Sports Data Analysis and Visualization

Code, data and visuals for storytellers

Matt Waite & Derek Willis

2025-08-06

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 **25**
5.1 A more complex example . . . 28

6 Filters and selections **31**
6.1 Selecting data to make it easier to read . . . 34
6.2 Using conditional filters to set limits . . . 35
6.3 Top list . . . 37

7 Transforming data **38**
7.1 Making long data wide . . . 40
7.2 Why this matters . . . 41

8 Significance tests **44**
8.1 Accepting the null hypothesis . . . 47

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

10 Multiple regression **55**

11 Residuals **62**
11.1 Fouls . . . 66

12 Z-scores **70**
12.1 Calculating a Z score in R . . . 71
12.2 Writing about z-scores . . . 75

13 Clustering **76**
13.1 Advanced metrics . . . 82

14 Simulations **87**
14.1 Cold streaks . . . 89
14.2 The hottest of streaks . . . 90

15 Intro to ggplot with bar charts **92**
15.1 The bar chart . . . 94
15.2 Scales . . . 97
15.3 Styling . . . 99
15.4 One last trick: coord flip . . . 101

16 Stacked bar charts **103**

17 Circular bar plots **108**
17.1 Does November basketball matter? . . . 109
17.2 Does it show you where you are? . . . 112

18 Waffle charts **115**
18.1 Making waffles with vectors . . . 115

19 Line charts **121**
19.1 This is too simple. . . . 123
19.2 But what if I wanted to add a lot of lines. . . . 126

20 Step charts **129**

21 Dumbbell and lollipop charts **133**
21.1 Dumbbell plots . . . 133
21.2 Lollipop charts . . . 138

22 Scatterplots **142**
22.1 Let's see it fail . . . 146

23 Bubble charts **149**

24 Beeswarm plots **159**
24.1 A few other options . . . 163

25 Bump charts **166**

26 Tables **173**

27 Facet wraps **186**
27.1 Facet grid vs facet wraps . . . 189
27.2 Other types . . . 191

28 Arranging multiple plots together **193**

29 Encircling points on a scatterplot **199**
29.1 A different, more local example . . . 205

30 Text cleaning **208**
30.1 Stripping out text . . . 208
30.2 Another example: splitting columns . . . 210

31 Headlines **215**

32 Annotations **222**

33 Finishing touches **226**
33.1 Graphics vs visual stories . . . 226
33.2 Getting ggplot closer to output . . . 227
33.3 Waffle charts require special attention . . . 235
33.4 Advanced text wrangling . . . 238

34 Intro to rvest **244**
34.1 A slightly more complicated example . . . 245
34.2 An even more complicated example . . . 247

35 Getting CSVs from Sports Reference **250**

36 Building your own blog with Quarto **254**
36.1 Setup . . . 254
36.2 Creating a post . . . 256
36.3 Seeing your site . . . 257

36.4 Publishing your site 257

37 Project Checklist **262**
37.1 Writing checklist 262
37.2 Headline checklist 262
37.3 Graphics checklist 262
37.4 Code checklist 263
37.5 Last thing 263

38 Using packages to get data **264**
38.1 Using cfbfastR as a cautionary tale 264
38.2 Another example 266

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. … produced 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 RStudio 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 RStudio. When you see that, you'll know what to do: click the green arrow.

1.2 About this book

This book is the collection of class materials for the Fall 2024 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 2024 & Matt Waite 2019-2023.
- 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					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			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

Column

Rows

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 (type `sheet.new` in the url bar). In the first cell of the first row, copy and paste this formula in:

```
=IMPORTHTML("https://www.sports-reference.com/cbb/schools/maryland/men/2025-gamelogs.html", "
```

The spreadsheet might request that you "allow access" to complete the import. You should. 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/nbaadvancedplayers2425.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.

```r
nbaplayers <- read_csv("data/nbaadvancedplayers2425.csv")
```

```
Rows: 735 Columns: 30
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (4): Player, Team, Pos, Player-additional
dbl (25): Rk, Age, G, GS, MP, PER, TS%, 3PAr, FTr, ORB%, DRB%, TRB%, AST%, S...
lgl  (1): Awards

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.
```

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 30
     Rk Player        Age Team  Pos       G    GS    MP   PER `TS%` `3PAr`   FTr
  <dbl> <chr>       <dbl> <chr> <chr> <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl> <dbl>
1     1 Mikal Brid~    28 NYK   SF       82    82  3036  14   0.585  0.391 0.1
2     2 Josh Hart      29 NYK   SG       77    77  2897  16.5 0.611  0.327 0.266
3     3 Anthony Ed~    23 MIN   SG       79    79  2871  20.1 0.595  0.503 0.308
4     4 Devin Book~    28 PHO   SG       75    75  2795  19.3 0.589  0.388 0.34
5     5 James Hard~    35 LAC   PG       79    79  2789  20   0.582  0.516 0.446
6     6 DeMar DeRo~    35 SAC   SF       77    77  2768  17.7 0.569  0.196 0.337
# i 18 more variables: `ORB%` <dbl>, `DRB%` <dbl>, `TRB%` <dbl>, `AST%` <dbl>,
#   `STL%` <dbl>, `BLK%` <dbl>, `TOV%` <dbl>, `USG%` <dbl>, OWS <dbl>,
#   DWS <dbl>, WS <dbl>, `WS/48` <dbl>, OBPM <dbl>, DBPM <dbl>, BPM <dbl>,
#   VORP <dbl>, Awards <lgl>, `Player-additional` <chr>
```

```r
nrow(nbaplayers)
```

```
[1] 735
```

Another way to look at nrow – we have 735 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 735 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: 5 x 2
  Pos   total
  <chr> <int>
1 C       136
2 PF      134
3 PG      135
4 SF      139
5 SG      191
```

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 position with the most players, we need to sort it in descending order. That looks like this:

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

```
# A tibble: 5 x 2
  Pos   total
  <chr> <int>
1 SG      191
2 SF      139
3 C       136
4 PG      135
5 PF      134
```

So the most common position in the NBA? Shooting guard, followed by small forward.

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(Team, Pos) |>
  summarise(
    total = n()
  ) |> arrange(desc(total))
```

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

```
# A tibble: 158 x 3
# Groups:   Team [32]
   Team  Pos   total
   <chr> <chr> <int>
 1 2TM   SG       20
```

```
 2 2TM   PG       15
 3 2TM   C        14
 4 2TM   PF       14
 5 2TM   SF       14
 6 CHO   SG       11
 7 OKC   SG        8
 8 ATL   SG        7
 9 CLE   SG        7
10 GSW   SG        7
# i 148 more rows
```

So wait, what team is 2TM?

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 2TM, meaning Total. Is there a team abbreviated 2TM? No. So ignore them here.

Charlotte had 11 shooting 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.

```r
nbaplayers |>
  group_by(Pos) |>
  summarise(
    count = n(),
    mean_minutes = mean(MP),
    median_minutes = median(MP)
  )
```

```
# A tibble: 5 x 4
  Pos   count mean_minutes median_minutes
  <chr> <int>        <dbl>          <dbl>
```

```
1 C       136         856.           606.
2 PF      134         894.           724
3 PG      135         962.           717
4 SF      139         866.           648
5 SG      191         935.           732
```

Let's look at centers. The average center plays 855 minutes and the median is 606 minutes.

Why?

Let's let sort help us.

```r
nbaplayers |> arrange(desc(MP))
```

```
# A tibble: 735 x 30
      Rk Player       Age Team  Pos       G    GS    MP   PER `TS%` `3PAr`   FTr
   <dbl> <chr>      <dbl> <chr> <chr> <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl> <dbl>
 1     1 Mikal Bri~    28 NYK   SF       82    82  3036  14   0.585  0.391 0.1
 2     2 Josh Hart     29 NYK   SG       77    77  2897  16.5 0.611  0.327 0.266
 3     3 Anthony E~    23 MIN   SG       79    79  2871  20.1 0.595  0.503 0.308
 4     4 Devin Boo~    28 PHO   SG       75    75  2795  19.3 0.589  0.388 0.34
 5     5 James Har~    35 LAC   PG       79    79  2789  20   0.582  0.516 0.446
 6     6 DeMar DeR~    35 SAC   SF       77    77  2768  17.7 0.569  0.196 0.337
 7     7 Trae Young    26 ATL   PG       76    76  2739  18.3 0.567  0.467 0.408
 8     8 Tyler Her~    25 MIA   SG       77    77  2725  19.7 0.605  0.486 0.237
 9     9 OG Anunoby    27 NYK   PF       74    74  2706  15.4 0.591  0.448 0.22
10    10 Jalen Gre~    22 HOU   SG       82    82  2697  15.1 0.544  0.46  0.234
# i 725 more rows
# i 18 more variables: `ORB%` <dbl>, `DRB%` <dbl>, `TRB%` <dbl>, `AST%` <dbl>,
#   `STL%` <dbl>, `BLK%` <dbl>, `TOV%` <dbl>, `USG%` <dbl>, OWS <dbl>,
#   DWS <dbl>, WS <dbl>, `WS/48` <dbl>, OBPM <dbl>, DBPM <dbl>, BPM <dbl>,
#   VORP <dbl>, Awards <lgl>, `Player-additional` <chr>
```

The player with the most minutes on the floor is a small forward. So that means there's Mikal Bridges rolling up 3,036 minutes in a season, and then there's OKC sensation Alex Reese. Never heard of Alex Reese? Might be because he logged two minute 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.

```
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: 5 x 6
  Pos    total avgminutes minminutes maxminutes stdev
  <chr>  <dbl>      <dbl>      <dbl>      <dbl> <dbl>
1 SG    178577       935.          3       2897  786.
2 PG    129889       962.          4       2789  791.
3 SF    120323       866.          5       3036  795.
4 PF    119786       894.          2       2706  760.
5 C     116389       856.          3       2674  734.
```

So again, no surprise, shooting guards spend the most minutes on the floor in the NBA. They average 934 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: 146 Columns: 14
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (2): Institution, Conference
dbl (12): 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, ...

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`.

```r
head(attendance)
```

```
# A tibble: 6 x 14
  Institution Conference `2013` `2014` `2015` `2016` `2017` `2018` `2019` `2020`
  <chr>       <chr>       <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 Air Force   MWC        228562 168967 156158 177519 174924 166205 162505   5600
2 Akron       MAC        107101  55019 108588  62021 117416  92575 107752     NA
3 Alabama     SEC        710538 710736 707786 712747 712053 710931 707817  97120
4 Appalachia~ FBS Indep~ 149366     NA     NA     NA     NA     NA     NA     NA
5 Appalachia~ Sun Belt       NA 138995 128755 156916 154722 131716 166640     NA
6 Arizona     Big 12     285713 354973 308355 338017 255791 318051 237194     NA
# i 4 more variables: `2021` <dbl>, `2022` <dbl>, `2023` <dbl>, `2024` <dbl>
```

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 '2024' as the new data, and we'll use '2023' 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.

```r
attendance |> mutate(
  change = (`2024` - `2023`)/`2023`
)
```

```
# A tibble: 146 x 15
   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         Big 12       285713 354973 308355 338017 255791 318051 237194
 7 Arizona St.     Big 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 The American 169781 171310 185946 163267 185543 190156 185935
# i 136 more rows
# i 6 more variables: `2020` <dbl>, `2021` <dbl>, `2022` <dbl>, `2023` <dbl>,
#   `2024` <dbl>, change <dbl>
```

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 = ((`2024` - `2023`)/`2023`)*100
)
```

```
# A tibble: 146 x 15
   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         Big 12       285713 354973 308355 338017 255791 318051 237194
 7 Arizona St.     Big 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 The American 169781 171310 185946 163267 185543 190156 185935
# i 136 more rows
# i 6 more variables: `2020` <dbl>, `2021` <dbl>, `2022` <dbl>, `2023` <dbl>,
#   `2024` <dbl>, change <dbl>
```

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

```
attendance |> mutate(
  change = ((`2024` - `2023`)/`2023`)*100
) |> arrange(desc(change))
```

```
# A tibble: 146 x 15
   Institution   Conference   `2013` `2014` `2015` `2016` `2017` `2018` `2019`
   <chr>         <chr>         <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 1 SMU           ACC          112347 129169 147301 142272 139609 116299 141798
 2 Ohio St.      Big Ten      734528 744075 750705 750944 752464 713630 723679
 3 Colorado St.  MWC          111598 159450 149500 165598 192369 177025 140025
 4 Indiana       Big Ten      354823 249941 310195 301190 263715 286753 247463
 5 Texas         SEC          593857 564618 540210 587283 556667 586277 577834
 6 Central Mich. MAC           66119  97838  94029 104447  67520  77038  81386
 7 North Texas   The American 126182 115627  68155 119269 134174 140131 128150
 8 Pittsburgh    ACC          348188 289204 288900 322531 254062 250178 303606
```

```
 9 Northern Ill. MAC          103344  67813  83649  55095  67748  52019  42590
10 Vanderbilt    SEC          249728 274063 192802 187451 219390 196313 184016
# i 136 more rows
# i 6 more variables: `2020` <dbl>, `2021` <dbl>, `2022` <dbl>, `2023` <dbl>,
#   `2024` <dbl>, change <dbl>
```

So who had the most growth in 2024 compared to the year before? SMU, followed by Ohio State and Colorado State. How about Central Michigan at #6!

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 2024-25 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:

```r
players <- read_csv("data/players25.csv")
```

```
Rows: 5818 Columns: 57
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (10): Team, Player, Class, Height, Hometown, High School, Summary, Pos, ...
dbl (47): #, Weight, G, GS, MP, FG, FGA, FG%, 3P, 3PA, 3P%, 2P, 2PA, 2P%, eF...

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: 5,818 x 58
   Team    Player   `#` Class Height Weight Hometown `High School` Summary     G
   <chr>   <chr>  <dbl> <chr> <chr>   <dbl> <chr>    <chr>         <chr>   <dbl>
 1 Wiscon~ Isaac~    15 JR    6-2       165 Oregon,~ Oregon (WI)   0.2 Pt~    13
 2 TCU Ho~ Cole ~    35 SR    6-3       180 San Jos~ Bellarmine C~ 3.0 Pt~     1
 3 TCU Ho~ Drew ~    30 FR    6-2       180 Dallas,~ Highland Par~ 1.5 Pt~     2
 4 Stanfo~ Derin~     1 SO    6-4       190 Istanbu~ The Ashevill~ 0.6 Pt~     5
 5 St. Bo~ Jack ~    15 SO    6-0       172 Olean, ~ Olean (NY)    3.0 Pt~     1
 6 Seattl~ Eric ~    19 FR    5-10      167 Beijing~ Ruamrudee In~ 3.0 Pt~     1
 7 Samfor~ Corey~    10 FR    6-4       175 Fairbur~ Landmark Chr~ 1.5 Pt~     2
 8 Queens~ Aneek~    85 SO    5-10      145 Foster ~ San Mateo (C~ 3.0 Pt~     1
 9 Old Do~ CJ Pa~    15 FR    6-6       182 Norfolk~ Maury (VA)    1.0 Pt~     3
10 Oklaho~ Jake ~    30 SR    6-3       187 Norman,~ Loyola Acade~ 0.4 Pt~     7
# i 5,808 more rows
# i 48 more variables: GS <dbl>, MP <dbl>, FG <dbl>, FGA <dbl>, `FG%` <dbl>,
#   `3P` <dbl>, `3PA` <dbl>, `3P%` <dbl>, `2P` <dbl>, `2PA` <dbl>, `2P%` <dbl>,
#   `eFG%` <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>,
#   Pos <chr>, PER <dbl>, `TS%` <dbl>, `3PAr` <dbl>, FTr <dbl>, PProd <dbl>,
#   `ORB%` <dbl>, `DRB%` <dbl>, `TRB%` <dbl>, `AST%` <dbl>, `STL%` <dbl>, ...
```

You'll be forgiven if you did not hear about Winthrop's shooting sensation Henry Harrison. He played in seven games, took three shots and actually hit them all. They all happened to be three pointers, which is three 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 2023-24 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: 146 Columns: 14
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (2): Institution, Conference
dbl (12): 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, ...

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 14
  Institution Conference `2013` `2014` `2015` `2016` `2017` `2018` `2019` `2020`
  <chr>       <chr>       <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 Maryland    ACC        288946     NA     NA     NA     NA     NA     NA     NA
2 Maryland    Big Ten        NA 281884 310389 237690 237859 201562 226871     NA
# i 4 more variables: `2021` <dbl>, `2022` <dbl>, `2023` <dbl>, `2024` <dbl>
```

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

```
attendance |> filter(`2024` >= 500000)
```

```
# A tibble: 17 x 14
   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 Oklahoma       SEC           508334 510972 512139 521142 519119 607146 499533
12 Penn St.       Big Ten       676112 711358 698590 701800 746946 738396 739747
13 South Carolina SEC           576805 569664 472934 538441 550099 515396 545737
14 Tennessee      SEC           669087 698276 704088 706776 670454 650887 702912
15 Texas          SEC           593857 564618 540210 587283 556667 586277 577834
16 Texas A&M      SEC           697003 630735 725354 713418 691612 698908 711258
17 Wisconsin      Big Ten       552378 556642 546099 476144 551766 540072 535301
# i 5 more variables: `2020` <dbl>, `2021` <dbl>, `2022` <dbl>, `2023` <dbl>,
#   `2024` <dbl>
```

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 14
   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        Big 12     285713 354973 308355 338017 255791 318051 237194
 3 Arizona St.    Big 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 BYU            Big 12     367349 342843 351191 351413 337599 314855 357281
 9 California     ACC        345303 286051 292797 279769 219290 300061 254597
10 Cincinnati     Big 12     190624 173037 222578 235095 170603 183112 215908
# i 41 more rows
# i 5 more variables: `2020` <dbl>, `2021` <dbl>, `2022` <dbl>, `2023` <dbl>,
#   `2024` <dbl>
```

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: 69 x 14
   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        Big 12     285713 354973 308355 338017 255791 318051 237194
 3 Arizona St.    Big 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 BYU            Big 12     367349 342843 351191 351413 337599 314855 357281
 9 California     ACC        345303 286051 292797 279769 219290 300061 254597
10 Cincinnati     Big 12     190624 173037 222578 235095 170603 183112 215908
# i 59 more rows
# i 5 more variables: `2020` <dbl>, `2021` <dbl>, `2022` <dbl>, `2023` <dbl>,
#   `2024` <dbl>
```

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.

```r
attendance |> filter(Conference %in% powerfive) |> select(Institution, Conference, `2024`)
```

```
# A tibble: 69 x 3
   Institution    Conference `2024`
   <chr>          <chr>       <dbl>
 1 Alabama        SEC        700539
 2 Arizona        Big 12     327230
 3 Arizona St.    Big 12     293901
 4 Arkansas       SEC        480478
 5 Auburn         SEC        704344
 6 Baylor         Big 12     253810
 7 Boston College ACC        279236
 8 BYU            Big 12     377091
 9 California     ACC        274214
10 Cincinnati     Big 12     219688
# i 59 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/players25.csv")
```

```
Rows: 5818 Columns: 57
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (10): Team, Player, Class, Height, Hometown, High School, Summary, Pos, ...
dbl (47): #, Weight, G, GS, MP, FG, FGA, FG%, 3P, 3PA, 3P%, 2P, 2PA, 2P%, eF...

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: 5,818 x 58
   Team    Player   `#` Class Height Weight Hometown `High School` Summary     G
   <chr>   <chr>  <dbl> <chr> <chr>   <dbl> <chr>    <chr>         <chr>   <dbl>
 1 Wiscon~ Isaac~    15 JR    6-2       165 Oregon,~ Oregon (WI)   0.2 Pt~    13
 2 TCU Ho~ Cole ~    35 SR    6-3       180 San Jos~ Bellarmine C~ 3.0 Pt~     1
 3 TCU Ho~ Drew ~    30 FR    6-2       180 Dallas,~ Highland Par~ 1.5 Pt~     2
 4 Stanfo~ Derin~     1 SO    6-4       190 Istanbu~ The Ashevill~ 0.6 Pt~     5
 5 St. Bo~ Jack ~    15 SO    6-0       172 Olean, ~ Olean (NY)    3.0 Pt~     1
 6 Seattl~ Eric ~    19 FR    5-10      167 Beijing~ Ruamrudee In~ 3.0 Pt~     1
 7 Samfor~ Corey~    10 FR    6-4       175 Fairbur~ Landmark Chr~ 1.5 Pt~     2
 8 Queens~ Aneek~    85 SO    5-10      145 Foster ~ San Mateo (C~ 3.0 Pt~     1
 9 Old Do~ CJ Pa~    15 FR    6-6       182 Norfolk~ Maury (VA)    1.0 Pt~     3
10 Oklaho~ Jake ~    30 SR    6-3       187 Norman,~ Loyola Acade~ 0.4 Pt~     7
```

```
# i 5,808 more rows
# i 48 more variables: GS <dbl>, MP <dbl>, FG <dbl>, FGA <dbl>, `FG%` <dbl>,
#   `3P` <dbl>, `3PA` <dbl>, `3P%` <dbl>, `2P` <dbl>, `2PA` <dbl>, `2P%` <dbl>,
#   `eFG%` <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>,
#   Pos <chr>, PER <dbl>, `TS%` <dbl>, `3PAr` <dbl>, FTr <dbl>, PProd <dbl>,
#   `ORB%` <dbl>, `DRB%` <dbl>, `TRB%` <dbl>, `AST%` <dbl>, `STL%` <dbl>, ...
```

In that season, we've got several players that can lay claim to the title of 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.

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

```
# A tibble: 3,682 x 58
   Team    Player   `#` Class Height Weight Hometown `High School` Summary     G
   <chr>   <chr>  <dbl> <chr> <chr>   <dbl> <chr>    <chr>         <chr>   <dbl>
 1 Arizon~ Conra~    55 SO    6-0       165 Granoll~ Club Joventu~ 1.6 Pt~    22
 2 Syracu~ Nahee~    10 SR    7-4       248 Philade~ Plymouth Whi~ 2.4 Pt~    18
 3 Butler~ Augus~     0 SO    6-8       220 Salvado~ NBA Academy ~ 5.1 Pt~    10
 4 UConn ~ Samso~    35 SR    6-10      205 Lomé, T~ The Patrick ~ 7.5 Pt~    34
 5 Dayton~ Isaac~    13 JR    6-11      251 Port Al~ Alberni Dist~ 3.2 Pt~    21
 6 Utah S~ Isaac~    23 FR    6-8       240 Idaho F~ Hillcrest (I~ 2.1 Pt~    19
 7 Northw~ Lado ~     0 SR    6-10      215 Dallas,~ Dallas Lakel~ 2.4 Pt~    18
 8 Purdue~ Will ~    44 SO    7-2       260 Stockho~ Riksbasketgy~ 1.8 Pt~    22
 9 Baylor~ Josh ~    17 JR    6-10      230 Asaba, ~ NBA Academy ~ 7.4 Pt~    23
10 Milwau~ Dariu~    34 SR    6-8       225 Murray,~ Murray (KY)   3.0 Pt~    32
# i 3,672 more rows
# i 48 more variables: GS <dbl>, MP <dbl>, FG <dbl>, FGA <dbl>, `FG%` <dbl>,
#   `3P` <dbl>, `3PA` <dbl>, `3P%` <dbl>, `2P` <dbl>, `2PA` <dbl>, `2P%` <dbl>,
#   `eFG%` <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>,
#   Pos <chr>, PER <dbl>, `TS%` <dbl>, `3PAr` <dbl>, FTr <dbl>, PProd <dbl>,
#   `ORB%` <dbl>, `DRB%` <dbl>, `TRB%` <dbl>, `AST%` <dbl>, `STL%` <dbl>, ...
```

Now you get Radford's TJ NeSmith, who played in seven games and was on the floor for 113 minutes. So he played a little bit, but not a lot. But in that time, he only attempted 32 shots, and made 75 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, `2024`) |>
```

```
Selecting by 2024
```

```
# A tibble: 10 x 3
   Institution Conference `2024`
   <chr>       <chr>       <dbl>
 1 Ohio St.    Big Ten    936550
 2 Michigan    Big Ten    884382
 3 Penn St.    Big Ten    864665
 4 Texas       SEC        817852
 5 Texas A&M   SEC        719927
 6 Tennessee   SEC        713405
 7 LSU         SEC        708645
 8 Auburn      SEC        704344
 9 Alabama     SEC        700539
10 Florida     SEC        630116
```

That's all there is to it. Just remember – for it to work correctly, you need to arrange 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: 146 Columns: 14
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (2): Institution, Conference
dbl (12): 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, ...

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: 146 x 14
   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         Big 12       285713 354973 308355 338017 255791 318051 237194
 7 Arizona St.     Big 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 The American 169781 171310 185946 163267 185543 190156 185935
# i 136 more rows
# i 5 more variables: `2020` <dbl>, `2021` <dbl>, `2022` <dbl>, `2023` <dbl>,
#   `2024` <dbl>
```

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 = "Attendan
```

```
# A tibble: 1,752 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 Air Force   MWC        2020        5600
 9 Air Force   MWC        2021      136984
10 Air Force   MWC        2022      188482
# i 1,742 more rows
```

We've gone from 147 rows to more than 1,700, but that's expected when we have 10+ 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", values_to

longdata
```

```
# A tibble: 1,752 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 Air Force   MWC        2020        5600
 9 Air Force   MWC        2021      136984
10 Air Force   MWC        2022      188482
# i 1,742 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.

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

```
Warning: Values from `Attendance` are not uniquely identified; output will contain
list-cols.
* Use `values_fn = list` to suppress this warning.
* Use `values_fn = {summary_fun}` to summarise duplicates.
* Use the following dplyr code to identify duplicates.
  {data} |>
  dplyr::summarise(n = dplyr::n(), .by = c(Institution, Conference, Year)) |>
  dplyr::filter(n > 1L)
```

```
# A tibble: 144 x 14
   Institution     Conference   `2013` `2014` `2015` `2016` `2017` `2018` `2019`
   <chr>           <chr>        <list> <list> <list> <list> <list> <list> <list>
 1 Air Force       MWC          <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 2 Akron           MAC          <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 3 Alabama         SEC          <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 4 Appalachian St. FBS Indepen~ <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 5 Appalachian St. Sun Belt     <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 6 Arizona         Big 12       <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 7 Arizona St.     Big 12       <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 8 Arkansas        SEC          <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 9 Arkansas St.    Sun Belt     <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
10 Army West Point The American <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
# i 134 more rows
# i 5 more variables: `2020` <list>, `2021` <list>, `2022` <list>,
#   `2023` <list>, `2024` <list>
```

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).

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

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

```
ggplot(maryland, aes(x=Year, y=Attendance, group=1)) +
  geom_line() +
  scale_y_continuous(labels = scales::comma) +
  labs(x="Year", y="Attendance", title="Attendance Up and Down Under Locksley", subtitle="Hea
  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"
  )
```

Attendance Up and Down Under Locksley
Heading back to the 300,000 mark?
300,000
275,000
250,000
225,000
200,000
2013
2014
2015
2016
2017
2018
2019
2021
2022
2023
2024
Year
Source: NCAA | By Derek Willis

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`.

```r
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:

```r
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 - Op
```

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 2024 women's college soccer season.

Then load the tidyverse.

```
library(tidyverse)
```

Now import the data.

```
correlations <- read_csv("https://raw.githubusercontent.com/dwillis/NCAAWomensSoccerData/mai
```

```
Rows: 6451 Columns: 36
-- Column specification --------------------------------------------------------
Delimiter: ","
chr   (6): team, opponent, home_away, outcome, minutes, defensive_minutes
dbl  (29): team_score, opponent_score, overtime, gp, goals, assists, points,...
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 college soccer 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 fouls influence the number of goals they score in a season? How much difference can we explain in goals with fouls?

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 fouls are bad. Fouls 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` which is the difference between `goals` and `defensive_goals`, and then we'll use correlations to see how related those two things are.

```
newcorrelations <- correlations |>
  mutate(
    differential = goals - defensive_goals
    )
```

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 `fouls`, which is the yards of penalties a team gets in a game. We do that by referencing `differential` and `fouls` and specifying we want a `pearson` correlation. The number we get back is the correlation coefficient.

```
newcorrelations |> summarise(correlation = cor(differential, fouls, method="pearson"))
```

```
# A tibble: 1 x 1
  correlation
        <dbl>
1      0.0181
```

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, fouls and whether or not the team scores more goals than it gives up are at 0.01806104 You could say they're 1.8 percent related toward the positive – more fouls, the higher your differential. Another way to say it? They're almost 98 percent not related.

What about the number of yellow cards instead of fouls? Do more aggressive defensive teams also score more?

```
newcorrelations |> summarise(correlation = cor(differential, yellow_cards, method="pearson"))
```

```
# A tibble: 1 x 1
  correlation
        <dbl>
1     -0.0723
```

So wait, what does this all mean?

It means that when you look at every game in college soccer, the number of goals and yellow cards do have a small impact on the score difference between your team and the other team. But the relationship is barely anything at all. For yellow cards, it's a negative correlation of about 7 percent. So like 93+ percent plus not related. So neither fouls nor yellow cards have much of any relationship with the difference in goal-scoring.

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 put this fully to rest.

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 fouls, 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 fouls. 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 ~ fouls, data = newcorrelations)
summary(fit)
```

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

Residuals:
     Min       1Q   Median       3Q      Max 
-10.0486  -1.0714  -0.0373   0.9969  10.0196 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
```

```
(Intercept) -0.065158   0.077276  -0.843    0.399
fouls        0.011381   0.007845   1.451    0.147

Residual standard error: 2.264 on 6449 degrees of freedom
Multiple R-squared:  0.0003262, Adjusted R-squared:  0.0001712
F-statistic: 2.104 on 1 and 6449 DF,  p-value: 0.1469
```

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 fouls, the p-value is 0.1469. 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 fouls and a team's aggregate point differential are **not statistically significant**. The differences in score difference and fouls 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 fouls. Our correlation coefficient said they're not quite 2 percent related to each other, but fouls' ability to explain the difference between teams? About 0.0001712 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 -0.065158 and the fouls coefficient is 0.011381. 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 0.011381. and b is -0.065158. So we would multiply a teams total fouls by 0.011381 and then add -0.065158 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? Almost none. Did I say we should quit? Yeah.

So fouls are totally meaningless to the outcome of a game.

You can see the problem in a graph. On the X axis is fouls, on the y is point differential. 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 fouls and a positive point differential. On right would be teams with high fouls and negative point differentials. Do you see that below?

9.1 A more predictive example

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

So instead of looking at fouls, let's make a new metric: shots on goal. Can we predict the score differential by looking at the shots they put on goal?

First, let's look at the correlation coefficent.

```
newcorrelations |>
  summarise(correlation = cor(differential, so_g, method="pearson"))
```

```
# A tibble: 1 x 1
  correlation
        <dbl>
1       0.602
```

Answer: 60 percent. Not a perfect relationship, but pretty good. But how meaningful is that relationship and how predictive is it?

```r
net <- lm(differential ~ so_g, data = newcorrelations)
summary(net)
```

```

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

Residuals:
    Min      1Q  Median      3Q     Max 
-9.2804 -1.0389  0.0988  1.2365  5.8234 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -2.13264    0.04233  -50.38   <2e-16 ***
so_g         0.37923    0.00626   60.58   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.808 on 6449 degrees of freedom
Multiple R-squared:  0.3627,	Adjusted R-squared:  0.3626 
F-statistic:  3670 on 1 and 6449 DF,  p-value: < 2.2e-16
```

First we check p-value: 0.00000000000000022. 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 soccer knows this is true. It makes intuitive sense.

Second, Adjusted R-squared: 0.3626. So we can predict about 36 percent of the difference in the score differential by simply looking at the shots on goal the team has.

Third, the coefficients: In this case, our `y=mx+b` formula looks like `y = 0.37923x + -2.13264`. So if we were applying this, let's look at Maryland's 3-2 loss to George Mason on Sept. 1, 2024. Maryland's shots on goal in that game? 9. What does our model say the point differential should have been?

```r
(0.37923*9)+-2.13264
```

```
[1] 1.28043
```

So by our model, Maryland should have won by at least one goal. Some games are closer than others. But when you can explain 36 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 men's college basketball games between 2015 and 2025.

We need the tidyverse.

```
library(tidyverse)
```

And the data.

```
logs <- read_csv("https://dwillis.github.io/sports-data-files/cbblogs1525.csv")
```

```
Rows: 110070 Columns: 60
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (10): Season, TeamFull, Opponent, HomeAway, W_L, URL, Conference, Team,...
dbl  (49): 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.
```

One way to show how successful a basketball 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 add in net rebounds. And because we'll need it later, let's add the turnover margin.

```r
logs <- logs |> mutate(
  Differential = TeamScore - OpponentScore,
  NetRebounds = TeamTotalRebounds - OpponentTotalRebounds,
  TurnoverMargin = TeamTurnovers - OpponentTurnovers)
```

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 turnovers.

```r
rebounds <- lm(Differential ~ NetRebounds, data=logs)
summary(rebounds)
```

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

Residuals:
    Min      1Q  Median      3Q     Max 
-59.670  -8.589  -0.254   8.244  77.991 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 0.672182   0.039281   17.11   <2e-16 ***
NetRebounds 1.083550   0.004037  268.42   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 13.01 on 110063 degrees of freedom
  (5 observations deleted due to missingness)
Multiple R-squared:  0.3956,	Adjusted R-squared:  0.3956 
F-statistic: 7.205e+04 on 1 and 110063 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 about 39 percent of the difference in differential with the net rebounds 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 turnover margin to our prediction model:

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

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

Residuals:
    Min      1Q  Median      3Q     Max 
-46.255  -6.895  -0.058   6.828  49.013 

Coefficients:
                Estimate Std. Error t value Pr(>|t|)    
(Intercept)     0.316402   0.031167   10.15   <2e-16 ***
NetRebounds     1.206137   0.003236  372.77   <2e-16 ***
TurnoverMargin -1.565739   0.006136 -255.19   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 10.31 on 110062 degrees of freedom
  (5 observations deleted due to missingness)
Multiple R-squared:  0.6203,    Adjusted R-squared:  0.6203 
F-statistic: 8.99e+04 on 2 and 110062 DF,  p-value: < 2.2e-16
```

First things first: What is the adjusted R-squared? This model predicts about 62 percent of the differential.

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

Third: Compare the residual standard error. We went from 13 to 10.3. 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 basketball. We can't say this will work for every basketball 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 all the columns with numeric values, except for Game, and then show me the differential, net yards, turnover margin and then everything else.

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

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 NetRebounds TurnoverMargin   TeamFGPCT
Differential             1.0000000   0.6289884    -0.37535059  0.60950692
NetRebounds              0.6289884   1.0000000     0.14846725  0.36955761
TurnoverMargin          -0.3753506   0.1484672     1.00000000 -0.03443205
TeamFGPCT                0.6095069   0.3695576    -0.03443205  1.00000000
TeamTotalRebounds        0.4935285   0.7541971     0.09234795  0.02975520
OpponentFGPCT           -0.6179621  -0.3802828     0.03821898 -0.11788392
OpponentTotalRebounds   -0.4332605  -0.7208197    -0.12773386 -0.52924893
                      TeamTotalRebounds OpponentFGPCT OpponentTotalRebounds
Differential                 0.49352854  -0.617962063          -0.433260526
NetRebounds                  0.75419707  -0.380282772          -0.720819716
TurnoverMargin               0.09234795   0.038218977          -0.127733865
TeamFGPCT                    0.02975520  -0.117883924          -0.529248931
TeamTotalRebounds            1.00000000  -0.549069900          -0.088502459
OpponentFGPCT               -0.54906990   1.000000000          -0.002714132
OpponentTotalRebounds       -0.08850246  -0.002714132           1.000000000
```

Notice right away – NetRebounds is highly correlated. But NetRebounds is also highly correlated with TeamTotalRebounds. And that makes sense: TeamTotalRebounds feeds into NetRebounds. Including both 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? Add or remove some of the columns above and re-run the correlation matrix.

We can add more just by simply adding them. Let's add the average FG PCT for both the team and opponent. They're correlated to Differential, but not as much as you might expect.

```
model2 <- lm(Differential ~ NetRebounds + TurnoverMargin + TeamFGPCT + OpponentFGPCT, data=lo
summary(model2)
```

```
Call:
lm(formula = Differential ~ NetRebounds + TurnoverMargin + TeamFGPCT +
    OpponentFGPCT, data = logs)

Residuals:
    Min      1Q  Median      3Q     Max
-26.166  -3.690  -0.026   3.661  37.241

Coefficients:
                 Estimate Std. Error  t value Pr(>|t|)
(Intercept)      0.230468   0.148604    1.551    0.121
NetRebounds      0.660276   0.002011  328.384   <2e-16 ***
TurnoverMargin  -1.315355   0.003303 -398.287   <2e-16 ***
TeamFGPCT       91.472656   0.241323  379.047   <2e-16 ***
OpponentFGPCT  -91.705986   0.241322 -380.016   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 5.493 on 110060 degrees of freedom
  (5 observations deleted due to missingness)
Multiple R-squared:  0.8923,    Adjusted R-squared:  0.8923
F-statistic: 2.279e+05 on 4 and 110060 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 Buzz Williams, and you want to win conference titles. To do that, we need to know what attributes of a team we should 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 since the 2017 season:

```
logs |>
  filter(Team == "Michigan" & Season == '2020-2021' | Team == "Wisconsin" & Season == '2019-2
  summarise(
    meanNetRebounds = mean(NetRebounds),
    meanTurnoverMargin = mean(TurnoverMargin),
    meanTeamFGPCT = mean(TeamFGPCT),
    meanOpponentFGPCT = mean(OpponentFGPCT)
  )
```

```
# A tibble: 1 x 4
  meanNetRebounds meanTurnoverMargin meanTeamFGPCT meanOpponentFGPCT
            <dbl>              <dbl>         <dbl>             <dbl>
1            9.43               1.71         0.475             0.400
```

Now it's just plug and chug.

```
# (netrebounds estimate * meanNetRebounds) + (turnover margin estimate * meanTurnoverMargin)
(0.660276*9.432099) + (-1.315355*1.711934) + (91.472656*0.4745391) + (-91.705986*0.4000823) +
```

```
[1] 10.92387
```

So a team with those numbers is going to average scoring nearly 11 more points per game than their opponent. Not a ton, but hey, the Big Ten has been a competitive conference lately.

How does that compare to Maryland in 2024-25 season?

```
logs |>
  filter(
    Team == "Maryland" & Season == '2024-2025'
    ) |>
  summarise(
```

```r
    meanNetRebounds = mean(NetRebounds),
    meanTurnoverMargin = mean(TurnoverMargin),
    meanTeamFGPCT = mean(TeamFGPCT),
    meanOpponentFGPCT = mean(OpponentFGPCT)
  )
```

```
# A tibble: 1 x 4
  meanNetRebounds meanTurnoverMargin meanTeamFGPCT meanOpponentFGPCT
            <dbl>              <dbl>         <dbl>             <dbl>
1            2.47              -3.44         0.469             0.415
```

```r
(0.660276*2.472222) + (-1.315355*-3.444444) + (91.472656*0.4687778) + (-91.705986*0.4150278)
```

```
[1] 11.2133
```

By this model, it predicted UMD would, on average, outscore its opponents by almost the same margin over that season. The reality?

```r
logs |>
     filter(
         Team == "Maryland" & Season == '2024-2025'
     ) |> summarise(avg_score = mean(TeamScore), avg_opp = mean(OpponentScore))
```

```
# A tibble: 1 x 2
  avg_score avg_opp
      <dbl>   <dbl>
1      81.1    67.2
```

We outscored them by nearly 14 points on average, which suggests that perhaps Maryland found a way to be even more successful outside of these parameters. What would you change?

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's women soccer should have outscored George Mason by a goal a year ago. The match was a 3-2 loss. So our residual is -2.

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 under-performing and over-performing the model. And the more robust the model – the better your r-squared value is – the more meaningful that label of under or over-performing is.

Let's go back to our model for men's college basketball. For our predictor, let's use Net FG Percentage - the difference between the two teams' shooting success.

Then load the tidyverse.

```
library(tidyverse)
```

```
logs <- read_csv("data/cbblogs1525.csv")
```

```
Rows: 110070 Columns: 60
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (10): Season, TeamFull, Opponent, HomeAway, W_L, URL, Conference, Team,...
dbl  (49): 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 make the columns we'll need.

```r
residualmodel <- logs |> mutate(differential = TeamScore - OpponentScore, FGPctMargin = TeamF
```

Now let's create our model.

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

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

Residuals:
    Min      1Q  Median      3Q     Max 
-51.053  -6.278  -0.221   5.928  68.482 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)    
(Intercept)   0.55356    0.02885   19.19   <2e-16 ***
FGPctMargin 123.43359    0.25881  476.93   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9.556 on 110063 degrees of freedom
  (5 observations deleted due to missingness)
Multiple R-squared:  0.6739,	Adjusted R-squared:  0.6739 
F-statistic: 2.275e+05 on 1 and 110063 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 under-performed the model by 51 points (!), and a team has overperformed it by 68 points (!!). The median residual, where half are above and half are below, is just slightly below the fit line. Close here is good.

Next: Look at the Adjusted R-squared value. What that says is that 67 percent of a team's scoring differential can be predicted by their FG percentage margin.

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 basketball, it doesn't surprise you that the more you shoot better than 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(fit
```

```
Error in `mutate()`:
i In argument: `predicted = predict(fit)`.
Caused by error:
! `predicted` must be size 110070 or 1, not 110065.
```

Uh, oh. What's going on here? When you get a message like this, where R is complaining about the size of the data, it most likely means that your model is using some columns that have NA values. In this case, the number of columns looks small - perhaps 3 - so let's just get rid of those rows by using the calculated columns from our model:

```
residualmodel <- residualmodel |> filter(!is.na(FGPctMargin))
```

Now we can try re-running the code to add the predicted and residuals columns:

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

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 and limit our results to games involving Maryland.

```
residualmodel |> filter(Team == 'Maryland') |> arrange(desc(residuals)) |> select(Date, Team
```

```
# A tibble: 301 x 8
   Date       Team   Opponent W_L   differential FGPctMargin predicted residuals
   <date>     <chr>  <chr>    <chr>        <dbl>       <dbl>     <dbl>     <dbl>
 1 2024-11-19 Maryl~ Canisius W               71      0.287      36.0       35.0
 2 2016-11-17 Maryl~ St. Mar~ W               48      0.115      14.7       33.3
 3 2024-11-08 Maryl~ Mount S~ W               34      0.0260      3.76      30.2
 4 2023-01-28 Maryl~ Nebraska W               19     -0.087     -10.2       29.2
 5 2023-12-12 Maryl~ Alcorn ~ W               40      0.085      11.0       29.0
 6 2024-12-17 Maryl~ Saint F~ W               54      0.211      26.6       27.4
 7 2022-11-25 Maryl~ Coppin ~ W               16     -0.0640     -7.35      23.3
 8 2018-12-11 Maryl~ Loyola ~ W               23      0.0100      1.79      21.2
 9 2024-12-21 Maryl~ Syracuse W               27      0.048       6.48      20.5
10 2024-01-27 Maryl~ Nebraska W               22      0.0110      1.91      20.1
# i 291 more rows
```

So looking at this table, what you see here are the teams who scored more than their FG percentage margin would indicate. One of the predicted values should jump off the page at you.

Look at that Maryland-Nebraska game from 2023. The Huskers shot better than the Terps, and the model predicted Nebraska would win by 10 points. Instead, Maryland **won by 19**!

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 Fouls

Now let's look at it where it doesn't work as well: the total number of fouls

```
fouls <- logs |>
  mutate(
    differential = TeamScore - OpponentScore,
    TotalFouls = TeamPersonalFouls+OpponentPersonalFouls
  )
```

```
pfit <- lm(differential ~ TotalFouls, data = fouls)
summary(pfit)
```

```
Call:
lm(formula = differential ~ TotalFouls, data = fouls)

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

```
-81.081 -10.645  -0.129   9.596 106.596

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  4.143416   0.248854   16.65   <2e-16 ***
TotalFouls  -0.080580   0.006875  -11.72   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 16.72 on 110063 degrees of freedom
  (5 observations deleted due to missingness)
Multiple R-squared:  0.001246,  Adjusted R-squared:  0.001237
F-statistic: 137.4 on 1 and 110063 DF,  p-value: < 2.2e-16
```

So from top to bottom:

- Our min and max go from -81 to positive 107
- Our adjusted R-squared is … 0.001237. Not much at all.
- Our p-value is … is less than than .05, so that's something.

So what we can say about this model is that it's statistically significant, but doesn't really explain much. It's not meaningless, but on its own the total number of fouls doesn't go very far in explaining the point differential. Normally, we'd stop right here – why bother going forward with a predictive model that isn't terribly predictive? But let's do it anyway. Oh, and see that "(5 observations deleted due to missingness)" bit? That means we need to lose some incomplete data again.

```r
fouls <- fouls |> filter(!is.na(TotalFouls))
fouls$predicted <- predict(pfit)
fouls$residuals <- residuals(pfit)
```

```r
fouls |> arrange(desc(residuals)) |> select(Team, Opponent, W_L, TeamScore, OpponentScore, To
```

```
# A tibble: 109,029 x 7
   Team              Opponent W_L   TeamScore OpponentScore TotalFouls residuals
   <chr>             <chr>    <chr>     <dbl>         <dbl>      <dbl>     <dbl>
 1 Bryant            Thomas ~ W           147            39         34     107.
 2 McNeese State     Dallas ~ W           140            37         33     102.
 3 Appalachian State Toccoa ~ W           135            34         35      99.7
 4 James Madison     Carlow   W           135            40         29      93.2
 5 Lamar             Howard ~ W           121            32         35      87.7
 6 Georgia Southern  Carver ~ W           139            51         38      86.9
```

```
 7 Youngstown State  Francis~ W           134            46          35      86.7
 8 North Carolina C~ St. And~ W           127            40          32      85.4
 9 Sam Houston State Southwe~ W           120            33          22      84.6
10 Tennessee-Martin  Champio~ W           115            29          28      84.1
# i 109,019 more rows
```

First, note all of the biggest misses here are all blowout games. The worst games of the season, the worst being Bryant vs. Thomas. The model missed that differential by … 107 points. The margin of victory? 108 points. In other words, this model is not great! 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'
```

100
50
0
−50
differential
20
40
60
80
TotalFouls

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 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/wbblogs24.csv")
```

```
Rows: 11425 Columns: 60
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (10): Season, GameType, TeamFullName, Opponent, HomeAway, W_L, OT, URL,...
dbl  (49): 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.
```

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, TeamFGPCT, TeamTotalRebounds, OpponentFGPCT, OpponentTotalRebounds
```

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

```
teamtotals <- teamquality |>
  group_by(Conference, Team) |>
  summarise(
    FGAvg = mean(TeamFGPCT),
    ReboundAvg = mean(TeamTotalRebounds),
    OppFGAvg = mean(OpponentFGPCT),
    OffRebAvg = mean(OpponentTotalRebounds)
    )
```

```
`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 360 basketball teams with Z scores. What does it look like?

```r
head(teamzscore)
```

```
# A tibble: 6 x 11
# Groups:   Conference [1]
  Conference Team         FGAvg ReboundAvg OppFGAvg OffRebAvg FGzscore RebZscore
  <chr>      <chr>        <dbl>      <dbl>    <dbl>     <dbl>    <dbl>     <dbl>
1 A-10 WBB   Davidson     0.436       31.3    0.378      27.1    0.878   -0.0939
2 A-10 WBB   Dayton       0.400       35.7    0.437      28.7   -0.225    1.29
3 A-10 WBB   Duquesne     0.424       33.8    0.390      32.4    0.517    0.687
4 A-10 WBB   Fordham      0.403       32.5    0.408      30.5   -0.136    0.286
5 A-10 WBB   George Mason 0.397       34.9    0.381      32.7   -0.335    1.03
6 A-10 WBB   George Wash~ 0.370       33.7    0.390      29.7   -1.17     0.680
# i 3 more variables: OppZscore <dbl>, OppRebZScore <dbl>, TotalZscore <dbl>
```

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: 359 x 11
# Groups:   Conference [33]
   Conference   Team      FGAvg ReboundAvg OppFGAvg OffRebAvg FGzscore RebZscore
   <chr>        <chr>     <dbl>      <dbl>    <dbl>     <dbl>    <dbl>     <dbl>
 1 SEC WBB      South Ca~ 0.495       42.2    0.324      28.3    2.23      1.79
 2 MAAC WBB     Fairfield 0.464       33.1    0.361      27.4    2.26      1.26
 3 Summit WBB   South Da~ 0.476       35.1    0.370      26.5    2.05      0.993
 4 Sun Belt WBB James Ma~ 0.424       42.4    0.363      31.0    1.18      1.60
 5 WCC WBB      Gonzaga   0.482       34.2    0.403      25.4    1.82      1.05
 6 SWAC WBB     Jackson ~ 0.397       38.5    0.341      29.9    0.815     1.54
 7 Big East WBB Connecti~ 0.498       33.5    0.356      27.8    2.26      0.848
 8 OVC WBB      Southern~ 0.452       35.6    0.359      31.1    2.02      1.13
 9 MWC WBB      Nevada-L~ 0.463       37.6    0.381      28.2    1.69      1.84
10 Big 12 WBB   Texas     0.490       36.2    0.384      24.4    1.66      0.793
# i 349 more rows
# i 3 more variables: OppZscore <dbl>, OppRebZScore <dbl>, TotalZscore <dbl>
```

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

But closer to home, how is Maryland doing?

```
teamzscore |>
  filter(Conference == "Big Ten WBB") |>
  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 WBB Iowa                 4.41
 2 Big Ten WBB Indiana              3.84
 3 Big Ten WBB Nebraska             3.15
 4 Big Ten WBB Penn State           1.88
```

```
 5 Big Ten WBB Illinois             1.20
 6 Big Ten WBB Michigan             0.396
 7 Big Ten WBB Maryland            -0.651
 8 Big Ten WBB Ohio State          -0.662
 9 Big Ten WBB Minnesota           -0.856
10 Big Ten WBB Wisconsin           -0.905
11 Big Ten WBB Michigan State      -1.17
12 Big Ten WBB Rutgers             -1.91
13 Big Ten WBB Purdue              -2.54
14 Big Ten WBB Northwestern        -6.20
```

So, as we can see, with our composite Z Score, Maryland is below average; not great. But better than Ohio State. Notice how, by this measure, Indiana and Iowa are far ahead of most of the conference, with Nebraska a somewhat surprising third.

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

```r
powerfive_plus_one <- c("SEC WBB", "Big Ten WBB", "Pac-12 WBB", "Big 12 WBB", "ACC WBB", "Bi
teamzscore |>
  filter(Conference %in% powerfive_plus_one) |>
  arrange(desc(TotalZscore)) |>
  select(Team, TotalZscore)
```

```
Adding missing grouping variables: `Conference`
```

```
# A tibble: 80 x 3
# Groups:   Conference [6]
   Conference   Team            TotalZscore
   <chr>        <chr>                 <dbl>
 1 SEC WBB      South Carolina         7.55
 2 Big East WBB Connecticut            5.79
 3 Big 12 WBB   Texas                  5.55
 4 Pac-12 WBB   Stanford               5.12
 5 SEC WBB      Louisiana State        5.09
 6 Pac-12 WBB   UCLA                   4.73
 7 ACC WBB      Virginia Tech          4.53
 8 Big Ten WBB  Iowa                   4.41
 9 Big Ten WBB  Indiana                3.84
10 ACC WBB      Duke                   3.65
# i 70 more rows
```

This makes a certain amount of sense: three of the Final Four teams - South Carolina, UConn and Iowa are in the top 10. N.C. State, the fourth team, ranks 16th. Virginia Tech is an interesting #7 here. It doesn't necessarily mean they were the seventh-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 are 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 Derik Queen, who played one season at Maryland before decamping for the NBA. Had he stayed, he might have been among the all-time Terp greats. So who does Derik Queen 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 last season.

Now load that data.

```r
players <- read_csv("data/players25.csv")
```

```
Rows: 5818 Columns: 57
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (10): Team, Player, Class, Height, Hometown, High School, Summary, Pos, ...
dbl (47): #, Weight, G, GS, MP, FG, FGA, FG%, 3P, 3PA, 3P%, 2P, 2PA, 2P%, eF...

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, Derik Queen is listed as a center, so let's just look at centers. Third, we want to limit the data to things that make sense to look at for Queen – things like shooting, rebounds, blocks, turnovers and points.

```r
playersselected <- players |>
  filter(MP>0) |> filter(Pos == "C") |>
  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 56, 26, 31, 28, 37, 8, 6, 66

Cluster means:
          MP         FG%        TRB        BLK        TOV        PTS
1 -0.1463575  0.06087454 -0.2258969 -0.2232569 -0.1338834 -0.3110702
2  1.2340831  0.56874622  1.3141886  2.1358712  0.7182216  0.8191132
3  0.6496780  0.13395291  0.4474867  0.1569653  0.5771021  0.4205940
4  1.4049575  0.25956500  1.4393448  0.4696840  1.4729983  1.5331265
5 -0.8886908  1.08843491 -0.8211314 -0.6281013 -0.8224037 -0.7114553
6  2.0601633 -0.09150173  2.1302168  1.6202760  2.7011350  3.0675147
7 -1.2088294 -3.91383033 -1.0828274 -0.8552925 -1.0785801 -0.9524554
8 -0.9047839 -0.69202775 -0.8462928 -0.6914856 -0.8336223 -0.7930986

Clustering vector:
  [1] 2 3 1 8 4 1 1 2 1 1 1 8 3 3 1 5 4 8 8 1 3 3 8 6 8 8 8 8 3 5 4 2 2 8 1 2 1
 [38] 6 5 1 7 1 1 5 3 8 3 1 8 2 5 1 5 5 8 1 1 4 5 8 1 5 5 4 1 1 8 1 5 8 1 1 1 1
 [75] 1 4 5 1 1 3 4 3 5 6 5 8 3 1 3 2 2 8 5 5 1 5 2 1 5 8 8 8 1 3 4 6 3 8 8 8 2
[112] 8 8 8 8 3 1 8 2 5 4 1 1 2 4 8 4 8 1 8 8 7 5 8 3 3 8 3 1 1 4 3 8 1 8 3 8 8
[149] 4 3 3 3 4 5 2 6 8 7 1 2 8 5 1 8 2 1 2 1 3 4 6 4 2 4 8 8 5 4 5 5 5 4 2 5 4
[186] 2 8 8 1 8 5 8 8 3 8 1 1 8 2 1 4 8 4 5 7 8 8 2 1 5 5 8 8 8 2 5 7 1 1 5 6 5
[223] 5 3 4 4 8 7 2 8 5 8 4 3 4 3 8 4 1 1 8 6 2 8 1 4 1 8 2 8 3 1 5 3 8 2 3 1

Within cluster sum of squares by cluster:
[1] 47.05815771 54.36226600 31.47483028 33.25682799 24.38259758 21.56104504
[7]  0.03684809 31.99020794
 (between_SS / total_SS =  84.2 %)

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 2 and 8 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 5 is above average on minutes played. Groups 2 is slightly below, group 1 is well above.

So which group is Derik Queen 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 Derik Queen 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 Derik Queen and then plot him against the rest of college basketball on rebounds versus minutes played.

```
dq <- playercluster |> filter(Player == "Derik Queen")

dq
```

```
       Player                      Team Pos   MP   FG. TRB BLK TOV PTS
1 Derik Queen Maryland Terrapins Men's   C 1093 0.526 323  39  85 594
  k8.cluster
1          6
```

So Derik's in cluster 1, which if you look at our clusters, puts him in the cluster with nearly all above average metrics. What does that look like? We know Derik was a rebounding machine, so where do group 1 people grade out on rebounds?

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

Not bad, not bad. But who are Derik Queen's peers? If we look at the numbers in Group 1, there's 8 of them.

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

playercluster |> filter(k8.cluster == 6) |> arrange(desc(MP))
```

```
            Player                         Team Pos   MP   FG. TRB BLK TOV PTS
1 Ryan Kalkbrenner     Creighton Bluejays Men's   C 1206 0.653 303  93  51 672
2      Derik Queen     Maryland Terrapins Men's   C 1093 0.526 323  39  85 594
3     Robbie Avila  Saint Louis Billikens Men's   C 1053 0.492 215  24 101 535
4    Bent Leuchten    UC-Irvine Anteaters Men's   C 1045 0.544 354  50  60 599
5 Hunter Dickinson        Kansas Jayhawks Men's   C 1025 0.526 340  44  74 592
6    Nathan Bittle           Oregon Ducks Men's   C  971 0.511 265  75  56 496
7  Noah Williamson         Bucknell Bison Men's   C  945 0.538 244  35  91 562
8   Carter Welling Utah Valley Wolverines Men's   C  784 0.478 201  57  69 420
  k8.cluster
1          6
2          6
3          6
4          6
5          6
6          6
```

```
7           6
8           6
```

So here are the 7 centers most like Derik Queen last season. Were they the best centers in the country?

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 Maryland'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 == "C") |>
  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)
```

```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 8 again.

```r
advk8 <- kmeans(playersadvscaled, centers = 8, nstart = 25)
```

What do we have here?

```r
advk8
```

```
K-means clustering with 8 clusters of sizes 10, 54, 2, 25, 56, 63, 31, 17

Cluster means:
          PER         TS%      PProd       AST%        WS/40         BPM
1 -2.80397873 -3.32576860 -0.9757080 -1.1460236 -2.779002133 -2.44465692
2  0.58549934  0.46575451  0.9259213 -0.1446943  0.564620860  0.56788941
3  3.91910490  2.61177952 -0.9430676 -1.3207832  3.141851512  3.45044891
4  1.03333914  0.26289465  2.0174109  1.7580590  0.882444035  1.09048724
```

```
5 -0.80660282 -0.61517254 -0.7087302 -0.3516492 -0.770575283 -0.81273103
6 -0.06088192  0.22607412 -0.4488703 -0.5876646 -0.008179258  0.02469389
7 -0.14080517 -0.02569281 -0.3445817  0.9967488 -0.156052595 -0.28053836
8  0.94831844  1.01849923 -0.5965913  0.2223432  1.027107675  0.72184555

Clustering vector:
  [1] 2 2 6 1 4 6 6 2 5 6 6 5 2 7 6 5 2 5 5 6 6 2 6 4 5 5 7 6 4 7 2 4 2 5 7 2 6
 [38] 4 6 6 1 6 7 7 2 5 2 6 6 2 6 6 3 8 1 7 5 1 2 6 6 7 3 8 2 6 8 5 6 7 6 2 8 7
 [75] 5 6 4 6 5 6 2 4 2 8 4 6 5 2 6 4 2 2 7 8 8 7 6 2 7 7 5 5 6 6 6 2 4 5 5 1 7
[112] 7 7 6 6 5 2 6 5 2 6 4 6 5 6 2 5 4 7 6 5 5 1 5 5 7 6 5 5 6 6 2 6 5 5 6 2 6
[149] 5 4 2 2 7 2 8 2 4 8 1 6 4 5 5 7 5 2 7 4 7 4 2 4 2 2 2 5 5 2 6 8 6 4 2 5 4
[186] 6 5 5 5 7 8 5 6 2 5 5 7 7 2 6 2 5 2 8 1 7 5 2 6 6 5 5 6 6 2 8 1 6 6 8 4 8
[223] 6 2 4 2 6 1 2 5 8 7 2 2 2 7 5 2 7 5 6 4 2 5 6 4 6 5 4 8 2 6 5 5 5 2 7 5

Within cluster sum of squares by cluster:
[1] 61.68774 54.61697  3.33234 59.95049 83.85300 64.48091 53.46295 43.85311
 (between_SS / total_SS =  72.4 %)

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 4 is all above. Which cluster is Derik Queen in?

```r
playeradvcluster <- data.frame(playersadvanced, advk8$cluster)
```

```r
dqadv <- playeradvcluster |> filter(Player == "Derik Queen")

dqadv
```

```
       Player                      Team Pos  PER   TS. PProd AST. WS.40 BPM
1 Derik Queen Maryland Terrapins Men's   C 24.7 0.591   584 11.6 0.216 9.2
  advk8.cluster
1             4
```

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

Now who are his peers?

```r
playeradvcluster |>
  filter(advk8.cluster == 4) |>
  arrange(desc(PProd))
```

```
               Player                                 Team Pos  PER   TS. PProd
1    Ryan Kalkbrenner            Creighton Bluejays Men's   C 29.1 0.684   589
2         Derik Queen            Maryland Terrapins Men's   C 24.7 0.591   584
3       Bent Leuchten           UC-Irvine Anteaters Men's   C 28.7 0.648   582
4    Hunter Dickinson               Kansas Jayhawks Men's   C 25.8 0.571   558
5        Robbie Avila         Saint Louis Billikens Men's   C 21.0 0.605   542
6     Noah Williamson                Bucknell Bison Men's   C 23.5 0.594   515
7       Nathan Bittle                  Oregon Ducks Men's   C 25.5 0.598   476
8       Essam Mostafa Middle Tennessee Blue Raiders Men's   C 29.0 0.621   468
9      Kyler Filewich              Wofford Terriers Men's   C 24.6 0.529   452
10        Oumar Ballo              Indiana Hoosiers Men's   C 23.7 0.618   404
11      Payton Sparks          Ball State Cardinals Men's   C 26.1 0.618   404
12     Mitchell Saxen            Saint Mary's Gaels Men's   C 22.3 0.547   400
13    Tomislav Ivisic      Illinois Fighting Illini Men's   C 22.0 0.597   400
14     Carter Welling        Utah Valley Wolverines Men's   C 21.0 0.560   388
15      Jeff Woodward               Colgate Raiders Men's   C 27.9 0.655   376
16    Amar Kuljuhovic   North Dakota Fighting Hawks Men's   C 19.4 0.572   373
17       Jacob Hutson        Northern Iowa Panthers Men's   C 19.5 0.557   370
18    Christoph Tilly           Santa Clara Broncos Men's   C 23.5 0.623   367
19 Justin Vander Baan            Lafayette Leopards Men's   C 23.0 0.547   364
20        Miles Rubin          Loyola (IL) Ramblers Men's   C 22.1 0.683   322
21       Michael Walz              Richmond Spiders Men's   C 19.5 0.587   269
22    Lawson Lovering                     Utah Utes Men's   C 15.5 0.505   262
23     Dylan Cardwell                 Auburn Tigers Men's   C 20.3 0.627   240
24   Brandon Garrison             Kentucky Wildcats Men's   C 16.6 0.547   231
25          Aday Mara                   UCLA Bruins Men's   C 28.8 0.594   216
   AST. WS.40  BPM advk8.cluster
1  10.0 0.236 12.2             4
2  11.6 0.216  9.2             4
3  12.5 0.277  9.0             4
4  14.4 0.197 10.3             4
5  24.1 0.158  4.7             4
6  12.7 0.154  2.2             4
7  14.4 0.203 11.4             4
8   9.7 0.237  6.3             4
9  24.1 0.168  3.5             4
10 14.7 0.174  8.8             4
11 13.6 0.197  2.2             4
```

```
12 11.2 0.196  8.5            4
13 15.8 0.187 10.7            4
14 12.1 0.160  2.9            4
15 22.6 0.209  3.0            4
16 14.0 0.111  0.2            4
17 14.9 0.148  1.8            4
18 16.0 0.193  6.7            4
19 24.0 0.157  3.8            4
20 11.7 0.167  5.1            4
21 27.3 0.135  3.2            4
22 22.0 0.097  3.6            4
23 12.7 0.189 11.4            4
24 18.0 0.116  6.3            4
25 16.7 0.222 11.4            4
```

Sorting on Points Produced, Derik Queen is second out of the 25 centers who land in Cluster 4. Seems advanced metrics rate him pretty highly.

14 Simulations

On Feb. 21, 2023, fans of Maryland women's basketball got a show from Brinae Alexander. The graduate transfer guard hit 6 of 9 three-pointers in a 96-68 destruction of Iowa (Caitlin Clark scored 18, her second-lowest output of the season). It was glorious.

But how *rare* was it? Did Alexander get lucky that night?

Luck is something that comes up a lot in sports. Is a team lucky? 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 10 three point attempts (0-9 makes) 1000 times with Alexander's 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 Brinae Alexander is going to shoot above and below her season three point shooting percentage. If we randomly assign values in that distribution 1000 times, how many times will it come up 6, like this example?

First, we'll load the tidyverse

```r
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.2
v ggplot2   3.5.2     v tibble    3.3.0
v lubridate 1.9.4     v tidyr     1.3.1
v purrr     1.0.4
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becom
```

```r
set.seed(1234)

simulations <- rbinom(n = 1000, size = 10, prob = .439)

table(simulations)
```

```
simulations
  0   1   2   3   4   5   6   7   8   9
  5  21  82 177 235 234 130  82  30   4
```

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 her season-long shooting percentage, it's not out of the realm of randomness that she'd make 6 of those 9 attempts. In 1000 simulations, it comes up 130 times. So more than one time in 10, Brinae Alexander will go 6-9 from deep. While it's more likely that she'd hit 4 or 5 of those attempts, a one-in-ten chance isn't nothing.

14.1 Cold streaks

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

```
set.seed(1234)

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

hist(simulations)
```

```
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 17 times it will result in a single made three-pointer. It's slightly more common that the team would hit 9 threes out of 15. So going that cold is not totally out of the realm of random chance, but it's pretty rare.

14.2 The hottest of streaks

Two years ago, Terps baseball player Sam Hojnar hit two home runs in consecutive games. He hit 16 HRs for the season in 233 at-bats, so his home run probability per at-bat was just under seven percent. We'll use that and the number of games (56) to calculate the odds that he'd hit two home runs in each of two consecutive games.

```
# Hojnar's statistics
home_runs <- 16
games <- 56
at_bats <- 233

# Calculate probabilities
home_run_prob_per_at_bat <- home_runs / at_bats
avg_at_bats_per_game <- at_bats / games

# Set simulation parameters
num_simulations <- 100000

# Run simulation
set.seed(1234)  # For reproducibility

simulation_results <- tibble(
  sim = 1:num_simulations,
  game1 = rbinom(num_simulations, round(avg_at_bats_per_game), home_run_prob_per_at_bat),
  game2 = rbinom(num_simulations, round(avg_at_bats_per_game), home_run_prob_per_at_bat)
) |>
  mutate(two_hr_each_game = game1 == 2 & game2 == 2)

# Calculate probability
probability <- simulation_results |>
  summarise(prob = mean(two_hr_each_game)) |>
  pull(prob)

# Print results
cat("Estimated probability of hitting exactly two home runs in each of two consecutive games
```

```
Estimated probability of hitting exactly two home runs in each of two consecutive games: 0.0
```

Let's parse that code. We set up our simulations as usual, with one change: because this already seems like a pretty rare event, we're running 100,000 simulations, and we have to

calculate the odds for two games, not one. Then we're looking for results where the number of home runs in both games is two. The `simulation_results` dataframe shows that 70 times out of 100,000 Hojnar would hit two home runs in consecutive games, based on his own performance. That's very, very, very unlikely, and quite a hot streak.

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: 146 Columns: 14
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (2): Institution, Conference
dbl (12): 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, ...

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.

```r
attendance |>
  arrange(desc(`2024`)) |>
  top_n(10) |>
  select(Institution, `2024`)
```

```
Selecting by 2024

# A tibble: 10 x 2
   Institution `2024`
   <chr>        <dbl>
 1 Ohio St.    936550
 2 Michigan    884382
 3 Penn St.    864665
 4 Texas       817852
 5 Texas A&M   719927
 6 Tennessee   713405
 7 LSU         708645
 8 Auburn      704344
 9 Alabama     700539
10 Florida     630116
```

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

```r
top10 <- attendance |>
  arrange(desc(`2024`)) |>
  top_n(10) |>
  select(Institution, `2024`)
```

```
Selecting by 2024
```

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=`2024`)
  )
```

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, `2024`),
```

```
      weight=`2024`
      )
    )
```

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, -`2024`),
      weight=`2024`
      )
    )
```

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, -`2024`),
      weight=`2024`
      )
    ) +
  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, -`2024`),
      weight=`2024`)
    ) +
  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, -`2024`),
        weight=`2024`)) +
  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, -`2024`),
        weight=`2024`)) +
  scale_y_continuous(labels=comma) +
  labs(
    title="Top 10 Football Programs By Attendance",
```

```
    x="School",
    y="Attendance") +
  theme_minimal()
```

If you like, you can write your 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, -`2024`),
        weight=`2024`)) +
  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 basketball games from this past season.

Load the tidyverse.

```
library(tidyverse)
```

And the data.

```
games <- read_csv("data/logs25.csv")
```

```
Rows: 11962 Columns: 60
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (10): Season, GameType, TeamFullName, Opponent, HomeAway, W_L, OT, URL,...
dbl  (49): 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.
```

What we have here is every game in college basketball this past season. The question we want to answer is this: Who were the best rebounders in the Big Ten? And what role did offensive and defensive rebounds play in making that happen?

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 rebounds and total rebounds, and then we can calculate defensive rebounds. So if we have all the games a team played, and the offensive rebounds and total rebounds for each of those games, what we need to do to get the season totals is just add them up.

```
games |>
  filter(!is.na(TeamTotalRebounds)) |>
  group_by(Conference, Team) |>
  summarise(
    SeasonOffRebounds = sum(TeamOffRebounds),
    SeasonTotalRebounds = sum(TeamTotalRebounds)
  ) |>
  mutate(
    SeasonDefRebounds = SeasonTotalRebounds - SeasonOffRebounds
  ) |>
  select(
    -SeasonTotalRebounds
  ) |>
  filter(Conference == "Big Ten MBB")
```

```
# A tibble: 18 x 4
# Groups:   Conference [1]
   Conference  Team                SeasonOffRebounds SeasonDefRebounds
   <chr>       <chr>                           <dbl>             <dbl>
 1 Big Ten MBB Illinois                          408               982
 2 Big Ten MBB Indiana                           292               754
 3 Big Ten MBB Iowa                              250               731
 4 Big Ten MBB Maryland                          321               881
 5 Big Ten MBB Michigan                          346               950
 6 Big Ten MBB Michigan State                    386               972
 7 Big Ten MBB Minnesota                         271               702
 8 Big Ten MBB Nebraska                          270               876
 9 Big Ten MBB Northwestern                      302               709
10 Big Ten MBB Ohio State                        252               716
11 Big Ten MBB Oregon                            285               829
12 Big Ten MBB Penn State                        242               711
13 Big Ten MBB Purdue                            302               776
14 Big Ten MBB Rutgers                           304               720
15 Big Ten MBB Southern California               262               759
16 Big Ten MBB UCLA                              325               692
17 Big Ten MBB Washington                        257               672
18 Big Ten MBB Wisconsin                         279               945
```

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

```
rebounds <- games |>
  filter(!is.na(TeamTotalRebounds)) |>
  group_by(Conference, Team) |>
  summarise(
    SeasonOffRebounds = sum(TeamOffRebounds),
    SeasonTotalRebounds = sum(TeamTotalRebounds)
  ) |>
  mutate(
    SeasonDefRebounds = SeasonTotalRebounds - SeasonOffRebounds
  ) |>
  select(
    -SeasonTotalRebounds
  ) |>
  filter(Conference == "Big Ten MBB")
```

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.

```
rebounds |>
  pivot_longer(
    cols=starts_with("Season"),
    names_to="Type",
    values_to="Rebounds")
```

```
# A tibble: 36 x 4
# Groups:   Conference [1]
   Conference  Team     Type               Rebounds
   <chr>       <chr>    <chr>                 <dbl>
 1 Big Ten MBB Illinois SeasonOffRebounds       408
 2 Big Ten MBB Illinois SeasonDefRebounds       982
 3 Big Ten MBB Indiana  SeasonOffRebounds       292
 4 Big Ten MBB Indiana  SeasonDefRebounds       754
 5 Big Ten MBB Iowa     SeasonOffRebounds       250
 6 Big Ten MBB Iowa     SeasonDefRebounds       731
 7 Big Ten MBB Maryland SeasonOffRebounds       321
 8 Big Ten MBB Maryland SeasonDefRebounds       881
 9 Big Ten MBB Michigan SeasonOffRebounds       346
10 Big Ten MBB Michigan SeasonDefRebounds       950
# i 26 more rows
```

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

```
reboundslong <- rebounds |>
  pivot_longer(
    cols=starts_with("Season"),
    names_to="Type",
    values_to="Rebounds")
```

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 offensive rebounds on top of defensive rebounds and we can see how they compare.

```
ggplot() +
  geom_bar(
    data=reboundslong,
    aes(x=Team, weight=Rebounds, 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 Rebounds.

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

And just like that … Michigan State, the team with the best record in the league, comes out #2 behind Illinois. Maryland is fifth, which seems about right.

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/wbblogs24.csv")
```

```
Rows: 11425 Columns: 60
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (10): Season, GameType, TeamFullName, Opponent, HomeAway, W_L, OT, URL,...
dbl  (49): 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.
```

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.

Sports Reference's win columns are weird, so we need to scan through them and find W and L and we'll give them numbers using case_when. I'll also filter out post-season tournament games.

```
winlosslogs <- logs |>
  filter(Date < '2025-03-15') |>
  mutate(winloss = case_when(
    grepl("W", W_L) ~ 1,
    grepl("L", W_L) ~ 0)
)
```

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

```
dates <- winlosslogs |> group_by(Date, Conference) |> summarise(wins = sum(winloss))
```

```
`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 WBB", "Big Ten WBB", "Pac-12 WBB", "Big 12 WBB", "ACC WBB")
```

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 <- winlosslogs |> mutate(month = floor_date(Date, unit="months")) |> group_by(month, Co
```

```
`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 <- winlosslogs |> mutate(week = floor_date(Date, unit="weeks")) |> group_by(week, Confe
```

```
`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 like we did before.

```
oppwinlosslogs <- logs |>
  mutate(winloss = case_when(
    grepl("W", W_L) ~ 0,
    grepl("L", W_L) ~ 1)
) |>
  filter(Date < "2025-03-15")
```

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(winloss)) |> mutate(wi
```

Now we have a dataframe of 614 opponent winning records. Wait, what? There's like ~350 teams in major college basketball, so why 614? 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", W_L) ~ 1,
    grepl("L", W_L) ~ 0)
) |>
  filter(Date < "2025-03-15")
```

```
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 WBB", "Big Ten WBB", "Pac-12 WBB", "Big 12 WBB", "ACC WBB")
```

```
winswithopppct |> filter(winpct > .6) |> mutate(week = floor_date(Date, unit="weeks")) |> gro
```

```
`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_mini
```

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 game against Michigan State this season. 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. Net rushing vs passing.

```
md <- c("Rushing"=175, "Passing"=314)
ms <- c("Rushing"=100, "Passing"=221)
```

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"=314, "Michigan State"=221)
```

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

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

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(ms,
        rows = 10,
        title="Michigan State's offense",
        xlab="1 square = 1 yard",
        colors = c("black", "green")
        )
)
```

What do you notice about this chart? Notice how the squares aren't the same size? Well, Maryland out-gained Michigan State 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 489, Maryland's total yards of offense. So to do that, we need to add a padding of 168 to Michigan State. REMEMBER: Don't name it or it'll show up in the legend.

```
md <- c("Rushing"=175, "Passing"=314)
ms <- c("Rushing"=100, "Passing"=221, 168)
```

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(ms,
        rows = 10,
        title="Michigan State's offense",
```

```
        xlab="1 square = 1 yard",
        colors = c("black", "green", "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(ms/2,
        rows = 10,
        title="Michigan State's offense",
        xlab="1 square = 1 yard",
        colors = c("black", "green", "white")
        )
)
```

News flash: Michigan State is changing its fight song to “Everybody Hurts” by REM.

19 Line charts

So far, we've talked about bar charts – stacked or otherwise – are good for showing relative size of a thing compared to another thing. Stacked Bars and Waffle charts are good at showing proportions of a whole.

Line charts are good for showing change over time.

Let's look at how we can answer this question: How did Maryland men's basketball team

We'll need the logs of every game in college basketball for this.

Let's start getting all that we need. We can use the tidyverse shortcut.

```
library(tidyverse)
```

And now load the data.

```
logs <- read_csv("data/logs25.csv")
```

```
Rows: 11962 Columns: 60
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (10): Season, GameType, TeamFullName, Opponent, HomeAway, W_L, OT, URL,...
dbl  (49): 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.
```

This data has every game from every team in it, so we need to use filtering to limit it, because we just want to look at Maryland. If you don't remember, flip back to chapter 6.

```
umd <- logs |> filter(Team == "Maryland")
```

Because this data has just Maryland data in it, the dates are formatted correctly, and the data is long data (instead of wide), we have what we need to make line charts.

Line charts, unlike bar charts, do have a y-axis. So in our ggplot step, we have to define what our x and y axes are. In this case, the x axis is our Date – the most common x axis in line charts is going to be a date of some variety – and y in this case is up to us. Three-point shooting has been an increasingly important feature of basketball, so let's chart that.

```
ggplot() + geom_line(data=umd, aes(x=Date, y=Team3PPCT))
```

The problem here is that the Y axis doesn't start with zero. That makes this look more dramatic than it is. To make the axis what you want, you can use `scale_x_continuous` or `scale_y_continuous` and pass in a list with the bottom and top value you want. You do that like this:

```
ggplot() +
  geom_line(data=umd, aes(x=Date, y=Team3PPCT)) +
  scale_y_continuous(limits = c(0, .6))
```

Note also that our X axis labels are automated. It knows it's a date and it just labels it by month.

19.1 This is too simple.

With datasets, we want to invite comparison. So let's answer the question visually. Let's put two lines on the same chart. How does Maryland compare to conference leader Michigan State, for example?

```
msu <- logs |> filter(Team == "Michigan State")
```

In this case, because we have two different datasets, we're going to put everything in the geom instead of the ggplot step. We also have to explicitly state what dataset we're using by saying `data=` in the geom step.

First, let's chart Maryland. Read carefully. First we set the data. Then we set our aesthetic. Unlike bars, we need an X and a Y variable. In this case, our X is the date of the game, Y is the thing we want the lines to move with. In this case, the Team Field Goal Percentage – TeamFGPCT.

```
ggplot() + geom_line(data=umd, aes(x=Date, y=Team3PPCT), color="red")
```

Now, by using +, we can add Michigan State to it. REMEMBER COPY AND PASTE IS A THING. Nothing changes except what data you are using.

```
ggplot() +
  geom_line(data=umd, aes(x=Date, y=Team3PPCT), color="red") +
  geom_line(data=msu, aes(x=Date, y=Team3PPCT), color="green")
```

Let's flatten our lines out by zeroing the Y axis. We'll set the upper limit of the y-axis to 0.70 because Illinois shot fantastically well in one December game.

```
ggplot() +
  geom_line(data=umd, aes(x=Date, y=Team3PPCT), color="red") +
  geom_line(data=msu, aes(x=Date, y=Team3PPCT), color="green")
```

```
  scale_y_continuous(limits = c(0, .70))
```

```
<ScaleContinuousPosition>
 Range:
 Limits:    0 --  0.7
```

So visually speaking, the difference between Maryland and Michigan State's seasons isn't that great - they both had some variation in shooting from three, but mostly that smoothed out a bit after the end of 2024.

19.2 But what if I wanted to add a lot of lines.

Fine. How about all Power Five Schools? This data for example purposes. You don't have to do it.

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

p5conf <- logs |> filter(Conference %in% powerfive)
```

I can keep layering on layers all day if I want. And if my dataset has more than one team in it, I need to use the `group` command. And, the layering comes in order – so if you're going to layer a bunch of lines with a smaller group of lines, you want the bunch on the bottom. So to do that, your code stacks from the bottom. The first geom in the code gets rendered first. The second gets layered on top of that. The third gets layered on that and so on.

```
ggplot() +
  geom_line(data=p5conf, aes(x=Date, y=Team3PPCT, group=Team), color="grey") +
  geom_line(data=umd, aes(x=Date, y=Team3PPCT), color="red") +
  geom_line(data=msu, aes(x=Date, y=Team3PPCT), color="green")
```

```
  scale_y_continuous(limits = c(0, .65))
```

```
<ScaleContinuousPosition>
 Range:
 Limits:    0 -- 0.65
```

What do we see here? How have Maryland's and Michigan State's seasons evolved against all the rest of the teams in major college basketball?

But how does that compare to the average? We can add that pretty easily by creating a new dataframe with it and add another geom_line.

```r
average <- logs |> group_by(Date) |> summarise(mean_shooting=mean(Team3PPCT))
```

```r
ggplot() +
  geom_line(data=p5conf, aes(x=Date, y=Team3PPCT, group=Team), color="grey") +
  geom_line(data=umd, aes(x=Date, y=Team3PPCT), color="red") +
  geom_line(data=msu, aes(x=Date, y=Team3PPCT), color="green") +
  geom_line(data=average, aes(x=Date, y=mean_shooting), color="black") +
  scale_y_continuous(limits = c(0, .70))
```

20 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 Washington 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 Maryland's women basketball 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/wbblogs25.csv")
```

```
Rows: 11467 Columns: 60
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (10): Season, GameType, TeamFullName, Opponent, HomeAway, W_L, OT, URL,...
dbl  (49): 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(TeamFullName) |>
  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 WBB")
```

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 Maryland, plus another team: Illinois

```
umd <- bigdiff |> filter(Team == "Maryland")
ill <- 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.

```
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=ill, aes(x=Date, y=CumDiff, group=Team), color="orange") +
  annotate("text", x=(as.Date("2024-12-10")), y=300, label="Maryland") +
  annotate("text", x=(as.Date("2025-02-01")), y=175, label="Illinois") +
  labs(
    x="Date",
    y="Cumulative Point Differential",
    title="Maryland and Illinois Had Similar Seasons",
    subtitle="The Terps were middle of the pack in cumulative point differential.",
    caption="Source: Sports-Reference.com | By Derek Willis") +
  theme_minimal()
```

21 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)
```

21.1 Dumbbell plots

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

And load it.

```
logs <- read_csv("data/ncaa_womens_volleyball_matchstats_2024.csv")
```

```
Rows: 10149 Columns: 40
-- Column specification --------------------------------------------------------
Delimiter: ","
chr   (4): team, opponent, home_away, result
dbl  (35): ncaa_id, team_score, opponent_score, s.x, kills, errors, total_at...
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 errors versus their opponents' errors. To get this, we're going to add up all errors and opponent errors 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.

```
big10 <- c("Nebraska", "Iowa", "Minnesota", "Illinois", "Northwestern", "Wisconsin", "Indiana

errors <- logs |>
  filter(team %in% big10) |>
  group_by(team) |>
  summarise(
    total_errors = sum(errors),
    opp_errors = sum(defensive_errors))
```

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=errors,
    aes(y=team, x=total_errors, xend=opp_errors)
  )
```

```
Warning: Using the `size` aesthetic with geom_segment was deprecated in ggplot2 3.4.0.
i Please use the `linewidth` aesthetic instead.
```

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

So our choice of colors here is important. We want team errors to be seen as bad and opponent errors to be seen as good. So lets try red for team errors and green for opponent errors 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, defensive errors are x, they're good, so they're green.

```
ggplot() +
  geom_dumbbell(
    data=errors,
    aes(y=team, x=opp_errors, xend=total_errors),
    colour = "grey",
    colour_x = "green",
    colour_xend = "red")
```

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

```
ggplot() +
  geom_dumbbell(
    data=errors,
    aes(y=team, x=opp_errors, xend=total_errors),
    linewidth = 1,
    color = "grey",
    colour_x = "green",
    colour_xend = "red") +
  theme_minimal()
```

```
Warning in geom_dumbbell(data = errors, aes(y = team, x = opp_errors, xend =
total_errors), : Ignoring unknown parameters: `linewidth`
```

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 errors?

```
ggplot() +
  geom_dumbbell(
    data=errors,
    aes(y=reorder(team, opp_errors), x=opp_errors, xend=total_errors),
    linewidth = 1,
    color = "grey",
    colour_x = "green",
    colour_xend = "red") +
  theme_minimal()
```

```
Warning in geom_dumbbell(data = errors, aes(y = reorder(team, opp_errors), :
Ignoring unknown parameters: `linewidth`
```

Maryland's opponents have committed some errors - that's good news for the Terps - but there's a clear gap between Maryland and the top teams in the conference when it comes to committing errors. And the gap between "good" errors and "bad" errors for the Terps is tiny.

21.2 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=errors,
    aes(y=team, x=opp_errors),
    horizontal = TRUE
    )
```

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

```
ggplot() +
  geom_lollipop(
    data=errors,
    aes(y=reorder(team, opp_errors), x=opp_errors),
    horizontal = TRUE
    ) + theme_minimal() +
  labs(title = "Nebraska, Penn State force more errors", y="team")
```

How about some layering?

```
md <- errors |> filter(team == "Maryland")
```

```
ggplot() +
  geom_lollipop(
    data=errors,
    aes(y=reorder(team, opp_errors), x=opp_errors),
    horizontal = TRUE
    ) +
  geom_lollipop(
    data=md,
    aes(y=team, x=opp_errors),
    horizontal = TRUE,
    color = "red"
  ) +
  theme_minimal() +
  labs(title = "Maryland near the bottom of Big Ten teams", y="team")
```

The headline says it all.

22 Scatterplots

With the exception of those curmudgeons who love defense, everybody loves scoring. We enjoy blowout games (when our teams win) and bemoan low-scoring affairs. But does how you score matter? Does it make a difference if you rely on a single star to go one-on-one versus if you move the ball around? While scoring is a necessary condition for winning, it's not the only one. Plenty of bad teams score a lot - they just happen to give up more runs or points or goals.

So how do we tell if good passing makes a difference? How can we test the ingredients of success and determine what's a significant factor and what's a hot take?

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**.

To demonstrate this, we'll look at women's college lacrosse from the 2025 season, and we'll see how assists and wins are related.

First, we need libraries and every college lacrosse game from the last season. **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.

```
teams <- read_csv("data/wlax_team_stats_2025.csv")
```

```
Rows: 131 Columns: 37
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (5): school_name, conference_name, ncaa_division_formatted, sport_id, g...
dbl (32): fouls, goals, assists, points, shots, shots_on_goal, ground_balls,...

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 got columns for overall_wins and assists. 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 assists, y is our wins.

```
ggplot() + geom_point(data=teams, aes(x=assists, y=overall_wins))
```

Let's talk about this. Ok, there's really a clear pattern here - as assists increase, so do wins, generally. 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=teams, aes(x=assists, y=overall_wins)) +
  geom_smooth(data=teams, aes(x=assists, y=overall_wins), method="lm")
```

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

A line climbing from left to right is good. It means there's a solid positive relationship here. The numbers don't suggest anything. Still, it's worth asking: can we know exactly how strong of a relationship is this? How much can assists scored 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(overall_wins ~ assists, data = teams)
summary(fit)
```

```
Call:
lm(formula = overall_wins ~ assists, data = teams)

Residuals:
    Min      1Q  Median      3Q     Max 
-5.8395 -1.3759 -0.0456  1.4786  5.3421 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
```

```
(Intercept) -1.141803   0.561467  -2.034    0.044 *
assists      0.103026   0.005547  18.573   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.212 on 129 degrees of freedom
Multiple R-squared:  0.7278,    Adjusted R-squared:  0.7257
F-statistic:   345 on 1 and 129 DF,  p-value: < 2.2e-16
```

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 0.7257, which we can interpret as the number of assists a team records predicts about 72 percent of the variance in wins. Pretty good!

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 0.00000000000000022, so this isn't random chance. Which makes sense, because it's harder to win when you don't score.

Normally, we'd stop here, but let's look at the third element: The coefficient. In this case, the coefficient for assists is 0.103026. What this model predicts, given that and the intercept of -1.141803, is this: For every assist you tally, you add about .10 towards your wins total. So if you have 50 assists in a season, you'll be a 5-win team. Notch 150, you're closer to a 15-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 goals and b is the intercept.

Let's use Maryland as an example. They recorded 118 assists last season.

y = 0.103026 * 118 + -1.141803 or 11 wins

How many wins did Maryland have? 15.

What does that mean? It means that Maryland over-performed, according to this model. Seems like assists is an ok predictor for Maryland, but maybe not a great one. Perhaps the Terps relied on one-on-one scoring opportunities instead of passing. Where is Maryland on the plot? We know we can use layering for that.

```r
umd <- teams |> filter(school_name == "Maryland")
```

```r
ggplot() +
  geom_point(data=teams, aes(x=assists, y=overall_wins)) +
  geom_smooth(data=teams, aes(x=assists, y=overall_wins), method="lm") +
  geom_point(data=umd, aes(x=assists, y=overall_wins), color="red")
```

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

Maryland's not the most interesting team on this plot, though. Who is?

22.1 Let's see it fail

Scatterplots also are useful for shooting down the hottest of takes. What about green cards? Are they related to wins in lacrosse?

```
ggplot() +
  geom_point(data=teams, aes(x=green_cards, y=overall_wins)) +
  geom_smooth(data=teams, aes(x=green_cards, y=overall_wins), method="lm")
```

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

This is basically a flat line and a lot of dots spread all over the plot. There appears to be absolutely no relationship, but let's test that out just to be sure.

Let's get our linear regression stats.

```
gcfit <- lm(overall_wins ~ green_cards, data = teams)
summary(gcfit)
```

```
Call:
lm(formula = overall_wins ~ green_cards, data = teams)

Residuals:
    Min      1Q  Median      3Q     Max 
-8.6510 -2.6487 -0.6428  2.3527 13.3531 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)    
(Intercept) 8.6319063  1.1930828   7.235 3.69e-11 ***
green_cards 0.0004536  0.0303506   0.015    0.988    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 4.241 on 129 degrees of freedom
Multiple R-squared:  1.731e-06, Adjusted R-squared:  -0.00775
F-statistic: 0.0002233 on 1 and 129 DF,  p-value: 0.9881
```

The p-value is way above 0.05, so the impact of green cards to wins (or losses) could just be random. The adjusted R-squared is all of -0.00775 percent. We've do not have something here. Let's use our coefficients to look at Maryland's 2024-25 season.

```r
(0.0004536 * 31) + 8.6319063
```

```
[1] 8.645968
```

This model says that based only on Maryland's green cards, they should have won 8, maybe 9 games. They won 15. Not a great model for many teams.

The power in combining scatterplots + regression is that we can answer two questions: is there a relationship, and how meaningful is it?

23 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.

Here's one: placing players or teams into groups based on performance.

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/footballlogs24.csv")
```

```
Rows: 1344 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    5.81    5.56
```

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 = 5.811255) +
  geom_hline(yintercept = 5.562701)
```

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.

```
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 = 5.811255) +
  geom_hline(yintercept = 5.562701) +
  geom_text_repel(
    data=bigten,
    aes(x=OffensiveYPP, y=DefensiveYPP, label=Team)
  )
```

Well, what do you know about that? Maryland was … in the "worst" quadrant last 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 = 5.811255) +
  geom_hline(yintercept = 5.562701) +
  geom_text_repel(
    data=bigten,
    aes(x=OffensiveYPP, y=DefensiveYPP, label=Team)
  ) +
  labs(title="A Step Back for Maryland", subtitle="The Terps offense and defense puts it amo
  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()
    )
```

```
Warning: ggrepel: 2 unlabeled data points (too many overlaps). Consider
increasing max.overlaps
```

A Step Back for Maryland
The Terps offense and defense puts it among the worst teams in the conference.
Purdue
Rutgers
Northwestern
Maryland
Illinois
UCLA
USC
Michigan
Iowa
Nebraska
Washington
Minnesota
Penn State
Oregon
Ohio State
Indiana
DefensiveYPP
OffensiveYPP
TotalWins
0.0
2.5
5.0
7.5
10.0
Source: NCAA | By Derek Willis

24 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(janitor)
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. Maryland didn't really have someone like Diamond Miller or Abby Meyers on the team, and some of the better scorers have graduated. Who are the shooters-in-waiting?

First we'll load our player data.

```
players <- read_csv("data/wbb_players25.csv") |> clean_names()
```

```
Rows: 5171 Columns: 55
-- Column specification --------------------------------------------------------
Delimiter: ","
```

```
chr  (9): Team, Player, Class, Height, Summary, Pos, Conference, Season, Awards
dbl (46): #, G, GS, MP, FG, FGA, FG%, 3P, 3PA, 3P%, 2P, 2PA, 2P%, eFG%, FT, ...

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)
```

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    37.0   112.0   142.9   222.0   660.0 
```

The median number of shots is 112, but we only really care about prolific ones. So let's use 222 attempts – the third quartile – as our cutoff.

```
shooters <- activeplayers |> filter(fga > 222)
```

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 (`ts_percent` in the data):

```
ggplot() + geom_beeswarm(data=shooters, aes(x=pos, y=ts_percent), color="grey")
```

You can see that there's a lot fewer centers who have attempted more than 222 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 222 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 Terrapins Women's") |>
  filter(fga>222) |>
  arrange(desc(ts_percent))
```

Five Terps took more than 222 shots. Number not on the roster this season? Three: Sarah Te-Biasu, Shyanne Sellers and Christina Dalce.

But how good are they as true shooters? Let's add them to the graphic:

```
ggplot() +
  geom_beeswarm(
    data=shooters,
    aes(x=pos, y=ts_percent), color="grey") +
```

```
  geom_beeswarm(
    data=umd,
    aes(x=pos, y=ts_percent), color="red")
```

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,
    aes(x=pos, y=ts_percent), color="grey") +
  geom_beeswarm(
    data=umd,
    aes(x=pos, y=ts_percent), color="red") +
  geom_text_repel(
    data=umd,
    aes(x=pos, y=ts_percent, label=player))
```

So Sarah Te-Biasu was our best shooter by true shooting percentage, with Shyanne Sellers close behind. The other three were at or above average shooters for that volume of shooting.

24.1 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,
    aes(x=pos, y=ts_percent), color="grey") +
  geom_quasirandom(
    data=umd,
    aes(x=pos, y=ts_percent), color="red") +
  geom_text_repel(
    data=umd,
    aes(x=pos, y=ts_percent, label=player))
```

Quasirandom spreads out the dots you see in beeswarm using – you guessed it – quasirandom spacing.

```
ggplot() +
  geom_jitter(
    data=shooters,
    aes(x=pos, y=ts_percent), color="grey") +
  geom_jitter(
    data=umd,
    aes(x=pos, y=ts_percent), color="red") +
  geom_text_repel(
    data=umd,
    aes(x=pos, y=ts_percent, 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.

25 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 2024-25 college football playoff rankings:

```
rankings <- read_csv("data/cfbranking24.csv")
```

```
Rows: 150 Columns: 3
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): Team
dbl (2): Week, Rank

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 3
   Week  Rank Team
  <dbl> <dbl> <chr>
1    10     1 Oregon
2    10     2 Ohio State
3    10     3 Georgia
4    10     4 Miami
5    10     5 Texas
6    10     6 Penn State
```

So Oregon was ranked in the first (yawn), followed by Ohio State (double yawn), Georgia (of course), Miami (!) 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 compute_group(...): 'StatBump' needs at least two observations per
group
Warning in compute_group(...): 'StatBump' needs at least two observations per
group
```

Well, it's a start.

The warning that you're seeing is that there are two teams last season that made one appearance in 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 == 15 & 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()`

```r
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.

```r
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("HEX CODE", "HEX CODE")) + # replace HEX CODE with each team
  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, label
  geom_text(data = newrankings |> filter(Week == max(Week)), aes(x = Week + .2, y=Rank, label
  scale_color_manual(values = c("HEX CODE", "HEX CODE")) # replace HEX CODE with each team's
  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, label
  geom_text(data = newrankings |> filter(Week == max(Week)), aes(x = Week + .2, y=Rank, label
  labs(title="The top ten was anything but boring in 2024-25", subtitle="", y= "Rank", x = "W
  theme_minimal() +
  theme(
    legend.position = "none",
    panel.grid.major = element_blank()
    ) +
  scale_color_manual(values = c("HEX CODE", "HEX CODE")) # replace HEX CODE with each team's
  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, label
  geom_text(data = newrankings |> filter(Week == max(Week)), aes(x = Week + .2, y=Rank, label
  labs(title="The top ten was anything but boring in 2024-25", subtitle="", y= "Rank", x = "W
  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("HEX CODE", "HEX CODE")) # replace HEX CODE with each team's
  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, label
  geom_text(data = newrankings |> filter(Week == max(Week)), aes(x = Week + .2, y=Rank, label
  labs(title="The top ten was anything but boring in 2024-25", subtitle="", y= "Rank", x = "W
  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("HEX CODE", "HEX CODE")) # replace HEX CODE with each team's
  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))
```

26 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 2015-2025.

Load libraries.

```r
library(tidyverse)
library(gt)
```

And the data.

```r
logs <- read_csv("data/logs2425.csv")
```

```
Rows: 23899 Columns: 60
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (10): Season, GameType, TeamFullName, Opponent, HomeAway, W_L, OT, URL,...
dbl  (49): 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 ask this question: which college basketball team saw the greatest decrease in three point attempts per game between 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 | 2023-2024 season threes | 2024-2025 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 == "2023-2024" | Season == "2024-2025") |>
  group_by(Team, Season) |>
  summarise(Total3PA = sum(Team3PA)) |>
  pivot_wider(names_from=Season, values_from = Total3PA) |>
  filter(!is.na(`2024-2025`)) |>
  mutate(PercentChange = (`2024-2025`-`2023-2024`)/`2023-2024`) |>
  arrange(PercentChange) |>
  ungroup() |>
  slice_head(n=10) # just want a top 10 list, but can't use top_n!
```

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

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()
```

So there you have it. Temple changed its team so much they took 35 percent fewer threes in 2024-25 from the season before. Where did Maryland come out? We ranked pretty low in college basketball in terms of fewer threes from the season before, because the Terps actually took 12% more.

`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.

Team	2023-2024	2024-2025	PercentChange
Temple	931	597	-0.3587540
Tarleton State	623	408	-0.3451043
Morehead State	912	614	-0.3267544
Florida International	840	585	-0.3035714
Milwaukee	861	613	-0.2880372
Boston College	826	597	-0.2772397
College of Charleston	1072	782	-0.2705224
Fairfield	943	729	-0.2269353
Arkansas-Pine Bluff	898	696	-0.2249443
Georgia	883	692	-0.2163080

Team	2023-2024	2024-2025	Percent Change
Temple	931	597	-0.3587540
Tarleton State	623	408	-0.3451043
Morehead State	912	614	-0.3267544
Florida International	840	585	-0.3035714
Milwaukee	861	613	-0.2880372
Boston College	826	597	-0.2772397
College of Charleston	1072	782	-0.2705224
Fairfield	943	729	-0.2269353
Arkansas-Pine Bluff	898	696	-0.2249443
Georgia	883	692	-0.2163080

```
threechange |>
  gt() |>
  cols_label(
    PercentChange = "Percent Change"
  )
```

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`

[

Did Maryland Shoot Fewer Threes in 2024-25?

No, the Terps shot more. But these 10 teams completely changed their offenses.] Did Maryland Shoot Fewer Threes in 2024-25?

No, the Terps shot more. But these 10 teams completely changed their offenses.

Team	2023-2024	2024-2025	Percent Change
Temple	931	597	-0.3587540
Tarleton State	623	408	-0.3451043
Morehead State	912	614	-0.3267544
Florida International	840	585	-0.3035714
Milwaukee	861	613	-0.2880372
Boston College	826	597	-0.2772397
College of Charleston	1072	782	-0.2705224
Fairfield	943	729	-0.2269353
Arkansas-Pine Bluff	898	696	-0.2249443
Georgia	883	692	-0.2163080

```
threechange |>
  gt() |>
  cols_label(
    PercentChange = "Percent Change"
  ) |>
  tab_header(
    title = "Did Maryland Shoot Fewer Threes in 2024-25?",
    subtitle = "No, the Terps shot more. But these 10 teams completely changed their offenses
  )
```

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 Fewer Threes in 2024-25?",
    subtitle = "No, the Terps shot more. But these 10 teams completely changed their offenses
```

[

Did Maryland Shoot Fewer Threes in 2024-25?

No, the Terps shot more. But these 10 teams completely changed their offenses.] Did Maryland Shoot Fewer Threes in 2024-25?

No, the Terps shot more. But these 10 teams completely changed their offenses.

Team	2023-2024	2024-2025	Percent Change
Temple	931	597	-0.3587540
Tarleton State	623	408	-0.3451043
Morehead State	912	614	-0.3267544
Florida International	840	585	-0.3035714
Milwaukee	861	613	-0.2880372
Boston College	826	597	-0.2772397
College of Charleston	1072	782	-0.2705224
Fairfield	943	729	-0.2269353
Arkansas-Pine Bluff	898	696	-0.2249443
Georgia	883	692	-0.2163080

```
  ) |> tab_style(
    style = cell_text(color = "black", weight = "bold", align = "left"),
    locations = cells_title("title")
  )
```

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 Fewer Threes in 2024-25?",
    subtitle = "No, the Terps shot more. But these 10 teams completely changed their offenses
  ) |> 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 Fewer Threes in 2024-25?

No, the Terps shot more. But these 10 teams completely changed their offenses.] Did Maryland Shoot Fewer Threes in 2024-25?

No, the Terps shot more. But these 10 teams completely changed their offenses.

Team	2023-2024	2024-2025	Percent Change
Temple	931	597	-0.3587540
Tarleton State	623	408	-0.3451043
Morehead State	912	614	-0.3267544
Florida International	840	585	-0.3035714
Milwaukee	861	613	-0.2880372
Boston College	826	597	-0.2772397
College of Charleston	1072	782	-0.2705224
Fairfield	943	729	-0.2269353
Arkansas-Pine Bluff	898	696	-0.2249443
Georgia	883	692	-0.2163080

```
  )
```

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 Fewer Threes in 2024-25?",
    subtitle = "No, the Terps shot more. But these 10 teams completely changed their offenses
  ) |> 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.spo
```

[

Did Maryland Shoot Fewer Threes in 2024-25?

No, the Terps shot more. But these 10 teams completely changed their offenses.] Did Maryland Shoot Fewer Threes in 2024-25?

No, the Terps shot more. But these 10 teams completely changed their offenses.

Team	2023-2024	2024-2025	Percent Change
Temple	931	597	-0.3587540
Tarleton State	623	408	-0.3451043
Morehead State	912	614	-0.3267544
Florida International	840	585	-0.3035714
Milwaukee	861	613	-0.2880372
Boston College	826	597	-0.2772397
College of Charleston	1072	782	-0.2705224
Fairfield	943	729	-0.2269353
Arkansas-Pine Bluff	898	696	-0.2249443
Georgia	883	692	-0.2163080

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 Fewer Threes in 2024-25?",
    subtitle = "No, the Terps shot more. But these 10 teams completely changed their offenses
  ) |>
  tab_style(
    style = cell_text(color = "black", weight = "bold", align = "left"),
    locations = cells_title("title")
  ) |>
  tab_style(
    style = cell_text(color = "black", align = "left"),
```

[

Did Maryland Shoot Fewer Threes in 2024-25?

No, the Terps shot more. But these 10 teams completely changed their offenses.] Did Maryland Shoot Fewer Threes in 2024-25?

No, the Terps shot more. But these 10 teams completely changed their offenses.

Team	**2023-2024**	**2024-2025**	Percent Change
Temple	931	597	-0.3587540
Tarleton State	623	408	-0.3451043
Morehead State	912	614	-0.3267544
Florida International	840	585	-0.3035714
Milwaukee	861	613	-0.2880372
Boston College	826	597	-0.2772397
College of Charleston	1072	782	-0.2705224
Fairfield	943	729	-0.2269353
Arkansas-Pine Bluff	898	696	-0.2249443
Georgia	883	692	-0.2163080

By: Derek Willis | **Source:** Sports Reference

```
    locations = cells_title("subtitle")
  ) |>
  tab_source_note(
    source_note = md("**By:** Derek Willis  |  **Source:** [Sports Reference](https://www.spo
  ) |>
  tab_style(
     locations = cells_column_labels(columns = everything()),
     style = list(
       cell_borders(sides = "bottom", weight = px(3)),
       cell_text(weight = "bold", size=12)
     )
   )
```

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 Fewer Threes in 2024-25?",
    subtitle = "No, the Terps shot more. But these 10 teams completely changed their offenses
  ) |>
  tab_source_note(
    source_note = md("**By:** Derek Willis  |  **Source:** [Sports Reference](https://www.spo
  ) |>
  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")
```

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 Fewer Threes in 2024-25?",
    subtitle = "No, the Terps shot more. But these 10 teams completely changed their offenses
  ) |>
  tab_source_note(
    source_note = md("**By:** Derek Willis  |  **Source:** [Sports Reference](https://www.spo
  ) |>
  tab_style(
```

[

Did Maryland Shoot Fewer Threes in 2024-25?

No, the Terps shot more. But these 10 teams completely changed their offenses.] Did Maryland Shoot Fewer Threes in 2024-25?

No, the Terps shot more. But these 10 teams completely changed their offenses.

Team	**2023-2024**	**2024-2025**	Percent Change
Temple	931	597	-0.3587540
Tarleton State	623	408	-0.3451043
Morehead State	912	614	-0.3267544
Florida International	840	585	-0.3035714
Milwaukee	861	613	-0.2880372
Boston College	826	597	-0.2772397
College of Charleston	1072	782	-0.2705224
Fairfield	943	729	-0.2269353
Arkansas-Pine Bluff	898	696	-0.2249443
Georgia	883	692	-0.2163080

By: Derek Willis | **Source:** Sports Reference

```
    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 Fewer Threes in 2024-25?

No, the Terps shot more. But these 10 teams completely changed their offenses.] Did Maryland Shoot Fewer Threes in 2024-25?

No, the Terps shot more. But these 10 teams completely changed their offenses.

Team	**2023-2024**	**2024-2025**	Percent Change
Temple	931	597	-35.9%
Tarleton State	623	408	-34.5%
Morehead State	912	614	-32.7%
Florida International	840	585	-30.4%
Milwaukee	861	613	-28.8%
Boston College	826	597	-27.7%
College of Charleston	1072	782	-27.1%
Fairfield	943	729	-22.7%
Arkansas-Pine Bluff	898	696	-22.5%
Georgia	883	692	-21.6%

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 Boston College. 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 Fewer Threes in 2024-25?",
    subtitle = "No, the Terps shot more. But these 10 teams completely changed their offenses
  ) |>
  tab_source_note(
    source_note = md("**By:** Derek Willis  |  **Source:** [Sports Reference](https://www.spo
  ) |>
  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 == "Boston College")
  )
```

Two things here:

1. Dear God that color scheme is awful, which is fitting for a school that has a non-rhyming mascot.
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.

[

Did Maryland Shoot Fewer Threes in 2024-25?

No, the Terps shot more. But these 10 teams completely changed their offenses.] Did Maryland Shoot Fewer Threes in 2024-25?

No, the Terps shot more. But these 10 teams completely changed their offenses.

Team	**2023-2024**	**2024-2025**	Percent Change
Temple	931	597	-35.9%
Tarleton State	623	408	-34.5%
Morehead State	912	614	-32.7%
Florida International	840	585	-30.4%
Milwaukee	861	613	-28.8%
Boston College	826	597	-27.7%
College of Charleston	1072	782	-27.1%
Fairfield	943	729	-22.7%
Arkansas-Pine Bluff	898	696	-22.5%
Georgia	883	692	-21.6%

By: Derek Willis | **Source:** Sports Reference

27 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/logs25.csv")
```

```
Rows: 11962 Columns: 60
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (10): Season, GameType, TeamFullName, Opponent, HomeAway, W_L, OT, URL,...
dbl  (49): 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 MBB") |> filter(!is.na(TeamFGPCT))
```

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: the average field goal percentage for the whole conference.

```
big10_avg <- big10 |> summarise(avg_fgpct = mean(TeamFGPCT)) |> pull(avg_fgpct)
```

```
ggplot() +
  geom_hline(yintercept=big10_avg, 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.

27.1 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=big10_avg, 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=big10_avg, 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.

27.2 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
Oregon
Penn State
Purdue
Rutgers
uthern Califor
UCLA
Washington
Wisconsin
OpponentFGPCT
TeamFGPCT
0.0
0.2
0.4
0.6
W_L
L
W

28 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: 146 Columns: 14
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (2): Institution, Conference
dbl (12): 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, ...

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 = ((`2024`-`2023`)/`2023`)*100)
```

Let's chart the top 10 and bottom 10 of college football ticket growth … and shrinkage.

```
top10 <- attendance |> top_n(10, wt=change) |> arrange(desc(change))
bottom10 <- attendance |> top_n(10, wt=-change) |> arrange(change)
```

Ohio State is benefiting from the extended playoffs, but hello, SMU and Indiana!

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 attenc
  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 Derek
  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.

```
nw <- bottom10 |> filter(Institution == "Northwestern")
sm <- top10 |> filter(Institution == "SMU")
```

```
bar1 <- ggplot() +
  geom_bar(data=top10, aes(x=reorder(Institution, change), weight=change)) +
```

```
  geom_bar(data=sm, aes(x=reorder(Institution, change), weight=change), fill="blue") +
  coord_flip() +
  labs(title="College football winners...", subtitle = "Not every football program saw attenc
  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=nw, aes(x=reorder(Institution, change), weight=change), fill="purple") +
  coord_flip() +
  labs(title = "... and losers", subtitle= "", x="", y="",  caption="Source: NCAA | By Derek
  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 2023 But some reall
SMU
Ohio St.
Colorado St.
Indiana
Texas
Central Mich.
North Texas
Pittsburgh
Northern Ill.
Vanderbilt
0
20
40
60
Percent change
Memphis
Georgia Tech
Northwestern
Arizona St.
Kansas
Baylor
UTSA
Tulane
Stanford
Houston
−30
−20
−10
0
Source: NCAA | By Derek Willis

29 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)
library(ggrepel)
```

Let's say we want to highlight the top scorers in women's college lacrosse. 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/wlax_playerstats_2025.csv") |> filter(player_GP > 0)
```

```
Rows: 4043 Columns: 45
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (9): season_name, sport_id, school_name, ncaa_division_formatted, team_...
dbl (32): season, school_id, team_id, ncaa_division, player_id, player_jerse...
lgl  (4): player_height, player_GB, player_FOW, player_FO_taken

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 games and points.

```
ggplot() + geom_point(data=players, aes(x=player_GP, y=player_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 120 points.

```
topscorers <- players |> filter(player_PTS > 120)
```

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=player_GP, y=player_PTS), color="grey") +
  geom_point(data=topscorers, aes(x=player_GP, y=player_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=player_GP, y=player_PTS), color="grey") +
  geom_point(data=topscorers, aes(x=player_GP, y=player_PTS), color="black") +
  geom_encircle(data=topscorers, aes(x=player_GP, y=player_PTS), s_shape=1, expand=1, colour=
```

Whoa. That's … not good.

Let's go the opposite direction.

```
ggplot() +
  geom_point(data=players, aes(x=player_GP, y=player_PTS), color="grey") +
  geom_point(data=topscorers, aes(x=player_GP, y=player_PTS), color="black") +
  geom_encircle(data=topscorers, aes(x=player_GP, y=player_PTS), s_shape=0, expand=0, colour=
```

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, use geom_text_repel to make some space for the names and extend the x-axis for more room.

```
ggplot() +
  geom_point(data=players, aes(x=player_GP, y=player_PTS), color="grey") +
  geom_point(data=topscorers, aes(x=player_GP, y=player_PTS), color="black") +
  geom_encircle(data=topscorers, aes(x=player_GP, y=player_PTS), s_shape=.5, expand=.03, colo
  geom_text_repel(data=topscorers, aes(x=player_GP, y=player_PTS, label=player_full_name), hj
  scale_x_continuous(limits = c(0, 28))
```

Now let's clean this up and make it presentable. If you look at the top scorers, all four are guards. So here's what a chart telling that story might look like.

```
ggplot() +
  geom_point(data=players, aes(x=player_GP, y=player_PTS), color="grey") +
  geom_point(data=topscorers, aes(x=player_GP, y=player_PTS), color="black") +
  geom_encircle(data=topscorers, aes(x=player_GP, y=player_PTS), s_shape=.5, expand=.03, colo
  geom_text_repel(data=topscorers, aes(x=player_GP, y=player_PTS, label=player_full_name), hj
  scale_x_continuous(limits = c(0, 28)) +
  labs(title="Madison Taylor Alone At Top", subtitle="The Northwestern star is by far the top
  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()
    )
```

29.1 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 2024-25 Maryland roster.

```
players24 <- read_csv("https://thescoop.org/sports-data-files/players25.csv")
```

```
Rows: 5818 Columns: 57
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (10): Team, Player, Class, Height, Hometown, High School, Summary, Pos, ...
dbl (47): #, Weight, G, GS, MP, FG, FGA, FG%, 3P, 3PA, 3P%, 2P, 2PA, 2P%, eF...

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.
```

```
mdtop <- players24 |> filter(Team == "Maryland Terrapins Men's") |> top_n(3, `3PA`)
```

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=players24, aes(x=MP, y=`3PA`), color="grey") +
  geom_point(data=mdtop, aes(x=MP, y=`3PA`), color="red") +
  geom_encircle(data=mdtop, aes(x=MP, y=`3PA`), s_shape=.02, expand=.10, colour="red") +
  geom_text_repel(data=mdtop, aes(x=MP, y=`3PA`, label=Player)) +
  geom_text(aes(x=400, y=200, label="Maryland's top three shooters")) +
  labs(title="Terp Task: Replace the Big Three", subtitle="None of Maryland's most prolific 
  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()
    )
```

```
Warning: Removed 929 rows containing missing values or values outside the scale range
(`geom_point()`).
```

None of them return this season.

30 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.

30.1 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:

```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 "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
# i 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
# i 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.

30.2 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 Set` Season
  <dbl> <chr>                     <chr> <chr> <chr> <dbl> <dbl>     <dbl> <chr>
1     1 Lindsey Ruddins, UC Sant~ 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 Robert~ Sr.   5-10  OH      106   560      5.28 2017-~
4     4 Carlyle Nusbaum, Lipscom~ Jr.   5-10  OH      100   522      5.22 2017-~
5     5 Veronica Jones-Perry, BY~ Jr.   6-0   OH      118   569      4.82 2017-~
6     6 Torrey Van Winden, Cal P~ So.   6-3   OH      101   477      4.72 2017-~
```

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.

```r
kills |> separate(Player, into=c("Player", "School"), sep=",") |> head()
```

```
# A tibble: 6 x 10
   Rank Player             School Cl    Ht    Pos       S Kills `Per Set` 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-Pe~ " 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-~
```

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 \\.

```r
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 Conference Cl    Ht    Pos       S Kills `Per Set` Season
  <dbl> <chr>   <chr>  <chr>      <chr> <chr> <chr> <dbl> <dbl>     <dbl> <chr>
1     1 Lindse~ " UC ~ Big West)  So.   6-2   OH       90   526      5.84 2017-~
2     2 Pilar ~ " Ark~ SEC)       Sr.   5-11  OH      116   634      5.47 2017-~
3     3 Laura ~ " Ora~ Summit Le~ Sr.   5-10  OH      106   560      5.28 2017-~
4     4 Carlyl~ " Lip~ ASUN)      Jr.   5-10  OH      100   522      5.22 2017-~
5     5 Veroni~ " BYU~ WCC)       Jr.   6-0   OH      118   569      4.82 2017-~
6     6 Torrey~ " Cal~ Big West)  So.   6-3   OH      101   477      4.72 2017-~
```

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.

```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="\\(") |>
  head()
```

```
# A tibble: 6 x 11
   Rank Player  School Conference Cl    Ht    Pos       S Kills `Per Set` Season
  <dbl> <chr>   <chr>  <chr>      <chr> <chr> <chr> <dbl> <dbl>     <dbl> <chr>
1     1 Lindse~ " UC ~ Big West)  So.   6-2   OH       90   526      5.84 2017-~
2     2 Pilar ~ " Ark~ SEC)       Sr.   5-11  OH      116   634      5.47 2017-~
3     3 Laura ~ " Ora~ Summit Le~ Sr.   5-10  OH      106   560      5.28 2017-~
4     4 Carlyl~ " Lip~ ASUN)      Jr.   5-10  OH      100   522      5.22 2017-~
5     5 Veroni~ " BYU~ WCC)       Jr.   6-0   OH      118   569      4.82 2017-~
6     6 Torrey~ " Cal~ Big West)  So.   6-3   OH      101   477      4.72 2017-~
```

One last thing: see the trailing paren?

```
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 Conference Cl    Ht    Pos       S Kills `Per Set` Season
  <dbl> <chr>   <chr>  <chr>      <chr> <chr> <chr> <dbl> <dbl>     <dbl> <chr>
1     1 Lindse~ " UC ~ Big West   So.   6-2   OH       90   526      5.84 2017-~
2     2 Pilar ~ " Ark~ SEC        Sr.   5-11  OH      116   634      5.47 2017-~
3     3 Laura ~ " Ora~ Summit Le~ Sr.   5-10  OH      106   560      5.28 2017-~
4     4 Carlyl~ " Lip~ ASUN       Jr.   5-10  OH      100   522      5.22 2017-~
5     5 Veroni~ " BYU~ WCC        Jr.   6-0   OH      118   569      4.82 2017-~
6     6 Torrey~ " Cal~ Big West   So.   6-3   OH      101   477      4.72 2017-~
```

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.

```
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 Conference Cl     Feet Inches Pos       S Kills `Per Set`
```

```
  <dbl> <chr>   <chr>  <chr>      <chr> <dbl>  <dbl> <chr> <dbl> <dbl>     <dbl>
1     1 Lindse~ " UC ~ Big West   So.       6      2 OH       90   526      5.84
2     2 Pilar ~ " Ark~ SEC        Sr.       5     11 OH      116   634      5.47
3     3 Laura ~ " Ora~ Summit Le~ Sr.       5     10 OH      106   560      5.28
4     4 Carlyl~ " Lip~ ASUN       Jr.       5     10 OH      100   522      5.22
5     5 Veroni~ " BYU~ WCC        Jr.       6      0 OH      118   569      4.82
6     6 Torrey~ " Cal~ Big West   So.       6      3 OH      101   477      4.72
# i 1 more variable: Season <chr>
```

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 Conference Cl     Feet Inches Pos       S Kills `Per Set`
  <dbl> <chr>   <chr>  <chr>      <chr> <dbl>  <dbl> <chr> <dbl> <dbl>     <dbl>
1     1 Lindse~ " UC ~ Big West   So.       6      2 OH       90   526      5.84
2     2 Pilar ~ " Ark~ SEC        Sr.       5     11 OH      116   634      5.47
3     3 Laura ~ " Ora~ Summit Le~ Sr.       5     10 OH      106   560      5.28
4     4 Carlyl~ " Lip~ ASUN       Jr.       5     10 OH      100   522      5.22
5     5 Veroni~ " BYU~ WCC        Jr.       6      0 OH      118   569      4.82
6     6 Torrey~ " Cal~ Big West   So.       6      3 OH      101   477      4.72
# i 2 more variables: Season <chr>, Height <dbl>
```

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.

31 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="This i
  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 yards p
  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", 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'
```

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 overperformed
  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.

32 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 2024.

Now load it.

```
logs <- read_csv("data/footballlogs24.csv")
```

```
Rows: 1344 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.

```r
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.

```r
averages <- ypp |> ungroup() |> summarise(AvgOffYardsPer = mean(OffensiveYPP), AvgDefYardsPe

averages
```

```
# A tibble: 1 x 2
  AvgOffYardsPer AvgDefYardsPer
           <dbl>          <dbl>
1           5.81           5.56
```

I also want to highlight playoff teams.

```r
playoff_teams <- c("Indiana", "Notre Dame", "SMU", "Ohio State", "Penn State", "Tennessee",

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.

```
plot <- 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=6.7, y=4.3, 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 the playoff te
  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()
  )

print(plot)
```

All the playoff teams were good
Every BCS team had above average offenses and defenses.
Defensive Points Per Play
Offensive Yards Per Play
Bad Defense, Bad Offense
Good Offense, Good Defense
Boise State
Clemson
SMU
Arizona State
Penn State
Georgia
Oregon
Ohio State
Tennessee
Notre Dame
Indiana
Texas
Source: Sports−Reference.com | By Derek Willis

33 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:

33.1 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.

33.2 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: college football team scoring from 2009-2018.

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.

33.3 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("orange", "blue", "white")) +
  labs(caption="Source: NCAA | Graphic by Derek Willis")
)
```

33.4 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 defense t
    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 defense t
    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 sp
    title="Maryland's overperforming offense",
    subtitle="The Terps' offense was the strength of the team, having to overcome a defense t
    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 their
      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 defense t
  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"))
```

34 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/2023/leader/national/team/offense/split01/categ
```

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 LSU         13    79    12    78     1      1    592       45.5
2     2 Oregon      14    83    12    75     5      0    619       44.2
3     3 USC         13    74    10    66     2      0    544       41.8
4     4 Oklahoma    13    71    15    69     0      1    542       41.7
5     5 Georgia     14    71    21    71     0      1    562       40.1
6     6 Memphis     13    68    13    61     2      0    512       39.4
```

We have data, ready for analysis.

34.1 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 2024.

```r
scoringoffenseurl24 <- "http://www.cfbstats.com/2024/leader/national/team/offense/split01/ca

scoringoffense24 <- scoringoffenseurl24 |>
  read_html() |>
  html_nodes(xpath = '//*[@id="content"]/div[2]/table') |>
  html_table()

scoringoffense24 <- scoringoffense24[[1]]
```

First, how are we going to know, in the data, which year our data is from? We can use mutate.

```
scoringoffense24 <- scoringoffense24 |> mutate(YEAR = 2024)
```

```
Error in `mutate()`:
! Can't transform a data frame with `NA` or `""` names.
```

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.

```
scoringoffense23 <- scoringoffense |> rename(Rank = 1) |> mutate(YEAR = 2023)
scoringoffense24 <- scoringoffense24 |> rename(Rank = 1) |> mutate(YEAR = 2024)
```

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(scoringoffense23, scoringoffense24)
```

Note in the environment tab we now have a data frame called combined that has 268 observations – which just so happens to be what 134 from 2024 and 2023 add up to.

```
head(combined)
```

```
# A tibble: 6 x 11
   Rank Name         G    TD    FG `1XP` `2XP` Safety Points `Points/G`  YEAR
  <int> <chr>    <int> <int> <int> <int> <int>  <int>  <int>      <dbl> <dbl>
1     1 LSU         13    79    12    78     1      1    592       45.5  2023
2     2 Oregon      14    83    12    75     5      0    619       44.2  2023
3     3 USC         13    74    10    66     2      0    544       41.8  2023
4     4 Oklahoma    13    71    15    69     0      1    542       41.7  2023
5     5 Georgia     14    71    21    71     0      1    562       40.1  2023
6     6 Memphis     13    68    13    61     2      0    512       39.4  2023
```

34.2 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/men/2024-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]] |> slice(-1) |> slice(-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 1     Abile~ 34    16    18    .471  -4.12 -1.12 NA    10    10    NA    8
2 2     Air F~ 31    9     22    .290  -4.77 1.43  NA    2     16    NA    4
3 3     Akron~ 35    24    11    .686  2.77  -2.08 NA    13    5     NA    13
```

```
4 4     Alaba~ 37    25    12    .676  20.69 11.80 NA    13    5     NA    14
5 5     Alaba~ 35    12    23    .343  -14.~ -7.52 NA    9     9     NA    7
6 6     Alaba~ 32    13    19    .406  -11.~ -6.87 NA    8     10    NA    7
# i 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. 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, Winl
```

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 OverallWins OverallLosses WinPct OverallSRS OverallSOS
  <dbl> <chr>       <dbl>       <dbl>         <dbl>  <dbl>      <dbl>      <dbl>
1     1 Abilene Ch~    34          16            18  0.471      -4.12      -1.12
2     2 Air Force      31           9            22  0.29       -4.77       1.43
3     3 Akron NCAA     35          24            11  0.686       2.77      -2.08
4     4 Alabama NC~    37          25            12  0.676      20.7       11.8
5     5 Alabama A&M    35          12            23  0.343     -14.4       -7.52
6     6 Alabama St~    32          13            19  0.406     -11.7       -6.87
# i 25 more variables: ConferenceWins <dbl>, ConferenceLosses <dbl>,
#   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>, ...
```

35 Getting CSVs from Sports Reference

Every table of data on Sports Reference can be turned into a csv using the magic of copy and paste. There's some tricks to it.

Step 1: In R Studio, open a plain text file.

Step 2: On the table you want in Sports Reference, find the Share & Export tab and go to Get table as CSV (For Excel). NOTE: THIS TABLE HAS TWO HEADER ROWS. You will need to fix that later.

Step 3: Highlight and copy the data from Sports Reference.

```
601,Nathan Walker,28,STL,C,1,0,0,0,0,0,0.0,0,0,0,0,0,0,0,0,,12,11:38,1,1,0,1,0.0,walkena01
602,Sean Walker,28,LAK,D,4,0,0,0,1,2,0.1,0,0,0,0,0,0,0,6,0.0,52,12:53,4,6,0,0,,walkese01
603,Austin Watson,31,OTT,LW,3,0,1,1,1,0,0.1,0,0,0,0,1,0,0,3,0.0,26,8:45,0,6,0,0,,watsoau01
604,MacKenzie Weegar,29,CGY,D,3,0,3,3,0,0,0.4,0,0,0,0,3,0,0,5,0.0,62,20:48,5,8,0,0,,weegama01
605,Alexander Wennberg,28,SEA,C,4,1,0,1,0,2,0.1,1,0,0,0,0,0,0,5,20.0,80,20:07,5,0,22,37,37.3,wennbal01
606,Zach Werenski,25,CBJ,D,4,1,1,2,-2,0,0.3,1,0,0,0,1,0,0,15,6.7,100,25:00,11,3,0,0,,werenza01
607,Blake Wheeler,36,WPG,RW,2,0,1,1,1,0,0.1,0,0,0,0,1,0,0,5,0.0,31,15:37,1,0,2,2,50.0,wheelbl01
608,Colin White,26,FLA,C,3,2,2,4,4,0,0.6,2,0,0,0,2,0,0,6,33.3,35,11:49,1,1,1,2,33.3,whiteco02
609,Colton White,25,ANA,D,1,0,0,0,-1,0,-0.1,0,0,0,0,0,0,0,1,0.0,14,13:38,1,1,0,0,,whiteco03
610,Zach Whitecloud,26,VEG,D,4,0,2,2,-2,6,0.1,0,0,0,0,2,0,0,3,0.0,62,15:33,9,5,0,0,,whiteza01
611,Chris Wideman,33,MTL,D,4,0,0,0,-1,2,0.1,0,0,0,0,0,0,0,6,0.0,67,16:43,4,0,0,0,,widemch01
612,Miles Wood,27,NJD,LW,3,1,0,1,-1,6,0.1,1,0,0,0,0,0,0,5,20.0,33,11:01,0,3,0,0,,woodmi01
613,Shane Wright,19,SEA,C,2,0,0,0,0,0,0.0,0,0,0,0,0,0,0,1,0.0,13,6:32,1,0,1,2,33.3,wrighsh01
614,Arber Xhekaj,22,MTL,D,4,0,1,1,0,8,0.2,0,0,0,0,1,0,0,5,0.0,58,14:28,8,16,0,0,,xhekaar01
615,Kailer Yamamoto,24,EDM,RW,2,0,1,1,1,0,0.1,0,0,0,0,1,0,0,2,0.0,33,16:24,1,1,0,0,,yamamka01
616,Pavel Zacha,25,BOS,C,4,1,2,3,-3,0,0.1,0,1,0,0,2,0,0,11,9.1,60,15:00,2,1,4,3,57.1,zachapa01
617,Filip Zadina,23,DET,RW,1,0,0,0,0,0,0.0,0,0,0,0,0,0,0,0,,14,14:25,1,1,0,0,,zadinfi01
618,Nikita Zadorov,27,CGY,D,3,0,0,0,3,4,0.3,0,0,0,0,0,0,0,5,0.0,51,16:57,1,7,0,0,,zadorni01
619,Nikita Zaitsev,31,OTT,D,1,0,0,0,0,0,0.0,0,0,0,0,0,0,0,0,,12,12:12,0,6,0,0,,zaitsni01
620,Yegor Zamula,22,PHI,D,3,0,1,1,0,0,0.1,0,0,0,0,1,0,0,2,0.0,41,13:41,1,1,0,0,,zamulye01
621,Jakub Zboril,25,BOS,D,4,0,0,0,1,0,0.0,0,0,0,0,0,0,0,3,0.0,61,15:15,4,3,0,0,,zborija01
622,Trevor Zegras,21,ANA,C,4,2,1,3,-3,0,0.2,1,1,0,0,0,1,0,6,33.3,74,18:27,1,3,12,20,37.5,zegratr01
623,Fabian Zetterlund,23,NJD,W,1,0,1,1,1,0,0.1,0,0,0,0,1,0,0,1,0.0,10,9:40,0,1,0,0,,zettefa01
624,Mika Zibanejad,29,NYR,C,4,4,4,8,2,0,0.9,0,3,1,1,2,2,0,24,16.7,84,20:59,2,2,36,33,52.2,zibanmi01
625,Artem Zub,27,OTT,D,3,1,1,2,1,2,0.3,1,0,0,0,1,0,0,3,33.3,60,20:01,2,8,0,0,,zubar01
626,Mats Zuccarello,35,MIN,LW,3,2,3,5,-7,2,0.3,0,2,0,0,0,3,0,12,16.7,64,21:10,1,4,3,3,50.0,zuccama01
627,Jason Zucker,31,PIT,LW,3,1,2,3,4,0,0.3,1,0,0,0,1,1,0,9,11.1,46,15:27,2,8,0,0,,zuckeja01
```

More 2022-23 NHL Pages

2022-23 NHL Season

More 2022-23 NHL Pages

Step 4: Paste it into your plain text file in R Studio and fix your headers. Remember that data in R – and pretty much every other data analysis platform – must have one row for headers. What you need to do now, in the text file, is check and see if you have any headers in that second row that repeat. If you have multiple G for goals – one is total goals, one is goals per

something – you need to fix that manually. In my example, the first row is header names and none of them repeat. I'm good to move forward.

Step 6: Save your text file as whatever you want to name it DOT csv and put it where you've got your project data.

Step 7: Import your data like you would in any other assignment. You just created a CSV file.

36 Building your own blog with Quarto

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.

36.1 Setup

With those requirements in mind, we're going to use a library called Quarto, which supercharges R Markdown to make documents, websites, blogs and books. It's new, and growing, and you can do a lot with it.

You'll first go to Quarto's website and wait just one moment before you do anything.

Before you install Quarto, you should close R Studio. So close R Studio, install it, restart R Studio and come back to this tutorial. Then install Quarto.

After that, we're ready to start making a blog. To do that, go to File > New Project and select New Directory, and follow the instructions here. In the Project Type view, you may have to scroll down to find "Quarto blog" but that's the choice you want.

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. Downloads would work, and ideally it would be "alongside" your class repository. Remember where it is because you're going to need this later.

Choose "None" as your engine and check the "Open in new session" box. Do NOT create a new repository at this point, then create the project.

When you hit Create Project, you should get an R Studio screen. You might have two files open – _quarto.yml and index.qmd. If you don't, open them (you can use the files pane in the bottom right). The important one for now is _quarto.yml. This file will look different depending on some choices you make.

The choices you need to make:

- The website title should be yours.
- The Github account (more on this soon) and Twitter handles should be yours.
- If you want a custom favicon logo, you'll have to include one. You can ignore this for a while.
- You should pick a not-default theme. If you want to have themes for light and dark mode, the code below handles that. If you want just one theme, the default code you get in your _quarto.yml file does that.
- Add the execute: freeze bits below to your file. This way, your code only executes when you change it. This will become more important later when you have multiple posts.

```
project:
  type: website

website:
  title: "Derek's Collection of Miscellany"
  description: "A blog about random fiddling with code and data."
  navbar:
    right:
      - about.qmd
      - icon: github
        href: https://github.com/dwillis
      - icon: twitter
        href: https://twitter.com/derekwillis
  favicon: logo.png

format:
  html:
    theme:
      light: litera
      dark: cyborg
    css: styles.css

editor: source
```

```
execute:
  freeze: auto
```

36.2 Creating a post

Under the files tab on the lower right, you'll see a folder named posts. In there, you'll find two folders with sample posts in them. There's not much to them, but they're useful to look at before we delete them.

To create a post, you're going to first create a folder for it. **The folder name matters because it's going to be part of your url.** For this to work your folder name has to do a couple of things:

- Contain only lowercase letters and dashes instead of spaces. So My Awesome Post will not work. It should be my-awesome-post.
- Contain words that are in your post, and would be something people would search for. If your post is about Maryland women's run into the NCAA Tournament, make sure your post title and folder name contains those words.
- In other words, **you need to think about what your post is about before you make this folder**. Can you change it later? Yes. But the sooner you start focusing on a topic the better.

After you have created the folder, go to the button next to the create folder button and create a Quarto Document. Name that document index.qmd. **DO NOT NAME IT CAPITAL I INDEX. IT'S LOWER CASE I.**

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 is an example you should copy and edit for all of your posts.

```
---
title: "CHANGE THIS TO BE THE TITLE OF YOUR POST"
author: "THIS IS YOUR NAME"
date: '2025-XX-XX'
categories: [lowercaseword, lowercaseword, lowercaseword]
image: "image.png"
format:
  html:
    code-fold: true
---
```

For us, the cold-fold bit is important, and it is not standard. This will allow users to see your code that makes your graphics, without displaying it. Users can decide if they want to see the code.

Now that you have a folder and an index.qmd file with a proper yaml header, you're ready to write your post. This document works just the same as any .qmd file we've worked with all semester. Create text between code blocks, and your code blocks will execute to produce graphics.

36.3 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.

You can use the Render button in your editor to do that, and if you want to see it inside RStudio you can check the "Preview in Viewer Pane" from the drop-down menu just to the right of the Render button. Otherwise, the site will render in your browser locally.

36.4 Publishing your site

Quarto will take your .qmd 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: Your GitHub account

Go to your GitHub 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 dwillis. 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. You might already have this done if, for example, you've set it up for JOUR352. If so, you can skip to the very next step (where you set up publishing).

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 dwillis.github.io because my username is dwillis. 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.

Finally, setup your repository so that it will publish your site every time you push updated or new files from the `main` branch. You can do that under "Settings":

Step 3: (optional if you're fine using the command line interface)

Don't close your GitHub window.

In a new tab, download the GitHub Desktop App and install it if you don't have it already. 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 - so if you put it in Downloads, it's Downloads.** 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 `_quarto.yml` in your blog project. Near the top, below type: website, add this line:

```
output-dir: ../sportsblog/docs
```

If the name of your directory is something other than "sportsblog", you'll want to change it in that line to match the actual directory.

Save _quarto.yml.

Then, add a .nojekyll file to the root of your repository that tells GitHub Pages not to do additional processing of your published site using Jekyll (the GitHub default site generation tool). The easiest way to go that is to create a new text file in your blog project directory named `.nojekyll` using RStudio. It should be empty. Just make sure it is named exactly that, with the leading period.

Once you are ready to build your site, you can do so using the `quarto render` command or hit the Render button in RStudio, and then git push (or using GitHub Desktop).

Step 5 (GitHub Desktop only)

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.

37 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?

37.1 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?

37.2 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?

37.3 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?

37.4 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.

37.5 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.

38 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.

38.1 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 1 against Buffalo.

```
maryland <- cfbd_pbp_data(
 2024,
  week=1,
  season_type = "regular",
  team = "Maryland",
  epa_wpa = TRUE,
)
```

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(2022, week=1, season_type = "regular", team = "Maryland", epa_wpa = TRU
Sys.sleep(2)
wk2 <- cfbd_pbp_data(2022, week=2, season_type = "regular", team = "Maryland", epa_wpa = TRU
Sys.sleep(2)
wk3 <- cfbd_pbp_data(2022, week=3, season_type = "regular", team = "Maryland", epa_wpa = TRU
Sys.sleep(2)
wk4 <- cfbd_pbp_data(2022, week=4, season_type = "regular", team = "Maryland", epa_wpa = TRU
Sys.sleep(2)
wk5 <- cfbd_pbp_data(2022, week=5, season_type = "regular", team = "Maryland", epa_wpa = TRU
Sys.sleep(2)
wk6 <- cfbd_pbp_data(2022, week=6, season_type = "regular", team = "Maryland", epa_wpa = TRU
Sys.sleep(2)
wk8 <- cfbd_pbp_data(2022, week=8, season_type = "regular", team = "Maryland", epa_wpa = TRU
Sys.sleep(2)
wk9 <- cfbd_pbp_data(2022, week=9, season_type = "regular", team = "Maryland", epa_wpa = TRU
Sys.sleep(2)
wk10 <- cfbd_pbp_data(2022, week=10, season_type = "regular", team = "Maryland", epa_wpa = T
```

```
Sys.sleep(2)
wk11 <- cfbd_pbp_data(2022, week=11, season_type = "regular", team = "Maryland", epa_wpa = TR
Sys.sleep(2)
wk12 <- cfbd_pbp_data(2022, week=12, season_type = "regular", team = "Maryland", epa_wpa = TR

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 2 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)
```

The most heartbreaking play of the season, according to our data? A third quarter run for two yards against Northwestern. Hmm - Maryland won that game, though. The other top plays - mostly against Purdue and a blocked punt by Ohio State - seem more in line with what we want.

38.2 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 six seasons?

```
progressr::with_progress({
  wbb_player_box <- wehoop::load_wbb_player_box(2021:2025)
})
```

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", !is.na(points)) |>
  group_by(athlete_display_name, season) |>
  summarise(totalPoints = sum(as.numeric(points))) |>
  arrange(desc(totalPoints)) |>
  ungroup() |>
  top_n(10, wt=totalPoints)
```

Maryland relied on Diamond Miller's scoring last year more than they have any player's in the past six seasons.