Capstone Project 1: League of Legends

Introduction

League of Legends is a multiplayer online battle arena (MOBA) developed and published by Riot Games. Two teams navigate an arena fighting to destroy the opposing team's base. League of Legends is an extremely complex game, with numerous factors influencing the outcome of each game. This project aims to create a model capable of predicting the outcome of a game given these features. This model will be built on the League of Legends dataset acquired from Kaggle. The dataset contains information and statistics on professional League of Legends games from 2014 to 2018.

This analysis has the potential to influence what direction Riot Games will take regarding the gameplay. By understanding what game mechanics are the most significant, the game developers can choose to target and balance those factors, potentially increasing the popularity of League of Legends and Riot Games' revenue. Additionally, if professional teams are able to determine the expected outcome of their competitive matches, and what is likely to lead to their victory or defeat, they can prepare better for their matches.

Game Explanation

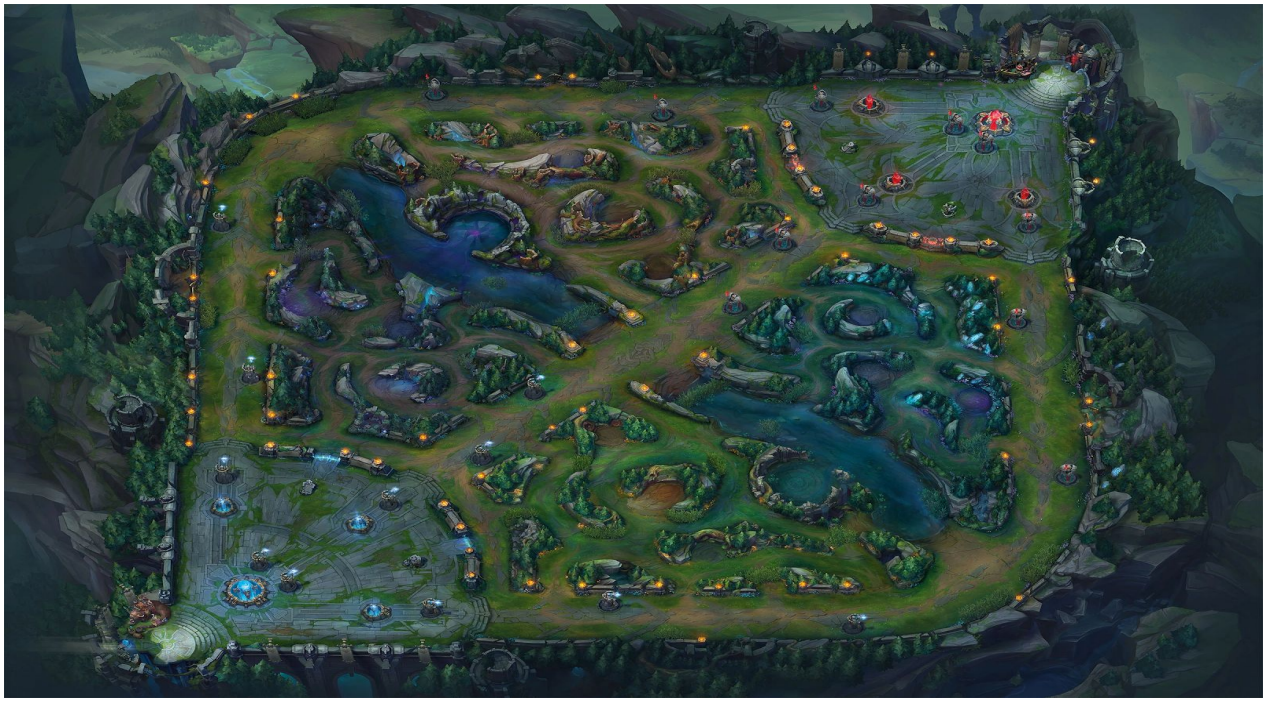

Figure 1. League of Legends Game Map

League of legends is a multiplayer online battle arena (MOBA). There are two teams of 5 players and the purpose of the game is to destroy the opposing team's base structure, also known as a Nexus. In order to reach the Nexus, the opposing team must destroy the towers protecting the Nexus. There are three lanes with 3 towers each. There is a structure in each lane called an inhibitor which buffs the invading team's minions. There are an additional two towers located by the Nexus. The purpose of the towers is to protect their teams. Minions are creatures that appear through each lane and autonomously attempt to invade the enemy team's base and destroy their structures. In competitive games, the roles are as follows:

- Top Laner, located in top lane
- Jungle, resides within the area of the map that is not part of the lanes
- Mid Laner, located in the middle lane
- ADC, located at the bottom lane
- Support, located at the bottom lane

There are multiple objectives throughout the game which empower the team they are taken by. The names of these objectives are the rift herald, dragons, and barons. These are highly contested and are often fought for by each team. There is no time limit within the game, and champions (which the players control) are strengthened through items bought by gold.

Dataset Description

The league of legends dataset contains game metrics regarding competitive league of legends matches from 2014 to 2018. It contains seven csv files, one of which is an aggregation of the other six. The other six cover the following topics:

1. Bans
2. Gold
3. Kills
4. Matchinfo
5. Monsters
6. Structures

For this project, the majority of the processing is done on the summary csv file, titled "LeagueofLegends.csv." This is done to avoid unnecessary merging and joining steps, particularly with regards to visualizing the data..

Data Wrangling

First, the `.head()`, `.describe()`, and `.info()` methods were called on the LeagueofLegends dataframe. It was observed that some columns contained missing values. Upon further inspection, this was a result of the format of the specific tournament the data was describing. Values such as a team name, or dragons were missing. Because this dataset was fairly large (~8000 samples) and the number of missing values was quite small (~40), it was decided that the rows containing missing values could be dropped without significant consequence. Afterwards, each column of the dataframe was inspected in order to determine whether it would need to be processed and how to properly process it. Categorical variables were dummy encoded, the variances of lists were calculated, and columns describing objectives were counted. A summary of how each column was processed is displayed in the table below. The functions used to perform each of these transformations are also included.

The list of towers taken by each team required an additional processing step. In order to ensure that the returned values remain numerical, each tower received a unique id and converted to a column, where the value was the time the tower was taken. If the tower was not destroyed within the span of that specific match, the value was set to 0.

Table I. Summary of data wrangling procedures.

Column(s)	Procedure
Address, rResult	Dropped
League, Year, Season, Type, blueTeamTag, redTeamTag, blueTop, blueTopChamp, blueJungle, blueJungleChamp, blueMiddle, blueMiddleChamp, blueADC, blueADCChamp, blueSupport, blueSupportChamp, redTop, redTopChamp, redJungle, redJungleChamp, redMiddle, redMiddleChamp, redADC, redADCChamp, redSupport, redSupportChamp	Dummy encoded
golddiff, goldblue, goldred, goldblueTop, goldblueJungle, goldblueMiddle, goldblueADC, goldblueSupport, goldredTop, goldredJungle, goldredMiddle, goldredADC, goldredSupport	Calculate Variance
bKills, bInhibs, bDragons, bBarons, bHeralds, rKills, rInhibs, rDragons, rDragons, rHeralds	Count
bDragons, rDragons	Pivot

The functions used to process the columns are the following:

```
def bans_to_encodable_cols(df, cols, col_names):
    for col, names in zip(cols, col_names):
        df[col] = df[col].apply(ast.literal_eval)
        encodable_cols = pd.DataFrame(df[col].values.tolist(), index=league_df.index, columns=names)
        df = pd.concat([df.drop(col, axis=1), encodable_cols], axis = 1)

    return df
```

```
def delete_col_get_dummies(df, cols):

    for col in cols:
        if 'blue' in col:
            if 'Top' in col:
                df = pd.concat([df.drop(col, axis=1), pd.get_dummies(df[col], prefix='bTop')], axis=1) # add prefix or suffix
            elif 'Jungle' in col:
                df = pd.concat([df.drop(col, axis=1), pd.get_dummies(df[col], prefix='bJungle')], axis=1) # add prefix or suffix
            elif 'Middle' in col:
                df = pd.concat([df.drop(col, axis=1), pd.get_dummies(df[col], prefix='bMiddle')], axis=1) # add prefix or suffix
            elif 'ADC' in col:
                df = pd.concat([df.drop(col, axis=1), pd.get_dummies(df[col], prefix='bADC')], axis=1) # add prefix or suffix
            elif 'Support' in col:
                df = pd.concat([df.drop(col, axis=1), pd.get_dummies(df[col], prefix='bSupport')], axis=1) # add prefix or suffi
            else:
                df = pd.concat([df.drop(col, axis=1), pd.get_dummies(df[col], prefix='b')], axis=1) # add prefix or suffix
        elif 'red' in col:
            if 'Top' in col:
                df = pd.concat([df.drop(col, axis=1), pd.get_dummies(df[col], prefix='rTop')], axis=1) # add prefix or suffix
            elif 'Jungle' in col:
                df = pd.concat([df.drop(col, axis=1), pd.get_dummies(df[col], prefix='rJungle')], axis=1) # add prefix or suffix
            elif 'Middle' in col:
                df = pd.concat([df.drop(col, axis=1), pd.get_dummies(df[col], prefix='rMiddle')], axis=1) # add prefix or suffix
            elif 'ADC' in col:
                df = pd.concat([df.drop(col, axis=1), pd.get_dummies(df[col], prefix='rADC')], axis=1) # add prefix or suffix
            elif 'Support' in col:
                df = pd.concat([df.drop(col, axis=1), pd.get_dummies(df[col], prefix='rSupport')], axis=1) # add prefix or suffi
            else:
                df = pd.concat([df.drop(col, axis=1), pd.get_dummies(df[col], prefix='r')], axis=1) # add prefix or suffix
        else:
            df = pd.concat([df.drop(col, axis=1), pd.get_dummies(df[col])], axis=1) # add prefix or suffix

    return df
```

```
def get_drag_type(drag_lst):
    return [drag[1] for drag in drag_lst]

def process_dragons(df, cols, col_names):

    drag_df_lists = []
    for col, names in zip(cols, col_names):
        df[col] = df[col].apply(ast.literal_eval).apply(get_drag_type)
        df['num_of_drags'] = df[col].apply(len)

        dict_list = df[col].apply(Counter).tolist()
        drag_df = pd.DataFrame.from_dict(dict_list)
        drag_df.fillna(0, inplace=True)
        drag_df.columns = names
        drag_df.reset_index(drop=True, inplace=True)
        drag_df_lists.append(drag_df)

    df.reset_index(drop=True, inplace=True)
    df = pd.concat([df.drop(cols, axis=1)]+drag_df_lists, axis=1)

    return df
```

```
def var_of_column(df, cols):
    for col in cols:
        df[col] = df[col].apply(ast.literal_eval).apply(np.var)

    return df
```

```
def count_num(df, cols):

    for col in cols:
        df[col] = df[col].apply(ast.literal_eval).apply(len)

    return df
```

```
tower_dict = {'TOP_LANE': {'OUTER_TURRET': 0, 'INNER_TURRET': 1, 'BASE_TURRET': 2},
              'MID_LANE': {'OUTER_TURRET': 3, 'INNER_TURRET': 4, 'BASE_TURRET': 5},
              'BOT_LANE': {'OUTER_TURRET': 6, 'INNER_TURRET': 7, 'BASE_TURRET': 8}}

def convert_to_tower_list(tower_str):

    tower_list = [0,0,0,0,0,0,0,0,0]
    for towers_destroyed in ast.literal_eval(tower_str):
        if len(towers_destroyed) > 0:
            try:
                index = tower_dict[towers_destroyed[1]][towers_destroyed[2]]
                tower_list[index] = towers_destroyed[0]
            except:
                pass

    return tower_list


def process_towers(df, cols, col_names):

    tower_df_lists = []
    for col, names in zip(cols, col_names):
        df[col] = df[col].apply(convert_to_tower_list)
        towers_df = pd.DataFrame(df[col].values.tolist(), columns=names)
        towers_df.head()
        towers_df.reset_index(drop=True, inplace=True)
        tower_df_lists.append(towers_df)

    df.reset_index(drop=True, inplace=True)
    df=pd.concat([df.drop(cols, axis=1)] + tower_df_lists, axis=1)

    return df
```

Figure 2. Functions used to process and clean data.

Data Storytelling

Upon completion of the data wrangling, the data was explored in order to find possible trends within the data. One of notable data visualizations was a heatmap showing where most of the in-game kills occurred during each year. The years 2016 and 2017 were the most insightful, showing that much of the in-game kills occurred around the center of the map, particularly near the baron and dragon pit. This makes sense as these are two highly contested objectives. The plots are shown in the figure below.

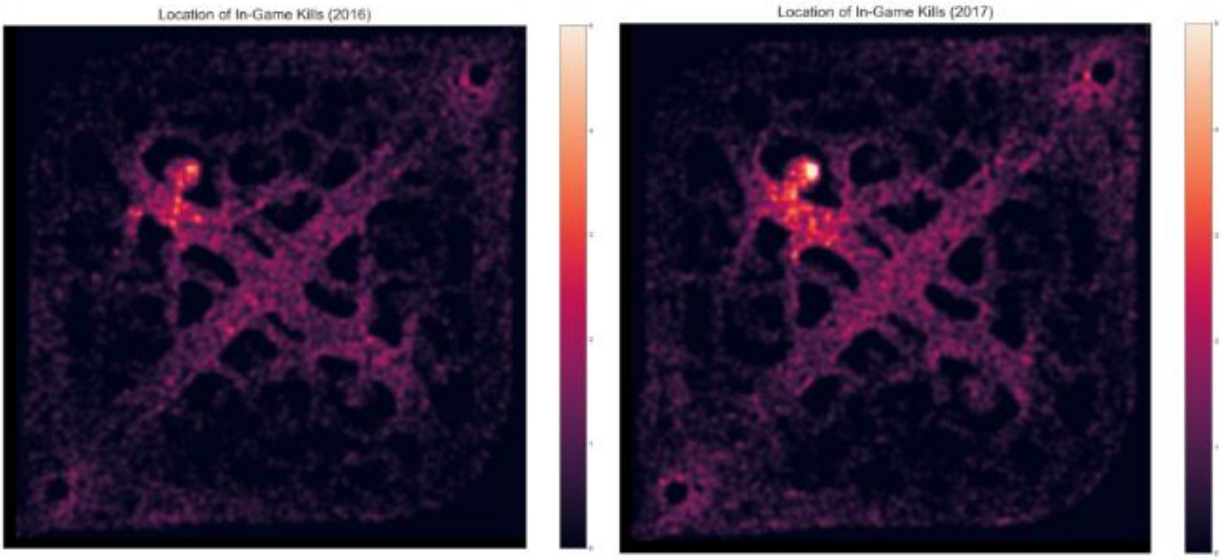

Figure 3. Heatmaps of competitive in-game kills during 2016 and 2017.

Additionally, in order to determine which champions were the most impactful within the game, which also might indicate a higher chance to win the game, the most frequently picked champs in each role per season by side was shown through a table. An example of the table is shown below.

Table II. Table depicting most frequently picked champs on the blue side for the top lane.

Three Most Picked Top Champs for Blue Side

Year	Season		
2014	Summer	Maokai	21
		Ryze	18
		Rumble	13
2015	Spring	Maokai	123
		Gnar	104
		Rumble	73
	Summer	Maokai	172
		Rumble	123
		Gnar	94
2016	Spring	Poppy	200
		Nautilus	119
		Fiora	106
	Summer	Trundle	213
		Gnar	169
		Shen	166
2017	Spring	Nautilus	238
		Maokai	234
		Shen	195
	Summer	JarvanIV	220
		Renekton	204
		Shen	183
2018	Spring	Gnar	51
		Ornn	34
		Gangplank	34

From these plots, it was easy to determine which champions were considered strong within their respective roles. The most picked champions were not consistent per side, which is likely a result of how the pick phase within the game occurs. The blue side picks first, which means that the most frequently chosen champions for the red side are often responses to the most frequently picked champions for the blue side.

In order to accomplish the same task of determining which champions were the most impactful within the game, the champions who had the highest ban rate were plotted. An example is shown below.

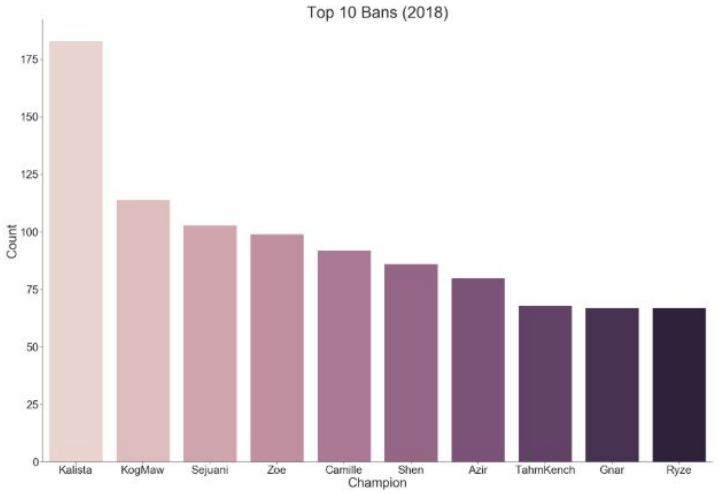

Figure 4. Countplot the 10 most frequently banned champions for 2018.

From these plots, the most frequently banned champions for each year were as follows:

1. 2014 - Alistar
2. 2015 - Kalista
3. 2016 - Nidalee
4. 2017 - Leblanc
5. 2018 - Kalista

Statistical Data Analysis

In order to determine whether some variables were indicators of whether a team would win, the p-value of the following variables were calculated using bootstrapping:

1. Gamelength
2. TIme of first tower taken
3. Number of each type of dragon
    a. Earth
    b. Ocean
    c. Air
    d. Fire
4. Mean gold of winning team by role
    a. Top

b. Jungle
c. Mid
d. ADC
e. Support

A function was defined that generalized the bootstrapping procedure. It is shown in Figure 5.

```
def calculate_p(blue, red, feature, N=10000):
    bs_mean_diff = np.empty(N)

    mean_total   = np.mean(feature)
    blue_shifted = blue - np.mean(blue) + mean_total
    red_shifted  = red - np.mean(red) + mean_total

    for i in range(N):
        bs_blue = np.random.choice(blue_shifted, size=len(league))
        bs_red = np.random.choice(red_shifted, size=len(league))
        bs_mean_diff[i] = np.mean(bs_blue) - np.mean(bs_red)

    emperical_mean_diff = np.mean(blue) - np.mean(red)
    p = np.sum(bs_mean_diff >= emperical_mean_diff) / len(gamelength_mean_diff)
    return p
```

Figure 5. Function to calculate p-value through bootstrapping.

Of the variables tested, only the null hypothesis of the difference in infernal dragons taken between the blue team and red team could be rejected using an alpha of 0.05. There was a clear difference in the number of infernal dragons the blue team retrieved when they won versus the number of infernal dragons the red team retrieved when they won.

Additionally, correlations between champions were plotted to determine whether certain champions were more likely to be chosen together. An example of this correlation plot is shown in Figure 6.

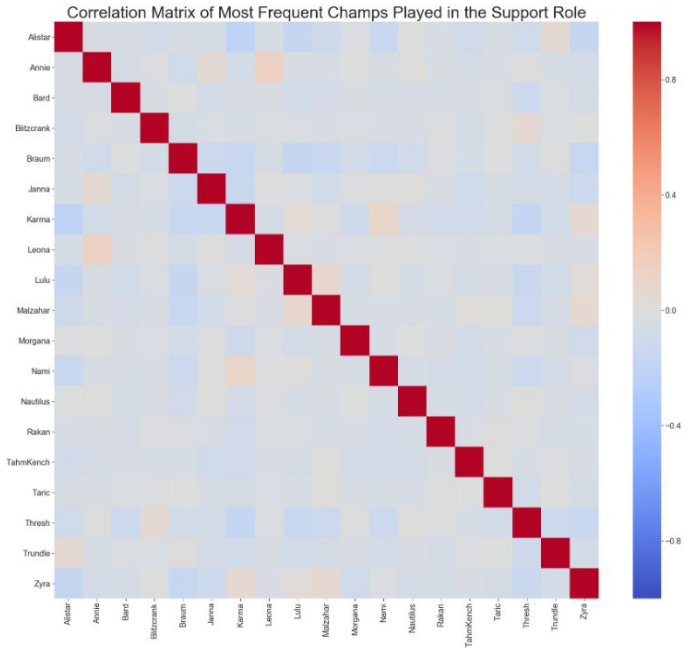

Figure 6. Correlation matrix between support champions.

Due to the diverse champion pool within the game, many of the correlations are quite low, however, there are a few champions that are highly correlated. In the plot above, it can be observed that the champions Leona and Annie are significantly correlated. Relevant correlations are described below are:

1. Top Lane
    a. None
2. Jungle
    a. Graves and Rengar
    b. Lee sin and Khazix
3. Mid
    a. Syndra and Ekko
4. ADC
    a. Graves and Corki
    b. Tristana and Xayah
5. Support
    a. Leona and Annie
    b. Nami and Karma

Machine Learning Analysis

When performing analysis on this dataset, the types of classifiers used must first be determined. Because this was a labeled dataset, supervised machine learning techniques were used. Of the possible supervised machine learning techniques, the random forest classifier, support vector classifier, and neural network were selected. The random forest classifier was selected because of its robustness and ability to perform well on datasets with many features. The support vector machine was selected because it tends to perform well when there is a clear separation between classes. The dataset contains only two classes, games where the blue team won and games where the red team won, where only one team can win, so there is indeed a clear separation between classes. Finally, a neural network was chosen because of its ability to model complex functions. League of Legends games are extremely complex with thousands of factors that can impact the game. Neural networks were chosen because they are likely able to understand how these factors interact and determine the outcome of a game.

Random Forest

The first step in fitting a random forest to the dataset is splitting the dataset into a training and testing data set. This was done using Scikit-Learn's `train_test_split`function. This is shown in Figure 7 below.

```
# First split the data into training and testing sets
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(league_wrangled, result,
                                                    test_size=0.33, random_state=42)


print('Training Features Shape:', X_train.shape)
print('Training Labels Shape:', y_train.shape)
print('Testing Features Shape:', X_test.shape)
print('Testing Labels Shape:', y_test.shape)
```

```
Training Features Shape: (5079, 5719)
Training Labels Shape: (5079,)
Testing Features Shape: (2503, 5719)
Testing Labels Shape: (2503,)
```

Figure 7. Splitting dataset into training and testing sets.

Next, a RandomForestClassifier class object is created from Scikit-Learn's implementation and then fit onto the dataset.

```
# Import the Random Forest classifier
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(n_estimators=1000, random_state=42)
rf.fit(X_train, y_train)
```

Figure 8. Creating and fitting Random Forest.

The classifier was able to achieve an accuracy of 0.9648. The confusion matrix of the results are plotted in the heatmap in Figure 9.

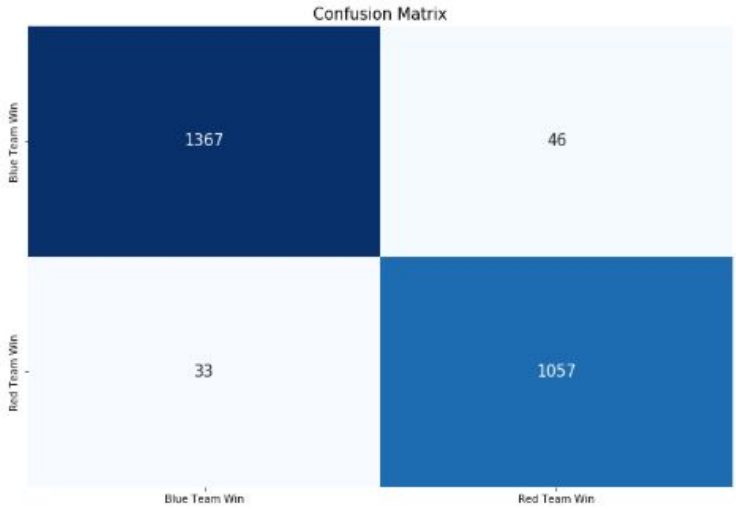

Figure 9. Confusion matrix of results.

From the confusion matrix, the classifier incorrectly predicted only 79 samples out of the 2503 samples within the testing dataset. Additionally, the most significant features were displayed using the random forest classifier's attribute `.feature_importances_`. The five most significant are displayed below in Figure 10.

```
Variable: bInhibs              Importance: 0.0745
Variable: rInhibs              Importance: 0.06558
Variable: rmiddle_inner        Importance: 0.04422
Variable: bbmiddle_base        Importance: 0.04298
Variable: rbmiddle_base        Importance: 0.04043
```

Figure 10. Five most important features in random forest.

Based on the values, it is clear that games are not decided by a single feature alone, but rather a combination of multiple features.

Support Vector Classifier (SVC)

In order to create the support vector classifier, Scikit-Learn's implementation will be used. In order to train the SVC, the training and testing dataset must first be scaled. This is done using Scikit-Learn's StandardScaler() object. The features must be scaled in order to prevent the feature with the largest range from completely dominating the results of the SVC. The SVC is instantiated and trained in the same way as the random forest classifier: by first instantiating the SVC object and calling the fit method on the object. This is shown below in Figure 11.

```
# Import SVM
from sklearn.svm import SVC

svc = SVC(gamma="scale")
svc.fit(X_train_ss, y_train_ss)
```

Figure 11. Creation and Training of Support Vector Classifier.

The SVC was able to achieve an accuracy of 0.9373, performing slightly worse than the random forest.

Neural Network

Due to how complicated neural networks can be, and the number of features, a simple network architecture is created first. This is done using TensorFlow's Keras API. The architecture is shown in Figure 12.

```
Model: "base_league_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 5719)]            0
_________________________________________________________________
dense_302 (Dense)            (None, 2859)              16353480
_________________________________________________________________
dense_303 (Dense)            (None, 1)                 2860
=================================================================
Total params: 16,356,340
Trainable params: 16,356,340
Non-trainable params: 0
_________________________________________________________________
```

Figure 12. Simple Neural Network Architecture.

The neural network is then trained on the scaled training and testing data. This model performed the worst, at an accuracy of 0.56 and loss of 6.677.

Improving models

The three models used in this analysis all contain many hyperparameters capable of affecting the performance of each. To optimize their performances, GridSearchCV is used. This is done by creating parameter grids, which GridSearchCV iterates over to determine which combination of parameters results in the best score. The parameter grids are shown in Figure 13 below.

```
from sklearn.model_selection import GridSearchCV

rf_parameters = {
    'n_estimators':[100,200,500,1000],
    'criterion':['gini', 'entropy'],
    'max_features':['auto', 'sqrt', 'log2', None]
}

svc_parameters = {
    'C':[0.001, 0.01, 0.1, 1, 10],
    'gamma': ['auto', 'scale']
}

nn_parameters = {
    'dropout_rate': [0.5, 0.2],
    'neurons': [1000, 250],
    'batch_size': [32, 8],
    'epochs': [10, 100]
}
```

Figure 13. GridSearchCV parameter grids.

Using GridSearchCV, the Random Forest Classifier's accuracy increased from 0.9684 to 0.9812, while the Support Vector Classifier's accuracy decreased to 0.9368 from 0.9372. Due to time constraints, it is difficult to test neural network architectures, but it appears that it would require a much larger neural network in order to accurately classify the game winner based on the results of GridSearchCV. The neural network, despite having 4 additional layers, was only able to reach an accuracy of 0.524.

An additional approach taken for the neural network was to train the network using the features with the highest feature importances from the random forest. This method was unsuccessful and the network remained at an accuracy of 0.5344.

Conclusions

League of Legends is an extremely complex game, with high-stakes during competitive matches. This project attempted to create a model with the ability to predict the outcomes of such competitive matches. The most successful model in solving this problem was the random forest classifier, with a classification accuracy of 0.9812. The other problem this project attempted to solve was determining which features are mostly likely to indicate a win for either team. The random forest classifier was also able to solve this problem due to its ability to calculate feature importances. The most important features according to the random forest were the number of inhibitors taken by each team, the time the middle turret was taken by either team, the number of kills by each team, and the number of barons retrieved by each team. Inhibitors are extremely important objectives within the game as they pressure the opposing team and allow the other team to focus on other parts of the map. The middle turret is extremely important because it allows the team who took it access to many parts of the other team’s side of the map. The number of kills are important in removing the other team’s presence on the map and acquiring gold to strengthen the champion who acquired the kill. Finally, acquiring the baron gives an extremely powerful teamwide buff that strengthens the acquiring team’s champions and minions. This gives them a greater ability to siege objectives on the map, based on map pressure and fighting ability.