League of Legens Win Prediction - Chose your own - edx Capstone

Daniel Obermeier

2024-05-30

Executive Summary

In this project, we build a machine learning classifier that helps us to predict the winner in an online game called "League of Legends." League of Legends (LoL) is one of the most popular competitive online games in which professional Esports player regularly compete for high monetary prizes. The goal of our analysis is not only to predict the winner but also to understand which features help to predict the win to learn about strategies that increase the chances of winning the game. Such insights are valuable for professional player who play the game for a winning but can also inform casual players on how to do better.

The data we use is a sample of almost 10,000 competitive games. The features in our data set are different performance indicators measures after 10 minutes. Therefore, our analysis seeks to reveal what players need to do in the first 10 minutes to increase their chance of winning the game.

After exploring the data in a short exploratory data analysis, we build different models and compare their performance based on their prediction accuracy and the F1 score they can achieve on a test set not used for training. We will use a simple logistic regression as a based line and among others a naive bayes classifier, a k-nearest-neighbor classifier, a support vector machines, a random forest classifier, an ensemble method, and xgboosting.

The final outcome reveals that a **xgboost** classifier yields the best accuracy and F1 score.

At the end of this report, we will also discuss limitation and avenues to further improve the models we explored in this project.

Table of content

Important note: This script requires 25min to run on a MacBook Pro M3 Max with 36GB unified memory and potentially much longer on a less powerful machine

1. Introduction

League of Legends (LoL) is one of the most popular online games that has ever existed. At its peak in 2022, it attracted more than 180 million monthly active players [1]. Even today, millions of players compete in LoL. LoL also offers an Esports League where only the best of the best compete for staggering price pools. For instance, the prize pool for the biggest LoL tournament in 2022 alone was over 2.2 million US dollars [2].

Given the high stakes and the highly competitive nature of this game, the goal of our project is to provide data-based insights into how to win more games. In this project, we will use a data set (available here)

comprising almost 10k high end (ELO DIAMOND I to MASTER) LoL games. In addition to a variable that indicates which team (blue or red) has won, this data set contains features measuring key performance indicators after the first 10 minutes of the game. Based on these features, we try to predict the final winner. The predictive power of different variables will provide us with insights what strategies players should focus on in the first 10 minutes of a game to increase their chance of winning the game.

Technically, we will build different classification models and compare their performance. We will use the **accuracy** and the **F1 score** as our key performance metrics and validate our best performing model on a final holdout data set that we did not use for training.

The accuracy can be compute as follows:

$$Accuracy = \frac{True\ Positives\ (TP) + True\ Negatives\ (TN)}{Total\ Number\ of\ Instances}$$

The F1 one score can be compute as follows:

$$F1score = \frac{2 * Precision * Recall}{Precision + Recall}$$

Where Precision is:

$$Precision = \frac{True\ Positives\ (TP)}{True\ Positives\ (TP) = False\ Positives\ (FP)}$$

And Recall (also called sensitivity) is:

$$Recall = \frac{True\ Positives\ (TP)}{True\ Positives\ (TP) = False\ Negatives\ (FN)}$$

Later on, we will compute these metrics using the caret package.

Install libraries

To run this script, we need to load some libraries. Before we load the libraries, we check if they are installed and install them if necessary.

Importing the data set

The data set we are using for this project is available here but I also provide it as part of my Github repository. The original author of the data set is Yi Lan Ma (kaggle profile).

We can import the data set by using its relative path like this:

```
lol_data <- read.csv("0_Capstone_data_LeagueOfLegends_10min.csv")
```

Variable description

As we can see in the table below, the data set comprises 40 variables. The **gameID** variable identifies the game. **blueWins** is our outcome variable. It indicates who won the game. One means the blue team has won. Zero means the red team won. The remaining variables (38 in total) are the same 19 features for each of the two teams. They include number of kills, deaths, assists, amount of gold etc. We will explore a few of these variables later on in greater detail. Table 1 shows the names of all variables.

Table 1: Variable Names in the Dataset

Variable
gameId

blueWins
blueWardsPlaced
blueWardsDestroyed
blueFirstBlood

blueKills
blueDeaths
blueAssists
blueEliteMonsters
blueDragons

blueHeralds
blueTowersDestroyed
blueTotalGold
blueAvgLevel
blueTotalExperience

blueTotalMinionsKilled
blueTotalJungleMinionsKilled
blueGoldDiff
blueExperienceDiff
blueCSPerMin

blueGoldPerMin
redWardsPlaced
redWardsDestroyed
redFirstBlood
redKills

redDeaths
redAssists
redEliteMonsters
redDragons
redHeralds

redTowersDestroyed
redTotalGold
redAvgLevel
redTotalExperience
redTotalMinionsKilled

redTotalJungleMinionsKilled
redGoldDiff
redExperienceDiff
redCSPerMin
redGoldPerMin

Data cleaning

As the data has been preprocessed by its author, we do not need to do much cleaning.

Still, we are checking for missing values. Good news is there are 0 missing values. Thus, we can proceed.

Creating data sets (training set, testing set, and final holdout set)

Before we start with the data exploration and building our model, we create three data sets. First, we partition the data set so that 10% of the total data goes into our final holdout data set (called: **final_holdout_test**). We will not use this data set for any type of analysis (i.e., exploratory data analysis or creating different

models). We will only use it to evaluate the performance of our final model on data completely new to the model. Second, we split the remaining 90% of the into a training data set (called: **lol_train**), which we use to build our model and a testing data set, which we use to evaluate the models we build on an ongoing basis. We follow common practice and again use 90% of the remaining data for the training data set and 10% for the test set (**lol_test**).

After split the data sets, we can confirm that our holdout data set has roughly 10% (10.0010122%) of the data. Our train data 80.9899787% (i.e.,90% of 90% of our total data) set has and the test data set has 9.009009% (i.e.,10% of 90% of our total data) of the total data.

2. Analysis

To analyse our data, we proceed in two steps. First, we conduct an exploratory data analysis to get a feeling for our data and decide which features we want to select or create if necessary. Second, we build different models using the train data set and evaluate their performance on our test data set.

Exploratory data analysis

We will conduct our exploratory data analysis only on our training data set.

Descriptive Statistics

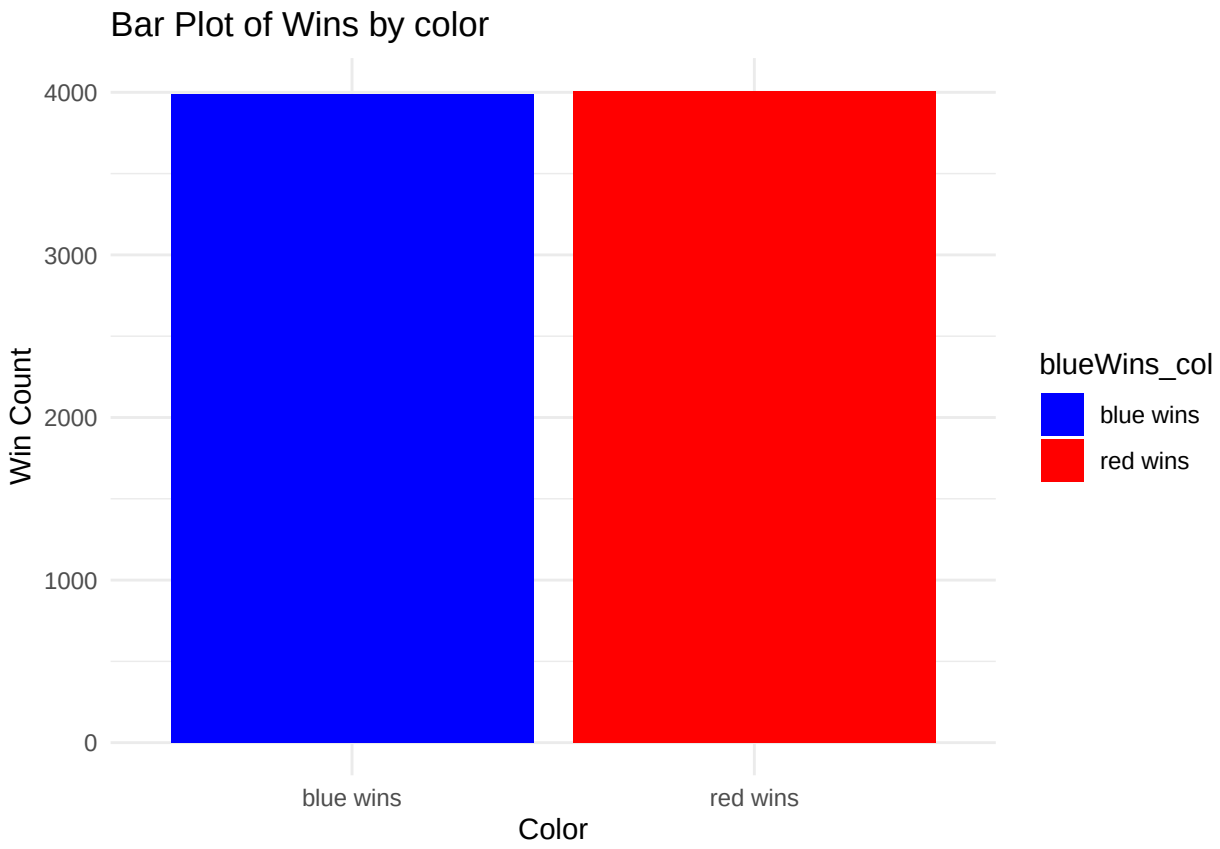
Table 2 show summary statistics of all variables in our data set.

Table 2: Summary Statistics

	vars	n	mean	sd	min	max
gameId	1	8001	4.500016e+09	2.769659e+07	4295358071.0	4527990640.0
blueWins	2	8001	4.988126e-01	5.000298e-01	0.0	1.0
blueWardsPlaced	3	8001	2.245007e+01	1.841242e+01	5.0	250.0
blueWardsDestroyed	4	8001	2.809649e+00	2.142460e+00	0.0	24.0
blueFirstBlood	5	8001	5.038120e-01	5.000167e-01	0.0	1.0
blueKills	6	8001	6.177353e+00	2.992811e+00	0.0	22.0
blueDeaths	7	8001	6.129484e+00	2.909636e+00	0.0	22.0
blueAssists	8	8001	6.632421e+00	4.039028e+00	0.0	29.0
blueEliteMonsters	9	8001	5.524309e-01	6.249255e-01	0.0	2.0
blueDragons	10	8001	3.648294e-01	4.814124e-01	0.0	1.0
blueHeralds	11	8001	1.876015e-01	3.904181e-01	0.0	1.0
blueTowersDestroyed	12	8001	5.186850e-02	2.448149e-01	0.0	3.0
blueTotalGold	13	8001	1.650267e+04	1.528533e+03	12178.0	23701.0
blueAvgLevel	14	8001	6.916960e+00	3.043576e-01	4.6	8.0
blueTotalExperience	15	8001	1.793062e+04	1.196784e+03	10098.0	22224.0
blueTotalMinionsKilled	16	8001	2.166549e+02	2.177116e+01	120.0	283.0
blueTotalJungleMinionsKilled	17	8001	5.052206e+01	9.886356e+00	0.0	88.0
blueGoldDiff	18	8001	1.292201e+01	2.432954e+03	-10329.0	8863.0
blueExperienceDiff	19	8001	-3.117898e+01	1.908861e+03	-8531.0	8348.0
blueCSPerMin	20	8001	2.166549e+01	2.177116e+00	12.0	28.3
blueGoldPerMin	21	8001	1.650267e+03	1.528533e+02	1217.8	2370.1
redWardsPlaced	22	8001	2.237483e+01	1.863514e+01	6.0	276.0
redWardsDestroyed	23	8001	2.717910e+00	2.136537e+00	0.0	24.0
redFirstBlood	24	8001	4.961880e-01	5.000167e-01	0.0	1.0
redKills	25	8001	6.129484e+00	2.909636e+00	0.0	22.0
redDeaths	26	8001	6.177353e+00	2.992811e+00	0.0	22.0
redAssists	27	8001	6.644294e+00	4.018763e+00	0.0	25.0
redEliteMonsters	28	8001	5.706787e-01	6.263267e-01	0.0	2.0
redDragons	29	8001	4.113236e-01	4.921044e-01	0.0	1.0
redHeralds	30	8001	1.593551e-01	3.660298e-01	0.0	1.0
redTowersDestroyed	31	8001	4.311960e-02	2.162532e-01	0.0	2.0
redTotalGold	32	8001	1.648975e+04	1.478677e+03	11212.0	22732.0
redAvgLevel	33	8001	6.924759e+00	3.026684e-01	4.8	8.0
redTotalExperience	34	8001	1.796180e+04	1.188616e+03	10610.0	22269.0
redTotalMinionsKilled	35	8001	2.173620e+02	2.181527e+01	107.0	282.0
redTotalJungleMinionsKilled	36	8001	5.133471e+01	1.000640e+01	4.0	92.0

	vars	n	mean	sd	min	max
redGoldDiff	37	8001	-1.292201e+01	2.432954e+03	-8863.0	10329.0
redExperienceDiff	38	8001	3.117898e+01	1.908861e+03	-8348.0	8531.0
redCSPerMin	39	8001	2.173620e+01	2.181528e+00	10.7	28.2
redGoldPerMin	40	8001	1.648975e+03	1.478677e+02	1121.2	2273.2

Based on our summary statistics and the plot below, we can already see that there is no imbalance regarding which side (blue or red) wins. This is good news. The game seems balanced.

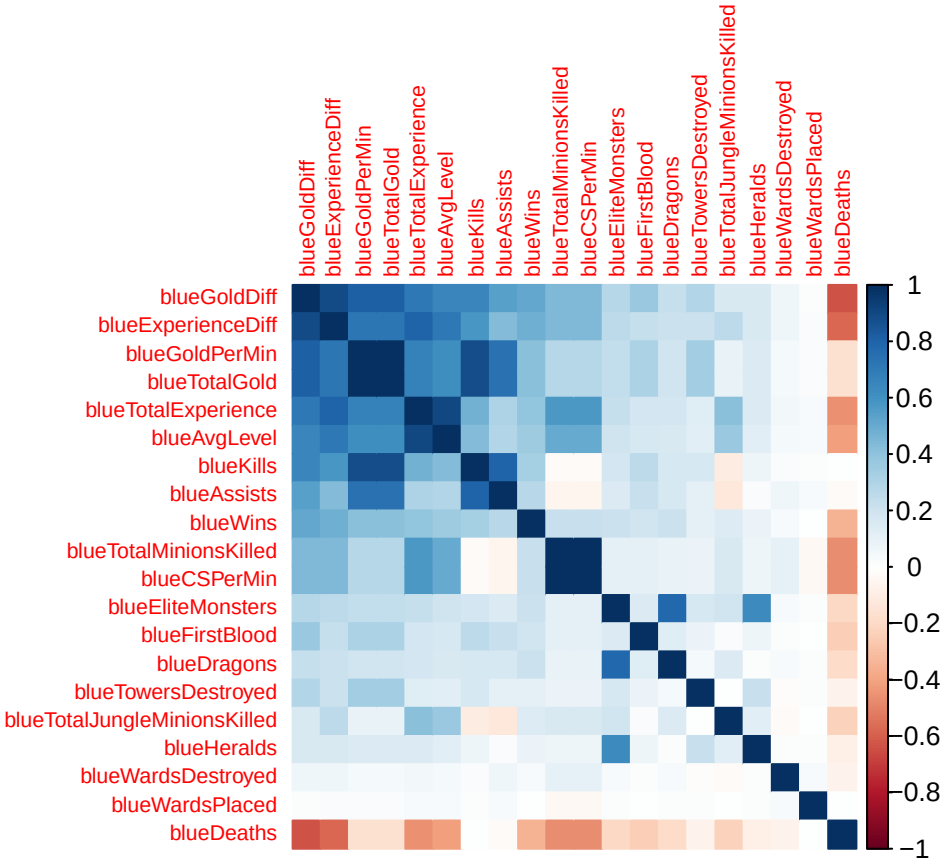


Correlations

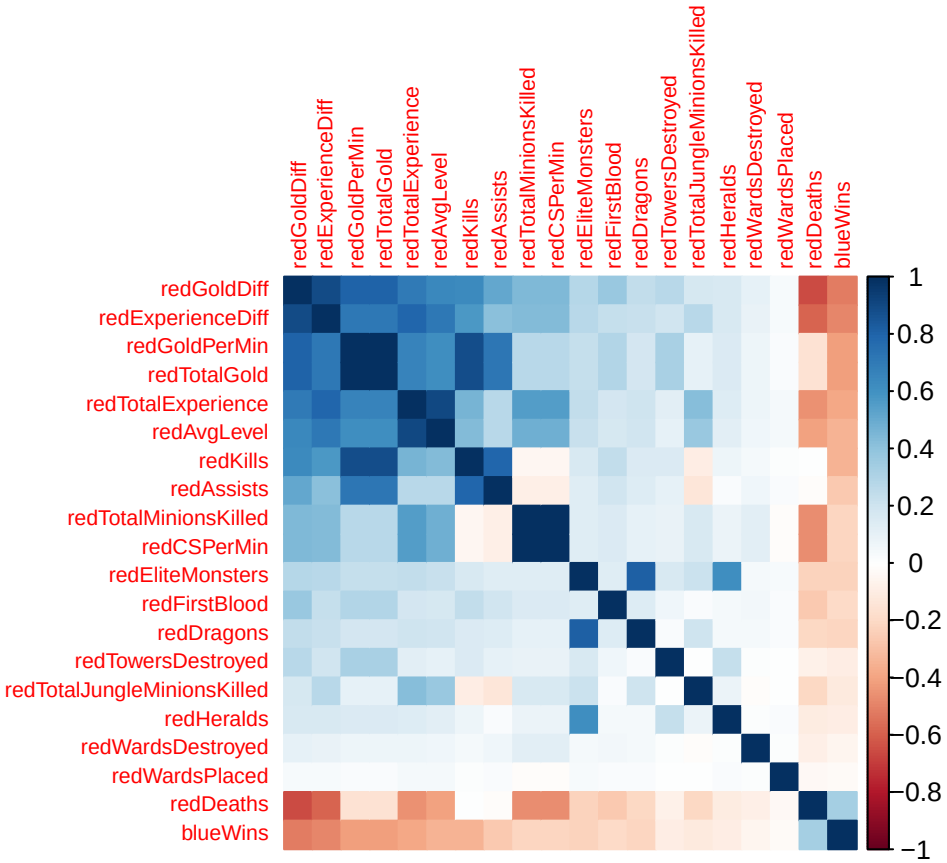
As it is quite complex to understand how 38 features influence our outcome variable, we analyze correlation patterns to check which variables we should focus on and which variables we can potentially ignore.

We plot two correlation plots. One comprises all variables for the blue team. The other all variables for the red team.

Correlation plot blue team variables:



Correlation plot red team variables:

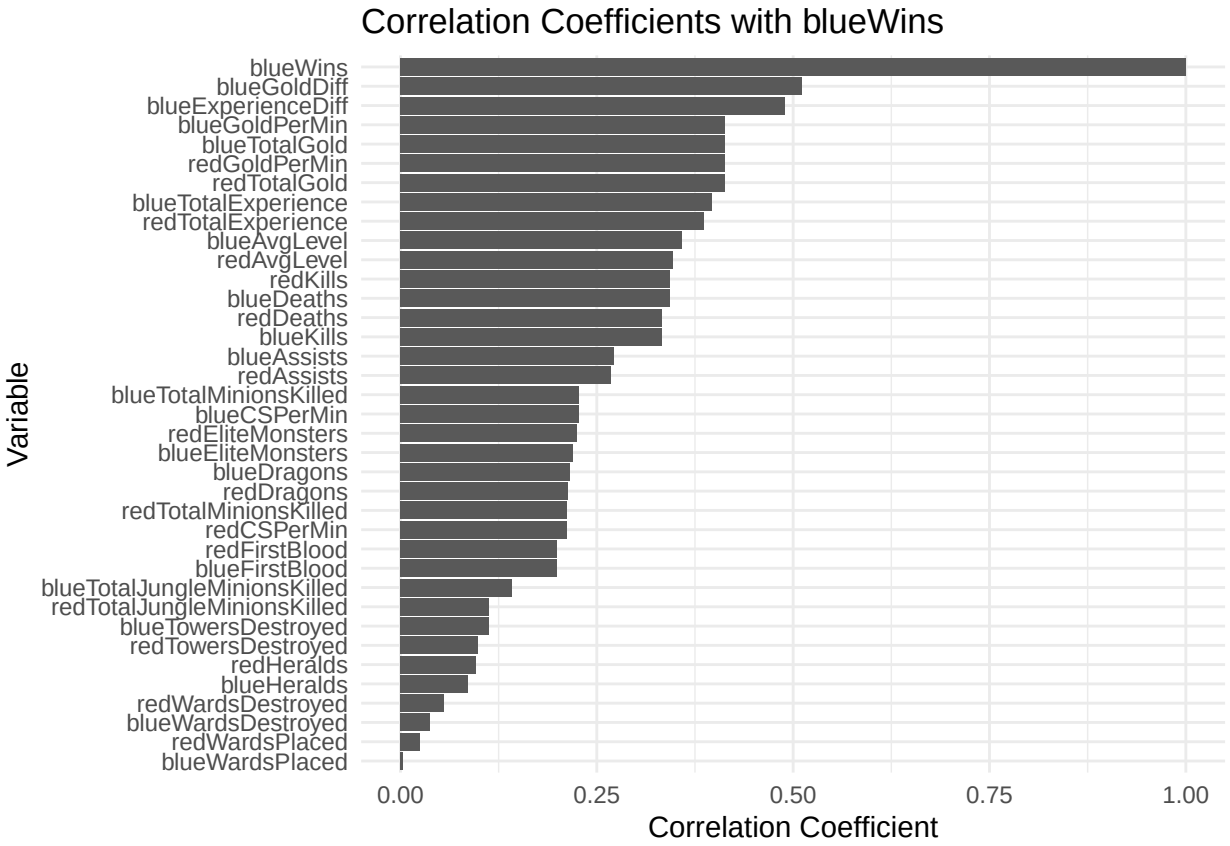


Based on the correlations, we can already observe some interesting patterns. For instance, we can see that the number of wards placed and destroyed hardly correlate with any other variable and most importantly not with what team won. Therefore, these variables might not be important to explain who won the final game and we could consider dropping them. Further, we can see that some variables are almost perfect correlates (e.g., redGoldDiff & redExperienceDiff, redTotalGold & redGoldPerMin, redTotalExperience &redAvgLevel, redCSperMin & redTotalMinionsKilled, redAssists & redKills). These correlations make perfect sense as for example each levels depend on experience, thus we would expect that total experience and the level are highly correlated. Or every kill es rewarded with gold and experience, thus gold and experience are highly correlated.

Finally, the variables redGoldDiff and blueGoldDiff are perfectly correlated variables as they measure the difference of gold between the two teams. To avoid multicolinearity issues, we will only keep blueGoldDiff. The same applies to redExperienceDiff.

Regarding our target variable (blueWins), we can also see some interesting patterns. For example, we see a negative correlation between the gold of the red team and the win of a blue team and a positive correlation between the gold of the blue team and the win of the blue team. For the deaths of the red team, we see a positive correlation with the win of the blue team and a negative correlation between the deaths of the blue team and the win of the blue team. Based on this we can already form the hypotheses that the amount of goal a team makes in the first 10 minutes and the number of deaths it suffers seem to explain to some extent if they win or lose.

To further investigate this pattern, we can sort the correlations of all variables with our target variable by their magnitude. And create a bar plot to better see their magnitude. Based on this plot, we can see that gold, experience, kills, and deaths have the highest correlations.



Building models

Now that we have a basic understanding of our data, we proceed with building different models.

Naive model

For a first, naive model, we just predict that blue always wins. As the wins are quite balanced, it does not really matter if we predict that blue or red wins all the time.

```
actual <- factor(lol_test$blueWins)
naive_pred <- factor(rep(1, length(lol_test$blueWins)))

# Confusion matrix
conf_matrix <- confusionMatrix(naive_pred, actual, mode = "everything", positive="1")

# Performance metrics
naive_accuracy <- conf_matrix$overall["Accuracy"]
naive_f1 <- conf_matrix$byClass["F1"]
```

By predicting blue always wins, we can achieve an accuracy of 0.4910112 and a F1 score of 0.6586285. This is also what we would expect given that wins are balanced.

Baseline model (Logistic regression)

As guessing only one side wins all the time is not a very good baseline, we check if we can do better with a simple logistic regression. In this regression, we include all variables we identified before.

```
# train logistic regression model
log_reg_model <- glm(blueWins ~ .,
```

```
                     data = lol_train,
                     family = binomial)

# Make predictions on test set
log_reg_preds <- predict(log_reg_model,
                         newdata = lol_test,
                         type = "response")

# Convert probabilities to binary predictions
binary_log_reg_preds <- ifelse(log_reg_preds > 0.5, 1, 0)

# Confusion matrix
conf_matrix <- confusionMatrix(as.factor(binary_log_reg_preds),
                               as.factor(lol_test$blueWins),
                               mode = "everything",
                               positive="1")

# Performance metrics
log_reg_accuracy <- conf_matrix$overall["Accuracy"]
log_reg_f1 <- conf_matrix$byClass["F1"]
```

We can see that already a simple logistic regression significantly improve our prediction performance. Our accuracy increases to 0.7067416 and the F1 score to 0.6989619. This will serve as our new baseline.

Naive Bayes

Next, we fit a naive bayes classifer. As naive bayes does not really have tunable hyperparameter, we only fit one model.

```
# Train the Naive Bayes model
naive_bayes_model <- naiveBayes(blueWins ~ .,
                          data = lol_train)

# Make predictions on test set
naive_bayes_preds <- predict(naive_bayes_model,
                             newdata = lol_test)

# Confusion matrix
conf_matrix <- confusionMatrix(as.factor(naive_bayes_preds),
                               as.factor(lol_test$blueWins),
                               mode = "everything",
                               positive="1")

# Performance metrics
naive_bayes_accuracy <- conf_matrix$overall["Accuracy"]
naive_bayes_f1 <- conf_matrix$byClass["F1"]
```

Our Naive Bayes model only slightly increase the accuracy to 0.7089888 and the F1 score to 0.7012687.

K-nearest neighbors

For k-nearest the most important hyperparameter is k, the number of neighbors. We use 10-fold crossvalidation to search the opimal number of neighbors.

```
# Define training control with 10-fold cross-validation
train_control <- trainControl(method = "cv", number = 10)
```

```r
# Train and tune the k-NN model
knn_model <- train(blueWins ~ ., data = lol_train, method = "knn",
                trControl = train_control,
                tuneLength = 30)  # Automatically test k from 1 to 10


# Make predictions on the test set
knn_preds <- predict(knn_model, newdata = lol_test)

# Convert probabilities to binary predictions
binary_knn_preds <- ifelse(knn_preds > 0.5, 1, 0)


# Confusion matrix
conf_matrix <- confusionMatrix(as.factor(binary_knn_preds),
                               as.factor(lol_test$blueWins),
                               mode = "everything",
                               positive="1")

# Performance metrics
knn_accuracy <- conf_matrix$overall["Accuracy"]
knn_f1 <- conf_matrix$byClass["F1"]

# best tunes
bestneighbors <- knn_model$bestTune[1]
```

We can see that our knn model slightly increase our accuracy to 0.7134831 and the F1 score to 0.7072331. We achieve these values with k=63.

Random Forest

The next model we try is a random forest model. For these models, we have a set of hyperparameters (e.g., the number of trees or mtry which is the number of features to consider when looking for the best split). Again, we use 10-fold cross validation to find the best model.

```r
# Define training control with 10-fold cross-validation
train_control <- trainControl(method = "cv", number = 10)

# Define the grid of hyperparameters to tune
tune_grid <- expand.grid(mtry = c(1, 2, 3, 5, 10))


# Train and tune the Random Forest model
rf_model <- train(as.factor(blueWins) ~ ., data = lol_train,
                  method = "rf",
                  ntree = 200,
                  trControl = train_control,
                  tuneGrid = tune_grid,
                  nSamp = 500)


fit_rf <- randomForest(lol_train[, -1], lol_train$blueWins,
                       mtry = rf_model$bestTune$mtry)
```

```r
# Make predictions on the test set
rf_preds <- predict(fit_rf, newdata = lol_test[, -1])


# Convert probabilities to binary predictions
binary_rf_preds <- ifelse(rf_preds > 0.5, 1, 0)


# Confusion matrix
conf_matrix <- confusionMatrix(as.factor(binary_rf_preds),
                               as.factor(lol_test$blueWins),
                               mode = "everything",
                               positive="1")

# Performance metrics
rf_accuracy <- conf_matrix$overall["Accuracy"]
rf_f1 <- conf_matrix$byClass["F1"]
```

We can see that random forest model actually performs worse than knn with an accuracy of 0.6966292 and an F1 score of 0.7072331.

Support Vector Machine (SVM)

Next, we test a svm classifier and again tune it with 10-fold crossvalidation.

```r
# Define training control with 10-fold cross-validation
train_control <- trainControl(method = "cv", number = 10)

# Define the grid of hyperparameters to tune
tune_grid <- expand.grid(
  C = 2^(-5:2),  # Regularization parameter
  sigma = 2^(-5:2)  # Kernel parameter for RBF kernel
)


# Train and tune the SVM model
svm_model <- train(as.factor(blueWins) ~ ., data = lol_train, method = "svmRadial",
                trControl = train_control, tuneGrid = tune_grid)


svm_preds <- predict(svm_model, newdata = lol_test)

# Confusion matrix
conf_matrix <- confusionMatrix(as.factor(svm_preds),
                               as.factor(lol_test$blueWins),
                               mode = "everything",
                               positive="1")

# Performance metrics
svm_accuracy <- conf_matrix$overall["Accuracy"]
svm_f1 <- conf_matrix$byClass["F1"]
```

Again, our results are slightly worse than the results from our best model. Accuracy: 0.7; F1 score: 0.6913295.

Ensemble

We know that each model comes up with different predictions. We can try to improve our model's performance even further by combining different models in an ensemble and use their joint predictive power. Our hope is that the majority vote of a combination of different models is better than the best performing model.

To create our ensemble, we take the three best performing models and predict that blue wins only if at least two models say so.

```
# compute a majority vote of all three models
ensemble_sum <- as.numeric(binary_log_reg_preds) +
                as.numeric(naive_bayes_preds) +
                as.numeric(binary_knn_preds)

binary_ensemble_preds <- ifelse(ensemble_sum > 1, 1, 0)

# Confusion matrix
conf_matrix <- confusionMatrix(as.factor(binary_ensemble_preds),
                               as.factor(lol_test$blueWins),
                               mode = "everything",
                               positive="1")

# Performance metrics
ensemble_accuracy <- conf_matrix$overall["Accuracy"]
ensemble_f1 <- conf_matrix$byClass["F1"]
```

Interestingly, we can see that our accuracy is slightly worse (accuracy ensemble: 0.7067416) than the our best accuracy so far but our F1 score has increased to 0.7172264

XGBoost

The final model we test is an XGBoost classifier. It relies on extreme gradient boosting and is quite successfully used in various machine learning challenges on kaggle.com.We use 10-fold crossvalidation to find the best model.

IMPORTANT: This model takes quite some time to train. If you run this code on a less powerful machine you might want consider skipping it.

```
lol_train$blueWins <- as.factor(lol_train$blueWins)
lol_test$blueWins <- as.factor(lol_test$blueWins)

# Define training control with 10-fold cross-validation
train_control <- trainControl(method = "cv",
                              number = 10,
                              verboseIter = FALSE)

# Define the grid of hyperparameters to tune
tune_grid <- expand.grid(
  nrounds = c(50, 100),
  max_depth = c(3, 6, 9),
  eta = c(0.01, 0.1, 0.3),
  gamma = c(0, 1),
  colsample_bytree = c(0.5, 0.7, 1),
  min_child_weight = c(1, 3, 5),
  subsample = c(0.5, 0.7, 1)
)
```

```r
# Train and tune the XGBoost model
xgboost_model <- train(x = as.matrix(lol_train[,-1]),
                       y = lol_train$blueWins,
                       method = "xgbTree",
                       trControl = train_control,
                       tuneGrid = tune_grid)

# Make predictions on the test set
xgboost_preds <- predict(xgboost_model, newdata = as.matrix(lol_test[,-1]))

# Confusion matrix
conf_matrix <- confusionMatrix(as.factor(xgboost_preds),
                               as.factor(lol_test$blueWins),
                               mode = "everything",
                               positive="1")

# Performance metrics
xgboost_accuracy <- conf_matrix$overall["Accuracy"]
xgboost_f1 <- conf_matrix$byClass["F1"]
```

We can see that our accuracy of 0.7 and F1 score of 0.6934558is slightly worse than those we obtained for simpler models.

3. Discussion

The following table compares the performance of all models. As we decide to prioritize the F1 score over the simple accuracy, we choose the ensemble as our final model. Further, we also keep the XGBoost model as it offers a simple way to assess the importance of individual features which come in handy if we want to explain how a model makes predictions and if we want to derive recommendations for a potential strategy users should follow in the first 10 minutes of a game.

```
## # A tibble: 8 x 3
##   Model               Accuracy    F1
##   <chr>                  <dbl> <dbl>
## 1 Guessing               0.491 0.659
## 2 Logistic regression    0.707 0.699
## 3 Naive bayes            0.709 0.701
## 4 Knn                    0.713 0.707
## 5 Random forest          0.697 0.697
## 6 SVM                    0.7   0.691
## 7 Ensemble               0.707 0.717
## 8 XGBoost                0.7   0.693
```

Predictions on hold out data set

To test the performance of our best performing model, we make predictions on the final holdout data set we have not used during the training process. As explained above, we will use our ensemble model and the xgboost model.

We can use the following code to use our pre-trained models to make predictions on the holdout test set. This is what the code looks like for our ensemble:

```r
# make predictions on the final holdout test set
log_reg_final_preds <-predict(log_reg_model, newdata = final_holdout_test_selected[,-1])
naive_bayes_final_preds <-predict(naive_bayes_model, newdata = final_holdout_test_selected[,-1])
```

```
knn_final_preds <-predict(knn_model, newdata = final_holdout_test_selected[,-1])

# convert prediction to binary labels
binary_log_reg_final_preds <- ifelse(log_reg_final_preds > 0.5, 1, 0)
binary_knn_final_preds <- ifelse(knn_final_preds > 0.5, 1, 0)

ensemble_sum <-as.numeric(binary_log_reg_final_preds) +
               as.numeric(binary_knn_final_preds) +
              as.numeric(naive_bayes_final_preds)

# create majority vote
binary_final_ensemble_preds <- ifelse(ensemble_sum > 1, 1, 0)


# Confusion matrix
conf_matrix <- confusionMatrix(as.factor(binary_final_ensemble_preds),
                               as.factor(final_holdout_test_selected$blueWins),
                               mode = "everything",
                               positive="1")

# Calculate accuracy
ensemble_final_accuracy <- conf_matrix$overall["Accuracy"]


# Calculate F1
ensemble_final_f1 <- conf_matrix$byClass["F1"]
```

And this is what the code looks like for our xgboost model:

```
# make predictions on the final holdout testset)
xgboost_preds <- predict(xgboost_model, newdata = as.matrix(final_holdout_test_selected[,-1]))


# Confusion matrix
conf_matrix <- confusionMatrix(as.factor(xgboost_preds),
                               as.factor(final_holdout_test_selected$blueWins),
                               mode = "everything",
                               positive="1")

# Calculate accuracy
xgboost_final_accuracy <- conf_matrix$overall["Accuracy"]

# Calculate F1
xgboost_final_f1 <- conf_matrix$byClass["F1"]
```

The following table shows that both models performance almost equally on the holdout test set.

```
## # A tibble: 2 x 3
##   Model    Accuracy    F1
##   <chr>       <dbl> <dbl>
## 1 Ensemble    0.734 0.747
## 2 XGBoost     0.741 0.741
```

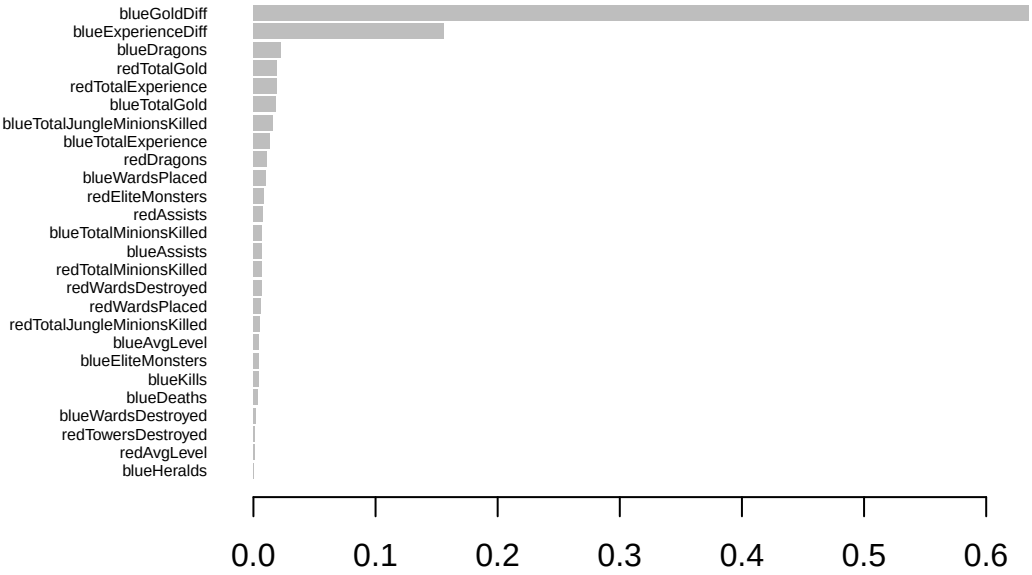
Overall, a performance of above 74% is not bad at all considering most games in our skill group take up to 30 minutes [4]. However, we must also consider that all strategies we drive from our models by no means

guarantee a win.

Exploring feature importance

One advantage of our XGBoost model over our ensemble method is that it is fairly easy to investigate the feature importance of our XGBoost model and derive insights regarding the "winning" strategy. We can simply call the best xgboost model we received after fine tuning, extract the features' importance, and plot them.

The following plot shows that the most important feature for our prediction is the Gold difference between two teams. Four of the other top 5 features are also related to experience and gold which are in turn associated with killing mobs. Only on position three and seven, we see killing the blue dragon and kills of jungle minions.



Based on the insights we got from the graph above, we can derive strategic advice that would sound like this: To increase your changes of winning the game, in the first 10 minutes, you should focus on collecting as much gold and experience as possible and prevent your opponent from doing the same. Not the absolute value seems important but the difference between the opponents. If you find time, you should go for the dragon and jungle minions. Destroying towers and getting first blood and heralds are less important.

Of course, this strategy does not guarantee that you will win in the end and needs further insights from experienced players but it is a start. Further, your opponent might adjust according to your strategy. All these factors need to be considered to secure the win but at least it is a start.

4. Conclusion

In this script, we used a data from the first 10 minutes of 10,000 competitive LoL matches to predict the final winner. With only a few variables, we were already able to achieve an accuracy of almost 75%. This result is remarkable considering that games usually go on for 30 minutes. Our results can provide ambitious LoL players with interesting insights on what they should do in the first 10 minutes of a match in order to secure a win.

Our approach also has some limitations that lend themselves to further research and potential model improvements. First, while 10,000 matches sounds like a lot, way more data is available and can increase model performance. Second, the set of available features is rather limited. Additional variables about the team composition or the chosen heroes might further improve our models performance. Third, with more data on the teams it would be possible to create more sophisticated strategies. For example, one could use

unsupervised learning to classify different play styles and assess which play style performs better against other play styles or even individual teams.

Overall, this project already shows the power of data-based insights and that data science is a suitable tool to derive interesting strategies for competitive online gaming.

References

1. https://prioridata.com/data/league-of-legends/#League_of_Legends_Player_Count
2. https://economictimes.indiatimes.com/news/international/us/prize-pool-for-league-of-legends-worlds-check-how-much-money-will-be-distributed-this-year/articleshow/94541119.cms?from=mdr
3. https://www.kaggle.com/datasets/bobbyscience/league-of-legends-diamond-ranked-games-10-min/data
4. https://www.leagueofgraphs.com/stats/game-durations