

Advanced Sports Data Analysis

Using machine learning to make predictions in sports

Matt Waite

1/4/23

Table of contents

1	Introduction	5
1.1	Requirements and Conventions	6
1.2	About this book	6
2	Installations	7
2.0.1	Part 1: Update and patch your operating system	7
2.0.2	Part 2: Install R and R Studio	12
2.0.3	Part 3: Installing R libraries	14
2.0.4	Part 4: Install Slack	15
2.0.5	Part 5: Install the tutorials	16
3	Modeling and logistic regression	19
3.1	The basics	19
3.2	Feature engineering	20
3.2.1	Exercise 1: setting up your data	20
3.2.2	Exercise 2: Opponent side	21
3.2.3	Exercise 3: Joining	22
3.3	Modeling	22
3.3.1	Exercise 4: Who won?	22
3.3.2	Exercise 5: Looking at the factors	23
3.3.3	Exercise 6: Releveling the factors	23
3.4	Visualizing the decision boundary	24
3.5	The logistic regression	25
3.5.1	Exercise 7: What are we splitting?	25
3.5.2	Exercise 8: Making a workflow	26
3.6	Evaluating the fit	27
3.6.1	Exercise 9: Metrics	28
3.6.2	Exercise 10: Confusion matrix	28
3.7	Comparing it to test data	29
3.7.1	Exercise 11: Testing	29
3.8	How well did it do with Nebraska?	30
4	Decision trees and random forests	32
4.1	The basics	32

4.2	Setup	34
4.2.1	Exercise 1: setting up your data	35
4.2.2	Exercise 2: setting up the receipe	35
4.2.3	Exercise 3: making workflows	36
4.2.4	Exercise 4: fitting our models	36
4.2.5	Exercise 5: The first metrics	37
4.2.6	Exercise 6: Random forest metrics	37
5	XGBoost	40
5.1	The basics	40
5.2	Hyperparameters	43
5.2.1	Exercise 1: tuning	44
5.2.2	Exercise 2: making a workflow	44
5.2.3	Exercise 3: creating our cross-fold validation set	45
5.3	Finalizing our model	47
5.3.1	Exercise 4: making our final workflow	48
5.3.2	Exercise 5: prediction time	49
5.3.3	Exercise 6: metrics	49
6	LightGBM	51
6.1	The basics	51
6.2	Setup	53
6.2.1	Exercise 1: setting up your data	53
6.2.2	Exercise 2: setting up the receipe	54
6.2.3	Exercise 3: making workflows	55
6.2.4	Exercise 4: fitting our models	55
6.3	Prediction time	56
6.3.1	Exercise 5: The first metrics	56
6.3.2	Exercise 6: LightGBM metrics	57
7	Support Vector Machines	59
7.0.1	Exercise 1: setting up your data	61
7.0.2	Exercise 2: setting up the receipe	61
7.0.3	Exercise 3: making workflows	62
7.0.4	Exercise 4: fitting our models	63
7.1	Prediction time	63
7.1.1	Exercise 5: The first metrics	64
7.1.2	Exercise 6: SVM metrics	64
8	Making predictions with new games	66
8.1	Redoing the feature engineering	69

9	Using linear regression to predict a number	73
9.1	The basics	73
9.2	Feature engineering	73
9.3	Setting up the modeling process	74
9.3.1	Exercise 1: Your first fit	75
9.3.2	Exercise 2: What is the truth?	76
9.3.3	Exercise 3: How does it fare?	76
9.4	Multiple regression	77
9.4.1	Exercise 4: Adding another variable	77
10	Random forests to predict a number	80
10.1	The basics	80
10.1.1	Exercise 1: What data are we feeding the recipe?	81
10.1.2	Exercise 2: making workflows	82
10.1.3	Exercise 3: fitting our models	82
10.1.4	Exercise 4: The first metrics	83
10.1.5	Exercise 5: Random forest metrics	83
11	XGBoost for regression	85
11.1	The basics	85
11.2	Implementing XGBoost	86
12	LightGBM for regression	91
12.1	The basics	91
12.2	Implementing LightGBM	93

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.

2 Installations

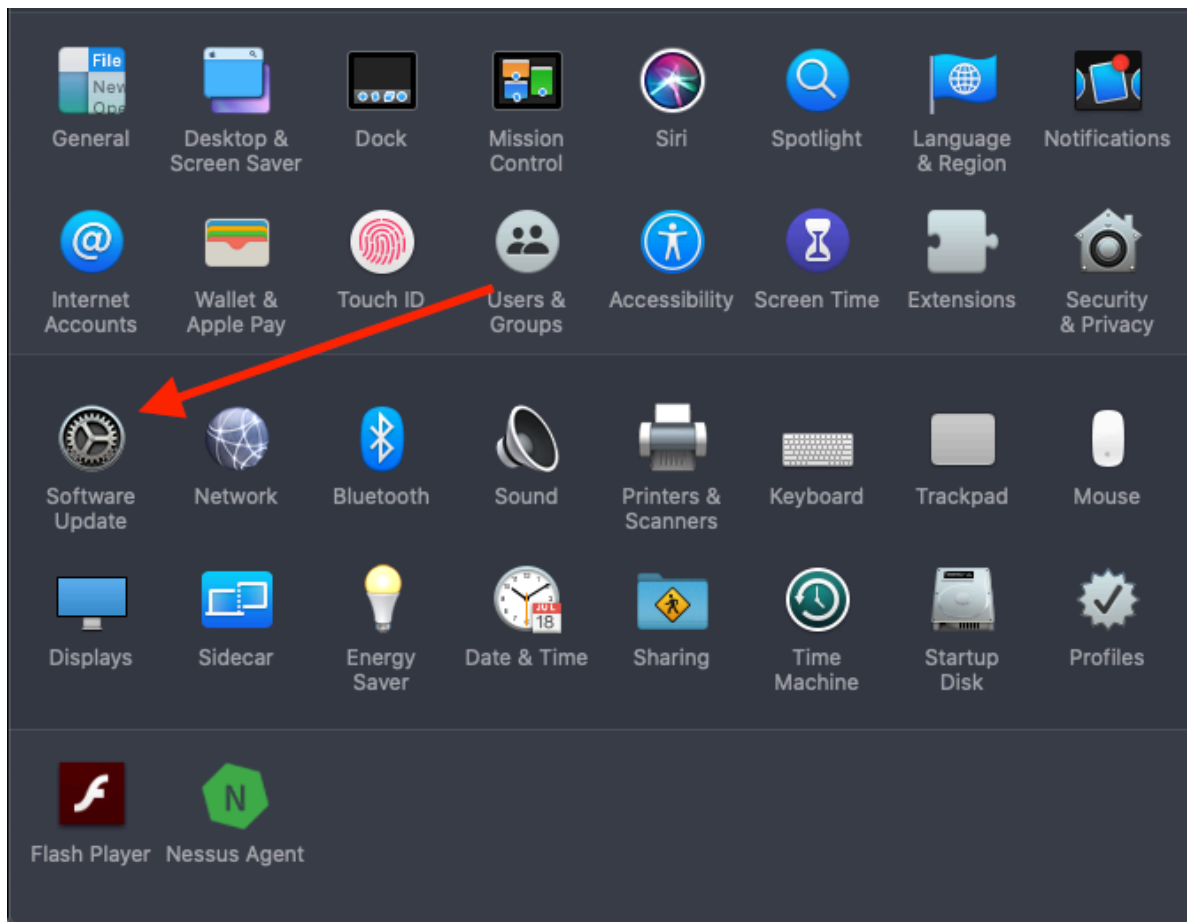
You're going to do things most of you aren't used to doing with your computer in this class. In order to do that, you need to clean up your computer. I've seen what your computer looks like. It's disgusting.

2.0.1 Part 1: Update and patch your operating system

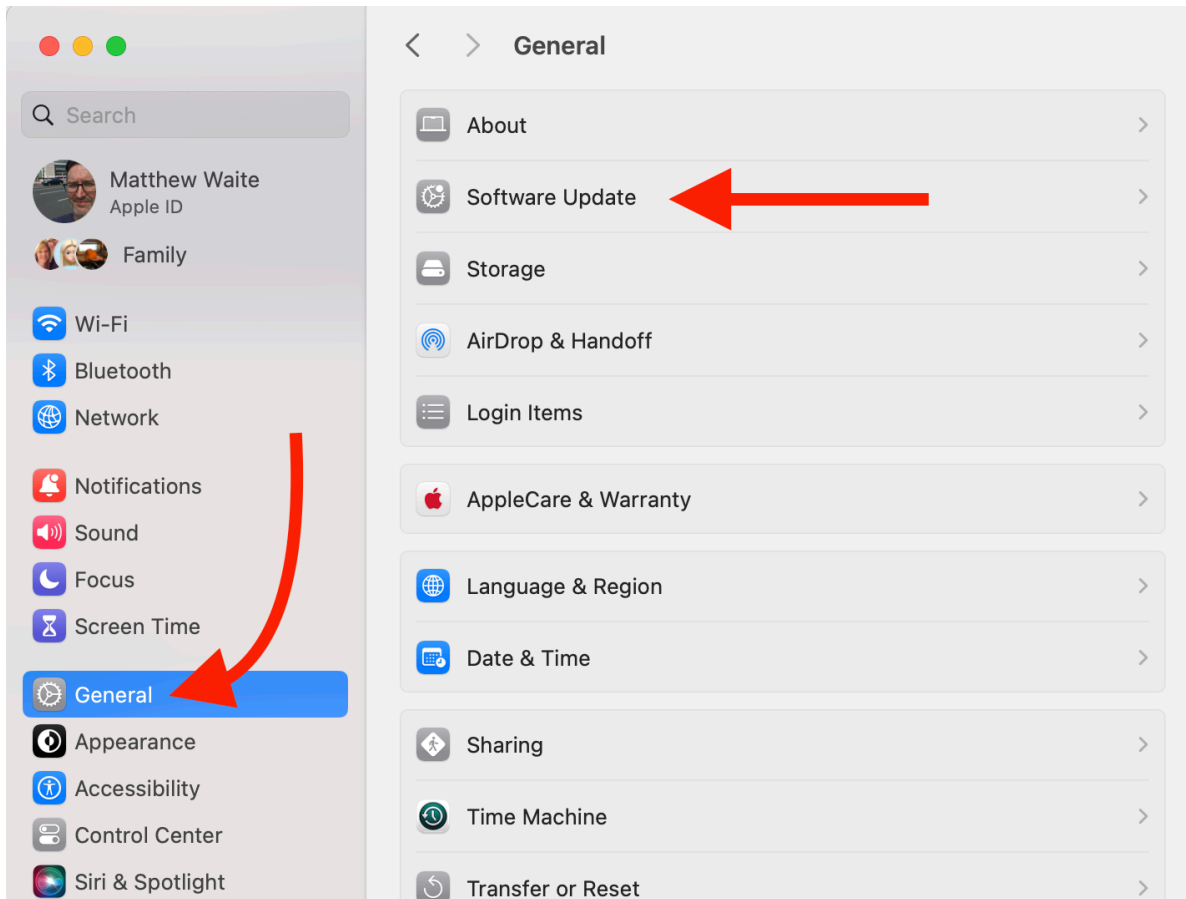
On a Mac:

1. Open System Preferences.

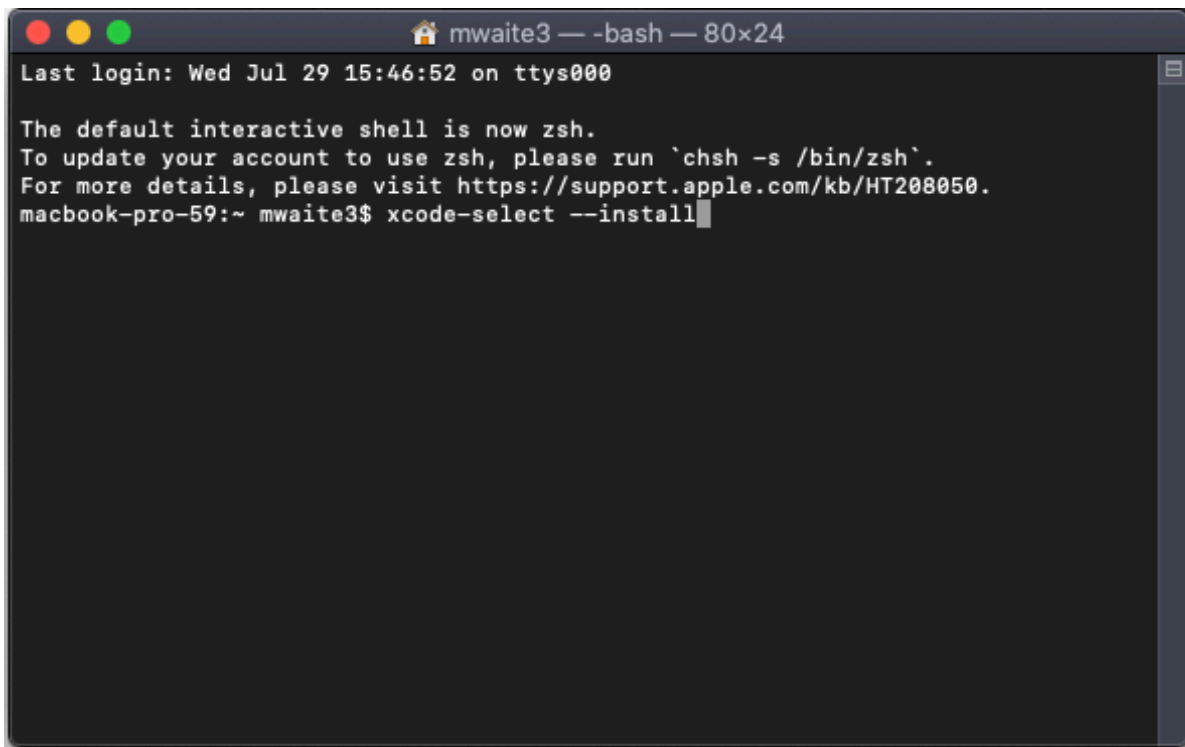
Depending on how old your Mac OS is, you might see this:



Or you might see this:



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.
3. If you aren’t on Sonoma 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.

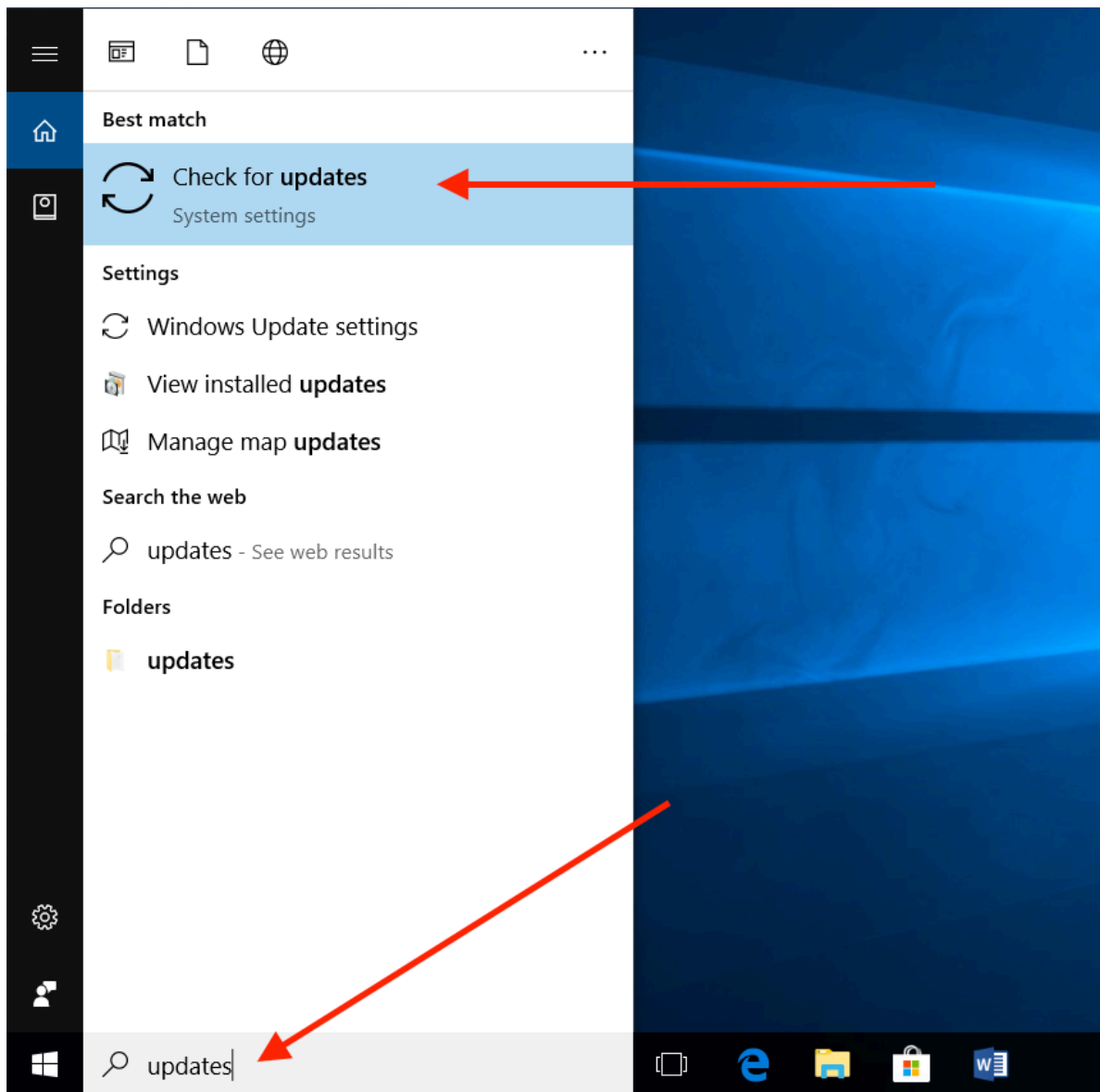
A terminal window with a dark background and light gray text. The window title bar shows three colored circles (red, yellow, green) on the left, a home icon followed by 'mwaite3' and '— -bash — 80x24' in the center, and a close button on the right. The terminal text reads: 'Last login: Wed Jul 29 15:46:52 on ttys000', 'The default interactive shell is now zsh.', 'To update your account to use zsh, please run `chsh -s /bin/zsh`.', 'For more details, please visit https://support.apple.com/kb/HT208050.', and 'macbook-pro-59:~ mwaite3\$ xcode-select --install' with a cursor at the end of the last line.

```
mwaite3 — -bash — 80x24
Last login: Wed Jul 29 15:46:52 on ttys000

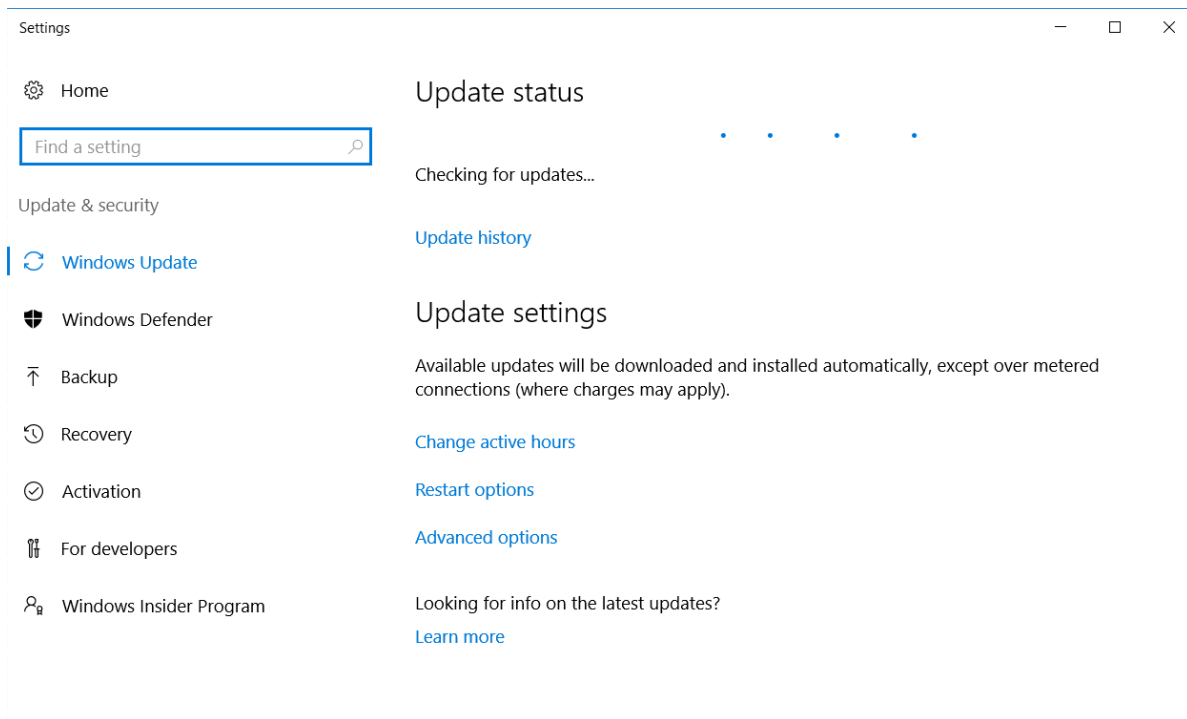
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
macbook-pro-59:~ mwaite3$ xcode-select --install
```

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 install those updates regularly. Don't ignore them. Don't snooze them. Install them.

2.0.2 Part 2: Install R and R Studio

1. Go [here](#). Go to Step 1 and click Download and Install R
 - If you're on a Mac, click on Download R for MacOS. If you have a newer Mac with an M1/M2/M3 chip, you want the arm64 version. If you're on an older Mac with an Intel chip, you want the X86_64 version.
 - If you're on Windows, install the base package *AND* install Rtools. When either downloads, run the executable and accept the defaults and license agreement.

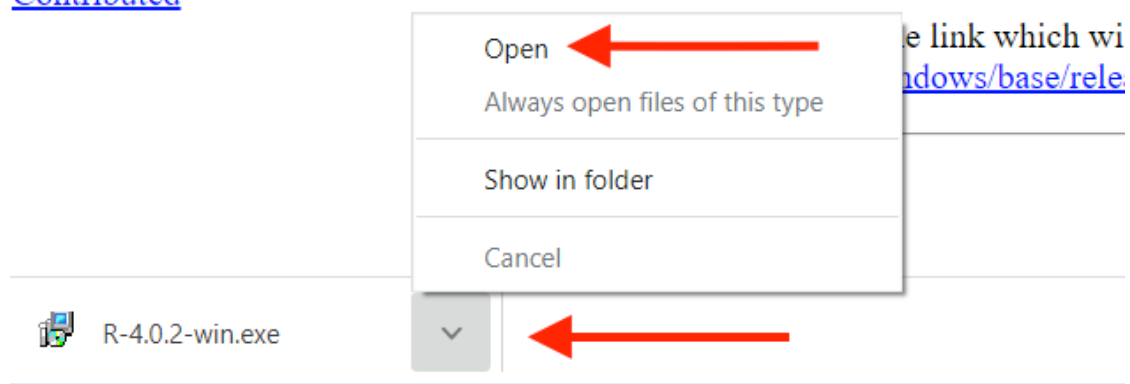
Documentation

[Manuals](#)

[FAQs](#)

[Contributed](#)

- Packages to this release are incorporated
- A build of the development version (v [snapshot build](#)).
- [Previous releases](#)



2. Go back to [here](#). Go to Step 2 and click R Studio Desktop for your version.

Mac users:

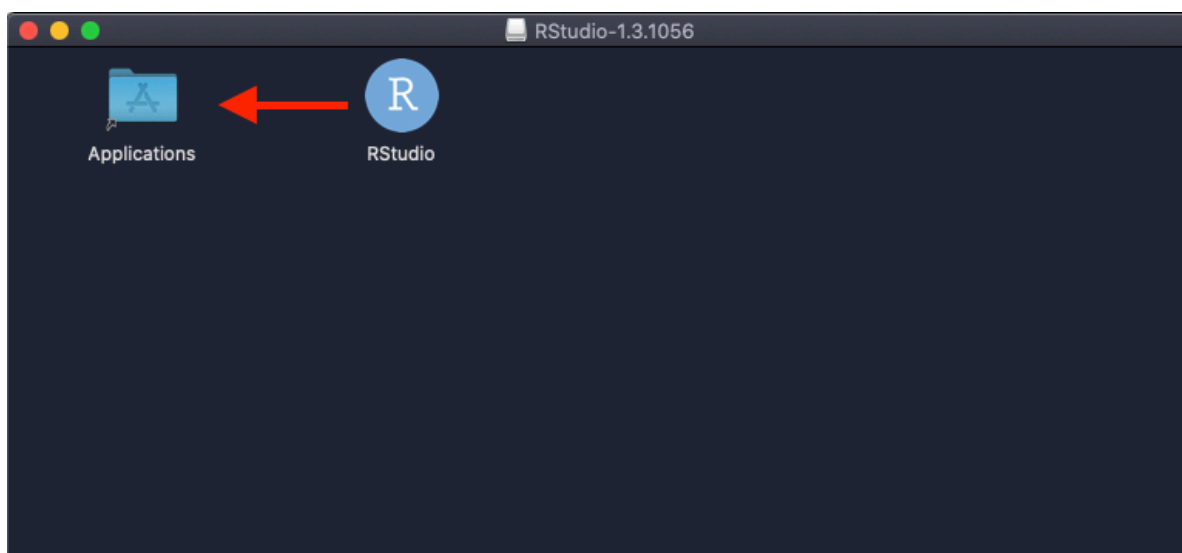


Figure 2.1: Make sure you drag the R Studio icon into the Applications folder icon.

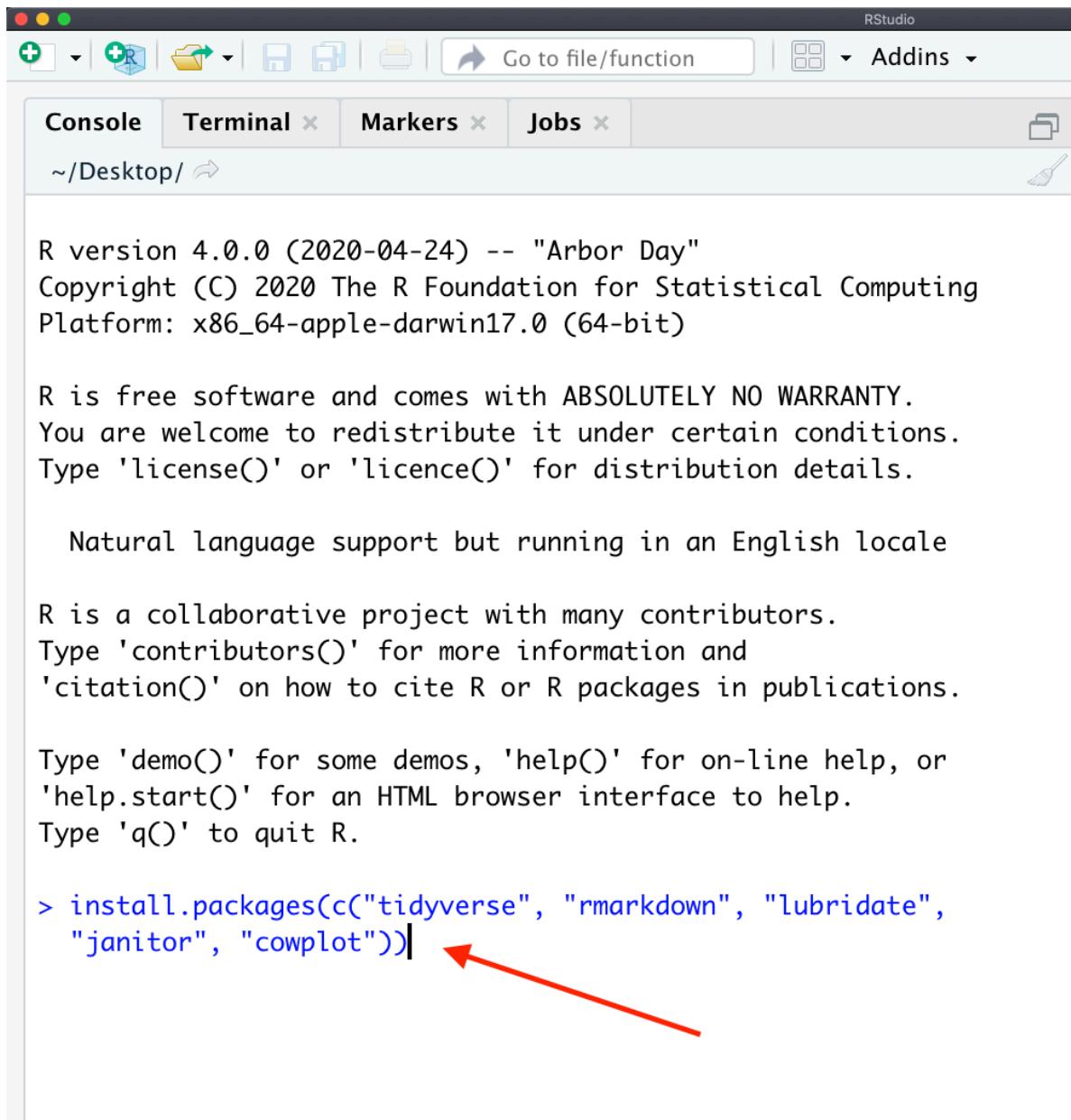
Windows users:

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", "lightgbm", "ranger", "xgboost",  
"kernlab", "corrr", "zoo"))
```



The screenshot shows the RStudio interface. The top bar includes the RStudio logo, a toolbar with icons for file operations, a search bar labeled 'Go to file/function', and a dropdown menu labeled 'Addins'. Below the top bar is a tabbed interface with 'Console', 'Terminal', 'Markers', and 'Jobs'. The 'Console' tab is active, showing the R startup message and a command to install packages. The command is `> install.packages(c("tidyverse", "rmarkdown", "lubridate", "janitor", "cowplot"))`, with a red arrow pointing to the closing parenthesis. The console output includes the R version (4.0.0), copyright information, and a list of helpful commands.

```
R version 4.0.0 (2020-04-24) -- "Arbor Day"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

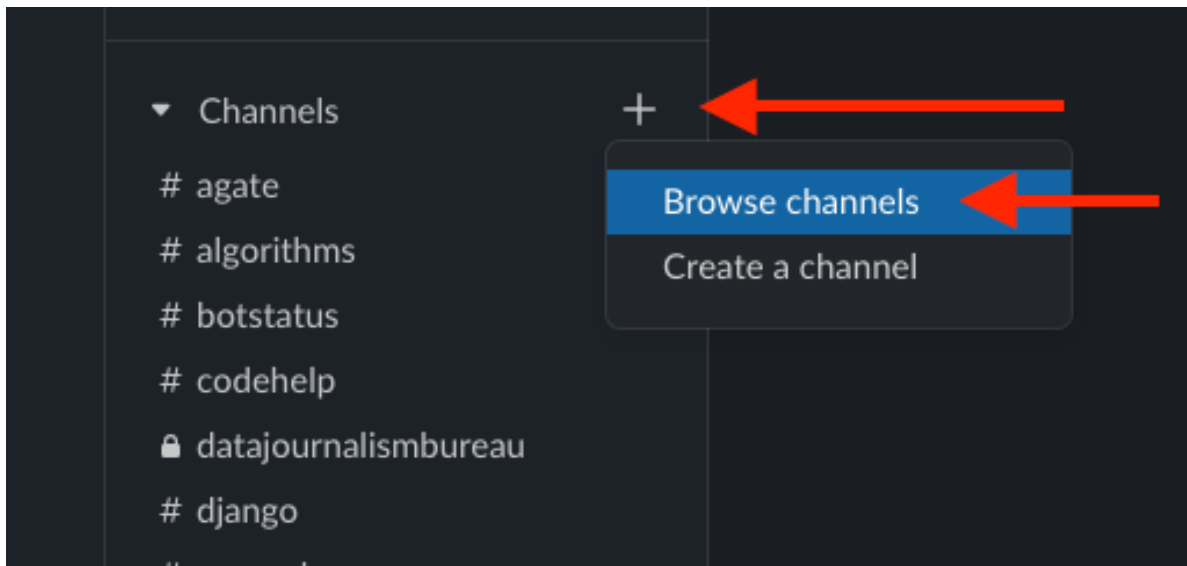
> install.packages(c("tidyverse", "rmarkdown", "lubridate",
  "janitor", "cowplot"))
```

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

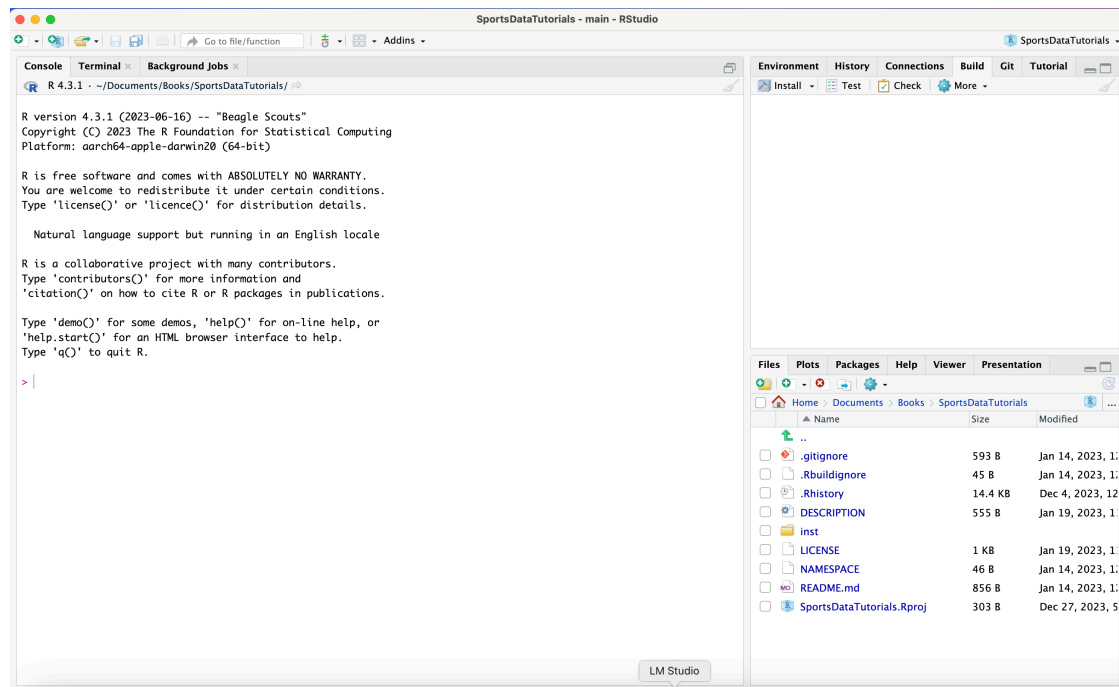
version. But I can usually solve your problem within a few minutes if you respond right away, and I know that you have your phone on you and are checking it. So the desktop version is for work, the phone version is for notifications.

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.

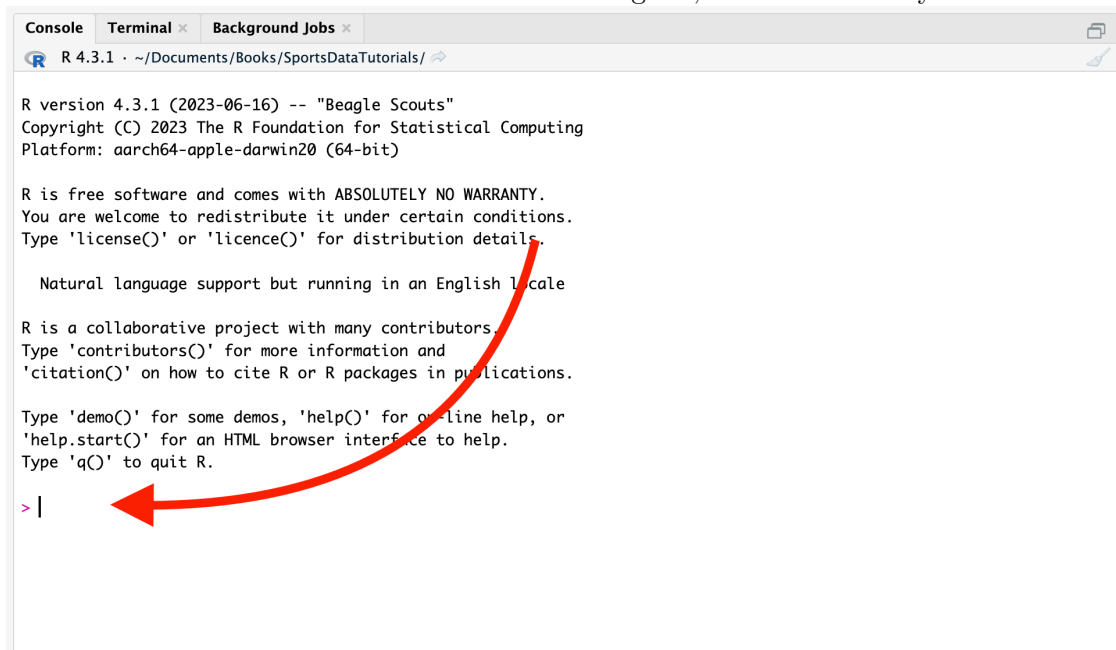


2.0.5 Part 5: Install the tutorials

To get the tutorials, do the following.



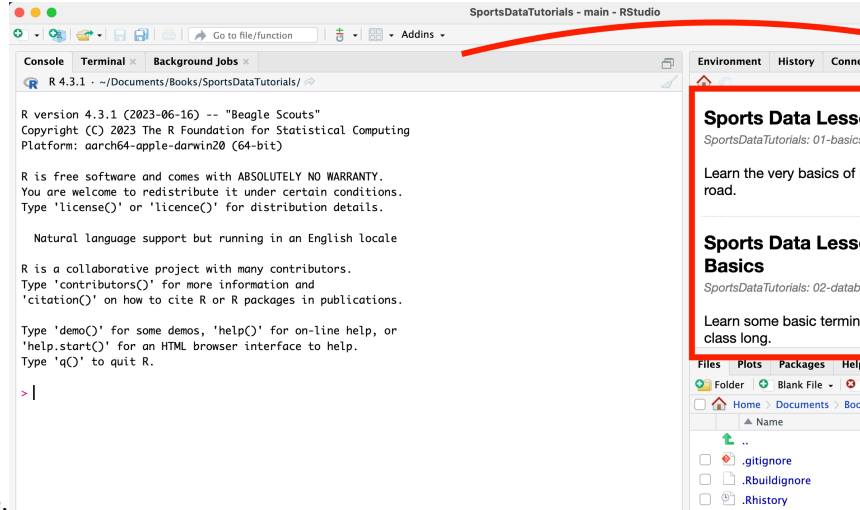
1. Open R Studio.
2. R Studio defaults to the console view. This is good, This is where you want to



be.

3. In the console, enter the following:
`devtools::install_github("mattwaite/SportsDataTutorials", force=TRUE)`

4. You should see some automated output. If you are told there are newer libraries and asked if you want to install them, just hit enter. When it is done, quit R Studio and restart it.



This is what it will look like when done.

5. Now do this:

```
devtools::install_github("mattwaite/AdvancedSportsDataTutorials")
```
6. Restart R Studio. You should now see both the Sports Data Tutorials and the Advanced Sports Data Tutorials in the tutorial pane.

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 then we’re going to remove all games involving non-Division I schools.

```
games <- load_mbb_team_box(seasons = 2015:2024)

nond1 <- games |> group_by(team_id, season) |> tally() |> filter(n < 10) |> select(team_id)
nond1 <- pull(nond1)
```

```
df <- games |> filter(!team_id %in% nond1 & !opponent_team_id %in% nond1)
```

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 offensive and defensive efficiency – points per 100 possessions.

Then, we're going to create season cumulative averages of those efficiencies, but we're going to lag them by one game. Why? Because we don't know what the offensive efficiency is for that game *before* that game. We only know what it was going into the game. So the lag gives us what each team's metrics are *before* they play the game. That's good. We need that for modeling. Otherwise, we're cheating.

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

3.2.1 Exercise 1: setting up your data

```
teamside <- df |>
  group_by(team, season) |>
  arrange(game_date) |>
  mutate(
    team_???????????? = field_goals_attempted - offensive_rebounds + turnovers + (.475 * fr
```

```

team_points_per_???????? = team_score/team_possessions,
team_defensive_points_per_possession = opponent_team_score/team_possessions,
team_offensive_efficiency = team_points_per_possession * 100,
team_defensive_efficiency = team_defensive_points_per_possession * 100,
team_season_offensive_efficiency = lag(cummean(team_offensive_efficiency), n=1),
team_season_defensive_efficiency = lag(cummean(team_defensive_efficiency), n=1),
score_margin = team_score - opponent_team_score,
absolute_score_margin = abs(score_margin)
) |>
filter(absolute_score_margin <= 40)

```

Now we're going to repeat the process for the opponent, but we're going to use some tricks here. Because we have box score data, it means every game is in here twice. So every team in one box score is the opponent in another. So we're going to use that in our favor to create the opponent side of the equation here and then we'll join it back together afterwards. We start by dropping the `opponent_team_id`, which we'll get back by renaming the `team_id` to that, then renaming our team stats to opponent stats.

3.2.2 Exercise 2: Opponent side

```

opponentside <- teamside |>
  select(-?????????????) |>
  rename(
    opponent_team_id = team_id,
    opponent_season_offensive_efficiency = team_season_offensive_efficiency,
    opponent_season_defensive_efficiency = team_season_defensive_efficiency
  ) |>
  select(
    game_id,
    opponent_team_id,
    opponent_season_offensive_efficiency,
    opponent_season_defensive_efficiency
  )

```

Now, let's join them.

3.2.3 Exercise 3: Joining

```
bothsides <- ???????? |> inner_join(?????????)
```

Joining with ``by = join_by(game_id, opponent_team_id)``

3.3 Modeling

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 the cumulative average efficiency. 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.

3.3.1 Exercise 4: Who won?

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 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.

The logic? If the team score is greater than the opponent team score, that's a win. Otherwise, they take the L as the kids say these days.

```
bothsides <- bothsides |> mutate(  
  team_result = as.factor(case_when(  
    team_score > ???????????? ~ "W",  
    opponent_team_score > ?????????? ~ "L"  ))
```

```
))) |> na.omit()
```

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

3.3.2 Exercise 5: 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(bothsides$????_???????)
```

```
[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`.

3.3.3 Exercise 6: Releveling the factors

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

```
levels(bothsides$team_result)
```

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

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

```
modelgames <- bothsides |>
  select(
    game_id,
    game_date,
    team_short_display_name,
    opponent_team_short_display_name,
    season,
    team_season_offensive_efficiency,
    team_season_defensive_efficiency,
    opponent_season_offensive_efficiency,
    opponent_season_defensive_efficiency,
```

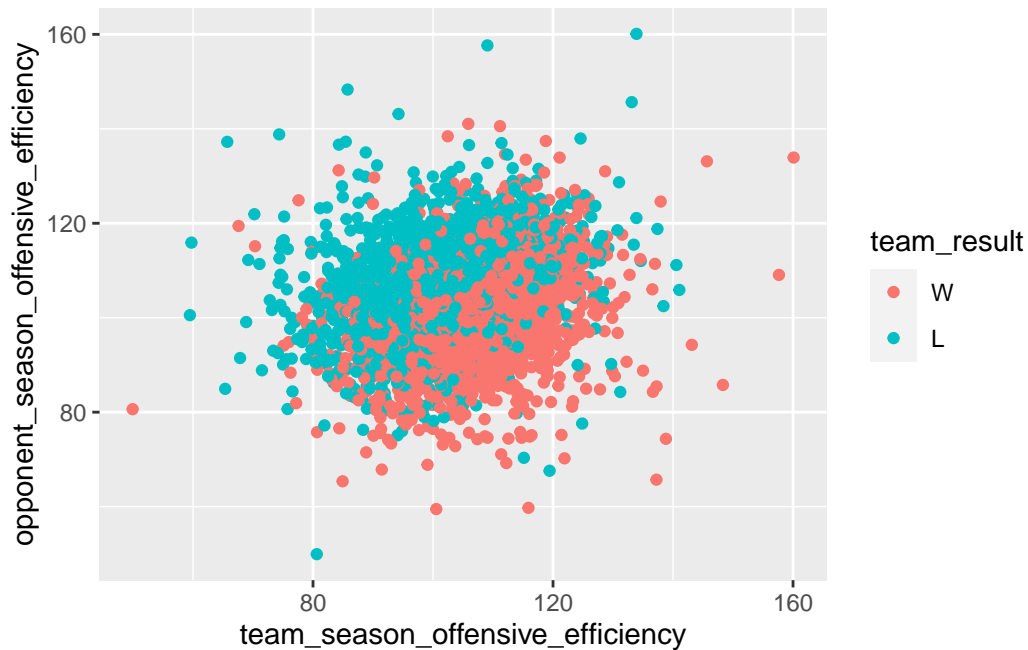
```
team_result  
) |> na.omit()
```

3.4 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 scores this efficiently and the opponent scores this efficiently, 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=bothsides, aes(x=team_season_offensive_efficiency, y=opponent_season_offensive_ef
```



3.5 The logistic regression

To create a model, we have to go through a process. That process starts with splitting data where we know the outcomes 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`.

3.5.1 Exercise 7: 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_team_short_display_name,
    step_normalize(all_predictors()))
```

```
summary(log_recipe)
```

```
# A tibble: 10 x 4
```

	variable	type	role	source
	<chr>	<list>	<chr>	<chr>
1	game_id	<chr [2]>	ID	original
2	game_date	<chr [1]>	ID	original
3	team_short_display_name	<chr [3]>	ID	original
4	opponent_team_short_display_name	<chr [3]>	ID	original
5	season	<chr [2]>	ID	original
6	team_season_offensive_efficiency	<chr [2]>	predictor	original
7	team_season_defensive_efficiency	<chr [2]>	predictor	original
8	opponent_season_offensive_efficiency	<chr [2]>	predictor	original
9	opponent_season_defensive_efficiency	<chr [2]>	predictor	original
10	team_result	<chr [3]>	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.

3.5.2 Exercise 8: 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)
```

3.6 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)

head(trainpredict)

# A tibble: 6 x 11
  .pred_class game_id game_date team_short_display_n~1 opponent_team_short_~2
  <fct>      <int> <date>      <chr>                                <chr>
1 W          401581565 2023-11-24 FAU                      Texas A&M
2 W          401581511 2023-11-19 Weber St                Colgate
3 W          401582025 2023-12-22 UMass Lowell              Boston U
4 L          401574608 2023-11-24 Fairfield                New Hampshire
5 L          401589495 2024-01-21 Long Island                C Connecticut
6 W          401587134 2023-12-30 St John's                Hofstra
# i abbreviated names: 1: team_short_display_name,
#   2: opponent_team_short_display_name
# i 6 more variables: season <int>, team_season_offensive_efficiency <dbl>,
#   team_season_defensive_efficiency <dbl>,
#   opponent_season_offensive_efficiency <dbl>,
#   opponent_season_defensive_efficiency <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)

head(trainpredict)
```

```
# A tibble: 6 x 13
  .pred_W .pred_L .pred_class game_id game_date team_short_display_name
  <dbl>   <dbl> <fct>      <int> <date>      <chr>
1  0.544  0.456 W          401581565 2023-11-24 FAU
2  0.648  0.352 W          401581511 2023-11-19 Weber St
```

```

3  0.659  0.341 W          401582025 2023-12-22 UMass Lowell
4  0.223  0.777 L          401574608 2023-11-24 Fairfield
5  0.213  0.787 L          401589495 2024-01-21 Long Island
6  0.631  0.369 W          401587134 2023-12-30 St John's
# i 7 more variables: opponent_team_short_display_name <chr>, season <int>,
#   team_season_offensive_efficiency <dbl>,
#   team_season_defensive_efficiency <dbl>,
#   opponent_season_offensive_efficiency <dbl>,
#   opponent_season_defensive_efficiency <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`.

3.6.1 Exercise 9: Metrics

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

```

# A tibble: 2 x 3
  .metric .estimator .estimate
  <chr>    <chr>        <dbl>
1 accuracy binary      0.667
2 kap     binary      0.334

```

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.

3.6.2 Exercise 10: Confusion matrix

```

trainpredict |>
  conf_mat(???_result, .pred_?????)

```

	Truth	
Prediction	W	L
W	1838	918
L	922	1846

3.7 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.

3.7.1 Exercise 11: Testing

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

```
# A tibble: 2 x 3
  .metric .estimator .estimate
  <chr>    <chr>        <dbl>
1 accuracy binary        0.670
2 kap     binary        0.340
```

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 458 221
      L 235 468

```

How does that compare to the training data?

3.8 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 == 2024)

nutest <- testpredict |> filter(team_short_display_name == "Nebraska" & season == 2024)

bind_rows(nutrain, nutest) |>
  arrange(game_date) |>
  select(.pred_W, .pred_class, team_result, team_short_display_name, opponent_team_short_d
gt()

```

.pred_W	.pred_class	team_result	team_short_display_name	opponent_team_short_display_name
0.9830817	W	W	Nebraska	Rider
0.7665254	W	W	Nebraska	Oregon St
0.6680374	W	W	Nebraska	Duquesne
0.9405191	W	W	Nebraska	Fullerton
0.4532627	L	L	Nebraska	Creighton
0.5431983	W	L	Nebraska	Minnesota
0.5008632	W	W	Nebraska	Michigan St
0.4898538	L	W	Nebraska	Kansas St
0.7587208	W	W	Nebraska	North Dakota
0.8574548	W	W	Nebraska	SC State
0.6635577	W	W	Nebraska	Indiana
0.5284632	W	L	Nebraska	Wisconsin
0.3418171	L	W	Nebraska	Purdue
0.5302580	W	L	Nebraska	Iowa
0.6331887	W	L	Nebraska	Rutgers
0.5004076	W	W	Nebraska	Northwestern
0.4364291	L	W	Nebraska	Ohio State
0.5474274	W	L	Nebraska	Maryland
0.4237322	L	W	Nebraska	Wisconsin

0.3593238	L	L	Nebraska	Illinois
0.4907129	L	L	Nebraska	Northwestern
0.6195147	W	W	Nebraska	Michigan
0.5751784	W	W	Nebraska	Penn State
0.6584005	W	W	Nebraska	Indiana

By our cumulative metrics, are there any surprises? Should we have beaten Creighton or Purdue?

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 than 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 = 2015:2024)

teamstats <- teamgames |>
  mutate(
    possessions = field_goals_attempted - offensive_rebounds + turnovers + (.475 * free_th
    ppp = team_score/possessions,
    oppp = opponent_team_score/possessions
```



```

)

rollingteamstats <- teamstats |>
  group_by(team_short_display_name, season) |>
  arrange(game_date) |>
  mutate(
    team_rolling_ppp = rollmean(lag(ppp, n=1), k=5, align="right", fill=NA),
    team_rolling_oppp = rollmean(lag(oppp, n=1), k=5, align="right", fill=NA)
  ) |>
  ungroup()

team_side <- rollingteamstats |>
  select(
    game_id,
    team_id,
    team_short_display_name,
    opponent_team_id,
    game_date,
    season,
    team_score,
    team_rolling_ppp,
    team_rolling_oppp
  )

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

games <- team_side |> inner_join(opponent_side)

```

Joining with `by = join_by(game_id, opponent_team_id, game_date, season)`

```

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

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

modelgames <- games |>
  select(
    game_id,
    game_date,
    team_short_display_name,
    opponent_short_display_name,
    season,
    team_rolling_ppp,
    team_rolling_oppp,
    opponent_rolling_ppp,
    opponent_rolling_oppp,
    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 recipe

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: 10 x 4

	variable <chr>	type <list>	role <chr>	source <chr>
1	game_id	<chr [2]>	ID	original
2	game_date	<chr [1]>	ID	original
3	team_short_display_name	<chr [3]>	ID	original
4	opponent_short_display_name	<chr [3]>	ID	original
5	season	<chr [2]>	ID	original
6	team_rolling_ppp	<chr [2]>	predictor	original
7	team_rolling_opp	<chr [2]>	predictor	original
8	opponent_rolling_ppp	<chr [2]>	predictor	original
9	opponent_rolling_opp	<chr [2]>	predictor	original
10	team_result	<chr [3]>	outcome	original

Now, we're going to create two different model specifications. The first will be the logistic regression model definition 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(log_mod) |>  
  add_recipe(recipe)   
  
rf_workflow <-  
  workflow() |>  
  add_model(rf_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 = train_data)   
  
rf_fit <-  
  rf_workflow |>  
  fit(data = train_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        0.639
2 kap      binary        0.278

```

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	0.993
2	kap	binary	0.985

Holy buckets! We made a model that's 99 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     0.636
2 kap     binary     0.273
```

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     0.618
2 kap     binary     0.237
```

Right at 62 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	W	L
W	5358	3091
L	3042	5371

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

	Truth	
Prediction	W	L
W	5174	3208
L	3226	5254

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 = 2015:2024)

teamstats <- teamgames |>
```



```

mutate(
  possessions = field_goals_attempted - offensive_rebounds + turnovers + (.475 * free_th
  ppp = team_score/possessions,
  oppp = opponent_team_score/possessions
)

rollingteamstats <- teamstats |>
  group_by(team_short_display_name, season) |>
  arrange(game_date) |>
  mutate(
    team_rolling_ppp = rollmean(lag(ppp, n=1), k=5, align="right", fill=NA),
    team_rolling_oppp = rollmean(lag(oppp, n=1), k=5, align="right", fill=NA)
  ) |>
  ungroup()

team_side <- rollingteamstats |>
  select(
    game_id,
    team_id,
    team_short_display_name,
    opponent_team_id,
    game_date,
    season,
    team_score,
    team_rolling_ppp,
    team_rolling_oppp
  )

opponent_side <- team_side |>
  select(-opponent_team_id) |>
  rename(
    opponent_team_id = team_id,
    opponent_short_display_name = team_short_display_name,
    opponent_score = team_score,
    opponent_rolling_ppp = team_rolling_ppp,
    opponent_rolling_oppp = team_rolling_oppp
  ) |>
  mutate(opponent_id = as.numeric(opponent_team_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"
  )))

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

modelgames <- games |>
  select(
    game_id,
    game_date,
    team_short_display_name,
    opponent_short_display_name,
    season,
    team_rolling_ppp,
    team_rolling_oppp,
    opponent_rolling_ppp,
    opponent_rolling_oppp,
    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: 10 x 4

	variable	type	role	source
	<chr>	<list>	<chr>	<chr>
1	game_id	<chr [2]>	ID	original

```

2 game_date                <chr [1]> ID          original
3 team_short_display_name  <chr [3]> ID          original
4 opponent_short_display_name <chr [3]> ID          original
5 season                  <chr [2]> ID          original
6 team_rolling_ppp        <chr [2]> predictor original
7 team_rolling_opp        <chr [2]> predictor original
8 opponent_rolling_ppp    <chr [2]> predictor original
9 opponent_rolling_opp    <chr [2]> predictor original
10 team_result            <chr [3]> 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):

- `mtry` – the number of predictors that will be randomly sampled at each split when making trees.
- 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:

- 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.

- 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.

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

5.2.1 Exercise 1: tuning

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

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

5.2.2 Exercise 2: making a workflow

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

```
game_wflow <-  
  workflow() |>  
  add_model(xg_mod) |>  
  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 multidimensional 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     3         6    10      4.14e-3  1.84e- 4
2     2         9    16      1.27e-4  6.39e- 3
3    10         2    23      1.02e-5  9.98e-10
4     5        12    20      4.56e-7  5.62e-10
5     9         8    33      1.79e+1  6.33e- 2
6     6         8    15      1.19e-9  2.62e- 3
7     5         5     7      1.44e-1  7.97e- 5
8     7        12    28      3.41e-8  4.06e- 5
9     1        13    25      1.04e+1  1.29e-10
10    4         5     6      1.37e+0  6.46e- 7
# i 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.

5.2.3 Exercise 3: 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 [60703/6745]> Fold01
2 <split [60703/6745]> Fold02
3 <split [60703/6745]> Fold03
4 <split [60703/6745]> Fold04
5 <split [60703/6745]> Fold05
6 <split [60703/6745]> Fold06
7 <split [60703/6745]> Fold07
8 <split [60703/6745]> Fold08
9 <split [60704/6744]> Fold09
10 <split [60704/6744]> 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 your 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     3     10       6  1.84e- 4    0.00414 accuracy binary    0.628
2     3     10       6  1.84e- 4    0.00414 roc_auc  binary    0.680
3     2     16       9  6.39e- 3    0.000127 accuracy binary    0.633
4     2     16       9  6.39e- 3    0.000127 roc_auc  binary    0.687
5    10     23       2  9.98e-10    0.0000102 accuracy binary    0.500
6    10     23       2  9.98e-10    0.0000102 roc_auc  binary    0.5
7     5     20      12  5.62e-10    0.000000456 accuracy binary    0.500
```

```

      8      5      20          12  5.62e-10    0.0000000456 roc_auc  binary    0.5
      9      9      33          8  6.33e- 2    17.9          accuracy binary    0.636
     10      9      33          8  6.33e- 2    17.9          roc_auc  binary    0.689
# i 50 more rows
# i 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     9     33         8  0.0633      17.9    accuracy binary    0.636
2     1     34        11  0.0170      0.0000000846 accuracy binary    0.635
3     2     16         9  0.00639     0.000127    accuracy binary    0.633
4     2     30        11  0.0276     0.00000530    accuracy binary    0.633
5     2     20         7  0.0000202    0.0372    accuracy binary    0.632
# i 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 = 9
  trees = 30
  min_n = 33
  tree_depth = 8
  learn_rate = 0.0633417088937915
  loss_reduction = 17.9327965928095

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 4: 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: 292.2 Kb
call:
  xgboost::xgb.train(params = list(eta = 0.0633417088937915, max_depth = 8L,
    gamma = 17.9327965928095, colsample_bytree = 1, colsample_bynode = 1,
    min_child_weight = 33L, subsample = 1), data = x$data, nrounds = 30,
    watchlist = x$watchlist, verbose = 0, nthread = 1, objective = "binary:logistic")
params (as set within xgb.train):
  eta = "0.0633417088937915", max_depth = "8", gamma = "17.9327965928095", colsample_bytree =
xgb.attributes:
  niter
callbacks:
  cb.evaluation.log()
# of features: 4
niter: 30
nfeatures : 4
evaluation_log:
  iter training_logloss
    1      0.6866136
    2      0.6808034
---
    29      0.6308232
    30      0.6303655

```

5.3.2 Exercise 5: 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 6: 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        0.646

```

```
2 kap      binary      0.293
```

How about the test data?

```
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     0.632
2 kap     binary     0.264
```

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 computationally 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 = 2015:2024)

teamstats <- teamgames |>
  mutate(
    possessions = field_goals_attempted - offensive_rebounds + turnovers + (.475 * free_th
    ppp = team_score/possessions,
```

```

    oppp = opponent_team_score/possessions
  )

rollingteamstats <- teamstats |>
  group_by(team_short_display_name, season) |>
  arrange(game_date) |>
  mutate(
    team_rolling_ppp = rollmean(lag(ppp, n=1), k=5, align="right", fill=NA),
    team_rolling_oppp = rollmean(lag(oppp, n=1), k=5, align="right", fill=NA)
  ) |>
  ungroup()

team_side <- rollingteamstats |>
  select(
    game_id,
    team_id,
    team_short_display_name,
    opponent_team_id,
    game_date,
    season,
    team_score,
    team_rolling_ppp,
    team_rolling_oppp
  )

opponent_side <- team_side |>
  select(-opponent_team_id) |>
  rename(
    opponent_team_id = team_id,
    opponent_short_display_name = team_short_display_name,
    opponent_score = team_score,
    opponent_rolling_ppp = team_rolling_ppp,
    opponent_rolling_oppp = team_rolling_oppp
  ) |>
  mutate(opponent_id = as.numeric(opponent_team_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"
    )))

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

modelgames <- games |>
  select(
    game_id,
    game_date,
    team_short_display_name,
    opponent_short_display_name,
    season,
    team_rolling_ppp,
    team_rolling_oppp,
    opponent_rolling_ppp,
    opponent_rolling_oppp,
    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 recipe

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: 10 x 4

	variable	type	role	source
	<chr>	<list>	<chr>	<chr>
1	game_id	<chr [2]>	ID	original
2	game_date	<chr [1]>	ID	original
3	team_short_display_name	<chr [3]>	ID	original
4	opponent_short_display_name	<chr [3]>	ID	original
5	season	<chr [2]>	ID	original
6	team_rolling_ppp	<chr [2]>	predictor	original
7	team_rolling_opp	<chr [2]>	predictor	original
8	opponent_rolling_ppp	<chr [2]>	predictor	original
9	opponent_rolling_opp	<chr [2]>	predictor	original
10	team_result	<chr [3]>	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      0.993
2 kap     binary      0.985
```

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      0.655
2 kap     binary      0.310
```

About 66 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      0.618
2 kap     binary      0.237
```

Right at 62 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      0.633  
2 kap     binary      0.267
```

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.

7 Support Vector Machines

The last method of statistical learning that we'll use is the Support Vector Machine. The concept to understand about the support vector machine is the concept of a hyperplane. First think of a hyperplane as a line – a means to separate wins and losses along an axis where a certain stat says this way is a win and that way is a loss. Each dot in the scatter plot is a support vector. The hyperplane separates the support vectors into their classes – it's the line between winning and losing. When you have just two stats like that, it's pretty easy. Where support vector machines get hard is when you have many predictors that create the hyperplane. Then, instead of a line, it becomes a multidimensional shape in a multidimensional space.

The further away from our hyperplane, the more confident we are in the prediction.

We're going to implement a support vector machine along side a logistic regression and a lightGBM model.

Let's start with our libraries, per usual.

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

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 = 2015:2024)

teamstats <- teamgames |>
  mutate(
    possessions = field_goals_attempted - offensive_rebounds + turnovers + (.475 * free_th
    ppp = team_score/possessions,
    oppp = opponent_team_score/possessions
  )
```

```

rollingteamstats <- teamstats |>
  group_by(team_short_display_name, season) |>
  arrange(game_date) |>
  mutate(
    team_rolling_ppp = rollmean(lag(ppp, n=1), k=5, align="right", fill=NA),
    team_rolling_oppp = rollmean(lag(oppp, n=1), k=5, align="right", fill=NA)
  ) |>
  ungroup()

team_side <- rollingteamstats |>
  select(
    game_id,
    team_id,
    team_short_display_name,
    opponent_team_id,
    game_date,
    season,
    team_score,
    team_rolling_ppp,
    team_rolling_oppp
  )

opponent_side <- team_side |>
  select(-opponent_team_id) |>
  rename(
    opponent_team_id = team_id,
    opponent_short_display_name = team_short_display_name,
    opponent_score = team_score,
    opponent_rolling_ppp = team_rolling_ppp,
    opponent_rolling_oppp = team_rolling_oppp
  ) |>
  mutate(opponent_id = as.numeric(opponent_team_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"
  )))

```

```

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

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

```

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

7.0.1 Exercise 1: setting up your data

```

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

```

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

7.0.2 Exercise 2: setting up the recipe

So what data are we feeding into our recipe?

```

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

summary(game_recipe)

```

```
# A tibble: 10 x 4
```

	variable	type	role	source
	<chr>	<list>	<chr>	<chr>
1	game_id	<chr [2]>	ID	original
2	game_date	<chr [1]>	ID	original
3	team_short_display_name	<chr [3]>	ID	original
4	opponent_short_display_name	<chr [3]>	ID	original
5	season	<chr [2]>	ID	original
6	team_rolling_ppp	<chr [2]>	predictor	original
7	team_rolling_opp	<chr [2]>	predictor	original
8	opponent_rolling_ppp	<chr [2]>	predictor	original
9	opponent_rolling_opp	<chr [2]>	predictor	original
10	team_result	<chr [3]>	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 lightgbm, the third is the svm.

```
log_mod <-  
  logistic_reg() %>%  
  set_engine("glm") %>%  
  set_mode("classification")  
  
lightgbm_mod <-  
  boost_tree() %>%  
  set_engine("lightgbm") %>%  
  set_mode(mode = "classification")  
  
svm_mod <-  
  svm_poly() %>%  
  set_engine("kernlab") %>%  
  set_mode("classification")
```

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

7.0.3 Exercise 3: making workflows

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

```

lightgbm_workflow <-
  workflow() %>%
  add_model(lightgbm_mod) %>%
  add_recipe(game_recipe)

svm_workflow <-
  workflow() %>%
  add_model(svm_mod) %>%
  add_recipe(game_recipe)

```

Now we can fit our models to the data.

7.0.4 Exercise 4: fitting our models

```

log_fit <-
  log_workflow %>%
  fit(data = game_train)

lightgbm_fit <-
  lightgbm_workflow %>%
  fit(data = game_train)

svm_fit <-
  svm_workflow %>%
  fit(data = game_train)

```

Setting default kernel parameters

7.1 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)

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)

svmpredict <- svm_fit %>% predict(new_data = game_train) %>%
  bind_cols(game_train)

svmpredict <- svm_fit %>% predict(new_data = game_train, type="prob") %>%
  bind_cols(svmpredict)

```

Now, how did we do?

7.1.1 Exercise 5: The first metrics

What prediction dataset do we feed into our metrics? Let's look first at the lightGBM.

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

```

# A tibble: 2 x 3
  .metric .estimator .estimate
  <chr>    <chr>        <dbl>
1 accuracy binary        0.655
2 kap      binary        0.310

```

And now the SVM.

7.1.2 Exercise 6: SVM metrics

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

```

# A tibble: 2 x 3
  .metric .estimator .estimate
  <chr>    <chr>        <dbl>
1 accuracy binary        0.639
2 kap      binary        0.278

```


Looks like the LightGBM did a little better than the SVM on training. But remember: Where a model makes its money is in data that it has never seen before.

First, we look at 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      0.633  
2 kap      binary      0.267
```

Right at 63 percent. And now SVM.

```
svmtestpredict <- svm_fit %>% predict(new_data = game_test) %>%  
  bind_cols(game_test)  
  
svmtestpredict <- svm_fit %>% predict(new_data = game_test, type="prob") %>%  
  bind_cols(svmtestpredict)  
  
metrics(svmtestpredict, team_result, .pred_class)  
  
# A tibble: 2 x 3  
  .metric .estimator .estimate  
  <chr>    <chr>      <dbl>  
1 accuracy binary      0.636  
2 kap      binary      0.272
```

Slightly better – very slightly. But it shows that SVM is a bit more robust to new data than the lightGBM.

8 Making predictions with new games

And we've arrived at tournament time. Now we're going to take all this modeling and apply it to new games. To do this, it takes a few extra steps that only make sense after you've done them.

Those steps are:

1. We're going to build our models the way we have been doing so all along. We need a one game lag to mimic not knowing the stats before the game. But now that we've worked with them this much, we should have an idea of what model works best for us, so we're only going to do that one model. We don't need multiple. We'll stop at the fit.
2. After we have a fit, we have to redo our feature engineering but this time without all the lags. We're now at a point where the data we have is what we need. No more shifting it back one game because we're officially in the future.
3. The, we have to go through a process of joining to get the two teams we need for a particular matchup.
4. Once we have our games, we can apply the fit to them and get a prediction.

It seems like a lot, and in terms of lines of code, it is. But it's really a lot of copy paste work. Let's get started.

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

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 = 2015:2024)

teamstats <- teamgames |>
  mutate(
```

```

    possessions = field_goals_attempted - offensive_rebounds + turnovers + (.475 * free_th
    ppp = team_score/possessions,
    oppp = opponent_team_score/possessions
  )

rollingteamstats <- teamstats |>
  group_by(team_short_display_name, season) |>
  arrange(game_date) |>
  mutate(
    team_rolling_ppp = rollmean(lag(ppp, n=1), k=5, align="right", fill=NA),
    team_rolling_oppp = rollmean(lag(oppp, n=1), k=5, align="right", fill=NA)
  ) |>
  ungroup()

team_side <- rollingteamstats |>
  select(
    game_id,
    team_id,
    team_short_display_name,
    opponent_team_id,
    game_date,
    season,
    team_score,
    team_rolling_ppp,
    team_rolling_oppp
  )

opponent_side <- team_side |>
  select(-opponent_team_id) |>
  rename(
    opponent_team_id = team_id,
    opponent_short_display_name = team_short_display_name,
    opponent_score = team_score,
    opponent_rolling_ppp = team_rolling_ppp,
    opponent_rolling_oppp = team_rolling_oppp
  ) |>
  mutate(opponent_id = as.numeric(opponent_team_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"
  )))

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

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

```

We're going to go through the steps of modeling again, starting with splitting our `modelgames` data and going all the way down to the fit. For your notebooks, you only need one model and one fit, so it's time to clean it all up.

```

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

game_recipe <-
  recipe(team_result ~ ., data = game_train) |>
  update_role(game_id, game_date, team_short_display_name, opponent_short_display_name, season,
    step_normalize(all_predictors())

svm_mod <-
  svm_poly() |>
  set_engine("kernlab") |>
  set_mode("classification")

```

```

svm_workflow <-
  workflow() |>
  add_model(svm_mod) |>
  add_recipe(game_recipe)

svm_fit <-
  svm_workflow |>
  fit(data = game_train)

```

Setting default kernel parameters

8.1 Redoing the feature engineering

Now that we have our fit based on known data, we need to redo our feature engineering so that we now have up to the moment data. You get that by just simply removing the lags. **Make sure you remove the lag(parts AND the n=1) parts both.**

```

rollingteamstats <- teamstats |>
  group_by(team_short_display_name, season) |>
  arrange(game_date) |>
  mutate(
    team_rolling_ppp = rollmean(ppp, k=5, align="right", fill=NA),
    team_rolling_opp = rollmean(opp, k=5, align="right", fill=NA),
  ) |>
  ungroup()

team_side <- rollingteamstats |>
  select(
    game_id,
    team_id,
    team_short_display_name,
    opponent_team_id,
    game_date,
    season,
    team_score,
    team_rolling_ppp,
    team_rolling_opp
  )

opponent_side <- team_side |>

```

```

select(-opponent_team_id) |>
rename(
  opponent_team_id = team_id,
  opponent_short_display_name = team_short_display_name,
  opponent_score = team_score,
  opponent_rolling_ppp = team_rolling_ppp,
  opponent_rolling_oppp = team_rolling_oppp
) |>
mutate(opponent_id = as.numeric(opponent_team_id)
)

games <- team_side |> inner_join(opponent_side)

```

Joining with ``by = join_by(game_id, opponent_team_id, game_date, season)``

```

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

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

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

```

Now we need to get the first games. Let's start with the first round of last seasons's Big Ten Tournament – we'll pretend it's happening today and the results are the same. To do

this, we're first going to make a tibble with the teams in the `team_short_display_name` and `opponent_short_display_name`.

```
round1games <- tibble(  
  team_short_display_name="Ohio State",  
  opponent_short_display_name="Wisconsin"  
) |> add_row(  
  team_short_display_name="Minnesota",  
  opponent_short_display_name="Nebraska"  
)
```

Now with that, we need to get all the team data for our game and join it to our round 1 games. This will get the latest information, drop the `team_result` and all the opponent information and then add it to our `round1games` dataframe. Then it will do it again, but this time for the opponent side of the game.

```
round1games <- modelgames |>  
  group_by(team_short_display_name) |>  
  filter(game_date == max(game_date) & season == 2024) |>  
  ungroup() |>  
  select(-team_result, -starts_with("opponent")) |>  
  right_join(round1games)
```

Joining with ``by = join_by(team_short_display_name)``

```
round1games <- modelgames |>  
  group_by(opponent_short_display_name) |>  
  filter(game_date == max(game_date) & season == 2024) |>  
  ungroup() |>  
  select(-team_result, -starts_with("team"), -game_id, -game_date, -season) |>  
  right_join(round1games)
```

Joining with ``by = join_by(opponent_short_display_name)``

Now, just like before, we apply our fits. The select at the end is just to move the right stuff up to the front of the table and make it easy for us to see what we need to see.

And who does our model think will win the first round?

```

round1 <- svm_fit |> predict(new_data = round1games) |>
  bind_cols(round1games) |> select(.pred_class, team_short_display_name, opponent_short_di

round1 <- svm_fit |> predict(new_data = round1games, type="prob") |>
  bind_cols(round1) |> select(.pred_class, .pred_W, .pred_L, team_short_display_name, oppo

round1

```

```

# A tibble: 2 x 12
  .pred_class .pred_W .pred_L team_short_display_name opponent_short_display_n~1
  <fct>       <dbl>   <dbl> <chr>                                <chr>
1 L           0.321   0.679 Ohio State                        Wisconsin
2 W           0.516   0.484 Minnesota                        Nebraska
# i abbreviated name: 1: opponent_short_display_name
# i 7 more variables: opponent_rolling_ppp <dbl>, opponent_rolling_oppp <dbl>,
#   game_id <int>, game_date <date>, season <int>, team_rolling_ppp <dbl>,
#   team_rolling_oppp <dbl>

```


9 Using linear regression to predict a number

9.1 The basics

Linear models are something you've understood since you took middle school math and learned the equation of a line. Remember $y = mx + b$? It's back. And, unlike what you complained bitterly in middle school, it's very, very useful.

What a linear model says, in words is that we can predict y if we multiply a value – a coefficient – by our x value offset with b , which is really the y -intercept, but think of it like where the line starts. Or, expressed as $y = mx + b$: `points = true_shooting_percentage * ? + some starting point`. Think of some starting point as what the score should be if the `true_shooting_percentage` is zero. Should be zero, right? Intuitively, yes, but it won't always work out so easily.

What we're trying to do here is predict how many fantasy points a player should score given their draft position (and later other stats). For this, we'll look at wide receivers, and we're going to build a model based on the past 10 draft classes.

9.2 Feature engineering

First we'll need libraries. You might need to install `corrr` with `install.packages("corrr")` run in your console.

```
library(tidyverse)
library(tidymodels)
library(corrr)
```

And we'll need some data. In this case, we've got draft data from `cfbfastR` and fantasy data from Pro-Football Reference.

```
fantasy <- read_csv("https://mattwaite.github.io/sportsdatafiles/fantasyfootball20132022.csv")

wr <- read_csv("https://mattwaite.github.io/sportsdatafiles/wr20132022.csv")
```

```
wrdrafted <- wr |>
  inner_join(fantasy, by=c("name"="Player", "year"="Season"))
```

That leaves us with a dataframe of 263 observations – drafted wide receivers with their fantasy stats attached to them.

Let's thin the herd here a bit and just get our selected stats for modeling. We're really just going to have a handful of things: name, year, their college team and the team that drafted them, their overall draft number, their pre-draft grade from ESPN and the number of fantasy points they scored in their first year in the league.

```
wrselected <- wrdrafted |>
  select(
    name,
    year,
    college_team,
    nfl_team,
    overall,
    pre_draft_grade,
    FantPt
  )
```

9.3 Setting up the modeling process

With most modeling tasks we need to start with setting a random number seed to aid our random splitting of data into training and testing.

```
set.seed(1234)
```

Random numbers play a large role in a lot of data science algorithms, so setting one helps our reproducibility.

After that, we split our data. There's a number of ways to do this – R has a bunch and you'll find all kinds of examples online – but Tidymodels has made this easy.

```
player_split <- initial_split(wrselected, prop = .8)

player_train <- training(player_split)
player_test <- testing(player_split)
```

Let's start with a simple linear regression with one variable. We're just going to use the overall draft position to predict fantasy points. How well does the draft pick do that? are top picks big point getters and low picks low point scorers? Is there a pattern?

A lot of what comes next is familiar to you. We're going to make a model, make a fit, then add the results to our training data and see where that gets us. The fit is made up of the `FantPt` and `overall` divided by a `~`, which can be verbalized as "is approximately modeled by." So `FantPt` is approximately modeled by `overall`.

Our metrics, though will say new things.

9.3.1 Exercise 1: Your first fit

```
lm_model <- linear_reg() |>
  set_engine("lm")

fit_lm <- lm_model |>
  fit(?????? ~ ???????, data = player_train)
```

We can look now at the pieces of the equation of a line here.

```
tidy(fit_lm, conf.int = TRUE)
```

```
# A tibble: 2 x 7
  term      estimate std.error statistic  p.value conf.low conf.high
<chr>      <dbl>     <dbl>     <dbl>   <dbl>   <dbl>    <dbl>
1 (Intercept)  89.8        5.48      16.4 4.10e-38  79.0     101.
2 overall    -0.365      0.0426     -8.57 3.58e-15 -0.449   -0.281
```

The two most important things to see here are the terms and the estimates. Start with `overall`. What that says is for every pick in the draft, a player should score about a third of a fantasy point *less* than the previous pick. So the first pick scores -.3, the second pick scores -.6 and so on. So the higher the pick, the lower the number. Thus our slope.

HOWEVER, the intercept has something to say about this. What the intercept says is that players start with about 87 fantasy points. The slope then adjusts them downwards each pick they go.

Think again about $y = mx + b$. We have our terms here: y is fantasy points, m is $-.3 \times$ the pick number and b is 87. Let's pretend for a minute that you were drafted with the 10th pick. That would mean, on the slope, you're down 3 points, so $-3 + 87$ is 84. Our model would predict the 10th pick of the draft, if they were a wide receiver, would score 84 fantasy points in their rookie season.

9.3.2 Exercise 2: What is the truth?

That sounds good, but how good is our model?

```
trainresults <- player_train |>
  bind_cols(predict(fit_lm, player_train))

metrics(trainresults, truth = ??????, estimate = .pred)
```



```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>       <dbl>
1 rmse    standard      41.8
2 rsq     standard       0.280
3 mae     standard      32.9
```

Our first step in evaluating a linear model is to get the r-squared value. The yardstick library (part of Tidymodels) does this nicely. We tell it to produce `metrics` on a dataset, and we have to tell it what the real world result is (the truth column) and what the estimate column is (`.pred`).

We have two numbers we're going to focus on – `rsq` or `r squared`, and `rmse` or `root mean squared error`. `R squared` is the amount that changes in overall predict changes in fantasy points. You can read it as a percentage. So changes in overall draft position account for about 28 percent of the change in fantasy points. Not great, but we're just starting.

The `rmse` is how off your predictions are on average. In this case, our fantasy point prediction is off by 40 (plus or minus) on average. Given that we started with 87, that's also not great.

9.3.3 Exercise 3: How does it fare?

We need to make those numbers smaller. But first, we should see how it does with test data.

```
testresults <- player_???? |>
  bind_cols(predict(fit_lm, player_????))

metrics(testresults, truth = FantPt, estimate = .pred)
```



```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>       <dbl>
```

```
1 rmse      standard      41.4
2 rsq       standard       0.291
3 mae       standard      31.4
```

Our r squared is up a bit, but so is our rmse. So we didn't change all much, which is good. That means our model is robust to new data.

9.4 Multiple regression

The problem with simple regressions? They're simple. Anyone who has watched a sport knows there's a lot more to the outcome than just one number.

Enter the multiple regression.

Multiple regressions are a step toward reality – where more than one thing influences the outcome. However, the more variance we attempt to explain, the more error and uncertainty we introduce into our model.

To add a variable to your regression model to make a multiple regression model, you simply use `+` and add it in. Let's add `pre_draft_grade`.

9.4.1 Exercise 4: Adding another variable

```
lm_model <- linear_reg() |>
  set_engine("lm")

fit_lm <- lm_model |>
  fit(FantPt ~ overall + ???_????_????, data = player_train)
```

Let's look at the pieces of the equation of a line again.

```
tidy(fit_lm, conf.int = TRUE)
```

```
# A tibble: 3 x 7
  term          estimate std.error statistic   p.value conf.low conf.high
<chr>          <dbl>     <dbl>     <dbl>   <dbl>   <dbl>   <dbl>
1 (Intercept)    56.8       27.4        2.07 0.0398    2.68   111.
2 overall       -0.302     0.0756       -3.99 0.0000953 -0.451 -0.153
3 pre_draft_grade 0.369     0.294        1.25 0.212    -0.212 0.950
```

So our intercept is five points lower. Our `overall` estimate is about the same – each pick lowers the fantasy point expectation – but the `pre_draft_grade` now adds five tenths of a point for each point of pre-draft grade.

How does that impact our draft model?

```
trainresults <- player_train |>
  bind_cols(predict(fit_lm, player_train))

metrics(trainresults, truth = FantPt, estimate = .pred)
```

```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>         <dbl>
1 rmse    standard        41.9
2 rsq     standard         0.287
3 mae     standard        32.7
```

Huh. Our `r squared` is almost unchanged and our `rmse` is up. What gives?

There are multiple ways to find the right combination of inputs to your models. With multiple regressions, the most common is the correlation matrix. We're looking to maximize `r-squared` by choosing inputs that are highly correlated to our target value, but *not* correlated with other things. Example: We can assume that overall draft pick and pre-draft grade are highly correlated to fantasy points, but the problem lies in if they are highly correlated to each other. If so, we're just adding error and not getting any new predictive value.

Using `corrr`, we can create a correlation matrix in a dataframe to find columns that are highly correlated with our target – `FantPt`. To do this, we need to select the columns we're working with – `overall` and `pre_draft_grade`.

```
wrselected |>
  select(FantPt, overall, pre_draft_grade) |>
  correlate()
```

```
# A tibble: 3 x 4
  term          FantPt overall pre_draft_grade
  <chr>         <dbl>   <dbl>         <dbl>
1 FantPt       NA      -0.532         0.451
2 overall     -0.532    NA          -0.814
3 pre_draft_grade 0.451   -0.814         NA
```

Reading this when you have a lot of numbers can be a lot, and it helps to take some notes as you go.

You read up and down and left and right – it's a matrix. Follow the FantPt row across to the overall column and you'll see they're about 50 percent negatively correlated – -1 is a perfect negative correlation. Now look to the right to pre_draft_grade. They're almost the same – but this time it's about 45 percent positively correlated.

Now look at the overall column, and go down to the pre_draft_grade. The correlation: -0.8141840. Remember that a -1 is a perfect negative correlation. For every 1 one goes up, the other goes down 1. That's really close to -1.

What does that mean? It means including both is going to just add error without adding much value. They're so similar. You pick the one that is more highly correlated with fantasy points – overall pick.

10 Random forests to predict a number

10.1 The basics

And now we return to decision trees and random forests. Recall that tree-based algorithms are based on decision trees, which are very easy to understand. A random forest is, as the name implies, a large number of decision trees, and they use a random choice of inputs at each fork in the tree. 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.

This time, we're going to clean up our code a bit and make it more like we are accustomed to with our previous work, where we'll make recipes and workflows.

As always, we start with libraries.

```
library(tidyverse)
library(tidymodels)

set.seed(1234)
```

And we'll need some data. We'll use our draft data from `cfbfastR` and fantasy data from Pro-Football Reference.

```
fantasy <- read_csv("https://mattwaite.github.io/sportsdatafiles/fantasyfootball20132022.csv")

wr <- read_csv("https://mattwaite.github.io/sportsdatafiles/wr20132022.csv")

wrdrafted <- wr |>
  inner_join(fantasy, by=c("name"="Player", "year"="Season"))
```

And again we have a dataframe of 263 observations – drafted wide receivers with their fantasy stats attached to them.

Let's narrow that down to just our columns we need:


```

wrselected <- wrdrafted |>
  select(
    name,
    year,
    college_team,
    nfl_team,
    overall,
    pre_draft_grade,
    FantPt
  ) |> na.omit()

```

Before we get to the recipe, let's split our data.

```

player_split <- initial_split(wrselected, prop = .8)

player_train <- training(player_split)
player_test <- testing(player_split)

```

Now we're ready to start.

10.1.1 Exercise 1: What data are we feeding the recipe?

```

player_recipe <-
  recipe(FantPt ~ ., data = player_?????) |>
  update_role(name, year, college_team, nfl_team, new_role = "ID")

summary(player_recipe)

```

```

# A tibble: 7 x 4
  variable      type      role      source
  <chr>         <list>   <chr>    <chr>
1 name         <chr [3]> ID      original
2 year         <chr [2]> ID      original
3 college_team <chr [3]> ID      original
4 nfl_team     <chr [3]> ID      original
5 overall      <chr [2]> predictor original
6 pre_draft_grade <chr [2]> predictor original
7 FantPt       <chr [2]> outcome  original

```

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

```
linear_mod <-
  linear_reg() |>
  set_engine("lm") |>
  set_mode("regression")

rf_mod <-
  rand_forest() |>
  set_engine("ranger") |>
  set_mode("regression")
```

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

10.1.2 Exercise 2: making workflows

```
linear_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.

10.1.3 Exercise 3: fitting our models

```
linear_fit <-
  linear_workflow |>
  fit(data = ???_?????)

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

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

```
linearpredict <-
  linear_fit |>
  predict(new_data = player_train) |>
  bind_cols(player_train)

rfpredict <-
  rf_fit |>
  predict(new_data = player_train) |>
  bind_cols(player_train)
```

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

10.1.4 Exercise 4: The first metrics

What prediction dataset do we feed into our metrics?

```
metrics(??????????????, FantPt, .pred)
```

```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>    <chr>         <dbl>
1 rmse    standard        40.9
2 rsq     standard         0.316
3 mae     standard        31.5
```

Same as last time. An r squared in the high 20s, an rmse in the 40s. Nothing to write home about. How did the random forest do?

10.1.5 Exercise 5: Random forest metrics

```
metrics(??predict, FantPt, .pred)
```

```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>    <chr>         <dbl>
1 rmse    standard        22.6
2 rsq     standard         0.812
3 mae     standard        17.2
```

Hopefully you learned your lesson the last time we did random forests – they have a habit of bringing up your hopes on training only to dash them on testing, especially when you have two highly correlated values.

Is that what happened here?

```
rftestpredict <-  
  rf_fit |>  
  predict(new_data = player_test) |>  
  bind_cols(player_test)  
  
metrics(rftestpredict, FantPt, .pred)
```

```
# A tibble: 3 x 3  
  .metric .estimator .estimate  
  <chr>   <chr>       <dbl>  
1 rmse    standard      49.3  
2 rsq     standard       0.157  
3 mae     standard      38.2
```

Indeed.

Safe to say we've reached the limits of overall draft pick and pre-draft grades for predictive value. Time to add more.

11 XGBoost for regression

11.1 The basics

And now we return to XGBoost, but now for regression. Recall that 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.

So far, our linear regression and random forest methods aren't that great. Does XGBoost, with its method of optimizing for reduced error, fare any better? Let's try, but this time we're going to add more information. We're going to add college receiving stats.

As always, we start with libraries.

```
library(tidyverse)
library(tidymodels)

set.seed(1234)
```

We're going to load a new data file this time. It's called `wrdraftedstats` and it now has wide receivers who were drafted, their fantasy stats, and their college career stats.

```
wrdraftedstats <- read_csv("https://mattwaite.github.io/sportsdatafiles/wrdraftedstats2013")
```

Let's narrow that down to just our columns we need. We're going to add `total_yards` and `total_touchdowns` to our data to see what happens to our predictions.

```
wrselected <- wrdraftedstats %>%
  select(
    name,
    year,
    college_team,
    nfl_team,
    overall,
```

```

    total_yards,
    total_touchdowns,
    FantPt
  ) %>% na.omit()

```

Before we get to the recipe, let's split our data.

```

player_split <- initial_split(wrselected, prop = .8)

player_train <- training(player_split)
player_test <- testing(player_split)

```

11.2 Implementing XGBoost

Our player recipe will remain unchanged because we're using the `.` notation to mean "everything that isn't an ID or a predictor."

```

player_recipe <-
  recipe(FantPt ~ ., data = player_train) %>%
  update_role(name, year, college_team, nfl_team, new_role = "ID")

summary(player_recipe)

```

```

# A tibble: 8 x 4
  variable      type  role      source
  <chr>         <chr> <chr>    <chr>
1 name         nominal ID      original
2 year         numeric ID      original
3 college_team nominal ID      original
4 nfl_team     nominal ID      original
5 overall      numeric predictor original
6 total_yards  numeric predictor original
7 total_touchdowns numeric predictor original
8 FantPt      numeric outcome  original

```

Our prediction will use the overall draft pick, their total yards in college and their total touchdowns in college. Does that predict Fantasy points better? Let's implement multiple models side by side.

```

linear_mod <-
  linear_reg() %>%
  set_engine("lm") %>%
  set_mode("regression")

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

xg_mod <- boost_tree(
  trees = tune(),
  learn_rate = tune(),
  tree_depth = tune(),
  min_n = tune(),
  loss_reduction = tune(),
  sample_size = tune(),
  mtry = tune(),
) %>%
  set_mode("regression") %>%
  set_engine("xgboost")

```

Now to create workflows.

```

linear_workflow <-
  workflow() %>%
  add_model(linear_mod) %>%
  add_recipe(player_recipe)

rf_workflow <-
  workflow() %>%
  add_model(rf_mod) %>%
  add_recipe(player_recipe)

xg_workflow <-
  workflow() %>%
  add_model(xg_mod) %>%
  add_recipe(player_recipe)

```

Now to tune the XGBoost model.

```

xgb_grid <- grid_latin_hypercube(
  trees(),
  tree_depth(),
  min_n(),
  loss_reduction(),
  sample_size = sample_prop(),
  finalize(mtry(), player_train),
  learn_rate()
)

player_folds <- vfold_cv(player_train)

xgb_res <- tune_grid(
  xg_workflow,
  resamples = player_folds,
  grid = xgb_grid,
  control = control_grid(save_pred = TRUE)
)

best_rmse <- select_best(xgb_res, "rmse")

final_xgb <- finalize_workflow(
  xg_workflow,
  best_rmse
)

```

Because there's not a ton of data here, this goes relatively quickly. Now to create fits.

```

linear_fit <-
  linear_workflow %>%
  fit(data = player_train)

rf_fit <-
  rf_workflow %>%
  fit(data = player_train)

xg_fit <-
  final_xgb %>%
  fit(data = player_train)

```

And now to make predictions.


```

linearpredict <-
  linear_fit %>%
  predict(new_data = player_train) %>%
  bind_cols(player_train)

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

xgpredict <-
  xg_fit %>%
  predict(new_data = player_train) %>%
  bind_cols(player_train)

```

For your assignment: Interpret the metrics output of each. Compare them. How does each model do relative to each other?

```
metrics(linearpredict, FantPt, .pred)
```

```

# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>         <dbl>
1 rmse    standard        43.6
2 rsq     standard         0.279
3 mae     standard        34.2

```

```
metrics(rfpredict, FantPt, .pred)
```

```

# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>         <dbl>
1 rmse    standard        24.6
2 rsq     standard         0.839
3 mae     standard        18.5

```

```
metrics(xgpredict, FantPt, .pred)
```

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

	.metric	.estimator	.estimate
	<chr>	<chr>	<dbl>
1	rmse	standard	38.1
2	rsq	standard	0.455
3	mae	standard	28.6

For your assignment: Implement metrics for test. What happens?

12 LightGBM for regression

12.1 The basics

Reminder from before: 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.

Did you catch that bold part? LightGBM is prone to overfitting on small datasets. *Our dataset is small.*

LightGBM also uses histograms of the data to make choices, where XGBoost is computationally 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.

What changes from before to now? Very little. Just the output – we're predicting a number this time, not a category.

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

```
library(tidyverse)
library(tidymodels)
library(bonsai)

set.seed(1234)
```

We'll use the same data – wide receivers with college stats, draft information and fantasy points.

```
wrraftedstats <- read_csv("https://mattwaite.github.io/sportsdatafiles/wrraftedstats2013")
```

We thin up the inputs.

```

wrselected <- wrdraftedstats %>%
  select(
    name,
    year,
    college_team,
    nfl_team,
    overall,
    total_yards,
    total_touchdowns,
    FantPt
  ) %>% na.omit()

```

And split our data.

```

player_split <- initial_split(wrselected, prop = .8)

player_train <- training(player_split)
player_test <- testing(player_split)

```

Now a recipe.

```

player_recipe <-
  recipe(FantPt ~ ., data = player_train) %>%
  update_role(name, year, college_team, nfl_team, new_role = "ID")

summary(player_recipe)

```

```

# A tibble: 8 x 4
  variable      type    role    source
  <chr>         <chr>  <chr>  <chr>
1 name         nominal ID      original
2 year         numeric ID      original
3 college_team nominal ID      original
4 nfl_team     nominal ID      original
5 overall      numeric predictor original
6 total_yards  numeric predictor original
7 total_touchdowns numeric predictor original
8 FantPt       numeric outcome  original

```

12.2 Implementing LightGBM

We're going to implement XGBoost and LightGBM side by side. That will give us the chance to compare.

We start with model definition.

```
xg_mod <- boost_tree(  
  trees = tune(),  
  learn_rate = tune(),  
  tree_depth = tune(),  
  min_n = tune(),  
  loss_reduction = tune(),  
  sample_size = tune(),  
  mtry = tune(),  
) %>%  
set_mode("regression") %>%  
set_engine("xgboost")  
  
lightgbm_mod <-  
  boost_tree() %>%  
  set_engine("lightgbm") %>%  
  set_mode(mode = "regression")
```

Now we create workflows.

```
xg_workflow <-  
  workflow() %>%  
  add_model(xg_mod) %>%  
  add_recipe(player_recipe)  
  
lightgbm_workflow <-  
  workflow() %>%  
  add_model(lightgbm_mod) %>%  
  add_recipe(player_recipe)
```

We'll tune the XGBoost model.

```
xgb_grid <- grid_latin_hypercube(  
  trees(),  
  tree_depth(),  
  min_n(),
```

```

    loss_reduction(),
    sample_size = sample_prop(),
    finalize(mtry(), player_train),
    learn_rate()
)

player_folds <- vfold_cv(player_train)

xgb_res <- tune_grid(
  xg_workflow,
  resamples = player_folds,
  grid = xgb_grid,
  control = control_grid(save_pred = TRUE)
)

best_rmse <- select_best(xgb_res, "rmse")

final_xgb <- finalize_workflow(
  xg_workflow,
  best_rmse
)

```

Now we make fits.

```

xg_fit <-
  final_xgb %>%
  fit(data = player_train)

lightgbm_fit <-
  lightgbm_workflow %>%
  fit(data = player_train)

```

With the fits in hand, we can bind the predictions to the data.

```

xgpredict <-
  xg_fit %>%
  predict(new_data = player_train) %>%
  bind_cols(player_train)

lightgbmpredict <-
  lightgbm_fit %>%
  predict(new_data = player_train) %>%

```

```
bind_cols(player_train)
```

For your assignment: How do these two compare?

```
metrics(xgpredict, FantPt, .pred)
```

```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>         <dbl>
1 rmse    standard        38.3
2 rsq     standard         0.449
3 mae     standard        29.2
```

```
metrics(lightgbmpredict, FantPt, .pred)
```

```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>         <dbl>
1 rmse    standard        31.8
2 rsq     standard         0.638
3 mae     standard        24.4
```

For your assignment: How do these models fare in testing?