Forfeit at 15 – A League of Legends Analysis of Early Match Data

CMPT353 – Computational Data Science
Jason Combs
301352433
Dec. 9, 2020

Abstract

League of Legends is a very popular 5v5 team-based strategy game with over 115 million players. While the game can be very enjoyable, at times the game can be very frustrating. A single match of League usually takes about 25-40 minutes on average but can be shortened if there is a majority votes on a team to forfeit at 15 minutes. Because of how long an average match is, a player that is not enjoying themselves and feels that their team is losing will likely want to "FF at 15" and move onto the next game. However, many forfeited matches in League of Legends are still winnable games, should a team choose to not give up. In this project, we plan to explore the likelihood to win a game using data from the first 10 minutes of a League of Legends match and predict the outcome: win or loss.

Data

The data I will be using for this analysis was originally obtained by Michel's Fanboy [1]. The data consists of the first 10 minutes of about 10K solo queue ranked matches played at the Diamond 1 – Master tier level. All games are from the EUW1 region (Europe West). [The dataset can be found here.](https://www.kaggle.com/bobbyscience/league-of-legends-diamond-ranked-games-10-min)[1]

Getting Set up

Imports

```
# Basic imports
import numpy as np
import pandas as pd

# For cleaning data
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import FunctionTransformer
from sklearn import preprocessing

# Visualization imports
import matplotlib.pyplot as plt
import seaborn as sns

# Imports for analysis
from scipy import stats
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import VotingClassifier
```

[1] https://www.kaggle.com/bobbyscience/league-of-legends-diamond-ranked-games-10-min

Importing our data

Lastly, we need to store our data within a Pandas DataFrame.

```
data = pd.read_csv('high_diamond_ranked_10min.csv')
print(data)
```

	gameId	blueWins	blueWardsPlaced	blueWardsDestroyed	blueFirstBlood	...	redTotalJungleMinionsKilled	redGoldDiff	redExperienceDiff	redCSPerMin	redGoldPerMin
0	4519157822	0	28	2	1	...	55	-643	8	19.7	1656.7
1	4523371949	0	12	1	0	...	52	2908	1173	24.0	1762.0
2	4521474530	0	15	0	0	...	28	1172	1033	20.3	1728.5
3	4524384067	0	43	1	0	...	47	1321	7	23.5	1647.8
4	4436033771	0	75	4	0	...	67	1004	-230	22.5	1740.4
...	...	...	...	...	...	...	...	...	...	...	...
9874	4527873286	1	17	2	1	...	34	-2519	-2469	22.9	1524.6
9875	4527797466	1	54	0	0	...	56	-782	-888	20.6	1545.6
9876	4527713716	0	23	1	0	...	60	2416	1877	26.1	1831.9
9877	4527628313	0	14	4	1	...	40	839	1085	24.7	1529.8
9878	4523772935	1	18	0	1	...	46	-927	58	20.1	1533.9

As shown above, our target is the $blueWins$ column. This is column we want to predict. A 0 in this column represents a defeat for the Blue team and a 1 represents a victory. Likewise, 0 represents a Red victory and 1 represents Red defeat.

Data Cleaning

To start off, we will not be needing the $gameId$ for the purposes of this project. We remove this column like so:

```
data = data.drop(columns=['gameId']) # Game ID is not needed for our analysis
```

There are also other values in our dataset we do not need.

```
data = data.drop(columns=['blueDeaths', 'redDeaths', \
    'blueEliteMonsters', 'redEliteMonsters', 'blueTotalMinionsKilled', 'redTotalMinionsKilled', \
        'blueTotalJungleMinionsKilled', 'redTotalJungleMinionsKilled', 'blueCSPerMin', 'redCSPerMin', \
            'redFirstBlood', 'redGoldDiff', 'redExperienceDiff']) # Attributes not needed
```

I found that $blueDeaths$ and $redDeaths$ was redundant since we have attributes $blueKills$ and $redKills$. The amount of $blueKills$ in a game is the same as the amount of $redDeaths$ in a game. Having this redundancy will reduce the accuracy of our prediction model. I also found that $blueEliteMonsters$ was redundant because the number of $blueHeralds + blueDragons = blueEliteMonsters$. For the same reasons as $blueEliteMonsters$, I drop $redEliteMonsters$. For context, an elite monster is a large monster objective in a League match that provides benefits to the team that slays it. Since Baron Nashor spawns at 20 minutes, Baron is not in this dataset. I also felt the need to drop data on $blueTotalMinionsKilled$, $redTotalMinionsKilled$, $blueTotalJungleMinionsKilled$, and $redTotalJungleMinionsKilled$. More context: when killed, minions and jungle monsters are one of the main source of gold generation in the game (the other being killing players). Gold can be used in the shop to purchase items to get stronger. So, "farming" minions and jungle monsters for gold is a big part about winning a match. The reason I remove this data however is because we already have data on $blueTotalGold$ (and $redTotalGold$) which is Blue team's total gold generation after 10 minutes. I believe the amount of gold is a more accurate and better way to know which team is stronger. For the same reasons already mentioned, I remove $blueCSPerMin$ and $redCSPerMin$ which are statistics that represent how many minions and jungle monsters a team kills in a minute. I remove $redFirstBlood$ which represents which team got the first kill of the game since we already have $blueFirstBlood$ which tells us this information. I also remove $redGoldDiff$ which represents the gold differential after 10 minutes between teams Red and Blue since we have $blueGoldDiff$. Lastly, I remove $redExperienceDiff$ because we already have $blueExperienceDiff$. Note that these statistics represent the amount of experience champion (which is a playable character) has. A champion can level up with enough experience and get stronger.

Next off, I added a few columns to the dataset to show "Victory" or "Defeat" instead of boring "0" and "1". This step is not necessary, but it makes our data easier to understand and visualize in some plots! Note that I later remove these columns before doing our data analysis.

```
blueTeamdict = {1 : "Victory", 0 : "Defeat"}
redTeamdict = {0 : "Victory", 1 : "Defeat"}
data['blueResult'] = data['blueWins'].map(blueTeamdict)
data['redResult'] = data['blueWins'].map(redTeamdict)
```

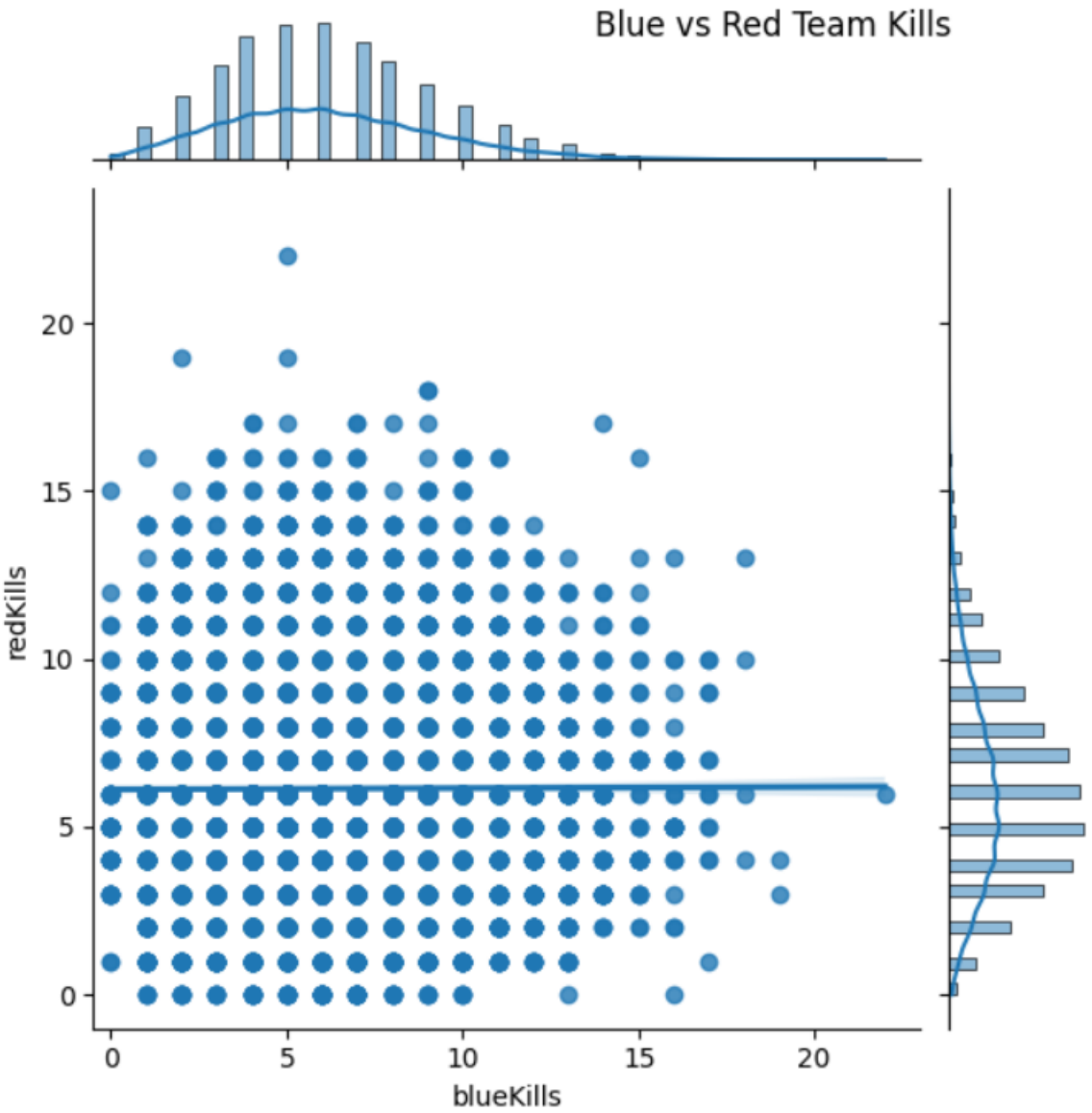
Now that our data cleaning setup is complete, lets move onto the more exciting stuff!

Data Visualization

We got our data ready, lets take a look!

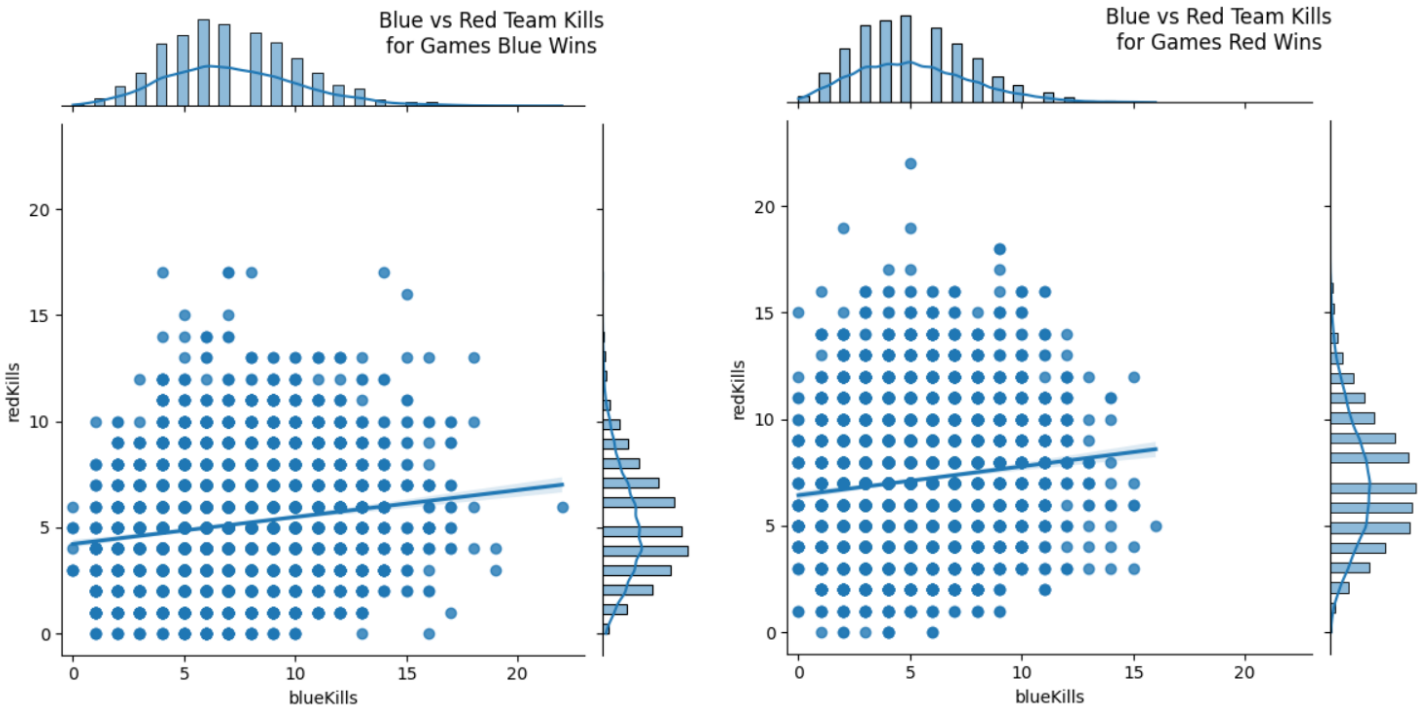
I use Seaborn to make a scatterplot of team Blue kills versus team Red kills just for fun. Here's the result:

```
sns.jointplot(x=data['blueKills'], y=data['redKills'], kind="reg")
plt.suptitle('Blue vs Red Team Kills', x=0.7)
plt.xlim(-.5, 23)
plt.ylim(-1, 24)
plt.show()
```



Note that the data appears to be evenly distributed around the bottom left corner of the plot. This is to be expected because this means that over approximately 10K games where approximately 50% of these games were wins for Blue team (and the other 50% were wins for Red team), the number of kills between each team is roughly even.

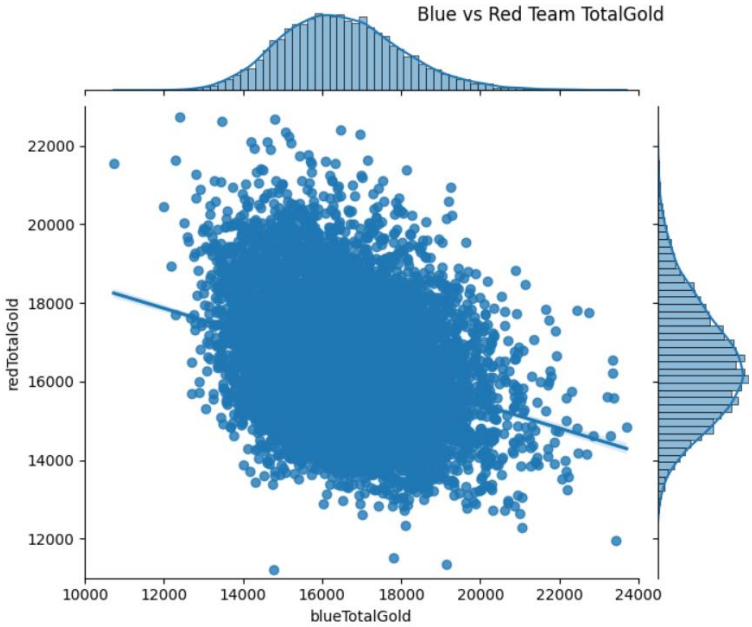
I also made two more scatter plots to compare what the data looks like in games where Blue wins or Red wins:



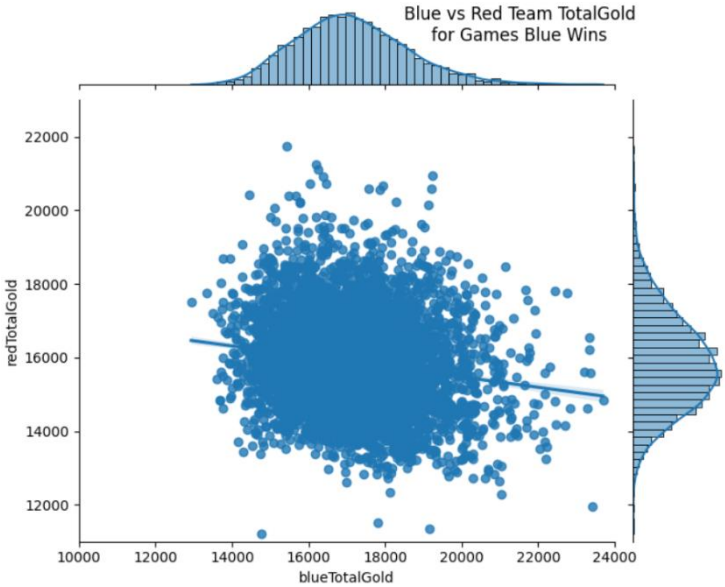
We can see that on games where Blue wins, the data points are scattered more to the right showing they had more kills than team Red. Similarly, games where Red wins has the data points scattered more towards the top of the plot. I also provided a histogram to better showcase the amount dots in each area in the scatterplot since we cannot see how many points are in an area just from the scatterplot.

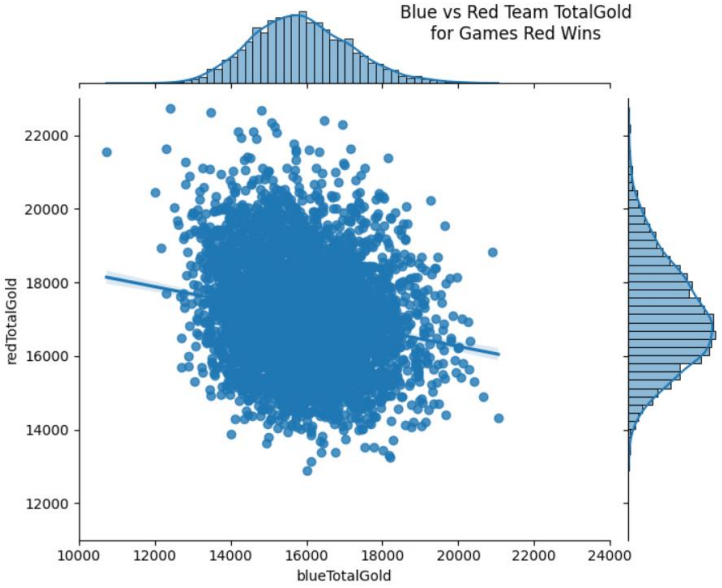
I chose to plot the number of kills for each team because kills in a match are very important when deciding the outcome of a game and can benefit the team that got the kill in many ways. For one, a kill gives a player (and any teammate who assisted in the kill) a large sum of gold. Additionally, within next minute or so of getting the kill, the player has free reign to farm gold and experience without the pressure of fighting the enemy. The dead player must also wait to respawn back at base, which results in giving the player less time to earn gold and experience. A kill can also make it easier for a team to secure an objective like Dragon or Rift Herald for a further advantage. One last benefit I will mention is that the kill can cause the player that got the kill to “snowball”. Just like how a snowball rolls down a hill and gets larger and larger, a player can get stronger and stronger based on the amount of gold and experience they earn. This advantage makes it easier to dominate your lane opponent, which in turn, helps the chances for your team to defeat and overpower the enemy.

Speaking of advantages, gold is the most important advantage to have if a team wants to win a match. Gold controls nearly everything. The whole point of the early game (first 10 minutes and afterwards) is to earn as much gold as possible over your opponent and grow that gold lead into an advantage whether that advantage be getting kills, or taking objectives. Here is what the gold generation for both teams looks like after the first 10 minutes of a game:



I show this result to get a base idea of how the data looks. Here is how it looks in games where one team beats the other.





Once again, we can see how the data is shifted to the right when team Blue wins. This is because Blue has earned more gold than Red. The same case can be seen in the second plot. Since mass of data points may be hard to read, histograms have been provided. Note how the histogram on the x-axis in the first image shifts over to the left in the second since due to the change in gold.

There is more data I could plot and show, but these are the two major things I wanted to show in this report before talking about predicting the outcome of a match (up next!).

Analysis

Back to working with our data. My first step was to split up my data into training data and validation data:

```
X_train, X_valid, y_train, y_valid = train_test_split(X, y)
```

The y values are the targets (victory or defeat). The X values are our data (every other column of data that we did not remove). The purpose of splitting our data up like this is so that we can train a Machine Learning model. Our models will be training on X_train and compare what the result of a match is, y_train. The model learns this way by making many comparisons. Once the model is trained, we want to see how well the model does. We cannot test the model on the data it was trained on (the training data) since the model has learned from that data. We want to see how our model does on data it has never seen before: our validation data. I will use many different models and show the training and validation scores.

The first model I tried was a Naïve Bayes classifier, ($GaussianNB$):

```
gauss_model = make_pipeline(
        MinMaxScaler(),
        GaussianNB()
)
gauss_model.fit(X_train, y_train)
print(gauss_model.score(X_train, y_train))
print(gauss_model.score(X_valid, y_valid))
```

I will be used $MinMaxScaler()$ on some of my models to have values scaled in a [0, 1] range since some models have trouble when a value like gold is 23,000 and another value like towers destroyed is only 1. The training and validation scores of this model were:

```
0.7284383857470644
0.7246963562753036
```

I then tried a Random Forest classifier ($RandomForestClassifier$):

```
rf_model = make_pipeline(
        MinMaxScaler(),
        RandomForestClassifier(n_estimators=100, max_depth=4)
)
rf_model.fit(X_train, y_train)
print(rf_model.score(X_train, y_train))
print(rf_model.score(X_valid, y_valid))
```

```
0.734782021865299
0.7323886639676114
```

A C-Support Vector classifier (SVC):

```
svc_model = make_pipeline(
        MinMaxScaler(),
        SVC(kernel='linear', C=0.01)
)
svc_model.fit(X_train, y_train)
print(svc_model.score(X_train, y_train))
print(svc_model.score(X_valid, y_valid))
```

```
0.7270886759346741
0.719838056680162
```

A Gradient Boosting classifier ($GradientBoostingClassifier$):

```
gradient_model = GradientBoostingClassifier(n_estimators=50, max_depth=2, min_samples_leaf=0.1)
gradient_model.fit(X_train, y_train)
print(gradient_model.score(X_train, y_train))
print(gradient_model.score(X_valid, y_valid))
```

```
0.7362667026589284
0.7319838056680162
```

And three MLP classifiers ($MLPClassifier$):

```
mlp_model1 = make_pipeline(
    MinMaxScaler(),
    MLPClassifier(max_iter=10000, activation='identity', solver='sgd', hidden_layer_sizes=(8, 5))
)
mlp_model1.fit(X_train, y_train)
print(mlp_model1.score(X_train, y_train))
print(mlp_model1.score(X_valid, y_valid))
```

```
0.7261438790660009
0.7299595141700405
```

```
mlp_model2 = make_pipeline(
    MinMaxScaler(),
    MLPClassifier(max_iter=10000, activation='identity', solver='sgd', hidden_layer_sizes=(8, 6, 4))
)
mlp_model2.fit(X_train, y_train)
print(mlp_model2.score(X_train, y_train))
print(mlp_model2.score(X_valid, y_valid))
```

```
0.7292482116344986
0.7384615384615385
```

```
mlp_model3 = make_pipeline(
    MinMaxScaler(),
    MLPClassifier(max_iter=10000, activation='identity', solver='adam', hidden_layer_sizes=(5, 4, 3))
)
mlp_model3.fit(X_train, y_train)
print(mlp_model3.score(X_train, y_train))
print(mlp_model3.score(X_valid, y_valid))
```

```
0.7257389661222837
0.7336032388663968
```

As you can see, the training and validation scores of all models are very close in value. This means that we are not overfitting or underfitting on our training data, which is good! You can also see that I tried, in ~~Vayne~~ vain (League of Legends joke!), to get better numbers on my models. It doesn't seem easy to get any better than this that's for sure. So, I thought, since all these classifiers are similar, maybe a Voting Classifier ($VotingClassifier$) would help:

```
model = VotingClassifier(estimators=[
    ('gauss', gauss_model),
    ('rf', rf_model),
    ('svc', svc_model),
    ('grad', gradient_model),
    ('mlp1', mlp_model1),
    ('mlp2', mlp_model2),
    ('mlp3', mlp_model3)
    ],
    voting='hard')
model.fit(X_train, y_train)
print(model.score(X_train, y_train))
print(model.score(X_valid, y_valid))
```

This voting classifier is essentially getting all the results from each model and doing a vote on the best decisions the models have learnt. The result should be better than any of the models individually:

```
0.7326224861654744
0.7299595141700405
```

From all these results, it seems apparent that results at around 0.73 is as best we are going to get. This is FINE though… And totally expected! We are only using the first 10 minutes of match data to make these predictions. If we had more than just the first 10 minutes, we would have much better predictions. Still, predicting the result of a match based on only the first 10 minutes of a game and getting the prediction correct 73% of the time, is actually really good!! Now for all those League players who get tired of playing and spam "FF" in the chat, this project can reveal *if they really should* forfeit a match and give it second thought.

Conclusion

I wish I had more time do work on this project. One of the things I really wanted to do was find out which elements of the game have the most influence on the result of a match, whether it be the number of kills or gold generation per minute. A problem I encountered was trying to extract my own data using a League of Legends API key. It was very difficult to figure out, and even harder to get a dataset I could be happy with. Nevertheless, shoutout to Michel for this amazing dataset! In retrospect, I wish I did more data visualization to show off my results. I also wish I found another dataset I could have made predictions from too.

References:

[1.] Kaggle. Michel's Fanboy. League of Legends Diamond Ranked Games (10 min). [My dataset] https://www.kaggle.com/bobbyscience/league-of-legends-diamond-ranked-games-10-min [Accessed on Nov. 2020]

[2.] Kaggle. Jay Tegge. League of Legends data analysis. https://www.kaggle.com/jaytegge/league-of-legends-data-analysis [Accessed on Nov. 2020]

[3.] Towards Data Science. Marco Sanguineti. https://towardsdatascience.com/riot-api-a-machine-learning-and-data-analysis-application-c8524b4160b4 [Accessed on Nov. 2020]