Predicting the Outcome of a League of Legends Game @ 10 Minutes

Kavin Indirajith

2023-03-19

Contents

Introduction

The purpose of this project is to develop a model that will predict the outcome of a League of Legends game.

What is League of Legends?

League of Legends, developed by Riot Games, is the most widely played multiplayer online battle arena in the world, with over 150 million active players monthly. Essentially, the game is a team-based strategy game where two teams of five powerful champions face off to destroy the other's base, also known as the Nexus. There are 3 lanes, a jungle, and 5 roles. The players get to choose from over 150 champions to play in game. A core aspect of this game is competitive ranked gameplay. The ranked game mode is where you are matched with players of similar skill level and play games to climb a ranked ladder. The ranked ladder from lowest to highest is Iron, Bronze, Silver, Gold, Platinum, Diamond, Master, Grandmaster, and Challenger. The Iron to Diamond ranks each have 4 subdivisions from 1 to 4, with 1 being the highest subdivision and 4 being the lowest.

What Are We Trying to Do?

The question we are trying to answer is "How likely is a team likely to win a game based on the game statistics at 10 minutes?" A lot of important things can happen in the first ten minutes of the game, some of which can decide the flow of the game. So what I would like to do is see if it is possible to develop a model that can accurately predict the outcome of the game based on the stats at 10 minutes.

Why?

I am a gamer and I like to play and watch League of Legends a lot. I have always been interested in analyzing whether a game outcome is predictable and what aspect of the game is most impactful on the outcome. Through this analysis, I reflect upon my own performance in game and amend it to focus on these aspects as well as gain better insight into the mechanics of the game itself.

Loading in the Required Packages and Data

```
# Loading in required packages
library(tidyverse)
library(scales)
library(dplyr)
library(tidymodels)
library(readr)
library(kknn)
library(janitor)
library(ISLR)
library(discrim)
library(corrr)
library(corrplot)
library(tidytext)
library(ggplot2)
library(finalfit)
library(ranger)
```

```r
library(randomForest)
library(kernlab)
library(doParallel)
library(vip)
registerDoParallel(cores = parallel::detectCores())
tidymodels_prefer()

# Read in dataset
game_data <- read.csv("~/Documents/PSTAT 131/Final Project/data/unprocessed/high_diamond_ranked_10min.cs

# Look at the first couple rows of the dataset
game_data %>% head()
```

```
##       gameId blueWins blueWardsPlaced blueWardsDestroyed blueFirstBlood
## 1 4519157822        0              28                  2             1
## 2 4523371949        0              12                  1             0
## 3 4521474530        0              15                  0             0
## 4 4524384067        0              43                  1             0
## 5 4436033771        0              75                  4             0
## 6 4475365709        1              18                  0             0
##   blueKills blueDeaths blueAssists blueEliteMonsters blueDragons blueHeralds
## 1         9          6          11                 0           0           0
## 2         5          5           5                 0           0           0
## 3         7         11           4                 1           1           0
## 4         4          5           5                 1           0           1
## 5         6          6           6                 0           0           0
## 6         5          3           6                 1           1           0
##   blueTowersDestroyed blueTotalGold blueAvgLevel blueTotalExperience
## 1                   0         17210          6.6               17039
## 2                   0         14712          6.6               16265
## 3                   0         16113          6.4               16221
## 4                   0         15157          7.0               17954
## 5                   0         16400          7.0               18543
## 6                   0         15899          7.0               18161
##   blueTotalMinionsKilled blueTotalJungleMinionsKilled blueGoldDiff
## 1                    195                           36          643
## 2                    174                           43        -2908
## 3                    186                           46        -1172
## 4                    201                           55        -1321
## 5                    210                           57        -1004
## 6                    225                           42          698
##   blueExperienceDiff blueCSPerMin blueGoldPerMin redWardsPlaced
## 1                 -8         19.5         1721.0             15
## 2              -1173         17.4         1471.2             12
## 3              -1033         18.6         1611.3             15
## 4                 -7         20.1         1515.7             15
## 5                230         21.0         1640.0             17
## 6                101         22.5         1589.9             36
##   redWardsDestroyed redFirstBlood redKills redDeaths redAssists
## 1                 6             0        6         9          8
## 2                 1             1        5         5          2
## 3                 3             1       11         7         14
## 4                 2             1        5         4         10
```

```
## 5                2            1        6         6          7
## 6                5            1        3         5          2
##   redEliteMonsters redDragons redHeralds redTowersDestroyed redTotalGold
## 1                0          0         0                  0        16567
## 2                2          1         1                  1        17620
## 3                0          0         0                  0        17285
## 4                0          0         0                  0        16478
## 5                1          1         0                  0        17404
## 6                0          0         0                  0        15201
##   redAvgLevel redTotalExperience redTotalMinionsKilled
## 1         6.8              17047                   197
## 2         6.8              17438                   240
## 3         6.8              17254                   203
## 4         7.0              17961                   235
## 5         7.0              18313                   225
## 6         7.0              18060                   221
##   redTotalJungleMinionsKilled redGoldDiff redExperienceDiff redCSPerMin
## 1                          55        -643                8        19.7
## 2                          52        2908             1173        24.0
## 3                          28        1172             1033        20.3
## 4                          47        1321                7        23.5
## 5                          67        1004             -230        22.5
## 6                          59        -698             -101        22.1
##   redGoldPerMin
## 1        1656.7
## 2        1762.0
## 3        1728.5
## 4        1647.8
## 5        1740.4
## 6        1520.1
```

This data was obtained from the Kaggle dataset, League of Legends Diamond Ranked Games (10 min), which was scraped from the Riot API by Kaggle user Yi Lan Ma. This data contains contains the first 10 min. stats of approx. 10k ranked games from the ranks Diamond 1 to Master.

Exploratory Data Analysis

Before we can start doing any sort of model building and prediction, we have know the ins and outs of our data in order to build the best model. As this dataset was assembled by someone else, we don't how conducive it will be for our purposes. Thus, we need to explore it and see whether or not we need to make changes to it and, if so, how extensive those changes will need to be.

Exploring the Data

Let's first run a couple functions on the data to get an idea of what we're working with.

```
# Look at the size of the dataset
dim(game_data)
```

```
## [1] 9879   40
```

```r
# Check for missing values
sum(is.na(game_data))
```

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

We can see that the data set contains 9879 rows and 40 columns as well as zero missing values. Out of those 40 columns, one of those is `gameId`, which is unneeded for our purposes, and `blueWins`, which is our response variable. Removing these two columns still leaves us with 38 predictors, quite a bit more than what we would like.

Let's take a look at these 38 predictors and their descriptions.

`blueWardsPlaced`: The number of wards placed by the blue team

`blueWardsDestroyed`: The number of wards destroyed by the blue team

`blueFirstBlood`: First kill of the game. 1 if the blue team did the first kill, 0 otherwise.

`blueKills`: Number of enemies killed by the blue team

`blueDeaths`: Number of deaths from the blue team

`blueAssists`: Number of kill assists from the blue team

`blueEliteMonster`: Number of elite monsters killed by the blue team (Dragons and Heralds)

`blueDragons`: Number of dragons killed by the blue team

`blueHeralds`: Number of heralds killed by the blue team

`blueTowersDestroyed`: Number of towers destroyed by the blue team

`blueTotalGold`: Blue team total gold

`blueAvgLevel`: Blue team average champion level

`blueTotalExperience`: Blue team total experience

`blueTotalMinions`: Blue team total minions killed (CS)

`blueTotalJungleMinionsKilled`: Blue team total jungle monsters killed

`blueGoldDiff`: Blue team gold difference compared to the enemy team

`blueExperienceDiff`: Blue team experience difference compared to the enemy team

`blueCSPerMin`: Blue team CS (minions) per minute

`blueGoldPerMin`: Blue team gold per minute

`redWardsPlaced`: The number of wards placed by the red team

`redWardsDestroyed`: The number of wards destroyed by the red team

`redFirstBlood`: First kill of the game. 1 if the red team did the first kill, 0 otherwise

`redKills`: Number of enemies killed by the red team

`redDeaths`: Number of deaths from the red team

`redAssists`: Number of kill assists from the red team

`redEliteMonsters`: Number of elite monsters killed by the red team (Dragons and Heralds)

`redDragons`: Number of dragons killed by the red team

`redHeralds`: Number of heralds killed by the red team

`redTowersDestroyed`: Number of towers destroyed by the red team

`redTotalGold`: Red team total gold

`redAvgLevel`: Red team average champion level

`redTotalExperience`: Red team total experience

`redTotalMinionsKilled`: Red team total minions killed (CS)

`redTotalJungleMinionsKilled`: Red team total jungle monsters killed

`redGoldDiff`: Red team gold difference compared to the enemy team

`redExperienceDiff`: Red team experience difference compared to the enemy team

`redCSPerMin`: Red team CS (minions) per minute

`redGoldPerMin`: Red team gold per minute

Looking through the data, a lot of the predictors are either mutually exclusive. inversely correlated, or something along those lines. For example, `redFirstBlood` and `blueFirstBlood` are mutually exclusive, as only a single team can achieve that per game. Additionally, the variables `blueGoldDiff` and `redGoldDiff` have the same absolute value as gold difference is calculated by subtracting the respective team's gold from the opposite team. Factors like these allow us to drop or combine a number of predictors, reducing our effective predictors drastically.

Tidying the Data

Now, let's do some tidying and modify the predictors, so we'll have a more reasonable amount. Essentially, what we will be doing is combine the predictors such that the new predictors will be the difference between blue team and red team for that respective variable. This will narrow down our predictors significantly, by over half. After doing this, we will be able to drop all our old predictors as they will all be incorporated into the new predictors, save for a few.

```
# modifies the original data set
game_data_cleaned <- game_data %>%
  clean_names() %>%
  mutate(
    first_blood = blue_first_blood,
    dragons = blue_dragons - red_dragons,
    heralds = blue_heralds - red_heralds,
    towers_destroyed = blue_towers_destroyed - red_towers_destroyed,
    wards_placed = blue_wards_placed - red_wards_placed,
    wards_destroyed = blue_wards_destroyed - red_wards_destroyed,
    experience = blue_experience_diff,
    kills = blue_kills - red_kills,
    assists = blue_assists - red_assists,
    gold = blue_gold_diff,
    minions = blue_total_minions_killed - red_total_minions_killed,
    avg_level = blue_avg_level - red_avg_level,
    elite_monsters = blue_elite_monsters - red_elite_monsters,
    total_jungle_minions_killed = blue_total_jungle_minions_killed - red_total_jungle_minions_killed
  ) %>%
  select(
    -game_id, -blue_kills, -blue_assists, -blue_elite_monsters, -blue_avg_level, -blue_total_jungle_mini
    -red_deaths, -red_wards_placed,
    -red_wards_destroyed, -red_cs_per_min, -red_gold_per_min,
```

```
    -red_kills, -red_assists, -red_elite_monsters, -red_avg_level, -red_total_jungle_minions_killed,
    -blue_total_gold, -blue_total_experience,
    -blue_cs_per_min, -blue_gold_per_min, -blue_first_blood,
    -red_towers_destroyed, -red_dragons,
    -red_heralds, -blue_towers_destroyed,
    -blue_dragons, -blue_heralds, -blue_gold_diff, -red_gold_diff, -blue_experience_diff, -red_experien
    -red_first_blood, -red_total_gold,
    -red_total_experience,
    -blue_total_minions_killed, -blue_deaths, -blue_wards_placed, -blue_wards_destroyed,

  )
```

After all these modifications, let's take a peek at our new dataset.

```
game_data_cleaned %>% head()
```

```
##   blue_wins first_blood dragons heralds towers_destroyed wards_placed
## 1         0           1       0       0               0           13
## 2         0           0      -1      -1              -1            0
## 3         0           0       1       0               0            0
## 4         0           0       0       1               0           28
## 5         0           0      -1       0               0           58
## 6         1           0       1       0               0          -18
##   wards_destroyed experience kills assists  gold minions avg_level
## 1              -4         -8     3       3   643      -2      -0.2
## 2               0      -1173     0       3 -2908     -66      -0.2
## 3              -3      -1033    -4     -10 -1172     -17      -0.4
## 4              -1         -7    -1      -5 -1321     -34       0.0
## 5               2        230     0      -1 -1004     -15       0.0
## 6              -5        101     2       4   698       4       0.0
##   elite_monsters total_jungle_minions_killed
## 1              0                         -19
## 2             -2                          -9
## 3              1                          18
## 4              1                           8
## 5             -1                         -10
## 6              1                         -17
```

```
game_data_cleaned %>% dim()
```

```
## [1] 9879   15
```

We now have a total of 14 predictors, much more reasonable than before.

Let's get a look at each of the new predictors as well as our response variable.

`blue_wins`: Game outcome for blue team (1 for win, 0 for loss)

`first_blood`: First blood for blue team (1 for yes, 0 for no)

`dragons`: Difference between number of dragons slain by blue and red team (0: none, 1: blue, -1: red)

`heralds`: Difference between number of heralds slain by blue and red team (0: none, 1: blue, -1: red)

`experience`: Difference between total experience gained by blue and red team

`towers_destroyed`: Difference between total number of towers destroyed by blue and red team

`wards_placed`: Difference between number of wards placed by blue and red team

`wards_destroyed`: Difference between number of wards destroyed by blue and red team

`kills`: Difference between total number of kills by blue and red team

`assists`: Difference between total number of assists by blue and red team

`gold`: Difference between total gold possessed by blue and red team

`minions`: Difference between total number of minions killed by blue and red team

`avg_level`: Difference between average champion level of blue and red team

`elite_monsters`: Difference between number of elite monsters slain by blue and red team

`total_jungle_minions_killed`: Difference between total jungle minions killed by blue and red team

A couple of these variables, namely `blue_wins, first_blood, dragons,` and `heralds`, need to be changed into categorical variables as their values represent categories.

```
game_data_cleaned$blue_wins <- as.factor(game_data_cleaned$blue_wins)
game_data_cleaned$first_blood <- as.factor(game_data_cleaned$first_blood)
game_data_cleaned$dragons <- as.factor(game_data_cleaned$dragons)
game_data_cleaned$heralds <- as.factor(game_data_cleaned$heralds)

# saves cleaned data as csv
write_csv(game_data_cleaned, "~/Documents/PSTAT 131/Final Project/data/processed/cleaned_data.csv")
```

With those adjustments, we have finished the tidying of the data and can now move on to the exploratory data analysis.

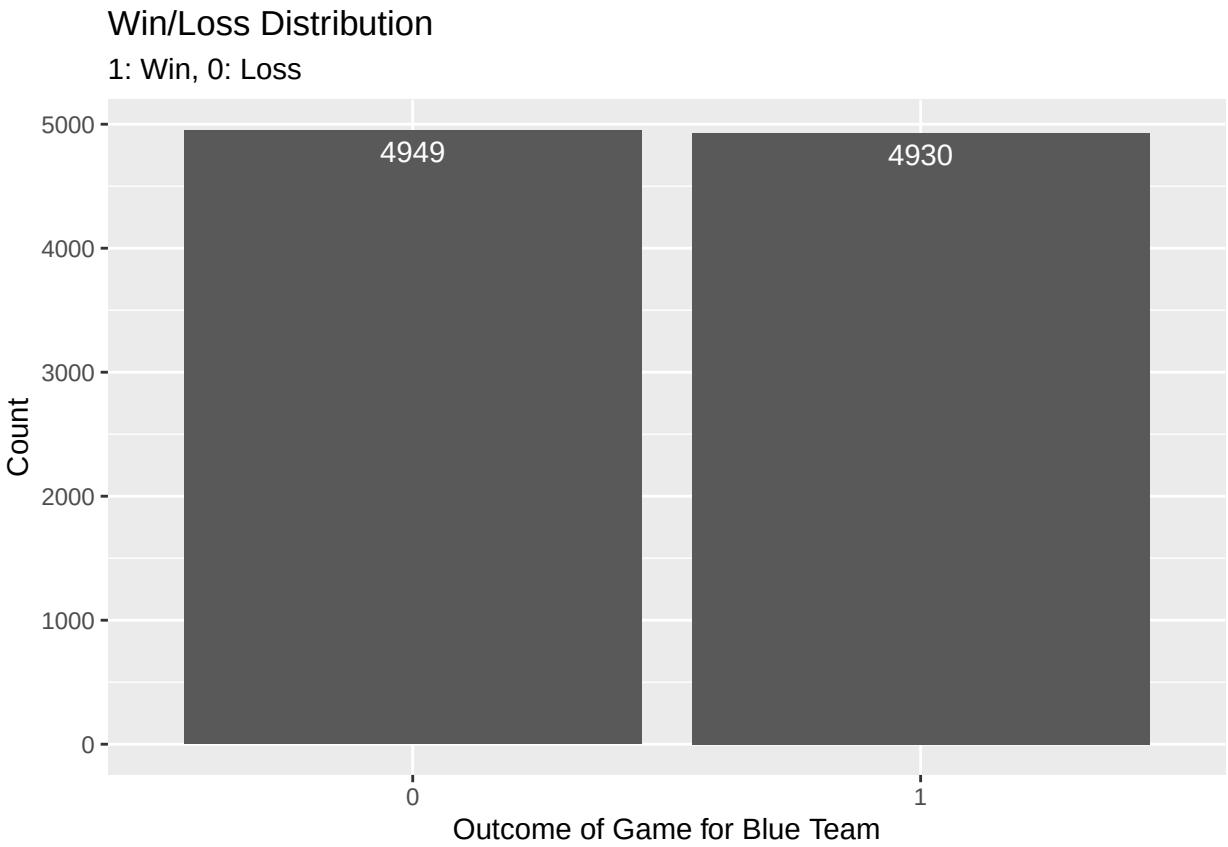
Visual EDA

Now that we have the exact dataset we will be working with, we can start looking at the relationship between the outcome variable and specific predictors using some visualizations.

Win/Loss Distribution

First, let's look at the outcome variable, `blue_wins` to see the distribution of wins and losses.

```
ggplot(data = game_data_cleaned, aes(x = blue_wins)) +
  geom_bar() +
  geom_text(aes(label = after_stat(count)),
            stat = "count", vjust = 1.5, colour = "white") +
  labs(x = "Outcome of Game for Blue Team",
            y = "Count",
            title = "Win/Loss Distribution",
            subtitle = "1: Win, 0: Loss")
```

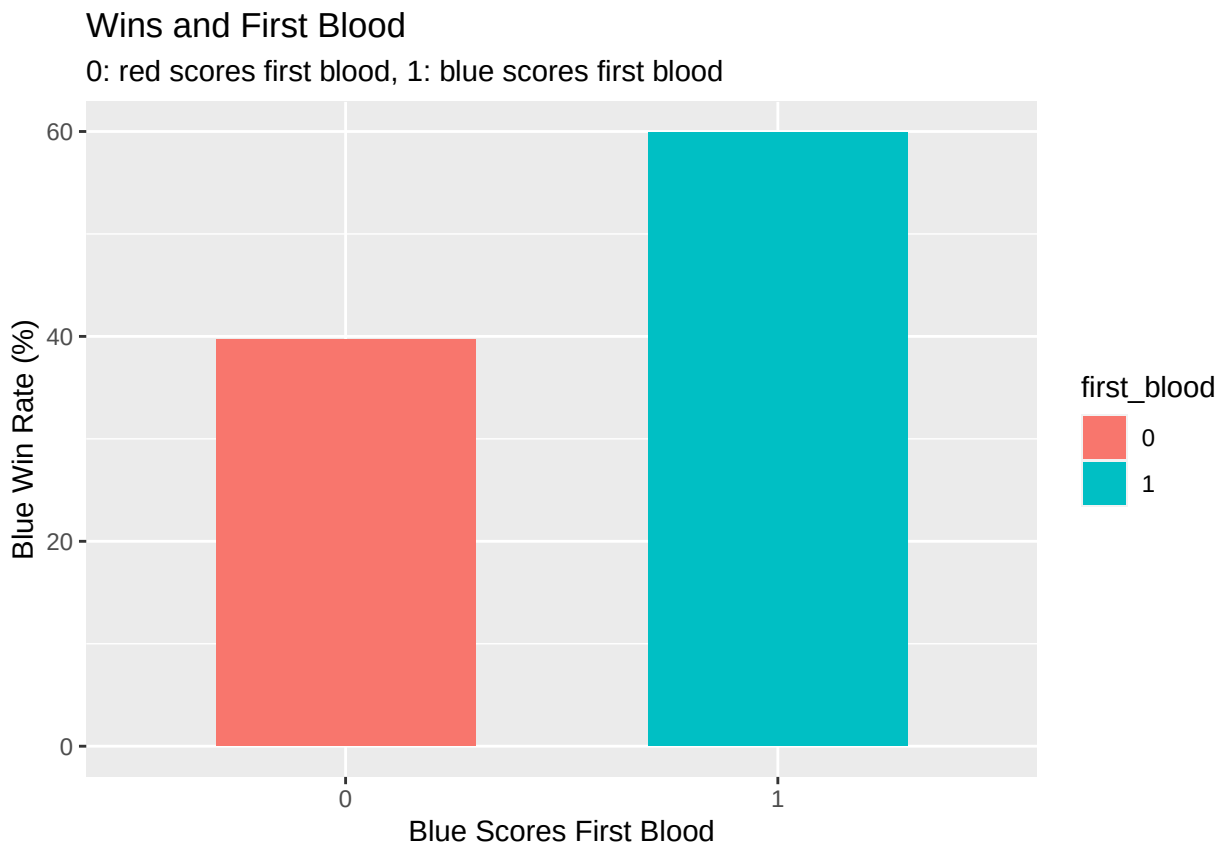


From the graph, we can see that the outcome distribution about even, with a near equal amount of wins and losses. There are 19 more losses than wins.

First Blood

Next, let's take a look at the relationship between first bloods and wins.

```
game_data_cleaned %>%
  group_by(first_blood) %>%
  mutate(wins = ifelse(blue_wins == "1", 1, 0)) %>%
  summarise(
    total = n(),
    prob = (sum(wins)/total)*100
  ) %>%
  ggplot(aes(x = first_blood, y = prob, , fill = first_blood)) +
  geom_bar(stat = "identity", width = 0.6) +
  scale_x_discrete (limits = c("0", "1")) +
  labs(
    x = "Blue Scores First Blood",
    y = "Blue Win Rate (%)",
    title = "Wins and First Blood",
    subtitle = "0: red scores first blood, 1: blue scores first blood"
  )
```



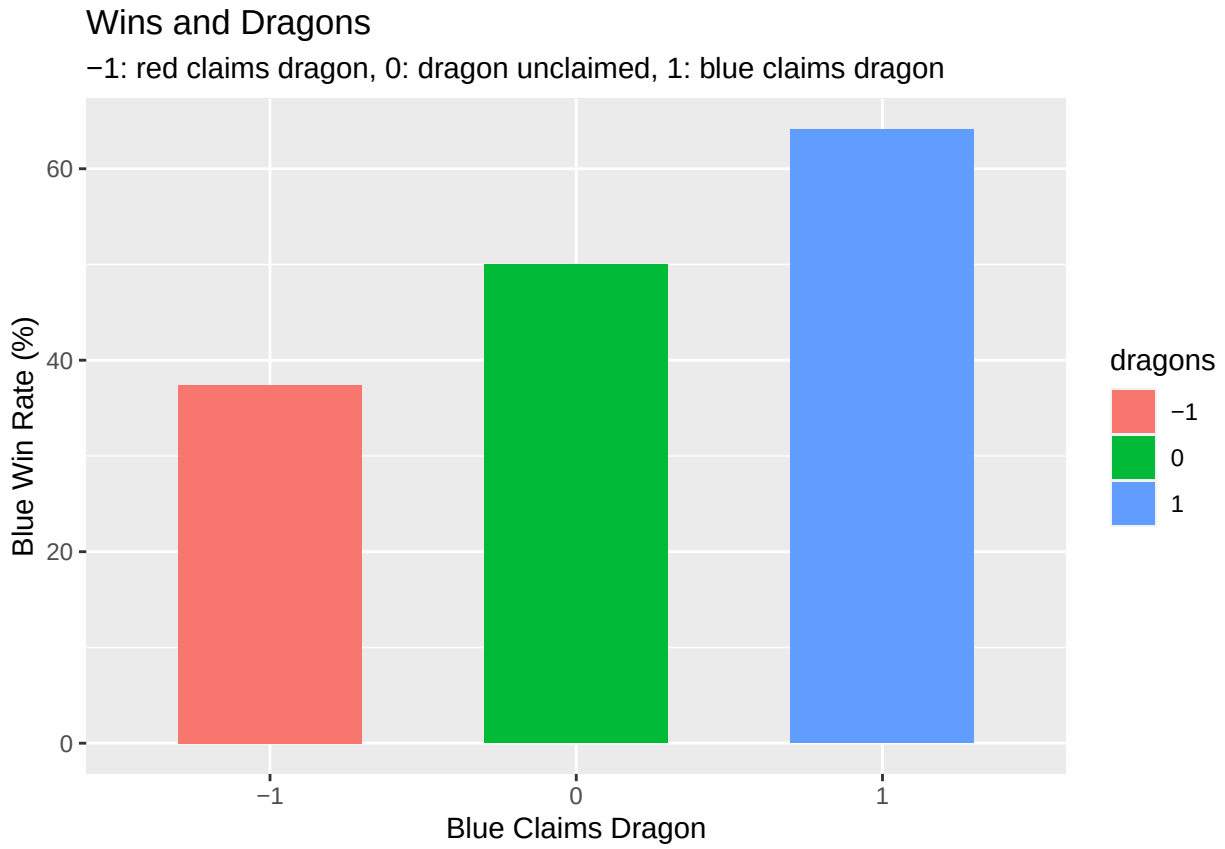
First blood is the term for the first kill in the game. This is quite important as first bloods reward more experience and gold than a normal kill, so whatever team scores first blood gains a big, but not unbridgeable, head start. From the bar chart, we can see that blue team had a higher win rate when they scored first blood than when they did not. Thus, we can see that there is a positive correlation between scoring first blood and winning.

Dragons

Dragons are an integral objective in League of Legends as killing one will give the team a buff. A single dragon will spawn within the first 10 minutes and only a single team can claim the buff by slaying the dragon. Let's see how much claiming the dragon's buff impacts the outcome of the game.

```
game_data_cleaned%>%
  group_by(dragons) %>%
  mutate(wins = ifelse(blue_wins == "1", 1, 0)) %>%
  summarise(
    total = n(),
    prob = (sum(wins)/total)*100
  ) %>%
  ggplot(aes(x = dragons, y = prob, fill = dragons)) +
  geom_bar(stat = "identity", width = 0.6) +
  scale_x_discrete (limits = c("-1", "0", "1")) +
  labs(
    x = "Blue Claims Dragon",
    y = "Blue Win Rate (%)",
```

```
    title = "Wins and Dragons",
    subtitle = "-1: red claims dragon, 0: dragon unclaimed, 1: blue claims dragon"
  )
```



From the chart, we can see that the win rate when blue claims the dragon is over 60%, the win rate when red claims the dragon is below 40%, and the win rate when the dragon is unclaimed is about 50%. From these percentages, we can see that claiming the dragon has a positive correlation with winning the game, increasing the win rate by ~10%.

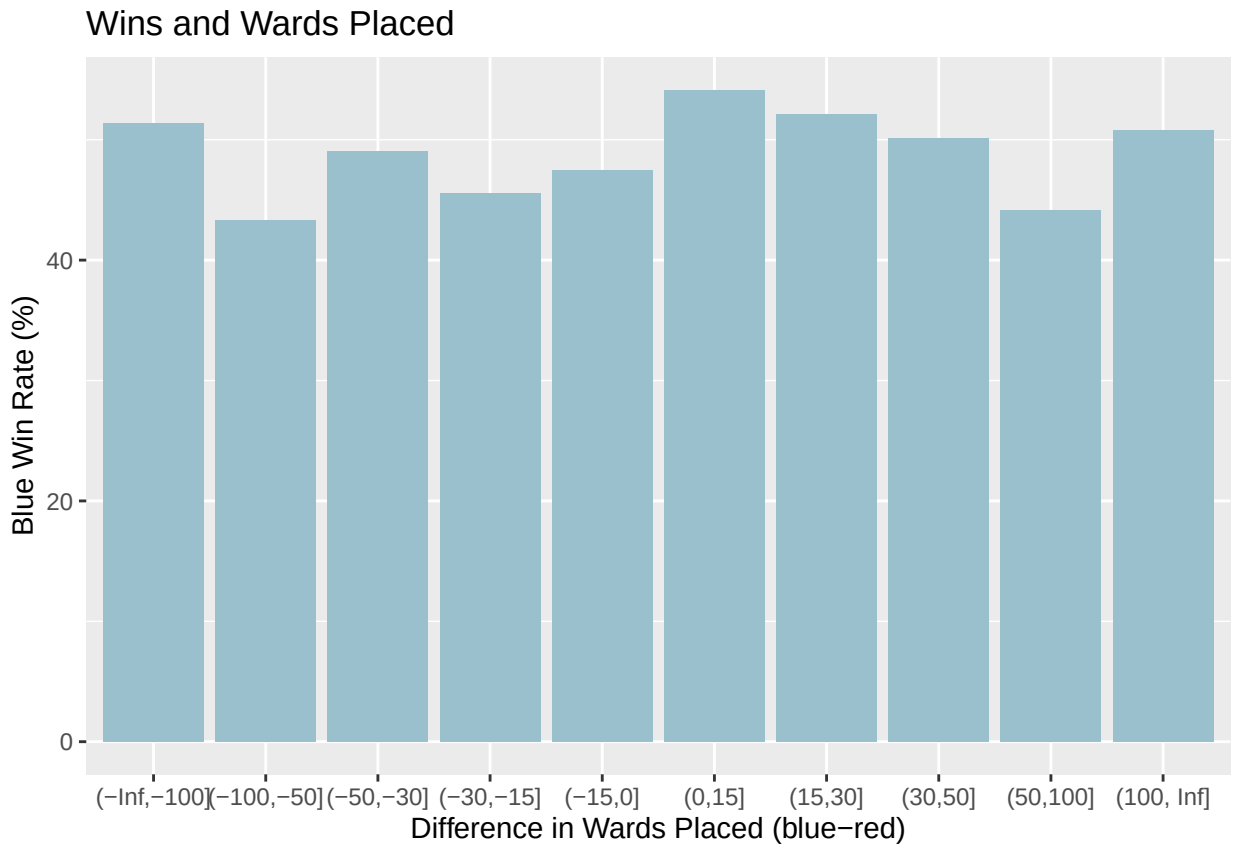
Wards Placed

League of Legends is a game with a map that is constantly covered in the fog of war. This means what you can see on the map is limited to what is around you and your teammates. To rectify this, items called wards can be placed around the map so that you have vision of an area even while you and your teammates are not near it. This is incredibly important for map and objective control as it will allow you to know where enemies are and what they might be doing.

```
# splits values of variable into intervals
split <- function(x) {
  cut(x,
    breaks = c(-Inf, -100, -50, -30, -15, 0, 15, 30, 50, 100, Inf)
  )
}

game_data_cleaned%>%
```

```
  mutate(
    wins = ifelse(blue_wins == "1", 1, 0),
    wards = split(wards_placed)) %>%
  group_by(wards) %>%
  summarise(
    total = n(),
    prob = (sum(wins)/total)*100
  ) %>%
  ggplot(aes(x = wards, y = prob)) +
  geom_bar(stat = "identity", fill = "lightblue3") +
  labs(
    x = "Difference in Wards Placed (blue-red)",
    y = "Blue Win Rate (%)",
    title = "Wins and Wards Placed"
  )
```



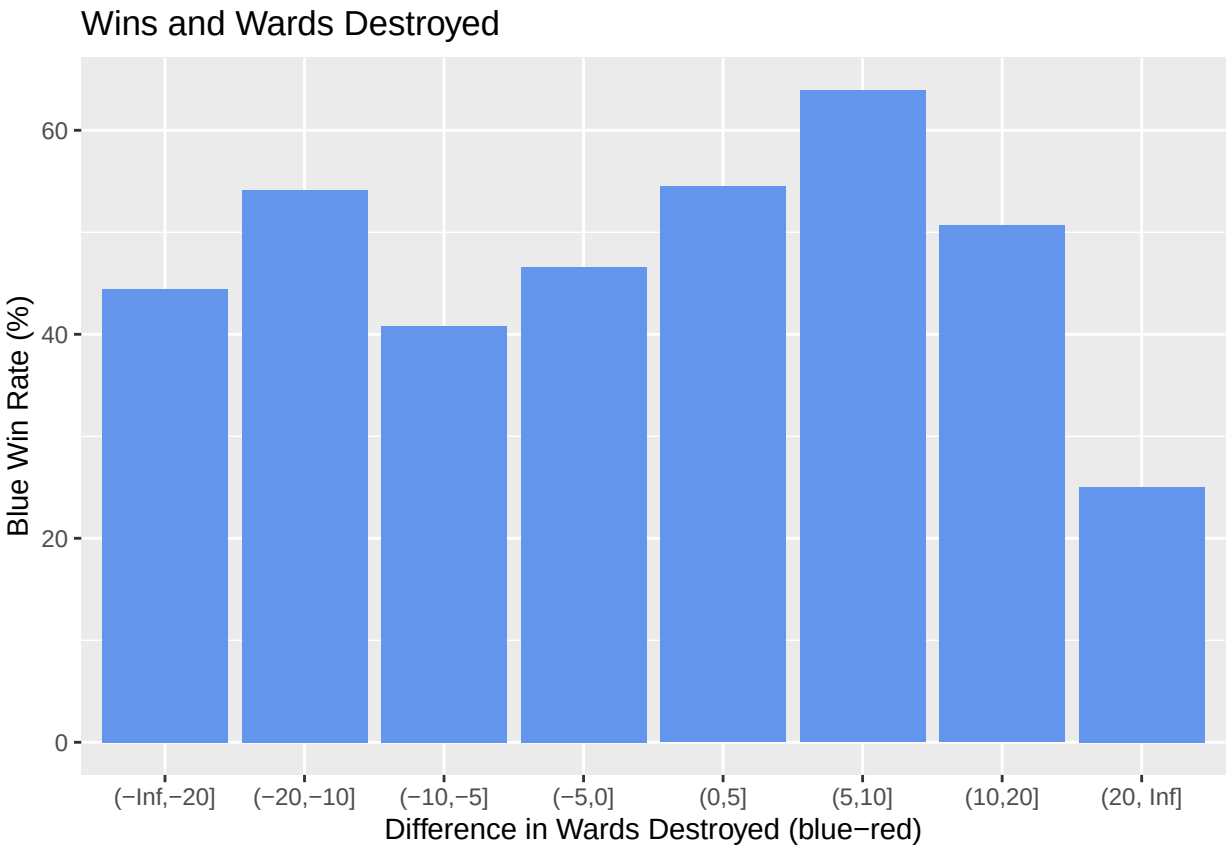
The chart shows that there is actually not a huge correlation between the amount of wards placed and winning the game. The win rate hovers between 40% and 60% for all intervals and there is no recognizable pattern. When the red team placed over 100 more wards than the blue team, the blue team had a win rate of around 50%. But when the red team placed between 50 and 100 more wards than the blue team, the blue team's win rate was closer to 40%. Points such as those really highlight how the amount of wards placed has a negligible, if any, effect on the outcome of the game.

Wards Destroyed

Destroying wards denies your opponent vision and can be a deciding factor in claiming objectives and winning fights. Let's see how much of an effect destroying wards has on winning the game.

```
# splits values of variable into intervals
split2 <- function(x) {
  cut(x,
    breaks = c(-Inf, -20, -10, -5, 0, 5, 10, 20, Inf)
  )
}

game_data_cleaned %>%
  mutate(
    wins = ifelse(blue_wins == "1", 1, 0),
    wards_d = split2(wards_destroyed)) %>%
  group_by(wards_d) %>%
  summarise(
    total = n(),
    prob = (sum(wins)/total)*100
  ) %>%
  ggplot(aes(x = wards_d, y = prob)) +
  geom_bar(stat = "identity", fill = "cornflowerblue") +
  labs(
    x = "Difference in Wards Destroyed (blue-red)",
    y = "Blue Win Rate (%)",
    title = "Wins and Wards Destroyed"
  )
```

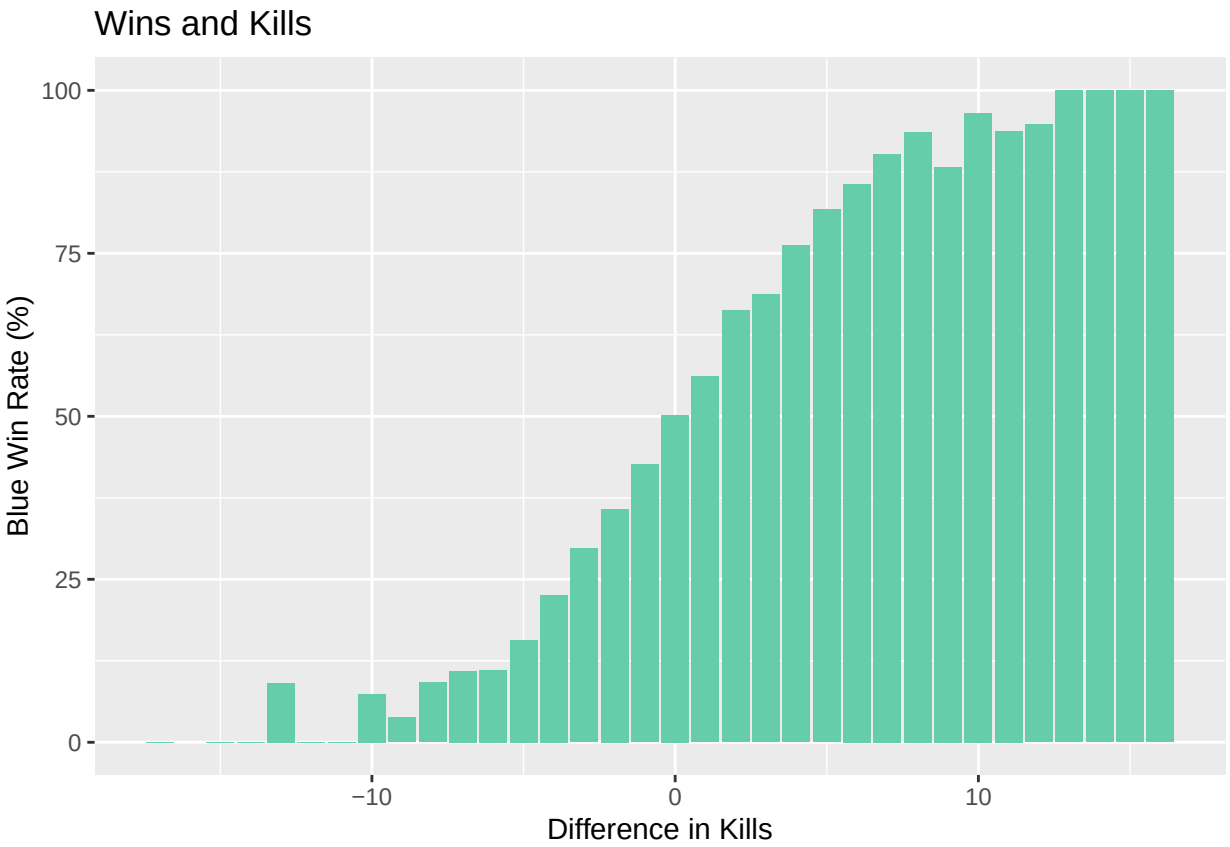


There seems to be a positive correlation between the amount of wards destroyed and the winning the game. From the interval (-10,5] to (5,10], the win rate increases as the blue team gains an advantage in ward destruction. However, the win rate drops off after that, suggesting that focusing too much on destroying wards can have a negative effect on the win rate. The higher win rate where the red team has a large advantage in ward destruction makes that suggestion more plausible.

Kills

Kills are arguably the best way to gain an advantage over the opposing team as kills grant you a lot of gold and experience, allowing your champion to become stronger quickly. As such, the amount of kills would have a great impact on the outcome of the game.

```
game_data_cleaned %>%
  group_by(kills) %>%
  mutate(wins = ifelse(blue_wins == "1", 1, 0)) %>%
  summarise(
    total = n(),
    prob = (sum(wins)/total)*100
  ) %>%
  ggplot(aes(x = kills, y = prob)) +
  geom_bar(stat = "identity", fill = "aquamarine3") +
  labs(
    x = "Difference in Kills",
    y = "Blue Win Rate (%)",
    title = "Wins and Kills")
```



As expected, the amount of kills is very positively correlated with the outcome of the game. Interestingly, when neither team has an advantage in kills, the win rate is 50%. As the blue team gains an upper hand in kills, their win rate increases aggressively, eventually hitting an over 90% win rate once they have a ten kill lead. When red team has the advantage, the blue team win rate drops drastically, where it hits zero once they are down ten kills.

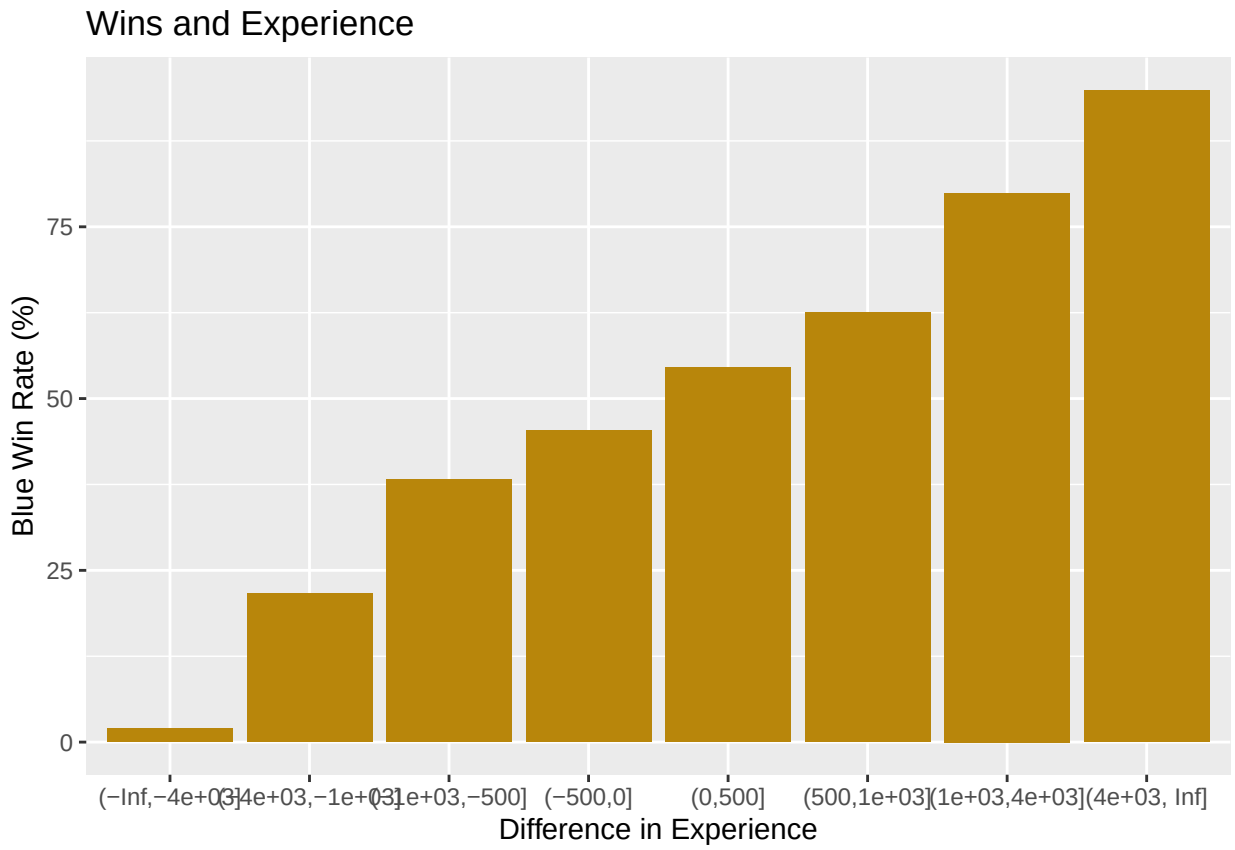
Experience

Experience is what your champion collects in order to level up. Experience can be gained in a variety of ways, those ways being killing minions, dragons, heralds, and enemy champions as well as destroying towers.

```
split3 <- function(x) {
  cut(x,
    breaks = c(-Inf, -4000, -1000, -500, 0, 500, 1000, 4000, Inf)
  )
}

game_data_cleaned %>%
  mutate(
    wins = ifelse(blue_wins == "1", 1, 0),
    exp = split3(experience)) %>%
  group_by(exp) %>%
  summarise(
    total = n(),
    prob = (sum(wins)/total)*100
```

```
  ) %>%
  ggplot(aes(x = exp, y = prob)) +
  geom_bar(stat = "identity", fill = "darkgoldenrod") +
  labs(
    x = "Difference in Experience",
    y = "Blue Win Rate (%)",
    title = "Wins and Experience"
  )
```



There is a clear positive correlation between the difference in experience and the outcome of the game. As the difference in experience increases, the blue team's win rate increases and vice versa. Once the difference in experience is over 1000 for either team, their respective win rate hits over 75%. Minor differences in experience do not seem to have a great effect on the win rate as the win rate for differences from -500 to 500 is about 50%. Anything over that difference has more significant impact on the win rate.

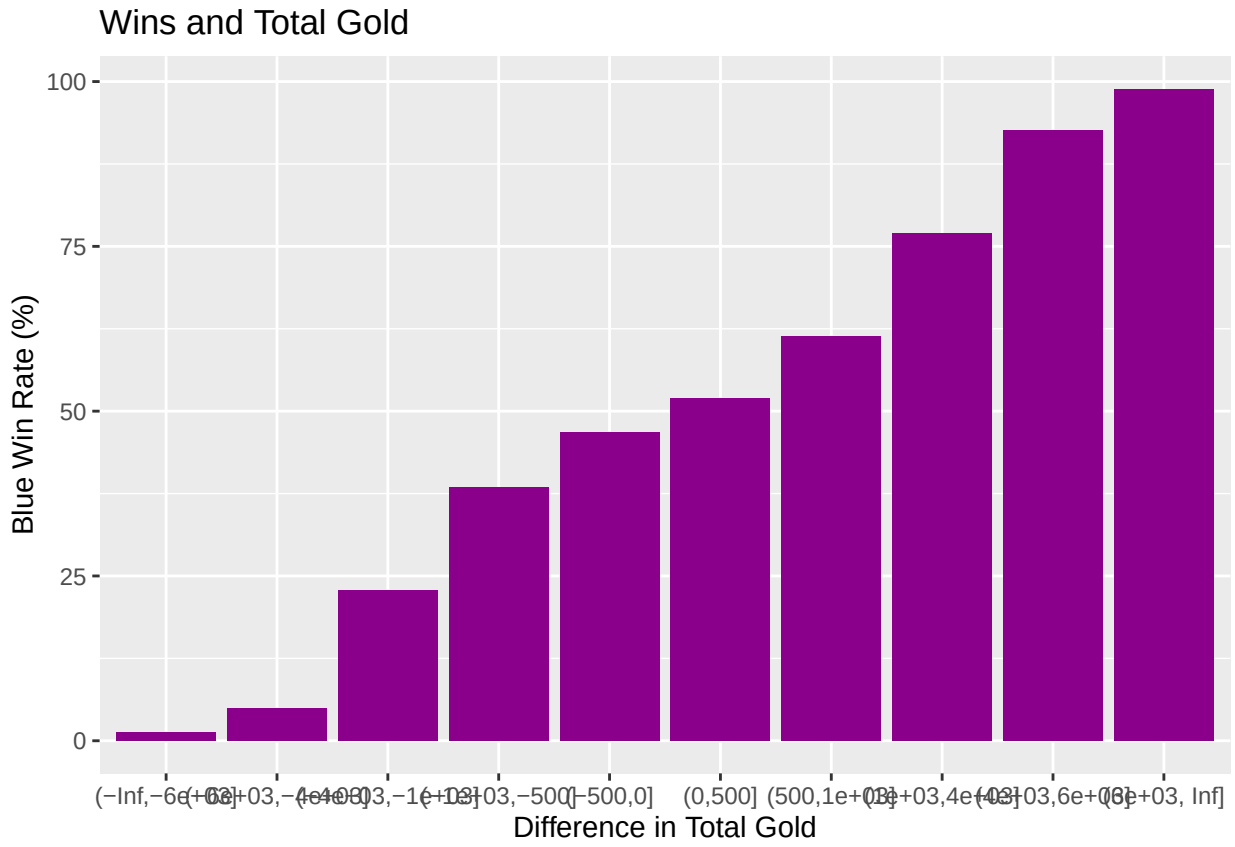
Total Gold

Gold is an important resource in League of Legends as it allows you to buy items, which are a core component in making your champion stronger, as well as wards. Gold can be gained through the same ways experience is gained. As such, gold is likely to have positive correlation with the outcome of the game, similar to experience. Let's see if that's true.

```
split4 <- function(x) {
  cut(x,
    breaks = c(-Inf,-6000, -4000, -1000, -500, 0, 500, 1000, 4000, 6000, Inf)
```

```
  )
}

game_data_cleaned %>%
  mutate(
    wins = ifelse(blue_wins == "1", 1, 0),
    gold = split4(gold)) %>%
  group_by(gold) %>%
  summarise(
    total = n(),
    prob = (sum(wins)/total)*100
  ) %>%
  ggplot(aes(x = gold, y = prob)) +
  geom_bar(stat = "identity", fill = "darkmagenta") +
  labs(
    x = "Difference in Total Gold",
    y = "Blue Win Rate (%)",
    title = "Wins and Total Gold"
  )
```

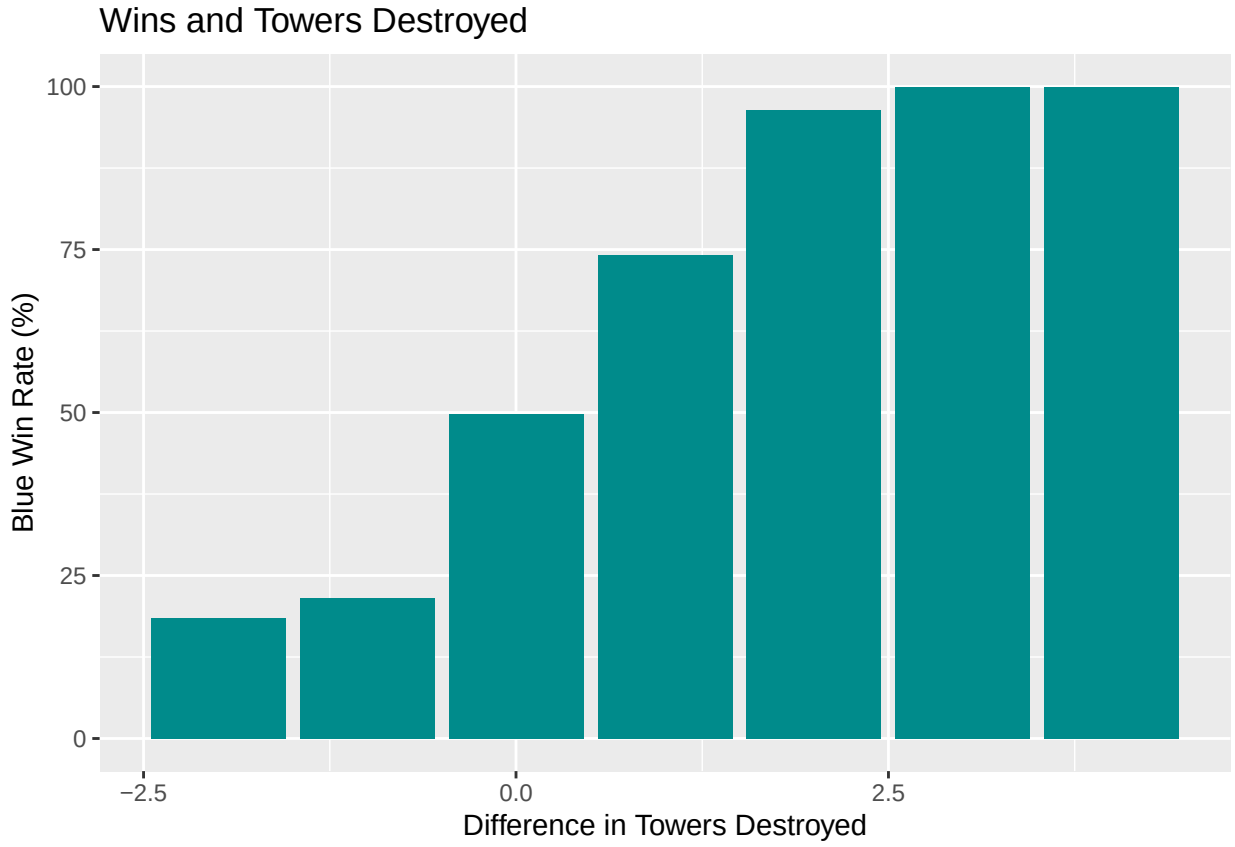


As expected, there is a positive correlation between the difference in total gold and the outcome of the game. As the difference increases positively, the blue team win rate increases correspondingly. The inverse is true for the red team. When the gold difference is over 1000 for either team, the respective team's win rate shoots up to 75%. Once again, minor differences in total gold do not seem to affect the win rate noticeably, with the win rate sitting at about 50%. Overall, the difference in gold only starts to impact the outcome of the game once it crosses the threshold of 500.

Towers Destroyed

Towers are structures that must be destroyed in order the reach the enemy Nexus, the structure that must be destroyed in order the win the game. As mentioned before, destroying a tower grants experience and gold.

```
game_data_cleaned %>%
  group_by(towers_destroyed) %>%
  mutate(wins = ifelse(blue_wins == "1", 1, 0)) %>%
  summarise(
    total = n(),
    prob = (sum(wins)/total)*100
  ) %>%
  ggplot(aes(x = towers_destroyed, y = prob)) +
  geom_bar(stat = "identity", fill = "darkcyan") +
  labs(
    x = "Difference in Towers Destroyed",
    y = "Blue Win Rate (%)",
    title = "Wins and Towers Destroyed"  )
```

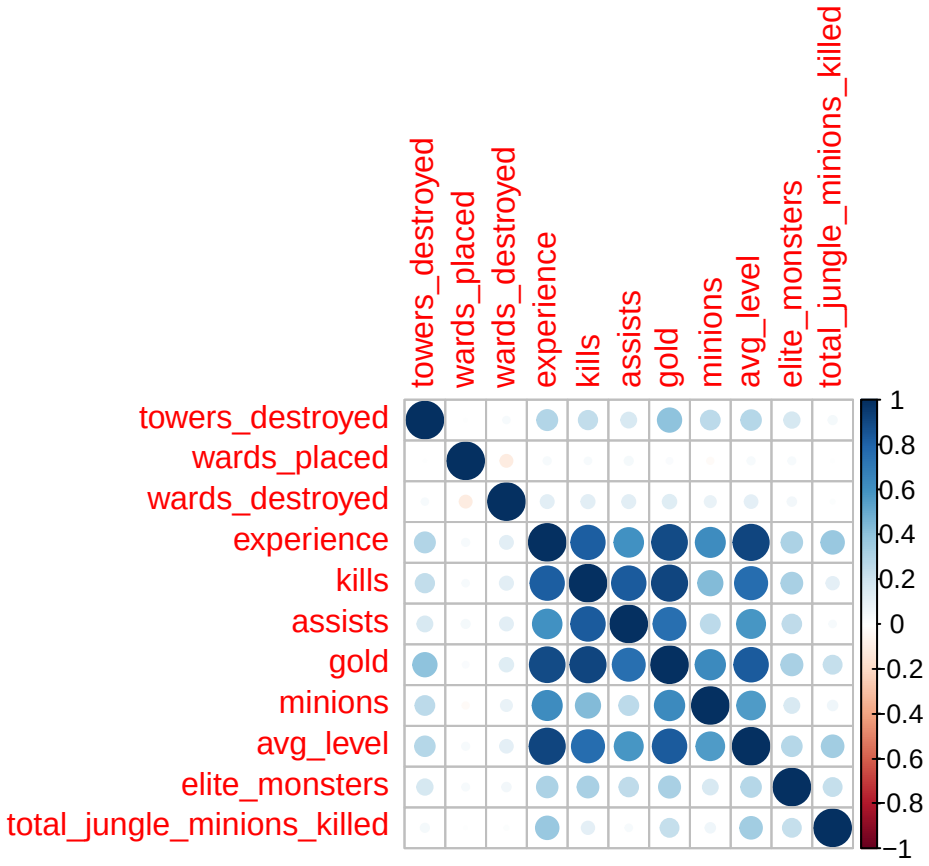


Unsurprisingly, a higher difference in towers destroyed increases the win rate as the team is getting closer to the main objective. When the difference in towers is equal, the win rate is 50%. In short, there is a positive correlation between the amount of towers destroyed and winning the game such that the team with the advantage in towers destroyed has a higher chance of winning.

Correlation Plot

Finally, let's take a look at the relationships between all the continuous variables using a correlation plot.

```
game_data_cleaned %>%
  select(-blue_wins, -first_blood, -dragons, -heralds) %>%
  cor() %>%
  corrplot()
```



We can see that there are 6 variables that have a high positive correlation with each other. These 5 variables are `experience`, `kills`, `assists`, `gold`, `minions`, and `avg_level`. This makes sense since each of these variables are very closely related, with three of those variables directly feeding into the other three. From this, we can expect that these 6 variable will likely be the most important for predicting the outcome of the game.

Setting Up for the Models

Now that we have a better idea of how a few of the predictors affect the outcome of a game, we can begin setting up for our models so we can start building them. We will split the cleaned data set into training and test data, create recipe to be used by those models, and establish cross-validation within our models.

Data Split

First, we will be splitting our data into a training and test dataset. The training data set will be used to train the models we will be building and the test data set will be used to test the accuracy of our models once they are built. To begin, we will set a seed so that the random split will be the exact same every time we go back and work on the code. The proportion that I will be using for the split will be 75/25 as this split will allow for more training data while also having enough observations to be tested later. Having done this, we can perform our split and stratify on our response variable, `blue_wins`.

```
set.seed(1234)
game_split <- initial_split(game_data_cleaned, prop = 0.75, strata = "blue_wins")

game_train <- training(game_split)
game_test <- testing(game_split)
```

Dimensions of training data:

```
dim(game_train)
```

```
## [1] 7408   15
```

Dimensions of testing data:

```
dim(game_test)
```

```
## [1] 2471   15
```

The training data has 7408 observations and the testing data has 2471 observations.

Recipe Creation

As we will be working with the same predictors, model conditions, and response variable the whole time, we'll create one central recipe for our models to use. We will be using the same 14 predictors mentioned before, where we dummy code `blue_wins`, `first_blood`, `dragons`, and `heralds` as they are categorical variable. We also normalize the predictors by centering and scaling them.

```
win_recipe <- recipe(blue_wins ~ ., data = game_train) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_center(all_predictors()) %>%
  step_scale(all_predictors())
```

K-Fold Cross Validation

We'll stratify on the response variable, `blue_wins`, to make sure the data is not imbalanced and create 10 folds. We use k-fold cross validation in order to get a better estimate of the testing accuracy.

```
game_folds <- vfold_cv(game_train, v = 10, strata = blue_wins)
```

Model Building

With all this done, we can finally start building the models that will do the predicting. I will be fitting 8 different models. The metric the we will be using to evaluate the performance of the models is **ROC AUC** because it shows the most significant level of efficiency in a binary classification model where the data is not perfectly balanced. To be more specific, the `roc_auc` metric measures the area under the curve of ROC (receiver operating characteristic) curve, which is is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied.

Model Building Process

The models all had a similar building process, which I will outline below.

1. Set up the model by specifying the type of model, the parameters to be tuned (if any), the engine the model comes from, and the mode, which in our case would be classification.

```
# logistic regression
# no parameters to tune
log_model <- logistic_reg() %>%
  set_mode("classification") %>%
  set_engine("glm")

# elastic net logistic regression
# tune penalty and mixture
enlog_model <- logistic_reg(mixture = tune(),
                            penalty = tune()) %>%
  set_mode("classification") %>%
  set_engine("glmnet")

# k-nearest neighbors
# tune number of neighbors
knn_model <- nearest_neighbor(neighbors = tune()) %>%
  set_mode("classification") %>%
  set_engine("kknn")

# random forest
# tune mtry, trees, and min_n
rf_model <- rand_forest(mtry = tune(), trees = tune(), min_n = tune()) %>%
  set_mode("classification") %>%
  set_engine("ranger", importance = "impurity")

# ridge regresson
# tune penalty, mixture is 0 for ridge
ridge_model <- logistic_reg(mixture = 0,
                        penalty = tune()) %>%
  set_mode("classification") %>%
  set_engine("glmnet")

# lasso regression
# tune penalty, mixture is 1 for lasso
lasso_model <- logistic_reg(penalty = tune(),
                        mixture = 1) %>%
```

```
  set_mode("classification") %>%
  set_engine("glmnet")

# linear discriminant analysis
# no parameters to tune
lda_model <- discrim_linear() %>%
  set_mode("classification") %>%
  set_engine("MASS")

# support vector machine
# tune cost and degree is 1 for sum
svm_model <- svm_poly(degree = 1, cost = tune()) %>%
  set_mode("classification") %>%
  set_engine("kernlab")
```

2. Set up the workflow for the model and add the recipe and model

```
# logistic regression
log_wkflow <- workflow() %>%
  add_model(log_model) %>%
  add_recipe(win_recipe)

# elastic net logistic regression
enlog_wkflow <- workflow() %>%
  add_model(enlog_model) %>%
  add_recipe(win_recipe)

# k-nearest neighbors
knn_wkflow <- workflow() %>%
  add_model(knn_model) %>%
  add_recipe(win_recipe)

# support vector machine
svm_wkflow <- workflow() %>%
  add_model(svm_model) %>%
  add_recipe(win_recipe)

# random forest
rf_wkflow <- workflow() %>%
  add_model(rf_model) %>%
  add_recipe(win_recipe)

# lasso regresion
lasso_wkflow <- workflow() %>%
  add_model(lasso_model) %>%
  add_recipe(win_recipe)

# ridge regression
ridge_wkflow <- workflow() %>%
  add_model(ridge_model) %>%
  add_recipe(win_recipe)

# linear discriminant analysis
```

```
lda_wkflow <- workflow() %>%
  add_model(lda_model) %>%
  add_recipe(win_recipe)
```

3. Create a tuning grid to specify ranges of the hyperparameters and the levels of each

```
# elastic net
elastic_grid <- grid_regular(penalty(), mixture(range = c(0,1)), levels = 10)

# k-nearest neighbors
knn_grid <- grid_regular(neighbors(range = c(1, 100)), levels = 5)

# random forest
rf_grid <- grid_regular(mtry(range = c(1, 12)),
                        trees(range = c(1,100)), min_n(range = c(5,20)), levels = 8)

# lasso regression
penalty_grid <- grid_regular(penalty(range = c(-10,10)), levels = 50)

# ridge regression
# same as lasso

# support vector machine
svm_grid <- grid_regular(cost(), levels = 5)
```

4. Tune the model and specify the workflow, k-fold cross validation folds, and the tuning grid for our chosen parameters to tune.

```
# elastic net logistic regression
enlog_tune <- tune_grid(enlog_wkflow, resamples = game_folds, grid = elastic_grid)

# k-nearest neighbors
knn_tune <- tune_grid(knn_wkflow, resamples = game_folds, grid = knn_grid)

# random forest
rf_tune <- tune_grid(rf_wkflow, resamples = game_folds, grid = rf_grid)

# lasso regression
lasso_tune <- tune_grid(lasso_wkflow, resamples = game_folds, grid = penalty_grid)

# ridge regression
ridge_tune <- tune_grid(ridge_wkflow, resamples = game_folds, grid = penalty_grid)

# support vector machine
svm_tune <- tune_grid(svm_wkflow, resamples = game_folds, grid = svm_grid)

# linear discriminant analysis
lda_fit <- tune_grid(lda_wkflow,
                     resamples = game_folds)

# logistic regression
log_fit <- tune_grid(log_wkflow,
                     resamples = game_folds)
```

5. Save the tuned models to an RDS file to avoid rerunning the model.

```
# write_rds() to save

# logistic regression
write_rds(log_fit, file = "~/Documents/PSTAT 131/Final Project/data/tuned_models/log.rds")

# linear discriminant analysis
write_rds(lda_fit, file = "~/Documents/PSTAT 131/Final Project/data/tuned_models/lda.rds")

# elastic net logistic regression
write_rds(enlog_tune, file = "~/Documents/PSTAT 131/Final Project/data/tuned_models/elasticnet.rds")

# k-nearest neighbors
write_rds(knn_tune, file = "~/Documents/PSTAT 131/Final Project/data/tuned_models/knn.rds")

# random forest
write_rds(rf_tune, file = "~/Documents/PSTAT 131/Final Project/data/tuned_models/randomforest.rds")

# lasso regression
write_rds(lasso_tune, file = "~/Documents/PSTAT 131/Final Project/data/tuned_models/lasso.rds")

# ridge regression
write_rds(ridge_tune, file = "~/Documents/PSTAT 131/Final Project/data/tuned_models/ridge.rds")

# support vector machine
write_rds(svm_tune, file = "~/Documents/PSTAT 131/Final Project/data/tuned_models/svm.rds")
```

6. Load back in the saved files.

```
# logistic regression
log_tuned <- read_rds(file = "~/Documents/PSTAT 131/Final Project/data/tuned_models/log.rds")

# linear discriminant analysis
lda_tuned <- read_rds(file = "~/Documents/PSTAT 131/Final Project/data/tuned_models/lda.rds")

# elastic net logistic regression
enlog_tuned <- read_rds(file = "~/Documents/PSTAT 131/Final Project/data/tuned_models/elasticnet.rds")

# k-nearest neighbors
knn_tuned <- read_rds(file = "~/Documents/PSTAT 131/Final Project/data/tuned_models/knn.rds")

# random forest
rf_tuned <- read_rds(file = "~/Documents/PSTAT 131/Final Project/data/tuned_models/randomforest.rds")

# ridge regression
ridge_tuned <- read_rds(file = "~/Documents/PSTAT 131/Final Project/data/tuned_models/ridge.rds")

# lasso regression
lasso_tuned <- read_rds(file = "~/Documents/PSTAT 131/Final Project/data/tuned_models/lasso.rds")

# support vector machine
svm_tuned <- read_rds(file = "~/Documents/PSTAT 131/Final Project/data/tuned_models/svm.rds")
```

7. Select the most accurate model from the tuning grid, and then finalize the workflow with those specific tuning parameters.

```
# elastic net logistic regression
enlog_best <- select_best(enlog_tuned, metric = "roc_auc")
final_enlog_wkflow <- finalize_workflow(enlog_wkflow, enlog_best)
final_enlog_fit <- fit(final_enlog_wkflow, game_train)

# k-nearest neighbors
knn_best <- select_best(knn_tuned, metric = "roc_auc")
final_knn_wkflow <- finalize_workflow(knn_wkflow, knn_best)
final_knn_fit <- fit(final_knn_wkflow, game_train)

# random forest
rf_best <- select_best(rf_tuned, metric = "roc_auc")
final_rf_wkflow <- finalize_workflow(rf_wkflow, rf_best)
final_rf_fit <- fit(final_rf_wkflow, game_train)

# ridge regression
ridge_best <- select_best(ridge_tuned, metric = "roc_auc")
final_ridge_wkflow <- finalize_workflow(ridge_wkflow, ridge_best)
final_ridge_fit <- fit(final_ridge_wkflow, game_train)

# lasso regression
lasso_best <- select_best(lasso_tuned, metric = "roc_auc")
final_lasso_wkflow <- finalize_workflow(lasso_wkflow, lasso_best)
final_lasso_fit <- fit(final_lasso_wkflow, game_train)

# support vector machine
svm_best <- select_best(svm_tuned)
final_svm_wkflow <- finalize_workflow(svm_wkflow, svm_best)
final_svm_fit <- fit(final_svm_wkflow, game_train)

# logistic regression
log_best <- select_best(log_tuned, metric = "roc_auc")
final_log_wkflow <- finalize_workflow(log_wkflow, log_best)
final_log_fit <- fit(final_log_wkflow, game_train)

# linear discriminant analysis
lda_best <- select_best(lda_tuned, metric = "roc_auc")
final_lda_wkflow <- finalize_workflow(lda_wkflow, lda_best)
final_lda_fit <- fit(final_lda_wkflow, game_train)
```

Model Result

With all the models run, let's see which one performed the best.

```
log_reg_auc <- augment(final_log_fit, new_data = game_train) %>%
  roc_auc(blue_wins, estimate = .pred_0) %>%
  select(.estimate)

lda_auc <- augment(final_lda_fit, new_data = game_train) %>%
```

```
  roc_auc(blue_wins, estimate = .pred_0) %>%
  select(.estimate)

lasso_auc <- augment(final_lasso_fit, new_data = game_train) %>%
  roc_auc(blue_wins, estimate = .pred_0) %>%
  select(.estimate)

ridge_auc <- augment(final_ridge_fit, new_data = game_train) %>%
  roc_auc(blue_wins, estimate = .pred_0) %>%
  select(.estimate)

knn_auc <- augment(final_knn_fit, new_data = game_train) %>%
  roc_auc(blue_wins, estimate = .pred_0) %>%
  select(.estimate)

rf_auc <- augment(final_rf_fit, new_data = game_train) %>%
  roc_auc(blue_wins, estimate = .pred_0) %>%
  select(.estimate)

enlog_auc <- augment(final_enlog_fit, new_data = game_train) %>%
  roc_auc(blue_wins, estimate = .pred_0) %>%
  select(.estimate)

svm_auc <- augment(final_svm_fit, new_data = game_train) %>%
  roc_auc(blue_wins, estimate = .pred_0) %>%
  select(.estimate)

game_roc_aucs <- c(log_reg_auc$.estimate,
                           lda_auc$.estimate,
                           ridge_auc$.estimate,
                           lasso_auc$.estimate,
                           knn_auc$.estimate,
                           rf_auc$.estimate,
                           enlog_auc$.estimate,
                   svm_auc$.estimate)

game_mod_names <- c("Logistic Regression",
            "LDA",
            "Ridge",
            "Lasso",
            "K-Nearest Neighbors",
            "Random Forest",
            "Elastic Net",
            "Support Vector Machine")
```

```
game_results <- tibble(Model = game_mod_names,
                           ROC_AUC = game_roc_aucs)

game_results <- game_results %>%
  arrange(-game_roc_aucs)

game_results
```

```
## # A tibble: 8 x 2
##   Model                   ROC_AUC
##   <chr>                     <dbl>
## 1 Random Forest             0.876
## 2 K-Nearest Neighbors       0.840
## 3 Logistic Regression       0.817
## 4 Elastic Net               0.817
## 5 Support Vector Machine    0.817
## 6 Lasso                     0.817
## 7 LDA                       0.817
## 8 Ridge                     0.814
```

As we can see from the table, the random forest model performed the best, with an ROC AUC score of 0.8754. The next best performing model was the K-Nearest Neighbors model with an ROC AUC score of 0.8396. These results are obtained from the training data, so now it is time to test the model on the testing data. The model we will use for this is, obviously, the random forest model.

Results from the Best Model

Closer Look at the Best Performing Model

Let's take a look at which tuned parameters were chosen as the best random forest model.

```
show_best(rf_tuned, metric = "roc_auc") %>%
  select(-.estimator, .config) %>%
  slice(1)
```

```
## # A tibble: 1 x 8
##    mtry trees min_n .metric  mean     n std_err .config
##   <int> <int> <int> <chr>   <dbl> <int>   <dbl> <chr>
## 1     1    85    15 roc_auc 0.810    10 0.00372 Preprocessor1_Model369
```

Random Forest #339 with with 1 predictor, 85 trees, and minimum node size of 15 was selected as the best performing random forest model out of all random forest models.

Testing the Model

Now it is time to test our random forest model on data it has not seen: the testing data set we set aside before.

```
rf_roc_auc <- augment(final_rf_fit, new_data = game_test, type = 'prob') %>%
  roc_auc(blue_wins, .pred_0) %>%
  select(.estimate)

rf_roc_auc
```

```
## # A tibble: 1 x 1
##   .estimate
##       <dbl>
## 1     0.790
```

With a final ROC AUC score of 0.7912, we can say that our model did pretty well. An AUC value between 0.7 and 0.8 is generally considered an acceptable result. To get simpler explanation of our model results, we can also look at the model accuracy.

```
augment(final_rf_fit, new_data = game_test, type = 'prob') %>%
  accuracy(blue_wins, .pred_class) %>%
  select(.estimate)
```

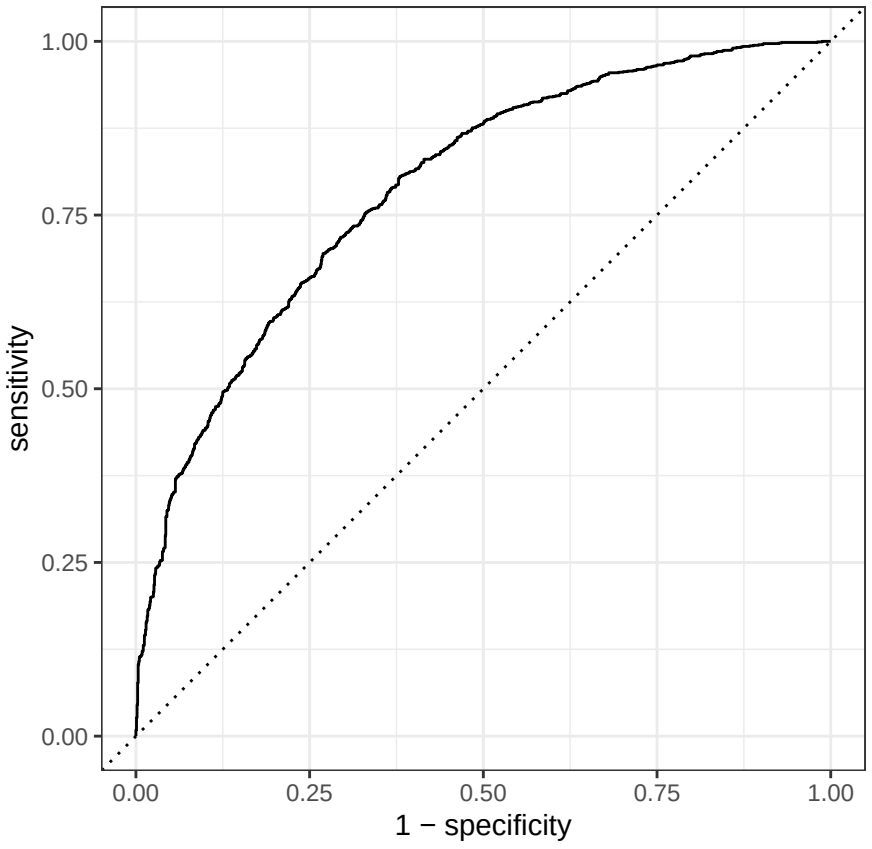
```
## # A tibble: 1 x 1
##   .estimate
##       <dbl>
## 1     0.710
```

The accuracy of our model is 0.7094, meaning it will predict the outcome correctly about 71% of the time. So while our model did well, it could've done better. However, as this is predicting the outcome of a game played by real humans where there are many unaccountable factors, it is quite impressive that we are able to predict the outcome relatively accurately, especially in the higher ranks.

ROC Curve

We can visualize our AUC score by plotting the curve. The higher up and to the left the curve is, the better the model's AUC.

```
augment(final_rf_fit, new_data = game_test, type = 'prob') %>%
  roc_curve(blue_wins, .pred_0) %>%
  autoplot()
```

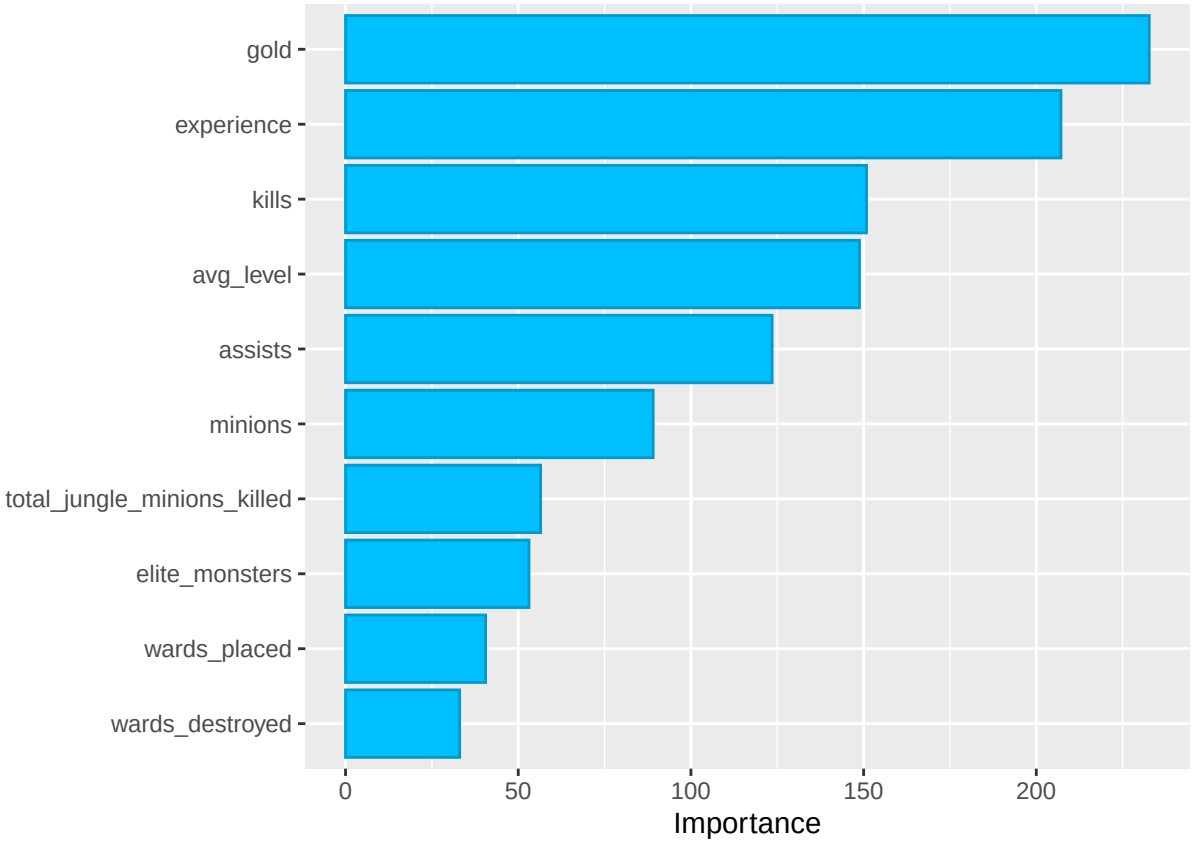


As seen above, while our curve does not look a right angle of a square, it still has a nice curve that confirms our ROC AUC score.

Variable Importance

As our best model was a random forest model, we can use a variable importance plot to see which variables were most important in predicting the outcome.

```
final_rf_fit %>%
  extract_fit_parsnip() %>%
  vip(aesthetics = list(color = "deepskyblue3", fill = "deepskyblue"))
```



From the graph, we can see that `gold` and `kills` mattered the most in predicting the outcome. In fact, the top 6 variables are the 6 most correlated variables which we saw in the correlation plot from the Visual EDA. These results make sense as gold allows your champion to become stronger through buying items and kills allow you to hinder the enemy as well as grant you gold.

Conclusion

Through extensive research, testing, and analysis, the best model to predict the outcome of a League Legends game at 10 minutes is a random forest model, although it was not perfect. This wasn't surprising as the random forest tends to be better for most data as it is more flexible due to it being nonparametric and making no assumptions about the outcome. The worst performing model was ridge regression. For next

steps, I could try implementing a Naive Bayes and Decision Tree model as these models could produce better results than the random forest.

Regarding the model, I think that the model's performance, while could be improved, is likely close to as accurate as it could be, simply due to the fact that it is predicting the outcome of a game played by humans. League of Legends is by nature an infuriating game. Players will sometimes play impossibly well at the start and play terrible at the end. Certain players play better under pressure and some players play worse when they start off poorly. Adding to this is that there is limited communication between teammates due to lack of voice chat. Consequently, crazy upsets can happen due to such factors. Human behavior and skill can never be fully quantified in data and can never be accounted for 100% of the time. Also, keep in mind that the model was built off data collected 10 minutes into the game. League games typically go for 30-40 minutes, so a variety of things could happen after 10 minutes that could change the flow of the game and, therefore, the outcome. Thus, I would say that our model accuracy of 70% is a big achievement.

Overall, this project provided a great opportunity to apply all the data science and machine learning skills that I have learned as well as incorporate my passion for gaming into the project.