PSTAT131 Final Project

Yifei Zhang

2022-05-11

Introduction

About This Project

If you recognize characters from the picture above, you can probably tell we are going to explore the E sport/ video games field in this project. We are looking more specifically into the determining factors for winning a match in a game with the notoriously toxic gaming community, League of Legends. There are many types of people in this world, and a lot of us fall into two categories, the The League of Legends player category, and the victims of the first category, the player's friend who get forced to watch them playing knowing they are going to lose and even have a temper tantrum afterwards sometimes category. I am a part of the latter group, and to avoid spending extra half an hour watching/playing a game that I know is going to lose, which will lead us to a bad mental state, our friendships on the edge, I want to make a model that can predict the game result as accurately as possible given the first ten minutes game play statistics. Or at least get to know what the most important factors in winning a match are.

About This Dataset

The League of Legends Diamond Ranked Games dataset includes the first ten minutes statistics of approximately ten thousands ranked League of Legends matches (solo queue) ranging from diamond to master ranking. For background information, League of Legends is a multiplayer online battle arena (MOBA) game where there are 3 lanes, a jungle, and 5 player roles each for the 2 teams (blue and red). The first one to take down the enemy Nexus wins the game.

Here is some basic information about the The League of Legends Diamond Ranked Games dataset. The data is obtained from user MICHEL'S FANBOI who seems to have changed their username pretty frequently, on Kaggle, and their source is Riot Games, the developer of League of Legends, API. You can find the dataset following the link here https://www.kaggle.com/datasets/bobbyscience/league-of-legends-diamond-ranked-games-10-min?resource=download. In this dataset there are 9879 observations, and 38 predictors in total.

Tidying

Loading Packages and Data

```
library(ggplot2)
library(tidyverse)
library(tidymodels)
library(corrplot)
library(ggthemes)
library(discrim)
library(poissonreg)
library(corrr)
library(klaR)
library(ISLR)
library(ISLR2)
library(purrr)
library(janitor)
library(psych)
library(randomForest)
library(xgboost)
library(rpart.plot)
library(ranger)
library(vip)
library(pROC)
tidymodels_prefer()
```

```
loldata <- read_csv("DATA/high_diamond_ranked_10min.csv")
```

Check for Missing Values

Before tidying we want to make sure we are not working with a significant amount of missing data. If we do, we want to make sure to tidy the records with significant amount of missing entries out for better accuracy.

```
is.null(loldata)
```

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

Change Variable Names

Since we do not have any null entries, wee can directly move on to the next part, normally we would need to deselect some rows or fill in the null with zeros before moving on.

Although from a glance the variable names look pretty unique and not problem causing, we want to use clean the names to avoid potential problems in the future, such as forgetting to capitalize certain letters in the variable name.

```
# save the cleaned data
lol <- clean_names(loldata)

# print the new names
# for later variable selection purpose, also makes life easier
colnames(lol)
```

```
##  [1] "game_id"                          "blue_wins"
##  [3] "blue_wards_placed"                "blue_wards_destroyed"
##  [5] "blue_first_blood"                 "blue_kills"
##  [7] "blue_deaths"                      "blue_assists"
##  [9] "blue_elite_monsters"              "blue_dragons"
## [11] "blue_heralds"                     "blue_towers_destroyed"
## [13] "blue_total_gold"                  "blue_avg_level"
## [15] "blue_total_experience"            "blue_total_minions_killed"
## [17] "blue_total_jungle_minions_killed" "blue_gold_diff"
## [19] "blue_experience_diff"             "blue_cs_per_min"
## [21] "blue_gold_per_min"                "red_wards_placed"
## [23] "red_wards_destroyed"              "red_first_blood"
## [25] "red_kills"                        "red_deaths"
## [27] "red_assists"                      "red_elite_monsters"
## [29] "red_dragons"                      "red_heralds"
## [31] "red_towers_destroyed"             "red_total_gold"
## [33] "red_avg_level"                    "red_total_experience"
## [35] "red_total_minions_killed"         "red_total_jungle_minions_killed"
## [37] "red_gold_diff"                    "red_experience_diff"
## [39] "red_cs_per_min"                   "red_gold_per_min"
```

Select important Variables

Since there are only two teams, Blue and Red, and many of the variables are coded in 1 and 0 that represents either blue or red got it just in the opposite way, a lot of them are repetitive to look at. For example, there are if the entry for our blue_wins is 0, then we know the corresponding entry for red_wins is 1. So we want to deselect some repetitive variables from our data set. It does not matter if blue or red wins, if blue loses then obviously red wins. In this case, we will work on classifying if team blue wins or not.

```
lol_blue <-  lol[ , 0:21]
colnames(lol_blue) # check if we have the right columns
```

```
##  [1] "game_id"                "blue_wins"
##  [3] "blue_wards_placed"      "blue_wards_destroyed"
##  [5] "blue_first_blood"       "blue_kills"
##  [7] "blue_deaths"            "blue_assists"
```

```
##  [9] "blue_elite_monsters"              "blue_dragons"
## [11] "blue_heralds"                     "blue_towers_destroyed"
## [13] "blue_total_gold"                  "blue_avg_level"
## [15] "blue_total_experience"            "blue_total_minions_killed"
## [17] "blue_total_jungle_minions_killed" "blue_gold_diff"
## [19] "blue_experience_diff"             "blue_cs_per_min"
## [21] "blue_gold_per_min"
```

For this part please take a look at the Exploratory Data Analysis section first. We want to extract the unique variables that appears to be significantly correlated with *blue_wins*. For example, although both *blue_total_gold"* and *blue_gold_per_min* appear to be highly positively correlated with *blue_wins*, the latter is directly correlated to the prior, and we do not need it.

```
important <- lol_blue[c( "blue_wins", "blue_first_blood",
                   "blue_kills",  "blue_deaths",
                   "blue_assists","blue_elite_monsters",
                   "blue_dragons", "blue_total_gold",
                   "blue_avg_level", "blue_total_experience",
                   "blue_gold_diff", "blue_experience_diff",
                   "blue_total_minions_killed")]
```

```
important
```

```
## # A tibble: 9,879 x 13
##    blue_wins blue_first_blood blue_kills blue_deaths blue_assists
##        <dbl>            <dbl>      <dbl>       <dbl>        <dbl>
##  1         0                1          9           6           11
##  2         0                0          5           5            5
##  3         0                0          7          11            4
##  4         0                0          4           5            5
##  5         0                0          6           6            6
##  6         1                0          5           3            6
##  7         1                1          7           6            7
##  8         0                0          5          13            3
##  9         0                0          7           7            8
## 10         1                1          4           5            5
## # ... with 9,869 more rows, and 8 more variables: blue_elite_monsters <dbl>,
## #   blue_dragons <dbl>, blue_total_gold <dbl>, blue_avg_level <dbl>,
## #   blue_total_experience <dbl>, blue_gold_diff <dbl>,
## #   blue_experience_diff <dbl>, blue_total_minions_killed <dbl>
```

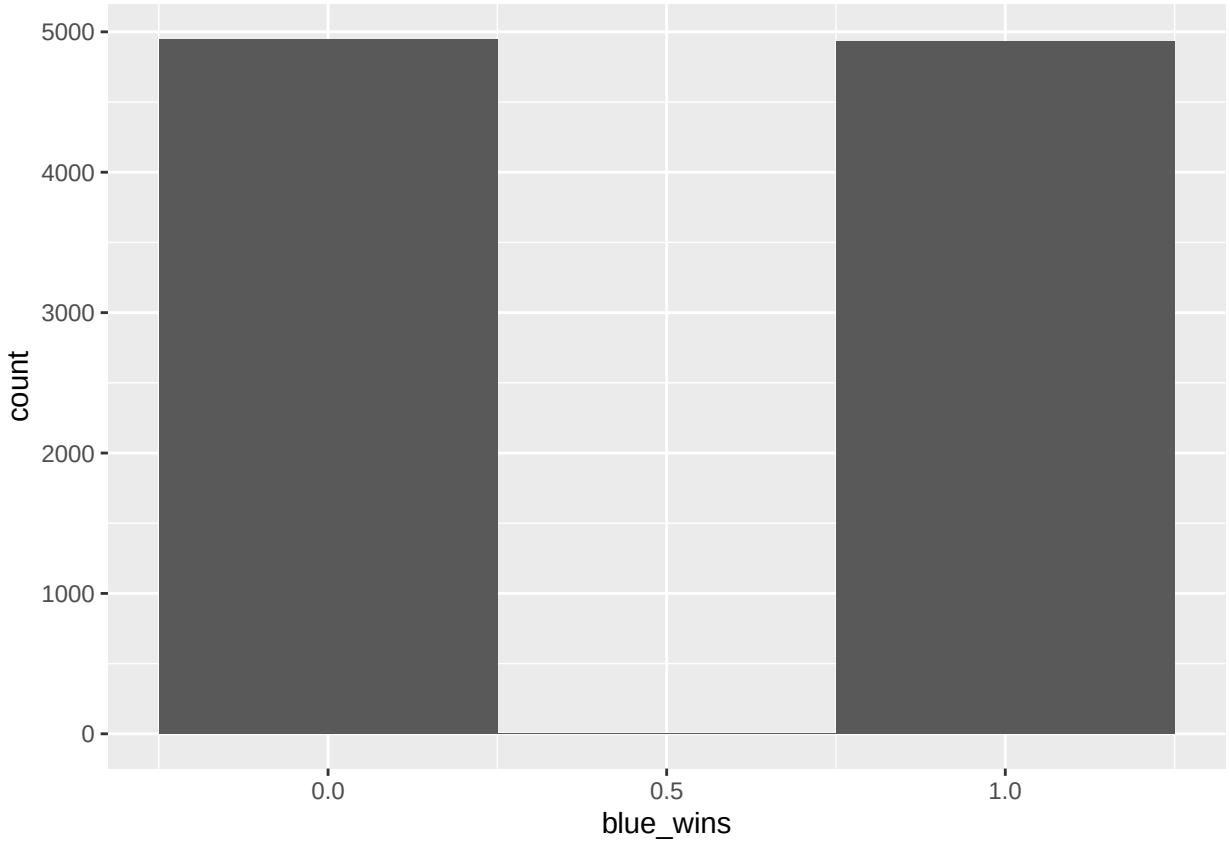
Exploratory Data Analysis

Check Fairness of Data

We want to make sure our data sample was randomly drawn.

From the result we can see there is a very slight difference(insignificant) between the number of blue wins and loses.

```
lol %>%
  ggplot(aes(x = blue_wins)) +
  geom_histogram(bins = 3)
```



Summary Table

Before start working on our model, we want to have a look at the summary table to have a general idea on what we are working with. This is more convenient than looking through our raw dataset that has way too many entries to look at.

```
describe(lol)
```

```
##                          vars    n          mean          sd
## game_id                     1 9879 4500084044.85 27573278.49
## blue_wins                   2 9879          0.50        0.50
## blue_wards_placed           3 9879         22.29       18.02
## blue_wards_destroyed        4 9879          2.82        2.17
## blue_first_blood            5 9879          0.50        0.50
## blue_kills                  6 9879          6.18        3.01
## blue_deaths                 7 9879          6.14        2.93
## blue_assists                8 9879          6.65        4.06
## blue_elite_monsters         9 9879          0.55        0.63
## blue_dragons               10 9879          0.36        0.48
## blue_heralds               11 9879          0.19        0.39
```

```
## blue_towers_destroyed              12 9879         0.05       0.24
## blue_total_gold                    13 9879     16503.46    1535.45
## blue_avg_level                     14 9879         6.92       0.31
## blue_total_experience              15 9879     17928.11    1200.52
## blue_total_minions_killed          16 9879       216.70      21.86
## blue_total_jungle_minions_killed   17 9879        50.51       9.90
## blue_gold_diff                     18 9879        14.41    2453.35
## blue_experience_diff               19 9879       -33.62    1920.37
## blue_cs_per_min                    20 9879        21.67       2.19
## blue_gold_per_min                  21 9879      1650.35     153.54
## red_wards_placed                   22 9879        22.37      18.46
## red_wards_destroyed                23 9879         2.72       2.14
## red_first_blood                    24 9879         0.50       0.50
## red_kills                          25 9879         6.14       2.93
## red_deaths                         26 9879         6.18       3.01
## red_assists                        27 9879         6.66       4.06
## red_elite_monsters                 28 9879         0.57       0.63
## red_dragons                        29 9879         0.41       0.49
## red_heralds                        30 9879         0.16       0.37
## red_towers_destroyed               31 9879         0.04       0.22
## red_total_gold                     32 9879     16489.04    1490.89
## red_avg_level                      33 9879         6.93       0.31
## red_total_experience               34 9879     17961.73    1198.58
## red_total_minions_killed           35 9879       217.35      21.91
## red_total_jungle_minions_killed    36 9879        51.31      10.03
## red_gold_diff                      37 9879       -14.41    2453.35
## red_experience_diff                38 9879        33.62    1920.37
## red_cs_per_min                     39 9879        21.73       2.19
## red_gold_per_min                   40 9879      1648.90     149.09
##                                       median       trimmed         mad
## game_id                        4510920346.0 4504103897.37 19856848.76
## blue_wins                             0.0          0.50        0.00
## blue_wards_placed                    16.0         18.29        2.97
## blue_wards_destroyed                  3.0          2.61        1.48
## blue_first_blood                      1.0          0.51        0.00
## blue_kills                            6.0          6.04        2.97
## blue_deaths                           6.0          6.00        2.97
## blue_assists                          6.0          6.31        4.45
## blue_elite_monsters                   0.0          0.47        0.00
## blue_dragons                          0.0          0.33        0.00
## blue_heralds                          0.0          0.11        0.00
## blue_towers_destroyed                 0.0          0.00        0.00
## blue_total_gold                   16398.0      16439.35     1506.32
## blue_avg_level                        7.0          6.92        0.30
## blue_total_experience             17951.0      17946.56     1154.95
## blue_total_minions_killed           218.0        217.21       22.24
## blue_total_jungle_minions_killed     50.0         50.36        8.90
## blue_gold_diff                       14.0         11.84     2360.30
## blue_experience_diff                -28.0        -35.52     1856.22
## blue_cs_per_min                      21.8         21.72        2.22
## blue_gold_per_min                  1639.8       1643.93      150.63
## red_wards_placed                     16.0         18.37        2.97
## red_wards_destroyed                   2.0          2.51        1.48
## red_first_blood                       0.0          0.49        0.00
```

```
## red_kills                              6.0          6.00        2.97
## red_deaths                             6.0          6.04        2.97
## red_assists                            6.0          6.33        4.45
## red_elite_monsters                     0.0          0.50        0.00
## red_dragons                            0.0          0.39        0.00
## red_heralds                            0.0          0.08        0.00
## red_towers_destroyed                   0.0          0.00        0.00
## red_total_gold                     16378.0      16428.95     1466.29
## red_avg_level                          7.0          6.93        0.30
## red_total_experience               17974.0      17982.25     1150.50
## red_total_minions_killed             218.0        217.89       22.24
## red_total_jungle_minions_killed       51.0         51.06       10.38
## red_gold_diff                        -14.0        -11.84     2360.30
## red_experience_diff                   28.0         35.52     1856.22
## red_cs_per_min                        21.8         21.79        2.22
## red_gold_per_min                    1637.8       1642.90      146.63
##                                        min          max       range  skew
## game_id                         4295358071.0 4527990640.0 232632569.0 -1.46
## blue_wins                              0.0          1.0         1.0  0.00
## blue_wards_placed                      5.0        250.0       245.0  4.14
## blue_wards_destroyed                   0.0         27.0        27.0  2.85
## blue_first_blood                       0.0          1.0         1.0 -0.02
## blue_kills                             0.0         22.0        22.0  0.54
## blue_deaths                            0.0         22.0        22.0  0.51
## blue_assists                           0.0         29.0        29.0  0.89
## blue_elite_monsters                    0.0          2.0         2.0  0.69
## blue_dragons                           0.0          1.0         1.0  0.57
## blue_heralds                           0.0          1.0         1.0  1.60
## blue_towers_destroyed                  0.0          4.0         4.0  5.59
## blue_total_gold                    10730.0      23701.0     12971.0  0.47
## blue_avg_level                         4.6          8.0         3.4 -0.34
## blue_total_experience              10098.0      22224.0     12126.0 -0.25
## blue_total_minions_killed             90.0        283.0       193.0 -0.27
## blue_total_jungle_minions_killed       0.0         92.0        92.0  0.12
## blue_gold_diff                    -10830.0      11467.0     22297.0  0.03
## blue_experience_diff               -9333.0       8348.0     17681.0  0.02
## blue_cs_per_min                        9.0         28.3        19.3 -0.27
## blue_gold_per_min                   1073.0       2370.1      1297.1  0.47
## red_wards_placed                       6.0        276.0       270.0  4.56
## red_wards_destroyed                    0.0         24.0        24.0  2.95
## red_first_blood                        0.0          1.0         1.0  0.02
## red_kills                              0.0         22.0        22.0  0.51
## red_deaths                             0.0         22.0        22.0  0.54
## red_assists                            0.0         28.0        28.0  0.82
## red_elite_monsters                     0.0          2.0         2.0  0.62
## red_dragons                            0.0          1.0         1.0  0.35
## red_heralds                            0.0          1.0         1.0  1.85
## red_towers_destroyed                   0.0          2.0         2.0  5.34
## red_total_gold                     11212.0      22732.0     11520.0  0.41
## red_avg_level                          4.8          8.2         3.4 -0.40
## red_total_experience               10465.0      22269.0     11804.0 -0.28
## red_total_minions_killed             107.0        289.0       182.0 -0.29
## red_total_jungle_minions_killed        4.0         92.0        88.0  0.23
## red_gold_diff                     -11467.0      10830.0     22297.0 -0.03
```

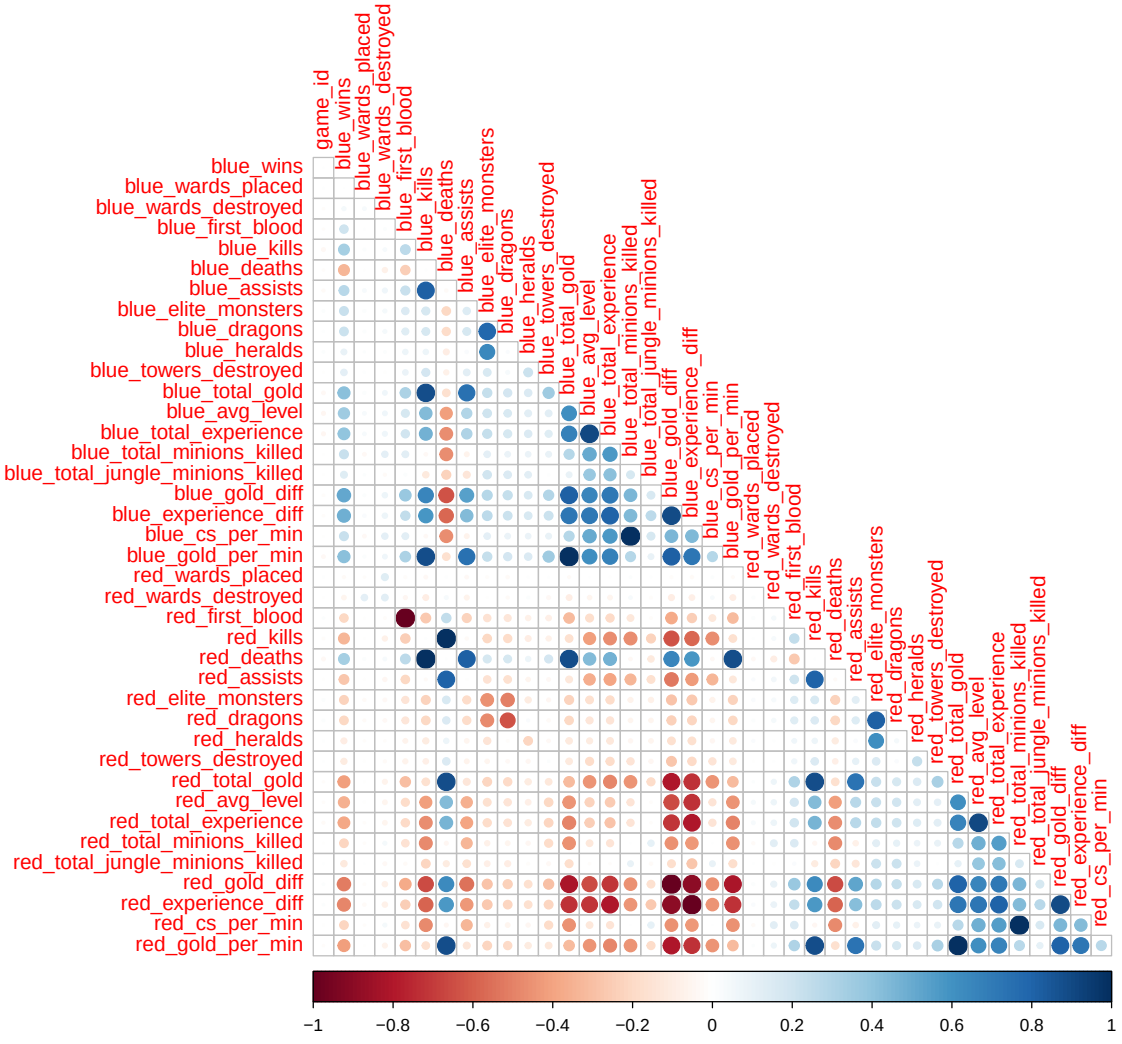
```
## red_experience_diff                   -8348.0       9333.0     17681.0 -0.02
## red_cs_per_min                           10.7         28.9        18.2 -0.29
## red_gold_per_min                       1121.2       2273.2      1152.0  0.41
##                                  kurtosis        se
## game_id                              3.33 277416.26
## blue_wins                           -2.00      0.01
## blue_wards_placed                   23.42      0.18
## blue_wards_destroyed                17.18      0.02
## blue_first_blood                    -2.00      0.01
## blue_kills                           0.26      0.03
## blue_deaths                          0.21      0.03
## blue_assists                         1.16      0.04
## blue_elite_monsters                 -0.50      0.01
## blue_dragons                        -1.67      0.00
## blue_heralds                         0.55      0.00
## blue_towers_destroyed               39.83      0.00
## blue_total_gold                      0.48     15.45
## blue_avg_level                       1.11      0.00
## blue_total_experience                0.68     12.08
## blue_total_minions_killed            0.17      0.22
## blue_total_jungle_minions_killed     0.38      0.10
## blue_gold_diff                       0.30     24.68
## blue_experience_diff                 0.36     19.32
## blue_cs_per_min                      0.17      0.02
## blue_gold_per_min                    0.48      1.54
## red_wards_placed                    30.45      0.19
## red_wards_destroyed                 18.22      0.02
## red_first_blood                     -2.00      0.01
## red_kills                            0.21      0.03
## red_deaths                           0.26      0.03
## red_assists                          0.78      0.04
## red_elite_monsters                  -0.57      0.01
## red_dragons                         -1.88      0.00
## red_heralds                          1.44      0.00
## red_towers_destroyed                30.53      0.00
## red_total_gold                       0.22     15.00
## red_avg_level                        1.23      0.00
## red_total_experience                 0.82     12.06
## red_total_minions_killed             0.23      0.22
## red_total_jungle_minions_killed      0.41      0.10
## red_gold_diff                        0.30     24.68
## red_experience_diff                  0.36     19.32
## red_cs_per_min                       0.23      0.02
## red_gold_per_min                     0.22      1.50
```

Narrow Down Variables

Here we are checking the correlation between all the variables, but we specifically need to pay more attention to what is correlated with blue_wins.

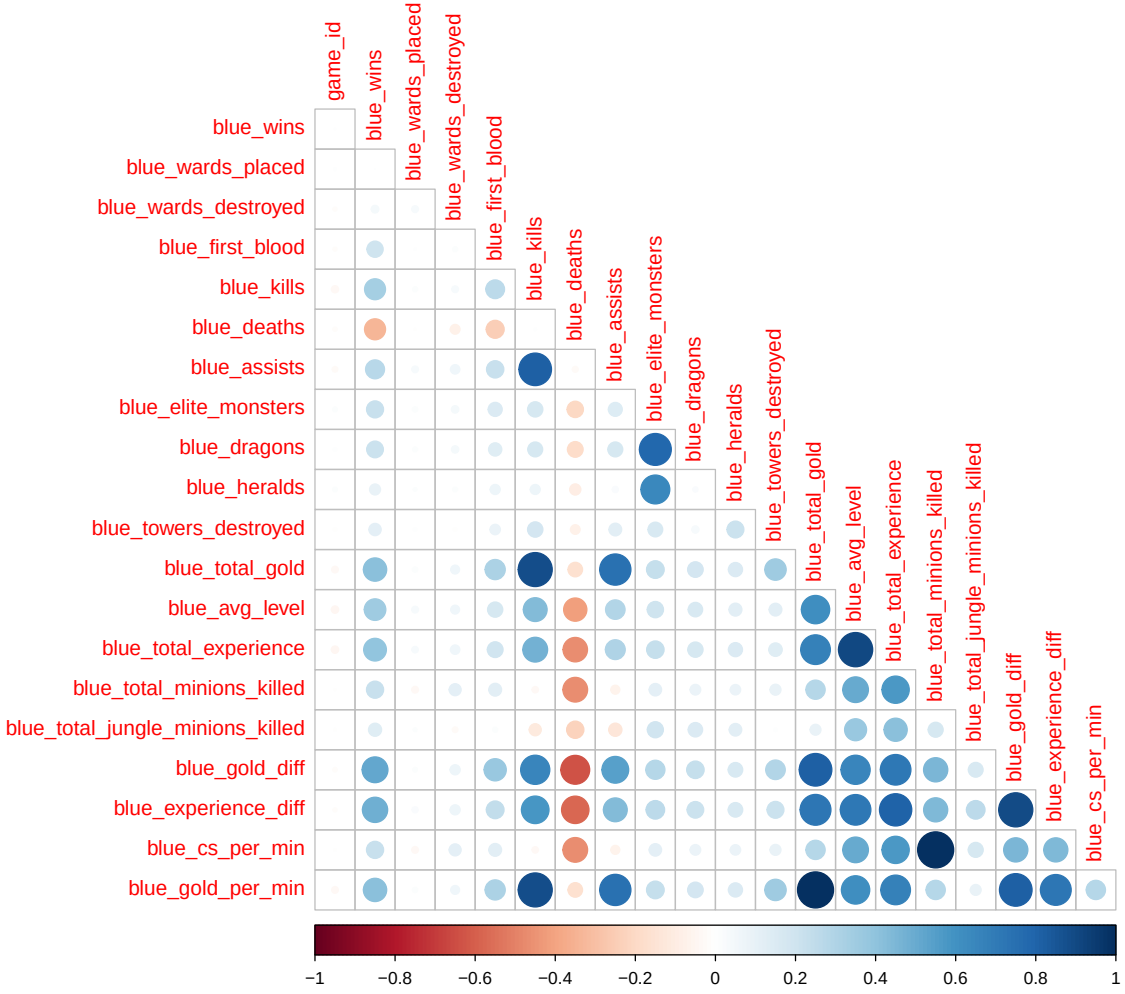
From the result of this auto-plot, we can see there are variables that correlate with blue_wins at about the same scale but in totally opposite ways, which proves our assumption that there are repetitive variables to be correct.

```
lol %>%
  select(is.numeric) %>%
  cor(use = "complete.obs") %>%
  corrplot(type = "lower", diag = FALSE)
```



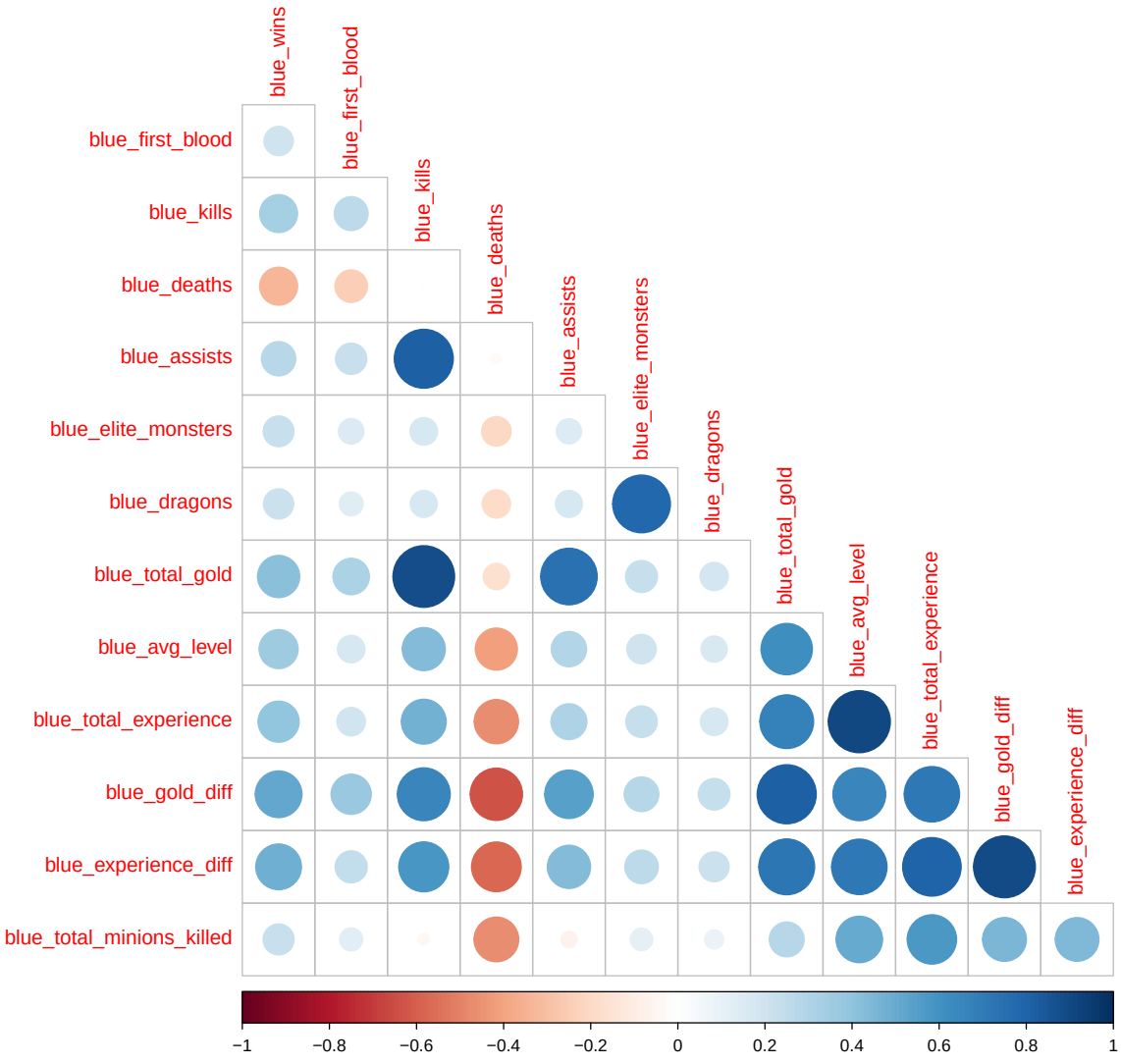
We want to deselect some variables based on the result of the following graph.

```
lol_blue %>%
  select(is.numeric) %>%
  cor(use = "complete.obs") %>%
  corrplot(type = "lower", diag = FALSE)
```



Here is the final correlation chart that only includes important variables towards *blue_wins*.

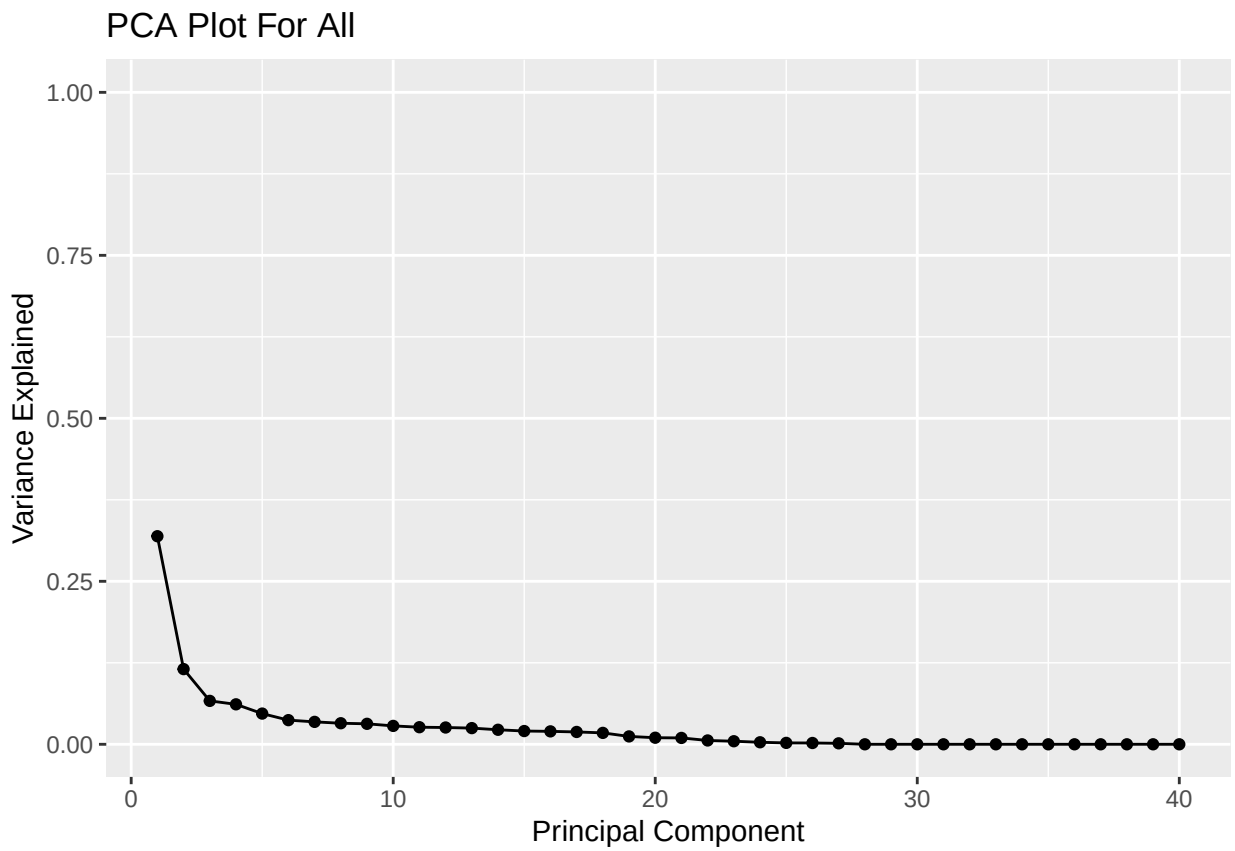
```
important %>%
  select(is.numeric) %>%
  cor(use = "complete.obs") %>%
  corrplot(type = "lower", diag = FALSE)
```



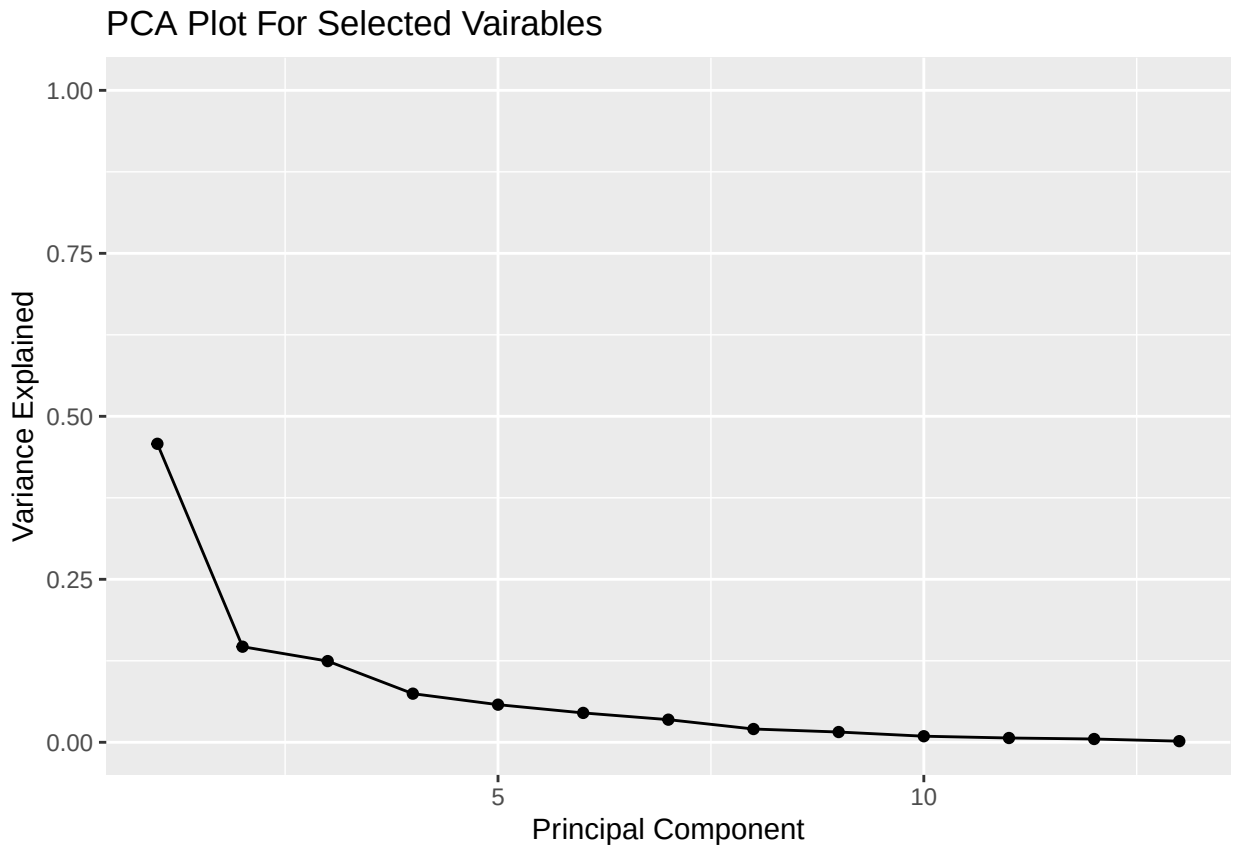
Principal Components Analysis

Although we are doing supervised learning, we can still use Principal Components Analysis (PCA) to see if the variables are strongly related. We are going to do two PCAs. And from the results we are getting. The relationships between variables seems weak.

```
qplot(c(1 : 40), var_explained) +
  geom_line() +
  xlab("Principal Component") +
  ylab("Variance Explained") +
  ggtitle("PCA Plot For All") +
  ylim(0, 1)
```




```
qplot(c(1 : 13), var_explained) +
  geom_line() +
  xlab("Principal Component") +
  ylab("Variance Explained") +
  ggtitle("PCA Plot For Selected Vairables") +
  ylim(0, 1)
```



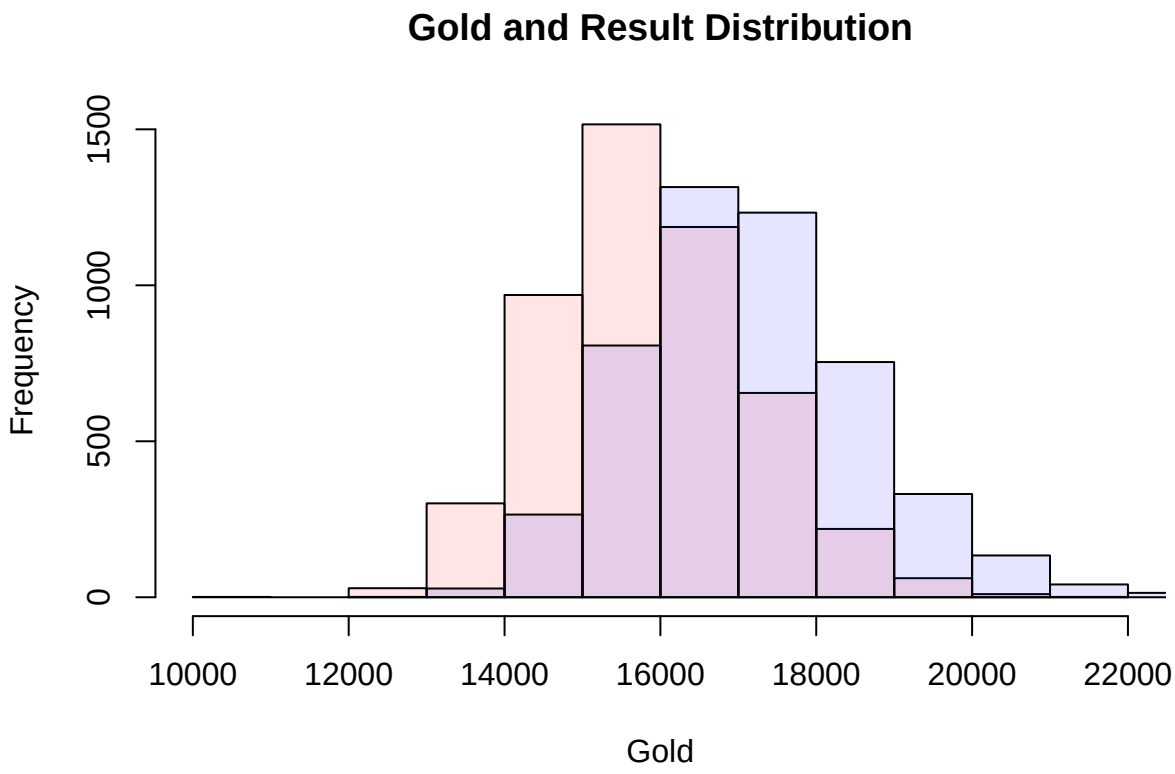
Visualization

From the Previous EDA results we know two of the most important aspects that are highly correlated with *blue_wins* are gold and experience. We want to visualize the distribution difference between winning and losing caused by those two aspects. And for those two aspects we want to do Total, and Difference separately. For the following plots, red represents red wins/ blue loses, blue represents blue wins/red loses, purple is the overlapping portion.

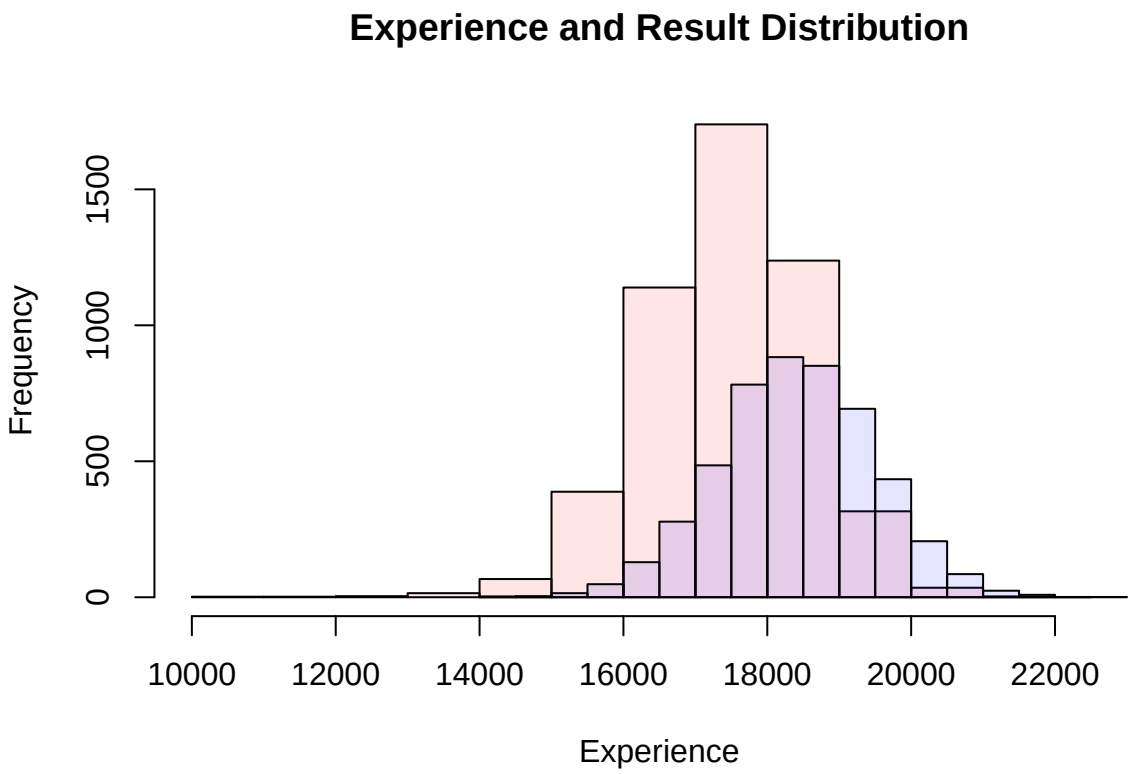
Focus on Total Gold and Experience Below is what the distribution of total gold and experience for the blue team looks like.

Speaking from the results, it looks like winning in general requires more experience and gold.

```
#put them together
plot(lost_gold, col = rgb(1, 0, 0, 0.1),
     main = "Gold and Result Distribution",
     xlab = "Gold")
plot(won_gold, col = rgb(0, 0, 1, 0.1), add = TRUE)
```




```
#put them together
plot(lost_exp, col = rgb(1, 0, 0, 0.1),
     main = "Experience and Result Distribution",
     xlab = "Experience")
plot(won_exp, col = rgb(0, 0, 1, 0.1), add = TRUE)
```

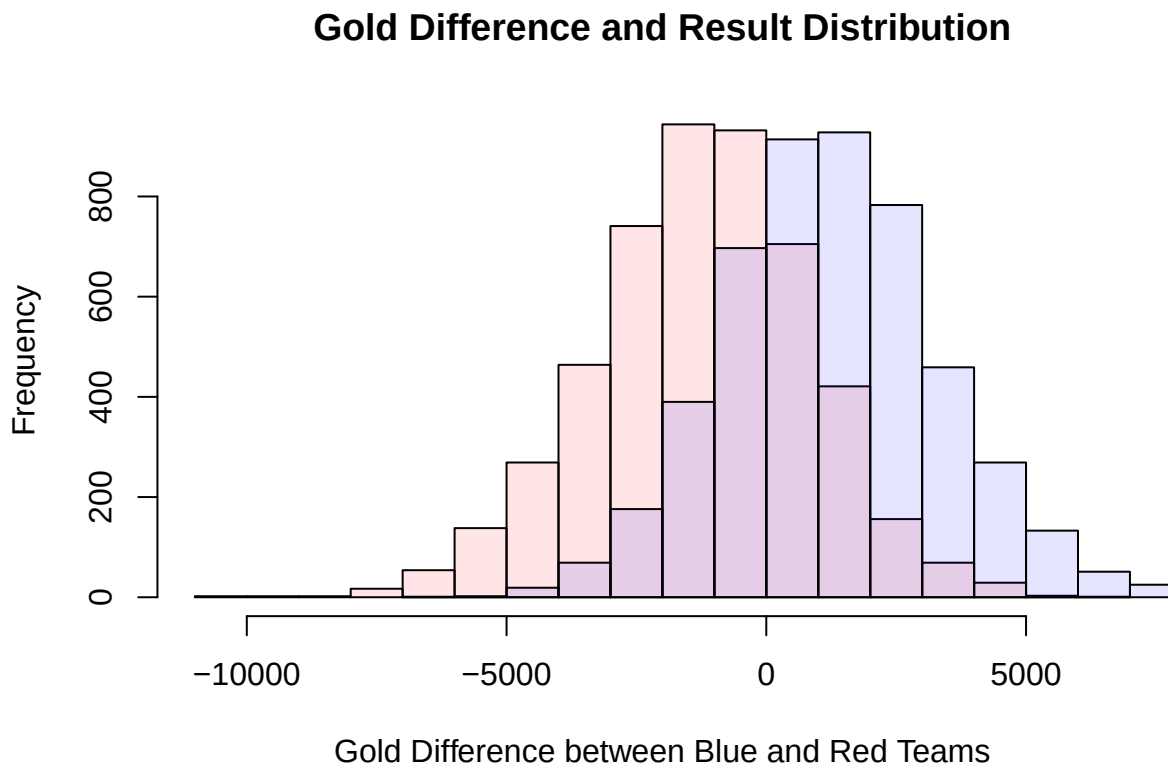


Focus on Gold and Experience Difference

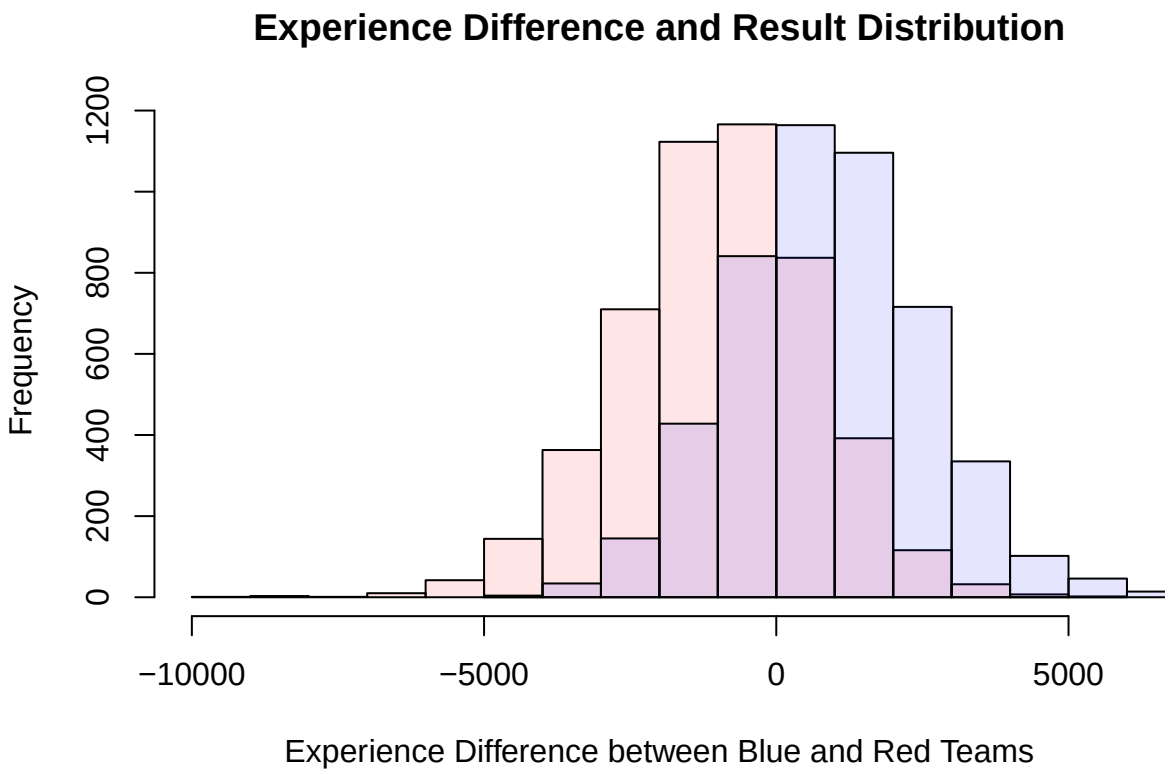
From the above EDA results we can see the distribution of total gold and experience for the blue team looks like. But it does not tell us much about what the opposing team situation. So if we want to see the bigger picture we need to check the difference the two teams have on gold and experience.

From here we can conclude that although having a positive difference in gold and experience between blue and red teams seems to be an important indicator to victory, there is more to it. We still have blue losing when it has more gold and experience than the red team and vice versa.

```
#put them together
plot(lost_gold_diff, col = rgb(1, 0, 0, 0.1),
     main = "Gold Difference and Result Distribution",
     xlab = "Gold Difference between Blue and Red Teams")
plot(won_gold_diff, col = rgb(0, 0, 1, 0.1), add = TRUE)
```




```
#put them together
plot(lost_exp_diff, col = rgb(1, 0, 0, 0.1),
     main = "Experience Difference and Result Distribution",
     xlab = "Experience Difference between Blue and Red Teams")
plot(won_exp_diff, col = rgb(0, 0, 1, 0.1), add = TRUE)
```



Setting Seed and Data Spliting

The year is 2022 so I am setting the seed to be 2022. It is easy to remember, and it is not too small.

The data is split with a 75% training, 25% testing split. Stratified with blue_wins.

We want to factor *blue_wins* and *blue_first_blood*, because they represent yes and no towards if blue team won and if blue team got the first blood, they do not really represent any numeric values.

```
set.seed(2022)

important <- important  %>%
  mutate(blue_wins = factor(blue_wins,
                            levels = c(0, 1)),
         blue_first_blood = factor(blue_first_blood),
         )

lol_split <- important %>%
  initial_split(strata = blue_wins, prop = 0.75)
lol_train <- training(lol_split)
lol_test <- testing(lol_split)

dim(lol_train) # check if we have the right proportion
```

```
## [1] 7408   13
```

Modeling

Now we are going to start model building and fitting.

First we need to create a recipe for these model. We have no missing data so we do not need step_impute_linear in this case, if we do we will need it. Since we have factor variables we need to use step_dummy to make them into numeric variables. And we use step_normalize to center and scale our now numeric data.

```
lol_recipe <- recipe(blue_wins ~ ., data = lol_train) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_normalize(all_predictors())
```

We now use k-fold cross-validation, with k = 5. Ideally we want to use 10, but it takes a lot of time.

```
lol_folds <- vfold_cv(lol_train, v = 5, strata = 'blue_wins')
lol_folds
```

```
## #  5-fold cross-validation using stratification
## # A tibble: 5 x 2
##   splits               id
##   <list>               <chr>
## 1 <split [5925/1483]> Fold1
## 2 <split [5926/1482]> Fold2
## 3 <split [5927/1481]> Fold3
## 4 <split [5927/1481]> Fold4
## 5 <split [5927/1481]> Fold5
```

Now we set up control for lda and qda models

```
control <- control_resamples(save_pred = TRUE)
```

Logistic Regression

Making a model, a workflow and a fit for logistic regression with the glm engine. Since we are classifying if the team win or lose the mode is classification, which applies to other models we are going to build later as well.

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

log_wkflow <- workflow() %>%
  add_model(log_reg) %>%
  add_recipe(lol_recipe)

log_fit <- fit_resamples(log_wkflow, lol_folds)
```

check our result

```
collect_metrics(log_fit)
```

```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.730     5 0.00283 Preprocessor1_Model1
## 2 roc_auc  binary     0.807     5 0.00414 Preprocessor1_Model1
```

Linear Discriminant Analysis

Same steps as the previous model. We technically can use the same recipe as the previous model. We can use the same folds as before so no need to create new folds.

Making a model, a workflow and a fit for Linear Discriminant Analysis, this time we are using the MASS engine.

```
lda_mod <- discrim_linear() %>%
  set_engine("MASS") %>%
  set_mode("classification")

lda_wkflow <- workflow() %>%
  add_recipe(lol_recipe ) %>%
  add_model(lda_mod)

lda_fit <- fit_resamples(resamples = lol_folds,
                         lda_wkflow,
                         control = control)
```

check our result

```
collect_metrics(lda_fit)
```

```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.729     5 0.00256 Preprocessor1_Model1
## 2 roc_auc  binary     0.807     5 0.00437 Preprocessor1_Model1
```

Quadratic Discriminant Analysis

Same steps as the previous model. Using the same recipe, folds and control.

Making a model, a workflow and a fit for Linear Discriminant Analysis, This time we are also using the mass engine, but we use a different function, instead of discrim_linear(), we use discrim_quad().

```
qda_mod <- discrim_quad() %>%
  set_engine("MASS") %>%
  set_mode("classification")

qda_wkflow <- workflow() %>%
  add_recipe(lol_recipe ) %>%
```

```
  add_model(qda_mod)

qda_fit <- fit_resamples(resamples = lol_folds,
                         qda_wkflow,
                         control = control)
```

check our result

```
collect_metrics(qda_fit)
```

```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.712     5 0.00525 Preprocessor1_Model1
## 2 roc_auc  binary     0.785     5 0.00475 Preprocessor1_Model1
```

Now let us gather who did the best on the folds so far

```
collect_metrics(log_fit)
```

```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.730     5 0.00283 Preprocessor1_Model1
## 2 roc_auc  binary     0.807     5 0.00414 Preprocessor1_Model1
```

```
collect_metrics(lda_fit)
```

```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.729     5 0.00256 Preprocessor1_Model1
## 2 roc_auc  binary     0.807     5 0.00437 Preprocessor1_Model1
```

```
collect_metrics(qda_fit)
```

```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.712     5 0.00525 Preprocessor1_Model1
## 2 roc_auc  binary     0.785     5 0.00475 Preprocessor1_Model1
```

```
log_best_roc_auc <- collect_metrics(log_fit) %>%
  slice(2) %>%
  pull(mean)
log_best_roc_auc
```

```
## [1] 0.8073058
```

Let's see how the best performing model so far works on the testing set.

It is doing a little better than the training set, we were not over fitting, which is good. But it is not doing much better than the training set. We are going to work on more models to see if there are better models out there at predicting the results.

```
log_test <- fit(log_wkflow, lol_test)
predict(log_test, new_data = lol_test, type = "class") %>%
  bind_cols(lol_test %>% select(blue_wins)) %>%
  accuracy(truth = blue_wins, estimate = .pred_class)
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.732
```

Decision Tree

First we do the same thing, set engine, mode, and workflow.

```
tree_spec <- decision_tree() %>%
  set_engine("rpart")

class_tree_spec <- tree_spec %>%
  set_mode("classification")
```

```
class_tree_wf <- workflow() %>%
  add_model(class_tree_spec %>% set_args(cost_complexity = tune())) %>%
  add_recipe(lol_recipe)
```

Then we start setting up our grid, and re-sampling.

```
param_grid <- grid_regular(cost_complexity(range = c(-5, -1)), levels = 5)
```

```
tune_res <- tune_grid(
  class_tree_wf,
  resamples = lol_folds,
  grid = param_grid,
  metrics = metric_set(roc_auc)
)
```

To save time knitting, I used write_rds to save the result we got. Now we just need to read it.

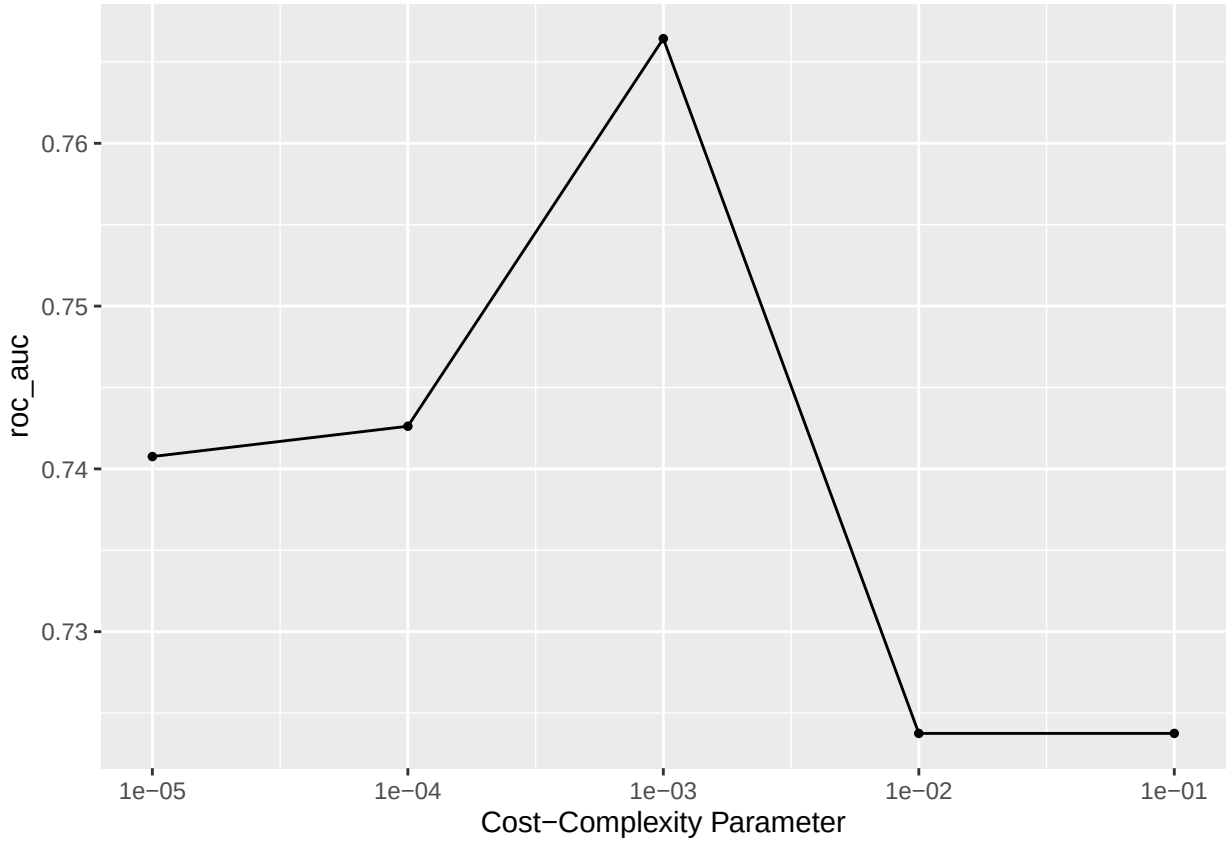
```
decision_tree <- read_rds("model_results/dt_tune.rds")
decision_tree
```

```
## # Tuning results
## # 5-fold cross-validation using stratification
## # A tibble: 5 x 4
##   splits             id    .metrics         .notes
##   <list>             <chr> <list>           <list>
```

```
## 1 <split [5925/1483]> Fold1 <tibble [5 x 5]> <tibble [0 x 1]>
## 2 <split [5926/1482]> Fold2 <tibble [5 x 5]> <tibble [0 x 1]>
## 3 <split [5927/1481]> Fold3 <tibble [5 x 5]> <tibble [0 x 1]>
## 4 <split [5927/1481]> Fold4 <tibble [5 x 5]> <tibble [0 x 1]>
## 5 <split [5927/1481]> Fold5 <tibble [5 x 5]> <tibble [0 x 1]>
```

Here is an plot presenting us the relationship between roc_auc and the cost complicity parameter. ROC_AUC peaks around 1eˆ-3.

```
autoplot(decision_tree)
```



Now we are collecting the best performing pruned tree, and complexity

```
collection1 <- collect_metrics(decision_tree) %>% arrange(desc(mean))
collection1
```

```
## # A tibble: 5 x 7
##   cost_complexity .metric .estimator  mean     n std_err .config
##             <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1         0.001   roc_auc binary     0.766     5 0.00400 Preprocessor1_Model3
## 2         0.0001  roc_auc binary     0.743     5 0.00237 Preprocessor1_Model2
## 3         0.00001 roc_auc binary     0.741     5 0.00302 Preprocessor1_Model1
## 4         0.01    roc_auc binary     0.724     5 0.00599 Preprocessor1_Model4
## 5         0.1     roc_auc binary     0.724     5 0.00599 Preprocessor1_Model5
```

```
best_pruned <- select_best(decision_tree, metric = "roc_auc")
best_pruned
```

```
## # A tibble: 1 x 2
##   cost_complexity .config
##             <dbl> <chr>
## 1           0.001 Preprocessor1_Model3
```

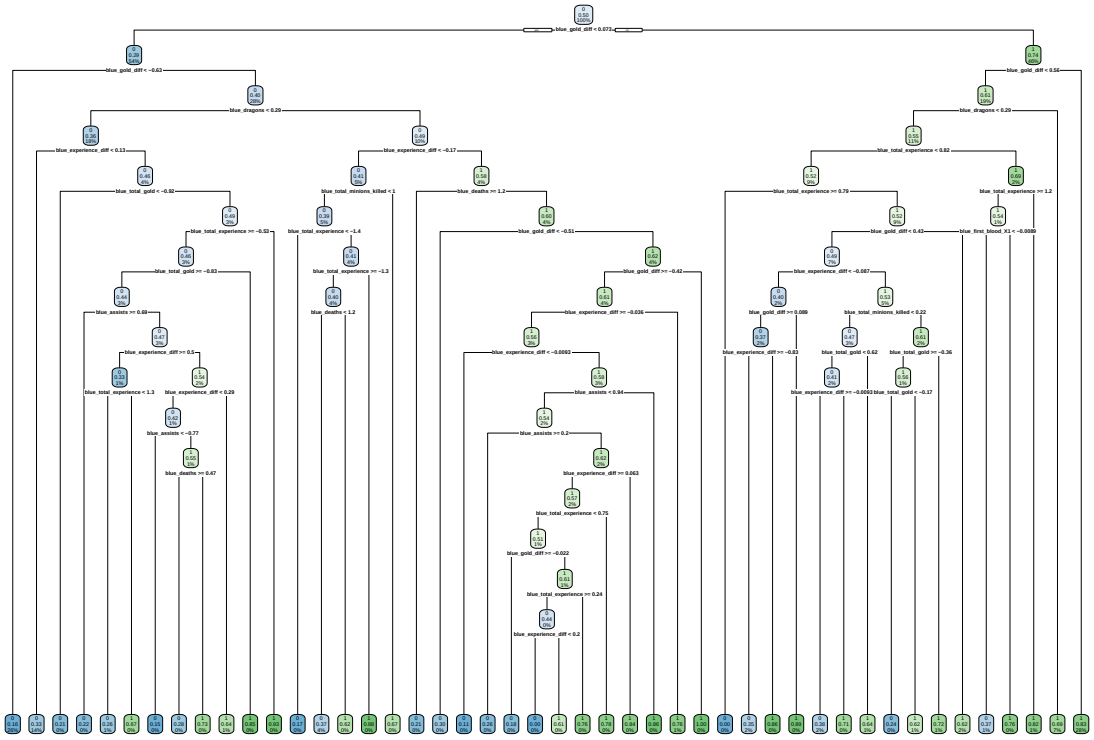
```
tree_best_roc_auc <- collection1 %>%
  slice(1) %>%
  pull(mean)
tree_best_roc_auc
```

```
## [1] 0.766432
```

```
best_complexity <- select_best(decision_tree)
class_tree_final <- finalize_workflow(class_tree_wf, best_complexity)
class_tree_final_fit <- fit(class_tree_final, data = lol_train)
```

This is what our decision tree looks like.

```
class_tree_final_fit %>%
  extract_fit_engine() %>%
  rpart.plot()
```



Random Forest

First we do the same thing, set engine, mode, and workflow. We are tuning our parameters, setting our engine to be ranger and importance to be impurity.

```
forest_spec <- rand_forest(
  mode = "classification",
  mtry = tune(),
  trees = tune(),
  min_n = tune()
)%>%
  set_engine("ranger", importance = "impurity")

forest_wf <- workflow() %>%
  add_model(forest_spec %>%
              set_args(mtry = tune(), trees = tune(),
                       min_n = tune()
                       )
            ) %>%
  add_recipe(lol_recipe)
```

Then we set up the grid.

```
forest_grid <- grid_regular(mtry(range = c(1, 12)),
                            trees(range = c(1, 10)),
                            min_n(range = c(1, 10)),
                            levels = 23)
forest_grid
```

```
## # A tibble: 1,200 x 3
##     mtry trees min_n
##    <int> <int> <int>
##  1     1     1     1
##  2     2     1     1
##  3     3     1     1
##  4     4     1     1
##  5     5     1     1
##  6     6     1     1
##  7     7     1     1
##  8     8     1     1
##  9     9     1     1
## 10    10     1     1
## # ... with 1,190 more rows
```

Then we tune the model and repeated cross fold validation

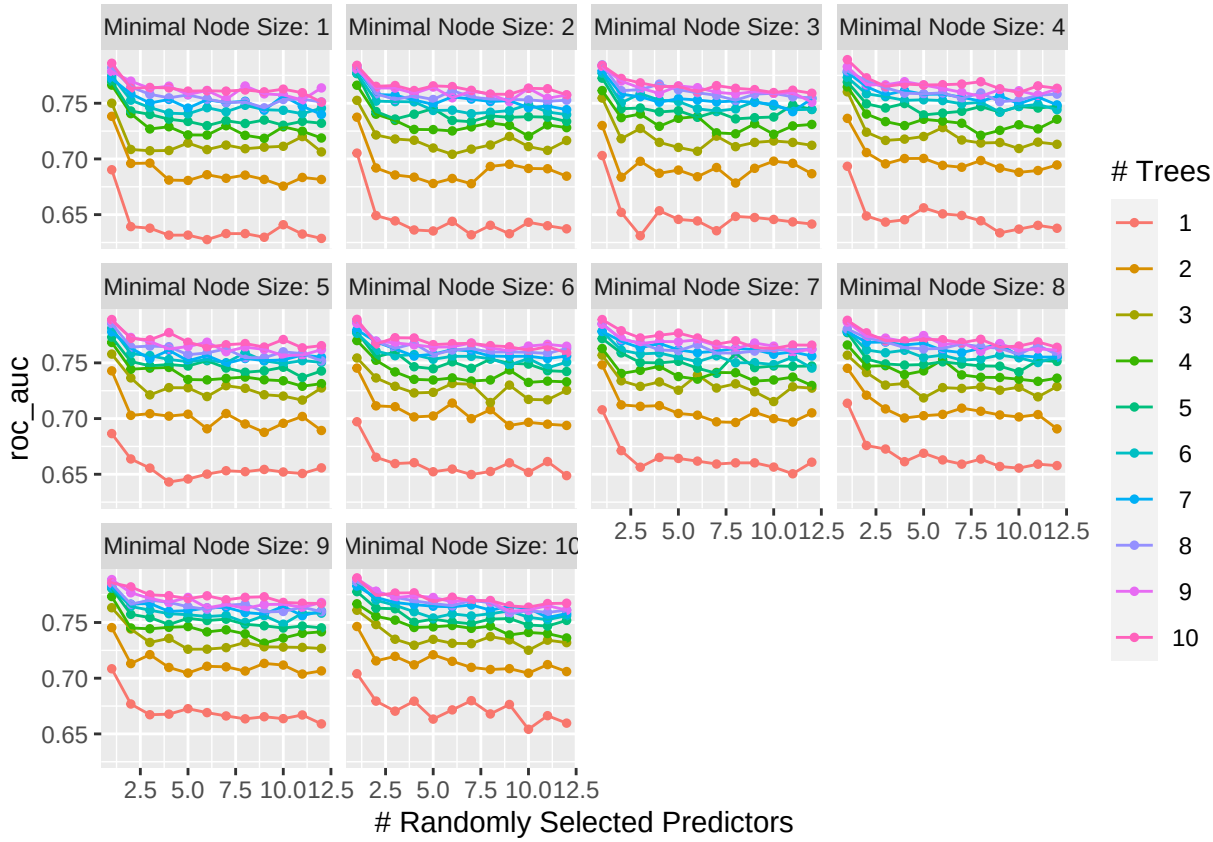
```
forest_tune_res <- tune_grid(
  forest_wf,
  resamples = lol_folds,
  grid = forest_grid,
  metrics = metric_set(roc_auc)
)
```

We load our results here

```
forest <- read_rds("model_results/forest_tune.rds")
```

From the result it seems our model works better with less randomly selected predictors and more nodes and trees.

```
autoplot(forest)
```



Then we collect the best performing forest, and present its roc_auc value.

```
collection2 <- collect_metrics(forest) %>%
  arrange(desc(mean))

collection2
```

```
## # A tibble: 1,200 x 9
##    mtry trees min_n .metric .estimator  mean     n std_err .config
##   <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1     1    10    10 roc_auc binary     0.790     5 0.00573 Preprocessor1_Model~
## 2     1     9    10 roc_auc binary     0.789     5 0.00492 Preprocessor1_Model~
## 3     1    10     4 roc_auc binary     0.789     5 0.00508 Preprocessor1_Model~
## 4     1    10     6 roc_auc binary     0.789     5 0.00446 Preprocessor1_Model~
## 5     1    10     5 roc_auc binary     0.789     5 0.00632 Preprocessor1_Model~
## 6     1    10     7 roc_auc binary     0.789     5 0.00588 Preprocessor1_Model~
```

```
##  7     1     9     9 roc_auc binary     0.789     5 0.00443 Preprocessor1_Model~
##  8     1    10     8 roc_auc binary     0.788     5 0.00489 Preprocessor1_Model~
##  9     1     9     5 roc_auc binary     0.788     5 0.00769 Preprocessor1_Model~
## 10     1     8     6 roc_auc binary     0.787     5 0.00491 Preprocessor1_Model~
## # ... with 1,190 more rows
```

```
best_forest <- select_best(forest, metric = "roc_auc")
best_forest
```

```
## # A tibble: 1 x 4
##    mtry trees min_n .config
##   <int> <int> <int> <chr>
## 1     1    10    10 Preprocessor1_Model1189
```

```
forest_best_roc_auc <- collection2 %>%
  slice(1) %>%
  pull(mean)

forest_best_roc_auc
```

```
## [1] 0.790079
```

Boosted Tree

First we do the same thing, set engine, mode, and workflow. This time we are using the xgboost engine, and we also tune our parameters this time.

```
boost_spec <- boost_tree(
  mode = "classification",
  engine = "xgboost",
  trees = tune(),
)
boost_wf <- workflow() %>%
  add_model(boost_spec %>%
              set_args(trees = tune()
                       )
            ) %>%
  add_recipe(lol_recipe)
```

Then we set up the grid.

```
boost_grid <- grid_regular(trees(range = c(1, 200)),
                           levels = 10)
boost_grid
```

```
## # A tibble: 10 x 1
##    trees
##    <int>
##  1     1
##  2    23
##  3    45
```

```
##  4    67
##  5    89
##  6   111
##  7   133
##  8   155
##  9   177
## 10   200
```

Then we tune the model and repeated cross fold validation

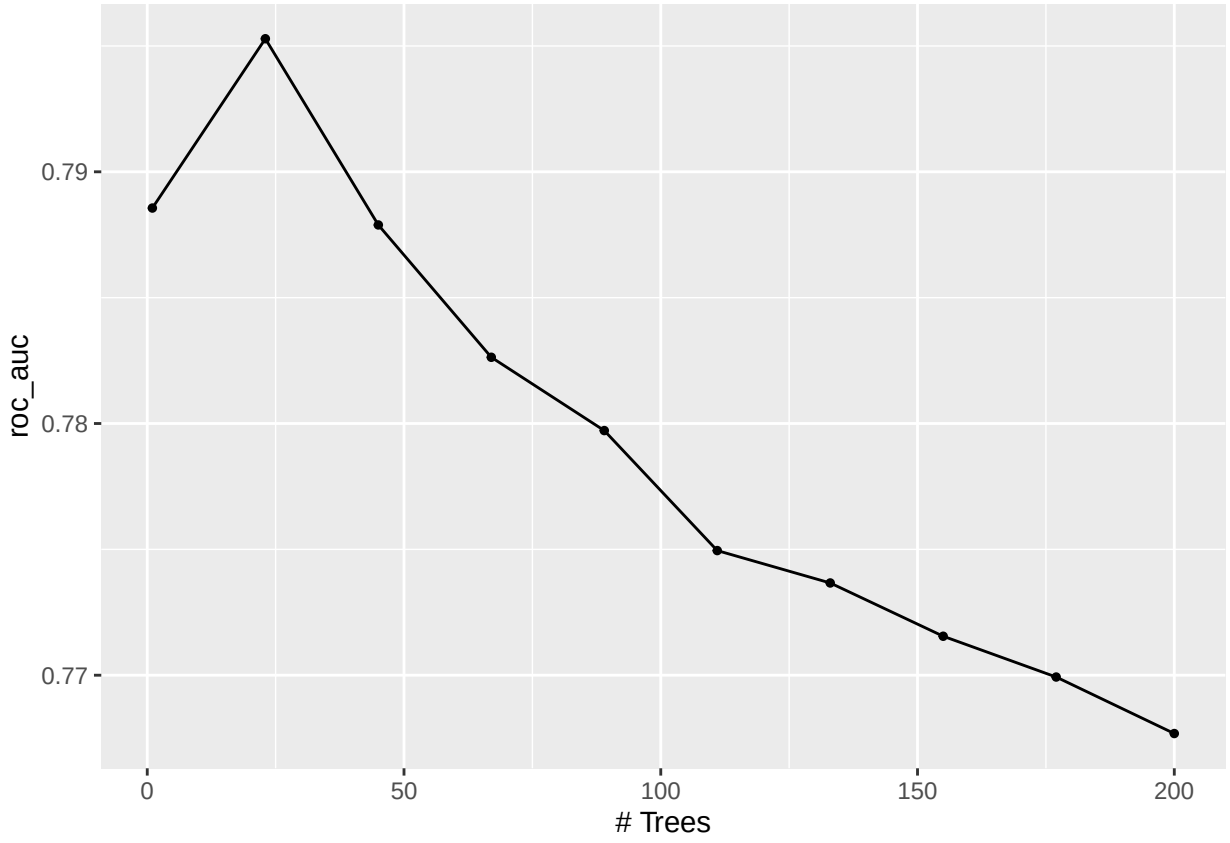
```
boost_tune_res <- tune_grid(
  boost_wf,
  resamples = pokemon_folds,
  grid = boost_grid,
  metrics = metric_set(roc_auc)
)
```

We load our results here

```
boosted <- read_rds("model_results/bt_tune.rds")
```

From the result it seems this model works better with less trees.

```
autoplot(boosted)
```



Then we collect the best performing forest, and present its roc_auc value.

```
collection3 <- collect_metrics(boosted) %>% arrange(desc(mean))
collection3
```

```
## # A tibble: 10 x 7
##    trees .metric .estimator  mean     n std_err .config
##    <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1    23 roc_auc binary     0.795     5 0.00386 Preprocessor1_Model02
##  2     1 roc_auc binary     0.789     5 0.00468 Preprocessor1_Model01
##  3    45 roc_auc binary     0.788     5 0.00564 Preprocessor1_Model03
##  4    67 roc_auc binary     0.783     5 0.00619 Preprocessor1_Model04
##  5    89 roc_auc binary     0.780     5 0.00559 Preprocessor1_Model05
##  6   111 roc_auc binary     0.775     5 0.00553 Preprocessor1_Model06
##  7   133 roc_auc binary     0.774     5 0.00482 Preprocessor1_Model07
##  8   155 roc_auc binary     0.772     5 0.00497 Preprocessor1_Model08
##  9   177 roc_auc binary     0.770     5 0.00488 Preprocessor1_Model09
## 10   200 roc_auc binary     0.768     5 0.00460 Preprocessor1_Model10
```

```
best_boost <- select_best(boosted, metric = "roc_auc")
best_boost
```

```
## # A tibble: 1 x 2
##   trees .config
##   <int> <chr>
## 1    23 Preprocessor1_Model02
```

```
boost_best_roc_auc <- collection3 %>%
  slice(1) %>%
  pull(mean)
boost_best_roc_auc
```

```
## [1] 0.795287
```

Elastic Net- Lasso

First we do the same thing, set engine, mode, and workflow. This time we are tuning penalty and mixture using multinom_reg with the glmnet engine.

```
elastic_net_spec <- multinom_reg(penalty = tune(),
                                 mixture = tune()) %>%
  set_mode("classification") %>%
  set_engine("glmnet")

en_workflow <- workflow() %>%
  add_recipe(lol_recipe) %>%
  add_model(elastic_net_spec)

en_grid <- grid_regular(penalty(range = c(-5, 5)),
                        mixture(range = c(0, 1)), levels = 12)
```

We tune our grids, and re-sample. Essentially doing folded cross validation again but with a different model.

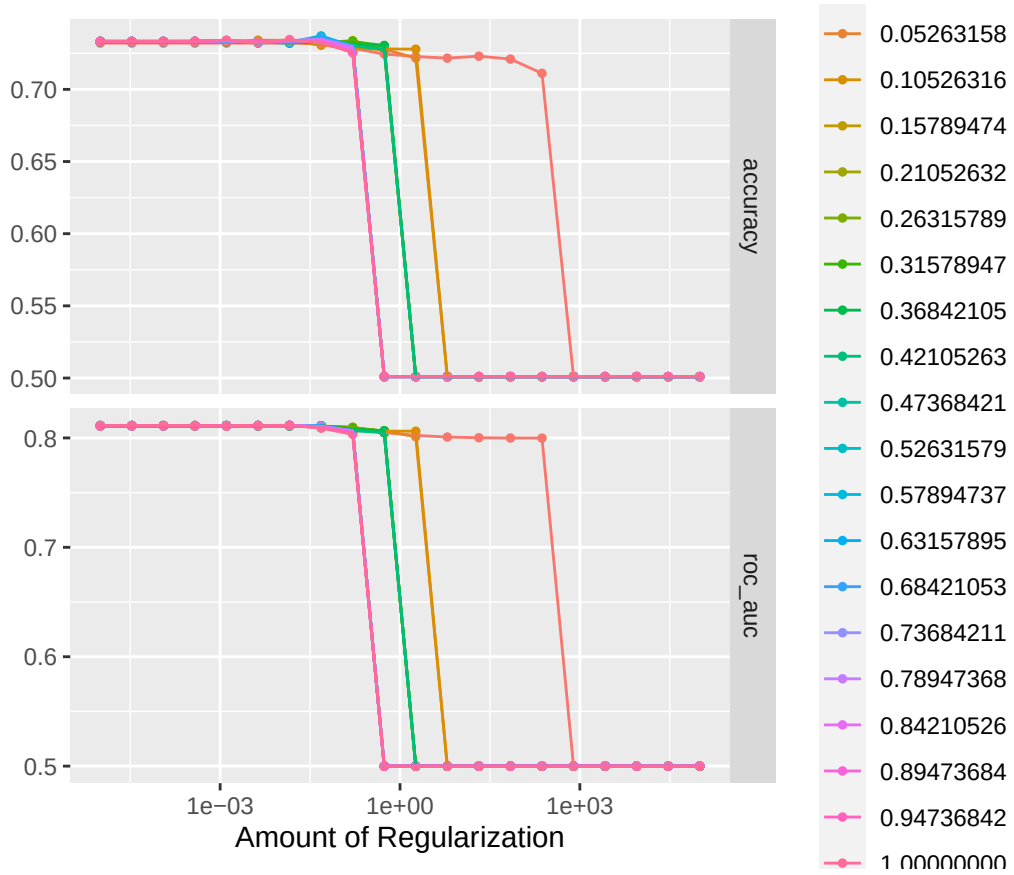
```
tune_res <- tune_grid(
  en_workflow,
  resamples = lol_folds,
  grid = en_grid
)
```

Now we read the results we saved in rds.

```
elastic <- read_rds("model_results/en.rds")
```

It looks like the model does well with low regulation and high penalty.

```
autoplot(elastic )
```



Then we collect the best performing forest, and present its roc_auc value.

```
collection4 <- collect_metrics(elastic) %>% arrange(desc(mean))
collection4
```

```
## # A tibble: 800 x 8
##    penalty mixture .metric .estimator  mean     n std_err .config
##      <dbl>   <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1 0.0144    1     roc_auc binary     0.811     5 0.00393 Preprocessor1_Model387
##  2 0.0144    0.947 roc_auc binary     0.811     5 0.00395 Preprocessor1_Model367
```

```
##  3 0.0144    0.842 roc_auc binary     0.811     5 0.00398 Preprocessor1_Model327
##  4 0.0144    0.789 roc_auc binary     0.811     5 0.00399 Preprocessor1_Model307
##  5 0.0144    0.895 roc_auc binary     0.811     5 0.00396 Preprocessor1_Model347
##  6 0.0144    0.737 roc_auc binary     0.811     5 0.00399 Preprocessor1_Model287
##  7 0.0144    0.684 roc_auc binary     0.811     5 0.00400 Preprocessor1_Model267
##  8 0.00428   1     roc_auc binary     0.811     5 0.00397 Preprocessor1_Model386
##  9 0.00428   0.947 roc_auc binary     0.811     5 0.00397 Preprocessor1_Model366
## 10 0.0144    0.632 roc_auc binary     0.811     5 0.00400 Preprocessor1_Model247
## # ... with 790 more rows
```

```
best_en_model <- select_best(elastic, metric = "roc_auc")
best_en_model
```

```
## # A tibble: 1 x 3
##   penalty mixture .config
##     <dbl>   <dbl> <chr>
## 1  0.0144       1 Preprocessor1_Model387
```

```
en_best_roc_auc <- collection4 %>%
  slice(1) %>%
  pull(mean)
en_best_roc_auc
```

```
## [1] 0.811487
```

Now Let Us Compare All the Models.

We are now comparing the best performing fold from each model, and choosing the best performing model. Since we already know LDA and QDA did not out perform Logistic regression model. That left us with the logistic regression, decision tree, random forest, boosted tree and elastic net models.

We are making a ROC_AUC score table to compare them.

Our Current best performing model is the Elastic Net Tunneling Lasso model.

```
roc_auc_table <- matrix(c("Logistic Regression", "Decision Tree",
                          "Random forest", "Boosted Tree", "Elastic Net",
                          log_best_roc_auc, tree_best_roc_auc,
                          forest_best_roc_auc, boost_best_roc_auc,
                          en_best_roc_auc), ncol = 5, byrow = TRUE)
roc_auc_table
```

```
##      [,1]                 [,2]               [,3]
## [1,] "Logistic Regression" "Decision Tree"    "Random forest"
## [2,] "0.807305823311029"   "0.766432049836951" "0.790079046644655"
##      [,4]               [,5]
## [1,] "Boosted Tree"     "Elastic Net"
## [2,] "0.795287020698528" "0.811487010787354"
```

Fitting the Best Model on Trainning and Test Sets

Now we want to see how fitting the optimal tuned model to training and test sets looks like.

The training data gives us a ROC_AUC score of 0.8119945, and the testing set gives us a ROC_AUC score of 0.8050239. ROC_AUC scores are capped at 1, so we are doing pretty good, expect since our score is lower for the testing set, we probably over fitted in the process.

```
best_model <- select_best(elastic, metric = "roc_auc")
final <- finalize_workflow(en_workflow, best_model)

fit_final_train <- fit(final, data = lol_train)
fit_final_train
```

```
## == Workflow [trained] ==========================================================
## Preprocessor: Recipe
## Model: multinom_reg()
##
## -- Preprocessor ----------------------------------------------------------------
## 2 Recipe Steps
##
## * step_dummy()
## * step_normalize()
##
## -- Model -----------------------------------------------------------------------
##
## Call:  glmnet::glmnet(x = maybe_matrix(x), y = y, family = "multinomial",      alpha = ~1)
##
##    Df  %Dev   Lambda
## 1   0  0.00 0.253500
## 2   1  3.15 0.231000
## 3   1  5.81 0.210400
## 4   1  8.06 0.191700
## 5   1  9.98 0.174700
## 6   1 11.63 0.159200
## 7   1 13.05 0.145000
## 8   2 14.33 0.132200
## 9   2 15.45 0.120400
## 10  2 16.41 0.109700
## 11  2 17.25 0.099980
## 12  2 17.97 0.091090
## 13  2 18.59 0.083000
## 14  2 19.13 0.075630
## 15  2 19.59 0.068910
## 16  3 20.06 0.062790
## 17  3 20.53 0.057210
## 18  3 20.94 0.052130
## 19  3 21.29 0.047500
## 20  3 21.59 0.043280
## 21  3 21.85 0.039430
## 22  3 22.07 0.035930
## 23  3 22.25 0.032740
## 24  3 22.41 0.029830
## 25  3 22.54 0.027180
```

```
## 26  3 22.65 0.024760
## 27  3 22.75 0.022560
## 28  3 22.83 0.020560
## 29  3 22.90 0.018730
## 30  3 22.95 0.017070
## 31  3 23.00 0.015550
## 32  3 23.04 0.014170
## 33  3 23.08 0.012910
## 34  3 23.10 0.011770
## 35  4 23.13 0.010720
## 36  4 23.15 0.009768
## 37  4 23.17 0.008900
## 38  6 23.18 0.008109
## 39  6 23.20 0.007389
## 40  6 23.21 0.006732
## 41  6 23.22 0.006134
## 42  6 23.23 0.005589
## 43  7 23.24 0.005093
## 44  7 23.25 0.004640
## 45  7 23.25 0.004228
## 46  7 23.26 0.003853
##
## ...
## and 27 more lines.
```

```
predicted_data_train <- augment(fit_final_train, new_data = lol_train) %>%
  select(blue_wins, starts_with(".pred"))

predicted_data_train %>% roc_auc(blue_wins, .pred_0)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc binary         0.809
```

```
fit_final_test <- fit(final, data = lol_test)
fit_final_test
```

```
## == Workflow [trained] ==========================================================
## Preprocessor: Recipe
## Model: multinom_reg()
##
## -- Preprocessor ----------------------------------------------------------------
## 2 Recipe Steps
##
## * step_dummy()
## * step_normalize()
##
## -- Model -----------------------------------------------------------------------
##
## Call:  glmnet::glmnet(x = maybe_matrix(x), y = y, family = "multinomial",      alpha = ~1)
##
##    Df  %Dev   Lambda
```

```
## 1   0  0.00 0.261800
## 2   1  3.37 0.238500
## 3   1  6.20 0.217400
## 4   1  8.60 0.198000
## 5   2 10.68 0.180500
## 6   2 12.54 0.164400
## 7   2 14.15 0.149800
## 8   2 15.54 0.136500
## 9   2 16.74 0.124400
## 10  2 17.79 0.113300
## 11  2 18.70 0.103300
## 12  2 19.48 0.094090
## 13  2 20.16 0.085730
## 14  2 20.75 0.078110
## 15  2 21.26 0.071170
## 16  2 21.70 0.064850
## 17  2 22.07 0.059090
## 18  2 22.40 0.053840
## 19  2 22.67 0.049060
## 20  3 22.99 0.044700
## 21  4 23.27 0.040730
## 22  4 23.51 0.037110
## 23  4 23.72 0.033810
## 24  4 23.90 0.030810
## 25  4 24.05 0.028070
## 26  4 24.17 0.025580
## 27  4 24.28 0.023310
## 28  4 24.37 0.021240
## 29  6 24.46 0.019350
## 30  6 24.53 0.017630
## 31  6 24.60 0.016060
## 32  6 24.66 0.014640
## 33  6 24.70 0.013340
## 34  6 24.74 0.012150
## 35  6 24.78 0.011070
## 36  6 24.80 0.010090
## 37  6 24.83 0.009193
## 38  6 24.85 0.008376
## 39  8 24.87 0.007632
## 40  8 24.90 0.006954
## 41  8 24.93 0.006336
## 42  8 24.95 0.005773
## 43  8 24.97 0.005260
## 44  8 24.98 0.004793
## 45  8 25.00 0.004367
## 46  8 25.01 0.003979
##
## ...
## and 22 more lines.
```

```
predicted_data_test <- augment(fit_final_train, new_data = lol_test) %>%
  select(blue_wins, starts_with(".pred"))

predicted_data_test %>% roc_auc(blue_wins, .pred_0)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc binary         0.814
```

Conclusion

As we discovered in the process of making this project, we learned that the gold and experience difference are the main predictors in predicting the outcome of a diamond ranked solo LOL rank match. Other variables like how many minions they killed are reflected through gold and experience difference. Majority of the variables do not directly influence the result of the matches. But they will contribute in other aspects since they contribute to gold and experience.

It does not matter who gets the first blood, if we want to win a solo rank in diamond rank, we need to maintain calm and try to earn as much money as possible in the first 10 minutes, and as we have more money we are more capable of leveling up our weapons and kill more enemies and earn more money, in a good cycle. Speaking from experience and our EDA if it is a close match, and you want to win, try your best to kill the elite monsters, not the towers.

Now if we are given a set of data for the first ten minutes of a diamond ranked match, we have a close to 80% accuracy to know if the team won the match or not with our Elastic Net tuned Lasso model. Now have fun trying to convince your friends, they will probably hanging on that 20% which is fair, anyway good luck.