Final Project

Cyril Wang

Contents


```
# set up stuff
set.seed(123) # if needed
library(tidyverse)
library(tidymodels)
library(dplyr)
library(discrim)
library(glmnet)
library(vembedr)
library(janitor)
library(kknn)
library(kernlab)
library(yardstick)
library(vip)
```

Introduction

In this project, we will be working with data from the NBA, or National Basketball Association, in order to create a model that aims to predict the a player's number of win shares, or an estimate of the number of wins that a player contributes to their team, based on their statistics and shooting tendencies.

The NBA

The National Basketball Association, or NBA for short, is a professional basketball league consisting of 30 teams located in North America. Founded in 1946, the NBA has continued to expand over the years and has become one of the most popular sports leagues in the world, with millions of people tuning in to watch the sport every day. Here is a video that showcases the appeal of the NBA:

```
embed_url("https://www.youtube.com/watch?v=SWYPm24qVd8")
```

To quickly summarize how the sport works, two teams face off at a time, with each time sending out 5 players each so that the game is 5 vs 5. The goal of the game is to score more points than the opposing team. Players alternate playing offense and defense, where if a team makes a shot, the other team then gets the opportunity to play offense, which gives rise to a rapid-pace game that fans love. Players can score in a variety of ways, either taking shots (shooting the basketball) close to the basket/goal to score 2 points, or to take further shots to score 3 points.

Here is a picture to help visualize:

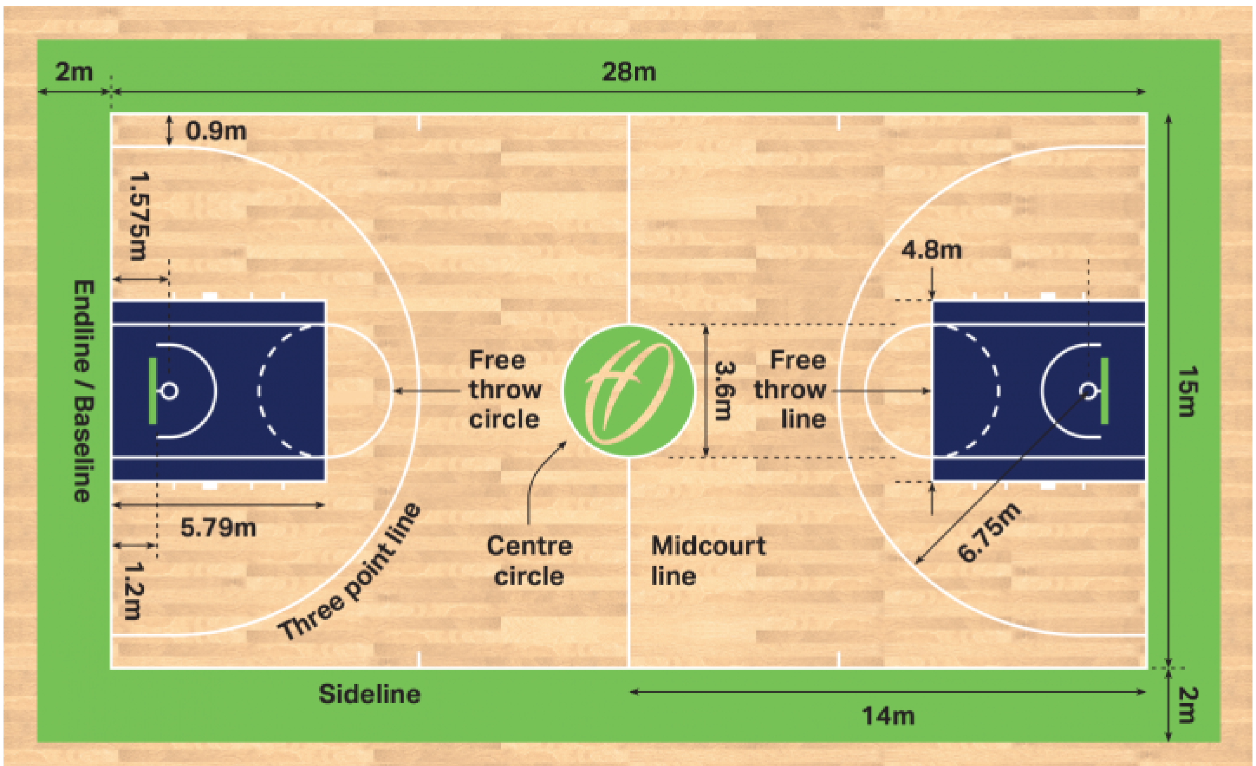


Figure 1: NBA court diagram

There are other ways to impact the game, such as assists (a pass that helps a teammate score), rebounds (grabbing and taking possession of the ball after a shot attempt), steals (taking the ball away from a player on offense), turnovers (losing possessions of the ball while on offense), and many more.

In recent years, there has been an increasing trend of the use of analytics in the game of basketball, with various statisticians creating different statistics (known as advanced stats) in order to measure a player's impact on the game, in other words, trying to classify how good a player is. In the recent Most Valuable Player race, which is an award that is awarded to the best player in the NBA, the player that won led the league in most of these advanced stats, highlighting their usage in the current era of the NBA.

In particular, I chose to predict a player's number of win shares/48, which attempts to measure how many wins that a player contributes for their team in 48 minutes, in order to standardize the stat for every player since players that play more will naturally contribute more towards winning. So why should we care? In what is some basketball fans' highlight of the year, the NBA draft is where teams select new players from college or international leagues for their teams. The draft is notorious for being difficult to predict which players will be good, so this model hopes to try to predict player's impact on winning based on their stats and in-game patterns. In addition, I hope to highlight which predictors/statistics contribute most towards winning.

Data

The data that I used was taken from here, which collects NBA-relevant data such as team standings, league averages, and player stats (per game and total). Specifically, I used multiple datasets from there, namely combining the per-game statistics of players from the 2020-2021 season and the advanced stats of players from the 2020-2021 season. The advanced stats dataset contains important variables like win shares. To specify, the per-game data set contains all the player's per-game data for statistics that the NBA officially tracks, such as points, assists, rebounds, etc… Meanwhile, the advanced stats data set contains statistics that are calculated, rather than being official recorded. For example, one such variable is the 3-point rate, the calculated percentage of shots that a player takes that are 3-pointers. To see all the variables and their descriptions, please refer to the codebook included.

Loading and cleaning the data

First, I loaded the data using the csv's downloaded from the above sites.

```
nba <- read.csv(file = 'data/nba2021.csv')
nba_advanced <- read.csv(file = 'data/nba2021advanced.csv')
```

Then for the per-game stats dataset, I cleaned up the names manually to remove extraneous x's from the variable names and to add on the percent sign to some, which was lost in the reading in of data.

```
# remove x's from variable name
nba = nba %>% rename("Player" = 'Player.') %>% rename("3P" = 'X3P') %>% rename("3PA" = 'X3PA') %>% renam

# rename variables and drop blank columns
nba_advanced = nba_advanced  %>% rename("Player" = 'Player.') %>% rename("3PAr" = 'X3PAr') %>% rename("
```

Next, I dropped variables that were irrelevant to the task or redundant variables, such as the team that a player played on, number of games played, games started, etc… For the advanced stats data set, I removed much of the other advanced stats, keeping only win shares per 48 minutes and statistics that described a player's behavior while playing (such as 3-point rate)

```
# drop irrelevant variables
nba = nba %>% select(-Rk, -Tm, -G, -GS, -MP, -eFG.)
nba_advanced = nba_advanced %>% select(Player, Pos, Age, `3PAr`, FTr, `USG%`,WS_48)
head(nba)
```

```
##                                Player Pos Age  FG  FGA   FG%  3P 3PA   3P%  2P
## 1        Precious Achiuwa\\achiupr01  PF  21 2.0  3.7 0.544 0.0 0.0 0.000 2.0
## 2            Jaylen Adams\\adamsja01  PG  24 0.1  1.1 0.125 0.0 0.3 0.000 0.1
## 3            Steven Adams\\adamsst01   C  27 3.3  5.3 0.614 0.0 0.1 0.000 3.3
```

```
## 4              Bam Adebayo\\adebaba01   C  23 7.1 12.5 0.570 0.0 0.1 0.250 7.1
## 5        LaMarcus Aldridge\\aldrila01   C  35 5.4 11.4 0.473 1.2 3.1 0.388 4.2
## 6 Nickeil Alexander-Walker\\alexani01  SG  22 4.2 10.0 0.419 1.7 4.8 0.347 2.5
##    2PA   2P%  FT FTA   FT% ORB DRB TRB AST STL BLK TOV  PF  PTS
## 1  3.7 0.546 0.9 1.8 0.509 1.2 2.2 3.4 0.5 0.3 0.5 0.7 1.5  5.0
## 2  0.9 0.167 0.0 0.0    NA 0.0 0.4 0.4 0.3 0.0 0.0 0.0 0.1  0.3
## 3  5.3 0.620 1.0 2.3 0.444 3.7 5.2 8.9 1.9 0.9 0.7 1.3 1.9  7.6
## 4 12.4 0.573 4.4 5.5 0.799 2.2 6.7 9.0 5.4 1.2 1.0 2.6 2.3 18.7
## 5  8.3 0.505 1.6 1.8 0.872 0.7 3.8 4.5 1.9 0.4 1.1 1.0 1.8 13.5
## 6  5.2 0.485 1.0 1.4 0.727 0.3 2.8 3.1 2.2 1.0 0.5 1.5 1.9 11.0
```

```
head(nba_advanced)
```

```
##                                Player Pos Age  3PAr   FTr USG%  WS_48
## 1         Precious Achiuwa\\achiupr01  PF  21 0.004 0.482 19.5  0.085
## 2             Jaylen Adams\\adamsja01  PG  24 0.250 0.000 18.6 -0.252
## 3             Steven Adams\\adamsst01   C  27 0.010 0.438 11.7  0.119
## 4              Bam Adebayo\\adebaba01   C  23 0.010 0.443 23.7  0.197
## 5        LaMarcus Aldridge\\aldrila01   C  35 0.270 0.159 22.2  0.080
## 6 Nickeil Alexander-Walker\\alexani01  SG  22 0.478 0.144 23.2  0.035
```

Finally, I combined the data sets together, merging them based on their overlapping variables.

```
# combine datasets
nba_combined <- merge(nba, nba_advanced, by=c("Player", "Pos", "Age"))
head(nba_combined)
```

```
##                      Player Pos Age  FG  FGA   FG%  3P 3PA   3P%  2P 2PA   2P%
## 1    Aaron Gordon\\gordoaa01  PF  25 4.6 10.0 0.463 1.2 3.5 0.335 3.4 6.5 0.533
## 2   Aaron Holiday\\holidaa01  PG  24 2.6  6.6 0.390 1.0 2.8 0.368 1.6 3.8 0.406
## 3   Aaron Nesmith\\nesmiaa01  SF  21 1.7  3.9 0.438 0.9 2.3 0.370 0.8 1.5 0.543
## 4     Abdel Nader\\naderab01  SF  27 2.4  4.8 0.491 0.8 1.8 0.419 1.6 3.0 0.534
## 5     Adam Mokoka\\mokokad01  SG  22 0.5  1.4 0.368 0.1 0.7 0.100 0.4 0.6 0.667
## 6 Al-Farouq Aminu\\aminual01  PF  30 1.7  4.3 0.384 0.3 1.6 0.216 1.3 2.7 0.484
##    FT FTA   FT% ORB DRB TRB AST STL BLK TOV  PF  PTS  3PAr   FTr USG%  WS_48
## 1 1.9 3.0 0.651 1.5 4.1 5.7 3.2 0.7 0.7 1.9 1.8 12.4 0.353 0.299 20.7  0.066
## 2 1.0 1.3 0.819 0.2 1.1 1.3 1.9 0.7 0.2 1.0 1.4  7.2 0.417 0.190 19.5  0.009
## 3 0.5 0.6 0.786 0.6 2.2 2.8 0.5 0.3 0.2 0.5 1.9  4.7 0.607 0.157 13.7  0.076
## 4 1.2 1.5 0.757 0.3 2.3 2.6 0.8 0.4 0.4 0.8 1.4  6.7 0.371 0.319 19.0  0.095
## 5 0.0 0.1 0.000 0.1 0.3 0.4 0.4 0.1 0.1 0.4 0.4  1.1 0.526 0.053 18.9 -0.112
## 6 0.8 1.0 0.818 1.0 3.8 4.8 1.3 0.8 0.4 1.2 1.3  4.4 0.374 0.222 13.6  0.010
```

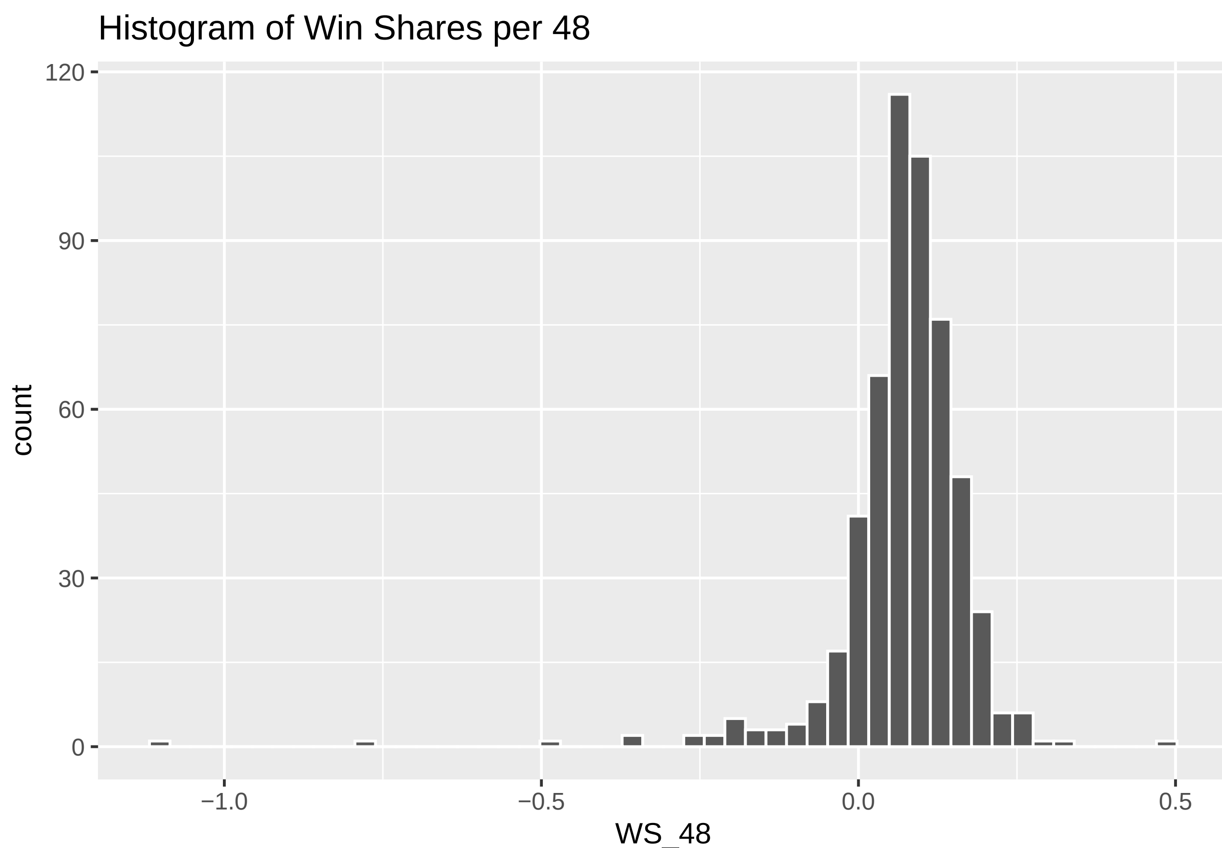
Our combined data set has 540 observations and 28 variables.

Finally, with all that done, we can now begin our exploratory data analysis of our data set.

Exploratory Data Analysis

To start things off, let us examine the distribution of our response variable, WS per 48.

```r
# win shares distribution
ggplot(nba_combined, aes(x=WS_48)) +
  geom_histogram(bins= 50, color = "white") +
  labs(
    title = "Histogram of Win Shares per 48"
  )
```



We see that it appears to be centered around 0.1 (which makes sense since league average is 0.1). There does seem to be a few outliers, so let's take a look at the data.

```r
nba_advanced2 <- read.csv(file = 'data/nba2021advanced.csv')
head((nba_advanced2 %>% arrange(desc(WS.48))))
```

```
##    Rk                   Player. Pos Age  Tm  G   MP  PER   TS. X3PAr   FTr
## 1 212   Udonis Haslem\\hasleud01   C  40 MIA  1    3 54.6 1.000 0.000 0.000
## 2 399  Gary Payton II\\paytoga02  PG  28 GSW 10   40 29.2 0.847 0.308 0.308
## 3 263    Nikola Joki?\\jokicni01   C  25 DEN 72 2488 31.3 0.647 0.183 0.305
## 4 146     Joel Embiid\\embiijo01   C  26 PHI 51 1585 30.3 0.636 0.171 0.610
## 5 527 Robert Williams\\williro04   C  23 BOS 52  985 25.7 0.719 0.008 0.283
## 6 122  Dewayne Dedmon\\dedmode01   C  31 MIA 16  210 24.5 0.735 0.077 0.415
##   ORB. DRB. TRB. AST. STL. BLK. TOV. USG.  X  OWS DWS   WS WS.48 X.1 OBPM DBPM
## 1  0.0 37.5 19.1  0.0  0.0  0.0  0.0 30.1 NA  0.0 0.0  0.0 0.475  NA 24.1  7.0
## 2  5.4 23.6 14.6  4.1  7.0  2.2  6.3 16.8 NA  0.2 0.1  0.3 0.331  NA  1.0  8.2
## 3  9.4 26.1 17.8 40.4  1.9  1.9 13.1 29.6 NA 12.2 3.4 15.6 0.301  NA  9.1  3.0
## 4  8.0 29.1 18.7 16.2  1.5  3.9 12.2 35.3 NA  5.6 3.2  8.8 0.266  NA  6.3  1.2
```

```
## 5 14.9 25.6 20.2 14.2  2.1  8.6 15.2 15.0 NA  3.4 1.9  5.3 0.258  NA  2.9  3.1
## 6 15.5 31.0 23.4  9.6  2.1  3.0 14.5 19.3 NA  0.7 0.4  1.1 0.256  NA  1.2  0.5
##    BPM VORP
## 1 31.1  0.0
## 2  9.2  0.1
## 3 12.1  8.8
## 4  7.5  3.8
## 5  6.0  2.0
## 6  1.7  0.2
```

```
head((nba_advanced2 %>% arrange((WS.48))))
```

```
##    Rk                     Player. Pos Age  Tm G MP   PER  TS. X3PAr FTr ORB.
## 1 393 Anžejs Pase??iks\\pasecan01   C  25 WAS 1  6 -40.6 0.00 1.000   0 17.7
## 2 307      Will Magnay\\magnawi01   C  22 NOP 1  3 -35.1 0.00 1.000   0  0.0
## 3 503      Noah Vonleh\\vonleno01  PF  25 BRK 4 11 -19.0 0.00 0.667   0  0.0
## 4 203     Jared Harper\\harpeja01  PG  23 NYK 8 16 -10.8 0.26 0.250   1  0.0
## 5 195    Ashton Hagans\\haganas01  PG  21 MIN 2  4 -12.4   NA    NA  NA  0.0
## 6 521 Greg Whittington\\whittgr01  PF  27 DEN 4 12 -10.2 0.00 0.667   0  0.0
##   DRB. TRB. AST. STL. BLK.  TOV. USG.  X  OWS DWS   WS  WS.48 X.1  OBPM DBPM
## 1  0.0  8.8 18.7    0    0  83.3 41.4 NA -0.1   0 -0.1 -1.113  NA -40.7 -5.9
## 2  0.0  0.0  0.0    0    0  50.0 28.0 NA  0.0   0  0.0 -0.787  NA -30.7 -8.6
## 3  9.6  5.0 10.2    0    0  40.0 19.8 NA -0.1   0 -0.1 -0.488  NA -20.9 -5.8
## 4 13.4  6.8  7.7    0    0  34.2 24.4 NA -0.1   0 -0.1 -0.365  NA -16.6 -5.5
## 5  0.0  0.0  0.0    0    0 100.0 10.5 NA  0.0   0  0.0 -0.353  NA -13.7 -7.4
## 6  0.0  0.0  0.0    0    0   0.0 10.9 NA -0.1   0 -0.1 -0.259  NA -11.4 -5.7
##     BPM VORP
## 1 -46.6 -0.1
## 2 -39.3  0.0
## 3 -26.7 -0.1
## 4 -22.1 -0.1
## 5 -21.1  0.0
## 6 -17.2  0.0
```

We see that the outliers were due to a extremely low sample size, with most of them playing a handful of games, so let's filter out the players who have played less than 10 games to get a better view of things.

```
nba <- read.csv(file = 'data/nba2021.csv') %>% filter(G >= 10)
nba_advanced <- read.csv(file = 'data/nba2021advanced.csv') %>% filter(G >= 10)

# remove x's from variable name
nba = nba %>% rename("Player" = 'Player.') %>% rename("3P" = 'X3P') %>% rename("3PA" = 'X3PA') %>% renam

# rename variables and drop blank columns
nba_advanced = nba_advanced  %>% rename("Player" = 'Player.') %>% rename("3PAr" = 'X3PAr') %>% rename("

# drop irrelevant variables
nba = nba %>% select(-Rk, -Tm, -G, -GS, -MP, -eFG.)
nba_advanced = nba_advanced %>% select(Player, Pos, Age, `3PAr`, FTr, `USG%`,WS_48)

# combine datasets
nba_combined <- merge(nba, nba_advanced, by=c("Player", "Pos", "Age"))
# head(nba_combined) # new data set has 490 observations
```

Missing values

While we are on the issue of filtering out players, let's examine the issue of missing values within our data set. Let's check which variables, if any, contain missing values.

```
colSums(is.na(nba_combined))
```

```
## Player    Pos    Age     FG    FGA    FG%     3P    3PA    3P%     2P    2PA
##      0      0      0      0      0      0      0      0     13      0      0
##    2P%     FT    FTA    FT%    ORB    DRB    TRB    AST    STL    BLK    TOV
##      0      0      0      6      0      0      0      0      0      0      0
##     PF    PTS   3PAr    FTr   USG%  WS_48
##      0      0      0      0      0      0
```

We see that two of our predictors, 3P% and FT%, have missing values. Examining deeper,

```
subset(nba_combined, is.na(nba_combined$`3P%`))
```

```
##                              Player  Pos Age  FG  FGA   FG% 3P 3PA 3P%  2P  2PA
## 71          Clint Capela\\capelca01    C  26 6.6 11.0 0.594  0   0  NA 6.6 11.0
## 87        Daniel Gafford\\gaffoda01 PF-C  22 2.9  4.2 0.684  0   0  NA 2.9  4.2
## 123       Devontae Cacok\\cacokde01   PF  24 0.9  1.5 0.586  0   0  NA 0.9  1.5
## 129            Donta Hall\\halldo01   PF  23 1.9  2.7 0.714  0   0  NA 1.9  2.7
## 141             Ed Davis\\davised01    C  31 0.8  1.9 0.432  0   0  NA 0.8  1.9
## 156    Freddie Gillespie\\gillefr01   PF  23 2.2  4.2 0.524  0   0  NA 2.2  4.2
## 200         Jakob Poeltl\\poeltja01    C  25 3.8  6.2 0.616  0   0  NA 3.8  6.2
## 287 Kostas Antetokounmpo\\antetko01   PF  23 0.2  0.7 0.300  0   0  NA 0.2  0.7
## 339    Mitchell Robinson\\robinmi01    C  22 3.7  5.7 0.653  0   0  NA 3.7  5.7
## 345          Moses Brown\\brownmo01    C  21 3.4  6.2 0.545  0   0  NA 3.4  6.2
## 364         Norvel Pelle\\pelleno01    C  27 0.6  1.2 0.533  0   0  NA 0.6  1.2
## 432            Tacko Fall\\fallta01    C  25 1.1  1.5 0.724  0   0  NA 1.1  1.5
## 472       Udoka Azubuike\\azubuud01    C  21 0.3  0.6 0.444  0   0  NA 0.3  0.6
##       2P%  FT FTA   FT% ORB DRB  TRB AST STL BLK TOV  PF  PTS 3PAr   FTr USG%
## 71  0.594 2.1 3.6 0.573 4.7 9.6 14.3 0.8 0.7 2.0 1.2 2.3 15.2    0 0.327 19.9
## 87  0.684 1.3 2.0 0.667 1.7 2.5  4.3 0.5 0.5 1.4 0.8 1.8  7.0    0 0.480 16.7
## 123 0.586 0.3 0.6 0.455 0.6 1.0  1.6 0.1 0.3 0.2 0.3 0.4  2.0    0 0.379 17.2
## 129 0.714 1.8 2.6 0.676 1.8 2.9  4.8 0.8 0.4 0.8 0.6 1.4  5.6    0 0.971 14.0
## 141 0.432 0.4 0.5 0.833 2.0 3.0  5.0 0.9 0.6 0.6 0.3 2.4  2.1    0 0.273  7.9
## 156 0.524 1.2 1.7 0.697 2.1 2.8  4.9 0.5 0.7 1.0 0.6 2.2  5.6    0 0.393 12.2
## 200 0.616 0.9 1.8 0.508 3.2 4.8  7.9 1.9 0.7 1.8 1.2 2.5  8.6    0 0.288 13.4
## 287 0.300 0.4 0.9 0.462 0.3 1.0  1.3 0.1 0.1 0.3 0.7 0.5  0.8    0 1.300 20.7
## 339 0.653 0.8 1.7 0.491 3.6 4.5  8.1 0.5 1.1 1.5 0.8 2.8  8.3    0 0.301 11.8
## 345 0.545 1.8 2.9 0.619 3.6 5.3  8.9 0.2 0.7 1.1 1.0 2.2  8.6    0 0.470 16.9
## 364 0.533 0.3 0.5 0.667 0.5 1.0  1.5 0.2 0.1 0.7 0.2 1.1  1.5    0 0.400 10.8
## 432 0.724 0.3 0.8 0.333 0.8 1.9  2.7 0.2 0.1 1.1 0.3 1.2  2.5    0 0.517 13.2
## 472 0.444 0.5 0.7 0.800 0.3 0.6  0.9 0.0 0.1 0.3 0.2 0.6  1.1    0 1.111 12.4
##      WS_48
## 71   0.207
## 87   0.209
## 123  0.142
## 129  0.223
## 141  0.135
## 156  0.116
```

```
## 200  0.148
## 287 -0.174
## 339  0.192
## 345  0.138
## 364  0.112
## 432  0.177
## 472  0.119
```

```
subset(nba_combined, is.na(nba_combined$`FT%`))
```

```
##                             Player Pos Age  FG FGA   FG%  3P 3PA   3P%  2P 2PA
## 122      Devon Dotson\\dotsode01  PG  21 1.0 1.9 0.524 0.1 0.6 0.143 0.9 1.3
## 211      Jared Dudley\\dudleja01  PF  35 0.2 0.8 0.222 0.2 0.5 0.333 0.0 0.3
## 239        Jordan Bone\\bonejo01  PG  23 1.6 3.9 0.426 0.7 2.3 0.313 0.9 1.6
## 267    Keljin Blevins\\blevike01  SF  25 0.3 1.2 0.250 0.1 0.5 0.250 0.2 0.7
## 438 Terrance Ferguson\\fergute01  SG  22 0.1 0.5 0.143 0.0 0.4 0.000 0.1 0.2
## 444        Theo Pinson\\pinsoth01  SG  25 0.1 0.5 0.111 0.0 0.5 0.000 0.1 0.1
##       2P% FT FTA FT% ORB DRB TRB AST STL BLK TOV  PF PTS  3PAr FTr USG%  WS_48
## 122 0.714  0   0  NA 0.2 0.3 0.5 0.6 0.4 0.0 0.0 0.3 2.1 0.333   0 18.2  0.177
## 211 0.000  0   0  NA 0.3 1.4 1.8 0.4 0.1 0.1 0.2 0.6 0.5 0.667   0  5.9  0.053
## 239 0.591  0   0  NA 0.3 1.4 1.7 1.3 0.1 0.0 0.2 0.8 4.0 0.593   0 12.6  0.045
## 267 0.250  0   0  NA 0.2 0.4 0.6 0.2 0.1 0.0 0.3 0.5 0.7 0.400   0 14.3 -0.137
## 438 0.500  0   0  NA 0.0 0.1 0.1 0.2 0.1 0.0 0.3 0.5 0.2 0.714   0  9.7 -0.161
## 444 1.000  0   0  NA 0.0 0.3 0.3 0.1 0.0 0.0 0.1 0.2 0.1 0.889   0 13.1 -0.205
```

it seems that all the missing values come from players who have not attempted any 3s or free throws, rendering their 3P and FT percentages to NA. Instead of removing these players completely, I have decided to change all their values to 0. Although we could have tried to resolve the issue by imputing values, I figured that predicting free throw and 3P percentages would bound to be inaccurate, and gave them 0's to simplify things.

```
nba_combined[is.na(nba_combined)] <- 0
```

Exploratory Data Analysis (continued)

Continuing on, let's try examining a few other variables.

Positions

In the NBA, there are 5 positions (point guard, shooting guard, small forward, power forward, center), where each position's duty varies. For example, point guards are tasked with controlling and passing the basketball more while centers, who are taller, are tasked with rebounding among others. So I am curious to see if win shares differs based on a player's position.

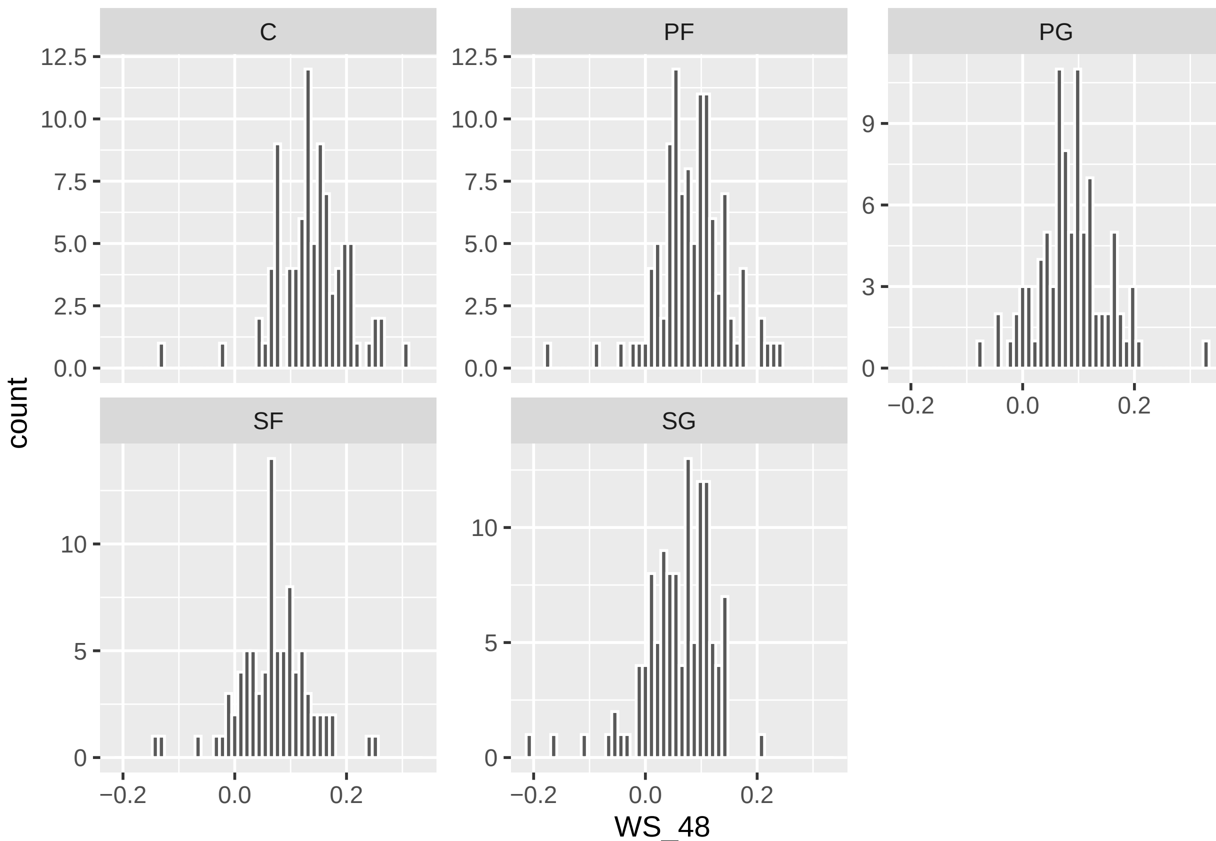
We first need to modify the data set a bit, since there are some players that have multiple positions, such as PG-SG. I decided to remove that, assigning the second position as the position of the player (so in this case, the player would be classified as a SG).

```
# first edit positions so there are no multi-postions (i.e. PG-SG)
for (i in 1:length(nba_combined$Pos)){
  if (grepl('-', nba_combined$Pos[i], fixed=TRUE)) {
```

```
    pos = which(strsplit(nba_combined$Pos[i], "")[[1]]=="-")
    nba_combined$Pos[i] = substr(nba_combined$Pos[i], pos + 1, nchar(nba_combined$Pos[i]))
  }
}
```

Then, we can plot the distribution of WS per 48 grouped by position.

```
ggplot(nba_combined, aes(x=WS_48)) +
  geom_histogram(bins= 50, color = "white") +
  facet_wrap(~Pos, scales="free_y")
```

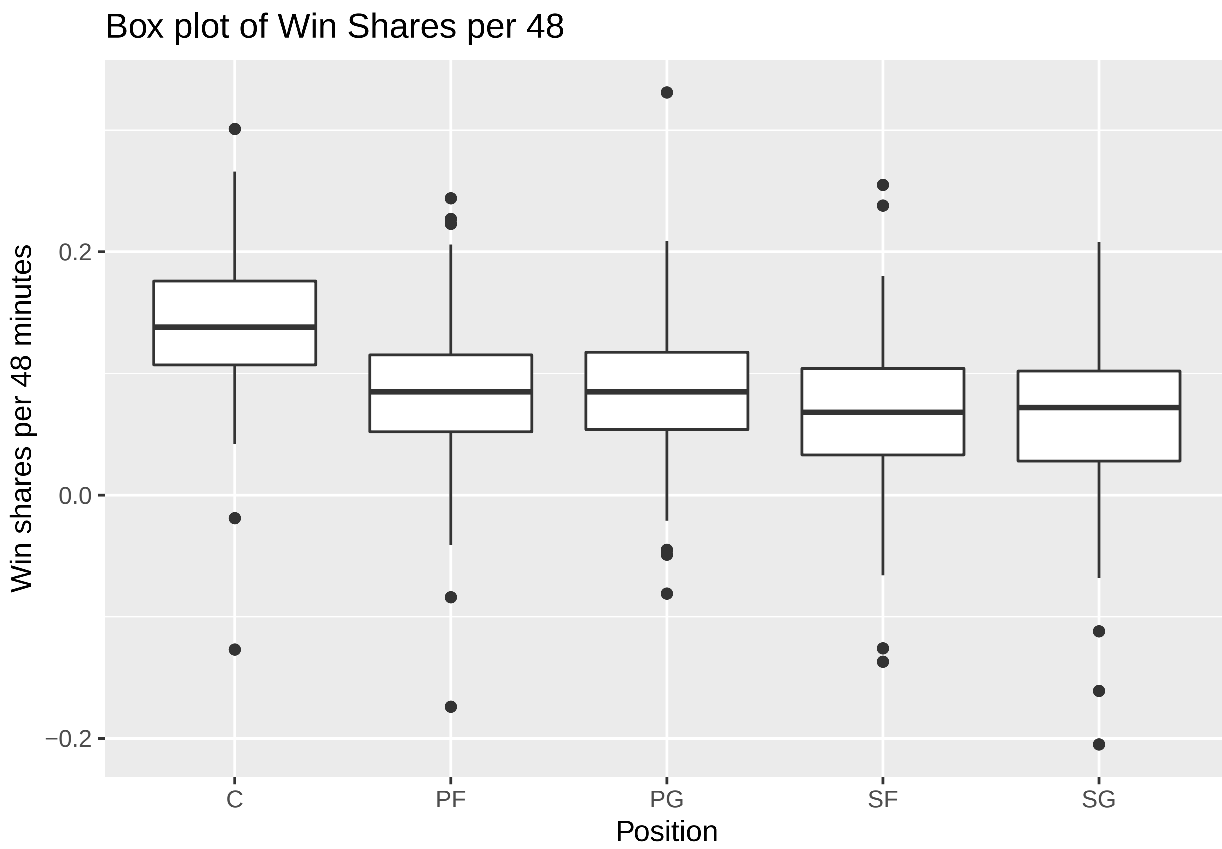



```
  labs(
    title = "Histogram of Win Shares per 48"
  )
```

```
## $title
## [1] "Histogram of Win Shares per 48"
##
## attr(,"class")
## [1] "labels"
```

Looking at the data, it actually seems that centers have the highest win share per 48 minutes, but it is a little difficult to tell. Let us try a different type of graph.

```
ggplot(nba_combined, aes(x=Pos, y=WS_48)) +
  geom_boxplot() +
  labs(
    title = "Box plot of Win Shares per 48",
    x= "Position",
    y = "Win shares per 48 minutes"
  )
```



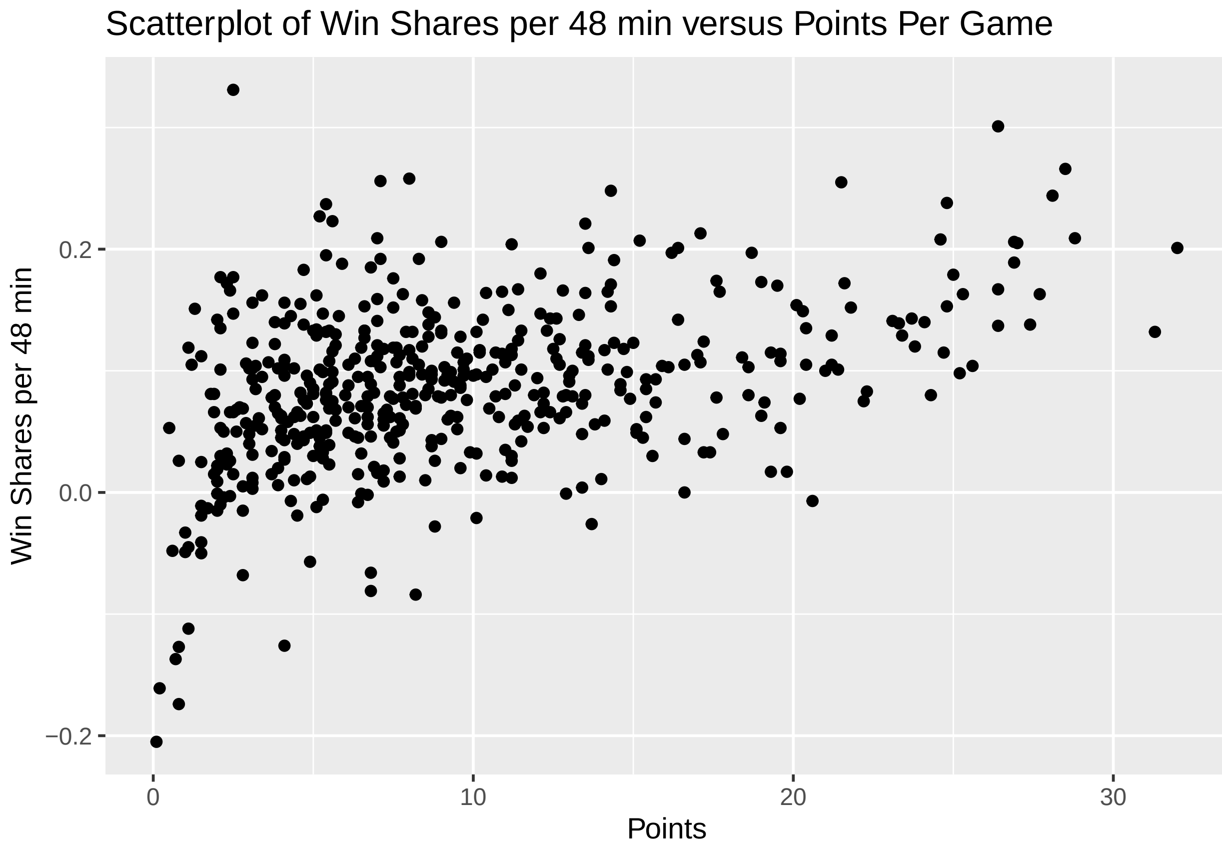
As we can see, our boxplot does show that centers, on average, do have higher win share per 48 min compared to the other positions, with point guards and power forwards having the next highest averages. I am a little suprised that small forward has the lowest win shares per 48 min, just based on my experience with the NBA, as the small forward position is typically viewed as the most important for a team to win a championship, and is the position of some of the best and most famous players such as LeBron James, Kevin Durant, etc…

Points, Age, and Usage

Next, I am interested in the relationship between win shares per 48 minutes and the points a player averages per game, their age, and their usage rate (or how involved they were on the court).

First up is points per game, which I suspect would be positively correlated, since scoring more points is the goal of the game.

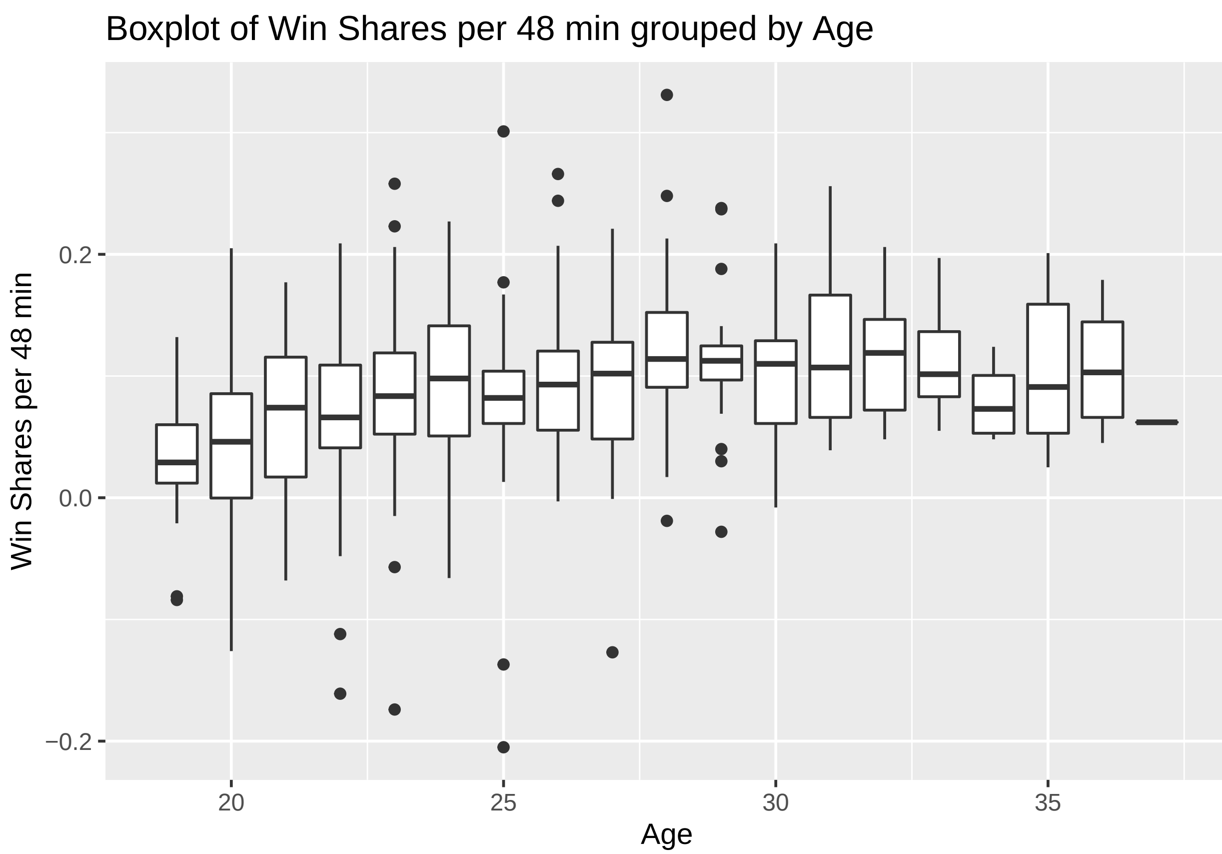
```
ggplot(nba_combined, aes(x=PTS, y=WS_48)) + geom_point() + labs(
  title = "Scatterplot of Win Shares per 48 min versus Points Per Game",
  x = "Points", y=  "Win Shares per 48 min"
)
```



It is a little difficult to make out a pattern, but it does seem that there is a slight exponential relationship between the two. In general, it does seem that scoring more does have an impact on win shares, although the difference between scoring 10 points and 30 points is not that large, indicting that there are other aspects of basketball that impact winning just as much.

Here is age, which is a tricky one to assess. Naturally, we expect younger ages to be worse players due to their experience and thus have lower win shares, but also for older ages to no longer be athletic enough to keep up.

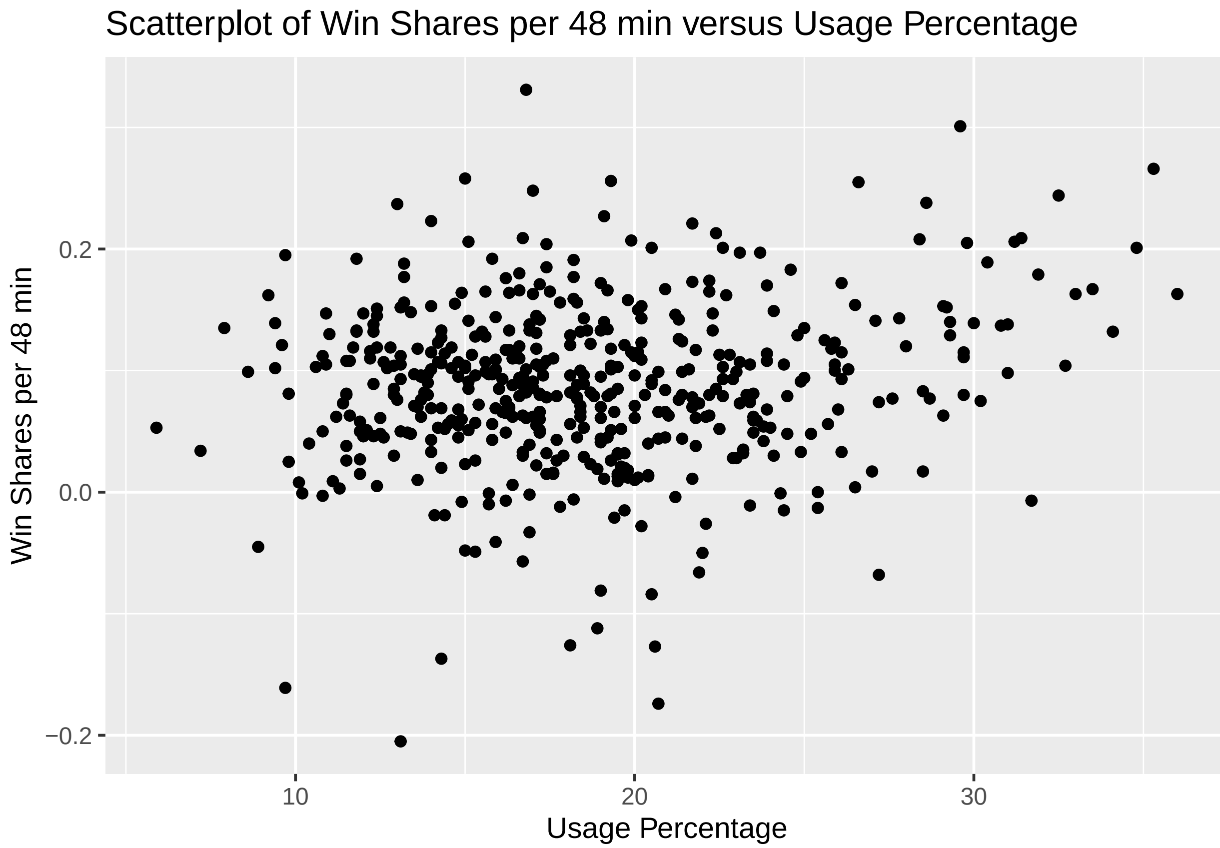
```
ggplot(nba_combined, aes(x=Age, y=WS_48, group = Age)) + geom_boxplot() + labs(
  title = "Boxplot of Win Shares per 48 min grouped by Age",
  x = "Age", y=  "Win Shares per 48 min"
)
```



The results seem to mirror our hypothesis, as the average win shares per 48 increases as age increaes, up until around age 30, where it then does not seem to follow a specific pattern.

Finally, let us examine how usage rate impacts win shares and winning. To be specific, usage rate refers to the percentage of team plays that is used by a player while on the floor, or whether they were tasked with being the main contributor of the offense. In general, this is also an indicament of how much the player controls the ball while on offense.

```
ggplot(nba_combined, aes(x=`USG%`, y=WS_48)) + geom_point() + labs(
  title = "Scatterplot of Win Shares per 48 min versus Usage Percentage",
  x = "Usage Percentage", y=  "Win Shares per 48 min"
)
```

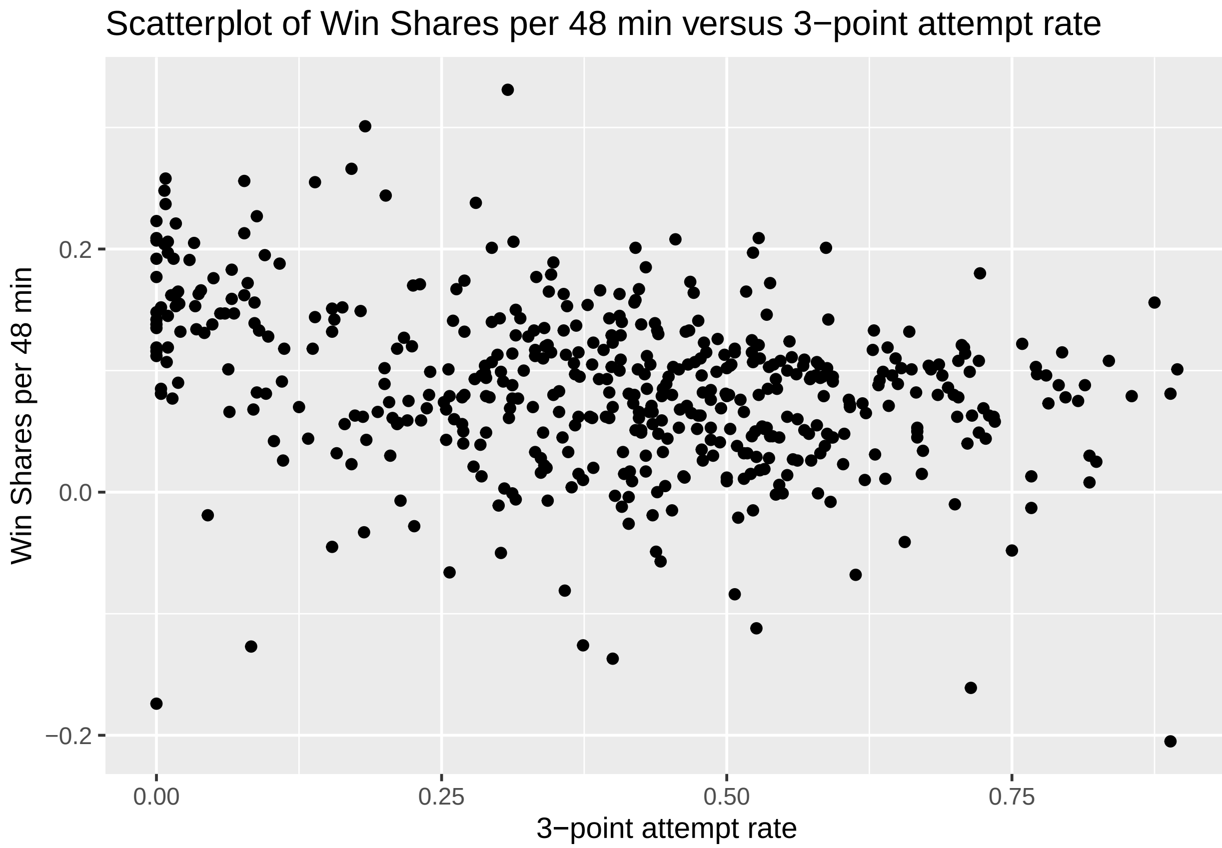


Surprisingly, there is not a discernable pattern in the scatterplot, with maybe a slight positive trend, but I had expected that players that were more involved would have a higher amount of win shares. This suggests that there are other aspects of basketball that a player can contribute that is not directly involved with the ball.

3-point rate

One final relationship that I want to explore is how a player's shot tendencies affect winning. As a bit of background, there are 3 ways to score: free throws (1 point), 2-pointers, and 3-pointers. I am focusing on 3-pointers, which is a basket that is scored by shooting the basketball and making it from beyond the three-point line, a arc that is around 23 feet from the basket. In recent years, players have begun to take more and more threes spurred on by the influence of math and statistics, with the motto that "3>2" suggesting that the three-point shot is a better shot. To be clearer, the idea is that if a player shoots above a certain percentage on three-pointers, it turns into a better and more efficient shot than taking a regular, 2-point shot within the arc. So, I wanted to examine if this is actually the case and if players that shoot a lot of three-pointers earn more win shares. The variable that I am plotting is the 3-point attempt rate, or the percentage of a player's shots that are 3-pointers.

```
ggplot(nba_combined, aes(x=`3PAr`, y=WS_48)) + geom_point() + labs(
  title = "Scatterplot of Win Shares per 48 min versus 3-point attempt rate",
  x = "3-point attempt rate", y=  "Win Shares per 48 min"
)
```

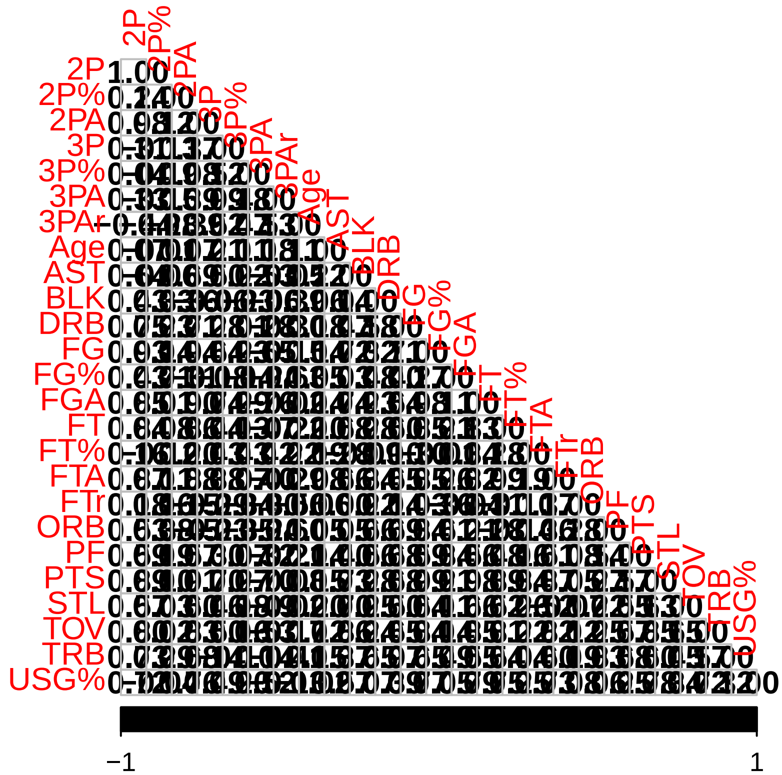


Once again, it doesn't seem that taking more 3-points is correlated with more win shares. This makes sense since players that shoot 3-pointers for the majority of their shots are often specialists and are often lacking in other areas of the game like defense. In addition, we saw that centers, who typically do not take very many 3-pointers, had a higher average win shares per 48 minutes than the other positions.

Correlation

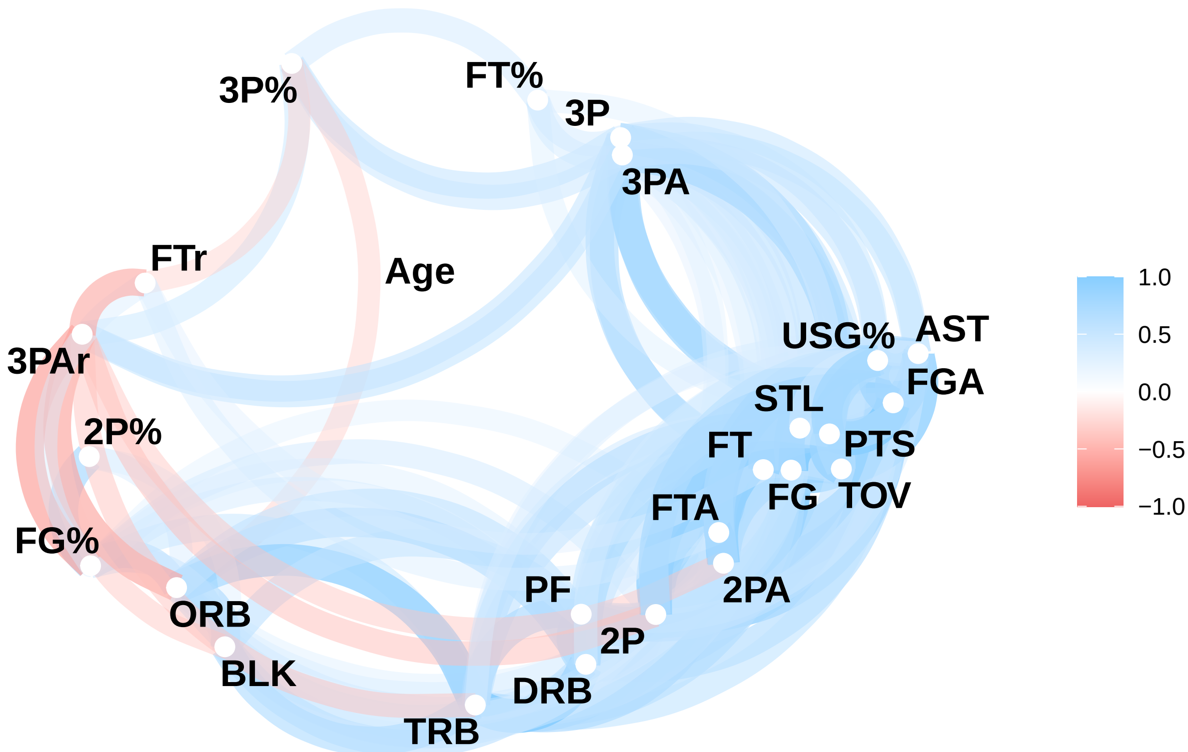
Lastly, I wanted to create a correlation matrix of all the numeric variables in order to see if some are highly correlated.

```
library(corrplot)
cor(select(nba_combined, where(is.numeric), -c(WS_48))) %>%
corrplot(method='number', order='alphabet', type= 'lower', col='black')
```



We see that we have a lot of predictors variables that it makes it hard to read, so let's try something else. Instead, I used the corrr package to use its network_plot() function, which helps visualize the correlation between predictors in a cooler way without all the numbers.

```
library(corrr)
select(nba_combined, where(is.numeric), -c(WS_48)) %>% correlate() %>% network_plot()
```



From this, we see that a lot of our variables are actually fairly strongly correlated, such as the big clump of variables on the left. This suggests that we should try to mitigate this correlation between predictors if we can, also depending on the models that we choose to fit.

Model Fitting Prep

Before we begin building and fitting our models, we first need to . . .

Data Split

1. split our data into training and testing data sets using a proportion of 0.7, and to stratify sample the data on our response variable of `WS_48`. In addition, in preparation for our recipe later on, I decided to convert `Pos` into a factor variable in order to be able to dummy code it later on when creating the recipe.

```
nba_combined$Pos <- as.factor(nba_combined$Pos)
nba_split <- initial_split(nba_combined, prop=0.7, strata = "WS_48")
nba_train <- training(nba_split)
nba_test <- testing(nba_split)
```

Folding the data

2. fold the training data using v-fold cross-validation into 10 folds and also stratifying the folds based on a player's win shares per 48 min.

```r
nba_folds <- vfold_cv(nba_train, v=10, strata='WS_48')
```

Create recipe

3. Set up a recipe to predict `WS_48`. I chose to use all predictors except for Player name and our response variable WS per 48 min. In addition to the above, I dummy coded the categorical variables, which just ended up being `Pos`, removed columns with a single unique value (which shouldn't have been an issue), and centered and scaled all predictors.

```r
nba_recipe <- recipe(`WS_48` ~ Pos + Age + FG + FGA + `FG%` + `3P` + `3PA` + `3P%` +`2P` + `2PA` + `2P%
                    step_dummy(all_nominal_predictors()) %>%
                    step_zv(all_predictors()) %>%
                    step_normalize(all_predictors())
nba_recipe
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         26
##
## Operations:
##
## Dummy variables from all_nominal_predictors()
## Zero variance filter on all_predictors()
## Centering and scaling for all_predictors()
```

Model Building

The models that I chose to use were ridge regression, random forest, boosted trees, support vector machine, and k-nearest neighbors.

Before we dive straight into model building, however, let's revisit one thing.

Decorrelation

Remember the correlation plot that I displayed earlier? You might be wondering, how come we don't have to do anything to decorrelate our predictors despite so many of them being correlated? Referring to this textbook's recommendations for preprocessing, although linear regression requires decorrelation, because we chose to use ridge regression, it handles it for us, so we do not have to worry about it. The other models like rand_forest() do not require decorrelation either, except for support vector machines, which I will address once we get there.

So now, let's get into building our models and fitting them to the training data.

ridge regression

Our first model is a ridge regression model, which we set up by using the `linear_reg()` function with the `glmnet` engine, setting `mixture` = 0, and tuning penalty.

```r
# ridge model
ridge_spec <-
  linear_reg(penalty = tune(), mixture = 0) %>%
  set_mode("regression") %>%
  set_engine("glmnet")
```

After setting up the model, we create a workflow using it and our recipe above.

```r
# create workflow
ridge_workflow <- workflow() %>%
  add_recipe(nba_recipe) %>%
  add_model(ridge_spec)
```

Next, we need to set up the tune grid. We allow R to automatically configure our tune grid for us, using the `extract_parameter_set_dials()` function. We use `levels = 50` since our engine `glmnet` will fit all at the same time.

```r
ridge_params <- extract_parameter_set_dials(ridge_workflow)
ridge_grid <- grid_regular(ridge_params, levels = 50)
```

Finally, combining all from the above, we fit our model to the folded data and saved the result for later use.

```r
# fit to model
ridge_res <- tune_grid(
  ridge_workflow,
  resamples = nba_folds,
  grid = ridge_grid
)

# save(ridge_res, file = "r_scripts/ridge_res.rda")
```

random forest

In our next model, the random forest model, we essentially follow the same pattern. First, I set up the model using the `ranger` engine, and tuned mtry, trees, and min_n, then set up the workflow. Next, I used the same process to create the tune grid, except that I manually updated the values for mtry, ranging from 1 to 26 because those are the number of predictors that we have. Finally, we fit and tuned the model and saved it.

```r
rf_spec <- rand_forest(mtry = tune(), trees = tune(), min_n = tune()) %>%
        set_engine("ranger", importance="impurity") %>%
  set_mode("regression")


rf_workflow <- workflow() %>%
  add_recipe(nba_recipe) %>%
    add_model(rf_spec)

rf_params <- extract_parameter_set_dials(rf_workflow) %>% update(mtry = mtry(range= c(1, 26)))
rf_grid <- grid_regular(rf_params, levels = 10)
#rf_grid <- grid_regular(mtry(range = c(1, 26)), trees(range = c(1, 1000)), min_n(range = c(2, 40)), le
```

```
rf_res <- tune_grid(
  rf_workflow,
  resamples = nba_folds,
  grid = rf_grid
)

# save(rf_res, file = "r_scripts/rf_res.rda")
```

boosted trees

Similar to the random forest, the way that I set up the boosted tree was exactly the same. Create model, tuning mtry, trees, and min_n, and set up the workflow. Then, created the tuning grid while specifying the values for mtry, and finally fit and saved the results.

```
# create boosted trees model
boost_spec <- boost_tree(trees=tune(), min_n = tune(), mtry = tune()) %>%
            set_engine("xgboost") %>%
            set_mode("regression")

# create workflow
boost_workflow <- workflow() %>%
                  add_model(boost_spec) %>%
                  add_recipe(nba_recipe)


# define grid to tune parameters
boost_params <- extract_parameter_set_dials(boost_workflow) %>% update(mtry = mtry(range= c(1, 26)))
boost_grid <- grid_regular(boost_params, levels = 10)
```

```
# fit everything to model
boost_res <- tune_grid(
  boost_workflow,
  resamples = nba_folds,
  grid=boost_grid
)

# save(boost_res, file = "r_scripts/boost_res.rda")
```

support vector machine

Setting up the model is essentially the same as the above few, except for the fact that we have to address the correlation between predictors. In order to decorrelate our predictors, we will use PCA. We first run `step_YeoJohnson` in order to handle the effect of skewed values. Then, we run step__pca, tuning `num_comp`, which is the number of componenets to maintain as predictors, of which we set the range to be from 1 to 26. Finally, besides those steps, much of how we set it up was the same.

```
svm_recipe <- recipe(`WS_48` ~ Pos + Age + FG + FGA + `FG%` + `3P` + `3PA` + `3P%` +`2P` + `2PA` + `2P%
                    step_YeoJohnson(all_numeric_predictors()) %>%
                    step_dummy(all_nominal_predictors()) %>%
                    step_zv(all_predictors()) %>%
                    step_normalize(all_predictors()) %>%
```

```r
                    step_pca(all_numeric_predictors(), num_comp = tune()) %>%
                    step_normalize(all_numeric_predictors())

svm_spec <- svm_rbf(cost=tune(), rbf_sigma=tune()) %>%
            set_engine("kernlab") %>%
            set_mode("regression")

svm_workflow <- workflow() %>%
                add_model(svm_spec) %>%
                add_recipe(svm_recipe)

svm_params <- extract_parameter_set_dials(svm_workflow) %>% update(num_comp = num_comp(c(1, 26)))
svm_grid <- grid_regular(svm_params, levels = 10)
```

```r
svm_res <- tune_grid(
  svm_workflow,
  resamples = nba_folds,
  grid = svm_grid
)

# save(svm_res, file = "r_scripts/svm_res.rda")
```

K-Nearest Neighbors

So as to not sound repetitive, fitting this model was much of the same behavior, of which I am sure that you are most capable of understanding now after repeating it 4 other times. So, I leave it to you to figure out the code for yourself.

```r
knn_spec <-
  nearest_neighbor(
    neighbors = tune(),
    mode = "regression") %>%
  set_engine("kknn")

knn_workflow <- workflow() %>%
  add_model(knn_spec) %>%
  add_recipe(nba_recipe)

knn_params <- parameters(knn_spec)
knn_grid <- grid_regular(knn_params, levels = 10)
```

```r
knn_res <- tune_grid(
  knn_workflow,
  resamples = nba_folds,
  grid = knn_grid)

# save(knn_res, knn_workflow, file = "r_scripts/knn_res.rda")
```

Finally, with all the models fit and looking prim and proper, we move on to analyzing our results and how well each model performed.

Evaluating model performance

To not have to run all the models every time, we decided to save them and load them all here to use in evaluating model performance.

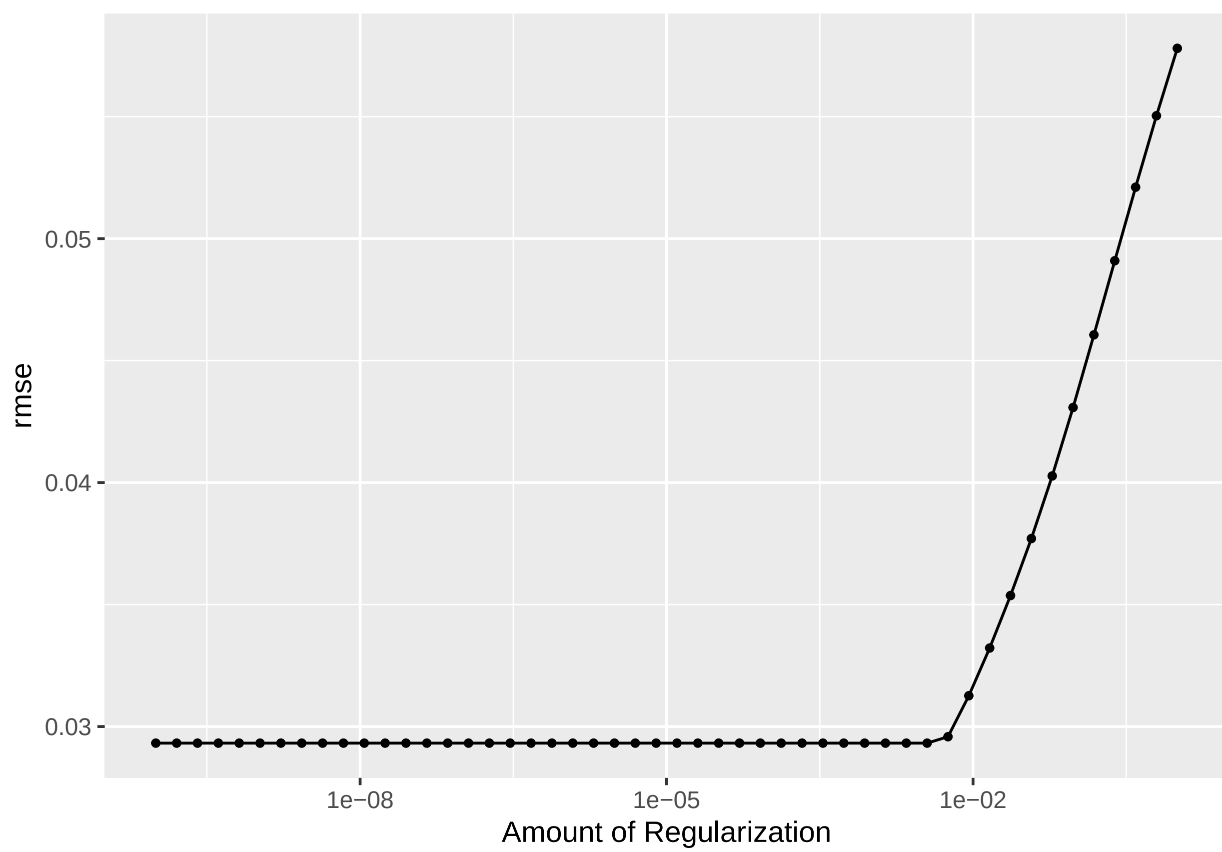
```
load("r_scripts/ridge_res.rda")
load("r_scripts/rf_res.rda")
load("r_scripts/boost_res.rda")
load("r_scripts/svm_res.rda")
load("r_scripts/knn_res.rda")
```

For each model, we will approach it the same way. I will demonstrate it with our first model, the ridge regression.

ridge regression

First, we run the `autoplot()` function in order to see how our performance metrics change in response the different values of our tuned variables. The metric that we are going to use to evaluate the performance is `rmse`, which is a measure of how much our estimates deviate from the actual values on average.

```
autoplot(ridge_res, metric = "rmse")
```



It appears that rmse is very small when `penalty` is small, and stays relatively consistent until `penalty` reaches around 0.01, where rmse increaes linearly.

Next, we find the best performing model of this type of model, so in this case, we try to find the best-performing ridge regression model, by using `collect_metrics` and `arrange` to arrange each in order of performance. We can also achieve this result by using `show_best`.

```
ridge_metrics <- collect_metrics(ridge_res) %>% arrange((mean))
ridge_metrics <- ridge_metrics[which(ridge_metrics$.metric == 'rmse'),]
head(ridge_metrics)
```

```
## # A tibble: 6 x 7
##    penalty .metric .estimator   mean     n std_err .config
##      <dbl> <chr>   <chr>       <dbl> <int>   <dbl> <chr>
## 1 1   e-10 rmse    standard   0.0293    10 0.00189 Preprocessor1_Model01
## 2 1.60e-10 rmse    standard   0.0293    10 0.00189 Preprocessor1_Model02
## 3 2.56e-10 rmse    standard   0.0293    10 0.00189 Preprocessor1_Model03
## 4 4.09e-10 rmse    standard   0.0293    10 0.00189 Preprocessor1_Model04
## 5 6.55e-10 rmse    standard   0.0293    10 0.00189 Preprocessor1_Model05
## 6 1.05e- 9 rmse    standard   0.0293    10 0.00189 Preprocessor1_Model06
```

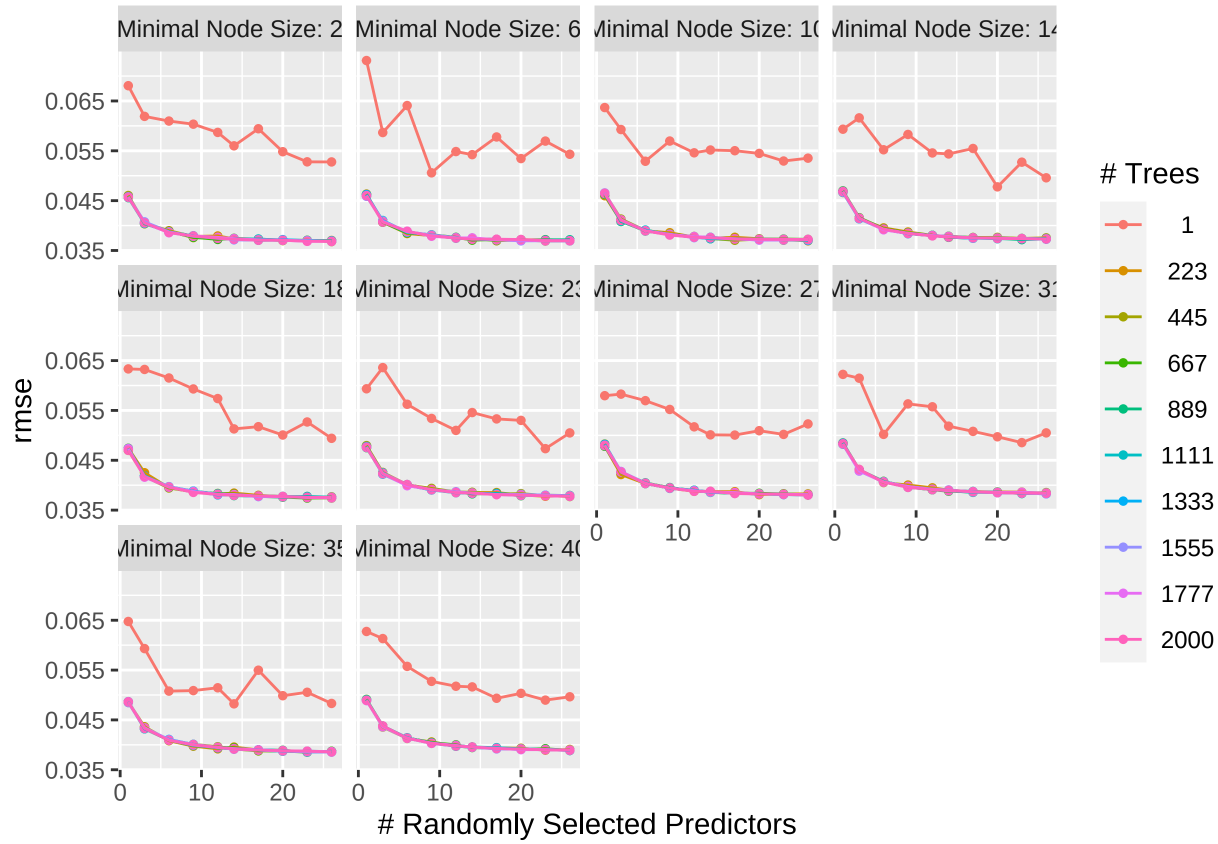
```
show_best(ridge_res, metric = "rmse")
```

```
## # A tibble: 5 x 7
##    penalty .metric .estimator   mean     n std_err .config
##      <dbl> <chr>   <chr>       <dbl> <int>   <dbl> <chr>
## 1 1   e-10 rmse    standard   0.0293    10 0.00189 Preprocessor1_Model01
## 2 1.60e-10 rmse    standard   0.0293    10 0.00189 Preprocessor1_Model02
## 3 2.56e-10 rmse    standard   0.0293    10 0.00189 Preprocessor1_Model03
## 4 4.09e-10 rmse    standard   0.0293    10 0.00189 Preprocessor1_Model04
## 5 6.55e-10 rmse    standard   0.0293    10 0.00189 Preprocessor1_Model05
```

From this, we see that the best-performing ridge regression model has an rmse value of `0.02932064`, which means that we are a mere 0.02932064 away from the actual `WS_48` on average.

random forest

```
autoplot(rf_res, metric = "rmse")
```




```r
rf_metrics <- collect_metrics(rf_res) %>% arrange((mean))
rf_metrics <- rf_metrics[which(rf_metrics$.metric == 'rmse'),]
head(rf_metrics)
```

```
## # A tibble: 6 x 9
##    mtry trees min_n .metric .estimator   mean     n std_err .config
##   <int> <int> <int> <chr>   <chr>       <dbl> <int>   <dbl> <chr>
## 1    26   445     2 rmse    standard   0.0368    10 0.00284 Preprocessor1_Model~
## 2    26  2000     2 rmse    standard   0.0368    10 0.00283 Preprocessor1_Model~
## 3    26  1555     2 rmse    standard   0.0368    10 0.00285 Preprocessor1_Model~
## 4    23  1777     2 rmse    standard   0.0368    10 0.00293 Preprocessor1_Model~
## 5    23  2000     2 rmse    standard   0.0369    10 0.00291 Preprocessor1_Model~
## 6    26  1111     2 rmse    standard   0.0369    10 0.00284 Preprocessor1_Model~
```

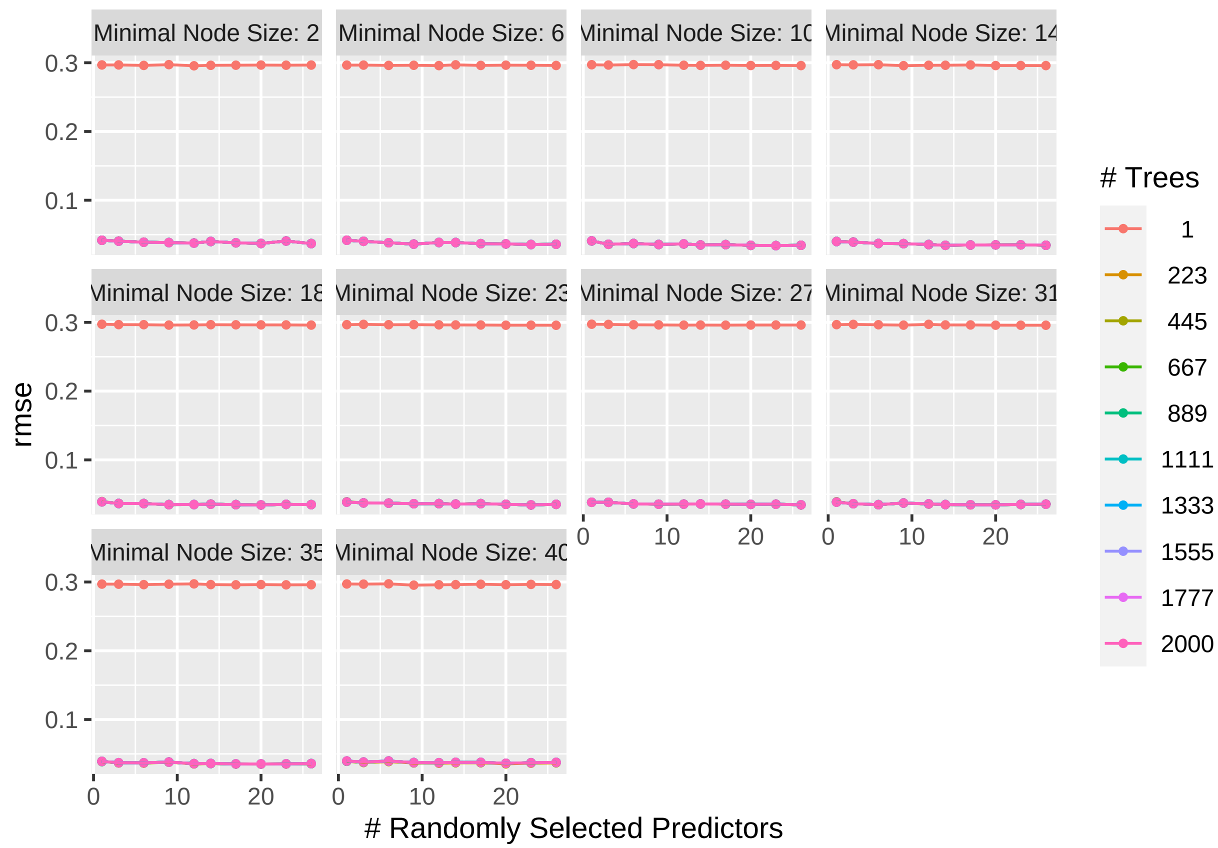
```r
show_best(rf_res, metric='rmse')
```

```
## # A tibble: 5 x 9
##    mtry trees min_n .metric .estimator   mean     n std_err .config
##   <int> <int> <int> <chr>   <chr>       <dbl> <int>   <dbl> <chr>
## 1    26   445     2 rmse    standard   0.0368    10 0.00284 Preprocessor1_Model~
## 2    26  2000     2 rmse    standard   0.0368    10 0.00283 Preprocessor1_Model~
## 3    26  1555     2 rmse    standard   0.0368    10 0.00285 Preprocessor1_Model~
## 4    23  1777     2 rmse    standard   0.0368    10 0.00293 Preprocessor1_Model~
## 5    23  2000     2 rmse    standard   0.0369    10 0.00291 Preprocessor1_Model~
```

The `autoplot` of our random forest models show us that rmse is small when the number of trees increase, although there is not much difference when `trees` is between 200 to 2000. Looking at the other tuned parameters, I can't tell much difference between the values of `min_n`, but is more apparent that the more predictors selected (`mtry`) the better performing the model is. Finally, our best-performing random forest model has an rmse of 0.03678628, when mtry = 26, trees = 445, and min_n = 2.

boosted trees

```
autoplot(boost_res, metric = "rmse")
```




```
boost_metrics <- collect_metrics(boost_res) %>% arrange((mean))
boost_metrics <- boost_metrics[which(rf_metrics$.metric == 'rmse'),]
head(boost_metrics)
```

```
## # A tibble: 6 x 9
##    mtry trees min_n .metric .estimator   mean     n std_err .config
##   <int> <int> <int> <chr>   <chr>       <dbl> <int>   <dbl> <chr>
## 1    23   667    10 rmse    standard   0.0340    10 0.00257 Preprocessor1_Model~
## 2    23   889    10 rmse    standard   0.0340    10 0.00257 Preprocessor1_Model~
## 3    23   445    10 rmse    standard   0.0340    10 0.00257 Preprocessor1_Model~
## 4    23  1111    10 rmse    standard   0.0340    10 0.00257 Preprocessor1_Model~
## 5    23  1333    10 rmse    standard   0.0340    10 0.00257 Preprocessor1_Model~
## 6    23  1555    10 rmse    standard   0.0340    10 0.00257 Preprocessor1_Model~
```

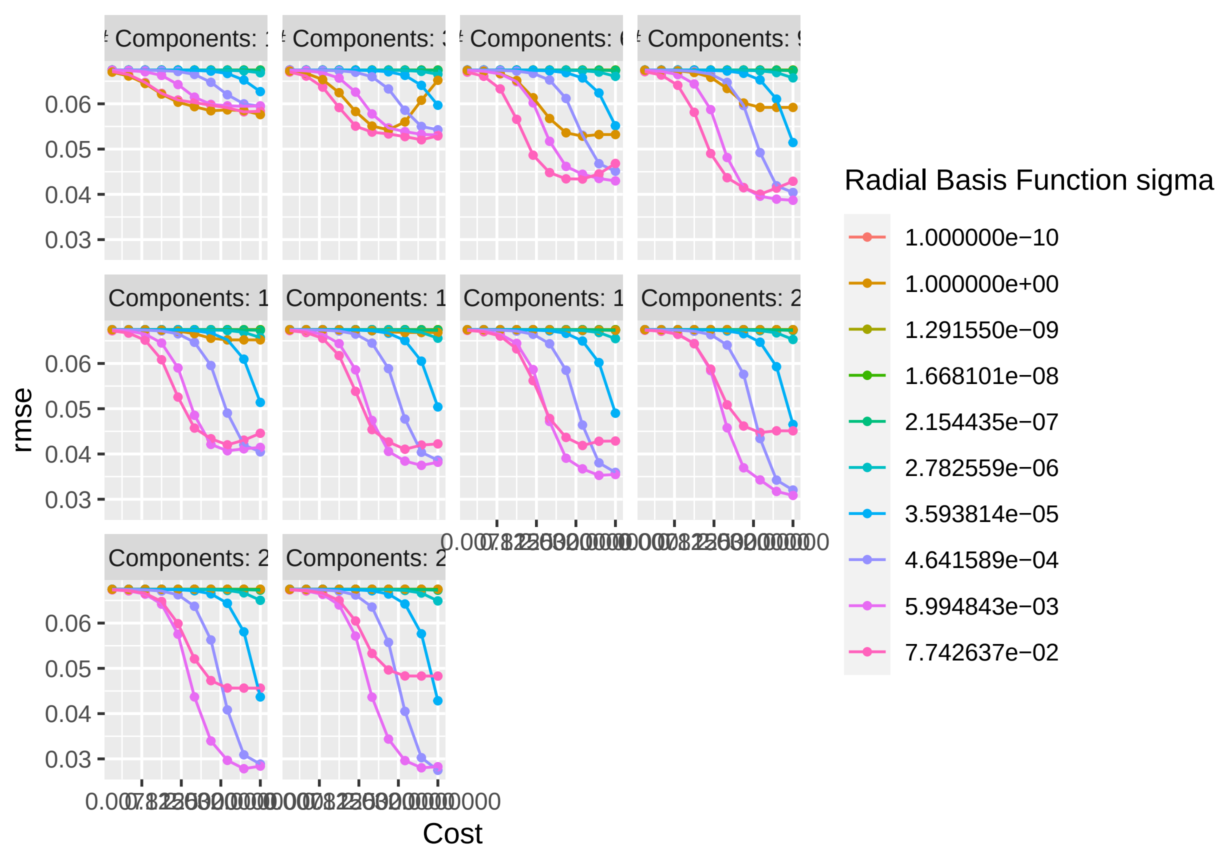
```r
show_best(boost_res, metric = "rmse")
```

```
## # A tibble: 5 x 9
##    mtry trees min_n .metric .estimator   mean     n std_err .config
##   <int> <int> <int> <chr>   <chr>       <dbl> <int>   <dbl> <chr>
## 1    23   667    10 rmse    standard   0.0340    10 0.00257 Preprocessor1_Model~
## 2    23   889    10 rmse    standard   0.0340    10 0.00257 Preprocessor1_Model~
## 3    23   445    10 rmse    standard   0.0340    10 0.00257 Preprocessor1_Model~
## 4    23  1111    10 rmse    standard   0.0340    10 0.00257 Preprocessor1_Model~
## 5    23  1333    10 rmse    standard   0.0340    10 0.00257 Preprocessor1_Model~
```

Looking at our `autoplot()` result, it seems that min_n and mtry do not affect rmse, and only the number of trees seems to affect it, with the more trees the better. But, this could be because the scale of our graph is too small, so we cannot see the effects of our tuned parameters. Finally, our best-performing random forest model has an rmse of 0.03403662, when mtry = 23, trees = 667, and min_n = 10.

svm

```r
autoplot(svm_res, metric= "rmse")
```




```r
svm_metrics <- collect_metrics(svm_res) %>% arrange((mean))
svm_metrics <- svm_metrics[which(rf_metrics$.metric == 'rmse'),]
head(svm_metrics)
```

```
## # A tibble: 6 x 9
##    cost rbf_sigma num_comp .metric .estimator   mean     n std_err .config
##   <dbl>     <dbl>    <int> <chr>   <chr>       <dbl> <int>   <dbl> <chr>
## 1  32    0.000464       26 rmse    standard   0.0275    10 0.00227 Preprocessor~
## 2  10.1  0.00599        23 rmse    standard   0.0278    10 0.00430 Preprocessor~
## 3  10.1  0.00599        26 rmse    standard   0.0280    10 0.00374 Preprocessor~
## 4  32    0.00599        26 rmse    standard   0.0283    10 0.00348 Preprocessor~
## 5  32    0.00599        23 rmse    standard   0.0284    10 0.00428 Preprocessor~
## 6  32    0.000464       23 rmse    standard   0.0288    10 0.00222 Preprocessor~
```

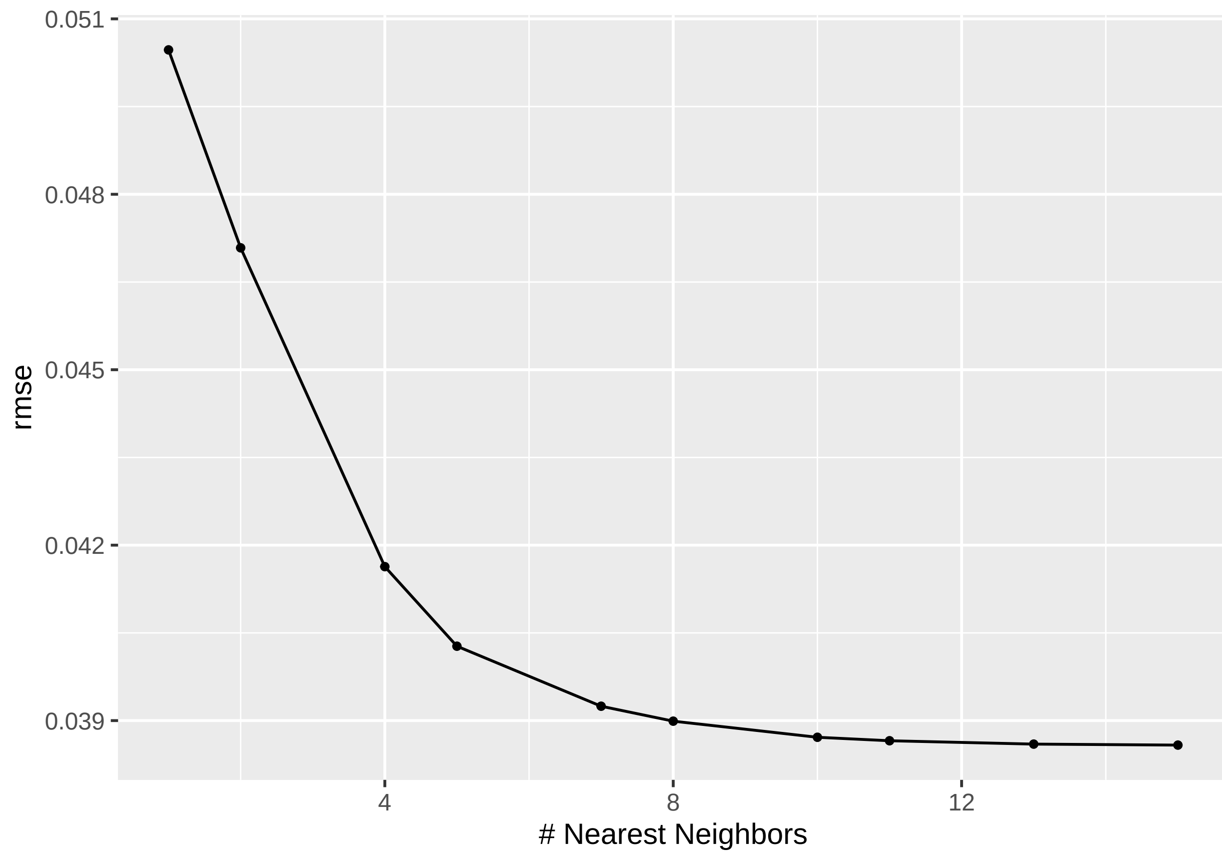
```
select_best(svm_res, metric='rmse')
```

```
## # A tibble: 1 x 4
##    cost rbf_sigma num_comp .config
##   <dbl>     <dbl>    <int> <chr>
## 1    32  0.000464       26 Preprocessor10_Model070
```

From `autoplot()`, we see the higher values of componenents and cost result in lower rmse, but it does not seem that `rbf_sigma` follows that same pattern. Our best-performing support vector machine model has an rmse of 0.02748319, with cost = 32, rbf_sigma = 0.0004641589, and num_comp = 26, which is our lowest thus far!

nearest neighbour

```
autoplot(knn_res, metric= "rmse")
```




```
collect_metrics(knn_res) %>% arrange((mean))
```

```
## # A tibble: 20 x 7
##    neighbors .metric .estimator   mean     n  std_err .config
##        <int> <chr>   <chr>       <dbl> <int>    <dbl> <chr>
##  1        15 rmse    standard   0.0386     3 0.00134  Preprocessor1_Model10
##  2        13 rmse    standard   0.0386     3 0.00106  Preprocessor1_Model09
##  3        11 rmse    standard   0.0387     3 0.000671 Preprocessor1_Model08
##  4        10 rmse    standard   0.0387     3 0.000478 Preprocessor1_Model07
##  5         8 rmse    standard   0.0390     3 0.000228 Preprocessor1_Model06
##  6         7 rmse    standard   0.0392     3 0.000511 Preprocessor1_Model05
##  7         5 rmse    standard   0.0403     3 0.00127  Preprocessor1_Model04
##  8         4 rmse    standard   0.0416     3 0.00171  Preprocessor1_Model03
##  9         2 rmse    standard   0.0471     3 0.00265  Preprocessor1_Model02
## 10         1 rmse    standard   0.0505     3 0.00386  Preprocessor1_Model01
## 11         1 rsq     standard   0.384      3 0.0977   Preprocessor1_Model01
## 12         2 rsq     standard   0.415      3 0.0900   Preprocessor1_Model02
## 13         4 rsq     standard   0.493      3 0.0858   Preprocessor1_Model03
## 14         5 rsq     standard   0.519      3 0.0816   Preprocessor1_Model04
## 15         7 rsq     standard   0.537      3 0.0717   Preprocessor1_Model05
## 16         8 rsq     standard   0.541      3 0.0655   Preprocessor1_Model06
## 17        10 rsq     standard   0.548      3 0.0569   Preprocessor1_Model07
## 18        11 rsq     standard   0.550      3 0.0531   Preprocessor1_Model08
## 19        13 rsq     standard   0.553      3 0.0445   Preprocessor1_Model09
## 20        15 rsq     standard   0.555      3 0.0383   Preprocessor1_Model10
```

```r
select_best(knn_res, metric='rmse')
```

```
## # A tibble: 1 x 2
##   neighbors .config
##       <int> <chr>
## 1        15 Preprocessor1_Model10
```

Our final model, `autoplot` shows us that rmse decreases as we increase the number of nearest neighbors. Correspondingly, our best-performing nearest neighbor model has an rmse of 0.03858120 when neighbors = 15.

Thus, from all our models, we see that our best performing model was the support vector machine model with parameters cost = 32, rbf_sigma = 0.0004641589, and num_comp = 26.

Final Model Fitting: Fitting the best model to the testing data set

Using our best model, we now fit it to the training data set.

First, let's select the best performing svm model and double check that it matches what we have above.

```r
best_svm <- select_best(svm_res, metric='rmse')
best_svm
```

```
## # A tibble: 1 x 4
##    cost rbf_sigma num_comp .config
##   <dbl>     <dbl>    <int> <chr>
## 1    32  0.000464       26 Preprocessor10_Model070
```

Since it matches, we can now finalize our workflow by using the parameters of the best model and the `finalize_workflow` function.

```r
svm_final <- finalize_workflow(svm_workflow, best_svm)
svm_final_fit <- fit(svm_final, data=nba_train)
```

Testing Set

Finally, we conclude our analysis by fitting the model to the testing set.

```r
nba_test_res <- predict(svm_final_fit, new_data = nba_test) %>% bind_cols(nba_test %>% select(`WS_48`))

nba_metrics <- metric_set(rsq, rmse)
nba_test_res %>% nba_metrics(truth=`WS_48`, estimate = .pred)
```

```
## # A tibble: 2 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rsq     standard      0.844
## 2 rmse    standard      0.0265
```

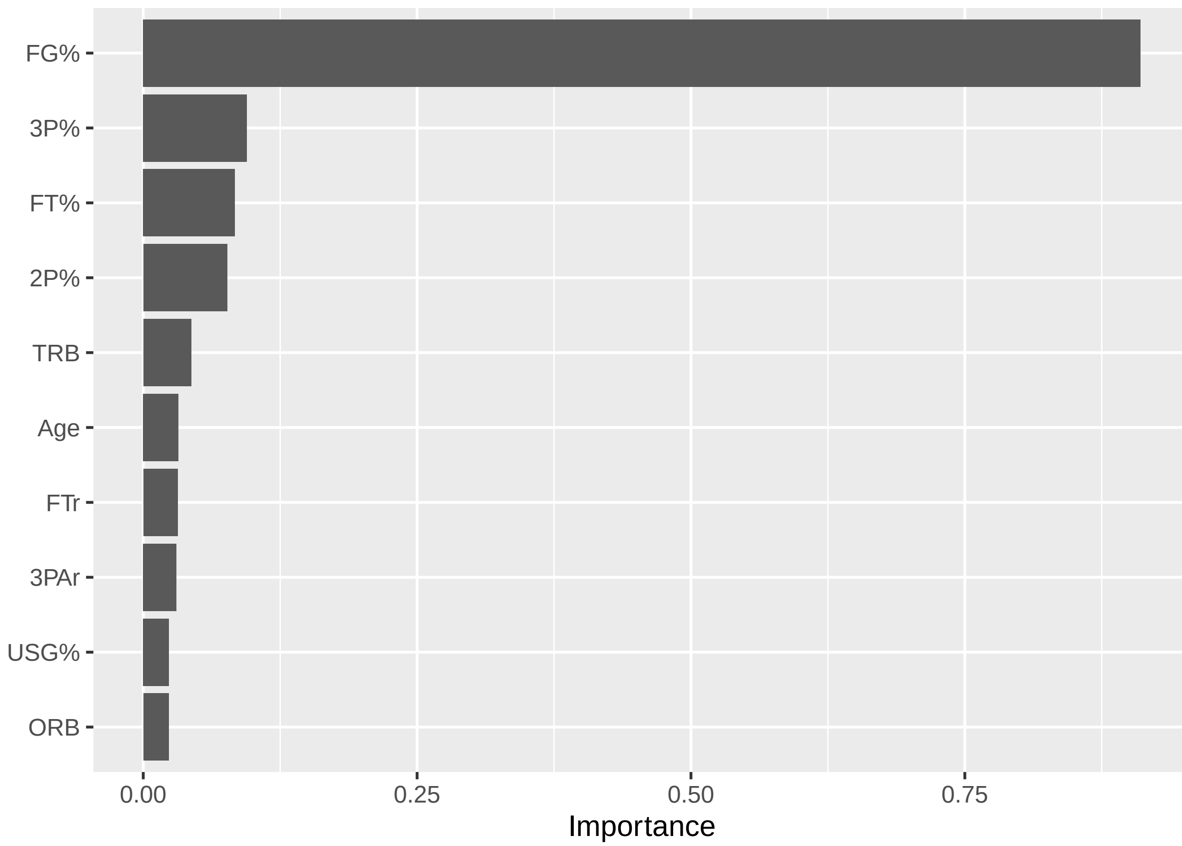
Our training set results returned an rmse of 0.02574668, which means that we are only 0.02574668 away from the actual `WS_48` on average. This is really, really good! (suspciously a little too good). In comparison to the testing data, it is fairly close to our testing set's rmse value of 0.02748319, which indicates that we likely did not overfit our model. So we see that our model does a good job at predicting the number of win shares that a player will accumulate in 48 minutes based on the given predictors.

Variable Importance

One interest that I mentioned earlier was to see which variables are most conducive towards winning. One way that we can do this is to use vip, or variable importance plot, and see which variables are most important as hinted by the name. Support vector machine models are not currently supported by `vip()`, so we instead will use a random forest model as substitute, but results should not differ dramatically (I hope).

```
best_rf <- select_best(rf_res, metric='rmse')
rf_final <- finalize_workflow(rf_workflow, best_rf)
rf_final_fit <- fit(rf_final, data=nba_train)

vip(rf_final_fit %>% extract_fit_parsnip())
```



We see that field goal percentage is the overwhelming winner here. So in general, players who are efficient in scoring (or makes more shots than they miss) will always be valuable to NBA teams. What is somewhat surprising is that `PTS` is not that important, as I had suspected it would be, but perhaps there are other factors in play…

Conclusion

To summarize all that we've done, I started with an nba dataset and wanted to see if I could predict player's win shares per 48 minute using the 2020-2021 NBA season data. After fitting and running various models, I ultimately decided on using a support vector machine model, and its predictions of `WS_48` only deviated by 0.02574668 on average when run on the testing set.

Looking at the results of our fitted models and its predictions, we see that our results are extremely good. Is this due to our model? Or could there be other factors. One factor that I considered is that we may have used too many predictors by using every single NBA statistic. But, perhaps more crucially, one aspect that I failed to consider when running through this project is that win shares is not an actually statistic that someone calculates while the game is going on, but it is a calculated statistic. The formula for win shares is overly complicated, so I will spare you the details, but since it is calculated, it may be possible that some predictors are linearly dependent with win shares, and thus will naturally be good predictors. Perhaps it may be a good idea for to explore the math behind win shares in the future.

Speaking of future studies, just looking at the website where I pulled the data from, it seems that there is a lot of data to work with and explore. There are so many variables in what constitutes an impactful player in the NBA, that even despite all the time and effort that the brilliant statisticians in the industry put in, they are still unable to correctly predict good players 100% of the time. Scrolling through the dataset, there are so many variables and data sets that we can use to play around with. For instance, rather than using win shares, we could actually track how many wins a player's team actually got. We could do classification problems, such as trying to predict MVP results or trying to predict who wins the championship. There are a lot of future options that I am interested in playing around with.

All in all, I hope that you have found my exploration and fiddling around with the NBA dataset interesting enough. And I hope that even if you've never watched basketball before, that you will give it a try and become as enthralled as I, and many others, are. Thank you for reading!