Sports Data Analysis and Visualization

Code, data and visuals for storytellers

Matt Waite & Derek Willis

7/19/2022

Table of contents

1 Throwing cold water on hot takes **6**
1.1 Requirements and Conventions . . . 6
1.2 About this book . . . 7

2 The very basics **8**
2.1 Adding libraries, part 1 . . . 9
2.2 Adding libraries, part 2 . . . 10
2.3 Notebooks . . . 10

3 Data, structures and types **12**
3.1 Rows and columns . . . 12
3.2 Types . . . 13
3.3 A simple way to get data . . . 14
3.4 Cleaning the data . . . 14

4 Aggregates **17**
4.1 Basic data analysis: Group By and Count . . . 18
4.2 Other aggregates: Mean and median . . . 22
4.3 Even more aggregates . . . 24

5 Mutating data **26**
5.1 A more complex example . . . 29

6 Filters and selections **32**
6.1 Selecting data to make it easier to read . . . 35
6.2 Using conditional filters to set limits . . . 35
6.3 Top list . . . 38

7 Transforming data **39**
7.1 Making long data wide . . . 41
7.2 Why this matters . . . 42

8 Significance tests **44**
8.1 Accepting the null hypothesis . . . 46

9 Correlations and regression **49**
9.1 A more predictive example . . . 53

10 Multiple regression **56**

11 Residuals **76**
11.1 Penalties . . . 80

12 Z-scores **84**
12.1 Calculating a Z score in R . . . 85
12.2 Writing about z-scores . . . 89

13 Clustering **90**
13.1 Advanced metrics . . . 97

14 Simulations **102**
14.1 Cold streaks . . . 103

15 Intro to ggplot with bar charts **105**
15.1 The bar chart . . . 107
15.2 Scales . . . 110
15.3 Styling . . . 112
15.4 One last trick: coord flip . . . 114

16 Stacked bar charts **116**

17 Circular bar plots **121**
17.1 Does November basketball matter? . . . 122
17.2 Does it show you where you are? . . . 125

18 Waffle charts **128**
18.1 Making waffles with vectors . . . 128

19 Step charts **134**

20 Dumbbell and lollipop charts **138**
20.1 Dumbbell plots . . . 138
20.2 Lollipop charts . . . 143

21 Scatterplots **147**
21.1 Let's see it work . . . 152

22 Bubble charts **155**

23 Beeswarm plots **165**
23.1 A few other options . . . 169

24 Bump charts **172**

25 Tables **182**

26 Facet wraps **194**
26.1 Facet grid vs facet wraps . . . 197
26.2 Other types . . . 199

27 Arranging multiple plots together **201**

28 Encircling points on a scatterplot **207**
28.1 A different, more local example . . . 213

29 Text cleaning **215**
29.1 Stripping out text . . . 215
29.2 Another example: splitting columns . . . 217

30 Headlines **222**

31 Annotations **230**

32 Finishing touches **233**
32.1 Graphics vs visual stories . . . 233
32.2 Getting ggplot closer to output . . . 234
32.3 Waffle charts require special attention . . . 242
32.4 Advanced text wrangling . . . 246

33 Intro to rvest **252**
33.1 A slightly more complicated example . . . 253
33.2 An even more complicated example . . . 255

34 Advanced rvest **258**
34.1 One last bit . . . 262

35 Building your own blog with blogdown **263**
35.1 Setup . . . 263
35.2 Seeing your site . . . 266
35.3 Editing existing Markdown content . . . 267
35.4 Creating a new post in R Markdown . . . 269
35.5 Publishing your site . . . 271

36 Project Checklist **275**
36.1 Writing checklist 275
36.2 Headline checklist 275
36.3 Graphics checklist 275
36.4 Code checklist 276
36.5 Last thing 276

37 Using packages to get data **277**
37.1 Using cfbfastR as a cautionary tale 277
37.2 Another example 280

1 Throwing cold water on hot takes

Why do teams struggle? There are lots of potential reasons: injuries, athletes in the wrong position, poor execution. Or it could be external factors: well-prepared opponents, the weather, the altitude or, of course, the refs.

You could turn the question around: why do teams succeed? Again, there are plenty of possibilities that get tossed around on talk radio, on the sports pages and across social media. A lot of hot takes.

The more fundamental question that this course will empower you to answer is this: what do teams and athletes *do*? Using data, you'll learn to ask questions and visualize the answers, ranging across sports and scenarios. What did the 2021-22 Maryland men's lacrosse team do well en route to the national championship? How has the transfer portal (and additional eligibility) changed the nature of programs? In football, do penalties have any relationship on scoring?

To get into these and other questions, we'll use a lot of different tools and techniques, but this class rests on three pillars:

1. Simple, easy to understand statistics …
2. … extracted using simple code …
3. … visualized simply to reveal new and interesting things in sports.

Do you need to be a math whiz to read this book? No. I'm not one either. What we're going to look at is pretty basic, but that's also why it's so powerful.

Do you need to be a computer science major to write code? Nope. I'm not one of those either. But anyone can think logically, and write simple code that is repeatable and replicable.

Do you need to be an artist to create compelling visuals? I think you see where this is going. No. I can barely draw stick figures, but I've been paid to make graphics in my career. With a little graphic design know how, you can create publication worthy graphics with code.

1.1 Requirements and Conventions

This book is all in the R statistical language. To follow along, you'll do the following:

1. Install the R language on your computer. Go to the R Project website, click download R and select a mirror closest to your location. Then download the version for your computer.
2. Install R Studio Desktop. The free version is great.

Going forward, you'll see passages like this:

```
install.packages("tidyverse")
```

Don't do it now, but that is code that you'll need to run in your R Studio. When you see that, you'll know what to do.

1.2 About this book

This book is the collection of class materials for the Fall 2022 JOUR479X course in the Philip Merrill College of Journalism at the University of Maryland. There's some things you should know about it:

- It is free for students.
- The topics will remain the same but the text is going to be constantly tinkered with.
- What is the work of the author is copyright Derek Willis 2022 & Matt Waite 2019.
- The text is Attribution-NonCommercial-ShareAlike 4.0 International Creative Commons licensed. That means you can share it and change it, but only if you share your changes with the same license and it cannot be used for commercial purposes. I'm not making money on this so you can't either.
- As such, the whole book – authored in Quarto – is open sourced on Github. Pull requests welcomed!

2 The very basics

R is a programming language, one specifically geared toward statistical analysis. Like all programming languages, it has certain built-in functions and you can interact with it in multiple ways. The first, and most basic, is the console.

Think of the console like talking directly to R. It's direct, but it has some drawbacks and some quirks we'll get into later. For now, try typing this into the console and hit enter:

```
2+2
```

```
[1] 4
```

Congrats, you've run some code. It's not very complex, and you knew the answer before hand, but you get the idea. We can compute things. We can also store things. **In programming languages, these are called variables**. We can assign things to variables using `<-`. And then we can do things with them. **The `<-` is a called an assignment operator**.

```
number <- 2

number * number
```

```
[1] 4
```

Now assign a different number to the variable number. Try running `number * number` again. Get what you expected?

We can have as many variables as we can name. **We can even reuse them (but be careful you know you're doing that or you'll introduce errors)**. Try this in your console.

```
firstnumber <- 1
secondnumber <- 2

(firstnumber + secondnumber) * secondnumber
```

```
[1] 6
```

We can store anything in a variable. A whole table. An array of numbers. Every college basketball game played in the last 10 years. A single word. A whole book. All the books of the 18th century. They're really powerful. We'll explore them at length.

2.1 Adding libraries, part 1

The real strength of any given programming language is the external libraries that power it. The base language can do a lot, but it's the external libraries that solve many specific problems – even making the base language easier to use.

For this class, we're going to need several external libraries.

The first library we're going to use is called Swirl. So in the console, type `install.packages('swirl')` and hit enter. That installs swirl.

Now, to use the library, type `library(swirl)` and hit enter. That loads swirl. Then type `swirl()` and hit enter. Now you're running swirl. Follow the directions on the screen. When you are asked, you want to install course 1 R Programming: The basics of programming in R. Then, when asked, you want to do option 1, R Programming, in that course.

When you are finished with the course – it will take just a few minutes – it will first ask you if you want credit on Coursera. You do not. Then type 0 to exit (it will not be very clear that's what you do when you are done).

2.2 Adding libraries, part 2

We'll mostly use two libraries for analysis – `dplyr` and `ggplot2`. To get them, and several other useful libraries, we can install a single collection of libraries called the tidyverse. Type this into your console: `install.packages('tidyverse')`

NOTE: This is a pattern. You should always install libraries in the console.

Then, to help us with learning and replication, we're going to use R Notebooks. So we need to install that library. Type this into your console: `install.packages('rmarkdown')`

2.3 Notebooks

For the rest of the class, we're going to be working in notebooks. In notebooks, you will both run your code and explain each step, much as I am doing here.

To start a notebook, you click on the green plus in the top left corner and go down to R Notebook. Do that now.

You will see that the notebook adds a lot of text for you. It tells you how to work in notebooks – and you should read it. The most important parts are these:

To add text, simply type. To add code you can click on the *Insert* button on the toolbar or by pressing *Cmd+Option+I* on Mac or *Ctl+Alt+I* on Windows.

Highlight all that text and delete it. You should have a blank document. This document is called a R Markdown file – it's a special form of text, one that you can style, and one you can include R in the middle of it. Markdown is a simple markup format that you can use to create documents. So first things first, let's give our notebook a big headline. Add this:

```
# My awesome notebook
```

Now, under that, without any markup, just type This is my awesome notebook.

Under that, you can make text bold by writing `It is **really** awesome`.

If you want it italics, just do this on the next line: `No, it's _really_ awesome. I swear.`

To see what it looks like without the markup, click the Preview or Knit button in the toolbar. That will turn your notebook into a webpage, with the formatting included.

Throughout this book, we're going to use this markdown to explain what we are doing and, more importantly, why we are doing it. Explaining your thinking is a vital part of understanding what you are doing.

That explaination, plus the code, is the real power of notebooks. To add a block of code, follow the instructions from above: click on the *Insert* button on the toolbar or by pressing *Cmd+Option+I* on Mac or *Ctl+Alt+I* on Windows.

In that window, use some of the code from above and add two numbers together. To see it run, click the green triangle on the right. That runs the chunk. You should see the answer to your addition problem.

And that, just that, is the foundation you need to start this book.

3 Data, structures and types

Data are everywhere (and data is plural of datum, thus the use of are in that statement). It surrounds you. Every time you use your phone, you are creating data. Lots of it. Your online life. Any time you buy something. It's everywhere. Sports, like life, is no different. Sports is drowning in data, and more comes along all the time.

In sports, and in this class, we'll be dealing largely with two kinds of data: event level data and summary data. It's not hard to envision event level data in sports. A pitch in baseball. A hit. A play in football. A pass in soccer. They are the events that make up the game. Combine them together – summarize them – and you'll have some notion of how the game went. What we usually see is summary data – who wants to scroll through 50 pitches to find out a player went 2-3 with a double and an RBI? Who wants to scroll through hundreds of pitches to figure out the Rays beat the Yankees?

To start with, we need to understand the shape of data.

> EXERCISE: Try scoring a child's board game. For example, Chutes and Ladders. If you were placed in charge of analytics for the World Series of Chutes and Ladders, what is your event level data? What summary data do you keep? If you've got the game, try it.

3.1 Rows and columns

Data, oversimplifying it a bit, is information organized. Generally speaking, it's organized into rows and columns. Rows, generally, are individual elements. A team. A player. A game. Columns, generally, are components of the data, sometimes called variables. So if each row is a player, the first column might be their name. The second is their position. The third is their batting average. And so on.

G	Date		Opp	W/L		Column			FG%	3P	3PA	3P%
1	2018-11-06		Mississippi Valley State	W	106	37	38	69	0.551	15	37	0.405
2	2018-11-11		Southeastern Louisiana	W	87	35	31	55	0.564	8	25	0.32
3	2018-11-14		Seton Hall	W	80	57	26	63	0.413	8	23	0.348
4	2018-11-19	N	Missouri State	W	85	62	31	61	0.508	13	33	0.394
5	2018-11-20	N	Texas Tech	L	52	70	17	48	0.354	5	23	0.217
6	2018-11-24		Western Illinois	W	73	49	29	63	0.46	4	20	0.2
7	2018-11-26	@	Clemson	W	68	66	26	55	0.473	7	22	0.318
8	2018-12-02		Rows	W	75	60	22	46	0.478	6	14	0.429
9	2018-12-05	@	Minnesota	L	78	85	28	61	0.459	7	21	0.333
10	2018-12-08		Creighton	W	94	75	32	60	0.533	14	27	0.519
11	2018-12-16	N	Oklahoma State	W	79	56	25	53	0.472	8	18	0.444
12	2018-12-22		Cal State Fullerton	W	86	62	26	59	0.441	10	22	0.455
13	2018-12-29		Southwest Minnesota State	W	79	38	29	66	0.439	11	24	0.458
14	2019-01-02	@	Maryland	L	72	74	24	51	0.471	9	22	0.409
15	2019-01-06	@	Iowa	L	84	93	30	65	0.462	4	23	0.174
16	2019-01-10		Penn State	W	70	64	27	55	0.491	9	23	0.391
17	2019-01-14	@	Indiana	W	66	51	27	60	0.45	8	24	0.333
18	2019-01-17		Michigan State	L	64	70	22	67	0.328	5	26	0.192
19	2019-01-21	@	Rutgers	L	69	76	25	60	0.417	9	22	0.409

One of the critical components of data analysis, especially for beginners, is having a mental picture of your data. What does each row mean? What does each column in each row signify? How many rows do you have? How many columns?

3.2 Types

There are scores of data types in the world, and R has them. In this class, we're primarily going to be dealing with data frames, and each element of our data frames will have a data type.

Typically, they'll be one of four types of data:

- Numeric: a number, like the number of touchdown passes in a season or a batting average.
- Character: Text, like a name, a team, a conference.
- Date: Fully formed dates – 2019-01-01 – have a special date type. Elements of a date, like a year (ex. 2019) are not technically dates, so they'll appear as numeric data types.
- Logical: Rare, but every now and then we'll have a data type that's Yes or No, True or False, etc.

Question: Is a zip code a number? Is a jersey number a number? Trick question, because the answer is no. Numbers are things we do math on. If the thing you want is not something you're going to do math on – can you add two phone numbers together? – then make it a

character type. If you don't, most every software system on the planet will drop leading zeros. For example, every zip code in Boston starts with 0. If you record that as a number, your zip code will become a four digit number, which isn't a zip code anymore.

3.3 A simple way to get data

One good thing about sports is that there's lots of interest in it. And that means there's outlets that put sports data on the internet. Now I'm going to show you a trick to getting it easily.

The site sports-reference.com takes NCAA (and other league) stats and puts them online. For instance, here's their page on Maryland's men's basketball's game logs, which you should open now.

Now, in a new tab, log into Google Docs/Drive and open a new spreadsheet. In the first cell of the first row, copy and paste this formula in:

```
=IMPORTHTML("https://www.sports-reference.com/cbb/schools/maryland/2022-gamelogs.html", "tabl
```

If it worked right, you've got the data from that page in a spreadsheet.

3.4 Cleaning the data

The first thing we need to do is recognize that we don't have data, really. We have the results of a formula. You can tell by putting your cursor on that field, where you'll see the formula again. This is where you'd look:

	A	B	C	D	E	F	G	H	I
1								School	
2	G	Date		Opp	W/L	Tm	Opp	FG	FGA
3	1	2021-11-09		Quinnipiac	W	83	69	31	63
4	2	2021-11-11		George Washingto	W	71	64	24	64
5	3	2021-11-13		Vermont	W	68	57	22	60
6	4	2021-11-17		George Mason	L	66	71	25	55
7	5	2021-11-19		Hofstra	W	69	67	25	60
8	6	2021-11-25	N	Richmond	W	86	80	28	57
9	7	2021-11-27	N	Louisville	L	55	63	20	52
10	8	2021-12-01		Virginia Tech	L	58	62	23	55
11	9	2021-12-05		Northwestern	L	61	67	17	59
12	10	2021-12-12	N	Florida	W	70	68	24	49
13	11	2021-12-28		Lehigh	W	76	55	24	57
14	12	2021-12-30		Brown	W	81	67	28	57

The solution is easy:

Edit > Select All or type command/control A Edit > Copy or type command/control c Edit > Paste Special > Values Only or type command/control shift v

You can verify that it worked by looking in that same row 1 column A, where you'll see the formula is gone.

									School	
G			Opp	W/L	Tm		Opp		FG	FGA
			Quinnipiac	W		83		69	31	63
				W		71		64	24	64
				W		68		57	22	60
				L		66		71	25	55
				W		69		67	25	60
				W		86		80	28	57
7	44527	N		L		55		63	20	52
8	44531			L		58		62	23	55
9	44535			L		61		67	17	59
10	44542	N		W		70		68	24	49
11	44558		Lehigh	W		76		55	24	57
12	44560		Brown	W		81		67	28	57
13	44564	@	Iowa	L		75		80	28	61
14	44567	@	Illinois	L		64		76	23	50
15	44570		Wisconsin	L		69		70	25	55
16	44573	@	Northwestern	W (2 OT)		94		87	30	71

Now you have data, but your headers are all wrong. You want your headers to be one line – not two, like they have. And the header names repeat – first for our team, then for theirs. So you have to change each header name to be UsORB or TeamORB and OpponentORB instead of just ORB.

After you've done that, note we have repeating headers. There's two ways to deal with that – you could just hightlight it and go up to Edit > Delete Rows XX-XX depending on what rows you highlighted. That's the easy way with our data.

But what if you had hundreds of repeating headers like that? Deleting them would take a long time.

You can use sorting to get rid of anything that's not data. So click on Data > Sort Range. You'll want to check the "Data has header row" field. Then hit Sort.

Now all you need to do is search through the data for where your junk data – extra headers, blanks, etc. – got sorted and delete it. After you've done that, you can export it for use in R. Go to File > Download as > Comma Separated Values. Remember to put it in the same directory as your R Notebook file so you can import the data easily.

4 Aggregates

R is a statistical programming language that is purpose built for data analysis.

Base R does a lot, but there are a mountain of external libraries that do things to make R better/easier/more fully featured. We already installed the tidyverse – or you should have if you followed the instructions for the last assignment – which isn't exactly a library, but a collection of libraries. Together, they make up the tidyverse. Individually, they are extraordinarily useful for what they do. We can load them all at once using the tidyverse name, or we can load them individually. Let's start with individually.

The two libraries we are going to need for this assignment are `readr` and `dplyr`. The library `readr` reads different types of data in as a dataframe. For this assignment, we're going to read in csv data or Comma Separated Values data. That's data that has a comma between each column of data.

Then we're going to use `dplyr` to analyze it.

To use a library, you need to import it. Good practice – one I'm going to insist on – is that you put all your library steps at the top of your notebook.

That code looks like this:

```
library(readr)
```

To load them both, you need to run that code twice:

```
library(readr)
library(dplyr)
```

You can keep doing that for as many libraries as you need. I've seen notebooks with 10 or more library imports.

But the tidyverse has a neat little trick. We can load most of the libraries we'll need for the whole semester with one line:

```
library(tidyverse)
```

From now on, if that's not the first line of your notebook, you're probably doing it wrong.

4.1 Basic data analysis: Group By and Count

The first thing we need to do is get some data to work with. We do that by reading it in. In our case, we're going to read data from a csv file – a comma-separated values file.

The CSV file we're going to read from is a Basketball Reference page of advanced metrics for NBA players this past season. The Sports Reference sites are a godsend of data, a trove of stuff, and we're going to use it a lot in this class.

So step 2, after setting up our libraries, is most often going to be importing data. In order to analyze data, we need data, so it stands to reason that this would be something we'd do very early.

The code looks *something* like this, but hold off copying it just yet:

```
nbaplayers <- read_csv("~/SportsData/nbaadvancedplayers2122.csv")
```

Let's unpack that.

The first part – nbaplayers – is the name of your variable. A variable is just a name of a thing that stores stuff. In this case, our variable is a data frame, which is R's way of storing data (technically it's a tibble, which is the tidyverse way of storing data, but the differences aren't important and people use them interchangeably). **We can call this whatever we want.** I always want to name data frames after what is in it. In this case, we're going to import a dataset of NBA players. Variable names, by convention are one word all lower case. You can end a variable with a number, but you can't start one with a number.

The <- bit is the variable assignment operator. It's how we know we're assigning something to a word. Think of the arrow as saying "Take everything on the right of this arrow and stuff it into the thing on the left." So we're creating an empty vessel called `nbaplayers` and stuffing all this data into it.

The `read_csv` bits are pretty obvious, except for one thing. What happens in the quote marks is the path to the data. In there, I have to tell R where it will find the data. The easiest thing to do, if you are confused about how to find your data, is to put your data in the same folder as as your notebook (you'll have to save that notebook first). If you do that, then you just need to put the name of the file in there (nbaadvancedplayers2122.csv). In my case, in my home directory (that's the ~ part), there is a folder called SportsData that has the file called nbaadvancedplayers2122.csv in it. Some people – insane people – leave the data in their downloads folder. The data path then would be `~/Downloads/nameofthedatafilehere.csv` on PC or Mac.

What you put in there will be different from mine. So your first task is to import the data.

```
nbaplayers <- read_csv("data/nbaadvancedplayers2122.csv")
```

```
New names:
Rows: 812 Columns: 29
-- Column specification
-------------------------------------------------------- Delimiter: "," chr
(3): Player, Pos, Tm dbl (24): Rk, Age, G, MP, PER, TS%, 3PAr, FTr, ORB%, DRB%,
TRB%, AST%, STL%,... lgl (2): ...20, ...25
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.
* `` -> `...20`
* `` -> `...25`
```

Now we can inspect the data we imported. What does it look like? To do that, we use `head(nbaplayers)` to show the headers and **the first six rows of data**. If we wanted to see them all, we could just simply enter `nbaplayers` and run it.

To get the number of records in our dataset, we run `nrow(nbaplayers)`

```r
head(nbaplayers)
```

```
# A tibble: 6 x 29
     Rk Player     Pos     Age Tm        G    MP   PER `TS%` `3PAr`   FTr `ORB%`
  <dbl> <chr>      <chr> <dbl> <chr> <dbl> <dbl> <dbl> <dbl>  <dbl> <dbl>  <dbl>
1     1 Precious ~ C        22 TOR      73  1725  12.7 0.503  0.259 0.217    8.7
2     2 Steven Ad~ C        28 MEM      76  1999  17.6 0.56   0.003 0.518   17.9
3     3 Bam Adeba~ C        24 MIA      56  1825  21.8 0.608  0.008 0.466    8.7
4     4 Santi Ald~ PF       21 MEM      32   360  10.2 0.452  0.364 0.242    9.4
5     5 LaMarcus ~ C        36 BRK      47  1050  19.6 0.604  0.1   0.223    7.8
6     6 Nickeil A~ SG       23 TOT      65  1466  10.5 0.475  0.497 0.16     2.7
# ... with 17 more variables: `DRB%` <dbl>, `TRB%` <dbl>, `AST%` <dbl>,
#   `STL%` <dbl>, `BLK%` <dbl>, `TOV%` <dbl>, `USG%` <dbl>, ...20 <lgl>,
#   OWS <dbl>, DWS <dbl>, WS <dbl>, `WS/48` <dbl>, ...25 <lgl>, OBPM <dbl>,
#   DBPM <dbl>, BPM <dbl>, VORP <dbl>
```

```r
nrow(nbaplayers)
```

```
[1] 812
```

Another way to look at nrow – we have 812 players from this season in our dataset.

What if we wanted to know how many players there were by position? To do that by hand, we'd have to take each of the 651 records and sort them into a pile. We'd put them in groups and then count them.

`dplyr` has a **group by** function in it that does just this. A massive amount of data analysis involves grouping like things together at some point. So it's a good place to start.

So to do this, we'll take our dataset and we'll introduce a new operator: %>%. The best way to read that operator, in my opinion, is to interpret that as "and then do this."

After we group them together, we need to count them. We do that first by saying we want to summarize our data (a count is a part of a summary). To get a summary, we have to tell it what we want. So in this case, we want a count. To get that, let's create a thing called total and set it equal to n(), which is `dplyr`s way of counting something.

Here's the code:

```
nbaplayers %>%
  group_by(Pos) %>%
  summarise(
    total = n()
  )
```

```
# A tibble: 11 x 2
   Pos   total
   <chr> <int>
 1 C       130
 2 C-PF      2
 3 PF      152
 4 PF-SF     1
 5 PG      156
 6 PG-SG     1
 7 SF      157
 8 SF-SG     5
 9 SG      199
10 SG-PG     4
11 SG-SF     5
```

So let's walk through that. We start with our dataset – `nbaplayers` – and then we tell it to group the data by a given field in the data which we get by looking at either the output of `head` or you can look in the environment where you'll see `nbaplayers`.

In this case, we wanted to group together positions, signified by the field name Pos. After we group the data, we need to count them up. In dplyr, we use `summarize` which can do more than just count things. Inside the parentheses in summarize, we set up the summaries we want. In this case, we just want a count of the positions: `total = n()`, says create a new field, called `total` and set it equal to `n()`, which might look weird, but it's common in stats.

The number of things in a dataset? Statisticians call in n. There are n number of players in this dataset. So `n()` is a function that counts the number of things there are.

And when we run that, we get a list of positions with a count next to them. But it's not in any order. So we'll add another And Then Do This %>% and use `arrange`. Arrange does what you think it does – it arranges data in order. By default, it's in ascending order – smallest to largest. But if we want to know the county with the most mountain lion sightings, we need to sort it in descending order. That looks like this:

```
nbaplayers %>%
  group_by(Pos) %>%
  summarise(
    total = n()
  ) %>% arrange(desc(total))
```

```
# A tibble: 11 x 2
   Pos   total
   <chr> <int>
 1 SG      199
 2 SF      157
 3 PG      156
 4 PF      152
 5 C       130
 6 SF-SG     5
 7 SG-SF     5
 8 SG-PG     4
 9 C-PF      2
10 PF-SF     1
11 PG-SG     1
```

So the most common position in the NBA? Shooting guard, followed by small forward (although it's close between SF and point guard).

We can, if we want, group by more than one thing. Which team has the most of a single position? To do that, we can group by the team – called Tm in the data – and position, or Pos in the data:

```
nbaplayers %>%
  group_by(Tm, Pos) %>%
  summarise(
    total = n()
  ) %>% arrange(desc(total))
```

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

# A tibble: 161 x 3
# Groups:   Tm [31]
   Tm    Pos   total
   <chr> <chr> <int>
 1 TOT   PF       18
 2 TOT   SG       18
 3 TOT   PG       17
 4 TOT   SF       14
 5 TOT   C        12
 6 BOS   SG        9
 7 DAL   PG        9
 8 LAC   SG        9
 9 POR   SF        9
10 SAS   SG        9
# ... with 151 more rows
```

So wait, what team is TOT?

Valuable lesson: whoever collects the data has opinions on how to solve problems. In this case, Basketball Reference, when a player get's traded, records stats for the player's first team, their second team, and a combined season total for a team called TOT, meaning Total. Is there a team abbreviated TOT? No. So ignore them here.

Boston has 8 shooting guards. Dallas has 9 point guards? You can learn a bit about how a team is assembled by looking at these simple counts.

4.2 Other aggregates: Mean and median

In the last example, we grouped some data together and counted it up, but there's so much more you can do. You can do multiple measures in a single step as well.

Sticking with our NBA player data, we can calculate any number of measures inside summarize. Here, we'll use R's built in mean and median functions to calculate … well, you get the idea.

Let's look just a the number of minutes each position gets.

```
nbaplayers %>%
  group_by(Pos) %>%
  summarise(
    count = n(),
```

```
    mean_minutes = mean(MP),
    median_minutes = median(MP)
  )
```

```
# A tibble: 11 x 4
   Pos   count mean_minutes median_minutes
   <chr> <int>        <dbl>          <dbl>
 1 C       130         835.           616
 2 C-PF      2        1506.          1506.
 3 PF      152         842.           502.
 4 PF-SF     1        1940           1940
 5 PG      156         815.           543
 6 PG-SG     1        2419           2419
 7 SF      157         775.           491
 8 SF-SG     5        1074.           935
 9 SG      199         821.           665
10 SG-PG     4        1350.          1340.
11 SG-SF     5         654.           434
```

So there's 812 players in the data. Let's look at shooting guards. The average shooting guard plays 820 minutes and the median is 665 minutes.

Why?

Let's let sort help us.

```
nbaplayers %>% arrange(desc(MP))
```

```
# A tibble: 812 x 29
      Rk Player    Pos     Age Tm        G    MP   PER `TS%` `3PAr`   FTr `ORB%`
   <dbl> <chr>     <chr> <dbl> <chr> <dbl> <dbl> <dbl> <dbl>  <dbl> <dbl>  <dbl>
 1    66 Mikal Br~ SF       25 PHO      82  2854  14.4 0.627  0.364 0.183    2.9
 2    67 Miles Br~ PF       23 CHO      80  2837  17.9 0.593  0.383 0.274    3.3
 3   134 DeMar De~ PF       32 CHI      76  2743  23.1 0.59   0.093 0.386    2.3
 4   526 Jayson T~ SF       23 BOS      76  2731  21.8 0.578  0.416 0.3      3.5
 5    47 Saddiq B~ SF       22 DET      82  2704  14   0.529  0.537 0.224    4.1
 6   214 Tyrese H~ SG-PG    21 TOT      77  2695  18.2 0.595  0.428 0.209    2.5
 7   576 Russell ~ PG       33 LAL      78  2678  15   0.512  0.215 0.324    4.4
 8   602 Trae You~ PG       23 ATL      76  2652  25.4 0.603  0.395 0.358    2.1
 9   358 Tyrese M~ PG       21 PHI      75  2650  16.2 0.594  0.311 0.247    1.1
10   168 Dorian F~ PF       28 DAL      80  2644  12.5 0.604  0.629 0.12     5.1
```

```
# ... with 802 more rows, and 17 more variables: `DRB%` <dbl>, `TRB%` <dbl>,
#   `AST%` <dbl>, `STL%` <dbl>, `BLK%` <dbl>, `TOV%` <dbl>, `USG%` <dbl>,
#   ...20 <lgl>, OWS <dbl>, DWS <dbl>, WS <dbl>, `WS/48` <dbl>, ...25 <lgl>,
#   OBPM <dbl>, DBPM <dbl>, BPM <dbl>, VORP <dbl>
```

The player with the most minutes on the floor is a small forward. So that means there's Mikal Bridges rolling up 2,854 minutes in a season, and then there's Boston sensation Justin Jackson. Never heard of Justin Jackson? Might be because he logged two minutes in one game this season.

That's a huge difference.

So when choosing a measure of the middle, you have to ask yourself – could I have extremes? Because a median won't be sensitive to extremes. It will be the point at which half the numbers are above and half are below. The average or mean will be a measure of the middle, but if you have a bunch of pine riders and then one ironman superstar, the average will be wildly skewed.

4.3 Even more aggregates

There's a ton of things we can do in summarize – we'll work with more of them as the course progresses – but here's a few other questions you can ask.

Which position in the NBA plays the most minutes? And what is the highest and lowest minute total for that position? And how wide is the spread between minutes? We can find that with `sum` to add up the minutes to get the total minutes, `min` to find the minimum minutes, `max` to find the maximum minutes and `sd` to find the standard deviation in the numbers.

```r
nbaplayers %>%
  group_by(Pos) %>%
  summarise(
    total = sum(MP),
    avgminutes = mean(MP),
    minminutes = min(MP),
    maxminutes = max(MP),
    stdev = sd(MP)) %>% arrange(desc(total))
```

```
# A tibble: 11 x 6
   Pos    total avgminutes minminutes maxminutes stdev
   <chr>  <dbl>      <dbl>      <dbl>      <dbl> <dbl>
 1 SG    163361       821.          1       2499  766.
 2 PF    127917       842.          1       2837  793.
```

```
 3 PG    127159       815.          1       2678  801.
 4 SF    121695       775.          2       2854  779.
 5 C     108505       835.          2       2476  721.
 6 SG-PG   5401      1350.         25       2695 1163.
 7 SF-SG   5371      1074.        212       2057  709.
 8 SG-SF   3272       654.        104       1791  697.
 9 C-PF    3013      1506.        877       2136  890.
10 PG-SG   2419      2419        2419       2419   NA
11 PF-SF   1940      1940        1940       1940   NA
```

So again, no surprise, shooting guards spend the most minutes on the floor in the NBA. They average 820 minutes, but we noted why that's trouble. The minimum is a one-minute wonder, max is some team failing at load management, and the standard deviation is a measure of how spread out the data is. In this case, not the highest spread among positions, but pretty high. So you know you've got some huge minutes players and a bunch of bench players.

5 Mutating data

One of the most common data analysis techniques is to look at change over time. The most common way of comparing change over time is through percent change. The math behind calculating percent change is very simple, and you should know it off the top of your head. The easy way to remember it is:

`(new - old) / old`

Or new minus old divided by old. Your new number minus the old number, the result of which is divided by the old number. To do that in R, we can use `dplyr` and `mutate` to calculate new metrics in a new field using existing fields of data.

So first we'll import the tidyverse so we can read in our data and begin to work with it.

```
library(tidyverse)
```

Now you'll need a common and simple dataset of total attendance at NCAA football games over the last few seasons.

You'll import it something like this.

```
attendance <- read_csv('data/attendance.csv')
```

```
Rows: 149 Columns: 9
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (2): Institution, Conference
dbl (7): 2013, 2014, 2015, 2016, 2017, 2018, 2019

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.
```

If you want to see the first six rows – handy to take a peek at your data – you can use the function `head`.

```
head(attendance)
```

```
# A tibble: 6 x 9
  Institution     Conference    `2013` `2014` `2015` `2016` `2017` `2018` `2019`
  <chr>           <chr>          <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 Air Force       MWC           228562 168967 156158 177519 174924 166205 162505
2 Akron           MAC           107101  55019 108588  62021 117416  92575 107752
3 Alabama         SEC           710538 710736 707786 712747 712053 710931 707817
4 Appalachian St. FBS Independ~ 149366     NA     NA     NA     NA     NA     NA
5 Appalachian St. Sun Belt          NA 138995 128755 156916 154722 131716 166640
6 Arizona         Pac-12        285713 354973 308355 338017 255791 318051 237194
```

The code to calculate percent change is pretty simple. Remember, with `summarize`, we used `n()` to count things. With `mutate`, we use very similar syntax to calculate a new value using other values in our dataset. So in this case, we're trying to do (new-old)/old, but we're doing it with fields. If we look at what we got when we did `head`, you'll see there's '2019' as the new data, and we'll use '2018' as the old data. So we're looking at one year. Then, to help us, we'll use arrange again to sort it, so we get the fastest growing school over one year.

```
attendance %>% mutate(
  change = (`2019` - `2018`)/`2018`
)
```

```
# A tibble: 149 x 10
   Institution Confe~1 `2013` `2014` `2015` `2016` `2017` `2018` `2019`   change
   <chr>       <chr>    <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>    <dbl>
 1 Air Force   MWC     228562 168967 156158 177519 174924 166205 162505 -0.0223
 2 Akron       MAC     107101  55019 108588  62021 117416  92575 107752  0.164
 3 Alabama     SEC     710538 710736 707786 712747 712053 710931 707817 -0.00438
 4 Appalachia~ FBS In~ 149366     NA     NA     NA     NA     NA     NA NA
 5 Appalachia~ Sun Be~     NA 138995 128755 156916 154722 131716 166640  0.265
 6 Arizona     Pac-12  285713 354973 308355 338017 255791 318051 237194 -0.254
 7 Arizona St. Pac-12  501509 343073 368985 286417 359660 291091 344161  0.182
 8 Arkansas    SEC     431174 399124 471279 487067 442569 367748 356517 -0.0305
 9 Arkansas S~ Sun Be~ 149477 149163 138043 136200 119538 119001 124017  0.0422
10 Army West ~ FBS In~ 169781 171310 185946 163267 185543 190156 185935 -0.0222
# ... with 139 more rows, and abbreviated variable name 1: Conference
```

What do we see right away? Do those numbers look like we expect them to? No. They're a decimal expressed as a percentage. So let's fix that by multiplying by 100.

```
attendance %>% mutate(
  change = ((`2019` - `2018`)/`2018`)*100
```

```
  )
```

```
# A tibble: 149 x 10
   Institution  Confe~1 `2013` `2014` `2015` `2016` `2017` `2018` `2019`  change
   <chr>        <chr>    <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>   <dbl>
 1 Air Force    MWC     228562 168967 156158 177519 174924 166205 162505  -2.23
 2 Akron        MAC     107101  55019 108588  62021 117416  92575 107752  16.4
 3 Alabama      SEC     710538 710736 707786 712747 712053 710931 707817  -0.438
 4 Appalachian~ FBS In~ 149366     NA     NA     NA     NA     NA     NA  NA
 5 Appalachian~ Sun Be~     NA 138995 128755 156916 154722 131716 166640  26.5
 6 Arizona      Pac-12  285713 354973 308355 338017 255791 318051 237194 -25.4
 7 Arizona St.  Pac-12  501509 343073 368985 286417 359660 291091 344161  18.2
 8 Arkansas     SEC     431174 399124 471279 487067 442569 367748 356517  -3.05
 9 Arkansas St. Sun Be~ 149477 149163 138043 136200 119538 119001 124017   4.22
10 Army West P~ FBS In~ 169781 171310 185946 163267 185543 190156 185935  -2.22
# ... with 139 more rows, and abbreviated variable name 1: Conference
```

Now, does this ordering do anything for us? No. Let's fix that with arrange.

```
  attendance %>% mutate(
    change = ((`2019` - `2018`)/`2018`)*100
  ) %>% arrange(desc(change))
```

```
# A tibble: 149 x 10
   Institution   Confe~1 `2013` `2014` `2015` `2016` `2017` `2018` `2019` change
   <chr>         <chr>    <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 1 Kansas        Big 12  265187 204462 190972 154969 186490 116544 237122  103.
 2 Coastal Caro. Sun Be~     NA     NA     NA     NA  89754  53173  90112   69.5
 3 La.-Monroe    Sun Be~  85177  90540  58659  67057  49640  71048 100367   41.3
 4 Virginia      ACC     370234 275243 302994 239576 275788 238232 335040   40.6
 5 Louisiana Te~ C-USA    93332 100272 125863 102059 142626  87623 122631   40.0
 6 North Caroli~ ACC     360500 328000 347500 301500 350500 218111 303000   38.9
 7 Fla. Atlantic C-USA    72758  70608 105703  60437 151580 102308 140857   37.7
 8 Texas St.     Sun Be~ 108371 123589 108998 108717 104680  78689 102840   30.7
 9 New Mexico S~ FBS In~ 112347     NA     NA     NA     NA  68529  89523   30.6
10 San Jose St.  MWC      98174  75340  91869  92515  85235  71276  92374   29.6
# ... with 139 more rows, and abbreviated variable name 1: Conference
```

So who had the most growth in 2019 compared to the year before? Kansas.

5.1 A more complex example

There's metric in basketball that's easy to understand – shooting percentage. It's the number of shots made divided by the number of shots attempted. Simple, right? Except it's a little too simple. Because what about three point shooters? They tend to be more vailable because the three point shot is worth more. What about players who get to the line? In shooting percentage, free throws are nowhere to be found.

Basketball nerds, because of these weaknesses, have created a new metric called True Shooting Percentage. True shooting percentage takes into account all aspects of a players shooting to determine who the real shooters are.

Using `dplyr` and `mutate`, we can calculate true shooting percentage. So let's look at a new dataset, one of every college basketball player's season stats in 2021-22 season. It's a dataset of 5,688 players, and we've got 59 variables – one of them is True Shooting Percentage, but we're going to ignore that.

Import it like this:

```
players <- read_csv("data/players22.csv")
```

```
Rows: 5688 Columns: 59
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (10): Team, Player, Class, Pos, Height, Hometown, High School, Summary, ...
dbl (49): #, Weight, Rk.x, G, GS, MP, FG, FGA, FG%, 2P, 2PA, 2P%, 3P, 3PA, 3...

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.
```

The basic true shooting percentage formula is `(Points / (2*(FieldGoalAttempts + (.44 * FreeThrowAttempts)))) * 100`. Let's talk that through. Points divided by a lot. It's really field goal attempts plus 44 percent of the free throw attempts. Why? Because that's about what a free throw is worth, compared to other ways to score. After adding those things together, you double it. And after you divide points by that number, you multiply the whole lot by 100.

In our data, we need to be able to find the fields so we can complete the formula. To do that, one way is to use the Environment tab in R Studio. In the Environment tab is a listing of all the data you've imported, and if you click the triangle next to it, it'll list all the field names, giving you a bit of information about each one.

So what does True Shooting Percentage look like in code?

Let's think about this differently. Who had the best true shooting season last year?

```
players %>%
  mutate(trueshooting = (PTS/(2*(FGA + (.44*FTA))))*100) %>%
  arrange(desc(trueshooting))
```

```
# A tibble: 5,688 x 60
   Team     Player   `#` Class Pos   Height Weight Homet~1 High ~2 Summary  Rk.x
   <chr>    <chr>  <dbl> <chr> <chr> <chr>   <dbl> <chr>   <chr>   <chr>   <dbl>
 1 Washing~ Noah ~    32 JR    G     6-3       185 Seattl~ Garfie~ 0.6 Pt~    13
 2 Troy Tr~ DJ Ro~    34 FR    G     6-1       160 Monroe~ Oakman~ 1.0 Pt~    17
 3 Syracus~ Arthu~    33 SO    G     6-3       185 Wallin~ Sheeha~ 0.8 Pt~    13
 4 Stony B~ Prest~    12 <NA>  G     6-4       205 Syracu~ Jamesv~ 3.0 Pt~    14
 5 USC Tro~ Amar ~    55 SR    G     5-7       150 Long B~ Fremon~ 3.0 Pt~    13
 6 South C~ Ford ~    44 JR    G     6-3       183 Charlo~ United~ 3.0 Pt~    16
 7 UNC Gre~ Jevon~    15 <NA>  G     6-3       170 Durham~ <NA>    1.0 Pt~    13
 8 New Mex~ Safi ~     2 <NA>  G     5-11      165 Las Cr~ <NA>    3.0 Pt~    15
 9 New Mex~ Camer~    20 <NA>  G     6-1       165 Kirtla~ Kirtla~ 0.8 Pt~    15
10 Mercer ~ John ~    15 SO    F     6-6       210 Atlant~ Marist~ 1.5 Pt~    14
# ... with 5,678 more rows, 49 more variables: G <dbl>, GS <dbl>, MP <dbl>,
```

```
#   FG <dbl>, FGA <dbl>, `FG%` <dbl>, `2P` <dbl>, `2PA` <dbl>, `2P%` <dbl>,
#   `3P` <dbl>, `3PA` <dbl>, `3P%` <dbl>, FT <dbl>, FTA <dbl>, `FT%` <dbl>,
#   ORB <dbl>, DRB <dbl>, TRB <dbl>, AST <dbl>, STL <dbl>, BLK <dbl>,
#   TOV <dbl>, PF <dbl>, PTS <dbl>, Rk.y <dbl>, PER <dbl>, `TS%` <dbl>,
#   `eFG%` <dbl>, `3PAr` <dbl>, FTr <dbl>, PProd <dbl>, `ORB%` <dbl>,
#   `DRB%` <dbl>, `TRB%` <dbl>, `AST%` <dbl>, `STL%` <dbl>, `BLK%` <dbl>, ...
```

You'll be forgiven if you did not hear about USC's shooting sensation Amar Ross. He played in one game, took one shot and actually hit it. It happened to be a three pointer, which is one more three pointer than I've hit in college basketball. So props to him. Does that mean he had the best true shooting season in college basketball in the 2021-22 season?

Not hardly.

We'll talk about how to narrow the pile and filter out data in the next chapter.

6 Filters and selections

More often than not, we have more data than we want. Sometimes we need to be rid of that data. In `dplyr`, there's two ways to go about this: filtering and selecting.

Filtering creates a subset of the data based on criteria. All records where the count is greater than 10. All records that match "Maryland". Something like that.

Selecting simply returns only the fields named. So if you only want to see School and Attendance, you select those fields. When you look at your data again, you'll have two columns. If you try to use one of your columns that you had before you used select, you'll get an error.

Let's work with our football attendance data to show some examples.

First we'll need the tidyverse.

```
library(tidyverse)
```

Now import the data.

```
attendance <- read_csv('data/attendance.csv')
```

```
Rows: 149 Columns: 9
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (2): Institution, Conference
dbl (7): 2013, 2014, 2015, 2016, 2017, 2018, 2019

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.
```

So, first things first, let's say we don't care about all this Air Force, Akron, Alabama crap and just want to see Dear Old Maryland We do that with `filter` and then we pass it a condition.

Before we do that, a note about conditions. Most of the conditional operators you'll understand – greater than and less than are $>$ and $<$. The tough one to remember is equal to. In

conditional statements, equal to is == not =. If you haven't noticed, = is a variable assignment operator, not a conditional statement. So equal is == and NOT equal is !=.

So if you want to see Institutions equal to Maryland, you do this:

```
attendance %>% filter(Institution == "Maryland")
```

```
# A tibble: 2 x 9
  Institution Conference `2013` `2014` `2015` `2016` `2017` `2018` `2019`
  <chr>       <chr>       <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 Maryland    ACC        288946     NA     NA     NA     NA     NA     NA
2 Maryland    Big Ten        NA 281884 310389 237690 237859 201562 226871
```

Or if we want to see schools that had more than half a million people buy tickets to a football game in a season, we do the following. NOTE THE BACKTICKS.

```
attendance %>% filter(`2019` >= 500000)
```

```
# A tibble: 16 x 9
   Institution    Conference    `2013` `2014` `2015` `2016` `2017` `2018` `2019`
   <chr>          <chr>          <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 1 Alabama        SEC           710538 710736 707786 712747 712053 710931 707817
 2 Auburn         SEC           685252 612157 612157 695498 605120 591236 600355
 3 Clemson        ACC           574333 572262 588266 566787 565412 562799 566074
 4 Florida        SEC           524638 515001 630457 439229 520290 576299 508103
 5 Georgia        SEC           556476 649222 649222 556476 556476 649222 649722
 6 LSU            SEC           639927 712063 654084 708618 591034 705733 705892
 7 Michigan       Big Ten       781144 734364 771174 883741 669534 775156 780215
 8 Nebraska       Big Ten       727466 638744 629983 631402 628583 623240 625436
 9 Notre Dame     FBS Independ~ 484770 484770 484770 484770 543354 465732 534017
10 Ohio St.       Big Ten       734528 744075 750705 750944 752464 713630 723679
11 Penn St.       Big Ten       676112 711358 698590 701800 746946 738396 739747
12 South Carolina SEC           576805 569664 472934 538441 550099 515396 545737
13 Tennessee      SEC           669087 698276 704088 706776 670454 650887 702912
14 Texas          Big 12        593857 564618 540210 587283 556667 586277 577834
15 Texas A&M      SEC           697003 630735 725354 713418 691612 698908 711258
16 Wisconsin      Big Ten       552378 556642 546099 476144 551766 540072 535301
```

But what if we want to see all of the Power Five conferences? We *could* use conditional logic in our filter. The conditional logic operators are `|` for OR and `&` for AND. NOTE: AND means all conditions have to be met. OR means any of the conditions work. So be careful about boolean logic.

```
attendance %>% filter(Conference == "Big 10" | Conference == "SEC" | Conference == "Pac-12
```

```
# A tibble: 51 x 9
   Institution    Conference `2013` `2014` `2015` `2016` `2017` `2018` `2019`
   <chr>          <chr>       <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 1 Alabama        SEC        710538 710736 707786 712747 712053 710931 707817
 2 Arizona        Pac-12     285713 354973 308355 338017 255791 318051 237194
 3 Arizona St.    Pac-12     501509 343073 368985 286417 359660 291091 344161
 4 Arkansas       SEC        431174 399124 471279 487067 442569 367748 356517
 5 Auburn         SEC        685252 612157 612157 695498 605120 591236 600355
 6 Baylor         Big 12     321639 280257 276960 275029 262978 248017 318621
 7 Boston College ACC        198035 239893 211433 192942 215546 263363 205111
 8 California     Pac-12     345303 286051 292797 279769 219290 300061 254597
 9 Clemson        ACC        574333 572262 588266 566787 565412 562799 566074
10 Colorado       Pac-12     230778 226670 236331 279652 282335 274852 297435
# ... with 41 more rows
```

But that's a lot of repetitive code. And a lot of typing. And typing is the devil. So what if we could create a list and pass it into the filter? It's pretty simple.

We can create a new variable – remember variables can represent just about anything – and create a list. To do that we use the `c` operator, which stands for concatenate. That just means take all the stuff in the parenthesis after the c and bunch it into a list.

Note here: text is in quotes. If they were numbers, we wouldn't need the quotes.

```
powerfive <- c("SEC", "Big Ten", "Pac-12", "Big 12", "ACC")
```

Now with a list, we can use the %in% operator. It does what you think it does – it gives you data that matches things IN the list you give it.

```
attendance %>% filter(Conference %in% powerfive)
```

```
# A tibble: 65 x 9
   Institution    Conference `2013` `2014` `2015` `2016` `2017` `2018` `2019`
   <chr>          <chr>       <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 1 Alabama        SEC        710538 710736 707786 712747 712053 710931 707817
 2 Arizona        Pac-12     285713 354973 308355 338017 255791 318051 237194
 3 Arizona St.    Pac-12     501509 343073 368985 286417 359660 291091 344161
 4 Arkansas       SEC        431174 399124 471279 487067 442569 367748 356517
 5 Auburn         SEC        685252 612157 612157 695498 605120 591236 600355
```

```
 6 Baylor         Big 12     321639 280257 276960 275029 262978 248017 318621
 7 Boston College ACC        198035 239893 211433 192942 215546 263363 205111
 8 California     Pac-12     345303 286051 292797 279769 219290 300061 254597
 9 Clemson        ACC        574333 572262 588266 566787 565412 562799 566074
10 Colorado       Pac-12     230778 226670 236331 279652 282335 274852 297435
# ... with 55 more rows
```

6.1 Selecting data to make it easier to read

So now we have our Power Five list. What if we just wanted to see attendance from the most recent season and ignore all the rest? Select to the rescue.

```
attendance %>% filter(Conference %in% powerfive) %>% select(Institution, Conference, `2018
```

```
# A tibble: 65 x 3
   Institution    Conference `2018`
   <chr>          <chr>       <dbl>
 1 Alabama        SEC        710931
 2 Arizona        Pac-12     318051
 3 Arizona St.    Pac-12     291091
 4 Arkansas       SEC        367748
 5 Auburn         SEC        591236
 6 Baylor         Big 12     248017
 7 Boston College ACC        263363
 8 California     Pac-12     300061
 9 Clemson        ACC        562799
10 Colorado       Pac-12     274852
# ... with 55 more rows
```

If you have truly massive data, Select has tools to help you select fields that start_with the same things or ends with a certain word. The documentation will guide you if you need those someday. For 90 plus percent of what we do, just naming the fields will be sufficient.

6.2 Using conditional filters to set limits

Let's return to the problem of one-hit wonders in basketball mucking up our true shooting analysis. How can we set limits in something like a question of who had the best season? Let's grab every player from last season.

Let's get set up similar to the previous chapter.

```
players <- read_csv("data/players22.csv")
```

```
Rows: 5688 Columns: 59
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (10): Team, Player, Class, Pos, Height, Hometown, High School, Summary, ...
dbl (49): #, Weight, Rk.x, G, GS, MP, FG, FGA, FG%, 2P, 2PA, 2P%, 3P, 3PA, 3...

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.
```

```
players %>%
  mutate(trueshooting = (PTS/(2*(FGA + (.44*FTA))))*100) %>%
  arrange(desc(trueshooting))
```

```
# A tibble: 5,688 x 60
   Team     Player   `#` Class Pos   Height Weight Homet~1 High ~2 Summary  Rk.x
   <chr>    <chr>  <dbl> <chr> <chr> <chr>   <dbl> <chr>   <chr>   <chr>   <dbl>
 1 Washing~ Noah ~    32 JR    G     6-3       185 Seattl~ Garfie~ 0.6 Pt~    13
 2 Troy Tr~ DJ Ro~    34 FR    G     6-1       160 Monroe~ Oakman~ 1.0 Pt~    17
 3 Syracus~ Arthu~    33 SO    G     6-3       185 Wallin~ Sheeha~ 0.8 Pt~    13
 4 Stony B~ Prest~    12 <NA>  G     6-4       205 Syracu~ Jamesv~ 3.0 Pt~    14
 5 USC Tro~ Amar ~    55 SR    G     5-7       150 Long B~ Fremon~ 3.0 Pt~    13
 6 South C~ Ford ~    44 JR    G     6-3       183 Charlo~ United~ 3.0 Pt~    16
 7 UNC Gre~ Jevon~    15 <NA>  G     6-3       170 Durham~ <NA>    1.0 Pt~    13
 8 New Mex~ Safi ~     2 <NA>  G     5-11      165 Las Cr~ <NA>    3.0 Pt~    15
 9 New Mex~ Camer~    20 <NA>  G     6-1       165 Kirtla~ Kirtla~ 0.8 Pt~    15
10 Mercer ~ John ~    15 SO    F     6-6       210 Atlant~ Marist~ 1.5 Pt~    14
# ... with 5,678 more rows, 49 more variables: G <dbl>, GS <dbl>, MP <dbl>,
#   FG <dbl>, FGA <dbl>, `FG%` <dbl>, `2P` <dbl>, `2PA` <dbl>, `2P%` <dbl>,
#   `3P` <dbl>, `3PA` <dbl>, `3P%` <dbl>, FT <dbl>, FTA <dbl>, `FT%` <dbl>,
#   ORB <dbl>, DRB <dbl>, TRB <dbl>, AST <dbl>, STL <dbl>, BLK <dbl>,
#   TOV <dbl>, PF <dbl>, PTS <dbl>, Rk.y <dbl>, PER <dbl>, `TS%` <dbl>,
#   `eFG%` <dbl>, `3PAr` <dbl>, FTr <dbl>, PProd <dbl>, `ORB%` <dbl>,
#   `DRB%` <dbl>, `TRB%` <dbl>, `AST%` <dbl>, `STL%` <dbl>, `BLK%` <dbl>, ...
```

In that season, USC's Amar Ross is your One Shot One Three True Shooting champion.

In most contests, like the batting title in Major League Baseball, there's a minimum number of X to qualify. In baseball, it's at bats. In basketball, it attempts. So let's set a floor and see how it changes. What if we said you had to have played 100 minutes in a season? The top players in college basketball play more than 1000 minutes in a season. So 100 is not that much. Let's try it and see.

```
players %>%
  mutate(trueshooting = (PTS/(2*(FGA + (.44*FTA))))*100) %>%
  arrange(desc(trueshooting)) %>%
  filter(MP > 100)
```

```
# A tibble: 3,776 x 60
   Team     Player   `#` Class Pos   Height Weight Homet~1 High ~2 Summary  Rk.x
   <chr>    <chr>  <dbl> <chr> <chr> <chr>   <dbl> <chr>   <chr>   <chr>   <dbl>
 1 VMI Key~ Coope~    15 FR    G     6-5       185 Frisco~ Wakela~ 4.3 Pt~     8
 2 Samford~ Quinn~    13 SO    G     6-5       195 Johns ~ Mount ~ 2.8 Pt~     9
 3 Detroit~ Kyle ~     2 SO    G     6-0       180 Detroi~ Cass T~ 1.0 Pt~    11
 4 Pitt Pa~ Danie~     4 SR    F     6-7       220 Bladen~ Bladen~ 3.3 Pt~     9
 5 Hofstra~ Jarro~     2 SR    F     6-8       220 Cresce~ Cushin~ 4.2 Pt~     8
 6 Cornell~ Kelle~    15 SO    F     6-7       190 Plano,~ Presto~ 8.6 Pt~     4
 7 New Mex~ Virsh~    12 JR    G     6-2       165 Milwau~ Hillcr~ 4.9 Pt~    11
 8 Bowling~ Isaac~    12 JR    G     6-4       185 Bowlin~ Bowlin~ 2.6 Pt~    13
 9 Princet~ Max J~    40 SR    G     6-3       195 High P~ Woodbe~ 3.1 Pt~     7
10 Duke Bl~ Mark ~    15 SO    C     7-0       243 Virgin~ IMG Ac~ 11.0 P~     4
# ... with 3,766 more rows, 49 more variables: G <dbl>, GS <dbl>, MP <dbl>,
#   FG <dbl>, FGA <dbl>, `FG%` <dbl>, `2P` <dbl>, `2PA` <dbl>, `2P%` <dbl>,
#   `3P` <dbl>, `3PA` <dbl>, `3P%` <dbl>, FT <dbl>, FTA <dbl>, `FT%` <dbl>,
#   ORB <dbl>, DRB <dbl>, TRB <dbl>, AST <dbl>, STL <dbl>, BLK <dbl>,
#   TOV <dbl>, PF <dbl>, PTS <dbl>, Rk.y <dbl>, PER <dbl>, `TS%` <dbl>,
#   `eFG%` <dbl>, `3PAr` <dbl>, FTr <dbl>, PProd <dbl>, `ORB%` <dbl>,
#   `DRB%` <dbl>, `TRB%` <dbl>, `AST%` <dbl>, `STL%` <dbl>, `BLK%` <dbl>, ...
```

Now you get VMI's Cooper Sisco, who played in 15 games and was on the floor for 119 minutes. So he played regularly, but not a lot. But in that time, he only attempted 38 shots, and made 63 percent of them. In other words, when he shot, he probably scored. He just rarely shot.

So is 100 minutes our level? Here's the truth – there's not really an answer here. We're picking a cutoff. If you can cite a reason for it and defend it, then it probably works.

6.3 Top list

One last little dplyr trick that's nice to have in the toolbox is a shortcut for selecting only the top values for your dataset. Want to make a Top 10 List? Or Top 25? Or Top Whatever You Want? It's easy.

So what are the top 10 Power Five schools by season attendance. All we're doing here is chaining commands together with what we've already got. We're *filtering* by our list of Power Five conferences, we're *selecting* the three fields we need, now we're going to *arrange* it by total attendance and then we'll introduce the new function: `top_n`. The `top_n` function just takes a number. So we want a top 10 list? We do it like this:

```
attendance %>% filter(Conference %in% powerfive) %>% select(Institution, Conference, `2019
```

```
Selecting by 2019
```

```
# A tibble: 10 x 3
   Institution Conference `2019`
   <chr>       <chr>       <dbl>
 1 Michigan    Big Ten    780215
 2 Penn St.    Big Ten    739747
 3 Ohio St.    Big Ten    723679
 4 Texas A&M   SEC        711258
 5 Alabama     SEC        707817
 6 LSU         SEC        705892
 7 Tennessee   SEC        702912
 8 Georgia     SEC        649722
 9 Nebraska    Big Ten    625436
10 Auburn      SEC        600355
```

That's all there is to it. Just remember – for it to work correctly, you need to sort your data BEFORE you run top_n. Otherwise, you're just getting the first 10 values in the list. The function doesn't know what field you want the top values of. You have to do it.

7 Transforming data

Sometimes long data needs to be wide, and sometimes wide data needs to be long. I'll explain.

You are soon going to discover that long before you can visualize data, **you need to have it in a form that the visualization library can deal with**. One of the ways that isn't immediately obvious is **how your data is cast**. Most of the data you will encounter will be **wide – each row will represent a single entity with multiple measures for that entity**. So think of states. Your row of your dataset could have the state name, population, average life expectancy and other demographic data.

But what if your visualization library needs one row for each measure? So state, data type and the data. Maryland, Population, 6,177,224. That's one row. Then the next row is Maryland, Average Life Expectancy, 78.5 That's the next row. That's where recasting your data comes in.

We can use a library called `tidyr` to `pivot_longer` or `pivot_wider` the data, depending on what we need. We'll use a dataset of college football attendance to demonstrate.

First we need some libraries.

```
library(tidyverse)
```

Now we'll load the data.

```
attendance <- read_csv('data/attendance.csv')
```

```
Rows: 149 Columns: 9
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (2): Institution, Conference
dbl (7): 2013, 2014, 2015, 2016, 2017, 2018, 2019

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.
```

```r
attendance
```

```
# A tibble: 149 x 9
   Institution     Conference   `2013` `2014` `2015` `2016` `2017` `2018` `2019`
   <chr>           <chr>         <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 1 Air Force       MWC          228562 168967 156158 177519 174924 166205 162505
 2 Akron           MAC          107101  55019 108588  62021 117416  92575 107752
 3 Alabama         SEC          710538 710736 707786 712747 712053 710931 707817
 4 Appalachian St. FBS Indepen~ 149366     NA     NA     NA     NA     NA     NA
 5 Appalachian St. Sun Belt         NA 138995 128755 156916 154722 131716 166640
 6 Arizona         Pac-12       285713 354973 308355 338017 255791 318051 237194
 7 Arizona St.     Pac-12       501509 343073 368985 286417 359660 291091 344161
 8 Arkansas        SEC          431174 399124 471279 487067 442569 367748 356517
 9 Arkansas St.    Sun Belt     149477 149163 138043 136200 119538 119001 124017
10 Army West Point FBS Indepen~ 169781 171310 185946 163267 185543 190156 185935
# ... with 139 more rows
```

So as you can see, each row represents a school, and then each column represents a year. This is great for calculating the percent change – we can subtract a column from a column and divide by that column. But later, when we want to chart each school's attendance over the years, we have to have each row be one team for one year. Maryland in 2013, then Maryland in 2014, and Maryland in 2015 and so on.

To do that, we use `pivot_longer` because we're making wide data long. Since all of the columns we want to make rows start with 20, we can use that in our `cols` directive. Then we give that column a name – Year – and the values for each year need a name too. Those are the attendance figure. We can see right away how this works.

```r
attendance %>% pivot_longer(cols = starts_with("20"), names_to = "Year", values_to = "Atte
```

```
# A tibble: 1,043 x 4
   Institution Conference Year  Attendance
   <chr>       <chr>      <chr>      <dbl>
 1 Air Force   MWC        2013      228562
 2 Air Force   MWC        2014      168967
 3 Air Force   MWC        2015      156158
 4 Air Force   MWC        2016      177519
 5 Air Force   MWC        2017      174924
 6 Air Force   MWC        2018      166205
 7 Air Force   MWC        2019      162505
 8 Akron       MAC        2013      107101
```

```
 9 Akron       MAC        2014       55019
10 Akron       MAC        2015      108588
# ... with 1,033 more rows
```

We've gone from 149 rows to more than 1,000, but that's expected when we have 7 years for each team.

7.1 Making long data wide

We can reverse this process using `pivot_wider`, which makes long data wide.

Why do any of this?

In some cases, you're going to be given long data and you need to calculate some metric using two of the years – a percent change for instance. So you'll need to make the data wide to do that. You might then have to re-lengthen the data now with the percent change. Some project require you to do all kinds of flexing like this. It just depends on the data.

So let's take what we made above and turn it back into wide data.

```
longdata <- attendance %>% pivot_longer(cols = starts_with("20"), names_to = "Year", value

longdata
```

```
# A tibble: 1,043 x 4
   Institution Conference Year  Attendance
   <chr>       <chr>      <chr>      <dbl>
 1 Air Force   MWC        2013      228562
 2 Air Force   MWC        2014      168967
 3 Air Force   MWC        2015      156158
 4 Air Force   MWC        2016      177519
 5 Air Force   MWC        2017      174924
 6 Air Force   MWC        2018      166205
 7 Air Force   MWC        2019      162505
 8 Akron       MAC        2013      107101
 9 Akron       MAC        2014       55019
10 Akron       MAC        2015      108588
# ... with 1,033 more rows
```

To `pivot_wider`, we just need to say where our column names are coming from – the Year – and where the data under it should come from – Attendance.

```r
longdata %>% pivot_wider(names_from = Year, values_from = Attendance)
```

```
# A tibble: 149 x 9
   Institution     Conference   `2013` `2014` `2015` `2016` `2017` `2018` `2019`
   <chr>           <chr>         <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 1 Air Force       MWC          228562 168967 156158 177519 174924 166205 162505
 2 Akron           MAC          107101  55019 108588  62021 117416  92575 107752
 3 Alabama         SEC          710538 710736 707786 712747 712053 710931 707817
 4 Appalachian St. FBS Indepen~ 149366     NA     NA     NA     NA     NA     NA
 5 Appalachian St. Sun Belt         NA 138995 128755 156916 154722 131716 166640
 6 Arizona         Pac-12       285713 354973 308355 338017 255791 318051 237194
 7 Arizona St.     Pac-12       501509 343073 368985 286417 359660 291091 344161
 8 Arkansas        SEC          431174 399124 471279 487067 442569 367748 356517
 9 Arkansas St.    Sun Belt     149477 149163 138043 136200 119538 119001 124017
10 Army West Point FBS Indepen~ 169781 171310 185946 163267 185543 190156 185935
# ... with 139 more rows
```

And just like that, we're back.

7.2 Why this matters

This matters because certain visualization types need wide or long data. A significant hurdle you will face for the rest of the semester is getting the data in the right format for what you want to do.

So let me walk you through an example using this data.

Let's look at Maryland's attendance over the time period. In order to do that, I need long data because that's what the charting library, `ggplot2`, needs. You're going to learn a lot more about ggplot later. Since this data is organized by school and conference, I also need to remove records that have no attendance data (because we have a Maryland in the ACC row).

```r
maryland <- longdata %>% filter(Institution == "Maryland") %>% filter(!is.na(Attendance))
```

Now that we have long data for just Maryland, we can chart it.

```r
ggplot(maryland, aes(x=Year, y=Attendance, group=1)) +
  geom_line() +
  scale_y_continuous(labels = scales::comma) +
  labs(x="Year", y="Attendance", title="A Steady Decline After 2015", subtitle="But hope a
```

```r
theme_minimal() +
theme(
  plot.title = element_text(size = 16, face = "bold"),
  axis.title = element_text(size = 10),
  axis.title.y = element_blank(),
  axis.text = element_text(size = 7),
  axis.ticks = element_blank(),
  panel.grid.minor = element_blank(),
  panel.grid.major.x = element_blank(),
  legend.position="bottom"
)
```

8 Significance tests

Now that we've worked with data a little, it's time to start asking more probing questions of our data. One of the most probing questions we can ask – one that so few sports journalists ask – is if the difference between this thing and the normal thing is real.

We have a perfect natural experiment going on in sports right now to show how significance tests work. The NBA, to salvage a season and get to the playoffs, put their players in a bubble – more accurately a hotel complex at Disney World in Orlando – and had them play games without fans.

So are the games different from other regular season games that had fans?

To answer this, we need to understand that a significance test is a way to determine if two numbers are *significantly* different from each other. Generally speaking, we're asking if a subset of data – a sample – is different from the total data pool – the population. Typically, this relies on data being in a normal distribution.

If it is, then we know certain things about it. Like the mean – the average – will be a line right at the peak of cases. And that 66 percent of cases will be in that red area – the first standard deviation.

A significance test will determine if a sample taken from that group is different from the total.

Significance testing involves stating a hypothesis. In our case, our hypothesis is that there is a difference between bubble games without people and regular games with people.

In statistics, the **null hypothesis** is the opposite of your hypothesis. In this case, that there is no difference between fans and no fans.

What we're driving toward is a metric called a p-value, which is the probability that you'd get your sample mean *if the null hypothesis is true.* So in our case, it's the probability we'd see the numbers we get if there was no difference between fans and no fans. If that probability is below .05, then we consider the difference significant and we reject the null hypothesis.

So let's see. We'll need a log of every game last NBA season. In this data, there's a field called COVID, which labels the game as a regular game or a bubble game.

Load the tidyverse.

```
library(tidyverse)
```

And import the data.

```
logs <- read_csv("data/nbabubble.csv")
```

```
Rows: 2118 Columns: 43
-- Column specification --------------------------------------------------------
Delimiter: ","
chr   (7): Season, Conference, Team, HomeAway, Opponent, W_L, COVID
dbl  (35): Game, TeamScore, OpponentScore, TeamFG, TeamFGA, TeamFGPCT, Team3...
date  (1): Date

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.
```

First, let's just look at scoring. Here's a theory: fans make players nervous. The screaming makes players tense up, and tension makes for bad shooting. An alternative to this: screaming fans make you defend harder. So my hypothesis is that not only is the scoring different, it's lower.

First things first, let's create a new field, called `totalpoints` and add the two scores together. We'll need this, so we're going to make this a new dataframe called `points`.

```
points <- logs %>% mutate(totalpoints = TeamScore + OpponentScore )
```

Typically speaking, with significance tests, the process involves creating two different means and then running a bunch of formulas on them. R makes this easy by giving you a `t.test` function, which does all the work for you. What we have to tell it is what is the value we are testing, over which groups, and from what data. It looks like this:

```
t.test(totalpoints ~ COVID, data=points)
```

```
    Welch Two Sample t-test

data:  totalpoints by COVID
t = -5.232, df = 206.88, p-value = 4.099e-07
alternative hypothesis: true difference in means between group With Fans and group Without Fa
95 percent confidence interval:
 -11.64698  -5.27178
sample estimates:
   mean in group With Fans mean in group Without Fans
                  222.8929                   231.3523
```

Now let's talk about the output. I prefer to read these bottom up. So at the bottom, it says that the mean number of points score in an NBA game With Fans is 222.89. The mean scored in games Without Fans is 231.35. That means teams are scoring almost 8.5 points MORE without fans on average.

But, some games are defenseless track meets, some games are defensive slugfests. We learned that averages can be skewed by extremes. So the next thing we need to look at is the p-value. Remember, this is the probability that we'd get this sample mean – the without fans mean – if there was no difference between fans and no fans.

The probability? 4.099e-07 or 4.099 x 10 to the -7 power. Don't remember your scientific notation? That's .00000004099. The decimal, seven zeros and the number.

Remember, if the probability is below .05, then we determine that this number is statistically significant. We'll talk more about statistical significance soon, but in this case, statistical significance means that our hypothesis is correct: points are different without fans than with. And since our hypothesis is correct, we *reject the null hypothesis* and we can confidently say that bubble teams are scoring more than they were when fans packed arenas.

8.1 Accepting the null hypothesis

So what does it look like when your hypothesis is wrong?

Let's test another thing that may have been impacted by bubble games: home court advantage. If you're the home team, but you're not at home, does it affect you? It has to, right? Your fans aren't there. Home and away are just positions on the scoreboard. It can't matter, can it?

My hypothesis is that home court is no longer an advantage, and the home team will score less relative to the away team.

First things first: We need to make a dataframe where Team is the home team. And then we'll create a differential between the home team and away team. If home court is an advantage, the differential should average out to be positive – the home team scores more than the away team.

```
homecourt <- logs %>% filter(is.na(HomeAway) == TRUE) %>% mutate(differential = TeamScore
```

Now let's test it.

```
t.test(differential ~ COVID, data=homecourt)
```

```
    Welch Two Sample t-test

data:  differential by COVID
t = 0.36892, df = 107.84, p-value = 0.7129
alternative hypothesis: true difference in means between group With Fans and group Without Fa
95 percent confidence interval:
 -2.301628  3.354268
sample estimates:
   mean in group With Fans mean in group Without Fans
                  2.174047                   1.647727
```

So again, start at the bottom. With Fans, the home team averages 2.17 more points than the away team. Without fans, they average 1.64 more.

If you are a bad sportswriter or a hack sports talk radio host, you look at this and scream "the bubble killed home court!"

But two things: first, the home team is STILL, on average, scoring more than the away team on the whole.

And two: Look at the p-value. It's .7129. Is that less than .05? No, no it is not. So that means we have to **accept the null hypothesis** that there is no difference between fans and no fans when it comes to the difference between the home team and the away team's score.

Now, does this mean that the bubble hasn't impacted the magic of home court? Not necessarily. What it's saying is that the variance between one and the other is too large to be able to say that they're different. It could just be random noise that's causing the difference, and so it's not real. More to the point, it's saying that this metric isn't capable of telling you that there's no home court in the bubble.

We're going to be analyzing these bubble games for *years* trying to find the true impact of fans.

9 Correlations and regression

Throughout sports, you will find no shortage of opinions. From people yelling at their TV screens to an entire industry of people paid to have opinions, there are no shortage of reasons why this team sucks and that player is great. They may have their reasons, but a better question is, does that reason really matter?

Can we put some numbers behind that? Can we prove it or not?

This is what we're going to start to answer. And we'll do it with correlations and regressions.

First, we need data from the 2021 college football season.

Then load the tidyverse.

```
library(tidyverse)
```

Now import the data.

```
correlations <- read_csv("data/footballlogs21.csv")
```

```
Rows: 1655 Columns: 54
-- Column specification --------------------------------------------------------
Delimiter: ","
chr   (8): HomeAway, Opponent, Result, TeamFull, TeamURL, Outcome, Team, Con...
dbl  (45): Game, PassingCmp, PassingAtt, PassingPct, PassingYds, PassingTD, ...
date  (1): Date

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.
```

To do this, we need all FBS college football teams and their season stats from last year. How much, over the course of a season, does a thing matter? That's the question you're going to answer.

In our case, we want to know how much does a team's accumulated penalties influence the number of points they score in a season? How much difference can we explain in points with penalties?

We're going to use two different methods here and they're closely related. Correlations – specifically the Pearson Correlation Coefficient – is a measure of how related two numbers are in a linear fashion. In other words – if our X value goes up one, what happens to Y? If it also goes up 1, that's a perfect correlation. X goes up 1, Y goes up 1. Every time. Correlation coefficients are a number between 0 and 1, with zero being no correlation and 1 being perfect correlation **if our data is linear**. We'll soon go over scatterplots to visually determine if our data is linear, but for now, we have a hypothesis: More penalties are bad. Penalties hurt. So if a team gets lots of them, they should have worse outcomes than teams that get few of them. That is an argument for a linear relationship between them.

But is there one?

We're going create a new dataframe called newcorrelations that takes our data that we imported and adds a column called `differential` because we don't have separate offense and defense penalties, and then we'll use correlations to see how related those two things are.

```
newcorrelations <- correlations %>%
  mutate(
    differential = TeamScore - OpponentScore,
    TotalPenalties = Penalties+DefPenalties,
    TotalPenaltyYards = PenaltyYds+DefPenaltyYds
    )
```

In R, there is a `cor` function, and it works much the same as `mean` or `median`. So we want to see if `differential` is correlated with `TotalPenaltyYards`, which is the yards of penalties a team gets in a game. We do that by referencing `differential` and `TotalPenaltyYards` and specifying we want a `pearson` correlation. The number we get back is the correlation coefficient.

```
newcorrelations %>% summarise(correlation = cor(differential, TotalPenaltyYards, method="p
```

```
# A tibble: 1 x 1
  correlation
        <dbl>
1     -0.0157
```

So on a scale of -1 to 1, where 0 means there's no relationship at all and 1 or -1 means a perfect relationship, penalty yards and whether or not the team scores more points than it give up are at -0.01566025 You could say they're 1.5 percent related toward the negative – more penalties, the lower your differential. Another way to say it? They're 98.5 percent not related.

What about the number of penalties instead of the yards?

```r
newcorrelations %>%
  summarise(correlation = cor(differential, TotalPenalties, method="pearson"))
```

```
# A tibble: 1 x 1
  correlation
        <dbl>
1     -0.0151
```

So wait, what does this all mean?

It means that when you look at every game in college football, the number of penalties and penalty yards does have a negative impact on the score difference between your team and the other team. But the relationship between penalties, penalty yards and the difference between scores is barely anything at all. Like 98+ percent plus not related.

Normally, at this point, you'd quit while you were ahead. A correlation coefficient that shows there's no relationship between two things means stop. It's pointless to go on. But let's beat a dead horse a bit for the sake of talk radio callers who want to complain about undisciplined football teams.

Enter regression. Regression is how we try to fit our data into a line that explains the relationship the best. Regressions will help us predict things as well – if we have a team that has so many penalties, what kind of point differential could we expect? So regressions are about prediction, correlations are about description. Correlations describe a relationship. Regressions help us predict what that relationship means and what it might look like in the real world. Specifically, it tells us how much of the change in a dependent variable can be explained by the independent variable.

Another thing regressions do is give us some other tools to evaluate if the relationship is real or not.

Here's an example of using linear modeling to look at penalty yards. Think of the ~ character as saying "is predicted by". The output looks like a lot, but what we need is a small part of it.

```r
fit <- lm(differential ~ TotalPenaltyYards, data = newcorrelations)
summary(fit)
```

```
Call:
lm(formula = differential ~ TotalPenaltyYards, data = newcorrelations)

Residuals:
```

```
    Min      1Q  Median      3Q     Max 
-64.472 -13.895   0.597  14.661  73.916 

Coefficients:
                   Estimate Std. Error t value Pr(>|t|)
(Intercept)        2.868781   1.549579   1.851   0.0643 .
TotalPenaltyYards -0.008619   0.013535  -0.637   0.5244
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 22.26 on 1653 degrees of freedom
Multiple R-squared:  0.0002452, Adjusted R-squared:  -0.0003596 
F-statistic: 0.4055 on 1 and 1653 DF,  p-value: 0.5244
```

There's three things we need here:

1. First we want to look at the p-value. It's at the bottom right corner of the output. In the case of Total Penalty Yards, the p-value is .5244. The threshold we're looking for here is .05. If it's less than .05, then the relationship is considered to be *statistically significant*. Significance here does not mean it's a big deal. It means it's not random. That's it. Just that. Not random. So in our case, the relationship between total penalty yards and a team's aggregate point differential are **not statistically significant**. The differences in score difference and penalty yards could be completely random. This is another sign we should just stop with this.
2. Second, we look at the Adjusted R-squared value. It's right above the p-value. Adjusted R-squared is a measure of how much of the difference between teams aggregate point values can be explained by penalty yards. Our correlation coefficient said they're 1.5 percent related to each other, but penalty yards' ability to explain the difference between teams? About -0.00036 percent. That's … not much. It's really nothing. Again, we should quit.
3. The third thing we can look at, and we only bother if the first two are meaningful, is the coefficients. In the middle, you can see the (Intercept) is 2.868781 and the TotalPenaltyYards coefficient is -0.008619 Remember high school algebra? Remember learning the equation of a line? Remember swearing that learning `y=mx+b` is stupid because you'll never need it again? Surprise. It's useful again. In this case, we could try to predict a team's score differential in a game – will they score more than they give up – by using `y=mx+b`. In this case, y is the aggregate score, m is -0.008619 and b is 2.868781 So we would multiply a teams total penalty yards by -0.008619 and then add 2.868781 to it. The result would tell you what the total aggregate score in the game would be, according to our model. Chance that you're even close with this? About .08 percent. In other words, you've got a 99.92 percent chance of being completely wrong. Did I say we should quit? Yeah.

So penalty yards are totally meaningless to the outcome of a game.

You can see the problem in a graph. On the X axis is penalty yards, on the y is aggregate score. If these elements had a strong relationship, we'd see a clear pattern moving from right to left, sloping down. On the left would be the teams with few penalties and a positive point differential. On right would be teams with high penalty yards and negative point differentials. Do you see that below?

9.1 A more predictive example

So we've **firmly** established that penalties aren't predictive. But what is?

So instead of looking at penalty yards, let's make a new metric: Net Yards. Can we predict the score differential by looking at the yards a team gained minus the yards they gave up.

```
regressions <- newcorrelations %>% mutate(NetYards = OffensiveYards - DefYards)
```

First, let's look at the correlation coefficent.

```
regressions %>%
  summarise(correlation = cor(differential, NetYards, method="pearson"))
```

```
# A tibble: 1 x 1
  correlation
        <dbl>
1       0.832
```

Answer: 83 percent. Not a perfect relationship, but very good. But how meaningful is that relationship and how predictive is it?

```r
net <- lm(differential ~ NetYards, data = regressions)
summary(net)
```

```
Call:
lm(formula = differential ~ NetYards, data = regressions)

Residuals:
    Min      1Q  Median      3Q     Max 
-36.252  -8.373  -0.004   8.660  35.396 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 0.428075   0.304630   1.405     0.16    
NetYards    0.107464   0.001762  60.978   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 12.35 on 1653 degrees of freedom
Multiple R-squared:  0.6923,	Adjusted R-squared:  0.6921 
F-statistic:  3718 on 1 and 1653 DF,  p-value: < 2.2e-16
```

First we check p-value. See that e-16? That means scientific notation. That means our number is 2.2 times 10 to the -16 power. So -.000000000000000022. That's sixteen zeros between the decimal and 22. Is that less than .05? Uh, yeah. So this is really, really, really not random. But anyone who has watched a game of football knows this is true. It makes intuitive sense.

Second, Adjusted R-squared: 0.6921. So we can predict a whopping 69 percent of the difference in the score differential by simply looking at the net yards the team has.

Third, the coefficients: In this case, our `y=mx+b` formula looks like `y = 0.107464x + 0.428075`. So if we were applying this, let's look at Nebraska's 26-20 loss to Iowa in 2020. Nebraska's net yards that game? 16. That's right – positive net yards, but still a loss. What does our model say the point differential should have been?

```r
(0.107464*16)+0.428075
```

```
[1] 2.147499
```

So by our model, Nebraska should have won by 2.14 points. Some games are closer than others. But when you can explain 69 percent of the difference, this is the kind of result you get. What would improve the model? Using more data to start. And using more inputs.

10 Multiple regression

Last chapter, we looked at correlations and linear regression to predict how one element of a game would predict the score. But we know that a single variable, in all but the rarest instances, is not going to be that predictive. We need more than one. Enter multiple regression. Multiple regression lets us add – wait for it – multiple predictors to our equation to help us get a better fit to reality.

That presents it's own problems. So let's get set up. The dataset we'll use is all college football games between 2011 and 2021.

We need the tidyverse.

```
library(tidyverse)
```

And the data.

```
logs <- read_csv("data/footballlogs1121.csv")
```

```
Rows: 17292 Columns: 55
-- Column specification --------------------------------------------------------
Delimiter: ","
chr   (8): HomeAway, Opponent, Result, TeamFull, TeamURL, Outcome, Team, Con...
dbl  (46): Game, PassingCmp, PassingAtt, PassingPct, PassingYds, PassingTD, ...
date  (1): Date

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.
```

One way to show how successful a footballl team was for a game is to show the differential between the team's score and the opponent's score. Score a lot more than the opponent = good, score a lot less than the opponent = bad. And, relatively speaking, the more the better. So let's create that differential. Let's also get our net yardage stat back. And because we'll need it later, let's add the turnover margin.

```r
logs <- logs %>% mutate(
  Differential = TeamScore - OpponentScore,
  NetYards = OffensiveYards - DefYards,
  TurnoverMargin = DefTotalTurnovers - TotalTurnovers)
```

The linear model code we used before is pretty straight forward. Its `field` is predicted by `field`. Here's a simple linear model that looks at predicting a team's point differential by looking at their net yards.

```r
yards <- lm(Differential ~ NetYards, data=logs)
summary(yards)
```

```
Call:
lm(formula = Differential ~ NetYards, data = logs)

Residuals:
    Min      1Q  Median      3Q     Max 
-52.107  -8.719   0.008   8.740  64.517 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)    
(Intercept) 0.4860213  0.0995105   4.884 1.05e-06 ***
NetYards    0.1048648  0.0005579 187.950  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 13.05 on 17290 degrees of freedom
Multiple R-squared:  0.6714,	Adjusted R-squared:  0.6714 
F-statistic: 3.533e+04 on 1 and 17290 DF,  p-value: < 2.2e-16
```

Remember: There's a lot here, but only some of it we care about. What is the Adjusted R-squared value? What's the p-value and is it less than .05? In this case, we can predict 67 percent of the difference in differential with the net yardage in the game.

To add more predictors to this mix, we merely add them. But it's not that simple, as you'll see in a moment. So first, let's look at adding how well the other team shot to our prediction model:

```r
model1 <- lm(Differential ~ NetYards + TurnoverMargin, data=logs)
summary(model1)
```

```
Call:
lm(formula = Differential ~ NetYards + TurnoverMargin, data = logs)

Residuals:
    Min      1Q  Median      3Q     Max 
-40.235  -7.027  -0.013   6.979  40.690 

Coefficients:
                Estimate Std. Error t value Pr(>|t|)    
(Intercept)    0.4575074  0.0804069    5.69 1.29e-08 ***
NetYards       0.0968307  0.0004585  211.17  < 2e-16 ***
TurnoverMargin 4.1683996  0.0434749   95.88  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 10.55 on 17289 degrees of freedom
Multiple R-squared:  0.7855,	Adjusted R-squared:  0.7854 
F-statistic: 3.165e+04 on 2 and 17289 DF,  p-value: < 2.2e-16
```

First things first: What is the adjusted R-squared?

Second: what is the p-value and is it less than .05?

Third: Compare the residual standard error. We went from 13.05 to 10.55. The meaning of this is both really opaque and also simple – by adding data, we reduced the amount of error in our model. Residual standard error is the total distance between what our model would predict and what we actually have in the data. So lots of residual error means the distance between reality and our model is wider. So the width of our predictive range in this example shrank while we improved the amount of the difference we could predict. That's good, and not always going to be the case.

One of the more difficult things to understand about multiple regression is the issue of multicollinearity. What that means is that there is significant correlation overlap between two variables – the two are related to each other as well as to the target output – and all you are doing by adding both of them is adding error with no real value to the R-squared. In pure statistics, we don't want any multicollinearity at all. Violating that assumption limits the applicability of what you are doing. So if we have some multicollinearity, it limits our scope of application to college football We can't say this will work for every football league and level everywhere. What we need to do is see how correlated each value is to each other and throw out ones that are highly co-correlated.

So to find those, we have to create a correlation matrix that shows us how each value is correlated to our outcome variable, but also with each other. We can do that in the `Hmisc` library. We install that in the console with `install.packages("Hmisc")`

```
library(Hmisc)
```

We can pass in every numeric value to the Hmisc library and get a correlation matrix out of it, but since we have a large number of values – and many of them character values – we should strip that down and reorder them. So that's what I'm doing here. I'm saying give me differential first, and then columns 9-24, and then 26-41. Why the skip? There's a blank column in the middle of the data – a remnant of the scraper I used.

```
simplelogs <- logs %>% select_if(is.numeric) %>% select(-Game) %>% select(Differential, Ne
```

Before we proceed, what we're looking to do is follow the Differential column down, looking for correlation values near 1 or -1. Correlations go from -1, meaning perfect negative correlation, to 0, meaning no correlation, to 1, meaning perfect positive correlation. So we're looking for numbers near 1 or -1 for their predictive value. BUT: We then need to see if that value is also highly correlated with something else. If it is, we have a decision to make.

We get our correlation matrix like this:

```
cormatrix <- rcorr(as.matrix(simplelogs))

cormatrix$r
```

```
                  Differential      NetYards TurnoverMargin  PassingCmp
Differential       1.000000000  0.8193828921   0.4817945770  0.03189462
NetYards           0.819382892  1.0000000000   0.1827366085  0.19252541
TurnoverMargin     0.481794577  0.1827366085   1.0000000000 -0.09175875
PassingCmp         0.031894624  0.1925254141  -0.0917587501  1.00000000
PassingAtt        -0.193442062 -0.0049667607  -0.1936605946  0.88648831
PassingPct         0.421075099  0.4188826017   0.1576394821  0.50400102
PassingYds         0.227177556  0.3686946063  -0.0240542864  0.80496816
PassingTD          0.417455037  0.3783621756   0.1485348608  0.42204275
RushingAtt         0.360084341  0.4113849934   0.1948121211 -0.37237558
RushingYds         0.526916142  0.5496832583   0.1969216415 -0.32927065
RushingAvg         0.492266420  0.4883165539   0.1519580167 -0.19682596
RushingTD          0.574259900  0.4728966272   0.2595773700 -0.13523979
OffensivePlays     0.146569146  0.3819300224  -0.0090676305  0.52992916
OffensiveYards     0.590401997  0.7226189364   0.1323695404  0.39827882
OffenseAvg         0.613383438  0.6276006878   0.1647698240  0.15090319
FirstDownPass      0.180925552  0.3312579689  -0.0504852346  0.85716796
FirstDownRush      0.458727555  0.5118168604   0.1444863057 -0.24743530
FirstDownPen      -0.007005960 -0.0154904647  -0.0214868561  0.15697290
FirstDownTotal     0.467114203  0.6166218760   0.0621309761  0.48968064
```

Penalties	-0.017903020	0.0629558958	0.0220774418	0.13324395
PenaltyYds	0.017976255	0.0936503621	0.0369986640	0.12165628
Fumbles	-0.144093284	-0.0039235362	-0.4498550354	0.01770088
Interceptions	-0.344934680	-0.1810303806	-0.5692218839	0.10080226
TotalTurnovers	-0.351038748	-0.1373891488	-0.7178479005	0.08663908
TeamScore	0.782606647	0.6393768057	0.3764111701	0.18352285
OpponentScore	-0.771137655	-0.6337773800	-0.3721890594	0.13757141
DefPassingCmp	-0.065742099	-0.2227085150	0.0830207421	0.08175611
DefPassingAtt	0.146932784	-0.0372530270	0.1818797344	0.07447128
DefPassingPct	-0.417676595	-0.4175332244	-0.1545546756	0.03666536
DefPassingYds	-0.260766014	-0.3997275463	0.0161196496	0.12429306
DefPassingTD	-0.420538390	-0.3811614512	-0.1489762764	0.11354670
DefRushingAtt	-0.359935377	-0.4112852507	-0.1936519381	-0.03730603
DefRushingYds	-0.521570963	-0.5459501938	-0.1954755564	0.02040017
DefRushingAvg	-0.485805999	-0.4858354617	-0.1483786567	0.05240827
DefRushingTD	-0.559507533	-0.4627406315	-0.2559686416	0.06040706
DefPlays	-0.189246870	-0.4186160157	-0.0003833353	0.03882538
DefYards	-0.600164346	-0.7304094425	-0.1331557581	0.11544576
DefAvg	-0.619937584	-0.6395737484	-0.1627010959	0.10958795
DefFirstDownPass	-0.220163264	-0.3670968244	0.0417268437	0.08909292
DefFirstDownRush	-0.457149808	-0.5122150780	-0.1445511124	-0.01820746
DefFirstDownPen	-0.019271882	-0.0076547605	0.0153281866	0.04632885
DefFirstDownTotal	-0.488417807	-0.6333804763	-0.0671511075	0.06281285
DefPenalties	0.014623485	-0.0646154470	-0.0285450973	0.09448494
DefPenaltyYds	-0.030490097	-0.1030785810	-0.0461476661	0.10596327
DefFumbles	0.150496494	0.0056786733	0.4556786879	-0.03908465
DefInterceptions	0.329298137	0.1638159826	0.5735436517	-0.02745430
DefTotalTurnovers	0.343625559	0.1261236571	0.7238818567	-0.04583947
Season	-0.002329276	-0.0002658874	-0.0024626416	-0.02166584

	PassingAtt	PassingPct	PassingYds	PassingTD
Differential	-0.193442062	0.42107510	0.227177556	0.417455037
NetYards	-0.004966761	0.41888260	0.368694606	0.378362176
TurnoverMargin	-0.193660595	0.15763948	-0.024054286	0.148534861
PassingCmp	0.886488307	0.50400102	0.804968162	0.422042746
PassingAtt	1.000000000	0.09074926	0.686220278	0.272639292
PassingPct	0.090749256	1.00000000	0.472396660	0.395288884
PassingYds	0.686220278	0.47239666	1.000000000	0.629186441
PassingTD	0.272639292	0.39528888	0.629186441	1.000000000
RushingAtt	-0.465919608	0.02877704	-0.266608219	-0.063945811
RushingYds	-0.449195199	0.10059964	-0.200978776	0.064570560
RushingAvg	-0.303951321	0.13318824	-0.081793664	0.150467838
RushingTD	-0.291698729	0.23913489	0.011304129	-0.031201217
OffensivePlays	0.554656082	0.11707716	0.430565144	0.210542694

OffensiveYards	0.209585627	0.46010357	0.655231285	0.559573124
OffenseAvg	-0.082221057	0.47844207	0.508182385	0.525988785
FirstDownPass	0.746732132	0.46597133	0.883486913	0.522232057
FirstDownRush	-0.365598980	0.11963800	-0.163256243	0.063106189
FirstDownPen	0.196840044	-0.01381127	0.127403081	0.098810995
FirstDownTotal	0.330730783	0.42853261	0.563779868	0.454840678
Penalties	0.140105050	0.03501704	0.151175790	0.078755685
PenaltyYds	0.115616209	0.05431583	0.150611131	0.097490407
Fumbles	0.029239617	-0.02167409	0.007841798	-0.050135883
Interceptions	0.255037596	-0.23418785	0.018702145	-0.114332815
TotalTurnovers	0.209047464	-0.18857914	0.019052148	-0.117907381
TeamScore	-0.033843974	0.44742008	0.434199508	0.619316744
OpponentScore	0.269387727	-0.20415294	0.087052720	-0.022649539
DefPassingCmp	0.081485598	0.02232557	0.123262380	0.093446247
DefPassingAtt	0.056831463	0.05555851	0.142346274	0.145556240
DefPassingPct	0.075098401	-0.06938596	-0.003313337	-0.080533635
DefPassingYds	0.153613111	-0.02272677	0.149956095	0.075617981
DefPassingTD	0.171507531	-0.08180103	0.097791600	0.035225215
DefRushingAtt	0.081269071	-0.21235606	-0.035782304	-0.053997404
DefRushingYds	0.120955351	-0.17661659	-0.008032971	-0.072979224
DefRushingAvg	0.117202023	-0.11076568	0.011275532	-0.065824127
DefRushingTD	0.152662666	-0.15166734	0.025210808	-0.036809023
DefPlays	0.131518193	-0.14240113	0.107196169	0.093450443
DefYards	0.214262231	-0.15039186	0.114874460	0.006392178
DefAvg	0.175971717	-0.10417445	0.063801224	-0.059323525
DefFirstDownPass	0.106872461	-0.01142328	0.118827164	0.066226915
DefFirstDownRush	0.063486766	-0.15235069	-0.032599487	-0.070851943
DefFirstDownPen	0.044050302	0.02493714	0.076353807	0.069465654
DefFirstDownTotal	0.132485214	-0.10963615	0.081154910	0.014349146
DefPenalties	0.140114524	-0.04378656	0.088499677	0.083509714
DefPenaltyYds	0.160393719	-0.05548118	0.089222557	0.083371921
DefFumbles	-0.033278931	-0.01783199	-0.034307069	0.045014429
DefInterceptions	-0.065866611	0.06814516	0.008803248	0.089880441
DefTotalTurnovers	-0.070790063	0.03938345	-0.015643722	0.096341451
Season	-0.044569600	0.03441817	-0.002043618	0.005530594

	RushingAtt	RushingYds	RushingAvg	RushingTD
Differential	0.3600843413	0.526916142	0.492266420	0.574259900
NetYards	0.4113849934	0.549683258	0.488316554	0.472896627
TurnoverMargin	0.1948121211	0.196921642	0.151958017	0.259577370
PassingCmp	-0.3723755771	-0.329270655	-0.196825964	-0.135239789
PassingAtt	-0.4659196079	-0.449195199	-0.303951321	-0.291698729
PassingPct	0.0287770440	0.100599643	0.133188244	0.239134887
PassingYds	-0.2666082192	-0.200978776	-0.081793664	0.011304129

PassingTD	-0.0639458109	0.064570560	0.150467838	-0.031201217
RushingAtt	1.0000000000	0.734451160	0.359268430	0.487221774
RushingYds	0.7344511599	1.000000000	0.870643178	0.693492595
RushingAvg	0.3592684303	0.870643178	1.000000000	0.605350783
RushingTD	0.4872217737	0.693492595	0.605350783	1.000000000
OffensivePlays	0.4778214889	0.244706865	0.036088200	0.168578221
OffensiveYards	0.3503709147	0.608326684	0.605137446	0.543954967
OffenseAvg	0.1289887560	0.571949319	0.706441295	0.540829592
FirstDownPass	-0.2375102814	-0.196027729	-0.092364268	-0.002406688
FirstDownRush	0.7866599010	0.868590123	0.658177887	0.594571946
FirstDownPen	0.0008338221	-0.068987653	-0.086346687	-0.001505311
FirstDownTotal	0.4021363502	0.473908999	0.390784825	0.432732948
Penalties	-0.0295906413	-0.007742277	0.014989213	-0.030169808
PenaltyYds	-0.0044491950	0.032786220	0.054811472	0.007213099
Fumbles	0.0233874281	-0.028423220	-0.057642544	-0.057209329
Interceptions	-0.1918796053	-0.215213881	-0.173136020	-0.228300636
TotalTurnovers	-0.1276259014	-0.178854625	-0.166606579	-0.207408413
TeamScore	0.3372226409	0.568490689	0.558746580	0.695927224
OpponentScore	-0.2209965073	-0.246653066	-0.202165421	-0.190721866
DefPassingCmp	-0.0462627427	0.003976984	0.036355945	0.040521518
DefPassingAtt	0.0706050627	0.099958346	0.095927982	0.127410365
DefPassingPct	-0.2139918944	-0.177404538	-0.110892137	-0.151258656
DefPassingYds	-0.0476853142	-0.027340837	-0.007310635	0.001450072
DefPassingTD	-0.0601789437	-0.079466211	-0.071195691	-0.048242502
DefRushingAtt	-0.4089510299	-0.277883613	-0.119605853	-0.193211571
DefRushingYds	-0.2803086384	-0.228518367	-0.135774141	-0.196286215
DefRushingAvg	-0.1235463027	-0.138076523	-0.112418632	-0.152858227
DefRushingTD	-0.1903718594	-0.190803430	-0.145696279	-0.154120678
DefPlays	-0.3100509301	-0.159402466	-0.016421599	-0.053706387
DefYards	-0.2480158637	-0.192894387	-0.107406482	-0.145587083
DefAvg	-0.1169558858	-0.146495863	-0.128555148	-0.155348792
DefFirstDownPass	-0.0891812140	-0.044011457	-0.003968513	-0.009878681
DefFirstDownRush	-0.3183100274	-0.224225490	-0.104891596	-0.173320384
DefFirstDownPen	-0.0482601703	-0.003641089	0.030648349	0.012958001
DefFirstDownTotal	-0.3024445686	-0.193719536	-0.072661179	-0.127437469
DefPenalties	-0.0397578728	-0.066436558	-0.060610647	0.003208587
DefPenaltyYds	-0.0561434614	-0.090035419	-0.082525547	-0.023753105
DefFumbles	0.0748597567	0.026263167	-0.006248163	0.082361214
DefInterceptions	0.1399910698	0.118238785	0.076314786	0.152075032
DefTotalTurnovers	0.1531303753	0.105391169	0.052994374	0.167021957
Season	-0.0286039328	-0.004708464	0.018330491	-0.001251312

	OffensivePlays	OffensiveYards	OffenseAvg	FirstDownPass
Differential	0.1465691460	0.590401997	0.61338344	0.180925552

NetYards	0.3819300224	0.722618936	0.62760069	0.331257969
TurnoverMargin	-0.0090676305	0.132369540	0.16476982	-0.050485235
PassingCmp	0.5299291629	0.398278819	0.15090319	0.857167955
PassingAtt	0.5546560820	0.209585627	-0.08222106	0.746732132
PassingPct	0.1170771610	0.460103565	0.47844207	0.465971332
PassingYds	0.4305651443	0.655231285	0.50818239	0.883486913
PassingTD	0.2105426937	0.559573124	0.52598878	0.522232057
RushingAtt	0.4778214889	0.350370915	0.12898876	-0.237510281
RushingYds	0.2447068650	0.608326684	0.57194932	-0.196027729
RushingAvg	0.0360882003	0.605137446	0.70644129	-0.092364268
RushingTD	0.1685782208	0.543954967	0.54082959	-0.002406688
OffensivePlays	1.0000000000	0.537560956	0.03967020	0.518004676
OffensiveYards	0.5375609564	1.000000000	0.85280533	0.564648261
OffenseAvg	0.0396701997	0.852805326	1.00000000	0.351771450
FirstDownPass	0.5180046757	0.564648261	0.35177145	1.000000000
FirstDownRush	0.3767976651	0.537551693	0.40782894	-0.177747002
FirstDownPen	0.1962070566	0.050023611	-0.05328605	0.128819985
FirstDownTotal	0.7065134901	0.822246337	0.54463477	0.634690773
Penalties	0.1112697482	0.116514902	0.06726852	0.094623213
PenaltyYds	0.1105998934	0.147311501	0.10503342	0.094561833
Fumbles	0.0510223428	-0.015565376	-0.05062981	-0.002406073
Interceptions	0.0727604625	-0.150812253	-0.21868037	0.049907591
TotalTurnovers	0.0875247613	-0.122489744	-0.19594487	0.035597616
TeamScore	0.2835194051	0.790195841	0.76040029	0.351870113
OpponentScore	0.0596261325	-0.119678084	-0.18626714	0.075538370
DefPassingCmp	0.0373940890	0.102936400	0.09485008	0.088842935
DefPassingAtt	0.1228183894	0.192415847	0.15031629	0.096502631
DefPassingPct	-0.1267207052	-0.139518292	-0.09303100	0.007691644
DefPassingYds	0.1076645693	0.100412988	0.04634719	0.120755788
DefPassingTD	0.1136813184	0.017951152	-0.05401244	0.091108258
DefRushingAtt	-0.3038881650	-0.243285184	-0.10681048	-0.078728053
DefRushingYds	-0.1435139876	-0.182733479	-0.13540190	-0.023937131
DefRushingAvg	0.0001737604	-0.097345267	-0.12011271	0.016364020
DefRushingTD	-0.0274595797	-0.126714363	-0.14011879	0.015981874
DefPlays	-0.1609964490	-0.036073009	0.04911052	0.022099214
DefYards	-0.0205111475	-0.055679659	-0.06387431	0.079444979
DefAvg	0.0647173673	-0.061281375	-0.11871438	0.073314749
DefFirstDownPass	0.0222382198	0.062335972	0.05371647	0.091669749
DefFirstDownRush	-0.2363048466	-0.199327221	-0.09836444	-0.046765573
DefFirstDownPen	-0.0016500561	0.059055413	0.06909101	0.054021601
DefFirstDownTotal	-0.1528834681	-0.083636196	-0.01572435	0.042036284
DefPenalties	0.1017180239	0.020470653	-0.03262961	0.073496362
DefPenaltyYds	0.1064424321	0.002857766	-0.05656204	0.076997130

DefFumbles	0.0373577815	-0.007543036	-0.02658401	-0.044389124
DefInterceptions	0.0662533371	0.098313987	0.07964313	-0.011275396
DefTotalTurnovers	0.0737213597	0.068598981	0.04230882	-0.037182652
Season	-0.0711474936	-0.005286773	0.03920997	-0.017213258

	FirstDownRush	FirstDownPen	FirstDownTotal	Penalties
Differential	0.4587275549	-0.0070059601	0.4671142034	-0.017903020
NetYards	0.5118168604	-0.0154904647	0.6166218760	0.062955896
TurnoverMargin	0.1444863057	-0.0214868561	0.0621309761	0.022077442
PassingCmp	-0.2474352953	0.1569728980	0.4896806443	0.133243949
PassingAtt	-0.3655989801	0.1968400438	0.3307307829	0.140105050
PassingPct	0.1196380018	-0.0138112671	0.4285326060	0.035017037
PassingYds	-0.1632562426	0.1274030815	0.5637798680	0.151175790
PassingTD	0.0631061892	0.0988109951	0.4548406779	0.078755685
RushingAtt	0.7866599010	0.0008338221	0.4021363502	-0.029590641
RushingYds	0.8685901225	-0.0689876530	0.4739089986	-0.007742277
RushingAvg	0.6581778868	-0.0863466872	0.3907848252	0.014989213
RushingTD	0.5945719459	-0.0015053106	0.4327329482	-0.030169808
OffensivePlays	0.3767976651	0.1962070566	0.7065134901	0.111269748
OffensiveYards	0.5375516932	0.0500236111	0.8222463375	0.116514902
OffenseAvg	0.4078289419	-0.0532860472	0.5446347746	0.067268520
FirstDownPass	-0.1777470018	0.1288199853	0.6346907730	0.094623213
FirstDownRush	1.0000000000	-0.0644575126	0.5781427575	-0.038754051
FirstDownPen	-0.0644575126	1.0000000000	0.2736798978	0.129344343
FirstDownTotal	0.5781427575	0.2736798978	1.0000000000	0.070788193
Penalties	-0.0387540514	0.1293443431	0.0707881929	1.000000000
PenaltyYds	-0.0033090010	0.1300378785	0.0970509324	0.902829237
Fumbles	-0.0138953108	-0.0017378205	-0.0113985746	-0.003266748
Interceptions	-0.1808243429	0.0270078656	-0.0877659342	0.028163149
TotalTurnovers	-0.1437506809	0.0189792176	-0.0728124801	0.018840723
TeamScore	0.4890655439	0.0680319223	0.6340050291	0.041498010
OpponentScore	-0.2207230222	0.0805886349	-0.0856684865	0.070576634
DefPassingCmp	-0.0344067943	0.0486347366	0.0520405278	0.099823324
DefPassingAtt	0.0431212196	0.0442460281	0.1126625808	0.144271744
DefPassingPct	-0.1542580711	0.0328592747	-0.0983380215	-0.036100979
DefPassingYds	-0.0510138720	0.0824907680	0.0711591224	0.094302865
DefPassingTD	-0.0748223833	0.0784393803	0.0314529663	0.087646045
DefRushingAtt	-0.3160553041	-0.0438033149	-0.3007572661	-0.040971492
DefRushingYds	-0.2233740347	0.0070967276	-0.1813288384	-0.069317517
DefRushingAvg	-0.1068187536	0.0424931339	-0.0583656854	-0.062420100
DefRushingTD	-0.1668375949	0.0302555426	-0.1034632197	0.004355500
DefPlays	-0.2509114803	0.0029767857	-0.1681078047	0.104277549
DefYards	-0.2081311198	0.0718022160	-0.0782111605	0.024191887
DefAvg	-0.1092241576	0.0840351763	-0.0070382665	-0.028334770

DefFirstDownPass	-0.0665512008	0.0591695441	0.0286098677	0.078247923
DefFirstDownRush	-0.2284698051	-0.0135859963	-0.2136267203	-0.078203416
DefFirstDownPen	-0.0235836018	0.0676167199	0.0334158092	0.547778561
DefFirstDownTotal	-0.2243099072	0.0441982586	-0.1245484452	0.127662433
DefPenalties	-0.0763456747	0.5443248528	0.1255374004	0.184220592
DefPenaltyYds	-0.0981877995	0.6607288539	0.1391194637	0.187784680
DefFumbles	0.0036192814	-0.0222634761	-0.0344597905	0.014221867
DefInterceptions	0.0837185725	0.0031988196	0.0526327383	0.055046719
DefTotalTurnovers	0.0649203964	-0.0120310687	0.0170179179	0.050357718
Season	-0.0003574311	0.0658170522	-0.0001756986	0.032140649

	PenaltyYds	Fumbles	Interceptions	TotalTurnovers
Differential	0.017976255	-0.144093284	-0.344934680	-0.351038748
NetYards	0.093650362	-0.003923536	-0.181030381	-0.137389149
TurnoverMargin	0.036998664	-0.449855035	-0.569221884	-0.717847900
PassingCmp	0.121656283	0.017700875	0.100802257	0.086639084
PassingAtt	0.115616209	0.029239617	0.255037596	0.209047464
PassingPct	0.054315829	-0.021674094	-0.234187853	-0.188579144
PassingYds	0.150611131	0.007841798	0.018702145	0.019052148
PassingTD	0.097490407	-0.050135883	-0.114332815	-0.117907381
RushingAtt	-0.004449195	0.023387428	-0.191879605	-0.127625901
RushingYds	0.032786220	-0.028423220	-0.215213881	-0.178854625
RushingAvg	0.054811472	-0.057642544	-0.173136020	-0.166606579
RushingTD	0.007213099	-0.057209329	-0.228300636	-0.207408413
OffensivePlays	0.110599893	0.051022343	0.072760463	0.087524761
OffensiveYards	0.147311501	-0.015565376	-0.150812253	-0.122489744
OffenseAvg	0.105033416	-0.050629809	-0.218680373	-0.195944865
FirstDownPass	0.094561833	-0.002406073	0.049907591	0.035597616
FirstDownRush	-0.003309001	-0.013895311	-0.180824343	-0.143750681
FirstDownPen	0.130037879	-0.001737821	0.027007866	0.018979218
FirstDownTotal	0.097050932	-0.011398575	-0.087765934	-0.072812480
Penalties	0.902829237	-0.003266748	0.028163149	0.018840723
PenaltyYds	1.000000000	-0.012630902	0.016778227	0.004243593
Fumbles	-0.012630902	1.000000000	0.018972477	0.667472431
Interceptions	0.016778227	0.018972477	1.000000000	0.757164086
TotalTurnovers	0.004243593	0.667472431	0.757164086	1.000000000
TeamScore	0.080950037	-0.092240194	-0.274800918	-0.264927507
OpponentScore	0.054539982	0.132110601	0.261029056	0.280719683
DefPassingCmp	0.110025970	-0.037507728	-0.016726110	-0.036962467
DefPassingAtt	0.160577291	-0.028688053	-0.056806282	-0.061050652
DefPassingPct	-0.043625494	-0.022528499	0.077221775	0.042786712
DefPassingYds	0.094811538	-0.033907392	0.023092463	-0.004954594
DefPassingTD	0.090155476	0.040095597	0.099723139	0.100466810
DefRushingAtt	-0.055326346	0.073573856	0.143056836	0.154613148

DefRushingYds	-0.088465239	0.023567107	0.125528901	0.108887368
DefRushingAvg	-0.079535197	-0.009735953	0.084689469	0.056706358
DefRushingTD	-0.016717598	0.077755272	0.159387853	0.169507873
DefPlays	0.107034241	0.040011704	0.076787249	0.083330069
DefYards	0.010285882	-0.009712671	0.112468308	0.077417161
DefAvg	-0.047198014	-0.031188217	0.094090856	0.049691577
DefFirstDownPass	0.080864763	-0.044846058	0.002583476	-0.027375736
DefFirstDownRush	-0.095197482	0.002481689	0.090616330	0.069109586
DefFirstDownPen	0.659406009	-0.019707902	0.008669650	-0.006419123
DefFirstDownTotal	0.142990772	-0.034572761	0.066996246	0.027308850
DefPenalties	0.187837432	0.019046107	0.054887750	0.053322298
DefPenaltyYds	0.192157556	0.020885214	0.067362991	0.063815032
DefFumbles	0.015171119	0.028222683	-0.010522821	0.010602007
DefInterceptions	0.063522621	-0.003324980	-0.080038973	-0.061782861
DefTotalTurnovers	0.057307989	0.015797080	-0.066642101	-0.039312041
Season	0.041380515	-0.091613051	-0.046325016	-0.094356100

	TeamScore	OpponentScore	DefPassingCmp	DefPassingAtt
Differential	0.782606647	-0.7711376548	-0.065742099	0.146932784
NetYards	0.639376806	-0.6337773800	-0.222708515	-0.037253027
TurnoverMargin	0.376411170	-0.3721890594	0.083020742	0.181879734
PassingCmp	0.183522847	0.1375714078	0.081756108	0.074471280
PassingAtt	-0.033843974	0.2693877275	0.081485598	0.056831463
PassingPct	0.447420085	-0.2041529357	0.022325574	0.055558508
PassingYds	0.434199508	0.0870527203	0.123262380	0.142346274
PassingTD	0.619316744	-0.0226495389	0.093446247	0.145556240
RushingAtt	0.337222641	-0.2209965073	-0.046262743	0.070605063
RushingYds	0.568490689	-0.2466530664	0.003976984	0.099958346
RushingAvg	0.558746580	-0.2021654206	0.036355945	0.095927982
RushingTD	0.695927224	-0.1907218665	0.040521518	0.127410365
OffensivePlays	0.283519405	0.0596261325	0.037394089	0.122818389
OffensiveYards	0.790195841	-0.1196780843	0.102936400	0.192415847
OffenseAvg	0.760400294	-0.1862671392	0.094850078	0.150316289
FirstDownPass	0.351870113	0.0755383695	0.088842935	0.096502631
FirstDownRush	0.489065544	-0.2207230222	-0.034406794	0.043121220
FirstDownPen	0.068031922	0.0805886349	0.048634737	0.044246028
FirstDownTotal	0.634005029	-0.0856684865	0.052040528	0.112662581
Penalties	0.041498010	0.0705766343	0.099823324	0.144271744
PenaltyYds	0.080950037	0.0545399820	0.110025970	0.160577291
Fumbles	-0.092240194	0.1321106008	-0.037507728	-0.028688053
Interceptions	-0.274800918	0.2610290560	-0.016726110	-0.056806282
TotalTurnovers	-0.264927507	0.2807196825	-0.036962467	-0.061050652
TeamScore	1.000000000	-0.2071608155	0.109054508	0.232810162
OpponentScore	-0.207160815	1.0000000000	0.214849738	0.007192626

DefPassingCmp	0.109054508	0.2148497376	1.000000000	0.889169550
DefPassingAtt	0.232810162	0.0071926256	0.889169550	1.000000000
DefPassingPct	-0.203561294	0.4482036746	0.524263816	0.121178071
DefPassingYds	0.052809184	0.4638127943	0.809833235	0.696006663
DefPassingTD	-0.029821238	0.6303918795	0.431879339	0.289172942
DefRushingAtt	-0.219556636	0.3411033576	-0.360875046	-0.455926255
DefRushingYds	-0.247861743	0.5661708581	-0.306845464	-0.424151942
DefRushingAvg	-0.204993156	0.5538078889	-0.174207785	-0.276195127
DefRushingTD	-0.185557564	0.6895100944	-0.110193768	-0.260234894
DefPlays	0.025805738	0.3238005855	0.542068467	0.563145905
DefYards	-0.142747354	0.7971871550	0.423393858	0.243931898
DefAvg	-0.203285692	0.7663467449	0.188706076	-0.035596062
DefFirstDownPass	0.038455283	0.3853239116	0.857692613	0.751320448
DefFirstDownRush	-0.223859546	0.4894785676	-0.221569780	-0.337360680
DefFirstDownPen	0.061182898	0.0928602425	0.166804480	0.202043383
DefFirstDownTotal	-0.115520917	0.6494188757	0.508437577	0.356660496
DefPenalties	0.066070970	0.0445916607	0.129895425	0.138446185
DefPenaltyYds	0.042702194	0.0915892334	0.122383691	0.117249826
DefFumbles	0.141278302	-0.0920205611	0.016440817	0.027710829
DefInterceptions	0.249097965	-0.2627429407	0.096162025	0.244286780
DefTotalTurnovers	0.277708094	-0.2559984593	0.082526971	0.200544151
Season	-0.002772666	0.0008248417	-0.018540089	-0.042521400

	DefPassingPct	DefPassingYds	DefPassingTD	DefRushingAtt
Differential	-0.417676595	-0.260766014	-0.420538390	-0.3599353770
NetYards	-0.417533224	-0.399727546	-0.381161451	-0.4112852507
TurnoverMargin	-0.154554676	0.016119650	-0.148976276	-0.1936519381
PassingCmp	0.036665358	0.124293059	0.113546697	-0.0373060319
PassingAtt	0.075098401	0.153613111	0.171507531	0.0812690709
PassingPct	-0.069385960	-0.022726766	-0.081801027	-0.2123560581
PassingYds	-0.003313337	0.149956095	0.097791600	-0.0357823042
PassingTD	-0.080533635	0.075617981	0.035225215	-0.0539974042
RushingAtt	-0.213991894	-0.047685314	-0.060178944	-0.4089510299
RushingYds	-0.177404538	-0.027340837	-0.079466211	-0.2778836134
RushingAvg	-0.110892137	-0.007310635	-0.071195691	-0.1196058529
RushingTD	-0.151258656	0.001450072	-0.048242502	-0.1932115706
OffensivePlays	-0.126720705	0.107664569	0.113681318	-0.3038881650
OffensiveYards	-0.139518292	0.100412988	0.017951152	-0.2432851839
OffenseAvg	-0.093030997	0.046347193	-0.054012439	-0.1068104821
FirstDownPass	0.007691644	0.120755788	0.091108258	-0.0787280528
FirstDownRush	-0.154258071	-0.051013872	-0.074822383	-0.3160553041
FirstDownPen	0.032859275	0.082490768	0.078439380	-0.0438033149
FirstDownTotal	-0.098338021	0.071159122	0.031452966	-0.3007572661
Penalties	-0.036100979	0.094302865	0.087646045	-0.0409714917

PenaltyYds	-0.043625494	0.094811538	0.090155476	-0.0553263459
Fumbles	-0.022528499	-0.033907392	0.040095597	0.0735738560
Interceptions	0.077221775	0.023092463	0.099723139	0.1430568358
TotalTurnovers	0.042786712	-0.004954594	0.100466810	0.1546131484
TeamScore	-0.203561294	0.052809184	-0.029821238	-0.2195566362
OpponentScore	0.448203675	0.463812794	0.630391880	0.3411033576
DefPassingCmp	0.524263816	0.809833235	0.431879339	-0.3608750459
DefPassingAtt	0.121178071	0.696006663	0.289172942	-0.4559262549
DefPassingPct	1.000000000	0.485640919	0.396712529	0.0270470082
DefPassingYds	0.485640919	1.000000000	0.635168968	-0.2450718636
DefPassingTD	0.396712529	0.635168968	1.000000000	-0.0519968871
DefRushingAtt	0.027047008	-0.245071864	-0.051996887	1.0000000000
DefRushingYds	0.098898952	-0.173227043	0.075275024	0.7372773767
DefRushingAvg	0.131138863	-0.058336875	0.155945194	0.3685807935
DefRushingTD	0.235599068	0.037077048	-0.017146078	0.4850629636
DefPlays	0.144680188	0.459040923	0.236980179	0.4787196637
DefYards	0.465276672	0.676589899	0.568293505	0.3536814657
DefAvg	0.482353624	0.536790215	0.535776930	0.1443238711
DefFirstDownPass	0.480900639	0.888664032	0.533631013	-0.2171565936
DefFirstDownRush	0.121155689	-0.131706218	0.076073487	0.7867025315
DefFirstDownPen	0.003820705	0.140363190	0.112734839	0.0035156804
DefFirstDownTotal	0.437970783	0.588446490	0.469188344	0.4006276517
DefPenalties	0.034805644	0.148218718	0.078087345	-0.0289769270
DefPenaltyYds	0.057035973	0.153240799	0.101887901	-0.0002624621
DefFumbles	-0.021573479	0.006122048	-0.051995314	0.0241542687
DefInterceptions	-0.221344974	0.019078257	-0.107800922	-0.1877947529
DefTotalTurnovers	-0.179426252	0.018225936	-0.114258347	-0.1247212469
Season	0.039348192	-0.001115403	0.009035752	-0.0302753145

	DefRushingYds	DefRushingAvg	DefRushingTD	DefPlays
Differential	-0.521570963	-0.4858059992	-0.55950753	-0.1892468699
NetYards	-0.545950194	-0.4858354617	-0.46274063	-0.4186160157
TurnoverMargin	-0.195475556	-0.1483786567	-0.25596864	-0.0003833353
PassingCmp	0.020400167	0.0524082686	0.06040706	0.0388253781
PassingAtt	0.120955351	0.1172020227	0.15266267	0.1315181930
PassingPct	-0.176616590	-0.1107656838	-0.15166734	-0.1424011276
PassingYds	-0.008032971	0.0112755317	0.02521081	0.1071961685
PassingTD	-0.072979224	-0.0658241267	-0.03680902	0.0934504428
RushingAtt	-0.280308638	-0.1235463027	-0.19037186	-0.3100509301
RushingYds	-0.228518367	-0.1380765232	-0.19080343	-0.1594024655
RushingAvg	-0.135774141	-0.1124186325	-0.14569628	-0.0164215986
RushingTD	-0.196286215	-0.1528582274	-0.15412068	-0.0537063869
OffensivePlays	-0.143513988	0.0001737604	-0.02745958	-0.1609964490
OffensiveYards	-0.182733479	-0.0973452671	-0.12671436	-0.0360730089

OffenseAvg	-0.135401903	-0.1201127138	-0.14011879	0.0491105176
FirstDownPass	-0.023937131	0.0163640197	0.01598187	0.0220992137
FirstDownRush	-0.223374035	-0.1068187536	-0.16683759	-0.2509114803
FirstDownPen	0.007096728	0.0424931339	0.03025554	0.0029767857
FirstDownTotal	-0.181328838	-0.0583656854	-0.10346322	-0.1681078047
Penalties	-0.069317517	-0.0624201004	0.00435550	0.1042775488
PenaltyYds	-0.088465239	-0.0795351966	-0.01671760	0.1070342410
Fumbles	0.023567107	-0.0097359530	0.07775527	0.0400117044
Interceptions	0.125528901	0.0846894689	0.15938785	0.0767872489
TotalTurnovers	0.108887368	0.0567063584	0.16950787	0.0833300691
TeamScore	-0.247861743	-0.2049931560	-0.18555756	0.0258057383
OpponentScore	0.566170858	0.5538078889	0.68951009	0.3238005855
DefPassingCmp	-0.306845464	-0.1742077853	-0.11019377	0.5420684671
DefPassingAtt	-0.424151942	-0.2761951274	-0.26023489	0.5631459054
DefPassingPct	0.098898952	0.1311388631	0.23559907	0.1446801883
DefPassingYds	-0.173227043	-0.0583368755	0.03707705	0.4590409235
DefPassingTD	0.075275024	0.1559451940	-0.01714608	0.2369801788
DefRushingAtt	0.737277377	0.3685807935	0.48506296	0.4787196637
DefRushingYds	1.000000000	0.8728276975	0.68653666	0.2661328663
DefRushingAvg	0.872827698	1.0000000000	0.59912628	0.0697715849
DefRushingTD	0.686536655	0.5991262808	1.00000000	0.1936563469
DefPlays	0.266132866	0.0697715849	0.19365635	1.0000000000
DefYards	0.608024020	0.6056805614	0.54318625	0.5690131366
DefAvg	0.575893588	0.7029090219	0.53951828	0.0989574665
DefFirstDownPass	-0.166549093	-0.0651290247	0.02530950	0.5395244090
DefFirstDownRush	0.868130797	0.6622995127	0.58676834	0.3976390150
DefFirstDownPen	-0.058501149	-0.0724607031	0.01421475	0.2025721800
DefFirstDownTotal	0.478659827	0.4023743432	0.43690670	0.7238038053
DefPenalties	-0.007920018	0.0148168905	-0.02579004	0.1096674930
DefPenaltyYds	0.037696089	0.0593782862	0.01579238	0.1154187184
DefFumbles	-0.027539861	-0.0548817072	-0.05382363	0.0497622508
DefInterceptions	-0.207119041	-0.1621560599	-0.22012697	0.0666170352
DefTotalTurnovers	-0.172652188	-0.1567592350	-0.19940138	0.0820286905
Season	-0.005270541	0.0171753556	0.00203257	-0.0700555324

	DefYards	DefAvg	DefFirstDownPass	DefFirstDownRush
Differential	-0.600164346	-0.619937584	-0.220163264	-0.4571498075
NetYards	-0.730409443	-0.639573748	-0.367096824	-0.5122150780
TurnoverMargin	-0.133155758	-0.162701096	0.041726844	-0.1445511124
PassingCmp	0.115445761	0.109587949	0.089092921	-0.0182074572
PassingAtt	0.214262231	0.175971717	0.106872461	0.0634867657
PassingPct	-0.150391858	-0.104174452	-0.011423276	-0.1523506906
PassingYds	0.114874460	0.063801224	0.118827164	-0.0325994868
PassingTD	0.006392178	-0.059323525	0.066226915	-0.0708519426

RushingAtt	-0.248015864	-0.116955886	-0.089181214	-0.3183100274
RushingYds	-0.192894387	-0.146495863	-0.044011457	-0.2242254902
RushingAvg	-0.107406482	-0.128555148	-0.003968513	-0.1048915955
RushingTD	-0.145587083	-0.155348792	-0.009878681	-0.1733203836
OffensivePlays	-0.020511147	0.064717367	0.022238220	-0.2363048466
OffensiveYards	-0.055679659	-0.061281375	0.062335972	-0.1993272215
OffenseAvg	-0.063874308	-0.118714381	0.053716468	-0.0983644383
FirstDownPass	0.079444979	0.073314749	0.091669749	-0.0467655734
FirstDownRush	-0.208131120	-0.109224158	-0.066551201	-0.2284698051
FirstDownPen	0.071802216	0.084035176	0.059169544	-0.0135859963
FirstDownTotal	-0.078211160	-0.007038267	0.028609868	-0.2136267203
Penalties	0.024191887	-0.028334770	0.078247923	-0.0782034163
PenaltyYds	0.010285882	-0.047198014	0.080864763	-0.0951974818
Fumbles	-0.009712671	-0.031188217	-0.044846058	0.0024816890
Interceptions	0.112468308	0.094090856	0.002583476	0.0906163296
TotalTurnovers	0.077417161	0.049691577	-0.027375736	0.0691095861
TeamScore	-0.142747354	-0.203285692	0.038455283	-0.2238595459
OpponentScore	0.797187155	0.766346745	0.385323912	0.4894785676
DefPassingCmp	0.423393858	0.188706076	0.857692613	-0.2215697799
DefPassingAtt	0.243931898	-0.035596062	0.751320448	-0.3373606803
DefPassingPct	0.465276672	0.482353624	0.480900639	0.1211556887
DefPassingYds	0.676589899	0.536790215	0.888664032	-0.1317062177
DefPassingTD	0.568293505	0.535776930	0.533631013	0.0760734873
DefRushingAtt	0.353681466	0.144323871	-0.217156594	0.7867025315
DefRushingYds	0.608024020	0.575893588	-0.166549093	0.8681307972
DefRushingAvg	0.605680561	0.702909022	-0.065129025	0.6622995127
DefRushingTD	0.543186252	0.539518281	0.025309501	0.5867683446
DefPlays	0.569013137	0.098957466	0.539524409	0.3976390150
DefYards	1.000000000	0.863432564	0.591834233	0.5429004217
DefAvg	0.863432564	1.000000000	0.390449700	0.4190103381
DefFirstDownPass	0.591834233	0.390449700	1.000000000	-0.1437868206
DefFirstDownRush	0.542900422	0.419010338	-0.143786821	1.0000000000
DefFirstDownPen	0.069408365	-0.030643088	0.139316480	-0.0530593626
DefFirstDownTotal	0.832226265	0.573548214	0.656129992	0.5834812190
DefPenalties	0.113558402	0.068145987	0.092600853	-0.0386169884
DefPenaltyYds	0.151712216	0.112328207	0.098773293	0.0015978502
DefFumbles	-0.015655527	-0.049744979	-0.001905308	-0.0130664543
DefInterceptions	-0.139476214	-0.203408513	0.045479791	-0.1746156397
DefTotalTurnovers	-0.114393675	-0.184266567	0.032759858	-0.1389816404
Season	-0.004839722	0.039367726	-0.016589707	-0.0009425108

	DefFirstDownPen	DefFirstDownTotal	DefPenalties	DefPenaltyYds
Differential	-0.019271882	-0.488417807	0.014623485	-0.0304900973
NetYards	-0.007654761	-0.633380476	-0.064615447	-0.1030785810

TurnoverMargin	0.015328187	-0.067151108	-0.028545097	-0.0461476661
PassingCmp	0.046328848	0.062812851	0.094484942	0.1059632679
PassingAtt	0.044050302	0.132485214	0.140114524	0.1603937193
PassingPct	0.024937144	-0.109636154	-0.043786555	-0.0554811829
PassingYds	0.076353807	0.081154910	0.088499677	0.0892225570
PassingTD	0.069465654	0.014349146	0.083509714	0.0833719205
RushingAtt	-0.048260170	-0.302444569	-0.039757873	-0.0561434614
RushingYds	-0.003641089	-0.193719536	-0.066436558	-0.0900354185
RushingAvg	0.030648349	-0.072661179	-0.060610647	-0.0825255475
RushingTD	0.012958001	-0.127437469	0.003208587	-0.0237531054
OffensivePlays	-0.001650056	-0.152883468	0.101718024	0.1064424321
OffensiveYards	0.059055413	-0.083636196	0.020470653	0.0028577656
OffenseAvg	0.069091014	-0.015724349	-0.032629606	-0.0565620426
FirstDownPass	0.054021601	0.042036284	0.073496362	0.0769971298
FirstDownRush	-0.023583602	-0.224309907	-0.076345675	-0.0981877995
FirstDownPen	0.067616720	0.044198259	0.544324853	0.6607288539
FirstDownTotal	0.033415809	-0.124548445	0.125537400	0.1391194637
Penalties	0.547778561	0.127662433	0.184220592	0.1877846801
PenaltyYds	0.659406009	0.142990772	0.187837432	0.1921575565
Fumbles	-0.019707902	-0.034572761	0.019046107	0.0208852137
Interceptions	0.008669650	0.066996246	0.054887750	0.0673629913
TotalTurnovers	-0.006419123	0.027308850	0.053322298	0.0638150323
TeamScore	0.061182898	-0.115520917	0.066070970	0.0427021938
OpponentScore	0.092860242	0.649418876	0.044591661	0.0915892334
DefPassingCmp	0.166804480	0.508437577	0.129895425	0.1223836907
DefPassingAtt	0.202043383	0.356660496	0.138446185	0.1172498257
DefPassingPct	0.003820705	0.437970783	0.034805644	0.0570359731
DefPassingYds	0.140363190	0.588446490	0.148218718	0.1532407989
DefPassingTD	0.112734839	0.469188344	0.078087345	0.1018879012
DefRushingAtt	0.003515680	0.400627652	-0.028976927	-0.0002624621
DefRushingYds	-0.058501149	0.478659827	-0.007920018	0.0376960885
DefRushingAvg	-0.072460703	0.402374343	0.014816890	0.0593782862
DefRushingTD	0.014214751	0.436906703	-0.025790035	0.0157923804
DefPlays	0.202572180	0.723803805	0.109667493	0.1154187184
DefYards	0.069408365	0.832226265	0.113558402	0.1517122162
DefAvg	-0.030643088	0.573548214	0.068145987	0.1123282066
DefFirstDownPass	0.139316480	0.656129992	0.092600853	0.0987732932
DefFirstDownRush	-0.053059363	0.583481219	-0.038616988	0.0015978502
DefFirstDownPen	1.000000000	0.286129797	0.128239445	0.1301074980
DefFirstDownTotal	0.286129797	1.000000000	0.068601081	0.1022128964
DefPenalties	0.128239445	0.068601081	1.000000000	0.9002742132
DefPenaltyYds	0.130107498	0.102212896	0.900274213	1.0000000000
DefFumbles	-0.003582233	-0.010683524	-0.006850053	-0.0161501279

DefInterceptions	0.024027115	-0.083477752	0.021819923	0.0099909789
DefTotalTurnovers	0.015638654	-0.069316535	0.011872019	-0.0029940736
Season	0.068661759	0.000468085	0.029931261	0.0388464539

	DefFumbles	DefInterceptions	DefTotalTurnovers	Season
Differential	0.150496494	0.329298137	0.343625559	-0.0023292757
NetYards	0.005678673	0.163815983	0.126123657	-0.0002658874
TurnoverMargin	0.455678688	0.573543652	0.723881857	-0.0024626416
PassingCmp	-0.039084655	-0.027454297	-0.045839474	-0.0216658388
PassingAtt	-0.033278931	-0.065866611	-0.070790063	-0.0445696003
PassingPct	-0.017831990	0.068145163	0.039383451	0.0344181671
PassingYds	-0.034307069	0.008803248	-0.015643722	-0.0020436177
PassingTD	0.045014429	0.089880441	0.096341451	0.0055305937
RushingAtt	0.074859757	0.139991070	0.153130375	-0.0286039328
RushingYds	0.026263167	0.118238785	0.105391169	-0.0047084641
RushingAvg	-0.006248163	0.076314786	0.052994374	0.0183304913
RushingTD	0.082361214	0.152075032	0.167021957	-0.0012513115
OffensivePlays	0.037357782	0.066253337	0.073721360	-0.0711474936
OffensiveYards	-0.007543036	0.098313987	0.068598981	-0.0052867729
OffenseAvg	-0.026584005	0.079643130	0.042308820	0.0392099715
FirstDownPass	-0.044389124	-0.011275396	-0.037182652	-0.0172132576
FirstDownRush	0.003619281	0.083718572	0.064920396	-0.0003574311
FirstDownPen	-0.022263476	0.003198820	-0.012031069	0.0658170522
FirstDownTotal	-0.034459790	0.052632738	0.017017918	-0.0001756986
Penalties	0.014221867	0.055046719	0.050357718	0.0321406487
PenaltyYds	0.015171119	0.063522621	0.057307989	0.0413805146
Fumbles	0.028222683	-0.003324980	0.015797080	-0.0916130514
Interceptions	-0.010522821	-0.080038973	-0.066642101	-0.0463250158
TotalTurnovers	0.010602007	-0.061782861	-0.039312041	-0.0943560996
TeamScore	0.141278302	0.249097965	0.277708094	-0.0027726660
OpponentScore	-0.092020561	-0.262742941	-0.255998459	0.0008248417
DefPassingCmp	0.016440817	0.096162025	0.082526971	-0.0185400889
DefPassingAtt	0.027710829	0.244286780	0.200544151	-0.0425214004
DefPassingPct	-0.021573479	-0.221344974	-0.179426252	0.0393481920
DefPassingYds	0.006122048	0.019078257	0.018225936	-0.0011154025
DefPassingTD	-0.051995314	-0.107800922	-0.114258347	0.0090357515
DefRushingAtt	0.024154269	-0.187794753	-0.124721247	-0.0302753145
DefRushingYds	-0.027539861	-0.207119041	-0.172652188	-0.0052705412
DefRushingAvg	-0.054881707	-0.162156060	-0.156759235	0.0171753556
DefRushingTD	-0.053823632	-0.220126972	-0.199401378	0.0020325702
DefPlays	0.049762251	0.066617035	0.082028691	-0.0700555324
DefYards	-0.015655527	-0.139476214	-0.114393675	-0.0048397221
DefAvg	-0.049744979	-0.203408513	-0.184266567	0.0393677257
DefFirstDownPass	-0.001905308	0.045479791	0.032759858	-0.0165897071

```
DefFirstDownRush  -0.013066454     -0.174615640      -0.138981640 -0.0009425108
DefFirstDownPen   -0.003582233      0.024027115       0.015638654  0.0686617594
DefFirstDownTotal -0.010683524     -0.083477752      -0.069316535  0.0004680850
DefPenalties      -0.006850053      0.021819923       0.011872019  0.0299312611
DefPenaltyYds     -0.016150128      0.009990979      -0.002994074  0.0388464539
DefFumbles         1.000000000      0.022387771       0.664523186 -0.0926405371
DefInterceptions   0.022387771      1.000000000       0.761957546 -0.0495394191
DefTotalTurnovers  0.664523186      0.761957546       1.000000000 -0.0970400378
Season            -0.092640537     -0.049539419      -0.097040038  1.0000000000
```

Notice right away – NetYards is highly correlated. But NetYards's also highly correlated with RushingYards, OffensiveYards and DefYards. And that makes sense: those things all feed into NetYards. Including all of these measures would be pointless – they would add error without adding much in the way of predictive power.

> **Your turn**: What else do you see? What other values have predictive power and aren't co-correlated?

We can add more just by simply adding them. Let's add the average yard per play for both offense and defense. They're correlated to NetYards, but not as much as you might expect.

```r
model2 <- lm(Differential ~ NetYards + TurnoverMargin + DefAvg + OffenseAvg, data=logs)
summary(model2)
```

```
Call:
lm(formula = Differential ~ NetYards + TurnoverMargin + DefAvg +
    OffenseAvg, data = logs)

Residuals:
    Min      1Q  Median      3Q     Max
-38.262  -6.258  -0.008   6.236  37.513

Coefficients:
                 Estimate Std. Error t value Pr(>|t|)
(Intercept)     0.5826970  0.4081448   1.428    0.153
NetYards        0.0551813  0.0007608  72.535   <2e-16 ***
TurnoverMargin  3.8548574  0.0392644  98.177   <2e-16 ***
DefAvg         -3.8991399  0.0698008 -55.861   <2e-16 ***
OffenseAvg      3.8612863  0.0688997  56.042   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 9.452 on 17286 degrees of freedom
  (1 observation deleted due to missingness)
Multiple R-squared:  0.8277,    Adjusted R-squared:  0.8277
F-statistic: 2.076e+04 on 4 and 17286 DF,  p-value: < 2.2e-16
```

Go down the list:

What is the Adjusted R-squared now? What is the p-value and is it less than .05? What is the Residual standard error?

The final thing we can do with this is predict things. Look at our coefficients table. See the Estimates? We can build a formula from that, same as we did with linear regressions.

How does this apply in the real world? Let's pretend for a minute that you are Mike Locksley, and you have a mess on your hands. Your job is to win conference titles. To do that, we need to know what attributes of a team should we emphasize. We can do that by looking at what previous Big Ten conference champions looked like.

So if our goal is to predict a conference champion team, we need to know what those teams did. Here's the regular season conference champions in this dataset.

```
logs %>%
  filter(Team == "Ohio State" & Season == 2021 | Team == "Ohio State" & Season == 2020 | T
  summarise(
    meanNetYards = mean(NetYards),
    meanTurnoverMargin = mean(TurnoverMargin),
    meanDefAvg = mean(DefAvg),
    meanOffenseAvg = mean(OffenseAvg)
  )
```

```
# A tibble: 1 x 4
  meanNetYards meanTurnoverMargin meanDefAvg meanOffenseAvg
         <dbl>              <dbl>      <dbl>          <dbl>
1         194.              0.688       5.15           7.28
```

Now it's just plug and chug.

```
(0.0551813*193.5833 ) + (3.8548574*0.6875) + (-3.8991399*5.145833   ) + (3.8612863*7.28125
```

```
[1] 21.96576
```

So a team with those numbers is going to average scoring 21 more points per game than their opponent. Sound like Ohio State in the last three years?

How does that compare to Maryland for last season?

```
logs %>%
  filter(
    Team == "Maryland" & Season == 2021
    ) %>%
  summarise(
    meanNetYards = mean(NetYards),
    meanTurnoverMargin = mean(TurnoverMargin),
    meanDefAvg = mean(DefAvg),
    meanOffenseAvg = mean(OffenseAvg)
  )
```

```
# A tibble: 1 x 4
  meanNetYards meanTurnoverMargin meanDefAvg meanOffenseAvg
         <dbl>              <dbl>      <dbl>          <dbl>
1         39.3             -0.538       5.67           6.24
```

```
(0.0551813*39.30769 ) + (3.8548574*-0.5384615) + (-3.8991399*.669231    ) + (3.8612863*6.2
```

```
[1] 22.15512
```

By this model, it predicted we would average outscoring our opponents by 22 points over the season! The reality? We were outscored by 8.4 on average.

11 Residuals

When looking at a linear model of your data, there's a measure you need to be aware of called residuals. The residual is the distance between what the model predicted and what the real outcome is. Take our model at the end of the correlation and regression chapter. Our model predicted Maryland, given a -20 net yardage margin would lose to Indiana by 2.2 points. They lost by 16. So our residual is -13.8.

Residuals can tell you several things, but most important is if a linear model the right model for your data. If the residuals appear to be random, then a linear model is appropriate. If they have a pattern, it means something else is going on in your data and a linear model isn't appropriate.

Residuals can also tell you who is underperforming and overperforming the model. And the more robust the model – the better your r-squared value is – the more meaningful that label of under or overperforming is.

Let's go back to our net yards model.

Then load the tidyverse.

```
library(tidyverse)
```

```
logs <- read_csv("data/footballlogs21.csv")
```

```
Rows: 1655 Columns: 54
-- Column specification --------------------------------------------------------
Delimiter: ","
chr   (8): HomeAway, Opponent, Result, TeamFull, TeamURL, Outcome, Team, Con...
dbl  (45): Game, PassingCmp, PassingAtt, PassingPct, PassingYds, PassingTD, ...
date  (1): Date

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.
```

First, let's make the columns we'll need.

```
residualmodel <- logs %>% mutate(differential = TeamScore - OpponentScore, NetYards = Offe
```

Now let's create our model.

```
fit <- lm(differential ~ NetYards, data = residualmodel)
summary(fit)
```

```
Call:
lm(formula = differential ~ NetYards, data = residualmodel)

Residuals:
    Min      1Q  Median      3Q     Max 
-36.252  -8.373  -0.004   8.660  35.396 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 0.428075   0.304630   1.405     0.16    
NetYards    0.107464   0.001762  60.978   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 12.35 on 1653 degrees of freedom
Multiple R-squared:  0.6923,	Adjusted R-squared:  0.6921 
F-statistic:  3718 on 1 and 1653 DF,  p-value: < 2.2e-16
```

We've seen this output before, but let's review because if you are using scatterplots to make a point, you should do this. First, note the Min and Max residual at the top. A team has underperformed the model by 36 points (!), and a team has overperformed it by 35 points (!!). The median residual, where half are above and half are below, is just slightly above the fit line. Close here is good.

Next: Look at the Adjusted R-squared value. What that says is that 69 percent of a team's scoring output can be predicted by their net yards.

Last: Look at the p-value. We are looking for a p-value smaller than .05. At .05, we can say that our correlation didn't happen at random. And, in this case, it REALLY didn't happen at random. But if you know a little bit about football, it doesn't surprise you that the more you outgain your opponent, the more you win by. It's an intuitive result.

What we want to do now is look at those residuals. We want to add them to our individual game records. We can do that by creating two new fields – predicted and residuals – to our dataframe like this:

```
residualmodel <- residualmodel %>% mutate(predicted = predict(fit), residuals = residuals(
```

Now we can sort our data by those residuals. Sorting in descending order gives us the games where teams overperformed the model. To make it easier to read, I'm going to use select to give us just the columns we need to see.

```
residualmodel %>% arrange(desc(residuals)) %>% select(Team, Opponent, Result, differential
```

```
# A tibble: 1,655 x 7
   Team          Opponent              Result    diffe~1 NetYa~2 predi~3 resid~4
   <chr>         <chr>                 <chr>       <dbl>   <dbl>   <dbl>   <dbl>
 1 Rutgers       Temple                W (61-14)      47     104   11.6     35.4
 2 Notre Dame    Wisconsin             W (41-13)      28     -72   -7.31    35.3
 3 Marshall      Navy                  W (49-7)       42      66    7.52    34.5
 4 Indiana       Idaho                 W (56-14)      42      77    8.70    33.3
 5 Nevada        Colorado State        W (52-10)      42      83    9.35    32.7
 6 Arizona State Arizona               W (38-15)      23     -82   -8.38    31.4
 7 Tennessee     Tennessee Tech        W (56-0)       56     238   26.0     30.0
 8 Charlotte     Florida International W (45-33)      12    -160  -16.8     28.8
 9 Colorado      Washington            W (20-17)       3    -243  -25.7     28.7
10 SMU           Abilene Christian     W (56-9)       47     169   18.6     28.4
# ... with 1,645 more rows, and abbreviated variable names 1: differential,
#   2: NetYards, 3: predicted, 4: residuals
```

So looking at this table, what you see here are the teams who scored more than their net yards would indicate. One of them should jump off the page at you.

Look at that Notre Dame-Wisconsin game. Not only did the Badgers outgain the Irish, but the model predicted they would win by a touchdown. Instead, Notre Dame **won by 28**!

But, before we can bestow any validity on this model, we need to see if this linear model is appropriate. We've done that some looking at our p-values and R-squared values. But one more check is to look at the residuals themselves. We do that by plotting the residuals with the predictor. We'll get into plotting soon, but for now just seeing it is enough.

The lack of a shape here – the seemingly random nature – is a good sign that a linear model works for our data. If there was a pattern, that would indicate something else was going on in our data and we needed a different model.

Another way to view your residuals is by connecting the predicted value with the actual value.

```
`geom_smooth()` using formula 'y ~ x'
```

The blue line here separates underperformers from overperformers.

11.1 Penalties

Now let's look at it where it doesn't work: Penalties.

```
penalties <- logs %>%
  mutate(
    differential = TeamScore - OpponentScore,
    TotalPenalties = Penalties+DefPenalties,
    TotalPenaltyYards = PenaltyYds+DefPenaltyYds
  )
```

```
pfit <- lm(differential ~ TotalPenaltyYards, data = penalties)
summary(pfit)
```

```
Call:
lm(formula = differential ~ TotalPenaltyYards, data = penalties)

Residuals:
```

```
    Min      1Q  Median      3Q     Max 
-64.472 -13.895   0.597  14.661  73.916 

Coefficients:
                   Estimate Std. Error t value Pr(>|t|)  
(Intercept)        2.868781   1.549579   1.851   0.0643 .
TotalPenaltyYards -0.008619   0.013535  -0.637   0.5244  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 22.26 on 1653 degrees of freedom
Multiple R-squared:  0.0002452, Adjusted R-squared:  -0.0003596 
F-statistic: 0.4055 on 1 and 1653 DF,  p-value: 0.5244
```

So from top to bottom:

- Our min and max go from -64 to positive 73
- Our adjusted R-squared is … -0.0003596 Not much at all.
- Our p-value is … 0.5244, which is more than than .05.

So what we can say about this model is that it's statistically insignificant and utterly meaningless. Normally, we'd stop right here – why bother going forward with a predictive model that isn't predictive? But let's do it anyway.

```
penalties$predicted <- predict(pfit)
penalties$residuals <- residuals(pfit)
```

```
penalties %>% arrange(desc(residuals)) %>% select(Team, Opponent, Result, TotalPenaltyYard
```

```
# A tibble: 1,655 x 5
   Team          Opponent                  Result   TotalPenaltyYards residuals
   <chr>         <chr>                     <chr>                <dbl>     <dbl>
 1 Oklahoma      Western Carolina          W (76-0)                91      73.9
 2 Pitt          New Hampshire             W (77-7)                67      67.7
 3 Miami (FL)    Central Connecticut State W (69-0)               100      67.0
 4 West Virginia Long Island               W (66-0)               100      64.0
 5 Buffalo       Wagner                    W (69-7)               144      60.4
 6 Auburn        Alabama State             W (62-0)               124      60.2
 7 Maryland      Howard                    W (62-0)                90      59.9
 8 Georgia       Vanderbilt                W (62-0)                46      59.5
 9 Baylor        Texas Southern            W (66-7)                65      56.7
10 Texas         Rice                      W (58-0)                52      55.6
```

```
# ... with 1,645 more rows
```

First, note all of the biggest misses here are all blowout games. The worst games of the season, the worst being Oklahoma vs Western Carolina. The model missed that differential by … 73 points. The margin of victory? 76 points. In other words, this model is terrible. But let's look at it anyway.

Well … it actually says that a linear model is appropriate. Which an important lesson – just because your residual plot says a linear model works here, that doesn't say your linear model is good. There are other measures for that, and you need to use them.

Here's the segment plot of residuals – you'll see some really long lines. That's a bad sign. Another bad sign? A flat fit line. It means there's no relationship between these two things. Which we already know.

```
`geom_smooth()` using formula 'y ~ x'
```

80
40
0
−40
differential
50
100
150
200
250
TotalPenaltyYards

12 Z-scores

Z-scores are a handy way to standardize numbers so you can compare things across groupings or time. In this class, we may want to compare teams by year, or era. We can use z-scores to answer questions like who was the greatest X of all time, because a z-score can put them in context to their era.

A z-score is a measure of how a particular stat is from the mean. It's measured in standard deviations from that mean. A standard deviation is a measure of how much variation – how spread out – numbers are in a data set. What it means here, with regards to z-scores, is that zero is perfectly average. If it's 1, it's one standard deviation above the mean, and 34 percent of all cases are between 0 and 1.

If you think of the normal distribution, it means that 84.3 percent of all case are below that 1. If it were -1, it would mean the number is one standard deviation below the mean, and 84.3 percent of cases would be above that -1. So if you have numbers with z-scores of 3 or even 4, that means that number is waaaaaay above the mean.

So let's use last year's Maryland women's basketball team, which if haven't been paying attention to current events, was very talented but had a few struggles.

12.1 Calculating a Z score in R

For this we'll need the logs of all college basketball games last season.

Load the tidyverse.

```
library(tidyverse)
```

And load the data.

```
gamelogs <- read_csv("data/wbb_game_logs_2022.csv")
```

```
Rows: 30438 Columns: 110
-- Column specification --------------------------------------------------------
Delimiter: ","
chr    (7): away_team_name, home_team_name, league, season_type, neutral_sit...
dbl  (101): end_year, neutral_site, ncaa_game_id, win, mp, pts, fgm, fga, fg...
date   (1): date
time   (1): gametime_east

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.
```

The first thing we need to do is select some fields we think represent team quality and a few things to help us keep things straight. So I'm going to pick shooting percentage, rebounding and the opponent version of the same two:

```
teamquality <- gamelogs %>%
  select(conference, team, fg_pct, trb, opp_fg_pct, opp_trb)
```

And since we have individual game data, we need to collapse this into one record for each team. We do that with … group by.

```
teamtotals <- teamquality %>%
  group_by(conference, team) %>%
  summarise(
    FGAvg = mean(fg_pct),
    ReboundAvg = mean(trb),
    OppFGAvg = mean(opp_fg_pct),
    OffRebAvg = mean(opp_trb)
    )
```

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

To calculate a z-score in R, the easiest way is to use the `scale` function in base R. To use it, you use `scale(FieldName, center=TRUE, scale=TRUE)`. The center and scale indicate if you want to subtract from the mean and if you want to divide by the standard deviation, respectively. We do.

When we have multiple z-scores, it's pretty standard practice to add them together into a composite score. That's what we're doing at the end here with `TotalZscore`. Note: We have to invert OppZscore and OppRebZScore by multiplying it by a negative 1 because the lower someone's opponent number is, the better.

```r
teamzscore <- teamtotals %>%
  mutate(
    FGzscore = as.numeric(scale(FGAvg, center = TRUE, scale = TRUE)),
    RebZscore = as.numeric(scale(ReboundAvg, center = TRUE, scale = TRUE)),
    OppZscore = as.numeric(scale(OppFGAvg, center = TRUE, scale = TRUE)) * -1,
    OppRebZScore = as.numeric(scale(OffRebAvg, center = TRUE, scale = TRUE)) * -1,
    TotalZscore = FGzscore + RebZscore + OppZscore + OppRebZScore
  )
```

So now we have a dataframe called `teamzscore` that has 1,317 basketball teams with Z scores. What does it look like?

```r
head(teamzscore)
```

```
# A tibble: 6 x 11
# Groups:   conference [1]
  conference team  FGAvg Rebou~1 OppFG~2 OffRe~3 FGzsc~4 RebZs~5 OppZs~6 OppRe~7
  <chr>      <chr> <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
1 AAC        Cinc~ 0.374    35.4   0.388    41.0 -0.981   -1.17   -0.445 -1.55
2 AAC        East~ 0.364    37.8   0.381    38.8 -1.58    -0.293  -0.100 -0.724
3 AAC        Hous~ 0.389    37.3   0.396    36.7 -0.105   -0.475  -0.839  0.0627
4 AAC        Memp~ 0.392    39.3   0.392    34.5  0.0931   0.235  -0.648  0.874
5 AAC        SMU   0.383    39.0   0.374    38.5 -0.450    0.110   0.246 -0.620
6 AAC        Sout~ 0.385    40.1   0.367    32.9 -0.365    0.518   0.579  1.47
# ... with 1 more variable: TotalZscore <dbl>, and abbreviated variable names
#   1: ReboundAvg, 2: OppFGAvg, 3: OffRebAvg, 4: FGzscore, 5: RebZscore,
#   6: OppZscore, 7: OppRebZScore
```

A way to read this – a team with a TotalZScore of 0 is precisely average. The larger the positive number, the more exceptional they are. The larger the negative number, the more truly terrible they are.

So who are the best teams in the country?

```
teamzscore %>% arrange(desc(TotalZscore))
```

```
# A tibble: 1,317 x 11
# Groups:   conference [102]
   confere~1 team  FGAvg Rebou~2 OppFG~3 OffRe~4 FGzsc~5 RebZs~6 OppZs~7 OppRe~8
   <chr>     <chr> <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
 1 SWAC      Jack~ 0.446    47.5   0.347    34.0   2.09    2.30     2.22    2.01
 2 ASUN      Libe~ 0.447    42.0   0.344    28.5   1.91    2.16     1.97    2.34
 3 USA South Sout~ 0.426    47.1   0.325    35.9   1.47    2.31     1.95    1.41
 4 <NA>      Dako~ 0.521    40     0.271    34     2.87    0.626    2.24    1.37
 5 <NA>      Lour~ 0.392    57     0.291    35     0.925   2.97     1.96    1.24
 6 <NA>      Mt. ~ 0.5      45     0.306    34     2.55    1.32     1.74    1.37
 7 SEC       Sout~ 0.432    48.3   0.330    30.2   0.558   2.02     2.19    2.10
 8 Pac-12    Stan~ 0.452    42.2   0.348    32.6   1.78    1.76     1.57    1.42
 9 CSAC      Clar~ 0.433    42.9   0.325    38.2   1.85    0.551    2.22    1.85
10 OVC       Murr~ 0.480    39.8   0.386    30.1   1.81    1.75     1.32    1.46
# ... with 1,307 more rows, 1 more variable: TotalZscore <dbl>, and abbreviated
#   variable names 1: conference, 2: ReboundAvg, 3: OppFGAvg, 4: OffRebAvg,
#   5: FGzscore, 6: RebZscore, 7: OppZscore, 8: OppRebZScore
```

Don't sleep on Jackson State! If we look for Power Five schools, South Carolina and Stanford are at the top, which checks out.

But closer to home, how is Maryland doing.

```
teamzscore %>%
  filter(conference == "Big Ten") %>%
  arrange(desc(TotalZscore)) %>%
  select(team, TotalZscore)
```

```
Adding missing grouping variables: `conference`
```

```
# A tibble: 14 x 3
# Groups:   conference [1]
   conference team         TotalZscore
```

```
   <chr>      <chr>              <dbl>
 1 Big Ten    Michigan           4.75
 2 Big Ten    Iowa               4.42
 3 Big Ten    Nebraska           3.11
 4 Big Ten    Rutgers            1.69
 5 Big Ten    Indiana            1.47
 6 Big Ten    Maryland           1.42
 7 Big Ten    Michigan St.       1.20
 8 Big Ten    Ohio St.           0.150
 9 Big Ten    Purdue            -1.37
10 Big Ten    Northwestern      -1.79
11 Big Ten    Minnesota         -2.07
12 Big Ten    Illinois          -3.96
13 Big Ten    Penn St.          -4.32
14 Big Ten    Wisconsin         -4.70
```

So, as we can see, with our composite Z Score, Maryland is better than average but not great. But better than Ohio State. Notice how, by this measure, Michigan and Iowa are far ahead of most of the conference, with Rutgers a surprising fourth (mostly thanks to its defense and rebounding).

We can limit our results to just Power Five conferences plus the Big East:

```
powerfive_plus_one <- c("SEC", "Big Ten", "Pac-12", "Big 12", "ACC", "Big East")
teamzscore %>%
  filter(conference %in% powerfive_plus_one) %>%
  arrange(desc(TotalZscore)) %>%
  select(team, TotalZscore)
```

```
Adding missing grouping variables: `conference`
```

```
# A tibble: 76 x 3
# Groups:   conference [6]
   conference team           TotalZscore
   <chr>      <chr>                <dbl>
 1 SEC        South Carolina        6.87
 2 Pac-12     Stanford              6.52
 3 Big East   UConn                 5.67
 4 Big Ten    Michigan              4.75
 5 ACC        NC State              4.74
 6 Big Ten    Iowa                  4.42
 7 Big 12     Baylor                4.40
```

```
 8 SEC        Tennessee             3.90
 9 Big East   Marquette             3.90
10 Big Ten    Nebraska              3.11
# ... with 66 more rows
```

This makes a certain amount of sense: three of the Final Four teams - South Carolina, Stanford and UConn are 1,2,3 in this list. Louisville, the fourth team, ranks 13th. Marquette is an interesting #9 here. It doesn't necessarily mean they were the ninth-best team, but given their competition they shot the ball and rebounded the ball very well.

12.2 Writing about z-scores

The great thing about z-scores is that they make it very easy for you, the sports analyst, to create your own measures of who is better than who. The downside: Only a small handful of sports fans know what the hell a z-score is.

As such, you should try as hard as you can to avoid writing about them.

If the word z-score appears in your story or in a chart, you need to explain what it is. "The ranking uses a statistical measure of the distance from the mean called a z-score" is a good way to go about it. You don't need a full stats textbook definition, just a quick explanation. And keep it simple.

Never use z-score in a headline. Write around it. Away from it. Z-score in a headline is attention repellent. You won't get anyone to look at it. So "Tottenham tops in z-score" bad, "Tottenham tops in the Premiere League" good.

13 Clustering

One common effort in sports is to classify teams and players – who are this players peers? What teams are like this one? Who should we compare a player to? Truth is, most sports commentators use nothing more sophisticated that looking at a couple of stats or use the "eye test" to say a player is like this or that.

There's better ways.

In this chapter, we're going to use a method that sounds advanced but it really quite simple called k-means clustering. It's based on the concept of the k-nearest neighbor algorithm. You're probably already scared. Don't be.

Imagine two dots on a scatterplot. If you took a ruler out and measured the distance between those dots, you'd know how far apart they are. In math, that's called the Euclidean distance. It's just the space between them in numbers. Where k-nearest neighbor comes in, you have lots of dots and you want measure the distance between all of them. What does k-means clustering do? It lumps them into groups based on the average distance between them. Players who are good on offense but bad on defense are over here, good offense good defense are over here. And using the Euclidean distance between them, we can decide who is in and who is out of those groups.

For this exercise, I want to look at Jalen Smith, who played two seasons at Maryland before decamping for the NBA. Had he stayed, he could have been among the all-time Terp greats. So who does Jalen Smith compare to?

To answer this, we'll use k-means clustering.

First thing we do is load some libraries and set a seed, so if we run this repeatedly, our random numbers are generated from the same base. If you don't have the cluster library, just add it on the console with `install.packages("cluster")`

```
library(tidyverse)
library(cluster)

set.seed(1234)
```

I've gone and scraped stats for every player in that season.

Now load that data.

```r
players <- read_csv("data/players20.csv")
```

```
Rows: 5452 Columns: 57
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (8): Team, Player, Class, Pos, Height, Hometown, High School, Summary
dbl (49): #, Weight, Rk.x, G, GS, MP, FG, FGA, FG%, 2P, 2PA, 2P%, 3P, 3PA, 3...

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.
```

To cluster this data properly, we have some work to do.

First, it won't do to have players who haven't played, so we can use filter to find anyone with greater than 0 minutes played. Next, Jalen Smith is a forward, so let's just look at forwards. Third, we want to limit the data to things that make sense to look at for Smith – things like shooting, rebounds, blocks, turnovers and points.

```r
playersselected <- players %>%
  filter(MP>0) %>% filter(Pos == "F") %>%
  select(Player, Team, Pos, MP, `FG%`, TRB, BLK, TOV, PTS) %>%
  na.omit()
```

Now, k-means clustering doesn't work as well with data that can be on different scales. So comparing a percentage to a count metric – shooting percentage to points – would create chaos because shooting percentages are a fraction of 1 and points, depending on when they are in the season, could be quite large. So we have to scale each metric – put them on a similar basis using the distance from the max value as our guide. Also, k-means clustering won't work with text data, so we need to create a dataframe that's just the numbers, but scaled. We can do that with another select, and using mutate_all with the scale function. The `na.omit()` means get rid of any blanks, because they too will cause errors.

```r
playersscaled <- playersselected %>%
  select(MP, `FG%`, TRB, BLK, TOV, PTS) %>%
  mutate_all(scale) %>%
  na.omit()
```

With k-means clustering, we decide how many clusters we want. Most often, researchers will try a handful of different cluster numbers and see what works. But there are methods for finding the optimal number. One method is called the Elbow method. One implementation of this, borrowed from the University of Cincinnati's Business Analytics program, does this quite nicely with a graph that will help you decide for yourself.

All you need to do in this code is change out the data frame – `playersscaled` in this case – and run it.

```
# function to compute total within-cluster sum of square
wss <- function(k) {
  kmeans(playersscaled, k, nstart = 10 )$tot.withinss
}

# Compute and plot wss for k = 1 to k = 15
k.values <- 1:15

# extract wss for 2-15 clusters
wss_values <- map_dbl(k.values, wss)

plot(k.values, wss_values,
       type="b", pch = 19, frame = FALSE,
       xlab="Number of clusters K",
       ylab="Total within-clusters sum of squares")
```

The Elbow method – so named because you're looking for the "elbow" where the line flattens out. In this case, it looks like a K of 8 is ideal. So let's try that. We're going to use the kmeans function, saving it to an object called k5. We just need to tell it our dataframe name, how

many centers (k) we want, and we'll use a sensible default for how many different configurations to try.

```
k8 <- kmeans(playersscaled, centers = 8, nstart = 25)
```

Let's look at what we get.

```
k8
```

```
K-means clustering with 8 clusters of sizes 263, 161, 44, 188, 90, 392, 103, 398

Cluster means:
          MP         FG%        TRB        BLK        TOV        PTS
1  0.8578410 -0.04802976  0.5296900 -0.1249576  0.8012144  0.6761765
2  0.6114599  0.43888680  0.6835198  1.1508637  0.3303347  0.3928252
3  1.3194410  0.57964043  1.7227785  3.8227947  0.9327078  1.2461273
4  1.5250965  0.34205779  1.6794426  0.7714193  1.7162316  1.8637568
5 -1.1562229  2.05173492 -1.0114963 -0.6981471 -1.0058198 -0.9280719
6 -0.1008779  0.08931655 -0.1790649 -0.1316087 -0.1660950 -0.2669699
7 -1.2907157 -2.43019592 -1.1384508 -0.8183787 -1.1343745 -1.0786200
8 -0.9856353 -0.29446744 -0.9105630 -0.6706989 -0.8922622 -0.8719061

Clustering vector:
   [1] 4 2 6 7 8 4 3 8 8 8 8 8 5 4 4 2 1 8 5 1 8 8 5 7 4 2 6 5 4 2 3 1 6 6 7 7 4
  [38] 4 2 4 4 6 8 1 8 8 4 6 8 8 8 4 8 8 1 2 8 8 4 1 6 8 5 4 4 1 6 8 2 2 6 8 5 5
  [75] 3 4 1 8 5 8 5 4 1 6 6 8 8 8 1 6 2 5 6 6 6 5 4 3 8 7 1 6 8 8 5 5 7 4 1 6 5
 [112] 4 4 1 6 8 5 4 1 6 8 8 1 2 6 6 8 7 1 6 7 4 1 1 8 3 2 7 4 1 8 4 1 8 8 4 1 8
 [149] 2 5 6 2 6 6 8 1 1 2 6 8 4 8 8 8 2 6 6 6 8 7 2 8 2 1 2 6 1 4 1 6 8 8 8 1 1
 [186] 1 8 8 7 4 6 6 8 6 8 4 6 1 6 6 6 1 2 1 6 8 8 3 1 8 6 2 6 6 6 8 4 6 8 8 8 7
 [223] 1 2 6 8 2 2 2 6 4 1 8 8 1 1 6 6 8 7 4 4 2 2 8 4 3 8 7 4 1 7 8 4 6 6 6 6 4
 [260] 2 6 6 6 1 2 8 7 4 6 6 8 3 1 1 8 1 2 6 8 8 7 1 2 2 8 4 4 1 6 8 4 4 1 6 5 4
 [297] 8 8 5 1 1 2 8 8 3 4 1 8 4 1 6 6 7 4 1 6 6 8 2 6 8 8 4 6 5 8 7 4 4 8 8 4 2
 [334] 6 6 6 6 7 7 7 6 5 5 8 8 4 1 1 6 5 7 8 4 4 5 8 8 1 6 8 7 6 6 5 8 2 8 7 4 1
 [371] 1 2 8 1 2 6 8 7 6 1 6 8 4 1 2 1 8 1 4 6 8 5 8 5 4 4 6 6 8 4 1 6 8 3 1 1 6
 [408] 4 6 6 8 6 8 3 6 6 8 1 2 1 6 8 1 2 5 4 3 8 8 1 6 6 8 8 5 1 6 8 8 1 1 8 8 1
 [445] 6 8 4 1 6 8 8 5 7 1 8 6 8 7 3 6 6 6 6 8 2 6 8 4 1 1 6 7 4 6 1 8 2 6 6 8 8
 [482] 6 6 8 5 5 8 4 1 6 7 1 1 6 8 5 4 2 8 8 1 2 8 2 6 8 7 4 3 8 4 6 8 4 3 6 6 8
 [519] 6 2 5 1 6 6 6 4 3 6 8 8 4 4 6 6 6 1 2 6 8 5 4 4 2 6 6 4 3 6 7 7 4 2 6 8 8
 [556] 8 4 1 2 2 8 8 8 7 4 4 8 4 3 1 5 1 1 6 6 6 4 2 5 8 1 2 6 8 4 8 8 8 7 4 1 6
 [593] 8 8 8 2 2 8 7 7 4 1 6 5 8 6 6 5 4 2 1 8 4 3 6 6 8 8 4 1 6 6 8 7 2 6 8 4 4
 [630] 5 8 8 8 7 2 2 8 1 1 3 6 8 7 3 1 2 6 8 7 4 2 6 8 7 4 2 1 8 8 1 8 7 7 8 1 4
 [667] 6 6 8 8 2 6 8 1 3 5 8 4 4 6 2 6 1 1 1 8 4 2 8 4 1 6 6 8 4 1 1 8 8 6 2 2 6
```

```
 [704] 7 1 4 2 6 7 1 1 5 3 2 1 8 6 6 8 8 4 4 2 6 8 4 3 1 8 8 3 1 6 6 8 8 1 2 6 8
 [741] 8 6 2 6 6 8 7 4 6 6 6 6 8 8 1 6 6 8 1 6 6 6 8 5 5 1 1 4 1 8 7 1 2 2 5 4 1
 [778] 8 8 8 1 2 8 6 8 7 4 3 8 8 8 8 5 7 1 6 8 8 6 6 6 6 8 7 2 6 6 6 6 2 5 6 8 8
 [815] 3 1 1 6 2 6 8 5 1 6 6 6 8 8 7 2 6 6 5 2 2 2 6 8 8 4 2 6 6 3 6 8 8 1 6 8 7
 [852] 2 6 8 8 3 6 5 8 5 8 5 1 6 1 8 8 4 1 1 8 8 8 2 2 6 6 6 2 6 2 6 8 7 1 6 8 2
 [889] 2 1 5 4 1 1 2 6 8 8 1 1 6 6 7 4 1 1 2 1 6 8 1 5 8 7 4 4 2 8 5 2 6 6 6 5 5
 [926] 4 4 1 5 1 6 6 6 4 1 2 6 8 8 1 1 2 6 7 5 3 6 7 1 7 1 8 8 8 4 1 6 6 8 4 6 8
 [963] 4 6 8 8 4 2 6 6 6 6 8 4 6 6 8 3 2 6 6 8 7 4 2 6 8 7 7 1 1 6 8 6 8 6 6 7 1
[1000] 1 6 6 6 8 7 1 2 1 6 8 1 1 8 4 6 8 3 8 7 4 6 8 8 8 7 1 2 6 8 8 2 6 8 8 8 8
[1037] 2 2 1 6 4 6 8 8 8 3 1 1 6 6 8 8 7 2 1 8 8 1 6 8 5 8 1 6 6 7 4 2 6 6 6 8 1
[1074] 1 6 1 2 1 6 5 2 6 8 1 6 2 8 6 6 6 8 8 4 1 2 6 6 4 8 6 8 1 2 1 6 6 7 7 4 6
[1111] 8 5 2 1 2 6 6 5 5 8 5 4 4 8 5 8 4 6 6 2 6 8 7 1 6 7 7 4 2 8 4 4 2 2 6 5 4
[1148] 1 6 8 8 8 4 3 6 6 7 5 2 6 6 8 5 1 2 6 8 8 8 8 8 7 1 6 6 8 6 8 7 3 6 6 6 4
[1185] 2 3 4 4 2 8 4 4 8 8 6 2 2 8 4 4 2 8 2 2 1 8 8 4 3 2 8 2 1 8 6 1 1 1 6 6 5
[1222] 5 2 1 7 8 4 4 8 8 1 1 2 6 6 2 1 6 1 6 6 7 4 6 8 8 4 6 6 8 4 1 1 6 5 2 2 8
[1259] 6 6 6 8 4 1 8 7 4 5 2 1 6 6 6 8 4 1 6 1 8 8 7 3 6 8 5 1 3 6 6 6 8 3 4 1 1
[1296] 1 6 6 6 8 4 2 2 8 8 1 6 8 4 2 5 2 6 6 8 7 4 6 8 8 8 8 1 2 7 1 1 8 6 8 6 8
[1333] 7 8 8 4 3 6 2 1 2 6 8 5 1 1 8 8 7 8 4 1 6 1 2 6 4 4 6 4 6 5 6 8 4 1 6 8 8
[1370] 7 4 6 5 8 7 1 6 6 6 8 6 7 4 6 1 1 6 6 8 8 8 1 1 6 8 2 6 5 1 1 8 5 1 4 6 8
[1407] 5 4 8 7 1 2 6 6 7 1 6 8 1 1 1 6 8 6 6 6 6 7 1 1 6 8 8 8 4 1 2 8 1 1 6 8 4
[1444] 1 1 2 8 1 8 4 5 8 4 1 6 6 8 7 1 1 6 6 8 7 4 1 1 6 8 2 6 6 6 6 8 8 1 4 1 1
[1481] 8 8 7 1 4 6 5 8 1 6 2 8 7 2 3 2 6 8 8 4 2 6 6 8 7 1 1 6 8 1 2 6 6 4 6 6 1
[1518] 1 4 1 8 5 7 4 6 8 4 1 4 6 8 4 6 3 6 8 4 1 6 8 6 8 5 4 1 2 5 7 4 1 8 8 8 7
[1555] 7 2 6 8 8 7 1 1 6 8 2 1 1 1 6 8 8 8 7 3 4 6 5 4 6 6 8 4 6 6 2 8 8 7 1 1 2
[1592] 6 1 1 6 6 5 1 2 6 8 8 2 6 6 6 5 7 2 1 6 6 1 8 1 1 1 6 8 6 6 6 8 8 8 4 1 8
[1629] 6 5 4 1 8 5 8 1 2 6 6

Within cluster sum of squares by cluster:
[1] 310.41877 241.01243 137.33736 443.93894 116.53203 384.86933  56.08423
[8] 221.54760
 (between_SS / total_SS =  80.5 %)

Available components:

[1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss"
[6] "betweenss"    "size"         "iter"         "ifault"
```

Interpreting this output, the very first thing you need to know is that **the cluster numbers are meaningless**. They aren't ranks. They aren't anything. After you have taken that on board, look at the cluster sizes at the top. Clusters 3 and 4 are pretty large compared to others. That's notable. Then we can look at the cluster means. For reference, 0 is going to be average. So group 1 is below average on minutes played. Groups 8 is slightly above, group 5 is well above.

So which group is Jalen Smith in? Well, first we have to put our data back together again. In K8, there is a list of cluster assignments in the same order we put them in, but recall we have no names. So we need to re-combine them with our original data. We can do that with the following:

```
playercluster <- data.frame(playersselected, k8$cluster)
```

Now we have a dataframe called playercluster that has our player names and what cluster they are in. The fastest way to find Jalen Smith is to double click on the playercluster table in the environment and use the search in the top right of the table. Because this is based on some random selections of points to start the groupings, these may change from person to person, but Smith is in Group 2 in my data.

We now have a dataset and can plot it like anything else. Let's get Cam Mack and then plot him against the rest of college basketball on rebounds versus minutes played.

```
cm <- playercluster %>% filter(Player == "Jalen Smith")

cm
```

```
       Player                Team Pos  MP   FG. TRB BLK TOV PTS k8.cluster
1 Jalen Smith Maryland Terrapins   F 861 0.534 293  64  49 425          3
```

So Jalen's in cluster 2, which if you look at our clusters, puts him in the cluster with all above average metrics. What does that look like? We know Jalen was a rebounding machine, so where do group 2 people grade out on rebounds?

```
ggplot() +
  geom_point(data=playercluster, aes(x=MP, y=TRB, color=k8.cluster)) +
  geom_point(data=cm, aes(x=MP, y=TRB), color="red")
```

Not bad, not bad. But who are Jalen Smith's peers? If we look at the numbers in Group 2, there's 495 of them. So let's limit them to just Big Ten guards. Unfortunately, my scraper didn't quite work and in the place of Conference is the coach's name. So I'm going to have to do this the hard way and make a list of Big Ten teams and filter on that. Then I'll sort by minutes played.

```
big10 <- c("Nebraska Cornhuskers", "Iowa Hawkeyes", "Minnesota Golden Gophers", "Illinois

playercluster %>% filter(k8.cluster == 2) %>% filter(Team %in% big10) %>% arrange(desc(MP)
```

```
            Player                     Team Pos  MP   FG. TRB BLK TOV PTS
1       Kyle Young     Ohio State Buckeyes   F 573 0.585 144  13  14 188
2 Brandon Johns Jr     Michigan Wolverines   F 546 0.492 111  20  21 169
3     Ryan Kriener           Iowa Hawkeyes   F 514 0.559 118  20  35 220
4     E.J. Liddell     Ohio State Buckeyes   F 441 0.443  98  28  26 175
5   Marcus Bingham Michigan State Spartans   F 330 0.409 111  40  14 107
  k8.cluster
1          2
2          2
3          2
4          2
5          2
```

So there are the 4 forwards most like Jalen Smith in the Big Ten. Are they the best forwards in the conference?

13.1 Advanced metrics

How much does this change if we change the metrics? I used pretty standard box score metrics above. What if we did it using Player Efficiency Rating, True Shooting Percentage, Point Production, Assist Percentage, Win Shares Per 40 Minutes and Box Plus Minus (you can get definitions of all of them by hovering over the stats on Nebraksa's stats page).

We'll repeat the process. Filter out players who don't play, players with stats missing, and just focus on those stats listed above.

```
playersadvanced <- players %>%
  filter(MP>0) %>%
  filter(Pos == "F") %>%
  select(Player, Team, Pos, PER, `TS%`, PProd, `AST%`, `WS/40`, BPM) %>%
  na.omit()
```

Now to scale them.

```
playersadvscaled <- playersadvanced %>%
  select(PER, `TS%`, PProd, `AST%`, `WS/40`, BPM) %>%
  mutate_all(scale) %>%
  na.omit()
```

Let's find the optimal number of clusters.

```
# function to compute total within-cluster sum of square
wss <- function(k) {
  kmeans(playersadvscaled, k, nstart = 10 )$tot.withinss
}

# Compute and plot wss for k = 1 to k = 15
k.values <- 1:15

# extract wss for 2-15 clusters
wss_values <- map_dbl(k.values, wss)
```

```
Warning: did not converge in 10 iterations

Warning: did not converge in 10 iterations
```

```r
plot(k.values, wss_values,
       type="b", pch = 19, frame = FALSE,
       xlab="Number of clusters K",
       ylab="Total within-clusters sum of squares")
```

Looks like 4.

```r
advk4 <- kmeans(playersadvscaled, centers = 4, nstart = 25)
```

What do we have here?

```r
advk4
```

```
K-means clustering with 4 clusters of sizes 627, 320, 612, 82

Cluster means:
         PER        TS%      PProd        AST%      WS/40        BPM
1  0.3328142  0.5094144 -0.0942388 -0.09932351  0.3805037  0.3628638
2  0.9655878  0.4538599  1.5112893  0.82146770  0.8477533  0.8427975
3 -0.5374881 -0.3986254 -0.5452426 -0.22446363 -0.5022825 -0.4844985
4 -2.3014614 -2.6912101 -1.1077608 -0.77100101 -2.4690239 -2.4475330
```

```
Clustering vector:
   [1] 2 1 1 3 3 2 2 3 1 3 3 1 1 2 2 1 1 3 1 3 3 3 1 4 2 1 1 1 2 2 2 2 2 1 3 4 2
  [38] 2 1 2 2 3 1 2 1 3 2 3 3 3 3 2 1 3 1 1 1 3 2 2 1 1 1 2 2 1 1 3 1 1 1 3 1 2
  [75] 2 2 1 1 1 3 1 2 1 1 3 3 3 3 3 1 1 1 3 3 1 3 2 2 3 4 1 1 3 1 1 1 4 2 1 1 1
 [112] 2 2 2 1 3 1 2 2 1 3 3 2 3 3 1 3 3 1 1 4 2 1 1 3 1 1 4 2 2 1 2 1 3 3 2 2 3
 [149] 3 1 2 1 3 3 3 2 1 1 1 1 2 1 3 3 2 1 3 3 1 3 2 3 1 1 2 1 1 1 1 1 3 3 3 1 3
 [186] 3 1 3 4 2 1 1 1 3 3 2 1 3 1 3 1 2 1 3 1 1 3 2 1 3 3 1 1 1 3 4 2 1 3 3 3 4
 [223] 3 1 3 3 1 2 1 1 2 3 3 3 3 3 1 3 3 4 2 2 1 1 1 2 1 1 4 2 3 4 3 2 2 1 1 3 2
 [260] 1 1 1 1 2 1 3 4 2 3 3 3 2 1 3 3 2 1 1 3 3 4 1 1 1 1 2 2 3 1 3 2 2 1 3 1 2
 [297] 3 3 1 3 3 3 3 3 2 2 1 3 2 1 3 3 4 2 1 3 3 3 1 1 3 1 2 1 1 3 4 2 2 1 3 2 1
 [334] 1 1 1 1 4 4 3 3 1 1 1 4 2 1 1 3 1 4 3 2 2 1 3 3 2 3 3 3 2 3 2 3 1 1 3 2 2
 [371] 1 1 3 3 3 3 3 3 1 1 1 3 3 3 1 3 1 2 2 1 1 1 3 3 2 2 1 3 1 2 1 1 3 2 3 3 3
 [408] 2 1 1 1 1 3 2 1 3 3 3 3 3 3 3 1 1 1 2 1 3 3 3 1 1 3 3 1 2 1 3 3 1 3 3 3 2
 [445] 1 3 2 1 1 3 3 2 4 2 1 1 3 4 2 3 1 1 3 3 1 1 3 2 1 1 3 4 2 1 3 3 1 1 1 3 3
 [482] 1 1 1 1 1 3 2 1 3 4 2 3 1 3 1 1 1 3 3 1 1 3 1 1 3 3 2 2 1 2 1 3 2 2 1 1 3
 [519] 3 1 1 1 1 3 1 2 2 3 1 3 2 2 1 3 1 1 1 3 3 1 2 2 1 1 3 1 2 1 1 4 4 2 1 3 1
 [556] 3 1 2 2 2 1 1 3 3 4 2 2 3 2 2 1 1 2 1 3 3 3 2 3 3 3 2 1 1 4 2 1 1 3 4 1 3
 [593] 1 3 3 3 2 1 1 3 4 2 2 1 1 3 1 1 1 2 1 1 3 2 2 1 3 1 2 2 2 3 3 3 3 2 1 1 2
 [630] 2 1 3 3 3 4 2 1 3 2 1 1 3 3 4 2 1 1 1 1 3 2 1 3 1 4 2 3 1 3 3 1 3 3 3 4 1
 [667] 1 1 1 3 3 1 3 1 3 2 1 3 2 1 1 1 3 1 3 1 1 2 1 3 2 2 3 1 3 2 2 1 3 3 1 1 1
 [704] 1 4 2 1 1 3 3 3 3 1 2 3 3 1 3 3 3 3 2 2 1 1 1 1 1 3 3 3 1 1 3 3 3 3 1 3 1
 [741] 3 3 1 1 3 1 3 4 2 3 1 3 3 3 1 3 1 3 3 1 1 1 1 3 1 1 2 1 1 1 1 4 2 2 1 1 2
 [778] 1 3 3 3 3 3 3 3 3 4 2 1 3 1 1 1 1 4 1 1 1 3 3 3 3 1 3 3 3 1 3 3 1 1 1 1 3
 [815] 1 2 2 1 1 1 1 1 1 1 1 3 1 1 3 4 1 3 3 1 2 2 3 1 3 3 2 1 1 3 2 1 1 3 1 3 3
 [852] 4 2 3 3 3 2 1 1 3 1 1 1 3 3 3 3 3 2 2 3 3 3 3 1 1 1 1 1 1 1 1 1 3 2 1 3 3
 [889] 1 1 3 1 2 1 3 3 3 3 3 1 2 1 3 3 2 1 1 1 3 3 3 1 3 1 4 2 2 2 1 3 2 1 1 3 1
 [926] 3 2 2 2 1 2 3 3 3 2 3 1 1 1 1 1 3 1 1 4 1 2 1 3 3 4 3 3 3 3 2 2 1 3 1 1 3
 [963] 3 2 3 3 3 2 2 1 1 3 1 3 2 3 3 4 2 1 1 3 3 4 2 1 1 1 3 3 3 3 3 1 3 3 2 1 4
[1000] 2 1 3 3 3 1 4 1 1 3 3 1 1 1 3 2 1 3 1 1 4 1 3 3 3 3 4 1 2 3 1 3 2 1 3 3 3
[1037] 3 2 1 1 3 2 1 3 3 3 2 1 1 3 1 3 3 4 1 3 3 3 1 3 1 1 3 1 3 3 3 1 1 3 3 3 1
[1074] 1 3 3 1 2 3 3 1 1 3 3 2 1 1 3 3 1 3 1 3 2 3 1 3 3 2 3 1 3 3 1 3 3 3 4 4 2
[1111] 1 3 1 2 1 1 1 1 1 1 3 1 2 3 1 1 3 2 3 1 1 1 1 4 3 3 3 3 2 1 3 2 2 2 2 2 1
[1148] 2 1 1 3 3 1 2 2 1 3 4 2 1 1 1 3 1 2 1 1 3 3 1 3 3 3 1 1 1 3 1 3 4 2 3 1 3
[1185] 2 1 1 2 2 2 1 2 2 3 3 1 1 3 3 2 2 1 1 2 1 1 1 1 2 2 1 3 1 3 3 1 2 1 1 3 1
[1222] 1 1 3 1 4 4 2 2 3 3 1 3 1 3 3 2 3 1 1 1 3 3 2 1 3 3 2 1 3 3 2 2 1 3 1 2 1
[1259] 3 2 1 3 1 2 2 1 3 2 3 2 1 1 1 1 1 2 3 3 1 3 3 3 2 1 3 1 2 1 3 3 3 3 2 1 1
[1296] 3 1 3 3 3 3 2 1 1 3 3 3 3 3 2 2 1 2 1 1 3 3 2 3 3 2 3 3 2 2 3 2 2 3 3 3 3
[1333] 3 4 4 3 1 1 1 1 1 1 3 3 1 2 1 1 3 2 3 2 1 1 3 1 3 2 2 1 2 3 1 3 3 2 3 3 3
[1370] 3 4 2 1 1 3 4 3 1 3 3 1 3 4 2 1 3 3 3 3 3 3 3 2 1 1 4 1 1 3 2 3 4 2 2 1 1
[1407] 3 1 2 3 4 3 3 3 3 4 3 3 3 1 1 1 1 3 3 3 3 2 4 3 1 3 3 1 3 2 2 1 3 2 1 1 3
[1444] 2 2 1 1 1 1 3 2 1 1 2 3 3 3 3 4 2 2 1 3 3 4 2 1 1 3 3 1 3 3 3 3 3 3 2 2 1
[1481] 2 1 3 3 2 2 3 1 3 1 1 1 3 3 1 1 2 3 3 3 2 1 3 1 3 4 2 2 1 3 2 2 3 1 2 1 3
```

```
[1518] 1 1 2 3 3 3 4 4 2 1 3 2 1 2 3 3 2 1 2 3 1 2 1 1 1 3 3 1 2 1 1 1 4 2 1 1 3
[1555] 3 4 4 1 1 3 1 4 2 1 1 4 3 1 3 3 3 3 3 3 4 2 2 1 1 2 1 1 1 2 1 3 1 1 3 3 2
[1592] 1 1 3 1 1 1 1 1 2 3 1 1 3 1 3 3 3 3 4 1 2 1 3 3 3 1 3 3 3 4 3 3 3 3 4 1 2
[1629] 1 1 3 1 2 2 1 1 1 1 2 1 3

Within cluster sum of squares by cluster:
[1] 1427.2032 1442.9566 1161.4081  669.2421
 (between_SS / total_SS =  52.2 %)

Available components:

[1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss"
[6] "betweenss"    "size"         "iter"         "ifault"
```

Looks like this time, cluster 1 is all below average and cluster 5 is all above. Which cluster is Jalen Smith in?

```r
playeradvcluster <- data.frame(playersadvanced, advk4$cluster)
```

```r
jsadv <- playeradvcluster %>% filter(Player == "Jalen Smith")

jsadv
```

```
       Player                Team Pos  PER   TS. PProd AST. WS.40  BPM
1 Jalen Smith Maryland Terrapins   F 28.9 0.623   392  5.2 0.252 11.8
  advk4.cluster
1             2
```

Cluster 4 on my dataset. So in this season, we can say he's in a big group of players who are all above average on these advanced metrics.

Now who are his Big Ten peers?

```r
playeradvcluster %>%
  filter(advk4.cluster == 4) %>%
  filter(Team %in% big10) %>%
  arrange(desc(PProd))
```

```
            Player                 Team Pos   PER TS. PProd AST.  WS.40   BPM
1 Harrison Hookfin Ohio State Buckeyes   F  -8.3 0.2     3    0 -0.206 -15.0
```

```
2       Samad Qawi   Wisconsin Badgers   F -17.5 0.0     1    0 -0.206 -14.6
3 Nathan Childress    Indiana Hoosiers   F  -6.0 0.0     1    0 -0.200 -27.3
  advk4.cluster
1             4
2             4
3             4
```

Sorting on Points Produced, Cam Mack is third out of the 15 forwards in the Big Ten who land in Cluster 4. Seems advanced metrics rate Jalen Smith pretty highly.

14 Simulations

In the 2017-2018 season, James Palmer Jr. took 139 three point attempts and made 43 of them for a .309 shooting percentage. A few weeks into the next season, he was 7 for 39 – a paltry .179.

Is something wrong or is this just bad luck?

Luck is something that comes up a lot in sports. Is a team unlucky? Or a player? One way we can get to this, we can get to that is by simulating things based on their typical percentages. Simulations work by choosing random values within a range based on a distribution. The most common distribution is the normal or binomial distribution. The normal distribution is where the most cases appear around the mean, 66 percent of cases are within one standard deviation from the mean, and the further away from the mean you get, the more rare things become.

Let's simulate 39 three point attempts 1000 times with his season long shooting percentage and see if this could just be random chance or something else.

We do this using a base R function called `rbinom` or binomial distribution. So what that means is there's a normally distributed chance that James Palmer Jr. is going to shoot above and below his career three point shooting percentage. If we randomly assign values in that distribution 1000 times, how many times will it come up 7, like this example?

```r
set.seed(1234)

simulations <- rbinom(n = 1000, size = 39, prob = .309)

table(simulations)
```

```
simulations
  3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22
  1   4   5  12  35  44  76 117 134 135 135  99  71  53  37  21  15   2   3   1
```

How do we read this? The first row and the second row form a pair. The top row is the number of shots made. The number immediately under it is the number of simulations where that occurred.

```
simulations
  3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  1   4   5  12  35  44  76 117 134 135 135  99  71  53  37
 18  19  20  21  22
 21  15   2   3   1
```

So what we see is given his season long shooting percentage, it's not out of the realm of randomness that with just 39 attempts for Palmer, he's only hit only 7. In 1000 simulations, it comes up 35 times. Is he below where he should be? Yes. Will he likely improve and soon? Unless something is very wrong, yes. And indeed, by the end of the season, he finished with a .313 shooting percentage from 3 point range. So we can say he was just unlucky.

14.1 Cold streaks

During the final regular-season game in the 2021-22 season, Maryland, shooting .326 on the season from behind the arc, went 1-15 in the first half. How strange is that?

```r
set.seed(1234)

simulations <- rbinom(n = 1000, size = 15, prob = .326)

hist(simulations)
```

```r
table(simulations)
```

```
simulations
  0   1   2   3   4   5   6   7   8   9  10  11
  5  17  61 135 204 222 164  93  60  33   3   3
```

Short answer: Pretty weird, but not totally unheard of. If you simulate 15 threes 1000 times, about 50 times it will result in a single made three-pointer. It's slightly more common that the team would hit 8 threes out of 15. So going that cold is not totally out of the realm of random chance, but it's pretty uncommon.

15 Intro to ggplot with bar charts

With `ggplot2`, we dive into the world of programmatic data visualization. The `ggplot2` library implements something called the grammar of graphics. The main concepts are:

- aesthetics - which in this case means the data which we are going to plot
- geometries - which means the shape the data is going to take
- scales - which means any transformations we might make on the data
- facets - which means how we might graph many elements of the same dataset in the same space
- layers - which means how we might lay multiple geometries over top of each other to reveal new information.

Hadley Wickham, who is behind all of the libraries we have used in this course to date, wrote about his layered grammar of graphics in this 2009 paper that is worth your time to read.

Here are some `ggplot2` resources you'll want to keep handy:

- The ggplot documentation.
- The ggplot cookbook

Let's dive in using data we've already seen before – football attendance. This workflow will represent a clear picture of what your work in this class will be like for much of the rest of the semester. One way to think of this workflow is that your R Notebook is now your digital sketchbook, where you will try different types of visualizations to find ones that work. Then, you will either write the code that adds necessary and required parts to finish it, or you'll export your work into a program like Illustrator to finish the work.

To begin, we'll use data we've seen before: college football attendance.

Now load the tidyverse.

```
library(tidyverse)
```

And the data.

```
attendance <- read_csv('data/attendance.csv')
```

```
Rows: 149 Columns: 9
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (2): Institution, Conference
dbl (7): 2013, 2014, 2015, 2016, 2017, 2018, 2019

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.
```

First, let's get a top 10 list by announced attendance in the most recent season we have data. We'll use the same tricks we used in the filtering assignment.

```
attendance %>%
  arrange(desc(`2019`)) %>%
  top_n(10) %>%
  select(Institution, `2019`)
```

```
Selecting by 2019

# A tibble: 10 x 2
   Institution `2019`
   <chr>        <dbl>
 1 Michigan    780215
 2 Penn St.    739747
 3 Ohio St.    723679
 4 Texas A&M   711258
 5 Alabama     707817
 6 LSU         705892
 7 Tennessee   702912
 8 Georgia     649722
 9 Nebraska    625436
10 Auburn      600355
```

That looks good, so let's save it to a new data frame and use that data frame instead going forward.

```
top10 <- attendance %>%
  arrange(desc(`2019`)) %>%
  top_n(10) %>%
  select(Institution, `2019`)
```

```
Selecting by 2019
```

15.1 The bar chart

The easiest thing we can do is create a simple bar chart of our data. **Bar charts show magnitude. They invite you to compare how much more or less one thing is compared to others.**

We could, for instance, create a bar chart of the total attendance. To do that, we simply tell `ggplot2` what our dataset is, what element of the data we want to make the bar chart out of (which is the aesthetic), and the geometry type (which is the geom). It looks like this:

```
ggplot() + geom_bar(data=top10, aes(x=Institution))
```

Note: top10 is our data, `aes` means aesthetics, `x=Institution` explicitly tells `ggplot2` that our x value – our horizontal value – is the Institution field from the data, and then we add on the `geom_bar()` as the geometry. And what do we get when we run that?

```
ggplot() +
  geom_bar(
    data=top10,
    aes(x=Institution)
  )
```

We get … weirdness. We expected to see bars of different sizes, but we get all with a count of

1. What gives? Well, this is the default behavior. What we have here is something called a histogram, where `ggplot2` helpfully counted up the number of times the Institution appears and counted them up. Since we only have one record per Institution, the count is always 1. How do we fix this? By adding `weight` to our aesthetic.

```
ggplot() +
  geom_bar(
    data=top10,
    aes(x=Institution, weight=`2019`)
  )
```

Closer. But … what order is that in? And what happened to our count numbers on the left? Why are they in scientific notation?

Let's deal with the ordering first. `ggplot2`'s default behavior is to sort the data by the x axis variable. So it's in alphabetical order. To change that, we have to `reorder` it. With `reorder`, we first have to tell `ggplot` what we are reordering, and then we have to tell it HOW we are reordering it. So it's reorder(FIELD, SORTFIELD).

```
ggplot() +
  geom_bar(
    data=top10,
```

```
    aes(
      x=reorder(Institution, `2019`),
      weight=`2019`
      )
    )
```

Better. We can argue about if the right order is smallest to largest or largest to smallest. But this gets us close. By the way, to sort it largest to smallest, put a negative sign in front of the sort field.

```
ggplot() +
  geom_bar(
    data=top10,
    aes(
      x=reorder(Institution, -`2019`),
      weight=`2019`
      )
    )
```

15.2 Scales

To fix the axis labels, we need try one of the other main elements of the `ggplot2` library, which is transform a scale. More often that not, that means doing something like putting it on a logarithmic scale or some other kind of transformation. In this case, we're just changing how it's represented. The default in `ggplot2` for large values is to express them as scientific notation. Rarely ever is that useful in our line of work. So we have to transform them into human readable numbers.

The easiest way to do this is to use a library called `scales` and it's already installed.

```
library(scales)
```

```
Attaching package: 'scales'

The following object is masked from 'package:purrr':

    discard
```

```
The following object is masked from 'package:readr':

    col_factor
```

To alter the scale, we add a piece to our plot with `+` and we tell it which scale is getting altered and what kind of data it is. In our case, our Y axis is what is needing to be altered, and it's continuous data (meaning it can be any number between x and y, vs discrete data which are categorical). So we need to add `scale_y_continuous` and the information we want to pass it is to alter the labels with a function called `comma`.

```
ggplot() +
  geom_bar(
    data=top10,
    aes(
      x=reorder(Institution, -`2019`),
      weight=`2019`
      )
    ) +
  scale_y_continuous(labels=comma)
```

Better.

15.3 Styling

We are going to spend a lot more time on styling, but let's add some simple labels to this with a new bit called `labs` which is short for labels.

```
ggplot() +
  geom_bar(
    data=top10,
    aes(
      x=reorder(Institution, -`2019`),
      weight=`2019`)
    ) +
  scale_y_continuous(labels=comma) +
  labs(
    title="Top 10 Football Programs By Attendance",
    x="School",
    y="Attendance"
)
```

The library has lots and lots of ways to alter the styling – we can programmatically control nearly every part of the look and feel of the chart. One simple way is to apply themes in the

library already. We do that the same way we've done other things – we add them. Here's the light theme.

```
ggplot() +
  geom_bar(
    data=top10,
    aes(x=reorder(Institution, -`2019`),
        weight=`2019`)) +
  scale_y_continuous(labels=comma) +
  labs(
    title="Top 10 Football Programs By Attendance",
    x="School",
    y="Attendance") +
  theme_light()
```

Or the minimal theme:

```
ggplot() +
  geom_bar(
    data=top10,
    aes(x=reorder(Institution, -`2019`),
```

```
      weight=`2019`)) +
  scale_y_continuous(labels=comma) +
  labs(
    title="Top 10 Football Programs By Attendance",
    x="School",
    y="Attendance") +
  theme_minimal()
```

Later on, we'll write our own themes. For now, the built in ones will get us closer to something that looks good.

15.4 One last trick: coord flip

Sometimes, we don't want vertical bars. Maybe we think this would look better horizontal. How do we do that? By adding `coord_flip()` to our code. It does what it says – it inverts the coordinates of the figures.

```
ggplot() +
  geom_bar(
    data=top10,
```

```
  aes(x=reorder(Institution, -`2019`),
      weight=`2019`)) +
scale_y_continuous(labels=comma) +
labs(
  title="Top 10 Football Programs By Attendance",
  x="School",
  y="Attendance") +
theme_minimal() +
coord_flip()
```

16 Stacked bar charts

One of the elements of data visualization excellence is **inviting comparison**. Often that comes in showing **what proportion a thing is in relation to the whole thing**. With bar charts, we're showing magnitude of the whole thing. If we have information about the parts of the whole, **we can stack them on top of each other to compare them, showing both the whole and the components**. And it's a simple change to what we've already done.

We're going to use a dataset of college football games from last season.

Load the tidyverse.

```
library(tidyverse)
```

And the data.

```
football <- read_csv("data/footballlogs21.csv")
```

```
Rows: 1655 Columns: 54
-- Column specification --------------------------------------------------------
Delimiter: ","
chr   (8): HomeAway, Opponent, Result, TeamFull, TeamURL, Outcome, Team, Con...
dbl  (45): Game, PassingCmp, PassingAtt, PassingPct, PassingYds, PassingTD, ...
date  (1): Date

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.
```

What we have here is every game in college football for the 2019-2020 season. The question we want to answer is this: Who had the most prolific offenses in the Big Ten? And how did they get there?

So to make this chart, we have to just add one thing to a bar chart like we did in the previous chapter. However, it's not that simple.

We have game data, and we need season data. To get that, we need to do some group by and sum work. And since we're only interested in the Big Ten, we have some filtering to do too. For this, we're going to measure offensive production by rushing yards and passing yards. So

if we have all the games a team played, and the rushing and passing yards for each of those games, what we need to do to get the season totals is just add them up.

```
football %>%
  group_by(Conference, Team) %>%
  summarise(
    SeasonRushingYards = sum(RushingYds),
    SeasonPassingYards = sum(PassingYds),
  ) %>% filter(Conference == "Big Ten Conference")
```

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

# A tibble: 14 x 4
# Groups:   Conference [1]
   Conference         Team           SeasonRushingYards SeasonPassingYards
   <chr>              <chr>                       <dbl>              <dbl>
 1 Big Ten Conference Illinois                     2049               1874
 2 Big Ten Conference Indiana                      1388               2106
 3 Big Ten Conference Iowa                         1733               2521
 4 Big Ten Conference Maryland                     1774               3960
 5 Big Ten Conference Michigan                     3001               3202
 6 Big Ten Conference Michigan State               2295               3298
 7 Big Ten Conference Minnesota                    2589               2106
 8 Big Ten Conference Nebraska                     2168               3201
 9 Big Ten Conference Northwestern                 1741               2128
10 Big Ten Conference Ohio State                   2347               4952
11 Big Ten Conference Penn State                   1401               3491
12 Big Ten Conference Purdue                       1104               4620
13 Big Ten Conference Rutgers                      1798               2239
14 Big Ten Conference Wisconsin                    2740               2082
```

By looking at this, we can see we got what we needed. We have 14 teams and numbers that look like season totals for yards. Save that to a new dataframe.

```
football %>%
  group_by(Conference, Team) %>%
  summarise(
    SeasonRushingYards = sum(RushingYds),
    SeasonPassingYards = sum(PassingYds),
  ) %>% filter(Conference == "Big Ten Conference") -> yards
```

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

Now, the problem we have is that ggplot wants long data and this data is wide. So we need to use `tidyr` to make it long, just like we did in the transforming data chapter.

```
yards %>%
  pivot_longer(
    cols=starts_with("Season"),
    names_to="Type",
    values_to="Yards")
```

```
# A tibble: 28 x 4
# Groups:   Conference [1]
   Conference         Team     Type               Yards
   <chr>              <chr>    <chr>              <dbl>
 1 Big Ten Conference Illinois SeasonRushingYards  2049
 2 Big Ten Conference Illinois SeasonPassingYards  1874
 3 Big Ten Conference Indiana  SeasonRushingYards  1388
 4 Big Ten Conference Indiana  SeasonPassingYards  2106
 5 Big Ten Conference Iowa     SeasonRushingYards  1733
 6 Big Ten Conference Iowa     SeasonPassingYards  2521
 7 Big Ten Conference Maryland SeasonRushingYards  1774
 8 Big Ten Conference Maryland SeasonPassingYards  3960
 9 Big Ten Conference Michigan SeasonRushingYards  3001
10 Big Ten Conference Michigan SeasonPassingYards  3202
# ... with 18 more rows
```

What you can see now is that we have two rows for each team: One for rushing yards, one for passing yards. This is what ggplot needs. Save it to a new dataframe.

```
yards %>%
  pivot_longer(
    cols=starts_with("Season"),
    names_to="Type",
    values_to="Yards") -> yardswide
```

Building on what we learned in the last chapter, we know we can turn this into a bar chart with an x value, a weight and a geom_bar. What we are going to add is a `fill`. The `fill` will stack bars on each other based on which element it is. In this case, we can fill the bar by Type, which means it will stack the number of rushing yards on top of passing yards and we can see how they compare.

```
ggplot() +
  geom_bar(data=yardswide, aes(x=Team, weight=Yards, fill=Type)) +
  coord_flip()
```

What's the problem with this chart?

There's a couple of things, one of which we'll deal with now: The ordering is alphabetical (from the bottom up). So let's `reorder` the teams by Yards.

```
ggplot() +
  geom_bar(data=yardswide, aes(x=reorder(Team, Yards), weight=Yards, fill=Type)) +
  coord_flip()
```

And just like that … Ohio State comes out on top, and Maryland third. Not exactly a "three yards and a cloud of dust" conference.

17 Circular bar plots

Does November basketball really not matter? Are games played early in the season, before teams have had a chance to learn how to play together and when many teams feast on cupcake schedules, meaningful come March?

Let's look, using a new form of chart called a circular bar plot. It's a chart type that combines several forms we've used before: bar charts to show magnitude, stacked bar charts to show proportion, but we're going to add bending the chart around a circle to add some visual interesting-ness to it. We're also going to use time as an x-axis value to make a not subtle circle of time reference – a common technique with circular bar charts.

We'll use a dataset of every women's college basketball game last season.

Load your libraries.

```
library(tidyverse)
library(lubridate)
```

And load your data.

```
logs <- read_csv("data/wbb_game_logs_2022.csv")
```

```
Rows: 30438 Columns: 110
-- Column specification --------------------------------------------------------
Delimiter: ","
chr    (7): away_team_name, home_team_name, league, season_type, neutral_sit...
dbl  (101): end_year, neutral_site, ncaa_game_id, win, mp, pts, fgm, fga, fg...
date   (1): date
time   (1): gametime_east

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.
```

17.1 Does November basketball matter?

So let's test the notion of November Basketball Doesn't Matter. What matters in basketball? Let's start simple: Wins.

We can group by date and conference and sum up the wins. How many wins by day does each conference get?

```
dates <- logs %>% group_by(date, conference) %>% summarise(wins = sum(win))
```

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

Earlier, we did stacked bar charts. We have what we need to do that now.

```
ggplot() + geom_bar(data=dates, aes(x=date, weight=wins, fill=conference)) + theme_minimal
```

Eeek. This is already looking not great. But to make it a circular bar chart, we add `coord_polar()` to our chart.

```
ggplot() + geom_bar(data=dates, aes(x=date, weight=wins, fill=conference)) + theme_minimal
```

Based on that, the day is probably too thin a slice, and there's way too many conferences in college basketball. Let's group this by months and filter out all but the power five conferences.

```
p5 <- c("SEC", "Big Ten", "Pac-12", "Big 12", "ACC")
```

To get months, we're going to use a function in the library `lubridate` called `floor_date`, which combine with mutate will give us a field of just months.

```
wins <- logs %>% mutate(month = floor_date(date, unit="months")) %>% group_by(month, confe
```

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

Now we can use wins to make our circular bar chart of wins by month in the Power Five.

```
ggplot() + geom_bar(data=wins, aes(x=month, weight=wins, fill=conference)) + theme_minimal
```

Yikes. That looks a lot like a broken pie chart. So months are too thick of a slice. Let's use weeks in our floor date to see what that gives us.

```
wins <- logs %>% mutate(week = floor_date(date, unit="weeks")) %>% group_by(week, conferen
```

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

```
ggplot() + geom_bar(data=wins, aes(x=week, weight=wins, fill=conference)) + theme_minimal(
```

That looks better. But what does it say? Does November basketball matter? What this is saying is … yeah, it kinda does. The reason? Lots of wins get piled up in November and December, during non-conference play. So if you are a team with NCAA tournament dreams, you need to win games in November to make sure your tournament resume is where it needs to be come March. Does an individual win or loss matter? Probably not. But your record in November does.

17.2 Does it show you where you are?

So here is the problem we have:

1. We have data for every game. In the past, we were able to calculate the team wins and losses because the way the data records them is `team` is the main team, and they win or lose. The opponent is recorded, but not in its own column of that name. In addition, the opponent has the mirror image of this game as well, where they are `team`. So essentially every game is in here twice – one for each team that plays in the game.
2. We need to attach the opponent's winning percentage to each game so we can decide if it's a quality win for `team`.

First we need to populate `opponent` based on the whatever is not the `team`. Then what we have to do is invert the process that we've done before. We need to group by the opponent

and we need to invert the wins and losses. A win in the win column is a win for the `team`. That means each loss in the win column is a WIN for the opponent.

Once we invert, the data looks very similar to what we've done before. One other thing: I noticed there's some tournament games in here, so the filter at the end strips them out.

```
logs <- logs %>% mutate(opponent = case_when(
  team == home_team_name ~ away_team_name,
  team == away_team_name ~ home_team_name
))

oppwinlosslogs <- logs %>% mutate(winloss = case_when(
  grepl("W", win) ~ 0,
  grepl("L", win) ~ 1)
) %>% filter(date < "2022-03-14")
```

So now we have a dataframe called oppwinlosslogs that has an inverted winloss column. So now we can group by the Opponent and sum the wins and it will tell us how many games the Opponent won. We can also count the wins and get a winning percentage.

```
oppwinlosslogs %>% group_by(opponent) %>% summarise(games=n(), wins=sum(win)) %>% mutate(w
```

Now we have a dataframe of 1,316 opponent winning records. Wait, what? There's like ~350 teams in major college basketball, so why 1,316? If you look through it, there's a bunch of teams playing lower level teams. Given that they are lower level, they're likely cannon fodder and will lose the game, and we're going to filter them out in a minute.

Now we can join the opponent winning percentage to our winlosslogs data so we can answer our question about quality wins.

```
winlosslogs <- logs %>% mutate(winloss = case_when(
  grepl("W", win) ~ 1,
  grepl("L", win) ~ 0)
) %>% filter(date < "2022-03-13")
```

```
winlosslogs %>% left_join(opprecord, by=("opponent")) -> winswithopppct
```

Now that we have a table called winswithopppct, we can filter out non-power 5 teams and teams that won less than 60 percent of their games and run the same calculations in the book.

```
p5 <- c("SEC", "Big Ten", "Pac-12", "Big 12", "ACC")
```

```
winswithopppct %>% filter(winpct > .6) %>% mutate(week = floor_date(date, unit="weeks")) %
```

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

Now with our dataframe called qualitywins, we can chart it again.

```
ggplot() + geom_bar(data=qualitywins, aes(x=week, weight=wins, fill=conference)) + theme_m
```

Look at this chart and compare it to the first one.

18 Waffle charts

Pie charts are the devil. They should be an instant F in any data visualization class. The problem? How carefully can you evaluate angles and area? Unless they are blindingly obvious and only a few categories, not well. If you've got 25 categories, how can you tell the difference between 7 and 9 percent? You can't.

So let's introduce a better way: The Waffle Chart. Some call it a square pie chart. I personally hate that. Waffles it is.

A waffle chart is designed to show you parts of the whole – proportionality. How many yards on offense come from rushing or passing. How many singles, doubles, triples and home runs make up a teams hits. How many shots a basketball team takes are two pointers versus three pointers.

First, install the library in the console. We want a newer version of the `waffle` library than is in CRAN – where you normally get libraries from – so copy and paste this into your console:

```
install.packages("waffle")
```

Now load it:

```
library(waffle)
```

18.1 Making waffles with vectors

Let's look at Maryland's football bowl game win over Virginia Tech last year. Here's the box score, which we'll use for this part of the walkthrough.

Maybe the easiest way to do waffle charts, at least at first, is to make vectors of your data and plug them in. To make a vector, we use the `c` or concatenate function.

So let's look at offense. Rushing vs passing.

```
md <- c("Rushing"=206, "Passing"=275)
vt <- c("Rushing"=122, "Passing"=137)
```

So what does the breakdown of the night look like?

The waffle library can break this down in a way that's easier on the eyes than a pie chart. We call the library, add the data, specify the number of rows, give it a title and an x value label, and to clean up a quirk of the library, we've got to specify colors.

```
waffle(
        md,
        rows = 10,
        title="Maryland's offense",
        xlab="1 square = 1 yard",
        colors = c("black", "red")
)
```

Or, we could make this two teams in the same chart.

```
passing <- c("Maryland"=275, "Virginia Tech"=137)
```

```
waffle(
        passing,
        rows = 10,
        title="Maryland vs Virginia Tech: passing",
        xlab="1 square = 1 yard",
```

```
        colors = c("red", "black")
)
```

So what does it look like if we compare the two teams using the two vectors in the same chart? To do that – and I am not making this up – you have to create a waffle iron. Get it? Waffle charts? Iron?

```
iron(
 waffle(md,
        rows = 10,
        title="Maryland's offense",
        xlab="1 square = 1 yard",
        colors = c("black", "red")
        ),
 waffle(vt,
        rows = 10,
        title="Virginia Tech's offense",
        xlab="1 square = 1 yard",
        colors = c("black", "red")
        )
)
```

What do you notice about this chart? Notice how the squares aren't the same size? Well, Maryland out-gained Virginia Tech by a long way. So the squares aren't the same size because the numbers aren't the same. We can fix that by adding an unnamed padding number so the number of yards add up to the same thing. Let's make the total for everyone be 481, Maryland's total yards of offense. So to do that, we need to add a padding of 222 to Virginia Tech. REMEMBER: Don't name it or it'll show up in the legend.

```
md <- c("Rushing"=206, "Passing"=275)
vt <- c("Rushing"=122, "Passing"=137, 222)
```

Now, in our waffle iron, if we don't give that padding a color, we'll get an error. So we need to make it white. Which, given our white background, means it will disappear.

```
iron(
 waffle(md,
        rows = 10,
        title="Maryland's offense",
        xlab="1 square = 1 yard",
        colors = c("black", "red")
        ),
 waffle(vt,
        rows = 10,
```

```
        title="Virginia Tech's offense",
        xlab="1 square = 1 yard",
        colors = c("orange", "brown", "white")
        )
)
```

One last thing we can do is change the 1 square = 1 yard bit – which makes the squares really small in this case – by dividing our vector. Look, it's math on vectors!

```
iron(
 waffle(md/2,
        rows = 10,
        title="Maryland's offense",
        xlab="1 square = 1 yard",
        colors = c("black", "red")
        ),
 waffle(vt/2,
        rows = 10,
        title="Virginia Tech's offense",
        xlab="1 square = 1 yard",
        colors = c("orange", "brown", "white")
```

```
            )
)
```

News flash: Virginia Tech is changing its fight song to "Everybody Hurts" by REM.

19 Step charts

Step charts are **a method of showing progress** toward something. They combine showing change over time – **cumulative change over time** – with magnitude. They're good at inviting comparison.

There's great examples out there. First is the Washignton Post looking at Lebron passing Jordan's career point total. Another is John Burn-Murdoch's work at the Financial Times (which is paywalled) about soccer stars. Here's an example of his work outside the paywall.

To replicate this, we need cumulative data – data that is the running total of data at a given point. So think of it this way – Maryland scores 50 points in a basketball game and then 50 more the next, their cumulative total at two games is 100 points.

Step charts can be used for all kinds of things – showing how a player's career has evolved over time, how a team fares over a season, or franchise history. Let's walk through an example.

Let's look at Fred Hoiberg's team last season.

We'll need the tidyverse.

```
library(tidyverse)
```

And we need to load our logs data we just downloaded.

```
logs <- read_csv("data/logs21.csv")
```

```
Rows: 8229 Columns: 48
-- Column specification --------------------------------------------------------
Delimiter: ","
chr   (8): Season, TeamFull, Opponent, HomeAway, W_L, URL, Conference, Team
dbl  (39): Game, TeamScore, OpponentScore, TeamFG, TeamFGA, TeamFGPCT, Team3...
date  (1): Date

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.
```

Here we're going to look at the scoring differential of teams. If you score more than your opponent, you win. So it stands to reason that if you score a lot more than your opponent over the course of a season, you should be very good, right? Let's see.

The first thing we're going to do is calculate that differential. Then, we'll group it by the team. After that, we're going to summarize using a new function called `cumsum` or cumulative sum – the sum for each game as we go forward. So game 1's cumsum is the differential of that game. Game 2's cumsum is Game 1 + Game 2. Game 3 is Game 1 + 2 + 3 and so on.

```
difflogs <- logs %>%
  mutate(Differential = TeamScore - OpponentScore) %>%
  group_by(Team) %>%
  mutate(CumDiff = cumsum(Differential))
```

Now that we have the cumulative sum for each, let's filter it down to just Big Ten teams.

```
bigdiff <- difflogs %>% filter(Conference == "Big Ten")
```

The step chart is it's own geom, so we can employ it just like we have the others. It works almost exactly the same as a line chart, but it uses the cumulative sum instead of a regular value and, as the name implies, creates a step like shape to the line instead of a curve.

```
ggplot() + geom_step(data=bigdiff, aes(x=Date, y=CumDiff, group=Team))
```

Let's try a different element of the aesthetic: color, but this time inside the aesthetic. Last time, we did the color outside. When you put it inside, you pass it a column name and ggplot will color each line based on what thing that is, and it will create a legend that labels each line that thing.

```
ggplot() + geom_step(data=bigdiff, aes(x=Date, y=CumDiff, group=Team, color=Team))
```

From this, we can see a handful of teams in the Big Ten had negative point differentials last season. But which is which? And which one is Maryland? Too many colors and it's too hard to tell. How to sort that out? Let's add some helpers beyond layering.

Let's look at the top team plus Maryland.

```
umd <- bigdiff %>% filter(Team == "Maryland")
il <- bigdiff %>% filter(Team == "Illinois")
```

Let's introduce a couple of new things here. First, note when I take the color OUT of the aesthetic, the legend disappears.

The second thing I'm going to add is the annotation layer. In this case, I am adding a text annotation layer, and I can specify where by adding in a x and a y value where I want to put it. This takes some finesse. After that, I'm going to add labels and a theme.

```r
ggplot() +
  geom_step(data=bigdiff, aes(x=Date, y=CumDiff, group=Team), color="light grey") +
  geom_step(data=umd, aes(x=Date, y=CumDiff, group=Team), color="red") +
  geom_step(data=il, aes(x=Date, y=CumDiff, group=Team), color="orange") +
  annotate("text", x=(as.Date("2021-02-10")), y=-70, label="Maryland") +
  annotate("text", x=(as.Date("2021-03-01")), y=330, label="Illinois") +
  labs(
    x="Date",
    y="Cumulative Point Differential",
    title="Maryland's late charge",
    subtitle="The Terps started solidly but didn't gain much until late in the season.",
    caption="Source: Sports-Reference.com | By Derek Willis") +
  theme_minimal()
```

20 Dumbbell and lollipop charts

Second to my love of waffle charts because I'm always hungry, dumbbell charts are an excellently named way of **showing the difference between two things on a number line** – a start and a finish, for instance. Or the difference between two related things. Say, turnovers and assists.

Lollipop charts – another excellent name – are a variation on bar charts. They do a good job of showing magnitude and difference between things.

To use both of them, you need to add a new library:

`install.packages("ggalt")`

Let's give it a whirl.

```
library(tidyverse)
library(ggalt)
```

20.1 Dumbbell plots

For this, let's use college football game logs from last season.

And load it.

```
logs <- read_csv("data/footballlogs21.csv")
```

```
Rows: 1655 Columns: 54
-- Column specification --------------------------------------------------------
Delimiter: ","
chr   (8): HomeAway, Opponent, Result, TeamFull, TeamURL, Outcome, Team, Con...
dbl  (45): Game, PassingCmp, PassingAtt, PassingPct, PassingYds, PassingTD, ...
date  (1): Date

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.
```

For the first example, let's look at the difference between a team's giveaways – turnovers lost – versus takeaways, turnovers gained. To get this, we're going to add up all offensive turnovers and defensive turnovers for a team in a season and take a look at where they come out. To make this readable, I'm going to focus on the Big Ten.

```
turnovers <- logs %>%
  group_by(Team, Conference) %>%
  summarise(
    Giveaways = sum(TotalTurnovers),
    Takeaways = sum(DefTotalTurnovers)) %>%
  filter(Conference == "Big Ten Conference")
```

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

Now, the way that the `geom_dumbbell` works is pretty simple when viewed through what we've done before. There's just some tweaks.

First: We start with the y axis. The reason is we want our dumbbells going left and right, so the label is going to be on the y axis.

Second: Our x is actually two things: x and xend. What you put in there will decide where on the line the dot appears.

```
ggplot() +
  geom_dumbbell(
    data=turnovers,
    aes(y=Team, x=Takeaways, xend=Giveaways)
  )
```

Well, that's a chart alright, but what dot is the giveaways and what are the takeaways? To fix this, we'll add colors.

So our choice of colors here is important. We want giveaways to be seen as bad and takeaways to be seen as good. So lets try red for giveaways and green for takeaways. To make this work, we'll need to do three things: first, use the English spelling of color, so `colour`. The, uh, `colour` is the bar between the dots, the `x_colour` is the color of the x value dot and the `xend_colour` is the color of the xend dot. So in our setup, takeaways are x, they're good, so they're green.

```
ggplot() +
  geom_dumbbell(
    data=turnovers,
    aes(y=Team, x=Takeaways, xend=Giveaways),
    colour = "grey",
    colour_x = "green",
    colour_xend = "red")
```

Better. Let's make two more tweaks. First, let's make the whole thing bigger with a `size` element. And let's add `theme_minimal` to clean out some cruft.

```
ggplot() +
  geom_dumbbell(
    data=turnovers,
    aes(y=Team, x=Takeaways, xend=Giveaways),
    size = 1,
    color = "grey",
    colour_x = "green",
    colour_xend = "red") +
  theme_minimal()
```

And now we have a chart that tells a story – got green on the right? That's good. A long distance between green and red? Better. But what if we sort it by good turnovers?

```
ggplot() +
  geom_dumbbell(
    data=turnovers,
    aes(y=reorder(Team, Takeaways), x=Takeaways, xend=Giveaways),
    size = 1,
    color = "grey",
    colour_x = "green",
    colour_xend = "red") +
  theme_minimal()
```

Believe it or not, Iowa had the most takeaways in the Big Ten this season.

20.2 Lollipop charts

Sticking with takeaways, lollipops are similar to bar charts in that they show magnitude. And like dumbbells, they are similar in that we start with a y – the traditional lollipop chart is on its side – and we only need one x. The only additional thing we need to add is that we need to tell it that it is a horizontal chart.

```
ggplot() +
  geom_lollipop(
    data=turnovers,
    aes(y=Team, x=Takeaways),
    horizontal = TRUE
    )
```

We can do better than this with a simple theme_minimal and some better labels.

```
ggplot() +
  geom_lollipop(
    data=turnovers,
    aes(y=reorder(Team, Takeaways), x=Takeaways),
    horizontal = TRUE
    ) + theme_minimal() +
  labs(title = "Nebraska's defense improved, but needs more takeaways", y="Team")
```

How about some layering?

```
nu <- turnovers %>% filter(Team == "Nebraska")
```

```
ggplot() +
  geom_lollipop(
    data=turnovers,
    aes(y=reorder(Team, Takeaways), x=Takeaways),
    horizontal = TRUE
    ) +
  geom_lollipop(
    data=nu,
    aes(y=Team, x=Takeaways),
    horizontal = TRUE,
    color = "red"
  ) +
  theme_minimal() +
  labs(title = "Nebraska's defense wasn't as bad as you think", y="Team")
```

The headline says it all.

21 Scatterplots

On the Monday, Sept. 21, 2020 edition of the Pick Six Podcast, Omaha World Herald reporter Sam McKewon talked a little about the Nebraska mens basketball team. Specifically the conversation was about a new roster release, and how the second year of Fred Hoiberg ball was going to look very different, starting with the heights of the players. After a near complete roster turnover, the players on the team now were nearly all taller than 6'4", and one of the shorter ones is penciled in as the starting point guard.

Why is that important? One reason, McKewon posited, is that teams made a lot of three point shots on Nebraska. In fact, Nebraska finished dead last in the conference in three points shots made against them. McKewon chalked this up to bad perimeter defense, and that Nebraska needed to improve it. Being taller – or more specifically having the longer arms that go with being taller – will help with that, McKewon said.

Better perimeter defense, better team.

Well, we know how the season went: 7 wins, same as the previous year.

But the question before you is this: is what McKewon said true? Does keeping a lid on your opponent's ability to score three pointers mean more wins?

This is what we're going to start to answer today. And we'll do it with scatterplots and regressions. Scatterplots are very good at showing **relationships between two numbers**.

First, we need libraries and every college basketball game from the 2019-2020 season. Why that season? Because I did this with the COVID season and everything was wonky and made no sense. To make it make sense, you had to do a ton of twisting and turning and that was pointless. **What we're interested in is less about a specific team and more about a general point: Are these numbers related and by how much? What can they tell you about your team in general?**

Load the tidyverse.

```
library(tidyverse)
```

And the data.

```
logs <- read_csv("data/logs1520.csv")
```

```
Rows: 68617 Columns: 44
-- Column specification --------------------------------------------------------
Delimiter: ","
chr   (6): HomeAway, Opponent, W_L, Team, Conference, season
dbl  (36): X1, Game, TeamScore, OpponentScore, TeamFG, TeamFGA, TeamFGPCT, T...
lgl   (1): Blank
date  (1): Date

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.
```

To do this, we need all teams and their season stats. How much, team to team, does a thing matter? That's the question you're going to answer.

In our case, we want to know how much do three point shots made influence wins? How much difference can we explain in wins by knowing how many threes the other team made against you? We're going to total up the number of threes each team allowed and their season wins in one swoop.

To do this, we need to use conditional logic – `case_when` in this case – to determine if the team won or lost the game. In this case, we'll create a new column called `winloss`. Case when statements can be read like this: When This is True, Do This. This bit of code – which you can use in a lot of contexts in this class – uses the `grepl` function to look for the letter W in the W_L column and, if it finds it, makes winloss 1. If it finds an L, it makes it 0. Sum your winloss column and you have your season win total. The reason we have to use `grepl` to find W or L is because Sports Reference will record overtime wins differently than regular wins. Same with losses.

```
winlosslogs <- logs %>%
  mutate(
    winloss = case_when(
      grepl("W", W_L) ~ 1,
      grepl("L", W_L) ~ 0)
)
```

Now we can get a dataframe together that gives us the total wins for each team, and the total three point shots made. We'll call that new dataframe `threedef`.

```
threedef <- winlosslogs %>%
  group_by(Team, season) %>%
  summarise(
    Wins = sum(winloss),
    TotalOpp3P = sum(Opponent3P)
```

```
    ) %>% na.omit()
```

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

Now let's look at the scatterplot. With a scatterplot, we put what predicts the thing on the X axis, and the thing being predicted on the Y axis. In this case, X is our three pointers given up, y is our wins.

```
ggplot() + geom_point(data=threedef, aes(x=TotalOpp3P, y=Wins))
```

Let's talk about this. This seems kind of random, but clustered around the middle. There's not really a clear pattern here. But can we get a better sense of this? Yes, by adding another geom – `geom_smooth`. It's identical to our `geom_point`, but we add a method to the end, which in this case we're using the linear method or `lm`.

```
ggplot() +
  geom_point(data=threedef, aes(x=TotalOpp3P, y=Wins)) +
  geom_smooth(data=threedef, aes(x=TotalOpp3P, y=Wins), method="lm")
```

```
`geom_smooth()` using formula 'y ~ x'
```

A straight line across is bad. It means there's no pattern here. The numbers don't suggest anything. The dots are very spread out. Which is a clue that you should be asking a question here: how strong of a relationship is this? How much can threes given up explain wins? Can we put some numbers to this?

Of course we can. We can apply a linear model to this – remember Chapter 9? We're going to create an object called fit, and then we're going to put into that object a linear model – `lm` – and the way to read this is "wins are predicted by opponent threes". Then we just want the summary of that model.

```
fit <- lm(Wins ~ TotalOpp3P, data = threedef)
summary(fit)
```

```
Call:
lm(formula = Wins ~ TotalOpp3P, data = threedef)

Residuals:
     Min       1Q   Median       3Q      Max
```

```
-15.8715  -4.8477   0.0676   4.1383  21.0459

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 17.199150   0.893730  19.244   <2e-16 ***
TotalOpp3P  -0.001400   0.003773  -0.371    0.711
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6.265 on 2104 degrees of freedom
Multiple R-squared:  6.547e-05, Adjusted R-squared:  -0.0004098
F-statistic: 0.1378 on 1 and 2104 DF,  p-value: 0.7106
```

Remember from Chapter 9: There's just a few things you really need.

The first thing: R-squared. In this case, the Adjusted R-squared value is -0.0004098, which we can interpret as the number of threes the opponent makes predicts about a teeny tiny percent of the variance in wins. Which sounds not great.

Second: The P-value. We want anything less than .05. If it's above .05, the difference between them is not statistically significant – it's probably explained by random chance. In our case, we have 0.7106. Is that more than .05? Yes. Yes it is. So this is random. There's no relationship between the number of threes the opponent makes and the number of wins a team has.

Normally, we'd stop here, but let's look at the third element: The coefficient. In this case, the coefficient for TeamOpp3P is -0.001400. What this model predicts, given that and the intercept of 17.199150, is this: Every team starts with about 17 wins. For every 100 three pointers the other team makes, you lose .14 games off that total. So if you give up 100 threes in a season, you'll be a 17 win team. Give up 200, you're closer to a 16 win team, and so on. How am I doing that? Remember your algebra and y = mx + b. In this case, y is the wins, m is the coefficient, x is the number of threes given up and b is the intercept.

Let's use Maryland as an example. They had 227 threes scored on them in the 2019-2020 season.

y = -0.001400 * 227 + 17.199150 or 16.8 wins.

How many wins did Maryland have? 24.

What does that mean? It means that Maryland OVERPERFORMED according to this model. But our R-squared is so small that even if this weren't random, it would be largely pointless. Put another way: more than 99 percent of the difference in wins between teams is predicted by something else.

Where is Maryland on the plot? We know we can use layering for that.

```r
umd <- threedef %>% filter(Team == "Maryland Terrapins")
```

```r
ggplot() +
  geom_point(data=threedef, aes(x=TotalOpp3P, y=Wins)) +
  geom_smooth(data=threedef, aes(x=TotalOpp3P, y=Wins), method="lm") +
  geom_point(data=umd, aes(x=TotalOpp3P, y=Wins), color="red")
```

```
`geom_smooth()` using formula 'y ~ x'
```

21.1 Let's see it work

One thing we do know is related is shooting percentage. Shoot better, win games. Simple. But how well does that work?

We could just average together each team's shooting night, but this isn't the same thing as the season shooting average. It's the average of shooting percentages. To get the season average, we have to sum attempts and makes and do the math ourselves. Averaging the percentages will get us close, but be careful when you do things like this that you're describing your number correctly.

Let's do this the hard – read: correct – way.

```
shooting <- winlosslogs %>%
  group_by(Team, season) %>%
  summarise(
    Wins = sum(winloss),
    Shots = sum(TeamFGA),
    Makes = sum(TeamFG)
    ) %>%
  mutate(AvgShootingPct = Makes/Shots) %>%
  na.omit()
```

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

Now we can chart it and see what our relationship looks like.

```
ggplot() +
  geom_point(data=shooting, aes(x=AvgShootingPct, y=Wins)) +
  geom_smooth(data=shooting, aes(x=AvgShootingPct, y=Wins), method="lm")
```

```
`geom_smooth()` using formula 'y ~ x'
```

This is what a positive relationship looks like – sloping up and to the right. The better you shoot, the more you win. If it were reversed – the better you shot, the fewer wins you got, the line would start at the top left and go down to the right. The direction of the line indicates the relationship between the numbers.

Let's get our linear regression stats.

```
shootfit <- lm(Wins ~ AvgShootingPct, data = shooting)
summary(shootfit)
```

```
Call:
lm(formula = Wins ~ AvgShootingPct, data = shooting)

Residuals:
     Min       1Q   Median       3Q      Max 
-14.6777  -3.3738  -0.0529   3.0892  17.2843 

Coefficients:
               Estimate Std. Error t value Pr(>|t|)    
(Intercept)     -50.733      1.862  -27.24   <2e-16 ***
AvgShootingPct  153.367      4.218   36.36   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.91 on 2106 degrees of freedom
Multiple R-squared:  0.3857,	Adjusted R-squared:  0.3854 
F-statistic:  1322 on 1 and 2106 DF,  p-value: < 2.2e-16
```

The p-value isn't random. The adjusted R-squared is pushing 40 percent. We've got something here. Let's use our coefficients to look at Maryland's 2019-2020 season (not the COVID season, because all the numbers are super weird in that season).

```
(153.942 * .419) + -50.733
```

```
[1] 13.7687
```

So this model says that based only on Maryland's shooting percentage, they should have won 13 or 14 games. They won 24. So per our model, Maryland overperformed. Obviously there's room for improvement in the model here, but you can see the power in scatterplots + regression.

22 Bubble charts

Here is the real talk: Bubble charts are hard. The reason they are hard is not because of the code, or the complexity or anything like that. They're a scatterplot with magnitude added – the size of the dot in the scatterplot has meaning. The hard part is seeing when a bubble chart works and when it doesn't.

If you want to see it work spectacularly well, watch a semi-famous Ted Talk by Hans Rosling from 2006 where bubble charts were the centerpiece. It's worth watching. It'll change your perspective on the world. No seriously. It will.

And since then, people have wanted bubble charts. And we're back to the original problem: They're hard. There's a finite set of circumstances where they work.

First, I'm going to show you an example of them not working to illustrate the point.

I'm going to load up my libraries.

```
library(tidyverse)
```

So for this example, I want to look at where Big Ten teams compare to the rest of college football last season. Is the Big Ten's reputation for tough games and defenses earned? Can we see patterns in good team vs bad teams?

I'm going to create a scatterplot with offensive yards per play on the X axis and defensive yards per play on the y axis. We can then divide the grid into four quadrants. Teams with high yards per offensive play and low defensive yards per play are teams with good offenses and good defenses. The opposite means bad defense, bad offense. Then, to drive the point home, I'm going to make the dot the size of the total wins on the season – the bubble in my bubble charts.

We'll use last season's college football games.

And load it.

```
logs <- read_csv("data/footballlogs21.csv")
```

```
Rows: 1655 Columns: 54
-- Column specification --------------------------------------------------------
```

```
Delimiter: ","
chr   (8): HomeAway, Opponent, Result, TeamFull, TeamURL, Outcome, Team, Con...
dbl  (45): Game, PassingCmp, PassingAtt, PassingPct, PassingYds, PassingTD, ...
date  (1): Date

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.
```

To do this, I've got some work to do. First, I need to mutate the outcomes of the games into 1s and 0s so I can add up the wins. We've done this before, so this won't be new to you, just adjusted slightly from basketball data.

```
winlosslogs <- logs %>%
  mutate(
    wins = case_when(
      grepl("W", Outcome) ~ 1,
      grepl("L", Outcome) ~ 0)
)
```

Now I have some more work to do. My football logs data has the yards per play of each game, and I could average those together and get something very close to what I'm going to do, but averaging each games yards per play is not the same thing as calculating it, so we're going to calculate it.

```
winlosslogs %>%
  group_by(Team, Conference) %>%
  summarise(
    TotalPlays = sum(OffensivePlays),
    TotalYards = sum(OffensiveYards),
    DefensivePlays = sum(DefPlays),
    DefensiveYards = sum(DefYards),
    TotalWins = sum(wins)) %>%
  mutate(
    OffensiveYPP = TotalYards/TotalPlays,
    DefensiveYPP = DefensiveYards/DefensivePlays) -> ypp
```

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

A bubble chart is just a scatterplot with one additional element in the aesthetic – a size. Here's the scatterplot version.

```
ggplot() +
  geom_point(
    data=ypp, aes(x=OffensiveYPP, y=DefensiveYPP)
    )
```

Looks kind of random, eh? In this case, that's not that bad because we're not claiming a relationship. We're saying the location on the chart has meaning. So, do teams on the bottom right – good offense, good defense – win more games?

Let's add the size element.

```
ggplot() +
  geom_point(
    data=ypp,
    aes(x=OffensiveYPP, y=DefensiveYPP, size=TotalWins)
    )
```

What does this chart tell you? We can see a general pattern that there are more big dots on the bottom right than the upper left. But we can make this more readable by adding an alpha element outside the aesthetic – alpha in this case is transparency – and we can manually change the size of the dots by adding `scale_size` and a `range`.

```
ggplot() +
  geom_point(
    data=ypp,
    aes(x=OffensiveYPP, y=DefensiveYPP, size=TotalWins),
    alpha = .3) +
  scale_size(range = c(3, 8), name="Wins")
```

And by now, you now know to add in the Big Ten as a layer, I would hope.

```
bigten <- ypp %>% filter(Conference == "Big Ten Conference")
```

```
ggplot() +
  geom_point(
    data=ypp,
    aes(x=OffensiveYPP, y=DefensiveYPP, size=TotalWins),
    color="grey",
    alpha=.5) +
  geom_point(
    data=bigten,
    aes(x=OffensiveYPP, y=DefensiveYPP, size=TotalWins),
    color="red")
```

Let's add some things to this chart to help us out. First, let's add lines that show us the average of all teams for those two metrics. So first, we need to calculate those. Because I have grouped data, it's going to require me to ungroup it so I can get just the total average of those two numbers.

```
ypp %>%
  ungroup() %>%
  summarise(
    offense = mean(OffensiveYPP),
    defense = mean(DefensiveYPP)
    )
```

```
# A tibble: 1 x 2
  offense defense
    <dbl>   <dbl>
1    5.79    5.66
```

Now we can use those averages to add two more geoms – geom_vline and geom_hline, for vertical lines and horizontal lines.

```
ggplot() +
  geom_point(
    data=ypp,
    aes(x=OffensiveYPP, y=DefensiveYPP, size=TotalWins),
    color="grey",
    alpha=.5) +
  geom_point(
    data=bigten,
    aes(x=OffensiveYPP, y=DefensiveYPP, size=TotalWins),
    color="red") +
  geom_vline(xintercept = 5.863151) +
  geom_hline(yintercept = 5.448228)
```

Now, let's add another new geom for us, using a new library called `ggrepel`, which will help us label the dots without overwriting other labels. So we'll have to install that in the console:

'install.packages("ggrepel")

```
library(ggrepel)
```

And with that, we can add labels to the dots. The `geom_text_repel` is pretty much the exact same thing as your Big Ten geom point, but instead of a size, you include a label.

```r
ggplot() +
  geom_point(
    data=ypp,
    aes(x=OffensiveYPP, y=DefensiveYPP, size=TotalWins),
    color="grey",
    alpha=.5) +
  geom_point(
    data=bigten,
    aes(x=OffensiveYPP, y=DefensiveYPP, size=TotalWins),
    color="red") +
  geom_vline(xintercept = 5.863151) +
  geom_hline(yintercept = 5.448228) +
  geom_text_repel(
    data=bigten,
    aes(x=OffensiveYPP, y=DefensiveYPP, label=Team)
  )
```

Well, what do you know about that? Maryland was … really a mixed bag this season.

All that's left is some labels and some finishing touches.

```
ggplot() +
  geom_point(
    data=ypp,
    aes(x=OffensiveYPP, y=DefensiveYPP, size=TotalWins),
    color="grey",
    alpha=.5) +
  geom_point(
    data=bigten,
    aes(x=OffensiveYPP, y=DefensiveYPP, size=TotalWins),
    color="red") +
  geom_vline(xintercept = 5.863151) +
  geom_hline(yintercept = 5.448228) +
  geom_text_repel(
    data=bigten,
    aes(x=OffensiveYPP, y=DefensiveYPP, label=Team)
  ) +
  labs(title="Is Maryland moving up to the Big Ten's best?", subtitle="The Terps offense a
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_text(size=10),
    panel.grid.minor = element_blank()
    )
```

Is Maryland moving up to the Big Ten's best?
The Terps offense and defense puts it among ranked teams in the conference.
Northwestern
Rutgers
Indiana
Illinois
Maryland
Purdue
Michigan State
Nebraska
Ohio State
Minnesota
Iowa
Michigan
Penn State
Wisconsin
DefensiveYPP
OffensiveYPP
4
5
6
7
8
TotalWins
2.5
5.0
7.5
10.0
12.5
Source: NCAA | By Derek Willis

23 Beeswarm plots

A beeswarm plot is sometimes called a column scatterplot. It's an effective way to show how individual things – teams, players, etc. – are distributed along a numberline. The column is a grouping – say positions in basketball – and the dots are players, and the dots cluster where the numbers are more common. So think of it like a histogram mixed with a scatterplot crossed with a bar chart.

An example will help.

First things first: Install ggbeeswarm with `install.packages("ggbeeswarm")`

Like ggalt and ggrepel, ggbeeswarm adds a couple new geoms to ggplot. We'll need to load it, the tidyverse and, for later, ggrepel.

```
library(tidyverse)
library(ggbeeswarm)
library(ggrepel)
```

Another bit of setup: we need to set the seed for the random number generator. The library "jitters" the dots in the beeswarm randomly. If we don't set the seed, we'll get different results each time. Setting the seed means we get the same look.

```
set.seed(1234)
```

So let's look at last year's women's basketball team as a group of shooters. The team didn't have the kind of success that was envisioned – we know that – but what kind of a problem is it going to be that we're returning one regular starter (Diamond Miller) from it?

First we'll load our player data.

```
players <- read_csv("data/wbb_players_2022.csv")
```

```
Rows: 4781 Columns: 174
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (23): position, team, ncaa_conference, player, fg_pct, fg2m_pct, fg3m_p...
dbl (151): end_year, ncaa_id, win, loss, gp, gs, ga, win_pct, mp, sec, sec_t...
```

```
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.
```

We know this data has a lot of players who didn't play, so let's get rid of them.

```
activeplayers <- players %>% filter(mp>0)
```

Now let's ask what makes a good shooter? The best measure, in my book, is True Shooting Percentage. It's a combination of weighted field goal shooting – to account for three pointers – and free throws. Let's calculate that and add it to our data.

```
activeplayers <- activeplayers %>% mutate(tspct=pts/(2*(fga+0.44*fta)))
```

If we include *all* players, we'll have too many dots. So let's narrow it down. A decent tool for cutoffs? Field goal attempts. Let's get a quick look at them.

```
summary(activeplayers$fga)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
    0.0    28.0    99.0   127.7   197.0   691.0 
```

The median number of shots is 99, but we only really care about good ones. So let's use 197 attempts – the third quartile – as our cutoff.

```
shooters <- activeplayers %>% filter(fga > 197)
```

Now we've got enough for a beeswarm plot. It works very much like you would expect – the group value is the x, the number is the y. We're going to beeswarm by position, and the dots will be true shooting percentage.

```
ggplot() + geom_beeswarm(data=shooters, aes(x=position, y=tspct), color="grey")
```

You can see that there's a lot fewer centers who have attempted more than 152 shots than guards, but then there's a lot more guards in college basketball than anything else. In the guards column, note that fat width of the swarm is between .5 and .6. So that means most guards who shoot more than 197 shots end up in that area. They're the average shooter at that level. You can see, some are better, some are worse.

So where are the Maryland players in that mix?

We'll filter players on Maryland who meet our criteria.

```
umd <- activeplayers %>%
  filter(team == "Maryland") %>%
  filter(fga>197) %>%
  arrange(desc(tspct))
```

Seven Cornhuskers took more than 197 shots. Number returning this season? Two.

But how good are they as true shooters?

When you add another beeswarm, we need to pass another element in – we need to tell it if we're grouping on the x value. Not sure why, but you'll get a warning if you don't.

```
ggplot() +
  geom_beeswarm(
    data=shooters,
    groupOnX=TRUE,
    aes(x=position, y=tspct), color="grey") +
  geom_beeswarm(
    data=umd,
    groupOnX=TRUE,
    aes(x=position, y=tspct), color="red")
```

One very good guard, and some good forwards. Who are they?

This is where we can use ggrepel. Let's add a text layer and label the dots.

```
ggplot() +
  geom_beeswarm(
    data=shooters,
    groupOnX=TRUE,
    aes(x=position, y=tspct), color="grey") +
  geom_beeswarm(
    data=umd,
```

```
    groupOnX=TRUE,
    aes(x=position, y=tspct), color="red") +
  geom_text_repel(
    data=umd,
    aes(x=position, y=tspct, label=player))
```

So Katie Benzan was our best shooter by true shooting percentage. Most of the rest were at or above average shooters for that volume of shooting, but Miller and Sellers are the returnees, and both were near average.

23.1 A few other options

The ggbeeswarm library has a couple of variations on the geom_beeswarm that may work better for your application. They are `geom_quasirandom` and `geom_jitter`.

There's not a lot to change from our example to see what they do.

```
ggplot() +
  geom_quasirandom(
    data=shooters,
```

```
    groupOnX=TRUE,
    aes(x=position, y=tspct), color="grey") +
  geom_quasirandom(
    data=umd,
    groupOnX=TRUE,
    aes(x=position, y=tspct), color="red") +
  geom_text_repel(
    data=umd,
    aes(x=position, y=tspct, label=player))
```

Quasirandom spreads out the dots you see in beeswarm using – you guessed it – quasirandom spacing.

For `geom_jitter`, we need to remove the groupOnX value. Why? No clue.

```
ggplot() +
  geom_jitter(
    data=shooters,
    aes(x=position, y=tspct), color="grey") +
  geom_jitter(
    data=umd,
```

```
  aes(x=position, y=tspct), color="red") +
geom_text_repel(
  data=umd,
  aes(x=position, y=tspct, label=player))
```

`geom_jitter` spreads out the dots evenly across the width of the column, randomly deciding where in the line of the true shooting percentage they appear.

Which one is right for you? You're going to have to experiment and decide. This is the art in the art and a science.

24 Bump charts

The point of a bump chart is to show how the ranking of something changed over time – you could do this with the top 25 in football or basketball. I've seen it done with European soccer league standings over a season.

The requirements are that you have a row of data for a team, in that week, with their rank.

This is another extension to ggplot, and you'll install it the usual way: `install.packages("ggbump")`

```
library(tidyverse)
library(ggbump)
```

Let's use the 2020-21 college football playoff rankings:

```
rankings <- read_csv("data/cfbranking.csv")
```

```
Rows: 125 Columns: 4
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (2): Team, ShortTeam
dbl (2): Rank, Week

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.
```

Given our requirements of a row of data for a team, in that week, with their rank, take a look at the data provided. We have 5 weeks of playoff rankings, so we should see a ranking, the week of the ranking and the team at that rank. You can see the basic look of the data by using head()

```
head(rankings)
```

```
# A tibble: 6 x 4
   Rank  Week Team    ShortTeam
  <dbl> <dbl> <chr>   <chr>
```

```
1      1    17 Alabama Ala.
2      1    16 Alabama Ala.
3      1    15 Alabama Ala.
4      1    14 Alabama Ala.
5      1    13 Alabama Ala.
6     22    13 Auburn  Aub.
```

So Alabama was ranked in the first (yawn), followed by Clemson (double yawn), Ohio State and so on. Our data is in the form we need it to be. Now we can make a bump chart. We'll start simple.

```
ggplot() +
  geom_bump(
    data=rankings, aes(x=Week, y=Rank, color=Team))
```

```
Warning in f(...): 'StatBump' needs at least two observations per group

Warning in f(...): 'StatBump' needs at least two observations per group

Warning in f(...): 'StatBump' needs at least two observations per group
```

Well, it's a start.

The warning that you're seeing is that there's three teams last season who made one appearance on the college football playoff rankings and disappeared. Some fans would bite your arm off for that. Alas. We should eliminate them and thin up our chart a little. Let's just take teams that finished in the top 10. We're going to use a neat filter trick for this that you learned earlier using %in%.

```
top10 <- rankings %>% filter(Week == 17 & Rank <= 10)

newrankings <- rankings %>% filter(Team %in% top10$Team)
```

Now you have something called newrankings that shows how teams who finished in the top 10 at the end of the season ended up there. And every team who finished in the top 10 in week 17 had been in the rankings more than once in the 5 weeks before.

```
ggplot() +
  geom_bump(
    data=newrankings, aes(x=Week, y=Rank, color=Team))
```

First things first: I'm immediately annoyed by the top teams being at the bottom. I learned a neat trick from ggbump that's been in ggplot all along – `scale_y_reverse()`

```
ggplot() +
  geom_bump(
    data=newrankings, aes(x=Week, y=Rank, color=Team)) +
  scale_y_reverse()
```

Better. But, still not great. Let's add a point at each week.

```
ggplot() +
  geom_bump(data=newrankings, aes(x=Week, y=Rank, color=Team)) +
  geom_point(data=newrankings, aes(x=Week, y=Rank, color=Team), size = 4) +
  scale_y_reverse()
```

Another step. That makes it more subway-map like. But the colors are all wrong. To fix this, we're going to use `scale_color_manual` and we're going to Google the hex codes for each team. The legend will tell you what order your `scale_color_manual` needs to be.

```
ggplot() +
  geom_bump(data=newrankings, aes(x=Week, y=Rank, color=Team)) +
  geom_point(data=newrankings, aes(x=Week, y=Rank, color=Team), size = 4) +
  scale_color_manual(values = c("#9E1B32","#E00122", "#F56600", "#0021A5", "#BA0C2F", "#F1
  scale_y_reverse()
```

Another step. But the legend is annoying. And trying to find which red is Alabama vs Ohio State is hard. So what if we labeled each dot at the beginning and end? We can do that with some clever usage of geom_text and a little dplyr filtering inside the data step. We filter out the first and last weeks, then use hjust – horizontal justification – to move them left or right.

```
ggplot() +
  geom_bump(data=newrankings, aes(x=Week, y=Rank, color=Team)) +
  geom_point(data=newrankings, aes(x=Week, y=Rank, color=Team), size = 4) +
  geom_text(data = newrankings %>% filter(Week == min(Week)), aes(x = Week - .2, y=Rank, l
  geom_text(data = newrankings %>% filter(Week == max(Week)), aes(x = Week + .2, y=Rank, l
  scale_color_manual(values = c("#9E1B32","#E00122", "#F56600", "#0021A5", "#BA0C2F", "#F1
  scale_y_reverse()
```

Better, but the legend is still there. We can drop it in a theme directive by saying `legend.position = "none"`. We'll also throw a theme_minimal on there to drop the default grey, and we'll add some better labeling.

```
ggplot() +
  geom_bump(data=newrankings, aes(x=Week, y=Rank, color=Team)) +
  geom_point(data=newrankings, aes(x=Week, y=Rank, color=Team), size = 4) +
  geom_text(data = newrankings %>% filter(Week == min(Week)), aes(x = Week - .2, y=Rank, l
  geom_text(data = newrankings %>% filter(Week == max(Week)), aes(x = Week + .2, y=Rank, l
  labs(title="Was COVID college football boring?", subtitle="There was no drama at the top
  theme_minimal() +
  theme(
    legend.position = "none",
    panel.grid.major = element_blank()
    ) +
  scale_color_manual(values = c("#9E1B32","#E00122", "#F56600", "#0021A5", "#BA0C2F", "#F1
  scale_y_reverse()
```

Now let's fix our text hierarchy.

```
ggplot() +
  geom_bump(data=newrankings, aes(x=Week, y=Rank, color=Team)) +
  geom_point(data=newrankings, aes(x=Week, y=Rank, color=Team), size = 4) +
  geom_text(data = newrankings %>% filter(Week == min(Week)), aes(x = Week - .2, y=Rank, l
  geom_text(data = newrankings %>% filter(Week == max(Week)), aes(x = Week + .2, y=Rank, l
  labs(title="Was COVID college football boring?", subtitle="There was no drama at the top
  theme_minimal() +
  theme(
    legend.position = "none",
    panel.grid.major = element_blank(),
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_text(size=10),
    panel.grid.minor = element_blank()
    ) +
  scale_color_manual(values = c("#9E1B32","#E00122", "#F56600", "#0021A5", "#BA0C2F", "#F1
  scale_y_reverse()
```

And the last thing: anyone else annoyed at 7.5th place on the left? We can fix that too by specifying the breaks in scale_y_reverse. We can do that with the x axis as well, but since we haven't reversed it, we do that in `scale_x_continuous` with the same breaks. Also: forgot my source and credit line.

One last thing: Let's change the width of the chart to make the names fit. We can do that by adding `fig.width=X` in the `{r}` setup in your block. So something like this:

```
ggplot() +
  geom_bump(data=newrankings, aes(x=Week, y=Rank, color=Team)) +
  geom_point(data=newrankings, aes(x=Week, y=Rank, color=Team), size = 4) +
  geom_text(data = newrankings %>% filter(Week == min(Week)), aes(x = Week - .2, y=Rank, l
  geom_text(data = newrankings %>% filter(Week == max(Week)), aes(x = Week + .2, y=Rank, l
  labs(title="Was COVID college football boring?", subtitle="There was no drama at the top
  theme_minimal() +
  theme(
    legend.position = "none",
    panel.grid.major = element_blank(),
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_text(size=10),
    panel.grid.minor = element_blank()
```

```
  ) +
scale_color_manual(values = c("#9E1B32","#E00122", "#F56600", "#0021A5", "#BA0C2F", "#F1
scale_x_continuous(breaks=c(13,14,15,16,17)) +
scale_y_reverse(breaks=c(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15))
```

25 Tables

But not a table. A table with features.

Sometimes, the best way to show your data is with a table – simple rows and columns. It allows a reader to compare whatever they want to compare a little easier than a graph where you've chosen what to highlight. The folks that made R Studio and the tidyverse have a neat package called `gt`.

For this assignment, we'll need `gt` so go over to the console and run:

```
install.packages("gt")
```

So what does all of these libraries do? Let's gather a few and use data of every men's basketball game between 2015-2020.

Load libraries.

```
library(tidyverse)
library(gt)
```

And the data.

```
logs <- read_csv("data/logs1520.csv")
```

```
Rows: 68617 Columns: 44
-- Column specification --------------------------------------------------------
Delimiter: ","
chr   (6): HomeAway, Opponent, W_L, Team, Conference, season
dbl  (36): X1, Game, TeamScore, OpponentScore, TeamFG, TeamFGA, TeamFGPCT, T...
lgl   (1): Blank
date  (1): Date

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.
```

Let's ask this question: which college basketball team saw the greatest increase in three point attempts last season as a percentage of shots? The simplest way to calculate that is by percent change.

We've got a little work to do, putting together ideas we've used before. What we need to end up with is some data that looks like this:

```
Team | 2018-2019 season threes | 2019-2020 season threes | pct change
```

To get that, we'll need to do some filtering to get the right seasons, some grouping and summarizing to get the right number, some pivoting to get it organized correctly so we can mutate the percent change.

```
threechange <- logs %>%
  filter(season == "2018-2019" | season == "2019-2020") %>%
  group_by(Team, Conference, season) %>%
  summarise(Total3PA = sum(Team3PA)) %>%
  pivot_wider(names_from=season, values_from = Total3PA) %>%
  mutate(PercentChange = (`2019-2020`-`2018-2019`)/`2018-2019`) %>%
  arrange(desc(PercentChange)) %>%
  ungroup() %>%
  top_n(10) # just want a top 10 list
```

```
`summarise()` has grouped output by 'Team', 'Conference'. You can override
using the `.groups` argument.
Selecting by PercentChange
```

We've output tables to the screen a thousand times in this class with `head`, but `gt` makes them look decent with very little code.

```
threechange %>% gt()
```

Team	Conference	2018-2019	2019-2020	PercentChange
Mississippi Valley State Delta Devils	SWAC	554	837	0.5108303
Valparaiso Crusaders	MVC	585	843	0.4410256
Ball State Cardinals	MAC	621	842	0.3558776
San Jose State Spartans	MWC	641	861	0.3432137
Alabama Crimson Tide	SEC	718	957	0.3328691
Minnesota Golden Gophers	Big Ten	603	762	0.2636816
Georgia Southern Eagles	Sun Belt	631	792	0.2551506
Tennessee Tech Golden Eagles	OVC	620	776	0.2516129
San Francisco Dons	WCC	728	899	0.2348901

McNeese State Cowboys	Southland	547	675	0.2340037

So there you have it. Mississippi Valley State changed their team so much they took 51 percent more threes in 2019-20 from the season before. Where did Maryland come out? We ranked 49th in college basketball in terms of change from the season before.

`gt` has a mountain of customization options. The good news is that it works in a very familiar pattern. We'll start with fixing headers. What we have isn't bad, but PercentChange isn't good either. Let's fix that.

```
threechange %>%
  gt() %>%
  cols_label(
    PercentChange = "Percent Change"
  )
```

Team	Conference	2018-2019	2019-2020	Percent Change
Mississippi Valley State Delta Devils	SWAC	554	837	0.5108303
Valparaiso Crusaders	MVC	585	843	0.4410256
Ball State Cardinals	MAC	621	842	0.3558776
San Jose State Spartans	MWC	641	861	0.3432137
Alabama Crimson Tide	SEC	718	957	0.3328691
Minnesota Golden Gophers	Big Ten	603	762	0.2636816
Georgia Southern Eagles	Sun Belt	631	792	0.2551506
Tennessee Tech Golden Eagles	OVC	620	776	0.2516129
San Francisco Dons	WCC	728	899	0.2348901
McNeese State Cowboys	Southland	547	675	0.2340037

Better. Note the pattern: Actual header name = "What we want to see". So if we wanted to change Team to School, we'd do this: `Team = "School"` inside the `cols_label` bits.

Now we can start working with styling. The truth is most of your code in tables is going to be dedicated to styling specific things. The first thing we need: A headline and some chatter. They're required parts of a graphic, so they're a good place to start. We do that with `tab_header`

```
threechange %>%
  gt() %>%
  cols_label(
    PercentChange = "Percent Change"
  ) %>%
  tab_header(
```

```
    title = "Did Maryland Shoot More Threes in 2019-20?",
    subtitle = "Yes, but it wasn't a big change. These 10 teams completely changed their o
  )
```

Did Maryland Shoot More Threes in 2019-20?
Yes, but it wasn't a big change. These 10 teams completely changed their offenses.

Team	Conference	2018-2019	2019-2020	Percent Change
Mississippi Valley State Delta Devils	SWAC	554	837	0.5108303
Valparaiso Crusaders	MVC	585	843	0.4410256
Ball State Cardinals	MAC	621	842	0.3558776
San Jose State Spartans	MWC	641	861	0.3432137
Alabama Crimson Tide	SEC	718	957	0.3328691
Minnesota Golden Gophers	Big Ten	603	762	0.2636816
Georgia Southern Eagles	Sun Belt	631	792	0.2551506
Tennessee Tech Golden Eagles	OVC	620	776	0.2516129
San Francisco Dons	WCC	728	899	0.2348901
McNeese State Cowboys	Southland	547	675	0.2340037

We have a headline and some chatter, but … gross. Centered? The extra lines? No real difference in font weight? We can do better. We can style individual elements using `tab_style`. First, let's make the main headline – the `title` – bold and left aligned.

```
threechange %>%
  gt() %>%
  cols_label(
    PercentChange = "Percent Change"
  ) %>%
  tab_header(
    title = "Did Maryland Shoot More Threes in 2019-20?",
    subtitle = "Yes, but it wasn't a big change. These 10 teams completely changed their o
  ) %>% tab_style(
    style = cell_text(color = "black", weight = "bold", align = "left"),
    locations = cells_title("title")
  )
```

Did Maryland Shoot More Threes in 2019-20?
Yes, but it wasn't a big change. These 10 teams completely changed their offenses.

Team	Conference	2018-2019	2019-2020	Percent Change

Mississippi Valley State Delta Devils	SWAC	554	837	0.5108303
Valparaiso Crusaders	MVC	585	843	0.4410256
Ball State Cardinals	MAC	621	842	0.3558776
San Jose State Spartans	MWC	641	861	0.3432137
Alabama Crimson Tide	SEC	718	957	0.3328691
Minnesota Golden Gophers	Big Ten	603	762	0.2636816
Georgia Southern Eagles	Sun Belt	631	792	0.2551506
Tennessee Tech Golden Eagles	OVC	620	776	0.2516129
San Francisco Dons	WCC	728	899	0.2348901
McNeese State Cowboys	Southland	547	675	0.2340037

It's hard to see here, but the chatter below is also centered (it doesn't look like it because it fills the space). We can left align that too, but leave it normal weight (i.e. not bold).

```
threechange %>%
  gt() %>%
  cols_label(
    PercentChange = "Percent Change"
  ) %>%
  tab_header(
    title = "Did Maryland Shoot More Threes in 2019-20?",
    subtitle = "Yes, but it wasn't a big change. These 10 teams completely changed their o
  ) %>% tab_style(
    style = cell_text(color = "black", weight = "bold", align = "left"),
    locations = cells_title("title")
  ) %>% tab_style(
    style = cell_text(color = "black", align = "left"),
    locations = cells_title("subtitle")
  )
```

Did Maryland Shoot More Threes in 2019-20?

Yes, but it wasn't a big change. These 10 teams completely changed their offenses.

Team	Conference	2018-2019	2019-2020	Percent Change
Mississippi Valley State Delta Devils	SWAC	554	837	0.5108303
Valparaiso Crusaders	MVC	585	843	0.4410256
Ball State Cardinals	MAC	621	842	0.3558776
San Jose State Spartans	MWC	641	861	0.3432137
Alabama Crimson Tide	SEC	718	957	0.3328691
Minnesota Golden Gophers	Big Ten	603	762	0.2636816
Georgia Southern Eagles	Sun Belt	631	792	0.2551506

Tennessee Tech Golden Eagles	OVC	620	776	0.2516129
San Francisco Dons	WCC	728	899	0.2348901
McNeese State Cowboys	Southland	547	675	0.2340037

The next item on the required elements list: Source and credit lines. In `gt`, those are called `tab_source_notes` and we can add them like this:

```
threechange %>%
  gt() %>%
  cols_label(
    PercentChange = "Percent Change"
  ) %>%
  tab_header(
    title = "Did Maryland Shoot More Threes in 2019-20?",
    subtitle = "Yes, but it wasn't a big change. These 10 teams completely changed their o
  ) %>% tab_style(
    style = cell_text(color = "black", weight = "bold", align = "left"),
    locations = cells_title("title")
  ) %>% tab_style(
    style = cell_text(color = "black", align = "left"),
    locations = cells_title("subtitle")
  ) %>%
  tab_source_note(
    source_note = md("**By:** Derek Willis  |  **Source:** [Sports Reference](https://www.
  )
```

Did Maryland Shoot More Threes in 2019-20?

Yes, but it wasn't a big change. These 10 teams completely changed their offenses.

Team	Conference	2018-2019	2019-2020	Percent Change
Mississippi Valley State Delta Devils	SWAC	554	837	0.5108303
Valparaiso Crusaders	MVC	585	843	0.4410256
Ball State Cardinals	MAC	621	842	0.3558776
San Jose State Spartans	MWC	641	861	0.3432137
Alabama Crimson Tide	SEC	718	957	0.3328691
Minnesota Golden Gophers	Big Ten	603	762	0.2636816
Georgia Southern Eagles	Sun Belt	631	792	0.2551506
Tennessee Tech Golden Eagles	OVC	620	776	0.2516129
San Francisco Dons	WCC	728	899	0.2348901
McNeese State Cowboys	Southland	547	675	0.2340037

By: Derek Willis | **Source:** Sports Reference

We can do a lot with `tab_style`. For instance, we can make the headers bold and reduce the size a bit to reduce font congestion in the area.

```
threechange %>%
  gt() %>%
  cols_label(
    PercentChange = "Percent Change"
  ) %>%
  tab_header(
    title = "Did Maryland Shoot More Threes in 2019-20?",
    subtitle = "Yes, but it wasn't a big change. These 10 teams completely changed their o
  ) %>%
  tab_style(
    style = cell_text(color = "black", weight = "bold", align = "left"),
    locations = cells_title("title")
  ) %>%
  tab_style(
    style = cell_text(color = "black", align = "left"),
    locations = cells_title("subtitle")
  ) %>%
  tab_source_note(
    source_note = md("**By:** Derek Willis  |  **Source:** [Sports Reference](https://www.
  ) %>%
  tab_style(
     locations = cells_column_labels(columns = everything()),
     style = list(
       cell_borders(sides = "bottom", weight = px(3)),
       cell_text(weight = "bold", size=12)
     )
   )
```

Did Maryland Shoot More Threes in 2019-20?

Yes, but it wasn't a big change. These 10 teams completely changed their offenses.

Team	Conference	2018-2019	2019-2020	Percent Change
Mississippi Valley State Delta Devils	SWAC	554	837	0.5108303
Valparaiso Crusaders	MVC	585	843	0.4410256
Ball State Cardinals	MAC	621	842	0.3558776
San Jose State Spartans	MWC	641	861	0.3432137
Alabama Crimson Tide	SEC	718	957	0.3328691
Minnesota Golden Gophers	Big Ten	603	762	0.2636816
Georgia Southern Eagles	Sun Belt	631	792	0.2551506

Tennessee Tech Golden Eagles	OVC	620	776	0.2516129
San Francisco Dons	WCC	728	899	0.2348901
McNeese State Cowboys	Southland	547	675	0.2340037

By: Derek Willis | **Source:** Sports Reference

Next up: There's a lot of lines in this that don't need to be there. `gt` has some tools to get rid of them easily and add in some other readability improvements.

```
threechange %>%
  gt() %>%
  cols_label(
    PercentChange = "Percent Change"
  ) %>%
  tab_header(
    title = "Did Maryland Shoot More Threes in 2019-20?",
    subtitle = "Yes, but it wasn't a big change. These 10 teams completely changed their o
  ) %>%
  tab_source_note(
    source_note = md("**By:** Derek Willis  |  **Source:** [Sports Reference](https://www.
  ) %>%
  tab_style(
    style = cell_text(color = "black", weight = "bold", align = "left"),
    locations = cells_title("title")
  ) %>%
  tab_style(
    style = cell_text(color = "black", align = "left"),
    locations = cells_title("subtitle")
  ) %>%
  tab_style(
     locations = cells_column_labels(columns = everything()),
     style = list(
       cell_borders(sides = "bottom", weight = px(3)),
       cell_text(weight = "bold", size=12)
     )
   ) %>%
  opt_row_striping() %>%
  opt_table_lines("none")
```

Did Maryland Shoot More Threes in 2019-20?

Yes, but it wasn't a big change. These 10 teams completely changed their offenses.

Team	Conference	2018-2019	2019-2020	Percent Change
Mississippi Valley State Delta Devils	SWAC	554	837	0.5108303
Valparaiso Crusaders	MVC	585	843	0.4410256
Ball State Cardinals	MAC	621	842	0.3558776
San Jose State Spartans	MWC	641	861	0.3432137
Alabama Crimson Tide	SEC	718	957	0.3328691
Minnesota Golden Gophers	Big Ten	603	762	0.2636816
Georgia Southern Eagles	Sun Belt	631	792	0.2551506
Tennessee Tech Golden Eagles	OVC	620	776	0.2516129
San Francisco Dons	WCC	728	899	0.2348901
McNeese State Cowboys	Southland	547	675	0.2340037

By: Derek Willis | **Source:** Sports Reference

We're in pretty good shape here, but look closer. What else makes this table sub-par? How about the formatting of the percent change? We can fix that with a formatter.

```
threechange %>%
  gt() %>%
  cols_label(
    PercentChange = "Percent Change"
  ) %>%
  tab_header(
    title = "Did Maryland Shoot More Threes in 2019-20?",
    subtitle = "Yes, but it wasn't a big change. These 10 teams completely changed their o
  ) %>%
  tab_source_note(
    source_note = md("**By:** Derek Willis  |  **Source:** [Sports Reference](https://www.
  ) %>%
  tab_style(
    style = cell_text(color = "black", weight = "bold", align = "left"),
    locations = cells_title("title")
  ) %>%
  tab_style(
    style = cell_text(color = "black", align = "left"),
    locations = cells_title("subtitle")
  ) %>%
  tab_style(
     locations = cells_column_labels(columns = everything()),
     style = list(
       cell_borders(sides = "bottom", weight = px(3)),
```

```
      cell_text(weight = "bold", size=12)
    )
  ) %>%
opt_row_striping() %>%
opt_table_lines("none") %>%
  fmt_percent(
  columns = c(PercentChange),
  decimals = 1
)
```

Did Maryland Shoot More Threes in 2019-20?

Yes, but it wasn't a big change. These 10 teams completely changed their offenses.

Team	Conference	2018-2019	2019-2020	Percent Change
Mississippi Valley State Delta Devils	SWAC	554	837	51.1%
Valparaiso Crusaders	MVC	585	843	44.1%
Ball State Cardinals	MAC	621	842	35.6%
San Jose State Spartans	MWC	641	861	34.3%
Alabama Crimson Tide	SEC	718	957	33.3%
Minnesota Golden Gophers	Big Ten	603	762	26.4%
Georgia Southern Eagles	Sun Belt	631	792	25.5%
Tennessee Tech Golden Eagles	OVC	620	776	25.2%
San Francisco Dons	WCC	728	899	23.5%
McNeese State Cowboys	Southland	547	675	23.4%

By: Derek Willis | **Source:** Sports Reference

Throughout the semester, we've been using color and other signals to highlight things. Let's pretend we're doing a project on Minnesota. Note they're the only Big Ten team on this list. With a little `tab_style` magic, we can change individual rows and add color. The last `tab_style` block here will first pass off the styles we want to use – we're going to make the rows maroon and the text gold – and then for locations we specify where with a simple filter. What that means is that any rows we can address with logic – all rows with a value greater than X, for example – we can change the styling.

```
threechange %>%
  gt() %>%
  cols_label(
    PercentChange = "Percent Change"
  ) %>%
  tab_header(
```

```
    title = "Did Maryland Shoot More Threes in 2019-20?",
    subtitle = "Yes, but it wasn't a big change. These 10 teams completely changed their o
  ) %>%
  tab_source_note(
    source_note = md("**By:** Derek Willis  |  **Source:** [Sports Reference](https://www.
  ) %>%
  tab_style(
    style = cell_text(color = "black", weight = "bold", align = "left"),
    locations = cells_title("title")
  ) %>%
  tab_style(
    style = cell_text(color = "black", align = "left"),
    locations = cells_title("subtitle")
  ) %>%
  tab_style(
     locations = cells_column_labels(columns = everything()),
     style = list(
       cell_borders(sides = "bottom", weight = px(3)),
       cell_text(weight = "bold", size=12)
     )
   ) %>%
  opt_row_striping() %>%
  opt_table_lines("none") %>%
    fmt_percent(
    columns = c(PercentChange),
    decimals = 1
  ) %>%
  tab_style(
    style = list(
      cell_fill(color = "maroon"),
      cell_text(color = "gold")
      ),
    locations = cells_body(
      rows = Team == "Minnesota Golden Gophers")
  )
```

Did Maryland Shoot More Threes in 2019-20?

Yes, but it wasn't a big change. These 10 teams completely changed their offenses.

Team	Conference	2018-2019	2019-2020	Percent Change
Mississippi Valley State Delta Devils	SWAC	554	837	51.1%

Valparaiso Crusaders	MVC	585	843	44.1%
Ball State Cardinals	MAC	621	842	35.6%
San Jose State Spartans	MWC	641	861	34.3%
Alabama Crimson Tide	SEC	718	957	33.3%
Minnesota Golden Gophers	Big Ten	603	762	26.4%
Georgia Southern Eagles	Sun Belt	631	792	25.5%
Tennessee Tech Golden Eagles	OVC	620	776	25.2%
San Francisco Dons	WCC	728	899	23.5%
McNeese State Cowboys	Southland	547	675	23.4%

By: Derek Willis | **Source:** Sports Reference

Two things here:

1. Dear God that color scheme is awful, which is fitting for a school that worships a lawn-wrecking varmint.
2. We've arrived where we want to be: We've created a clear table that allows a reader to compare schools at will while also using color to draw attention to the thing we want to draw attention to. We've kept it simple so the color has impact.

26 Facet wraps

Sometimes the easiest way to spot a trend is to chart a bunch of small things side by side. Edward Tufte, one of the most well known data visualization thinkers on the planet, calls this "small multiples" where ggplot calls this a facet wrap or a facet grid, depending.

One thing we noticed earlier in the semester – it seems that a lot of teams shoot worse as the season goes on. Do they? We could answer this a number of ways, but the best way to show people would be visually. Let's use Small Multiples.

As always, we start with libraries.

```
library(tidyverse)
```

We're going to use the logs of college basketball games last season.

And load it.

```
logs <- read_csv("data/logs22.csv")
```

```
Rows: 10775 Columns: 48
-- Column specification --------------------------------------------------------
Delimiter: ","
chr   (8): Season, TeamFull, Opponent, HomeAway, W_L, URL, Conference, Team
dbl  (39): Game, TeamScore, OpponentScore, TeamFG, TeamFGA, TeamFGPCT, Team3...
date  (1): Date

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.
```

Let's narrow our pile and look just at the Big Ten.

```
big10 <- logs %>% filter(Conference == "Big Ten")
```

The first thing we can do is look at a line chart, like we have done in previous chapters.

```
ggplot() +
  geom_line(data=big10, aes(x=Date, y=TeamFGPCT, group=Team)) +
  scale_y_continuous(limits = c(0, .7))
```

And, not surprisingly, we get a hairball. We could color certain lines, but that would limit us to focus on one team. What if we did all of them at once? We do that with a `facet_wrap`. The only thing we MUST pass into a `facet_wrap` is what thing we're going to separate them out by. In this case, we precede that field with a tilde, so in our case we want the Team field. It looks like this:

```
ggplot() +
  geom_line(data=big10, aes(x=Date, y=TeamFGPCT, group=Team)) +
  scale_y_continuous(limits = c(0, .7)) +
  facet_wrap(~Team)
```

Answer: Not immediately clear, but we can look at this and analyze it. We could add a piece of annotation to help us out.

```
big10 %>% summarise(mean(TeamFGPCT))
```

```
# A tibble: 1 x 1
  `mean(TeamFGPCT)`
              <dbl>
1             0.453
```

```
ggplot() +
  geom_hline(yintercept=0.4532143, color="blue") +
  geom_line(data=big10, aes(x=Date, y=TeamFGPCT, group=Team)) +
  scale_y_continuous(limits = c(0, .7)) +
  facet_wrap(~Team)
```

What do you see here? How do teams compare? How do they change over time? I'm not asking you these questions because they're an assignment – I'm asking because that's exactly what this chart helps answer. Your brain will immediately start making those connections.

26.1 Facet grid vs facet wraps

Facet grids allow us to put teams on the same plane, versus just repeating them. And we can specify that plane as vertical or horizontal. For example, here's our chart from above, but using facet_grid to stack them.

```
ggplot() +
  geom_hline(yintercept=0.4532143, color="blue") +
  geom_line(data=big10, aes(x=Date, y=TeamFGPCT, group=Team)) +
  scale_y_continuous(limits = c(0, .7)) +
  facet_grid(Team ~ .)
```

And here they are next to each other:

```
ggplot() +
  geom_hline(yintercept=0.4532143, color="blue") +
  geom_line(data=big10, aes(x=Date, y=TeamFGPCT, group=Team)) +
  scale_y_continuous(limits = c(0, .7)) +
  facet_grid(. ~ Team)
```

Note: We'd have some work to do with the labeling on this – we'll get to that – but you can see where this is valuable comparing a group of things. One warning: Don't go too crazy with this or it loses it's visual power.

26.2 Other types

Line charts aren't the only things we can do. We can do any kind of chart in ggplot. Staying with shooting, where are team's winning and losing performances coming from when we talk about team shooting and opponent shooting?

```
ggplot() +
  geom_point(data=big10, aes(x=TeamFGPCT, y=OpponentFGPCT, color=W_L)) +
  scale_y_continuous(limits = c(0, .7)) +
  scale_x_continuous(limits = c(0, .7)) +
  facet_wrap(~Team)
```

Illinois
Indiana
Iowa
Maryland
Michigan
Michigan State
Minnesota
Nebraska
Northwestern
Ohio State
Penn State
Purdue
Rutgers
Wisconsin
OpponentFGPCT
TeamFGPCT
0.0
0.2
0.4
0.6
W_L
L
L (1 OT)
L (2 OT)
L (4 OT)
W
W (1 OT)
W (2 OT)

27 Arranging multiple plots together

Sometimes you have two or three (or more) charts that by themselves aren't very exciting and are really just one chart that you need to merge together. It would be nice to be able to arrange them programmatically and not have to mess with it in Adobe Illustrator.

Good news.

There is.

It's called `cowplot`, and it's pretty easy to use. First install cowplot with `install.packages("cowplot")`. Then let's load tidyverse and cowplot.

```
library(tidyverse)
library(cowplot)
```

We'll use the college football attendance data we've used before.

And load it.

```
attendance <- read_csv("data/attendance.csv")
```

```
Rows: 149 Columns: 9
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (2): Institution, Conference
dbl (7): 2013, 2014, 2015, 2016, 2017, 2018, 2019

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.
```

Making a quick percent change.

```
attendance <- attendance %>% mutate(change = ((`2019`-`2018`)/`2018`)*100)
```

Let's chart the top 10 and bottom 10 of college football ticket growth … and shrinkage.

```
top10 <- attendance %>% top_n(10, wt=change)
bottom10 <- attendance %>% top_n(10, wt=-change)
```

Okay, now to do this I need to **save my plots to an object**. We do this the same way we save things to a dataframe – with the arrow. We'll make two identical bar charts, one with the top 10 and one with the bottom 10.

```
bar1 <- ggplot() +
  geom_bar(data=top10, aes(x=reorder(Institution, change), weight=change)) +
  coord_flip()
```

```
bar2 <- ggplot() +
  geom_bar(data=bottom10, aes(x=reorder(Institution, change), weight=change)) +
  coord_flip()
```

With cowplot, we can use a function called `plot_grid` to arrange the charts:

```
plot_grid(bar1, bar2)
```

We can also stack them on top of each other:

```
plot_grid(bar1, bar2, ncol=1)
```

To make these publishable, we should add headlines, chatter, decent labels, credit lines, etc. But to do this, we'll have to figure out which labels go on which charts, so we can make it look decent. For example – both charts don't need x or y labels. If you don't have a title and subtitle on both, the spacing is off, so you need to leave one blank or the other blank. You'll just have to fiddle with it until you get it looking right.

```
bar1 <- ggplot() +
  geom_bar(data=top10, aes(x=reorder(Institution, change), weight=change)) +
  coord_flip() +
  labs(title="College football winners...", subtitle = "Not every football program saw att
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_text(size=10),
    panel.grid.minor = element_blank()
    )
```

```
bar2 <- ggplot() +
  geom_bar(data=bottom10, aes(x=reorder(Institution, change), weight=change)) +
  coord_flip() +
  labs(title = "... and losers", subtitle= "", x="", y="",  caption="Source: NCAA | By Der
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_text(size=10),
    panel.grid.minor = element_blank()
    )
```

```
plot_grid(bar1, bar2)
```

What's missing here? Color. Our eyes aren't drawn to anything (except maybe the top and bottom). So we need to help that. A bar chart without context or color to draw attention to something isn't much of a bar chart. Same with a line chart – if your line chart has one line, no context, no color, it's going to fare poorly.

```
ho <- bottom10 %>% filter(Institution == "Houston")
ks <- top10 %>% filter(Institution == "Kansas")
```

```
bar1 <- ggplot() +
  geom_bar(data=top10, aes(x=reorder(Institution, change), weight=change)) +
  geom_bar(data=ks, aes(x=reorder(Institution, change), weight=change), fill="#011E41") +
  coord_flip() +
  labs(title="College football winners...", subtitle = "Not every football program saw att
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_text(size=10),
    panel.grid.minor = element_blank()
    )
```

```
bar2 <- ggplot() +
  geom_bar(data=bottom10, aes(x=reorder(Institution, change), weight=change)) +
  geom_bar(data=ho, aes(x=reorder(Institution, change), weight=change), fill="#006F71") +
  coord_flip() +
  labs(title = "... and losers", subtitle= "", x="", y="",  caption="Source: NCAA | By Der
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_text(size=10),
    panel.grid.minor = element_blank()
    )
```

```
plot_grid(bar1, bar2)
```

College football winners..... and losers
Not every football program saw attendance shrink in 2018. But some re
Kansas
Coastal Caro.
La.−Monroe
Virginia
Louisiana Tech
North Carolina
Fla. Atlantic
Texas St.
New Mexico St.
San Jose St.
0
25
50
75
100
Percent change
UTSA
Rutgers
Florida St.
Colorado St.
Boston College
FIU
UCF
Arizona
UCLA
Houston
−30
−20
−10
0
Source: NCAA | By Derek Willis

28 Encircling points on a scatterplot

One thing we've talked about all semester is drawing attention to the thing you want to draw attention to. We've used color and labels to do that so far. Let's add another layer to it – a shape around the points you want to highlight.

Remember: The point of all of this is to draw the eye to what you are trying to show your reader. You want people to see the story you are trying to tell.

It's not hard to draw a shape in ggplot – it is a challenge to put it in the right place. But, there is a library to the rescue that makes this super easy – `ggalt`.

Install it in the console with `install.packages("ggalt")`

There's a bunch of things that `ggalt` does, but one of the most useful for us is the function `encircle`. Let's dive in.

```
library(tidyverse)
library(ggalt)
```

Let's say we want to highlight the top scorers in women's college basketball. So let's use our player data.

And while we're loading it, let's filter out anyone who hasn't played.

```
players <- read_csv("data/wbb_players_2022.csv") %>% filter(mp > 0)
```

```
Rows: 4781 Columns: 174
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (23): position, team, ncaa_conference, player, fg_pct, fg2m_pct, fg3m_p...
dbl (151): end_year, ncaa_id, win, loss, gp, gs, ga, win_pct, mp, sec, sec_t...

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.
```

We've done this before, but let's make a standard scatterplot of minutes and points.

```
ggplot() + geom_point(data=players, aes(x=mp, y=pts))
```

So we can see right away that there are some dots at the very top that we'd want to highlight. Who are these scoring machines?

Like we have done in the past, let's make a dataframe of top scorers. We'll set the cutoff at 700 points in a season.

```
topscorers <- players %>% filter(pts > 700)
```

And like we've done in the past, we can add it to the chart with another geom_point. We'll make all the players grey, we'll make all the top scorers black.

```
ggplot() +
  geom_point(data=players, aes(x=mp, y=pts), color="grey") +
  geom_point(data=topscorers, aes(x=mp, y=pts), color="black")
```

And like that, we're on the path to something publishable. We'll need to label those dots with `ggrepel` and we'll need to drop the default grey and add some headlines and all that. And, for the most part, we've got a solid chart.

But what if we could really draw the eye to those players. Let's draw a circle around them. In `ggalt`, there is a new geom called `geom_encircle`, which … does what you think it does. It encircles all the dots in a dataset.

So let's add geom_encircle and we'll just copy the data and the aes from our topscorers geom_point. Then, we need to give the encirclement a shape using s_shape – which is a number between 0 and 1 – and then how far away from the dots to draw the circle using expand, which is another number between 0 and 1.

Let's start with `s_shape` 1 and `expand` 1.

```
ggplot() +
  geom_point(data=players, aes(x=mp, y=pts), color="grey") +
  geom_point(data=topscorers, aes(x=mp, y=pts), color="black") +
  geom_encircle(data=topscorers, aes(x=mp, y=pts), s_shape=1, expand=1, colour="red")
```

Whoa. That's … not good.

Let's go the opposite direction.

```
ggplot() +
  geom_point(data=players, aes(x=mp, y=pts), color="grey") +
  geom_point(data=topscorers, aes(x=mp, y=pts), color="black") +
  geom_encircle(data=topscorers, aes(x=mp, y=pts), s_shape=0, expand=0, colour="red")
```

Better, but … the circle cuts through multiple dots.

This takes a little bit of finessing, but a shape of .5 means the line will have some bend to it – it'll look more like someone circled it with a pen. Then, the expand is better if you use hundredths instead of tenths. So .01 instead of .1. Here's mine after fiddling with it for a bit, and I'll add in player names as a label.

```
ggplot() +
  geom_point(data=players, aes(x=mp, y=pts), color="grey") +
  geom_point(data=topscorers, aes(x=mp, y=pts), color="black") +
  geom_text(data=topscorers, aes(x=mp, y=pts, label=player), hjust = 0, vjust=1) +
  geom_encircle(data=topscorers, aes(x=mp, y=pts), s_shape=.5, expand=.03, colour="red")
```

Now let's clean this up and make it presentable. If you look at the top scorers, four were Wooden Award finalists. So here's what a chart telling that story might look like.

```
ggplot() +
  geom_point(data=players, aes(x=mp, y=pts), color="grey") +
  geom_point(data=topscorers, aes(x=mp, y=pts), color="black") +
  geom_encircle(data=topscorers, aes(x=mp, y=pts), s_shape=.5, expand=.03, colour="red") +
  geom_text(aes(x=725, y=700, label="Top scorers")) +
  labs(title="Four top scorers are Wooden Award candidates", subtitle="Big scorers Caitlin
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_text(size=10),
    panel.grid.minor = element_blank()
    )
```

28.1 A different, more local example

You can use circling outside of the top of something. It's a bit obvious that the previous dots were top scorers. What about when they aren't at the top?

Works the same way – use layering and color smartly and tell the story with all your tools.

Let's grab the top three point attempt takers on the Maryland roster. As of now, only one will be coming back.

```
mdtop <- players %>% filter(team == "Maryland") %>% top_n(3, fg3a)
```

And just like above, we can plug in our players geom, our mdtop dataframe into another geom, then encircle that dataframe. Slap some headlines and annotations on it and here's what we get:

```
ggplot() + geom_point(data=players, aes(x=mp, y=fg3a), color="grey") + geom_point(data=mdt
  geom_text(aes(x=400, y=100, label="Maryland's top three shooters")) +
  labs(title="The Big Three?", subtitle="Maryland's top three point shooters not among the
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
```

```
axis.title = element_text(size = 8),
plot.subtitle = element_text(size=10),
panel.grid.minor = element_blank()
)
```

From the top, Maryland's dots are Katie Benzan, Chloe Bibby and … Shyanne Sellers, the true freshman.

29 Text cleaning

On occasion, you'll get some data from someone that … isn't quite what you need it to be. There's something flawed in it. Some extra text, some choice that the data provider made that you just don't agree with.

There's a ton of tools in the tidyverse to fix this, and you already have some tools in your toolbox. Let's take a look at a couple.

First, you know what you need.

```
library(tidyverse)
```

Now, two examples.

29.1 Stripping out text

Throughout this class, we've used data from Sports Reference. If you've used their Share > CSV method to copy data from a table, you may have noticed some extra cruft in the player name field. If you haven't seen it, I'll give you an example – a dataset of NBA players and their advanced metrics.

Now load it.

```
nbaplayers <- read_csv("data/nbaplayers.csv")
```

```
New names:
Rows: 624 Columns: 29
-- Column specification
-------------------------------------------------------- Delimiter: "," chr
(3): Player, Pos, Tm dbl (24): Rk, Age, G, MP, PER, TS%, 3PAr, FTr, ORB%, DRB%,
TRB%, AST%, STL%,... lgl (2): ...20, ...25
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.
* `` -> `...20`
* `` -> `...25`
```

Let's take a look:

```
head(nbaplayers)
```

```
# A tibble: 6 x 29
     Rk Player     Pos     Age Tm        G    MP   PER `TS%` `3PAr`   FTr `ORB%`
  <dbl> <chr>      <chr> <dbl> <chr> <dbl> <dbl> <dbl> <dbl>  <dbl> <dbl>  <dbl>
1     1 "Steven A~ C        26 OKC      58  1564  20.8 0.605  0.007 0.413   14.4
2     2 "Bam Adeb~ PF       22 MIA      65  2235  20.6 0.606  0.018 0.476    8.7
3     3 "LaMarcus~ C        34 SAS      53  1754  19.8 0.571  0.198 0.241    6.3
4     4 "Nickeil ~ SG       21 NOP      41   501   7.6 0.441  0.515 0.123    1.7
5     5 "Grayson ~ SG       24 MEM      30   498  11.4 0.577  0.517 0.199    1.1
6     6 "Jarrett ~ C        21 BRK      64  1647  20.3 0.658  0.012 0.574   12.5
# ... with 17 more variables: `DRB%` <dbl>, `TRB%` <dbl>, `AST%` <dbl>,
#   `STL%` <dbl>, `BLK%` <dbl>, `TOV%` <dbl>, `USG%` <dbl>, ...20 <lgl>,
#   OWS <dbl>, DWS <dbl>, WS <dbl>, `WS/48` <dbl>, ...25 <lgl>, OBPM <dbl>,
#   DBPM <dbl>, BPM <dbl>, VORP <dbl>
```

You can see that every players name is their name, then two backslashes, then some version of their name that must have meaning to Sports Reference, but not to us. So we need to get rid of that.

To do this, we're going to use a little regular expression magic. Regular expressions are a programmatic way to find any pattern in text. What we're looking for is that `\\` business. But, that presents a problem, because the `\` is a special character. It's called an escape character. That escape character means what comes next is potentially special. For instance, if you see `\n`, that's a newline character. So normally, if you see that, it would add a return.

So for us to get rid of the `\` we're going to have to escape the escape character with an escape character. And we have two of them. So we have to do it twice.

Yes. Really.

So if we wanted to find two backslashes, we need `\\\\`. Then, using regular expressions, we can say "and then everything else after this" with this: `.*`

No really. That's it. So we're looking for `\\\\.*`. That'll find two backslashes and then everything after it. If you think this is hard … you're right. Regular expressions are an entire month of a programming course by themselves. They are EXTREMELY powerful.

To find something in text, we'll use a function called `gsub`. The pattern in `gsub` is `pattern, what we want to replace it with, what column this can all be found in`. So in our example, the pattern is `\\\\.*`, what we want to replace it with is … nothing, and this is all in the Player column. Here's the code.

```r
nbaplayers %>% mutate(Player=gsub("\\\\.*","",Player)) %>% head()
```

```
# A tibble: 6 x 29
     Rk Player     Pos     Age Tm        G    MP   PER `TS%` `3PAr`   FTr `ORB%`
  <dbl> <chr>      <chr> <dbl> <chr> <dbl> <dbl> <dbl> <dbl>  <dbl> <dbl>  <dbl>
1     1 Steven Ad~ C        26 OKC      58  1564  20.8 0.605  0.007 0.413   14.4
2     2 Bam Adeba~ PF       22 MIA      65  2235  20.6 0.606  0.018 0.476    8.7
3     3 LaMarcus ~ C        34 SAS      53  1754  19.8 0.571  0.198 0.241    6.3
4     4 Nickeil A~ SG       21 NOP      41   501   7.6 0.441  0.515 0.123    1.7
5     5 Grayson A~ SG       24 MEM      30   498  11.4 0.577  0.517 0.199    1.1
6     6 Jarrett A~ C        21 BRK      64  1647  20.3 0.658  0.012 0.574   12.5
# ... with 17 more variables: `DRB%` <dbl>, `TRB%` <dbl>, `AST%` <dbl>,
#   `STL%` <dbl>, `BLK%` <dbl>, `TOV%` <dbl>, `USG%` <dbl>, ...20 <lgl>,
#   OWS <dbl>, DWS <dbl>, WS <dbl>, `WS/48` <dbl>, ...25 <lgl>, OBPM <dbl>,
#   DBPM <dbl>, BPM <dbl>, VORP <dbl>
```

Just like that, the trash is gone.

29.2 Another example: splitting columns

Text cleaning is really just a set of logic puzzles. What do I need to do? How can I get there step by step?

The NCAA does some very interesting things with data, making it pretty useless.

Let's import it and take a look.

```r
kills <- read_csv("data/killsperset.csv")
```

```
Rows: 150 Columns: 9
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (5): Player, Cl, Ht, Pos, Season
dbl (4): Rank, S, Kills, Per Set

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.
```

```r
head(kills)
```

```
# A tibble: 6 x 9
   Rank Player                      Cl    Ht    Pos       S Kills Per S~1 Season
  <dbl> <chr>                       <chr> <chr> <chr> <dbl> <dbl>   <dbl> <chr>
1     1 Lindsey Ruddins, UC Santa ~ So.   6-2   OH       90   526    5.84 2017-~
2     2 Pilar Victoria, Arkansas (~ Sr.   5-11  OH      116   634    5.47 2017-~
3     3 Laura Milos, Oral Roberts ~ Sr.   5-10  OH      106   560    5.28 2017-~
4     4 Carlyle Nusbaum, Lipscomb ~ Jr.   5-10  OH      100   522    5.22 2017-~
5     5 Veronica Jones-Perry, BYU ~ Jr.   6-0   OH      118   569    4.82 2017-~
6     6 Torrey Van Winden, Cal Pol~ So.   6-3   OH      101   477    4.72 2017-~
# ... with abbreviated variable name 1: `Per Set`
```

First things first, Player isn't just player, it's player, school and conference, all in one. And Ht is a character field – and in feet and inches.

So … this is a mess. But there is a pattern. See it? A comma after the player's name. The Conference is in parens. We can use that.

For this, we're going to use a `tidyr` function called `separate` to split columns into multiple columns based on a character. We'll do this step by step.

First, let's use that comma to split the player and the rest. Ignore the head at the end. That's just to keep it from showing you all 150.

```
kills %>% separate(Player, into=c("Player", "School"), sep=",") %>% head()
```

```
# A tibble: 6 x 10
   Rank Player               School Cl    Ht    Pos       S Kills Per S~1 Season
  <dbl> <chr>                <chr>  <chr> <chr> <chr> <dbl> <dbl>   <dbl> <chr>
1     1 Lindsey Ruddins      " UC ~ So.   6-2   OH       90   526    5.84 2017-~
2     2 Pilar Victoria       " Ark~ Sr.   5-11  OH      116   634    5.47 2017-~
3     3 Laura Milos          " Ora~ Sr.   5-10  OH      106   560    5.28 2017-~
4     4 Carlyle Nusbaum      " Lip~ Jr.   5-10  OH      100   522    5.22 2017-~
5     5 Veronica Jones-Perry " BYU~ Jr.   6-0   OH      118   569    4.82 2017-~
6     6 Torrey Van Winden    " Cal~ So.   6-3   OH      101   477    4.72 2017-~
# ... with abbreviated variable name 1: `Per Set`
```

Good start.

Now, let's get the conference separated. A problem is going to crop up here – the paren is a special character, so we have to escape it with the \\.

```
kills %>%
  separate(Player, into=c("Player", "School"), sep=",") %>%
```

```
  separate(School, into=c("School", "Conference"), sep="\\(") %>%
  head()
```

```
Warning: Expected 2 pieces. Additional pieces discarded in 3 rows [15, 42, 83].

# A tibble: 6 x 11
   Rank Player       School Confe~1 Cl    Ht    Pos       S Kills Per S~2 Season
  <dbl> <chr>        <chr>  <chr>   <chr> <chr> <chr> <dbl> <dbl>   <dbl> <chr>
1     1 Lindsey Rud~ " UC ~ Big We~ So.   6-2   OH       90   526    5.84 2017-~
2     2 Pilar Victo~ " Ark~ SEC)    Sr.   5-11  OH      116   634    5.47 2017-~
3     3 Laura Milos  " Ora~ Summit~ Sr.   5-10  OH      106   560    5.28 2017-~
4     4 Carlyle Nus~ " Lip~ ASUN)   Jr.   5-10  OH      100   522    5.22 2017-~
5     5 Veronica Jo~ " BYU~ WCC)    Jr.   6-0   OH      118   569    4.82 2017-~
6     6 Torrey Van ~ " Cal~ Big We~ So.   6-3   OH      101   477    4.72 2017-~
# ... with abbreviated variable names 1: Conference, 2: `Per Set`
```

Uh oh. Says we have problems in rows 15, 42 and 83. What are they? The NCAA has decided to put (FL), (NY) and (PA) into three teams to tell you they're in Florida, New York and Pennsylvania respectively. Well, we can fix that with some gsub and we'll use a switch called `fixed`, which when set to TRUE it means this literal string, no special characters.

```
 kills %>%
   separate(Player, into=c("Player", "School"), sep=",") %>%
   mutate(School = gsub("(FL)", "FL", School, fixed=TRUE)) %>%
   mutate(School = gsub("(NY)", "NY", School, fixed=TRUE)) %>%
   mutate(School = gsub("(PA)", "PA", School, fixed=TRUE)) %>%
   separate(School, into=c("School", "Conference"), sep="\\(") %>%
   head()
```

```
# A tibble: 6 x 11
   Rank Player       School Confe~1 Cl    Ht    Pos       S Kills Per S~2 Season
  <dbl> <chr>        <chr>  <chr>   <chr> <chr> <chr> <dbl> <dbl>   <dbl> <chr>
1     1 Lindsey Rud~ " UC ~ Big We~ So.   6-2   OH       90   526    5.84 2017-~
2     2 Pilar Victo~ " Ark~ SEC)    Sr.   5-11  OH      116   634    5.47 2017-~
3     3 Laura Milos  " Ora~ Summit~ Sr.   5-10  OH      106   560    5.28 2017-~
4     4 Carlyle Nus~ " Lip~ ASUN)   Jr.   5-10  OH      100   522    5.22 2017-~
5     5 Veronica Jo~ " BYU~ WCC)    Jr.   6-0   OH      118   569    4.82 2017-~
6     6 Torrey Van ~ " Cal~ Big We~ So.   6-3   OH      101   477    4.72 2017-~
# ... with abbreviated variable names 1: Conference, 2: `Per Set`
```

One last thing: see the trailing paren?

```r
kills %>%
  separate(Player, into=c("Player", "School"), sep=",") %>%
  mutate(School = gsub("(FL)", "FL", School, fixed=TRUE)) %>%
  mutate(School = gsub("(NY)", "NY", School, fixed=TRUE)) %>%
  mutate(School = gsub("(PA)", "PA", School, fixed=TRUE)) %>%
  separate(School, into=c("School", "Conference"), sep="\\(") %>%
  mutate(Conference=gsub(")", "", Conference)) %>%
  head()
```

```
# A tibble: 6 x 11
   Rank Player       School Confe~1 Cl    Ht    Pos       S Kills Per S~2 Season
  <dbl> <chr>        <chr>  <chr>   <chr> <chr> <chr> <dbl> <dbl>   <dbl> <chr>
1     1 Lindsey Rud~ " UC ~ Big We~ So.   6-2   OH       90   526    5.84 2017-~
2     2 Pilar Victo~ " Ark~ SEC     Sr.   5-11  OH      116   634    5.47 2017-~
3     3 Laura Milos  " Ora~ Summit~ Sr.   5-10  OH      106   560    5.28 2017-~
4     4 Carlyle Nus~ " Lip~ ASUN    Jr.   5-10  OH      100   522    5.22 2017-~
5     5 Veronica Jo~ " BYU~ WCC     Jr.   6-0   OH      118   569    4.82 2017-~
6     6 Torrey Van ~ " Cal~ Big We~ So.   6-3   OH      101   477    4.72 2017-~
# ... with abbreviated variable names 1: Conference, 2: `Per Set`
```

Looking good, no errors.

Now, what should we do about Ht? 6-2 is not going to tell me much when I want to run a regression of height to kills per set. And it's a character field. So we need to convert it to numbers.

Separate again comes to the rescue.

```r
kills %>%
  separate(Player, into=c("Player", "School"), sep=",") %>%
  mutate(School = gsub("(FL)", "FL", School, fixed=TRUE)) %>%
  mutate(School = gsub("(NY)", "NY", School, fixed=TRUE)) %>%
  mutate(School = gsub("(PA)", "PA", School, fixed=TRUE)) %>%
  separate(School, into=c("School", "Conference"), sep="\\(") %>%
  mutate(Conference=gsub(")", "", Conference)) %>%
  separate(Ht, into=c("Feet", "Inches"), sep="-") %>%
  mutate(Feet = as.numeric(Feet), Inches = as.numeric(Inches)) %>%
  head()
```

```
# A tibble: 6 x 12
   Rank Player       School Confe~1 Cl     Feet Inches Pos       S Kills Per S~2
  <dbl> <chr>        <chr>  <chr>   <chr> <dbl>  <dbl> <chr> <dbl> <dbl>   <dbl>
```

```
1     1 Lindsey Rud~ " UC ~ Big We~ So.       6      2 OH       90   526    5.84
2     2 Pilar Victo~ " Ark~ SEC     Sr.       5     11 OH      116   634    5.47
3     3 Laura Milos  " Ora~ Summit~ Sr.       5     10 OH      106   560    5.28
4     4 Carlyle Nus~ " Lip~ ASUN    Jr.       5     10 OH      100   522    5.22
5     5 Veronica Jo~ " BYU~ WCC     Jr.       6      0 OH      118   569    4.82
6     6 Torrey Van ~ " Cal~ Big We~ So.       6      3 OH      101   477    4.72
# ... with 1 more variable: Season <chr>, and abbreviated variable names
#   1: Conference, 2: `Per Set`
```

But how do we turn that into a height? Math!

```r
kills %>%
  separate(Player, into=c("Player", "School"), sep=",") %>%
  mutate(School = gsub("(FL)", "FL", School, fixed=TRUE)) %>%
  mutate(School = gsub("(NY)", "NY", School, fixed=TRUE)) %>%
  mutate(School = gsub("(PA)", "PA", School, fixed=TRUE)) %>%
  separate(School, into=c("School", "Conference"), sep="\\(") %>%
  mutate(Conference=gsub(")", "", Conference)) %>%
  separate(Ht, into=c("Feet", "Inches"), sep="-") %>%
  mutate(Feet = as.numeric(Feet), Inches = as.numeric(Inches)) %>%
  mutate(Height = (Feet*12)+Inches) %>%
  head()
```

```
# A tibble: 6 x 13
   Rank Player       School Confe~1 Cl     Feet Inches Pos       S Kills Per S~2
  <dbl> <chr>        <chr>  <chr>   <chr> <dbl>  <dbl> <chr> <dbl> <dbl>   <dbl>
1     1 Lindsey Rud~ " UC ~ Big We~ So.       6      2 OH       90   526    5.84
2     2 Pilar Victo~ " Ark~ SEC     Sr.       5     11 OH      116   634    5.47
3     3 Laura Milos  " Ora~ Summit~ Sr.       5     10 OH      106   560    5.28
4     4 Carlyle Nus~ " Lip~ ASUN    Jr.       5     10 OH      100   522    5.22
5     5 Veronica Jo~ " BYU~ WCC     Jr.       6      0 OH      118   569    4.82
6     6 Torrey Van ~ " Cal~ Big We~ So.       6      3 OH      101   477    4.72
# ... with 2 more variables: Season <chr>, Height <dbl>, and abbreviated
#   variable names 1: Conference, 2: `Per Set`
```

And now, in 10 lines of code, using separate, mutate and gsub, we've turned the mess that is the NCAA's data into actually useful data we can analyze.

These patterns of thought come in handy when facing messed up data.

30 Headlines

These are the pieces of a good graphic:

- Headline
- Chatter
- The main body
- Annotations
- Labels
- Source line
- Credit line

The first on that list is the first for a reason. The headline is an incredibly important part of any graphic: it's often the first thing a reader will see. It's got to entice people in, tell them a little bit about what they're going to see, and help tell the story.

The second item is the chatter – the text underneath that headline. It needs to work with the headline to further the story, drive people toward the point, maybe add some context.

The two bits of text are extremely important. Let's set up a chart and talk about how to do it wrong and how to do it better.

```
library(tidyverse)
library(ggrepel)
```

The data and the chart code isn't important for you to follow along. The code is nothing special. The issues will be with the words that you'll see below.

```
scoring <- read_csv("data/scoringoffense.csv")
```

```
Rows: 1253 Columns: 10
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): Name
dbl (9): G, TD, FG, 1XP, 2XP, Safety, Points, Points/G, Year

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.
```

```r
total <- read_csv("data/totaloffense.csv")
```

```
Rows: 1253 Columns: 9
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): Name
dbl (8): G, Rush Yards, Pass Yards, Plays, Total Yards, Yards/Play, Yards/G,...

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.
```

```r
offense <- total %>% left_join(scoring, by=c("Name", "Year"))

umd <- offense %>% filter(Name == "Maryland") %>% filter(Year == 2018)
```

```r
ggplot(offense, aes(x=`Yards/G`, y=`Points/G`)) +
  geom_point(color="grey") + geom_smooth(method=lm, se=FALSE) +
  labs(x="Total yards per game", y="Points per game", title="Headline here", subtitle="Thi
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_text(size=10),
    panel.grid.minor = element_blank()
    ) +
  geom_point(data=umd, aes(x=`Yards/G`, y=`Points/G`), color="red") +
  geom_text_repel(data=umd, aes(x=`Yards/G`, y=`Points/G`, label="Maryland 2018"))
```

```
`geom_smooth()` using formula 'y ~ x'
```

First, let's start with some headline basics:

- Your headline should be about what the chart is about, not what makes up the chart. What story is the chart telling? What made it interesting to you? Don't tell me what the stats are, tell me what it says.
- Your headline should be specific. Generic headlines are boring and ignored.
- Your headline should, most often, have a verb. It's not a 100 percent requirement, but a headline without a verb means you're trying to be cute and …
- Your headline shouldn't be overly cute. Trying to get away with slang, a very Of The Moment cultural reference that will be forgotten soon, or some inside joke is asking for trouble.
- Your headline should provoke a reaction.

Given our graph, here's a few that don't work.

```
ggplot(offense, aes(x=`Yards/G`, y=`Points/G`)) +
  geom_point(color="grey") + geom_smooth(method=lm, se=FALSE) +
  labs(x="Total yards per game", y="Points per game", title="Maryland's offense", subtitle
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
```

```
    plot.subtitle = element_text(size=10),
    panel.grid.minor = element_blank()
    ) +
  geom_point(data=umd, aes(x=`Yards/G`, y=`Points/G`), color="red") +
  geom_text_repel(data=umd, aes(x=`Yards/G`, y=`Points/G`, label="Maryland 2018"))
```

```
`geom_smooth()` using formula 'y ~ x'
```

The problems here:

- No verb.
- Generic, forgettable, doesn't say anything.
- What is this chart about? What does it say? We have no idea from the headline and chatter.
- Don't repeat words from the headline in the chatter. Nebraska Nebraska looks bad. Make one of the Huskers if you're going to do this.

Another example:

```
ggplot(offense, aes(x=`Yards/G`, y=`Points/G`)) +
  geom_point(color="grey") + geom_smooth(method=lm, se=FALSE) +
```

```
  labs(x="Total yards per game", y="Points per game", title="Points per game vs total yard
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_text(size=10),
    panel.grid.minor = element_blank()
    ) +
  geom_point(data=umd, aes(x=`Yards/G`, y=`Points/G`), color="red") +
  geom_text_repel(data=umd, aes(x=`Yards/G`, y=`Points/G`, label="Maryland 2018"))
```

```
`geom_smooth()` using formula 'y ~ x'
```

What's wrong here?

- The headline is about the stats, not the story.
- The headline lacks a verb.
- The headline lacks any interest, really.
- The headline at least moves in the direction of what this chart is about, but see the previous two.

- The chatter adds more flavor to it, but what does "below the blue line" even mean? We're leaving the reader with a lot of questions and no real answers. That;s bad.

Let's try to do this better.

```
ggplot(offense, aes(x=`Yards/G`, y=`Points/G`)) +
  geom_point(color="grey") + geom_smooth(method=lm, se=FALSE) +
  labs(x="Total yards per game", y="Points per game", title="Maryland's strength", subtitl
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_text(size=10),
    panel.grid.minor = element_blank()
    ) +
  geom_point(data=umd, aes(x=`Yards/G`, y=`Points/G`), color="red") +
  geom_text_repel(data=umd, aes(x=`Yards/G`, y=`Points/G`, label="Maryland 2018"))
```

```
`geom_smooth()` using formula 'y ~ x'
```

What works here:

- Provokes a reaction by asking a question. Drives at what the story is about.
- The chatter answers the question in the headline without talking about the blue line, a model, anything. A reader can see it.
- Simple, precise, direct language.

One more, same chart.

```
ggplot(offense, aes(x=`Yards/G`, y=`Points/G`)) +
  geom_point(color="grey") + geom_smooth(method=lm, se=FALSE) +
  labs(x="Total yards per game", y="Points per game", title="Maryland's offense overperfor
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_text(size=10),
    panel.grid.minor = element_blank()
    ) +
  geom_point(data=umd, aes(x=`Yards/G`, y=`Points/G`), color="red") +
  geom_text_repel(data=umd, aes(x=`Yards/G`, y=`Points/G`, label="Maryland 2018"))
```

```
`geom_smooth()` using formula 'y ~ x'
```

What works here:

- Strong verb: overperformed.
- Headline tells the story. Chatter bolsters it.
- Doesn't repeat Maryland or Terps

Taking time to sharpen your headlines will make your graphics better.

31 Annotations

Some of the best sports data visualizations start with a provocative question. How about this one: Who really belongs in the college football playoffs and why is it never Notre Dame?

For this, we're going to go back to some code we started in Bubble Charts and we're going to add some annotations to it. Annotations help us draw attention to things, or help the reader understand what they're looking at. They're labels on things, be that the teams we want to highlight or regions of the chart or lines or all of those things.

For this, we'll need to add a new library to the mix called `ggrepel`. You'll need to install it in the console with `install.packages("ggrepel")`.

```
library(tidyverse)
library(ggrepel)
```

Now we'll grab the data, each football game in 2021.

Now load it.

```
logs <- read_csv("data/footballlogs21.csv")
```

```
Rows: 1655 Columns: 54
-- Column specification --------------------------------------------------------
Delimiter: ","
chr   (8): HomeAway, Opponent, Result, TeamFull, TeamURL, Outcome, Team, Con...
dbl  (45): Game, PassingCmp, PassingAtt, PassingPct, PassingYds, PassingTD, ...
date  (1): Date

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.
```

I'm going to set up a point chart that places teams on two-axes – yards per play on offense on the x axis, and yards per play on defense. We did this in the bubble charts example.

```
logs %>%
  group_by(Team, Conference) %>%
```

```
  summarise(
    TotalPlays = sum(OffensivePlays),
    TotalYards = sum(OffensiveYards),
    DefensivePlays = sum(DefPlays),
    DefensiveYards = sum(DefYards)) %>%
  mutate(
    OffensiveYPP = TotalYards/TotalPlays,
    DefensiveYPP = DefensiveYards/DefensivePlays) -> ypp
```

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

To build the annotations, I want the average for offensive yards per play and defensive yards per play. We're going to use those as a proxy for quality. If your team averages more yards per play on offense, that's good. If they average fewer yards per play on defense, that too is good. So that sets up a situation where we have four corners, anchored by good at both and bad at both. The averages will create lines to divide those four corners up.

```
averages <- ypp %>% ungroup() %>% summarise(AvgOffYardsPer = mean(OffensiveYPP), AvgDefYar

averages
```

```
# A tibble: 1 x 2
  AvgOffYardsPer AvgDefYardsPer
           <dbl>          <dbl>
1           5.79           5.66
```

I also want to highlight playoff teams.

```
playoff_teams <- c("Cincinnati", "Alabama", "Michigan", "Ohio State")

playoffs <- ypp %>% filter(Team %in% playoff_teams)
```

Now we create the plot. We have two geom_points, starting with everyone, then playoff teams. I alter the colors on each to separate them. Next, I add a geom_hline to add the horizontal line of my defensive average and a geom_vline for my offensive average. Next, I want to add some text annotations, labeling two corners of my chart (the other two, in my opinion, become obvious). Then, I want to label all the playoff teams. I use `geom_text_repel` to do that – it's using the ggrepel library to push the text away from the dots, respective of other labels and

other dots. It means you don't have to move them around so you can read them, or so they don't cover up the dots.

The rest is just adding labels and messing with the theme.

```
ggplot() +
  geom_point(data=ypp, aes(x=OffensiveYPP, y=DefensiveYPP), color="light grey") +
  geom_point(data=playoffs, aes(x=OffensiveYPP, y=DefensiveYPP)) +
  geom_hline(yintercept=5.7, color="dark grey") +
  geom_vline(xintercept=5.7, color="dark grey") +
  geom_text(aes(x=7, y=5.5, label="Good Offense, Good Defense"), color="blue") +
  geom_text(aes(x=5, y=6, label="Bad Defense, Bad Offense"), color="blue") +
  geom_text_repel(data=playoffs, aes(x=OffensiveYPP, y=DefensiveYPP, label=Team)) +
  labs(x="Offensive Yards Per Play", y="Defensive Points Per Play", title="All four playof
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 10),
    axis.text = element_text(size = 7),
    axis.ticks = element_blank(),
    panel.grid.minor = element_blank(),
    panel.grid.major.x = element_blank()
  )
```

32 Finishing touches

The output from ggplot is good, but not great. We need to add some pieces to it. The elements of a good graphic are:

- Headline
- Chatter
- The main body
- Annotations
- Labels
- Source line
- Credit line

That looks like:

32.1 Graphics vs visual stories

While the elements above are nearly required in every chart, they aren't when you are making visual stories.

- When you have a visual story, things like credit lines can become a byline.
- In visual stories, source lines are often a note at the end of the story.
- Graphics don't always get headlines – sometimes just labels, letting the visual story headline carry the load.

An example from The Upshot. Note how the charts don't have headlines, source or credit lines.

32.2 Getting ggplot closer to output

Let's explore fixing up ggplot's output before we send it to a finishing program like Adobe Illustrator. We'll need a graphic to work with first.

```
library(tidyverse)
library(ggrepel)
```

Here's the data we'll use:

Let's load them and join them together.

```
scoring <- read_csv("data/scoringoffense.csv")
```

```
Rows: 1253 Columns: 10
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): Name
dbl (9): G, TD, FG, 1XP, 2XP, Safety, Points, Points/G, Year

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.
```

```
total <- read_csv("data/totaloffense.csv")
```

```
Rows: 1253 Columns: 9
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): Name
dbl (8): G, Rush Yards, Pass Yards, Plays, Total Yards, Yards/Play, Yards/G,...

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.
```

```
offense <- total %>% left_join(scoring, by=c("Name", "Year"))
```

We're going to need this later, so let's grab Maryland's 2018 stats from this dataframe.

```
umd <- offense %>%
  filter(Name == "Maryland") %>%
  filter(Year == 2018)
```

We'll start with the basics.

```
ggplot() +
  geom_point(data=offense, aes(x=`Yards/G`, y=`Points/G`), color="grey")
```

Let's take changing things one by one. The first thing we can do is change the figure size. Sometimes you don't want a square. We can use the `knitr` output settings in our chunk to do this easily in our notebooks.

```
ggplot() +
  geom_point(data=offense, aes(x=`Yards/G`, y=`Points/G`), color="grey")
```

Now let's add a fit line.

```
ggplot() +
  geom_point(data=offense, aes(x=`Yards/G`, y=`Points/G`), color="grey") +
  geom_smooth(data=offense, aes(x=`Yards/G`, y=`Points/G`), method=lm, se=FALSE)
```

And now some labels.

```
ggplot() +
  geom_point(data=offense, aes(x=`Yards/G`, y=`Points/G`), color="grey") +
  geom_smooth(data=offense, aes(x=`Yards/G`, y=`Points/G`), method=lm, se=FALSE) +
  labs(
    x="Total yards per game",
    y="Points per game",
    title="Maryland's overperforming offense",
    subtitle="The Terps' offense was the strength of the team. They overperformed.",
    caption="Source: NCAA | By Derek Willis"
    )
```

Let's get rid of the default plot look and drop the grey background.

```
ggplot() +
  geom_point(data=offense, aes(x=`Yards/G`, y=`Points/G`), color="grey") +
  geom_smooth(data=offense, aes(x=`Yards/G`, y=`Points/G`), method=lm, se=FALSE) +
  labs(
    x="Total yards per game",
    y="Points per game",
    title="Maryland's overperforming offense",
    subtitle="The Terps' offense was the strength of the team. They overperformed.",
    caption="Source: NCAA | By Derek Willis"
    ) +
  theme_minimal()
```

Off to a good start, but our text has no real heirarchy. We'd want our headline to stand out more. So let's change that. When it comes to changing text, the place to do that is in the theme element. There are a lot of ways to modify the theme. We'll start easy. Let's make the headline bigger and bold.

```
ggplot() +
  geom_point(data=offense, aes(x=`Yards/G`, y=`Points/G`), color="grey") +
  geom_smooth(data=offense, aes(x=`Yards/G`, y=`Points/G`), method=lm, se=FALSE) +
  labs(
    x="Total yards per game",
    y="Points per game",
    title="Maryland's overperforming offense",
    subtitle="The Terps' offense was the strength of the team. They overperformed.",
    caption="Source: NCAA | By Derek Willis") +
  theme_minimal() +
  theme(
    plot.title = element_text(size = 20, face = "bold")
    )
```

Now let's fix a few other things – like the axis labels being too big, the subtitle could be bigger and lets drop some grid lines.

```
ggplot() +
  geom_point(data=offense, aes(x=`Yards/G`, y=`Points/G`), color="grey") +
  geom_smooth(data=offense, aes(x=`Yards/G`, y=`Points/G`), method=lm, se=FALSE) +
  labs(
    x="Total yards per game",
    y="Points per game",
    title="Maryland's overperforming offense",
    subtitle="The Terps' offense was the strength of the team. They overperformed.",
    caption="Source: NCAA | By Derek Willis") +
  theme_minimal() +
  theme(
    plot.title = element_text(size = 20, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_text(size=10),
    panel.grid.minor = element_blank()
    )
```

Missing from this graph is the context that the headline promises. Where is Maryland? We haven't added it yet. So let's add a point and a label for it.

```
ggplot() +
  geom_point(data=offense, aes(x=`Yards/G`, y=`Points/G`), color="grey") +
  geom_smooth(data=offense, aes(x=`Yards/G`, y=`Points/G`), method=lm, se=FALSE) +
  labs(
    x="Total yards per game",
    y="Points per game",
    title="Maryland's overperforming offense",
    subtitle="The Terps' offense was the strength of the team. They overperformed.",
    caption="Source: NCAA | By Derek Willis") +
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_text(size=10),
    panel.grid.minor = element_blank()
    ) +
  geom_point(data=umd, aes(x=`Yards/G`, y=`Points/G`), color="red") +
  geom_text_repel(data=umd, aes(x=`Yards/G`, y=`Points/G`, label="Maryland 2018"))
```

If we're happy with this output – if it meets all of our needs for publication – then we can simply export it as a png file. We do that by adding `+ ggsave("plot.png", width=5, height=2)` to the end of our code. Note the width and the height are from our knitr parameters at the top – you have to repeat them or the graph will export at the default 7x7.

If there's more work you want to do with this graph that isn't easy or possible in R but is in Illustrator, simply change the file extension to `pdf` instead of `png`. The pdf will open as a vector file in Illustrator with everything being fully editable.

32.3 Waffle charts require special attention

Frequently in my classes, students use the waffle charts library quite extensively to make graphics. This is a quick walkthough on how to get a waffle chart into a publication ready state.

```
library(waffle)
```

Let's look at the offensive numbers from the 2021 Maryland v. Illinois football game. Maryland won 20-17, even though the Terps outgained the Illini by 113 yards. You can find the official stats on the NCAA's website.

I'm going to make two vectors for each team and record rushing yards and passing yards.

```
md <- c("Rushing"=131, "Passing"=350, 113)
il <- c("Rushing"=183, "Passing"=185, 0)
```

So what does the breakdown of Maryland's night look like? How balanced was the offense?

The waffle library can break this down in a way that's easier on the eyes than a pie chart. We call the library, add the data, specify the number of rows, give it a title and an x value label, and to clean up a quirk of the library, we've got to specify colors.

ADDITIONALLY

We can add labels and themes, but you have to be careful. The waffle library is applying it's own theme, but if we override something they are using in their theme, some things that are hidden come back and make it worse. So here is an example of how to use ggplot's `labs` and the theme to make a fully publication ready graphic.

```
waffle(md/10, rows = 5, xlab="1 square = 10 yards", colors = c("black", "red", "white")) +
  labs(
    title="Maryland vs Illinois on offense",
    subtitle="The Terps couldn't get much of a running game going.",
    caption="Source: NCAA | Graphic by Derek Willis") +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 10),
    axis.title.y = element_blank()
  )
```

But what if we're using a waffle iron? And what if we want to change the output size? It gets tougher.

Truth is, I'm not sure what is going on with the sizing. You can try it and you'll find that the outputs are … unpredictable.

Things you need to know about waffle irons:

- They're a convenience method, but all they're really doing is executing two waffle charts together. If you don't apply the theme to both waffle charts, it breaks.
- You will have to get creative about applying headline and subtitle to the top waffle chart and the caption to the bottom.
- Using ggsave doesn't work either. So you'll have to use R's pdf output.

Here is a full example. I start with my waffle iron code, but note that each waffle is pretty much a self contained thing. That's because a waffle iron isn't really a thing. It's just a way to group waffles together, so you have to make each waffle individually. My first waffle has the title and subtitle but no x axis labels and the bottom one has not title or subtitle but the axis labels and the caption.

```
iron(
 waffle(
   md/10,
```

```
    rows = 2,
    xlab="Maryland",
    colors = c("black", "red", "white")) +
    labs(
      title="Maryland vs Illinois: By the numbers",
      subtitle="The Terps couldn't run, the Illini could.") +
    theme(
     plot.title = element_text(size = 16, face = "bold"),
     axis.title = element_text(size = 10),
     axis.title.y = element_blank()
   ),
  waffle(
    il/10,
    rows = 2,
    xlab="Illinois\n1 square = 10 yards",
    colors = c("black", "red", "white")) +
    labs(caption="Source: NCAA | Graphic by Derek Willis")
)
```

32.4 Advanced text wrangling

Sometimes, you need a little more help with text than what is easily available. Sometimes you want a little more in your finishing touches. Let's work on some issues common in projects that can be fixed with new new libraries: multi-line chatter, axis labels that need more than just a word, axis labels that don't fit, and additional text boxes.

First things first, we'll need to install `ggtext` with install.packages. Then we'll load it.

```
library(ggtext)
```

Let's go back to our scatterplot above. As created, it's very simple, and the chatter doesn't say much. Let's write chatter that instead of being super spare is more verbose.

```
ggplot() +
  geom_point(data=offense, aes(x=`Yards/G`, y=`Points/G`), color="grey") +
  geom_smooth(data=offense, aes(x=`Yards/G`, y=`Points/G`), method=lm, se=FALSE) +
  labs(
    x="Total yards per game",
    y="Points per game",
    title="Maryland's overperforming offense",
    subtitle="The Terps' offense was the strength of the team, having to overcome a defens
    caption="Source: NCAA | By Derek Willis") +
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_text(size=10),
    panel.grid.minor = element_blank()
    ) +
  geom_point(data=umd, aes(x=`Yards/G`, y=`Points/G`), color="red") +
  geom_text_repel(data=umd, aes(x=`Yards/G`, y=`Points/G`, label="Maryland 2018"))
```

You can see the problem right away – it's too long and gets cut off. One way to fix this is to put `\n` where you think the line break should be. That's a newline character, so it would add a return there. But with ggtext, you can use simple HTML to style the text, which opens up a lot of options. We can use a to break the line. The other thing we need to do is in the theme element, change the `element_text` for plot.subtitle to `element_markdown`.

```
ggplot() +
  geom_point(data=offense, aes(x=`Yards/G`, y=`Points/G`), color="grey") +
  geom_smooth(data=offense, aes(x=`Yards/G`, y=`Points/G`), method=lm, se=FALSE) +
  labs(
    x="Total yards per game",
    y="Points per game",
    title="Maryland's overperforming offense",
    subtitle="The Terps' offense was the strength of the team, having to overcome a defens
    caption="Source: NCAA | By Derek Willis") +
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_markdown(size=10),
    panel.grid.minor = element_blank()
    ) +
```

```
  geom_point(data=umd, aes(x=`Yards/G`, y=`Points/G`), color="red") +
  geom_text_repel(data=umd, aes(x=`Yards/G`, y=`Points/G`, label="Maryland 2018"))
```

With ggtext, there's a lot more you can do with CSS, like change the color of text, that I don't recommend. Also, there's only a few HTML tags that have been implemented. For example, you can't add links because the `a` tag hasn't been added.

Another sometimes useful thing you can do is add much more explanation to your axis labels. This is going to be a silly example because "Points per game" is pretty self-explanatory, but roll with it. First, we create an unusually long y axis label, then, in theme, we add some code to `axis.title.y`.

```
ggplot() +
  geom_point(data=offense, aes(x=`Yards/G`, y=`Points/G`), color="grey") +
  geom_smooth(data=offense, aes(x=`Yards/G`, y=`Points/G`), method=lm, se=FALSE) +
  labs(
    x="Total yards per game",
    y="Points per game is an imperfect metric of offensive efficiency because defenses and
    title="Maryland's overperforming offense",
    subtitle="The Terps' offense was the strength of the team, having to overcome a defens
    caption="Source: NCAA | By Derek Willis") +
```

```
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_markdown(size=10),
    panel.grid.minor = element_blank(),
    axis.title.y = element_textbox_simple(
      orientation = "left-rotated",
      width = grid::unit(2.5, "in")
    )
    ) +
  geom_point(data=umd, aes(x=`Yards/G`, y=`Points/G`), color="red") +
  geom_text_repel(data=umd, aes(x=`Yards/G`, y=`Points/G`, label="Maryland 2018"))
```

One last advanced trick: Adding a text box explainer in the graphic. This should be used in somewhat rare circumstances – you don't want to pollute your data space with lots of text. If your graphic needs so much explainer text, you should be asking yourself hard questions about if your chart is clearly telling a story.

To add a text box explainer, you need to add a `geom_textbox` to your chart. The code below does that, and also adds a `geom_point` to anchor the box to a spot.

```
ggplot() +
  geom_point(data=offense, aes(x=`Yards/G`, y=`Points/G`), color="grey") +
  geom_smooth(data=offense, aes(x=`Yards/G`, y=`Points/G`), method=lm, se=FALSE) +
  geom_textbox(
    aes(x=225,
        y=50,
        label="Dots above the blue line indicate offenses that scored more points than the
        orientation = "upright",
        hjust=0,
        vjust=1), width = unit(2.8, "in")) +
  geom_point(aes(x=225, y=50), size=2) +
  labs(
    x="Total yards per game",
    y="Points per game",
    title="Maryland's overperforming offense",
    subtitle="The Terps' offense was the strength of the team, having to overcome a defens
    caption="Source: NCAA | By Derek Willis") +
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_markdown(size=10),
    panel.grid.minor = element_blank()
    ) +
  geom_point(data=umd, aes(x=`Yards/G`, y=`Points/G`), color="red") +
  geom_text_repel(data=umd, aes(x=`Yards/G`, y=`Points/G`, label="Maryland 2018"))
```

Maryland's overperforming offense
The Terps' offense was the strength of the team, having to overcome a defense tha
opponents. If you compare the offense to every other offense and how many points
yards they roll up, Maryland actually overperformed.
Dots above the blue line indicate offenses that scored more points than their yards per game would suggest they should.
Maryland 2018
Points per game
50
40
30
20
10
200
300
400
500
600
Total yards per game
Source: NCAA | By Derek Willis

33 Intro to rvest

All the way back in Chapter 2, we used Google Sheets and importHTML to get our own data out of a website. For me, that's a lot of pointing and clicking and copying and pasting. R has a library that can automate the harvesting of data from HTML on the internet. It's called `rvest`.

Let's grab a simple, basic HTML table from College Football Stats. There's nothing particularly strange about this table – it's simply formatted and easy to scrape.

First we'll need some libraries. We're going to use a library called `rvest`, which you can get by running `install.packages('rvest')` in the console.

```
library(rvest)
library(tidyverse)
```

The rvest package has functions that make fetching, reading and parsing HTML simple. The first thing we need to do is specify a url that we're going to scrape.

```
scoringoffenseurl <- "http://www.cfbstats.com/2021/leader/national/team/offense/split01/ca
```

Now, the most difficult part of scraping data from any website is knowing what exact HTML tag you need to grab. In this case, we want a `<table>` tag that has all of our data table in it. But how do you tell R which one that is? Well, it's easy, once you know what to do. But it's not simple. So I've made a short video to show you how to find it.

When you have simple tables, the code is very simple. You create a variable to receive the data, then pass it the url, read the html that was fetched, find the node you need using your XPath value you just copied and you tell rvest that it's a table.

```
scoringoffense <- scoringoffenseurl %>%
  read_html() %>%
  html_nodes(xpath = '//*[@id="content"]/div[2]/table') %>%
  html_table()
```

What we get from this is … not a dataframe. It's a list with one element in it, which just so happens to be our dataframe. When you get this, the solution is simple: just overwrite the variable you created with the first list element.

```r
scoringoffense <- scoringoffense[[1]]
```

And what do we have?

```r
head(scoringoffense)
```

```
# A tibble: 6 x 10
     `` Name                 G    TD    FG `1XP` `2XP` Safety Points `Points/G`
  <int> <chr>            <int> <int> <int> <int> <int>  <int>  <int>      <dbl>
1     1 Ohio State          13    76    20    76     0      1    594       45.7
2     2 Western Kentucky    14    79    23    72     2      0    619       44.2
3     3 Pittsburgh          14    76    17    71     0      1    580       41.4
4     4 Wake Forest         14    72    23    65     4      0    574       41
5     5 Coastal Carolina    13    72     9    65     2      2    532       40.9
6     6 Alabama             15    75    23    71     4      0    598       39.9
```

We have data, ready for analysis.

33.1 A slightly more complicated example

What if we want more than one year in our dataframe?

This is a common problem. What if we want to look at every scoring offense going back several years? The website has them going back to 2009. How can we combine them?

First, we should note, that the data does not have anything in it to indicate what year it comes from. So we're going to have to add that. And we're going to have to figure out a way to stack two dataframes on top of each other.

So let's grab 2020.

```r
scoringoffenseurl20 <- "http://www.cfbstats.com/2020/leader/national/team/offense/split01/

scoringoffense20 <- scoringoffenseurl20 %>%
  read_html() %>%
  html_nodes(xpath = '//*[@id="content"]/div[2]/table') %>%
  html_table()

scoringoffense20 <- scoringoffense20[[1]]
```

First, how are we going to know, in the data, which year our data is from? We can use mutate.

```
scoringoffense20 <- scoringoffense %>% mutate(YEAR = 2020)
```

```
Error in initialize(...): attempt to use zero-length variable name
```

Uh oh. Error. What does it say? It's … not clear, but a hint is that our first column doesn't have a name. Each column must be named. If you look at our data in the environment tab in the upper right corner, you'll see that indeed, the first column has no name. It's the FBS rank of each team. So we can fix that and mutate in the same step. We'll do that using `rename` and since the field doesn't have a name to rename it, we'll use a position argument. We'll say rename column 1 as Rank.

```
scoringoffense21 <- scoringoffense %>% rename(Rank = 1) %>% mutate(YEAR = 2021)
scoringoffense20 <- scoringoffense20 %>% rename(Rank = 1) %>% mutate(YEAR = 2020)
```

And now, to combine the two tables together length-wise – we need to make long data – we'll use a dpylr function called `bind_rows`. The good thing is bind_rows is simple.

```
combined <- bind_rows(scoringoffense21, scoringoffense20)
```

Note in the environment tab we now have a data frame called combined that has 257 observations – which just so happens to be what 127 from 2020 and 130 from 2021 add up to.

```
head(combined)
```

```
# A tibble: 6 x 11
   Rank Name               G    TD    FG `1XP` `2XP` Safety Points Point~1  YEAR
  <int> <chr>          <int> <int> <int> <int> <int>  <int>  <int>   <dbl> <dbl>
1     1 Ohio State        13    76    20    76     0      1    594    45.7  2021
2     2 Western Kentu~    14    79    23    72     2      0    619    44.2  2021
3     3 Pittsburgh        14    76    17    71     0      1    580    41.4  2021
4     4 Wake Forest       14    72    23    65     4      0    574    41    2021
5     5 Coastal Carol~    13    72     9    65     2      2    532    40.9  2021
6     6 Alabama           15    75    23    71     4      0    598    39.9  2021
# ... with abbreviated variable name 1: `Points/G`
```

33.2 An even more complicated example

What do you do when the table has non-standard headers?

Unfortunately, non-standard means there's no one way to do it – it's going to depend on the table and the headers. But here's one idea: Don't try to make it work.

I'll explain.

Let's try to get season team stats from Sports Reference. If you look at that page, you'll see the problem right away – the headers span two rows, and they repeat. That's going to be all kinds of no good. You can't import that. Dataframes must have names all in one row. If you have two-line headers, you have a problem you have to fix before you can do anything else with it.

First we'll grab the page.

```
url <- "https://www.sports-reference.com/cbb/seasons/2022-school-stats.html"
```

Now, similar to our example above, we'll read the html, use XPath to find the table, and then read that table with a directive passed to it setting the header to FALSE. That tells rvest that there isn't a header row. Just import it as data.

```
stats <- url %>%
  read_html() %>%
  html_nodes(xpath = '//*[@id="basic_school_stats"]') %>%
  html_table(header=FALSE)
```

What we get back is a list of one element (similar to above). So let's pop it out into a data frame.

```
stats <- stats[[1]]
```

And we'll take a look at what we have.

```
head(stats)
```

```
# A tibble: 6 x 38
  X1    X2     X3    X4    X5    X6    X7    X8    X9    X10   X11   X12   X13
  <chr> <chr>  <chr> <chr> <chr> <chr> <chr> <chr> <lgl> <chr> <chr> <lgl> <chr>
1 ""    ""     Over~ Over~ Over~ Over~ Over~ Over~ NA    Conf. Conf. NA    Home
2 "Rk"  "Scho~ G     W     L     W-L%  SRS   SOS   NA    W     L     NA    W
3 "1"   "Abil~ 36    25    11    .694  2.25  -2.09 NA    11    7     NA    14
4 "2"   "Air ~ 29    11    18    .379  -5.67 1.33  NA    4     13    NA    7
```

```
5 "3"   "Akro~ 34    24    10    .706  0.91  -3.93 NA    14    6     NA    12
6 "4"   "Alab~ 30    12    18    .400  -14.~ -8.86 NA    10    8     NA    7
# ... with 25 more variables: X14 <chr>, X15 <lgl>, X16 <chr>, X17 <chr>,
#   X18 <lgl>, X19 <chr>, X20 <chr>, X21 <lgl>, X22 <chr>, X23 <chr>,
#   X24 <chr>, X25 <chr>, X26 <chr>, X27 <chr>, X28 <chr>, X29 <chr>,
#   X30 <chr>, X31 <chr>, X32 <chr>, X33 <chr>, X34 <chr>, X35 <chr>,
#   X36 <chr>, X37 <chr>, X38 <chr>
```

So, that's not ideal. We have headers and data mixed together, and our columns are named X1 to X38. Also note: They're all character fields. Because the headers are interspersed with data, it all gets called character data. So we've got to first rename each field.

```
stats <- stats %>% rename(Rank=X1, School=X2, Games=X3, OverallWins=X4, OverallLosses=X5,
```

Now we have to get rid of those headers interspersed in the data. We can do that with filter that say keep all the stuff that isn't this.

```
stats <- stats %>% filter(Rank != "Rk" & Games != "Overall")
```

And finally, we need to change the file type of all the fields that need it. We're going to use a clever little trick, which goes like this: We're going to use `mutate_at`, which means mutate these fields. The pattern for `mutate_at` is `mutate_at` these variables and do this thing to them. But instead of specifying which of 38 variables we're going to mutate, we're going to specify the one we don't want to change, which is the name of the school. And we just want to convert them to numeric, which is simple. Here's what it looks like:

```
stats <- stats %>% mutate_at(vars(-School), as.numeric)
```

One last thing: Who needs columns called Blank1, Blank2, Blank3, etc?

```
stats <- stats %>% select(-starts_with("Blank"))
```

And just like that, we have a method for getting up to the minute season stats for every team in Division I.

```
head(stats)
```

```
# A tibble: 6 x 33
   Rank School      Games Overa~1 Overa~2 WinPct Overa~3 Overa~4 Confe~5 Confe~6
  <dbl> <chr>       <dbl>   <dbl>   <dbl>  <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
1     1 Abilene Ch~    36      25      11  0.694    2.25   -2.09      11       7
2     2 Air Force      29      11      18  0.379   -5.67    1.33       4      13
```

```
3     3 Akron NCAA     34      24      10  0.706    0.91   -3.93      14       6
4     4 Alabama A&M    30      12      18  0.4    -14.4    -8.86      10       8
5     5 UAB NCAA       35      27       8  0.771   10.9    -0.14      14       4
6     6 Alabama St~    31      10      21  0.323  -12.0    -5.45       8      10
# ... with 23 more variables: HomeWins <dbl>, HomeLosses <dbl>, AwayWins <dbl>,
#   AwayLosses <dbl>, ForPoints <dbl>, OppPoints <dbl>, Minutes <dbl>,
#   FieldGoalsMade <dbl>, FieldGoalsAttempted <dbl>, FieldGoalPCT <dbl>,
#   ThreePointMade <dbl>, ThreePointAttempts <dbl>, ThreePointPct <dbl>,
#   FreeThrowsMade <dbl>, FreeThrowsAttempted <dbl>, FreeThrowPCT <dbl>,
#   OffensiveRebounds <dbl>, TotalRebounds <dbl>, Assists <dbl>, Steals <dbl>,
#   Blocks <dbl>, Turnovers <dbl>, PersonalFouls <dbl>, and abbreviated ...
```

34 Advanced rvest

With the chapter, we learned how to grab one table from one page. But what if you needed more than that? What if you needed hundreds of tables from hundreds of pages? What if you needed to combine one table on one page into a bigger table, but hundreds of times. There's a way to do this, it just takes patience, a lot of logic, a lot of debugging and, for me, a fair bit of swearing.

So what we are after are game by game stats for each college basketball team in America.

We can see from this page that each team is linked. If we follow each link, we get a ton of tables. But they aren't what we need. There's a link to gamelogs underneath the team names.

So we can see from this that we've got some problems.

1. The team name isn't in the table. Nor is the conference.
2. There's a date we'll have to deal with.
3. Non-standard headers and a truly huge number of fields.
4. And how do we get each one of those urls without having to copy them all into some horrible list?

So let's start with that last question first and grab libraries we need.

```
library(tidyverse)
library(rvest)
library(lubridate)
```

First things first, we need to grab the url to each team from that first link.

```
url <- "https://www.sports-reference.com/cbb/seasons/2022-school-stats.html"
```

But notice first, we don't want to grab the table. The table doesn't help us. We need to grab the only *link* in the table. So we can do that by using the table xpath node, then grabbing the anchor tags in the table, then get only the link out of them (instead of the linked text).

```
schools <- url %>%
  read_html() %>%
```

```
  html_nodes(xpath = '//*[@id="basic_school_stats"]') %>%
  html_nodes("a") %>%
  html_attr('href')
```

Notice we now have a list called schools with … 353 elements. That's the number of teams in college basketball, so we're off to a good start. Here's what the fourth element is.

```
schools[4]
```

```
[1] "/cbb/schools/alabama-am/2022.html"
```

So note, that's the relative path to Alabama A&M's team page. By relative path, I mean it doesn't have the root domain. So we need to add that to each request or we'll get no where.

So that's a problem to note.

Before we solve that, let's just make sure we can get one page and get what we need.

We'll scrape Abilene Christian.

To merge all this into one big table, we need to grab the team name and their conference and merge it into the table. But those values come from somewhere else. The scraping works just about the same, but instead of html_table you use html_text.

So the first part of this is reading the html of the page so we don't do that for each element – we just do it once so as to not overwhelm their servers.

The second part is we're grabbing the team name based on it's location in the page.

Third: The conference.

Fourth is the table itself, noting to ignore the headers. The last bit fixes the headers, removes the garbage header data from the table, converts the data to numbers, fixes the date and mutates a team and conference value. It looks like a lot, and it took a bit of twiddling to get it done, but it's no different from what you did for your last homework.

```
page <- read_html("https://www.sports-reference.com/cbb/schools/abilene-christian/2022-gam

team <- page %>%
  html_nodes(xpath = '//*[@id="meta"]/div[2]/h1/span[2]') %>%
  html_text()

conference <- page %>%
    html_nodes(xpath = '//*[@id="meta"]/div[2]/p[1]/a') %>%
    html_text()
```

```
table <- page %>%
  html_nodes(xpath = '//*[@id="sgl-basic"]') %>%
  html_table(header=FALSE)

table <- table[[1]] %>% rename(Game=X1, Date=X2, HomeAway=X3, Opponent=X4, W_L=X5, TeamSco
```

Now what we're left with is how do we do this for ALL the teams. We need to send 353 requests to their servers to get each page. And each url is not the one we have – we need to alter it.

First we have to add the root domain to each request. And, each request needs to go to /2019-gamelogs.html instead of /2019.html. If you look at the urls two the page we have and the page we need, that's all that changes.

What we're going to use is what is known in programming as a loop. We can loop through a list and have it do something to each element in the loop. And once it's done, we can move on to the next thing.

Think of it like a program that will go though a list of your classmates and ask each one of them for their year in school. It will start at one end of the list and move through asking each one "What year in school are you?" and will move on after getting an answer.

Except we want to take a url, add something to it, alter it, then request it and grab a bunch of data from it. Once we're done doing all that, we'll take all that info and cram it into a bigger dataset and then move on to the next one. Here's what that looks like:

```
uri <- "https://www.sports-reference.com"

logs <- tibble()

for (i in schools){
  log_url <- gsub("/2022.html","/2022-gamelogs.html", i)
  school_url <- paste(uri, log_url, sep="")  # creating the url to fetch

  page <- read_html(school_url)

  team <- page %>%
    html_nodes(xpath = '//*[@id="meta"]/div[2]/h1/span[2]') %>%
    html_text()

  conference <- page %>%
    html_nodes(xpath = '//*[@id="meta"]/div[2]/p[1]/a') %>%
    html_text()
```

```
  table <- page %>%
    html_nodes(xpath = '//*[@id="sgl-basic"]') %>%
    html_table(header=FALSE)

table <- table[[1]] %>% rename(Game=X1, Date=X2, HomeAway=X3, Opponent=X4, W_L=X5, TeamSco

  logs <- bind_rows(logs, table)  # binding them all together
  Sys.sleep(3)  # Sys.sleep(3) pauses the loop for 3s so as not to overwhelm website's ser
}
```

The magic here is in `for (i in schools){`. What that says is for each iterator in schools – for each school in schools – do what follows each time. So we take the code we wrote for one school and use it for every school.

This part:

```
log_url <- gsub("/2022.html","/2022-gamelogs.html", i)
school_url <- paste(uri, log_url, sep="")  # creating the url to fetch

page <- read_html(school_url)
```

`log_url` is what changes our school page url to our logs url, and `school_url` is taking that log url and the root domain and merging them together to create the complete url. Then, page just reads that url we created.

What follows that is taken straight from our example of just doing one.

The last bits are using bind_rows to take our data and mash it into a bigger table, over and over and over again until we have them all in a single table. Then, we tell our scraper to wait a few seconds because we don't want our script to machine gun requests at their server as fast as it can go. That's a guaranteed way to get them to block scrapers, and could knock them off the internet. Aggressive scrapers aren't cool. Don't do it.

Lastly, we write it out to a csv file.

```
write_csv(logs, "logs.csv")
```

So with a little programming knowhow, a little bit of problem solving and the stubbornness not to quit on it, you can get a whole lot of data scattered all over the place with not a lot of code.

34.1 One last bit

Most tables that Sports Reference sites have are in plain vanilla HTML. But some of them – particularly player based stuff – are hidden with a little trick. The site puts the data in a comment – where a browser will ignore it – and then uses javascript to interpret the commented data. To a human on the page, it looks the same. To a browswer or a scraper, it's invisible. You'll get errors. How do you get around it?

1. Scrape the comments.
2. Turn the comment into text.
3. Then read that text as html.
4. Proceed as normal.

```
h <- read_html('https://www.baseball-reference.com/leagues/MLB/2017-standard-pitching.shtm

df <- h %>% html_nodes(xpath = '//comment()') %>%    # select comment nodes
    html_text() %>%    # extract comment text
    paste(collapse = '') %>%    # collapse to a single string
    read_html() %>%    # reparse to HTML
    html_node('table') %>%    # select the desired table
    html_table()
```

35 Building your own blog with blogdown

If you listen to the Measurables Podcast for about two episodes, you'll detect a pattern. The host asks each guest how they got started in sports analytics. To a one, they'll say they found public data and started blogging about their analysis of it. For nearly every single guest, this is their path into the field. They started messing around with data in a toolset, found something interesting and wrote a post about what they found and how they found it. Other analysts noticed it, or a hiring manager liked what they read, and the rest is history, as they say.

So, let's do that. Let's get you a blog so you can post your work.

Here's our requirements:

1. This doesn't cost you anything.
2. There's zero maintenance work or upkeep. No servers to manage. No account to pay for.
3. Since you're going to be writing about your code, you should be able to create your blog posts in R Studio.

35.1 Setup

With those requirements in mind, we're going to use a library called Blogdown, which creates blog posts from R Markdown files, similar to what you've been working with in this book.

It installs how you think it should. Go into the console and run this:

```
install.packages('blogdown')
```

After that, run this:

```
blogdown::install_hugo()
```

After that, we're ready to start making a blog. To do that, go to File > New Project and select New Directory (or Version Control if you know what you're doing).

In the Project Type view, you may have to scroll down to find "Website using blogdown" but that's your choice. Mine will probably look different than yours, but below is what the line you're looking for looks like.

And now we've come to our first decision point.

First, name the directory you're going to put this into. Keep it simple. Blog is fine. Then decide where on your computer you're going to put it. Put it somewhere you're going to remember. Don't put it on your Desktop. Put it in a folder. Remember what folder because you're going to need this later.

Now for the big decision: What theme to use. There's four choices, but the default theme is the most fully documented. If you want to use another theme, you'll be better off creating two projects: the one you're going to publish, and the lithium theme you can steal stuff from.

Here's the easiest themes to work with:

- yihui/hugo-lithium
- yihui/hugo-prose
- yihui/hugo-ivy
- yihui/hugo-xmin

When you hit Create Project, you should get an R Studio screen. You might have a file open called config.yaml. If you don't, open it (you can use the files pane in the bottom right). This file will look different depending on which theme you used. This configuration file needs to have a few things in it. This is the config.yaml for my website, but you should use it as a guide for what yours will need:

```
baseurl: https://dwillis.github.io/
languageCode: en-us
title: Derek Willis' GitHub Site
theme: hugo-lithium
googleAnalytics: ''
disqusShortname: ''

ignoreFiles:
  - \.Rmd$
  - \.Rmarkdown$
  - _cache$
  - \.knit\.md$
  - \.utf8\.md$
permalinks:
  post: /:year/:month/:day/:slug/
menu:
  main:
    - name: About
```

```
      url: /about/
    - name: GitHub
      url: https://github.com/dwillis
    - name: Twitter
      url: https://twitter.com/derekwillis
params:

  MathJaxCDN: //cdnjs.cloudflare.com/ajax/libs
  MathJaxVersion: 2.7.5
  description: Tools and teaching materials made by Derek Willis of the University of Marylan
  favicon: favicon.ico
  highlightjsCDN: //cdnjs.cloudflare.com/ajax/libs
  highlightjsLang:
    - r
    - yaml
  highlightjsTheme: GitHub
  highlightjsVersion: 9.12.0
  logo:
    alt: Logo
    height: 50
    url: logo.png
    width: 50
```

The top parts are generally filled in regardless of the theme. You'll obviously want to change the base url and the title. For Google Analytics, you'll add your tracking id there if and when you want to set that up. Leave ignorefiles and permalinks alone. For your menu – that is up to you. If someone wanting to hire you came across your site, what would you include here? Another thing to change? The description. It will be the text that appears below the name in Google.

35.2 Seeing your site

While you are creating your site, you're going to want to see it. Now's a good time to introduce serving your site in R Studio so you can see it before you deploy it.

In the console, type this: `blogdown:: serve_site()` or you can go to Addins > Serve Site.

Now, in the viewer tab on the right, you should see your site.

35.3 Editing existing Markdown content

There's two types of files you can work with – plain Markdown files or R Markdown files. Plain Markdown are generally pages without code. R Markdown are the ones with the code.

Let's start with a simple Markdown file – the homepage of the site. To edit files, use the Files tab. All of the material you'll be editing is in the content folder.

Let's start with editing the about.md file. Some templates also have a _index.md file, which is the static part of the homepage.

At the top, you will have something called a yaml header, which are things that set up the page. They are quite simple to figure out. Here's the yaml header the Ivy theme generates. Obviously, I need to change the author, the date and the title for my site.

```
---
author: Yihui Xie
date: "2017-08-06"
title: About Hugo Ivy
---
```

Change yours, then change the text below the yaml header, and then save it. Click on the Viewer tab and check out your work.

35.4 Creating a new post in R Markdown

You'll notice in your content folder that there is a folder called post, and in there are multiple posts. Depending on your theme, you might just have folders in post, and you might have some markdown files. They work mostly the same way.

The magic here is how you name them. You'll remember from config.yaml that we have a post url format that goes post / year / month / day / slug. The reason for that is that urls matter for Google. Clean urls with meaningful information in them rank higher.

So in post, you'll see the folders are named similarly – a date separated by dashes followed by a slug – the headline of your post in all lower case with dashes instead of spaces. So if I were writing a post called "I love sports data", the slug version of that would be i-love-sports-data. If I were writing it on December 15, which I am, my folder would be 2020-12-15-i-love-sports-data.

NOTE: CAPITALIZATION MATTERS. AS IN, DON'T. You don't see capitals in URLs, so don't use them.

So let's create a post.

The easiest way to do this is to click Addins > New Post. You need to fill out four things on the page: The Title, Author, Date (if you're changing it) and the Format. Be smart about the title – don't but TEST TEST TEST or TITLE HERE because it will build with that and it's hard to change later. Give some thought to the headline on your post before you hit done.

When you're done, hit Done and an RMarkdown file will appear, just as you have been using throughout this book.

The first thing you'll see is the yaml header. Typically, if we've filled out the previous screen correctly, we should be good. But if you need to fix something, you can do it here.

```
---
title: "Sports Data Analysis and Visualization"
author: "Matt Waite"
date: 2019-07-29T21:13:14-05:00
categories: ["Books"]
tags: ["R", "data", "sports"]
---
```

Below the yaml header? That's up to you. Go do some of that writing stuff you do.

35.5 Publishing your site

Blogdown is built on top of the Hugo static site generator. What that means is that Blogdown is going to take your Markdown files and create static html. What does static html mean? It means there's no server creating it on the fly – that's called dynamic html – so this can be hosted on the simplest of servers.

Publishing takes a lot of steps to get set up, but once it is, it's easy.

Step 1: Get a GitHub account

Go to GitHub and sign up for an account. NOTE: Your username will be part of your website address, and you could potentially be sending this to employers. I'm sure blaze420rryday has been your handle since middle school, but do you want an employer to see that? My GitHub user name is mattwaite. See what I'm getting at here?

Step 2: Set up your website repository

For the low low price of nothing, GitHub will host a website for you, and we are going to take them up on that bargain. There's several tricks to getting this to work, but none of them are hard. They just require you to be specific.

To start, you can click on the plus sign in the top right corner or hit the green new button, depending on what your page looks like vs mine (I'm reasonably active GitHub user, so mine will look different from yours).

In the create a new repository page, **the name of your respository needs to follow this pattern**: yourusernamehere.github.io where yourusernamehere is … your username. So my site is mattwaite.github.io because my username is mattwaite. This is why you do not want to select swaggylovedoctor as your username, no matter how attached to it you are. Your employment chances are zero with something dumb like that.

After you've named the repository correctly, leave it public, check add .gitignore, and then select a gitignore template. Type R into the search, select it, and the create repository button will become active. Click create repository.

Step 3:

Don't close your GitHub window.

In a new tab, download the GitHub Desktop App and install it.

GitHub is a version control system and a social network mixed together. Version control is like Track Changes in Word, but on steroids. If you plan on a career in analytics or data science, GitHub is a skill you will have to learn, but it's beyond the scope of this book.

To work around this, we're going to use GitHub Desktop to manage our interactions with GitHub. Open GitHub Desktop and log into your GitHub account in the app.

To get your repository into GitHub Desktop, go back to your tab with your GitHub repository in it. Find the Code button and click it. Click on the Open With GitHub Desktop button.

In the window that pops up, we need to set the Local Path. **This path needs to be the same place your blog project is located.** Click the Choose button and navigate to where your blog project is located. Then, just click Clone.

Step 4:

Let's get your site on the internet.

Switch back to your blog project in R Studio. We're going to output your site and upload it to GitHub.

To do this, we need to change config.yaml. We need to tell your website project that you want to publish to your GitHub folder, not the default.

Open config.yaml in your blog project. Near the top, above ignoreFiles, add this line, replacing yourusername with … your user name. This says publish to your GitHub project that you put in the same folder as your blog project:

```
publishDir: "../yourusername.github.io"
```

Save config.yaml.

Once you are ready to build your site, you need to click on the Build tab and then on Build Website.

If it worked, you should see output in the window that looks something like this (yours will be different from mine):

Step 5

Now go to the GitHub Desktop App. You should see something that looks like this, though yours will be different from mine.

This is where you will commit files to your repository. GitHub is a two-step process for getting files from your computer to GitHub. The first is committing the files. To do that, add a message in the box next to your avatar (where it says update index.html in the screenshot above). Then click Commit to main (mine says master because it's an older repository before GitHub switched to using main by default). The second step is to push it to GitHub. You do that on the screen that appears after you have committed your files. It should say you have No Local Changes, which is good because you sent them to GitHub. Now click Push origin.

In a few minutes, you should be able to go to username.github.io in a browser and you'll see your site. Here's mine.

36 Project Checklist

The main projects in JOUR479X Sports Data Analysis and Visualization involve writing a blog post, created on GitHub pages, about a sports topic that uses code, data and three graphics to tell a story. The assignment is worth 20 percent of the semester grade.

An example of a B+/A- post: Is Nebraska the best worst team in college basketball?

36.1 Writing checklist

1. Have you spell checked your writing?
2. Have you read what you have written out loud? Reading it out loud will help you find bad writing.
3. No really. Read it out loud. It makes a huge difference. If you stumble over your own words, you should rewrite it.
4. If what you read out loud doesn't sound like you, rewrite it.
5. Do you have complete sentences? Do you have active verbs?

36.2 Headline checklist

1. Are your headlines just labels? Example: Maryland's offense. NBA's Best Shooters. The best NFL punters. Those are labels, not headlines.
2. Do your headlines tell a story, or attempt to draw me into one?
3. Do your headlines include words that are what the chart is about?
4. Do your headlines point me toward what I should see in the chart?

36.3 Graphics checklist

1. Do you have the required parts? They are: headline, chatter, credit line, source line.
2. Do your graphics have annotations that help me determine what is going on? Example: Are important dots labeled on a scatterplot?
3. Are there lines to show me averages? Are those lines labeled?

4. Can I read bar chart axis text?

5. Do each one of your charts tell a story? Can I tell what that story is in a glance?
6. Do you use color sparingly to draw my attention to something specific?
7. Is there separation between your headline size and your chatter size?
8. Does your typography have a hierarchy? Does your headline stand out from your chatter? Are your axis labels smaller than your data labels?
9. Have you simplified the theme (i.e. dropped the default grey background)?
10. Are your axis labels something anyone can understand?

36.4 Code checklist

1. Are your variables named what they are? Meaning, did you call your data logs when it's made up of players? That's bad.
2. Is spaced out so it's one line per command? Look at the example post for guidance.
3. Is your code in the proper order? Libraries at the top. Data preparation before the graphics?
4. Did you add message=FALSE, warning=FALSE to any `{r}` blocks that spit out large amounts of automated text? Your library step doesn't need all that gibberish spitting out in your blog.

36.5 Last thing

Can you see your blog post on the internet?

The link you are turning in starts with YourGitHubUserNameHere.github.io. If you are turning in a GitHub.com url, you are turning in the wrong thing.

37 Using packages to get data

There is a growing number of packages and repositories of sports data, largely because there's a growing number of people who want to analyze that data. We've done it ourselves with simple Google Sheets tricks. Then there's RVest, which is a method of scraping the data yourself from websites. But with these packages, someone has done the work of gathering the data for you. All you have to learn are the commands to get it.

One very promising collection of libraries is something called the SportsDataverse, which has a collection of packages covering specific sports, all of which are in various stages of development. Some are more complete than others, but they are all being actively worked on by developers. Packages of interest in this class are:

- cfbfastR, for college football.
- hoopR, for men's professional and college basketball.
- wehoop, for women's professional and college basketball.
- baseballr, for professional and college baseball.
- worldfootballR, for soccer data from around the world.
- hockeyR, for NHL hockey data
- recruitR, for college sports recruiting

Not part of the SportsDataverse, but in the same neighborhood, is nflfastR, which can provide NFL play-by-play data.

Because they're all under development, not all of them can be installed with just a simple `install.packages("something")`. Some require a little work, some require API keys.

The main issue for you is to read the documentation carefully.

37.1 Using cfbfastR as a cautionary tale

cfbfastR presents us a good view into the promise and peril of libraries like this.

First, to make this work, follow the installation instructions and then follow how to get an API key from College Football Data and how to add that to your environment. But maybe wait to do that until you read the whole section.

After installations, we can load it up.

```
library(tidyverse)
library(cfbfastR)
```

You might be thinking, "Oh wow, I can get play by play data for college football. Let's look at what are the five most heartbreaking plays of last year's Maryland season." Because what better way to determine doom than by looking at the steepest drop-off in win probability, which is included in the data.

Great idea. Let's do it. You'll need to make sure that your API key has been added to your environment.

The first thing to do is read the documentation. You'll see that you can request data for each week. For example, here's week 2, which is actually Maryland's third game (the week 0 game is lumped in with week 1).

```
maryland <- cfbd_pbp_data(
 2021,
  week=2,
  season_type = "regular",
  team = "Maryland",
  epa_wpa = TRUE,
)
```

```
* 14:39:55 | Start processing of 1 game...
```

There's not an easy way to get all of a single team's games. A way to do it that's not very pretty but it works is like this:

```
wk1 <- cfbd_pbp_data(2021, week=1, season_type = "regular", team = "Maryland", epa_wpa = T
Sys.sleep(2)
wk2 <- cfbd_pbp_data(2021, week=2, season_type = "regular", team = "Maryland", epa_wpa = T
Sys.sleep(2)
wk3 <- cfbd_pbp_data(2021, week=3, season_type = "regular", team = "Maryland", epa_wpa = T
Sys.sleep(2)
wk4 <- cfbd_pbp_data(2021, week=4, season_type = "regular", team = "Maryland", epa_wpa = T
Sys.sleep(2)
wk5 <- cfbd_pbp_data(2021, week=5, season_type = "regular", team = "Maryland", epa_wpa = T
Sys.sleep(2)
wk6 <- cfbd_pbp_data(2021, week=6, season_type = "regular", team = "Maryland", epa_wpa = T
Sys.sleep(2)
wk8 <- cfbd_pbp_data(2021, week=8, season_type = "regular", team = "Maryland", epa_wpa = T
Sys.sleep(2)
```

```
wk9 <- cfbd_pbp_data(2021, week=9, season_type = "regular", team = "Maryland", epa_wpa = T
Sys.sleep(2)
wk10 <- cfbd_pbp_data(2021, week=10, season_type = "regular", team = "Maryland", epa_wpa =
Sys.sleep(2)
wk11 <- cfbd_pbp_data(2021, week=11, season_type = "regular", team = "Maryland", epa_wpa =
Sys.sleep(2)
wk12 <- cfbd_pbp_data(2021, week=12, season_type = "regular", team = "Maryland", epa_wpa =

umplays <- bind_rows(wk1, wk2, wk3, wk4, wk5, wk6, wk8, wk9, wk10, wk11, wk12)
```

The sys.sleep bits just pauses for two seconds before running the next block. Since we're requesting data from someone else's computer, we want to be kind. Week 7 was a bye week for Maryland, so if you request it, you'll get an empty request and a warning. The `bind_rows` parts puts all the dataframes into a single dataframe.

Now you're ready to look at heartbreak. How do we define heartbreak? How about like this: you first have to lose the game, it comes in the third or fourth quarter, it involves a play (i.e. not a timeout), and it results in the biggest drop in win probability.

```
umplays %>%
  filter(pos_team == "Maryland" & wk > 4 & play_type != "Timeout") %>%
  filter(period == 3 | period == 4) %>%
  mutate(HeartbreakLevel = wp_before - wp_after) %>%
  arrange(desc(HeartbreakLevel)) %>%
  top_n(5, wt=HeartbreakLevel) %>%
  select(period, clock.minutes, def_pos_team, play_type, play_text)
```

```
-- play-by-play data from CollegeFootballData.com ------------ cfbfastR 1.9.0 --

i Data updated: 2022-09-04 14:39:57 EDT

# A tibble: 5 x 5
  period clock.minutes def_pos_team   play_type                  play_text
   <int>         <int> <chr>          <chr>                      <chr>
1      3             0 Penn State     Rush                       Challen Faamat~
2      3            12 Michigan State Interception Return        Taulia Tagovai~
3      3            15 Penn State     Pass Incompletion          Taulia Tagovai~
4      3             4 Penn State     Fumble Recovery (Opponent) Taulia Tagovai~
5      4            11 Penn State     Pass Incompletion          Taulia Tagovai~
```

The most heartbreaking play of the season? A third quarter run for no yards against Penn State Next up: A lot of other plays against Penn State.

37.2 Another example

The wehoop package is mature enough to have a version on CRAN, so you can install it the usual way with `install.packages("wehoop")`. Another helpful library to install is progressr with `install.packages("progressr")`

```
library(wehoop)
```

Many of these libraries have more than play-by-play data. For example, wehoop has box scores and player data for both the WNBA and college basketball. From personal experience, WNBA data isn't hard to get, but women's college basketball is a giant pain.

So, who is Maryland's single season points champion over the last five seasons?

```
progressr::with_progress({
  wbb_player_box <- wehoop::load_wbb_player_box(2018:2022)
})
```

With progressr, you'll see a progress bar in the console, which lets you know that your command is still working, since some of these requests take minutes to complete. Player box scores is quicker – five seasons was a matter of seconds.

If you look at the wbb_player_box data we now have, we have each player in each game over each season – more than 300,000 records. Finding out who Maryland's top 10 single-season scoring leaders are is a matter of grouping, summing and filtering.

```
wbb_player_box %>%
  filter(team_short_display_name == "Maryland") %>%
  group_by(athlete_display_name, season) %>%
  summarise(totalPoints = sum(as.numeric(pts))) %>%
  arrange(desc(totalPoints)) %>%
  ungroup() %>%
  top_n(10, wt=totalPoints)
```

```
# A tibble: 11 x 3
   athlete_display_name season totalPoints
   <chr>                 <int>       <dbl>
 1 Kaila Charles          2018         610
 2 Angel Reese            2022         569
 3 Kaila Charles          2019         557
 4 Ashley Owusu           2021         518
 5 Diamond Miller         2021         501
 6 Taylor Mikesell        2019         448
```

```
 7 Kaila Charles          2020         433
 8 Stephanie Jones        2019         425
 9 Ashley Owusu           2022         385
10 Chloe Bibby            2021         382
11 Chloe Bibby            2022         382
```

This just in: Kaila Charles is good at basketball.