R Course

Will Calandra

R Education

Hello! Attached in this document are my R lecture notes, which I used to teach my R course for both Hoyalytics and Georgetown Baseball. These lectures require no prerequisites and are meant to introduce people to R programming, basic statistical principles, and data ethics. Feel free to review the content below!

Lecture 1

This lecture starts with an inspirational TED Talk introducing the concept of big data, which leads to a discussion on its impact and implications for technology and society. I then take my peers through a tour of the RStudio IDE and finish the lecture with building a simple die roll simulation program.

Lecture 2

This lecture is an overview of data structures and operators in R. I made the choice to teach this before modeling so that my peers could understand the inner workings of the language before they start to build models. This way, building and debugging would be a lot easier!

Lectures 3 + 4

Before introducing my peers to basic linear modeling (simple and multiple linear regression), this two-part lecture begins with a discussion surrounding data ethics, using Cambridge Analytica's role in the 2016 election as a case study. This lecture then walks people through building and evaluating fantasy football projection models from weekly player performance averages. Data visualization techniques using ggplot are also briefly introduced.

Lecture 5

This lecture introduces the concept of user-defined functions in R, which will help my peers write cleaner code and build more advanced programs. Looping is also introduced as a concept for people to learn repeatable procedures.

Lecture 6

My personal favorite, this lecture provides an overview of statistical hypothesis testing so that my peers can understand and verify results. Visual aids using ggplot assist people in exploring distributions and understanding reasons for statistical significance.

Extra Resources and GitHub

These are resources I shared with my peers to fill the gaps not covered in lecture. I also give a brief introduction to GitHub and walk people through how to use it for uploading their files.

Lecture 1 - Intro to R/RStudio

Note: This is a summary of our first R education session for those who were absent (and those who want to review)! Please reach out to me (Will Calandra) at wcc44@georgetown.edu (mailto:wcc44@georgetown.edu) if you have any questions or concerns. Thanks!

Downloading R and RStudio

Before we begin, let's download the necessary software, which will be R and RStudio! R is the base language that we install to run on our computer, and RStudio is the IDE (Integrated Development Environment) where we will type and run our code!

There are two steps to this downloading process…

- Download R - https://www.r-project.org/ (https://www.r-project.org/)
    - This is the link to the R website. Click on "CRAN" under "Download" and pick the location closest to you!
    - Then at the top of the page, click "Download R for XX" - this depends on your operating system! Then click the "base" option, and then the "Download" link at the top of the page again!
    - If you did this right, a win.exe file should be downloaded. Open this and follow the instructions! This is a free and open license, so don't worry :)
- Download RStudio
    - Now that R is downloaded on your computer, we will download our IDE, RStudio. Navigate to the RStudio website: https://posit.co/download/rstudio-desktop/#download (https://posit.co/download/rstudio-desktop/#download)
    - Scroll and click on the "Download" button for RStudio Desktop (!!)
    - Doing this should take you to a table with the titles "OS" "Download" "Size" and "SHA-256." Click on the download link for your OS and follow the instructions with the .exe download!
    - Once completed, RStudio should be an application in your computer. Find it, put it in an accessible place, and fire it up!

If you need help with downloading the software, feel free to reach out or watch a YouTube video so you can see how we do it!

Now, to frame this lecture, watch this inspirational TED Talk about the power of data below:

TED Talk

Link: https://www.youtube.com/watch?v=8pHzROP1D-w (https://www.youtube.com/watch?v=8pHzROP1D-w)

- "Big Data is Better Data" by Kenneth Cukier, who is a journalist at the Economist and a famous speaker on technology and society
    - Please try to watch the entirety of the video! Cukier gives us an inspiring yet fair perspective on "big data," its potential, and its dangers - below are some of my thoughts on it as we think about the impact and implications of our work

Some thoughts…

- The world is full of patterns, seen and unseen, and it is our job to find trends in data that elucidate these patterns and tell a story. Information is the world's most valuable asset, and we can use it to learn and make discoveries.
- Data and processing information is not a new concept. However, the development of technologies that can harness large volumes of data has become a game-changer for every industry. R is a tool that we use to

process information and turn it into actionable insight!
- We have become clever at representing real-word problems in a computer, through code and mathematics. We have turned traditional problems into “data problems” that can be simulated in a computer to learn FAST.
- In lecture, we discussed ethical concerns, potential for government regulations on algorithms, and how tech leaders are challenged with understanding how their solutions will scale and impact society. There is an entire field of data ethics in addition to studies in responsible AI which try to tackle concerns around privacy, discrimination, and various other issues that are prevalent in the infancy of the “big data” age.
- I really enjoyed our discussion, and those of you who attended lecture asked me some really thought-provoking questions. I was challenged by these and you did a good job driving the discussion forward!

RStudio IDE Tour:

- Thanks for watching… now head back to your RStudio! We now take a tour of the “Integrated Development Environment”, or IDE, for R. REmember that an IDE is just a fancy way for saying an application that you use to write, test, and run your R code
- In the top right, you will see a glass box, which is for your “projects,” serving as a folder in R for all of your files. It is important to put all your data and files in the same directory, so it is a good idea to stay organized by creating projects
    - Let’s create a “Hoyalytics” project - click on it, click “new project,” “new directory,” “new project,” give it the name “Hoyalytics,” and save it on your computer where’d you like - all your work will be done in here!
- There are 4 panels in RStudio’s IDE. Let’s look at each:

Top left - source code

- This is where you will write the code that you share with us
- R reads the code that you type in the source panel from left to right, top to bottom, just like a book - this is important to know because order matters!
- You type in here, and click the “Run” button to run the line of code that your cursor is on - you can also highlight multiple lines of code and click “Run” to run multiple lines of code at once

Bottom left - the console

- This is actually the place where your source code is sent for R to read - you can test your code in the console to make calculations and such, but code that you type in here won’t be saved or sent to anyone! The console is useful to look at your code, objects (more on this later), data, etc. - without impacting your source code!

Top right - the environment

- Here is where your “saved” values, data, objects, etc. are stored! When you run your code and save it in an object, that object and its characteristics will be displayed in here
- A handy tool to use is the “Import Dataset” capability where RStudio can help you import some data for you to work with!
- Clicking on the data and dataframes (more on this later) will allow you to view them, which can be helpful and easier than typing a “view()” function

Bottom right - files, plots, packages, and help

- Your directory, where you create and import files, will show up under the “files” tab - there is some functionality here to play around with, but think of this as just a useful tab to keep track of your files and know where things are in your computer

- As you run R code with visuals, they will display as they are produced in the “plots” tab - you can click the “zoom” button to blow them up full screen, and you can even “export” them as PDFs and such
- The “packages” tab includes packages (or libraries) in R, with the checked packages representing the packages installed on your computer - more on this in later lessons, but packages are libraries that are full of functions and capabilities that we can use (for free!) - clever people have developed packages that do neat stuff in R
- The “help” tab has some useful links with information that you can check out - we find the search bar in this tab to be the most useful for looking up certain packages or functions we may have questions about - this will give us what is called “documentation,” which details what functions do and tell us how to properly write code for these functions

Time to get started!

- So that covers the basics of RStudio… now we can create a file and get started!

Basic Die Roll Simulation Example

```
# Counter Example
```

- Starting with line 1, we have a comment, which is a way for the user to enter information about what they are doing without the line being interpreted by the compiler. In R, using the # symbol at the beginning of a line creates a comment for the entire line, where you can insert text referring to what the subsequent code is expected to do. In our example, we use #Counter Example to indicate to us that the following lines are meant to create a counter.

```
X <- 5
```

- In line 2, we define our first object, X, and assign it the value of 5. This is an integer object, meaning that the value 5 is stored in the name “X”, and we can reference it in the future by calling the object “X”. The “arrow” symbol <- is called the assignment operator and indicates that X is being assigned the value of 5. It can be created by using the less than “<” operator followed by the hyphen “-” symbol.

```
X <- X + 1
```

- In line 3, we are reassigning the object X with a new value. This line cannot be run unless the object has already been initialized (i.e. assigned a value) as we did in line 2. This will re-assign X to an integer value 1 greater than the value X is currently assigned. If you were to run lines 2 and 3, X would finish with a value of 6, but if you were to run line 3 again, X would now be assigned 7. This seems like weird behavior in R, and it has to do with your computer’s memory. for now, just know that it is good practice to define objects carefully and make sure that you only run each line of code once while you are doing your analysis.
- We are now going to set up the die roll simulation:

```
# Die roll
S <- 1:6
```

- The code in line 6 (S <- 1:6) aims to create a sequence of integer values from 1 to 6, inclusive, and assigns the list to the variable S. The 6 values that you see represent the 6 possible die values.

```
# Use function (argument1 = x, argument2 = y, ...)
S <- seq(from = 1, to = 6, by = 1)
```

- The code in line 9 also aims to create a sequence of integer values from 1 to 6, inclusive. This is our first example of a function, which can be formally defined as a set of instructions known by the compiler which are intended to achieve a specific purpose. In R, there are two main ways to look up what a function does. The first method is to go to the help tab (described in the previous section) and type in the name of the function. The other method is to type a question mark followed by the name of the function into the console (ex. ?seq()). In this case, the seq() function will be creating a sequence of integer values. “From,” “to,” and “by” are known as arguments (or parameters), and they will influence the process of the function. In the case of the seq() function, the “from” argument indicates the starting number of the sequence, the “to” argument indicates the concluding number of the sequence, and the “by” argument indicates the increment of the sequence. The single equals sign (“=”) is another assignment operator and must be used within the argument of the function to assign the correct value to the function’s parameter.

```
P <- c(1/6,1/6,1/6,1/6,1/6,1/6)
```

- Line 10 is creating a vector of 6 numbers and assigning it to the named variable “P.” Here we also see another function, called c(), which stands for concatenate. There are no specific names for concatenate, so we just put in the values of interest. We are using c() to create a vector object of 6 numbers which represent the probability of a six-sided fair die roll landing on each value. Why did we create this? Well, vectors are HUGE in R. Think about it: S is a sequence of 6 numbers (1, 2, 3, 4, 5, 6) with a specific order, so S is a vector. We create the corresponding vector P that corresponds with each entry of S as its probability of occurrence. We will combine these vectors in a clever way to get a randomized output of die values.

```
# Randomization set.seed() function
set.seed(27)
```

- Line 13 is a set.seed() function that is important to use whenever there is randomization. In the set.seed() function, R makes a random sequence of numbers, and keeps this randomization constant when you re-run code. This is important when we want to reproduce model results from randomized experiments. The number within the set.seed() function represents the “seed,” or sequence of random numbers. This number is arbitrary, but we’ll choose 27 here.

```
# Sample function (uses randomization above, run line 13 BEFORE 16 every time)
Samps <- sample(S, 23, replace = TRUE, prob = P)
```

- Line 16 is an object called “Samps,” which is where we will save the values of our sampling from the S sequence. Since S is a sequence from 1 to 6, we simulate the value of a die roll in our computer by randomly sampling from the S object. The “Samps” object is defined by a sample() function, which takes the corresponding parameters (sample, size, replacement, and probability). We can see that we cleverly use our objects to represent parameters, which is good practice, because we do not have to change our code each time a value in our objects is updated. Walking through this line of code, the sample() is the function, the “S” represents the object being sampled from (1 to 6), “23” represents the size of the sample (take 23 samples), “replace = TRUE” means that we are sampling with replacement, and the “prob = P” is the corresponding probability vector for each value of 1 to 6 (equal, ⅙ for each value). Run this and see what happens! Notice how we run the set.seed() function each time we run the Samps line. This is important

because the sample() function is random, so we want to reiterate over the 27th seed each time for consistency.

- Note how we didn't put the parameter names for S and 23. It is not required by R that you use the parameter names and the assignment operator "=" to assign their values, but it might help you out early on if you practice writing code in this manner (to better see what you are doing)!

```
print(Samps)
```

```
##  [1] 1 2 1 3 3 4 2 2 2 3 5 1 5 6 1 6 1 2 5 2 2 1 5
```

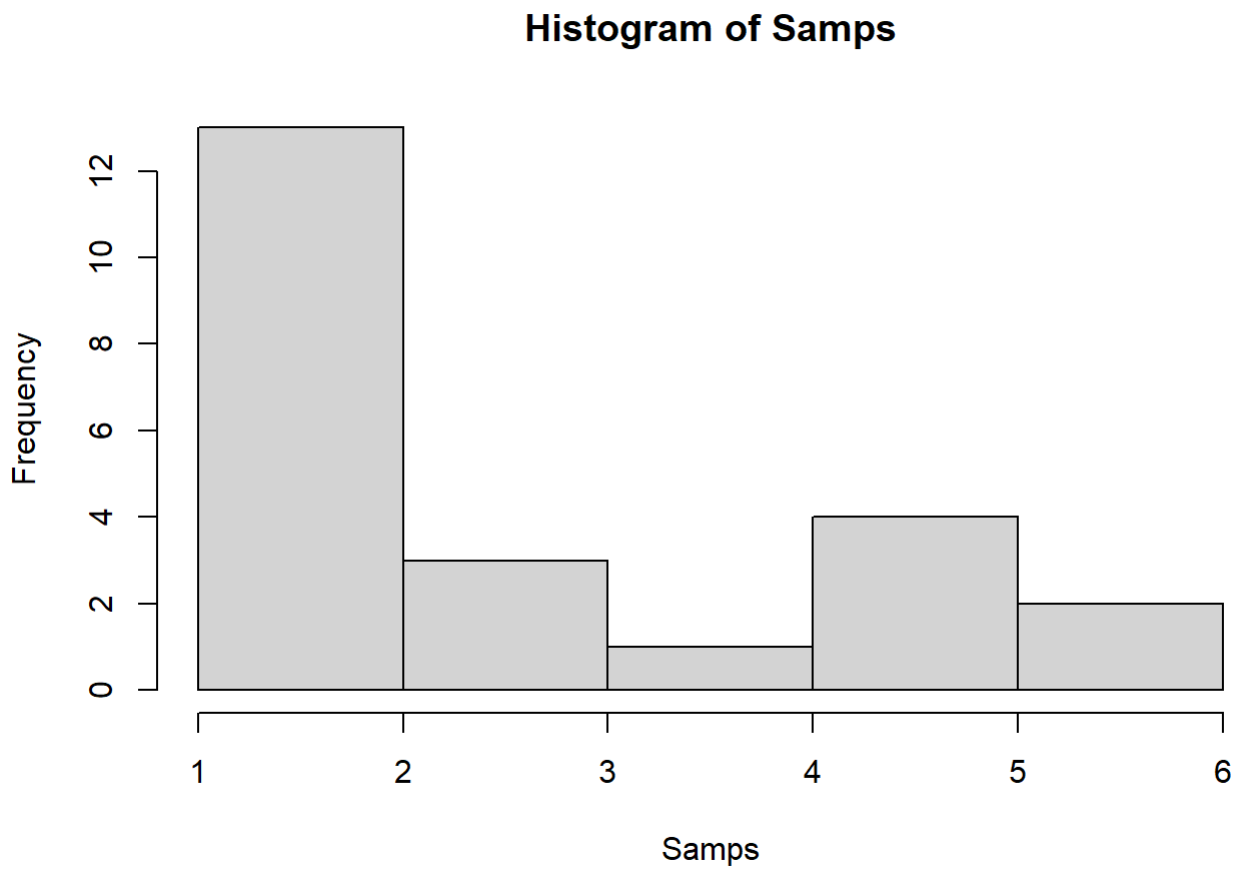
- A useful print() function call on line 17 lets us see some values of Samps, but not all of them. We'll see how we can do this later. Often, it's good practice to type print() in the console instead of your source code.

```
summary(Samps)
```

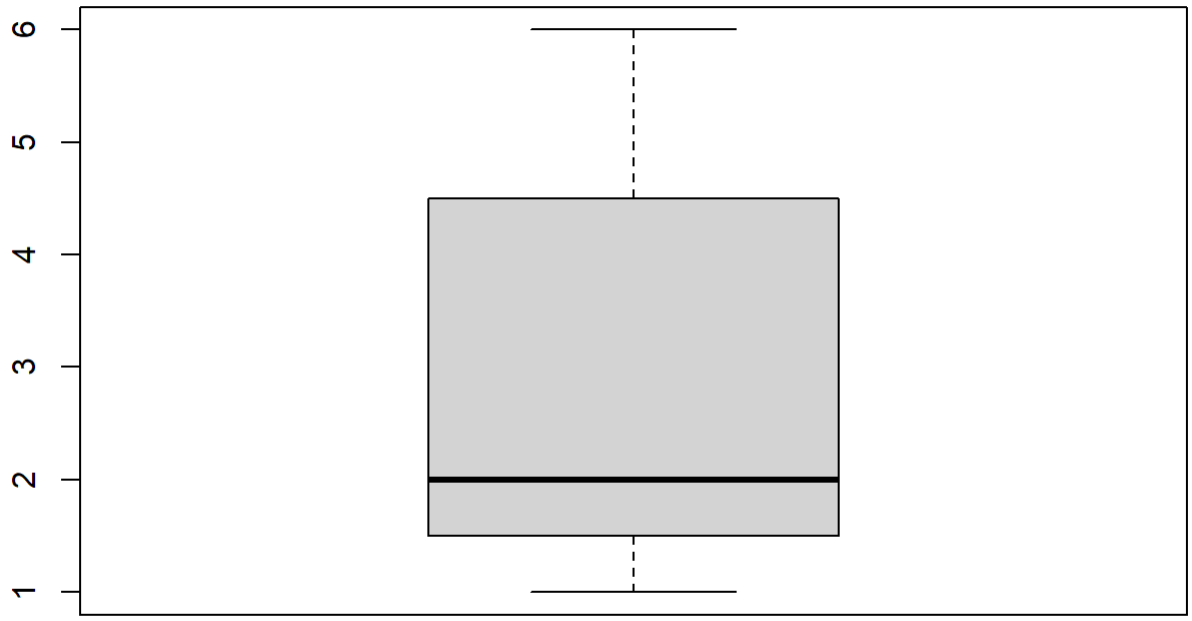
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.000   1.500   2.000   2.826   4.500   6.000
```

- Line 18 is a handy way to look at our results using the summary() function as it gives us the 5 number summary of "Samps," or our samples/die rolls - pretty neat!

```
# Visualizations!
hist(Samps)
```



```
boxplot(Samps)
```



- Our first visualizations in R! Notice the distribution of our die rolls: it's not really the theoretical uniform distribution we'd expect! That's because our sample size of 23 is small. Here's an exercise: try changing 23 to a much larger number. Then these visuals should be more appealing!

```
# Change data type and save
Samps <- as.data.frame(Samps)
```

- Notice how it is difficult for us to access the results of "Samps." This is because it is stored in R as a list. We want to be able to take a closer look and do more with our die rolls, so we will change the class of our object "Samps" from a "list" to what we call a "dataframe." Dataframes are practically Excel sheets; they are vectors/matrices representing columns and rows. The as.data.frame() function changes "Samps" into a dataframe… we will overwrite the Samps object to save this as "Samps" using the as.data.frame() function in line 25.

```
summary(Samps)
```

```
##      Samps
##  Min.   :1.000
##  1st Qu.:1.500
##  Median :2.000
##  Mean   :2.826
##  3rd Qu.:4.500
##  Max.   :6.000
```

- Line 27 offers a summary of Samps, proving that it is the same as our previous Samps object, just in a different class (form). Look at the top right in your global environment; you are able to click on Samps and access each die roll! Notice how it saved as a column name “Samps” with the values of our 23 die rolls. Congrats, you just simulated die rolls and got an output!

Notes

Thanks for reading! If you missed our lecture, trust me — watching, doing, and learning is more fun than reading, especially when it comes to learning how to code! This document should suffice for those of you who were absent, but it should not be your primary way of learning R… come to the lectures and use these for review! :)

Lecture 2 - Data Structures

Why Are We Learning Data Structures?

- R is convenient and R is smart, but sometimes R can be difficult. Therefore, you have to understand how functions work on different data structures to make R work for you instead of against you!
- Data cleaning - it’s an essential part of the analytics workflow. Real world data tends to be messy, so it is imperative that you master these skills before you get into deep and complex programming!
- Syntax - the syntax R uses works well with its data structures, because arguments to functions are pulled from a certain “class” (which will be discussed later)!
- Speed - the last thing you should consider for now, but still ultimately important. Working with faster data structures makes larger computations easier and improves application performance!

Classes

Logical

Overview

- T for TRUE and F for FALSE (both work, we prefer shorter for style)
    - Notice that this must be capitalized
    - Notice syntax highlighting (nice to use an IDE!)
- Used frequently to “turn on” or “turn off” options in functions

```
# Load R dataset
data <- mtcars

### Logical - either evaluates to true or false, is foundational for building computer logic
## T or F
# Use $ to access vectors
data$mpg == T
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
data$mpg > 15
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
## [13]  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
## [25]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
```

Logical operators

- ==
  - The most important operator - checks for equality, not assignment
  - Accidentally assigning with = when you meant == is one of the most common syntactical errors
- !=
  - Not equal to - you can get creative with this one! A lot of binary uses here
- “|”
  - The OR operator for conditionals (without the quotes)!
- &
  - The AND operator for conditionals!
- xor(x, y)
  - The OR operator but don’t include both!
- Click this link below for a nice summary graphic:
  - https://r4ds.had.co.nz/transform.html?q=near#logical-operators (https://r4ds.had.co.nz/transform.html?q=near#logical-operators)

```
## Operators - a bunch of different operators we use for comparison!
data$is0 <- data$vs == 0
data$is_not_0 <- data$vs != 0
data$mpg_15_17 <- data$mpg > 15 & data$mpg < 17
data$cyl_6_or_8 <- data$cyl == 6 | data$cyl == 8
data$xorTest <- xor(data$mpg_15_17 == T, data$cyl_6_or_8 == T) # function for xor (exactly one o
f these cases is true)
```

Numeric

Overview

- Just numbers (that’s it!)

Double and integer

- There are some special cases for numbers - doubles have decimal places, integers don't
- R will usually convert integers to doubles for you as needed (ha, C++)
- Why use integers if we have doubles? For memory: integers are fast and hold less data

Mathematical Operators

- +, -, /, *
  - Addition, subtraction, division, multiplication
- ^ or **
  - Either works for exponential!
- %/% and %%
  - Integer division (5 %/% 2 is 2)
  - Remainders (5 %% 2 is 1)

Relational Operators

- <, >, >=, <=
  - This is not assignment!
  - Less than, greater than, and the or equal to cases

```
## Mathematical Operators
# A bunch of different symbols which represent math operations!
data$cyl
```

```
##  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

```
data$cyl * 2
```

```
##  [1] 12 12  8 12 16 12 16  8  8 12 12 16 16 16 16 16 16  8  8  8  8 16 16 16 16
## [26]  8  8  8 16 12 16  8
```

```
data$cyl + 2
```

```
##  [1]  8  8  6  8 10  8 10  6  6  8  8 10 10 10 10 10 10  6  6  6  6 10 10 10 10
## [26]  6  6  6 10  8 10  6
```

```
data$cyl - 2
```

```
##  [1] 4 4 2 4 6 4 6 2 2 4 4 6 6 6 6 6 6 2 2 2 2 6 6 6 6 2 2 2 6 4 6 2
```

```
data$cyl / 2
```

```
##  [1] 3 3 2 3 4 3 4 2 2 3 3 4 4 4 4 4 4 2 2 2 2 4 4 4 4 2 2 2 4 3 4 2
```

```
data$cyl ^ 2
```

```
##  [1] 36 36 16 36 64 36 64 16 16 36 36 64 64 64 64 64 16 16 16 16 64 64 64 64
## [26] 16 16 16 64 36 64 16
```

```
data$cyl ** 2
```

```
##  [1] 36 36 16 36 64 36 64 16 16 36 36 64 64 64 64 64 16 16 16 16 64 64 64 64
## [26] 16 16 16 64 36 64 16
```

```
# Integer division - floors extra decimals
data$cyl %/% 2
```

```
##  [1] 3 3 2 3 4 3 4 2 2 3 3 4 4 4 4 4 4 2 2 2 2 4 4 4 4 2 2 2 4 3 4 2
```

```
# Modulus - remainder after division - handy to break up integers into pieces in calculations +
build logic
data$cyl %% 3
```

```
##  [1] 0 0 1 0 2 0 2 1 1 0 0 2 2 2 2 2 2 1 1 1 1 2 2 2 2 1 1 1 2 0 2 1
```

```
## Relational Operators (greater than, less than, equal to cases, etc)
data$mpg > 17.3
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
## [25]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
```

```
data$mpg >= 17.3
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
## [13]  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
## [25]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
```

Character

Brief Overview

- String
    - String transformations - regex! There are a ton of functions out there to help out (gsub() is a common one)
- Look out for “1”, “T”, etc.
    - They can be sneaky characters when you import data!
- Lots of data cleaning is getting data out of character form and into numeric form

```
### Character
# Load R dataset
data2 <- iris
# Click on "data2" and hover over the column headers - R gives you the data type (or class)!

# This is a "factor" data type - useful for grouping qualitative variables in analyses
print(data2$Species)
```

```
##   [1] setosa     setosa     setosa     setosa     setosa     setosa
##   [7] setosa     setosa     setosa     setosa     setosa     setosa
##  [13] setosa     setosa     setosa     setosa     setosa     setosa
##  [19] setosa     setosa     setosa     setosa     setosa     setosa
##  [25] setosa     setosa     setosa     setosa     setosa     setosa
##  [31] setosa     setosa     setosa     setosa     setosa     setosa
##  [37] setosa     setosa     setosa     setosa     setosa     setosa
##  [43] setosa     setosa     setosa     setosa     setosa     setosa
##  [49] setosa     setosa     versicolor versicolor versicolor versicolor
##  [55] versicolor versicolor versicolor versicolor versicolor versicolor
##  [61] versicolor versicolor versicolor versicolor versicolor versicolor
##  [67] versicolor versicolor versicolor versicolor versicolor versicolor
##  [73] versicolor versicolor versicolor versicolor versicolor versicolor
##  [79] versicolor versicolor versicolor versicolor versicolor versicolor
##  [85] versicolor versicolor versicolor versicolor versicolor versicolor
##  [91] versicolor versicolor versicolor versicolor versicolor versicolor
##  [97] versicolor versicolor versicolor versicolor virginica  virginica
## [103] virginica  virginica  virginica  virginica  virginica  virginica
## [109] virginica  virginica  virginica  virginica  virginica  virginica
## [115] virginica  virginica  virginica  virginica  virginica  virginica
## [121] virginica  virginica  virginica  virginica  virginica  virginica
## [127] virginica  virginica  virginica  virginica  virginica  virginica
## [133] virginica  virginica  virginica  virginica  virginica  virginica
## [139] virginica  virginica  virginica  virginica  virginica  virginica
## [145] virginica  virginica  virginica  virginica  virginica  virginica
## Levels: setosa versicolor virginica
```

```
# How to overwrite the Species column to change it from a factor data type to a character data t
ype
data2$Species <- as.character(data2$Species)

# Notice the "" marks - character coercion worked!
print(data2$Species)
```

```
##   [1] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##   [6] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [11] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [16] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [21] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [26] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [31] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [36] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [41] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [46] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [51] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [56] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [61] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [66] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [71] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [76] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [81] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [86] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [91] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [96] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
## [101] "virginica"  "virginica"  "virginica"  "virginica"  "virginica"
## [106] "virginica"  "virginica"  "virginica"  "virginica"  "virginica"
## [111] "virginica"  "virginica"  "virginica"  "virginica"  "virginica"
## [116] "virginica"  "virginica"  "virginica"  "virginica"  "virginica"
## [121] "virginica"  "virginica"  "virginica"  "virginica"  "virginica"
## [126] "virginica"  "virginica"  "virginica"  "virginica"  "virginica"
## [131] "virginica"  "virginica"  "virginica"  "virginica"  "virginica"
## [136] "virginica"  "virginica"  "virginica"  "virginica"  "virginica"
## [141] "virginica"  "virginica"  "virginica"  "virginica"  "virginica"
## [146] "virginica"  "virginica"  "virginica"  "virginica"  "virginica"
```

```
# gsub() in action - overwrite virginica to rose
data2$Species_Change <- gsub(pattern = "virginica",
                             replacement = "rose",
                             x = data2$Species)
data2$Species_Change
```

```
##   [1] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##   [6] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [11] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [16] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [21] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [26] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [31] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [36] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [41] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [46] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
##  [51] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [56] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [61] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [66] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [71] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [76] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [81] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [86] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [91] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
##  [96] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
## [101] "rose"       "rose"       "rose"       "rose"       "rose"
## [106] "rose"       "rose"       "rose"       "rose"       "rose"
## [111] "rose"       "rose"       "rose"       "rose"       "rose"
## [116] "rose"       "rose"       "rose"       "rose"       "rose"
## [121] "rose"       "rose"       "rose"       "rose"       "rose"
## [126] "rose"       "rose"       "rose"       "rose"       "rose"
## [131] "rose"       "rose"       "rose"       "rose"       "rose"
## [136] "rose"       "rose"       "rose"       "rose"       "rose"
## [141] "rose"       "rose"       "rose"       "rose"       "rose"
## [146] "rose"       "rose"       "rose"       "rose"       "rose"
```

Special Cases

- NULL
  - Don’t worry about this too much, it just means empty data. R doesn’t like this, so you may have to change these to NA
  - Use is.null() to determine if something is NULL
  - Note NULL is in caps
- NA
  - The standard way of representing missing data
  - Note that NA == NA is not always true (interesting class case)
  - So use is.na() to determine whether data is NA
  - Note NA is in caps
- NaN, Inf
  - You violated some math rules (i.e. divide by 0)

```
### Special cases
data$hp <- NULL # removes column
data$qsec <- NA # makes the column empty
5 / 0 # infinity
```

```
## [1] Inf
```

```
0 / 0 # not a number
```

```
## [1] NaN
```

(Atomic) Vectors

Overview

- R is all about vectors
  - It is the fundamental data structure of R
- One class
  - class() - to check a class of a vector
- Any length, no negatives
  - length() - to check the length of a vector
- Let's learn our first few functions, without formal structure yet
  - c(), seq(), rep()
- Everything is a vector in R. They sound scary from math class, but they are foundational for data analysis!

Indexing

- Use vec[] to call index of vectors
- Index search

Filtering

- Use vec[] to filter vectors

Functions and Operators are Nearly All Vectorized!

- This is the central concept behind why R is such a nice language to work with for data science
- Makes your life a lot easier
- No looping like other languages

Coercion and Messy Vectors

- as. can help you coerce to the class you want!
- You may lose data by doing this, so watch for errors or warnings
- Some messy vectors:
  - x <- c(2, T)
  - x <- c(2, "hello")
  - x <- c("hello", 3.5)

Recycling

- Beware of recycling
  - Recycling vectors of length one is obvious (adding 1 to every entry)
  - Recycling longer vectors gets messy
    - We did some of this in our first lecture - remember how R reads! Top to bottom, and you can overwrite in your script - but be careful in tracking your memory!

Some Statistical Functions to Call on Vectors

- mean(), sum(), var()
- Implicit coercion from logical to numeric

```
### Vectors
class(data) # class() function checks the data type/class --- dataframes are foundational to R - they are like excel sheets! Rows and columns of data
```

```
## [1] "data.frame"
```

```
length(data) # 15 columns in this dataframe
```

```
## [1] 15
```

```
class(data2) # also a dataframe
```

```
## [1] "data.frame"
```

```
x <- c("Hey", "What's", "Up") # spawn a vector of three entries
class(x) # check it's type
```

```
## [1] "character"
```

```
# Use seq, rep functions to generate a vector of 150 entries (the y vector repeated 50 times)
y <- seq(from = 1, to = 5)
z <- rep(y, 30)

z[30] # find the 30th entry in the z vector
```

```
## [1] 5
```

```
z_slice <- z[20:40] # subset and create a new z_slice vector (comes from the 20th through 40th (inclusive) entries of z)
z_slice <- as.data.frame(z_slice) # coerces the vector into a dataframe so we can see it!
z_slice
```

```
##    z_slice
## 1        5
## 2        1
## 3        2
## 4        3
## 5        4
## 6        5
## 7        1
## 8        2
## 9        3
## 10       4
## 11       5
## 12       1
## 13       2
## 14       3
## 15       4
## 16       5
## 17       1
## 18       2
## 19       3
## 20       4
## 21       5
```

```
t <- rep(x, 40) # same process of rep, just on x vector, which is character data

# Difference?
x <- 1
y <- c("1")
class(x)
```

```
## [1] "numeric"
```

```
class(y)
```

```
## [1] "character"
```

```
# The data types are different!

# Messy
x <- c(2, T)
x <- c(2, "hello")
t <- c("hello", 3.5)
# These are really bad to work with, different data types in the same vector
# Use filters to get one data type per column, and then coerce to the proper class

# Fix
x <- c(2, "2")
x <- as.numeric(x) # R is smart here - can read that "2" as a number, even though it isn't
t <- as.numeric(t) # Not so much here - can't convert characters to numbers
```

```
## Warning: NAs introduced by coercion
```

```
is.na(t) # useful function to find NAs in vectors
```

```
## [1]  TRUE FALSE
```

```
# Vector Functions
mean(z)
```

```
## [1] 3
```

```
sum(z)
```

```
## [1] 450
```

```
var(z)
```

```
## [1] 2.013423
```

This here below is for your reference (will come in handy for lectures 3 + 4)

Dataframes/Tibbles

Overview

- A dataframe is just a bunch of vectors lined up next to one another in columns. The gold standard package for operating on dataframes is called dplyr. It is written and maintained by Hadley Wickham, Chief Data Scientist at RStudio.
- Install the dplyr package:

```
install.packages("dplyr")
library(dplyr)
```

- Read the gold standard textbook on dplyr below:
    - https://r4ds.had.co.nz/transform.html (https://r4ds.had.co.nz/transform.html)

dplyr Functions

- Syntax: data first, no need to repeat df name! We'll find practical uses in our next lecture…
- mutate()
- filter()
- select()
- arrange()
- summarize()
- group_by()

Lectures 3 + 4 - Basic Linear Modeling

Netflix Clip - The Great Hack

- We will watch a clip from the documentary "The Great Hack," which gives us a look behind the scenes of Cambridge Analytica's role in the 2016 election.
- Their strategy was to gather data from prospective voters' Facebook profiles (their likes, comments, friends, groups, etc.) and find out what would persuade them to vote for their candidate. They were quite brilliant in how they targeted a select portion ("the persuadables") of the population to flip for their candidate. However, their methods were a huge breach of data privacy and ethical standards. More on this below.
- People had no idea that they were a part of Cambridge Analytica's experiment, as the company psychologically profiled people and sent back targeted ads to generate a reaction that they hoped would translate into a vote for their candidate. These ads didn't have to be truthful, just engaging, which can be dangerous and manipulative. The big issue is that they tapped into all of people's data and experimented on them without their consent.
- It doesn't matter if Cambridge Analytica was working for the left or right - this is a landmark breach of ethics that is worth studying. There are entire fields around ethics and "human-centered" technology that are fascinating - I highly recommend you do some research to learn more. People like Tristan Harris, Jaron Lanier, Cathy O'Neil, Joy Buolamwini (founder of Algorithmic Justice League - that sticker on my computer) are leading the charge to police the usage of algorithms and raise awareness as to how they are impacting society. Look 'em up!
- Why am I showing you this? Well, as you learn more about R and data, it's important to recognize that these tools are powerful (remember the TED talk), and algorithms in practice can be very influential. As such, with great power comes great responsibility. Down the line, remember the goals and assumptions you make as you build models and algorithms, and understand the potential effects your algorithm can have in practice, both good and bad. Often, we find intention and consequence don't match up (and I think the social media giants are realizing this now)!

Basic Linear Modeling

In this lecture, we will bring together our learning of basic R and data structures to evaluate the performance of ESPN Fantasy Football projections. We will import a real-world dataset of 2019 ESPN projection data from the web, manipulate the dataset to analyze projections of interest, and generate some basic plots and summary statistics to interpret our results. We will then build our own model to replicate the fantasy football scoring "equation" and compare our results with the ESPN projections.

Import Data

First, we will import the dataset from the fantasy football pros website: https://www.fantasyfootballdatapros.com/csv_files (https://www.fantasyfootballdatapros.com/csv_files)

The .csv download is also in our shared Google Drive. In order to import this dataset into R, in the top right of the RStudio IDE, click on the "import dataset" dropdown and select "from Excel" to load the file into R with ease.

Notice that a dialogue box comes up and gives you an error. This is because the file is not an Excel file but rather a .csv file. No need to panic, we just want to use the code given in the bottom right as a "skeleton" for importing our dataset. Copy the code below to your clipboard - it should look something like this:

```
#library(readxl)
#X2019FantasyESPN <- read_excel("Your computer path/2019FantasyESPN.csv")
#View(X2019FantasyESPN)
```

Notice that the code imports the library(readxl), saves the file as a "X2019FantasyESPN" object by calling the read_excel() function, and then calls the view() function so you can see how the data is loaded in R. In our case, we will use this structure to import the data, but we will change the functions. We will also get rid of the view() function, because it is not useful in scripting code (it will coerce you to view the file every time you run it - we only care about seeing the data the first time after import).

To read .csv files, we use this code:

```
# Load data
library(utils)
#ESPN_Proj <- read.csv("Your computer path/2019FantasyESPN.csv")
```

Notice how we used the library(utils) and the read.csv() function to load this dataset into R. Also notice the full file path of the .csv file is in my function; this is actually bad practice. An improved version of the code would not have the full file path. In order to achieve this, we would move the file to the same folder as this .R file, and run the code below:

```
# Load data
ESPN_Proj <- read.csv("2019FantasyESPN.csv")
```

The hashtag in front of the ESPN_Proj above is removed so things compile. This is good code! The file should have shown up in your global environment panel in the top right - click on it and see the dataframe appear! Neat, right?

Visualize data

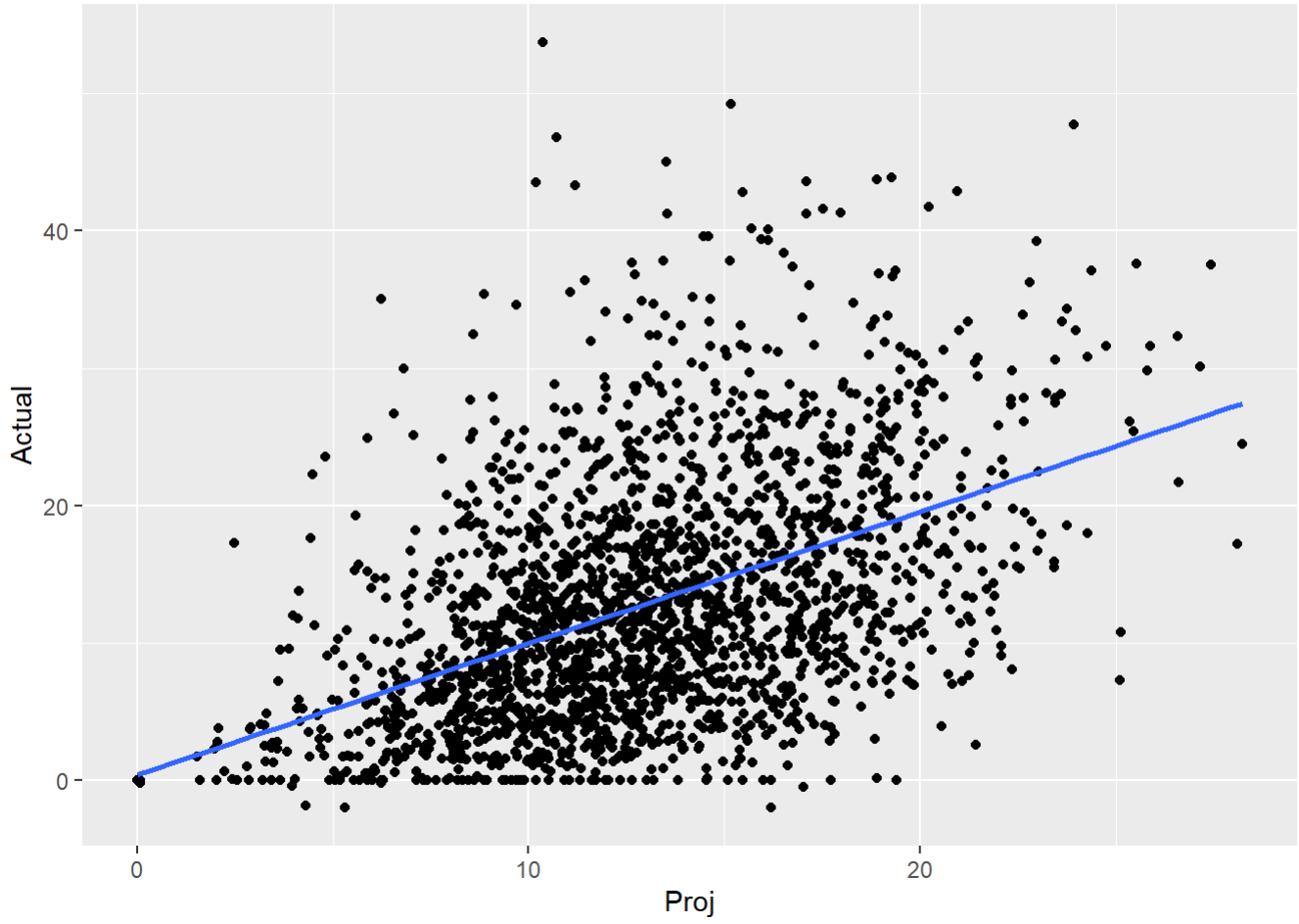
We will now conduct an exploratory analysis of the data we have. Let's use the library(ggplot2) package to visualize the dataset. If you click on the ESPN_Proj you'll notice we have a "Proj" and "Actual" column. This is ESPN's weekly fantasy projections versus the actual output. We will now build a scatter plot to see the relationship between these two variables. Is ESPN good at projecting players each week? If this is the case, what would we expect to see? Code below:

```
# Visualize performance
library(ggplot2)
ggplot(ESPN_Proj, aes(x=Proj,y=Actual)) + geom_point() + geom_smooth(method = lm, se=FALSE)
```

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

```
## Warning: Removed 154 rows containing non-finite values (`stat_smooth()`).
```

```
## Warning: Removed 154 rows containing missing values (`geom_point()`).
```



If ESPN is good at projecting players, we would expect to see a one-to-one relationship between these variables (a line with slope at 1, following the x=y line).

Let’s walk through the code. Notice that we call the ggplot() function. It accepts many parameters (which can be read in documentation), but for this analysis, we have the dataset (ESPN_Proj), a comma, and then what is called the aes(), or aesthetic. This is where you tell ggplot which variables you want to visualize. For us, we put the Proj variable on the x axis, and the Actual variable on the y axis. We then add more capability to ggplot() by adding the “+” operator, and then call the geom_point() function to tell ggplot() to plot the points of our data (no need to put anything in the parentheses, we already told ggplot what to plot in the aes()). To then draw the line representing the scatter plot’s linear relationship, we add the geom_smooth() capability, the method is a lm (linear model), and the se is false (we don’t display standard error here). Boom! Visual done.

Visually, ESPN’s performance doesn’t look bad - the slope is positive, and it seems like the data follows an upward trend in actual results v projected points. Notice how there is an error message here, as 154 rows were removed from our visualization. This is because there are some NA values in these columns; we can scroll through the ESPN_Proj to see these values in the “Actual” column. This represents some data loss in our dataset. A good test for missing values in your data can be done by typing this in the console and hitting enter:

```
# Count NA values
sum(is.na(ESPN_Proj$Actual))
```

```
## [1] 154
```

Notice that “154” displays in the console (a good refresher from our data structures lecture)! We will continue with this analysis knowing that there will be some data points that were not captured. Since there are only 154 points missing and we have 2577 observations, we will continue the analysis.

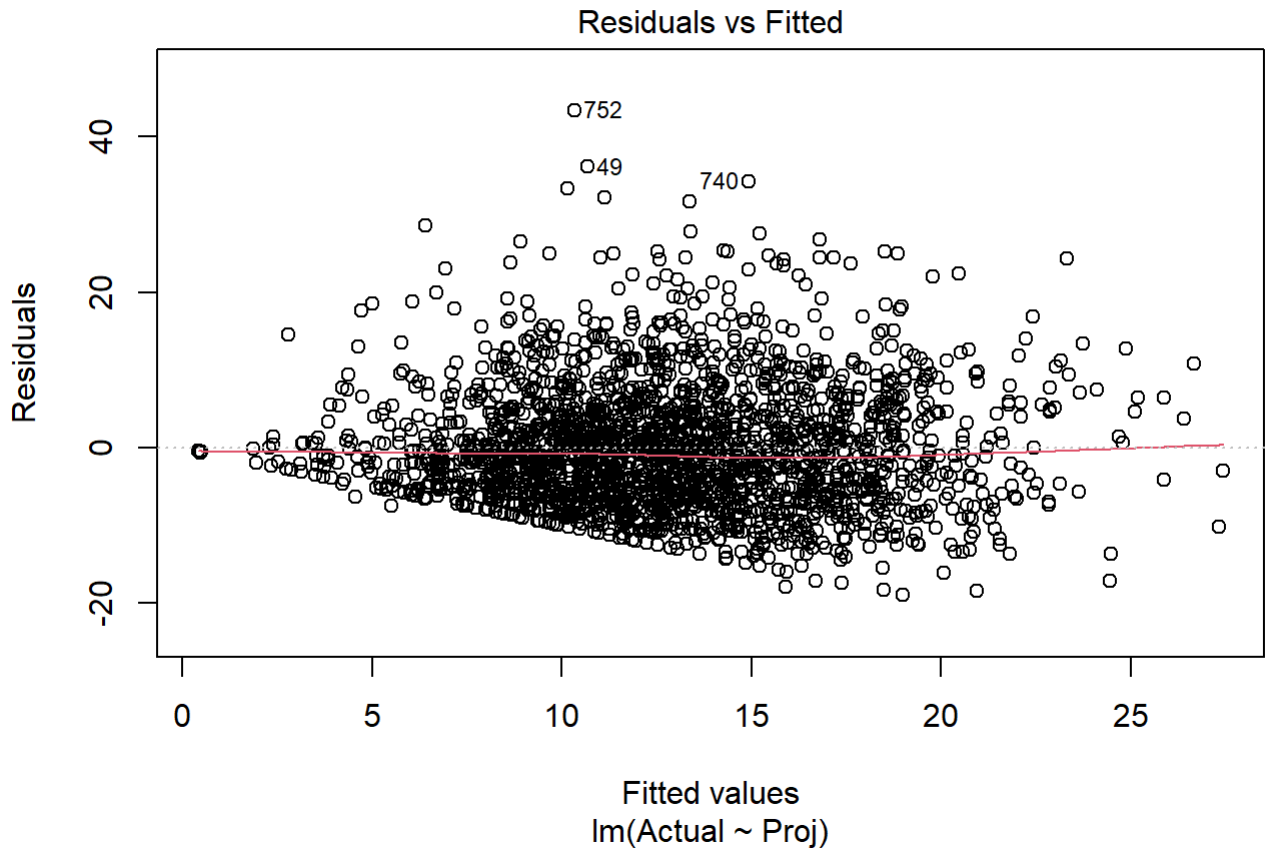
Linear model

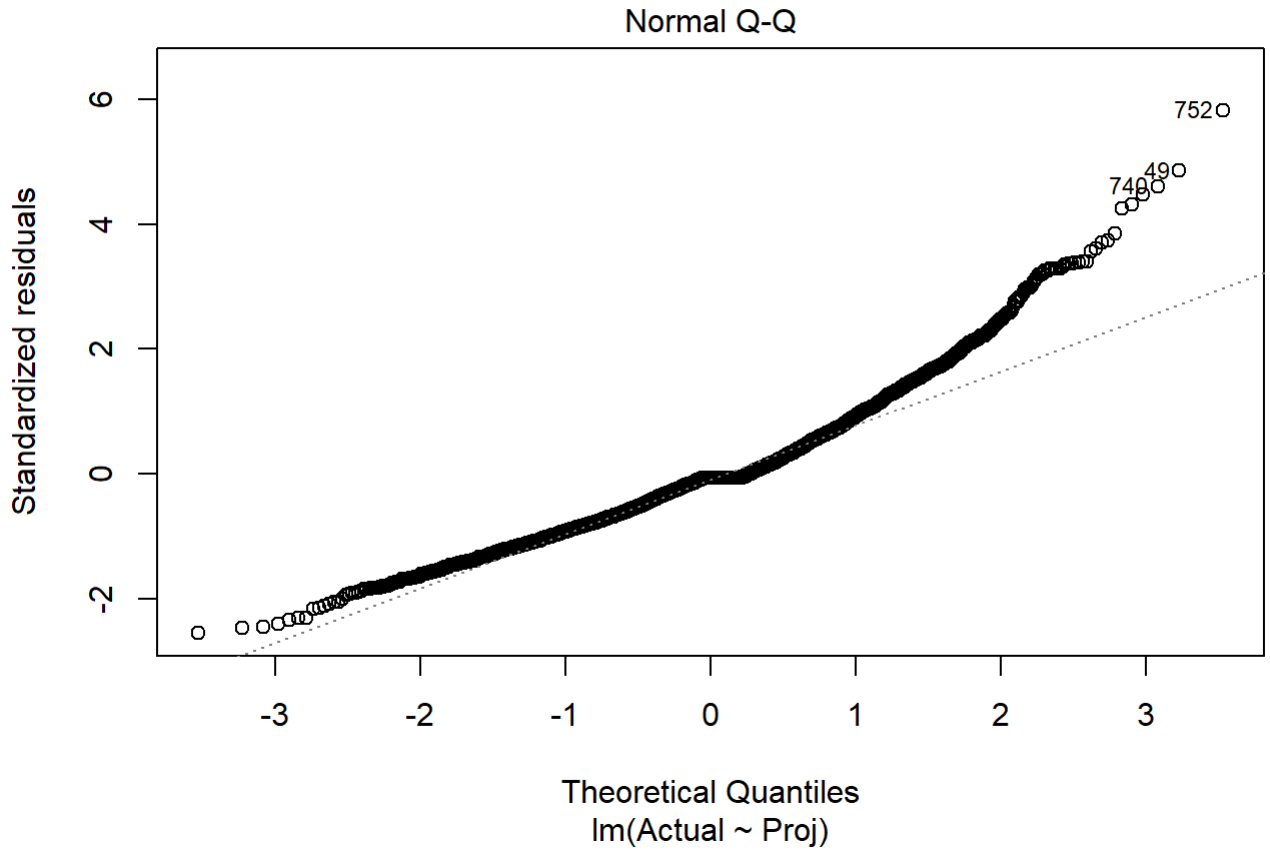
We will now fit a linear model on the Actual v Projected results to quantify what we see from this visualization. Let’s run these lines of code to see what we get!

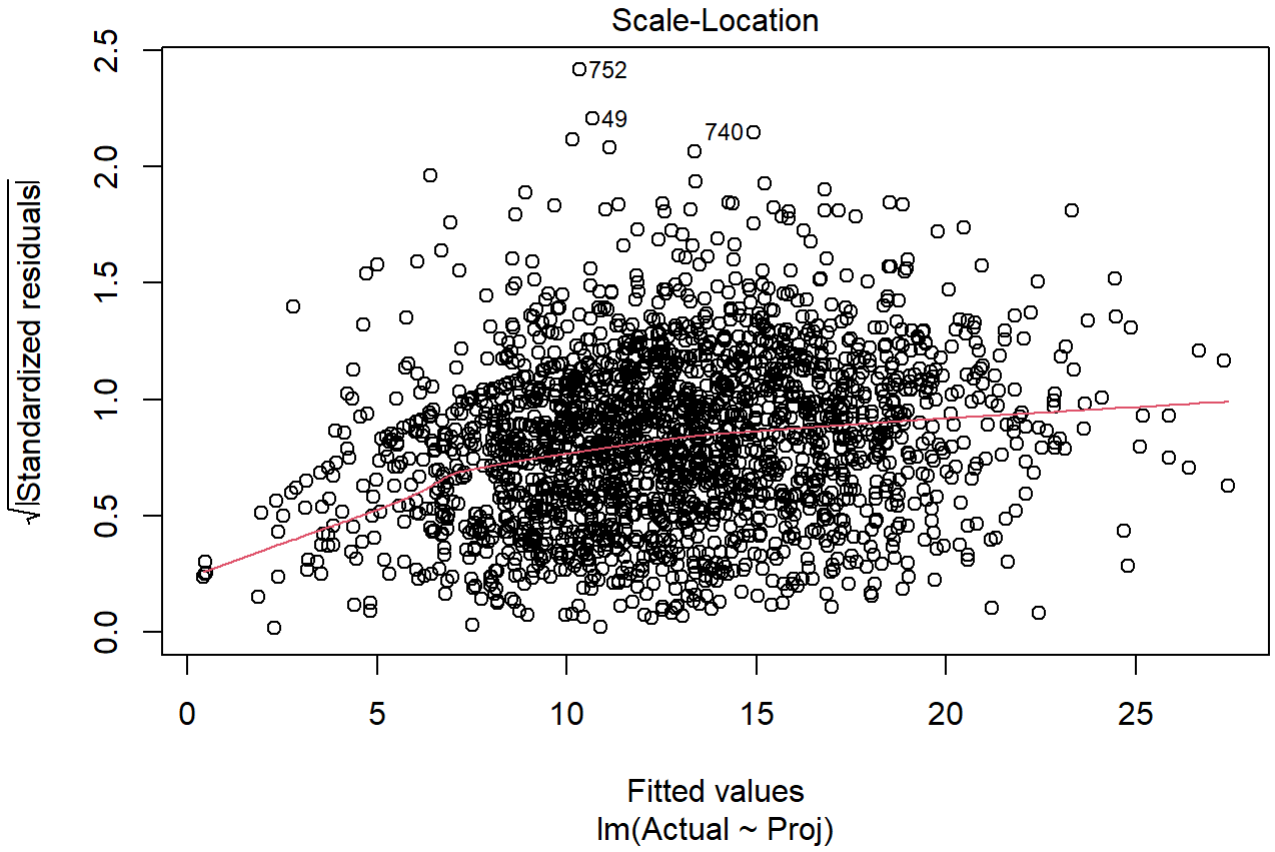
```
# Linear model on performance
Lm <- lm(data = ESPN_Proj, Actual ~ Proj)
summary(Lm)
```

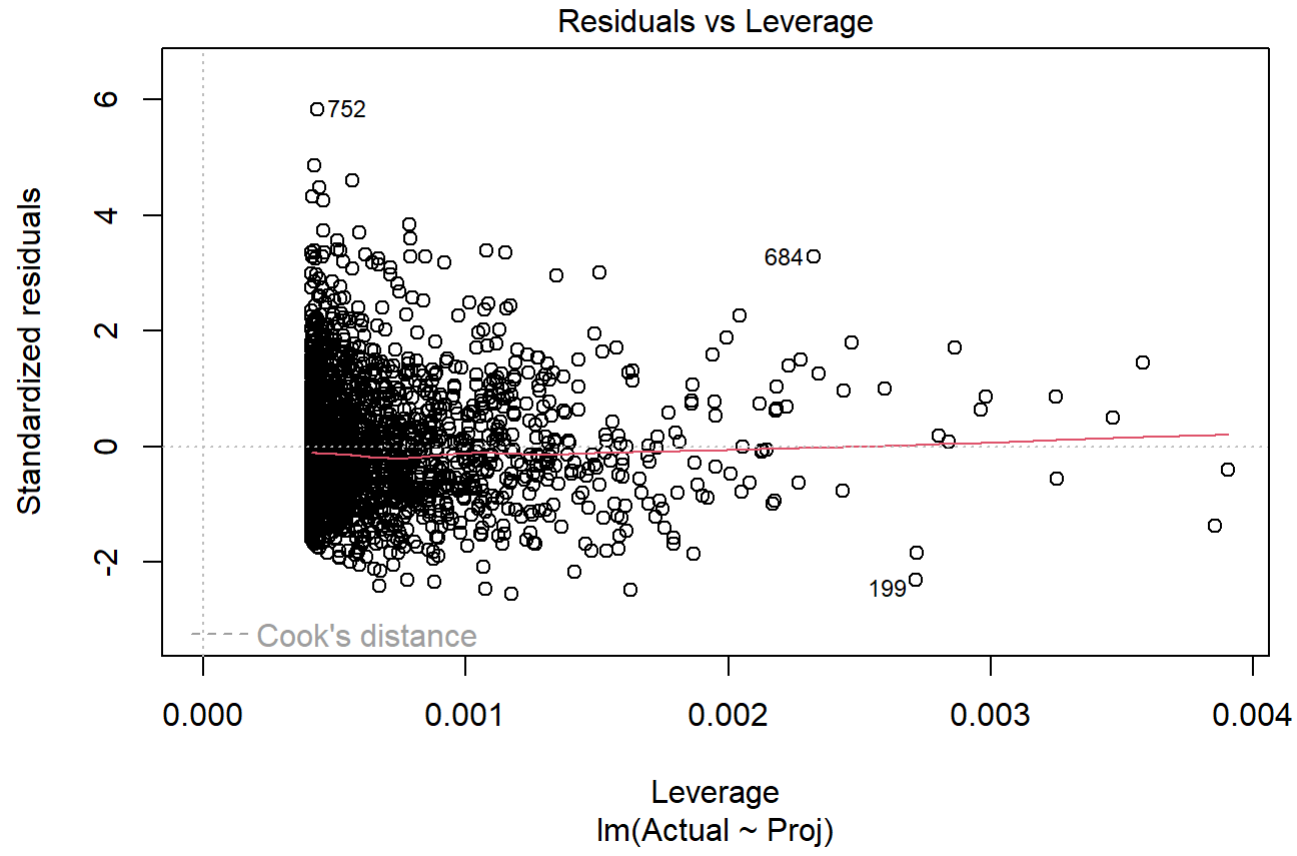
```
##
## Call:
## lm(formula = Actual ~ Proj, data = ESPN_Proj)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -18.983  -5.074  -0.416   3.673  43.375
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.41616    0.34476   1.207    0.228
## Proj         0.95687    0.02655  36.043   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7.439 on 2421 degrees of freedom
##   (154 observations deleted due to missingness)
## Multiple R-squared:  0.3492, Adjusted R-squared:  0.3489
## F-statistic:  1299 on 1 and 2421 DF,  p-value: < 2.2e-16
```

```
plot(Lm)
```


Residuals vs Fitted
752
49
740
Residuals
40
20
0
-20
0
5
10
15
20
25
Fitted values
lm(Actual ~ Proj)


Normal Q-Q
752
49
740
Standardized residuals
6
4
2
0
-2
-3
-2
-1
0
1
2
3
Theoretical Quantiles
lm(Actual ~ Proj)


Scale-Location
752
49
740
√|Standardized residuals|
0.0
0.5
1.0
1.5
2.0
2.5
0
5
10
15
20
25
Fitted values
lm(Actual ~ Proj)


Residuals vs Leverage
752
684
199
Cook's distance
Standardized residuals
-2
0
2
4
6
0.000
0.001
0.002
0.003
0.004
Leverage
lm(Actual ~ Proj)

The code here is pretty simple - we write a linear model with the lm() function (the actual variable as our y variable and the projected variable as our x variable, separated by the ~). We then call the summary() and plot() function to see what's going on with our model.

Notice below how R gives us a nice summary of the results from the linear model that we fit around this data! Let's first look at the residuals and interpret what they mean. We have a 5 number summary at the top (min, median, max, quartiles). For this project, I like to look at the middle 50% of the data (between 1Q and 3Q). Here, since residuals are actual - expected, we can say that ESPN will typically "overproject" by 5.074 pts or "underproject" by 3.673 points (50% of the time). Now let's look at the coefficients. From the estimate column we see that on average, a 1 point increase in ESPN projections corresponds to a 0.95687 point increase in actual output. Pretty good right? The t-value is 36.043, and the p-value is incredibly low. This means ESPN has a "statistically significant," or useful predictor for actual fantasy output! Looking at the bottom piece, we see the headline R^2 value of 0.3492, in that ~35% of the variation (spread) in actual points can be explained by the variation in ESPN's projections.

If you look at the plots below from the plot() function, R gives us a residual plot, a normal probability plot, a standardized residual plot, and a residual v leverage plot. Notice how the residual plot has a bit of a pattern; the residuals seem to fan out a bit for larger values of x. This is further shown with the normal probability plot as it starts to curve upward at the 2nd quintile; the standardized residual plot also shows this as it bends upwards for larger values of x. The residual v leverage plot does not appear to show any strong issues, but overall, these plots indicate a few issues, which signal that our linear model might not be the best choice for this data.

Filter for >10 pts

Next, we will filter for players of interest. It seems like our evaluation of ESPN's model is benefiting from the fact that we include all values of actual points and projected points (for instance, they predict a 0 point performance very well). However, we are not very interested in projecting players to score under 10 points - we would never play them in our lineups! As such, we will use the mutate() function from library(tidyverse) to add a binary column representing players that score over 10 points in a week. We will call this column "Over10" and use an ifelse() function to create it in our dataframe.

```
# Filter for > 10 pts
library(tidyverse)
```

```
## Warning: package 'tibble' was built under R version 4.2.3
```

```
## ── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
## ✓ dplyr     1.1.0     ✓ readr     2.1.4
## ✓ forcats   1.0.0     ✓ stringr   1.5.0
## ✓ lubridate 1.9.2     ✓ tibble    3.2.1
## ✓ purrr     1.0.1     ✓ tidyr     1.3.0
## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the ]8;;http://conflicted.r-lib.org/conflicted package]8;; to force all conflicts t
o become errors
```

```
ESPN_Proj %>%
  mutate(Over10 = ifelse(Actual > 10, 1, 0)) -> ESPN_Proj
```

We will then spawn a new dataframe by filtering for players who have an Over10 value of 1.

```
ESPN_Proj10 <- filter(ESPN_Proj, Over10 == 1)
```

We chose this method to show you a pretty complex way to select players who scored over 10 points (using concepts from prior lectures). Other options that are easier to implement are below:

```
#ESPN_Proj2 <- subset(ESPN_Proj, Actual > 10)
#ESPN_Proj3 <- filter(ESPN_Proj, Actual > 10)
```

Filter for WRs >10 pts

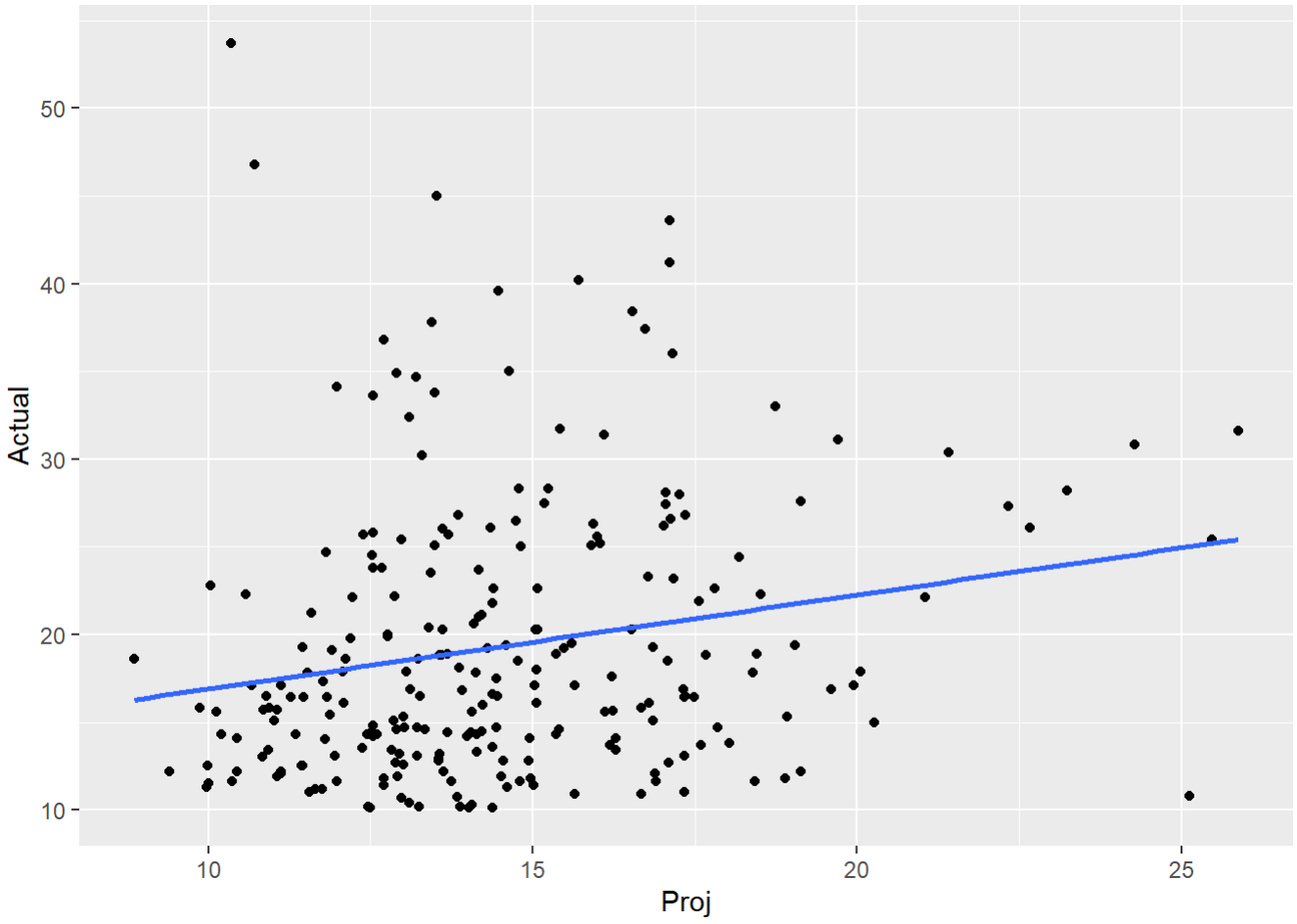
For our project, we will analyze ESPN's projections for WRs of interest (those who score over 10 points). We will use the filter() function to get our dataframe we can analyze:

```
# Filter for WRs > 10 pts
ESPN_WRs <- filter(ESPN_Proj10, Pos == "WR")
```

Again, we will visualize the data to analyze ESPN's projection performance for WRs of interest. We can reuse our code from our earlier visualization and fit a linear model with our new variables and dataframe. Code below:

```
# WR model for performance
ggplot(ESPN_WRs, aes(x=Proj,y=Actual))+geom_point()+geom_smooth(method=lm,se=FALSE)
```

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




```
Lm <- lm(data = ESPN_WRs, Actual ~ Proj)
summary(Lm)
```

```
##
## Call:
## lm(formula = Actual ~ Proj, data = ESPN_WRs)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -14.210  -5.294  -2.032   3.784  36.613
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  11.5344     2.4646   4.680 4.76e-06 ***
## Proj          0.5362     0.1662   3.226  0.00143 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7.742 on 244 degrees of freedom
## Multiple R-squared:  0.04091,    Adjusted R-squared:  0.03698
## F-statistic: 10.41 on 1 and 244 DF,  p-value: 0.001427
```

To keep this document concise, we decided not to show the entire plot(Lm) output for this linear fit. However, feel free to run it if you'd like as it would provide more insight into our analysis. Looking at the summary values, ESPN remains about the same for the middle 50% of the data, either overprojecting by 5.294 points or underprojecting by 3.784 points. We can see the median move to the left, meaning ESPN is overprojecting more often for these receivers. With our estimate slope of 0.5362, the linear fit is more flat, but be careful, because our y axis starts at 10 now (we filtered out 0-10 point scorers but kept the projections the same). Our t-value decreased by a magnitude of 12 from our prior model(!), but the p-value is still statistically significant. Looking at our R^2 value, it decreases all the way to .04091 (from ~.35). The story of ESPN's performance is a little different here, right?

2019 Week 16 Analysis

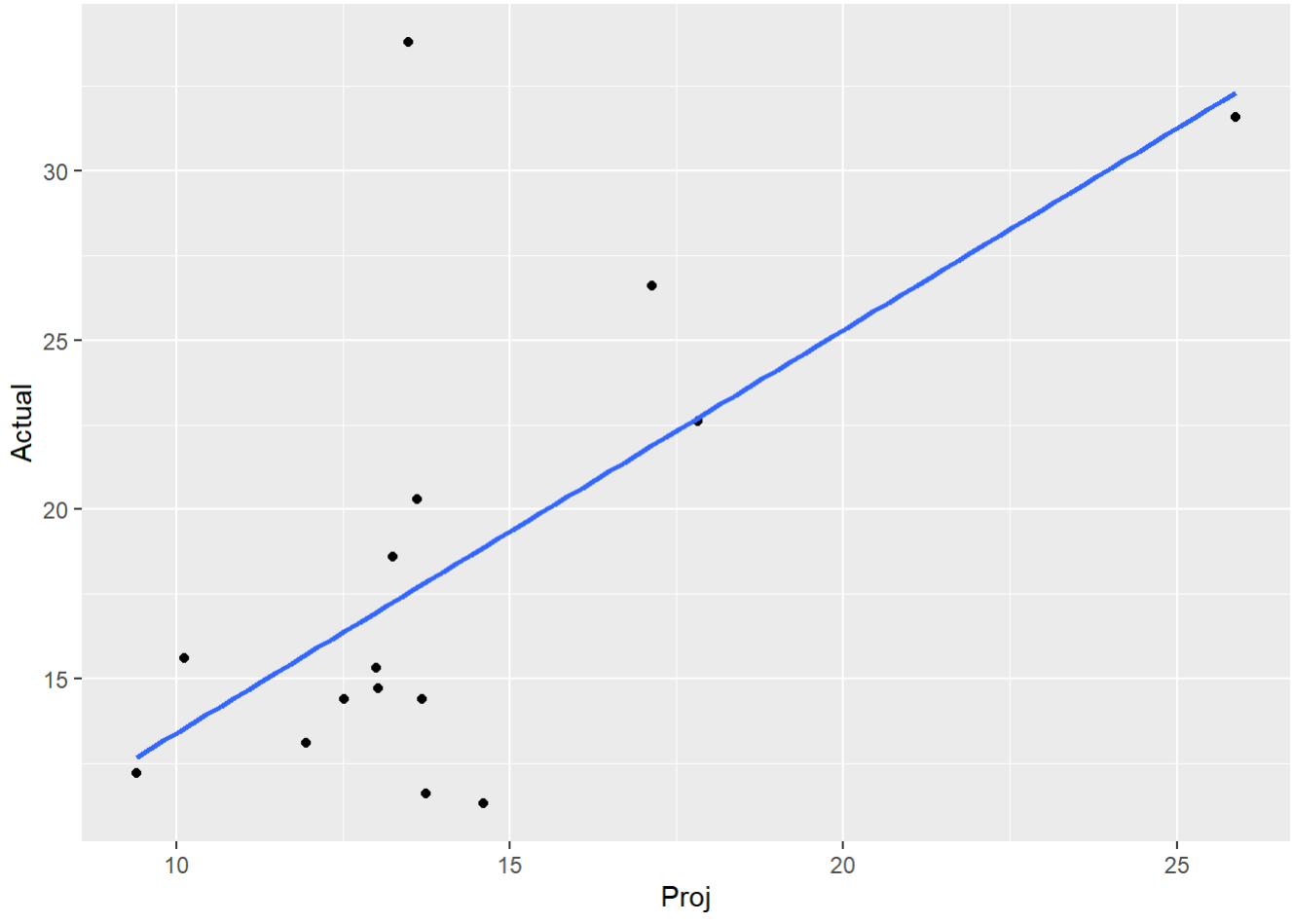
All of our prior analyses were conducted on data for the entire year, but let's zoom in to Week 16 of 2019. We can get this data simply by filtering our last dataframe:

```
# Filter for Wk 16
ESPN_WRs_Wk16 <- filter(ESPN_WRs, Week == 16)
```

We will now conduct our exploratory analysis again to see the big picture:

```
# Wk 16 model for WR performance
ggplot(ESPN_WRs_Wk16, aes(x=Proj,y=Actual))+geom_point()+geom_smooth(method=lm,se=FALSE)
```

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



```
Lm <- lm(data = ESPN_WRs_Wk16, Actual ~ Proj)
summary(Lm)
```

```
##
## Call:
## lm(formula = Actual ~ Proj, data = ESPN_WRs_Wk16)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -7.5799 -2.4432 -0.7012  1.7104 16.2642
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   1.4709     5.8032   0.253  0.80388
## Proj          1.1915     0.3948   3.018  0.00989 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.731 on 13 degrees of freedom
## Multiple R-squared:  0.412,  Adjusted R-squared:  0.3668
## F-statistic: 9.109 on 1 and 13 DF,  p-value: 0.009888
```

Our sample size decreases to 15 WRs, so we will have to be a bit careful, but looking at the plot and our summary values, ESPN does pretty well here in Week 16. The middle 50% of projections either overproject by 2.4432 points or underproject by 1.7104 points, and the median is slightly below 0, which means a slight overprojection at the center of the data. We find that our t-value remains at 3 and the p-value remains statistically significant. Lastly, the R^2 value increases to .412! Not too bad.

Our projection

Now that we have a pretty good idea of ESPN's performance, we will try to build a basic multiple linear regression model to create our own projections for Week 16 of 2019. We will now load a dataset of yearly data where we will build our projections from, and filter for WRs:

```
# Grab season WR data
Yearly_Data <- read.csv("2019FantasyYearlyData.csv")
Yearly_Data <- filter(Yearly_Data, Yearly_Data$Pos == "WR")
```

Our dataset contains full-season statistics. While we can build projections off of these, we'd like to get more context by scaling these stats to weekly averages. We can do this using the mutate() function.

```
# Add weekly data
Yearly_Data %>%
  mutate(RecYdsWk = ReceivingYds / G) %>%
  mutate(TgtWk = Tgt / G) %>%
  mutate(PPW = FantasyPoints / G) %>%
  mutate(RecTDsWk = ReceivingTD / G) %>%
  mutate(RecWk = Rec / G) %>%
  mutate(FumWk = FumblesLost / G) -> Yearly_Data
```

Notice how that code runs as one block: this is by clever use of the ” %>% ” operator.

Now, how will we project a player’s fantasy output? The fantasy scoring equation for WRs is given below: Score = 1*Rec* + *0.1*Rec Yds + 6*Rec TD* -*2*Fumble

We will try to fit a linear model to “project” each of these variables, and then apply this equation on each expected value to achieve our overall score projection.

Receiving Yards

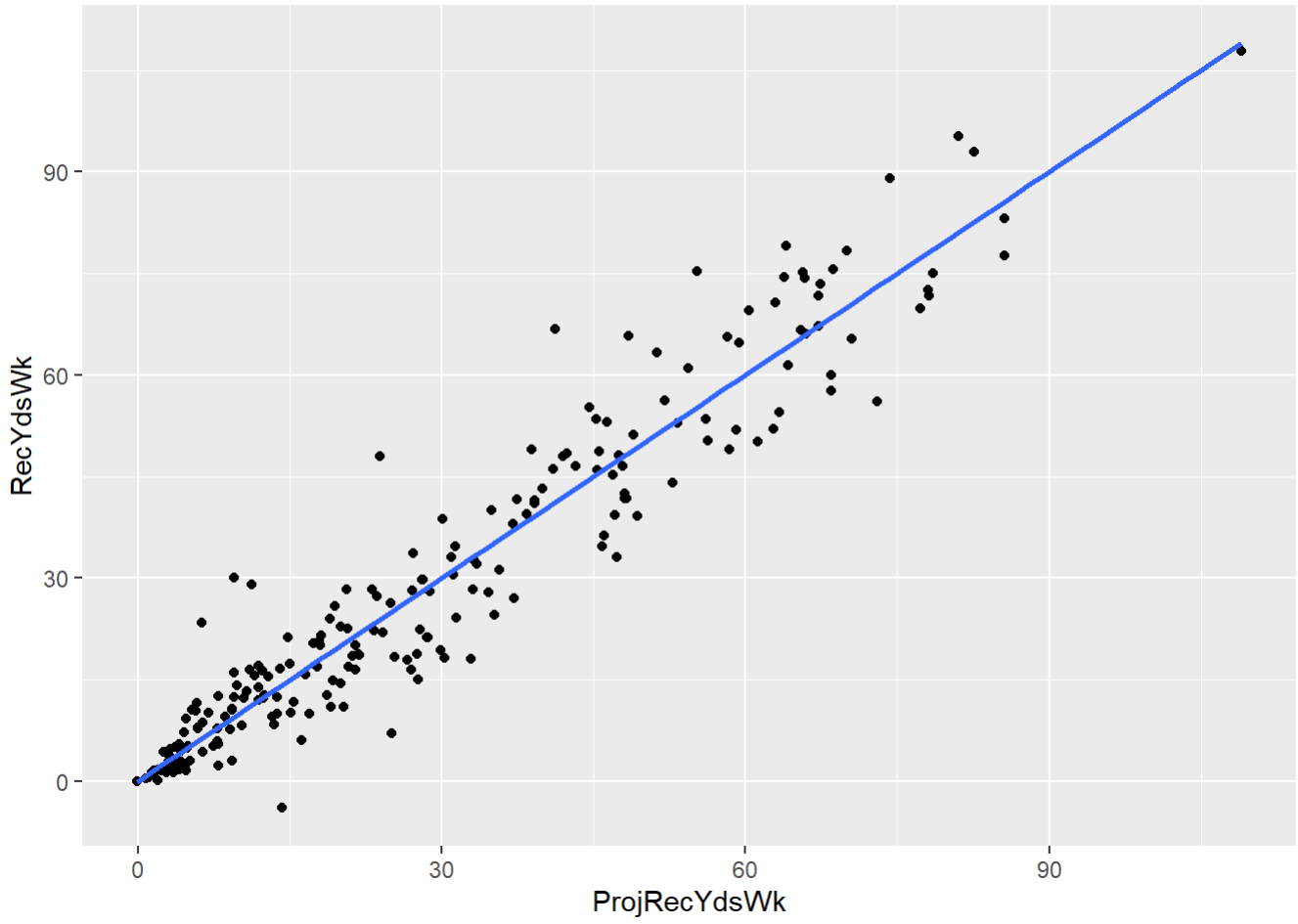
We will now start by fitting a linear model on receiving yards:

```
# Linear model for receiving yards
ProjRecYds <- lm(data = Yearly_Data, RecYdsWk ~ TgtWk+RecTDsWk+RecWk)

# Get projection
Yearly_Data$ProjRecYdsWk <- predict(ProjRecYds)

# Analyze projection performance
ggplot(Yearly_Data, aes(x=ProjRecYdsWk,y=RecYdsWk))+geom_point()+geom_smooth(method=lm,se=FALSE)
```

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




```
summary(ProjRecYds)
```

```
##
## Call:
## lm(formula = RecYdsWk ~ TgtWk + RecTDsWk + RecWk, data = Yearly_Data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -18.2472  -3.8393  -0.0515   3.2700  25.5848
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -0.1211     0.8161  -0.148   0.8821
## TgtWk         2.3731     0.8635   2.748   0.0065 **
## RecTDsWk     25.1320     3.6104   6.961 4.12e-11 ***
## RecWk         7.2490     1.3312   5.445 1.42e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7.086 on 213 degrees of freedom
## Multiple R-squared:  0.9179, Adjusted R-squared:  0.9168
## F-statistic: 794.3 on 3 and 213 DF,  p-value: < 2.2e-16
```

Notice how the syntax for this model is a little different than the past. This is because we are using a multiple linear regression model, where we try to predict a WR's receiving yards per week from a combination of their weekly targets, receiving TDs, and receptions. We create a new column in our dataframe called "ProjRecWk" to apply the linear model equation so we can build individualized player projections later from the predict() function.

Look at the visual and the summary stats on this model - it's really strong! We can predict a player's average receiving yards per week really well when given these variables. Our residuals indicate that we generally predict within ~3-4 receiving yards, all of our variables are statistically significant, and our R^2 value is through the roof. Done, right?

Well, there's actually a statistical check we have to do in order to make sure we aren't cheating. This check we will do is for multicollinearity. Multicollinearity is present when variables in our regression model are highly correlated with each other, which violates the independence assumption and artificially inflates our statistical significance. Let's check if this is the case by calling the vif() function.

```
# Multicollinearity check
library(car)
```

```
## Loading required package: carData
```

```
##
## Attaching package: 'car'
```

```
## The following object is masked from 'package:dplyr':
##
##     recode
```

```
## The following object is masked from 'package:purrr':
##
##     some
```

```
vif(ProjRecYds)
```

```
##     TgtWk  RecTDsWk     RecWk
## 24.537232  2.203048 25.052111
```

A vif value >10 is something that we flag by convention, and here, the ~25 values for targets and receptions indicate that we have to get rid of one of these variables. This makes intuitive sense as the variables are strongly correlated. As such, you can make an argument for getting rid of either variable, but we will pick targets over receptions to keep in our model. We will now re-run the projection below:

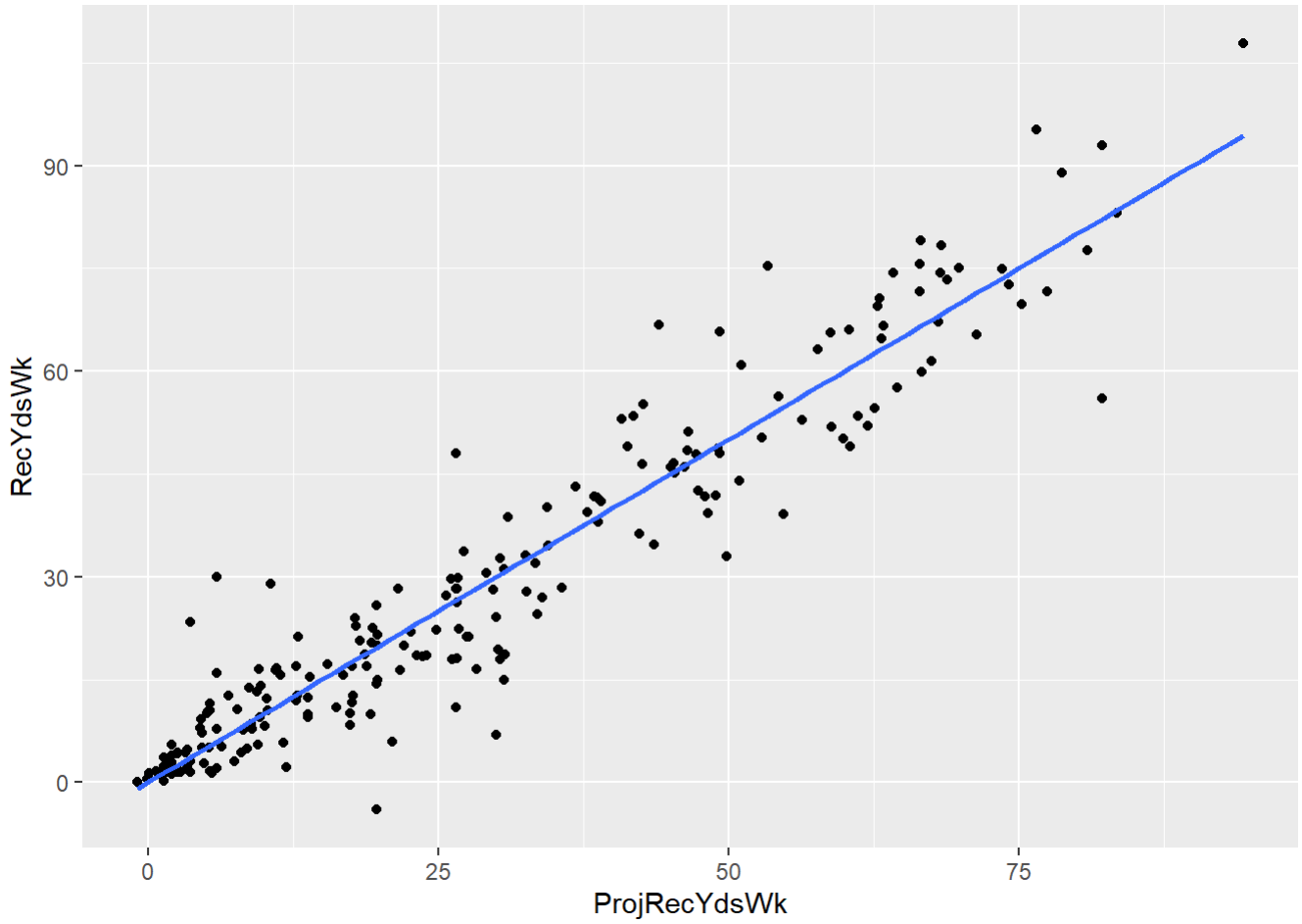
Receiving Yards Correction

```
# Rerun projections without receptions
# Linear model for receiving yards
ProjRecYds <- lm(data = Yearly_Data, RecYdsWk ~ TgtWk+RecTDsWk)

# Get projection
Yearly_Data$ProjRecYdsWk <- predict(ProjRecYds)

# Analyze projection
ggplot(Yearly_Data, aes(x=ProjRecYdsWk,y=RecYdsWk))+geom_point()+geom_smooth(method=lm,se=FALSE)
```

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




```
summary(ProjRecYds)
```

```
##
## Call:
## lm(formula = RecYdsWk ~ TgtWk + RecTDsWk, data = Yearly_Data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -26.2199  -4.4802   0.0065   3.8630  24.0453
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -0.9098     0.8552  -1.064    0.289
## TgtWk         6.8645     0.2721  25.228  < 2e-16 ***
## RecTDsWk     28.2134     3.7970   7.430 2.55e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7.546 on 214 degrees of freedom
## Multiple R-squared:  0.9065, Adjusted R-squared:  0.9057
## F-statistic:  1038 on 2 and 214 DF,  p-value: < 2.2e-16
```

```
# Check for multicollinearity
vif(ProjRecYds)
```

```
##    TgtWk RecTDsWk
## 2.148931 2.148931
```

We can see that the model still performs well, and our vif values are within convention. We can now continue to project receptions (code below is finalized after checking vif values… feel free to include whichever variables you'd like in your model, but make sure to check the vif before continuing).

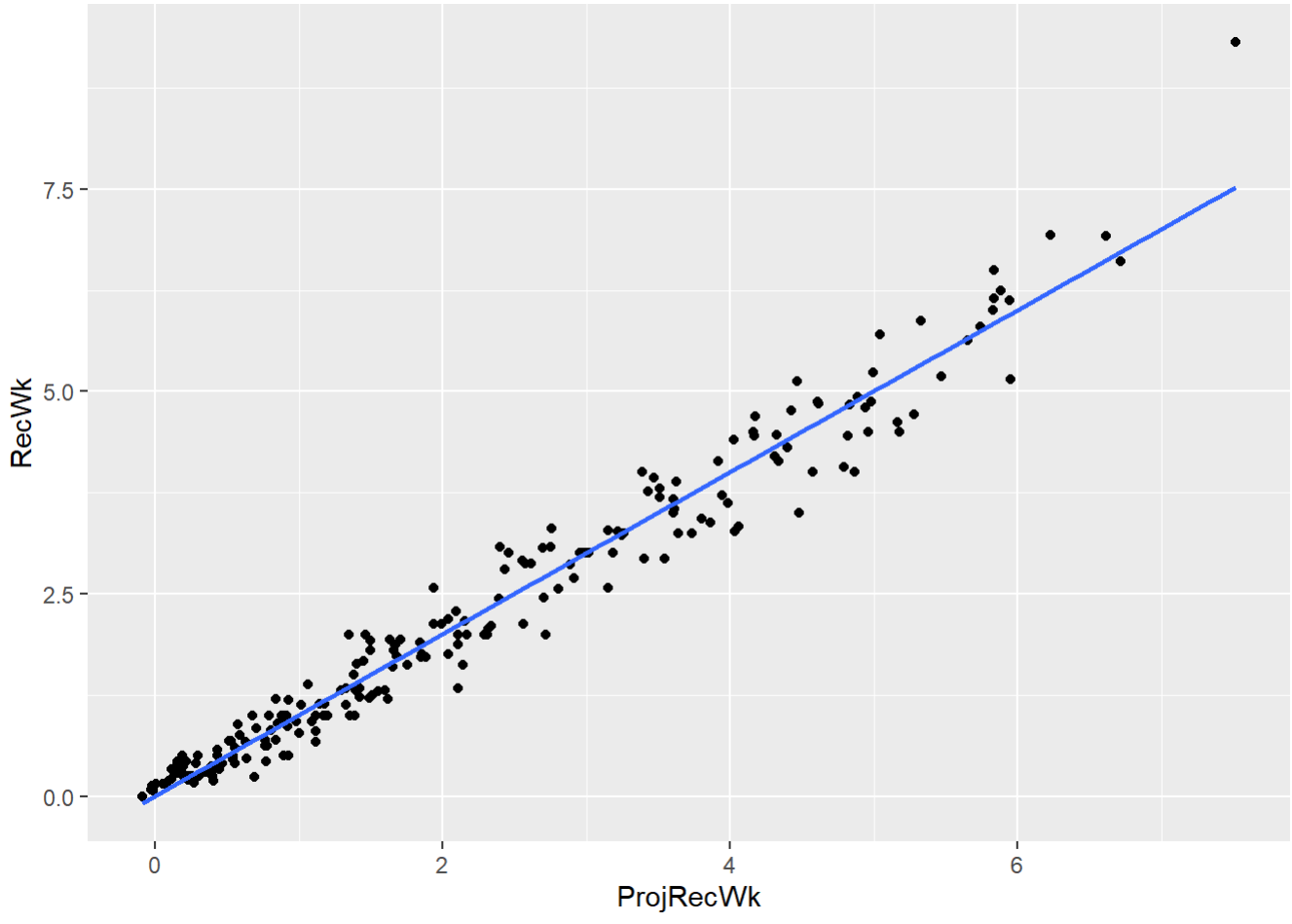
Receptions

```
# Linear model for receptions
ProjRec <- lm(data = Yearly_Data, RecWk ~ RecYdsWk+TgtWk)

# Get projection
Yearly_Data$ProjRecWk <- predict(ProjRec)

# Analyze projection
ggplot(Yearly_Data, aes(x=ProjRecWk,y=RecWk))+geom_point()+geom_smooth(method=lm,se=FALSE)
```

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



```
summary(ProjRec)
```

```
##
## Call:
## lm(formula = RecWk ~ RecYdsWk + TgtWk, data = Yearly_Data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.98006 -0.18721  0.01326  0.20501  1.79677
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.092992   0.038707  -2.402   0.0171 *
## RecYdsWk     0.016491   0.002754   5.988 8.89e-09 ***
## TgtWk        0.504287   0.024461  20.616  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.341 on 214 degrees of freedom
## Multiple R-squared:  0.9649, Adjusted R-squared:  0.9646
## F-statistic:  2946 on 2 and 214 DF,  p-value: < 2.2e-16
```

```
# Multicollinearity check
vif(ProjRec)
```

```
## RecYdsWk    TgtWk
## 8.504026 8.504026
```

We are able to project a wideout's average receptions per week very well given their average yards and targets per week. We should be a little cautious about this one though… reception yards and targets seem to overfit the data. But for now, we will bypass this (this model won't be very useful) and look at the remaining variable, receiving TDs:

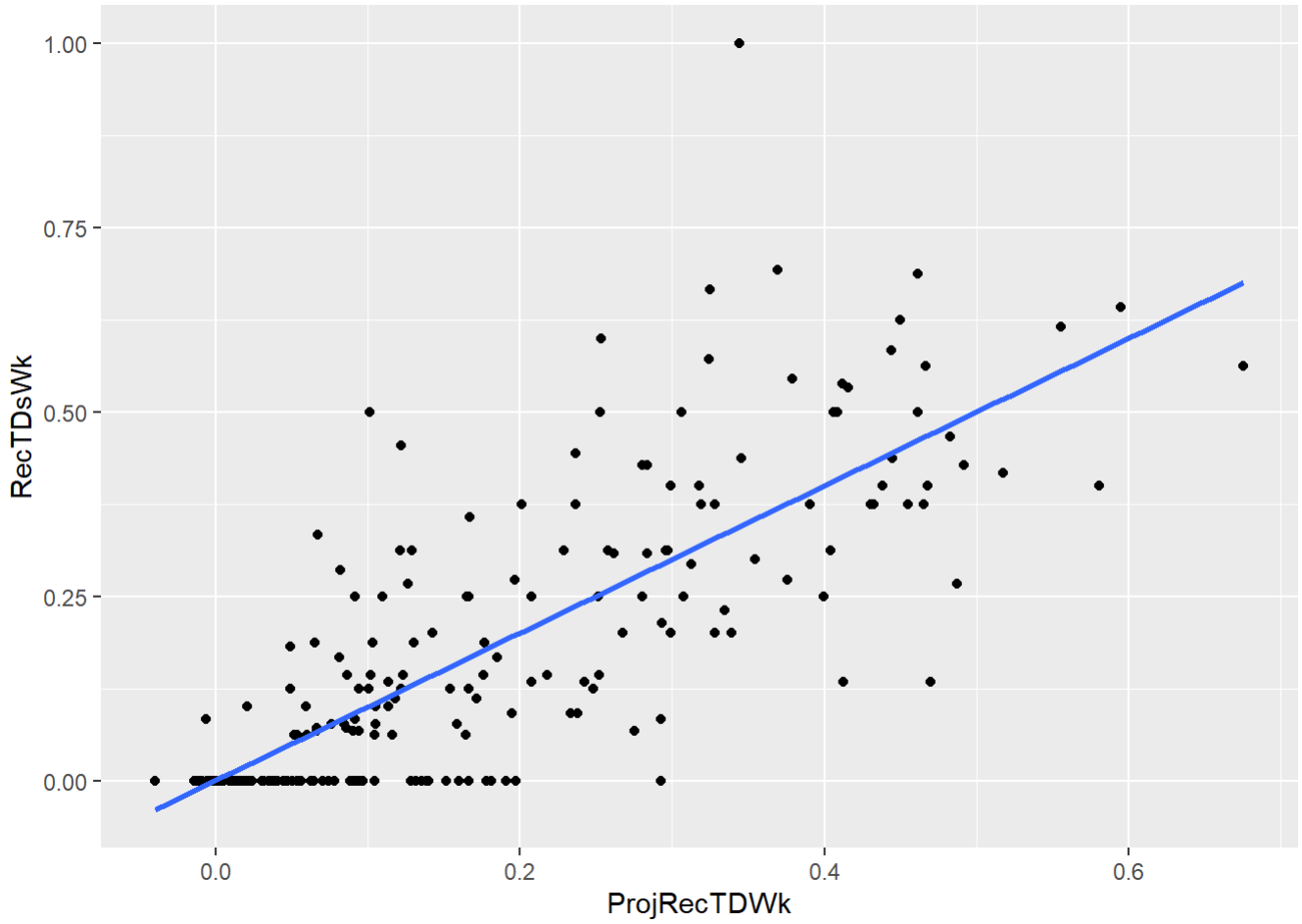
Receiving TDs

```
# Linear model for receiving touchdowns
ProjRecTD <- lm(data = Yearly_Data, RecTDsWk ~ RecYdsWk)

# Get projection
Yearly_Data$ProjRecTDWk <- predict(ProjRecTD)

# Analyze projection
ggplot(Yearly_Data, aes(x=ProjRecTDWk,y=RecTDsWk))+geom_point()+geom_smooth(method=lm,se=FALSE)
```

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




```
summary(ProjRecTD)
```

```
##
## Call:
## lm(formula = RecTDsWk ~ RecYdsWk, data = Yearly_Data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.33616 -0.06384 -0.00934  0.04576  0.65589
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.0141204  0.0127649  -1.106     0.27
## RecYdsWk     0.0063970  0.0003354  19.073   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1211 on 215 degrees of freedom
## Multiple R-squared:  0.6285, Adjusted R-squared:  0.6268
## F-statistic: 363.8 on 1 and 215 DF,  p-value: < 2.2e-16
```

Final projection

Notice how we left out fumbles in our projection for WR weekly stats. This is a decision made with the assumption that fumbles are a rare, highly variable random event; we see no use in projecting a player's fumble total for each week. You can check this empirically, but to save time, we will bypass fumbles and just impute the average for each player. Therefore, we now have every variable needed to write our projection equation (for PPR scoring being our scoring method):

```
# Generate final projection
Yearly_Data$Our_Proj_PPW <- Yearly_Data$ProjRecWk + 6*Yearly_Data$ProjRecTDWk + 0.1*Yearly_Data
$ProjRecYdsWk - 2*Yearly_Data$FumWk
```

Now, to compare this model to ESPN's model, we are going to use the merge() function to combine both dataframes of interest. In the ESPN_WRs_Wk16 dataframe, we have the WRs of interest already filtered (>10 pts and results for Week 16 rather than yearly), and in the Yearly_Data dataframe, we have our projections for each player. We will now combine these dataframes using the code below:

```
# Combine both dataframes - join on player
Merge <- merge(ESPN_WRs_Wk16, Yearly_Data, by = "Player")
```

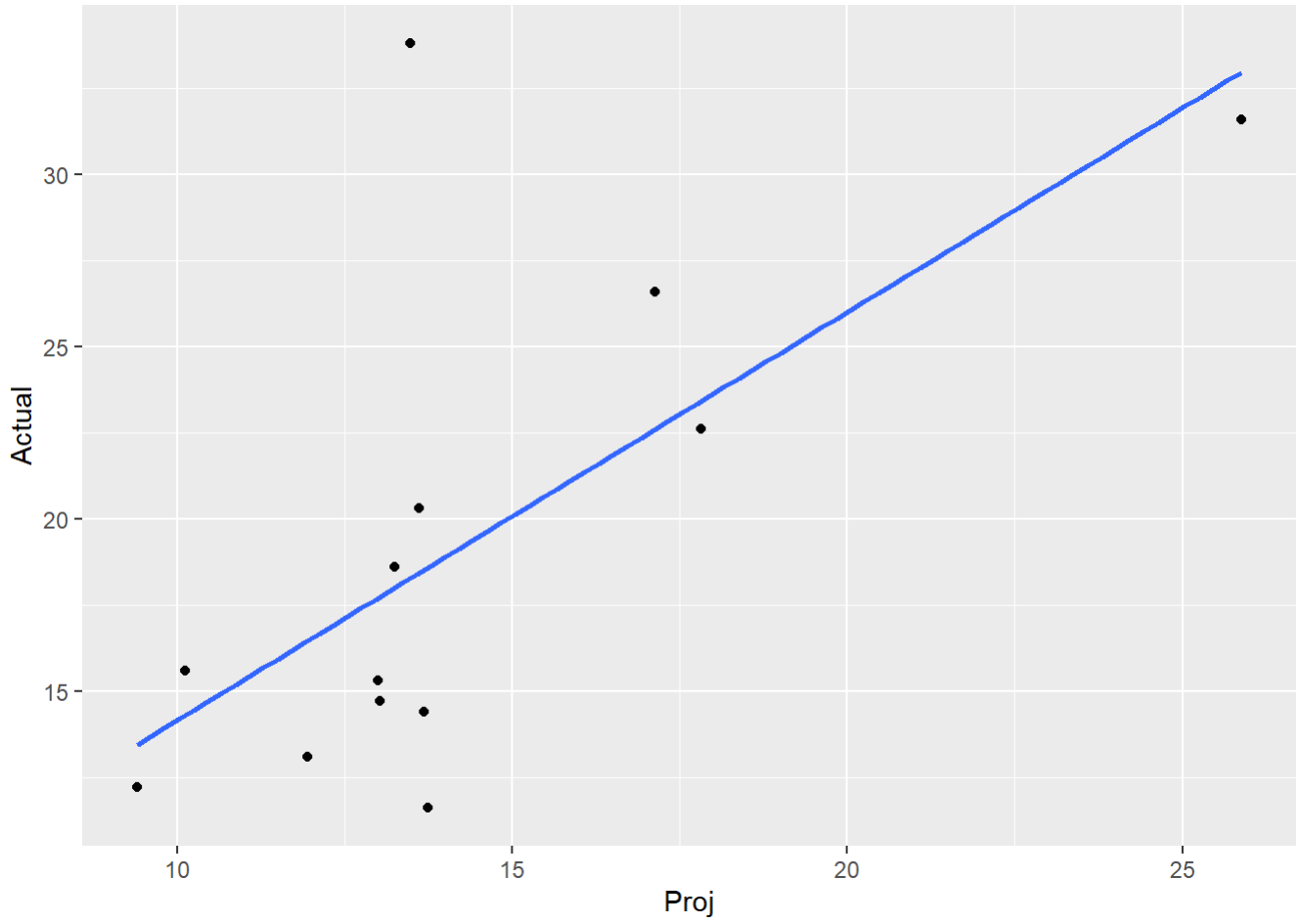
Notice how we merged by the column "Player." The merge function matches the two dataframes by their shared column titled "Player." It is imperative that these columns have the same name, or else the merge() function will return an error message. We can now compare the performance of our model v ESPN's!

ESPN Model

Here is a reminder of the performance of the ESPN model:

```
# ESPN model performance
ggplot(Merge, aes(x=Proj,y=Actual))+geom_point()+geom_smooth(method=lm,se=FALSE)
```

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




```
Lm <- lm(data = Merge, Actual ~ Proj)
summary(Lm)
```

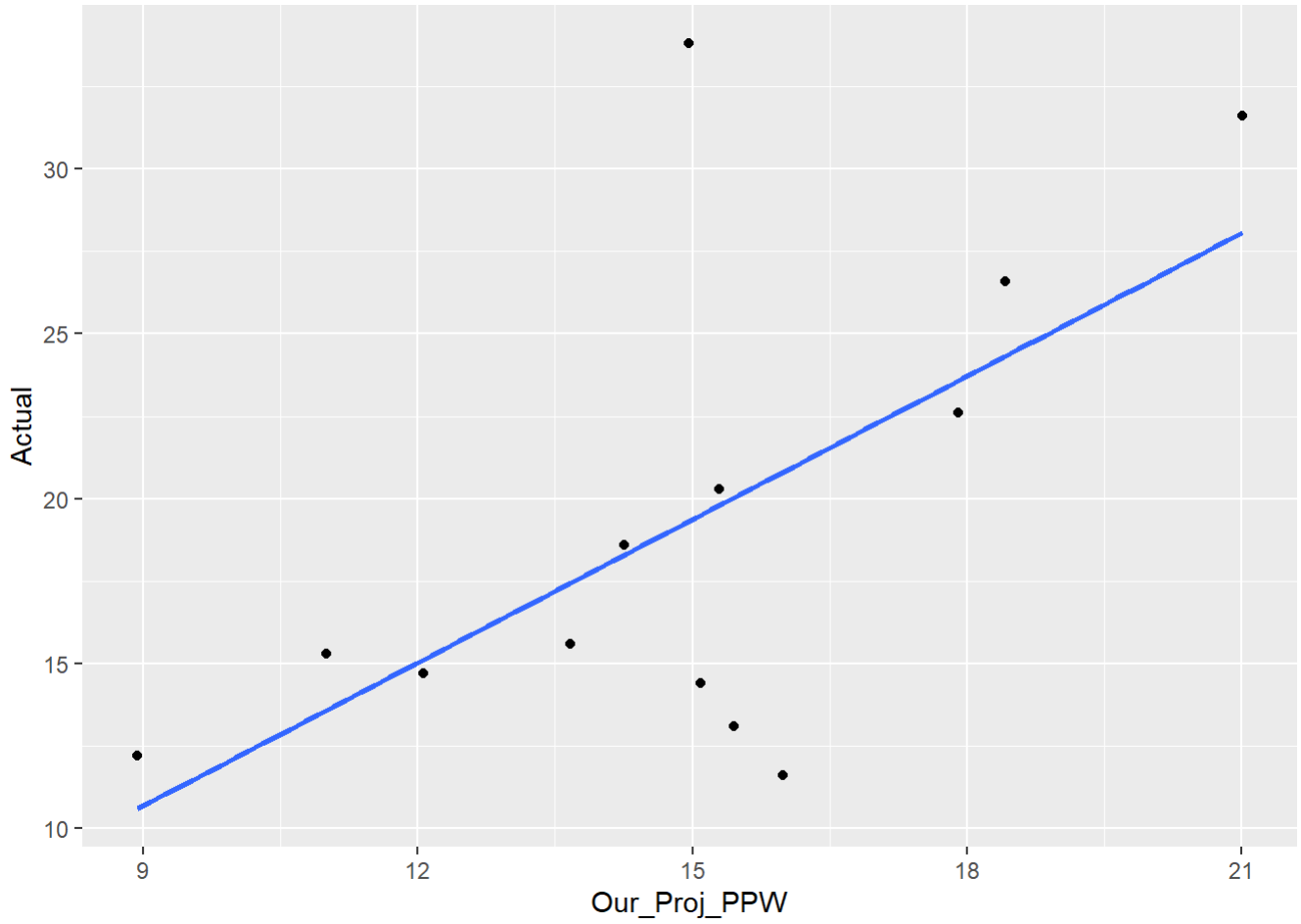
```
##
## Call:
## lm(formula = Actual ~ Proj, data = Merge)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -6.992 -3.029 -1.242  1.305 15.524
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   2.2969     5.8898    0.39   0.7040
## Proj          1.1852     0.3963    2.99   0.0123 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.71 on 11 degrees of freedom
## Multiple R-squared:  0.4484, Adjusted R-squared:  0.3983
## F-statistic: 8.943 on 1 and 11 DF,  p-value: 0.01229
```

Our model

And now our model!

```
# Our model performance
ggplot(Merge, aes(x=Our_Proj_PPW,y=Actual))+geom_point()+geom_smooth(method=lm,se=FALSE)
```

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




```
Lm <- lm(data = Merge, Actual ~ Our_Proj_PPW)
summary(Lm)
```

```
##
## Call:
## lm(formula = Actual ~ Our_Proj_PPW, data = Merge)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -9.204 -1.834  0.300  1.722 14.480
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   -2.3411     8.3070  -0.282   0.7833
## Our_Proj_PPW   1.4475     0.5453   2.654   0.0224 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.003 on 11 degrees of freedom
## Multiple R-squared:  0.3904, Adjusted R-squared:  0.335
## F-statistic: 7.046 on 1 and 11 DF,  p-value: 0.02241
```

Summary

Comparing the summary stats of the two models, ESPN's model is a slightly more accurate and statistically significant predictor for WRs based on the Week 16 dataset. However, looking at the middle 50% of the data, our projections typically fall within 2 points of the actual result, while ESPN's usually overproject by 3 or underproject by 1. Comparing the mins and maxes, ESPN better captures the lower point values than our model, while we do a better job with the higher point values. More analysis across multiple weeks (with more sample) could be conducted to dig deeper!

A Cautionary Tale and Considerations

Notice how in our yearly dataset we actually had a full year of data in which we built our projection system; therefore, we did factor in actual Week 16 and 17 data in our analysis. At the time of the Week 16 projection, the ESPN model did not have this data (our case would be called a data leakage). Furthermore, while it is impressive that we could predict the average statistics for each WR, predicting with an R^2 value of close to 1 is not really useful in a modeling sense. We are taught that a high R^2 value is good, but if it is close to perfect, why wouldn't we just extrapolate the average values rather than use the model? Therefore, it is important to balance model accuracy with usefulness, because if you overfit the data, your model will not fit well on new data. Additionally, ESPN surely doesn't use linear models for this data, and as evidenced by our residual plots, a linear model is likely not the best fit for this data. More methods would need to be explored. Additionally, there are a lot of variables we did not account for, like pace of play, weather, injuries, quarterback play, etc. Surely inclusion of this data would help make our model more robust (and may make us less prone to outliers). There are also many different criteria to select variables for your model (not just vif), which you can research. However, I hope you enjoyed this exercise and remain curious at looking at real-world datasets to see what we can discover and learn :)

Lecture 5 - Functional Programming

Functional Programming

In this lecture, we will learn very useful procedures to make our code reusable, simpler, and easier to read and write! We will learn about functions, which are just as important as vectors in R. R was made popular by its unique function and vector capabilities, making it a convenient language for coding. We will also learn a bit about loops, which appear often in functions that you'll write in the future. Let's get started!

Functions

You already know several functions that you have used in R - think of functions like sample(), lm(), filter(), and mutate(). Even the simple ones like plot() and mean() are functions as well!

How have we used these functions in our scripts? We call them directly, and they usually get their own lines in our script. Often, we will use a function to add its outputs to a dataframe, like this:

```
dataframe$NewColumnName <- someFunction(someInput)
```

Other times, if we just want to see some results for our calculations, we will use functions and write them like this:

```
plot(dataframe$column)
summary(dataframe$column)
vif(someLinearModel)
```

This is totally valid coding - code like this works perfectly and is absolutely fine to include in your scripts! But when we are doing really long analyses or complex projects, wouldn't it be pretty annoying to copy this code over and over again to see your results? Remember from our linear modeling lecture - how many times did we copy and paste a code block like this?

```
ProjRecYds <- lm(data = Yearly_Data, RecYdsWk ~ TgtWk+RecTDsWk)

Yearly_Data$ProjRecYdsWk <- predict(ProjRecYds)

ggplot(Yearly_Data, aes(x=ProjRecYdsWk,y=RecYdsWk))+geom_point()+geom_smooth(method=lm,se=FALSE)

summary(ProjRecYds)

vif(ProjRecYds)
```

It got pretty tedious after a few times, right? Well, the good news is that there is a better way to do this, and it is by using functions! A great use case for functions is when you find yourself repeating the same code but tweaking the inputs each time you want to run something. But what happens if you can't find a function in R that simplifies your work? The answer is that you can write your own custom functions in R to help you out! We'll see the usefulness of this at the end of this lecture when we understand custom (user-defined) functions a bit more.

Writing Our Own Functions

R has a neat way in which we can define a function in our script and "call" it (fancy way to say use a function) to do something for us.

Functions that are already built in R are ready to be called, so we use them like this:

```
functionName(parameter)

# or...

functionName(parameter1, parameter2, etc)
```

So how are we able to use a custom function? We have to define it first so we can call it! All functions in R are defined, but the functions already built are what we call “under the hood” - they are there, but we just can’t see their source code in our script. Notice how packages are basically just a bunch of functions with pre-built definitions, so when we load a package, we are downloading someone’s custom function definitions and using their functions on our computer. Pretty neat, right?

Defining Our Own functions

Source: https://www.tutorialspoint.com/r/r_functions.htm (https://www.tutorialspoint.com/r/r_functions.htm)

Every function you define will have a structure like this:

```
functionName <- function(parameter1, parameter2, ...) {
  functionBody
  returnValue
}
```

The components of a function are:

- Function name: Required. It is the name of the function (that you define, which is stored in R as an object by its name. Make sure you use the arrow to point to your function name.
- The word “function”: Required. You have to write this, or else R will not know what you are doing. “function” is the key word that lets R know you are defining a function here!
- Parameters: Optional. When you use a function, you pass a value(s) as its parameter(s). You can write a function without parameters (but you still have to include the parentheses before the brackets), but that typically isn’t best practice in a larger program.
- Function body: Required. Pardon the pun, but this is where the “functionality” of your function is built. You tell R what you want the function to do by writing some code in its body.
- Return value: Optional. You can have a function return a specific value that you define in your function body by using return(theValueYouWantToReturn).

After you do this, you can call your function in your script by doing this:

```
functionName(parameter1, parameter2, ...)

# or...

functionName()
```

Example

Let’s start by writing a really simple function that does a math calculation for us.

```r
# Simple math function
mathFunction <- function(a, b, c) {

result <- a * b + c
print(result)

}
```

What did we do here? We wrote a function called “mathFunction” with parameters “a, b, and c.” In the function’s body, we want the mathFunction to calculate a result for us, called “result,” which we calculate by multiplying a and b and then adding c. We do not have a return value for this function (though we could have “result” as the return value and it would do the same thing) - we just ask the function to print the result using print(result) in the body.

So how do we use this function? Now that we defined it, we can call it!

```r
## mathFunction calls
mathFunction(2, 3, 6)
```

```
## [1] 12
```

```r
#mathFunction(2, 3)
mathFunction(2, 3, 0)
```

```
## [1] 6
```

```r
mathFunction(2, 0, 4)
```

```
## [1] 4
```

```r
mathFunction(a = 10, b = 5, c = 3)
```

```
## [1] 53
```

```r
#mathFunction(a = 4, c = 7)
mathFunction(a = 2, c = 1, b = 6)
```

```
## [1] 13
```

Notice how we threw a bunch of different parameters in our function, and it works! Also notice that if we leave out a parameter, R returns an error. Since we did not define defaults, R does not know what value to assign that parameter, so we get an error. Coding tip: pick smart defaults for your parameters so you can easily detect errors in your code!

Also notice that order matters. That last line still works when we assign values to the parameters out of order, but otherwise, if you do not directly assign values to parameters, the order in which you enter parameters does matter.

Let’s quickly redefine this function with default parameters (and a return value):

```
## With default parameters
mathFunctionNew <- function(a = 1, b = 4, c = 5) {
  result <- a * b + c
  return(result)
}

mathFunctionNew(2, 3, 6)
```

```
## [1] 12
```

```
mathFunctionNew(2, 3)
```

```
## [1] 11
```

```
mathFunctionNew(2)
```

```
## [1] 13
```

```
mathFunctionNew(,3,)
```

```
## [1] 8
```

```
mathFunctionNew()
```

```
## [1] 9
```

```
mathFunctionNew(a = 10, b = 5, c = 3)
```

```
## [1] 53
```

```
mathFunctionNew(a = 4, c = 7)
```

```
## [1] 23
```

```
mathFunctionNew(a = 2, c = 1, b = 6)
```

```
## [1] 13
```

Notice the new behavior of our function (no errors), but also notice the weird behavior with our function when we remove parameters. This is why it's important to choose clever defaults! You can also have some error checks in your function body, but that's some pretty advanced programming you can research on your own.

So How Can This Be Useful?

Functions are very useful if you are clever about it! Let's run through a few examples of some useful applications of functions:

Source: https://github.com/hadley/adv-r/blob/master/FP-whole-game.Rmd (https://github.com/hadley/adv-r/blob/master/FP-whole-game.Rmd)

Data Cleaning

Let's say you have a dataframe of numbers and want the -2's to be NAs. Here is the dataframe:

```
# Generate dataframe
set.seed(101)
df <- data.frame(replicate(6, sample(c(1:10, -2), 6, rep = TRUE)))

# Rename headers, show dataframe
names(df) <- letters[1:6]
df
```

```
##    a b  c  d  e  f
## 1  9 6  2  6 -2  8
## 2  9 3  4  8  5  4
## 3  7 3  5 10 10  6
## 4  1 9 -2  5  5  8
## 5 10 3  1 10  4  7
## 6 -2 3  1 10  8 10
```

See how we got some -2's in there? How would we normally get rid of them? Something tedious like this:

```
df$a[df$a == -2] <- NA
df$b[df$b == -2] <- NA
df$c[df$c == -2] <- NA
df$d[df$d == -2] <- NA
df$e[df$e == -2] <- NA
df$f[df$f == -2] <- NA
```

Well, what if we could write a function to help with this? See below:

```
# Replace -2 with NA
fix_missing <- function(x) {
  x[x == -2] <- NA
  return(x)
}

# Call fix_missing()
df <- fix_missing(df)
df
```

```
##    a b  c  d  e  f
## 1  9 6  2  6 NA  8
## 2  9 3  4  8  5  4
## 3  7 3  5 10 10  6
## 4  1 9 NA  5  5  8
## 5 10 3  1 10  4  7
## 6 NA 3  1 10  8 10
```

Pretty easy and compact, right? We give the fix_missing function a parameter (our dataframe), and then we remove all -2's in the dataframe and return the new dataframe. We then call the function and save it as our df. Done!

Exploratory Data Analyses

Here's another example. Using the same dataframe, say we want to figure out some number summaries for each column. How would we do this normally? Something like this:

```
mean(df$a)
median(df$a)
sd(df$a)
mad(df$a)
IQR(df$a)

mean(df$b)
median(df$b)
sd(df$b)
mad(df$b)
IQR(df$b)
```

For EVERY column. Not too fun, right? Well, we can write a function to help us solve this problem:

```
## Summary stats function
summaryFunc <- function(x) {
           c(mean(x, na.rm = TRUE),
           median(x, na.rm = TRUE),
           sd(x, na.rm = TRUE),
           mad(x, na.rm = TRUE),
           IQR(x, na.rm = TRUE))
}

# Apply the function to our dataframe
lapply(df, summaryFunc)
```

```
## $a
## [1] 7.20000 9.00000 3.63318 1.48260 2.00000
##
## $b
## [1] 4.50000 3.00000 2.50998 0.00000 2.25000
##
## $c
## [1] 2.60000 2.00000 1.81659 1.48260 3.00000
##
## $d
## [1] 8.166667 9.000000 2.228602 1.482600 3.500000
##
## $e
## [1] 6.40000 5.00000 2.50998 1.48260 3.00000
##
## $f
## [1] 7.166667 7.500000 2.041241 1.482600 1.750000
```

Dataframes are lists, so we can use the lapply() function to apply the summaryFunc to this dataframe. Notice how quickly we got that output!

Model Building

Remember in linear modeling when we had blocks of code like this?

```
ProjRecYds <- lm(data = Yearly_Data, RecYdsWk ~ TgtWk+RecTDsWk)

Yearly_Data$ProjRecYdsWk <- predict(ProjRecYds)

ggplot(Yearly_Data, aes(x=ProjRecYdsWk,y=RecYdsWk))+geom_point()+geom_smooth(method=lm,se=FALSE)

summary(ProjRecYds)

vif(ProjRecYds)
```

We could get this into a function, right? It could look something like this:

```
LmFunc <- function(data, projectedVar, dataFrameVar, predictVar, explanatory1, explanatory2) {

  projectedVar <- lm(data = data, predictVar ~ explanatory1+explanatory2)

  data$dataFrameVar <- predict(projectedVar)

  # Return multiple results in a list
  results <- list(
    ggplot(data, aes(x=dataFrameVar,y=predictVar))+geom_point()+geom_smooth(method=lm,se=FALSE),
    summary(projectedVar),
    vif(projectedVar)
  )

  return(results)

}

# Call function
run <- LmFunc(Yearly_Data, ProjRecYds, ProjRecYdsWk, Yearly_Data$RecYdsWk, Yearly_Data$TgtWk, Ye
arly_Data$RecTDsWk)
run
```

This really simplifies the process, right? All you have to do is define the function once and your workflow becomes much easier, concise (limiting potential for mistakes), and faster!

But the real takeaway from these examples is that functions can be used in ANY part of the data workflow, from cleaning, to exploring, to analyzing, to interpreting results!

Looping

Now that we know a bit about functions, I'll briefly introduce looping. Looping is useful and often deployed in many programming languages - in this respect R is no different! Just like functions, looping is a special function that allows a computer's instructions to repeat. Loops often appear within functions, so that's why we will learn about them in this lecture!

There are two types of loops we will learn about for R:

- For loop
    - This loop will iterate over itself for a predetermined, specified number of times as stated by a condition. When you know how many times you need to iterate something, use a "for loop."
- While loop
    - This loop will check a condition first, and then as long as the condition is true, the loop will iterate over itself. When you want R to just iterate until some condition changes, then use a "while loop."

This may be a bit confusing to understand now, so let's do some examples (and you can see how they appear/work within functions)!

For Loop

Let's write a function where I want to print a sequence of squares. I can do this in a function like this:

```
## Function for squares
sqFunction <- function(x) {
    print(x^2)
}
sqFunction(1)
```

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

```
sqFunction(2)
```

```
## [1] 4
```

```
sqFunction(3)
```

```
## [1] 9
```

```
sqFunction(4)
```

```
## [1] 16
```

Notice how this function does work, but we have to call it so many times! This is where a loop would be useful - we want to call the function many times where we don't have to manually script for each call. We use a for loop for this one:

```
## For loop fix
sqFunction <- function(x) {

  for(i in 1:x) {
    print(i^2)
  }

}
sqFunction(4)
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
```

Notice how we used the for loop within the sqFunction - we used "x" as our parameter to control how many times sqFunction was called, or really to define how many times the for loop runs.

In programming, we often use "i" to represent the "index" of a loop, which in this case is 1, 2, 3, and 4. The loop runs for each index of i, and prints (because the print is in the loop. Notice what happens if you put the print outside of the for loop function)!

For more info on for loops: https://www.w3schools.com/r/r_for_loop.asp (https://www.w3schools.com/r/r_for_loop.asp)

While Loop

The while loop is a little easier to understand grammatically; the while loop will iterate “while” a condition is true. Let’s write a while loop that will “print integers while they are less than 6”:

```
## While loop example
i <- 1
while (i < 6) {
  print(i)
  i <- i + 1
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Notice first that we don’t have to put a loop in a function, we just leave it by itself here. Also notice that we initialize i outside of the while loop, so i can start at 1. We then check the condition (“is i less than 6?”), and if true, run what is inside the while loop. Notice that i increments up by 1 each time (by the i <- i + 1 line). This is how we get i to reach 6, which is when the loop stops. If you do not increment your i, your loop will continue forever (which is never good)!

For more info on while loops (there are some cool things you can do within loops) : https://www.w3schools.com/r/r_while_loop.asp (https://www.w3schools.com/r/r_while_loop.asp)

Loops Use Cases

In any case where you want to recursively define a column in your dataframe, looping is the way to do this! Say you are taking a moving average of something; looping can achieve that goal by iterating over each cell as you go! You can also nest loops in your if… else statements, in your functions, and in each other (oh my!). It is a difficult and seemingly scary concept as you are new to programming, but as you see them in practice and start using them, you will start to recognize when they are useful! We cover this very briefly, so checking out those links/finding other online resources will be helpful for you on this topic.

Some Additional Notes on Scope and Practices for Functions

This marks the end of the functional programming lecture! Some additional notes are here that will be helpful as you start to write and work with functions:

- Variables have what is called a “scope,” meaning R understands what variables (and saved values) mean ONLY within their scope.
    - In the past, we have only been using what are called “global” variables - global variables can be read anywhere in your code by R (after you define them, of course).
    - Functions in R have what are called “local” variables - local variables are only readable within their function but not outside of it. This is the case when you define a new variable inside of a function.
        - Example: the “result” variable in our mathFunc can be read by mathFunc, but not anywhere else!

- We have to be careful about scope! So, why don't we just define all of our variables globally? Well, this can lead to difficult debugging if our functions share the same variables. Therefore, it is best practice to have as few global variables as possible, so faulty operations stay within their functions and can be tested easily.
- You can also write functions inside of functions. If in defining your function you find replication (like in our number summary function, the na.rm), then this technique may be useful.
- Remember, R reads top to bottom. Therefore, it is good practice to list all of your function definitions at the top of your source code, so you don't have to worry about defining your functions in the middle of a script.
- Try to learn functions in R that take functions as parameters/arguments (like the lapply() function). It can help you see more outputs in your console and help you save complex outputs as objects in your code!
- I hope this helps you out, and I hope you can see the usefulness of functions in R. It really is a nice thing when you put it into practice!

Lecture 6 - Statistical Hypothesis Testing

Statistical Hypothesis Testing

In this lecture, we'll cover the basics of supporting your results empirically by using what we call statistical hypothesis testing. If you've ever heard the term "statistically significant," this is what we are talking about! Statistical hypothesis testing allows us to take our data (ex: the means in a dataset) and determine how extreme the phenomena we observe actually are!

Some definitions

- Statistical Hypothesis: a broader assumption/claim made by the researcher about the population of interest
  - Ex: A principal at a school thinks her students score an average of 7/10 in exams.
- Null hypothesis: The assumption/claim about the larger population
  - Ex: The mean score of students at this school is 7/10.
- Alternative hypothesis: Some proposition made by the researcher to test against the null hypothesis. This is "valid" if there is evidence in favor of the alternative hypothesis from observed data.
  - Ex: The mean score of students at this school is not 7/10.
- P-Value: The probability of finding the observed or more extreme results when the null hypothesis is true
  - Ex: From my sample of 30 students, there is a 4.2% chance that I get my observed mean of 6.4/10 or something more extreme (further from 7/10) if the actual mean score of students at this school is 7/10. My p-value is 0.042 in this example.
- Level of significance: This is how confident you want to be in your results, or basically what p-value you are comfortable with as the benchmark for rejecting the null hypothesis. Convention is 0.05 (people think this is arbitrary and it is, but we are generally comfortable with this threshold. A p-value threshold selection should depend on your tolerance for error types, which are discussed below).
  - Ex: I conduct the test of the school's mean test scores at a significance level of 0.05. So if my p-value is less than 0.05, I reject the null hypothesis and say I have significant evidence that the mean test scores of students at this school is not actually 7/10.
- Type I error: This comes right from the significance level; it's the probability that the null hypothesis is true when we actually rejected it from our test. It is equal to the significance level.
  - Ex: In my test above, I have a 5% chance of the null being true when I rejected it. So there is a 5% chance that the actual mean test scores of students at this school is 7/10 when I reject the claim (or, more precisely, the p-value is the % chance, or 4.2%. By declaring the 5% threshold, that is the zone where we could encounter a type I error).

- Type II error: The case in which you accepted (or failed to reject) a false null hypothesis. The probability of not making this mistake is known as the power of a test, which involves some calculations.
    - Ex: Say my p-value above was greater than 0.05, so I failed to reject the null hypothesis even though the mean test scores are actually not equal to 7/10.

Conditions

We can't conduct statistical hypothesis tests in every case. Each test has its separate conditions (you'll see why), but generally here are some conditions to think about when performing these tests:

- If you sample, it must be random. This makes sure there are no confounding variables in your sample data and you are not cherry picking for results. Again, some uncertainty needs to be there.
- Your observations should be independent or have no effect on subsequent observations. It is often difficult to control for this, but we try our best in our experimental setup!
- The data must come from a normal distribution or a solid sample size (generally, by the CLT, over 30).

Why sample? It allows our statistical computations and assumptions to be valid, and IRL, sampling is more efficient because it is virtually impossible (and costly) to get the data for the entire population of interest.

Notes

^These are a lot of definitions and conditions, so it is easier and more intuitive to learn about this as we go through some exercises.

General workflow for hypothesis testing:

- Check conditions for the test
- State the hypothesis, pick a significance level
- Analyze sample data
- Interpret results

One Sample T-test

Now that we've covered the basics and talked conceptually, we can start with some exercises. There are many tests R can conduct (we'll go through some), but the first one is called a one sample t-test. It takes one sample, calculates the mean value, and evaluates it against a hypothesized value.

Let's set up our data:

```
## Get sample data
set.seed(30)
x <- as.data.frame(rnorm(100, mean = 6.4, sd = 1))
```

We define our sample data (assumptions taken care of), x, as a normal distribution with a mean of 6.4 and a standard deviation of 1. Say this is our sample from the school - is it fair to say that the mean scores of the school are equal to 7?

```
## Run one sample t-test against mean 7
t.test(x, mu = 7)
```

```
##
##  One Sample t-test
##
## data:  x
## t = -6.3811, df = 99, p-value = 5.684e-09
## alternative hypothesis: true mean is not equal to 7
## 95 percent confidence interval:
##  6.114636 6.534644
## sample estimates:
## mean of x
##   6.32464
```

Look at this nice output! R already set up our hypotheses (alternative is mean is not equal to 7, null being equal to 7). R gives us a 95% confidence interval of the true mean based on this sample, it also gives us the observed sample mean, and we get the corresponding p-value! Note that the t-score and the corresponding degrees of freedom (df) are given. That's how the p-value is calculated, from the t-distribution (basically a normal curve with wider tails so it accounts for more wonky phenomena).

We can also conduct this t-test to give it a direction. What if the principal of this school wanted to claim that their school's test scores are better than a rival school's test scores of 6.8? How do our hypotheses change? We say the null hypothesis is that mean test scores are 6.8 (can never have a null hypothesis with direction), but the alternative hypothesis is that the school's mean test scores are greater than 6.8. We conduct this below:

```
## One sample t-test, alternative as greater than
t.test(x, mu = 6.8, alternative = 'greater')
```

```
##
##  One Sample t-test
##
## data:  x
## t = -4.4914, df = 99, p-value = 1
## alternative hypothesis: true mean is greater than 6.8
## 95 percent confidence interval:
##  6.148909      Inf
## sample estimates:
## mean of x
##   6.32464
```

That p-value is astronomical, so we don't have evidence to support our claim of greater test scores. Notice that 'inf' on the confidence interval - that's normal, because we are really only testing the lower bound here. This p-value is high, so let's flip around the signs to see if this principal's test scores are actually LESS than the rival school.

```
## One sample t-test, alternative as less than
t.test(x, mu = 6.8, alternative = 'less')
```

```
##
##  One Sample t-test
##
## data:  x
## t = -4.4914, df = 99, p-value = 9.59e-06
## alternative hypothesis: true mean is less than 6.8
## 95 percent confidence interval:
##      -Inf 6.500371
## sample estimates:
## mean of x
##   6.32464
```

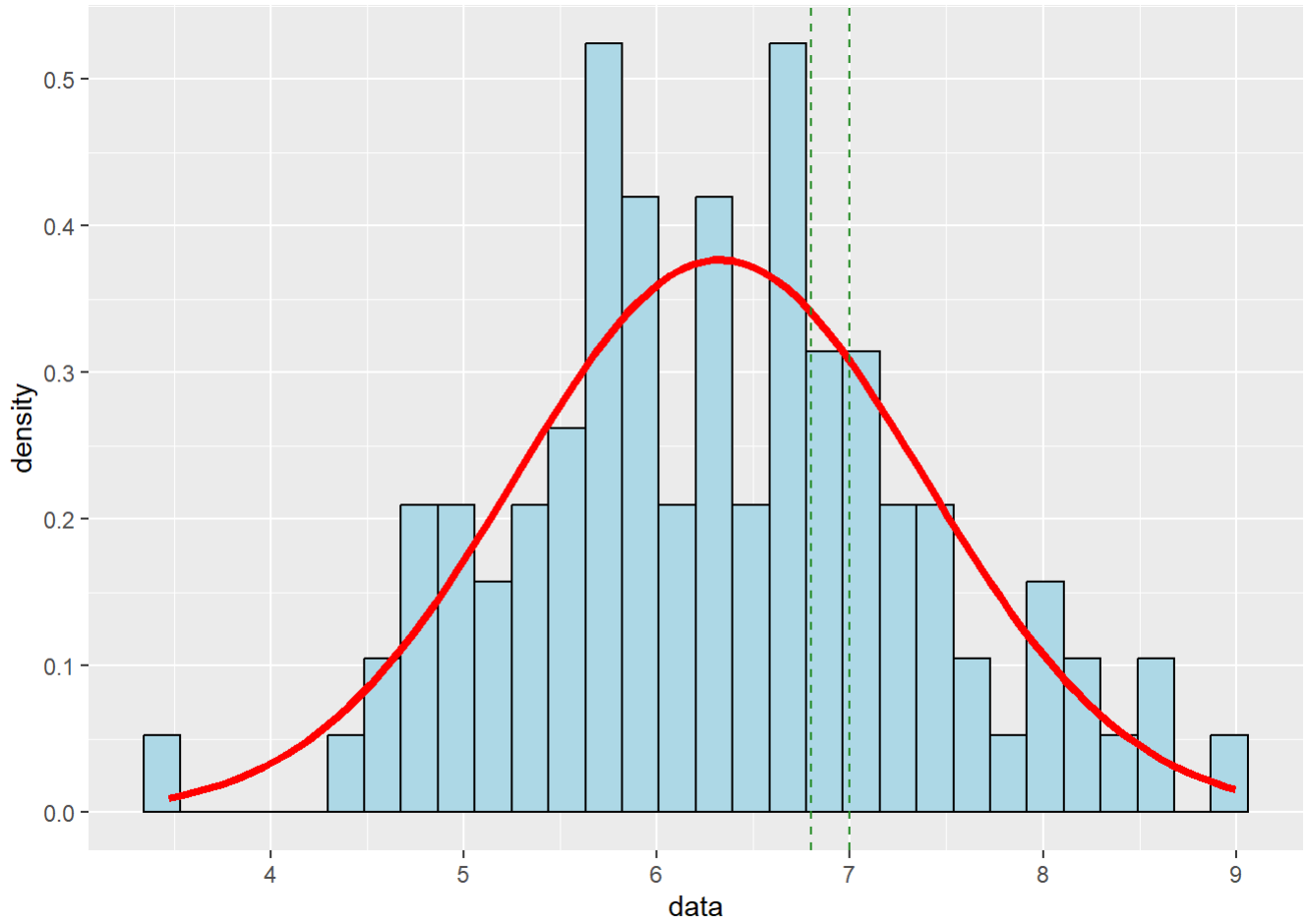
Uh oh, it's significant that the principal school's test scores are lower than the rival school. Ouch.

If we put our data viz skills to the test, you can see why we get the results that we do…

```
## Visualize distributions
library(ggplot2)
colnames(x) <- "data"

ggplot(x, aes(x=data)) + geom_histogram(aes(y = after_stat(density)), color = 'black', fill = 'l
ightblue') +
  stat_function(fun = dnorm, args = list(mean = mean(x$data), sd = sd(x$data)), lwd = 1.5, color
= 'red') +
  geom_vline(xintercept = 6.8, linetype = "dashed", color = "forestgreen") +
  geom_vline(xintercept = 7, linetype = "dashed", color = "forestgreen")
```

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



The dashed lines represent the hypothesized values (null hypotheses), the histogram is our sample data, and the curve is the distribution in which our data came from (usually it's an estimate we build from the sample data). You can see why we made the conclusions we did before!

Two Sample T-test

In the above example, we already had the average of the rival school. But let's say we didn't! Then we would have two samples in which we'd need to compare.

```
## Two-sample t-test
# Generate data
set.seed(30)
y <- as.data.frame(rnorm(100, mean = 6.8, sd = 0.5))
```

In a two-sample t-test, the null hypothesis is that the difference in the two population means is 0 (or they are equal), and the alternative hypothesis is that the difference is not 0 (or the means are not equal). Our test will account for differences in variance between the two groups, so our assumptions pass. Let's let R tell us the answer here!

```
# Test for differences
t.test(x,y)
```

```
##
##  Welch Two Sample t-test
##
## data:  x and y
## t = -3.6988, df = 145.59, p-value = 0.0003063
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.6715453 -0.2038145
## sample estimates:
## mean of x mean of y
##   6.32464   6.76232
```

We do have evidence that the means between the two schools' test scores are significant. But in what direction? Conducting that test below:

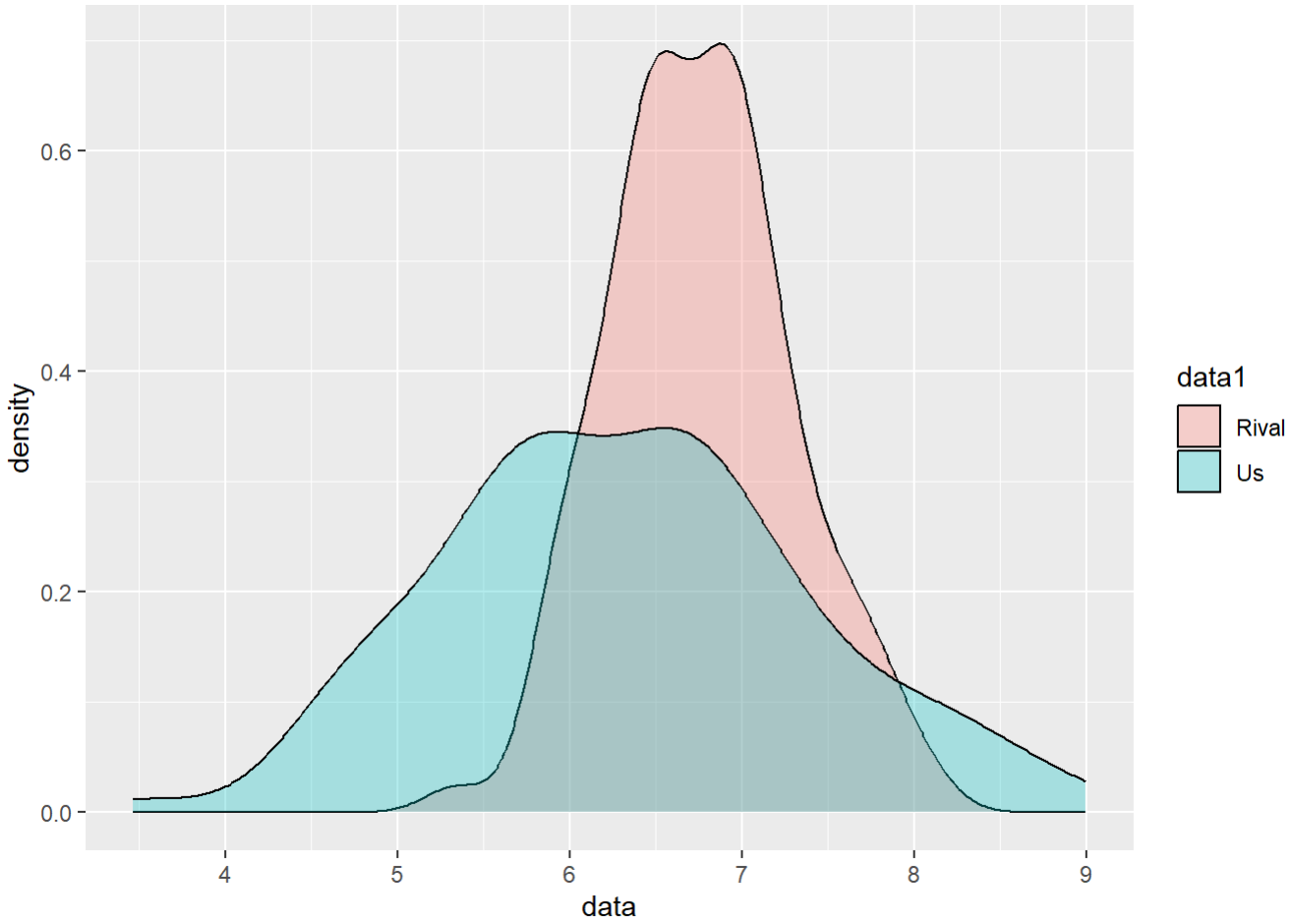
```
## Two sample t-test, one-tailed
t.test(x,y, alternative = 'less')
```

```
##
##  Welch Two Sample t-test
##
## data:  x and y
## t = -3.6988, df = 145.59, p-value = 0.0001531
## alternative hypothesis: true difference in means is less than 0
## 95 percent confidence interval:
##        -Inf -0.2417989
## sample estimates:
## mean of x mean of y
##   6.32464   6.76232
```

We do have enough evidence to say that the rival school's mean test scores are greater than the principal's school. This is an example of the difference between what we call a one-tailed (direction) and two-tailed (no direction, just testing equality) test.

```
# Visualize distributions
colnames(y) <- "data"
Schools <- rbind(x,y)
Schools$data1 <- ifelse(seq.int(nrow(Schools)) < 101, "Us", "Rival")
Schools$dataX <- ifelse(Schools$data1 == "Us", Schools$data, NA)
Schools$dataY <- ifelse(Schools$data1 == "Rival", Schools$data, NA)

ggplot(data = Schools, aes(x = data, group = data1, fill = data1)) + geom_density(alpha = 0.3)
```



A visual aid helps us see the distributions of the two schools. We can see that generally, most of the red curve is to the right of the blue curve, and our test confirms this result empirically. You can also tell the means are different, which confirms our result from the first test.

Proportion Testing

Proportion testing takes the same idea as t-testing but works with the normal distribution (z) rather than the t distribution (t). This is because the standard deviation of the proportion is a function of itself, so it's a bit more reliable of an estimate where we don't need wide tails.

For our example, we are going to simulate a coin flipping game, where we have two teams flip a coin 100 times. Whichever team accumulates more "heads" flips in the game wins! Assumptions will pass if we have a binary outcome, enough sample, and independence. We do!

Let's set up our data. We'll randomize results by "sampling" a coin with its respective probabilities of heads and tails. However, notice something: team 2 is cheating! They are using a weighted coin that is more likely to land on heads than tails. Let's see if we can catch them using our hypothesis testing! Code below sets up our data:

```
## Coin initialization
coin <- c("heads", "tails")
n <- 100

## Team 1, fair coin
Prob1 <- c(0.5,0.5)
set.seed(30)
samps1 <- as.data.frame(table(sample(coin, n, replace = TRUE, prob = Prob1)))

## Team 2, weighted coin
Prob2 <- c(0.6,0.4)
set.seed(30)
samps2 <- as.data.frame(table(sample(coin, n, replace = TRUE, prob = Prob2)))

## See data
samps1
```

```
##    Var1 Freq
## 1 heads   39
## 2 tails   61
```

```
samps2
```

```
##    Var1 Freq
## 1 heads   68
## 2 tails   32
```

R needs a summary table of our frequencies so we can place the proper inputs into its prop.test() function. Done here:

```
# Compare
summary <- cbind(samps1, samps2)
summary <- summary[-c(3)]
summary
```

```
##    Var1 Freq Freq.1
## 1 heads   39     68
## 2 tails   61     32
```

Whoa, that's a big win for team 2! Let's evaluate how extreme these results are - let's call our two-proportion test!

```
# Significant win?
prop.test(x = c(39,68), n = c(100,100), alternative = "less", conf.level = 0.95)
```

```
##
##  2-sample test for equality of proportions with continuity correction
##
## data:  c(39, 68) out of c(100, 100)
## X-squared = 15.757, df = 1, p-value = 3.601e-05
## alternative hypothesis: less
## 95 percent confidence interval:
##  -1.0000000 -0.1689876
## sample estimates:
## prop 1 prop 2
##   0.39   0.68
```

The p-value of this test is very low, so we did witness a statistically significant event. We can also use the confidence interval to show that the difference in proportions is quite large. This means that it is highly unlikely that the coins were fair for both sides. Let's take a look at team 2's results to see if they used a weighted coin:

```
# Cheated? One Sample proportion
prop.test(x = 68, n = 100, p = 0.5, alternative = "greater")
```

```
##
##  1-sample proportions test with continuity correction
##
## data:  68 out of 100, null probability 0.5
## X-squared = 12.25, df = 1, p-value = 0.0002326
## alternative hypothesis: true p is greater than 0.5
## 95 percent confidence interval:
##  0.5942311 1.0000000
## sample estimates:
##    p
## 0.68
```

Notice the difference in the syntax for a one-sample proportion test (one value for x, n, and inclusion of p - hypothesized value). We see that we have convincing evidence that team 2 did something funky with the coin. Uh oh.

Chi-Sq Test for Association + Goodness of Fit Test

The last set of tests we will cover involve the Chi-Sq distribution. The Chi-Sq distribution comes from a sum of independent squared random normal variables - it really has no parameters other than degrees of freedom (df), so it is useful for many statistical analyses.

We can use Chi-Sq tests for categorical variables. We will first use the test for association to determine if there is an association between variables (null is if the variables are not associated, alternative is that the variables are associated). We will then use the goodness of fit test to determine how closely our data resembles some real-world benchmark (null is if observed values are not different than expected values, alternative is if observed values are different than expected).

For the Chi-Sq test for association, we need a contingency table, in that the categories for one variable are in the rows and the categories for another variable are in the columns. Contingency tables are also referred to as "cross-tabs" or "two-way tables," which you may have seen before.

In our example, we'll use the iris dataset to see if there is first an association of sepal length size to species type. We will then test it against an expected distribution of sepal length size to check results.

First, the data and a visual aid:

Source: https://statsandr.com/blog/chi-square-test-of-independence-in-r/ (https://statsandr.com/blog/chi-square-test-of-independence-in-r/)

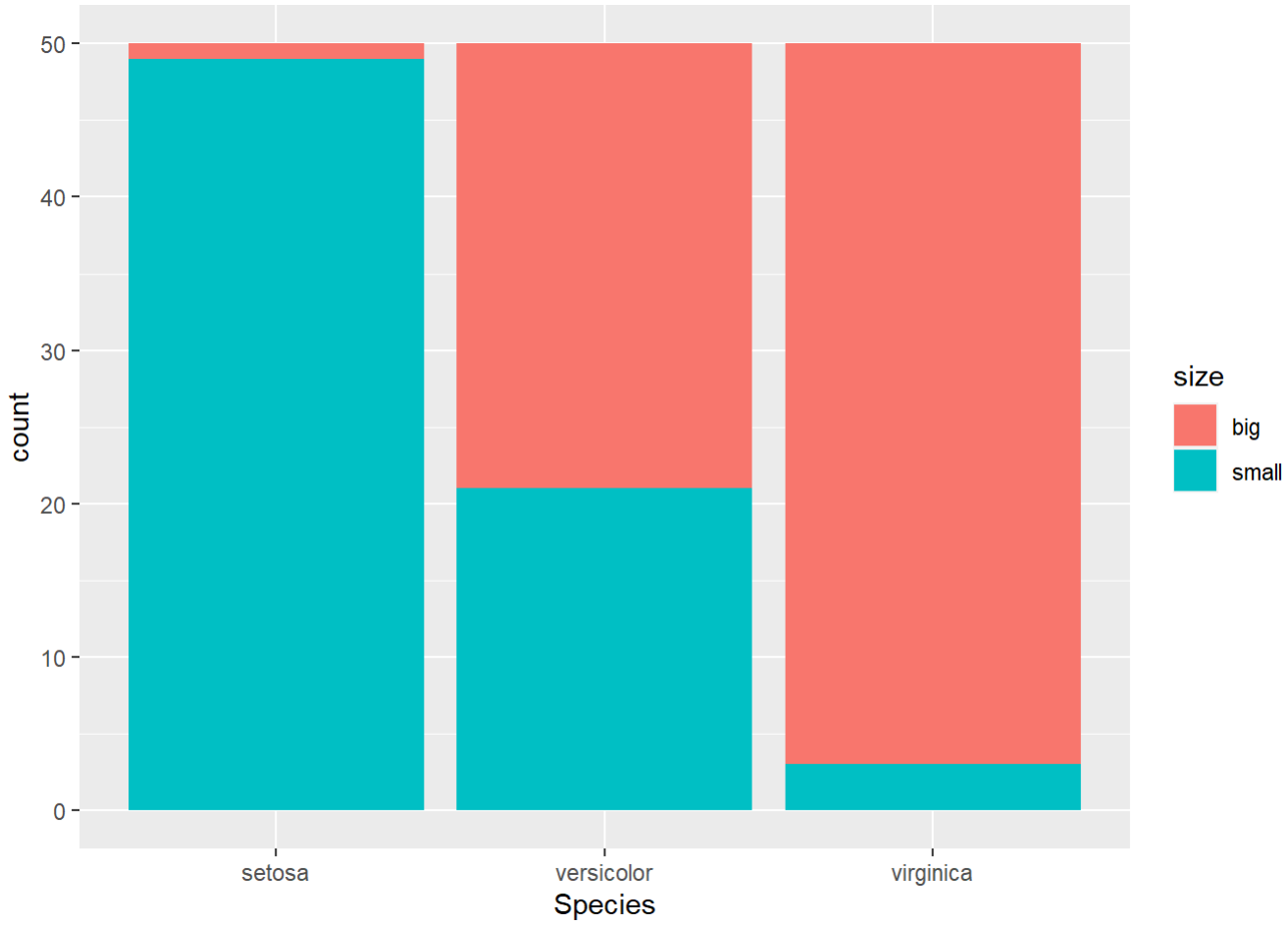
```
## Chi-sq test for association
data <- iris

data$size <- ifelse(data$Sepal.Length < median(data$Sepal.Length),
                    "small", "big"
)

# Generate contingency table
Table <- table(data$Species, data$size)
Table
```

```
##
##              big small
##   setosa       1    49
##   versicolor  29    21
##   virginica   47     3
```

```
# Visual Aid
ggplot(data) + aes(x = Species, fill = size) +
  geom_bar()
```



You can see the large differences between sepal length and species. This probably means there is an association between our two variables - sepal length and species. Let's see!

```
# Run chi-sq test
test <- chisq.test(Table)
test
```

```
##
##  Pearson's Chi-squared test
##
## data:  Table
## X-squared = 86.035, df = 2, p-value < 2.2e-16
```

```
test$expected
```

```
##
##                  big    small
##   setosa     25.66667 24.33333
##   versicolor 25.66667 24.33333
##   virginica  25.66667 24.33333
```

Notice how we do get a significant result. We can also save our test as an object and access its values; this is where I pull up the values in the case that there was no association. Notice how different our observed values are from these! We have evidence for an association between sepal length and species.

Using the Chi-Sq distribution, we can also test how closely the observed values follow some distribution. For instance, let's say we expected 1/6 of big sepal lengths coming from setosas, 1/3 coming from versicolors, and 1/2 coming from virginicas. We can test our observed values against the hypothetical with a goodness of fit test:

```
# Test against expected probability of big per species
big <- c(1, 29, 47)
test <- chisq.test(big, p = c(1/6, 1/3, 1/2))
test
```

```
##
##  Chi-squared test for given probabilities
##
## data:  big
## X-squared = 13.221, df = 2, p-value = 0.001346
```

```
test$expected
```

```
## [1] 12.83333 25.66667 38.50000
```

Notice how we add the probability vector, and the expected values of the test change. We can see that there is a significant difference between our observed values versus the expected result! Note: condition for the GOF test is that all expected counts must be >5 (which they are here).

Notes

This concludes our lecture on statistical hypothesis testing; my goal was to give you an overview of how tests work and how they are relatively easy to run in R. Some general notes while you perform these analyses…

- Don't forget to check your conditions! We were fine for our datasets, but IRL, it is important that you validate that your conditions are satisfied before you run the tests.
- Use a visual aid as often as possible. This will help validate your results and look at your sample data for context.
- Confidence intervals: look them up. We discuss them briefly in lecture, but they are foundational to statistical tests and an alternative way to test hypotheses.
- Your answers are not certain! Interpret your results and know that your results are not definitive. The data you collect is inherently uncertain. Therefore, it is important you understand that tests do a good job of extracting meaning from uncertainty but do not eliminate it from your analysis.
- Some other statistical tests to check out…
    - One way ANOVA test - for a categorical variable interacting with an interval variable (numeric)
    - McNemar test - for binary outcomes - often used in UX (i.e. task completion rates)
    - Not really a test, but correlation matrix - this provides a useful snapshot of the relationships in your data

Extra Resources and GitHub

Thank you very much for attending my R modules! I really appreciate your time and investment in yourself to learn a new skill! I encourage you to take on datasets and create your own projects just like we did together in the course. If you have any questions or want any coding/career advice, please feel free to reach out to Will Calandra at wcc44@georgetown.edu (mailto:wcc44@georgetown.edu)! I am always available to help out a fellow peer in the data community :)

Extra Resources

While we covered the basics and got a taste of advanced modeling during the R module, below is a list of resources I encourage you to explore as you continue your journey in the R programming language. As with anything, continual practice leads to the best results!

- R for Data Science - Hadley Wickham: https://r4ds.had.co.nz/ (https://r4ds.had.co.nz/)
    - A great free book that walks you through the basics of coding and modeling in R!
- RStudio Cheatsheets: https://www.rstudio.com/resources/cheatsheets/ (https://www.rstudio.com/resources/cheatsheets/)
    - The most essential information compiled all in one place (and on one page)! You can get very far with these cheatsheets…
- Tidyverse website: https://www.tidyverse.org/packages/ (https://www.tidyverse.org/packages/)
    - The tidyverse is critical to learn, but good news, it makes life so easy! Here you will find documentation to all of its libraries
    - In particular, use the link for ggplot here for data visualization help: https://ggplot2.tidyverse.org/ (https://ggplot2.tidyverse.org/).
- YaRrr! The Pirate's Guide to R: https://bookdown.org/ndphillips/YaRrr/ (https://bookdown.org/ndphillips/YaRrr/)
    - This book attempts to make R funny (they try pretty hard, cringe), but chapter 5 and beyond covers PLENTY of both beginner and advanced techniques - there is great work on data structures
- RStudio Books: https://www.rstudio.com/resources/books/ (https://www.rstudio.com/resources/books/)
    - More books on R! Start with the first one though (R for Data Science).
- R resources - Paul Vanderlaken: https://paulvanderlaken.com/2017/08/10/r-resources-cheatsheets-tutorials-books/#introductory (https://paulvanderlaken.com/2017/08/10/r-resources-cheatsheets-tutorials-books/#introductory)
    - Comprehensive list of more than you could ask for!
- DataCamp, Coursera, freeCodeCamp, etc.
    - A bunch of online free resources for video learners! Just search R courses and find the teacher you like!!
- Kaggle: https://www.kaggle.com/ (https://www.kaggle.com/)
    - A great online community full of datasets and coding solutions available for download!
    - Create a free account and filter datasets for ".csv" files
- UCI Machine Learning Repository - https://archive.ics.uci.edu/ml/index.php (https://archive.ics.uci.edu/ml/index.php)
    - Another great, free website with datasets! Use the filters cleverly to get the type of dataset you want for your analysis!
- KDNuggets' list of datasets - https://www.kdnuggets.com/datasets/index.html (https://www.kdnuggets.com/datasets/index.html)
    - A huge list of datasets here - just google which one you'd like and have fun!

GitHub Instructions

A step-by-step guide to submit files through GitHub…

1 - Create a GitHub account and share your username with us (with Will Calandra, wcc44@georgetown.edu (mailto:wcc44@georgetown.edu))! We will then invite you to our repository so you can make changes (i.e. upload your files).

2 - Download GitHub Desktop! This will make the process easier than using your terminal. This link walks you through the steps to get started: https://docs.github.com/en/desktop/installing-and-configuring-github-desktop/overview/getting-started-with-github-desktop (https://docs.github.com/en/desktop/installing-and-configuring-github-desktop/overview/getting-started-with-github-desktop)

3 - Once installed and authorized, open GitHub Desktop and go to “File,” “Clone Repository,” and change the local path to an empty folder in your directory (this part is critical: if you are getting the “git can only clone to empty folders” case, you need to change the folder name on your local machine). Click “Clone,” and if successful, in GitHub Desktop, your should be in our repository!

4 - Now, some terminology. “Cloning” a repo means that you took our repository from online and downloaded it to your computer. Notice now in your GitHub Desktop we have the current branch as main - this is good. A “branch” in GitHub is just like a folder on your computer… it’s a shared directory online. We will only be using the main branch, but tech companies can get organized by having different branches for different things. Notice how you also have “fetch origin” in GitHub Desktop. The repo will automatically update online, but not on your computer. As such, to get the most recent version, click “fetch origin” every time you open GitHub Desktop.

5 - Now, we’ll connect an “external editor,” or just another way to say IDE to GitHub Desktop. IDE (or integrated development environment) is a fancy way to say what platform you are coding in. We will use IDEs for their file management features to make this process quick and easy. I recommend downloading Visual Studio Code (link: https://code.visualstudio.com/download (https://code.visualstudio.com/download)); the rest of the instructions will be based off of that.

6 - Open Visual Studio Code (VSCode). In a new window, click “Clone Git Repository.” Then click “GitHub,” “Allow,” and authorize VSCode to use your GitHub account. Now close VSCode and go back to GitHub Desktop.

7 - In GitHub Desktop, click “File”, “Options,” “Integrations,” and change the external editor to Visual Studio Code.

8 - Now, back in the home page of GitHub Desktop, click “Open in Visual Studio Code” under “Open the repository in your external editor.”

9 - Now, in VSCode, notice “Subs” in your top left! Hover over it, click the “New Folder,” and add a folder with your first initial last name (ex: wcalandra). Now, go to “File,” “Open File,” and select the file you want to publish (make sure this file’s name is your lastname_file for example). When opened, it should show up at the top of your VSCode as a tab. Click and drag this tab into your folder. Then the file should be in there, and on the left in VSCode, both your folder and the file should be highlighted green!

10 - Go back to GitHub Desktop. Notice the changes? This means that it tracked your changes to the repository. Now it’s time to authorize these changes, or what Git calls, “make a commit.” Next to your GitHub logo on the left in GitHub Desktop, type what you are doing (ex: committing first file), and feel free to type in the description as well if needed (this part is optional). Then click “commit to main,” which will formally make the change in the online repository (in the main branch).

11 - However, it is not completed until you click “Push Origin.” This pushes your changes to the online repo. If done correctly, go to your browser and look at our repo. You should see your changes uploaded successfully! Congrats!

12 - Pretty neat, right! Collaboration, code review, maintenance, etc. are all done in Git by many companies around the world. Familiarize yourself with Git and its terminology - it is a very useful skill IRL! If you have any questions, comments, or are interested in not using GitHub Desktop but trying to connect your terminal with our repo instead, reach out to Will Calandra (netid: wcc44).