Advanced Sports Data Analysis

Using machine learning to make predictions in sports

Matt Waite

1/4/23

Table of contents

1 Introduction

The 2020 college football season, for most fans, will be one to forget. The season started unevenly for most teams, schedules were shortened, non-conference games were rare, few fans saw their team play in person, all because of the COVID-19 global pandemic.

For the Nebraska Cornhuskers, it was doubly forgettable. Year three of Scott Frost turned out to be another dud, with the team going 3-5. A common refrain from the coaching staff throughout the season, often after disappointing losses, was this: The team is close to turning a corner.

How close?

This is where modeling comes in in sports. Using modeling, we can determine what we should **expect** given certain inputs. To look at Nebraska's season, let's build a model of the season using three inputs based on narratives around the season: The offense struggled to score, the offense really struggled with turnovers, and the defense improved.

The specifics of how to do this will be the subject of this whole book, so we're going to focus on a simple explanation here.

First, we're going to create a measure of offensive efficiency – points per yard of offense. So if you roll up 500 yards of offense but only score 21 points, you'll score .042 points per yard. A team that gains 250 yards and scores 21 points is more efficient: they score .084 points per yard. So in this model, efficient teams are good.

Second, we'll do the same for the defense, using yards allowed and the opponent's score. Here, it's inverted: Defenses that keep points off the board are good.

Third, we'll use turnover margin. Teams that give the ball away are bad, teams that take the ball away are good, and you want to take it away more than you give it away.

Using logistic regression and these statistics, our model predicts that Nebraska is actually worse than they were: the Husker's **should** have been 2-6. Giving the ball away three times and only scoring 28 points against Rutgers **should** have doomed the team to a bad loss at the end of the season. But, it didn't.

So how much of a corner would the team need to turn?

With modeling, we can figure this out.

What would Nebraska's record if they had a +1 turnover margin and improves offensive production 10 percent?

As played, our model gave Nebraska a 32 percent chance of beating Minnesota. If Nebraska were to have a +1 turnover margin, instead of the -2 that really happened, that jumps to a 40 percent chance. If Nebraska were to improve their offense just 10 percent – score a touchdown every 100 yards of offense – Nebraska wins the game. Nebraska wins, they're 4-4 on the season (and they still don't beat Iowa).

So how close are they to turning the corner? That close.

1.1 Requirements and Conventions

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

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

Going forward, you'll see passages like this:

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

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

1.2 About this book

This book is the collection of class materials for the author's Advanced Sports Data Analysis class at the University of Nebraska-Lincoln's College of Journalism and Mass Communications. There's some things you should know about it:

- It is free for students.
- The topics will remain the same but the text is going to be constantly tinkered with.
- What is the work of the author is copyright Matt Waite 2023.
- The text is Attribution-NonCommercial-ShareAlike 4.0 International Creative Commons licensed. That means you can share it and change it, but only if you share your changes with the same license and it cannot be used for commercial purposes. I'm not making money on this so you can't either.

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

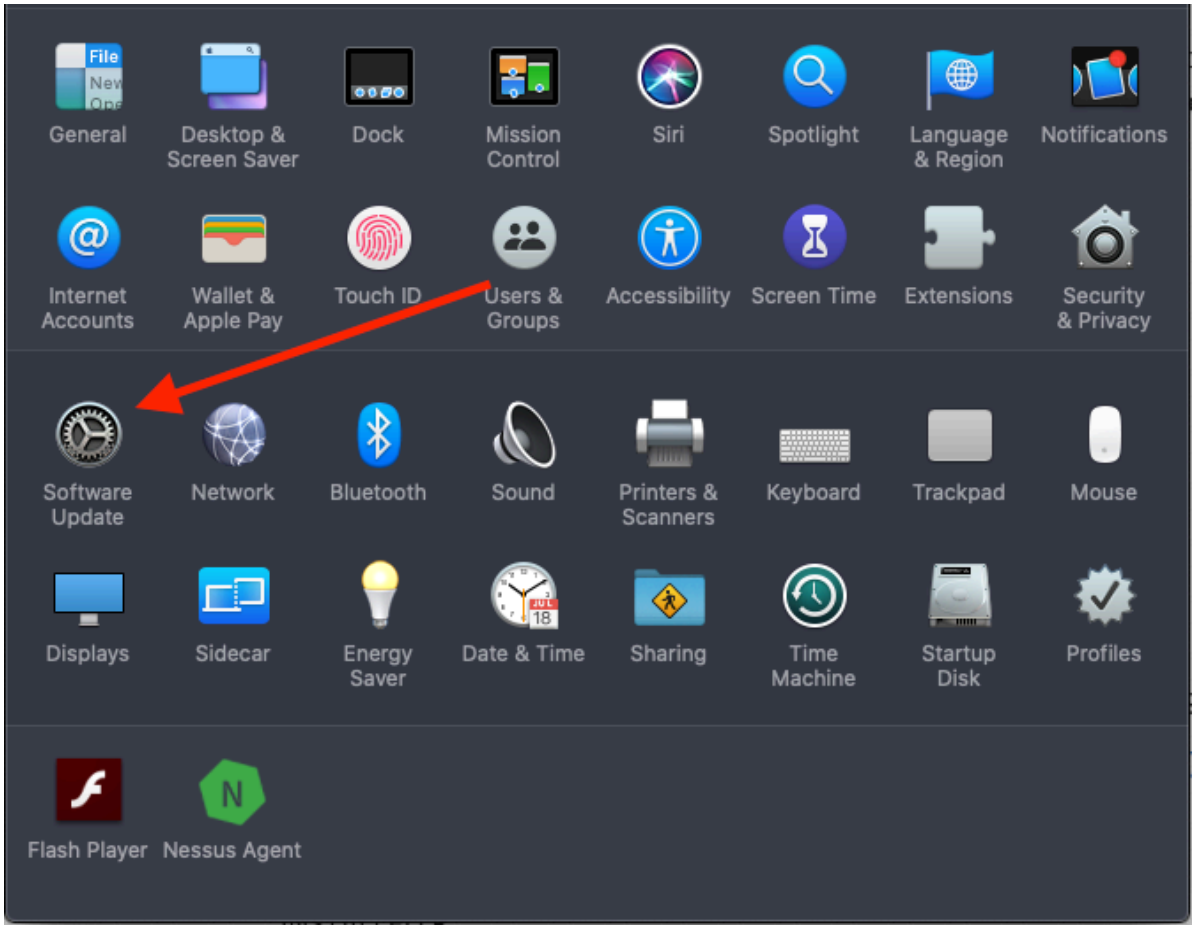
2 Installations

You're going to do things most of you aren't used to doing with your computer in this class. In order to do that, you need to clean up your computer. I've seen what your computer looks like. It's disgusting.

2.0.1 Part 1: Update and patch your operating system

On a Mac:

1. Open System Preferences. Then click on Software Update:



2. Check and see if you have the latest version of the Mac OS installed. If your computer says "Your Mac is up to date", then you're good to go, regardless of what comes next.

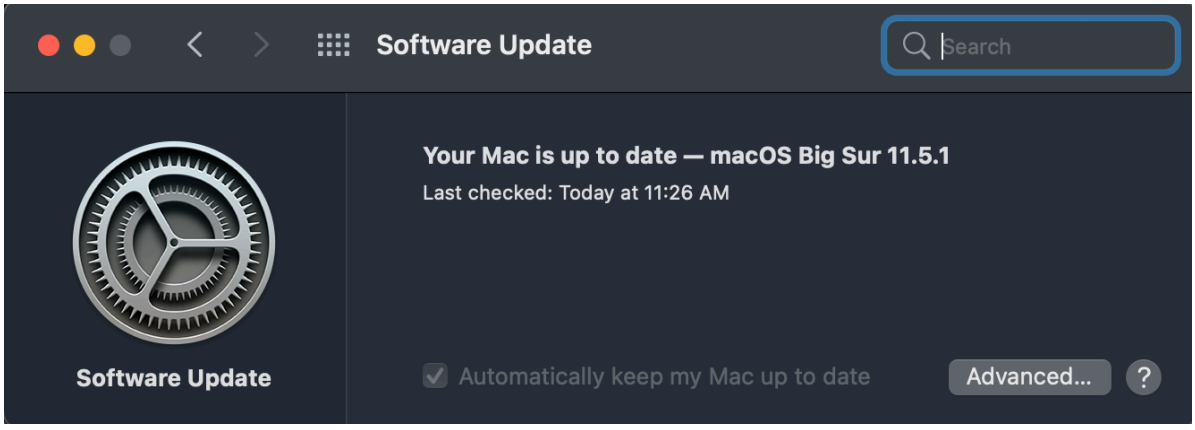
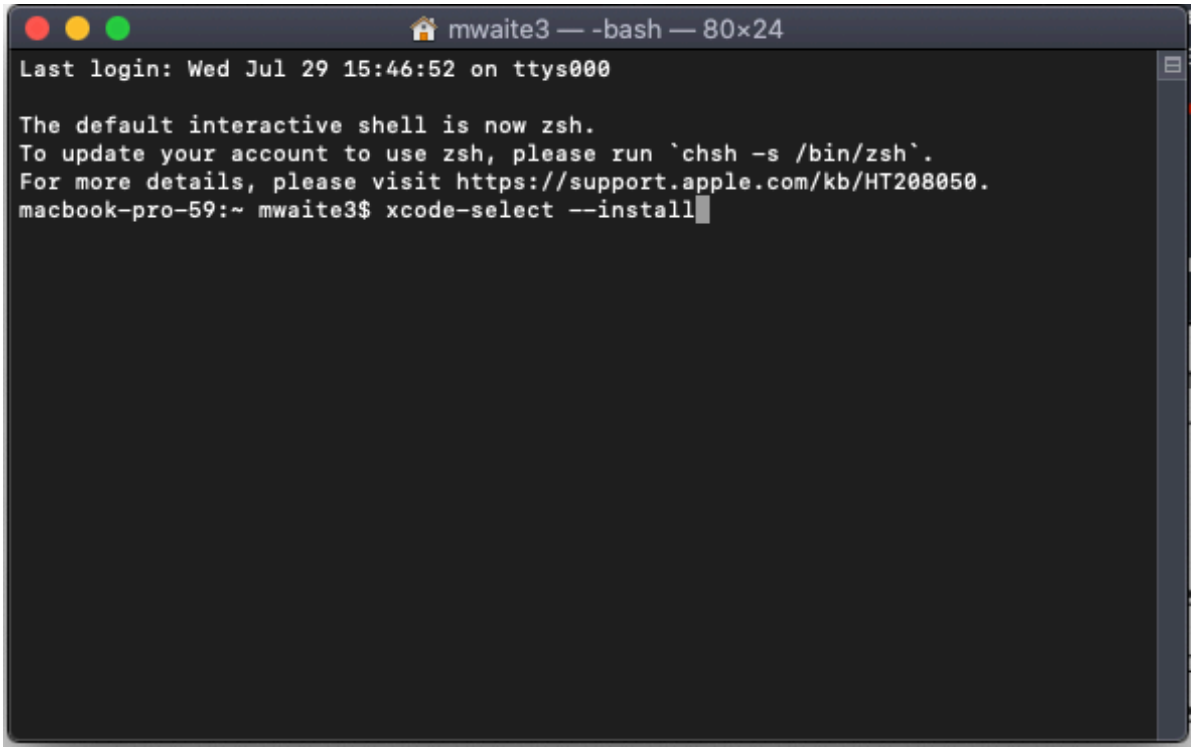


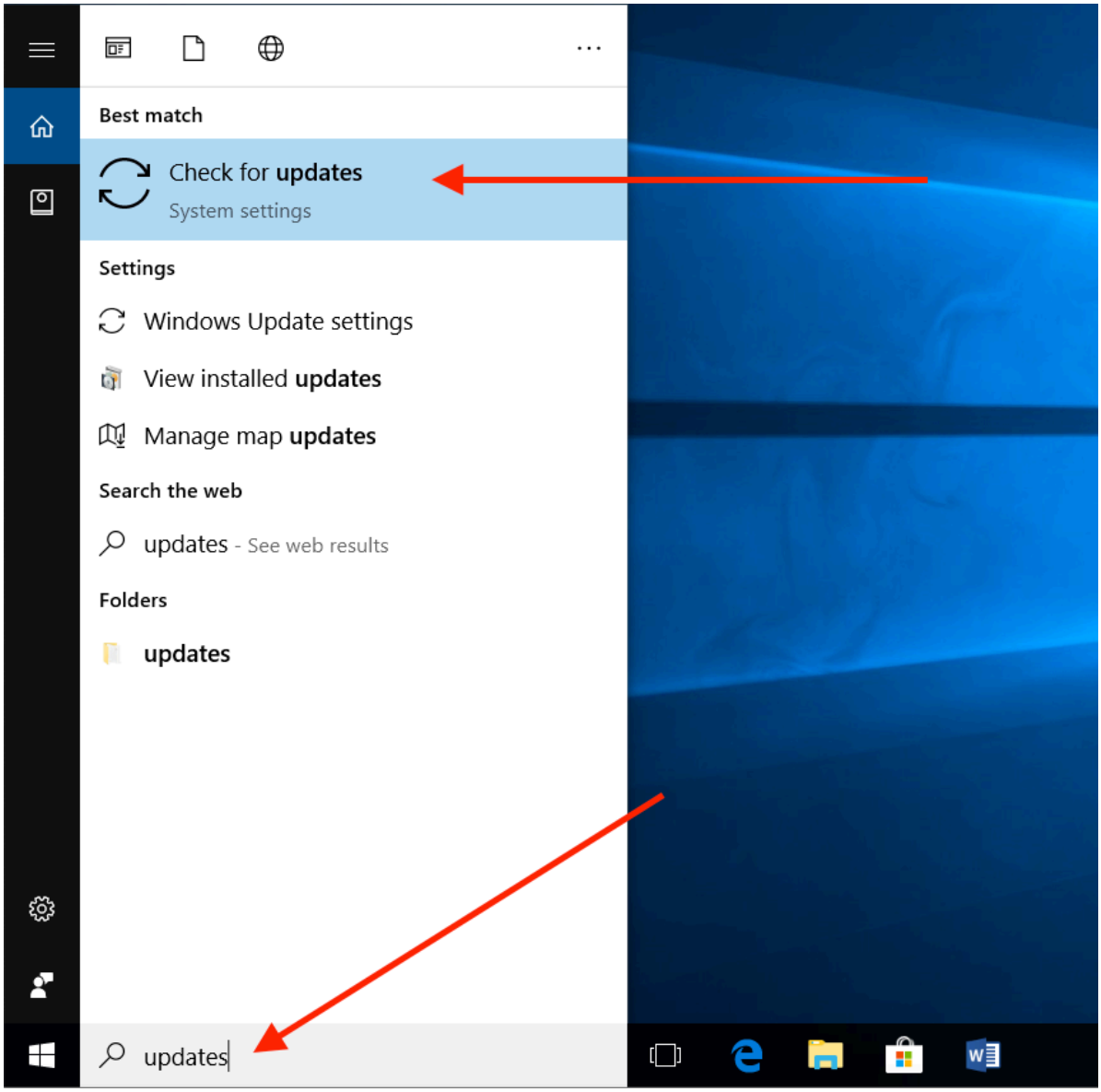
Figure 2.1: The latest version of the Mac OS is called Ventura.

3. If you aren't on Ventura and you can update to it, you should do it. This will take some time – hours, so don't do it when you need your laptop – but it's important for you and your computer to stay up to date on operating systems.
4. When you're done, make sure you click the Automatically keep my Mac up to date box and install those updates regularly. Don't ignore them. Don't snooze them. Install them.
5. With an up-to-date operating system, now install the command line tools. To do this, click on the magnifying glass in the top right of the screen and type terminal. Hit enter – the first entry is the terminal app.
6. In the terminal app, type `xcode-select --install` and hit enter. Let it run.

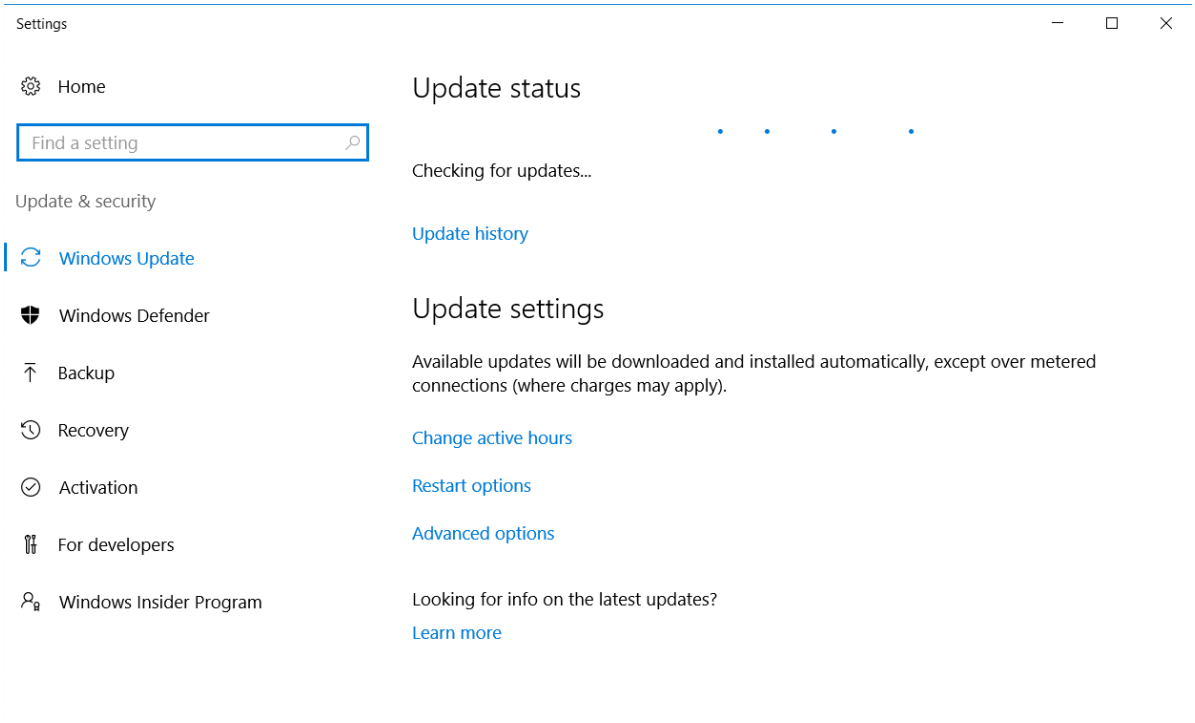


On Windows:

1. Type Updates into the Cortana search then click Check for updates



2. After the search for updates completes, apply any that you have. Depending on if you'd done this recently or if you have automatic updates set, this might take a long time or go very quickly.



3. When you're done, make sure you set up automatic updates for your Windows machine and ins

2.0.2 Part 2: Install R and R Studio

On a Mac:

1. Download the latest version of R from CRAN for Mac.

The Comprehensive R Archive Network

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2020-06-22, Taking Off Again) R-4.0.2.tar.gz, read what's new in the latest version.
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

R for macOS

This directory contains binaries for a base distribution and packages to run on macOS. Releases for old Mac OS X systems (through Mac OS X 10.5) and PowerPC Macs can be found in the old directory.

Note: Although we take precautions when assembling binaries, please use the normal precautions with downloaded executables.

Package binaries for R versions older than 3.2.0 are only available from the CRAN archive so users of such versions should adjust the CRAN mirror setting (`https://cran-archive.r-project.org`) accordingly.

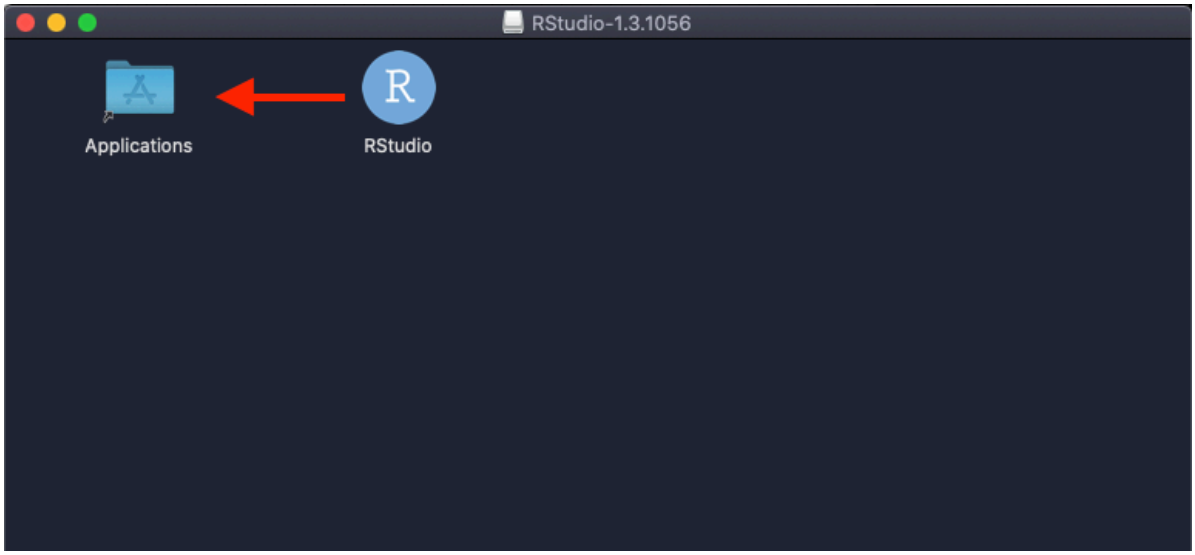
R 4.1.0 "Camp Pontanezen" released on 2021/05/19

Please check the SHA1 checksum of the downloaded image to ensure that it has not been tampered with or corrupted during the mirroring process. For example type
`openssl sha1 R-4.1.0.pkg`
in the *Terminal* application to print the SHA1 checksum for the R-4.1.0.pkg image. On Mac OS X 10.7 and later you can also validate the signature using
`pkgutil --check-signature R-4.1.0.pkg`

Latest release:

R-4.1.0.pkg (notarized and signed) SHA1-hash: df4d6fc17bbf6b7a27d4e015c0084d4bb6f7b428 (ca. 87MB)	**R 4.1.0** binary for macOS 10.13 (**High Sierra**) and higher, **Intel 64-bit** build, signed and notarized package. Contains R 4.1.0 framework, R.app GUI 1.76 in 64-bit for Intel Macs, Tcl/Tk 8.6.6 X11 libraries and Texinfo 6.7. The latter two components are optional and can be ommitted when choosing "custom install", they are only needed if you want to use the `tcltk` R package or build package documentation from sources. Note: the use of X11 (including `tcltk`) requires XQuartz to be installed since it is no longer part of OS X. Always re-install XQuartz when upgrading your macOS to a new major version. This release supports Intel Macs, but it is also known to work using Rosetta2 on M1-based Macs. For native Apple silicon arm64 binary see below. **Important:** this release uses Xcode 12.4 and GNU Fortran 8.2. If you wish to compile R packages from sources, you may need to download GNU Fortran 8.2 - see the tools directory.
R-4.1.0-arm64.pkg (notarized and signed) SHA1-hash: 7354c1b249cab9bafea6ae67c73563303a05fa17 (ca. 88MB)	**R 4.1.0** binary for macOS 11 (**Big Sur**) and higher, **Apple silicon arm64** build, signed and notarized package. Contains R 4.1.0 framework, R.app GUI 1.76 for Apple silicon Macs (M1 and higher), Tcl/Tk 8.6.11 X11 libraries and Texinfo 6.7. **Important: this version does NOT work on older Intel-based Macs.** The above installer package will be misidentified by Apple Installer as Intel architecture although it contains no Intel code. This means you may be asked to install Rosetta 2 even though it is not required. This issue has been fixed in the packaging for future R versions, so if you don't want to install Rosetta 2 please use R-4.1-branch big-sur arm64 installer from mac.R-project.org. Note: the use of X11 (including `tcltk`) requires XQuartz. Always re-install XQuartz when upgrading your macOS to a new major version.

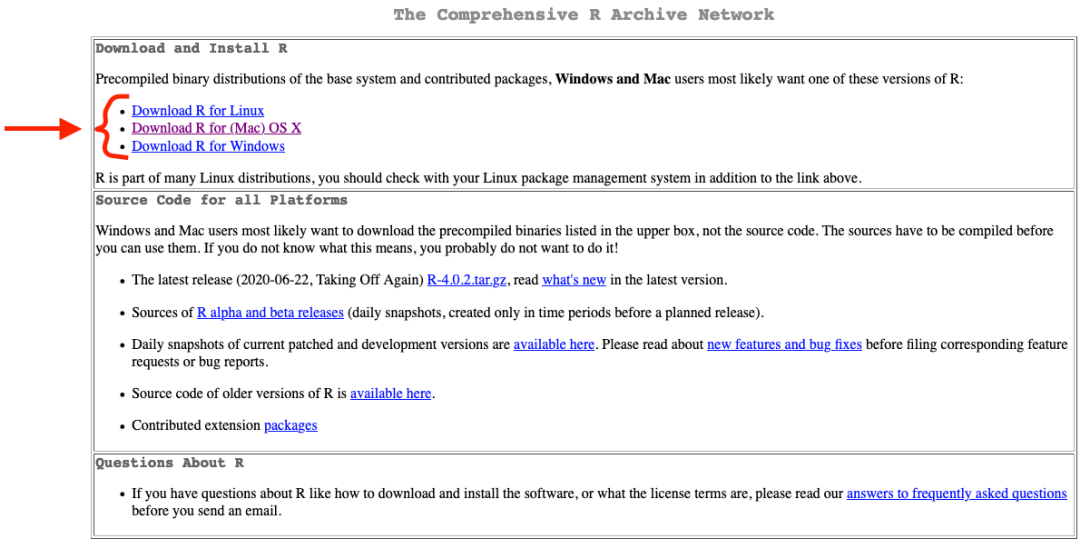
2. If it doesn't open automatically, double click on the file that downloads to your downloads folder, click okay and accept the defaults and the license agreement.
3. Download the latest R Studio for Mac under Step 2. The number will be different from the screenshot below, but the process is the same.
4. Click, hold and drag the RStudio icon into the Applications folder shortcut.



5. To get the RStudio icon to appear in your dock – you are going to use Rstudio for every single class we have this semester, so it would make sense – open a Finder window, go to your applications, open R Studio there, and then drag the icon to where you want it to appear in your dock. It will stay there after you have quit the program. To get rid of it after the semester is over, just drag the icon far enough out of the dock until you see a cloud icon appear.

On Windows:

1. Download R 4.2.2 from CRAN for Windows. The numbers in the screenshots below are 4.0.2, but the process is the same.



The Comprehensive R Archive Network

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2020-06-22, Taking Off Again) R-4.0.2.tar.gz, read what's new in the latest version.
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

R for Windows

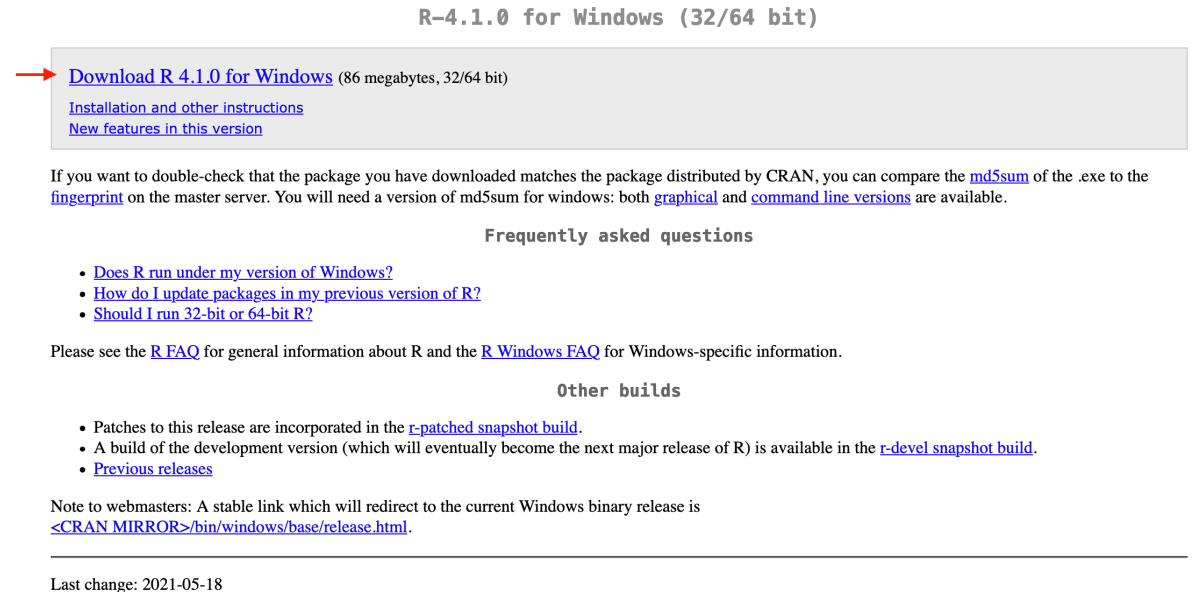
Subdirectories:

base	Binaries for base distribution. This is what you want to **install R for the first time**.
contrib	Binaries of contributed CRAN packages (for R >= 2.13.x; managed by Uwe Ligges). There is also information on third party software available for CRAN Windows services and corresponding environment and make variables.
old contrib	Binaries of contributed CRAN packages for outdated versions of R (for R < 2.13.x; managed by Uwe Ligges).
Rtools	Tools to build R and R packages. This is what you want to build your own packages on Windows, or to build R itself.

Please do not submit binaries to CRAN. Package developers might want to contact Uwe Ligges directly in case of questions / suggestions related to Windows binaries.

You may also want to read the R FAQ and R for Windows FAQ.

Note: CRAN does some checks on these binaries for viruses, but cannot give guarantees. Use the normal precautions with downloaded executables.



R-4.1.0 for Windows (32/64 bit)

Download R 4.1.0 for Windows (86 megabytes, 32/64 bit)

Installation and other instructions
New features in this version

If you want to double-check that the package you have downloaded matches the package distributed by CRAN, you can compare the md5sum of the .exe to the fingerprint on the master server. You will need a version of md5sum for windows: both graphical and command line versions are available.

Frequently asked questions

- Does R run under my version of Windows?
- How do I update packages in my previous version of R?
- Should I run 32-bit or 64-bit R?

Please see the R FAQ for general information about R and the R Windows FAQ for Windows-specific information.

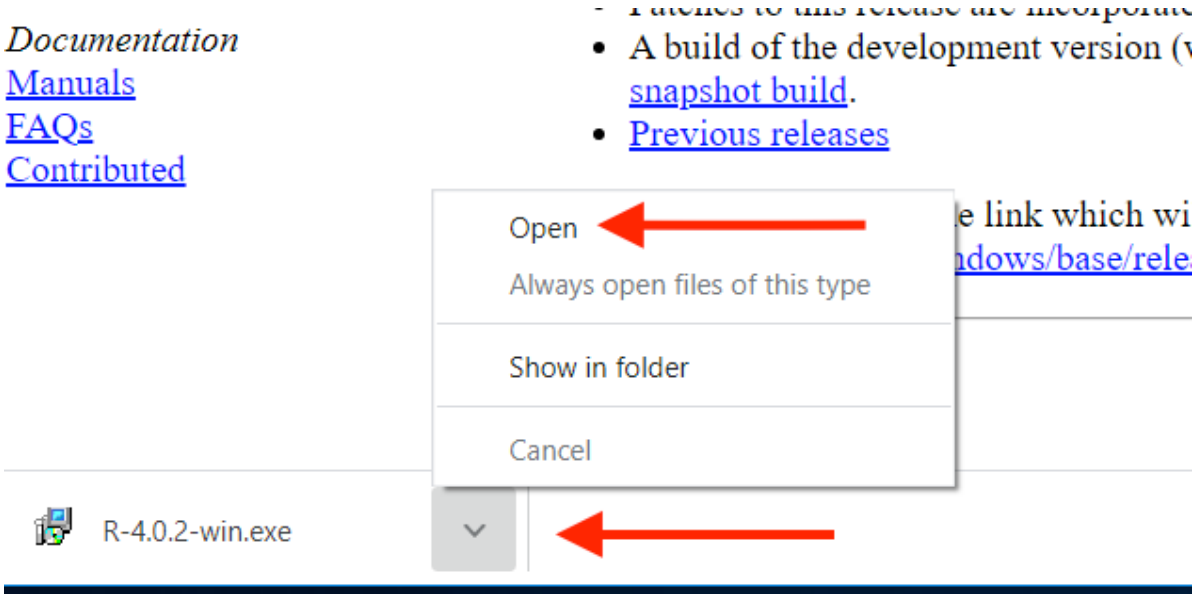
Other builds

- Patches to this release are incorporated in the r-patched snapshot build.
- A build of the development version (which will eventually become the next major release of R) is available in the r-devel snapshot build.
- Previous releases

Note to webmasters: A stable link which will redirect to the current Windows binary release is <CRAN MIRROR>/bin/windows/base/release.html.

Last change: 2021-05-18

2. Open and run the executable, accept the defaults and license agreement.

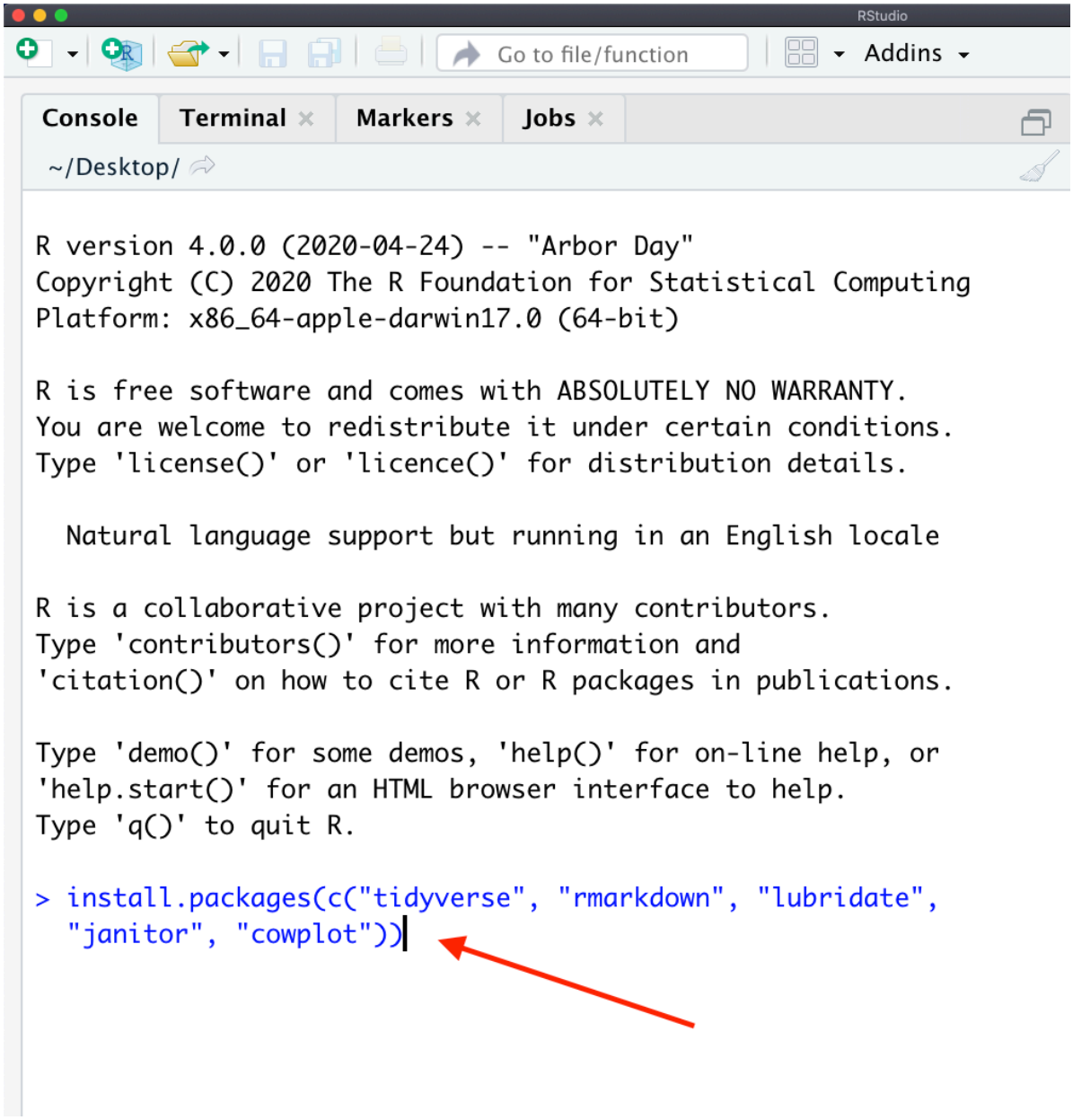


3. Go back to the screen where you downloaded the base R language, then download and install R Tools.
4. Download R Studio for Windows on step 2 of this page. Open the executable the same way you opened R. Hit next until it starts installing.
5. You can find it by typing RStudio into the Cortana search.

2.0.3 Part 3: Installing R libraries

1. Open R Studio. It should show the Console view by default. We'll talk a lot more about the console later.
2. Copy and paste this into the console and hit enter:

```
install.packages(c("tidyverse", "rmarkdown", "lubridate", "janitor", "cowplot",
"learnr", "remotes", "devtools", "hoopr", "nflfastR", "cfbfastR", "rvest",
"Hmisc", "cluster", "tidymodels", "bonsai"))
```

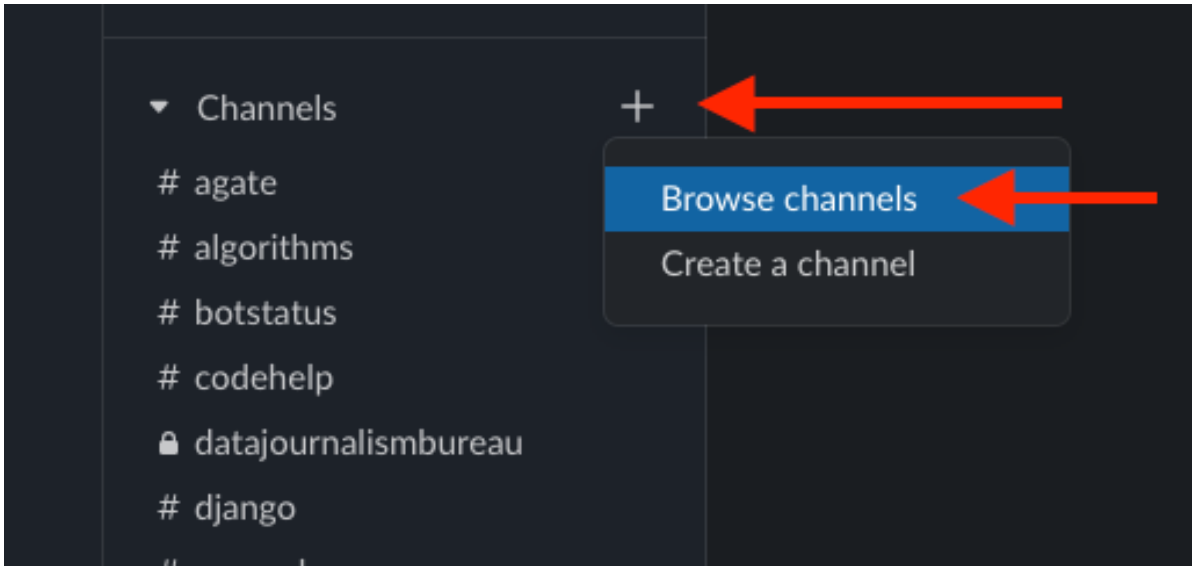


2.0.4 Part 4: Install Slack

1. Install Slack **on your computer and your phone** (you can find Slack in whatever app store you use). The reason I want it on both is because you are going to ask me for help with code via Slack. **Do not use screenshots unless specifically asked**. I want you to copy and paste your code. You can't do that on a phone. So you need the desktop

version. But I can usually solve your problem within a few minutes if you respond right away, and I know that you have your phone on you and are checking it. So the desktop version is for work, the phone version is for notifications.

2. Email me the address you want connected to Slack. Use one you'll actually check.
3. When you get the Slack invitation email, log in to the class slack via the apps, **not the website**.
4. Add the #r channel for general help I'll send to everyone in the channel and, if you want, the #jobstuff channel for news about jobs I come across.



3 Modeling and logistic regression

3.1 The basics

One of the most common – and seemingly least rigorous – parts of sports journalism is the prediction. There are no shortage of people making predictions about who will win a game or a league. Sure they have a method – looking at how a team is playing, looking at the players, consulting their gut – but rarely ever do you hear of a sports pundit using a model.

We're going to change that. Throughout this class, you'll learn how to use modeling to make predictions. Some of these methods will predict numeric values (like how many points will a team score based on certain inputs). Some will predict categorical values (W or L, Yes or No, All Star or Not).

There are lots of problems in the world where the answer is not a number but *a classification*: Did they win or lose? Did the player get drafted or no? Is this player a flight risk to transfer or not?

These are problems of classification and there are algorithms we can use to estimate the probability that X will be the outcome. How likely is it that this team with these stats will win this game?

Where this gets interesting is in the middle.

```
library(tidyverse)
library(tidymodels)
library(hoopR)
library(zoo)
library(gt)
set.seed(1234)
```

What we need to do here is get both sides of the game. We'll start with getting the box scores and cleaning them up a bit. We'll split the shooting columns into made and missed and turn everything into a number.

```
teamgames <- load_mbb_team_box(seasons = 2015:2023) %>%
  separate(field_goals_made_field_goals_attempted, into = c("field_goals_made","field_goal
  separate(three_point_field_goals_made_three_point_field_goals_attempted, into = c("three
```

```
  separate(free_throws_made_free_throws_attempted, into = c("free_throws_made","free_throw
  mutate_at(12:34, as.numeric)
```

3.2 Feature engineering

Feature engineering is the process of using what you know about something – domain knowledge – to find features in data that can be used in machine learning algorithms. Sports is a great place for this because not only do we know a lot because we follow the sport, but lots of other people are looking at this all the time. Creativity is good.

A number of basketball heads – including Ken Pomeroy of KenPom fame – have noticed that one of the predictors of the outcome of basketball games are possession metrics. How efficient are teams with the possessions they have? Can't score if you don't have the ball, so how good is a team at pushing the play and getting more possessions, giving themselves more chances to score?

One problem? Possessions aren't in typical metrics. They aren't usually tracked. But you can estimate them from typical box scores. The way to do that is like this:

`Possessions = Field Goal Attempts - Offensive Rebounds + Turnovers + (0.475 * Free Throw Attempts)`

If you look at the data we already have, however, you'll see possessions are not actually in the data. Which is unfortunate. But we can calculate it pretty easily.

Then we'll use the possessions estimate formula to get that, so we can then calculate points per possession.

We'll save that to a new dataframe called `teamstats`.

3.2.1 Exercise 1: setting up your data

```
teamstats <- teamgames %>%
  mutate(
    team_score = ((field_goals_made-three_point_field_goals_made) * 2) + (three_point_fiel
    possessions = ?????_?????_attempted - offensive_rebounds + ????????? + (.475 * free_th
    ppp = team_score/possessions
  )
```

Now we begin the process of creating a model. Modeling in data science has a **ton** of details, but the process for each model type is similar.

1. Split your data into training and testing data sets. A common split is 80/20.

2. Train the model on the training dataset.
3. Evaluate the model on the training data.
4. Apply the model to the testing data.
5. Evaluate the model on the test data.

From there, it's how you want to use the model. We'll walk through a simple example here, using a simple model – a logistic regression model.

What we're trying to do here is predict which team will win given their efficiency with the ball, expressed as points per possession. However, to make a prediction, we need to know their stats *BEFORE* the game – what we knew about the team going into the game in question. We can do that using `zoo` and rolling means.

A rolling mean is an average in a window of time. So if we averaged together the points per possession over 10 games, that's a 10 game rolling mean. The first real mean would be games 1-10. Then the window would shift one game with game 11, and the average would be games 2-11. Then 3-12, 4-13 and so on.

We'll add three new columns – the one game lagged rolling mean of shooting percentage, points per possession and true shooting percentage.

The problem we have to face here is that with our data, a rolling mean of games 6-15 would mean if we were trying to predict game 15, we couldn't *include* game 15. We'd have to look at games 5-14. If we included game 15, it would mean we had God like abilities to predict the future.

We do not. Introducing the `lag` function. The lag function just takes the window of data and shifts it however many spots back you want to shift it. In our case, we want to shift it one game back. And we're going to make a rolling window of 5 games.

3.2.2 Exercise 2: Lagging

```
rollingteamstats <- teamstats %>%
  arrange(game_date) %>%
  group_by(team_short_display_name, season) %>%
  mutate(
    team_score = ((field_goals_made-three_point_field_goals_made) * 2) + (three_point_fiel
    team_rolling_ppp = rollmean(???(ppp, n=?), k=5, align="right", fill=NA)
    ) %>%
  ungroup()
```

Now we need to do something that at first will seem kind of odd, but isn't when you think about it. Our data has half of the box score – just one team. But a game has two teams in it. To get it, we need the team AND the opponent. How can you decide if a team is going

to win if you don't know who they are playing and if that team is any good or not? So we need to create two dataframes that have column names that indicate these stats are the team stats and these stats are the opponent stats. We can do that with some selecting and some renaming.

```
team_side <- rollingteamstats %>%
  select(
    game_id,
    team_id,
    team_short_display_name,
    opponent_id,
    game_date,
    season,
    team_score,
    team_rolling_ppp
    ) %>%
  na.omit()

opponent_side <- team_side %>%
  select(-opponent_id) %>%
  rename(
    opponent_id = team_id,
    opponent_short_display_name = team_short_display_name,
    opponent_score = team_score,
    opponent_rolling_ppp = team_rolling_ppp
  ) %>%
  mutate(opponent_id = as.numeric(opponent_id)
)
```

Now we'll join them together.

3.2.3 Exercise 3: Joining the data together

```
games <- ????_side %>% inner_join(?????????_side)
```

```
Joining, by = c("game_id", "opponent_id", "game_date", "season")
```

The last problem to solve? Who won? We can add this with conditional logic. The other thing we're doing here is we're going to is we're going to convert our new team_result column into a factor. What is a factor? A factor is a type of data in R that stores categorical values

that have a limited number of differences. So wins and losses are a perfect factor. Modeling libraries are looking for factors so it can treat the differences in the data as categories, so that's why we're converting it here.

```
games <- games %>% mutate(
  team_result = as.factor(case_when(
    team_score > opponent_score ~ "W",
    opponent_score > team_score ~ "L"
))) %>% na.omit()
```

Now that we've done that, we need to look at the order of our factors.

3.2.4 Exercise 4: Looking at the factors

To do that, we first need to know what R sees when it sees our `team_result` factor. Is a win first or is a loss first?

```
levels(games$????_??????)
```

```
[1] "L" "W"
```

The order listed here is the order they are in. What this means is that our predictions will be done through the lens of losses. That doesn't make intuitive sense to us. We want to know who will win! We can reorder the factors with `relevel`.

3.2.5 Exercise 5: Releveling the factors

```
games$team_result <- relevel(games$team_result, ref="?")

levels(games$team_result)
```

```
[1] "W" "L"
```

For simplicity, let's limit the number of columns we're going to feed our model.

```
modelgames <- games %>%
  select(
    game_id,
```

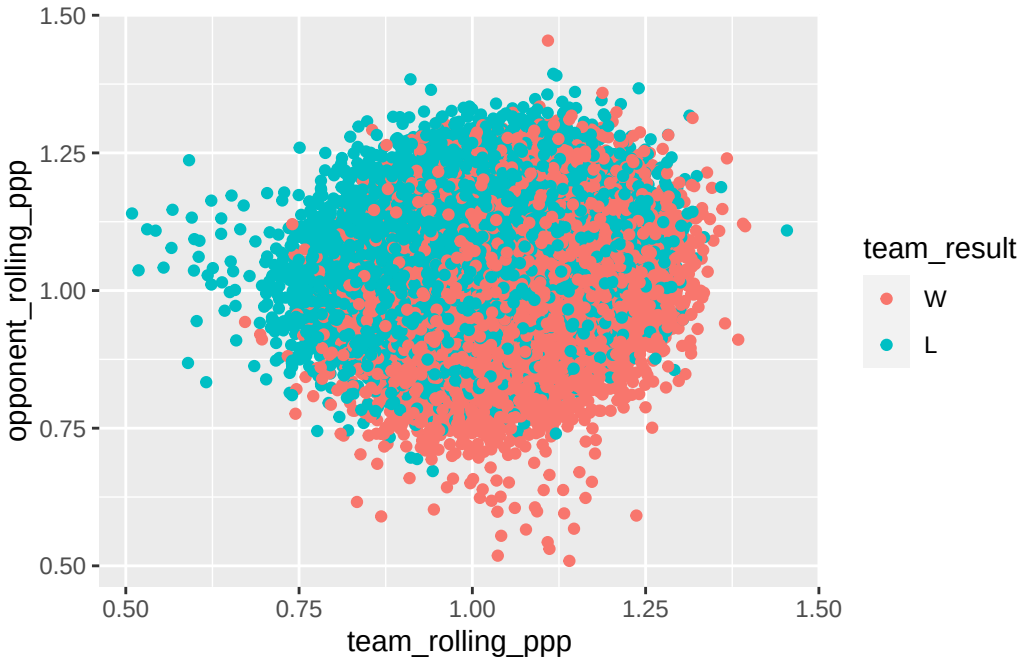
```
    game_date,
    team_short_display_name,
    opponent_short_display_name,
    season,
    team_rolling_ppp,
    opponent_rolling_ppp,
    team_result
    ) %>% na.omit()
```

3.3 Visualizing the decision boundary

This is just one dimension of the data, but it can illustrate how this works. You can almost see a line running through the middle, with a lot of overlap. The further left or right you go, the less overlap. You can read it like this: If this team shoots this well and the opponent shoots this well, most of the time this team wins. Or loses. It just depends on where the dot ends up.

That neatly captures the probabilities we're looking at here.

```
ggplot() +
  geom_point(
    data=games, aes(x=team_rolling_ppp, y=opponent_rolling_ppp, color=team_result))
```



3.4 The logistic regression

To create a model, we have to go through a process. That process starts with splitting data where we know the outomes into two groups – training and testing. The training data is what we will use to create our model. The testing data is how we will determine how good it is. Then, going forward, our model can predict games we *haven't* seen yet.

To do this, we're going to first split our `modelgames` data into two groups – with 80 percent of it in one, 20 percent in the other. We do that by feeding our simplified dataframe into the `initial_split` function. Then we'll explicitly name those into new dataframes called train and test.

3.4.1 Exercise 6: What are we splitting?

```
log_split <- initial_split(??????????, prop = .8)
log_train <- training(log_split)
log_test <- testing(log_split)
```

Now we have two dataframes – log_train and log_test – that we can now use for modeling.

First step to making a model is to set what type of model this will be. We're going to name our model object – log_mod works because this is a logistic regression model. We'll use the logistic_reg function in parsnip (the modeling library in Tidymodels) and set the engine to "glm". The mode in our case is "classification" because we're trying to classify something as a W or L. Later, we'll use "regression" to predict numbers.

```
log_mod <-
  logistic_reg() %>%
  set_engine("glm") %>%
  set_mode("classification")
```

The next step is to create a recipe. This is a series of steps we'll use to put our data into our model. For example – what is predicting what? And what aren't predictors and what are? And do we have to do any pre-processing of the data?

The first part of the recipe is the formula. In this case, we're saying – in real words – `team_result` *is approximately modeled by* our predictors, which we represent as `.` which means all the stuff. Then, importantly, we say what *isn't* a predictor next with update_role. So the team name, the game date and things like that are *not* predictors. So we need to tell it that. The last step is normalizing our numbers. With logistic regression, scale differences in numbers can skew things, so we're going to turn everything into Z-scores.

```
log_recipe <-
  recipe(team_result ~ ., data = log_train) %>%
  update_role(game_id, game_date, team_short_display_name, opponent_short_display_name, se
  step_normalize(all_predictors())

summary(log_recipe)
```

```
# A tibble: 8 x 4
  variable                    type    role      source
  <chr>                       <chr>   <chr>     <chr>
1 game_id                     numeric ID        original
2 game_date                   date    ID        original
3 team_short_display_name     nominal ID        original
4 opponent_short_display_name nominal ID        original
5 season                      numeric ID        original
6 team_rolling_ppp            numeric predictor original
7 opponent_rolling_ppp        numeric predictor original
8 team_result                 nominal outcome   original
```

Now we have enough for a workflow. A workflow is what we use to put it all together. In it, we add our model definition and our recipe.

3.4.2 Exercise 7: Making a workflow

```
log_workflow <-
  workflow() %>%
  add_model(log_???) %>%
  add_recipe(log_??????)
```

And now we fit our model (this can take a few minutes).

```
log_fit <-
  log_workflow %>%
  fit(data = log_train)
```

3.5 Evaluating the fit

With logistic regression, there's two things we're looking at: The prediction and the probabilities. We can get those with two different fits and combine them together.

First, you can see the predictions like this:

```
trainpredict <- log_fit %>% predict(new_data = log_train) %>%
  bind_cols(log_train)

trainpredict
```

```
# A tibble: 61,020 x 9
   .pred_class   game_id game_date  team_short_display_~ opponent_short_~ season
   <fct>           <int> <date>     <chr>                <chr>              <int>
 1 W           401083976 2019-02-03 UNC Wilmington       James Madison       2019
 2 L           400839389 2016-02-11 Nebraska             Wisconsin           2016
 3 W           400988107 2018-02-10 Wichita State        UConn               2018
 4 L           401373737 2022-02-25 UC Davis             UCSB                2022
 5 W           401309755 2021-03-05 Cincinnati           Vanderbilt          2021
 6 L           401377777 2022-02-27 Temple               Tulane              2022
 7 L           400868397 2016-03-04 Manhattan            Marist              2016
 8 L           400988599 2018-02-04 Seton Hall           Villanova           2018
 9 W           401172379 2020-01-18 Towson               James Madison       2020
10 W           400847285 2016-02-27 South Dakota St      Oral Roberts        2016
# ... with 61,010 more rows, and 3 more variables: team_rolling_ppp <dbl>,
#   opponent_rolling_ppp <dbl>, team_result <fct>
```

Then, we can just add it to `trainpredict` using `bind_cols`, which means we're going to bind the columns of this new fit to the old trainpredict.

```
trainpredict <- log_fit %>% predict(new_data = log_train, type="prob") %>%
  bind_cols(trainpredict)

trainpredict
```

```
# A tibble: 61,020 x 11
   .pred_W .pred_L .pred_class   game_id game_date  team_short_display_name
     <dbl>   <dbl> <fct>           <int> <date>     <chr>
 1   0.688   0.312 W           401083976 2019-02-03 UNC Wilmington
 2   0.430   0.570 L           400839389 2016-02-11 Nebraska
 3   0.618   0.382 W           400988107 2018-02-10 Wichita State
 4   0.346   0.654 L           401373737 2022-02-25 UC Davis
 5   0.506   0.494 W           401309755 2021-03-05 Cincinnati
 6   0.289   0.711 L           401377777 2022-02-27 Temple
 7   0.262   0.738 L           400868397 2016-03-04 Manhattan
 8   0.176   0.824 L           400988599 2018-02-04 Seton Hall
 9   0.715   0.285 W           401172379 2020-01-18 Towson
10   0.622   0.378 W           400847285 2016-02-27 South Dakota St
# ... with 61,010 more rows, and 5 more variables:
#   opponent_short_display_name <chr>, season <int>, team_rolling_ppp <dbl>,
#   opponent_rolling_ppp <dbl>, team_result <fct>
```

There's several metrics to look at to evaluate the model on our training data, but the two we will use are accuracy and roc_auc. They both are pointing toward how well the model did in two different ways. The accuracy metric looks at the number of predictions that are correct when compared to known results. The inputs here are the data, the column that has the actual result, and the column with the prediction, called `.pred_class`.

3.5.1 Exercise 8: Metrics

```
metrics(trainpredict, ????_??????, .pred_class)
```

```
# A tibble: 2 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.617
2 kap      binary         0.233
```

So how accurate is our model? If we're looking for perfection, we're far from it. But if we're looking to make straight up win loss bets … we're doing okay!

Another way to look at the results is the confusion matrix. The confusion matrix shows what was predicted compared to what actually happened. The squares are True Positives, False Positives, True Negatives and False Negatives. True values vs the total values make up the accuracy.

3.5.2 Exercise 9: Confusion matrix

```
trainpredict %>%
  conf_mat(????_result, .pred_?????)
```

```
          Truth
Prediction     W     L
         W 18923 11744
         L 11647 18706
```

3.6 Comparing it to test data

Now we can apply our fit to the test data to see how robust it is. If the metrics are similar, that's good – it means our model is robust. If the metrics change a lot, that's bad. It means our model is guessing.

```
testpredict <- log_fit %>% predict(new_data = log_test) %>%
  bind_cols(log_test)

testpredict <- log_fit %>% predict(new_data = log_test, type="prob") %>%
  bind_cols(testpredict)
```

And now some metrics on the test data.

3.6.1 Exercise 10: Testing

```
metrics(????predict, team_result, .pred_class)
```

```
# A tibble: 2 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.614
2 kap      binary         0.229
```

How does that compare to our training data? Is it lower? Higher? Are the changes large – like are we talking about single digit changes or double digit changes? The less it changes, the better.

And now the confusion matrix.

```
testpredict %>%
  conf_mat(team_result, .pred_class)
```

```
          Truth
Prediction    W    L
         W 4639 2954
         L 2929 4734
```

How does that compare to the training data?

3.7 How well did it do with Nebraska?

Let's grab predictions for Nebraska from both our test and train data and take a look.

```
nutrain <- trainpredict %>% filter(team_short_display_name == "Nebraska" &  season == 2023

nutest <- testpredict %>% filter(team_short_display_name == "Nebraska" & season == 2023)

bind_rows(nutrain, nutest) %>%
  arrange(game_date) %>%
  select(.pred_W, .pred_class, team_result, team_short_display_name, opponent_short_displa
  gt()
```

.pred_W	.pred_class	team_result	team_short_display_name	opponent_short_display_name
0.5497281	W	W	Nebraska	Florida St
0.5973002	W	W	Nebraska	Boston College
0.5175988	W	W	Nebraska	Creighton

0.4341151	L	L	Nebraska	Indiana
0.2667989	L	L	Nebraska	Purdue
0.4810879	L	L	Nebraska	Kansas St
0.3176467	L	W	Nebraska	Queens
0.3050453	L	W	Nebraska	Iowa
0.3458684	L	L	Nebraska	Michigan St
0.4699218	L	W	Nebraska	Minnesota
0.4310008	L	L	Nebraska	Illinois
0.2072669	L	L	Nebraska	Purdue
0.3605986	L	W	Nebraska	Ohio State
0.2536178	L	L	Nebraska	Penn State
0.3102129	L	L	Nebraska	Northwestern
0.2234788	L	L	Nebraska	Maryland
0.3468551	L	L	Nebraska	Illinois

By our rolling metrics, are there any surprises? Should we have beaten Creighton or Iowa?

How could you improve this?

4 Decision trees and random forests

4.1 The basics

Tree-based algorithms are based on decision trees, which are very easy to understand. A decision tree can basically be described as a series of questions. Does this player have more or less than x seasons of experience? Do they have more or less then y minutes played? Do they play this or that position? Answer enough questions, and you can predict what that player should have on average.

The upside of decision trees is that if the model is small, you can explain it to anyone. They're very easy to understand. The trouble with decision trees is that if the model is small, they're a bit of a crude instrument. As such, multiple tree based methods have been developed as improvements on the humble decision tree.

The most common is the random forest.

Let's implement one. We start with libraries.

```
library(tidyverse)
library(tidymodels)
library(hoopR)
library(zoo)

set.seed(1234)
```

Let's use what we had from the last tutorial – a rolling window of points per possession for team and opponent. I've gone ahead and run it all in the background. You can see modelgames by using `head` in the block.

```
teamgames <- load_mbb_team_box(seasons = 2015:2023) %>%
  separate(field_goals_made_field_goals_attempted, into = c("field_goals_made","field_goal
  separate(three_point_field_goals_made_three_point_field_goals_attempted, into = c("three
  separate(free_throws_made_free_throws_attempted, into = c("free_throws_made","free_throw
  mutate_at(12:34, as.numeric)

teamstats <- teamgames %>%
```

```
  mutate(
    team_score = ((field_goals_made-three_point_field_goals_made) * 2) + (three_point_fiel
    possessions = field_goals_attempted - offensive_rebounds + turnovers + (.475 * free_th
    ppp = team_score/possessions
  )

rollingteamstats <- teamstats %>%
  group_by(team_short_display_name, season) %>%
  arrange(game_date) %>%
  mutate(
    team_score = ((field_goals_made-three_point_field_goals_made) * 2) + (three_point_fiel
    team_rolling_ppp = rollmean(lag(ppp, n=1), k=5, align="right", fill=NA)
    ) %>%
  ungroup()

team_side <- rollingteamstats %>%
  select(
    game_id,
    team_id,
    team_short_display_name,
    opponent_id,
    game_date,
    season,
    team_score,
    team_rolling_ppp
    )

opponent_side <- team_side %>%
  select(-opponent_id) %>%
  rename(
    opponent_id = team_id,
    opponent_short_display_name = team_short_display_name,
    opponent_score = team_score,
    opponent_rolling_ppp = team_rolling_ppp
  ) %>%
  mutate(opponent_id = as.numeric(opponent_id)
)

games <- team_side %>% inner_join(opponent_side)
```

```
Joining, by = c("game_id", "opponent_id", "game_date", "season")
```

```
games <- games %>% mutate(
  team_result = as.factor(case_when(
    team_score > opponent_score ~ "W",
    opponent_score > team_score ~ "L"
)))

modelgames <- games %>%
  select(
    game_id,
    game_date,
    team_short_display_name,
    opponent_short_display_name,
    season,
    team_rolling_ppp,
    opponent_rolling_ppp,
    team_result
    ) %>%
  na.omit()
```

For this tutorial, we're going to create two models from two workflows so that we can compare a logistic regression to a random forest.

4.2 Setup

A random forest is, as the name implies, a large number of decision trees, and they use a random set of inputs. The algorithm creates a large number of randomly selected training inputs, and randomly chooses the feature input for each branch, creating predictions. The goal is to create uncorrelated forests of trees. The trees all make predictions, and the wisdom of the crowds takes over. In the case of classification algorithm, the most common prediction is the one that gets chosen. In a regression model, the predictions get averaged together.

The random part of random forest is in how the number of tree splits get created and how the samples from the data are taken to generate the splits. They're randomized, which has the effect of limiting the influence of a particular feature and prevents overfitting – where your predictions are so tailored to your training data that they miss badly on the test data.

For random forests, we change the model type to rand_forest and set the engine to "ranger". There's multiple implementations of the random forest algorithm, and the differences between them are beyond the scope of what we're doing here.

We're going to go through the steps of modeling again, starting with splitting our `modelgames` data.

4.2.1 Exercise 1: setting up your data

```
game_split <- initial_split(??????????, prop = .8)
game_train <- training(game_split)
game_test <- testing(game_split)
```

For this walkthrough, we're going to do both a logistic regression and a random forest side by side to show the value of workflows.

The recipe we'll create is the same for both, so we'll use it twice.

4.2.2 Exercise 2: setting up the receipe

So what data are we feeding into our recipe?

```
game_recipe <-
  recipe(team_result ~ ., data = game_?????) %>%
  update_role(game_id, game_date, team_short_display_name, opponent_short_display_name, se
  step_normalize(all_predictors())

summary(game_recipe)
```

```
# A tibble: 8 x 4
  variable                    type    role      source
  <chr>                       <chr>   <chr>     <chr>
1 game_id                     numeric ID        original
2 game_date                   date    ID        original
3 team_short_display_name     nominal ID        original
4 opponent_short_display_name nominal ID        original
5 season                      numeric ID        original
6 team_rolling_ppp            numeric predictor original
7 opponent_rolling_ppp        numeric predictor original
8 team_result                 nominal outcome   original
```

Now, we're going to create two different model specifications. The first will be the logistic regression model definintion and the second will be the random forest.

```
log_mod <-
  logistic_reg() %>%
  set_engine("glm") %>%
  set_mode("classification")
```

```
rf_mod <-
  rand_forest() %>%
  set_engine("ranger") %>%
  set_mode("classification")
```

Now we have enough for our workflows. We have two models and one recipe.

4.2.3 Exercise 3: making workflows

```
log_workflow <-
  workflow() %>%
  add_model(???_mod) %>%
  add_recipe(????_recipe)

rf_workflow <-
  workflow() %>%
  add_model(??_mod) %>%
  add_recipe(????_recipe)
```

Now we can fit our models to the data.

4.2.4 Exercise 4: fitting our models

```
log_fit <-
  log_workflow %>%
  fit(data = ????_?????)

rf_fit <-
  rf_workflow %>%
  fit(data = ????_?????)
```

Now we can bind our predictions to the training data and see how we did.

```
logpredict <- log_fit %>% predict(new_data = game_train) %>%
  bind_cols(game_train)

logpredict <- log_fit %>% predict(new_data = game_train, type="prob") %>%
  bind_cols(logpredict)
```

```
rfpredict <- rf_fit %>% predict(new_data = game_train) %>%
  bind_cols(game_train)

rfpredict <- rf_fit %>% predict(new_data = game_train, type="prob") %>%
  bind_cols(rfpredict)
```

Now, how did we do? First, let's look at the logistic regression.

4.2.5 Exercise 5: The first metrics

What prediction dataset do we feed into our metrics?

```
metrics(??????????, team_result, .pred_class)
```

```
# A tibble: 2 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.617
2 kap      binary         0.233
```

Same as last time, the logistic regression model comes in at 62 percent accuracy, and when we expose it to testing data, it remains pretty stable. *This is a gigantic hint about what is to come.*

How about the random forest?

4.2.6 Exercise 6: Random forest metrics

```
metrics(??predict, team_result, .pred_class)
```

```
# A tibble: 2 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.960
2 kap      binary         0.920
```

Holy buckets! We made a model that's 96 percent accurate? GET ME TO VEGAS.

Remember: Where a model makes its money is in data that it has never seen before.

First, we look at logistic regression.

```
logtestpredict <- log_fit %>% predict(new_data = game_test) %>%
  bind_cols(game_test)

logtestpredict <- log_fit %>% predict(new_data = game_test, type="prob") %>%
  bind_cols(logtestpredict)

metrics(logtestpredict, team_result, .pred_class)
```

```
# A tibble: 2 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.614
2 kap      binary         0.229
```

Just about the same. That's a robust model.

Now, the inevitable crash with random forests.

```
rftestpredict <- rf_fit %>% predict(new_data = game_test) %>%
  bind_cols(game_test)

rftestpredict <- rf_fit %>% predict(new_data = game_test, type="prob") %>%
  bind_cols(rftestpredict)

metrics(rftestpredict, team_result, .pred_class)
```

```
# A tibble: 2 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.578
2 kap      binary         0.157
```

Right at 57 percent. A little bit lower than logistic regression. But did they come to the same answers to get those numbers? No.

```
logtestpredict %>%
  conf_mat(team_result, .pred_class)
```

```
          Truth
Prediction    L    W
         L 4734 2929
         W 2954 4639
```

```
rftestpredict %>%
  conf_mat(team_result, .pred_class)
```

```
          Truth
Prediction    L    W
         L 4467 3212
         W 3221 4356
```

Our two models, based on our very basic feature engineering, are only slightly better than flipping a coin. If we want to get better, we've got work to do.

5 XGBoost

5.1 The basics

As we learned in the previous chapter, random forests (and bagged methods) average together a large number of trees to get to an answer. Random forests add a wrinkle by randomly choosing features at each branch to make it so each tree is not correlated and the trees are rather deep. The idea behind averaging them together is to cut down on the variance in predictions – random forests tend to be somewhat harder to fit to unseen data because of the variance. Random forests are fairly simple to implement, and are very popular.

Boosting methods are another wrinkle in the tree based methods. Instead of deep trees, boosting methods intentionally pick shallow trees – called stumps – that, at least initially, do a poor job of predicting the outcome. Then, each subsequent stump takes the job the previous one did, optimizes to reduce the residuals – the gap between prediction and reality – and makes a prediction. And then the next one does the same, and so on and so on.

The path to a boosted method is complex, the results can take a lot of your computer's time, but the models are more generalizable, meaning they handle new data better than other methods. Among data scientists, boosted methods, such as xgboost and lightgbm, are very popular for solving a wide variety of problems.

Let's re-implement our predictions in an XGBoost algorithm. First, we'll load libraries.

```
library(tidyverse)
library(tidymodels)
library(zoo)
library(hoopR)

set.seed(1234)
```

We'll load our game data and do a spot of feature engineering that we used with our other models.

```
teamgames <- load_mbb_team_box(seasons = 2015:2023) %>%
  separate(field_goals_made_field_goals_attempted, into = c("field_goals_made","field_goal
  separate(three_point_field_goals_made_three_point_field_goals_attempted, into = c("three
```

```
  separate(free_throws_made_free_throws_attempted, into = c("free_throws_made","free_throw
  mutate_at(12:34, as.numeric)

teamstats <- teamgames %>%
  mutate(
    team_score = ((field_goals_made-three_point_field_goals_made) * 2) + (three_point_fiel
    possessions = field_goals_attempted - offensive_rebounds + turnovers + (.475 * free_th
    ppp = team_score/possessions
  )

rollingteamstats <- teamstats %>%
  group_by(team_short_display_name, season) %>%
  arrange(game_date) %>%
  mutate(
    team_score = ((field_goals_made-three_point_field_goals_made) * 2) + (three_point_fiel
    team_rolling_ppp = rollmean(lag(ppp, n=1), k=5, align="right", fill=NA)
    ) %>%
  ungroup()

team_side <- rollingteamstats %>%
  select(
    game_id,
    team_id,
    team_short_display_name,
    opponent_id,
    game_date,
    season,
    team_score,
    team_rolling_ppp
    )

opponent_side <- team_side %>%
  select(-opponent_id) %>%
  rename(
    opponent_id = team_id,
    opponent_short_display_name = team_short_display_name,
    opponent_score = team_score,
    opponent_rolling_ppp = team_rolling_ppp
  ) %>%
  mutate(opponent_id = as.numeric(opponent_id)
)
```

```
games <- team_side %>% inner_join(opponent_side)

games <- games %>% mutate(
  team_result = as.factor(case_when(
    team_score > opponent_score ~ "W",
    opponent_score > team_score ~ "L"
))) %>% na.omit()

modelgames <- games %>%
  select(
    game_id,
    game_date,
    team_short_display_name,
    opponent_short_display_name,
    season,
    team_rolling_ppp,
    opponent_rolling_ppp,
    team_result
    ) %>%
  na.omit()
```

Per usual, we split our data into training and testing.

```
game_split <- initial_split(modelgames, prop = .8)
game_train <- training(game_split)
game_test <- testing(game_split)
```

And our recipe.

```
game_recipe <-
  recipe(team_result ~ ., data = game_train) %>%
  update_role(game_id, game_date, team_short_display_name, opponent_short_display_name, se

summary(game_recipe)
```

```
# A tibble: 8 x 4
  variable                  type    role      source
  <chr>                     <chr>   <chr>     <chr>
1 game_id                   numeric ID        original
2 game_date                 date    ID        original
3 team_short_display_name   nominal ID        original
```

```
4 opponent_short_display_name nominal ID        original
5 season                      numeric ID        original
6 team_rolling_ppp            numeric predictor original
7 opponent_rolling_ppp        numeric predictor original
8 team_result                 nominal outcome   original
```

To this point, everything looks like what we've done before. Nothing has really changed. It's about to.

5.2 Hyperparameters

The hyperparameters are the inputs into the algorithm that make the fit. To find the ideal hyperparameters, you need to tune them. But first, let's talk about the hyperparameters we are going to tune (there are others we can, but every addition to the number of hyperparameters means more computation):

- Number of trees – this is the total number of trees in the sequence. A gradient boosting algorithm will minimize residuals forever, so you need to tell it where to stop. That stopping point is different for every problem. You can tune this, but I'll warn you – this is the most computationally expensive tuning. For our example, we're going to set the number of trees at 30. The default is 15. Tuning it can take a good computer more than an hour to complete, and you'll have gained .1 percent of accuracy. If we're Amazon and billions of dollars are on the line, it's worth it. For predicting NCAA tournament games, it is not.
- Learn rate – this controls how fast the algorithm goes down the gradient descent – how fast it learns. Too fast and you'll overshoot the optimal stopping point and start going up the error curve. Too slow and you'll never get to the optimal stopping point.
- Tree depth – controls the depth of each individual tree. Too short and you'll need a lot of them to get good results. Too deep and you risk overfitting.
- Minimum number of observations in the terminal node (min_n) – controls the complexity of each tree. Typical values range from 5-15, and higher values keep a model from figuring out relationships that are unique to that training set (ie overfitting).
- Loss reduction – this is the minimum loss reduction to make a new tree split. If the improvement hits this minimum, a split occurs. A low value and you get a complex tree. High value and you get a tree more robust to new data, but it's more conservative.

Others you can tune, but have sensible defaults:

- Sample size – The fraction of the total training set that can be used for each boosting round. Low values may lead to underfitting, high to overfitting.
- mtry – the number of predictors that will be randomly sampled at each split when making trees.

All of these combine to make the model, and each has their own specific ideal. How do we find it? Tuning.

First, we make a model and label each parameter as tune()

```
xg_mod <- boost_tree(
  trees = 30,
  mtry = tune(),
  learn_rate = ????(),
  tree_depth = ????(),
  min_n = ????(),
  loss_reduction = ????()
  ) %>%
  set_mode("classification") %>%
  set_engine("xgboost")
```

5.2.1 Exercise 1: making a workflow

Let's make a workflow now that we have our recipe and our model.

```
game_wflow <-
  workflow() %>%
  add_model(??_???) %>%
  add_recipe(????_recipe)
```

Now, to tune the model, we have to create a grid. The grid is essentially a random sample of parameters to try. The latin hypercube is a method of creating a near-random sample of parameter values in multidimentional distributions (ie there's more than one predictor). The latin hypercube is near-random because there has to be one sample in each row and column of the hypercube. Essentially, it removes the possibility of totally empty spaces in the cube. Why is that important? Because this hypercube is how your tuning is going to find the optimal outputs.

What follows is what parameters the hypercube will tune.

```
xgb_grid <- grid_latin_hypercube(
  finalize(mtry(), game_train),
  tree_depth(),
  min_n(),
  loss_reduction(),
  learn_rate(),
  size = 30
```

```
  )

  xgb_grid
```

```
# A tibble: 30 x 5
    mtry tree_depth min_n loss_reduction    learn_rate
   <int>      <int> <int>          <dbl>         <dbl>
 1     8         13     3       1.45e- 7 0.0000379
 2     6         11    27       3.91e- 1 0.00000483
 3     5          9    25       3.91e- 9 0.00000000226
 4     5          2     5       3.31e- 7 0.000000379
 5     5          3    26       1.67e- 8 0.0100
 6     2         10    22       4.58e-10 0.00151
 7     5          1    19       1.78e- 4 0.000425
 8     6         13    29       2.83e- 5 0.00000944
 9     6          8    13       5.71e- 3 0.0302
10     8         14    18       2.94e- 1 0.0000152
# ... with 20 more rows
```

How do we tune it? Using something called cross fold validation. Cross fold validation takes our grid, applies it to a set of subsets (in our case 10 subsets) and compares. It'll take a random square in the hypercube, try the combinations in there, and see what happens. It'll then keep doing that, over and over and over and over. When it's done, each validation set will have a set of tuned values and outcomes that we can evaluate and pick the optimal set to get a result.

5.2.2 Exercise 2: creating our cross-fold validation set

This will create the folds, which are just 10 random subsets of the training data.

```
  game_folds <- vfold_cv(game_?????)

  game_folds
```

```
#  10-fold cross-validation
# A tibble: 10 x 2
   splits               id
   <list>               <chr>
 1 <split [55193/6133]> Fold01
 2 <split [55193/6133]> Fold02
```

```
 3 <split [55193/6133]> Fold03
 4 <split [55193/6133]> Fold04
 5 <split [55193/6133]> Fold05
 6 <split [55193/6133]> Fold06
 7 <split [55194/6132]> Fold07
 8 <split [55194/6132]> Fold08
 9 <split [55194/6132]> Fold09
10 <split [55194/6132]> Fold10
```

Now we come to the part that is going to take some time on your computer. How long? It depends. On my old 2018 Intel Mac, this part took about 25-30 minutes on my machine and made it sounds like it was attempting liftoff. My 2022 M1 Mac? Right around 10 minutes. Depending on how new and how powerful you computer is, it could take minutes, or it could take hours. The point being, start it up, walk away, and let it burn.

```
xgb_res <- tune_grid(
  game_wflow,
  resamples = game_folds,
  grid = xgb_grid,
  control = control_grid(save_pred = TRUE)
)
```

Our grid has run on all of our validation samples, and what do we see?

```
collect_metrics(xgb_res)
```

```
# A tibble: 60 x 11
    mtry min_n tree_depth    learn_rate loss_reduction .metric  .estimator  mean
   <int> <int>      <int>         <dbl>          <dbl> <chr>    <chr>      <dbl>
 1     8     3         13 0.0000379      0.000000145   accuracy binary     0.588
 2     8     3         13 0.0000379      0.000000145   roc_auc  binary     0.623
 3     6    27         11 0.00000483     0.391         accuracy binary     0.606
 4     6    27         11 0.00000483     0.391         roc_auc  binary     0.647
 5     5    25          9 0.00000000226  0.00000000391 accuracy binary     0.500
 6     5    25          9 0.00000000226  0.00000000391 roc_auc  binary     0.510
 7     5     5          2 0.000000379    0.000000331   accuracy binary     0.576
 8     5     5          2 0.000000379    0.000000331   roc_auc  binary     0.617
 9     5    26          3 0.0100         0.0000000167  accuracy binary     0.611
10     5    26          3 0.0100         0.0000000167  roc_auc  binary     0.653
# ... with 50 more rows, and 3 more variables: n <int>, std_err <dbl>,
#   .config <chr>
```

Well we see 60 combinations and the metrics from them. But that doesn't mean much to us just eyeballing it. We want to see the best combination. There's a function to just show us the best one called … wait for it … `show_best`.

```
show_best(xgb_res, "accuracy")
```

```
# A tibble: 5 x 11
   mtry min_n tree_depth  learn_rate loss_reduction .metric  .estimator  mean
  <int> <int>      <int>       <dbl>          <dbl> <chr>    <chr>      <dbl>
1     7    39          5 0.0639        0.142        accuracy binary     0.615
2     6    13          8 0.0302        0.00571      accuracy binary     0.614
3     3    17          6 0.0210        0.0424       accuracy binary     0.614
4     6    36          6 0.00000128    0.000369     accuracy binary     0.614
5     4    28          5 0.000000100   0.0000000714 accuracy binary     0.612
# ... with 3 more variables: n <int>, std_err <dbl>, .config <chr>
```

The best combination as of this data update comes up with an accuracy of about 61.5 percent.

5.3 Finalizing our model

Let's capture our best set of hyperparameters so we can use them in our model.

```
best_acc <- select_best(xgb_res, "accuracy")
```

And now we put that into a final workflow. Pay attention to the main arguments in the output below.

```
final_xgb <- finalize_workflow(
  game_wflow,
  best_acc
)

final_xgb
```

```
== Workflow ====================================================================
Preprocessor: Recipe
Model: boost_tree()
```

```
-- Preprocessor ----------------------------------------------------------------
0 Recipe Steps

-- Model -----------------------------------------------------------------------
Boosted Tree Model Specification (classification)

Main Arguments:
  mtry = 7
  trees = 30
  min_n = 39
  tree_depth = 5
  learn_rate = 0.0638697307094032
  loss_reduction = 0.141518399318604

Computational engine: xgboost
```

There's our best set of hyperparameters. We've tuned this model to give the best possible set of results in those settings. Now we apply it like we have been doing all along.

5.3.1 Exercise 3: making our final workflow

We create a fit using our finalized workflow.

```
xg_fit <-
  ?????_xgb %>%
  fit(data = game_train)
```

We can see something things about that fit, including all the iterations of our XGBoost model. Remember: Boosted models work sequentially. One after the other. So you can see it at work. The error goes down with each iteration as we go down the gradient descent.

```
xg_fit %>%
  extract_fit_parsnip()
```

```
parsnip model object

##### xgb.Booster
raw: 82 Kb
call:
  xgboost::xgb.train(params = list(eta = 0.0638697307094032, max_depth = 5L,
    gamma = 0.141518399318604, colsample_bytree = 1, colsample_bynode = 1,
```

```
    min_child_weight = 39L, subsample = 1), data = x$data, nrounds = 30,
    watchlist = x$watchlist, verbose = 0, nthread = 1, objective = "binary:logistic")
params (as set within xgb.train):
  eta = "0.0638697307094032", max_depth = "5", gamma = "0.141518399318604", colsample_bytree
xgb.attributes:
  niter
callbacks:
  cb.evaluation.log()
# of features: 2
niter: 30
nfeatures : 2
evaluation_log:
    iter training_logloss
       1        0.6881808
       2        0.6837934
---
      29        0.6488872
      30        0.6486669
```

5.3.2 Exercise 4: prediction time

Now, like before, we can bind our predictions using our xg_fit to the game_train data.

```
trainresults <- game_train %>%
  bind_cols(predict(??_???, game_train))
```

5.3.3 Exercise 5: metrics

And now see how we did.

```
metrics(???????????, truth = team_result, estimate = .pred_class)
```

```
# A tibble: 2 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.621
2 kap      binary         0.242
```

How about the test data?

```r
testresults <- game_test %>%
  bind_cols(predict(xg_fit, game_test))

metrics(testresults, truth = team_result, estimate = .pred_class)
```

```
# A tibble: 2 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.613
2 kap      binary         0.225
```

Unlike the random forest, not nearly the drop in metrics between train and test.

6 LightGBM

6.1 The basics

LightGBM is another tree-based method that is similar to XGBoost but differs in ways that make it computationally more efficient. Where XGBoost and Random Forests are based on branches, LightGBM grows leaf-wise. Think of it like this – XGBoost uses lots of short trees with branches – it comes to a fork, makes a decision that reduces the amount of error the last tree got, and makes a new branch. LightGBM on the other hand, makes new leaves of each branch, which can mean lots of little splits, instead of big ones. That can lead to over-fitting on small datasets, but it also means it's much faster than XGBoost.

LightGBM also uses histograms of the data to make choices, where XGBoost is comptutationally optimizing those choices. Roughly translated – LightGBM is looking at the fat part of a normal distribution to make choices, where XGBoost is tuning parameters to find the optimal path forward. It's another reason why LightGBM is faster, but also not reliable with small datasets.

Let's implement a LightGBM model. We start with libraries.

```
library(tidyverse)
library(tidymodels)
library(hoopR)
library(zoo)
library(bonsai)

set.seed(1234)
```

We'll continue to use what we've done for feature engineering – a rolling window of points per possession for team and opponent. You should be quite familiar with this by now.

```
teamgames <- load_mbb_team_box(seasons = 2015:2023) %>%
  separate(field_goals_made_field_goals_attempted, into = c("field_goals_made","field_goal
  separate(three_point_field_goals_made_three_point_field_goals_attempted, into = c("three
  separate(free_throws_made_free_throws_attempted, into = c("free_throws_made","free_throw
  mutate_at(12:34, as.numeric)
```

```r
teamstats <- teamgames %>%
  mutate(
    team_score = ((field_goals_made-three_point_field_goals_made) * 2) + (three_point_fiel
    possessions = field_goals_attempted - offensive_rebounds + turnovers + (.475 * free_th
    ppp = team_score/possessions
  )

rollingteamstats <- teamstats %>%
  group_by(team_short_display_name, season) %>%
  arrange(game_date) %>%
  mutate(
    team_score = ((field_goals_made-three_point_field_goals_made) * 2) + (three_point_fiel
    team_rolling_ppp = rollmean(lag(ppp, n=1), k=5, align="right", fill=NA)
    ) %>%
  ungroup()

team_side <- rollingteamstats %>%
  select(
    game_id,
    team_id,
    team_short_display_name,
    opponent_id,
    game_date,
    season,
    team_score,
    team_rolling_ppp
    )

opponent_side <- team_side %>%
  select(-opponent_id) %>%
  rename(
    opponent_id = team_id,
    opponent_short_display_name = team_short_display_name,
    opponent_score = team_score,
    opponent_rolling_ppp = team_rolling_ppp
  ) %>%
  mutate(opponent_id = as.numeric(opponent_id)
)

games <- team_side %>% inner_join(opponent_side)
```

```
games <- games %>% mutate(
  team_result = as.factor(case_when(
    team_score > opponent_score ~ "W",
    opponent_score > team_score ~ "L"
))) %>% na.omit()

modelgames <- games %>%
  select(
    game_id,
    game_date,
    team_short_display_name,
    opponent_short_display_name,
    season,
    team_rolling_ppp,
    opponent_rolling_ppp,
    team_result
    ) %>%
  na.omit()
```

For this tutorial, we're going to create three models from three workflows so that we can compare a logistic regression to a random forest to a lightbgm model.

6.2 Setup

We're going to go through the steps of modeling again, starting with splitting our `modelgames` data.

6.2.1 Exercise 1: setting up your data

```
game_split <- initial_split(??????????, prop = .8)
game_train <- training(game_split)
game_test <- testing(game_split)
```

The recipe we'll create is the same for both, so we'll use it three times.

6.2.2 Exercise 2: setting up the receipe

So what data are we feeding into our recipe?

```
game_recipe <-
  recipe(team_result ~ ., data = game_?????) %>%
  update_role(game_id, game_date, team_short_display_name, opponent_short_display_name, se
  step_normalize(all_predictors())

summary(game_recipe)
```

```
# A tibble: 8 x 4
  variable                    type    role      source
  <chr>                       <chr>   <chr>     <chr>
1 game_id                     numeric ID        original
2 game_date                   date    ID        original
3 team_short_display_name     nominal ID        original
4 opponent_short_display_name nominal ID        original
5 season                      numeric ID        original
6 team_rolling_ppp            numeric predictor original
7 opponent_rolling_ppp        numeric predictor original
8 team_result                 nominal outcome   original
```

Now, we're going to create three different model specifications. The first will be the logistic regression model definition, the second will be the random forest, the third is the lightgbm.

```
log_mod <-
  logistic_reg() %>%
  set_engine("glm") %>%
  set_mode("classification")

rf_mod <-
  rand_forest() %>%
  set_engine("ranger") %>%
  set_mode("classification")

lightgbm_mod <-
  boost_tree() %>%
  set_engine("lightgbm") %>%
  set_mode(mode = "classification")
```

Now we have enough for our workflows. We have three models and one recipe.

6.2.3 Exercise 3: making workflows

```
log_workflow <-
  workflow() %>%
  add_model(???_mod) %>%
  add_recipe(????_recipe)

rf_workflow <-
  workflow() %>%
  add_model(??_mod) %>%
  add_recipe(????_recipe)

lightgbm_workflow <-
  workflow() %>%
  add_model(light???_mod) %>%
  add_recipe(game_recipe)
```

Now we can fit our models to the data.

6.2.4 Exercise 4: fitting our models

```
log_fit <-
  log_workflow %>%
  fit(data = ????_?????)

rf_fit <-
  rf_workflow %>%
  fit(data = ????_?????)

lightgbm_fit <-
  lightgbm_workflow %>%
  fit(data = ????_?????)
```

6.3 Prediction time

Now we can bind our predictions to the training data and see how we did.

```
logpredict <- log_fit %>% predict(new_data = game_train) %>%
  bind_cols(game_train)

logpredict <- log_fit %>% predict(new_data = game_train, type="prob") %>%
  bind_cols(logpredict)

rfpredict <- rf_fit %>% predict(new_data = game_train) %>%
  bind_cols(game_train)

rfpredict <- rf_fit %>% predict(new_data = game_train, type="prob") %>%
  bind_cols(rfpredict)

lightgbmpredict <- lightgbm_fit %>% predict(new_data = game_train) %>%
  bind_cols(game_train)

lightgbmpredict <- lightgbm_fit %>% predict(new_data = game_train, type="prob") %>%
  bind_cols(lightgbmpredict)
```

Now, how did we do?

6.3.1 Exercise 5: The first metrics

What prediction dataset do we feed into our metrics? Let's look first at the random forest, because it's a tree-based method just like lightgbm.

```
metrics(?????????, team_result, .pred_class)
```

```
# A tibble: 2 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.961
2 kap      binary         0.923
```

Same as last time, the random forest produces bonkers training numbers. Can you say overfit?

How about the lightgbm?

6.3.2 Exercise 6: LightGBM metrics

```
metrics(????????predict, team_result, .pred_class)
```

```
# A tibble: 2 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.628
2 kap      binary         0.256
```

About 63 percent accuracy. Which, if you'll recall, is a few percentage points better than logistic regression, and worse than random forest *WITH A HUGE ASTERISK*.

Remember: Where a model makes its money is in data that it has never seen before.

First, we look at random forest. The inevitable crash with random forests.

```
rftestpredict <- rf_fit %>% predict(new_data = game_test) %>%
  bind_cols(game_test)

rftestpredict <- rf_fit %>% predict(new_data = game_test, type="prob") %>%
  bind_cols(rftestpredict)

metrics(rftestpredict, team_result, .pred_class)
```

```
# A tibble: 2 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.577
2 kap      binary         0.153
```

Right at 57 percent. A little bit lower than logistic regression. But did they come to the same answers to get those numbers? No.

And now lightGBM.

```
lightgbmtestpredict <- lightgbm_fit %>% predict(new_data = game_test) %>%
  bind_cols(game_test)

lightgbmtestpredict <- lightgbm_fit %>% predict(new_data = game_test, type="prob") %>%
  bind_cols(lightgbmtestpredict)
```

```
metrics(lightgbmtestpredict, team_result, .pred_class)
```

```
# A tibble: 2 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.610
2 kap      binary         0.220
```

Our three models, based on our very basic feature engineering, are *still* only slightly better than flipping a coin. If we want to get better, we've still got work to do.