed have a plaintive, pleading quality to me - ‘Read me, please!’ ” - Linda Grant

What does it mean for your data to be “tidy”? While “tidy” has a clear English meaning of “organized,” the word “tidy” in data science using R means that your data follows a standardized format. We will follow Hadley Wickham’s definition of “tidy” data (Wickham, 2014) shown also in Figure 4.3:

A *dataset* is a collection of values, usually either numbers (if quantitative) or strings AKA text data (if qualitative/categorical). Values are organised in two ways. Every value belongs to a variable and an observation. A variable contains all values that measure the same underlying attribute (like height, temperature, duration) across units. An observation contains all values measured on the same unit (like a person, or a day, or a city) across attributes.

“Tidy” data is a standard way of mapping the meaning of a dataset to its structure. A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types. In *tidy data*:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

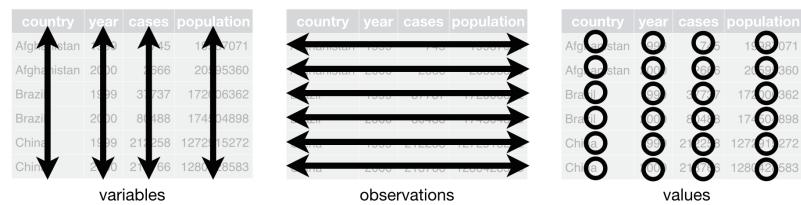


FIGURE 4.3: Tidy data graphic from *R for Data Science*.

For example, say you have the following table of stock prices in Table 4.1:

Although the data are neatly organized in a rectangular spreadsheet-type format, they do not follow the definition of data in “tidy” format. While there are three variables corresponding to three unique pieces of information (date, stock name, and stock price), there are not three columns. In “tidy” data format, each variable should be its own column, as shown in Table 4.2. Notice that both tables present the same information, but in different formats.

TABLE 4.1: Stock prices (non-tidy format)

Date	Boeing stock price	Amazon stock price	Google stock price
2009-01-01	\$173.55	\$174.90	\$174.34
2009-01-02	\$172.61	\$171.42	\$170.04

TABLE 4.2: Stock prices (tidy format)

Date	Stock Name	Stock Price
2009-01-01	Boeing	\$173.55
2009-01-01	Amazon	\$174.90
2009-01-01	Google	\$174.34
2009-01-02	Boeing	\$172.61
2009-01-02	Amazon	\$171.42
2009-01-02	Google	\$170.04

Now we have the requisite three columns Date, Stock Name, and Stock Price. On the other hand, consider the data in Table 4.3.

TABLE 4.3: Example of tidy data

Date	Boeing Price	Weather
2009-01-01	\$173.55	Sunny
2009-01-02	\$172.61	Overcast

In this case, even though the variable “Boeing Price” occurs just like in our non-“tidy” data in Table 4.1, the data *is* “tidy” since there are three variables corresponding to three unique pieces of information: Date, Boeing price, and the Weather that particular day.

Learning check

(LC4.1) What are common characteristics of “tidy” data frames?

(LC4.2) What makes “tidy” data frames useful for organizing data?

4.2.2 Converting to “tidy” data

In this book so far, you’ve only seen data frames that were already in “tidy” format. Furthermore, for the rest of this book, you’ll mostly only see data frames that are already in “tidy” format as well. This is not always the case however with all datasets in the world. If your original data frame is in wide (non-“tidy”) format and you would

like to use the `ggplot2` or `dplyr` packages, you will first have to convert it to “tidy” format. To do so, we recommend using the `pivot_longer()` function in the `tidyverse` package ([Wickham et al., 2024c](#)).

Going back to our `drinks_smaller` data frame from earlier:

```
drinks_smaller
```

```
# A tibble: 4 × 4
  country      beer  spirit   wine
  <chr>     <int>  <int>  <int>
1 China        79    192     8
2 Italy        85     42    237
3 Saudi Arabia  0      5     0
4 USA         249    158    84
```

We convert it to “tidy” format by using the `pivot_longer()` function from the `tidyverse` package as follows:

```
drinks_smaller_tidy <- drinks_smaller |>
  pivot_longer(names_to = "type",
              values_to = "servings",
              cols = -country)
drinks_smaller_tidy
```

```
# A tibble: 12 × 3
  country     type   servings
  <chr>      <chr>    <int>
1 China      beer       79
2 China      spirit     192
3 China      wine        8
4 Italy      beer       85
5 Italy      spirit      42
6 Italy      wine      237
7 Saudi Arabia beer       0
8 Saudi Arabia spirit     5
9 Saudi Arabia wine       0
10 USA      beer      249
11 USA      spirit     158
12 USA      wine       84
```

We set the arguments to `pivot_longer()` as follows:

1. `names_to` here corresponds to the name of the variable in the new “tidy”/long data frame that will contain the *column names* of the original data. Observe how we set `names_to = "type"`. In the resulting `drinks_smaller_tidy`, the column `type` contains the three types of alcohol `beer`, `spirit`, and `wine`. Since `type` is a variable name that doesn’t appear in `drinks_smaller`, we use quotation marks around it. You’ll receive an error if you just use `names_to = type` here.
2. `values_to` here is the name of the variable in the new “tidy” data frame that will contain the *values* of the original data. Observe how we set `values_to = "servings"` since each of the numeric values in each of the `beer`, `wine`, and `spirit` columns of the `drinks_smaller` data corresponds to a value of `servings`. In the resulting `drinks_smaller_tidy`, the column `servings` contains the $4 \times 3 = 12$ numerical values. Note again that `servings` doesn’t appear as a variable in `drinks_smaller` so it again needs quotation marks around it for the `values_to` argument.
3. The third argument `cols` is the columns in the `drinks_smaller` data frame you either want to or don’t want to “tidy.” Observe how we set this to `-country` indicating that we don’t want to “tidy” the `country` variable in `drinks_smaller` and rather only `beer`, `spirit`, and `wine`. Since `country` is a column that appears in `drinks_smaller` we don’t put quotation marks around it.

The third argument here of `cols` is a little nuanced, so let’s consider code that’s written slightly differently but that produces the same output:

```
drinks_smaller |>
  pivot_longer(names_to = "type",
               values_to = "servings",
               cols = c(beer, spirit, wine))
```

Note that the third argument now specifies which columns we want to “tidy” with `c(beer, spirit, wine)`, instead of the columns we don’t want to “tidy” using `-country`. We use the `c()` function to create a vector of the columns in `drinks_smaller` that we’d like to “tidy.” Note that since these three columns appear one after another in the `drinks_smaller` data frame, we could also do the following for the `cols` argument:

```
drinks_smaller |>
  pivot_longer(names_to = "type",
               values_to = "servings",
               cols = beer:wine)
```

With our `drinks_smaller_tidy` “tidy” formatted data frame, we can now produce the barplot you saw in Figure 4.2 using `geom_col()`. This is done in Figure 4.4. Recall from

Section 2.8 on barplots that we use `geom_col()` and not `geom_bar()`, since we would like to map the “pre-counted” `servings` variable to the y-aesthetic of the bars.

```
ggplot(drinks_smaller_tidy, aes(x = country, y = servings, fill = type)) +
  geom_col(position = "dodge")
```

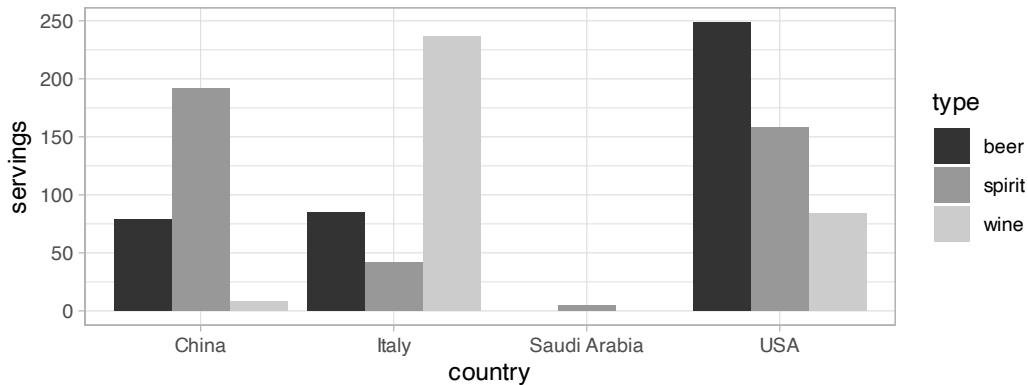


FIGURE 4.4: Comparing alcohol consumption in 4 countries using `geom_col()`.

Converting “wide” format data to “tidy” format often confuses new R users. The only way to learn to get comfortable with the `pivot_longer()` function is with practice, practice, and more practice using different datasets. For example, run `?pivot_longer` and look at the examples in the bottom of the help file. We’ll show another example of using `pivot_longer()` to convert a “wide” formatted data frame to “tidy” format in Section 4.3.

If however you want to convert a “tidy” data frame to “wide” format, you will need to use the `pivot_wider()` function instead. Run `?pivot_wider` and look at the examples in the bottom of the help file for examples.

You can also view examples of both `pivot_longer()` and `pivot_wider()` on the tidyverse.org⁹ webpage. There’s a nice example to check out the different functions available for data tidying and a case study using data from the World Health Organization on that webpage. Furthermore, each week the R4DS Online Learning Community posts a dataset in the weekly #TidyTuesday event¹⁰ that might serve as a nice place for you to find other data to explore and transform.

Learning check

⁹<https://tidyverse.tidyverse.org/dev/articles/pivot.html#pew>

¹⁰<https://github.com/rfordatascience/tidytuesday>

(LC4.3) Take a look at the `airline_safety` data frame included in the `fivethirtyeight` data package. Run the following:

```
airline_safety
```

After reading the help file by running `?airline_safety`, we see that `airline_safety` is a data frame containing information on different airline companies’ safety records. This data was originally reported on the data journalism website, FiveThirtyEight.com, in Nate Silver’s article, “Should Travelers Avoid Flying Airlines That Have Had Crashes in the Past?”¹¹. Let’s only consider the variables `airline` and those relating to fatalities for simplicity:

```
airline_safety_smaller <- airline_safety |>
  select(airline, starts_with("fatalities"))
airline_safety_smaller
```

```
# A tibble: 56 x 3
  airline          fatalities_85_99 fatalities_00_14
  <chr>              <int>            <int>
1 Aer Lingus           0                0
2 Aeroflot            128               88
3 Aerolineas Argentinas    0                0
4 Aeromexico          64                0
5 Air Canada           0                0
6 Air France           79              337
7 Air India            329              158
8 Air New Zealand      0                 7
9 Alaska Airlines      0                88
10 Alitalia            50                0
# i 46 more rows
```

This data frame is not in “tidy” format. How would you convert this data frame to be in “tidy” format, in particular so that it has a variable `fatalities_years` indicating the incident year and a variable `count` of the fatality counts?

¹¹<https://fivethirtyeight.com/features/should-travelers-avoid-flying-airlines-that-have-had-crashes-in-the-past/>

4.2.3 nycflights23 package

Recall the `nycflights23` package we introduced in Section 1.4 with data about all domestic flights departing from New York City in 2023. Let’s revisit the `flights` data frame by running `View(flights)`. We saw that `flights` has a rectangular shape, with each of its 435,352 rows corresponding to a flight and each of its 22 columns corresponding to different characteristics/measurements of each flight. This satisfied the first two criteria of the definition of “tidy” data from Subsection 4.2.1: that “Each variable forms a column” and “Each observation forms a row.” But what about the third property of “tidy” data that “Each type of observational unit forms a table”?

Recall that we saw in Subsection 1.4.3 that the observational unit for the `flights` data frame is an individual flight. In other words, the rows of the `flights` data frame refer to characteristics/measurements of individual flights. Also included in the `nycflights23` package are other data frames with their rows representing different observational units (Ismay et al., 2024):

- `airlines`: translation between two letter IATA carrier codes and airline company names (14 in total). The observational unit is an airline company.
- `planes`: aircraft information about each of 4,840 planes used, i.e., the observational unit is an aircraft.
- `weather`: hourly meteorological data (about 8,735 observations) for each of the three NYC airports, i.e., the observational unit is an hourly measurement of weather at one of the three airports.
- `airports`: airport names and locations. The observational unit is an airport.

The organization of the information into these five data frames follows the third “tidy” data property: observations corresponding to the same observational unit should be saved in the same table, i.e., data frame. You could think of this property as the old English expression: “birds of a feather flock together.”

4.3 Case study: Democracy in Guatemala

In this section, we’ll show you another example of how to convert a data frame that isn’t in “tidy” format (“wide” format) to a data frame that is in “tidy” format (“long/narrow” format). We’ll do this using the `pivot_longer()` function from the `tidyverse` package again.

Furthermore, we’ll make use of functions from the `ggplot2` and `dplyr` packages to produce a *time-series plot* showing how the democracy scores have changed over the 40 years from 1952 to 1992 for Guatemala. Recall that we saw time-series plots in Section 2.4 on creating linegraphs using `geom_line()`.

Let's use the `dem_score` data frame we imported in Section 4.1, but focus on only data corresponding to Guatemala.

```
guat_dem <- dem_score |>
  filter(country == "Guatemala")
guat_dem
```

	country	'1952'	'1957'	'1962'	'1967'	'1972'	'1977'	'1982'	'1987'	'1992'
<chr>		<dbl>								
1	Guatemala	2	-6	-5	3	1	-3	-7	3	3

Let's lay out the grammar of graphics we saw in Section 2.1.

First we know we need to set `data = guat_dem` and use a `geom_line()` layer, but what is the aesthetic mapping of variables? We'd like to see how the democracy score has changed over the years, so we need to map:

- year to the x-position aesthetic and
- `democracy_score` to the y-position aesthetic

Now we are stuck in a predicament, much like with our `drinks_smaller` example in Section 4.2. We see that we have a variable named `country`, but its only value is "Guatemala". We have other variables denoted by different year values. Unfortunately, the `guat_dem` data frame is not "tidy" and hence is not in the appropriate format to apply the grammar of graphics, and thus we cannot use the `ggplot2` package just yet.

We need to take the values of the columns corresponding to years in `guat_dem` and convert them into a new "names" variable called `year`. Furthermore, we need to take the democracy score values in the inside of the data frame and turn them into a new "values" variable called `democracy_score`. Our resulting data frame will have three columns: `country`, `year`, and `democracy_score`. Recall that the `pivot_longer()` function in the `tidyverse` package does this for us:

```
guat_dem_tidy <- guat_dem |>
  pivot_longer(names_to = "year",
              values_to = "democracy_score",
              cols = -country,
              names_transform = list(year = as.integer))
guat_dem_tidy
```

```
# A tibble: 9 x 3
  country    year democracy_score
  <chr>     <int>         <dbl>
1 Guatemala  1952            2
2 Guatemala  1957           -6
3 Guatemala  1962           -5
4 Guatemala  1967            3
5 Guatemala  1972            1
6 Guatemala  1977           -3
7 Guatemala  1982           -7
8 Guatemala  1987            3
9 Guatemala  1992            3
```

We set the arguments to `pivot_longer()` as follows:

1. `names_to` is the name of the variable in the new “tidy” data frame that will contain the *column names* of the original data. Observe how we set `names_to = "year"`. In the resulting `guat_dem_tidy`, the column `year` contains the years where Guatemala’s democracy scores were measured.
2. `values_to` is the name of the variable in the new “tidy” data frame that will contain the *values* of the original data. Observe how we set `values_to = "democracy_score"`. In the resulting `guat_dem_tidy` the column `democracy_score` contains the $1 \times 9 = 9$ democracy scores as numeric values.
3. The third argument is the columns you either want to or don’t want to “tidy.” Observe how we set this to `cols = -country` indicating that we don’t want to “tidy” the `country` variable in `guat_dem` and rather only variables 1952 through 1992.
4. The last argument of `names_transform` tells R what type of variable `year` should be set to. Without specifying that it is an `integer` as we’ve done here, `pivot_longer()` will set it to be a character value by default.

We can now create the time-series plot in Figure 4.5 to visualize how democracy scores in Guatemala have changed from 1952 to 1992 using a `geom_line()`. Furthermore, we’ll use the `labs()` function in the `ggplot2` package to add informative labels to all the `aes()`thetic attributes of our plot, in this case the `x` and `y` positions.

```
ggplot(guat_dem_tidy, aes(x = year, y = democracy_score)) +
  geom_line() +
  labs(x = "Year", y = "Democracy Score")
```

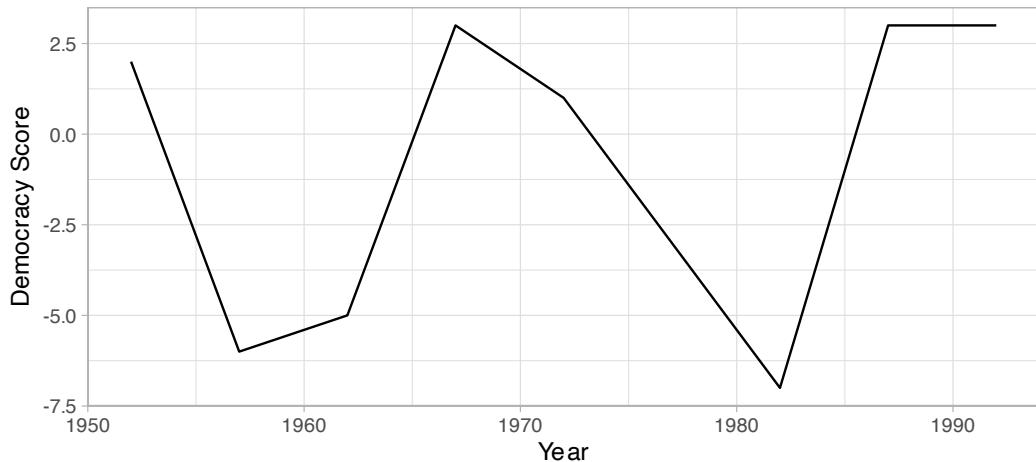


FIGURE 4.5: Democracy scores in Guatemala 1952-1992.

Note that if we forgot to include the `names_transform` argument specifying that `year` was not of character format, we would have gotten an error here since `geom_line()` wouldn't have known how to sort the character values in `year` in the right order.

Learning check

(LC4.4) Convert the `dem_score` data frame into a “tidy” data frame and assign the name of `dem_score_tidy` to the resulting long-formatted data frame.

(LC4.5) Read in the life expectancy data stored at https://moderndive.com/data/le_mess.csv and convert it to a “tidy” data frame.

4.4 tidyverse package

Notice at the beginning of the chapter we loaded the following four packages, which are among four of the most frequently used R packages for data science:

```
library(ggplot2)
library(dplyr)
library(readr)
library(tidyr)
```

Recall that `ggplot2` is for data visualization, `dplyr` is for data wrangling, `readr` is for importing spreadsheet data into R, and `tidyR` is for converting data to “tidy” format. There is a much quicker way to load these packages than by individually loading them: by installing and loading the `tidyverse` package. The `tidyverse` package acts as an “umbrella” package whereby installing/loading it will install/load multiple packages at once for you.

After installing the `tidyverse` package as you would a normal package as seen in Section 1.3, running:

```
library(tidyverse)
```

would be the same as running:

```
library(ggplot2)
library(dplyr)
library(readr)
library(tidyR)
library(purrr)
library(tibble)
library(stringr)
library(forcats)
```

The `purrr`, `tibble`, `stringr`, and `forcats` are left for a more advanced book; check out *R for Data Science*¹² to learn about these packages.

For the remainder of this book, we’ll start every chapter by running `library(tidyverse)`, instead of loading the various component packages individually. The `tidyverse` “umbrella” package gets its name from the fact that all the functions in all its packages are designed to have common inputs and outputs: data frames are in “tidy” format. This standardization of input and output data frames makes transitions between different functions in the different packages as seamless as possible. For more information, check out the tidyverse.org¹³ webpage for the package.

4.5 Conclusion

4.5.1 Additional resources

Solutions to all *Learning checks* can be found online in Appendix D¹⁴.

¹²<http://r4ds.had.co.nz/>

¹³<https://www.tidyverse.org/>

¹⁴<https://moderndive.com/D-appendixD.html>

An R script file of all R code used in this chapter is available at <https://www.moderndive.com/scripts/04-tidy.R>.

If you want to learn more about using the `readr` and `tidyverse` package, we suggest that you check out RStudio’s “Data Import Cheat Sheet.” In the current version of RStudio in mid 2024, you can access this cheatsheet by going to the RStudio Menu Bar -> Help -> Cheat Sheets -> “Browse Cheat Sheets...” -> Scroll down the page to the “Data import with reader, `readxl`, and `googlesheets4...`” for information on using the `readr`, `readxl` and `googlesheets4` packages to import data, and the “Data tidying with `tidyverse` cheatsheet” for information on using the `tidyverse` package to “tidy” data.

4.5.2 What’s to come?

Congratulations! You’ve completed the “Data Science with `tidyverse`” portion of this book. We’ll now move to the “Statistical modeling with `moderndive`” portion of this book in Chapters 5 and 6, where you’ll leverage your data visualization and wrangling skills to model relationships between different variables in data frames.

However, we’re going to leave Chapter 10 on “Inference for Regression” until after we’ve covered statistical inference in Chapters 7, 8, and 9. Onwards and upwards into Statistical/Data Modeling as shown in Figure 4.6!

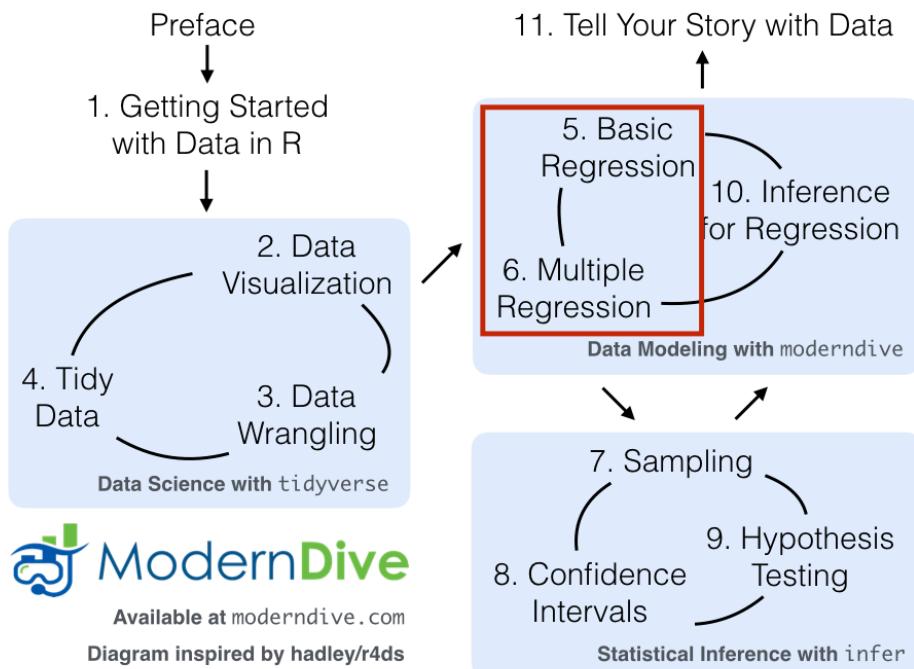


FIGURE 4.6: *ModernDive* flowchart - on to Part II!

Part II

Statistical Modeling with moderndive

5

Simple Linear Regression

We have introduced data visualization in Chapter 2, data wrangling in Chapter 3, and data importing and “tidy” data in Chapter 4. In this chapter we work with **regression**, a method that helps us study the relationship between an *outcome variable* or *response* and one or more *explanatory variables* or *regressors*. The method starts by proposing a *statistical model*. Data is then collected and used to estimate the coefficients or parameters for the model, and these results are typically used for two purposes:

1. For **explanation** when we want to describe how changes in one or more of the regressors are associated to changes in the response, quantify those changes, establish which of the regressors truly have an association with the response, or determine whether the model used to describe the relationship between the response and the explanatory variables seems appropriate.
2. For **prediction** when we want to determine, based on the observed values of the regressors what will the value of the response before this value has been observed. We are not concerned about how all the regressors relate and interact with one another or with the response, we simply want as good predictions as possible.

As an illustration, assume that we want to study the relationship between blood pressure and potential risk factors such as daily salt intake, age, and physical activity levels. The response is blood pressure, and the regressors are the risk factors. If we use linear regression for explanation, we may want to determine whether reducing daily salt intake has a real effect on lowering blood pressure, or by how much blood pressure decreases if an individual reduces their salt intake by half. This information may help target individuals of a specific age group with advice on dietary changes to manage blood pressure. On the other hand, if we use linear regression for prediction, we would like to determine, as accurately as possible, the blood pressure of a given individual based on the data collected about their salt intake, age, and physical activity levels. In this chapter, we will use linear regression for explanation.

The most basic and commonly-used type of regression is *linear regression*. Linear regression involves a *numerical* response and one or more regressors that can be *numerical* or *categorical*. It is called linear regression because the **statistical model** that describes the relationship between the expected response and the regressors is assumed to be linear. In particular, when the model has a single regressor, the linear

regression is the equation of a line. Linear regression is the foundation for almost any other type of regression or related method.

In Chapter 5 we introduce linear regression with only one regressor. In Section 5.1, the explanatory variable is numerical. This scenario is known as *simple linear regression*. In Section 5.2, the explanatory variable is categorical.

In Chapter 6 on multiple regression, we extend these ideas and work with models with two explanatory variables. In Section 6.1, we work with two numerical explanatory variables. In Section 6.2, we work with one numerical and one categorical explanatory variable and study model with and without interactions.

In Chapter 10 on inference for regression, we revisit the regression models and analyze the results using *statistical inference*, a method discussed in Chapters 7, 8, and 9 on sampling, bootstrapping and confidence intervals, and hypothesis testing and *p*-values, respectively. The focus there is also be on using linear regression for prediction instead of explanation.

We begin with regression with a single explanatory variable, . We also introduce the *correlation coefficient*, discuss about “correlation versus causation,” and determine whether the model *fits* the data observed.

Needed packages

We now load all the packages needed for this chapter (this assumes you’ve already installed them). In this chapter, we introduce some new packages:

1. The `tidyverse` “umbrella” (Wickham, 2023) package. Recall from our discussion in Section 4.4 that loading the `tidyverse` package by running `library(tidyverse)` loads the following commonly used data science packages all at once:
 - `ggplot2` for data visualization
 - `dplyr` for data wrangling
 - `tidyverse` for converting data to “tidy” format
 - `readr` for importing spreadsheet data into R
 - As well as the more advanced `purrr`, `tibble`, `stringr`, and `forcats` packages
1. The `moderndive` package of datasets and functions for tidyverse-friendly introductory linear regression as well as a data frame summary function.

If needed, read Section 1.3 for information on how to install and load R packages.

```
library(tidyverse)
library(moderndive)
```

5.1 One numerical explanatory variable

Before we introduce the model needed for simple linear regression, we present an example. Why do some countries exhibit high fertility rates while others have significantly lower ones? Are there correlations between fertility rates and life expectancy across different continents and nations? Could underlying socioeconomic factors be influencing these trends? These are all questions that are of interest to demographers and policy makers, as understanding fertility rates is important for planning and development. By analyzing the data set of UN member states, which includes variables such as country codes (ISO), fertility rates, and life expectancy for 2022, researchers can uncover patterns and make predictions about fertility rates based on life expectancy.

In this section, we aim to explain differences in fertility rates as a function of one numerical variable: life expectancy. Could it be that countries with higher life expectancy also have lower fertility rates? Could it be instead that countries with higher life expectancy tend to have higher fertility rates? Or could it be that there is no relationship between life expectancy and fertility rates? We answer these questions by modeling the relationship between fertility rates and life expectancy using *simple linear regression*, where we have:

1. A numerical outcome variable y (the country's fertility rate) and
2. A single numerical explanatory variable x (the country's life expectancy).

5.1.1 Exploratory data analysis

The data on the 193 current UN member states (as of 2024) can be found in the `un_member_states_2024` data frame included in the `moderndive` package. However, to keep things simple we include only those rows that don't have missing data with `na.omit()` and `select()` only the subset of the variables we'll consider in this chapter, and save this data in a new data frame called `UN_data_ch5`:

```
UN_data_ch5 <- un_member_states_2024 |>
  select(iso,
         life_exp = life_expectancy_2022,
         fert_rate = fertility_rate_2022,
         obes_rate = obesity_rate_2016)|>
  na.omit()
```

A crucial step before doing any kind of analysis or modeling is performing an *exploratory data analysis*, or EDA for short. EDA gives you a sense of the distributions

of the individual variables in your data, whether any potential relationships exist between variables, whether there are outliers and/or missing values, and (most importantly) how to build your model. Here are three common steps in an EDA:

1. Most crucially, looking at the raw data values.
2. Computing summary statistics, such as means, medians, and interquartile ranges.
3. Creating data visualizations.

We perform the first common step in an exploratory data analysis: looking at the raw data values. Because this step seems so trivial, unfortunately many data analysts ignore it. However, getting an early sense of what your raw data looks like can often prevent many larger issues down the road.

You can do this by using RStudio's spreadsheet viewer or by using the `glimpse()` function as introduced in Subsection 1.4.3 on exploring data frames:

```
glimpse(UN_data_ch5)
```

```
Rows: 181
Columns: 4
$ iso      <chr> "AFG", "ALB", "DZA", "AGO", "ATG", "ARG", "ARM", "AUS", ~
$ life_exp <dbl> 53.6, 79.5, 78.0, 62.1, 77.8, 78.3, 76.1, 83.1, 82.3, 7~
$ fert_rate <dbl> 4.3, 1.4, 2.7, 5.0, 1.6, 1.9, 1.6, 1.6, 1.5, 1.6, 1.4, ~
$ obes_rate <dbl> 5.5, 21.7, 27.4, 8.2, 18.9, 28.3, 20.2, 29.0, 20.1, 19.~
```

Observe that `Rows: 181` indicates that there are 181 rows/observations in `UN_data_ch5` after filtering out the missing values, where each row corresponds to one observed country/member state. It is important to note that the *observational unit* is an individual country. Recall from Subsection 1.4.3 that the observational unit is the “type of thing” that is being measured by our variables.

A full description of all the variables included in `un_member_states_2024` can be found by reading the associated help file (`run ?un_member_states_2024` in the console). However, we fully describe only the 4 variables we selected in `UN_data_ch5`:

1. `iso`: An identification variable used to distinguish between the 181 countries in the filtered dataset.
2. `fert_rate`: A numerical variable representing the country's fertility rate in 2022 corresponding to the expected number of children born per woman in child-bearing years. This is the outcome variable y of interest.
3. `life_exp`: A numerical variable representing the country's average life expectancy in 2022 in years. This is the primary explanatory variable x of interest.

4. `obes_rate`: A numerical variable representing the country's obesity rate in 2016. This will be another explanatory variable x that we use in the *Learning check* at the end of this subsection.

An alternative way to look at the raw data values is by choosing a random sample of the rows in `UN_data_ch5` by piping it into the `slice_sample()` function from the `dplyr` package. Here we set the `n` argument to be 5, indicating that we want a random sample of 5 rows. We display the results in Table 5.1. Note that due to the random nature of the sampling, you will likely end up with a different subset of 5 rows.

```
UN_data_ch5 |>
  slice_sample(n = 5)
```

TABLE 5.1: A random sample of 5 out of the 193 total countries (181 without missing data)

iso	life_exp	fert_rate	obes_rate
PRT	81.5	1.4	20.8
MNE	77.8	1.7	23.3
CPV	73.8	1.9	11.8
VNM	75.5	1.9	2.1
IDN	73.1	2.1	6.9

We have looked at the raw values in our `UN_data_ch5` data frame and got a preliminary sense of the data. We can now compute summary statistics. We start by computing the mean and median of our numerical outcome variable `fert_rate` and our numerical explanatory variable `life_exp`. We do this by using the `summarize()` function from `dplyr` along with the `mean()` and `median()` summary functions we saw in Section 3.3.

```
UN_data_ch5 |>
  summarize(mean_life_exp = mean(life_exp),
            mean_fert_rate = mean(fert_rate),
            median_life_exp = median(life_exp),
            median_fert_rate = median(fert_rate))
```

```
# A tibble: 1 x 4
  mean_life_exp mean_fert_rate median_life_exp median_fert_rate
            <dbl>           <dbl>           <dbl>           <dbl>
1         73.6            2.50          75.1             2
```

However, what if we want other summary statistics as well, such as the standard deviation (a measure of spread), the minimum and maximum values, and various percentiles?

Typing out all these summary statistic functions in `summarize()` would be long and tedious. Instead, we use the convenient `tidy_summary()` function from the `moderndive` package. This function takes in a data frame, summarizes it, and returns commonly used summary statistics in tidy format. We take our `UN_data_ch5` data frame, `select()` only the outcome and explanatory variables `fert_rate` and `life_exp`, and pipe them into the `tidy_summary` function:

```
UN_data_ch5 |>
  select(fert_rate, life_exp) |>
  tidy_summary()
```

```
# A tibble: 2 x 11
  column      n group type    min    Q1   mean median    Q3    max    sd
  <chr>     <int> <chr> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 fert_rate  181 <NA>  numeric  1.1   1.6  2.50    2     3.2   6.6  1.15
2 life_exp   181 <NA>  numeric  53.6  69.4 73.6    75.1  78.3  86.4  6.80
```

We can also do this more directly by providing which columns we'd like a summary of inside the `tidy_summary()` function:

```
UN_data_ch5 |>
  tidy_summary(columns = c(fert_rate, life_exp))
```

```
# A tibble: 2 x 11
  column      n group type    min    Q1   mean median    Q3    max    sd
  <chr>     <int> <chr> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 fert_rate  181 <NA>  numeric  1.1   1.6  2.50    2     3.2   6.6  1.15
2 life_exp   181 <NA>  numeric  53.6  69.4 73.6    75.1  78.3  86.4  6.80
```

Both styles of code return the same results. For the numerical variables `fert_rate` and `life_exp` it returns:

- `column`: the name of the column being summarized
- `n`: the number of non-missing values
- `group`: NA (missing) for numerical columns, but will break down a categorical variable into its levels
- `type`: which type of column is it (`numeric`, `character`, `factor`, or `logical`)
- `min`: the *minimum* value
- `Q1`: the 1st quartile: the value at which 25% of observations are smaller than it (the *25th percentile*)
- `mean`: the average value for measuring central tendency

- `median`: the 2nd quartile: the value at which 50% of observations are smaller than it (*the 50th percentile*)
- `q3`: the 3rd quartile: the value at which 75% of observations are smaller than it (*the 75th percentile*)
- `max`: the *maximum* value
- `sd`: the standard deviation value for measuring spread

Looking at this output, we can see how the values of both variables distribute. For example, the median fertility rate was 2, whereas the median life expectancy was 75.14 years. Furthermore, the middle 50% of fertility rates was between 1.6 and 3.2 (the first and third quartiles), whereas the middle 50% of life expectancies falls within 69.36 to 78.31.

The `tidy_summary()` function only returns what are known as *univariate* summary statistics: functions that take a single variable and return some numerical summary of that variable. However, there also exist *bivariate* summary statistics: functions that take in two variables and return some summary of those two variables. In particular, when the two variables are numerical, we can compute the *correlation coefficient*. Generally speaking, *coefficients* are quantitative expressions of a specific phenomenon. A *correlation coefficient* is a quantitative expression of the *strength of the linear relationship between two numerical variables*. Its value ranges between -1 and 1 where:

- -1 indicates a perfect *negative relationship*: As one variable increases, the value of the other variable tends to go down, following a straight line.
- 0 indicates no relationship: The values of both variables go up/down independently of each other.
- +1 indicates a perfect *positive relationship*: As the value of one variable goes up, the value of the other variable tends to go up as well in a linear fashion.

Figure 5.1 gives examples of 9 different correlation coefficient values for hypothetical numerical variables x and y . For example, observe in the top right plot that for a correlation coefficient of -0.75 there is a negative linear relationship between x and y , but it is not as strong as the negative linear relationship between x and y when the correlation coefficient is -0.9 or -1.

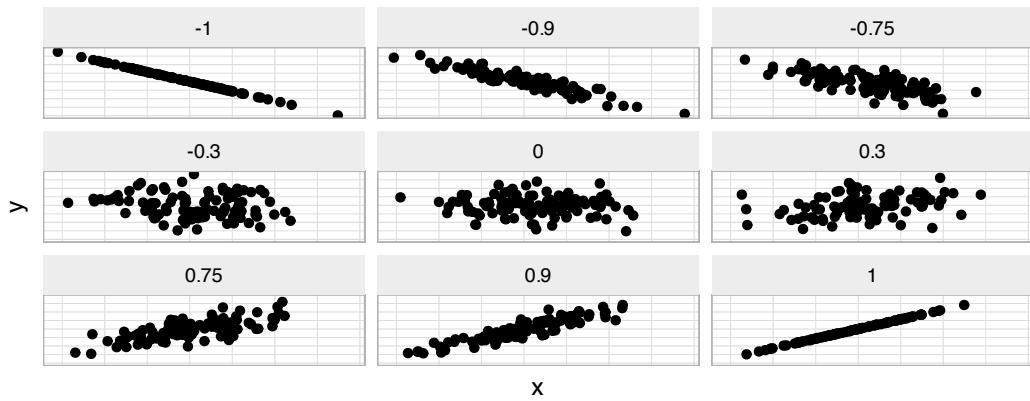


FIGURE 5.1: Nine different correlation coefficients.

The correlation coefficient can be computed using the `get_correlation()` function in the `moderndive` package. In this case, the inputs to the function are the two numerical variables for which we want to calculate the correlation coefficient.

We put the name of the outcome variable on the left-hand side of the ~ “tilde” sign, while putting the name of the explanatory variable on the right-hand side. This is known as R’s *formula notation*. We will use this same “formula” syntax with regression later in this chapter.

```
UN_data_ch5 |>
  get_correlation(formula = fert_rate ~ life_exp)
```

```
# A tibble: 1 x 1
  cor
  <dbl>
1 -0.812
```

An alternative way to compute correlation is to use the `cor()` summary function within a `summarize()`:

```
UN_data_ch5 |>
  summarize(correlation = cor(fert_rate, life_exp))
```

In our case, the correlation coefficient of -0.812 indicates that the relationship between fertility rate and life expectancy is “moderately negative.” There is a certain amount of subjectivity in interpreting correlation coefficients, especially those that are not close to the extreme values of -1, 0, and 1. To develop your intuition about correlation

coefficients, play the “Guess the Correlation” 1980’s style video game mentioned in Subsection 5.4.1.

We now perform the last step in EDA: creating data visualizations. Since both the `fert_rate` and `life_exp` variables are numerical, a scatterplot is an appropriate graph to visualize this data. We do this using `geom_point()` and display the result in Figure 5.2. Furthermore, we set the `alpha` value to `0.1` to check for any overplotting.

```
ggplot(UN_data_ch5,
       aes(x = life_exp, y = fert_rate)) +
  geom_point(alpha = 0.1) +
  labs(x = "Life Expectancy", y = "Fertility Rate")
```

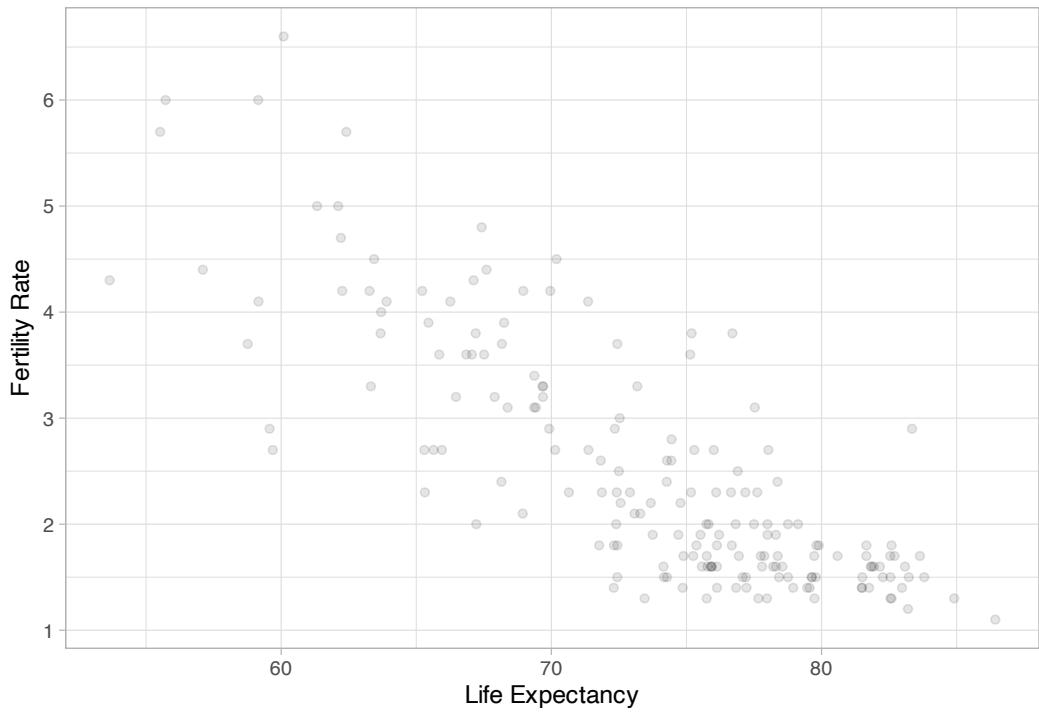


FIGURE 5.2: Scatterplot of relationship of life expectancy and fertility rate

We do not see much for overplotting due to little to no overlap in the points. Most life expectancy entries appear to fall between 70 and 80 years, while most fertility rate entries fall between 1.5 and 3.5 births. Furthermore, while opinions may vary, it is our opinion that the relationship between fertility rate and life expectancy is “moderately negative.” This is consistent with our earlier computed correlation coefficient of -0.812.

We build on the scatterplot in Figure 5.2 by adding a “best-fitting” line: of all possible lines we can draw on this scatterplot, it is the line that “best” fits through the cloud

of points. We do this by adding a new `geom_smooth(method = "lm", se = FALSE)` layer to the `ggplot()` code that created the scatterplot in Figure 5.2. The `method = "lm"` argument sets the line to be a “linear model.” The `se = FALSE` argument suppresses *standard error* uncertainty bars. (We’ll define the concept of *standard error* later in Subsection 7.3.2.)

```
ggplot(UN_data_ch5,
       aes(x = life_exp, y = fert_rate)) +
  geom_point(alpha = 0.1) +
  labs(
    x = "Life Expectancy",
    y = "Fertility Rate",
    title = "Scatterplot of relationship of life expectancy and fertility rate"
  ) +
  geom_smooth(method = "lm", se = FALSE)
```

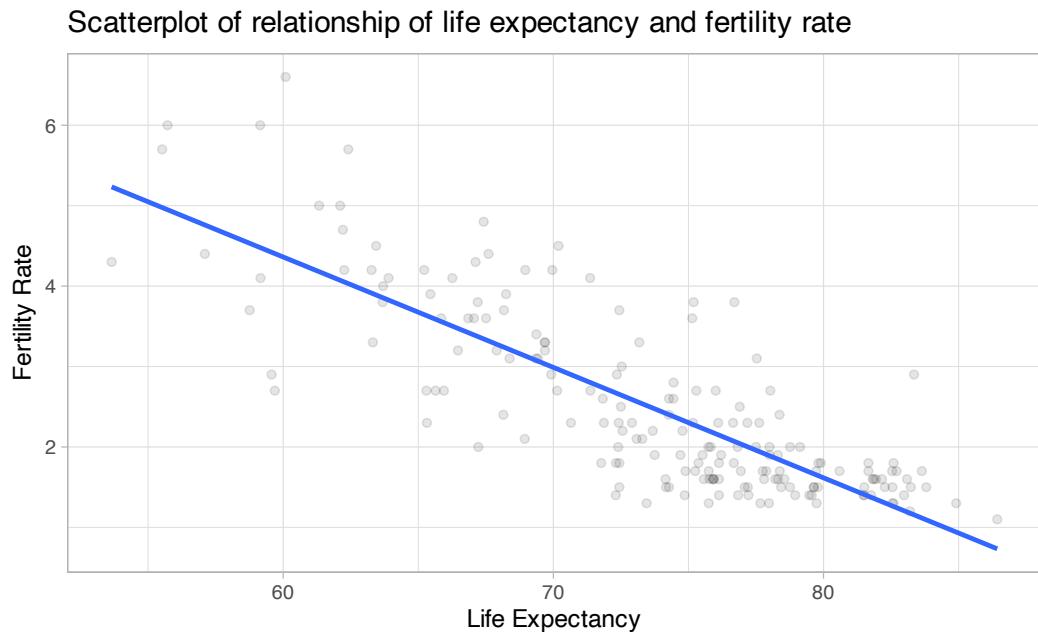


FIGURE 5.3: Regression line.

The line in the resulting Figure 5.3 is called a “regression line.” The regression line is a visual summary of the relationship between two numerical variables, in our case the outcome variable `fert_rate` and the explanatory variable `life_exp`. The negative slope of the blue line is consistent with our earlier observed correlation coefficient of -0.812 suggesting that there is a negative relationship between these two variables: as a country’s population has higher life expectancy it tends to have a lower fertility

rate. We'll see later, however, that while the correlation coefficient and the slope of a regression line always have the same sign (positive or negative), they typically do not have the same value.

Furthermore, a regression line is “best-fitting” in that it minimizes some mathematical criteria. We present these mathematical criteria in Subsection 5.3.2, but we suggest you read this subsection only after first reading the rest of this section on regression with one numerical explanatory variable.

Learning check

(LC5.1) Conduct a new exploratory data analysis with the same outcome variable y being `fert_rate` but with `obes_rate` as the new explanatory variable x . Remember, this involves three things:

- (a) Looking at the raw data values.
- (b) Computing summary statistics.
- (c) Creating data visualizations.

What can you say about the relationship between obesity rate and fertility rate based on this exploration?

5.1.2 Simple linear regression

You may recall from secondary/high school algebra that the equation of a line is $y = a + b \cdot x$. (Note that the \cdot symbol is equivalent to the \times “multiply by” mathematical symbol. We'll use the \cdot symbol in the rest of this book as it is more succinct.) It is defined by two coefficients a and b . The intercept coefficient a is the value of y when $x = 0$. The slope coefficient b for x is the increase in y for every increase of one in x . This is also called the “rise over run.”

However, when defining a regression line like the regression line in Figure 5.3, we use slightly different notation: the equation of the regression line is $\hat{y} = b_0 + b_1 \cdot x$. The intercept coefficient is b_0 , so b_0 is the value of \hat{y} when $x = 0$. The slope coefficient for x is b_1 , i.e., the increase in \hat{y} for every increase of one in x . Why do we put a “hat” on top of the y ? It's a form of notation commonly used in regression to indicate that we have a “fitted value,” or the value of y on the regression line for a given x value. We discuss this more in the upcoming Subsection 5.1.3.

We know that the regression line in Figure 5.3 has a negative slope b_1 corresponding to our explanatory x variable `life_exp`. Why? Because as countries tend to have

higher `life_exp` values, they tend to have lower `fert_rate` values. However, what is the numerical value of the slope b_1 ? What about the intercept b_0 ? We not compute these two values by hand, but rather we use a computer!

We can obtain the values of the intercept b_0 and the slope for `life_exp` b_1 by outputting the *linear regression coefficients*. This is done in two steps:

1. We first “fit” the linear regression model using the `lm()` function and save it in `demographics_model`.
2. We get the regression coefficients by applying `coef()` to `demographics_model`.

```
# Fit regression model:
demographics_model <- lm(fert_rate ~ life_exp,
                           data = UN_data_ch5)

# Get regression coefficients
coef(demographics_model)
```

We first focus on interpreting the regression coefficients, and later revisit the code that produced it. The coefficients are the intercept $b_0 = 12.599$ and the slope $b_1 = -0.137$ for `life_exp`. Thus the equation of the regression line in Figure 5.3 follows:

$$\begin{aligned}\hat{y} &= b_0 + b_1 \cdot x \\ \widehat{\text{fertility_rate}} &= b_0 + b_{\text{life_expectancy}} \cdot \text{life_expectancy} \\ &= 12.6 + -0.137 \cdot \text{life_expectancy}\end{aligned}$$

The intercept $b_0 = 12.599$ is the average fertility rate $\hat{y} = \widehat{\text{fertility_rate}}$ for those countries that had a `life_exp` of 0. Or in graphical terms, where the line intersects the y axis for $x = 0$. Note, however, that while the intercept of the regression line has a mathematical interpretation, it has no *practical* interpretation here, since observing a `life_exp` of 0 is impossible. Furthermore, looking at the scatterplot with the regression line in Figure 5.3, no countries had a life expectancy anywhere near 0.

Of greater interest is the slope $b_1 = b_{\text{life_expectancy}}$ for `life_exp` of -0.137. This summarizes the relationship between the fertility rate and life expectancy variables. Note that the sign is negative, suggesting a negative relationship between these two variables. This means countries with higher life expectancies tend to have lower fertility rates. Recall from earlier that the correlation coefficient is -0.812. They both have the same negative sign, but have a different value. Recall further that the correlation’s interpretation is the “strength of linear association”. The slope’s interpretation is a little different:

For every increase of 1 unit in `life_exp`, there is an *associated* decrease of, *on average*, 0.137 units of `fert_rate`.

We only state that there is an *associated* increase and not necessarily a *causal* increase. For example, perhaps it may not be that higher life expectancies directly cause lower fertility rates. Instead, the following could hold true: wealthier countries tend to have stronger educational backgrounds, improved health, a higher standard of living, and have lower fertility rates, while at the same time these wealthy countries also tend to have higher life expectancies. In other words, just because two variables are strongly associated, it does not necessarily mean that one causes the other. This is summed up in the often quoted phrase, “correlation is not necessarily causation.” We discuss this idea further in Subsection 5.3.1.

Furthermore, we say that this associated decrease is *on average* 0.137 units of `fert_rate`, because you might have two countries whose `life_exp` values differ by 1 unit, but their difference in fertility rates may not be exactly -0.137 . What the slope of -0.137 is saying is that across all possible countries, the *average* difference in fertility rate between two countries whose life expectancies differ by one is -0.137 .

Now that we have learned how to compute the equation for the regression line in Figure 5.3 using the model coefficient values and how to interpret the resulting intercept and slope, we revisit the code that generated these coefficients:

```
# Fit regression model:  
demographics_model <- lm(fert_rate ~ life_exp,  
                           data = UN_data_ch5)  
# Get regression coefficients:  
coef(demographics_model)
```

First, we “fit” the linear regression model to the data using the `lm()` function and save this as `demographics_model`. When we say “fit”, we mean “find the best fitting line to this data.” `lm()` stands for “linear model” and is used as follows: `lm(y ~ x, data = data_frame_name)` where:

- `y` is the outcome variable, followed by a tilde `~`. In our case, `y` is set to `fert_rate`.
- `x` is the explanatory variable. In our case, `x` is set to `life_exp`.
- The combination of `y ~ x` is called a *model formula*. (Note the order of `y` and `x`.) In our case, the model formula is `fert_rate ~ life_exp`. We saw such model formulas earlier when we computed the correlation coefficient using the `get_correlation()` function in Subsection 5.1.1.
- `data_frame_name` is the name of the data frame that contains the variables `y` and `x`. In our case, `data` is the `UN_data_ch5` data frame.

Second, we take the saved model in `demographics_model` and apply the `coef()` function to it to obtain the regression coefficients. This gives us the components of the regression equation line: the intercept b_0 and the slope b_1 .

Learning check

(LC5.2) Fit a new simple linear regression using `lm(fert_rate ~ obes_rate, data = UN_data_ch5)` where `obes_rate` is the new explanatory variable x . Get information about the “best-fitting” line from the regression coefficients by applying the `coef()` function. How do the regression results match up with the results from your earlier exploratory data analysis?

5.1.3 Observed/fitted values and residuals

We just saw how to get the value of the intercept and the slope of a regression line from the output of the `coef()` function. Now instead say we want information on individual observations. For example, we focus on the 21st of the 181 countries in the `UN_data_ch5` data frame in Table 5.2. This corresponds to the UN member state of Bosnia and Herzegovina (BIH).

TABLE 5.2: Data for the 21st country out of 193

iso	life_exp	fert_rate	obes_rate
BIH	78	1.3	17.9

What is the value \hat{y} on the regression line corresponding to this country’s `life_exp` value of 77.98? In Figure 5.4 we mark three values corresponding to these results for Bosnia and Herzegovina and give their statistical names:

- Circle: The *observed value* $y = 1.3$ is this country’s actual fertility rate.
- Square: The *fitted value* \hat{y} is the value on the regression line for $x = \text{life_exp} = 77.98$. This value is computed using the intercept and slope in the previous regression table:

$$\hat{y} = b_0 + b_1 \cdot x = 12.599 + -0.137 \cdot 77.98 = 1.894$$

- Arrow: The length of this arrow is the *residual* and is computed by subtracting the fitted value \hat{y} from the observed value y . The residual can be thought of as a model’s error or “lack of fit” for a particular observation. In the case of this country, it is $y - \hat{y} = 1.3 - 1.894 = -0.594$.

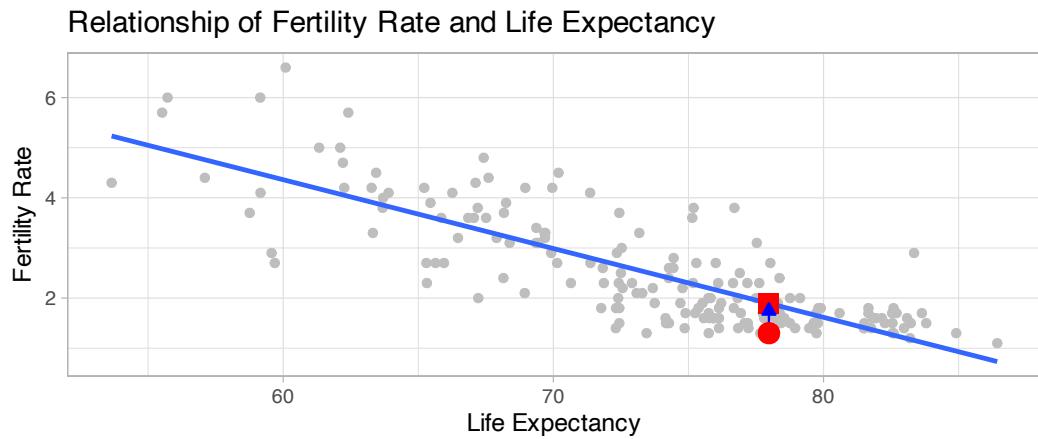


FIGURE 5.4: Example of observed value, fitted value, and residual.

Now say we want to compute both the fitted value $\hat{y} = b_0 + b_1 \cdot x$ and the residual $y - \hat{y}$ for *all* 181 UN member states with complete data as of 2024. Recall that each country corresponds to one of the 181 rows in the `UN_data_ch5` data frame and also one of the 181 points in the regression plot in Figure 5.4.

We could repeat the previous calculations we performed by hand 181 times, but that would be tedious and time consuming. Instead, we do this using a computer with the `get_regression_points()` function. We apply the `get_regression_points()` function to `demographics_model`, which is where we saved our `lm()` model in the previous section. In Table 5.3 we present the results of only the 21st through 24th courses for brevity's sake.

```
regression_points <- get_regression_points(demographics_model)
regression_points
```

TABLE 5.3: Regression points (for only the 21st through 24th countries)

ID	fert_rate	life_exp	fert_rate_hat	residual
21	1.3	78.0	1.89	-0.594
22	2.7	65.6	3.59	-0.888
23	1.6	75.9	2.18	-0.576
24	1.7	80.6	1.53	0.165

This function is an example of what is known in computer programming as a *wrapper function*. It takes other pre-existing functions and “wraps” them into a single function that hides its inner workings. This concept is illustrated in Figure 5.5.

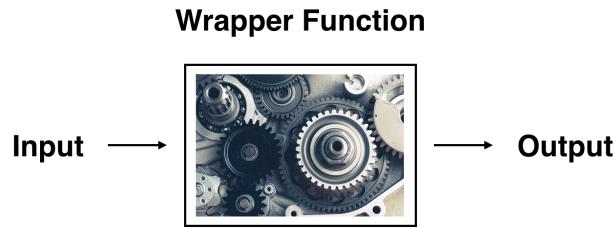


FIGURE 5.5: The concept of a wrapper function.

So all you need to worry about is what the inputs look like and what the outputs look like; you leave all the other details “under the hood of the car.” In our regression modeling example, the `get_regression_points()` function takes a saved `lm()` linear regression model as input and returns a data frame of the regression predictions as output. If you are interested in learning more about the `get_regression_points()` function’s inner workings, check out Subsection 5.3.3.

We inspect the individual columns and match them with the elements of Figure 5.4:

- The `fert_rate` column represents the observed outcome variable y . This is the y -position of the 181 black points.
- The `life_exp` column represents the values of the explanatory variable x . This is the x -position of the 181 black points.
- The `fert_rate_hat` column represents the fitted values \hat{y} . This is the corresponding value on the regression line for the 181 x values.
- The `residual` column represents the residuals $y - \hat{y}$. This is the 181 vertical distances between the 181 black points and the regression line.

Just as we did for the 21st country in the `UN_data_ch5` dataset (in the first row of the table), we repeat the calculations for the 24th country (in the fourth row of Table 5.3). This corresponds to the country of Brunei (BRN):

- `fert_rate = 1.7` is the observed `fert_rate` y for this country.
- `life_exp = 80.59` is the value of the explanatory variable `life_exp` x for Brunei.
- `fert_rate_hat = 1.53 = 12.599 + (-0.137) · 80.59` is the fitted value \hat{y} on the regression line for this country.
- `residual = 0.165 = 1.7 - 1.53` is the value of the residual for this country. In other words, the model’s fitted value was off by 0.165 fertility rate units for Brunei.

At this point, you can skip ahead if you like to Subsection 5.3.2 to learn about the processes behind what makes “best-fitting” regression lines. As a primer, a “best-fitting” line refers to the line that minimizes the *sum of squared residuals* out of all possible lines we can draw through the points. In Section 5.2, we’ll discuss another common scenario of having a categorical explanatory variable and a numerical outcome variable.

Learning check

(LC5.3) Generate a data frame of the residuals of the model where you used `obesity_rate_2024` as the explanatory x variable.

5.2 One categorical explanatory variable

It is an unfortunate truth that life expectancy is not the same across all countries in the world. International development agencies are interested in studying these differences in life expectancy in the hopes of identifying where governments should allocate resources to address this problem. In this section, we explore differences in life expectancy in two ways:

1. Differences between continents: Are there significant differences in average life expectancy between the six populated continents of the world: Africa, North America, South America, Asia, Europe, and Oceania?
2. Differences within continents: How does life expectancy vary within the world's five continents? For example, is the spread of life expectancy among the countries of Africa larger than the spread of life expectancy among the countries of Asia?

To answer such questions, we use an updated version of the `gapminder` data frame we visualized in Figure 2.1 in Subsection 2.1.2 on the grammar of graphics. This updated data `un_member_states_2024` data we worked with earlier in this chapter, and it is included in the `moderndive` package. This dataset has international development statistics such as life expectancy, GDP per capita, and population for 193 countries for years near 2024.

We use this data for basic regression again, but now using an explanatory variable x that is categorical, as opposed to the numerical explanatory variable model we used in the previous Section 5.1.

1. A numerical outcome variable y (a country's life expectancy) and
2. A single categorical explanatory variable x (the continent that the country is a part of).

When the explanatory variable x is categorical, the concept of a “best-fitting” regression line is a little different than the one we saw previously in Section 5.1 where the

explanatory variable x was numerical. We study these differences shortly in Subsection 5.2.2, but first we conduct an exploratory data analysis.

5.2.1 Exploratory data analysis

The data on the 193 countries can be found in the `un_member_states_2024` data frame included in the `moderndive` package. However, to keep things simple, we `select()` only the subset of the variables we'll consider in this chapter and focus only on rows where we have no missing values with `na.omit()`. We'll save this data in a new data frame called `gapminder2022`:

```
gapminder2022 <- un_member_states_2024 |>
  select(country, life_exp = life_expectancy_2022,
         continent, gdp_per_capita) |>
  na.omit()
```

We perform the first common step in an exploratory data analysis: looking at the raw data values. You can do this by using RStudio's spreadsheet viewer or by using the `glimpse()` command as introduced in Subsection 1.4.3 on exploring data frames:

```
glimpse(gapminder2022)
```

```
Rows: 188
Columns: 4
$ country      <chr> "Afghanistan", "Albania", "Algeria", "Andorra", "A~
$ life_exp     <dbl> 53.6, 79.5, 78.0, 83.4, 62.1, 77.8, 78.3, 76.1, 83~
$ continent    <fct> Asia, Europe, Africa, Europe, Africa, North Americ~
$ gdp_per_capita <dbl> 356, 6810, 4343, 41993, 3000, 19920, 13651, 7018, ~
```

Observe that `Rows: 188` indicates that there are 188 rows/observations in `gapminder2022`, where each row corresponds to one country. In other words, the *observational unit* is an individual country. Furthermore, observe that the variable `continent` is of type `<fct>`, which stands for *factor*, which is R's way of encoding categorical variables.

A full description of all the variables included in `un_member_states_2024` can be found by reading the associated help file (run `?un_member_states_2024` in the console). However, we fully describe only the 39 variables we selected in `gapminder2022`:

1. `country`: An identification variable of type character/text used to distinguish the 142 countries in the dataset.
2. `life_exp`: A numerical variable of that country's life expectancy at birth. This is the outcome variable y of interest.

3. continent: A categorical variable with five levels. Here “levels” correspond to the possible categories: Africa, Asia, Americas, Europe, and Oceania. This is the explanatory variable x of interest.
4. gdp_per_capita: A numerical variable of that country’s GDP per capita in US inflation-adjusted dollars that we’ll use as another outcome variable y in the *Learning check* at the end of this subsection.

We look at a random sample of five out of the 188 countries in Table 5.4.

```
gapminder2022 |> sample_n(size = 5)
```

TABLE 5.4: Random sample of 5 out of 193 countries

country	life_exp	continent	gdp_per_capita
Panama	77.6	North America	17358
Micronesia, Federated States of	74.4	Oceania	3714
Burundi	67.4	Africa	259
United Arab Emirates	79.6	Asia	53708
India	67.2	Asia	2411

Note that random sampling will likely produce a different subset of 5 rows for you than what’s shown. Now that we have looked at the raw values in our `gapminder2022` data frame and got a sense of the data, we move on to computing summary statistics. We once again apply the `tidy_summary()` function from the `moderndive` package. Recall from our previous EDA that this function takes in a data frame, summarizes it, and returns commonly used summary statistics. We take our `gapminder2022` data frame, `select()` only the outcome and explanatory variables `life_exp` and `continent`, and pipe them into the `tidy_summary()` function:

```
gapminder2022 |>
  select(life_exp, continent) |>
  tidy_summary()
```

```
# A tibble: 7 x 11
  column      n group    type   min    Q1  mean median    Q3  max    sd
  <chr>     <int> <chr> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 life_exp    188 <NA>  nume~  53.6  69.4  73.8  75.2  78.4  89.6  6.93
2 continent     52 Africa fact~  NA     NA     NA     NA     NA     NA     NA
3 continent     44 Asia   fact~  NA     NA     NA     NA     NA     NA     NA
4 continent     43 Europe fact~  NA     NA     NA     NA     NA     NA     NA
5 continent     23 North  A~ fact~  NA     NA     NA     NA     NA     NA     NA
6 continent     14 Oceania fact~  NA     NA     NA     NA     NA     NA     NA
7 continent     12 South  A~ fact~  NA     NA     NA     NA     NA     NA     NA
```

The `tidy_summary()` output now reports summaries for categorical variables and for the numerical variables we reviewed before. Let's focus just on discussing the results for the categorical factor variable `continent`:

- `n`: The number of non-missing entries for each group
- `group`: Breaks down a categorical variable into its unique levels. For this variable, it corresponds to Africa, Asia, North and South America, Europe, and Oceania.
- `type`: The data type of the variable. Here, it is a `factor`.
- `min` to `sd`: These are missing since calculating the five-number summary, the mean, and standard deviation for categorical variables doesn't make sense.

Turning our attention to the summary statistics of the numerical variable `life_exp`, we observe that the global median life expectancy in 2022 was 75.14. Thus, half of the world's countries (96 countries) had a life expectancy less than 75.14. The mean life expectancy of 73.55 is lower, however. Why is the mean life expectancy lower than the median?

We can answer this question by performing the last of the three common steps in an exploratory data analysis: creating data visualizations. We visualize the distribution of our outcome variable $y = \text{life_exp}$ in Figure 5.6.

```
ggplot(gapminder2022, aes(x = life_exp)) +  
  geom_histogram(binwidth = 5, color = "white") +  
  labs(x = "Life expectancy", y = "Number of countries",  
       title = "Histogram of distribution of worldwide life expectancies")
```

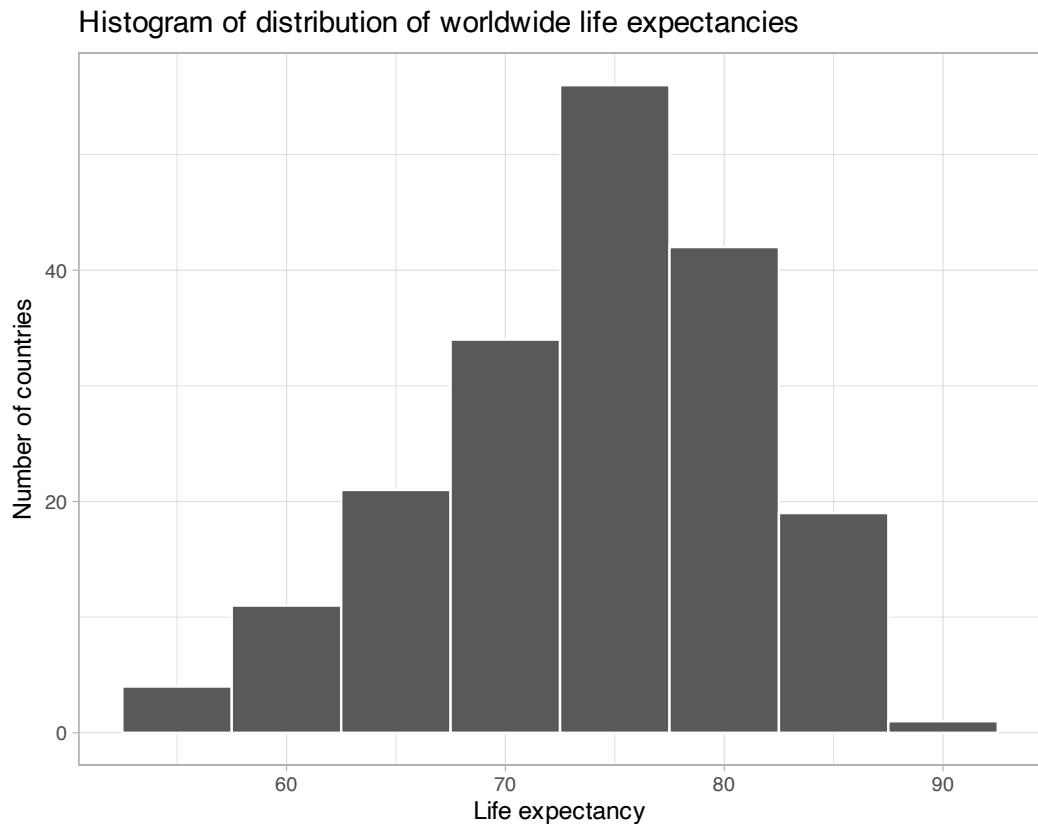


FIGURE 5.6: Histogram of life expectancy in 2022.

We see that this data is *left-skewed*, also known as *negatively skewed*: there are a few countries with low life expectancy that are bringing down the mean life expectancy. However, the median is less sensitive to the effects of such outliers; hence, the median is greater than the mean in this case.

Remember, however, that we want to compare life expectancies both between continents and within continents. In other words, our visualizations need to incorporate some notion of the variable `continent`. We can do this easily with a faceted histogram. Recall from Section 2.6 that facets allow us to split a visualization by the different values of another variable. We display the resulting visualization in Figure 5.7 by adding a `facet_wrap(~ continent, nrow = 2)` layer.

```
ggplot(gapminder2022, aes(x = life_exp)) +
  geom_histogram(binwidth = 5, color = "white") +
  labs(x = "Life expectancy",
       y = "Number of countries",
       title = "Histogram of distribution of worldwide life expectancies") +
  facet_wrap(~ continent, nrow = 2)
```

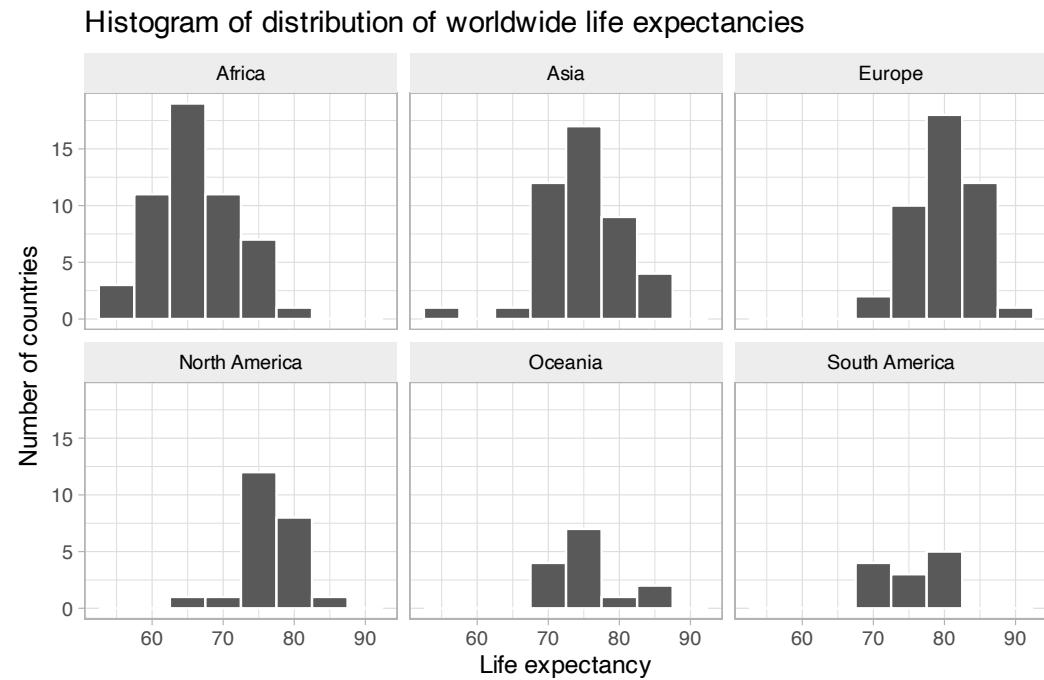


FIGURE 5.7: Life expectancy in 2022.

Observe that unfortunately the distribution of African life expectancies is much lower than the other continents. In Europe, life expectancies tend to be higher and furthermore do not vary as much. On the other hand, both Asia and Africa have the most variation in life expectancies.

Recall that an alternative method to visualize the distribution of a numerical variable split by a categorical variable is by using a side-by-side boxplot. We map the categorical variable `continent` to the x -axis and the different life expectancies within each continent on the y -axis in Figure 5.8.

```
ggplot(gapminder2022, aes(x = continent, y = life_exp)) +
  geom_boxplot() +
  labs(x = "Continent", y = "Life expectancy",
       title = "Life expectancy by continent")
```

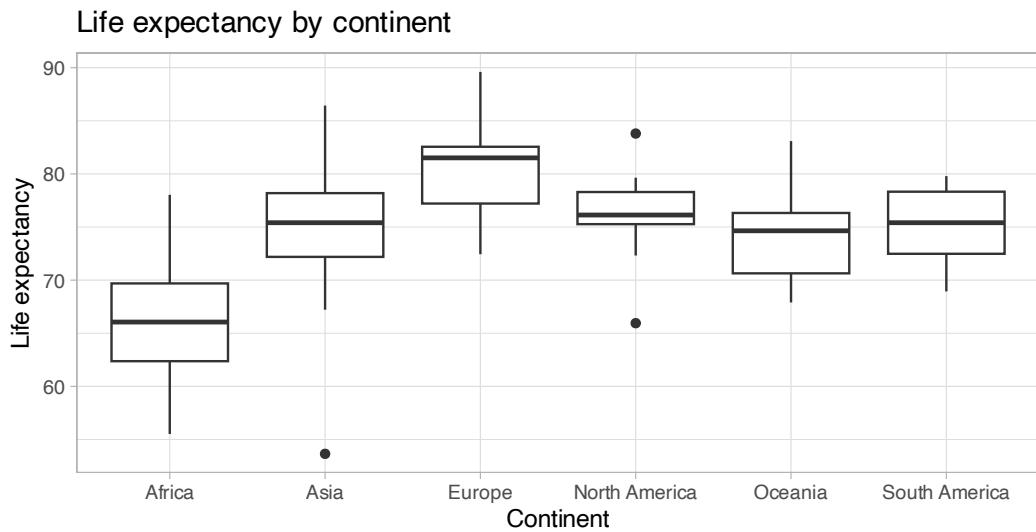


FIGURE 5.8: Life expectancy in 2022.

Some people prefer comparing the distributions of a numerical variable between different levels of a categorical variable using a boxplot instead of a faceted histogram. This is because we can make quick comparisons between the categorical variable's levels with imaginary horizontal lines. For example, observe in Figure 5.8 that we can quickly convince ourselves that Europe has the highest median life expectancies by drawing an imaginary horizontal line near $y = 81$. Furthermore, as we observed in the faceted histogram in Figure 5.7, Africa and Asia have the largest variation in life expectancy as evidenced by their large interquartile ranges (the heights of the boxes).

It's important to remember, however, that the solid lines in the middle of the boxes correspond to the medians (the middle value) rather than the mean (the average). So, for example, if you look at Asia, the solid line denotes the median life expectancy of around 75 years. This tells us that half of all countries in Asia have a life expectancy below 75 years, whereas half have a life expectancy above 75 years.

We compute the median and mean life expectancy for each continent with a little more data wrangling and display the results in Table 5.5.

```
life_exp_by_continent <- gapminder2022 |>
  group_by(continent) |>
  summarize(median = median(life_exp),
            mean = mean(life_exp))
life_exp_by_continent
```

Observe the order of the second column `median` life expectancy:

TABLE 5.5: Life expectancy by continent

continent	median	mean
Africa	66.1	66.3
Asia	75.4	75.0
Europe	81.5	79.9
North America	76.1	76.3
Oceania	74.6	74.4
South America	75.4	75.2

- Africa is lowest,
- Oceania, South America, Asia, and North America are next with similar medians, and then
- Europe has the highest.

This ordering corresponds to the ordering of the solid black lines inside the boxes in our side-by-side boxplot in Figure 5.8.

We now turn our attention to the values in the third column `mean`. Using Africa's mean life expectancy of 66.31 as a *baseline for comparison*, we start making comparisons to the mean life expectancies of the other four continents and put these values in Table 5.6, which we'll revisit later on in this section.

1. For Asia, it is $74.95 - 66.31 = 8.64$ years higher.
2. For Europe, it is $79.91 - 66.31 = 13.6$ years higher.
3. For North America, it is $76.29 - 66.31 = 9.98$ years higher.
4. For Oceania, it is $74.42 - 66.31 = 8.11$ years higher.
5. For South America, it is $75.23 - 66.31 = 8.92$ years higher.

TABLE 5.6: Mean life expectancy by continent and relative differences from mean for Africa

continent	mean	Difference versus Africa
Africa	66.3	0.00
Asia	75.0	8.64
Europe	79.9	13.60
North America	76.3	9.99
Oceania	74.4	8.11
South America	75.2	8.92

Learning check

(LC5.4) Conduct a new exploratory data analysis with the same explanatory variable `x` being `continent` but with `gdp_per_capita` as the new outcome variable `y`. What can

you say about the differences in GDP per capita between continents based on this exploration?

5.2.2 Linear regression

In Subsection 5.1.2 we introduced simple linear regression, which involves modeling the relationship between a numerical outcome variable y and a numerical explanatory variable x . In our life expectancy example, we now instead have a categorical explanatory variable `continent`. Our model will not yield a “best-fitting” regression line like in Figure 5.3, but rather *offsets* relative to a baseline for comparison.

As we did in Subsection 5.1.2 when studying the relationship between fertility rates and life expectancy, we output the regression coefficients for this model. Recall that this is done in two steps:

1. We first “fit” the linear regression model using the `lm(y ~ x, data)` function and save it in `life_exp_model`.
2. We get the regression coefficients by applying the `coef()` function to `life_exp_model`.

```
life_exp_model <- lm(life_exp ~ continent, data = gapminder2022)
coef(life_exp_model)
```

	(Intercept)	continentAsia	continentEurope
continentNorth America	66.31	8.64	13.60
	continentOceania	continentSouth America	
continentAfrica	9.99	8.11	8.92

We once again focus on the values in these coefficient values. Why are there now 6 entries? We break them down one-by-one:

1. `intercept` corresponds to the mean life expectancy of countries in Africa of 66.31 years.
2. `continentAsia` corresponds to countries in Asia and the value +8.64 is the same difference in mean life expectancy relative to Africa we displayed in Table 5.6. In other words, the mean life expectancy of countries in Asia is $66.31 + 8.64 = 74.95$.
3. `continentEurope` corresponds to countries in Europe and the value +13.6 is the same difference in mean life expectancy relative to Africa we displayed in Table 5.6. In other words, the mean life expectancy of countries in Europe is $66.31 + 13.6 = 79.91$.

4. `continentNorth America` corresponds to countries in North America and the value $+9.98$ is the same difference in mean life expectancy relative to Africa we displayed in Table 5.6. In other words, the mean life expectancy of countries in North America is $66.31 + 9.98 = 76.29$.
5. `continentOceania` corresponds to countries in Oceania and the value $+8.11$ is the same difference in mean life expectancy relative to Africa we displayed in Table 5.6. In other words, the mean life expectancy of countries in Oceania is $66.31 + 8.11 = 74.42$.
6. `continentSouth America` corresponds to countries in South America and the value $+8.92$ is the same difference in mean life expectancy relative to Africa we displayed in Table 5.6. In other words, the mean life expectancy of countries in South America is $66.31 + 8.92 = 75.23$.

To summarize, the 6 values for the regression coefficients correspond to the “baseline for comparison” continent Africa (the intercept) as well as five “offsets” from this baseline for the remaining 5 continents: Asia, Europe, North America, Oceania, and South America.

You might be asking at this point why was Africa chosen as the “baseline for comparison” group. This is the case for no other reason than it comes first alphabetically of the six continents; by default R arranges factors/categorical variables in alphanumeric order. You can change this baseline group to be another continent if you manipulate the variable `continent`’s factor “levels” using the `forcats` package. See Chapter 15¹ of *R for Data Science* (Gromelund and Wickham, 2017) for examples.

We now write the equation for our fitted values $\hat{y} = \widehat{\text{life exp}}$.

$$\begin{aligned}\hat{y} = \widehat{\text{life exp}} &= b_0 + b_{\text{Asia}} \cdot \mathbb{1}_{\text{Asia}}(x) + b_{\text{Europe}} \cdot \mathbb{1}_{\text{Europe}}(x) \\ &\quad + b_{\text{North America}} \cdot \mathbb{1}_{\text{North America}}(x) + b_{\text{Oceania}} \cdot \mathbb{1}_{\text{Oceania}}(x) + b_{\text{South America}} \cdot \mathbb{1}_{\text{South America}}(x) \\ &= 66.31 + 8.64 \cdot \mathbb{1}_{\text{Asia}}(x) + 13.6 \cdot \mathbb{1}_{\text{Euro}}(x) \\ &\quad + 9.98 \cdot \mathbb{1}_{\text{North America}}(x) + 8.11 \cdot \mathbb{1}_{\text{Oceania}}(x) + 8.92 \cdot \mathbb{1}_{\text{South America}}(x)\end{aligned}$$

Whoa! That looks daunting! Don’t fret, however, as once you understand what all the elements mean, things simplify greatly. First, $\mathbb{1}_A(x)$ is what’s known in mathematics as an “indicator function.” It returns only one of two possible values, 0 and 1, where

$$\mathbb{1}_A(x) = \begin{cases} 1 & \text{if } x \text{ is in } A \\ 0 & \text{if otherwise} \end{cases}$$

In a statistical modeling context, this is also known as a *dummy variable*. In our case, we consider the first such indicator variable $\mathbb{1}_{\text{Amer}}(x)$. This indicator function returns 1 if a country is in the Asia, 0 otherwise:

¹<https://r4ds.had.co.nz/factors.html>

$$\mathbb{1}_{\text{Amer}}(x) = \begin{cases} 1 & \text{if country } x \text{ is in Asia} \\ 0 & \text{otherwise} \end{cases}$$

Second, b_0 corresponds to the intercept as before; in this case, it is the mean life expectancy of all countries in Africa. Third, the b_{Asia} , b_{Europe} , $b_{\text{North America}}$, b_{Oceania} , and $b_{\text{South America}}$ represent the 5 “offsets relative to the baseline for comparison” in the regression coefficients.

We put this all together and compute the fitted value $\hat{y} = \widehat{\text{life exp}}$ for a country in Africa. Since the country is in Africa, all five indicator functions $\mathbb{1}_{\text{Asia}}(x)$, $\mathbb{1}_{\text{Europe}}(x)$, $\mathbb{1}_{\text{North America}}(x)$, $\mathbb{1}_{\text{Oceania}}(x)$, and $\mathbb{1}_{\text{South America}}(x)$ will equal 0, and thus:

$$\begin{aligned} \widehat{\text{life exp}} &= b_0 + b_{\text{Asia}} \cdot \mathbb{1}_{\text{Asia}}(x) + b_{\text{Europe}} \cdot \mathbb{1}_{\text{Europe}}(x) \\ &\quad + b_{\text{North America}} \cdot \mathbb{1}_{\text{North America}}(x) + b_{\text{Oceania}} \cdot \mathbb{1}_{\text{Oceania}}(x) + b_{\text{South America}} \cdot \mathbb{1}_{\text{South America}}(x) \\ &= 66.31 + 8.64 \cdot \mathbb{1}_{\text{Asia}}(x) + 13.6 \cdot \mathbb{1}_{\text{Europe}}(x) \\ &\quad + 9.98 \cdot \mathbb{1}_{\text{North America}}(x) + 8.11 \cdot \mathbb{1}_{\text{Oceania}}(x) + 8.92 \cdot \mathbb{1}_{\text{South America}}(x) \\ &= 66.31 + 8.64 \cdot 0 + 13.6 \cdot 0 + 9.98 \cdot 0 + 8.11 \cdot 0 + 8.92 \cdot 0 \\ &= 66.31 \end{aligned}$$

In other words, all that is left is the intercept b_0 , corresponding to the average life expectancy of African countries of 66.31 years. Next, say we are considering a country in Asia. In this case, only the indicator function $\mathbb{1}_{\text{Asia}}(x)$ for Asia will equal 1, while all the others will equal 0, and thus:

$$\begin{aligned} \widehat{\text{life exp}} &= b_0 + b_{\text{Asia}} \cdot \mathbb{1}_{\text{Asia}}(x) + b_{\text{Europe}} \cdot \mathbb{1}_{\text{Europe}}(x) \\ &\quad + b_{\text{North America}} \cdot \mathbb{1}_{\text{North America}}(x) + b_{\text{Oceania}} \cdot \mathbb{1}_{\text{Oceania}}(x) + b_{\text{South America}} \cdot \mathbb{1}_{\text{South America}}(x) \\ &= 66.31 + 8.64 \cdot \mathbb{1}_{\text{Asia}}(x) + 13.6 \cdot \mathbb{1}_{\text{Europe}}(x) \\ &\quad + 9.98 \cdot \mathbb{1}_{\text{North America}}(x) + 8.11 \cdot \mathbb{1}_{\text{Oceania}}(x) + 8.92 \cdot \mathbb{1}_{\text{South America}}(x) \\ &= 66.31 + 8.64 \cdot 1 + 13.6 \cdot 0 + 9.98 \cdot 0 + 8.11 \cdot 0 + 8.92 \cdot 0 \\ &= 66.31 + 8.64 \\ &= 74.95 \end{aligned}$$

which is the mean life expectancy for countries in Asia of 74.95 years in Table 5.6. Note the “offset from the baseline for comparison” is +8.64 years.

We do one more. Say we are considering a country in South America. In this case, only the indicator function $\mathbb{1}_{\text{South America}}(x)$ for South America will equal 1, while all the others will equal 0, and thus:

$$\begin{aligned}
 \widehat{\text{life exp}} &= b_0 + b_{\text{Asia}} \cdot \mathbb{1}_{\text{Asia}}(x) + b_{\text{Europe}} \cdot \mathbb{1}_{\text{Europe}}(x) \\
 &\quad + b_{\text{North America}} \cdot \mathbb{1}_{\text{North America}}(x) + b_{\text{Oceania}} \cdot \mathbb{1}_{\text{Oceania}}(x) + b_{\text{South America}} \cdot \mathbb{1}_{\text{South America}}(x) \\
 &= 66.31 + 8.64 \cdot \mathbb{1}_{\text{Asia}}(x) + 13.6 \cdot \mathbb{1}_{\text{Europe}}(x) \\
 &\quad + 9.98 \cdot \mathbb{1}_{\text{North America}}(x) + 8.11 \cdot \mathbb{1}_{\text{Oceania}}(x) + 8.92 \cdot \mathbb{1}_{\text{South America}}(x) \\
 &= 66.31 + 8.64 \cdot 0 + 13.6 \cdot 0 + 9.98 \cdot 0 + 8.11 \cdot 0 + 8.92 \cdot 1 \\
 &= 66.31 + 8.92 \\
 &= 75.23
 \end{aligned}$$

which is the mean life expectancy for South American countries of 75.23 years in Table 5.6. The “offset from the baseline for comparison” here is +8.64 years.

We generalize this idea a bit. If we fit a linear regression model using a categorical explanatory variable x that has k possible categories, the regression table will return an intercept and $k - 1$ “offsets.” In our case, since there are $k = 6$ continents, the regression model returns an intercept corresponding to the baseline for comparison group of Africa and $k - 1 = 5$ offsets corresponding to Asia, Europe, North America, Oceania, and South America.

Understanding a regression table output when you are using a categorical explanatory variable is a topic those new to regression often struggle with. The only real remedy for these struggles is practice, practice, practice. However, once you equip yourselves with an understanding of how to create regression models using categorical explanatory variables, you’ll be able to incorporate many new variables into your models, given the large amount of the world’s data that is categorical.

Learning check

(LC5.5) Fit a new linear regression using `lm(gdp_per_capita ~ continent, data = gapminder2022)` where `gdp_per_capita` is the new outcome variable y . Get information about the “best-fitting” line from the regression coefficients. How do the regression results match up with the results from your previous exploratory data analysis?

5.2.3 Observed/fitted values and residuals

Recall in Subsection 5.1.3, we defined the following three concepts:

1. Observed values y , or the observed value of the outcome variable
2. Fitted values \hat{y} , or the value on the regression line for a given x value
3. Residuals $y - \hat{y}$, or the error between the observed value and the fitted value

We obtained these values and other values using the `get_regression_points()` function from the `moderndive` package. This time, however, we add an argument setting `ID = "country"`: this is telling the function to use the variable `country` in `gapminder2022` as an *identification variable* in the output. This will help contextualize our analysis by matching values to countries.

```
regression_points <- get_regression_points(life_exp_model, ID = "country")
regression_points
```

TABLE 5.7: Regression points (First 10 out of 142 countries)

country	life_exp	continent	life_exp_hat	residual
Afghanistan	53.6	Asia	75.0	-21.300
Albania	79.5	Europe	79.9	-0.438
Algeria	78.0	Africa	66.3	11.720
Andorra	83.4	Europe	79.9	3.512
Angola	62.1	Africa	66.3	-4.200
Antigua and Barbuda	77.8	North America	76.3	1.505
Argentina	78.3	South America	75.2	3.083
Armenia	76.1	Asia	75.0	1.180
Australia	83.1	Oceania	74.4	8.674
Austria	82.3	Europe	79.9	2.362

Observe in Table 5.7 that `life_exp_hat` contains the fitted values $\hat{y} = \widehat{\text{life exp}}$. If you look closely, there are only 5 possible values for `life_exp_hat`. These correspond to the five mean life expectancies for the 5 continents that we displayed in Table 5.6 and computed using the regression coefficient values.

The `residual` column is simply $y - \hat{y} = \text{life_exp} - \text{life_exp_hat}$. These values can be interpreted as the deviation of a country's life expectancy from its continent's average life expectancy. For example, look at the first row of Table 5.7 corresponding to Afghanistan. The residual of $y - \hat{y} = 53.6 - 74.95 = -21.4$ is telling us that Afghanistan's life expectancy is a whopping 21.4 years lower than the mean life expectancy of all Asian countries. This can in part be explained by the many years of war that country had suffered.

Learning check

(LC5.6) Using either the sorting functionality of RStudio's spreadsheet viewer or using the data wrangling tools you learned in Chapter 3, identify the five countries with the five smallest (most negative) residuals? What do these negative residuals say about their life expectancy relative to their continents' life expectancy?

(LC5.7) Repeat this process, but identify the five countries with the five largest (most positive) residuals. What do these positive residuals say about their life expectancy relative to their continents' life expectancy?

5.3 Related topics

5.3.1 Correlation is not necessarily causation

Throughout this chapter we have been cautious when interpreting regression slope coefficients. We always discussed the “associated” effect of an explanatory variable x on an outcome variable y . For example, our statement from Subsection 5.1.2 that “for every increase of 1 unit in `life_exp`, there is an *associated* decrease of on average 0.137 units of `fert_rate`.” We include the term “associated” to be extra careful not to suggest we are making a *causal* statement. So while `life_exp` is negatively correlated with `fert_rate`, we can’t necessarily make any statements about life expectancy’s direct causal effect on fertility rates without more information.

Here is another example: a not-so-great medical doctor goes through medical records and finds that patients who slept with their shoes on tended to wake up more with headaches. So this doctor declares, “Sleeping with shoes on causes headaches!”



FIGURE 5.9: Does sleeping with shoes on cause headaches?

However, there is a good chance that if someone is sleeping with their shoes on, it is potentially because they are intoxicated from alcohol. Furthermore, higher levels of drinking leads to more hangovers, and hence more headaches. The amount of alcohol consumption here is what’s known as a *confounding/lurking* variable. It “lurks” behind the scenes, confounding the causal relationship (if any) of “sleeping with shoes on” with “waking up with a headache.” We can summarize this in Figure 5.10 with a *causal graph* where:

- Y is a *response* variable; here it is “waking up with a headache.”
- X is a *treatment* variable whose causal effect we are interested in; here it is “sleeping with shoes on.”

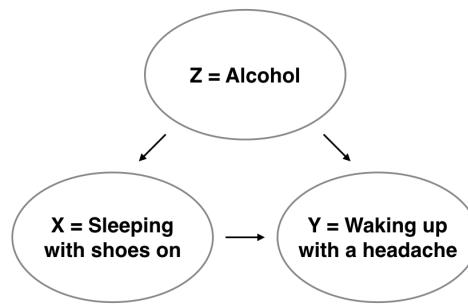


FIGURE 5.10: Causal graph.

To study the relationship between Y and X, we could use a regression model where the outcome variable is set to Y and the explanatory variable is set to be X, as you've been doing throughout this chapter. However, Figure 5.10 also includes a third variable with arrows pointing at both X and Y:

- Z is a *confounding* variable that affects both X and Y, thereby “confounding” their relationship. Here the confounding variable is alcohol.

Alcohol will cause people to be both more likely to sleep with their shoes on as well as be more likely to wake up with a headache. Thus any regression model of the relationship between X and Y should also use Z as an explanatory variable. In other words, our doctor needs to take into account who had been drinking the night before. In the next chapter, we'll start covering multiple regression models that allow us to incorporate more than one variable in our regression models.

Establishing causation is a tricky problem and frequently takes either carefully designed experiments or methods to control for the effects of confounding variables. Both these approaches attempt, as best they can, either to take all possible confounding variables into account or negate their impact. This allows researchers to focus only on the relationship of interest: the relationship between the outcome variable Y and the treatment variable X.

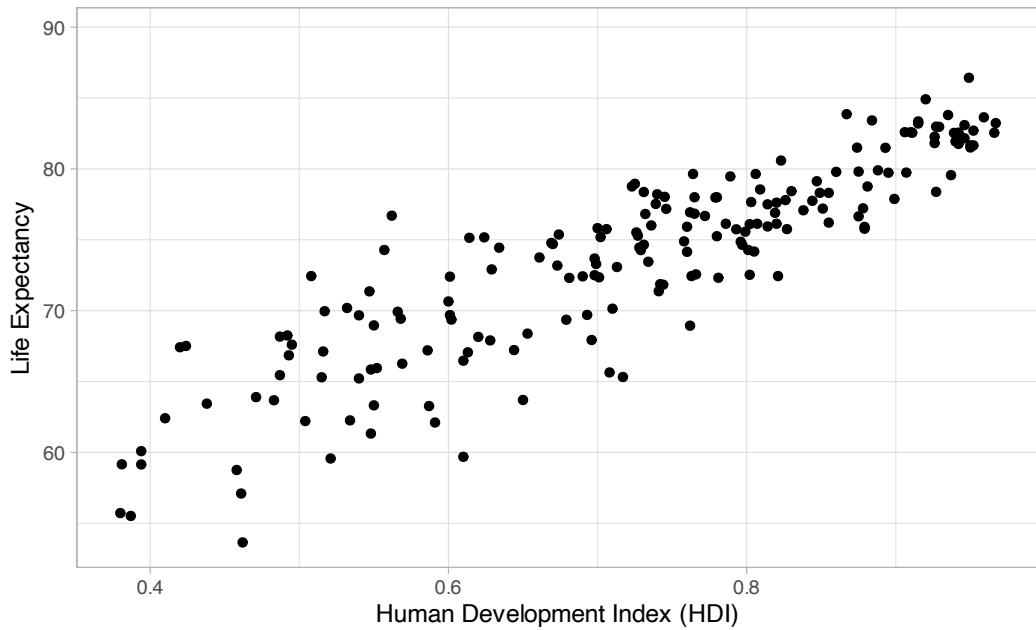
As you read news stories, be careful not to fall into the trap of thinking that correlation necessarily implies causation. Check out the Spurious Correlations² website for some rather comical examples of variables that are correlated, but are definitely not causally related.

Coming back to our UN member states data, a confounding variable could be the level of economic development of a country. This could affect both life expectancy and fertility rates. A proxy for looking at the level of economic development could be the Human Development Index (HDI). It measures a country's average achievements in three basic aspects of human development: health (life expectancy), education (mean and expected years of schooling), and standard of living (gross national income per

²<http://www.tylervigen.com/spurious-correlations>

capita). This is stored in the `hdi_2022` column of `un_member_states_2024`. We explore its relationship with life expectancy and fertility rates:

```
ggplot(data = un_member_states_2024,
       aes(x = hdi_2022, y = life_expectancy_2022)) +
  geom_point() +
  labs(x = "Human Development Index (HDI)", y = "Life Expectancy")
```



```
ggplot(data = un_member_states_2024,
       aes(x = hdi_2022, y = fertility_rate_2022)) +
  geom_point() +
  labs(x = "Human Development Index (HDI)", y = "Fertility Rate")
```

```
un_member_states_2024 |>
  get_correlation(life_expectancy_2022 ~ hdi_2022, na.rm = TRUE)
```

```
# A tibble: 1 × 1
```

```
cor
<dbl>
1 0.889
```

```
un_member_states_2024 |>
  get_correlation(fertility_rate_2022 ~ hdi_2022, na.rm = TRUE)
```

```
# A tibble: 1 × 1
  cor
  <dbl>
1 -0.849
```

Looking at both of the scatterplots above, we see a strong positive linear relationship between the Human Development Index (HDI) and life expectancy, as well as a strong negative correlation between the Human Development Index (HDI) and fertility rates. The correlation coefficients between the Human Development Index (HDI) and life expectancy, as well as the Human Development Index (HDI) and fertility rates, are also given. These findings as well as some additional understanding of socioeconomic factors suggest that the Human Development Index (HDI) is a confounding variable in the relationship between life expectancy and fertility rates.

5.3.2 Best-fitting line

Regression lines are also known as “best-fitting” lines. But what do we mean by “best”? We unpack the criteria that is used in regression to determine “best.” Recall Figure 5.4, where for Bosnia and Herzegovina we marked the *observed value* y with a circle, the *fitted value* \hat{y} with a square, and the *residual* $y - \hat{y}$ with an arrow. We re-display Figure 5.4 in the top-left plot of Figure 5.11 in addition to three more arbitrarily chosen countries:

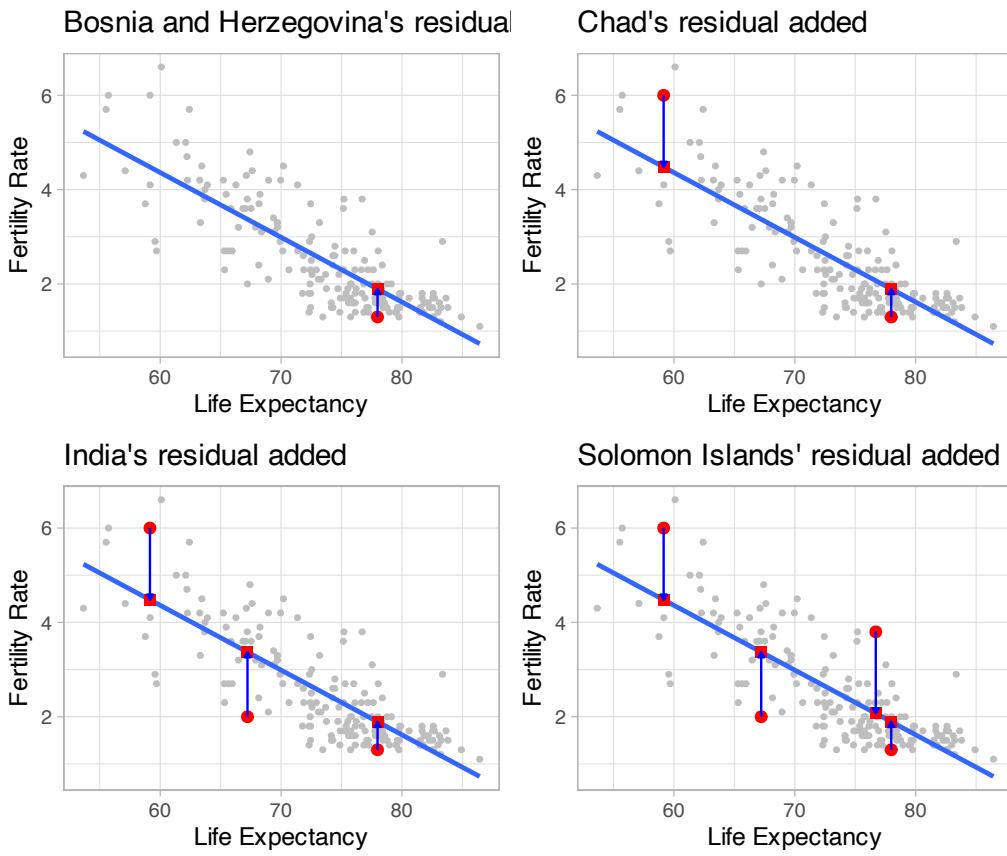


FIGURE 5.11: Example of observed value, fitted value, and residual.

The four plots refer to:

1. The country of Bosnia and Herzegovina had a life expectancy $x = 77.98$ and fertility rate $y = 1.3$. The residual in this case is $1.3 - 1.894 = -0.594$, which we mark with a arrow in the top-left plot.
2. The addition of Chad, which had a life expectancy $x = 59.15$ and fertility rate $y = 6$. The residual in this case is $6 - 4.479 = 1.521$, which we mark with a new arrow in the top-right plot.
3. The addition of India, which had a life expectancy $x = 67.22$ and fertility rate $y = 2$. The residual in this case is $2 - 3.371 = -1.371$, which we mark with a new arrow in the bottom-left plot.
4. The addition of the Solomon Islands, which had a life expectancy $x = 76.7$ and fertility rate $y = 3.8$. The residual in this case is $3.8 - 2.069 = 1.731$, which we mark with a new arrow in the bottom-right plot.

Now say we repeated this process of computing residuals for all 181 countries with complete information, then we squared all the residuals, and then we summed them.

We call this quantity the *sum of squared residuals*; it is a measure of the *lack of fit* of a model. Larger values of the sum of squared residuals indicate a bigger lack of fit. This corresponds to a worse fitting model.

If the regression line fits all the points perfectly, then the sum of squared residuals is 0. This is because if the regression line fits all the points perfectly, then the fitted value \hat{y} equals the observed value y in all cases, and hence the residual $y - \hat{y} = 0$ in all cases, and the sum of even a large number of 0's is still 0.

Furthermore, of all possible lines we can draw through the cloud of 181 points, the regression line minimizes this value. In other words, the regression and its corresponding fitted values \hat{y} minimizes the sum of the squared residuals:

$$\sum_{i=1}^n (y_i - \hat{y}_i)^2$$

We use our data wrangling tools from Chapter 3 to compute the sum of squared residuals exactly:

```
# Fit regression model:
demographics_model <- lm(fert_rate ~ life_exp,
                           data = UN_data_ch5)

# Get regression points:
regression_points <- get_regression_points(demographics_model)
regression_points
# Compute sum of squared residuals
regression_points |>
  mutate(squared_residuals = residual^2) |>
  summarize(sum_of_squared_residuals = sum(squared_residuals))
```

```
# A tibble: 181 x 5
  ID fert_rate life_exp fert_rate_hat residual
  <int>     <dbl>    <dbl>      <dbl>    <dbl>
1     1       4.3     53.6      5.23   -0.934
2     2       1.4     79.5      1.69   -0.289
3     3       2.7     78.0      1.89   0.813
4     4       5       62.1      4.07   0.928
5     5       1.6     77.8      1.92   -0.318
6     6       1.9     78.3      1.85   0.052
7     7       1.6     76.1      2.15   -0.548
8     8       1.6     83.1      1.19   0.408
9     9       1.5     82.3      1.30   0.195
10    10      1.6     74.2      2.42   -0.819
# i 171 more rows
```

Any other straight line drawn in the figure would yield a sum of squared residuals greater than 81.265. This is a mathematically guaranteed fact that you can prove using calculus and linear algebra. That's why alternative names for the linear regression line are the *best-fitting line* and the *least-squares line*. Why do we square the residuals (i.e., the arrow lengths)? So that both positive and negative deviations of the same amount are treated equally. (That being said, while taking the absolute value of the residuals would also treat both positive and negative deviations of the same amount equally, squaring the residuals is used for reasons related to calculus: taking derivatives and minimizing functions. To learn more we suggest you consult a textbook on mathematical statistics.)

Learning check

(LC5.8) Note in Figure 5.12 there are 3 points marked with dots and:

- The “best” fitting solid regression line
- An arbitrarily chosen dotted line
- Another arbitrarily chosen dashed line

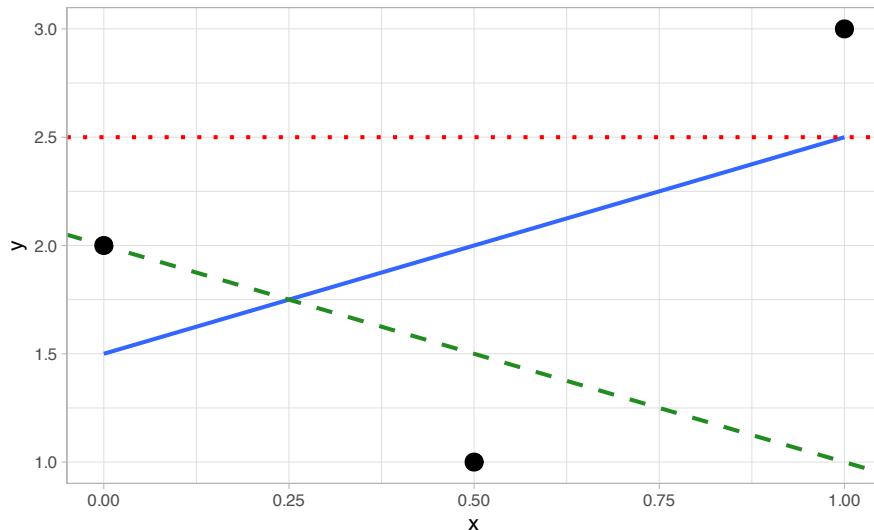


FIGURE 5.12: Regression line and two others.

Compute the sum of squared residuals by hand for each line and show that of these three lines, the regression line has the smallest value.

5.3.3 `get_regression_x()` functions

Recall in this chapter we introduced a wrapper function from the `moderndive` package:

- `get_regression_points()` that returns point-by-point information from a regression model in Subsection 5.1.3.

What is going on behind the scenes with the `get_regression_points()` function? We mentioned in Subsection 5.1.2 that this was an example of a *wrapper function*. Such functions take other pre-existing functions and “wrap” them into single functions that hide the user from their inner workings. This way all the user needs to worry about is what the inputs look like and what the outputs look like. In this subsection, we’ll “get under the hood” of these functions and see how the “engine” of these wrapper functions works.

The `get_regression_points()` function is a wrapper function, returning information about the individual points involved in a regression model like the fitted values, observed values, and the residuals. `get_regression_points()` uses the `augment()` function in the `broom` package³ to produce the data shown in Table 5.8. Additionally, it uses the `clean_names()` from the `janitor` package⁴ (Firke, 2023) to clean up the variable names.

```
library(broom)
library(janitor)
demographics_model |>
  augment() |>
  mutate_if(is.numeric, round, digits = 3) |>
  clean_names() |>
  select(-c("std_resid", "hat", "sigma", "cooksdi", "std_resid"))
```

In this case, it outputs only the variables of interest to students learning regression: the outcome variable y (`fert_rate`), all explanatory/predictor variables (`life_exp`), all resulting fitted values \hat{y} used by applying the equation of the regression line to `life_exp`, and the residual $y - \hat{y}$.

If you are even more curious about how these and other wrapper functions work, take a look at the source code for these functions on GitHub⁵.

³<https://broom.tidyverse.org/>

⁴<https://github.com/sfirke/janitor>

⁵https://github.com/moderndive/moderndive/blob/master/R/regression_functions.R

TABLE 5.8: Regression points using augment() from broom package

	fert_rate	life_exp	fitted	resid
	4.3	53.6	5.23	-0.934
	1.4	79.5	1.69	-0.289
	2.7	78.0	1.89	0.813
	5.0	62.1	4.07	0.928
	1.6	77.8	1.92	-0.318
	1.9	78.3	1.85	0.052
	1.6	76.1	2.15	-0.548
	1.6	83.1	1.19	0.408
	1.5	82.3	1.30	0.195
	1.6	74.2	2.42	-0.819

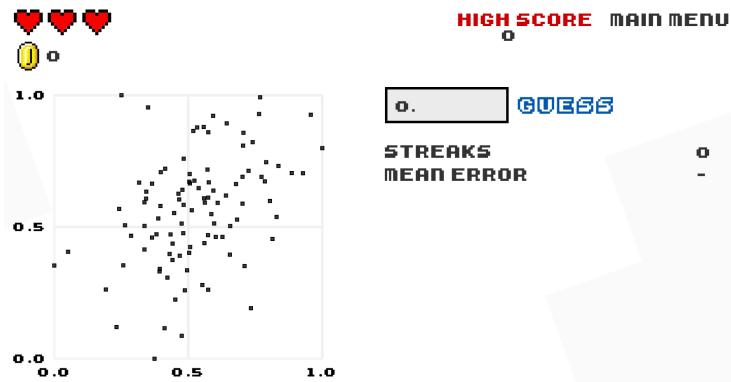
5.4 Conclusion

5.4.1 Additional resources

Solutions to all *Learning checks* can be found online in Appendix D⁶.

An R script file of all R code used in this chapter is available at <https://www.moderndive.com/scripts/05-regression.R>.

As we suggested in Subsection 5.1.1, interpreting coefficients that are not close to the extreme values of -1, 0, and 1 can be somewhat subjective. To help develop your sense of correlation coefficients, we suggest you play the 80s-style video game called, “Guess the Correlation”, at <http://guessthecorrelation.com/>.

**FIGURE 5.13:** Preview of “Guess the Correlation” game.

⁶<https://moderndive.com/D-appendixD.html>

5.4.2 What's to come?

In this chapter, you've studied the term *simple linear regression*, where you fit models that only have one explanatory variable. In Chapter 6, we'll study *multiple regression*, where our regression models can now have more than one explanatory variable moving a little bit more advanced than the basic form of simple linear regression! In particular, we'll consider two scenarios: regression models with one numerical and one categorical explanatory variable and regression models with two numerical explanatory variables. This will allow you to construct more sophisticated and more powerful models, all in the hopes of better explaining your outcome variable y .

6

Multiple Regression

In Chapter 5 we studied simple linear regression as a model that represents the relationship between two variables: an outcome variable or response y and an explanatory variable or regressor x . Furthermore to keep things simple, we only considered models with one explanatory x variable that was either numerical in Section 5.1 or categorical in Section 5.2.

In this chapter we introduce multiple linear regression, the direct extension to simple linear regression when more than one explanatory variable is taking into account to explain changes in the outcome variable. As we show in the next few sections, much of the material developed for simple linear regression translates directly into multiple linear regression, but the interpretation of the associated effect of any one explanatory variable must be made taking into account the other explanatory variables included in the model.

Needed packages

If needed, read Section 1.3 for information on how to install and load R packages.

```
library(tidyverse)
library(moderndive)
library(ISLR2)
```

6.1 One numerical and one categorical explanatory variable

We continue using the UN member states data set introduced in Section 5.1. Recall that we studied the relationship between the outcome variable fertility rate, y , and the regressor life expectancy, x .

In this section, we introduce one additional regressor to this model: the categorical variable `income` group with four categories: `Low income`, `Lower middle income`, `Upper middle`

income, and High income. We now want to study how fertility rate changes due to changes in life expectancy and different income levels. To do this, we use *multiple regression*. Observe that we now have:

1. A numerical outcome variable y , the fertility rate in a given country or state, and
2. Two explanatory variables:
3. A numerical explanatory variable x_1 , the life expectancy.
4. A categorical explanatory variable x_2 , the income group.

6.1.1 Exploratory data analysis

The UN member states data frame is included in the `moderndive` package. To keep things simple, we `select()` only the subset of the variables needed here, and save this data in a new data frame called `UN_data_ch6`. Note that the variables used are different than the ones chosen in Chapter 5. We also set the `income` variable to be a `factor` so that its levels show up in the expected order.

```
UN_data_ch6 <- un_member_states_2024 |>
  select(country, life_expectancy_2022,
         fertility_rate_2022, income_group_2024)|>
  na.omit()|>
  rename(life_exp = life_expectancy_2022, fert_rate = fertility_rate_2022,
         income = income_group_2024)|>
  mutate(income = factor(income,
                        levels = c("Low income", "Lower middle income",
                                  "Upper middle income", "High income")))
```

Recall the three common steps in an exploratory data analysis we saw in Subsection 5.1.1:

1. Inspecting a sample of raw values.
2. Computing summary statistics.
3. Creating data visualizations.

We first look at the raw data values by either looking at `UN_data_ch6` using RStudio's spreadsheet viewer or by using the `glimpse()` function from the `dplyr` package:

```
glimpse(UN_data_ch6)
```

Rows: 182

```
Columns: 4
$ country    <chr> "Afghanistan", "Albania", "Algeria", "Angola", "Antigua~
$ life_exp   <dbl> 53.6, 79.5, 78.0, 62.1, 77.8, 78.3, 76.1, 83.1, 82.3, 7~
$ fert_rate  <dbl> 4.3, 1.4, 2.7, 5.0, 1.6, 1.9, 1.6, 1.6, 1.5, 1.6, 1.4, ~
$ income     <fct> Low income, Upper middle income, Lower middle income, L~
```

The variable `country` contains all the UN member states. R reads this variable as character, `<chr>`, and beyond the country identification it will not be needed for the analysis. The variables life expectancy, `life_exp`, and fertility rate, `fert_rate`, are numerical, and the variable income, `income`, is categorical. In R, categorical variables are called factors and the categories are factor levels.

We also display a random sample of 5 rows of the 182 rows corresponding to different countries in Table 6.1. Remember due to the random nature of the sampling, you will likely end up with a different subset of 5 rows.

```
UN_data_ch6 |> sample_n(size = 5)
```

TABLE 6.1: A random sample of 5 out of 182 UN member states

country	life_exp	fert_rate	income
Trinidad and Tobago	75.9	1.6	High income
Micronesia, Federated States of	74.4	2.6	Lower middle income
North Macedonia	76.8	1.4	Upper middle income
Portugal	81.5	1.4	High income
Madagascar	68.2	3.7	Low income

Life expectancy, `life_exp`, is an estimate of how many years, on average, a person in a given country is expected to live. Fertility rate, `fert_rate`, is the average number of live births per woman of childbearing age in a country. As we did in our exploratory data analyses in Sections 5.1.1 and 5.2.1 from the previous chapter, we obtain summary statistics:

```
UN_data_ch6 |>
  select(life_exp, fert_rate, income) |>
  tidy_summary()
```

```
# A tibble: 6 x 11
  column      n group    type   min    Q1  mean median    Q3  max    sd
  <chr>     <int> <chr>   <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 life_exp    182 <NA>  nume~  53.6  69.4  73.7    75.2  78.4  86.4  6.86
2 fert_rate    182 <NA>  nume~   0.9   1.6   2.49     2     3.2   6.6  1.16
```

```

3 income      25 Low inc~ fact~ NA    NA    NA    NA    NA    NA    NA
4 income      52 Lower m~ fact~ NA    NA    NA    NA    NA    NA    NA
5 income      49 Upper m~ fact~ NA    NA    NA    NA    NA    NA    NA
6 income      56 High in~ fact~ NA    NA    NA    NA    NA    NA    NA

```

Recall that each row in `UN_data_ch6` represents a particular country or UN member state. The `tidy_summary()` function shows a summary for the numerical variables life expectancy (`life_exp`), fertility rate (`fert_rate`), and the categorical variable income group (`income`). When the variable is numerical, the `tidy_summary()` function provides the total number of observations in the data frame, the five-number summary, the mean, and the standard deviation. For example, the first row of our summary refers to life expectancy as `life_exp`. There are 182 observations for this variable, it is a numerical variable, and the first quartile, `Q1`, is 69.4; this means that the life expectancy of 25% of the UN member states is less than 69.4 years. When a variable in the data set is categorical, also called a `factor`, the summary shows all the categories or factor levels and the number of observations for each level. For example, income group, `income`, is a factor with four factor levels: `Low Income`, `Lower middle income`, `Upper middle income`, and `High income`. The summary also provides the number of states for each factor level; observe, for example, that the data set has 56 UN members states that are considered `High Income` states.

Furthermore, we can compute the correlation coefficient between our two numerical variables: `life_exp` and `fert_rate`. Recall from Subsection 5.1.1 that correlation coefficients only exist between numerical variables. We observe that they are “strongly negatively” correlated.

```

UN_data_ch6 |>
  get_correlation(formula = fert_rate ~ life_exp)

```

```

# A tibble: 1 × 1
  cor
  <dbl>
1 -0.815

```

We are ready to create data visualizations, the last of our exploratory data analysis. Given that the outcome variable `fert_rate` and explanatory variable `life_exp` are both numerical, we can create a scatterplot to display their relationship, as we did in Figure 5.2. But this time, we incorporate the categorical variable `income` by mapping this variable to the `color` aesthetic, thereby creating a *colored* scatterplot.

```
ggplot(UN_data_ch6, aes(x = life_exp, y = fert_rate, color = income)) +
  geom_point() +
  labs(x = "Life Expectancy", y = "Fertility Rate", color = "Income group") +
  geom_smooth(method = "lm", se = FALSE)
```

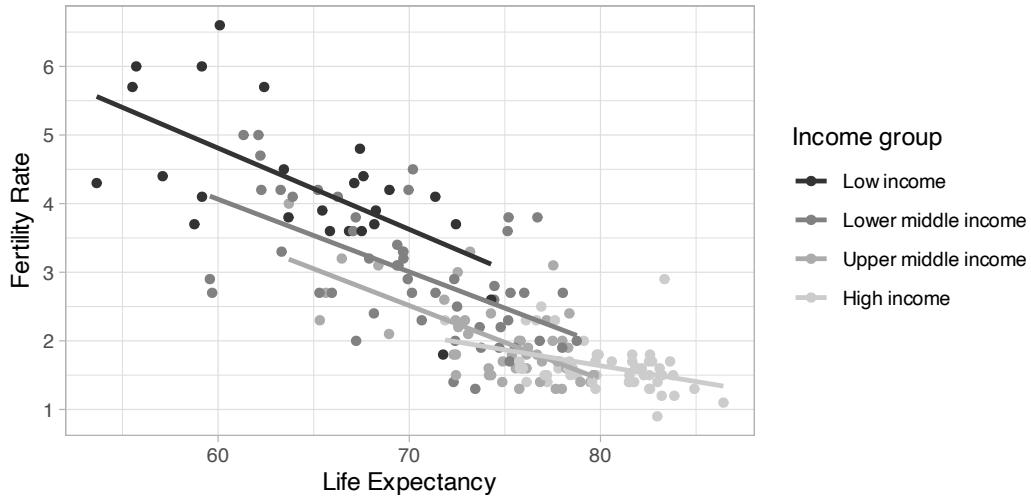


FIGURE 6.1: Colored scatterplot of relationship of teaching score and age.

In the resulting Figure 6.1, observe that `ggplot()` assigns a default color scheme to the points and to the lines associated with the four levels of `income`: `Low income`, `Lower middle income`, `Upper middle income`, and `High income`. Furthermore, the `geom_smooth(method = "lm", se = FALSE)` layer automatically fits a different regression line for each group.

We can see some interesting trends. First, observe that we get a different line for each income group. Second, the slopes for all the income groups are negative. Third, the slope for the `High income` group is clearly less steep than the slopes for all other three groups. So, the changes in fertility rate due to changes in life expectancy are dependent on the level of income of a given country. Fourth, observe that high income countries have, in general, high life expectancy and low fertility rates.

6.1.2 Model with interactions

We can represent the four regression lines in Figure 6.1 as a multiple regression model with *interactions*.

Before we do this, however, we review a linear regression with only one categorical explanatory variable. Recall in Subsection 5.2.2 we fit a regression model for each country life expectancy as a function of the corresponding continent. We produce the corresponding analysis here, now using the fertility rate as the response variable and

the income group as the categorical explanatory variable. We'll use slightly different notation to what was done previously to make the model more general.

A linear model with a categorical explanatory variable is called a one-factor model where factor refers to the categorical explanatory variable and the categories are also called factor levels. We represent the categories using indicator functions or dummy variables. In our UN data example, The variable `income` has four categories or levels: `Low income`, `Lower middle income`, `Upper middle income`, and `High income`. The corresponding dummy variables needed are:

$$\begin{aligned} D_1 &= \begin{cases} 1 & \text{if the UN member state has low income} \\ 0 & \text{otherwise} \end{cases} \\ D_2 &= \begin{cases} 1 & \text{if the UN member state has lower middle income} \\ 0 & \text{otherwise} \end{cases} \\ D_3 &= \begin{cases} 1 & \text{if the UN member state has high middle income} \\ 0 & \text{otherwise} \end{cases} \\ D_4 &= \begin{cases} 1 & \text{if the UN member state has high income} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

So, for example, if a given UN member state has `Low income`, its dummy variables are $D_1 = 1$ and $D_2 = D_3 = D_4 = 0$. Similarly, if another UN member state has `High middle income`, then its dummy variables would be $D_1 = D_2 = D_4 = 0$ and $D_3 = 1$. Using dummy variables, the mathematical formulation of the linear regression for our example is:

$$\hat{y} = \widehat{\text{fert rate}} = b_0 + b_2 D_2 + b_3 D_3 + b_4 D_4$$

or if we want to express it in terms of the i th observation in our data set, we can include the i th subscript:

$$\hat{y}_i = \widehat{\text{fert rate}} = b_0 + b_2 D_{2i} + b_3 D_{3i} + b_4 D_{4i}$$

Recall that the coefficient b_0 represents the intercept and the coefficients b_2 , b_3 , and b_4 are the offsets based on the appropriate category. The dummy variables, D_2 , D_3 , and D_4 , take the values of zero or one depending on the corresponding category of any given country. Observe also that D_1 does not appear in the model. The reason for this is entirely mathematical: if the model would contain an intercept and all the dummy variables, the model would be over-specified, that is, it would contain one redundant explanatory variable. The solution is to drop one of the variables. We keep the intercept because it provides flexibility when interpreting more complicated models, and we drop one of the dummy variables which, by default in R, is the first dummy variable, D_1 . This does not mean that we are losing information of the first level D_1 . If a country is part of the `Low income` level, $D_1 = 1$, $D_2 = D_3 = D_4 = 0$, so most of the terms in the regression are zero and the linear regression becomes:

$$\hat{y} = \widehat{\text{fert rate}} = b_0$$

So the intercept represents the average fertility rate when the country is a `Low income` country. Similarly, if another country is part of the `High middle income` level, then $D_1 = D_2 = D_4 = 0$ and $D_3 = 1$ so the linear regression becomes:

$$\hat{y} = \text{fert rate} = b_0 + b_3$$

The average fertility rate for a `High middle income` country is $b_0 + b_3$. Observe that b_3 is an *offset* for life expectancy between the baseline level and the `High middle income` level. The same logic applies to the model for each possible income category.

We obtain the regression coefficients using the `lm()` function and the command `coef()` to obtain the coefficients of the linear regression:

```
one_factor_model <- lm(fert_rate ~ income, data = UN_data_ch6)
coef(one_factor_model)
```

We present these results on a table with the mathematical notation used above:

	Coefficients	Values
(Intercept)	b0	4.28
incomeLower middle income	b2	-1.30
incomeUpper middle income	b3	-2.25
incomeHigh income	b4	-2.65

The first level, `Low income`, is the “baseline” group. The average fertility rate for `Low income` UN member states is 4.28. Similarly, the average fertility rate for `Upper middle income` member states is $4.28 + -2.25 = 2.03$.

We are now ready to study the multiple linear regression model with interactions shown in Figure 6.1. In this figure we can identify three different effects. First, for any fixed level of life expectancy, observe that there are four different fertility rates. They represent the effect of the categorical explanatory variable, `income`. Second, for any given regression line, the slope represents the change in average fertility rate due to changes on life expectancy. This is the effect of the numerical explanatory variable `life_exp`. Third, observe that the slope of the line depends on the income level; as an illustration, observe that for `High income` member states the slope is less steep than for `Low income` member states. When the slope changes due to changes in the explanatory variable, we call this an **interaction** effect.

The mathematical formulation of the linear regression model with two explanatory variables, one numerical and one categorical, and interactions is:

$$\begin{aligned}\hat{y} = \text{fert rate} = & b_0 + b_{02}D_2 + b_{03}D_3 + b_{04}D_4 \\ & + b_1x \\ & + b_{12}xD_2 + b_{13}xD_3 + b_{14}xD_4\end{aligned}$$

The linear regression shows how the average life expectancy is affected by the categorical variable, the numerical variable, and the interaction effects. There are eight coefficients in our model and we have separated their coefficients into three lines to highlight their different roles. The first line shows the intercept and the effects of the categorical explanatory variables. Recall that D_2 , D_3 and D_4 are the dummy variables in the model and each is equal to one or zero depending the category of the country at hand; correspondingly, the coefficients b_{02} , b_{03} , and b_{04} are the offsets with respect to the baseline level of the intercept, b_0 . Recall that the first dummy variable has been dropped and the intercept captures this effect. The second line in the equation represent the effect of the numerical variable, x . In our example x is the value of life expectancy. The coefficient b_1 is the slope of the line and represents the change in fertility rate due to one unit change in life expectancy. The third line in the equation represents the interaction effects on the slopes. Observe that they are a combination of life expectancy, x , and income level, D_2 , D_3 , and D_4 . What these interaction effects do is to modify the slope for different levels of income. For example, for a `Low income` member state, the dummy variables are $D_1 = 1$, $D_2 = D_3 = D_4 = 0$ and our linear regression is:

$$\begin{aligned}\hat{y} &= \widehat{\text{fert rate}} = b_0 + b_{02} \cdot 0 + b_{03} \cdot 0 + b_{04} \cdot 0 + b_1 x + b_{12} x \cdot 0 + b_{13} x \cdot 0 + b_{14} x \cdot 0 \\ &= b_0 + b_1 x\end{aligned}$$

Similarly, for a `High income` member state, the dummy variables are $D_1 = D_2 = D_3 = 0$, and $D_4 = 1$. We take into account the offsets for the intercept, b_{04} , and the slope, b_{14} , and the linear regression becomes:

$$\begin{aligned}\hat{y} &= \widehat{\text{fert rate}} = b_0 + b_{02} \cdot 0 + b_{03} \cdot 0 + b_{04} \cdot 1 + b_1 x + b_{12} x \cdot 0 + b_{13} x \cdot 0 + b_{14} x \cdot 1 \\ &= b_0 + b_{04} + b_1 x + b_{14} x \\ &= (b_0 + b_{04}) + (b_1 + b_{14}) \cdot x\end{aligned}$$

Observe how the intercept and the slope are different for a `High income` member state when compared to the baseline `Low income` member state. As an illustration, we construct this multiple linear regression for the UN member state data set in R. We first “fit” the model using the `lm()` “linear model” function and then obtain the coefficients using the function `coef()`. In R, the formula used is `y ~ x1 + x2 + x1:x2` where `x1` and `x2` are the variable names in the data set and represent the main effects while `x1:x2` is the interaction term. For simplicity, we can also write `y ~ x1 * x2` as the `*` sign accounts for both, main effects and interaction effects. R would let both `x1` and `x2` be either explanatory or numerical, and we need to make sure the data set format is appropriate for the regression we want to run. Here is the code for our example:

```
# Fit regression model:
model_int <- lm(fert_rate ~ life_exp * income, data = UN_data_ch6)
```

```
# Get the coefficients of the model
coef(model_int)
```

TABLE 6.2: Regression table for interaction model

	Coefficients	Values
(Intercept)	b0	11.918
incomeLower middle income	b02	-1.504
incomeUpper middle income	b03	-1.893
incomeHigh income	b04	-6.580
life_exp	b1	-0.118
incomeLower middle income:life_exp	b12	0.013
incomeUpper middle income:life_exp	b13	0.011
incomeHigh income:life_exp	b14	0.072

We can match the coefficients with the values obtained: the fitted fertility rate $\hat{y} = \text{fert rate}$ for Low income countries is

$$\widehat{\text{fert rate}} = b_0 + b_1 \cdot x = 11.92 + -0.12 \cdot x,$$

which is the equation of the regression line in Figure 6.1 for low income countries. The regression has an intercept of 11.92 and a slope of -0.12. Since life expectancy is greater than zero for all countries, the intercept has no practical interpretation and we only need it to produce the most appropriate line. The interpretation of the slope is the following: for Low income countries, every additional year of life expectancy reduces the average fertility rate by 0.12 units.

As discussed earlier, the intercept and slope for all the other income groups are obtained by taking into account the appropriate offsets. As an illustration, for High income countries $D_4 = 1$ and all other dummy variables are equal to zero. The regression line becomes

$$\hat{y} = \widehat{\text{fert rate}} = b_0 + b_1 x + b_{04} + b_{14} x = (b_0 + b_{04}) + (b_1 + b_{14}) x$$

where x is life expectancy, `life_exp`. The intercept is $b_0 + b_{04} = '(\text{Intercept}) + \text{incomeHighincome}' = 11.92 + (-6.58) = 5.34$, and the slope for these High income countries is $\text{life_exp} + \text{life_exp:incomeHigh income}$ $b_1 + b_{14} = -0.12 + 0.07 = -0.05$. For High income countries, every additional year of life expectancy reduces the average fertility rate by 0.05 units. The intercepts and slopes for other income levels are obtained similarly.

Since the life expectancy for Low income countries has a steeper slope than High income countries, one additional year of life expectancy will decrease fertility rates more for the low income group than for the high income group. This is consistent with our observation from Figure 6.1. When the associated effect of one variable depends on

the value of another variable we say that there is an interaction effect. This is the reason why the regression slopes are different for different income groups.

6.1.3 A model without interactions

We can simplify the previous model by removing the interaction effects. The model still represents different income groups with different regression lines by allowing different intercepts but all the lines have the same slope: they are parallel as shown in Figure 6.2.

To plot parallel slopes we use the function `geom_parallel_slopes()` that is included in the `moderndive` package. To use this function you need to load both the `ggplot2` and `moderndive` packages. Observe how the code is identical to the one used for the model with interactions in Figure 6.1, but now the `geom_smooth(method = "lm", se = FALSE)` layer is replaced with `geom_parallel_slopes(se = FALSE)`.

```
ggplot(UN_data_ch6, aes(x = life_exp, y = fert_rate, color = income)) +
  geom_point() +
  labs(x = "Life expectancy", y = "Fertility rate", color = "Income group") +
  geom_parallel_slopes(se = FALSE)
```

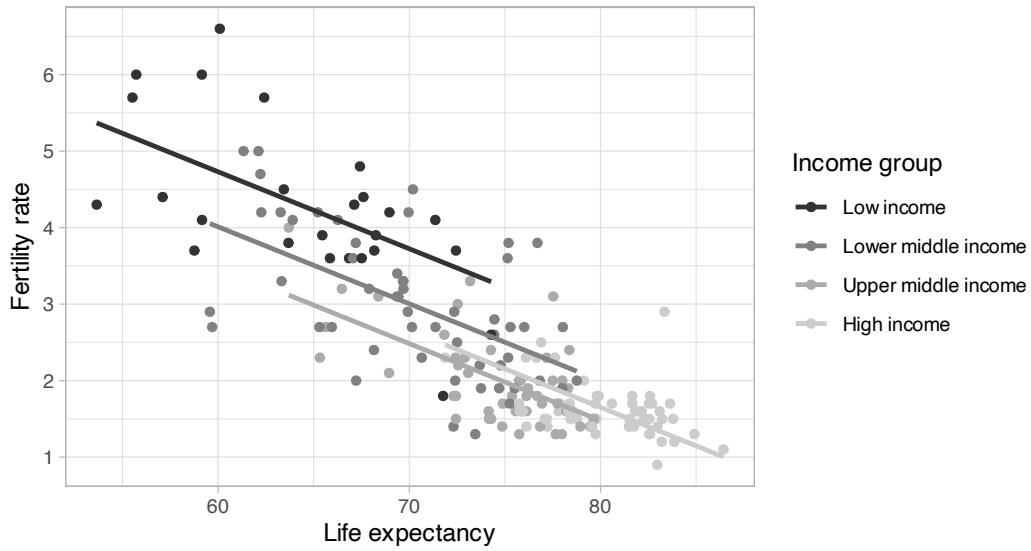


FIGURE 6.2: Parallel slopes model of score with age and gender.

The regression lines for each income group are shown in Figure 6.2. Observe that the lines are now parallel: they all have the same negative slope. The interpretation of this result is that the change in fertility rate due to changes in life expectancy in a given country are the same regardless the income group of this country.

On the other hand, any two regression lines in Figure 6.2 have different intercepts representing the income group; in particular, observe that for any fixed level of life expectancy the fertility rate is greater for `Low income` and `Lower middle income` countries than for `Upper middle income` and `High income` countries.

The mathematical formulation of the linear regression model with two explanatory variables, one numerical and one categorical, and without interactions is:

$$\hat{y} = b_0 + b_{02}D_2 + b_{03}D_3 + b_{04}D_4 + b_1x.$$

Observe that the dummy variables only affect the intercept now, and the slope is fully described by b_1 for any income group. In the UN data example, a `High income` country, with $D_4 = 1$ and the other dummy variables equal to zero, will be represented by

$$\hat{y} = (b_0 + b_{04}) + b_1x.$$

To obtain the coefficients for this regression in R, the formula used is `y ~ x1 + x2` where `x1` and `x2` are the variable names in the data set and represent the main effects. Observe that the term `x1:x2` representing the interaction is no longer included. R would let both `x1` and `x2` to be either explanatory or numerical; therefore, we should always check that the variable format is appropriate for the regression we want to run. Here is the code for the UN data example:

```
# Fit regression model:
model_no_int <- lm(fert_rate ~ life_exp + income, data = UN_data_ch6)

# Get the coefficients of the model
coef(model_no_int)
```

TABLE 6.3: Regression table for a model without interactions

	Coefficients	Values
(Intercept)	b0	10.768
incomeLower middle income	b02	-0.719
incomeUpper middle income	b03	-1.239
incomeHigh income	b04	-1.067
life_exp	b1	-0.101

In this model without interactions, the slope is the same for all the regression lines, $b_1 = -0.101$. Assuming that this model is correct, for any UN member state, every additional year of life expectancy reduces the average fertility rate by 0.101 units, regardless of the income level of the member state. The intercept of the regression line for `Low income` member states is 10.768 while for `High income` member states is $10.768 - 1.067 = 9.701$. The intercepts for other income levels can be obtained similarly. We compare the visualizations for both models side-by-side in Figure 6.3.

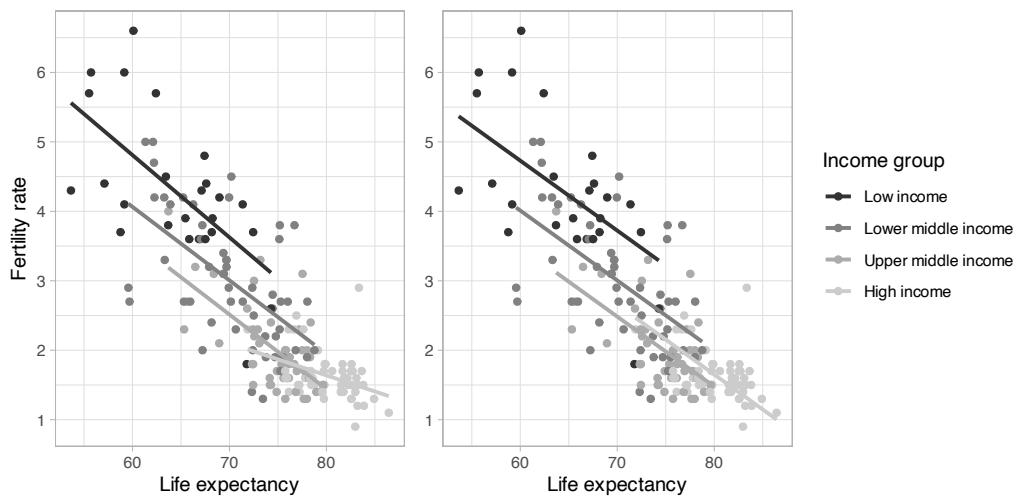


FIGURE 6.3: Comparison of interaction and parallel slopes models.

Which one is the preferred model? Looking at the scatterplot and the clusters of points in Figure 6.3, it does appear that lines with different slopes capture better the behavior of different groups of points. The lines do not appear to be parallel and the interaction model seems more appropriate.

6.1.4 Observed responses, fitted values and residuals

In this subsection, we work with the regression model with interactions. The coefficients for this model were obtained earlier, saved in `model_int`, and shown below:

TABLE 6.4: Regression table for interaction model

	Coefficients	Values
(Intercept)	b0	11.918
incomeLower middle income	b02	-1.504
incomeUpper middle income	b03	-1.893
incomeHigh income	b04	-6.580
life_exp	b1	-0.118
incomeLower middle income:life_exp	b12	0.013
incomeUpper middle income:life_exp	b13	0.011
incomeHigh income:life_exp	b14	0.072

We can use these coefficient to obtain the fitted values and residuals for any given observation. As an illustration, we chose two observations from the UN member states data set, provided the values for the explanatory variables and response, as well as the fitted values and residuals:

ID	fert_rate	income	life_exp	fert_rate_hat	residual
1	1.3	High income	79.7	1.65	-0.35
2	5.7	Low income	62.4	4.52	1.18

The first observation is a `High income` country with a life expectancy of 79.74 years and an observed fertility rate equal to 1.3. The second observation is a `Low income` country with a life expectancy of 62.41 years and an observed fertility rate equal to 5.7. The fitted value, \hat{y} , called `fert_rate_hat` in the table, is the estimated value of the response determined by the regression line. This value is obtained by using the values of the explanatory variables and the coefficients of the linear regression. In addition, recall the difference between the observed response value and the fitted value, $y - \hat{y}$, is called the residual.

We illustrate this in Figure 6.4. The vertical line on the left represents the life expectancy value for the `Low income` country. The y-value for the large dot on the regression line that intersects the vertical line is the fitted value for fertility rate, \hat{y} , and the y-value for the large dot above the line is the observed fertility rate, y . The difference between these values, $y - \hat{y}$, is called the residual and in this case is positive. Similarly, the vertical line on the right represents the life expectancy value for the `High income` country, the y-value for the large dot on the regression line is the fitted fertility rate. The observed y-value for fertility rate is below the regression line making the residual negative.

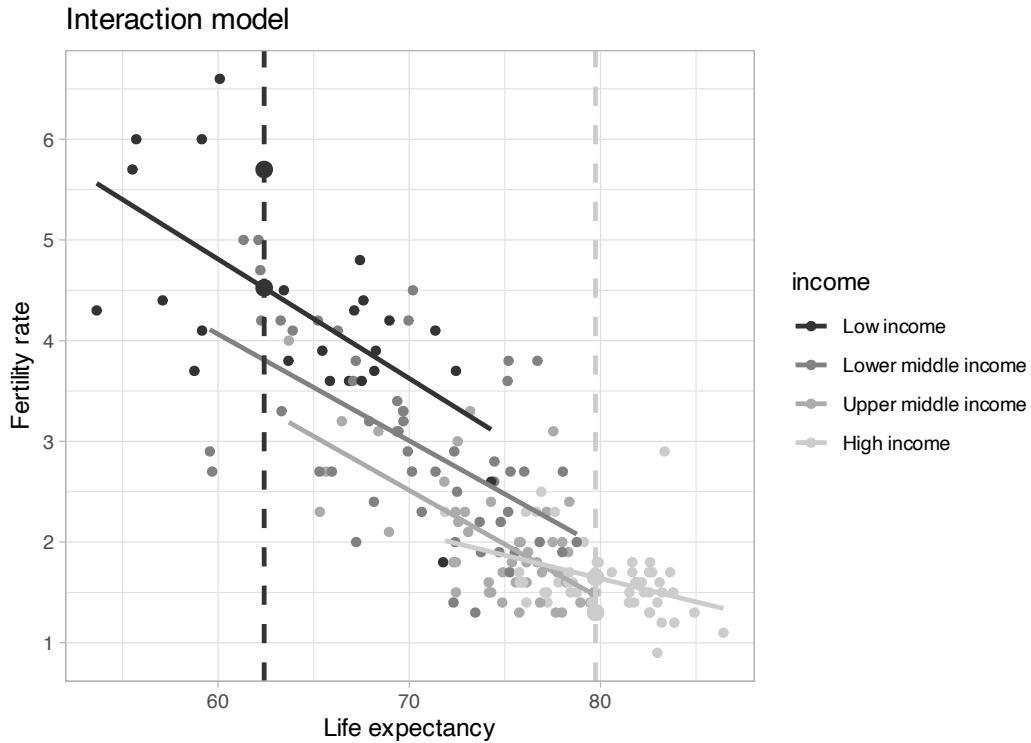


FIGURE 6.4: Fitted values for two new professors.

We can generalize the study of fitted values and residuals for all the countries in the `UN_data_ch6` data set, as shown in Table 6.5.

```
regression_points <- get_regression_points(model_int)
regression_points
```

TABLE 6.5: Regression points (First 10 out of 182 countries)

ID	fert_rate	income	life_exp	fert_rate_hat	residual
1	4.3	Low income	53.6	5.56	-1.262
2	1.4	Upper middle income	79.5	1.50	-0.098
3	2.7	Lower middle income	78.0	2.16	0.544
4	5.0	Lower middle income	62.1	3.84	1.159
5	1.6	High income	77.8	1.74	-0.139
6	1.9	Upper middle income	78.3	1.62	0.278
7	1.6	Upper middle income	76.1	1.86	-0.256
8	1.6	High income	83.1	1.50	0.105
9	1.5	High income	82.3	1.53	-0.033
10	1.6	Upper middle income	74.2	2.07	-0.469

Learning check

(LC6.1) Compute the observed response values, fitted values, and residuals for the model without interactions.

6.2 Two numerical explanatory variables

We now consider regression models with two numerical explanatory variables. To illustrate this situation we use the `Credit` data set from the `ISLR2` R package. This data set contains simulated information for 400 customers. For the regression model we use the credit card balance (`Balance`) as the response variable; and the credit limit (`Limit`), and the income (`Income`) as the numerical explanatory variables.

6.2.1 Exploratory data analysis

We load the `Credit` data frame and construct a new data frame `credit_ch6` with only the variables needed. We do this by using the `select()` verb as we did in Subsection

3.8.1 and, in addition, we save the selecting variables with different names: Balance becomes debt, Limit becomes credit_limit, and Income becomes income. Here is the code:

```
library(ISLR2)
credit_ch6 <- Credit |> as_tibble() |>
  select(debt = Balance, credit_limit = Limit,
         income = Income, credit_rating = Rating, age = Age)
```

You can observe the effect of our use of `select()` but looking at the raw values either in RStudio's spreadsheet viewer or by using `glimpse()`.

```
glimpse(credit_ch6)
```

```
Rows: 400
Columns: 5
$ debt      <dbl> 333, 903, 580, 964, 331, 1151, 203, 872, 279, 1350,~
$ credit_limit <dbl> 3606, 6645, 7075, 9504, 4897, 8047, 3388, 7114, 330~
$ income     <dbl> 14.9, 106.0, 104.6, 148.9, 55.9, 80.2, 21.0, 71.4, ~
$ credit_rating <dbl> 283, 483, 514, 681, 357, 569, 259, 512, 266, 491, 5~
$ age        <dbl> 34, 82, 71, 36, 68, 77, 37, 87, 66, 41, 30, 64, 57,~
```

Furthermore, we present a random sample of five out of the 400 credit card holders in Table 6.6. As observed before, each time you run this code a different subset of five rows is obtained.

```
credit_ch6 |> sample_n(size = 5)
```

TABLE 6.6: Random sample of 5 credit card holders

debt	credit_limit	income	credit_rating	age
1809	13414	186.6	949	41
126	4411	58.4	326	85
580	4632	21.8	355	50
1404	5533	26.4	433	50
50	3327	35.0	253	54

Note that income is in thousands of dollars while debt and credit limit are in dollars. We can also compute summary statistics using the `tidy_summary()` function. We only `select()` the columns of interest for our model:

```
credit_ch6 |>
  select(debt, credit_limit, income) |>
  tidy_summary()

# A tibble: 3 × 11
  column     n group type   min    Q1    mean median    Q3    max    sd
  <chr>   <int> <chr> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 debt      400 <NA>  nume~  0     68.8  520.  460.  863   1999   460.
2 credit~   400 <NA>  nume~  855   3088  4736. 4622. 5873. 13913  2308.
3 income    400 <NA>  nume~  10.4   21.0   45.2   33.1   57.5   187.   35.2
```

The mean and median credit card `debt` are \$520.0 and \$459.5, respectively. The first quartile for `debt` is 68.8; this means that 25% of card holders had debts of \$68.80 or less. Correspondingly, the mean and median credit card limit, `credit_limit`, are around \$4,736 and \$4,622, respectively. Note also that the third quartile of `income` is 57.5; so 75% of card holders had incomes below \$57,500.

We visualize the relationship of the response variable with each of the two explanatory variables using the R code below. There plots are shown in Figure 6.5.

```
ggplot(credit_ch6, aes(x = credit_limit, y = debt)) +
  geom_point() +
  labs(x = "Credit limit (in $)", y = "Credit card debt (in $)",
       title = "Debt and credit limit") +
  geom_smooth(method = "lm", se = FALSE)

ggplot(credit_ch6, aes(x = income, y = debt)) +
  geom_point() +
  labs(x = "Income (in $1000)", y = "Credit card debt (in $)",
       title = "Debt and income") +
  geom_smooth(method = "lm", se = FALSE)
```

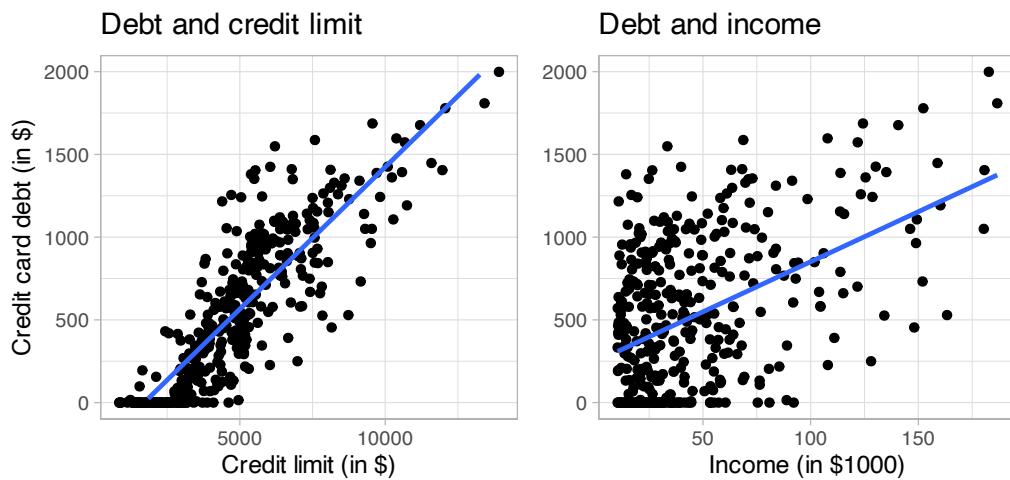


FIGURE 6.5: Relationship between credit card debt and credit limit/income.

The left plot in Figure 6.5 shows a positive and linear association between credit limit and credit card debt: as credit limit increases so does credit card debt. Observe also that many customers have no credit card debt and there is a cluster of points at the credit card debt value of zero. The right plot in Figure 6.5 shows also positive and somewhat linear association between income and credit card debt, but this association seems weaker and actually appears positive only for incomes larger than \$50,000. For lower income values it is not clear there is any association at all.

Since variables `debt`, `credit_limit`, and `income` are numerical, and more importantly, the associations between the response and explanatory variables appear to be linear or close to linear, we can also obtain the correlation coefficient between any two of these variables. Recall that the correlation coefficient is appropriate if the association between the variables is linear. One way to do this is using the `get_correlation()` command as seen in Subsection 5.1.1, once for each explanatory variable with the response `debt`:

```
credit_ch6 |> get_correlation(debt ~ credit_limit)
credit_ch6 |> get_correlation(debt ~ income)
```

Alternatively, using the `select()` verb and command `cor()` we can obtain all correlations simultaneously by returning a *correlation matrix* as shown in Table 6.7. This matrix shows the correlation coefficient for any pair of variables in the appropriate row/column combination.

```
credit_ch6 |>
  select(debt, credit_limit, income) |>
  cor()
```

TABLE 6.7: Correlation coefficients between credit card debt, credit limit, and income

	debt	credit_limit	income
debt	1.000	0.862	0.464
credit_limit	0.862	1.000	0.792
income	0.464	0.792	1.000

We describe some findings presented in the correlation matrix:

1. The diagonal values are all 1 because, based on the definition of the correlation coefficient, the correlation of a variable with itself is always 1.
2. The correlation between `debt` and `credit_limit` is 0.862. This indicates a strong and positive linear relationship: the greater the credit limit is, the larger is the credit card debt, on average.
3. The correlation between `debt` and `income` is 0.464. The linear relationship is positive albeit somewhat weak. In other words, higher income is only weakly associated to higher debt.
4. Observe also that the correlation coefficient between the two explanatory variables, `credit_limit` and `income`, is 0.792.

A useful property of the correlation coefficient is that it is *invariant to linear transformations*; this means that the correlation between two variables, x and y , will be the same as the correlation between $(a \cdot x + b)$ and y for any constants a and b . To illustrate this, observe that the correlation coefficient between `income` in *thousands of dollars* and credit card `debt` was 0.464. If we now find the correlation income in *dollars*, by multiplying `income` by 1000, and credit card `debt` we get:

```
credit_ch6 |> get_correlation(debt ~ 1000 * income)
```

cor
0.464

The correlation is exactly the same.

We return to our exploratory data analysis of the multiple regression. The plots in Figure 6.5 and the corresponding the response and each of the explanatory variables *separately*. In Figure 6.6 we show a 3-dimensional (3D) scatterplot representing the

joint relationship of all three variables simultaneously. Each of the 400 observations in the `credit_ch6` data frame are marked with a point where

1. The response variable y , `debt`, is on the vertical axis.
2. The regressors x_1 , `income`, and x_2 , `credit_limit`, are on the two axes that form the bottom plane.

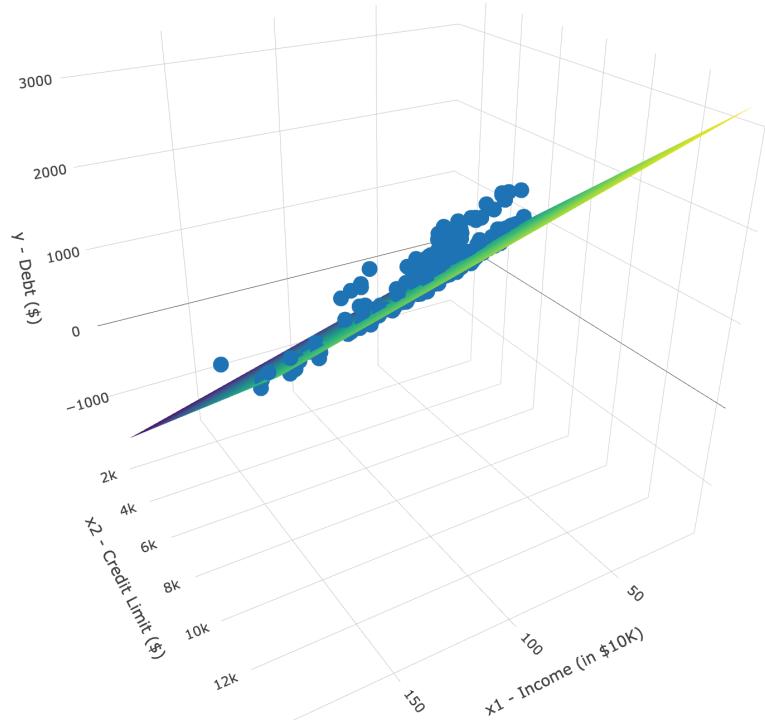


FIGURE 6.6: 3D scatterplot and regression plane.

In addition, Figure 6.6 includes a *regression plane*. Recall from Subsection 5.3.2 that the linear regression with one numerical explanatory variable selects the “best-fitting” line: the line that minimizes the *sum of squared residuals*. When linear regression is performed with two numerical explanatory variables, the solution is a “best-fitting” plane: the plane that minimizes the sum of squared residuals. Visit this website¹ to open an interactive version of this plot in your browser.

Learning check

(LC6.2) Conduct a new exploratory data analysis with the same outcome variable y `debt` but with `credit_rating` and `age` as the new explanatory variables x_1 and x_2 .

¹<https://moderndive.com/regression-plane-ISLR2>

What can you say about the relationship between a credit card holder's debt and their credit rating and age?

6.2.2 Multiple regression with two numerical regressors

As shown in Figure 6.6, the linear regression with two numerical regressors produces the “best-fitting” plane. We start with a model with no interactions for the two numerical explanatory variables `income` and `credit_limit`. In R we consider a model fit with a formula of the form $y \sim x_1 + x_2$. We obtain the regression coefficients using the `lm()` function and the command `coef()` to obtain the coefficients of the linear regression. The regression coefficients are shown in what follows.

```
debt_model <- lm(debt ~ credit_limit + income, data = credit_ch6)
coef(debt_model)
```

We present these results on a table with the mathematical notation used above:

	Coefficients	Values
(Intercept)	b0	-385.179
credit_limit	b1	0.264
income	b2	-7.663

1. We obtain the linear regression coefficients using `lm(y ~ x1 + x2, data)` where x_1 and x_2 are the two numerical explanatory variables used.
2. We extract the coefficients from the output using the `coef()` command.

We interpret the coefficients obtained. The `intercept` value is -\$385.179. If the range of values that the regressors could take include a `credit_limit` of \$0 and an `income` of \$0, the intercept would represent the average credit card debt for an individual with those levels of `credit_limit` and `income`. This is not the case in our data and the intercept has no practical interpretation; it is mainly used to determine where the plane should cut the y -intercept to produce the smallest sum of squared residuals.

Each slope in a multiple linear regression is considered a partial slope and represents the marginal or additional contribution of a regressor when it is added to a model that already contains other regressors. This partial slope is typically different than the slope we may find in a simple linear regression for the same regressor. The reason is that, typically, regressors are correlated, so when one regressor is part of a model, indirectly it's also explaining part of the other regressor. When the second regressor is added to the model, it helps explain only changes in the response that were not already accounted for by the first regressor. For example, the slope for `credit_limit` is

\$0.264. Keeping `income` fixed to some value, for an additional increase of credit limit by one dollar the credit debt increases, on average, by \$0.264. Similarly the slope of `income` is -\$7.663. Keeping `credit_limit` fixed to some level, for a one unit increase of `income` (\$1000 in actual income), there is an associated decrease of \$7.66 in credit card debt, on average.

Putting these results together, the equation of the regression plane that gives us fitted values $\hat{y} = \widehat{\text{debt}}$ is:

$$\begin{aligned}\hat{y} &= \widehat{\text{debt}} = b_0 + b_1 \cdot x_1 + b_2 \cdot x_2 \\ &= -385.179 + 0.263 \cdot x_1 - 7.663 \cdot x_2\end{aligned}$$

where x_1 represents credit limit and x_2 income.

To illustrate the role of partial slopes further, observe that the right plot in Figure 6.5 shows the relationship between `debt` and `income` in isolation, a *positive* relationship, so the slope of `income` is positive. We can obtain the value of this slope by constructing a simple linear regression using `income` as the only regressor:

```
# Fit regression model:
simple_model <- lm(debt ~ income, data = credit_ch6)

# Get the coefficients of the model
coef(simple_model)
```

	Coefficients	Values
(Intercept)	b0'	246.51
income	b2'	6.05

The regression line is given by the following with the coefficients denoted using the prime ('') designation since they are different values than what we saw previously

$$\begin{aligned}\hat{y} &= \widehat{\text{debt}} = b'_0 + b'_2 \cdot x_2 \\ &= 246.515 + 6.048 \cdot x_2\end{aligned}$$

where x_2 is `income`. By contrast, when `credit_limit` and `income` are considered *jointly* to explain changes in `debt`, the equation for the multiple linear regression was:

$$\begin{aligned}\hat{y} &= \widehat{\text{debt}} = b_0 + b_1 \cdot x_1 + b_2 \cdot x_2 \\ &= -385.179 + 0.263 \cdot x_1 - 7.663 \cdot x_2\end{aligned}$$

So the slope for `income` in a simple linear regression is 6.048, and the slope for `income` in a multiple linear regression is -7.663. As surprising as these results may appear at first, they are perfectly valid and consistent as the slope of a simple linear regression has a different role than the partial slope of a multiple linear regression. The latter is the additional effect of `income` on `debt` when `credit_limit` has already been taken into account.

Learning check

(LC6.3) Fit a new simple linear regression using `lm(debt ~ credit_rating + age, data = credit_ch6)` where `credit_rating` and `age` are the new numerical explanatory variables x_1 and x_2 . Get information about the “best-fitting” regression plane from the regression table by applying the `get_regression_table()` function. How do the regression results match up with the results from your previous exploratory data analysis?

6.2.3 Observed/fitted values and residuals

As shown in Subsection 6.1.4 for the UN member states example, we find the fitted values and residuals for our credit card debt regression model. The fitted values for the credit card debt, $\hat{\text{debt}}$, are computed using the equation for the regression plane obtained in the previous section:

$$\hat{y} = \widehat{\text{debt}} = -385.179 + 0.263 \cdot x_1 - 7.663 \cdot x_2$$

where x_1 is `credit_limit` and x_2 is `income`. The residuals are the difference between the observed credit card debt and the fitted credit card debt, $y - \hat{y}$, for each observation in the data set. In R, we find the fitted values, `debt_hat`, and residuals, `residual`, using the `get_regression_points()` function. In Table 6.8 we present the first 10 rows of output. Remember that the coordinates of each of the points in our 3D scatterplot in Figure 6.6 can be found in the `income`, `credit_limit`, and `debt` columns.

```
get_regression_points(debt_model)
```

6.3 Conclusion

6.3.1 Additional resources

Solutions to all *Learning checks* can be found online in Appendix D².

An R script file of all R code used in this chapter is available at <https://www.moderndive.com/scripts/06-multiple-regression.R>.

²<https://moderndive.com/D-appendixD.html>

TABLE 6.8: Regression points (First 10 credit card holders out of 400)

ID	debt	credit_limit	income	debt_hat	residual
1	333	3606	14.9	454	-120.8
2	903	6645	106.0	559	344.3
3	580	7075	104.6	683	-103.4
4	964	9504	148.9	986	-21.7
5	331	4897	55.9	481	-150.0
6	1151	8047	80.2	1127	23.6
7	203	3388	21.0	349	-146.4
8	872	7114	71.4	948	-76.0
9	279	3300	15.1	371	-92.2
10	1350	6819	71.1	873	477.3

6.3.2 What's to come?

This chapter concludes the “Statistical/Data Modeling with `moderndive`” portion of this book. We are ready to proceed to Part III: “Statistical Inference with `infer`.“ Statistical inference is the science of inferring about some unknown quantity using sampling. So far, we have only studied the regression coefficients and their interpretation. In future chapters we learn how we can use information obtained from a sample to make inferences about the entire population.

Once we have covered Chapters 7 on sampling, 8 on confidence intervals, and 9 on hypothesis testing, we revisit the regression models in Chapter 10 on inference for regression. This will complete the topics we want to cover in this book, as shown in Figure 6.7!

Furthermore in Chapter 10, we revisit the concept of residuals $y - \hat{y}$ and discuss their importance when interpreting the results of a regression model. We perform what is known as a *residual analysis* of the residual variable of all `get_regression_points()` outputs. Residual analyses allow you to verify what are known as the *conditions for inference for regression*.

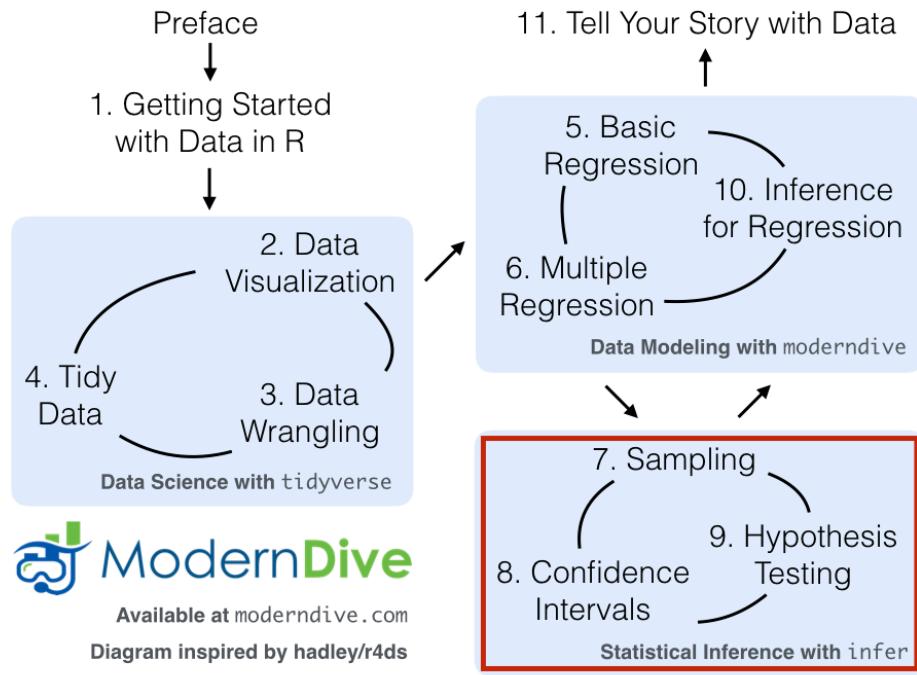


FIGURE 6.7: *ModernDive* flowchart - on to Part III!

Part III

Statistical Inference with *infer*

7

Sampling

In this chapter, we kick off the third portion of this book on statistical inference by learning about *sampling*. The concepts behind sampling form the basis of confidence intervals and hypothesis testing, which we'll cover in Chapters 8 and 9. We will see that the tools that you learned in the data science portion of this book, in particular data visualization and data wrangling, will also play an important role in the development of your understanding. As mentioned before, the concepts throughout this text all build into a culmination allowing you to “tell your story with data.”

Needed packages

Let’s load all the packages needed for this chapter (this assumes you’ve already installed them). Recall from our discussion in Section 4.4 that loading the `tidyverse` package by running `library(tidyverse)` loads the following commonly used data science packages all at once:

- `ggplot2` for data visualization
- `dplyr` for data wrangling
- `tidyR` for converting data to “tidy” format
- `readR` for importing spreadsheet data into R
- As well as the more advanced `purrr`, `tibble`, `stringr`, and `forcats` packages

If needed, read Section 1.3 for information on how to install and load R packages.

```
library(tidyverse)
library(moderndive)
```

7.1 Sampling bowl activity

Let’s start with a hands-on activity.

7.1.1 What proportion of this bowl's balls are red?

Take a look at the bowl in Figure 7.1. It has a certain number of red and a certain number of white balls all of equal size. (Note that in this printed version of the book “red” corresponds to the darker-colored balls, and “white” corresponds to the lighter-colored balls. We kept the reference to “red” and “white” throughout this book since those are the actual colors of the balls as seen in the background of the image on our book’s cover¹.) Furthermore, it appears the bowl has been mixed beforehand, as there does not seem to be any coherent pattern to the spatial distribution of the red and white balls.

Let’s now ask ourselves, what proportion of this bowl’s balls are red?



FIGURE 7.1: A bowl with red and white balls.

One way to answer this question would be to perform an exhaustive count: remove each ball individually, count the number of red balls and the number of white balls, and divide the number of red balls by the total number of balls. However, this would be a long and tedious process.

7.1.2 Using the shovel once

Instead of performing an exhaustive count, let’s insert a shovel into the bowl as seen in Figure 7.2. Using the shovel, let’s remove $5 \cdot 10 = 50$ balls, as seen in Figure 7.3.

¹https://moderndive.com/images/logos/book_cover.png



FIGURE 7.2: Inserting a shovel into the bowl.

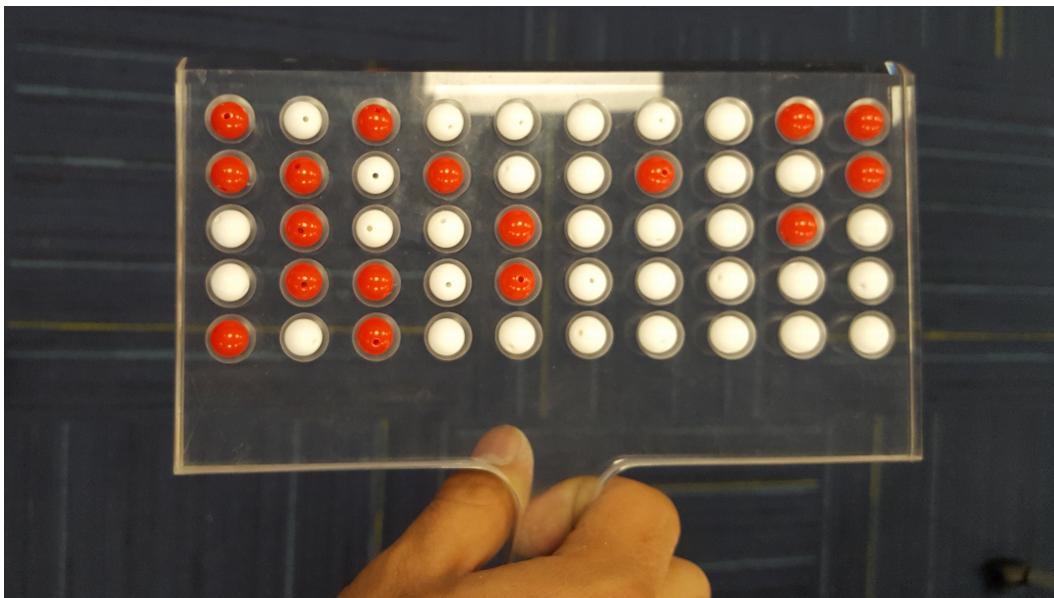


FIGURE 7.3: Removing 50 balls from the bowl.

Observe that 17 of the balls are red and thus $0.34 = 34\%$ of the shovel's balls are red. We can view the proportion of balls that are red in this shovel as a guess of the proportion of balls that are red in the entire bowl. While not as exact as doing an exhaustive count of all the balls in the bowl, our guess of 34% took much less time and energy to make.

However, say, we started this activity over from the beginning. In other words, we replace the 50 balls back into the bowl and start over. Would we remove exactly 17 red balls again? In other words, would our guess at the proportion of the bowl's balls that are red be exactly 34% again? Maybe?

What if we repeated this activity several times following the process shown in Figure 7.4? Would we obtain exactly 17 red balls each time? In other words, would our guess at the proportion of the bowl's balls that are red be exactly 34% every time? Surely not. Let's repeat this exercise several times with the help of 33 groups of friends to understand how the value differs with repetition.

7.1.3 Using the shovel 33 times

Each of our 33 groups of friends will do the following:

- Use the shovel to remove 50 balls each.
- Count the number of red balls and thus compute the proportion of the 50 balls that are red.
- Return the balls into the bowl.
- Mix the contents of the bowl a little to not let a previous group's results influence the next group's.



FIGURE 7.4: Repeating sampling activity 33 times.

Each of our 33 groups of friends make note of their proportion of red balls from their sample collected. Each group then marks their proportion of their 50 balls that were red in the appropriate bin in a hand-drawn histogram as seen in Figure 7.5.

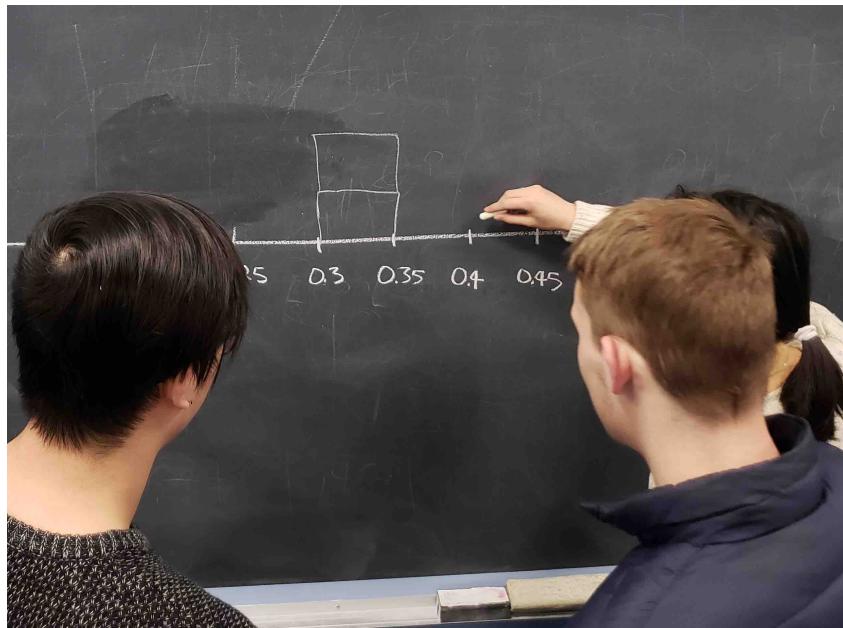


FIGURE 7.5: Constructing a histogram of proportions.

Recall from Section 2.5 that histograms allow us to visualize the *distribution* of a numerical variable. In particular, where the center of the values falls and how the values vary. A partially completed histogram of the first 10 out of 33 groups of friends' results can be seen in Figure 7.6.

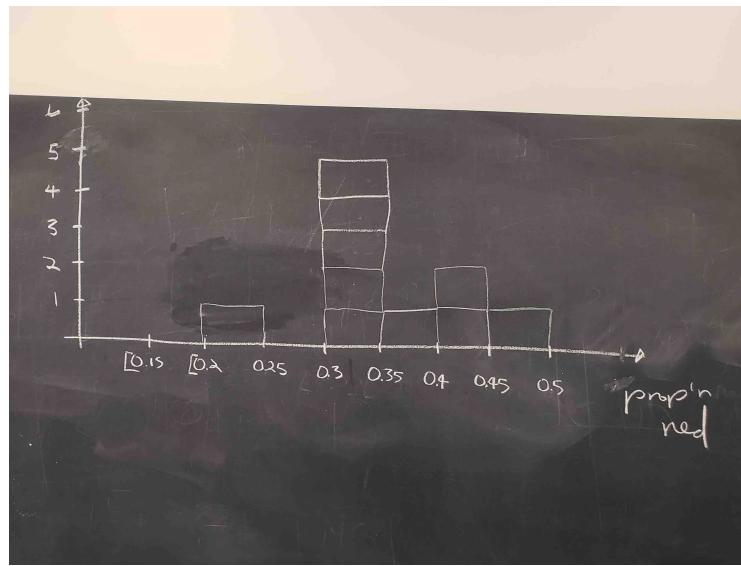


FIGURE 7.6: Hand-drawn histogram of first 10 out of 33 proportions.

Observe the following in the histogram in Figure 7.6:

- At the low end, one group removed 50 balls from the bowl with proportion red between 0.20 and 0.25.
- At the high end, another group removed 50 balls from the bowl with proportion between 0.45 and 0.5 red.
- However, the most frequently occurring proportions were between 0.30 and 0.35 red, right in the middle of the distribution.
- The shape of this distribution is somewhat bell-shaped.

Let's construct this same hand-drawn histogram in R using your data visualization skills that you honed in Chapter 2. We saved our 33 groups of friends' results in the `tactile_prop_red` data frame included in the `moderndive` package. Run the following to display the first 10 of 33 rows:

```
tactile_prop_red
```

```
# A tibble: 33 x 4
  group      replicate red_balls prop_red
  <chr>        <int>     <int>    <dbl>
1 Ilyas, Yohan      1       21    0.42
2 Morgan, Terrance  2       17    0.34
3 Martin, Thomas   3       21    0.42
4 Clark, Frank     4       21    0.42
5 Riddhi, Karina   5       18    0.36
6 Andrew, Tyler    6       19    0.38
7 Julia             7       19    0.38
8 Rachel, Lauren   8       11    0.22
9 Daniel, Caroline 9       15    0.3
10 Josh, Maeve     10      17    0.34
# i 23 more rows
```

Observe for each `group` that we have their names, the number of `red_balls` they obtained, and the corresponding proportion out of 50 balls that were red named `prop_red`. We also have a `replicate` variable enumerating each of the 33 groups. We chose this name because each row can be viewed as one instance of a replicated (in other words repeated) activity: using the shovel to remove 50 balls and computing the proportion of those balls that are red.

Let's visualize the distribution of these 33 proportions using `geom_histogram()` with `binwidth = 0.05` in Figure 7.7. This is a computerized and complete version of the partially completed hand-drawn histogram you saw in Figure 7.6. Note that setting `boundary = 0.4` indicates that we want a binning scheme such that one of the bins' boundary is at 0.4. This helps us to more closely align this histogram with the hand-drawn histogram in Figure 7.6.

```
ggplot(tactile_prop_red, aes(x = prop_red)) +
  geom_histogram(binwidth = 0.05, boundary = 0.4, color = "white") +
  labs(x = "Proportion of 50 balls that were red",
       title = "Distribution of 33 proportions red")
```

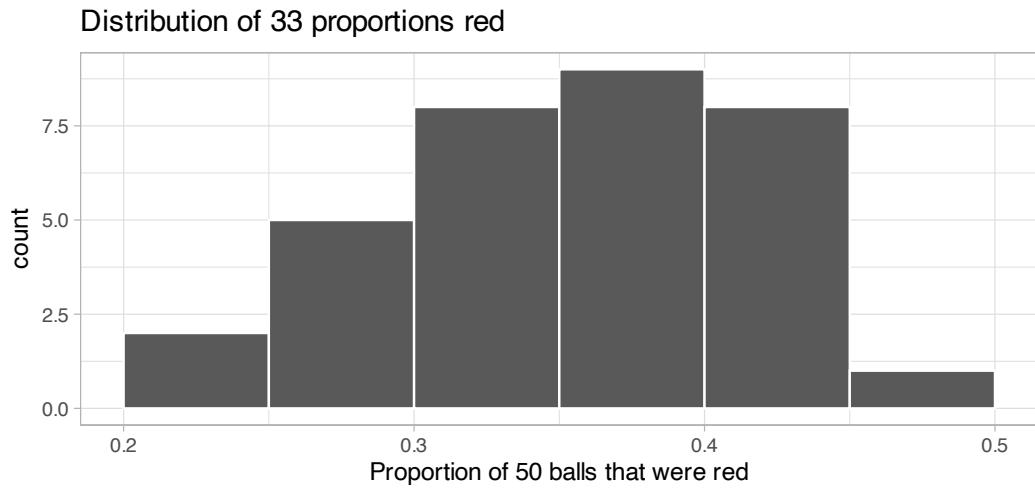


FIGURE 7.7: Distribution of 33 proportions based on 33 samples of size 50.

7.1.4 What did we just do?

What we just demonstrated in this activity is the statistical concept of *sampling*. We would like to know the proportion of the bowl's balls that are red. Because the bowl has a large number of balls, performing an exhaustive count of the red and white balls would be time-consuming. We thus extracted a *sample* of 50 balls using the shovel to make an *estimate*. Using this sample of 50 balls, we estimated the proportion of the *bowl's* balls that are red to be 34%.

Moreover, because we mixed the balls before each use of the shovel, the samples were randomly drawn. Because each sample was drawn at random, the samples were different from each other. Because the samples were different from each other, we obtained the different proportions red observed in Figure 7.7. This is known as the concept of *sampling variation*.

The purpose of this sampling activity was to develop an understanding of two key concepts relating to sampling:

1. Understanding the effect of sampling variation.
2. Understanding the effect of sample size on sampling variation.

In Section 7.2, we'll mimic the hands-on sampling activity we just performed on a computer. This will allow us not only to repeat the sampling exercise much more

than 33 times, but it will also allow us to use shovels with different numbers of slots than just 50.

Afterwards, we'll present you with definitions, terminology, and notation related to sampling in Section 7.3. As in many disciplines, such necessary background knowledge may seem inaccessible and even confusing at first. However, as with many difficult topics, if you truly understand the underlying concepts and practice, practice, practice, you'll be able to master them.

To tie the contents of this chapter to the real world, we'll present an example of one of the most recognizable uses of sampling: polls. In Section 7.4 we'll look at a particular case study: a 2013 poll on then U.S. President Barack Obama's popularity among young Americans, conducted by Kennedy School's Institute of Politics at Harvard University. To close this chapter, we'll generalize the "sampling from a bowl" exercise to other sampling scenarios and present a theoretical result known as the *Central Limit Theorem*.

Learning check

(LC7.1) Why was it important to mix the bowl before we sampled the balls?

(LC7.2) Why is it that our 33 groups of friends did not all have the same numbers of balls that were red out of 50, and hence different proportions red?

7.2 Virtual sampling

In the previous Section 7.1, we performed a *tactile* sampling activity by hand. In other words, we used a physical bowl of balls and a physical shovel. We performed this sampling activity by hand first so that we could develop a firm understanding of the root ideas behind sampling. In this section, we'll mimic this tactile sampling activity with a *virtual* sampling activity using a computer. In other words, we'll use a virtual analog to the bowl of balls and a virtual analog to the shovel.

7.2.1 Using the virtual shovel once

Let's start by performing the virtual analog of the tactile sampling exercise we performed in Section 7.1. We first need a virtual analog of the bowl seen in Figure 7.1. To this end, we included a data frame named `bowl` in the `moderndive` package. The rows of `bowl` correspond exactly with the contents of the actual bowl.

```
bowl
```

```
# A tibble: 2,400 x 2
  ball_ID color
  <int> <chr>
1      1 white
2      2 white
3      3 white
4      4 red
5      5 white
6      6 white
7      7 red
8      8 white
9      9 red
10     10 white
# i 2,390 more rows
```

Observe that `bowl` has 2400 rows, telling us that the bowl contains 2400 equally sized balls. The first variable `ball_ID` is used as an *identification variable* as discussed in Subsection 1.4.4; none of the balls in the actual bowl are marked with numbers. The second variable `color` indicates whether a particular virtual ball is red or white. View the contents of the bowl in RStudio’s data viewer and scroll through the contents to convince yourself that `bowl` is indeed a virtual analog of the actual bowl in Figure 7.1.

Now that we have a virtual analog of our bowl, we next need a virtual analog to the shovel seen in Figure 7.2 to generate virtual samples of 50 balls. We’re going to use the `rep_sample_n()` function included in the `moderndive` package. This function allows us to take repeated, or replicated, `samples` of size `n`.

```
virtual_shovel <- bowl %>%
  rep_sample_n(size = 50)
virtual_shovel
```

```
# A tibble: 50 x 3
# Groups:   replicate [1]
  replicate ball_ID color
  <int>    <int> <chr>
1        1      1970 white
2        1      842  red
3        1     2287 white
4        1      599 white
5        1      108 white
```

```

6      1    846 red
7      1    390 red
8      1    344 white
9      1    910 white
10     1   1485 white
# i 40 more rows

```

Observe that `virtual_shovel` has 50 rows corresponding to our virtual sample of size 50. The `ball_ID` variable identifies which of the 2400 balls from `bowl` are included in our sample of 50 balls while `color` denotes its color. However, what does the `replicate` variable indicate? In `virtual_shovel`'s case, `replicate` is equal to 1 for all 50 rows. This is telling us that these 50 rows correspond to the first repeated/replicated use of the shovel, in our case our first sample. We'll see shortly that when we "virtually" take 33 samples, `replicate` will take values between 1 and 33.

Let's compute the proportion of balls in our virtual sample that are red using the `dplyr` data wrangling verbs you learned in Chapter 3. First, for each of our 50 sampled balls, let's identify if it is red or not using a test for equality with `==`. Let's create a new Boolean variable `is_red` using the `mutate()` function from Section 3.5:

```

virtual_shovel %>%
  mutate(is_red = (color == "red"))

```

```

# A tibble: 50 x 4
# Groups:   replicate [1]
  replicate ball_ID color is_red
  <int>    <int> <chr> <lgl>
1       1    1970 white FALSE
2       1     842 red   TRUE
3       1   2287 white FALSE
4       1     599 white FALSE
5       1     108 white FALSE
6       1     846 red   TRUE
7       1     390 red   TRUE
8       1     344 white FALSE
9       1     910 white FALSE
10      1   1485 white FALSE
# i 40 more rows

```

Observe that for every row where `color == "red"`, the Boolean (logical) value `TRUE` is returned and for every row where `color` is not equal to "red", the Boolean `FALSE` is returned.

Second, let's compute the number of balls out of 50 that are red using the `summarize()` function. Recall from Section 3.3 that `summarize()` takes a data frame with many rows

and returns a data frame with a single row containing summary statistics, like the `mean()` or `median()`. In this case, we use the `sum()`:

```
virtual_shovel %>%
  mutate(is_red = (color == "red")) %>%
  summarize(num_red = sum(is_red))
```

```
# A tibble: 1 × 2
  replicate num_red
  <int>    <int>
1       1      12
```

Why does this work? Because R treats `TRUE` like the number `1` and `FALSE` like the number `0`. So summing the number of `TRUE`s and `FALSE`s is equivalent to summing `1`'s and `0`'s. In the end, this operation counts the number of balls where `color` is `red`. In our case, 12 of the 50 balls were red. However, you might have gotten a different number red because of the randomness of the virtual sampling.

Third and lastly, let's compute the proportion of the 50 sampled balls that are red by dividing `num_red` by 50:

```
virtual_shovel %>%
  mutate(is_red = color == "red") %>%
  summarize(num_red = sum(is_red)) %>%
  mutate(prop_red = num_red / 50)
```

```
# A tibble: 1 × 3
  replicate num_red prop_red
  <int>    <int>    <dbl>
1       1      12     0.24
```

In other words, 24% of this virtual sample's balls were red. Let's make this code a little more compact and succinct by combining the first `mutate()` and the `summarize()` as follows:

```
virtual_shovel %>%
  summarize(num_red = sum(color == "red")) %>%
  mutate(prop_red = num_red / 50)
```

```
# A tibble: 1 x 3
  replicate num_red prop_red
  <int>     <int>     <dbl>
1         1       12      0.24
```

Great! 24% of `virtual_shovel`'s 50 balls were red! So based on this particular sample of 50 balls, our guess at the proportion of the `bowl`'s balls that are red is 24%. But remember from our earlier tactile sampling activity that if we repeat this sampling, we will not necessarily obtain the same value of 24% again. There will likely be some variation. In fact, our 33 groups of friends computed 33 such proportions whose distribution we visualized in Figure 7.6. We saw that these estimates *varied*. Let's now perform the virtual analog of having 33 groups of students use the sampling shovel!

7.2.2 Using the virtual shovel 33 times

Recall that in our tactile sampling exercise in Section 7.1, we had 33 groups of students each use the shovel, yielding 33 samples of size 50 balls. We then used these 33 samples to compute 33 proportions. In other words, we repeated/replicated using the shovel 33 times. We can perform this repeated/replicated sampling virtually by once again using our virtual shovel function `rep_sample_n()`, but by adding the `reps = 33` argument. This is telling R that we want to repeat the sampling 33 times.

We'll save these results in a data frame called `virtual_samples`. While we provide a preview of the first 10 rows of `virtual_samples` in what follows, we highly suggest you scroll through its contents using RStudio's spreadsheet viewer by running `View(virtual_samples)`.

```
virtual_samples <- bowl %>%
  rep_sample_n(size = 50, reps = 33)
virtual_samples
```

```
# A tibble: 1,650 x 3
# Groups:   replicate [33]
  replicate ball_ID color
  <int>     <int> <chr>
1         1       875 white
2         1      1851 red
3         1      1548 red
4         1      1975 white
5         1       835 white
6         1        16 white
7         1       327 white
```

```

8      1    1803 red
9      1     740 red
10     1     179 red
# i 1,640 more rows

```

Observe in the spreadsheet viewer that the first 50 rows of `replicate` are equal to 1 while the next 50 rows of `replicate` are equal to 2. This is telling us that the first 50 rows correspond to the first sample of 50 balls while the next 50 rows correspond to the second sample of 50 balls. This pattern continues for all `reps = 33` replicates and thus `virtual_samples` has $33 \cdot 50 = 1650$ rows.

Let's now take `virtual_samples` and compute the resulting 33 proportions red. We'll use the same `dplyr` verbs as before, but this time with an additional `group_by()` of the `replicate` variable. Recall from Section 3.4 that by assigning the grouping variable “meta-data” before we `summarize()`, we'll obtain 33 different proportions red. We display a preview of the first 10 out of 33 rows:

```

virtual_prop_red <- virtual_samples %>%
  group_by(replicate) %>%
  summarize(red = sum(color == "red")) %>%
  mutate(prop_red = red / 50)
virtual_prop_red

```

```

# A tibble: 33 x 3
  replicate   red prop_red
      <int> <int>    <dbl>
1       1     23    0.46
2       2     19    0.38
3       3     18    0.36
4       4     19    0.38
5       5     15    0.3
6       6     21    0.42
7       7     21    0.42
8       8     16    0.32
9       9     24    0.48
10      10    14    0.28
# i 23 more rows

```

As with our 33 groups of friends' tactile samples, there is variation in the resulting 33 virtual proportions red. Let's visualize this variation in a histogram in Figure 7.8. Note that we add `binwidth = 0.05` and `boundary = 0.4` arguments as well. Recall that setting `boundary = 0.4` ensures a binning scheme with one of the bins' boundaries at 0.4. Since the `binwidth = 0.05` is also set, this will create bins with boundaries at 0.30, 0.35, 0.45, 0.5, etc. as well.

```
ggplot(virtual_prop_red, aes(x = prop_red)) +
  geom_histogram(binwidth = 0.05, boundary = 0.4, color = "white") +
  labs(x = "Proportion of 50 balls that were red",
       title = "Distribution of 33 proportions red")
```

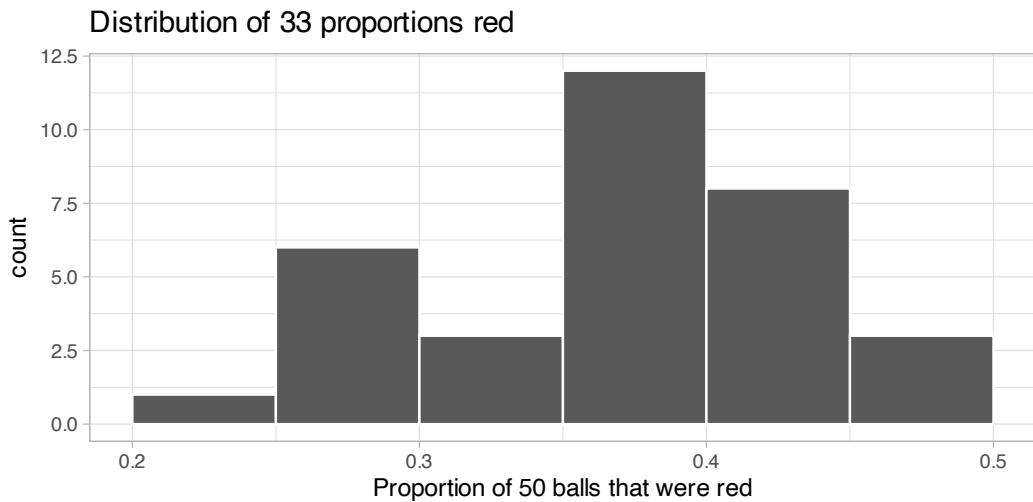


FIGURE 7.8: Distribution of 33 proportions based on 33 samples of size 50.

Observe that we occasionally obtained proportions red that are less than 30%. On the other hand, we occasionally obtained proportions that are greater than 45%. However, the most frequently occurring proportions were between 35% and 40% (for 11 out of 33 samples). Why do we have these differences in proportions red? Because of *sampling variation*.

Let's now compare our virtual results with our tactile results from the previous section in Figure 7.9. Observe that both histograms are somewhat similar in their center and variation, although not identical. These slight differences are again due to random sampling variation. Furthermore, observe that both distributions are somewhat bell-shaped.

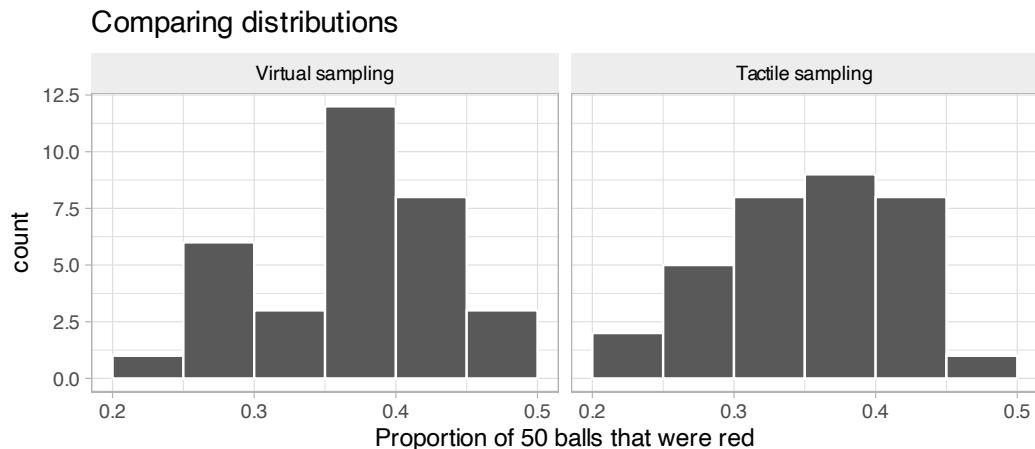


FIGURE 7.9: Comparing 33 virtual and 33 tactile proportions red.

Learning check

(LC7.3) Why couldn't we study the effects of sampling variation when we used the virtual shovel only once? Why did we need to take more than one virtual sample (in our case 33 virtual samples)?

7.2.3 Using the virtual shovel 1000 times

Now say we want to study the effects of sampling variation not for 33 samples, but rather for a larger number of samples, say 1000. We have two choices at this point. We could have our groups of friends manually take 1000 samples of 50 balls and compute the corresponding 1000 proportions. However, this would be a tedious and time-consuming task. This is where computers excel: automating long and repetitive tasks while performing them quite quickly. Thus, at this point we will abandon tactile sampling in favor of only virtual sampling. Let's once again use the `rep_sample_n()` function with `sample_size` set to be 50 once again, but this time with the number of replicates `reps` set to 1000. Be sure to scroll through the contents of `virtual_samples` in RStudio's viewer.

```
virtual_samples <- bowl %>%
  rep_sample_n(size = 50, reps = 1000)
virtual_samples
```

```
# A tibble: 50,000 x 3
# Groups:   replicate [1,000]
  replicate ball_ID color
  <int>    <int> <chr>
1       1      1236 red
2       1      1944 red
3       1      1939 white
4       1       780 white
5       1      1956 white
6       1      1003 white
7       1      2113 white
8       1      2213 white
9       1       782 white
10      1       898 white
# i 49,990 more rows
```

Observe that now `virtual_samples` has $1000 \cdot 50 = 50,000$ rows, instead of the $33 \cdot 50 = 1650$ rows from earlier. Using the same data wrangling code as earlier, let's take the data frame `virtual_samples` with $1000 \cdot 50 = 50,000$ rows and compute the resulting 1000 proportions of red balls.

```
virtual_prop_red <- virtual_samples %>%
  group_by(replicate) %>%
  summarize(red = sum(color == "red")) %>%
  mutate(prop_red = red / 50)
virtual_prop_red
```

```
# A tibble: 1,000 x 3
  replicate red prop_red
  <int> <int>    <dbl>
1       1     18    0.36
2       2     19    0.38
3       3     20    0.4
4       4     15    0.3
5       5     17    0.34
6       6     16    0.32
7       7     23    0.46
8       8     23    0.46
9       9     15    0.3
10      10    18    0.36
# i 990 more rows
```

Observe that we now have 1000 replicates of `prop_red`, the proportion of 50 balls that are red. Using the same code as earlier, let's now visualize the distribution of these 1000 replicates of `prop_red` in a histogram in Figure 7.10.

```
ggplot(virtual_prop_red, aes(x = prop_red)) +  
  geom_histogram(binwidth = 0.05, boundary = 0.4, color = "white") +  
  labs(x = "Proportion of 50 balls that were red",  
       title = "Distribution of 1000 proportions red")
```

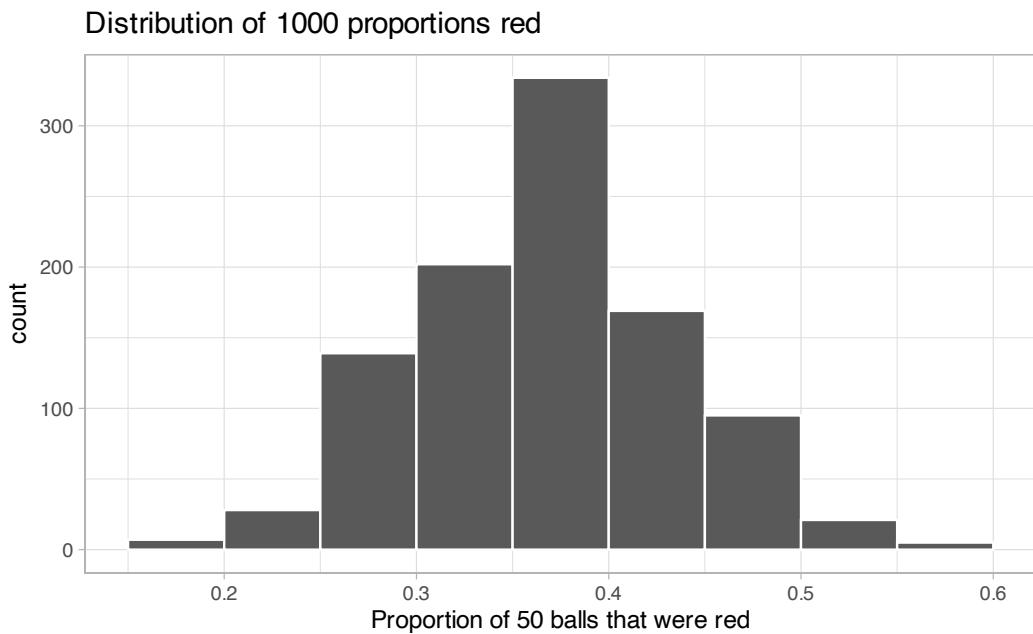


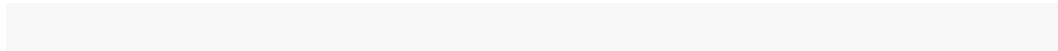
FIGURE 7.10: Distribution of 1000 proportions based on 1000 samples of size 50.

Once again, the most frequently occurring proportions of red balls occur between 35% and 40%. Every now and then, we obtain proportions as low as between 20% and 25%, and others as high as between 55% and 60%. These are rare, however. Furthermore, observe that we now have a much more symmetric and smoother bell-shaped distribution. This distribution is, in fact, approximated well by a normal distribution. At this point we recommend you read the “Normal distribution” section (Appendix A.2) for a brief discussion on the properties of the normal distribution.

Learning check

- (LC7.4) Why did we not take 1000 “tactile” samples of 50 balls by hand?

(LC7.5) Looking at Figure 7.10, would you say that sampling 50 balls where 30% of them were red is likely or not? What about sampling 50 balls where 10% of them were red?



7.2.4 Using different shovels

Now say instead of just one shovel, you have three choices of shovels to extract a sample of balls with: shovels of size 25, 50, and 100.

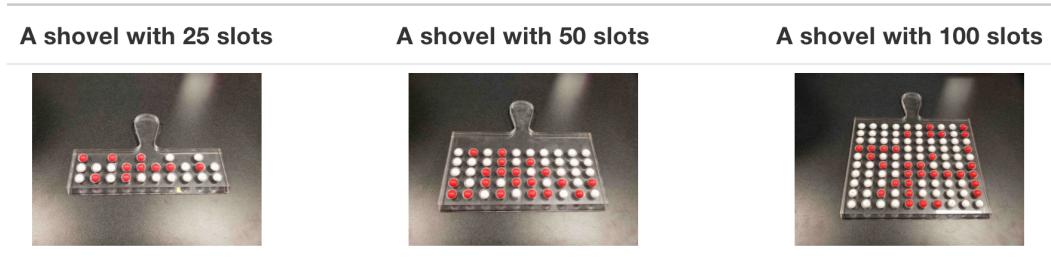


FIGURE 7.11: Three shovels to extract three different sample sizes.

If your goal is still to estimate the proportion of the bowl’s balls that are red, which shovel would you choose? In our experience, most people would choose the largest shovel with 100 slots because it would yield the “best” guess of the proportion of the bowl’s balls that are red. Let’s define some criteria for “best” in this subsection.

Using our newly developed tools for virtual sampling, let’s unpack the effect of having different sample sizes! In other words, let’s use `rep_sample_n()` with `size` set to 25, 50, and 100, respectively, while keeping the number of repeated/replicated samples at 1000:

1. Virtually use the appropriate shovel to generate 1000 samples with `size` balls.
2. Compute the resulting 1000 replicates of the proportion of the shovel’s balls that are red.
3. Visualize the distribution of these 1000 proportions red using a histogram.

Run each of the following code segments individually and then compare the three resulting histograms.

```
# Segment 1: sample size = 25 -----
# 1.a) Virtually use shovel 1000 times
```

```
virtual_samples_25 <- bowl %>%
  rep_sample_n(size = 25, reps = 1000)

# 1.b) Compute resulting 1000 replicates of proportion red
virtual_prop_red_25 <- virtual_samples_25 %>%
  group_by(replicate) %>%
  summarize(red = sum(color == "red")) %>%
  mutate(prop_red = red / 25)

# 1.c) Plot distribution via a histogram
ggplot(virtual_prop_red_25, aes(x = prop_red)) +
  geom_histogram(binwidth = 0.05, boundary = 0.4, color = "white") +
  labs(x = "Proportion of 25 balls that were red", title = "25")

# Segment 2: sample size = 50 -----
# 2.a) Virtually use shovel 1000 times
virtual_samples_50 <- bowl %>%
  rep_sample_n(size = 50, reps = 1000)

# 2.b) Compute resulting 1000 replicates of proportion red
virtual_prop_red_50 <- virtual_samples_50 %>%
  group_by(replicate) %>%
  summarize(red = sum(color == "red")) %>%
  mutate(prop_red = red / 50)

# 2.c) Plot distribution via a histogram
ggplot(virtual_prop_red_50, aes(x = prop_red)) +
  geom_histogram(binwidth = 0.05, boundary = 0.4, color = "white") +
  labs(x = "Proportion of 50 balls that were red", title = "50")

# Segment 3: sample size = 100 -----
# 3.a) Virtually using shovel with 100 slots 1000 times
virtual_samples_100 <- bowl %>%
  rep_sample_n(size = 100, reps = 1000)

# 3.b) Compute resulting 1000 replicates of proportion red
virtual_prop_red_100 <- virtual_samples_100 %>%
  group_by(replicate) %>%
  summarize(red = sum(color == "red")) %>%
  mutate(prop_red = red / 100)
```

```
# 3.c) Plot distribution via a histogram
ggplot(virtual_prop_red_100, aes(x = prop_red)) +
  geom_histogram(binwidth = 0.05, boundary = 0.4, color = "white") +
  labs(x = "Proportion of 100 balls that were red", title = "100")
```

For easy comparison, we present the three resulting histograms in a single row with matching x and y axes in Figure 7.12.

Comparing distributions of proportions red for three different shovel sizes.

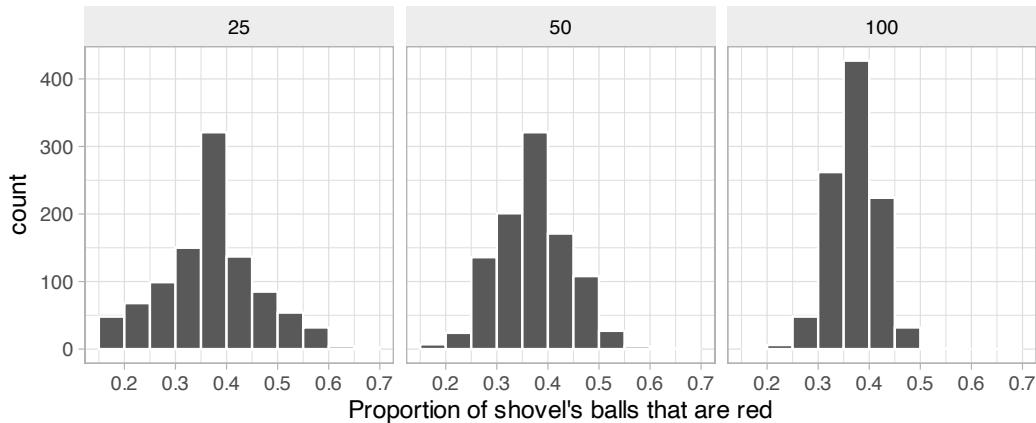


FIGURE 7.12: Comparing the distributions of proportion red for different sample sizes.

Observe that as the sample size increases, the variation of the 1000 replicates of the proportion of red decreases. In other words, as the sample size increases, there are fewer differences due to sampling variation and the distribution centers more tightly around the same value. Eyeballing Figure 7.12, all three histograms appear to center around roughly 40%.

We can be numerically explicit about the amount of variation in our three sets of 1000 values of `prop_red` using the *standard deviation*. A standard deviation is a summary statistic that measures the amount of variation within a numerical variable (see Appendix A.1 for a brief discussion on the properties of the standard deviation). For all three sample sizes, let's compute the standard deviation of the 1000 proportions red by running the following data wrangling code that uses the `sd()` summary function.

```
# n = 25
virtual_prop_red_25 %>%
  summarize(sd = sd(prop_red))
```

```
# n = 50
virtual_prop_red_50 %>%
  summarize(sd = sd(prop_red))

# n = 100
virtual_prop_red_100 %>%
  summarize(sd = sd(prop_red))
```

Let's compare these three measures of distributional variation in Table 7.1.

TABLE 7.1: Comparing standard deviations of proportions red for three different shovels

Number of slots in shovel	Standard deviation of proportions red
25	0.094
50	0.069
100	0.045

As we observed in Figure 7.12, as the sample size increases, the variation decreases. In other words, there is less variation in the 1000 values of the proportion red. So as the sample size increases, our guesses at the true proportion of the bowl's balls that are red get more precise.

Learning check

(LC7.6) In Figure 7.12, we used shovels to take 1000 samples each, computed the resulting 1000 proportions of the shovel's balls that were red, and then visualized the distribution of these 1000 proportions in a histogram. We did this for shovels with 25, 50, and 100 slots in them. As the size of the shovels increased, the histograms got narrower. In other words, as the size of the shovels increased from 25 to 50 to 100, did the 1000 proportions

- A. vary less,
- B. vary by the same amount, or
- C. vary more?

(LC7.7) What summary statistic did we use to quantify how much the 1000 proportions red varied?

- A. The interquartile range
- B. The standard deviation

- C. The range: the largest value minus the smallest.

7.3 Sampling framework

In both our tactile and our virtual sampling activities, we used sampling for the purpose of estimation. We extracted samples in order to *estimate* the proportion of the bowl's balls that are red. We used sampling as a less time-consuming approach than performing an exhaustive count of all the balls. Our virtual sampling activity built up to the results shown in Figure 7.12 and Table 7.1: comparing 1000 proportions red based on samples of size 25, 50, and 100. This was our first attempt at understanding two key concepts relating to sampling for estimation:

1. The effect of *sampling variation* on our estimates.
2. The effect of sample size on *sampling variation*.

Now that you have built some intuition relating to sampling, let's now attach words and labels to the various concepts we've explored so far. Specifically in the next section, we'll introduce terminology and notation as well as statistical definitions related to sampling. This will allow us to succinctly summarize and refer to the ideas behind sampling for the rest of this book.

7.3.1 Terminology and notation

Let's now attach words and labels to the various sampling concepts we've seen so far by introducing some terminology and mathematical notation. While they may seem daunting at first, we'll make sure to tie each of them to sampling bowl activities you performed earlier. Furthermore, throughout this book we'll give you plenty of opportunity for practice, as the best method for mastering these terms is repetition.

The first set of terms and notation relate to **populations**:

1. A **population** is a collection of individuals or observations we are interested in. This is also commonly denoted as a **study population**. We mathematically denote the population's size using upper-case N .
2. A **population parameter** is some numerical summary about the population that is unknown but you wish you knew. For example, when this quantity is a mean like the average height of all Canadians, the population parameter of interest is the *population mean*.

3. A **census** is an exhaustive enumeration or counting of all N individuals in the population. We do this in order to compute the population parameter's value *exactly*. Of note is that as the number N of individuals in our population increases, conducting a census gets more expensive (in terms of time, energy, and money).

So in our sampling activities, the **population** is the collection of $N = 2400$ identically sized red and white balls in the bowl shown in Figure 7.1. Recall that we also represented the bowl “virtually” in the data frame `bowl`:

```
bowl
```

```
# A tibble: 2,400 x 2
  ball_ID color
  <int> <chr>
1     1 white
2     2 white
3     3 white
4     4 red
5     5 white
6     6 white
7     7 red
8     8 white
9     9 red
10    10 white
# i 2,390 more rows
```

The **population parameter** here is the proportion of the bowl’s balls that are red. Whenever we’re interested in a proportion of some value in a population, the population parameter has a specific name: the *population proportion*. We denote population proportions with the letter p . We’ll see later on in Table 7.5 that we can also consider other types of population parameters, like population means and population regression slopes.

In order to compute this population proportion p exactly, we need to first conduct a **census** by going through all $N = 2400$ and counting the number that are red. We then divide this count by 2400 to obtain the proportion red.

You might be now asking yourself: “Wait. I understand that performing a census on the actual bowl would take a long time. But can’t we conduct a ‘virtual’ census using the virtual bowl?” You are absolutely correct! In fact when the authors of this book created the `bowl` data frame, they made its contents match the contents of actual bowl not by doing a census, but by reading the contents written on the box the bowl came in!

Let's conduct this "virtual" census by using the same `dplyr` verbs you used earlier to count the number of balls that are red:

```
bowl %>%
  summarize(red = sum(color == "red"))
```

```
# A tibble: 1 × 1
  red
  <int>
1 900
```

Since 900 of the 2400 are red, the proportion is $900/2400 = 0.375 = 37.5\%$. So we know the value of the population parameter: in our case, the population proportion p is equal to 0.375.

At this point, you might be further asking yourself: "If we had a way of knowing that the proportion of the balls that are red is 37.5%, then why did we do any sampling?" Great question! Normally, you wouldn't do any sampling! However, the sampling activities we did this chapter are merely simulations of how sampling is done in real-life! We perform these simulations in order to study:

1. The effect of *sampling variation* on our estimates.
2. The effect of sample size on *sampling variation*.

As we'll see in Section 7.4 on polls, in real-life sampling not only will the population size N be very large making a census expensive, but sometimes we won't even know how big the population is! For now however, we press on with our next set of terms and notation.

The second set of terms and notation relate to **samples**:

1. **Sampling** is the act of collecting a sample from the population, which we generally only do when we can't perform a census. We mathematically denote the sample size using lower case n , as opposed to upper case N which denotes the population's size. Typically the sample size n is much smaller than the population size N . Thus sampling is a much cheaper alternative than performing a census.
2. A **point estimate**, also known as a **sample statistic**, is a summary statistic computed from a sample that *estimates* the unknown population parameter.

So previously we conducted **sampling** using a shovel with 50 slots to extract samples of size $n = 50$. To perform the virtual analog of this sampling, recall that we used the `rep_sample_n()` function as follows:

```
virtual_shovel <- bowl %>%
  rep_sample_n(size = 50)
virtual_shovel
```

```
# A tibble: 50 x 3
# Groups:   replicate [1]
  replicate ball_ID color
  <int>    <int> <chr>
1       1      1970 white
2       1       842 red
3       1     2287 white
4       1       599 white
5       1      108 white
6       1      846 red
7       1      390 red
8       1      344 white
9       1      910 white
10      1     1485 white
# i 40 more rows
```

Using the sample of 50 balls contained in `virtual_shovel`, we generated an estimate of the proportion of the bowl's balls that are red `prop_red`.

```
virtual_shovel %>%
  summarize(num_red = sum(color == "red")) %>%
  mutate(prop_red = num_red / 50)
```

```
# A tibble: 1 x 3
  replicate num_red prop_red
  <int>    <int>    <dbl>
1       1      12     0.24
```

So in our case, the value of `prop_red` is the **point estimate** of the population proportion p since it estimates the latter's value. Furthermore, this point estimate has a specific name when considering proportions: the *sample proportion*. It is denoted using \hat{p} because it is a common convention in statistics to use a “hat” symbol to denote point estimates.

The third set of terms relate to **sampling methodology**: the method used to collect samples. You'll see here and throughout the rest of your book that the *way* you collect samples directly influences their quality.

1. A sample is said to be **representative** if it roughly “looks like” the population. In other words, if the sample’s characteristics are a “good” representation of the population’s characteristics.
2. We say a sample is **generalizable** if any results based on the sample can generalize to the population. In other words, if we can make “good” guesses about the population using the sample.
3. We say a sampling procedure is **biased** if certain individuals in a population have a higher chance of being included in a sample than others. We say a sampling procedure is **unbiased** if every individual in a population has an equal chance of being sampled.

We say a sample of n balls extracted using our shovel is **representative** of the population if its contents “roughly resemble” the contents of the bowl. If so, then the proportion of the shovel’s balls that are red can **generalize** to the proportion of the bowl’s $N = 2400$ balls that are red. Or expressed differently, \hat{p} is a “good guess” of p . Now say we cheated when using the shovel and removed a number of white balls in favor of red balls. Then this sample would be **biased** towards red balls, and thus the sample would no longer be representative of the bowl.

The fourth and final set of terms and notation relate to the goal of sampling:

1. One way to ensure that a sample is unbiased and representative of the population is by using **random sampling**.
2. **Inference** is the act of “making a guess” about some unknown. **Statistical inference** is the act of making a guess about a population using a sample.

In our case, since the `rep_sample_n()` function uses your computer’s random number generator², we were in fact performing **random sampling**.

Let’s now put all four sets of terms and notation together, keeping our sampling activities in mind:

- Since we extracted a sample of $n = 50$ balls at *random*, we mixed all of the equally sized balls before using the shovel, then
- the contents of the shovel are *unbiased* and *representative* of the contents of the bowl, thus
- any result based on the shovel can *generalize* to the bowl, thus
- the sample proportion \hat{p} of the $n = 50$ balls in the shovel that are red is a “good guess” of the population proportion p of the bowl’s $N = 2400$ balls that are red, thus
- instead of conducting a *census* of the 2400 balls in the bowl, we can **infer** about the bowl using the sample from the shovel.

²https://en.wikipedia.org/wiki/Random_number_generation

What you have been performing is **statistical inference**. This is one of the most important concepts in all of statistics. So much so, we included this term in the title of our book: “Statistical Inference via Data Science”. More generally speaking,

- If the sampling of a sample of size n is done at *random*, then
- the sample is *unbiased* and *representative* of the population of size N , thus
- any result based on the sample can *generalize* to the population, thus
- the point estimate is a “good guess” of the unknown population parameter, thus
- instead of performing a census, we can *infer* about the population using sampling.

In the upcoming Chapter 8 on confidence intervals, we’ll introduce the `infer` package, which makes statistical inference “tidy” and transparent. It is why this third portion of the book is called “Statistical inference via `infer`.”

Learning check

(LC7.8) In the case of our bowl activity, what is the *population parameter*? Do we know its value?

(LC7.9) What would performing a census in our bowl activity correspond to? Why did we not perform a census?

(LC7.10) What purpose do *point estimates* serve in general? What is the name of the point estimate specific to our bowl activity? What is its mathematical notation?

(LC7.11) How did we ensure that our tactile samples using the shovel were random?

(LC7.12) Why is it important that sampling be done *at random*?

(LC7.13) What are we *inferring* about the bowl based on the samples using the shovel?

7.3.2 Statistical definitions

To further attach words and labels to the various sampling concepts we’ve seen so far, we also introduce some important statistical definitions related to sampling. As a refresher of our 1000 repeated/replicated virtual samples of size $n = 25$, $n = 50$, and $n = 100$ in Section 7.2, let’s display Figure 7.12 again as Figure 7.13.

Comparing distributions of proportions red for three different shovel sizes.

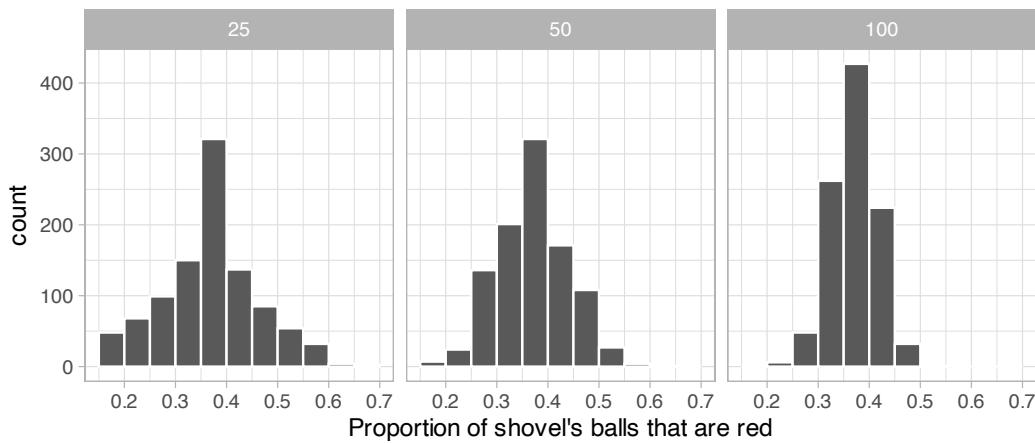


FIGURE 7.13: Previously seen three distributions of the sample proportion \hat{p} .

These types of distributions have a special name: **sampling distributions of point estimates**. Their visualization displays the effect of sampling variation on the distribution of any point estimate, in this case, the sample proportion \hat{p} . Using these sampling distributions, for a given sample size n , we can make statements about what values we can typically expect. Unfortunately, the term *sampling distribution* is often confused with a *sample's distribution* which is merely the distribution of the values in a single sample.

For example, observe the centers of all three sampling distributions: they are all roughly centered around $0.4 = 40\%$. Furthermore, observe that while we are somewhat likely to observe sample proportions of red balls of $0.2 = 20\%$ when using the shovel with 25 slots, we will almost never observe a proportion of 20% when using the shovel with 100 slots. Observe also the effect of sample size on the sampling variation. As the sample size n increases from 25 to 50 to 100, the variation of the sampling distribution decreases and thus the values cluster more and more tightly around the same center of around 40%. We quantified this variation using the standard deviation of our sample proportions in Table 7.1, which we display again as Table 7.2:

TABLE 7.2: Previously seen comparing standard deviations of proportions red for three different shovels

Number of slots in shovel	Standard deviation of proportions red
25	0.094
50	0.069
100	0.045

So as the sample size increases, the standard deviation of the proportion of red balls decreases. This type of standard deviation has another special name: **standard error of a point estimate**. Standard errors quantify the effect of sampling variation

induced on our estimates. In other words, they quantify how much we can expect different proportions of a shovel's balls that are red to *vary* from one sample to another sample to another sample, and so on. As a general rule, as sample size increases, the standard error decreases.

Similarly to confusion between *sampling distributions* with a *sample's distribution*, people often confuse the *standard error* with the *standard deviation*. This is especially the case since a standard error is itself a kind of standard deviation. The best advice we can give is that a standard error is merely a *kind* of standard deviation: the standard deviation of any point estimate from sampling. In other words, all standard errors are standard deviations, but not every standard deviation is necessarily a standard error.

To help reinforce these concepts, let's re-display Figure 7.12 but using our new terminology, notation, and definitions relating to sampling in Figure 7.14.

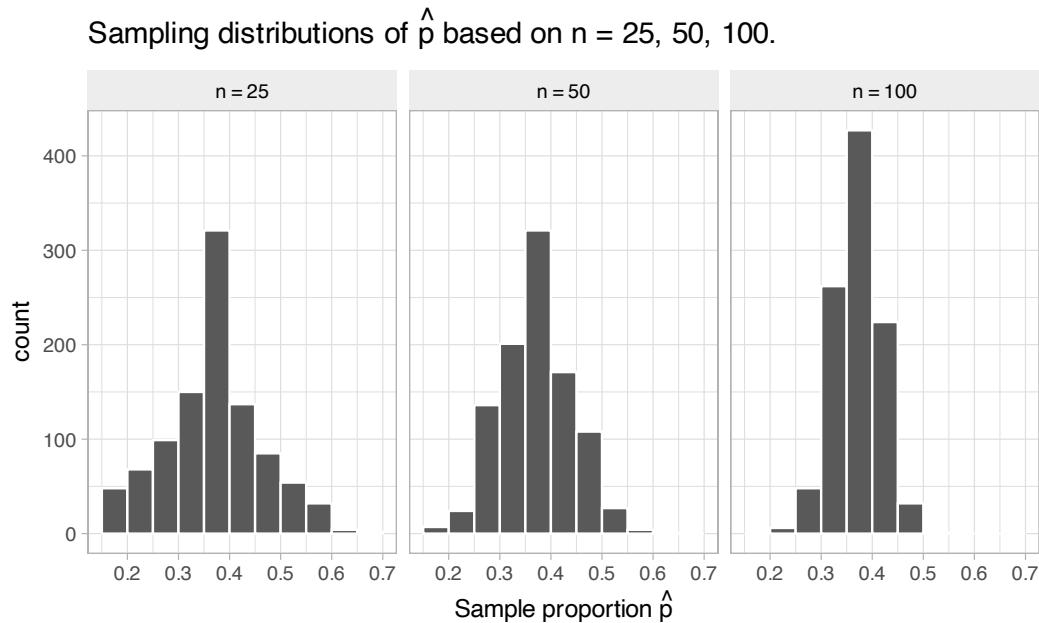


FIGURE 7.14: Three sampling distributions of the sample proportion \hat{p} .

Furthermore, let's re-display Table 7.1 but using our new terminology, notation, and definitions relating to sampling in Table 7.3.

Remember the key message of this last table: that as the sample size n goes up, the “typical” error of your point estimate will go down, as quantified by the *standard error*. In fact, in Subsection 7.6.2 we'll see that the standard error for the sample proportion \hat{p} can also be approximated via a mathematical theory-based formula, a formula that has n in the denominator.

TABLE 7.3: Standard errors of the sample proportion based on sample sizes of 25, 50, and 100

Sample size (n)	Standard error of \hat{p}
n = 25	0.094
n = 50	0.069
n = 100	0.045

Learning check

(LC7.14) What purpose did the *sampling distributions* serve?

(LC7.15) What does the *standard error* of the sample proportion \hat{p} quantify?

7.3.3 The moral of the story

Let's recap this section so far. We've seen that if a sample is generated at random, then the resulting point estimate is a "good guess" of the true unknown population parameter. In our sampling activities, since we made sure to mix the balls first before extracting a sample with the shovel, the resulting sample proportion \hat{p} of the shovel's balls that were red was a "good guess" of the population proportion p of the bowl's balls that were red.

However, what do we mean by our point estimate being a "good guess"? Sometimes, we'll get an estimate that is less than the true value of the population parameter, while at other times we'll get an estimate that is greater. This is due to sampling variation. However, despite this sampling variation, our estimates will "on average" be correct and thus will be centered at the true value. This is because our sampling was done at random and thus in an unbiased fashion.

In our sampling activities, sometimes our sample proportion \hat{p} was less than the true population proportion p , while at other times it was greater. This was due to the sampling variability. However, despite this sampling variation, our sample proportions \hat{p} were "on average" correct and thus were centered at the true value of the population proportion p . This is because we mixed our bowl before taking samples and thus the sampling was done at random and thus in an unbiased fashion. This is also known as having an *accurate* estimate.

Recall from earlier that the value of the population proportion p of the $N = 2400$ balls in the bowl was $900/2400 = 0.375 = 37.5\%$. We computed this value by performing a virtual census of `bowl`. Let's re-display our sampling distributions from Figures 7.12 and 7.14, but now with a vertical red line marking the true population proportion

p of balls that are red = 37.5% in Figure 7.15. We see that while there is a certain amount of error in the sample proportions \hat{p} for all three sampling distributions, on average the \hat{p} are centered at the true population proportion red p .

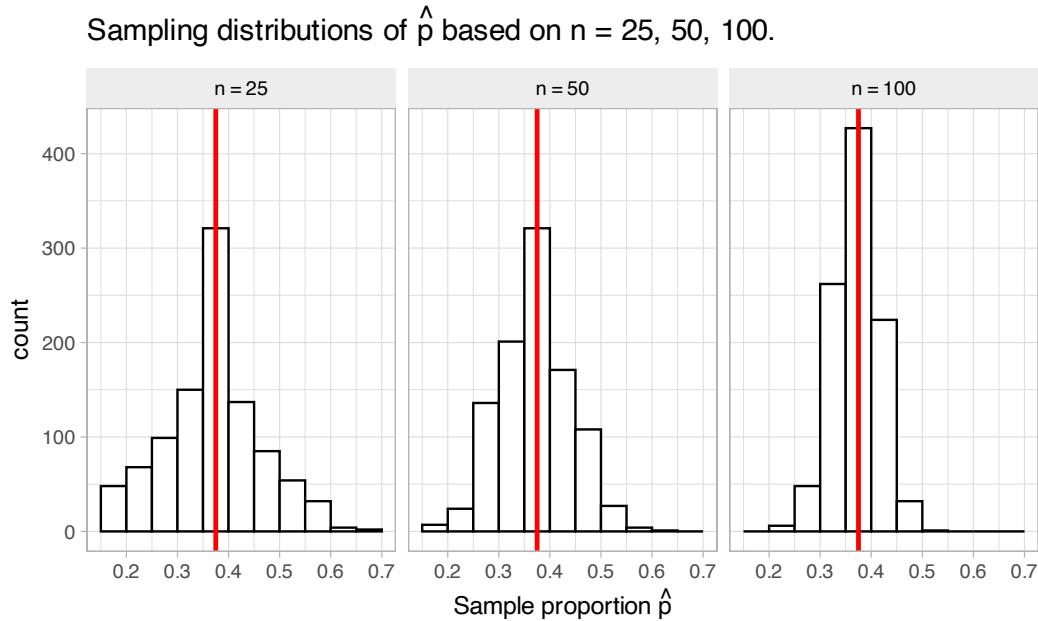


FIGURE 7.15: Three sampling distributions with population proportion p marked by vertical line.

We also saw in this section that as your sample size n increases, your point estimates will vary less and less and be more and more concentrated around the true population parameter. This variation is quantified by the decreasing *standard error*. In other words, the typical error of your point estimates will decrease. In our sampling exercise, as the sample size increased, the variation of our sample proportions \hat{p} decreased. You can observe this behavior in Figure 7.15. This is also known as having a *precise* estimate.

So random sampling ensures our point estimates are *accurate*, while on the other hand having a large sample size ensures our point estimates are *precise*. While the terms “accuracy” and “precision” may sound like they mean the same thing, there is a subtle difference. Accuracy describes how “on target” our estimates are, whereas precision describes how “consistent” our estimates are. Figure 7.16 illustrates the difference.

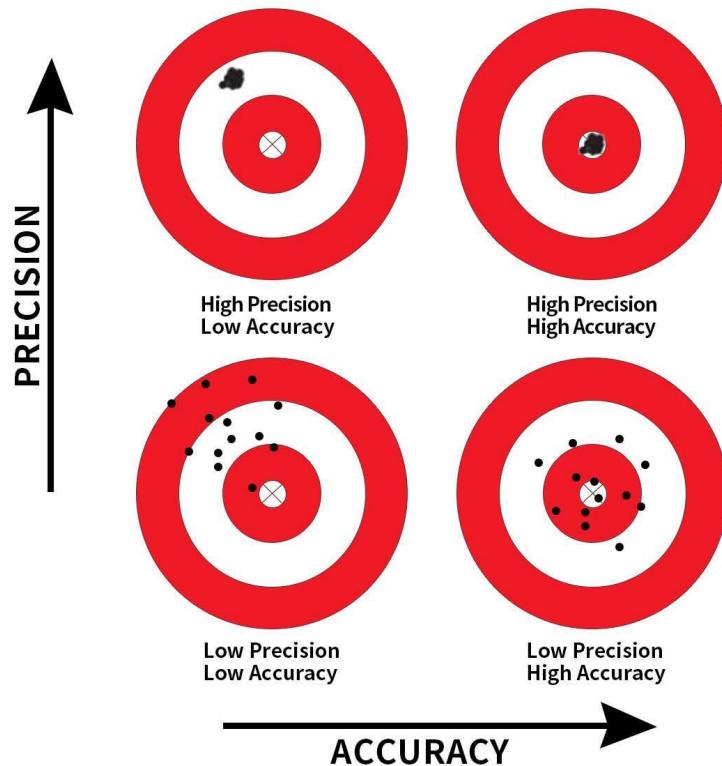


FIGURE 7.16: Comparing accuracy and precision.

At this point, you might be asking yourself: “Why did we take 1000 repeated samples of size $n = 25, 50$, and 100 ? Shouldn’t we be taking only *one* sample that’s as large as possible?”. If you did ask yourself these questions, your suspicion is correct! Recall from earlier when we asked ourselves “If we had a way of knowing that the proportion of the balls that are red is 37.5%, then why did we do any sampling?” Similarly, we took 1000 repeated samples as a simulation of how sampling is done in real-life! We used these simulations to study:

1. The effect of *sampling variation* on our estimates.
2. The effect of sample size on *sampling variation*.

This is not how sampling is done in real life! In a real-life scenario, we wouldn’t take 1000 repeated/replicated samples, but rather a single sample that’s as large as we can afford. In Section 7.4, we’re going to study a real-life example of sampling: polls.

Learning check

(LC7.16) The table that follows is a version of Table 7.3 matching sample sizes n to different *standard errors* of the sample proportion \hat{p} , but with the rows randomly re-ordered and the sample sizes removed. Fill in the table by matching the correct sample sizes to the correct standard errors.

TABLE 7.4: Standard errors of \hat{p} based on $n = 25, 50, 100$

Sample size	Standard error of \hat{p}
$n =$	0.094
$n =$	0.045
$n =$	0.069

For the following four *Learning checks*, let the *estimate* be the sample proportion \hat{p} : the proportion of a shovel's balls that were red. It estimates the population proportion p : the proportion of the bowl's balls that were red.

(LC7.17) What is the difference between an *accurate* and a *precise* estimate?

(LC7.18) How do we ensure that an estimate is *accurate*? How do we ensure that an estimate is *precise*?

(LC7.19) In a real-life situation, we would not take 1000 different samples to infer about a population, but rather only one. Then, what was the purpose of our exercises where we took 1000 different samples?

(LC7.20) Figure 7.16 with the targets shows four combinations of “accurate versus precise” estimates. Draw four corresponding *sampling distributions* of the sample proportion \hat{p} , like the one in the leftmost plot in Figure 7.15.

7.4 Case study: Polls

Let's now switch gears to a more realistic sampling scenario than our bowl activity: a poll. In practice, pollsters do not take 1000 repeated samples as we did in our previous sampling activities, but rather take only a *single sample* that's as large as possible.

On December 4, 2013, National Public Radio in the US reported on a poll of President Obama's approval rating among young Americans aged 18-29 in an article, “Poll: Support For Obama Among Young Americans Eroding.”³ The poll was conducted by

³<https://www.npr.org/sections/itsallpolitics/2013/12/04/248793753/poll-support-for-obama-among-young-americans-eroding>

the Kennedy School's Institute of Politics at Harvard University. A quote from the article:

After voting for him in large numbers in 2008 and 2012, young Americans are souring on President Obama.

According to a new Harvard University Institute of Politics poll, just 41 percent of millennials — adults aged 18-29 — approve of Obama's job performance, his lowest-ever standing among the group and an 11-point drop from April.

Let's tie elements of the real-life poll in this news article with our "tactile" and "virtual" bowl activity from Sections 7.1 and 7.2 using the terminology, notations, and definitions we learned in Section 7.3. You'll see that our sampling activity with the bowl is an idealized version of what pollsters are trying to do in real life.

First, who is the **(study) population** of N individuals or observations of interest?

- Bowl: $N = 2400$ identically sized red and white balls
- Obama poll: $N = ?$ young Americans aged 18-29

Second, what is the **population parameter**?

- Bowl: The population proportion p of *all* the balls in the bowl that are red.
- Obama poll: The population proportion p of *all* young Americans who approve of Obama's job performance.

Third, what would a **census** look like?

- Bowl: Manually going over all $N = 2400$ balls and exactly computing the population proportion p of the balls that are red.
- Obama poll: Locating all N young Americans and asking them all if they approve of Obama's job performance. In this case, we don't even know what the population size N is!

Fourth, how do you perform **sampling** to obtain a sample of size n ?

- Bowl: Using a shovel with n slots.
- Obama poll: One method is to get a list of phone numbers of all young Americans and pick out n phone numbers. In this poll's case, the sample size of this poll was $n = 2089$ young Americans.

Fifth, what is your **point estimate** also known as the **sample statistic** of the unknown population parameter?

- Bowl: The sample proportion \hat{p} of the balls in the shovel that were red.
- Obama poll: The sample proportion \hat{p} of young Americans in the sample that approve of Obama's job performance. In this poll's case, $\hat{p} = 0.41 = 41\%$, the quoted percentage in the second paragraph of the article.

Sixth, is the sampling procedure **representative**?

- Bowl: Are the contents of the shovel representative of the contents of the bowl? Because we mixed the bowl before sampling, we can feel confident that they are.
- Obama poll: Is the sample of $n = 2089$ young Americans representative of *all* young Americans aged 18-29? This depends on whether the sampling was random.

Seventh, are the samples **generalizable** to the greater population?

- Bowl: Is the sample proportion \hat{p} of the shovel's balls that are red a "good guess" of the population proportion p of the bowl's balls that are red? Given that the sample was representative, the answer is yes.
- Obama poll: Is the sample proportion $\hat{p} = 0.41$ of the sample of young Americans who supported Obama a "good guess" of the population proportion p of all young Americans who supported Obama at this time in 2013? In other words, can we confidently say that roughly 41% of *all* young Americans approved of Obama at the time of the poll? Again, this depends on whether the sampling was random.

Eighth, is the sampling procedure **unbiased**? In other words, do all observations have an equal chance of being included in the sample?

- Bowl: Since each ball was equally sized and we mixed the bowl before using the shovel, each ball had an equal chance of being included in a sample and hence the sampling was unbiased.
- Obama poll: Did all young Americans have an equal chance at being represented in this poll? Again, this depends on whether the sampling was random.

Ninth and lastly, was the sampling done at **random**?

- Bowl: As long as you mixed the bowl sufficiently before sampling, your samples would be random.
- Obama poll: Was the sample conducted at random? We can't answer this question without knowing about the *sampling methodology* used by Kennedy School's Institute of Politics at Harvard University. We'll discuss this more at the end of this section.

In other words, the poll by Kennedy School's Institute of Politics at Harvard University can be thought of as *an instance* of using the shovel to sample balls from the bowl. Furthermore, if another polling company conducted a similar poll of young Americans at roughly the same time, they would likely get a different estimate than 41%. This is due to *sampling variation*.

Let's now revisit the sampling paradigm from Subsection 7.3.1:

In general:

- If the sampling of a sample of size n is done at *random*, then
- the sample is *unbiased* and *representative* of the population of size N , thus
- any result based on the sample can *generalize* to the population, thus
- the point estimate is a “good guess” of the unknown population parameter, thus
- instead of performing a census, we can *infer* about the population using sampling.

Specific to the bowl:

- Since we extracted a sample of $n = 50$ balls at *random*, in other words we mixed all of the equally sized balls before using the shovel, then
- the contents of the shovel are *unbiased* and *representative* of the contents of the bowl, thus
- any result based on the shovel can *generalize* to the bowl, thus
- the sample proportion \hat{p} of the $n = 50$ balls in the shovel that are red is a “good guess” of the population proportion p of the bowl’s $N = 2400$ balls that are red, thus
- instead of conducting a *census* of the 2400 balls in the bowl, we can *infer* about the bowl using the sample from the shovel.

Specific to the Obama poll:

- If we had a way of contacting a *randomly* chosen sample of 2089 young Americans and polling their approval of President Obama in 2013, then
- these 2089 young Americans would be an *unbiased* and *representative* sample of all young Americans in 2013, thus
- any results based on this sample of 2089 young Americans can *generalize* to the entire population of all young Americans in 2013, thus
- the reported sample approval rating of 41% of these 2089 young Americans is a *good guess* of the true approval rating among all young Americans in 2013, thus
- instead of performing an expensive census of all young Americans in 2013, we can *infer* about all young Americans in 2013 using polling.

So as you can see, it was critical for the sample obtained by Kennedy School's Institute of Politics at Harvard University to be truly random in order to infer about *all* young

Americans' opinions about Obama. Was their sample truly random? It's hard to answer such questions without knowing about the *sampling methodology* they used. For example, if this poll was conducted using only mobile phone numbers, people without mobile phones would be left out and therefore not represented in the sample. What about if Kennedy School's Institute of Politics at Harvard University conducted this poll on an internet news site? Then people who don't read this particular internet news site would be left out. Ensuring that our samples were random was easy to do in our sampling bowl exercises; however, in a real-life situation like the Obama poll, this is much harder to do.

Learning check

Comment on the representativeness of the following *sampling methodologies*:

(LC7.21) The Royal Air Force wants to study how resistant all their airplanes are to bullets. They study the bullet holes on all the airplanes on the tarmac after an air battle against the Luftwaffe (German Air Force).

(LC7.22) Imagine it is 1993, a time when almost all households had landlines. You want to know the average number of people in each household in your city. You randomly pick out 500 phone numbers from the phone book and conduct a phone survey.

(LC7.23) You want to know the prevalence of illegal downloading of TV shows among students at a local college. You get the emails of 100 randomly chosen students and ask them, "How many times did you download a pirated TV show last week?"

(LC7.24) A local college administrator wants to know the average income of all graduates in the last 10 years. So they get the records of five randomly chosen graduates, contact them, and obtain their answers.

7.5 Central Limit Theorem

This chapter began with (virtual) access to a large bowl of balls (our population) and a desire to figure out the proportion of red balls. Despite having access to this population, in reality, you almost **never** will have **access to the population**, either because the population is too large, ever changing, or too expensive to take a census of. Accepting this reality means accepting that we need to use *statistical inference*.

In Section 7.3.1, we stated that "statistical inference is the act of making a guess about a population using a sample." But how do we *do* this inference? In the previous

section, we defined the *sampling framework* only to state that in reality we take one large sample, instead of many samples as done in the sampling framework (which we modeled physically by taking many samples from the bowl).

In reality, we take *only one* sample and use that *one* sample to make statements about the population parameter. This ability of making statements about the population is allowable by a famous theorem, or mathematically proven truth, called the *Central Limit Theorem*. What you visualized in Figures 7.12 and 7.14 and summarized in Tables 7.1 and 7.3 was a demonstration of this theorem. It loosely states that when sample means are based on larger and larger sample sizes, the sampling distribution of these sample means becomes both more and more normally shaped and more and more narrow.

In other words, as our sample size gets larger (1) the sampling distribution of a point estimate (like a sample proportion) increasingly follows a *normal distribution* and (2) the variation of these sampling distributions gets smaller, as quantified by their standard errors. We discuss the properties of the normal distribution in Appendix A.2.

Shuyi Chiou, Casey Dunn, and Pathikrit Bhattacharyya created a 3-minute and 38-second video at <https://youtu.be/jvoxEYmQHNM> explaining this crucial statistical theorem using the average weight of wild bunny rabbits and the average wingspan of dragons as examples. Figure 7.17 shows a preview of this video.

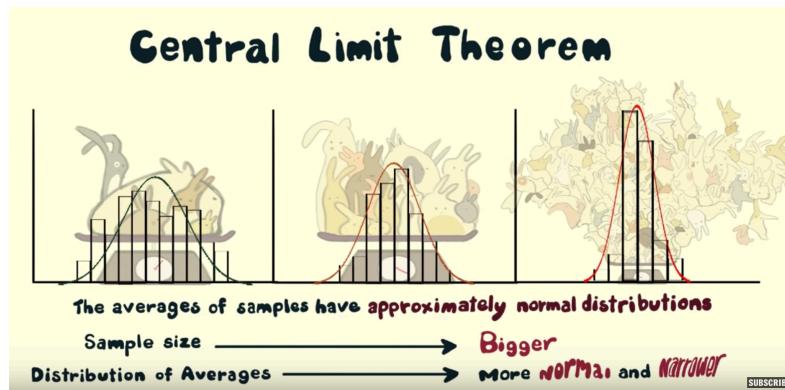


FIGURE 7.17: Preview of Central Limit Theorem video.

Here's what is so surprising about the Central Limit Theorem: regardless of the shape of the underlying population distribution, the sampling distribution of means (such as the sample mean of bunny weights or the sample mean of the length of dragon wings) and proportions (such as the sample proportion red in our shovels) will be **normal**. Normal distributions are defined by where they are centered and how wide they are, and the Central Limit Theorem gives us both:

1. The sampling distribution of the point estimate is centered at the true population parameter.

2. We have an estimate for how wide the sampling distribution of the point estimate is, given by the standard error (which we will discuss further in Chapter 8).

What the Central Limit Theorem creates for us is a ladder between a *single* sample and the population. By the Central Limit Theorem, we can say that (1) our sample's point estimate is drawn from a normal distribution centered at the true population parameter and (2) that the width of that normal distribution is governed by the standard error of our point estimate. Relating this to our bowl, if we pull one sample and get the sample proportion of red balls \hat{p} , this value of \hat{p} is drawn from the normal curve centered at the true population proportion of red balls p with the computed standard error.

7.6 Conclusion

7.6.1 Sampling scenarios

In this chapter, we performed both tactile and virtual sampling exercises to infer about an unknown proportion. We also presented a case study of sampling in real life with polls. In each case, we used the sample proportion \hat{p} to estimate the population proportion p . However, we are not just limited to scenarios related to proportions. In other words, we can use sampling to estimate other population parameters using other point estimates as well. We present four more such scenarios in Table 7.5.

TABLE 7.5: Scenarios of sampling for inference

Scenario	Population parameter	Notation	Point estimate	Symbol(s)
1	Population proportion	p	Sample proportion	\hat{p}
2	Population mean	μ	Sample mean	\bar{x} or $\hat{\mu}$
3	Difference in population proportions	$p_1 - p_2$	Difference in sample proportions	$\hat{p}_1 - \hat{p}_2$
4	Difference in population means	$\mu_1 - \mu_2$	Difference in sample means	$\bar{x}_1 - \bar{x}_2$ or $\hat{\mu}_1 - \hat{\mu}_2$
5	Population regression slope	β_1	Fitted regression slope	b_1 or $\hat{\beta}_1$

In the rest of this book, we'll cover all the remaining scenarios as follows:

- In Chapter 8, we'll cover examples of statistical inference for

- Scenario 2: The mean age μ of all pennies in circulation in the US.
- Scenario 3: The difference $p_1 - p_2$ in the proportion of people who yawn *when seeing someone else yawn first* minus the proportion of people who yawn *without seeing someone else yawn first*. This is an example of *two-sample* inference.
- In Chapter 9, we'll cover an example of statistical inference for
 - Scenario 4: The difference $\mu_1 - \mu_2$ in mean IMDb ratings for action and romance movies. This is another example of *two-sample* inference.
- In Chapter 10, we'll cover an example of statistical inference for regression by revisiting the regression models for teaching score as a function of various instructor demographic variables you saw in Chapters 5 and 6.
 - Scenario 5: The slope β_1 of the population regression line.

7.6.2 Theory-based standard-errors

There exists in many cases a formula that approximates the standard error! In the case of our bowl where we used the sample proportion red \hat{p} to estimate the proportion of the bowl's balls that are red, the formula that approximates the standard error for the sample proportion \hat{p} is:

$$\text{SE}_{\hat{p}} \approx \sqrt{\frac{\hat{p}(1 - \hat{p})}{n}}$$

For example, say you sampled $n = 50$ balls and observed 21 red balls. This equals a sample proportion \hat{p} of $21/50 = 0.42$. So, using the formula, an approximation of the standard error of \hat{p} is

$$\text{SE}_{\hat{p}} \approx \sqrt{\frac{0.42(1 - 0.42)}{50}} = \sqrt{0.004872} = 0.0698 \approx 0.070$$

Say instead you sampled $n = 100$ balls and observed 42 red balls. This once again equals a sample proportion \hat{p} of $42/100 = 0.42$. However using the formula, an approximation of the standard error of \hat{p} is now

$$\text{SE}_{\hat{p}} \approx \sqrt{\frac{0.42(1 - 0.42)}{100}} = \sqrt{0.002436} = 0.0494$$

Observe that the standard error has gone down from 0.0698 to 0.0494. In other words, the “typical” error of our estimates using $n = 100$ will go down relative to $n = 50$ and hence be more *precise*. Recall that we illustrated the difference between accuracy and precision of estimates in Figure 7.16.

The key observation to make in the formula is that there is an n in the denominator. As the sample size n increases, the standard error decreases. We've demonstrated this fact using our virtual shovels in Subsection 7.3.3.

Furthermore, this is one of the key messages of the Central Limit Theorem we saw in Subsection 7.5: as the sample size n increases, the distribution of averages gets narrower as quantified by the standard deviation of the sampling distribution of the sample mean. This standard deviation of the sampling distribution of the sample means in turn has a special name: the standard error of the sample mean.

Why is this formula true? Unfortunately, we don't have the tools at this point to prove this; you'll need to take a more advanced course in probability and statistics. (It is related to the concepts of Bernoulli and Binomial Distributions. You can read more about its derivation here⁴ if you like.)

7.6.3 Additional resources

Solutions to all *Learning checks* can be found online in Appendix D⁵.

An R script file of all R code used in this chapter is available at <https://www.moderndive.com/scripts/07-sampling.R>.

7.6.4 What's to come?

Recall in our Obama poll case study in Section 7.4 that based on this particular sample, the best guess by Kennedy School's Institute of Politics at Harvard University of the U.S. President Obama's approval rating among all young Americans was 41%. However, this isn't the end of the story. If you read the article further, it states:

The online survey of 2,089 adults was conducted from Oct. 30 to Nov. 11, just weeks after the federal government shutdown ended and the problems surrounding the implementation of the Affordable Care Act began to take center stage. The poll's margin of error was plus or minus 2.1 percentage points.

Note the term *margin of error*, which here is "plus or minus 2.1 percentage points." Most polls won't produce an estimate that's perfectly right; there will always be a certain amount of error caused by *sampling variation*. The margin of error of plus or minus 2.1 percentage points is saying that a typical range of errors for polls of this type is about $\pm 2.1\%$, in words from about 2.1% too small to about 2.1% too big. We can restate this as the interval of $[41\% - 2.1\%, 41\% + 2.1\%] = [38.9\%, 43.1\%]$ (this notation indicates the interval contains all values between 38.9% and 43.1%, including the end points of 38.9% and 43.1%). We'll see in the next chapter that such intervals are known as *confidence intervals*.

⁴http://onlinestatbook.com/2/sampling_distributions/samp_dist_p.html

⁵<https://moderndive.com/D-appendixD.html>

8

Bootstrapping and Confidence Intervals

In Chapter 7, we studied sampling. We started with a “tactile” exercise where we wanted to know the proportion of balls in the sampling bowl in Figure 7.1 that are red. While we could have performed an exhaustive count, this would have been a tedious process. So instead, we used a shovel to extract a sample of 50 balls and used the resulting proportion that were red as an *estimate*. Furthermore, we made sure to mix the bowl’s contents before every use of the shovel. Because of the randomness created by the mixing, different uses of the shovel yielded different proportions red and hence different estimates of the proportion of the bowl’s balls that are red.

We then mimicked this “tactile” sampling exercise with an equivalent “virtual” sampling exercise performed on the computer. Using our computer’s random number generator, we quickly mimicked the above sampling procedure a large number of times. In Subsection 7.2, we quickly repeated this sampling procedure 1000 times, using three different “virtual” shovels with 25, 50, and 100 slots. We visualized these three sets of 1000 estimates in Figure 7.15 and saw that as the sample size increased, the variation in the estimates decreased.

In doing so, what we did was construct *sampling distributions*. The motivation for taking 1000 repeated samples and visualizing the resulting estimates was to study how these estimates varied from one sample to another; in other words, we wanted to study the effect of *sampling variation*. We quantified the variation of these estimates using their standard deviation, which has a special name: the *standard error*. In particular, we saw that as the sample size increased from 25 to 50 to 100, the standard error decreased and thus the sampling distributions narrowed. Larger sample sizes led to more *precise* estimates that varied less around the center.

We then tied these sampling exercises to terminology and mathematical notation related to sampling in Subsection 7.3.1. Our *study population* was the large bowl with $N = 2400$ balls, while the *population parameter*, the unknown quantity of interest, was the population proportion p of the bowl’s balls that were red. Since performing a *census* would be expensive in terms of time and energy, we instead extracted a *sample* of size $n = 50$. The *point estimate*, also known as a *sample statistic*, used to estimate p was the sample proportion \hat{p} of these 50 sampled balls that were red. Furthermore, since the sample was obtained at *random*, it can be considered as *unbiased* and *representative* of the population. Thus any results based on the sample could be *generalized* to the population. Therefore, the proportion of the shovel’s balls that were red was a “good guess” of the proportion of the bowl’s balls that are red. In other words, we used the sample to *infer* about the population.

However, as described in Section 7.2, both the tactile and virtual sampling exercises are not what one would do in real life; this was merely an activity used to study the effects of sampling variation. In a real-life situation, we would not take 1000 samples of size n , but rather take a *single* representative sample that's as large as possible. Additionally, we knew that the true proportion of the bowl's balls that were red was 37.5%. In a real-life situation, we will not know what this value is. Because if we did, then why would we take a sample to estimate it?

An example of a realistic sampling situation would be a poll, like the Obama poll¹ you saw in Section 7.4. Pollsters did not know the true proportion of *all* young Americans who supported President Obama in 2013, and thus they took a single sample of size $n = 2089$ young Americans to estimate this value.

So how does one quantify the effects of sampling variation when you only have a *single sample* to work with? You cannot directly study the effects of sampling variation when you only have one sample. One common method to study this is *bootstrapping resampling*, which will be the focus of the earlier sections of this chapter.

Furthermore, what if we would like not only a single estimate of the unknown population parameter, but also a *range of highly plausible* values? Going back to the Obama poll article, it stated that the pollsters' estimate of the proportion of all young Americans who supported President Obama was 41%. But in addition it stated that the poll's "margin of error was plus or minus 2.1 percentage points." This "plausible range" was $[41\% - 2.1\%, 41\% + 2.1\%] = [38.9\%, 43.1\%]$. This range of plausible values is what's known as a *confidence interval*, which will be the focus of the later sections of this chapter.

Needed packages

Let's load all the packages needed for this chapter (this assumes you've already installed them). Recall from our discussion in Section 4.4 that loading the `tidyverse` package by running `library(tidyverse)` loads the following commonly used data science packages all at once:

- `ggplot2` for data visualization
- `dplyr` for data wrangling
- `tidy` for converting data to tidy format
- `readr` for importing spreadsheet data into R
- As well as the more advanced `purrr`, `tibble`, `stringr`, and `forcats` packages

If needed, read Section 1.3 for information on how to install and load R packages.

¹<https://www.npr.org/sections/itsallpolitics/2013/12/04/248793753/poll-support-for-obama-among-young-americans-eroding>

```
library(tidyverse)
library(moderndive)
library(infer)
```

8.1 Pennies activity

As we did in Chapter 7, we'll begin with a hands-on tactile activity.

8.1.1 What is the average year on US pennies in 2019?

Try to imagine all the pennies being used in the United States in 2019. That's a lot of pennies! Now say we're interested in the average year of minting of *all* these pennies. One way to compute this value would be to gather up all pennies being used in the US, record the year, and compute the average. However, this would be near impossible! So instead, let's collect a *sample* of 50 pennies from a local bank in downtown Northampton, Massachusetts, USA as seen in Figure 8.1.



FIGURE 8.1: Collecting a sample of 50 US pennies from a local bank.

An image of these 50 pennies can be seen in Figure 8.2. For each of the 50 pennies starting in the top left, progressing row-by-row, and ending in the bottom right, we assigned an “ID” identification variable and marked the year of minting.



FIGURE 8.2: 50 US pennies labelled.

The `moderndive` package contains this data on our 50 sampled pennies in the `pennies_sample` data frame:

```
pennies_sample
```

```
# A tibble: 50 x 2
  ID   year
  <int> <dbl>
1     1  2002
2     2  1986
3     3  2017
4     4  1988
5     5  2008
6     6  1983
7     7  2008
8     8  1996
9     9  2004
10    10  2000
# i 40 more rows
```

The `pennies_sample` data frame has 50 rows corresponding to each penny with two variables. The first variable `ID` corresponds to the ID labels in Figure 8.2, whereas the second variable `year` corresponds to the year of minting saved as a numeric variable, also known as a double (`dbl`).

Based on these 50 sampled pennies, what can we say about *all* US pennies in 2019? Let's study some properties of our sample by performing an exploratory data analysis. Let's first visualize the distribution of the year of these 50 pennies using our data visualization tools from Chapter 2. Since `year` is a numerical variable, we use a histogram in Figure 8.3 to visualize its distribution.

```
ggplot(pennies_sample, aes(x = year)) +  
  geom_histogram(binwidth = 10, color = "white")
```

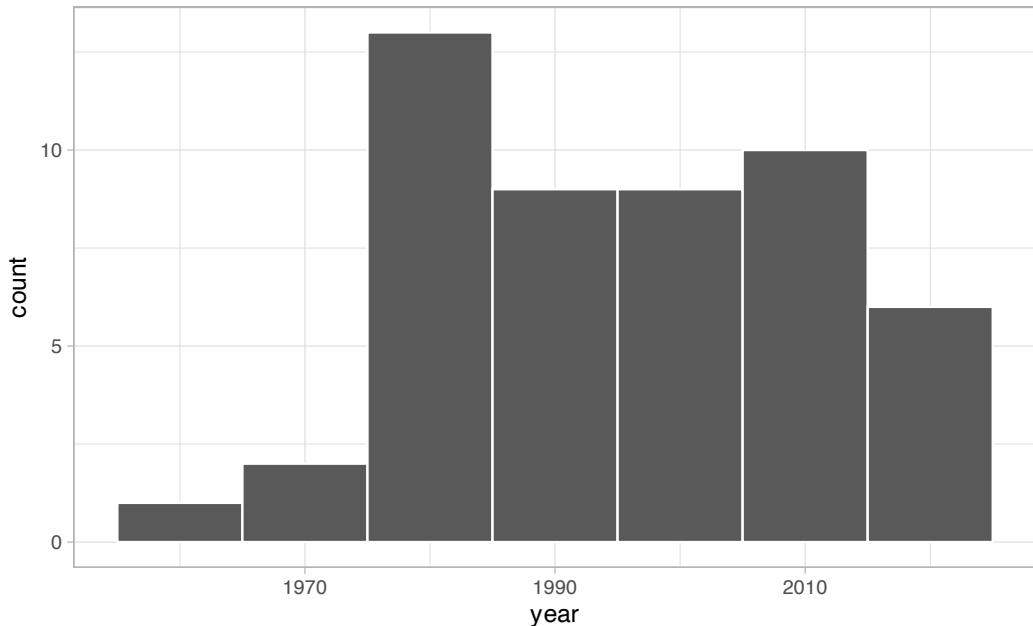


FIGURE 8.3: Distribution of year on 50 US pennies.

Observe a slightly left-skewed distribution, since most pennies fall between 1980 and 2010 with only a few pennies older than 1970. What is the average year for the 50 sampled pennies? Eyeballing the histogram it appears to be around 1990. Let's now compute this value exactly using our data wrangling tools from Chapter 3.

```
x_bar <- pennies_sample %>%  
  summarize(mean_year = mean(year))  
x_bar
```

```
# A tibble: 1 x 1  
  mean_year  
    <dbl>  
1 1995.44
```

Thus, if we're willing to assume that `pennies_sample` is a representative sample from *all* US pennies, a “good guess” of the average year of minting of all US pennies would be 1995.44. In other words, around 1995. This should all start sounding similar to what we did previously in Chapter 7!

In Chapter 7, our *study population* was the bowl of $N = 2400$ balls. Our *population parameter* was the *population proportion* of these balls that were red, denoted by p . In order to estimate p , we extracted a sample of 50 balls using the shovel. We then computed the relevant *point estimate*: the *sample proportion* of these 50 balls that were red, denoted mathematically by \hat{p} .

Here our population is $N =$ whatever the number of pennies are being used in the US, a value which we don't know and probably never will. The population parameter of interest is now the *population mean* year of all these pennies, a value denoted mathematically by the Greek letter μ (pronounced “mu”). In order to estimate μ , we went to the bank and obtained a sample of 50 pennies and computed the relevant point estimate: the *sample mean* year of these 50 pennies, denoted mathematically by \bar{x} (pronounced “x-bar”). An alternative and more intuitive notation for the sample mean is $\hat{\mu}$. However, this is unfortunately not as commonly used, so in this book we'll stick with convention and always denote the sample mean as \bar{x} .

We summarize the correspondence between the sampling bowl exercise in Chapter 7 and our pennies exercise in Table 8.1, which are the first two rows of the previously seen Table 7.5.

TABLE 8.1: Scenarios of sampling for inference

Scenario	Population parameter	Notation	Point estimate	Symbol(s)
1	Population proportion	p	Sample proportion	\hat{p}
2	Population mean	μ	Sample mean	\bar{x} or $\hat{\mu}$

Going back to our 50 sampled pennies in Figure 8.2, the point estimate of interest is the sample mean \bar{x} of 1995.44. This quantity is an *estimate* of the population mean year of *all* US pennies μ .

Recall that we also saw in Chapter 7 that such estimates are prone to *sampling variation*. For example, in this particular sample in Figure 8.2, we observed three pennies with the year 1999. If we sampled another 50 pennies, would we observe exactly three pennies with the year 1999 again? More than likely not. We might observe none, one, two, or maybe even all 50! The same can be said for the other 26 unique years that are represented in our sample of 50 pennies.

To study the effects of *sampling variation* in Chapter 7, we took many samples, something we could easily do with our shovel. In our case with pennies, however, how

would we obtain another sample? By going to the bank and getting another roll of 50 pennies.

Say we're feeling lazy, however, and don't want to go back to the bank. How can we study the effects of sampling variation using our *single sample*? We will do so using a technique known as *bootstrap resampling with replacement*, which we now illustrate.

8.1.2 Resampling once

Step 1: Let's print out identically sized slips of paper representing our 50 pennies as seen in Figure 8.4.



FIGURE 8.4: Step 1: 50 slips of paper representing 50 US pennies.

Step 2: Put the 50 slips of paper into a hat or tuque as seen in Figure 8.5.



FIGURE 8.5: Step 2: Putting 50 slips of paper in a hat.

Step 3: Mix the hat's contents and draw one slip of paper at random as seen in Figure 8.6. Record the year.

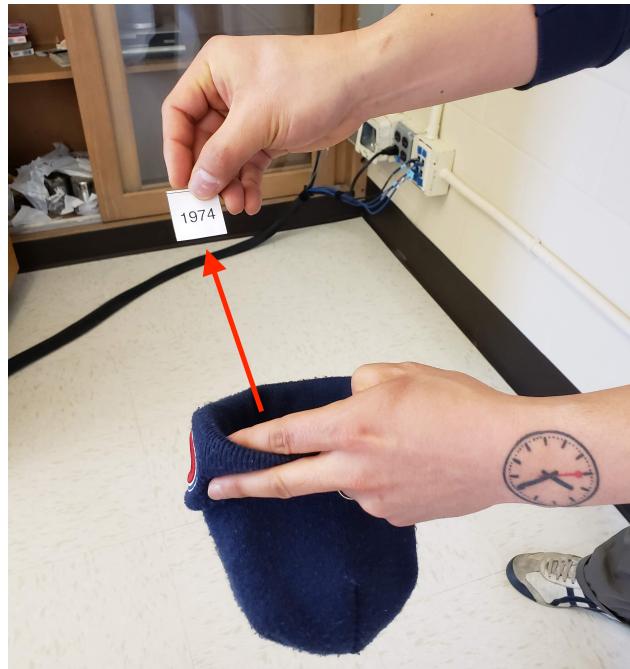


FIGURE 8.6: Step 3: Drawing one slip of paper at random.

Step 4: Put the slip of paper back in the hat! In other words, replace it as seen in Figure 8.7.



FIGURE 8.7: Step 4: Replacing slip of paper.

Step 5: Repeat Steps 3 and 4 a total of 49 more times, resulting in 50 recorded years.

What we just performed was a *resampling* of the original sample of 50 pennies. We are not sampling 50 pennies from the population of all US pennies as we did in our trip to the bank. Instead, we are mimicking this act by resampling 50 pennies from our original sample of 50 pennies.

Now ask yourselves, why did we replace our resampled slip of paper back into the hat in Step 4? Because if we left the slip of paper out of the hat each time we performed Step 4, we would end up with the same 50 original pennies! In other words, replacing the slips of paper induces *sampling variation*.

Being more precise with our terminology, we just performed a *resampling with replacement* from the original sample of 50 pennies. Had we left the slip of paper out of the hat each time we performed Step 4, this would be *resampling without replacement*.

Let's study our 50 resampled pennies via an exploratory data analysis. First, let's load the data into R by manually creating a data frame `pennies_resample` of our 50 resampled values. We'll do this using the `tibble()` command from the `dplyr` package. Note that the 50 values you resample will almost certainly not be the same as ours given the inherent randomness.

```
pennies_resample <- tibble(  
  year = c(1976, 1962, 1976, 1983, 2017, 2015, 2015, 1962, 2016, 1976,  
  2006, 1997, 1988, 2015, 2015, 1988, 2016, 1978, 1979, 1997,  
  1974, 2013, 1978, 2015, 2008, 1982, 1986, 1979, 1981, 2004,  
  2000, 1995, 1999, 2006, 1979, 2015, 1979, 1998, 1981, 2015,  
  2000, 1999, 1988, 2017, 1992, 1997, 1990, 1988, 2006, 2000)  
)
```

The 50 values of `year` in `pennies_resample` represent a resample of size 50 from the original sample of 50 pennies. We display the 50 resampled pennies in Figure 8.8.



FIGURE 8.8: 50 resampled US pennies labelled.

Let's compare the distribution of the numerical variable `year` of our 50 resampled pennies with the distribution of the numerical variable `year` of our original sample of 50 pennies in Figure 8.9.

```
ggplot(pennies_resample, aes(x = year)) +
  geom_histogram(binwidth = 10, color = "white") +
  labs(title = "Resample of 50 pennies")
ggplot(pennies_sample, aes(x = year)) +
  geom_histogram(binwidth = 10, color = "white") +
  labs(title = "Original sample of 50 pennies")
```

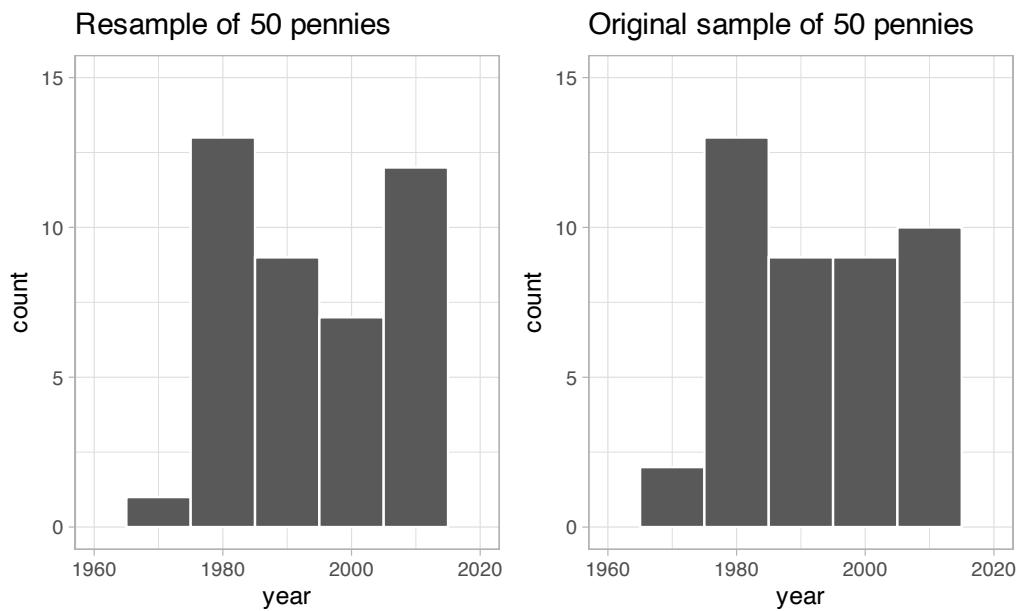


FIGURE 8.9: Comparing `year` in the resampled `pennies_resample` with the original sample `pennies_sample`.

Observe in Figure 8.9 that while the general shapes of both distributions of `year` are roughly similar, they are not identical.

Recall from the previous section that the sample mean of the original sample of 50 pennies from the bank was 1995.44. What about for our resample? Any guesses? Let's have `dplyr` help us out as before:

```
pennies_resample %>%
  summarize(mean_year = mean(year))
```

```
# A tibble: 1 × 1
  mean_year
  <dbl>
1     1996
```

We obtained a different mean year of 1996. This variation is induced by the resampling *with replacement* we performed earlier.

What if we repeated this resampling exercise many times? Would we obtain the same mean `year` each time? In other words, would our guess at the mean year of all pennies in the US in 2019 be exactly 1996 every time? Just as we did in Chapter 7, let's perform this resampling activity with the help of some of our friends: 35 friends in total.

8.1.3 Resampling 35 times

Each of our 35 friends will repeat the same five steps:

1. Start with 50 identically sized slips of paper representing the 50 pennies.
2. Put the 50 small pieces of paper into a hat or beanie cap.
3. Mix the hat's contents and draw one slip of paper at random. Record the year in a spreadsheet.
4. Replace the slip of paper back in the hat!
5. Repeat Steps 3 and 4 a total of 49 more times, resulting in 50 recorded years.

Since we had 35 of our friends perform this task, we ended up with $35 \cdot 50 = 1750$ values. We recorded these values in a shared spreadsheet² with 50 rows (plus a header row) and 35 columns. We display a snapshot of the first 10 rows and five columns of this shared spreadsheet in Figure 8.10.

Arianna	Artemis	Bea	Camryn	Cassandra
1988	2018	2016	2002	2015
2002	1988	1971	1997	1976
2015	1999	1986	2002	2015
1998	2015	2002	2013	1981
1979	1962	1992	1997	1988
1971	2004	1976	1979	1985
1971	2018	2015	2018	1979
2015	1988	1985	1971	1971
1988	2013	1976	1998	1978
1979	1988	1999	1996	1979
1982	2008	2013	1999	1986
2004	1983	1997	1983	1974

FIGURE 8.10: Snapshot of shared spreadsheet of resampled pennies.

For your convenience, we've taken these $35 \cdot 50 = 1750$ values and saved them in `pennies_resamples`, a “tidy” data frame included in the `moderndive` package. We saw what it means for a data frame to be “tidy” in Subsection 4.2.1.

```
pennies_resamples
```

```
# A tibble: 1,750 x 3
# Groups:   name [35]
  replicate name    year
```

²https://docs.google.com/spreadsheets/d/1y3kOsU_wDrDd5eiJbEtLeHT9L5SvpZb_TrzwFBsouk0/

```
<int> <chr> <dbl>
1      1 Arianna 1988
2      1 Arianna 2002
3      1 Arianna 2015
4      1 Arianna 1998
5      1 Arianna 1979
6      1 Arianna 1971
7      1 Arianna 1971
8      1 Arianna 2015
9      1 Arianna 1988
10     1 Arianna 1979
# i 1,740 more rows
```

What did each of our 35 friends obtain as the mean year? Once again, `dplyr` to the rescue! After grouping the rows by `name`, we summarize each group of 50 rows by their mean year:

```
resampled_means <- pennies_resamples %>%
  group_by(name) %>%
  summarize(mean_year = mean(year))
resampled_means
```

```
# A tibble: 35 x 2
  name      mean_year
  <chr>        <dbl>
1 Arianna    1992.5
2 Artemis    1996.42
3 Bea        1996.32
4 Camryn    1996.9
5 Cassandra  1991.22
6 Cindy      1995.48
7 Claire     1995.52
8 Dahlia    1998.48
9 Dan        1993.86
10 Eindra    1993.56
# i 25 more rows
```

Observe that `resampled_means` has 35 rows corresponding to the 35 means based on the 35 resamples. Furthermore, observe the variation in the 35 values in the variable `mean_year`. Let's visualize this variation using a histogram in Figure 8.11. Recall that adding the argument `boundary = 1990` to the `geom_histogram()` sets the binning structure so that one of the bin boundaries is at 1990 exactly.

```
ggplot(resampled_means, aes(x = mean_year)) +
  geom_histogram(binwidth = 1, color = "white", boundary = 1990) +
  labs(x = "Sampled mean year")
```

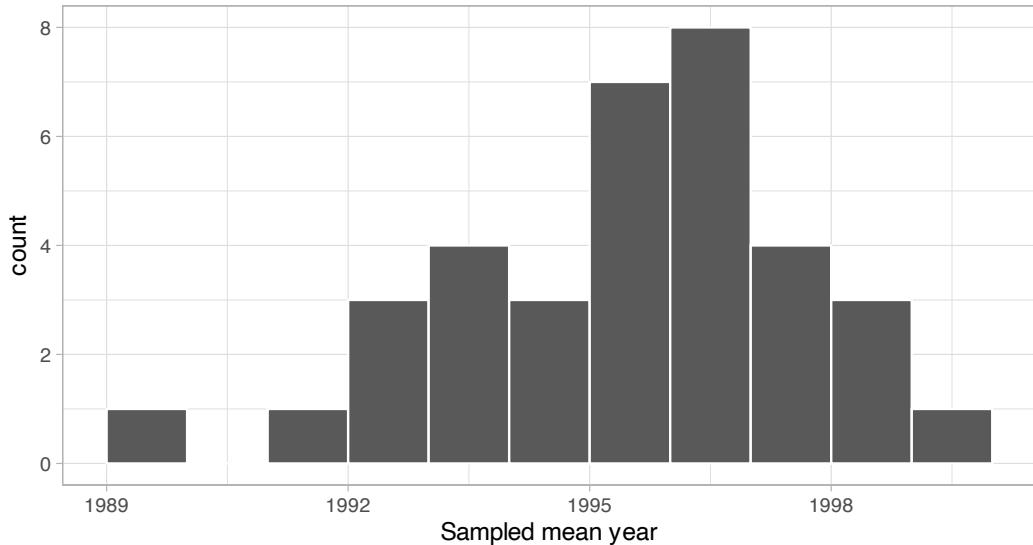


FIGURE 8.11: Distribution of 35 sample means from 35 resamples.

Observe in Figure 8.11 that the distribution looks roughly normal and that we rarely observe sample mean years less than 1992 or greater than 2000. Also observe how the distribution is roughly centered at 1995, which is close to the sample mean of 1995.44 of the *original sample* of 50 pennies from the bank.

8.1.4 What did we just do?

What we just demonstrated in this activity is the statistical procedure known as *bootstrap resampling with replacement*. We used *resampling* to mimic the sampling variation we studied in Chapter 7 on sampling. However, in this case, we did so using only a *single* sample from the population.

In fact, the histogram of sample means from 35 resamples in Figure 8.11 is called the *bootstrap distribution*. It is an *approximation* to the *sampling distribution* of the sample mean, in the sense that both distributions will have a similar shape and similar spread. In fact in the upcoming Section 8.7, we'll show you that this is the case. Using this bootstrap distribution, we can study the effect of sampling variation on our estimates. In particular, we'll study the typical “error” of our estimates, known as the *standard error*.

In Section 8.2 we'll mimic our tactile resampling activity virtually on the computer, allowing us to quickly perform the resampling many more than 35 times. In Section 8.3 we'll define the statistical concept of a *confidence interval*, which builds off the concept of bootstrap distributions.

In Section 8.4, we'll construct confidence intervals using the `dplyr` package, as well as a new package: the `infer` package for “tidy” and transparent statistical inference. We'll introduce the “tidy” statistical inference framework that was the motivation for the `infer` package pipeline. The `infer` package will be the driving package throughout the rest of this book.

As we did in Chapter 7, we'll tie all these ideas together with a real-life case study in Section 8.6. This time we'll look at data from an experiment about yawning from the US television show *Mythbusters*.

8.2 Computer simulation of resampling

Let's now mimic our tactile resampling activity virtually with a computer.

8.2.1 Virtually resampling once

First, let's perform the virtual analog of resampling once. Recall that the `pennies_sample` data frame included in the `moderndive` package contains the years of our original sample of 50 pennies from the bank. Furthermore, recall in Chapter 7 on sampling that we used the `rep_sample_n()` function as a virtual shovel to sample balls from our virtual bowl of 2400 balls as follows:

```
virtual_shovel <- bowl %>%
  rep_sample_n(size = 50)
```

Let's modify this code to perform the resampling with replacement of the 50 slips of paper representing our original sample 50 pennies:

```
virtual_resample <- pennies_sample %>%
  rep_sample_n(size = 50, replace = TRUE)
```

Observe how we explicitly set the `replace` argument to `TRUE` in order to tell `rep_sample_n()` that we would like to sample pennies *with* replacement. Had we not set `replace = TRUE`, the function would've assumed the default value of `FALSE` and

hence done resampling *without* replacement. Additionally, since we didn't specify the number of replicates via the `reps` argument, the function assumes the default of one replicate `reps = 1`. Lastly, observe also that the `size` argument is set to match the original sample size of 50 pennies.

Let's look at only the first 10 out of 50 rows of `virtual_resample`:

```
virtual_resample
```

```
# A tibble: 50 x 3
# Groups:   replicate [1]
  replicate     ID   year
  <int> <int> <dbl>
1       1      37 1962
2       1      1 2002
3       1      45 1997
4       1      28 2006
5       1      50 2017
6       1      10 2000
7       1      16 2015
8       1      47 1982
9       1      23 1998
10      1      44 2015
# i 40 more rows
```

The `replicate` variable only takes on the value of 1 corresponding to us only having `reps = 1`, the `ID` variable indicates which of the 50 pennies from `pennies_sample` was resampled, and `year` denotes the year of minting. Let's now compute the mean year in our virtual resample of size 50 using data wrangling functions included in the `dplyr` package:

```
virtual_resample %>%
  summarize(resample_mean = mean(year))
```

```
# A tibble: 1 x 2
  replicate resample_mean
  <int>          <dbl>
1       1          1996
```

As we saw when we did our tactile resampling exercise, the resulting mean year is different than the mean year of our 50 originally sampled pennies of 1995.44.

8.2.2 Virtually resampling 35 times

Let's now perform the virtual analog of our 35 friends' resampling. Using these results, we'll be able to study the variability in the sample means from 35 resamples of size 50. Let's first add a `reps = 35` argument to `rep_sample_n()` to indicate we would like 35 replicates. Thus, we want to repeat the resampling with the replacement of 50 pennies 35 times.

```
virtual_resamples <- pennies_sample %>%
  rep_sample_n(size = 50, replace = TRUE, reps = 35)
virtual_resamples
```

```
# A tibble: 1,750 x 3
# Groups:   replicate [35]
  replicate   ID   year
  <int> <int> <dbl>
1       1     21 1981
2       1     34 1985
3       1      4 1988
4       1     11 1994
5       1     26 1979
6       1      8 1996
7       1     19 1983
8       1     21 1981
9       1     49 2006
10      1      2 1986
# i 1,740 more rows
```

The resulting `virtual_resamples` data frame has $35 \cdot 50 = 1750$ rows corresponding to 35 resamples of 50 pennies. Let's now compute the resulting 35 sample means using the same `dplyr` code as we did in the previous section, but this time adding a `group_by(replicate)`:

```
virtual_resampled_means <- virtual_resamples %>%
  group_by(replicate) %>%
  summarize(mean_year = mean(year))
virtual_resampled_means
```

```
# A tibble: 35 x 2
  replicate mean_year
  <int>     <dbl>
```

```

1      1  1995.58
2      2  1999.74
3      3  1993.7
4      4  1997.1
5      5  1999.42
6      6  1995.12
7      7  1994.94
8      8  1997.78
9      9  1991.26
10     10 1996.88
# i 25 more rows

```

Observe that `virtual_resampled_means` has 35 rows, corresponding to the 35 resampled means. Furthermore, observe that the values of `mean_year` vary. Let's visualize this variation using a histogram in Figure 8.12.

```

ggplot(virtual_resampled_means, aes(x = mean_year)) +
  geom_histogram(binwidth = 1, color = "white", boundary = 1990) +
  labs(x = "Resample mean year")

```

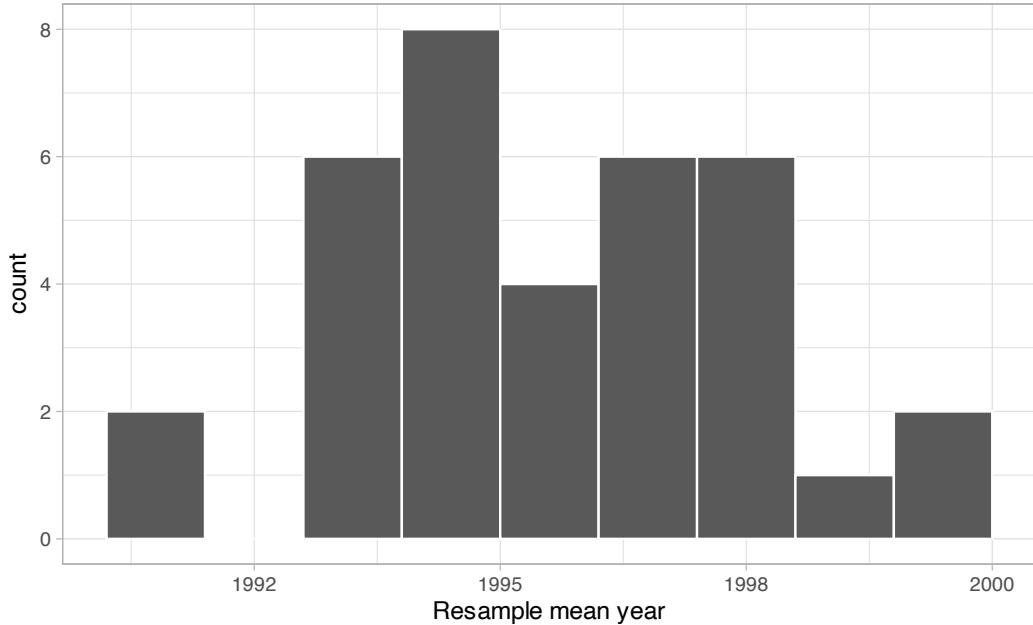


FIGURE 8.12: Distribution of 35 sample means from 35 resamples.

Let's compare our virtually constructed bootstrap distribution with the one our 35 friends constructed via our tactile resampling exercise in Figure 8.13. Observe how they are somewhat similar, but not identical.

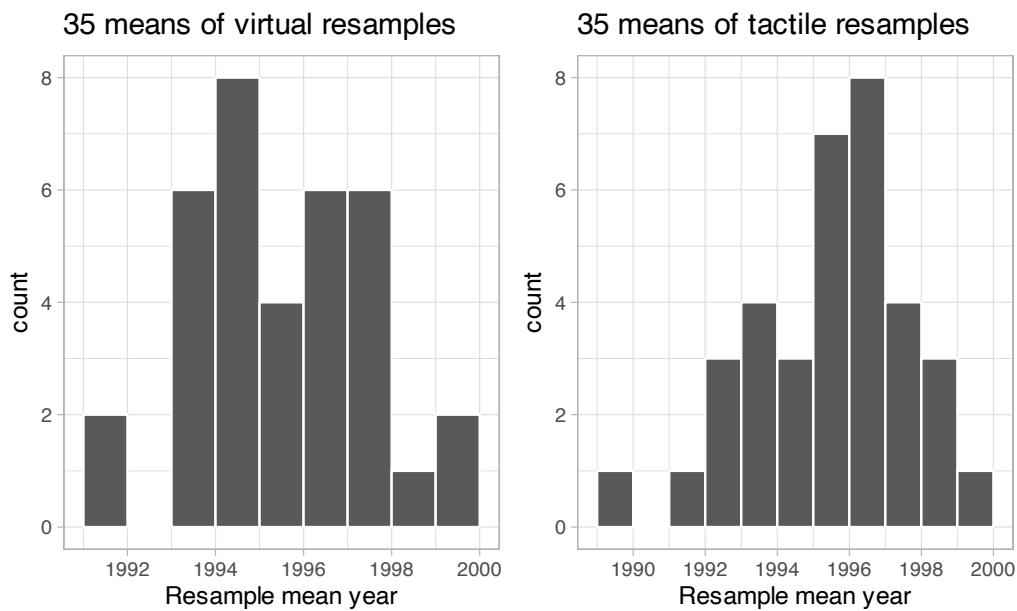


FIGURE 8.13: Comparing distributions of means from resamples.

Recall that in the “resampling with replacement” scenario we are illustrating here, both of these histograms have a special name: the *bootstrap distribution of the sample mean*. Furthermore, recall they are an approximation to the *sampling distribution* of the sample mean, a concept you saw in Chapter 7 on sampling. These distributions allow us to study the effect of sampling variation on our estimates of the true population mean, in this case the true mean year for *all* US pennies. However, unlike in Chapter 7 where we took multiple samples (something one would never do in practice), bootstrap distributions are constructed by taking multiple resamples from a *single* sample: in this case, the 50 original pennies from the bank.

8.2.3 Virtually resampling 1000 times

Remember that one of the goals of resampling with replacement is to construct the bootstrap distribution, which is an approximation of the sampling distribution. However, the bootstrap distribution in Figure 8.12 is based only on 35 resamples and hence looks a little coarse. Let’s increase the number of resamples to 1000, so that we can hopefully better see the shape and the variability between different resamples.

```
# Repeat resampling 1000 times
virtual_resamples <- pennies_sample %>%
  rep_sample_n(size = 50, replace = TRUE, reps = 1000)
```

```
# Compute 1000 sample means
virtual_resampled_means <- virtual_resamples %>%
  group_by(replicate) %>%
  summarize(mean_year = mean(year))
```

However, in the interest of brevity, going forward let's combine these two operations into a single chain of pipe (%>%) operators:

```
virtual_resampled_means <- pennies_sample %>%
  rep_sample_n(size = 50, replace = TRUE, reps = 1000) %>%
  group_by(replicate) %>%
  summarize(mean_year = mean(year))
virtual_resampled_means
```

```
# A tibble: 1,000 x 2
  replicate mean_year
  <int>     <dbl>
1       1    1992.6
2       2    1994.78
3       3    1994.74
4       4    1997.88
5       5    1990
6       6    1999.48
7       7    1990.26
8       8    1993.2
9       9    1994.88
10      10   1996.3
# i 990 more rows
```

In Figure 8.14 let's visualize the bootstrap distribution of these 1000 means based on 1000 virtual resamples:

```
ggplot(virtual_resampled_means, aes(x = mean_year)) +
  geom_histogram(binwidth = 1, color = "white", boundary = 1990) +
  labs(x = "sample mean")
```

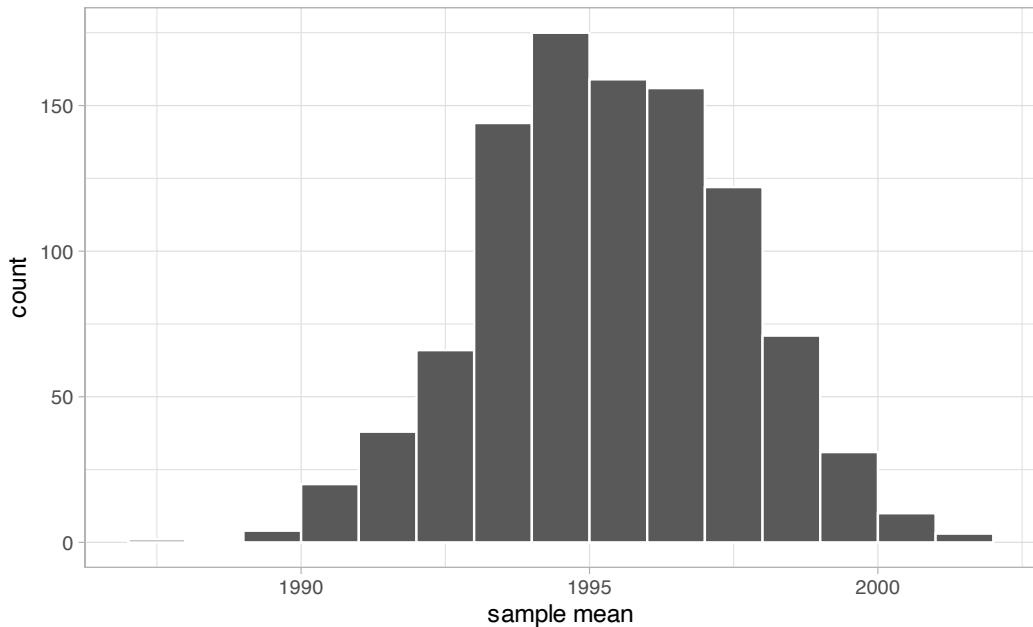


FIGURE 8.14: Bootstrap resampling distribution based on 1000 resamples.

Note here that the bell shape is starting to become much more apparent. We now have a general sense for the range of values that the sample mean may take on. But where is this histogram centered? Let's compute the mean of the 1000 resample means:

```
virtual_resampled_means %>%
  summarize(mean_of_means = mean(mean_year))
```

```
# A tibble: 1 x 1
  mean_of_means
  <dbl>
1 1995.36
```

The mean of these 1000 means is 1995.36, which is quite close to the mean of our original sample of 50 pennies of 1995.44. This is the case since each of the 1000 resamples is based on the original sample of 50 pennies.

Congratulations! You've just constructed your first bootstrap distribution! In the next section, you'll see how to use this bootstrap distribution to construct *confidence intervals*.

Learning check

(LC8.1) What is the chief difference between a bootstrap distribution and a sampling distribution?

(LC8.2) Looking at the bootstrap distribution for the sample mean in Figure 8.14, between what two values would you say *most* values lie?

8.3 Understanding confidence intervals

Let's start this section with an analogy involving fishing. Say you are trying to catch a fish. On the one hand, you could use a spear, while on the other you could use a net. Using the net will probably allow you to catch more fish!

Now think back to our pennies exercise where you are trying to estimate the true population mean year μ of *all* US pennies. Think of the value of μ as a fish.

On the one hand, we could use the appropriate *point estimate/sample statistic* to estimate μ , which we saw in Table 8.1 is the sample mean \bar{x} . Based on our sample of 50 pennies from the bank, the sample mean was 1995.44. Think of using this value as "fishing with a spear."

What would "fishing with a net" correspond to? Look at the bootstrap distribution in Figure 8.14 once more. Between which two years would you say that "most" sample means lie? While this question is somewhat subjective, saying that most sample means lie between 1992 and 2000 would not be unreasonable. Think of this interval as the "net."

What we've just illustrated is the concept of a *confidence interval*, which we'll abbreviate with "CI" throughout this book. As opposed to a point estimate/sample statistic that estimates the value of an unknown population parameter with a single value, a *confidence interval* gives what can be interpreted as a range of plausible values. Going back to our analogy, point estimates/sample statistics can be thought of as spears, whereas confidence intervals can be thought of as nets.

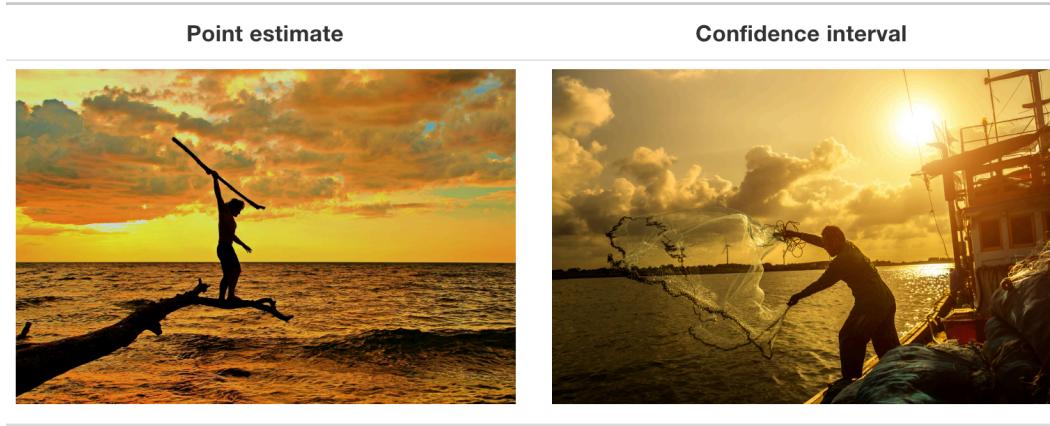


FIGURE 8.15: Analogy of difference between point estimates and confidence intervals.

Our proposed interval of 1992 to 2000 was constructed by eye and was thus somewhat subjective. We now introduce two methods for constructing such intervals in a more exact fashion: the *percentile method* and the *standard error method*.

Both methods for confidence interval construction share some commonalities. First, they are both constructed from a bootstrap distribution, as you constructed in Sub-section 8.2.3 and visualized in Figure 8.14.

Second, they both require you to specify the *confidence level*. Commonly used confidence levels include 90%, 95%, and 99%. All other things being equal, higher confidence levels correspond to wider confidence intervals, and lower confidence levels correspond to narrower confidence intervals. In this book, we'll be mostly using 95% and hence constructing “95% confidence intervals for μ ” for our pennies activity.

8.3.1 Percentile method

One method to construct a confidence interval is to use the middle 95% of values of the bootstrap distribution. We can do this by computing the 2.5th and 97.5th percentiles, which are 1991.059 and 1999.283, respectively. This is known as the *percentile method* for constructing confidence intervals.

For now, let's focus only on the concepts behind a percentile method constructed confidence interval; we'll show you the code that computes these values in the next section.

Let's mark these percentiles on the bootstrap distribution with vertical lines in Figure 8.16. About 95% of the `mean_year` variable values in `virtual_resampled_means` fall between 1991.059 and 1999.283, with 2.5% to the left of the leftmost line and 2.5% to the right of the rightmost line.

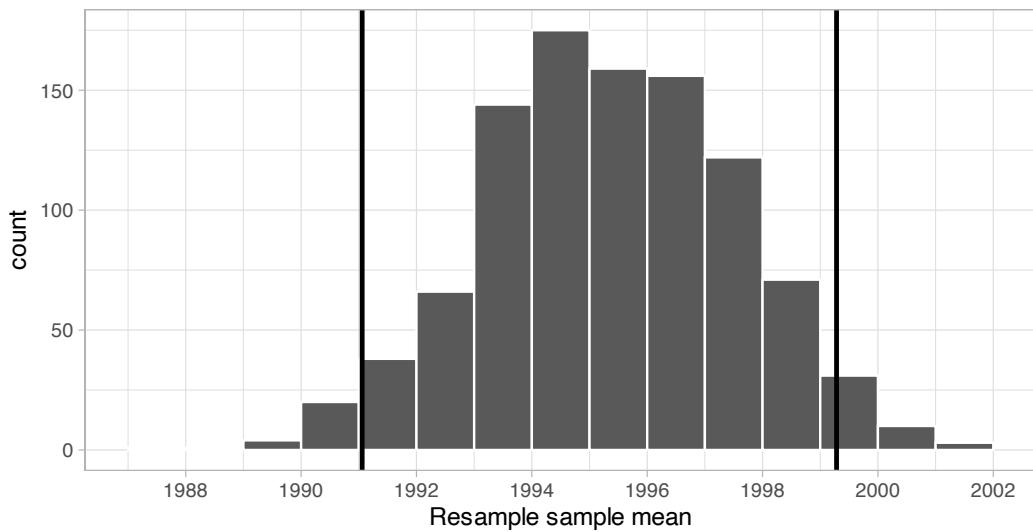


FIGURE 8.16: Percentile method 95% confidence interval. Interval endpoints marked by vertical lines.

8.3.2 Standard error method

Recall in Appendix A.2, we saw that if a numerical variable follows a normal distribution, or, in other words, the histogram of this variable is bell-shaped, then roughly 95% of values fall between ± 1.96 standard deviations of the mean. Given that our bootstrap distribution based on 1000 resamples with replacement in Figure 8.14 is normally shaped, let's use this fact about normal distributions to construct a confidence interval in a different way.

First, recall the bootstrap distribution has a mean equal to 1995.36. This value almost coincides exactly with the value of the sample mean \bar{x} of our original 50 pennies of 1995.44. Second, let's compute the standard deviation of the bootstrap distribution using the values of `mean_year` in the `virtual_resampled_means` data frame:

```
virtual_resampled_means %>%
  summarize(SE = sd(mean_year))
```

```
# A tibble: 1 × 1
  SE
  <dbl>
1 2.15466
```

What is this value? Recall that the bootstrap distribution is an approximation to the sampling distribution. Recall also that the standard deviation of a sampling

distribution has a special name: the *standard error*. Putting these two facts together, we can say that 2.155 is an approximation of the standard error of \bar{x} .

Thus, using our 95% rule of thumb about normal distributions from Appendix A.2, we can use the following formula to determine the lower and upper endpoints of a 95% confidence interval for μ :

$$\begin{aligned}\bar{x} \pm 1.96 \cdot SE &= (\bar{x} - 1.96 \cdot SE, \bar{x} + 1.96 \cdot SE) \\ &= (1995.44 - 1.96 \cdot 2.15, 1995.44 + 1.96 \cdot 2.15) \\ &= (1991.15, 1999.73)\end{aligned}$$

Let's now add the SE method confidence interval with dashed lines in Figure 8.17.

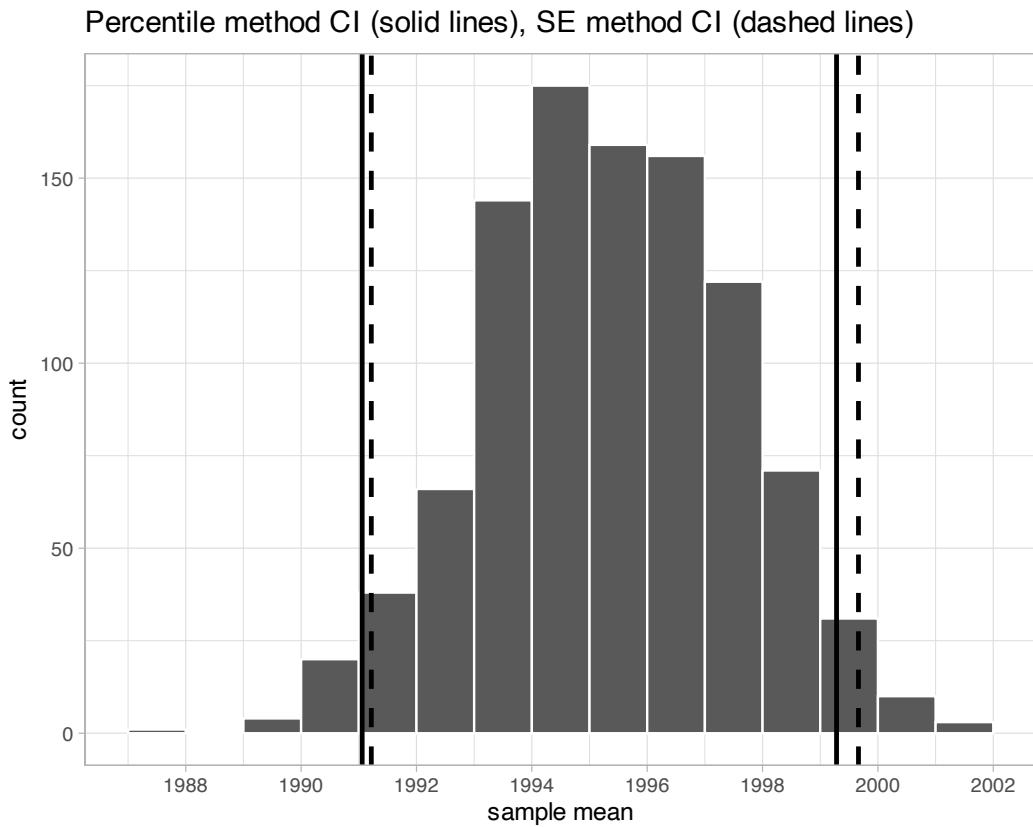


FIGURE 8.17: Comparing two 95% confidence interval methods.

We see that both methods produce nearly identical 95% confidence intervals for μ with the percentile method yielding (1991.06, 1999.28) while the standard error method produces (1991.22, 1999.66). However, recall that we can only use the standard error rule when the bootstrap distribution is roughly normally shaped.

Now that we've introduced the concept of confidence intervals and laid out the intuition behind two methods for constructing them, let's explore the code that allows us to construct them.

Learning check

(LC8.3) What condition about the bootstrap distribution must be met for us to be able to construct confidence intervals using the standard error method?

(LC8.4) Say we wanted to construct a 68% confidence interval instead of a 95% confidence interval for μ . Describe what changes are needed to make this happen. Hint: we suggest you look at Appendix A.2 on the normal distribution.

8.4 Constructing confidence intervals

Recall that the process of resampling with replacement we performed by hand in Section 8.1 and virtually in Section 8.2 is known as *bootstrapping*. The term bootstrapping originates in the expression of “pulling oneself up by their bootstraps,” meaning to “succeed only by one’s own efforts or abilities.”³

From a statistical perspective, bootstrapping alludes to succeeding in being able to study the effects of sampling variation on estimates from the “effort” of a single sample. Or more precisely, it refers to constructing an approximation to the sampling distribution using only one sample.

To perform this resampling with replacement virtually in Section 8.2, we used the `rep_sample_n()` function, making sure that the size of the resamples matched the original sample size of 50. In this section, we'll build off these ideas to construct confidence intervals using a new package: the `infer` package for “tidy” and transparent statistical inference.

8.4.1 Original workflow

Recall that in Section 8.2, we virtually performed bootstrap resampling with replacement to construct bootstrap distributions. Such distributions are approximations to the sampling distributions we saw in Chapter 7, but are constructed using only a single sample. Let's revisit the original workflow using the `%>%` pipe operator.

³https://en.wiktionary.org/wiki/pull_onceself_up_by_one%27s_bootstraps

First, we used the `rep_sample_n()` function to resample `size = 50` pennies with replacement from the original sample of 50 pennies in `pennies_sample` by setting `replace = TRUE`. Furthermore, we repeated this resampling 1000 times by setting `reps = 1000`:

```
pennies_sample %>%
  rep_sample_n(size = 50, replace = TRUE, reps = 1000)
```

Second, since for each of our 1000 resamples of size 50, we wanted to compute a separate sample mean, we used the `dplyr` verb `group_by()` to group observations/rows together by the `replicate` variable...

```
pennies_sample %>%
  rep_sample_n(size = 50, replace = TRUE, reps = 1000) %>%
  group_by(replicate)
```

... followed by using `summarize()` to compute the sample `mean()` year for each `replicate` group:

```
pennies_sample %>%
  rep_sample_n(size = 50, replace = TRUE, reps = 1000) %>%
  group_by(replicate) %>%
  summarize(mean_year = mean(year))
```

For this simple case, we can get by with using the `rep_sample_n()` function and a couple of `dplyr` verbs to construct the bootstrap distribution. However, using only `dplyr` verbs only provides us with a limited set of tools. For more complicated situations, we'll need a little more firepower. Let's repeat this using the `infer` package.

8.4.2 `infer` package workflow

The `infer` package is an R package for statistical inference. It makes efficient use of the `%>%` pipe operator we introduced in Section 3.1 to spell out the sequence of steps necessary to perform statistical inference in a “tidy” and transparent fashion. Furthermore, just as the `dplyr` package provides functions with verb-like names to perform data wrangling, the `infer` package provides functions with intuitive verb-like names to perform statistical inference.

Let's go back to our pennies. Previously, we computed the value of the sample mean \bar{x} using the `dplyr` function `summarize()`:

```
pennies_sample %>%
  summarize(stat = mean(year))
```

We'll see that we can also do this using `infer` functions `specify()` and `calculate()`:

```
pennies_sample %>%
  specify(response = year) %>%
  calculate(stat = "mean")
```

You might be asking yourself: “Isn’t the `infer` code longer? Why would I use that code?”. While not immediately apparent, you’ll see that there are three chief benefits to the `infer` workflow as opposed to the `dplyr` workflow.

First, the `infer` verb names better align with the overall resampling framework you need to understand to construct confidence intervals and to conduct hypothesis tests (in Chapter 9). We’ll see flowchart diagrams of this framework in the upcoming Figure 8.23 and in Chapter 9 with Figure 9.10.

Second, you can jump back and forth seamlessly between confidence intervals and hypothesis testing with minimal changes to your code. This will become apparent in Subsection 9.3.2 when we’ll compare the `infer` code for both of these inferential methods.

Third, the `infer` workflow is much simpler for conducting inference when you have *more than one variable*. We’ll see two such situations. We’ll first see situations of *two-sample* inference where the sample data is collected from two groups, such as in Section 8.6 where we study the contagiousness of yawning and in Section 9.1 where we compare promotion rates of two groups at banks in the 1970s. Then in Section 10.4, we’ll see situations of *inference for regression* using the regression models you fit in Chapter 5.

Let’s now illustrate the sequence of verbs necessary to construct a confidence interval for μ , the population mean year of minting of all US pennies in 2019.

1. specify variables

FIGURE 8.18: Diagram of the `specify()` verb.

As shown in Figure 8.18, the `specify()` function is used to choose which variables in a data frame will be the focus of our statistical inference. We do this by specifying the `response` argument. For example, in our `pennies_sample` data frame of the 50 pennies sampled from the bank, the variable of interest is `year`:

```
pennies_sample %>%
  specify(response = year)
```

```
Response: year (numeric)
# A tibble: 50 x 1
  year
  <dbl>
 1 2002
 2 1986
 3 2017
 4 1988
 5 2008
 6 1983
 7 2008
 8 1996
 9 2004
10 2000
# i 40 more rows
```

Notice how the data itself doesn't change, but the `Response: year (numeric)` *meta-data* does. This is similar to how the `group_by()` verb from `dplyr` doesn't change the data, but only adds “grouping” meta-data, as we saw in Section 3.4.

We can also specify which variables will be the focus of our statistical inference using a `formula = y ~ x`. This is the same formula notation you saw in Chapters 5 and 6 on

regression models: the response variable y is separated from the explanatory variable x by a \sim (“tilde”). The following use of `specify()` with the `formula` argument yields the same result seen previously:

```
pennies_sample %>%
  specify(formula = year ~ NULL)
```

Since in the case of pennies we only have a response variable and no explanatory variable of interest, we set the x on the right-hand side of the \sim to be `NULL`.

While in the case of the pennies either specification works just fine, we’ll see examples later on where the `formula` specification is simpler. In particular, this comes up in the upcoming Section 8.6 on comparing two proportions and Section 10.4 on inference for regression.

2. generate replicates

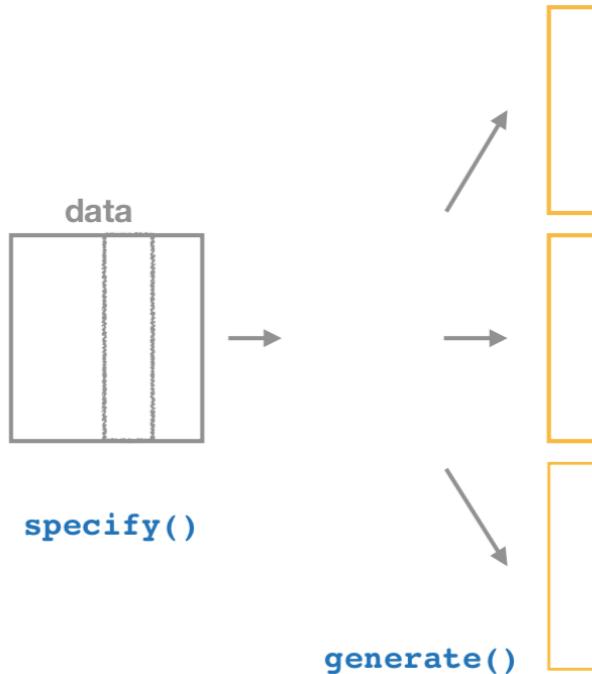


FIGURE 8.19: Diagram of `generate()` replicates.

After we `specify()` the variables of interest, we pipe the results into the `generate()` function to generate replicates. Figure 8.19 shows how this is combined with `specify()` to start the pipeline. In other words, repeat the resampling process a large number of times. Recall in Sections 8.2.2 and 8.2.3 we did this 35 and 1000 times.

The `generate()` function's first argument is `reps`, which sets the number of replicates we would like to generate. Since we want to resample the 50 pennies in `pennies_sample` with replacement 1000 times, we set `reps = 1000`. The second argument `type` determines the type of computer simulation we'd like to perform. We set this to `type = "bootstrap"` indicating that we want to perform bootstrap resampling. You'll see different options for `type` in Chapter 9.

```
pennies_sample %>%
  specify(response = year) %>%
  generate(reps = 1000, type = "bootstrap")
```

```
Response: year (numeric)
# A tibble: 50,000 x 2
# Groups:   replicate [1,000]
  replicate year
  <int> <dbl>
1       1 1981
2       1 1988
3       1 2006
4       1 2016
5       1 2002
6       1 1985
7       1 1979
8       1 2000
9       1 2006
10      1 2016
# i 49,990 more rows
```

Observe that the resulting data frame has 50,000 rows. This is because we performed resampling of 50 pennies with replacement 1000 times and $50,000 = 50 \cdot 1000$.

The variable `replicate` indicates which resample each row belongs to. So it has the value 1 50 times, the value 2 50 times, all the way through to the value 1000 50 times. The default value of the `type` argument is `"bootstrap"` in this scenario, so if the last line was written as `generate(reps = 1000)`, we'd obtain the same results.

Comparing with original workflow: Note that the steps of the `infer` workflow so far produce the same results as the original workflow using the `rep_sample_n()` function we saw earlier. In other words, the following two code chunks produce similar results:

<pre># infer workflow: pennies_sample %>% specify(response = year) %>% generate(reps = 1000)</pre>	<pre># Original workflow: pennies_sample %>% rep_sample_n(size = 50, replace = TRUE, reps = 1000)</pre>
--	--

3. calculate summary statistics

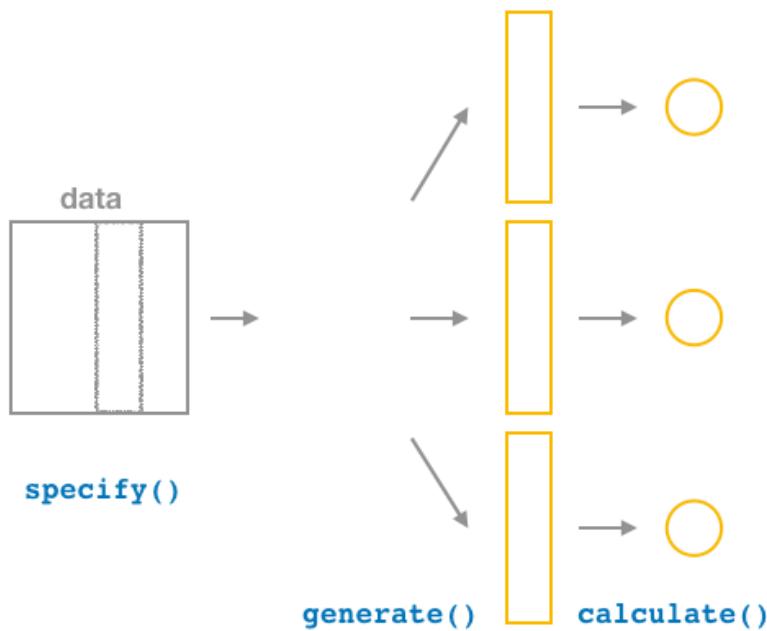


FIGURE 8.20: Diagram of calculate() summary statistics.

After we `generate()` many replicates of bootstrap resampling with replacement, we next want to summarize each of the 1000 resamples of size 50 to a single sample statistic value. As seen in the diagram, the `calculate()` function does this.

In our case, we want to calculate the mean year for each bootstrap resample of size 50. To do so, we set the `stat` argument to `"mean"`. You can also set the `stat` argument to a variety of other common summary statistics, like `"median"`, `"sum"`, `"sd"` (standard deviation), and `"prop"` (proportion). To see a list of all possible summary statistics you can use, type `?calculate` and read the help file.

Let's save the result in a data frame called `bootstrap_distribution` and explore its contents:

```

bootstrap_distribution <- pennies_sample %>%
  specify(response = year) %>%
  generate(reps = 1000) %>%
  calculate(stat = "mean")
bootstrap_distribution
  
```

A tibble: 1,000 x 2

```

replicate      stat
<int>    <dbl>
1          1 1995.7
2          2 1994.04
3          3 1993.62
4          4 1994.5
5          5 1994.08
6          6 1993.6
7          7 1995.26
8          8 1996.64
9          9 1994.3
10        10 1995.94
# i 990 more rows

```

Observe that the resulting data frame has 1000 rows and 2 columns corresponding to the 1000 replicate values. It also has the mean year for each bootstrap resample saved in the variable stat.

Comparing with original workflow: You may have recognized at this point that the calculate() step in the infer workflow produces the same output as the group_by() %>% summarize() steps in the original workflow.

<pre># infer workflow: pennies_sample %>% specify(response = year) %>% generate(reps = 1000) %>% calculate(stat = "mean")</pre>	<pre># Original workflow: pennies_sample %>% rep_sample_n(size = 50, replace = TRUE, reps = 1000) %>% group_by(replicate) %>% summarize(stat = mean(year))</pre>
--	--

4. visualize the results

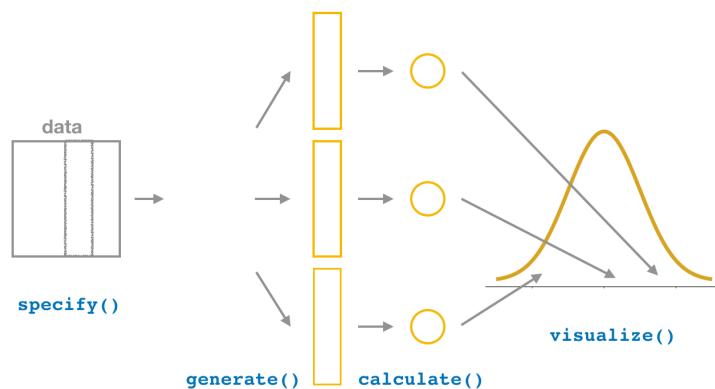


FIGURE 8.21: Diagram of visualize() results.

The `visualize()` verb provides a quick way to visualize the bootstrap distribution as a histogram of the numerical `stat` variable's values. The pipeline of the main `infer` verbs used for exploring bootstrap distribution results is shown in Figure 8.21.

```
visualize(bootstrap_distribution)
```

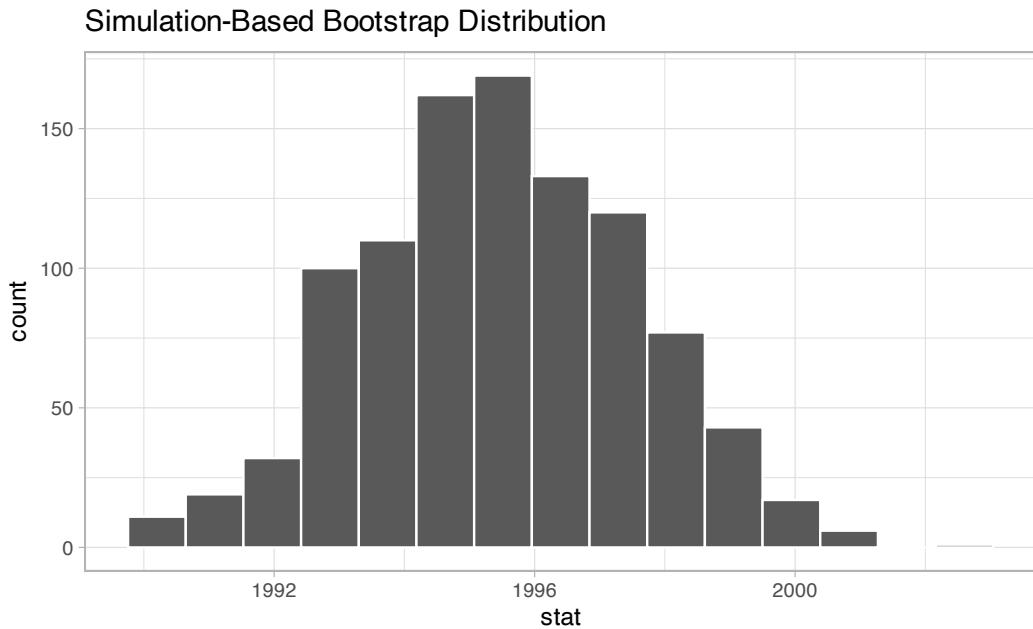


FIGURE 8.22: Bootstrap distribution.

Comparing with original workflow: In fact, `visualize()` is a *wrapper function* for the `ggplot()` function that uses a `geom_histogram()` layer. Recall that we illustrated the concept of a wrapper function in Figure 5.5 in Subsection 5.1.2.

```
# infer workflow:                      # Original workflow:
visualize(bootstrap_distribution)      ggplot(bootstrap_distribution,
                                                aes(x = stat)) +
                                                geom_histogram()
```

The `visualize()` function can take many other arguments which we'll see momentarily to customize the plot further. It also works with helper functions to do the shading of the histogram values corresponding to the confidence interval values.

Let's recap the steps of the `infer` workflow for constructing a bootstrap distribution and then visualizing it in Figure 8.23.

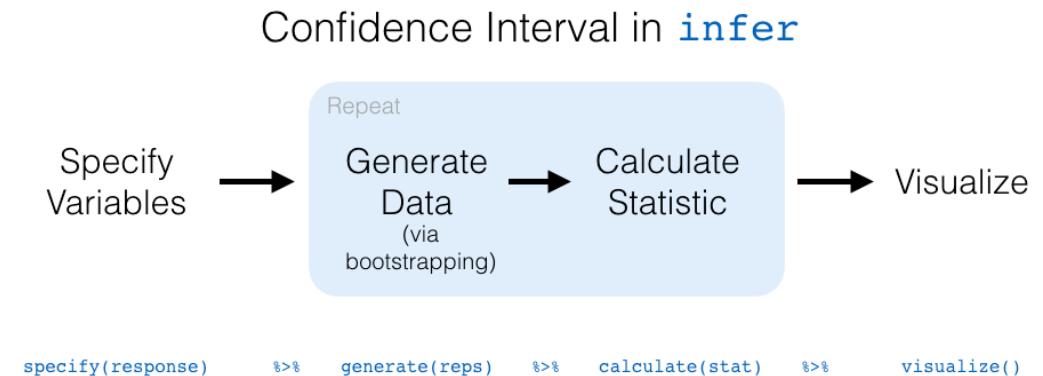


FIGURE 8.23: `infer` package workflow for confidence intervals.

Recall how we introduced two different methods for constructing 95% confidence intervals for an unknown population parameter in Section 8.3: the *percentile method* and the *standard error method*. Let's now check out the `infer` package code that explicitly constructs these. There are also some additional neat functions to visualize the resulting confidence intervals built-in to the `infer` package!

8.4.3 Percentile method with `infer`

Recall the percentile method for constructing 95% confidence intervals we introduced in Subsection 8.3.1. This method sets the lower endpoint of the confidence interval at the 2.5th percentile of the bootstrap distribution and similarly sets the upper endpoint at the 97.5th percentile. The resulting interval captures the middle 95% of the values of the sample mean in the bootstrap distribution.

We can compute the 95% confidence interval by piping `bootstrap_distribution` into the `get_confidence_interval()` function from the `infer` package, with the `confidence_level` set to 0.95 and the `confidence_interval_type` to be "percentile". Let's save the results in `percentile_ci`.

```

percentile_ci <- bootstrap_distribution %>%
  get_confidence_interval(level = 0.95, type = "percentile")
percentile_ci
    
```

```

# A tibble: 1 x 2
  lower_ci upper_ci
  <dbl>    <dbl>
1 1991.24  1999.42
    
```

Alternatively, we can visualize the interval (1991.24, 1999.42) by piping the `bootstrap_distribution` data frame into the `visualize()` function and adding a `shade_confidence_interval()` layer. We set the `endpoints` argument to be `percentile_ci`.

```
visualize(bootstrap_distribution) +
  shade_confidence_interval(endpoints = percentile_ci)
```

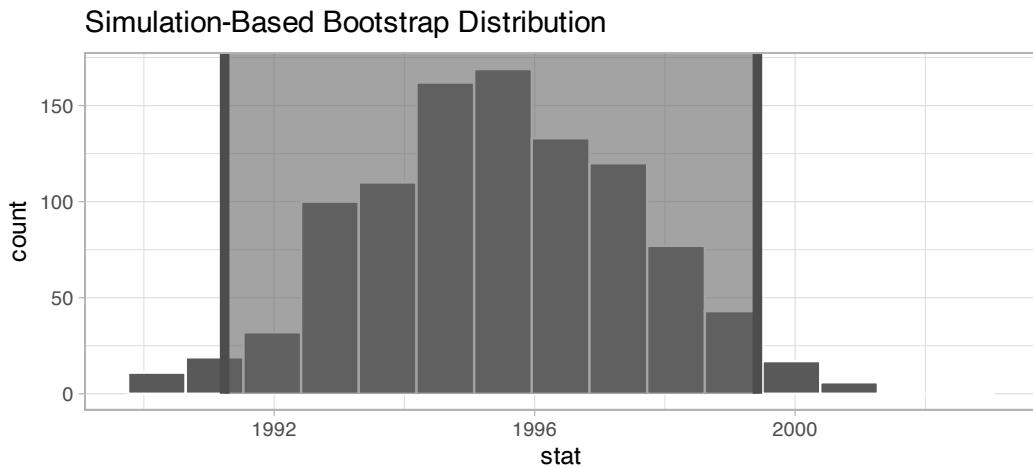


FIGURE 8.24: Percentile method 95% confidence interval shaded corresponding to potential values.

Observe in Figure 8.24 that 95% of the sample means stored in the `stat` variable in `bootstrap_distribution` fall between the two endpoints marked with the darker lines, with 2.5% of the sample means to the left of the shaded area and 2.5% of the sample means to the right. You also have the option to change the colors of the shading using the `color` and `fill` arguments.

You can also use the shorter named function `shade_ci()` and the results will be the same. This is for folks who don't want to type out all of `confidence_interval` and prefer to type out `ci` instead. Try out the following code!

```
visualize(bootstrap_distribution) +
  shade_ci(endpoints = percentile_ci, color = "hotpink", fill = "khaki")
```

8.4.4 Standard error method with `infer`

Recall the standard error method for constructing 95% confidence intervals we introduced in Subsection 8.3.2. For any distribution that is normally shaped, roughly

95% of the values lie within two standard deviations of the mean. In the case of the bootstrap distribution, the standard deviation has a special name: the *standard error*.

So in our case, 95% of values of the bootstrap distribution will lie within ± 1.96 standard errors of \bar{x} . Thus, a 95% confidence interval is

$$\bar{x} \pm 1.96 \cdot SE = (\bar{x} - 1.96 \cdot SE, \bar{x} + 1.96 \cdot SE).$$

Computation of the 95% confidence interval can once again be done by piping the `bootstrap_distribution` data frame we created into the `get_confidence_interval()` function. However, this time we set the first `type` argument to be "se". Second, we must specify the `point_estimate` argument in order to set the center of the confidence interval. We set this to be the sample mean of the original sample of 50 pennies of 1995.44 we saved in `x_bar` earlier.

```
standard_error_ci <- bootstrap_distribution %>%  
  get_confidence_interval(type = "se", point_estimate = x_bar)
```

Using `level = 0.95` to compute confidence interval.

```
standard_error_ci
```

```
# A tibble: 1 x 2  
  lower_ci upper_ci  
    <dbl>    <dbl>  
1 1991.35 1999.53
```

If we would like to visualize the interval (1991.35, 1999.53), we can once again pipe the `bootstrap_distribution` data frame into the `visualize()` function and add a `shade_confidence_interval()` layer to our plot. We set the `endpoints` argument to be `standard_error_ci`. The resulting standard-error method based on a 95% confidence interval for μ can be seen in Figure 8.25.

```
visualize(bootstrap_distribution) +  
  shade_confidence_interval(endpoints = standard_error_ci)
```

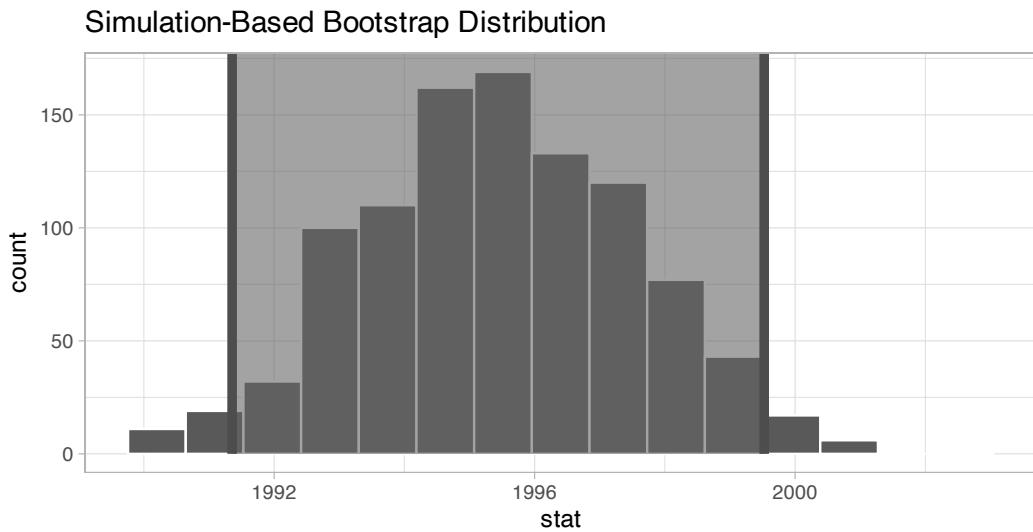


FIGURE 8.25: Standard-error-method 95% confidence interval.

As noted in Section 8.3, both methods produce similar confidence intervals:

- Percentile method: (1991.24, 1999.42)
- Standard error method: (1991.35, 1999.53)

Learning check

(LC8.5) Construct a 95% confidence interval for the *median* year of minting of *all* US pennies. Use the percentile method and, if appropriate, then use the standard-error method.

8.5 Interpreting confidence intervals

Now that we've shown you how to construct confidence intervals using a sample drawn from a population, let's now focus on how to interpret their effectiveness. The effectiveness of a confidence interval is judged by whether or not it contains the true value of the population parameter. Going back to our fishing analogy in Section 8.3, this is like asking, "Did our net capture the fish?".

So, for example, does our percentile-based confidence interval of (1991.24, 1999.42) “capture” the true mean year μ of *all* US pennies? Alas, we’ll never know, because we don’t know what the true value of μ is. After all, we’re sampling to estimate it!

In order to interpret a confidence interval’s effectiveness, we need to *know* what the value of the population parameter is. That way we can say whether or not a confidence interval “captured” this value.

Let’s revisit our sampling bowl from Chapter 7. What proportion of the bowl’s 2400 balls are red? Let’s compute this:

```
bowl %>%
  summarize(p_red = mean(color == "red"))
```

```
# A tibble: 1 × 1
  p_red
  <dbl>
1 0.375
```

In this case, we *know* what the value of the population parameter is: we know that the population proportion p is 0.375. In other words, we know that 37.5% of the bowl’s balls are red.

As we stated in Subsection 7.3.3, the sampling bowl exercise doesn’t really reflect how sampling is done in real life, but rather was an *idealized* activity. In real life, we won’t know what the true value of the population parameter is, hence the need for estimation.

Let’s now construct confidence intervals for p using our 33 groups of friends’ samples from the bowl in Chapter 7. We’ll then see if the confidence intervals “captured” the true value of p , which we know to be 37.5%. That is to say, “Did the net capture the fish?”.

8.5.1 Did the net capture the fish?

Recall that we had 33 groups of friends each take samples of size 50 from the bowl and then compute the sample proportion of red balls \hat{p} . This resulted in 33 such estimates of p . Let’s focus on Ilyas and Yohan’s sample, which is saved in the `bowl_sample_1` data frame in the `moderndive` package:

```
bowl_sample_1
```

```
# A tibble: 50 × 1
```

```

color
<chr>
1 white
2 white
3 red
4 red
5 white
6 white
7 red
8 white
9 white
10 white
# i 40 more rows

```

They observed 21 red balls out of 50 and thus their sample proportion \hat{p} was $21/50 = 0.42 = 42\%$. Think of this as the “spear” from our fishing analogy.

Let’s now follow the `infer` package workflow from Subsection 8.4.2 to create a percentile-method-based 95% confidence interval for p using Ilyas and Yohan’s sample. Think of this as the “net.”

1. specify variables

First, we `specify()` the response variable of interest `color`:

```

bowl_sample_1 %>%
  specify(response = color)

```

Error: A level of the response variable `color` needs to be specified for the `success` argument in `specify()`.

Whoops! We need to define which event is of interest! `red` or `white` balls? Since we are interested in the proportion `red`, let’s set `success` to be "red":

```

bowl_sample_1 %>%
  specify(response = color, success = "red")

```

```

Response: color (factor)
# A tibble: 50 x 1
  color
  <fct>

```

```
1 white
2 white
3 red
4 red
5 white
6 white
7 red
8 white
9 white
10 white
# i 40 more rows
```

2. generate replicates

Second, we generate() 1000 replicates of *bootstrap resampling with replacement* from bowl_sample_1 by setting reps = 1000 and type = "bootstrap".

```
bowl_sample_1 %>%
  specify(response = color, success = "red") %>%
  generate(reps = 1000, type = "bootstrap")
```

```
Response: color (factor)
# A tibble: 50,000 x 2
# Groups:   replicate [1,000]
  replicate color
  <int> <fct>
1       1 white
2       1 white
3       1 white
4       1 white
5       1 red
6       1 white
7       1 white
8       1 white
9       1 white
10      1 red
# i 49,990 more rows
```

Observe that the resulting data frame has 50,000 rows. This is because we performed resampling of 50 balls with replacement 1000 times and thus $50,000 = 50 \cdot 1000$. The variable replicate indicates which resample each row belongs to. So it has the value 1 50 times, the value 2 50 times, all the way through to the value 1000 50 times.

3. calculate summary statistics

Third, we summarize each of the 1000 resamples of size 50 with the proportion of *successes*. In other words, the proportion of the balls that are "red". We can set the summary statistic to be calculated as the proportion by setting the `stat` argument to be "`prop`". Let's save the result as `sample_1_bootstrap`:

```
sample_1_bootstrap <- bowl_sample_1 %>%
  specify(response = color, success = "red") %>%
  generate(reps = 1000, type = "bootstrap") %>%
  calculate(stat = "prop")
sample_1_bootstrap
```

```
# A tibble: 1,000 x 2
  replicate   stat
  <int> <dbl>
1       1  0.32
2       2  0.42
3       3  0.44
4       4  0.4
5       5  0.44
6       6  0.52
7       7  0.38
8       8  0.44
9       9  0.34
10      10  0.42
# i 990 more rows
```

Observe there are 1000 rows in this data frame and thus 1000 values of the variable `stat`. These 1000 values of `stat` represent our 1000 replicated values of the proportion, each based on a different resample.

4. visualize the results

Fourth and lastly, let's compute the resulting 95% confidence interval.

```
percentile_ci_1 <- sample_1_bootstrap %>%
  get_confidence_interval(level = 0.95, type = "percentile")
percentile_ci_1
```

```
# A tibble: 1 x 2
```

```
lower_ci  upper_ci
<dbl>    <dbl>
1      0.3     0.56
```

Let's visualize the bootstrap distribution along with the `percentile_ci_1` percentile-based 95% confidence interval for p in Figure 8.26. We'll adjust the number of bins to better see the resulting shape. Furthermore, we'll add a dashed vertical line at Ilyas and Yohan's observed $\hat{p} = 21/50 = 0.42 = 42\%$ using `geom_vline()`.

```
sample_1_bootstrap %>%
  visualize(bins = 15) +
  shade_confidence_interval(endpoints = percentile_ci_1) +
  geom_vline(xintercept = 0.42, linetype = "dashed")
```

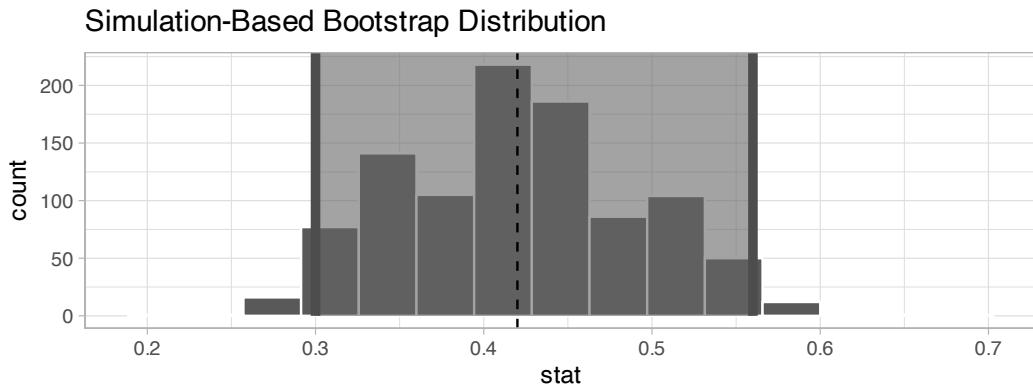


FIGURE 8.26: Bootstrap distribution.

Did Ilyas and Yohan's net capture the fish? Did their 95% confidence interval for p based on their sample contain the true value of p of 0.375? Yes! 0.375 is between the endpoints of their confidence interval (0.3, 0.56).

However, will *every* 95% confidence interval for p capture this value? In other words, if we had a different sample of 50 balls and constructed a different confidence interval, would it necessarily contain $p = 0.375$ as well? Let's see!

Let's first take a different sample from the bowl, this time using the computer as we did in Chapter 7:

```
bowl_sample_2 <- bowl %>% rep_sample_n(size = 50)
bowl_sample_2
```

```
# A tibble: 50 x 3
# Groups:   replicate [1]
  replicate ball_ID color
  <int>    <int> <chr>
1       1     1665 red
2       1     1312 red
3       1     2105 red
4       1      810 white
5       1     189 white
6       1    1429 white
7       1    2294 red
8       1    1233 white
9       1    1951 white
10      1    2061 white
# i 40 more rows
```

Let's reapply the same `infer` functions on `bowl_sample_2` to generate a different 95% confidence interval for p . First, we create the new bootstrap distribution and save the results in `sample_2_bootstrap`:

```
sample_2_bootstrap <- bowl_sample_2 %>%
  specify(response = color,
         success = "red") %>%
  generate(reps = 1000,
            type = "bootstrap") %>%
  calculate(stat = "prop")
sample_2_bootstrap
```

```
# A tibble: 1,000 x 2
  replicate  stat
  <int> <dbl>
1       1  0.48
2       2  0.38
3       3  0.32
4       4  0.32
5       5  0.34
6       6  0.26
7       7  0.3
8       8  0.36
9       9  0.44
10      10  0.36
# i 990 more rows
```

We once again compute a percentile-based 95% confidence interval for p :

```
percentile_ci_2 <- sample_2_bootstrap %>%
  get_confidence_interval(level = 0.95, type = "percentile")
percentile_ci_2
```

```
# A tibble: 1 × 2
  lower_ci upper_ci
  <dbl>    <dbl>
1     0.2     0.48
```

Does this new net capture the fish? In other words, does the 95% confidence interval for p based on the new sample contain the true value of p of 0.375? Yes again! 0.375 is between the endpoints of our confidence interval (0.2, 0.48).

Let's now repeat this process 100 more times: we take 100 virtual samples from the bowl and construct 100 95% confidence intervals. Let's visualize the results in Figure 8.27 where:

1. We mark the true value of $p = 0.375$ with a vertical line.
2. We mark each of the 100 95% confidence intervals with horizontal lines. These are the “nets.”
3. The horizontal line is colored grey if the confidence interval “captures” the true value of p marked with the vertical line. The horizontal line is colored black otherwise.

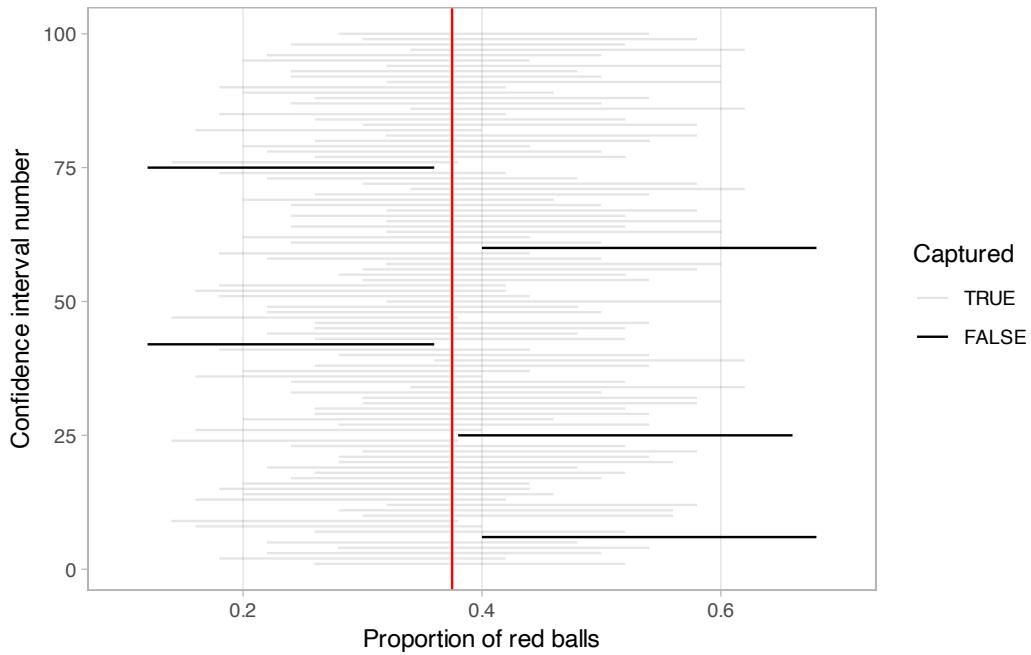


FIGURE 8.27: 100 percentile-based 95% confidence intervals for p .

Of the 100 95% confidence intervals, 95 of them captured the true value $p = 0.375$, whereas 5 of them didn't. In other words, 95 of our nets caught the fish, whereas 5 of our nets didn't.

This is where the “95% confidence level” we defined in Section 8.3 comes into play: for every 100 95% confidence intervals, we *expect* that 95 of them will capture p and that five of them won't.

Note that “expect” is a probabilistic statement referring to a long-run average. In other words, for every 100 confidence intervals, we will observe *about* 95 confidence intervals that capture p , but not necessarily exactly 95. In Figure 8.27 for example, 95 of the confidence intervals capture p .

To further accentuate our point about confidence levels, let's generate a figure similar to Figure 8.27, but this time constructing 80% standard-error method based confidence intervals instead. Let's visualize the results in Figure 8.28 with the scale on the x-axis being the same as in Figure 8.27 to make comparison easy. Furthermore, since all standard-error method confidence intervals for p are centered at their respective point estimates \hat{p} , we mark this value on each line with dots.

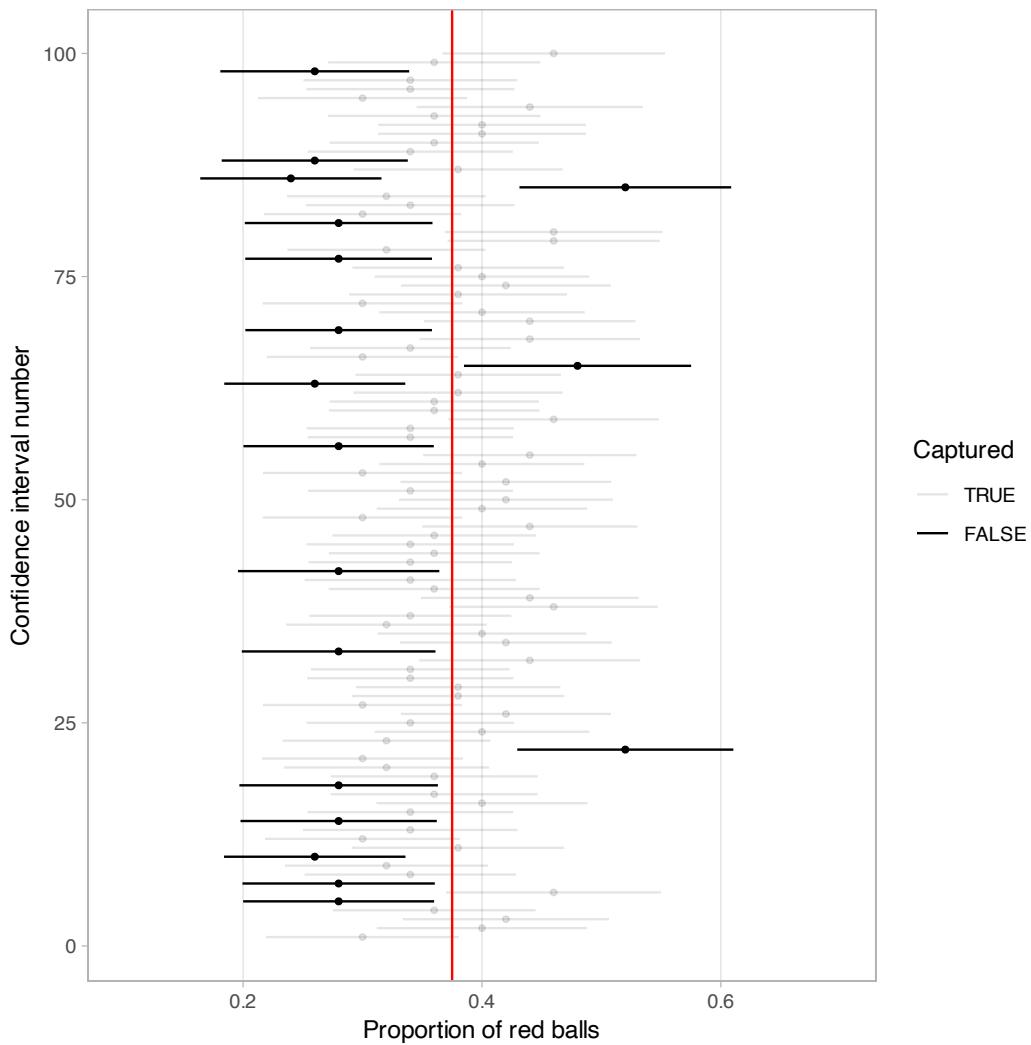


FIGURE 8.28: 100 SE-based 80% confidence intervals for p with point estimate center marked with dots.

Observe how the 80% confidence intervals are narrower than the 95% confidence intervals, reflecting our lower degree of confidence. Think of this as using a smaller “net.” We’ll explore other determinants of confidence interval width in the upcoming Subsection 8.5.3.

Furthermore, observe that of the 100 80% confidence intervals, 82 of them captured the population proportion $p = 0.375$, whereas 18 of them did not. Since we lowered the confidence level from 95% to 80%, we now have a much larger number of confidence intervals that failed to “catch the fish.”

8.5.2 Precise and shorthand interpretation

Let's return our attention to 95% confidence intervals. The precise and mathematically correct interpretation of a 95% confidence interval is a little long-winded:

Precise interpretation: If we repeated our sampling procedure a large number of times, we expect about 95% of the resulting confidence intervals to capture the value of the population parameter.

This is what we observed in Figure 8.27. Our confidence interval construction procedure is 95% *reliable*. That is to say, we can expect our confidence intervals to include the true population parameter about 95% of the time.

A common but incorrect interpretation is: "There is a 95% probability that the confidence interval contains p ." Looking at Figure 8.27, each of the confidence intervals either does or doesn't contain p . In other words, the probability is either a 1 or a 0.

So if the 95% confidence level only relates to the reliability of the confidence interval construction procedure and not to a given confidence interval itself, what insight can be derived from a given confidence interval? For example, going back to the pennies example, we found that the percentile method 95% confidence interval for μ was (1991.24, 1999.42), whereas the standard error method 95% confidence interval was (1991.35, 1999.53). What can be said about these two intervals?

Loosely speaking, we can think of these intervals as our "best guess" of a plausible range of values for the mean year μ of *all* US pennies. For the rest of this book, we'll use the following shorthand summary of the precise interpretation.

Short-hand interpretation: We are 95% "confident" that a 95% confidence interval captures the value of the population parameter.

We use quotation marks around "confident" to emphasize that while 95% relates to the reliability of our confidence interval construction procedure, ultimately a constructed confidence interval is our best guess of an interval that contains the population parameter. In other words, it's our best net.

So returning to our pennies example and focusing on the percentile method, we are 95% "confident" that the true mean year of pennies in circulation in 2019 is somewhere between 1991.24 and 1999.42.

8.5.3 Width of confidence intervals

Now that we know how to interpret confidence intervals, let's go over some factors that determine their width.

Impact of confidence level

One factor that determines confidence interval widths is the pre-specified confidence level. For example, in Figures 8.27 and 8.28, we compared the widths of 95% and 80% confidence intervals and observed that the 95% confidence intervals were wider. The quantification of the confidence level should match what many expect of the word "confident." In order to be more confident in our best guess of a range of values, we need to widen the range of values.

To elaborate on this, imagine we want to guess the forecasted high temperature in Seoul, South Korea on August 15th. Given Seoul's temperate climate with four distinct seasons, we could say somewhat confidently that the high temperature would be between 50°F - 95°F (10°C - 35°C). However, if we wanted a temperature range we were *absolutely* confident about, we would need to widen it.

We need this wider range to allow for the possibility of anomalous weather, like a freak cold spell or an extreme heat wave. So a range of temperatures we could be near certain about would be between 32°F - 110°F (0°C - 43°C). On the other hand, if we could tolerate being a little less confident, we could narrow this range to between 70°F - 85°F (21°C - 30°C).

Let's revisit our sampling bowl from Chapter 7. Let's compare $10 \cdot 3 = 30$ confidence intervals for p based on three different confidence levels: 80%, 95%, and 99%.

Specifically, we'll first take 30 different random samples of size $n = 50$ balls from the bowl. Then we'll construct 10 percentile-based confidence intervals using each of the three different confidence levels.

Finally, we'll compare the widths of these intervals. We visualize the resulting confidence intervals in Figure 8.29 along with a vertical line marking the true value of $p = 0.375$.

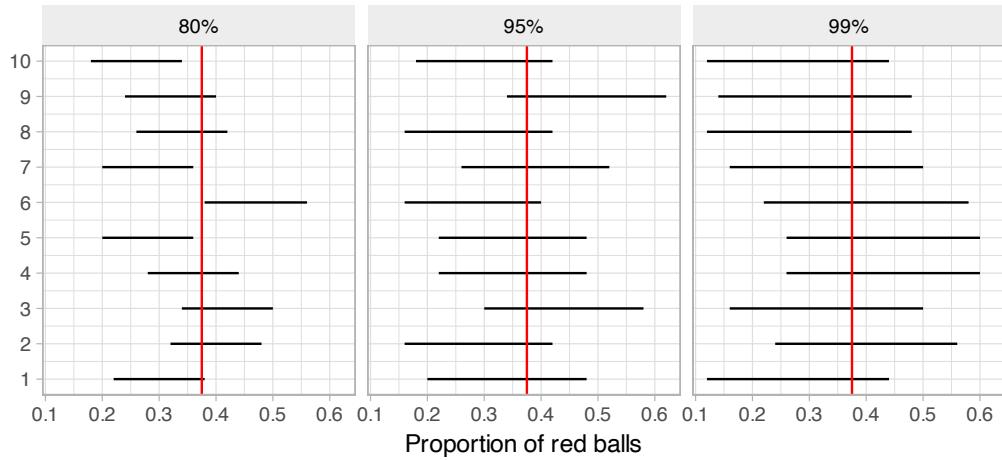


FIGURE 8.29: Ten 80, 95, and 99% confidence intervals for p based on $n = 50$.

Observe that as the confidence level increases from 80% to 95% to 99%, the confidence intervals tend to get wider as seen in Table 8.2 where we compare their average widths.

TABLE 8.2: Average width of 80, 95, and 99% confidence intervals

Confidence level	Mean width
80%	0.162
95%	0.262
99%	0.338

So in order to have a higher confidence level, our confidence intervals must be wider. Ideally, we would have both a high confidence level and narrow confidence intervals. However, we cannot have it both ways. If we want to *be more confident*, we need to allow for wider intervals. Conversely, if we would like a narrow interval, we must tolerate a lower confidence level.

The moral of the story is: **Higher confidence levels tend to produce wider confidence intervals.** When looking at Figure 8.29 it is important to keep in mind that we kept the sample size fixed at $n = 50$. Thus, all $10 \cdot 3 = 30$ random samples from the bowl had the same sample size. What happens if instead we took samples of different sizes? Recall that we did this in Subsection 7.2 using virtual shovels with 25, 50, and 100 slots.

Impact of sample size

This time, let's fix the confidence level at 95%, but consider three different sample sizes for n : 25, 50, and 100. Specifically, we'll first take 10 different random samples of size 25, 10 different random samples of size 50, and 10 different random samples

of size 100. We'll then construct 95% percentile-based confidence intervals for each sample. Finally, we'll compare the widths of these intervals. We visualize the resulting 30 confidence intervals in Figure 8.30. Note also the vertical line marking the true value of $p = 0.375$.

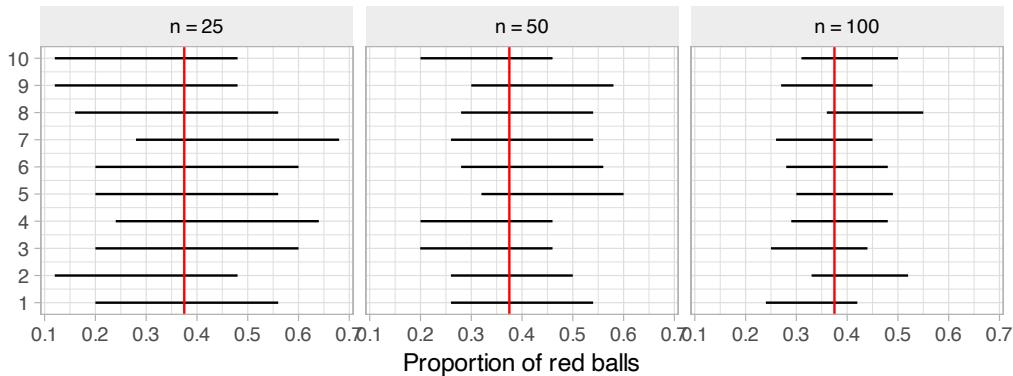


FIGURE 8.30: Ten 95% confidence intervals for p with $n = 25, 50$, and 100 .

Observe that as the confidence intervals are constructed from larger and larger sample sizes, they tend to get narrower. Let's compare the average widths in Table 8.3.

TABLE 8.3: Average width of 95% confidence intervals based on $n = 25$, 50 , and 100

Sample size	Mean width
$n = 25$	0.380
$n = 50$	0.268
$n = 100$	0.189

The moral of the story is: **Larger sample sizes tend to produce narrower confidence intervals.** Recall that this was a key message in Subsection 7.3.3. As we used larger and larger shovels for our samples, the sample proportions red \hat{p} tended to vary less. In other words, our estimates got more and more *precise*.

Recall that we visualized these results in Figure 7.15, where we compared the *sampling distributions* for \hat{p} based on samples of size n equal 25, 50, and 100. We also quantified the sampling variation of these sampling distributions using their standard deviation, which has that special name: the *standard error*. So as the sample size increases, the standard error decreases.

In fact, the standard error is another related factor in determining confidence interval width. We'll explore this fact in Subsection 8.7.2 when we discuss theory-based methods for constructing confidence intervals using mathematical formulas. Such methods are an alternative to the computer-based methods we've been using so far.

8.6 Case study: Is yawning contagious?

Let's apply our knowledge of confidence intervals to answer the question: "Is yawning contagious?". If you see someone else yawn, are you more likely to yawn? In an episode of the US show *Mythbusters*⁴, the hosts conducted an experiment to answer this question. The episode is available to view in the United States on the Discovery Network website here⁵ and more information about the episode is also available on IMDb⁶.

8.6.1 *Mythbusters* study data

Fifty adult participants who thought they were being considered for an appearance on the show were interviewed by a show recruiter. In the interview, the recruiter either yawned or did not. Participants then sat by themselves in a large van and were asked to wait. While in the van, the *Mythbusters* team watched the participants using a hidden camera to see if they yawned. The data frame containing the results of their experiment is available in the `mythbusters_yawn` data frame included in the `moderndive` package:

```
mythbusters_yawn
```

```
# A tibble: 50 x 3
  subj group  yawn
  <int> <chr> <chr>
1     1 seed   yes
2     2 control yes
3     3 seed   no
4     4 seed   yes
5     5 seed   no
6     6 control no
7     7 seed   yes
8     8 control no
9     9 control no
10    10 seed  no
# i 40 more rows
```

The variables are:

⁴<http://www.discovery.com/tv-shows/mythbusters/mythbusters-database/yawning-contagious/>

⁵<https://www.discovery.com/tv-shows/mythbusters/videos/is-yawning-contagious>

⁶<https://www.imdb.com/title/tt0768479/>

- `subj`: The participant ID with values 1 through 50.
- `group`: A binary *treatment* variable indicating whether the participant was exposed to yawning. "seed" indicates the participant was exposed to yawning while "control" indicates the participant was not.
- `yawn`: A binary *response* variable indicating whether the participant ultimately yawned.

Recall that you learned about treatment and response variables in Subsection 5.3.1 in our discussion on confounding variables.

Let's use some data wrangling to obtain counts of the four possible outcomes:

```
mythbusters_yawn %>%
  group_by(group, yawn) %>%
  summarise(count = n())
```

```
`summarise()` has grouped output by 'group'. You can override using the
`.groups` argument.
```

```
# A tibble: 4 x 3
# Groups:   group [2]
  group  yawn  count
  <chr>  <chr> <int>
1 control no      12
2 control yes     4
3 seed    no      24
4 seed    yes     10
```

Let's first focus on the "control" group participants who were not exposed to yawning. 12 such participants did not yawn, while 4 such participants did. So out of the 16 people who were not exposed to yawning, $4/16 = 0.25 = 25\%$ did yawn.

Let's now focus on the "seed" group participants who were exposed to yawning where 24 such participants did not yawn, while 10 such participants did yawn. So out of the 34 people who were exposed to yawning, $10/34 = 0.294 = 29.4\%$ did yawn. Comparing these two percentages, the participants who were exposed to yawning yawned $29.4\% - 25\% = 4.4\%$ more often than those who were not.

8.6.2 Sampling scenario

Let's review the terminology and notation related to sampling we studied in Subsection 7.3.1. In Chapter 7 our *study population* was the bowl of $N = 2400$ balls. Our *population parameter* of interest was the *population proportion* of these balls

that were red, denoted mathematically by p . In order to estimate p , we extracted a sample of 50 balls using the shovel and computed the relevant *point estimate*: the *sample proportion* that were red, denoted mathematically by \hat{p} .

Who is the study population here? All humans? All the people who watch the show *Mythbusters*? It's hard to say! This question can only be answered if we know how the show's hosts recruited participants! In other words, what was the *sampling methodology* used by the *Mythbusters* to recruit participants? We alas are not provided with this information. Only for the purposes of this case study, however, we'll *assume* that the 50 participants are a representative sample of all Americans given the popularity of this show. Thus, we'll be assuming that any results of this experiment will generalize to all $N = 327$ million Americans (2018 population).

Just like with our sampling bowl, the population parameter here will involve proportions. However, in this case it will be the *difference in population proportions* $p_{seed} - p_{control}$, where p_{seed} is the proportion of *all* Americans who if exposed to yawning will yawn themselves, and $p_{control}$ is the proportion of *all* Americans who if not exposed to yawning still yawn themselves. Correspondingly, the point estimate/sample statistic based the *Mythbusters*' sample of participants will be the *difference in sample proportions* $\hat{p}_{seed} - \hat{p}_{control}$. Let's extend Table 7.5 of scenarios of sampling for inference to include our latest scenario.

TABLE 8.4: Scenarios of sampling for inference

Scenario	Population parameter	Notation	Point estimate	Symbol(s)
1	Population proportion	p	Sample proportion	\hat{p}
2	Population mean	μ	Sample mean	\bar{x} or $\hat{\mu}$
3	Difference in population proportions	$p_1 - p_2$	Difference in sample proportions	$\hat{p}_1 - \hat{p}_2$

This is known as a *two-sample* inference situation since we have two separate samples. Based on their two-samples of size $n_{seed} = 34$ and $n_{control} = 16$, the point estimate is

$$\hat{p}_{seed} - \hat{p}_{control} = \frac{24}{34} - \frac{12}{16} = 0.04411765 \approx 4.4\%$$

However, say the *Mythbusters* repeated this experiment. In other words, say they recruited 50 new participants and exposed 34 of them to yawning and 16 not. Would they obtain the exact same estimated difference of 4.4%? Probably not, again, because of *sampling variation*.

How does this sampling variation affect their estimate of 4.4%? In other words, what would be a plausible range of values for this difference that accounts for this sampling variation? We can answer this question with confidence intervals! Furthermore, since the *Mythbusters* only have a single two-sample of 50 participants, they would have to construct a 95% confidence interval for $p_{seed} - p_{control}$ using *bootstrap resampling with replacement*.

We make a couple of important notes. First, for the comparison between the "seed" and "control" groups to make sense, however, both groups need to be *independent* from each other. Otherwise, they could influence each other's results. This means that a participant being selected for the "seed" or "control" group has no influence on another participant being assigned to one of the two groups. As an example, if there were a mother and her child as participants in the study, they wouldn't necessarily be in the same group. They would each be assigned randomly to one of the two groups of the explanatory variable.

Second, the order of the subtraction in the difference doesn't matter so long as you are consistent and tailor your interpretations accordingly. In other words, using a point estimate of $\hat{p}_{seed} - \hat{p}_{control}$ or $\hat{p}_{control} - \hat{p}_{seed}$ does not make a material difference, you just need to stay consistent and interpret your results accordingly.

8.6.3 Constructing the confidence interval

As we did in Subsection 8.4.2, let's first construct the bootstrap distribution for $\hat{p}_{seed} - \hat{p}_{control}$ and then use this to construct 95% confidence intervals for $p_{seed} - p_{control}$. We'll do this using the `infer` workflow again. However, since the difference in proportions is a new scenario for inference, we'll need to use some new arguments in the `infer` functions along the way.

1. specify variables

Let's take our `mythbusters_yawn` data frame and `specify()` which variables are of interest using the `y ~ x` formula interface where:

- Our response variable is `yawn`: whether or not a participant yawned. It has levels "yes" and "no".
- The explanatory variable is `group`: whether or not a participant was exposed to yawning. It has levels "seed" (exposed to yawning) and "control" (not exposed to yawning).

```
mythbusters_yawn %>%
  specify(formula = yawn ~ group)
```

Error: A level of the response variable 'yawn' needs to be specified for the 'success' argument in 'specify()'.

Alas, we got an error message similar to the one from Subsection 8.5.1: `infer` is telling us that one of the levels of the categorical variable `yawn` needs to be defined as the `success`. Recall that we define `success` to be the event of interest we are trying to

count and compute proportions of. Are we interested in those participants who "yes" yawned or those who "no" didn't yawn? This isn't clear to R or someone just picking up the code and results for the first time, so we need to set the `success` argument to "yes" as follows to improve the transparency of the code:

```
mythbusters_yawn %>%
  specify(formula = yawn ~ group, success = "yes")
```

```
Response: yawn (factor)
Explanatory: group (factor)
# A tibble: 50 x 2
  yawn   group
  <fct> <fct>
1 yes    seed
2 yes    control
3 no     seed
4 yes    seed
5 no     seed
6 no     control
7 yes    seed
8 no     control
9 no     control
10 no    seed
# i 40 more rows
```

2. generate replicates

Our next step is to perform *bootstrap resampling with replacement* like we did with the slips of paper in our pennies activity in Section 8.1. We saw how it works with both a single variable in computing bootstrap means in Section 8.4 and in computing bootstrap proportions in Section 8.5, but we haven't yet worked with bootstrapping involving multiple variables.

In the `infer` package, bootstrapping with multiple variables means that each *row* is potentially resampled. Let's investigate this by focusing only on the first six rows of `mythbusters_yawn`:

```
first_six_rows <- head(mythbusters_yawn)
first_six_rows
```

```
# A tibble: 6 x 3
  subj group  yawn
```

```

<int> <chr> <chr>
1    1 seed   yes
2    2 control yes
3    3 seed   no
4    4 seed   yes
5    5 seed   no
6    6 control no

```

When we bootstrap this data, we are potentially pulling the subject's readings multiple times. Thus, we could see the entries of "seed" for group and "no" for yawn together in a new row in a bootstrap sample. This is further seen by exploring the `sample_n()` function in `dplyr` on this smaller 6-row data frame comprised of `head(mythbusters_yawn)`. The `sample_n()` function can perform this bootstrapping procedure and is similar to the `rep_sample_n()` function in `infer`, except that it is not repeated, but rather only performs one sample with or without replacement.

```

first_six_rows %>%
  sample_n(size = 6, replace = TRUE)

```

```

# A tibble: 6 x 3
  subj group  yawn
  <int> <chr> <chr>
1    1 seed   yes
2    6 control no
3    1 seed   yes
4    5 seed   no
5    4 seed   yes
6    4 seed   yes

```

We can see that in this bootstrap sample generated from the first six rows of `mythbusters_yawn`, we have some rows repeated. The same is true when we perform the `generate()` step in `infer` as done in what follows. Using this fact, we generate 1000 replicates, or, in other words, we bootstrap resample the 50 participants with replacement 1000 times.

```

mythbusters_yawn %>%
  specify(formula = yawn ~ group, success = "yes") %>%
  generate(reps = 1000, type = "bootstrap")

```

```

Response: yawn (factor)
Explanatory: group (factor)

```

```
# A tibble: 50,000 x 3
# Groups:   replicate [1,000]
  replicate yawn group
  <int> <fct> <fct>
1       1 yes   seed
2       1 yes   control
3       1 no    control
4       1 no    control
5       1 yes   seed
6       1 yes   seed
7       1 yes   seed
8       1 yes   seed
9       1 no    seed
10      1 yes   seed
# i 49,990 more rows
```

Observe that the resulting data frame has 50,000 rows. This is because we performed resampling of 50 participants with replacement 1000 times and $50,000 = 1000 \cdot 50$. The variable `replicate` indicates which resample each row belongs to. So it has the value 1 50 times, the value 2 50 times, all the way through to the value 1000 50 times.

3. calculate summary statistics

After we generate() many replicates of bootstrap resampling with replacement, we next want to summarize the bootstrap resamples of size 50 with a single summary statistic, the difference in proportions. We do this by setting the `stat` argument to "diff in props":

```
mythbusters_yawn %>%
  specify(formula = yawn ~ group, success = "yes") %>%
  generate(reps = 1000, type = "bootstrap") %>%
  calculate(stat = "diff in props")
```

```
Error: Statistic is based on a difference; specify the `order` in which to
subtract the levels of the explanatory variable.
```

We see another error here. We need to specify the order of the subtraction. Is it $\hat{p}_{seed} - \hat{p}_{control}$ or $\hat{p}_{control} - \hat{p}_{seed}$. We specify it to be $\hat{p}_{seed} - \hat{p}_{control}$ by setting `order = c("seed", "control")`. Note that you could've also set `order = c("control", "seed")`. As we stated earlier, the order of the subtraction does not matter, so long as you stay consistent throughout your analysis and tailor your interpretations accordingly.

Let's save the output in a data frame `bootstrap_distribution_yawning`:

```
bootstrap_distribution_yawning <- mythbusters_yawn %>%
  specify(formula = yawn ~ group, success = "yes") %>%
  generate(reps = 1000, type = "bootstrap") %>%
  calculate(stat = "diff in props", order = c("seed", "control"))
bootstrap_distribution_yawning
```

```
# A tibble: 1,000 × 2
  replicate      stat
  <int>     <dbl>
1       1  0.0357143
2       2  0.229167 
3       3  0.00952381
4       4  0.0106952 
5       5  0.00483092
6       6  0.00793651
7       7 -0.0845588 
8       8 -0.00466200
9       9  0.164686 
10      10  0.124777
# i 990 more rows
```

Observe that the resulting data frame has 1000 rows and 2 columns corresponding to the 1000 `replicate` ID's and the 1000 differences in proportions for each bootstrap resample in `stat`.

4. visualize the results

In Figure 8.31 we `visualize()` the resulting bootstrap resampling distribution. Let's also add a vertical line at 0 by adding a `geom_vline()` layer.

```
visualize(bootstrap_distribution_yawning) +
  geom_vline(xintercept = 0)
```

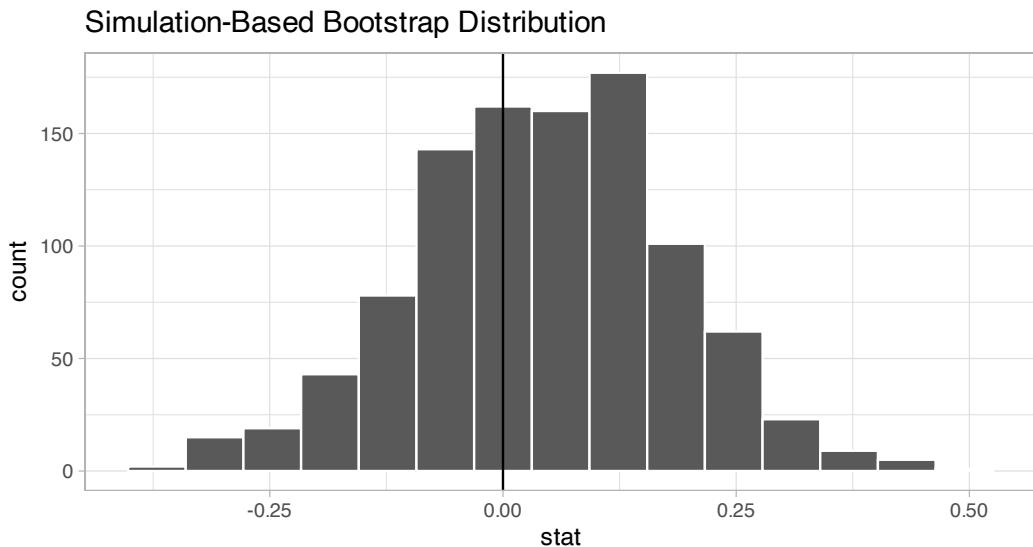


FIGURE 8.31: Bootstrap distribution.

First, let's compute the 95% confidence interval for $p_{seed} - p_{control}$ using the percentile method, in other words, by identifying the 2.5th and 97.5th percentiles which include the middle 95% of values. Recall that this method does not require the bootstrap distribution to be normally shaped.

```
bootstrap_distribution_yawning %>%
  get_confidence_interval(type = "percentile", level = 0.95)
```

```
# A tibble: 1 x 2
  lower_ci upper_ci
  <dbl>    <dbl>
1 -0.238276 0.302464
```

Second, since the bootstrap distribution is roughly bell-shaped, we can construct a confidence interval using the standard error method as well. Recall that to construct a confidence interval using the standard error method, we need to specify the center of the interval using the `point_estimate` argument. In our case, we need to set it to be the difference in sample proportions of 4.4% that the *Mythbusters* observed.

We can also use the `infer` workflow to compute this value by excluding the `generate()` 1000 bootstrap replicates step. In other words, do not generate replicates, but rather use only the original sample data. We can achieve this by commenting out the `generate()` line, telling R to ignore it:

```
obs_diff_in_props <- mythbusters_yawn %>%
  specify(formula = yawn ~ group, success = "yes") %>%
  # generate(reps = 1000, type = "bootstrap") %>%
  calculate(stat = "diff in props", order = c("seed", "control"))
obs_diff_in_props
```

```
Response: yawn (factor)
Explanatory: group (factor)
# A tibble: 1 × 1
  stat
  <dbl>
1 0.0441176
```

We thus plug this value in as the `point_estimate` argument.

```
myth_ci_se <- bootstrap_distribution_yawning %>%
  get_confidence_interval(type = "se", point_estimate = obs_diff_in_props)
```

Using `level = 0.95` to compute confidence interval.

```
myth_ci_se

# A tibble: 1 × 2
  lower_ci upper_ci
  <dbl>    <dbl>
1 -0.227291 0.315526
```

Let's visualize both confidence intervals in Figure 8.32, with the percentile-method interval marked with black lines and the standard-error-method marked with grey lines. Observe that they are both similar to each other.

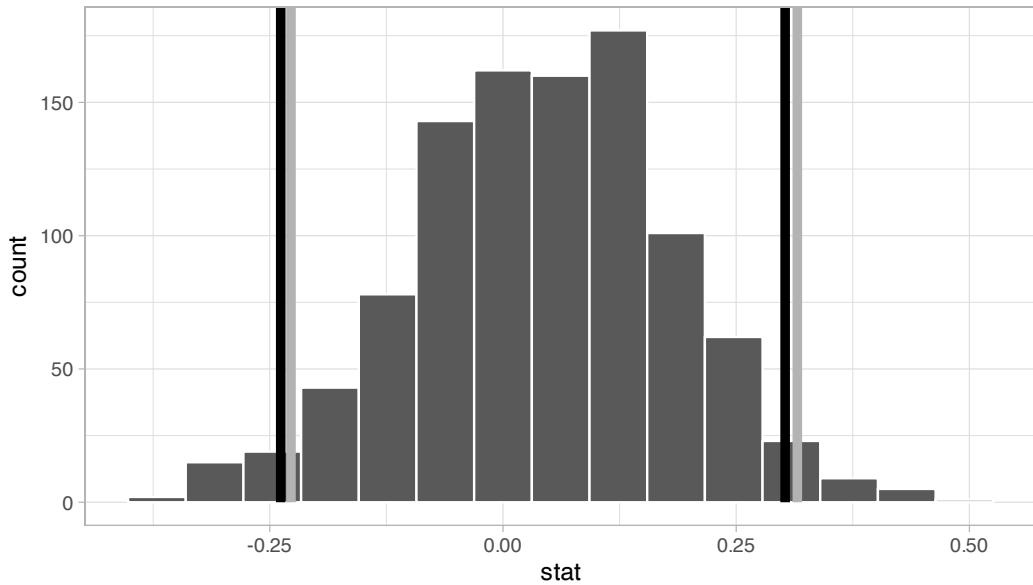


FIGURE 8.32: Two 95% confidence intervals: percentile method (black) and standard error method (grey).

8.6.4 Interpreting the confidence interval

Given that both confidence intervals are quite similar, let's focus our interpretation to only the percentile-method confidence interval of (-0.238, 0.302). Recall from Sub-section 8.5.2 that the precise statistical interpretation of a 95% confidence interval is: if this construction procedure is repeated 100 times, then we expect about 95 of the confidence intervals to capture the true value of $p_{seed} - p_{control}$. In other words, if we gathered 100 samples of $n = 50$ participants from a similar pool of people and constructed 100 confidence intervals each based on each of the 100 samples, about 95 of them will contain the true value of $p_{seed} - p_{control}$ while about five won't. Given that this is a little long winded, we use the shorthand interpretation: we're 95% “confident” that the true difference in proportions $p_{seed} - p_{control}$ is between (-0.238, 0.302).

There is one value of particular interest that this 95% confidence interval contains: zero. If $p_{seed} - p_{control}$ were equal to 0, then there would be no difference in proportion yawning between the two groups. This would suggest that there is no associated effect of being exposed to a yawning recruiter on whether you yawn yourself.

In our case, since the 95% confidence interval includes 0, we cannot conclusively say if either proportion is larger. Of our 1000 bootstrap resamples with replacement, sometimes \hat{p}_{seed} was higher and thus those exposed to yawning yawned themselves more often. At other times, the reverse happened.

Say, on the other hand, the 95% confidence interval was entirely above zero. This would suggest that $p_{seed} - p_{control} > 0$, or, in other words $p_{seed} > p_{control}$, and thus we'd have evidence suggesting those exposed to yawning do yawn more often.

8.7 Conclusion

8.7.1 Comparing bootstrap and sampling distributions

Let's talk more about the relationship between *sampling distributions* and *bootstrap distributions*.

Recall back in Subsection 7.2, we took 1000 virtual samples from the `bowl` using a virtual shovel, computed 1000 values of the sample proportion red \hat{p} , then visualized their distribution in a histogram. Recall that this distribution is called the *sampling distribution of \hat{p}* . Furthermore, the standard deviation of the sampling distribution has a special name: the *standard error*.

We also mentioned that this sampling activity does not reflect how sampling is done in real life. Rather, it was an *idealized version* of sampling so that we could study the effects of sampling variation on estimates, like the proportion of the shovel's balls that are red. In real life, however, one would take a single sample that's as large as possible, much like in the Obama poll we saw in Section 7.4. But how can we get a sense of the effect of sampling variation on estimates if we only have one sample and thus only one estimate? Don't we need many samples and hence many estimates?

The workaround to having a *single* sample was to perform *bootstrap resampling with replacement* from the single sample. We did this in the resampling activity in Section 8.1 where we focused on the mean year of minting of pennies. We used pieces of paper representing the original sample of 50 pennies from the bank and resampled them with replacement from a hat. We had 35 of our friends perform this activity and visualized the resulting 35 sample means \bar{x} in a histogram in Figure 8.11.

This distribution was called the *bootstrap distribution* of \bar{x} . We stated at the time that the bootstrap distribution is an *approximation* to the sampling distribution of \bar{x} in the sense that both distributions will have a similar shape and similar spread. Thus the *standard error* of the bootstrap distribution can be used as an approximation to the *standard error* of the sampling distribution.

Let's show you that this is the case by now comparing these two types of distributions. Specifically, we'll compare

1. the sampling distribution of \hat{p} based on 1000 virtual samples from the `bowl` from Subsection 7.2 to

2. the bootstrap distribution of \hat{p} based on 1000 virtual resamples with replacement from Ilyas and Yohan's single sample `bowl_sample_1` from Subsection 8.5.1.

Sampling distribution

Here is the code you saw in Subsection 7.2 to construct the sampling distribution of \hat{p} shown again in Figure 8.33, with some changes to incorporate the statistical terminology relating to sampling from Subsection 7.3.1.

```
# Take 1000 virtual samples of size 50 from the bowl:
virtual_samples <- bowl %>%
  rep_sample_n(size = 50, reps = 1000)
# Compute the sampling distribution of 1000 values of p-hat
sampling_distribution <- virtual_samples %>%
  group_by(replicate) %>%
  summarize(red = sum(color == "red")) %>%
  mutate(prop_red = red / 50)
# Visualize sampling distribution of p-hat
ggplot(sampling_distribution, aes(x = prop_red)) +
  geom_histogram(binwidth = 0.05, boundary = 0.4, color = "white") +
  labs(x = "Proportion of 50 balls that were red",
       title = "Sampling distribution")
```

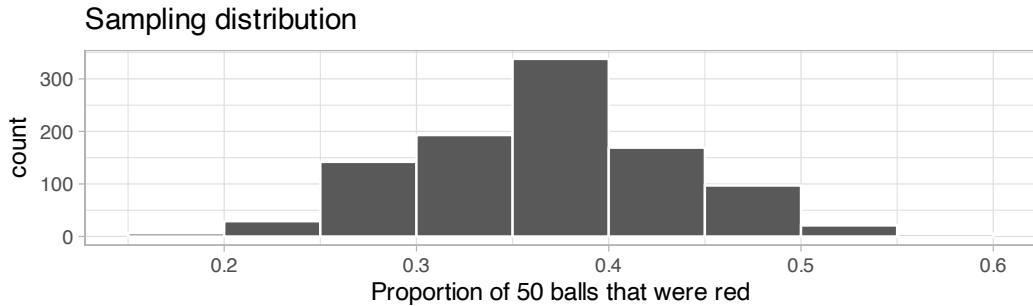


FIGURE 8.33: Previously seen sampling distribution of sample proportion red for $n = 1000$.

An important thing to keep in mind is the default value for `replace` is `FALSE` when using `rep_sample_n()`. This is because when sampling 50 balls with a shovel, we are extracting 50 balls one-by-one *without* replacing them. This is in contrast to bootstrap resampling *with* replacement, where we resample a ball and put it back, and repeat this process 50 times.

Let's quantify the variability in this sampling distribution by calculating the standard deviation of the `prop_red` variable representing 1000 values of the sample proportion \hat{p} . Remember that the standard deviation of the sampling distribution is the *standard error*, frequently denoted as `se`.

```
sampling_distribution %>% summarize(se = sd(prop_red))
```

```
# A tibble: 1 x 1
  se
  <dbl>
1 0.0673987
```

Bootstrap distribution

Here is the code you previously saw in Subsection 8.5.1 to construct the bootstrap distribution of \hat{p} based on Ilyas and Yohan's original sample of 50 balls saved in `bowl_sample_1`.

```
bootstrap_distribution <- bowl_sample_1 %>%
  specify(response = color, success = "red") %>%
  generate(reps = 1000, type = "bootstrap") %>%
  calculate(stat = "prop")
```

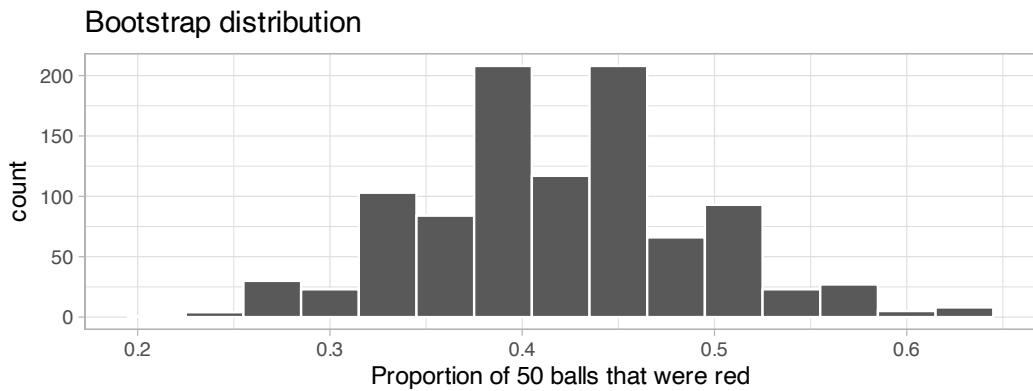


FIGURE 8.34: Bootstrap distribution of proportion red for $n = 1000$.

```
bootstrap_distribution %>% summarize(se = sd(stat))
```

```
# A tibble: 1 × 1
  se
  <dbl>
1 0.0712212
```

Comparison

Now that we have computed both the sampling distribution and the bootstrap distributions, let's compare them side-by-side in Figure 8.35. We'll make both histograms have matching scales on the x- and y-axes to make them more comparable. Furthermore, we'll add:

1. To the sampling distribution on the top: a solid line denoting the proportion of the bowl's balls that are red $p = 0.375$.
2. To the bootstrap distribution on the bottom: a dashed line at the sample proportion $\hat{p} = 21/50 = 0.42 = 42\%$ that Ilyas and Yohan observed.

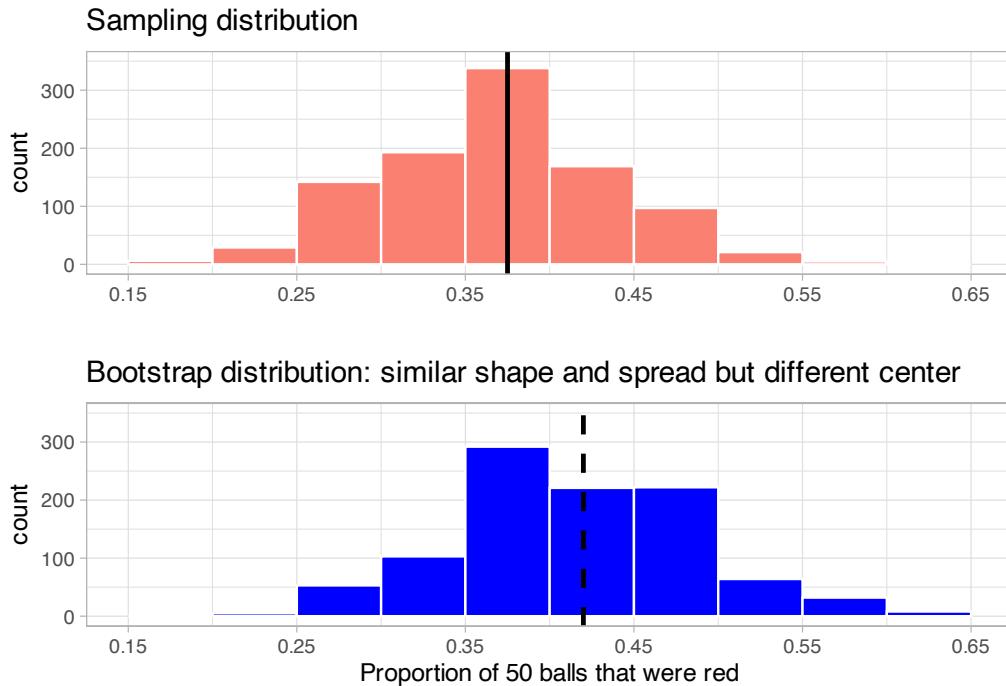


FIGURE 8.35: Comparing the sampling and bootstrap distributions of \hat{p} .

There is a lot going on in Figure 8.35, so let's break down all the comparisons slowly. First, observe how the sampling distribution on top is centered at $p = 0.375$. This is because the sampling is done at random and in an unbiased fashion. So the estimates \hat{p} are centered at the true value of p .

However, this is not the case with the following bootstrap distribution. The bootstrap distribution is centered at 0.42, which is the proportion red of Ilyas and Yohan's 50 sampled balls. This is because we are resampling from the same sample over and over again. Since the bootstrap distribution is centered at the original sample's proportion, it doesn't necessarily provide a better estimate of $p = 0.375$. This leads us to our first lesson about bootstrapping:

The bootstrap distribution will likely not have the same center as the sampling distribution. In other words, bootstrapping cannot improve the quality of an estimate.

Second, let's now compare the spread of the two distributions: they are somewhat similar. In the previous code, we computed the standard deviations of both distributions as well. Recall that such standard deviations have a special name: *standard errors*. Let's compare them in Table 8.5.

TABLE 8.5: Comparing standard errors

Distribution type	Standard error
Sampling distribution	0.067
Bootstrap distribution	0.071

Notice that the bootstrap distribution's standard error is a rather good *approximation* to the sampling distribution's standard error. This leads us to our second lesson about bootstrapping:

Even if the bootstrap distribution might not have the same center as the sampling distribution, it will likely have very similar shape and spread. In other words, bootstrapping will give you a good estimate of the *standard error*.

Thus, using the fact that the bootstrap distribution and sampling distributions have similar spreads, we can build confidence intervals using bootstrapping as we've done all throughout this chapter!

8.7.2 Theory-based confidence intervals

So far in this chapter, we've constructed confidence intervals using two methods: the percentile method and the standard error method. Recall also from Subsection 8.3.2 that we can only use the standard-error method if the bootstrap distribution is bell-shaped (i.e., normally distributed).

In a similar vein, if the sampling distribution is normally shaped, there is another method for constructing confidence intervals that does not involve using your computer. You can use a *theory-based method* involving mathematical formulas!

The formula uses the rule of thumb we saw in Appendix A.2 that 95% of values in a normal distribution are within ± 1.96 standard deviations of the mean. In the case of sampling and bootstrap distributions, remember that the standard deviation has a special name: the *standard error*. Recall further in Subsection 7.6.2 you saw that there is a theory-based formula to approximate the standard error for sample proportions \hat{p} :

$$\text{SE}_{\hat{p}} \approx \sqrt{\frac{\hat{p}(1 - \hat{p})}{n}}$$

If you've forgotten this fact and what it says about the relationship between “precision” of your estimates and your sample size n , we highly recommend you re-read Subsection 7.6.2.

Recall from `bowl_sample_1` that Yohan and Ilyas sampled $n = 50$ balls and observed a sample proportion \hat{p} of $21/50 = 0.42$. An approximation of the standard error of \hat{p} based on Yohan and Ilyas' sample is thus:

$$\text{SE}_{\hat{p}} \approx \sqrt{\frac{0.42(1 - 0.42)}{50}} = \sqrt{0.004872} = 0.0698 \approx 0.070$$

Let's compare this theory-based standard error to the standard error of the sampling and bootstrap distributions you computed previously in Subsection 8.7.1 in Table 8.6. Notice how they are all similar!

TABLE 8.6: Comparing standard errors

Distribution type	Standard error
Sampling distribution	0.067
Bootstrap distribution	0.071
Formula approximation	0.070

Using the theory-based standard error, let's present a theory-based method for constructing 95% confidence intervals that does not involve using a computer, but rather mathematical formulas. Note that this theory-based method only holds if the sampling distribution is normally shaped, so that we can use the 95% rule of thumb about normal distributions discussed in Appendix A.2.

1. Collect a single representative sample of size n that's as large as possible.
2. Compute the *point estimate*: the *sample proportion* \hat{p} . Think of this as the center of your “net.”
3. Compute the approximation to the standard error

$$\text{SE}_{\hat{p}} \approx \sqrt{\frac{\hat{p}(1 - \hat{p})}{n}}$$

4. Compute a quantity known as the *margin of error* (more on this later after we list the five steps):

$$\text{MoE}_{\hat{p}} = 1.96 \cdot \text{SE}_{\hat{p}} = 1.96 \cdot \sqrt{\frac{\hat{p}(1 - \hat{p})}{n}}$$

5. Compute both endpoints of the confidence interval.

- The lower end-point. Think of this as the left end-point of the net:

$$\hat{p} - \text{MoE}_{\hat{p}} = \hat{p} - 1.96 \cdot \text{SE}_{\hat{p}} = \hat{p} - 1.96 \cdot \sqrt{\frac{\hat{p}(1 - \hat{p})}{n}}$$

- The upper endpoint. Think of this as the right end-point of the net:

$$\hat{p} + \text{MoE}_{\hat{p}} = \hat{p} + 1.96 \cdot \text{SE}_{\hat{p}} = \hat{p} + 1.96 \cdot \sqrt{\frac{\hat{p}(1 - \hat{p})}{n}}$$

- Alternatively, you can succinctly summarize a 95% confidence interval for p using the \pm symbol:

$$\hat{p} \pm \text{MoE}_{\hat{p}} = \hat{p} \pm (1.96 \cdot \text{SE}_{\hat{p}}) = \hat{p} \pm \left(1.96 \cdot \sqrt{\frac{\hat{p}(1 - \hat{p})}{n}} \right)$$

So going back to Yohan and Ilyas' sample of $n = 50$ balls that had 21 red balls, the 95% confidence interval for p is

$$\begin{aligned} 0.41 \pm 1.96 \cdot 0.0698 &= 0.41 \pm 0.137 \\ &= (0.41 - 0.137, 0.41 + 0.137) \\ &= (0.273, 0.547). \end{aligned}$$

Yohan and Ilyas are 95% “confident” that the true proportion red of the bowl’s balls is between 28.3% and 55.7%. Given that the true population proportion p was 0.375, in this case they successfully captured the fish.

In Step 4, we defined a statistical quantity known as the *margin of error*. You can think of this quantity as how much the net extends to the left and to the right of the center of our net. The 1.96 multiplier is rooted in the 95% rule of thumb we introduced earlier and the fact that we want the confidence level to be 95%. The value of the margin of error entirely determines the width of the confidence interval.

Recall from Subsection 8.5.3 that confidence interval widths are determined by an interplay of the confidence level, the sample size n , and the standard error.

Let's revisit the poll of President Obama's approval rating among young Americans aged 18-29 which we introduced in Section 7.4. Pollsters found that based on a representative sample of $n = 2089$ young Americans, $\hat{p} = 0.41 = 41\%$ supported President Obama.

If you look towards the end of the article, it also states: "The poll's margin of error was plus or minus 2.1 percentage points." This is precisely the MoE:

$$\begin{aligned}\text{MoE} &= 1.96 \cdot \text{SE} = 1.96 \cdot \sqrt{\frac{\hat{p}(1 - \hat{p})}{n}} = 1.96 \cdot \sqrt{\frac{0.41(1 - 0.41)}{2089}} \\ &= 1.96 \cdot 0.0108 = 0.021 = 2.1\%\end{aligned}$$

Their poll results are based on a confidence level of 95% and the resulting 95% confidence interval for the proportion of all young Americans who support Obama is:

$$\hat{p} \pm \text{MoE} = 0.41 \pm 0.021 = (0.389, 0.431) = (38.9\%, 43.1\%).$$

Confidence intervals based on 33 tactile samples

Let's revisit our 33 friends' samples from the bowl from Subsection 7.1.3. We'll use their 33 samples to construct 33 theory-based 95% confidence intervals for p . Recall this data was saved in the `tactile_prop_red` data frame included in the `moderndive` package:

1. `rename()` the variable `prop_red` to `p_hat`, the statistical name of the sample proportion \hat{p} .
2. `mutate()` a new variable `n` making explicit the sample size of 50.
3. `mutate()` other new variables computing:
 - The standard error `SE` for \hat{p} using the previous formula.
 - The margin of error `MoE` by multiplying the `SE` by 1.96
 - The left endpoint of the confidence interval `lower_ci`
 - The right endpoint of the confidence interval `upper_ci`

```
conf_ints <- tactile_prop_red %>%
  rename(p_hat = prop_red) %>%
  mutate(
    n = 50,
    SE = sqrt(p_hat * (1 - p_hat) / n),
    MoE = 1.96 * SE,
    lower_ci = p_hat - MoE,
```

```
    upper_ci = p_hat + MoE
)
```

In Figure 8.36, let's plot the 33 confidence intervals for p saved in `conf_ints` along with a vertical line at $p = 0.375$ indicating the true proportion of the bowl's balls that are red. Furthermore, let's mark the sample proportions \hat{p} with dots since they represent the centers of these confidence intervals.

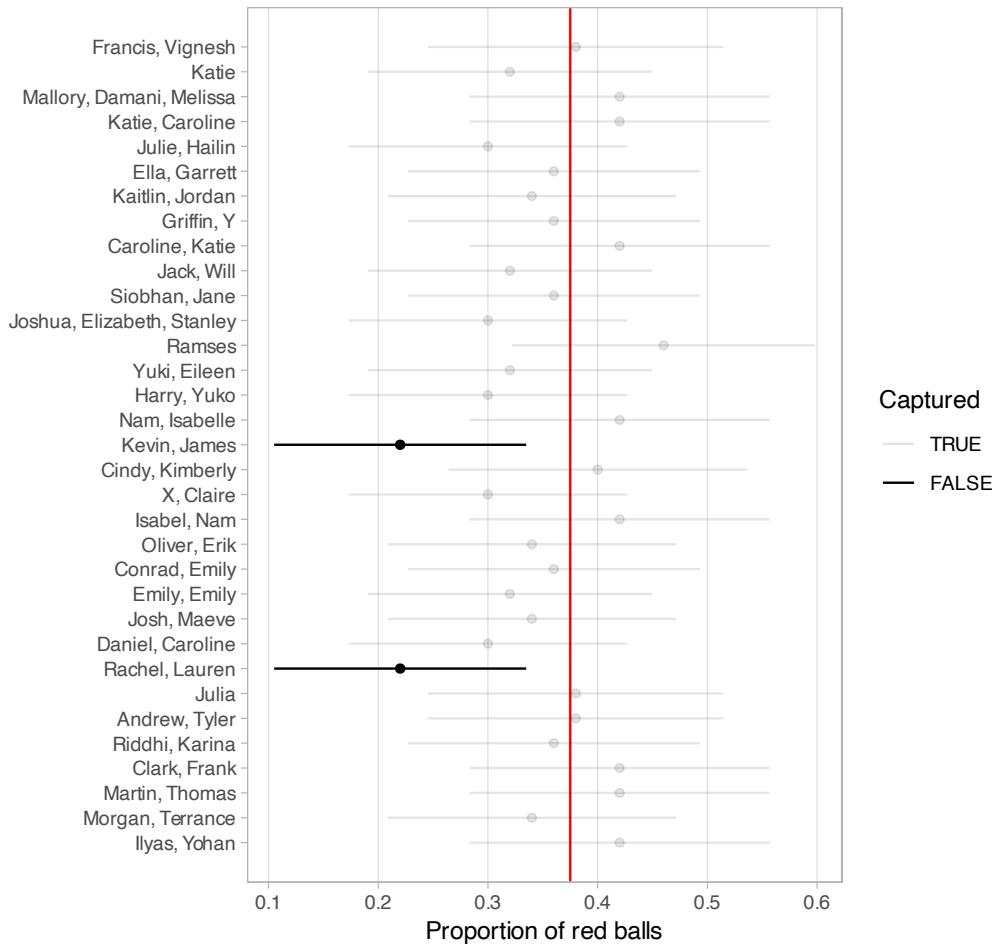


FIGURE 8.36: 33 confidence intervals at the 95% level based on 33 tactile samples of size $n = 50$.

Observe that 31 of the 33 confidence intervals “captured” the true value of p , for a success rate of $31 / 33 = 93.94\%$. While this is not quite 95%, recall that we *expect* about 95% of such confidence intervals to capture p . The actual observed success rate will vary slightly.

Theory-based methods like this have largely been used in the past because we didn't have the computing power to perform simulation-based methods such as bootstrapping. They are still commonly used, however, and if the sampling distribution is normally distributed, we have access to an alternative method for constructing confidence intervals as well as performing hypothesis tests as we will see in Chapter 9.

The kind of computer-based statistical inference we've seen so far has a particular name in the field of statistics: *simulation-based inference*. This is because we are performing statistical inference using computer simulations. In our opinion, two large benefits of simulation-based methods over theory-based methods are that (1) they are easier for people new to statistical inference to understand and (2) they also work in situations where theory-based methods and mathematical formulas don't exist.

8.7.3 Additional resources

Solutions to all *Learning checks* can be found online in Appendix D⁷.

An R script file of all R code used in this chapter is available at <https://www.moderndive.com/scripts/08-confidence-intervals.R>.

If you want more examples of the `infer` workflow to construct confidence intervals, we suggest you check out the `infer` package homepage, in particular, a series of example analyses available at <https://infer.netlify.app/articles/>.

8.7.4 What's to come?

Now that we've equipped ourselves with confidence intervals, in Chapter 9 we'll cover the other common tool for statistical inference: hypothesis testing. Just like confidence intervals, hypothesis tests are used to infer about a population using a sample. However, we'll see that the framework for making such inferences is slightly different.

⁷<https://moderndive.com/D-appendixD.html>

9

Hypothesis Testing

Now that we've studied confidence intervals in Chapter 8, let's study another commonly used method for statistical inference: hypothesis testing. Hypothesis tests allow us to take a sample of data from a population and infer about the plausibility of competing hypotheses. For example, in the upcoming music popularity activity by genre in Section 9.1, you'll study data collected from Spotify to investigate whether metal music is more popular than deep-house music.

The good news is we've already covered many of the necessary concepts to understand hypothesis testing in Chapters 7 and 8. We will expand further on these ideas here and also provide a general framework for understanding hypothesis tests. By understanding this general framework, you'll be able to adapt it to many different scenarios.

The same can be said for confidence intervals. There was one general framework that applies to confidence intervals, and the `infer` package was designed around this framework. While the specifics may change slightly for different types of confidence intervals, the general framework stays the same.

We believe that this approach is much better for long-term learning than focusing on specific details for specific confidence intervals using theory-based approaches. As you'll now see, we prefer this general framework for hypothesis tests as well.

Needed packages

Let's load all the packages needed for this chapter (this assumes you've already installed them). Recall from our discussion in Section 4.4 that loading the `tidyverse` package by running `library(tidyverse)` loads the following commonly used data science packages all at once:

- `ggplot2` for data visualization
- `dplyr` for data wrangling
- `tidy` for converting data to “tidy” format
- `readr` for importing spreadsheet data into R
- As well as the more advanced `purrr`, `tibble`, `stringr`, and `forcats` packages

If needed, read Section 1.3 for information on how to install and load R packages.

```
library(tidyverse)
library(infer)
library(moderndive)
library(nycflights23)
library(ggplot2movies)
```

9.1 Music popularity activity

Let's start with an activity studying the effect of music genre on Spotify song popularity.

9.1.1 Is metal music more popular than deep house music?

Imagine you're a music analyst for Spotify, and you're curious about whether fans of metal or deep house are more passionate about their favorite genres. You want to determine if there's a significant difference in the popularity of these two genres. Popularity, in this case, is measured by Spotify, say, as the average number of streams and recent user interactions on tracks classified under each genre. (Note that Spotify doesn't actually disclose how this metric is calculated, so we have to take our best guess.) This question sets the stage for our exploration into hypothesis testing.

Metal music, characterized by its aggressive sounds, powerful vocals, and complex instrumentals, has cultivated a loyal fanbase that often prides itself on its deep appreciation for the genre's intensity and technical skill. On the other hand, deep house music, with its smooth, soulful rhythms and steady beats, attracts listeners who enjoy the genre's calming and immersive vibe, often associated with late-night clubs and chill-out sessions.

By comparing the popularity metrics between these two genres, we can determine if one truly resonates more with listeners on Spotify. This exploration not only deepens our understanding of musical preferences but also serves as a practical introduction to the principles of hypothesis testing.

To begin the analysis, 2000 were selected at random from Spotify's song archive. We'll use "song" and "track" interchangeably going forward. There were 1000 metal tracks and 1000 deep house tracks selected.

The `moderndive` package contains the data on the songs by genre in the `spotify_by_genre` data frame. There are six genres selected in that data (`country`, `deep-house`, `dubstep`, `hip-hop`, `metal`, and `rock`). You'll have the opportunity to explore relationships with the other genres and popularity in the Learning Checks. Let's explore this data by focusing on just `metal` and `deep-house` by looking at twelve randomly selected rows and

our columns of interest. Note here that we also group our selection so that three songs of each of the four possible groupings of `track_genre` and `popular_or_not` are selected.

```
spotify_metal_deephouse <- spotify_by_genre |>
  filter(track_genre %in% c("metal", "deep-house")) |>
  select(track_id, track_genre, artists, track_name, popularity, popular_or_not)
spotify_metal_deephouse |>
  group_by(track_genre, popular_or_not) |>
  sample_n(size = 3) |>
  arrange(track_id)
```

	track_id	track_genre	artists	track_name	popularity	popular_or_not
1	14VmsVz9jMLnTv~	deep-house	LYOD;T~	On My Way	51	popular
2	1uK6lSRKwD9T5F~	deep-house	Sunmoon	Just the ~	52	popular
3	2Eo5TuDu2fPROZ~	deep-house	Tensna~	Latching ~	51	popular
4	2MvIMgtWyK880i~	metal	Slipkn~	Psychosoc~	66	popular
5	2RrgY5XzFmlLtq~	deep-house	BCX	Miracle I~	41	not popular
6	31IfnfvSlw299v~	metal	blesst~	I Wouldn'~	0	not popular
7	3AJmjefnCKNq2V~	deep-house	Junge ~	I'm The O~	49	not popular
8	3KXW97hQNrLi0b~	metal	Poison	Every Ros~	0	not popular
9	3eMOYZRLqXMBw9~	metal	Armore~	S.O.S.	54	popular
10	3txlbadheKtYDs~	deep-house	James ~	Good Luck~	47	not popular
11	5toE2GI4iMIpel~	metal	Hollyw~	Riot	26	not popular
12	7HjNOz8Y7H7uSy~	metal	Breaki~	Ashes of ~	61	popular

The variable `track_id` acts as an identification variable for all 2000 rows, the `track_genre` variable indicates what genre the song is classified under, the `artists` and `track_name` columns provide additional information about the track by providing the artist and the name of the song, `popularity` is the metric mentioned earlier given by Spotify, and `popular_or_not` is a categorical representation of the `popularity` column with any value of 50 (the 75th percentile of `popularity`) referring to `popular` and anything at or below 50 as `not_popular`. The decision made by the authors to call a song “popular” if it is above the 75th percentile (3rd quartile) of `popularity` is arbitrary and could be changed to any other value.)

Let’s perform an exploratory data analysis of the relationship between the two categorical variables `track_genre` and `popular_or_not`. Recall that we saw in Subsection 2.8.3 that one way we can visualize such a relationship is by using a stacked barplot.

```
ggplot(spotify_metal_deephouse, aes(x = track_genre, fill = popular_or_not)) +
  geom_bar() +
  labs(x = "Genre of track")
```

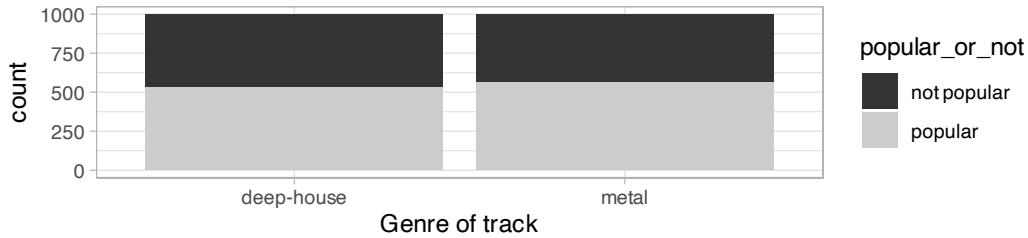


FIGURE 9.1: Barplot relating genre to popularity.

Observe in Figure 9.1 that, in this sample, metal songs are only slightly more popular than deep house songs by looking at the height of the popular bars. Let's quantify these popularity rates by computing the proportion of songs classified as `popular` for each of the two genres using the `dplyr` package for data wrangling. Note the use of the `tally()` function here which is a shortcut for `summarize(n = n())` to get counts.

```
spotify_metal_deephouse |>
  group_by(track_genre, popular_or_not) |>
  tally() # Same as summarize(n = n())
```

```
# A tibble: 4 x 3
# Groups:   track_genre [2]
  track_genre popular_or_not     n
  <chr>       <chr>        <int>
1 deep-house  not popular    471
2 deep-house  popular       529
3 metal        not popular    437
4 metal        popular        563
```

So of the 1000 metal songs, 563 were popular, for a proportion of $563/1000 = 0.563 = 56.3\%$. On the other hand, of the 1000 deep house songs, 529 were popular, for a proportion of $529/1000 = 0.529 = 52.9\%$. Comparing these two rates of promotion, it appears that metal songs were popular at a rate $0.563 - 0.529 = 0.034 = 3.4\%$ higher than deep house songs. This is suggestive of an advantage for metal songs in terms of popularity.

The question is, however, does this provide *conclusive* evidence that there is greater popularity for metal songs compared to deep house songs? Could a difference in

popularity rates of 3.4% still occur by chance, even in a hypothetical world where no difference in popularity existed between the two genres? In other words, what is the role of *sampling variation* in this hypothesized world? To answer this question, we'll again rely on a computer to run *simulations*.

9.1.2 Shuffling once

First, try to imagine a hypothetical universe where difference in the popularity of metal and deep house songs existed. In such a hypothetical universe, the genre of a song would have no bearing on their chances of popularity. Bringing things back to our `spotify_metal_deephause` data frame, the `popular_or_not` variable would thus be an irrelevant label. If these `popular_or_not` labels were irrelevant, then we could randomly reassigned them by “shuffling” them to no consequence!

To illustrate this idea, let's narrow our focus to 52 chosen songs of the 2000 that you saw earlier. The `track_genre` column shows what the original genre of the song was. Note that to keep this smaller data set of 52 rows to be a representative sample of the 2000 rows, we have sampled such that the popularity rate for each of `metal` and `deep-house` is close to the original rates of 0.563 and 0.529, respectively, prior to shuffling. This data is available in the `spotify_52_original` data frame in the `moderndive` package.

```
spotify_52_original
```

```
# A tibble: 52 x 6
  track_id      track_genre artists track_name popularity popular_or_not
  <chr>        <chr>     <chr>   <chr>      <dbl> <chr>
1 3fvxmytTns1Ap~ deep-house Jess B~ Temptatio~       63 popular
2 6Nd6ntkzr4t8o1~ metal      Whites~ Here I Go~       69 popular
3 7MIKwg3dDCWhxM~ metal      Blind ~ No Rain       1 not popular
4 1fQaoh3imrMunW~ metal      Avenge~ Shepherd ~     70 popular
5 200vM6F7VMXf66~ deep-house Nora V~ I Wanna D~     56 popular
6 7hjN0z8Y7H7uSy~ metal      Breaki~ Ashes of ~     61 popular
7 577MNC3o3L01NI~ metal      Bon Jo~ Thank You~     67 popular
8 5KtlHyEw7BMr1u~ deep-house Starle~ Arms Arou~     55 popular
9 7bXWSg5gd0Em0e~ deep-house The Hi~ I Wonder      43 not popular
10 4nzIpIeHWiuFQn~ metal     Defton~ Ohms          0 not popular
# i 42 more rows
```

In our hypothesized universe of no difference in genre popularity, popularity is irrelevant and thus it is of no consequence to randomly “shuffle” the values of `popular_or_not`. The `popular_or_not` column in the `spotify_52_shuffled` data frame in the `moderndive` package shows one such possible random shuffling.

```
spotify_52_shuffled
```

```
# A tibble: 52 x 6
  track_id      track_genre artists track_name popularity popular_or_not
  <chr>        <chr>      <chr>   <chr>       <dbl> <chr>
1 3fvxsxmytTns1Ap~ deep-house Jess B~ Temptatio~       63 popular
2 6Nd6ntkzr4t8o1~ metal     Whites~ Here I Go~       69 not popular
3 7MIKwg3dDCWhxM~ metal     Blind ~ No Rain       1 popular
4 1fQaoh3imrMunW~ metal     Avenge~ Shepherd ~     70 not popular
5 200vM6F7VMXf66~ deep-house Nora V~ I Wanna D~     56 popular
6 7HjN0z8Y7H7uSy~ metal     Breaki~ Ashes of ~     61 not popular
7 577MNC3o3L0lNI~ metal     Bon Jo~ Thank You~     67 not popular
8 5Kt1HyEw7BMr1u~ deep-house Starle~ Arms Arou~     55 not popular
9 7bXWSg5gd0Em0e~ deep-house The Hi~ I Wonder       43 not popular
10 4nzIpIeHWiuFQn~ metal    Defton~ Ohms           0 not popular
# i 42 more rows
```

Observe in the `popular_or_not` column how the `popular` and `not popular` results are now listed in a different order. Some of the original `popular` now are `not popular`, some of the `not popular` are `popular`, and others are the same as the original.

Again, such random shuffling of the `popular_or_not` label only makes sense in our hypothesized universe of no difference in popularity between genres. Is there a tactile way for us to understand what is going on with this shuffling? One way would be by using standard deck of 52 playing cards, which we display in Figure 9.2.

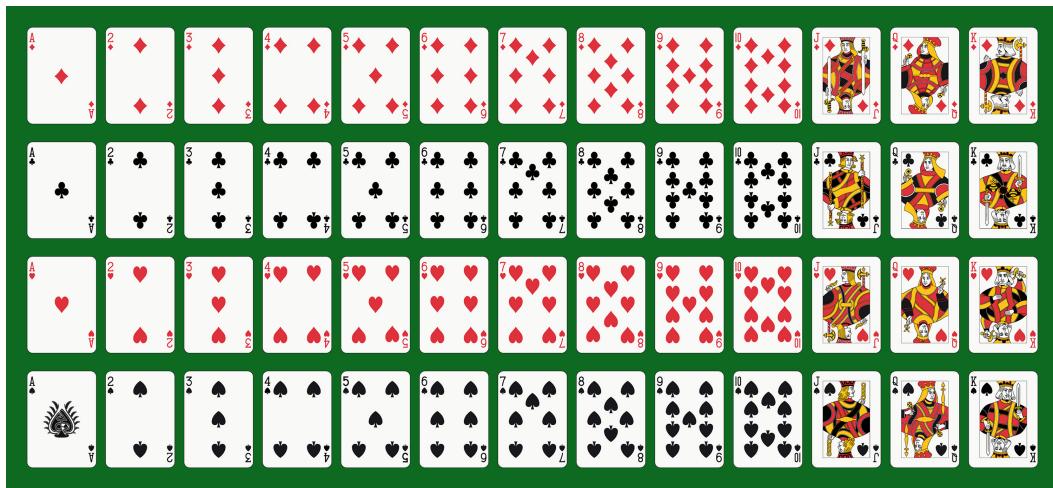


FIGURE 9.2: Standard deck of 52 playing cards.

Since we started with equal sample sizes of 1000 songs for each genre, we can think about splitting the deck in half to have 26 cards in two piles (one for `metal` and

another for deep-house). After shuffling these 52 cards as seen in Figure 9.3, we split the deck equally into the two piles of 26 cards each. Then, we can flip the cards over one-by-one, assigning “popular” for each red card and “not popular” for each black card keeping a tally of how many of each genre are popular.



FIGURE 9.3: Shuffling a deck of cards.

Let’s repeat the same exploratory data analysis we did for the original `spotify_metal_deephouse` data on our `spotify_52_original` and `spotify_52_shuffled` data frames. Let’s create a barplot visualizing the relationship between `track_genre` and the new shuffled `popular_or_not` variable, and compare this to the original un-shuffled version in Figure 9.4.

```
ggplot(spotify_52_shuffled,
       aes(x = track_genre, fill = popular_or_not)) +
  geom_bar() +
  labs(x = "Genre of track")  
  
'summarise()' has grouped output by 'track_genre'. You can override using  
the `groups` argument.  
'summarise()' has grouped output by 'track_genre'. You can override using  
the `groups` argument.
```

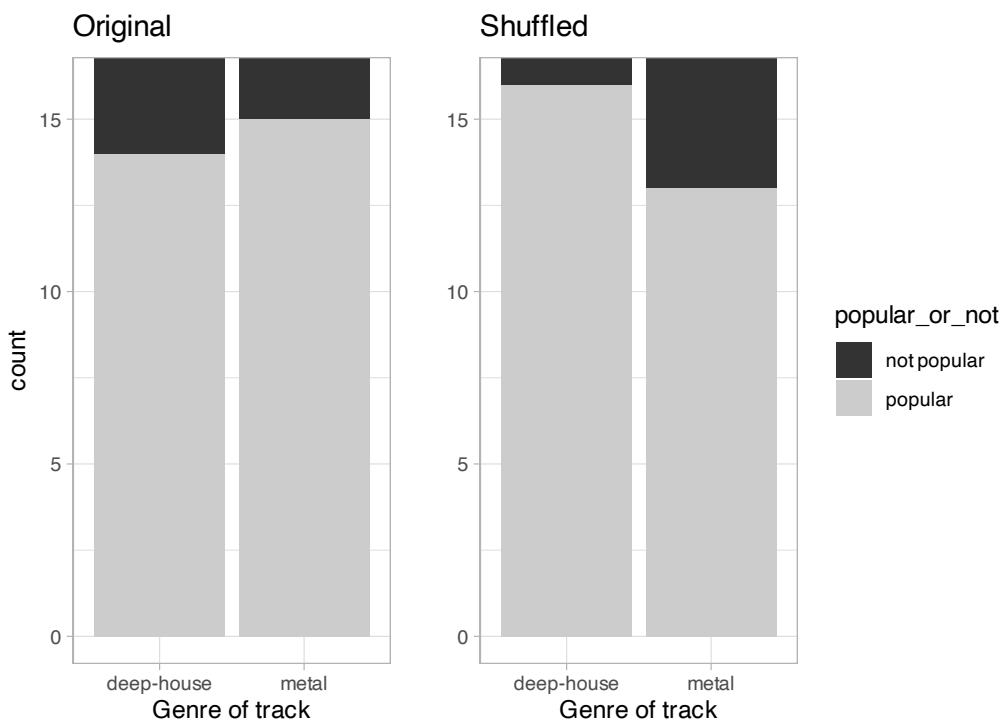


FIGURE 9.4: Barplots of relationship of genre with ‘popular or not’ (left) and shuffled ‘popular or not’ (right).

The difference in metal versus deep house popularity rates is now different. Compared to the original data in the left barplot, the new “shuffled” data in the right barplot has popularity rates that are actually in the opposite direction as they were originally. This is because the shuffling process has removed any relationship between genre and popularity.

Let’s also compute the proportion of tracks that are now “popular” in the `popular_or_not` column for each genre:

```
spotify_52_shuffled |>
  group_by(track_genre, popular_or_not) |>
  tally()
```

```
# A tibble: 4 x 3
# Groups:   track_genre [2]
  track_genre popular_or_not     n
  <chr>       <chr>        <int>
1 deep-house  not popular    10
2 deep-house  popular       16
```

3 metal	not popular	13
4 metal	popular	13

So in this one sample of a hypothetical universe of no difference in genre popularity, $13/26 = 0.5 = 50\%$ of metal songs were popular. On the other hand, $16/26 = 0.615 = 61.5\%$ of deep house songs were popular. Let's next compare these two values. It appears that metal tracks were popular at a rate that was $0.5 - 0.615 = -0.115 = -11.5\%$ different than deep house songs.

Observe how this difference in rates is not the same as the difference in rates of $0.034 = 3.4\%$ we originally observed. This is once again due to *sampling variation*. How can we better understand the effect of this sampling variation? By repeating this shuffling several times!

9.1.3 What did we just do?

What we just demonstrated in this activity is the statistical procedure known as *hypothesis testing* using a *permutation test*. The term “permutation” is the mathematical term for “shuffling”: taking a series of values and reordering them randomly, as you did with the playing cards. In fact, permutations are another form of *resampling*, like the bootstrap method you performed in Chapter 8. While the bootstrap method involves resampling *with* replacement, permutation methods involve resampling *without* replacement.

We needn't stay defined to this dataset of 52 rows only though. It was useful to see how we could use the deck of cards as a way to visualize tactily how the shuffling would assign values of popular or not popular to different songs. These same ideas can be applied to each of the 2000 tracks in our `spotify_metal_deephause` data. We can think for this data about an inference about an unknown difference of population proportions with the 2000 tracks being our sample. We denote this as $p_m - p_d$, where p_m is the population proportion of songs with metal names being popular and p_d is the equivalent for deep house songs. Recall that this is one of the scenarios for inference we've seen so far in Table 9.1.

TABLE 9.1: Scenarios of sampling for inference

Scenario	Population parameter	Notation	Point estimate	Symbol(s)
1	Population proportion	p	Sample proportion	\hat{p}
2	Population mean	μ	Sample mean	\bar{x} or $\hat{\mu}$
3	Difference in population proportions	$p_1 - p_2$	Difference in sample proportions	$\hat{p}_1 - \hat{p}_2$

So, based on our sample of $n_m = 1000$ metal tracks and $n_f = 1000$ deep house tracks, the *point estimate* for $p_m - p_d$ is the *difference in sample proportions* $\hat{p}_m - \hat{p}_f = 0.563 - 0.529 = 0.034 = 3.4\%$. This difference in favor of metal songs of 0.034 is greater than 0, suggesting metal songs are more popular than deep house songs.

However, the question we ask ourselves was “is this difference meaningfully greater than 0?” In other words, is that difference indicative of true popularity, or can we just attribute it to *sampling variation*? Hypothesis testing allows us to make such distinctions.

9.2 Understanding hypothesis tests

Much like the terminology, notation, and definitions relating to sampling you saw in Section 7.3, there are a lot of terminology, notation, and definitions related to hypothesis testing as well. Learning these may seem like a very daunting task at first. However, with practice, practice, and more practice, anyone can master them.

First, a **hypothesis** is a statement about the value of an unknown population parameter. In our genre popularity activity, our population parameter of interest is the difference in population proportions $p_m - p_d$. Hypothesis tests can involve any of the population parameters in Table 7.5 of the five inference scenarios we’ll cover in this book and also more advanced types we won’t cover here.

Second, a **hypothesis test** consists of a test between two competing hypotheses: (1) a **null hypothesis** H_0 (pronounced “H-naught”) versus (2) an **alternative hypothesis** H_A (also denoted H_1).

Generally the null hypothesis is a claim that there is “no effect” or “no difference of interest.” In many cases, the null hypothesis represents the status quo or a situation that nothing interesting is happening. Furthermore, generally the alternative hypothesis is the claim the experimenter or researcher wants to establish or find evidence to support. It is viewed as a “challenger” hypothesis to the null hypothesis H_0 . In our genre popularity activity, an appropriate hypothesis test would be:

$$\begin{aligned} H_0 &: \text{metal and deep house have the same popularity rate} \\ \text{vs } H_A &: \text{metal is popular at a higher rate than deep house} \end{aligned}$$

Note some of the choices we have made. First, we set the null hypothesis H_0 to be that there is no difference in popularity rate and the “challenger” alternative hypothesis H_A to be that there is a difference. While it would not be wrong in principle to reverse the two, it is a convention in statistical inference that the null hypothesis is set to reflect a “null” situation where “nothing is going on.” As we discussed earlier, in this case, H_0 corresponds to there being no difference in promotion rates. Furthermore, we set H_A to be that metal is popular at a *higher* rate, a subjective choice reflecting

a prior suspicion we have that this is the case. We call such alternative hypotheses *one-sided alternatives*. If someone else however does not share such suspicions and only wants to investigate that there is a difference, whether higher or lower, they would set what is known as a *two-sided alternative*.

We can re-express the formulation of our hypothesis test using the mathematical notation for our population parameter of interest, the difference in population proportions $p_m - p_d$:

$$\begin{aligned} H_0 : p_m - p_d &= 0 \\ \text{vs } H_A : p_m - p_d &> 0 \end{aligned}$$

Observe how the alternative hypothesis H_A is one-sided with $p_m - p_d > 0$. Had we opted for a two-sided alternative, we would have set $p_m - p_d \neq 0$. To keep things simple for now, we'll stick with the simpler one-sided alternative. We'll present an example of a two-sided alternative in Section 9.5.

Third, a **test statistic** is a *point estimate/sample statistic* formula used for hypothesis testing. Note that a sample statistic is merely a summary statistic based on a sample of observations. Recall we saw in Section 3.3 that a summary statistic takes in many values and returns only one. Here, the samples would be the $n_m = 1000$ metal songs and the $n_f = 1000$ deep house songs. Hence, the point estimate of interest is the difference in sample proportions $\hat{p}_m - \hat{p}_d$.

Fourth, the **observed test statistic** is the value of the test statistic that we observed in real life. In our case, we computed this value using the data saved in the `spotify_metal_deephouse` data frame. It was the observed difference of $\hat{p}_m - \hat{p}_d = 0.563 - 0.529 = 0.034 = 3.4\%$ in favor of metal songs.

Fifth, the **null distribution** is the sampling distribution of the test statistic *assuming the null hypothesis H_0 is true*. Ooof! That's a long one! Let's unpack it slowly. The key to understanding the null distribution is that the null hypothesis H_0 is *assumed* to be true. We're not saying that H_0 is true at this point, we're only assuming it to be true for hypothesis testing purposes. In our case, this corresponds to our hypothesized universe of no difference in popularity rates. Assuming the null hypothesis H_0 , also stated as "Under H_0 ," how does the test statistic vary due to sampling variation? In our case, how will the difference in sample proportions $\hat{p}_m - \hat{p}_f$ vary due to sampling under H_0 ? Recall from Subsection 7.3.2 that distributions displaying how point estimates vary due to sampling variation are called *sampling distributions*. The only additional thing to keep in mind about null distributions is that they are sampling distributions *assuming the null hypothesis H_0 is true*.

Sixth, the ***p*-value** is the probability of obtaining a test statistic just as extreme or more extreme than the observed test statistic *assuming the null hypothesis H_0 is true*. Double ooof! Let's unpack this slowly as well. You can think of the *p*-value as a quantification of "surprise": assuming H_0 is true, how surprised are we with what we observed? Or in our case, in our hypothesized universe of no difference in genre popularity, how surprised are we that we observed a difference in popularity rates

of 0.034 from our collected samples assuming H_0 is true? Very surprised? Somewhat surprised?

The p -value quantifies this probability, or what proportion had a more “extreme” result? Here, extreme is defined in terms of the alternative hypothesis H_A that metal applicants are promoted at a higher rate than deep house applicants. In other words, how often was the popularity of metal *even more* pronounced than $0.563 - 0.529 = 0.034 = 3.4\%$?

Seventh and lastly, in many hypothesis testing procedures, it is commonly recommended to set the **significance level** of the test beforehand. It is denoted by the Greek letter α (pronounced “alpha”). This value acts as a cutoff on the p -value, where if the p -value falls below α , we would “reject the null hypothesis H_0 .”

Alternatively, if the p -value does not fall below α , we would “fail to reject H_0 .” Note the latter statement is not quite the same as saying we “accept H_0 .” This distinction is rather subtle and not immediately obvious. So we’ll revisit it later in Section 9.4.

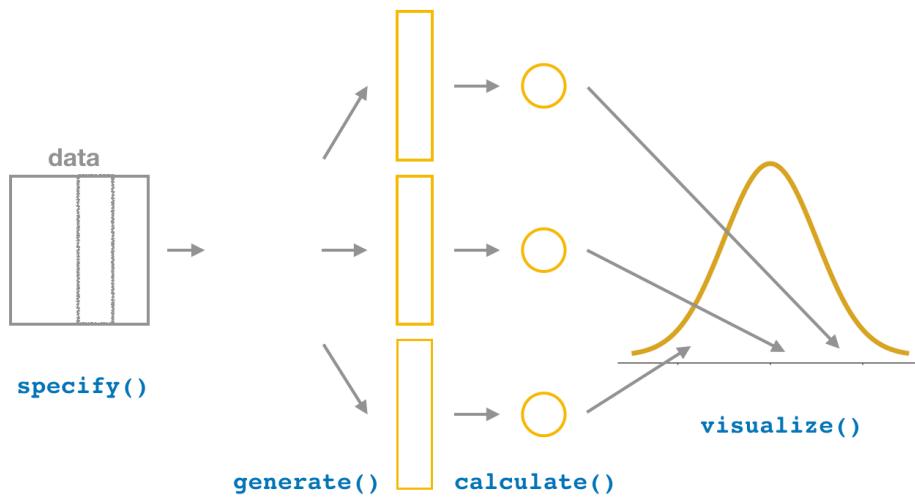
While different fields tend to use different values of α , some commonly used values for α are 0.1, 0.01, and 0.05; with 0.05 being the choice people often make without putting much thought into it. We’ll talk more about α significance levels in Section 9.4, but first let’s fully conduct the hypothesis test corresponding to our genre popularity activity using the `infer` package.

9.3 Conducting hypothesis tests

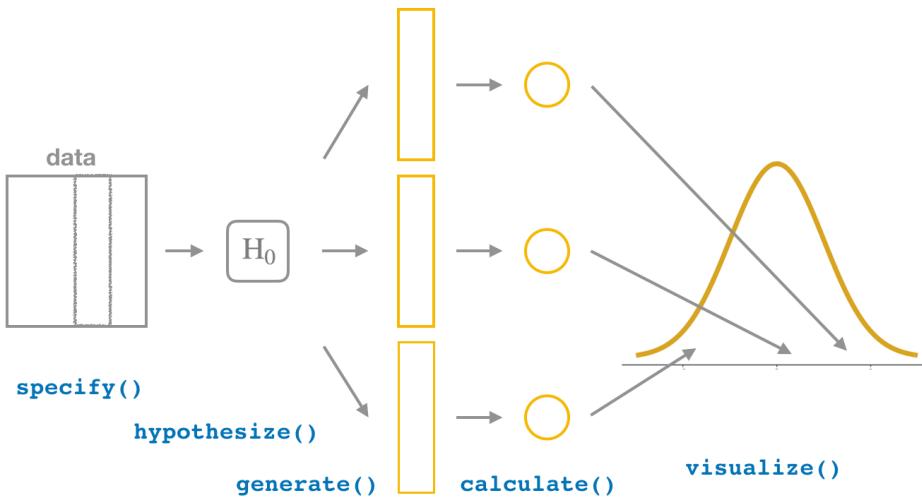
In Section 8.4, we showed you how to construct confidence intervals. We first illustrated how to do this using `dplyr` data wrangling verbs and the `rep_sample_n()` function from Subsection 7.2.3 which we used as a virtual shovel. In particular, we constructed confidence intervals by resampling with replacement by setting the `replace = TRUE` argument to the `rep_sample_n()` function.

We then showed you how to perform the same task using the `infer` package workflow. While both workflows resulted in the same bootstrap distribution from which we can construct confidence intervals, the `infer` package workflow emphasizes each of the steps in the overall process in Figure 9.5. It does so using function names that are intuitively named with verbs:

1. `specify()` the variables of interest in your data frame.
2. `generate()` replicates of bootstrap resamples with replacement.
3. `calculate()` the summary statistic of interest.
4. `visualize()` the resulting bootstrap distribution and confidence interval.

**FIGURE 9.5:** Confidence intervals with the infer package.

In this section, we'll now show you how to seamlessly modify the previously seen `infer` code for constructing confidence intervals to conduct hypothesis tests. You'll notice that the basic outline of the workflow is almost identical, except for an additional `hypothesize()` step between the `specify()` and `generate()` steps, as can be seen in Figure 9.6.

**FIGURE 9.6:** Hypothesis testing with the infer package.

Furthermore, we'll use a pre-specified significance level $\alpha = 0.1$ for this hypothesis test. Let's leave discussion on the choice of this α value until later on in Section 9.4.

9.3.1 infer package workflow

1. specify variables

Recall that we use the `specify()` verb to specify the response variable and, if needed, any explanatory variables for our study. In this case, since we are interested in any potential effects of genre on popularity rates, we set `popular_or_not` as the response variable and `track_genre` as the explanatory variable. We do so using `formula = response ~ explanatory` where `response` is the name of the response variable in the data frame and `explanatory` is the name of the explanatory variable. So in our case it is `popular_or_not ~ track_genre`.

Furthermore, since we are interested in the proportion of songs "promoted", and not the proportion of songs not promoted, we set the argument `success` to "promoted".

```
spotify_metal_deephouse |>
  specify(formula = popular_or_not ~ track_genre, success = "popular")
```

```
Response: popular_or_not (factor)
Explanatory: track_genre (factor)
# A tibble: 2,000 x 2
  popular_or_not track_genre
  <fct>          <fct>
  1 popular       deep-house
  2 popular       deep-house
  3 popular       deep-house
  4 popular       deep-house
  5 popular       deep-house
  6 popular       deep-house
  7 popular       deep-house
  8 popular       deep-house
  9 popular       deep-house
 10 popular      deep-house
# i 1,990 more rows
```

Again, notice how the `spotify_metal_deephouse` data itself doesn't change, but the `Response: decision (factor)` and `Explanatory: gender (factor)` *meta-data* do. This is similar to how the `group_by()` verb from `dplyr` doesn't change the data, but only adds "grouping" meta-data, as we saw in Section 3.4. We also now focus on only the two columns of interest in the data for our problem at hand with `popular_or_not` and `track_genre`.

2. hypothesize the null

In order to conduct hypothesis tests using the `infer` workflow, we need a new step not present for confidence intervals: `hypothesize()`. Recall from Section 9.2 that our hypothesis test was

$$H_0 : p_m - p_d = 0$$

vs. $H_A : p_m - p_d > 0$

In other words, the null hypothesis H_0 corresponding to our “hypothesized universe” stated that there was no difference in genre popularity rates. We set this null hypothesis H_0 in our `infer` workflow using the `null` argument of the `hypothesize()` function to either:

- “point” for hypotheses involving a single sample or
- “independence” for hypotheses involving two samples.

In our case, since we have two samples (the metal songs and the deep house songs), we set `null = "independence"`.

```
spotify_metal_deephause |>
  specify(formula = popular_or_not ~ track_genre, success = "popular") |>
  hypothesize(null = "independence")
```

```
Response: popular_or_not (factor)
Explanatory: track_genre (factor)
Null Hypothesis: independence
# A tibble: 2,000 x 2
  popular_or_not track_genre
  <fct>          <fct>
  1 popular       deep-house
  2 popular       deep-house
  3 popular       deep-house
  4 popular       deep-house
  5 popular       deep-house
  6 popular       deep-house
  7 popular       deep-house
  8 popular       deep-house
  9 popular       deep-house
 10 popular      deep-house
# i 1,990 more rows
```

Again, the data has not changed yet. This will occur at the upcoming `generate()` step; we're merely setting meta-data for now.

Where do the terms "point" and "independence" come from? These are two technical statistical terms. The term "point" relates from the fact that for a single group of observations, you will test the value of a single point. Going back to the pennies example from Chapter 8, say we wanted to test if the mean weight of all chocolate-covered almonds was equal to 3.5 grams or not. We would be testing the value of a "point" μ , the mean weight in grams of *all* chocolate-covered almonds, as follows

$$H_0 : \mu = 1993 \\ \text{vs } H_A : \mu \neq 3.5$$

The term "independence" relates to the fact that for two groups of observations, you are testing whether or not the response variable is *independent* of the explanatory variable that assigns the groups. In our case, we are testing whether the `popular_or_not` response variable is "independent" of the explanatory variable `track_genre`.

3. generate replicates

After we `hypothesize()` the null hypothesis, we `generate()` replicates of "shuffled" datasets assuming the null hypothesis is true. We do this by repeating the shuffling exercise you performed in Section 9.1 several times on the full dataset of 2000 rows. Let's use the computer to repeat this 1000 times by setting `reps = 1000` in the `generate()` function. However, unlike for confidence intervals where we generated replicates using `type = "bootstrap"` resampling with replacement, we'll now perform shuffles/permuations by setting `type = "permute"`. Recall that shuffles/permuations are a kind of resampling, but unlike the bootstrap method, they involve resampling *without* replacement.

```
spotify_generate <- spotify_metal_deephouse |>
  specify(formula = popular_or_not ~ track_genre, success = "popular") |>
  hypothesize(null = "independence") |>
  generate(reps = 1000, type = "permute")
nrow(spotify_generate)
```

[1] 2000000

Observe that the resulting data frame has 2,000,000 rows. This is because we performed shuffles/permuations for each of the 2000 rows 1000 times and $2,000,000 = 1000 \cdot 2000$. If you explore the `spotify_generate` data frame with `View()`, you'll notice that the variable `replicate` indicates which resample each row belongs to. So it has the value 1 2000 times, the value 2 2000 times, all the way through to the value 1000 2000 times.

4. calculate summary statistics

Now that we have generated 1000 replicates of “shuffles” assuming the null hypothesis is true, let’s `calculate()` the appropriate summary statistic for each of our 1000 shuffles. From Section 9.2, point estimates related to hypothesis testing have a specific name: *test statistics*. Since the unknown population parameter of interest is the difference in population proportions $p_m - p_d$, the test statistic here is the difference in sample proportions $\hat{p}_m - \hat{p}_d$.

For each of our 1000 shuffles, we can calculate this test statistic by setting `stat = "diff in props"`. Furthermore, since we are interested in $\hat{p}_m - \hat{p}_d$ we set `order = c("metal", "deep-house")`. As we stated earlier, the order of the subtraction does not matter, so long as you stay consistent throughout your analysis and tailor your interpretations accordingly.

Let’s save the result in a data frame called `null_distribution`:

```
null_distribution <- spotify_metal_deephause |>
  specify(formula = popular_or_not ~ track_genre, success = "popular") |>
  hypothesize(null = "independence") |>
  generate(reps = 1000, type = "permute") |>
  calculate(stat = "diff in props", order = c("metal", "deep-house"))
null_distribution
```

```
Response: popular_or_not (factor)
Explanatory: track_genre (factor)
Null Hypothesis: independence
# A tibble: 1,000 x 2
  replicate      stat
  <int>     <dbl>
1       1  0.0140000
2       2 -0.0420000
3       3  0.0220000
4       4 -0.0140000
5       5 -0.0180000
6       6 -0.0160000
7       7  0.0160000
8       8 -0.0400000
9       9  0.0140000
10      10  0.0120000
# i 990 more rows
```

Observe that we have 1000 values of `stat`, each representing one instance of $\hat{p}_m - \hat{p}_d$ in a hypothesized world of no difference in genre popularity. Observe as well that we

chose the name of this data frame carefully: `null_distribution`. Recall once again from Section 9.2 that sampling distributions when the null hypothesis H_0 is assumed to be true have a special name: the *null distribution*.

What was the *observed* difference in popularity rates? In other words, what was the *observed test statistic* $\hat{p}_m - \hat{p}_f$? Recall from Section 9.1 that we computed this observed difference by hand to be $0.563 - 0.529 = 0.034 = 3.4\%$. We can also compute this value using the previous `infer` code but with the `hypothesize()` and `generate()` steps removed. Let's save this in `obs_diff_prop`:

```
obs_diff_prop <- spotify_metal_deephause |>
  specify(formula = popular_or_not ~ track_genre, success = "popular") |>
  calculate(stat = "diff in props", order = c("metal", "deep-house"))
obs_diff_prop
```

```
Response: popular_or_not (factor)
Explanatory: track_genre (factor)
# A tibble: 1 x 1
  stat
  <dbl>
1 0.0340000
```

Note that there is also a wrapper function in `infer` called `observe()` that can be used to calculate the observed test statistic. However, we chose to use the `specify()`, `calculate()`, and `hypothesize()` functions to help you continue to use the common verbs and build practice with them.

```
spotify_metal_deephause |>
  observe(formula = popular_or_not ~ track_genre,
          success = "popular",
          stat = "diff in props",
          order = c("metal", "deep-house"))
```

```
Response: popular_or_not (factor)
Explanatory: track_genre (factor)
# A tibble: 1 x 1
  stat
  <dbl>
1 0.0340000
```

5. visualize the p-value

The final step is to measure how surprised we are by a promotion difference of 3.4% in a hypothesized universe of no difference in genre popularity. If the observed difference of 0.034 is highly unlikely, then we would be inclined to reject the validity of our hypothesized universe.

We start by visualizing the *null distribution* of our 1000 values of $\hat{p}_m - \hat{p}_d$ using `visualize()` in Figure 9.7. Recall that these are values of the difference in popularity rates assuming H_0 is true. This corresponds to being in our hypothesized universe of no difference in genre popularity.

```
visualize(null_distribution, bins = 25)
```

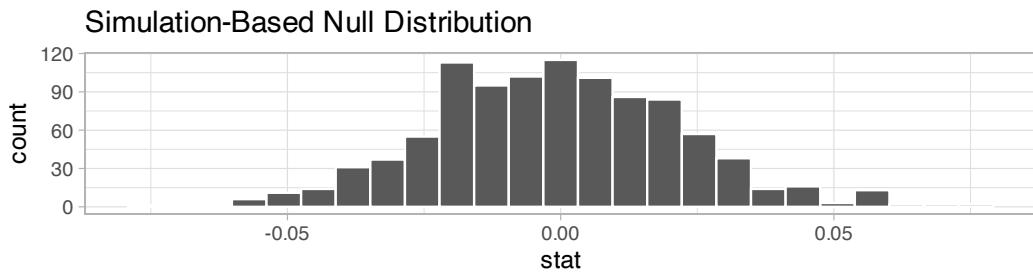


FIGURE 9.7: Null distribution.

Let's now add what happened in real life to Figure 9.7, the observed difference in popularity rates of $0.563 - 0.529 = 0.034 = 3.4\%$. However, instead of merely adding a vertical line using `geom_vline()`, let's use the `shade_p_value()` function with `obs_stat` set to the observed test statistic value we saved in `obs_diff_prop`.

Furthermore, we'll set the `direction = "right"` reflecting our alternative hypothesis $H_A : p_m - p_d > 0$. Recall our alternative hypothesis H_A is that $p_m - p_d > 0$, stating that there is a difference in popularity rates in favor of metal songs. “More extreme” here corresponds to differences that are “bigger” or “more positive” or “more to the right.” Hence we set the `direction` argument of `shade_p_value()` to be `"right"`.

On the other hand, had our alternative hypothesis H_A been the other possible one-sided alternative $p_m - p_d < 0$, suggesting popularity in favor of deep house songs, we would've set `direction = "left"`. Had our alternative hypothesis H_A been two-sided $p_m - p_d \neq 0$, suggesting discrimination in either direction, we would've set `direction = "both"`.

```
visualize(null_distribution, bins = 25) +
  shade_p_value(obs_stat = obs_diff_prop, direction = "right")
```

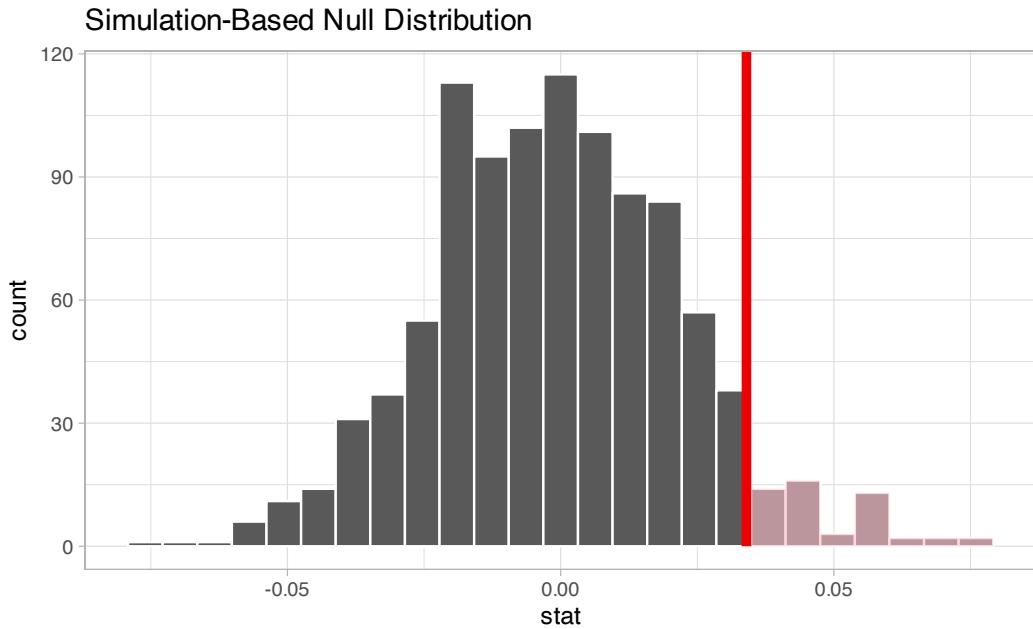


FIGURE 9.8: Shaded histogram to show *p*-value.

In the resulting Figure 9.8, the solid dark line marks $0.034 = 3.4\%$. However, what does the shaded-region correspond to? This is the *p*-value. Recall the definition of the *p*-value from Section 9.2:

A *p*-value is the probability of obtaining a test statistic just as or more extreme than the observed test statistic *assuming the null hypothesis H_0 is true*.

So judging by the shaded region in Figure 9.8, it seems we would somewhat rarely observe differences in promotion rates of $0.034 = 3.4\%$ or more in a hypothesized universe of no difference in genre popularity. In other words, the *p*-value is somewhat small. Hence, we would be inclined to reject this hypothesized universe, or using statistical language we would “reject H_0 .”

What fraction of the null distribution is shaded? In other words, what is the exact value of the *p*-value? We can compute it using the `get_p_value()` function with the same arguments as the previous `shade_p_value()` code:

```
null_distribution |>
  get_p_value(obs_stat = obs_diff_prop, direction = "right")
```

```
# A tibble: 1 × 1
  p_value
  <dbl>
1 0.065
```

Keeping the definition of a p -value in mind, the probability of observing a difference in promotion rates as large as $0.034 = 3.4\%$ due to sampling variation alone in the null distribution is $0.065 = 6.5\%$. Since this p -value is smaller than our pre-specified significance level $\alpha = 0.1$, we reject the null hypothesis $H_0 : p_m - p_d = 0$. In other words, this p -value is sufficiently small to reject our hypothesized universe of no difference in genre popularity. We instead have enough evidence to change our mind in favor of difference in genre popularity being a likely culprit here. Observe that whether we reject the null hypothesis H_0 or not depends in large part on our choice of significance level α . We'll discuss this more in Subsection 9.4.3.

9.3.2 Comparison with confidence intervals

One of the great things about the `infer` package is that we can jump seamlessly between conducting hypothesis tests and constructing confidence intervals with minimal changes! Recall the code from the previous section that creates the null distribution, which in turn is needed to compute the p -value:

```
null_distribution <- spotify_metal_deephause |>
  specify(formula = popular_or_not ~ track_genre, success = "popular") |>
  hypothesize(null = "independence") |>
  generate(reps = 1000, type = "permute") |>
  calculate(stat = "diff in props", order = c("metal", "deep-house"))
```

To create the corresponding bootstrap distribution needed to construct a 90% confidence interval for $p_m - p_d$, we only need to make two changes. First, we remove the `hypothesize()` step since we are no longer assuming a null hypothesis H_0 is true. We can do this by deleting or commenting out the `hypothesize()` line of code. Second, we switch the type of resampling in the `generate()` step to be "bootstrap" instead of "permute".

```
bootstrap_distribution <- spotify_metal_deephause |>
  specify(formula = popular_or_not ~ track_genre, success = "popular") |>
  # Change 1 - Remove hypothesize():
  # hypothesize(null = "independence") |>
  # Change 2 - Switch type from "permute" to "bootstrap":
  generate(reps = 1000, type = "bootstrap") |>
  calculate(stat = "diff in props", order = c("metal", "deep-house"))
```

Using this `bootstrap_distribution`, let's first compute the percentile-based confidence intervals, as we did in Section 8.4:

```
percentile_ci <- bootstrap_distribution |>
  get_confidence_interval(level = 0.90, type = "percentile")
percentile_ci
```

```
# A tibble: 1 × 2
  lower_ci   upper_ci
  <dbl>      <dbl>
1 0.000355780 0.0701690
```

Using our shorthand interpretation for 90% confidence intervals from Subsection 8.5.2, we are 90% “confident” that the true difference in population proportions $p_m - p_d$ is between (0, 0.07). Let's visualize `bootstrap_distribution` and this percentile-based 90% confidence interval for $p_m - p_d$ in Figure 9.9.

```
visualize(bootstrap_distribution) +
  shade_confidence_interval(endpoints = percentile_ci)
```

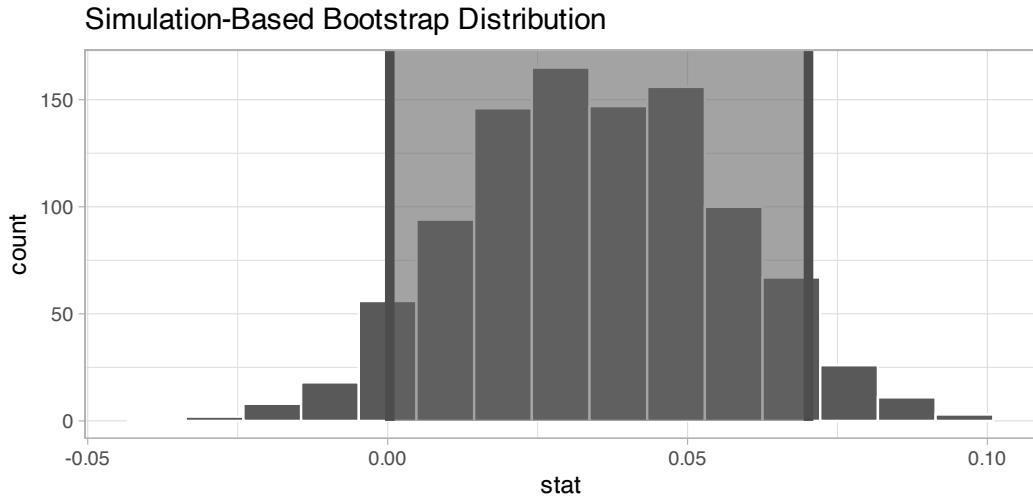


FIGURE 9.9: Percentile-based 95% confidence interval.

Notice a key value that is not included in the 95% confidence interval for $p_m - p_d$: the value 0 (but just barely!). In other words, a difference of 0 is not included in our net, suggesting that p_m and p_d are truly different! Furthermore, observe how the entirety of the 95% confidence interval for $p_m - p_d$ lies above 0, suggesting that this difference is in favor of metal.

Learning check

(LC9.1) Why does the following code produce an error? In other words, what about the response and predictor variables make this not a possible computation with the `infer` package?

```
library(moderndive)
library(infer)
null_distribution_mean <- spotify_metal_deephouse |>
  specify(formula = popular_or_not ~ track_genre, success = "popular") |>
  hypothesize(null = "independence") |>
  generate(reps = 1000, type = "permute") |>
  calculate(stat = "diff in means", order = c("metal", "deep-house"))
```

(LC9.2) Why are we relatively confident that the distributions of the sample proportions will be good approximations of the population distributions of popularity proportions for the two genres?

(LC9.3) Using the definition of *p-value*, write in words what the *p*-value represents for the hypothesis test comparing the popularity rates for metal and deep house.

9.3.3 “There is only one test”

Let’s recap the steps necessary to conduct a hypothesis test using the terminology, notation, and definitions related to sampling you saw in Section 9.2 and the `infer` workflow from Subsection 9.3.1:

1. `specify()` the variables of interest in your data frame.
2. `hypothesize()` the null hypothesis H_0 . In other words, set a “model for the universe” assuming H_0 is true.
3. `generate()` shuffles assuming H_0 is true. In other words, *simulate* data assuming H_0 is true.
4. `calculate()` the *test statistic* of interest, both for the observed data and your *simulated* data.
5. `visualize()` the resulting *null distribution* and compute the *p-value* by comparing the null distribution to the observed test statistic.

While this is a lot to digest, especially the first time you encounter hypothesis testing, the nice thing is that once you understand this general framework, then you can

understand *any* hypothesis test. In a famous blog post, computer scientist Allen Downey called this the “There is only one test”¹ framework, for which he created the flowchart displayed in Figure 9.10.

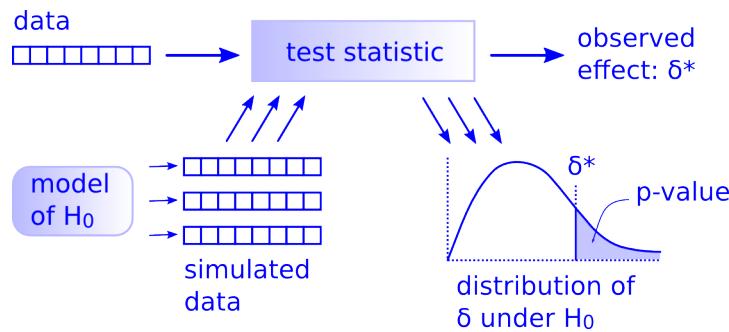


FIGURE 9.10: Allen Downey’s hypothesis testing framework.

Notice its similarity with the “hypothesis testing with `infer`” diagram you saw in Figure 9.6. That’s because the `infer` package was explicitly designed to match the “There is only one test” framework. So if you can understand the framework, you can easily generalize these ideas for all hypothesis testing scenarios. Whether for population proportions p , population means μ , differences in population proportions $p_1 - p_2$, differences in population means $\mu_1 - \mu_2$, and as you’ll see in Chapter 10 on inference for regression, population regression slopes β_1 as well. In fact, it applies more generally even than just these examples to more complicated hypothesis tests and test statistics as well.

Learning check

(LC9.4) Describe in a paragraph how we used Allen Downey’s diagram to conclude if a statistical difference existed between the popularity rate of metal and deep house for the Spotify example.

9.4 Interpreting hypothesis tests

Interpreting the results of hypothesis tests is one of the more challenging aspects of this method for statistical inference. In this section, we’ll focus on ways to help with deciphering the process and address some common misconceptions.

¹<http://allendowney.blogspot.com/2016/06/there-is-still-only-one-test.html>

9.4.1 Two possible outcomes

In Section 9.2, we mentioned that given a pre-specified significance level α there are two possible outcomes of a hypothesis test:

- If the p -value is less than α , then we *reject* the null hypothesis H_0 in favor of H_A .
- If the p -value is greater than or equal to α , we *fail to reject* the null hypothesis H_0 .

Unfortunately, the latter result is often misinterpreted as “accepting the null hypothesis H_0 .” While at first glance it may seem that the statements “failing to reject H_0 ” and “accepting H_0 ” are equivalent, there actually is a subtle difference. Saying that we “accept the null hypothesis H_0 ” is equivalent to stating that “we think the null hypothesis H_0 is true.” However, saying that we “fail to reject the null hypothesis H_0 ” is saying something else: “While H_0 might still be false, we don’t have enough evidence to say so.” In other words, there is an absence of enough proof. However, the absence of proof is not proof of absence.

To further shed light on this distinction, let’s use the United States criminal justice system as an analogy. A criminal trial in the United States is a similar situation to hypothesis tests whereby a choice between two contradictory claims must be made about a defendant who is on trial:

1. The defendant is truly either “innocent” or “guilty.”
2. The defendant is presumed “innocent until proven guilty.”
3. The defendant is found guilty only if there is *strong evidence* that the defendant is guilty. The phrase “beyond a reasonable doubt” is often used as a guideline for determining a cutoff for when enough evidence exists to find the defendant guilty.
4. The defendant is found to be either “not guilty” or “guilty” in the ultimate verdict.

In other words, *not guilty* verdicts are not suggesting the defendant is *innocent*, but instead that “while the defendant may still actually be guilty, there wasn’t enough evidence to prove this fact.” Now let’s make the connection with hypothesis tests:

1. Either the null hypothesis H_0 or the alternative hypothesis H_A is true.
2. Hypothesis tests are conducted assuming the null hypothesis H_0 is true.
3. We reject the null hypothesis H_0 in favor of H_A only if the evidence found in the sample suggests that H_A is true. The significance level α is used as a guideline to set the threshold on just how strong of evidence we require.
4. We ultimately decide to either “fail to reject H_0 ” or “reject H_0 .”

So while gut instinct may suggest “failing to reject H_0 ” and “accepting H_0 ” are equivalent statements, they are not. “Accepting H_0 ” is equivalent to finding a defendant innocent. However, courts do not find defendants “innocent,” but rather they find

them “not guilty.” Putting things differently, defense attorneys do not need to prove that their clients are innocent, rather they only need to prove that clients are not “guilty beyond a reasonable doubt”.

So going back to our songs activity in Section 9.3, recall that our hypothesis test was $H_0 : p_m - p_d = 0$ versus $H_A : p_m - p_d > 0$ and that we used a pre-specified significance level of $\alpha = 0.1$. We found a p -value of 0.065. Since the p -value was smaller than $\alpha = 0.1$, we rejected H_0 . In other words, we found needed levels of evidence in this particular sample to say that H_0 is false at the $\alpha = 0.1$ significance level. We also state this conclusion using non-statistical language: we found enough evidence in this data to suggest that there was gender discrimination at play.

9.4.2 Types of errors

Unfortunately, there is some chance a jury or a judge can make an incorrect decision in a criminal trial by reaching the wrong verdict. For example, finding a truly innocent defendant “guilty”. Or on the other hand, finding a truly guilty defendant “not guilty.” This can often stem from the fact that prosecutors don’t have access to all the relevant evidence, but instead are limited to whatever evidence the police can find.

The same holds for hypothesis tests. We can make incorrect decisions about a population parameter because we only have a sample of data from the population and thus sampling variation can lead us to incorrect conclusions.

There are two possible erroneous conclusions in a criminal trial: either (1) a truly innocent person is found guilty or (2) a truly guilty person is found not guilty. Similarly, there are two possible errors in a hypothesis test: either (1) rejecting H_0 when in fact H_0 is true, called a **Type I error** or (2) failing to reject H_0 when in fact H_0 is false, called a **Type II error**. Another term used for “Type I error” is “false positive,” while another term for “Type II error” is “false negative.”

This risk of error is the price researchers pay for basing inference on a sample instead of performing a census on the entire population. But as we’ve seen in our numerous examples and activities so far, censuses are often very expensive and other times impossible, and thus researchers have no choice but to use a sample. Thus in any hypothesis test based on a sample, we have no choice but to tolerate some chance that a Type I error will be made and some chance that a Type II error will occur.

To help understand the concepts of Type I error and Type II errors, we apply these terms to our criminal justice analogy in Figure 9.11.

	Truly not guilty	Truly guilty
Verdict		
Not guilty verdict	Correct	Type II error
Guilty verdict	Type I error	Correct

FIGURE 9.11: Type I and Type II errors in criminal trials.

Thus a Type I error corresponds to incorrectly putting a truly innocent person in jail, whereas a Type II error corresponds to letting a truly guilty person go free. Let's show the corresponding table in Figure 9.12 for hypothesis tests.

	H0 true	HA true
Verdict		
Fail to reject H0	Correct	Type II error
Reject H0	Type I error	Correct

FIGURE 9.12: Type I and Type II errors in hypothesis tests.

9.4.3 How do we choose alpha?

If we are using a sample to make inferences about a population, we run the risk of making errors. For confidence intervals, a corresponding “error” would be constructing a confidence interval that does not contain the true value of the population parameter. For hypothesis tests, this would be making either a Type I or Type II error. Obviously, we want to minimize the probability of either error; we want a small probability of making an incorrect conclusion:

- The probability of a Type I Error occurring is denoted by α . The value of α is called the *significance level* of the hypothesis test, which we defined in Section 9.2.
- The probability of a Type II Error is denoted by β . The value of $1 - \beta$ is known as the *power* of the hypothesis test.

In other words, α corresponds to the probability of incorrectly rejecting H_0 when in fact H_0 is true. On the other hand, β corresponds to the probability of incorrectly failing to reject H_0 when in fact H_0 is false.

Ideally, we want $\alpha = 0$ and $\beta = 0$, meaning that the chance of making either error is 0. However, this can never be the case in any situation where we are sampling for

inference. There will always be the possibility of making either error when we use sample data. Furthermore, these two error probabilities are inversely related. As the probability of a Type I error goes down, the probability of a Type II error goes up.

What is typically done in practice is to fix the probability of a Type I error by pre-specifying a significance level α and then try to minimize β . In other words, we will tolerate a certain fraction of incorrect rejections of the null hypothesis H_0 , and then try to minimize the fraction of incorrect non-rejections of H_0 .

So for example if we used $\alpha = 0.01$, we would be using a hypothesis testing procedure that in the long run would incorrectly reject the null hypothesis H_0 one percent of the time. This is analogous to setting the confidence level of a confidence interval.

So what value should you use for α ? Different fields have different conventions, but some commonly used values include 0.10, 0.05, 0.01, and 0.001. However, it is important to keep in mind that if you use a relatively small value of α , then all things being equal, p -values will have a harder time being less than α . Thus, we would reject the null hypothesis less often. In other words, we would reject the null hypothesis H_0 only if we have *very strong* evidence to do so. This is known as a “conservative” test.

On the other hand, if we used a relatively large value of α , then all things being equal, p -values will have an easier time being less than α . Thus we would reject the null hypothesis more often. In other words, we would reject the null hypothesis H_0 even if we only have *mild* evidence to do so. This is known as a “liberal” test.

Learning check

(LC9.5) What is wrong about saying, “The defendant is innocent.” based on the US system of criminal trials?

(LC9.6) What is the purpose of hypothesis testing?

(LC9.7) What are some flaws with hypothesis testing? How could we alleviate them?

(LC9.8) Consider two α significance levels of 0.1 and 0.01. Of the two, which would lead to a more *liberal* hypothesis testing procedure? In other words, one that will, all things being equal, lead to more rejections of the null hypothesis H_0 .

9.5 Case study: Are action or romance movies rated higher?

Let's apply our knowledge of hypothesis testing to answer the question: "Are action or romance movies rated higher on IMDb?". IMDb² is a database on the internet providing information on movie and television show casts, plot summaries, trivia, and ratings. We'll investigate if, on average, action or romance movies get higher ratings on IMDb.

9.5.1 IMDb ratings data

The `movies` dataset in the `ggplot2movies` package contains information on 58,788 movies that have been rated by users of IMDb.com.

```
movies
```

```
# A tibble: 58,788 x 24
   title      year length budget rating votes    r1    r2    r3    r4    r5
   <chr>     <int>  <int>  <dbl>  <int>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 1 $           1971    121     NA    6.4    348    4.5    4.5    4.5    4.5   14.5
 2 $1000 a ~  1939     71     NA     6     20     0    14.5    4.5   24.5   14.5
 3 $21 a Da~  1941      7     NA    8.2     5     0     0     0     0     0
 4 $40,000   1996    70     NA    8.2     6   14.5     0     0     0     0
 5 $50,000 ~  1975    71     NA    3.4    17   24.5    4.5     0   14.5   14.5
 6 $pent      2000    91     NA    4.3    45    4.5    4.5    4.5   14.5   14.5
 7 $windle    2002    93     NA    5.3   200    4.5     0    4.5    4.5   24.5
 8 '15'       2002    25     NA    6.7    24    4.5    4.5    4.5    4.5
 9 '38        1987    97     NA    6.6    18    4.5    4.5    4.5     0     0
10 '49-'17   1917    61     NA     6    51    4.5     0    4.5    4.5    4.5
# i 58,778 more rows
# i 13 more variables: r6 <dbl>, r7 <dbl>, r8 <dbl>, r9 <dbl>, r10 <dbl>,
#   mpaa <chr>, Action <int>, Animation <int>, Comedy <int>, Drama <int>,
#   Documentary <int>, Romance <int>, Short <int>
```

We'll focus on a random sample of 68 movies that are classified as either "action" or "romance" movies but not both. We disregard movies that are classified as both so that we can assign all 68 movies into either category. Furthermore, since the original `movies` dataset was a little messy, we provide a pre-wrangled version of our data in the `movies_sample` data frame included in the `moderndive` package. If you're curious, you can look at the necessary data wrangling code to do this on GitHub³.

²<https://www.imdb.com/>

³https://github.com/moderndive/moderndive/blob/master/data-raw/process_data_sets.R

```
movies_sample
```

```
# A tibble: 68 x 4
  title                  year rating genre
  <chr>                 <int>  <dbl> <chr>
1 Underworld              1985    3.1 Action
2 Love Affair              1932    6.3 Romance
3 Junglee                 1961    6.8 Romance
4 Eversmile, New Jersey   1989     5   Romance
5 Search and Destroy      1979     4   Action
6 Secreto de Romelia, El  1988    4.9 Romance
7 Amants du Pont-Neuf, Les 1991    7.4 Romance
8 Illicit Dreams          1995    3.5 Action
9 Kabhi Kabhie             1976    7.7 Romance
10 Electric Horseman, The 1979    5.8 Romance
# i 58 more rows
```

The variables include the `title` and `year` the movie was filmed. Furthermore, we have a numerical variable `rating`, which is the IMDb rating out of 10 stars, and a binary categorical variable `genre` indicating if the movie was an `Action` or `Romance` movie. We are interested in whether `Action` or `Romance` movies got a higher `rating` on average.

Let's perform an exploratory data analysis of this data. Recall from Subsection 2.7.1 that a boxplot is a visualization we can use to show the relationship between a numerical and a categorical variable. Another option you saw in Section 2.6 would be to use a faceted histogram. However, in the interest of brevity, let's only present the boxplot in Figure 9.13.

```
ggplot(data = movies_sample, aes(x = genre, y = rating)) +
  geom_boxplot() +
  labs(y = "IMDb rating")
```

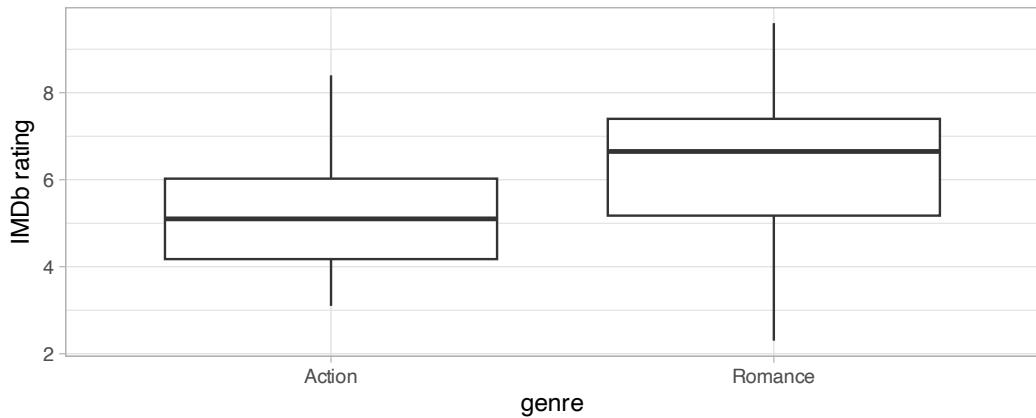


FIGURE 9.13: Boxplot of IMDb rating vs. genre.

Eyeballing Figure 9.13, romance movies have a higher median rating. Do we have reason to believe, however, that there is a *significant* difference between the mean rating for action movies compared to romance movies? It's hard to say just based on this plot. The boxplot does show that the median sample rating is higher for romance movies.

However, there is a large amount of overlap between the boxes. Recall that the median isn't necessarily the same as the mean either, depending on whether the distribution is skewed.

Let's calculate some summary statistics split by the binary categorical variable `genre`: the number of movies, the mean rating, and the standard deviation split by `genre`. We'll do this using `dplyr` data wrangling verbs. Notice in particular how we count the number of each type of movie using the `n()` summary function.

```
movies_sample |>
  group_by(genre) |>
  summarize(n = n(), mean_rating = mean(rating), std_dev = sd(rating))
```

```
# A tibble: 2 x 4
  genre      n  mean_rating  std_dev
  <chr>    <int>     <dbl>    <dbl>
1 Action     32      5.275   1.36121
2 Romance    36      6.32222  1.60963
```

Observe that we have 36 movies with an average rating of 6.322 stars and 32 movies with an average rating of 5.275 stars. The difference in these average ratings is thus $6.322 - 5.275 = 1.047$. So there appears to be an edge of 1.047 stars in favor of romance movies. The question is, however, are these results indicative of a true difference for *all*

romance and action movies? Or could we attribute this difference to chance *sampling variation*?

9.5.2 Sampling scenario

Let's now revisit this study in terms of terminology and notation related to sampling we studied in Subsection 7.3.1. The *study population* is all movies in the IMDb database that are either action or romance (but not both). The *sample* from this population is the 68 movies included in the `movies_sample` dataset.

Since this sample was randomly taken from the population `movies`, it is representative of all romance and action movies on IMDb. Thus, any analysis and results based on `movies_sample` can generalize to the entire population. What are the relevant *population parameter* and *point estimates*? We introduce the fourth sampling scenario in Table 9.2.

TABLE 9.2: Scenarios of sampling for inference

Scenario	Population parameter	Notation	Point estimate	Symbol(s)
1	Population proportion	p	Sample proportion	\hat{p}
2	Population mean	μ	Sample mean	\bar{x} or $\hat{\mu}$
3	Difference in population proportions	$p_1 - p_2$	Difference in sample proportions	$\hat{p}_1 - \hat{p}_2$
4	Difference in population means	$\mu_1 - \mu_2$	Difference in sample means	$\bar{x}_1 - \bar{x}_2$ or $\hat{\mu}_1 - \hat{\mu}_2$

So, whereas the sampling bowl exercise in Section 7.1 concerned *proportions*, the pennies exercise in Section 8.1 concerned *means*, the case study on whether yawning is contagious in Section 8.6 and the promotions activity in Section 9.1 concerned *differences in proportions*, we are now concerned with *differences in means*.

In other words, the population parameter of interest is the difference in population mean ratings $\mu_a - \mu_r$, where μ_a is the mean rating of all action movies on IMDb and similarly μ_r is the mean rating of all romance movies. Additionally the point estimate/sample statistic of interest is the difference in sample means $\bar{x}_a - \bar{x}_r$, where \bar{x}_a is the mean rating of the $n_a = 32$ movies in our sample and \bar{x}_r is the mean rating of the $n_r = 36$ in our sample. Based on our earlier exploratory data analysis, our estimate $\bar{x}_a - \bar{x}_r$ is $5.275 - 6.322 = -1.047$.

So there appears to be a slight difference of -1.047 in favor of romance movies. The question is, however, could this difference of -1.047 be merely due to chance and

sampling variation? Or are these results indicative of a true difference in mean ratings for *all* romance and action movies on IMDb? To answer this question, we'll use hypothesis testing.

9.5.3 Conducting the hypothesis test

We'll be testing:

$$\begin{aligned} H_0 : \mu_a - \mu_r &= 0 \\ \text{vs } H_A : \mu_a - \mu_r &\neq 0 \end{aligned}$$

In other words, the null hypothesis H_0 suggests that both romance and action movies have the same mean rating. This is the “hypothesized universe” we'll *assume* is true. On the other hand, the alternative hypothesis H_A suggests that there is a difference. Unlike the one-sided alternative we used in the promotions exercise $H_A : p_m - p_f > 0$, we are now considering a two-sided alternative of $H_A : \mu_a - \mu_r \neq 0$.

Furthermore, we'll pre-specify a low significance level of $\alpha = 0.001$. By setting this value low, all things being equal, there is a lower chance that the *p*-value will be less than α . Thus, there is a lower chance that we'll reject the null hypothesis H_0 in favor of the alternative hypothesis H_A . In other words, we'll reject the hypothesis that there is no difference in mean ratings for all action and romance movies, only if we have quite strong evidence. This is known as a “conservative” hypothesis testing procedure.

1. specify variables

Let's now perform all the steps of the `infer` workflow. We first `specify()` the variables of interest in the `movies_sample` data frame using the formula `rating ~ genre`. This tells `infer` that the numerical variable `rating` is the outcome variable, while the binary variable `genre` is the explanatory variable. Note that unlike previously when we were interested in proportions, since we are now interested in the mean of a numerical variable, we do not need to set the `success` argument.

```
movies_sample |>
  specify(formula = rating ~ genre)
```

```
Response: rating (numeric)
Explanatory: genre (factor)
# A tibble: 68 x 2
  rating genre
  <dbl> <fct>
1     3.1 Action
```

```

2   6.3 Romance
3   6.8 Romance
4   5   Romance
5   4   Action
6   4.9 Romance
7   7.4 Romance
8   3.5 Action
9   7.7 Romance
10  5.8 Romance
# i 58 more rows

```

Observe at this point that the data in `movies_sample` has not changed. The only change so far is the newly defined Response: `rating` (numeric) and Explanatory: `genre` (factor) *meta-data*.

2. hypothesize the null

We set the null hypothesis $H_0 : \mu_a - \mu_r = 0$ by using the `hypothesize()` function. Since we have two samples, action and romance movies, we set `null` to be "independence" as we described in Section 9.3.

```

movies_sample |>
  specify(formula = rating ~ genre) |>
  hypothesize(null = "independence")

```

```

Response: rating (numeric)
Explanatory: genre (factor)
Null Hypothesis: independence
# A tibble: 68 x 2
  rating genre
  <dbl> <fct>
1   3.1 Action
2   6.3 Romance
3   6.8 Romance
4   5   Romance
5   4   Action
6   4.9 Romance
7   7.4 Romance
8   3.5 Action
9   7.7 Romance
10  5.8 Romance
# i 58 more rows

```

3. generate replicates

After we have set the null hypothesis, we generate “shuffled” replicates assuming the null hypothesis is true by repeating the shuffling/permuation exercise you performed in Section 9.1.

We’ll repeat this resampling without replacement of `type = "permute"` a total of `reps = 1000` times. Feel free to run the code below to check out what the `generate()` step produces.

```
movies_sample |>
  specify(formula = rating ~ genre) |>
  hypothesize(null = "independence") |>
  generate(reps = 1000, type = "permute") |>
  View()
```

4. calculate summary statistics

Now that we have 1000 replicated “shuffles” assuming the null hypothesis H_0 that both Action and Romance movies on average have the same ratings on IMDb, let’s `calculate()` the appropriate summary statistic for these 1000 replicated shuffles. From Section 9.2, summary statistics relating to hypothesis testing have a specific name: *test statistics*. Since the unknown population parameter of interest is the difference in population means $\mu_a - \mu_r$, the test statistic of interest here is the difference in sample means $\bar{x}_a - \bar{x}_r$.

For each of our 1000 shuffles, we can calculate this test statistic by setting `stat = "diff in means"`. Furthermore, since we are interested in $\bar{x}_a - \bar{x}_r$, we set `order = c("Action", "Romance")`. Let’s save the results in a data frame called `null_distribution_movies`:

```
null_distribution_movies <- movies_sample |>
  specify(formula = rating ~ genre) |>
  hypothesize(null = "independence") |>
  generate(reps = 1000, type = "permute") |>
  calculate(stat = "diff in means", order = c("Action", "Romance"))
null_distribution_movies
```

	replicate	stat
	<int>	<dbl>
1	1	0.511111
2	2	0.345833
3	3	-0.327083

```

4      4 -0.209028
5      5 -0.433333
6      6 -0.102778
7      7  0.387153
8      8  0.168750
9      9  0.257292
10     10  0.334028
# i 990 more rows

```

Observe that we have 1000 values of `stat`, each representing one instance of $\bar{x}_a - \bar{x}_r$. The 1000 values form the *null distribution*, which is the technical term for the sampling distribution of the difference in sample means $\bar{x}_a - \bar{x}_r$ assuming H_0 is true. What happened in real life? What was the observed difference in promotion rates? What was the *observed test statistic* $\bar{x}_a - \bar{x}_r$? Recall from our earlier data wrangling, this observed difference in means was $5.275 - 6.322 = -1.047$. We can also achieve this using the code that constructed the null distribution `null_distribution_movies` but with the `hypothesize()` and `generate()` steps removed. Let's save this in `obs_diff_means`:

```

obs_diff_means <- movies_sample |>
  specify(formula = rating ~ genre) |>
  calculate(stat = "diff in means", order = c("Action", "Romance"))
obs_diff_means

```

```

Response: rating (numeric)
Explanatory: genre (factor)
# A tibble: 1 x 1
  stat
  <dbl>
1 -1.04722

```

5. visualize the p-value

Lastly, in order to compute the *p-value*, we have to assess how “extreme” the observed difference in means of -1.047 is. We do this by comparing -1.047 to our null distribution, which was constructed in a hypothesized universe of no true difference in movie ratings. Let's visualize both the null distribution and the *p-value* in Figure 9.14. Unlike our example in Subsection 9.3.1 involving promotions, since we have a two-sided $H_A : \mu_a - \mu_r \neq 0$, we have to allow for both possibilities for *more extreme*, so we set `direction = "both"`.

```

visualize(null_distribution_movies, bins = 10) +
  shade_p_value(obs_stat = obs_diff_means, direction = "both")

```

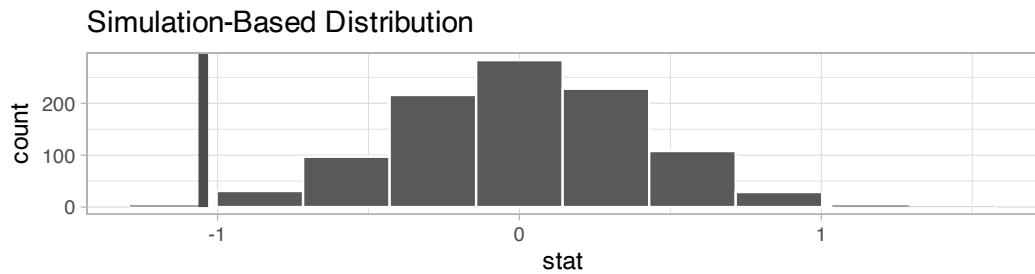


FIGURE 9.14: Null distribution, observed test statistic, and *p*-value.

Let's go over the elements of this plot. First, the histogram is the *null distribution*. Second, the solid line is the *observed test statistic*, or the difference in sample means we observed in real life of $5.275 - 6.322 = -1.047$. Third, the two shaded areas of the histogram form the *p-value*, or the probability of obtaining a test statistic just as or more extreme than the observed test statistic *assuming the null hypothesis H_0 is true*.

What proportion of the null distribution is shaded? In other words, what is the numerical value of the *p*-value? We use the `get_p_value()` function to compute this value:

```
null_distribution_movies |>
  get_p_value(obs_stat = obs_diff_means, direction = "both")
```

```
# A tibble: 1 × 1
  p_value
  <dbl>
1 0.004
```

This *p*-value of 0.004 is very small. In other words, there is a very small chance that we'd observe a difference of $5.275 - 6.322 = -1.047$ in a hypothesized universe where there was truly no difference in ratings.

But this *p*-value is larger than our (even smaller) pre-specified α significance level of 0.001. Thus, we are inclined to fail to reject the null hypothesis $H_0 : \mu_a - \mu_r = 0$. In non-statistical language, the conclusion is: we do not have the evidence needed in this sample of data to suggest that we should reject the hypothesis that there is no difference in mean IMDb ratings between romance and action movies. We, thus, cannot say that a difference exists in romance and action movie ratings, on average, for all IMDb movies.

Learning check

(LC9.9) Conduct the same analysis comparing action movies versus romantic movies using the median rating instead of the mean rating. What was different and what was the same?

(LC9.10) What conclusions can you make from viewing the faceted histogram looking at `rating` versus `genre` that you couldn't see when looking at the boxplot?

(LC9.11) Describe in a paragraph how we used Allen Downey's diagram to conclude if a statistical difference existed between mean movie ratings for action and romance movies.

(LC9.12) Why are we relatively confident that the distributions of the sample ratings will be good approximations of the population distributions of ratings for the two genres?

(LC9.13) Using the definition of *p*-value, write in words what the *p*-value represents for the hypothesis test comparing the mean rating of romance to action movies.

(LC9.14) What is the value of the *p*-value for the hypothesis test comparing the mean rating of romance to action movies?

(LC9.15) Test your data wrangling knowledge and EDA skills:

- Use `dplyr` and `tidyverse` to create the necessary data frame focused on only action and romance movies (but not both) from the `movies` data frame in the `ggplot2movies` package.
- Make a boxplot and a faceted histogram of this population data comparing ratings of action and romance movies from IMDb.
- Discuss how these plots compare to the similar plots produced for the `movies_sample` data.

9.6 Conclusion

9.6.1 Theory-based hypothesis tests

Much as we did in Subsections 7.6.2 and 8.7.2 when we showed you theory-based methods for computing standard errors and constructing confidence intervals that

involved mathematical formulas, we now present an example of a traditional theory-based method to conduct hypothesis tests. This method relies on probability models, probability distributions, and a few assumptions to construct the null distribution. This is in contrast to the approach we've been using throughout this book where we relied on computer simulations to construct the null distribution.

These traditional theory-based methods have been used for decades mostly because researchers didn't have access to computers that could run thousands of calculations quickly and efficiently. Now that computing power is much cheaper and more accessible, simulation-based methods are much more feasible. However, researchers in many fields continue to use theory-based methods. Hence, we make it a point to include an example here.

As we'll show in this section, any theory-based method is ultimately an approximation to the simulation-based method. The theory-based method we'll focus on is known as the *two-sample t-test* for testing differences in sample means. However, the test statistic we'll use won't be the difference in sample means $\bar{x}_1 - \bar{x}_2$, but rather the related *two-sample t-statistic*. The data we'll use will once again be the `movies_sample` data of action and romance movies from Section 9.5.

Two-sample t-statistic

A common task in statistics is the process of “standardizing a variable.” By standardizing different variables, we make them more comparable. For example, say you are interested in studying the distribution of temperature recordings from Phoenix, Arizona, USA and comparing it to that of the temperature recordings in Montreal, Quebec, Canada. Given that US temperatures are generally recorded in degrees Fahrenheit and Canadian temperatures are generally recorded in degrees Celsius, how can we make them comparable? One approach would be to convert degrees Fahrenheit into Celsius, or vice versa. Another approach would be to convert them both to a common “standardized” scale, like Kelvin units of temperature.

One common method for standardizing a variable from probability and statistics theory is to compute the *z-score*:

$$z = \frac{x - \mu}{\sigma}$$

where x represents one value of a variable, μ represents the mean of that variable, and σ represents the standard deviation of that variable. You first subtract the mean μ from each value of x and then divide $x - \mu$ by the standard deviation σ . These operations will have the effect of *re-centering* your variable around 0 and *re-scaling* your variable x so that they have what are known as “standard units.” Thus for every value that your variable can take, it has a corresponding *z-score* that gives how many standard units away that value is from the mean μ . *z*-scores are normally distributed with mean 0 and standard deviation 1. This curve is called a “*z*-distribution” or “standard normal” curve and has the common, bell-shaped pattern from Figure 9.15 discussed in Appendix A.2.

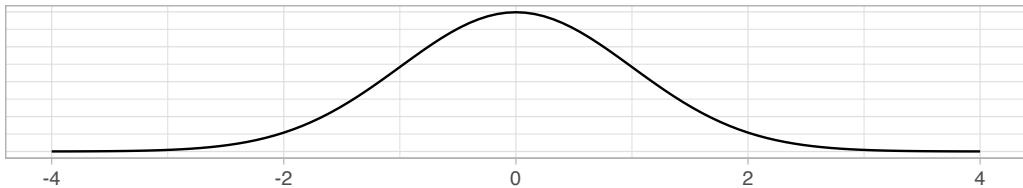


FIGURE 9.15: Standard normal z curve.

Bringing these back to the difference of sample mean ratings $\bar{x}_a - \bar{x}_r$ of action versus romance movies, how would we standardize this variable? By once again subtracting its mean and dividing by its standard deviation. Recall two facts from Subsection 7.3.3. First, if the sampling was done in a representative fashion, then the sampling distribution of $\bar{x}_a - \bar{x}_r$ will be centered at the true population parameter $\mu_a - \mu_r$. Second, the standard deviation of point estimates like $\bar{x}_a - \bar{x}_r$ has a special name: the standard error.

Applying these ideas, we present the *two-sample t-statistic*:

$$t = \frac{(\bar{x}_a - \bar{x}_r) - (\mu_a - \mu_r)}{\text{SE}_{\bar{x}_a - \bar{x}_r}} = \frac{(\bar{x}_a - \bar{x}_r) - (\mu_a - \mu_r)}{\sqrt{\frac{s_a^2}{n_a} + \frac{s_r^2}{n_r}}}$$

Oofda! There is a lot to try to unpack here! Let's go slowly. In the numerator, $\bar{x}_a - \bar{x}_r$ is the difference in sample means, while $\mu_a - \mu_r$ is the difference in population means. In the denominator, s_a and s_r are the *sample standard deviations* of the action and romance movies in our sample `movies_sample`. Lastly, n_a and n_r are the sample sizes of the action and romance movies. Putting this together under the square root gives us the standard error $\text{SE}_{\bar{x}_a - \bar{x}_r}$.

Observe that the formula for $\text{SE}_{\bar{x}_a - \bar{x}_r}$ has the sample sizes n_a and n_r in them. So as the sample sizes increase, the standard error goes down. We've seen this concept numerous times now, in particular (1) in our simulations using the three virtual shovels with $n = 25, 50$, and 100 slots in Figure 7.15, (2) in Subsection 8.5.3 where we studied the effect of using larger sample sizes on the widths of confidence intervals, and (3) in Subsection 7.6.2 where we studied the formula-based approximation to the standard error of the sample proportion \hat{p} .

So how can we use the two-sample *t*-statistic as a test statistic in our hypothesis test? First, assuming the null hypothesis $H_0 : \mu_a - \mu_r = 0$ is true, the right-hand side of the numerator (to the right of the $-$ sign), $\mu_a - \mu_r$, becomes 0.

Second, similarly to how the Central Limit Theorem from Subsection 7.5 states that sample means follow a normal distribution, it can be mathematically proven that the two-sample *t*-statistic follows a *t distribution with degrees of freedom* “roughly equal” to $df = n_a + n_r - 2$. To better understand this concept of *degrees of freedom*, we

next display three examples of t -distributions in Figure 9.16 along with the standard normal z curve.

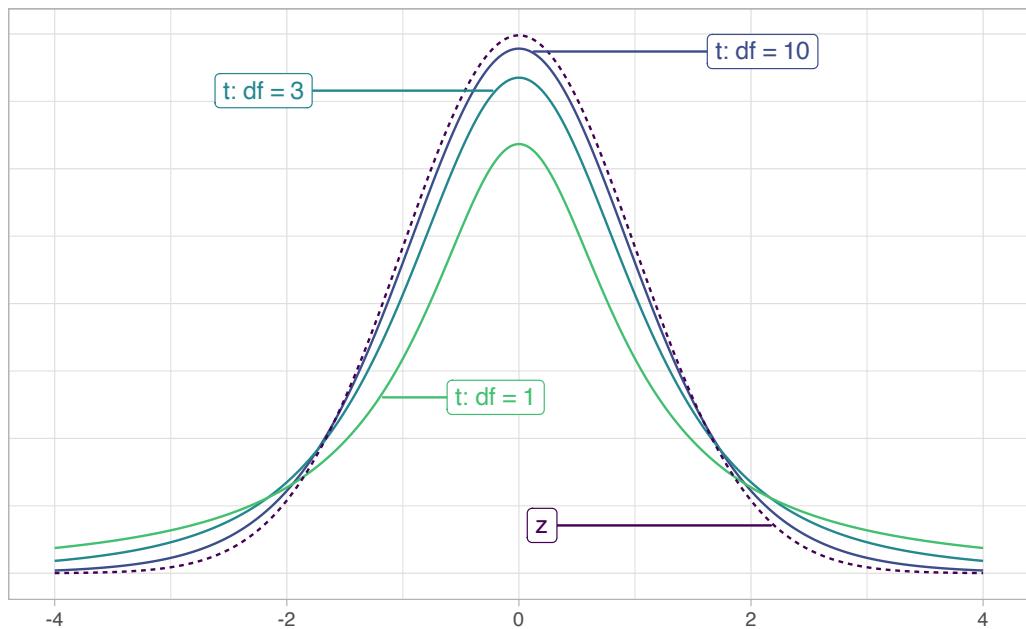


FIGURE 9.16: Examples of t -distributions and the z curve.

Begin by looking at the center of the plot at 0 on the horizontal axis. As you move up from the value of 0, follow along with the labels and note that the bottom curve corresponds to 1 degree of freedom, the curve above it is for 3 degrees of freedom, the curve above that is for 10 degrees of freedom, and lastly the dotted curve is the standard normal z curve.

Observe that all four curves have a bell shape, are centered at 0, and that as the degrees of freedom increase, the t -distribution more and more resembles the standard normal z curve. The “degrees of freedom” measures how different the t distribution will be from a normal distribution. t -distributions tend to have more values in the tails of their distributions than the standard normal z curve.

This “roughly equal” statement indicates that the equation $df = n_a + n_r - 2$ is a “good enough” approximation to the true degrees of freedom. The true formula is a bit more complicated than this simple expression, but we’ve found the formula to be beyond the reach of those new to statistical inference and it does little to build the intuition of the t -test.

The message to retain, however, is that small sample sizes lead to small degrees of freedom and thus small sample sizes lead to t -distributions that are different than the z curve. On the other hand, large sample sizes correspond to large degrees of freedom and thus produce t distributions that closely align with the standard normal z -curve.

So, assuming the null hypothesis H_0 is true, our formula for the test statistic simplifies a bit:

$$t = \frac{(\bar{x}_a - \bar{x}_r) - 0}{\sqrt{\frac{s_a^2}{n_a} + \frac{s_r^2}{n_r}}} = \frac{\bar{x}_a - \bar{x}_r}{\sqrt{\frac{s_a^2}{n_a} + \frac{s_r^2}{n_r}}}$$

Let's compute the values necessary for this two-sample t -statistic. Recall the summary statistics we computed during our exploratory data analysis in Section 9.5.1.

```
movies_sample |>
  group_by(genre) |>
  summarize(n = n(), mean_rating = mean(rating), std_dev = sd(rating))
```

```
# A tibble: 2 x 4
  genre      n mean_rating std_dev
  <chr>    <int>     <dbl>    <dbl>
1 Action      32     5.275   1.36121
2 Romance     36     6.32222  1.60963
```

Using these values, the observed two-sample t -test statistic is

$$\frac{\bar{x}_a - \bar{x}_r}{\sqrt{\frac{s_a^2}{n_a} + \frac{s_r^2}{n_r}}} = \frac{5.28 - 6.32}{\sqrt{\frac{1.36^2}{32} + \frac{1.61^2}{36}}} = -2.906$$

Great! How can we compute the p -value using this theory-based test statistic? We need to compare it to a null distribution, which we construct next.

Null distribution

Let's revisit the null distribution for the test statistic $\bar{x}_a - \bar{x}_r$ we constructed in Section 9.5. Let's visualize this in the left-hand plot of Figure 9.17.

```
# Construct null distribution of xbar_a - xbar_r:
null_distribution_movies <- movies_sample |>
  specify(formula = rating ~ genre) |>
  hypothesize(null = "independence") |>
  generate(reps = 1000, type = "permute") |>
  calculate(stat = "diff in means", order = c("Action", "Romance"))
visualize(null_distribution_movies, bins = 10)
```

The `infer` package also includes some built-in theory-based test statistics as well. So instead of calculating the test statistic of interest as the "diff in means" $\bar{x}_a - \bar{x}_r$, we can calculate this defined two-sample t -statistic by setting `stat = "t"`. Let's visualize this in the right-hand plot of Figure 9.17.

```
# Construct null distribution of t:
null_distribution_movies_t <- movies_sample |>
  specify(formula = rating ~ genre) |>
  hypothesize(null = "independence") |>
  generate(reps = 1000, type = "permute") |>
# Notice we switched stat from "diff in means" to "t"
  calculate(stat = "t", order = c("Action", "Romance"))
  visualize(null_distribution_movies_t, bins = 10)
```

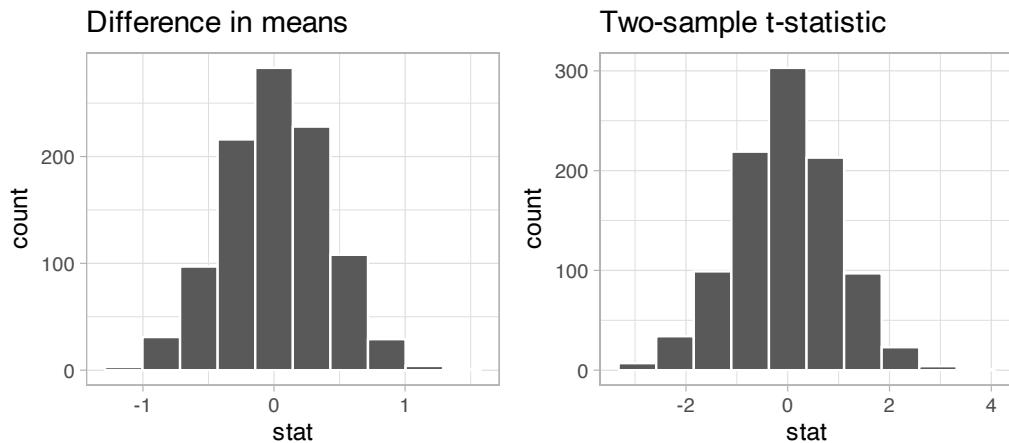


FIGURE 9.17: Comparing the null distributions of two test statistics.

Observe that while the shape of the null distributions of both the difference in means $\bar{x}_a - \bar{x}_r$ and the two-sample t -statistics are similar, the scales on the x-axis are different. The two-sample t -statistic values are spread out over a larger range.

However, a traditional theory-based t -test doesn't look at the simulated histogram in `null_distribution_movies_t`, but instead it looks at the t -distribution curve with degrees of freedom equal to roughly 65.85. This calculation is based on the complicated formula referenced previously, which we approximated with $df = n_a + n_r - 2 = 32 + 36 - 2 = 66$. Let's overlay this t -distribution curve over the top of our simulated two-sample t -statistics using the `method = "both"` argument in `visualize()`.

```
visualize(null_distribution_movies_t, bins = 10, method = "both")
```

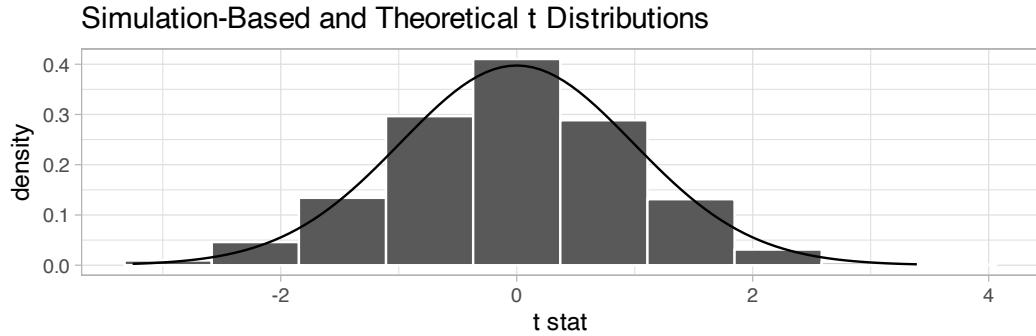


FIGURE 9.18: Null distribution using t-statistic and t-distribution.

Observe that the curve does a good job of approximating the histogram here. To calculate the p -value in this case, we need to figure out how much of the total area under the t -distribution curve is at or “more extreme” than our observed two-sample t -statistic. Since $H_A : \mu_a - \mu_r \neq 0$ is a two-sided alternative, we need to add up the areas in both tails.

We first compute the observed two-sample t -statistic using `infer` verbs. This shortcut calculation further assumes that the null hypothesis is true: that the population of action and romance movies have an equal average rating.

```
obs_two_sample_t <- movies_sample |>
  specify(formula = rating ~ genre) |>
  calculate(stat = "t", order = c("Action", "Romance"))
obs_two_sample_t
```

```
Response: rating (numeric)
Explanatory: genre (factor)
# A tibble: 1 x 1
  stat
  <dbl>
1 -2.90589
```

We want to find the percentage of values that are at or below `obs_two_sample_t = -2.906` or at or above `-obs_two_sample_t = 2.906`. We use the `shade_p_value()` function with the `direction` argument set to "both" to do this:

```
visualize(null_distribution_movies_t, method = "both") +
  shade_p_value(obs_stat = obs_two_sample_t, direction = "both")
```

Warning: Check to make sure the conditions have been met for the theoretical method. `infer` currently does not check these for you.

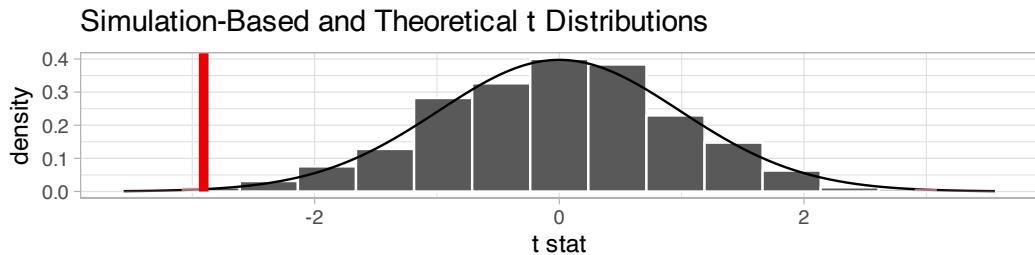


FIGURE 9.19: Null distribution using t -statistic and t -distribution with p -value shaded.

(We'll discuss this warning message shortly.) What is the p -value? We apply `get_p_value()` to our null distribution saved in `null_distribution_movies_t`:

```
null_distribution_movies_t |>
  get_p_value(obs_stat = obs_two_sample_t, direction = "both")
```

```
# A tibble: 1 x 1
  p_value
  <dbl>
1 0.002
```

We have a very small p -value, and thus it is very unlikely that these results are due to *sampling variation*. Thus, we are inclined to reject H_0 .

Let's come back to that earlier warning message: Check to make sure the conditions have been met for the theoretical method. `{infer}` currently does not check these for you. To be able to use the t -test and other such theoretical methods, there are always a few conditions to check. The `infer` package does not automatically check these conditions, hence the warning message we received. These conditions are necessary so that the underlying mathematical theory holds. In order for the results of our two-sample t -test to be valid, three conditions must be met:

1. Nearly normal populations or large sample sizes. A general rule of thumb that works in many (but not all) situations is that the sample size n should be greater than 30.

2. Both samples are selected independently of each other.
3. All observations are independent from each other.

Let's see if these conditions hold for our `movies_sample` data:

1. This is met since $n_a = 32$ and $n_r = 36$ are both larger than 30, satisfying our rule of thumb.
2. This is met since we sampled the action and romance movies at random and in an unbiased fashion from the database of all IMDb movies.
3. Unfortunately, we don't know how IMDb computes the ratings. For example, if the same person rated multiple movies, then those observations would be related and hence not independent.

Assuming all three conditions are roughly met, we can be reasonably certain that the theory-based t -test results are valid. If any of the conditions were clearly not met, we couldn't put as much trust into any conclusions reached. On the other hand, in most scenarios, the only assumption that needs to be met in the simulation-based method is that the sample is selected at random. Thus, in our experience, we prefer simulation-based methods as they have fewer assumptions, are conceptually easier to understand, and since computing power has recently become easily accessible, they can be run quickly. That being said since much of the world's research still relies on traditional theory-based methods, we also believe it is important to understand them.

You may be wondering why we chose `reps = 1000` for these simulation-based methods. We've noticed that after around 1000 replicates for the null distribution and the bootstrap distribution for most problems you can start to get a general sense for how the statistic behaves. You can change this value to something like 10,000 though for `reps` if you would like even finer detail but this will take more time to compute. Feel free to iterate on this as you like to get an even better idea about the shape of the null and bootstrap distributions as you wish.

9.6.2 When inference is not needed

We've now walked through several different examples of how to use the `infer` package to perform statistical inference: constructing confidence intervals and conducting hypothesis tests. For each of these examples, we made it a point to always perform an exploratory data analysis (EDA) first; specifically, by looking at the raw data values, by using data visualization with `ggplot2`, and by data wrangling with `dplyr` beforehand. We *highly* encourage you to always do the same. As a beginner to statistics, EDA helps you develop intuition as to what statistical methods like confidence intervals and hypothesis tests can tell us. Even as a seasoned practitioner of statistics, EDA helps guide your statistical investigations. In particular, is statistical inference even needed?

Let's consider an example. Say we're interested in the following question: Of *all* flights leaving a New York City airport, are Hawaiian Airlines flights in the air for longer than Alaska Airlines flights? Furthermore, let's assume that 2023 flights are a representative sample of all such flights. Then we can use the `flights` data frame in the `nycflights23` package we introduced in Section 1.4 to answer our question. Let's filter this data frame to only include Hawaiian and Alaska Airlines using their `carrier` codes `HA` and `AS`:

```
flights_sample <- flights |>
  filter(carrier %in% c("HA", "AS"))
```

There are two possible statistical inference methods we could use to answer such questions. First, we could construct a 95% confidence interval for the difference in population means $\mu_{HA} - \mu_{AS}$, where μ_{HA} is the mean air time of all Hawaiian Airlines flights and μ_{AS} is the mean air time of all Alaska Airlines flights. We could then check if the entirety of the interval is greater than 0, suggesting that $\mu_{HA} - \mu_{AS} > 0$, or, in other words suggesting that $\mu_{HA} > \mu_{AS}$. Second, we could perform a hypothesis test of the null hypothesis $H_0 : \mu_{HA} - \mu_{AS} = 0$ versus the alternative hypothesis $H_A : \mu_{HA} - \mu_{AS} > 0$.

However, let's first construct an exploratory visualization as we suggested earlier. Since `air_time` is numerical and `carrier` is categorical, a boxplot can display the relationship between these two variables, which we display in Figure 9.20.

```
ggplot(data = flights_sample, mapping = aes(x = carrier, y = air_time)) +
  geom_boxplot() +
  labs(x = "Carrier", y = "Air Time")
```

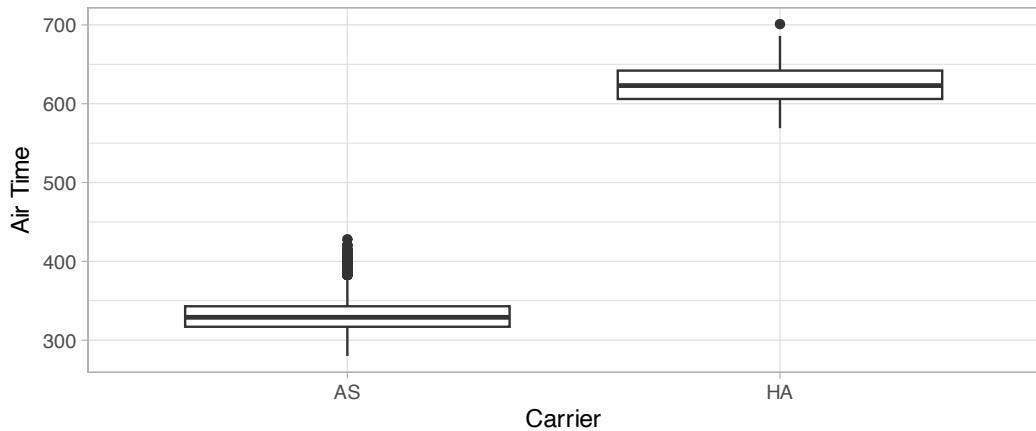


FIGURE 9.20: Air time for Hawaiian and Alaska Airlines flights departing NYC in 2023.

This is what we like to call “no PhD in Statistics needed” moments. You don’t have to be an expert in statistics to know that Alaska Airlines and Hawaiian Airlines have *significantly* different air times. The two boxplots don’t even overlap! Constructing a confidence interval or conducting a hypothesis test would frankly not provide much more insight than Figure 9.20.

Let’s investigate why we observe such a clear cut difference between these two airlines using data wrangling. Let’s first group by the rows of `flights_sample` not only by `carrier` but also by destination `dest`. Subsequently, we’ll compute two summary statistics: the number of observations using `n()` and the mean airtime:

```
flights_sample |>
  group_by(carrier, dest) |>
  summarize(n = n(), mean_time = mean(air_time, na.rm = TRUE))
```

```
'summarise()' has grouped output by 'carrier'. You can override using the
`.groups` argument.
```

```
# A tibble: 8 x 4
# Groups:   carrier [2]
  carrier dest      n  mean_time
  <chr>   <chr> <int>     <dbl>
1 AS       LAS      1    299
2 AS       LAX     980    323.929
3 AS       PDX     710    326.383
4 AS       PSP      18    309.611
5 AS       SAN    1034    325.457
6 AS       SEA    2417    324.787
7 AS       SFO    2683    343.542
8 HA       HNL     366    623.287
```

It turns out that from New York City in 2023, Alaska flew to seven different airports on the West Coast region of the US while Hawaiian only flew to `HNL` (Honolulu) from NYC. Given the clear difference in distance from New York City to the West Coast versus New York City to Honolulu, it is not surprising that we observe such different (*statistically significantly different*, in fact) air times in flights.

This is a clear example of not needing to do anything more than a simple exploratory data analysis using data visualization and descriptive statistics to get an appropriate conclusion. This is why we highly recommend you perform an EDA of any sample data before running statistical inference methods like confidence intervals and hypothesis tests.

9.6.3 Problems with p-values

On top of the many common misunderstandings about hypothesis testing and p -values we listed in Section 9.4, another unfortunate consequence of the expanded use of p -values and hypothesis testing is a phenomenon known as “p-hacking.” p-hacking is the act of “cherry-picking” only results that are “statistically significant” while dismissing those that aren’t, even if at the expense of the scientific ideas. There are lots of articles written recently about misunderstandings and the problems with p -values. We encourage you to check some of them out:

1. Misuse of p -values⁴
2. What a nerdy debate about p -values shows about science - and how to fix it⁵
3. Statisticians issue warning over misuse of P values⁶
4. You Can’t Trust What You Read About Nutrition⁷
5. A Litany of Problems with p -values⁸

Such issues were getting so problematic that the American Statistical Association (ASA) put out a statement in 2016 titled, “The ASA Statement on Statistical Significance and P -Values,”⁹ with six principles underlying the proper use and interpretation of p -values. The ASA released this guidance on p -values to improve the conduct and interpretation of quantitative science and to inform the growing emphasis on reproducibility of science research.

We as authors much prefer the use of confidence intervals for statistical inference, since in our opinion they are much less prone to large misinterpretation. However, many fields still exclusively use p -values for statistical inference and this is one reason for including them in this text. We encourage you to learn more about “p-hacking” as well and its implication for science.

9.6.4 Additional resources

Solutions to all *Learning checks* can be found online in Appendix D¹⁰.

An R script file of all R code used in this chapter is available at <https://www.moderndive.com/scripts/09-hypothesis-testing.R>.

If you want more examples of the `infer` workflow for conducting hypothesis tests, we suggest you check out the `infer` package homepage, in particular, a series of example analyses available at <https://infer.netlify.app/articles/>.

⁴https://en.wikipedia.org/wiki/Misuse_of_p-values

⁵<https://www.vox.com/science-and-health/2017/7/31/16021654/p-values-statistical-significance-redefine-0005>

⁶<https://www.nature.com/news/statisticians-issue-warning-over-misuse-of-p-values-1.19503>

⁷<https://fivethirtyeight.com/features/you-can-t-trust-what-you-read-about-nutrition/>

⁸<http://www.flharrell.com/post/pval-litany/>

⁹<https://www.amstat.org/asa/files/pdfs/P-ValueStatement.pdf>

¹⁰<https://moderndive.com/D-appendixD.html>

9.6.5 What's to come

We conclude with the `infer` pipeline for hypothesis testing in Figure 9.21.

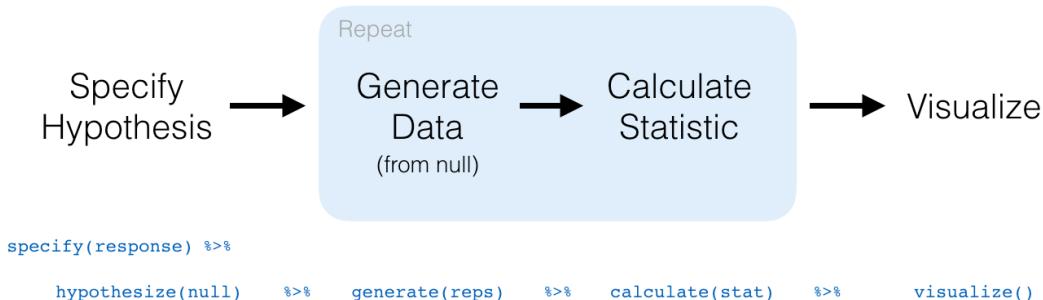


FIGURE 9.21: `infer` package workflow for hypothesis testing.

Now that we've armed ourselves with an understanding of confidence intervals from Chapter 8 and hypothesis tests from this chapter, we'll now study inference for regression in the upcoming Chapter 10.

10

Inference for Regression

In our penultimate chapter, we'll revisit the regression models we first studied in Chapters 5 and 6. Armed with our knowledge of confidence intervals and hypothesis tests from Chapters 8 and 9, we'll be able to apply statistical inference to further our understanding of relationships between outcome and explanatory variables.

Needed packages

Let's load all the packages needed for this chapter (this assumes you've already installed them). Recall from our discussion in Section 4.4 that loading the `tidyverse` package by running `library(tidyverse)` loads the following commonly used data science packages all at once:

- `ggplot2` for data visualization
- `dplyr` for data wrangling
- `tidyverse` for converting data to “tidy” format
- `readr` for importing spreadsheet data into R
- As well as the more advanced `purrr`, `tibble`, `stringr`, and `forcats` packages

If needed, read Section 1.3 for information on how to install and load R packages.

```
library(tidyverse)
library(moderndive)
library(infer)
```

10.1 Regression refresher

Before jumping into inference for regression, let's remind ourselves of the University of Texas Austin teaching evaluations analysis in Section 5.1.

10.1.1 Teaching evaluations analysis

Recall using simple linear regression we modeled the relationship between

1. A numerical outcome variable y (the instructor's teaching score) and
2. A single numerical explanatory variable x (the instructor's "beauty" score).

We first created an `evals_ch5` data frame that selected a subset of variables from the `evals` data frame included in the `moderndive` package. This `evals_ch5` data frame contains only the variables of interest for our analysis, in particular the instructor's teaching score and the "beauty" rating `bty_avg`:

```
evals_ch5 <- evals %>%
  select(ID, score, bty_avg, age)
glimpse(evals_ch5)
```

```
Rows: 463
Columns: 4
$ ID      <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17~
$ score   <dbl> 4.7, 4.1, 3.9, 4.8, 4.6, 4.3, 2.8, 4.1, 3.4, 4.5, 3.8, 4.~
$ bty_avg <dbl> 5.00, 5.00, 5.00, 5.00, 3.00, 3.00, 3.00, 3.33, 3.33, 3.1~
$ age     <int> 36, 36, 36, 36, 59, 59, 59, 51, 51, 40, 40, 40, 40, 4~
```

In Subsection 5.1.1, we performed an exploratory data analysis of the relationship between these two variables of `score` and `bty_avg`. We saw there that a weakly positive correlation of 0.187 existed between the two variables.

This was evidenced in Figure 10.1 of the scatterplot along with the "best-fitting" regression line that summarizes the linear relationship between the two variables of `score` and `bty_avg`. Recall in Subsection 5.3.2 that we defined a "best-fitting" line as the line that minimizes the *sum of squared residuals*.

```
ggplot(evals_ch5,
       aes(x = bty_avg, y = score)) +
  geom_point() +
  labs(x = "Beauty Score",
       y = "Teaching Score",
       title = "Relationship between teaching and beauty scores") +
  geom_smooth(method = "lm", se = FALSE)
```

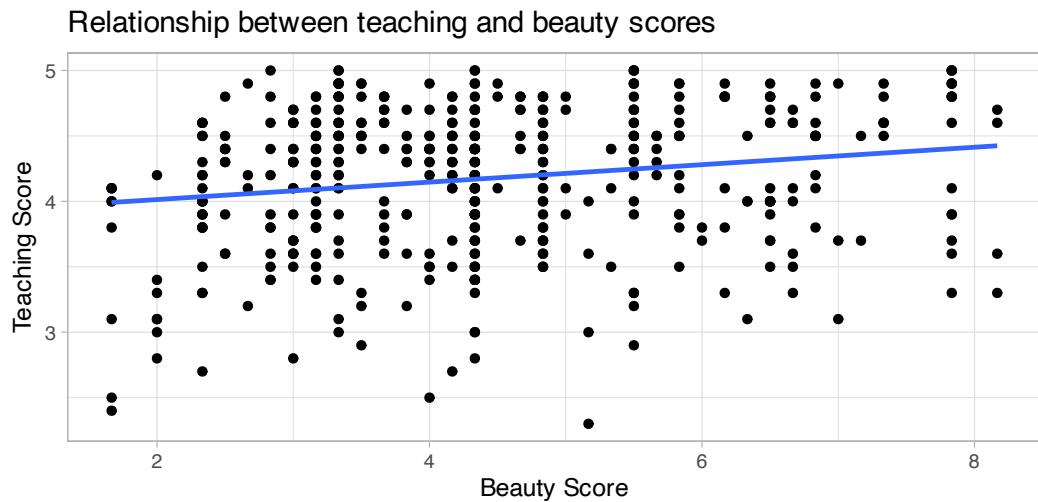


FIGURE 10.1: Relationship with regression line.

Looking at this plot again, you might be asking, “Does that line really have all that positive of a slope?”. It does increase from left to right as the `bty_avg` variable increases, but by how much? To get to this information, recall that we followed a two-step procedure:

1. We first “fit” the linear regression model using the `lm()` function with the formula `score ~ bty_avg`. We saved this model in `score_model`.
2. We get the regression table by applying the `get_regression_table()` function from the `moderndive` package to `score_model`.

```
# Fit regression model:
score_model <- lm(score ~ bty_avg, data = evals_ch5)
# Get regression table:
get_regression_table(score_model)
```

TABLE 10.1: Previously seen linear regression table

term	estimate	std_error	statistic	p_value	lower_ci	upper_ci
intercept	3.880	0.076	50.96	0	3.731	4.030
bty_avg	0.067	0.016	4.09	0	0.035	0.099

Using the values in the `estimate` column of the resulting regression table in Table 10.1, we could then obtain the equation of the “best-fitting” regression line in Figure 10.1:

$$\begin{aligned}\hat{y} &= b_0 + b_1 \cdot x \\ \widehat{\text{score}} &= b_0 + b_{\text{bty_avg}} \cdot \text{bty_avg} \\ &= 3.880 + 0.067 \cdot \text{bty_avg}\end{aligned}$$

where b_0 is the fitted intercept and b_1 is the fitted slope for `bty_avg`. Recall the interpretation of the $b_1 = 0.067$ value of the fitted slope:

For every increase of one unit in “beauty” rating, there is an associated increase, on average, of 0.067 units of evaluation score.

Thus, the slope value quantifies the relationship between the y variable `score` and the x variable `bty_avg`. We also discussed the intercept value of $b_0 = 3.88$ and its lack of practical interpretation, since the range of possible “beauty” scores does not include 0.

10.1.2 Sampling scenario

Let’s now revisit this study in terms of the terminology and notation related to sampling we studied in Subsection 7.3.1.

First, let’s view the instructors for these 463 courses as a *representative sample* from a greater *study population*. In our case, let’s assume that the study population is *all* instructors at UT Austin and that the sample of instructors who taught these 463 courses is a representative sample. Unfortunately, we can only *assume* these two facts without more knowledge of the *sampling methodology* used by the researchers.

Since we are viewing these $n = 463$ courses as a sample, we can view our fitted slope $b_1 = 0.067$ as a *point estimate* of the *population slope* β_1 . In other words, β_1 quantifies the relationship between teaching `score` and “beauty” average `bty_avg` for *all* instructors at UT Austin. Similarly, we can view our fitted intercept $b_0 = 3.88$ as a *point estimate* of the *population intercept* β_0 for *all* instructors at UT Austin.

Putting these two ideas together, we can view the equation of the fitted line $\hat{y} = b_0 + b_1 \cdot x = 3.880 + 0.067 \cdot \text{bty_avg}$ as an estimate of some true and unknown *population line* $y = \beta_0 + \beta_1 \cdot x$. Thus we can draw parallels between our teaching evaluations analysis and all the sampling scenarios we’ve seen previously. In this chapter, we’ll focus on the final scenario of regression slopes as shown in Table 10.2.

Since we are now viewing our fitted slope b_1 and fitted intercept b_0 as *point estimates* based on a *sample*, these estimates will again be subject to *sampling variability*. In other words, if we collected a new sample of data on a different set of $n = 463$ courses and their instructors, the new fitted slope b_1 will likely differ from 0.067. The same

TABLE 10.2: Scenarios of sampling for inference

Scenario	Population parameter	Notation	Point estimate	Symbol(s)
1	Population proportion	p	Sample proportion	\hat{p}
2	Population mean	μ	Sample mean	\bar{x} or $\hat{\mu}$
3	Difference in population proportions	$p_1 - p_2$	Difference in sample proportions	$\hat{p}_1 - \hat{p}_2$
4	Difference in population means	$\mu_1 - \mu_2$	Difference in sample means	$\bar{x}_1 - \bar{x}_2$ or $\hat{\mu}_1 - \hat{\mu}_2$
5	Population regression slope	β_1	Fitted regression slope	b_1 or $\hat{\beta}_1$

goes for the new fitted intercept b_0 . But by how much will these estimates *vary*? This information is in the remaining columns of the regression table in Table 10.1. Our knowledge of sampling from Chapter 7, confidence intervals from Chapter 8, and hypothesis tests from Chapter 9 will help us interpret these remaining columns.

10.2 Interpreting regression tables

We've so far focused only on the two leftmost columns of the regression table in Table 10.1: `term` and `estimate`. Let's now shift our attention to the remaining columns: `std_error`, `statistic`, `p_value`, `lower_ci` and `upper_ci` in Table 10.3.

TABLE 10.3: Previously seen regression table

term	estimate	std_error	statistic	p_value	lower_ci	upper_ci
intercept	3.880	0.076	50.96	0	3.731	4.030
bty_avg	0.067	0.016	4.09	0	0.035	0.099

Given the lack of practical interpretation for the fitted intercept b_0 , in this section we'll focus only on the second row of the table corresponding to the fitted slope b_1 . We'll first interpret the `std_error`, `statistic`, `p_value`, `lower_ci` and `upper_ci` columns. Afterwards in the upcoming Subsection 10.2.5, we'll discuss how R computes these values.

10.2.1 Standard error

The third column of the regression table in Table 10.1 `std_error` corresponds to the *standard error* of our estimates. Recall the definition of **standard error** we saw in Subsection 7.3.2:

The *standard error* is the standard deviation of any point estimate computed from a sample.

So what does this mean in terms of the fitted slope $b_1 = 0.067$? This value is just one possible value of the fitted slope resulting from *this particular sample* of $n = 463$ pairs of teaching and “beauty” scores. However, if we collected a different sample of $n = 463$ pairs of teaching and “beauty” scores, we will almost certainly obtain a different fitted slope b_1 . This is due to *sampling variability*.

Say we hypothetically collected 1000 such samples of pairs of teaching and “beauty” scores, computed the 1000 resulting values of the fitted slope b_1 , and visualized them in a histogram. This would be a visualization of the *sampling distribution* of b_1 , which we defined in Subsection 7.3.2. Further recall that the standard deviation of the *sampling distribution* of b_1 has a special name: the *standard error*.

Recall that we constructed three sampling distributions for the sample proportion \hat{p} using shovels of size 25, 50, and 100 in Figure 7.12. We observed that as the sample size increased, the standard error decreased as evidenced by the narrowing sampling distribution.

The *standard error* of b_1 similarly quantifies how much variation in the fitted slope b_1 one would expect between different samples. So in our case, we can expect about 0.016 units of variation in the `bty_avg` slope variable. Recall that the `estimate` and `std_error` values play a key role in *inferring* the value of the unknown population slope β_1 relating to *all* instructors.

In Section 10.4, we’ll perform a simulation using the `infer` package to construct the bootstrap distribution for b_1 in this case. Recall from Subsection 8.7.1 that the bootstrap distribution is an *approximation* to the sampling distribution in that they have a similar shape. Since they have a similar shape, they have similar *standard errors*. However, unlike the sampling distribution, the bootstrap distribution is constructed from a *single* sample, which is a practice more aligned with what’s done in real life.

10.2.2 Test statistic

The fourth column of the regression table in Table 10.1 `statistic` corresponds to a *test statistic* relating to the following *hypothesis test*:

$$H_0 : \beta_1 = 0 \\ \text{vs } H_A : \beta_1 \neq 0.$$

Recall our terminology, notation, and definitions related to hypothesis tests we introduced in Section 9.2.

A *hypothesis test* consists of a test between two competing hypotheses: (1) a *null hypothesis* H_0 versus (2) an *alternative hypothesis* H_A .

A *test statistic* is a point estimate/sample statistic formula used for hypothesis testing.

Here, our *null hypothesis* H_0 assumes that the population slope β_1 is 0. If the population slope β_1 is truly 0, then this is saying that there is *no true relationship* between teaching and “beauty” scores for *all* the instructors in our population. In other words, x = “beauty” score would have no associated effect on y = teaching score. The *alternative hypothesis* H_A , on the other hand, assumes that the population slope β_1 is not 0, meaning it could be either positive or negative. This suggests either a positive or negative relationship between teaching and “beauty” scores. Recall we called such alternative hypotheses *two-sided*. By convention, all hypothesis testing for regression assumes two-sided alternatives.

Recall our “hypothesized universe” of no gender discrimination we *assumed* in our *promotions* activity in Section 9.1. Similarly here when conducting this hypothesis test, we’ll assume a “hypothesized universe” where there is no relationship between teaching and “beauty” scores. In other words, we’ll assume the null hypothesis $H_0 : \beta_1 = 0$ is true.

The *statistic* column in the regression table is a tricky one, however. It corresponds to a standardized *t-test statistic*, much like the *two-sample t statistic* we saw in Subsection 9.6.1 where we used a theory-based method for conducting hypothesis tests. In both these cases, the *null distribution* can be mathematically proven to be a *t-distribution*. Since such test statistics are tricky for individuals new to statistical inference to study, we’ll skip this and jump into interpreting the *p-value*. If you’re curious, we have included a discussion of this standardized *t-test statistic* in Subsection 10.5.1.

10.2.3 p-value

The fifth column of the regression table in Table 10.1 *p_value* corresponds to the *p-value* of the hypothesis test $H_0 : \beta_1 = 0$ versus $H_A : \beta_1 \neq 0$.

Again recalling our terminology, notation, and definitions related to hypothesis tests we introduced in Section 9.2, let's focus on the definition of the *p*-value:

A *p*-value is the probability of obtaining a test statistic just as extreme or more extreme than the observed test statistic *assuming the null hypothesis H_0 is true*.

Recall that you can intuitively think of the *p*-value as quantifying how “extreme” the observed fitted slope of $b_1 = 0.067$ is in a “hypothesized universe” where there is no relationship between teaching and “beauty” scores.

Following the hypothesis testing procedure we outlined in Section 9.4, since the *p*-value in this case is 0, for any choice of significance level α we would reject H_0 in favor of H_A . Using non-statistical language, this is saying: we reject the hypothesis that there is no relationship between teaching and “beauty” scores in favor of the hypothesis that there is. That is to say, the evidence suggests there is a significant relationship, one that is positive.

More precisely, however, the *p*-value corresponds to how extreme the observed test statistic of 4.09 is when compared to the appropriate *null distribution*. In Section 10.4, we'll perform a simulation using the `infer` package to construct the null distribution in this case.

An extra caveat here is that the results of this hypothesis test are only valid if certain “conditions for inference for regression” are met, which we'll introduce shortly in Section 10.3.

10.2.4 Confidence interval

The two rightmost columns of the regression table in Table 10.1 (`lower_ci` and `upper_ci`) correspond to the endpoints of the 95% *confidence interval* for the population slope β_1 . Recall our analogy of “nets are to fish” what “confidence intervals are to population parameters” from Section 8.3. The resulting 95% confidence interval for β_1 of (0.035, 0.099) can be thought of as a range of plausible values for the population slope β_1 of the linear relationship between teaching and “beauty” scores.

As we introduced in Subsection 8.5.2 on the precise and shorthand interpretation of confidence intervals, the statistically precise interpretation of this confidence interval is: “if we repeated this sampling procedure a large number of times, we expect about 95% of the resulting confidence intervals to capture the value of the population slope β_1 .” However, we'll summarize this using our shorthand interpretation that “we're 95% ‘confident’ that the true population slope β_1 lies between 0.035 and 0.099.”

Notice in this case that the resulting 95% confidence interval for β_1 of (0.035, 0.099) does not contain a very particular value: β_1 equals 0. Recall we mentioned that if the population regression slope β_1 is 0, this is equivalent to saying there is *no* relationship between teaching and “beauty” scores. Since $\beta_1 = 0$ is not in our plausible range of values for β_1 , we are inclined to believe that there, in fact, *is* a relationship between teaching and “beauty” scores and a positive one at that. So in this case, the conclusion about the population slope β_1 from the 95% confidence interval matches the conclusion from the hypothesis test: evidence suggests that there is a meaningful relationship between teaching and “beauty” scores.

Recall from Subsection 8.5.3, however, that the *confidence level* is one of many factors that determine confidence interval widths. So for example, say we used a higher confidence level of 99% instead of 95%. The resulting confidence interval for β_1 would be wider and thus might now include 0. The lesson to remember here is that any confidence-interval-based conclusion depends highly on the confidence level used.

What are the calculations that went into computing the two endpoints of the 95% confidence interval for β_1 ?

Recall our sampling bowl example from Subsection 8.7.2 discussing `lower_ci` and `upper_ci`. Since the sampling and bootstrap distributions of the sample proportion \hat{p} were roughly normal, we could use the rule of thumb for bell-shaped distributions from Appendix A.2 to create a 95% confidence interval for p with the following equation:

$$\hat{p} \pm \text{MoE}_{\hat{p}} = \hat{p} \pm 1.96 \cdot \text{SE}_{\hat{p}} = \hat{p} \pm 1.96 \cdot \sqrt{\frac{\hat{p}(1 - \hat{p})}{n}}$$

We can generalize this to other point estimates that have roughly normally shaped sampling and/or bootstrap distributions:

$$\text{point estimate} \pm \text{MoE} = \text{point estimate} \pm 1.96 \cdot \text{SE}.$$

We'll show in Section 10.4 that the sampling/bootstrap distribution for the fitted slope b_1 is in fact bell-shaped as well. Thus we can construct a 95% confidence interval for β_1 with the following equation:

$$b_1 \pm \text{MoE}_{b_1} = b_1 \pm 1.96 \cdot \text{SE}_{b_1}.$$

What is the value of the standard error SE_{b_1} ? It is in fact in the third column of the regression table in Table 10.1: 0.016. Thus

$$\begin{aligned} b_1 \pm 1.96 \cdot \text{SE}_{b_1} &= 0.067 \pm 1.96 \cdot 0.016 = 0.067 \pm 0.031 \\ &= (0.036, 0.098) \end{aligned}$$

This closely matches the (0.035, 0.099) confidence interval in the last two columns of Table 10.1.

Much like hypothesis tests, however, the results of this confidence interval also are only valid if the “conditions for inference for regression” to be discussed in Section 10.3 are met.

10.2.5 How does R compute the table?

Since we didn’t perform the simulation to get the values of the standard error, test statistic, p -value, and endpoints of the 95% confidence interval in Table 10.1, you might be wondering how were these values computed. What did R do behind the scenes? Does R run simulations like we did using the `infer` package in Chapters 8 and 9 on confidence intervals and hypothesis testing?

The answer is no! Much like the theory-based method for constructing confidence intervals you saw in Subsection 8.7.2 and the theory-based hypothesis test you saw in Subsection 9.6.1, there exist mathematical formulas that allow you to construct confidence intervals and conduct hypothesis tests for inference for regression. These formulas were derived in a time when computers didn’t exist, so it would’ve been impossible to run the extensive computer simulations we have in this book. We present these formulas in Subsection 10.5.1 on “theory-based inference for regression.”

In Section 10.4, we’ll go over a simulation-based approach to constructing confidence intervals and conducting hypothesis tests using the `infer` package. In particular, we’ll convince you that the bootstrap distribution of the fitted slope b_1 is indeed bell-shaped.

10.3 Conditions for inference for regression

Recall in Subsection 8.3.2 we stated that we could only use the standard-error-based method for constructing confidence intervals if the bootstrap distribution was bell shaped. Similarly, there are certain conditions that need to be met in order for the results of our hypothesis tests and confidence intervals we described in Section 10.2 to have valid meaning. These conditions must be met for the assumed underlying mathematical and probability theory to hold true.

For inference for regression, there are four conditions that need to be met. Note the first four letters of these conditions are highlighted in bold in what follows: **LINE**. This can serve as a nice reminder of what to check for whenever you perform linear regression.

1. Linearity of relationship between variables
2. Independence of the residuals
3. Normality of the residuals
4. Equality of variance of the residuals

Conditions **L**, **N**, and **E** can be verified through what is known as a *residual analysis*. Condition **I** can only be verified through an understanding of how the data was collected.

In this section, we'll go over a refresher on residuals, verify whether each of the four **LINE** conditions hold true, and then discuss the implications.

10.3.1 Residuals refresher

Recall our definition of a residual from Subsection 5.1.3: it is the *observed value* minus the *fitted value* denoted by $y - \hat{y}$. Recall that residuals can be thought of as the error or the “lack-of-fit” between the observed value y and the fitted value \hat{y} on the regression line in Figure 10.1. In Figure 10.2, we illustrate one particular residual out of 463 using an arrow, as well as its corresponding observed and fitted values using a circle and a square, respectively.

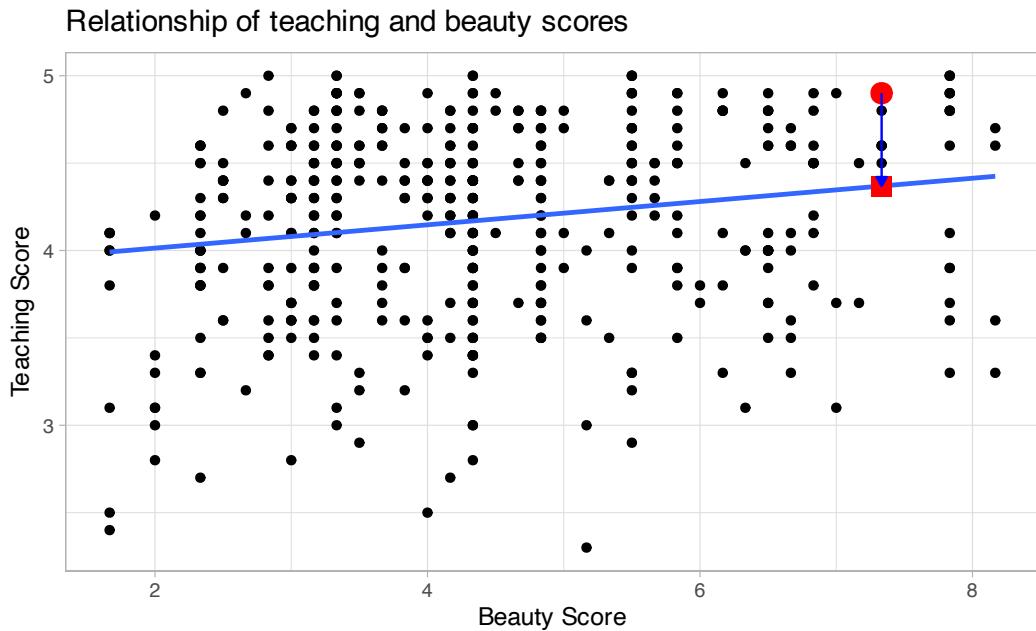


FIGURE 10.2: Example of observed value, fitted value, and residual.

Furthermore, we can automate the calculation of all $n = 463$ residuals by applying the `get_regression_points()` function to our saved regression model in `score_model`. Observe how the resulting values of `residual` are roughly equal to `score - score_hat` (there is potentially a slight difference due to rounding error).

```
# Fit regression model:
score_model <- lm(score ~ bty_avg, data = evals_ch5)
```

```
# Get regression points:
regression_points <- get_regression_points(score_model)
regression_points
```

	ID	score	bty_avg	score_hat	residual
	<int>	<dbl>	<dbl>	<dbl>	<dbl>
1	1	4.7	5	4.214	0.486
2	2	4.1	5	4.214	-0.114
3	3	3.9	5	4.214	-0.314
4	4	4.8	5	4.214	0.586
5	5	4.6	3	4.08	0.52
6	6	4.3	3	4.08	0.22
7	7	2.8	3	4.08	-1.28
8	8	4.1	3.333	4.102	-0.002
9	9	3.4	3.333	4.102	-0.702
10	10	4.5	3.167	4.091	0.409
# i 453 more rows					

A *residual analysis* is used to verify conditions **L**, **N**, and **E** and can be performed using appropriate data visualizations. While there are more sophisticated statistical approaches that can also be done, we'll focus on the much simpler approach of looking at plots.

10.3.2 Linearity of relationship

The first condition is that the relationship between the outcome variable y and the explanatory variable x must be Linear. Recall the scatterplot in Figure 10.1 where we had the explanatory variable x as “beauty” score and the outcome variable y as teaching score. Would you say that the relationship between x and y is linear? It's hard to say because of the scatter of the points about the line. In the authors' opinions, we feel this relationship is “linear enough.”

Let's present an example where the relationship between x and y is clearly not linear in Figure 10.3. In this case, the points clearly do not form a line, but rather a U-shaped polynomial curve. In this case, any results from an inference for regression would not be valid.

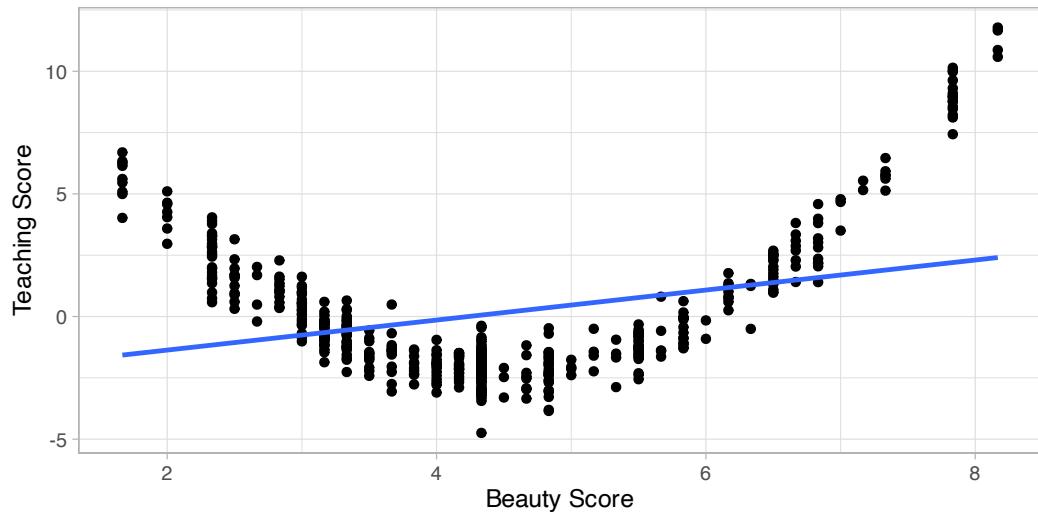


FIGURE 10.3: Example of a clearly non-linear relationship.

10.3.3 Independence of residuals

The second condition is that the residuals must be **Independent**. In other words, the different observations in our data must be independent of one another.

For our UT Austin data, while there is data on 463 courses, these 463 courses were actually taught by 94 unique instructors. In other words, the same professor is often included more than once in our data. The original `evals` data frame that we used to construct the `evals_ch5` data frame has a variable `prof_ID`, which is an anonymized identification variable for the professor:

```
evals %>%
  select(ID, prof_ID, score, bty_avg)
```

```
# A tibble: 463 x 4
  ID prof_ID score bty_avg
  <int>   <int> <dbl>   <dbl>
1     1       1   4.7    5
2     2       1   4.1    5
3     3       1   3.9    5
4     4       1   4.8    5
5     5       2   4.6    3
6     6       2   4.3    3
7     7       2   2.8    3
8     8       3   4.1  3.333
9     9       3   3.4  3.333
```

```
10      10      4    4.5   3.167
# i 453 more rows
```

For example, the professor with `prof_ID` equal to 1 taught the first 4 courses in the data, the professor with `prof_ID` equal to 2 taught the next 3, and so on. Given that the same professor taught these first four courses, it is reasonable to expect that these four teaching scores are related to each other. If a professor gets a high `score` in one class, chances are fairly good they'll get a high `score` in another. This dataset thus provides different information than if we had 463 unique instructors teaching the 463 courses.

In this case, we say there exists *dependence* between observations. The first four courses taught by professor 1 are dependent, the next 3 courses taught by professor 2 are related, and so on. Any proper analysis of this data needs to take into account that we have *repeated measures* for the same profs.

So in this case, the independence condition is not met. What does this mean for our analysis? We'll address this in Subsection 10.3.6 coming up, after we check the remaining two conditions.

10.3.4 Normality of residuals

The third condition is that the residuals should follow a **Normal** distribution. Furthermore, the center of this distribution should be 0. In other words, sometimes the regression model will make positive errors: $y - \hat{y} > 0$. Other times, the regression model will make equally negative errors: $y - \hat{y} < 0$. However, *on average* the errors should equal 0 and their shape should be similar to that of a bell.

The simplest way to check the normality of the residuals is to look at a histogram, which we visualize in Figure 10.4.

```
ggplot(regression_points, aes(x = residual)) +
  geom_histogram(binwidth = 0.25, color = "white") +
  labs(x = "Residual")
```

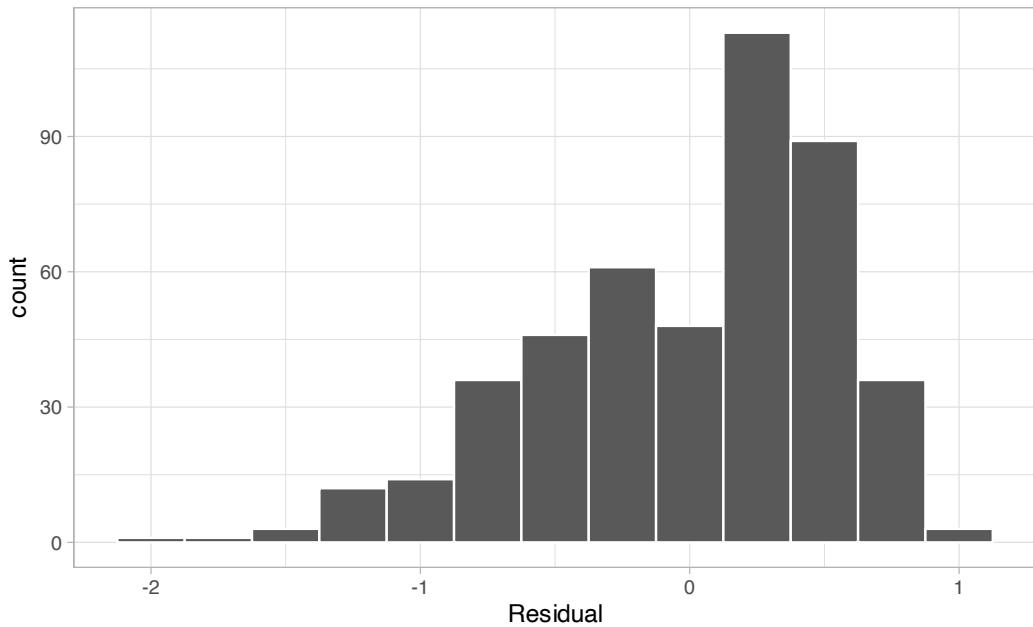


FIGURE 10.4: Histogram of residuals.

This histogram shows that we have more positive residuals than negative. Since the residual $y - \hat{y}$ is positive when $y > \hat{y}$, it seems our regression model's fitted teaching scores \hat{y} tend to *underestimate* the true teaching scores y . Furthermore, this histogram has a slight *left-skew* in that there is a tail on the left. This is another way to say the residuals exhibit a *negative skew*.

Is this a problem? Again, there is a certain amount of subjectivity in the response. In the authors' opinion, while there is a slight skew to the residuals, we feel it isn't drastic. On the other hand, others might disagree with our assessment.

Let's present examples where the residuals clearly do and don't follow a normal distribution in Figure 10.5. In this case of the model yielding the clearly non-normal residuals on the right, any results from an inference for regression would not be valid.

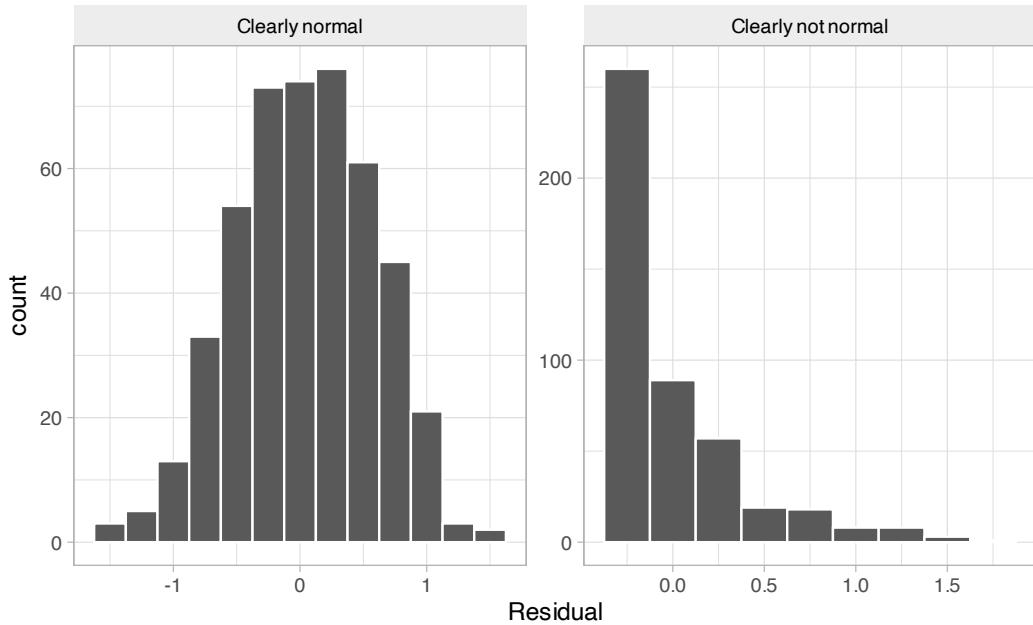


FIGURE 10.5: Example of clearly normal and clearly not normal residuals.

10.3.5 Equality of variance

The fourth and final condition is that the residuals should exhibit **Equal variance** across all values of the explanatory variable x . In other words, the value and spread of the residuals should not depend on the value of the explanatory variable x .

Recall the scatterplot in Figure 10.1: we had the explanatory variable x of “beauty” score on the x-axis and the outcome variable y of teaching score on the y-axis. Instead, let’s create a scatterplot that has the same values on the x-axis, but now with the residual $y - \hat{y}$ on the y-axis as seen in Figure 10.6.

```
ggplot(regression_points, aes(x = bty_avg, y = residual)) +
  geom_point() +
  labs(x = "Beauty Score", y = "Residual") +
  geom_hline(yintercept = 0, col = "blue", size = 1)
```

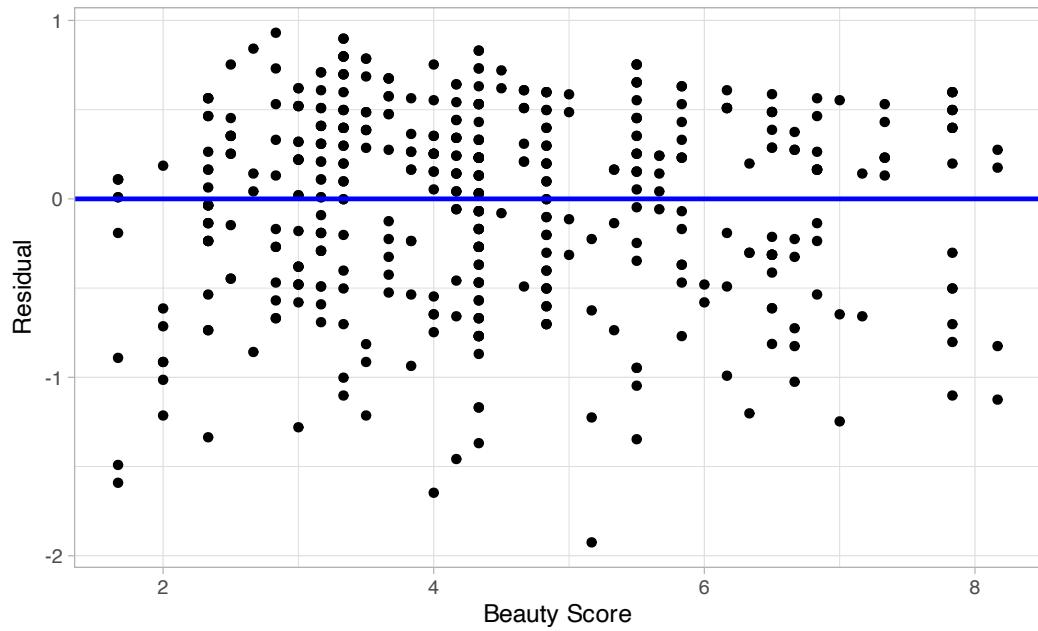


FIGURE 10.6: Plot of residuals over beauty score.

You can think of Figure 10.6 as a modified version of the plot with the regression line in Figure 10.1, but with the regression line flattened out to $y = 0$. Looking at this plot, would you say that the spread of the residuals around the line at $y = 0$ is constant across all values of the explanatory variable x of “beauty” score? This question is rather qualitative and subjective in nature, thus different people may respond with different answers. For example, some people might say that there is slightly more variation in the residuals for smaller values of x than for higher ones. However, it can be argued that there isn’t a *drastic* non-constancy.

In Figure 10.7 let’s present an example where the residuals clearly do not have equal variance across all values of the explanatory variable x .

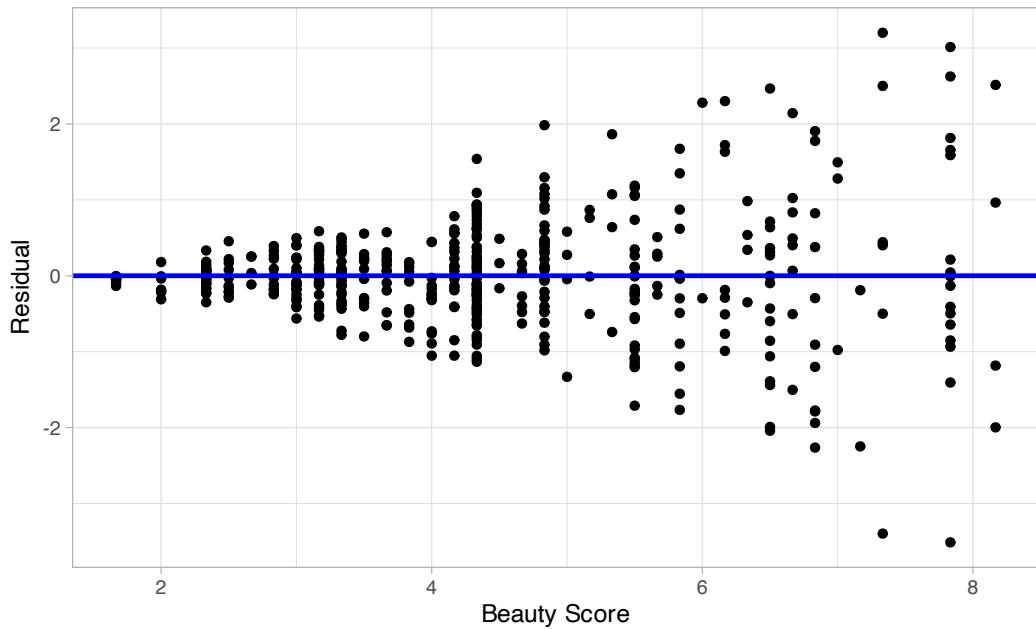


FIGURE 10.7: Example of clearly non-equal variance.

Observe how the spread of the residuals increases as the value of x increases. This is a situation known as *heteroskedasticity*. Any inference for regression based on a model yielding such a pattern in the residuals would not be valid.

10.3.6 What's the conclusion?

Let's list our four conditions for inference for regression again and indicate whether or not they were satisfied in our analysis:

1. Linearity of relationship between variables: Yes
2. Independence of residuals: No
3. Normality of residuals: Somewhat
4. Equality of variance: Yes

So what does this mean for the results of our confidence intervals and hypothesis tests in Section 10.2?

First, the Independence condition. The fact that there exist dependencies between different rows in `evals_ch5` must be addressed. In more advanced statistics courses, you'll learn how to incorporate such dependencies into your regression models. One such technique is called *hierarchical/multilevel modeling*.

Second, when conditions **L**, **N**, **E** are not met, it often means there is a shortcoming in our model. For example, it may be the case that using only a single explanatory

variable is insufficient, as we did with “beauty” score. We may need to incorporate more explanatory variables in a multiple regression model as we did in Chapter 6, or perhaps use a transformation of one or more of your variables, or use an entirely different modeling technique. To learn more about addressing such shortcomings, you’ll have to take a class on or read up on more advanced regression modeling methods.

In our case, the best we can do is view the results suggested by our confidence intervals and hypothesis tests as preliminary. While a preliminary analysis suggests that there is a significant relationship between teaching and “beauty” scores, further investigation is warranted; in particular, by improving the preliminary `score ~ bty_avg` model so that the four conditions are met. When the four conditions are roughly met, then we can put more faith into our confidence intervals and p -values.

The conditions for inference in regression problems are a key part of regression analysis that are of vital importance to the processes of constructing confidence intervals and conducting hypothesis tests. However, it is often the case with regression analysis in the real world that not all the conditions are completely met. Furthermore, as you saw, there is a level of subjectivity in the residual analyses to verify the **L**, **N**, and **E** conditions. So what can you do? We as authors advocate for transparency in communicating all results. This lets the stakeholders of any analysis know about a model’s shortcomings or whether the model is “good enough.” So while this checking of assumptions has lead to some fuzzy “it depends” results, we decided as authors to show you these scenarios to help prepare you for difficult statistical decisions you may need to make down the road.

Learning check

(LC10.1) Continuing with our regression using `age` as the explanatory variable and `teaching score` as the outcome variable.

- Use the `get_regression_points()` function to get the observed values, fitted values, and residuals for all 463 instructors.
- Perform a residual analysis and look for any systematic patterns in the residuals. Ideally, there should be little to no pattern but comment on what you find here.

10.4 Simulation-based inference for regression

Recall in Subsection 10.2.5 when we interpreted the third through seventh columns of a regression table, we stated that R doesn't do simulations to compute these values. Rather R uses theory-based methods that involve mathematical formulas.

In this section, we'll use the simulation-based methods you previously learned in Chapters 8 and 9 to recreate the values in the regression table in Table 10.1. In particular, we'll use the `infer` package workflow to

- Construct a 95% confidence interval for the population slope β_1 using bootstrap resampling with replacement. We did this previously in Sections 8.4 with the `pennies` data and 8.6 with the `mythbusters_yawn` data.
- Conduct a hypothesis test of $H_0 : \beta_1 = 0$ versus $H_A : \beta_1 \neq 0$ using a permutation test. We did this previously in Sections 9.3 with the `promotions` data and 9.5 with the `movies_sample` IMDb data.

10.4.1 Confidence interval for slope

We'll construct a 95% confidence interval for β_1 using the `infer` workflow outlined in Subsection 8.4.2. Specifically, we'll first construct the bootstrap distribution for the fitted slope b_1 using our single sample of 463 courses:

1. `specify()` the variables of interest in `evals_ch5` with the formula: `score ~ bty_avg`.
2. `generate()` replicates by using `bootstrap` resampling with replacement from the original sample of 463 courses. We generate `reps = 1000` replicates using `type = "bootstrap"`.
3. `calculate()` the summary statistic of interest: the fitted slope b_1 .

Using this bootstrap distribution, we'll construct the 95% confidence interval using the percentile method and (if appropriate) the standard error method as well. It is important to note in this case that the bootstrapping with replacement is done *row-by-row*. Thus, the original pairs of `score` and `bty_avg` values are always kept together, but different pairs of `score` and `bty_avg` values may be resampled multiple times. The resulting confidence interval will denote a range of plausible values for the unknown population slope β_1 quantifying the relationship between teaching and “beauty” scores for *all* professors at UT Austin.

Let's first construct the bootstrap distribution for the fitted slope b_1 :

```
bootstrap_distn_slope <- evals_ch5 %>%
  specify(formula = score ~ bty_avg) %>%
  generate(reps = 1000, type = "bootstrap") %>%
  calculate(stat = "slope")
bootstrap_distn_slope
```

```
# A tibble: 1,000 x 2
  replicate      stat
  <int>     <dbl>
1       1 0.0651055
2       2 0.0382313
3       3 0.108056
4       4 0.0666601
5       5 0.0715932
6       6 0.0854565
7       7 0.0624868
8       8 0.0412859
9       9 0.0796269
10      10 0.0761299
# i 990 more rows
```

Observe how we have 1000 values of the bootstrapped slope b_1 in the `stat` column. Let's visualize the 1000 bootstrapped values in Figure 10.8.

```
visualize(bootstrap_distn_slope)
```

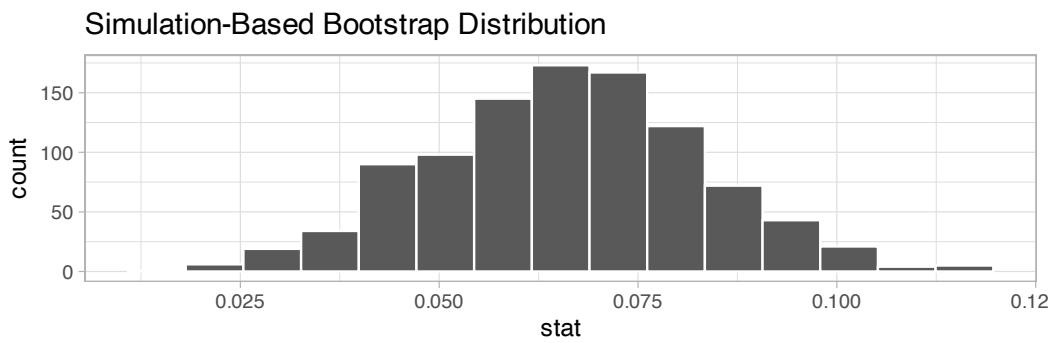


FIGURE 10.8: Bootstrap distribution of slope.

Observe how the bootstrap distribution is roughly bell-shaped. Recall from Subsection 8.7.1 that the shape of the bootstrap distribution of b_1 closely approximates the shape of the sampling distribution of b_1 .

Percentile-method

First, let's compute the 95% confidence interval for β_1 using the percentile method. We'll do so by identifying the 2.5th and 97.5th percentiles which include the middle 95% of values. Recall that this method does not require the bootstrap distribution to be normally shaped.

```
percentile_ci <- bootstrap_distn_slope %>%
  get_confidence_interval(type = "percentile", level = 0.95)
percentile_ci

# A tibble: 1 x 2
  lower_ci  upper_ci
  <dbl>     <dbl>
1 0.0323411 0.0990027
```

The resulting percentile-based 95% confidence interval for β_1 of (0.032, 0.099) is similar to the confidence interval in the regression Table 10.1 of (0.035, 0.099).

Standard error method

Since the bootstrap distribution in Figure 10.8 appears to be roughly bell-shaped, we can also construct a 95% confidence interval for β_1 using the standard error method.

In order to do this, we need to first compute the fitted slope b_1 , which will act as the center of our standard error-based confidence interval. While we saw in the regression table in Table 10.1 that this was $b_1 = 0.067$, we can also use the `infer` pipeline with the `generate()` step removed to calculate it:

```
observed_slope <- evals %>%
  specify(score ~ bty_avg) %>%
  calculate(stat = "slope")
observed_slope
```

```
Response: score (numeric)
Explanatory: bty_avg (numeric)
# A tibble: 1 x 1
  stat
  <dbl>
1 0.0666370
```

We then use the `get_ci()` function with `level = 0.95` to compute the 95% confidence interval for β_1 . Note that setting the `point_estimate` argument to the `observed_slope` of 0.067 sets the center of the confidence interval.

```
se_ci <- bootstrap_distn_slope %>%  
  get_ci(level = 0.95, type = "se", point_estimate = observed_slope)  
se_ci
```

```
# A tibble: 1 x 2
  lower_ci upper_ci
  <dbl>     <dbl>
1 0.0333767 0.0998974
```

The resulting standard error-based 95% confidence interval for β_1 of $(0.033, 0.1)$ is slightly different than the confidence interval in the regression Table 10.1 of $(0.035, 0.099)$.

Comparing all three

Let's compare all three confidence intervals in Figure 10.9, where the percentile-based confidence interval is marked with solid lines, the standard error based confidence interval is marked with dashed lines, and the theory-based confidence interval (0.035, 0.099) from the regression table in Table 10.1 is marked with dotted lines.

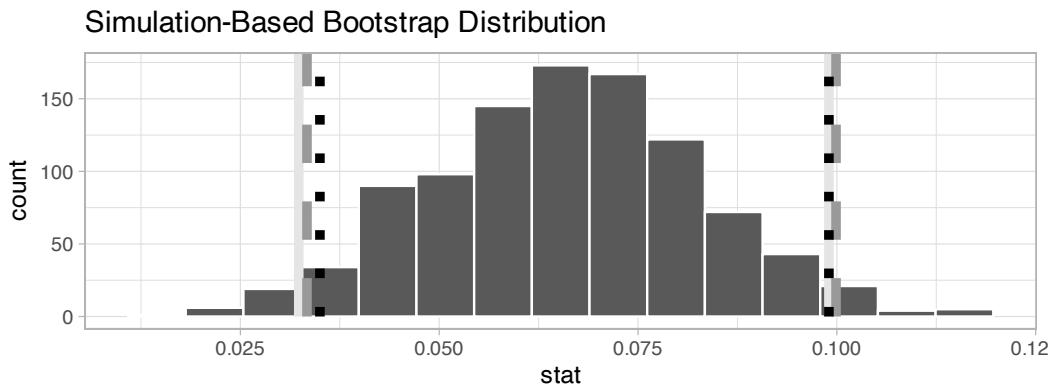


FIGURE 10.9: Comparing three confidence intervals for the slope.

Observe that all three are quite similar! Furthermore, none of the three confidence intervals for β_1 contain 0 and are entirely located above 0. This is suggesting that there is in fact a meaningful positive relationship between teaching and “beauty” scores.

10.4.2 Hypothesis test for slope

Let’s now conduct a hypothesis test of $H_0 : \beta_1 = 0$ vs. $H_A : \beta_1 \neq 0$. We will use the `infer` package, which follows the hypothesis testing paradigm in the “There is only one test” diagram in Figure 9.10.

Let’s first think about what it means for β_1 to be zero as assumed in the null hypothesis H_0 . Recall we said if $\beta_1 = 0$, then this is saying there is no relationship between the teaching and “beauty” scores. Thus assuming this particular null hypothesis H_0 means that in our “hypothesized universe” there is no relationship between `score` and `bty_avg`. We can therefore shuffle/permute the `bty_avg` variable to no consequence.

We construct the null distribution of the fitted slope b_1 by performing the steps that follow. Recall from Section 9.2 on terminology, notation, and definitions related to hypothesis testing where we defined the *null distribution*: the sampling distribution of our test statistic b_1 assuming the null hypothesis H_0 is true.

1. `specify()` the variables of interest in `evals_ch5` with the formula: `score ~ bty_avg`.
2. `hypothesize()` the null hypothesis of `independence`. Recall from Section 9.3 that this is an additional step that needs to be added for hypothesis testing.
3. `generate()` replicates by permuting/shuffling values from the original sample of 463 courses. We generate `reps = 1000` replicates using `type = "permute"` here.
4. `calculate()` the test statistic of interest: the fitted slope b_1 .

In this case, we `permute` the values of `bty_avg` across the values of `score` 1000 times. We can do this shuffling/permuting since we assumed a “hypothesized universe” of no relationship between these two variables. Then we calculate the “slope” coefficient for each of these 1000 generated samples.

```
null_distn_slope <- evals %>%
  specify(score ~ bty_avg) %>%
  hypothesize(null = "independence") %>%
  generate(reps = 1000, type = "permute") %>%
  calculate(stat = "slope")
```

Observe the resulting null distribution for the fitted slope b_1 in Figure 10.10.

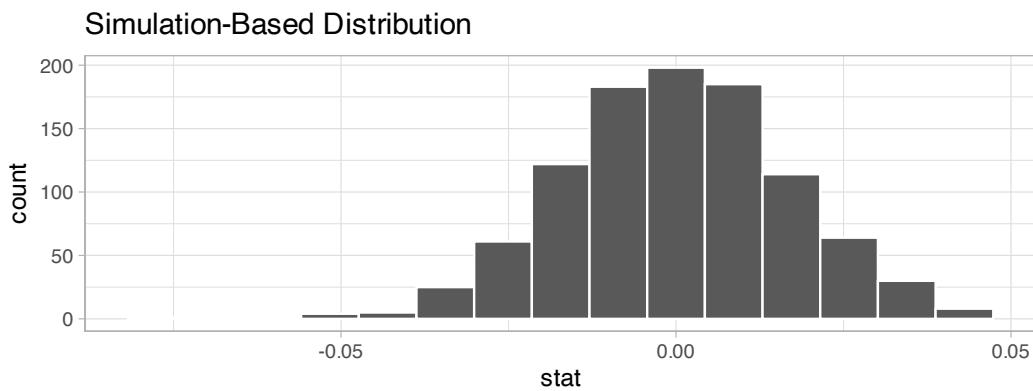


FIGURE 10.10: Null distribution of slopes.

Notice how it is centered at $b_1 = 0$. This is because in our hypothesized universe, there is no relationship between `score` and `bty_avg` and so $\beta_1 = 0$. Thus, the most typical fitted slope b_1 we observe across our simulations is 0. Observe, furthermore, how there is variation around this central value of 0.

Let’s visualize the p -value in the null distribution by comparing it to the observed test statistic of $b_1 = 0.067$ in Figure 10.11. We’ll do this by adding a `shade_p_value()` layer to the previous `visualize()` code.

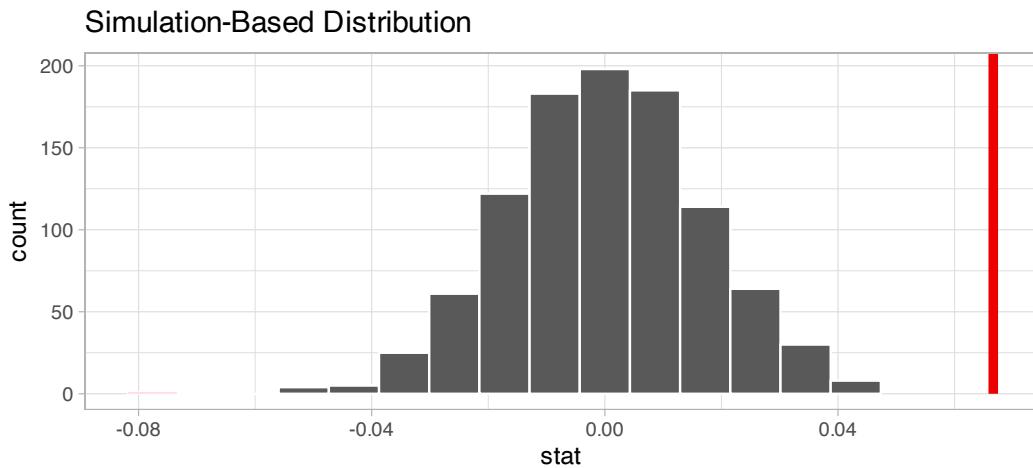


FIGURE 10.11: Null distribution and p -value.

Since the observed fitted slope 0.067 falls far to the right of this null distribution and thus the shaded region doesn't overlap it, we'll have a p -value of 0. For completeness, however, let's compute the numerical value of the p -value anyways using the `get_p_value()` function. Recall that it takes the same inputs as the `shade_p_value()` function:

```
null_distn_slope %>%
  get_p_value(obs_stat = observed_slope, direction = "both")
```

```
# A tibble: 1 x 1
  p_value
  <dbl>
1      0
```

This matches the p -value of 0 in the regression table in Table 10.1. We therefore reject the null hypothesis $H_0 : \beta_1 = 0$ in favor of the alternative hypothesis $H_A : \beta_1 \neq 0$. We thus have evidence that suggests there is a significant relationship between teaching and “beauty” scores for *all* instructors at UT Austin.

When the conditions for inference for regression are met and the null distribution has a bell shape, we are likely to see similar results between the simulation-based results we just demonstrated and the theory-based results shown in the regression table in Table 10.1.

Learning check

(LC10.2) Repeat the inference but this time for the correlation coefficient instead of the slope. Note the implementation of `stat = "correlation"` in the `calculate()` function of the `infer` package.

10.5 Conclusion

10.5.1 Theory-based inference for regression

Recall in Subsection 10.2.5 when we interpreted the regression table in Table 10.1, we mentioned that R does not compute its values using simulation-based methods for constructing confidence intervals and conducting hypothesis tests as we did in Chapters 8 and 9 using the `infer` package. Rather, R uses a theory-based approach using mathematical formulas, much like the theory-based confidence intervals you saw in Subsection 8.7.2 and the theory-based hypothesis tests you saw in Subsection 9.6.1. These formulas were derived in a time when computers didn't exist, so it would've been incredibly labor intensive to run extensive simulations.

In particular, much like the formula for the standard error for the sample proportion \hat{p} we saw in Subsection 7.6.2 and the formula for the standard error for the difference in sample means $\bar{x}_1 - \bar{x}_2$ we saw in Subsection 9.6.1, there is a formula for the standard error of the fitted slope b_1 :

$$\text{SE}_{b_1} = \frac{\frac{s_y}{s_x} \cdot \sqrt{1 - r^2}}{\sqrt{n - 2}}$$

As with many formulas in statistics, there's a lot going on here, so let's first break down what each symbol represents. First s_x and s_y are the *sample standard deviations* of the explanatory variable `bty_avg` and the response variable `score`, respectively. Second, r is the sample *correlation coefficient* between `score` and `bty_avg`. This was computed as 0.187 in Chapter 5. Lastly, n is the number of pairs of points in the `evals_ch5` data frame, here 463.

To put this formula into words, the standard error of b_1 depends on the relationship between the variability of the response variable and the variability of the explanatory variable as measured in the s_y/s_x term. Next, it looks into how the two variables relate to each other in the $\sqrt{1 - r^2}$ term.

However, the most important observation to make in the previous formula is that there is an $n - 2$ in the denominator. In other words, as the sample size n increases, the standard error SE_{b_1} decreases. Just as we demonstrated in Subsection 7.3.3 when

we used shovels with $n = 25, 50$, and 100 slots, the amount of sampling variation of the fitted slope b_1 will depend on the sample size n . In particular, as the sample size increases, both the sampling and bootstrap distributions narrow and the standard error SE_{b_1} decreases. Hence, our estimates of b_1 for the true population slope β_1 get more and more *precise*.

R then uses this formula for the standard error of b_1 in the third column of the regression table and subsequently to construct 95% confidence intervals. But what about the hypothesis test? Much like with our theory-based hypothesis test in Subsection 9.6.1, R uses the following *t-statistic* as the test statistic for hypothesis testing:

$$t = \frac{b_1 - \beta_1}{\text{SE}_{b_1}}$$

And since the null hypothesis $H_0 : \beta_1 = 0$ is assumed during the hypothesis test, the *t*-statistic becomes

$$t = \frac{b_1 - 0}{\text{SE}_{b_1}} = \frac{b_1}{\text{SE}_{b_1}}$$

What are the values of b_1 and SE_{b_1} ? They are in the `estimate` and `std_error` column of the regression table in Table 10.1. Thus the value of 4.09 in the table is computed as $0.067/0.016 = 4.188$. Note there is a difference due to some rounding error here.

Lastly, to compute the *p*-value, we need to compare the observed test statistic of 4.09 to the appropriate null distribution. Recall from Section 9.2, that a null distribution is the sampling distribution of the test statistic *assuming the null hypothesis H_0 is true*. Much like in our theory-based hypothesis test in Subsection 9.6.1, it can be mathematically proven that this distribution is a *t*-distribution with degrees of freedom equal to $df = n - 2 = 463 - 2 = 461$.

Don't worry if you're feeling a little overwhelmed at this point. There is a lot of background theory to understand before you can fully make sense of the equations for theory-based methods. That being said, theory-based methods and simulation-based methods for constructing confidence intervals and conducting hypothesis tests often yield consistent results. As mentioned before, in our opinion, two large benefits of simulation-based methods over theory-based are that (1) they are easier for people new to statistical inference to understand, and (2) they also work in situations where theory-based methods and mathematical formulas don't exist.

10.5.2 Summary of statistical inference

We've finished the last two scenarios from the "Scenarios of sampling for inference" table in Subsection 7.6.1, which we re-display in Table 10.4.

Armed with the regression modeling techniques you learned in Chapters 5 and 6, your understanding of sampling for inference in Chapter 7, and the tools for statistical

TABLE 10.4: Scenarios of sampling for inference

Scenario	Population parameter	Notation	Point estimate	Symbol(s)
1	Population proportion	p	Sample proportion	\hat{p}
2	Population mean	μ	Sample mean	\bar{x} or $\hat{\mu}$
3	Difference in population proportions	$p_1 - p_2$	Difference in sample proportions	$\hat{p}_1 - \hat{p}_2$
4	Difference in population means	$\mu_1 - \mu_2$	Difference in sample means	$\bar{x}_1 - \bar{x}_2$ or $\hat{\mu}_1 - \hat{\mu}_2$
5	Population regression slope	β_1	Fitted regression slope	b_1 or $\hat{\beta}_1$

inference like confidence intervals and hypothesis tests in Chapters 8 and 9, you’re now equipped to study the significance of relationships between variables in a wide array of data! Many of the ideas presented here can be extended into multiple regression and other more advanced modeling techniques.

10.5.3 Additional resources

Solutions to all *Learning checks* can be found online in Appendix D¹.

An R script file of all R code used in this chapter is available at <https://www.moderndive.com/scripts/10-inference-for-regression.R>.

10.5.4 What’s to come

You’ve now concluded the last major part of the book on “Statistical Inference with `infer`.” The closing Chapter 11 concludes this book with various short case studies involving real data, such as house prices in the city of Seattle, Washington in the US. You’ll see how the principles in this book can help you become a great storyteller with data!

¹<https://moderndive.com/D-appendixD.html>

Part IV

Conclusion

11

Tell Your Story with Data

Recall in the Preface and at the end of chapters throughout this book, we displayed the “*ModernDive* flowchart” mapping your journey through this book.

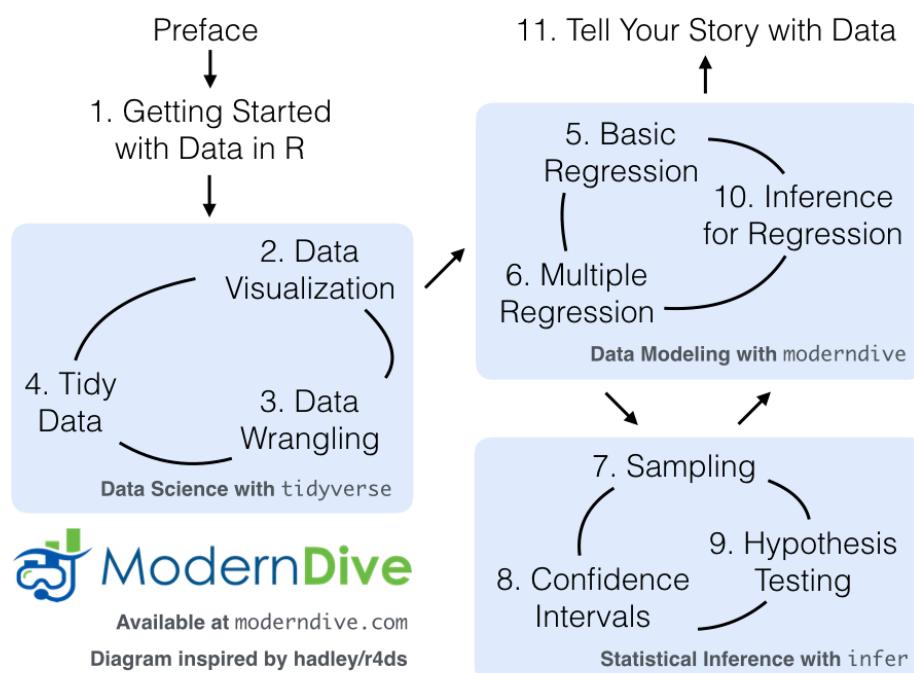


FIGURE 11.1: *ModernDive* flowchart.

11.1 Review

Let’s go over a refresher of what you’ve covered so far. You first got started with data in Chapter 1 where you learned about the difference between R and RStudio, started coding in R, installed and loaded your first R packages, and explored your first dataset: all domestic departure flights from a major New York City airport in

2023. Then you covered the following three parts of this book (Parts 2 and 4 are combined into a single portion):

1. Data science with `tidyverse`. You assembled your data science toolbox using `tidyverse` packages. In particular, you
 - Ch.2: Visualized data using the `ggplot2` package.
 - Ch.3: Wrangled data using the `dplyr` package.
 - Ch.4: Learned about the concept of “tidy” data as a standardized data frame input and output format for all packages in the `tidyverse`. Furthermore, you learned how to import spreadsheet files into R using the `readr` package.
2. Statistical/Data modeling with `moderndive`. Using these data science tools and helper functions from the `moderndive` package, you fit your first data models. In particular, you
 - Ch.5: Discovered basic regression models with only one explanatory variable.
 - Ch.6: Examined multiple regression models with more than one explanatory variable.
3. Statistical inference with `infer`. Once again using your newly acquired data science tools, you unpacked statistical inference using the `infer` package. In particular, you
 - Ch.7: Learned about the role that sampling variability plays in statistical inference and the role that sample size plays in this sampling variability.
 - Ch.8: Constructed confidence intervals using bootstrapping.
 - Ch.9: Conducted hypothesis tests using permutation.
4. Statistical/Data modeling with `moderndive` (revisited): Armed with your understanding of statistical inference, you revisited and reviewed the models you constructed in Ch.5 and Ch.6. In particular, you
 - Ch.10: Interpreted confidence intervals and hypothesis tests in a regression setting.

We’ve guided you through your first experiences of “thinking with data,”¹ an expression originally coined by Dr. Diane Lambert. The philosophy underlying this expression guided your path in the flowchart in Figure 11.1.

This philosophy is also well-summarized in “Practical Data Science for Stats”²: a collection of pre-prints focusing on the practical side of data science workflows and

¹<https://arxiv.org/pdf/1410.3127.pdf>

²<https://peerj.com/collections/50-practicaldatascistats/>

statistical analysis curated by Dr. Jennifer Bryan³ and Dr. Hadley Wickham⁴. They quote:

There are many aspects of day-to-day analytical work that are almost absent from the conventional statistics literature and curriculum. And yet these activities account for a considerable share of the time and effort of data analysts and applied statisticians. The goal of this collection is to increase the visibility and adoption of modern data analytical workflows. We aim to facilitate the transfer of tools and frameworks between industry and academia, between software engineering and statistics and computer science, and across different domains.

In other words, to be equipped to “think with data” in the 21st century, analysts need practice going through the “data/science pipeline”⁵ we saw in the Preface (re-displayed in Figure 11.2). It is our opinion that, for too long, statistics education has only focused on parts of this pipeline, instead of going through it in its *entirety*.

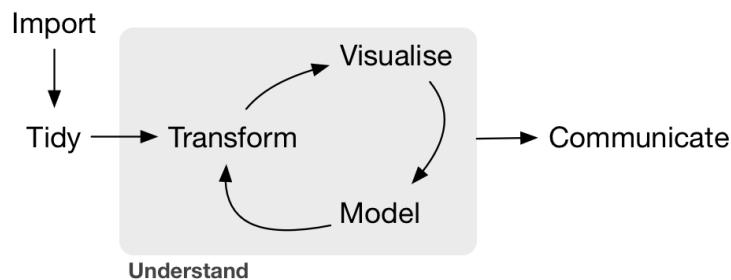


FIGURE 11.2: Data/science pipeline.

To conclude this book, we’ll present you with some additional case studies of working with data. In Section 11.2 we’ll take you through a full-pass of the “Data/Science Pipeline” in order to analyze the sale price of houses in Seattle, WA, USA. In Section 11.3, we’ll present you with some examples of effective data storytelling drawn from the data journalism website, FiveThirtyEight.com⁶. We present these case studies to you because we believe that you should not only be able to “think with data,” but also be able to “tell your story with data.” Let’s explore how to do this!

³<https://twitter.com/jennybryan>

⁴<https://twitter.com/hadleywickham>

⁵<http://r4ds.had.co.nz/explore-intro.html>

⁶<https://fivethirtyeight.com/>

Needed packages

Let's load all the packages needed for this chapter (this assumes you've already installed them). Read Section 1.3 for information on how to install and load R packages.

```
library(tidyverse)
library(moderndive)
library(skimr)
library(fivethirtyeight)
```

11.2 Case study: Seattle house prices

Kaggle.com⁷ is a machine learning and predictive modeling competition website that hosts datasets uploaded by companies, governmental organizations, and other individuals. One of their datasets is the “House Sales in King County, USA”⁸. It consists of sale prices of homes sold between May 2014 and May 2015 in King County, Washington, USA, which includes the greater Seattle metropolitan area. This dataset is in the `house_prices` data frame included in the `moderndive` package.

The dataset consists of 21,613 houses and 21 variables describing these houses (for a full list and description of these variables, see the help file by running `?house_prices` in the console). In this case study, we'll create a multiple regression model where:

- The outcome variable y is the sale `price` of houses.
- Two explanatory variables:
 1. A numerical explanatory variable x_1 : house size `sqft_living` as measured in square feet of living space. Note that 1 square foot is about 0.09 square meters.
 2. A categorical explanatory variable x_2 : house `condition`, a categorical variable with five levels where 1 indicates “poor” and 5 indicates “excellent.”

11.2.1 Exploratory data analysis: Part I

As we've said numerous times throughout this book, a crucial first step when presented with data is to perform an exploratory data analysis (EDA). Exploratory data analysis can give you a sense of your data, help identify issues with your data, bring to light any outliers, and help inform model construction.

⁷<https://www.kaggle.com/>

⁸<https://www.kaggle.com/harlfoxem/housesalesprediction>

Recall the three common steps in an exploratory data analysis we introduced in Subsection 5.1.1:

1. Looking at the raw data values.
2. Computing summary statistics.
3. Creating data visualizations.

First, let's look at the raw data using `View()` to bring up RStudio's spreadsheet viewer and the `glimpse()` function from the `dplyr` package:

```
View(house_prices)
glimpse(house_prices)
```

```
Rows: 21,613
Columns: 21
$ id              <chr> "7129300520", "6414100192", "5631500400", "24872008~ 
$ date            <date> 2014-10-13, 2014-12-09, 2015-02-25, 2014-12-09, 20~ 
$ price            <dbl> 221900, 538000, 180000, 604000, 510000, 1225000, 25~ 
$ bedrooms         <int> 3, 3, 2, 4, 3, 4, 3, 3, 3, 3, 2, 3, 3, 5, 4, 3, ~ 
$ bathrooms        <dbl> 1.00, 2.25, 1.00, 3.00, 2.00, 4.50, 2.25, 1.50, 1.0~ 
$ sqft_living      <int> 1180, 2570, 770, 1960, 1680, 5420, 1715, 1060, 1780~ 
$ sqft_lot          <int> 5650, 7242, 10000, 5000, 8080, 101930, 6819, 9711, ~ 
$ floors           <dbl> 1.0, 2.0, 1.0, 1.0, 1.0, 2.0, 1.0, 1.0, 2.0, 1~ 
$ waterfront        <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FA~ 
$ view              <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~ 
$ condition         <fct> 3, 3, 3, 5, 3, 3, 3, 3, 3, 3, 4, 4, 4, 3, 3, 3, ~ 
$ grade             <fct> 7, 7, 6, 7, 8, 11, 7, 7, 7, 8, 7, 7, 7, 7, 9, 7, ~ 
$ sqft_above         <int> 1180, 2170, 770, 1050, 1680, 3890, 1715, 1060, 1050~ 
$ sqft_basement     <int> 0, 400, 0, 910, 0, 1530, 0, 0, 730, 0, 1700, 300, 0~ 
$ yr_built          <int> 1955, 1951, 1933, 1965, 1987, 2001, 1995, 1963, 196~ 
$ yr_renovated      <int> 0, 1991, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~ 
$ zipcode           <fct> 98178, 98125, 98028, 98136, 98074, 98053, 98003, 98~ 
$ lat               <dbl> 47.5, 47.7, 47.7, 47.5, 47.6, 47.7, 47.3, 47.4, 47.~ 
$ long              <dbl> -122, -122, -122, -122, -122, -122, -122, -122, -12~ 
$ sqft_living15     <int> 1340, 1690, 2720, 1360, 1800, 4760, 2238, 1650, 178~ 
$ sqft_lot15         <int> 5650, 7639, 8062, 5000, 7503, 101930, 6819, 9711, 8~
```

Here are some questions you can ask yourself at this stage of an EDA: Which variables are numerical? Which are categorical? For the categorical variables, what are their levels? Besides the variables we'll be using in our regression model, what other variables do you think would be useful to use in a model for house price?

Observe, for example, that while the `condition` variable has values 1 through 5, these are saved in R as `fct` standing for “factors.” This is one of R’s ways of saving cate-

gorical variables. So you should think of these as the “labels” 1 through 5 and not the numerical values 1 through 5.

Let’s now perform the second step in an EDA: computing summary statistics. Recall from Section 3.3 that *summary statistics* are single numerical values that summarize a large number of values. Examples of summary statistics include the mean, the median, the standard deviation, and various percentiles.

We could do this using the `summarize()` function in the `dplyr` package along with R’s built-in *summary functions*, like `mean()` and `median()`. However, recall in Section 3.5, we saw the following code that computes a variety of summary statistics of the variable `gain`, which is the amount of time that a flight makes up mid-air:

```
gain_summary <- flights %>%
  summarize(
    min = min(gain, na.rm = TRUE),
    q1 = quantile(gain, 0.25, na.rm = TRUE),
    median = quantile(gain, 0.5, na.rm = TRUE),
    q3 = quantile(gain, 0.75, na.rm = TRUE),
    max = max(gain, na.rm = TRUE),
    mean = mean(gain, na.rm = TRUE),
    sd = sd(gain, na.rm = TRUE),
    missing = sum(is.na(gain))
  )
```

To repeat this for all three `price`, `sqft_living`, and `condition` variables would be tedious to code up. So instead, let’s use the convenient `skim()` function from the `skimr` package we first used in Subsection 6.1.1, being sure to only `select()` the variables of interest for our model:

```
house_prices %>%
  select(price, sqft_living, condition) %>%
  skim()
```

```
Skim summary statistics
n obs: 21613
n variables: 3

— Variable type:factor
  variable missing complete      n n_unique          top_counts ordered
  condition      0     21613 21613       5 3: 14031, 4: 5679, 5: 1701, 2: 172   FALSE

— Variable type:integer
```

```

variable missing complete      n   mean      sd   p0   p25   p50   p75   p100
sqft_living       0    21613 21613 2079.9 918.44 290 1427 1910 2550 13540

— Variable type:numeric
variable missing complete      n   mean      sd   p0   p25   p50   p75   p100
price            0    21613 21613 540088.14 367127.2 75000 321950 450000 645000 7700000

```

Observe that the mean price of \$540,088 is larger than the median of \$450,000. This is because a small number of very expensive houses are inflating the average. In other words, there are “outlier” house prices in our dataset. (This fact will become even more apparent when we create our visualizations next.)

However, the median is not as sensitive to such outlier house prices. This is why news about the real estate market generally report median house prices and not mean/average house prices. We say here that the median is more *robust to outliers* than the mean. Similarly, while both the standard deviation and interquartile-range (IQR) are both measures of spread and variability, the IQR is more *robust to outliers*.

Let’s now perform the last of the three common steps in an exploratory data analysis: creating data visualizations. Let’s first create *univariate* visualizations. These are plots focusing on a single variable at a time. Since `price` and `sqft_living` are numerical variables, we can visualize their distributions using a `geom_histogram()` as seen in Section 2.5 on histograms. On the other hand, since `condition` is categorical, we can visualize its distribution using a `geom_bar()`. Recall from Section 2.8 on barplots that since `condition` is not “pre-counted”, we use a `geom_bar()` and not a `geom_col()`.

```

# Histogram of house price:
ggplot(house_prices, aes(x = price)) +
  geom_histogram(color = "white") +
  labs(x = "price (USD)", title = "House price")

# Histogram of sqft_living:
ggplot(house_prices, aes(x = sqft_living)) +
  geom_histogram(color = "white") +
  labs(x = "living space (square feet)", title = "House size")

# Barplot of condition:
ggplot(house_prices, aes(x = condition)) +
  geom_bar() +
  labs(x = "condition", title = "House condition")

```

In Figure 11.3, we display all three of these visualizations at once.

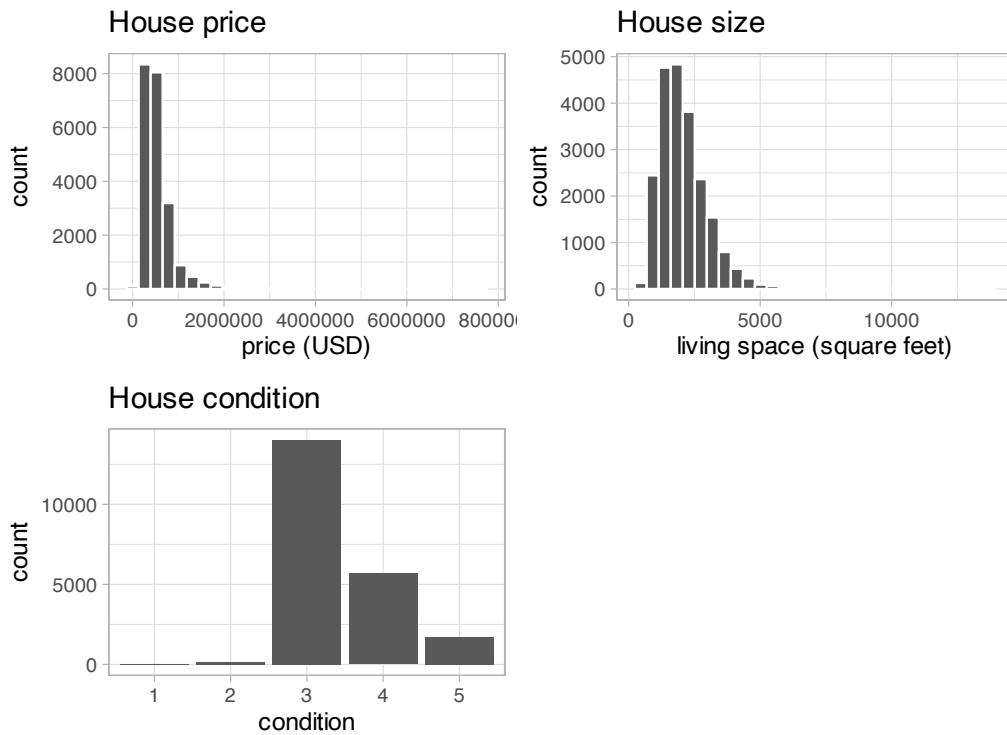


FIGURE 11.3: Exploratory visualizations of Seattle house prices data.

First, observe in the bottom plot that most houses are of condition “3”, with a few more of conditions “4” and “5”, and almost none that are “1” or “2”.

Next, observe in the histogram for `price` in the top-left plot that a majority of houses are less than two million dollars. Observe also that the x-axis stretches out to 8 million dollars, even though there does not appear to be any houses close to that price. This is because there are a *very small number* of houses with prices closer to 8 million. These are the outlier house prices we mentioned earlier. We say that the variable `price` is *right-skewed* as exhibited by the long right tail.

Further, observe in the histogram of `sqft_living` in the middle plot as well that most houses appear to have less than 5000 square feet of living space. For comparison, a football field in the US is about 57,600 square feet, whereas a standard soccer/association football field is about 64,000 square feet. Observe also that this variable is also right-skewed, although not as drastically as the `price` variable.

For both the `price` and `sqft_living` variables, the right-skew makes distinguishing houses at the lower end of the x-axis hard. This is because the scale of the x-axis is compressed by the small number of quite expensive and immensely-sized houses.

So what can we do about this skew? Let’s apply a *log10 transformation* to these variables. If you are unfamiliar with such transformations, we highly recommend you read Appendix A.3 on logarithmic (log) transformations. In summary, log transformations

allow us to alter the scale of a variable to focus on *multiplicative* changes instead of *additive* changes. In other words, they shift the view to be on *relative* changes instead of *absolute* changes. Such multiplicative/relative changes are also called changes in *orders of magnitude*.

Let's create new \log_{10} transformed versions of the right-skewed variable `price` and `sqft_living` using the `mutate()` function from Section 3.5, but we'll give the latter the name `log10_size`, which is shorter and easier to understand than the name `log10_sqft_living`.

```
house_prices <- house_prices %>%
  mutate(
    log10_price = log10(price),
    log10_size = log10(sqft_living)
  )
```

Let's display the before and after effects of this transformation on these variables for only the first 10 rows of `house_prices`:

```
house_prices %>%
  select(price, log10_price, sqft_living, log10_size)
```

```
# A tibble: 21,613 x 4
  price log10_price sqft_living log10_size
  <dbl>      <dbl>       <int>      <dbl>
1 221900     5.34616     1180     3.07188
2 538000     5.73078     2570     3.40993
3 180000     5.25527     770      2.88649
4 604000     5.78104     1960     3.29226
5 510000     5.70757     1680     3.22531
6 1225000    6.08814     5420     3.73400
7 257500     5.41078     1715     3.23426
8 291850     5.46516     1060     3.02531
9 229500     5.36078     1780     3.25042
10 323000    5.50920     1890     3.27646
# i 21,603 more rows
```

Observe in particular the houses in the sixth and third rows. The house in the sixth row has `price` \$1,225,000, which is just above one million dollars. Since 10^6 is one million, its `log10_price` is around 6.09.

Contrast this with all other houses with `log10_price` less than six, since they all have `price` less than \$1,000,000. The house in the third row is the only house with

`sqft_living` less than 1000. Since $1000 = 10^3$, it's the lone house with `log10_size` less than 3.

Let's now visualize the before and after effects of this transformation for `price` in Figure 11.4.

```
# Before log10 transformation:
ggplot(house_prices, aes(x = price)) +
  geom_histogram(color = "white") +
  labs(x = "price (USD)", title = "House price: Before")

# After log10 transformation:
ggplot(house_prices, aes(x = log10_price)) +
  geom_histogram(color = "white") +
  labs(x = "log10 price (USD)", title = "House price: After")
```

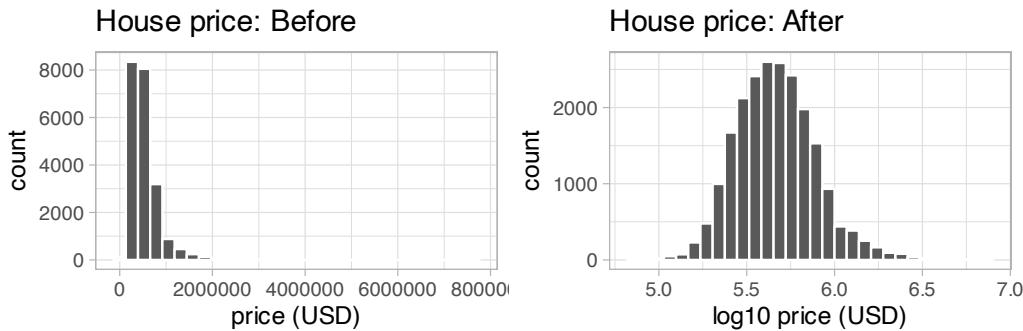


FIGURE 11.4: House price before and after log10 transformation.

Observe that after the transformation, the distribution is much less skewed, and in this case, more symmetric and more bell-shaped. Now you can more easily distinguish the lower priced houses.

Let's do the same for house size, where the variable `sqft_living` was log10 transformed to `log10_size`.

```
# Before log10 transformation:
ggplot(house_prices, aes(x = sqft_living)) +
  geom_histogram(color = "white") +
  labs(x = "living space (square feet)", title = "House size: Before")

# After log10 transformation:
```

```
ggplot(house_prices, aes(x = log10_size)) +
  geom_histogram(color = "white") +
  labs(x = "log10 living space (square feet)", title = "House size: After")
```

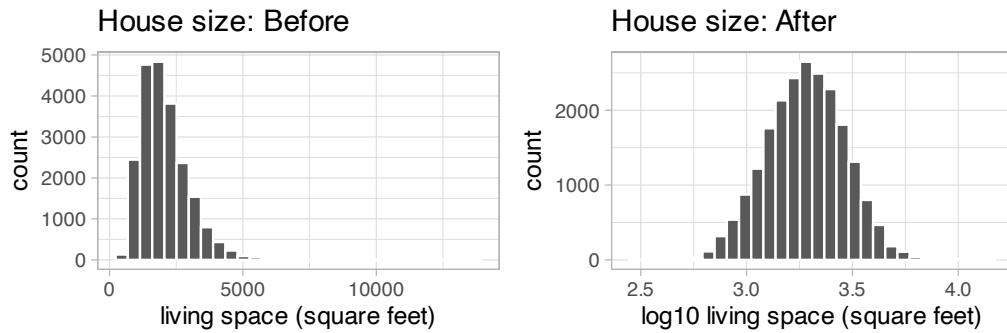


FIGURE 11.5: House size before and after log10 transformation.

Observe in Figure 11.5 that the log10 transformation has a similar effect of unskewing the variable. We emphasize that while in these two cases the resulting distributions are more symmetric and bell-shaped, this is not always necessarily the case.

Given the now symmetric nature of `log10_price` and `log10_size`, we are going to revise our multiple regression model to use our new variables:

1. The outcome variable y is the sale `log10_price` of houses.
2. Two explanatory variables:
 1. A numerical explanatory variable x_1 : house size `log10_size` as measured in log base 10 square feet of living space.
 2. A categorical explanatory variable x_2 : house condition, a categorical variable with five levels where 1 indicates “poor” and 5 indicates “excellent.”

11.2.2 Exploratory data analysis: Part II

Let’s now continue our EDA by creating *multivariate* visualizations. Unlike the *univariate* histograms and barplot in the earlier Figures 11.3, 11.4, and 11.5, *multivariate* visualizations show relationships between more than one variable. This is an important step of an EDA to perform since the goal of modeling is to explore relationships between variables.

Since our model involves a numerical outcome variable, a numerical explanatory variable, and a categorical explanatory variable, we are in a similar regression modeling situation as in Section 6.1 where we studied the UT Austin teaching scores dataset.

Recall in that case the numerical outcome variable was teaching score, the numerical explanatory variable was instructor age, and the categorical explanatory variable was (binary) gender.

We thus have two choices of models we can fit: either (1) an *interaction model* where the regression line for each condition level will have both a different slope and a different intercept or (2) a *parallel slopes model* where the regression line for each condition level will have the same slope but different intercepts.

Recall from Subsection 6.1.3 that the `geom_parallel_slopes()` function is a special purpose function that Evgeni Chasnovski created and included in the `moderndive` package, since the `geom_smooth()` method in the `ggplot2` package does not have a convenient way to plot parallel slopes models. We plot both resulting models in Figure 11.6, with the interaction model on the left.

```
# Plot interaction model
ggplot(house_prices,
       aes(x = log10_size, y = log10_price, col = condition)) +
  geom_point(alpha = 0.05) +
  geom_smooth(method = "lm", se = FALSE) +
  labs(y = "log10 price",
       x = "log10 size",
       title = "House prices in Seattle")
# Plot parallel slopes model
ggplot(house_prices,
       aes(x = log10_size, y = log10_price, col = condition)) +
  geom_point(alpha = 0.05) +
  geom_parallel_slopes(se = FALSE) +
  labs(y = "log10 price",
       x = "log10 size",
       title = "House prices in Seattle")
```

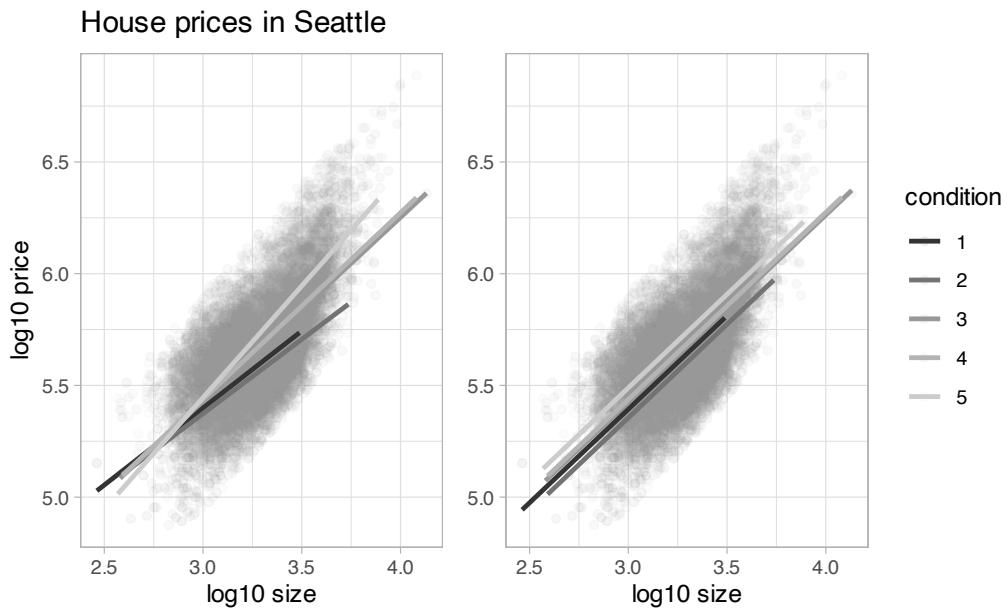


FIGURE 11.6: Interaction and parallel slopes models.

In both cases, we see there is a positive relationship between house price and size, meaning as houses are larger, they tend to be more expensive. Furthermore, in both plots it seems that houses of condition 5 tend to be the most expensive for most house sizes as evidenced by the fact that the line for condition 5 is highest, followed by conditions 4 and 3. As for conditions 1 and 2, this pattern isn't as clear. Recall from the univariate barplot of `condition` in Figure 11.3, there are only a few houses of condition 1 or 2.

Let's also show a faceted version of just the interaction model in Figure 11.7. It is now much more apparent just how few houses are of condition 1 or 2.

```
ggplot(house_prices,
       aes(x = log10_size, y = log10_price, col = condition)) +
  geom_point(alpha = 0.4) +
  geom_smooth(method = "lm", se = FALSE) +
  labs(y = "log10 price",
       x = "log10 size",
       title = "House prices in Seattle") +
  facet_wrap(~ condition)
```

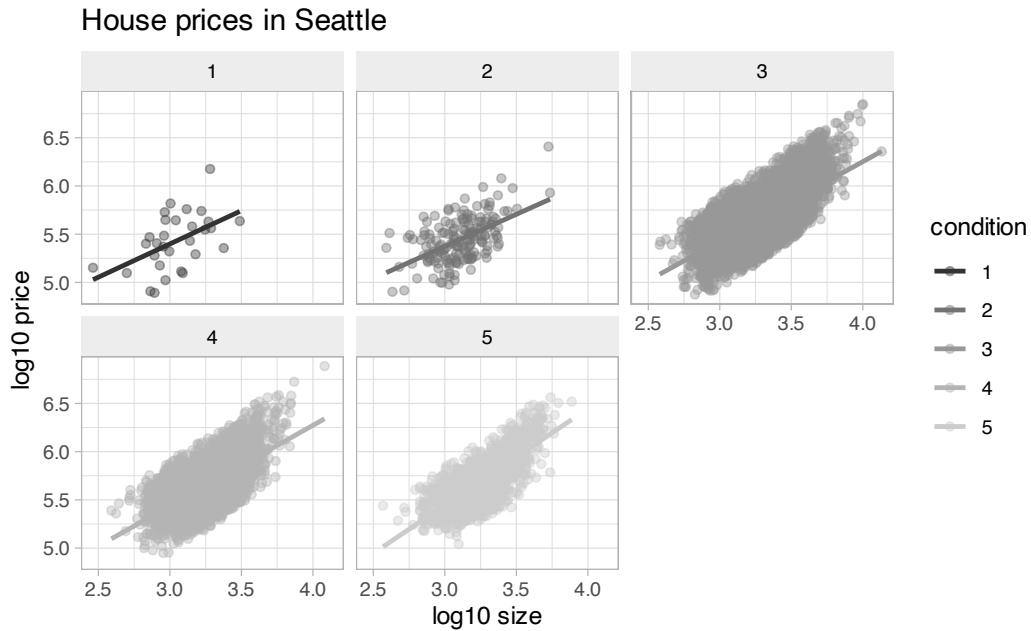


FIGURE 11.7: Faceted plot of interaction model.

Which exploratory visualization of the interaction model is better, the one in the left-hand plot of Figure 11.6 or the faceted version in Figure 11.7? There is no universal right answer. You need to make a choice depending on what you want to convey, and own that choice, with including and discussing both also as an option as needed.

11.2.3 Regression modeling

Which of the two models in Figure 11.6 is “better”? The interaction model in the left-hand plot or the parallel slopes model in the right-hand plot?

We had a similar discussion in Subsection ?? on *model selection*. Recall that we stated that we should only favor more complex models if the additional complexity is *warranted*. In this case, the more complex model is the interaction model since it considers five intercepts and five slopes total. This is in contrast to the parallel slopes model which considers five intercepts but only one common slope.

Is the additional complexity of the interaction model warranted? Looking at the left-hand plot in Figure 11.6, we’re of the opinion that it is, as evidenced by the slight x-like pattern to some of the lines. Therefore, we’ll focus the rest of this analysis only on the interaction model. This visual approach is somewhat subjective, however, so feel free to disagree! What are the five different slopes and five different intercepts for the interaction model? We can obtain these values from the regression table. Recall our two-step process for getting the regression table:

```
# Fit regression model:
price_interaction <- lm(log10_price ~ log10_size * condition,
                         data = house_prices)

# Get regression table:
get_regression_table(price_interaction)
```

TABLE 11.1: Regression table for interaction model

term	estimate	std_error	statistic	p_value	lower_ci	upper_ci
intercept	3.330	0.451	7.380	0.000	2.446	4.215
log10_size	0.690	0.148	4.652	0.000	0.399	0.980
condition: 2	0.047	0.498	0.094	0.925	-0.930	1.024
condition: 3	-0.367	0.452	-0.812	0.417	-1.253	0.519
condition: 4	-0.398	0.453	-0.879	0.380	-1.286	0.490
condition: 5	-0.883	0.457	-1.931	0.053	-1.779	0.013
log10_size:condition2	-0.024	0.163	-0.148	0.882	-0.344	0.295
log10_size:condition3	0.133	0.148	0.893	0.372	-0.158	0.424
log10_size:condition4	0.146	0.149	0.979	0.328	-0.146	0.437
log10_size:condition5	0.310	0.150	2.067	0.039	0.016	0.604

Recall we saw in Subsection 6.1.2 how to interpret a regression table when there are both numerical and categorical explanatory variables. Let's now do the same for all 10 values in the `estimate` column of Table 11.1.

In this case, the “baseline for comparison” group for the categorical variable `condition` are the condition 1 houses, since “1” comes first alphanumerically. Thus, the `intercept` and `log10_size` values are the intercept and slope for `log10_size` for this baseline group. Next, the `condition2` through `condition5` terms are the *offsets* in intercepts relative to the condition 1 intercept. Finally, the `log10_size:condition2` through `log10_size:condition5` are the *offsets* in slopes for `log10_size` relative to the condition 1 slope for `log10_size`.

Let's simplify this by writing out the equation of each of the five regression lines using these 10 `estimate` values. We'll write out each line in the following format:

$$\widehat{\log 10(\text{price})} = \hat{\beta}_0 + \hat{\beta}_{\text{size}} \cdot \log 10(\text{size})$$

1. Condition 1:

$$\widehat{\log 10(\text{price})} = 3.33 + 0.69 \cdot \log 10(\text{size})$$

2. Condition 2:

$$\begin{aligned}\widehat{\log 10(\text{price})} &= (3.33 + 0.047) + (0.69 - 0.024) \cdot \log 10(\text{size}) \\ &= 3.377 + 0.666 \cdot \log 10(\text{size})\end{aligned}$$

3. Condition 3:

$$\begin{aligned}\widehat{\log 10(\text{price})} &= (3.33 - 0.367) + (0.69 + 0.133) \cdot \log 10(\text{size}) \\ &= 2.963 + 0.823 \cdot \log 10(\text{size})\end{aligned}$$

4. Condition 4:

$$\begin{aligned}\widehat{\log 10(\text{price})} &= (3.33 - 0.398) + (0.69 + 0.146) \cdot \log 10(\text{size}) \\ &= 2.932 + 0.836 \cdot \log 10(\text{size})\end{aligned}$$

5. Condition 5:

$$\begin{aligned}\widehat{\log 10(\text{price})} &= (3.33 - 0.883) + (0.69 + 0.31) \cdot \log 10(\text{size}) \\ &= 2.447 + 1 \cdot \log 10(\text{size})\end{aligned}$$

These correspond to the regression lines in the left-hand plot of Figure 11.6 and the faceted plot in Figure 11.7. For homes of all five condition types, as the size of the house increases, the price increases. This is what most would expect. However, the rate of increase of price with size is fastest for the homes with conditions 3, 4, and 5 of 0.823, 0.836, and 1, respectively. These are the three largest slopes out of the five.

11.2.4 Making predictions

Say you're a realtor and someone calls you asking you how much their home will sell for. They tell you that it's in condition = 5 and is sized 1900 square feet. What do you tell them? Let's use the interaction model we fit to make predictions!

We first make this prediction visually in Figure 11.8. The predicted `log10_price` of this house is marked with a black dot. This is where the following two lines intersect:

- The regression line for the condition = 5 homes and
- The vertical dashed black line at `log10_size` equals 3.28, since our predictor variable is the log10 transformed square feet of living space of $\log 10(1900) = 3.28$.

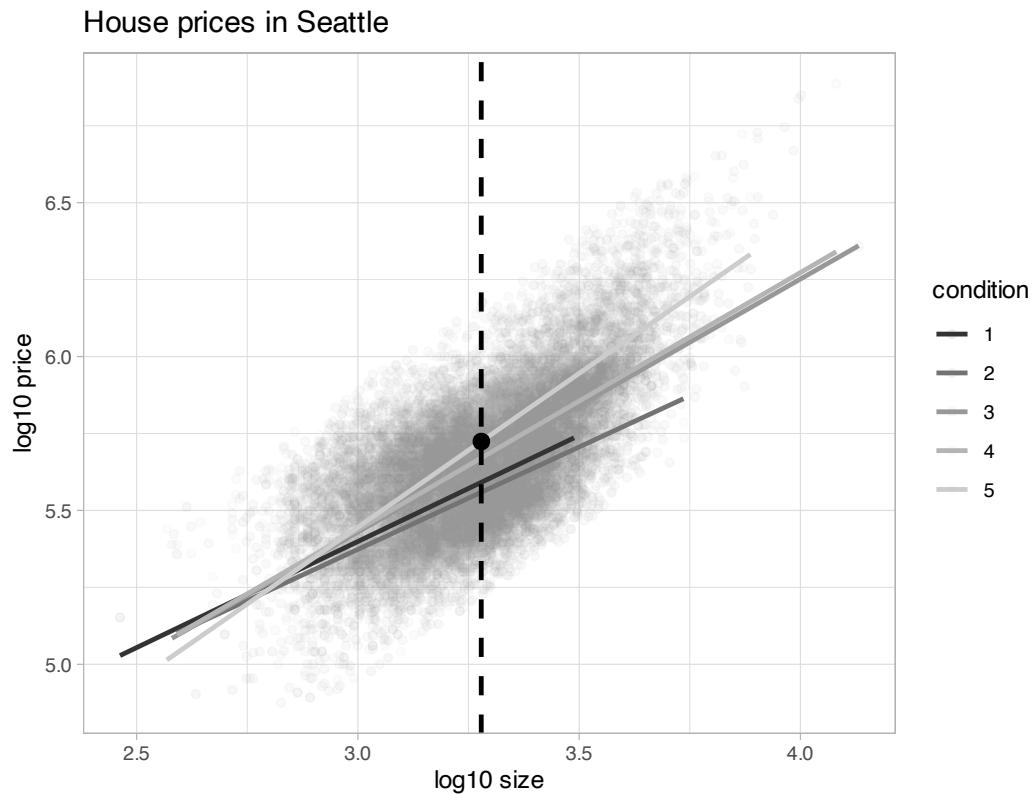


FIGURE 11.8: Interaction model with prediction.

Eyeballing it, it seems the predicted `log10_price` seems to be around 5.75. Let's now obtain the exact numerical value for the prediction using the equation of the regression line for the condition = 5 houses, being sure to `log10()` the square footage first.

```
2.45 + 1 * log10(1900)
```

```
[1] 5.73
```

This value is very close to our earlier visually made prediction of 5.75. But wait! Is our prediction for the price of this house \$5.75? No! Remember that we are using `log10_price` as our outcome variable! So, if we want a prediction in dollar units of `price`, we need to unlog this by taking a power of 10 as described in Appendix A.3.

```
10^(2.45 + 1 * log10(1900))
```

```
[1] 535493
```

So our predicted price for this home of condition 5 and of size 1900 square feet is \$535,493.

Learning check

(LC11.1) Repeat the regression modeling in Subsection 11.2.3 and the prediction making you just did on the house of condition 5 and size 1900 square feet in Subsection 11.2.4, but using the parallel slopes model you visualized in Figure 11.6. Show that it's \$524,807!

11.3 Case study: Effective data storytelling

As we've progressed throughout this book, you've seen how to work with data in a variety of ways. You've learned effective strategies for plotting data by understanding which types of plots work best for which combinations of variable types. You've summarized data in spreadsheet form and calculated summary statistics for a variety of different variables. Furthermore, you've seen the value of statistical inference as a process to come to conclusions about a population by using sampling. Lastly, you've explored how to fit linear regression models and the importance of checking the conditions required so that all confidence intervals and hypothesis tests have valid interpretation. All throughout, you've learned many computational techniques and focused on writing R code that's reproducible.

We now present another set of case studies, but this time on the “effective data storytelling” done by data journalists around the world. Great data stories don’t mislead the reader, but rather engulf them in understanding the importance that data plays in our lives through storytelling.

11.3.1 Bechdel test for Hollywood gender representation

We recommend you read and analyze Walt Hickey’s FiveThirtyEight.com article, “The Dollar-And-Cents Case Against Hollywood’s Exclusion of Women.”⁹ In it, Walt completed a multidecade study of how many movies pass the Bechdel test¹⁰, an informal test of gender representation in a movie that was created by Alison Bechdel.

⁹<http://fivethirtyeight.com/features/the-dollar-and-cents-case-against-hollywoods-exclusion-of-women/>

¹⁰<https://bechdeltest.com/>

As you read over the article, think carefully about how Walt Hickey is using data, graphics, and analyses to tell the reader a story. In the spirit of reproducibility, FiveThirtyEight have also shared the data and R code¹¹ that they used for this article. You can also find the data used in many more of their articles on their GitHub¹² page.

ModernDive co-authors Chester Ismay and Albert Y. Kim along with Jennifer Chun went one step further by creating the `fivethirtyeight` package which provides access to these datasets more easily in R. For a complete list of all 129 datasets included in the `fivethirtyeight` package, check out the package webpage at <https://fivethirtyeight-r.netlify.app/articles/fivethirtyeight.html>.

Furthermore, example “vignettes” of fully reproducible start-to-finish analyses of some of these data using `dplyr`, `ggplot2`, and other packages in the `tidyverse` are available here¹³. For example, a vignette showing how to reproduce one of the plots at the end of the article on the Bechdel test is available here¹⁴.

11.3.2 US Births in 1999

The `US_births_1994_2003` data frame included in the `fivethirtyeight` package provides information about the number of daily births in the United States between 1994 and 2003. For more information on this data frame including a link to the original article on FiveThirtyEight.com, check out the help file by running `?US_births_1994_2003` in the console.

It's always a good idea to preview your data, either by using RStudio's spreadsheet View() function or using `glimpse()` from the `dplyr` package:

```
glimpse(US_births_1994_2003)
```

We'll focus on the number of births for each date, but only for births that occurred in 1999. Recall from Section 3.2 we can do this using the `filter()` function from the `dplyr` package:

¹¹<https://github.com/fivethirtyeight/data/tree/master/bechdel>

¹²<https://github.com/fivethirtyeight/data>

¹³ <https://fivethirtyeight-r.netlify.app/articles/>

¹⁴<https://fivethirtyeightdata.github.io/fivethirtyeightdata/articles/bechdel.html>

```
US_births_1999 <- US_births_1994_2003 %>%
  filter(year == 1999)
```

As discussed in Section 2.4, since `date` is a notion of time and thus has sequential ordering to it, a linegraph would be a more appropriate visualization to use than a scatterplot. In other words, we should use a `geom_line()` instead of `geom_point()`. Recall that such plots are called *time series* plots.

```
ggplot(US_births_1999, aes(x = date, y = births)) +
  geom_line() +
  labs(x = "Date",
       y = "Number of births",
       title = "US Births in 1999")
```

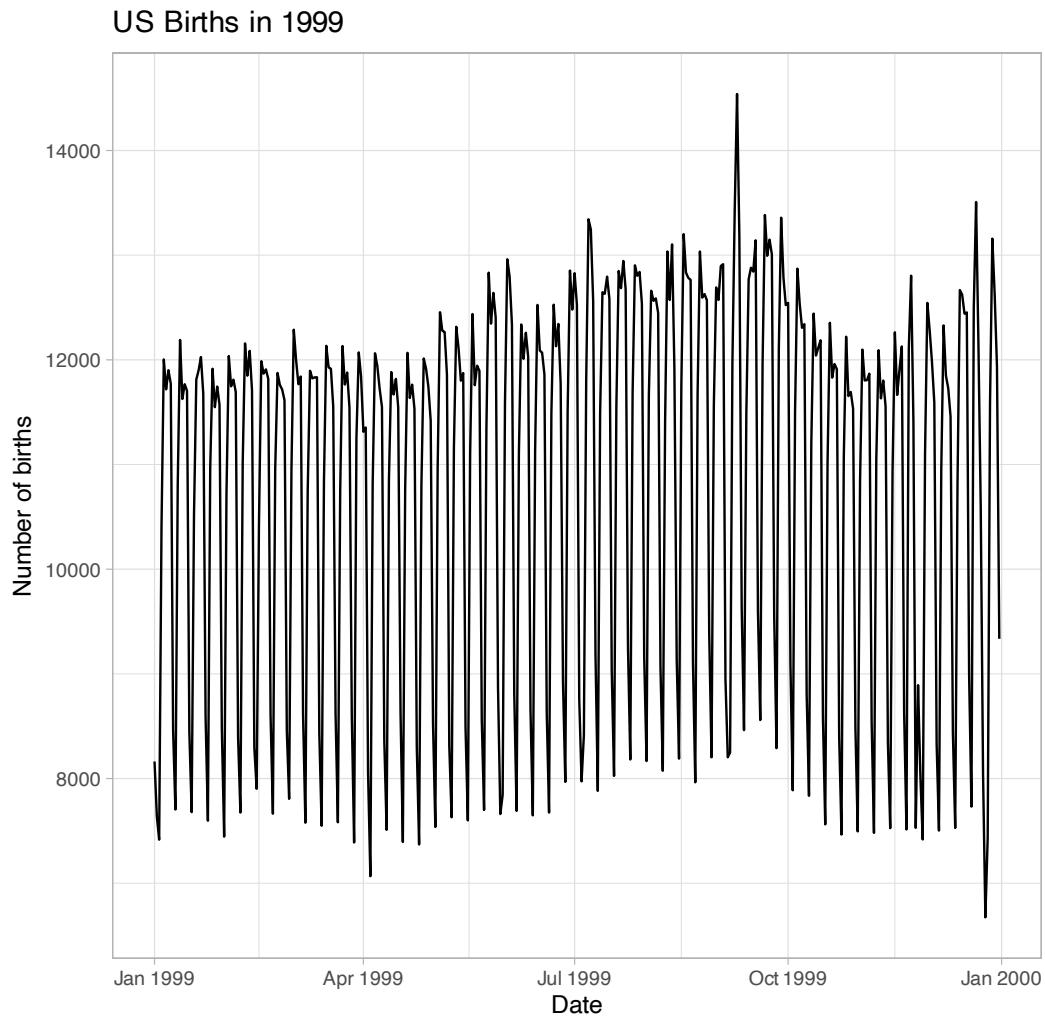


FIGURE 11.9: Number of births in the US in 1999.

We see a big dip occurring just before January 1st, 2000, most likely due to the holiday season. However, what about the large spike of over 14,000 births occurring just before October 1st, 1999? What could be the reason for this anomalously high spike?

Let's sort the rows of `US_births_1999` in descending order of the number of births. Recall from Section 3.6 that we can use the `arrange()` function from the `dplyr` function to do this, making sure to sort `births` in descending order:

```
US_births_1999 %>%
  arrange(desc(births))
```

```
# A tibble: 365 x 6
```

```

      year month date_of_month date      day_of_week births
      <int> <int>       <int> <date>    <ord>      <int>
1 1999     9          9 1999-09-09 Thurs    14540
2 1999    12         21 1999-12-21 Tues    13508
3 1999     9          8 1999-09-08 Wed    13437
4 1999     9         21 1999-09-21 Tues    13384
5 1999     9         28 1999-09-28 Tues    13358
6 1999     7          7 1999-07-07 Wed    13343
7 1999     7          8 1999-07-08 Thurs   13245
8 1999     8         17 1999-08-17 Tues    13201
9 1999     9         10 1999-09-10 Fri     13181
10 1999    12         28 1999-12-28 Tues    13158
# i 355 more rows

```

The date with the highest number of births (14,540) is in fact 1999-09-09. If we write down this date in month/day/year format (a standard format in the US), the date with the highest number of births is 9/9/99! All nines! Could it be that parents deliberately induced labor at a higher rate on this date? Maybe? Whatever the cause may be, this fact makes a fun story!

Learning check

(LC11.2) What date between 1994 and 2003 has the fewest number of births in the US? What story could you tell about why this is the case?

Time to think with data and further tell your story with data! How could statistical modeling help you here? What types of statistical inference would be helpful? What else can you find and where can you take this analysis? What assumptions did you make in this analysis? We leave these questions to you as the reader to explore and examine.

Remember to get in touch with us via our contact info in the Preface. We'd love to see what you come up with!

Please check out additional problem sets and labs at <https://moderndive.com/labs> as well.

11.3.3 Scripts of R code

An R script file of all R code used in this chapter is available at <https://www.moderndive.com/scripts/11-tell-your-story-with-data.R>.

R code files saved as *.R files for all relevant chapters throughout the entire book are in the following table.

chapter	link
1	https://moderndive.com/scripts/01-getting-started.R
2	https://moderndive.com/scripts/02-visualization.R
3	https://moderndive.com/scripts/03-wrangling.R
4	https://moderndive.com/scripts/04-tidy.R
5	https://moderndive.com/scripts/05-regression.R
6	https://moderndive.com/scripts/06-multiple-regression.R
7	https://moderndive.com/scripts/07-sampling.R
8	https://moderndive.com/scripts/08-confidence-intervals.R
9	https://moderndive.com/scripts/09-hypothesis-testing.R
10	https://moderndive.com/scripts/10-inference-for-regression.R
11	https://moderndive.com/scripts/11-tell-your-story-with-data.R

Concluding remarks

Now that you've made it to this point in the book, we suspect that you know a thing or two about how to work with data in R! You've also gained a lot of knowledge about how to use simulation-based techniques for statistical inference and how these techniques help build intuition about traditional theory-based inferential methods like the *t*-test.

The hope is that you've come to appreciate the power of data in all respects, such as data wrangling, tidying datasets, data visualization, statistical/data modeling, and statistical inference. In our opinion, while each of these is important, data visualization may be the most important tool for a citizen or professional data scientist to have in their toolbox. If you can create truly beautiful graphics that display information in ways that the reader can clearly understand, you have great power to tell your tale with data. Let's hope that these skills help you tell great stories with data into the future. Thanks for coming along this journey as we dove into modern data analysis using R and the *tidyverse*!

A

Statistical Background

A.1 Basic statistical terms

Note that all the following statistical terms apply only to *numerical* variables, except the *distribution* which can exist for both numerical and categorical variables.

A.1.1 Mean

The *mean* is the most commonly reported measure of center. It is commonly called the *average* though this term can be a little ambiguous. The mean is the sum of all of the data elements divided by how many elements there are. If we have n data points, the mean is given by:

$$\text{Mean} = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

A.1.2 Median

The median is calculated by first sorting a variable's data from smallest to largest. After sorting the data, the middle element in the list is the *median*. If the middle falls between two values, then the median is the mean of those two middle values.

A.1.3 Standard deviation and variance

We will next discuss the *standard deviation* (*sd*) of a variable. The formula can be a little intimidating at first but it is important to remember that it is essentially a measure of how far we expect a given data value will be from its mean:

$$sd = \sqrt{\frac{(x_1 - \text{Mean})^2 + (x_2 - \text{Mean})^2 + \cdots + (x_n - \text{Mean})^2}{n - 1}}$$

The *variance* of a variable is merely the standard deviation squared.

$$\text{variance} = sd^2 = \frac{(x_1 - \text{Mean})^2 + (x_2 - \text{Mean})^2 + \cdots + (x_n - \text{Mean})^2}{n - 1}$$

A.1.4 Five-number summary

The *five-number summary* consists of five summary statistics: the minimum, the first quantile AKA 25th percentile, the second quantile AKA median or 50th percentile, the third quantile AKA 75th, and the maximum. The five-number summary of a variable is used when constructing boxplots, as seen in Section 2.7.

The quantiles are calculated as

- first quantile (Q_1): the median of the first half of the sorted data
- third quantile (Q_3): the median of the second half of the sorted data

The *interquartile range (IQR)* is defined as $Q_3 - Q_1$ and is a measure of how spread out the middle 50% of values are. The IQR corresponds to the length of the box in a boxplot.

The median and the IQR are not influenced by the presence of outliers in the ways that the mean and standard deviation are. They are, thus, recommended for skewed datasets. We say in this case that the median and IQR are more *robust to outliers*.

A.1.5 Distribution

The *distribution* of a variable shows how frequently different values of a variable occur. Looking at the visualization of a distribution can show where the values are centered, show how the values vary, and give some information about where a typical value might fall. It can also alert you to the presence of outliers.

Recall from Chapter 2 that we can visualize the distribution of a numerical variable using binning in a histogram and that we can visualize the distribution of a categorical variable using a barplot.

A.1.6 Outliers

Outliers correspond to values in the dataset that fall far outside the range of “ordinary” values. In the context of a boxplot, by default they correspond to values below $Q_1 - (1.5 \cdot IQR)$ or above $Q_3 + (1.5 \cdot IQR)$.

A.2 Normal distribution

Let’s next discuss one particular kind of distribution: *normal distributions*. Such bell-shaped distributions are defined by two values: (1) the *mean* μ (“mu”) which locates the center of the distribution and (2) the *standard deviation* σ (“sigma”)

which determines the variation of the distribution. In Figure A.1, we plot three normal distributions where:

1. The solid normal curve has mean $\mu = 5$ & standard deviation $\sigma = 2$.
2. The dotted normal curve has mean $\mu = 5$ & standard deviation $\sigma = 5$.
3. The dashed normal curve has mean $\mu = 15$ & standard deviation $\sigma = 2$.

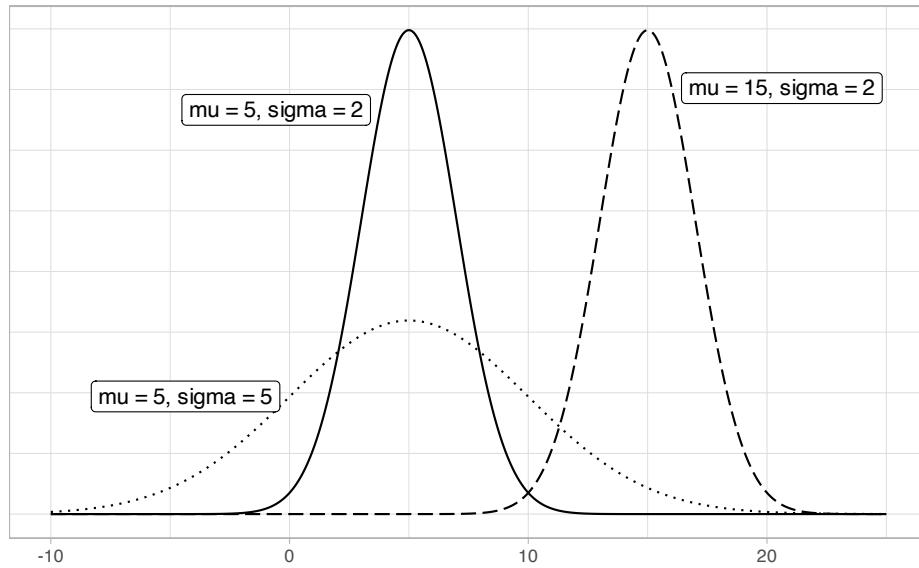


FIGURE A.1: Three normal distributions.

Notice how the solid and dotted line normal curves have the same center due to their common mean $\mu = 5$. However, the dotted line normal curve is wider due to its larger standard deviation of $\sigma = 5$. On the other hand, the solid and dashed line normal curves have the same variation due to their common standard deviation $\sigma = 2$. However, they are centered at different locations.

When the mean $\mu = 0$ and the standard deviation $\sigma = 1$, the normal distribution has a special name. It's called the *standard normal distribution* or the *z-curve*.

Furthermore, if a variable follows a normal curve, there are *three rules of thumb* we can use:

1. 68% of values will lie within ± 1 standard deviation of the mean.
2. 95% of values will lie within $\pm 1.96 \approx 2$ standard deviations of the mean.
3. 99.7% of values will lie within ± 3 standard deviations of the mean.

Let's illustrate this on a standard normal curve in Figure A.2. The dashed lines are at $-3, -1.96, -1, 0, 1, 1.96$, and 3 . These 7 lines cut up the x-axis into 8 segments. The areas under the normal curve for each of the 8 segments are marked and add up to 100%. For example:

1. The middle two segments represent the interval -1 to 1. The shaded area above this interval represents $34\% + 34\% = 68\%$ of the area under the curve. In other words, 68% of values.
2. The middle four segments represent the interval -1.96 to 1.96. The shaded area above this interval represents $13.5\% + 34\% + 34\% + 13.5\% = 95\%$ of the area under the curve. In other words, 95% of values.
3. The middle six segments represent the interval -3 to 3. The shaded area above this interval represents $2.35\% + 13.5\% + 34\% + 34\% + 13.5\% + 2.35\% = 99.7\%$ of the area under the curve. In other words, 99.7% of values.

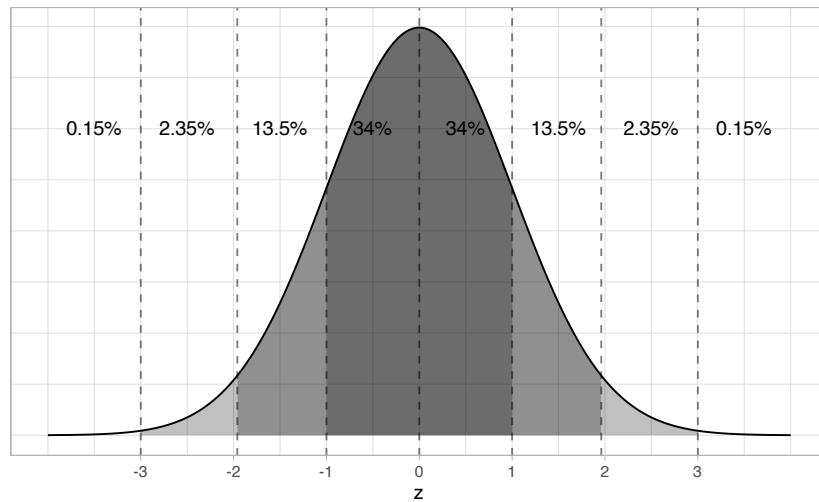


FIGURE A.2: Rules of thumb about areas under normal curves.

Learning check

Say you have a normal distribution with mean $\mu = 6$ and standard deviation $\sigma = 3$.

(LCA.1) What proportion of the area under the normal curve is less than 3? Greater than 12? Between 0 and 12?

(LCA.2) What is the 2.5th percentile of the area under the normal curve? The 97.5th percentile? The 100th percentile?

A.3 log10 transformations

At its simplest, \log_{10} transformations return base 10 *logarithms*. For example, since $1000 = 10^3$, running $\log_{10}(1000)$ returns 3 in R. To undo a \log_{10} transformation, we raise 10 to this value. For example, to undo the previous \log_{10} transformation and return the original value of 1000, we raise 10 to the power of 3 by running $10^{(3)} = 1000$ in R.

Log transformations allow us to focus on changes in *orders of magnitude*. In other words, they allow us to focus on *multiplicative changes* instead of *additive ones*. Let's illustrate this idea in Table A.1 with examples of prices of consumer goods in 2019 US dollars.

TABLE A.1: \log_{10} transformed prices, orders of magnitude, and examples

Price	$\log_{10}(\text{Price})$	Order of magnitude	Examples
\$1	0	Singles	Cups of coffee
\$10	1	Tens	Books
\$100	2	Hundreds	Mobile phones
\$1,000	3	Thousands	High definition TVs
\$10,000	4	Tens of thousands	Cars
\$100,000	5	Hundreds of thousands	Luxury cars and houses
\$1,000,000	6	Millions	Luxury houses

Let's make some remarks about \log_{10} transformations based on Table A.1:

1. When purchasing a cup of coffee, we tend to think of prices ranging in single dollars, such as \$2 or \$3. However, when purchasing a mobile phone, we don't tend to think of their prices in units of single dollars such as \$313 or \$727. Instead, we tend to think of their prices in units of hundreds of dollars like \$300 or \$700. Thus, cups of coffee and mobile phones are of different *orders of magnitude* in price.
2. Let's say we want to know the \log_{10} transformed value of \$76. This would be hard to compute exactly without a calculator. However, since \$76 is between \$10 and \$100 and since $\log_{10}(10) = 1$ and $\log_{10}(100) = 2$, we know $\log_{10}(76)$ will be between 1 and 2. In fact, $\log_{10}(76)$ is 1.880814.
3. \log_{10} transformations are *monotonic*, meaning they preserve orders. So if Price A is lower than Price B, then $\log_{10}(\text{Price A})$ will also be lower than $\log_{10}(\text{Price B})$.
4. Most importantly, increments of one in \log_{10} -scale correspond to *relative multiplicative changes* in the original scale and not *absolute additive changes*. For example, increasing a $\log_{10}(\text{Price})$ from 3 to 4 corresponds to a multiplicative increase by a factor of 10: \$100 to \$1000.

B

Versions of R Packages Used

If you are seeing different results than what is in the book, we recommend installing the exact version of the packages we used. This can be done by first installing the `remotes` package via `install.packages("remotes")`. Then, use `install_version()` replacing the `package` argument with the package name in quotes and the `version` argument with the particular version number to install.

```
remotes::install_version(package = "skimr", version = "1.0.6")
```

package	version
bookdown	0.40.1
broom	1.0.6
dplyr	1.1.4
fivethirtyeight	0.6.2
forcats	1.0.0
gapminder	1.0.0
ggplot2	3.5.1
ggplot2movies	0.0.1
ggrepel	0.9.5
infer	1.0.7
ISLR	1.4
janitor	2.2.0
kableExtra	1.4.0.4
knitr	1.48
lubridate	1.9.3
mvtnorm	1.2-5
nycflights23	0.1.0
patchwork	1.2.0
purrr	1.0.2
readr	2.1.5
scales	1.3.0
sessioninfo	1.2.2
skimr	2.1.5
stringr	1.5.1
tibble	3.2.1
tidyverse	1.3.1
viridis	0.6.5
viridisLite	0.4.2

Bibliography

- Bray, A., Ismay, C., Chasnovski, E., Couch, S., Baumer, B., and Cetinkaya-Rundel, M. (2024). *infer: Tidy Statistical Inference*. R package version 1.0.7.
- Chihara, L. M. and Hesterberg, T. C. (2011). *Mathematical Statistics with Resampling and R*. John Wiley & Sons, Hoboken, NJ, first edition.
- Diez, D. M., Barr, C. D., and Çetinkaya Rundel, M. (2014). *Introductory Statistics with Randomization and Simulation*. CreateSpace Independent Publishing Platform, Scotts Valley, CA, first edition.
- Firke, S. (2023). *janitor: Simple Tools for Examining and Cleaning Dirty Data*. R package version 2.2.0.
- Grolemund, G. and Wickham, H. (2017). *R for Data Science*. O'Reilly Media, Sebastopol, CA, first edition.
- Ismay, C., Couch, S. P., and Wickham, H. (2024). *nycflights23: Flights and Other Useful Metadata for NYC Outbound Flights in 2023*. R package version 0.1.0.
- Ismay, C. and Kennedy, P. C. (2016). *Getting Used to R, RStudio, and R Markdown*.
- Kim, A. Y. and Ismay, C. (2024). *moderndive: Tidyverse-Friendly Introductory Linear Regression*. R package version 0.6.1.9000.
- Kim, A. Y., Ismay, C., and Chunn, J. (2021). *fivethirtyeight: Data and Code Behind the Stories and Interactives at FiveThirtyEight*. R package version 0.6.2.
- Robbins, N. (2013). *Creating More Effective Graphs*. Chart House, New York, NY, first edition.
- Wickham, H. (2014). Tidy data. *Journal of Statistical Software*, Volume 59(Issue 10).
- Wickham, H. (2023). *tidyverse: Easily Install and Load the Tidyverse*. R package version 2.0.0.
- Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., Woo, K., Yutani, H., Dunnington, D., and van den Brand, T. (2024a). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 3.5.1.

- Wickham, H., François, R., Henry, L., Müller, K., and Vaughan, D. (2023). *dplyr: A Grammar of Data Manipulation*. R package version 1.1.4.
- Wickham, H., Hester, J., and Bryan, J. (2024b). *readr: Read Rectangular Text Data*. R package version 2.1.5.
- Wickham, H., Vaughan, D., and Girlich, M. (2024c). *tidyr: Tidy Messy Data*. R package version 1.3.1.
- Wilkinson, L. (2005). *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag, Secaucus, NJ, first edition.
- Xie, Y. (2024). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.40.1.

Index

- Abelson, Hal, [xx](#)
accuracy, [214](#)
adding transparency to plots, [31](#)
- Baggerly, Keith, [xxii](#)
barplot
 faceted, [57](#)
 side-by-side, [57](#)
 stacked, [56](#)
Bechdel, Alison, [398](#)
bias, [219](#)
bivariate, [125](#)
bookdown, [xxvii](#)
Boolean algebra, [5](#)
bootstrap, [240](#)
 colloquial definition, [252](#)
 distribution, [240, 289](#)
 approximation of sampling distribution, [289](#)
 statistical reference, [252](#)
boxplots, [44](#)
 side-by-side, [47](#)
 whiskers, [46](#)
Bryan, Jenny, [383](#)
- categorical, [15](#)
Central Limit Theorem, [222](#)
Cobb, George, [xxiv](#)
colors(), [40](#)
computational reproducibility, [xxii](#)
conditionals, [5](#)
confidence interval, [248](#)
 analogy to fishing, [248](#)
 confidence level, [249](#)
 impact of confidence level on interval width, [276](#)
 impact of sample size on interval width, [277](#)
- interpretation, [274](#)
confounding variable, [148](#)
console, [4](#)
correlation (coefficient), [125, 175, 176](#)
CSV file, [98](#)
- data analysis, [xx](#)
 exploratory, [121](#)
data frame, [5, 12, 97](#)
data science pipeline, [xxiv](#)
data types, [4](#)
degrees of freedom, [339](#)
distribution, [23, 37, 189](#)
 normal, [406](#)
 standard normal, [407](#)
- dplyr
 arrange(), [83](#)
 desc(), [84](#)
 filter, [66](#)
 glimpse(), [15](#)
 group_by(), [74](#)
 inner_join(), [86](#)
 by, [87](#)
 mutate(), [79](#)
 n(), [76](#)
 relocate(), [92](#)
 rename(), [92](#)
 select(), [91](#)
 slice_sample(), [123](#)
 summarize(), [70](#)
 top_n(), [92](#)
 ungroup(), [75](#)
- dummy variable, [144](#)
- explanatory variable, [34](#)
- factors, [5, 47](#)
- five named graphs, [27, 59](#)
- FiveThirtyEight, [383](#)

frequencies, 50
 functions, 5
 argument order, 60
 na.rm argument, 71
 wrapper, 133

 generalizability, 219
 geom_histogram()
 bins, 40
 binwidth, 41
 ggplot2
 +, 29
 aes(), 25
 alpha, 31
 data, 25
 diamonds, 74
 facet, 26
 facet_wrap(), 43, 139
 fill, 40
 geom, 26
 geom_bar(), 51
 geom_col(), 51
 geom_histogram(), 39
 geom_jitter(), 32
 geom_line(), 35, 112
 geom_point(), 33
 geom_smooth(), 128
 ggplot(), 26, 28
 mapping, 28
 position, 26, 57
 GitHub issues, xxvi
 Grammar of Graphics, The, 24
 Grolemund, Garrett, xxi, 82

 heteroskedasticity, 366
 Hickey, Walt, 398
 histograms, 38
 bins, 38
 hypothesis testing, 308
 alternative hypothesis, 308
 hypothesis, 308
 null distribution, 309
 null hypothesis, 308
 observed test statistic, 309
 one-sided alternative, 309

 p-value, 309
 significance level, 310
 test statistic, 309
 tradeoff between alpha and beta, 326
 two-sided alternative, 309
 Type I error, 324
 Type II error, 324
 US criminal trial analogy, 323

 infer
 calculate(), 258, 315
 generate(), 257, 314
 get_confidence_interval(), 261
 get_p_value(), 318
 hypothesize(), 313
 observed statistic shortcut, 254
 rep_sample_n(), 243
 shade_confidence_interval(), 262
 shade_p_value(), 317
 specify(), 255, 312
 switching between tests and
 confidence intervals, 319
 visualize(), 260, 317
 interaction model, 163
 interquartile range (IQR), 46, 406

 joining data
 key variable, 85

 knitr
 kable(), 16

 Lambert, Diane, 382
 levels, 50
 linegraphs, 34
 literate programming, xx
 lm(), 131
 log transformations, 388, 409
 long data format, 103

 margin of error, 295
 mean(), 71
 meta-data, 74, 98, 255
 missing values, 70
 moderndive
 geom_parallel_slopes(), 168

- get_correlation(), 126
- get_regression_points(), 155
- get_regression_table(), 351
- mythbusters_yawn, 278
- pennies_sample, 230
- objects, 4
- observational unit, 15, 122
- offset, 143
- operators, 67
 - ==, 67
 - ?, 18
 - assignment (<-), 62
 - dollar sign, 17
 - in, 68
 - logical, 5
 - not, 68
 - or, 68
 - pipe, 64, 253
- outliers, 49, 387
- overplotting, 30
- p-hacking, 347
- parallel slopes model, 168
- permutation, 307
- pie charts, 53
 - problems with, 54
- plots, 23
- point estimate, 219
- precision, 215
- programming language basics, 4
- quantitative, 15
- R, 1
 - errors, 6
 - formula notation, 126
 - installation, 2
 - messages, 6
 - packages, 7
 - warnings, 6
- R Markdown, xxvii
- R packages, 8
 - broom
 - augment(), 155
 - dplyr, 8
- fivethirtyeight, 100, 399
- ggplot2, 8, 23
- infer, 8
- installation, 9
- ISLR2
 - Credit data frame, 172
- janitor
 - clean_names(), 155
- loading, 10
- loading error, 10
- moderndive, 8, 27, 120, 135
 - tidy_summary(), 124
- nycflights23, 12, 35, 65, 345
- readr
 - read_csv(), 98
- skimr
 - skim(), 386
- tidy, 106
- utils
 - View(), 14
- regression
 - basic, 120
 - conditions for inference (LINE), 358
 - equation of a line, 129
 - intercept, 130
 - slope, 130
 - fitted value, 129
 - interpretation of the slope, 130
 - line, 128
 - linear, 119
 - multiple linear, 160
 - observed values, 146
 - regression plane, 177
 - residual, 132
 - simple linear, 121, 350
- resampling, 235
- residual analysis, 359
- Robinson, David, xxv
- RStudio, 1
 - import data, 100
 - installation, 2
- sample statistic, 219
- sampling, 191, 218
 - census, 218

population, 218
population parameter, 218
random, 219
representative, 219
variation, 191
with replacement, 241
sampling distributions, 212, 289
relationship to sample size, 212
sampling methodology, 209, 219, 221,
280, 352
scatterplots, 27
`sd()`, 71
simulation-based inference, 298
skew, 139, 231, 363, 388
standard deviation, 204
standard error, 212, 240, 354
statistics, xx
sum of squared residuals, 153, 177
summary statistics, 69
tibble, 13
tidy data, 104
tidyverse
 pivot_longer(), 106
 pivot_wider(), 108
time series plots, 35, 400
two-sample inference, 224, 254, 280
two-sample t-statistic, 338
univariate, 125
using `==` instead of `=`, 62
variables
 confounding, 149
 response, 279
 response / outcome / dependent,
 148
 treatment, 148, 279
vectors, 4, 17, 68, 88
Wickham, Hadley, xxi, 82, 104
wide data format, 103
Wilkinson, Leland, 23
Xie, Yihui, xxii, xxvii
z-score, 337