CYO Project Submission for HarvardX PH125.9x - League of Legends Wins Prediction

Jordan Georgiev

2024-12-08

1 Introduction

The CYO (Choose Your Own) project is part of the HarvardX PH125.9x Capstone course. For this project, we need to apply the knowledge gained in all previous course and use machine learning techniques that go beyond standard linear regression to analyze and predict variables from a publicly available dataset.

When researching publicly available datasets, I was looking for something not as big as the Movielens 10M dataset we used in the previous Movielens project in order to be able to test more demanding algorithms. Being a gamer myself I stumbled upon the "League of Legends Diamond Ranked Games (10 min)" available in the Key2Stats Learning platform. Since League Of Legends is a very popular game I also used to play, I was intrigued and wanted to give it a try. The dataset had around 10,000 rows and 40 variables, so it seemed as a good fit for a challenging project.

1.1 League of Legends

League of Legends is a very popular MOBA (multiplayer online battle arena) game where 2 teams (blue and red) face each other in combat. There are 3 lanes, a jungle, and 5 champions (roles). The goal is to destroy the enemy Nexus located in the opposing team's base to win the game. Very popular map used also in rated tournaments is Summoner's Rift. A simplified representation of Summoner's Rift. It is a square map separated by a diagonal line called "the river".

Game Mechanics

The Nexus "is guarded by the enemy champions and defensive structures known as"turrets". Each team's Nexus is located in their base, where players start the game and reappear after death. Non-player characters (NPC) known as minions are generated from each team's Nexus and advance towards the enemy base along three lanes guarded by turrets: top, middle, and bottom.

Each team's base contains three "inhibitors", one behind the third tower from the center of each lane. Destroying one of the enemy team's inhibitors causes stronger allied minions to spawn in that lane, and allows the attacking team to damage the enemy Nexus and the two turrets guarding it.

The regions in between the lanes are collectively known as the "jungle", which is inhabited by "monsters" that, like minions, respawn at regular intervals. Like minions, monsters provide gold and XP when killed. Another, more powerful class of monster resides within the river that separates each team's jungle. These monsters require multiple players to defeat and grant special abilities to their slayers' team. For example, teams can gain a powerful allied unit after killing the Rift Herald, permanent strength boosts by killing dragons, and stronger, more durable minions by killing Baron Nashor."[1]

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

Variables

This dataset contains the first 10min. stats of approx. 10k ranked games (solo queue) from a high ELO (Diamond I to Master). Players have roughly the same level. Each game is unique.[^2]

There are 19 features per team (38 in total) collected after 10min in-game. This includes kills, deaths, gold, experience, level... The features are using the same names for both teams (19 features per team) and are prefixed with "red" and "blue" accordingly.

The column ***blueWins*** is the target value we will be trying to predict. A value of 1 means the blue team has won, 0 means a lost (red team won).

Glossary

Some game specific terminology needs explanation in order to understand the game mechanics and also explain the names of several features in the dataset.

Warding totem: An item that a player can put on the map to reveal the nearby area. Very useful for map/objectives control.

Minions: NPC 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. The 4th dragon killed by a team gives a massive stats bonus. The 5th dragon (Elder Dragon) offers a huge advantage to the team.

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.

2 Project goal and evaluation

Our goal is to develop and train a machine learning model capable of ***predicting the wins for the blue team*** which are stored in the column ***blueWins*** of the dataset. The model will be trained on the dataset which contains the first 10 min of around 10,000 games played by elite players. We will evaluate 39 features and although the typical League of Legends game can last somewhere between 20 minutes to several hours, the first 10 minutes should give us enough insight on the winning strategy of players in the early phase of the game. The performance of the models chosen to be trained will be evaluated on their accuracy, sensitivity, specificity and precision. The best scoring model will be fine-tuned further in order to increase its accuracy.

2.1 Approach

The following steps will be followed and documented during the CYO project to ensure our models works.

1. **Setup and data loading**: load and prepare the dataset provided, clean and preprocess the data to handle missing values.
2. **Exploratory data analysis**: Analyse and visualize the downloaded dataset to understand the features and predictors.
3. **Data preparation**: Split data into training and test sets, adjust data if necessary.
4. **Model creation and evaluation**: create and validate the model.
5. **Model fine-tuning**: Fine-tune the model to increase performance and accuracy.
6. **Reporting**: document the analysis and the model and create the final report.

3 Setup and data loading

First we download the necessary packages (if not installed allready), load the libraries. We download the dataset and store it in a dataframe called ***game_data***. Let us inspect the data.

	1	2	3
X	1.0	2.0	3.0
gameId	4519157822.0	4523371949.0	4521474530.0
blueWins	0.0	0.0	0.0
blueWardsPlaced	28.0	12.0	15.0
blueWardsDestroyed	2.0	1.0	0.0
blueFirstBlood	1.0	0.0	0.0
blueKills	9.0	5.0	7.0
blueDeaths	6.0	5.0	11.0
blueAssists	11.0	5.0	4.0
blueEliteMonsters	0.0	0.0	1.0
blueDragons	0.0	0.0	1.0
blueHeralds	0.0	0.0	0.0
blueTowersDestroyed	0.0	0.0	0.0
blueTotalGold	17210.0	14712.0	16113.0
blueAvgLevel	6.6	6.6	6.4
blueTotalExperience	17039.0	16265.0	16221.0
blueTotalMinionsKilled	195.0	174.0	186.0
blueTotalJungleMinionsKilled	36.0	43.0	46.0
blueGoldDiff	643.0	-2908.0	-1172.0
blueExperienceDiff	-8.0	-1173.0	-1033.0
blueCSPerMin	19.5	17.4	18.6
blueGoldPerMin	1721.0	1471.2	1611.3
redWardsPlaced	15.0	12.0	15.0
redWardsDestroyed	6.0	1.0	3.0
redFirstBlood	0.0	1.0	1.0
redKills	6.0	5.0	11.0
redDeaths	9.0	5.0	7.0
redAssists	8.0	2.0	14.0
redEliteMonsters	0.0	2.0	0.0
redDragons	0.0	1.0	0.0
redHeralds	0.0	1.0	0.0
redTowersDestroyed	0.0	1.0	0.0
redTotalGold	16567.0	17620.0	17285.0
redAvgLevel	6.8	6.8	6.8
redTotalExperience	17047.0	17438.0	17254.0
redTotalMinionsKilled	197.0	240.0	203.0
redTotalJungleMinionsKilled	55.0	52.0	28.0
redGoldDiff	-643.0	2908.0	1172.0
redExperienceDiff	8.0	1173.0	1033.0
redCSPerMin	19.7	24.0	20.3
redGoldPerMin	1656.7	1762.0	1728.5

Column X contains just the row numbers of the dataset. The column gameId will not be needed, as it is unique. We can remove them from our dataset.

```
# Remove columns "X" and "gameId" from the data frame
game_data <- game_data[ , !(names(game_data) %in% c("X", "gameId"))]
```

Table 1: Dataset summary

	mean	sd	min	max	range	se
blueWins	0.4990384	0.5000244	0.0	1.0	1.0	0.0050308
blueWardsPlaced	22.2882883	18.0191765	5.0	250.0	245.0	0.1812919
blueWardsDestroyed	2.8248811	2.1749984	0.0	27.0	27.0	0.0218828
blueFirstBlood	0.5048082	0.5000022	0.0	1.0	1.0	0.0050305
blueKills	6.1839255	3.0110280	0.0	22.0	22.0	0.0302941
blueDeaths	6.1376658	2.9338177	0.0	22.0	22.0	0.0295173
blueAssists	6.6451058	4.0645199	0.0	29.0	29.0	0.0408934
blueEliteMonsters	0.5499544	0.6255265	0.0	2.0	2.0	0.0062935
blueDragons	0.3619800	0.4805974	0.0	1.0	1.0	0.0048353
blueHeralds	0.1879745	0.3907116	0.0	1.0	1.0	0.0039310
blueTowersDestroyed	0.0514222	0.2443692	0.0	4.0	4.0	0.0024586
blueTotalGold	16503.4555117	1535.4466363	10730.0	23701.0	12971.0	15.4482125
blueAvgLevel	6.9160036	0.3051458	4.6	8.0	3.4	0.0030701
blueTotalExperience	17928.1101326	1200.5237644	10098.0	22224.0	12126.0	12.0785352
blueTotalMinionsKilled	216.6995647	21.8584374	90.0	283.0	193.0	0.2199189
blueTotalJungleMinionsKilled	50.5096670	9.8982822	0.0	92.0	92.0	0.0995872
blueGoldDiff	14.4141107	2453.3491794	-10830.0	11467.0	22297.0	24.6832801
blueExperienceDiff	-33.6203057	1920.3704382	-9333.0	8348.0	17681.0	19.3209519
blueCSPerMin	21.6699565	2.1858437	9.0	28.3	19.3	0.0219919
blueGoldPerMin	1650.3455512	153.5446636	1073.0	2370.1	1297.1	1.5448212
redWardsPlaced	22.3679522	18.4574268	6.0	276.0	270.0	0.1857012
redWardsDestroyed	2.7231501	2.1383561	0.0	24.0	24.0	0.0215141
redFirstBlood	0.4951918	0.5000022	0.0	1.0	1.0	0.0050305
redKills	6.1376658	2.9338177	0.0	22.0	22.0	0.0295173
redDeaths	6.1839255	3.0110280	0.0	22.0	22.0	0.0302941
redAssists	6.6621115	4.0606124	0.0	28.0	28.0	0.0408540
redEliteMonsters	0.5731349	0.6264816	0.0	2.0	2.0	0.0063031
redDragons	0.4130985	0.4924151	0.0	1.0	1.0	0.0049542
redHeralds	0.1600364	0.3666584	0.0	1.0	1.0	0.0036890
redTowersDestroyed	0.0430205	0.2168999	0.0	2.0	2.0	0.0021822
redTotalGold	16489.0414010	1490.8884057	11212.0	22732.0	11520.0	14.9999097
redAvgLevel	6.9253163	0.3053114	4.8	8.2	3.4	0.0030718
redTotalExperience	17961.7304383	1198.5839120	10465.0	22269.0	11804.0	12.0590182
redTotalMinionsKilled	217.3492256	21.9116680	107.0	289.0	182.0	0.2204545
redTotalJungleMinionsKilled	51.3130884	10.0278849	4.0	92.0	88.0	0.1008911
redGoldDiff	-14.4141107	2453.3491794	-11467.0	10830.0	22297.0	24.6832801
redExperienceDiff	33.6203057	1920.3704382	-8348.0	9333.0	17681.0	19.3209519
redCSPerMin	21.7349226	2.1911668	10.7	28.9	18.2	0.0220454
redGoldPerMin	1648.9041401	149.0888406	1121.2	2273.2	1152.0	1.4999910

Let us check for missing values to tidy data if necessary.

```
# Check for missing values to tidy data if necessary
if (anyNA(game_data)) {
  colSums(is.na(game_data))
} else {
  print("No NA values found in game_data.")
```

```
}
```

```
## [1] "No NA values found in game_data."
```

When analyzing the dataset we found some columns which are interconnected - all the columns ending in "Diff".

Table 2: Paired columns

blueGoldDiff	blueExperienceDiff	redGoldDiff	redExperienceDiff
643	-8	-643	8
-2908	-1173	2908	1173
-1172	-1033	1172	1033
-1321	-7	1321	7
-1004	230	1004	-230

We also check for mirrored columns - same values in blue and red columns. The kills of blue count for deaths for red and vice versa.

Table 3: Mirrored columns

blueKills	redDeaths	redKills	blueDeaths
9	9	6	6
5	5	5	5
7	7	11	11
4	4	5	5
6	6	6	6

Other columns are mutually exclusive: 1 in one column leads to 0 in other. The reason is, e.g. there can be only one dragon in the game, so the first team to summon one gets 1. Both can be 0 though if no team summoned a dragon.

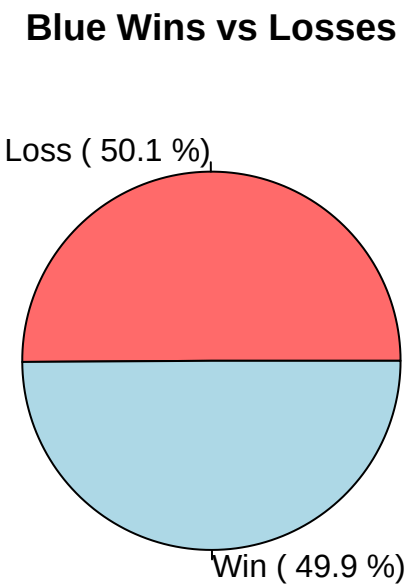
Table 4: Mutually exclusive columns

blueFirstBlood	redFirstBlood	blueDragons	redDragons	blueHeralds	redHeralds
1	0	0	0	0	0
0	1	0	1	0	1
0	1	1	0	0	0
0	1	0	0	1	0
0	1	0	1	0	0
0	1	1	0	0	0

The data seems tidy now, so we can continue to the exploratory data analysis (EDA).

4 Exploratory data analysis

The win to loss ratio of the dataset is very well balanced.



Now we are looking into correlations between variables/features. We remove self correlations - a variable correlating with itself (which usually results to cor = 1). We now examine how all "blue*" variables correlate to "red" variables and filter the ones with correlation coefficient > 0.4.

Table 5: Blue vs Red variables, corr > 0.5

Var1	Var2	Corr
blueKills	redDeaths	1.0000000
blueGoldPerMin	redDeaths	0.8887509
blueTotalGold	redDeaths	0.8887509
blueDeaths	redGoldPerMin	0.8857283
blueDeaths	redTotalGold	0.8857283
blueAssists	redDeaths	0.8136672
blueDeaths	redAssists	0.8040235
blueGoldDiff	redDeaths	0.6541476
blueDeaths	redGoldDiff	0.6399995
blueExperienceDiff	redDeaths	0.5837299
blueDeaths	redExperienceDiff	0.5776129
blueDragons	redEliteMonsters	-0.5065463
blueWins	redGoldDiff	-0.5111191
blueGoldDiff	redAssists	-0.5280806
blueAssists	redGoldDiff	-0.5497611
blueExperienceDiff	redKills	-0.5776129
blueKills	redExperienceDiff	-0.5837299
blueDragons	redDragons	-0.6319305

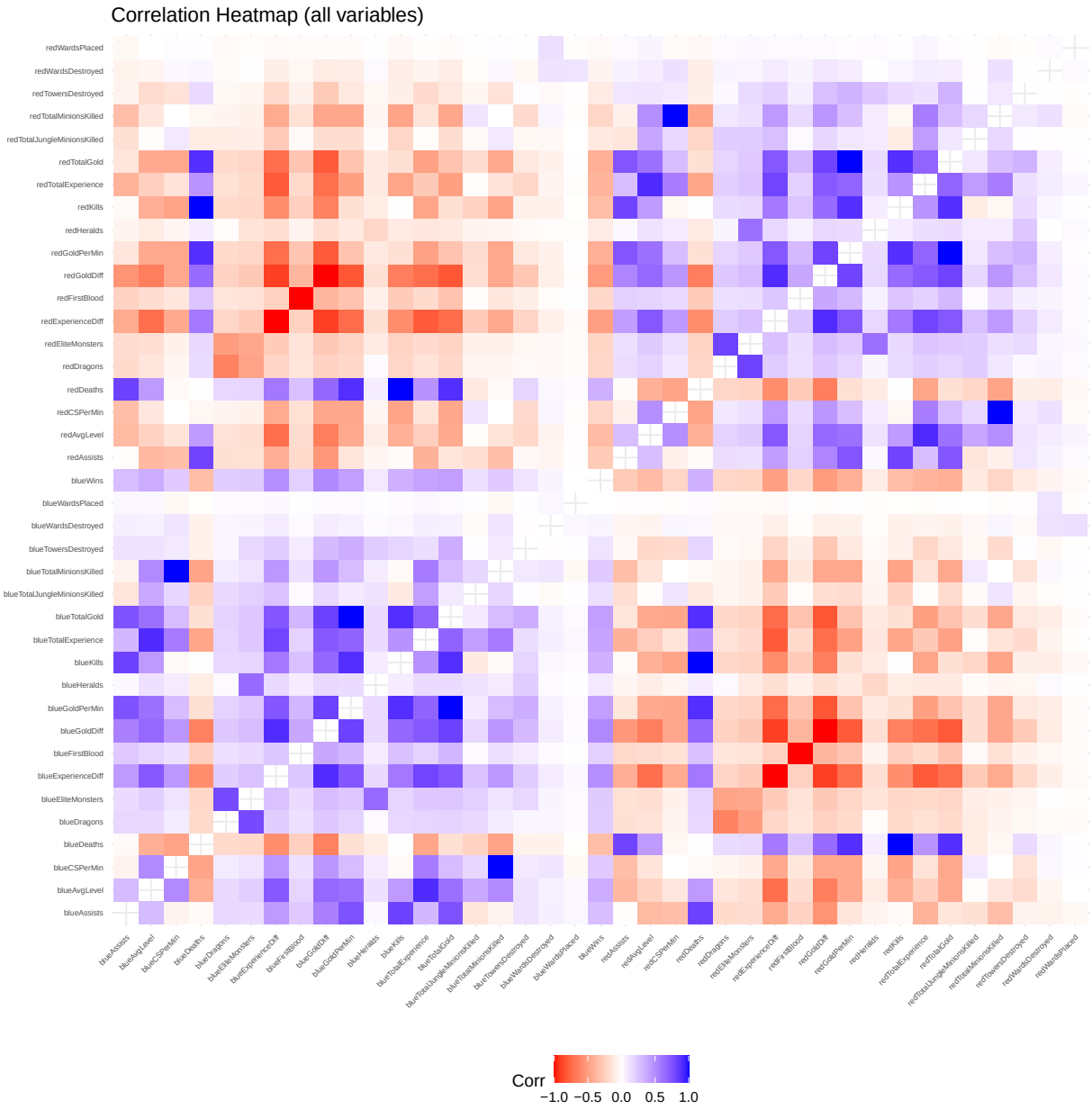
blueGoldDiff	redKills	-0.6399995
blueGoldDiff	redAvgLevel	-0.6529287
blueAvgLevel	redGoldDiff	-0.6535384
blueKills	redGoldDiff	-0.6541476
blueGoldDiff	redTotalExperience	-0.7144046
blueTotalExperience	redGoldDiff	-0.7179684
blueAvgLevel	redExperienceDiff	-0.7188222
blueExperienceDiff	redTotalGold	-0.7211897
blueExperienceDiff	redGoldPerMin	-0.7211897
blueExperienceDiff	redAvgLevel	-0.7219250
blueTotalGold	redExperienceDiff	-0.7293450
blueGoldPerMin	redExperienceDiff	-0.7293450
blueExperienceDiff	redTotalExperience	-0.8000887
blueTotalExperience	redExperienceDiff	-0.8008146
blueGoldDiff	redTotalGold	-0.8043473
blueGoldDiff	redGoldPerMin	-0.8043473
blueTotalGold	redGoldDiff	-0.8168028
blueGoldPerMin	redGoldDiff	-0.8168028
blueGoldDiff	redExperienceDiff	-0.8947295

It is also interesting to investigate which "blue" variables correlate positively to "redDeaths".

Table 6: Blue variables vs redDeaths

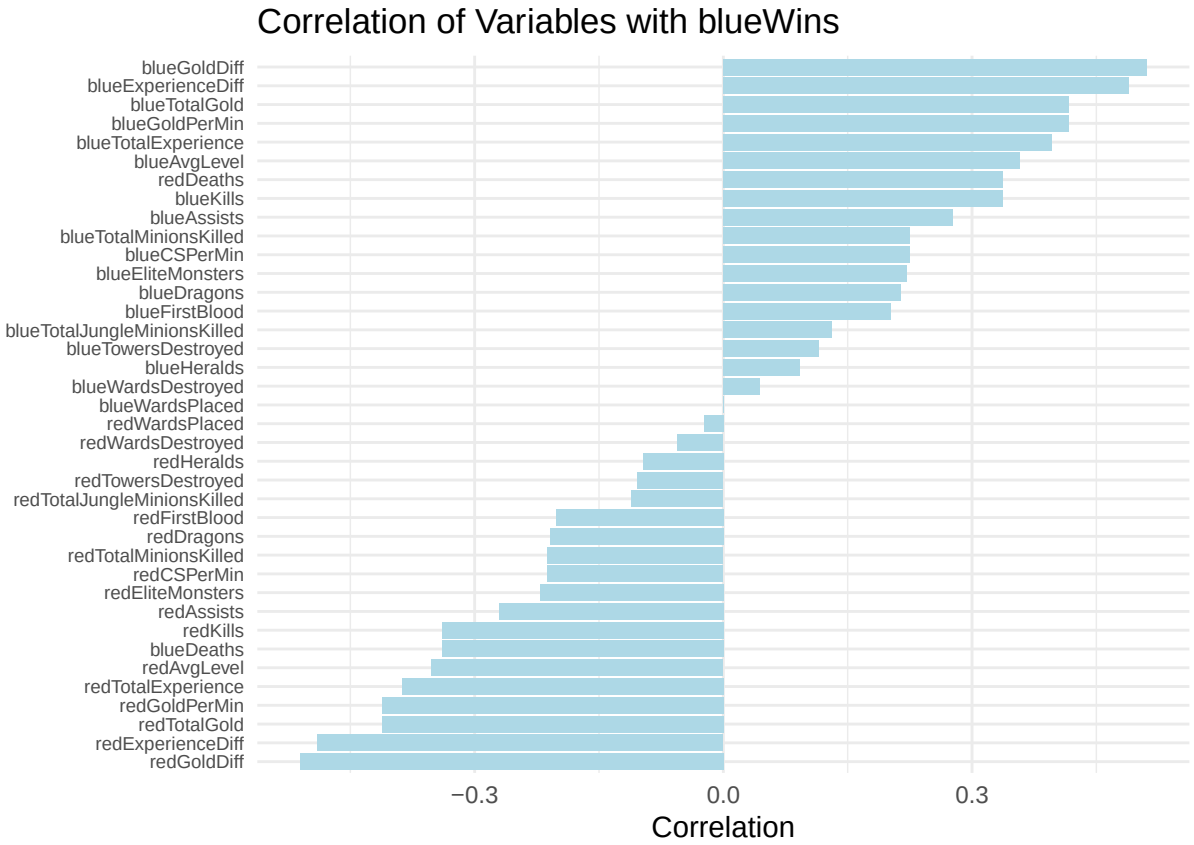
Var1	Var2	Corr
blueKills	redDeaths	1.0000000
blueGoldPerMin	redDeaths	0.8887509
blueTotalGold	redDeaths	0.8887509
blueAssists	redDeaths	0.8136672
blueGoldDiff	redDeaths	0.6541476
blueExperienceDiff	redDeaths	0.5837299
blueTotalExperience	redDeaths	0.4721548
blueAvgLevel	redDeaths	0.4348668
blueWins	redDeaths	0.3373576
blueFirstBlood	redDeaths	0.2694251

We can see the high correlations of variables to each other (except for "blueWins") in the following heatmap.

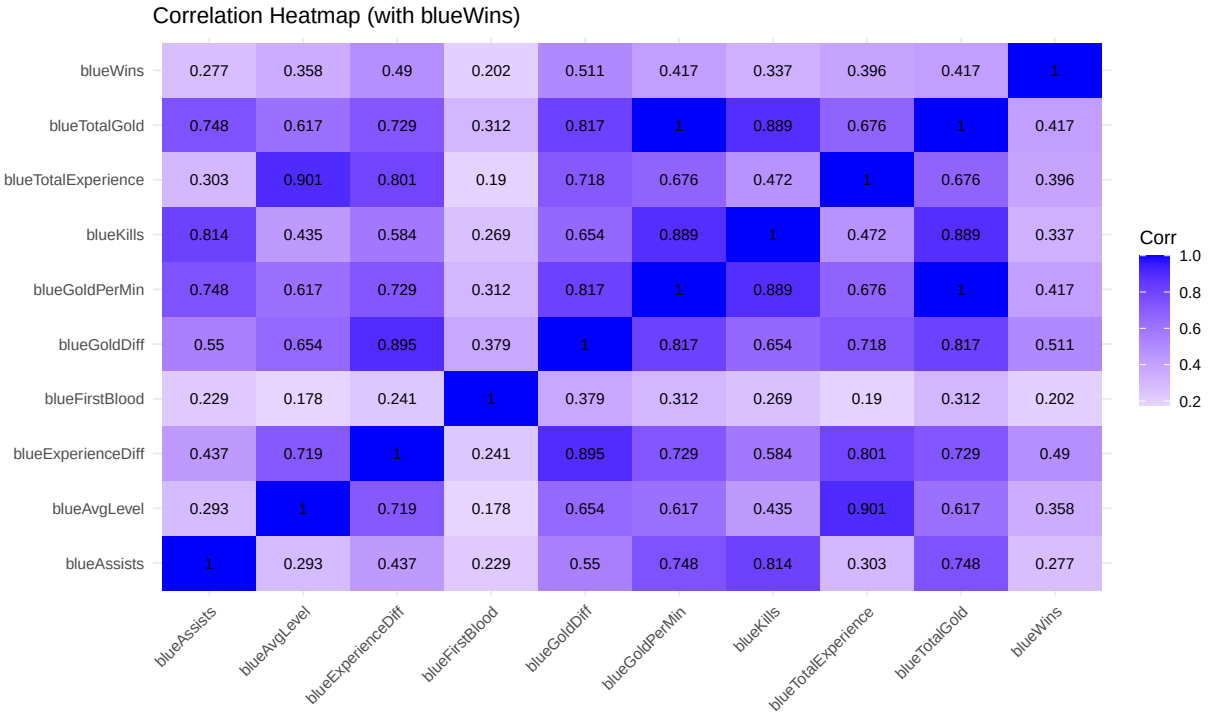


Since kills earn gold and experience, it is obvious, that deaths on the red side have a very high correlation to "gold" and "experience" variables on the blue side. Also the "blueFirstBlood" - the first blue kill in the game is correlated to "redDeaths".

Now let us correlate all columns (except "blueWins") to "blueWins".



Now we examine how the "blue variables", which correlate high to "redDeaths", correlate with "blueWins".



5 Data preparation

Here we prepare our dataset for modelling by splitting it into a test_set (10%) and a train_set (90%). We compare the number of rows and columns in each set. We also check the means of "blueWins" in both newly created sets to be sure that the data is well distributed.

Finally, we convert the column "blueWins" in both sets from integer to factor and we are ready to train our models.

```
# Partition the data to create test (10%) and training sets (90%)
set.seed(9, sample.kind="Rounding")
```

```
## Warning in set.seed(9, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
# Col blueWin is the one we will predict - partition the data (test_set 10% of game_data)
test_index <- createDataPartition(y = game_data$blueWins, times = 1, p = 0.1, list = FALSE)
train_set <- game_data[-test_index,]
test_set <- game_data[test_index,]
rm(test_index)

# Create an overview table with row and column count
dataset_overview <- data.frame(
  Dataset = c("train_set", "test_set"),
  Rows = c(nrow(train_set), nrow(test_set)),
  Columns = c(ncol(train_set), ncol(test_set))
)
tab1 <- kable(dataset_overview, booktabs = TRUE, escape = FALSE) %>%
  kable_styling(position = "center", latex_options = c("striped"))
tab1
```

Dataset	Rows	Columns
train_set	8891	39
test_set	988	39

```
# Mean of blueWins
mean(train_set$blueWins)
```

```
## [1] 0.4989315
```

```
mean(test_set$blueWins)
```

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

```
# Convert blueWins to a factor
train_set$blueWins <- as.factor(train_set$blueWins)
test_set$blueWins <- as.factor(test_set$blueWins)
```

6 Model development

We will use four different methods to train our model - Logisic regression, Decision Trees, Random Forest and kNN. We utilize the R caret package and its train() function to train each model using k-fold cross-validation (trControl = trainControl(method = "cv", number = 5)). This helps to estimate the model's performance more robustly by evaluating it on different subsets of the training data.

A short overview of each model, its pros and cons (see Irizzary, 2019.[2]):

Logistic Regression

Logistic regression is a statistical method used to model the probability of an event occurring (in this case, the "blueWins" outcome) based on one or more predictor variables. It uses a sigmoid function to map the linear combination of predictors to a probability between 0 and 1. Mathematical Representation:

$$P(Y = 1 \mid X) = \frac{1}{1 + exp(-(\beta_0 + \beta_1 X_1 + ... + \beta_p X_p))}$$

$$P(Y = 1 \mid X)$$

is the probability of the event occurring (blue team winning) given the predictor variables.

$$X_1, ..., X_p$$

are the predictor variables (e.g., blueKills, blueAssists, blueGoldDiff).

$$\beta_0, \beta_1, ..., \beta_p$$

are the model coefficients.

Pros Simple and interpretable. Provides probabilities for the outcome. Relatively fast to train.

Cons Assumes a linear relationship between the log-odds of the outcome and the predictors. Can be sensitive to outliers and may not perform well with highly non-linear relationships.

Decision Tree

Decision trees create a tree-like model where each node represents a decision based on the value of a particular feature. They recursively partition the data into smaller subsets based on the chosen features, aiming to create homogeneous subsets within each leaf node. The process continues recursively until a stopping criterion is met (e.g., maximum depth, minimum number of samples in a node). In essence, the decision tree model is represented by a series of if-else conditions.

Pros

- Easy to understand and visualize.
- Can handle both categorical and numerical features.
- Can capture non-linear relationships in the data.

Cons

- Prone to overfitting, especially with deep trees.
- Can be unstable, with small changes in the data leading to significant changes in the tree structure.

[2]https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html

Random Forest

Random Forest is an ensemble learning method that constructs multiple decision trees on different subsets of the data and then aggregates their predictions. It introduces randomness by selecting random subsets of features at each node during tree construction, reducing correlation between trees. Each tree is built on a bootstrap sample of the training data (Bagging, Bootstrap Aggregation), meaning a random sample of the data with replacement. This introduces diversity among the trees. At each node of each tree, only a random subset of features is considered for splitting, further increasing diversity (Feature Randomness). For regression, the average of the predictions from all trees is used.

Pros

- High accuracy and robustness to overfitting.
- Handles high-dimensional data well.
- Can handle both categorical and numerical features. #### Cons {-}
- Can be computationally expensive to train, especially with large datasets and many trees.
- Less interpretable than individual decision trees.

K-Nearest Neighbors (kNN)

kNN is a non-parametric, instance-based learning algorithm. It classifies a new data point based on the majority class of its k-nearest neighbors in the training data. The distance between data points is typically calculated using metrics like Euclidean distance or Manhattan distance.[3]

Pros

- Simple and easy to implement.
- No explicit training phase required.
- Can capture complex non-linear relationships.

Cons

- Can be computationally expensive for large datasets.
- Sensitive to the choice of the 'k' parameter and the distance metric.
- Can be sensitive to noisy data and the curse of dimensionality.

```r
# Set a seed for reproducibility
set.seed(123)

# Define a formula for the model
formula <- blueWins ~ .

# Logistic Regression
logistic_model <- train(
  blueWins ~ ., data = train_set,
  method = "glm",
  family = "binomial",
  trControl = trainControl(method = "cv", number = 5)
)

# Decision Tree
decision_tree_model <- train(
  blueWins ~ ., data = train_set,
```

[3] https://www.datacamp.com/tutorial/k-nearest-neighbor-classification-scikit-learn

```
  method = "rpart",
  trControl = trainControl(method = "cv", number = 5)
)

# Random Forest
random_forest_model <- train(
  blueWins ~ ., data = train_set,
  method = "rf",
  trControl = trainControl(method = "cv", number = 5)
)

# K-Nearest Neighbors (KNN)
knn_model <- train(
  blueWins ~ ., data = train_set,
  method = "knn",
  trControl = trainControl(method = "cv", number = 5)
)
```

```
## [1] 0.6982344
```

Here are the results of our model fitting. The highest accuracy is achieved through Logistic Regression

```
# Print the results table
tab7 <- kable(model_results, booktabs = TRUE, escape = FALSE, caption = "Model results") %>%
  kable_styling(position = "center", latex_options = c("striped"))
tab7
```

Table 8: Model results

	Model	Accuracy	Sensitivity	Specificity	Precision	F1
Accuracy	Logistic	0.7439271	0.7408907	0.7469636	0.7454175	0.7431472
Accuracy1	DecisionTree	0.7388664	0.7631579	0.7145749	0.7277992	0.7450593
Accuracy2	RandomForest	0.7287449	0.7307692	0.7267206	0.7278226	0.7292929
Accuracy3	KNN	0.7085020	0.7004049	0.7165992	0.7119342	0.7061224

The Logistic Regression model shows the highest Accuracy (0.7439271), Specificity (0.7469636) and Precision (0.7454175) values of all models. We will try to tune the model parameters in our next steps with the goal to increase its accuracy further.

7 Model fine tuning

In our last step we can use hyperparameter tuning using the R glmnet package. We will add alpha (elastic net mixing parameter) and lambda (regularization parameter) to the tuneGrid variable of our model (see parameter values in the code).

```
# Fine-tuning Logistic Regression using glmnet
logistic_tune_grid <- expand.grid(
  alpha = c(0, 0.5, 1),
  lambda = seq(0.001, 0.1, length = 10)
)

logistic_model_tuned <- train(
  blueWins ~ ., data = train_set,
  method = "glmnet",
```

```
  trControl = trainControl(method = "cv", number = 5),
  tuneGrid = logistic_tune_grid
)

# Predict, create a confusion matrix and extract metrics
model <- logistic_model_tuned
preds <- predict(logistic_model_tuned, test_set)
cm <- confusionMatrix(preds, test_set$blueWins)

# Extract relevant metrics
accuracy <- cm$overall["Accuracy"]
sensitivity <- cm$byClass["Sensitivity"]
specificity <- cm$byClass["Specificity"]
precision <- cm$byClass["Precision"]
f1 <- cm$byClass["F1"]

# Add to results table
model_results <- rbind(model_results, data.frame(
  Model = "Logistic tuned",
  Accuracy = accuracy,
  Sensitivity = sensitivity,
  Specificity = specificity,
  Precision = precision,
  F1 = f1
))
```

The results of our tuned model from its confusion matrix are added added to the result table for comparison. The accuracy value increased by just around 2% compared to our initial model. The final accuracy is 0.7459514.

Table 9: Model results + tuned model

	Model	Accuracy	Sensitivity	Specificity	Precision	F1
Accuracy	Logistic	0.7439271	0.7408907	0.7469636	0.7454175	0.7431472
Accuracy1	DecisionTree	0.7388664	0.7631579	0.7145749	0.7277992	0.7450593
Accuracy2	RandomForest	0.7287449	0.7307692	0.7267206	0.7278226	0.7292929
Accuracy3	KNN	0.7085020	0.7004049	0.7165992	0.7119342	0.7061224
Accuracy4	Logistic tuned	0.7459514	0.7449393	0.7469636	0.7464503	0.7456940

Through the variable importance table we gain insight into which variables were used to predict the final result. In our case the presence of high-level game units (dragons, elite monsters) on the battlefield can change the outcome of the game strongly. Also the importance of blueFirstBlood (meaning a more aggressive, attacking style of the blue player) can help winning the game.

Table 10: Variable Importance

	Overall
blueDragons	100.0000000
redDragons	60.9954981
redEliteMonsters	17.8549500
blueEliteMonsters	12.4996906
blueDeaths	0.6358332

redKills	0.5207252
blueFirstBlood	0.2557544
blueGoldPerMin	0.1771146
redGoldPerMin	0.1500880
blueGoldDiff	0.0543516
redGoldDiff	0.0539509
blueExperienceDiff	0.0476656
redExperienceDiff	0.0470627
redFirstBlood	0.0431848
blueTotalGold	0.0180358
redTotalGold	0.0154453
blueTotalExperience	0.0046656

8 Results and conclusion

By testing four different models to create a machine learning algorithm to predict the value of "blueWins" we achieved the best results with Logistic regression - the probability to correctly predict the win for the blue team lies at around 75%. The accuracy of the model can further be increased by increasing the number of observation in our dataset (e.g. a larger dataset with several millions observations) or increasing the number of variables from late game - currently only the first 10 minutes of each game were captured in our training dataset.

Additional model training runs were conducted (but are not covered in this report) by adjusting the number of variables for all models, such as: - elimination of the "connected" columns identified in our EDA earlier (Accuracy LR: 0.7439271) - reducing the features to blueWins, blueExperienceDiff and blueGoldDiff (Accuracy LR: 0.7317814) - Reducing the features to the 3 above + compound features (blueKDA + blueKillParticipation) (Accuracy LR: 0.7327935)

The only option to (very slightly) increase the accuracy of the models was to train them on a dataset with only "blue" columns (Accuracy LR: 0.7469636). The improvements weren not substantial enough to be detailed further in this report.

Future work to improve the predictive accuracy and precision may include evaluation of other machine learning algorithms, such as XGBoost (Extreme Gradient Boosting) and LightGBM (Light Gradient Boosting Machine).