Project League of Legends - Win Rate Prediction

Hong Wei

10 June 2020

1. Introduction

League of Legends (LoL) is one of world's most popular multiplayer online battle arena video games. The game was developed and released by Riot Games in 2009. As of August 2018, the game has a stunning active user base of over 111 million players.

1.1 About the game[1]

The basic of the game is as such: A team (blue or red) of 5 players can each choose one character (known as champions), to battle against another team of 5 players, in an online battle arena. The goal is to destroy the opponent's base, which is located at opposite end of the arena. Whichever team first destroys the opponent's base, wins the game. Champions begin each match at a low level, then slowly gain experience and level by killing non-player characters such as minions and monsters, or by killing enemy champions. Each game typically lasts 20 to 50 minutes.

[1]Information source: https://en.wikipedia.org/wiki/League_of_Legends

1.1.1 Game map of LoL

Image source: https://na.leagueoflegends.com/en-us/how-to-play/

1.1.2 A LoL gampeplay screenshot

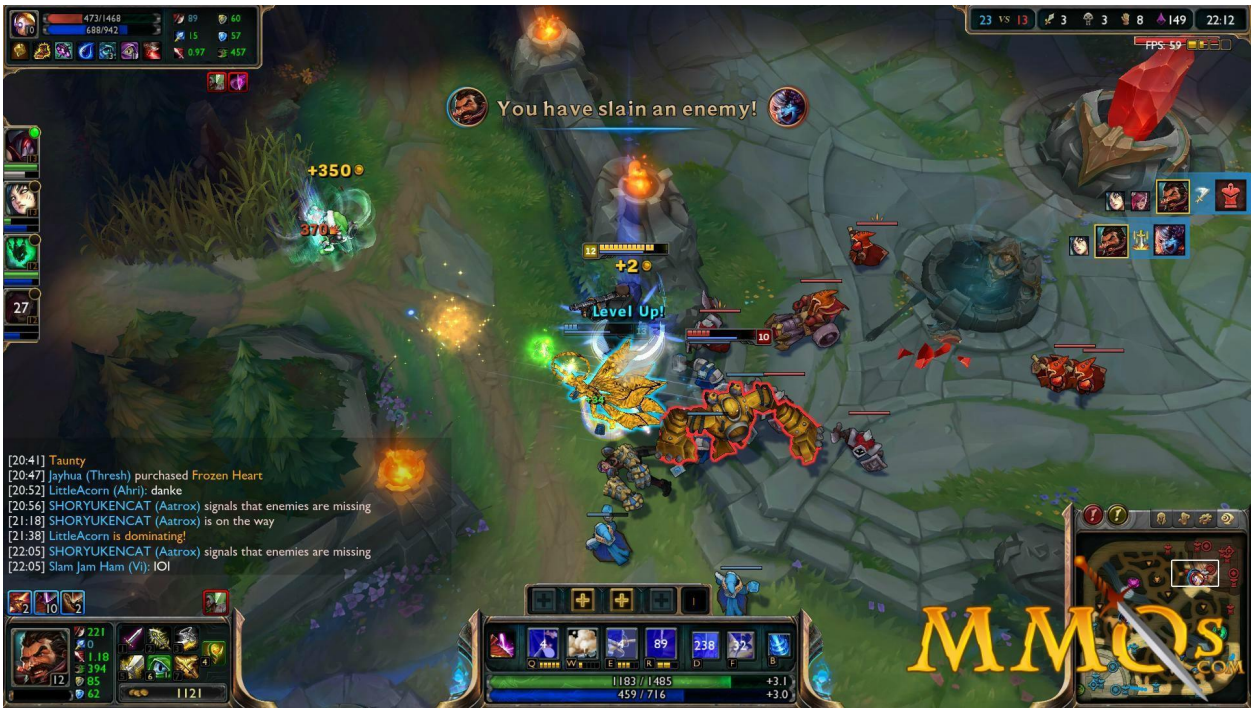


Image source: https://mmos.com/review/league-of-legends

1.2 About the dataset

We have obtained the dataset from https://www.kaggle.com/bobbyscience/league-of-legends-diamond-ranked-games-10-min. Utilizing the Riot Games API, the author has retrieved the game statistics at 10-minute mark and the final results for about 9.8k games of LoL between closely-ranked players. Each game is unique as identified by the `gameId`. The game statistics are 19 features per team as at the 10-minute mark, such as kills, deaths, gold and experience.

1.3 Objective and Goal

The objective of this project is to construct a classification task using the machine learning techniques and other tools which we have learned throughout the HarvardX Data Science Professional Certificate Programme.

Our task will be to predict the win rate of a team using the features as at 10 minute into a game. Throughout this project, we will start by preprocessing the dataset and then conduct some data exploration. We will use several machine learning algorithms to decide on the final model. Lastly we will evaluate our final model on an untouched `validation` set.

Our minimum expectation is to reach an accuracy of prediction of at least 50%, and our goal is 70% using our final model.

---

2. Data Preprocessing

2.1 Overview of data

In this code, we load the libraries required for our work, and the dataset from the `games.csv` file which is uploaded to my Github, and also included together in the project submssion. The dataset was downloaded from the source and yet to be edited.

```
# Load libraries
library(tidyverse)
library(caret)
library(corrplot)
library(stringr)
library(rpart)
library(gbm)

# Load data file from github
dl <- tempfile()
download.file("https://github.com/limhongwei/lol/raw/master/games.csv", dl)
games <- read.csv(dl, stringsAsFactors = FALSE)
```

A quick look at the data:

```
# Brief look at dataset
glimpse(games)
```

```
## Rows: 9,879
## Columns: 40
## $ gameId                      <dbl> 4519157822, 4523371949, 4521474530, 45...
## $ blueWins                    <int> 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1,...
## $ blueWardsPlaced             <int> 28, 12, 15, 43, 75, 18, 18, 16, 16, 13...
## $ blueWardsDestroyed          <int> 2, 1, 0, 1, 4, 0, 3, 2, 3, 1, 3, 2, 1,...
## $ blueFirstBlood              <int> 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1,...
## $ blueKills                   <int> 9, 5, 7, 4, 6, 5, 7, 5, 7, 4, 4, 11, 7...
## $ blueDeaths                  <int> 6, 5, 11, 5, 6, 3, 6, 13, 7, 5, 4, 11,...
## $ blueAssists                 <int> 11, 5, 4, 5, 6, 6, 7, 3, 8, 5, 6, 7, 1...
## $ blueEliteMonsters           <int> 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1,...
## $ blueDragons                 <int> 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1,...
## $ blueHeralds                 <int> 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0,...
## $ blueTowersDestroyed         <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
## $ blueTotalGold               <int> 17210, 14712, 16113, 15157, 16400, 158...
## $ blueAvgLevel                <dbl> 6.6, 6.6, 6.4, 7.0, 7.0, 7.0, 6.8, 6.4...
## $ blueTotalExperience         <int> 17039, 16265, 16221, 17954, 18543, 181...
## $ blueTotalMinionsKilled      <int> 195, 174, 186, 201, 210, 225, 225, 209...
## $ blueTotalJungleMinionsKilled <int> 36, 43, 46, 55, 57, 42, 53, 48, 61, 39...
## $ blueGoldDiff                <int> 643, -2908, -1172, -1321, -1004, 698, ...
## $ blueExperienceDiff          <int> -8, -1173, -1033, -7, 230, 101, 1563, ...
## $ blueCSPerMin                <dbl> 19.5, 17.4, 18.6, 20.1, 21.0, 22.5, 22...
## $ blueGoldPerMin              <dbl> 1721.0, 1471.2, 1611.3, 1515.7, 1640.0...
## $ redWardsPlaced              <int> 15, 12, 15, 15, 17, 36, 57, 15, 15, 16...
## $ redWardsDestroyed           <int> 6, 1, 3, 2, 2, 5, 1, 0, 2, 2, 2, 1, 1,...
## $ redFirstBlood               <int> 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0,...
## $ redKills                    <int> 6, 5, 11, 5, 6, 3, 6, 13, 7, 5, 4, 11,...
```

```
## $ redDeaths                    <int> 9, 5, 7, 4, 6, 5, 7, 5, 7, 4, 4, 11, 7...
## $ redAssists                   <int> 8, 2, 14, 10, 7, 2, 9, 11, 5, 4, 5, 9,...
## $ redEliteMonsters             <int> 0, 2, 0, 0, 1, 0, 0, 1, 2, 0, 1, 0, 0,...
## $ redDragons                   <int> 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0,...
## $ redHeralds                   <int> 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,...
## $ redTowersDestroyed           <int> 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
## $ redTotalGold                 <int> 16567, 17620, 17285, 16478, 17404, 152...
## $ redAvgLevel                  <dbl> 6.8, 6.8, 6.8, 7.0, 7.0, 7.0, 6.4, 6.6...
## $ redTotalExperience           <int> 17047, 17438, 17254, 17961, 18313, 180...
## $ redTotalMinionsKilled        <int> 197, 240, 203, 235, 225, 221, 164, 157...
## $ redTotalJungleMinionsKilled  <int> 55, 52, 28, 47, 67, 59, 35, 54, 53, 43...
## $ redGoldDiff                  <int> -643, 2908, 1172, 1321, 1004, -698, -2...
## $ redExperienceDiff            <int> 8, 1173, 1033, 7, -230, -101, -1563, 8...
## $ redCSPerMin                  <dbl> 19.7, 24.0, 20.3, 23.5, 22.5, 22.1, 16...
## $ redGoldPerMin                <dbl> 1656.7, 1762.0, 1728.5, 1647.8, 1740.4...
```

It shows a dataset of 9879 games, with 19 features of Blue and Red Teams. `blueWins` is `1` if the blue team wins, `0` otherwise.

2.2 Glossary of features

Feature	Description
Warding totem	An item that a player can put on the map to reveal the nearby area. It can be placed and/or destroyed by opponent.
Minions	NPC (non-player character) that belong to both teams. They give gold when killed by players.
Jungle minions	NPC that belong to NO TEAM. They give gold and buffs when killed by players.
Elite monsters	Monsters with high hp/damage that give a massive bonus (gold/XP/stats) when killed by a team.
Dragons	Elite monster which gives team bonus when killed.
Herald	Elite monster which gives stats bonus when killed by the player. It helps to push a lane and destroys structures.
Towers	Structures you have to destroy to reach the enemy Nexus. They give gold.
Level	Champion level. Start at 1. Max is 18.
Kill/Death/Assist	KDA count of the team when battling enemy players.

2.3 Preprocessing Plan

After examining the database structure briefly, here is our plan for preprocessing:

- Check for NA values, if any.
- The dataset is considered to be in wide table format - where Blue team features and Red team features are the mostly the same (except for the lack of "redWins"), but separated in columns. So we will need to reshape the data so that the columns are the predictors that we want to use.
- Remove irrelevant variables, and derived variables.
- Remove near zero variance variables, if any.

2.3.1 Check NA values

Using this code:

```
# Check NA values
sum(sapply(1:ncol(games), function(x) sum(is.na(games[,x]))))
```

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

The result shows that there is no NA values.

2.3.2 Reshape data

In this code, we reshape the data by so that it is in a tidy format for our machine learning.

```
# Tidy data
temp <- games %>%
  mutate(redWins = ifelse(blueWins==1,0,1)) %>%
  gather(key, value, -gameId)
temp$key <- str_replace_all(temp$key, "blue", "blue_")
temp$key <- str_replace_all(temp$key, "red", "red_")
games_clean <- temp %>%
  separate(key, c("team", "variable"), "_") %>%
  spread(variable, value)
```

Here is how the data looks like after reshaping:

```
# Brief look at tidy data
glimpse(games_clean)
```

```
## Rows: 19,758
## Columns: 22
## $ gameId                   <dbl> 4295358071, 4295358071, 4296004784, 429600...
## $ team                     <chr> "blue", "red", "blue", "red", "blue", "red...
## $ Assists                  <dbl> 13, 7, 6, 2, 10, 10, 14, 2, 6, 8, 0, 11, 8...
## $ AvgLevel                 <dbl> 6.6, 7.0, 7.4, 6.4, 7.2, 6.8, 7.4, 6.6, 6....
## $ CSPerMin                 <dbl> 19.5, 21.1, 26.3, 23.2, 20.7, 25.2, 23.4, ...
## $ Deaths                   <dbl> 6, 10, 2, 6, 7, 9, 3, 12, 6, 6, 6, 4, 2, 6...
## $ Dragons                  <dbl> 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, ...
## $ EliteMonsters            <dbl> 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 2, 0, 1, ...
## $ ExperienceDiff           <dbl> -797, 797, 3092, -3092, 261, -261, 2399, -...
## $ FirstBlood               <dbl> 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, ...
## $ GoldDiff                 <dbl> 1864, -1864, 2944, -2944, 339, -339, 4188,...
## $ GoldPerMin               <dbl> 1834.8, 1648.4, 1735.4, 1441.0, 1710.2, 16...
## $ Heralds                  <dbl> 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, ...
## $ Kills                    <dbl> 10, 6, 6, 2, 9, 7, 12, 3, 6, 6, 4, 6, 6, 2...
## $ TotalExperience          <dbl> 17087, 17884, 19609, 16517, 18327, 18066, ...
## $ TotalGold                <dbl> 18348, 16484, 17354, 14410, 17102, 16763, ...
## $ TotalJungleMinionsKilled <dbl> 52, 59, 64, 44, 46, 45, 40, 52, 44, 53, 55...
## $ TotalMinionsKilled       <dbl> 195, 211, 263, 232, 207, 252, 234, 207, 22...
## $ TowersDestroyed          <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
## $ WardsDestroyed           <dbl> 6, 4, 5, 6, 5, 4, 5, 1, 5, 2, 2, 6, 2, 4, ...
## $ WardsPlaced              <dbl> 17, 19, 19, 21, 15, 18, 16, 19, 17, 19, 18...
## $ Wins                     <dbl> 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, ...
```

2.3.2 Remove irrelevant variables, and derived variables

First of all, `gameId` is an identifier of a game, therefore it is irrelevant to the win rate.

The variable `team` can be removed too. Choosing "blue" or "red" should not have effect on the win rate, since the probability of winning should be 0.5. This is confirmed by:

```
# Win rate of choosing Blue team should be 0.5
mean(games$blueWins==1)
```

```
## [1] 0.4990384
```

Examining the dataset columns and the first few values, we observe that these few variables are derived from other variables in the same dataset:

- GoldPerMin = TotalGold ÷ 10 minutes
- CsPerMin = TotalMinionsKilled ÷ 10 minutes
- We suspect that EliteMonsters equals to the sum of Dragons and Heralds since both of them are elite monsters

```
# Check derived variables
games_clean %>%
  mutate(count1 = GoldPerMin - TotalGold/10,
         count2 = CSPerMin - TotalMinionsKilled/10,
         count3 = EliteMonsters - Dragons - Heralds) %>%
  summarize(count1 = sum(count1),
            count2 = sum(count2),
            count3 = sum(count3))
```

```
##   count1 count2 count3
## 1      0      0      0
```

The summary of all counts showing zeroes, indicates the 3 variables above are derived and therefore redundant.

2.3.3 Remove near zero variance variables

We also use the `nearZeroVar` function to check if there is any variable which has very few unique values relative to the number of samples, and the ratio of the frequency of the most common value to the frequency of the second most common value is large.

```
# Check near zero variance variables
nearZeroVar(games_clean, saveMetrics = TRUE)
```

```
##                          freqRatio percentUnique zeroVar   nzv
## gameId                    1.000000   50.00000000   FALSE FALSE
## team                      1.000000    0.01012248   FALSE FALSE
## Assists                   1.053919    0.15183723   FALSE FALSE
## AvgLevel                  1.092884    0.09616358   FALSE FALSE
## CSPerMin                  1.005222    0.81485980   FALSE FALSE
## Deaths                    1.010707    0.10628606   FALSE FALSE
```

```
## Dragons                    1.580384    0.01012248   FALSE FALSE
## EliteMonsters              1.229823    0.01518372   FALSE FALSE
## ExperienceDiff             1.000000   36.11195465   FALSE FALSE
## FirstBlood                 1.000000    0.01012248   FALSE FALSE
## GoldDiff                   1.000000   42.64095556   FALSE FALSE
## GoldPerMin                 1.153846   30.53952829   FALSE FALSE
## Heralds                    4.746946    0.01012248   FALSE FALSE
## Kills                      1.010707    0.10628606   FALSE FALSE
## TotalExperience            1.066667   25.85788035   FALSE FALSE
## TotalGold                  1.153846   30.53952829   FALSE FALSE
## TotalJungleMinionsKilled   1.000000    0.40489928   FALSE FALSE
## TotalMinionsKilled         1.005222    0.81485980   FALSE FALSE
## TowersDestroyed           23.741206    0.02530621   FALSE  TRUE
## WardsDestroyed             1.153549    0.13665351   FALSE FALSE
## WardsPlaced                1.013174    0.89583966   FALSE FALSE
## Wins                       1.000000    0.01012248   FALSE FALSE
```

The table shows that `TowersDestroyed` has too few unique values, and might adversely affect the machine learning process.

Here we remove the variables mentioned above, and also change the class of `Wins` to a factor class.

```
# Remove irrelevant, derived and near zero variance variables
games_clean <- games_clean %>%
  select(-gameId, -team, -GoldPerMin, -CSPerMin, -EliteMonsters, -TowersDestroyed) %>%
  mutate(Wins = factor(Wins))
```

After data preprocessing, our dataset now has 1 dependent variable `Wins` (win rate), and 15 independent variables (predictors). We are now ready for data exploration in the next section.

```
# Brief look at data ready for exploration
str(games_clean)
```

```
## 'data.frame':    19758 obs. of  16 variables:
##  $ Assists                : num  13 7 6 2 10 10 14 2 6 8 ...
##  $ AvgLevel               : num  6.6 7 7.4 6.4 7.2 6.8 7.4 6.6 6.6 6.8 ...
##  $ Deaths                 : num  6 10 2 6 7 9 3 12 6 6 ...
##  $ Dragons                : num  0 1 1 0 1 0 1 0 0 0 ...
##  $ ExperienceDiff         : num  -797 797 3092 -3092 261 ...
##  $ FirstBlood             : num  0 1 1 0 1 0 1 0 1 0 ...
##  $ GoldDiff               : num  1864 -1864 2944 -2944 339 ...
##  $ Heralds                : num  0 0 0 0 0 1 0 1 0 0 ...
##  $ Kills                  : num  10 6 6 2 9 7 12 3 6 6 ...
##  $ TotalExperience        : num  17087 17884 19609 16517 18327 ...
##  $ TotalGold              : num  18348 16484 17354 14410 17102 ...
##  $ TotalJungleMinionsKilled: num  52 59 64 44 46 45 40 52 44 53 ...
##  $ TotalMinionsKilled     : num  195 211 263 232 207 252 234 207 220 216 ...
##  $ WardsDestroyed         : num  6 4 5 6 5 4 5 1 5 2 ...
##  $ WardsPlaced            : num  17 19 19 21 15 18 16 19 17 19 ...
##  $ Wins                   : Factor w/ 2 levels "0","1": 2 1 2 1 1 2 1 2 2 1 ...
```

3. Data Exploration

3.1 Splitting the data

Before we perform any data exploration, we first split the data into a `main` set and a `validation` set. This is to ensure that the **validation set is not being used to form any opinion or conclusion during the data exploration or model training process**. The validation set will only be used for the evaluation of the final algorithm.

We choose a ratio of 80/20 as we think that the dataset is not exactly too large, and there are data on all observations (unlike a sparse dataset). So, we allocate more room to the validation set. The 80% `main` set will be further split into 90/10 for model training. In short, the dataset will be split into:

- 72% for training
- 8% for testing
- 10% for validation

```
# First split of data into 80/20
set.seed(2020, sample.kind = "Rounding")
# if using R 3.5 or earlier, use `set.seed(2020)` instead
test_index <- createDataPartition(games_clean$Wins, times = 1, p = 0.2, list = FALSE)
main <- games_clean[-test_index,]
validation <- games_clean[test_index,]
```

We will be using the `main` set throughout this section and next section (Model Training).

3.2 Win rate

As expected, the win rate should be 0.5.

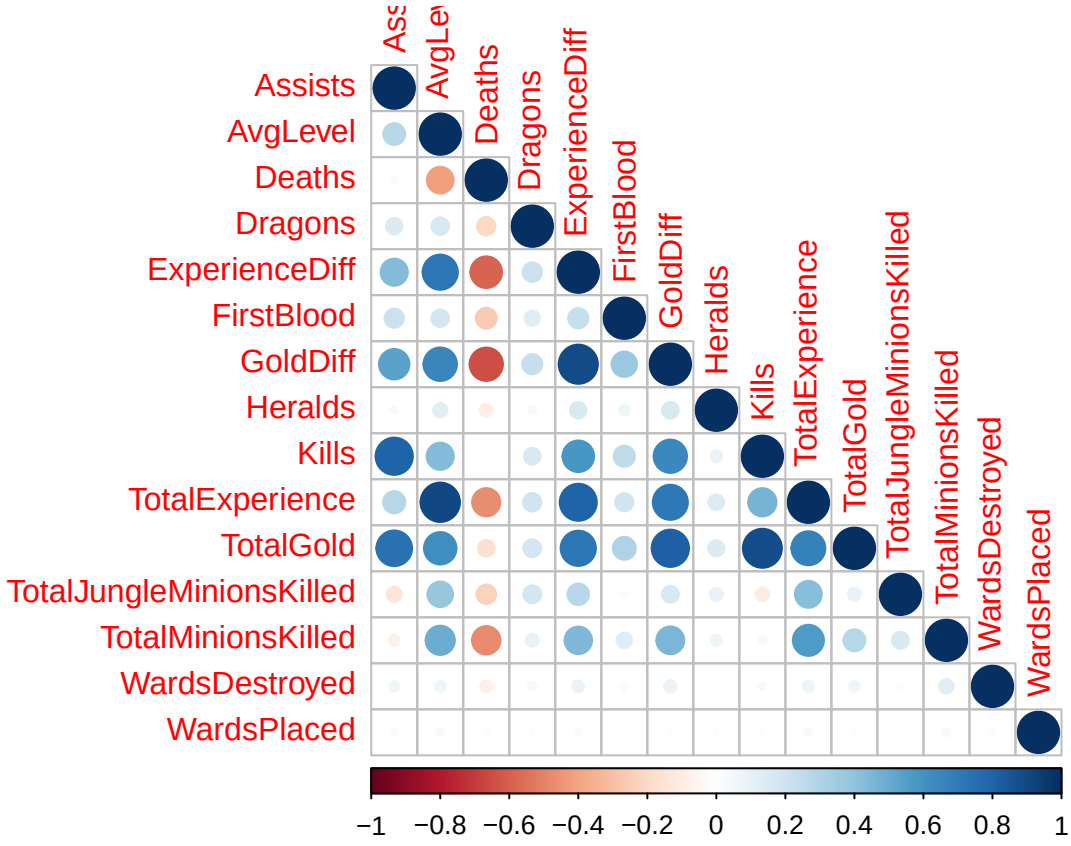
```
# Win rate of choosing either Blue or Red should be 0.5
mean(main$Wins==1)
```

```
## [1] 0.5
```

3.3 Correlations

Let's generate a correlation plot between the predictors.

```
# Correlation among predictors
temp <- main %>% select(-Wins)
tempcor <- cor(temp)
corrplot::corrplot(tempcor, type = "lower")
```

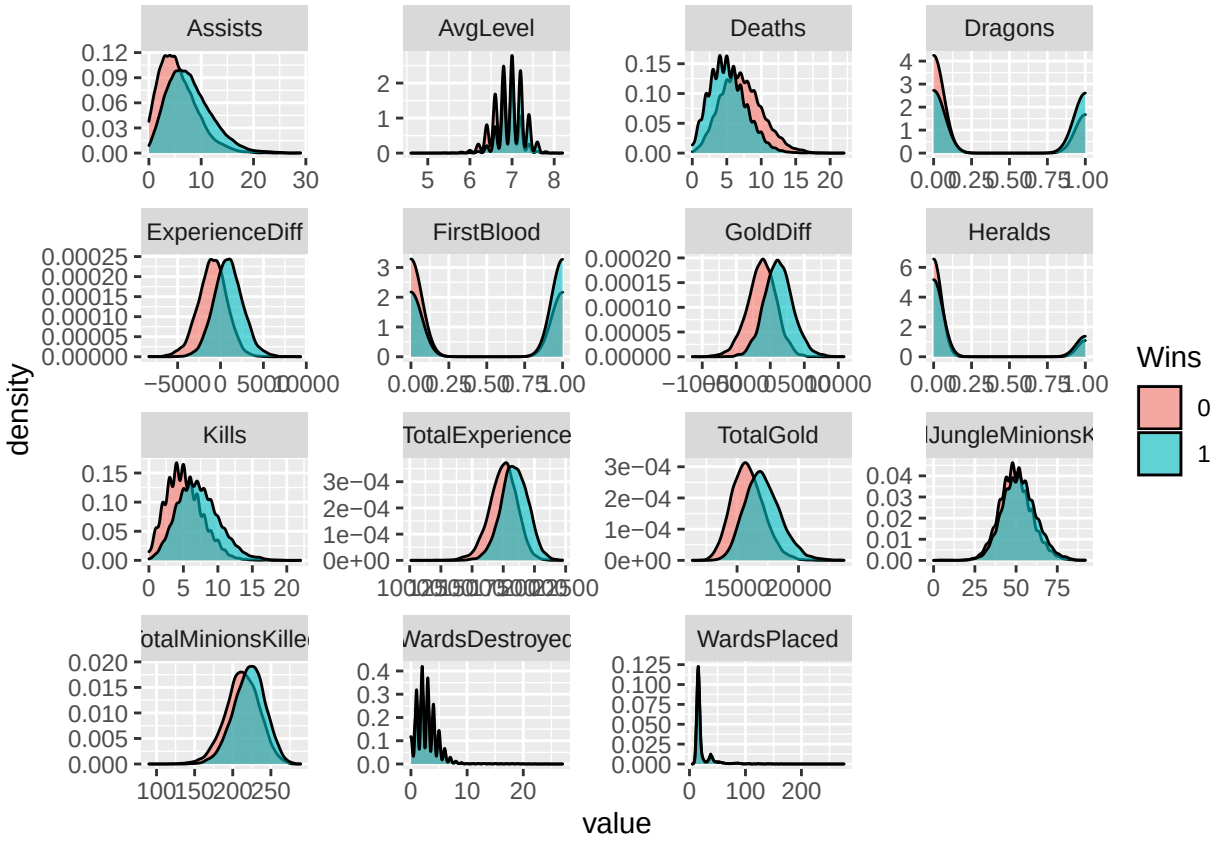


`Kills` and `TotalGold` have high positive correlation since the more kills the team get, the more gold the team will receive. `GoldDiff` and `Deaths` have high negative correlation too.

3.4 Distributions

We also take a look at the distribution of all the predictors.

```
# Distribution of preodictors
main %>% gather(key, value, -Wins) %>%
  ggplot(aes(value, fill = Wins)) +
  geom_density(aes(y = ..density..), kernel = "gaussian", alpha = 0.6) +
  facet_wrap(~key, scales = "free")
```



`WardsPlaced` and `WardsDestroyed` seem to be very skewed to the left. The range of `WardsPlaced` seems to be quite far-fetched. We have some doubts as whether a team can place more than 200 wards within 10 minutes into a game. The range of `WardsDestroyed` does not seem to match the wide range of `WardsPlaced` either. There could be some errors in these two variables. We will need to relook into this if we have the chance to reach out to an LoL player.

```
# Range of 'WardsPlaced' and 'WardsDestroyed' seem to be far-fetched
range(main$WardsPlaced)
```

```
## [1]   5 276
```

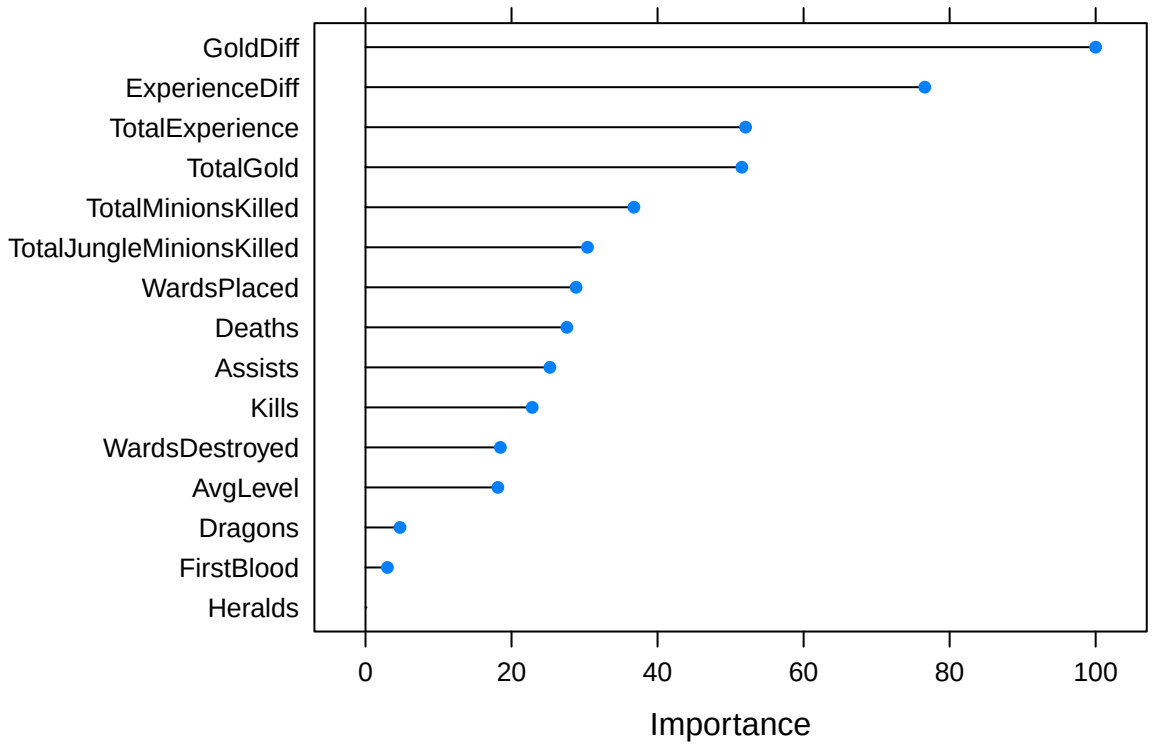
```
range(main$WardsDestroyed)
```

```
## [1]  0 27
```

3.5 Variable importance

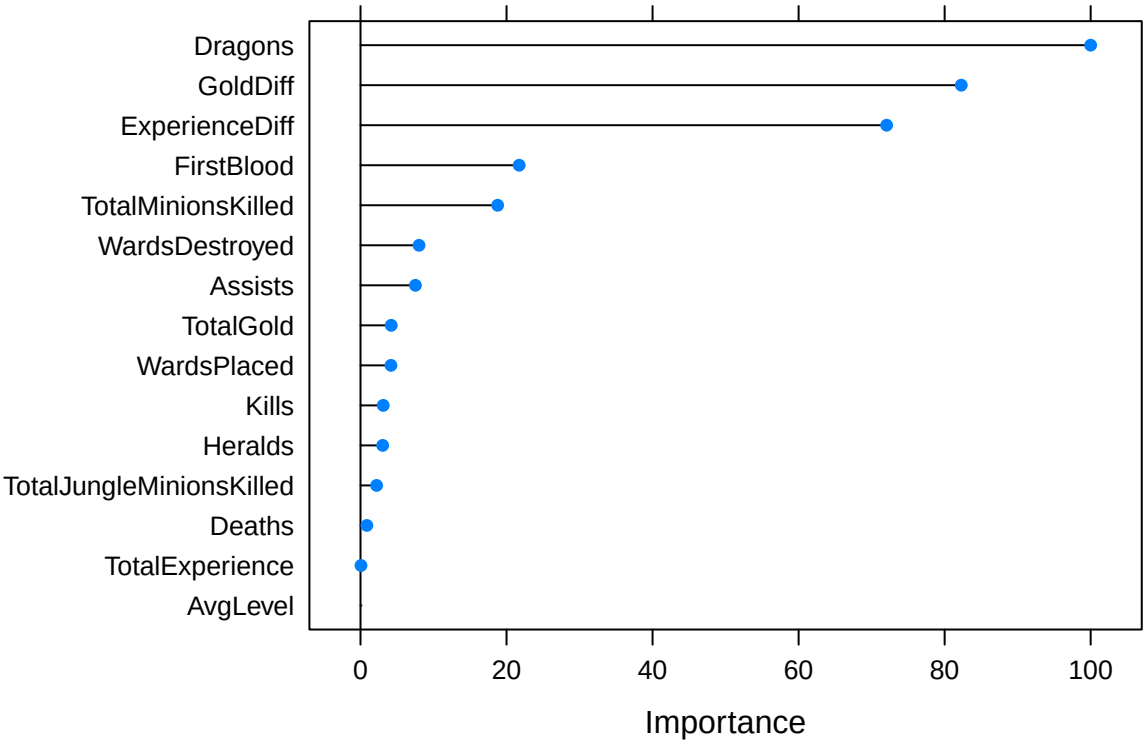
We utilise the `varImp` function in the `caret` package to help us to gauge which are the most important features using different algorithms.

```
# Variable importance using Random Forest
main_rf <- train(Wins ~ ., method = "rf",
                 data = main,
                 ntree = 100)
plot(varImp(main_rf), main = "Variable Importance using Random Forest")
```



For comparison, we also use Logistic Regression to see if any variable stands out in both algorithms.

```
# Variable importance using Logistic Regression
main_glm <- train(Wins ~ .,
                  method = "glm",
                  data = main)
plot(varImp(main_glm), main = "Variable Importance using Logistic Regression")
```



`GoldDiff` and `ExperienceDiff` come out top in both algorithms.

---

4. Model Training

In this section, we begin to train our data using a variety of models. We choose some models suitable for classification task.

4.1 Second split

Here we perform a second split of `main` set into 90/10 for `train_set` and `test_set`.

```
# Second split of 90/10 for model training
set.seed(2020, sample.kind = "Rounding")
# if using R 3.5 or earlier, use `set.seed(2020)` instead
test_index <- createDataPartition(main$Wins, times = 1, p = 0.1, list = FALSE)
train_set <- main[-test_index,]
test_set <- main[test_index,]
```

4.2 Model Training

We choose a variety of models (11 models suitable for classification task) here:

```
# Model training of train_set vs test_set
models <- c("naive_bayes", "gamLoess", "lda", "qda", "glm", "glmboost",
            "pls", "multinom", "knn", "gbm", "rpart")
```

In this code, we use the `train_set` to fit the models. We will use all the predictors in the data.

```
fits <- lapply(models, function(model){
  set.seed(2020, sample.kind = "Rounding")
  # if using R 3.5 or earlier, use `set.seed(2020)` instead
  print(model)
  train(Wins ~ ., method = model, data = train_set)
})
names(fits) <- models
pred <- sapply(fits, function(object) predict(object, newdata = test_set))
```

Then we predict the win rates for `test_set`. We extract the **Accuracy**, **Sensitivity**, and **Specificity** values for each model, which we will use to measure the performance of the model.

```
# Generate output of model training
out <- sapply(seq(1,length(models),1), function(x){
  cm <- confusionMatrix(factor(pred[,x]), test_set$Wins)
  acc <- cm$overall["Accuracy"]
  sens <- cm$byClass["Sensitivity"]
  spec <- cm$byClass["Specificity"]
  return(c(acc, sens, spec))
})
```

We consolidate the results using this code:

```
# Generate results of model training
labels <- sapply(seq(1,length(models),1), function(x) getModelInfo(models[x])[[1]]$label)
results <- tibble(Model = models, Label = labels)
temp <- out %>% t %>% as.data.frame()
results <- cbind(results, temp)
```

Here is the results:

Table 2: Results of Model Training

Model	Label	Accuracy	Sensitivity	Specificity
naive_bayes	Naive Bayes	0.7300885	0.7218710	0.7383059
gamLoess	Generalized Additive Model using LOESS	0.7338812	0.7446271	0.7231353
lda	Linear Discriminant Analysis	0.7357775	0.7458913	0.7256637
qda	Quadratic Discriminant Analysis	0.7117573	0.7319848	0.6915297
glm	Bayesian Generalized Linear Model	0.7383059	0.7471555	0.7294564
glmboost	Boosted Generalized Linear Model	0.7439949	0.7446271	0.7433628
pls	Generalized Partial Least Squares	0.7383059	0.7496839	0.7269279
multinom	Penalized Multinomial Regression	0.7383059	0.7471555	0.7294564
knn	k-Nearest Neighbors	0.6959545	0.6788875	0.7130215
gbm	Gradient Boosting Machines	0.7376738	0.7458913	0.7294564
rpart	CART	0.7357775	0.7496839	0.7218710

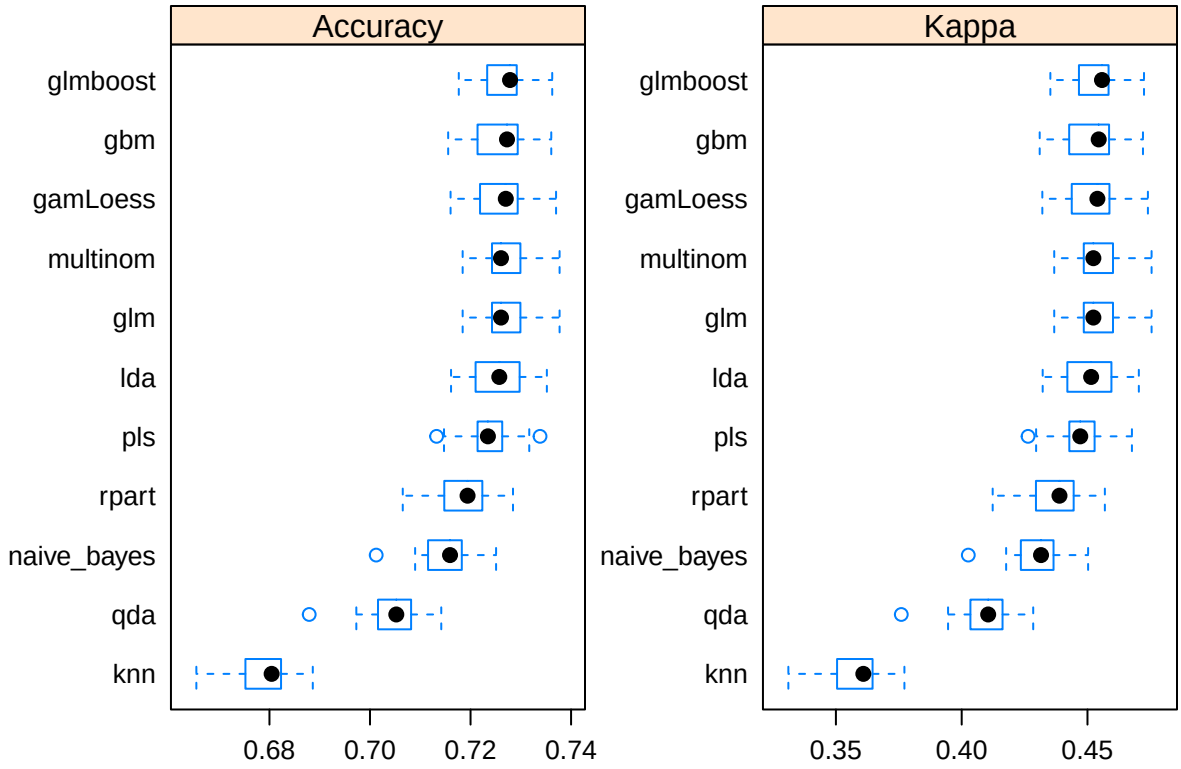
The mean accuracy is:

```
mean(results$Accuracy)
```

```
## [1] 0.730893
```

Let's do a quick comparison plot between the models:

```
# Comparing models
models_compare <- resamples(fits)
scales <- list(x=list(relation="free"), y=list(relation="free"))
bwplot(models_compare, scales=scales)
```



It seems that logistic regression models such as `glmboost`, `gbm` and `gamLoess` have higher accuracies compared to other models.

4.3 Ensemble

Instead of choosing a single model, we will use an ensemble technique for our final model. Ensemble allows us to combine two or more models and it should make our final model to be more robust and less likely to be biased.

4.3.1 Ensemble with average

This approach is basically taking the average of the predictions of all the models we used in model training. The accuracy of this ensemble method is simply the average accuracy.

```
# Ensemble by Average
acc_avg <- mean(results$Accuracy)
acc_avg
```

```
## [1] 0.730893
```

4.3.2 Ensemble with majority vote

A majority vote approach allows us to take recommendation from multiple models prediction. Using the following code, we construct an ensemble by majority vote where prediction of `Wins` = 0 if more than 50%

of the models predict 0, and vice versa.

```
# Ensemble by Majority Vote
votes <- rowMeans(pred=="0")
y_hat <- ifelse(votes > 0.5, "0", "1")
acc_mv <- mean(y_hat == test_set$Wins)
acc_mv
```

```
## [1] 0.7383059
```

The accuracy of ensemble method 0.7383059 shows an improvement from the first ensemble approach above.

4.3.3 Ensemble with stacking

Stacking is an interesting ensemble approach where we perform a second layer of training on top of the first stage of model training. We choose to use a logistic regression model `glm` in the second layer.

In order for our logistic regression model at the top layer to work, we will need to remove the highly correlated models among the 11 models during our model training. We use a cutoff correlation level of 0.75. Here are the models after the cutoff:

```
# Remove models with more than 0.75 correlation
a <- modelCor(models_compare)
x <- findCorrelation(a, cutoff = .75)
selected <- colnames(a[,-x])
selected
```

```
## [1] "naive_bayes" "qda"         "knn"         "rpart"
```

The approach to ensemble stacking:

1. First we fit `selected` models on the `train_set`.
2. We add the predicted values into the `train_set`, let's call it `stack_train`.
3. We also add the predicted values (already done in model training) into `test_set`, let's call it `stack_test`.
4. Now we fit another model (we use a logistic regression model `glm`) to `stack_train`, where the predicted values in step 2 are now the predictors.
5. We use the model trained in step 4, to make prediction on `stack_test`.

We achieve the steps above using the following code:

```
# Ensemble by Stacking
pred_train <- sapply(fits[selected], function(object) predict(object, newdata = train_set))
stack_train <- train_set %>% cbind(pred_train)
stack_test <- test_set %>% cbind(pred[,selected])
set.seed(2020, sample.kind = "Rounding")
# if using R 3.5 or earlier, use `set.seed(2020)` instead
model_glm <- train(stack_train[selected], stack_train$Wins, method = "glm")
y_hat_stack <- predict(model_glm, stack_test[selected])
acc_stack <- mean(y_hat_stack==stack_test$Wins)
acc_stack
```

```
## [1] 0.6959545
```

Let's look at our result so far:

```
# Result of different ensembles
ensem <- tibble(Method = c("Ensemble by Average",
                           "Ensemble by Majority Vote",
                           "Ensemble by Stacking"),
                Accuracy = c(acc_avg, acc_mv, acc_stack))
```

Table 3: Results of Ensemble Approach

Method	Accuracy
Ensemble by Average	0.7308930
Ensemble by Majority Vote	0.7383059
Ensemble by Stacking	0.6959545

4.4 Final model

While all of three ensemble approaches each yields accuracy that is much higher than 50%, we will choose the **Ensemble with Majority Vote** as our final model. It is a more balanced model which considers all 11 models in the model training, and it also gives the highest accuracy among the three approaches.

---

5. Result

We are now ready to evaluate our final model on the `validation` set to see how it performs. To be on the conservative side, we remain to use the `train_set` (which is a smaller set than `main` set) to train the predictors.

Here is the code:

```
# Test final model using validation set
fits <- lapply(models, function(model){
  set.seed(2020, sample.kind = "Rounding")
  # if using R 3.5 or earlier, use `set.seed(2020)` instead
  print(model)
  train(Wins ~ ., method = model, data = validation)
})
names(fits) <- models
pred <- sapply(fits, function(object) predict(object, newdata = validation))

# Ensemble with majority vote on validation set
votes <- rowMeans(pred=="0")
y_hat <- ifelse(votes > 0.5, "0", "1")
acc_val <- mean(y_hat == validation$Wins)
acc_val
```

Our final model on validation yields an accuracy of 0.7416498, and it is above our target of 70%.

---

6. Conclusion

6.1 Summary

We started the project with a dataset with near 40 predictors. We have made considerable effort to reshape the data, and we use our judgement to remove variables which are less likely to help with our predictions. We also explore the data using different approaches - correlations, distributions and variable importance. Then we use a variety of classification models to form our ensemble model. After testing several ensemble approaches, we decide to use **Ensemble with Majority Voting** as our final model. Upon testing on the untouched `validation` set, our final result of 0.7416498 exceeds our goal of 70%.

6.2 Comments

Throughout our model training, we try to achieve an accuracy as high as possible without overfitting. While we are satisfied with the accuracy that we achieve in this project, we believe that our final model can be improved if we could collect more data from the Riot Games API. Some of the game statistics that could be helpful are heroes choice, minimum and maximum hero level of a team etc.

When it comes to model training, we did not include or use any individual model tuning. This is because when we tried to tune some of the models such as k-nearest neighbour and random forest, the accuracy did not improve by much and the computation time is too long.

Finally, I just want to say that I have gained significant knowledge and experience through this project and the courseworks. I am also glad that I were given the opportunity to apply this knowledge on real-world data and potentially meaning application.

---

7. Acknowledgement

Special thanks to the sources:

- Michel's fanboi from Kaggle https://www.kaggle.com/bobbyscience/league-of-legends-diamond-ranked-games-10-min
- League of Legends Wikipedia Page https://en.wikipedia.org/wiki/League_of_Legends
- HarvardX Data Science Professional Certificate Programme by edx
- Professor Rafael Irizarry https://rafalab.github.io/dsbook/index.html