

O'REILLY®

Football Analytics with Python & R

Learning Data Science Through
the Lens of Sports



Early
Release

RAW &
UNEDITED

Eric A. Eager &
Richard A. Erickson

Football Analytics with Python and R

Learning Data Science Through the Lens of Sports

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Eric A. Eager and Richard A. Erickson



Football Analytics with R and Python

by Eric A. Eager and Richard A. Erickson

Copyright © 2023 Eric A. Eager and Richard A. Erickson. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<https://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Michelle Smith

Development Editor: Corbin Collins

Production Editor: Aleeya Rahman

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

October 2023: First Edition

Revision History for the Early Release

- 2022-07-21: First Release
- 2023-03-10: Second Release
- 2023-05-31: Third Release

See <https://oreilly.com/catalog/errata.csp?isbn=9781492099628> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Football*

Analytics with Python and R, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-09956-7

Preface

Life is one about stories. For many, these stories revolve around sports, and football specifically. A lot of us remember playing games in the backyard with our family and friends, while some were lucky enough to play those games under the lights and in front of fans. We love our heroes and and loathe our rivals. We tell these stories to help us understand the past and try to predict the future. Football analytics, at its core, allows us to use information to tell more accurate stories about the game we love. They give us ammunition for some of the game's most vexing questions

For example, do the most championships make the best quarterback? Or the best passing statistics? What do the “best passing statistics” even mean? Are there things that players do that transcend the math? If so, are they simply transcending our current understanding of the game mathematically, only to be shown as brilliant once we have new and better information? Who should your favorite team draft if they are trying to win this year? What about in three years from now? Who should your grandmother draft in fantasy football? Which team should you bet on during your layover in Las Vegas during a football Sunday?

Following the success of the *Moneyball* approach in baseball (captured by Michael Lewis in the 2004 book published by W. W. Norton & Company), people now increasingly use math and statistics to help them properly formulate and answer questions like these.

While football, an interconnected game of 22 players on a 100-yard field, might seem less amendable to statistical analysis than baseball, a game with many independent battles between pitcher and batter, folks have made substantial progress towards the aim of understanding and changing the game for the better. Our aim in this book is to get you started towards understanding. Hopefully some of you will take what you learn in this put and contribute to the great data revolution of the game we all love. Each chapter of this book focuses on a problem in American football, while covering skills to answer the problem.

- Chapter 1 provides an overview of football analytics to this date, including

some problems have been solved within the last few years. It then explores publicly-available play-by-play data to measure aggressiveness by NFL quarterbacks by looking at their average depth of target.

- **Chapter 2** introduces exploratory data analysis (EDA) to examine which subset of quarterback passing data - short passing or long passing - is more stable year-to-year and how to use such analyses to look at regression candidates year-to-year or even week-to-week
- **Chapter 3** uses linear regression to normalize rushing data for NFL ball carriers. Normalizing data helps us adjust for context like how many yards a team needs for a first down, which can affect the raw data outputs that a player produces
- **Chapter 4** expands upon the work in **Chapter 3** to include more variables for which to normalize. For example, down and distance both affect expectation for a ball carrier, and hence both should be included in such a model. You then determine if such a normalization adds to stability in rushing data. Do running backs matter?
- **Chapter 5** shows how to use logistic regression to model quarterbacks completion percentage.
- **Chapter 6** shows how using Poisson regression can help us model game outcomes, and how that applies to the betting markets.
- **Chapter 7** uses web-scraping techniques to obtain NFL draft data since the start of the millennium. You then analyze whether there are teams that are better or worse than expected when it comes to picking players, after adjusting for the expectation of the pick.
- **Chapter 8** uses principle component analysis and clustering to analyze NFL Scouting Combine data to determine player types via unsupervised learning.
- **Chapter 9** describes advanced tools for people wanting to take their analytics game to the next level.

In general, the chapters in the book build upon each other because tools shown in one chapter may be used later. The book also includes three appendices:

- **Appendix A** provides an introduction to Python and R for people who are new to the programs and need direction for installing the programs.
- **Appendix B** provides an introduction to summary statistics and data wrangling using an example to understand passing yards.
- **Appendix C** provides an overview of more data wrangling skills.

To help readers, we have also included a **Glossary** to define terms.

We also use case studies as the focus of this book. **Table P-1** lists the case studies by chapter and what skills the case study solves.

Table P-1. Case studies in this book, technique, and concepts covered.

Case study	Technique	Location
Motivating example: Home field advantage	Example framing a problem	“A Football Example”
Pass depth for quarterbacks	Obtaining NFL data in R and Python	“Example Data: Who Throws Deep?”
Passing yards across seasons	Introduction to stability analysis with EDA	“Player-Level Stability of Passing Yards per Attempt”
Predictor of rushing yards	Building a simple model to estimate rushing yards over expected (RYOE)	“Who Was the Best in RYOE?”
Multiple predictors of rushing yards	Building a multiple regression to estimate RYOE	“Applying Multiple Linear Regression”
Pass completion	Using a logistic regression to	“GLM Application to

percentage	estimate pass completion percentage	Completion Percentage”
Betting on propositions (or props)	Using a Poisson regression to understand betting	“Application of Poisson Regression: Prop Markets”
Quantifying the Jets/Colts 2018 trade	Evaluating a draft trade	“The Jets/Colts 2018 Trade Evaluated”
Evaluating all teams drafting	Comparing drafting outcomes across all NFL teams	“Are Some Teams Better at Drafting Players than Others?”
Player combine attributes	Using multivariate methods to classify player attributes	“Clustering Combine Data”

Who This Book Is For

Our book has two target audiences. First, we wrote the book for people who want to learn about football analytics by *doing* football analytics. We share examples and exercises that help you work through the problems you'd face. While doing this, we show you how we think about football data and then analyze it. You might be a fan who wants to know more about your team, a fantasy football player, somebody who cares about which teams win each week, or an aspiring football data analyst. Second, we wrote this book for people who want an introduction to data science but do not want to learn from classic datasets such as flower measurements from the 1930s or Titanic survivorship tables from 1912. Even if you will be applying data science to widgets at work, at least you can learn using an enjoyable topic like American football.

We assume you have a high school background in math, but are maybe a bit rusty. That is to say, you've completed a pre-calculus course. You might currently be a high school student or somebody who has not had a math course in 30 years. We'll explain concepts as we go. We also focus on helping you see how football can supply a fun math story problems. Our book will help you understand some of the basic skills used daily by football analysts. For fans, this will likely be enough data science skills. For the aspiring football analyst, we hope that our book serves as a springboard for your dreams and lifelong learning.

Who This Book Is Not For

We wrote this book for beginners and have included appendices for people with minimal-to-no prior programming experience. People who have extensive experience with statistics and programming in R or Python would likely not benefit from this book (other than by seeing what kind of introductory problems exist in football analytics). Instead, they should move on more advanced books, such as *R for Data Science* by Wickham and Grolemund to learn more about R or *Python for Data Analysis, 3rd Edition* by McKinney to learn more about Python. Or, maybe you want to move into more advanced books on topics we touch upon in this book, such as multivariate statistics, regression analysis, or R Shiny application. We focus on simple examples rather than complex analysis. We seek to help people get started quickly and connect with real-world data. To use a quote often attributed to Antoine de Saint-Exupéry:

If you wish to build a ship, do not divide the men into teams and send them to the forest to cut wood. Instead, teach them to long for the vast and endless sea.

Thus, we seek quickly connect you to football data, hoping this connection will inspire and encourage you to continue learning tools in greater depth.

How We Think About Data and How to Use This Book

We encourage you to work through this book by not only reading the code, but also running the code, completing the exercises, and then asking your own football questions with data. Besides working through our examples, feel free to add in your own questions and create your own ideas. Consider creating a blog or GitHub page to showcase your new skills or share what you learn. Work through the book with a friend or two. Help each other when you get stuck and talk about what you find with data. The last step is especially important. We regularly think about and fine-tune how we share our datasets as we work as professional data scientists.

In our day jobs, we help people make decisions using data. In this book, we seek to share our tools and thought processes with you. Our formal academic training covered mathematics and statistics, but we did not truly develop our data science skills until we were required to analyze messy, ecological and environmental data from the natural world. We had to clean, merge, and wrangle untidy data all while figuring out what to do with gaps in the information available to us. And then we had to try to explain the meaning hidden within messy datasets.

During the middle of the last decade, Eric starting applying his skills to football, first as a writer and then as an analyst, for a company called Pro Football Focus (PFF). Eventually he left academia to join PFF full time, helping run their first data science group. During his time with PFF he worked with all 32 NFL teams and over 130 college teams, before moving to his new role at SumerSports. Meanwhile, Richard continues to work with ecological data. He helps people make decisions with this data. For example, how many fish need to be harvested from where to control an undesired species?

Although we both have advanced degrees, the ability to think clearly and explore data is more important than formal education. According to a quote attributed to Albert Einstein, “Imagination is more important than knowledge.” We think this holds true for football analytics as well. Asking the right question and finding a good enough answer is more important than what you currently know. Daily, we see how quantitative tools help us to expand our imagination by increasing our ability for looking at and thinking about different questions and datasets. Thus, we are required to imagine important questions to guide our use of analytics.

A Football Example

Let’s say we wanted to know if the Green Bay Packers have a home field advantage. Perhaps we disagree with a friend over whether the Frozen Tundra truly is the advantage that everyone says it is, or if fans are wasting their hard-earned money on an outsized chance of getting frostbite. Conceptually, we take the following steps:

1. We find data to help us answer our question.
2. We wrangle the data into a format to help us answer our question.

3. We explore the data by plotting and calculating summary statistics.
4. We fit a model to help us quantify and confirm our observations.
5. Lastly, and most importantly, we share our results (optionally, but possibly most importantly, we settle the wagers to the “questions” we answered with data).

For the Packers home field advantage example, here are concrete for the steps.

1. We use the `nflreadr` package to obtain data, which is freely available to use.
2. We wrangle the data to give us score differential for each game.
3. We create a plot such as [Figure P-1](#) to help visualize the data.
4. We use a model to estimate average difference point differential for home and away games.
5. We share our results with our friends that we were debating this topic with.

Given the data, the Packers typically have a point differential of 2 points higher at home compare to being on the road. This is in line with where homefield advantage is assumed to be across the league, although this number has evolved substantially over time.

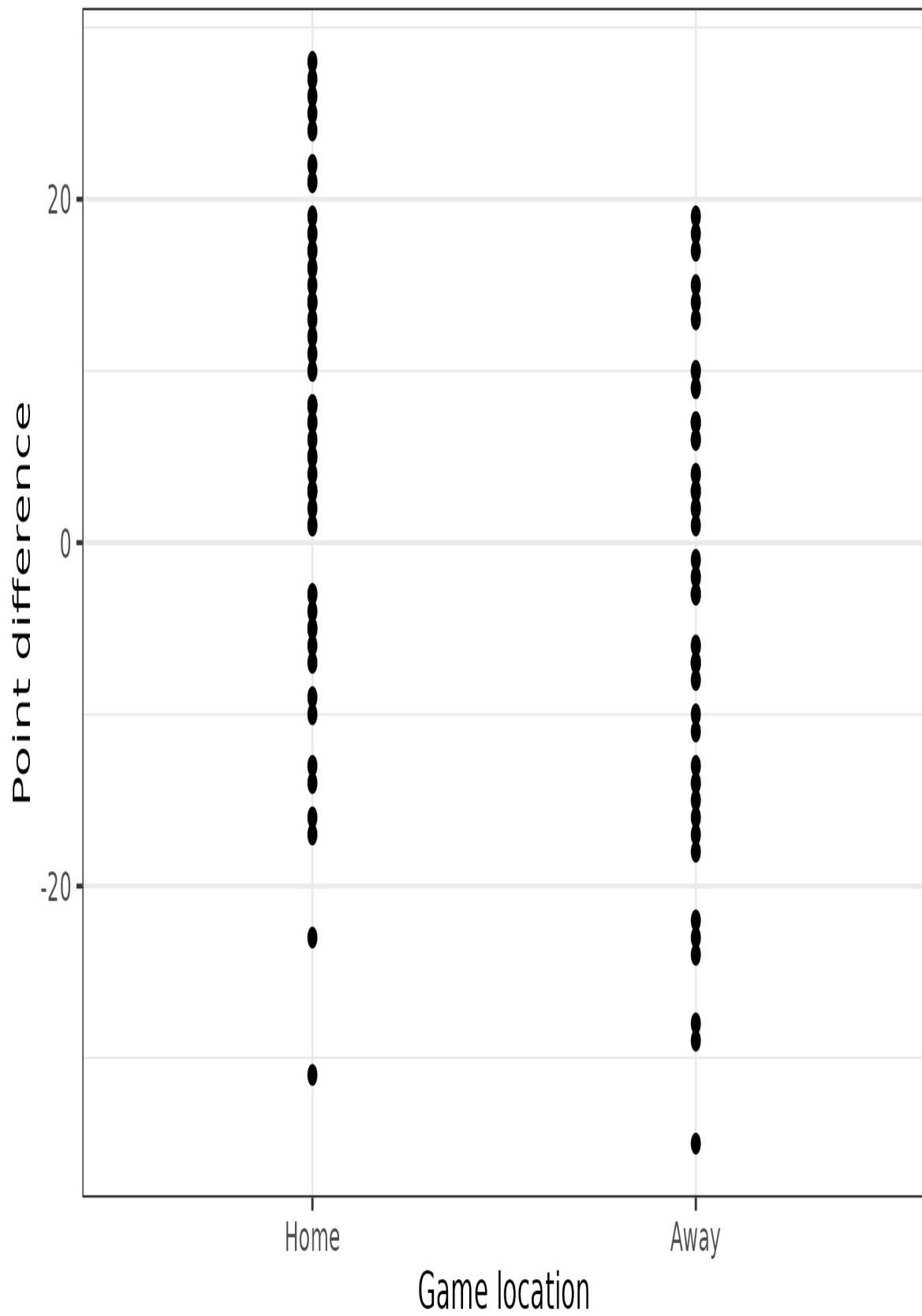


Figure P-1. “Green Bay score differential during home and away games from 2016 to 2022.”

Hopefully this observation raises more questions for you. For example, how much variability exists around this estimate? What happens if you throw out 2020, when fans were not allowed to attend games at the Packers’ home stadium? Does homefield advantage affect the first half or second half of games more? How do you adjust for quality of play and schedule differential? How does travel distance affect homefield advantage? Familiarity? Both (since familiarity and proximity are related)? How does weather differential affect homefield advantage? How does homefield advantage affect winning games compared to losing games?

Let’s quickly look at homefield advantage during games won by the Packers compared to games lost. Look at [Figure P-2](#). Summaries for the data in this figure show that the Packers lose by one point more on the road compared to home games, but, win at home by almost 6 points more than they do on the road.

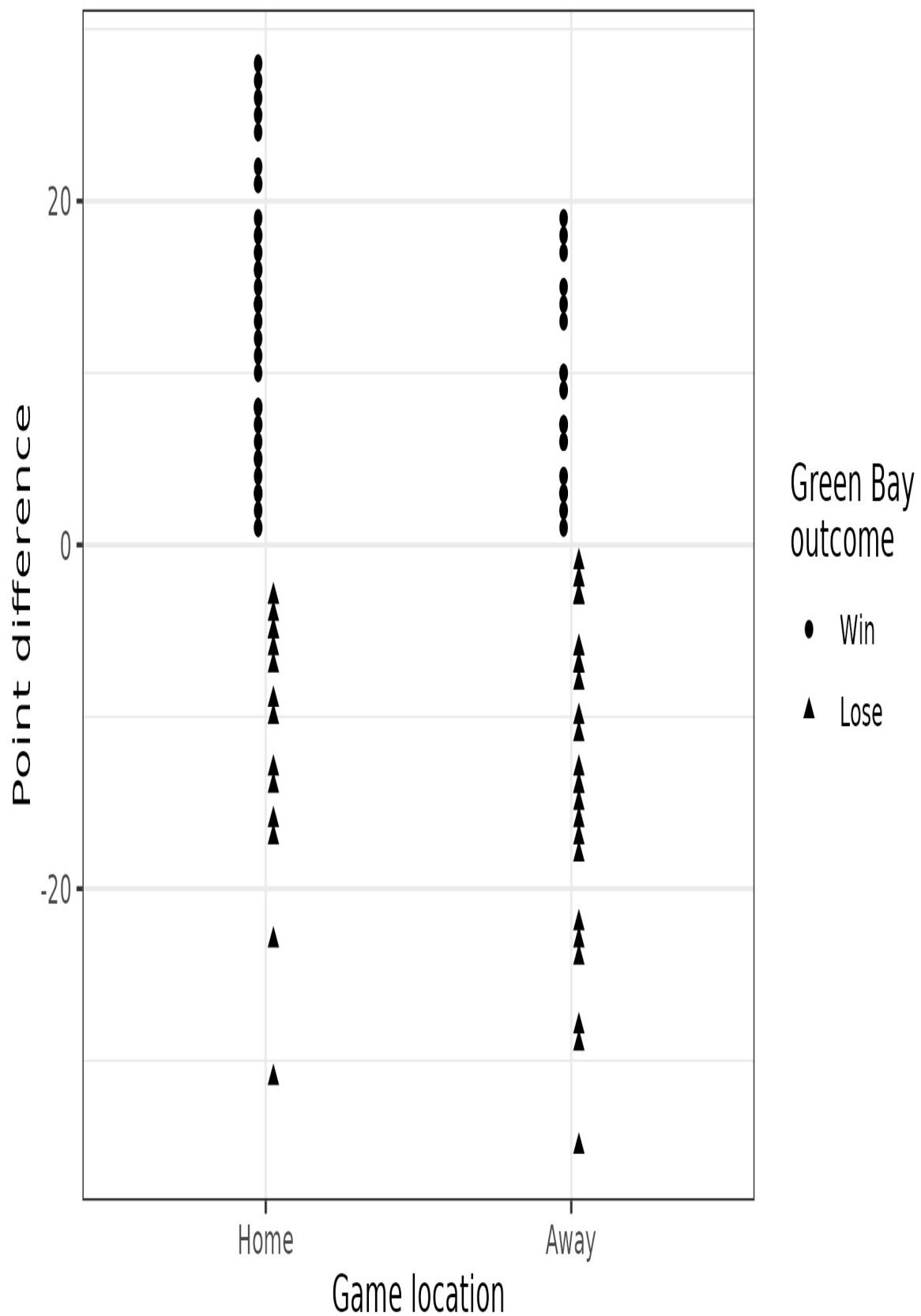


Figure P-2. “Green Bay score differential during home and away games from 2016 to 2022 during winning and losing games.”

That being said, the Green Bay Packers are a good team and will likely have a positive point differential no matter what. The question of homefield advantage is actually not a trivial one and has perplexed analysts and bettors alike for decades. Hopefully this example, spurs more questions for you. If so, you’re reading the right book.

We cover step 1, obtaining football data, in [Chapter 1](#). We cover step 2, exploring data, in [Chapter 2](#) on Exploratory Data Analysis. We cover step 3, data wrangling, in case studies throughout the book as well as in [Appendix B](#) and [Appendix C](#). We cover step 4 with basic statistics in [Chapter 2](#) and [Appendix B](#) and then cover models in [Chapter 3](#), [Chapter 4](#), [Chapter 5](#), [Chapter 6](#), [Chapter 7](#), and [Chapter 8](#). We cover step 5 in through various chapters as we describe what we have found. Lastly, we round out the book with [Chapter 9](#) that describes some of the advanced tools we use daily.

What You Will Learn from Our Book

We have included materials in this book to help you start your journey into football analytics. For enthusiastic fans, our book may be enough to get you up and running. For people aspiring to be quantitative football analysts, our book will hopefully serve as a springboard. For people seeking to become or improve their data science skills, our book provides worked examples of how data can be used to answer questions. We specifically teach the following:

- How to visualize data
- How to summarize data
- How to model data
- How to present the results of data analysis
- How to use the previous methods to tell a story

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at https://github.com/oreillymedia/title_title.

If you have a technical question or a problem using the code examples, please send email to bookquestions@oreilly.com.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning

NOTE

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <https://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at

<https://learning.oreilly.com/library/view/football-analytics-with/9781492099611>.

Email bookquestions@oreilly.com to comment or ask technical questions about this book.

For news and information about our books and courses, visit <https://oreilly.com>.

Find us on Facebook: <https://facebook.com/oreilly>

Follow us on Twitter: <https://twitter.com/oreillymedia>

Watch us on YouTube: <https://www.youtube.com/oreillymedia>

Acknowledgments

We thank our editors at O'Reilly including Michelle Smith, Corbin Collins, and Clare Laylock for their support. We thank Nick Adams and Danny Elfanbaum for technical assistance with the O'Reilly Atlas system. We thank the Boyan Angelov, Richie Cotton, Ryan Day, Chester Ismay, Kaelen Medeiros, George Mound, John Oliver, and Tobias Zwingmann for technical feedback. We also thank Richie Cotton for tips on writing a successful book proposal.

Eric would like to thank his wife, Stephanie, for her patience and constant support of his dreams, regardless of how crazy. He would also like to than Neil Hornsby, whose vision in building PFF gave him a platform that been the foundation for everything he's done until now, and Thomas Dimitroff, for answering his email back in the fall of 2020. He'd also like to thank Paul and Jack Jones for their vision in starting SumerSports in 2022. Finally, Eric would like to thank his parents, who, despite not being huge football fans themselves, kindled the flames of his passion throughout his childhood.



Figure P-3. “Margo (left) and Sadie (right) walking. Who is walking whom? Photo credit Richard Erickson.”

Richard Erickson thanks his daughter Margo for sleeping well so he could write this after she went down for the night. He also thanks Sadie for foregoing walks while he was writing as a well as patiently, and impatiently reminding him to take stretch breaks ([Figure P-3](#)). Richard also thanks his parents for raising him with curiosity and his brother for prodding him to learn to program and for help with this book’s proposal. Lastly, Richard thanks Tom Horvath and other from Hale, Skemp, Hanson, Skemp, and Sleik for support while writing this book.

Chapter 1. Football Analytics

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the first chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

American football (also known as *gridiron football* or *North American football* and henceforth simply called *football*) is undergoing a drastic shift towards the quantitative. Prior to the last half of a decade or so, most of football analytics was confined to a few seminal pieces of work. Firstly, former Brigham Young, Chicago Bears, Cincinnati Bengals, and San Diego Chargers quarterback Virgil Carter created the notion of an *expected point* in his 1971 paper “[On Operations Research in Football](#)” - before he teamed with the legendary Bill Walsh as the first quarterback to execute what is now known as the West Coast Offense.

The idea of an *expected point* is incredibly important in football, as the game by its very nature is discrete - a collection of a finite number of plays (also called *downs*) which require the offense to go a certain distance (in yards) before they have to surrender the ball to the opposing team. If the line to gain is the opponent’s end zone, the offense scores a touchdown, which is worth, on average, seven points after a post-touchdown conversion. Hence, the *expected point* provides an estimated, or *expected* value for how many *points* one would expect a team to score given the current game situation on that drive.

Football statistics have largely been confined to offensive players, and have been doled out in the currency of yards and touchdowns gained. The problem with

this is obvious. If a player catches a pass to gain seven yards, but eight is required to get a first down or a touchdown the player did not gain at first down. Conversely, if a player gains five yards, when five is required to get a first down or a touchdown the player gained a first down. Hence, “enough” yards can be better than “more” yards depending upon the context of the play. As a second example, if it takes a team two plays to travel 70 yards to score a touchdown, with one player gaining the first 65 yards and the second gaining the final five, why should the second player get all of the credit for the score?

In 1988, Bob Carroll, Pete Palmer, and John Thorn wrote *The Hidden Game of Football* (Grand Central Pub), which further explored the notions of expected points. In 2007, Brian Burke, who was a Navy pilot before creating the website advancedfootballanalytics.com, formulated the expected points and expected points added approach, along with building a win probability model responsible for some key insights, including the **4th Down Bot** at the *New York Times* website. Players may be evaluated by how many expected points or win probability points were added to their teams when they did things like throw or catch passes, run the ball, or sack the quarterback.

The work of Burke inspired the open-source work of Ron Yurko, Sam Ventura, and Max Horowitz of Carnegie Mellon. The trio built `nflscrapr`, an R package that scraped NFL play-by-play data. `nflscrapr` was built to display their own versions of *expected points added* (EPA) and *win probability* (WP) models. Using this framework they also replicated the famous *wins above replacement* (WAR) framework from baseball for quarterbacks, running backs, and wide receivers, which was published in 2018. This work was later extended using different data and methods by Eric and his collaborator George Chahouri in 2020. Eric’s version of WAR, and its analogous model for college football, are used throughout the industry to this day.

`nflscrapr` served as a catalyst for the popularization of modern tools to study football using data, most of which use a framework that will be replicated constantly throughout this book. The process of building an *expectation* for an outcome - in the form of points, completion percentage, rushing yards, draft pick outcome, and many more - and measuring players or teams via the residual is a process that transcends sports. In soccer, for example, expected goals (or xG) are the cornerstone metric upon which players and clubs are measured in *The*

Beautiful Game. While shot quality - the expected rate at which a shot is made in basketball - is a ubiquitous measure for players and teams on the hardwood. The features that go into these models, and the forms that they take, are the subject of constant research whose surface we will scratch in this book.

The rise of tools like `nflscrapr` allowed more people to show their analytical skills and flourish in the public sphere. Analysts were hired based upon their tweets on Twitter because of their ingenious approaches to measuring team performance. Decisions to punt or go for it on a fourth down were evaluated by how they affected the team's win probability. Ben Baldwin and Sebastian Carl created a spin-off R package, `nflfastR`. `nflfastR` updated the models of Yurko, Ventura, and Horowitz, along with adding many of their own - models that we'll use in this book. More recently, the data contained in the `nflfastR` package has been cloned into Python via the `nfl_data_py` package by Cooper Adams.

We hope that this book will give you the basic tools to approach some of the initial problems in football analytics, and serve as a jumping off point for future work.

TIP

People looking for the cutting edge of sports analytics, including football, may want to check out the MIT Sloan Sports Analytics Conference. Since its founding in 2006, Sloan has emerged as a leading venue for the presentation of new tools for football (and other) analytics. Other, more accessible conferences, like the Carnegie Mellon Sports Analytics Conference and New England Statistics Symposium, are fantastic places for students and practitioners to present their work. Most of these conferences have hackathons for people looking to make an impression on the industry.

Baseball Has the Three True Outcomes: Does Football?

Baseball pioneered the use quantitative metrics and the creation of Society for American Baseball Research (SABR) led to the term *sabermetrics* to describe baseball analysis. Because of this long history we start by looking at the metrics commonly used in baseball, specifically, the *three true outcomes*. One of the reasons the game of baseball has trended towards the *three true outcomes*

(walks, strikeouts, and home runs) is that they were the easiest to predict from one season to the next. What batted balls did when they were in play was more noisy, and was the source of much of the variance in perceived play from one year to the next. The three true outcomes of baseball have also been the source for more elaborate data collection methods and subsequent analysis in an attempt to tease additional signal from batted-ball data.

Stability analysis is a cornerstone of team and player evaluation. *Stable* production is sticky or repeatable, and is the kind of production decision makers should want to buy into year in and year out. *Stability analysis* therefore examines if something is stable, in our case football observation and model outputs, and you will use this analysis in “[Player-Level Stability of Passing Yards per Attempt](#)”, “[Is RYOE a Better Metric?](#)”, “[Analyzing RYOE](#)” and “[Is CPOE More Stable Than Completion Percentage?](#)”. On the other hand, how well a team or player does in *high-leverage situations* (that is to say, plays that have greater effects on the outcome of the game such as converting third downs) can have an outsized impact on win-loss record or eventual playoff fate, but if it doesn’t help us predict what we want year in and year out, it might be better to ignore, or sell such things to other decision makers.

Using play-by-play data from `nflfastR`, [Chapter 2](#) shows you how to slice and dice football passing data into subsets that partition a player’s performance into stable and unstable production. Through exploratory data analysis techniques, you can see if any players break the mold, and what to do with them. We preview how this work can aid in the process of feature engineering for prediction.

Do Running Backs Matter?

For most of the history of football, the best players played running back (in fact, early football didn’t include the forward pass until President Teddy Roosevelt worked with college football to introduce passing to make the game safer in 1906). The importance of the running back used to be an accepted truism across all levels of football, until the forward pass became an integral part of the game. Following the forward pass, rule and technology changes - along with Carter (mentioned earlier in this chapter) and his coach, Walsh - made throwing the

football more efficient relative to running the football.

Many of our childhood memories from the 1990s revolve around Emmitt Smith and Barry Sanders trading the privilege of being the NFL rushing champion every other year. College football fans from the 1980s may remember Herschel Walker giving way to Bo Jackson in the SEC conference. Even many younger fans from the 2000s and 2010s can still remember Adrian Peterson earning the last non-quarterback most valuable player (MVP) award. During the 2012 season, he rushed for over 2,000 yards while carrying an otherwise-bad Vikings team to the playoffs.

However, the current prevailing wisdom among football analytics folks is that the running back position does not matter the same way that other positions do. This is for a few reasons. First, running the football is not as efficient as passing. This is plain to see with simple analyses using yards per play, but also through more advanced means like expected points added. Even the worst passing plays usually produce, on average, more yards or expected points per play than running.

Second, differences in the actual player running the ball does not elicit the kind of change in rushing production similar differences do for quarterbacks, wide receivers, offensive or defensive linemen. In other words, additional resources used to pay for the services of this running back over that running back are probably not worth it, especially if they can be used on other positions. The marketplace that is the NFL has provided additional evidence that this is true, as we have seen running back salaries and draft capital used on the position in the draft decline to lows not previously seen.

This didn't keep the New York Giants from using the second-overall pick in the 2018 NFL Draft on Penn State's Saquon Barkley, which was met with jeers from the analytics community, and a counter from Giants General Manager Dave Gettleman. In a post-draft press conference for the ages, Gettleman, sitting next to reams of binded paper, made fun of the analytics jabs towards his pick by mimicking a person typing furiously on a typewriter.

[Chapter 3](#) and [Chapter 4](#) look at methods for controlling play-by-play rushing data for a situation to see how much of the variability in rushing success has to do with the player running the ball.

How Data Can Help Us Contextualize Passing Statistics

As stated above, the passing game dominates football, and in [Appendix B](#) we show you how to examine the basics of passing game data. In recent years analysts have taken a deeper look into what constitutes accuracy at the quarterback position - as raw completion percentage numbers, even among quarterbacks who aren't considered elite players, have skyrocketed. The work of Josh Hermsmeyer with the Baltimore Ravens and later [fivethirtyeight](#) established the significance of *air yards*, which is the distance traveled by the pass from the line of scrimmage to the intended receiver.

While Hermsmeyer's initial research was in the fantasy football space, it spawned a significant amount of basic research into the passing game, giving rise to metrics like *completion percentage over expected* (CPOE), which is one of the most predictive quarterback metrics about quarterback quality available today.

In [Chapter 5](#) we introduce generalized linear models in the form of logistic regression. You'll use this to estimate the completion probability of a pass given multiple situational factors that affect a throw's expected success. We then look at a player's residuals and see if there is more or less stability in the residuals - the CPOE - than in actual completion percentage.

Can You Beat the Odds?

In 2018 the Professional and Amateur Sports Protection Act of 1992 (or *PASPA* for short; the federal ban on sports betting in the United States - outside of the state of Nevada) was repealed. This Act of Congress opened the floodgates for states to make legal what many people were already doing illegally - betting on football.

The difficult thing about sports betting is the house advantage - referred to as the *vigorish* or *vig* - makes it so that a bettor has to win more than 50% of his or her bets to break even. Thus, a cost exists for simply playing the game that needs to be overcome in order to beat the sportsbook (or simply the *book* for short).

American football is the largest market in North America. Most successful sports

bettors in this market use some form of analytics to overcome this house advantage. [Chapter 6](#) examines the passing touchdowns per game prop market show how a bettor would arrive at an internal price for such a market, and compare it to the market price.

Do Teams Beat the Draft?

Owners, fans, and the broader NFL community evaluate coaches and general managers based on the quality of talent that they bring into their team from one year to the next. One complaint against Coach Bill Belichick, maybe the best non-player coach in the history of the NFL, in recent seasons is that he has not drafted well. Has that been a sequence of fundamental missteps or is it just a run of bad luck?

One argument in support of coaches such as Belichick may be “Well, they are always drafting in the back of the draft, since they are usually a good team.” Luckily, one can use math to control for this, and to see if we can reject the hypothesis that all front offices are equally good at drafting after accounting for *draft capital* used. *Draft capital* accounts for the resources used during the NFL draft, notably, the number of picks, pick rounds, and pick numbers.

In [Chapter 7](#) we scrape publicly-available draft data and test the hypothesis that all front offices are equally good at drafting after accounting for draft capital used, with surprising results. In [Chapter 8](#) we scrape publicly-available combine data and use dimension reduction tools and clustering to see how groups of players emerge.

Tools for Football Analytics

Football analytics, and more broadly, data science, require a diverse set of tools. Successful practitioners in these fields require an understanding of these tools. Statistical programming languages, like Python and R, are a backbone of our data science toolbox. These languages allow us to clean our datasets, conduct our analyses, and readily reuse our methods. Although many people commonly use spreadsheets (such as Microsoft Excel or Google Sheets) for data cleaning and analysis, we find spreadsheets do not scale well. For example, when one has

to work with large datasets like tracking data, which can contain thousands of rows of data per play, spreadsheets simply are not up to the task. Likewise, people commonly use Business Intelligence (BI) tools such Power BI and Tableau because of their power and ability to scale. But these tools tend to focus on point-and-click methods and require licenses, especially for commercial use.

Programming languages also allow for easy reuse because copy and pasting formulas in spreadsheets can be tedious and error prone. Lastly, spreadsheets (and, more broadly, point-and-click tools) allow undocumented errors. For example, spreadsheets do not have a method to catch a copying and pasting mistake. Furthermore, modern data science tools allow code, data, and results to be blended together in easy-to-use interfaces. Common languages include Python, R, Julia, Matlab, and SAS. Additional languages continue to appear as computer science advances forward.

As practitioners of data science, we use R and Python daily for our work, which has collectively spanned the space of applied mathematics, applied statistics, theoretical ecology and, of course, football analytics. Of the languages listed previously, Python and R offer the benefit of larger user bases (and hence likely contain the tools and models we need). Both R and Python (as well as Julia) are open source. As of this writing Julia does not have the user base of R or Python, but may either be the cutting edge of statistical computing, a dead end that fizzles out, or possibly both.

Open source means two types of freedom. First anybody can access all the code in the language, like free speech. This allows volunteers to help improve the languages, such as ensuring that users can debug the code and extend the languages through add-on packages (like the `nflfastR` package in R or the `nfl_py_data` package in Python). Second, open source also offers the benefit of free to use for users, like free drinks. Hence users do not need to pay thousands of dollars annually in licensing fees. We were initially trained in R, but have learned Python over the course of our jobs. Either language is well suited for football analytics (and sports analytics in general).

NOTE

[Appendix A](#) includes instructions for obtaining R and Python for readers who do not currently have access to these languages. This includes either downloading and installing the programs or using web-

hosted resources. The appendix also describes programs to help you more easily work with these languages such as editors and integrated development environments (IDEs).

We encourage you to pick one language for the book and learn that language well. Should you need to learn a second programming language, it is easier if you understand the programming concepts behind a first language well. Then you can relate the concepts back to your understanding of your original computer language.

TIP

For readers who want to learn the basic of programming before proceeding with our book, we recommend Al Swigart's *Invent Your Own Computer Games with Python, 4th edition* (No Starch Press, 2016) or Garrett Grolemund's book, *Hands-On Programming with R* (O'Reilly, 2014). Either of resources will hold your hand to help you learn the basics of programming.

Although many people pick favorite languages and sometimes have arguments with each other over which coding language is better (similar to Coke versus Pepsi or Ford versus General Motors), we have seen both R and Python used in production and also used with large data and complex models. For example, we have used R with 100 GB files on servers with sufficient memory. Both of us began our careers coding almost exclusively in R, but have learned to use Python when the situation has called for it. Furthermore, the tools often have complementary roles, especially for advanced methods, and knowing both languages lets you have options for problems you may encounter.

TIP

When picking a language, we suggest you *use what your friends use*. If all your friends speak Spanish, it's probably going to be easier for you to communicate with them if you learn Spanish as well. You can then teach them your native language too. Likewise, the same holds for programming. Your friends can then help you debug and troubleshoot. If you still need help deciding, open up both languages and play around for a little bit. See which one you like better. Personally, we like R when working with data, because of R's data manipulation tools, and Python when building and deploying new models because of Python's cleaner syntax for writing functions.

First Steps in Python and R

TIP

Readers familiar with R and Python still would still benefit from skimming this section to see how we teach a tool you are familiar with.

Opening a computer terminal may be intimidating for many people. For example, many of our friends and family will walk by our computers, see code up on the screens, and immediately turn their heads in disgust (Richard's dad) or fear (most other normal people). However, terminals are quite powerful and allow more to be done with less, once you learn the language. This section will help you get started using Python or R.

The first steps for using R or Python are to install them on your computer or to use a web-based version of the programs. Different options exist for installing or otherwise accessing Python and R and then using them on your computer.

[Appendix A](#) contains steps for this and some different installation options.

NOTE

People, like Richard, who follow the Green Bay Packers are commonly called *Cheeseheads*. Likewise, people who use Python are commonly called *Pythonistas*, and people who use R are commonly called *useRs*.

Once you have access to R or Python, you have an expensive graphing calculator (for example, your \$1,000 laptop). In fact both Eric and Richard, in lieu of using an actual calculator, will often calculate silly things like point spreads or totals in the console if in need of a quick calculation. Let's see some things you can do. Type `2+2` in either the Python or R console:

```
2 + 2  
4
```

NOTE

People use comments to leave notes to themselves and others in code. Both Python and R use the `#` symbol for comments (the *pound symbol* for the authors or *hash-tag* for younger readers). Comments are code that the computer does not read, but help humans to understand the code. In this book, we will use two comment symbols to tell you if a code block is Python (`## Python`) or R (`\## R`)

You may also save numbers as variables. In Python, you could define `z` to be 2 and then re-use `z` and divide by 3:

```
## Python
z = 2
z / 3
0.6666666666666666
```

TIP

In R, either `\<\-` or `=` may be used to create variables. We use `\<\-` for two reasons. First in this book this helps you see the difference between R and Python code. Second we use this style in our day-to-day programming as well. [Chapter 9](#) discusses code styles more. Regardless of which operator you use, be consistent with your programming style in any language. Your future self (and others who read your code) will thank you.

In R, you can also define `z` to be 2 and then re-use `z` and divide by 3.

```
## R
z <- 2
z / 3
[1] 0.6666667
```

NOTE

Python and R format outputs differently. Python does not round up and includes more digits. Conversely, R shows fewer digits and rounds up.

Example Data: Who Throws Deep?

Now that you have seen some basics in R, let's dive into an example with football data. You will use the `nflfastR` data for many of the examples in this

book. This data may be installed as a an R package, or the Python package `nfl_data_py`. Specifically, we will explore the broad (and overly simple) question: “Who were the most aggressive quarterbacks in 2021?” We will start off introducing the package using R because the data originated with R.

NOTE

Both Python and R have flourished because they readily allow add-on packages. Conda exists as one tool for managing these add-ons. [Chapter 9](#) and [Appendix A](#) discusses these add-ons in greater detail. In general, packages in Python can be installed by typing `pip install <package name>` or `conda install <package name>` in the terminal such as Bash shell on Linux, the zsh shell on macOS, or command prompt on Windows. Sometimes, you will need to use `pip3`, depending upon your operating system’s configuration if you are using the `pip` package manager system. For a concrete example, to install the `seaborn` package, you could type `pip install seaborn` in your terminal. In general, packages in R can be installed by opening R and then typing `install.packages("<package name>")`. For example, to install the `tidyverse`, open R and run `install.packages("tidyverse")`

nflfastR in R

Starting with R, install the `nflfastR` package:

```
## R  
install.packages("nflfastR")
```

TIP

Using single quotes around a name, such as `'x'`, or double quotes, such as `"x"`, are both acceptable to languages such as Python or R. Make sure the open and close quotes match. For example, `'x"` would not be acceptable to the languages. You may use both single and double quotes to place quotes inside of quotes. For example, in a figure caption you might write `"Panthers' points earned"` or `'Air temperature ("true temperature")'`. Or, in Python, you can use a combination of quotes later for inputs such as `"team == 'GB'"` because you need to nest quotes inside of quotes.

Next, load this package as well as the `tidyverse`, which gives you tools to manipulate and plot the data:

```
## R  
library("tidyverse")
```

```
library("nflfastR")
```

NOTE

Base R contains data frames as `data.frame()`. We use Tibbles from the Tidyverse instead, because these print nicer to screens and include other useful features. Many users consider Base R's `data.frame()` to be a legacy object, although you will likely when looking at help files and examples on the web. Lastly, you might see the `data.table` package in R. The `data.table` extension of data frames is similar to a Tibble and work better with larger data (for example, 10 or 100GB files) and have a more compact coding syntax, but come with the trade-off of being less user friendly compared to Tibbles. In our own work, we use these when we need high performance at the trade-off of code readability.

Once you've loaded the packages, load the data from each play, or the *play-by-play* (pbp) data, for the 2021 season using the `load_pbp()` function from `nflfastR` and call the data `pbp_r` (the `_r` ending helps you tell that the code is from an R example in this book):

```
## R
pbp_r <- load_pbp(2021)
```

NOTE

We generally include `_py` in the name of Python data frames and `_r` in the names of R data frames to help you see the language for different code objects.

After loading the data as `pbp_r`, pass (or *pipe*) the data along to be “filtered” using `|>`. Filter the data using the `filter()` function by only selecting data where passing plays (`play_type == "pass"`) occurred and where `air_yards` are not missing, or `NA` in R syntax (in plain English, the pass had a recorded depth). [Chapter 2](#), [Appendix B](#), and [Appendix C](#) cover data manipulation more, and all most all examples in this book use data wrangling to format data. So right now, simply type this code. You can probably figure out what the code is doing, but do not worrying about understanding it too much:

```
## R
pbp_r_p <-
  pbp_r |>
```

```
filter(play_type == 'pass' & !is.na(air_yards))
```

Now look at the average depth of target for every quarterback in the NFL in 2021 who threw 100 or more passes with a designated depth. To avoid multiple players who have the same name, which happens more than you'd think, summarize by both player id and player name. With this, "group by" both the `passer_id` and `passer`. Then "summarize" to calculate the number of plays (`n()`) and mean air yards per pass (also known as the *average depth of target* or `adot`) per player. Also, "filter" to only include players with 100 or more plays as well as removing any rows without a passer name (specifically those with missing or NA values).

With this and the previous example commands, the function `is.na(passer)` checks to see if a value in the `passer` column has the value NA and returns TRUE for columns with an NA value. [Appendix B](#) covers this logic and terminology in greater detail. Next, an exclamation point (!) turns this expression into the opposite of "not missing value", so that you keep cells with a value. As an aside, we, the authors, find the use of double negatives confusing as well. Lastly, arrange by the `adot` values and then print all (or infinity, `Inf`) values:

```
## R
pbp_r |>
  group_by(passer_id, passer) |>
  summarize(n = n(), adot = mean(air_yards, na.rm = TRUE)) |>
  filter(n >= 100 & !is.na(passer)) |>
  arrange(-adot) |>
  print(n = Inf)

# A tibble: 45 × 4
# Groups:   passer_id [45]
  passer_id  passer          n    adot
  <chr>      <chr>       <int> <dbl>
1 00-0035704 D.Lock        137  10.2
2 00-0029263 R.Wilson     482   9.89
3 00-0036945 J.Fields     374   9.84
4 00-0034796 L.Jackson    492   9.34
5 00-0036389 J.Hurts      587   9.19
6 00-0034855 B.Mayfield   514   8.78
7 00-0026498 M.Stafford   830   8.51
8 00-0031503 J.Winston    197   8.32
9 00-0029604 K.Cousins    635   8.23
10 00-0034857 J.Allen     831   8.22
```

11	00-0031280	D.Carr	781	8.13
12	00-0031237	T.Bridgewater	494	8.04
13	00-0019596	T.Brady	879	7.94
14	00-0035228	K.Murray	612	7.94
15	00-0036971	T.Lawrence	704	7.91
16	00-0036972	M.Jones	640	7.90
17	00-0033077	D.Prescott	753	7.81
18	00-0036442	J.Burrow	793	7.75
19	00-0023459	A.Rodgers	641	7.73
20	00-0031800	T.Heinicke	610	7.69
21	00-0035993	T.Huntley	242	7.68
22	00-0032950	C.Wentz	620	7.64
23	00-0029701	R.Tannehill	657	7.61
24	00-0037013	Z.Wilson	467	7.57
25	00-0036355	J.Herbert	791	7.55
26	00-0033119	J.Brissett	274	7.55
27	00-0033357	T.Hill	158	7.44
28	00-0028118	T.Taylor	187	7.43
29	00-0030520	M.Glennon	190	7.38
30	00-0035710	D.Jones	436	7.34
31	00-0036898	D.Mills	466	7.32
32	00-0031345	J.Garoppolo	579	7.31
33	00-0034869	S.Darnold	487	7.26
34	00-0026143	M.Ryan	652	7.16
35	00-0032156	T.Siemian	218	7.13
36	00-0036212	T.Tagovailoa	457	7.10
37	00-0033873	P.Mahomes	933	7.08
38	00-0030565	G.Smith	117	7.06
39	00-0027973	A.Dalton	277	6.99
40	00-0027939	C.Newton	147	6.97
41	00-0022924	B.Roethlisberger	738	6.76
42	00-0033106	J.Goff	568	6.44
43	00-0034177	T.Boyle	100	6.31
44	00-0034401	M.White	146	5.89
45	00-0027688	C.McCoy	114	5.17

`adot`, a commonly-used measure of quarterback aggressiveness, gives a quantitative method to rank quarterbacks by their aggression, as measured by mean air yards per pass (can you think of other ways to measure aggressiveness that pass depth alone leave out?). Look at the results and think, do they make sense to you, or are you surprised given your personal opinions of quarterbacks?

WARNING

If you get unexpected errors on any of the commands, double check that you are in the correct language environment. You may be trying to use Python in the R environment or R in the Python environment.

nfl_data_py in Python

In Python, the `nfl_data_py` package by Cooper Adams exists as a clone R of the `nflfastR` package for data. To use this data, first import the `pandas` package with the alias (or short nick-name) `pd` for working with data and import the `nfl_data_py` package as `nfl`:

```
## Python
import pandas as pd
import nfl_data_py as nfl
```

Next tell Python to import the data for 2021 ([Chapter 2](#) shows how to import multiple years). Note that you need to include the year, `2021`, in a Python list as `[2021]`:

```
## Python
pbp_py = nfl.import_pbp_data([2021])
```

Like the R code, filter the data in Python (Pandas calls filtering a `query`). Python allows you to readily pass the filter criteria (`filter_crit`) into `query()` as an object and we have you do this to save space line space. Then “group by” `passer_id` and `passer` before “aggregating” the data using a Python dictionary (`dict()` or `{}` for short) with the `.agg()` function:

```
## Python
filter_crit = 'play_type == "pass" & air_yards.notnull()'

pbp_py_p = (
    pbp_py.query(filter_crit)
    .groupby(["passer_id", "passer"])
    .agg({"air_yards": ["count", "mean"]})
)
```

Pandas also requires the column heads to be reformatted using a `list()` function and `map()` function to change the header from being two rows to being a single row. Next, print the outputs after sorting by the mean of the air yards using the `query()` function (`to_string()` allows all of the outputs to be

printed):

```
## Python
pbp_py_p.columns = list(map("_".join, pbp_py_p.columns.values))
sort_crit = "air_yards_count > 100"
print(
    pbp_py_p.query(sort_crit) \
    .sort_values(by="air_yards_mean", ascending=[False]) \
    .to_string()
)
          air_yards_count  air_yards_mean
passer_id   passer
00-0035704 D.Lock                110    10.154545
00-0029263 R.Wilson              400     9.887500
00-0036945 J.Fields              268     9.835821
00-0034796 L.Jackson             378     9.341270
00-0036389 J.Hurts               473     9.190275
00-0034855 B.Mayfield            416     8.776442
00-0026498 M.Stafford            740     8.508108
00-0031503 J.Winston              161     8.322981
00-0029604 K.Cousins              556     8.228417
00-0034857 J.Allen               708     8.224576
00-0031280 D.Carr                676     8.128698
00-0031237 T.Bridgewater          426     8.037559
00-0019596 T.Brady                808     7.941832
00-0035228 K.Murray               515     7.941748
00-0036971 T.Lawrence              598     7.913043
00-0036972 M.Jones                557     7.901257
00-0033077 D.Prescott              638     7.811912
00-0036442 J.Burrow                659     7.745068
00-0023459 A.Rodgers               556     7.730216
00-0031800 T.Heinicke              491     7.692464
00-0035993 T.Huntley                185     7.675676
00-0032950 C.Wentz                  516     7.641473
00-0029701 R.Tannehill              554     7.606498
00-0037013 Z.Wilson                382     7.565445
00-0036355 J.Herbert                671     7.554396
00-0033119 J.Brissett                224     7.549107
00-0033357 T.Hill                  132     7.439394
00-0028118 T.Taylor                  149     7.429530
00-0030520 M.Glennon                 164     7.378049
00-0035710 D.Jones                  360     7.344444
00-0036898 D.Mills                  392     7.318878
00-0031345 J.Garoppolo                511     7.305284
00-0034869 S.Darnold                  405     7.259259
00-0026143 M.Ryan                  559     7.159213
00-0032156 T.Siemian                  187     7.133690
00-0036212 T.Tagovailoa                387     7.103359
00-0033873 P.Mahomes                  780     7.075641
```

00-0027973	A.Dalton	235	6.987234
00-0027939	C.Newton	126	6.968254
00-0022924	B.Roethlisberger	647	6.761978
00-0033106	J.Goff	489	6.441718
00-0034401	M.White	132	5.886364

Hopefully, this chapter whet your appetite for using math to examine football data. We glossed over some of the many topics you will get to learn more about in future chapters such as data sorting, summarizing data, and cleaning data. You have also had a chance to compare Python and R for some basic tasks for working with data, including modeling. [Appendix B](#) also dives deeper into the air yards data to cover basic statistics and data wrangling.

Data Science Tools Used in This Chapter

This chapter included the following topics:

- You saw how to obtaining data from one seasons using the `nflfastR` package either directly in R or via the `nfl_data_py` package in Python
- You saw how to `filter()` in R or `query()` data in Python to select and subset data for analysis.
- You saw how to `summarize()` data by groups in R with help from `group_by()` and aggregate (`agg()`) data by groups in Python with help from `groupby()`.
- You saw how printing data frame outputs to your screen can help you look at data.
- You saw how to remove missing data in R using `is.na()` and `notnull()` in Python.

Suggested Readings

If you get really interested in analytics without the programming, here are some sources we read to develop our philosophy and strategies for football analytics:

- Carroll, Bob, Palmer, Pete, and Thorn, John. *The Hidden Game of Football*:

A Revolutionary Approach to the Game and Its Statistics (University of Chicago Press, 2023). Originally published in 1988, this cult classic introduces a number of the ideas that have later been formulated into the cornerstone of what is modern football analytics.

- Lewis, Michael. *Moneyball: The Art of Winning an Unfair Game* (WW Norton & Company, 2004). Lewis describes the rise of analytics in baseball and shows how the stage was set for other sports. The book helps us think about how modeling and data can help guide sports. A movie was also made of this book as well.
- Silver, Nate. *The Signal and the Noise: Why So Many Predictions Fail, but Some Don't* (Penguin, 2012). Silver describes why models work in some instances and fail in others. He draws upon his experience with poker, baseball analytics, and running the political prediction website fivethirtyeight.com. The book does a good job of showing how to think quantitatively for big picture problems without getting bogged down into details.

Lastly, we encourage you to read the [documentation for the `nflfastR` package](#). Diving into this package will help you to better understand much of the data used in this book.

Chapter 2. Exploring Data Analysis: Stable Versus Unstable Quarterback Statistics

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

In any field of study, there's a measure of "gut feel" that can often separate the truly great subject matter experts from the average ones, the average ones from the early-career professionals, or the early-career professionals from the novices. In football that skill is said to manifest itself in player evaluation, where some scouts are perceived to have a knack for identifying talent through great intuition earned over years of honing their craft. Player traits that translate from one situation to the next - whether that be from college football to the professional ranks, or one coach's scheme to the other's - require recognition and further investigation, while production that is gained from non-sustainable means discarded. Exerts in player evaluation also know how to properly communicate the fruits of his or her labor in order to gain maximum collective benefit from it.

While traditional scouting and football analytics are often considered at odds with one another, the statistical evaluation of players requires essentially the same process. Great football analysts are able to, when evaluating a player's data (or multiple players' data), find the right data specs to interrogate, production

metrics to use, and situational factors to control for, and information to discard completely. How does one acquire such an acumen? The same way a scout does. Through years of deliberate practice and refinement, an analyst gains not only a set of tools for player, team, scheme, and game evaluation, but a knack for the right question to ask at the right time.

TIP

Famous statistician, [John Tukey noted](#): `Far better an approximate answer to the right question, which is often vague, than an exact answer to the wrong question, which can always be made precise.' Practically this quote illustrates that, in football, or broader data science, asking the question that meets your needs is more important than using your existing data and models precisely.

One advantage that statistical approaches have over traditional methods is that they are scalable. Once an analyst develops a tried-and-true method for player evaluation, he or she can use the power of computing to run that analysis on many players at once - a task that is incredibly cumbersome for traditional scouting methods.

In this chapter, we give you some of the first tools necessary to develop a knack for football evaluation using statistics. The first idea to explore in this topic is the idea of *stability*. Stability is important when evaluating anything, especially in sports. This is because stability measures provide a comparative way in which to determine how much of a skill is *fundamental* to the player - how much of what happened in a given setting is transferable to another setting - and how much of past performance can be attributed to *variance*. In the words of the founder of [fivethirtyeight.com](#), Nate Silver, stability analysis helps us tease apart what is *signal* and what is *noise*. If a player does very well in the stable components of football, but poorly in the unstable ones, they might be a buy low candidate in the future, while the opposite might be a sell high one.

Exact definitions of stability vary based upon specific contexts, but in this book we refer to the stability of a evaluation metric as the metric's consistency over a predetermined time frame. For example, for fantasy football analysts, that time frame might be week to week, while for an analyst building a draft model for his or her team, it might be from a player's final few seasons in college to his first few seasons as a pro. The football industry generally uses Pearson's correlation

coefficient, or its square, the coefficient of determination, to measure stability.

While the precise numerical threshold it has to hit for a metric to be considered stable is usually context- or era-specific, a higher coefficient means a more stable statistic. For example, pass-rushing statistics are generally pretty stable, while coverage statistics are not (Eric talked about this more in a recent [paper from the Sloan Sports Analytics Conference](#)). If measuring two different pass-rushing metrics against each other the less-stable one might have a lower correlation coefficient than the more-stable coverage metric.

NOTE

Fantasy fans will know *stability analysis* by another term: *sticky stats*. This is because the estimates stick around and are consistent.

Stability analysis is part of a subset of statistical analysis called *exploratory data analysis* (EDA), which was coined by the American statistician John Tukey. In contrast to formal modeling and hypothesis testing, EDA is an approach of analyzing data sets to summarize their main characteristics, often using statistical graphics and other data visualization methods. EDA is an often-overlooked step in the process of using statistical analysis to understand the game of football - both by newcomers and veterans of the discipline - but for different reasons.

NOTE

John Tukey also coined other terms, some that you may know or will hopefully know by the end of this book including *boxplot* (a type of graph), *analysis of variance* (ANOVA for short; a type of statistical test), *software* (computer programs), and *bit* (the smallest unit of computer data, usually represented as 0/1; you're probably more familiar with larger units such as the byte, which is eight bits, and larger units like gigabytes). Tukey and his students also helped the Princeton University football team implement data analysis using basic statistical methods by examining over 20 years of football data. However, the lack of modern computers limited his work, and many of the tools you learn in this book are more advanced than the methods he had access to. For example, one of his former students, [Gregg Lange](#), remembered how a simple mistake required him to reload 100 pounds of data cards into a computer. To read more about Tukey's life and contributions, checkout the obituary written by David Brillinger that appeared in the [Annals of Statistics](#).

Defining Questions

Asking the right questions is as important as solving them. In fact, as the Tukey quote highlights, the right answer to the wrong question is useless in and of itself, while the right question can lead you to prosperous outcomes even if you fall short of the correct answer. Learning to ask the right question is a process honed by learning from asking the wrong questions. Positive results are the spoils earned from fighting through countless negative results.

To be scientific, a question needs to be about a hypothesis that is both testable and falsifiable. For example, “throwing deep passes is more valuable than short passes, but it’s difficult to say whether or not a quarterback is good at deep passes” is a reasonable hypothesis, but to make it scientific you need to define what “valuable” means and what you mean when we say a player is “good” (or “bad”) at deep passes. To that aim, you need data.

Obtaining and Filtering Data

To study the stability of passing data, use the `nflfastR` package in R or the `nfl_data_py` package in Python. Start by loading the data from 2016 to 2021 as play-by-play, or `pbp` for short using the tools you learned in [Chapter 1](#). In Python, load the `pandas` and `numpy` packages as well as the `nfl_data_py` package:

```
## Python
import pandas as pd
import numpy as np
import nfl_data_py as nfl
```

WARNING

Python starts numbering at 0. R starts numbering at 1. Many an aspiring data scientist has been tripped up by this if using both languages. Because of this, you need to add `+ 1` to the input of the last value in `range()` in this example.

Next, tell Python what years to load using `range()`. Then, import the NFL data for those seasons:

```
## Python
seasons = range(2016, 2021 + 1)
pbp_py = nfl.import_pbp_data(seasons)
```

In R, first load the required packages. The `tidyverse` package helps you wrangle and plot the data. The `nflfastR` package provides you with the data. The `ggthemes` package assists with plotting formatting:

```
## R
library("tidyverse")
library("nflfastR")
library("ggthemes")
```

In R, you may use the shortcut `2016:2021` to specify the range 2016 to 2021:

```
## R
pbp_r <- load_pbp(2016:2021)
```

WARNING

With any dataset, understand the metadata, or, the data about the data. For example, what do 0 and 1 mean? Which is 'yes' and which is 'no'? Or, do the authors use 1 and 2 for the levels? We have heard about scientific studies being retracted because the data analysts and scientists misunderstood the metadata and the uses of 1 and 2 versus the standard 0 and 1. Thus, scientists had to tell people their study was flawed because they did not understand their own data structure. For example, a [2021 article in *Significance*](#) describes an occurrence of this mistake.

To get the subset of data you need for this analysis, subset down to just the passing plays, which can be done with the following code:

```
## Python
pbp_py_p = \
    pbp_py\
    .query("play_type == 'pass' & air_yards.notnull()")\
    .reset_index()
```

In R, `filter()` the data using the same criteria:

```
## R
pbp_r_p <-
    pbp_r |>
```

```
filter(play_type == "pass" & !is.na(air_yards))
```

Here `play_type` being equal to `pass` will eliminate both running plays and plays that are negated because of a penalty. Sometimes you want to include plays that have a penalty, for example if you are using a grade-based system like the one at PFF. Grade-base systems attempt to measure how well the player played on a play independent of the final statistics of the play, so keeping data where `play_type == no_play` might have value.

For the sake of this exercise, though, we have you omit such plays. You also omit plays where the `air_yards` is `NA` (in R) or `NULL` (in Python). These plays occur when a pass is not aimed at an intended receiver because it's either batted down at the line of scrimmage, thrown away, or spiked. While those passes certainly count towards a passer's final statistics, and are fundamental to who he is as a player, they are not necessarily relevant to the question being asking here.

Next, some data cleaning and wrangling needs to be done.

First, define a “long” pass as a pass that has air yards greater than or equal to 20 yards, and a “short” pass as one with air yards less than 20 yards. The NFL actually has a categorical variable for pass length (`pass_length`) in their data, but the classifications are not completely clear to the observer (see the exercises at the end of the chapter). Luckily, you can easily calculate this on your own (and, use a different criterion if desired, such as 15 yards or 25 yards).

Second, the passing yards for incomplete passes are recorded a `NA` in R or `NULL` in Python, but should set to zero for this analysis (as long as you've subsetted properly above).

In Python, the `numpy` (imported as `np`) package's `where()` function helps with this change. First, create the filtering criteria:

```
## Python
pbp_py_p["pass_length_air_yards"] = np.where(
    pbp_py_p["air_yards"] >= 20, "long", "short"
)
```

Then, use the filtering criteria to replace missing values:

```
## Python
pbp_py_p["passing_yards"] = \
    np.where(
        pbp_py_p["passing_yards"].isnull(), 0,
        pbp_py_p["passing_yards"]
    )
```

In R, the `ifelse()` function inside `mutate()` allows the same change:

```
## R
pbp_r_p <-
    pbp_r_p |>
    mutate(
        pass_length_air_yards = ifelse(air_yards >= 20, "long",
"short"),
        passing_yards = ifelse(is.na(passing_yards), 0, passing_yards)
    )
```

[Appendix B](#) covers data manipulation topics such as filtering in great detail. Refer to this source if you need help better understanding our data wrangling. We are glossing over these details to help you get into the data right away with interesting questions.

TIP

Naming objects can be actually surprisingly hard when programming. Try to balance simple names that are easier to type with longer, more informative names. This can be especially important if you start writing scripts with longer names. The most important part of naming is to create understandable names for both others and your future self.

Summarizing Data

Briefly examine some basic numbers to used to describe the `passing_yards` data. In Python, select the `passing_yards` column and then use the `describe()` function:

```
## Python
pbp_py_p["passing_yards"]\n    .describe()
count      112666.000000
mean       7.217519
```

```

std           9.707331
min        -16.000000
25%         0.000000
50%         5.000000
75%        11.000000
max        98.000000
Name: passing_yards, dtype: float64

```

In R, take the data frame and select (or `pull()`) the `passing_yards` column and then calculate the `summary()` statistics:

```

## R
pbp_r_p |>
  pull(passing_yards) |>
  summary()
   Min. 1st Qu. Median      Mean 3rd Qu.      Max.
-16.000  0.000  5.000  7.218  11.000  98.000

```

In the outputs, here are what the names describe ([Appendix B](#) shows how to calculate these values):

- The `count` (only in Python) is the number of records in the data.
- The `mean` in Python (`Mean` in R) is the arithmetic average.
- The `std` (only in Python) is the standard deviation.
- The lowest or minimum (`min` in Python or `Min.` in R) value.
- The 1st quartile (25% in Python or `1st Qu.` in R) or value for which 1/4th of all values are smaller.
- The `Median` (in R) or `50%` (in Python) is the middle value for which 1/2 of the values are bigger and 1/2 are smaller.
- The 3rd quartile (75% in Python or `3rd Qu.` in R) or value for which 3/4th of all values are smaller.
- The largest or maximum (`max` in Python or `Max.` in R) value.

What you really want to see is a summary of the data under different values of `pass_length_air_yards`. For short passes, filter out the long passes and then summarize, in Python:

```

## Python
pbp_py_p\ 
    .query('pass_length_air_yards == "short"')["passing_yards"]\ 
    .describe()
count      99305.000000
mean       6.543366
std        7.712207
min       -16.000000
25%        0.000000
50%        5.000000
75%        10.000000
max       95.000000
Name: passing_yards, dtype: float64

```

And in R:

```

## R
pbp_r_p |>
  filter(pass_length_air_yards == "short") |>
  pull(passing_yards) |>
  summary()
Min. 1st Qu. Median     Mean 3rd Qu.    Max.
-16.000   0.000   5.000   6.543  10.000  95.000

```

Likewise, you can filter long passes in Python:

```

## Python
pbp_py_p\ 
    .query('pass_length_air_yards == "long"')["passing_yards"]\ 
    .describe()
count      13361.000000
mean       12.228127
std        18.002176
min       0.000000
25%        0.000000
50%        0.000000
75%        26.000000
max       98.000000
Name: passing_yards, dtype: float64

```

And in R:

```

## R
pbp_r_p |>
  filter(pass_length_air_yards == "long") |>
  pull(passing_yards) |>

```

```

summary()
Min. 1st Qu. Median Mean 3rd Qu. Max.
0.00 0.00 0.00 12.23 26.00 98.00

```

The thing to notice here is that the *interquartile range*, the difference between the first and third quartile, is much larger for longer passes than for short passes, even though the maximum passing yards are about the same. The minimum values are going to be higher for long passes, since it's almost impossible to gain negative yards on a pass that travels 20 or more yards in the air.

You can perform the same analysis for expected points added (EPA), which was introduced in [Chapter 1](#). Recall that EPA is a more continuous measure of play success that uses situational factors to assign a point value to each play. You can do this in Python:

```

## Python
pbp_py_p \
    .query('pass_length_air_yards == "short"')["epa"] \
    .describe()
count    99304.000000
mean      0.122903
std       1.427851
min      -12.219461
25%      -0.605068
50%      0.003059
75%      0.963851
max       8.241420
Name: epa, dtype: float64

```

And in R:

```

## R
pbp_r_p |>
  filter(pass_length_air_yards == "short") |>
  pull(epa) |>
  summary()
    Min. 1st Qu. Median Mean 3rd Qu. Max.
NA's -12.219462 -0.605068 0.003059 0.122903 0.963851 8.241420
1

```

Likewise, you can do this for long passes in Python:

```

## Python
pbp_py_p\ 
    .query('pass_length_air_yards == "long"')["epa"]\
    .describe()
count      13361.000000
mean       0.396043
std        2.185827
min       -10.477922
25%       -0.819668
50%       -0.460198
75%        2.161345
max        8.789743
Name: epa, dtype: float64

```

Or in R:

```

## R
pbp_r_p |>
  filter(pass_length_air_yards == "long") |>
  pull(epa) |>
  summary()
  Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-10.4779 -0.8197 -0.4602  0.3960   2.1613   8.7897

```

You get the same dynamic here - wider outcomes for longer passes than shorter ones. Longer passes are more *variable* than shorter passes. By *variable*, we are referring to a great amount of differences among values.

Furthermore, if you look at the mean passing yards per attempt and EPA per attempt for longer passes, they are both higher than those for short passes (while the relationship flips for the median, why is that?). Thus, on average, you can informally confirm the first part of our guiding hypothesis for the chapter: “throwing deep passes is more valuable than short passes, but it’s difficult to say whether or not a quarterback is good at deep passes”.

TIP

Line breaks and white-space are important for coding. These breaks help make your code easier to read. Python and R also handle line breaks differently, but, sometimes, both languages treat line breaks as special commands. In both languages, you often split function inputs to create shorter lines that are easier to read. For example, you can space a function like:

```

## Python or R
my_plot(data=big_name_data_frame,

```

```
x="long_x_name",  
y="long_y_name")
```

to break up line names and make our code easier to read. In R, make sure the comma stays on a previous line. In Python, you may need to use a \ for line breaks. For example, you use:

```
## Python  
x =\  
    2 + 4
```

Or, put the entire command in parentheses:

```
## Python  
x = (  
    2 + 4  
)
```

You can also write one Python function per line using:

```
## Python  
my_out = \  
    my_long_long_long_data\  
    .function_1()\  
    .function_2()
```

Plotting Data

While numerical summaries of data are useful, and many people are more algebraic thinkers than they are geometric ones (Eric is this way), many people need to visualize something other than numbers. Reasons we like to plot data include:

- Checking to make sure the data looks okay. For example, are there values that are too large? Too small? Do other wonky data points exist?
- Are there outliers in the data? Do they arise naturally (e.g. Patrick Mahomes in almost every passing efficiency chart) or unnaturally (e.g. a probability below 0 or greater than 1)?
- Do any broad trends emerge at first glance?

Histograms

Histograms, a type of plot, allow you to see data by summing the counts of data points into bars. These bars are called *bins*.

WARNING

If you have previous versions of the packages used in this book installed, you may need to upgrade if our code examples will not work.

In Python, we use `seaborn` package for most plotting in the book. First import `seaborn` using the alias `sns`. Then plot using the `displot()` function as shown in [Figure 2-1](#):

```
## Python
import seaborn as sns
import matplotlib.pyplot as plt

sns.displot(data=pbp_py, x="passing_yards");
plt.show();
```

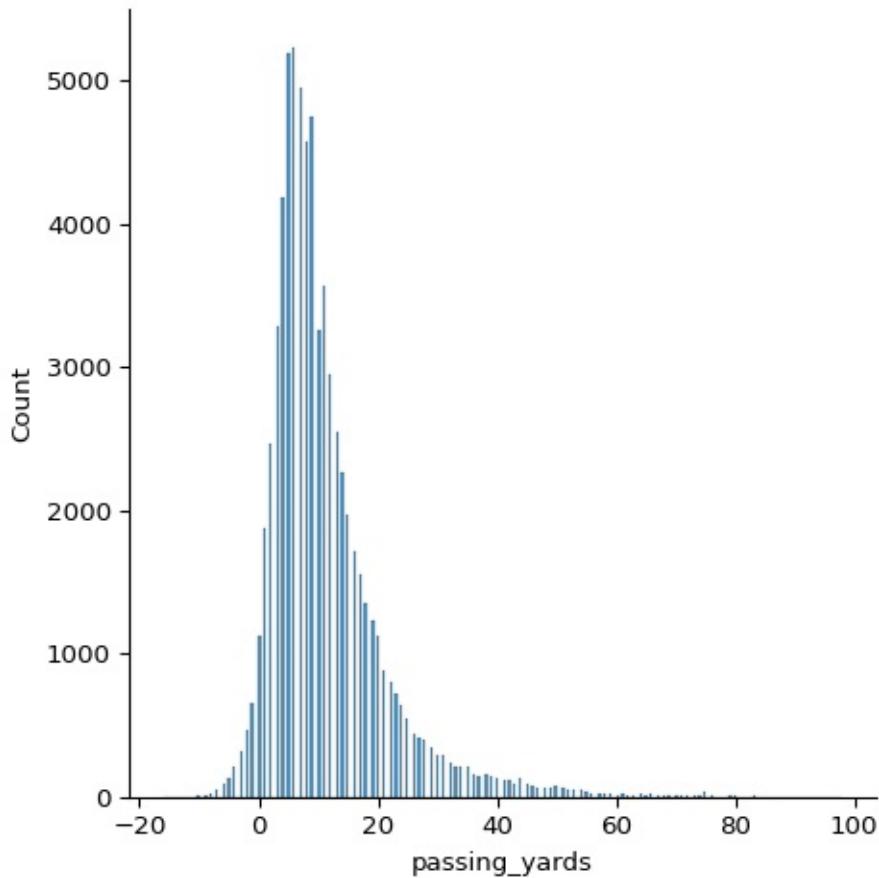


Figure 2-1. Example histogram in Python using `seaborn` for the `passing_yards` variable.

WARNING

On macOS, you will also need to include `import matplotlib.pyplot as plt` when you load other packages. Likewise macOS users will also need to include a `plt.show()` for their plot to appear after their plotting code. We also found we needed to use `plt.show()` with some editors on Linux (such as Visual Code) but not others such as Jupyter Lab. If in doubt, include this option code. Running `plt.show()` will do no harm, but, might be needed to make your figures appear. Windows may or may not require this.

Likewise, R allows for histograms to be easily created.

NOTE

Although base R comes with its own plotting tools, we use `ggplot2` for this book. `ggplot2` has its own language, based upon the *Grammar of Graphics* by Leland Wilkinson (Springer, 2005) and implemented in R by Hadley Wickham during his doctoral studies at Iowa State University.

Pedagogically, we agree with David Robinson, who describes his reasons for teaching plotting with `ggplot2` over base R in a blog post titled [Don't teach built-in plotting to beginners \(teach ggplot2\)](#).

In R, create a histogram using `ggplot2` in R with the `ggplot()` function such as [Figure 2-2](#). In the function, use the `pbp_r_p` dataset and set the aesthetic for `x` to be `passing_yards`. Then add the geometry `geom_histogram()`. [Figure 2-2](#) lets you see the distribution:

```
## R
ggplot(pbp_r, aes(x = passing_yards)) +
  geom_histogram()
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
Warning: Removed 219281 rows containing non-finite values
(`stat_bin()`).
```

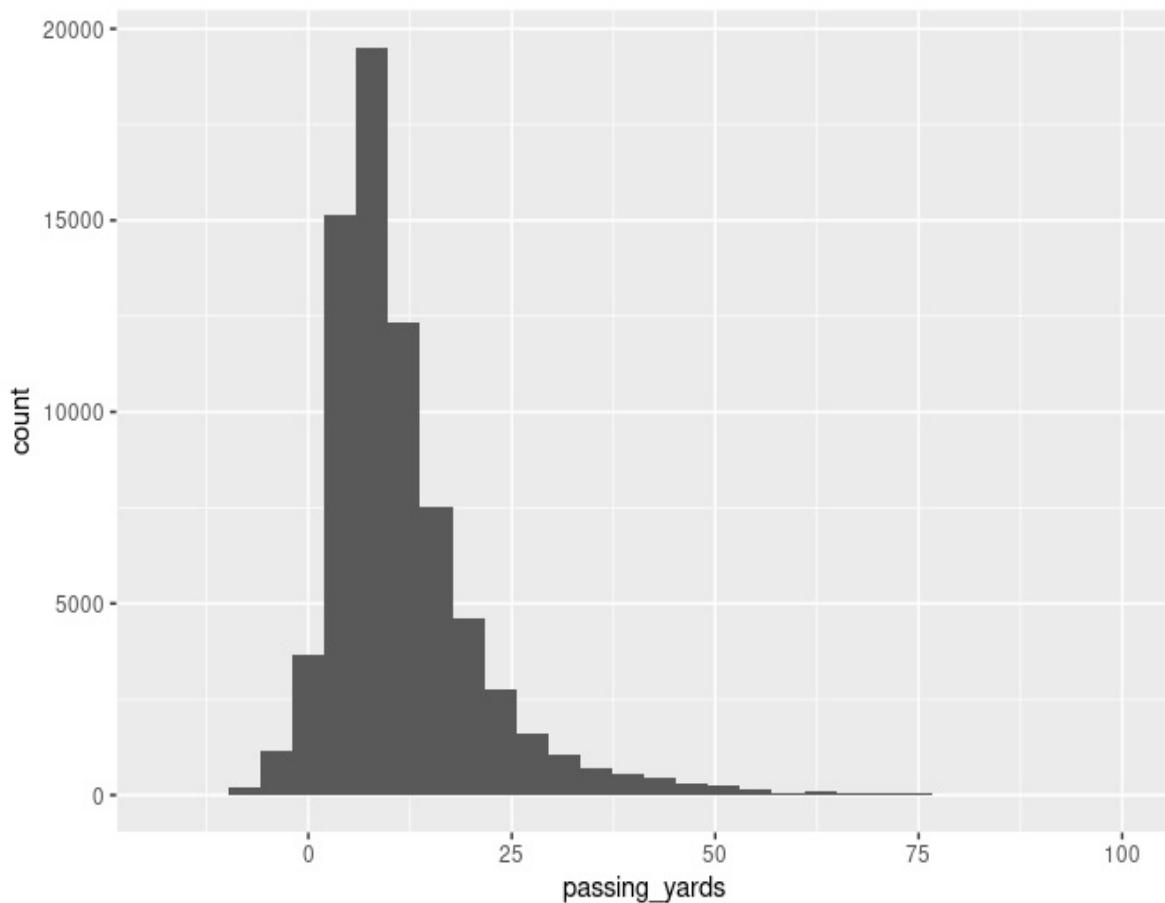


Figure 2-2. Example histogram in R using ggplot2 for the passing_yards variable.

WARNING

Intentionally using the wrong number of bins to hide important attributes of your data is considered fraud by the larger statistical community. Be thoughtful and intentional when you select the number of bins for a histogram. This process requires many iterations as you explore different numbers of histogram bins.

Figure 2-1 and **Figure 2-2** let you understand the basis of our data. Passing yards gained ranges from about -10 yards to about 75 yards, with most plays gaining between 0 (often an incompletion) and 10 yards. Notice how R provides you a warning to be careful with the `binwidth` and the number of bins and also warns you about the removal of missing values. Rather than using the default, set each bin to be 1 yard wide. The second warning about missing values can either be ignored, or, you can filter out missing values prior to plotting to avoid the warning. With such a binwidth, the data no longer looks normal, because of the many, many incomplete passes.

Next you will make a histogram for each `pass_depth_air_yards` value. We will show you how to create the short pass in Python (**Figure 2-3**) and the long pass in R (**Figure 2-4**).

In Python, change the theme to be `colorblind` for the palette option and use a `whitegrid` option to create plots similar to `ggplot2`'s black and white theme by using:

```
## Python
import seaborn as sns
import matplotlib.pyplot as plt

sns.set_theme(style="whitegrid", palette="colorblind")
```

Next, filter out the short passes:

```
## Python
pbp_py_p_short = \
    pbp_py_p\
    .query('pass_length_air_yards == "short"')
```

Then, create a histogram and use `set_axis_labels` to change the plots labels to create, also make the figure look better to create **Figure 2-3**:

```

## Python
# Plot, change labels, and then show the output
pbp_py_hist_short = \
    sns.displot(data=pbp_py_p_short,
                binwidth=1,
                x="passing_yards");
pbp_py_hist_short\
    .set_axis_labels(
        "Yards gained (or lost) during a passing play", "Count"
    );
plt.show();

```

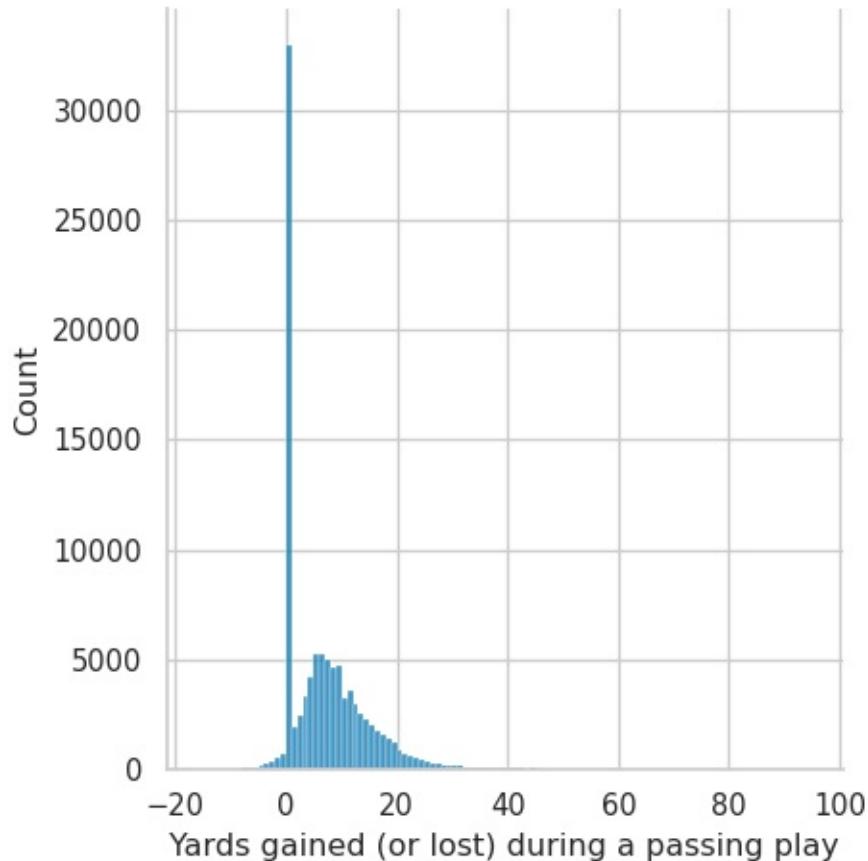


Figure 2-3. Refined histogram in Python using *seaborn* for the *passing_yards* variable.

In R, filter out the long passes and make the plot look better by adding labels to the x- and y-axes and using the black and white theme (`theme_bw()`) creating Figure 2-4:

```

## R
pbp_r_p |>
  filter(pass_length_air_yards == "long") |>

```

```

ggplot(aes(passing_yards)) +
  geom_histogram(binwidth = 1) +
  ylab("Count") +
  xlab("Yards gained (or lost) during passing plays on long passes")
+
  theme_bw()

```

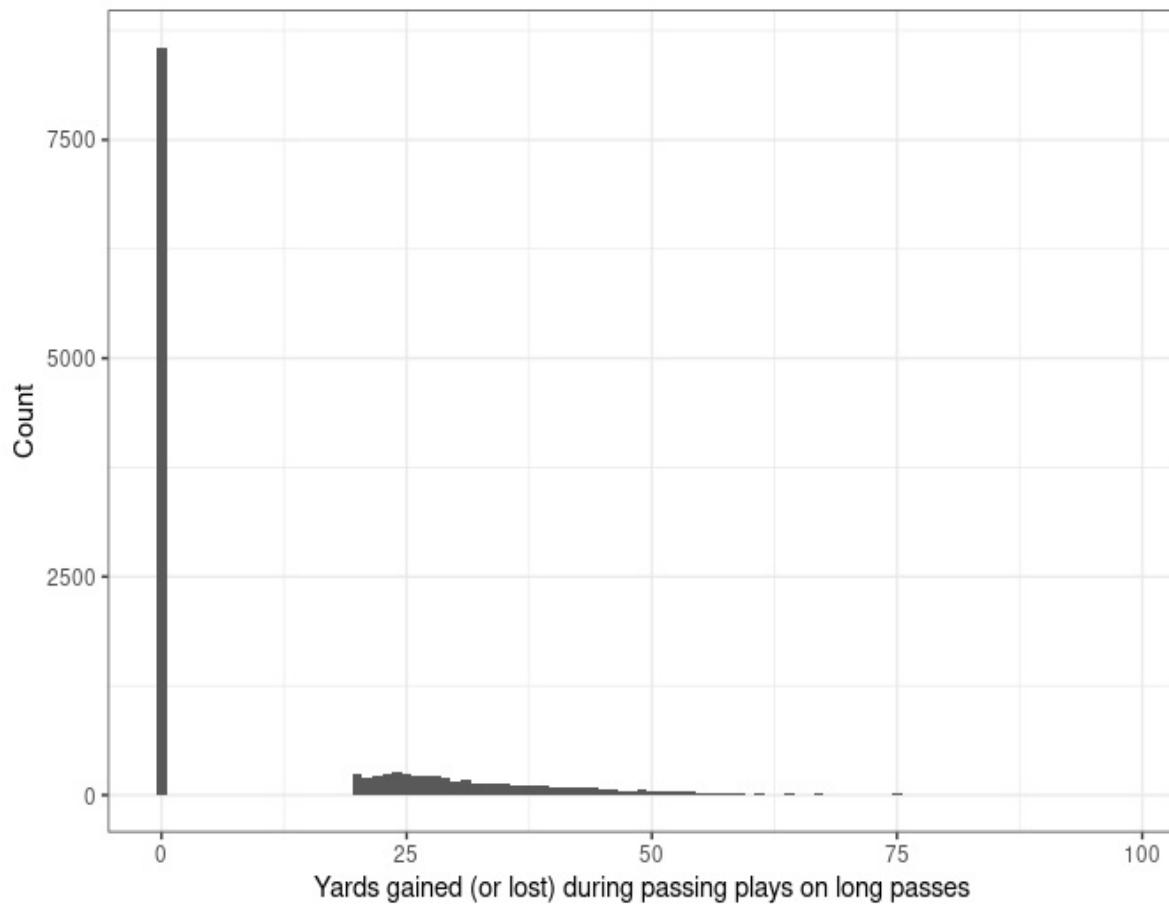


Figure 2-4. Refined histogram in R using ggplot2 for the passing_yards variable.

NOTE

We will use the black-and-white theme `theme_bw()` for the remainder of the book as the default for R plots and `sns.set_theme(style="whitegrid", palette="colorblind")` for Python plots. We like these because we think it looks better on paper.

These histograms represent pictorially what you saw numerically in “[Summarizing Data](#)”. Specifically, shorter passes have less variable outcomes than longer passes. You can do the same thing with EPA and find similar results.

For the rest of the chapter we will stick with passing yards per attempt for our examples, with EPA per pass attempt left to the exercises for you.

NOTE

Notice how `ggplot2`, and, more broadly, R, work well with piping objects and avoid intermediate objects. In contrast, Python works well by saving intermediate objects. Both approaches have trade-offs. For example, saving intermediate objects allows you to see the output of intermediate steps of your plotting. In contrast, re-writing the same object name can be tedious. This difference represents a philosophical difference between the two languages. Neither is inherently right or wrong, but represents trade-offs.

Boxplots

Histograms allow people to *see* the distribution of data points. However, histograms can be cumbersome, especially when exploring many variables. Boxplots are a compromise between histograms and numerical summaries (see [Table 2-1](#) for the numerical values). *Boxplots* get their name because they have a rectangular *box* containing the middle 50% of the sorted data, where the line in the middle of the box is the median, or the line where half of the sorted data falls above the line and half of the data falls under the line.

Some people call boxplots by the name *box-and-whisker* plots because lines extend above and under the box. These whiskers contain the remainder of the data other than outliers. Boxplots in both `seaborn` and `ggplot` use a default for *outliers* to be points that are more than 1.5 times the range between the 25% and 75 percentiles (or interquartile range) from either the first or third quartile. These outliers are plotted with dots.

Table 2-1. Table 4.1: Parts of a boxplot.

Part name	Range of data
Top dots	Outliers above the data
Top whisker	100% to 75% of data, excluding outliers

Top portion of box	75% to 50% of data
Line in middle of box	50% of data
Bottom portion of box	50% to 25% of data
Bottom whisker	25% to 0% of data, excluding outliers
Bottom dots	Outliers under the data

Different types of outliers exist. Outliers may be problem data points (for example, somebody entered -10 yards when they meant 10 yards), but often exist as parts of the data. Understanding the reasons behind these data points often provides keen insights to the data because outliers reflect the best and worst outcomes and may have interesting stories behind the points. Unless outliers exist due to errors (such as the wrong data being entered), outliers usually should be included in the data used to train models.

NOTE

We place a semicolon (;) after the Python plot commands to suppress text descriptions of the plot. These semicolons are optional and simply a preference of the authors.

In Python, use the `boxplot` function from `seaborn` and change the axes labels to create [Figure 2-5](#):

```
## Python
pass_boxplot = \
    sns.boxplot(data=pbp_py_p,
                x="pass_length_air_yards",
                y="passing_yards");
pass_boxplot.set(
    xlabel="Pass length (long >= 20 yards, short < 20 yards)",
    ylabel="Yards gained (or lost) during a passing play",
);
plt.show();
```

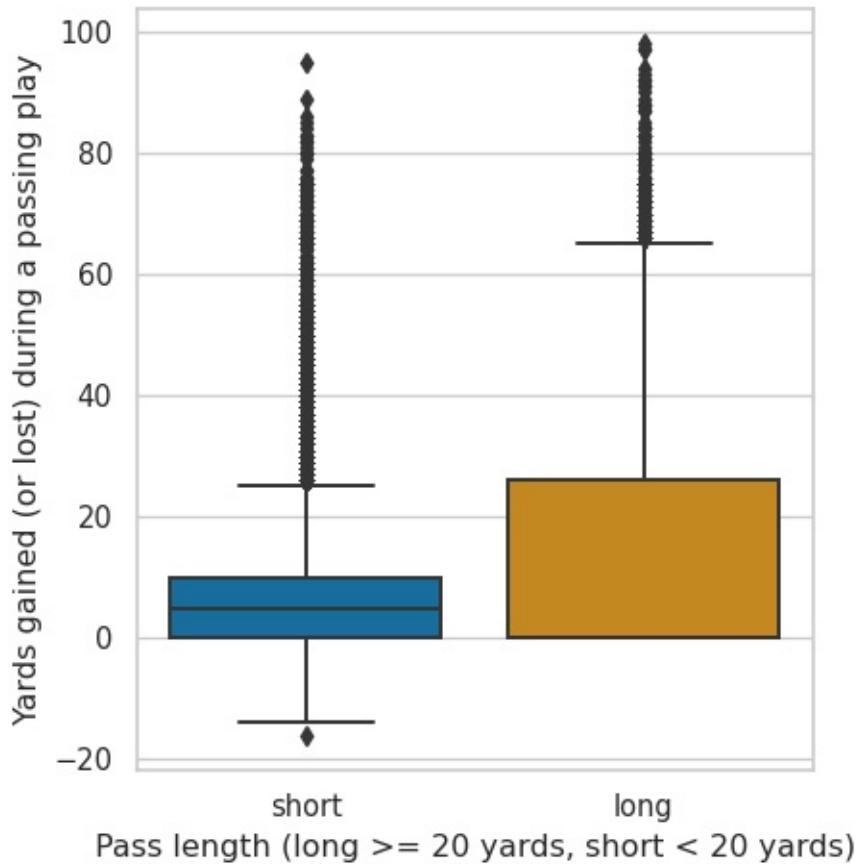


Figure 2-5. Boxplot of yards gained from long and short air yard passes, created with seaborn.

In R, use `geom_boxplot` with `ggplot2` to create **Figure 2-6**:

```
## R
ggplot(pbp_r_p, aes(x = pass_length_air_yards, y = passing_yards)) +
  geom_boxplot() +
  theme_bw() +
  xlab("Pass length in yards (long >= 20 yards, short < 20 yards)") +
  ylab("Yards gained (or lost) during a passing play")
```

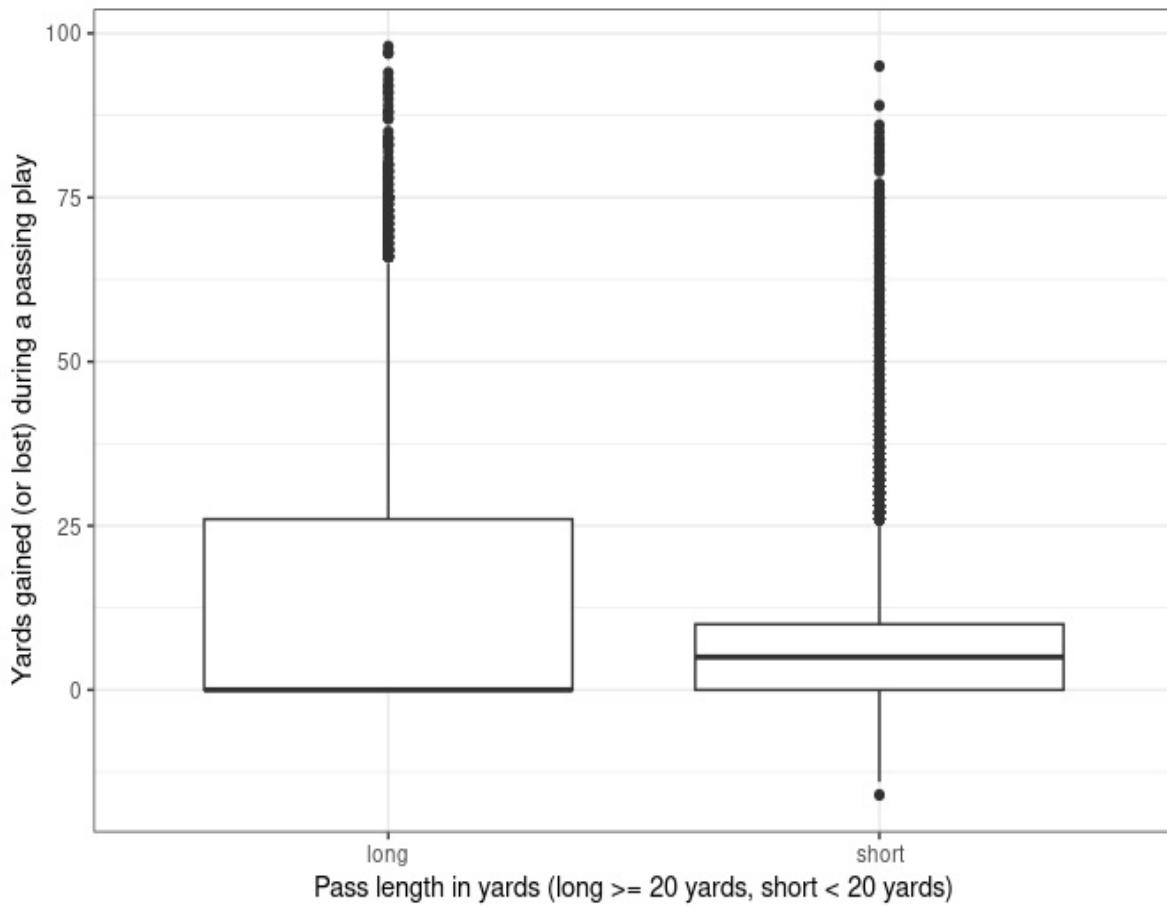


Figure 2-6. Boxplot of yards gained from long and short air yard passes, created with `ggplot2`.

Player-Level Stability of Passing Yards per Attempt

Now that you've become acquainted with our data, it's time to use the data to do some player evaluation. The first thing you have to do is aggregate across a pre-specified time frame to get a value for each player. While week-level outputs certainly matter, especially for fantasy football and betting (see [Chapter 7](#)), most of the time when teams are thinking about trying to acquire a player they use season-level data (or sometimes data over many seasons).

Thus, you aggregate at the season level here, by using the `groupby()` syntax in Python and `group_by()` syntax in R. The *group by* concept borrows from SQL-type database languages. When thinking about the process here, “group by” may be thought of as a verb. For example, you use the play-by-play data and

then “group by” the “seasons” and then “aggregate” (in Python) or “summarize” (in R) to calculate the mean of the quarterback’s passing yards per attempt.

For this problem, take the play-by-play data frame (`pbp_py` or `pbp_r`) and then *group by* `passer_player_name`, `passer_player_id`, `season`, and `pass_length`. Group by both the player id and the player name column because some players have the same name (or, at least same first initial and last name), but the name is important for studying the results of the analysis. Start with the whole data set first before transitioning to the different subsets.

In Python, use `groupby()` with a list of the variables (`["item1", "item2"]` in Python syntax) you want to group by. Then, aggregate the data for `passing_yards` for the `mean`:

```
## Python
pbp_py_p_s = \
    pbp_py_p \
    .groupby(["passer_id", "passer", "season"]) \
    .agg({"passing_yards": ["mean", "count"]})
```

With Python, also collapse the columns to make the data frame easier to handle (`list()` creates a list, `map()` iterates over items, like a `for` loop without the loop syntax, see [Chapter 7](#) details on `for` loops):

```
## Python
pbp_py_p_s.columns = list(map("_".join, pbp_py_p_s.columns.values))
```

Next, rename the columns to names that are shorter more intuitive:

```
pbp_py_p_s \
    .rename(columns={'passing_yards_mean': 'ypa',
                    'passing_yards_count': 'n'},
            inplace=True)
```

In R, pipe the `pbp_p` to the `group_by()` function and then use the `summarize()` function to calculate the `mean()` of `passing_yards`, as well as calculate the number, `n()` of passing attempts for each player in each season. Include `.groups = "drop"` to tell R to drop the groupings from the resulting data frame. The resulting mean of `passing_yards` is the *yards-per-*

attempt (YPA). YPA is a quarterbacks average passing distance per play. Use the `<-` function to save the resulting calculations as a new data frame, `pbp_r_p_s`:

```
## R
pbp_r_p_s <-
  pbp_r_p |>
  group_by(passer_player_name, passer_player_id, season) |>
  summarize(
    ypa = mean(passing_yards, na.rm = TRUE),
    n = n(),
    .groups = "drop"
)
```

Now look at the top of the resulting data frame by using `in head()` in Python and `sort()` by the 'ypa' to help you better see the results.
`ascending=False` tells Python to sort high-to-low rather (for example arranging the values to 9, 8, 7) than low-to-high (for example arranging the values to be 7, 8, 9):

```
## Python
pbp_py_p_s\
  .sort_values(by=["ypa"], ascending=False)\\
  .head()
      ypa   n
passer_id  passer  season
00-0035544 T.Kennedy 2021    75.0  1
00-0033132 K.Byard   2018    66.0  1
00-0031235 O.Beacham 2018    53.0  2
00-0030669 A.Wilson  2018    52.0  1
00-0029632 M.Sanu    2017    51.0  1
```

In R, use `arrange()` with `ypa` to sort the outputs. The negative symbol `(-)` tells R to reveres the order (for example, 7, 8, 9 becomes 9, 8, 7 when sorted):

```
## R
pbp_r_p_s |>
  arrange(-ypa) |>
  print()
# A tibble: 640 × 5
  passer_player_name  passer_player_id  season     ypa     n
  <chr>                <chr>            <dbl> <dbl> <int>
1 T.Kennedy             00-0035544       2021    75     1
2 K.Byard               00-0033132       2018    66     1
3 O.Beacham              00-0031235       2018    53     2
```

```

4 A.Wilson      00-0030669    2018    52    1
5 M.Sanu        00-0029632    2017    51    1
6 C.McCaffrey   00-0033280    2018    50    1
7 W.Snead        00-0030663    2016    50    1
8 T.Boyd        00-0033009    2021    46    1
9 R.Golden       00-0028954    2017    44    1
10 J.Crowder     00-0031941   2020    43    1
# i 630 more rows

```

Now this isn't really informative, yet, since the players with the highest yards per attempt values are players that threw a pass or two (usually a trick play) that were completed for big yardage. Fix this by filtering for a certain number of passing attempts in a season (let's say 100), and see what you get.

[Appendix C](#) contains more tips tricks for data wrangling if you need more help understanding what is going on with this code. In Python, reuse `pbp_py_p_s` and the previous code, but include a `query()` for players with 100 or more pass attempts using '`n >= 100`':

```

## Python
pbp_py_p_s_100 = \
    pbp_py_p_s\
    .query("n >= 100")\
    .sort_values(by=["ypa"], ascending=False)

```

Now looking at the head of the data:

```

## Python
pbp_py_p_s_100.head()

```

			ypa	n
passer_id	passer	season		
00-0023682	R.Fitzpatrick	2018	9.617886	246
00-0026143	M.Ryan	2016	9.442155	631
00-0029701	R.Tannehill	2019	9.069971	343
00-0033537	D.Watson	2020	8.898524	542
00-0031345	J.Garoppolo	2017	8.863636	176

In R, “group by” the same variables and then “summarize” (this time including the number of observations per group with `n()`). Keep piping the results and “filter” for passers with 100 or more (`n >= 100`) passes and “arrange” the output:

```

## R
pbp_r_p_100 <-
  pbp_r_p |>
  group_by(passer_id, passer, season) |>
  summarize(
    n = n(), ypa = mean(passing_yards),
    .groups = "drop"
  ) |>
  filter(n >= 100) |>
  arrange(-ypa)

```

Then, print the top 20 results:

```

## R
pbp_r_p_100 |>
  print(n = 20)
# A tibble: 253 × 5
  passer_id  passer      season     n     ypa
  <chr>       <chr>       <dbl> <int> <dbl>
1 00-0023682 R.Fitzpatrick 2018    246   9.62
2 00-0026143 M.Ryan      2016    631   9.44
3 00-0029701 R.Tannehill  2019    343   9.07
4 00-0033537 D.Watson    2020    542   8.90
5 00-0031345 J.Garoppolo 2017    176   8.86
6 00-0033873 P.Mahomes   2018    651   8.71
7 00-0036442 J.Burrow    2021    659   8.67
8 00-0026498 M.Stafford  2019    289   8.65
9 00-0031345 J.Garoppolo 2021    511   8.50
10 00-0033319 N.Mullens  2018    270   8.43
11 00-0033537 D.Watson   2017    202   8.41
12 00-0033077 D.Prescott 2020    221   8.40
13 00-0029604 K.Cousins   2020    513   8.31
14 00-0031345 J.Garoppolo 2019    532   8.28
15 00-0025708 M.Moore    2016    122   8.28
16 00-0033873 P.Mahomes  2019    596   8.28
17 00-0020531 D.Brees    2017    606   8.26
18 00-0029263 R.Wilson   2018    446   8.25
19 00-0033077 D.Prescott 2019    595   8.24
20 00-0029263 R.Wilson   2019    573   8.22
# i 233 more rows

```

Even the most astute of readers probably didn't expect the Harvard-educated Ryan Fitzpatrick's season as Jameis Winston's backup to appear at the top of this list. Alas, you do see the MVP seasons of Matt Ryan (2016) and Patrick Mahomes (2018), and a bunch of quarterbacks (including Ryan) coached by the great Kyle Shanahan.

Deep Passes vs Short Passes

Now, down to the business of the chapter, testing the the second part of the hypothesis “throwing deep passes is more valuable than short passes, but it’s difficult to say whether or not a quarterback is good at deep passes”. For this stability analysis, do the following steps:

1. Calculate the yards per attempt for each passer for each season.
2. Calculate the yards per attempt for each passer for the previous season.
3. Look at the correlation from the values calculated in steps 1 and 2 to see the stability.

Now, use similar code as before, but include `pass_length_air_yards` with the “group by” commands to include pass yards. With this operation, naming becomes hard.

We have you use the dataset (*play-by-play*, `pbp`), the language (either Python, `_py`, or R, `_r`), passing plays (`_p`), seasons data (`_s`), and `finally` `pass_length` (`_pl`).

For both langues, you will create a copy of the dataframe and then shift the year by adding one. Then, merge the new dataframe with the original dataframe. This will let you have the current and previous years values.

TIP

Longer names are tedious, but we have found unique names to be important so you may quickly search through code using tools like ‘find-and-replace’ to see what is occurring with your code (with ‘find’) or change names (with ‘replace’).

In Python, create `pbp_r_p_s_pl`, using several steps. First, “group by” and “aggregate” to get the mean and count:

```
## Python
pbp_py_p_s_pl = \
    pbp_py_p\
        .groupby(["passer_id", "passer", "season",
"pass_length_air_yards"])\
```

```
.agg({"passing_yards": ["mean", "count"]})
```

Next, flatten the column names and rename `passing_yards_mean` to be `ypa` and `passing_yards_count` to be `n` in order to have shorter names that are easier to work with:

```
## Python
pbp_py_p_s_pl.columns = \
    list(map("_".join, pbp_py_p_s_pl.columns.values))
pbp_py_p_s_pl \
    .rename(columns={"passing_yards_mean": "ypa",
                    "passing_yards_count": "n"}, 
            inplace=True)
```

Followed by resetting the index:

```
## Python
pbp_py_p_s_pl.reset_index(inplace=True)
```

Select only short-passing data from passers with more than 100 such plays and long-passing data for passers with more than 30 such plays:

```
## Python
q_value = (
    '(n >= 100 & ' +
    'pass_length_air_yards == "short") | ' +
    '(n >= 30 & ' +
    'pass_length_air_yards == "long")'
)
pbp_py_p_s_pl = pbp_py_p_s_pl.query(q_value).reset_index()
```

Then, create a list of columns to save (`cols_save`) and a new dataframe with only these columns (`air_yards_py`). Include a `.copy()` so edits will not be passed back to the original dataframe:

```
## Python
cols_save = \
    ["passer_id", "passer", "season",
     "pass_length_air_yards", "ypa"]
air_yards_py = \
    pbp_py_p_s_pl[cols_save].copy()
```

Next, copy `air_yards_py` to create `air_yards_lag_py` and then add one from the season using the shortcut command, `+=` and rename the `passing_yards_mean` to include `lag`:

```
## Python
air_yards_lag_py =\
    air_yards_py\
    .copy()
air_yards_lag_py["season"] += 1
air_yards_lag_py\
    .rename(columns={"ypa": "ypa_last"},\
    inplace=True)
```

Finally, `merge()` the two dataframes together to create `air_yards_both_py` use an “inner join” so only shared years will be saved and join “on” `passer_id`, `passer`, `season`, and``pass_length_air_yards``:

```
## Python
pbp_py_p_s_pl =\
    air_yards_py\
    .merge(air_yards_lag_py,
        how='inner',
        on=['passer_id', 'passer',
            'season', 'pass_length_air_yards'])
```

Check the results of your choice in Python by examining a couple of quarterback of your choice such as Tom Brady (`T.Brady`) and Aaron Rodgers (`A.Rodgers`) and only include the necessary columns to have an easier to view data frame:

```
## Python
print(
    pbp_py_p_s_pl[["pass_length_air_yards", "passer",
                    "season", "ypa", "ypa_last"]]\\
    .query('passer == "T.Brady" | passer == "A.Rodgers"')\
    .sort_values(["passer", "pass_length_air_yards", "season"])\\
    .to_string())
)
   pass_length_air_yards      passer  season       ypa  ypa_last
45           long  A.Rodgers  2019  12.092593  12.011628
47           long  A.Rodgers  2020  16.097826  12.092593
49           long  A.Rodgers  2021  14.302632  16.097826
43          short  A.Rodgers  2017   6.041475   6.693523
```

44	short	A.Rodgers	2018	6.697446	6.041475
46	short	A.Rodgers	2019	6.207224	6.697446
48	short	A.Rodgers	2020	6.718447	6.207224
50	short	A.Rodgers	2021	6.777083	6.718447
0	long	T.Brady	2017	13.264706	15.768116
2	long	T.Brady	2018	10.232877	13.264706
4	long	T.Brady	2019	10.828571	10.232877
6	long	T.Brady	2020	12.252101	10.828571
8	long	T.Brady	2021	12.242424	12.252101
1	short	T.Brady	2017	7.071429	7.163022
3	short	T.Brady	2018	7.356452	7.071429
5	short	T.Brady	2019	6.048276	7.356452
7	short	T.Brady	2020	6.777600	6.048276
9	short	T.Brady	2021	6.634697	6.777600

TIP

We suggesting using at least two players to check you code. For example, Tom Brady is the first player by *passer_id* and only looking at his values might not show a mistake that does not affect the first player in the dataframe.

In R, similar steps are taken to create `pbp_r_p_s_pl`. First, create `air_yards_r` by selecting the columns needed and arrange the dataframe:

```
## R
air_yards_r <-
  pbp_r_p |>
  select(passer_id, passer, season,
         pass_length_air_yards, passing_yards) |>
  arrange(passer_id, season,
          pass_length_air_yards) |>
  group_by(passer_id, passer,
           pass_length_air_yards, season) |>
  summarize(n = n(),
            ypa = mean(passing_yards),
            .groups = "drop") |>
  filter((n >= 100 & pass_length_air_yards == "short") |
         (n >= 30 & pass_length_air_yards == "long")) |>
  select(-n)
```

Next, create the lag data frame including a mutate to the seasons and add one:

```
## R
air_yards_lag_r <-
```

```

air_yards_r |>
mutate(season = season + 1) |>
rename(ypa_last = ypa)

```

Last, join the dataframes to create pbp_r_p_s_pl

```

## R
pbp_r_p_s_pl <-
air_yards_r |>
inner_join(air_yards_lag_r,
by = c("passer_id", "pass_length_air_yards",
"season", "passer"))

```

Check the results in R by examining passers of your choice such as Tom Brady (T.Brady) and Aaron Rodgers (A.Rodgers):

```

## R
pbp_r_p_s_pl |>
filter(passer %in% c("T.Brady", "A.Rodgers")) |>
print(n = Inf)
# A tibble: 18 × 6
  passer_id  passer  pass_length_air_yards  season    ypa  ypa_last
  <chr>       <chr>            <chr>      <dbl> <dbl>     <dbl>
1 00-0019596 T.Brady   long          2017  13.3    15.8
2 00-0019596 T.Brady   long          2018  10.2    13.3
3 00-0019596 T.Brady   long          2019  10.8    10.2
4 00-0019596 T.Brady   long          2020  12.3    10.8
5 00-0019596 T.Brady   long          2021  12.2    12.3
6 00-0019596 T.Brady   short         2017   7.07   7.16
7 00-0019596 T.Brady   short         2018   7.36   7.07
8 00-0019596 T.Brady   short         2019   6.05   7.36
9 00-0019596 T.Brady   short         2020   6.78   6.05
10 00-0019596 T.Brady  short         2021   6.63   6.78
11 00-0023459 A.Rodgers long          2019  12.1    12.0
12 00-0023459 A.Rodgers long          2020  16.1    12.1
13 00-0023459 A.Rodgers long          2021  14.3    16.1
14 00-0023459 A.Rodgers short        2017   6.04   6.69
15 00-0023459 A.Rodgers short        2018   6.70   6.04
16 00-0023459 A.Rodgers short        2019   6.21   6.70
17 00-0023459 A.Rodgers short        2020   6.72   6.21
18 00-0023459 A.Rodgers short        2021   6.78   6.72

```

TIP

We use the philosophy `Assume your code is wrong until you have convinced yourself it is correct'. Hence, we often peek at our code to make sure we understand what the code is doing versus what we

think the code is doing. Practically, this means following the advice of former US President Reagan, 'Trust but verify' your code.

The dataframes you've created (either `pbp_py_p_s_pl` in Python or `pbp_r_p_s_pl` in R) now contains 6 columns. Look at the `info()` for the dataframe in Python:

```
## Python
pbp_py_p_s_pl\
.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 259 entries, 0 to 258
Data columns (total 6 columns):
 #   Column           Non-Null Count  Dtype  
---  --  
 0   passer_id        259 non-null    object  
 1   passer           259 non-null    object  
 2   season           259 non-null    int64  
 3   pass_length_air_yards  259 non-null  object  
 4   ypa              259 non-null    float64 
 5   ypa_last         259 non-null    float64 
dtypes: float64(2), int64(1), object(3)
memory usage: 12.3+ KB
```

Or, `glimpse()` at the dataframe in R:

```
## R
pbp_r_p_s_pl |>
  glimpse()
Rows: 259
Columns: 6
$ passer_id          <chr> "00-0019596", "00-0019596", "00-
0019596", "00-00...
$ passer             <chr> "T.Brady", "T.Brady", "T.Brady",
"T.Brady", "T.B...
$ pass_length_air_yards <chr> "long", "long", "long", "long", "long",
"short", ...
$ season            <dbl> 2017, 2018, 2019, 2020, 2021, 2017,
2018, 2019, ...
$ ypa               <dbl> 13.264706, 10.232877, 10.828571,
12.252101, 12.2...
$ ypa_last          <dbl> 15.768116, 13.264706, 10.232877,
10.828571, 12.2...
```

For the 6 columns:

- `passer_id` is the unique passer identification number for the player.
- `passer` is the (potentially) non-unique first initial and last name for the passer.
- `pass_length_air_yards` is the type of pass (either long or short) you defined earlier.
- `season` is the final season in the season-pair (e.g. `season` being 2017 means you're comparing 2016 and 2017).
- `ypa` is the yards per attempt during the season stated `season` (e.g. 2017 in the previous example).
- `ypa_last` is the yards per attempt during the season previous to the stated `season` (e.g. 2016 in the previous example).

Now, that we've reminded ourselves what's in the data, let's dig in to the data and see how many quarterbacks you have. With Python, use the `passer_id` column and find the `unique()` values and then find the length of this object:

```
## Python
len(pbp_py_p_s_pl.passer_id.unique())
56
```

With R, use the `distinct` function with `passer_id` and then see how many rows exist:

```
## R
pbp_r_p_s_pl |>
  distinct(passer_id) |>
  nrow()
[1] 56
```

So we have a decent sample size of quarterbacks. We can plot this data using a scatterplot. *Scatterplots* plot points on a figure, which is in contrast to histograms that plot bins of data and boxplots that plot summaries of the data such as medians. Scatterplots allow you to “see” the data directly. The horizontal axis is called the *x-axis* and typically includes the predictor or causal variable, if one

exists. The vertical axis is called the y -axis and typically include the response or effect variables, if one exists. With our example, we will use the YPA from the previous year as the predictor for YPA in the current year. Plot this in R using `geom_point()` and call this plot `scatter_ypa_r` and then print `scatter_ypa_r` to create [Figure 2-7](#):

```
## R
scatter_ypa_r <-
  ggplot(pbp_r_p_s_pl, aes(x = ypa_last, y = ypa)) +
  geom_point() +
  facet_grid(cols = vars(pass_length_air_yards)) +
  labs(
    x = "Yards per Attempt, Year n",
    y = "Yards per Attempt, Year n + 1"
  ) +
  theme_bw() +
  theme(strip.background = element_blank())

print(scatter_ypa_r)
```

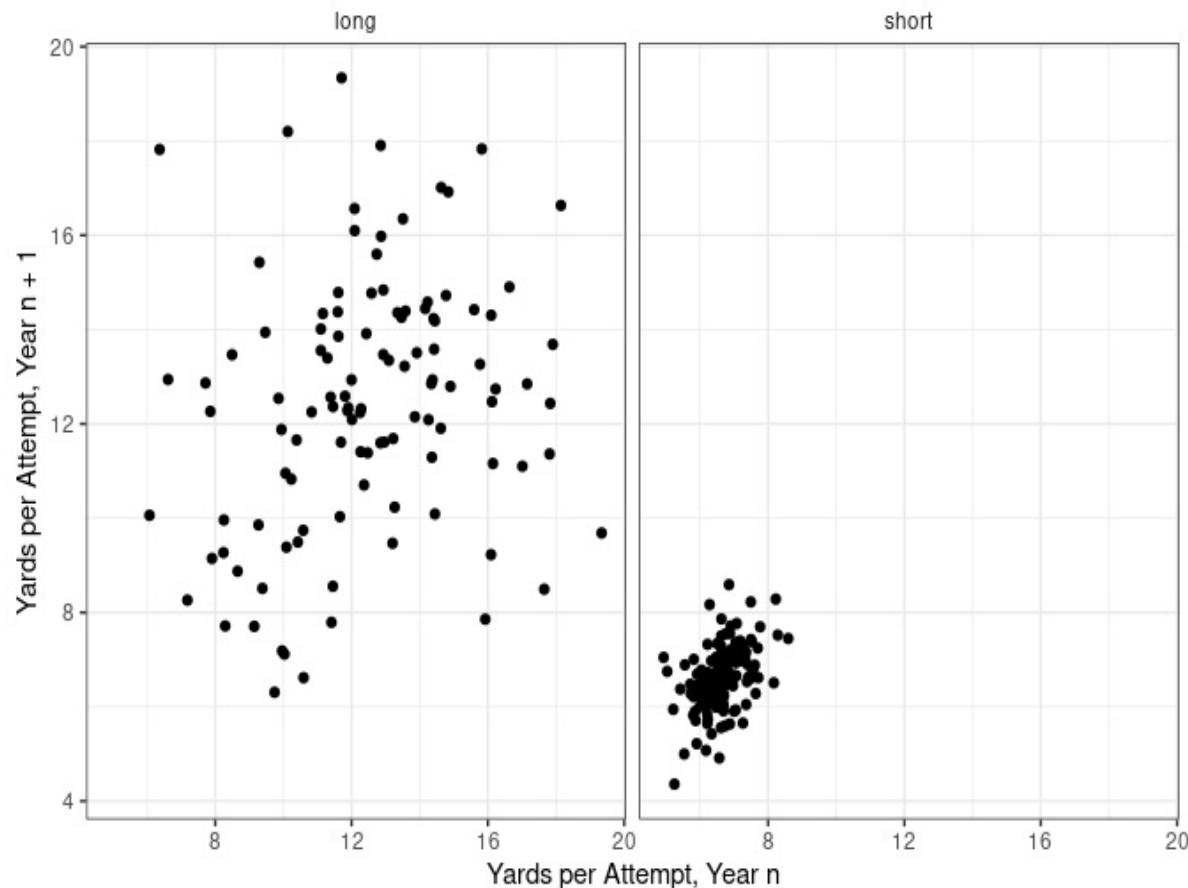


Figure 2-7. Stability of yards per attempt plotted with *ggplot2*. Notice that both sub-plots have the same x and y scales.

Figure 2-7 is encouraging for short passes. It appears that quarterbacks who are good on short passes one year are good the following year, and vice versa. Notice that the long passes are much more unwieldy. To help us examine these trends better, include a line of best fit to the data (this is why we had you save `scatter_ypa_r`, so that we could re-use here) to create **Figure 2-8**:

```
## R
# add geom_smooth() to the previously saved plot
scatter_ypa_r +
  geom_smooth(method = "lm")
```

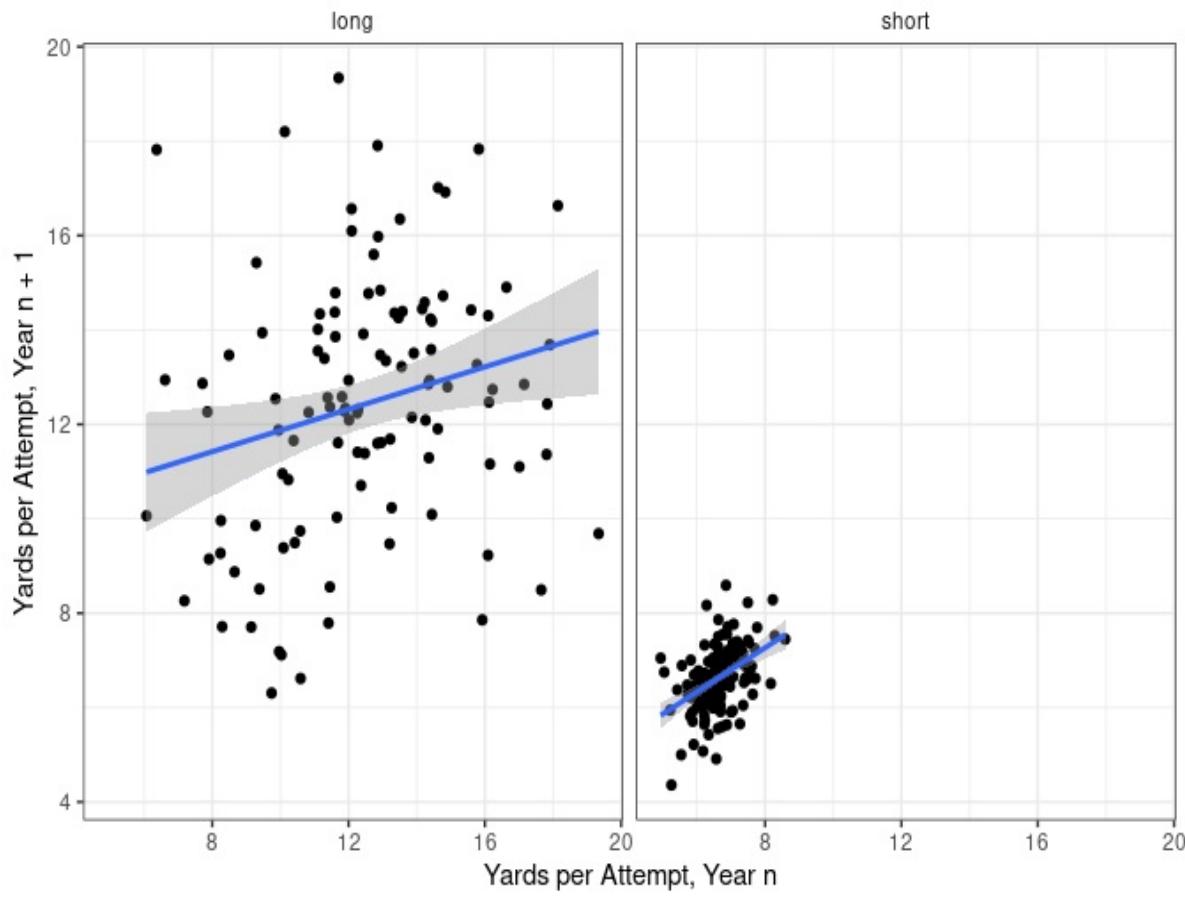


Figure 2-8. Stability of yards per attempt plotted with *ggplot2* including a trend line.

For both pass types, the lines in **Figure 2-8** have a slightly positive slope (that is to say, the lines are increasing across the plot), but this is hard to see. Obtain this estimate using the correlations so look at the numerical:

```

## R
pbp_r_p_s_pl |>
  filter(!is.na(ypa) & !is.na(ypa_last)) |>
  group_by(pass_length_air_yards) |>
  summarize(correlation = cor(ypa, ypa_last))
# A tibble: 2 × 2
  pass_length_air_yards correlation
  <chr>                  <dbl>
1 long                   0.232
2 short                  0.444

```

These figures and analyse may be repeated in Python to create [Figure 2-9](#):

```

## Python
sns.lmplot(data=pbp_py_p_s_pl,
             x="ypa",
             y="ypa_last",
             col="pass_length_air_yards");
plt.show();

```

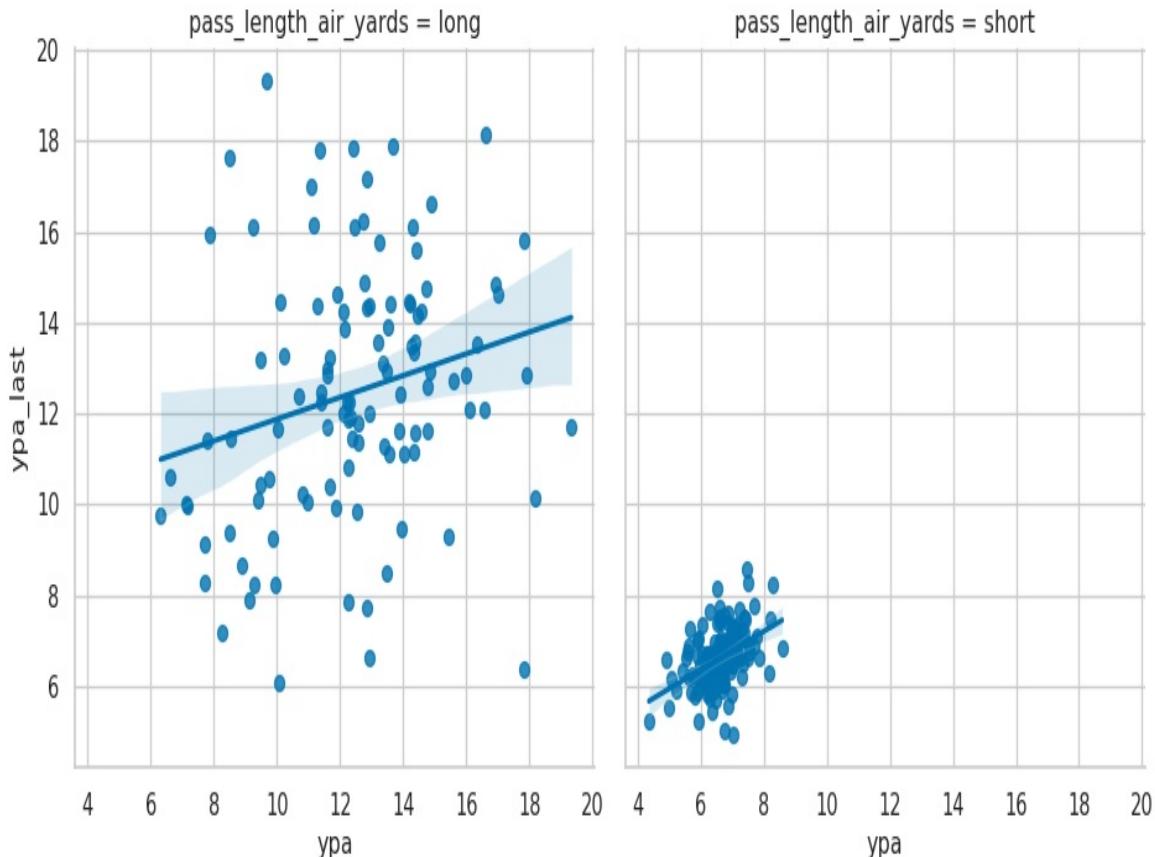


Figure 2-9. Stability of yards per attempt plotted with `seaborn` including a trend line.

Likewise, the correlation may be one using `pandas` as well:

```
## Python
pbp_py_p_s_p1\ 
    .query("ypa.notnull() & ypa_last.notnull()")\
    .groupby("pass_length_air_yards")[["ypa", "ypa_last"]]\ 
    .corr()
pass_length_air_yards
long
short
```

		ypa	ypa_last
long	ypa	1.000000	0.231867
	ypa_last	0.231867	1.000000
short	ypa	1.000000	0.443761
	ypa_last	0.443761	1.000000

The Pearson's correlation coefficient numerically captures what [Figure 2-8](#) and [Figure 2-9](#) show.

While both datasets include a decent amount of noise, vis-a-vis Pearson's correlation coefficient, a quarterback's performance on shorter passes is twice as stable as on longer passes. Thus, you can confirm the second part of the guiding hypothesis of the chapter: “throwing deep passes is more valuable than short passes, but it's difficult to say whether or not a quarterback is good at deep passes”

NOTE

A Pearson's correlation coefficient can vary from -1 to 1. In the case of stability, a number closer to +1 implies strong, positive correlations and more stability, and a number closer to 0 implies weak correlations at best (and an unstable measure). A Pearson's correlation coefficient of -1 implies a decreasing correlation and does not exist for stability but would mean a high value this year would be correlated with a low value next year.

So, What Should We Do with This Insight?

Generally speaking, noisy data is a place to look for players (or teams or units within teams) that have popup seasons that are not likely to repeat themselves. Just like a baseball player who sees a 20-point jump in his average based on a higher *BABIP* (*batting average on balls in play*) one year might be someone you want to avoid rostering in fantasy or real baseball, a weaker quarterback who generates a high yards per attempt (or EPA per pass attempt) on deep passes one

year - without a corresponding increase in such metrics on shorter passes, the more stable of the two – might be what analysts call a *regression candidate*. For example, let's look at the leaderboard for 2017 deep passing yards per attempt in Python:

```
## Python
pbp_py_p_s_p1\
    .query(
        'pass_length_air_yards == "long" & season == 2017'\
        )[['passer_id', "passer", "ypa"]]\```
    .sort_values(["ypa"], ascending=False)\```
    .head(10)
        passer_id      passer      ypa
39  00-0023436    A.Smith  19.338235
74  00-0026498    M.Stafford 17.830769
10  00-0020531    D.Brees  16.632353
171 00-0032950    C.Wentz  13.555556
31  00-0022942    P.Rivers 13.347826
0   00-0019596    T.Brady  13.264706
119 00-0029604    K.Cousins 12.847458
106 00-0029263    R.Wilson 12.738636
181 00-0033077    D.Prescott 12.585366
101 00-0028986    C.Keenum  11.904762
```

There's some good names on this list (Drew Brees, Tom Brady, Russell Wilson), but also some so-so names. What happens if you look at the same list in 2018:

```
## Python
pbp_py_p_s_p1\
    .query(
        'pass_length_air_yards == "long" & season == 2018'\
        )[['passer_id', "passer", "ypa"]]\```
    .sort_values(["ypa"], ascending=False)\```
    .head(10)
        passer_id      passer      ypa
108 00-0029263    R.Wilson  15.597403
12  00-0020531    D.Brees  14.903226
183 00-0033077    D.Prescott 14.771930
190 00-0033106    J.Goff  14.445946
33  00-0022942    P.Rivers 14.357143
143 00-0031280    D.Carr  14.339286
168 00-0032268    M.Mariota 13.941176
60  00-0026143    M.Ryan  13.465753
173 00-0032950    C.Wentz  13.222222
22  00-0022803    E.Manning 12.941176
```

Alex Smith, who was long thought of as a dink-and-dunk specialist, dropped off this list completely. He actually led the league in passer rating in 2017, before being traded by Kansas City to Washington for a third-round pick and star cornerback Kendall Fuller (there's a team that knows how to sell high!).

While there are some repeats in the list for yards per attempt on deep passes, there are many new names that emerge. Specifically, if you filter for Matt Ryan's name in the dataset, you'll find that he averaged 17.7 yards per attempt on deep passes in 2016 (when he won NFL MVP). In 2017 that value fell to 8.5, then back up to 13.5 in 2018. Did Ryan's ability drastically change during these three years, or was he subject to significant statistical variability? The math would suggest the latter. In fantasy football or betting, he would have been a *sell high* candidate in 2017 and a *buy low* candidate in 2018 as a result.

Data Science Tools Used in This Chapter

This chapter including the following topics:

- You saw how to obtaining data from multiple seasons using the `nflfastR` package either directly in R or via the `nfl_data_py` package in Python
- You saw how to change columns based upon conditions using `where` in Python and `ifelse()` statements in R.
- You saw how to `describe()` data with `pandas` and `summarize()` data in R.
- You saw how to reorder values using `sort_by()` in Python or `arrange()` in R.
- You saw how to calculate the difference between years using `merge()` in Python and `join()` in R.

Exercises with Your Data

1. Create the same histograms in “[Histograms](#)”, but for EPA per pass attempt.
2. Create the same boxplots in “[Histograms](#)”, but for EPA per pass attempt.

3. Perform the same stability analysis in “[Player-Level Stability of Passing Yards per Attempt](#)”, but for EPA per pass attempt. Do you see the same qualitative results as when you use yards per attempt? Are there players with similar yards per attempt numbers one year to the next but have drastically different EPA per pass attempt numbers across years? Where could this come from?
4. One of the reasons that data for long pass attempts is less stable than on short pass attempts is that there are fewer of them, which is largely a product of 20 yards being an arbitrary cutoff for long passes (by companies like PFF). Find a cutoff that equally splits the data and perform the same analysis. Do the results stay the same?

Suggested Readings

If you want to learn more about plotting, here are some resources that we found helpful:

- [*The Visual Display of Quantitative Information*](#) by Edward Tufte. This book is classic on how to think about data. The book does not contain code, but instead shows how to see information for data. The guidance in the book is priceless.
- [**ggplot2 package documentation**](#). For our readers using R, this is the place to start to learn more about `ggplot2`. The page includes beginner resources and links to advanced resources. The page also includes examples that are great to browse.
- [**seaborn package documentation**](#) For our readers using Python, this is the place to start for learning more about `seaborn`. The page includes beginner resources and links to advanced resources. The page also includes examples that are great to browse. The gallery on this page is especially helpful when trying to think about how to visualize data.
- [*ggplot2: Elegant Graphics for Data Analysis, Third Edition*](#) by Hadley Wickham, Danielle Navarro, and Thomas Lin Pedersen. The third edition is currently under development and accessible online. This book explains how

`ggplot2` works in great detail but also provides a good method for thinking about plotting data using words. The method to become an expert in `ggplot2` is to read this book while analyzing and tweaking each line of code presented within the book. But, this is not necessarily an easy route.

Chapter 3. Simple Linear Regression: Rushing Yards Over Expected

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Football is a contextual sport. Consider whether or not a pass is completed. This depends on a number of factors: Was the quarterback under pressure (making it harder to complete)? Was the defense expecting a pass (which would make it harder to complete)? What was the depth of the pass (completion percentage goes down with depth of target)?

What turns people off to football analytics are conclusions that they feel lack a contextual understanding of the game. “Raw numbers” can be misleading. Sam Bradford once set the NFL record for completion percentage in a season as a member of the Minnesota Vikings in 2016. This was impressive, since he joined the team early in the season as a part of a trade and had to acclimate quickly to a new environment. While that was impressive, it did not necessarily mean he was the best quarterback in the NFL that year, or even the most accurate one. For one, he averaged just 6.6 yards average depth of target that year, which was 37th in the NFL per PFF. That left his yards per pass attempt at a relatively average 7.0, tied for just 20th in football. [Chapter 4](#) provides more context for that

number as well as shows you how to adjust it yourself.

Luckily, given the great work of the people supporting `nflfastR`, you can provide your own context for metrics by applying the statistical tool known as *regression*. Through regression, you can *normalize* or *control for* variables (or “features”) that have been shown to affect the process of player production. Whether or not a feature predicts a player’s production is incredibly hard to prove in practice. Also, players emerge who come along and challenge our assumptions in this regard (such as Patrick Mahomes of the Kansas City Chiefs or Derrick Henry of the Tennessee Titans). Furthermore, data often fails to capture many factors that affect performance. As in life, you cannot account for everything, but hopefully capture the most important things. Or, as one of Richard’s professors at Texas Tech likes define as the Mick Jagger Theorem: “You can’t always get what you want, but if you try sometimes, you just might get what you need.”

The process of normalization-in both the public and private football analytics space-generally requires models that are more involved than a simple linear regression, the model covered in this this chapter. But we have to start somewhere. And, simple linear regression provides a nice start to modeling because it is both understandable and the foundation for many other types of analyses.

NOTE

Many different fields use simple linear regression, which leads to different terms. Mathematically, the predictor variable is usually x and the response variable usually y . Some synonyms for x include predictor variable, feature, explanatory variable, and independent variable. Some synonyms for y include response variable, target, and dependent variable. Likewise, medical studies often *correct for* exogenous or confounding data (*variables* to statisticians or *features* to data scientists) such as education-level, age, or other socioeconomic-data. You are learning the same concepts in this chapter and [Chapter 4](#) with the terms *normalize* or *control for*.

Simple linear regression consists of a model with a single explanatory variable that is assumed to be linearly related to a single dependent variable, or *feature*. Statistically,

That is to say, a *simple linear regression* fits the statistically “best” straight line using one independent, predictor variable to estimate a response variable as a

function of the predictor. *Simple* refers to only having one predictor variable as well an intercept, an assumption [Chapter 4](#) shows you how to relax. *Linear* refers to the straight line (compared to a curved-line or polynomial line for readers who remember high school algebra).

Regression originally referred to idea that observations will return or *regress* to the average over time, as noted by [Galton in 1877](#). For example, if a running back has above average rushing yards per carry one year, one would statistically expect them to revert or *regress* to the the league average in future years, all else being equal. The linear assumption made in many models is often onerous, but is generally fine as a first pass.

To start applying simple linear regression, you are going to work on a problem that has been solved already in the public space during the 2020 Big Data Bowl. During the 2020 Big Data Bowl, participants used *tracking data* (the positioning, direction, and orientation of all 22 on-field players every tenth of a second) to model the expected rushing yards gained on a play. This value was then subtracted from a player's actual rushing yards on a play to determine their *rushing yards over expected* (RYOE). As we talked about in the introductory chapter of this book, this kind of residual analysis is a cornerstone exercise in all of sports analytics.

The RYOE metric has since made its way onto broadcasts of actual NFL games. Additional work has been done to improve the metric, and to make a version of it that uses scouting data instead of tracking data by Tej Seth at PFF such as [an R Shiny app for RYOE](#). Regardless of the actual mechanics of the model, the broad idea is to adjust for the situation a rusher has to undergo to gain yards.

NOTE

The [Big Data Bowl](#) is the brainchild of Michael Lopez. Lopez is currently the NFL's Director of Data and Analytics. Like Eric, Lopez was previously a professor, in Lopez's case, at Skidmore College as a Professor of Statistics. Lopez's [homepage](#) contains useful tips, insight, and advice both for sports as well as careers.

To mimic RYOE, but on a much smaller scale, you will use the *yards-to-go* on a given play. Recall that each football play has a down and distance, where *down*

refers to the place in the four-down sequence a team is in to either pick up 10 yards or score either a touchdown or field goal. *Distance*, or *yards-to-go* refers to the distance left to achieve that goal, and is coded in the data as *ydstogo*.

A reasonable person would expect that the specific down and yards-to-go affect RYOE. This observation occurs because it is easier to run the ball when more yards-to-go exist because the defense will usually try to prevent longer plays. For example, when an offense faces third down and 10 yards to go, the defense is playing back in hopes of avoiding a big play. Conversely, while on second down and one yard to go, the defense is playing up to try to prevent a first down or touchdown.

For many years, teams deployed what was called a *short-yardage back*, a (usually larger) running back that would be tasked with gaining the (small) number of yards on third or fourth down when only one or two yards were required for a first down or touchdown. These players were prized in fantasy football for their abilities to *vulture* (or take credit for) touchdowns that a team's starting running back often did much of the work for their team to score. But the short-yardage backs' yards-per-carry values were not impressive compared to the starting running back. This short-yardage back's yards-per-carry lacked context compared to the starting running back's. Hence, metrics like RYOE help to normalize the context of a running back's plays.

Many example players exist from the history of the NFL. Mike Alstott, the Tampa Bay Buccaneers' second-round pick in the 1996, often served as the short-yardage back on the upstart Bucs teams of the late-90s/early-2000s. In contrast, his backfield mate, Warrick Dunn, the team's first-round choice in 1997, served in the "early-down" role. As a result, their yards per carry numbers were drastically different as members of the same team, with Alstott's being 3.7 yards and Dunn's being 4.0 yards. Thus, regression can help you account for that and make better comparisons to create metrics such as RYOE.

NOTE

The wisdom of drafting a running back in the top two rounds once, let alone in consecutive years, is a whole other topic in football analytics. We talk about the draft in great detail in [Chapter 7](#).

Understanding simple linear regression from this chapter also serves as a foundation for skills covered in other chapters, such as more complex RYOE models in [Chapter 4](#), completion percentage over expected in the passing game in [Chapter 5](#), touchdown passes per game in [Chapter 6](#), and models used to evaluate draft data in [Chapter 7](#). Many people, including the authors, call linear models both the workhorse and foundation for applied statistics and data science.

Exploratory Data Analysis

Prior to running a simple linear regression it's always good to plot the data as a part of the modeling process, using the EDA skills you learned about in [Chapter 2](#). You will do this using `seaborn` in Python or `ggplot2` in R. Before you calculate the RYOE, data needs to be loaded and wrangled. You will use the data from 2016 to 2022. First, load the packages and data.

TIP

Make sure you have installed the `statsmodels` package using `pip install statsmodels` in the terminal.

If you're using Python, use this code to load the data:

```
## Python
import pandas as pd
import numpy as np
import nfl_data_py as nfl
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt
import seaborn as sns

seasons = range(2016, 2022 + 1)
pbp_py = nfl.import_pbp_data(seasons)
```

If you're using R, use this code to load the data:

```
## R
library(tidyverse)
library(nflfastR)
```

```
pbp_r <- load_pbp(2016:2022)
```

After loading the data, select the running plays. Use the filtering criteria, `play_type == "run"`. Also, remove the plays without a rusher and replace the missing rushing yards with 0. In Python use `rusher_id.notnull()` as part of your query and then replace missing `rushing_yards` values with 0:

```
## Python
pbp_py_run =\
    pbp_py.query('play_type == "run" & rusher_id.notnull()')\
        .reset_index()
pbp_py_run\
    .loc[pbp_py_run.rushing_yards.isnull(), "rushing_yards"] = 0
```

In R, use `!is.na(rusher_id)` as part of your `filter()` step and then a `mutate()` with an `ifelse()` function to replace the missing values:

```
## R
pbp_r_run <-
  pbp_r |>
  filter(play_type == "run" & !is.na(rusher_id)) |>
  mutate(rushing_yards = ifelse(is.na(rushing_yards), 0,
rushing_yards))
```

Next, plot the raw data prior to build a model. In Python, use `displot()` from `seaborn` to create [Figure 3-1](#):

```
## Python
sns.set_theme(style="whitegrid", palette="colorblind")
sns.scatterplot(data=pbp_py_run, x="ydstogo", y="rushing_yards");
plt.show();
```

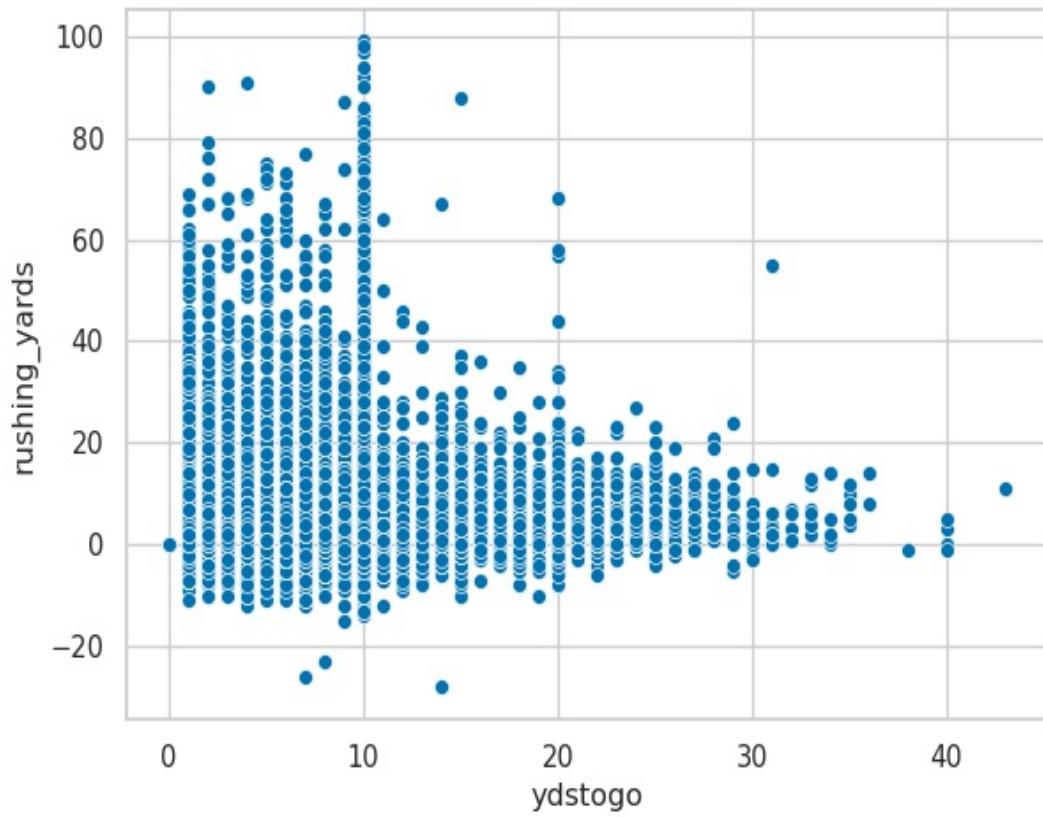


Figure 3-1. Yards-to-go plotted against rushing yards using *seaborn*.

In R, use `geom_point()` from `ggplot2` in Figure 3-2:

```
ggplot(pbp_r_run, aes(x = ydstogo, y = rushing_yards)) +  
  geom_point() +  
  theme_bw()
```

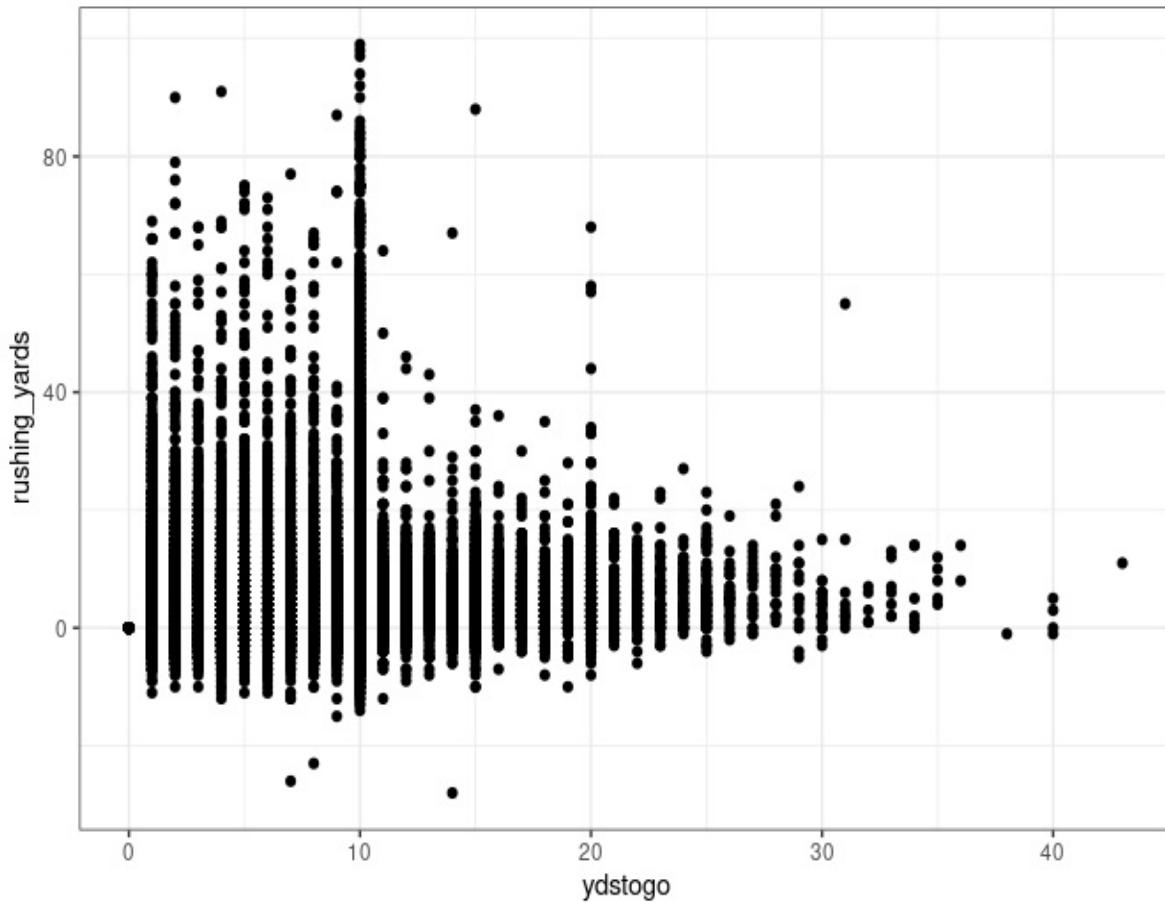


Figure 3-2. Yards-to-go plotted against rushing yards using *ggplot2*.

Figure 3-1 and Figure 3-2 are dense graph of points, and it's hard to see if a relation exists between the yards-to-go and the number of rushing yards gained on a play. There are some of things you can do to make the plot easier to read. First, add a trend line to see if the data slopes upward, downward, or neither up nor down.

In Python, use `regplot` to create Figure 3-3:

```
## Python
sns.regplot(data=pbp_py_run, x="ydstogo", y="rushing_yards");
plt.show();
```

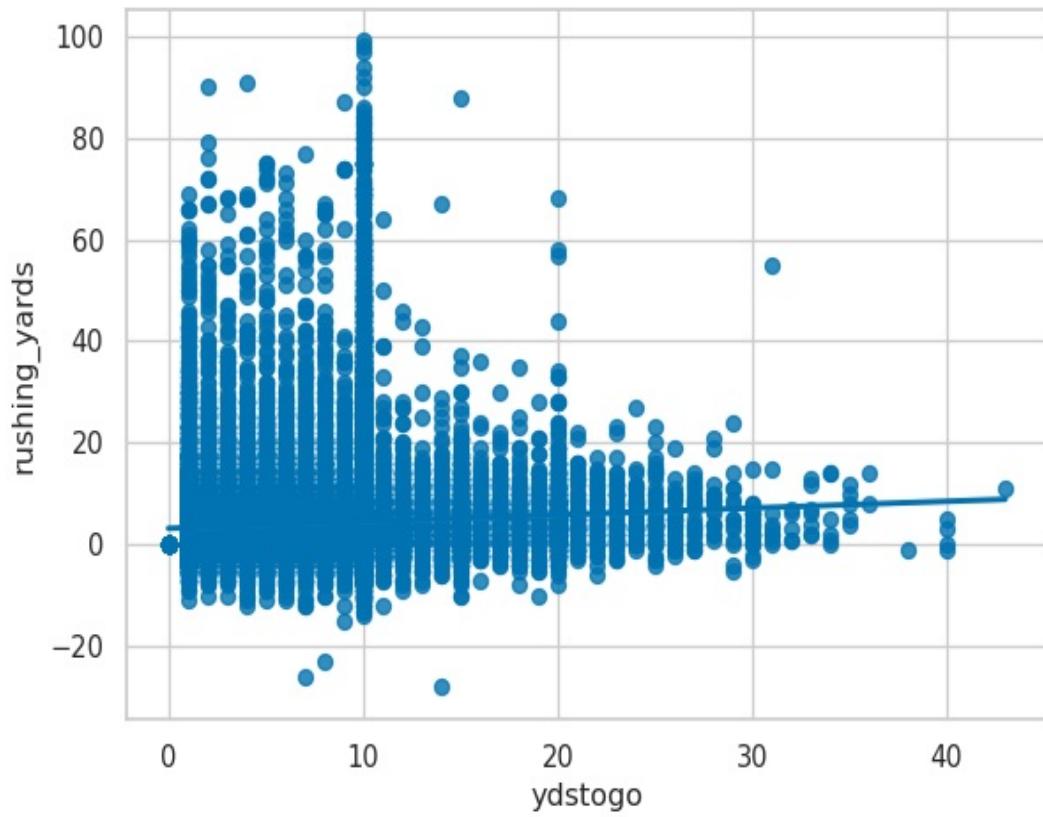


Figure 3-3. Yards-to-go plotted against rushing yards with a trend line using *seaborn*.

In R, use `stat_smooth(method = "lm")` with your code from [Figure 3-2](#) to create [Figure 3-4](#):

```
ggplot(pbp_r_run, aes(x = ydstogo, y = rushing_yards)) +  
  geom_point() +  
  theme_bw() +  
  stat_smooth(method = "lm")
```

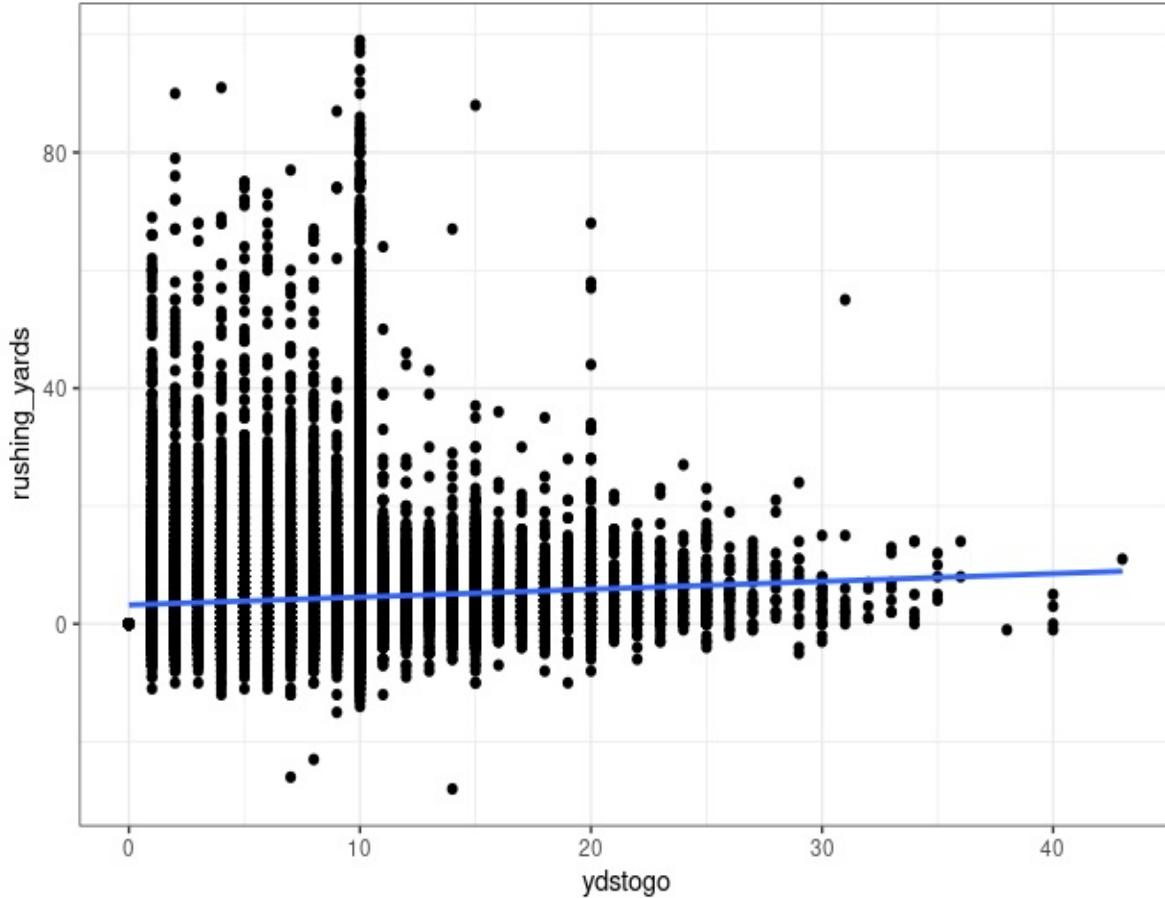


Figure 3-4. Yards-to-go plotted against rushing yards with a trendline using `ggplot2`.

In [Figure 3-3](#) and [Figure 3-4](#) and you see a see a positive slope, albeit a very small one. This shows us that rushing gains increase slightly as yards-to-go increases. Another approach to try and and examine the data would be *binning and averaging*. This borrows from the ideas of a histogram (covered in “[Histograms](#)”), but rather than using the count for each bin, an average is used for each bins. In this case, the bins are easy to define: they are the `ydstogo` values, which are integers.

Now, average over each the yards-per-carry gained in each bin. In Python, aggregate the data and then plot to create [Figure 3-5](#):

```
## Python
pbp_py_run_ave =\
    pbp_py_run.groupby(["ydstogo"])\
        .agg({"rushing_yards": ["mean"]})

pbp_py_run_ave.columns = \
```

```

list(map("_".join, pbp_py_run_ave.columns))
pbp_py_run_ave\
    .reset_index(inplace=True)

sns.regplot(data=pbp_py_run_ave, x="ydstogo", y="rushing_yards_mean");
plt.show();

```

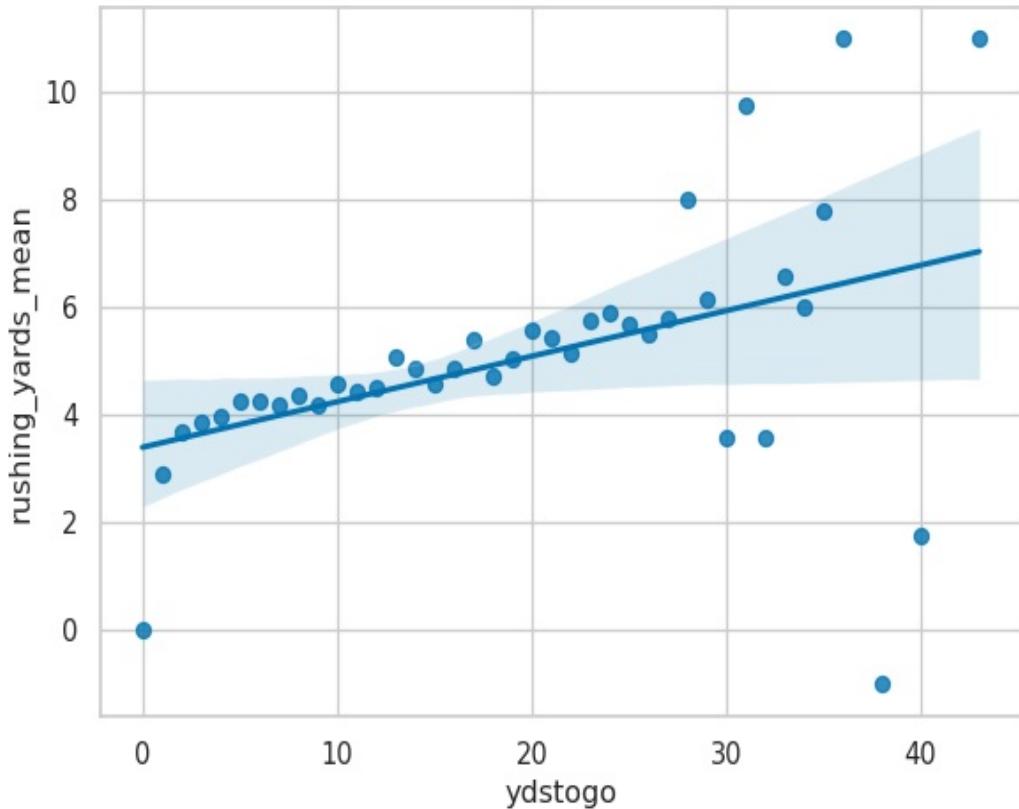


Figure 3-5. Average yards per carry plotted with seaborn.

In R, create a new variable, *yards-per-carry* (ypc) and then plot the results in Figure 3-6:

```

## R
pbp_r_run_ave <-
  pbp_r_run |>
  group_by(ydstogo) |>
  summarize(ypc = mean(rushing_yards))

ggplot(pbp_r_run_ave, aes(x = ydstogo, y = ypc)) +
  geom_point() +
  theme_bw()

```

```
stat_smooth(method = "lm")
```

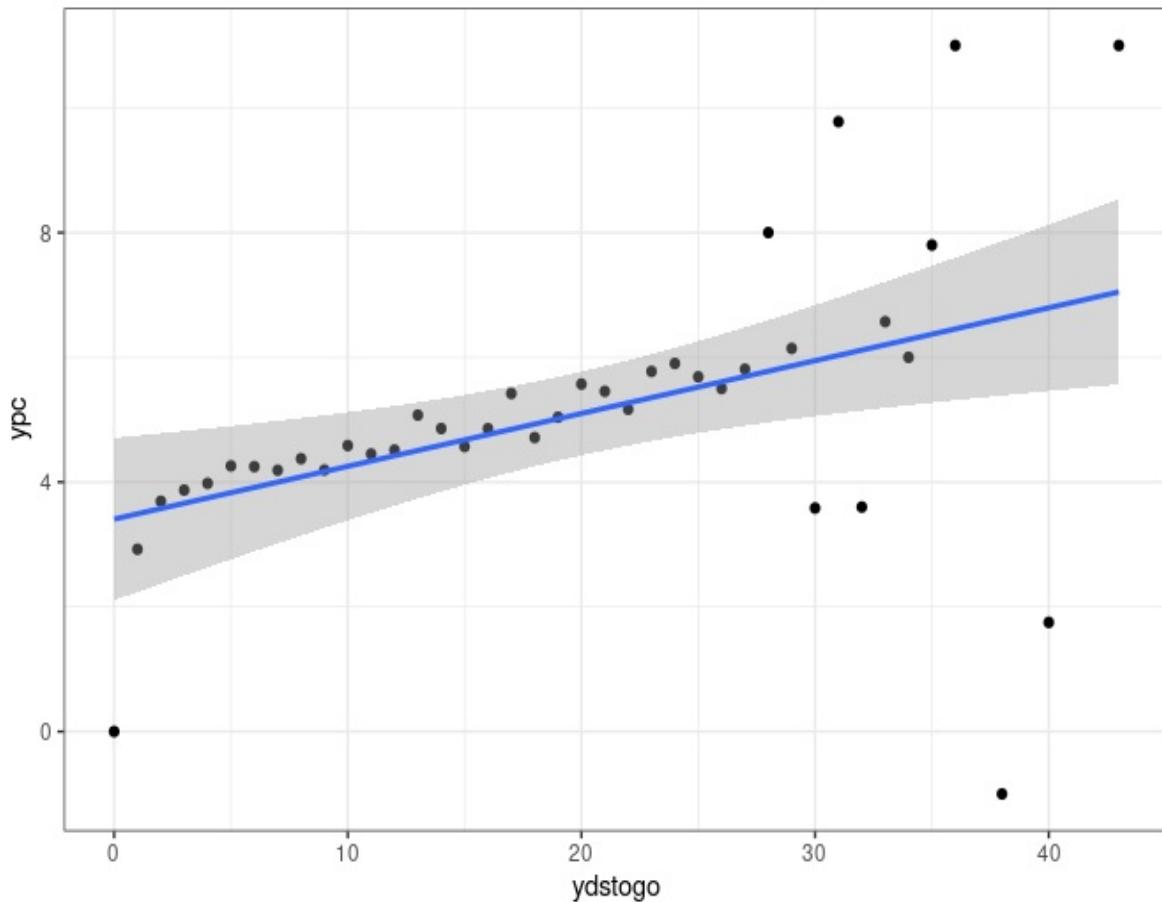


Figure 3-6. Average yards per carry plotted with `ggplot2`.

TIP

Figure 3-5 and Figure 3-6 let you see a positive linear relationship between average yards gained and yards to go. While binning and averaging is not a substitute for regressing along the entire dataset, the approach can give you insight into whether such an endeavor is worth doing in the first place and helps you to better see "the data.

Simple Linear Regression

Now that you've wrangled and interrogated the data, you're ready to run a simple linear regression. Python and R use the same formula notation for the functions we show you in this book. For example, to build a simple linear regression where `ydstogo` predicts `rushing_yards`, you use the formula

```
rushing_yards ~ 1 + ydstogo.
```

TIP

The left-hand side of the formula contains the target or response variable. The right-hand side of the formula contains the response or predictor variables. [Chapter 4](#) shows how to use multiple predictors that are separated by a + sign.

You can read this formula as `rushing_yards` are predicted by (the tilde, `~`, which is located next to the “1” key on U.S. keyboards and whose location varies on other keyboards) an intercept (`1`) and a slope parameter for yards-to-go (`ydstogo`). The `1` is an optional value to explicitly tell you where the model contains an intercept. We, and most people’s code we read, do not usually include an intercept in the formula, but we include it here to help you explicitly think about this term in the model.

NOTE

Formulas with `statsmodels` are usually similar or identical to R. This is because computer languages often borrow from other computer languages. Python’s `statsmodels` borrowed formulas from R, similar to `pandas` borrowing dataframes from R. R also borrows ideas, and R is in fact an open source recreation of the S language. As another example, both the `tidyverse` in R and `pandas` in Python both borrow syntax and ideas for cleaning data from SQL-type languages.

We are using the `statsmodels` package because it is better for statistics compared to the more popular Python package `scikit-learn` that is better for machine learning. Additionally, `statsmodels` uses similar syntax as R, which allows for you to more readily compare the two languages.

In Python, use the `statsmodels` packages’ `formula.api` imported as `smf`, to run an *ordinary least-squared regression*, `ols()`. To build the model, you will need to tell Python how to fit the regression. For this, model the number of rushing yards for play (`rushing_yards`) is predicted by an intercept (`1`) and the number of yards-to-go for the play (`ydstogo`), which is written as a formula: `rushing_yards ~ 1 + ydstogo`.

Using Python, build the model, fit the model and then look at the model's summary:

```
## Python
import statsmodels.formula.api as smf

yard_to_go_py =\
    smf.ols(formula='rushing_yards ~ 1 + ydstogo', data=pbp_py_run)

print(yard_to_go_py.fit().summary())
              OLS Regression Results
=====
=====
Dep. Variable:          rushing_yards    R-squared:
0.007
Model:                  OLS      Adj. R-squared:
0.007
Method:                 Least Squares   F-statistic:
623.7
Date:      Sun, 14 May 2023   Prob (F-statistic):
3.34e-137
Time:      09:56:26        Log-Likelihood:
-3.0107e+05
No. Observations:      92425      AIC:
6.021e+05
Df Residuals:          92423      BIC:
6.022e+05
Df Model:                   1
Covariance Type:        nonrobust
=====
=====
            coef      std err       t      P>|t|      [0.025
0.975]
-----
-----
Intercept      3.2188      0.047     68.142      0.000      3.126
3.311
ydstogo       0.1329      0.005     24.974      0.000      0.122
0.143
=====
=====
Omnibus:           81985.726  Durbin-Watson:
1.994
Prob(Omnibus):      0.000      Jarque-Bera (JB):
4086040.920
Skew:                  4.126      Prob(JB):
0.00
Kurtosis:             34.511      Cond. No.
```

20.5

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

This summary output includes a description of the model. Many of the summary items should be straightforward such as the dependent variable, (Dep . Variable), Date, and Time. The number of observations (No . Observations) relates to the degrees of freedom. The *degrees of freedom* are how many “extra” observations exist compared to the number of parameters fit.

With this model, two parameters were fit, a slope for rushing_yards and an intercept, Intercept. Hence, the Df Residuals equals the No .

Observations - 2. The R^2 value corresponds to how well the model fits the data. If the $R^2 = 1.0$, the model fits the data perfectly. Conversely, if the $R^2 = 0$, the model does not predict the data at all. In this case, the low R^2 of 0.007 shows the simple model does not predict the data well.

Other outputs of interest include the coefficient estimates for the Intercept and ydstogo. The Intercept is the number of rushing yards expected to be gained if there are zero yards to go for a first down or a touchdown (which never actually occurs in practice). The slope for ydstogo corresponds to the expected number of additional rushing yards expected to be gained for each additional yard to go. For example, a rushing play with 2 yards-to-go would be expected to produce

$$3.2(\text{intercept}) + 0.1(\text{slope}) \times 2(\text{number of yards} - \text{to} - \text{go}) = 3 \text{ yards on average.}$$

With the coefficients, a point estimate exists (coef) as well as the standard error (std err). The *standard error* (or SE or short) captures the uncertainty around the estimate for the coefficient, something Appendix B describes in greater detail. The t-value comes from a statistical distribution (specifically, the *t*-distribution) and is used to generate the SE and confidence interval (CI). The *p*-value provides the probability of obtaining the observed *t*-value assuming the null hypothesis of the coefficient being zero is true.

The *p*-value ties into null hypothesis significance testing (NHST), something that most introductory statistics courses cover, but is increasingly falling out of use by practicing statisticians. Lastly, the summary includes the 95% CI for the coefficients. The lower CI is the [0.025 column and the upper CI is the 0.975] column ($97.5 - 2.5 = 95\%$). The 95% CI should contain the true estimate for the coefficient 95% of the time, assuming the observation process is repeated many times. However, you never know *which* 5% of the time you are wrong.

A similar, linear model (`lm()`) may be fit using R and then the summary results printed:

```
## R
yard_to_go_r <-
  lm(rushing_yards ~ 1 + ydstogo, data = pbp_r_run)

summary(yard_to_go_r)
Call:
lm(formula = rushing_yards ~ 1 + ydstogo, data = pbp_r_run)

Residuals:
    Min      1Q  Median      3Q     Max 
-33.079  -3.352  -1.415   1.453  94.453 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 3.21876   0.04724   68.14   <2e-16 ***
ydstogo     0.13287   0.00532   24.97   <2e-16 ***
---
Signif. codes:  0 '****' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6.287 on 92423 degrees of freedom
Multiple R-squared:  0.006703, Adjusted R-squared:  0.006692 
F-statistic: 623.7 on 1 and 92423 DF,  p-value: < 2.2e-16
```

The general structure of the regression output differs in R compared to Python. However, the items provided are similar with the main difference occurring with formatting. R provides a the model formula, or `Call`, followed by a summary of the residuals. *Residuals* compare how well data compares to the model's fit. For RYOE, you actually will use the residuals later. Then, the summary provides the `Coefficients` and their uncertainty. Last, model details are printed, which are similar to the Python details.

WARNING

Checking degrees of freedom may seem strange to people starting out modeling. However, this can be a great check for your data to make sure the model is using all of your inputs correctly and values are not being lost. We have a friend who spends most of a semester teaching her graduate students in statistics how to compare degrees of freedom across different models. When writing this book, the Python and R versions of the `nflfastR` data were giving different values for model estimates and the degrees of freedom helped us to figure out the packages needed to be updated on our machines. Do not underestimate the utility and power of understanding degrees of freedom.

Lastly, before moving on to look at RYOE, we need to save the residuals to create an RYOE column in the data. *Residuals* are the difference between a model's expected (or predicted) output and the observed data. With `pandas` in Python, create a new column RYOE in the `pbp_py_run` dataframe from the model's residuals:

```
## Python
pbp_py_run["ryoe"] =\
    yard_to_go_py\
    .fit()\\
    .resid
```

TIP

Linear models in Python and R have capabilities and tools that we only scratch the surface of in this book. Learning the details of these tools, using resources such as those listed in “[Suggested Readings](#)”, will help you to better unlock the power of linear models.

Likewise, in R mutate `pbp_r_run` data to create a new column, `ryoe`:

```
## R
pbp_r_run <-
  pbp_r_run |>
  mutate(ryoe = resid(yard_to_go_r))
```

NOTE

R was created for teaching statistics, based upon the S language. Given this history and the state of statistics in the early 1990s, R has linear models well integrated into the language. In contrast, Python has a clone of R for linear models for statistical inference, specifically the `statsmodels` package. The

main package for models in Python, scikit-learn (`sklearn`) focuses on machine learning rather than statistical inference. Understanding the history of R and Python can provide insight into *why* the languages exist as they do as well as leveraging their respective strengths. We would also argue that if all one needs and wants to do is fit regression models for statistical inference, R would be the better software choice.

Who Was the Best in RYOE?

Now, look at the leaderboard for RYOE from 2016 to 2022, first in total yards over expected and average yards over expected per carry. Like the passer data in [Chapter 2](#), you will need to group by both the `rusher` and `rusher_id` because some players have the same last name and first initial.

With `pandas`, group by `seasons`, `rusher_id`, and `rusher`. Then, aggregate `ryoe` with the count, sum, and mean and `rushing_yards` with the mean. This gives the following columns:

- The **count** of is the *number of carries a rusher has*.
- The **sum** of RYOE is the *total RYOE*.
- The **mean** of RYOE is the *RYOE per carry*.
- The **mean** of rushing yards is the *yards-per-carry*.

Then, flatten the columns and reset the index to make the dataframe easier to work with.

NOTE

`pandas` usually requires changes to dataframes to be re-written. For example, `df = df.reset_index()` would be required to save the new dataframe after the index has been reset. Some functions have a shortcut option, `inplace=True`) so this is not needed. For example, `df.reset_index(inplace=True)` could be used instead.

Next, rename the columns to give them football-specific names. Lastly, query the result to only print players with more than 50 carries:

```

## Python
ryoe_py =\
    pbp_py_run\
    .groupby(["season", "rusher_id", "rusher"])\\
    .agg({
        "ryoe": ["count", "sum", "mean"],
        "rushing_yards": "mean"
    })

ryoe_py.columns = \
    list(map("_".join, ryoe_py.columns))
ryoe_py.reset_index(inplace=True)

ryoe_py =\
    ryoe_py\
    .rename(columns={
        "ryoe_count": "n",
        "ryoe_sum": "ryoe_total",
        "ryoe_mean": "ryoe_per",
        "rushing_yards_mean": "yards_per_carry",
    })
    .query("n > 50")

print(ryoe_py.sort_values("ryoe_total", ascending=False))
    season   rusher_id      rusher     n   ryoe_total   ryoe_per
yards_per_carry
1989      2021  00-0036223  J.Taylor  332  417.501295  1.257534
5.454819
1440      2020  00-0032764  D.Henry  397  362.768406  0.913774
5.206549
1258      2019  00-0034796  L.Jackson 135  353.652105  2.619645
6.800000
1143      2019  00-0032764  D.Henry  387  323.921354  0.837006
5.131783
1474      2020  00-0033293  A.Jones  222  288.358241  1.298911
5.540541
...
...
419       2017  00-0029613  D.Martin  139 -198.461432 -1.427780
2.920863
122       2016  00-0029613  D.Martin  144 -199.156646 -1.383032
2.923611
675       2018  00-0027325  L.Blount  155 -247.528360 -1.596957
2.696774
1058      2019  00-0030496  L.Bell   245 -286.996618 -1.171415
3.220408
267       2016  00-0032241  T.Gurley  278 -319.803875 -1.150374
3.183453

[534 rows x 7 columns]

```

To print the entire table in Python once, run `print(ryoe_py.query("n > 50").to_string())`, something we did not do in order to save space. Alternatively, you can change the printing in your entire session by using the `pandas set_option()` function. For example, `pd.set_option("display.min_rows", 10)` would always print 10 rows.

With R, use the `pbp_r_run` data, group by `season`, `rusher_id`, and `rusher`. Then, `summarize()` to get the number per group, total RYOE, average RYOE, and yards-per-carry. Lastly, filter to only include players with more than 50 carries:

```
## R
ryoe_r <-
  pbp_r_run |>
  group_by(season, rusher_id, rusher) |>
  summarize(
    n = n(),
    ryoe_total = sum(ryoe),
    ryoe_per = mean(ryoe),
    yards_per_carry = mean(rushing_yards)
  ) |>
  arrange(-ryoe_total) |>
  filter(n > 50)

ryoe_r
# A tibble: 534 × 7
# Groups:   season, rusher_id [534]
  season rusher_id  rusher      n ryoe_total ryoe_per
  <dbl> <chr>       <chr>     <int>      <dbl>      <dbl>
<dbl>
  1 2021 00-0036223 J.Taylor     332      418.      1.26
  5.45
  2 2020 00-0032764 D.Henry     397      363.      0.914
  5.21
  3 2019 00-0034796 L.Jackson   135      354.      2.62
  6.8
  4 2019 00-0032764 D.Henry     387      324.      0.837
  5.13
  5 2020 00-0033293 A.Jones     222      288.      1.30
  5.54
  6 2019 00-0031687 R.Mostert    190      282.      1.48
  5.83
  7 2016 00-0033045 E.Elliott    344      279.      0.810
```

```

5.10
8 2021 00-0034791 N.Chubb      228      276.    1.21
5.52
9 2022 00-0034796 L.Jackson    73       276.    3.78
7.82
10 2020 00-0034791 N.Chubb     221      254.    1.15
5.48
# i 524 more rows

```

We did not have you print out all the tables, to save space. However, using `|> print(n = Inf)` at the end would allow you to see the the entire table in R. Alternatively, you could change all printing for an R session by running `options(pillar.print_min = n)`.

For the filtered lists we had you only print the lists with players that carried the ball 50 or more times to leave out outliers as well as to save page space. We don't have to do that for total yards since players with so few carries will not accumulate that many rushing yards over expected, anyway.

By total rushing yards over expected, 2021 Jonathan Taylor was the best running back in football since 2016, generating over 400 RYOE, following by the aforementioned Henry, who generated 374 RYOE during his 2,000-yard 2020 season. The third player on the list, Lamar Jackson, is actually a quarterback, who in 2019 earned the NFL's Most Valuable Player award both rushing for over 1,200 yards (an NFL record for a quarterback) and leading the league in touchdown passes. In April of 2022 Jackson signed the richest contract in NFL history for his efforts.

One interesting quirk of the NFL data is that only *designed runs* for quarterbacks (plays that are actually running plays and not broken down passing plays where the quarterback pulls the ball down and runs with it) are counted in this data set. So Jackson generating this much RYOE on just a subset of his runs is incredibly impressive.

Next, sort the data by RYOE per carry. We only include code for R, but the previous Python code is readily adaptable:

```

## R
ryoe_r |>
  arrange(-ryoe_per)
# A tibble: 534 × 7

```

```

# Groups:   season, rusher_id [534]
  season rusher_id rusher      n ryoef_total ryoef_per
  <dbl> <chr>       <chr> <int>    <dbl>      <dbl>
<dbl>
  1 2022 00-0034796 L.Jackson     73    276.      3.78
7.82
  2 2019 00-0034796 L.Jackson     135    354.      2.62
6.8
  3 2019 00-0035228 K.Murray      56    122.      2.17
6.5
  4 2020 00-0034796 L.Jackson     121    249.      2.06
6.26
  5 2021 00-0034750 R.Penny       119    229.      1.93
6.29
  6 2022 00-0036945 J.Fields      85    160.      1.88
7 2022 00-0033357 T.Hill        96    178.      1.86
5.99
  8 2021 00-0034253 D.Hilliard    56    101.      1.80
6.25
  9 2022 00-0034750 R.Penny       57    99.2      1.74
6.07
10 2019 00-0034400 J.Wilkins     51    87.8      1.72
6.02
# i 524 more rows

```

Looking at RYOE per carry yields three Jackson years in the top four, sandwiching a year by Kyler Murray, who is also a quarterback. Murray was the first-overall pick in the 2019 NFL draft and elevated an Arizona Cardinals offense from the worst in football in 2018 to a much more respectable standing in 2019, through a combination of running and passing. Rashaad Penny, the Seahawks first-round pick in 2018, finally emerged in 2021 to earn almost two yards over expected per carry while leading the NFL in yards per carry overall (6.3). This earned Penny a second contract with Seattle the following offseason.

A reasonable question we should ask is whether total yards or yards-per-carry are a better measure of a player's ability. There is a general consensus around the idea that "volume is earned" both by fantasy football analysts and draft analysts alike. The idea is that there is hidden signal in the data when a player plays enough to generate a lot of carries.

Data is an incomplete representation of reality, and there are things that players do that aren't captured. If a coach plays a player a lot it's a good indication that that player is a good player. Furthermore, there is a negative relationship

generally between volume and efficiency for that same reason. If a player is good enough to play a lot, the defense is able to key on him more easily and reduce his efficiency. This is why we often see backup running backs have high yards per carry values relative to the starter in front of them (for example, look at Tony Pollard and Ezekiel Elliott for the Dallas Cowboys 2019-2022). There are other factors at play as well (such as Zeke's draft status and contract) that don't necessarily make volume the be-all-end-all, but it's important to inspect.

Is RYOE a Better Metric?

Any time you create a new metric for player or team evaluation in football, you have to test its predictive power. You can put as much thought into your new metric as the next person, and the adjustments, like in this chapter, can be well-founded. But if the metric is not more stable than previous iterations of the evaluation, you either have to conclude that the work was in vain, or that the underlying context that surrounds a player's performance is actually the entity that carries the signal. Thus, you're attributing too much of what is happening in terms of production on the individual player.

In [Chapter 2](#), you conducted stability analysis on passing data. Now, you will look at RYOE per carry for players with 50 or more carries versus traditional yards per carry values. We don't have you look at total RYOE or rushing yards since that embeds two measures of performance (volume and efficiency). Volume is in both measures, which muddies things.

This code is similar to that from "[Deep Passes vs Short Passes](#)" and we do not provide a walk through here, simply the code with comments. Hence, we refer you back to that section for details.

In Python, use this code

```
## Python
# only keep columns needed
cols_keep = \
    ["season", "rusher_id", "rusher",
     "ryoe_per", "yards_per_carry"]

# create current dataframe
ryoe_now_py = \
    ryoe_py[cols_keep].copy()
```

```

# create last-year's dataframe
ryoe_last_py =\
    ryoe_py[cols_keep].copy()

# rename columns
ryoe_last_py\
    .rename(columns = {'ryoe_per': 'ryoe_per_last',
                       'yards_per_carry': 'yards_per_carry_last'},
            inplace=True)

# add 1 to season
ryoe_last_py["season"] += 1

# merge together
ryoe_lag_py =\
    ryoe_now_py\
        .merge(ryoe_last_py,
               how='inner',
               on=['rusher_id', 'rusher',
                    'season'])

```

Lastly, examine the correlation for yards-per-carry:

```

## Python
ryoe_lag_py[["yards_per_carry_last", "yards_per_carry"]].corr()
            yards_per_carry_last  yards_per_carry
yards_per_carry_last           1.00000      0.32261
yards_per_carry                 0.32261      1.00000

```

Repeat with RYOE:

```

## Python
ryoe_lag_py[["ryoe_per_last", "ryoe_per"]].corr()
            ryoe_per_last  ryoe_per
ryoe_per_last           1.000000  0.348923
ryoe_per                  0.348923  1.000000

```

With R, use this code:

```

## R
# create current dataframe
ryoe_now_r <-
    ryoe_r |>
    select(-n, -ryoe_total)

```

```

# create last-year's dataframe
# and add 1 to season
ryoe_last_r <-
  ryoe_r |>
  select(-n, -ryoe_total) |>
  mutate(season = season + 1) |>
  rename(ryoe_per_last = ryoe_per,
         yards_per_carry_last = yards_per_carry)

# merge together
ryoe_lag_r <-
  ryoe_now_r |>
  inner_join(ryoe_last_r,
              by = c("rusher_id", "rusher", "season")) |>
  ungroup()

```

Then, select the two yards-per-carry columns and examine the correlation:

```

## R
ryoe_lag_r |>
  select(yards_per_carry, yards_per_carry_last) |>
  cor(use = "complete.obs")
      yards_per_carry yards_per_carry_last
yards_per_carry           1.0000000    0.3226097
yards_per_carry_last       0.3226097    1.0000000

```

Repeat the correlation with the RYOE columns:

```

## R
ryoe_lag_r |>
  select(ryoe_per, ryoe_per_last) |>
  cor(use = "complete.obs")
      ryoe_per ryoe_per_last
ryoe_per      1.0000000    0.3489235
ryoe_per_last 0.3489235    1.0000000

```

These results show that, for players with more than 50 rushing attempts in back-to-back seasons, this version of RYOE per carry is slightly more stable year-to-year than yards per carry (because the correlation coefficient is larger). This means that yards per carry includes with it some information inherent to the situations in which a running back is put, which can vary some year-to-year. After extracted out, our new metric for running back performance is slightly more predictive year-to-year.

As far as the question of whether (or, more accurately, how much) running backs matter, the difference between the correlation coefficients above suggest that there's not much in the way of a conclusion to be drawn so far. Prior to this chapter, you've really only looked at statistics for one position (quarterback), and the more-stable metric in the profile for that position (yards per pass attempt on short passes, with r values nearing 0.5) is much more stable than yards per carry and RYOE. Furthermore, you haven't done a thorough analysis of the running game relative to the passing game, which is essential to round out any argument that running backs are or are not important.

The key questions for football analysts about running backs are:

1. Are running back contributions valuable?
2. Are their contributions repeatable across years?

In [Chapter 4](#), you will add variables to control for other factors affecting the running game. For example, the “down” part of down and distance is surely important, because a defense will play even tighter on fourth down and one yard to go than it will on third down and that same one yard to go. A team trailing (or losing) by 14 points will have an easier time of running the ball than a team up by 14 points-all else being equal-because a team that is ahead might be playing “prevent” defense, a defense that is willing to allow yards to their opponent, just not *too* many yards.

Data Science Tools Used in This Chapter

This chapter including the following topics:

- You saw how to fit a simple linear regression in Python using `OLS()` or in R using `lm()`.
- You learned how to understand and read the coefficients from a simple linear regression.
- You saw how to plot a simple linear regression using `regplot` with `seaborn` or `geom_smooth()` in R.
- You saw how to use correlations using the `corr()` function in Python and

R to conduct stability analysis.

Exercises

1. What happens if you repeat the correlation analysis with 100 carries as the threshold? What happens to the differences in r values?
2. Assume all of Mike Alstott's carries were on third down and 1 yard to go, while all of Warrick Dunn's carries came on first down and 10 yards to go. Is that enough to explain the discrepancy in their yards per carry values (3.7 versus 4.0)? Use the coefficient from the simple linear model in this chapter to understand this question.
3. What happens if you repeat the analyses in this chapter with yards to go to the endzone (`yardline_100`) as your feature?
4. Repeat the processes within this chapter with receivers and the passing game. To do this, you have to filter by `play_type == "pass"` and `receiver_id` not being NA or null.

Suggested Readings

Eric wrote several articles on running backs and running back value while at PFF. Examples articles include:

- The NFL's best running backs on perfectly and non-perfectly blocked runs in 2021
- Are NFL running backs easily replaceable: the story of the 2018 NFL season
- Explaining Dallas Cowboys RB Ezekiel Elliott's 2018 PFF Grade

Additionally, many books exist on regression and introductory statistics and “Future readings” lists several introductory statistics books. For regression, here are some books we found useful:

- *Regression and Other Stories* (Cambridge University Press, 2020) by

Andrew Gelman, Jennifer Hill, and Aki Vehtari. This book shows how to apply regression analysis to real world problems. For people looking for more *worked* case studies, we recommend this book to help you learn how to think about applying regression.

- *Regression Modeling Strategies: With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis* (Springer, 2nd edition, 2015) by Frank Harell. This book helped one of the authors think through the world of regression modeling. It is advanced, but provides a good oversight into regression analysis. The book is written at an advanced undergraduate/introductory graduate-level. Although hard, working through this book provides mastery of regression analysis.

Chapter 4. Multiple Regression: Rushing Yards Over Expected

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

In the previous chapter you looked at rushing yards through a different lens, by controlling for the number of yards needed to gain for a first down or a touchdown. You found that common sense prevailed; the more yards a team needed to earn for a first down or a touchdown, the easier it was to gain yards on the ground. This told you that adjusting for such things was going to be an important part of understanding running back play.

A limitation of simple linear regression is that there will be more than one important variable to adjust for in the running game. While distance to first down or touchdown matters a great deal, perhaps the down is important as well. A team is more likely to run the ball on first down and 10 yards to go than they are to run the ball on third down and 10 yards to go, and hence the defense is more likely to be more geared up to stop the run on the former than the latter.

Another example is point differential. There are multiple ways in which the score of the game influences the expectation, as a team that is ahead by a lot of points will not crowd the line of scrimmage as much as a team that is locked into a close game with their opponent. In general, there are a plethora of variables

that need to be *controlled for* when evaluating a football play. The way we do this is through multiple linear regression.

Definition of Multiple Linear Regression

We know that running the football isn't just affected by one thing, so we need to build a model that predicts rushing yards, but includes more features to account for other things that might affect the prediction. Hence, to build upon [Chapter 3](#), we need multiple regression.

Multiple regression, estimates for the effect of several (*multiple*) predictors on a single response using a linear combination (or *regression*) of the predictor variables. [Chapter 3](#) presented *simple linear regression*, which is a special case of multiple regression. In a simple linear regression, two parameters exist: an *intercept* (or average value) and a *slope*. These model the effect of a continuous predictor on the response. In [Chapter 3](#), the Python/R formula for your simple linear regression had rushing yards predicted by an intercept and yards-to-go as well as the intercept: `rushing_yards ~ 1 + ydstogo`.

However, you may be interested in multiple predictor variables. For example, consider the multiple regression where you “correct” for down when estimating expected rushing yards. You would use an equation (or formula) where rushing yards are predicted by yards-to-go and down: `rushing_yards ~ ydstogo + down`. Yards-to-go is an example of a *continuous* predictor variable.

Informally, think of *continuous predictors* as numbers such as a players' weights like 135, 302, or 274. People sometimes use the term “slope” for continuous predictor variables. Down is an example of *discrete* predictor variable.

Informally, think of discrete predictors as groups or categories such as position like running back or quarterback. By default, the `formula` option in `statsmodels` and base R treat discrete predictors as *contrasts*.

Let's dive into the example formula, `rushing_yards ~ ydstogo + down`. R would estimate an intercept for the lowest (or alphabetically first) down and then the *contrast* or difference for the other downs. For example, four predictors would be estimated: an intercept that is the mean `rushing_yards` for yards-to-go, a contrast for second downs compared to first down, a contrast for third downs compared to first down, and a contrast for fourth downs

compared to first down. To see this, look at the *design matrix* or *model matrix* in R (Python has similar features, but not shown here). First, create a demonstration dataset:

```
## R
library(tidyverse)

demo_data_r <- tibble(down = c("first", "second"),
                      ydstogo = c(10, 5))
```

Then, create a model matrix using the formula's right hand side and the `model.matrix()` function:

```
## R
model.matrix(~ ydstogo + down,
              data = demo_data_r)
  (Intercept) ydstogo downsecond
1             1        10         0
2             1         5         1
attr("assign")
[1] 0 1 2
attr("contrasts")
attr("contrasts")$down
[1] "contr.treatment"
```

Notice how there are three columns, an intercept, a slope for `ydstogo`, and a contrast for second down (`downsecond`).

However, you can also estimate an intercept for each down using a `-1` such as `rushing_yards ~ ydstogo + down - 1`. This would estimate four predictors: an intercept for first down, an intercept for second down, an intercept for third down, and an intercept for fourth down. Use R to look at the example model matrix:

```
## R
model.matrix(~ ydstogo + down - 1,
              data = demo_data_r)
  ydstogo downfirst downsecond
1       10        1         0
2        5        0         1
attr("assign")
[1] 1 2 2
attr("contrasts")
```

```
attr(, "contrasts")$down  
[1] "contr.treatment"
```

Notice how there are the same number of columns as before, but each down has its own column.

WARNING

Computers languages such as Python and R get confused by some groups such as `down`. The computer tries to treat these predictors as continuous such as the numbers 1, 2, 3, and 4 rather than `first down`', `second down`', `third down`', and `fourth down`'. In `pandas you will change `down` to be a string (`str`) and in R you will change `down` to be a character.

The formula for multiple regression allows for many discrete and continuous predictors. When multiple discrete predictors are present, such as `down + team`, the first variable, `down` in this case, can either be estimated as an intercept or contrast parameters. All other discrete predictors are estimated as contrasts with the first groupings treated as part of the intercept. Rather than getting caught up in thinking about “slopes” and “intercepts”, you can use the term *coefficients* to describe the estimated predictor variables for multiple regression.

Exploratory Data Analysis

In the case of rushing yards, you’re going to use the following variables as *features* in the multiple linear regression model: `down` (`down`), distance (`ydstgo`), yards to go to the endzone (`yardline_100`), run location (`run_location`), and score differential (`score_differential`). There are other variables that could also be used, of course, but for now you’re using these variables in large part because they all affect rushing yards in some way or the other.

First, load in the data and packages you will be using, filter for only the run data (like you did in [Chapter 3](#)), and also remove plays that were not a regular down. Do this with Python:

```

import pandas as pd
import numpy as np
import nfl_data_py as nfl
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt
import seaborn as sns

seasons = range(2016, 2022 + 1)
pbp_py = nfl.import_pbp_data(seasons)

pbp_py_run = \
    pbp_py\
        .query('play_type == "run" & rusher_id.notnull() & ' +
               "down.notnull() & run_location.notnull()")\
        .reset_index()

pbp_py_run\
    .loc[pbp_py_run.rushing_yards.isnull(), "rushing_yards"] = 0

```

Or with R:

```

library(tidyverse)
library(nflfastR)

pbp_r <- load_pbp(2016:2022)

pbp_r_run <-
  pbp_r |>
    filter(play_type == "run" & !is.na(rusher_id) &
           !is.na(down) & !is.na(run_location)) |>
    mutate(rushing_yards = ifelse(is.na(rushing_yards),
                                 0,
                                 rushing_yards
    ))

```

First, let look at a histogram for down and rushing yards gained with Python and create [Figure 4-1](#):

```

## Python
# Change theme for chapter
sns.set_theme(style="whitegrid", palette="colorblind")

# Change down to be an integer
pbp_py_run.down =\
    pbp_py_run.down.astype(str)

# Plot rushing yards by down

```

```

g = \
    sns.FacetGrid(data=pbp_py_run,
                   col="down", col_wrap=2);
g.map_dataframe(sns.histplot, x="rushing_yards");
plt.show();

```

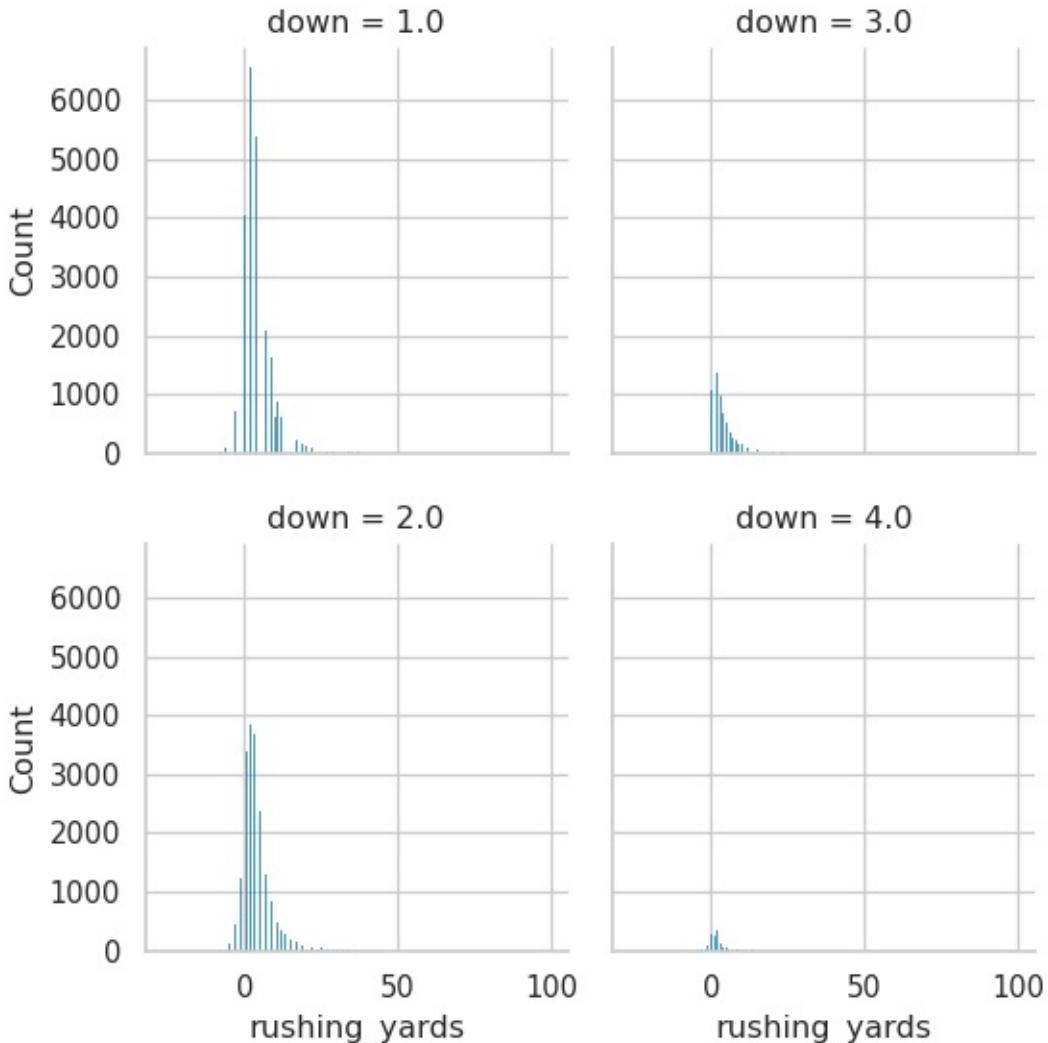


Figure 4-1. Histogram of rushing yards by downs with seaborn.

Or [Figure 4-2](#) with R:

```

## R
# Change down to be an integer
pbp_r_run <-
  pbp_r_run |>
  mutate(down = as.character(down))

```

```
# Plot rushing yards by down
ggplot(pbp_r_run, aes(x = rushing_yards)) +
  geom_histogram(binwidth = 1) +
  facet_wrap(vars(down), ncol = 2,
             labeller = label_both) +
  theme_bw() +
  theme(strip.background = element_blank())
```

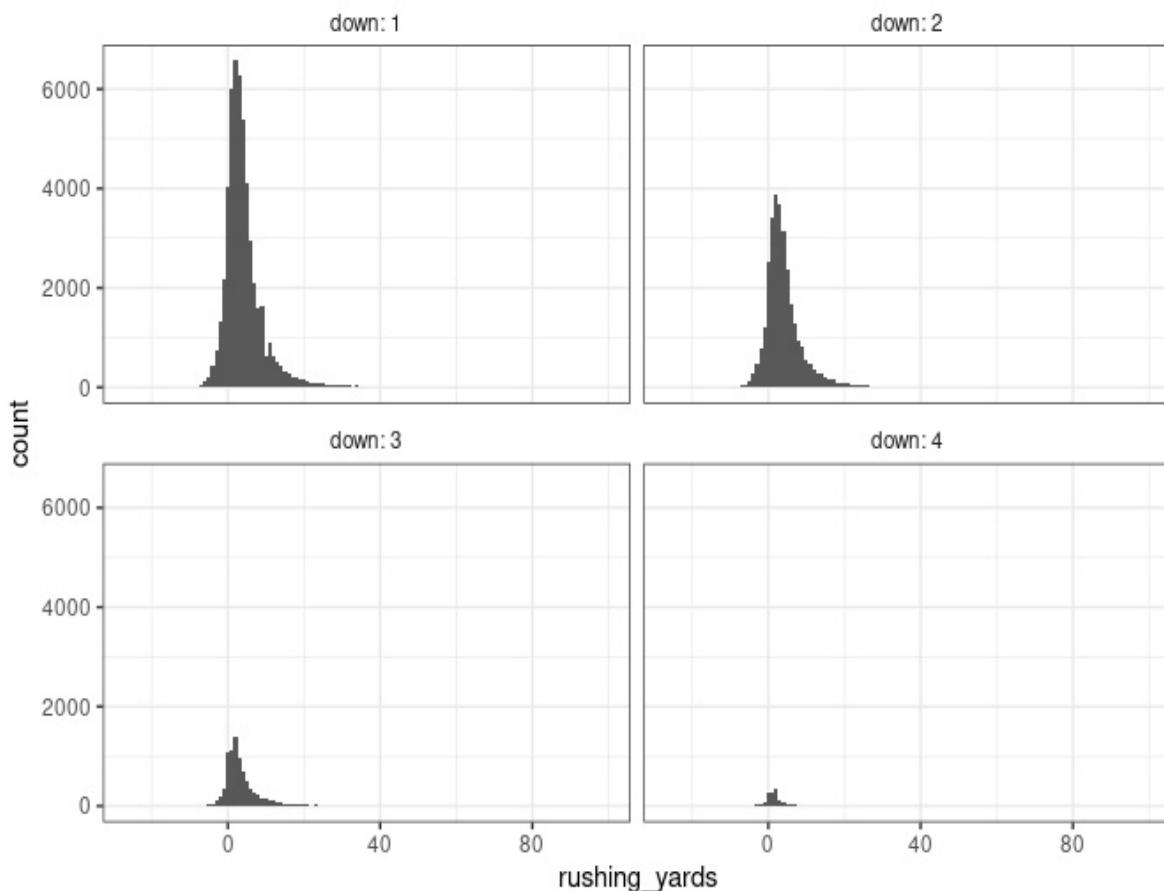


Figure 4-2. Histogram of rushing yards by downs with `ggplot2`.

This is interesting, as it looks like down decreases rushing yards. However, there is a confounder, which is that rushes often happen on late downs with smaller distances. Lets look at only situations where `ydstogo == 10`. In Python create [Figure 4-3](#):

```
## Python
sns.boxplot(data=pbp_py_run.query("ydstogo == 10"),
             x="down",
             y="rushing_yards");
plt.show()
```

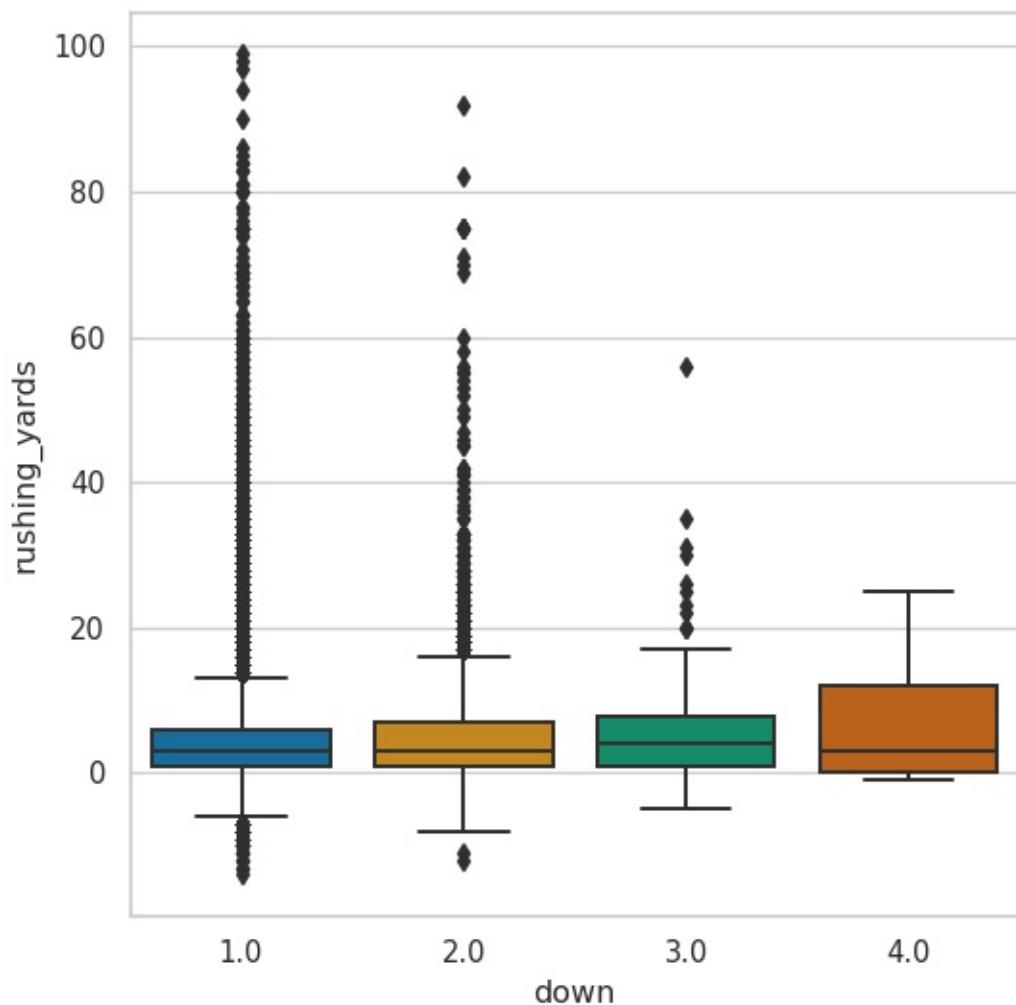


Figure 4-3. Boxplot of rushing yards by downs with *seaborn* for plays with 10 yards to go.

Or with R create Figure 4-4:

```
## R
pbp_r_run |>
  filter(ydstogo == 10) |>
  ggplot(aes(x = down, y = rushing_yards)) +
  geom_boxplot() +
  theme_bw()
```

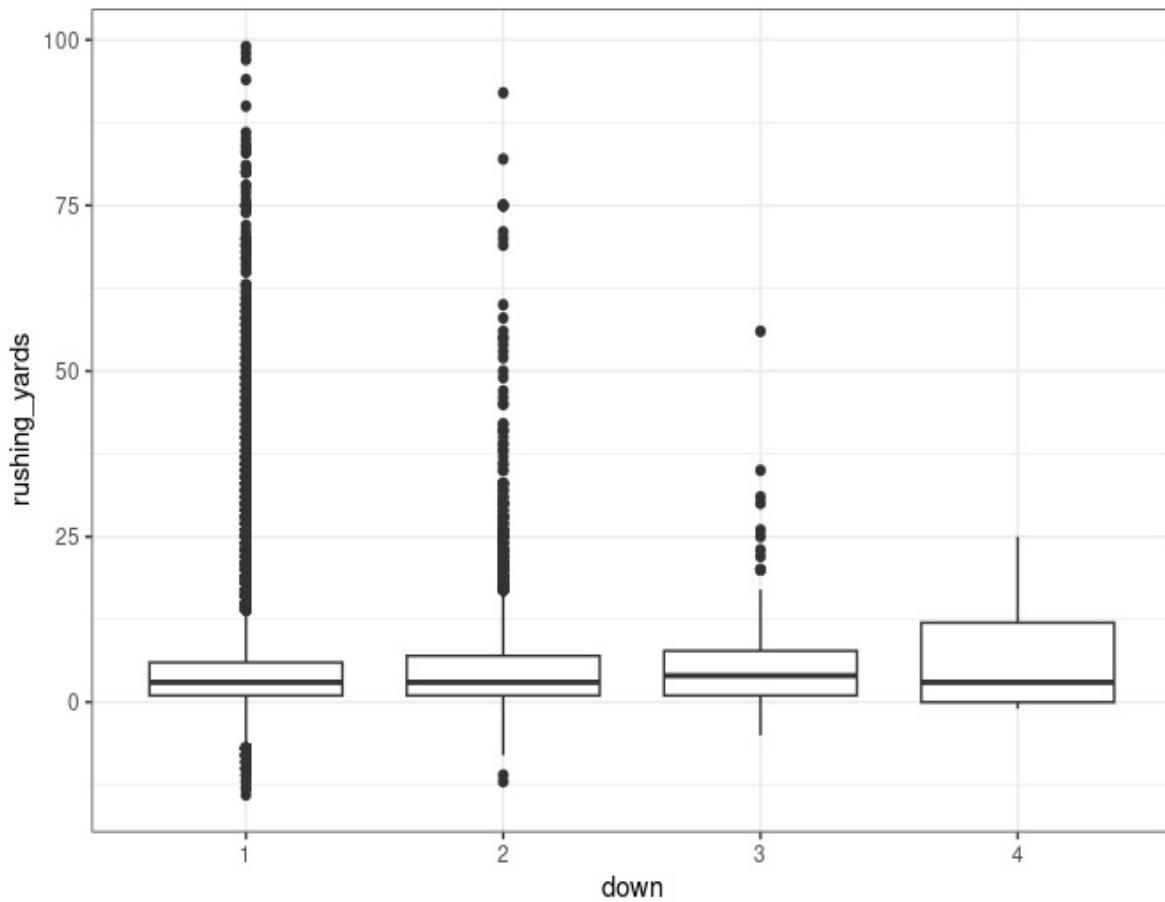


Figure 4-4. Boxplot of rushing yards by downs with `ggplot2` for plays with 10 yards to go.

Okay, now you see what you expect. This is an example of *Simpson's Paradox*, where relationships between variables change during different groupings of other variables. Nonetheless, it's clear that down affects the rushing yards on a play and should be accounted for. Similarly, let's look at yards to the endzone in `seaborn` with [Figure 4-5](#) (and change the transparency with `scatter_kws= {'alpha': 0.25}` and the regression line color with `line_kws= {'color': 'red'}`):

```
## Python
sns.regplot(
    data=pbp_py_run,
    x="yardline_100",
    y="rushing_yards",
    scatter_kws={"alpha": 0.25},
    line_kws={"color": "red"},
);
plt.show();
```

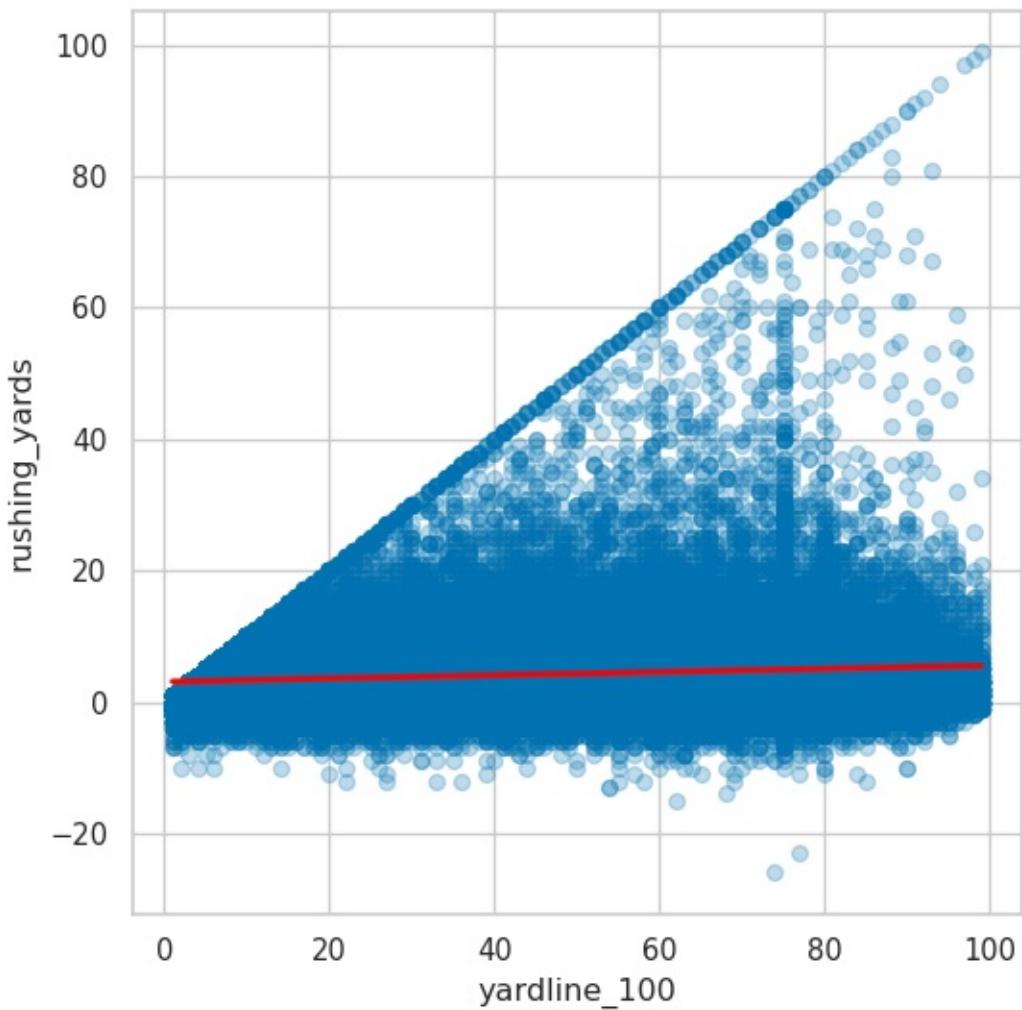


Figure 4-5. Scatterplot with linear trendline for ball position (yards to go to the endzone) and rushing yards from a play with seaborn.

Or with R create [Figure 4-6](#) (and change the transparency with `alpha = 0.25` to help you see the overlapping points):

```
## R
ggplot(pbp_r_run, aes(x = yardline_100, y = rushing_yards)) +
  geom_point(alpha = 0.25) +
  stat_smooth(method = "lm") +
  theme_bw()
```

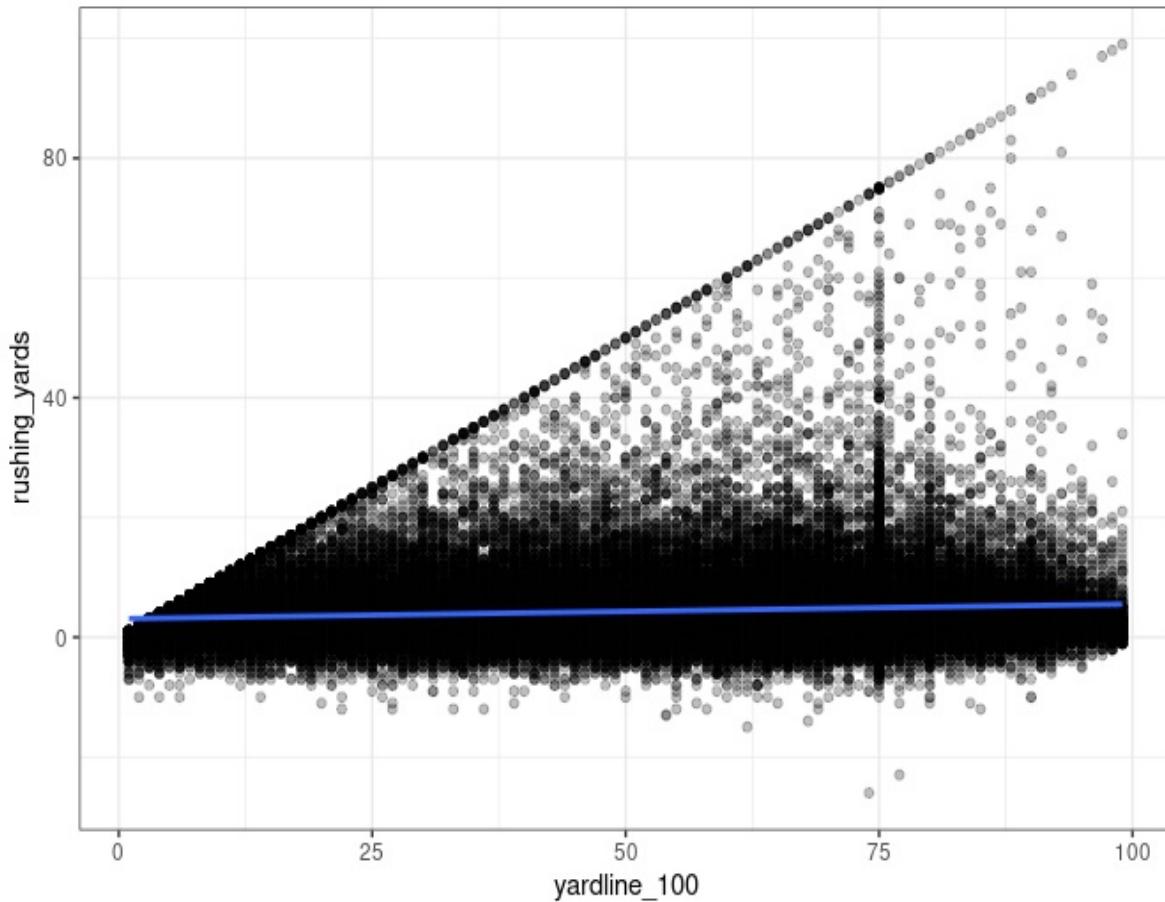


Figure 4-6. Scatterplot with linear trendline for ball position (yards to go to the endzone) and rushing yards from a play with ggplot2.

This doesn't appear to do much, but let's look what happens after you bin and average with Python:

```
## Python
pbp_py_run_y100 =\
    pbp_py_run\
    .groupby("yardline_100")\
    .agg({"rushing_yards": ["mean"]})

pbp_py_run_y100.columns =\
    list(map("_".join, pbp_py_run_y100.columns))

pbp_py_run_y100.reset_index(inplace=True)
```

So that you may create Figure 4-7:

```
sns.regplot(
```

```

data=pbp_py_run_y100,
x="yardline_100",
y="rushing_yards_mean",
scatter_kws={"alpha": 0.25},
line_kws={"color": "red"},

);
plt.show();

```

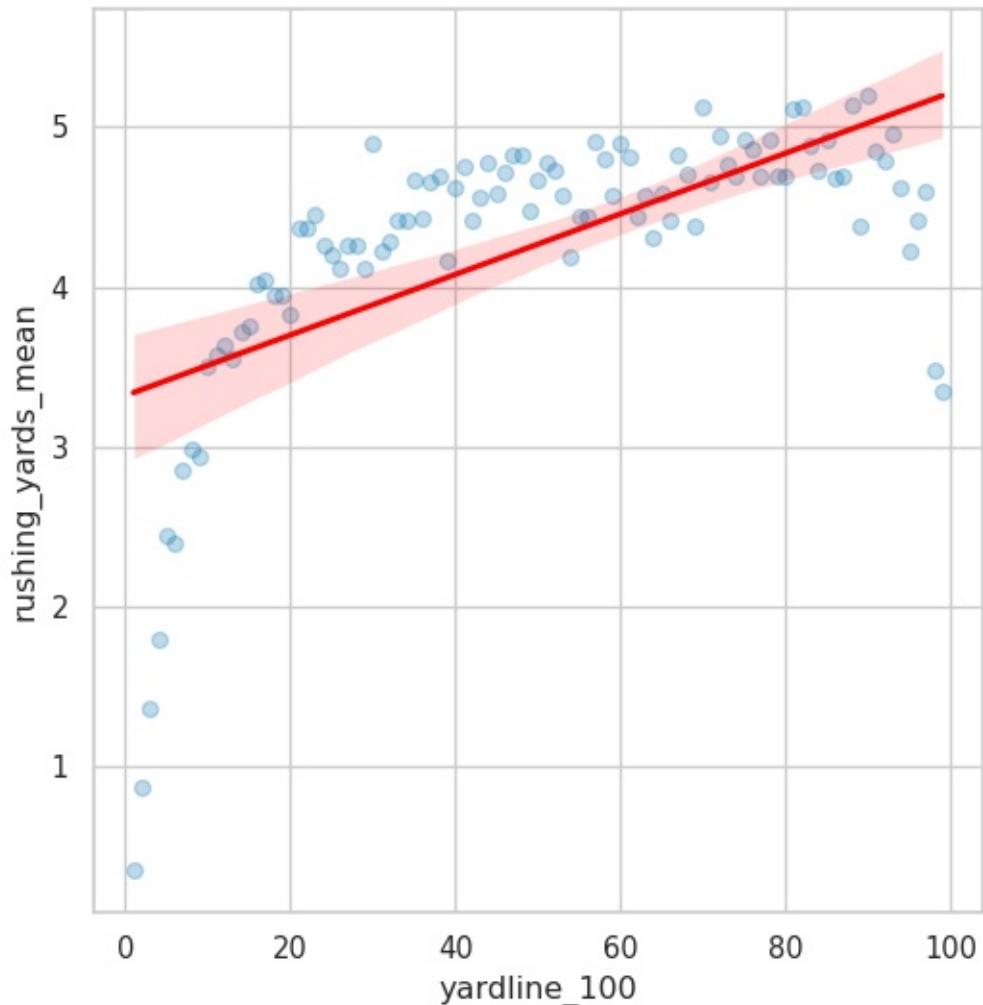


Figure 4-7. Scatterplot with linear trendline for ball position and rushing yards with *seaborn* for data binned by yard.

Or with R in create **Figure 4-8**:

```

## R
pbp_r_run |>
  group_by(yardline_100) |>

```

```

summarize(rushing_yards_mean = mean(rushing_yards)) |>
ggplot(aes(x = yardline_100, y = rushing_yards_mean)) +
geom_point() +
stat_smooth(method = "lm") +
theme_bw()

```

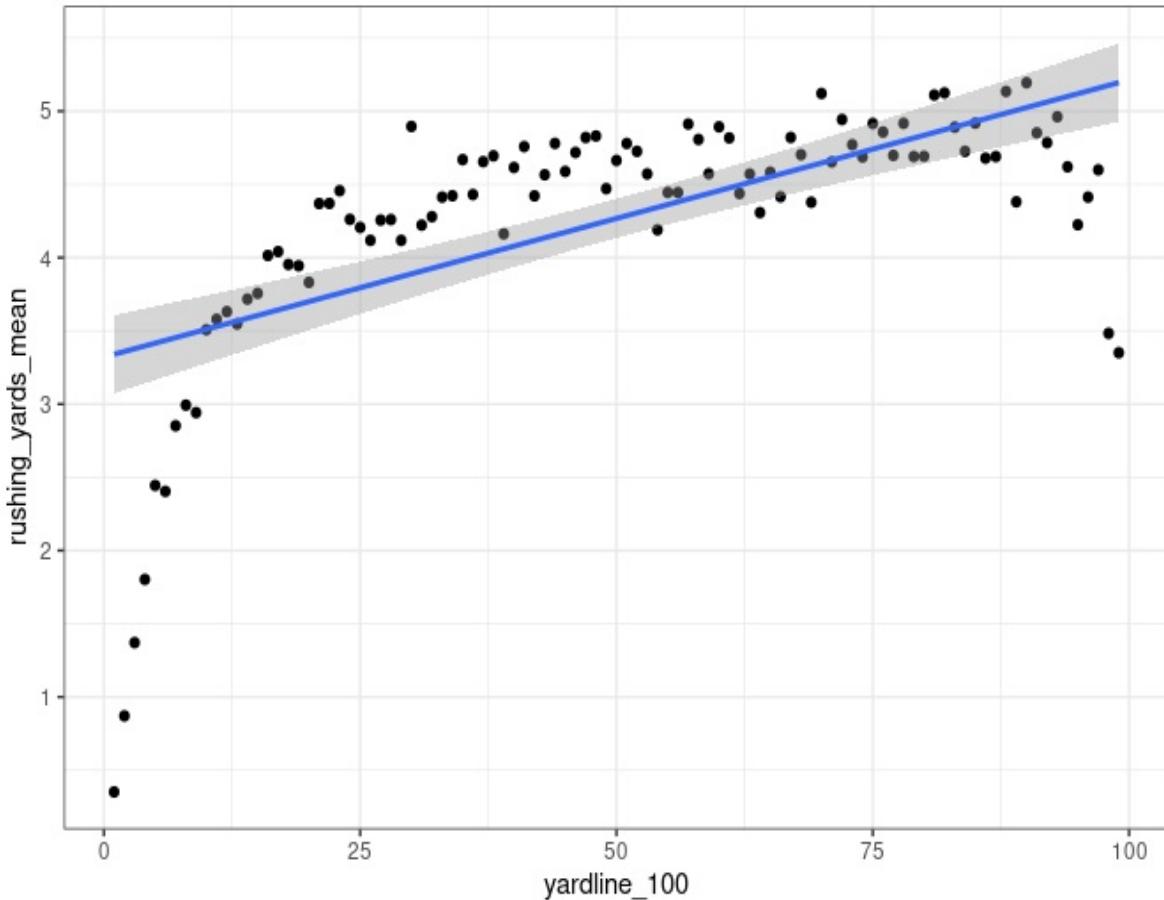


Figure 4-8. Scatterplot with linear trendline for ball position and rushing yards with `ggplot2` for data binned by yard.

Figure 4-7 and Figure 4-8 show some football insight. Running plays less than about 15 yards to go are limited by distance because there is limited distance to the endzone, and, tougher red zone defense. Likewise, plays with more than 90 yards to go are going out of your own end zone. So, defense will be trying hard to force a safety and offense will be more likely to either punt or play conservatively to avoid allowing a safety.

Here you get a clear positive (but non-linear) relationship between average rushing yards and yards to go to the endzone, so it benefits you to include this feature in the models. In “Assumption of Linearity”, you can see what happens

to the model if values less than 15 yards or greater than 90 yards are removed. In practice, more complicated models can effectively deal with these nonlinearities, but we save that for a different book. Now, you look at run location with Python in [Figure 4-9](#):

```
## Python
sns.boxplot(data=pbp_py_run,
             x="run_location",
             y="rushing_yards");
plt.show();
```

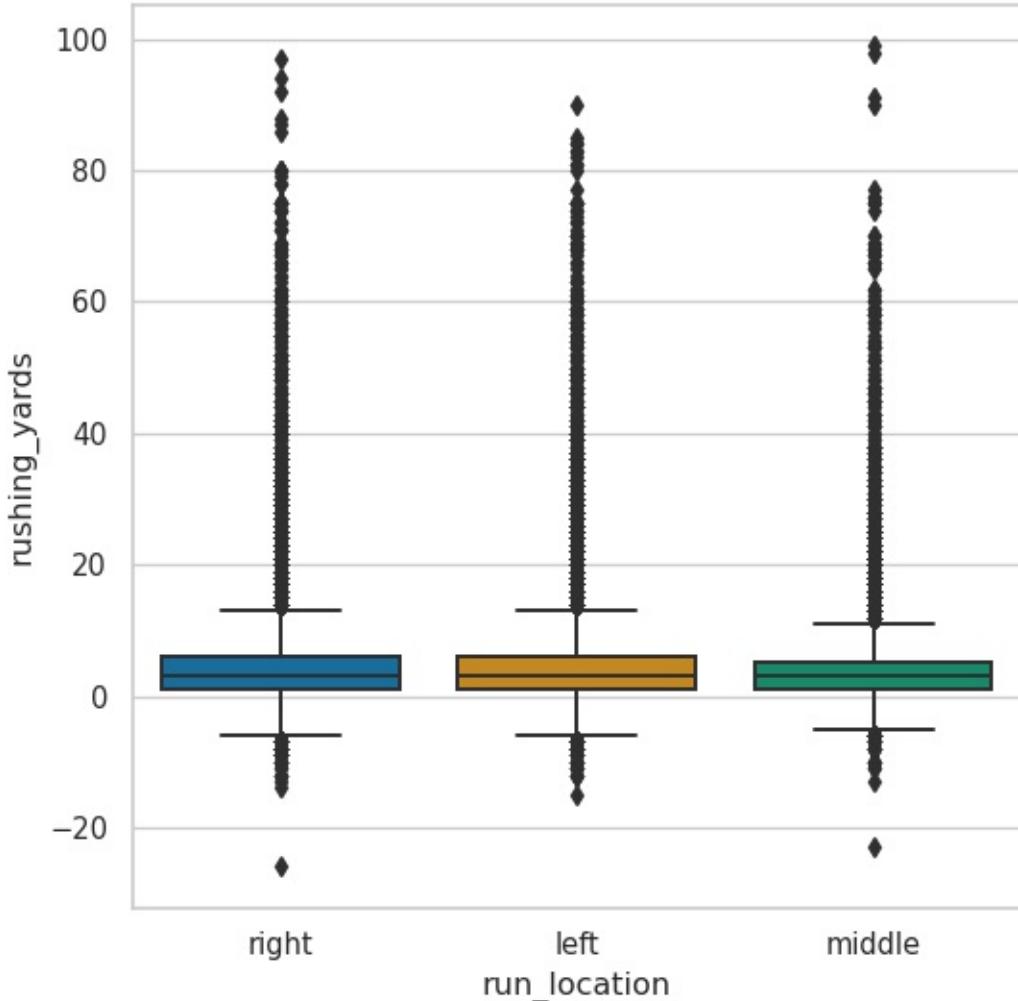


Figure 4-9. Boxplot of rushing yards by run location with seaborn.

Or with R in [Figure 4-10](#):

```
## R
ggplot(pbp_r_run, aes(run_location, rushing_yards)) +
  geom_boxplot() +
  theme_bw()
```

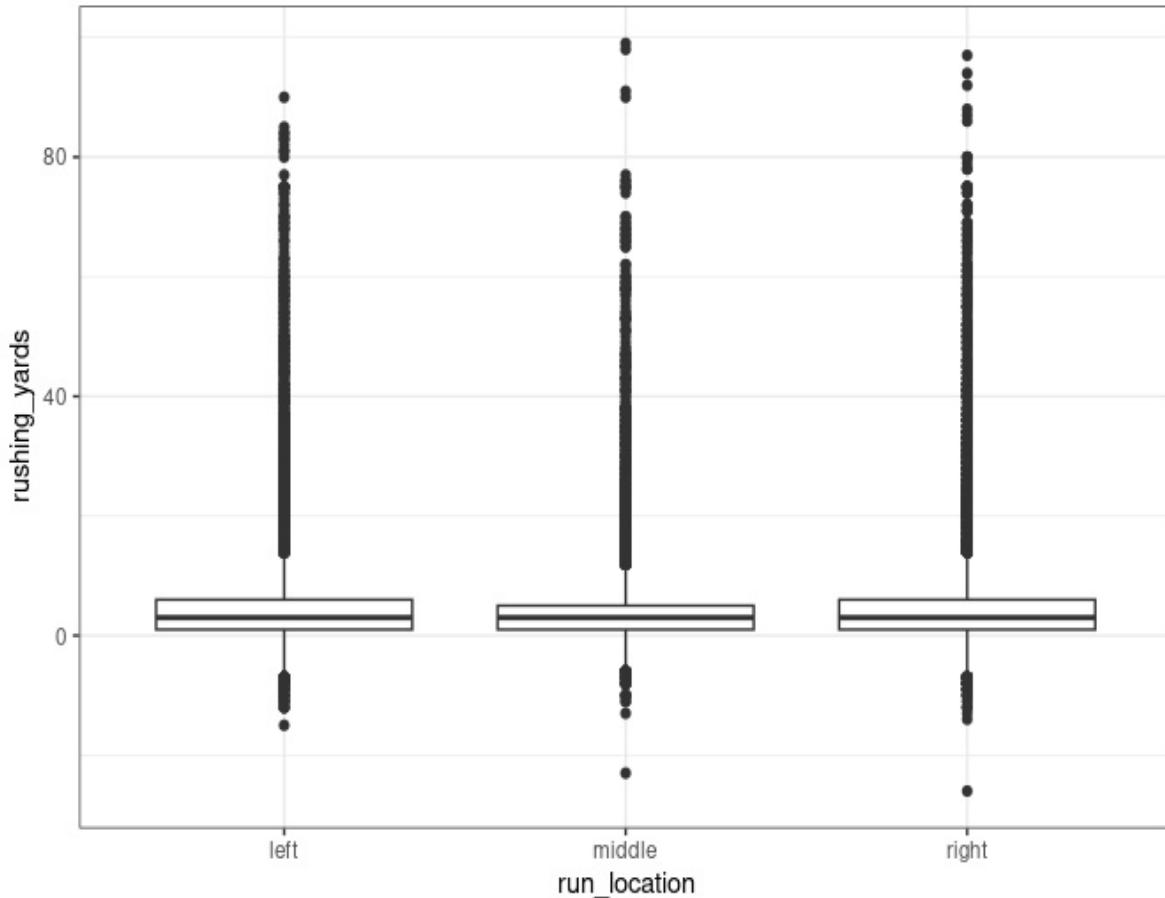


Figure 4-10. Boxplot of rushing yards by run location with *ggplot2*.

Not only are the means/medians slightly different here, the variances/interquartile ranges appear to vary, so keep them in the models. Another comment about the run location is that the 75th percentile in each case is really low. Three quarters of the time, regardless of whether you go left, right, or down the middle, you go 10 yards or less. It's only in a few rare cases that you get a long rush.

Lastly, look at score differential, using the binning and aggregating you used for yards to go to the endzone in Python

```
## Python
pbp_py_run_sd = \
```

```
pbp_py_run\  
.groupby("score_differential")\  
.agg({"rushing_yards": ["mean"]})  
)  
  
pbp_py_run_sd.columns =\  
list(map("_".join, pbp_py_run_sd.columns))  
  
pbp_py_run_sd.reset_index(inplace=True)
```

So that you can create [Figure 4-11](#):

```
## Python  
sns.regplot(  
    data=pbp_py_run_sd,  
    x="score_differential",  
    y="rushing_yards_mean",  
    scatter_kws={"alpha": 0.25},  
    line_kws={"color": "red"},  
);  
plt.show();
```

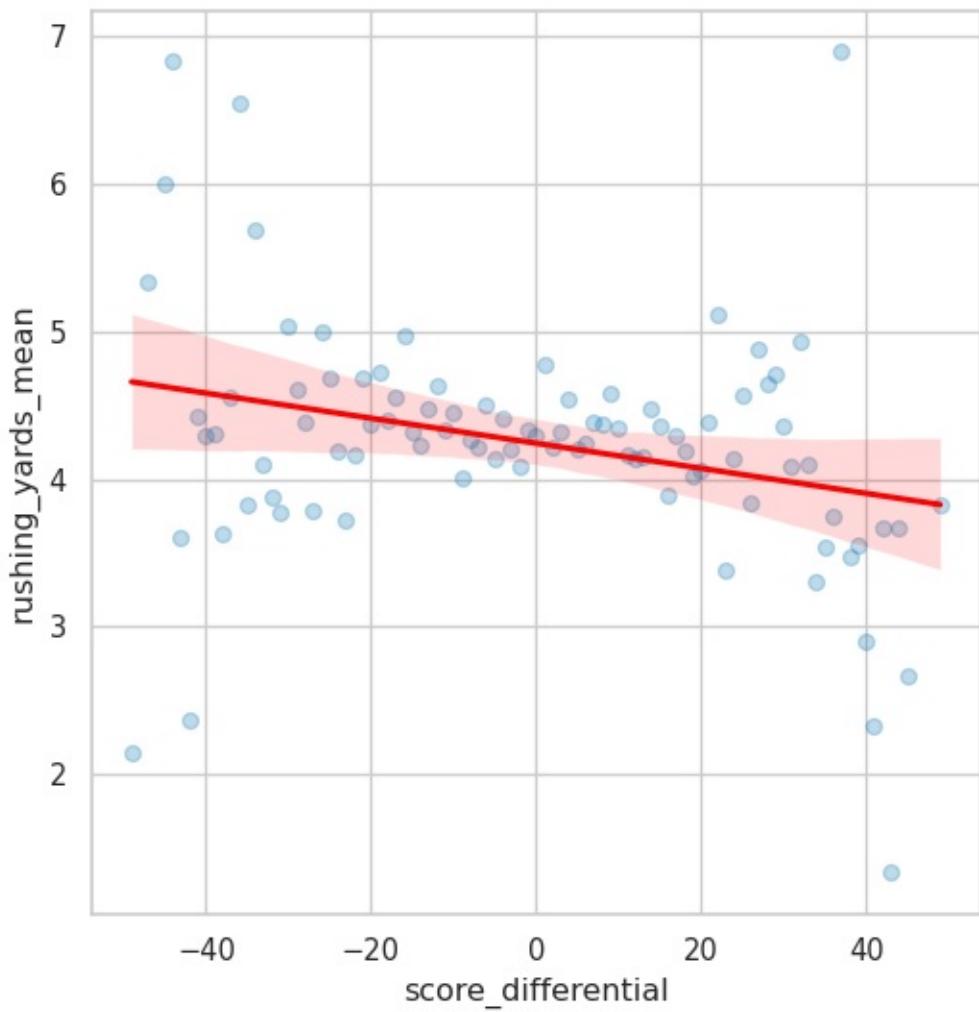


Figure 4-11. Scatterplot with linear trendline for score differential and rushing yards with seaborn for data binned by score differential.

Or, in R create **Figure 4-12** with:

```
## R
pbp_r_run |>
  group_by(score_differential) |>
  summarize(rushing_yards_mean = mean(rushing_yards)) |>
  ggplot(aes(score_differential, rushing_yards_mean)) +
  geom_point() +
  stat_smooth(method = "lm") +
  theme_bw()
```

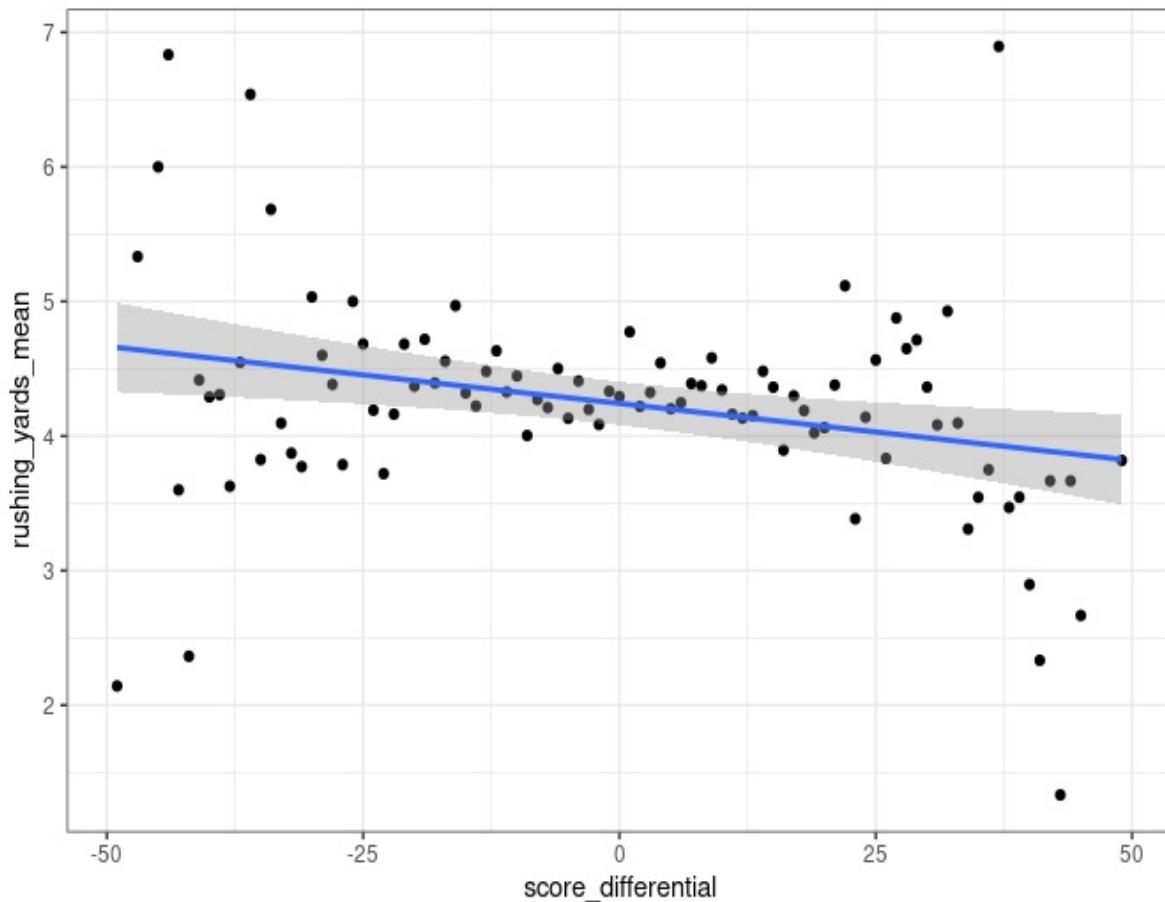


Figure 4-12. Scatterplot with linear trendline for score differential and rushing yards with ggplot2 for data binned by score differential.

You can see a clear negative relationship, as hypothesized above. Hence you will leave score differential in the model.

TIP

When you see code for plots in this book, think about how you would want to change them to make the plots better. Also, look at the code and if you do not understand some arguments, search and figure out how to change the plotting code. The best way to become a better data plotter is to explore and create your own plots.

Applying Multiple Linear Regression

Now apply the multiple linear regression to re-derive rushing yards over expected (RYOE). We gloss over steps that are covered in [Chapter 3](#). First, fit

the model. Then, save the calculated residuals as the RYOE. Recall that residuals are the difference between the value predicted by the model and observed in the data. This could be calculated directly by taking the observed rushing yards and subtracting the predicted rushing yards from the model. However, residuals are commonly used in statistics, and Python and R both include residuals as part of the model's fit. This derivation differs from the method in [Chapter 3](#) because you have created a more complicated model.

The model predicts `rushing_yards` by creating:

- An intercept (1).
- A term contrasting the second, third, and fourth downs to the first down (`down`).
- A coefficient for `ydstogo`.
- An *interaction* between `ydstogo` and `down` that estimates a `ydstogo` contrast for each `down` (`ydstogo : down`).
- A coefficient for yards to go to the endzone (`yardline_100`).
- The location of the run play on the field (`run_location`).
- The difference between each team's scores (`score_differential`).

NOTE

Multiple methods exist for writing interactions using formulas. The longest, but most straight forward approach with the example from "[Applying Multiple Linear Regression](#)" would be `down + ydstogo + as.factor(down):ydstogo`. This may be abbreviated as `down * ydstogo`. Thus, the example formula, `rushing_yards ~ 1 + down + ydstogo + down:ydstogo + yardline_100 + run_location + score_differential` could be written as `rushing_yards ~ down*ydstogo + yardline_100 + run_location + score_differential` and saves writing three terms.

In Python, fit the model with the `statsmodels` package, then save the residuals as RYOE:

```
## Python
```

```

pbp_py_run.down =\
    pbp_py_run.down.astype(str)

expected_yards_py =\
    smf.ols(
        data=pbp_py_run,
        formula="rushing_yards ~ 1 + down + ydstogo + " +
        "down:ydstogo + yardline_100 + " +
        "run_location + score_differential")\
        .fit()

pbp_py_run["ryoe"] =\
    expected_yards_py.resid

```

NOTE

We included line breaks in our code to make the code fit on the pages for this book. For example in Python, we used the string "rushing_yards ~ 1 + down + ydstogo + " followed by ` and then a line break. Then, each of the next two strings, ` "down:ydstogo + yardline_100 + "` and ` "run_location + score_differential"` get their own line breaks and we when then used the ` character to *add* each of the strings together in [“Applying Multiple Linear Regression”](#). We used line breaks with adding strings in Python because we had to make the code fit within the page width limits. These breaks are not required, but help to make the code look better to human eyes and ideally be more readable.

Likewise, fit the model in, and save the residuals as RYOE in R:

```

## R
pbp_r_run <-
  pbp_r_run |>
  mutate(down = as.character(down))

expected_yards_r <-
  lm(rushing_yards ~ 1 + down + ydstogo + down:ydstogo +
      yardline_100 + run_location + score_differential,
      data = pbp_r_run
  )

pbp_r_run <-
  pbp_r_run |>
  mutate(ryoe = resid(expected_yards_r))

```

Now, examine the summary of the model in Python:

```

## Python
print(expected_yards_py.summary())
               OLS Regression Results
=====
=====
Dep. Variable:      rushing_yards    R-squared:
0.016
Model:                  OLS    Adj. R-squared:
0.016
Method:                Least Squares    F-statistic:
136.6
Date:      Sat, 20 May 2023    Prob (F-statistic):
3.43e-313
Time:          15:42:31    Log-Likelihood:
-2.9764e+05
No. Observations:      91442    AIC:
5.953e+05
Df Residuals:          91430    BIC:
5.954e+05
Df Model:                   11
Covariance Type:        nonrobust
=====
=====
                           coef      std err      t      P>|t|
[0.025      0.975]
-----
-----
Intercept              1.6085      0.136     11.849      0.000
1.342      1.875
down[T.2.0]             1.6153      0.153     10.577      0.000
1.316      1.915
down[T.3.0]             1.2846      0.161      7.990      0.000
0.969      1.600
down[T.4.0]             0.2844      0.249      1.142      0.254
-0.204      0.773
run_location[T.middle] -0.5634      0.053     -10.718      0.000
-0.666      -0.460
run_location[T.right]  -0.0382      0.049      -0.784      0.433
-0.134      0.057
ydstogo                0.2024      0.014     14.439      0.000
0.175      0.230
down[T.2.0]:ydstogo   -0.1466      0.016      -8.957      0.000
-0.179      -0.115
down[T.3.0]:ydstogo   -0.0437      0.019      -2.323      0.020
-0.081      -0.007
down[T.4.0]:ydstogo   0.2302      0.090      2.567      0.010
0.054      0.406
yardline_100            0.0186      0.001     21.230      0.000
0.017      0.020
score_differential     -0.0040      0.002      -2.023      0.043

```

```

-0.008      -0.000
=====
=====
Omnibus:           80510.527   Durbin-Watson:
1.979
Prob(Omnibus):    0.000     Jarque-Bera (JB):
3941200.520
Skew:              4.082     Prob(JB):
0.00
Kurtosis:          34.109    Cond. No.
838.
=====
=====
```

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Or examine the summary of the model R:

```

## R
print(summary(expected_yards_r))
Call:
lm(formula = rushing_yards ~ 1 + down + ydstogo + down:ydstogo +
    yardline_100 + run_location + score_differential, data =
pbp_r_run)

Residuals:
    Min      1Q  Median      3Q      Max
-32.233 -3.130 -1.173  1.410  94.112

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.608471  0.135753 11.849 < 2e-16 ***
down2        1.615277  0.152721 10.577 < 2e-16 ***
down3        1.284560  0.160775  7.990 1.37e-15 ***
down4        0.284433  0.249106  1.142  0.2535
ydstogo      0.202377  0.014016 14.439 < 2e-16 ***
yardline_100 0.018576  0.000875 21.230 < 2e-16 ***
run_locationmiddle -0.563369  0.052565 -10.718 < 2e-16 ***
run_locationright -0.038176  0.048684 -0.784  0.4329
score_differential -0.004028  0.001991 -2.023  0.0431 *
down2:ydstogo    -0.146602  0.016367 -8.957 < 2e-16 ***
down3:ydstogo    -0.043703  0.018814 -2.323  0.0202 *
down4:ydstogo    0.230179  0.089682  2.567  0.0103 *
---
Signif. codes:  0 '****' 0.001 '***' 0.01 '**' 0.05 '*' 0.1 '.' 1
```

```
Residual standard error: 6.272 on 91430 degrees of freedom
Multiple R-squared:  0.01617,   Adjusted R-squared:  0.01605
F-statistic: 136.6 on 11 and 91430 DF,  p-value: < 2.2e-16
```

Each estimated coefficient helps you tell a story about rushing yards during plays:

- Running plays on the second down (`down[T.2.0]` in Python or `down2` in R) have more expected yards per carry than the first down, all else being equal, in this case about 1.6 yards.
- Running plays on the third down (`down[T.3.0]` in Python or `down3` in R) have more expected yards per carry than the first down, all else being equal, in this case about 1.3 yards.
- The interaction term tells you that this especially true when there are fewer yards to go for the first down (the interaction terms are all negative). From a football standpoint this just means that second and third and short are more favorable to the offense running the ball than first down and 10.
- Conversely, running plays on the fourth down have slightly more yards gained compared to first down (`down[T.4.0]` in Python or `down4` in R), all else being equal, but not as many as second or third down.
- As the number of yards-to-go increases (`ydstogo`), so does the rushing yards, all else being equal, with each yard to go worth about a fifth of a yard. This is because the `ydstogo` estimate is positive.
- As the ball is farther from the endzone, rushing plays produced slightly more yards per play (about 0.02 per yard to go to the endzone; `yardline_100`). For example, even if there are 100 yards-to-go, a coefficient of 0.02 means that it would only result in 2 extra yards rushed in the play, which doesn't have a big impact compared to other coefficients.
- Rushing plays in the middle of the field earn about a half of yard less than plays to the left based upon the contrast estimate between the middle and left side of the field (`run_location[T.middle]` in Python or `run_locationmiddle` in R).
- The negative `score_differential` coefficient differs statistically

from zero. Thus, when teams are ahead (have a positive score differential), they gain fewer yards per play on the average run play. However, this effect is small (0.004) and thus not important compared to other coefficients that it can be ignored (for example, being up by 50 points would only decrease the number of yards by 0.2 per carry).

Notice from the coefficients, that it's harder to run to the middle of the field than to the outside, all other things being equal. You indeed see that distance and yards to go to the both the first down marker and the endzone both positively affect rushing yards, the further you are away from your goals as an offense, the more defense will surrender to you on average.

TIP

The `kableExtra` package in R helps to produce well formatted tables in RMarkdown and Quarto documents as well on screen. You'll need to install the package, if you have not done so already.

Tables exist as another method to present regression coefficients. For example, the `broom` package allows tidy tables to be created in R that can then be formatted for using the `kableExtra` package such as [Table 4-1](#). Specifically, with this code, take the model fit `expected_yards_r` and then pipe the model to extract the “tidy” model outputs (including the 95% confidence intervals) using `tidy(conf.int = TRUE)`. Then, convert the table to a `kable` table and show two digits using `tbl(format = "pipe", digits = 2)`. Last, apply styling from the `kableExtra` package using `kable_styling()`:

```
## R
library(broom)
library(kableExtra)
expected_yards_r |>
  tidy(conf.int = TRUE) |>
  kbl(format = "pipe", digits = 2) |>
  kable_styling()
```

Table 4-1. Table 3.1: Example regression coefficient table. term is the regression coefficient, estimate is the estimated value for the coefficient, std.error is the standard error, statistic is the t-score, p.value is the p-

value, *conf.low* is the bottom of the 95% CI, and *conf.high* is the top of the 95% CI.

term	estimate	std.error	statistic	p.value
(Intercept)	1.61	0.14	11.85	0.00
down2	1.62	0.15	10.58	0.00
down3	1.28	0.16	7.99	0.00
down4	0.28	0.25	1.14	0.25
ydstogo	0.20	0.01	14.44	0.00
yardline_100	0.02	0.00	21.23	0.00
run_locationmiddle	-0.56	0.05	-10.72	0.00
run_locationright	-0.04	0.05	-0.78	0.43
score_differential	0.00	0.00	-2.02	0.04
down2:ydstogo	-0.15	0.02	-8.96	0.00
down3:ydstogo	-0.04	0.02	-2.32	0.02
down4:ydstogo	0.23	0.09	2.57	0.01

TIP

Writing about regressions can be hard, and knowing your audience and their background is key. For example, the paragraph `Notice from the coefficients...' would be appropriate for a casual blog, but not for a peer-reviewed sports journal. Likewise, a table like [Table 4-1](#) might be included in a journal articles, but is more likely to be included as part of a technical report or journal article's supplemental materials. Our walk-through of individual coefficients in a bulleted list might be appropriate for a report to a client who wants an item-by-item description or teaching venue such as blog or book on football analytics.

Analyzing RYOE

Just like with the first version of RYOE in [Chapter 3](#), now you will analyze the new version of your metric for rushers. First, create the summary tables for the RYOE totals, means, and yards-per-carry in Python. Next, save only data from players with more than 50 carries. Also, rename the columns and sort by total RYOE:

```
## Python
ryoe_py =\
    pbp_py_run\
    .groupby(["season", "rusher_id", "rusher"])\\
    .agg({
        "ryoe": ["count", "sum", "mean"],
        "rushing_yards": ["mean"]})

ryoe_py.columns =\
    list(map("_".join, ryoe_py.columns))
ryoe_py.reset_index(inplace=True)

ryoe_py =\
    ryoe_py\
    .rename(columns={
        "ryoe_count": "n",
        "ryoe_sum": "ryoe_total",
        "ryoe_mean": "ryoe_per",
        "rushing_yards_mean": "yards_per_carry",
    })\
    .query("n > 50")

print(ryoe_py\
    .sort_values("ryoe_total", ascending=False)
```

```

)
   season    rusher_id      rusher ...  ryoе_total  ryoе_per
yards_per_carry
1870    2021  00-0036223    J.Taylor ...  471.232840  1.419376
5.454819
1350    2020  00-0032764    D.Henry ...  345.948778  0.875820
5.232911
1183    2019  00-0034796    L.Jackson ...  328.524757  2.607339
6.880952
1069    2019  00-0032764    D.Henry ...  311.641243  0.807361
5.145078
1383    2020  00-0033293    A.Jones ...  301.778866  1.365515
5.565611
...
...
627     2018  00-0027029    L.McCoy ... -208.392834 -1.294365
3.192547
51      2016  00-0027155    R.Jennings ... -228.084591 -1.226261
3.344086
629     2018  00-0027325    L.Blount ... -235.865233 -1.531592
2.714286
991     2019  00-0030496    L.Bell ... -338.432836 -1.381359
3.220408
246     2016  00-0032241    T.Gurley ... -344.314622 -1.238542
3.183453

[533 rows x 7 columns]

```

Next, sort by mean RYOЕ per carry in Python:

```

## Python
print(
    ryoе_py\
        .sort_values("ryoе_per", ascending=False)
)
   season    rusher_id      rusher ...  ryoе_total  ryoе_per
yards_per_carry
2103    2022  00-0034796    L.Jackson ...  280.752317  3.899338
7.930556
1183    2019  00-0034796    L.Jackson ...  328.524757  2.607339
6.880952
1210    2019  00-0035228    K.Murray ...  137.636412  2.596913
6.867925
2239    2022  00-0036945    J.Fields ...  177.409631  2.304021
6.506494
1467    2020  00-0034796    L.Jackson ...  258.059489  2.186945
6.415254
...
...
```

```

...
1901    2021 00-0036414      C.Akers ... -129.834294 -1.803254
2.430556
533    2017 00-0032940 D.Washington ... -105.377929 -1.848736
2.684211
1858    2021 00-0035860      T.Jones ... -100.987077 -1.870131
2.629630
60     2016 00-0027791 J.Starks ... -129.298259 -2.052353
2.301587
1184    2019 00-0034799 K.Ballage ... -191.983153 -2.594367
1.824324

```

[533 rows x 7 columns]

These same tables may be created in R as well:

```

## R
ryoe_r <-
  pbp_r_run |>
  group_by(season, rusher_id, rusher) |>
  summarize(
    n = n(), ryoe_total = sum(ryoe), ryoe_per = mean(ryoe),
    yards_per_carry = mean(rushing_yards)
  ) |>
  filter(n > 50)

ryoe_r |>
  arrange(-ryoe_total) |>
  print()
# A tibble: 533 × 7
# Groups:   season, rusher_id [533]
  season rusher_id  rusher          n ryoe_total ryoe_per
  <dbl> <chr>       <chr>     <int>      <dbl>      <dbl>
<dbl>
  1 2021 00-0036223 J.Taylor     332      471.      1.42
5.45
  2 2020 00-0032764 D.Henry     395      346.      0.876
5.23
  3 2019 00-0034796 L.Jackson   126      329.      2.61
6.88
  4 2019 00-0032764 D.Henry     386      312.      0.807
5.15
  5 2020 00-0033293 A.Jones     221      302.      1.37
5.57
  6 2022 00-0034796 L.Jackson   72       281.      3.90
7.93
  7 2019 00-0031687 R.Mostert    190      274.      1.44

```

```

5.83
8 2016 00-0033045 E.Elliott    342      274.    0.800
5.14
9 2020 00-0034796 L.Jackson   118      258.    2.19
6.42
10 2021 00-0034791 N.Chubb    228      248.    1.09
5.52
# i 523 more rows

```

Then, sort by mean RYOE per carry in R:

```

## R
ryoe_r |>
  filter(n > 50) |>
  arrange(-ryoe_per) |>
  print()
# A tibble: 533 × 7
# Groups:   season, rusher_id [533]
  season rusher_id  rusher       n ryoe_total ryoe_per
  <dbl> <chr>        <chr>     <int>      <dbl>      <dbl>
<dbl>
  1 2022 00-0034796 L.Jackson    72      281.      3.90
7.93
  2 2019 00-0034796 L.Jackson   126      329.      2.61
6.88
  3 2019 00-0035228 K.Murray    53      138.      2.60
6.87
  4 2022 00-0036945 J.Fields    77      177.      2.30
6.51
  5 2020 00-0034796 L.Jackson   118      258.      2.19
6.42
  6 2017 00-0027939 C.Newton    92      191.      2.08
6.17
  7 2020 00-0035228 K.Murray    70      144.      2.06
6.06
  8 2021 00-0034750 R.Penny     119      242.      2.03
6.29
  9 2019 00-0034400 J.Wilkins   51      97.8      1.92
6.02
 10 2022 00-0033357 T.Hill      95      171.      1.80
6.05
# i 523 more rows

```

The above outputs are similar to those from [Chapter 3](#), but come from a model that “corrects for” additional features when estimating RYOE.

When it comes to total RYOE, Jonathan Taylor's 2021 season still reigns supreme, but we see the emergence of players like Raheem Mostert, who helped the 49ers to a Super Bowl in 2019, and Nick Chubb, who has been one of the league's best running backs since entering the league in 2018. Le'Veon Bell of the Steelers makes an appearance: we'll talk about him later.

As for RYOE per carry (for players with 50 or more carries in a season) Rashaad Penny's brilliant 2021 shines again, but his injury-shortened 2022 campaign emerges as well. The only quarterback in the group is Daniel Jones, who guided the previously dormant New York Giants to the playoffs in his fourth year in the NFL. It's instructive that after accounting for more variables, the quarterbacks like Lamar Jackson and others appear less impressive than before. The Bills' Mike Gillislee was an interesting case in 2016, as rival New England signed him as a restricted free agent the following year, and he didn't do much for the rest of his career thereafter.

As for the stability of this metric relative to yards per carry, recall that in the previous chapter we got a slight bump in stability though not necessarily enough to say that RYOE per carry is definitively a superior metric to yards per carry for rushers. Let's re-do this analysis:

```
## Python
# only keep columns needed
cols_keep = \
    ["season", "rusher_id", "rusher",
     "ryoe_per", "yards_per_carry"]

# create current dataframe
ryoe_now_py = \
    ryoe_py[cols_keep].copy()

# create last-year's dataframe
ryoe_last_py = \
    ryoe_py[cols_keep].copy()

# rename columns
ryoe_last_py\
    .rename(columns = {'ryoe_per': 'ryoe_per_last',
                      'yards_per_carry': 'yards_per_carry_last'},
            inplace=True)

# add 1 to season
ryoe_last_py["season"] += 1
```

```
# merge together
ryoe_lag_py =\
    ryoe_now_py\
    .merge(ryoe_last_py,
           how='inner',
           on=['rusher_id', 'rusher',
                'season'])
```

Then, examine the correlation for yards per carry:

```
## Python
ryoe_lag_py[["yards_per_carry_last", "yards_per_carry"]]\n
    .corr()
            yards_per_carry_last  yards_per_carry
yards_per_carry_last          1.000000      0.347267
yards_per_carry                  0.347267      1.000000
```

Repeat with RYOE:

```
## Python
ryoe_lag_py[["ryoe_per_last", "ryoe_per"]]\n
    .corr()
            ryoe_per_last  ryoe_per
ryoe_per_last          1.000000  0.373582
ryoe_per                  0.373582  1.000000
```

These calculations may also be done using R:

```
## R
# create current dataframe
ryoe_now_r <-
    ryoe_r |>
    select(-n, -ryoe_total)

# create last-year's dataframe
# and add 1 to season
ryoe_last_r <-
    ryoe_r |>
    select(-n, -ryoe_total) |>
    mutate(season = season + 1) |>
    rename(ryoe_per_last = ryoe_per,
           yards_per_carry_last = yards_per_carry)

# merge together
ryoe_lag_r <-
```

```

ryoe_now_r |>
inner_join(ryoe_last_r,
            by = c("rusher_id", "rusher", "season")) |>
ungroup()

```

Then, select the two yards-per-carry columns and examine the correlation:

```

## R
ryoe_lag_r |>
  select(yards_per_carry, yards_per_carry_last) |>
  cor(use = "complete.obs")
      yards_per_carry yards_per_carry_last
yards_per_carry           1.000000        0.347267
yards_per_carry_last       0.347267        1.000000

```

Repeat with the RYOE columns:

```

## R
ryoe_lag_r |>
  select(ryoe_per, ryoe_per_last) |>
  cor(use = "complete.obs")
      ryoe_per ryoe_per_last
ryoe_per      1.0000000    0.3735821
ryoe_per_last 0.3735821    1.0000000

```

This is an interesting result! The year-to-year stability for RYOE slightly improved with the new model, meaning after stripping out more and more context from the situation, we are indeed extracting some additional signal in running back ability above expectation. The issue is that this is still a minimal improvement, with the difference in r values less than 0.03. Additional work should interrogate the problem further, using better data (like tracking data) and better models (like tree-based models).

TIP

You have hopefully noticed the code in this chapter and [Chapter 3](#) are repetitive and similar. If we were repeating code on a regular basis like this, we would write our own set of functions and place the functions in a package to allow us to easily re-use our code. “[Packages](#)” provides an overview of this topic.

So, Do Running Backs Matter?

This question seems silly on its face. Of course running backs, the players that carry the ball 50-to-250 times a year, matter. Jim Brown, Franco Harris, Barry Sanders, Marshall Faulk, Adrian Peterson, Derrick Henry, Jim Taylor; the history of the NFL cannot be written without mentioning the feats of great runners.

TIP

Ben Baldwin hosts a ‘nerd-to-human translator’, which helps to describe our word choices in this chapter. He notes *Ŵt he ≠ rdssay: ru ∩ ∈ gbacksdon’tmaer. ’Andthen* What the nerds mean: the results of run plays are primarily determined by run blocking and defenders in the box, not who is carrying the ball. Running backs are interchangeable, and investing a lot of resources (in the draft or free agency) doesn’t make sense.’ And then provides supporting evidence. Check out his page for this evidence more useful tips.

However, there are a couple of things to note. First, passing the football has gotten increasingly easier over the course of the past few decades. An [article](#) by Kevin Clark notes how “scheme changes, rule changes, and athlete changes” all help quarterbacks pass for more yards. We’ll add technology (e.g. the gloves the players wear), along with the adoption of college offenses and the creative ways in which they pass the football, to Kevin’s list.

In other words, the notion that “when you pass, only three things can happen (a completion, an incompletion, or an interception) and two of them are bad” doesn’t take into account the increasing rate at which the first thing (a completion) happens. Passing has long been more efficient than running (see “[Exercises](#)”), but now the variance in passing the ball has shrunk enough to prefer it in most cases to the low variance (and low efficiency) of running the ball.

In addition to the fact that running the ball is less efficient than passing the ball, whether measured through yards per play or something like EPA, the player running the ball evidently has less influence over the outcome of a running play than previously thought.

There are other factors that we couldn’t account for using `nflfastR` data that

also work against running backs influencing their production. Eric, [while he was at PFF](#), showed that offensive line play carried a huge signal when it came to rushing plays, using PFF's player grades by offensive linemen on a given play.

Eric also [showed during the 2021 season](#) that the concept of a perfectly-blocked run - a run play where no run blocker made a negative play (such as he got beaten by a defender) changed the outcome of a running play by roughly *half of an expected point*. While the running back can surely aid in the creation of a perfectly-blocked play through setting up his blockers and the like, it's unlikely the magnitude of an individual running back's influence is anywhere close to this value.

The NFL's generally caught up to this phenomenon, spending less and less as a percentage of the salary cap on running backs for the past decade plus, as this [article](#) from fivethirtyeight website shows. Be that as it may, there is still some contention about the position, with some players who are said to "break the mold" getting big contracts, usually to the disappointment of their teams.

An example of a situation where a team paid their star running back, and likely regretted it, was when the Dallas Cowboys in the fall of 2019, signed Ezekiel Elliott to a six-year, \$90 million contract, with \$50 million in guarantees. Elliott was holding out for a new deal, no doubt giving Cowboys owner Jerry Jones flashbacks to 1993, when the NFL's all-time leading rusher Emmitt Smith held out for the first two games for the defending champion Cowboys. The Cowboys, after losing both of the games without Smith, caved to the star's demands, and he rewarded them by earning the NFL's MVP award during the regular season. He finished the season off by earning the Super Bowl MVP as well, helping lead Dallas to the second of their three championships in the mid-90s.

Elliott had had a similar start to his career as Smith, leading the NFL in rushing yards per game in each of his first three seasons leading up to his holdout, and leading the league in total rushing yards during both seasons where he played the full year (Elliott was suspended for parts of the 2017 season). The Cowboys won their division in 2016 and 2018, and won a playoff game, only their second since 1996, in 2018.

As predicted by many in the analytics community, Zeke struggled to live up to the deal, with his rushing yards per game falling from 84.8 in 2019, to 65.3 in 2020, and flattening out to 58.9 and 58.4 in 2021 and 2022, respectively. Elliott's

yards per carry in 2022 fell beneath 4.0 to 3.8, and he eventually lost his starting job - and job altogether - to his backup Tony Pollard, in the 2023 offseason.

Not only did Elliott fail to live up to his deal, but his contract was onerous enough that the Cowboys had to jettison productive players like wide receiver Amari Cooper to get under the salary cap, a cascading effect that weakened the team more than simply the negative surplus play of a running back could.

An example where the team held their ground, and likely breathed a sigh of relief, occurred in 2018. Le'Veon Bell, who showed up above in "[Analyzing RYOE](#)", held out of training camp and the regular season for the Pittsburgh Steelers in a contract dispute, refusing to play on the *franchise tag* - an instrument a team uses to keep a player around for one season at the average of the top five salaries at their respective position. Players, who usually want longer-term deals, will often balk at playing under this contract type, and Bell - who gained over 5,000 yards on the ground during his first four years with the club, and led the NFL in touches in 2017 (carries and receptions) - was one such player.

The problem for Bell was that he was easily replaced in Pittsburgh. James Conner, a cancer survivor from the University of Pittsburgh selected in the third round of the 2017 draft, rushed for over 950 yards, at 4.5 yards per carry (Bell's career average was 4.3 as a Steeler), with 13 total touchdowns - earning a Pro Bowl berth.

Bell was allowed to leave as a free agent the following year, joining the lowly Jets, where he lasted 17 games and averaged just 3.3 yards per carry and scored only four times before being released during the middle of his second season. He did land with Kansas City in 2020 - a team that made the Super Bowl. He didn't play in that game, though, and the Chiefs lost 31-9 to Tampa Bay. He was out of football after the conclusion of the 2021 season.

Zeke and Bell's stories are maybe the most dramatic falls from grace for a starting running back, but are not the only ones, which is why it's important to be able to correctly attribute production to both the player and the scheme/rest of the roster in proper proportions, so as not to make a poor investment salary-wise.

Assumption of Linearity

Figure 4-7 and **Figure 4-8** clearly show a non-linear line. Technically, linear regression assumes both the residuals are normally distributed and that the observed relationship is linear. However, the two usually go hand in hand.

Base R contains useful diagnostic tools for looking at the results of linear models. The tools use base R's `plot()` function (and this is one of the few times we like `plot()`, it creates a simple function that we only use for internal diagnosis, not to share with others). First, use `par(mfrow=c(2, 2))` to create 4 sub-plots. Then, `plot()` the multiple regression you previously fit and saved as `expected_yards_r` to create **Figure 4-13**:

```
## R
par(mfrow = c(2, 2))
plot(expected_yards_r)
```

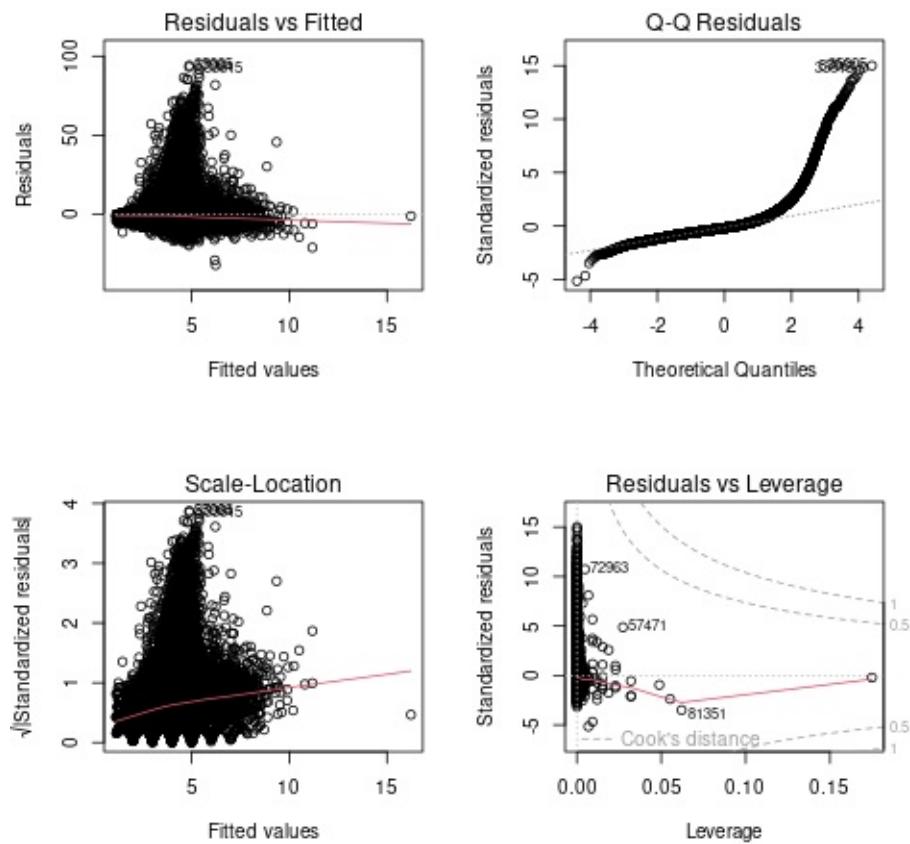


Figure 4-13. Four diagnostic plots for a regression model. The top left shows the model's estimated (or 'fitted') values compared to the difference in predicted versus fitted (or 'residual'). The top right shows cumulative distribution of the model's fits against the theoretical values. The bottom left shows the fitted values versus the square root of the standardized residuals. The bottom right shows the influence of a

parameters on the model's fit against the standardized residual.

Figure 4-13 contains four sub-plots. The “Residuals vs Fitted” sub-plot does not look *too* pathological. This figure simply shows many data points do not fit the model well and that a skew exists with the data. The “Normal Q-Q” sub-plot shows that many data points start to diverge from the expected model. Thus, our model does not fit the data well in some cases. The “Scale-Location” sub-plot shows similar patterns as the “Residuals vs Fitted”. Also, this plot has a weird pattern with “w”-like lines in near zero due to the integer (e.g., 0, 1, 2, 3) nature of the data. Lastly, “Residuals vs Leverage” shows that some data observations have a great deal of “leverage” on the model’s estimates, but these fall within the expected range based upon their Cook’s distance.

But, what happens if you remove plays less than 15 yards or greater than 90 to create Figure 4-14:

```
## R
expected_yards_filter_r <-
  pbp_r_run |>
  filter(rushing_yards > 15 & rushing_yards < 90) |>
  lm(
    formula = rushing_yards ~ 1 + down + ydstogo + down:ydstogo +
    yardline_100 + run_location + score_differential
  )

par(mfrow = c(2, 2))
plot(expected_yards_filter_r)
```

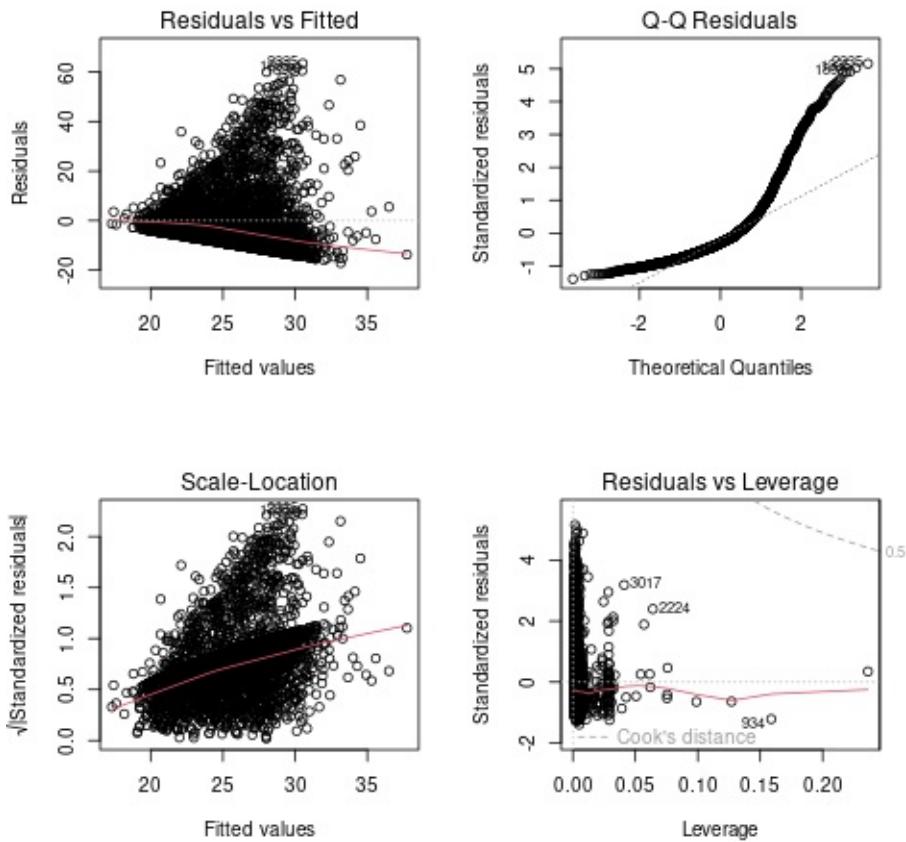


Figure 4-14. Figure 4-13 recreated with only rushing plays with more than 15 yards and less than 95 yards.

Figure 4-14 shows the new linear model, `expected_yards_filter_r`, fits better. Although the “Residuals vs Fitted” sub-plot has a wonky straight line (reflecting that the data has now been censored), the other sub-plots look better. The most improved sub-plot is the “Normal Q-Q” plot. Figure 4-13 had a scale from -5 to 15 whereas the plot now has a scale from -1 to 5. As one last check, look at the summary of the model and notice the improved model fit. The R^2 value improved from ~0.01 to 0.05:

```
## R
summary(expected_yards_filter_r)
Call:
lm(formula = rushing_yards ~ 1 + down + ydstogo + down:ydstogo +
    yardline_100 + run_location + score_differential, data =
filter(pbp_r_run,
       rushing_yards > 15 & rushing_yards < 95))
```

Residuals:					
Min	1Q	Median	3Q	Max	

```

-17.158 -7.795 -3.766 3.111 63.471

Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) 21.950963 2.157834 10.173 <2e-16 ***
down2 -2.853904 2.214676 -1.289 0.1976
down3 -0.696781 2.248905 -0.310 0.7567
down4 0.418564 3.195993 0.131 0.8958
ydstogo -0.420525 0.204504 -2.056 0.0398 *
yardline_100 0.130255 0.009975 13.058 <2e-16 ***
run_locationmiddle 0.680770 0.562407 1.210 0.2262
run_locationright 0.635015 0.443208 1.433 0.1520
score_differential 0.048017 0.019098 2.514 0.0120 *
down2:ydstogo 0.207071 0.224956 0.920 0.3574
down3:ydstogo 0.165576 0.234271 0.707 0.4798
down4:ydstogo 0.860361 0.602634 1.428 0.1535
---
Signif. codes: 0 '****' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 12.32 on 3781 degrees of freedom
Multiple R-squared: 0.05074, Adjusted R-squared: 0.04798
F-statistic: 18.37 on 11 and 3781 DF, p-value: < 2.2e-16

```

In summary, looking at the model's residuals helped you to see the model does not do well for plays less than 15 yards or longer than 95 yards. Knowing and quantifying this limitation at least helps you to know what you do not know with your model.

Data Science Tools Used in This Chapter

This chapter including the following topics:

- You saw how to fit a multiple regression in Python using `OLS()` or in R using `lm()`.
- You learned how to understand and read the coefficients from a multiple regression.
- You learned how to fit mixed-effect models in Python and R.
- You saw that sometimes simple models do better than complex models.
- You reapplied data wrangling tools you learned in previous chapters.

- You learned how to examine a regression's fit.

Exercises

1. Change the carries threshold from 50 carries to 100 carries. Do we still see the stability differences that we found in this chapter?
2. Use the full `nflfastR` dataset to show that rushing is less efficient than passing, both using yards per play and expected points added per play. Also inspect the variability of these two play types.
3. Is rushing more valuable than passing in some situations (e.g. near the opposing team's end zone?).
4. Inspect James Conner's RYOE values for his career relative to Bell's. What do you notice about the metric for both backs?
5. Repeat the processes within this chapter with receivers in the passing game. To do this, you have to filter by `play_type == "pass"` and `receiver_id` not being `NA` or `null`. Finding features will be difficult, but consider the process in this chapter for guidance. For example, use `down` and `distance`, but maybe also use something like `air_yards` in your model to try to set an expectation.

Suggested Readings

The books listed in “Suggested Readings” also apply for this chapter. Building upon this list, here are some other resources you may find helpful:

- *The Chicago Guide to Writing about Numbers* (Chicago Press, 2015 Second Edition) by Jane Miller provides great examples for describing number in different forms of writing.
- *The Chicago Guide to Writing about Multivariate Analysis* (Chicago Press, 2013 second edition) also by Jane Miller provides many different examples describing multiple regression. Although we disagree with her use of “multivariate regression” as a synonym for “multiple regression”, the book

does a great job proving examples of describing regression outputs.

- *Practical Linear Algebra for Data Science* (O'Reilly Media, 2022) by Mike X Cohen. Linear algebra forms the foundation almost all statistical methods including multiple regression. An understanding of linear algebra will help you better understand regression.
- **FiveThirtyEight** contains a great deal of data journalism and was started and is still run by Nate Silver. Looks through their posts and try to tell where they use regression models for posts.
- **Statistical Modeling, Causal Inference, and Social Science** by Andrew Gelman is a blog that often discusses regression modeling. Gelman is a more academic political scientist version of Nate Silver.
- **Statistical Thinking** by Frank Harrell is a blog that also commonly discusses regression analysis. Harrell is a more academic statistician version of Nate Silver and Harrell usually focuses more on statistics. However, many of his posts are often relevant to people doing any type of regression analysis.

Chapter 5. Generalized Linear Models: Completion Percentage Over Expected

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

In [Chapter 3](#) and [Chapter 4](#), you used both simple and multiple regression to adjust play-by-play data for the *context* of the play. In the case of ball carriers, you adjusted for the situation (such as down, distance, yards to go) to calibrate individual player statistics on the play level, and later the season level. This approach clearly can be applied to the passing game, and more specifically, quarterbacks. As discussed in [Chapter 3](#), Minnesota quarterback Sam Bradford set the NFL record for seasonal completion percentage during the 2016 season, completing a whopping 71.6% of his passes.

Bradford, however, was just a middle-of-the-pack quarterback in terms of efficiency - whether measured by yards per pass attempt, expected points per passing attempt, touchdown passes. The Vikings only won seven of his 15 starts that year. The reason Bradford’s completion percentage was so high was that he averaged just 6.6 yards depth per target (37th in the NFL, per PFF). In general, passes that are thrown longer distances are completed at a lower rate. To see this, you will create [Figure 5-1](#) in Python or [Figure 5-2](#) in R: First, load the data and

then filter pass plays (`play_type == "pass"`) with a passer (`passer_id.notnull()` in Python or `!is.na(passer_id)` in R), and a pass depth (`air_yards.notnull()` in Python or `!is.na(air_yards)` in R). In Python, use this code:

```
import pandas as pd
import numpy as np
import nfl_data_py as nfl
import statsmodels.formula.api as smf
import statsmodels.api as sm
import matplotlib.pyplot as plt
import seaborn as sns

seasons = range(2016, 2022 + 1)
pbp_py = nfl.import_pbp_data(seasons)

pbp_py_pass = \
    pbp_py\
    .query('play_type == "pass" & passer_id.notnull() &' + 
           'air_yards.notnull()')\
    .reset_index()
```

Or with R use this code:

```
library(tidyverse)
library(nflfastR)
library(broom)

pbp_r <- load_pbp(2016:2022)
pbp_r_pass <-
  pbp_r |>
  filter(play_type == "pass" & !is.na(passer_id) &
         !is.na(air_yards))
```

Next, restrict air yards to be greater than 0 yards and less than or equal to 20 yards in order to ensure you have a large enough sample size and summarize the data to calculate the completion percentage, `comp_pct`. Then plot results to create [Figure 5-1](#):

```
## Python
# Change theme for chapter
sns.set_theme(style="whitegrid", palette="colorblind")

# Format and then plot
```

```

pass_pct_py = \
    pbp_py_pass\
    .query('0 < air_yards <= 20')\
    .groupby('air_yards')\
    .agg({'complete_pass': ["mean"]})

pass_pct_py.columns = \
    list(map('_'.join, pass_pct_py.columns))

pass_pct_py\
    .reset_index(inplace=True)
pass_pct_py\
    .rename(columns={'complete_pass_mean': 'comp_pct'}, inplace=True)

sns.regplot(data=pass_pct_py, x='air_yards', y='comp_pct',
            line_kws={'color': 'red'});
plt.show();

```

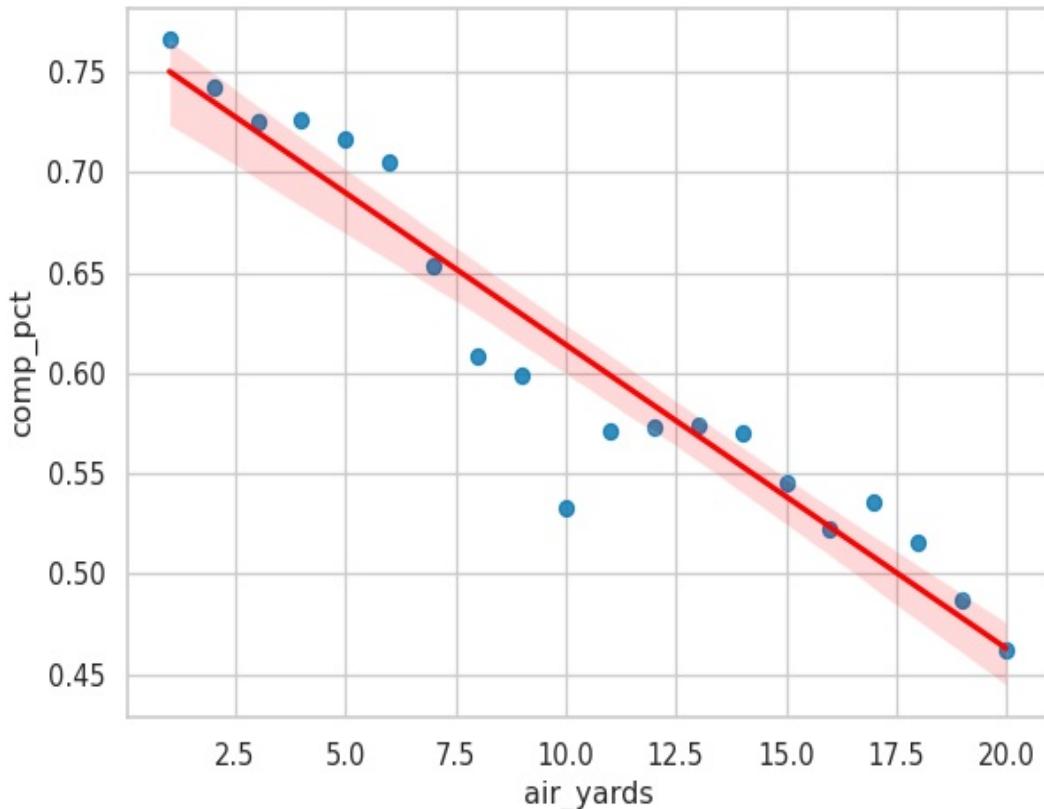


Figure 5-1. Scatterplot with linear trendline for air yards and completion percentage, plotted with seaborn.

Or, use R to create Figure 5-2:

```
## R
pass_pct_r <-
  pbp_r_pass |>
  filter(0 < air_yards & air_yards <= 20) |>
  group_by(air_yards) |>
  summarize(comp_pct = mean(complete_pass),
            .groups = 'drop')

pass_pct_r |>
  ggplot(aes(x = air_yards, y=comp_pct)) +
  geom_point() +
  stat_smooth(method='lm') +
  theme_bw() +
  ylab("Percent completion") +
  xlab("Air yards")
```

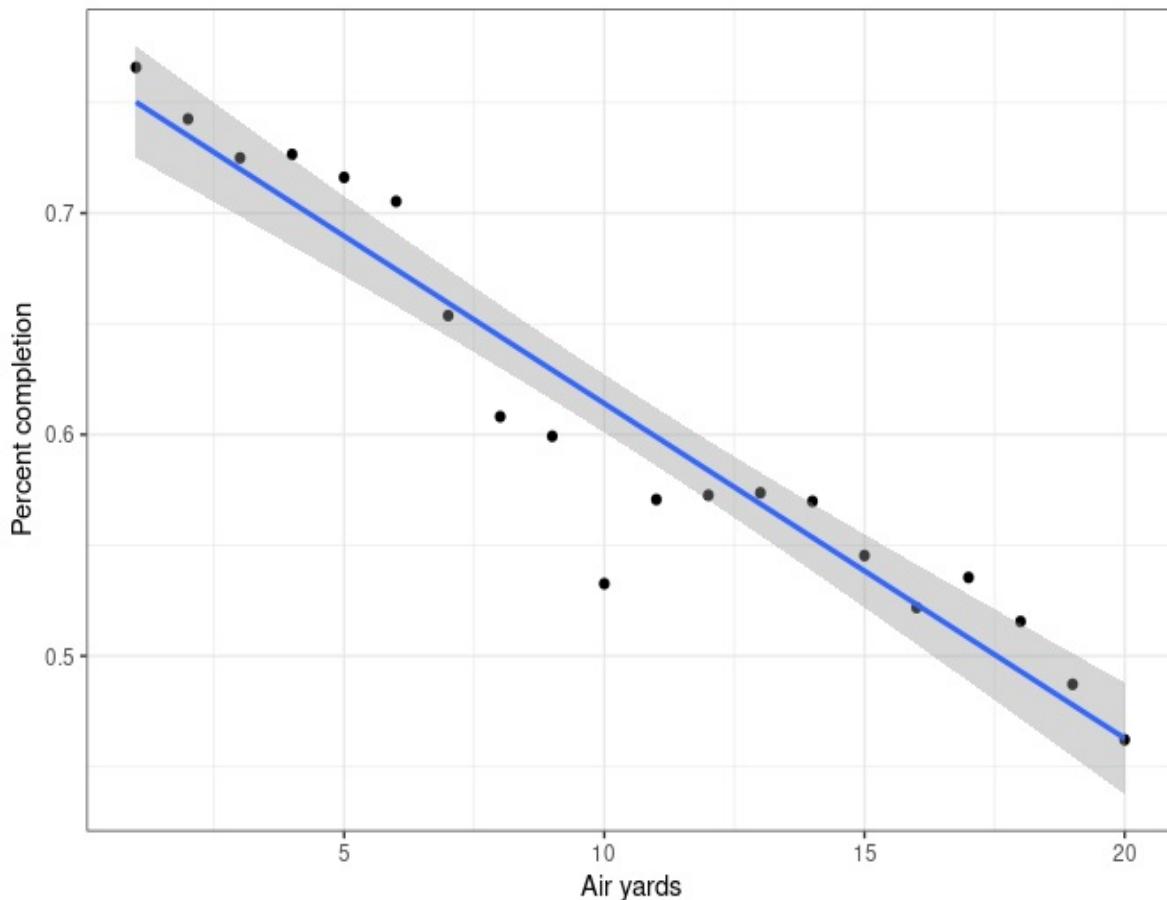


Figure 5-2. Scatterplot with linear trendline for air yards and completion percentage, plotted with `ggplot2`.

Figure 5-1 and **Figure 5-2** clearly show a trend, as expected. Thus, any discussion of quarterback accuracy - as measured by completion percentage - needs to be accompanied by some adjustment for style of play.

Completion percentage over expected, referred to as *CPOE* in the football analytics world, is one of the adjusted metrics that has made its way into the mainstream. Ben Baldwin's [website rbsdm.com](#), a great reference for open football data, has CPOE as one of the main metrics displayed, in large part because CPOE ensemble with EPA per passing play has shown to be the most predictive public metric for quarterback play from year-to-year. The [NFL's Next Generation Stats \(NGS\)](#) group has their own version of CPOE, which includes tracking data-engineered features like receiver separation from nearest defender, prominently displayed on its website. ESPN uses the metric consistently in its broadcasts.

There are some issues with measuring quarterback performance this way, which we will touch on at the end of the chapter, but CPOE is here to stay. We will start start the process of walking you through its development using *generalized linear models*.

General Linear Models

Chapter 3 described the definition and some key assumptions of simple linear regression and these assumptions included:

- The assumption the predictor is linearly related to a single dependent variable, or feature.
- The assumption that one predictor variable (simple linear regression) or more predictor variables (multiple regression) describe the dependent variable.

Another key assumption of multiple regression is that the distribution of the residuals follows a normal or bell-curve distribution. Although almost all datasets violate this last assumption, usually the assumption works “well enough.” However, some data structures cause multiple regression to fail or produce nonsensical results. For example, completion percentage is a value *bounded* between 0 and 1 (*bounded* means the value cannot be smaller than 0 or

larger than 1), as a pass is either incomplete ($pass_complete = 0$) or complete ($pass_complete = 1$). Hence a linear regression, which assumes no bounds on the response variable, is often inappropriate.

Likewise, other data commonly violates this assumption. For example, count data often has too many zeros to be normal and also cannot be negative (such as sacks per game). Likewise binary data with two outcomes (such as win/lose or incomplete/complete pass) and discrete outcomes (such as passing location, which can be right, left, or middle) do not work with multiple regression as response data.

Generalized linear models (or GLMs for short) *generalize* or extended *linear models* (the broad name for multiple and simple regressions as well as some other similar models) to relax the previous assumption of normality. One special type of GLM can be used to model binary data and is covered in this chapter. [Chapter 6](#) covers how to use another type of GLM, the Poisson regression, with count data.

Other types of data can be analyzed by GLMs. For example, discrete outcomes can be analyzed using ordinal regression (also known as ordinal classification), but are beyond the scope of this book. Lastly, linear models (also known as “linear regression” and “ordinary least squares”) are a special type of a GLM, specifically a GLM with a normal, or Gaussian, family.

To understand the basic theory behind how GLMs work, look at a completed pass that can either be 1 (completed) or 0 (incomplete). Because there are two possible outcomes, this a *binary* response and you can assume the a *binomial* distribution does a “good enough job” of describing the data. A normal distribution assumes two parameters: one for the center of the bell-curve (the mean) and a second to describe the width of the the bell-curve (the standard deviation). In contrast, a binomial distribution only requires one parameter: a probability of success. With the pass example, this would be the probability of completing a pass. However, statistically modeling probability is hard because it is bounded by 0 and 1. So, a *link function* converts (or “links”) probability (a value ranging from 0 to 1) to value ranging from $-\infty$ to ∞ . The most common link function is the *logit*, which gives a name to one the most common types of GLMs, the *logistic regression*.

Building a GLM

To apply GLMs, and specifically a logistic regression, you will start with a simple example. Let's start by examining the relation between `air_yards` as our one features for predicting completed passes. As suggested in [Figure 5-1](#) and [Figure 5-2](#), longer passes are less likely to be completed. Now, use a model to quantify this relation.

With Python, use the `glm()` function from `statsmodels.formula.api` (imported as `smf`) as well as the `binomial` family from `statsmodels.api` (imported as `sm`) with the play-by-play data to fit a GLM and then look at the model fit's summary:

```
## Python
complete_ay_py = \
    smf.glm(formula='complete_pass ~ air_yards',
            data=pbp_py_pass,
            family=sm.families.Binomial())\
    .fit();

complete_ay_py.summary()
<class 'statsmodels.iolib.summary.Summary'>
"""
                    Generalized Linear Model Regression Results
=====
=====
Dep. Variable:      complete_pass    No. Observations: 131606
Model:              GLM    Df Residuals: 131604
Model Family:       Binomial    Df Model: 1
Link Function:      Logit    Scale: 1.0000
Method:             IRLS    Log-Likelihood: -81073.
Date:               Sun, 14 May 2023    Deviance: 1.6215e+05
Time:                  09:59:01    Pearson chi2: 1.32e+05
No. Iterations:          5    Pseudo R-squ. (CS): 0.07013
Covariance Type:      nonrobust
=====
=====
```

	coef	std err	z	P> z	[0.025
0.975]					

Intercept	1.0720	0.008	133.306	0.000	1.056
1.088					
air_yards	-0.0573	0.001	-91.806	0.000	-0.059
-0.056					
=====					
=====					
"""					

Likewise, with R, use the `glm()` function that is included with the core R packages and include a `binomial` family and then look at the summary:

```
## R
complete_ay_r <-
  glm(complete_pass ~ air_yards,
      data = ppb_r_pass,
      family = "binomial")

summary(complete_ay_r)
Call:
glm(formula = complete_pass ~ air_yards, family = "binomial",
     data = ppb_r_pass)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) 1.0719692  0.0080414 133.31   <2e-16 ***
air_yards   -0.0573223  0.0006244 -91.81   <2e-16 ***
---
Signif. codes:  0 '****' 0.001 '***' 0.01 '**' 0.05 '*' 0.1 '.' 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 171714  on 131605  degrees of freedom
Residual deviance: 162145  on 131604  degrees of freedom
AIC: 162149

Number of Fisher Scoring iterations: 4
```

TIP

Many of the tools and functions, such as `summary()`, that exist for working with OLS outputs in Python and `lm` outputs in R work on `glm` outputs as well.

Notice how the outputs from both Python and R are similar to the outputs in [Chapter 3](#) and [Chapter 4](#). For both of these models, as `air_yards` increase, and the probability of completion decreases. You care if the coefficients differ from 0 to see if the coefficient is statistically important.

WARNING

Some plots in this book such as [Figure 5-3](#) and [Figure 5-4](#) can take a while (several minutes or more) to plot. If you find yourself slowed down by plotting times on a regular basis when working with your data, consider plotting summaries of data rather than raw data. For example, the binning that was used in “[Exploratory Data Analysis](#)” is one approach. Another tools not covered in this book are `hexbin` plots such as the `hexbin` package in R or `hexbin` plot function in `matplotlib`

To help you see the results from logistic regressions, both Python and R have plotting tools. With Python, use `regplot()` from `seaborn`, but set the `logistic` option to be `True` (to see why a linear regression is a bad idea for this model, use the default option of `False` and notice how the line goes above and below the data) to create [Figure 5-3](#):

```
## Python
sns.regplot(data=pbp_py_pass, x='air_yards', y='complete_pass',
             logistic=True,
             line_kws={'color': 'red'},
             scatter_kws={'alpha':0.05});
plt.show();
```

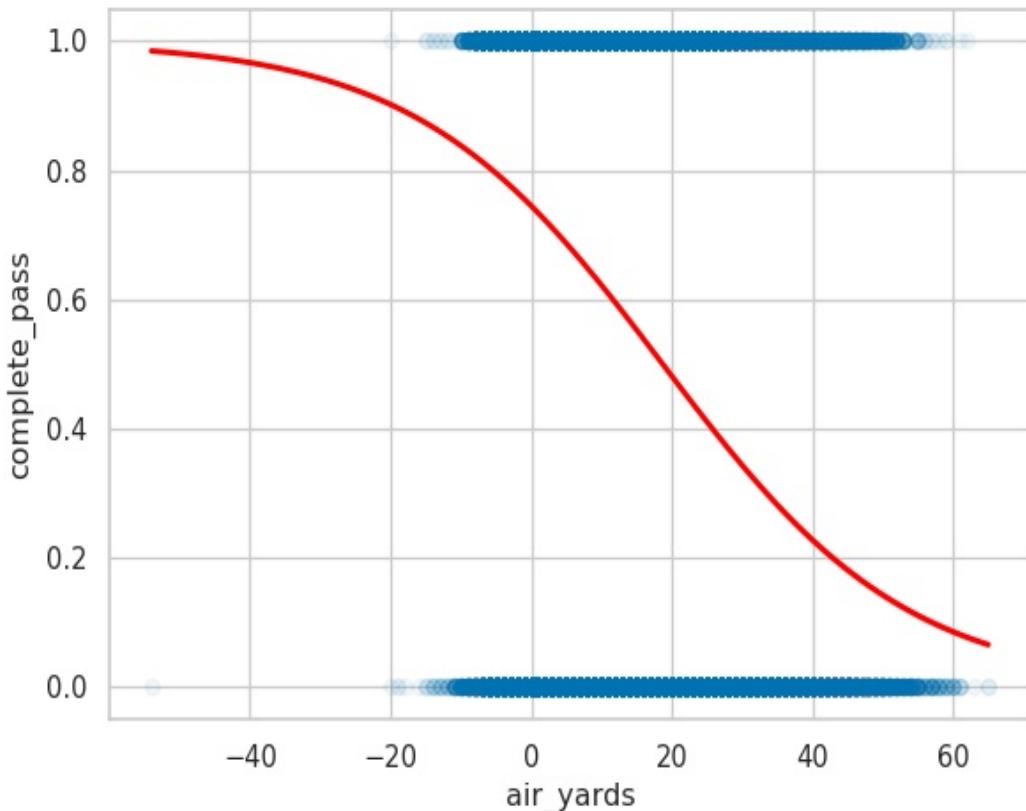


Figure 5-3. Pass competition as a function of air yards, plotted with a logistic curve in seaborn. The curve line is the logistic function. The semitransparent point are the binary outcome for completed passes. Due to the large number of overlapping points, the logistic line is necessary to see any trends in the data.

Likewise, a similar plot may be created using `ggplot2` in R, with jittering on the y-axis to help you see overlapping points, to create [Figure 5-4](#):

```
## R
ggplot(data=pbp_r_pass,
       aes(x=air_yards, y=complete_pass)) +
  geom_jitter(height = 0.05, width = 0,
              alpha = 0.05) +
  stat_smooth(method = 'glm',
              method.args=list(family="binomial")) +
  theme_bw() +
  ylab("Completed pass (1 = yes, 0 = no)") +
  xlab("air yards")
```

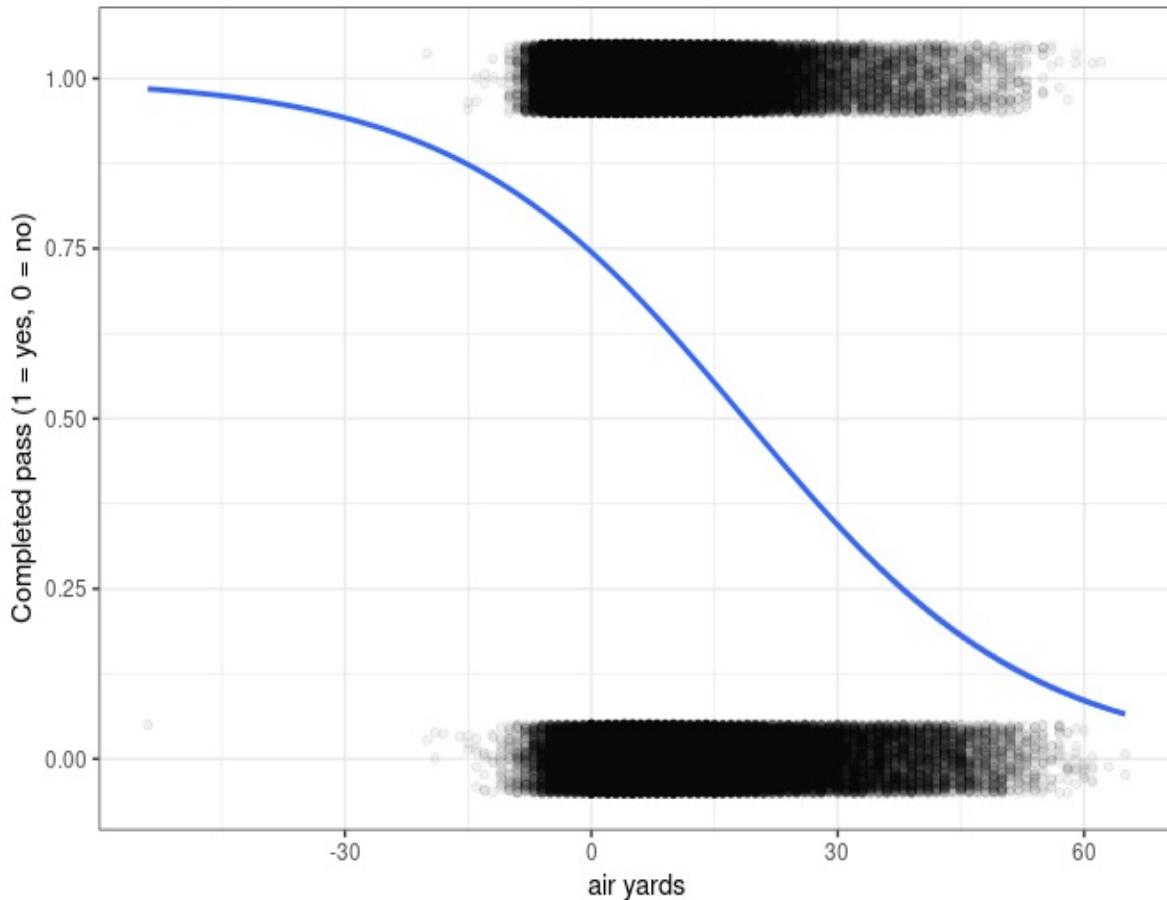


Figure 5-4. Pass competition as a function of air yards, plotted with a logistic curve in `ggplot2`. The curve line is the logistic function. The semitransparent point are the binary outcome for completed passes.

The points are jittered on the y-axis so that they are non-overlapping. Due to the large number of overlapping points, the logistic line is necessary to see any trends in the data.

GLM Application to Completion Percentage

Using the results from “Building a GLM”, extract the expected completion percentage by appending the residuals to the play-by-play pass data frame. With a linear model (or linear regression), the CPOE would simply be the residual because only one type exists. However, different types of residuals exist for GLMs, so you will calculate the residual manually (rather than extracting from the fit) to ensure you know what you are using.

In Python, do this by extracting the predicted value using `predict()` from the model you previously fit and then subtract from the observed value to calculate the CPOE:

```

## Python
pbp_py_pass["exp_completion"] = \
    complete_ay_py.predict()

pbp_py_pass["cpoe"] = \
    pbp_py_pass["complete_pass"] - \
    pbp_py_pass["exp_completion"]

```

WARNING

Because the GLMs model occurs on a different scale than the observed data, different methods exist for calculating the residuals and predicted values. For example, the help file for the `predict()` function in R notes different types of predictions exist and notes the type of prediction required. The default is on the scale of the linear predictors; the alternative 'response' is on the scale of the response variable. Thus for a default binomial model the default predictions are of log-odds (probabilities on logit scale) and `type = 'response'` gives the predicted probabilities.'

In R, take the `pbp_r_pass` datafame and then create a new column using `mutate()`. The new column is called `exp_completion` and gets values by extracting the predicted model fits using the `predict()` function with `type = "resp"` on the model you previously fit. Then subtract from `complete_pass` to calculate the CPOE:

```

## R
pbp_r_pass <-
  pbp_r_pass |>
  mutate(exp_completion = predict(complete_ay_r, type = "resp"),
         cpoe = complete_pass - exp_completion)

```

TIP

The code in this chapter is similar to the code in [Chapter 3](#) and [Chapter 4](#). If you need more details, please refer back to these chapters. Note, however, that GLMs differ from LMs with their output units and structure.

First, look at the leaders in CPOE since 2016, versus leaders in actual completion percentage. Recall that you're only looking at passes that have non-NA `air_yards` readings. Also, only include quarterbacks with 100 or more

attempts. Filtering out the NA data helps you to remove irrelevant plays. Filter out to only include quarterbacks with 100 or more attempts avoids quarterbacks who only had a few plays and would likely be outliers.

In Python, calculate the mean CPOE, the mean completed pass percentage, and then sort by `compl`:

```
## Python
cpoe_py = \
    pbp_py_pass\
    .groupby(["season", "passer_id", "passer"])\\
    .agg({ "cpoe": [ "count", "mean"],\n          "complete_pass": [ "mean"]})\n\n    cpoe_py.columns = \
        list(map('_'.join, cpoe_py.columns))
    cpoe_py.reset_index(inplace=True)\n\n    cpoe_py = \
        cpoe_py\
        .rename(columns = { "cpoe_count": "n",\n                           "cpoe_mean": "cpoe",\n                           "complete_pass_mean": "compl"})\
        .query("n > 100")\n\n    print(
        cpoe_py\
        .sort_values("cpoe", ascending=False)
    )\n\n    season      passer_id      passer      n      cpoe      compl\n299      2019  00-0020531  D.Brees  406  0.094099  0.756158\n193      2018  00-0020531  D.Brees  566  0.086476  0.738516\n467      2020  00-0033537  D.Watson 542  0.073453  0.704797\n465      2020  00-0033357   T.Hill  121  0.072505  0.727273\n22       2016  00-0026143   M.Ryan  631  0.068933  0.702060\n...\n91       2016  00-0033106   J.Goff  204 -0.108739  0.549020\n526      2021  00-0027939   C.Newton 126 -0.109908  0.547619\n112      2017  00-0025430  D.Stanton 159 -0.110229  0.496855\n730      2022  00-0037327  S.Thompson 150 -0.116812  0.520000\n163      2017  00-0031568   B.Petty  112 -0.151855  0.491071\n\n[300 rows x 6 columns]
```

In R, calculate the mean CPOE, the mean completed pass percentage (`compl`), and arrange by `compl`:

```

## R
pbp_r_pass |>
  group_by(season, passer_id, passer) |>
  summarize(n = n(),
            cpoe = mean(cpoe, na.rm = TRUE),
            compl = mean(complete_pass, na.rm = TRUE),
            .groups = "drop") |>
  filter(n >= 100) |>
  arrange(-cpoe) |>
  print(n = 20)
# A tibble: 300 × 6
  season    passer_id   passer      n     cpoe    compl
  <dbl> <chr>        <chr> <int> <dbl> <dbl>
1 2019 00-0020531 D.Brees 406 0.0941 0.756
2 2018 00-0020531 D.Brees 566 0.0865 0.739
3 2020 00-0033537 D.Watson 542 0.0735 0.705
4 2020 00-0033357 T.Hill 121 0.0725 0.727
5 2016 00-0026143 M.Ryan 631 0.0689 0.702
6 2019 00-0029701 R.Tannehill 343 0.0689 0.691
7 2020 00-0023459 A.Rodgers 607 0.0618 0.705
8 2017 00-0020531 D.Brees 606 0.0593 0.716
9 2018 00-0026143 M.Ryan 607 0.0590 0.695
10 2021 00-0036442 J.Burrow 659 0.0564 0.703
11 2016 00-0020531 D.Brees 664 0.0548 0.708
12 2018 00-0032950 C.Wentz 399 0.0546 0.699
13 2018 00-0023682 R.Fitzpatrick 246 0.0541 0.667
14 2022 00-0030565 G.Smith 605 0.0539 0.701
15 2016 00-0027854 S.Bradford 551 0.0529 0.717
16 2018 00-0029604 K.Cousins 603 0.0525 0.705
17 2017 00-0031345 J.Garoppolo 176 0.0493 0.682
18 2022 00-0031503 J.Winston 113 0.0488 0.646
19 2021 00-0023459 A.Rodgers 556 0.0482 0.694
20 2020 00-0034857 J.Allen 692 0.0478 0.684
# i 280 more rows

```

Future Hall of Famer Drew Brees not only has some of the most accurate seasons in NFL history, he also has some of the most accurate seasons in NFL history even after adjusting for pass depth. In the results, Brees had four entries, all in the top 11. Cleveland Browns quarterback Deshaun Watson, who earned the richest fully-guaranteed deal in NFL history at the time in 2022, scored incredibly well in 2020 at CPOE, while in 2016 Matt Ryan not only earned the league's Most Valuable Player (MVP) award, but also led the Falcons to the Super Bowl while generating a 6.9% CPOE per pass attempt. Ryan's 2018 season also appears among the leaders.

In 2020 Aaron Rodgers was the league MVP, while in 2021 Joe Burrow led the

league in yards per attempt and completion percentage en route to leading the Bengals to their first Super Bowl appearance since 1988. Sam Bradford's 2016 season is still a top-5 season all time in terms of completion percentage, since passed a few times by Drew Brees and fellow Saints quarterback Taysom Hill (who also appeared in the RYOE chapter), but does not appear as a top player historically in CPOE, as discussed above.

Pass depth is certainly not the only variable that matters in terms of completion percentage. Let's add a few more features to the model, namely down (`down`), distance to go for a first down (`ydstogo`), distance to go to the end zone (`yardline_100`), pass location (`pass_location`), and whether or not the quarterback was hit (`qb_hit`; more on this later). The formula will also include an interaction between `down` and `ydstogo`.

First, change variables to factors in Python, select the columns you will use and drop the NA values:

```
## Python
# remove missing data and format data
pbp_py_pass['down'] = pbp_py_pass['down'].astype(str)
pbp_py_pass['qb_hit'] = pbp_py_pass['qb_hit'].astype(str)

pbp_py_pass_no_miss = \
    pbp_py_pass[["passer", "passer_id", "season",
                 "down", "qb_hit", "complete_pass",
                 "ydstogo", "yardline_100",
                 "air_yards",
                 "pass_location"]]\ \
    .dropna(axis = 0)
```

Then build and fit the model in Python:

```
## Python
complete_more_py = \
    smf.glm(formula='complete_pass ~ down * ydstogo + ' +
              'yardline_100 + air_yards + ' +
              'pass_location + qb_hit',
              data=pbp_py_pass_no_miss,
              family=sm.families.Binomial())\ \
              .fit()
```

Next, extract the outputs and calculate the CPOE:

```

## Python
pbp_py_pass_no_miss["exp_completion"] = \
    complete_more_py.predict()

pbp_py_pass_no_miss["cpoe"] = \
    pbp_py_pass_no_miss["complete_pass"] - \
    pbp_py_pass_no_miss["exp_completion"]

```

Now summarize the outputs and reformat and rename the columns:

```

## Python
cpoe_py_more = \
    pbp_py_pass_no_miss\
    .groupby(["season", "passer_id", "passer"])\\
    .agg({"cpoe": ["count", "mean"],
          "complete_pass": ["mean"],
          "exp_completion": ["mean"]})

cpoe_py_more.columns = \
    list(map('_'.join, cpoe_py_more.columns))
cpoe_py_more.reset_index(inplace=True)

cpoe_py_more = \
    cpoe_py_more\
    .rename(columns = {"cpoe_count": "n",
                      "cpoe_mean": "cpoe",
                      "complete_pass_mean": "compl",
                      "exp_completion_mean": "exp_completion"})\
    .query("n > 100")

```

Finally, print the top 20 entries (we encourage you to print more, we only print a limited number of rows to save page space):

```

## Python
print(
    cpoe_py_more\
    .sort_values("cpoe", ascending=False)
)

```

	season	passer_id	passer	n	cpoe	compl
exp_completion						
193	2018	00-0020531	D.Brees	566	0.088924	0.738516
0.649592						
299	2019	00-0020531	D.Brees	406	0.087894	0.756158
0.668264						
465	2020	00-0033357	T.Hill	121	0.082978	0.727273
0.644295						
22	2016	00-0026143	M.Ryan	631	0.077565	0.702060

```

0.624495
467    2020  00-0033537  D.Watson  542  0.072763  0.704797
0.632034
...
...
390    2019  00-0035040  D.Blough  174 -0.100327  0.540230
0.640557
506    2020  00-0036312  J.Luton   110 -0.107358  0.545455
0.652812
91     2016  00-0033106  J.Goff    204 -0.112375  0.549020
0.661395
526    2021  00-0027939  C.Newton  126 -0.123251  0.547619
0.670870
163    2017  00-0031568  B.Petty   112 -0.166726  0.491071
0.657798

[300 rows x 7 columns]

```

Likewise, in R remove missing data and format the data:

```

## R
pbp_r_pass_no_miss <-
  pbp_r_pass |>
  mutate(down = factor(down),
         qb_hit = factor(qb_hit)) |>
  filter(complete.cases(down, qb_hit, complete_pass,
                        ydstogo, yardline_100, air_yards,
                        pass_location, qb_hit))

```

Then run the model in R and save the outputs:

```

## R
complete_more_r <-
  pbp_r_pass_no_miss |>
  glm(formula = complete_pass ~ down * ydstogo + yardline_100 +
       air_yards + pass_location + qb_hit,
       family = "binomial")

```

Next, calculate the CPOE:

```

## R
pbp_r_pass_no_miss <-
  pbp_r_pass_no_miss |>
  mutate(exp_completion = predict(complete_more_r, type = "resp"),
         cpoe = complete_pass - exp_completion)

```

Next, summarize the data:

```
## R
cpoe_more_r <-
  pbp_r_pass_no_miss |>
  group_by(season, passer_id, passer) |>
  summarize(n = n(),
            cpoe = mean(cpoe , na.rm = TRUE),
            compl = mean(complete_pass),
            exp_completion = mean(exp_completion),
            .groups = "drop") |>
  filter(n > 100)
```

Finally, print the top 20 entries (we encourage you to print more, we only print a limited number of rows to save page space):

```
## R
cpoe_more_r |>
  arrange(-cpoe) |>
  print(n = 20)
# A tibble: 300 × 7
  season    passer_id   passer      n    cpoe  compl exp_completion
  <dbl>     <chr>       <chr>    <int>  <dbl>  <dbl>        <dbl>
1 2018 00-0020531 D.Brees      566 0.0889 0.739 0.650
2 2019 00-0020531 D.Brees      406 0.0879 0.756 0.668
3 2020 00-0033357 T.Hill      121 0.0830 0.727 0.644
4 2016 00-0026143 M.Ryan      631 0.0776 0.702 0.624
5 2020 00-0033537 D.Watson    542 0.0728 0.705 0.632
6 2019 00-0029701 R.Tannehill  343 0.0667 0.691 0.624
7 2016 00-0027854 S.Bradford  551 0.0615 0.717 0.655
8 2018 00-0023682 R.Fitzpatrick 246 0.0613 0.667 0.605
9 2020 00-0023459 A.Rodgers   607 0.0612 0.705 0.644
10 2018 00-0026143 M.Ryan      607 0.0597 0.695 0.636
11 2018 00-0032950 C.Wentz     399 0.0582 0.699 0.641
12 2017 00-0020531 D.Brees      606 0.0574 0.716 0.659
13 2021 00-0036442 J.Burrow     659 0.0559 0.703 0.647
14 2016 00-0025708 M.Moore     122 0.0556 0.689 0.633
15 2022 00-0030565 G.Smith     605 0.0551 0.701 0.646
16 2021 00-0023459 A.Rodgers   556 0.0549 0.694 0.639
17 2017 00-0031345 J.Garoppolo  176 0.0541 0.682 0.628
18 2018 00-0033537 D.Watson    548 0.0539 0.682 0.629
19 2019 00-0029263 R.Wilson     573 0.0538 0.663 0.609
20 2018 00-0029604 K.Cousins    603 0.0533 0.705 0.652
# i 280 more rows
```

Notice that Brees' top seasons actually flip, with his 2018 season now the best in

terms of CPOE, followed by 2019. This flip occurs because the model has slightly estimates for the players based upon the models have different features. Matt Ryan's 2016 MVP season eclipses Watson's 2020 campaign, while Sam Bradford actually enters back into the fray once we throw game conditions into the mix. Journeyman Ryan Fitzpatrick, who led the NFL in yards per attempt in 2018 for Tampa Bay while splitting time with Jameis Winston, joins the top group as well.

Is CPOE More Stable Than Completion Percentage?

Just like we did for running backs, it's important to determine if CPOE is more stable than simple completion percentage. If it is, we can be more sure that we're isolating the player's performance more so than his surrounding conditions. To this aim, let's dig into code.

First, with Python:

```
## Python
# only keep columns needed
cols_keep =\
    ["season", "passer_id", "passer",
     "cpoe", "compl", "exp_completion"]

# create current dataframe
cpoe_now_py =\
    cpoe_py_more[cols_keep].copy()

# create last-year's dataframe
cpoe_last_py =\
    cpoe_now_py[cols_keep].copy()

# rename columns
cpoe_last_py\
    .rename(columns = {'cpoe': 'cpoe_last',
                      'compl': 'compl_last',
                      'exp_completion': 'exp_completion_last'},
           inplace=True)

# add 1 to season
cpoe_last_py["season"] += 1
```

```
# merge together
cpoe_lag_py =\
    cpoe_now_py\
    .merge(cpoe_last_py,
           how='inner',
           on=['passer_id', 'passer',
                'season'])
```

Then, examine the correlation for pass competition:

```
## Python
cpoe_lag_py[['compl_last', 'compl']].corr()
            compl_last      compl
compl_last      1.000000  0.445465
compl          0.445465  1.000000
```

Followed by CPOE:

```
## Python
cpoe_lag_py[['cpoe_last', 'cpoe']].corr()
            cpoe_last      cpoe
cpoe_last      1.000000  0.464974
cpoe          0.464974  1.000000
```

These calculations may also be done in R:

```
## R
# create current dataframe
cpoe_now_r <-
  cpoe_more_r |>
  select(-n)

# create last-year's dataframe
# and add 1 to season
cpoe_last_r <-
  cpoe_more_r |>
  select(-n) |>
  mutate(season = season + 1) |>
  rename(cpoe_last = cpoe,
         compl_last = compl,
         exp_completion_last = exp_completion
  )

# merge together
cpoe_lag_r <-
  cpoe_now_r |>
```

```

inner_join(cpoe_last_r,
           by = c("passer_id", "passer", "season")) |>
ungroup()

```

Then, select the two passing completion columns and examine the correlation:

```

## R
cpoe_lag_r |>
  select(compl_last, compl) |>
  cor(use="complete.obs")
      compl_last     compl
compl_last  1.0000000  0.4454646
compl       0.4454646  1.0000000

```

Repeat with the CPOE columns:

```

## R
cpoe_lag_r |>
  select(cpoе_last, cpoе) |>
  cor(use="complete.obs")
      cpoе_last     cpoе
cpoe_last  1.0000000  0.4649739
cpoe       0.4649739  1.0000000

```

It looks like CPOE is slightly more stable than pass competition! Thus, from a consistency perspective, you're slightly improving on the situation by building CPOE.

First, and most importantly: the features embedded in the expectation for completion percentage could be fundamental to the quarterback. Some quarterbacks, like Drew Brees, just throw shorter passes characteristically. Others take more hits (in fact, many - **including Eric** - have argued that taking hits is at least partially the quarterback's fault). Some teams throw a lot on early downs, which are easier passes to complete empirically, while others only throw on late downs. Quarterbacks don't switch teams that often, so even if the situation is necessarily inherent to the quarterback himself, the scheme in which they play may be stable.

Last, look at the stability of expected completions:

```

## R
cpoe_lag_r |>

```

```

  select(exp_completion_last, exp_completion) |>
  cor(use="complete.obs")
    exp_completion_last exp_completion
exp_completion_last           1.000000   0.473959
exp_completion                 0.473959   1.000000

```

The most stable metric in this chapter is actually the average *expected completion percentage* for a quarterback.

A Question About Residual Metrics

The results of this chapter shed some light on issues that can arise in modeling football, and sports in general. Trying to strip out context from a metric is rife with opportunities to make mistakes. The assumption that a player doesn't dictate the situation that they are embedded in on a given play is likely violated, and repeatedly.

For example, the NFL NGS version of CPOE includes receiver separation, which at first blush seems like an external factor to the quarterback: whether or not the receiver gets open is not the job of the quarterback. However, quarterbacks contribute to this in a few ways. Firstly, the player they decide to throw to - the separation they choose from - is their choice. Secondly, quarterbacks can move defenders - and hence change the separation profile - with their eyes. Many will remember the no-look pass by Matthew Stafford in Super Bowl LVI.

Lastly, whether or not a quarterback actually passes the ball has some signal. As stated above, Joe Burrow led the league in yards per attempt and completion percentage in 2021. He also led the NFL in sacks taken with 51. Other quarterbacks escape pressure, but to run, while others will throw it away. These alter expectations for reasons that are (at least partially) quarterback-driven.

So, what should someone do? The answer here is the answer to a lot of questions, which is: it depends. If you're trying to determine who the most-accurate passer in the NFL is, this might not necessarily be sufficient (no one metric is likely sufficient to answer this or any football question definitively).

If you're trying to predict player performance for the sake of player acquisition, fantasy football, or sports betting, it's probably okay to try to strip out context in

an expectation and then apply the “over expected” analysis to a well-calibrated model. For example, if you’re trying to predict Patrick Mahomes’ completion percentage during the course of a season, you have to add the expected completion percentage given his circumstances and his CPOE. The latter is considered by many to be almost completely Mahomes, but the former also has some of his game embedded in it as well. To assume the two are completely independent will likely lead to errors.

As you gain access to more and better data, you will also gain the potential to reduce some of these modeling mistakes. It will take diligence in modeling to do so, however. That’s what makes this process fun.

TIP

We encourage you to refine your data skills before buying better" data. Once you reach the limitations of the free data, you'll realize if and why you need better data. And, you'll be able to actually use that data.

A Brief Primer on Odds Ratios

With a logistic regression, the coefficients may be understood in log-odds terminology as well. Most people do not understand log-odds because odds are not commonly used by everyday life. Furthermore, the “odds” in odds ratios are different than betting odds.

WARNING

Odds ratios sometimes can help you understand logistic regression. Other times, they can lead you astray, far astray.

For example, with odds-ratios, if you expect three passes to be completed for every two passes that were incomplete, then the odds ratio would be 3-to-2. The 3-to-2 odds may also be written in decimal form as an odds ratio of 1.5 ($\frac{3}{2} = 1.5$), with an implied “1” in 1.5-to-1.

Odds ratios may be calculated by taking the exponential of the logistic

regression coefficients (the e^x or `exp()` function on many calculators). For example, the `broom` package has a `tidy()` function that readily allows odds ratios to be calculated and displayed:

```
## R
complete_ay_r |>
  tidy(exponentiate = TRUE, conf.int = TRUE)
# A tibble: 2 × 7
  term      estimate std.error statistic p.value conf.low conf.high
  <chr>     <dbl>     <dbl>     <dbl>    <dbl>    <dbl>    <dbl>
1 (Intercept) 2.92     0.00804   133.      0     2.88    2.97
2 air_yards    0.944    0.000624  -91.8     0     0.943   0.945
```

On the odds ratio scale, you care if a value differs from 1 because odds of 1:1 implies the predictor does not change the outcome of an event or the coefficient has no effect on the model's prediction. This intercept now tells you that a pass with zero yards-to-go will be completed with odds of 2.92. However, each additional yard decreases the odds ratio by odds of 0.94.

To see how much, multiply the intercept and the `air_yards` coefficient. Take the `air_yards` coefficient to the exponent for each additional yard (for example `air_yards2` for 2 yards or `air_yards9` for 9 yards). For example, looking at [Figure 5-3](#) or [Figure 5-4](#), you can see that passes with air yards approximately greater than 20 yards have a less than 50% chance of completion. If you multiply 2.92 by 0.94 to the 20 power (2.92×0.94^{20}), you can see that the probability of completing a pass with 20 is 85, which is slightly less than 50% and agrees with [Figure 5-3](#) and [Figure 5-4](#).

To help better understand odds ratios, we will show you how to calculate odds-ratios in R (we use R because the language has nicer tools for working with `glm()` outputs and following the calculations is more important than being able to do them). First, calculate the completion percentage for all data in the play-by-play pass dataset. Next, calculate odds by taking the completion percentage and dividing by one minus the completion percentage. Then, take the natural log to calculate the log-odds:

```
## R
pbp_r_pass |>
  summarize(comp_pct = mean(complete_pass)) |>
  mutate(odds = comp_pct / (1 - comp_pct),
```

```

    log_odds = log(odds))
# A tibble: 1 × 3
  comp_pct  odds  log_odds
  <dbl>   <dbl>     <dbl>
1     0.642   1.79     0.583

```

Next, compare this output to the logistic regression output for a logistic regression with only a global intercept (that is, an average across all observations). First, build the model with a global intercept (`complete_pass ~ 1`). Look at the `tidy()` coefficients for the raw and exponentiated outputs:

```

## R
complete_global_r <-
  glm(complete_pass ~ 1,
      data = pbp_r_pass,
      family = "binomial")

complete_global_r |>
  tidy()
# A tibble: 1 × 5
  term      estimate std.error statistic p.value
  <chr>      <dbl>     <dbl>     <dbl>     <dbl>
1 (Intercept)  0.583    0.00575    101.       0
complete_global_r |>
  tidy(exponentiate = TRUE)
# A tibble: 1 × 5
  term      estimate std.error statistic p.value
  <chr>      <dbl>     <dbl>     <dbl>     <dbl>
1 (Intercept)  1.79     0.00575    101.       0

```

Compare the outputs to the numbers you previously calculated using the percentages. Now, hopefully, you will never need to calculate odds ratios by hand again!

Data Science Tools Used in This Chapter

This chapter including the following topics:

- You learned how to fit a logistic regression in Python and R using `glm()`.
- You learned how to understand and read the coefficients from a logistic regression including odds ratios.

- You reapplied data wrangling tools you learned in previous chapters.

Exercises

1. Repeat this analysis without `qb_hit` as one of the features. How does it change the leaderboard? What can you take from this?
2. What other features could be added to the logistic regression? How does it affect the stability results in this chapter?
3. Try this analysis for receivers. Does anything interesting emerge?
4. Try this analysis for defenses. Does anything interesting emerge?

Suggested Readings

The resources suggested in “Suggested Readings” and “Suggested Readings” will help you understand generalized linear models. Some other readings include

- A PFF article by Eric about quarterbacks: [Quarterbacks in control: A PFF data study of who controls pressure rates](#).
- *Beyond Multiple Linear Regression: Applied Generalized Linear Models and Multilevel Models in R* by Paul Roback and Julie Legler (2021, CRC Press). As the title suggests, this book goes beyond linear regression and does a nice job of teaching generalized linear models including the model’s important assumptions.
- *Bayesian Data Analysis* 3rd edition (2013,CRC Press), by Andrew Gelman, John Carlin, Hal Stern, David Dunson, Aki Vehtari, and Donald Rubin. This books is a classic for advanced modeling skills, but also requires a solid understanding of linear algebra.

Chapter 6. Using Data Science for Sports Betting: Poisson Regression and Passing

Touchdowns :stem: latexmath

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Much progress has been made in the arena of sports betting in the United States specifically, and the world broadly. While Eric was at the Super Bowl during February of 2023, almost every single television and radio show in sight was sponsored by some sort of gaming entity. Just five years earlier sports betting was a taboo topic, discussed only by the fringes, and not legal in any state other than Nevada. That all changed in the spring of 2018, when the Professional and Amateur Sports Protection Act (PASPA) was repealed by the United States Supreme Court, allowing states to determine if and how they would legalize sports betting within their borders.

Sports betting started slowly legalizing throughout the U.S., with the likes of New Jersey and Pennsylvania early adopters, before spreading west to spots like Illinois and Arizona. In all, almost two-thirds of the states in the country have some form of legalized wagering, which has caused a gold rush in offering new

and varied products for gamblers - both recreational and professional.

The betting markets are the single best predictor of what is going to occur on the football field any given weekend for one reason: the wisdom of the crowds (a topic covered in a book *The Wisdom of the Crowds* by Jamie Surowiecki; Random House 2004). Market-making *books* (in gambling, *books* are companies that take bets, and *bookies* are individuals that do the same thing) like Pinnacle, Betcris, and Circa have oddsmakers that *set the line* using a process called *origination* to create the initial price for a game.

Early in the week bettors - both recreational and professional alike - will stake their opinions through making wagers - wagers that are allowed to increase in size as the week - and information like weather and injuries - progress. The *closing line*, in theory, contains all of the opinions, expressed through wagers, of all bettors who have enough influence to move the line into place.

Because markets are assumed to tend towards efficiency, the final prices on a game - described below - are the most accurate (public) predictions available to us. To beat the betting markets, you need to have an *edge*, which is information that gives an advantage not available to other bettors. An edge can generally come from two different sources: better data than the market, or a better way of synthesizing data than the market. The former is usually the product of obtaining information (like injuries) more quickly than the rest of the market, or collecting *longitudinal* data (detailed observations through time, in this game-level data) that no one else bothers to collect (like the early days of PFF)). The latter is generally the approach of most bettors, who use statistical techniques to process data and produce models to set their own prices and bet the discrepancies between their price (an *internal*) and that of the market. That will be the main theme of this chapter.

The Main Markets in Football

In American football, the three main markets have long been the *spread*, the *total*, and the *moneyline*. The *spread* is the most popular market and is pretty easy to understand; it is a point value that is meant to split outcomes in half over a large sample of games. the Washington Commanders are - in a theoretical world where they can play an infinite number of games under the same

conditions against the New York Giants - an average of four points better on the field, than oddsmakers would make the spread between the Commanders and the Giants four points. With this example, five outcomes can occur for betting.

- A person bets for the Commanders to win. The Commanders win by five or more points and the person wins their bet.
- A person bets for the Commanders to win. The Commanders lose outright or do not win by five or more points and the person loses their bet.
- A person bets for the Giants to win. The Giants either win in the game outright or lose by three or fewer points and the person wins their bet.
- A person bets for the Giants to win. The Giants lose outright and the person loses their bet.
- A person bets for either team to win and the game *lands on* (that is to say, the final score is) a four-point differential in favor of the Commanders. This game would be deemed a *push*, with the bettor getting their money back.

It's expected that a point spread bettor without an advantage over the sportsbook would win about 50 percent of his or her bets. This 50 percent comes from the understanding of probability where the spread (theoretically) captures all information about the system. For example, betting on a coin being heads will be correct half of the time and incorrect half of the time over a long time period. For the sportsbook to earn money off of this player, they charge a *vigorish*, or *vig*, on each bet. For point spread bets, this is typically 10 cents on the dollar. Hence, if you wanted to win 100 dollars betting Washington (-4), you would *lay* 110 dollars to win 100. Such a bet requires a 52.38 percent (computed by taking $110/(110 + 100)$) success rate on non-pushed bets over the long haul to break even. Hence, to win at point spread betting, you need have roughly a two-and-a-half-point (52.38 percent - 50 percent is roughly 2.5%) edge over the sportsbook, which given the fact that over 99 percent of sports bettors lose money, is a lot harder than it sounds.

TIP

The old expression 'The house always wins' occurs because the house plays the long game, and has built-in advantages. As an example with a roulette table, the house advantage in the green zero number

or numbers (so you have less than a 50-50 chance of getting red or black). In the case of sports betting, the house gets the vig, which can almost guarantee them a profit unless their odds are consistently and systematically wrong.

To bet a *total*, you simply bet on whether or not the sum of the two team's points goes over or under a specified amount. For example, if the Commanders and Giants had a market total of 43.5 points (-110), to bet under you'd have to lay 110 dollars to win 100 and hope that the sum of the Commanders' and Giants' points were 43 or less. Forty-four or more points would result in a loss of your initial 110-dollar stake. No pushes are possible when the spread or total has a 0.5 tacked onto it. There are bettors who specialize in totals, both full-game totals and totals that are only applicable for certain segments of the game (such as first half or first quarter).

The last of the traditional bets in American football is the *moneyline bet*. Essentially for a *moneyline* bet you're betting on a team to win the game straight up. Since it's rare that a game is a true 50/50 (pick 'em) proposition, to bet a moneyline you either have to lay a lot of money to win a little money (when a team is a *favorite*) or you get to bet a little money to win a lot of money (when a team is an *underdog*). For example, if the Commanders are considered 60 percent likely to win against the Giants, they would have a moneyline price (using North American odds, other countries use decimal odds) of -150; the bettor needs to lay 150 dollars to win 100 dollars. The decimal odds for this bet are $(100 + 150)/150 = 1.67$, which is the ratio of the total return to the investment. -150 is arrived at partially through convention, namely the minus sign in front of the odds for a favorite, and through the computation of $100 \times \frac{0.6}{(1-0.6)} = 150$. The decimal odds here are $(100 + 150)/100 = 2.5$.

The Giants would have a price of +150, meaning that a successful bet of 100 dollars would pay out 150 dollars in addition to the original bet. This is arrived at in the reciprocal way: $100 \times \frac{1-0.4}{0.4} = 150$, with the convention that the plus sign goes in front of the price for the underdog.

NOTE

The book takes some vigorish in moneyline bets, too, so instead of Washington being -150 and New

York being +150, you might see something like Washington being something closer to -160 and New York being something closer to +140 (vigs vary by book). The daylight between the absolute values of -160 and +140 is present in all but rare promotional markets for big games like the Super Bowl.

Application of Poisson Regression: Prop Markets

A model worth its salt in the three main football betting markets using regression is beyond the scope of this book, as they require ratings for each team's offense, defense, and special teams and require adjustments for things like weather and injuries. They are also the markets that attract the highest number of bettors and the most betting *handle* (or total amount of money placed by bettor), which makes them the most efficient betting markets in the United States and among the most efficient betting markets in the world.

Since PASPA, however, sportsbook operators have rushed to create alternatives for bettors who don't want to swim in the deep seas of the spreads, totals, and moneylines of the football betting markets through the proliferation of *proposition* (or *prop*) markets. Historically reserved for big events like the Super Bowl, now all NFL games and most college football games offer bettors the opportunity to bet on all kinds of events (or *props*): Who will score the first touchdown? How many interceptions will Patrick Mahomes have? How many receptions will Tyreek Hill have? Given the sheer volume of available wagers here, it's much, much more difficult for the sportsbook to get each of these prices right, and hence a bigger opportunity for bettors exists in these prop markets.

In this chapter you'll examine the touchdown pass market for NFL quarterbacks. Generally speaking, the quarterback that is starting a game will have a prop market of over/under 0.5 touchdown passes, over/under 1.5 touchdown passes, and for the very best quarterbacks, over/under 2.5 touchdown passes. The number of touchdown passes is called the *index* in this case. Since the number of touchdown passes a quarterback throws in a game is so discrete, the most-important aspect of the prop offering is the price on over and under, which is how the markets create a continuum of offerings in response to bettors' opinions.

Thus, for the most-popular index, over/under 1.5 touchdown passes, one player

might have a price of -140 (lay 140 to win 100) on the over, while another player may have a price of +140 (bet 100 to win 140) on over the same number of touchdown passes. The former is a *favorite* to go over 1.5 touchdown passes, while the latter is an underdog to do so. How these are determined, and whether or not you should bet them, is determined largely by analytics.

The Poisson Distribution

To create or bet a proposition bet-or participate in any betting market, you have to be able to estimate the likelihood, or probability, of events happening. In the canonical example in this chapter, this is the number of touchdown passes thrown by a particular quarterback in a given game.

The simplest way to try to specify these probabilities is to empirically look at the frequencies of each outcome: zero touchdown passes, one touchdown pass, two touchdown passes, and so on. Let's look at NFL quarterbacks with at least 10 passing plays in a given week from 2016 to 2022 to see the frequencies of different touchdown pass outcomes. We use the 10 passing plays threshold as a proxy for being the team's starter, which isn't perfect, but will do for now. Generally speaking, passing touchdown props are only offered for the starters in a given game. You will also use the same filter as [Chapter 5](#) to remove non-passing plays. First, load the data in Python:

```
import pandas as pd
import numpy as np
import nfl_data_py as nfl
import statsmodels.formula.api as smf
import statsmodels.api as sm
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import poisson

seasons = range(2016, 2022 + 1)
pbp_py = \
    nfl.import_pbp_data(seasons)

pbp_py_pass = \
    pbp_py.\
        query('passer_id.notnull()')\
        .reset_index()
```

Or in R:

```
## R
#| eval: false
library(nflfastR)
library(tidyverse)

pbp_r <-
  load_pbp(2016:2022)

pbp_r_pass <-
  pbp_r |>
  filter(!is.na(passer_id))
```

Then, replace null or NA values with 0 for `pass_touchdown`. Python also requires plays without a `passer_id` and `passer` to be changed to "none" so that the data will be summarized correctly.

Next, aggregate by `season`, `week`, `passer_id`, and `passer` to calculate the number of passes per week and the number of touchdown passes per week. Then, filter to exclude players with less than 10 plays as a passer for each week. Next, calculate the number of touchdown passes per quarterback per week.

Lastly, save the `total_line` because you will use this later. This is just *nflfastR*'s name for the market for total points scored, which we discussed earlier in this chapter. We assume that games with different totals will have different opportunities for touchdown passes (e.g. higher totals will have more touchdown passes, on average). The `total_line` is the same throughout a game, so you need to use a function so that Python or R can aggregate a value for the game. A function like `mean()` or `max()` will give you the value for the game, and we used `mean()`. Use this code in Python:

```
## Python
pbp_py_pass\ 
  .loc[pbp_py_pass.pass_touchdown.isnull(), "pass_touchdown"] = 0

pbp_py_pass\ 
  .loc[pbp_py_pass.passer.isnull(), "passer"] = 'none'

pbp_py_pass\ 
  .loc[pbp_py_pass.passer_id.isnull(), "passer_id"] = 'none'

pbp_py_pass_td_y = \
```

```

pbp_py_pass\

    .groupby(["season", "week", "passer_id", "passer"])\

    .agg({"pass_touchdown": ["sum"],

          "total_line": ["count", "mean"]})\

pbp_py_pass_td_y.columns =\

    list(map("_".join, pbp_py_pass_td_y.columns))\

pbp_py_pass_td_y.reset_index(inplace=True)\

pbp_py_pass_td_y\

    .rename(columns={

        "pass_touchdown_sum": "pass_td_y",

        "total_line_mean": "total_line",\

        "total_line_count": "n_passes"

    }),\

    inplace=True,\

)

pbp_py_pass_td_y =\

    pbp_py_pass_td_y\

    .query("n_passes >= 10")\

pbp_py_pass_td_y\

    .groupby("pass_td_y")\

    .agg({"n_passes": "count"})\

    n_passes\

pass_td_y\

    0.0      902\

    1.0     1286\

    2.0     1050\

    3.0      506\

    4.0      186\

    5.0       31\

    6.0        4

```

Or this code in R:

```

## R
pbp_r_pass_td_y <-

  pbp_r_pass |>

  mutate(
    pass_touchdown = ifelse(is.na(pass_touchdown), 0,
                           pass_touchdown)) |>

  group_by(season, week, passer_id, passer) |>

  summarize(
    n_passes = n(),
    pass_td_y = sum(pass_touchdown),

```

```

    total_line = mean(total_line)
) |>
filter(n_passes >= 10)

pbp_r_pass_td_y |>
  group_by(pass_td_y) |>
  summarize(n = n())
# A tibble: 7 × 2
  pass_td_y     n
  <dbl> <int>
1 0         902
2 1        1286
3 2        1050
4 3         506
5 4         186
6 5          31
7 6           4

```

TIP

We were able to group by *season* and *week* because each team only has one game per week. We grouped by *passer_id* and *passer* because *passer_id* is unique (some quarterbacks might have the same name, or at least first initial and last name). We included *passer* because this helps to better understand the data. When using groupings like this on new data, think through how to create unique groups for your specific needs.

Now you can see why the most-popular index is 1.5, since the meat of the empirical distribution is centered at around one touchdown pass, with players with at least 10 pass attempts more likely to throw two or more touchdown passes than they are to throw zero. The mean of the distribution is 1.48 touchdown passes, as you can see here in Python:

```

## Python
pbp_py_pass_td_y\
.describe()
      season      week  pass_td_y  n_passes  total_line
count  3965.000000  3965.000000  3965.000000  3965.000000  3965.000000
mean   2019.048928    9.620177   1.469609   38.798487   45.770618
std    2.008968    5.391064   1.164085   10.620958   4.409124
min   2016.000000    1.000000   0.000000   10.000000   32.000000
25%   2017.000000    5.000000   1.000000   32.000000   42.500000
50%   2019.000000   10.000000   1.000000   39.000000   45.500000
75%   2021.000000   14.000000   2.000000   46.000000   48.500000
max   2022.000000   22.000000   6.000000   84.000000   63.500000

```

Or in R:

```

pbp_r_pass_td_y |>
  ungroup() |>
  select(-passer, -passer_id) |>
  summary()
    season           week        n_passes      pass_td_y
total_line
  Min.   :2016   Min.   : 1.00   Min.   :10.0   Min.   :0.00   Min.
:32.00
  1st Qu.:2017   1st Qu.: 5.00   1st Qu.:32.0   1st Qu.:1.00   1st
Qu.:42.50
  Median :2019   Median :10.00   Median :39.0   Median :1.00   Median
:45.50
  Mean   :2019   Mean   : 9.62   Mean   :38.8   Mean   :1.47   Mean
:45.77
  3rd Qu.:2021   3rd Qu.:14.00   3rd Qu.:46.0   3rd Qu.:2.00   3rd
Qu.:48.50
  Max.   :2022   Max.   :22.00   Max.   :84.0   Max.   :6.00   Max.
:63.50

```

Counts of values are a good place to start, but sometimes you'll need something more. In general there are a number of issues in relying solely on them to make inferences and predictions. The most important issue that arises is one of generalization. This is where probability distributions come in handy.

Touchdown passes aren't the only prop market in which you're going to want to make bets; things like interceptions, sacks, and other low-frequency markets may all have similar quantitative features, and it would benefit us to have a small set of tools in the toolbox from which to work. Furthermore, other markets like passing yards, for which there are 10-fold discrete outcomes, can often have more potential outcomes than outcomes that have occurred in the history of a league and, very likely, the history of a player. A general framework is evidently necessary here.

This is where *probability distributions* come in handy. A probability distribution is a mathematical object that assigns to each possible outcome a value between zero and one, called a *probability*. For discrete outcomes, like touchdown passes in a game, this is pretty easy to understand, and while it might require a formula to compute for each outcome, you can in general get the answer to the question “what is the probability that $X = 0$?” For outcomes that are continuous, like heights, it’s a bit more of a chore and requires tools from calculus. We will stick

with using discrete probability distributions in this book.

One of the most popular discrete probability distributions is the Poisson distribution. This distribution assigns a probability to obtaining discrete values for the integer x (such as $x = 0, 1, 2, 3, \dots$) the value $\frac{e^{\lambda} \lambda^x}{x!}$. In this equation, the Greek letter λ ('lambda') is the average value of the population and “!” is the factorial function. The Poisson distribution models the likelihood of a given number of events occurring in a fixed interval of time or space.

NOTE

The definition for of a factorial is $n! = n \times (n - 1) \times (n - 2) \times (n - 3) \dots \times 2 \times 1$ and $0! = 1$. You might also remember them from math class for their use with permutations. For example, how many ways can three letters (a, b, and c) be arrange? $4! = 6$ or aba, acb, bac, bcb, cab, and cba.

Critical assumptions of the Poisson distribution are:

- The events occur with equal probability.
- The events are independent of the time since the last event.

These assumptions are not exactly satisfied in football, as a team that scores one touchdown in a game may or may not be likely to have “figured out” the defense on the other side of the field, but it’s at least a distribution to consider in modeling touchdown passes in a game by a quarterback.

TIP

Both Python and R have powerful tools for working with statistical distributions. We only brush on these topics in this book. We have found books such as Ben Bolker’s *Ecological Data Analysis with R* (Princeton Press, 2008), to be great resources on applied statistical distributions and their application.

To see if a Poisson is reasonable, let’s look at a bar graph of the frequencies and compare this with the Poisson distribution of the same mean, λ . Do this using this Python code to create [Figure 6-1](#):

```
## Python
```

```

pass_td_y_mean_py =\
    pbp_py_pass_td_y\
        .pass_td_y\
        .mean()

plot_pos_py =\
    pd.DataFrame(
        {"x": range(0, 7),
         "expected": [poisson.pmf(x, pass_td_y_mean_py) for x in
range(0, 7)]}
    )

sns.histplot(pbp_py_pass_td_y["pass_td_y"], stat="probability");
plt.plot(plot_pos_py.x, plot_pos_py.expected);
plt.show();

```

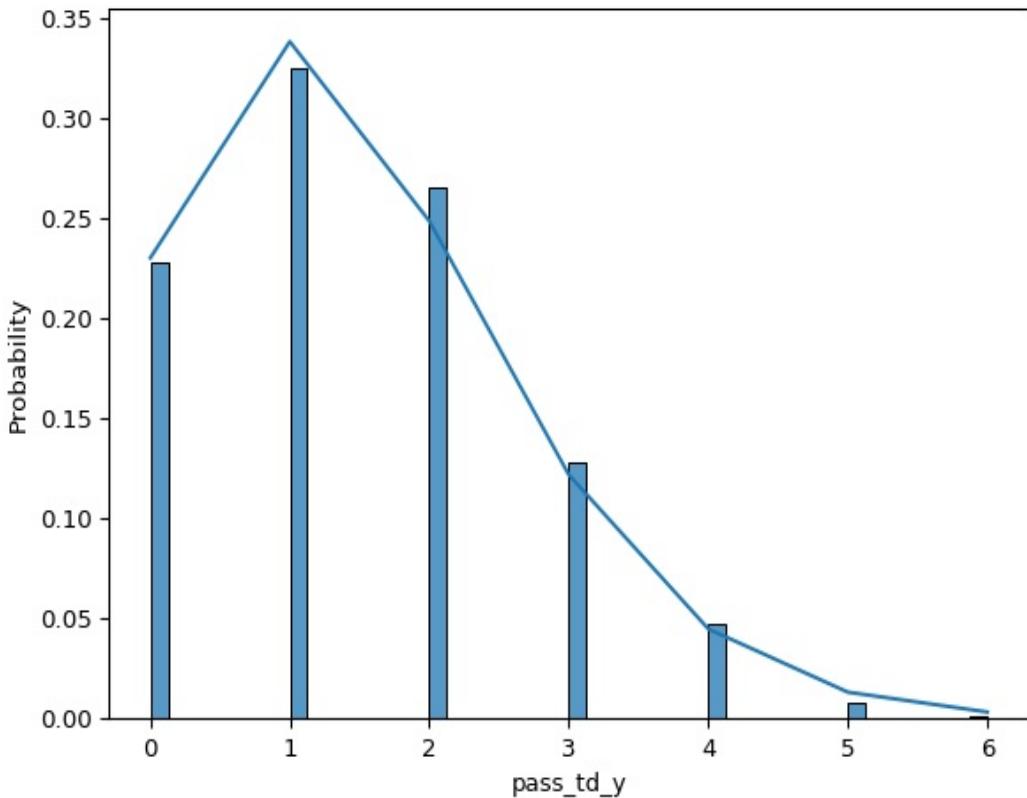


Figure 6-1. Histogram (vertical bars) of normalized observed touchdowns per game per quarterback with at least 10 games. The term normalized refers to the fact that all of the bars sum to 1. The line shows the theoretical expected values from the Poisson distribution. This figure was plotted with seaborn.

Or this R code to create Figure 6-2:

```

## R
pass_td_y_mean_r <-
  pbp_r_pass_td_y |>
  pull(pass_td_y) |>
  mean()

plot_pos_r <-
  tibble(x = seq(0, 7)) |>
  mutate(expected = dpois(
    x = x,
    lambda = pass_td_y_mean_r
  ))

ggplot() +
  geom_histogram(
    data = pbp_r_pass_td_y,
    aes(
      x = pass_td_y,
      y = after_stat(count / sum(count))
    ),
    binwidth = 0.5
  ) +
  geom_line(
    data = plot_pos_r, aes(x = x, y = expected),
    color = "red", linewidth = 1
  ) +
  theme_bw() +
  xlab("Touchdown passes per player per game for 2016 to 2022") +
  ylab("Probability")

```

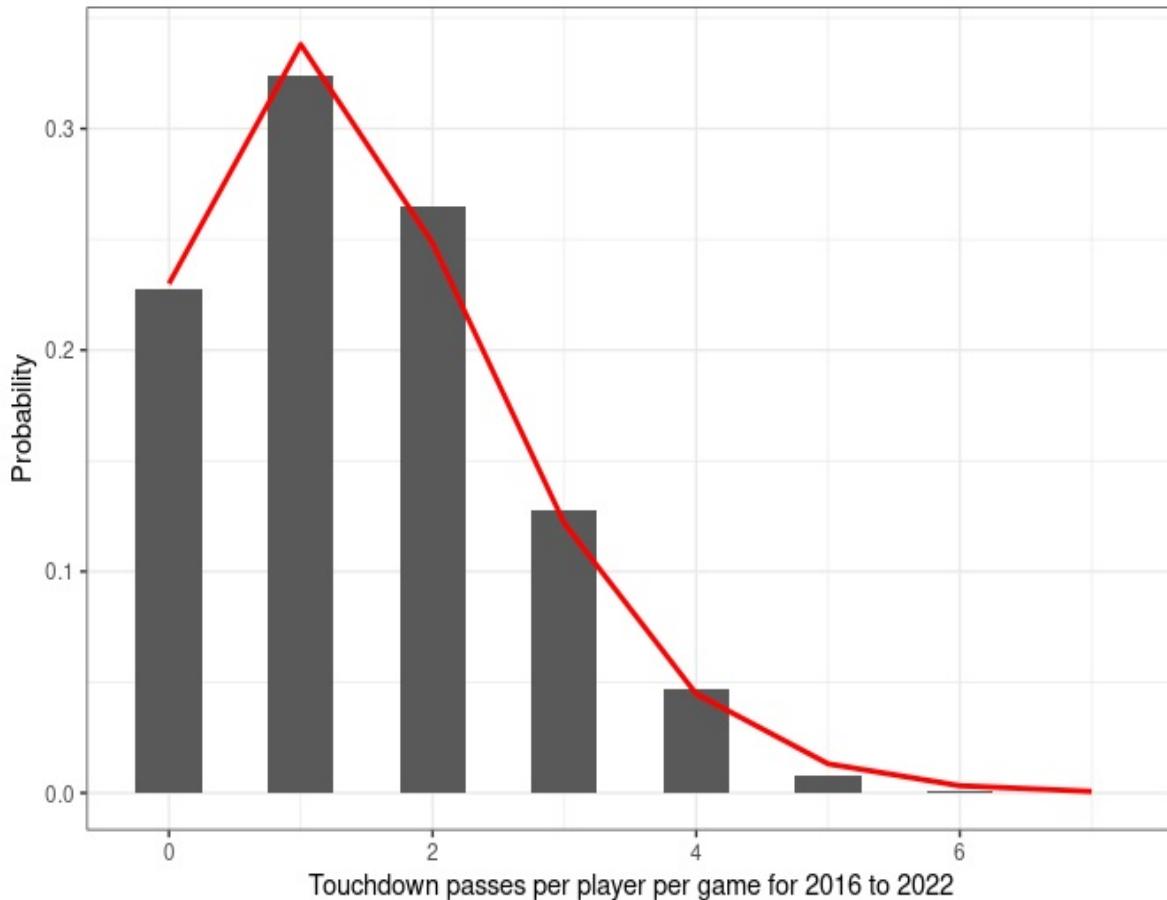


Figure 6-2. Histogram (vertical bars) of normalized observed touchdowns per game per quarterback with at least 10 games. The normalized means that all of the bars sum to 1. The line shows the theoretical expected values from the Poisson distribution. This figure was plotted with `ggplot2`.

The Poisson distribution seems to slightly overestimate the likelihood of one touchdown pass, and as such slightly underestimate the likelihood of zero, two, or more touchdown passes in a game.

Although not large, such discrepancies can be the difference between winning and losing in sports betting, so if you actually want to wager on your opinions some adjustments need to be made—or a different distribution entirely (such as a negative Binomial distribution or quasi-Poisson that accounts for over-dispersion) should be used. But for the sake of this book we’re going to assume a Poisson distribution is sufficient to handicap touchdown pass probabilities in this chapter.

Individual Player Markets and Modeling

Each quarterback, and the player's opponent, varies in quality, and hence each player will have a different market each game regarding touchdown passes. For example, in Super Bowl LVII between the Kansas City Chiefs and Philadelphia Eagles, Patrick Mahomes had a touchdown pass prop of 2.5 touchdown passes (per DraftKings Sportsbook), with the over priced at +150 (bet 100 to win 150) and the under priced at -185 (bet 185 to win 100).

As discussed above, these prices reflect probabilities that one is betting against. In the case of the over, since the price is > 0 , the formula for computing the break-even probability is $100/(100 + 150) = 0.4$. In other words, to bet the over, you have to be confident that Mahomes has better than a 40% chance to throw three or more touchdown passes against the Eagles defense, which ranked number one in the NFL during the 2022 season.

For the under, the break-even probability is $185/(100 + 185) = 0.649$, and hence you have to be more than 64.9% confident that Mahomes will throw for two or fewer touchdowns to make a bet on the under. Notice that these probabilities add to 104.9%, with the 4.9% representing the house edge, or *hold*, which is created by the book charging the vig discussed above.

To specify these probabilities for Mahomes, you could simply go through the history of games Mahomes has played and look at the proportion of games with two or fewer touchdowns and three or more touchdowns to compare. This is faulty for a few reasons, the first of which is that it doesn't consider exogenous factors like opponent-strength, weather, changes to supporting cast, or similar factors. It also doesn't factor changes to league-wide environments like what happened during COVID, where the lack of crowd noise significantly helped offenses.

Now, there are manifold ways of incorporating this into a model, and one who is betting for real should consider as many factors as is reasonable. Here, you will use the aforementioned *total* for the game - the number of points expected by the betting markets to be scored. This factors in defensive strength, pace, and weather come together into one number. In addition to this number, you will use as a feature the mean number of touchdown passes by the quarterback of interest over the previous two seasons.

TIP

`for`-loops are a powerful tool in programming. When we are starting to build `for`-loops, and especially nested `for`-loops, we will start and simply print the indices. For example, we would use code in Python and then fill in details for each index:

```
## Python
for season_idx in range(2017, 2022 + 1):
    print(season_idx)
    for week_idx in range(1, 22 + 1):
        print(week_idx)
```

We do this for two reasons. First, this makes sure the indexing is working. Second, we now have `season_idx` and `week_idx` in memory. If we can get our code working for this two index examples, there is a good chance our code will work for the rest of the index values `for`-loop.

This is what you're going to call x in your training of the model. With Python, use this code:

```
## Python
# pass_ty_d greater than or equal to 10 per week
pbp_py_pass_td_y_geq10 =\
    pbp_py_pass_td_y.query("n_passes >= 10")

# take the average touchdown passes for each QB for the previous
# season
# and current season up to the current game
x_py = pd.DataFrame()
for season_idx in range(2017, 2022 + 1):
    for week_idx in range(1, 22 + 1):
        week_calc_py = (
            pbp_py_pass_td_y_geq10\
                .query("(season == " +
                    str(season_idx - 1) +
                    ") | " +
                    "(season == " +
                    str(season_idx) +
                    "&" +
                    "week < " +
                    str(week_idx) +
                    ")")\
                .groupby(["passer_id", "passer"])\\
                .agg({"pass_td_y": ["count", "mean"]}))
    )
    week_calc_py.columns =\
        list(map("_".join, week_calc_py.columns))
```

```

week_calc_py.reset_index(inplace=True)
week_calc_py\
    .rename(columns={
        "pass_td_y_count": "n_games",
        "pass_td_y_mean": "pass_td_rate"},\
    inplace=True)
week_calc_py["season"] = season_idx
week_calc_py["week"] = week_idx
x_py = pd.concat([x_py, week_calc_py])

```

WARNING

Nested loops can quickly escalate computational times and decrease code readability. If you find yourself using many nested loops, consider learning other coding methods such as vectorization. Here, we use loops because loops are easier to understand, and, computer performance is not important.

Or with R, use this code:

```

## R
# pass_ty_d greater than or equal to 10 per week
pbp_r_pass_td_y_geq10 <-
  pbp_r_pass_td_y |>
  filter(n_passes >= 10)

# take the average touchdown passes for each QB for the previous
# season
# and current season up to the current game
x_r <- tibble()

for (season_idx in seq(2017, 2022)) {
  for (week_idx in seq(1, 22)) {
    week_calc_r <-
      pbp_r_pass_td_y_geq10 |>
      filter((season == (season_idx - 1)) |
              (season == season_idx & week < week_idx)) |>
      group_by(passer_id, passer) |>
      summarize(
        n_games = n(),
        pass_td_rate = mean(pass_td_y),
        .groups = "keep"
      ) |>
      mutate(season = season_idx, week = week_idx)

    x_r <- bind_rows(x_r, week_calc_r)
  }
}

```

TIP

By historic convention, many people use `i`, `j`, and `k` for the indices in for loops such as `for i in ...`. Richard prefers to use longer terms like `season_idx` or `week_idx` for three reasons. First, the words are more descriptive and help him see what is going on in the code. Second, the words are easier to the "find" tools. Third, the words are less likely to be repeated elsewhere in the code.

Notice here for every player going into each week, you have their average number of touchdown passes. Let's look at Patrick Mahomes going into Super Bowl LVII in Python:

```
## Python
x_py.query('passer == "P.Mahomes"]').tail()
  passer_id    passer  n_games  pass_td_rate  season  week
39  00-0033873 P.Mahomes      36     2.444444  2022    18
40  00-0033873 P.Mahomes      37     2.405405  2022    19
40  00-0033873 P.Mahomes      37     2.405405  2022    20
40  00-0033873 P.Mahomes      38     2.394737  2022    21
40  00-0033873 P.Mahomes      39     2.384615  2022    22
```

Or, in R:

```
## R
x_r |>
  filter(passer == "P.Mahomes") |>
  tail()
# A tibble: 6 × 6
  passer_id    passer  n_games  pass_td_rate  season  week
  <chr>        <chr>    <int>       <dbl>    <int>   <int>
1 00-0033873 P.Mahomes      35       2.43    2022    17
2 00-0033873 P.Mahomes      36       2.44    2022    18
3 00-0033873 P.Mahomes      37       2.41    2022    19
4 00-0033873 P.Mahomes      37       2.41    2022    20
5 00-0033873 P.Mahomes      38       2.39    2022    21
6 00-0033873 P.Mahomes      39       2.38    2022    22
```

Looks like the books set a decent number. Mahomes's average prior to that game using data from 2021 and 2022 up to week 22 was 2.38 touchdown passes per game. We now have to create our response variable, which is simply the dataframe `pbp_pass_td_y_geq10` above with the added game total. In Python, use a `merge()` function:

```
## Python
pbp_py_pass_td_y_geq10 =\
    pbp_py_pass_td_y_geq10.query("season != 2016")\
    .merge(x_py,
        on=["season", "week", "passer_id", "passer"],
        how="inner")
```

In R, use an `inner_join()` function (Appendix C provides more details on joins):

```
### R
pbp_r_pass_td_y_geq10 <-
    pbp_r_pass_td_y_geq10 |>
    inner_join(x_r,
        by = c(
            "season", "week",
            "passer_id", "passer"
        )
    )
```

You've now merged the datasets together to get your training dataset for the model. Before you model that data, quickly peek at it using `ggplot2` in R. First, plot passing touchdowns in each game for each passer using a line (using the `passer_id` column rather than `passer`). In the plot, facet by season and add meaningful caption. Save this as `weekly_passing_id_r_plot` and look at Figure 6-3:

```
## R
weekly_passing_id_r_plot <-
    pbp_r_pass_td_y_geq10 |>
    ggplot(aes(x = week, y = pass_td_y, group = passer_id)) +
    geom_line(alpha = 0.25) +
    facet_wrap(vars(season), nrow = 3) +
    theme_bw() +
    theme(strip.background = element_blank()) +
    ylab("Total passing touchdowns") +
    xlab("Week of season")
weekly_passing_id_r_plot
```

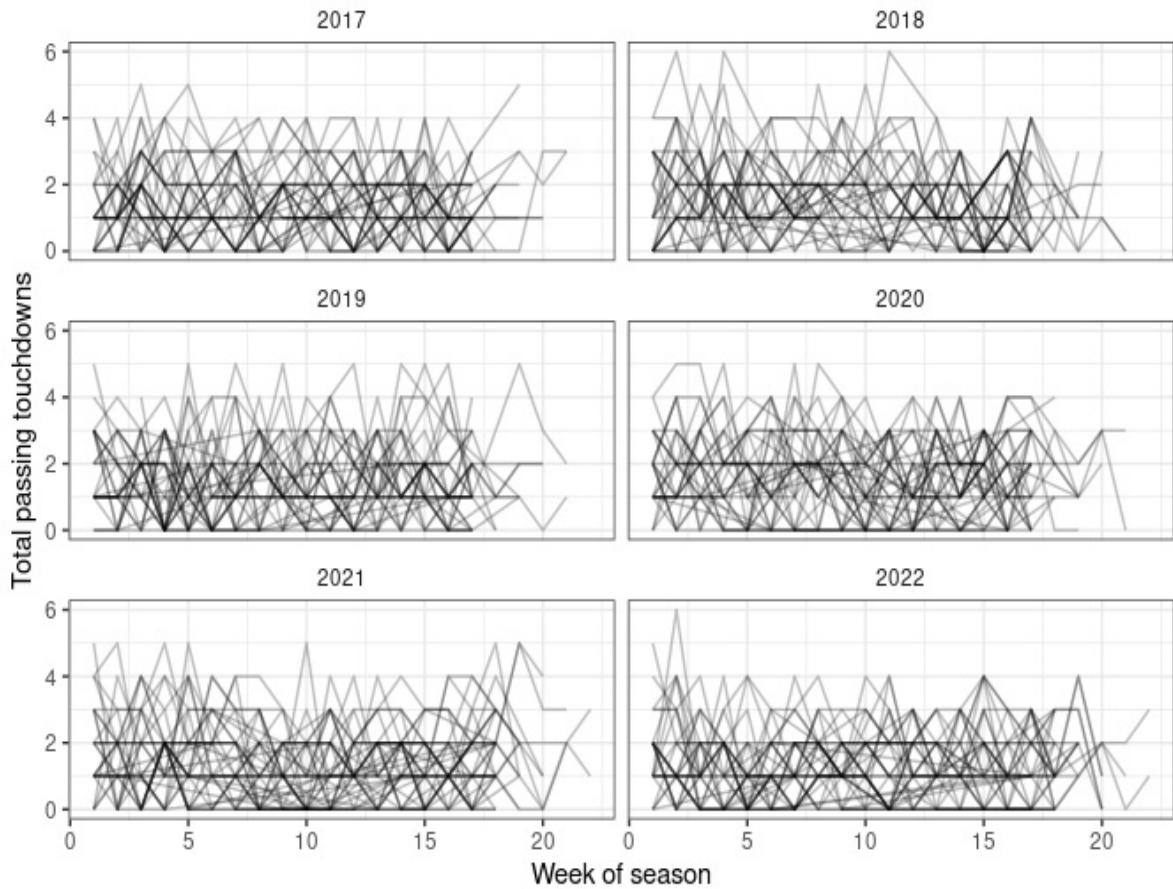


Figure 6-3. Weekly passing touchdowns throughout the 2017 to 2022 seasons. Each line corresponds to passer.

Figure 6-3 shows the variability in the passing touchdowns per game. The values seem to be constant through time and no trends emerge. Add a Poisson regression trend line to the plot to create **Figure 6-4**:

```
## R
weekly_passing_id_r_plot +
  geom_smooth(method = 'glm', method.args = list("family" =
"poisson"),
  se=FALSE,
  linewidth = 0.5, color = 'blue',
  alpha = 0.25)
```

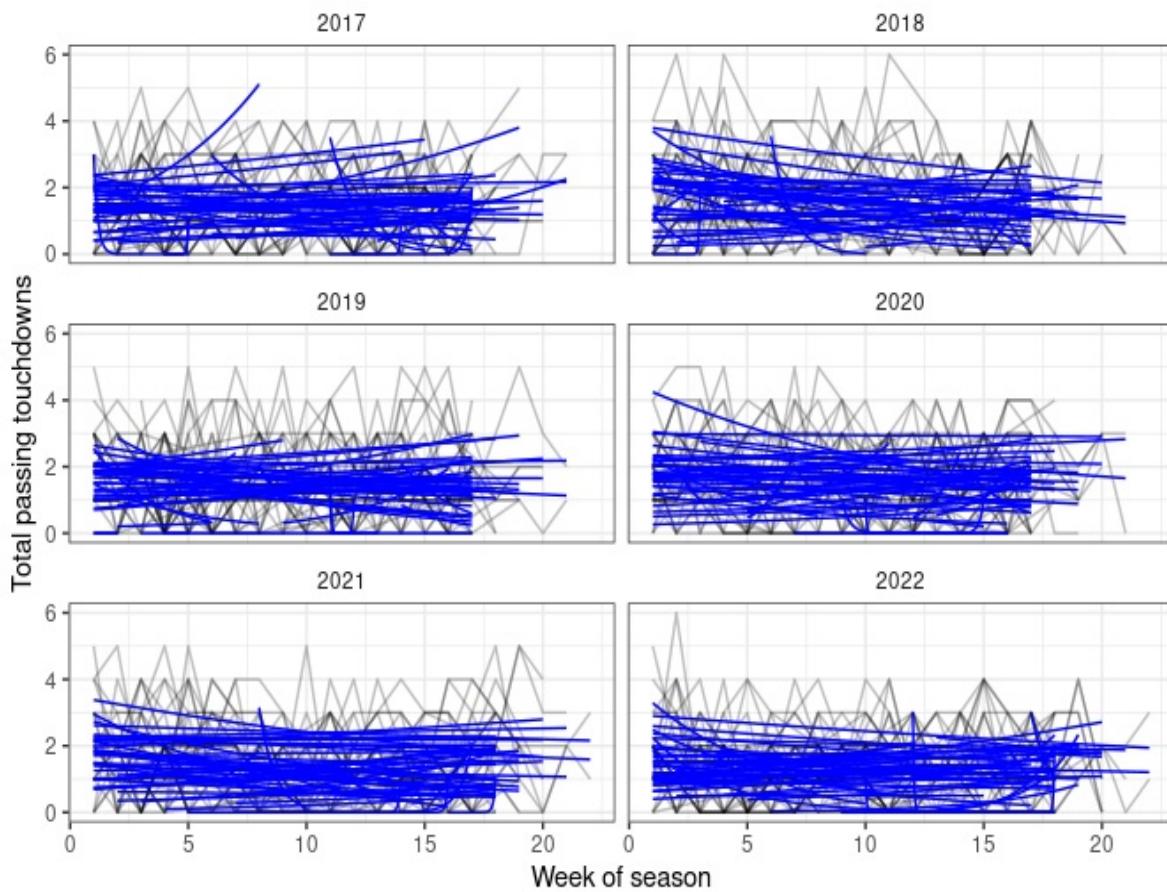


Figure 6-4. Weekly passing touchdowns throughout the 2017 to 2022 seasons. Each line corresponds to passer. The trendline is from a Poisson regression.

At first glance, no trends emerge in [Figure 6-4](#). Players generally have a stable expected total passing touchdowns per game over the course of the seasons, with substantial variation week-to-week. Next, investigate this data using a model.

NOTE

[Figure 6-3](#): and [Figure 6-4](#) do not provide much insight. We include them in this book to help you see the process of using EDA to check your data as you go. Most likely, these figures would not be used in communication unless a client specifically asked if a trend existed through time or you were writing a long, technical report or homework assignment

Since you are assuming a Poisson distribution for the number of touchdown passes thrown in a game by a player, you use a *Poisson regression* as your model. The code to fit a Poisson regression is similar to a logistic regression

from Chapter 5. However, the `family` is now Poisson rather than binomial. Poisson is required to do a Poisson regression. In Python use this code to fit the model, save the outputs to the column in the data called `exp_pass_td`, and look at the summary:

```
## Python
pass_fit_py = \
    smf.glm(
        formula="pass_td_y ~ pass_td_rate + total_line",
        data=pbp_py_pass_td_y_geq10,
        family=sm.families.Poisson())\
    .fit()

pbp_py_pass_td_y_geq10["exp_pass_td"] = \
    pass_fit_py\
    .predict()

print(pass_fit_py.summary())
                Generalized Linear Model Regression Results
=====
=====
Dep. Variable:          pass_td_y    No. Observations:      3297
Model:                  GLM     Df Residuals:           3294
Model Family:            Poisson   Df Model:                 2
Link Function:           Log     Scale:                   1.0000
Method:                 IRLS    Log-Likelihood: -4873.8
Date:          Sat, 20 May 2023   Deviance:             3395.2
Time:              15:47:24     Pearson chi2:           2.83e+03
No. Iterations:          5     Pseudo R-squ. (CS):  0.07146
Covariance Type:         nonrobust
=====
=====
                    coef      std err       z     P>|z|    [0.025
0.975]
-----
Intercept      -0.9851      0.148    -6.641      0.000    -1.276
-0.694
pass_td_rate    0.3066      0.029   10.706      0.000     0.251
```

0.363					
total_line	0.0196	0.003	5.660	0.000	0.013
0.026					
=====	=====	=====	=====	=====	=====
=====	=====	=====	=====	=====	=====

Likewise, in R use the following code to fit the model, save the outputs to the column in the data called `exp_pass_td` (note that you need to use `type = "response"` to put the output on the data scale rather than the coefficient/model scale), and look at the summary:

```
## R
pass_fit_r <-
  glm(pass_td_y ~ pass_td_rate + total_line,
      data = pbp_r_pass_td_y_geq10,
      family = "poisson"
  )

pbp_r_pass_td_y_geq10 <-
  pbp_r_pass_td_y_geq10 |>
  ungroup() |>
  mutate(exp_pass_td = predict(pass_fit_r, type = "response"))

summary(pass_fit_r) |>
  print()
Call:
glm(formula = pass_td_y ~ pass_td_rate + total_line, family =
  "poisson",
  data = pbp_r_pass_td_y_geq10)

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept) -0.985076   0.148333 -6.641 3.12e-11 ***
pass_td_rate 0.306646   0.028643 10.706 < 2e-16 ***
total_line    0.019598   0.003463   5.660 1.52e-08 ***
---
Signif. codes:  0 '****' 0.001 '***' 0.01 '**' 0.05 '*' 0.1 '.' 1

(Dispersion parameter for poisson family taken to be 1)

Null deviance: 3639.6 on 3296 degrees of freedom
Residual deviance: 3395.2 on 3294 degrees of freedom
AIC: 9753.5

Number of Fisher Scoring iterations: 5
```

WARNING

The coefficients and predictions from a Poisson regression depend upon the scale for the mathematical link function, just like the logistic regression in “[A Brief Primer on Odds Ratios](#)”. “[Understanding Poisson Regression Coefficients](#)” provides a brief primer on understanding the outputs from a Poisson regression.

Look at the coefficients and let’s interpret them here. For the Poisson regression, coefficients are on an exponential scale (see the preceding warning on this topic for extra details). In Python access the model’s parameters and than take the exponential using the Numpy library (`np.exp()`):

```
## Python
np.exp(pass_fit_py.params)
Intercept      0.373411
pass_td_rate   1.358860
total_line     1.019791
dtype: float64
```

In R use the `tidy()` function to look at the coefficients:

```
## R
library(broom)
tidy(pass_fit_r, exponentiate = TRUE, conf.int = TRUE)
# A tibble: 3 × 7
  term            estimate std.error statistic p.value conf.low conf.high
  <chr>          <dbl>     <dbl>     <dbl>    <dbl>    <dbl>    <dbl>
1 (Intercept)    0.373     0.148     -6.64  3.12e-11    0.279   0.499
2 pass_td_rate   1.36      0.0286    10.7   9.55e-27    1.28     1.44
3 total_line     1.02      0.00346    5.66   1.52e- 8    1.01     1.03
```

First, look at the `pass_td_rate` coefficient. For this coefficient, multiply every additional touchdown pass in the player’s history by `1.36` to get the expected number of touchdown passes. Second, look at the `total_line` coefficient. For this coefficient, multiply the total line by `1.02`. In this case, the total line is pretty efficient and is within 2% of the expected value ($1 - 1.02 = -0.02$). Both coefficients differ statistically from zero on the (additive) model

scale or differ statistically from 1 on the (multiplicative) data scale.

Now, look at Mahomes' data from Super Bowl LVII, leaving out the actual result for now (as well as the `passer_id` to save space) in Python:

```
## Python
# specify filter criteria on own line for space
filter_by = 'passer == "P.Mahomes" & season == 2022 & week == 22'
# specify columns on own line for space
cols_look = [
    "season",
    "week",
    "passer",
    "total_line",
    "n_games",
    "pass_td_rate",
    "exp_pass_td",
]
pbp_py_pass_td_y_geq10.query(filter_by)[cols_look]
    season  week      passer  total_line  n_games  pass_td_rate
exp_pass_td
3295      2022     22  P.Mahomes        51.0       39      2.384615
2.107833
```

Or, in R:

```
## R
pbp_r_pass_td_y_geq10 |>
  filter(passer == "P.Mahomes",
         season == 2022, week == 22) |>
  select(-pass_td_y, -n_passes, -passer_id, -week, -season, -
n_games)
# A tibble: 1 × 4
  passer    total_line pass_td_rate exp_pass_td
  <chr>        <dbl>        <dbl>        <dbl>
1 P.Mahomes      51          2.38        2.11
```

Now, what are these numbers?

- *n_games* shows that there are a total of 39 games (21 from the 2022 season and 18 from the previous season) in Patrick Mahomes' sample that we are considering.
- *pass_td_rate* is the current average number of touchdown passes per game

by Patrick Mahomes in the sample that we are considering.

- `exp_pass_td` is the expected number of touchdown passes from the model for Patrick Mahomes in the Super Bowl.

And, what do the numbers mean? These numbers show that Mahomes was expected to go under his previous average, even though his game total of 51 was relatively high. Likely, part of this expected touchdown passes results from the concept of “regression towards the mean.” That is to say, when people are above average, statistical models expect them to decrease to be closer to average (and the converse would be true as well, below average players would be expected to increase to be closer to the average). Because Mahomes is the best quarterback in the game at this point, the model predicts he is more likely to have a decrease rather than increase. See [Chapter 3](#) for more about discussion about *regression towards the mean*.

Now, the average number doesn’t really tell you that much. It’s under 2.5, but the betting market already makes under 2.5 the favorite outcome. The question you’re trying to ask is “it is too much or too little of a favorite?” To do this, you can use `exp_pass_td` as Mahomes’ λ value and compute these probabilities explicitly using the probability mass function (*pmf*) and cumulative density function (*cdf*).

A *probability mass function* gives you the probability of a discrete event occurring, assuming a statistical distribution. With our example, this is probability for a passer completing a single number of touchdown passes per game such as 1 touchdown pass or 3 touchdown passes. A *cumulative density function* gives you sum of the probability of different events occurring, assuming a statistical distribution. With our example, this would be the probability of completing X or fewer touchdown passes. For example using $X = 2$, this would be the probability of completing 0, 1, and 2 touchdown passes in a given week.

Python uses relatively straight forward names for PMFs and CDFs. Simply append the name to a distribution. In Python, use `poisson.pmf()` to give the probability of 0, 1, or 2 touchdown passes being scored in a game by a quarterback. Use `1 - poisson.cdf()` to calculate the probability of more than 2 touchdown passes being scored in a game by a quarterback:

```

## Python
pbp_py_pass_td_y_geq10["p_0_td"] = \
    poisson.pmf(k=0,
                 mu=pbp_py_pass_td_y_geq10["exp_pass_td"])

pbp_py_pass_td_y_geq10["p_1_td"] = \
    poisson.pmf(k=1,
                 mu=pbp_py_pass_td_y_geq10["exp_pass_td"])

pbp_py_pass_td_y_geq10["p_2_td"] = \
    poisson.pmf(k=2,
                 mu=pbp_py_pass_td_y_geq10["exp_pass_td"])

pbp_py_pass_td_y_geq10["p_g2_td"] = \
    1 - poisson.cdf(k=2,
                      mu=pbp_py_pass_td_y_geq10["exp_pass_td"])

```

Then look at the outputs for Mahome going into the “big game” (or the Super Bowl for readers who aren’t familiar with football) from Python:

```

## Python
# specify filter criteria on own line for space
filter_by = 'passer == "P.Mahomes" & season == 2022 & week == 22'

# specify columns on own line for space
cols_look = [
    "passer",
    "total_line",
    "n_games",
    "pass_td_rate",
    "exp_pass_td",
    "p_0_td",
    "p_1_td",
    "p_2_td",
    "p_g2_td",
]

pbp_py_pass_td_y_geq10\
    .query(filter_by)[cols_look]
    passer  total_line  n_games  ...      p_1_td      p_2_td
p_g2_td
3295  P.Mahomes        51.0       39  ...  0.256104  0.269912
0.352483

[1 rows x 9 columns]

```

R uses more confusing names for statistical distributions. `dpois()` gives the

PMF and the d comes from density because continuous distributions (such as the normal distribution) have density rather than mass. `ppois()` gives the CDF. In R, use the `dpois()` function to give the probability of 0, 1, or 2 touchdown passes being scored in a game by a quarterback. Use the `ppois()` function to calculate the probability of more than 2 touchdown passes being scored in a game by a quarterback:

```
## R
pbp_r_pass_td_y_geq10 <-
  pbp_r_pass_td_y_geq10 |>
  mutate(
    p_0_td = dpois(x = 0,
                    lambda = exp_pass_td),
    p_1_td = dpois(x = 1,
                    lambda = exp_pass_td),
    p_2_td = dpois(x = 2,
                    lambda = exp_pass_td),
    p_g2_td = ppois(q = 2,
                     lambda = exp_pass_td,
                     lower.tail = FALSE)
  )
```

Then look at the outputs for Mahomes going into the big game from R:

```
## R
pbp_r_pass_td_y_geq10 |>
  filter(passer == "P.Mahomes", season == 2022, week == 22) |>
  select(-pass_td_y, -n_games, -n_passes,
         -passer_id, -week, -season)
# A tibble: 1 × 8
  passer      total_line pass_td_rate exp_pass_td p_0_td p_1_td p_2_td
  <chr>        <dbl>       <dbl>        <dbl>   <dbl>   <dbl>   <dbl>
1 P.Mahomes     51        2.38        2.11   0.122   0.256   0.270
  0.352
```

TIP

Notice in these examples we used the values for Mahome's λ rather than hardcoding the value. We do this for several reasons. First, it allows us to see where the number comes from rather than being a *magical mystery number* that we do not know the source for. Second, it allows the number to self-update if the data were to change. This abstraction also allows our code to more readily generalize to other situations if we need or want to re-use the code for other situations. As you can see from this example,

calculating 1 number can often be the same amount of code due to the beauty of vectorization in Python (with numpy) and R. *Vectorization* is when a computer language does a function on a vector (such as a column), rather than a single value (such as a scalar).

Okay, so you've estimated a probability of 35.2% that Mahomes would throw three or more touchdown passes, which is under the 40% you need to make a bet on the over.¹ The 64.8% that he throws two or fewer touchdown passes is slightly less than the 64.9% we need to place a bet on the under. To the astute reader, you will notice that this is why most sports bettors don't win in the long term. At least at DraftKings Sportsbook, the math tells you not to make a bet, or find a different sportsbook altogether! FanDuel, one of the other major sportsbooks in the US, actually had a different index, 1.5 touchdowns, and offered -205 on the over (67.2% breakeven) and +164 (37.9%) on the under, which again offered no value, as we made the likelihood of one or fewer touchdown passes 37.8% and two or more at 62.2%, with the under again just slightly under the breakeven probability for a bet. Luckily for those involved, Mahomes went over both 1.5 and 2.5 touchdown passes en route to the MVP. So the moral of the story is generally "in all but a few cases, don't bet".

Understanding Poisson Regression Coefficients

The coefficients from a GLM depend upon the link function, similar to the logistic regression that was covered in "[A Brief Primer on Odds Ratios](#)". By default in Python and R, the Poisson regression requires that you take the exponential to be on the same scale as the data. However, this changes the coefficients, on the link-scale, from being additive -like linear regression coefficients- to being multiplicative -like logistic regression coefficients.

To demonstrate this property, we will first use a simulation to help you better understand Poisson regression. First, consider 10 *draws* (or samples) from a Poisson distribution with a mean of 1 and save this to be object `x`. Then, look at the values for `x` using `print()` as well as the mean.

NOTE

For readers following along in both languages, the Python and R examples will usually produce different

random values (unless, by chance, the values are the same). This is because both languages use slightly different random number generators, and, even if the languages were somehow using the same identical random number generator function, the function call to generate the random numbers would be different.

In Python, use this code to generate the random numbers:

```
## Python
from scipy.stats import poisson

x = poisson.rvs(mu=1, size=10)
```

Then print the numbers:

```
## Python
print(x)
[0 2 2 0 1 0 2 1 1 0]
```

As well as their mean:

```
## Python
print(x.mean())
0.9
```

In R, use this code to generate the numbers:

```
## R
x <- rpois(n = 10, lambda = 1)
```

And then print the numbers:

```
## R
print(x)
[1] 1 0 1 1 1 0 0 3 1 3
```

And then look at their mean:

```
## R
print(mean(x))
[1] 1.1
```

Next, fit a GLM with a global intercept and look at the coefficient on the model scale and the exponential scale. In Python, use this code:

```
# Python
import statsmodels.formula.api as smf
import statsmodels.api as sm
import numpy as np
import pandas as pd

# create data frame for glm
df_py = pd.DataFrame({"x": x})

# fit GLM
glm_out_py = smf.glm(formula="x ~ 1", data=df_py,
family=sm.families.Poisson()).fit()

# Look at output on model scale
print(glm_out_py.params)

# Look at output on exponential scale
Intercept    -0.105361
dtype: float64
print(np.exp(glm_out_py.params))
Intercept     0.9
dtype: float64
```

In R, use this code:

```
## R
library(broom)

# fit GLM
glm_out_r <-
  glm(x ~ 1, family = "poisson")

# Look at output on model scale
print(tidy(glm_out_r))
# A tibble: 1 × 5
  term      estimate std.error statistic p.value
  <chr>      <dbl>     <dbl>      <dbl>    <dbl>
1 (Intercept)  0.0953     0.302     0.316    0.752
# Look at output on exponential scale
print(tidy(glm_out_r, exponentiate = TRUE))
# A tibble: 1 × 5
  term      estimate std.error statistic p.value
  <chr>      <dbl>     <dbl>      <dbl>    <dbl>
1 (Intercept)  1.10      0.302     0.316    0.752
```

Notice how the coefficient on the exponential scale (the second printed table, with `exponentiate = TRUE`) is the same as the mean of the simulated data. However, what if you have two coefficients, such as both a slope and an intercept?

As a concrete example, consider the number of touchdowns per game for the Baltimore Ravens. During this season, the Ravens' quarterback was unfortunately injured during game 13. Hence, one might reasonably expect the Ravens' number of passes to decrease over the course of the season. A formal method to test this would be to ask the question: "Does the average (or expected) number of touchdowns per game change through the season?" and then use a Poisson regression to statistically evaluate this.

To test this, first wrangle the data and then plot it to help you see what is going on with the data. You will also shift the week by subtracting 1. This allows Week 1 to be the intercept of the model. You will also set the axis ticks and change the axis labels. In Python, use this code to create [Figure 6-5](#):

```
## Python
# subset the data
bal_td_py = (
    pbp_py
    .query('posteam=="BAL" & season == 2022')
    .groupby(["game_id", "week"])
    .agg({"touchdown": ["sum"]})
)

# reformat the columns
bal_td_py.columns = list(map("_".join, bal_td_py.columns))
bal_td_py.reset_index(inplace=True)

# shift week so intercept 0 = week 1
bal_td_py["week"] = bal_td_py["week"] - 1

# create list of weeks for plot
weeks_plot = np.linspace(start=0, stop=18, num=10)
weeks_plot

# plot the data
array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14., 16., 18.])
ax = sns.regplot(data=bal_td_py, x="week", y="touchdown_sum");
ax.set_xticks(ticks = weeks_plot, labels = weeks_plot);
plt.xlabel("Week")
plt.ylabel("Touchdowns per game")
```

```
plt.show();
```

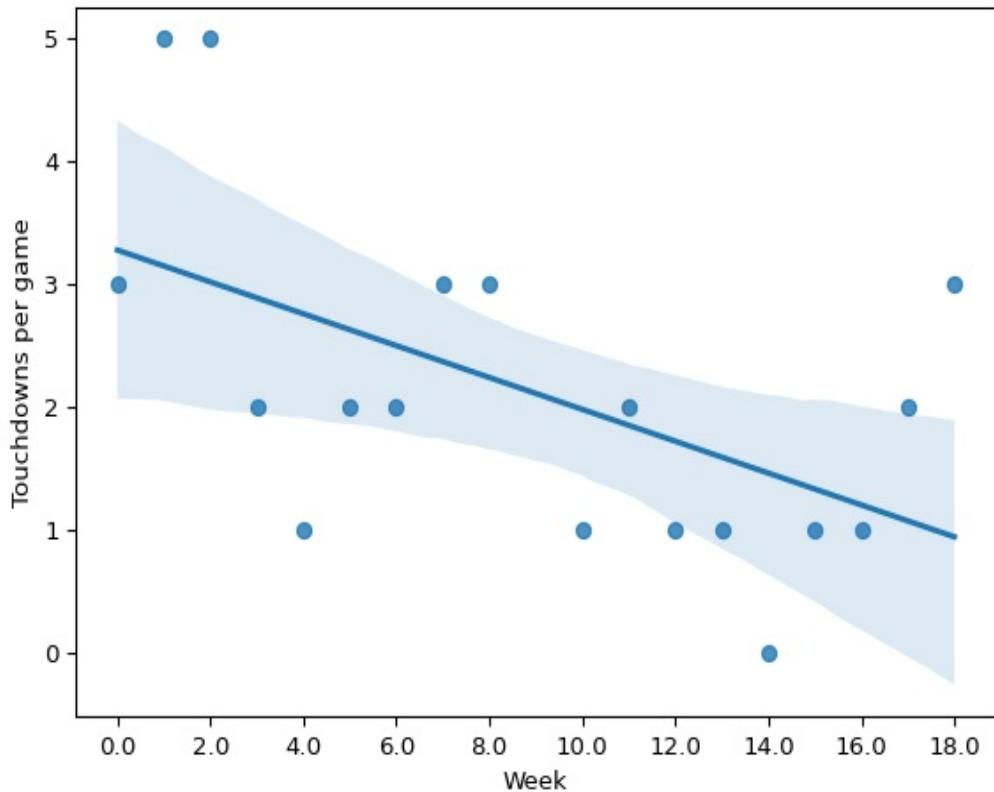


Figure 6-5. Mahomes's touchdowns per game during the 2022 season, plotted with `seaborn`. Notice the use of a linear regression trendline.

TIP

Most Python plotting tools are wrappers for the `matplotlib` package and `seaborn` is no exception. Hence, customizing `seaborn` will usually require the use of `matplotlib` commands and if you want to become an expert in plotting with Python, you will likely need to become comfortable with the clunky `matplotlib` commands.

In R, use this code to create Figure 6-6:

```
## r
bal_td_r <-
  pbp_r |>
```

```

filter(postteam == "BAL" & season == 2022) |>
group_by(game_id, week) |>
summarize(
  td_per_game =
    sum(touchdown, na.rm = TRUE),
  .groups = "drop"
) |>
mutate(week = week - 1)

ggplot(bal_td_r, aes(x = week, y = td_per_game)) +
  geom_point() +
  theme_bw() +
  stat_smooth(
    method = "glm", formula = "y ~ x",
    method.args = list(family = "poisson")
) +
  xlab("Week") +
  ylab("Touchdowns per game") +
  scale_y_continuous(breaks = seq(0, 6)) +
  scale_x_continuous(breaks = seq(1, 20, by = 2))

```

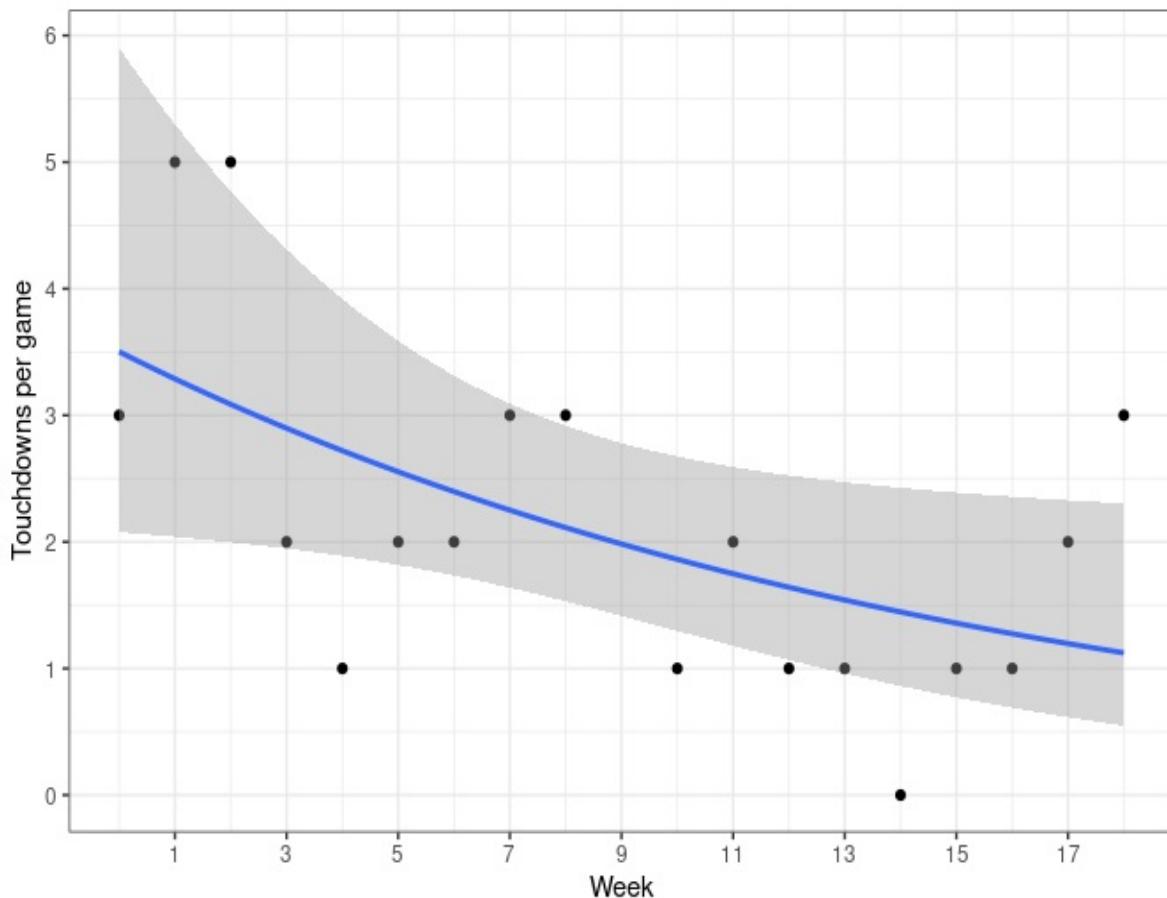


Figure 6-6. Mahomes's touchdowns per game during the 2022 season, plotted with `ggplot2`. Notice the

use of a Poisson regression trend line.

When comparing [Figure 6-5](#) and [Figure 6-6](#), notice how `ggplot2` allows you to use a Poisson regression for the trendline, whereas `seaborn` only allows a liner model. Both figures show that Baltimore, on average, had a decreasing number of touchdowns per game. Now, let's build a model and look at the coefficients. In Python:

```
## Python
glm_bal_td_py = \
    smf.glm(formula="touchdown_sum ~ week",
             data=bal_td_py,
             family=sm.families.Poisson())\
    .fit()
```

Then, look at the coefficients on the link (or log) scale:

```
## Python
print(glm_bal_td_py.params)
Intercept      1.253350
week          -0.063162
dtype: float64
```

Followed by the exponential (or data) scale:

```
## Python
print(np.exp(glm_bal_td_py.params))
Intercept      3.502055
week          0.938791
dtype: float64
```

Or, use R to fit the model:

```
## R
glm_bal_td_r <-
  glm(td_per_game ~ week,
      data = bal_td_r,
      family = "poisson"
  )
```

Then, look at the coefficient on the link (or log) scale:

```
## R
```

```

print(tidy(glm_bal_td_r))
# A tibble: 2 × 5
  term      estimate std.error statistic   p.value
  <chr>     <dbl>     <dbl>     <dbl>     <dbl>
1 (Intercept) 1.25      0.267      4.70  0.00000260
2 week       -0.0632    0.0300     -2.10  0.0353

```

Followed by the exponential (or data) scale:

```

## R
print(tidy(glm_bal_td_r, exponentiate = TRUE))
# A tibble: 2 × 5
  term      estimate std.error statistic   p.value
  <chr>     <dbl>     <dbl>     <dbl>     <dbl>
1 (Intercept) 3.50      0.267      4.70  0.00000260
2 week       0.939     0.0300     -2.10  0.0353

```

Now, let's look the meaning of the coefficients. First, examining the link-scale value of intercept, **1.253**, and slope, **-0.063**, does not appear to provide much context. However, the slope term is also known as the the *risk ratio* or *relative risk* for each week. After exponentiating the coefficients, you get an intercept value of **3.502** and a slope value of **0.939**.

These numbers do have easy-to-understand value. The intercept is the number of expected passes during the first game (this is why we had you subtract 1 from week, so that Week 1 would be 0 and the intercept) and the value is **3.502**.

Looking at [Figure 6-5](#) and [Figure 6-6](#), this seems reasonable because Week 0 was 3 and Weeks 1 and 2 were 5.

NOTE

Using multiplication on the log scale is the same as using addition on non-transformed numbers. For example, $x + y = \log(e^x \times e^y)$. To demonstrate with real numbers consider $1 + 3 = 4$. And (ignoring rounding errors), $e^1 = 2.718$ and $e^3 = 20.09$. Using these results, $2.718 \times 20.09 = 54.60$. Lastly, back transforming gives you: $\log(54.60) = 4.00$.

Then, for the next week, the expected number of touchdowns would be $3.502 \times \$0.939 = 3.288378$. For the following week, $3.502 \times \$0.939 \times \$0.939 = 3.071678$. Hence, the equation to estimate the expected number of

touchdowns for a given week during the 2022 season for the Baltimore Ravens would be $3.502 \$ \$ 0.939^{\text{week}}$ on the log scale. Or this could be re-written as $\exp(1.253 + -0.063 \$ \$ \text{week})$.

Although we had to cherry pick this example to explain a Poisson regression, hopefully the example served its purpose and helps you see how to interpret Poisson regression terms.

Closing Thoughts on GLMS

This chapter and [Chapter 5](#) have introduced you to GLMs and how the models can help you both understand and predict with data. In fact, the simple predictions done in these chapters would help you also understand the basics of simple machine learning methods. When working with GLMs, notice both example models forced you to deal with the link function because the coefficients are estimated on a different scale compared to the data. Additionally, you saw two different types of error families, the binomial in [Chapter 5](#) and the Poisson in this chapter.

In the Exercises in this chapter, we have you look at the quasi-Poisson to account for dispersion when there are either too many or too few zeros compared to what would be expected. We also have you look at the negative-binomial as another alternative to Poisson for modeling count data. Two other common types of GLMs included the gamma regression and the lognormal regression. The gamma and lognormal are similar. The lognormal might be better for some data that is positive, but right skewed such as pass data.

An alphabet soup of extensions exist for linear models and GLMS. Hierarchical, multi-level, or random-effect models allow for features such as repeated observations on the same individual or group to model uncertainty. Generalized additive models (GAMs) allow for curves, rather than straight lines, to be fit for models. Generalized additive mixed-effect models (GAMMs) merge GAMs and random-effect models. Likewise, tools such as model selection can help you determine what type of model to use.

Basically, linear models have many different options because of their long history and use in statistics and different scientific fields. We have a colleague

who can, and would, talk about the assumptions of regressions for hours on end. Likewise, year-long graduate-level courses exist on the different models we have covered over the course a few chapters. However, even simply knowing the basics of regression models will greatly help you to up your football analytics game.

Data Science Tools Used in This Chapter

This chapter including the following topics:

- You saw how to fit a Poisson regression in Python and R using `glm()`.
- You learned how to understand and read the coefficients from a Poisson regression including relative risk.
- You saw how to connect to datasets using `merge()` in Python or an `inner_join()` in R.
- You reapplied data wrangling tools you learned in previous chapters.
- You learned about betting odds.

Exercises

1. What happens in the model for touchdown passes if you don't include the total for the game? Does it change any of the probabilities enough to recommend a bet for Patrick Mahomes' touchdown passes in the Super Bowl?
2. Repeat the work in this chapter for interceptions thrown. Examine Patrick Mahomes' interception prop in Super Bowl LVII, which was 0.5 interceptions, with over priced at -120 and under priced at -110.
3. What about Jalen Hurts, the other quarterback in Super Bowl LVII, whose touchdown prop was 1.5 (-115 to the over and -115 to the under) and interception prop was 0.5 (+105/-135)? Was there value in betting either of those markets?

4. Look through the `nflfastR` dataset for additional features to add to the models for both touchdowns and interceptions. Does pre-game weather affect either total? Does the size of the point spread?
5. Repeat this chapter's GLMS, but rather than using a Poisson distribution, use a quasi-Poisson that estimates that estimates a dispersion parameters.
6. Repeat this chapter's GLMS, but rather than using Poisson distribution, use a negative binomial.

Suggested Readings

The resources suggested in “Suggested Readings”, “Suggested Readings”, and “Suggested Readings” will help you understand generalized linear models such as Poisson regression more. One last regression book you may find helpful would be: *Applied Generalized Linear Models and Multilevel Models in R* by Paul Roback and Julie Legler (CRC Press, 2021). This book provides an accessible introduction for people wanting to learn more about regression analysis and advanced tools such as GLMs.

To learn more about the application of probability to both betting and the broader world, check out the following resources:

- *Logic of Sports Betting* by Ed Miller and Matthew Davidow (Self published, 2019) provides details about how betting works for those wanting more details.
- *Wisdom of the Crowds* by James Surowiecki (Anchor, 2005) describes how “the market” or collective guess of the crowds can predict outcomes.
- *Sharp Sports Betting* by Stanford Wong (Huntington Press, 2021) will help readers understand betting more and provides an introduction to the topic.
- *Sharper, a Guide to Modern Sports Betting* by True Pokerjo (Self published 2016) talks about sports betting for those seeking insight.
- *The Foundations of Statistics* by Leonard Jimmie Savage (John Wiley and Sons 1954; Dover Press 1972 reprinting) is a classical statistical text on how to use statistics to make decisions. The first half of the book does a

good job explaining statistics in real world application. The second becomes theoretical for those seeking a mathematically rigorous foundation (Richard skimmed and then skipped much of the second half).

- **The Good Judgment Project** seeks to apply the “wisdom of the crowds” to predict real world problems. Initially funded by the US Intelligence community, this project’s “forecasts were so accurate that they even outperformed intelligence analysts with access to classified data.”

¹ All betting odds in this chapter came from [Betstamp](#).

Chapter 7. Webscraping: Obtaining and Analyzing Draft Picks

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

One of the great triumphs in public analysis of American football is the `nflscrapR` and, after that, the `nflfastR` package. These packages allow for easy analysis of the game we all love. Including data into your computing space is often as simple as downloading a package in Python or R, and away you go.

Sometimes it’s not that easy, though. Often there are situations where you need to *scrape* data off of the web yourself (that is to say, use a computer program to download your data). While it is beyond the scope of this book to teach you all of webscraping in Python and R, there are some pretty easy commands that can get you a significant amount of data to analyze.

In this chapter, you are going to scrape NFL Draft and NFL Scouting Combine data from [Pro Football Reference](#), a website we mentioned in passing in [Chapter 6](#). It’s a wonderful resource out of Philadelphia, PA that provides free data for every sport imaginable. You will use the website to get data for the NFL Draft and NFL Scouting Combine.

The NFL Draft is a yearly event held in various cities around the country. In the draft, teams select from a pool of players that have completed at least three post-high school years. While there used to be more rounds, the NFL draft currently consists of seven rounds. The draft order in each round is determined by how well each team played the year before. Weaker teams pick higher in the draft than the stronger teams. Teams can trade draft picks for other draft picks or players.

The NFL Scouting Combine is a yearly event held in Indianapolis, IN. In the combine, a pool of athletes eligible for the NFL Draft meet with evaluators from NFL teams to perform various physical and psychological tests. Additionally, this is generally thought of as the NFL's yearly convention, where deals between teams and agents are originated and, sometimes, finalized.

The combination of these two data sets are a great resource for beginners in the football analytics space for a couple of reasons. First, the data is collected over a small set of days once a year and does not change thereafter. Although some players may re-test physically at a later date, and players can often leave the team that drafted them for a number of reasons, the draft teams cannot change. Thus, once you obtain the data, it's generally good to use for almost an entire calendar year, after which you can simply add the new data when it's obtained the following year.

You will start by scraping all combine and draft data from 2022, and then fold in later years for analysis.

TIP

Web scraping is a lot of trial and error, especially when you're getting started. In general, we find an example that works and then change one piece at a time until we get what we need.

for Loops

We are going to cover a fundamental programming skill before we start web scraping: **for** loops. We mentioned **for** loops in “[Individual Player Markets and Modeling](#)” and will expand upon them in this section.

TIP

We use `for` loops in the book because they are conceptually simple to use and understand. Other tools exist such as `map()` in Python or *apply* functions in R such as `lappy()`, or `apply()`. These functions are quicker, easier for advanced users to understand and read, and often less error prone. However, we wrote a book about introductory football analytics, not advanced data science programming. Hence, we usually stick to `for` loops for this book. See resources such [Chapter 9](#) in Hadley Wickham's *Advanced R*, 2nd Edition (CRC Press, 2019) that describe these methods and why to use them.

Often when programming, you want to repeat code. You can use tools such as `for` loops for this task. The simplest `for` loops in a language print the index of the loop. The loop takes an index, commonly the character `i`, and goes over a range of values.

In Python, this would be:

```
##Python
for i in range(10):
    print(i)
```

In R, this would be:

```
## R
for (i in seq(1, 10)) {
    print(i)
}
```

In this case, notice how Python's loop does not need curly brackets (`{}`) around the code. Instead, Python uses white space (the spaces) to show the loop. Design choices like this are one reason some people consider Python to be a more elegant language than R. For example, you could write `for (i in seq(1, 10)){ print(i) }` on one line in R. However, this one line of code is harder to read.

TIP

Regardless of your computer language, whitespaces (spaces, line breaks, and the like) are your friend because they make your code easier to read.

Web Scraping with Python

TIP

Before you start web scraping, go to the webpage first so you can see what you are trying to download.

The following code allows us to scrape with Python. Save the *Uniform Resource Locator* (more commonly known as *URL* or web address) to an object, `url`. In this case, the URL is simply the URL for the 2022 NFL draft. Next, use `read_html` from the `pandas` package to simply read in tables from the given URL. Remember that Python starts counting with 0. Thus, the zeroth element of the dataframe, `draft_py` from `read_html()` is simply the first table on the web page. You will also need to change NA draft approximate values to be 0:

```
## Python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import statsmodels.formula.api as smf
import numpy as np

url = "https://www.pro-football-reference.com/years/2022/draft.htm"

draft_py.loc[draft_py["DrAV"].isnull(), "DrAV"] = 0
draft_py = pd.read_html(url, header=1)[0]
```

You can peek at the data using `print()`:

```
## Python
print(draft_py)
   Unnamed: 0  Rnd  Pick    Tm ...     Sk College/Univ      Unnamed:
28  Season
0          0    1     1  CLE ...  19.0      Penn St.  College
Stats  2000
1          1    1     2  WAS ...  23.5      Penn St.  College
Stats  2000
2          2    1     3  WAS ...    NaN      Alabama  College
Stats  2000
3          3    1     4  CIN ...    NaN  Florida St.  College
Stats  2000
4          4    1     5  BAL ...    NaN  Tennessee  College
```

Stats	2000
5866	5866	7	258	GNB	...	NaN		Nebraska	College		
Stats	2022										
5867	5867	7	259	KAN	...	NaN		Marshall	College		
Stats	2022										
5868	5868	7	260	LAC	...	NaN		Purdue	College		
Stats	2022										
5869	5869	7	261	LAR	...	NaN	Michigan St.	College			
Stats	2022										
5870	5870	7	262	SFO	...	NaN		Iowa St.	College		
Stats	2022										

[5871 rows x 31 columns]

WARNING

When webscraping be careful not to *hit* or pull from websites too many times. You may find yourself locked out of websites. If this occurs, you will need to wait a while until you try again. Additionally, many websites have rules (or, more formally called, *Terms & Conditions*) that provide guidance on if and how you scrap their pages.

Although kind of ugly, this is workable! To scrape multiple years, for example 2000 to 2022, you can do a simple **for** loop - which is often possible due to systematic changes in the data. Experimentation is key.

TIP

To avoid multiple pulls from web pages, we cached files when writing the book and only downloaded files when needed. For example, we used (and hid from you, the reader) this code earlier in this chapter:

```
## Python
import pandas as pd
import os.path

file_name = "draft_demo_py.csv"

if not os.path.isfile(file_name):
    ## Python
    url = \
        "https://www.pro-football-reference.com/years/" + \
        "2022/draft.htm"
    draft_py = pd.read_html(url, header=1)[0]

    conditions = [
```

```

        (draft_py.Tm == "SDG"),
        (draft_py.Tm == "OAK"),
        (draft_py.Tm == "STL"),
    ]
choices = ["LAC", "LV", "LAR"]

draft_py["Tm"] = \
    np.select(conditions, choices, default = draft_py.Tm)
draft_py.loc[draft_py["DrAV"].isnull(), "DrAV"] = 0
draft_py.to_csv(file_name)
else:
    draft_py = pd.read_csv(file_name)
    draft_py.loc[draft_py["DrAV"].isnull(), "DrAV"] = 0

```

Also, when creating our own `for` loops, we often start with a simple index value (for example, set `i = 1`) and then make our code work. After making our code work, we add in the the `for ...` line to run the code over many different values.

WARNING

When setting the index value to a value such as `1` while building `for` loops, make sure you remove the place holder index (such as `i = 1`) from your code. Otherwise, your loop will simply run over the same functions or data multiple times.

Now, let's download some more data in Python. As part of this process, you need to clean up the data. This includes telling pandas which row has the header. In this case, the 2nd row, or `header=1`. Recall that Python starts counting with `0` so `1` corresponds to the second entry. Likewise, you need to save the `season` as part of the loop to its own columns. Also, remove rows that contain extra heading information (strangely, some rows in the dataset are duplicates of the data's header) by only saving rows where a value in the row is not equal to the column's name (for example, use `tm != "Tm"`):

```

## Python
draft_py = pd.DataFrame()
for i in range(2000, 2022 + 1):
    url = "https://www.pro-football-reference.com/years/" + \
        str(i) + \
        "/draft.htm"

```

```

web_data = pd.read_html(url, header=1)[0]
web_data["Season"] = i
web_data = web_data.query('Tm != "Tm"')
draft_py = pd.concat([draft_py, web_data])

draft_py.reset_index(drop=True, inplace=True)

```

Some teams moved, because of these, the team names (*Tm*) need to be changed to reflect the new names. `np.select()` can be used for this, using a `conditions` that has the teams old names, and a `choices` that has the new names. The default in `np.select()` also need to be changed so that non-moved teams stay the same:

```

## Python
# the Chargers moved to Los Angeles from San Diego
# the Raiders moved from Oakland to Las Vegas
# the Rams moved from St. Louis to Los Angeles
conditions = [
    (draft_py.Tm == "SDG"),
    (draft_py.Tm == "OAK"),
    (draft_py.Tm == "STL"),
]
choices = ["LAC", "LVR", "LAR"]

draft_py["Tm"] = \
    np.select(conditions, choices, default = draft_py.Tm)

```

Finally, replace missing draft approximate values with 0 before you reset the index before saving the file:

```

## Python
draft_py.loc[draft_py["DrAV"].isnull(), "DrAV"] = 0
draft_py.to_csv("data_py.csv", index=False)

```

Now, you can peek at the data:

```

## Python
print(draft_py.head())
   Unnamed: 0  Rnd  Pick    Tm  ...     Sk College/Univ      Unnamed: 28
Season
0          0    1     1  CLE  ...  19.0      Penn St.  College Stats
2000
1          1    1     2  WAS  ...  23.5      Penn St.  College Stats
2000

```

```

2           2   1    3   WAS   ...   NaN      Alabama  College Stats
2000
3           3   1    4   CIN   ...   NaN  Florida St.  College Stats
2000
4           4   1    5   BAL   ...   NaN  Tennessee  College Stats
2000

[5 rows x 31 columns]

```

Let's look at the other columns available to us:

```

## Python
print(draft_py.columns)
Index(['Unnamed: 0', 'Rnd', 'Pick', 'Tm', 'Player', 'Pos', 'Age',
       'To', 'AP1',
       'PB', 'St', 'wAV', 'DrAV', 'G', 'Cmp', 'Att', 'Yds', 'TD',
       'Int',
       'Att.1', 'Yds.1', 'TD.1', 'Rec', 'Yds.2', 'TD.2', 'Solo',
       'Int.1', 'Sk',
       'College/Univ', 'Unnamed: 28', 'Season'],
      dtype='object')

```

TIP

With R and Python, we usually often need to tell the computer to save our updates. Hence, we often save objects over the same name, such as `draft_py = draft_py.drop(labels = 0, axis = 0)`. In general, understand this copy behavior so that you do not delete data you want or need later.

Finally, here's the metadata (or *data dictionary*, that is to say, data about data) for the data you will care about for the purposes of this analysis:

- The season in which the player was drafted (**Season**)
- Which selection number they were taken at (**Pick**)
- The player's drafting team (**Tm**)
- The player's name (**Player**)
- The player's position (**Pos**)
- The player's whole career approximate value (**wAv**)

- The player's approximate value for drafting team (DrAV)

Lastly, you might want to re-order and select on certain columns. For example, you might only want six columns and also to change their order:

```
## Python
draft_py_use = \
    draft_py[["Season", "Pick", "Tm", "Player", "Pos", "wAV", "DrAV"]]

print(draft_py_use)
   Season  Pick   Tm        Player  Pos    wAV  DrAV
0     2000     1  CLE  Courtney Brown  DE  27.0  21.0
1     2000     2  WAS   LaVar Arrington  LB  46.0  45.0
2     2000     3  WAS    Chris Samuels   T  63.0  63.0
3     2000     4  CIN    Peter Warrick  WR  27.0  25.0
4     2000     5  BAL    Jamal Lewis  RB  69.0  53.0
...
5866   2022   258  GNB    Samori Toure  WR   1.0   1.0
5867   2022   259  KAN   Nazeeh Johnson  SAF   1.0   1.0
5868   2022   260  LAC    Zander Horvath  RB   0.0   0.0
5869   2022   261  LAR      AJ Arcuri  OT   1.0   1.0
5870   2022   262  SF0    Brock Purdy  QB   6.0   6.0

[5871 rows x 7 columns]
```

You'll generally want to save this data locally for future use. That is to say, you do not want to download and clean data each time you use it.

WARNING

The data you obtain from webscraping may be different from our example data. For example, a technical reviewer had the *draft pick* column be treated as a discrete character rather than a continuous integer or numeric. If your data seems strange (such as plots that seem off), examine your data using tools from [Chapter 2](#) and other chapters to check your data types.

Webscraping in R

WARNING

Some Python and R packages require outside dependencies, especially on macOS and Linux. If you try to install a package and get an error message, try reading the error message. Often, we find the error

messages to be cryptic, so we end up using a search engine to try our debugging.

You can use the `rvest` package to do a similar loops in R. First, load the package and create an empty `tibble`:

```
## R
library(janitor)
library(tidyverse)
library(rvest)
library(htmlTable)
library(zoo)

draft_r <- tibble()
```

Then, loop over the years 2000 to 2023. Ranges can be specified using a colon, such as `2000:2022`. However, we prefer to explicitly use the `seq()` command because it is more robust.

A key difference of the R code compared to the Python code is that the `html_nodes` command is called with the pipe. The code also needs to extract the web dataframe (`web_df`) from the raw `web_data`. The `row_to_names()` function cleans up for the empty header row and replaces the data's header with the first row. `janitor::clean_names()` cleans up the column names because some columns have duplicate names. `mutate()` saves the season in the loop.

Next, use `filter()` with the data to remove any rows that contain duplicate headers as extra rows:

```
## R
library(tidyverse)
library(rvest)
library(htmlTable)
library(zoo)

for (i in seq(from = 2000, to = 2022)) {
  url <- paste0(
    "https://www.pro-football-reference.com/years/",
    i,
    "/draft.htm"
  )
```

```

web_data <-
  read_html(url) |>
  html_nodes(xpath = '//*[@id="drafts"]') |>
  html_table()
web_df <-
  web_data[[1]]
web_df_clean <-
  web_df |>
  janitor::row_to_names(row_number = 1) |>
  janitor::clean_names(case = "none") |>
  mutate(Season = i) |> # add seasons
  filter(Tm != "Tm") # Remove any extra column headers

draft_r <-
  bind_rows(
    draft_r,
    web_df_clean
  )
}

```

Rename teams (T_m) to reflect teams that have moved using `case_when()` before saving the output so you do now need to download it again:

```

## R
# the chargers moved to Los Angeles from San Diego
# the Raiders moved from Oakland to Las Vegas
# the Rams moved from St. Louis to Los Angeles
draft_r <-
  draft_r |>
  mutate(Tm = case_when(Tm == "SDG" ~ "LAC",
                        Tm == "OAK" ~ "LVR",
                        Tm == "STL" ~ "LAR",
                        TRUE ~ Tm),
         DrAV = ifelse(is.na(DrAV), 0, DrAV))
write_csv(draft_r, "draft_data_r.csv")

```

Now that you have data, subset to grab the data you'll need for the analysis later:

```

## R
draft_r_use <-
  draft_r |>
  select(Season, Pick, Tm, Player, Pos, wAV, DrAV)

print(draft_r_use)
# A tibble: 5,871 × 7
  Season   Pick   Tm     Player   Pos     wAV   DrAV
  <dbl> <dbl> <chr> <chr> <chr> <dbl> <dbl>
1       1     1  ARI    D. Johnson  1.00  -0.05  0.00
2       1     2  ARI    D. Johnson  1.00  -0.05  0.00
3       1     3  ARI    D. Johnson  1.00  -0.05  0.00
4       1     4  ARI    D. Johnson  1.00  -0.05  0.00
5       1     5  ARI    D. Johnson  1.00  -0.05  0.00

```

1	2000	1	CLE	Courtney Brown	DE	27	21
2	2000	2	WAS	LaVar Arrington	LB	46	45
3	2000	3	WAS	Chris Samuels	T	63	63
4	2000	4	CIN	Peter Warrick	WR	27	25
5	2000	5	BAL	Jamal Lewis	RB	69	53
6	2000	6	PHI	Corey Simon	DT	45	41
7	2000	7	ARI	Thomas Jones	RB	62	7
8	2000	8	PIT	Plaxico Burress	WR	70	34
9	2000	9	CHI	Brian Urlacher HOF	LB	119	119
10	2000	10	BAL	Travis Taylor	WR	30	23

i 5,861 more rows

NOTE

Compare the two web scraping methods. Python functions tend to be more self-contained and tend to belong to an object (and, functions that belong to an object in Python are called *methods*). In contrast, R tends to use multiple functions on the same object. This is a design trait of the languages. Python is a more object-oriented language, whereas R is a more functional language. Which style do you like better?

One thing to notice here that won't affect the analysis in this chapter, but will if you want to move forward with the data. Notice that the ninth pick in the 2000 NFL Draft has the name *Brian Urlacher HOF* with the *HOF* part denoting that he eventually made the National Football League Hall of Fame. If you want to use this data and merge it with another data set, you will have to alter the names to make sure that things like that are taken out.

Analyzing the NFL Draft

The NFL Draft occurs annually and allows teams to select eligible players. The event started in 1936 as a method to allow all teams to remain competitive and obtain talented players. During the draft, each team gets one pick per round. The order for each round is based upon a team's record with tie-breaking rules for teams with the same record. Thus, the team that won the Super Bowl picks last and the team that lost the Super Bowl picks second-to-last. However, teams often trade players and include draft picks as part of their trades. Hence picks can have extra value that people want to quantify and understand.

There are a number of cool questions you can ask about the draft, especially if you're drafting players either for a fantasy team or a real team. How much is

each draft pick worth (and in what denomination)? Are some teams better at drafting players than others? Are some positions better gambles than others in the draft?

At first blush, the most straightforward question to answer with this data is the first question (“How much is each draft pick worth (and in what denomination)?”), which is assigning a value to each draft pick. The reason this is important is that teams will often trade picks to each other in an effort to align the utility of their draft picks with the team’s current needs.

For example, in 2018 the New York Jets, who were currently holding the sixth-overall pick in the draft, traded that pick, along with the 37th and 49th pick in the 2018 draft, along with a second-round pick in the 2019 draft, to the Indianapolis Colts for the third pick in the 2018. Top draft picks are generally reserved for quarterbacks, and the Jets, after losing out on the Kirk Cousins sweepstakes in free agency, needed a quarterback. The Colts “earned” the third-overall pick in the 2018 in large part because they struggled in 2017 with their franchise quarterback, Andrew Luck, who was on the mend with a shoulder injury. In other words the third-overall pick had less utility to the Colts than it held for the Jets, so the teams made the trade.

How do the teams decide what is a “fair” market value for these picks? You have to go back to 1989, when the Dallas Cowboys, after being bought by Arkansas oil millionaire Jerry Jones, jettisoned long-time Hall of Fame head coach Tom Landry and replaced him with a college coach, Jimmy Johnson. The Cowboys were coming off of a three-win season in 1988, and their roster was relatively bare when it came to difference-making players. As the legend goes, Johnson, while on a jog, decided to trade his star running back, and future United States senatorial candidate, Herschel Walker, for a package that ended up being three first-round picks, three-second round picks, a third-round pick, a sixth-round pick and a few players.

The Cowboys would finish 1989 with the NFL’s worst record at 1-15, which improved the position of their remaining, natural draft picks (they used their 1990 first-rounder early on a quarterback in the supplemental draft). Johnson would go on to make more draft pick trades during the early years of his tenure than any other coach or executive in the game, using a value chart made by Mike McCoy, which is now known as the “Jimmy Johnson chart”, designed to match

the value of each pick in each round. The chart assigns the first-overall pick 3000 “points”, with the value of each subsequent pick falling off exponentially.

The Jimmy Johnson chart is still the chart of choice for many NFL teams, but has been shown to overvalue top picks in the draft relative to subsequent picks. In fact, when considering the cost of signing each pick, Nobel Prize-winning economist and sports analytics legend Cade Massey showed in the paper [“The Losers Curse: Decision Making and Market Efficiency in the NFL Draft”](#) that the surplus value of the first pick, that is, the on-field value of the pick relative to the salary earned by the player picked, is not maximized by the first pick, but rather by picks either in the middle or end of the first round picks or early in the second round picks.

Much commentary exists on this topic, some of it useful, and some of it interactive via code. For example, Ben Baldwin [blogs on Open Source Football](#) and shares code that allows people to create their own draft value curve. Eric’s former colleague at PFF, Timo Riske, wrote an [article](#) about “the surplus value of draft picks”. Additionally, others have done research on this topic including [Michael Lopez](#) and [Ben Baldwin](#) (the same Ben Baldwin that helped create [nflfastR](#)) and have produced even shallower draft curves.

Here we aim to have you reproduce that research using the amount of approximate value generated by each player picked for his drafting team or *draft approximate value (drAV)*. Plotting this, you can clearly see that there is some efficiency in the draft, that teams get better at picking the earlier in the draft (there is some bias here, in that teams also play their high draft picks more, but you can show per-play efficiency drops off with draft slot as well).

First, select years prior to 2019. The reason to filter out years after 2019 is that those players are still playing through their rookie contracts, and haven’t yet had a chance to sign freely with their next team.

Then plot the data to compare pick to average draft value. In Python, use this code to create [Figure 7-1](#):

```
## Python
# Change theme for chapter
sns.set_theme(style="whitegrid", palette="colorblind")

draft_py_use_pre2019 = \
```

```

draft_py_use\

.query("Season <= 2019")

## format columns as numeric or integers
draft_py_use_pre2019 = \
    draft_py_use_pre2019\

.astype({"Pick": int, "DrAV": float})

sns.regplot(data=draft_py_use_pre2019,
             x="Pick",
             y="DrAV",
             line_kws={"color": "red"},
             scatter_kws={'alpha':0.2});

plt.show();

```

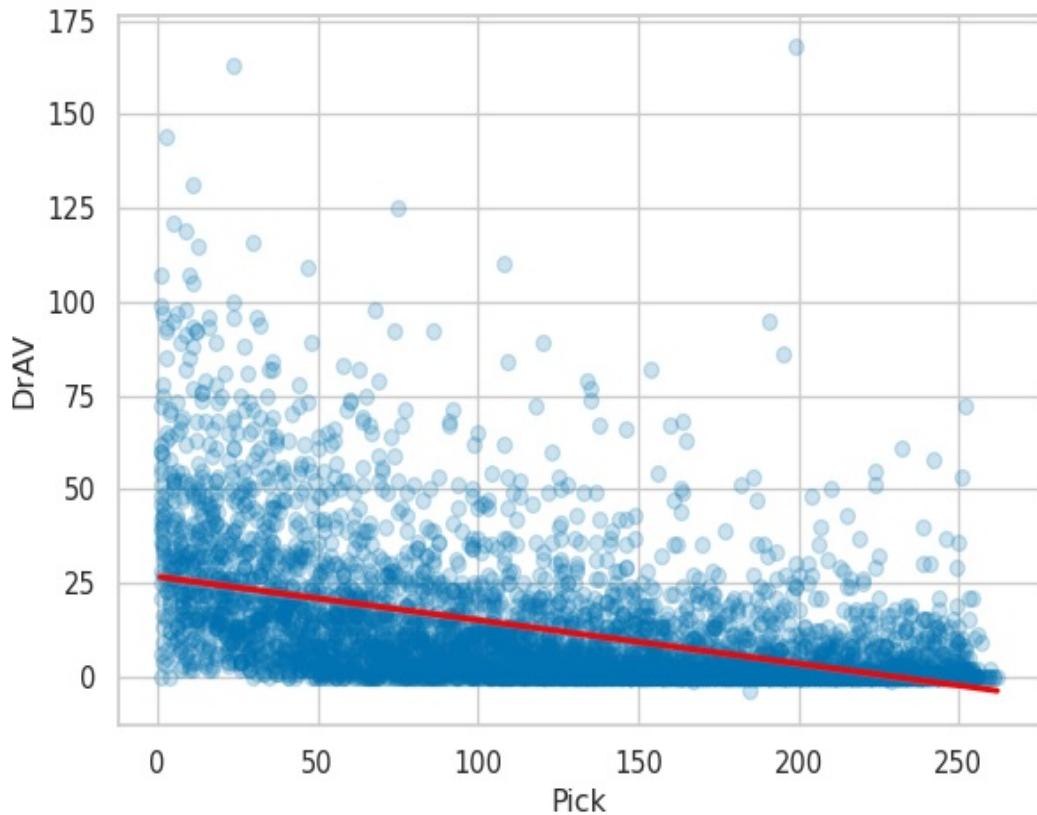


Figure 7-1. Scatterplot with a linear trendline for draft pick number against draft approximate value, plotted with seaborn.

In R, use this code to create **Figure 7-2**:

```

## R
draft_r_use_pre2019 <-

```

```

draft_r_use |>
  mutate(DrAV = as.numeric(DrAV),
         wAV = as.numeric(wAV),
         Pick = as.integer(Pick)) |>
  filter(Season <= 2019)

ggplot(draft_r_use_pre2019, aes(Pick, DrAV)) +
  geom_point(alpha = 0.2) +
  stat_smooth() +
  theme_bw()

```

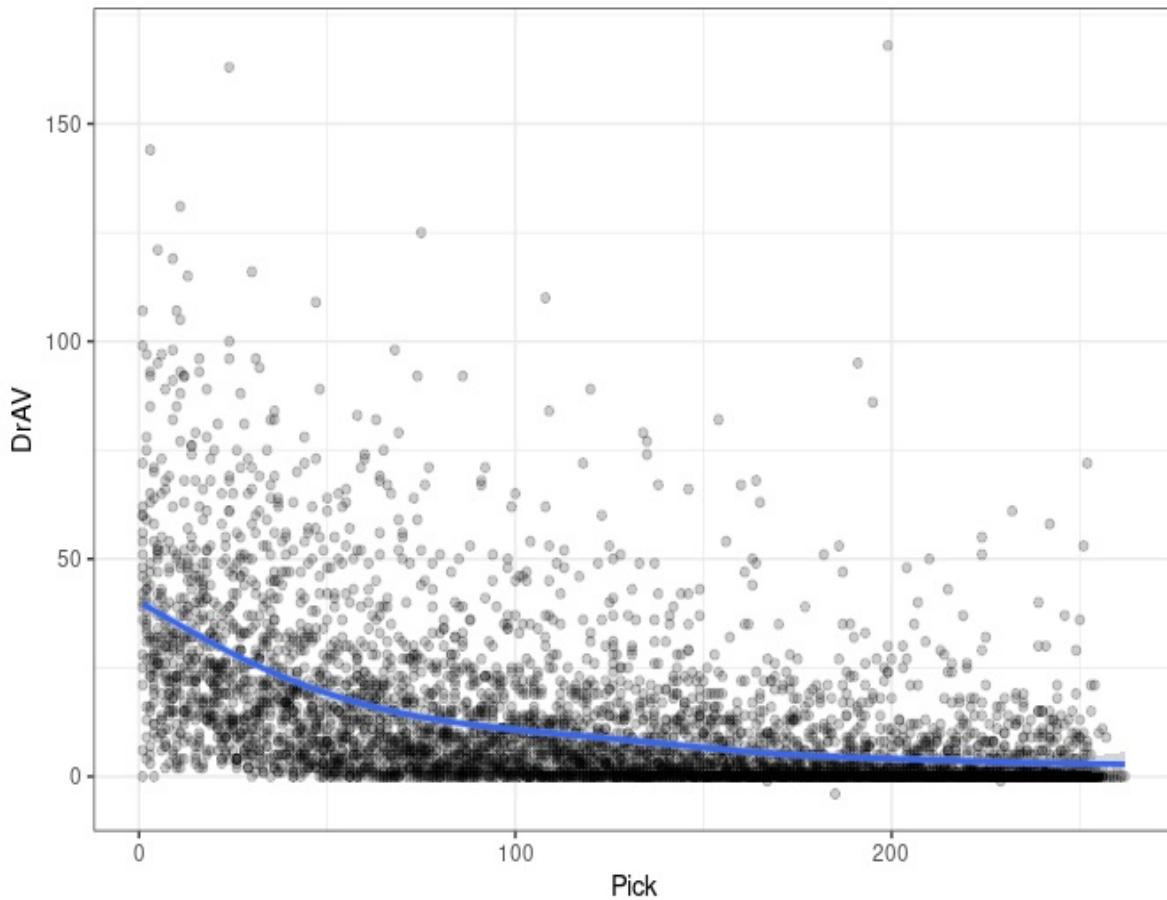


Figure 7-2. Scatterplot with a smoothed, spline trendline for draft pick number against draft approximate value, plotted with ggplot2.

Figure 7-1 and **Figure 7-2** show that the value of a pick decreases as the pick number increases.

Now, a real question when trying to derive this curve is: what are teams looking for when they draft? Are they looking for the average value produced by the pick? Are they looking for the median value produced by the pick? Some other

percentile?

The median is likely going to give zero values for later picks, which is clearly not true since teams trade them all of the time, but the mean might also overvalue them due to the hits later in the draft by some teams. Statistically, the median is zero if at least 50% of picks had zero value. But, the mean can be affected by some really good players who were drafted late. The Patriots, for example, took Tom Brady with the 199th pick in the 2000 draft and he became one of the best football player of all time. A Business Insider [article](#) by Cork Gaines tells the backstory of this pick. For now, use the mean, but in the exercises you'll use the median and see if anything changes. “[Quantile Regression](#)” provides a brief overview of another type of regression, quantile regression, that might be worth exploring if you want to dive into different model types.

NOTE

Datasets that contain series (such as daily temperatures or draft picks) often contain both patterns and noise. One way to smooth out this noise is by calculating an average over several sequential observations. This average is often called a *rolling average* with other names including *moving mean*, *running average* or similar variations with *rolling*, *moving*, or *running* used to describe mean or average. Key inputs to a rolling average include the *window* (number of inputs to use), *method* (such as mean or median), and what to do with the start and end of the series (for example, should the first entries without a full window be dropped or another rule used?).

To smooth it out, first calculate the average value for each pick. A couple of the lower picks had NaN values, so replace these values with 0. Then, calculate the six-pick moving mean DrAV surrounding each pick's average value (that is to say, each DrAV value as well as the 6 before and 6 after for a window of 13). Also, tell the `rolling()` function to use a `min_periods=1` and to center the mean (that is to say, the rolling average is centered on the current DrAV). Last, `groupby()` pick and then calculate the average DrAV for each pick position. In Python, use this code:

```
## Python
draft_chart_py = \
    draft_py_use_pre2019\
    .groupby(["Pick"])\
```

```
.agg({"DrAV": ["mean"]})  
  
draft_chart_py.columns = \  
    list(map("_".join, draft_chart_py.columns))  
  
draft_chart_py.loc[draft_chart_py.DrAV_mean.isnull()] = 0  
  
draft_chart_py["roll_DrAV"] = (  
    draft_chart_py["DrAV_mean"]  
    .rolling(window=13, min_periods=1, center=True)  
    .mean()  
)
```

TIP

For the exercise in Python, you will want to change the `rolling()` `mean()` to be other functions, such as `median()`, not the `groupby()` `mean()` that is part of the `agg()`.

Then plot the results to create [Figure 7-3](#):

```
## Python  
sns.scatterplot(draft_chart_py, x="Pick", y="roll_DrAV")  
plt.show()
```

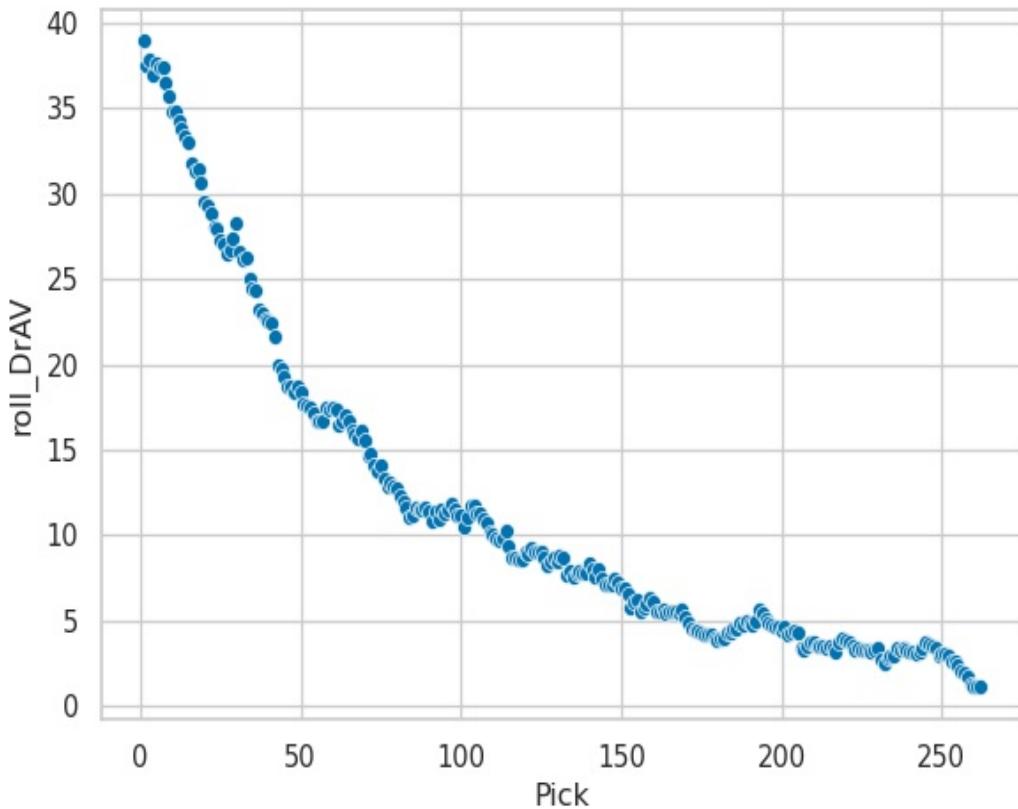


Figure 7-3. Scatterplot for draft pick number against draft approximate value, plotted with seaborn.

In R, `group_by()` pick and `summarize()` with the `mean()`. Replace `na` values with 0. Then, use the `rollapply()` function from the `zoo` package (make sure you have run `library(zoo)` and installed the package, since this is the first time you've used this package in the book). With `rollapply()`, use a `width = 13`, and the `mean` function (`FUN = mean`).

Tell the `mean` function to ignore `na` values with `na.rm = TRUE` and fill in missing values with `NA`, `center` the mean (that is the rolling average is centered on the current DrAV), and calculate the mean when less than 13 observations are present (such as the start and end of the dataframe):

```
## R
draft_chart_r <-
  draft_r_use_pre2019 |>
  group_by(Pick) |>
  summarize(mean_DrAV = mean(DrAV, na.rm = TRUE)) |>
  mutate(mean_DrAV = ifelse(is.na(mean_DrAV),
```

```

          0, mean_DrAV
)) |>
mutate(
  roll_DrAV =
    rollapply(mean_DrAV,
              width = 13,
              FUN = mean,
              na.rm = TRUE,
              fill = "extend",
              partial = TRUE
    )
)

```

TIP

For the exercise in R, you will want to change the `rollapply()` `mean()` to be other functions such as `median()`, not the `group_by()` `mean()` that is part of the `summarize()`.

Then plot to create [Figure 7-4](#):

```

## R
ggplot(draft_chart_r, aes(Pick, roll_DrAV)) +
  geom_point() +
  geom_smooth() +
  theme_bw() +
  ylab("Rolling average (\u00B1 6) DrAV") +
  xlab("Draft pick")

```

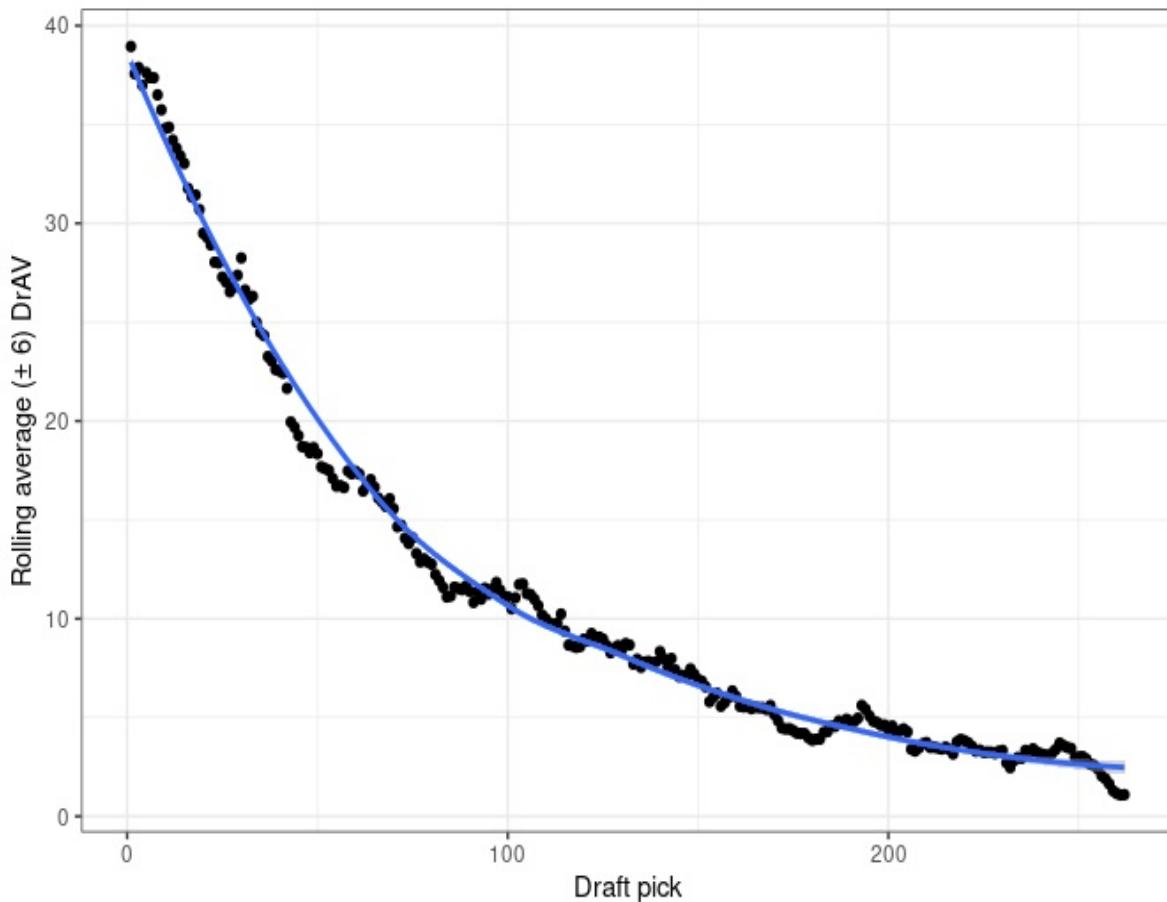


Figure 7-4. Scatterplot with smoothed trendline for draft pick number against draft approximate value, plotted with ggplot2.

Figure 7-3 and **Figure 7-4** help you see that lower draft picks generally have high value added to team based. From here, you can simply fit a model to the data to help quantitatively this result. This model will allow you to use numbers rather than only examining a figure. There are a number of models you can use, and some of them (like LOESS curves or GAMs) are beyond the scope of this book. We have you fit a simple linear model to the logarithm of the data, while fixing the y-intercept - and transform back using an exponential function.

WARNING

Taking $\log(0)$ is mathematically undefined. So, people will often add a value to allow this transformation to occur. Often a small number like 1 or 0.001 is used. Beware this transformation can change your model results sometimes, so you may want to try different values for the number you add.

In Python first drop the index (so you can access `Pick` with the model) and then plot using:

```
## Python
draft_chart_py.reset_index(inplace=True)

draft_chart_py["roll_DrAV_log"] =\
    np.log(draft_chart_py["roll_DrAV"] + 1)

DrAV_pick_fit_py = \
    smf.ols(formula="roll_DrAV_log ~ Pick",
            data=draft_chart_py)\ \
    .fit()

print(DrAV_pick_fit_py.summary())
              OLS Regression Results
=====
=====
Dep. Variable:      roll_DrAV_log    R-squared:
0.970
Model:                 OLS    Adj. R-squared:
0.970
Method:                Least Squares    F-statistic:
8497.
Date:        Sat, 20 May 2023    Prob (F-statistic):
1.38e-200
Time:             15:43:23    Log-Likelihood:
177.05
No. Observations:      262    AIC:
-350.1
Df Residuals:          260    BIC:
-343.0
Df Model:                   1
Covariance Type:    nonrobust
=====
=====
           coef    std err          t      P>|t|      [0.025
0.975]
-----
Intercept     3.4871      0.015    227.712      0.000      3.457
3.517
Pick        -0.0093      0.000    -92.180      0.000     -0.010
-0.009
=====
=====
Omnibus:            3.670    Durbin-Watson:
0.101
```

```

Prob(Omnibus):          0.160    Jarque-Bera (JB): 
3.748                  0.274    Prob(JB): 
Skew:                   0.154    Cond. No. 
Kurtosis:                2.794    ===== 
Kurtosis:               304.    ===== 
===== 
===== 

```

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

And then merge back into `draft_chart_py` and look at the top of the data:

```

## Python
draft_chart_py["fitted_DrAV"] = \
    np.exp(DrAV_pick_fit_py.predict()) - 1

draft_chart_py\
    .head()
   Pick DrAV_mean  roll_DrAV  roll_DrAV_log  fitted_DrAV
0     1      47.60  38.950000      3.687629  31.386918
1     2      39.85  37.575000      3.652604  31.086948
2     3      44.45  37.883333      3.660566  30.789757
3     4      31.15  36.990000      3.637323  30.495318
4     5      43.65  37.627273      3.653959  30.203606

```

In R use:

```

## R
DrAV_pick_fit_r <-
  draft_chart_r |>
  lm(formula = log(roll_DrAV + 1) ~ Pick)

summary(DrAV_pick_fit_r)
Call:
lm(formula = log(roll_DrAV + 1) ~ Pick, data = draft_chart_r)

Residuals:
    Min      1Q  Median      3Q      Max  
-0.32443 -0.07818 -0.02338  0.08797  0.34123 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 3.4870598  0.0153134 227.71   <2e-16 *** 
Pick        -0.0093052  0.0001009  -92.18   <2e-16 *** 

```

```

---
Signif. codes:  0 '****' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.1236 on 260 degrees of freedom
Multiple R-squared:  0.9703,    Adjusted R-squared:  0.9702
F-statistic:  8497 on 1 and 260 DF,  p-value: < 2.2e-16

```

And then merge back into `draft_chart_r` and look at the top of the data:

```

## R
draft_chart_r <-
  draft_chart_r |>
  mutate(
    fitted_DrAV =
      pmax(
        0,
        exp(predict(DrAV_pick_fit_r)) - 1
      )
  )
draft_chart_r |>
  head()
# A tibble: 6 × 4
  Pick mean_DrAV roll_DrAV fitted_DrAV
  <int>     <dbl>     <dbl>     <dbl>
1     1       47.6     39.0     31.4
2     2       39.8     37.6     31.1
3     3       44.4     37.9     30.8
4     4       31.2     37.0     30.5
5     5       43.6     37.6     30.2
6     6       34.7     37.4     29.9

```

So, to recap, in this section, you just calculated the estimated value for each pick. Notice this fit likely underestimates the value of the pick at the very beginning of the draft, because of the two-parameter nature of the exponential regression. This can be improved upon with a different model type, like the examples mentioned above. The shortcomings notwithstanding, this estimate will allow you to explore different draft situations, something you'll do in the very next section.

The Jets/Colts 2018 Trade Evaluated

Now that we have an estimate for the worth of each draft pick, let's look at what this model would have said about the trade between the Jets and the Colts in

Table 7-1:

```
## R
library(kableExtra)
future_pick <-
  tibble(
    Pick = "Future 2nd round",
    Value = "14.8 (discounted at rate of 25%)"
  )

team <- tibble("Receiving team" = c("Jets", rep("Colts", 4)))

tbl_1 <-
  draft_chart_r |>
  filter(Pick %in% c(3, 6, 37, 49)) |>
  select(Pick, fitted_DrAV) |>
  rename(Value = fitted_DrAV) |>
  mutate(
    Pick = as.character(Pick),
    Value = as.character(round(Value, 1))
  ) |>
  bind_rows(future_pick)

team |>
  bind_cols(tbl_1) |>
  kbl(format = "pipe") |>
  kable_styling()
```

Table 7-1. Table 4.1: Trade between the Jets and Colts. The future pick is discounted at 25% because a rookie contract is four years and the waiting one year is 25%.

Receiving team	Pick	Value
Jets	3	30.8
Colts	6	29.9
Colts	37	22.2
Colts	49	19.7

Colts	Future 2nd round	14.8 (discounted at rate of 25%)
-------	------------------	----------------------------------

Adding up these values from [Table 7-1](#), it looks like the Jets got fleeced, losing an expected 55.6 DrAV in the trade. That's more than the value of the first-overall pick! These are just the statistically “expected” from a generic draft pick at that position using the model developed in this chapter based upon previous drafts. Now, as of the writing of this book, what was predicted has come to fruition. You can use new data to see the actual DrAV for the players by creating [Table 7-2](#):

```
## R
library(kableExtra)

future_pick <-
  tibble(
    Pick = "Future 2nd round",
    Value = "14.8 (discounted at rate of 25)"
  )

results_trade <-
  tibble(
    Team = c("Jets", rep("Colts", 5)),
    Pick = c(
      3, 6, 37,
      "49-traded for 52",
      "49-traded for 169",
      "52 in 2019"
    ),
    Player = c(
      "Sam Darnold",
      "Quenton Nelson",
      "Braden Smith",
      "Kemoko Turay",
      "Jordan Wilkins",
      "Rock Ya-Sin"
    ),
    "DrAV" = c(25, 55, 32, 5, 8, 11)
  )

results_trade |>
  kbl(format = "pipe") |>
  kable_styling()
```

Table 7-2. Table 4.2: Trade results between the Jets and Colts.

Team	Pick	Player	DrAV
Jets	3	Sam Darnold	25
Colts	6	Quenton Nelson	55
Colts	37	Braden Smith	32
Colts	49-traded for 52	Kemoko Turay	5
Colts	49-traded for 169	Jordan Wilkins	8
Colts	52 in 2019	Rock Ya-Sin	11

So, the final tally was Jets 25 DrAV, Colts 111, a loss of 86 DrAV, which is almost three times the first overall pick!

This isn't always the case when a team trades up to draft a quarterback. For example, the Chiefs traded up for Patrick Mahomes, using two first-round picks and a third-round pick in 2017 to select the signal caller from Texas Tech, and that worked out to the tune of 85 DrAV, and (as of 2022) two Super Bowl championships.

There are more robust ways to price draft picks, which can be found in the sources mentioned in “[Analyzing the NFL Draft](#)”. Generally speaking, using market-based data – such as the size of a player’s first contract after their rookie deal – is the industry standard. Pro Football Reference’s DrAV values are a decent proxy but have some issues, namely that they don’t properly account for positional value – that a quarterback is much, much more valuable if they draft that position compared to any other position. For more on draft curves, the book *The Drafting Stage: Creating a Marketplace for NFL Draft Picks* (Self-published, 2020) by Eric’s former colleague at PFF, Brad Spielberger, and the founder of OverTheCap, Jason Fitzgerald, is a great place to start.

Are Some Teams Better at Drafting Players than Others?

The question of whether some teams are better at drafting than others is a hard one, because of the way in which draft picks are assigned to teams. The best teams choose at the end of each round, and as we saw above, the better players are picked before the weaker ones. So there could be a situation where we mistakenly assume the worst teams are the best drafters, and vice versa. To account for this, we need to adjust expectations for each pick, using the model we created above. Doing so, and taking the average and standard deviations of the difference between DrAV and fitted_DrAV, and aggregating over the 2000 - 2019 drafts, we arrive at the following ranking using Python:

```
## Python
draft_py_use_pre2019 = \
    draft_py_use_pre2019\
    .merge(draft_chart_py[["Pick", "fitted_DrAV"]], \
           on="Pick")

draft_py_use_pre2019["OE"] = (
    draft_py_use_pre2019["DrAV"] -
    draft_py_use_pre2019["fitted_DrAV"]
)

draft_py_use_pre2019\
    .groupby("Tm")\
    .agg({"OE": ["count", "mean", "std"]})\
    .reset_index()\
    .sort_values([("OE", "mean")], ascending=False)
Tm      OE
          count      mean        std
26  PIT    161  3.523873  18.878551
11  GNB    180  3.371433  20.063320
  8  DAL    160  2.461129  16.620351
  1  ATL    148  2.291654  16.124529
21  NOR    131  2.263655  18.036746
22  NWE    176  2.162438  20.822443
13  IND    162  1.852253  15.757658
  4  CAR    148  1.842573  16.510813
  2  BAL    170  1.721930  16.893993
27  SEA    181  1.480825  16.950089
16  LAC    144  1.393089  14.608528
  5  CHI    149  0.672094  16.052031
20  MIN    167  0.544533  13.986365
```

15	KAN	154	0.501463	15.019527
25	PHI	162	0.472632	15.351785
6	CIN	176	0.466203	15.812953
14	JAX	158	0.182685	13.111672
30	TEN	172	0.128566	12.662670
12	HOU	145	-0.075827	12.978999
28	SFO	184	-0.092089	13.449491
31	WAS	150	-0.450485	9.951758
24	NYJ	137	-0.534640	13.317478
0	ARI	149	-0.601563	14.295335
23	NYG	145	-0.879900	12.471611
29	TAM	153	-0.922181	11.409698
3	BUF	161	-0.985761	12.458855
17	LAR	175	-1.439527	11.985219
19	MIA	151	-1.486282	10.470145
9	DEN	159	-1.491545	12.594449
10	DET	155	-1.765868	12.061696
18	LVR	162	-2.587423	10.217426
7	CLE	170	-3.557266	10.336729

NOTE

The `.reset_index()` function in Python helps us because a dataframe in pandas has row names (an *index*) that can get confused when appending values.

Or in R:

```
## R
draft_r_use_pre2019 <-
  draft_r_use_pre2019 |>
  left_join(draft_chart_r |> select(Pick, fitted_DrAV),
             by = "Pick"
  )

draft_r_use_pre2019 |>
  group_by(Tm) |>
  summarize(
    total_picks = n(),
    DrAV_OE = mean(DrAV - fitted_DrAV, na.rm = TRUE),
    DrAV_sigma = sd(DrAV - fitted_DrAV, na.rm = TRUE)
  ) |>
  arrange(-DrAV_OE) |>
  print(n = Inf)
# A tibble: 32 × 4
  Tm      total_picks DrAV_OE DrAV_sigma
  <chr>        <int>   <dbl>      <dbl>
```

1	PIT	161	3.52	18.9
2	GNB	180	3.37	20.1
3	DAL	160	2.46	16.6
4	ATL	148	2.29	16.1
5	NOR	131	2.26	18.0
6	NWE	176	2.16	20.8
7	IND	162	1.85	15.8
8	CAR	148	1.84	16.5
9	BAL	170	1.72	16.9
10	SEA	181	1.48	17.0
11	LAC	144	1.39	14.6
12	CHI	149	0.672	16.1
13	MIN	167	0.545	14.0
14	KAN	154	0.501	15.0
15	PHI	162	0.473	15.4
16	CIN	176	0.466	15.8
17	JAX	158	0.183	13.1
18	TEN	172	0.129	12.7
19	HOU	145	-0.0758	13.0
20	SFO	184	-0.0921	13.4
21	WAS	150	-0.450	9.95
22	NYJ	137	-0.535	13.3
23	ARI	149	-0.602	14.3
24	NYG	145	-0.880	12.5
25	TAM	153	-0.922	11.4
26	BUF	161	-0.986	12.5
27	LAR	175	-1.44	12.0
28	MIA	151	-1.49	10.5
29	DEN	159	-1.49	12.6
30	DET	155	-1.77	12.1
31	LVR	162	-2.59	10.2
32	CLE	170	-3.56	10.3

To no one's surprise, some of the storied franchises in the NFL have drafted the best above the draft curve since 2000: the Pittsburgh Steelers, Green Bay Packers, and the Dallas Cowboys.

It also won't surprise anyone that last three teams on this list, the Cleveland Browns, Oakland/Las Vegas Raiders, and the Detroit Lions, all have huge droughts in terms of team success as of the time of this writing. The Raiders haven't won their division since they last played in the Super Bowl in 2002, the Lions haven't won theirs since it was called the NFC Central in 1993, and the Cleveland Browns left the league, became the Baltimore Ravens, and came back since their last division title in 1989.

The question is, is this success and futility statistically significant? In

[Appendix B](#), we talked about standard errors and credible intervals. One reason why we added the standard deviation to this table is so that we could easily compute the standard error for each team. This may be done in Python:

```
## Python
draft_py_use_pre2019 = \
    draft_py_use_pre2019\
    .merge(draft_chart_py[["Pick", "fitted_DrAV"]], \
           on="Pick")

draft_py_use_pre2019_tm = ( \
    draft_py_use_pre2019.groupby("Tm") \
    .agg({"OE": ["count", "mean", "std"]}) \
    .reset_index() \
    .sort_values([("OE", "mean")], ascending=False)
)

draft_py_use_pre2019_tm.columns = \
    list(map("_".join, draft_py_use_pre2019_tm.columns))

draft_py_use_pre2019_tm.reset_index(inplace=True)

draft_py_use_pre2019_tm["se"] = ( \
    draft_py_use_pre2019_tm["OE_std"] / \
    np.sqrt(draft_py_use_pre2019_tm["OE_count"]))
)

draft_py_use_pre2019_tm["lower_bound"] = ( \
    draft_py_use_pre2019_tm["OE_mean"] - 1.96 * \
    draft_py_use_pre2019_tm["se"])
)

draft_py_use_pre2019_tm["upper_bound"] = ( \
    draft_py_use_pre2019_tm["OE_mean"] + 1.96 * \
    draft_py_use_pre2019_tm["se"])
)

print(draft_py_use_pre2019_tm)
   index  Tm_  OE_count  ...      se  lower_bound  upper_bound
0      26  PIT       161  ...  1.487838     0.607710  6.440036
1      11  GNB       180  ...  1.495432     0.440387  6.302479
2       8  DAL       160  ...  1.313954    -0.114221  5.036479
3       1  ATL       148  ...  1.325428    -0.306186  4.889493
4      21  NOR       131  ...  1.575878    -0.825066  5.352375
5      22  NWE       176  ...  1.569551    -0.913882  5.238757
6      13  IND       162  ...  1.238039    -0.574302  4.278809
7       4  CAR       148  ...  1.357180    -0.817501  4.502647
8       2  BAL       170  ...  1.295710    -0.817661  4.261522
```

9	27	SEA	181	...	1.259890	-0.988560	3.950210
10	16	LAC	144	...	1.217377	-0.992970	3.779149
11	5	CHI	149	...	1.315034	-1.905372	3.249560
12	20	MIN	167	...	1.082297	-1.576770	2.665836
13	15	KAN	154	...	1.210308	-1.870740	2.873667
14	25	PHI	162	...	1.206150	-1.891423	2.836686
15	6	CIN	176	...	1.191946	-1.870012	2.802417
16	14	JAX	158	...	1.043109	-1.861808	2.227178
17	30	TEN	172	...	0.965520	-1.763852	2.020984
18	12	HOU	145	...	1.077847	-2.188407	2.036754
19	28	SFO	184	...	0.991510	-2.035448	1.851270
20	31	WAS	150	...	0.812558	-2.043098	1.142128
21	24	NYJ	137	...	1.137789	-2.764706	1.695427
22	0	ARI	149	...	1.171119	-2.896957	1.693831
23	23	NYG	145	...	1.035711	-2.909893	1.150093
24	29	TAM	153	...	0.922419	-2.730123	0.885761
25	3	BUF	161	...	0.981895	-2.910275	0.938754
26	17	LAR	175	...	0.905997	-3.215282	0.336228
27	19	MIA	151	...	0.852048	-3.156297	0.183732
28	9	DEN	159	...	0.998805	-3.449202	0.466113
29	10	DET	155	...	0.968819	-3.664752	0.133017
30	18	LVR	162	...	0.802757	-4.160827	-1.014020
31	7	CLE	170	...	0.792791	-5.111136	-2.003396

[32 rows x 8 columns]

Or in R:

```
## R
draft_r_use_pre2019 |>
  group_by(Tm) |>
  summarize(
    total_picks = n(),
    DrAV_OE = mean(DrAV - fitted_DrAV,
      na.rm = TRUE
    ),
    DrAV_sigma = sd(DrAV - fitted_DrAV,
      na.rm = TRUE
    )
  ) |>
  mutate(
    se = DrAV_sigma / sqrt(total_picks),
    lower_bound = DrAV_OE - 1.96 * se,
    upper_bound = DrAV_OE + 1.96 * se
  ) |>
  arrange(-DrAV_OE) |>
  print(n = Inf)
# A tibble: 32 × 7
```

Tm	total_picks	DrAV_OE	DrAV_sigma	se	lower_bound	upper_bound
<chr>	<int>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1 PIT	161	3.52	18.9	1.49	0.608	6.44
2 GNB	180	3.37	20.1	1.50	0.440	6.30
3 DAL	160	2.46	16.6	1.31	-0.114	5.04
4 ATL	148	2.29	16.1	1.33	-0.306	4.89
5 NOR	131	2.26	18.0	1.58	-0.825	5.35
6 NWE	176	2.16	20.8	1.57	-0.914	5.24
7 IND	162	1.85	15.8	1.24	-0.574	4.28
8 CAR	148	1.84	16.5	1.36	-0.818	4.50
9 BAL	170	1.72	16.9	1.30	-0.818	4.26
10 SEA	181	1.48	17.0	1.26	-0.989	3.95
11 LAC	144	1.39	14.6	1.22	-0.993	3.78
12 CHI	149	0.672	16.1	1.32	-1.91	3.25
13 MIN	167	0.545	14.0	1.08	-1.58	2.67
14 KAN	154	0.501	15.0	1.21	-1.87	2.87
15 PHI	162	0.473	15.4	1.21	-1.89	2.84
16 CIN	176	0.466	15.8	1.19	-1.87	2.80
17 JAX	158	0.183	13.1	1.04	-1.86	2.23
18 TEN	172	0.129	12.7	0.966	-1.76	2.02
19 HOU	145	-0.0758	13.0	1.08	-2.19	2.04
20 SFO	184	-0.0921	13.4	0.992	-2.04	1.85
21 WAS	150	-0.450	9.95	0.813	-2.04	1.14
22 NYJ	137	-0.535	13.3	1.14	-2.76	1.70
23 ARI	149	-0.602	14.3	1.17	-2.90	1.69
24 NYG	145	-0.880	12.5	1.04	-2.91	1.15
25 TAM	153	-0.922	11.4	0.922	-2.73	0.886
26 BUF	161	-0.986	12.5	0.982	-2.91	0.939
27 LAR	175	-1.44	12.0	0.906	-3.22	0.336
28 MIA	151	-1.49	10.5	0.852	-3.16	0.184
29 DEN	159	-1.49	12.6	0.999	-3.45	0.466
30 DET	155	-1.77	12.1	0.969	-3.66	0.133
31 LVR	162	-2.59	10.2	0.803	-4.16	-1.01
32 CLE	170	-3.56	10.3	0.793	-5.11	-2.00

TIP

Look at this long code outputs, the 95% confidence intervals can help you see which teams *DrAV_OE* differed from zero. In both the Python and R outputs, 95% confidence intervals are *lower_bound* and *upper_bound*. If this interval does not contain the value, you can consider it statistically different from zero. If the *DrAV_OE* is greater than the interval, the team did statistically better than average. If the *DrAV_OE* is less than the interval, the team did statistically worse than average.

So, using a 95 percent confidence interval it looks like two teams are statistically-significantly better at drafting players than other teams, pick-for-

pick (the Steelers and Packers), while two teams are statistically-significantly worse at drafting players than the other teams (the Raiders and Browns). This is consistent with the research on the topic, which suggests that over a reasonable time interval (such as the average length of a general manager or coach's career) it's very hard to discern drafting talent.

The way to “win” the NFL draft is to make draft pick trades like the Colts did against the Jets, and give yourself more bites at the apple, as it were. A [PFF article by Timo Riske](#) discusses this more.

One place where that famously failed, however, is with one of the two teams that has been historically bad at drafting since 2002, the Oakland/Las Vegas Raiders. In 2018 the Raiders traded their best player, edge player Khalil Mack, to the Chicago Bears for two first-round picks and an exchange of later-round picks. The Raiders were unable to ink a contract extension with Mack, whom the Bears later signed for the richest deal in the history of NFL defensive players. The [Sloan Sports Analytics Conference](#) - the most high-profile gathering of sports analytics professionals in the world–lauded the trade for the Raiders, giving them the award for the best transaction at their 2019 conference.

Generally speaking, trading one player for a bunch of players is going to go well for the team that acquires the bunch of players, even if they are draft picks. However, the Raiders, proven above to be statistically notorious for bungling the picks, were unable to do much with the selections, with the best pick of the bunch being running back Josh Jacobs. Jacobs did lead the NFL in rushing yards in 2022, but prior to that point had failed to distinguish himself in the NFL, failing to earn a fifth year on his rookie contract. The other first-round pick in the trade, Damon Arnette, lasted less than two years with the team, while with Mack on the roster the Bears made the playoffs twice and won a division title in his first year with the club in 2018.

Now, you've seen the basics of web scraping. What you do with this data is largely up to you! Like almost anything, the more you web scrape, the better you will become.

TIP

A suggestion for finding URLs is to use your web browser's (such as Chrome, Edge, or Firefox)

inspection tool. This shows the HTML code for the web page you are visiting. You can use this to help find which path for the table that you want based upon the HTML and CSS selectors.

Data Science Tools Used in This Chapter

This chapter covered the following topics:

- How to web scrape data in Python and R for the NFL Draft and NFL Scouting Combine data from [Pro Football Reference](#).
- How to use `for` loops in both Python and R.
- You learned how to calculate rolling averages in Python using `rolling()` and in R using `rollapply()`.
- You reapplied data wrangling tools you learned in previous chapters.

Suggested Reading

Many different books and other resources exist on web scraping. Besides the package documentation for `rvest` in R and `read_html` in Pandas, here are two good ones to start with:

- *R Web Scraping Quick Start Guide* by Olgun Aydin (Packet Publishing, 2018).
- *Web Scraping with Python*, 2nd Edition by Ryan Mitchell (O'Reilly Media, 2015).

The Drafting Stage: Creating a Marketplace for NFL Draft Picks by Eric's former colleague at PFF, Brad Spielberger, and the founder of the website [OverTheCap](#), Jason Fitzgerald, (self-published, 2020) provides an overview of the NFL draft along with many great details.

Exercises

1. Change the webscraping examples to different ranges of years for the NFL

draft. Does an error come up? Why?

2. Use the process laid out in this chapter to scrape NFL Scouting Combine Data using the general URL <https://www.pro-football-reference.com/draft/YEAR-combine.htm> (You will need to change YEAR). This is a preview for [Chapter 8](#) where you will dive into this data further.
3. With scouting combine data, plot the 40-yard dash times for each player, with point color determined by the position the player plays. Which position is the fastest? The slowest? Do the same for the other events. What are the patterns you see?
4. Use the scouting combine data in conjunction with the draft data scraped in this chapter. What is the relationship between where a player is selected in the NFL Draft and their 40-yard dash time? Is this relationship more pronounced for some positions? How does your answer to question 3 influence your approach to this question?
5. For the draft curve exercise, change the six-pick moving average to a six-pick moving median. What happens? Is this preferable?

Chapter 8. Principal Component Analysis and Clustering: Player Attributes

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

In the era of big data, there is a strong urge to simply “throw the kitchen sink” at data in an attempt to find patterns and draw value from such patterns. In football this urge, while maybe a bit behind the general population, is strong as well. While in general this approach should be taken with caution, due to the non-stationary and general small-sampled nature of the game, if handled with care the process of *unsupervised learning* (in contrast to *supervised learning*, both of which are defined in a couple of paragraphs) can yield insights that are useful to us as football analysts.

In this chapter you will use NFL Scouting Combine data that you will obtain through Pro Football Reference from 2000 to 2023. As mentioned in [Chapter 7](#), the NFL Scouting Combine is a yearly event, usually held in Indianapolis, Indiana, where NFL players go through a battery of physical (and other) tests in preparation for the NFL Draft. The entire football industry gets together for what is essentially its yearly conference, and with many of the top players no longer testing while they are there (opting to test at more-friendly *Pro Days* held at their

college campuses), the importance of the on-field events in the eyes of many have waned. Furthermore, the addition of tracking data into our lives as analysts has given rise to more accurate (and timely) estimates of player athleticism, which is a fast-rising set of problems in and of its own.

Several recent articles provide discussion on the NFL Scouting Combine. “[Using Tracking and Charting Data to Better Evaluate NFL Players: A Review](#)” by Eric and other from the Sloan Sports Conference discusses other ways to measure player data. “[Beyond the 40-yard dash: How player tracking could modernize the NFL combine](#)” by Sam Fortier describes how player tracking data might be used in the *Washington Post*. “[Inside the Rams’ major changes to their draft process, and why they won’t go back to ‘normal’](#)” by Jourdan Rodrigue provides coverage of the Ram’s drafting process on the *Athletic*.

The eventual obsolete nature of (at least the on-field testing portion of) the scouting combine notwithstanding, the data provides a good vehicle to study both principal component analysis and clustering. *Principal component analysis* (PCA) is the process of taking a set of features that possess collinearity of some form (*collinearity* in this context means the predictors conceptually contain duplicative information and are numerically correlated) and “mushing” them into smaller subset of features that are each (linearly) independent of one another. Athleticism data is chock-full of these sorts of things. For example, how fast a player runs the 40-yard dash depends very much on how much they weigh - although not perfectly - whereas how high they jump is correlated with how far they can jump. Being able to take a set of features that are all-important in their own right, but not completely independent of each other, and creating a smaller set of data column-wise, is a process that is often referred to as *dimensionality reduction* in data science and related fields.

Clustering is the process of dividing data points into similar groups (“clusters”) based on a set of features. There are many ways to cluster data, but if the groups are not known *a priori*, this process falls into the category of *unsupervised learning*. *Supervised learning* algorithms require a predefined response variable to be trained on. In contrast, *unsupervised learning* essentially allows the data to create the response variable - which in the case of clustering is the cluster - out of thin air. Clustering is a really effective approach in team and individual sports because players are often grouped - either formally or informally - into position

groups in team sports. Sometimes the nature of these position groups change over time, and data can help us detect that change in an effort to help teams adjust their process of building rosters with players that fit into their ideas of positions. Additionally, in both team and individual sports, players have “styles” that can be uncovered in the data. While a set of features depicting a player’s style - often displayed through some sort of plot - can be helpful to the mathematically inclined, for traditionalists it’s often desirable to have player types described by groups - hence the value of clustering here as well.

One can imagine that the process of running a PCA in the data before clustering is essential. If how fast a player runs the 40-yard dash carries much of the same signal as how high they jump when tested for the vertical jump, treating them both as one variable - while not “double counting” in the traditional sense - will be overcounting some traits in favor of others. Thus, you’ll start with the process of running a PCA on data, which is adapted from Eric’s [DataCamp course](#).

TIP

We present basic, introductory methods for multivariate statistics in this chapter. Advanced methods emerge on a regular basis. For example Uniform Manifold Approximation and Projection is emerging as one popular new distance-based tool at the time of this writing and Topological Data Analysis (that uses geometric properties rather than distance) is another. If you understand the basic methods that we present, learning these new tools will be easier, and you will have a benchmark by which to compare these new methods.

Web Scrapping and Visualizing NFL Combine Data

To obtain the data, use similar web scrapping tools as in [Chapter 7](#). Note the change from `draft` to `combine` in the code for the URL. Also, the data requires some cleaning. Sometimes the data contains extra headings. To remove these, remove rows where a value equals the heading (such as `Ht != "Ht"`). In both languages, height (`Ht`) needs to be converted from “foot-inch” to “inches”.

With Python, use this code:

	Ht	Wt	...	Shuttle	Season
count	7970.000000	7975.000000	...	4993.000000	7999.000000
mean	73.801255	242.550094	...	4.400925	2011.698087
std	2.646040	45.296794	...	0.266781	6.950760
min	64.000000	144.000000	...	3.730000	2000.000000
25%	72.000000	205.000000	...	4.200000	2006.000000
50%	74.000000	232.000000	...	4.360000	2012.000000
75%	76.000000	279.500000	...	4.560000	2018.000000
max	82.000000	384.000000	...	5.560000	2023.000000

[8 rows x 9 columns]

```
## Python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

combine_py = pd.DataFrame()
for i in range(2000, 2023 + 1):
    url = "https://www.pro-football-reference.com/draft/" + str(i) +
"-combine.htm"
    web_data = pd.read_html(url)[0]
    web_data["Season"] = i
    web_data = web_data.query('Ht != "Ht"')
    combine_py = pd.concat([combine_py, web_data])

combine_py.reset_index(drop=True, inplace=True)
combine_py.to_csv("combine_data_py.csv", index=False)

combine_py[["Ht-ft", "Ht-in"]] = \
    combine_py["Ht"].str.split("-", expand=True)

combine_py = \
    combine_py\
        .astype({
            "Wt": float,
            "40yd": float,
            "Vertical": float,
            "Bench": float,
            "Broad Jump": float,
            "3Cone": float,
            "Shuttle": float,
            "Ht-ft": float,
            "Ht-in": float
        })

combine_py["Ht"] = (
    combine_py["Ht-ft"] * 12.0 +
    combine_py["Ht-in"]
)
```

```

combine_py["Ht"] = (
    combine_py["Ht-ft"] * 12.0 +
    combine_py["Ht-in"]
)

combine_py\ 
    .drop(["Ht-ft", "Ht-in"], axis=1, inplace=True)

combine_py.describe()

```

With R, use this code:

Player	Pos	School	College
Length:7999 Class :character :character Mode :character :character	Length:7999 Class :character Mode :character	Length:7999 Class :character Mode :character	Length:7999 Class

	Ht	Wt	40yd	Vertical
Bench				
Min. : 2.00	Min. :64.0	Min. :144.0	Min. :4.220	Min. :17.50
1st Qu.:16.00	1st Qu.:72.0	1st Qu.:205.0	1st Qu.:4.530	1st Qu.:30.00
Median :21.00	Median :74.0	Median :232.0	Median :4.690	Median :33.00
Mean :20.74	Mean :73.8	Mean :242.6	Mean :4.774	Mean :32.93
3rd Qu.:25.00	3rd Qu.:76.0	3rd Qu.:279.5	3rd Qu.:4.970	3rd Qu.:36.00
Max. :49.00	Max. :82.0	Max. :384.0	Max. :6.050	Max. :46.50
NA's :2802	NA's :29	NA's :24	NA's :583	NA's :1837
Broad Jump		3Cone	Shuttle	Drafted (tm/rnd/yr)
Min. : 74.0	Min. :109.0	Min. :6.280	Min. :3.730	Length:7999
1st Qu.:116.0	1st Qu.:121.0	1st Qu.:6.980	1st Qu.:4.200	Class :character
Mean :114.8	Mean :147.0	Mean :7.190	Mean :4.360	Mode :character
3rd Qu.:1913	3rd Qu.:3126	3rd Qu.:7.530	3rd Qu.:4.560	
Max. :9.120	NA's :3006	Max. :9.120	Max. :5.560	
Season				
Min. :2000				

```

1st Qu.:2006
Median :2012
Mean    :2012
3rd Qu.:2018
Max.    :2023
## R
library(tidyverse)
library(rvest)
library(htmlTable)
library(multiUS)
library(ggthemes)

combine_r <- tibble()
for (i in seq(from = 2000, to = 2023)) {
  url <- paste0("https://www.pro-football-reference.com/draft/",
                i,
                "-combine.htm")
  web_data <-
    read_html(url) |>
    html_table()
  web_data_clean <-
    web_data[[1]] |>
    mutate(Season = i) |>
    filter(Ht != "Ht")
  combine_r <-
    bind_rows(combine_r,
              web_data_clean)
}
write_csv(combine_r , "combine_data_r.csv")

combine_r <-
  combine_r |>
  mutate(ht_ft = as.numeric(str_sub(Ht, 1, 1)),
         ht_in = str_sub(Ht, 2, 4),
         ht_in = as.numeric(str_remove(ht_in, "-")),
         Ht = ht_ft * 12 + ht_in) |>
  select(-Ht_ft, -Ht_in)
summary(combine_r)

```

Notice here that there are many, many players that don't have a full set of data, as evidenced by the NAs in the R table. A lot of players just take heights and weights at the combine, which is why the number of NAs there are few. You will resolve this issue in a bit, but let's look at the reason you need PCA in the first place: the pairwise correlations between events. First, look at height versus weight. In Python, create **Figure 8-1**:

```
# Python
sns.set_theme(style="whitegrid", palette="colorblind")

sns.regplot(data=combine_py, x="Ht", y="Wt");
plt.show();
```

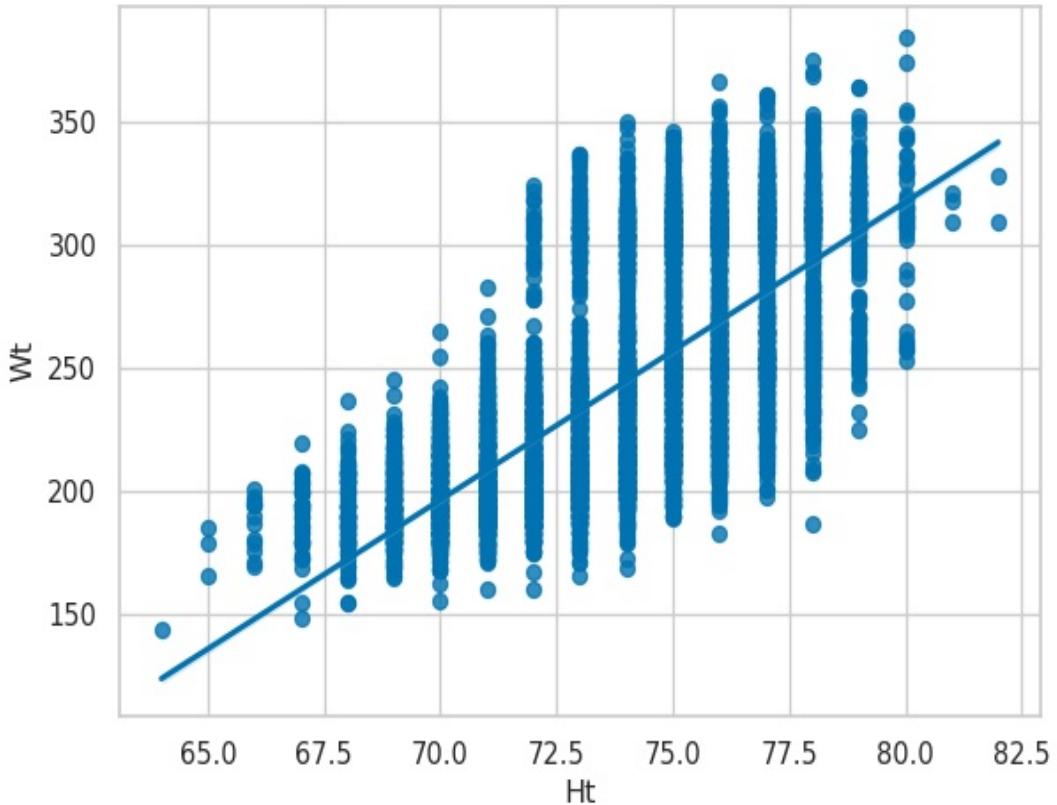


Figure 8-1. Scatterplot with trendline for player height plotted against player weight, plotted with seaborn.

Or, in R create [Figure 8-2](#):

```
# R
ggplot(combine_r, aes(x = Ht, y = Wt)) +
  geom_point() +
  theme_bw() +
  xlab("Player Height (inches)") +
  ylab("Player Weight (pounds)") +
  geom_smooth(method = "lm", formula = y ~ x)
```

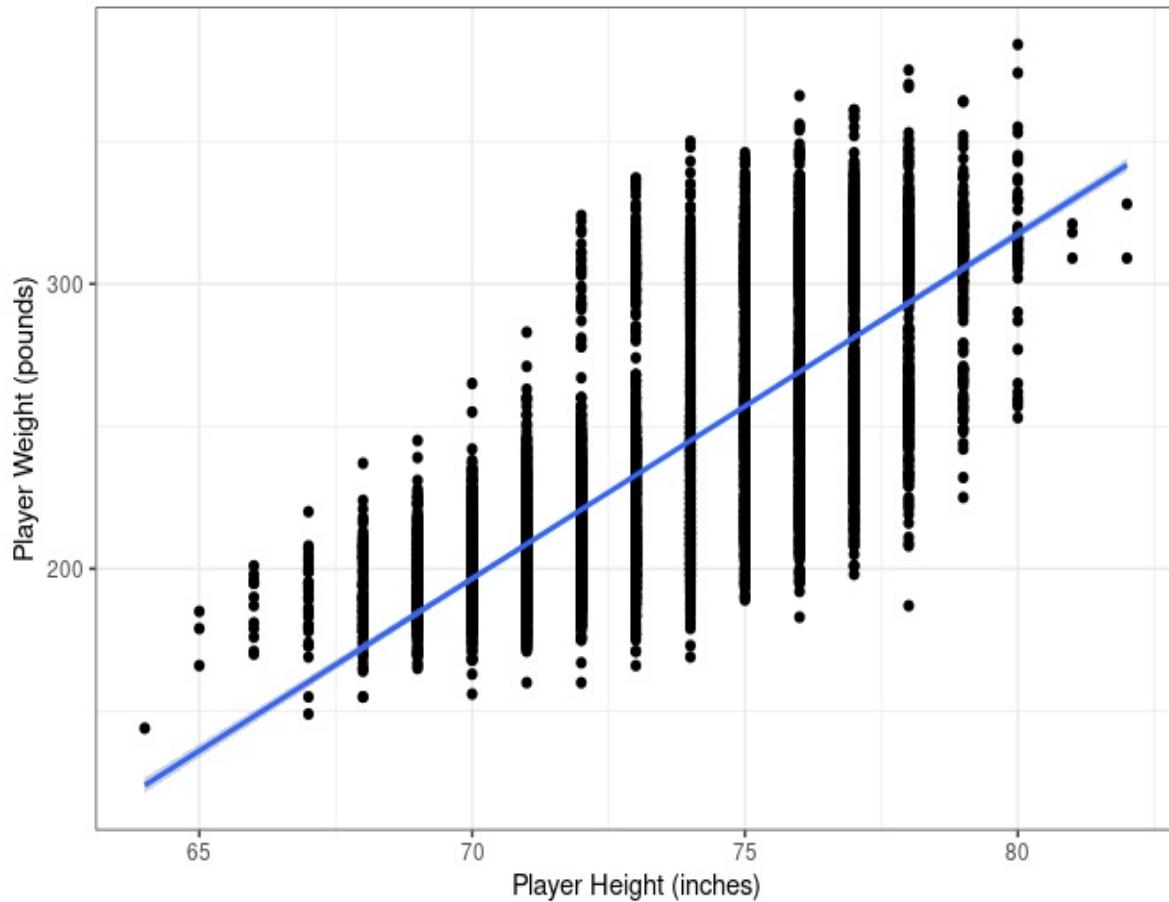


Figure 8-2. Scatterplot with trendline for player height plotted against player weight, plotted with `ggplot2`.

This makes sense – the taller you are the more you weigh. Hence, there really aren't two independent pieces of information here. Let's look at weight versus 40-yard dash. In Python, create **Figure 8-3**:

```
# Python
sns.regplot(data=combine_py,
             x="Wt",
             y="40yd",
             line_kws={"color": "red"});
plt.show();
```

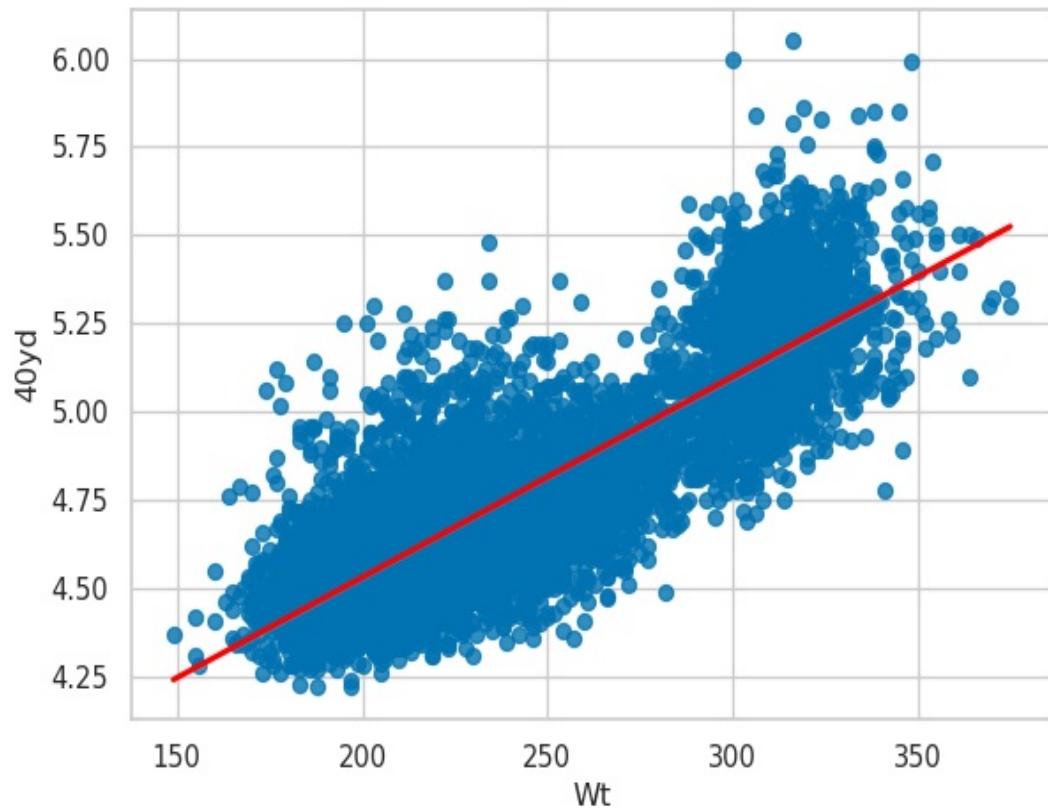


Figure 8-3. Scatterplot with trendline for player weight plotted against 40-yard dash time, plotted with seaborn.

Or, in R create [Figure 8-4](#) using a spline (or “curve”) in contrast to Python that uses straight line:

```
# R
ggplot(combine_r, aes(x = Wt, y = `40yrd`)) +
  geom_point() +
  theme_bw() +
  xlab("Player Weight (pounds)") +
  ylab("Player 40-yard dash (seconds)") +
  geom_smooth(method = "lm", formula = y ~ x)
```

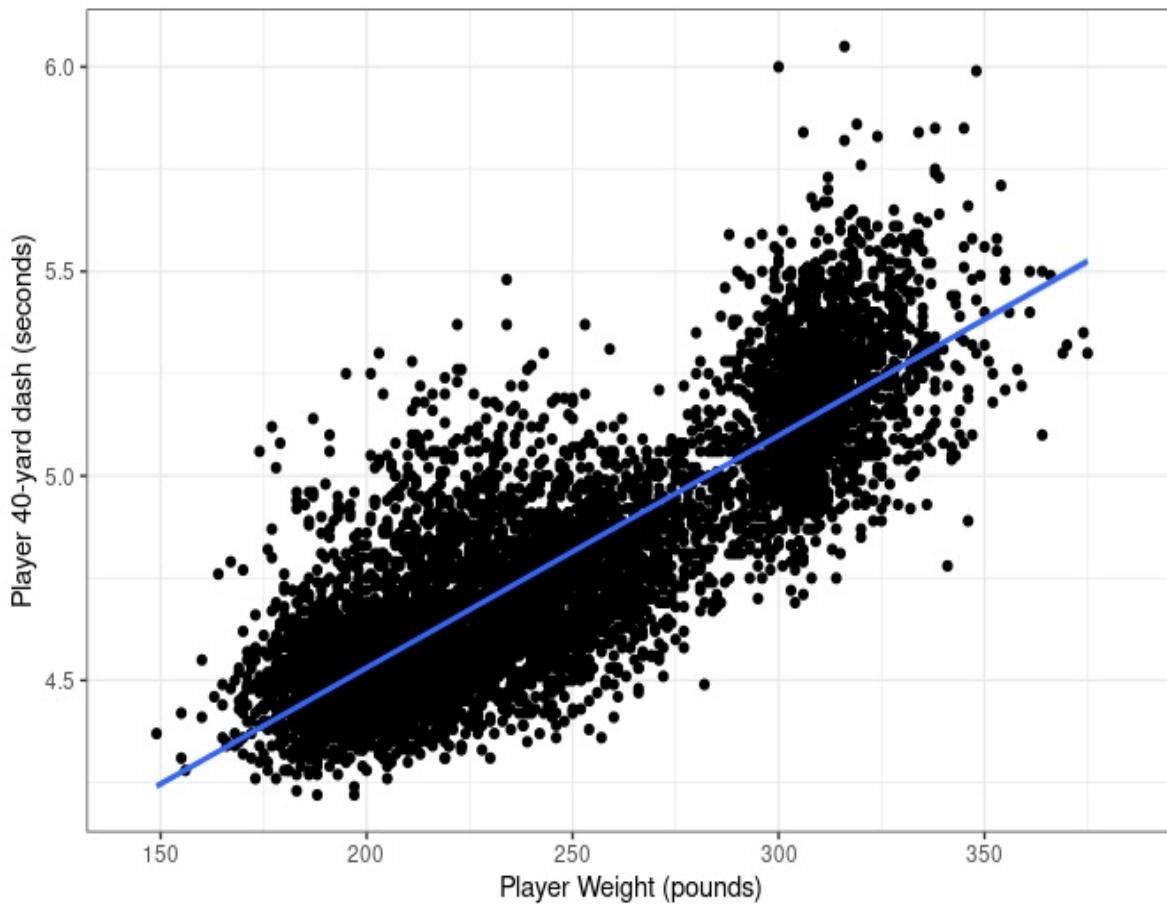


Figure 8-4. Scatterplot with trendline for player weight plotted against 40-yard dash time, plotted with ggplot2.

WARNING

In most computer languages, starting object names with numbers, like `40yd` is bad. The computer does not know what to do because the computer thinks something like arithmetic is about to happen but then the computer gets some letters instead. R lets you use improper names by placing back-ticks (`) around them, like you did to create [Figure 8-3](#)

Here, again, you have a positive correlation. Notice here that two clusters are emerging already in the data, though – a lot of really heavy players (above 300 lbs) and a lot of really light players (near 225 lbs). These two groups serve as an example of a bimodal example – rather than having a single center like a normal distribution, two groups exist. Now, let's look at 40-yard dash and vertical jump. In Python, create [Figure 8-5](#):

```
# Python
sns.regplot(data=combine_py,
             x="40yd",
             y="Vertical",
             line_kws={"color": "red"});
plt.show();
```

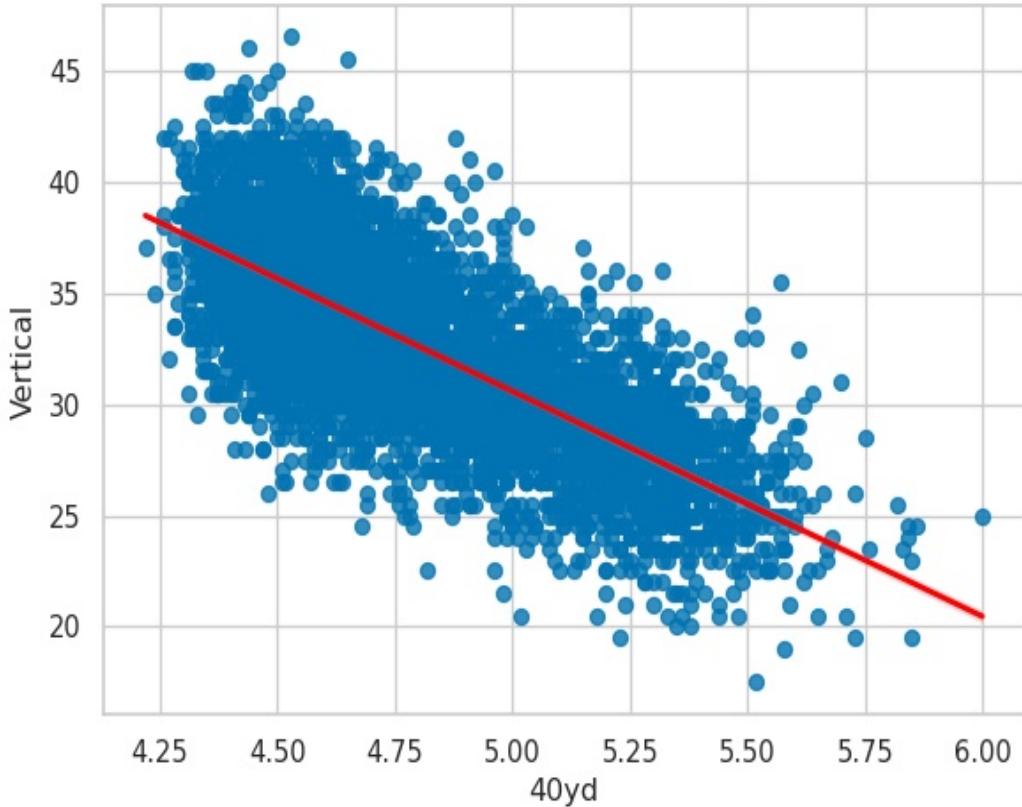


Figure 8-5. Scatterplot with trendline for player 40-yard dash time plotted against vertical jump, plotted with seaborn.

Or, in R create Figure 8-6:

```
# R
ggplot(combine_r, aes(x = `40yd`, y = Vertical)) +
  geom_point() +
  theme_bw() +
  xlab("Player 40-yard dash (seconds)") +
  ylab("Player vertical jump (inches)")
  geom_smooth(method = "lm", formula = y ~ x)
geom_smooth: na.rm = FALSE, orientation = NA, se = TRUE
stat_smooth: na.rm = FALSE, orientation = NA, se = TRUE, method = lm,
formula = y ~ x
```

`position_identity`

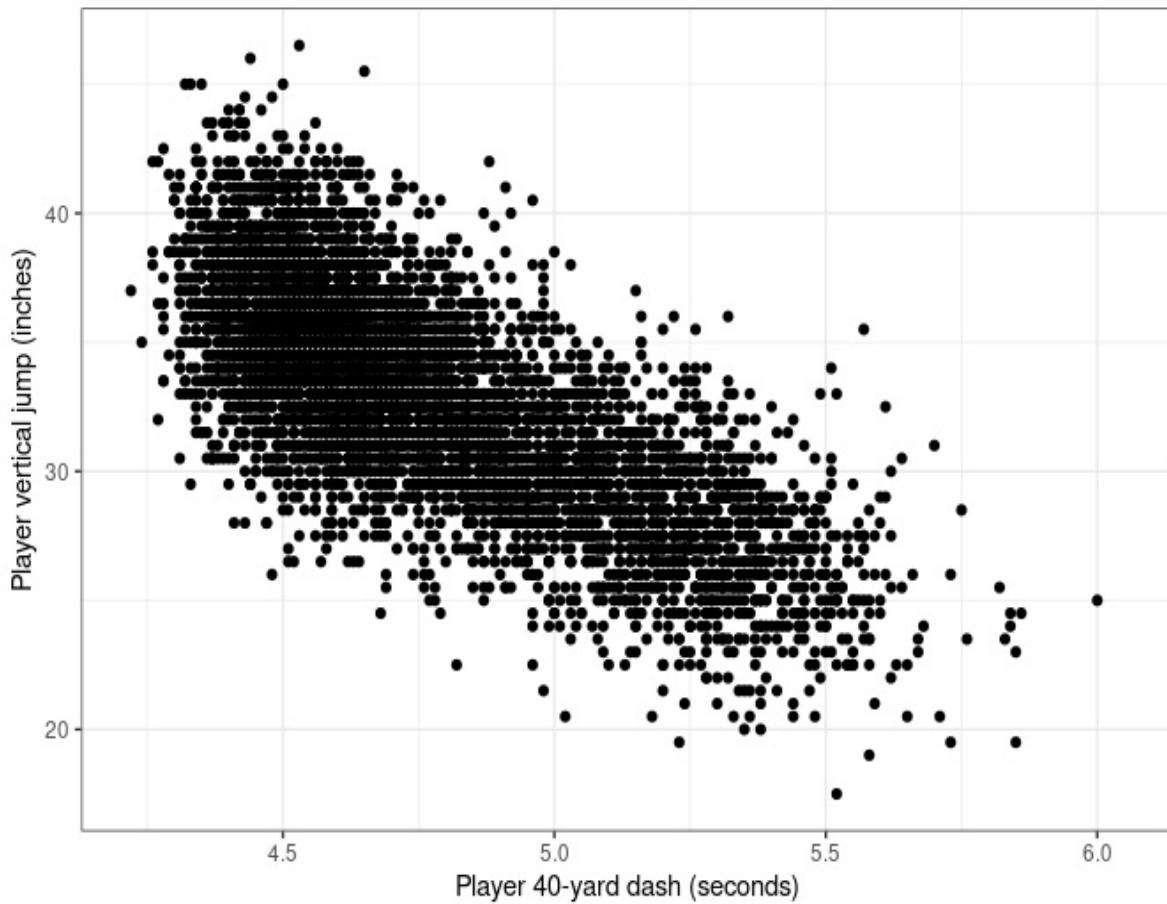


Figure 8-6. Scatterplot with trendline for player 40-yard dash time plotted against vertical jump, plotted with ggplot2.

Here, you have a negative relationship; the faster the player (lower the 40-yard dash, in seconds) the higher the vertical jump (in inches). Does agility (as measured by three-cone drill, see [Figure 8-7](#)) also track with 40-yard dash?



Figure 8-7. Three cone drill. In this drill, players run around the cones following the path and their time is

recorded.

In Python, create **Figure 8-8**:

```
# Python
sns.regplot(data=combine_py,
              x="40yd",
              y="3Cone",
              line_kws={"color": "red"});
plt.show();
```

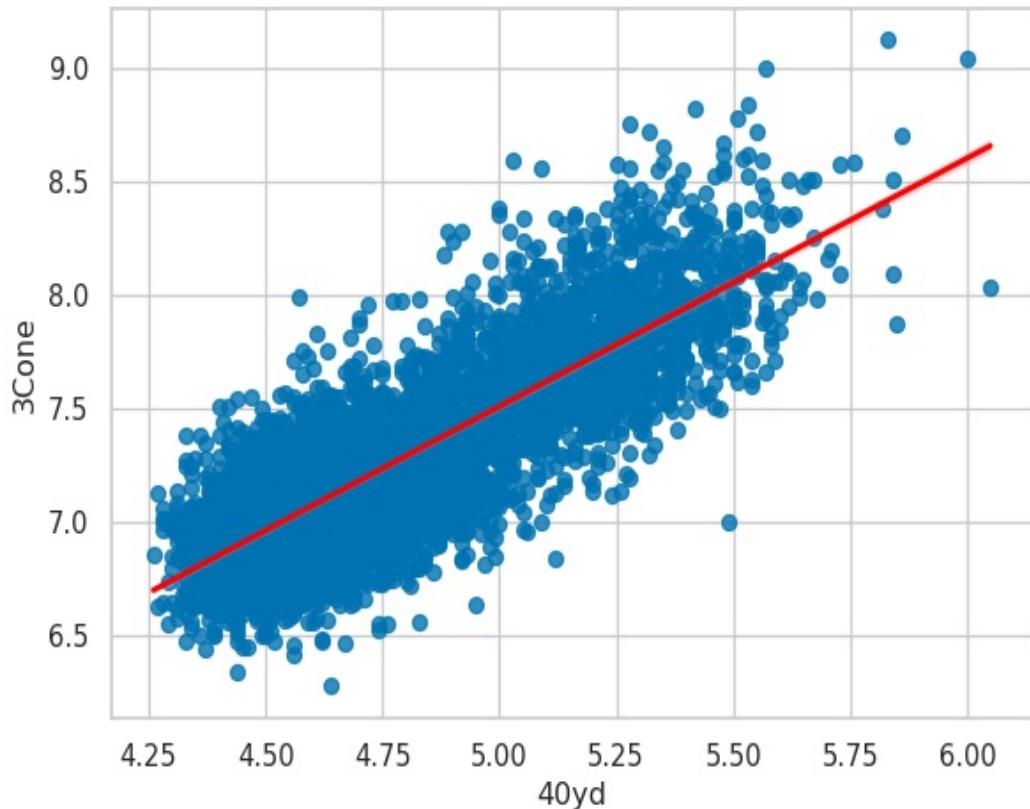


Figure 8-8. Scatterplot with trendline for player 40-yard dash time plotted against their three-cone drill, plotted with seaborn.

Or, in R create **Figure 8-9**:

```
# R
ggplot(combine_r, aes(x = `40yd`, y = `3Cone`)) +
  geom_point() +
  theme_bw() +
  xlab("Player 40-yard dash (seconds)") +
```

```

ylab("Player 3 cone drill (inches)")
geom_smooth(method = "lm", formula = y ~ x)
geom_smooth: na.rm = FALSE, orientation = NA, se = TRUE
stat_smooth: na.rm = FALSE, orientation = NA, se = TRUE, method = lm,
formula = y ~ x
position_identity

```

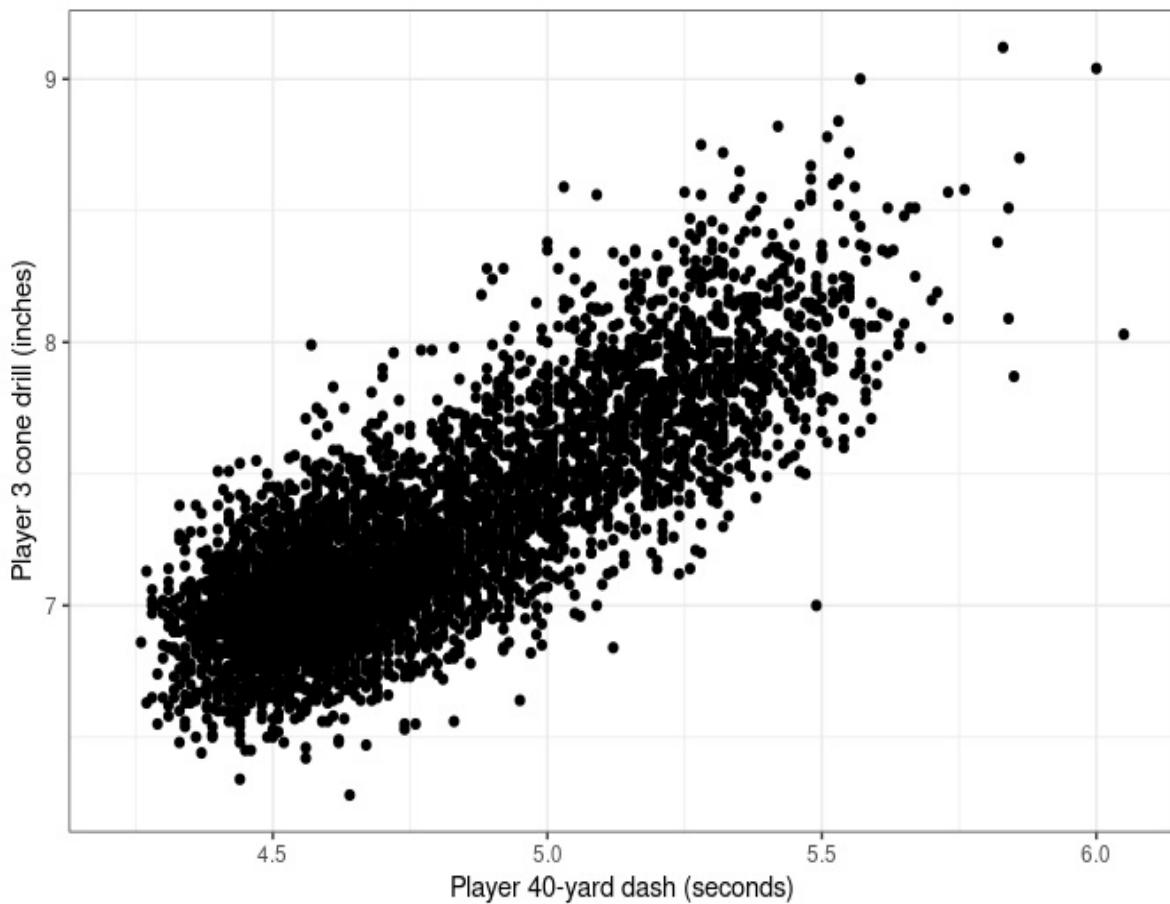


Figure 8-9. Scatterplot with trendline for player 40-yard dash time plotted against three cone drill time, plotted with ggplot2.

Another positive relationship. Hence, it's clear that athleticism is measured in many ways, and it's reasonable to assume that none of them are independent of the other ones.

To commence the process of PCA, we first have to “fill in” the missing data. We use k-nearest neighbors, which is beyond the scope of this book. As you learn more about your data structure, you may want to do your own research to find more other methods for replacing missing values. Just run the code.

NOTE

We included the web-scraping step (again) and imputation step so that the book would be self-contained and you can re-create all of our data yourself. That being said, the book's Git repository also contains the data files in case the websites change or you have trouble downloading the code.

With this method, only players with a recorded height and weight at the combine had their data imputed. We have included an `if-else` statement so this code will only run if the file has not been downloaded and saved to the current directory.

In Python, run:

```
## Python
import numpy as np
import os
from sklearn.impute import KNNImputer

combine_knn_py_file = "combine_knn_py.csv"
col_impute = ["Ht", "Wt", "40yd", "Vertical",
              "Bench", "Broad Jump", "3Cone",
              "Shuttle"]

if not os.path.isfile(combine_knn_py_file):
    combine_knn_py = combine_py.drop(col_impute, axis=1)
    imputer = KNNImputer(n_neighbors=10)
    knn_out_py = imputer.fit_transform(combine_py[col_impute])
    knn_out_py = pd.DataFrame(knn_out_py)
    knn_out_py.columns = col_impute
    combine_knn_py = pd.concat([combine_knn_py, knn_out_py], axis=1)
    combine_knn_py.to_csv(combine_knn_py_file)

else:
    combine_knn_py = pd.read_csv(combine_knn_py_file)

combine_knn_py.describe()
   Unnamed: 0      Season     ...      3Cone      Shuttle
count  7999.000000  7999.000000  ...  7999.000000  7999.000000
mean   3999.000000  2011.698087  ...    7.239512   4.373727
std    2309.256735   6.950760  ...    0.374693   0.240294
min     0.000000  2000.000000  ...    6.280000   3.730000
25%   1999.500000  2006.000000  ...    6.978000   4.210000
50%   3999.000000  2012.000000  ...    7.122000   4.310000
75%   5998.500000  2018.000000  ...    7.450000   4.510000
max   7998.000000  2023.000000  ...    9.120000   5.560000
```

```
[8 rows x 10 columns]
```

In R, run:

```
## R
combine_knn_r_file <- "combine_knn_r.csv"

if (!file.exists(combine_knn_r_file)) {
  imput_input <-
    combine_r |>
    select(Ht:Shuttle) |>
    as.data.frame()

  knn_out_r <-
    KNNimp(imput_input, k = 10,
            scale = TRUE,
            meth = "median") |>
    as_tibble()

  combine_knn_r <-
    combine_r |>
    select(Player:College, Season) |>
    bind_cols(knn_out_r)
  write_csv(x = combine_knn_r,
            file = combine_knn_r_file)
} else {
  combine_knn_r = read_csv(combine_knn_r_file)
}
Rows: 7999 Columns: 13
— Column specification —
Delimiter: ","
chr (4): Player, Pos, School, College
dbl (9): Season, Ht, Wt, 40yd, Vertical, Bench, Broad Jump, 3Cone,
Shuttle

i Use `spec()` to retrieve the full column specification for this
data.
i Specify the column types or set `show_col_types = FALSE` to quiet
this message.
combine_knn_r |>
  summary()
  Player           Pos           School          College
Length:7999      Length:7999      Length:7999      Length:7999
Class :character Class :character Class :character Class
:character       Mode  :character   Mode :character   Mode
```

```
:character
```

Season	Ht	Wt	40yd	
Vertical				
Min. :2000 :17.50	Min. :64.0	Min. :144.0	Min. :4.22	Min.
1st Qu.:2006 Qu.:30.50	1st Qu.:72.0	1st Qu.:205.0	1st Qu.:4.53	1st
Median :2012 :33.50	Median :74.0	Median :232.0	Median :4.68	Median
Mean :2012 :32.93	Mean :73.8	Mean :242.5	Mean :4.77	Mean
3rd Qu.:2018 Qu.:35.50	3rd Qu.:76.0	3rd Qu.:279.0	3rd Qu.:4.97	3rd
Max. :2023 :46.50	Max. :82.0	Max. :384.0	Max. :6.05	Max.
Bench				
Min. : 2.00	Min. : 74.0	Min. :6.280	Min. :3.730	
1st Qu.:16.00	1st Qu.:109.0	1st Qu.:6.975	1st Qu.:4.210	
Median :19.50	Median :116.0	Median :7.140	Median :4.330	
Mean :20.04	Mean :114.7	Mean :7.252	Mean :4.383	
3rd Qu.:24.00	3rd Qu.:121.0	3rd Qu.:7.470	3rd Qu.:4.525	
Max. :49.00	Max. :147.0	Max. :9.120	Max. :5.560	
Broad Jump				
3Cone				
Shuttle				

Notice, no more missing data. But, for reasons that are beyond this book and similar to reasons explained in this Stack Overflow [post](#), the two methods produce similar, but slightly different results because they use slightly different assumptions and methods.

Introduction to PCA

Before you fit the PCA used for later analysis, let's take a short break to look at how a PCA works. Conceptually, a PCA reduces the number of dimensions of data to use the fewest number of dimensions possible. Graphically, *dimensions* refers to the number of axes needed to describe the data. Tabularly, *dimensions* refers to the number of columns needed to describe the data. Algebraically, *dimensions* refers to the number of independent variables needed to describe the data.

Look back at length-weight relation shown in [Figure 8-1](#) and [Figure 8-2](#). Do we need an x-axis and a y-axis to describe this data? Probably not. Let's fit a PCA,

and then you can look at the outputs. You will use the raw data after removing missing values. In Python, use the `PCA` function from scikit-learn package:

```
## Python
from sklearn.decomposition import PCA

pca_wt_ht = PCA(svd_solver="full")
wt_ht_py = \
    combine_py[["Wt", "Ht"]]\n    .query("Wt.notnull() & Ht.notnull())\n    .copy()

pca_fit_wt_ht_py = \
    pca_wt_ht.fit_transform(wt_ht_py)
```

Or, with R use the `prcomp` function from the `stats` package that is part of the core set of R packages:

```
## r
wt_ht_r <-
  combine_r |>
  select(Wt, Ht) |>
  filter(!is.na(Wt) & !is.na(Ht))

pca_fit_wt_ht_r <-
  prcomp(wt_ht_r)
```

Now, let's look at the model details. In Python, use this code to look at the variance explained by each of the new *principal components* (PCs) or new axes of data:

```
## Python
print(pca_wt_ht.explained_variance_ratio_)
[0.99829949  0.00170051]
```

In R, look at the `summary()` of the fit model:

```
## R
summary(pca_fit_wt_ht_r)
Importance of components:
              PC1      PC2
Standard deviation   45.3195 1.8704
Proportion of Variance 0.9983 0.0017
Cumulative Proportion 0.9983 1.0000
```

Both of the outputs show that 1-PC (or axis of data) contains 99.8% of the variability within the data. This tells you that only the first PC is important for the data.

To see the new representation of the data, plot the new data. In Python, use `matplotlib`'s `plot()` to create a simple scatterplot of the outputs to create [Figure 8-10](#):

```
## Python
plt.plot(pca_fit_wt_ht_py[:, 0], pca_fit_wt_ht_py[:, 1], "o");
plt.show();
```

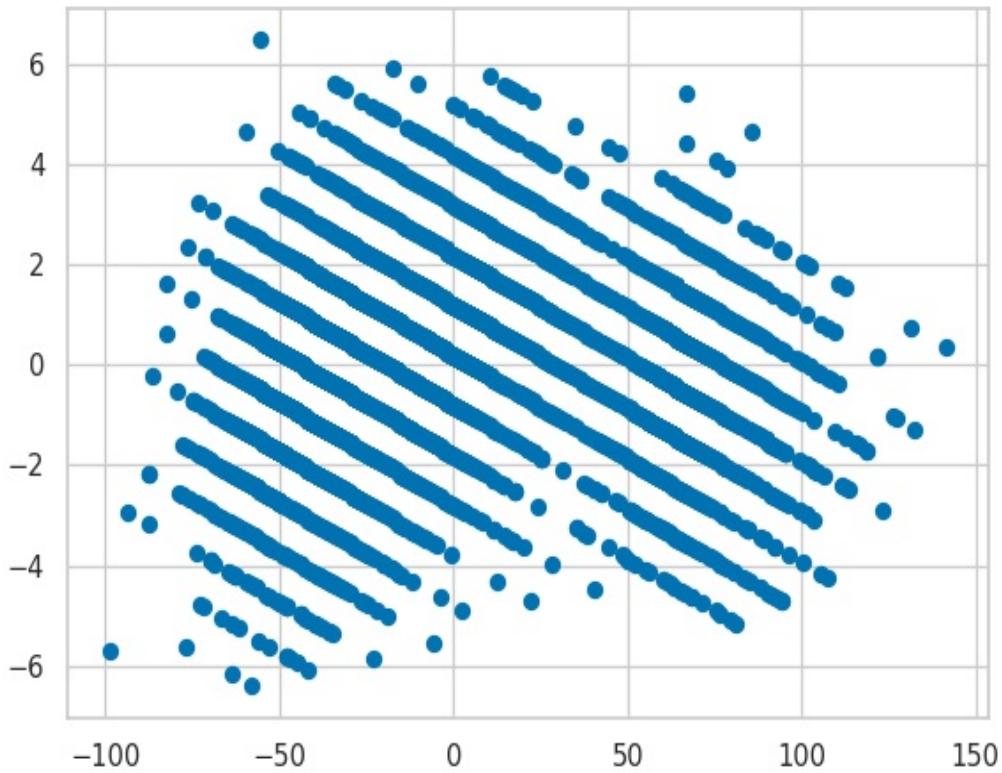


Figure 8-10. Scatterplot of the PCA rotation for weight and height with `plot()` from `matplotlib`.

In R, use `ggplot` to create [Figure 8-11](#):

```
## Python
pca_fit_wt_ht_r$x |>
```

```
as_tibble() |>
ggplot(aes(x = PC1, y = PC2)) +
geom_point() +
theme_bw()
```

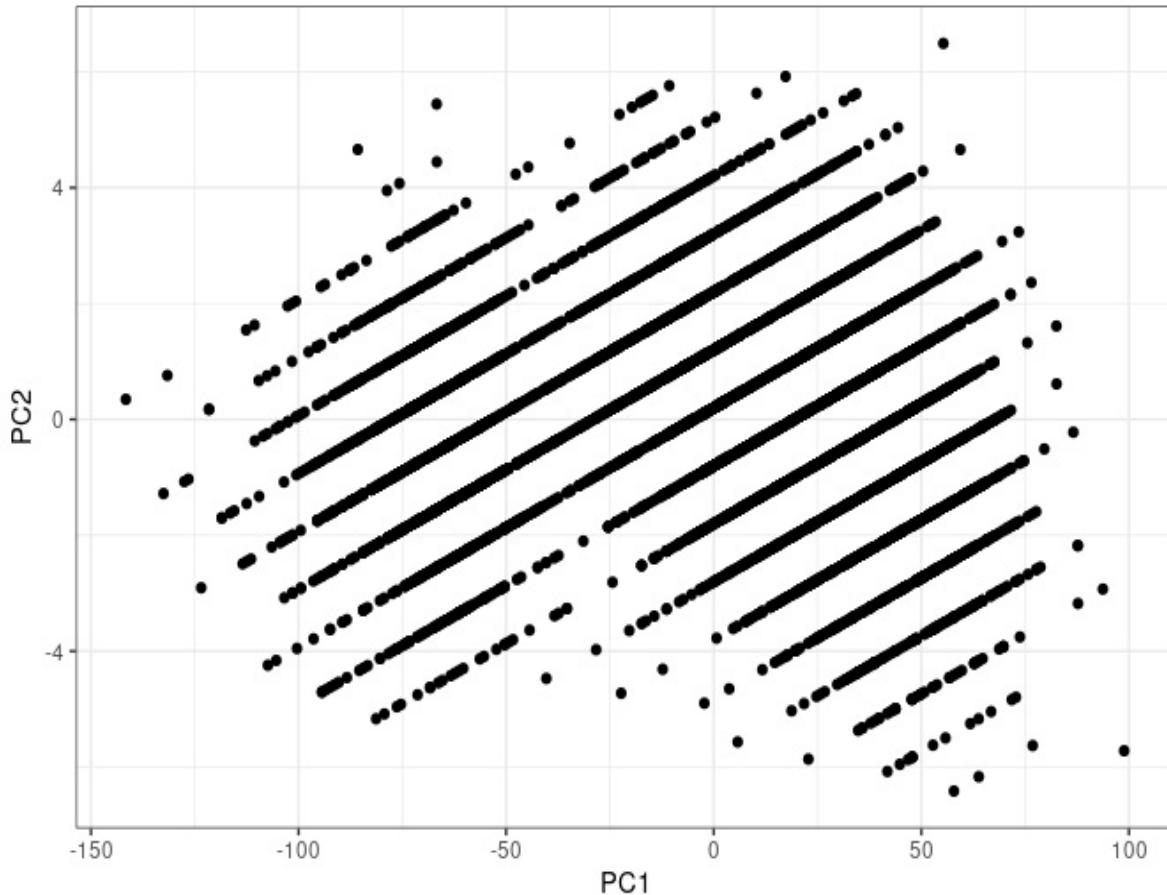


Figure 8-11. Scatterplot of the PCA rotation for weight and height with ggplot2.

There's a lot going on in these figures. First, compare [Figure 8-1](#) to [Figure 8-10](#) or [Figure 8-2](#) to [Figure 8-11](#). If you are good at spatial pattern recognition, you might see that the figures have the same data, just rotated (try looking at the outliers that fall away from the main crowd, you might be able to see how these data points moved). That's because PCA rotates data to create fewer dimensions of data. Because this example only has two dimensions, the pattern is visible in the data.

You can get these rotation values by looking at the `components_` in Python or by printing the fitted PCA in R. Although you will not use them more in this book, these have uses in machine learning when people need to convert data in

and out of the PCA space. In Python, use this code to extract the rotation values:

```
## Python
pca_wt_ht.components_
array([[ 0.9991454 ,  0.04133366],
       [-0.04133366,  0.9991454 ]])
```

In R, use this code to extract the rotation values:

```
## R
print(pca_fit_wt_ht_r)
Standard deviations (1, ... , p=2):
[1] 45.319497 1.870442

Rotation (n x k) = (2 x 2):
          PC1        PC2
Wt -0.99914540 -0.04133366
Ht -0.04133366  0.99914540
```

Your values may be different signs than our values (for example, if we have -0.999 and you have 0.999), but that's okay and varies randomly. For example, Python's PCA has three positive numbers with the book's numbers, whereas R's PCA has the same numbers as being the opposite with three negative numbers. These numbers say that you take a player's weight and multiply it by 0.999 and add 0.041 times the player's height to create the first principal component.

Why are these values so different? Look back again at [Figure 8-10](#) and [Figure 8-11](#). Notice how the x-axis and y-axis have vastly different scales. This can also cause different input features to have different levels of influence. Also, unequal numbers sometimes cause computational problems.

In the next section, you will put all features on the same unit level by scaling the inputs. *Scaling* refers to transforming a feature, usually to have a mean of zero and standard deviation of 1. Thus, the different units and magnitudes of the features no longer are important after scaling.

PCA on All Data

Now, apply R and Python's built-in algorithms to perform the PCA analysis on all data. This will help you to create new, and fewer, predictor variables that are

independent of each other. First, scale that data and then run the PCA. In Python:

```
## Python
from sklearn.decomposition import PCA

scaled_combine_knn_py = (
    combine_knn_py[col_impute] -
    combine_knn_py[col_impute].mean()) / \
    combine_knn_py[col_impute].std()

pca = PCA(svd_solver="full")
pca_fit_py = \
    pca.fit_transform(scaled_combine_knn_py)
```

Or in R:

```
## R
scaled_combine_knn_r <-
  scale(combine_knn_r |> select(Ht:Shuttle))

pca_fit_r <-
  prcomp(scaled_combine_knn_r)
```

The object `pca_fit` is more of a model object than it is a data object. It has some interesting nuggets in there. For one, you can look at the weights for each of the principal components. In Python:

```
## Python
rotation = pd.DataFrame(pca.components_, index=col_impute)
print(rotation)

          0         1         2       ...         5         6
7
Ht      0.280591  0.393341  0.390341   ... -0.367237  0.381342
0.377843
Wt      0.506953  0.273279 -0.063500   ...  0.359464 -0.110215
-0.130109
40yd    -0.709435 -0.001356 -0.082813   ... -0.096306  0.068683
-0.019891
Vertical -0.203781  0.033044  0.012393   ...  0.296674  0.523851
0.509379
Bench    -0.142324  0.161150  0.593645   ... -0.369323 -0.035026
-0.428910
Broad Jump 0.206559 -0.080594 -0.613440   ... -0.641948  0.277464
-0.094715
3Cone    -0.005106 -0.044482  0.027751   ... -0.298070 -0.677284
0.620678
```

```

Shuttle      -0.237684  0.857359 -0.327257   ...   0.035926 -0.162377
-0.047622

[8 rows x 8 columns]

```

Or in R:

```

## R
print(pca_fit_r$rotation)
          PC1        PC2        PC3        PC4        PC5
Ht      -0.2797884  0.4656585  0.747620897  0.21562254 -0.06128240
Wt      -0.3906321  0.2803488  0.002635803 -0.04180851  0.14466721
40yd    -0.3937993 -0.0994878  0.045495814 -0.01403113  0.48636319
Vertical 0.3456004  0.4186011  0.002756780 -0.53609337  0.54959455
Bench   -0.2668254  0.6109690 -0.642865102  0.18678232 -0.16950424
Broad Jump 0.3674448  0.3388903  0.139855673 -0.31774247 -0.43822503
3Cone   -0.3823998 -0.1115644 -0.060381468 -0.51566448  0.04602633
Shuttle -0.3770497 -0.1373520  0.049975069 -0.51225797 -0.46240812
          PC6        PC7        PC8
Ht       0.1394355 -0.164300591 -0.22194154
Wt      -0.1553924  0.089682873  0.84494838
40yd    -0.4083595  0.543622221 -0.36595878
Vertical 0.3348374  0.043881290 -0.04282478
Bench   0.0604662 -0.009981142 -0.27362455
Broad Jump -0.6246519  0.216613922 -0.02156424
3Cone   -0.2978993 -0.675688165 -0.15604839
Shuttle 0.4415283  0.404889985 -0.03684955

```

WARNING

PCA components are based upon a mathematical property of matrices called Eigen values and Eigen vectors. These are scalar and your PC's might be have opposite signs compared to our examples (for example, if our HT for PC1 is negative, yours might be positive.). Not to worry. Just note that is why the signs are different. Also, this appears to have occurred with R and Python when we were writing this book.

Notice that the first principal component weighs forty-yard dash and weight about the same (with factor weight of -0.39). Notice that most of the non-size metrics that are “better when smaller” have a negative weight (forty-yard dash, the agility drills), while those that are better when bigger (vertical and broad jump) are positive.

NOTE

Our Python and R examples start to diverge slightly because of differences in the imputation methods and PCA across the two languages. However, the qualitative results are the same.

This is a good sign that you're onto something. You can look at how much of the proportion of variance is explained by each principal component. In Python:

```
## Python
print(pca.explained_variance_)
[5.60561713 0.83096684 0.62448842 0.37527929 0.21709371 0.13913206
 0.12108346 0.08633909]
```

Or in R look at the standard deviation squared:

```
## R
print(pca_fit_r$sdev^2)
[1] 5.67454385 0.84556662 0.61894619 0.35175168 0.19651463 0.11815058
0.11166060
[8] 0.08286586
```

Notice here that, as expected, the first principal component handles a significant amount of the variability in the data, but subsequent PCs do have some influence over it as well. If you take these standard deviations in R, convert them to variances by squaring them (such as PCA1^2) and then dividing by the sum of all variances, you can see the percent variance explained by each axis. Python's PCA includes this without extra math.

In Python:

```
## Python
pca_percent_py = \
    pca.explained_variance_ratio_.round(4) * 100
print(pca_percent_py)
[70.07 10.39  7.81  4.69  2.71  1.74  1.51  1.08]
```

Or, in R:

```
## R
pca_var_r <- pca_fit_r$sdev^2
pca_percent_r <-
```

```

    round(pca_var_r / sum(pca_var_r) * 100, 2)
print(pca_percent_r)
[1] 70.93 10.57  7.74  4.40  2.46  1.48  1.40  1.04

```

Now, to access the actual PCs, deploy the following code to get something you can more readily use. In Python:

```

## Python
pca_fit_py = pd.DataFrame(pca_fit_py)
pca_fit_py.columns = \
    ["PC" + str(x + 1) for x in range(len(pca_fit_py.columns))]

combine_knn_py = \
    pd.concat([combine_knn_py, pca_fit_py], axis=1)

```

Or in R:

```

## R
combine_knn_r <-
  combine_knn_r |>
  bind_cols(pca_fit_r$x)

```

The first thing to do from here is graph the first few principal components to see if you have any natural clusters emerging in them. In Python create [Figure 8-12](#):

```

## Python
sns.scatterplot(data=combine_knn_py,
                 x="PC1",
                 y="PC2");
plt.show();

```

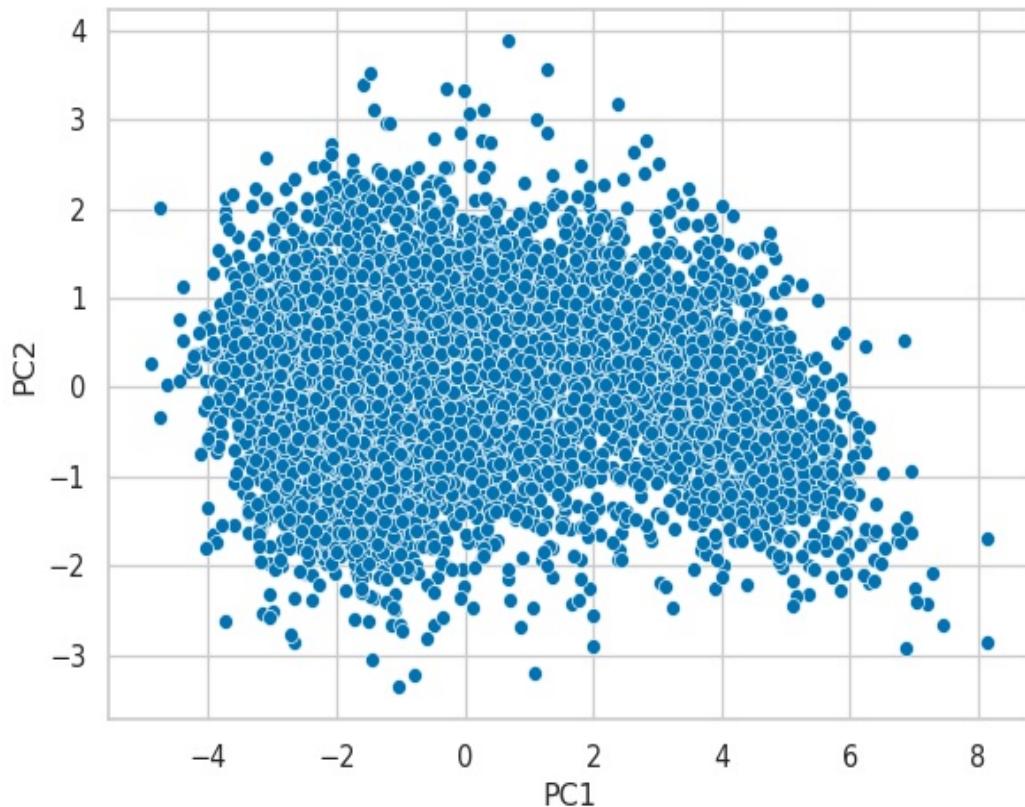


Figure 8-12. Plot of first two PCA components with seaborn.

In R create Figure 8-13:

```
## R
ggplot(combine_knn_r, aes(x = PC1, y = PC2)) +
  geom_point() +
  theme_bw() +
  xlab(paste0("PC1 = ", pca_percent_r[1], "%")) +
  ylab(paste0("PC2 = ", pca_percent_r[2], "%"))
```

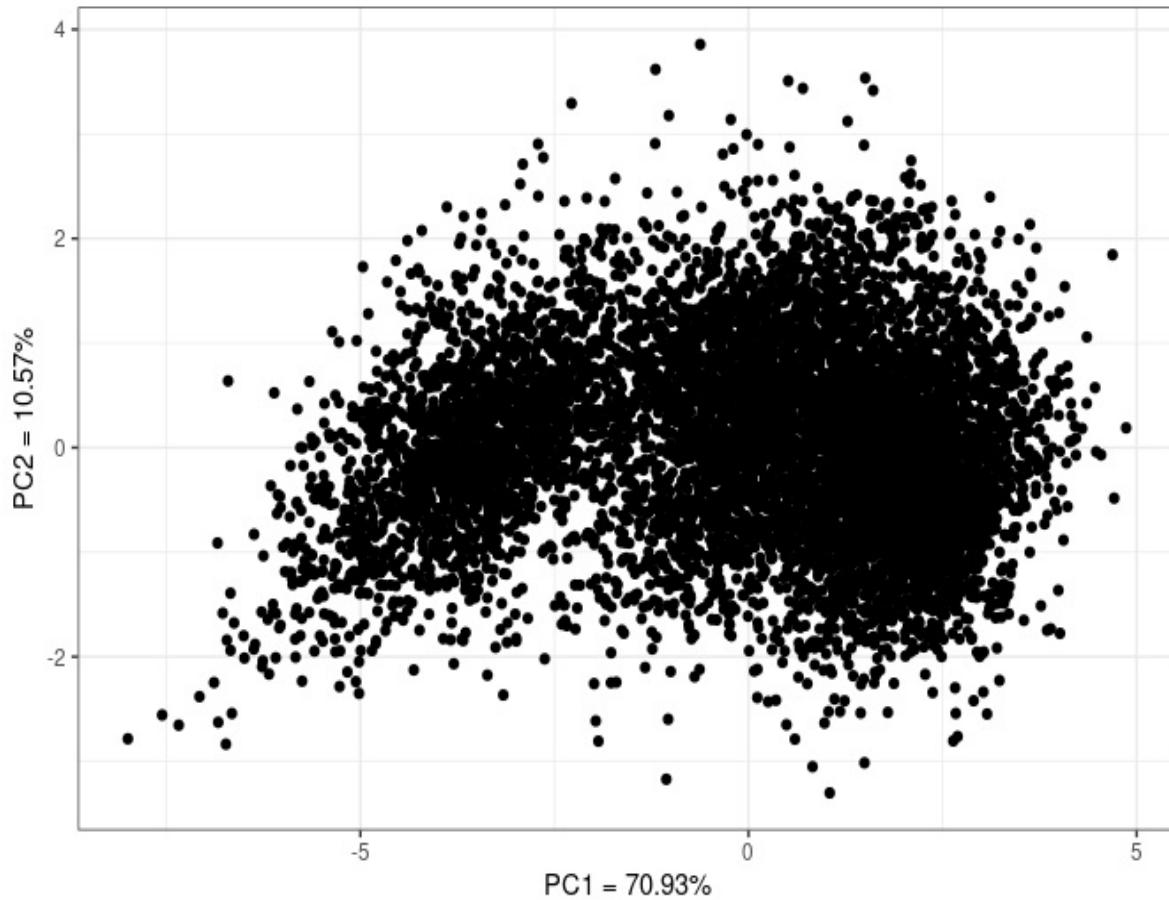


Figure 8-13. Plot of first two PCA components with ggplot2.

NOTE

Because PCs are based upon eigen values, your figures might be flipped from each of our example figures. For example, our R and Python figures are mirror images of each other.

You can already see two clusters! What if you use some more of the data to reveal other possibilities? Shade each of the points by the value of the third principal component. In Python create [Figure 8-14](#):

```
## Python
sns.scatterplot(data=combine_knn_py,
                 x="PC1",
                 y="PC2",
                 hue="PC3");
plt.show();
```

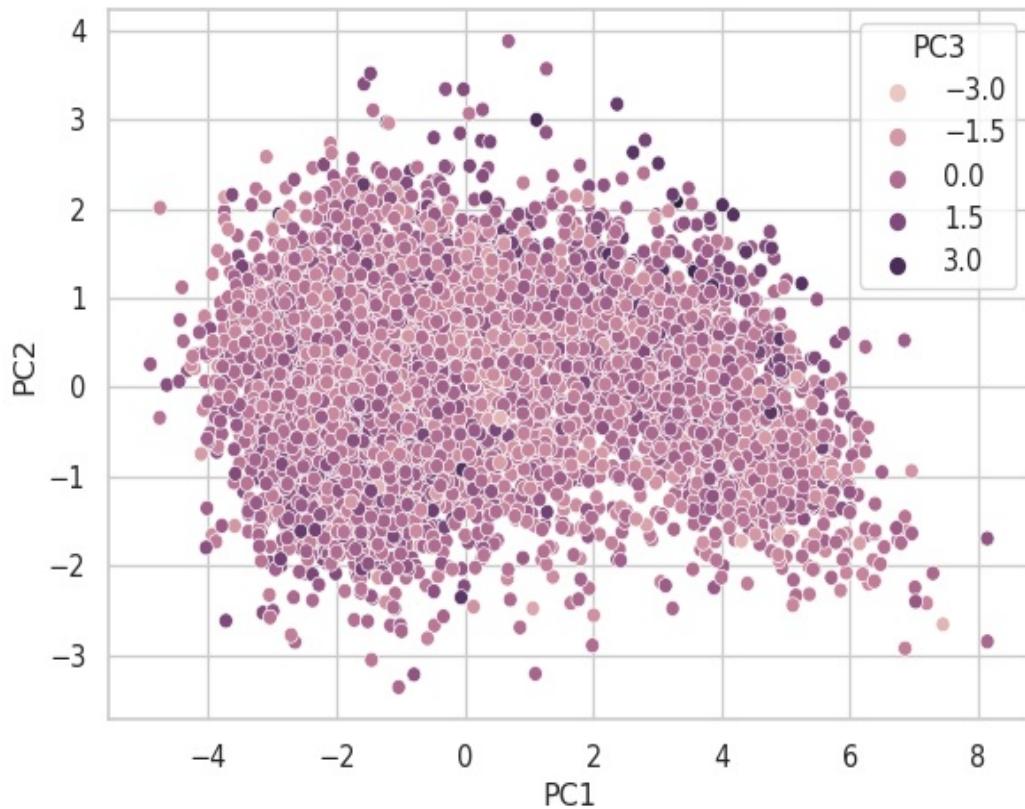


Figure 8-14. Plot of first two PCA components with `seaborn` with the third PCA component as the point color.

In R create Figure 8-15:

```
## R
ggplot(combine_knn_r,
       aes(x = PC1, y = PC2, color = PC3)) +
  geom_point() +
  theme_bw() +
  xlab(paste0("PC1 = ", pca_percent_r[1], "%")) +
  ylab(paste0("PC2 = ", pca_percent_r[2], "%")) +
  scale_color_continuous(
    paste0("PC3 = ", pca_percent_r[3], "%"),
    low="skyblue", high="navyblue")
```

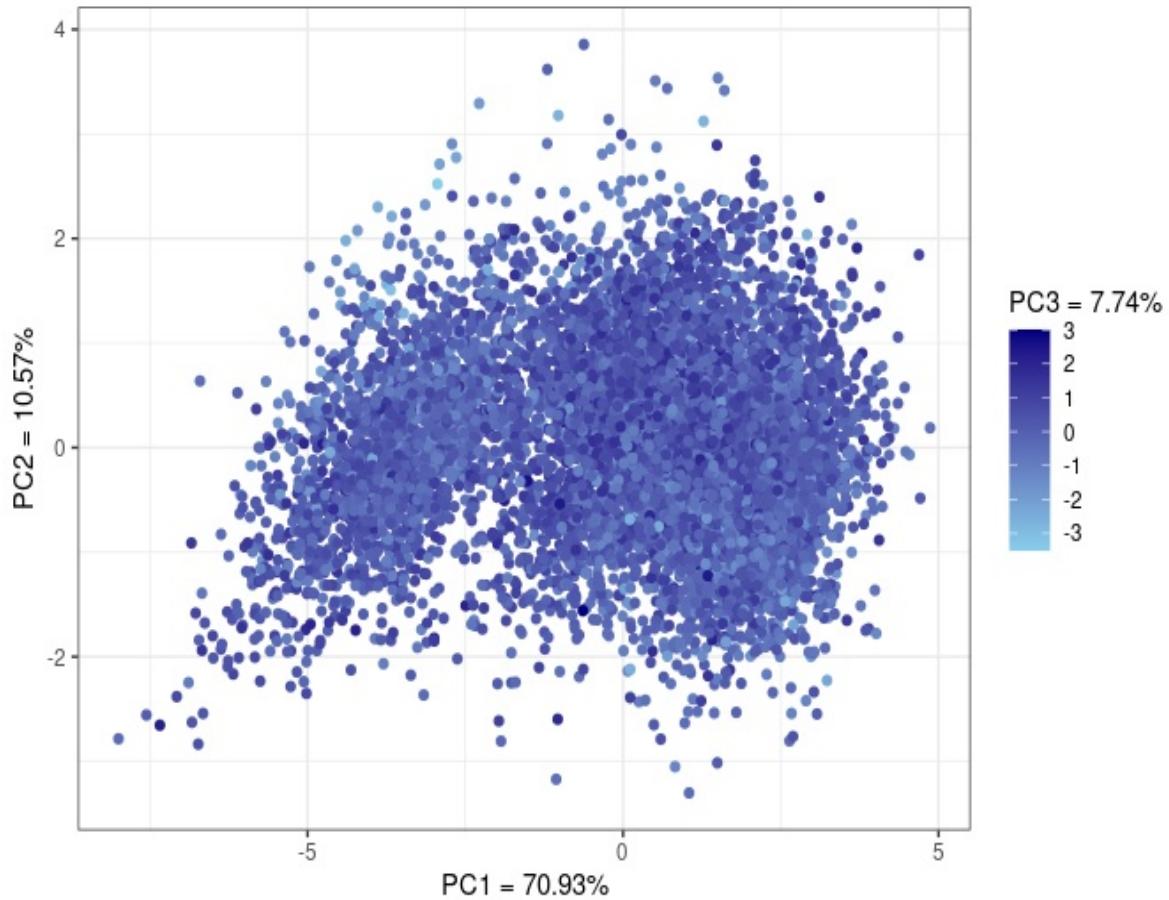


Figure 8-15. Plot of first two PCA components with `ggplot2` with the third PCA component as the point color.

Interesting. Looks like players on the edges of the plot have lower values of PC3, corresponding to darker shades. What if you shade by position?

In Python create [Figure 8-16](#):

```
## Python
sns.scatterplot(data=combine_knn_py,
                 x="PC1",
                 y="PC2",
                 hue="Pos");
plt.show();
```

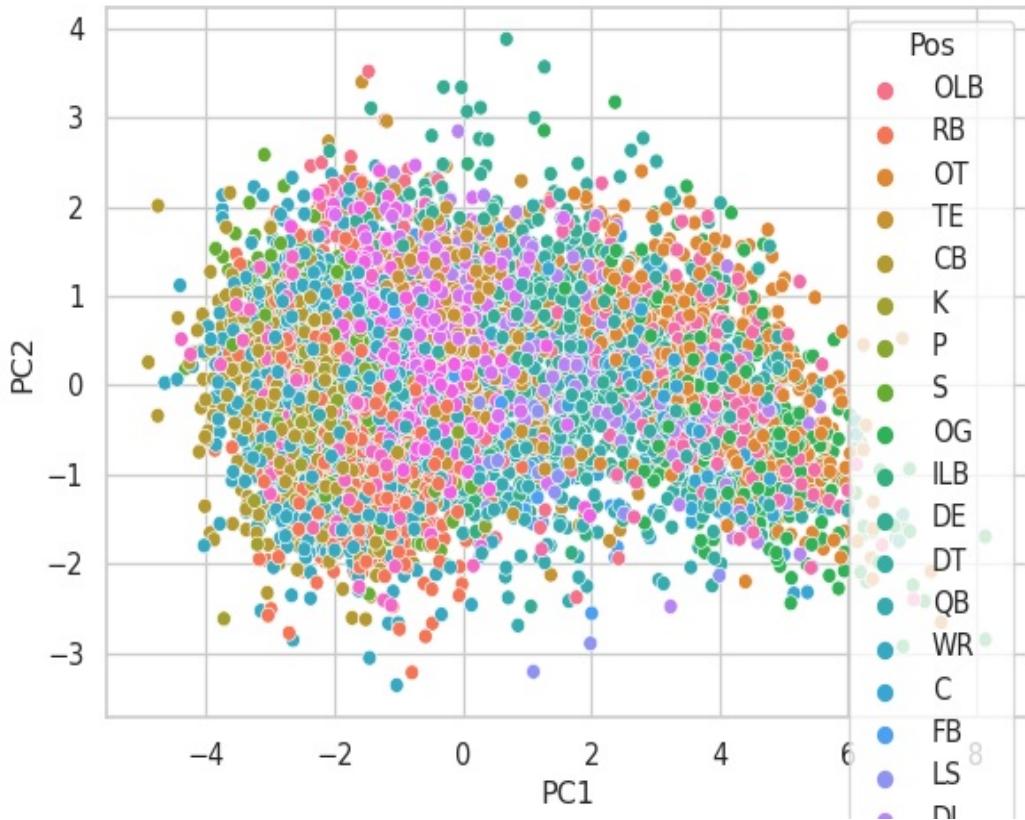


Figure 8-16. Plot of first two PCA components with seaborn with the point player position as the color

In R use a colorblind friendly palette to create Figure 8-17:

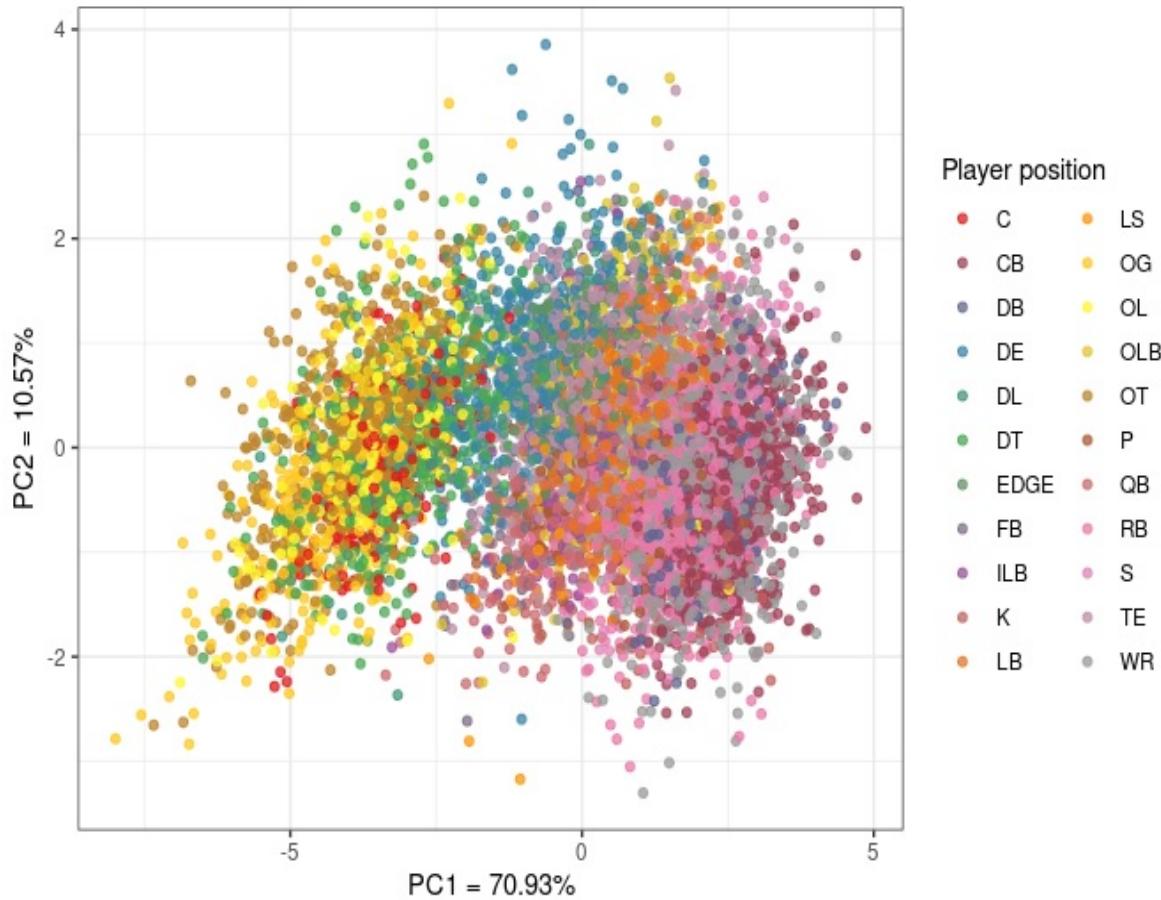


Figure 8-17. Plot of first two PCA components with `ggplot2` with the point player position as the color

Okay, this is fun. It does look like the positions create clear groupings in the data. Upon first blush, it looks like you can split this data into about five-to-seven clusters.

TIP

About 8% of men and 0.5% of women in the US have color vision deficiency (more commonly known as colorblindness). This ranges from only seeing black-and-white to, more commonly, not being able to tell all colors apart. For example, Richard has trouble with reds and greens, which also make purple colors hard for him to see. Hence, try to pick colors more people can use. Tools such as [Color Oracle](#) and [Sim Daltonism](#) let you test your figures and see them like a person with colorblindness.

Clustering Combine Data

The clustering algorithm you're going to use here is k-means clustering, which

aims to partition a dataset into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster.

We break this section into two units because the numerical methods diverge. This does not mean either method is “wrong” or “better” than the other. Instead, this helps you see that methods can simply be “different”. Also, understanding and interpreting multivariate statistical methods (especially unsupervised methods) can be subjective. One of Richard’s professors at Texas Tech would describe this process as similar to reading tea leaves because you can often find whatever pattern you are looking for if you are not careful.

WARNING

Despite our comparison of multivariate statistics being similar to ‘reading tea leaves’, the approaches are commonly used (and rightly so) because the methods are powerful and useful. You, as a user and modeler, however, need to understand this limitations if you are going to use the methods.

Clustering Combine Data in Python

WARNING

If you are coding along (and hopefully you are!), your results will likely be different than ours for the clustering. You will need to look at the results to see which cluster number corresponds to the groups on your computer.

To start with Python, use `kmeans` from the `scipy` package and fit for 6 centers (we set the `seed` to be `1234` so that we would get the same results each time we ran the code for the book):

```
## Python
from scipy.cluster.vq import vq, kmeans

k_means_fit_py = \
    kmeans(combine_knn_py[['PC1', 'PC2']], 6, seed = 1234)
```

Next, attach these clusters to the dataset:

```

## Python
combine_knn_py["cluster"] = \
    vq(combine_knn_py[["PC1", "PC2"]], k_means_fit_py[0])[0]

combine_knn_py.head()
   Unnamed: 0          Player  Pos  ...      PC7      PC8 cluster
0            0  John Abraham  OLB  ... -0.146522  0.292073     3
1            1  Shaun Alexander  RB  ... -0.073008  0.060237     1
2            2  Darnell Alford  OT  ... -0.491523 -0.068370     0
3            3  Kyle Allamon  TE  ...  0.328718 -0.059768     2
4            4  Rashard Anderson  CB  ... -0.674786 -0.276374     1

[5 rows x 24 columns]

```

Not much can be gleaned from the head of the data here. However, one thing you can do is see if the clusters do bring like positions and player types together. Looking at cluster 1:

```

print(
    combine_knn_py.query("cluster == 1")
    .groupby("Pos")
    .agg({"Ht": ["count", "mean"], "Wt": ["count", "mean"]})
)
   Ht                               Wt
   count        mean  count        mean
Pos
CB      219  72.442922  219  197.506849
DB       27  72.074074   27  201.074074
DE       13  75.384615   13  250.307692
EDGE      5  74.800000    5  247.400000
FB       8  72.500000    8  235.000000
ILB      20  73.700000   20  237.700000
K        9  73.777778    9  207.666667
LB       45  73.673333   45  234.486667
OLB      78  73.987179   78  236.397436
P        24  74.416667   24  206.041667
QB      40  74.450000   40  217.000000
RB     163  71.225767  163  216.932515
S       221  72.800905  221  209.330317
TE      20  75.450000   20  244.100000
WR     409  73.795844  409  207.871638

```

A little bit of everything here, but it's mostly players away from the ball; cornerbacks, safeties, and wide receivers, with some running backs mixed in. Among those position groups, these players are heavier players.

Depending upon the random number generator in your computer, different clusters will have different numbers, so be weary of comparing across clustering regimes. Now, let's look at the summary for all clusters using a plot. In Python, create [Figure 8-18](#):

```
## Python
combine_knn_py_cluster = \
    combine_knn_py\
    .groupby(["cluster", "Pos"])\\
    .agg({"Ht": ["count", "mean"],
          "Wt": ["mean"]})
)

combine_knn_py_cluster.columns = \
    list(map("_".join, combine_knn_py_cluster.columns))

combine_knn_py_cluster.reset_index(inplace=True)

combine_knn_py_cluster\
    .rename(columns={"Ht_count": "n",
                    "Ht_mean": "Ht",
                    "Wt_mean": "Wt"},\
            inplace=True)

combine_knn_py_cluster.cluster = \
    combine_knn_py_cluster.cluster.astype(str)

sns.catplot(combine_knn_py_cluster, x="n", y="Pos",
            col="cluster", col_wrap=3, kind="bar");
plt.show();
```

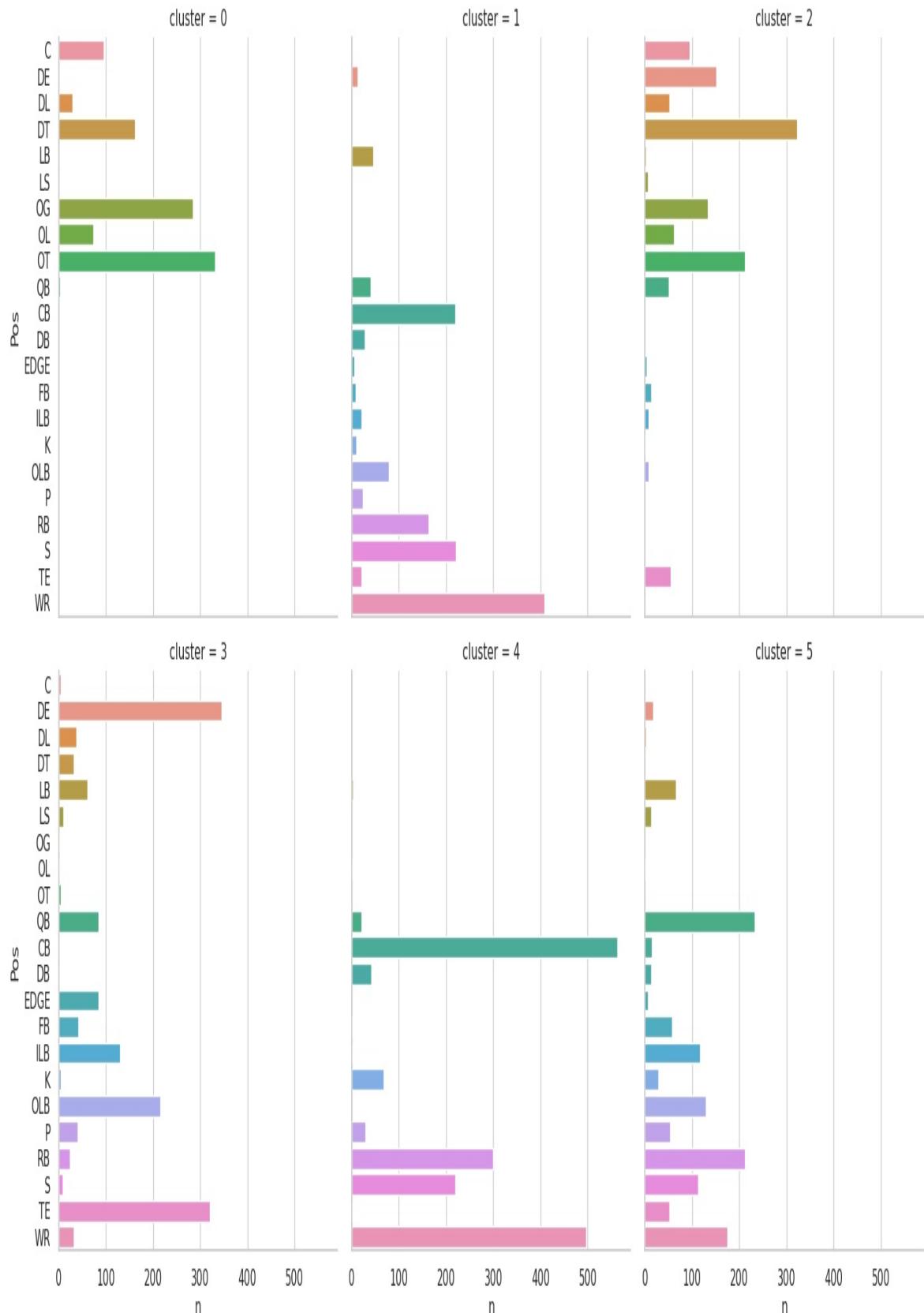


Figure 8-18. Plot of positions by cluster with seaborn.

Here, cluster 0 is largely bigger players, like offensive linemen and interior defensive linemen. Cluster 2 includes similar positions as cluster 0, while adding defensive ends and tight ends, which are also represented in great measure in cluster 3, along with outside linebackers. Cluster 1 was talked about above. Cluster 4 has a lot of the same positions as cluster 1, but more defensive backs and wide receivers (we'll take a look at size below). Cluster 5 includes a lot of quarterbacks, as well as a decent number of other *skill positions* (*skill positions* in football are those that typically hold the ball and are responsible for scoring).

Let's look at the summary by cluster to compare weight and height:

```
## Python
combine_knn_py_cluster\
    .groupby("cluster")\
    .agg({"Ht": ["mean"], "Wt": ["mean"]})
          Ht           Wt
          mean         mean
cluster
  0      75.866972  293.708339
  1      73.631939  223.254368
  2      74.966517  272.490225
  3      75.230958  250.847219
  4      71.099940  205.290840
  5      73.098379  229.605847
```

As hypothesized above, cluster 1 and cluster 4, while having largely the same positions represented, are such that cluster 1 includes much bigger players in terms of height and weight. Similarly for clusters 0 and 2.

Clustering Combine Data in R

In start with R, use the `kmeans` function that comes in the `stats` package with R's core packages. Here, `iter.max` is the maximum number of iterations allowed to find the clusters, and `centers` is the number of clusters. This is needed because the algorithm requires multiple attempts or *iterations* to fit the model. This is accomplished using the following script (we have you set R's random seed with `set.seed(123)` so that you get consistent results):

```
## R
```

```

set.seed(123)
k_means_fit_r <-
  kmeans(combine_knn_r |> select(PC1, PC2),
         centers = 6, iter.max = 10)

```

Next, attach these clusters to the dataset:

```

## R
combine_knn_r <-
  combine_knn_r |>
  mutate(cluster = k_means_fit_r$cluster)

combine_knn_r |>
  select(Pos, Ht:Shuttle, cluster) |>
  head()
# A tibble: 6 × 10
  Pos      Ht     Wt `40yd` Vertical Bench `Broad Jump` `3Cone` Shuttle
  <chr> <dbl> <dbl>    <dbl>    <dbl> <dbl>       <dbl> <dbl> <dbl>
1 OLB      76    252     4.55    38.5   23.5      124    6.94   4.22
2 RB       72    218     4.58    35.5    19        120    7.07   4.24
3 OT       76    334     5.56    25      23        94     8.48   4.98
4 TE       74    253     4.97    29      21       104    7.29   4.49
5 CB       74    206     4.55    34      15       123    7.18   4.15
6 K        70    202     4.55    36      16      120.    6.94   4.17

```

As with the Python example, not much can be gleaned from the head of the data here, other than the fact that in R the index begins at 1, rather than 0. Looking at the first cluster:

```

## R
combine_knn_r |>
  filter(cluster == 1) |>
  group_by(Pos) |>
  summarize(n = n(), Ht = mean(Ht), Wt = mean(Wt)) |>
  arrange(-n) |>
  print(n = Inf)
# A tibble: 21 × 4
  Pos      n     Ht     Wt
  <chr> <dbl> <dbl> <dbl>
1 OLB      6    76.0  4.55
2 RB       4    72.0  4.58
3 OT       3    76.0  5.56
4 TE       1    74.0  4.97
5 CB       4    74.0  4.55
6 K        4    70.0  4.55
7 LB       1    74.0  4.55
8 FB       1    74.0  4.55
9 DL       1    74.0  4.55
10 DLB     1    74.0  4.55
11 DLB    12    74.0  4.55
12 DLB    12    74.0  4.55
13 DLB    12    74.0  4.55
14 DLB    12    74.0  4.55
15 DLB    12    74.0  4.55
16 DLB    12    74.0  4.55
17 DLB    12    74.0  4.55
18 DLB    12    74.0  4.55
19 DLB    12    74.0  4.55
20 DLB    12    74.0  4.55
21 DLB    12    74.0  4.55

```

```

<chr> <int> <dbl> <dbl>
1 QB      236   74.8  223.
2 TE      200   76.2  255.
3 DE      193   75.3  266.
4 ILB     127   73.0  242.
5 OLB     116   73.5  242.
6 FB      65    72.3  247.
7 P       60    74.8  221.
8 RB      49    71.1  226.
9 LB      43    72.9  235.
10 DT     38    73.9  288.
11 WR     29    74.9  219.
12 LS      28    73.9  241.
13 EDGE    27    75.3  255.
14 K       23    73.2  213.
15 DL     22    75.5  267.
16 S       11    72.6  220.
17 C       3     74.7  283
18 OG     2     75.5  300
19 CB     1     73    214
20 DB     1     72    197
21 OL     1     72    238

```

This cluster has its biggest representation among quarterbacks, tight ends and defensive ends. This makes some sense football-wise, as many tight ends are converted quarterbacks, like former Vikings tight end (and head coach) Mike Tice, who was a quarterback at Maryland before converting to the pivot as a professional.

Depending upon the random number generator in your computer, different clusters will have different numbers: In R, create [Figure 8-19](#):

```

## R
combine_knn_r_cluster <-
  combine_knn_r |>
  group_by(cluster, Pos) |>
  summarize(n = n(), Ht = mean(Ht), Wt = mean(Wt),
            .groups="drop")

combine_knn_r_cluster |>
  ggplot(aes(x = n, y = Pos)) +
  geom_col(position='dodge') +
  theme_bw() +
  facet_wrap(vars(cluster)) +
  theme(strip.background = element_blank()) +
  ylab("Position") +

```

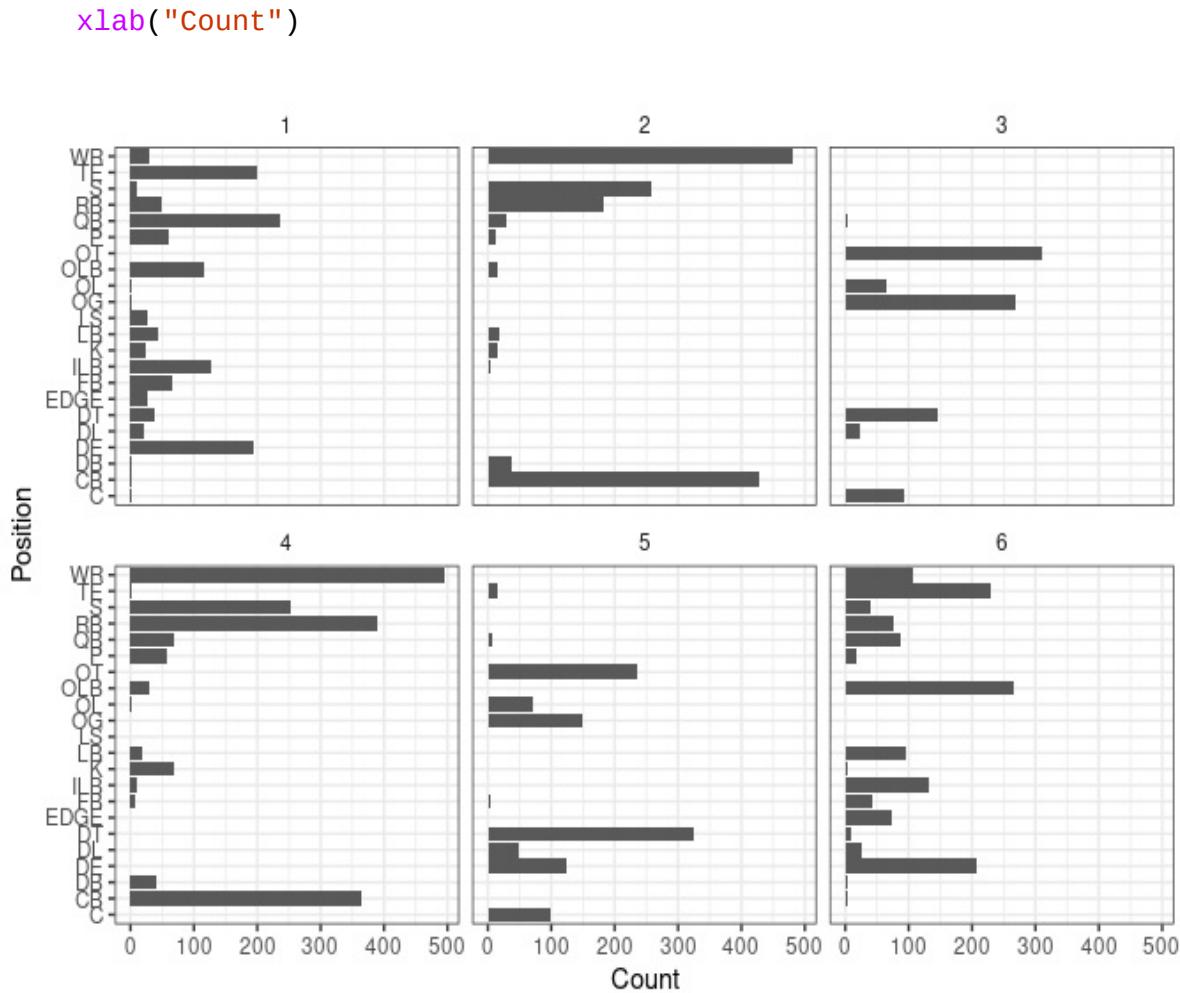


Figure 8-19. Plot of positions by cluster with *ggplot2*.

Here you get a similar result as above for the Python, with offensive and defensive linemen grouped together in some clusters, while skill position players find other clusters more frequently. Let's look at the summary by cluster to compare weight and height. In R use:

```
## R
combine_knn_r_cluster |>
  group_by(cluster) |>
  summarize(ave_ht = mean(Ht),
            ave_wt = mean(Wt))
# A tibble: 6 × 3
  cluster  ave_ht  ave_wt
    <int>   <dbl>   <dbl>
1       1    73.8    242.
2       2    72.4    214.
3       3    75.6    291.
```

4	4	71.7	211.
5	5	75.7	281.
6	6	75.0	246.

Clusters 2 and 4 include players further away from the ball and hence smaller, while clusters 3 and 5 are bigger players playing along the line of scrimmage. Clusters 1 (described above) and 6 more “tweener” athletes that play positions like quarterback, tight end, outside linebacker, and in some cases, defensive end (*tweener* players are those who can play multiple positions well, but may or may not excel and be the best at any position).

Closing Thoughts on Clustering

Even this initial analysis shows the power of this approach, as after some minor adjustments to the data, you’re able to produce reasonable groupings of the players without first having to define the clusters beforehand. For players for whom you don’t have a ton of initial data, this can get the conversation started vis-a-vis comparables, fits, and other things. It can also weed out players that do not fit a particular coach’s scheme.

You can drill down even further once the initial groupings are made. Among just wide receivers, is this wide receiver a taller/slower type that wins on contested catches? Is he a shorter/shiftier player who wins with separation? Is he a unicorn of the mold of Randy Moss or Calvin Johnson? Does he make redundant players that are already on the roster - older players whose salaries the team might want to get rid of, or does he supplement the other players as a replacement for a player who has left through free agency or a trade? Arif Hasan discussed these traits for a specific coach as an example in [“Vikings Combine Trends: What might they look for in their offensive draftees?”](#) in *The Athletic*.

Clustering has been used on more advanced problems to group together things like a receiver’s pass routes. In this problem, you’re is using model-based curve clustering on the actual (x,y) trajectories of the players to do with math what companies like PFF have been doing with their eyes for some time - chart each play for analysis. As mentioned above, most old-school coaches and front office members are in favor of groupings, so methods like these will always have appeal in American football for that reason. Dani Chu and collaborators describe approaches such as route identification in [“Route identification in the National](#)

Football League” that also exists as an open-access [pre-print](#).

Some of the drawbacks of k-means clustering specifically are that it’s very sensitive to initial conditions and the number of clusters that are chosen. There are steps you can take - including random number seeding (as done above) - that can help reduce these issues, but you need to be known when going in. As with everything, vigilance is required as new data comes in - that outputs are not too drastically altered with each passing year. Some years, because of evolution in the game, you might have to add or delete a cluster, but this decision should be made after thorough and careful analysis of the downstream effects of doing so.

Data Science Tools Used in This Chapter

In this chapter you learned how to:

- Adapt web scraping tools from [Chapter 7](#) for a slightly different web page.
- Use PCA to reduce the number of dimensions.
- Use cluster analysis to examine data for groups.

Exercises

- What happens when you do PCA analysis on the original data, with all of the NAs included? How does this affect your future football analytics workflow?
- Perform the k-means clustering in this chapter with the first three principle components. Do you see any differences? The first four?
- Perform the k-means clustering in this chapter with five and seven clusters. How does this change the results?
- What other problem in this book would be enhanced with a clustering approach?

Suggested Reading

Many basic statistics books cover the methods presented in this chapter. Two example books include:

- *Essential Math for Data Science* by Thomas Nield (O'Reilly Media, 2021) provides a gentle introduction to statistics as well as mathematics for applied data scientists.
- *R for Data Science* by Hadley Wickham and Garrett Grolemund (O'Reilly Media, 2017) provides an introduction to many tools and methods for applied data scientists.

Chapter 9. Advanced Tools and Next Steps

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

This book has focused on the basics of football analytics using Python and R. We personally use both on a regular basis. However, we also use tools beyond these two programming languages. For people who want to keep growing, you will need to leave your comfort zone. This chapter provides an overview of other tools we use. We start with modeling tools we use, but have not mentioned yet either because the topics were too advanced, or because we could not find public data that would easily allow you to code along.

We then move on to computer tools. The topics are both disjointed and interwoven at the same time. That is to say, you can learn one skill independently, but often, using one skill works best with other skills. As a football comparison, a linebacker needs to be able to defend the run, rush the passer, and cover players who are running pass routes, often in the same series of a game. Some skills (such as the ability to read a play) and player traits (such as speed) will help with all three linebacker situations, but they are often drilled separately. The most valuable players are great at all three.

This chapter is based upon the authors' experiences working as data scientists as

well as an article Richard wrote for natural resource managers (“[Paths to computational fluency for natural resource educators, researchers, and managers](#)”). We suggest you learn the topics in the order we present them, and we list reasons why in [Table 9-1](#). Once you gain some comfort with a skill, move on to another area. Eventually, you’ll make it back to a skill area and see where to grow in that area. As you learn more technologies you can become better at learning new technologies!

TIP

Emily Robinson and Jacqueline Nolis have written a book, *Build a Career in Data Science* (Manning, 2020) that provides broader coverage of skills for a career in a data science.

Table 9-1. Table 1: Advanced tools, our reasons for using them, and example products.

Tool	Reason	Examples
Command line	Efficiently and automatically work with your operating system, use other tools that are command line-only.	Bash, PowerShell, zsh
Version Control	Keep track of changes to code, collaborate on code, share and publish code.	Git, Apache Subversion (SVN), Mercurial
Linting	Clean code, internal consistency for style, reduce errors, and improve quality.	Pylint, lintr, Black
Package creation and hosting	Reuse your own code, share your code internally or externally, and more easily maintain your code.	Pip, Conda, CRAN

Environment	Reproducible results, taking models to production or the cloud, and ensuring same tools across collaborations.	Conda, Docker, Poetry
Interactives and reports	Allow other people to explore data without knowing how to code, prototype tools before handing off to DevOps.	Jupyter Notebooks, Shiny, Quarto
Cloud	Deployment of tools, advanced computing resources, sharing data.	Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform

TIP

All of the advanced tools we mention have free documentation either included with them or online. However, finding and using this documentation can be hard and often requires the user to locate the diamonds in the rough. Hence, paid tutorials and resources such as books often, but not always, offer you a quality product. If you are broke grad student, you might want to spend your time diving through the free resources to find the gems. If you are working professional with kids and not much time, you probably want to pay for learning resources. Basically, finding quality learning materials comes down to time versus money.

Advanced Modeling Tools

Within this book, we have covered a wide range of models. For many people, these tools will be enough to advance your football analytics game. However, other people will want to go farther and push the envelope. In this section, we describe some methods we use on a regular basis. Notice that many of these topics are interwoven. Hence, learning one topic might lead you to learn about other topic as well.

Time Series Analysis

Football data, especially feature rich, high-resolution data within games, lends itself to look at trends through time. *Time series analysis* estimates at trends through time. The methods are commonly used in finance, with applications in other fields such as ecology, physics, and social sciences. Basically, these models can provide better estimations when past observations are important for future predictions (also known as *auto-correlations*). Here are some resources we've found helpful:

- *Time Series Analysis and Its Application*, 4th Edition by Robert H. Shumay and David S. Stofer (Spring, 2017). Provides a detailed introduction for time series analysis, using R.
- *Practical Time Series Analysis* by Aileen Nielsen (O'Reilly, 2019) provides a gentler introduction to time series analysis with a focus on application, especially to machine learning.
- **Prophet** by Facebook's Core Data Science is a time-series modeling tool that can be powerful when used correctly.

Multivariate Statistics Beyond PCA

Chapter 8 provided a brief introduction to multivariate methods such as PCA and clustering. These two methods are the tip of the iceberg. Different methods exist, such as redundancy analysis (RDA), that allow both multivariate predictor and response variables. These methods form the basis of many entry-level unsupervised learning methods because the methods find their own predictive groups. Additionally, PCA assumes *Euclidean distance* (the same distance you may or may not remember from the Pythagorean theorem, for example in two dimensions $c = \sqrt{(a^2 + b^2)}$). Other types of distances exist, and multivariate methods cover these. Lastly, many different classification method exit. For example, some multivariate methods extend to time series analysis, such as dynamic factor analysis (DFA).

Beyond direct application of these tools, understanding these methods will give you a firm foundation if you want to learn machine learning tools. Some books we learned from or think would be helpful include the following:

- *Numerical Ecology*, 3rd Edition, by P. Legendre, L. Legendre (Elsevier,

2012) provides a generally accessible overview of many different multivariate methods.

- *Analysis of multivariate time series using the MARSS package* vignette by E. Holmes, M. Scheuerell, and E. Ward comes with the MARSS package and may be found on the the [MARSS CRAN page](#). This detailed introduction describes how to do time series analysis with R on multivariate data.

Quantile Regression

Usually regression models the average (or mean) expected value. Quantile regression models other parts of a distribution. Specifically, a user specified quantile. Whereas a boxplot covered in “[Boxplots](#)” has predefined quantiles, the user specifies which quantile they want with quantile regression. For example, when looking at combine data, you might wonder how player speeds change through time. A traditional multiple regression would look at the average player through time. A quantile regression would help you see if the faster players get faster through time by looking. Quantile regression would also be helpful when looking at the draft data in [Chapter 7](#). Resources for learning about quantile regression include the package documentation:

- The [quantreg package](#) in R has a well written 21 page [vignette](#) that is great regardless of language.
- The [statsmodels package](#) in Python also has quantile regression [documentation](#).

Bayesian Statistics and Hierarchical Models

Up to this point, our entire use of probability has been built on the assumption that events occur based upon long-term occurrence, or frequency of events occurring. This type of probability is known as *frequentist* statistics. However, other views of probabilities exist.

Notably, a Bayesian perspective views the world in terms of degrees of belief or certainty. For example, a frequentist 95% confidence interval (CI) around a mean contains the mean 95% of the time, if you repeat your observations many, many,

many times like. Conversely, a Bayesian 95% credible interval (CrI) indicates a range for which you are 95% certain to contain the mean. It's a subtle difference, but important.

A Bayesian perspective begins with a prior understanding of the system, updates the prior using observed data, and then generates a posterior distribution. In practice, Bayesian methods offer three major advantages:

- They can fit more complicated models when other methods might not have enough data.
- They can include multiple sources of information more readily.
- A Bayesian view of statistics is what many people have, even if they do not know the name for it.

For example, consider picking which team will win. The prior information can either come from other data, your best guess, or any other source. If you are humble, you might think you will be right 50% of the time, and would guess you would get 2 games right and 2 games wrong. If you are over-confident, you might think you will be right 80% of the time, or get 8 games right and 2 games wrong. If you are under-confident, you might think you will be right 20% of the time and get 8 games wrong and 2 games right. This is your prior distribution.

For this example, a Beta distribution gives you the probability distribution given the number of “successes” and “failures”. Graphically, this gives you [Figure 9-1](#)

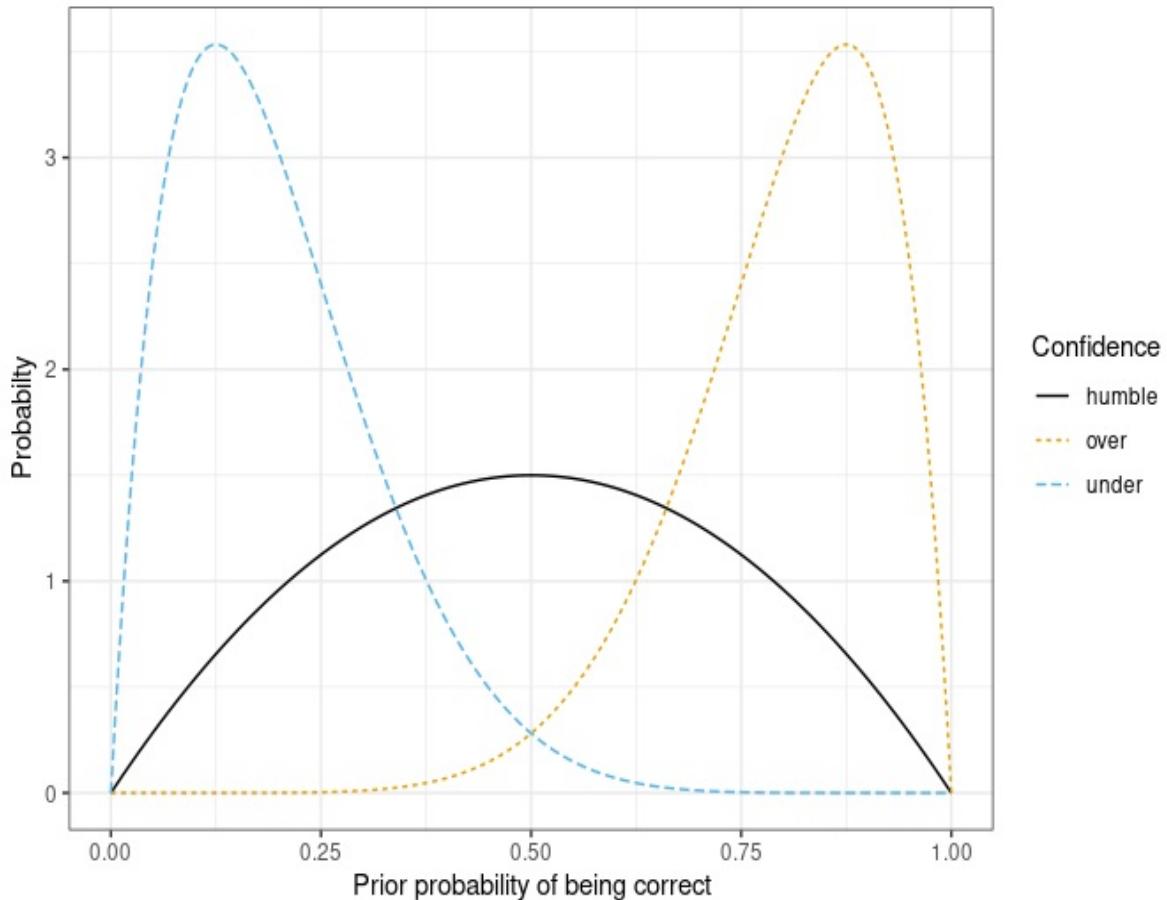


Figure 9-1. Prior distribution for predicting results of games.

After observing 50 games, perhaps you were correct for 30 games and wrong for 20 games. A frequentist would say you are correct 60% of the time (30/50). To a Bayesian, this is the observed likelihood. With a Beta distribution, this would be 60 successes and 40 failures. [Figure 9-2](#) shows the likelihood probability.

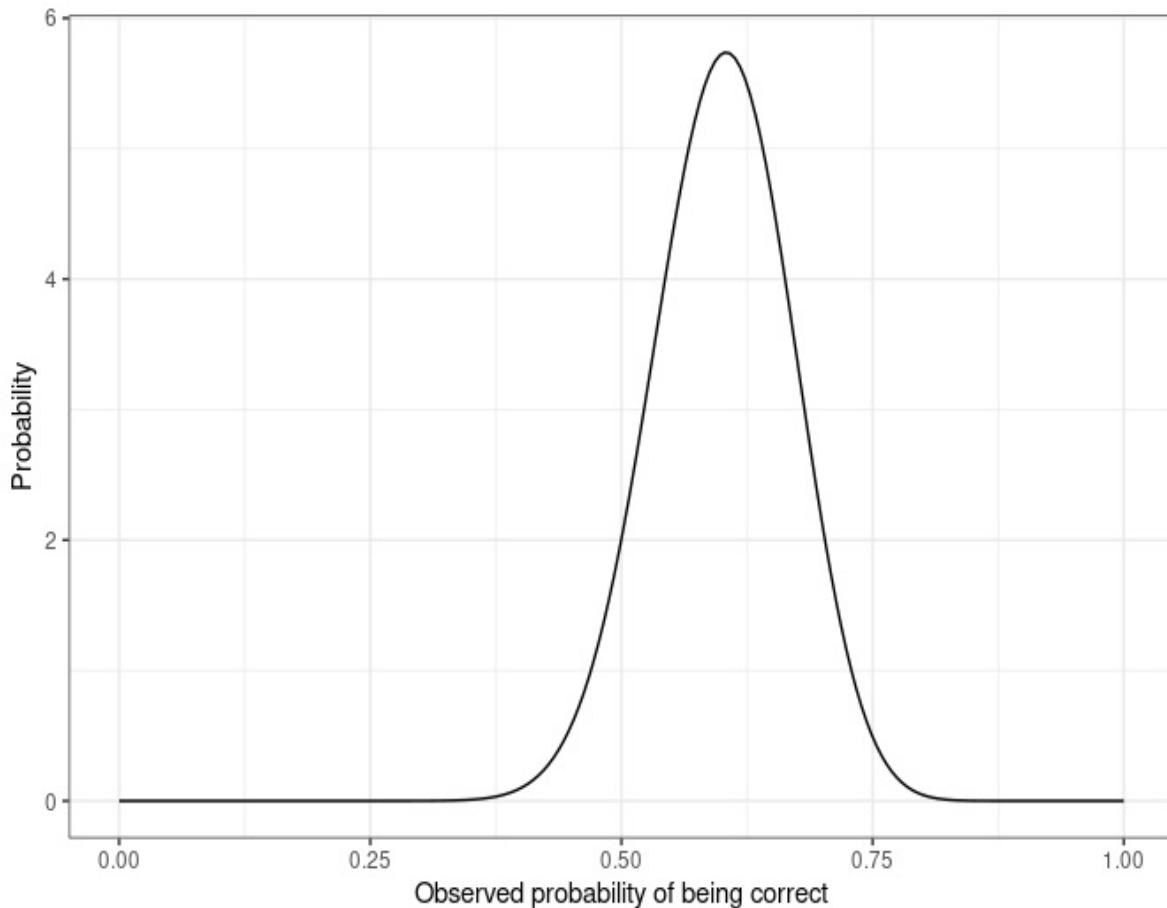


Figure 9-2. Likelihood distribution for predicting results of games.

Next, a Bayesian multiplies [Figure 9-1](#) by [Figure 9-2](#) to create the posterior [Figure 9-3](#). All three guesses are close, but the prior distribution informs the posterior.

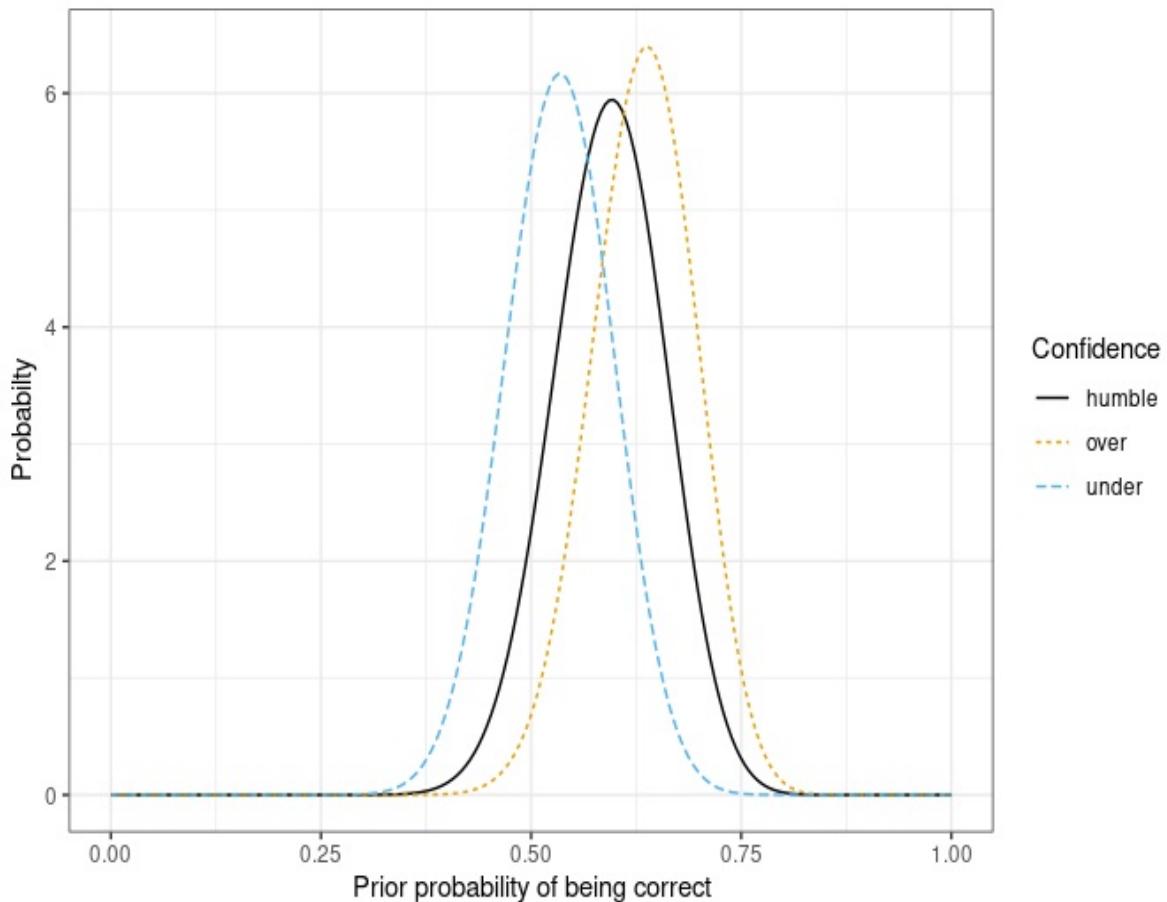


Figure 9-3. Posterior distribution for predicting results of games. Notice the influence of the prior distribution on the posterior.

This simple example illustrates how Bayesian methods work for an easy problem. However, Bayesian models also allow much more complicated models such as multi-level models to be fit (for example, examine a regression with both team-level and player-level features). Additionally, Bayesian models' posterior distribution captures uncertainty not also present with other estimation methods. For those of you wanting to know more about thinking like a Bayesian or doing Bayesian statistics, here are some books we have found to be helpful:

- *The Theory That Would Not Die: How Bayes' Rule Cracked the Enigma Code, Hunted Down Russian Submarines, and Emerged Triumphant from Two Centuries of Controversy* by Sharon Bertsch McGrayen (Yale, 2012) describes how people have used Bayesian statistics to make decisions through high-profile examples such as the US Navy searching for missing nuclear weapons and submarines.

- *The Foundations of Statistics* by Leonard “Jimmy” Savage (John Wiley and Sons, 1954; Dover Press, 1972 reprinting) provides an overview of how to think like a Bayesian, especially in the context of decision making such as betting or management.
- *Doing Bayesian Data Analysis*, 2nd Edition by John Kruschke (Elsevier, 2014) is also known as the “puppy book” because of its cover. This book provides a gentle introduction to Bayesian statistics.
- *Bayesian Data Analysis*, 3rd Edition, by Andrew Gelman, John Carlin, Hal Stern, David Dunson, Aki Vehtari, and Donald Rubin (CRC Press, 2014). This book, often called “BDA3” by Stan users, provides a rigorous and detailed coverage of Bayesian methods. Richard could not read this book until he had taken two years of advanced undergraduate and intro graduate-level math courses.
- *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*, 2nd Edition by Richard McElreath (CRC Press, 2020). This book is between the “puppy book” and “BDA3” in rigour and is an intermediate-level text for people wanting to learn Bayesian statistics.

Survival Analysis/Time-to-event

How long does a quarterback last in the pocket until he either throws the ball or is sacked? *Time-to-event* or *survival* analysis would help you answer that question. We did not include cover this in the book because we could not find public data for this analysis. However, for people with more detailed time data, this analysis would help you understand how long until events occur. Some books we found useful on this topic include the following:

- *Regression Modeling Strategies: With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis*, 2nd edition, (Springer, 2015) by Frank Harell. Besides being useful for regression, this book also includes survival analysis.
- *Think Stats* by Allen B. Downey (O’Reilly, 2014) includes an accessible chapter on survival analysis using Python.

Bayesian Networks/Structural Equation Modeling

Chapter 8 alluded to the interconnectedness of data. Taking this a step further, sometimes data has no clearcut cause or effect, or cause and effect variables are linked. For example, consider combine draft attributes. A player's weight might be linked to a player's running speed (for example, lighter players run quicker). Running speed and weight might both be associated with a running back's rushing yards.

How to tease apart these confounding variables? Tools such as structural equation modeling and Bayesian networks allow these relations to be estimated. Some books we found to be helpful include:

- *Book of Why* by Judea Pearl and Dana Mackenzie (Basic Books, 2018) walks through how to think in networks. The book also provides a great conceptual introduction to network models.
- *Bayesian Networks with Examples in R*, 2nd Edition by Marco Scutari and Jean-Baptiste Denis (CRC Press, 2021). The book provides a nice introduction to Bayesian networks.
- *Structural Equation Modeling and Natural Systems* by Jim Grace (Cambridge University, 2006) provides a gentle introduction to these models using ecological data.

Machine Learning

Machine learning is not any single tool, but rather a collection of tools and a method of thinking about data. Most of our book has focused on statistical understanding of data. In contrast, machine learning thinks about how to use data to make predictions in an automated fashion. Many great books exist on this topic, but we do not have any strong recommendations. Instead, build a solid foundation in math, stats, and programming, and then you should be well equipped to understand.

Command Line Tools

Command lines allow you interact with computers using code. *Command lines*

have several related names, that, although having specific technical definitions, are often used interchangeably. One is *shell* is another name because this is the outside or “shell” of the operating system that to humans, such as yourself, touch. Another is *terminal* because this is the software that uses the input and output text. Historically, terminal literally referred to the hardware you used. As a more modern definition, this can refer to the software as well, for example Richard’s Linux computer calls his “command line” application the “terminal.” Lastly, *console* refers to the physical terminal. An [Ask Ubuntu question and answers](#) provides detailed discussion on this topic, as well as some pictorial examples (notably, [this answer](#)).

Although old (for example, Unix development started in the late 1960s), people still use these tools because of their power. For example, deleting 1000s of files would likely take many clicks with a mouse but only one line of code in the command line.

When starting out, command lines can be confusing, just like starting with Python or R. Likewise, using the command line is a fundamental skill, similar to running or coordination drills in football. The command line is also used with most of the advanced skills we list, but will also enhance your understanding of languages such as R or Python. For example, understanding the command line will enhance your understanding of programming languages by making you think about file structures and how computer operating systems work. But which command line to use?

There are two options we suggest you consider. First, the Bourne Again Shell (shortened to bash, named after the Bourne shell that it supersedes, that was named after the shell’s creator Stephen Bourne) traditionally has been the default shell on Linux and macOS. This shell is also now available on Windows and is often the default for cloud computers (such as Amazon Web Services, Microsoft Azule, and Google Cloud Platform) and high performance supercomputers. Most likely, you will use start the the bash shell.

A second option is Windows PowerShell. Historically, this was only for Windows, but is now available for other operating systems as well. Windows PowerShell would be the best choice to learn if you also do a lot of information technology work in a corporate setting. The tools in PowerShell would be able to help you automate parts of your job like security updates and software installs.

If you have macOS or Linux, you already have a terminal with a bash or bash-clone terminal (macOS switched to using the zsh shell language due to copyright issues, but zsh and bash are interchangeable for many situations, including our basic examples). Simply open the terminal app on your computer and follow along. If you use Windows, we suggest downloading [Git for Windows](#) that comes with a lightweight bash shell. For Windows users who discover the utility of bash, you may eventually want to move to using Windows Subsystem for Linux, WSL. This program gives you a powerful, complete version of Linux on your Windows computer.

Bash Example

A terminal interface forces you to think about file structure on your computer. When you open your terminal, type `pwd` to print (on the screen) the current working directory. For example on Richard's Linux computer running Pop!_OS (a flavor of Linux), this looks the following:

```
(base) raerickson@pop-os:~$ pwd  
/home/raerickson
```

where `/home/raerickson` is the current working directory. To see the files in the working directory, type the list command, `ls` (we also think about `ls` as being short for *list stuff* as a way to remember the command):

```
raerickson@pop-os:~$ ls  
Desktop           Games          Public  
Documents         R             Untitled.ipynb  
Downloads        miniconda3    Videos  
Firefox_wallpaper.png Music         Templates  
Pictures          test.py
```

You can see all directories and files in Richard's user directory. File paths are also important. The three basic paths to know are your current directory, your computer's home directory, and up a level:

- `./` is the current directory.
- `~/` is your computer's default home directory.

- `..`/ is the previous directory.

For example, consider if your current directory is `/home/raerickson`. With this example:

- `..`/ would be the *home* directory.
- `.`/ would be the *raerickson* directory.
- `/` would be the lowest level in your computer.
- `~/` would be the default home directory, which is `/home/raerickson` on Richard's computer.

NOTE

For practical purposes, *directory* and *folder* are the same term, and, with the examples in this book, you can use either term.

You can use change directory, `cd`, to change your current working directory. For example, to get to the home directory, you could type:

```
cd ..
```

Or you could type:

```
cd /home/
```

The first option uses a *relative* path. The second option uses an *absolute* path. In general, relative paths are better than absolute, especially for languages like Python and R when other people might be re-using code across different machines.

You can also use the command line to move file and directories. For example, to copy `test.py`, you need to make sure you are in the same directory as the file. To do this, use `cd` to navigate to the directory with `test.py`. Type `ls` to make sure you can see the file. Then use `cp` (the copy function) to copy the file to *Documents*.

```
cp test.py ./Documents
```

You can also use `cp` with different file paths. For example, let's say you're in *Documents* and want to move *test.py* to *python_code*. You could use the file paths with `cp`:

```
cp ../test.py ./python_code
```

In this example, you are currently in */home/raerickson/Documents*. You can take the file from */home/raerickson/test.py/* using *../test.py* to the directory */home/raerickson/Documents/python_code* using *./python_code*.

You can also copy directories. To do this, use the recursive option (or, in Linux parlance, flag) `-r` with `copy`. For example, to copy *python_code*, you would use `cp ./python_code new_location`. A move function also exists, that does not leave the original object behind. The move command is `mv`.

WARNING

Command line file deletions do not go to a recycling or trash directory on your computer. Deletions are permanent.

Lastly, you can remove directories and files using the terminal. We recommend you be very careful. To delete, or remove, files, use `rm file_name`, where *file_name* is the file to delete. To delete a directory use `rm -r directory` where *directory* is the directory you want to remove. To help you get started, [Table 9-2](#) contains common bash commands we use on a regular basis.

Table 9-2. Table 2.1: Common bash commands and their names and descriptions.

Command	Name and description
<code>pwd</code>	Print working directory to show you where you are at.
<code>cd</code>	Change directory to change where you are at on in your

computer.

`cp` Copy a file.

`cp -r` Copy a directory.

`mv` Move a file.

`mv -r` Move a directory.

`rm` Remove a file.

`rm -r` Remove a directory.

Suggested Reading for bash

The bash shell is not only a way to interact with your computer, but also comes its own programming language. We generally only touch the surface of its tools in our daily work, but some people extensively program in the language.

- [Software Carpentry](#) offers free tutorials on the [Unix Shell](#). More generally, we recommend them as a general resource for many of the topics covered in this book.
- *Learning the bash Shell*, 3rd Edition by Bill Newham and Cameron Rosenblatt (O'Reilly, 2005) provide introductions and coverage of the advanced tools and features of the language.
- *Data Science at the Command Line: Obtain, Scrub, Explore, and Model Data with Unix Power Tools*, 2nd Edition by Jeroen Janssens (O'Reilly 2021) shows how to use helpful command-line tools.
- We suggest you browse several books and find the one that meets your needs. Richard learned from *Unix Shells By Example* 4th ed. (Pearson, 2004) by Ellie Quigley.
- Online vendors offer courses on bash. We do not have any

recommendations.

Version Control

When working on code on a regular basis, we face problems such as *how do we keep track of changes?* or *how do we share code?* The solution to this problem is version control software. Historically, several programs existed. These programs emerged from the need to collaborate and see what others did as well as keeping track of your own changes to code. Currently, Git is the major version control program (around the time of publication, it has a high market share, ranging from 70% to 90%).

Git emerged because Linus Torvalds faced that problem with the operating system he created, Linux. He needed a lightweight, efficient program to track changes from an army of volunteer programmers around the world. Existing programs used too much memory because they kept multiple versions of each file. Instead, he created a program that only tracked changes across files. He called this program Git.

NOTE

Fun fact. Linus Torvalds has, half-jokingly, claimed to name his two software programs after himself. Linux is a recursive acronym, Linux is not Unix (Linux), but is also close to his own first name. Git is British English slang for an arrogant person or jerk. Torvalds, by his own admission, can be difficult to work with. As an example, searching for images of Mr. Torvalds will show him responding to a reporter's question with an obscene gesture.

Git

Git, at its heart, is an open source program that allows anybody to track changes to code. People can use Git on their own computer to track their own changes. We will start with some basic concepts of Git here. First, you need to obtain Git.

- For Windows users, we like [Git Bash for Windows](#).
- For macOS Users, we encourage you to make sure you have Terminal installed. If you install XCode, Git will be included, but this will be an

older version of Git. Instead, we encourage you to upgrade Git from the Git project [homepage](#).

- For Linux users, we encourage you to upgrade the Git that comes with your OS to be safe.
- For people wanting a GUI on Windows or macOS systems, we suggest you check out [GitHub's Desktop](#). The [Git project page](#) lists many other clients, including GUIs for Linux as well as Windows and macOS.

TIP

Command line Git is more powerful than any graphical user interface (GUI), but it's also more difficult. We show the concepts using the command line, but encourage you to use a GUI. Two good options include GitHub's GUI and the default Git GUI that comes with Git.

After obtaining Git, you need to tell Git where to keep track of your code:

1. Open a terminal.
2. Change your working directory to your project directory using `cd path/to_my_code/`.
3. In one line, type `git init` then press Enter/Return. `git` is telling the terminal to use the Git program, and `init` is telling the Git program to use the `init` command.
4. Tell Git what code to track. This may be done for individual files using `git add filename` or for all files with `git add .` (The period is a shortcut for all files and directories in the current directory).
5. Commit your changes to the code with `git commit -m "initial commit"`. With this command, `git` is telling the terminal which program to use. `commit` is telling git to commit. The *flag* `-m` is telling commit to accept the message in quotes, "`my changes`". With future edits, you will want to use descriptive terms here.

WARNING

Be careful with which files you track. Seldomly will you want to track data files (such as .csv files) or output files such as images or tables. Be extra carefully if posting code to public repositories such as GitHub. `.gitignore` files will let you block tracking for all file types using commands such as `*.csv` to block tracking CSV files.

Now, you may edit your code. Let's say you edit the file `my_code.R` and then change it. Type `git status` to see that this file has been changed. You may add the changes to the file by typing `git add my_code.R`. Then you need to commit the changes with `git commit -m "example changes"`.

TIP

The learning curve for Git pays itself off if you accidentally delete one or more important files. Rather than losing days, weeks, months, or longer of work, you only loose the amount of time it takes you to search for undoing a delete with Git. *Trust us, we know from experience and the school of hard knocks.*

GitHub and GitLab

After you become comfortable with Git (at least comfortable enough to start sharing your code), you will want to back up and share code. When Richard was in grad school, around 2007, he had to use a terminal to remotely log on to his advisor's computer and use Git to obtain code for his PhD project. He had to do this because easy-to-use commercial solutions (like GitHub) did not yet exist for sharing code.. Luckily commercial services now host Git repositories.

The largest provider is [GitHub](#). This company and service are now owned by Microsoft. Their business model is to allow hosting for free, but charge for business users and extra features for users. The second largest provider is [GitLab](#). It has a similar business model, but is also more developer focused. GitLab also includes an option for free self-hosting using its open source software. For example, O'Reilly Media and one of our employers both self-host their own GitLab repositories.

Regardless of which commercial platform you use, all use the same underlying Git technology and command-line tools. Even through the providers offer

different websites, GUI tools, and bells and whistles, the underlying Git program is the same. Our default go to is GitHub, but we know that some people prefer to avoid Microsoft and use GitLab. Another choice is BitBucket, but we are less familiar with this platform.

The idea for a remote repository is that your code is backed up there and other people can also access it. If you want, you can share your code with others. For open source software, people can report bugs and also contribute new features and bug fixes for your software. We also like the GitHub's and GitLab's online GUIs because they allow people to see who has updated and changed code. Another feature we like about these web pages is that they will render Jupyter Notebooks and Markdown files as static webpages.

GitHub Webpages and Resumes

A fun way to learn about Git and GitHub is to build a resume. A search for GitHub resumes should help you find online tutorials (we do not include links because these pages are constantly changing). A Git-based resume allows you to show your skills and create a marketable product demonstrating your skills. You can also use this to show off your football products, whether for fun or as part of job hunting. For example, we will have our interns create these pages as a way to document what they have learned while also learning Git better. An example resume is a former intern, [John Oliver's page](#).

Suggested Reading for Git

- Git tutorials on project [homepage](#), especially the videos that provide an overview of the Git technology for people with no background.
- Software Carpentry offers [git tutorials](#).
- Training materials on GitHub provide resources. We do not provide a direct link because these change through time.
- Git books from O'Reilly (2012) such as *Version Control with Git*, 3rd Edition, by Jon Loeliger and Matthew J. McCullough can provide resources all in one location.

Style Guides and Linting

When we write, we use different styles. A text to our partner that we're at the store might be *At store, see u.* followed by *a k. plz buy milk.* A report to our boss would be very different and a report to an external client even more formal. Coding can also have different styles. Style guides exist to create a consistent code. However, programmers are a creative and pragmatic bunch and have created tools to help themselves follow styles. Broadly, these tools are called linting.

NOTE

The term *linting* comes from removing lint from clothes, like lint rolling a sweater to remove specks of debris.

Different standards exist for different languages. For Python, PEP8 is probably the most common style guide, although other style guides exist. For R, the Tidyverse/Google style guides are probably the most common style.

NOTE

Open source project often split and then re-join, ultimately becoming woven together. R style guides are no exception. Google first created an R style Guide. Then the Tidyverse Style Guide based itself upon Google's R Style Guide. But then, Google adapted the Tidyverse Style Guide for R with their own modifications. This interwoven history is described on the Tidyverse Style Guide [page](#) and Google's R Style Guide [page](#).

To learn more about the styles, please visit the [PEP8 Style Guide](#), [Tidyverse style homepages](#), or [Google's style guides for many languages](#).

NOTE

Google's style guides are hosted on GitHub (<https://google.github.io/styleguide/>), using a Markdown language that is undoubtedly tracked using Git.

For linting, [Table 9-1](#) lists some example linting programs. We also encourage you to look at the documentation for your code editor. These often will include add-ons (or plug-ins) that allow you lint your code as you write.

Packages

We often end up writing custom functions while programming in Python or R. We need ways to easily reuse these functions and share them with others. We do this by placing the functions in packages. For example, Richard has created Bayesian models used for fisheries analysis that use the Stan language as called through R. He has released these models as an R package, [fishStan](#). The outputs from these models are then used in a fisheries model, which has been released as a Python [package](#).

With a package, we keep all of our functions in the same place. Not only does this allow for reuse, but it also allows us to fix one bug and not hunt down multiple versions of the same file. We can also include tests to make sure our functions work as expected, even after updating or changing functions. Thus, packages allow us to create reusable and easy-to-maintain code.

We can use packages to share packages for others. Probably the most common method to release packages is now on GitHub repos. There is a low barrier to entry, hence anyone can release packages. Python also has multiple package managers where people can submit packages to including `pip` and `conda forge`. Likewise, R currently has one major package manager (and historically, had more): The Comprehensive R Archive Network (CRAN). These repositories have different levels of quality standards prior to submission and thus some gatekeeping occurs compared to a direct release on sites like GitHub.

Suggested Reading for Packages

- To learn about R packages, we used Hadley Wickham's [online book](#) that is also available as a dead tree version from O'Reilly.
- To learn about Python packages, we used the Official Python tutorial, [*Packaging Python Projects*](#).

Computer Environments

Imagine you run an R or Python package, but then the package does not run next session. Eventually, hours later, you figure out that a package was updated by its owner and now you need to update your code (yes, similar situations have occurred to us). One method to prevent this problem is to keep track of your computer's environment. Likewise, computer users might have problems working with others. For example, Eric wrote this book on a Windows computer, whereas Richard used a Linux computer.

A computer's environment is the computer's collection of software and hardware. For example, you might be using a 2022 Dell XPS 13" laptop for your hardware. Your software environment might include your operating system (OS), such as Windows 11 release 22H2 (10.0.22621.1105), as well as the versions of R, Python, and their packages, such as R 4.1.3 with `ggplot2` version 3.4.0. In general, most people are concerned about the programs for the computing environment. When an environment does not match across users (for example, Richard and Eric) or across time (for example, Eric's compute in 2023 compared to 2021), programs will sometimes not run, such as the problem shows in [Figure 9-4](#).

TIP

We cannot do justice for virtual environments like `conda` in this book. However, many programmers and data scientists would argue their use helps to differentiate experienced professionals from amateurs.

Tools like `conda` let you lock down your computer's environment and share the specific programs used. Tools like `Docker` go a step farther and not only control the environment, but also the operating systems. Both of these programs work best when the user understands the terminal.

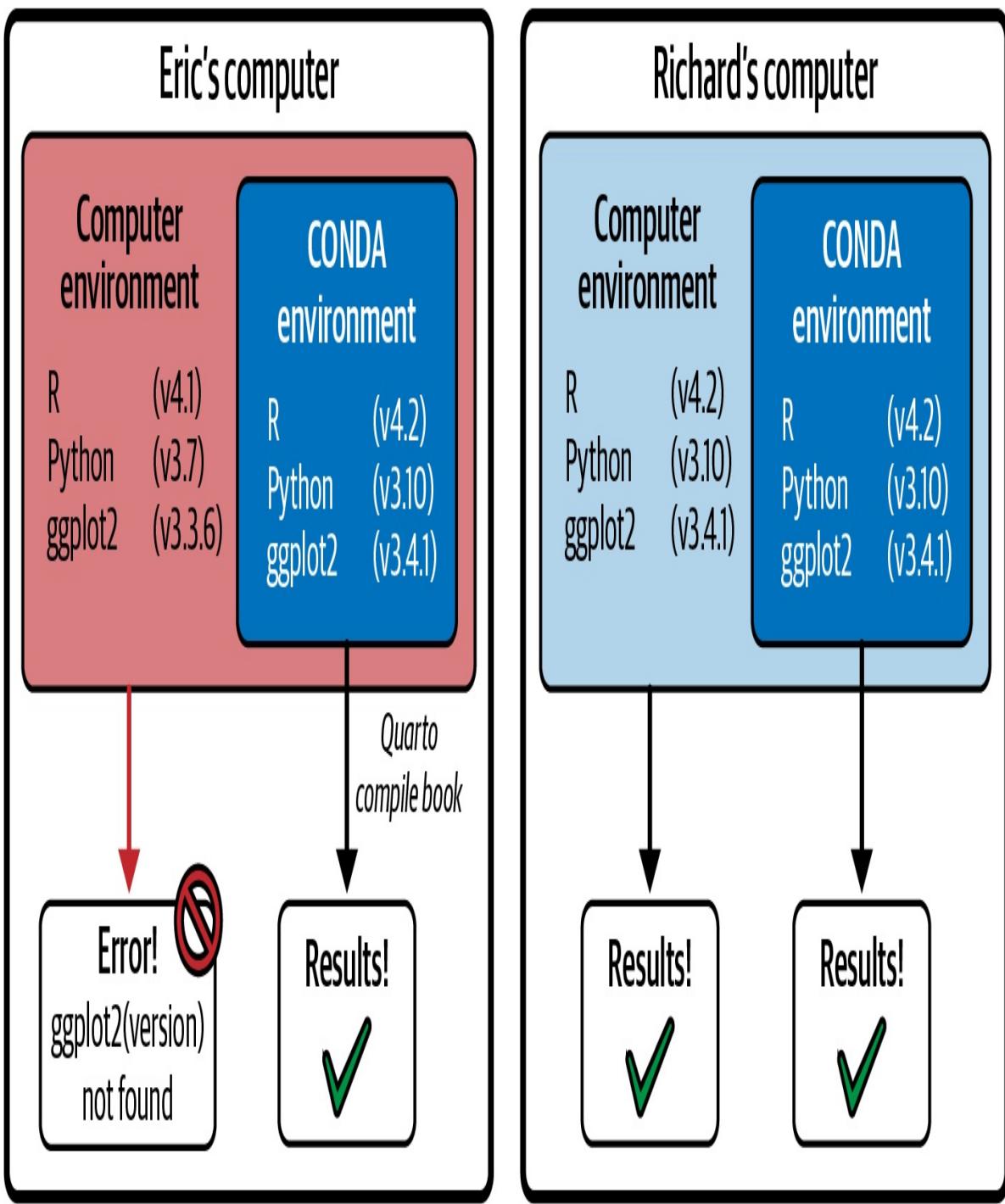


Figure 9-4. Example of computer environments and how version may vary across users and machines.

Interactives and Report Tools to Share Data

Most people do not code. However, many people want access to data, and, hopefully, you want to share code. One tool to share data and models are

interactive applications or *interactives*. These allow people to interact with your code and results. For small projects, such as the ones some users may want to share after completing this book, programs like **Posit's Shiny** or web hosted **Jupyter Notebooks with widgets** may meet your needs. People working in the data science industry, like Eric, will also use these tools to prototype models before handing the proof of concept tool over to a team of computer scientists to create a production-grade product.

Interactive work great for dynamic tools to see data. Other times, you may want or need to write reports. Markdown-based tools let you merge code, data, figures, and text all into one. For example, Eric writes reports to clients in **RMarkdown**, Richard writes software documentation in **Jupyter Notebooks**, Richard writes scientific papers in **LaTeX**, and this book was written in **Quarto**. If starting out, we suggest Quarto because the language expands upon RMarkdown to also work with Python and other languages (RMarkdown itself was created to be an easier-to-use version of LaTeX). Jupyter Notebooks can also be helpful for reports and longer documents (for example, books have been written using Jupyter Notebooks), but tend to work better for dynamic applications like interactives.

Artificial Intelligence Tools

Currently, tools exist that help people code using artificial intelligence (AI) or similar type tools. For example, many code editors have autocompletion tools. These tools, at their core, are functionally AI. During the writing of this book, new artificial intelligence tools have emerged that hold great potential to assist people coding. For example, ChatGPT can be used to generate code based upon user input prompts. Likewise, programs such as GitHub Copilot exist that help people code based upon input prompts and Google launched their own competing program, Codey.

However, AI tools are still new and challenges exist with their use. For example, the tools will produce factual **errors** and well documented **biases**. Besides well documented factual errors and biases, the programs consume user data. Although this helps create a better program through feedback, people can accidentally release data they did not intent to release. For example, Samsung staff

accidentally release semiconductor software and proprietary data to ChatGPT. Likewise, [Copilot's Business Privacy Statement](#) notes that “It collects data to provide the service, some of which is then saved for further analysis and product improvements.”

WARNING

Do not upload data and code to AI services unless you understand how the services may use and store your data and code.

We predict that AI-based coding tools will greatly enhance coding, but also required skilled operators. For example, spellcheckers and grammar checkers did not remove the need for editors. They simply reduced one part of editors’s jobs.

Conclusion

American football is the most popular sport in the United States, and one of the most popular sports in the world. Millions of fans travel countless miles to travel to see their favorite teams every year, and more than a billion dollars in television deals are signed every time it’s time for a contract to be renewed. Football is a great vehicle for all sorts of involvement, whether that be entertainment, leisure, pride, or investment. Now, hopefully, it is also a vehicle for math.

Throughout this book we’ve laid out the various ways in which a mathematically-inclined person could better understand the game through statistical and computational tools learned in many undergraduate programs throughout the world. These very same approaches have helped us as an industry push the conversation forward into new terrain, where analytically-driven approaches are creating new problems for us to solve, which will no doubt create additional problems for you, the reader of this book, to solve in the future.

The last decade of football analytics has seen us move the conversation towards the “signal and the noise” framework popularized by Nate Silver. For example, Eric and his former co-worker George Chahrouri asked the question if “we want to predict the future, should we put more stock in how a quarterback plays under

pressure or from a clean pocket?” in a PFF [article](#).

We’ve also seen a dramatic shift in the value of players by position, which has largely been in line with the work of analytical firms like **PFF who help people construct rosters by valuing positions**. Likewise, websites like rbsdm.com have allowed fans, analysts, and scribes to contextualize the game they love and/or cover using data.

On a similar note, the legalization of sports betting across much of the United States has increased the need to be able to tease out the meaningful from the misleading. Even the NFL draft, at one point an afterthought in the football calendar, has become a high-stakes poker game and, as such, has attracted the best minds in the game working on making player selection, and asset allocation, as efficient as possible.

The future is bright in the space as well. With the recent proliferation of player tracking data, the insights from this book should serve as a jumping off point in a field with an every-growing set of problems that should make the game more enjoyable. After all, almost every analytical advancement in the game (more passing, more fourth-down attempts) has made the game more entertaining. We predict that trend will continue.

Furthermore, sports analytics in general, and football analytics specifically, has opened doors for many more people to participate in sports and be actively engaged than in previous generations. For example, Eric’s internship program, as of May of 2023, has sent four people into NFL front offices, with hopefully many more to come. By expanding who can participate, as well as add value to the game, football now has the opportunity to become a lot more compelling for future generations.

Hopefully our book has increased your interest in football and football analytics. For someone who just wants to dabble in this up-and-coming field, perhaps it contains everything you need. For those that want to gain an edge in fantasy football or in your office pool, you can update our examples to get the current year’s data. For people seeking to dive deeper, the references in each chapter should provide a jumping-off point for future inquiry.

Lastly, here are some websites we look at for more Football information:

- Eric’s current employer: www.sumersports.com

- Eric's former employer: www.pff.com
- A website focusing on advanced NFL stats: www.footballoutsiders.com
- Ben Baldwin's page that links to many other great resources: rbsdm.com

Happy coding as you dive deeper into football data!

Appendix A. Python and R Basics

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Appendix A in the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

This appendix provides details on the basics of Python and R. For readers unfamiliar with either language, we recommend you spend an hour or two working through basic tutorials such as those listed on the project's homepages listed in “[Local Installation](#)”.

TIP

In both Python and R, you may access built-in help files by typing `help("function")`. For example, typing `help("")`` loads a help file for the addition operator. In R, you can often also use the ``?`` shortcut (for example ``?`"`), but need to use quotes for some operators such as `+` or `-`.

Obtaining Python and R

Two options exist for running Python or R. You may install the program locally on a computer or you may use a cloud-hosted version. From the arrival of the *personal computer* (or PC for short) in the late-1980s up until the 2010s, almost all computing was local. People downloaded and installed software to their local computer unless their employer hosted (and paid for) cloud access or local

servers. Most people still use this approach and use local computing rather than the cloud. More recently, consumer-scale cloud tools have become readily available. Trade-offs exist for using both approaches:

- Benefits of using a personal computer:
 - Upfront costs for hardware
 - You control all hardware and software (includes storage devices)
 - You can install anything you want
 - Will run without internet connections such as during internet outages, in remote locations, or while flying
- Benefits of using consumer-scale cloud-based computing:
 - Only pay for computing resources you need
 - Many free consumer services exist
 - You do not need expensive hardware, for example could use Chromebook or even tablets such as an iPad
 - Cloud-based computing is cheaper to upscale when you need more resources
 - Depending upon platform and support levels, you're not responsible for installing programs

Local Installation

Directions for installing Python and R appears on the project's respective pages:

- [Python](#)
- [R Project](#)

We do not include directions here because the programs may change and depend upon your specific operating systems.

Cloud-based Options

Many different vendors offer cloud-based Python and R. Some current vendors in 2023 include:

- Google's [Colab](#)
- Posit's [Cloud](#)
- DataCamp's [WorkSpaces](#)
- O'Reilly's [Interactive Learning](#)

Scripts

Script files (or *scripts* for short) allow you to save your work for future applications of your code. Although our examples show typing directly in the terminal, typing in the terminal is not effective or easy. These files typically have an ending specific for the language. Python files end in *.py* and R files end in *.R*.

During the course of this book, you will be using Python and R interactively. However, some people also run these languages as batch files. A *batch file* simply tells the computer to run an entire file and then spits out the outputs from the file. An example batch file might calculate summary statistics that get run weekly during the NFL season by a company like PFF and then placed into client reports.

In addition to batch files, some people have large amounts of text needed to describe their code or code outputs. Other formats often work better than script files for these applications. For example, Jupyter Notebooks allow people to create, embed, and use run-able code with easy to read documents. The *Notebooks* in the name “Jupyter Notebooks” suggests a similarity to a scientist’s laboratory or field notebook where text and observations become intermingled.

Likewise, Markdown-based files (such as Quarto or the older RMarkdown files) allow people to embed static code and code outputs into documents. We use both for our work. When we want other people to be readily able to interactively run our code, we use Jupyter Notebooks. When we want to create static reports such as project files for NFL teams or scientific articles, we use RMarkdown-based workflows.

Early drafts of this book were written in Jupyter Notebooks because we were not aware of O'Reilly's support for Quarto-based workflows. Later drafts were written with Quarto because we (as authors) know this language better. Hence, we as authors, adapted our tool choice based upon our publisher's systems, which illustrates the importance of flexibility in your toolbox.

WARNING

Many programs, such as Microsoft Word, use ``smart quotes'' and other text formatting. Because of this, be wary of any code copied over from files that are not plain text files such as PDFs, Word Documents, or webpages.

Packages in Python and R

Both Python and R succeed and thrive as languages because they can be extended via packages. Python has multiple package management systems, which can be confusing for novice and advanced users. For example, novice users often don't know which system to use whereas advanced users run into conflicting versions of packages across systems.

The simplest method to install Python packages is probably from the terminal outside of Python using the `pip` command. For example, you may install `seaborn` using `pip install seaborn`. R currently only has one major repository of interest to readers of this book, the Comprehensive R Archive Network (CRAN; a second repository, bioconductor mainly support bioinformatics projects). An R package may be installed inside R using the `install.packages()` function. For example, you may install `ggplot2` with `install.packages("ggplot2")`.

You may also, at some point, find yourself needing to install packages from sites like GitHub because the package is either not CRAN, or, you need the development version to fix a bug that is blocking you from using your code. However, these install methods are beyond the scope of this book, but may be easily found using a search engine. Furthermore, different environment management software exists that allow you to lock down package versions. We discussed these in [Chapter 9](#).

nlfastR and nfl_py_data Tips

“[Packages in Python and R](#)” describes general packages. This section provides some specific observations we have about the `nlfastR` and `nfl_py_data` packages. First, we the authors love these packages. They area great free resources of data.

TIP

Update these packages on a regular basis during the seasons. That will give you access the newest data.

You may have noticed you have to do a fair amount of wrangling with data. That’s not a downfall of these packages. Instead, this shows the flexibility and depth of these data sources. We also have you cleanup the datasets, rather than provide you with clean datasets, because we want to you to learn how cleanup your data. Also, these datasets are updated nightly during the season, so, by showing you how to use these packages, we give you the tools obtain your own data.

To learn more about both packages, we encourage you to dive into the details of the packages. Once you’ve looked over the basic functions and structure of the packages, you may even want to dive into the packages and look at the code and “peek under-the-hood.” Both `nlfastR` and `nfl_data_py` are on GitHub. This site allows you to both report issues, bugs, and suggest changes— ideally by providing your own code!

Lastly, we encouraged you to give back to the open source community that supplies these tools. Although not everyone can contribute code, helping with other tasks like documentation are more accessible.

Integrated Development Environments

You can use powerful code-editing tools called *integrated development environments* (IDEs). Much like football fans fight over who is the best quarterback of all time, programmers often argue over which IDEs are best. Although powerful (for example, IDEs often include syntax checkers similar to a

spell checker and auto-completion tools), IDEs can have downsides. Some IDEs are complex, which can be great for expert users, but overwhelming for beginners and casual users. For example, the emacs text editor has been jokingly described as an operating system with a good text editor or two built into it. Others, such as the web comic [XKCD poke fun at IDEs such as emacs](#) as well. Likewise, some professional programs feel that the shortcuts built into some IDEs limit or constrain understanding of languages because they do not require the programmer to have as deep of understanding of the language they are working in. However, for most users, especially casual users, the benefits of IDEs far outweigh the downsides. If you already use another IDE for a different language at work or elsewhere, that IDE also likely works with Python and possibly R as well.

When writing this book, we used different editors at different times and for different file types. Editors we used included including RStudio Desktop, Jupyter Lab, Visual Studio Code, and emacs. Many good IDEs exist, including many we do not list. Some options include:

- Microsoft's [Visual Studio Code](#)
- Posit's [RStudio Desktop](#)
- JetBrains's [PyCharm](#)
- Project Jupyter's [Jupyter Lab](#)
- GNU Project's [emacs](#)

Basic Python Data Types

As a crash course in Python, different types of objects exist within the language. The most basic type, as least for this book, include *integers, floating point numbers, logical, strings, lists, and dictionaries*. In general, Python takes care about thinking about these data types for you and will usually change the type of number for you.

Integers (or *ints for short*) are whole numbers like 1, 2, and 3. Sometimes, you need integers to index objects (for example, taking the 2nd element, for a vector x using $x[1]$; recall Python starts counting with 0). For example, you can

create an integer x in Python:

```
## Python
x = 1
x.__class__
<class 'int'>
```

Floating point numbers (or *floats* for short) are decimal numbers that recorded to a finite number of digits such as `float16`. Python will turn ints into floats when needed such as:

```
## Python
y = x/2
y.__class__
<class 'float'>
```

Computers cannot remember all digits and computer numbers are not mathematical numbers. Consider the parlor trick Richard's math co-advisor taught him in grad school:

```
## Python
1 + 1 + 1 == 3
True
```

But, this does not always work:

```
## Python
0.1 + 0.1 + 0.1 == 0.3
False
```

This is due to a rounding error that occurs on the computer chip calculating.

Python also has `True` and `False` logical operators that are called *Boolean objects* or *bool* for short. The previous examples showed how bools results from logical operators.

Letters are stored as *strings* (or *str* for short). Numbers stored as strings do not behave like numbers. For example, look at:

```
## Python
a = "my two"
```

```
c = 2
a * c
'my two'my two'
```

Python has groups of values. Lists may be created using `list()` or `[]`. For example:

```
## Python
my_list = [1, 2, 3]
my_list_2 = list(("a", "b", "c"))
```

Dictionaries may also store values with a “key” and are created with `dict()` or `{}`. We use the `{}` almost exclusively but show the other method for completeness. For example

```
## Python
my_dict = dict([('fred', 2), ('sally', 5), ('joe', 3)])
my_dict_2 = {"a": 0, "b": 1}
```

Additionally, the `numpy` package allows data to be stored in arrays and the `pandas` package allow data to be stored in dataframes. Arrays must be the same data type, but dataframes may have mixed columns (such as one column of numbers and a second column of names).

Basic R Data Types

As a crash course in R, different types of objects exist within the language, like in Python. The most basic type, at least for this book, include *integers*, *numeric floating point numbers*, *logical*, *characters*, and *lists*. In general, R takes care about thinking about these structures for you and will usually change the type of structures for you.

Integers are whole numbers like 1, 2, and 3. Sometimes, you need integers to index objects (for example, taking the 2nd element, for a vector `x` using `x[2]`; recall R starts counting with 1). For example, you can create an integer `x` in R using `L`:

```
## R
x <- 1L
class(x)
[1] "integer"
```

Numerical floating point numbers (or *numeric* for short) are decimal numbers that recorded to a finite number of digits. R will turn integers into numeric when needed such as:

```
## R
y <- x / 2
class(y)
[1] "numeric"
```

Computers cannot remember all digits and computer numbers are not mathematical numbers. Consider the parlor trick Richard's math co-advisor taught him in grad school:

```
## R
1 + 1 + 1 == 3
[1] TRUE
```

But, this does not always work:

```
## R
0.1 + 0.1 + 0.1 == 0.3
[1] FALSE
```

This is due to a rounding error that occurs on the computer chip calculating.

R also has **TRUE** and **FALSE** logical operators that are called *logical* for short. The previous examples showed how logical outputs results from logical operators.

Letters are stored as *characters* in R. These do not work with numeric operators. For example, look at:

```
## R
a <- "my two"
c <- 2
a * c
```

R has groups of values called lists or vectors. Lists may be created using *combine* or *concatenate* function. For example:

```
## R
```

```
my_list <- c(1, 2, 3)
```

Additionally, **base R** contains matrices to store numbers and dataframes to store mixed columns (such as one column of numbers and a second column of names). We use tibbles from the **tidyverse** in this book as an upgraded version of dataframes.

Appendix B. Summary Statistics and Data Wrangling: Passing the Ball

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Appendix B in the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

This appendix contains materials to help you understand some basic statistics. If the topics are new to you, we encourage you to read this material after [Chapter 1](#) or [Chapter 2](#) and before you dive too much into the book.

In [Chapter 2](#), you looked at quarterback performance at different pass depths in an effort to understand which aspect of play was fundamental to performance and which aspect was noisier, possibly leading you astray as you aimed to make predictions about future performance. You were lucky enough to have the data more-or-less in ready-made form for you to perform this analysis. You did have to create your own variable for analysis, but such *data wrangling* was minimal at best.

Sports analytics generally, and football analytics specifically, are still in their early stages of development. As such, datasets may not always be the cleanest, or *tidy*. *Tidy* datasets are usually in a table form that computers can easily read and humans can easily understand. Furthermore, data analysis in any field (and football analytics is no different), often requires datasets that were created for

different purposes. This is where data wrangling can come in handy. Because so many people have had to clean up messy data, many terms exist in this field. Some synonyms for *data wrangling* include data cleaning, data manipulating, data mutating, shaping, tidying, and munging. More specifically, these terms describes the process of using a programming language such as Python or R to update datasets to meet your needs.

NOTE

Tidy data has two definitions. Broadly, *tidy* data refers to clean data. More formally, Hadley Wickham defines *tidy data* in an article titled [Tidy Data](#). His definition says that *tidy data* requires: '(1) Each variable forms a column, (2) each observation forms a row, and (3) each type of observational unit forms a table.'

During the course of our careers, we have found that data wrangling takes the most time for our projects. For example, one of our bosses once pinged us on Google chat because he was having trouble fitting a new model. His problem turned out to not be the model, but rather data formatting. Figuring out how to format the data to work with the model took about 30 minutes. However, running the new model only took about 30 seconds in R after we figured out the data structure issue he was having.

TIP

Computer tools are ever changing and data wrangling is no exception. During the course of his career, Richard has had to learn four different tools for data wrangling: base R (around 2007), `data.table` in R (around 2012), the `tidyverse` in R (around 2015), and `pandas` in Python (around 2020). Hence, you will also likely need to update your skill set for the tools taught in this book. However, the fundamentals never change as long as you understand the basics.

Programming languages like Python or R are our most-effective tools to change our data to usable. The languages allow scripting, there by letting us to track of our changes and see what we did and if we introduced any errors into our data. Many people like to use spreadsheet programs such as Microsoft Excel or Google Sheets for data manipulation. Unfortunately, these programs do not keep track of changes easily. Likewise, hand-editing data does not scale, so as the size

of the problem becomes too large, such as when you are working with player *tracking data*, which has one record for every player anywhere from 10 to 25 times per second per play, you will not be able to quickly and efficiently build a workflow that works in a spreadsheet environment. Thus, editing one or two files by hand is easy to do with Excel, but editing one or two thousand files by hand is not easy to do. Conversely, programming languages, such as Python or R, readily scale. For example, if you have to format data after each weeks' games, Python or R could easily be used as part of a data pipeline, but spreadsheet-based data wrangling would be difficult to automate into a data pipeline.

NOTE

Data pipeline refers to the concept that data needs to flow from one location to another, and, undergo changes such as formatting. For example, when Eric worked at PFF a data pipeline might take weekly numbers, format the numbers, run a model, and then generate a report. In computer science, a *pipe* operator refers to passing the outputs from one function directly to another.

That being said, we understand many people like to use tools they are familiar with. If you are switching over to Python or R from using programs like Excel, we encourage you to switch one step at a time. As an analogy, think about a cook licking the batter spoon to taste the dish. When cooking at home for your family, many people do this. But, the chef at a restaurant would hopefully be fired for licking and re-using their spoon. Likewise, recreational data analysis can reasonably use program like Excel to edit data. But, professional data analysis requires the us of code to wrangle data.

TIP

We encourage you to start doing one step at a time in Python or R if you already use a program like Excel. For example, let's say you currently format your football data in Excel, plot the data in Excel, and then fit a linear regression model in Excel. Start by plotting your data in Python or R the next time you work with your data. Once you get the hang of that, start fitting your model in Python or R. Finally, switch to formatting data in Python or R. For readers looking for help with this transition, *Advancing into Analytics: From Excel to Python and R* by George Mount (O'Reilly 2021) provides resources.

Besides data wrangling, you will also learn about some basic statistics in this appendix. *Statistics*, means different things to different people. During our day jobs, we see four uses of the word. Commonly, people use the word to refer to raw or objective data. For example, the (x,y) coordinates of a targeted pass might be referred to as the *stats* for a play. Sometimes a *statistic* can be something that is estimated, like expected points added per play, or completion percentage above expected by a quarterback or offense. More formally, *statistics* can refer to the systematic collection and analysis of data. For example, somebody might run statistical analysis as part of science experiment or as a business analyst using data science. Finally, the corresponding field of study related to the collection and analysis of data is called *statistics*. For example, you might have taken a statistics course in high school or know somebody who works as a professional statistician.

This appendix focuses on the use of *statistic* as something that can be estimated, specifically summary statistics and we show you how to summarize data using statistics. For example, rather than needing to read the play-by-play report for a game, can you get an understanding of what occurred by looking at the summary statistics from the game? In Eric's job, he generally doesn't have the time to watch every game even once, let alone multiple times, nor does he have the chance to pour through each game's play-by-play data manually. As such, he generally builds systems that can deliver to him *key performance indicators* (KPIs) that can help see trends emerge in an efficient way. Summary statistics can also serve as the *features* for models. For example, if someone wants to bet on the number of touchdowns a quarterback will throw for in a certain game, his average number of touchdown passes thrown in a game is likely to be very helpful.

Basic Statistics

Although not as glamorous as plotting, basic summary statistics are often more important because they serve as a foundation for data analysis, and, many plots.

Averages

Perhaps the simplest statistic is the *average*, or *mean*, or for the mathematically

minded, the *expectation* of a set of numbers. Commonly, when we talk about the *average* for a dataset, we are talking about a the central tendency of the data, the value that “balances” the data. We show how to calculate these by hand in the next section.

We intentionally do not include code for this section, as one of Eric’s professors at the University of Nebraska - Lincoln, Dave Logan, would say, “the details of most calculations should be done once in everyone’s life, but not twice.” We will show you how use Python and R to calculate these later in this appendix.

To work through this exercise, let’s explore passing plays again, like in [Chapter 1](#). This time you will actually look at a quarterback’s `air_yards` and study its properties in an attempt to understand his and his team’s approach to the passing game. We use data from a 2020 game between Richard’s favorite team, the Green Bay Packers, and one of their division rivals, the Detroit Lions, and only then look at plays that have an `air_yards` reading by Detroit over the middle of the field. [“Filtering and Selecting Columns”](#) shows to obtain and filter this data. Looking a small set of data will allow you to easily do hand calculations.

First, let’s calculate a *mean* by hand. The air yards are:

5, -1, 5, 8, 5, 6, 1, 0, 16, and 17. To calculate the mean, first *sum* (or add up all of) the numbers:

$$5 + -1 + 5 + 8 + 5 + 6 + 1 + 0 + 16 + 17 = 68$$

Next, divide by the total number plays with air yards:

$$\frac{68}{11} = 6.2$$

This allows you to estimate the mean air yards to the middle of the field to be 6.2 yards for Detroit during their first game against Green Bay in 2020. Also, we rounded the output to be 6.2. We need to round because the decimal of the resulting mean does not end and we only really need to know the first digit after the decimal, since the data is in integers. More formally, this is known as the number of significant digits or figures.

TIP

Significant digits are important when reporting results. Although formal rules exist, a rule of thumb that works most of the time is to simply report the number of digits that are useful.

Another way to estimate the “center” of a data set is to examine the *median*. The median is simply the middle number, or the value of the average individual (rather than the average value). To calculate the median, write the numbers in order from smallest to largest and then find the middle number, or the average of the two middle numbers if there are an even number:

The last common method to estimate an average number is to examine the *mode*. The mode is the most common value in the data set. To calculate the mode, we need to create a table with counts and air yards such as this:

	air_yards	count									
1	-1	1	1	1	0	1	2				0
2	1	3	5	3	4	16	6				1
5	8	1	6	1	16						
7	17	1									

?(caption)

With this example, 5 is the mode because there were three observations with a reading of 5 air yards. Data can be multi-modal, that is to say, have multiple modes. For example, if two outcomes have the same number of occurrences, then a bimodal outcome occurred.

Each central tendency measure has its pluses and minuses. The mean of a set of numbers depends heavily upon outliers (see “[Boxplots](#)” for a formal definition of *outliers*). If 2022 NFL MVP Patrick Mahomes, who at the time of this writing makes about \$45 million annually, walks over to a set of nine practice squad players (each making \$207,000 annually as of 2022), the average salary of the group is about \$4.5 million per player, which doesn’t accurately represent anything about that group. The median and mode of this data set doesn’t change at all (\$207,000) with the inclusion of Mahomes, as the middle player scoots over half of a spot, and the group with the highest reading is the practice squad salary. That being said, much of the theoretical mathematics that has been built works a lot better with the mean, and discarding data points because they are

outside of the confines of the middle of the data set is not great practice, either. Thus, as with everything, the answer to which one you should use is “it depends.”

Finally, there are other kinds of means, but they will not appear in this book. For example, in sports betting (or financial investing in general) one will often care about the geometric mean of a data set rather than the arithmetic mean - which we’ve computed above. The geometric mean is simply computed by multiplying all of the numbers in the data set together and then taking the root corresponding to how many elements of the data set there are. The reason this is preferable in betting or financial markets is clear - numbers grow (or decline) exponentially in this case rather than additively.

Let’s examine these three measures of central tendency for air yards for all pass locations for both teams from the 2020 Green Bay and Detroit’s first game in **Figure B-1**. This subset of the data is more interesting to examine, but would have been harder to examine *by hand*. First, notice the blue line, which is the mean, is to the right of the median, which means the data is skewed or has outliers to the right. Second, the median is the same as the mode:

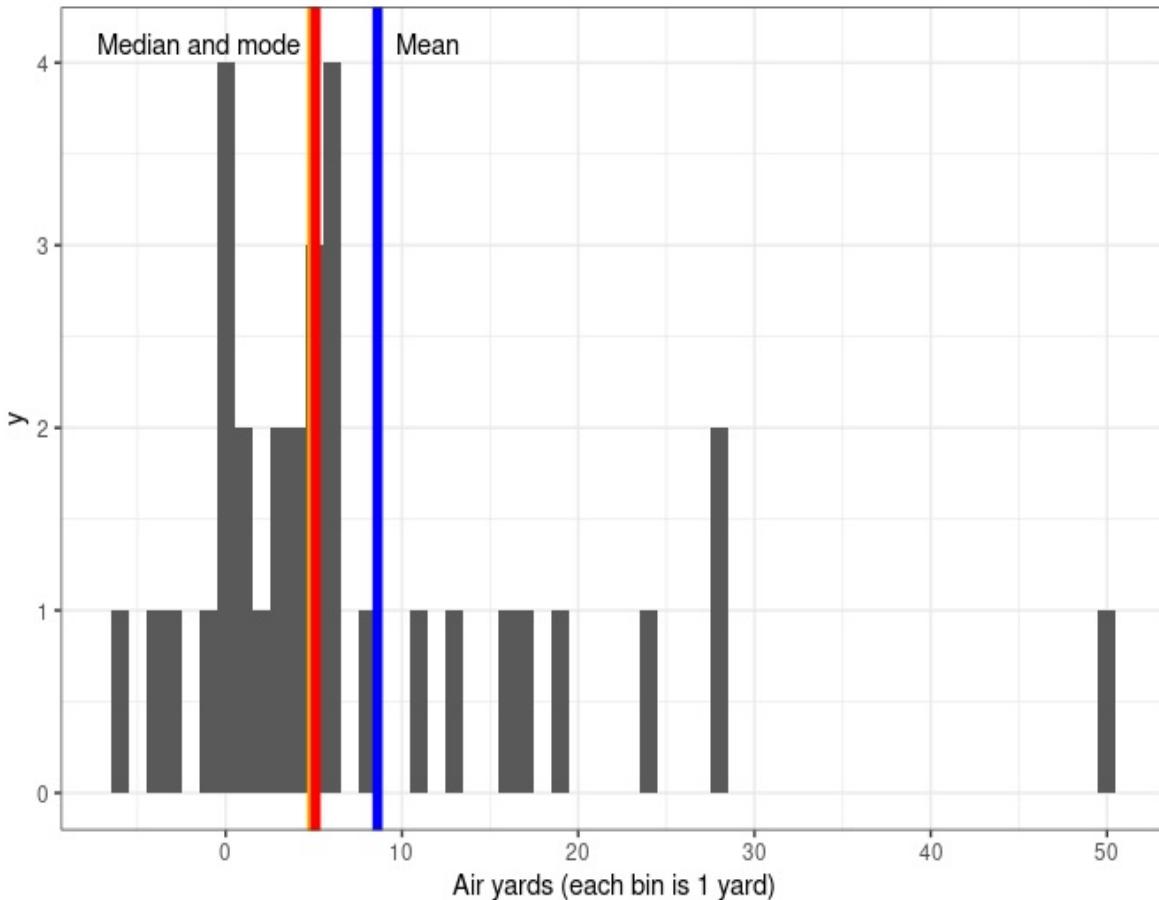


Figure B-1. Comparison of different types of “averages”, also known as the “central tendency” of the data. The mean is the right-most vertical (blue online) line. The mean is a term most people, including us, usually refer to as the average. The leftmost and overlapping vertical lines are the median and the mode (red and orange online). Both of these lines are off-set because they are the same value for this example.

So, what does this tell us about Detroit’s passing game in this game? Firstly, the difference between mean and median shows that many pass plays are short, with the exception of a few long passes. This histogram shows us the “shape” of the data - and the story it tells us about the game, nicely, and probably better than simple summary statistics would. This doesn’t necessarily scale up with the size of the data, but shows why it’s always good to plot your data when you start and the reason we covered plotting as part of EDA in [Chapter 2](#).

Variability and Distribution

The previous section dealt with central tendency. Many situations, however, require you to know about the *variability* in the data. The easiest way to examine the variability in the data is the *range*, which is just the distance between the

smallest (minimum, or min) and biggest number (maximum, or max) in the data. For data shown in the table in the prior section, the min is -1 and the max is 17, so the range is $17 - -1 = 17 + 1 = 18$.

Another methods to examine the range and distribution of a dataset is examine the *quantiles*. These focus on a specific parts of the distribution, and their use can avoid the strength of severe outliers. The n th quantile is the data point where n percent of the data lies underneath that data point. You've already seen one of these before, as the 50th quantile is the same thing as the median. “**Boxplots**” covered quantiles, specifically within the context of boxplots. The 25th, 50th, and 75th quantiles are commonly referred to as *quartiles*, and the difference between the first quartile and the third the *interquartile range* or *IQR* for short.

Recall that boxplots show us where the middle 50% of the data occurs. Sometimes, other types of quantiles may be used as well. The benefit of quantiles are that they are estimated end points other than the central tenancy. For example, the mean or median allows you to examine how well average players do, but a quantile allows us to examine how well the best players do (for example, what does a player need to be better than 95% of other players?).

The most-common method for examining the variability in a dataset is to look at the the *variance* and its square root, the *standard deviation*. Broadly, the variance is the average squared deviation between the data points and the mean. The square is used so that the distance between each data point and the mean is counted as positive - so variability beneath and above the mean don't “cancel out”. Using the Detroit Lions air yards example, you can do this calculation by hand with the numbers supplied in the following table. We include a column “mean” in case you are doing this calculations in a spreadsheet such as Excel and to also help you see where the numbers come from.

	air_yards	mean	difference	difference squared
1	5	8.03125	-3.03125	9.18848
2	13	8.03125	4.96875	24.6885
6	3	8.03125	-5.03125	25.3135
13	6	8.03125	-2.03125	4.12598
15	6	8.03125	-2.03125	4.12598
18	-1	8.03125	-9.03125	81.5635
19	5	8.03125	-3.03125	9.18848
22	3	8.03125	-5.03125	25.3135

\n	24		4		8.03125		-4.03125		16.251
\n	30		28		8.03125		19.9688		398.751
\n	31		28		8.03125		19.9688		398.751
\n	32		11		8.03125		2.96875		8.81348
\n	33		-6		8.03125		-14.0312		196.876
\n	34		-4		8.03125		-12.0312		144.751
\n	35		-3		8.03125		-11.0312		121.688
\n	42		0		8.03125		-8.03125		64.501
\n	43		8		8.03125		-0.03125		0.000976562
\n	48		1		8.03125		-7.03125		49.4385
\n	49		6		8.03125		-2.03125		4.12598
\n	51		2		8.03125		-6.03125		36.376
\n	53		5		8.03125		-3.03125		9.18848
\n	54		6		8.03125		-2.03125		4.12598
\n	55		19		8.03125		10.9688		120.313
\n	58		0		8.03125		-8.03125		64.501
\n	60		0		8.03125		-8.03125		64.501
\n	61		1		8.03125		-7.03125		49.4385
\n	63		4		8.03125		-4.03125		16.251
\n	66		24		8.03125		15.9688		255.001
\n	69		0		8.03125		-8.03125		64.501
\n	73		16		8.03125		7.96875		63.501
\n	74		17		8.03125		8.96875		80.4385
\n	77		50		8.03125		41.9688		1761.38
'									

?(**caption**)

After you create this table, you then take the sum of the difference squared column, which is 4177. You divide that result by 76 because this is the number of observations minus 1. The reason to subtract 1 from the denominator has to do with the number of *degrees of freedom* at dataset has - the amount of data necessary to have a unique answer to a mathematical question.

Calculating $\frac{4177}{(77-1)}$, gives the variance, 54.96. The units for variance with this example would be yards \times yards or yards². This does not relate that well to the central tendency of the data, so we take the square root to get the standard deviation: 7.41 yards, which is now in the units of the original data. Every statistical software can calculate this value for you, and comparing variances and standard deviations across different datasets helps one compare the variability of different sources of data easy and in a way that scales up. Often, one will divide the standard deviation by the mean to get the *coefficient of variation* which is a unit-less measure of variability that takes into account the size of data points.

This might be important when comparing, say, passing yards per play to kickoff returns per play. One is on the order of 10 and the other is on the order of 20, and the variability understandably scales with that.

Uncertainty Around Estimates

When people give you predictions or summaries, a reasonable question is how much certainty exists around the prediction or summary? You can show uncertainty around the mean using the *standard error of the mean*, often abbreviated as SEM or simply SE for *standard error*. More informatively, you can estimate *confidence intervals*, abbreviated CI. The most commonly used CI is 95% due to historical convention from statistics. The CI will contain the true or correct estimate 95% of the time if we repeat our observation process many, many times. If you accept this probability view of the world, you will know your CIs will include the mean 95% of the time, but you just won't know which 95% of the time.

The standard error is not the same thing as the standard deviation, but they are related. The former is trying to find the variability in the estimate of the population's mean, while the latter is trying to find the variability in the population itself. To compute the standard error from the standard deviation we simply divide the standard deviation by the square root of the sample size. Thus, as the sample size n grows, your estimate for the population's mean becomes tighter, the standard error decreases, while the variability in the population is fixed the whole time.

To calculate upper and lower bound of a confidence interval, use the *empirical rule* as a guide. The empirical says that, approximately, 68 percent of the data lands within one standard deviation of the data, 95 percent within two standard deviations, and 99.7 percent within three. These values change with different distributions, and work best with a normal distribution (also known as a bell curve), but are pretty stable with respect to situation. The actual value for a 95% confidence interval is 1.96.

Continuing with the previous example, calculate the standard error:

$7.41/\sqrt{77} = 0.84$. Thus, you can write the mean as **8.03 ± 0.84 (SE)**, or with the 95% confidence interval as 8.03 (95% CI 6.38 to 9.68), which is

calculated by $8.03 \pm 0.84 \times 1.96 = 1.64$.

WARNING

Always include uncertainty, such as a confidence interval, around estimates such as mean when presenting to a technical audience. Presenting a *naked* mean is considered bad form because it does not allow the reader to see how much uncertainty exists around an estimate.

Based upon statistical convention, you can compare 95% CIs to examine if estimates differ. For example, the estimate of **8.03(6.38 to 9.6895%; CI)** differs from zero because the 95% CI does not include zero. Thus, you can say the air yards by the Detroit Lions in week 2 of 2020 differs from zero when accounting for statistical uncertainty. If you were comparing two estimated means, you could compare both 95% CIs. If the CIs did not overlap, then you can say the means are statistically different.

NOTE

People use 5%/95% out of convention. Wasserstein, Schirm, & Lazar discuss this in a 2019 editorial in the *The American Statistician*. Their editorial presents many different perspectives on alternative methods for statistical inference.

[Chapter 3](#) and other chapters also cover more about statistical inferences. [Chapter 5](#) also covers more about different methods for estimating variances and confidence intervals. Now, enough about theory and hand calculations, let's see how to estimate these values in Python and R.

Filtering and Selecting Columns

To calculate summary statistics with Python and R, first load your data and the required R and Python packages:

```
## R
library(tidyverse)
library(nflfastR)
```

```
# Load all data
pbp_r <- load_pbp(2020)
```

You can use similar code in Python:

```
## Python
import pandas as pd
import numpy as np
import nfl_data_py as nfl

# Load all data
pbp_py = nfl.import_pbp_data([2020])
2020 done.
Downcasting floats.
```

After loading the data, select a subset of the data you want to use. Filtering or querying data is a fundamental skill for data science. At its core, filtering data uses logic statements. These statements can be really frustrating at times - never assume that you've done it correctly the first time. Richard remembers spending a half-a-day in grad school stuck in the computer lab trying to filter out example air quality data with R. Now, this task takes him about 30 seconds.

NOTE

Logic operators simply refer to computer code that compares a statement and provides a binary response. In Python, logical results are either `True` or `False`. In R, logical results are either `TRUE` or `FALSE`.

Both Python and R have different methods for filtering data. We focus on the tools we use, but other useful approaches exist. `pandas` data frames have a `.query()` function that we like to use because it is more compact than `.loc[]`. However, some filtering requires `.loc[]` because `.query()` does not work in all situations. Likewise, the `tidyverse` in R has a `filter()` function. You can use these functions with logical operators.

In R, this can be done using the `filter()` function. Two true statements may be combined with an and (`&`) symbol. For example, so select Green Bay (GB) as the `home_team` and Detroit (DET) as the away team, use `home_team == 'GB' & away_team == 'DET'` with the `filter()` function. Likewise,

the `select()` function allows you to only work with the columns you need, creating an a smaller and easier to use data frame:

```
## R
# Filter out game data
gb_det_2020_r <-
  pbp_r |>
  filter(home_team == 'GB' & away_team == 'DET')

# select pass data
gb_det_2020_pass_r <-
  gb_det_2020_r |>
  select(posteam, yards_after_catch, air_yards,
         pass_location, qb_scramble)
```

Python uses a `query()` function. The input requires a set of quotes around the input, unlike R (for example, pandas uses "home_team == 'GB' & away_team == 'DET'"). pandas uses a list of column names to select specific columns of interest:

```
## Python
# Filter out game data
gb_det_2020_py = \
  pbp_py.query("home_team == 'GB' & away_team == 'DET'")

# Select pass some pass related columns
gb_det_2020_pass_py = \
  gb_det_2020_py[
    ["posteam", "yards_after_catch",
     "air_yards", "pass_location",
     "qb_scramble"]]
```

Calculating Summary Statistics with Python and R

With our dataset in hand, we calculate the summary statistics introduced above using Python and R. In Python, `describe()` the data see similar summaries that also include the median, count, minimum, and maximum values:

```
## Python
print(gb_det_2020_pass_py.describe())
  yards_after_catch  air_yards  qb_scramble
```

count	38.000000	62.000000	181.000000
mean	6.263158	8.612903	0.016575
std	5.912352	10.938509	0.128025
min	-2.000000	-6.000000	0.000000
25%	2.250000	1.250000	0.000000
50%	4.000000	5.000000	0.000000
75%	9.000000	12.750000	0.000000
max	20.000000	50.000000	1.000000

In R, use the `summary()` function:

```
## R
summary(gb_det_2020_pass_r)
  posteam      yards_after_catch    air_yards    pass_location
Length:181          Min.   :-2.000    Min.   :-6.000    Length:181
Class :character    1st Qu.: 2.250    1st Qu.: 1.250    Class
:character
Mode  :character    Median : 4.000    Median : 5.000    Mode
:character
                  Mean   : 6.263    Mean   : 8.613
                  3rd Qu.: 9.000    3rd Qu.:12.750
                  Max.   :20.000    Max.   :50.000
                  NA's    :143     NA's    :119

  qb_scramble
Min.   :0.00000
1st Qu.:0.00000
Median :0.00000
Mean   :0.01657
3rd Qu.:0.00000
Max.   :1.00000
```

One benefit of using `summary()` is that shows the missing or NA values in R. This can help you see possible problems in the data. R also includes `1st Qu.` and `3rd Qu.`, which are the first and third quartiles in R, which as stated above are two of the special quantiles.

Both languages allow you to create customized summaries. For Python, use the `.agg()` function to aggregate the data frame. Use a dictionary insides of Python to tell Pandas which column to aggregate and what functions to use. Recall that Python defines dictionaries using `{"key" : [values]}` notation for shorter notation or `dict("key" : [values])` for a longer notation. In this case, the dictionary uses the column "air_yards" as the key and the aggregating functions as the list values:

```

## Python
print(gb_det_2020_pass_py.agg(
{
    "air_yards": ["min", "max", "mean", "median",
                  "std", "var", "count"]
}
))
            air_yards
min      -6.000000
max     50.000000
mean     8.612903
median    5.000000
std      10.938509
var     119.650978
count    62.000000

```

You can also summarize the data in R in a customized and repeatable way as well by piping the data using `|>` to the `summarize()` function. Then tell R to what functions to use on which columns. Use `min()` for the minimum, `max()` for the maximum, `mean()` for the mean, `median()` for the median, `sd()` for standard deviation, `var()` for the variance, and `n()` for the count. Also to tell R what to call the output columns and assign new names. We chose these names because they are short as well as relatively easy to both type and understand where the new numbers come from:

```

## R
gb_det_2020_pass_r |>
  summarize(min_yac = min(air_yards),
            max_yac = max(air_yards),
            mean_yac = mean(air_yards),
            median_yac = median(air_yards),
            sd_yac = sd(air_yards),
            var_yac = var(air_yards),
            n_yac = n())
# A tibble: 1 × 7
  min_yac max_yac mean_yac median_yac sd_yac var_yac n_yac
  <dbl>   <dbl>     <dbl>       <dbl>   <dbl>   <dbl> <int>
1      NA      NA      NA        NA      NA      NA    181

```

R only give us NA values. What is going on? Recall that these columns have missing data, so tell R to ignore them using the `na.rm = TRUE` option in the functions:

```

## R
gb_det_2020_pass_r |>
  summarize(min_yac = min(air_yards, na.rm = TRUE),
            max_yac = max(air_yards, na.rm = TRUE),
            mean_yac = mean(air_yards, na.rm = TRUE),
            median_yac = median(air_yards, na.rm = TRUE),
            sd_yac = sd(air_yards, na.rm = TRUE),
            var_yac = var(air_yards, na.rm = TRUE),
            n_yac = n())
# A tibble: 1 × 7
  min_yac max_yac mean_yac median_yac sd_yac var_yac n_yac
  <dbl>    <dbl>     <dbl>       <dbl>   <dbl>   <dbl>   <int>
1      -6      50     8.61        5     10.9    120.    181

```

Both Python and R allow you to “group by” variables or calculate statistics by grouping variables such as the mean `air_yards` for each `posteam`. Python has a grouping function, `groupby()`, that can take "posteam" to calculate the statistics by the possession team (notice Python does not use piping). Instead, string together function one after each other. This is due to the object orientated nature of Python compared to the procedural nature of R, both of which have benefits and drawbacks you have to trade off with one another:

```

## Python
print(gb_det_2020_pass_py.groupby("posteam").agg(
{
  "air_yards": ["min", "max", "mean",
                "median", "std", "var", "count"]
})
)
air_yards
      min    max      mean median      std      var count
posteam
DET          -6.0  50.0  8.031250    5.0  11.607796  134.740927    32
GB           -4.0  34.0  9.233333    5.0  10.338023  106.874713    30

```

With Python, you can include a second variable by including a second entry in the dictionary. Also, Pandas, unlike the tidyverse, allows you to calculate different summaries for each variable by changing the dictionary values:

```

## Python
print(gb_det_2020_pass_py.groupby("posteam").agg(
{
  "yards_after_catch": ["min", "max", "mean",
                        "median", "std", "var", "count"],
  "air_yards": ["min", "max", "mean",
                "median", "std", "var", "count"]
})
)

```

```

    "air_yards": ["min", "max", "mean",
                  "median", "std", "var", "count"]
            }
        ))
      yards_after_catch
      min   max   mean ... air_yards
      count
      posteam
      DET           0.0  20.0  6.900000 ... 11.607796 134.740927
      32
      GB           -2.0  19.0  5.555556 ... 10.338023 106.874713
      30

[2 rows x 14 columns]

```

R also includes a “group by” function, `group_by()` that may be used with piping:

```

## R
gb_det_2020_pass_r |>
  group_by(posteam) |>
  summarize(min_yac = min(air_yards, na.rm = TRUE),
             max_yac = max(air_yards, na.rm = TRUE),
             mean_yac = mean(air_yards, na.rm = TRUE),
             median_yac = median(air_yards, na.rm = TRUE),
             sd_yac = sd(air_yards, na.rm = TRUE),
             var_yac = var(air_yards, na.rm = TRUE),
             n_yac = n())
# A tibble: 3 × 8
  posteam min_yac max_yac mean_yac median_yac sd_yac var_yac n_yac
  <chr>     <dbl>    <dbl>     <dbl>       <dbl>    <dbl>    <dbl>    <int>
1 DET         -6       50      8.03        5     11.6    135.     78
2 GB          -4       34      9.23        5     10.3    107.     89
3 <NA>        Inf     -Inf     NaN          NA     NA      NA      14

```

A Note about Presenting Summary Statistics

The key for presenting summary statistics is to make sure you use the information available to you to effectively tell your story. Firstly, to know your target audience is extremely important. For example, if you’re talking to Cris Collinsworth about his next *Sunday Night Football* broadcast (something Eric did at his previous job) or to your buddies at the bar during a game, you’re going to present the information differently.

Furthermore, if you’re presenting your work to the Director of Research and Strategy for an NFL team, you’re probably going to have to supply different, specifically more, information than in the aforementioned two examples.

Likewise, when talking to the Director of Research and Strategy, you will likely need to justify both your estimates and your methodological choices.

Conversely, unless you’re having beers with Eric and Richard (or other quants), you probably will not be discussing modeling choices over beers!

The “why” is key and you’ll have to dig into data and truly understand it well, so that you can speak it in a number of different languages. For example, is the dynamic you’re seeing due to coverage differences, the wide receivers, or changes in the quarterback’s fundamental ability?

Secondly, use statistics and modeling to support your story, but do not use them as your entire story. Say “Drew Brees is still the most accurate passer in football, even after you adjust for situation” rather than “Drew Brees has the highest completion percentage above expected. Period.” Adding context to one’s work is something that we, as authors, have noticed helps the best quantitative people stand out compared to many quantitative people. In fact, communication skills about numbers helped both us get our current jobs.

Thirdly, while a picture may be worth a thousands words, walk your reader through the picture. A graph with no context is likely worse than no graph at all because all the graph without context will do is confuse your audience. For a non-technical audience, you may include a figure and mention the “averages” in your words. Thus, the raw summary statistics may not even be shown in your writing. For more technical audiences, include the details and uncertainty either in text for one or two number or in a table or supplemental materials for more summary statistics.

Improving your presentation

We have found there are two good ways to improve our presenting of summary statistics. First, present early and present often to people who will give you constructive feedback. Make sure they can understand your message, and if they cannot, ask them what is unclear and figure out how to more clearly make your point. For example, the authors like to give lectures and seminar to students

because we will ask our students how they might explain a figure and then they help us to more clearly think about data. Also, if you cannot explain concepts to high school and college students, you do not clearly understand the ideas well.

Second, look at other people's work. Read blogs, read other books, read articles, watch or listen to podcasts. Other people's examples will help you see what is clear and what is not. Besides casual reading, read critically. What works? What does not work? Why did the authors make a choice? How would you help the author better explain their findings to you? If you have a chance, ask the authors if you see them or interact with them on social media. A diplomatic tweet, will likely start a conversation. For example, replying to a tweet *I liked your model output and the insight it gave me to Friday's game. Why did you use X rather than Y?* Conversely, replying to a tweet with *your model sucked, you should use my favorite model.* will likely be ignored or possibly start a pointless flame war and decrease not only the original poster's view of you, but also other people who read the tweet.

Exercises

1. Repeat the processes within this chapter with a different game.
2. Repeat the processes within this chapter with a different feature, like rushing yards.

Future readings

Many different books describe introductory statistics. If you want to learn more about statistics, we suggest reading the first 1-2 chapters of several books until you find one that *speaks* to you. Some books you may wish to consider include:

- *Advancing into Analytics: From Excel to Python and R* by George Mount (O'Reilly Media 2021). This book assumes a reader knows Excel well, but then helps the reader to transition to either R or Python. The book covers the basis of statistics.
- *Statistical Inference via Data Science: A ModernDive into R and the Tidyverse* by Chester Ismay and Albert Y. Kim (CRC Press 2019), also

updated at the book's [homepage](#). This book provides a robust introduction to statistical inferences for people who also want to learn R.

- *Practical Statistics for Data Scientists*, Second Edition, by Peter Bruce, Andrew Bruce (O'Reilly Media, 2020). This book provides an introduction to statistics for people who already know some R or Python.
- *Introductory Statistics with R* by Peter Dalgaard (Springer 2008) is a classic book, while somewhat dated for code, was the book Richard first learned statistics (and R) from.
- *Essential Math for Data Science* by Thomas Nield (O'Reilly Media 2021) provides a gentle introduction to statistics as well as mathematics for applied data scientists.

Appendix C. Data Wrangling Fundamentals

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be Appendix C in the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

“Tidy datasets are all alike, but every messy dataset is messy in its own way.”
– Hadley Wickham

This appendix focuses on some basics of *data wrangling*, or the process of formatting and cleaning data prior to using the data. We include some common, but sometimes confusing tools we use on a regular basis. We need a large toolbox, because, as noted by Hadley Wickham above, each messy data has its own pathologies. For a more in depth side-by comparison of Python and R, check out appendix in *Python and R for the Modern Data Scientist* by Boyan Angelov and Rick Scavetta (O'Reilly 2021).

NOTE

Data wrangling has many terms because almost everybody working with data needs to clean the data. Other terms include *data cleaning*, *data formatting*, *data tiding*, *data transformation*, *data manipulation*, *data munging*, and *data mutating*. Basically, people use different terms, so do not be surprised if you see different terms in different sources. Also, in our experience, people inconsistently use different terms. The key take home is that you'll need to clean, format, transform, or otherwise change your own data at some point. Hence, we included this appendix.

Logic Operators

Logic operators are the same across most languages, including Python and R. The upcoming [Table C-1](#) lists some common operators. Explore these operators by creating a vector in R:

```
## R
score <- c(21, 7, 0, 14)
team <- c("GB", "DEN", "KC", "NYJ")
```

Or, create arrays with `numpy` in Python:

```
## Python
import numpy as np
score = np.array([21, 7, 0, 14])
team = np.array(['GB', 'DEN', 'KC', 'NYJ'])
```

WARNING

Python's `numpy`'s arrays differ from base Python's lists and have different behaviors with mathematical functions.

Basic operators are easy to figure out, like `>` for greater than or `<` for less than. For example, you can see which elements are greater than 7 in Python:

```
## Python
score > 7
array([ True, False, False,  True])
```

As you can see, when you use these operators with an array, the operation is performed against each element individually, and all the results are placed into a new array. This is similar to how the operations work in R, as you will see. For example, with R you can see which elements are less than 15 in R:

```
## R
score < 15
[1] FALSE  TRUE  TRUE  TRUE
```

Less than or equal to or greater than or equal to use the equals sign plus the

operator. `>=` is greater than or equal to, and `<=` is less than or equal to. For example, compare the next code example to the previous code example:

```
## Python
score <= 14
array([False,  True,  True,  True])
```

Other operators are less obvious. Because we already use `=` to define objects, `==` is used for equals. For example, you can find elements of `team` that are equal to `GB`. Make sure you put `team` in quotes (`"GB"`). Otherwise, the computer thinks you are trying to use an object named `GB`:

```
## Python
team == "GB"
array([ True, False, False, False])
```

Using an `in`-type operator is really useful when there are a number of ways to chart a player playing a similar position. For example, “DE” (defensive end), “OLB” (outside linebacker), and “ED” (edge defender) mean similar things in football, and subsetting a dataset for when a player is designated as any one of those labels is often something one does in analysis.

In numpy, you can do this the `.isin()` function:

```
## Python
position = np.array(['QB', 'DE', 'OLB', 'ED'])
np.isin(position, ['DE', 'OLB', 'ED'])
array([False,  True,  True,  True])
```

pandas has a similar function for dataframes, covered in [“Filtering and Sorting Data”](#).

R has a slightly different operator, an `%in%` function:

```
## R
position <- c("QB", "DE", "OLB", "ED")
position %in% c("DE", "OLB", "ED")
[1] FALSE TRUE TRUE TRUE
```

When using `%in%`, be careful with the order. For example, compare `position`

```
%in% c("DE", "OLB", "ED") from the last example with c("DE",  
"OLB", "ED") %in% position:
```

```
## R  
c("DE", "OLB", "ED") %in% position  
[1] TRUE TRUE TRUE
```

TIP

Using `in` operators can be hard. We will often grab a test subset of our data to make sure our code works as expected. More broadly, do not trust your code until you have convinced yourself that your code works as expected. Casually, use `print()` statements to peak at your code and make sure the code does what you think the code is doing. We do this for one-off projects. Formally, unit-testing exists as a method to test code. Python comes with the `unittest` package and R has the `testthat` package for formal testing. We use unit-testing on code we plan to re-use, or, import code where failure has a large cost.

You can also string together operators using the `and` operator (`&`) or the `or` operator (`|`). Using multiple operators requires the values are in order as pairs. Our example implies that `score` corresponds to `team`. Both vectors are of length 4 with our examples.

For example, you can see what entries are greater than or equal to 7 for `score` and have a `team` value of "DEN". When working with the `numpy` arrays, you need to use the `where()` function, but this logic will be the same and use similar notation with Pandas later in this chapter. The results reveal which entry meets the criteria:

```
## Python  
np.where((score >= 7) & (team == "DEN"))  
(array([1]),)
```

You can also use an `or` operator for a similar comparison to see what values of `score` are greater than 7 *or* what values of `team` are equal to "DEN":

```
## R  
score > 7 | team == "DEN"  
[1] TRUE TRUE FALSE TRUE
```

You can string together multiple conditions parentheses. For example, you can see what has `score` values greater than or equal to 7 *and* `team` equal to “DEN” *or* `score` equal to 0:

```
## Python
np.where((score >= 7) & (team == "DEN") | (score == 0))
(array([1, 2]),)
```

Likewise, similar notation may be used in R:

```
## R
(score >= 7 & team == "DEN") | (score == 0)
[1] FALSE TRUE TRUE FALSE
```

Table C-1. Table 1.1: Common logical operators. Note that pandas sometimes uses `~` rather than `!` for not in some situations.

Symbol	Example	Name	Question
<code>==</code>	<code>score == 2</code>	equals	Is score equal to 2?
<code>!=</code>	<code>score != 2</code>	not equals	Is score not equal to 2?
<code>></code>	<code>score > 2</code>	greater than	Is score greater than 2?
<code><</code>	<code>score < 2</code>	less than	Is score less than 2?
<code>>=</code>	<code>score >= 2</code>	greater than or equal to	Is score greater than or equal to 2?
<code><=</code>	<code>score <= 2</code>	less than or equal to	Is score less than or equal to 2?
<code> </code>	<code>(score > 2) (team == "GB")</code>	or	Is score less than 2 or team equal to GB?

&	(score > 2) & (team == "GB")	Is score less than 2 and team equal to GB?
---	------------------------------	--

Filtering and Sorting Data

In the previous section, you learned about logical operators. These functions serve as the foundation of filtering data. In fact, when we get stuck with filtering, we often build small test cases like the ones in “[Logic Operators](#)” to make sure we understand our data and how our filters work (or, as is sometimes the case, do not work).

TIP

Filtering can be hard. Start small and build complexity into your filtering commands. Keep adding details until you are able to solve your problem. Sometimes, you might need to do two or more smaller filters rather than one grand filter operation. This is okay. Get your code working before worrying about optimization.

You will work with the Green Bay-Detroit data from the second week of the 2020 season. First, read in the data and do a simple filter to look at plays that had yards after catch greater than 15 yards to get an idea of where some big plays were generated.

In R, load the `tidyverse` and `nflfastR` packages and then load the data for 2020:

```
## R
library(tidyverse)
library(nflfastR)

# Load all data
pbp_r <- load_pbp(2020)
```

In Python, import the `pandas`, `numpy`, and `nfl_data_py` packages and then load the data for 2020:

```
## Python
```

```

import pandas as pd
import numpy as np
import nfl_data_py as nfl

# Load all data

pbp_py = nfl.import_pbp_data([2020])
2020 done.
Downcasting floats.

```

In R, use the `filter()` function next. The first argument into filter is `data`. The second argument is the `filter` criteria. Filter out the Detroit at Green Bay game and select some passing columns:

```

# Filter out game data
gb_det_2020_r_pass <-
  pbp_r |>
  filter(home_team == 'GB' & away_team == 'DET') |>
  select(posteam, yards_after_catch, air_yards,
         pass_location, qb_scramble)

```

Next, `filter()` the plays with `yards_after_catch` that were greater than 15:

```

gb_det_2020_r_pass |>
  filter(yards_after_catch > 15)
# A tibble: 5 × 5
  posteam yards_after_catch air_yards pass_location qb_scramble
  <chr>          <dbl>     <dbl> <chr>           <dbl>
1 DET              16        13 left            0
2 GB               19         3 right           0
3 GB               19         6 right           0
4 DET              16         1 middle          0
5 DET              20        16 middle          0

```

TIP

With R and Python, you do not always need to use argument names. Instead, the languages match arguments with their predefined order. This order is listed in the help files. For example, with `gb_det_2020_r_pass |> filter(yards_after_catch > 15)`, you could have written `gb_det_2020_r_pass |> filter(filter = yards_after_catch > 15)`. We usually define argument names for more complex functions or when we want to be clear. It is better to err on the side of being explicit and using the argument names because doing this makes your code easier to read.

Notice in this example that plays that generated a lot of yards after the catch come in many shapes and sizes, including short throws with one yard in the air, and longer throws with 16 yards in the air. You can also filter with multiple arguments using the and operator, `\&`. For example, you can filter by yards after catch being greater than 15 and Detroit on offense:

```
## R
gb_det_2020_r_pass |>
  filter(yards_after_catch > 15 & postteam == "DET")
# A tibble: 3 × 5
  postteam yards_after_catch air_yards pass_location qb_scramble
  <chr>          <dbl>     <dbl> <chr>                  <dbl>
1 DET              16        13  left                   0
2 DET              16         1  middle                 0
3 DET              20        16  middle                 0
```

However, what if you want to look at plays with yards after catch being greater than 15 yards or air yards being greater than 20 year and Detroit the offensive team? If you try `yards_after_catch > 15 | air_yards > 20 & postteam == "DET"` in the filter, you get results with both Green Bay and Detroit rather than only Detroit. This is because the order of the operations is different than you intended:

```
## R
gb_det_2020_r_pass |>
  filter(yards_after_catch > 15 | air_yards > 20 &
         postteam == "DET")
# A tibble: 9 × 5
  postteam yards_after_catch air_yards pass_location qb_scramble
  <chr>          <dbl>     <dbl> <chr>                  <dbl>
1 DET              16        13  left                   0
2 GB               19         3  right                  0
3 DET              NA        28  left                   0
4 DET              NA        28  right                  0
5 GB               19         6  right                  0
6 DET              16         1  middle                 0
7 DET               0        24  right                  0
8 DET              20        16  middle                 0
9 DET              NA        50  left                   0
```

You get all plays with yards after catching being greater than 15 or all plays with yards greater than 20 and Detroit starting with possession of the ball. Instead,

add a set of parentheses to the filter: `(yards_after_catch > 15 | air_yards > 20) & posteam == "DET".`

WARNING

The *order of operations* refers to how people do math functions and computers evaluate code. The key takeaway is that both order and grouping of function changes the output. For example, $1 + 2 * 3 = 1 + 6 = 7$ is different from $(1 + 2) * 3 = 3 * 3 = 9$. When you combine operators, the default order of operations sometimes leads to unexpected outcomes, like in the previous example you expected to filter out GB but did not. To avoid this type of confusion, parentheses help you to explicitly choose the order that you intend.

The use of parentheses in both coding and mathematics align, so the order of operations start with the inner most set of parentheses and then move outward:

```
## R
gb_det_2020_r_pass |>
  filter((yards_after_catch > 15 | air_yards > 20) &
         posteam == "DET")
# A tibble: 7 × 5
  posteam yards_after_catch air_yards pass_location qb_scramble
  <chr>          <dbl>     <dbl>    <chr>           <dbl>
1 DET              16        13  left             0
2 DET              NA        28  left             0
3 DET              NA        28  right            0
4 DET              16        1   middle            0
5 DET              0         24  right            0
6 DET              20        16  middle            0
7 DET              NA        50  left             0
```

You can also change the filter to only look at possession teams that are not Detroit using the not equal to operator, `!=`. In this case, the “not equal to” operator gives you Green Bay’s admissible offensive plays, but this would not always be the case. For example, if you were working with season long data with all teams, the “not equal to” operator would give you data for the 31 other NFL teams:

```
## R
gb_det_2020_r_pass |>
  filter((yards_after_catch > 15 | air_yards > 20) &
         posteam != "DET")
```

```
# A tibble: 8 × 5
#> #> posteam yards_after_catch air_yards pass_location qb_scramble
#> #> <chr>          <dbl>     <dbl> <chr>           <dbl>
#> 1 GB            NA        26 left            0
#> 2 GB            NA        25 left            0
#> 3 GB            19       3 right           0
#> 4 GB            NA        24 right           0
#> 5 GB            4        26 right           0
#> 6 GB            NA        28 left            0
#> 7 GB            19       6 right           0
#> 8 GB            7        34 right           0
```

In Python with `pandas`, filtering is done with similar logical structure to the `tidyverse` in R, but with different syntax. First, Python uses a `.query()` function. Second, the logical operator is inside of quotes:

```
## Python
gb_det_2020_py_pass = \
    pbp_py \
    .query("home_team == 'GB' & away_team == 'DET'") \
    [["posteam", "yards_after_catch", "air_yards",
      "pass_location", "qb_scramble"]]

print(gb_det_2020_py_pass.query("yards_after_catch > 15"))
#> #> posteam yards_after_catch air_yards pass_location qb_scramble
#> #> 4034   DET        16.0      13.0      left        0.0
#> #> 4077   GB         19.0      3.0       right       0.0
#> #> 4156   GB         19.0      6.0       right       0.0
#> #> 4171   DET        16.0      1.0       middle      0.0
#> #> 4199   DET        20.0      16.0      middle      0.0
```

Notice the or operator, `\|`, works the same with both languages:

```
## Python
print(gb_det_2020_py_pass.query("yards_after_catch > 15 | air_yards > 20"))
#> #> posteam yards_after_catch air_yards pass_location qb_scramble
#> #> 4034   DET        16.0      13.0      left        0.0
#> #> 4051   GB         NaN       26.0      left        0.0
#> #> 4055   GB         NaN       25.0      left        0.0
#> #> 4077   GB         19.0      3.0       right       0.0
#> #> 4089   DET        NaN       28.0      left        0.0
#> #> 4090   DET        NaN       28.0      right       0.0
#> #> 4104   GB         NaN       24.0      right       0.0
#> #> 4138   GB         4.0       26.0      right       0.0
#> #> 4142   GB         NaN       28.0      left        0.0
```

4156	GB	19.0	6.0	right	0.0
4171	DET	16.0	1.0	middle	0.0
4176	DET	0.0	24.0	right	0.0
4182	GB	7.0	34.0	right	0.0
4199	DET	20.0	16.0	middle	0.0
4203	DET	NaN	50.0	left	0.0

WARNING

In R or Python, you can use with single quotes ('') or double quotes (""). When using functions such as `.query()` in Python, you see why the languages contain two different methods for quoting. You could use `"postteam == 'DET'"` or `'postteam == "DET"`. The languages do not care if you use single or double quotes, but you need to be consistent within the same function call.

In Python, when your code gets too long to easily read on a line, you need a backslash, \, for Python to understand the line break. This is because Python treats white space as a special type of code, where as R usually treats *white space*, such as spaces, indentations, or line breaks, simply as aesthetic. To a novice, this part of Python can be frustrating, but the use of white space is actually a beautiful part of the language once you gains experience to appreciate it.

Next, look at the use of parentheses with the or operator and and operator, just like R:

```
print(gb_det_2020_py_pass.query("(yards_after_catch > 15 | \
                                    air_yards > 20) & \
                                    postteam == 'DET'"))
      postteam  yards_after_catch  air_yards pass_location  qb_scramble
4034      DET          16.0       13.0        left          0.0
4089      DET           NaN       28.0        left          0.0
4090      DET           NaN       28.0       right          0.0
4171      DET          16.0       1.0       middle          0.0
4176      DET           0.0       24.0       right          0.0
4199      DET          20.0       16.0       middle          0.0
4203      DET           NaN       50.0        left          0.0
```

Cleaning

Having accurate data is important for sports analytics, as the edge in sports like

football can be as little as one or two percentage points over your opponents, the sportsbook, or other players in a fantasy football. Cleaning data by hand using programs such as Excel can be tedious and also leaves no log of what values were changed. Also, fixing one or two systematic errors by hand can easily be done with Excel, but, fixing or reformatting thousands of cell in Excel would be difficult and time consuming. Luckily, you can use scripting to help you clean data.

NOTE

When estimating which team will win a game, the *edge* refers to the difference between the predictor's estimated probability and the market's estimated probability (plus the book's commission, or vigorish). For example, if sportsbooks are offering the Minnesota Vikings at a price of 2-1 to win a game against the Green Bay Packers in Lambeau Field, they are saying that to bet the Vikings you need to believe that they have more than a $1/(2+1) \times 100\% = 33.3\%$ chance to win the game. If you make Vikings 36% to win the game, you have a three percent edge betting the Vikings. As information and the synthesizing of information have become more prevalent, edges have become smaller (e.g. the markets have become more efficient). Professional bettors are always in search of a) better data and b) better ways to synthesize data, to outrun the increasingly-efficient markets they play in.

Consider this example DataFrame in pandas:

```
wrong_number = \
pd.DataFrame({ "col1": [ "a", "b"],
                "col2": [ "10", "12"],
                "col3": [ 2, 44]})
```

Notice how `col2` has a `10` (one-oh) rather than a `10` (one-zero or ten). These types of mistakes are actually fairly common in hand-entered data. This may be fixed using code.

NOTE

Both R and Python allow you to access dataframes using a coordinate like system with rows as the first entry and columns as the second entry. Think of this like a game of Battleship or Bingo when people call out cells like `A4` or `B2`. pandas has `.loc[]` that to access rows or columns by names. For example, to access the first value in the `posteam` column of the play-by-play data, run `pbp_py.loc[1, "posteam"]` (1 is the row name or index and `posteam` is the column name). To access the first row of the first column, run `print(pbp_py.iloc[1, 0])`. Compare these two methods. What column is `0`? It is better to use filters or explicit names. This way, if your data changes, you call the correct cell.

Also, this way future you and other people will also know why you are trying to access specific cells.

Use the locate function, `.loc()`, to locate the wrong cell. Also, select the column, `col2`. Last, replace the wrong value with a `10` (ten):

```
## Python
wrong_number.loc[wrong_number.col2 == "10", "col2"] = 10
```

Look at the dataframe's information, though, and you will see that `col2` is still an object rather than a number or integer:

```
## Python
wrong_number.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2 entries, 0 to 1
Data columns (total 3 columns):
 #   Column   Non-Null Count   Dtype  
---  --          --          --      
 0   col1      2 non-null     object 
 1   col2      2 non-null     object 
 2   col3      2 non-null     int64  
dtypes: int64(1), object(2)
memory usage: 176.0+ bytes
```

WARNING

Both R and Python usually require users to save data files as outputs after editing (a counter-example being the `inplace=True` option in some `pandas` functions) Otherwise, the computer will not save your changes. Failure to update or save objects can cost you hours of debugging code, as we have learned from our own experiences.

Change this by using the `to_numeric()` function from `pandas` and then look at the information for the dataframe. Next, save the results to `col2` and rewrite the old data. If you skip this step, the computer will not save your edits:

```
## Python
wrong_number["col2"] = \
    pd.to_numeric(wrong_number["col2"])
wrong_number.info()
<class 'pandas.core.frame.DataFrame'>
```

```

RangeIndex: 2 entries, 0 to 1
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype  
--- 
 0   col1    2 non-null      object 
 1   col2    2 non-null      int64  
 2   col3    2 non-null      int64  
dtypes: int64(2), object(1)
memory usage: 176.0+ bytes

```

Notice how now the column has been changed to an integer.

If you want to save these changes for later, you can use the `to_csv()` function to save the outputs. Generally, you will want to use a new filename that makes sense to you now, to others, and to your future self. Because the dataframe does not have a meaningful row names or index, tell pandas to not save this information using `index=False`:

```

## Python
wrong_number.to_csv("wrong_number_corrected.csv", index = False)

```

R uses slightly different syntax. First, use the `mutate()` function to change the column. Next, tell R to change `col2` using `col2 = ...`. Then use the `ifelse()` function to tell R to change `col2` if it is equal to `10` (one oh) to be `10` (one-zero or ten), else use the current value in `col2`:

```

## R
wrong_number <-
  tibble(col1 = c("a", "b"),
         col2 = c("10", "12"),
         col3 = c(2, 44))
wrong_number <-
  wrong_number |>
  mutate(col2 = ifelse(col2 == "10", 10, col2))

```

Next, just like in Python, change `col2` to be numeric. In R, use the `as.numeric()` function. Then look at the dataframes structure using `str()`:

```

## R
wrong_number <- mutate(wrong_number, col2 = as.numeric(col2))
str(wrong_number)
tibble [2 x 3] (S3: tbl_df/tbl/data.frame)

```

```
$ col1: chr [1:2] "a" "b"  
$ col2: num [1:2] 10 12  
$ col3: num [1:2] 2 44
```

Finally, just like in Python, save the file using a name that makes sense to both the current you and future you. Hopefully this name makes sense to other people. Creating names can be one of the most difficult parts of programming. With R, use the `write_csv()` function:

```
## R  
write_csv(x = wrong_number,  
          file = "wrong_numbers_corrected.csv")
```

WARNING

Python uses `False` for the logical false and `True` for the logical true. R uses `FALSE` for false and `TRUE` for true. If you are switching between the languages, be careful with these terms.

Piping in R

With programming, sometimes you want to pass outputs from one function to another without needing to save the intermediate outputs. In mathematics this is called *composition*, and while teaching college math classes, Eric observed this to be one of the more misunderstood procedures due to the confusing notation. In computer programming, this is called *piping* because outputs are piped from one function to another.

Luckily, R has allowed composition through the piping operators with the `tidyverse` with a pipe function, `%>%`. As of R version 4.1 released in 2021, base R also now include a `|>` pipe operator. We use the base R pipe operator in this book, but you may see both *in the wild* when looking at other peoples code or websites.

NOTE

The `tidyverse` pipe allows piping to any function's input option using a period `.`. This period is optional with the `tidyverse` pipe. And, `tidyverse` pipe will, by default, use the first function input with piping. For example, you might code `read_csv("my_file.csv") %>% func(x =`

`col1, data = .), or read_csv("my_file.csv") %>% function(col1). With |>, you can only pass to the first input, thus you would need to define all inputs prior to the one you are piping (in this case, data). With the |> pipe, your code would be written as read_csv("my_file.csv") |> function(x = col1).`

WARNING

Any reference material can become dated, especially online tutorials. The piping example demonstrates how any tutorial created before R 4.1 would not include the new piping notation. Thus, when using a tutorial, examine when the material was written and ensure you can recreate a tutorial before applying it to your problem. And, when using sites such as [StackOverflow](#), we look at several of the top answers and questions to make sure the accepted answer has not become outdated as languages change. The best answer in 2013 may not be the best answer in 2023.

We cover piping here for two reasons. First, you will likely see it when you start to look at other people's code as you teach yourself. Second, piping allows you to be more efficient with coding once you get the hang of it.

Checking and Cleaning Data for Outliers

Data often contains errors. Perhaps people collecting or entering the data made a mistake. Or, maybe an instrument like a weather station malfunctioned.

Sometimes, computer systems corrupt or otherwise change files. In football, quite often there will be errors in things like number of air yards generated, yards after the catch earned, or even the player targeted. Resolving these errors quickly, and often through data wrangling, is a required process of learning more about the game. Chapter 2 presented tools to help you catch these errors.

You'll go through and find and remove an outlier with both languages. Revisiting the `wrong_number` dataframe, perhaps `col3` should only be single digits. The `summary()` function would help you see this value is wrong in R:

```
## R
wrong_number |>
  summary()
  col1           col2           col3
Length:2        Min.   :10.0    Min.   : 2.0
Class :character 1st Qu.:10.5   1st Qu.:12.5
```

```
Mode   :character    Median :11.0    Median :23.0
       Mean   :11.0    Mean   :23.0
       3rd Qu.:11.5   3rd Qu.:33.5
       Max.   :12.0    Max.   :44.0
```

Likewise, the `describe()` function in Python would help you catch an outlier:

```
## Python
wrong_number.describe()
      col2      col3
count  2.000000  2.000000
mean   11.000000 23.000000
std    1.414214 29.698485
min    10.000000 2.000000
25%   10.500000 12.500000
50%   11.000000 23.000000
75%   11.500000 33.500000
max   12.000000 44.000000
```

Using the tools covered in the previous section, you can remove the outlier.

Merging Multiple Datasets

Sometimes you will need to combine datasets. For example, often you will want to adjust the results of a play - say the number of passing yards - by the weather in which the game was played. Both `pandas` and the `tidyverse` readily allow merging datasets. For example, perhaps you have team and game data you want to merge both datasets. Or, maybe you want to merge weather data to the play-by-play data.

For this example, create two dataframes and then merge them together. One dataframe will be city information that contains the teams' names and city. The other will be a schedule. We have you create a small example for multiple reasons. First, a small toy dataset is easier to handle and see compared to a large dataset. Second, we often create toy datasets to make sure our merges work.

TIP

When learning something new (like merging dataframes), start with a small example you understand. The small example will be easier to debug, fail faster, and easier to understand compared to a large example or actual dataset.

You might be wondering, why merge these dataframes? We often have to do merges like this when summarizing data because we want or need a prettier name. Likewise, we often need to change names for plots. Next, you may be wondering, why not hand type these values into a spreadsheet? Hand typing can be tedious and error prone. Plus, doing tens, hundreds, or even thousands of games would take a long time to hand type.

As you create the dataframes in R, remember that each column you create is a vector:

```
## R
library(tidyverse)
city_data <- data.frame(city = c("DET", "GB", "HOU"),
                         team = c("Lions", "Packers", "Texans"))
schedule <- data.frame(home = c("GB", "DET"),
                        away = c("DET", "HOU"))
```

As you create the dataframes in Python, remember that the `DataFrame()` function uses a dictionary to create columns and elements in the columns:

```
## Python
import pandas as pd
city_data = \
    pd.DataFrame({"city" : ["DET", "GB", "HOU"],
                  "team" : ["Lions", "Packers", "Texans"]})
schedule = \
    pd.DataFrame({"home" : ["GB", "DET"],
                  "away" : ["DET", "HOU"]})
```

Now that you have the datasets, use them to explore different merges. Both `pandas` and the `tidyverse` base their merge functions upon SQL. The join functions require a common, shared key or multiple keys between the two dataframes. In the `tidyverse`, this argument is called *by*, for example joining city and schedule dataframes by *team name* and *home team* columns. In `pandas`, this argument is called *on*, for example joining city and schedule dataframes on *team name* and *home team* columns.

There are four main joins we use on a regular basis, and these are included with the `tidyverse` and `pandas`. `pandas` has both a `merge()` and a `join()`

function. `merge()` contains almost everything that `join()` does, plus some more, so we will only include `merge()` here. With both Python and R, there are two datasets, a left one and a right one. The left dataset is the one on the left (or the first datasets) and the right dataset is the one on the right (or the second dataset).

For the example, you want to create a new dataframe that includes both the schedule and the teams' names. Use this to explore the different types of joins. Think of this example like the fairy tale of Goldilocks and the four joins (based on the original story of [Goldilocks and the Three Bears](#)). Rather than a girl trying the bear's beds and food, you'll be exploring data joins, listed in [Table C-2](#). This problem actually has two steps. The first step is to add in the home team's name. The second step is to add in the away team's name. At the end, we will show you the complete workflow because it also involves renaming columns.

TIP

Football analytics, like the broader field of data science, usually involves breaking big jobs down into smaller jobs. As you become more experienced, you will become better at seeing the small steps and knowing where and how to re-use them. When faced with intimidating problems, we break them down into smaller steps that we can readily solve. Often our first step is to write or draw out our coding needs, much like you may have outlined a paper in high school or college before writing the paper.

First, examine a *full* or *outer join*. This merges both dataframes based upon all values in both dataframes' keys. If one or both keys contain values not found in the other dataset, these are replaced by missing values (`NA` in R, `NaN` in Python). For both languages, `schedule` will be your left dataframe and `city_data` will be your right dataframe. Because both dataframes do not have the same key (or, specifically the column with the same names), the computer needs to know how to pair up the keys (specifically, which columns link the two dataframes).

In R, use the `full_join()` function. Put `schedule` in first, followed by `city_data`. Tell R to *join* the dataframes using `home` as the left key matching up with `city` as the right key:

```
## R
print(
  full_join(schedule, city_data,
```

```

by = c("home" = "city"))
)
home away team
1  GB  DET Packers
2  DET HOU Lions
3  HOU <NA> Texans

```

Notice how you get three entries because the `city_data` has three rows. The missing value is replaced by NA. Notice how R dropped the duplicate column and only has three columns.

In Python, use the `.merge()` function on the `schedule` dataframe. And notice that `schedule` is on the left. The first argument is `city_data`. Tell Pandas to how to merge, specifically an `outer` merge. Then tell Pandas to use `home` as the left key and `city` as the right key:

```

## Python
print(schedule.merge(city_data, how = "outer",
                      left_on = "home", right_on = "city"))
   home away city      team
0    GB   DET    GB  Packers
1    DET   HOU    DET    Lions
2    NaN   NaN   HOU    Texans

```

Notice Pandas kept all four columns. Also, note how both `home` and `away` are `NaN` for the new dataframe.

NOTE

This example demonstrates how Python tends to be an object-orientated language and R tends to be a functional language. Python uses `.merge()` as an object contained by the dataframe `schedule`. R uses a `full_join()` as a function on two different objects, `schedule` and `city_data`. Although R and Python both contain object-orientated and functional features, this example nicely demonstrates the underlying philosophies of the two languages.

Think of this distinction of language types similar to how some football teams are built for a run offense and others as a pass offense. Under certain circumstances one language can be better than the other, but usually both contain the tools for given job. Advanced data scientists recognize these tradeoffs between languages and will switch languages to fit their needs.

Next, do an *inner join*. This only joins the shared key values. Whereas an outer join may possibly grow dataframes, an inner join shrinks dataframes. The R syntax is very similar to the previous example, only the function name changes. However, notice how the output only has three values:

```
## R
print(inner_join(schedule, city_data, by = c("home" = "city")))
  home away   team
1   GB   DET Packers
2   DET   HOU   Lions
```

Like R, the Python code is similar. In Python, use the same function, but a different `how` argument:

```
## Python
print(schedule.merge(city_data, how = "inner",
                     left_on = "home", right_on = "city"))
  home away city   team
0   GB   DET   GB Packers
1   DET   HOU   DET   Lions
```

Next, do a *right join*. The right join keeps all of the values from the right dataframe. For this specific case, the outputs are the same as the outer join. This is an artifact of the example and may not always be the case. With R, simply change the function name to be `right_join()`:

```
## R
print(right_join(schedule, city_data, by = c("home" = "city")))
  home away   team
1   GB   DET Packers
2   DET   HOU   Lions
3   HOU <NA> Texans
```

With Python, change the `how` to be `right`:

```
## Python
print(schedule.merge(city_data, how = "right",
                     left_on = "home", right_on = "city"))
  home away city   team
0   DET   HOU   DET   Lions
1   GB   DET   GB Packers
2   NaN   NaN   HOU Texans
```

A *left join* is the opposite of a right join. This keeps all of the values from the left dataframe. In fact, rather than switching the function, you could switch the order of inputs. Consider merging dataframes A and B in Python that share a common column, key:

```
## Python
A.merge(B, how = "left", on = "key")
```

This could also be written in reverse:

```
## Python
B.merge(A, how = "right", on = "key")
```

Here is what the R code and output looks like:

```
## R
print(left_join(schedule, city_data, by = c("home" = "city")))
  home away   team
1   GB   DET Packers
2   DET   HOU   Lions
```

The Python code also looks similar to the right join. For both of the outputs, the left join was the same as the inner join. This is an artifact of the example choice and will not always be the case. Here, the left dataframe had fewer rows than the right dataframe. Hence, this occurred in the example:

```
## Python
print(schedule.merge(city_data, how = "left",
                     left_on = "home", right_on = "city"))
  home away city   team
0   GB   DET    GB Packers
1   DET   HOU    DET   Lions
```

Table C-2. Table 6.1: Common join types in R and Python.

Name	Brief description	Tidyverse function	Pandas merge how
Full/outer join	Merges based upon	full_join(left_data, right	left_data.merge(right_data, how =

	all key values	_data)	"outer"
Inner join	Only merges based upon shared key values	inner_join(left_data, right_data)	left_data.merge(right_data, how = "inner")
Left join	Only merges based upon <i>left</i> data's key values	left_join(left_data, right_data)	left_data.merge(right_data, how = "left")
Right join	Only merges based upon <i>right</i> data's key values	right_join(left_data, right_data)	left_data.merge(right_data, how = "right")

Let's return to the initial problem: "How do you merge the dataframe to include the team names for both the home and away teams?"

Multiple solutions exist, as is often the case with programming. We use multiple left joins because we think about adding data to schedule and putting this dataframe on the left. However, you might think about the problem differently, which is okay. In fact, you might be able to think about and come up with a better way to do this that is either quicker, easier to read, or uses less code.

NOTE

Unlike high school math, both statistics and coding often have no single best or right way to do something. Instead, many unique solutions exist. Some people play a game called *code golf* where they try to solve a problem using the fewest lines of code such as the Stack Exchange [Code Golf page](#). But the fewest lines of code is usually not the best answer in real life. Instead, focus on writing code that you and other people can read later. Also new tools such as GitLab's Copilot can help you see and compare different methods for coding the same task.

So, we will use a series of left joins (although, we could also do everything in reverse using right joins). Here is our step-by-step solution:

1. Merge in for home team.

2. Rename column in R, rename and delete column in Python.
3. Merge in for away team. This step is needed for clarity and to avoid duplicate names.
4. Rename column in R, rename and delete column in Python.
5. Make sure output is saved to new dataframe, `schedule_name`.

Some notes about how and why we use these specific steps. Whether we merged by the away or home order is not important, and we arbitrarily selected order. We needed to rename columns to avoid duplicate names later and also to keep column names clear. The importance of this will become evident when you have to clean up your own mess or somebody else's messy code! Lastly, we encourage you to start with one line of code and keep adding more code until you understand the big picture. That's how we constructed this example.

With the R example, use piping to avoid re-writing objects like you did for the Python example. First, take the `schedule` dataframe and then left join to the `city_data`. Tell R to join by (or match) the `home` column to the `city` column. Then rename the `team` column to be the `home_team` column. This helps us keep the team columns straight in the final dataframe. Then repeat these steps and join the away team data:

```
## R
schedule_name <-
  schedule |>
  left_join(city_data, by = c("home" = "city")) |>
  rename(home_team = team) |>
  left_join(city_data, by = c("away" = "city")) |>
  rename(away_team = team)
print(schedule_name)
  home away home_team away_team
1  GB   DET    Packers     Lions
2  DET   HOU      Lions     Texans
```

With Python, create temporary objects rather than piping. This is because pandas's piping is not as intuitive to us and requires writing custom functions, something beyond the scope of this book. Furthermore, some people like writing out code to see all of the steps and we want to show you a second method for this example.

In Python, first do a left merge. Tell Python we use `home` for the left merge on and `city` for the right merge on. Then rename the `team` column to be `home_team`. The Pandas' `rename()` function requires a dictionary as a input. Then, tell Pandas to remove (or `.drop()`) the city column to avoid confusion later. Then repeat these steps for the away team:

```
## Python
step_1 = schedule.merge(city_data, how = "left",
                       left_on = "home", right_on = "city")
step_2 = step_1.rename(columns =
                      {"team": "home_team"}).drop(columns = "city")
step_3 = step_2.merge(city_data, how = "left",
                      left_on = "away", right_on = "city")
schedule_name = step_3.rename(columns =
                               {"team": "home_team"}).drop(columns =
"city")
print(schedule_name)
   home  away  home_team  home_team
0    GB   DET     Packers      Lions
1    DET   HOU     Lions      Texans
```

Glossary

Air yards

The distance traveled by the pass from the line of scrimmage to the intended receiver, whether or not the pass was complete.

Average depth of target (aDOT)

The average air yards traveled on targeted passes for quarterbacks and targets for receivers.

Bins

The discrete categories of numbers used to summarize data in a histogram.

Book

Short for sportsbook in football gambling. This the person, group, casino, or other similar enterprise that takes wagers on sporting (and other) events.

Bounded

When a number of “bounded”, its value is constrained by some other values. For example, a percentage is bounded by 0 and 100%.

Boxplots

A type of box that uses a “box” to show the middle 50% of data and stems for the upper and lower quartiles. Commonly, outliers are plotted as dots. The plots are also known as “box-and-whisker” plots.

Binomial

Refers to a type of statistical distribution with a “binary” (0/1-type) response. Examples might include wins/losses for a game or sacks/no-sacks from a play.

Cheeseheads

Fans of the Green Bay Packers. For example, Richard is a Cheesehead because he likes the Packers. Eric is not a fan of the Packers and is therefore not a Cheesehead.

Closing line

The final price offered by a sportsbook before a game starts. In theory, this contains all of the opinions, expressed through wagers, of all bettors who have enough influence to move the line into place.

Clustering

A type of statistical methods for dividing data points into similar groups (“clusters”) based on a set of features.

Coefficient

The predictor estimates from a regression-type analysis. Slopes and intercepts are special cases of coefficients.

Completion percentage over expected (CPOE)

The rate at which a quarterback has successful (“completed”) passes compared to what would predicted (“expected”) given a situation based upon an expected completion percentage model.

Context

What is going on around a situation such the factors going on for a play such as the down, yards to go, and field position.

Controlled for

Including one or more extra variables in a regression or regression-like model. For example, pass completion might be controlled for yards to go. See also “corrected for” and “normalized”.

Corrected for

A synonym for “controlled for”.

Confidence interval (CI)

A measure of uncertainty around a point estimate such as a mean or regression coefficient. For example, a 95% CI around a mean will contain the true mean 95% of the time if you repeat the observation process many, many, many times. But, you will never know which 5% of times you are wrong.

Data dictionary

Data about data. Also, this can often be thought of as a synonym for metadata.

Data pipeline

The concept that data may need to flow from one location to another, and, undergo changes such as formatting along the way. Also, see “pipe”

Data wrangling

The process of getting data into the format you need to solve your problems. Other terms include data cleaning, data formatting, data tiding, data transformation, data manipulation, data munging, and data mutating.

Degrees of freedom

The “extra” number of data points left over from fitting a model.

Down

The finite number of plays a play has to go a certain distance (measured in yards) and either score or obtain a new set of plays before they lose possession of the ball.

Dimensions (of data)

Data dimensions are how many variables are needed to describe data. Graphically, this is number of axes needed to describe the data. Tabularly, this is the number of columns needed to describe the data. Algebraically, this

is the number of independent variables needed to describe the data.

Dimensionality reduction

A statistical approach for reducing the number of features by creating new, independent features. Principal component analysis (PCA) is an example of one type of dimensionality reduction.

Distance

The number of yards remaining to either obtain a new first down or score a touchdown.

Draft capital

The term covers the resources a team uses during the NFL draft including the number of picks, pick rounds, and pick numbers.

Draft approximate value (DrAV)

The approximate value generated by a player picked for his drafting team. This is a metric developed by Pro Football Reference.

Edge

An advantage over the betting markets for predicting outcomes, usually expressed as a percentage.

Expected point

The estimated, or *expected* value for how many *points* one would expect a team to score given the current game situation on that drive.

Expected points added (EPA)

The difference between a team's expected points from one play to the next, measuring the success of the play.

Exploratory data analysis (EDA)

A subset of statistical analysis that analyzes data by describing or summarizing their main characteristics. Usually this involves both graphical

summaries such as plots and numerical summaries such means and standard deviations.

Feature

A predictor variable in a model. The term is used more commonly by data scientists whereas statisticians tend to use terms like “predictor” or “dependent” variable.

Fixed-effect model

A linear regression or similar model without random-effects. See random-effect model.

for loop

A computer programming tool that repeats (or “loops”) over a function “for” a pre-defined number of iterations.

Generalized linear models (GLM)

An extension of linear models (such as simple linear regression and multiple regression) to include a link function and non-normal response variable such as logistic regression with binary data or Poisson regression with count data.

Gridiron football

A synonym for American football.

Group by

A concept from SQL-type languages that describes taking data and creating sub-groups (or “grouping”) based upon (or “by”) a variable. For example, you might take Aaron Rodgers passing yards and “group by” season to calculate his average passing yards per season.

Handle

The total amount of money placed by bettors across all markets.

High-leverage situations

Important plays that determine the outcome of games. For example, converting the ball on third downs or fourth and goal. These plays, while of great importance, are generally not predictive game-to-game or season-to-season.

Histogram

A type of plot that summarizes counts of data into discrete bins.

Hit

Two uses in this book. A football colliding with another is a “hit”. Additionally, computer script trying to download from a webpage “hits” the page when trying to download.

Interaction

In a regression-type model, sometimes two predictors or features change together. A “interaction” between these two terms allows this to be included within the model.

Intercept

Where a simple linear regression crosses through zero. Also, sometimes used to refer to multiple regression coefficients with a discrete predictor variable.

Internal

For sports bettors, the price they would offer the game at if they were a sportsbook. The discrepancy between this value and the actual price of the game determines the edge.

Interquartile range

The difference between the first and third quartile. See quartile.

Link function

The function that maps between the observed scale and model scale in a generalized linear model. For example, a logit function can link between the observed probability of an outcome occurring on the bounded 0-1 probability

scale to the log-odds scale that ranges from negative to positive infinity.

Log odds

Odds on the log scale.

Long pass

A pass typically longer than 20 yards, although the actual threshold may vary.

Metadata

The data describing the data. For example, is a “time column in minutes:seconds or decimal minutes.” Also, this can often be thought of as a synonym for Data dictionary.

Mixed-effect models

A model with both fixed-effects and random-effects. See random-effect model. Synonyms include hierarchical models, multi-level models, and repeated-measure or repeated-observation models.

Moneyline

A bet in American football where you bet on a team winning straight up.

Multiple regression

A type of regression with more than 1 predictor variable. Simple linear regression is a special type of multiple regression.

North American football

A synonym for American football.

Normalize

This term has multiple definitions. In the book, we use it to refer to accounting for other variables in regression analysis. Also see “correct for” or “control for” Another definition you may see is that a variable may be *normalized* and then be transformed to have a mean of 0 and standard

deviation of 1, thereby following a normal distribution.

Odds

In betting and logistic regression, odds refers to the number of times an event occurs to the number of time the event does not occur. For example, if Kansas City has a 4-to-1 odds of winning this week, people would expect them to win one game for every four games they lose under a similar situation. Odds can either emerge empirically through events occur and models estimating the odds, or, through betting where odds emerge through the “wisdom” of the crows.

Odds-ratio

Odds in ratio format. For example, 3-to-2 odds can be written as 3:2 or 1.5 as odds ratios.

Open source

A term used to describe software where code must freely accessible (that is to say, anybody can look at the code) and freely available (that is to say, not cost money).

Origination

The process by which an oddsmaker or a bettor sets the line.

Outliers

Data points that are far away from other data point.

Overfit

A model is said to be “overfit” when either too many parameters have been estimated compared to the amount of data, or, the model fits one situation too well and does not apply to other situations.

P-values

The probability of obtaining the observed test statistic assuming the null hypothesis of no difference is true. These values are increasingly falling out

of favor with professional statisticians due to their common misuse.

Pearson's correlation coefficient

A value between -1 to 1. A value of 1 means two groups are perfectly positively correlated, and, as one increases, the other increases. A value of -1 means two groups are perfectly negatively correlated, and, as one increases, the other decreases. A value of zero means no correlation and the values for one group do not have any relation to the values from another group.

Pipe

Passing the outputs from one function directly to another function. See “data pipeline.””

Play-by-play (data)

Play-by-play data is the recorded results for each play of a football. Often this data is “row poor”, in that there are far more features (columns) than plays (rows).

Principal component analysis (PCA)

A statistical tool for creating fewer, independent features from a set of features.

Probability

A number between zero and one that describes the chance of an event occurring. Different definitions of the understanding of probability exist include frequentist definition, which is the long-term average under similar conditions- such as flipping a coin; and Bayesian, which is the belief in an event occurring - such as betting markets.

Probability distributions

A mathematically function that assigns a value (specifically a probability) between zero and one to an event occurring.

Proposition (bets)

A type of bet focusing on a specific outcome occurring such as who will score the first touchdown. Also called “prop” for short.

Push

A game where the outcome lands on the spread and the better is refunded their money.

Pythonistas

People who use the Python language.

Quartile

A quarter of the data based upon numerical ranking. There are four quartiles of data.

Random-effect model

A model with coefficients that are assumed to come from a shared distribution.

Regression

A type of statistical model that describes the relationship between one response variable (a simple linear regression) and one or more predictor variables (a multiple regression). Also a special type of linear model.

Regression candidate

With regression, observations are expected to “regress to the mean (average)” value through time. For example, a player who has a good year this year would be reasonably expected to have a year closer to average next year, especially if the source of their good year is a relatively unstable, or “noisy”, statistic.

Relative risk

A method for understanding the results from a Poisson regression, similar to the outputs from a logistic regression with odds-ratios.

Residual

The difference between a models' predicted value for an observation and the value for an observation.

Run yards over average (RYOE)

The number of running yards a player obtains compared the value expected (or average) value from a model given the play's situation.

Sabermetrics

Quantitative analysis of baseball, named after the Society for American Baseball Research (SABER).

Scrape

As in webscraping, or using computer programs to download data from websites.

Scatterplot

A type of plot that plots points on both axes.

Set the line

The process of the oddsmaker(s) creating the odds.

Simpson's Paradox

A statistical phenomena where relationships between variables change based upon different groupings using other variables.

Slope

The change in a trend through time and often used to describe regression coefficients with continuous predictor variables.

Spread (bet)

A betting market in American football that is the most popular and easy to understand. The spread is the point value that is meant to split outcomes in

half over a large sample of games. This doesn't necessarily mean the sportsbook wants half of the bets on either side of the spread, though.

Stability

Within the context of this book, stability of a evaluation metric as the metric's ability to predict itself over a predetermined time frame. Also, see “stability analysis” and “sticky stats”.

Stability analysis

The measurement of how well a metric or model outputs holds up through time. For example, with Football, one would care about the stability of making predictions across seasons.

Sticky stats

A term commonly used in fantasy football for numbers that are stable through time.

Short-yardage back

A running back who tends to play when only a few (or “short”) number of yards are required to obtain a first down or a touchdown.

Short pass

A pass typically less than 20 yards, although the actual threshold may vary (e.g. the first-down marker).

Standard error

A measure of the uncertainty around a distribution given the uncertainty and sample size.

Standard deviation

A measure of the spread or dispersion in a distribution.

Supervised learning

A type of statistical and machine learning algorithms where people know the

groups ahead of time and the algorithm may be trained on data.

Total (bet)

A simply bet on whether or not the sum of the two team's points goes over or under a specified amount.

Total (for a game) (bet)

The number of points expected by the betting market for game.

Three true outcomes

Baseball's first, and arguably most important outcomes that can be modeled across area walks, strikeouts, and home runs. These outcomes also do not depend upon the defense other than rare exceptions.

Unsupervised learning

A type of statistical and machine learning algorithms where people do not know the groups ahead of time.

UseR

People who use the R language.

Variable

Depending upon context two definitions are used in the book. First, observations can be variable. For example, pass yards might be highly variable for a quarterback meaning the quarterback lacks consistency. Second, a model can a variable. For example, air yards might be a predictor variable for the response variable completion in a regression model.

Vig

See Vigorish.

Vigorish

The “house” (casino, bookie, or other similar institution that takes bets) advantage during that ensures the house almost always makes money over

the long-term.

Win probability (WP)

A model to predict the probability a team wins the game a given point during the game.

Wins above replacement

A framework for estimating the number of wins a player is worth during the course of a season, set of seasons, or a career. First created in baseball.

Yards-per-attempt (YPA)

Also known as yards per passing attempt, YPA is the average number of yards a quarterback throws for during a defined time period such as game or season.

Yards-per-carry (YPC)

YPC refers to the average number of yards a player runs the ball during a defined time period such as game or season.

Yards-to-go

The number of yards necessary to either obtain a first down or score during a play.

About the Authors

Eric A Eager is the Vice President and Partner at SumerSports, a football analytics startup founded by billionaire hedge fund manager Paul Tudor Jones and his son, Jack Jones. Eric currently hosts the “SumerSports Show” with former Falcons General Manager and two-time NFL Executive of the Year Thomas Dimitroff. Prior to joining Sumer, he founded the industry-leading analytics group at Pro Football Focus (PFF), which is owned by former Bengals wide receiver and current Sunday Night Football color commentator, Cris Collinsworth.

During his career, Eric has built tools used by all 32 NFL teams and over 100 college football teams, and various media entities. His simulation model has been used by NBC’s Steve Kornacki during his “Road to the Playoffs” segment on NBC since 2020, and he’s also been a part of the Fox NFL Game of the Week broadcast with Joe Buck and Troy Aikman. His former podcast, the PFF Forecast, was the most popular football analytics podcast in the world when he left the show.

Eric studied applied mathematics and mathematical biology at the University of Nebraska – Lincoln, where he wrote his PhD thesis on how stochasticity and nonlinear processes affect population dynamics. For the first six years of his career he was a professor at the University of Wisconsin La Crosse, where he published more than 25 peer-reviewed research papers on the interface of mathematics, biology, and the scholarship of teaching and learning. Six of those papers were published with Richard Erickson, his co-author for this book. After leaving academia for football full-time in 2018, he’s maintained a connection to teaching by building “Linear Algebra for Data Science in R” for DataCamp in 2018, as well as teaching Wharton’s Moneyball Academy course to high school students each summer since 2020.

Eric maintains a strong interest in mentorship, as he wants up-and-coming football analysts to have a more straightforward path to a career than he had. He enjoys reading, writing, biking, rowing, and watching the WNBA with his family. He lives in Atlanta, GA with his wife, Stephanie, and daughters Madeline and Chloe.

Richard A Erickson helps people use mathematics and statistics to understand our world as well as make decisions with this data. He is a lifelong Green Bay

Packer fan, and, like thousands of other cheeseheads, a team owner. He has taught over 32,000 students statistics through graduate-level courses, workshops, and his DataCamp courses on Generalized Linear Models in R and Hierarchical Models in R. He also uses Python on a regular basis to model scientific problems.

Richard received his PhD in Environmental Toxicology with an applied math minor from Texas Tech where he wrote his dissertation on modeling population-level effects of pesticides. He has modeled and analyzed diverse datasets including topics such as soil productivity for the US Department of Agriculture, impacts of climate change on disease dynamics, and improving rural healthcare. Richard currently works as a research scientist and has over 80 peer-reviewed publications. Besides teaching Eric about R and Python, Richard also taught Eric to like cheese curds.

Richard lives in La Crosse, WI with his daughter, Margo, and Bernese mountain dog, Sadie. When not cheering for his Packers, he likes silent sports, notably biking, cross-country skiing, sea kayaking, and scuba diving.