Sports Data Analysis and Visualization

Code, data, visuals and the Tidyverse for journalists and other storytellers

By Matt Waite

July 29, 2019

Contents

Chapter 1

Throwing cold water on hot takes

The 2018 season started out disastrously for the Nebraska Cornhuskers. The first game against a probably overmatched opponent? Called on account of an epic thunderstorm that plowed right over Memorial Stadium. The next game? Loss. The one following? Loss. The next four? All losses, after the fanbase was whipped into a hopeful frenzy by the hiring of Scott Frost, national title winning quarterback turned hot young coach come back home to save a mythical football program from the mediocrity it found itself mired in.

All that excitement lay in tatters.

On sports talk radio, on the sports pages and across social media and cafe conversations, one topic kept coming up again and again to explain why the team was struggling: Penalties. The team was just committing too many of them. In fact, six games and no wins into the season, they were dead last in the FBS penalty yards.

Worse yet for this line of reasoning? Nebraska won game 7, against Minnesota, committing only six penalties for 43 yards, just about half their average over the season. Then they won game 8 against FCS patsy Bethune Cookman, committing only five penalties for 35 yards. That's a whopping 75 yards less than when they were losing. See? Cut the penalties, win games screamed the radio show callers.

The problem? It's not true. Penalties might matter for a single drive. They may even throw a single game. But if you look at every top-level college football team since 2009, the number of penalty yards the team racks up means absolutely nothing to the total number of points they score. There's no relationship between them. Penalty yards have no discernible influence on points beyond just random noise.

Put this another way: If you were Scott Frost, and a major college football program was paying you $5 million a year to make your team better, what should you focus on in practice? If you had growled at some press conference that you're going to work on penalties in practice until your team stops committing them, the results you'd get from all that wasted practice time would be impossible to separate from just random chance. You very well may reduce your penalty yards and still lose.

How do I know this? Simple statistics.

That's one of the three pillars of this book: Simple stats. The three pillars are:

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

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

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

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

1.1 Requirements and Conventions

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

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

Going forward, you'll see passages like this:

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

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

1.2 About this book

This book is the collection of class materials for the author's Sports Data Analysis and Visualization class at the University of Nebraska-Lincoln's College of

Journalism and Mass Communications. There's some things you should know about it:

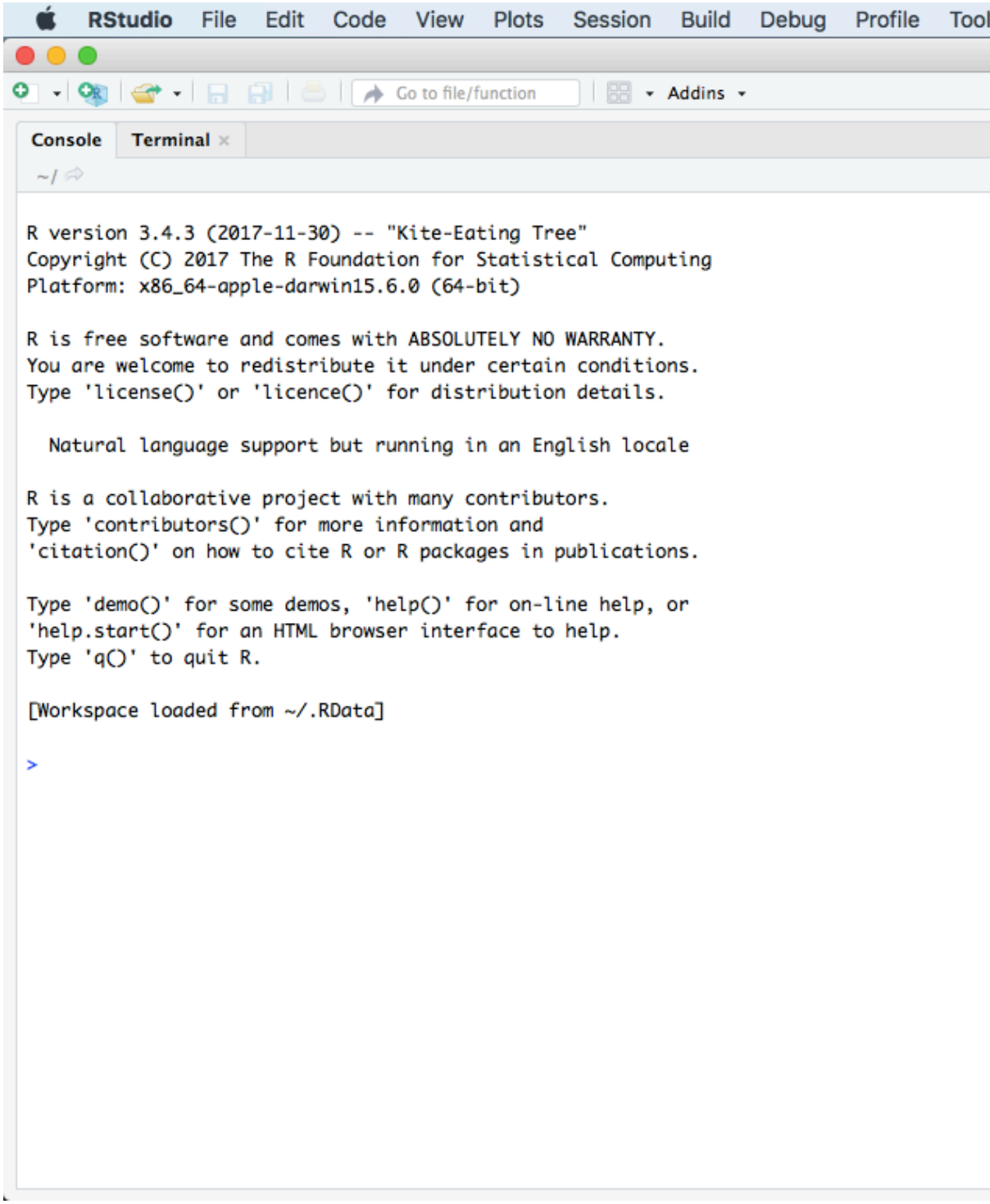
- It is free for students.
- The topics will remain the same but the text is going to be constantly tinkered with.
- What is the work of the author is copyright Matt Waite 2019.
- The text is Attribution-NonCommercial-ShareAlike 4.0 International Creative Commons licensed. That means you can share it and change it, but only if you share your changes with the same license and it cannot be used for commercial purposes. I'm not making money on this so you can't either.

- As such, the whole book – authored in Bookdown – is open sourced on Github. Pull requests welcomed!

Chapter 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.


RStudio File Edit Code View Plots Session Build Debug Profile Tool
Go to file/function
Addins
Console
Terminal
~/
R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
[Workspace loaded from ~/.RData]
>

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 run `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. 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')`

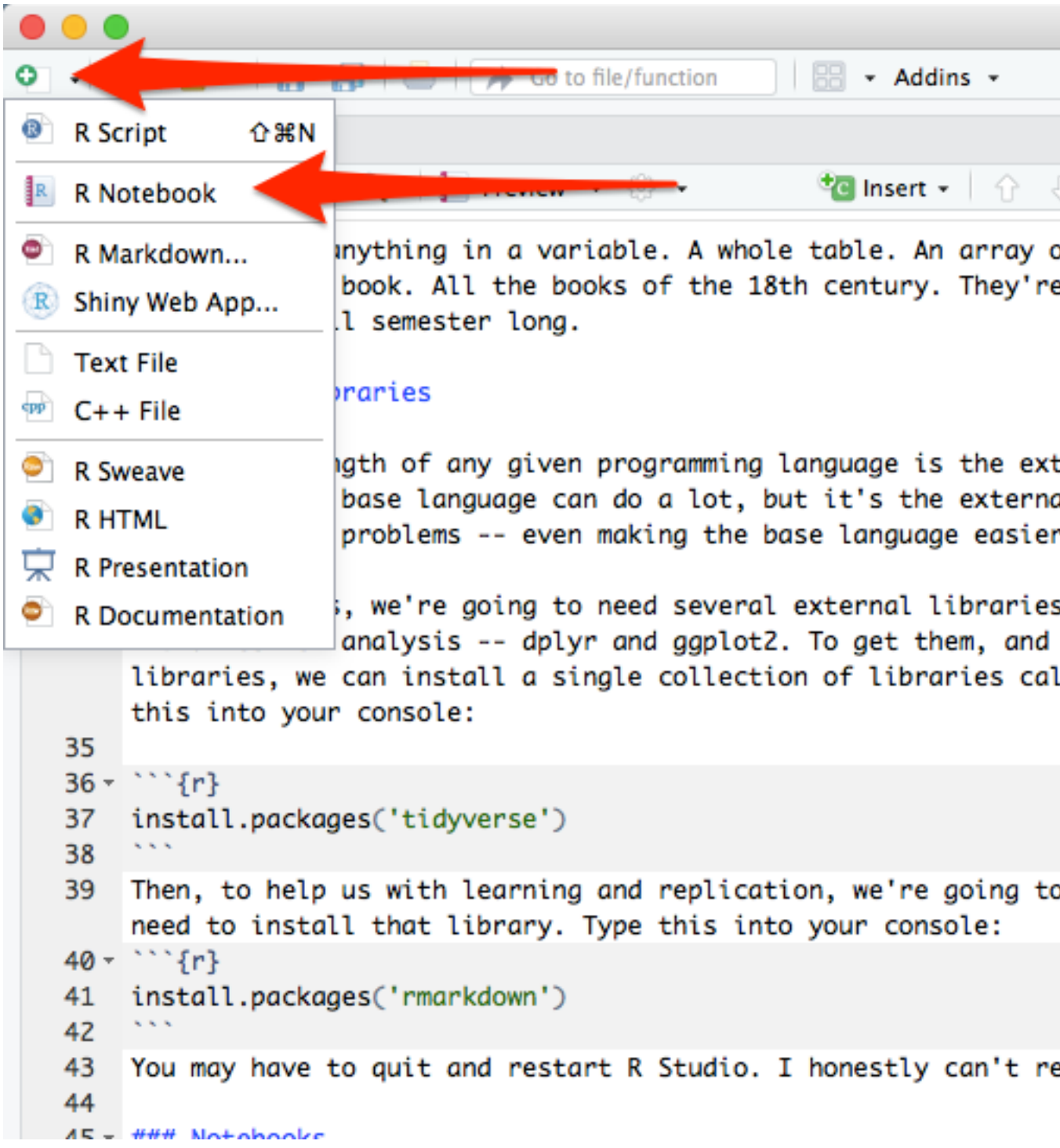
NOTE: This is a pattern. You should always install libraries in the console.

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.


Go to file/function
Addins
R Script ⇧⌘N
R Notebook
R Markdown...
Shiny Web App...
Text File
C++ File
R Sweave
R HTML
R Presentation
R Documentation
Insert
anything in a variable. A whole table. An array o
book. All the books of the 18th century. They're
l semester long.
raries
ngth of any given programming language is the ext
base language can do a lot, but it's the externa
problems -- even making the base language easier
, we're going to need several external libraries
analysis -- dplyr and ggplot2. To get them, and
libraries, we can install a single collection of libraries cal
this into your console:
35
36 ```{r}
37 install.packages('tidyverse')
38 ```
39 Then, to help us with learning and replication, we're going to
need to install that library. Type this into your console:
40 ```{r}
41 install.packages('rmarkdown')
42 ```
43 You may have to quit and restart R Studio. I honestly can't re
44
45 ### Notebooks

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.

Chapter 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	
1	2018-11-06		Mississippi Valley State	W	10
2	2018-11-11		Southeastern Louisiana	W	8
3	2018-11-14		Seton Hall	W	8
4	2018-11-19	N	Missouri State	W	8
5	2018-11-20	N	Texas Tech	L	5
6	2018-11-24		Western Illinois	W	7
7	2018-11-26	@	Clemson	W	6
8	2018-12-02		Rows	W	7
9	2018-12-05	@	Minnesota	L	7
10	2018-12-08		Creighton	W	9
11	2018-12-16	N	Oklahoma State	W	7
12	2018-12-22		Cal State Fullerton	W	8
13	2018-12-29		Southwest Minnesota State	W	7
14	2019-01-02	@	Maryland	L	7
15	2019-01-06	@	Iowa	L	8
16	2019-01-10		Penn State	W	7
17	2019-01-14	@	Indiana	W	6
18	2019-01-17		Michigan State	L	6
19	2019-01-21	@	Rutgers	L	6

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 Nebraska basketball's game logs, which you should open now.

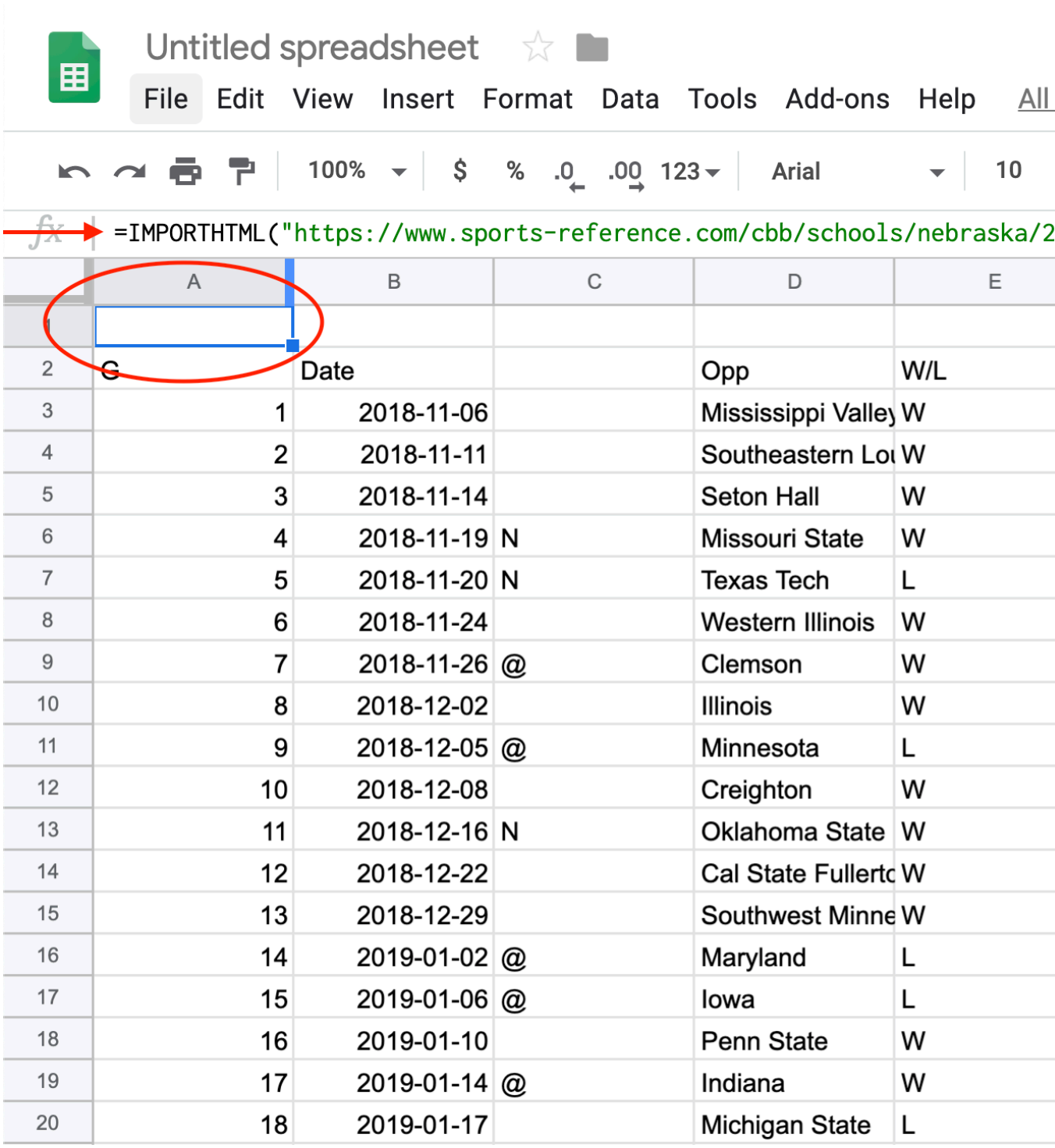
Now, in a new tab, log into Google Docs/Drive and open a new spreadsheet. In the first cell of the first row, copy and paste this formula in:

```
=IMPORTHTML("https://www.sports-reference.com/cbb/schools/nebraska/2019-gamelogs.html", "table",
```

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:

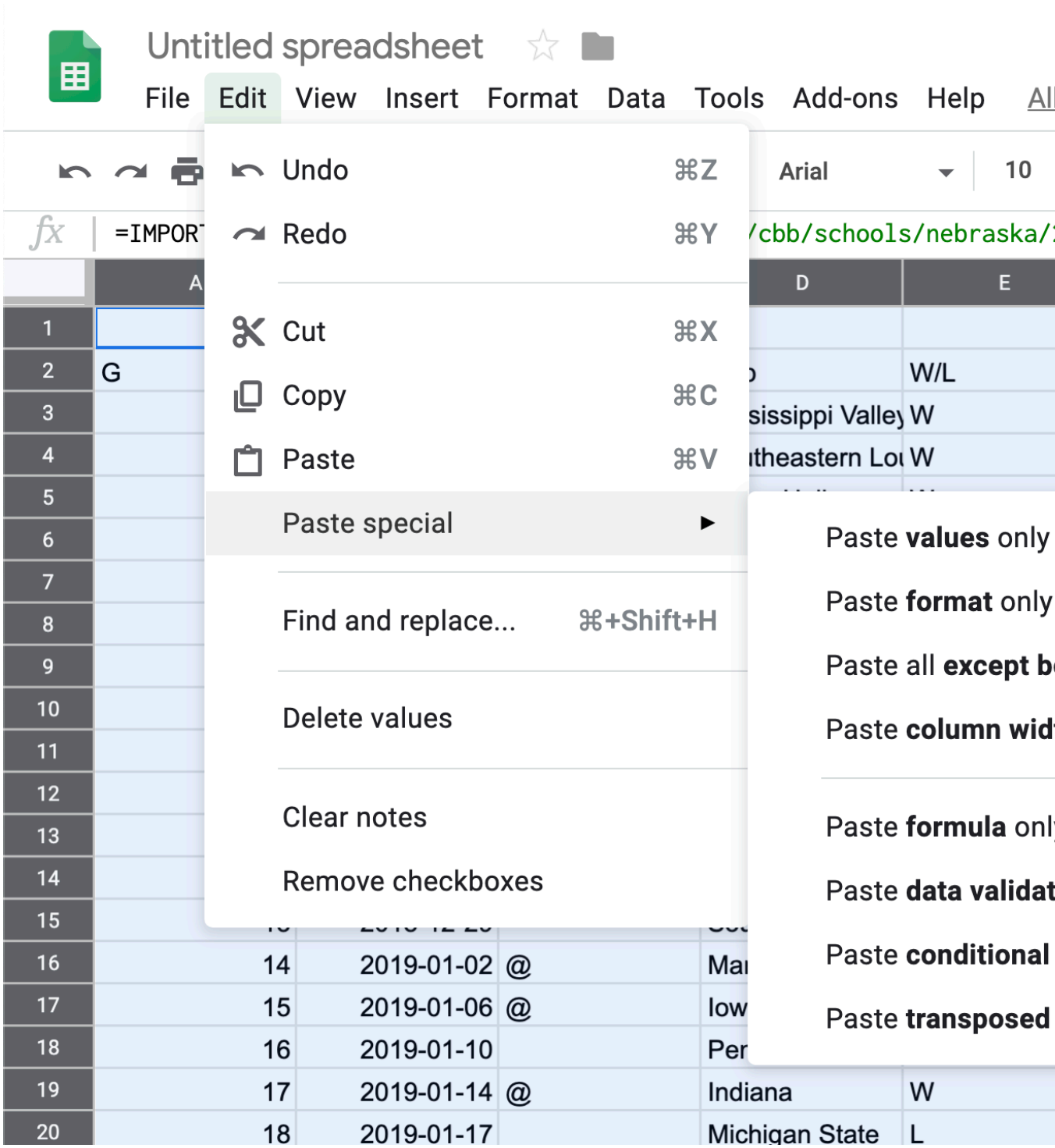


	A	B	C	D	E
1					
2	G	Date		Opp	W/L
3	1	2018-11-06		Mississippi Valley	W
4	2	2018-11-11		Southeastern Lou	W
5	3	2018-11-14		Seton Hall	W
6	4	2018-11-19	N	Missouri State	W
7	5	2018-11-20	N	Texas Tech	L
8	6	2018-11-24		Western Illinois	W
9	7	2018-11-26	@	Clemson	W
10	8	2018-12-02		Illinois	W
11	9	2018-12-05	@	Minnesota	L
12	10	2018-12-08		Creighton	W
13	11	2018-12-16	N	Oklahoma State	W
14	12	2018-12-22		Cal State Fullerto	W
15	13	2018-12-29		Southwest Minne	W
16	14	2019-01-02	@	Maryland	L
17	15	2019-01-06	@	Iowa	L
18	16	2019-01-10		Penn State	W
19	17	2019-01-14	@	Indiana	W
20	18	2019-01-17		Michigan State	L

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.

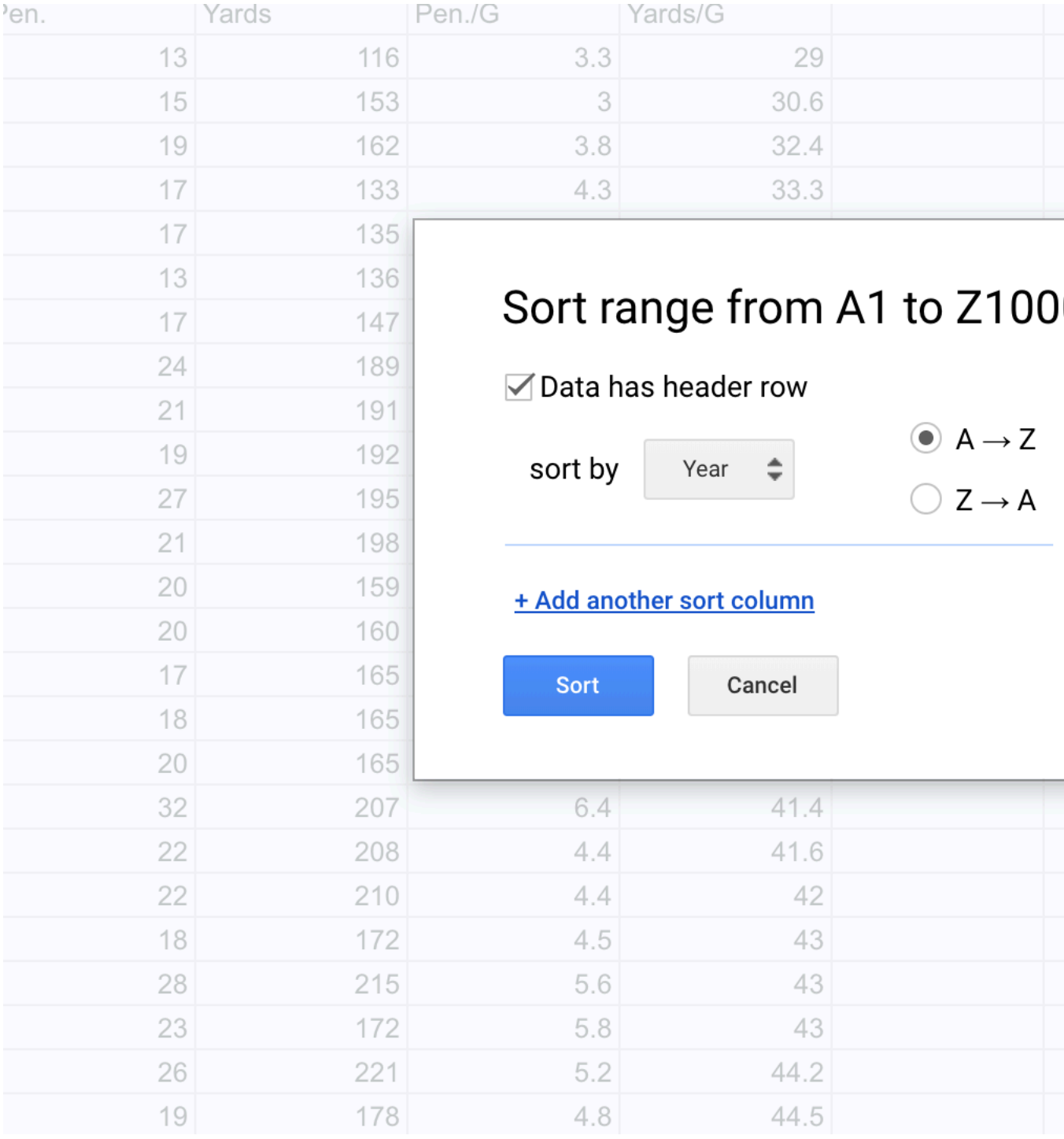

Untitled spreadsheet
File Edit View Insert Format Data Tools Add-ons Help
Undo ⌘Z
Redo ⌘Y
Cut ⌘X
Copy ⌘C
Paste ⌘V
Paste special
Find and replace... ⌘+Shift+H
Delete values
Clear notes
Remove checkboxes
Paste values only
Paste format only
Paste all except b
Paste column wid
Paste formula onl
Paste data validat
Paste conditional
Paste transposed
Arial
10
=IMPOR
/cbb/schools/nebraska/
W/L
sissippi Valley W
theastern Lou W
14 2019-01-02 @
15 2019-01-06 @
16 2019-01-10
17 2019-01-14 @ Indiana W
18 2019-01-17 Michigan State L

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.


Pen.
Yards
Pen./G
Yards/G
13 116 3.3 29
15 153 3 30.6
19 162 3.8 32.4
17 133 4.3 33.3
17 135
13 136
17 147
24 189
21 191
19 192
27 195
21 198
20 159
20 160
17 165
18 165
20 165
32 207 6.4 41.4
22 208 4.4 41.6
22 210 4.4 42
18 172 4.5 43
28 215 5.6 43
23 172 5.8 43
26 221 5.2 44.2
19 178 4.8 44.5
Sort range from A1 to Z100
Data has header row
sort by
Year
A → Z
Z → A
+ Add another sort column
Sort
Cancel

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.

Chapter 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 notebooks.

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 of advanced metrics for NBA players this season. You can download the CSV here. 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("~/Box/SportsData/nbaadvancedplayers1920.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 (nbaadvancedplayers1920.csv). In my case, I've got a folder called Box in my home directory (that's the ~ part), and in there is a folder called SportsData that has the file called nbaadvancedplayers1920.csv in

it. Some people – insane people – leave the data in their downloads folder. The data path then would be `~/Downloads/nameofthedatafilehere.csv` on PC or Mac.

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

```
nbaplayers <- read_csv("data/nbaadvancedplayers1920.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Player = col_character(),
##   Pos = col_character(),
##   Tm = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

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 `mountainlions` and run it.

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

```
head(nbaplayers)
```

```
## # A tibble: 6 x 27
##      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 Steve~ C        26 OKC      63  1680  20.5 0.604  0.006 0.421   14
## 2     2 Bam A~ PF       22 MIA      72  2417  20.3 0.598  0.018 0.484    8.5
## 3     3 LaMar~ C        34 SAS      53  1754  19.7 0.571  0.198 0.241    6.3
## 4     4 Kyle ~ PF       23 MIA       2    13   4.7 0.5    0     0       17.9
## 5     5 Nicke~ SG       21 NOP      47   591   8.9 0.473  0.5   0.139    1.6
## 6     6 Grays~ SG       24 MEM      38   718  12   0.609  0.562 0.179    1.2
## # ... with 15 more variables: `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>
```

```
nrow(nbaplayers)
```

```
## [1] 651
```

Another way to look at nrow – we have 651 players from this season in our dataset.

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

pile. We'd put them in groups and then count them.

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

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

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

Here's the code:

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

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 9 x 2
##   Pos   total
##   <chr> <int>
## 1 C       111
## 2 C-PF      2
## 3 PF      135
## 4 PF-C      2
## 5 PG      111
## 6 SF      113
## 7 SF-PF     4
## 8 SF-SG     3
## 9 SG      170
```

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

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

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

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

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

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 9 x 2
##   Pos   total
##   <chr> <int>
## 1 SG      170
## 2 PF      135
## 3 SF      113
## 4 C       111
## 5 PG      111
## 6 SF-PF     4
## 7 SF-SG     3
## 8 C-PF      2
## 9 PF-C      2
```

So the most common position in the NBA? Shooting guard, followed by power 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(Tm, Pos) %>%
  summarise(
    total = n()
  ) %>% arrange(desc(total))
```

```
## `summarise()` regrouping output by 'Tm' (override with `.groups` argument)
```

```
## # A tibble: 159 x 3
## # Groups:   Tm [31]
```

```
##    Tm    Pos   total
##    <chr> <chr> <int>
##  1 TOT   PF       13
##  2 TOT   SG       13
##  3 SAC   PF        9
##  4 TOT   SF        9
##  5 BRK   SG        8
##  6 LAL   SG        8
##  7 TOT   PG        8
##  8 ATL   SG        7
##  9 BRK   SF        7
## 10 DAL   SG        7
## # ... with 149 more rows
```

So wait, what team is TOT?

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

Sacramento has 9 power forward. Brooklyn has 8 shooting guards, as do the Lakers. You can learn a bit about how a team is assembled by looking at these simple counts.

4.2 Other aggregates: Mean and median

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

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

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

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

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 9 x 4
```

```
##   Pos   count mean_minutes median_minutes
##   <chr> <int>        <dbl>          <dbl>
## 1 C       111         891.           887
## 2 C-PF      2         316.           316.
## 3 PF      135         790.           567
## 4 PF-C      2        1548.          1548.
## 5 PG      111         944.           850
## 6 SF      113         877.           754
## 7 SF-PF     4         638.           286.
## 8 SF-SG     3        1211           1688
## 9 SG      170         843.           654.
```

So there's 651 players in the data. Let's look at shooting guards. The average shooting guard plays 842 minutes and the median is 653.5 minutes.

Why?

Let's let sort help us.

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

```
## # A tibble: 651 x 27
##       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   323 CJ Mc~ SG       28 POR      70  2556  17   0.541  0.378 0.136    1.9
##  2    55 Devin~ SG       23 PHO      70  2512  20.6 0.618  0.31  0.397    1.3
##  3   198 James~ SG       30 HOU      68  2483  29.1 0.626  0.557 0.528    2.9
##  4    27 Harri~ PF       27 SAC      72  2482  13.3 0.574  0.338 0.337    3.4
##  5   297 Damia~ PG       29 POR      66  2474  26.9 0.627  0.5   0.384    1.4
##  6   204 Tobia~ PF       27 PHI      72  2469  17.2 0.556  0.304 0.184    3.1
##  7   479 P.J. ~ PF       34 HOU      72  2467   8.3 0.559  0.702 0.113    4.7
##  8   175 Shai ~ SG       21 OKC      70  2428  17.7 0.568  0.247 0.352    2.2
##  9     2 Bam A~ PF       22 MIA      72  2417  20.3 0.598  0.018 0.484    8.5
## 10   343 Donov~ SG       23 UTA      69  2364  18.8 0.558  0.352 0.24     2.6
## # ... with 641 more rows, and 15 more variables: `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>
```

The player with the most minutes on the floor is a shooting guard. Shooting guard is the most common position, so that means there's CJ McCollum rolling up 2,556 minutes in a season, and then there's Cleveland Cavalier's sensation J.P. Macura. Never heard of J.P. Macura? Might be because he logged one 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))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 9 x 6
##   Pos    total avgminutes minminutes maxminutes stdev
##   <chr>  <dbl>      <dbl>      <dbl>      <dbl> <dbl>
## 1 SG    143229       843.          1       2556 735.
## 2 PF    106654       790.          5       2482 719.
## 3 PG    104745       944.          8       2474 727.
## 4 SF     99109       877.         11       2316 709.
## 5 C      98914       891.          3       2336 619.
## 6 SF-SG   3633      1211          87       1858 977.
## 7 PF-C    3097      1548.        960       2137 832.
## 8 SF-PF   2553       638.         46       1936 873.
## 9 C-PF     633       316.        256        377  85.6
```

So again, no surprise, shooting guards spend the most minutes on the floor in the NBA. They average 842 minutes, but we noted why that's trouble. The minimum is the J.P. Macura Award, max is the Trailblazer's 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.

Chapter 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 we'll import a common and simple dataset of total attendance at NCAA football games over the last few seasons.

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

```
## Parsed with column specification:
## cols(
##   Institution = col_character(),
##   Conference = col_character(),
##   `2013` = col_double(),
##   `2014` = col_double(),
##   `2015` = col_double(),
##   `2016` = col_double(),
##   `2017` = col_double(),
##   `2018` = col_double()
## )
```

```
head(attendance)
```

```
## # A tibble: 6 x 8
##   Institution     Conference      `2013` `2014` `2015` `2016` `2017` `2018`
##   <chr>           <chr>            <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
## 1 Air Force       MWC             228562 168967 156158 177519 174924 166205
## 2 Akron           MAC             107101  55019 108588  62021 117416  92575
## 3 Alabama         SEC             710538 710736 707786 712747 712053 710931
## 4 Appalachian St. FBS Independent 149366     NA     NA     NA     NA     NA
## 5 Appalachian St. Sun Belt            NA 138995 128755 156916 154722 131716
## 6 Arizona         Pac-12          285713 354973 308355 338017 255791 318051
```

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 '2018' as the new data, and we'll use '2017' as the old data. So we're looking at one year. Then, to help us, we'll use arrange again to sort it, so we get the fastest growing school over one year.

```
attendance %>% mutate(
  change = (`2018` - `2017`)/`2017`
)
```

```
## # A tibble: 150 x 9
##    Institution   Conference   `2013` `2014` `2015` `2016` `2017` `2018`   change
##    <chr>         <chr>         <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>    <dbl>
##  1 Air Force     MWC          228562 168967 156158 177519 174924 166205 -0.0498
##  2 Akron         MAC          107101  55019 108588  62021 117416  92575 -0.212
##  3 Alabama       SEC          710538 710736 707786 712747 712053 710931 -0.00158
##  4 Appalachian ~ FBS Indepen~ 149366     NA     NA     NA     NA     NA NA
##  5 Appalachian ~ Sun Belt         NA 138995 128755 156916 154722 131716 -0.149
##  6 Arizona       Pac-12       285713 354973 308355 338017 255791 318051  0.243
##  7 Arizona St.   Pac-12       501509 343073 368985 286417 359660 291091 -0.191
##  8 Arkansas      SEC          431174 399124 471279 487067 442569 367748 -0.169
##  9 Arkansas St.  Sun Belt     149477 149163 138043 136200 119538 119001 -0.00449
## 10 Army West Po~ FBS Indepen~ 169781 171310 185946 163267 185543 190156  0.0249
## # ... with 140 more rows
```

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 = ((`2018` - `2017`)/`2017`)*100
)
```

```
## # A tibble: 150 x 9
##    Institution    Conference   `2013` `2014` `2015` `2016` `2017` `2018`  change
##    <chr>          <chr>         <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>   <dbl>
##  1 Air Force      MWC          228562 168967 156158 177519 174924 166205  -4.98
##  2 Akron          MAC          107101  55019 108588  62021 117416  92575 -21.2
##  3 Alabama        SEC          710538 710736 707786 712747 712053 710931  -0.158
##  4 Appalachian S~ FBS Indepen~ 149366     NA     NA     NA     NA     NA  NA
##  5 Appalachian S~ Sun Belt         NA 138995 128755 156916 154722 131716 -14.9
##  6 Arizona        Pac-12       285713 354973 308355 338017 255791 318051  24.3
##  7 Arizona St.    Pac-12       501509 343073 368985 286417 359660 291091 -19.1
##  8 Arkansas       SEC          431174 399124 471279 487067 442569 367748 -16.9
##  9 Arkansas St.   Sun Belt      149477 149163 138043 136200 119538 119001  -0.449
## 10 Army West Poi~ FBS Indepen~ 169781 171310 185946 163267 185543 190156   2.49
## # ... with 140 more rows
```

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

```
attendance %>% mutate(
  change = ((`2018` - `2017`)/`2017`)*100
) %>% arrange(desc(change))
```

```
## # A tibble: 150 x 9
##    Institution   Conference `2013` `2014` `2015` `2016` `2017` `2018` change
##    <chr>         <chr>       <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
##  1 Ga. Southern  Sun Belt       NA 105510 124681 104095  61031 100814   65.2
##  2 La.-Monroe    Sun Belt    85177  90540  58659  67057  49640  71048   43.1
##  3 Louisiana     Sun Belt   129878 154652 129577 121346  78754 111303   41.3
##  4 Hawaii        MWC        185931 192159 164031 170299 145463 205455   41.2
##  5 Buffalo       MAC        136418 122418 110743 104957  80102 110280   37.7
##  6 California    Pac-12     345303 286051 292797 279769 219290 300061   36.8
##  7 UCF           AAC        252505 226869 180388 214814 257924 352148   36.5
##  8 UTSA          C-USA      175282 165458 138048 138226 114104 148257   29.9
##  9 Eastern Mich. MAC         20255  75127  29381 106064  73649  95632   29.8
## 10 Louisville    ACC            NA 317829 294413 324391 276957 351755   27.0
## # ... with 140 more rows
```

So who had the most growth last year from the year before? Something going on at Georgia Southern.

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 2018-19 season. It's a dataset of 5,386 players, and we've got 59 variables – one of them is True Shooting Percentage, but we're going to ignore that.

```
players <- read_csv("data/players19.csv")
```

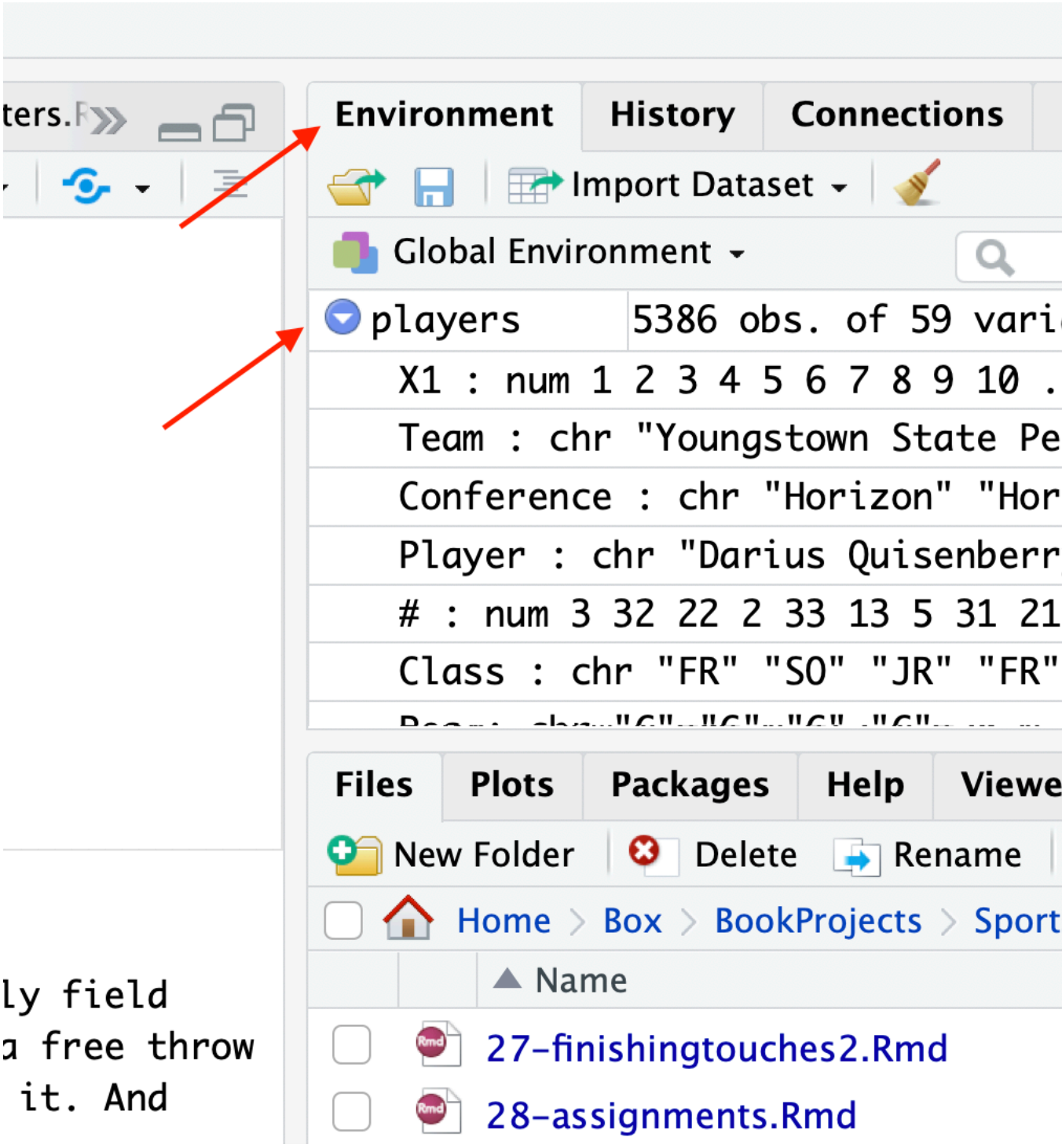
```
## Warning: Missing column names filled in: 'X1' [1]
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Team = col_character(),
##   Conference = col_character(),
##   Player = col_character(),
##   Class = col_character(),
##   Pos = col_character(),
##   Height = col_character(),
##   Hometown = col_character(),
##   `High School` = col_character(),
##   Summary = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

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.


Environment
History
Connections
Import Dataset
Global Environment
players
5386 obs. of 59 vari
X1 : num 1 2 3 4 5 6 7 8 9 10 .
Team : chr "Youngstown State Pe
Conference : chr "Horizon" "Hor
Player : chr "Darius Quisenberr
: num 3 32 22 2 33 13 5 31 21
Class : chr "FR" "SO" "JR" "FR"
Files
Plots
Packages
Help
Viewe
New Folder
Delete
Rename
Home
Box
BookProjects
Sport
Name
27-finishingtouches2.Rmd
28-assignments.Rmd
ly field
a free throw
it. And

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: 5,386 x 60
##       X1 Team  Conference Player   `#` Class Pos   Height Weight Hometown
##    <dbl> <chr> <chr>      <chr>  <dbl> <chr> <chr> <chr>   <dbl> <chr>
##  1   579 Texa~ Big 12     Drayt~     4 JR    G     6-0       156 Austin,~
##  2   843 Ston~ AEC        Nick ~    42 FR    F     6-7       240 Port Je~
##  3  1059 Sout~ Southland  Patri~    22 SO    F     6-3       210 Folsom,~
##  4  4269 Dayt~ A-10       Camro~    52 SO    G     5-7       160 Country~
##  5  4681 Cali~ Pac-12     David~    21 JR    G     6-4       185 Newbury~
##  6   326 Virg~ ACC        Grant~     1 FR    G     <NA>       NA Charlot~
##  7   410 Vand~ SEC        Mac H~    42 FR    G     6-6       182 Chattan~
##  8  1390 Sain~ A-10       Jack ~    31 JR    G     6-6       205 Mattoon~
##  9  2230 NJIT~ A-Sun      Patri~     3 SO    G     5-9       160 West Or~
## 10   266 Wash~ Pac-12     Reaga~    34 FR    F     6-6       225 Santa A~
## # ... with 5,376 more rows, and 50 more variables: `High School` <chr>,
## #   Summary <chr>, Rk.x <dbl>, G <dbl>, GS <dbl>, MP <dbl>, FG <dbl>,
## #   FGA <dbl>, `FG%` <dbl>, `2P` <dbl>, `2PA` <dbl>, `2P%` <dbl>, `3P` <dbl>,
## #   `3PA` <dbl>, `3P%` <dbl>, FT <dbl>, FTA <dbl>, `FT%` <dbl>, ORB <dbl>,
## #   DRB <dbl>, TRB <dbl>, AST <dbl>, STL <dbl>, BLK <dbl>, TOV <dbl>, PF <dbl>,
## #   PTS <dbl>, Rk.y <dbl>, PER <dbl>, `TS%` <dbl>, `eFG%` <dbl>, `3PAr` <dbl>,
## #   FTr <dbl>, PProd <dbl>, `ORB%` <dbl>, `DRB%` <dbl>, `TRB%` <dbl>,
## #   `AST%` <dbl>, `STL%` <dbl>, `BLK%` <dbl>, `TOV%` <dbl>, `USG%` <dbl>,
## #   OWS <dbl>, DWS <dbl>, WS <dbl>, `WS/40` <dbl>, OBPM <dbl>, DBPM <dbl>,
## #   BPM <dbl>, trueshooting <dbl>
```

You'll be forgiven if you did not hear about Texas Longhorns shooting sensation Drayton Whiteside. He played in six games, took one shot and actually hit it. It happened to be a three pointer, which is one more three pointer than I've hit in college basketball. So props to him. Does that mean he had the best true shooting season in college basketball last year? Not hardly.

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

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 "Nebraska". 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.

```
library(tidyverse)
```

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

```
## Parsed with column specification:
## cols(
##   Institution = col_character(),
##   Conference = col_character(),
##   `2013` = col_double(),
##   `2014` = col_double(),
##   `2015` = col_double(),
##   `2016` = col_double(),
##   `2017` = col_double(),
##   `2018` = col_double()
## )
```

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 Nebraska U. 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 Nebraska, you do this:

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

```
## # A tibble: 1 x 8
##   Institution Conference `2013` `2014` `2015` `2016` `2017` `2018`
##   <chr>       <chr>       <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
## 1 Nebraska    Big Ten    727466 638744 629983 631402 628583 623240
```

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

```
attendance %>% filter(`2018` >= 500000)
```

```
## # A tibble: 17 x 8
##    Institution    Conference `2013` `2014` `2015` `2016` `2017` `2018`
##    <chr>          <chr>       <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
##  1 Alabama        SEC        710538 710736 707786 712747 712053 710931
##  2 Auburn         SEC        685252 612157 612157 695498 605120 591236
##  3 Clemson        ACC        574333 572262 588266 566787 565412 562799
##  4 Florida        SEC        524638 515001 630457 439229 520290 576299
##  5 Georgia        SEC        556476 649222 649222 556476 556476 649222
##  6 LSU            SEC        639927 712063 654084 708618 591034 705733
##  7 Michigan       Big Ten    781144 734364 771174 883741 669534 775156
##  8 Michigan St.   Big Ten    506294 522765 522628 522666 507398 508088
##  9 Nebraska       Big Ten    727466 638744 629983 631402 628583 623240
## 10 Ohio St.       Big Ten    734528 744075 750705 750944 752464 713630
## 11 Oklahoma       Big 12     508334 510972 512139 521142 519119 607146
## 12 Penn St.       Big Ten    676112 711358 698590 701800 746946 738396
## 13 South Carolina SEC        576805 569664 472934 538441 550099 515396
## 14 Tennessee      SEC        669087 698276 704088 706776 670454 650887
## 15 Texas          Big 12     593857 564618 540210 587283 556667 586277
## 16 Texas A&M      SEC        697003 630735 725354 713418 691612 698908
## 17 Wisconsin      Big Ten    552378 556642 546099 476144 551766 540072
```

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" | Cor
```

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

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

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

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

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

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

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

```
## # A tibble: 65 x 8
##    Institution    Conference `2013` `2014` `2015` `2016` `2017` `2018`
##    <chr>          <chr>       <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
##  1 Alabama        SEC        710538 710736 707786 712747 712053 710931
##  2 Arizona        Pac-12     285713 354973 308355 338017 255791 318051
##  3 Arizona St.    Pac-12     501509 343073 368985 286417 359660 291091
##  4 Arkansas       SEC        431174 399124 471279 487067 442569 367748
##  5 Auburn         SEC        685252 612157 612157 695498 605120 591236
##  6 Baylor         Big 12     321639 280257 276960 275029 262978 248017
##  7 Boston College ACC        198035 239893 211433 192942 215546 263363
##  8 California     Pac-12     345303 286051 292797 279769 219290 300061
##  9 Clemson        ACC        574333 572262 588266 566787 565412 562799
## 10 Colorado       Pac-12     230778 226670 236331 279652 282335 274852
## # ... with 55 more rows
```

6.1 Selecting data to make it easier to read

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

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

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

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

6.2 Using conditional filters to set limits

Let's return to the blistering season of Drayton Whiteside using our dataset of every college basketball player's season stats in 2018-19 season. How can we set limits in something like a question of who had the best season? Let's get our Drayton Whiteside data from the previous chapter back up.

```
players <- read_csv("data/players19.csv")
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Team = col_character(),
##   Conference = col_character(),
##   Player = col_character(),
##   Class = col_character(),
##   Pos = col_character(),
##   Height = col_character(),
##   Hometown = col_character(),
```

```
##   `High School` = col_character(),
##   Summary = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

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

```
## # A tibble: 5,386 x 60
##       X1 Team  Conference Player   `#` Class Pos   Height Weight Hometown
##    <dbl> <chr> <chr>      <chr>  <dbl> <chr> <chr> <chr>   <dbl> <chr>
##  1   579 Texa~ Big 12     Drayt~     4 JR    G     6-0       156 Austin,~
##  2   843 Ston~ AEC        Nick ~    42 FR    F     6-7       240 Port Je~
##  3  1059 Sout~ Southland  Patri~    22 SO    F     6-3       210 Folsom,~
##  4  4269 Dayt~ A-10       Camro~    52 SO    G     5-7       160 Country~
##  5  4681 Cali~ Pac-12     David~    21 JR    G     6-4       185 Newbury~
##  6   326 Virg~ ACC        Grant~     1 FR    G     <NA>       NA Charlot~
##  7   410 Vand~ SEC        Mac H~    42 FR    G     6-6       182 Chattan~
##  8  1390 Sain~ A-10       Jack ~    31 JR    G     6-6       205 Mattoon~
##  9  2230 NJIT~ A-Sun      Patri~     3 SO    G     5-9       160 West Or~
## 10   266 Wash~ Pac-12     Reaga~    34 FR    F     6-6       225 Santa A~
## # ... with 5,376 more rows, and 50 more variables: `High School` <chr>,
## #   Summary <chr>, Rk.x <dbl>, G <dbl>, GS <dbl>, MP <dbl>, FG <dbl>,
## #   FGA <dbl>, `FG%` <dbl>, `2P` <dbl>, `2PA` <dbl>, `2P%` <dbl>, `3P` <dbl>,
## #   `3PA` <dbl>, `3P%` <dbl>, FT <dbl>, FTA <dbl>, `FT%` <dbl>, ORB <dbl>,
## #   DRB <dbl>, TRB <dbl>, AST <dbl>, STL <dbl>, BLK <dbl>, TOV <dbl>, PF <dbl>,
## #   PTS <dbl>, Rk.y <dbl>, PER <dbl>, `TS%` <dbl>, `eFG%` <dbl>, `3PAr` <dbl>,
## #   FTr <dbl>, PProd <dbl>, `ORB%` <dbl>, `DRB%` <dbl>, `TRB%` <dbl>,
## #   `AST%` <dbl>, `STL%` <dbl>, `BLK%` <dbl>, `TOV%` <dbl>, `USG%` <dbl>,
## #   OWS <dbl>, DWS <dbl>, WS <dbl>, `WS/40` <dbl>, OBPM <dbl>, DBPM <dbl>,
## #   BPM <dbl>, trueshooting <dbl>
```

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

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

```
## # A tibble: 3,659 x 60
##       X1 Team  Conference Player   `#` Class Pos   Height Weight Hometown
##    <dbl> <chr> <chr>      <chr>  <dbl> <chr> <chr> <chr>   <dbl> <chr>
```

```
##  1  4634 Cent~ Southland  Jorda~    33 JR    G     6-1       185 Harriso~
##  2  3623 Hart~ AEC        Max T~    20 SR    G     6-5       200 Rye, NY
##  3  2675 Mich~ Big Ten    Thoma~    15 FR    F     6-8       225 Clarkst~
##  4  5175 Litt~ Sun Belt   Kris ~    32 SO    F     6-8       194 Dewitt,~
##  5  5205 Ariz~ Pac-12     De'Qu~    32 SR    F     6-10      225 St. Tho~
##  6  4099 ETSU~ Southern   Lucas~    25 JR    C     7-0       220 De Lier~
##  7  3006 Loui~ Sun Belt   Brand~     0 SR    G     6-4       180 Hawthor~
##  8   570 Texa~ Big 12     Jaxso~    10 FR    F     6-11      220 Lovelan~
##  9  1704 Pepp~ WCC        Victo~    34 FR    C     6-9       200 Owerri,~
## 10  4056 East~ MAC        Jalen~    30 SO    F     6-9       215 Pasco, ~
## # ... with 3,649 more rows, and 50 more variables: `High School` <chr>,
## #   Summary <chr>, Rk.x <dbl>, G <dbl>, GS <dbl>, MP <dbl>, FG <dbl>,
## #   FGA <dbl>, `FG%` <dbl>, `2P` <dbl>, `2PA` <dbl>, `2P%` <dbl>, `3P` <dbl>,
## #   `3PA` <dbl>, `3P%` <dbl>, FT <dbl>, FTA <dbl>, `FT%` <dbl>, ORB <dbl>,
## #   DRB <dbl>, TRB <dbl>, AST <dbl>, STL <dbl>, BLK <dbl>, TOV <dbl>, PF <dbl>,
## #   PTS <dbl>, Rk.y <dbl>, PER <dbl>, `TS%` <dbl>, `eFG%` <dbl>, `3PAr` <dbl>,
## #   FTr <dbl>, PProd <dbl>, `ORB%` <dbl>, `DRB%` <dbl>, `TRB%` <dbl>,
## #   `AST%` <dbl>, `STL%` <dbl>, `BLK%` <dbl>, `TOV%` <dbl>, `USG%` <dbl>,
## #   OWS <dbl>, DWS <dbl>, WS <dbl>, `WS/40` <dbl>, OBPM <dbl>, DBPM <dbl>,
## #   BPM <dbl>, trueshooting <dbl>
```

Now you get Central Arkansas Bears Junior Jordan Grant, who played in 25 games and was on the floor for 152 minutes. So he played regularly. But in that time, he only attempted 16 shots, and made 68 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, `
```

```
## Selecting by 2018
```

```
## # A tibble: 10 x 3
```

```
##    Institution Conference `2018`
##    <chr>       <chr>       <dbl>
##  1 Michigan    Big Ten    775156
##  2 Penn St.    Big Ten    738396
##  3 Ohio St.    Big Ten    713630
##  4 Alabama     SEC        710931
##  5 LSU         SEC        705733
##  6 Texas A&M   SEC        698908
##  7 Tennessee   SEC        650887
##  8 Georgia     SEC        649222
##  9 Nebraska    Big Ten    623240
## 10 Oklahoma    Big 12     607146
```

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

Chapter 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. Nebraska, Population, 1,929,000. That's one row. Then the next row is Nebraska, Average Life Expectancy, 76. 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')
```

```
## Parsed with column specification:
## cols(
##   Institution = col_character(),
##   Conference = col_character(),
##   `2013` = col_double(),
##   `2014` = col_double(),
```

```
##   `2015` = col_double(),
##   `2016` = col_double(),
##   `2017` = col_double(),
##   `2018` = col_double()
## )
```

```
attendance
```

```
## # A tibble: 150 x 8
##    Institution     Conference      `2013` `2014` `2015` `2016` `2017` `2018`
##    <chr>           <chr>            <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
##  1 Air Force       MWC             228562 168967 156158 177519 174924 166205
##  2 Akron           MAC             107101  55019 108588  62021 117416  92575
##  3 Alabama         SEC             710538 710736 707786 712747 712053 710931
##  4 Appalachian St. FBS Independent 149366     NA     NA     NA     NA     NA
##  5 Appalachian St. Sun Belt            NA 138995 128755 156916 154722 131716
##  6 Arizona         Pac-12          285713 354973 308355 338017 255791 318051
##  7 Arizona St.     Pac-12          501509 343073 368985 286417 359660 291091
##  8 Arkansas        SEC             431174 399124 471279 487067 442569 367748
##  9 Arkansas St.    Sun Belt        149477 149163 138043 136200 119538 119001
## 10 Army West Point FBS Independent 169781 171310 185946 163267 185543 190156
## # ... with 140 more rows
```

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

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

```
## # A tibble: 900 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 Akron       MAC        2013      107101
```

```
##  8 Akron       MAC        2014       55019
##  9 Akron       MAC        2015      108588
## 10 Akron       MAC        2016       62021
## # ... with 890 more rows
```

We've gone from 150 rows to 900, but that's expected when we have 6 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: 900 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 Akron       MAC        2013      107101
##  8 Akron       MAC        2014       55019
##  9 Akron       MAC        2015      108588
## 10 Akron       MAC        2016       62021
## # ... with 890 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)
```

```
## # A tibble: 150 x 8
##    Institution     Conference      `2013` `2014` `2015` `2016` `2017` `2018`
##    <chr>           <chr>            <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
```

```
##  1 Air Force       MWC             228562 168967 156158 177519 174924 166205
##  2 Akron           MAC             107101  55019 108588  62021 117416  92575
##  3 Alabama         SEC             710538 710736 707786 712747 712053 710931
##  4 Appalachian St. FBS Independent 149366     NA     NA     NA     NA     NA
##  5 Appalachian St. Sun Belt            NA 138995 128755 156916 154722 131716
##  6 Arizona         Pac-12          285713 354973 308355 338017 255791 318051
##  7 Arizona St.     Pac-12          501509 343073 368985 286417 359660 291091
##  8 Arkansas        SEC             431174 399124 471279 487067 442569 367748
##  9 Arkansas St.    Sun Belt        149477 149163 138043 136200 119538 119001
## 10 Army West Point FBS Independent 169781 171310 185946 163267 185543 190156
## # ... with 140 more rows
```

And just like that, we're back.

7.2 Why this matters

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

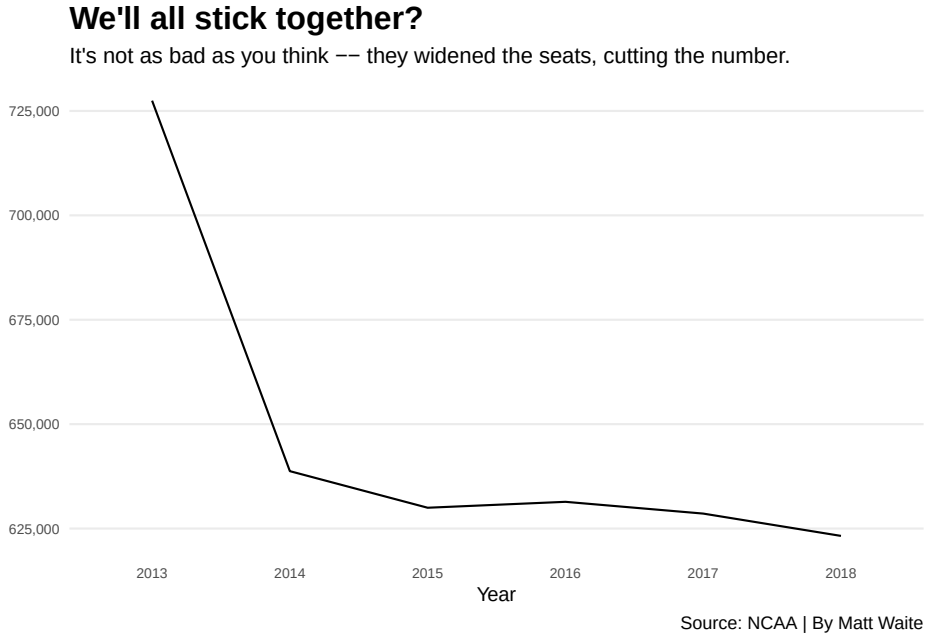
So let me walk you through an example using this data.

Let's look at Nebraska'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.

```
nebraska <- longdata %>% filter(Institution == "Nebraska")
```

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

```
ggplot(nebraska, aes(x=Year, y=Attendance, group=1)) +
  geom_line() +
  scale_y_continuous(labels = scales::comma) +
  labs(x="Year", y="Attendance", title="We'll all stick together?", subtitle="It's not
  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"
  )
```


We'll all stick together?
It's not as bad as you think −− they widened the seats, cutting the number.
725,000
700,000
675,000
650,000
625,000
2013
2014
2015
2016
2017
2018
Year
Source: NCAA | By Matt Waite

Chapter 8

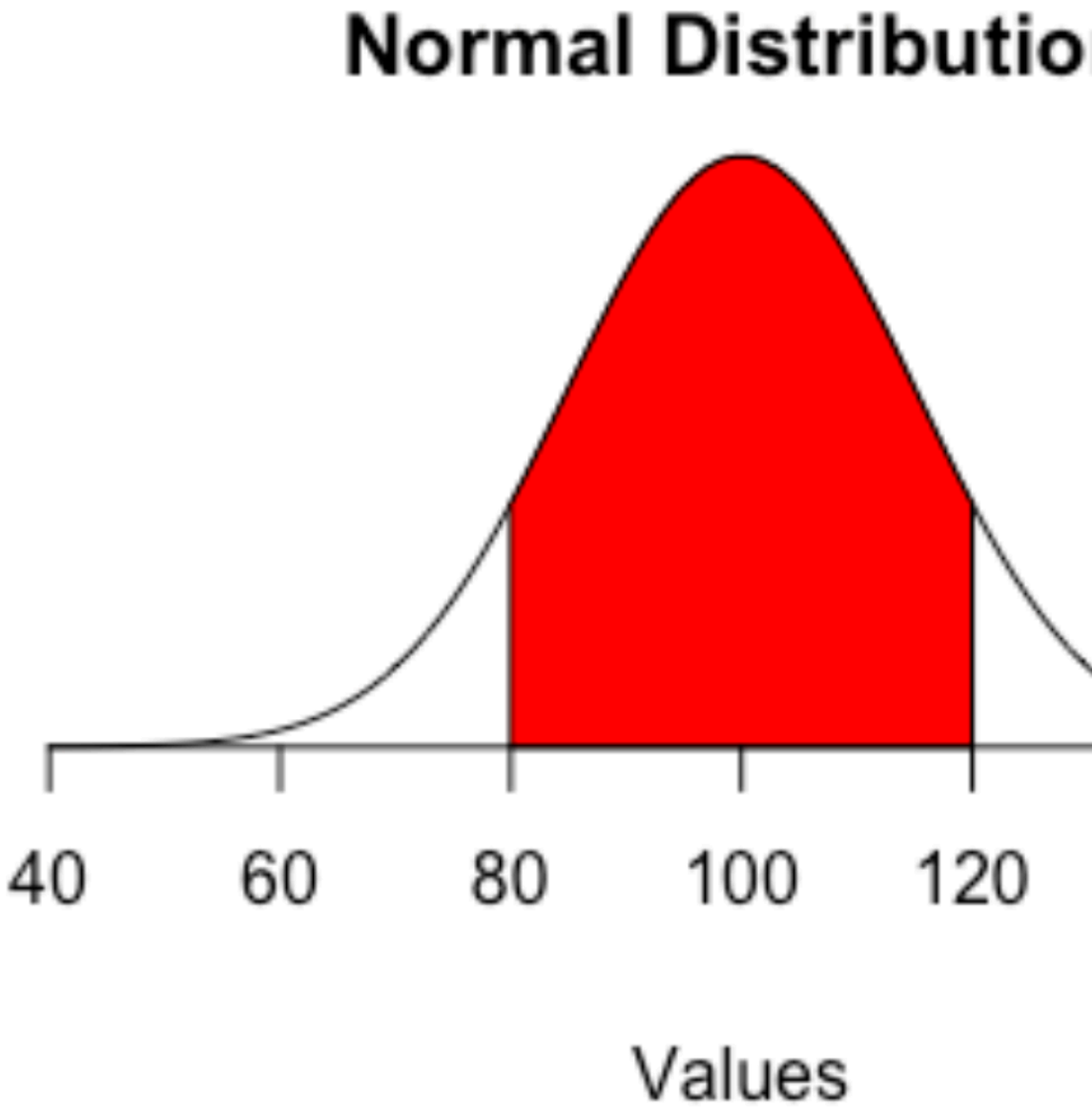
Significance tests

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. Here's a log of every game in this NBA season. There's a field called COVID, which labels the game as a regular game or a bubble game.

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

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Season = col_character(),
##   Conference = col_character(),
##   Team = col_character(),
##   Date = col_date(format = ""),
##   HomeAway = col_character(),
##   Opponent = col_character(),
##   W_L = col_character(),
##   COVID = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

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

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

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

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

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

```
##
##  Welch Two Sample t-test
##
## data:  totalpoints by COVID
## t = -5.232, df = 206.88, p-value = 4.099e-07
## alternative hypothesis: true difference in means is not equal to 0
## 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 - Oppor
```

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 is not equal to 0
## 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.

Chapter 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 libraries and data.

```
library(tidyverse)
```

```
correlations <- read_csv("data/footballlogs19.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = ""),
##   HomeAway = col_character(),
##   Opponent = col_character(),
##   Result = col_character(),
##   TeamFull = col_character(),
##   TeamURL = col_character(),
##   Outcome = col_character(),
##   Team = col_character(),
##   Conference = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

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

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

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

But is there one?

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

```
newcorrelations <- correlations %>%
  mutate(differential = TeamScore - OpponentScore, TotalPenalties = Penalties+DefPenalt
```

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

```
newcorrelations %>% summarise(correlation = cor(differential, TotalPenaltyYards, metho
```

```
## # A tibble: 1 x 1
##   correlation
##         <dbl>
## 1     -0.0137
```

So on a scale of -1 to 1, where 0 means there's no relationship at all and 1 or -1 means a perfect relationship, penalty yards and whether or not the team scores more points than it give up are at -0.014. You could say they're 1.4 percent related toward the negative – more penalties, the lower your differential. Another way to say it? They're 98.6 percent not related.

What about the number of penalties instead of the yards?

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

```
## # A tibble: 1 x 1
##   correlation
##         <dbl>
## 1    -0.00875
```

So wait, what does this all mean?

It means that when you look at every game in college football, the number of penalties and penalty yards does have a negative impact on the score difference between your team and the other team. But the relationship between penalties, penalty yards and the difference between scores is barely anything at all. Like 99 percent not related.

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

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

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

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

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

```
##
## Call:
## lm(formula = differential ~ TotalPenaltyYards, data = newcorrelations)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -73.13 -15.16   0.71  15.61  76.86
##
## Coefficients:
```

```
##                     Estimate Std. Error t value Pr(>|t|)
## (Intercept)         2.785946   1.525993   1.826   0.0681 .
## TotalPenaltyYards -0.007407   0.013244  -0.559   0.5760
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 23.3 on 1660 degrees of freedom
## Multiple R-squared:  0.0001884,Adjusted R-squared:  -0.0004139
## F-statistic: 0.3128 on 1 and 1660 DF,  p-value: 0.576
```

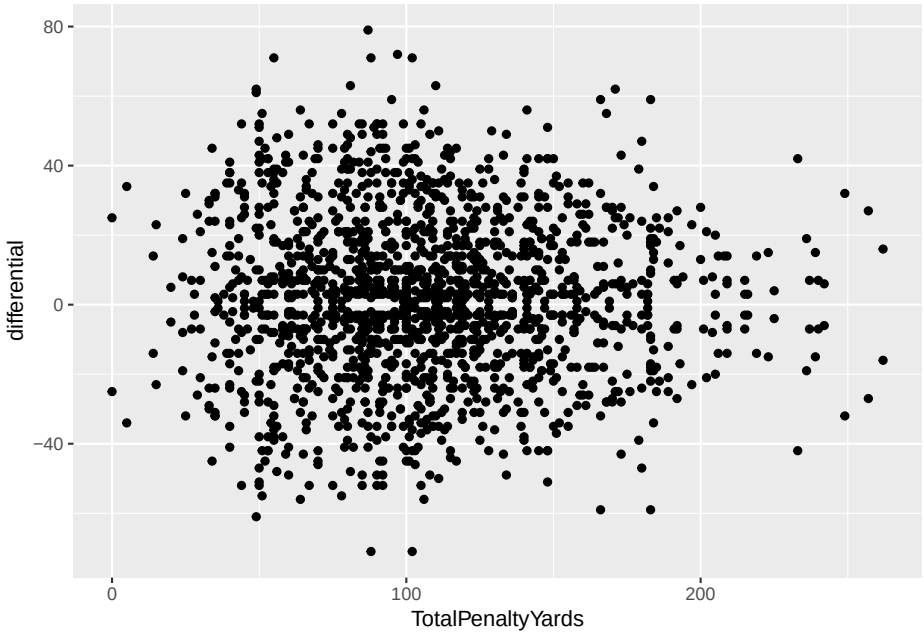
There's three things we need here:

1. First we want to look at the p-value. It's at the bottom right corner of the output. In the case of Total Penalty Yards, the p-value is .576. The threshold we're looking for here is .05. If it's less than .05, then the relationship is considered to be *statistically significant*. Significance here does not mean it's a big deal. It means it's not random. That's it. Just that. Not random. So in our case, the relationship between total penalty yards and a team's aggregate point differential are **not statistically significant**. The differences in score difference and penalty yards could be completely random. This is another sign we should just stop with this.
2. Second, we look at the Adjusted R-squared value. It's right above the p-value. Adjusted R-squared is a measure of how much of the difference between teams aggregate point values can be explained by penalty yards. Our correlation coefficient said they're 1.4 percent related to each other, but penalty yard's ability to explain the difference between teams? About .04 percent. That's … not much. It's really nothing. Again, we should quit.
3. The third thing we can look at, and we only bother if the first two are meaningful, is the coefficients. In the middle, you can see the (Intercept) is 2.79 and the TotalPenaltyYards coefficient is -0.007407. 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 -0.007407 and b is 2.79. So we would multiply a teams total penalty yards by -0.007407 and then add 2.79 to it. The result would tell you what the total aggregate score in the game would be, according to our model. Chance that your even close with this? About .04 percent. In other words, you've got a 99.96 percent chance of being completely wrong. Did I say we should quit? Yeah.

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

You can see the problem in a graph. On the X axis is penalty yards, on the y is aggregate score. If these elements had a strong relationship, we'd see a clear pattern moving from right to left, sloping down. On the left would be the teams

with few penalties and a positive point differential. On right would be teams with high penalty yards and negative point differentials. Do you see that below?



9.1 A more predictive example

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

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

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

First, let's look at the correlation coefficent.

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

```
## # A tibble: 1 x 1
##   correlation
##         <dbl>
## 1       0.850
```

Answer: 85 percent. Not a perfect relationship, but very good. But how meaningful is that relationship and how predictive is it?

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

```
##
## Call:
## lm(formula = differential ~ NetYards, data = regressions)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -39.981  -8.566   0.171   8.832  39.361
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 0.309946   0.302520   1.025    0.306
## NetYards    0.106536   0.001623  65.644   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 12.29 on 1660 degrees of freedom
## Multiple R-squared:  0.7219,Adjusted R-squared:  0.7217
## F-statistic:  4309 on 1 and 1660 DF,  p-value: < 2.2e-16
```

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

Second, Adjusted R-squared: 0.7217. So we can predict a whopping 72 percent of the difference in the score differential by simply looking at the net yards the team has.

Third, the coefficients: In this case, our `y=mx+b` formula looks like `y = .106536x + .309946`. So if we were applying this, let's look at Nebraska's 27-24 loss to Iowa in 2019. Nebraska's net yards that game? -40.

```
(.106536*-40)+.309946
```

```
## [1] -3.951494
```

So by our model, Nebraska should have lost by 3.95 points. That's really close to the 3 point groin kick of a loss that happened.

Chapter 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, are 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

That presents it's own problems. So let's get our libraries and our data, this time of every college basketball game since the 2014-15 season loaded up.

```
library(tidyverse)
```

```
logs <- read_csv("data/logs1519.csv")
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = ""),
##   HomeAway = col_character(),
##   Opponent = col_character(),
##   W_L = col_character(),
##   Blank = col_logical(),
##   Team = col_character(),
##   Conference = col_character(),
##   season = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

So 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.

```
logs <- logs %>% mutate(Differential = TeamScore - OpponentScore)
```

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 offensive shooting percentage.

```
shooting <- lm(TeamFGPCT ~ Differential, data=logs)
summary(shooting)
```

```
##
## Call:
## lm(formula = TeamFGPCT ~ Differential, data = logs)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.260485 -0.040230 -0.001096  0.039038  0.267457
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  4.399e-01  2.487e-04  1768.4   <2e-16 ***
## Differential 2.776e-03  1.519e-05   182.8   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.05949 on 57514 degrees of freedom
##   (4 observations deleted due to missingness)
## Multiple R-squared:  0.3675,Adjusted R-squared:  0.3674
## F-statistic: 3.341e+04 on 1 and 57514 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 37 percent of the difference in differential with how well a team shoots the ball.

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

```
model1 <- lm(Differential ~ TeamFGPCT + OpponentFGPCT, data=logs)
summary(model1)
```

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

```
##     Min      1Q  Median      3Q     Max
## -49.591  -6.185  -0.198   5.938  68.344
##
## Coefficients:
##                Estimate Std. Error  t value Pr(>|t|)
## (Intercept)      1.1195     0.3483    3.214  0.00131 **
## TeamFGPCT      118.5211     0.5279  224.518  < 2e-16 ***
## OpponentFGPCT -119.9369     0.5252 -228.372  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 9.407 on 57513 degrees of freedom
##   (4 observations deleted due to missingness)
## Multiple R-squared:  0.6683,Adjusted R-squared:  0.6683
## F-statistic: 5.793e+04 on 2 and 57513 DF,  p-value: < 2.2e-16
```

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

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

Third: Compare the residual standard error. We went from .05949 to 9.4. The meaning of this is both really opaque and also simple – we added a lot of error to our model by adding more measures – 158 times more. 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 grew pretty dramatically, but so did the amount of the difference we could predict. It's a trade off.

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 differential first, and then columns 9-24, and then 26-41. Why the skip? There's a blank column in the middle of the data – a remnant of the scraper I used.

```
simplelogs <- logs %>% select(Differential, 9:24, 26:41)
```

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       TeamFG      TeamFGA    TeamFGPCT
## Differential          1.000000000  0.584766682  0.107389235  0.606178206
## TeamFG                0.584766682  1.000000000  0.563220974  0.751715176
## TeamFGA               0.107389235  0.563220974  1.000000000 -0.109620267
## TeamFGPCT             0.606178206  0.751715176 -0.109620267  1.000000000
## Team3P                0.318300418  0.408787900  0.213352219  0.322872202
## Team3PA               0.056680627  0.179527313  0.426011924 -0.119421368
## Team3PPCT             0.367934059  0.380235821 -0.101463821  0.545986963
## TeamFT                0.238182740 -0.022308582 -0.137853824  0.084649669
## TeamFTA               0.206075949 -0.027927391 -0.129851346  0.070632302
## TeamFTPCT             0.138833800  0.016247282 -0.044394472  0.056887587
## TeamOffRebounds       0.136095147  0.161626257  0.545231683 -0.234244567
## TeamTotalRebounds     0.470722398  0.328460524  0.470719037  0.018581908
## TeamAssists           0.540398009  0.664057724  0.284659104  0.566152928
## TeamSteals            0.277670288  0.210221346  0.208743124  0.080191710
## TeamBlocks            0.257608076  0.140856644  0.074555286  0.107327505
## TeamTurnovers        -0.180578328 -0.143210529 -0.223971265  0.001901048
## TeamPersonalFouls    -0.194427271 -0.014722266  0.107325560 -0.094653222
## OpponentFG           -0.538515115  0.144061400  0.256737262 -0.020183466
## OpponentFGA           0.001768386  0.302143806  0.301593528  0.126415534
## OpponentFGPCT        -0.614427717 -0.058571888  0.068034775 -0.114791403
## Opponent3P           -0.283754971  0.131517138  0.135290090  0.053105214
## Opponent3PA           0.013910296  0.191131927  0.138445785  0.118723805
## Opponent3PPCT        -0.382427841  0.008026622  0.057261756 -0.031370545
## OpponentFT           -0.269300868  0.019511923  0.157025930 -0.091558712
## OpponentFTA          -0.226064714  0.012937366  0.159529646 -0.101685664
## OpponentFTPCT        -0.175223632  0.007923359  0.023732217 -0.006190565
## OpponentOffRebounds  -0.089347536 -0.036316958  0.002848058 -0.042399744
```

```
## OpponentTotalRebounds -0.420010794 -0.225202127  0.316139528 -0.512983306
## OpponentAssists       -0.491676030  0.004558539  0.149320067 -0.106252682
## OpponentSteals        -0.187754380 -0.102436608 -0.131734964 -0.021724636
## OpponentBlocks        -0.262252627 -0.160469663  0.218483865 -0.356255034
## OpponentTurnovers      0.274326954  0.155293275  0.198127970  0.024254833
## OpponentPersonalFouls  0.169025733 -0.023116620 -0.107189301  0.060150658
##                             Team3P     Team3PA     Team3PPCT       TeamFT
## Differential           0.318300418  0.05668063  0.3679340589  0.238182740
## TeamFG                 0.408787900  0.17952731  0.3802358207 -0.022308582
## TeamFGA                0.213352219  0.42601192 -0.1014638212 -0.137853824
## TeamFGPCT              0.322872202 -0.11942137  0.5459869634  0.084649669
## Team3P                 1.000000000  0.70114773  0.7073663404 -0.106344056
## Team3PA                0.701147726  1.00000000  0.0407645751 -0.160515313
## Team3PPCT              0.707366340  0.04076458  1.0000000000  0.005129556
## TeamFT                -0.106344056 -0.16051531  0.0051295561  1.000000000
## TeamFTA               -0.137499074 -0.18150913 -0.0180696209  0.927525817
## TeamFTPCT              0.048777304  0.01119250  0.0553684315  0.387017653
## TeamOffRebounds       -0.062026229  0.12484929 -0.1968568361  0.087168289
## TeamTotalRebounds      0.038344971  0.12095682 -0.0628970009  0.190691619
## TeamAssists            0.519530086  0.28786139  0.4326950943 -0.016343370
## TeamSteals             0.016545254  0.04598400 -0.0246657289  0.088535320
## TeamBlocks             0.004747719 -0.02895321  0.0294277389  0.092392379
## TeamTurnovers         -0.088374940 -0.10883919 -0.0209433827  0.051609207
## TeamPersonalFouls     -0.024028303  0.02499520 -0.0498165852  0.217846416
## OpponentFG             0.123800594  0.15638030  0.0296913406  0.057853338
## OpponentFGA            0.148931744  0.13062824  0.0812237901  0.193116094
## OpponentFGPCT          0.029908235  0.08057726 -0.0264843759 -0.075399282
## Opponent3P             0.079455775  0.07482590  0.0402012413  0.024228311
## Opponent3PA            0.085704376  0.05927299  0.0601150176  0.079894905
## Opponent3PPCT          0.029666235  0.04634676  0.0005076038 -0.035478488
## OpponentFT             0.009796521  0.06316300 -0.0390873639  0.161311559
## OpponentFTA           -0.002503282  0.05474884 -0.0480732723  0.183801456
## OpponentFTPCT          0.022780414  0.02587876  0.0086512859 -0.015688533
## OpponentOffRebounds   -0.007870292 -0.01895081  0.0086776821  0.064938518
## OpponentTotalRebounds -0.062384273  0.20289676 -0.2638845414 -0.064969878
## OpponentAssists        0.029413582  0.08254506 -0.0320289494 -0.057730062
## OpponentSteals        -0.053878305 -0.05298037 -0.0251316716 -0.001883349
## OpponentBlocks        -0.111782062 -0.05804217 -0.0965607977 -0.065055523
## OpponentTurnovers      0.009284106  0.06383515 -0.0488449748  0.136922084
## OpponentPersonalFouls -0.127197007 -0.15536393 -0.0268876881  0.793539202
##                            TeamFTA    TeamFTPCT TeamOffRebounds
## Differential           0.206075949  0.138833800    0.1360951470
## TeamFG                -0.027927391  0.016247282    0.1616262575
## TeamFGA               -0.129851346 -0.044394472    0.5452316831
## TeamFGPCT              0.070632302  0.056887587   -0.2342445674
## Team3P                -0.137499074  0.048777304   -0.0620262290
```

```
## Team3PA               -0.181509133  0.011192503    0.1248492948
## Team3PPCT             -0.018069621  0.055368431   -0.1968568361
## TeamFT                 0.927525817  0.387017653    0.0871682888
## TeamFTA                1.000000000  0.053233778    0.1415933172
## TeamFTPCT              0.053233778  1.000000000   -0.0948040467
## TeamOffRebounds        0.141593317 -0.094804047    1.0000000000
## TeamTotalRebounds      0.231278690 -0.037356471    0.6373027887
## TeamAssists           -0.028289202  0.025948025    0.0509277222
## TeamSteals             0.111199125 -0.025969502    0.1195581042
## TeamBlocks             0.104112579 -0.001425412    0.1060163877
## TeamTurnovers          0.072070652 -0.034614485    0.0371728710
## TeamPersonalFouls      0.250787085 -0.025827923    0.0542337992
## OpponentFG             0.043602296  0.036986356   -0.0464694335
## OpponentFGA            0.193466766  0.040334507    0.0242353640
## OpponentFGPCT         -0.091897172  0.012864509   -0.0688833747
## Opponent3P             0.009600704  0.031763685   -0.0063710321
## Opponent3PA            0.071193179  0.032554796    0.0003753868
## Opponent3PPCT         -0.047136861  0.013996880   -0.0056578317
## OpponentFT             0.180010001 -0.009352580    0.0434399899
## OpponentFTA            0.213209437 -0.025707797    0.0584669041
## OpponentFTPCT         -0.032862991  0.028078614   -0.0319032781
## OpponentOffRebounds    0.077003661 -0.016936223   -0.0143325753
## OpponentTotalRebounds  0.004736343 -0.177541483   -0.0603891339
## OpponentAssists       -0.063875391 -0.007401206   -0.0386521955
## OpponentSteals         0.006758108 -0.022033431    0.0326977763
## OpponentBlocks        -0.053973588 -0.041175463    0.1571812909
## OpponentTurnovers      0.169704736 -0.035463921    0.1154717115
## OpponentPersonalFouls  0.866395092  0.018757079    0.1240631120
##                       TeamTotalRebounds   TeamAssists   TeamSteals   TeamBlocks
## Differential              0.470722398  0.5403980088  0.277670288  0.257608076
## TeamFG                    0.328460524  0.6640577242  0.210221346  0.140856644
## TeamFGA                   0.470719037  0.2846591045  0.208743124  0.074555286
## TeamFGPCT                 0.018581908  0.5661529279  0.080191710  0.107327505
## Team3P                    0.038344971  0.5195300862  0.016545254  0.004747719
## Team3PA                   0.120956819  0.2878613903  0.045984003 -0.028953212
## Team3PPCT                -0.062897001  0.4326950943 -0.024665729  0.029427739
## TeamFT                    0.190691619 -0.0163433697  0.088535320  0.092392379
## TeamFTA                   0.231278690 -0.0282892019  0.111199125  0.104112579
## TeamFTPCT                -0.037356471  0.0259480253 -0.025969502 -0.001425412
## TeamOffRebounds           0.637302789  0.0509277222  0.119558104  0.106016388
## TeamTotalRebounds         1.000000000  0.2321524530  0.027446991  0.265518873
## TeamAssists               0.232152453  1.0000000000  0.164837110  0.144764562
## TeamSteals                0.027446991  0.1648371104  1.000000000  0.065539758
## TeamBlocks                0.265518873  0.1447645615  0.065539758  1.000000000
## TeamTurnovers             0.109155292 -0.0789200586  0.078278779  0.032775757
## TeamPersonalFouls        -0.007423332 -0.1050900267  0.005151965 -0.054105029
```

```
## OpponentFG                -0.229331788 -0.0022308763 -0.138728115 -0.143969401
## OpponentFGA                0.360268614  0.1863368268 -0.120696505  0.257245080
## OpponentFGPCT             -0.530432484 -0.1397140493 -0.068951590 -0.353110391
## Opponent3P                -0.053371243  0.0354785684 -0.062074442 -0.103465578
## Opponent3PA                0.232049186  0.1116023406 -0.039184667 -0.042234814
## Opponent3PPCT             -0.273572339 -0.0502063543 -0.047114732 -0.099440199
## OpponentFT                -0.095266106 -0.0835716395 -0.034152581 -0.070920662
## OpponentFTA               -0.022971823 -0.0841605708 -0.022178476 -0.056095076
## OpponentFTPCT             -0.194279344 -0.0278263543 -0.041125993 -0.052504157
## OpponentOffRebounds       -0.052416263 -0.0333847454  0.016707012  0.178200671
## OpponentTotalRebounds     -0.059965631 -0.2225952122  0.035155522  0.037788375
## OpponentAssists           -0.218597433  0.0006884142 -0.053327136 -0.151146052
## OpponentSteals             0.066119486 -0.0288668673  0.055697260  0.028453380
## OpponentBlocks             0.013924890 -0.1657235463 -0.002230784 -0.038978593
## OpponentTurnovers         -0.034355689  0.1314533533  0.730885169  0.031375703
## OpponentPersonalFouls      0.189144014 -0.0267820830  0.071442012  0.080582762
##                       TeamTurnovers TeamPersonalFouls   OpponentFG  OpponentFGA
## Differential          -0.180578328      -0.194427271 -0.538515115  0.001768386
## TeamFG                -0.143210529      -0.014722266  0.144061400  0.302143806
## TeamFGA               -0.223971265       0.107325560  0.256737262  0.301593528
## TeamFGPCT              0.001901048      -0.094653222 -0.020183466  0.126415534
## Team3P                -0.088374940      -0.024028303  0.123800594  0.148931744
## Team3PA               -0.108839191       0.024995197  0.156380301  0.130628244
## Team3PPCT             -0.020943383      -0.049816585  0.029691341  0.081223790
## TeamFT                 0.051609207       0.217846416  0.057853338  0.193116094
## TeamFTA                0.072070652       0.250787085  0.043602296  0.193466766
## TeamFTPCT             -0.034614485      -0.025827923  0.036986356  0.040334507
## TeamOffRebounds        0.037172871       0.054233799 -0.046469434  0.024235364
## TeamTotalRebounds      0.109155292      -0.007423332 -0.229331788  0.360268614
## TeamAssists           -0.078920059      -0.105090027 -0.002230876  0.186336827
## TeamSteals             0.078278779       0.005151965 -0.138728115 -0.120696505
## TeamBlocks             0.032775757      -0.054105029 -0.143969401  0.257245080
## TeamTurnovers          1.000000000       0.220285924  0.081879049  0.155947902
## TeamPersonalFouls      0.220285924       1.000000000 -0.015422966 -0.122639976
## OpponentFG             0.081879049      -0.015422966  1.000000000  0.515517123
## OpponentFGA            0.155947902      -0.122639976  0.515517123  1.000000000
## OpponentFGPCT         -0.023017156       0.078411084  0.754791141 -0.161220379
## Opponent3P            -0.018088322      -0.126817358  0.399027442  0.193563166
## Opponent3PA            0.041669476      -0.167647391  0.144074778  0.418730422
## Opponent3PPCT         -0.063187150      -0.015909552  0.395540055 -0.118020866
## OpponentFT             0.123594852       0.793147614 -0.013421944 -0.156152803
## OpponentFTA            0.154110278       0.865844664 -0.027151720 -0.151706668
## OpponentFTPCT         -0.034267574       0.026877590  0.037049836 -0.043324702
## OpponentOffRebounds    0.074131214       0.122282037  0.120715447  0.519792207
## OpponentTotalRebounds -0.106168146       0.195017438  0.275438081  0.424276325
## OpponentAssists        0.072644677      -0.022619097  0.638304131  0.231851475
```

```
## OpponentSteals          0.709987911       0.064446997  0.140823916  0.165329579
## OpponentBlocks          0.006463872       0.087211248  0.129076992  0.045565883
## OpponentTurnovers       0.188537020       0.101693555 -0.183558009 -0.215633733
## OpponentPersonalFouls   0.131539040       0.322258517  0.015334210  0.136789046
##                       OpponentFGPCT   Opponent3P   Opponent3PA Opponent3PPCT
## Differential          -0.614427717 -0.283754971  0.0139102958 -0.3824278411
## TeamFG                -0.058571888  0.131517138  0.1911319274  0.0080266219
## TeamFGA                0.068034775  0.135290090  0.1384457845  0.0572617563
## TeamFGPCT             -0.114791403  0.053105214  0.1187238045 -0.0313705446
## Team3P                 0.029908235  0.079455775  0.0857043764  0.0296662353
## Team3PA                0.080577258  0.074825900  0.0592729911  0.0463467602
## Team3PPCT             -0.026484376  0.040201241  0.0601150176  0.0005076038
## TeamFT                -0.075399282  0.024228311  0.0798949051 -0.0354784876
## TeamFTA               -0.091897172  0.009600704  0.0711931792 -0.0471368607
## TeamFTPCT              0.012864509  0.031763685  0.0325547961  0.0139968801
## TeamOffRebounds       -0.068883375 -0.006371032  0.0003753868 -0.0056578317
## TeamTotalRebounds     -0.530432484 -0.053371243  0.2320491861 -0.2735723395
## TeamAssists           -0.139714049  0.035478568  0.1116023406 -0.0502063543
## TeamSteals            -0.068951590 -0.062074442 -0.0391846669 -0.0471147320
## TeamBlocks            -0.353110391 -0.103465578 -0.0422348142 -0.0994401990
## TeamTurnovers         -0.023017156 -0.018088322  0.0416694763 -0.0631871498
## TeamPersonalFouls      0.078411084 -0.126817358 -0.1676473908 -0.0159095518
## OpponentFG             0.754791141  0.399027442  0.1440747785  0.3955400546
## OpponentFGA           -0.161220379  0.193563166  0.4187304220 -0.1180208656
## OpponentFGPCT          1.000000000  0.312295571 -0.1493674362  0.5522792378
## Opponent3P             0.312295571  1.000000000  0.6914518201  0.7094041257
## Opponent3PA           -0.149367436  0.691451820  1.0000000000  0.0303822862
## Opponent3PPCT          0.552279238  0.709404126  0.0303822862  1.0000000000
## OpponentFT             0.106226566 -0.106344743 -0.1743400433  0.0169282910
## OpponentFTA            0.086625216 -0.140194309 -0.1972872368 -0.0080249496
## OpponentFTPCT          0.076650746  0.053774302  0.0101886734  0.0623587723
## OpponentOffRebounds   -0.251623986 -0.085432899  0.0978389488 -0.2013096986
## OpponentTotalRebounds -0.005789348  0.005903551  0.0810576009 -0.0680836101
## OpponentAssists        0.553535793  0.513869716  0.2641728450  0.4428640799
## OpponentSteals         0.036468797 -0.011661373  0.0214481397 -0.0383569868
## OpponentBlocks         0.111935521 -0.004746412 -0.0495426307  0.0354134646
## OpponentTurnovers     -0.048082678 -0.095218199 -0.0944428800 -0.0421344973
## OpponentPersonalFouls -0.081776664 -0.011247805  0.0396475169 -0.0466461289
##                        OpponentFT  OpponentFTA OpponentFTPCT
## Differential          -0.269300868 -0.226064714  -0.175223632
## TeamFG                 0.019511923  0.012937366   0.007923359
## TeamFGA                0.157025930  0.159529646   0.023732217
## TeamFGPCT             -0.091558712 -0.101685664  -0.006190565
## Team3P                 0.009796521 -0.002503282   0.022780414
## Team3PA                0.063163000  0.054748838   0.025878762
## Team3PPCT             -0.039087364 -0.048073272   0.008651286
```

```
## TeamFT                0.161311559  0.183801456  -0.015688533
## TeamFTA               0.180010001  0.213209437  -0.032862991
## TeamFTPCT            -0.009352580 -0.025707797   0.028078614
## TeamOffRebounds       0.043439990  0.058466904  -0.031903278
## TeamTotalRebounds    -0.095266106 -0.022971823  -0.194279344
## TeamAssists          -0.083571639 -0.084160571  -0.027826354
## TeamSteals           -0.034152581 -0.022178476  -0.041125993
## TeamBlocks           -0.070920662 -0.056095076  -0.052504157
## TeamTurnovers         0.123594852  0.154110278  -0.034267574
## TeamPersonalFouls     0.793147614  0.865844664   0.026877590
## OpponentFG           -0.013421944 -0.027151720   0.037049836
## OpponentFGA          -0.156152803 -0.151706668  -0.043324702
## OpponentFGPCT         0.106226566  0.086625216   0.076650746
## Opponent3P           -0.106344743 -0.140194309   0.053774302
## Opponent3PA          -0.174340043 -0.197287237   0.010188673
## Opponent3PPCT         0.016928291 -0.008024950   0.062358772
## OpponentFT            1.000000000  0.928286066   0.393203255
## OpponentFTA           0.928286066  1.000000000   0.063446167
## OpponentFTPCT         0.393203255  0.063446167   1.000000000
## OpponentOffRebounds   0.086671729  0.136423744  -0.082982260
## OpponentTotalRebounds 0.197591588  0.232447345  -0.021281750
## OpponentAssists      -0.012378006 -0.031205800   0.041793598
## OpponentSteals        0.077614062  0.097206119  -0.022196700
## OpponentBlocks        0.101422181  0.110063752   0.008946765
## OpponentTurnovers     0.015778567  0.038679394  -0.052040732
## OpponentPersonalFouls 0.215609923  0.251289640  -0.029978048
##                      OpponentOffRebounds OpponentTotalRebounds OpponentAssists
## Differential                -0.089347536          -0.420010794   -0.4916760300
## TeamFG                      -0.036316958          -0.225202127    0.0045585394
## TeamFGA                      0.002848058           0.316139528    0.1493200670
## TeamFGPCT                   -0.042399744          -0.512983306   -0.1062526818
## Team3P                      -0.007870292          -0.062384273    0.0294135821
## Team3PA                     -0.018950808           0.202896760    0.0825450568
## Team3PPCT                    0.008677682          -0.263884541   -0.0320289494
## TeamFT                       0.064938518          -0.064969878   -0.0577300621
## TeamFTA                      0.077003661           0.004736343   -0.0638753907
## TeamFTPCT                   -0.016936223          -0.177541483   -0.0074012062
## TeamOffRebounds             -0.014332575          -0.060389134   -0.0386521955
## TeamTotalRebounds           -0.052416263          -0.059965631   -0.2185974327
## TeamAssists                 -0.033384745          -0.222595212    0.0006884142
## TeamSteals                   0.016707012           0.035155522   -0.0533271359
## TeamBlocks                   0.178200671           0.037788375   -0.1511460518
## TeamTurnovers                0.074131214          -0.106168146    0.0726446766
## TeamPersonalFouls            0.122282037           0.195017438   -0.0226190966
## OpponentFG                   0.120715447           0.275438081    0.6383041307
## OpponentFGA                  0.519792207           0.424276325    0.2318514751
```

```
## OpponentFGPCT               -0.251623986          -0.005789348    0.5535357935
## Opponent3P                  -0.085432899           0.005903551    0.5138697156
## Opponent3PA                  0.097838949           0.081057601    0.2641728450
## Opponent3PPCT               -0.201309699          -0.068083610    0.4428640799
## OpponentFT                   0.086671729           0.197591588   -0.0123780062
## OpponentFTA                  0.136423744           0.232447345   -0.0312058003
## OpponentFTPCT               -0.082982260          -0.021281750    0.0417935976
## OpponentOffRebounds          1.000000000           0.622115242    0.0095497736
## OpponentTotalRebounds        0.622115242           1.000000000    0.1792668711
## OpponentAssists              0.009549774           0.179266871    1.0000000000
## OpponentSteals               0.081573888          -0.038673692    0.1068223463
## OpponentBlocks               0.096186044           0.258597044    0.1337215898
## OpponentTurnovers            0.017562976           0.073936193   -0.1060361856
## OpponentPersonalFouls        0.071468553           0.020500608   -0.0849725350
##                       OpponentSteals OpponentBlocks OpponentTurnovers
## Differential          -0.187754380  -0.2622526274      0.2743269542
## TeamFG                -0.102436608  -0.1604696630      0.1552932747
## TeamFGA               -0.131734964   0.2184838647      0.1981279705
## TeamFGPCT             -0.021724636  -0.3562550337      0.0242548332
## Team3P                -0.053878305  -0.1117820624      0.0092841059
## Team3PA               -0.052980367  -0.0580421730      0.0638351465
## Team3PPCT             -0.025131672  -0.0965607977     -0.0488449748
## TeamFT                -0.001883349  -0.0650555225      0.1369220844
## TeamFTA                0.006758108  -0.0539735876      0.1697047361
## TeamFTPCT             -0.022033431  -0.0411754626     -0.0354639208
## TeamOffRebounds        0.032697776   0.1571812909      0.1154717115
## TeamTotalRebounds      0.066119486   0.0139248895     -0.0343556886
## TeamAssists           -0.028866867  -0.1657235463      0.1314533533
## TeamSteals             0.055697260  -0.0022307839      0.7308851693
## TeamBlocks             0.028453380  -0.0389785933      0.0313757033
## TeamTurnovers          0.709987911   0.0064638717      0.1885370196
## TeamPersonalFouls      0.064446997   0.0872112484      0.1016935547
## OpponentFG             0.140823916   0.1290769921     -0.1835580089
## OpponentFGA            0.165329579   0.0455658832     -0.2156337333
## OpponentFGPCT          0.036468797   0.1119355214     -0.0480826780
## Opponent3P            -0.011661373  -0.0047464115     -0.0952181989
## Opponent3PA            0.021448140  -0.0495426307     -0.0944428800
## Opponent3PPCT         -0.038356987   0.0354134646     -0.0421344973
## OpponentFT             0.077614062   0.1014221807      0.0157785673
## OpponentFTA            0.097206119   0.1100637520      0.0386793945
## OpponentFTPCT         -0.022196700   0.0089467648     -0.0520407316
## OpponentOffRebounds    0.081573888   0.0961860439      0.0175629757
## OpponentTotalRebounds -0.038673692   0.2585970440      0.0739361927
## OpponentAssists        0.106822346   0.1337215898     -0.1060361856
## OpponentSteals         1.000000000   0.0443672204      0.0740678539
## OpponentBlocks         0.044367220   1.0000000000      0.0001223389
```

```
## OpponentTurnovers        0.074067854   0.0001223389      1.0000000000
## OpponentPersonalFouls    0.030766974  -0.0514541037      0.2252310703
##                       OpponentPersonalFouls
## Differential                     0.16902573
## TeamFG                          -0.02311662
## TeamFGA                         -0.10718930
## TeamFGPCT                        0.06015066
## Team3P                          -0.12719701
## Team3PA                         -0.15536393
## Team3PPCT                       -0.02688769
## TeamFT                           0.79353920
## TeamFTA                          0.86639509
## TeamFTPCT                        0.01875708
## TeamOffRebounds                  0.12406311
## TeamTotalRebounds                0.18914401
## TeamAssists                     -0.02678208
## TeamSteals                       0.07144201
## TeamBlocks                       0.08058276
## TeamTurnovers                    0.13153904
## TeamPersonalFouls                0.32225852
## OpponentFG                       0.01533421
## OpponentFGA                      0.13678905
## OpponentFGPCT                   -0.08177666
## Opponent3P                      -0.01124781
## Opponent3PA                      0.03964752
## Opponent3PPCT                   -0.04664613
## OpponentFT                       0.21560992
## OpponentFTA                      0.25128964
## OpponentFTPCT                   -0.02997805
## OpponentOffRebounds              0.07146855
## OpponentTotalRebounds            0.02050061
## OpponentAssists                 -0.08497254
## OpponentSteals                   0.03076697
## OpponentBlocks                  -0.05145410
## OpponentTurnovers                0.22523107
## OpponentPersonalFouls            1.00000000
```

Notice right away – TeamFG is highly correlated. But it's also highly correlated with TeamFGPCT. And that makes sense. A team that doesn't shoot many shots is not going to have a high score differential. But the number of shots taken and the field goal percentage are also highly related. So 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?

We can add more just by simply adding them.

```
model2 <- lm(Differential ~ TeamFGPCT + OpponentFGPCT + TeamTotalRebounds + OpponentTo
summary(model2)
```

```
##
## Call:
## lm(formula = Differential ~ TeamFGPCT + OpponentFGPCT + TeamTotalRebounds +
##     OpponentTotalRebounds, data = logs)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -44.813  -5.586  -0.109   5.453  60.831
##
## Coefficients:
##                           Estimate Std. Error  t value Pr(>|t|)
## (Intercept)             -3.655461   0.606119   -6.031 1.64e-09 ***
## TeamFGPCT              100.880013   0.560363  180.026  < 2e-16 ***
## OpponentFGPCT          -97.563291   0.565004 -172.677  < 2e-16 ***
## TeamTotalRebounds        0.516176   0.006239   82.729  < 2e-16 ***
## OpponentTotalRebounds  -0.436402   0.006448  -67.679  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 8.501 on 57511 degrees of freedom
##   (4 observations deleted due to missingness)
## Multiple R-squared:  0.7291,Adjusted R-squared:  0.7291
## F-statistic: 3.87e+04 on 4 and 57511 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.

```
Differential = (TeamFGPCT*100.880013) + (OpponentFGPCT*-97.563291) + (TeamTotalRebounds
```

How does this apply in the real world? Let's pretend for a minute that you are Fred Hoiberg, and you have just been hired as Nebraska's Mens Basketball Coach. Your job is to win conference titles and go deep into the NCAA tournament. To do that, we need to know what attributes of a team should we emphasize. We can do that by looking at what previous Big Ten conference champions looked like.

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

```
logs %>% filter(Team == "Michigan State Spartans" & season == "2018-2019" | Team == "Michigan Sta
```

```
## # A tibble: 1 x 4
##   avgfgpct avgoppfgpct avgtotrebound avgopptotrebound
##      <dbl>       <dbl>          <dbl>             <dbl>
## 1    0.489       0.409           35.3              27.2
```

Now it's just plug and chug.

```
(0.4886133*100.880013) + (0.4090221*-97.563291) + (35.29834*0.516176) + (27.20994*-0.436402) - 3.
```

```
## [1] 12.076
```

So a team with those numbers is going to average scoring 12 more points per game than their opponent.

How does that compare to Nebraska of this past season? The last of the Tim Miles era?

```
logs %>%
  filter(
    Team == "Nebraska Cornhuskers" & season == "2018-2019"
    ) %>%
  summarise(
    avgfgpct = mean(TeamFGPCT),
    avgoppfgpct = mean(OpponentFGPCT),
    avgtotrebound = mean(TeamTotalRebounds),
    avgopptotrebound = mean(OpponentTotalRebounds)
    )
```

```
## # A tibble: 1 x 4
##   avgfgpct avgoppfgpct avgtotrebound avgopptotrebound
##      <dbl>       <dbl>          <dbl>             <dbl>
## 1    0.431       0.423           32.5              34.9
```

```
(0.4305833*100.880013) + (0.4226667*-97.563291) + (32.5*0.516176) + (34.94444*-0.436402) - 3.6554
```

```
## [1] 0.07093015
```

By this model, it predicted we would outscore our opponent by .07 points over the season. So we'd win slightly more than we'd lose. Nebraska's overall record? 19-17.

Chapter 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 Nebraska, given a -40 net yardage deficit would lose to Iowa by -3.95 points. They lost by -3. So our residual is -.95. If Iowa fakes that last second field goal and scores a touchdown, our residual would have been -7.

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

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

Let's go back to our net yards model.

Let's first attach libraries and get some data.

```
library(tidyverse)
```

```
logs <- read_csv("data/footballlogs19.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = ""),
##   HomeAway = col_character(),
##   Opponent = col_character(),
##   Result = col_character(),
```

```
##   TeamFull = col_character(),
##   TeamURL = col_character(),
##   Outcome = col_character(),
##   Team = col_character(),
##   Conference = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

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

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

Now let's create our model.

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

```
##
## Call:
## lm(formula = differential ~ NetYards, data = residualmodel)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -39.981  -8.566   0.171   8.832  39.361
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 0.309946   0.302520   1.025    0.306
## NetYards    0.106536   0.001623  65.644   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 12.29 on 1660 degrees of freedom
## Multiple R-squared:  0.7219,Adjusted R-squared:  0.7217
## F-statistic:  4309 on 1 and 1660 DF,  p-value: < 2.2e-16
```

We've seen this output before, but let's review because if you are using scatterplots to make a point, you should do this. First, note the Min and Max residual at the top. A team has underperformed the model by 39 points (!), and a team has overperformed it by 39 points (!!). The median residual, where half are above and half are below, is just slightly above the fit line. Close here is good.

Next: Look at the Adjusted R-squared value. What that says is that 72 percent of a team's scoring output can be predicted by their net yards.

Last: Look at the p-value. We are looking for a p-value smaller than .05. At .05, we can say that our correlation didn't happen at random. And, in this case, it REALLY didn't happen at random. But if you know a little bit about football,

it doesn't surprise you that the more you outgain your opponent, the more you win by. It's an intuitive result.

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

```
residualmodel$predicted <- predict(fit)
residualmodel$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.

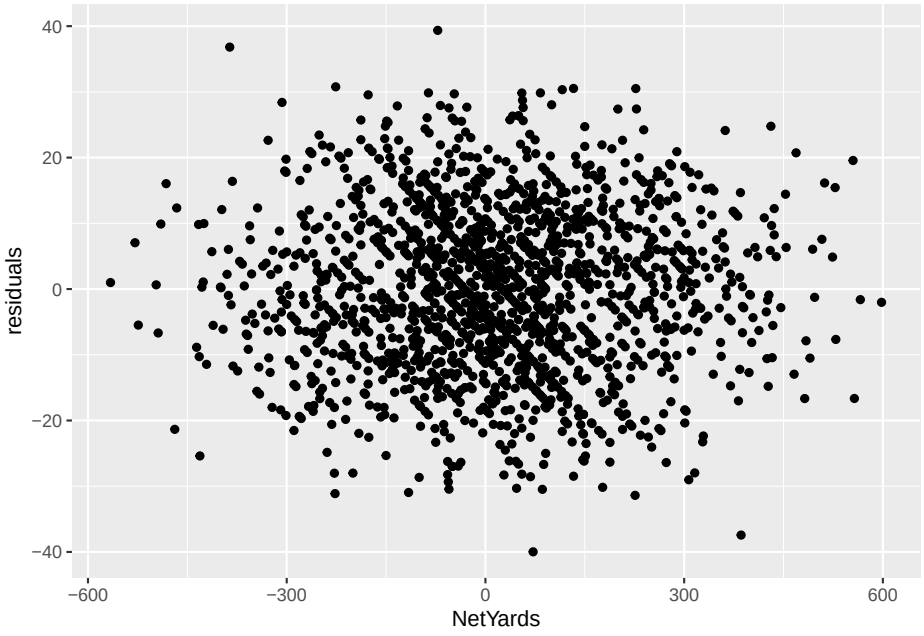
```
residualmodel %>% arrange(desc(residuals)) %>% select(Team, Opponent, Result, NetYards, residuals
```

```
## # A tibble: 1,662 x 5
##    Team          Opponent             Result    NetYards residuals
##    <chr>         <chr>                <chr>        <dbl>     <dbl>
##  1 Penn State    Buffalo              W (45-13)      -72      39.4
##  2 Illinois      Nebraska             L (38-42)     -386      36.8
##  3 Virginia Tech Miami (FL)           W (42-35)     -226      30.8
##  4 Tennessee     Chattanooga          W (45-0)       133      30.5
##  5 Baylor        Kansas               W (61-6)       227      30.5
##  6 Syracuse      Duke                 W (49-6)       116      30.3
##  7 Houston       North Texas          W (46-25)      -86      29.9
##  8 South Florida South Carolina State W (55-16)       83      29.8
##  9 Troy          Texas State          W (63-27)       55      29.8
## 10 Miami (FL)    Louisville           W (52-27)      -47      29.7
## # ... with 1,652 more rows
```

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

Remember Nebraska vs Illinois? We came back to win and everyone was happy and relieved at the same time? We outgained Illinois by **386 yards** in that game and won by 4. Our model predicted Nebraska should have won that game by 36.8 points. Illinois outscored the model by almost as many points as they had. Just goes to show you: you can have all the advantages and you can still screw it up. Just ask Buffalo: They were only outgained by Penn State by 70 yards and still lost by 32.

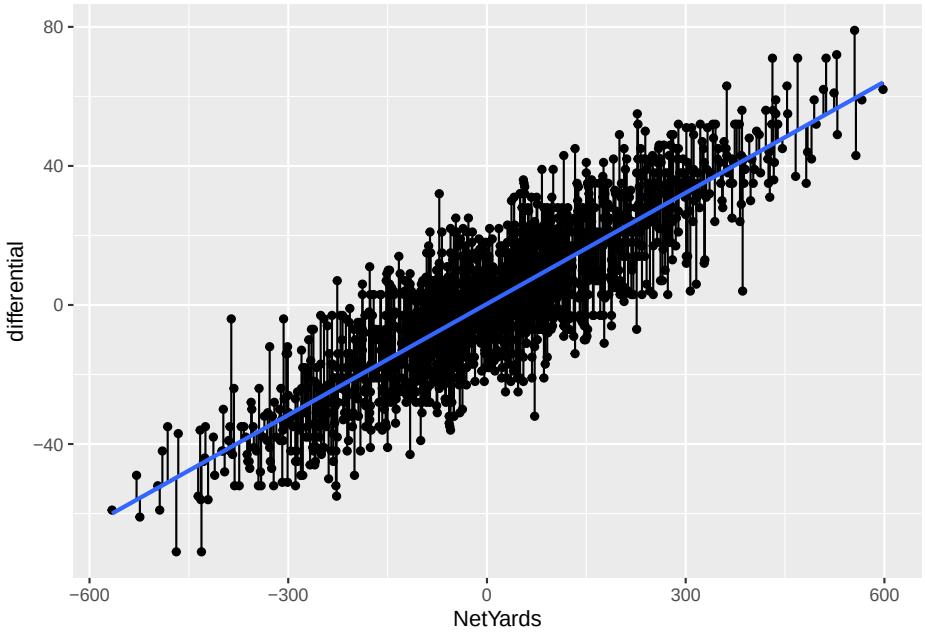
But, before we can bestow any validity on this model, we need to see if this linear model is appropriate. We've done that some looking at our p-values and R-squared values. But one more check is to look at the residuals themselves. We do that by plotting the residuals with the predictor. We'll get into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



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

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

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



The blue line here separates underperformers from overperformers.

11.1 Penalties

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

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

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

```
##
## Call:
## lm(formula = differential ~ TotalPenaltyYards, data = penalties)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -73.13 -15.16   0.71  15.61  76.86
##
## Coefficients:
##                    Estimate Std. Error t value Pr(>|t|)
```

```
## (Intercept)        2.785946   1.525993   1.826   0.0681 .
## TotalPenaltyYards -0.007407   0.013244  -0.559   0.5760
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 23.3 on 1660 degrees of freedom
## Multiple R-squared:  0.0001884,Adjusted R-squared:  -0.0004139
## F-statistic: 0.3128 on 1 and 1660 DF,  p-value: 0.576
```

So from top to bottom:

- Our min and max go from -73 to positive 77
- Our adjusted R-squared is … -0.0004139. Not much at all.
- Our p-value is … 0.576, which is more than than .05.

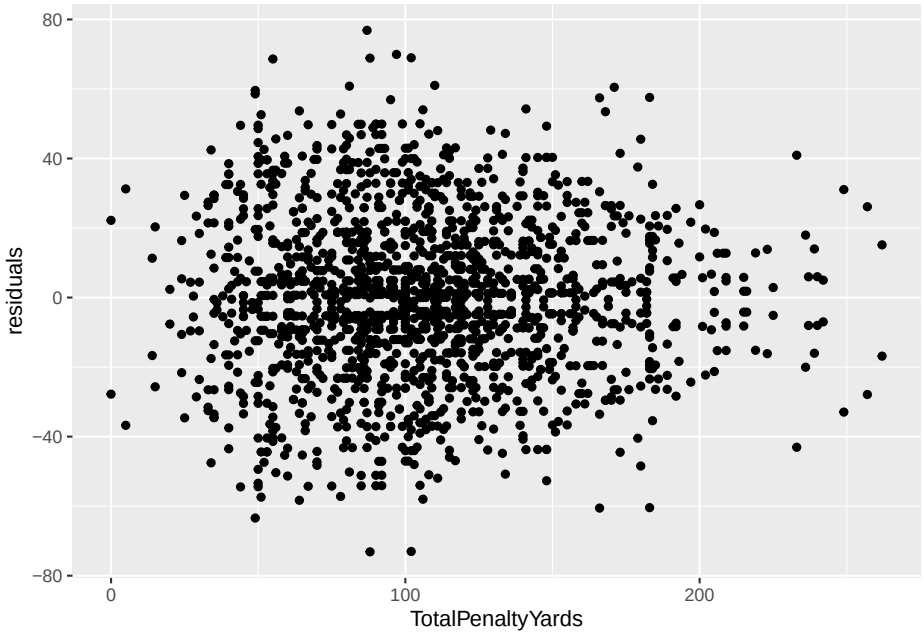
So what we can say about this model is that it's statistically insignificant and utterly meaningless. Normally, we'd stop right here – why bother going forward with a predictive model that isn't predictive? But let's do it anyway.

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

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

```
## # A tibble: 1,662 x 5
##    Team           Opponent            Result    TotalPenaltyYards residuals
##    <chr>          <chr>               <chr>                 <dbl>     <dbl>
##  1 Maryland       Howard              W (79-0)                 87      76.9
##  2 Penn State     Idaho               W (79-7)                 97      69.9
##  3 Ohio State     Miami (OH)          W (76-5)                102      69.0
##  4 Oregon         Nevada              W (77-6)                 88      68.9
##  5 Louisiana      Texas Southern      W (77-6)                 55      68.6
##  6 Miami (FL)     Bethune-Cookman     W (63-0)                110      61.0
##  7 Alabama        Western Carolina    W (66-3)                 81      60.8
##  8 UCF            Florida A&M         W (62-0)                171      60.5
##  9 South Carolina Charleston Southern W (72-10)                49      59.6
## 10 Wisconsin      Central Michigan    W (61-0)                 49      58.6
## # ... with 1,652 more rows
```

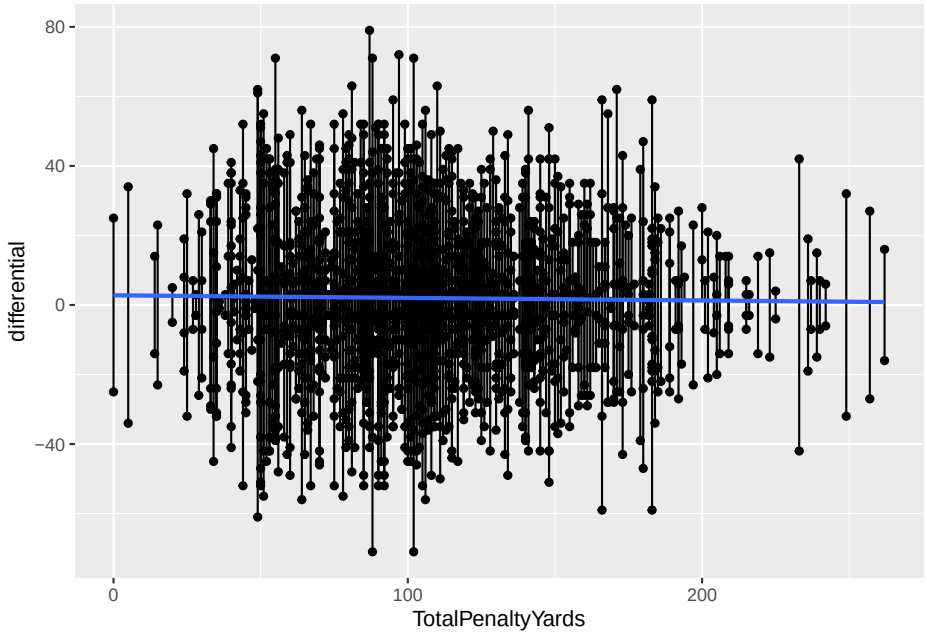
First, note all of the biggest misses here are all blowout games. The worst game of the season last year was, by far, Maryland vs Howard, a 79-0 shellacking that should never have happened. The differential was 79 points. The model missed that differential by … 76.9 points. In other words, this model is terrible. But let's look at it anyway.



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

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

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

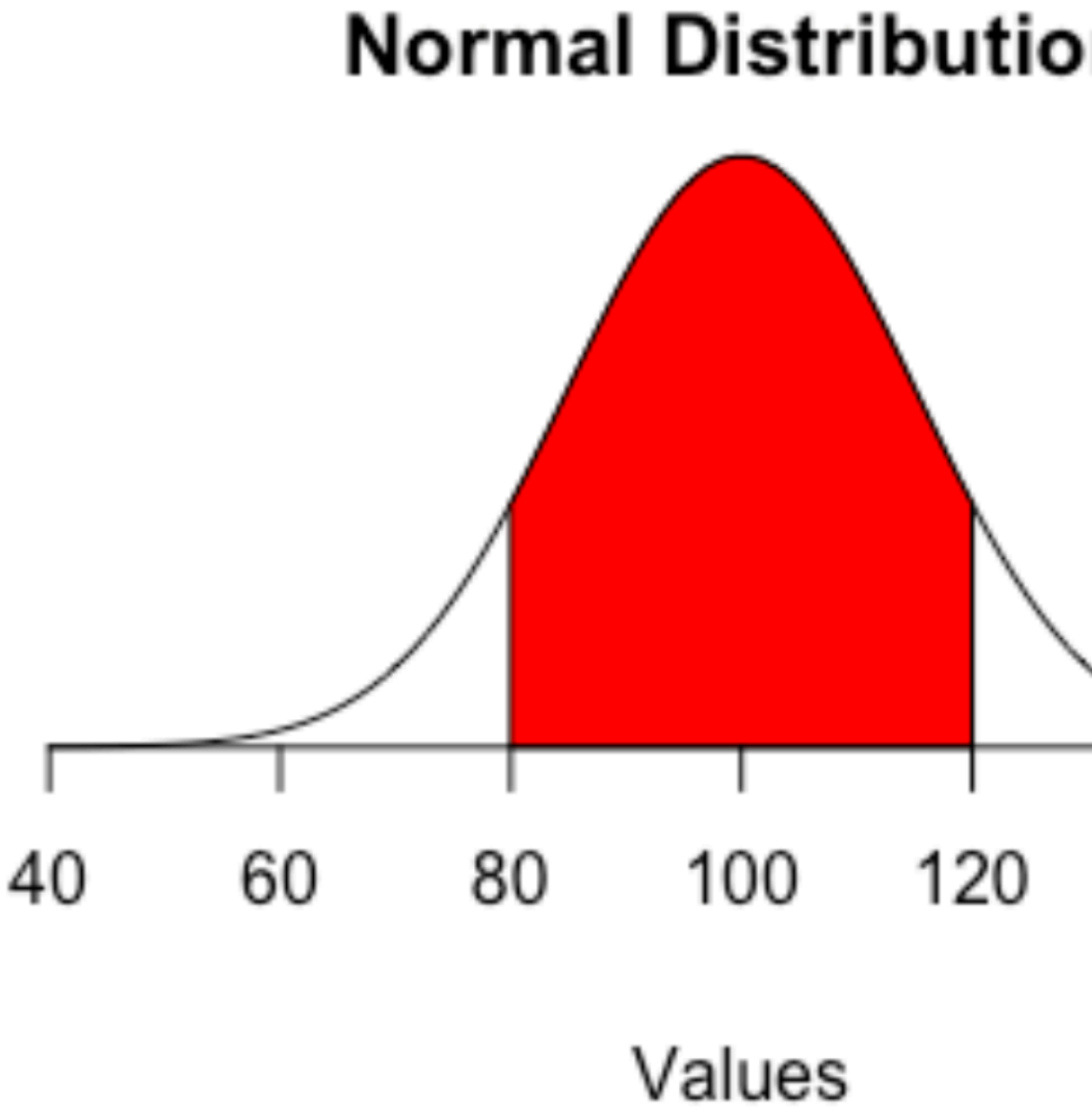

80
40
0
−40
differential
0
100
200
TotalPenaltyYards

Chapter 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 Nebraska basketball team, which if haven't been paying attention to current events, was not good at basketball.

12.1 Calculating a Z score in R

```
library(tidyverse)
```

Let's look at the current state of Nebraska basketball using the same logs data we've been using for the 2019-2020 season.

```
gamelogs <- read_csv("data/logs20.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = ""),
##   HomeAway = col_character(),
##   Opponent = col_character(),
##   W_L = col_character(),
##   Blank = col_logical(),
##   Team = col_character(),
##   Conference = col_character(),
##   season = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

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.

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

```
## `summarise()` regrouping output by 'Conference' (override with `.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.

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

```
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       Davi~ 0.454       31.1    0.437      30.4    0.505   -0.619
## 2 A-10       Dayt~ 0.525       32.5    0.413      29.0    2.59     0.0352
## 3 A-10       Duqu~ 0.444       32.4    0.427      32.4    0.216   -0.0168
## 4 A-10       Ford~ 0.384       30.0    0.402      33.9   -1.53    -1.13
## 5 A-10       Geor~ 0.424       33.8    0.440      30.5   -0.358    0.620
## 6 A-10       Geor~ 0.422       30.5    0.452      32.7   -0.410   -0.904
## # ... with 3 more variables: OppZscore <dbl>, OppRebZScore <dbl>,
## #   TotalZscore <dbl>
```

A way to read this – a team at zero 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: 353 x 11
## # Groups:   Conference [32]
##    Conference Team  FGAvg ReboundAvg OppFGAvg OffRebAvg FGzscore RebZscore
```

```
##    <chr>      <chr> <dbl>      <dbl>    <dbl>     <dbl>    <dbl>     <dbl>
##  1 Big West   UC-I~ 0.473       36.6    0.390      27.1    1.60     2.23
##  2 Big 12     Kans~ 0.482       35.9    0.378      29.0    2.36     1.13
##  3 WCC        Gonz~ 0.517       37.4    0.424      28.2    1.73     1.90
##  4 Southland  Step~ 0.490       34.2    0.427      26.6    1.76     1.05
##  5 Big Ten    Mich~ 0.460       37.7    0.382      29.6    1.38     1.55
##  6 OVC        Murr~ 0.477       35.3    0.401      29.2    1.31     1.36
##  7 Summit     Sout~ 0.492       35.5    0.423      31.3    1.58     1.52
##  8 A-10       Dayt~ 0.525       32.5    0.413      29.0    2.59     0.0352
##  9 A-10       Sain~ 0.457       37.4    0.403      30.5    0.598    2.21
## 10 ACC        Loui~ 0.457       36.6    0.392      29.8    1.11     1.37
## # ... with 343 more rows, and 3 more variables: OppZscore <dbl>,
## #   OppRebZScore <dbl>, TotalZscore <dbl>
```

Don't sleep on the Anteaters! Would have been a tough out at the tournament that never happened.

But closer to home, how is Nebraska doing.

```
teamzscore %>%
  filter(Conference == "Big Ten") %>%
  arrange(desc(TotalZscore))
```

```
## # A tibble: 14 x 11
## # Groups:   Conference [1]
##    Conference Team  FGAvg ReboundAvg OppFGAvg OffRebAvg FGzscore RebZscore
##    <chr>      <chr> <dbl>      <dbl>    <dbl>     <dbl>    <dbl>     <dbl>
##  1 Big Ten    Mich~ 0.460       37.7    0.382      29.6    1.38      1.55
##  2 Big Ten    Rutg~ 0.449       37      0.385      31.1    0.727     1.22
##  3 Big Ten    Ohio~ 0.447       33.6    0.400      28.4    0.592    -0.393
##  4 Big Ten    Illi~ 0.444       36.1    0.418      29.1    0.439     0.779
##  5 Big Ten    Indi~ 0.445       35.1    0.419      29.4    0.480     0.306
##  6 Big Ten    Mary~ 0.419       36.1    0.401      31.9   -0.952     0.794
##  7 Big Ten    Mich~ 0.463       33.0    0.428      31.9    1.56     -0.682
##  8 Big Ten    Penn~ 0.432       35.6    0.411      34.2   -0.237     0.550
##  9 Big Ten    Minn~ 0.426       35.5    0.411      33     -0.560     0.520
## 10 Big Ten    Iowa~ 0.452       34.2    0.430      32.4    0.918    -0.104
## 11 Big Ten    Purd~ 0.418       33.8    0.410      29.3   -1.02     -0.271
## 12 Big Ten    Wisc~ 0.426       31.3    0.410      32.0   -0.587    -1.49
## 13 Big Ten    Nort~ 0.417       30.5    0.422      34.8   -1.12     -1.84
## 14 Big Ten    Nebr~ 0.408       32.4    0.453      42.2   -1.62     -0.947
## # ... with 3 more variables: OppZscore <dbl>, OppRebZScore <dbl>,
## #   TotalZscore <dbl>
```

So, as we can see, with our composite Z Score, Nebraska is … not good. Not good at all.

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.

Chapter 13

Intro to ggplot

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 Wickam, 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 export your work into a program like Illustrator to finish the work.

To begin, we'll import the `ggplot2` and `dplyr` libraries. We'll read in the data, then create a new dataframe that represents our attendance data, similar to what we've done before.

```
library(tidyverse)
```

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

```
## Parsed with column specification:
## cols(
##   Institution = col_character(),
##   Conference = col_character(),
##   `2013` = col_double(),
##   `2014` = col_double(),
##   `2015` = col_double(),
##   `2016` = col_double(),
##   `2017` = col_double(),
##   `2018` = col_double()
## )
```

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

```
attendance %>%
  arrange(desc(`2018`)) %>%
  top_n(10) %>%
  select(Institution, `2018`)
```

```
## Selecting by 2018
```

```
## # A tibble: 10 x 2
##    Institution `2018`
##    <chr>        <dbl>
##  1 Michigan    775156
##  2 Penn St.    738396
##  3 Ohio St.    713630
##  4 Alabama     710931
##  5 LSU         705733
##  6 Texas A&M   698908
##  7 Tennessee   650887
##  8 Georgia     649222
##  9 Nebraska    623240
## 10 Oklahoma    607146
```

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

```
top10 <- attendance %>%
  arrange(desc(`2018`)) %>%
  top_n(10) %>%
  select(Institution, `2018`)
```

```
## Selecting by 2018
```

13.1 The bar chart

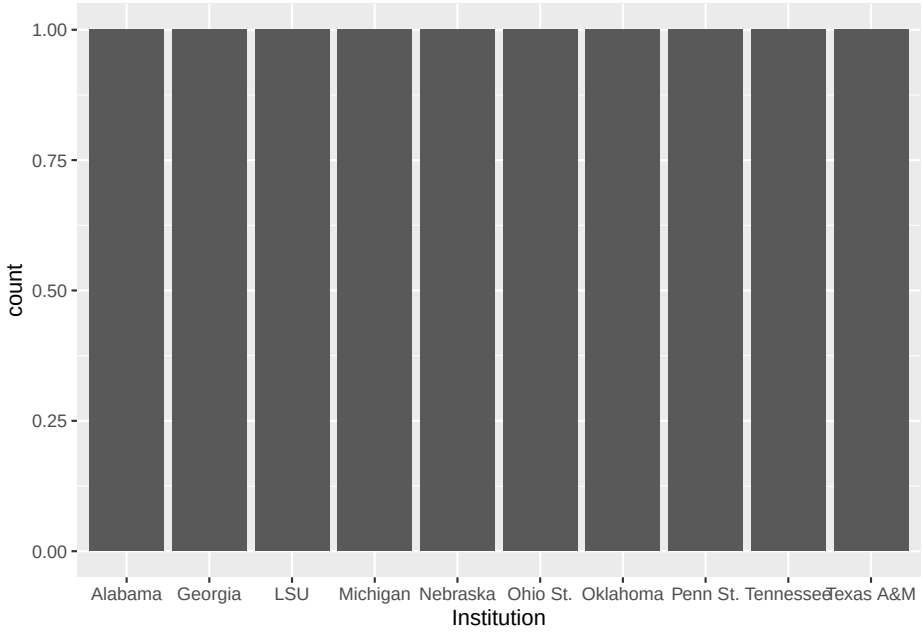
The easiest thing we can do is create a simple bar chart of our data. **Bar charts show magnitude. They invite you to compare how much more or less one thing is compared to others.**

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

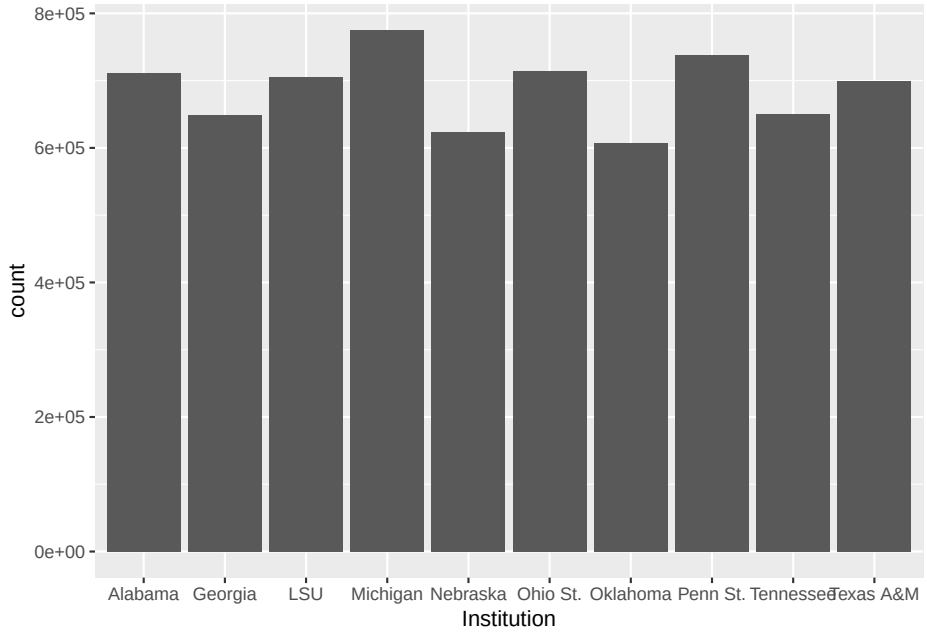
Note: attendance is our data, `aes` means aesthetics, `x=Institution` explicitly tells `ggplot2` that our x value – our horizontal value – is the Institution field from the data, and then we add on the `geom_bar()` as the geometry. And what do we get when we run that?

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



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.

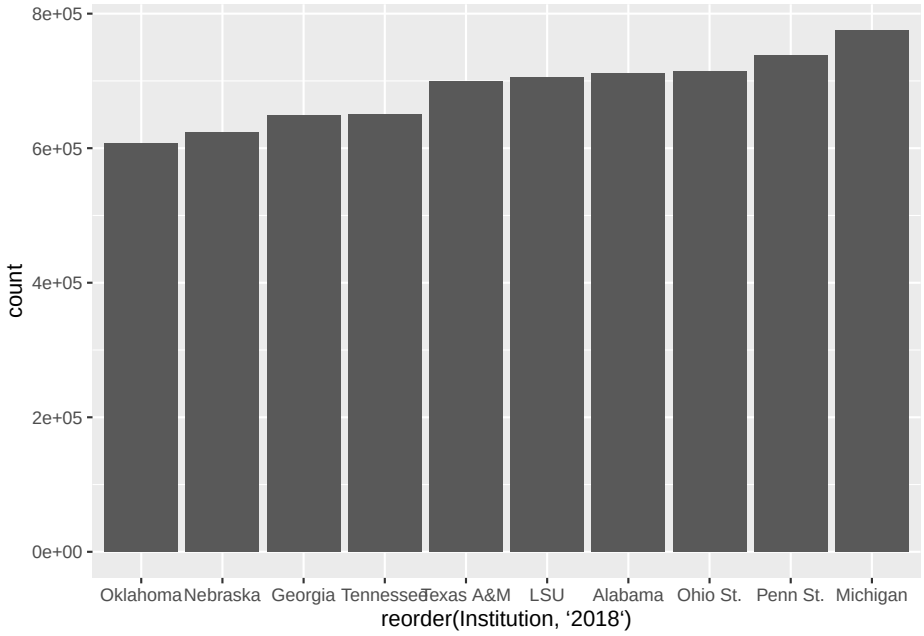
```
ggplot(top10, aes(x=Institution, weight=`2018`)) +
  geom_bar()
```



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

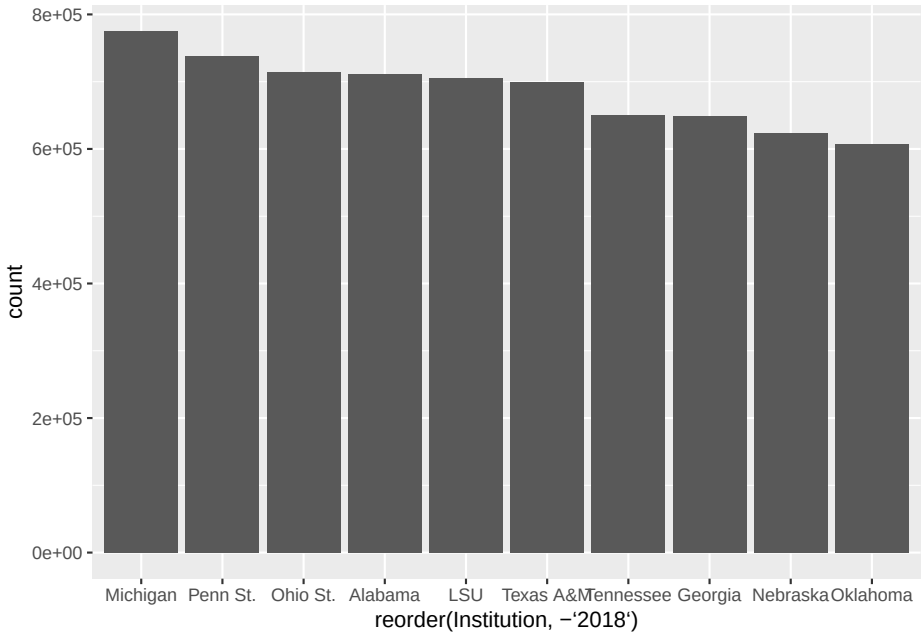
Let's deal with the ordering first. `ggplot2`'s default behavior is to sort the data by the x axis variable. So it's in alphabetical order. To change that, we have to `reorder` it. With `reorder`, we first have to tell `ggplot` what we are reordering, and then we have to tell it HOW we are reordering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

```
ggplot(top10, aes(x=reorder(Institution, `2018`), weight=`2018`)) + geom_bar()
```



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(top10, aes(x=reorder(Institution, -`2018`), weight=`2018`)) + geom_bar()
```



13.2 Scales

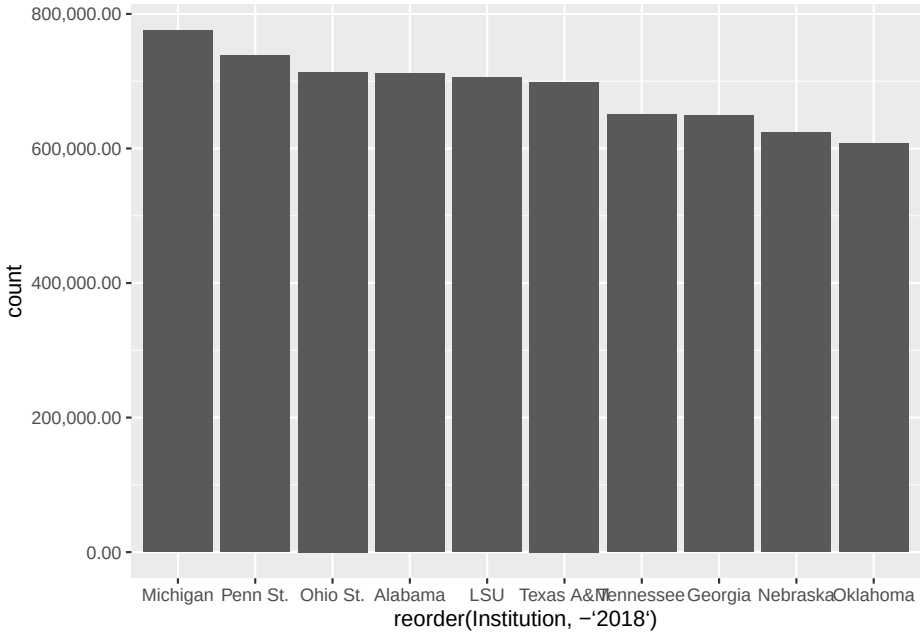
To fix the axis labels, we need try one of the other main elements of the `ggplot2` library, which is transform a scale. More often that not, that means doing something like putting it on a logarithmic scale or some other kind of transformation. In this case, we're just changing how it's represented. 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)
```

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(top10, aes(x=reorder(Institution, -`2018`), weight=`2018`)) +
  geom_bar() +
  scale_y_continuous(labels=comma)
```

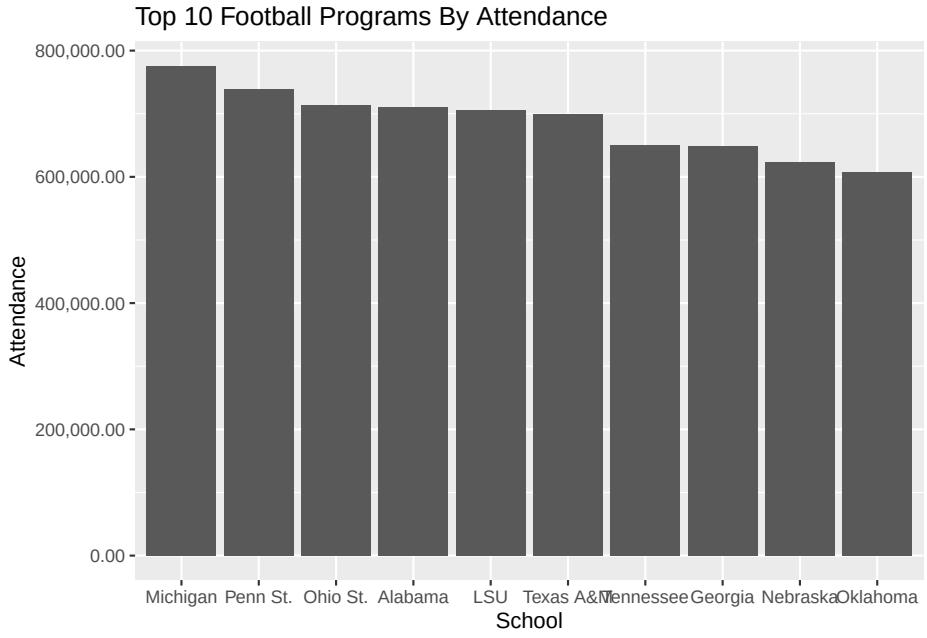


Better.

13.3 Styling

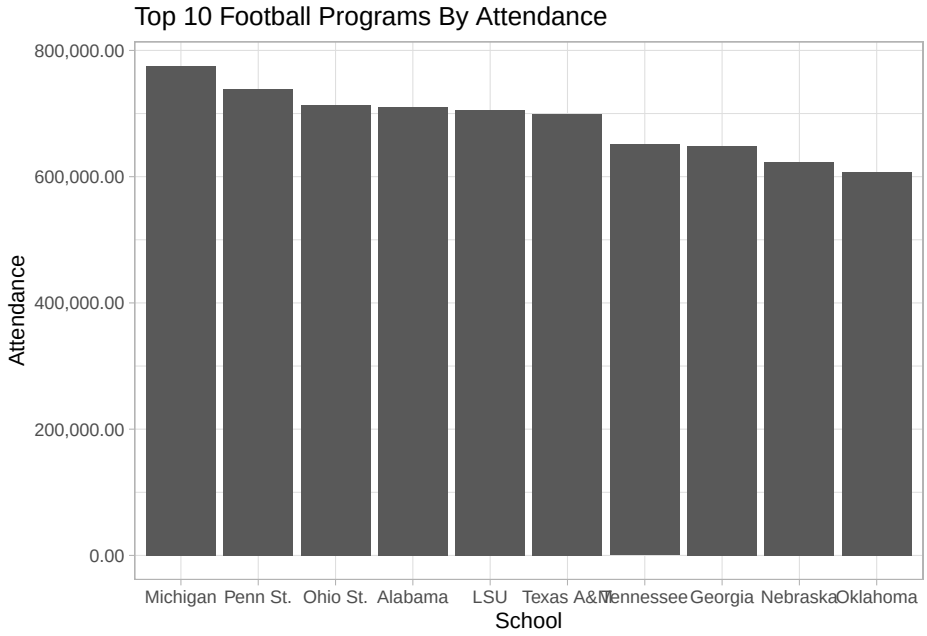
We are going to spend a lot more time on styling, but let's add some simple labels to this with a new bit called `labs` which is short for labels.

```
ggplot(top10, aes(x=reorder(Institution, -`2018`), weight=`2018`)) +
  geom_bar() +
  scale_y_continuous(labels=comma) +
  labs(title="Top 10 Football Programs By Attendance", x="School", y="Attendance")
```



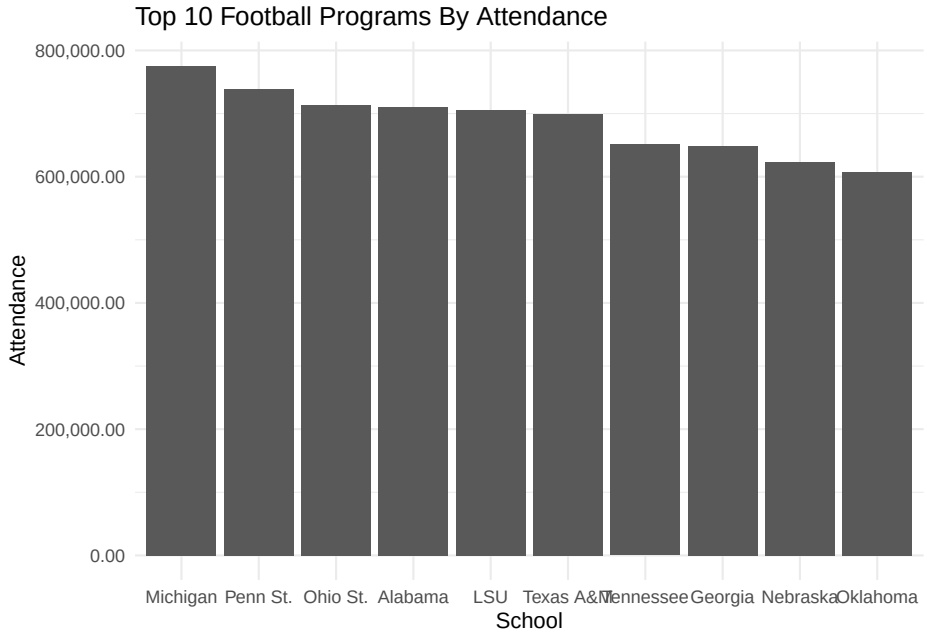
The library has lots and lots of ways to alter the styling – we can programmatically control nearly every part of the look and feel of the chart. One simple way is to apply themes in the library already. We do that the same way we've done other things – we add them. Here's the light theme.

```
ggplot(top10, aes(x=reorder(Institution, -`2018`), weight=`2018`)) +
  geom_bar() +
  scale_y_continuous(labels=comma) +
  labs(title="Top 10 Football Programs By Attendance", x="School", y="Attendance") +
  theme_light()
```



Or the minimal theme:

```
ggplot(top10, aes(x=reorder(Institution, -`2018`), weight=`2018`)) +
  geom_bar() +
  scale_y_continuous(labels=comma) +
  labs(title="Top 10 Football Programs By Attendance", x="School", y="Attendance") +
  theme_minimal()
```

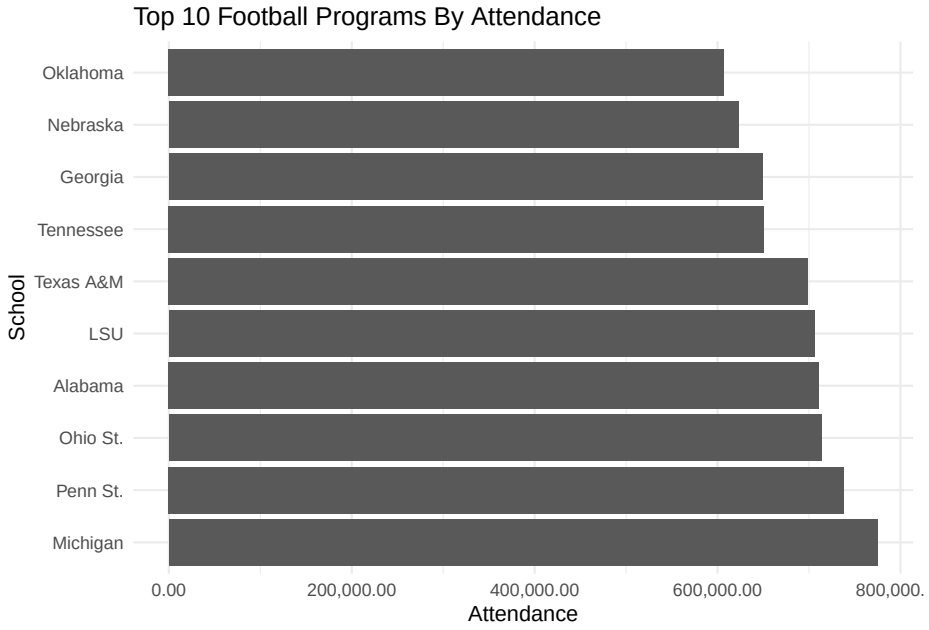


Later on, we'll write our own themes. For now, the built in ones will get us closer to something that looks good.

13.4 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(top10, aes(x=reorder(Institution, -`2018`), weight=`2018`)) +
  geom_bar() +
  scale_y_continuous(labels=comma) +
  labs(title="Top 10 Football Programs By Attendance", x="School", y="Attendance") +
  theme_minimal() +
  coord_flip()
```


Top 10 Football Programs By Attendance
Oklahoma
Nebraska
Georgia
Tennessee
Texas A&M
LSU
Alabama
Ohio St.
Penn St.
Michigan
School
0.00
200,000.00
400,000.00
600,000.00
800,000.
Attendance

Chapter 14

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.

```
library(tidyverse)
```

We're going to use a dataset of graduation rates by gender by school in the NCAA. You can get it here.

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

```
## Parsed with column specification:
## cols(
##   `Institution name` = col_character(),
##   `Primary Conference in Actual Year` = col_character(),
##   `Cohort year` = col_double(),
##   Gender = col_character(),
##   `Number of completers` = col_double(),
##   Total = col_double()
## )
```

What we have here is the name of the school, the conference, the cohort of when they started school, the gender, the number of that gender that graduated and the total number of graduates in that cohort.

Let's pretend for a moment we're looking at the graduation rates of men and women in the Big 10 Conference and we want to chart that. First, let's work on our data. We need to filter the "Big Ten Conference" school, and we want

the latest year, which is 2009. So we'll create a dataframe called `BIG09` and populate it.

```
BIG09 <- grads %>% filter(`Primary Conference in Actual Year`=="Big Ten Conference") %
```

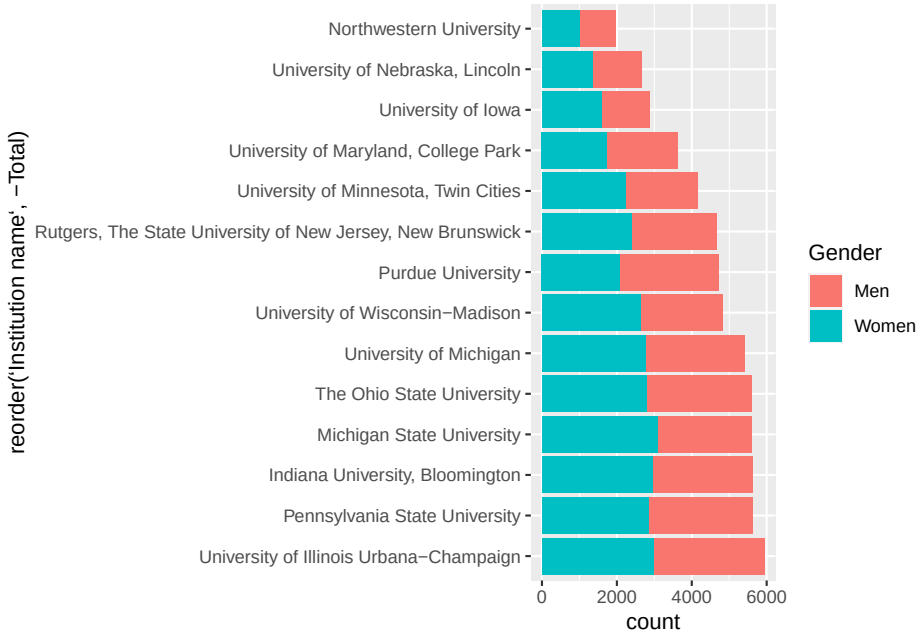
Reminder: `head()` will give you a quick look at what your data looks like, but **it will only show you the first six rows**.

```
head(BIG09)
```

```
## # A tibble: 6 x 6
##   `Institution nam~ `Primary Confer~ `Cohort year` Gender `Number of comp~ Total
##   <chr>             <chr>                    <dbl> <chr>              <dbl> <dbl>
## 1 University of Il~ Big Ten Confere~          2009 Men                 2973  5940
## 2 University of Il~ Big Ten Confere~          2009 Women               2967  5940
## 3 Northwestern Uni~ Big Ten Confere~          2009 Men                  963  1974
## 4 Northwestern Uni~ Big Ten Confere~          2009 Women               1011  1974
## 5 Indiana Universi~ Big Ten Confere~          2009 Men                 2667  5626
## 6 Indiana Universi~ Big Ten Confere~          2009 Women               2959  5626
```

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're 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 Gender, which means it will stack the number of male graduates on top of the number of female graduates and we can see how they compare.

```
ggplot(BIG09, aes(x=reorder(`Institution name`, -Total), weight=`Number of completers`
```



What's the problem with this chart?

Let me ask a different question – which schools have larger differences in male and female graduation rates? Can you compare Illnois to Northwestern? Not really. We've charted the total numbers. We need the percentage of the whole.

> **YOUR TURN**: Using what you know – hint: mutate – how could you chart this using percents of the whole instead of counts?

Chapter 15

Waffle charts

Pie charts are the devil. They should be an instant F in any data visualization class. I'll give you an example of why.

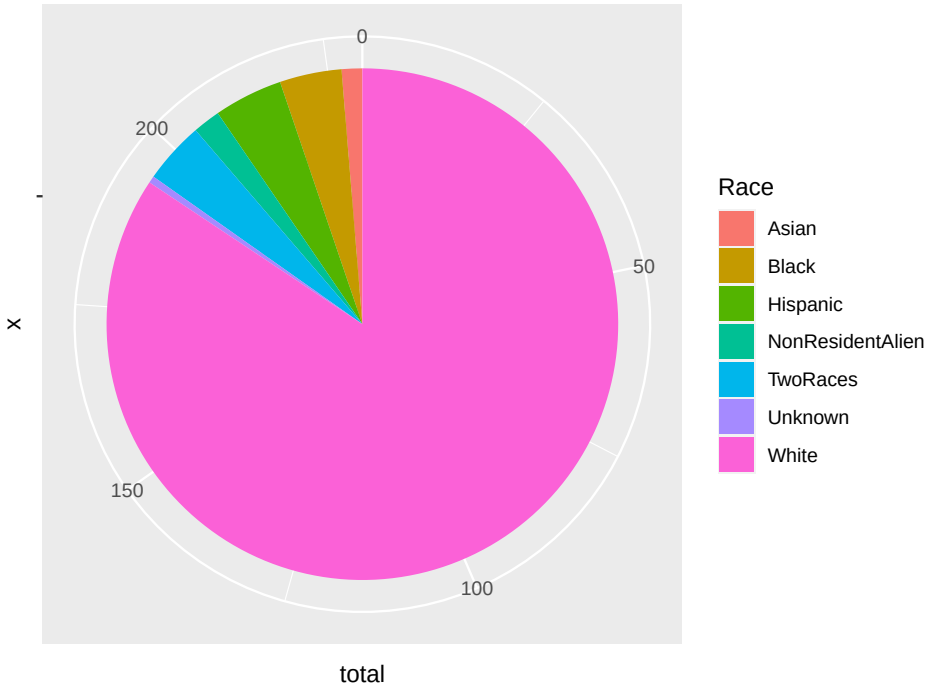
What's the racial breakdown of journalism majors at UNL?

Here it is in a pie chart:

```
library(tidyverse)

enrollment <- read.csv("~/Box/Courses/JOUR407-Data-Visualization/Data/collegeenrollment.csv")

jour <- filter(enrollment, MajorName == "Journalism")

jdf <- jour %>%
group_by(Race) %>%
summarise(
       total=sum(Count)) %>%
select(Race, total) %>%
filter(total != 0)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
ggplot(jdf, aes(x="", y=total, fill=Race)) + geom_bar(width = 1, stat = "identity") + coord_polar
```



You can see, it's pretty white. But … what about beyond that? How carefully can you evaluate angles and area?

Not well.

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:

```
install.packages('waffle')
```

Now load it:

```
library(waffle)
```

Let's look at the debacle that was Nebraska vs. Ohio State this fall in Football. Here's the box score, which we'll use for this walkthrough.

The easiest way to do waffle charts is to make vectors of your data and plug them in. To make a vector, we use the `c` or concatenate function, something we've done before.

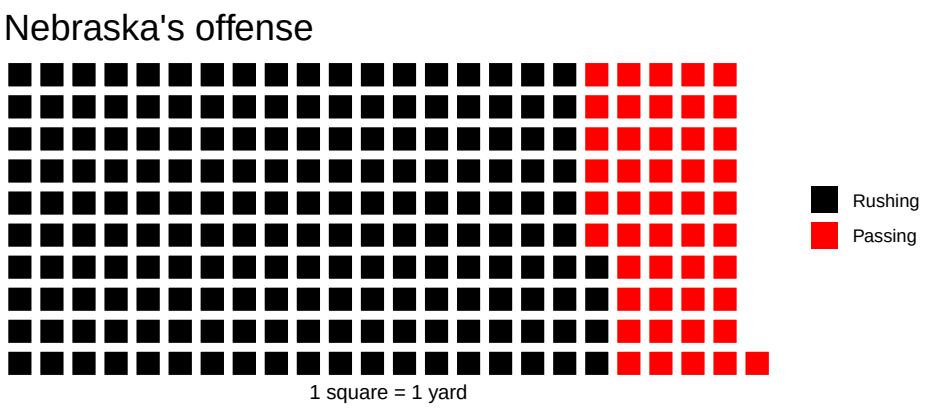
So let's look at offense. Rushing vs passing.

```
nu <- c("Rushing"=184, "Passing"=47)
oh <- c("Rushing"=368, "Passing"=212)
```

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.

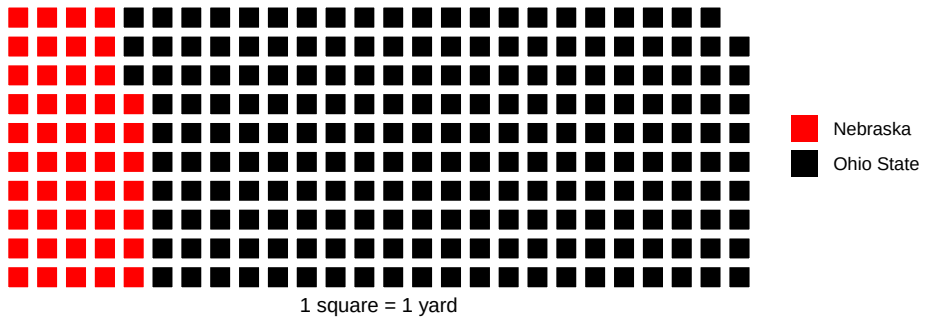
```
waffle(nu, rows = 10, title="Nebraska's offense", xlab="1 square = 1 yard", colors = c("black", '
```



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

```
passing <- c("Nebraska"=47, "Ohio State"=212)
```

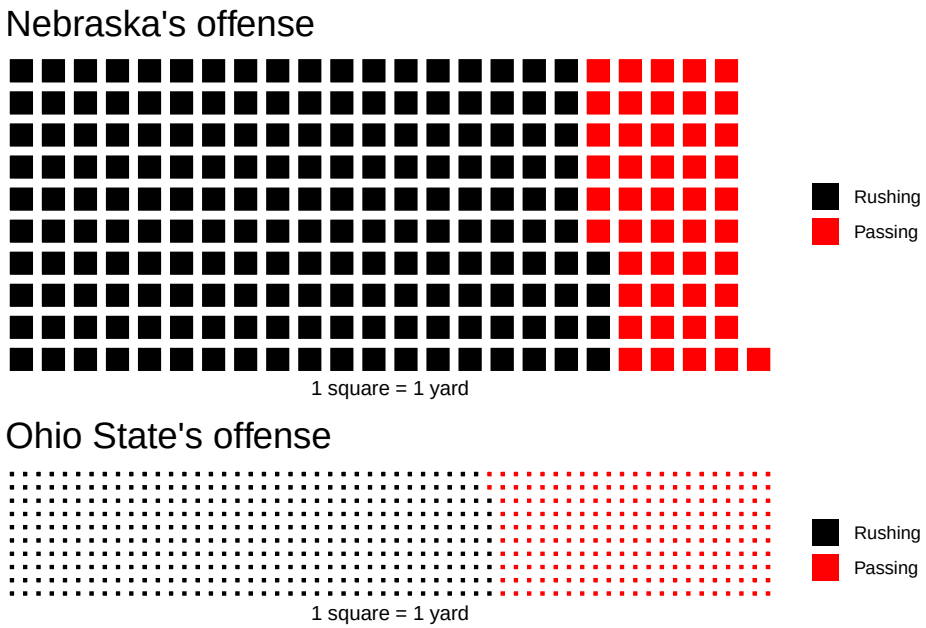
```
waffle(passing, rows = 10, title="Nebraska vs Ohio State: passing", xlab="1 square = 1 yard", col
```



15.1 Waffle Irons

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(nu, rows = 10, title="Nebraska's offense", xlab="1 square = 1 yard", colors = c
 waffle(oh, rows = 10, title="Ohio State's offense", xlab="1 square = 1 yard", colors =
)
```



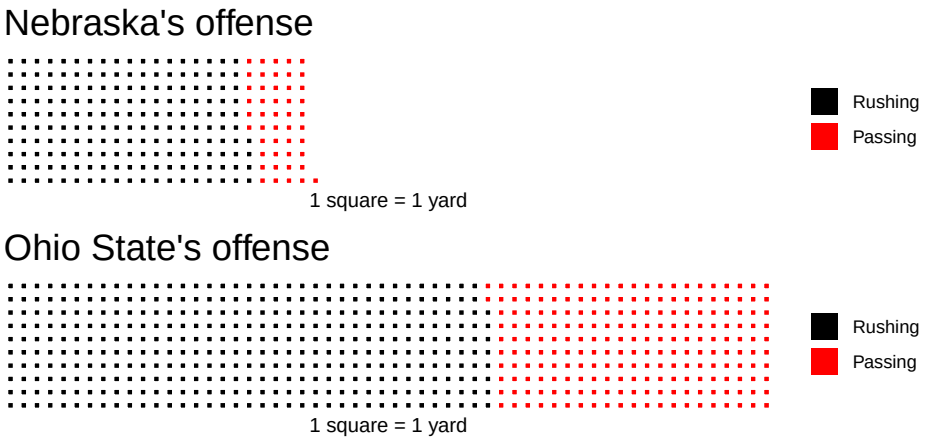
What do you notice about this chart? Notice how the squares aren't the same size? Well, Ohio State outgained Nebraska by a long way. So the squares aren't the same size because the numbers aren't the same. We can fix that by adding an unnamed padding number so the number of shots add up to the same thing. Let's make the total for everyone be 580, Ohio State's total yards of offense. So to do that, we need to add a padding of 349 to Nebraska. REMEMBER: Don't name it or it'll show up in the legend.

```
nu <- c("Rushing"=184, "Passing"=47, 349)
oh <- c("Rushing"=368, "Passing"=212, 0)
```

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.

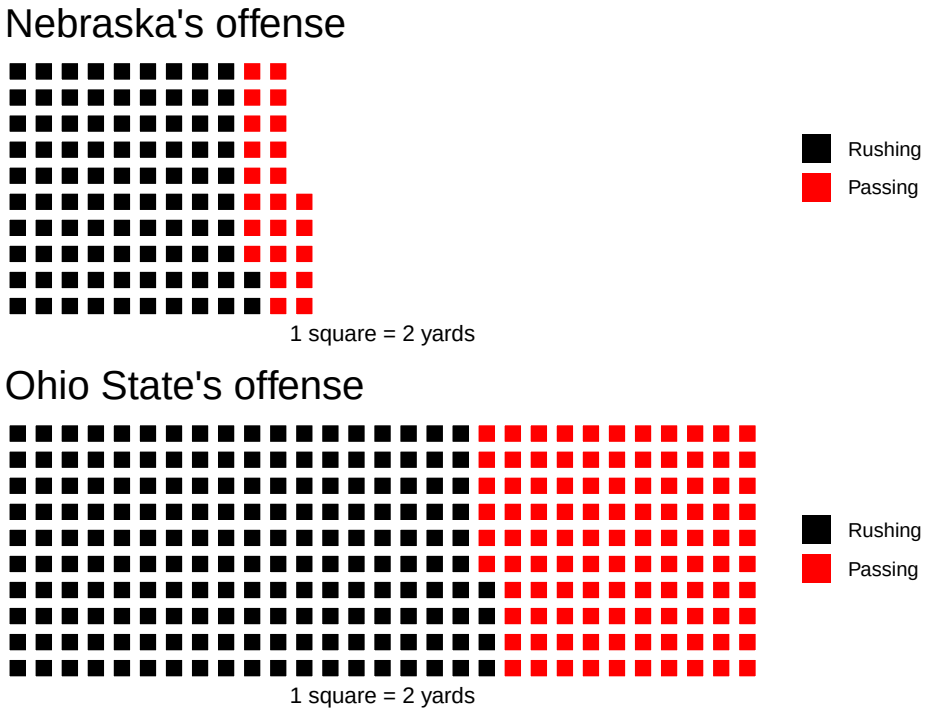
```
iron(
 waffle(nu, rows = 10, title="Nebraska's offense", xlab="1 square = 1 yard", colors = c
```

```
 waffle(oh, rows = 10, title="Ohio State's offense", xlab="1 square = 1 yard", colors = c("black"
)
```



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. Remember what you learned in Swirl about math on vectors?

```
iron(
 waffle(nu/2, rows = 10, title="Nebraska's offense", xlab="1 square = 2 yards", colors = c("black
 waffle(oh/2, rows = 10, title="Ohio State's offense", xlab="1 square = 2 yards", colors = c("bla
)
```



News flash: Ohio State crushed Nebraska.

Chapter 16

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: Why was Nebraska terrible at basketball last season?

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

```
library(tidyverse)
```

Now we'll import the data you need. Mine looks like this:

```
logs <- read_csv("data/logs19.csv")
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = ""),
##   HomeAway = col_character(),
##   Opponent = col_character(),
##   W_L = col_character(),
##   Blank = col_logical(),
##   Team = col_character(),
##   Conference = col_character(),
##   season = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

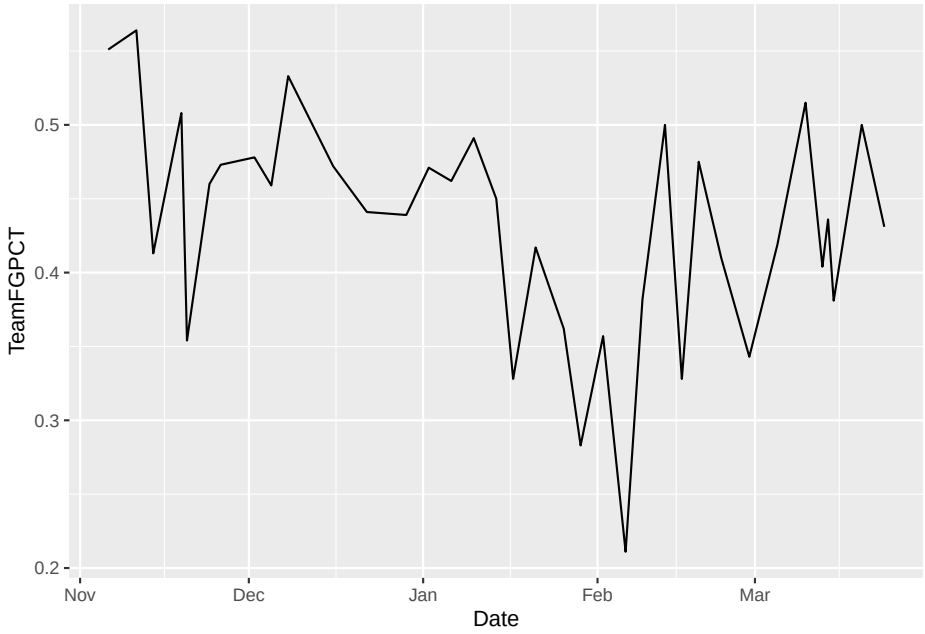
This data has every game from every team in it, so we need to use filtering to limit it, because we just want to look at Nebraska. If you don't remember, flip back to chapter 5.

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

Because this data has just Nebraska 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.

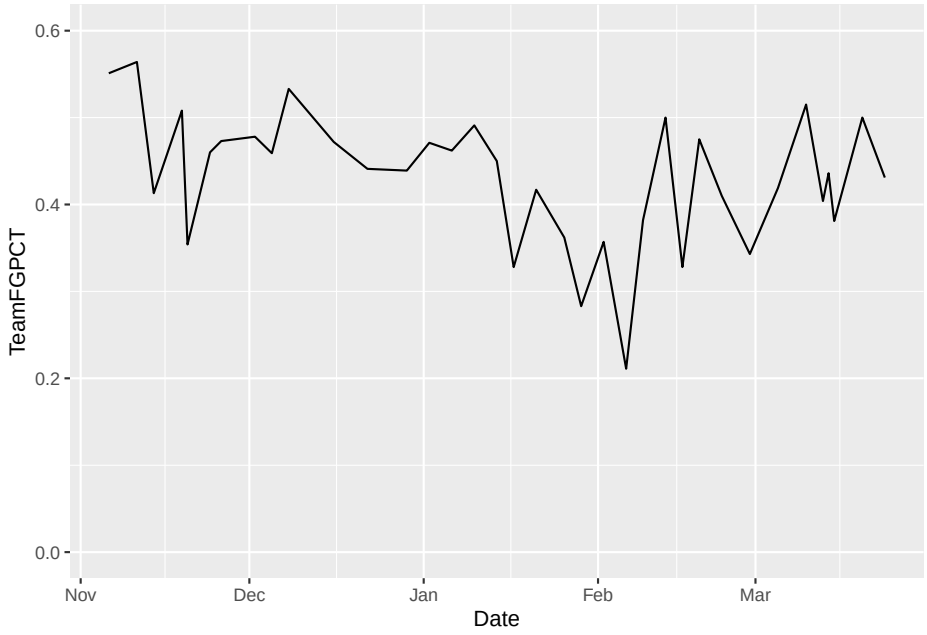
Line charts, unlike bar charts, do have a y-axis. So in our ggplot step, we have to define what our x and y axes are. In this case, the x axis is our Date – the most common x axis in line charts is going to be a date of some variety – and y in this case is up to us. We've seen from previous walkthroughs that how well a team shoots the ball has a lot to do with how well a team does in a season, so let's chart that.

```
ggplot(nu, aes(x=Date, y=TeamFGPCT)) + geom_line()
```



See a problem here? Note the Y axis doesn't start with zero. That makes this look worse than it is (and that February swoon is pretty bad). 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(nu, aes(x=Date, y=TeamFGPCT)) + geom_line() + 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.

16.1 This is too simple.

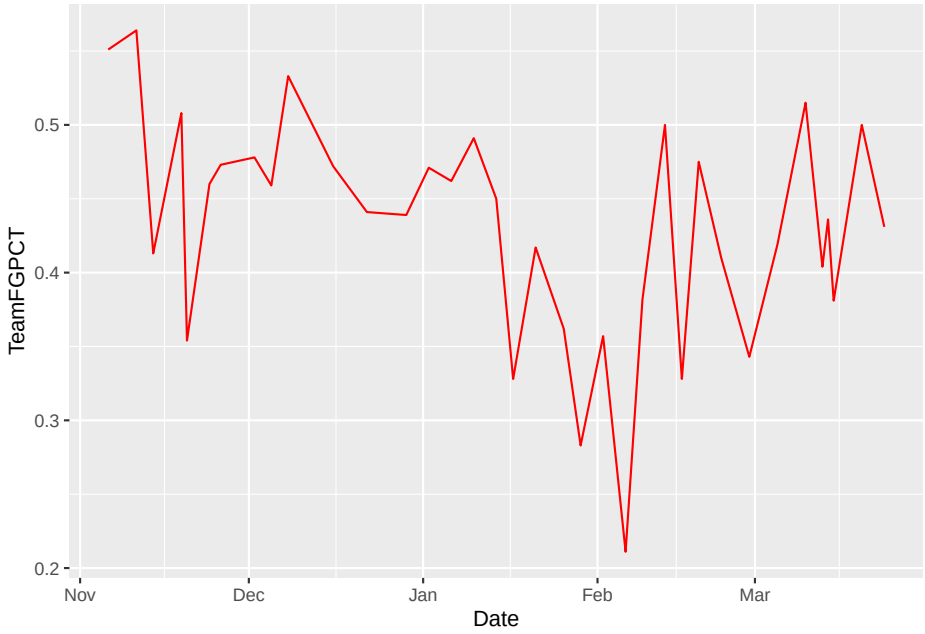
With datasets, we want to invite comparison. So let's answer the question visually. Let's put two lines on the same chart. How does Nebraska compare to Michigan State and Purdue, the eventual regular season co-champions?

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

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.

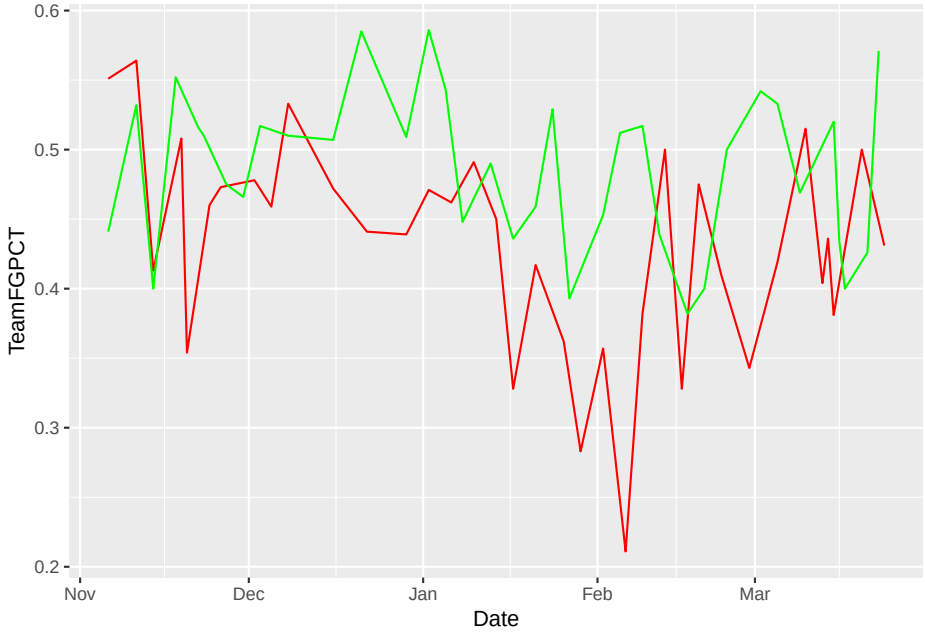
First, let's chart Nebraska. Read carefully. First we set the data. Then we set our aesthetic. Unlike bars, we need an X and a Y variable. In this case, our X is the date of the game, Y is the thing we want the lines to move with. In this case, the Team Field Goal Percentage – TeamFGPCT.

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



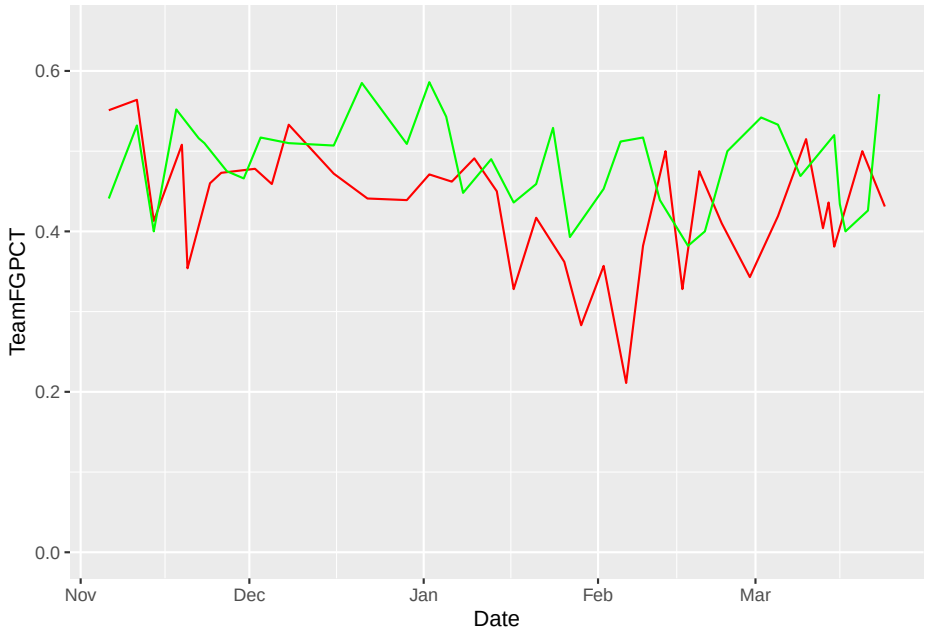
Now, by using +, we can add Michigan State to it. REMEMBER COPY AND PASTE IS A THING. Nothing changes except what data you are using.

```
ggplot() + geom_line(data=nu, aes(x=Date, y=TeamFGPCT), color="red") + geom_line(data=m
```



Let's flatten our lines out by zeroing the Y axis.

```
ggplot() + geom_line(data=nu, aes(x=Date, y=TeamFGPCT), color="red") + geom_line(data=msu, aes(x=
```



So visually speaking, the difference between Nebraska and Michigan State's season is that Michigan State stayed mostly on an even keel, and Nebraska went on a two month swoon.

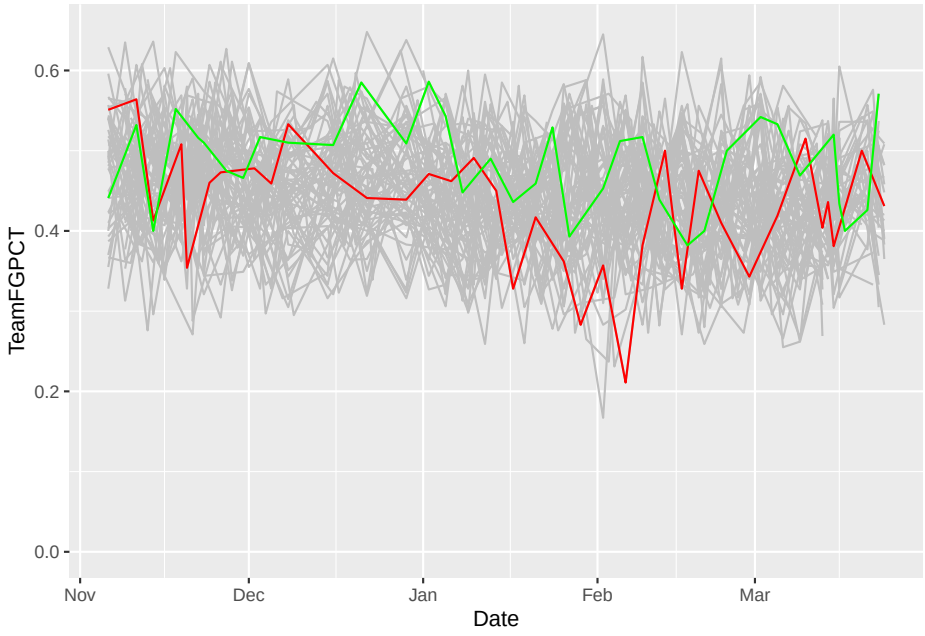
16.2 But what if I wanted to add a lot of lines.

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

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

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=TeamFGPCT, group=Team), color="grey") + geom_line
```



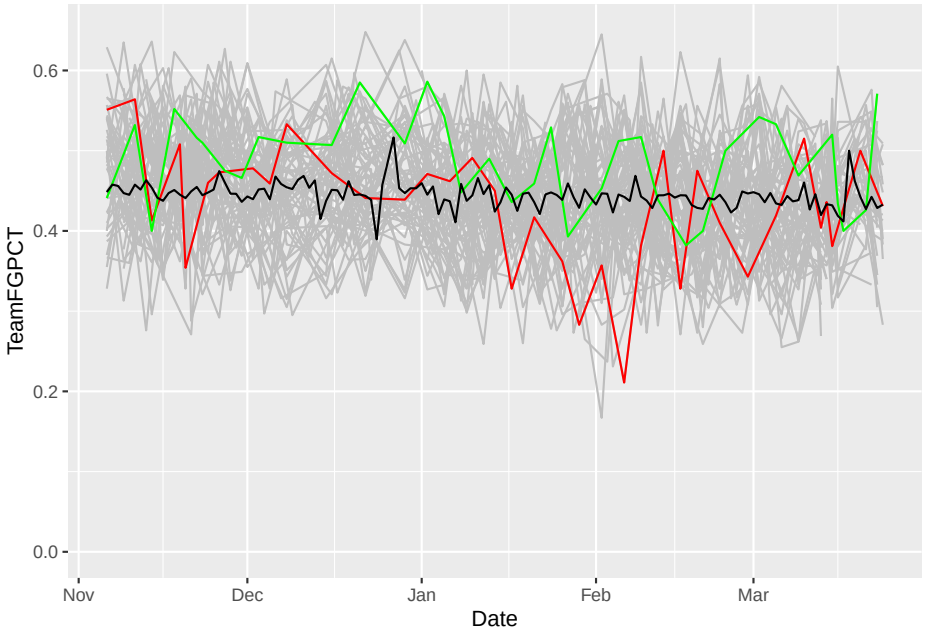
What do we see here? How has Nebraska and Michigan State's season evolved against all the rest of the teams in college basketball?

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.

```
average <- logs %>% group_by(Date) %>% summarise(mean_shooting=mean(TeamFGPCT))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
ggplot() + geom_line(data=p5conf, aes(x=Date, y=TeamFGPCT, group=Team), color="grey") +
```


0.6
0.4
0.2
0.0
TeamFGPCT
Nov
Dec
Jan
Feb
Mar
Date

Chapter 17

Step charts

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

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

To replicate this, we need cumulative data – data that is the running total of data at a given point. So think of it this way – Nebraska 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.

```
library(tidyverse)
```

And we'll use our basketball log data.

```
logs <- read_csv("data/logs19.csv")
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = ""),
##   HomeAway = col_character(),
##   Opponent = col_character(),
##   W_L = col_character(),
```

```
##   Blank = col_logical(),
##   Team = col_character(),
##   Conference = col_character(),
##   season = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

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

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

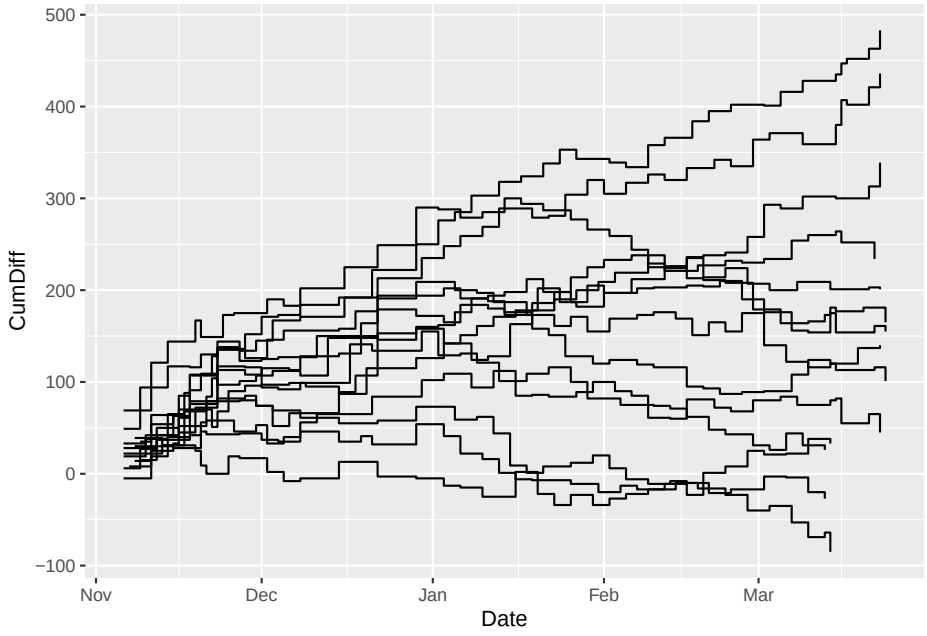
```
difflogs <- logs %>%
  mutate(Differential = TeamScore - OpponentScore) %>%
  group_by(Team) %>%
  mutate(CumDiff = cumsum(Differential))
```

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

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

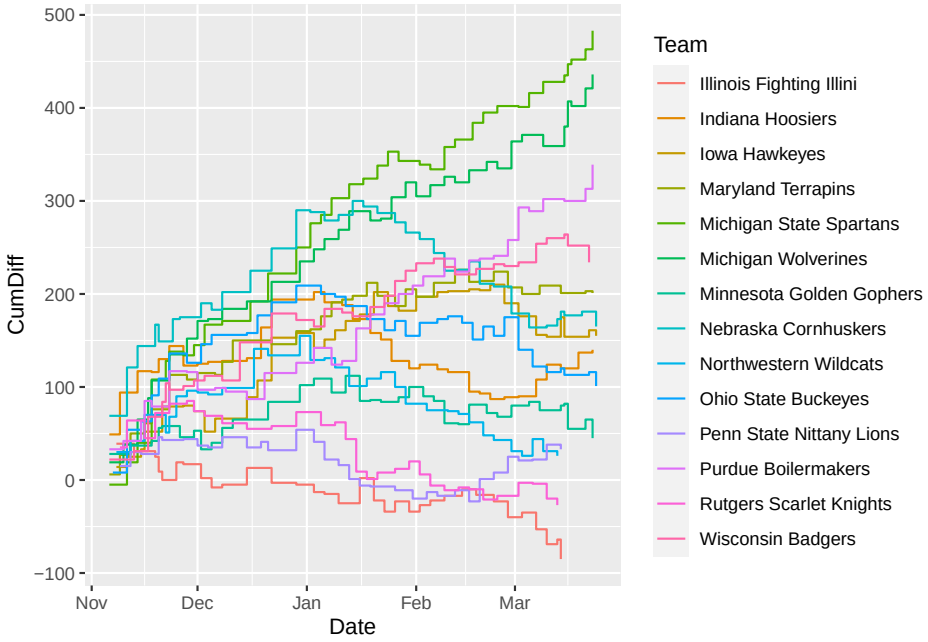
The step chart is it's own geom, so we can employ it just like we have the others. It works almost exactly the same as a line chart, but it uses the cumulative sum instead of a regular value and, as the name implies, creates a step like shape to the line instead of a curve.

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



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

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



From this, we can see two teams in the Big Ten had negative point differentials last season – Illinois and Rutgers.

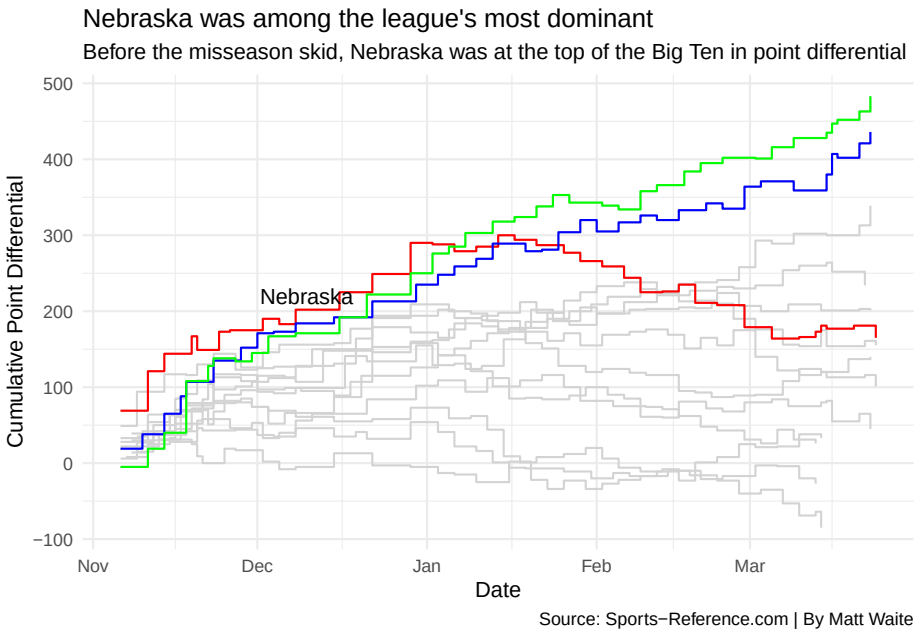
Let's look at those top teams plus Nebraska.

```
nu <- bigdiff %>% filter(Team == "Nebraska Cornhuskers")
mi <- bigdiff %>% filter(Team == "Michigan Wolverines")
ms <- bigdiff %>% filter(Team == "Michigan State Spartans")
```

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=nu, aes(x=Date, y=CumDiff, group=Team), color="red") +
  geom_step(data=mi, aes(x=Date, y=CumDiff, group=Team), color="blue") +
  geom_step(data=ms, aes(x=Date, y=CumDiff, group=Team), color="green") +
  annotate("text", x=(as.Date("2018-12-10")), y=220, label="Nebraska") +
  labs(x="Date", y="Cumulative Point Differential", title="Nebraska was among the leagu
  theme_minimal()
```


Nebraska was among the league's most dominant
Before the misseason skid, Nebraska was at the top of the Big Ten in point differential
Cumulative Point Differential
500
400
300
200
100
0
−100
Nebraska
Nov
Dec
Jan
Feb
Mar
Date
Source: Sports−Reference.com | By Matt Waite

Chapter 18

Ridge charts

Ridgeplots are useful for when you want to show how different groupings compare with a large number of datapoints. So let's look at how we do this, and in the process, we learn about ggplot extensions. The extensions add functionality to ggplot, which doesn't out of the box have ridgeplots (sometimes called joyplots).

In the console, run this: `install.packages("ggridges")`

Now we can add those libraries.

```
library(tidyverse)
library(ggridges)
```

So for this, let's look at every basketball game played since the 2014-15 season. That's more than 28,000 basketball games. Download that data here.

```
logs <- read_csv("data/logs1519.csv")
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = ""),
##   HomeAway = col_character(),
##   Opponent = col_character(),
##   W_L = col_character(),
##   Blank = col_logical(),
##   Team = col_character(),
##   Conference = col_character(),
##   season = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

So I want to group teams by wins. Wins are the only thing that matter – ask Tim Miles. So our data has a column called W_L that lists if the team won or lost. The problem is it doesn't just say W or L. If the game went to overtime, it lists that. That complicates counting wins. And, with ridgeplots, I want to be be able to separate EVERY GAME by how many wins the team had over a SEASON. So I've got some work to do.

First, here's a trick to find a string of text and make that. It's called `grepl` and the basic syntax is grepl for this string in this field and then do what I tell you. In this case, we're going to create a new field called winloss look for W or L (and ignore any OT notation) and give wins a 1 and losses a 0.

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

Now I'm going to add up all the winlosses for each team, which should give me the number of wins for each team.

```
winlosslogs %>% group_by(Team, Conference, season) %>% summarise(TeamWins = sum(winloss
```

```
## `summarise()` regrouping output by 'Team', 'Conference' (override with `.groups` arg
```

Now that I have season win totals, I can join that data back to my log data so each game has the total number of wins in each season.

```
winlosslogs %>% left_join(teamseasonwins, by=c("Team", "Conference", "season")) -> winl
```

Now I can use that same `case_when` logic to create some groupings. So I want to group teams together by how many wins they had over the season. For no good reason, I started with more than 25 wins, then did groups of 5 down to 10 wins. If you had fewer than 10 wins, God help your program.

The way to create a new field based on groupings like that is to use `case_when`, which says, basically, when This Thing Is True, Do This. So in our case, we're going to pass a couple of logical statements that when they are both true, our data gets labeled how we want to label it. So we `mutate` a field called grouping and then use `case_when`.

```
wintotallogs %>% mutate(grouping = case_when(
  TeamWins > 25 ~ "More than 25 wins",
  TeamWins >= 20 & TeamWins <=25 ~ "20-25 wins",
  TeamWins >= 15 & TeamWins <=19 ~ "15-19 wins",
  TeamWins >= 10 & TeamWins <=14 ~ "10-14 wins",
  TeamWins < 10 ~ "Less than 10 wins")
) -> wintotalgroupinglogs
```

So my `wintotalgroupinglogs` table has each game, with a field that gives the

total number of wins each team had that season and labeling each game with one of five groupings. I could use `dplyr` to do group_by on those five groups and find some things out about them, but ridgeplots do that visually.

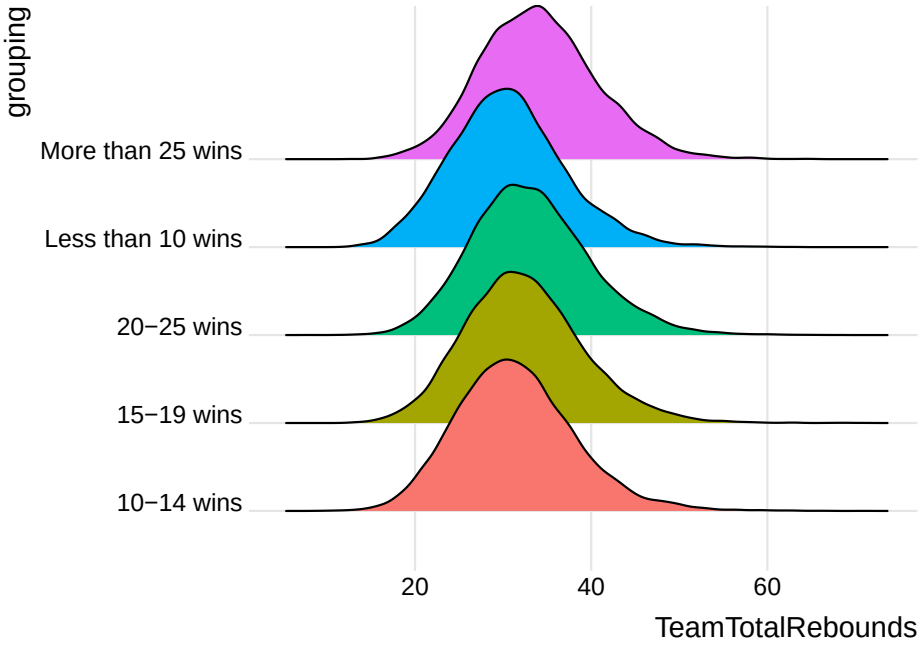
Let's look at the differences in rebounds by those five groups. Do teams that win more than 25 games rebound better than teams that win fewer games?

The answer might surprise you.

```
ggplot(wintotalgroupinglogs, aes(x = TeamTotalRebounds, y = grouping, fill = grouping)) +
  geom_density_ridges() +
  theme_ridges() +
  theme(legend.position = "none")
```

```
## Picking joint bandwidth of 0.88
```

```
## Warning: Removed 2 rows containing non-finite values (stat_density_ridges).
```



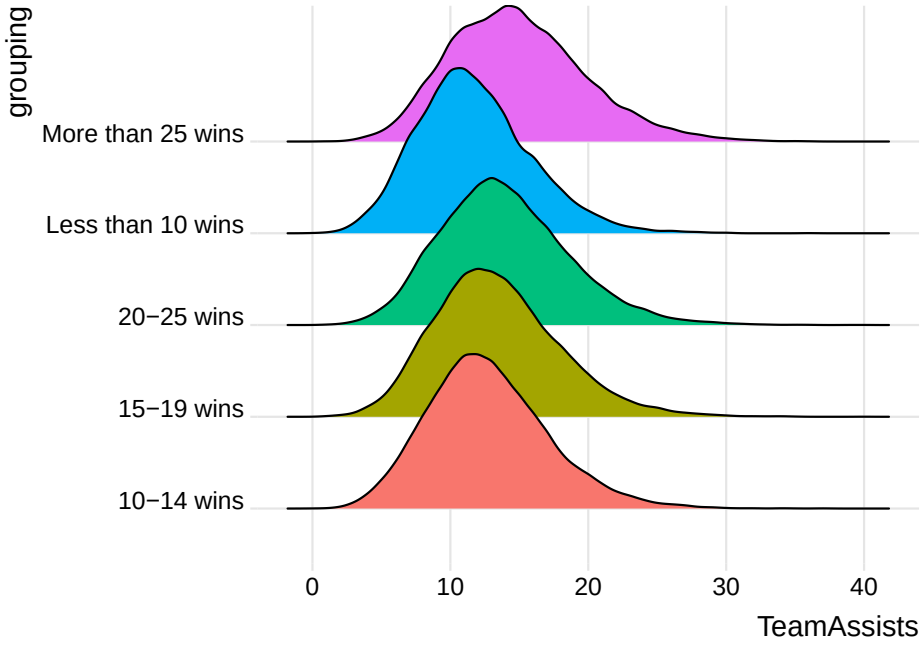
Answer? Not really. Game to game, maybe. Over five seasons? The differences are indistinguishable.

How about assists?

```
ggplot(wintotalgroupinglogs, aes(x = TeamAssists, y = grouping, fill = grouping)) +
  geom_density_ridges() +
  theme_ridges() +
  theme(legend.position = "none")
```

```
## Picking joint bandwidth of 0.601
```

```
## Warning: Removed 2 rows containing non-finite values (stat_density_ridges).
```

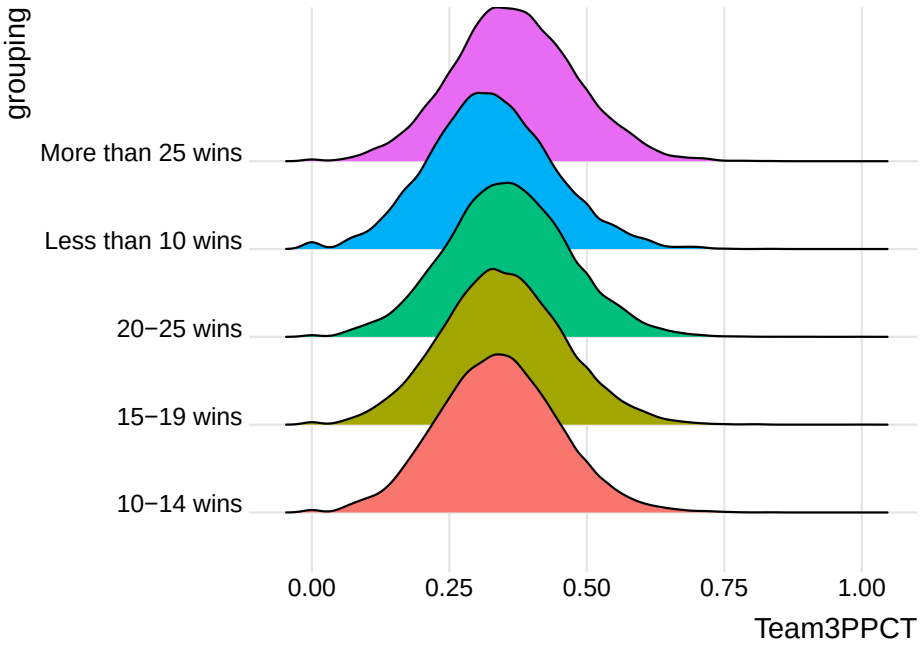


There's a little better, especially between top and bottom.

```
ggplot(wintotalgroupinglogs, aes(x = Team3PPCT, y = grouping, fill = grouping)) +
  geom_density_ridges() +
  theme_ridges() +
  theme(legend.position = "none")
```

```
## Picking joint bandwidth of 0.0156
```

```
## Warning: Removed 2 rows containing non-finite values (stat_density_ridges).
```

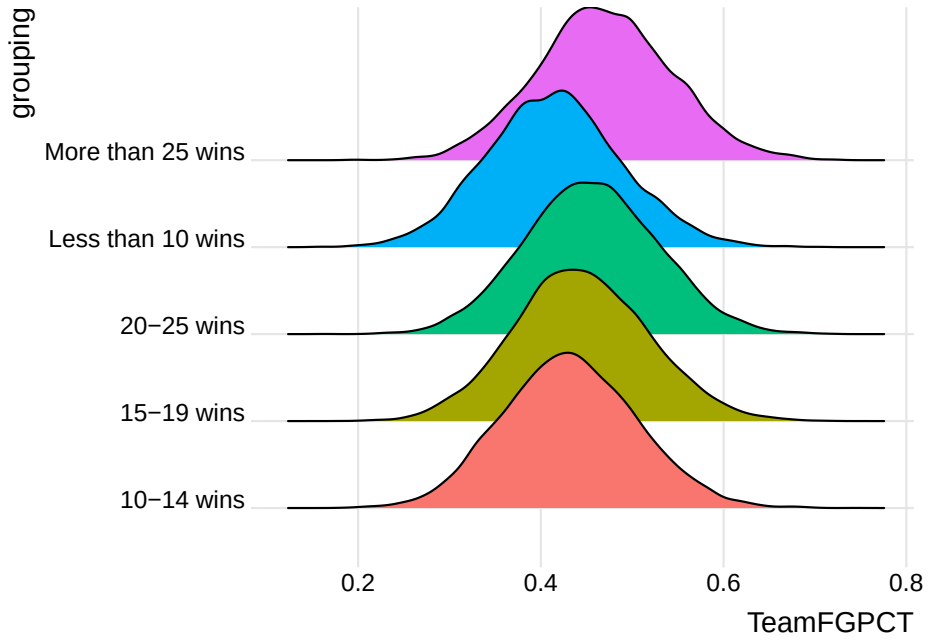


If you've been paying attention this semester, you know what's coming next.

```
ggplot(wintotalgroupinglogs, aes(x = TeamFGPCT, y = grouping, fill = grouping)) +
  geom_density_ridges() +
  theme_ridges() +
  theme(legend.position = "none")
```

```
## Picking joint bandwidth of 0.0102
```

```
## Warning: Removed 2 rows containing non-finite values (stat_density_ridges).
```


grouping
More than 25 wins
Less than 10 wins
20−25 wins
15−19 wins
10−14 wins
0.2
0.4
0.6
0.8
TeamFGPCT

Chapter 19

Lollipop charts

Second to my love of waffle charts because I'm always hungry, lollipop 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. They aren't a geom, specifically, but you can assemble them out of points and segments, which are geoms.

```
library(tidyverse)
```

```
logs <- read_csv("data/logs19.csv")
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = ""),
##   HomeAway = col_character(),
##   Opponent = col_character(),
##   W_L = col_character(),
##   Blank = col_logical(),
##   Team = col_character(),
##   Conference = col_character(),
##   season = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

For the first example, let's look at the difference between a team's shooting performance and the conference's shooting performance as a whole. To get this, we're going to add up all the shots made by the conference, all the attempts taken by the conference, and then mutate a percentage based on that.

```
conferenceshooting <- logs %>%
  group_by(Conference) %>%
  summarise(totalshots = sum(TeamFG), totalattempts = sum(TeamFGA)) %>%
  mutate(conferenceshootingpct = totalshots/totalattempts)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

Now I'm going to do the same with teams.

```
teamshooting <- logs %>%
  group_by(Team, Conference) %>%
  summarise(totalshots = sum(TeamFG), totalattempts = sum(TeamFGA)) %>%
  mutate(teamshootingpct = totalshots/totalattempts)
```

```
## `summarise()` regrouping output by 'Team' (override with `.groups` argument)
```

The last thing I need to do is join them together. So each team will have the conference shooting percentage as well as their own.

```
shooting <- teamshooting %>% left_join(conferenceshooting, by="Conference")
```

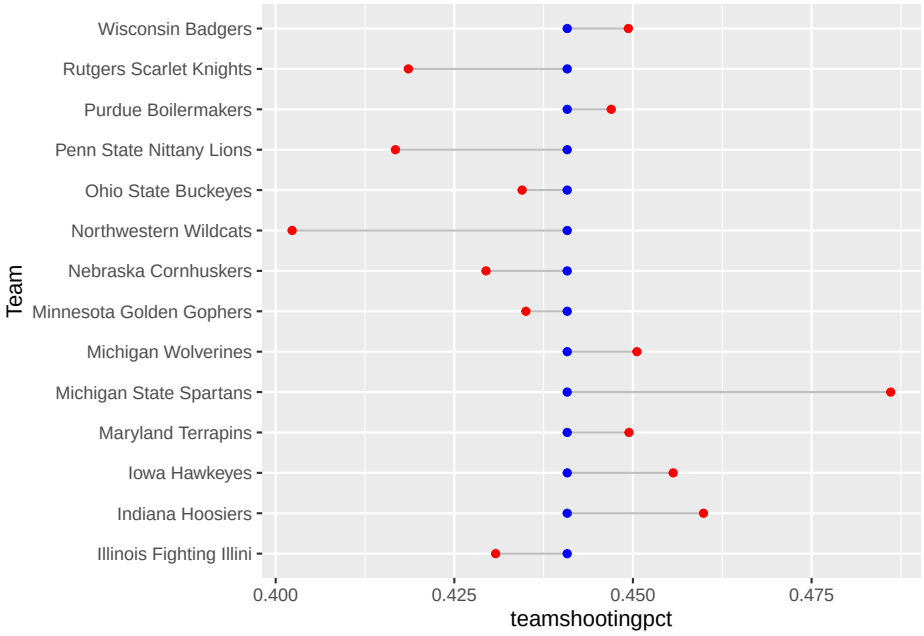
I have every team in college basketball, but that's insane.

```
big10 <- shooting %>% filter(Conference == "Big Ten")
```

So this takes a little doing, but the logic is pretty clear in the end.

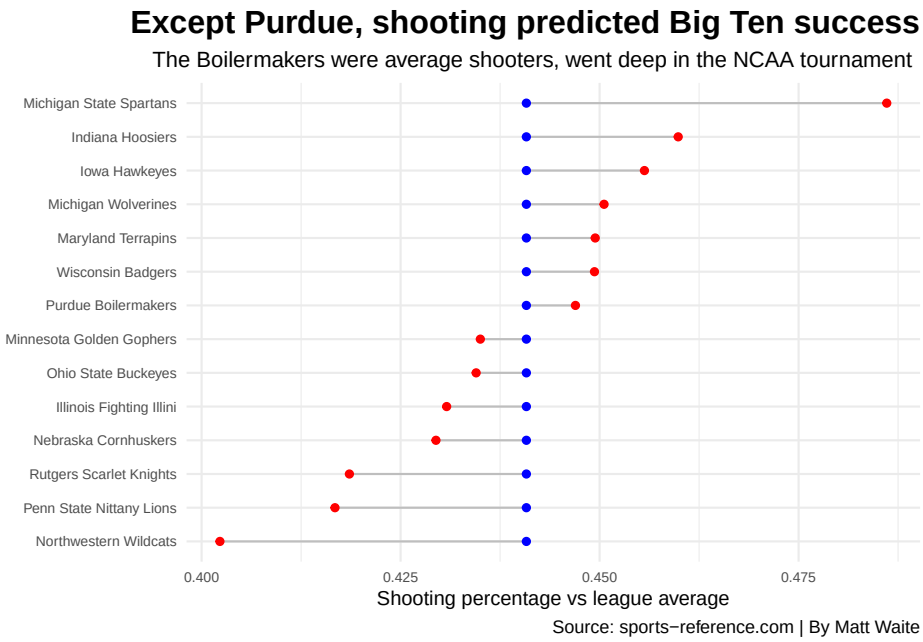
A lollipop chart is made up of two things – a line between two points, and two points. So we need a geom_segment and two geom_points. And because they get layered starting at the bottom, our segment is first. A geom segment is made up of two things – an x and a y value, and an x and y end. In this case, our x and xend are the same – the Team – and our y and yend are our two stats. For our points, both x values are the Team and the y is the different stats. What that does is put each point on the same line.

```
ggplot(big10) +
  geom_segment(aes(x=Team, xend=Team, y=teamshootingpct, yend=conferenceshootingpct), c
  geom_point(aes(x=Team, y=teamshootingpct), color="red") +
  geom_point(aes(x=Team, y=conferenceshootingpct), color="blue") +
  coord_flip()
```



We can do better by changing the order of the teams by their shooting performance and giving it some theme love.

```
ggplot(big10) +
  geom_segment(aes(x=reorder(Team, teamshootingpct), xend=Team, y=teamshootingpct, yend=conferenc
  geom_point(aes(x=reorder(Team, teamshootingpct), y=teamshootingpct), color="red") +
  geom_point(aes(x=reorder(Team, teamshootingpct), y=conferenceshootingpct), color="blue") +
  coord_flip() +
   labs(x="", y="Shooting percentage vs league average", title="Except Purdue, shooting predicted
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold", hjust = 1),
    plot.subtitle = element_text(hjust = 1.3),
    axis.title = element_text(size = 10),
    axis.title.y = element_blank(),
    axis.text = element_text(size = 7),
    axis.ticks = element_blank()
  )
```



What if we wanted to order them by wins? Our data has a column called W_L that lists if the team won or lost. The problem is it doens't just say W or L. If the game went to overtime, it lists that. That complicates counting wins. Here's a trick to find a string of text and make that. It's called `grepl` and the basic syntax is grepl for this string in this field and then do what I tell you. In this case, we're going to create a new field called winloss look for W or L (and ignore any OT notation) and give wins a 1 and losses a 0.

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

So let's look at turnovers and assists. We'll call it give and take. Does the difference between those two things indicate something when we sort them by wins?

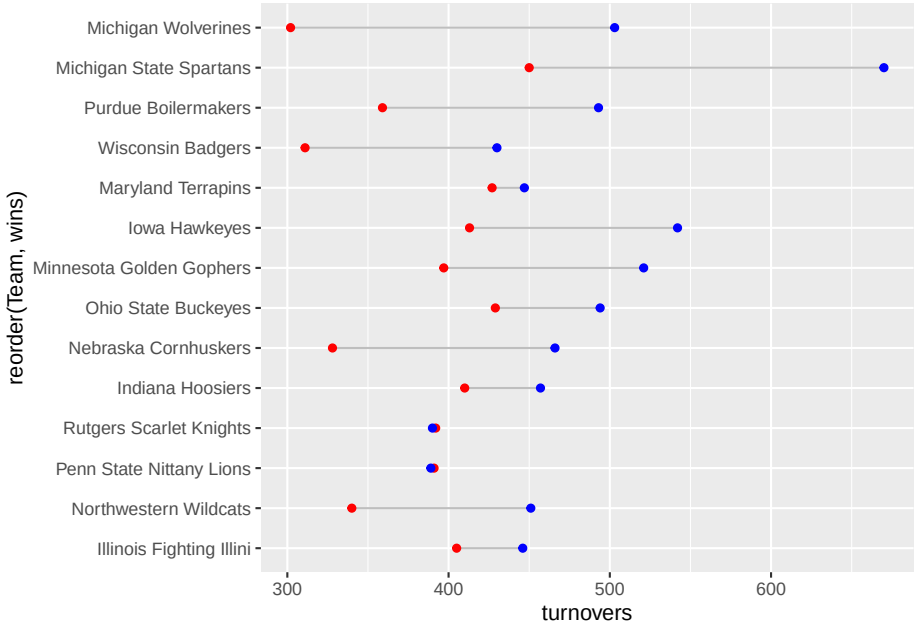
```
giveandtake <- winlosslogs %>% group_by(Conference, Team) %>% summarise(turnovers = su
```

```
## `summarise()` regrouping output by 'Conference' (override with `.groups` argument)
```

```
big10gt <- giveandtake %>% filter(Conference == "Big Ten")
```

```
ggplot(big10gt) +
  geom_segment(aes(x=reorder(Team, wins), xend=Team, y=turnovers, yend=assists), color=
  geom_point(aes(x=reorder(Team, wins), y=turnovers), color="red") +
  geom_point(aes(x=reorder(Team, wins), y=assists), color="blue") +
```

```
coord_flip()
```



Short answer: Not really. Something you need to get used to in data visualization – not everything works. Sometimes you find things, sometimes you don't. Don't publish a graphic that doesn't find anything. Let it stay in your notebook as an idea that didn't pan out.

Chapter 20

Scatterplots

In several chapters of this book, we've been fixated on the Nebraska basketball team's shooting percentage, which took a nose dive during the season and ultimately doomed Tim Miles job. The question is … does it matter?

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

First, we need libraries and data.

```
library(tidyverse)
```

```
logs <- read_csv("data/logs19.csv")
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = ""),
##   HomeAway = col_character(),
##   Opponent = col_character(),
##   W_L = col_character(),
##   Blank = col_logical(),
##   Team = col_character(),
##   Conference = col_character(),
##   season = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

To do this, we need all teams and their season stats. 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 shooting percentage influence wins? How much difference can we explain in wins with shooting percentage? We're going to total up the number of wins each team has and their season shooting percentage in one swoop.

Let's borrow from our ridgecharts work to get the correct wins and losses totals for each team.

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

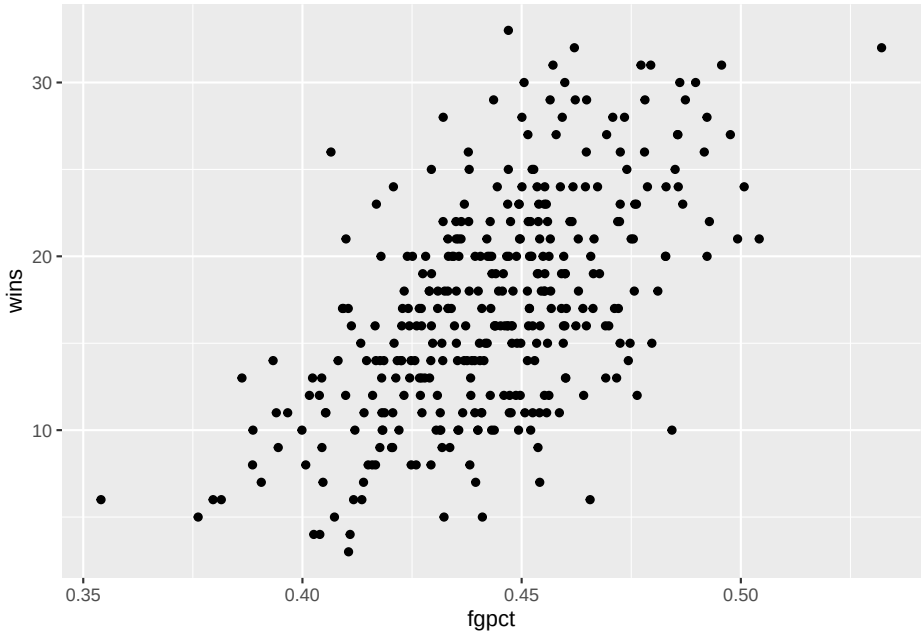
Now we can get a dataframe together that gives us the total wins for each team, and the total shots taken and made, which let's us calculate a season shooting percentage.

```
winlosslogs %>%
  group_by(Team) %>%
  summarise(
    wins = sum(winloss),
    totalFGAttempts = sum(TeamFGA),
    totalFG = sum(TeamFG)
  ) %>%
  mutate(fgpct = totalFG/totalFGAttempts) -> fgmodel
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

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

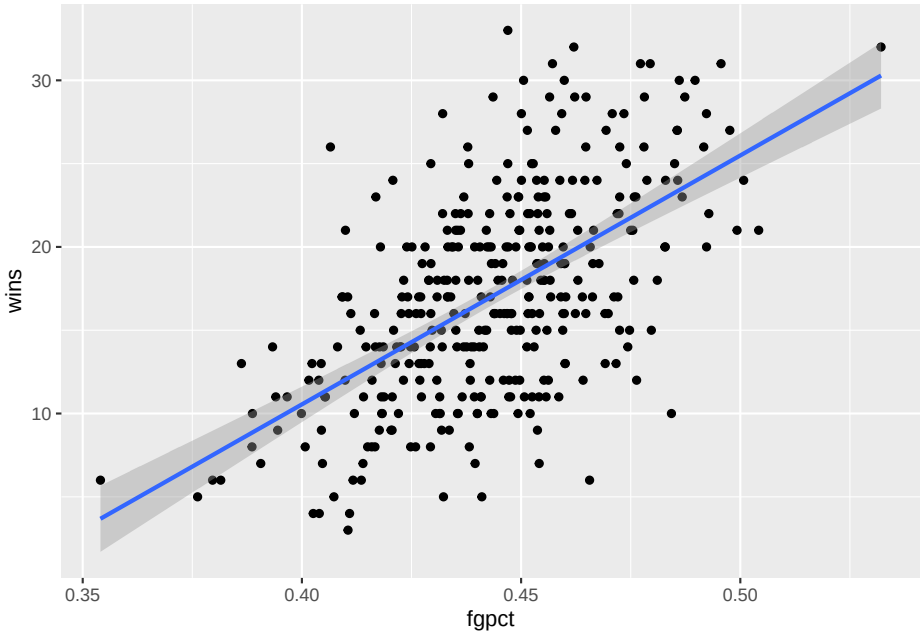
```
ggplot(fgmodel, aes(x=fgpct, y=wins)) + geom_point()
```



Let's talk about this. It seems that the data slopes up to the right. That would indicate a positive correlation between shooting percentage and wins. And that makes sense, no? You'd expect teams that shoot the ball well to win. But can we get a better sense of this? Yes, by adding another geom – `geom_smooth`.

```
ggplot(fgmodel, aes(x=fgpct, y=wins)) + geom_point() + geom_smooth(method=lm, se=TRUE)
```

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



But ... how strong a relationship is this? How much can shooting percentage 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 field goal percentage". Then we just want the summary of that model.

```
fit <- lm(wins ~ fgpct, data = fgmodel)
summary(fit)
```

```
##
## Call:
## lm(formula = wins ~ fgpct, data = fgmodel)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -14.3536  -3.4523  -0.1125   3.3834  15.4318
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -49.217      4.845  -10.16   <2e-16 ***
## fgpct        149.416     10.915   13.69   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## Residual standard error: 5.035 on 351 degrees of freedom
## Multiple R-squared:  0.348,Adjusted R-squared:  0.3462
## F-statistic: 187.4 on 1 and 351 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 .3462, which we can interpret as shooting percentage predicts about 35 percent of the variance in wins. Which sounds not great, but in social science, that's huge. That's great. A psychology major would murder for that R-squared.

Second: The P-value. We want anything less than .05. If it's above .05, the change between them is not statistically significant – it's probably explained by random chance. In our case, we have 2.2e-16, which is to say 2.2 with 16 zeros in front of it, or .000000000000000022. Is that less than .05? Yes. Yes it is. So this is not random. Again, we would expect this, so it's a good logic test.

Third: The coefficient. In this case, the coefficient for fgpct is 149.416. Since I didn't convert percentages to decimals, what this says is that for every percentage point improvement in shooting percentage, we can expect the team to win 1.49 more games plus or minus some error.

And we can use this to predict a team's wins: remember your algebra and y = mx + b. In this case, y is the wins, m is the coefficient, x is the shooting percentage and b is the intercept.

So can plug these together: Expected wins = 149.416 * shooting percentage - 49.217

Let's use Nebraska as an example. They shot about .43 on the season (.4294421 to be exact).

y = 149.416 * .4294421 - 49.217 or 14.95 wins. How many wins did Nebraska have? 19.

What does that mean? It means that as disappointing a season as it was, Nebraska actually OVERPERFORMED it's season shooting percentage. They shouldn't have won as many games as they did, according to our model.

Chapter 21

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)
```

Now data.

```
logs <- read_csv("data/logs19.csv")
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = ""),
##   HomeAway = col_character(),
##   Opponent = col_character(),
##   W_L = col_character(),
##   Blank = col_logical(),
##   Team = col_character(),
##   Conference = col_character(),
##   season = col_character()
## )
```

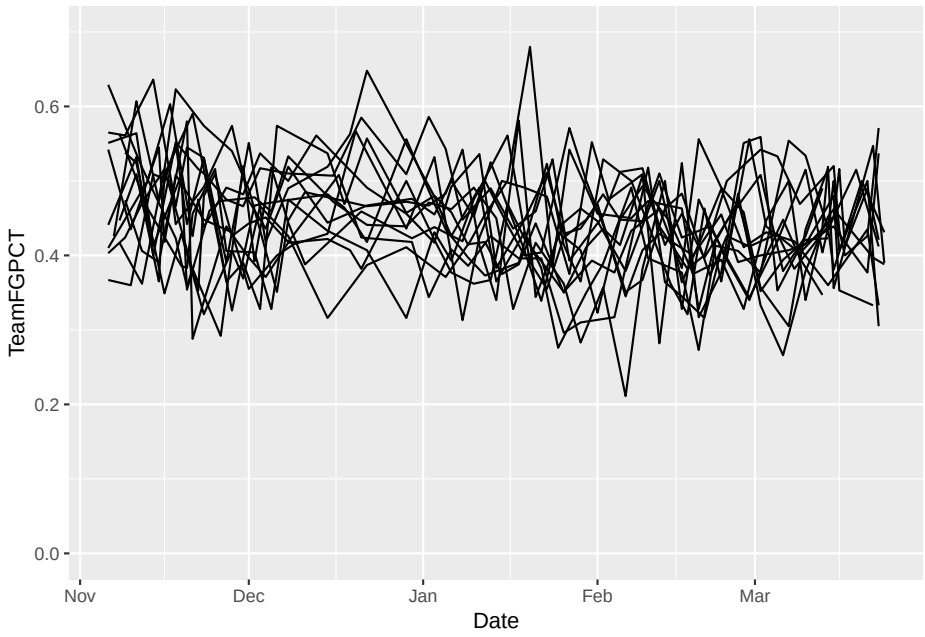
```
## See spec(...) for full column specifications.
```

Let's narrow our pile and look just at the Big Ten.

```
big10 <- logs %>% filter(Conference == "Big Ten")
```

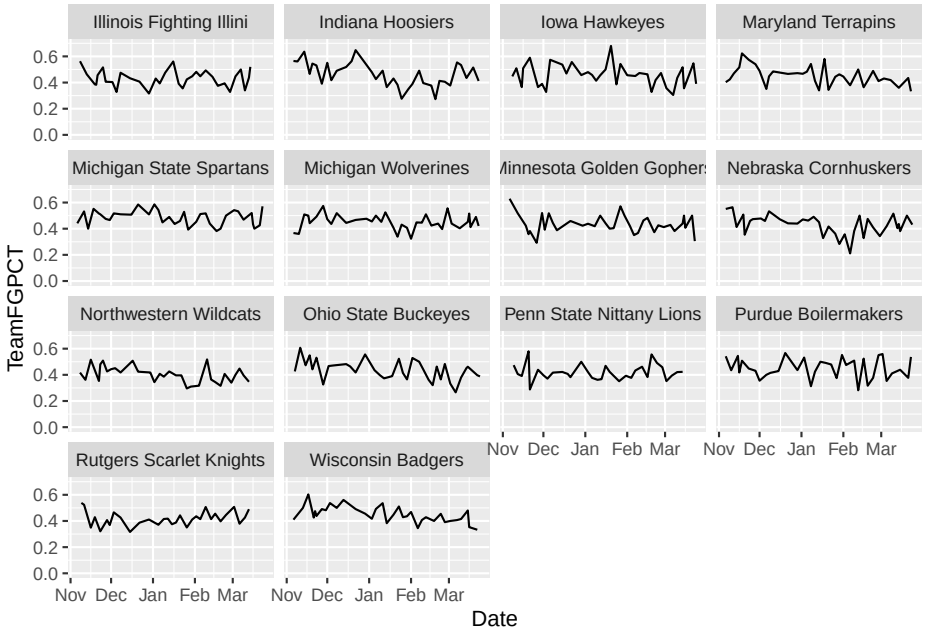
The first thing we can do is look at a line chart, like we have done in previous chapters.

```
ggplot() + geom_line(data=big10, aes(x=Date, y=TeamFGPCT, group=Team)) + scale_y_contin
```



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_contin
```



Answer: Not immediately clear, but we can look at this and analyze it. We could add a piece of annotation to help us out.

```
big10 %>% summarise(mean(TeamFGPCT))
```

```
## # A tibble: 1 x 1
##   `mean(TeamFGPCT)`
##               <dbl>
## 1             0.442
```

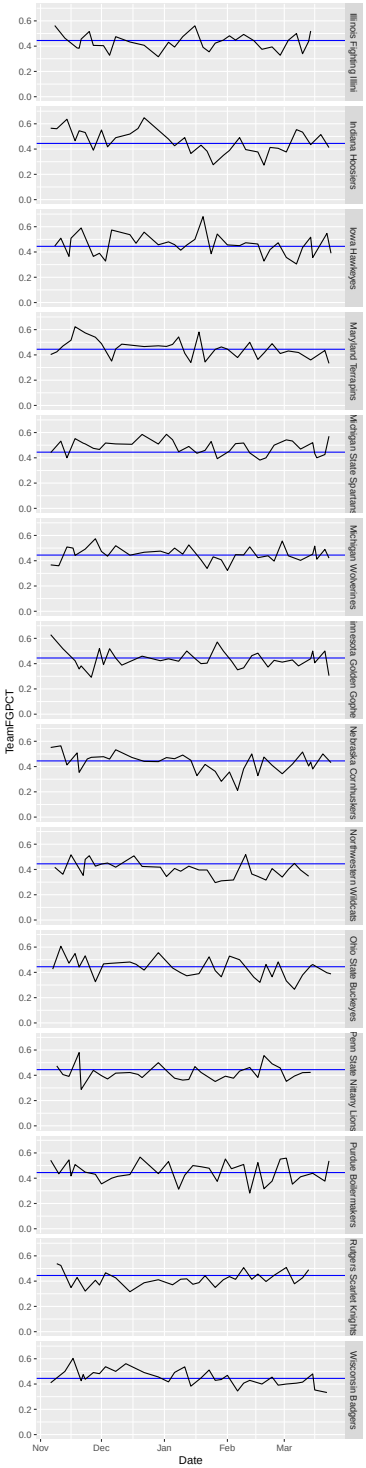
```
ggplot() + geom_hline(yintercept=.4447, color="blue") + geom_line(data=big10, aes(x=Date, y=TeamF
```

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.

21.1 Facet grid vs facet wraps

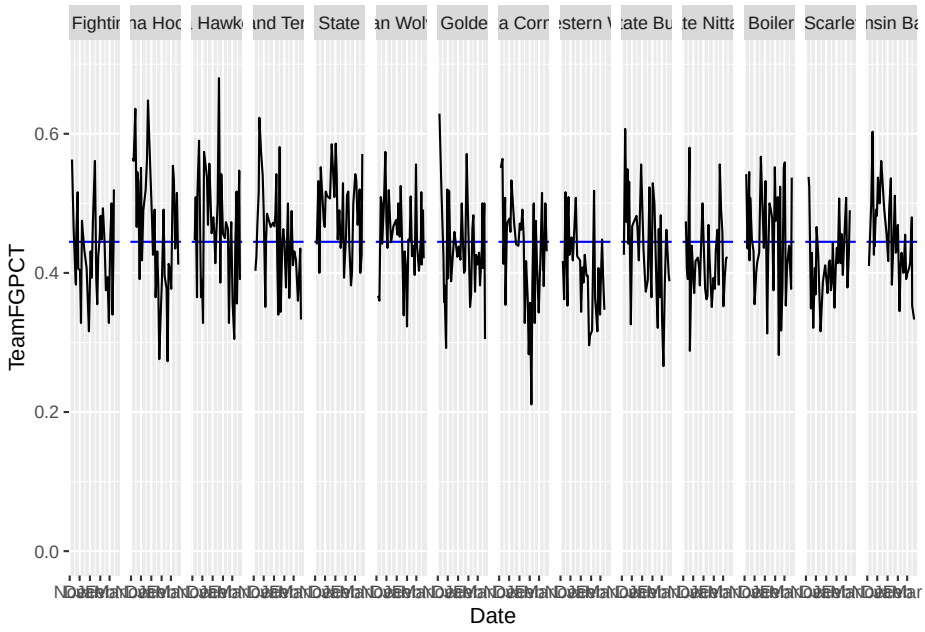
Facet grids allow us to put teams on the same plane, versus just repeating them. And we can specify that plane as vertical or horizontal. For example, here's our chart from above, but using facet_grid to stack them.

```
ggplot() + geom_hline(yintercept=.4447, color="blue") + geom_line(data=big10, aes(x=Dat
```


Illinois Fighting Illini
Indiana Hoosiers
Iowa Hawkeyes
Maryland Terrapins
Michigan State Spartans
Michigan Wolverines
Minnesota Golden Gophe
Nebraska Cornhuskers
Northwestern Wildcats
Ohio State Buckeyes
Penn State Nittany Lions
Purdue Boilermakers
Rutgers Scarlet Knights
Wisconsin Badgers
TeamFGPCT
0.0
0.2
0.4
0.6
Nov
Dec
Jan
Feb
Mar
Date

And here they are next to each other:

```
ggplot() + geom_hline(yintercept=.4447, color="blue") + geom_line(data=big10, aes(x=Dat
```

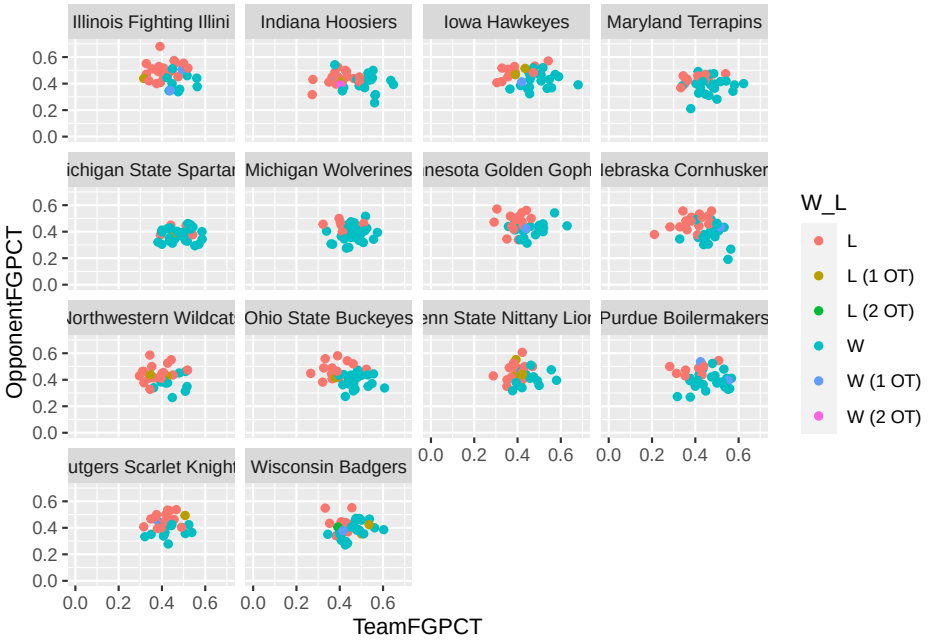


Note: We'd have some work to do with the labeling on this – we'll get to that – but you can see where this is valuable comparing a group of things. One warning: Don't go too crazy with this or it loses it's visual power.

21.2 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
```


Illinois Fighting Illini
Indiana Hoosiers
Iowa Hawkeyes
Maryland Terrapins
ichigan State Spartar
Michigan Wolverines
nesota Golden Goph
lebraska Cornhusker
Northwestern Wildcat
Ohio State Buckeyes
enn State Nittany Lior
Purdue Boilermakers
utgers Scarlet Knight
Wisconsin Badgers
OpponentFGPCT
TeamFGPCT
0.0
0.2
0.4
0.6
W_L
L
L (1 OT)
L (2 OT)
W
W (1 OT)
W (2 OT)

Chapter 22

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. R has a neat package called `kableExtra`.

For this assignment, we're going to need a bunch of new libraries. Go over to the console and run these:

```
install.packages("kableExtra")
install.packages("formattable")
install.packages("htmltools")
install.packages("webshot")
webshot::install_phantomjs()
```

So what does all of these libraries do? Let's gather a few and get some data.

```
library(tidyverse)
library(kableExtra)
```

```
offense <- read_csv("data/offensechange.csv")
```

```
## Parsed with column specification:
## cols(
##   Year = col_double(),
##   Name = col_character(),
##   G = col_double(),
##   `Rush Yards` = col_double(),
##   `Pass Yards` = col_double(),
##   Plays = col_double(),
##   `Total Yards` = col_double(),
```

```
##   `Yards/Play` = col_double(),
##   `Yards/G` = col_double()
## )
```

Let's ask this question: Which college football team saw the greatest improvement in yards per game last regular season? The simplest way to calculate that is by percent change.

```
changeTotalOffense <- offense %>%
  select(Name, Year, `Yards/G`) %>% # picking what we need.
  pivot_wider(names_from = Year, values_from = `Yards/G`) %>% # making it wide
  mutate(Change=(`2019`-`2018`)/`2018`) %>% # so we can calculate percent change
  arrange(desc(Change)) %>%
  top_n(10) # just want a top 10 list
```

```
## Selecting by Change
```

We've output tables to the screen a thousand times in this class with `head`, but kable makes them look decent with very little code.

```
changeTotalOffense %>% kable()
```

Name

2019

2018

Change

Central Michigan

433.6

254.7

0.7023950

LSU

564.1

402.1

0.4028849

UTSA

344.9

247.1

0.3957912

San Jose State

427.4

323.7

0.3203584

Navy

455.8

349.3

0.3048955

Louisville

447.3

352.6

0.2685763

SMU

489.8

387.2

0.2649793

BYU

443.8

364.9

0.2162236

New Mexico

400.3

330.0

0.2130303

Charlotte

411.8

343.1

0.2002332

So there you have it. Central Michigan improved the most (but look at who came in second!). Kable has a mountain of customization options. The good news is that it works in a very familiar pattern. We'll start with default styling.

```
changeTotalOffense %>%
  kable() %>%
  kable_styling()
```

Name

2019

2018

Change

Central Michigan

433.6

254.7

0.7023950

LSU

564.1

402.1

0.4028849

UTSA

344.9

247.1

0.3957912

San Jose State

427.4

323.7

0.3203584

Navy

455.8

349.3

0.3048955

Louisville

447.3

352.6

0.2685763

SMU

489.8

387.2

0.2649793

BYU

443.8

364.9

0.2162236

New Mexico

400.3

330.0

0.2130303

Charlotte

411.8

343.1

0.2002332

Let's do more than the defaults, which you can see are pretty decent. Let's stripe every other row with a little bit of grey, and let's smush the width of the rows.

```
changeTotalOffense %>%
  kable() %>%
  kable_styling(bootstrap_options = c("striped", "condensed"))
```

Name

2019

2018

Change

Central Michigan

433.6

254.7

0.7023950

LSU

564.1

402.1

0.4028849

UTSA

344.9

247.1

0.3957912

San Jose State

427.4

323.7

0.3203584

Navy

455.8

349.3

0.3048955

Louisville

447.3

352.6

0.2685763

SMU

489.8

387.2

0.2649793

BYU

443.8

364.9

0.2162236

New Mexico

400.3

330.0

0.2130303

Charlotte

411.8

343.1

0.2002332

Throughout the semester, we've been using color and other signals to highlight things. Let's pretend we're doing a project on LSU. We can use row_spec to highlight things.

What row_spec is doing here is we're specifying which row – 2 – and making all the text on that row bold. We're making the color of the text white, because we're going to set the background to a color – in this case, the hex color for LSU purple.

```
changeTotalOffense %>%
  kable() %>%
  kable_styling(bootstrap_options = c("striped", "condensed")) %>%
  row_spec(2, bold = T, color = "white", background = "#461D7C")
```

Name

2019

2018

Change

Central Michigan

433.6

254.7

0.7023950

LSU

564.1

402.1

0.4028849

UTSA

344.9

247.1

0.3957912

San Jose State

427.4

323.7

0.3203584

Navy

455.8

349.3

0.3048955

Louisville

447.3

352.6

0.2685763

SMU

489.8

387.2

0.2649793

BYU

443.8

364.9

0.2162236

New Mexico

400.3

330.0

0.2130303

Charlotte

411.8

343.1

0.2002332

There's also something called column_spec where we can change the styling on individual columns. What if we wanted to make all the team names bold?

```
changeTotalOffense %>%
  kable() %>%
  kable_styling(bootstrap_options = c("striped", "condensed")) %>%
  row_spec(2, bold = T, color = "white", background = "#461D7C") %>%
  column_spec(1, bold=T)
```

Name

2019

2018

Change

Central Michigan

433.6

254.7

0.7023950

LSU

564.1

402.1

0.4028849

UTSA

344.9

247.1

0.3957912

San Jose State

427.4

323.7

0.3203584

Navy

455.8

349.3

0.3048955

Louisville

447.3

352.6

0.2685763

SMU

489.8

387.2

0.2649793

BYU

443.8

364.9

0.2162236

New Mexico

400.3

330.0

0.2130303

Charlotte

411.8

343.1

0.2002332

Honestly, this is really good right here. You'd see this published … except for one thing: the percentages.

We could go back up to the top and multiply by 100, but we'd still be missing the percent sign. Well, there's another library for making interesting tables that, in my opinion, has some flaws but does some interesting things too called `formattable`.

Go to the console and install `formattable` with `install.packages("formattable")`.

```
library(formattable)
```

Then, we're going to use mutate here to use formattables `percent()` function to fix Change. Because it uses some HTML wizardry under the hood, we have to set kable to not `escape` the HTML.

```
changeTotalOffense %>%
  mutate(Change = percent(Change)) %>%
  kable(escape=F) %>%
  kable_styling(bootstrap_options = c("striped", "condensed")) %>%
  row_spec(2, bold = T, color = "white", background = "#461D7C") %>%
  column_spec(1, bold=T)
```

Name

2019

2018

Change

Central Michigan

433.6

254.7

70.24%

LSU

564.1

402.1

40.29%

UTSA

344.9

247.1

39.58%

San Jose State

427.4

323.7

32.04%

Navy

455.8

349.3

30.49%

Louisville

447.3

352.6

26.86%

SMU

489.8

387.2

26.50%

BYU

443.8

364.9

21.62%

New Mexico

400.3

330.0

21.30%

Charlotte

411.8

343.1

20.02%

Another way to highlight things: color ramps. We can change the color of the box covering the percentage using another mutate like this:

```
changeTotalOffense %>%
  mutate(Change = percent(Change)) %>%
  mutate(Change = cell_spec(
    Change, color = "white", bold = T,
    background = spec_color(1:10, end = 0.75, option = "B", direction = -1)
  )) %>%
  kable(escape=F) %>%
  kable_styling(bootstrap_options = c("striped", "condensed")) %>%
  row_spec(2, bold = T, color = "white", background = "#461D7C") %>%
  column_spec(1, bold=T)
```

Name

2019

2018

Change

Central Michigan

433.6

254.7

70.24%

LSU

564.1

402.1

40.29%

UTSA

344.9

247.1

39.58%

San Jose State

427.4

323.7

32.04%

Navy

455.8

349.3

30.49%

Louisville

447.3

352.6

26.86%

SMU

489.8

387.2

26.50%

BYU

443.8

364.9

21.62%

New Mexico

400.3

330.0

21.30%

Charlotte

411.8

343.1

20.02%

If this is intersting to you, there's more you can do.

22.1 Exporting tables

One of the major shortcomings with `formattable` is a very limited ability to export tables. Fortunately, `kable` appears to have solved that problem.

But, for this to be a finished product, we need to add a headline, credit line and source line.

Good news bad news: The bad news is that `kable` doesn't have a good way to add a headline. The good news is we can do that in illustrator pretty easily.

The other good news bad news: `kable` isn't going to export this as an actual vector file. It's going to make it an image and embed that into a pdf. That has consequences if we want to edit it.

To export it, we merely add a save_kable to the end and give it a file path.

```
changeTotalOffense %>%
  mutate(Change = percent(Change)) %>%
  mutate(Change = cell_spec(
    Change, color = "white", bold = T,
    background = spec_color(1:10, end = 0.75, option = "B", direction = -1)
  )) %>%
  kable(escape=F) %>%
  kable_styling(bootstrap_options = c("striped", "condensed")) %>%
  row_spec(2, bold = T, color = "white", background = "#461D7C") %>%
  column_spec(1, bold=T) %>%
  save_kable("lsu.pdf")
```

To fix this, we need to pull it into Illustrator. In there, we're going to expand the artboard, add a headline and chatter, then add source and credit lines at the bottom, and finish with touching up the artboards.

Here's a video walkthrough:

Chapter 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.

First, I'm going to show you an example of them not working to illustrate the point.

I'm going to load up my libraries: tidyverse per usual, rvest to get some data (we'll discuss rvest in greater detail in an upcoming chapter) and ggrepel because I end up using it every time I do a scatterplot.

```
library(tidyverse)
library(rvest)
library(ggrepel)
```

So for this example, I want to look at Nebraska's offense in the 2019 season. It … hasn't gone well. And typical of Nebraska teams for the last decade, they're turning the ball over a lot. So given the number of turnovers, how does Nebraska compare to other teams in the FBS?

I'm going to create a scatterplot yards per game on the X axis and points per game on the Y. They're pretty highly correlated with each other. And then

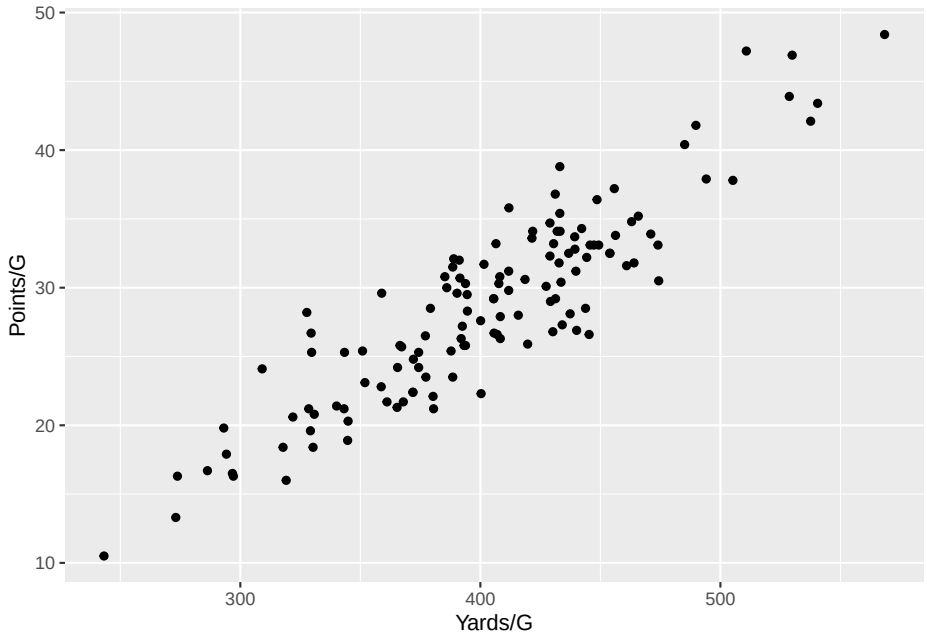
I'm going to make the dot the size of the turnovers – the bubble in my bubble charts.

Using Rvest, I'm going to grab total offense rankings, scoring offense rankings and turnover rankings and then merge them together with just the fields I need. This will all get explained more thoroughly coming up, but that's what this block of code does. When it's done, I'll have a dataframe called `offense` which I'll use to build my bubble chart.

```
yardsurl <- "http://cfbstats.com/2019/leader/national/team/offense/split01/category10/s

yards19 <- yardsurl %>%
  read_html() %>%
  html_nodes(xpath = '//*[@id="content"]/div[2]/table') %>%
  html_table()

yards19 <- yards19[[1]] %>% select(Name, `Yards/G`)

pointsurl <- "http://cfbstats.com/2019/leader/national/team/offense/split01/category09/

points19 <- pointsurl %>%
  read_html() %>%
  html_nodes(xpath = '//*[@id="content"]/div[2]/table') %>%
  html_table()

points19 <- points19[[1]] %>% select(Name, `Points/G`)

turnoversurl <- "http://cfbstats.com/2019/leader/national/team/offense/split01/category

turnovers19 <- turnoversurl %>%
  read_html() %>%
  html_nodes(xpath = '//*[@id="content"]/div[2]/table') %>%
  html_table()

turnovers19 <- turnovers19[[1]] %>% select(Name, `Total Lost`)


offense <- yards19 %>%
  left_join(points19, by=c("Name")) %>%
  left_join(turnovers19, by=c("Name"))
```

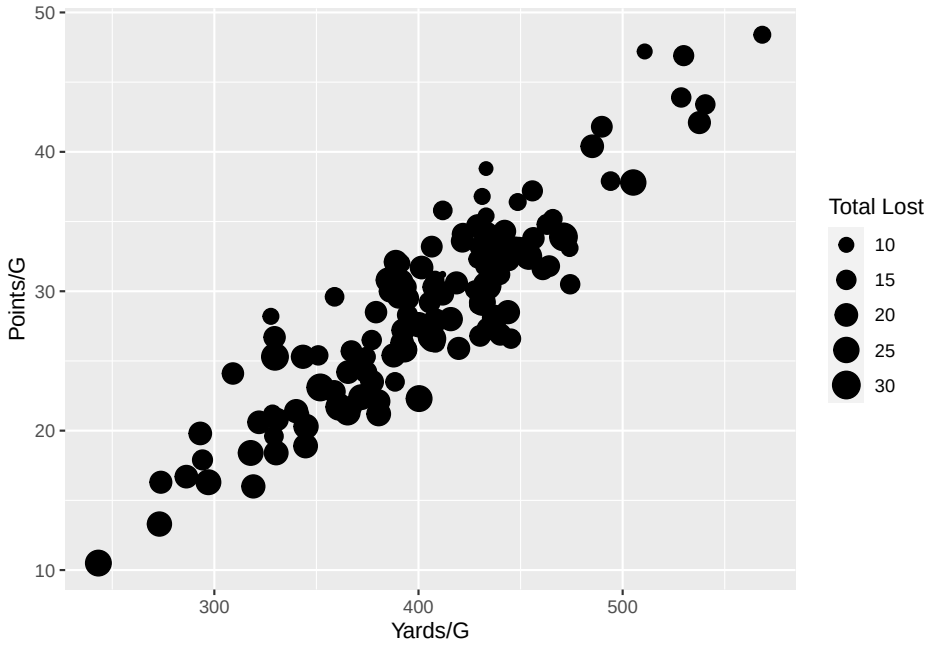
A bubble chart is just a scatterplot with one additional element in the aesthetic – a size. Here's the scatterplot version.

```
ggplot() + geom_point(data=offense, aes(x=`Yards/G`, y=`Points/G`))
```



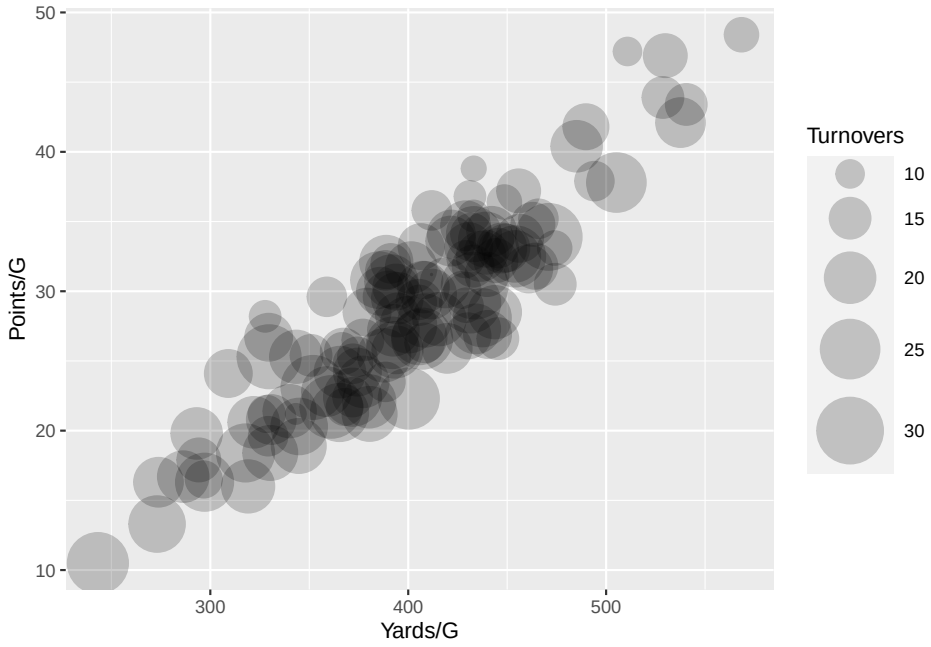
As expected, yards per game pretty tightly predicts points per game, but you could have guessed that without a chart. So let's add the size element.

```
ggplot() + geom_point(data=offense, aes(x=`Yards/G`, y=`Points/G`, size=`Total Lost`))
```



Eh. What does this chart tell you? Trick question, there's not much new here. The dots are too big. Also, we can't see when they overlap. We can fix that 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=offense, aes(x=`Yards/G`, y=`Points/G`, size=`Total Lost`),
```



Before we do any more work, let's return to the earlier question: What story does this tell? Can you discern a story from the bubbles? Are teams with lots of turnovers doing poorly and teams with few turnovers doing well? The problem is, you can't really tell. So this is a dead end for a bubble chart. If you get a big mess, it's a dead giveaway that you probably don't have a bubble chart.

So let's look at something else. Let's look at something that isn't directly correlated – we'll look at offensive points per game vs defensive points per game.

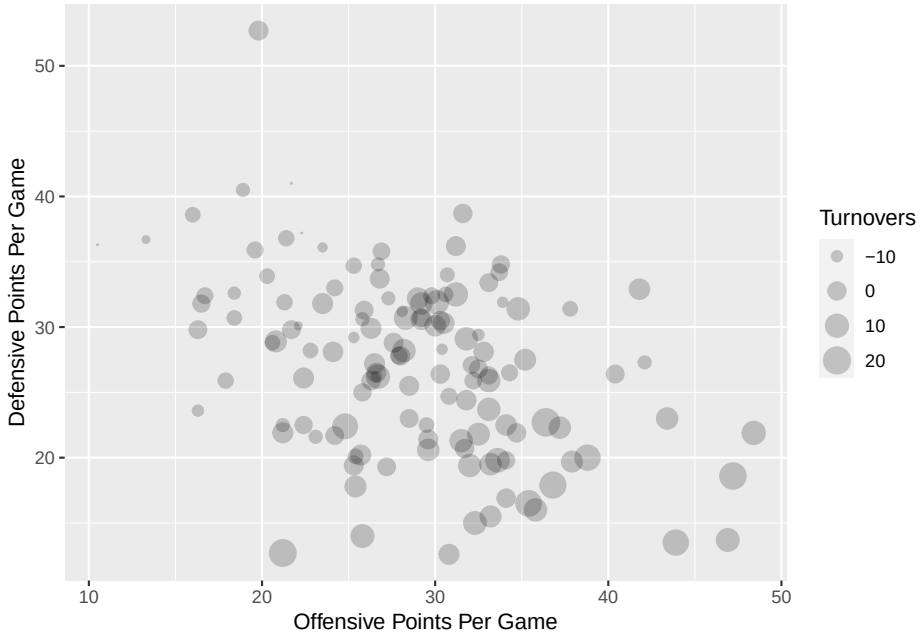
I'm going to edit the same rvest code to grab those points per game stats and merge it all together. When it's done, I'll have a dataframe called `football` and we can look at where good teams fall on the chart with turnover margin as a scaled dot.

```
ourl <- "http://cfbstats.com/2019/leader/national/team/offense/split01/category09/sort01.html"

o19 <- ourl %>%
  read_html() %>%
  html_nodes(xpath = '//*[@id="content"]/div[2]/table') %>%
  html_table()

o19 <- o19[[1]] %>% select(Name, `Points/G`)

durl <- "http://cfbstats.com/2019/leader/national/team/defense/split01/category09/sort01.html"

d19 <- durl %>%
```

```
  read_html() %>%
  html_nodes(xpath = '//*[@id="content"]/div[2]/table') %>%
  html_table()

d19 <- d19[[1]] %>% select(Name, `Points/G`)

turnoversurl <- "http://cfbstats.com/2019/leader/national/team/offense/split01/category

turnovers19 <- turnoversurl %>%
  read_html() %>%
  html_nodes(xpath = '//*[@id="content"]/div[2]/table') %>%
  html_table()

turnovers19 <- turnovers19[[1]] %>% select(Name, Margin)

football <- o19 %>%
  left_join(d19, by=c("Name")) %>%
  left_join(turnovers19, by=c("Name")) %>%
  rename(`Offensive Points Per Game` = `Points/G.x`, `Defensive Points Per Game`=`Point
```

Now we can do the bubble chart.

```
ggplot() + geom_point(data=football, aes(x=`Offensive Points Per Game`, y=`Defensive Po
```



Better! Teams are spread out a little more. Bottom right quadrant – the good defense, good offense quadrant – have some large dots. The upper left quadrant

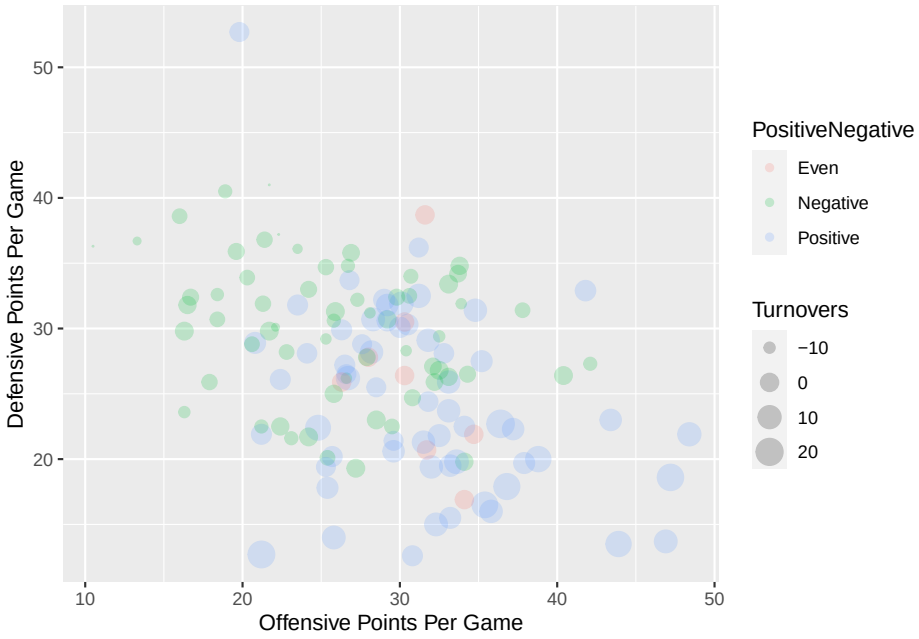
– bad defense, bad offense – have some very small dots, meaning they have a really terrible turnover margin.

But what would make this chart better – and what you saw in the Rosling video – is color. What if we colored the dots by if they were above or below zero? Meaning, do they have a positive or negative turnover margin? We can do that with a quick mutate and a case_when statement.

```
football <- football %>% mutate(PositiveNegative = case_when(
  Margin > 0 ~ "Positive",
  Margin < 0 ~ "Negative",
  Margin == 0 ~ "Even"
))
```

Now we can add `color=PositiveNegative` to the aesthetic and our dots will be colored by if they are positive, negative or zero.

```
ggplot() + geom_point(data=football, aes(x=`Offensive Points Per Game`, y=`Defensive Points Per G
```

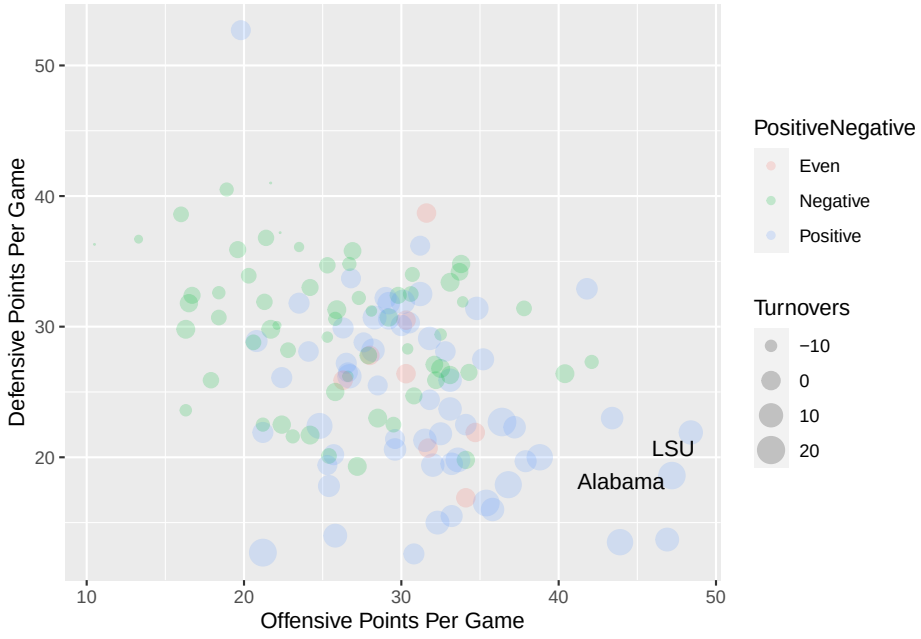


Now we're getting somewhere. What's the story that this chart tells? Blue dots – positive turnover margins – are all drifting toward that good offense, good defense quadrant. Green dots – negative turnover margins – are drifting toward that bad defense, bad offense quadrant.

Let's add some annotations. Let's look at the top two turnover margin teams and where they come out.

```
topteams <- football %>% filter(Name == "LSU" | Name == "Alabama")
```

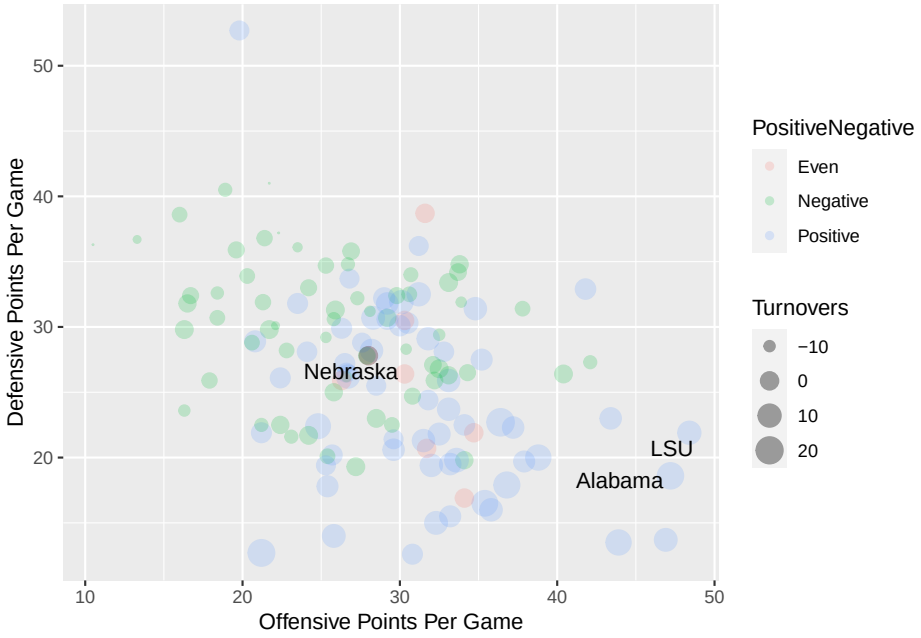
```
ggplot() +
  geom_point(data=football, aes(x=`Offensive Points Per Game`, y=`Defensive Points Per
  geom_text_repel(data=topteams, aes(x=`Offensive Points Per Game`, y=`Defensive Points
  scale_size(range = c(0, 6), name="Turnovers")
```



No surprise there. What about Nebraska?

```
nu <- football %>% filter(Name=="Nebraska")
```

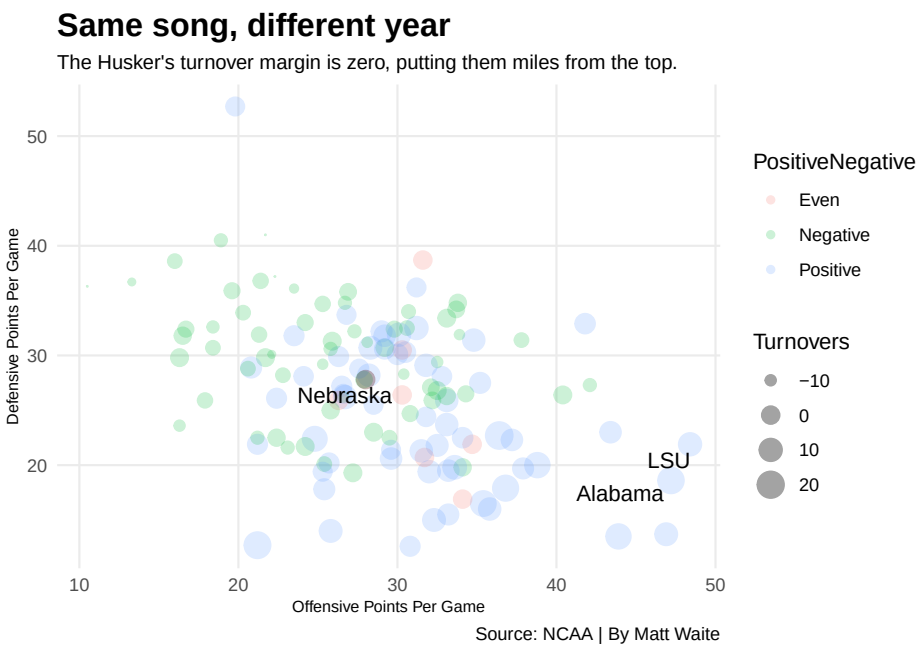
```
ggplot() +
  geom_point(data=football, aes(x=`Offensive Points Per Game`, y=`Defensive Points Per
  geom_text_repel(data=topteams, aes(x=`Offensive Points Per Game`, y=`Defensive Points
  geom_point(data=nu, aes(x=`Offensive Points Per Game`, y=`Defensive Points Per Game`
  geom_text_repel(data=nu, aes(x=`Offensive Points Per Game`, y=`Defensive Points Per
  scale_size(range = c(0, 6), name="Turnovers")
```



Sadly, no surprise there either.

The last things we need to do? Add some labels, apply our finishing touches.

```
ggplot() +
  geom_point(data=football, aes(x=`Offensive Points Per Game`, y=`Defensive Points Per Game`, siz
  geom_text_repel(data=topteams, aes(x=`Offensive Points Per Game`, y=`Defensive Points Per Game`
  geom_point(data=nu, aes(x=`Offensive Points Per Game`, y=`Defensive Points Per Game`, size=Marg
  geom_text_repel(data=nu, aes(x=`Offensive Points Per Game`, y=`Defensive Points Per Game`, labe
  scale_size(range = c(0, 6), name="Turnovers") +
  labs(title="Same song, different year", subtitle="The Husker's turnover margin is zero, putting
  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()
    )
```


Same song, different year
The Husker's turnover margin is zero, putting them miles from the top.
Defensive Points Per Game
50
40
30
20
10
20
30
40
50
Offensive Points Per Game
Nebraska
LSU
Alabama
PositiveNegative
Even
Negative
Positive
Turnovers
−10
0
10
20
Source: NCAA | By Matt Waite

Chapter 24

Circular bar plots

At the 27:36 mark in the Half Court Podcast, Omaha World Herald Writer Chris Heady said "November basketball doesn't matter, but it shows you where you are."

It's a tempting phrase to believe, especially a day after Nebraska lost the first game of the Fred Hoiberg era at home to a baseball school, UC Riverside. And it wasn't close. The Huskers, because of a total roster turnover, were a complete mystery before the game. And what happened during it wasn't pretty, so there was a little soul searching going on in Lincoln.

But does November basketball really not matter?

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 interstingness 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.

First we need some libraries.

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

Let's import every basketball game from this year.

```
logs <- read_csv("data/logs20.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Date = col_date(format = ""),
##   HomeAway = col_character(),
```

```
##   Opponent = col_character(),
##   W_L = col_character(),
##   Blank = col_logical(),
##   Team = col_character(),
##   Conference = col_character(),
##   season = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

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'm also going to filter out tournament basketball.

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

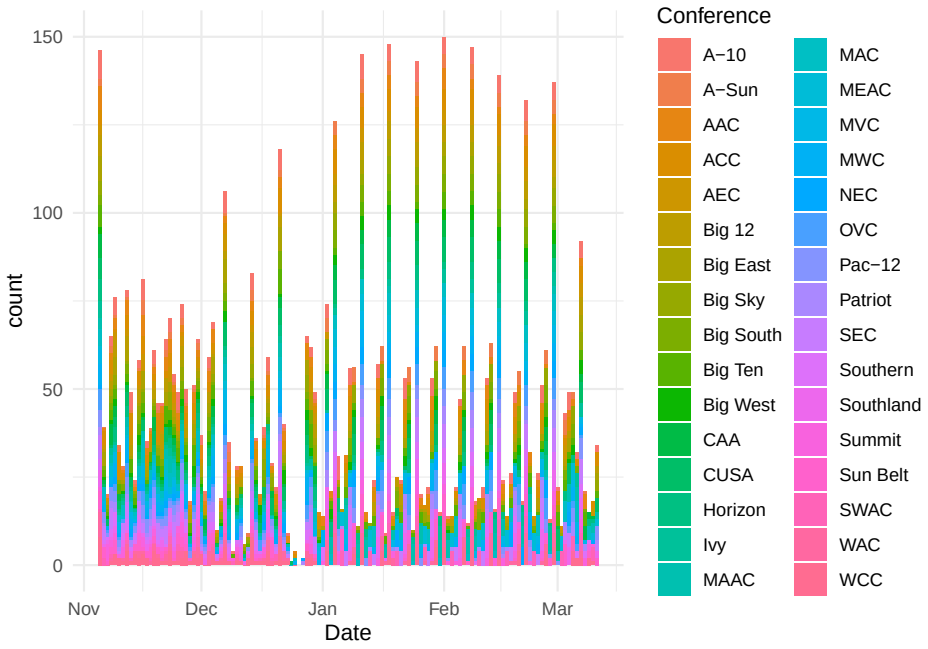
Now we can group by date and conference and sum up the wins. How many wins by day does each conference get?

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

```
## `summarise()` regrouping output by 'Date' (override with `.groups` argument)
```

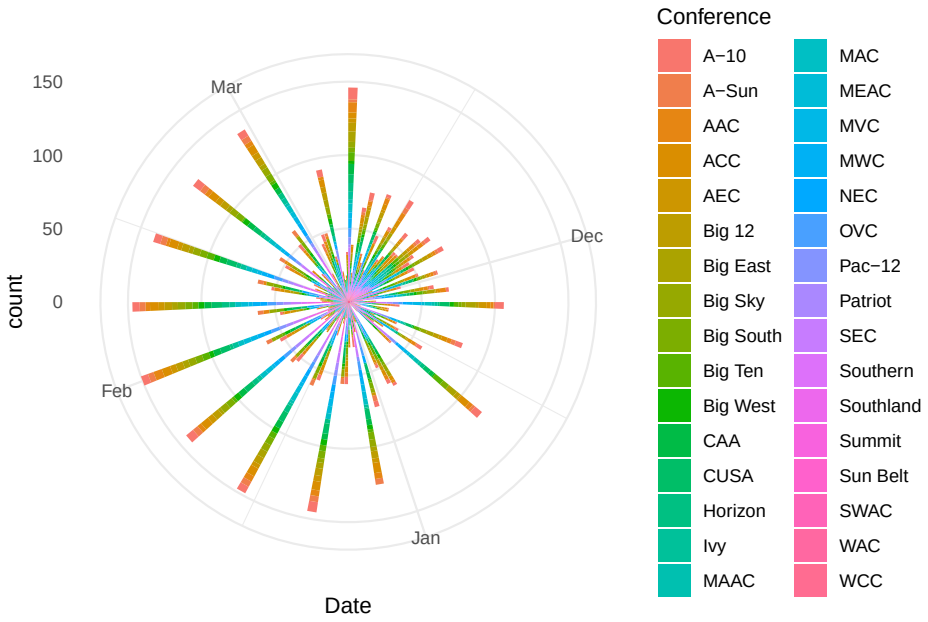
Earlier, we did stacked bar charts. We have what we need to do that now.

```
ggplot() + geom_bar(data=dates, aes(x=Date, weight=wins, fill=Conference)) + theme_mini
```



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() + co
```



Based on that, the day is probably too thin a slice, and there's way too many

conferences in college basketball. Let's group this by months and filter out all but the power five conferences.

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

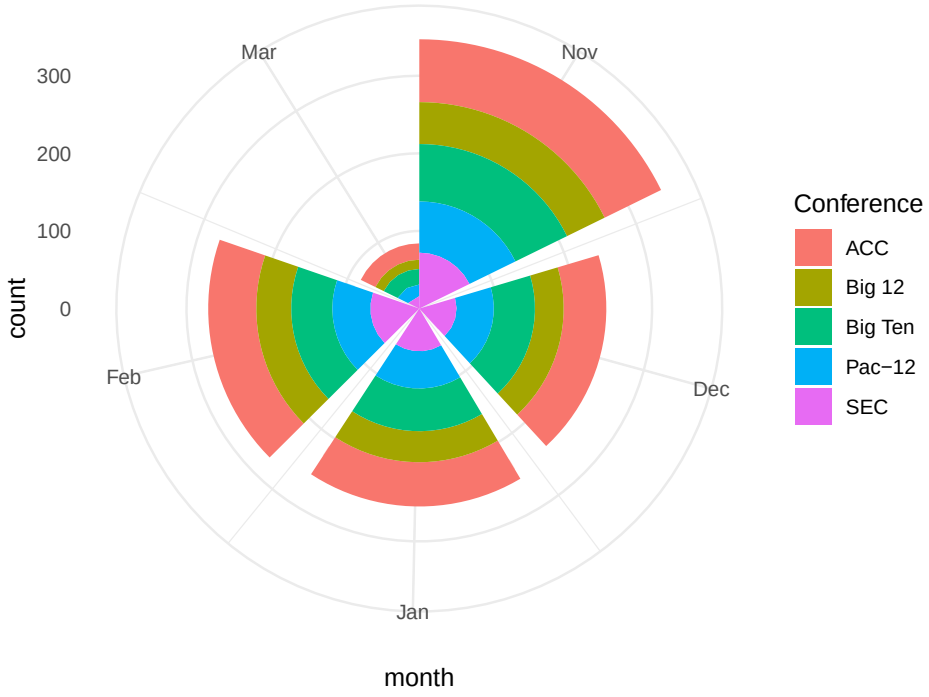
To get months, we're going to use a function in the library `lubridate` called `floor_date`, which combine with mutate will give us a field of just months.

```
wins <- winlosslogs %>% mutate(month = floor_date(Date, unit="months")) %>% group_by(mo
```

```
## `summarise()` regrouping output by 'month' (override with `.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_mini
```

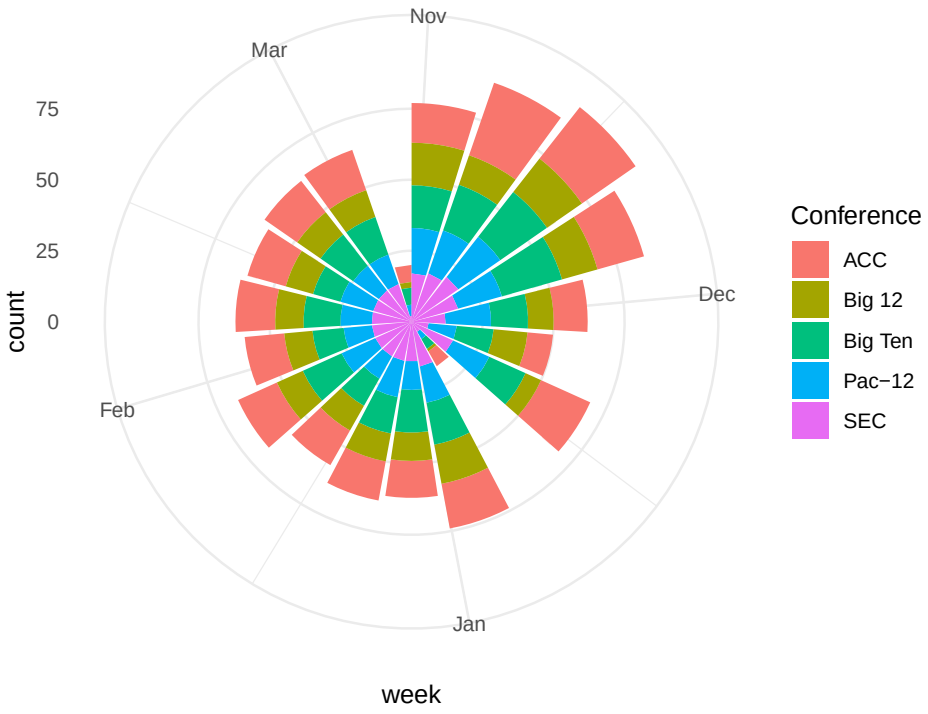


Yikes. That looks a lot like a broken pie chart. So months are too thick of a slice. Let's use weeks in our floor date to see what that gives us.

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

```
## `summarise()` regrouping output by 'week' (override with `.groups` argument)
```

```
ggplot() + geom_bar(data=wins, aes(x=week, weight=wins, fill=Conference)) + theme_minim
```



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.

Chapter 25

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. This is scoring offense for 2019. 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/2019/leader/national/team/offense/split01/category0
```

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.

```
scoringoffense <- scoringoffense[[1]]
```

And what do we have?

```
head(scoringoffense)
```

```
##           Name  G TD FG 1XP 2XP Safety Points Points/G
## 1 1        LSU 15 95 21  89   1      1    726     48.4
## 2 2    Alabama 13 83 12  80   0      0    614     47.2
## 3 3 Ohio State 14 88 13  87   0      1    656     46.9
## 4 4    Clemson 15 88 14  85   2      0    659     43.9
## 5 5        UCF 13 74 15  71   1      1    564     43.4
## 6 6   Oklahoma 14 76 19  75   1      0    590     42.1
```

We have data, ready for analysis.

25.1 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 2018.

```
scoringoffenseurl18 <- "http://www.cfbstats.com/2018/leader/national/team/offense/split

scoringoffense18 <- scoringoffenseurl18 %>%
  read_html() %>%
  html_nodes(xpath = '//*[@id="content"]/div[2]/table') %>%
  html_table()

scoringoffense18 <- scoringoffense18[[1]]
```

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

```
scoringoffense19 <- scoringoffense %>% mutate(YEAR = 2019)
```

```
## Error in env_bind_lazy(private$bindings, !!!set_names(promises, names_bindings)): attempt to u
```

Uh oh. Error. What does it say? Column 1 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.

```
scoringoffense19 <- scoringoffense %>% rename(Rank = 1) %>% mutate(YEAR = 2019)
scoringoffense18 <- scoringoffense18 %>% rename(Rank = 1) %>% mutate(YEAR = 2018)
```

And now, to combine the two tables together length-wise – we need to make long data – we'll use a base R function called `rbind`. The good thing is rbind is simple. The bad part is it can only do two tables at a time, so if you have more than that, you'll need to do it in steps.

```
combined <- rbind(scoringoffense19, scoringoffense18)
```

Note in the environment tab we now have a data frame called combined that has 260 observations – which just so happens to be what 130 from 2019 and 130 from 2018 add up to.

```
head(combined)
```

```
##   Rank       Name  G TD FG 1XP 2XP Safety Points Points/G YEAR
## 1    1        LSU 15 95 21  89   1      1    726     48.4 2019
## 2    2    Alabama 13 83 12  80   0      0    614     47.2 2019
## 3    3 Ohio State 14 88 13  87   0      1    656     46.9 2019
## 4    4    Clemson 15 88 14  85   2      0    659     43.9 2019
## 5    5        UCF 13 74 15  71   1      1    564     43.4 2019
## 6    6   Oklahoma 14 76 19  75   1      0    590     42.1 2019
```

25.2 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/2019-school-stats.html"
```

Now, similar to our example above, we'll read the html, use XPath to find the table, and then read that table with a directive passed to it setting the header to FALSE. That tells rvest that there isn't a header row. Just import it as data.

```
stats <- url %>%
  read_html() %>%
  html_nodes(xpath = '//*[@id="basic_school_stats"]') %>%
  html_table(header=FALSE)
```

What we get back is a list of one element (similar to above). So let's pop it out into a data frame.

```
stats <- stats[[1]]
```

And we'll take a look at what we have.

```
head(stats)
```

```
##   X1                     X2      X3      X4      X5      X6      X7      X8 X9
## 1                           Overall Overall Overall Overall Overall Overall NA
## 2 Rk                School       G       W       L    W-L%     SRS     SOS NA
## 3  1 Abilene Christian NCAA      34      27       7    .794   -1.91   -7.34 NA
## 4  2              Air Force      32      14      18    .438   -4.28    0.24 NA
## 5  3                  Akron      33      17      16    .515    4.86    1.09 NA
## 6  4            Alabama A&M      32       5      27    .156  -19.23   -8.38 NA
##     X10   X11 X12  X13  X14 X15  X16  X17 X18    X19    X20 X21    X22    X23
## 1 Conf. Conf.  NA Home Home  NA Away Away  NA Points Points  NA Totals Totals
## 2     W     L  NA    W    L  NA    W    L  NA    Tm.   Opp.  NA     MP     FG
## 3    14     4  NA   13    2  NA   10    4  NA   2502   2161  NA   1370    897
## 4     8    10  NA    9    6  NA    3    9  NA   2179   2294  NA   1300    802
## 5     8    10  NA   14    3  NA    1   10  NA   2271   2107  NA   1325    797
## 6     4    14  NA    4    7  NA    0   18  NA   1938   2285  NA   1295    736
##      X24    X25    X26    X27    X28    X29    X30    X31    X32    X33    X34
## 1 Totals Totals Totals Totals Totals Totals Totals Totals Totals Totals Totals
## 2    FGA    FG%     3P    3PA    3P%     FT    FTA    FT%    ORB    TRB    AST
## 3   1911   .469    251    660   .380    457    642   .712    325   1110    525
## 4   1776   .452    234    711   .329    341    503   .678    253   1077    434
## 5   1948   .409    297    929   .320    380    539   .705    312   1204    399
## 6   1809   .407    182    578   .315    284    453   .627    314   1032    385
##      X35    X36    X37    X38
## 1 Totals Totals Totals Totals
## 2    STL    BLK    TOV     PF
```

```
## 3    297     93    407    635
## 4    154     57    423    543
## 5    185    106    388    569
## 6    234     50    487    587
```

So, that's not ideal. We have headers and data mixed together, and our columns are named X1 to X34. 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, WinPct=
```

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 33 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. Here's what it looks like:

```
stats <- stats %>% mutate_at(vars(-School), as.numeric)
```

And just like that, we have a method for getting up to the minute season stats for every team in Division I.

```
head(stats)
```

```
##   Rank                School Games OverallWins OverallLosses WinPct OverallSRS
## 1    1 Abilene Christian NCAA    34          27             7  0.794      -1.91
## 2    2              Air Force    32          14            18  0.438      -4.28
## 3    3                  Akron    33          17            16  0.515       4.86
## 4    4             Alabama A&M    32           5            27  0.156     -19.23
## 5    5     Alabama-Birmingham    35          20            15  0.571       0.36
## 6    6          Alabama State    31          12            19  0.387     -15.60
##   OverallSOS ConferenceWins ConferenceLosses HomeWins HomeLosses AwayWins
## 1      -7.34             NA               14        4          NA       13
## 2       0.24             NA                8       10          NA        9
## 3       1.09             NA                8       10          NA       14
## 4      -8.38             NA                4       14          NA        4
## 5      -1.52             NA               10        8          NA       11
## 6      -7.84             NA                9        9          NA        8
##   AwayLosses ForPoints OppPoints Blank Minutes FieldGoalsMade
## 1          2        NA        10     4      NA           2502
## 2          6        NA         3     9      NA           2179
```

```
## 3          3        NA         1    10      NA           2271
## 4          7        NA         0    18      NA           1938
## 5          5        NA         6     6      NA           2470
## 6          3        NA         3    13      NA           2086
##   FieldGoalsAttempted FieldGoalPCT ThreePointMade ThreePointAttempts
## 1                2161           NA           1370                897
## 2                2294           NA           1300                802
## 3                2107           NA           1325                797
## 4                2285           NA           1295                736
## 5                2370           NA           1410                906
## 6                2235           NA           1250                712
##   ThreePointPct FreeThrowsMade FreeThrowsAttempted FreeThrowPCT
## 1          1911          0.469                 251          660
## 2          1776          0.452                 234          711
## 3          1948          0.409                 297          929
## 4          1809          0.407                 182          578
## 5          2003          0.452                 234          694
## 6          1764          0.404                 216          673
##   OffensiveRebounds TotalRebounds Assists Steals Blocks Turnovers PersonalFouls
## 1             0.380           457     642  0.712    325      1110           525
## 2             0.329           341     503  0.678    253      1077           434
## 3             0.320           380     539  0.705    312      1204           399
## 4             0.315           284     453  0.627    314      1032           385
## 5             0.337           424     630  0.673    367      1279           401
## 6             0.321           446     684  0.652    365      1094           313
##   X35 X36 X37 X38
## 1 297  93 407 635
## 2 154  57 423 543
## 3 185 106 388 569
## 4 234  50 487 587
## 5 218  82 399 578
## 6 203 102 451 565
```

Chapter 26

Advanced rvest

With the chapter, we learned how to grab one table from one page. But what if you needed more than that? What if you needed hundreds of tables from hundreds of pages? What if you needed to combine one table on one page into a bigger table, but hundreds of times. There's a way to do this, it just takes patience, a lot of logic, a lot of debugging and, for me, a fair bit of swearing.

So what we are after are game by game stats for each college basketball team in America.

We can see from this page that each team is linked. If we follow each link, we get a ton of tables. But they aren't what we need. There's a link to gamelogs underneath the team names.

So we can see from this that we've got some problems.

1. The team name isn't in the table. Nor is the conference.
2. There's a date we'll have to deal with.
3. Non-standard headers and a truly huge number of fields.
4. And how do we get each one of those urls without having to copy them all into some horrible list?

So let's start with that last question first and grab libraries we need.

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

First things first, we need to grab the url to each team from that first link.

```
url <- "https://www.sports-reference.com/cbb/seasons/2019-school-stats.html"
```

But notice first, we don't want to grab the table. The table doesn't help us. We need to grab the only *link* in the table. So we can do that by using the table

xpath node, then grabbing the anchor tags in the table, then get only the link out of them (instead of the linked text).

```
schools <- url %>%
  read_html() %>%
  html_nodes(xpath = '//*[@id="basic_school_stats"]') %>%
  html_nodes("a") %>%
  html_attr('href')
```

Notice we now have a list called schools with ... 353 elements. That's the number of teams in college basketball, so we're off to a good start. Here's what the fourth element is.

```
schools[4]
```

```
## [1] "/cbb/schools/alabama-am/2019.html"
```

So note, that's the relative path to Alabama A&M's team page. By relative path, I mean it doesn't have the root domain. So we need to add that to each request or we'll get no where.

So that's a problem to note.

Before we solve that, let's just make sure we can get one page and get what we need.

We'll scrape Abilene Christian.

To merge all this into one big table, we need to grab the team name and their conference and merge it into the table. But those values come from somewhere else. The scraping works just about the same, but instead of html_table you use html_text.

So the first part of this is reading the html of the page so we don't do that for each element – we just do it once so as to not overwhelm their servers.

The second part is we're grabbing the team name based on it's location in the page.

Third: The conference.

Fourth is the table itself, noting to ignore the headers. The last bit fixes the headers, removes the garbage header data from the table, converts the data to numbers, fixes the date and mutates a team and conference value. It looks like a lot, and it took a bit of twiddling to get it done, but it's no different from what you did for your last homework.

```
page <- read_html("https://www.sports-reference.com/cbb/schools/abilene-christian/2019

team <- page %>%
  html_nodes(xpath = '//*[@id="meta"]/div[2]/h1/span[2]') %>%
  html_text()
```

```
conference <- page %>%
    html_nodes(xpath = '//*[@id="meta"]/div[2]/p[1]/a') %>%
    html_text()

table <- page %>%
  html_nodes(xpath = '//*[@id="sgl-basic"]') %>%
  html_table(header=FALSE)

table <- table[[1]] %>% rename(Game=X1, Date=X2, HomeAway=X3, Opponent=X4, W_L=X5, TeamScore=X6,
```

Now what we're left with is how do we do this for ALL the teams. We need to send 353 requests to their servers to get each page. And each url is not the one we have – we need to alter it.

First we have to add the root domain to each request. And, each request needs to go to /2019-gamelogs.html instead of /2019.html. If you look at the urls two the page we have and the page we need, that's all that changes.

What we're going to use is what is known in programming as a loop. We can loop through a list and have it do something to each element in the loop. And once it's done, we can move on to the next thing.

Think of it like a program that will go though a list of your classmates and ask each one of them for their year in school. It will start at one end of the list and move through asking each one "What year in school are you?" and will move on after getting an answer.

Except we want to take a url, add something to it, alter it, then request it and grab a bunch of data from it. Once we're done doing all that, we'll take all that info and cram it into a bigger dataset and then move on to the next one. Here's what that looks like:

```
uri <- "https://www.sports-reference.com"

logs <- tibble()

for (i in schools){
  log_url <- gsub("/2019.html","/2019-gamelogs.html", i)
  school_url <- paste(uri, log_url, sep="")  # creating the url to fetch

  page <- read_html(school_url)

  team <- page %>%
    html_nodes(xpath = '//*[@id="meta"]/div[2]/h1/span[2]') %>%
    html_text()

  conference <- page %>%
```

```
    html_nodes(xpath = '//*[@id="meta"]/div[2]/p[1]/a') %>%
    html_text()

  table <- page %>%
    html_nodes(xpath = '//*[@id="sgl-basic"]') %>%
    html_table(header=FALSE)

table <- table[[1]] %>% rename(Game=X1, Date=X2, HomeAway=X3, Opponent=X4, W_L=X5, Tea

  logs <- rbind(logs, table)  # binding them all together
  Sys.sleep(5)  # Sys.sleep(3) pauses the loop for 3s so as not to overwhelm website's
}
```

The magic here is in `for (i in schools){`. What that says is for each iterator in schools – for each school in schools – do what follows each time. So we take the code we wrote for one school and use it for every school.

This part:

```
  log_url <- gsub("/2019.html","/2019-gamelogs.html", i)
  school_url <- paste(uri, log_url, sep="")  # creating the url to fetch

  page <- read_html(school_url)
```

`log_url` is what changes our school page url to our logs url, and `school_url` is taking that log url and the root domain and merging them together to create the complete url. Then, page just reads that url we created.

What follows that is taken straight from our example of just doing one.

The last bits are using rbind to take our data and mash it into a bigger table, over and over and over again until we have them all in a single table. Then, we tell our scraper to wait a few seconds because we don't want our script to machine gun requests at their server as fast as it can go. That's a guaranteed way to get them to block scrapers, and could knock them off the internet. Aggressive scrapers aren't cool. Don't do it.

Lastly, we write it out to a csv file.

```
write.csv(logs, "logs.csv")
```

So with a little programming knowhow, a little bit of problem solving and the stubbornness not to quit on it, you can get a whole lot of data scattered all over the place with not a lot of code.

26.1 One last bit

Most tables that Sports Reference sites have are in plain vanilla HTML. But some of them – particularly player based stuff – are hidden with a little trick.

The site puts the data in a comment – where a browser will ignore it – and then uses javascript to interpret the commented data. To a human on the page, it looks the same. To a browswer or a scraper, it's invisible. You'll get errors. How do you get around it?

1. Scrape the comments.
2. Turn the comment into text.
3. Then read that text as html.
4. Proceed as normal.

```
h <- read_html('https://www.baseball-reference.com/leagues/MLB/2017-standard-pitching.shtml')

df <- h %>% html_nodes(xpath = '//comment()') %>%    # select comment nodes
    html_text() %>%    # extract comment text
    paste(collapse = '') %>%    # collapse to a single string
    read_html() %>%    # reparse to HTML
    html_node('table') %>%    # select the desired table
    html_table()
```

Chapter 27

Annotations

Some of the best sports data visualizations start with a provocative question. At a college just under three hours from Kansas City, my classes are lousy with Chiefs fans. So the first day of classes in the spring of 2019, I asked them: Are the Chief's Screwed in the Playoffs? The answer ultimately was yes, and how I was able to make that argument before a playoff game had even been played is a good example of how labeling and annotations can make a chart much better.

Going 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 and join that data together using the Team name as the common element.

```
offense <- read_csv("data/nfloffense.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Team = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

```
defense <- read_csv("data/nfldefense.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Team = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

```
total <- offense %>% left_join(defense, by="Team")

head(total)
```

```
## # A tibble: 6 x 52
##   Team      G PointsFor OffYards OffPlays OffYardsPerPlay OffensiveTurnov~
##   <chr> <dbl>     <dbl>    <dbl>    <dbl>           <dbl>            <dbl>
## 1 Kans~    16       565     6810      996             6.8               18
## 2 Los ~    16       527     6738     1060             6.4               19
## 3 New ~    16       504     6067     1010             6                 16
## 4 New ~    16       436     6295     1073             5.9               18
## 5 Indi~    16       433     6179     1070             5.8               24
## 6 Pitt~    16       428     6453     1058             6.1               26
## # ... with 45 more variables: FumblesLost <dbl>, OffFirstDowns <dbl>,
## #   OffPassingComp <dbl>, OffPassingAtt <dbl>, OffPassingYards <dbl>,
## #   OffensivePassingTD <dbl>, OffPassingINT <dbl>,
## #   OffensivePassingYardsPerAtt <dbl>, OffensivePassingFirstDowns <dbl>,
## #   OffRushingAtt <dbl>, OffRushingYards <dbl>, OffRushingTD <dbl>,
## #   RushingYardsPerAtt <dbl>, RushingFirstDowns <dbl>,
## #   OffensivePenalties <dbl>, OffPenaltyYards <dbl>,
## #   OffFirstFromPenalties <dbl>, OffScoringPct <dbl>,
## #   OffensiveTurnoverPct <dbl>, OffensiveExpectedPoints <dbl>,
## #   PointsAllowed <dbl>, YdsAllowed <dbl>, PlaysFaced <dbl>,
## #   DefYardPerPlay <dbl>, Takeaways <dbl>, DefFumblesLost <dbl>,
## #   FirstDownsAllowed <dbl>, PassingCompsAllowed <dbl>, PassingAttFaced <dbl>,
## #   PassingYdsAllowed <dbl>, PassingTDAllowed <dbl>, DefPassingINT <dbl>,
## #   PassingYardsPerPlayAllowed <dbl>, PassingFirstDownsAllowed <dbl>,
## #   RushingAttFaced <dbl>, RushingYdsAllowed <dbl>, RushingTDAllowed <dbl>,
## #   RushingYardsPerAttAllowed <dbl>, RushingFirstDownsAllowed <dbl>,
## #   DefPenalties <dbl>, DefPenaltyYards <dbl>, DefFirstDownByPenalties <dbl>,
## #   OffensiveScoringPctAllowed <dbl>, DefTurnoverPercentage <dbl>,
## #   DefExpectedPoints <dbl>
```

I'm going to set up a point chart that places team on two-axes – yards per play on offense on the x axis, and yards per play on defense.

To build the annotations, I want the league 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.

```
league_averages <- total %>% summarise(AvgOffYardsPer = mean(OffYardsPerPlay), AvgDefYa

league_averages
```

```
## # A tibble: 1 x 2
##   AvgOffYardsPer AvgDefYardsPer
##            <dbl>          <dbl>
## 1           5.59           5.59
```

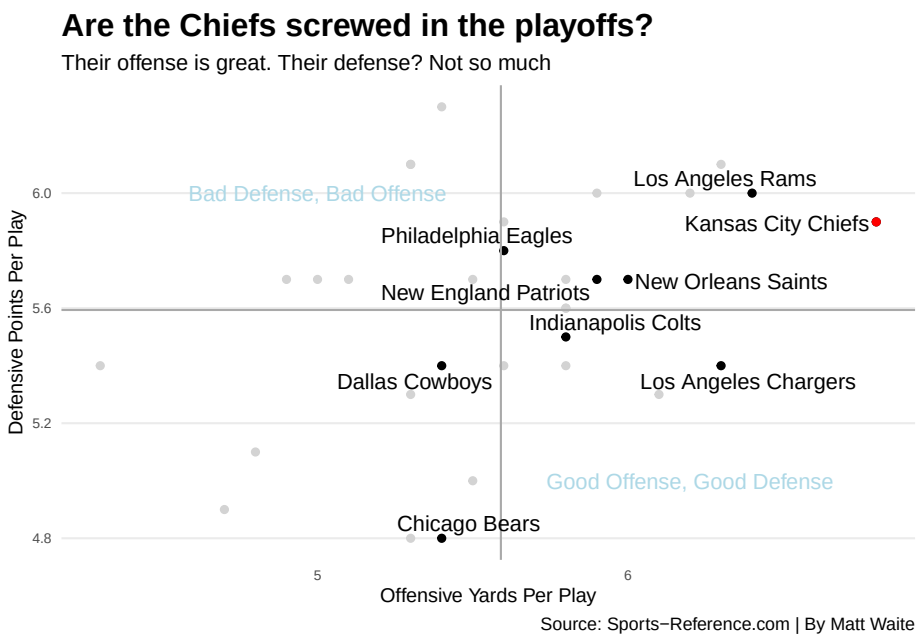
I also want to highlight playoff teams and, of course, the Chiefs, since that was my question. Are they screwed. First, we filter them from our total list.

```
playoff_teams <- c("Kansas City Chiefs", "New England Patriots", "Los Angeles Chargers", "Indiana

playoffs <- total %>% filter(Team %in% playoff_teams)

chiefs <- total %>% filter(Team == "Kansas City Chiefs")
```

Now we create the plot. We have three geom_points, starting with everyone, then playoff teams, then the Chiefs. I alter the colors on each to separate them. Next, I add a geom_hline to add the horizontal line of my defensive average and a geom_vline for my offensive average. Next, I want to add some text annotations, labeling two corners of my chart (the other two, in my opinion, become obvious). Then, I want to label all the playoff teams. I use `geom_text_repel` to do that – it's using the ggrepel library to push the text away from the dots, respective of other labels and other dots. It means you don't have to move them around so you can read them, or so they don't cover up the dots.

The rest is just adding labels and messing with the theme.

```
ggplot() +
  geom_point(data=total, aes(x=OffYardsPerPlay, y=DefYardPerPlay), color="light grey") +
  geom_point(data=playoffs, aes(x=OffYardsPerPlay, y=DefYardPerPlay)) +
  geom_point(data=chiefs, aes(x=OffYardsPerPlay, y=DefYardPerPlay), color="red") +
  geom_hline(yintercept=5.59375, color="dark grey") +
  geom_vline(xintercept=5.590625, color="dark grey") +
  geom_text(aes(x=6.2, y=5, label="Good Offense, Good Defense"), color="light blue") +
  geom_text(aes(x=5, y=6, label="Bad Defense, Bad Offense"), color="light blue") +
  geom_text_repel(data=playoffs, aes(x=OffYardsPerPlay, y=DefYardPerPlay, label=Team)) +
  labs(x="Offensive Yards Per Play", y="Defensive Points Per Play", title="Are the Chiefs screwed
  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()
  )
```


Are the Chiefs screwed in the playoffs?
Their offense is great. Their defense? Not so much
Bad Defense, Bad Offense
Los Angeles Rams
Kansas City Chiefs
Philadelphia Eagles
New Orleans Saints
New England Patriots
Indianapolis Colts
Dallas Cowboys
Los Angeles Chargers
Good Offense, Good Defense
Chicago Bears
Defensive Points Per Play
6.0
5.6
5.2
4.8
5
6
Offensive Yards Per Play
Source: Sports−Reference.com | By Matt Waite

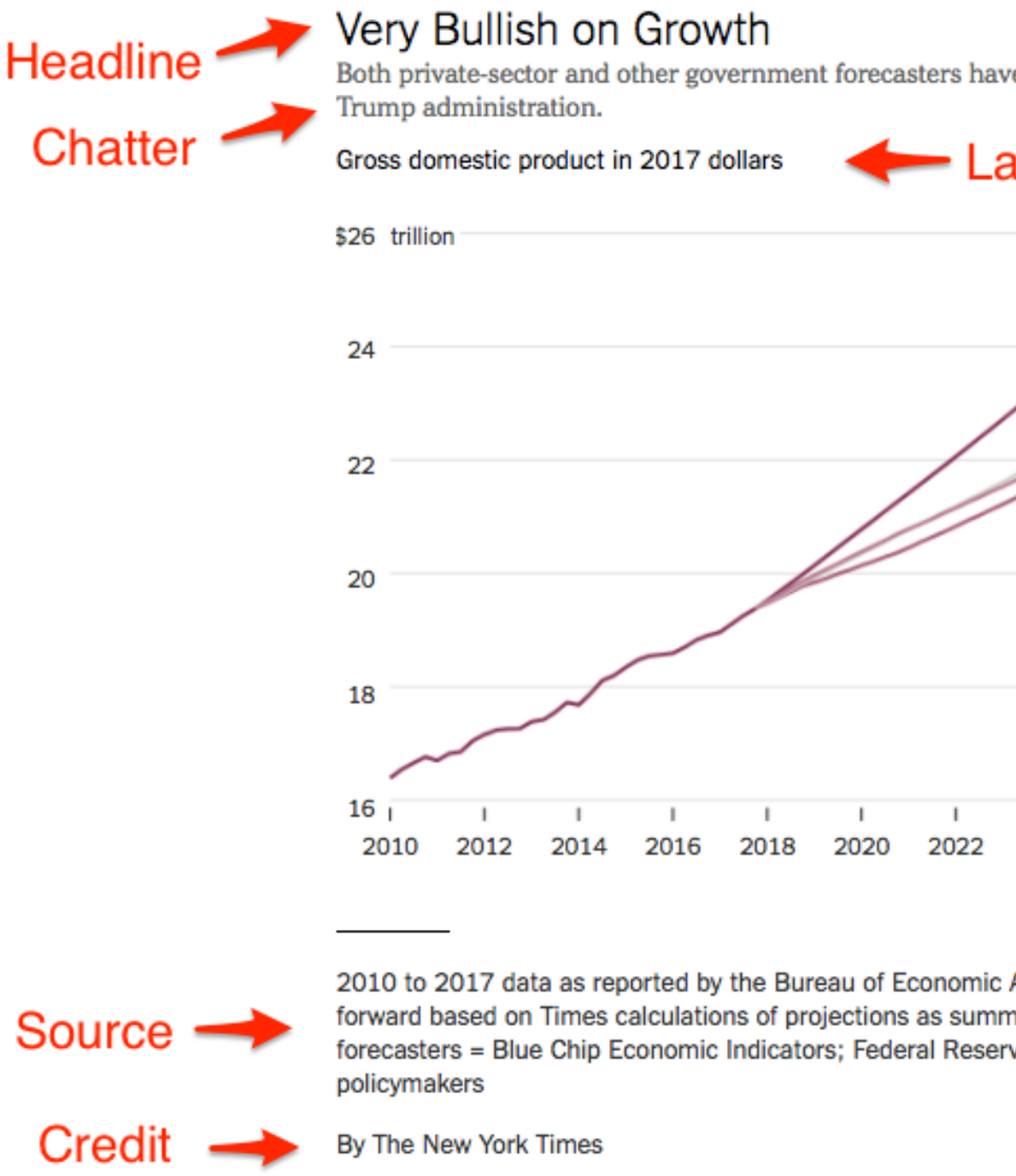
Chapter 28

Finishing touches, part 1

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:


Headline
Very Bullish on Growth
Chatter
Both private-sector and other government forecasters have
Trump administration.
Gross domestic product in 2017 dollars
La
$26 trillion
24
22
20
18
16
2010
2012
2014
2016
2018
2020
2022
Source
2010 to 2017 data as reported by the Bureau of Economic A
forward based on Times calculations of projections as summ
forecasters = Blue Chip Economic Indicators; Federal Reserv
policymakers
Credit
By The New York Times

28.1 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.

28.2 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)
```

```
scoring <- read_csv("data/scoringoffense.csv")
```

```
## Parsed with column specification:
## cols(
##   Name = col_character(),
##   G = col_double(),
##   TD = col_double(),
##   FG = col_double(),
##   `1XP` = col_double(),
##   `2XP` = col_double(),
##   Safety = col_double(),
##   Points = col_double(),
##   `Points/G` = col_double(),
##   Year = col_double()
## )
```

```
total <- read_csv("data/totaloffense.csv")
```

```
## Parsed with column specification:
## cols(
##   Name = col_character(),
##   G = col_double(),
##   `Rush Yards` = col_double(),
##   `Pass Yards` = col_double(),
##   Plays = col_double(),
##   `Total Yards` = col_double(),
##   `Yards/Play` = col_double(),
```

```
##   `Yards/G` = col_double(),
##   Year = col_double()
## )
```

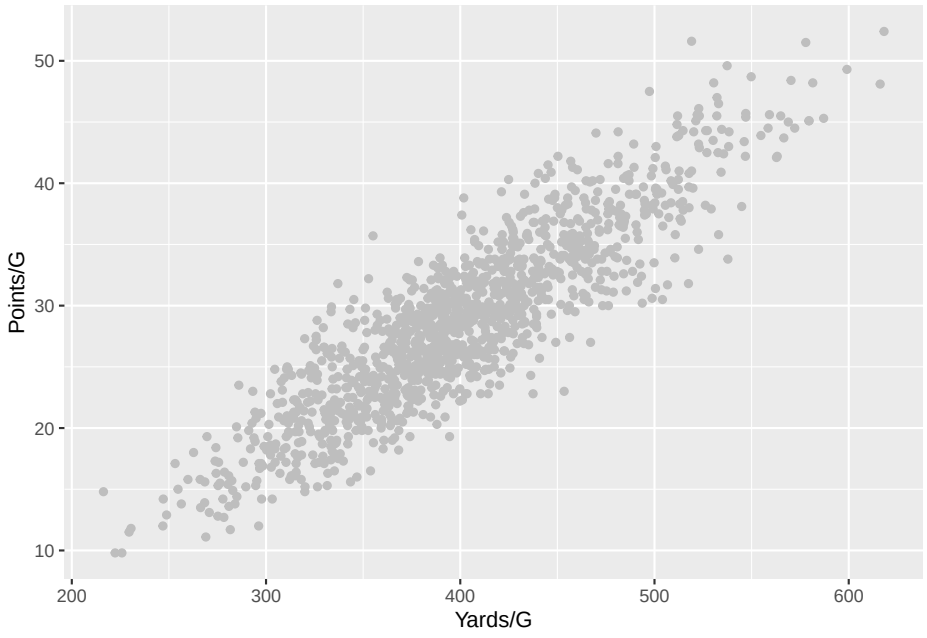
```
offense <- total %>% left_join(scoring, by=c("Name", "Year"))
```

We're going to need this later, so let's grab Nebraska's 2018 stats from this dataframe.

```
nu <- offense %>% filter(Name == "Nebraska") %>% filter(Year == 2018)
```

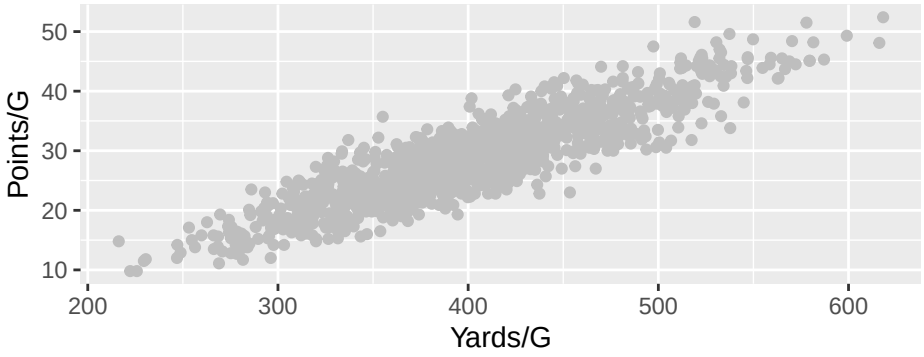
We'll start with the basics.

```
ggplot(offense, aes(x=`Yards/G`, y=`Points/G`)) +
  geom_point(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.

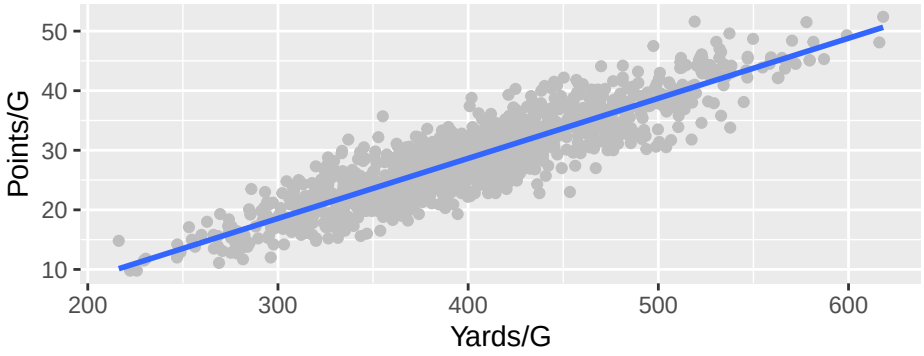
```
ggplot(offense, aes(x=`Yards/G`, y=`Points/G`)) +
  geom_point(color="grey")
```



Now let's add a fit line.

```
ggplot(offense, aes(x=`Yards/G`, y=`Points/G`)) +
  geom_point(color="grey") + geom_smooth(method=lm, se=FALSE)
```

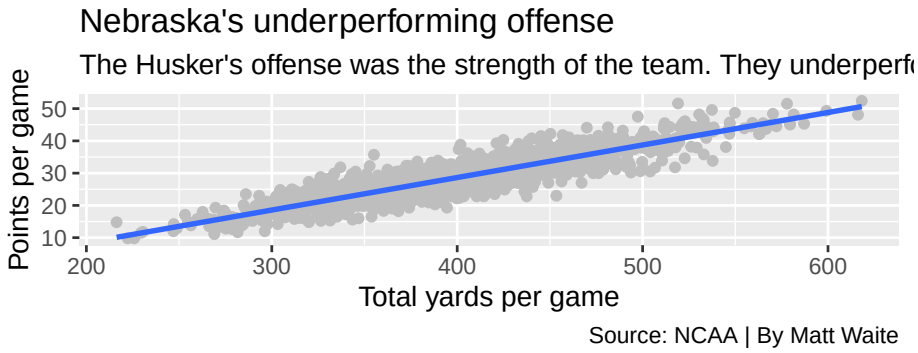
```
## `geom_smooth()` using formula 'y ~ x'
```



And now some labels.

```
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="Nebraska's underperforming offense",
```

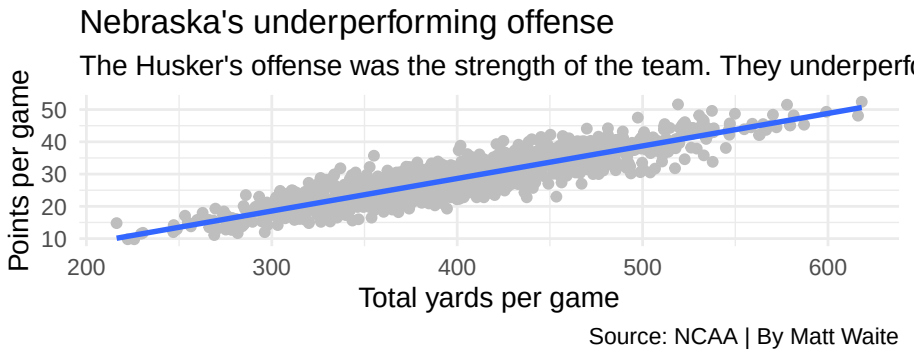
```
## `geom_smooth()` using formula 'y ~ x'
```



Let's get rid of the default plot look and drop the grey background.

```
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="Nebraska's underperformin
  theme_minimal()
```

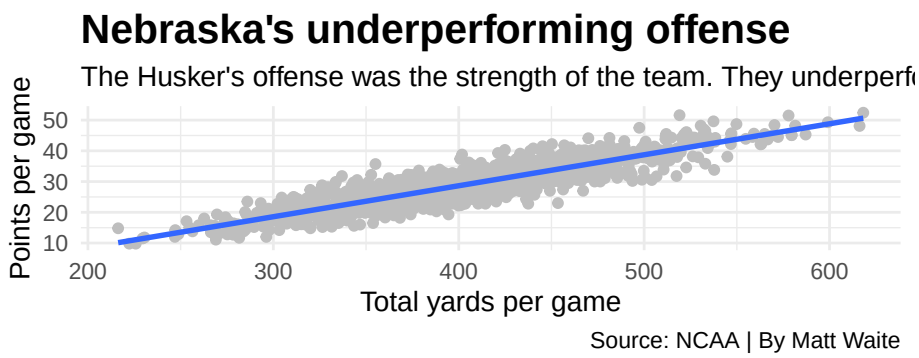
```
## `geom_smooth()` using formula 'y ~ x'
```



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(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="Nebraska's underperformin
  theme_minimal() +
  theme(
    plot.title = element_text(size = 16, face = "bold")
    )
```

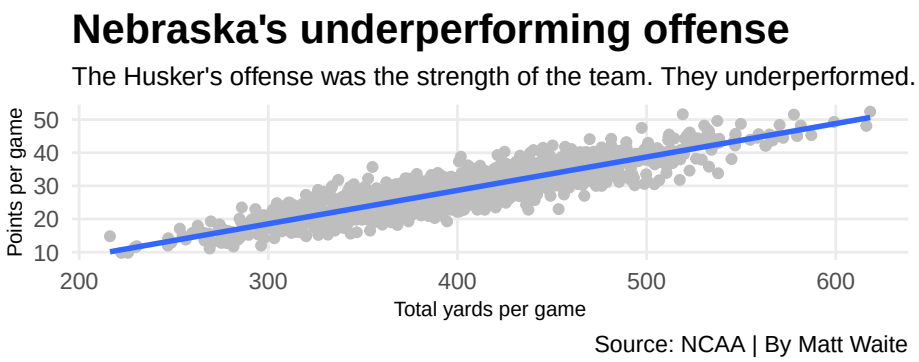
```
## `geom_smooth()` using formula 'y ~ x'
```



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(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="Nebraska's underperforming offense",
  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_smooth()` using formula 'y ~ x'
```

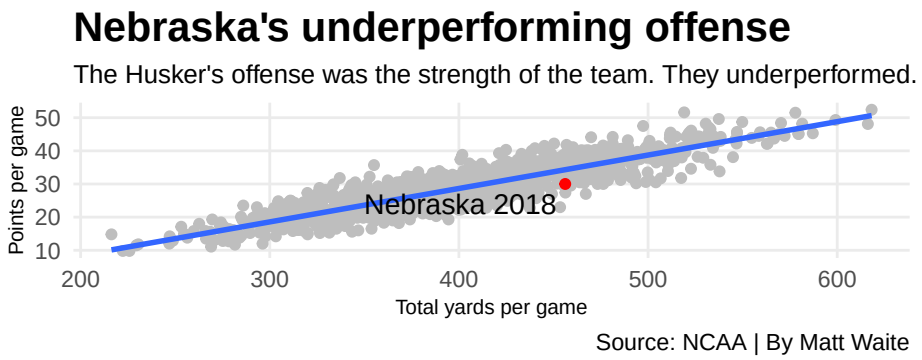


Missing from this graph is the context that the headline promises. Where is Nebraska? We haven't added it yet. So let's add a point and a label for it.

```
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="Nebraska's underperforming offense",
  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=nu, aes(x=`Yards/G`, y=`Points/G`), color="red") +
  geom_text_repel(data=nu, aes(x=`Yards/G`, y=`Points/G`, label="Nebraska 2018"))
```

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



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.

Chapter 29

Finishing Touches 2

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 Nebraska v. Wisconsin football game. Nebraska lost 41-24, but Wisconsin gained only 15 yards more than Nebraska did. 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.

```
nu <- c("Rushing"=111, "Passing"=407, 15)
wi <- c("Rushing"=370, "Passing"=163, 0)
```

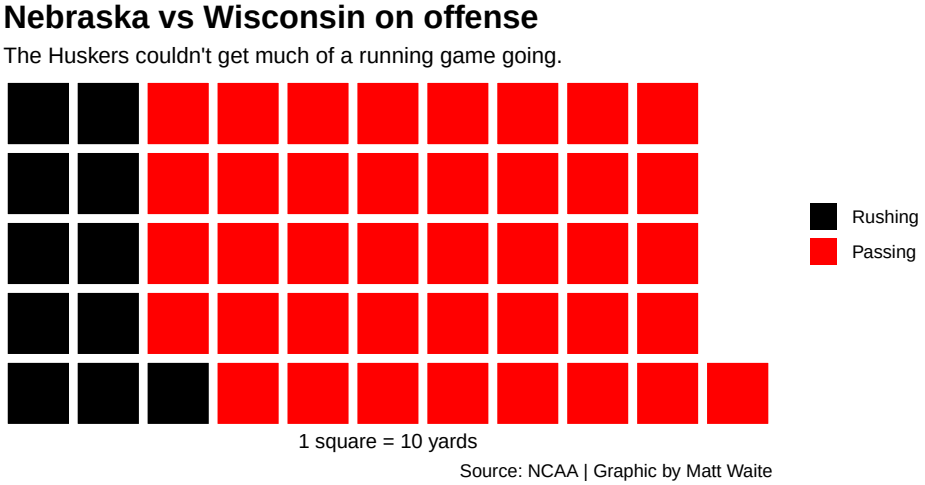
So what does the breakdown of Nebraska's night look like? How balanced was the offense?

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(nu/10, rows = 5, xlab="1 square = 10 yards", colors = c("black", "red", "white")
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 10),
    axis.title.y = element_blank()
  )
```



Note: The alignment of text sucks.

How to fix that? We can use ggsave to a pdf and fix it in Illustrator.

```
waffle(nu/10, rows = 5, xlab="1 square = 10 yards", colors = c("black", "red")) + labs
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 10),
    axis.title.y = element_blank()
  ) + ggsave("waffle.pdf")
```

But what if we're using a waffle iron? And what if we want to change the output size? It gets tougher.

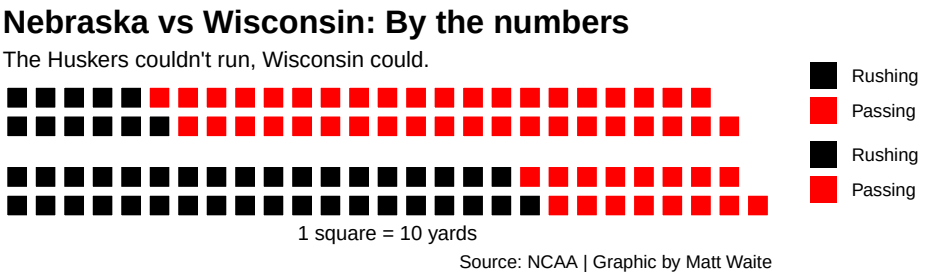
Truth is, I'm not sure what is going on with the sizing. You can try it and you'll find that the outputs are … unpredictable.

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(
   nu/10,
   rows = 2,
   colors = c("black", "red", "white")) +
   labs(title="Nebraska vs Wisconsin: By the numbers", subtitle="The Huskers couldn't run, Wiscon
   theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 10),
    axis.title.y = element_blank()
  ),
 waffle(
   wi/10,
   rows = 2,
   xlab="1 square = 10 yards",
   colors = c("black", "red", "white")) + labs(caption="Source: NCAA | Graphic by Matt Waite")
)
```



If you try to use ggsave on that, you'll only get the last waffle chart. Like I said, irons aren't really anything, so ggplot ignores them. So to do this, we have to use R's pdf capability.

Here's the same code, but wrapped in the R pdf functions. The first line says we're going to output this as a pdf with this name. Then my code, then `dev.off` to tell R that's what I want as a PDF. Don't forget that.

```
pdf("waffleiron.pdf")
iron(
 waffle(
   nu/10,
   rows = 2,
```

```
   colors = c("black", "red", "white")) +
   labs(title="Nebraska vs Wisconsin: By the numbers", subtitle="The Huskers couldn't r
   theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 10),
    axis.title.y = element_blank()
  ),
 waffle(
   wi/10,
   rows = 2,
   xlab="1 square = 10 yards",
   colors = c("black", "red", "white")) + labs(caption="Source: NCAA | Graphic by Matt
)
dev.off()
```

It probably still needs work in Illustrator, but less than before.

Chapter 30

Plotly

By John Strasheim

We've been working on making charts and graphs and outputting them as static images – png files. Why? Because images will embed into any website, work just fine on mobile, and require no special coding. But the wonder of the internet is that it's interactive. The trouble with interactive graphics, though, is the code can be exceedingly complicated and difficult for beginners to grasp, as is the case with the javascript visualization library D3. Or the tools are ultra simple and allow for minimal customization, such as tools like Tableau. The third problem is that the accessible interactive tools – the ones that don't require a ton of code knowledge – cost money to publish.

The library we're going to look at in this chapter is from a company called Plotly, which sits in between all these problems. You can create interactive graphs with point and click, but you can also do it in code. You can publish graphs for free, but if you're going to do it for a company or with a large audience, you need to pay. For our purposes, you'll see the power without needing to pony up.

You will need to install plotly. Go to the console – not in the notebook – and run `install.packages("plotly")`.

Then we'll load it:

```
library(tidyverse)
library(plotly)
```

To start out, we'll make a graph similar to something we've already done. We're going to use a dataset of batting stats from the 2019 season. This dataset has players who had more than 190 plate appearances and includes their basic stats and some advanced metrics.

```
batting <- read_csv("data/batting.csv")
```

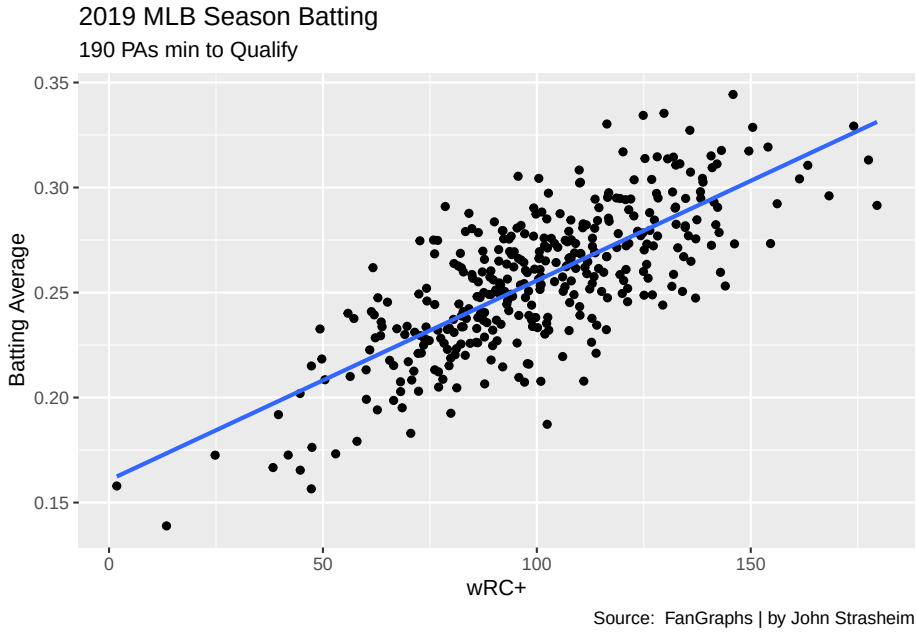
```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Name = col_character(),
##   Tm = col_character(),
##   Division = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

Let's look at the relationship between a player's batting average and wRC+ (weighted runs created plus).

```
ggplot() +
  geom_point(data=batting, aes(x=`wRC+`, y=AVG)) +
  geom_smooth(data=batting, aes(x=`wRC+`, y=AVG), method='lm', se=FALSE) +
  labs(x='wRC+', y='Batting Average', title= "2019 MLB Season Batting", subtitle="190 P
```

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



You can obviously add more aesthetics to make it look better, but you get the picture. If I'm a fan of sports though, obviously I want to see who the outlier points are or just scroll through and see each player at an individual location. We can always annotate data, but that process can be tedious.

Here is where plotly comes in.

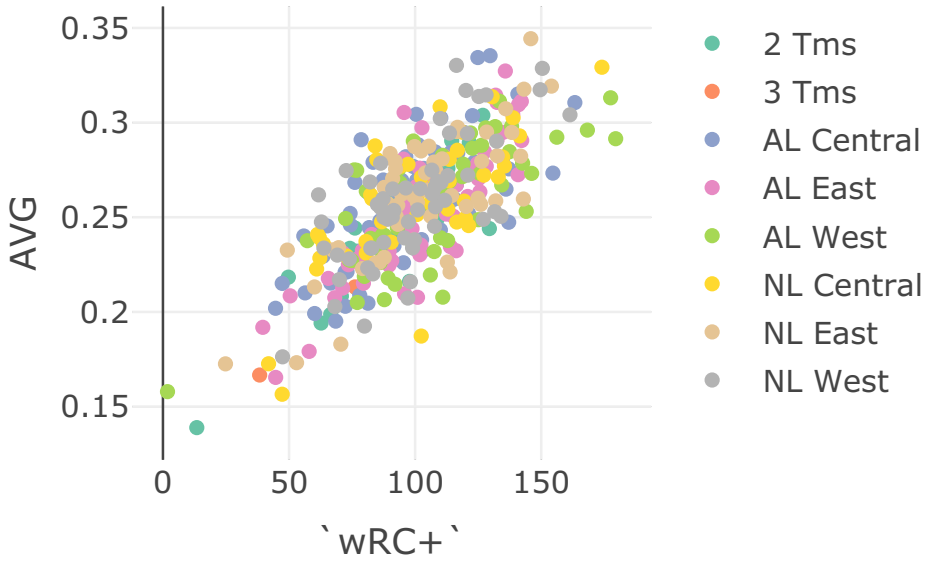
Plotly will make your visualizatons interactive. Additionally, you can zoom in on certain parts of the viz too. For example, you can drag a box around players with a .300 average, and see all the guys in that specific range.

Let's start simple with just the minimum needed for a scatterplot. For that, we need to specify our data source – `batting` – and set an X and a Y value, just like above. One difference? We prepend a ~ before the field names. We'll add a color to separate out players by division too.

```
plot_ly(data=batting, x= ~`wRC+`, y= ~AVG, color= ~Division)
```

```
## No trace type specified:
##   Based on info supplied, a 'scatter' trace seems appropriate.
##   Read more about this trace type -> https://plot.ly/r/reference/#scatter
```

```
## No scatter mode specifed:
##   Setting the mode to markers
##   Read more about this attribute -> https://plot.ly/r/reference/#scatter-mode
```



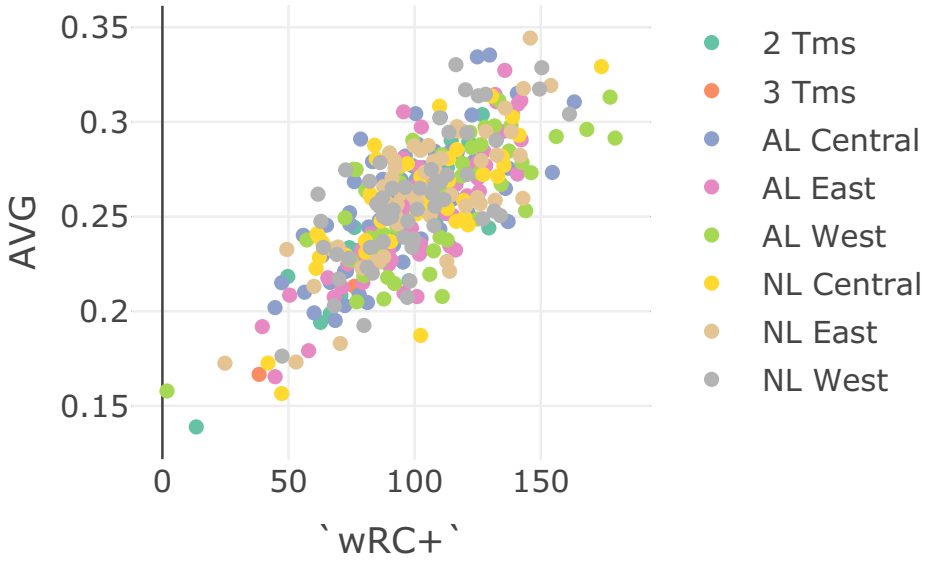
We get a chart, but hover over a point. Recognize those players? You can't unless you know each player's specific stats to divine who they are. That isn't very friendly, so let's add a hover element. Then we want to specify what we want our users to see when they hover over a data point, hence hoverinfo = "text". The next step will be to define what our text is. How that gets done is a little bit of HTML and a little bit of R. What is in quotes is what the users are going to see directly, what's after the quotes is what data is going to appear. So "Player:", Name translates to something like Player: Christian Yelich when the user hovers above Yelich's data point.

Then we add the HTML, `<br>`, or break. All this means is we're having a line

break so all of our data is not on one line. Simple. Do this process for whatever variables you want to have your users see. So for mine, I wanted my users to see the Player's name, wRC+, Batting Avg, and what division they are in.

```
plot_ly(data=batting, x= ~`wRC+`, y= ~`AVG`, color= ~`Division`,
        hoverinfo = "text",
        text = ~paste("Player:", Name,
                      '<br>wRC+:', `wRC+`,
                      '<br>AVG:', AVG,
                      '<br>Team:', Tm,
                      '<br>Division:', Division
                      ))
```

```
## No trace type specified:
##   Based on info supplied, a 'scatter' trace seems appropriate.
##   Read more about this trace type -> https://plot.ly/r/reference/#scatter

## No scatter mode specifed:
##   Setting the mode to markers
##   Read more about this attribute -> https://plot.ly/r/reference/#scatter-mode
```



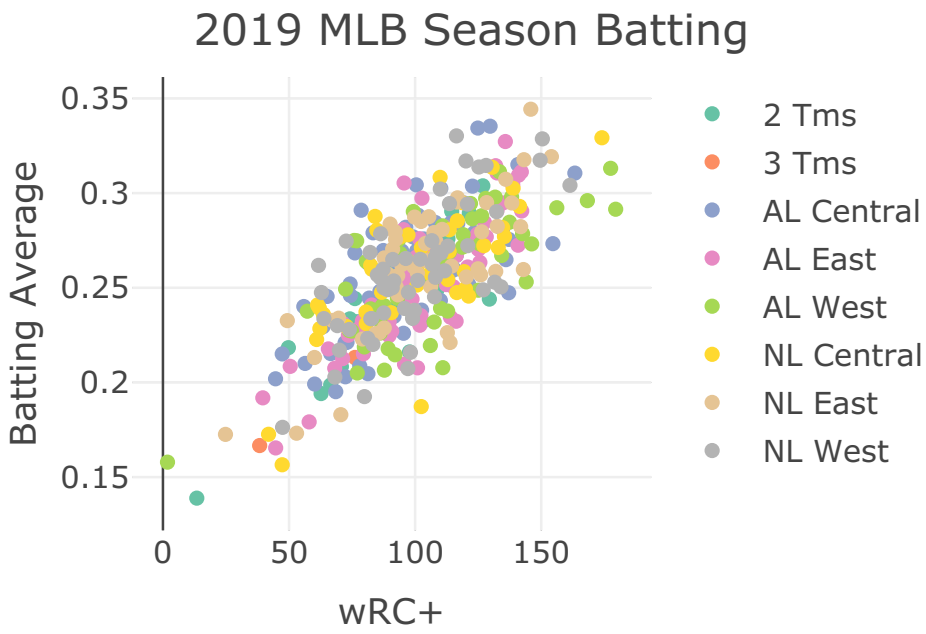
Now we can see each player a little better. If you look at the players on the farthest right, you'll find Mike Trout (shocker), Yordan Alvarez and Christian Yelich.

To finish, we're going to fix the layout a bit, very similar to how we've been doing it in ggplot. We're just telling plotly what we want the layout to be of our viz, starting with the title, and then doing the x and y axis names after that.

```
plot_ly(data=batting, x= ~`wRC+`, y= ~`AVG`, color= ~`Division`,
        hoverinfo = "text",
        text = ~paste("Player:", Name,
                      '<br>wRC+:', `wRC+`,
                      '<br>AVG:', AVG,
                      '<br>Team:', Tm,
                      '<br>Division:', Division
                      )) %>%
  layout(
    title = "2019 MLB Season Batting",
    xaxis = list(title = "wRC+"),
    yaxis = list(title = "Batting Average")
  )
```

```
## No trace type specified:
##   Based on info supplied, a 'scatter' trace seems appropriate.
##   Read more about this trace type -> https://plot.ly/r/reference/#scatter
```

```
## No scatter mode specifed:
##   Setting the mode to markers
##   Read more about this attribute -> https://plot.ly/r/reference/#scatter-mode
```



30.1 Publishing using Plotly

We want to now export our plotly visualization. First, you'll need to sign up for a free plotly account. Then you'll need to register your plotly username and your API key.

More info about how to do that can be found on Plotly's website.

For our purposes, we need to register our username and API key this way, where you put your username and API key where prompted:

```
Sys.setenv("plotly_username"="Enter your plotly username here")
Sys.setenv("plotly_api_key"="Enter your API key here")
```

Now run this line of code specifiying what variable you are exporting, and what you want the file to be named on plotly's servers. From plotly's website you can then do several different things like editing it on there, embedding it on websites, or create a shareable link.

To publish our chart, we need to save it to an object similar to how we've been creating dataframes. So something like this:

```
p <- plot_ly(data=batting, x= ~`wRC+`, y= ~`AVG`, color= ~`Division`,
        hoverinfo = "text",
        text = ~paste("Player:", Name,
                      '<br>wRC+:', `wRC+`,
                      '<br>AVG:', AVG,
                      '<br>Team:', Tm,
                      '<br>Division:', Division
                      )) %>%
  layout(
    title = "2019 MLB Season Batting",
    xaxis = list(title = "wRC+"),
    yaxis = list(title = "Batting Average")
  )
```

To publish it, we simply run the following, passing in our chart value `p` for plotly and we give it a filename.

```
api_create(p, filename="MLBOffense19")
```

If all goes well, a browser will pop up with your chart in it.

Chapter 31

Clustering

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

There's better ways.

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

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

For this exercise, I want to look at Cam Mack, Nebraska's point guard and probably the most interesting player on Fred Hoiberg's first team. This is Mack's first year in major college basketball – he played a year at a community college – so we don't have much to go on. But with a season in the books, who does Cam Mack compare to?

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

First thing we do is load some libraries and set a seed, so if we run this repeatedly, our random numbers are generated from the same base. If you don't have the

cluster library, just add it on the console with `install.packages("cluster")`

```
library(tidyverse)
library(cluster)

set.seed(1234)
```

I've gone and scraped stats for every player in this current season so let's load that up.

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

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Team = col_character(),
##   Player = col_character(),
##   Class = col_character(),
##   Pos = col_character(),
##   Height = col_character(),
##   Hometown = col_character(),
##   `High School` = col_character(),
##   Summary = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

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, Cam Mack is a guard, so let's just look at guards. Third, we want to limit the data to things that make sense to look at for Cam Mack – things like shooting, three point shooting, assists, turnovers and points.

```
playersselected <- players %>%
  filter(MP>0) %>% filter(Pos == "G") %>%
  select(Player, Team, Pos, MP, `FG%`, `3P%`, AST, 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.

```
playersscaled <- playersselected %>%
  select(MP, `FG%`, `3P%`, AST, TOV, PTS) %>%
  mutate_all(scale) %>%
  na.omit()
```

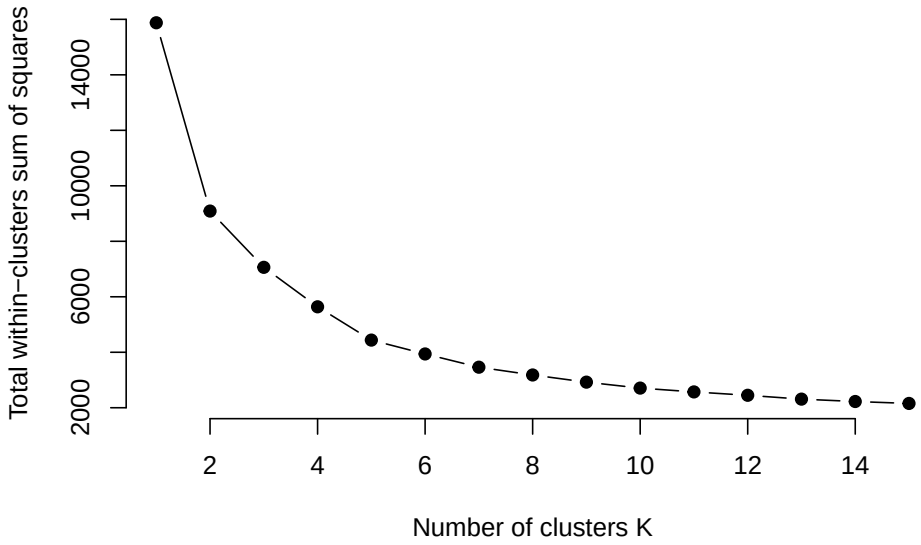
With k-means clustering, we decide how many clusters we want. Most often, researchers will try a handful of different cluster numbers and see what works. But there are methods for finding the optimal number. One method is called the Elbow method. One implementation of this, borrowed from the University 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)
```

```
## Warning: did not converge in 10 iterations

## Warning: did not converge in 10 iterations

## Warning: did not converge in 10 iterations

## Warning: did not converge in 10 iterations

## Warning: did not converge in 10 iterations

## Warning: did not converge in 10 iterations

## Warning: did not converge in 10 iterations
```

```
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 5 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.

```
k5 <- kmeans(playersscaled, centers = 5, nstart = 25)
```

Let's look at what we get.

```
k5
```

```
## K-means clustering with 5 clusters of sizes 242, 58, 495, 989, 863
##
## Cluster means:
##           MP         FG%         3P%        AST        TOV        PTS
## 1 -1.3602270 -1.98012908 -1.65325745 -0.9392522 -1.1172744 -1.1417360
## 2 -1.3785814  3.22934178  3.92207534 -0.9251265 -1.1485250 -1.1077564
## 3  1.2279089  0.26624900  0.17613521  1.5255303  1.5159849  1.4306790
## 4  0.5451701  0.14083415  0.12854961  0.1392254  0.2539075  0.3248641
## 5 -0.8549890  0.02411493 -0.04833668 -0.7090093 -0.7700257 -0.7982928
##
## Clustering vector:
##    [1] 3 4 4 4 4 5 5 5 5 4 4 3 4 5 5 5 3 4 5 5 1 1 3 4 4 4 4 4 5 2 4 4 3 4 4 5 1
##   [38] 3 3 4 4 5 4 5 5 3 4 4 4 5 5 4 3 4 4 4 4 5 5 1 4 3 5 4 4 5 5 5 5 5 4 4 3 4
##   [75] 4 5 1 5 1 3 4 4 4 5 5 5 5 3 4 4 4 4 5 1 3 4 4 5 5 5 3 4 5 4 5 5 5 5 2 4 4
##  [112] 5 5 5 4 5 1 3 4 4 4 5 5 5 3 4 4 5 1 1 3 4 4 5 5 5 3 4 4 4 4 5 5 3 3 4 4 4
##  [149] 4 1 3 4 4 4 5 5 5 1 3 4 4 4 4 3 5 1 3 3 4 4 4 4 5 5 4 4 4 4 5 5 5 5 5 5 3
##  [186] 4 1 1 3 4 4 4 5 5 1 5 3 3 4 4 5 5 5 2 1 3 4 4 3 4 4 5 5 5 1 3 4 5 5 5 1 1
##  [223] 3 3 4 4 4 4 4 1 3 4 4 3 5 3 4 3 4 4 5 5 5 4 4 4 4 4 4 5 1 3 3 3 4 4 4 5 5 1
```

```
##  [260] 1 3 3 4 4 4 5 5 5 1 3 3 4 4 4 5 5 5 3 4 4 4 1 3 3 4 4 5 1 4 3 3 4 3 4 5 5
##  [297] 3 4 4 4 5 5 5 5 5 3 4 4 5 1 5 1 1 3 3 4 4 5 5 3 3 4 4 4 5 5 4 4 4 4 5 5 5
##  [334] 3 3 4 4 4 4 5 3 3 4 5 5 5 1 3 4 4 4 4 5 1 3 3 4 5 4 5 5 1 1 3 4 4 4 4 5 5
##  [371] 5 3 3 4 4 5 5 5 5 5 5 5 3 3 4 3 4 5 5 5 1 3 3 4 5 5 3 3 4 4 5 1 3 3 5 5 5
##  [408] 3 3 4 5 5 1 3 4 4 4 4 5 5 5 2 3 4 4 4 4 5 5 5 3 3 4 5 5 4 3 4 4 4 4 5 1 3
##  [445] 3 4 4 5 5 5 5 3 3 4 4 5 1 3 4 4 4 5 5 5 5 4 3 4 4 4 5 5 5 3 4 4 5 5 5 5 1
##  [482] 3 3 4 4 5 5 1 1 3 3 4 5 5 4 5 3 3 4 4 4 5 1 1 4 4 4 3 5 5 3 4 4 4 4 5 1 3
##  [519] 4 4 4 5 5 2 5 3 3 4 4 4 5 1 2 3 3 4 3 5 5 5 1 3 4 4 4 4 5 5 5 5 3 3 4 5 5
##  [556] 5 3 3 4 4 4 5 5 1 4 4 4 4 5 1 1 1 4 4 4 5 5 5 1 3 3 3 5 5 5 5 1 1 3 3 4 4
##  [593] 5 4 3 3 4 4 4 4 5 5 2 3 4 4 5 1 1 1 3 4 4 4 4 5 5 1 1 3 3 4 4 5 3 4 4 4 5
##  [630] 5 1 3 4 3 4 5 5 1 3 4 4 4 5 2 1 3 4 4 4 4 5 4 4 4 4 4 4 5 5 5 3 4 3 5 5 5
##  [667] 3 4 4 3 5 5 5 1 2 3 4 4 5 5 5 3 3 4 4 4 5 5 5 1 3 4 4 4 3 5 5 4 4 4 5 1 1
##  [704] 1 4 3 4 4 4 4 2 5 1 4 3 3 4 3 5 1 3 3 4 4 5 5 1 5 4 3 4 5 5 4 4 3 4 4 5 4
##  [741] 5 5 3 3 4 5 3 3 4 5 5 5 5 2 1 3 4 4 5 5 3 4 4 4 4 5 5 3 3 3 4 4 5 2 4 4 4
##  [778] 5 5 5 5 5 3 4 4 4 4 5 5 1 1 4 4 4 4 4 4 2 5 2 5 5 4 4 3 4 4 4 5 3 4 4 4 5 5
##  [815] 5 4 3 3 4 4 1 5 1 1 3 4 4 4 5 1 3 4 4 5 5 5 5 5 5 4 4 4 4 4 5 1 1 3 4 4 4
##  [852] 5 5 4 5 3 4 4 4 5 3 4 5 5 5 3 4 4 4 5 5 5 3 3 4 4 5 5 5 2 3 4 4 4 1 4 3 4
##  [889] 5 4 5 5 5 2 3 4 4 1 5 5 1 4 3 4 4 5 4 4 3 4 5 5 1 1 1 3 3 4 4 5 1 1 3 4 5
##  [926] 5 5 4 4 4 4 5 4 5 5 5 5 3 3 4 3 5 5 5 5 5 3 4 4 4 5 5 3 4 4 4 5 5 5 3 4 4
##  [963] 4 4 5 5 3 4 4 5 5 5 1 1 3 3 4 4 4 5 5 5 1 3 3 4 5 5 5 5 5 5 3 3 4 5 4 1 3
## [1000] 4 4 5 5 5 1 3 3 4 4 5 5 5 5 5 3 4 4 4 4 4 5 5 1 4 3 4 4 4 4 4 5 2 1 3 3 4
## [1037] 4 1 5 3 4 4 4 5 1 5 4 3 4 5 4 5 4 1 4 3 4 4 4 5 1 3 4 3 3 4 5 5 1 3 4 3 5
## [1074] 5 1 1 3 4 3 4 5 5 5 5 5 5 5 3 4 4 5 5 5 5 3 3 3 4 5 5 5 5 1 3 4 4 4 5 4 5
## [1111] 5 2 3 3 5 4 5 5 5 5 3 3 4 4 4 5 4 5 5 5 1 4 4 4 5 4 4 5 5 2 5 1 3 4 4 4 5
## [1148] 5 3 3 3 4 5 1 3 3 4 4 5 1 4 3 4 4 4 5 5 5 3 3 4 4 4 5 1 5 3 4 4 5 5 2 1 5
## [1185] 1 3 4 3 5 1 5 1 3 4 4 5 1 5 1 5 1 3 4 4 4 5 5 5 1 5 3 4 3 4 4 5 3 3 4 4 5
## [1222] 3 3 4 5 5 4 3 4 4 4 5 5 3 3 4 4 4 1 5 2 3 4 4 4 5 5 1 4 4 4 4 5 4 5 5 3 3
## [1259] 4 5 5 1 1 5 3 3 3 4 5 5 1 3 4 4 4 4 1 3 4 4 5 1 1 3 3 4 4 4 5 2 5 4 3 3 4
## [1296] 4 5 1 3 4 4 4 4 5 2 1 3 4 5 5 5 5 1 1 3 4 4 4 4 5 5 4 4 4 4 4 5 3 4 4 4 4 5
## [1333] 5 1 5 3 3 4 4 4 5 5 5 4 4 4 3 4 5 5 4 3 4 5 4 4 5 5 4 3 5 4 5 5 2 1 3 3 4
## [1370] 4 4 5 1 1 3 4 4 4 5 1 2 2 4 4 4 4 5 4 5 1 3 4 4 5 5 5 5 1 3 3 4 5 5 5 3 4
## [1407] 3 5 5 2 3 4 4 4 4 5 5 1 1 4 4 4 4 4 4 5 1 2 1 3 5 5 5 5 5 1 3 3 4 4 4 5 5 2 1
## [1444] 1 1 4 5 5 5 5 3 4 4 4 4 5 5 5 1 4 4 4 4 4 4 5 3 4 4 4 4 4 5 5 1 5 3 3 5 5 5 5 5
## [1481] 3 4 4 5 5 5 1 3 3 4 5 5 5 5 5 3 3 4 4 5 5 3 3 3 4 4 5 2 1 4 3 4 4 4 5 5 4
## [1518] 4 4 4 5 5 5 5 5 4 4 4 4 5 5 5 3 3 4 4 5 1 1 3 3 3 4 4 5 3 4 4 5 5 5 5 4 4
## [1555] 4 4 4 3 4 1 4 3 3 5 3 4 4 4 5 5 5 3 4 5 5 5 5 5 1 3 4 3 4 5 5 5 5 2 3 4 4
## [1592] 4 5 3 4 3 5 1 5 5 4 4 4 4 4 4 4 5 1 1 4 4 4 4 4 4 4 5 5 1 3 4 3 4 5 5 4 5 5 2
## [1629] 1 1 1 3 3 3 4 4 5 3 4 4 4 4 5 1 4 4 4 4 4 4 5 5 1 3 3 4 5 4 5 5 2 1 3 4 4 4
## [1666] 5 1 1 4 4 4 5 2 3 3 4 4 4 5 5 5 4 3 4 4 4 5 5 5 3 4 4 4 5 1 5 3 4 4 4 5 4
## [1703] 5 5 5 3 3 4 4 5 3 4 4 4 4 5 5 5 3 4 4 5 5 5 5 2 3 3 4 4 4 5 5 5 4 3 4 3 4
## [1740] 3 4 4 4 4 5 1 3 4 4 5 5 1 2 3 3 4 4 5 5 1 3 4 4 4 4 1 4 3 4 4 5 5 3 4 4 5
## [1777] 5 5 5 1 4 3 3 4 5 5 5 3 4 4 4 2 5 5 5 1 3 3 3 4 4 4 5 2 3 3 4 4 5 3 4 4 4
## [1814] 5 4 5 1 5 2 1 4 3 3 5 5 5 3 3 4 4 5 5 1 1 3 3 4 4 5 5 5 3 3 3 3 4 1 3 4 3
## [1851] 4 4 5 5 5 5 4 4 4 3 3 5 2 3 4 3 5 5 5 5 1 3 4 4 4 5 1 4 3 3 5 4 5 2 1 3 4
## [1888] 3 5 5 5 1 4 4 4 4 4 4 5 1 5 4 4 4 5 4 5 5 3 4 4 5 5 4 3 4 4 5 5 5 2 3 4 4
## [1925] 4 4 5 5 5 3 4 3 4 5 5 1 1 3 4 4 5 5 5 4 3 4 5 4 5 3 4 4 5 5 5 1 4 4 4 4 5
```

```
## [1962] 3 4 4 4 4 4 5 2 2 3 3 3 4 4 5 5 1 1 1 3 4 4 3 4 5 5 5 5 5 4 4 5 4 5 5 1 3
## [1999] 3 4 4 5 5 5 3 4 4 3 5 4 5 5 5 3 3 3 4 4 1 3 4 4 4 4 5 5 3 4 4 4 1 1 2 2 3
## [2036] 4 5 4 1 3 3 4 5 1 3 4 4 4 4 4 5 5 2 1 3 4 4 5 5 1 3 4 5 5 5 1 1 5 3 3 4 4
## [2073] 5 5 5 5 3 3 3 5 5 5 1 3 3 4 4 4 5 5 5 1 3 4 4 3 5 5 5 1 3 3 4 4 4 5 5 2 1
## [2110] 4 4 4 4 4 5 5 5 3 3 3 4 5 5 5 5 4 4 5 5 5 1 4 3 4 4 4 4 5 5 3 4 3 4 4 5 1
## [2147] 3 4 4 4 5 5 1 5 3 3 4 4 5 5 5 3 3 4 4 4 5 5 2 3 4 4 5 5 2 1 2 3 3 4 4 5 5
## [2184] 3 4 4 4 5 5 5 1 4 3 4 4 4 4 5 5 2 5 4 4 3 4 4 5 5 5 5 5 4 4 4 3 4 5 5 1 3
## [2221] 4 4 4 5 5 5 5 3 4 4 5 5 3 3 4 4 5 5 5 5 5 3 3 4 4 4 5 5 1 5 3 4 4 4 5 5 5
## [2258] 1 4 4 3 4 4 5 5 3 4 3 4 4 5 5 5 4 4 4 4 5 4 5 1 5 3 3 3 4 5 5 5 4 3 4 4 5
## [2295] 5 4 4 4 4 4 5 5 5 2 3 4 4 4 5 5 1 3 4 4 4 5 5 4 4 4 4 5 4 5 5 3 4 4 5 5 5
## [2332] 4 3 4 4 4 4 5 5 5 5 3 4 3 4 5 5 5 2 3 3 3 5 1 3 3 4 5 5 2 1 4 3 4 4 4 4 4
## [2369] 4 4 4 1 1 3 3 5 5 3 3 4 4 5 1 3 4 4 5 5 5 3 3 4 4 4 1 5 3 4 5 5 5 5 1 1 3
## [2406] 3 4 4 4 5 5 5 2 3 4 4 4 1 5 5 4 4 3 4 4 4 5 5 5 4 3 4 4 4 5 5 5 1 3 3 4 5
## [2443] 5 3 3 4 4 4 5 5 2 3 4 4 4 5 5 5 3 4 4 4 4 4 5 2 3 3 4 4 5 5 5 5 3 4 3 4 4
## [2480] 5 1 1 3 3 4 4 5 5 1 3 3 4 3 4 4 5 5 5 3 3 5 5 5 1 5 3 3 4 4 5 5 1 5 1 3 4
## [2517] 4 4 4 3 3 4 4 4 4 1 1 5 5 4 5 5 5 5 5 1 3 4 4 4 5 5 5 1 3 4 4 4 5 5 3 4 4
## [2554] 5 5 5 5 3 4 4 5 5 5 1 1 1 3 4 4 4 5 5 1 1 3 3 4 4 5 5 5 1 3 3 4 4 4 5 1 5
## [2591] 3 3 4 4 5 5 3 4 4 4 4 5 5 5 5 5 5 3 3 3 5 5 5 5 1 4 4 3 4 1 5 1 1 3 3 4 5
## [2628] 1 1 3 4 4 4 5 5 5 5 5 5 3 4 4 4 5 5 5 5
##
## Within cluster sum of squares by cluster:
## [1]  319.4278  250.5515 1094.8283 1341.0495 1430.7781
##  (between_SS / total_SS =  72.1 %)
##
## 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 1 and 2 are pretty large compared to others. That's notable. Then we can look at the cluster means. For reference, 0 is going to be average. So group 1 is below average on minutes played. Group 2 is slightly above, group 5 is well above.

So which group is Cam Mack in? Well, first we have to put our data back together again. In K5, 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, k5$cluster)
```

Now we have a dataframe called playercluster that has our player names and what cluster they are in. The fastest way to find Cam Mack 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 Mack is in Group 1 in my data.

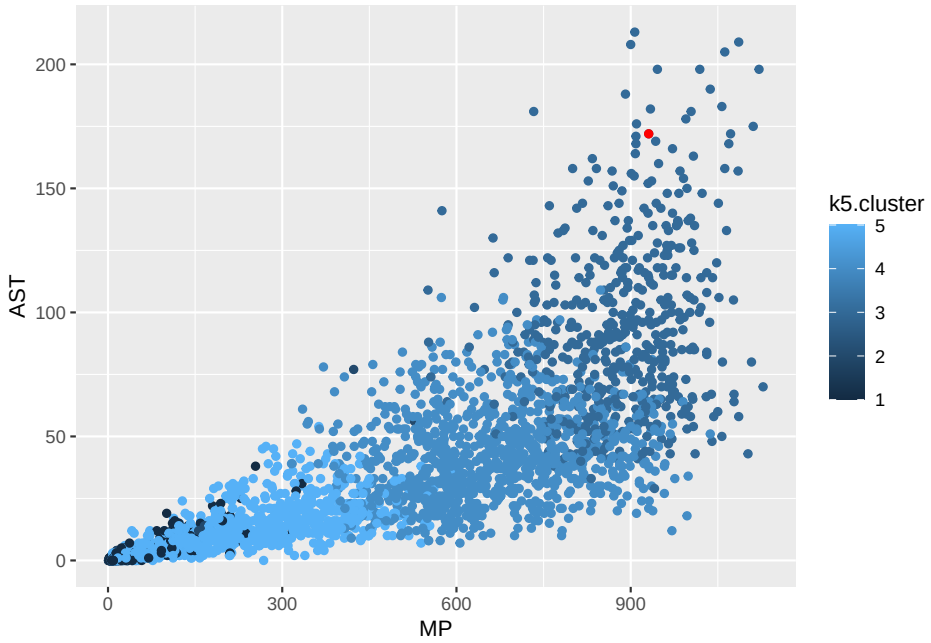
We now have a dataset and can plot it like anything else. Let's get Cam Mack and then plot him against the rest of college basketball on assists versus minutes played.

```
cm <- playercluster %>% filter(Player == "Cam Mack")

cm
```

```
##     Player                 Team Pos  MP  FG.  X3P. AST TOV PTS k5.cluster
## 1 Cam Mack Nebraska Cornhuskers   G 931 0.39 0.339 172  71 324           3
```

So Cam's in cluster 5, which if you look at our clusters, puts him in the cluster with all above average metrics. What does that look like? We know Cam was an assist machine, so where do group 5 people grade out on assists?

```
ggplot() +
  geom_point(data=playercluster, aes(x=MP, y=AST, color=k5.cluster)) +
  geom_point(data=cm, aes(x=MP, y=AST), color="red")
```



Not bad, not bad. But who are Cam Mack's peers? If we look at the numbers in Group 5, there's 495 of them. So let's limit them to just Big Ten guards. Unfortunately, my scraper didn't quite work and in the place of Conference is the coach's name. So I'm going to have to do this the hard way and make a list of Big Ten teams and filter on that. Then I'll sort by minutes played.

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

playercluster %>% filter(k5.cluster == 5) %>% filter(Team %in% big10) %>% arrange(desc
```

```
##               Player                    Team Pos  MP   FG.  X3P. AST TOV PTS
## 1  Da'Monte Williams Illinois Fighting Illini   G 592 0.302 0.211  37  20  64
## 2      Tre' Williams Minnesota Golden Gophers   G 445 0.286 0.263  23  24  84
## 3    Armaan Franklin         Indiana Hoosiers   G 407 0.346 0.241  37  28 106
## 4    Trevor Anderson       Wisconsin Badgers   G 341 0.349 0.280  37  16  48
## 5        Matej Kavas    Nebraska Cornhuskers   G 320 0.384 0.338  12   8 116
## 6        Kyle Ahrens Michigan State Spartans   G 318 0.433 0.417  13  19  88
## 7     Anthony Gaines   Northwestern Wildcats   G 265 0.375 0.313  17  11  59
## 8    Jordan Bohannon           Iowa Hawkeyes   G 250 0.298 0.328  33  11  88
## 9     Charlie Easley    Nebraska Cornhuskers   G 239 0.318 0.192  10  10  46
## 10      Foster Loyer Michigan State Spartans   G 226 0.446 0.462  29  14  88
## 11   Serrel Smith Jr      Maryland Terrapins   G 191 0.240 0.250  12  14  34
## 12      Adrien Nunez     Michigan Wolverines   G 164 0.294 0.256   2   6  42
## 13        Ryan Greer   Northwestern Wildcats   G 160 0.304 0.333  18   4  19
## 14      Walt McGrory       Wisconsin Badgers   G  57 0.385 0.286   5   1  15
## 15     Samari Curtis    Nebraska Cornhuskers   G  49 0.400 0.444   1   2  14
## 16     Conner George Michigan State Spartans   G  48 0.333 0.286   1   2  15
## 17   Tyler Underwood Illinois Fighting Illini   G  45 0.222 0.333   3   1   9
## 18      Jack Hoiberg Michigan State Spartans   G  41 0.333 0.250  12   2  19
## 19      Danny Hummer     Ohio State Buckeyes   G  37 0.300 0.286   5   3   9
## 20        Tommy Luce     Purdue Boilermakers   G  29 0.308 0.333   6   2  10
## 21        Austin Ash           Iowa Hawkeyes   G  24 0.250 0.273   2   2  12
## 22     Jared Wulbrun     Purdue Boilermakers   G  17 0.500 0.500   2   2   3
## 23      Tino Malnati   Northwestern Wildcats   G  10 0.500 0.000   2   0   2
## 24       Nick Brooks Rutgers Scarlet Knights   G   8 0.500 0.000   1   1   2
## 25   Stephen Beattie Penn State Nittany Lions   G   7 0.500 0.500   0   0   6
##    k5.cluster
## 1           5
## 2           5
## 3           5
## 4           5
## 5           5
## 6           5
## 7           5
## 8           5
## 9           5
## 10          5
## 11          5
## 12          5
## 13          5
## 14          5
```

```
## 15          5
## 16          5
## 17          5
## 18          5
## 19          5
## 20          5
## 21          5
## 22          5
## 23          5
## 24          5
## 25          5
```

So there are the 11 guards most like Cam Mack in the Big Ten. Safe to say, these are the 11 best guards in the conference.

31.1 Advanced metrics

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

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

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

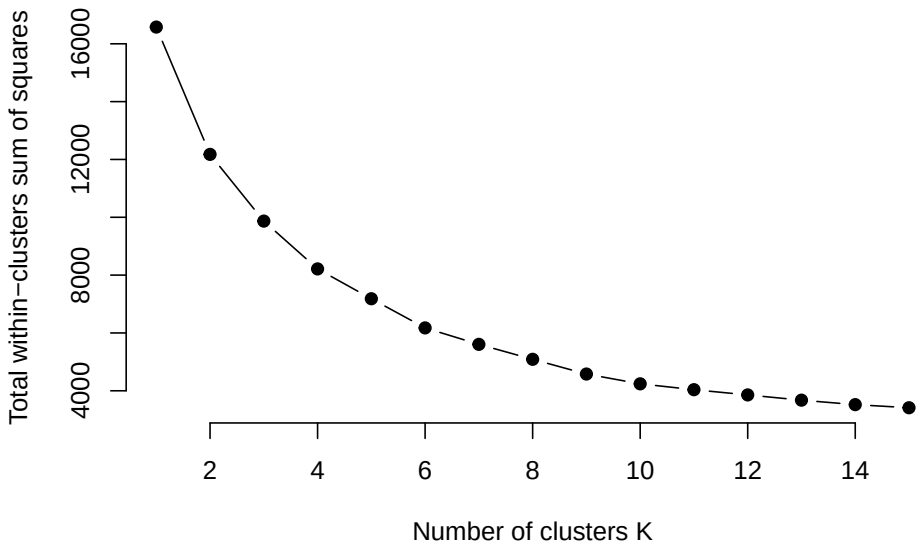
Now to scale them.

```
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)

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 5 again.

```
advk5 <- kmeans(playersadvscaled, centers = 5, nstart = 25)
```

What do we have here?

```
advk5
```

```
## K-means clustering with 5 clusters of sizes 104, 1253, 9, 632, 766
##
## Cluster means:
##          PER        TS%      PProd       AST%      WS/40        BPM
## 1 -2.6922417 -2.6802561 -1.1497380 -1.0109009 -2.8398288 -2.8455990
## 2  0.1594366  0.3741799 -0.2279088 -0.2650562  0.2130658  0.2118878
## 3  8.9269330  4.3359455 -1.1326482  2.2478231  8.0858448  6.3926472
## 4 -0.6089002 -0.6736304 -0.8322766 -0.2324539 -0.5651578 -0.6493960
## 5  0.5022214  0.2566711  1.2288970  0.7362004  0.4083263  0.5004325
##
## Clustering vector:
##    [1] 5 2 2 2 2 4 2 4 2 5 2 5 2 2 4 2 5 2 2 2 4 4 2 1 5 2 2 2 2 4 2 2 5 5 5 2 2
##   [38] 2 4 5 5 2 2 2 2 2 4 4 5 2 2 2 2 2 2 1 5 5 2 2 5 2 2 2 2 5 5 5 2 2 2 4 4 4
```

```
##   [75] 4 4 5 2 5 2 5 4 1 2 4 5 2 2 2 2 4 2 4 5 5 2 2 2 4 4 5 5 2 4 4 4 4 5 2 2 2
##  [112] 2 2 4 4 2 5 5 2 2 2 2 4 4 5 5 2 4 2 4 4 5 5 2 2 4 2 3 1 5 2 2 2 4 2 5 2 2
##  [149] 2 2 4 2 1 5 5 2 2 5 2 4 5 5 2 4 2 4 3 4 1 5 5 2 2 2 5 2 4 5 5 2 5 2 2 2 4
##  [186] 5 5 5 2 2 2 2 2 2 4 5 5 4 4 5 2 2 2 4 2 4 2 5 5 2 2 4 2 2 3 4 5 2 2 5 2 2
##  [223] 2 4 2 4 5 2 2 2 2 4 4 5 5 2 2 2 2 2 4 5 2 5 5 2 5 2 5 2 2 2 2 2 4 1 5 5 5
##  [260] 2 4 2 1 5 5 5 2 2 2 2 2 4 5 1 1 5 5 2 2 2 2 2 4 4 5 5 2 2 2 2 2 2 2 5 2 2
##  [297] 2 5 4 5 5 5 2 4 1 5 5 5 2 5 2 2 2 5 5 2 5 2 2 2 2 2 5 2 4 4 4 4 4 4 5 5 2
##  [334] 2 4 4 5 5 2 2 4 2 2 5 2 4 2 4 2 4 5 5 2 2 4 2 2 5 5 2 4 4 4 4 5 2 5 4 2 4
##  [371] 4 5 5 2 2 4 4 4 4 4 5 2 5 2 5 2 2 5 5 5 2 2 2 2 4 4 4 4 2 4 5 5 2 5 4 2 4
##  [408] 2 1 5 5 2 2 4 4 5 5 5 2 2 2 1 5 5 4 4 1 2 1 5 5 2 2 4 4 4 5 2 2 2 4 4 4 4
##  [445] 2 5 5 2 5 2 2 2 4 5 5 2 2 2 4 5 5 2 2 2 2 2 1 1 5 5 2 2 4 4 4 4 5 5 2 2 4
##  [482] 4 1 5 2 2 2 4 4 2 4 5 5 2 2 2 2 4 4 5 5 4 4 4 4 4 4 1 5 5 2 2 4 2 4 1 5 5
##  [519] 2 2 5 4 4 5 5 5 2 4 4 4 4 4 5 2 2 5 4 4 4 1 1 5 2 2 4 4 4 4 5 2 2 2 4 4 2
##  [556] 2 5 5 2 2 2 2 2 4 2 5 5 5 5 2 2 4 1 5 2 2 2 5 2 2 4 2 5 5 2 2 4 2 5 5 2 2 4
##  [593] 2 2 4 5 2 2 4 2 4 4 1 5 5 2 2 2 4 2 1 5 5 5 2 2 4 2 4 1 5 5 2 2 2 2 4 5 5
##  [630] 2 2 2 2 4 4 2 5 2 2 2 4 1 1 5 5 5 2 2 2 4 4 4 5 5 5 2 2 5 5 2 2 2 4 2 1 5
##  [667] 2 5 2 2 2 4 5 2 4 4 4 3 1 4 5 5 2 2 5 2 2 2 2 5 2 5 4 2 4 1 5 2 5 2 2 2 5
##  [704] 5 2 5 2 2 2 4 3 5 2 2 2 4 4 5 5 2 2 2 4 2 4 2 2 1 5 2 5 2 5 4 2 2 2 5 5 2
##  [741] 4 4 4 5 5 2 2 2 2 2 4 1 5 5 5 2 5 2 4 1 5 5 2 2 2 4 4 4 5 5 5 2 2 2 4 5 5
##  [778] 5 5 4 2 2 4 2 5 5 5 2 5 5 2 2 2 2 4 2 4 5 5 2 4 4 1 5 5 2 2 2 5 2 5 5 5 2
##  [815] 2 2 5 5 5 2 2 2 4 2 4 5 2 2 5 4 2 4 2 4 5 2 2 2 2 2 4 2 4 4 5 1 5 5 5 2 2
##  [852] 4 4 5 5 2 2 2 2 4 5 5 5 2 2 4 2 4 4 4 5 2 4 2 2 4 1 5 2 2 2 2 2 4 2 2 5 2
##  [889] 5 2 2 2 4 2 5 2 5 2 2 2 5 2 5 5 5 2 2 5 2 2 2 4 4 5 5 2 2 4 2 2 5 5 5 2 4
##  [926] 4 4 2 5 2 2 2 2 1 5 5 2 2 4 4 2 4 2 5 2 2 4 4 4 4 5 5 5 2 2 5 2 4 2 4 4 4
##  [963] 4 4 5 5 2 5 2 4 4 5 5 2 4 5 2 2 2 5 2 4 4 2 2 4 5 5 5 5 2 4 4 2 2 5 5 5 2
## [1000] 2 2 5 2 2 2 2 2 2 4 5 5 2 2 2 2 4 5 2 2 2 4 2 4 1 5 5 5 2 2 2 2 2 4 5 5 2 2
## [1037] 2 2 2 2 2 5 5 2 2 2 1 5 5 2 2 4 2 4 5 5 2 2 2 2 2 4 2 5 5 2 2 2 2 4 2 4 5
## [1074] 5 5 2 2 2 4 2 2 4 5 5 5 2 4 4 5 2 5 2 2 4 2 2 5 4 2 2 2 4 1 4 5 2 2 2 4 4
## [1111] 5 5 5 5 4 2 4 4 4 5 5 5 2 4 4 1 5 5 5 5 2 2 4 2 4 2 2 5 2 4 2 4 4 4 5 5 5
## [1148] 2 2 2 2 2 4 5 5 2 2 2 4 2 4 2 5 5 2 2 2 4 2 4 5 5 2 2 2 2 5 2 2 2 2 5 5 2
## [1185] 2 2 2 2 5 2 2 1 5 2 5 2 2 2 5 5 5 5 4 1 5 5 5 2 2 2 1 5 5 2 2 2 2 4 2 5 5
## [1222] 2 2 2 4 4 4 5 2 5 4 2 3 4 2 4 5 2 5 2 4 2 4 5 2 5 4 4 4 4 4 1 5 2 2 2 2 2
## [1259] 4 4 2 5 2 5 5 2 4 5 5 2 2 4 5 5 2 4 4 5 5 5 2 2 4 4 5 5 2 2 4 4 2 2 5 2 2
## [1296] 2 4 4 2 2 5 2 2 2 2 2 2 2 5 5 2 2 2 2 4 4 5 5 5 4 4 4 1 5 5 2 5 2 4 5 2 2 4
## [1333] 4 1 5 5 2 2 2 2 2 2 4 5 5 5 5 2 2 4 5 5 2 2 4 2 4 5 5 2 2 2 5 2 4 4 5 5 5 2
## [1370] 2 4 2 5 5 5 2 5 2 4 5 2 2 2 2 4 4 4 5 5 5 2 2 2 2 4 2 5 2 2 5 5 2 4 5 5 5
## [1407] 2 2 2 2 2 5 5 2 2 2 4 4 2 4 5 5 2 2 4 2 4 1 5 5 5 5 4 4 2 2 2 4 4 4 4 4 4
## [1444] 4 4 5 2 5 2 5 4 2 4 5 5 2 2 4 2 5 5 5 2 2 2 5 2 2 2 4 4 1 4 1 2 5 5 2 2 2
## [1481] 4 3 2 1 5 2 4 4 4 4 4 5 5 2 5 2 4 4 4 4 1 1 2 2 2 4 4 5 5 2 2 2 2 2 4 1 2
## [1518] 5 5 2 2 2 2 5 2 5 5 2 2 2 4 2 5 5 2 2 4 4 4 5 5 2 2 4 2 4 5 5 2 2 4 4 4 4
## [1555] 5 5 2 2 2 4 5 5 2 2 5 4 2 4 5 5 2 4 2 2 2 5 2 2 2 4 2 4 4 4 5 5 2 2 2 2 4
## [1592] 2 5 5 2 2 4 4 1 5 5 5 2 2 4 5 5 2 2 2 2 2 2 2 2 2 2 5 2 1 5 5 5 2 2 5 5 5
## [1629] 2 2 2 2 5 4 4 4 4 4 4 4 4 5 5 5 2 2 2 2 2 2 5 2 2 5 4 5 5 5 4 4 4 4 2 2 2 2
## [1666] 2 4 4 4 4 5 2 2 2 2 2 2 2 2 4 5 5 5 2 4 2 4 4 4 3 4 4 1 5 5 5 4 4 4 4 5 2 2
## [1703] 2 2 4 1 5 5 5 5 5 2 2 2 1 5 5 2 2 2 4 2 2 2 4 5 2 5 2 4 4 1 1 5 5 5 2 2 1 5
## [1740] 5 5 2 2 4 2 2 4 1 2 5 5 2 2 4 4 4 2 5 5 2 2 2 4 4 5 2 2 2 2 2 2 4 2 5 5 2
```

```
## [1777] 2 2 5 2 2 4 2 2 4 2 4 5 2 2 4 5 4 2 2 5 5 2 2 5 4 4 4 5 5 5 5 2 5 2 4 4 2
## [1814] 2 4 5 4 5 4 4 4 4 5 5 5 2 4 2 4 2 1 5 2 2 2 4 4 2 5 5 2 2 2 4 5 2 2 2 2 2
## [1851] 2 1 5 2 5 2 2 4 2 5 5 2 4 2 2 4 4 4 2 1 5 5 5 5 2 2 2 2 5 5 2 2 4 5 2 2 2
## [1888] 2 4 2 4 2 2 4 5 5 5 2 2 2 5 5 2 2 2 2 4 1 5 5 2 2 4 4 5 5 5 5 5 5 2 4 1 5
## [1925] 5 5 5 2 4 2 2 4 2 5 5 2 5 5 2 4 4 5 2 5 2 4 4 2 4 5 5 2 2 2 4 4 1 5 5 5 4
## [1962] 2 4 2 1 5 5 5 2 2 4 2 1 5 2 2 5 5 2 2 4 4 5 2 2 2 2 4 4 5 2 2 2 4 2 5 5 2
## [1999] 2 2 2 2 5 4 4 5 2 2 2 2 2 2 4 4 5 2 5 2 2 2 4 1 5 2 5 4 2 4 5 5 2 2 4 2 2
## [2036] 5 5 2 2 2 4 4 2 5 2 2 2 5 2 2 4 2 2 2 5 2 4 5 5 5 2 2 4 4 4 1 1 5 5 2 5 2
## [2073] 2 2 2 2 4 2 2 2 2 2 2 2 1 5 5 2 2 2 4 2 5 5 5 5 2 2 2 4 4 5 5 5 5 2 1 5 2
## [2110] 2 2 4 4 4 2 1 5 5 2 2 4 4 2 2 5 5 2 2 4 5 5 2 2 4 5 2 2 2 2 2 2 2 2 4 5 2
## [2147] 2 2 4 1 5 2 2 2 2 4 4 4 4 5 5 4 4 4 2 4 2 5 5 5 2 2 2 1 5 4 2 2 2 2 4 4 4
## [2184] 5 5 2 5 2 2 2 2 2 1 5 5 2 2 2 2 2 2 2 4 5 2 2 2 2 2 2 2 4 5 5 5 5 2 2 2 2 2 5
## [2221] 2 2 2 2 4 4 5 5 2 2 4 4 2 4 2 5 5 5 2 2 2 1 1 5 2 2 2 4 4 4 4 4 5 5 5 2 4
## [2258] 2 2 5 5 2 2 2 2 2 2 5 5 2 4 4 4 4 3 5 5 2 2 2 4 2 2 5 2 2 2 2 2 2 4 4 2 5 2
## [2295] 4 2 4 4 4 2 2 5 5 2 2 2 2 2 2 2 2 5 2 2 5 2 2 4 1 5 5 2 2 4 2 4 4 5 2 4 4
## [2332] 4 5 5 2 2 2 5 4 4 4 5 5 2 2 2 2 4 4 4 5 5 5 2 2 4 2 4 2 5 5 2 2 2 4 5 2 5
## [2369] 2 2 2 2 4 2 2 4 2 4 4 4 4 4 5 5 5 2 4 4 4 5 5 2 4 4 2 5 5 2 2 2 2 2 4 2 5
## [2406] 2 5 4 2 2 1 5 5 2 2 2 2 2 2 2 5 2 4 2 4 5 5 2 4 2 4 5 5 2 5 2 2 2 4 4 2 5
## [2443] 2 5 2 2 2 2 2 5 5 5 4 4 5 5 2 4 2 2 4 2 5 2 2 2 5 2 2 2 2 4 2 5 5 4 2 5 5
## [2480] 2 2 2 4 1 1 5 2 5 2 4 4 5 5 2 2 5 4 5 5 2 4 4 2 4 4 4 5 5 2 2 2 4 2 2 2 4
## [2517] 5 5 2 2 4 2 2 5 5 5 2 2 5 4 2 2 2 5 5 2 2 2 2 4 4 1 1 5 5 2 4 4 1 1 5 5 2
## [2554] 2 2 2 5 2 5 2 2 4 2 2 2 5 2 2 5 2 2 2 2 2 5 5 2 2 2 2 2 2 4 5 5 5 2 2 2 4 1
## [2591] 5 5 2 2 2 2 4 5 5 2 5 5 2 2 2 2 5 5 2 2 4 4 2 1 5 2 2 2 4 4 4 2 1 5 5 5 2
## [2628] 2 1 5 5 5 2 2 2 4 1 4 4 4 4 2 4 2 4 2 4 1 5 2 5 2 2 4 2 1 5 5 2 5 2 2 5 5
## [2665] 2 2 2 2 4 2 5 5 2 4 2 4 4 4 1 5 2 2 2 2 2 1 4 5 5 2 2 2 2 4 2 1 5 5 2 2 2
## [2702] 4 4 4 5 5 5 2 2 4 5 2 4 2 2 2 4 4 2 4 4 5 5 5 2 2 4 2 1 1 5 2 5 4 4 4 4 1
## [2739] 5 5 2 4 4 2 4 5 2 2 2 2 2 2 2 4 2 2 5 5 5 4 2 2 2 2
##
## Within cluster sum of squares by cluster:
## [1]  737.6141 2624.4456  722.9391 1248.0870 1849.5824
##  (between_SS / total_SS =  56.7 %)
##
## Available components:
##
## [1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss"
## [6] "betweenss"    "size"         "iter"         "ifault"
```

Looks like this time, cluster 1 is all below average and cluster 5 is mostly above. Which cluster is Cam Mack in?

```
playeradvcluster <- data.frame(playersadvanced, advk5$cluster)
```

```
cmadv <- playeradvcluster %>% filter(Player == "Cam Mack")

cmadv
```

```
##     Player                Team Pos  PER   TS. PProd AST. WS.40 BPM
```

```
## 1 Cam Mack Nebraska Cornhuskers   G 15.9 0.481   382 36.4 0.081 3.9
##   advk5.cluster
## 1             5
```

Cluster 3 on my dataset. So in this season, we can say he's in a big group of players who are all above average on these advanced metrics.

Now who are his Big Ten peers?

```
playeradvcluster %>%
  filter(advk5.cluster == 3) %>%
  filter(Team %in% big10) %>%
  arrange(desc(PProd))
```

```
##  [1] Player        Team          Pos           PER           TS.
##  [6] PProd         AST.          WS.40         BPM           advk5.cluster
## <0 rows> (or 0-length row.names)
```

Sorting on Points Produced, Cam Mack is sixth out of the 35 guards in the Big Ten who land in Cluster 3. Seems advanced metrics take a little bit of the shine off of Cam. But then, so does leaving the program after one suspension-riddled season.

Chapter 32

Rtweet and Text Analysis

By Collin K. Berke, Ph.D.

One of the best rivalries in college volleyball was played on Saturday, November 2, 2019. The seventh-ranked Penn State Nittany Lions (16-3) took on the eigth-ranked Nebraska Cornhuskers (16-3). This match featured two of the best middle blockers in the country, Nebraska's Lauren Stivrins and Penn State's Kaitlyn Hord. Stivrins was ranked No. 1 in Big Ten hitting percentage, a .466 before the match up. Hord, close behind, had a .423 hitting percentage and was ranked towards the top as one of the league's top blockers.

Alongside being a competition between premier players, this match was set to be a battle between two of the winningest coaches in NCAA Women's Volleyball history. Russ Rose, head coach of the Nittany Lions, came into the match with a 1289-209 (.860) record, 17 Big Ten Conference Championships, and 7 NCAA National Championships. For the Nebraska Cornhuskers' head coach, John Cook came into the match with a 721-148 (.830) record, 9 Big 12 Conference Championships, 4 Big Ten Conference Championships, and 5 NCAA National Championships. Check out this article here to get a better understanding of the significance of this game and rivalry.

Being a contest between two storied programs, premier players, and two of the most winningest coaches in NCAA volleyball history, this match was poised to be one of the premier Big Ten matches of the 2019 season. If history was to serve as a guide, this match would easily go into five exciting, nail-biting sets.

To no surprise–it did. Nebraska came out victorious, 3 sets to 2, winning the fifth set 15 - 13. Although we have commentators, analysts, and reporters to tell us the story of the game, wouldn't it be interesting to tell the story from the fan's perspective? Can what they say allow us to take a pulse of how the fan base feels during the game? We can answer this question using Twitter tweet data, which we will access with the `rtweet` package.

Question - How do people feel during a game? Positive? Negative? Neutral?

This chapter will teach you how to extract, analyze, and visualize Twitter text data to tell a story about peoples sentiments toward any sport team, player, or event. Although Twitter is conventionally thought of as a social media platform, at a general level, it can be thought of as a corpus of textual data, which is generated by millions of users, talking about a wide array of topics over time.

During this chapter, we will access text data held within the body of tweets, which we will extract and import into R through the use of an API (application programming interface). This can seem like a pretty technical term, but all it really is is a portal to which data can be shared between computers and humans. NPR has an API 101 post on their site, which you can read to get a rough idea of what an API is and how they are used.

In fact, many news organizations provide APIs for people to access and use their data. For example, many news services like The New York Times, NPR, The Associated Press and social media platforms like Facebook have APIs that can be used to access content or varying types of data. Many of these just require you to: a). have a developer account; b) have the proper API keys; and c). use their API in accordance with their terms of service. Every API you come across should have documentation outlining its use.

Above was a pretty hand-wavy explanation of APIs. Indeed, APIs have many different uses beyond just extracting data, but such a discussion is beyond the scope of this chapter. Nevertheless, APIs can be a powerful, useful tool to access data not normally available on web pages or other statistical reporting services.

32.1 Prerequisites

You will need to have a Twitter account to access and extract data. If needed, you can sign up for an account here.

32.1.1 Tools for text analysis

This chapter will also require you to load and acquaint yourself with functions in four packages, `rtweet`, `lubridate`, `stringr`, and `tidytext`. You may have used some of functions in other portions of this class. Others may be new to you.

- `rtweet` is a R package used to access Twitter data via the Twitter API.
- `lubridate` is a package that makes working with dates and times a bit easier.
- `stringr` is a package that provides several functions to make working with string data a little easier.

- `tidytext` is a package used to tidy, analyze, and visualize textual analyses. We will use this package to calculate tweet sentiments (e.g., positive and negative feelings).

This chapter will also use other packages you have gained familiarity with throughout the class: `dplyr` and `ggplot2`. To install these packages and load them for use in our analysis session, run the following code:

```
install.packages("rtweet") # installs the rtweet package

install.packages("tidytext") # installs the tidytext package

install.packages("tidyverse") # A collection of packages, includes the stringr packages

install.packages("lubridate") # Provides functions to make working with dates/times easier
```

```
# Load the packages to be used in your analysis session

library(rtweet)

library(tidytext)

library(tidyverse)

library(lubridate)

library(ggrepel)
```

32.1.2 Working with string data

String data is just basically letters, words, symbols, and even emojis. Take for example the following tweet:

```
knitr::include_graphics(rep("images/volleyballTweet.png"))
```

Everything contained in the message portion of the tweet is string data, even the emojis. When it comes to emojis, most have a special textual code that is

rendered by a browser or device that gets displayed as an image. For example, the ear of corn emoji is actually written as `:corn:`, but it gets rendered as an image when we view the tweet on our computers/devices. We can extract, analyze, and visualize this string data to tell a wide range of stories from users' tweets. Our goal being to show sentiment over the length of a Husker volleyball Match and football game.

32.2 Verifying your account to access Twitter data

Before you can access Twitter data, you will need to verify your account. The `rtweet` package makes this really easy to do. You will first need to run one of the package's functions for it to walk you through the authentication process. To do this, let's just search for the most recent 8000 (non-retweeted) tweets containing the **#huskers** and **#GBR** hashtags.

Before you run the following code chunk, though, be aware a few things will take place. First, a browser window will open up asking you to verify that `rtweet` is allowed to access Twitter data via the API on behalf of your account. Accept this request and enter your credentials if you are asked to. Once you do this, you should get a message in your browser stating you have successfully authenticated the `rtweet` package. The data will then begin to download. The amount of time needed to import this data will depend on how many tweets the hashtag(s) are associated with. More tweets generally means longer import times.

```
huskers <- search_tweets(
  "#huskers", n = 8000, include_rts = FALSE
)

gbr <- search_tweets(
  "#GBR", n = 8000, include_rts = FALSE
)
```

Important Note: Depending on when you run the above code chunk, the API will return different data then the data used for the examples later in this chapter. This is due to the query rate cap Twitter places on it's API. Twitter's API caps queries to 18,000 of the most recent tweets during the past couple of days. This cap resets every 15 minutes. The `rtweet` package does has functionality to pull data once your query limit resets. However, if you're looking to pull tweets for a very popular event (e.g., The Super Bowl), you may want to consider other options to extract this type of data. This is also important to understand because if you are looking to pull tweets for a specific event, you will need to make sure you are pulling this data within a reasonable time during or after the event. If you don't, these rate limits might not allow you access the data you need to do your analysis.

The data we will use later for the examples in the chapter can be found here and here. The first data set are tweets that use the #huskers hashtag. The second has data of tweets that use the #gbr hashtag. You will need to download both data sets, put them in the right directory, and import both for the below examples to work correctly. The code to import this data will look something like this:

```
huskerTweets <- read_csv("data/huskerTweets.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_character(),
##   created_at = col_datetime(format = ""),
##   display_text_width = col_double(),
##   is_quote = col_logical(),
##   is_retweet = col_logical(),
##   favorite_count = col_double(),
##   retweet_count = col_double(),
##   quote_count = col_logical(),
##   reply_count = col_logical(),
##   symbols = col_logical(),
##   ext_media_type = col_logical(),
##   quoted_created_at = col_datetime(format = ""),
##   quoted_favorite_count = col_double(),
##   quoted_retweet_count = col_double(),
##   quoted_followers_count = col_double(),
##   quoted_friends_count = col_double(),
##   quoted_statuses_count = col_double(),
##   quoted_verified = col_logical(),
##   retweet_status_id = col_logical(),
##   retweet_text = col_logical(),
##   retweet_created_at = col_logical()
##   # ... with 21 more columns
## )
```

```
## See spec(...) for full column specifications.
```

```
gbrTweets <- read_csv("data/gbrTweets.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_character(),
##   created_at = col_datetime(format = ""),
##   display_text_width = col_double(),
##   is_quote = col_logical(),
##   is_retweet = col_logical(),
##   favorite_count = col_double(),
##   retweet_count = col_double(),
```

```
##   quote_count = col_logical(),
##   reply_count = col_logical(),
##   symbols = col_logical(),
##   ext_media_type = col_logical(),
##   quoted_created_at = col_datetime(format = ""),
##   quoted_favorite_count = col_double(),
##   quoted_retweet_count = col_double(),
##   quoted_followers_count = col_double(),
##   quoted_friends_count = col_double(),
##   quoted_statuses_count = col_double(),
##   quoted_verified = col_logical(),
##   retweet_status_id = col_logical(),
##   retweet_text = col_logical(),
##   retweet_created_at = col_logical()
##   # ... with 21 more columns
## )
## See spec(...) for full column specifications.
```

```
## Warning: 1 parsing failure.
##  row     col           expected actual                 file
## 4365 symbols 1/0/T/F/TRUE/FALSE    GBR 'data/gbrTweets.csv'
```

This brings up a good point about saving any data you import from Twitter's API. **Always save your data.** Remember those rate limits? If you don't save your data and too many days pass, you will not be able to access that data again. To do this, you can use the `write_as_csv()` function from the `rtweet` package to save a `.csv` file of your data. The code to do this will look something like this:

```
write_as_csv(huskerTweets, "data/huskerTweets.csv")
```

Be aware that this function will overwrite data. If you make changes to your `huskerTweets` object and then run the `write_as_csv()` function again, it will overwrite your saved file with the modifications you made to your object. The lesson then is to always save an extra copy of your data in a separate directory, just in case you do accidentally make a mistake in overwriting your data.

32.3 The data used here

To provide a little context, I pulled the data in this chapter on Sunday, November 3, 2019. This was the day after Nebraska Football lost to Purdue, and Nebraska Volleyball won against Penn State. You can follow the steps above to download this data for the following examples.

To make it easier to work with, I am going to combine these two data sets into one using the `bind_rows()` function from `dplyr`. There is a slight problem

though, some people may have had a tweet that contained both the **#huskers** and **#GBR** hashtags in their tweet. So if we combine these two data sets, there might be duplicate data. To dedupe the data, we can apply a `distinct(text, .keep_all = TRUE)` to remove any duplicates. The `.keep_all = TRUE` argument just tells R to keep all columns in the data frame after our data has been deduped.

```
tweet_data <- bind_rows(huskerTweets, gbrTweets) %>%
  distinct(text, .keep_all = TRUE)
```

32.4 Data Exploration

Let's explore the data a bit. Run a `glimpse(tweet_data)` to get a view of what data was returned from twitter. My query on November 3rd, 2019 returned 11,523 non-retweeted tweets from 3,917 accounts using the `#huskers` and/or the `#GBR` hashtag within the tweet's body (again if you ran the code above, your data will be different) .

It's important to remember these tweets can come from accounts that are people, organizations, and even bots. So when drawing conclusions from this data, make sure to keep in mind that these tweets may not represent the sentiment of just one person. Additionally, it is important to remember that not all fans of a sports team are on or use Twitter, so it surely is not a valid representation of all fan sentiment. Indeed, you could also have fans of other teams using your hashtags.

```
glimpse(tweet_data)
```

```
## Rows: 11,523
## Columns: 90
## $ user_id                 <chr> "x17636179", "x17636179", "x17636179", "x15...
## $ status_id               <chr> "x1191100304940396544", "x11908526509059440...
## $ created_at              <dttm> 2019-11-03 21:10:16, 2019-11-03 04:46:11, ...
## $ screen_name             <chr> "SeanKeeler", "SeanKeeler", "SeanKeeler", "...
## $ text                    <chr> "ICYMI, #CSURams fans, a recap of @denverpo...
## $ source                  <chr> "Twitter Web App", "Twitter for iPhone", "T...
## $ display_text_width      <dbl> 269, 188, 182, 122, 135, 189, 172, 135, 94,...
## $ reply_to_status_id      <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ reply_to_user_id        <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ reply_to_screen_name    <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ is_quote                <lgl> FALSE, TRUE, FALSE, FALSE, FALSE, FALSE, FA...
## $ is_retweet              <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, F...
## $ favorite_count          <dbl> 0, 0, 1, 116, 19, 2, 40, 23, 149, 5, 31, 15...
## $ retweet_count           <dbl> 0, 0, 0, 6, 1, 0, 1, 2, 5, 1, 3, 1, 0, 0, 0...
## $ quote_count             <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ reply_count             <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
```

```
## $ hashtags                <chr> "CSURams Huskers ProudToBe AtThePeak CSU", ...
## $ symbols                 <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ urls_url                <chr> "tinyurl.com/y2b3kog9 tinyurl.com/yy7ntps4"...
## $ urls_t.co               <chr> "https://t.co/1FlAnRRgEq https://t.co/t8j1L...
## $ urls_expanded_url       <chr> "https://tinyurl.com/y2b3kog9 https://tinyu...
## $ media_url               <chr> NA, NA, NA, "http://pbs.twimg.com/ext_tw_vi...
## $ media_t.co              <chr> NA, NA, NA, "https://t.co/FSCP827hg0", "htt...
## $ media_expanded_url      <chr> NA, NA, NA, "https://twitter.com/HuskerSpor...
## $ media_type              <chr> NA, NA, NA, "photo", "photo", "photo", "pho...
## $ ext_media_url           <chr> NA, NA, NA, "http://pbs.twimg.com/ext_tw_vi...
## $ ext_media_t.co          <chr> NA, NA, NA, "https://t.co/FSCP827hg0", "htt...
## $ ext_media_expanded_url  <chr> NA, NA, NA, "https://twitter.com/HuskerSpor...
## $ ext_media_type          <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ mentions_user_id        <chr> "x8216772", "x8216772", "x24725032", "x1210...
## $ mentions_screen_name    <chr> "denverpost", "denverpost", "DPostSports", ...
## $ lang                    <chr> "en", "en", "en", "en", "en", "en", "en", "...
## $ quoted_status_id        <chr> NA, "x1190803840213209088", NA, NA, NA, NA,...
## $ quoted_text             <chr> NA, "From Nebraska to Fort Collins, how CSU...
## $ quoted_created_at       <dttm> NA, 2019-11-03 01:32:14, NA, NA, NA, NA, N...
## $ quoted_source           <chr> NA, "TweetDeck", NA, NA, NA, NA, NA, NA, NA...
## $ quoted_favorite_count   <dbl> NA, 5, NA, NA, NA, NA, NA, NA, NA, 194,...
## $ quoted_retweet_count    <dbl> NA, 1, NA, NA, NA, NA, NA, NA, NA, 16, ...
## $ quoted_user_id          <chr> NA, "x24725032", NA, NA, NA, NA, NA, NA, NA...
## $ quoted_screen_name      <chr> NA, "DPostSports", NA, NA, NA, NA, NA, NA, ...
## $ quoted_name             <chr> NA, "Denver Post Sports", NA, NA, NA, NA, N...
## $ quoted_followers_count  <dbl> NA, 34841, NA, NA, NA, NA, NA, NA, NA, NA, ...
## $ quoted_friends_count    <dbl> NA, 395, NA, NA, NA, NA, NA, NA, NA, NA, 60...
## $ quoted_statuses_count   <dbl> NA, 113697, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ quoted_location         <chr> NA, "Denver, Colorado", NA, NA, NA, NA, NA,...
## $ quoted_description      <chr> NA, "Sports news & analysis from @denverpos...
## $ quoted_verified         <lgl> NA, TRUE, NA, NA, NA, NA, NA, NA, NA, NA, F...
## $ retweet_status_id       <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ retweet_text            <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ retweet_created_at      <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ retweet_source          <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ retweet_favorite_count  <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ retweet_retweet_count   <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ retweet_user_id         <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ retweet_screen_name     <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ retweet_name            <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ retweet_followers_count <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ retweet_friends_count   <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ retweet_statuses_count  <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ retweet_location        <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ retweet_description     <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ retweet_verified        <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
```

```
## $ place_url               <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ place_name              <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ place_full_name         <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ place_type              <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ country                 <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ country_code            <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ geo_coords              <chr> "NA NA", "NA NA", "NA NA", "NA NA", "NA NA"...
## $ coords_coords           <chr> "NA NA", "NA NA", "NA NA", "NA NA", "NA NA"...
## $ bbox_coords             <chr> "NA NA NA NA NA NA NA NA", "NA NA NA NA NA ...
## $ status_url              <chr> "https://twitter.com/SeanKeeler/status/1191...
## $ name                    <chr> "Sean Keeler", "Sean Keeler", "Sean Keeler"...
## $ location                <chr> "Denver, CO", "Denver, CO", "Denver, CO", "...
## $ description             <chr> "@DenverPost staffer, dad, husband, drummer...
## $ url                     <chr> "https://t.co/z0eFbv9eaz", "https://t.co/z0...
## $ protected               <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, F...
## $ followers_count         <dbl> 5245, 5245, 5245, 30451, 30451, 30451, 3045...
## $ friends_count           <dbl> 1619, 1619, 1619, 701, 701, 701, 701, 701, ...
## $ listed_count            <dbl> 296, 296, 296, 247, 247, 247, 247, 247, 247...
## $ statuses_count          <dbl> 28124, 28124, 28124, 12801, 12801, 12801, 1...
## $ favourites_count        <dbl> 4498, 4498, 4498, 6033, 6033, 6033, 6033, 6...
## $ account_created_at      <dttm> 2008-11-25 23:53:13, 2008-11-25 23:53:13, ...
## $ verified                <lgl> FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, TRUE...
## $ profile_url             <chr> "https://t.co/z0eFbv9eaz", "https://t.co/z0...
## $ profile_expanded_url    <chr> "http://www.seankeeler.tumblr.com", "http:/...
## $ account_lang            <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ profile_banner_url      <chr> "https://pbs.twimg.com/profile_banners/1763...
## $ profile_background_url  <chr> "http://abs.twimg.com/images/themes/theme10...
## $ profile_image_url       <chr> "http://pbs.twimg.com/profile_images/118130...
```

As you can see, `glimpse()` returns a lot of columns that are not really relevant to our analysis. Let's apply a `select()` function to only retain the data relevant to our analysis, .

```
tweet_data <- tweet_data %>%
  select(user_id, status_id, created_at, screen_name, text, display_text_width,
         favorite_count, retweet_count, hashtags, description, followers_count)
```

32.5 Cleaning data for analysis

If you examine the data set, you will see this data needs some wrangling. First, we need to fix the `created_at` variable. Right now it is represented in Greenwich Mean Time (GMT), but we need it to be in Central Standard Time (CST). We do this so we can make sense of when during the game things happened. Second, the data is outside of the time frame we are interested in examining, so we need to filter the data to be windowed during the time of the game. We will filter the data by date and time, examining tweets a little before and after the game.

32.5.1 Fixing the date and focusing only on game tweets

We will use the `with_tz()` on our `created_at` variable within our `mutate()` function to transform the `created_at` column into Central Standard Time (CST). We do this by setting the `tzone` argument to `"America/Chicago"`. Once out time is adjusted, we need group tweets within a specific bin of time. For this example I have decided to bin tweets to the nearest 5 minute mark. We can do this by using the `round_time()` function provided to us by the `lubridate` package.

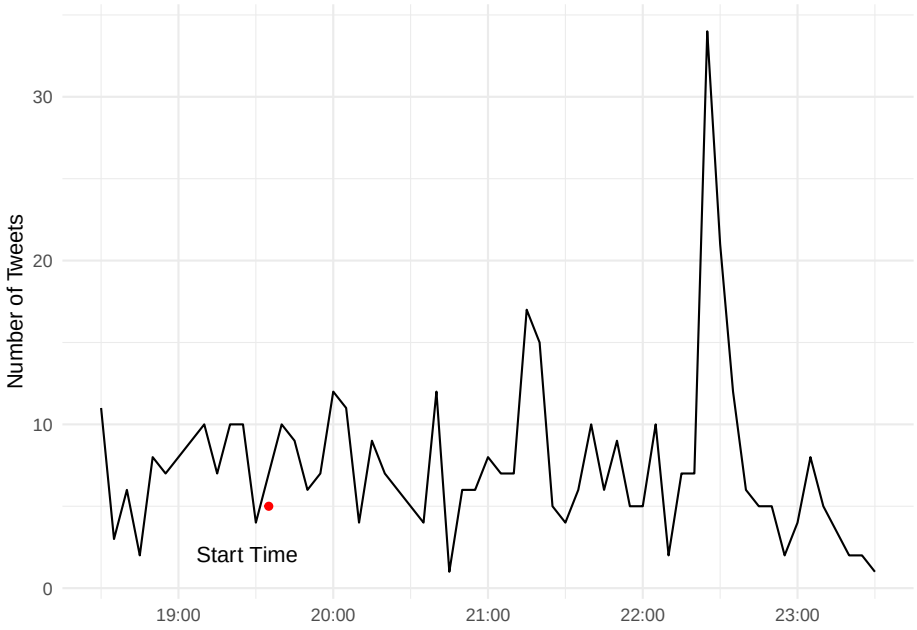
Then, since we are only interested in tweets during the game, we can apply a `dplyr` `filter()` function to window our data set to tweets being posted around the start and end of the game.

```
volleyball_tweets <- tweet_data %>%
  mutate(created_at = with_tz(created_at, tzone = "America/Chicago"),
         created_at = round_time(created_at, "5 mins", tz = "America/Chicago")) %>%
  filter(created_at >= "2019-11-02 18:30:00" & created_at <= "2019-11-02 23:30:00")
```

32.6 Number of tweets throughout the game

One question we might have pertains to the number of tweets that occur during the course of the match. To do this, all we need to do is `group_by()` our tweets by our `created_at` variable, and then use the `count()` function to count the number of tweets within each five minute bin. We then use `ggplot` to plot a line chart where `created_at` is placed on the x-axis and `n`, number of tweets, is placed on the y-axis.

```
start_time <- tibble(time = as_datetime("2019-11-02 19:35:00", tz = "America/Chicago")

volleyball_time <- volleyball_tweets %>%
  group_by(created_at) %>%
  count()

ggplot() +
  geom_line(data = volleyball_time, aes(x = created_at, y = n)) +
  geom_point(data = start_time, aes(x = time, y = 5), color = "red") +
  geom_text_repel(data = start_time, aes(x = time, y = 3, label = label), nudge_x = -2
  labs(y = "Number of Tweets",
       x = "Central Standard Time (CST)") +
  theme_minimal() +
  theme(axis.title.x = element_blank())
```



There you have it. A trend line plotting tweet volume throughout the course of the event. Do you see any areas where the match might have had a significant number of tweets?

32.7 Tidying the text data for analysis, applying the sentiment scores

Okay, that's cool–but what we really want to know is what are peoples' sentiments throughout the game? Did they feel positive or negative throughout the event? Were there times that were more positive or negative? To achieve this, we are going to use the `tidytext` package to tidy up our text data and apply a sentiment score to each word held within each tweet. Let's break this down step-by-step.

First, we need to get the dictionary that contains the sentiment scoring for thousands of words used in the English language. `afinn <- get_sentiments("afinn")` does just that for us. The development of these sentiment dictionaries is beyond this chapter. However, most of these dictionaries are crowd sourced by having people provide self-responses on how positive or negative a word is to them. For now, just understand the `affin` variable contains many words that have been rated for how positive or negative a word is on a scale that ranges from -5 to 5. -5 being the most negative, and 5 being the most positive. If you want to learn more about this dictionary or others, you can read more about them here.

Second, now that we have our dictionary imported, we need to clean up our tweets data set so we can apply sentiment scores to each word used within each tweet. There's one problem, though. Each row in our data set is a complete tweet. For us to apply a sentiment score for each word, each word needs to get its own row. This is where the `unnest_tokens()` function from the `tidytext` package comes into play. We use this function to create a data set that will create a new column called word, which will place every word from every tweet in our data set on its own row, which it knows which text data to this because we set the the second argument to the column name that holds our text data. In this case, we give it the `text` column. Once you run this code, if you look at the `volleyball_tweets_tidy` object, you should now have a data set where every row has its own word which was done for every tweet. This data frame should now be a super long data frame.

Lastly, the English language has many words that really don't mean anything in regards to sentiment. Take for example the word 'the'. This article really doesn't represent a positive or negative sentiment. Thus, these types of words need to be taken out of our data set to enhance improve the accuracy of our analysis. To do this, we will apply the `anti_join(stop_words)` to our `dplyr` chain. All this does is get rid of the stop words in our data set that really don't contribute to the sentiment scores we are eventually going to calculate.

If you get an error on the next bit of code, you'll likely need to install the textdata package on the console with `install.packages("textdata")`. Next, if this next block of code hangs, it's because in the console you're being asked if you want to download some data. You do indeed want to do that, so type 1 and hit enter.

```
afinn <- get_sentiments("afinn")

volleyball_tweets_tidy <- volleyball_tweets %>%
  unnest_tokens(word, text) %>%
  anti_join(stop_words)
```

```
## Joining, by = "word"
```

32.7.0.1 Fan sentiment over the game

Now that we have a tidied textual data set, all we need to do is apply our sentiment scores to these words using the `inner_join()`, then `group_by()` our `created_at` variable, and calculate the mean sentiment for each five minute interval. At this point, we will use `ggplot` to plot sentiment of the tweets over time. We will do this by plotting the `created_at` variable on the x-axis and the newly calculated `sentiment` variable on the y-axis. The rest is just adding annotations and styling, which we are already familiar with.

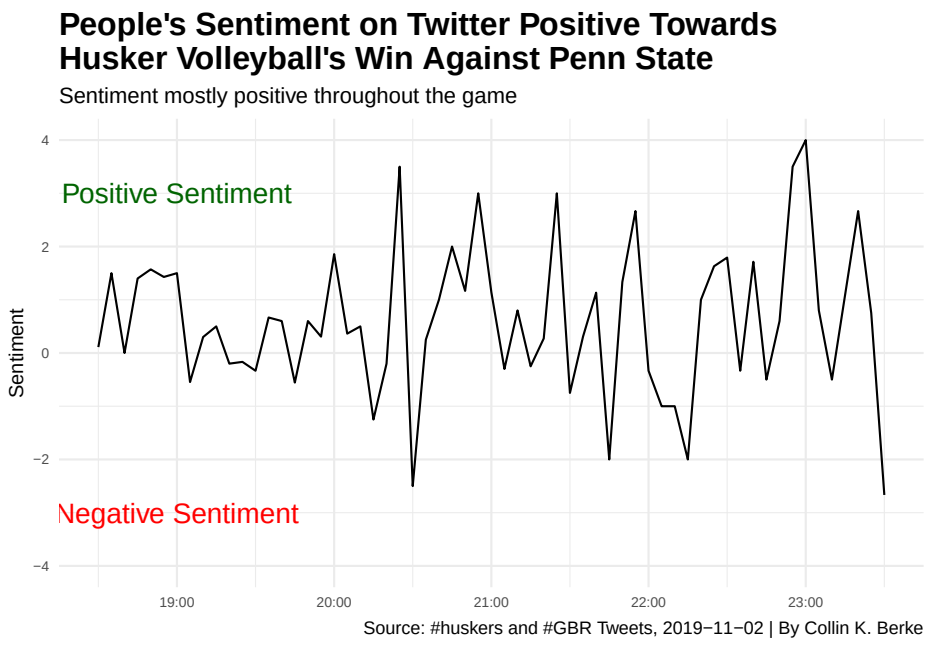
```
volleyball_tweets_sentiment <- volleyball_tweets_tidy %>%
  inner_join(afinn) %>%
```

```
  group_by(created_at) %>%
  summarise(sentiment = mean(value))
```

```
## Joining, by = "word"
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
ggplot() +
  geom_line(data = volleyball_tweets_sentiment, aes(x = created_at, y = sentiment)) +
  geom_text(aes(x = as_datetime("2019-11-02 19:00:00", tz = "America/Chicago"), y = 3), color = '
  geom_text(aes(x = as_datetime("2019-11-02 19:00:00", tz = "America/Chicago"), y = -3), color =
  labs(title = "People's Sentiment on Twitter Positive Towards\nHusker Volleyball's Win Against P
       subtitle = "Sentiment mostly positive throughout the game",
       caption = "Source: #huskers and #GBR Tweets, 2019-11-02 | By Collin K. Berke",
       y = "Sentiment",
       x = "Central Standard Time (CST)") +
  scale_y_continuous(limits = c(-4, 4)) +
  theme_minimal() +
  theme(axis.title.x = element_blank(),
        plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 10),
    axis.text = element_text(size = 7)
        )
```



When looking at this trend line, you can see that during the volleyball game, tweets using the #husker and #gbr hashtags had some wide variation in sen-

timent. Overall it seems that tweets during the volleyball match were mostly positive, where at times it dipped negative. Why might this be the case? Well, unfortunately, even though this was a big game for Husker volleyball, not many people were tweeting during the match (take a look at the number of tweets chart above). So if there was one word used in a tweet that was ranked as very negative in sentiment, it would have easily drove our average sentiment into the negative region quickly.

Also, we need to consider that this match took place after the Huskers loss to Purdue, which we will examine in the next example. This is important to know because people during the volleyball match may have also been tweeting about how poorly the football game went earlier in the day. Thus, low tweet volume mixed with the potential for tweets referencing something other than the match at hand may have had some influence on the sentiment scores.

There's also one last thing to keep in mind when you draw conclusions from this type of text data. Language is complex–it can have multiple meanings, which is highly influenced by context. Take for example the word 'destroy', like its use in the following statement: "This team is going to destroy the defense today." Although we clearly can see this is a positive statement, when a computer applies sentiment scores, the context of the statement is stripped away, and destroy will be scored as negative sentiment. In short, computers are not smart enough to include context when they calculate sentiment, yet. So, keep this limitation in mind when you draw conclusions from your sentiment analyses using text data.

32.7.1 Example 2 - Nebraska's loss to Purdue, what were fan's sentiments towards this loss?

The Nebraska Cornhuskers–a 3-point favorite going into West Lafayette, IN–squared off with the Purdue Boilermakers on November 2, 2019. Purdue was 2-6 on the season. Nebraska, with a 4-4 record coming off of a 38-31 home loss to Indiana, had many fans hoping Scott Frost could lead his team to a must needed win. Especially given the expectation was the Cornhuskers would go 6-6 on the season, and the team still had to play Wisconsin (6-2), Maryland (3-6), and Iowa (6-2) to get to those needed six wins to become bowl eligible. So, how did people particularly take this loss? Let's use our Twitter data to get an answer.

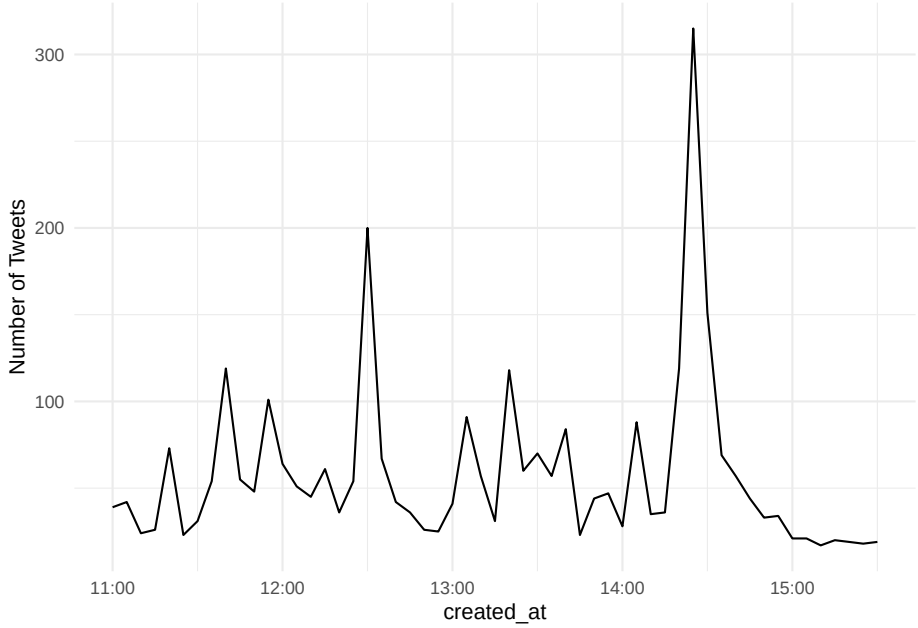
Again, we need to fix the time zone with the `with_tz()` function so the data is represented in Central Standard Time (CST). Then we apply our `filter()` command to window our data to when the game was taking place.

```
football_tweets <- tweet_data %>%
  mutate(created_at = with_tz(created_at, tzone = "America/Chicago"),
         created_at = round_time(created_at, "5 mins", tz = "America/Chicago")) %>%
  filter(created_at >= "2019-11-02 11:00:00" & created_at <= "2019-11-02 15:30:00")
```

Now, let's just get a sense of the number of tweets that occurred at certain points

in the game. We again need to do some data wrangling with `group_by()`, and then we use the `count()` function to add up all the tweets during each five minute interval. Once the data is wrangled, we can use our `ggplot` code to visualize tweet volume throughout the game.

```
football_time <- football_tweets %>%
  group_by(created_at) %>%
  count()

ggplot() +
  geom_line(data = football_time, aes(x = created_at, y = n)) +
  theme_minimal() +
  labs(y = "Number of Tweets")
```



Looking at this plot, we can see the tweet volume is a lot higher than that of the volleyball match. In fact, it looks like towards the end of the game there was a five minute interval where ~80 or so tweets occurred. Given the outcome of the game, I assume people were not real happy during this spike in activity. Well we have the tools to answer this question.

As before, let's pull in our sentiment library with the `get_sentiments()` function. Then lets tidy up our tweet data using the `unnest_tokens()` and `anti_join(stop_words)`. Remember this step just places every word within a tweet on its own row and filters out any words that don't have any real meaning to the calculation of sentiment (i.e., and, the, a, etc.).

```
afinn <- get_sentiments("afinn")

football_tweets_tidy <- football_tweets %>%
  unnest_tokens(word, text) %>%
  anti_join(stop_words)
```

```
## Joining, by = "word"
```

We now have our clean textual data, let's apply sentiment scores for each word using the `inner_join(affin)` function, `group_by(created_at)` to create a group for each five minute interval, and then use `summarise()` to calculate then mean sentiment for each time period.

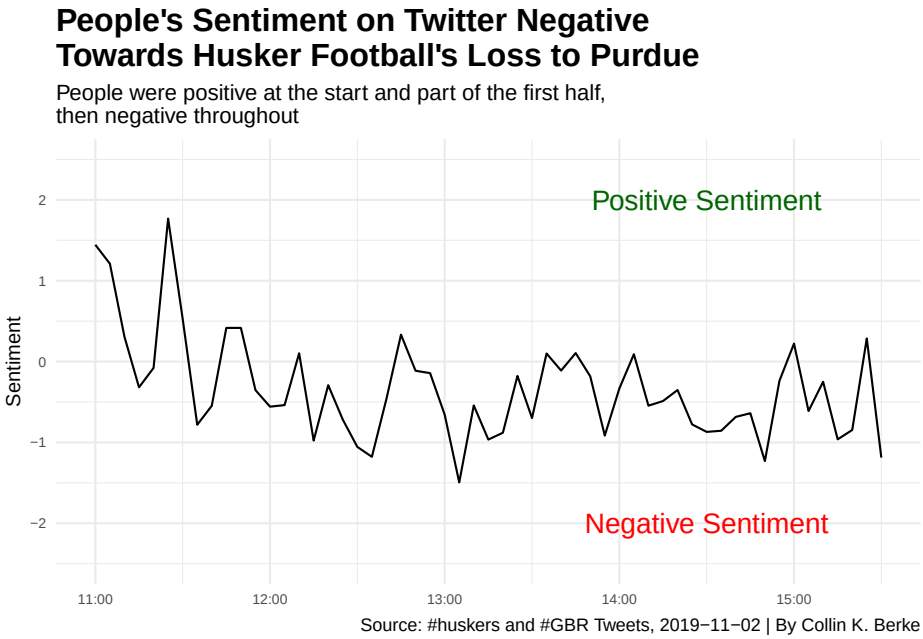
You can then use the `ggplot` code to plot sentiment over time, introduce annotations to highlight specific aspects within our plot, and then apply styling to the plot to move it closer to publication readiness. Now for the big reveal, how did people take the loss to a 2-6 Purdue? Run the code and find out.

```
football_tweets_sentiment <- football_tweets_tidy %>%
  inner_join(afinn) %>%
  group_by(created_at) %>%
  summarise(sentiment = mean(value)) %>%
  arrange(sentiment)
```

```
## Joining, by = "word"
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
ggplot() +
  geom_line(data = football_tweets_sentiment, aes(x = created_at, y = sentiment)) +
  geom_text(aes(x = as_datetime("2019-11-02 14:30:00", tz = "America/Chicago"), y = 2)
  geom_text(aes(x = as_datetime("2019-11-02 14:30:00", tz = "America/Chicago"), y = -2
  scale_y_continuous(limits = c(-2.5, 2.5)) +
  labs(title = "People's Sentiment on Twitter Negative\nTowards Husker Football's Loss
       subtitle = "People were positive at the start and part of the first half,\nthen
       caption = "Source: #huskers and #GBR Tweets, 2019-11-02 | By Collin K. Berke",
       y = "Sentiment",
       x = "Central Standard Time (CST)") +
  theme_minimal() +
  theme(axis.title.x = element_blank(),
        plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 10),
    axis.text = element_text(size = 7)
  )
```



As you can see, it started out pretty positive. Then, it started to go negative throughout the first half. However, there was a bump, which was around the time D-Lineman, Darrion Daniels almost scored a pick six. Around halftime, we can see a little bit of a bump towards positive sentiment. This was probably most likely due to people cheering on the Huskers to come out strong after the half. As the second half progressed, you can see things turned for the worst again, and sentiment became negative up until the end of the game, most likely because people realized they were going to get another L on the schedule. You can relive all this excitement again by catching the game recap here.

Chapter 33

Arranging multiple plots together

Sometimes you have two or three (or more) charts that are really just one chart that you need to merge them together. It would be nice to be able to arrange them programmatically and not have to mess with it in 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)
```

What follows is just stuff for me to set up a couple of bar charts. You can run it – it'll work on your machine without changing a thing – but what I'm doing here isn't important. The stuff you need to do is below.

```
attendance <- read_csv("https://raw.githubusercontent.com/mattwaite/sportsdatabook/Master/data/at
```

```
## Parsed with column specification:
## cols(
##   Institution = col_character(),
##   Conference = col_character(),
##   `2013` = col_double(),
##   `2014` = col_double(),
##   `2015` = col_double(),
##   `2016` = col_double(),
##   `2017` = col_double(),
##   `2018` = col_double()
```

```
## )
```

Making a quick percent change.

```
attendance <- attendance %>% mutate(change = ((`2018`-`2017`)/`2017`)*100)
```

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

```
top10 <- attendance %>% top_n(10, wt=change)
bottom10 <- attendance %>% top_n(10, wt=-change)
```

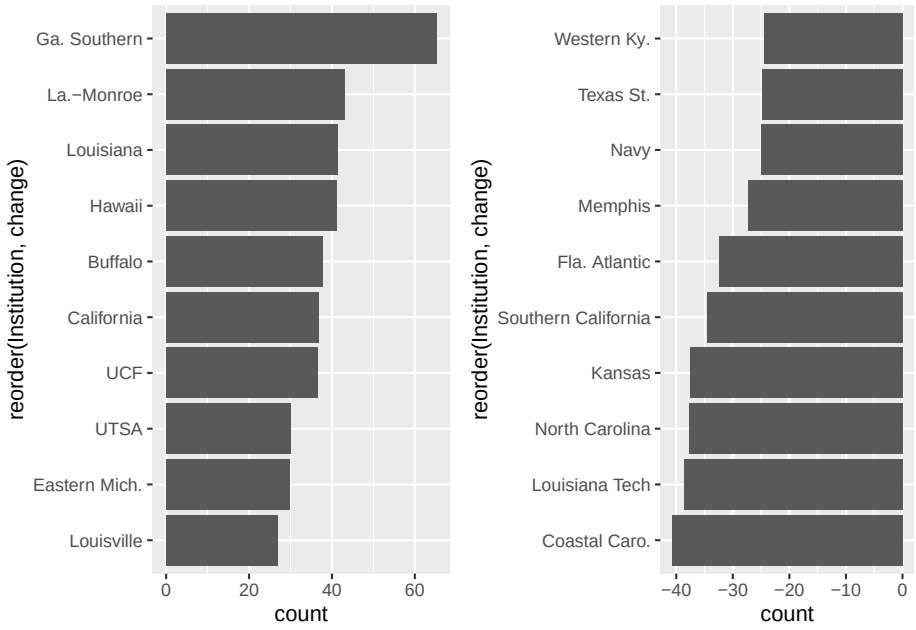
Okay, now to do this I need to save my plots to an object. We do this the same way we save things to a dataframe – with the arrow. We'll make two identical bar charts, one with the top 10 and one with the bottom 10.

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

```
bar2 <- ggplot() + geom_bar(data=bottom10, aes(x=reorder(Institution, change), weight=
```

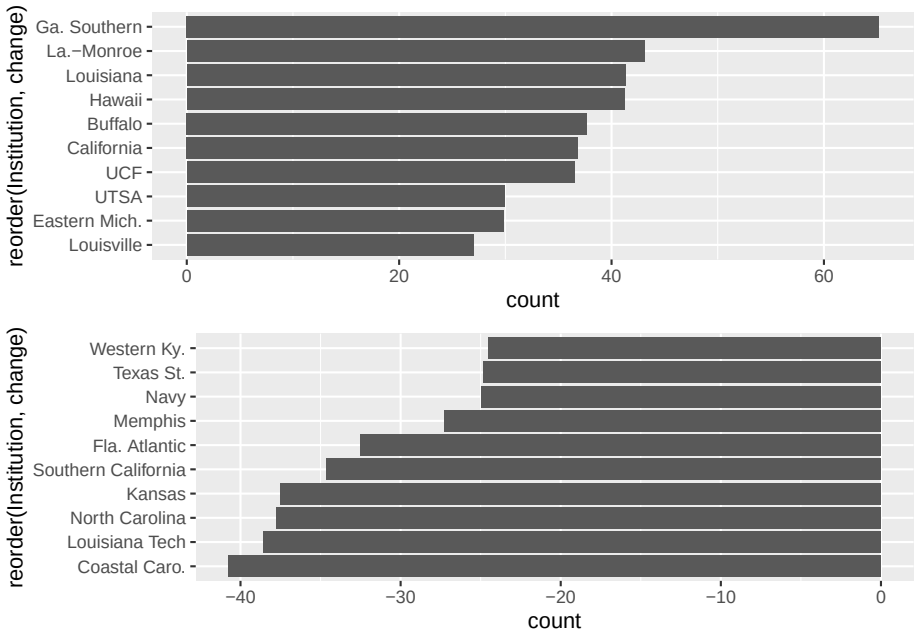
With cowplot, we can use a function called `plot_grid` to arrange the charts:

```
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.

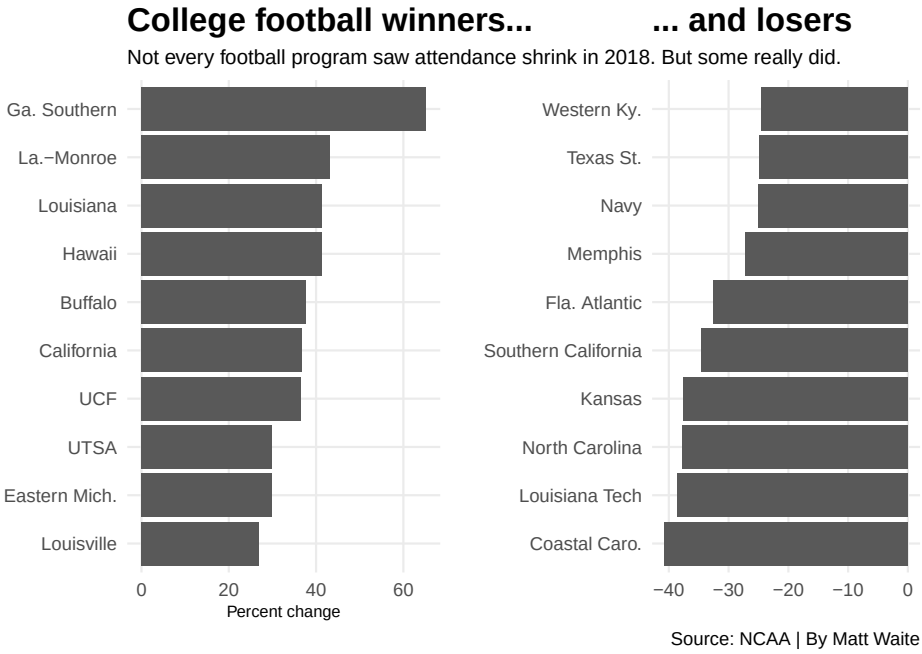
```
bar1 <- ggplot() + geom_bar(data=top10, aes(x=reorder(Institution, change), weight=change)) + coo
  theme(
    plot.title = element_text(size = 16, face = "bold"),
    axis.title = element_text(size = 8),
    plot.subtitle = element_text(size=10),
    panel.grid.minor = element_blank()
    )
```

```
bar2 <- ggplot() + geom_bar(data=bottom10, aes(x=reorder(Institution, change), weight=change)) +
  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()
    )
```

Saving a cowplot plot_grid is the same as anything else we did in the class:

```
plot_grid(bar1, bar2) + ggsave("test.png")
```

```
## Saving 6.5 x 4.5 in image
```



Chapter 34

Encircling points on a scatterplot

One thing we've talked about all semester is drawing attention to the thing you want to draw attention to. We've used color and labels to do that so far. Let's add another layer to it – a shape around the points you want to highlight.

Remember: The point of all of this is to draw the eye to what you are trying to show your reader. You want people to see the story you are trying to tell.

It's not hard to draw a shape in ggplot – it is a challenge to put it in the right place. But, there is a library to the rescue that makes this super easy – `ggalt`.

Install it in the console with `install.packages("ggalt")`

There's a bunch of things that `ggalt` does, but one of the most useful for us is the function `encircle`. Let's dive in.

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

Let's say we want to highlight the top scorers in college basketball. So let's load up our player data and while we're at it, let's filter out anyone who hasn't played.

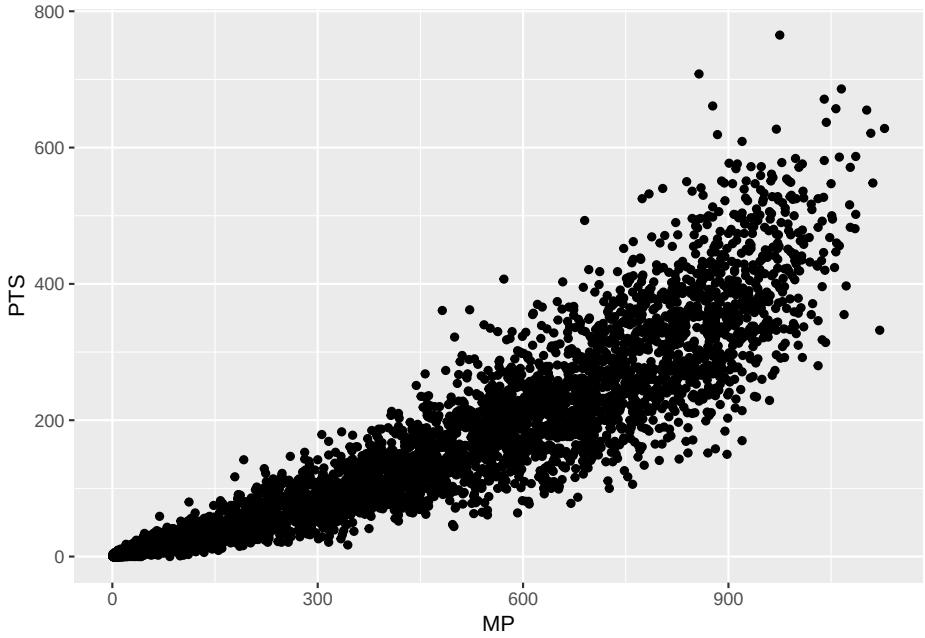
```
players <- read_csv("data/players20.csv") %>% filter(MP > 0)
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Team = col_character(),
##   Player = col_character(),
##   Class = col_character(),
```

```
##   Pos = col_character(),
##   Height = col_character(),
##   Hometown = col_character(),
##   `High School` = col_character(),
##   Summary = col_character()
## )

## See spec(...) for full column specifications.
```

We've done this before, but let's make a standard scatterplot of minutes and points.

```
ggplot() + geom_point(data=players, aes(x=MP, y=PTS))
```



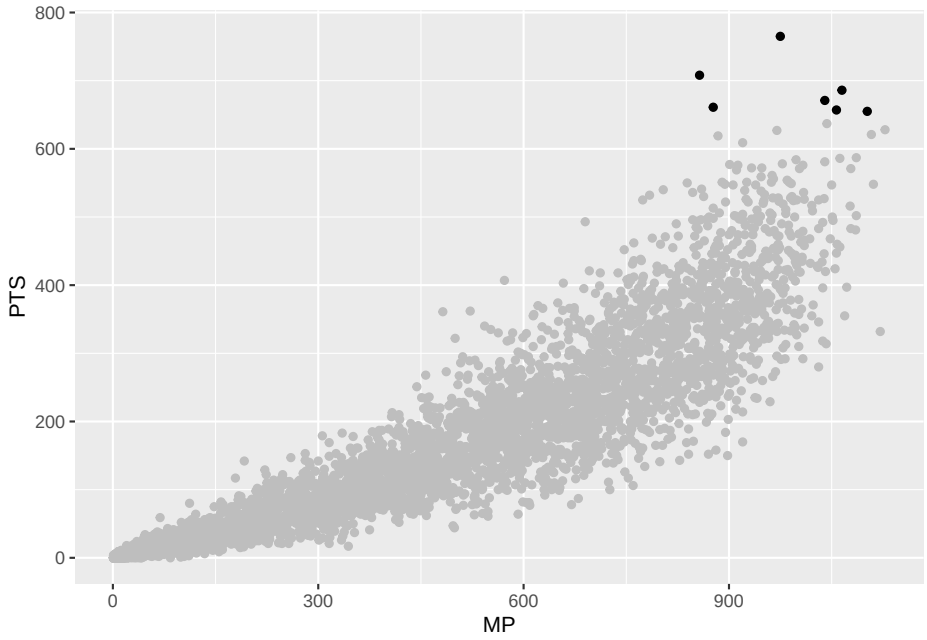
So we can see right away that there are some dots at the very top that we'd want to highlight. Who are these scoring machines?

Like we have done in the past, let's make a dataframe of top scorers. We'll set the cutoff at 650 points in a season.

```
topscorers <- players %>% filter(PTS > 650)
```

And like we've done in the past, we can add it to the chart with another geom_point. We'll make all the players grey, we'll make all the top scorers black.

```
ggplot() + geom_point(data=players, aes(x=MP, y=PTS), color="grey") + geom_point(data=t
```



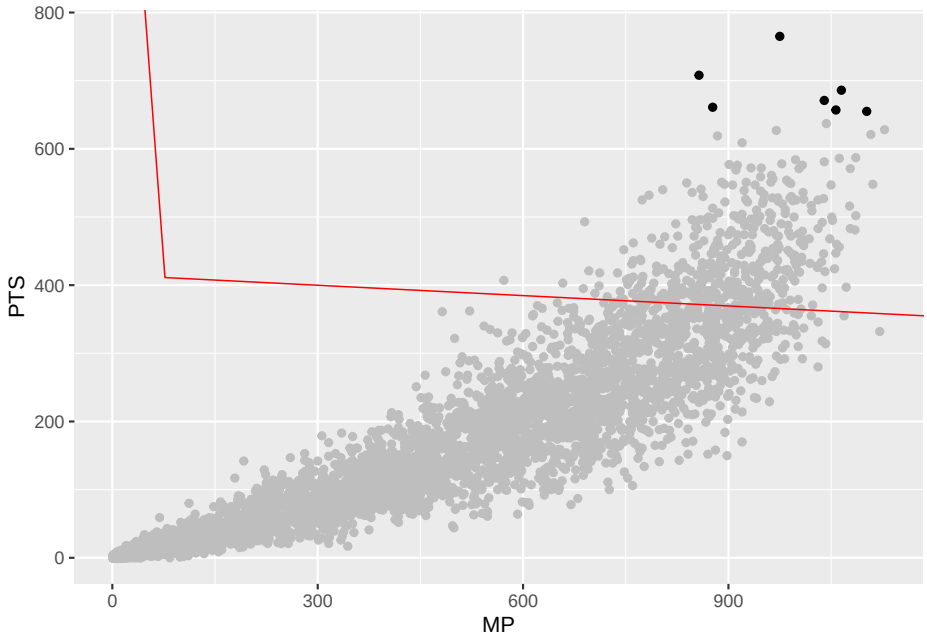
And like that, we're on the path to something publishable. We'll need to label those dots with `ggrepel` and we'll need to drop the default grey and add some headlines and all that. And, for the most part, we've got a solid chart.

But what if we could really draw the eye to those players. Let's draw a circle around them. In `ggalt`, there is a new geom called `geom_encircle`, which … does what you think it does. It encircles all the dots in a dataset.

So let's add geom_encircle and we'll just copy the data and the aes from our topscorers geom_point. Then, we need to give the encirclement a shape using s_shape – which is a number between 0 and 1 – and then how far away from the dots to draw the circle using expand, which is another number between 0 and 1.

Let's start with `s_shape` 1 and `expand` 1.

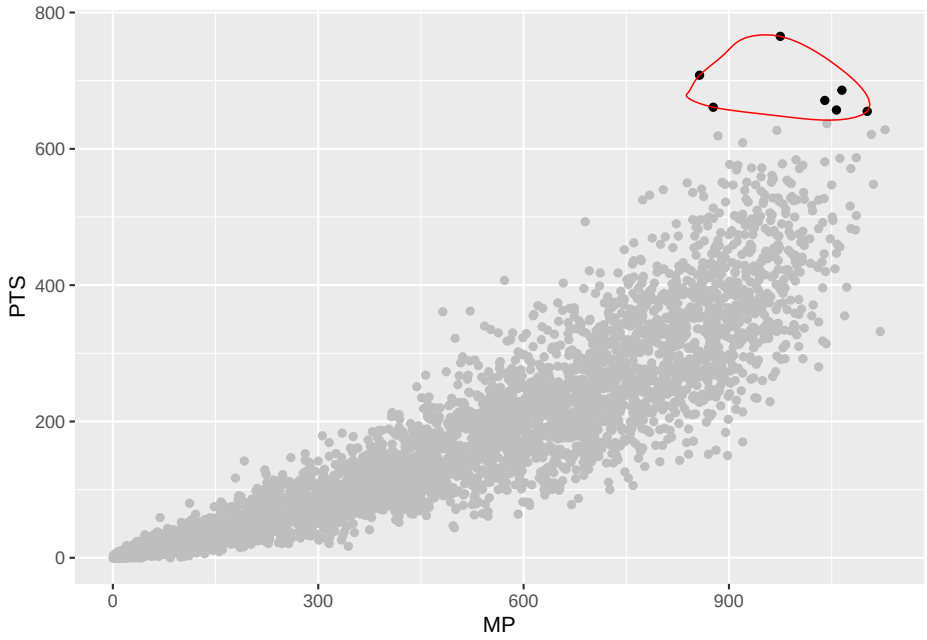
```
ggplot() +
  geom_point(data=players, aes(x=MP, y=PTS), color="grey") +
  geom_point(data=topscorers, aes(x=MP, y=PTS), color="black") +
  geom_encircle(data=topscorers, aes(x=MP, y=PTS), s_shape=1, expand=1, colour="red")
```



Whoa. That's … not good.

Let's go the opposite direction.

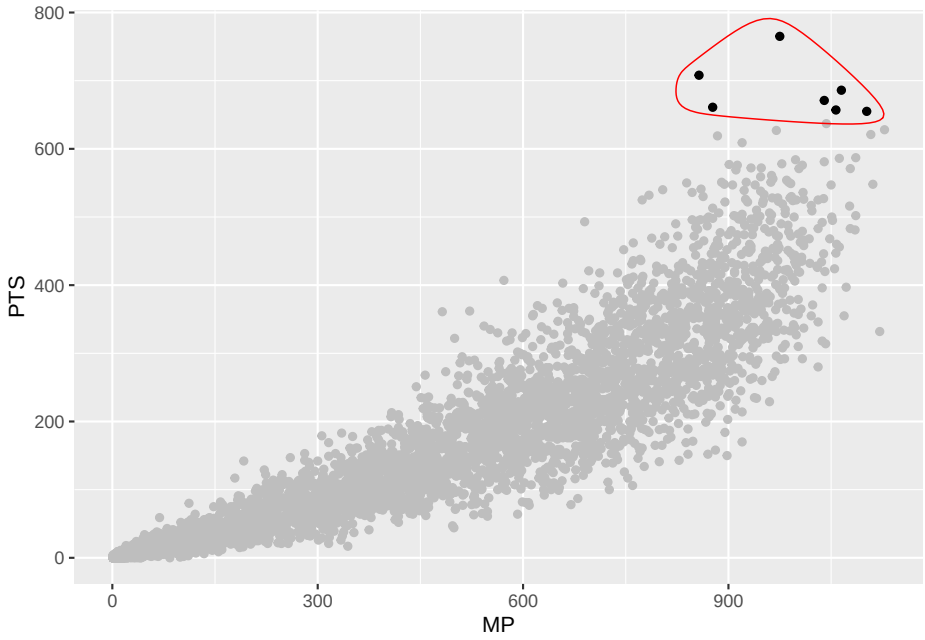
```
ggplot() +
  geom_point(data=players, aes(x=MP, y=PTS), color="grey") +
  geom_point(data=topscorers, aes(x=MP, y=PTS), color="black") +
  geom_encircle(data=topscorers, aes(x=MP, y=PTS), s_shape=0, expand=0, colour="red")
```



Better, but … the circle cuts through multiple dots.

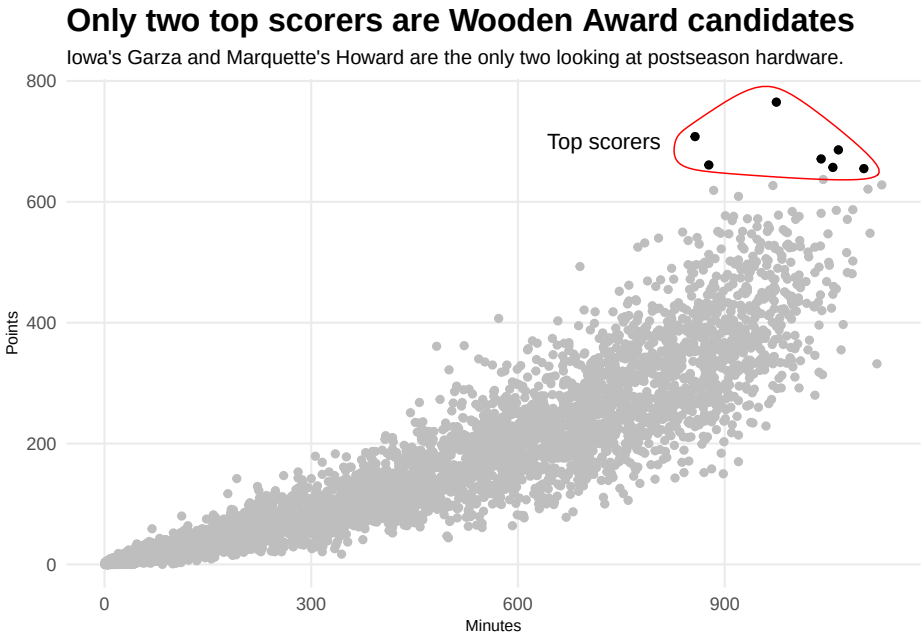
This takes a little bit of finessing, but a shape of .5 means the line will have some bend to it – it'll look more like someone circled it with a pen. Then, the expand is better if you use hundredths instead of tenths. So .01 instead of .1. Here's mine after fiddling with it for a bit.

```
ggplot() +
  geom_point(data=players, aes(x=MP, y=PTS), color="grey") +
  geom_point(data=topscorers, aes(x=MP, y=PTS), color="black") +
  geom_encircle(data=topscorers, aes(x=MP, y=PTS), s_shape=.5, expand=.03, colour="red")
```



Now let's clean this up and make it presentable. If you look at the top scorers, only two were Wooden Award finalists. So here's what a chart telling that story might look like.

```
ggplot() +
  geom_point(data=players, aes(x=MP, y=PTS), color="grey") +
  geom_point(data=topscorers, aes(x=MP, y=PTS), color="black") +
  geom_encircle(data=topscorers, aes(x=MP, y=PTS), s_shape=.5, expand=.03, colour="red"
  geom_text(aes(x=725, y=700, label="Top scorers")) +
  labs(title="Only two top scorers are Wooden Award candidates", subtitle="Iowa's Garza
  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()
    )
```



34.1 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.

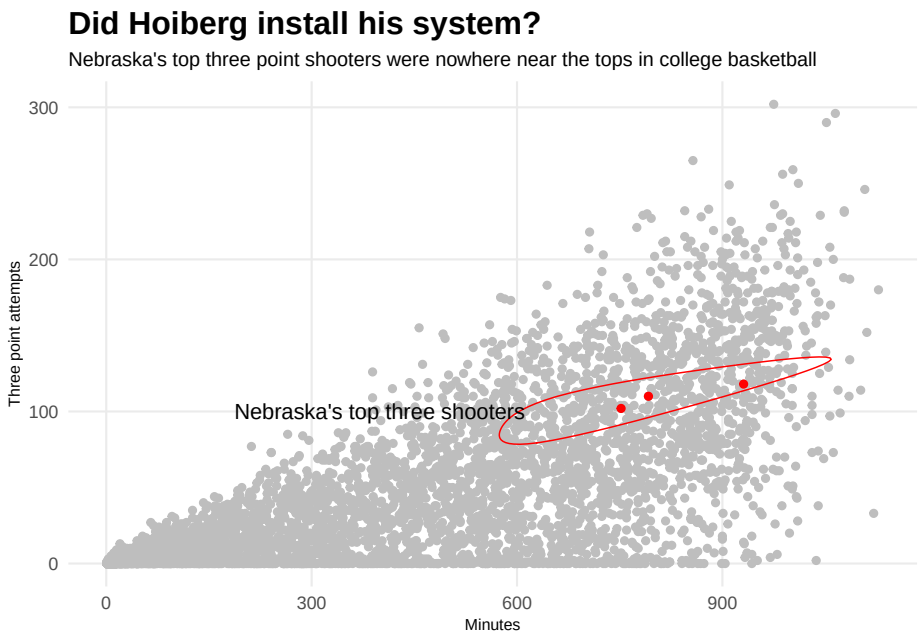
Let's grab the top three point attempt takers on the Nebraska roster. As of now, only one will be coming back.

```
nutop <- players %>% filter(Team == "Nebraska Cornhuskers") %>% top_n(3, `3PA`)
```

And just like above, we can plug in our players geom, our nutop dataframe into another geom, then encircle that dataframe. Slap some headlines and annotations on it and here's what we get:

```
ggplot() + geom_point(data=players, aes(x=MP, y=`3PA`), color="grey") + geom_point(data=nutop, ae
  geom_text(aes(x=400, y=100, label="Nebraska's top three shooters")) +
  labs(title="Did Hoiberg install his system?", subtitle="Nebraska's top three point shooters wer
  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()
```

```
)
```



The dot on the far right? Cam Mack. Oh what could have been.

Chapter 35

Bump charts

Not every library is on CRAN, which if you didn't know is the place where all the libraries we've installed up to this point are coming from. CRAN is a global repository of libraries, but they're libraries that are ready for a global audience of users, meaning they've been tested thoroughly and most of the bugs are out of them.

Developers are constantly working on new libraries, and some of them are interesting enough to try and stable enough to make a go of it. If you can handle a little uncertainty and understand that what you want to do might not work totally right, you can install those libraries from another code repository, GitHub. GitHub is a massive repository of all kinds of code developers want to share. This book is on GitHub, for instance.

To install a library from GitHub requires a couple of steps. The first is you need a set of tools to get them, called `devtools`. So go over to the console and install `devtools` like this:

```
install.packages("devtools")
```

The package we're going to try is called ggbump and it makes bump charts. Bump charts are like a subway map for rankings that change over time. We're going to make a bump chart of the college football playoff rankings this last season.

To get ggbump, run this in the console: `devtools::install_github("davidsjoberg/ggbump")`

Now we'll load up our libraries.

```
library(tidyverse)
library(ggbump)
```

Here's the dataset we'll use, which I scraped out of the college football playoff website:

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

```
## Parsed with column specification:
## cols(
##   Rank = col_double(),
##   Team = col_character(),
##   Record = col_character(),
##   Week = col_double(),
##   ShortTeam = col_character()
## )
```

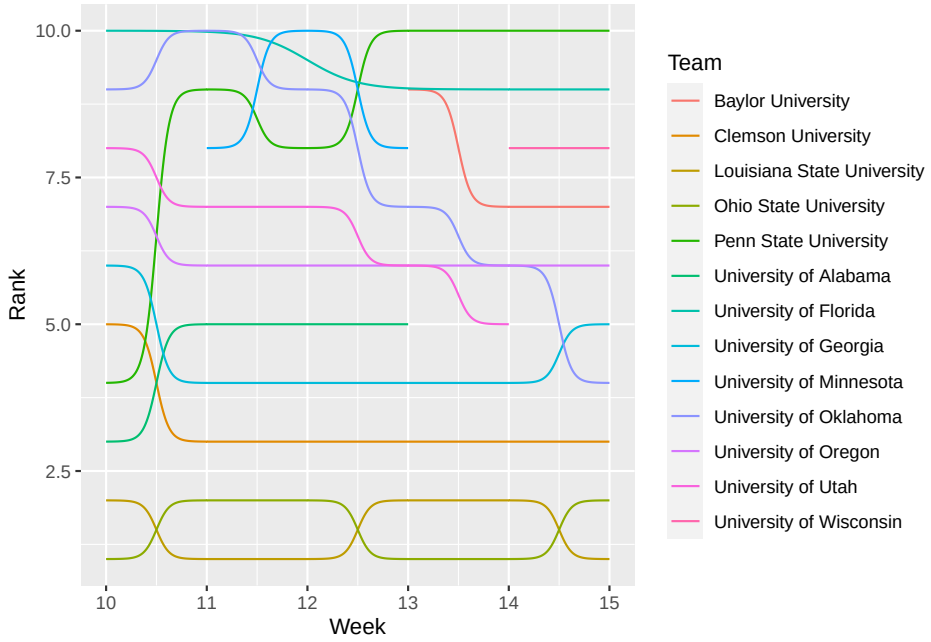
The point of a bump chart is to show how the ranking of something changed over time – you could do this with the top 25 in football or basketball. I've seen it done with European soccer league standings over a season. The requirements are that you have a row of data for a team, in that week, with their rank. So if you have 5 weeks of playoff rankings, like we do, you're going to have five rows of LSU, and five rows of Ohio State. you can see the basic look of the data by using head()

```
head(rankings)
```

```
## # A tibble: 6 x 5
##    Rank Team               Record  Week ShortTeam
##   <dbl> <chr>              <chr>  <dbl> <chr>
## 1     9 Baylor University  1-Oct     13 Baylor
## 2     7 Baylor University  1-Nov     14 Baylor
## 3     7 Baylor University  2-Nov     15 Baylor
## 4     5 Clemson University Sep-00    10 Clem.
## 5     3 Clemson University Oct-00    11 Clem.
## 6     3 Clemson University Nov-00    12 Clem.
```

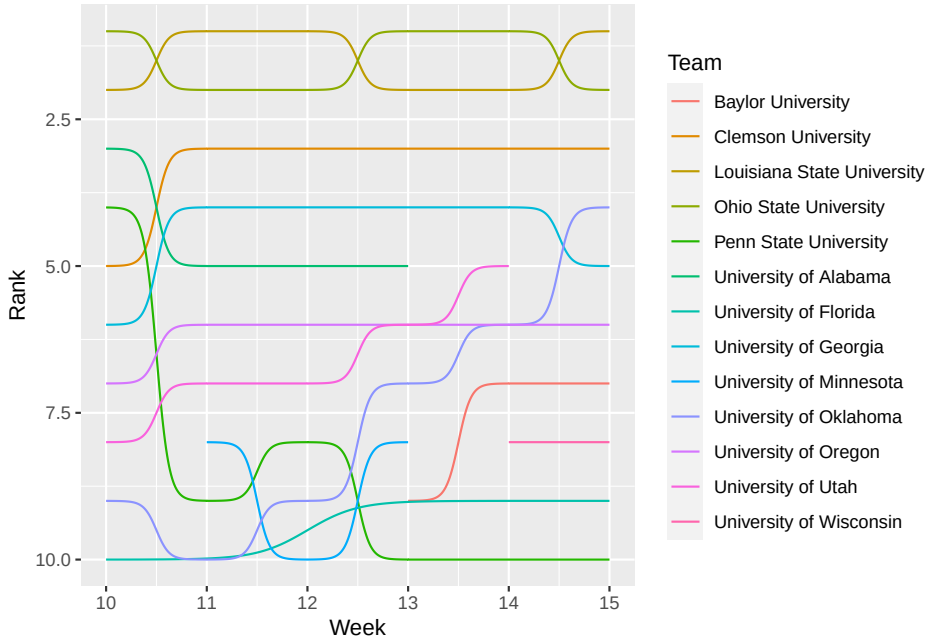
So Baylor was ranked in the 13th, 14th and 15th week, 9th, 7th and 7th, respectively. So our data is in the form we need it to be. Now we can make a bump chart. We'll start simple.

```
ggplot() + geom_bump(data=rankings, aes(x=Week, y=Rank, color=Team))
```



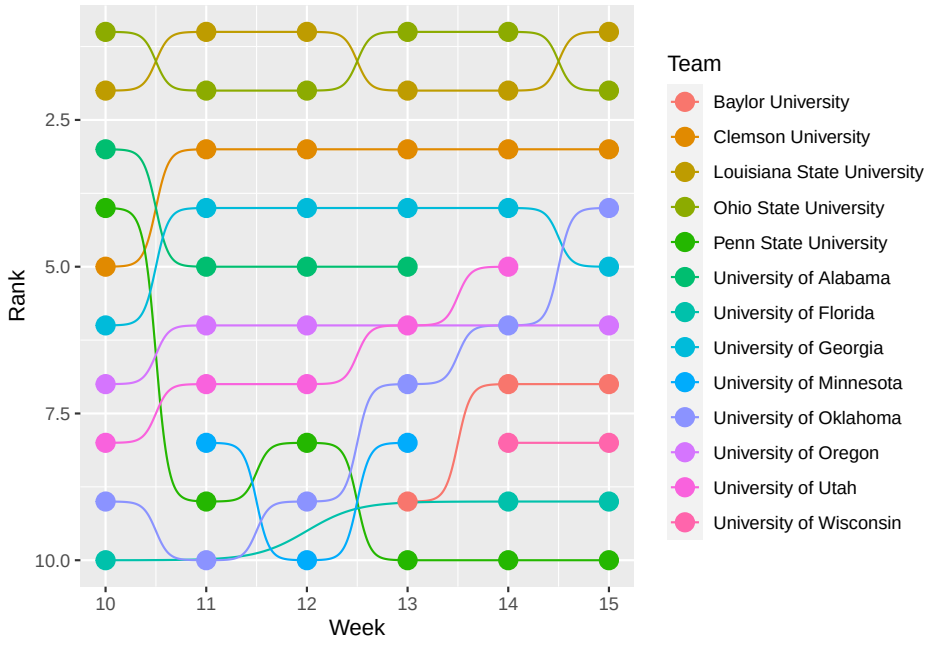
Well, it's a start. I'm immediately annoyed by the top teams being at the bottom. I learned a neat trick from ggbump that's been in ggplot all along – `scale_y_reverse()`

```
ggplot() + geom_bump(data=rankings, aes(x=Week, y=Rank, color=Team)) + scale_y_reverse()
```



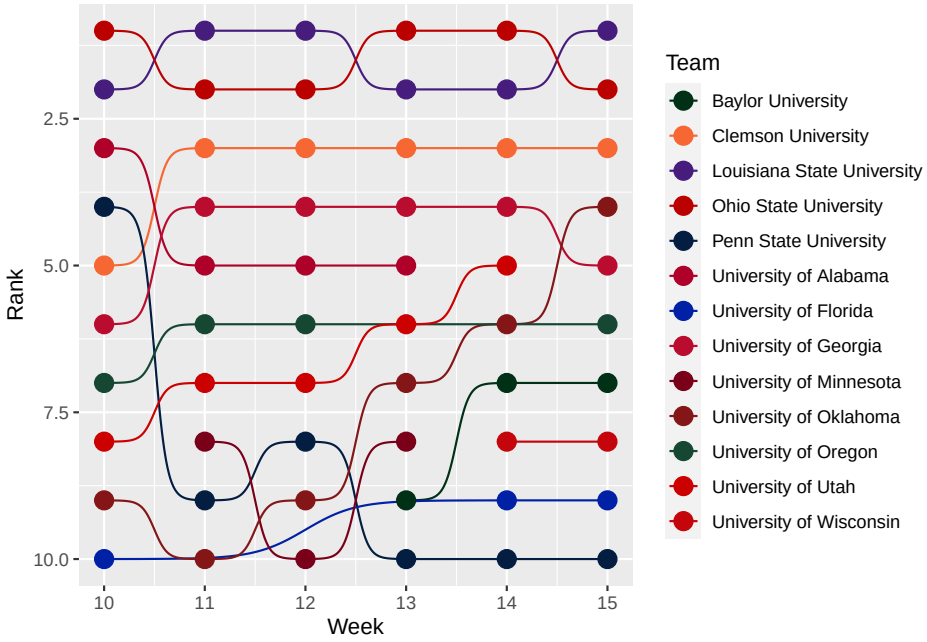
Better. But, still not great. Let's add a point at each week.

```
ggplot() +
  geom_bump(data=rankings, aes(x=Week, y=Rank, color=Team)) +
  geom_point(data=rankings, aes(x=Week, y=Rank, color=Team), size = 4) +
  scale_y_reverse()
```



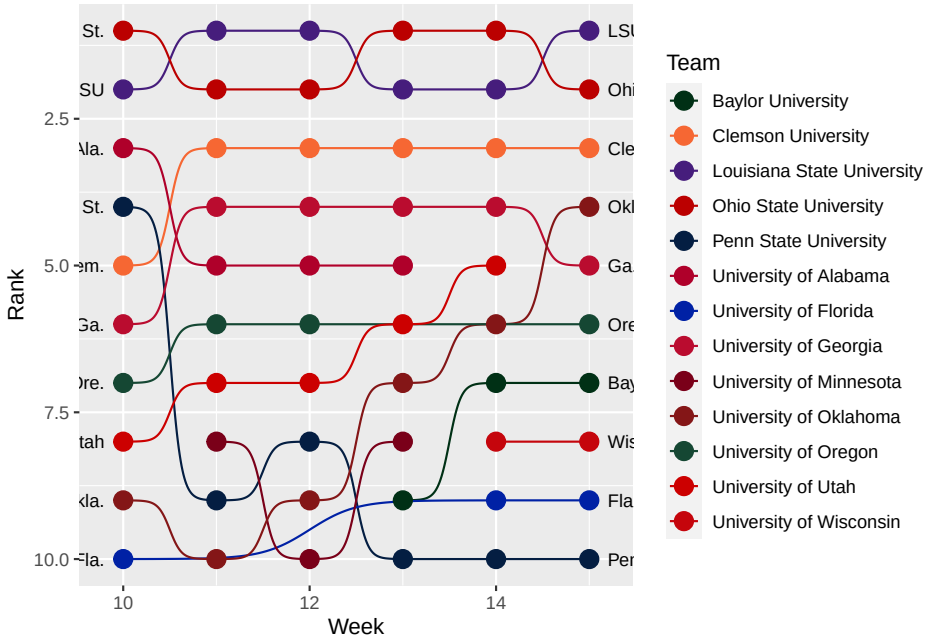
Another step. That makes it more subway map like. But the colors are all wrong. To fix this, we're going to use `scale_color_manual` and we're going to Google the hex codes for each team. The legend will tell you what order your `scale_color_manual` needs to be.

```
ggplot() +
  geom_bump(data=rankings, aes(x=Week, y=Rank, color=Team)) +
  geom_point(data=rankings, aes(x=Week, y=Rank, color=Team), size = 4) +
  scale_color_manual(values = c("#003015","#F66733", "#461D7C", "#bb0000", "#041E42",
  scale_y_reverse()
```



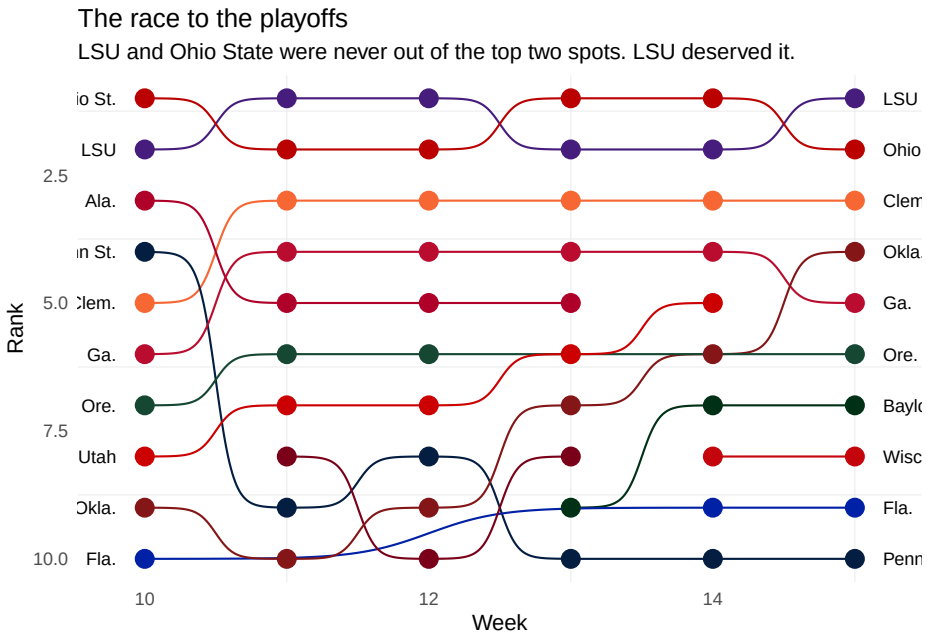
Another step. But the legend is annoying. And trying to find which red is Alabama vs Ohio State is hard. So what if we labeled each dot at the beginning and end? We can do that with some clever usage of geom_text and a little dplyr filtering inside the data step. We filter out the first and last weeks, then use hjust – horizontal justification – to move them left or right.

```
ggplot() +
  geom_bump(data=rankings, aes(x=Week, y=Rank, color=Team)) +
  geom_point(data=rankings, aes(x=Week, y=Rank, color=Team), size = 4) +
  geom_text(data = rankings %>% filter(Week == min(Week)), aes(x = Week - .2, y=Rank, label = Sho
  geom_text(data = rankings %>% filter(Week == max(Week)), aes(x = Week + .2, y=Rank, label = Sho
  scale_color_manual(values = c("#003015","#F66733", "#461D7C", "#bb0000", "#041E42", "#AF002A","
  scale_y_reverse()
```



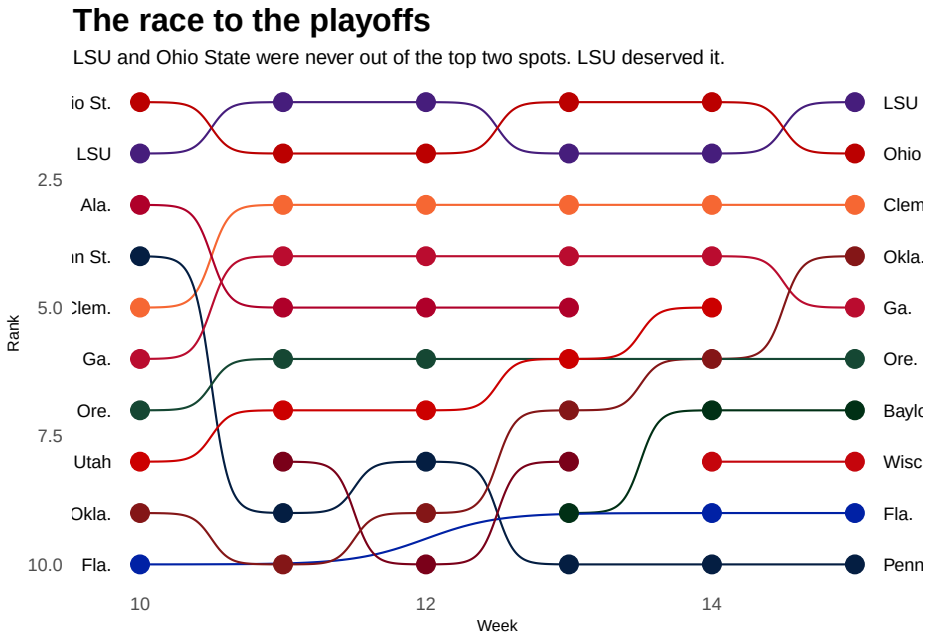
Better, but the legend is still there. We can drop it in a theme directive by saying `legend.position = "none"`. We'll also throw a theme_minimal on there to drop the default grey, and we'll add some better labeling.

```
ggplot() +
  geom_bump(data=rankings, aes(x=Week, y=Rank, color=Team)) +
  geom_point(data=rankings, aes(x=Week, y=Rank, color=Team), size = 4) +
  geom_text(data = rankings %>% filter(Week == min(Week)), aes(x = Week - .2, y=Rank,
  geom_text(data = rankings %>% filter(Week == max(Week)), aes(x = Week + .2, y=Rank,
  labs(title="The race to the playoffs", subtitle="LSU and Ohio State were never out o
  theme_minimal() +
  theme(
    legend.position = "none",
    panel.grid.major = element_blank()
    ) +
  scale_color_manual(values = c("#003015","#F66733", "#461D7C", "#bb0000", "#041E42",
  scale_y_reverse()
```



Now let's fix our text hierarchy.

```
ggplot() +
  geom_bump(data=rankings, aes(x=Week, y=Rank, color=Team)) +
  geom_point(data=rankings, aes(x=Week, y=Rank, color=Team), size = 4) +
  geom_text(data = rankings %>% filter(Week == min(Week)), aes(x = Week - .2, y=Rank, label = Sho
  geom_text(data = rankings %>% filter(Week == max(Week)), aes(x = Week + .2, y=Rank, label = Sho
  labs(title="The race to the playoffs", subtitle="LSU and Ohio State were never out of the top t
  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("#003015","#F66733", "#461D7C", "#bb0000", "#041E42", "#AF002A","
  scale_y_reverse()
```



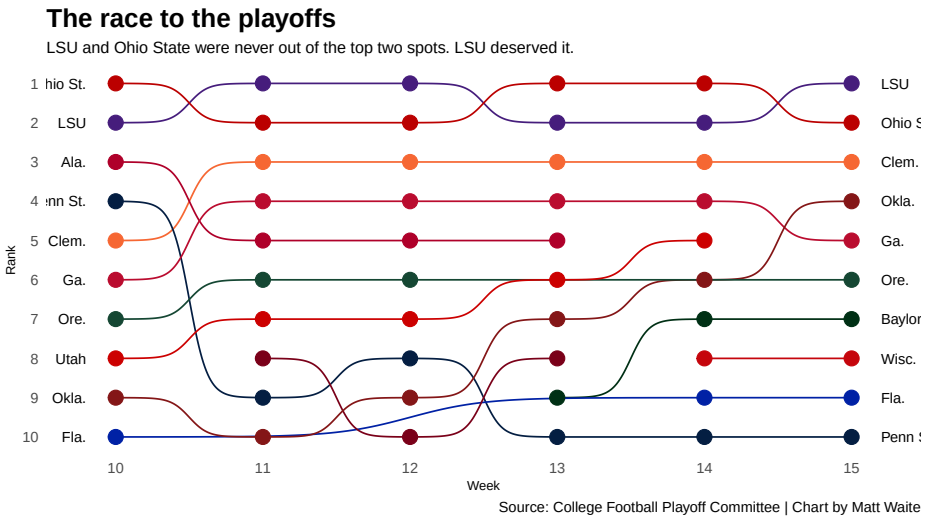
And the last thing: anyone else annoyed at 7.5th place on the left? We can fix that too by specifying the breaks in scale_y_reverse. We can do that with the x axis as well, but since we haven't reversed it, we do that in `scale_x_continuous` with the same breaks. Also: forgot my source and credit line.

One last thing: Let's change the width of the chart to make Ohio State and Penn State fit. We can do that by adding `fig.width=X` in the `{r}` setup in your block. So something like this:

```
{r fig.width=8}
```

```
ggplot() +
  geom_bump(data=rankings, aes(x=Week, y=Rank, color=Team)) +
  geom_point(data=rankings, aes(x=Week, y=Rank, color=Team), size = 4) +
  geom_text(data = rankings %>% filter(Week == min(Week)), aes(x = Week - .2, y=Rank,
  geom_text(data = rankings %>% filter(Week == max(Week)), aes(x = Week + .2, y=Rank,
  labs(title="The race to the playoffs", subtitle="LSU and Ohio State were never out o
  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("#003015","#F66733", "#461D7C", "#bb0000", "#041E42",
```

```
  scale_x_continuous(breaks=c(10,11,12,13,14,15)) +
  scale_y_reverse(breaks=c(1,2,3,4,5,6,7,8,9,10))
```



Chapter 36

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 toolboxt. Let's take a look at a couple.

First, you know what you need.

```
library(tidyverse)
```

Now, two examples.

36.1 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 – here's a dataset of NBA players and their advanced metrics.

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

```
## Warning: Missing column names filled in: 'X20' [20], 'X25' [25]
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Player = col_character(),
##   Pos = col_character(),
##   Tm = col_character(),
##   X20 = col_logical(),
##   X25 = col_logical()
```

```
## )
```

```
## See spec(...) for full column specifications.
```

Let's take a look:

```
head(nbaplayers)
```

```
## # A tibble: 6 x 29
##      Rk Player Pos     Age Tm        G    MP   PER `TS%` `3PAr`   FTr `ORB%`
##   <dbl> <chr>  <chr> <dbl> <chr> <dbl> <dbl> <dbl> <dbl>  <dbl> <dbl>  <dbl>
## 1     1 "Stev~ C        26 OKC      58  1564  20.8 0.605  0.007 0.413   14.4
## 2     2 "Bam ~ PF       22 MIA      65  2235  20.6 0.606  0.018 0.476    8.7
## 3     3 "LaMa~ C        34 SAS      53  1754  19.8 0.571  0.198 0.241    6.3
## 4     4 "Nick~ SG       21 NOP      41   501   7.6 0.441  0.515 0.123    1.7
## 5     5 "Gray~ SG       24 MEM      30   498  11.4 0.577  0.517 0.199    1.1
## 6     6 "Jarr~ C        21 BRK      64  1647  20.3 0.658  0.012 0.574   12.5
## # ... with 17 more variables: `DRB%` <dbl>, `TRB%` <dbl>, `AST%` <dbl>,
## #   `STL%` <dbl>, `BLK%` <dbl>, `TOV%` <dbl>, `USG%` <dbl>, X20 <lgl>,
## #   OWS <dbl>, DWS <dbl>, WS <dbl>, `WS/48` <dbl>, X25 <lgl>, OBPM <dbl>,
## #   DBPM <dbl>, BPM <dbl>, VORP <dbl>
```

For starters, see the two empty columns that don't have any names? X20 and X25. We can get rid of them. They're empty.

```
nbaplayers <- nbaplayers %>% select(-X20, -X25)
```

Now let's look at those names:

```
head(nbaplayers)
```

```
## # A tibble: 6 x 27
##      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 "Stev~ C        26 OKC      58  1564  20.8 0.605  0.007 0.413   14.4
## 2     2 "Bam ~ PF       22 MIA      65  2235  20.6 0.606  0.018 0.476    8.7
## 3     3 "LaMa~ C        34 SAS      53  1754  19.8 0.571  0.198 0.241    6.3
## 4     4 "Nick~ SG       21 NOP      41   501   7.6 0.441  0.515 0.123    1.7
## 5     5 "Gray~ SG       24 MEM      30   498  11.4 0.577  0.517 0.199    1.1
## 6     6 "Jarr~ C        21 BRK      64  1647  20.3 0.658  0.012 0.574   12.5
## # ... with 15 more variables: `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>
```

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 expres-

sions 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.

```
nbaplayers %>% mutate(Player=gsub("\\\\.*","",Player)) %>% head()
```

```
## # A tibble: 6 x 27
##      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 Steve~ C        26 OKC      58  1564  20.8 0.605  0.007 0.413   14.4
## 2     2 Bam A~ PF       22 MIA      65  2235  20.6 0.606  0.018 0.476    8.7
## 3     3 LaMar~ C        34 SAS      53  1754  19.8 0.571  0.198 0.241    6.3
## 4     4 Nicke~ SG       21 NOP      41   501   7.6 0.441  0.515 0.123    1.7
## 5     5 Grays~ SG       24 MEM      30   498  11.4 0.577  0.517 0.199    1.1
## 6     6 Jarre~ C        21 BRK      64  1647  20.3 0.658  0.012 0.574   12.5
## # ... with 15 more variables: `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>
```

Just like that, the trash is gone.

36.2 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. Here's an example.

Let's import it and take a look.

```
kills <- read_csv("data/killsperset.csv")
```

```
## Parsed with column specification:
## cols(
##   Rank = col_double(),
##   Player = col_character(),
##   Cl = col_character(),
##   Ht = col_character(),
##   Pos = col_character(),
##   S = col_double(),
##   Kills = col_double(),
##   `Per Set` = col_double(),
##   Season = col_character()
## )
```

```
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 San~ So.   6-2   OH       90   526      5.84 2017-2~
## 2     2 Pilar Victoria, Arkansa~ Sr.   5-11  OH      116   634      5.47 2017-2~
## 3     3 Laura Milos, Oral Rober~ Sr.   5-10  OH      106   560      5.28 2017-2~
## 4     4 Carlyle Nusbaum, Lipsco~ Jr.   5-10  OH      100   522      5.22 2017-2~
## 5     5 Veronica Jones-Perry, B~ Jr.   6-0   OH      118   569      4.82 2017-2~
## 6     6 Torrey Van Winden, Cal ~ So.   6-3   OH      101   477      4.72 2017-2~
```

First things first, Player isn't just player, it's player, school and conference, all in one. And Ht is a character field – and in feet and inches.

So … this is a mess. But there is a pattern. See it? A comma after the player's name. The Conference is in parens. We can use that.

For this, we're going to use a `tidyr` function called `separate` to split columns into multiple columns based on a character. We'll do this step by step.

First, let's use that comma to split the player and the rest. Ignore the head at the end. That's just to keep it from showing you all 150.

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

```
## # A tibble: 6 x 10
##    Rank Player     School         Cl    Ht    Pos       S Kills `Per Set` Season
##   <dbl> <chr>      <chr>          <chr> <chr> <chr> <dbl> <dbl>     <dbl> <chr>
## 1     1 Lindsey R~ " UC Santa Ba~ So.   6-2   OH       90   526      5.84 2017-~
```

```
## 2     2 Pilar Vic~ " Arkansas (S~ Sr.   5-11  OH      116   634      5.47 2017-~
## 3     3 Laura Mil~ " Oral Robert~ Sr.   5-10  OH      106   560      5.28 2017-~
## 4     4 Carlyle N~ " Lipscomb (A~ Jr.   5-10  OH      100   522      5.22 2017-~
## 5     5 Veronica ~ " BYU (WCC)"   Jr.   6-0   OH      118   569      4.82 2017-~
## 6     6 Torrey Va~ " Cal Poly (B~ 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 \\.

```
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 Pennslyvania respectively. Well, we can fix that with some gsub and we'll use a switch called `fixed`, which when set to TRUE it means this literal string, no special characters.

```
kills %>%
  separate(Player, into=c("Player", "School"), sep=",") %>%
  mutate(School = gsub("(FL)", "FL", School, fixed=TRUE)) %>%
  mutate(School = gsub("(NY)", "NY", School, fixed=TRUE)) %>%
  mutate(School = gsub("(PA)", "PA", School, fixed=TRUE)) %>%
  separate(School, into=c("School", "Conference"), sep="\\(") %>%
  head()
```

```
## # A tibble: 6 x 11
##    Rank Player  School 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 Linds~ " 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 Carly~ " Lip~ ASUN       Jr.       5     10 OH      100   522      5.22
## 5     5 Veron~ " BYU~ WCC        Jr.       6      0 OH      118   569      4.82
## 6     6 Torre~ " Cal~ Big West   So.       6      3 OH      101   477      4.72
## # ... with 1 more variable: Season <chr>
```

But how do we turn that into a height? Math!

```
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 Linds~ " 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 Carly~ " Lip~ ASUN       Jr.       5     10 OH      100   522      5.22
## 5     5 Veron~ " BYU~ WCC        Jr.       6      0 OH      118   569      4.82
## 6     6 Torre~ " Cal~ Big West   So.       6      3 OH      101   477      4.72
## # ... with 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.

Chapter 37

Assignments

This is a collection of assignments I've used in my Sports Data Analysis and Visualization course at the University of Nebraska-Lincoln. The overriding philosophy is to have students do lots of small assignments that directly apply what they learned, and often pulling from other assignments. Each small assignment is just a few points each – I make them 5 points each and make the grading a yes/no decision on 5 different questions – so a bad grade on one doesn't matter. Then, twice during the semester, I have them create blog posts with visualizations on a topic of their choosing. The topic must have a point of view – Nebraska's woes on third down are why the team is struggling, for example – and must be backed up with data. They have to write a completely documented R Notebook explaining what they did and why; they have to write a publicly facing blog post for a general audience and that post has to have at least three graphs backing up their point; and they have to give a lightning talk (no more than five minutes) in class about what they have found. Those two assignments are typically worth 50 percent of the course grade.

I think rubrics are crap, but I give students these questions as a guide to what I'm expecting:

1. Did you read the data into a dataframe?
2. Did you use the skill discussed in the chapter correctly?
3. Did you answer all the questions posed by the assignment?
4. Did you use Markdown comments to explain your steps, what you did and why?

Chapter 1: Intro

- Install Slack on your computer and your phone.
- If on a Mac, install the Command Line Tools.
- Install R for your computer.

- Install R Studio Desktop for your computer ONLY AFTER YOU HAVE INSTALLED R

Chapter 2: Basics

Part 1:

In the console, type `install.packages("swirl")`

Then `library("swirl")`

Then `swirl()`

Follow instructions on the screen. Each time you are asked if which one you want, you want the first one. The basics, the beginning, the first parts. All the first ones. Then just follow the instructions on the screen.

Part 2:

Create an R notebook (which you should have done if you were following along). In it, delete all the generated text from it, so you have a blank document. Then, write a sentence in the document telling me what the last thing you did in Swirl was. Then add a code block (see about inserting code in the chapter) and add two numbers together. Any two numbers. Run that code block. Save the file and submit the .Rmd file created when you save it to Canvas. That's it. Simple.

Chapter 3: Data, structures and types

Using what you learned in the chapter, fetch the list of the Big Ten's leading tacklers. Submit the CSV file to Canvas. In the comments, label each field type. What are they? Dates? Characters? Numeric?

Chapter 4: Aggregates

Import this dataset of every college basketball game in the 2018-19 season. Using what you learned in the chapter, answer the following questions:

1. What team shot the most shots?
2. What team averaged the most shots?
3. What team had the highest median number of shots?
4. How much difference is there between the top average shots team and the top median shots team? Why do you think that is?

Chapter 5: Mutating Data

Import this dataset of every college basketball game in the 2018-19 season. Using what you learned in the chapter, mutate a new variable: differential. Differential is the difference between the team score and the opponent score. A positive number means the team in question won. A negative number means the team in question lost. After creating the differential, average them together and sort them in descending order. Which team had the highest average point differential in college basketball? In other words, which team consistently won by the largest margins?

Chapter 6: Filters and selections

Import the data of every college basketball player's season stats in 2018-19 season. Using this data, let's get closer to a real answer to where the cutoff for true shooting season should be from the chapter. First, find the median number of shots attempted in the season, then set the cutoff filter for who had the best true shooting percentage using that number.

Chapter 7: Transforming data

Import this dataset of college football attendance data from 2013-2018. This data is long data – one team, one year, one row. We need it to be wide data. Hint: it'll be much easier if you select only the columns you need to make it wide instead of using them all. Submit your notebook.

Chapter 8: Simulations

On Feb. 6, Nebraska's basketball team had a nightmare night shooting the ball. They attempted 57 shots … and made only 12. The team shot .429 on the season. Simulate 1000 games of them taking 57 shots using their season long .429 as the probability that they'll make a shot. How many times do they make just 12?

Chapter 9: Correlations and regressions

Do the same thing described in the chapter, but for defense. Report your R-squared number, your p-value, what those mean and from that, how close does it come to predicting the Iowa Nebraska game?

Chapter 10: Multiple regression

You have been hired by Fred Hoiberg to build a team. He's interested in the model started in the chapter, but wants more.

There are more predictors to be added to our model. You are to find two. Two that contribute to the predictive quality of the model without largely overlapping another predictor.

In your notebook, report the adjusted r-squared you achieved.

You are to generate a new set of coefficients, a new formula and a new set of numbers of what a conference champion would expect in terms of differential. I've done a lot of work for you. Continue it. Add two more predictors and complete the prediction. And compare that to Nebraska of this season.

Turn in your notebook with these answers and comments to the code you added, making sure to add WHY you are doing things. Why did you select those two variables.

Chapter 11: Residuals

Using the same data from the chapter, model defensive third down percentage and defensive points allowed. Which teams are overperforming that model given the residual analysis?

Chapter 12: Z scores

Refine the composite Z Score I started in the chapter. Add two more elements to it. What else do you think is important to the success of a basketball team? I've got shooting, rebounds and the opponents shooting. What else would you add?

In an R Notebook, make your case for the two elements you are adding – what is your logic? Why these two?. Then, follow my steps here until you get to the `teamquality` dataframe step, where you'll need to add the fields you are going to add to the composite. Then you'll need to add your fields to the `teamtotals` dataframe. Then you'll need to adjust `teamzscore`.

Finally, look at your ranking of Big Ten teams and compare it to mine. Did adding more elements to the composite change anything? Explain the differences in your notebook. Which one do you think is more accurate? I won't be offended if you say yours, but why do you feel that way?

Chapter 13: Intro to ggplot

Take this same attendance data. I want you to produce a bar chart of the top 10 schools by percent change in attendance between 2018 and 2013. I want you to change the title and the labels and I want you to apply a theme different from the ones I used above. You can find more themes in the ggplot documentation.

Chapter 14: Stacked bar charts

I want you to make this same chart, except I want you to make the weight the percentage of the total number of graduates that gender represents. You'll be mutating a new field to create that percentage. You'll then chart it with the fill. The end result should be a stacked bar chart allowing you to compare genders between universities. Answer the following question: Which schools have the largest gender imbalances?

Chapter 15: Waffle charts

Compare Nebraska and Michgan's night on the basketball court using a Waffle chart and another metric than what I've done above for the game.

Here's the library's documentation. Here's the stats from the game.

Turn in your notebook with your waffle chart. It must contain these two things:

- Your waffle chart
- A written narrative of what it says. What does your waffle chart say about how that game turned out?

Chapter 16: Line Charts

Import this dataset of every college basketball game in the 2018-19 season.

- How does Nebraska's shooting percentage compare to the Big Ten over the season? Put the Big Ten on the same chart as Nebraska, you'll need

two dataframes, two geoms and with your Big Ten dataframe, you need to use `group` in the aesthetic.

- After working on this chart, your boss comes in and says they don't care about field goal percentage anymore. They just care about three-point shooting because they read on some blog that three-point shooting was all the rage. Change what you need to change to make your line chart now about how the season has gone behind the three-point line. How does Nebraska compare to the rest of the Big Ten?

Chapter 17: Step charts

Re-make the chart in the chapter, but with rebounding. I want you to visualize the differential between our rebounds and their rebounds, and then plot the step chart showing over the course of the season. Highlight Nebraska. Highlight the top team. Add annotation layers to label both of them.

Chapter 18: Ridge charts

You've been hired by Fred Hoiberg to tell him how to win the Big Ten. He's not impressed with that I came up with. So what you need to do is look for a *composite* measure that produces a meaningful ridgeplot. What that means is you're going to mutate `wintotalgroupinglogs` one more time. Is the differential between rebounding meaningful instead of just the total? Or assists? Or something else? Your call. Your goal is to produce a ridgeplot that tells The Mayor he needs to focus on doing X better than the opponent to win a Big Ten title.

Chapter 19: Lollipop charts

You've been hired by Fred Hoiberg to tell him how to win the Big Ten. He's not impressed with that I came up with. So what else could you look at with lollipop charts? Your call. Your goal is to produce a lollipop chart that tells The Mayor he needs to focus on the gap between X and Y if he wants to win a Big Ten title.

Chapter 20: Scatterplot

Using the data from the walkthrough, model and graph two other elements of Nebraska's season versus wins. How much does your choices of metrics predict the season? What do the scatterplots of what you chose look like? What do the linear models say (r-squared, p-values)? How predictive are they, i.e. using y=mx+b, how close to Nebraska's win total do your models get to?

Chapter 21: Facet Wraps

Which Big Ten teams were good a shooting three point shots? Which teams weren't? Using a facet grid, chart each teams three point shooting season against the league average.

Chapter 22

Import this dataset of every college basketball game in the 2018-19 season.

Create a dataframe that shows the 10 best or 10 worst at something. Or rank the Big Ten. Your choice. Then use formattable to show in both a table and visually how the best were different from the worst.

Export it to a PNG using the example above. Then, in Illustrator, add a headline, chatter, source and credit lines. Turn in the PNG file you export from Illustrator.

New Chapter

Import this dataset of every college basketball game in the 2018-19 season. Create a bubble chart looking at two stats to make your scatterplot and a third making the size of your bubble. Make the color the conference name.

I want to see your bubble chart, but more importantly, I want you to discuss if what you came up with makes an effective bubble chart. Does it tell a story? Does the size of the bubble enhance understanding? No is an acceptable answer. But explain why it did or didn't work.

New Chapter

Your turn: Let's evaluate the second part of the quote from the chapter: November basketball tells you where you are.

We've looked at wins. What else could you look at over the course of the season that tells you where you are? Pick a metric. Explain your choice. Make a circular bar chart. Evaluate the result. What does it say?

Chapter 23: Rvest

I am a huge Premiere League fan, so I want data on the league. For now, I just want teams. Scrape the team data at the top, but before you do, look at the header. Is it one row? Does that make it standard? Nope. So what now?

Chapter 24: Advanced Rvest

I don't usually assign an advanced rvest assignment because I don't want to turn 30 students loose on some poor provider's servers.

Note: There are no assignments for annotations and finishing touches. In my classes, I have students present two major visual stories where they have to incorporate the elements of those assignments as part of their grade.

Chapter 30: Plotly

First, create a simple ggplot like we did above exploring WRAA – weighted runs above average – as your x value and and plate appearances (PA) as your y variable.

Next, create a plotly visualization using the same two variables. Alter the hover elements to show relevant data. If you leave it the same from the chapter, you lose points.

Export your plotly visualization to plotly's website. Include a link of your viz in your notebook. In your notebook, discuss the relative advantages and disadvantages of this interactive plot versus the static plots we've been doing.

Chapter 31: Clustering

We looked at who Cam Mack's peers are, but what about the team? Use k-means clustering on a dataset of every college basketball team's season stats and determine who Nebraska's peers are.

To complete this assignment, you'll need to pick the metrics you want to measure teams by. One note – teams haven't played the same number of games, so it would be wise to either focus on the per game or percentage metrics, either using them or creating them yourself. You'll then need to scale them. You'll need to decide the optimal number of clusters (k) and then run them. Combine the data back, determine which cluster Nebraska is in in your clustering and then show Nebraska's peers.

In your notebook, write a few sentences and answering this question: Is the peer group Nebraska is in fair?

Chapter 32: Rtweet

Using the Rtweet library, gather tweets about the Maryland game Saturday night. WARNING: The API only lets you go back 18000 tweets – we settled on 8000 in the walkthrough. If you wait too long to run the scrape after the game, you won't get the tweets you need to complete it. If you don't intend to do the assignment Saturday night, at least scrape the data and save it as a CSV as detailed in the chapter. You can then analyze it any time you want.

Analyze sentiment and chart it as in the chapter. Compare the Maryland game to the Purdue game. Are they different? Is sentiment better or worse for one or the other? Describe the differences in your notebook.

Chapter 38

Appendix

These are some additional materials I use in my classes.

38.1 How to get help in this class

> This is the contents of a document I send out every semester. I use Slack to help students with code problems outside of class. It's much easier than other options, such as email. A suggestion: set ground rules on when you will and won't answer Slack messages.

1. **Use Slack**. Email is a miserable way to handle technological questions. I'm not answering code questions via email. On Slack we can have a back and forth where we solve this quickly instead of waiting on each other to respond to an email.
2. **Don't use screenshots**. Tell me what you're trying to do and then copy and paste your code into the Slack message.
3. **Always copy and paste the error message you are getting**. There is a near infinite number of things you could have done and a nearly limitless number of errors you could be getting. Both help.

Slack tips

- Slack uses Markdown in messages. Did you know your code blocks in R Notebook are Markdown themselves? If you copy the whole block – with the "‘ and everything, Slack will format it like this:

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

hist(simulations)

table(simulations)
```

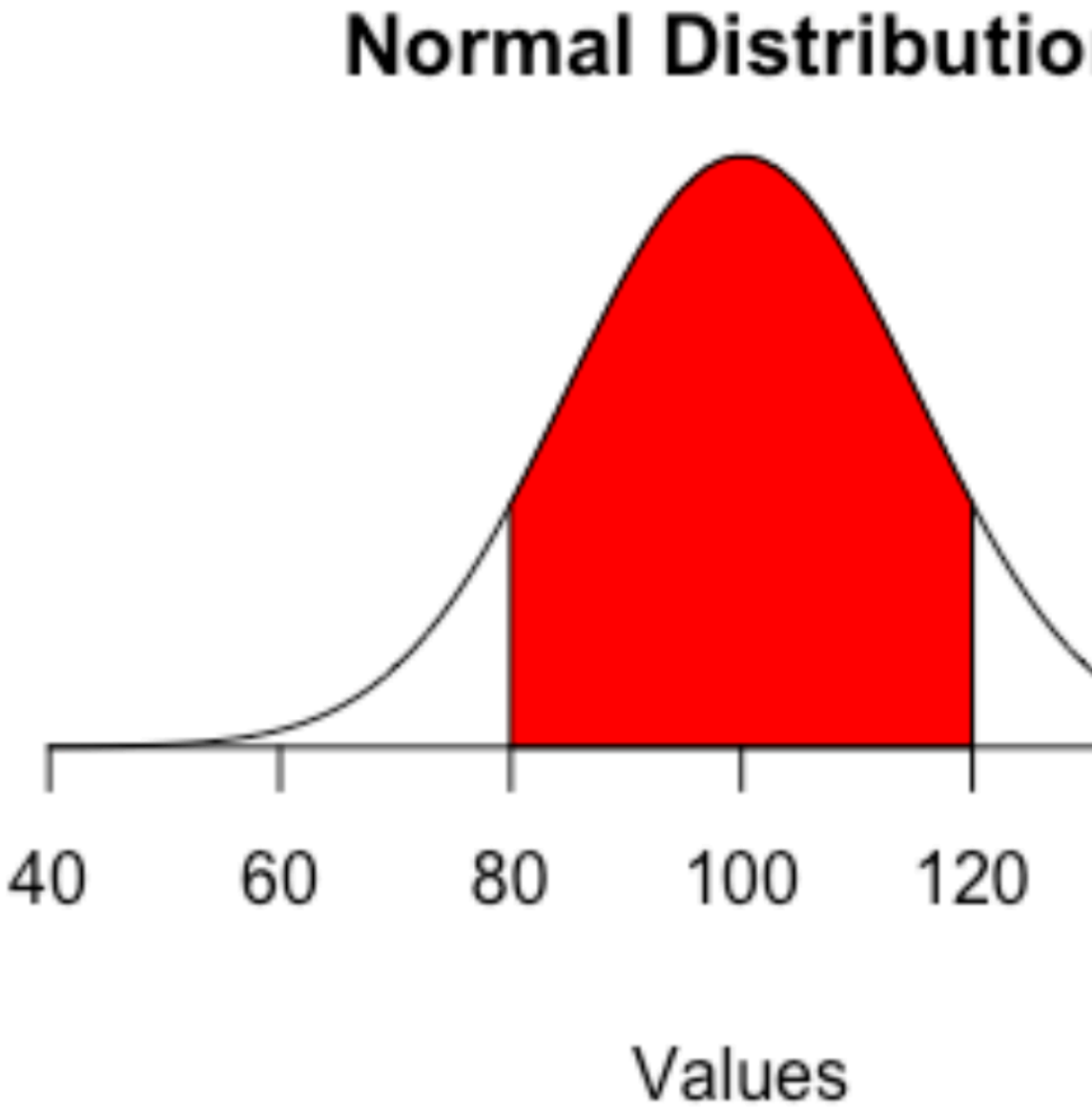
- Using the Slack app will also mean getting alerted to messages right away. If you are logging in through a web browser, you won't know when I've responded. If we're having a Slack conversation and I see you log in, send me a message, then disappear right away, I know you're using a browser and when that conversation drags because you aren't getting messages, **I'm going to get frustrated**. You likely aren't the only one asking for help at that moment.

Chapter 39

Simulations

Two seasons ago, James Palmer Jr. shot 139 three point attempts and made 43 of them for a .309 shooting percentage last year. A few weeks into last season, he was 7 for 39 – a paltry .179. Is something wrong or is this just bad luck?

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



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

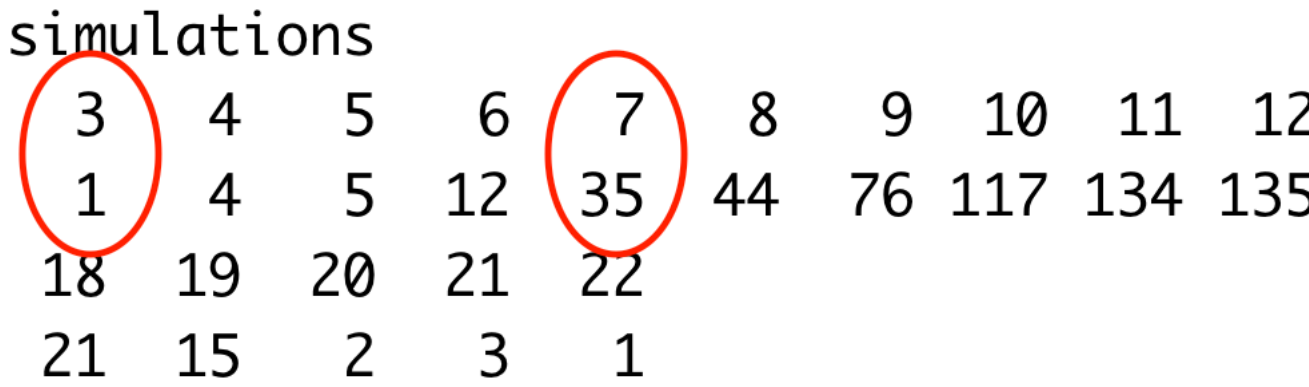
We do this using a base R function called `rbinom` or binomial distribution. So what that means is there's a normally distrubuted chance that James Palmer Jr. is going to shoot above and below his career three point shooting percentage.

If we randomly assign values in that distribution 1000 times, how many times will it come up 7, like this example?

```
set.seed(1234)

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

table(simulations)
```

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

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

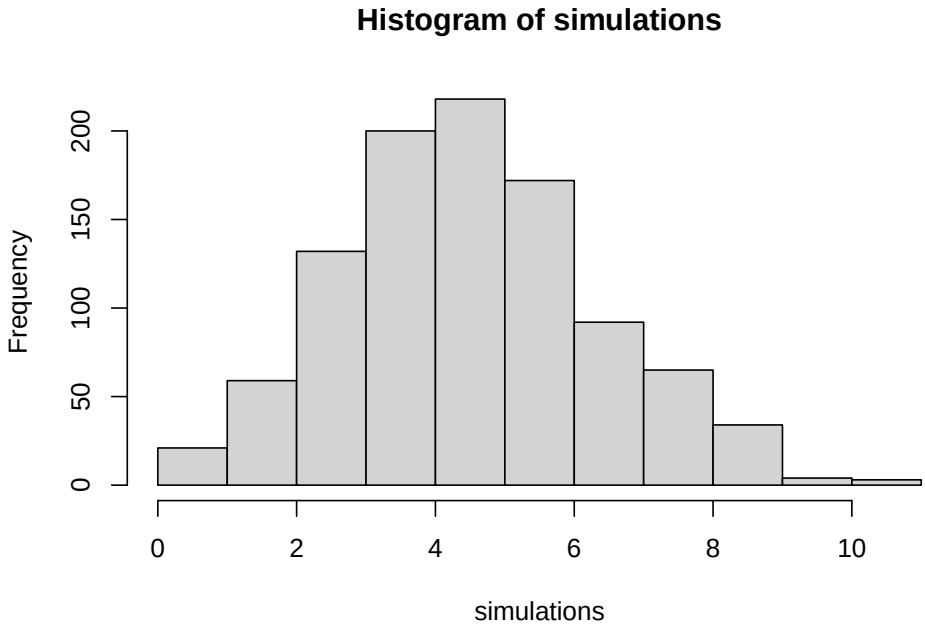


So what we see is given his season long shooting percentage, it's not out of the realm of randomness that with just 39 attempts for Palmer, he's only hit only 7. In 1000 simulations, it comes up 35 times. Is he below where he should be? Yes. Will he likely improve and soon? Unless something is very wrong,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

39.1 Cold streaks

During the Western Illinois game, the team, shooting .329 on the season from behind the arc, went 0-15 in the second half. How strange is that?

```
set.seed(1234)

simulations <- rbinom(n = 1000, size = 15, prob = .329)

hist(simulations)
```




```
table(simulations)
```

```
## simulations
##   0   1   2   3   4   5   6   7   8   9  10  11
##   5  16  59 132 200 218 172  92  65  34   4   3
```

Short answer: Really weird. If you simulate 15 threes 1000 times, sometimes you'll see them miss all of them, but only a few times – five times, in this case. Most of the time, the team won't go 0-15 even once. So going ice cold is not totally out of the realm of random chance, but it's highly unlikely.