League of Legends - Predicting Wins from Champions Selected

May 6, 2018

Justin Chou 156006871

1 Project Description

League of Legends (LoL) is an increasingly popular e-sport for both the gamer and the spectator. Each game brings together ten human players, each on different champions (playable characters), who purchase different items and cast different available spells in game. While most of the game is spent battling on the rift ("playing" the actual game), there is still much strategizing to be done before the game even starts – that is selecting which champions to play. Since there are a total of 140 champions and games consist of 5 players vs. 5 players, in total there are $\frac{\binom{140}{5} * \binom{135}{5}}{2} = 7.22 * 10^{16}$ unique possible games with respect to the possible combinations of champions for each team (no duplicate champions in a game). I believe this aspect of the game is worth investigating as the composition of teams fundamentally layout how the game will play out. Each champion has its own advantages, disadvantages, and synergies with other champions, so naturally, before the game starts, the probability of one team winning over the other should not be 50%. Thus, with previously played games as data, ideally one should be able to train a model that is capable of predicting the winning team based on the selected champions to a certain degree of accuracy, as there are other factors like individual player skill not taken into consideration.

2 Data Collection

My main goal was to collect large amounts of data on played league games – specifically, what champions are being played on each team, as well as which team wins. Fortunately, Riot Games supports an API for League of Legends at https://developer.riotgames.com/. Since the API does not store any universal list of all matches played, I had to collect data by crawling through matches. The main API method I use is MATCH-V3, which can return a list of matches played given an accountId parameter, and can also return details of a match given a matchId. I first collected the a list of matches played by one known account. I began with my personal account. I then collected the players in these matches, recorded their match histories, and repeated the process for these new players. Riot's API enforces a rate limit of 100 requests per 2 minutes, so I had to adjust my request rate accordingly. In the end, I was able to collect data for about 34,000 games, with approximately 360 duplicated rows that were ultimately removed. Below are the methods that scraped Riot's API to gather the data.

```
In [ ]: #See Main.py for full data scraping code & dependencies including RiotAPI
        #API key needs to be updated

        import csv
        import random
        import time
        from RiotAPI import RiotAPI

        def main():
            api_key='RGAPI-f2907155-7488-4446-9b9c-ef8332e10674'
            #collect_champions(api_key)
            scrape_match_data(api_key)

        def collect_champions(api_key):
            csvfile = 'champions_data.csv'
            output = open(csvfile,'a')
            api = RiotAPI(api_key)
            r = api.get_champion_list()
            while r is None or r.get('data') is None    :
                r = api.get_champion_list()

            print(r.get('data'))
            for champion in r.get('data'):
                temp = r.get('data')[champion]
                c = [temp.get('name'),temp.get('id')]
                writer = csv.writer(output, lineterminator='\n')
                writer.writerow(c)
            output.close()


        def scrape_match_data(api_key):
            csvfile = 'match_data2.csv'
            matches = []
            searched_matches = []
            accounts = []
            searched_accounts = []
            count = 0

            accounts.append('32639237')

            api = RiotAPI(api_key)

            while accounts:
                nextAccount = accounts.pop()
                r = api.get_matchlist(nextAccount)
                while r is None or r.get('matches') is None:
                    time.sleep(3)
                    #nextAccount = accounts.pop()
```

```
    r = api.get_matchlist(nextAccount)

searched_accounts.append(nextAccount)

for match in r.get('matches'):
    if match.get('platformId')=='NA1':
        Id = match.get('gameId')
        if Id not in searched_matches:
            matches.append(Id)
random.shuffle(matches)

output = open(csvfile, 'a')
while matches:
    nextMatch = matches.pop()
    searched_matches.append(nextMatch)
    count+=1
    print(count)

    r = api.get_match(nextMatch)
    while r is None or r.get('participantIdentities') is None:
        time.sleep(3)
        r = api.get_match(nextMatch)

    for player in r.get('participantIdentities'):
        accountId = player.get('player').get('currentAccountId')
        if accountId not in searched_accounts:
            accounts.append(accountId)
    random.shuffle(accounts)

    winning_champions = []
    losing_champions = []
    for player in r.get('participants'):
        win = player.get('stats').get('win')
        champion = player.get('championId')

        if win == True:
            winning_champions.append(champion)
        else:
            losing_champions.append(champion)
    game = []
    for winners in winning_champions:
        game.append(winners)
    for losers in losing_champions:
        game.append(losers)

    print(game)

    writer = csv.writer(output,lineterminator='\n')
```

```
                writer.writerow(game)
            output.close()
```

The data for this project was collected on April 12, 2018, during patch 8.7. I limited my data to certain parameters: firstly, the data gathered is only for the ranked 5v5 gamemode. This is one of the most played gamemodes, so I wanted to focus my data and ignore how champions were doing in other more obscure gamemodes. Additionally, I gathered data for the highest ranked games (ranked Diamond+) so that all of the players would be top players in their region. These players would have a relatively strong understanding of how to play the game, and would the similar skill levels would minimize the effects any external variables and maximize the influence of each individual champion on the game, or at least attempt to. It was gathered in the format of 10 columns: winning champion1, winning champion2, winning champion3, winning champion4, winning champion5, losing champion1, losing champion2, losing champion3, losing champion4, losing champion5. Below is a preview of the scraped raw data. See Matches_Raw sheet in data/league_ds.xlsx for full raw data.

```
In [1]: # scraped raw data:
        import pandas as pd
        from IPython.display import display

        raw = pd.read_excel('data/league_ds.xlsx', sheet_name='Matches_Raw')
        display(raw.head())
```

```
   champ1_win  champ2_win  champ3_win  champ4_win  champ5_win  champ1_lose  \
0          38          96          25         150          19           24
1          20         240          40          81         131            1
2         432          38         121         498          54           33
3          37          14          19         498         157           28
4          40          24         240         105          81          141

   champ2_lose  champ3_lose  champ4_lose  champ5_lose
0          432          145          136           14
1          202            2           80           13
2           81           35           14          142
3          412           91           17           67
4           14          202           37            8
```

The data collected had to be processed for the preliminary analysis. Champions were originally stored as int, under a unique championId key, so I had to pull the list of champions and their respective championIds using the LOL-STATIC-DATA_V3 method (see Champions_Raw sheet in data/league_ds.xlsx for championId legend). I decided to rename the columns to: TeamA champion1, TeamA champion2, TeamA champion3, TeamA champion4, TeamA champion5, TeamB champion1, TeamB champion2, TeamB champion3, TeamB champion4, TeamB champion5. I then added a final boolean column, TeamA_Victory, and set it as true for all the data I collected, so that there was a more definitive variable my model would try to predict.

I also decided to then expand my data by switching the champions on each team for each dataset (TeamA became TeamB, and vice versa) and set TeamA_Victory to false. This effectively doubles my dataset in size when I am ready to train my model. This also ensures that all of my data do not have the label of true for TeamA_Victory, so that a machine learning algorithm would not be able to simply predict "true" for every game and be able to accurately predict the outcome of a game. In total, I ended up with about 68,000 rows. Below is a preview of the processed data. See Matches_Processed sheet in league_ds for full processed data.

```
In [2]: processed = pd.read_excel('data/league_ds.xlsx', sheet_name='Matches_Processed')
        display(processed.head())
```

```
   TeamA_Champion1  TeamA_Champion2  TeamA_Champion3  TeamA_Champion4  \
0               38               96               25              150
1               20              240               40               81
2              432               38              121              498
3               37               14               19              498
4               40               24              240              105

   TeamA_Champion5  TeamB_Champion1  TeamB_Champion2  TeamB_Champion3  \
0               19               24              432              145
1              131                1              202                2
2               54               33               81               35
3              157               28              412               91
4               81              141               14              202

   TeamB_Champion4  TeamB_Champion5  TeamA_Victory
0              136               14           True
1               80               13           True
2               14              142           True
3               17               67           True
4               37                8           True
```

3 Data Exploration

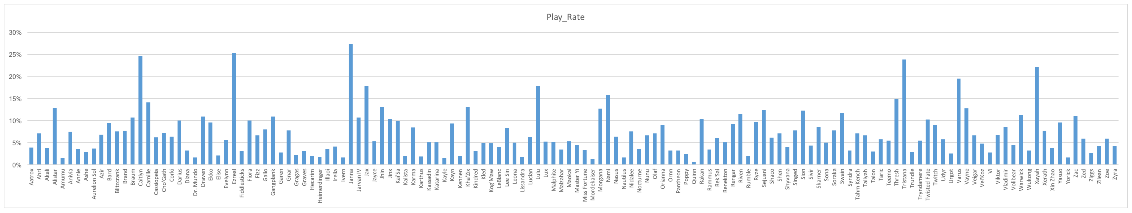
From the collected raw data, I was able to sum the total number of games each champion was played in, as well as total the number of games each champion was on the winning team. Play rate is calculated by dividing a champion's total number of games played by the total number of games gathered after preprocessing, which is 34,152 (See PlayRate_Sorted sheet in data/league_ds.xlsx). Win rate is then calculated by dividing a champion's total number of games won by total number of games played (WinRate_Sorted sheet). Thus, I procured the following charts (charts with abbreviated axes below, see full charts in Charts sheet in data/league_ds.xlsx):

$$\text{Play Rate} = \frac{\text{Number of Games Played}}{\text{Total Number of Games}}$$

$$\text{Win Rate} = \frac{\text{Number of Games Won}}{\text{Number of Games Played}}$$

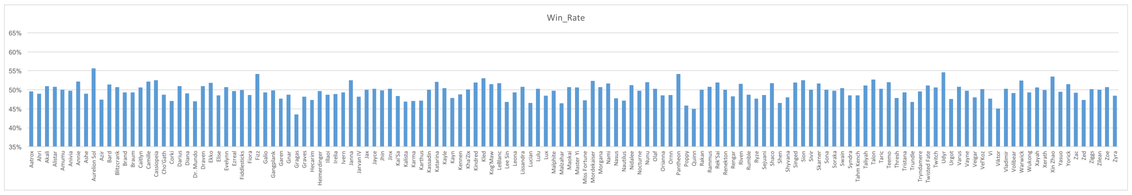
```
In [3]: from IPython.display import Image

        display(Image(filename="images/playrate.png"))
        print("Fig1. Play Rates for Champions".center(83))
```



```
Fig1. Play Rates for Champions
```

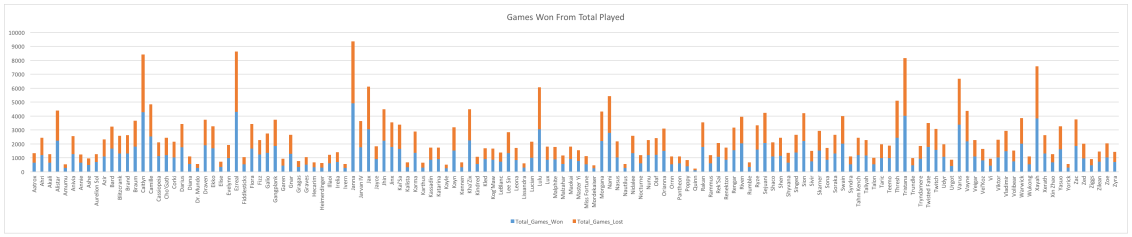
```
In [4]: display(Image(filename="images/winrate.png"))
        print("Fig2. Win Rates for Champions".center(83))
```



```
Fig2. Win Rates for Champions
```

My initial impressions are that there are no glaring outliers, and that the data I collected is valid. Most of the play rates hover around 5%, with a few more popular champions. Most of the win rates are also between 45%-55%. It was interesting to find Gragas had a terribly low win rate at 43.53%. After seeing these numbers, I thought it would be interesting to compare these total numbers:

```
In [5]: display(Image(filename="images/total.png"))
        print("Fig3. Games Won from Total Games Played".center(83))
```

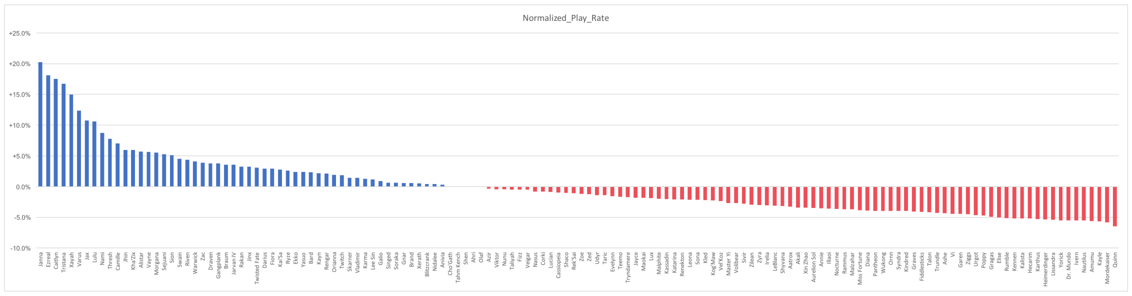


```
Fig3. Games Won from Total Games Played
```

This gave a more informed perspective on how many games each champion were actually winning and losing comparatively.

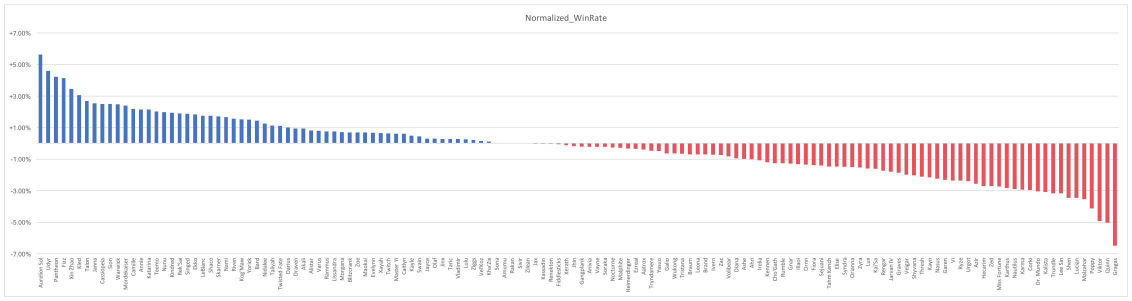
I then wanted to look at which champs were played the most and which champs won the most, so I sorted these numbers by their rank and generated the following:

```
In [6]: display(Image(filename="images/normalizedplayrate.png"))
        print("Fig4. Champions Sorted by Play Rate".center(83))
```



```
                   Fig4. Champions Sorted by Play Rate
```

```
In [30]: display(Image(filename="images/normalizedwinrate.png"))
         print("Fig5. Champions Sorted by Win Rate".center(83))
```



```
                    Fig5. Champions Sorted by Win Rate
```

For play rate, I compared each champion's play rate to the baseline probability of a champion being played if all champions were picked randomly, which is $\binom{139}{9}/\binom{140}{10} = 7.143\%$, and then calculated the difference with each champion's play rate. Thus, the above graph is interpreted as Janna is played at a rate of about +20% higher than 7.143%, or 37%, and Ahri is played at approximately the baseline of 7.143%. The chart for win rate was easier to normalize, because the baseline win rate is 50%. Interestingly enough, champions like Aurelion Sol and Gragas have a difference in winrates of more than 10%.

4 Next Steps

As I mentioned previously, the overall goal of my project is to be able to predict whether TeamA will win or lose, given the champions selected for TeamA and TeamB. While ChampionIds are integers, I will need to treat this data as categorical data. Additionally, the specific order of champions on a team does not necessarily matter, so my "problem" requires essentially two inputs: both a set of 5 categorical variables. My initial thoughts are that I am going to need to find a way to represent each team in an "orderless" manner – in other words, I should not use five independent variables for a team, but instead I should somehow be able to represent five champions as one cohesive set. For these reasons, I do not believe the best course of action is to use the data as ordinal data. ChampionId is defined by Riot, and it may not be the case that Annie (ChampionId 1) is severely different from Illaoi (420) and Rek'Sai (421).

The best approach would be to employ One Hot Encoding. Thus, I would transform the 5 columns representing TeamA's champions into 140 columns for 140 champions – the cells corresponding to the team's champions would be 1, while the others will be 0. A similar transformation would be applied to TeamB, effectively making each row 140+140 variables + 1 classification = 281 columns. I can then use logistic regression to try to find the appropriate weights for the 280 variables in attempts to classify each match as a TeamA victory or TeamB victory. With this high dimensionality, I am expecting the model's training to require substantial time and memory. I also have to be careful not to over fit my model to the data – I will be appropriately cross-validating my model across my vast data set. Additionally, it is worth noting that this specific model may not work in the future if new champions are added to the game, though a similar model can be trained if more data is collected with the new champion.

5 Preparing the Data

```
In [7]: # Data needs to be transformed from categorical to quantitative
        # One-Hot Encoding originally done in excel, saved to data/OHE_csv.csv

        data = pd.read_csv('data/OHE_csv.csv')

        display(data.head())
        #TeamA_1 - 1 if championId 1 is on team A, 0 otherwise
        #TeamB_1 - 1 if championId 1 is on team B, 0 otherwise
        #TEamA_Victory - 1 if TeamA is the winner of the game, 0 otherwise
```

```
   TeamA_1  TeamA_2  TeamA_3  TeamA_4  TeamA_5  TeamA_6  TeamA_7  TeamA_8  \
0        0        0        0        0        0        0        0        0
1        0        0        0        0        0        0        0        0
2        0        0        0        0        0        0        0        0
3        0        0        0        0        0        0        0        0
4        0        0        0        0        0        0        0        0

   TeamA_9  TeamA_10      ...        TeamB_412  TeamB_420  TeamB_421  \
0        0         0      ...                0          0          0
1        0         0      ...                0          0          0
2        0         0      ...                0          0          0
3        0         0      ...                1          0          0
```

```
4        0         0      ...                 0           0           0

   TeamB_427  TeamB_429  TeamB_432  TeamB_497  TeamB_498  TeamB_516  \
0          0          0          1          0          0          0
1          0          0          0          0          0          0
2          0          0          0          0          0          0
3          0          0          0          0          0          0
4          0          0          0          0          0          0

   TeamA_Victory
0              1
1              1
2              1
3              1
4              1

[5 rows x 281 columns]
```

```
In [8]: num_games = data.shape[0]
        #total 68302 games, 34151 games before data expansion

        num_features = data.shape[1]-1
        #280 features, 140 champions per team

        print("Total number of matches: {}".format(num_games))
        print("Number of features: {}".format(num_features))
```

```
Total number of matches: 68302
Number of features: 280
```

6 Train Data

```
In [9]: # Split Data into Training and Test Data
        from sklearn.cross_validation import train_test_split

        features = data.drop(['TeamA_Victory'],1)
        target = data['TeamA_Victory']

        x_train, x_test, y_train, y_test = train_test_split(features,
                                                             target,
                                                             test_size = 0.2,
                                                             random_state = 70,
                                                             stratify = target)
```

```
/Users/Justin/anaconda3/lib/python3.6/site-packages/sklearn/cross_validation.py:41: Deprecation
  "This module will be removed in 0.20.", DeprecationWarning)
```

```
In [10]: # Define training and prediction methods
         from time import time
         from sklearn.metrics import f1_score

         def train_classifier(clf, x_train, y_train):

             start = time()
             clf.fit(x_train, y_train)
             end = time()

             print("Time to train model: {:.4f} seconds".format(end - start))


         # returns calculated f1 score and accuracy
         def predict_labels(clf, features, target):
             #Makes predictions using a fit classifier based on F1 score.

             y_pred = clf.predict(features)

             return f1_score(target, y_pred), sum(target == y_pred) / float(len(y_pred))


         def train_predict(clf, x_train, y_train, x_test, y_test):
             #Train and predict using a classifer based on F1 score.

             # Indicate the classifier and the training set size
             print("Training a {} using a training set size of {}. . ."
                   .format(clf.__class__.__name__, len(x_train)))

             train_classifier(clf, x_train, y_train)

             f1, acc = predict_labels(clf, x_train, y_train)
             print("Training set F1 score and accuracy score: {:.4f} , {:.4f}."
                   .format(f1 , acc))

             f1, acc = predict_labels(clf, x_test, y_test)
             print("Test set F1 score and accuracy score: {:.4f} , {:.4f}."
                   .format(f1 , acc))


In [11]: # First attempt Logistic Regression, Naive Bayes, and Random Forest Algorithms

         from sklearn.linear_model import LogisticRegression
         from sklearn.naive_bayes import GaussianNB
         import xgboost as xgb

         clf_A = LogisticRegression(random_state = 23)
         clf_B = GaussianNB()
```

```
        clf_C = xgb.XGBClassifier(seed = 63)

        train_predict(clf_A, x_train, y_train, x_test, y_test)
        print(" ")

        train_predict(clf_B, x_train, y_train, x_test, y_test)
        print(" ")

        train_predict(clf_C, x_train, y_train, x_test, y_test)
        print(" ")
```

```
Training a LogisticRegression using a training set size of 54641. . .
Time to train model: 1.4196 seconds
Training set F1 score and accuracy score: 0.5461 , 0.5452.
Test set F1 score and accuracy score: 0.5289 , 0.5311.

Training a GaussianNB using a training set size of 54641. . .
Time to train model: 0.4038 seconds
Training set F1 score and accuracy score: 0.5367 , 0.5364.
Test set F1 score and accuracy score: 0.5256 , 0.5238.

Training a XGBClassifier using a training set size of 54641. . .
Time to train model: 43.0457 seconds
Training set F1 score and accuracy score: 0.5659 , 0.5654.
Test set F1 score and accuracy score: 0.5276 , 0.5300.
```

```
/Users/Justin/anaconda3/lib/python3.6/site-packages/sklearn/preprocessing/label.py:151: Depreca
  if diff:
/Users/Justin/anaconda3/lib/python3.6/site-packages/sklearn/preprocessing/label.py:151: Depreca
  if diff:
```

Unfortunately, all three algorithms were only 2% to 3% better than 50%, which is only marginally better than a blind guessing approach. These are early signs that data on champions picked alone are not enough to accurately predict the outcome of games. There are many other varibles not considered in this analysis, and champions picked for each team may only have a tiny influence on the outcome each game.

I decide to further tune the Logistic Regression model as it produced the best results in a rather short amount of time.

7 Tuning Logistic Regression

```
In [12]: # Fine and select best Cost parameter

        from sklearn.grid_search import GridSearchCV
        from sklearn.metrics import make_scorer
        # Test various cost parameters for highest accuracy
        C_param = {'C': [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 50, 100],
                   }

        # Choose classifier A, Logistic Regression
        f1_scorer = make_scorer(f1_score)

        grid_obj = GridSearchCV(clf_A,
                                scoring=f1_scorer,
                                param_grid=C_param,
                                cv=5)
        grid_results = grid_obj.fit(x_train,y_train)

        # Get the best estimator
        clf = grid_results.best_estimator_
        print(clf)

        # Report the final F1 score for training and testing after parameter tuning
        f1, acc = predict_labels(clf, x_train, y_train)
        print("F1 score and accuracy score for training set: {:.4f} , {:.4f}."
              .format(f1 , acc))

        f1, acc = predict_labels(clf, x_test, y_test)
        print("F1 score and accuracy score for test set: {:.4f} , {:.4f}."
              .format(f1 , acc))
```

```
/Users/Justin/anaconda3/lib/python3.6/site-packages/sklearn/grid_search.py:42: DeprecationWarn
  DeprecationWarning)


LogisticRegression(C=0.01, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=23, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
F1 score and accuracy score for training set: 0.5453 , 0.5450.
F1 score and accuracy score for test set: 0.5255 , 0.5284.
```

We use the GridSearchCV library to find the best parameter for our LogisticRegression using the F1 score over 5-fold validation. GridSearchCV returns the best parameter of C=0.01, but it seems our caluclated F1 value actually decreases for our original training and test set. Overall performance of the model, however, should have slightly improved as other training/test splits should have improved. Below, I calculate the confusion matrix for our slightly improved model.

```
In [13]: # Construct Confusion Matrix


        size=2
        matrix = [[0 for x in range(size)] for y in range(size)]
```

```
    table = [[0 for x in range(size)] for y in range(size)]

    test_predictions = clf.predict(x_test)

    count=0
    for each in y_test:
        actual = each
        prediction = test_predictions[count]

        matrix[actual][prediction] += 1

        count += 1

    table[0][0] = matrix[1][1]
    table[0][1] = matrix[1][0]
    table[1][0] = matrix[0][1]
    table[1][1] = matrix[0][0]

    print("Confusion Matrix:")
    table_row_header = ['Predicted Win', 'Predicted Loss']
    table_column_header = ['Actual Win', 'Actual Loss']
    row_format ="{:>15}" * (len(table_row_header) + 1)
    print(row_format.format("", *table_row_header))
    for team, row in zip(table_column_header, table):
        print(row_format.format(team, *row))

    accuracy = (matrix[1][1] + matrix[0][0]) / len(test_predictions)
    precision = (matrix[1][1]) / (matrix[1][1] + matrix[0][1])
    recall = (matrix[1][1]) / (matrix[1][1] + matrix[1][0])

    print("\n")
    print("Accuracy: {:.5f}".format(accuracy))
    print("Precision: {:.5f}".format(precision))
    print("Recall: {:.5f}".format(recall))
```

```
Confusion Matrix:
                 Predicted Win Predicted Loss
     Actual Win           3567           3263
    Actual Loss           3179           3652


Accuracy: 0.52844
Precision: 0.52876
Recall: 0.52225
```

8 Application

The ideal use of this model would be its application on real-time games. Currently, losses in the ranked game mode incur a deduction in a player's League Points (LP), which is a measure of one's rating. Players are able to exit games after champions are selected but before the game actually starts, resulting in the disbandment of the game as well as a lesser deducation in LP than if the player were to actually complete and lose the game. This action is called "dodging". So, if a player knows they will lose a game, it is strategic for them to dodge and receive the lesser penalty.

A real-use application of this model would then be able to predict the outcome of a game while a user is waiting for the game to start. A user should be able to input the champions on both teams and receive feedback.

```
In [14]: # Applies machine learning model to input data
         # Outputs its prediction
         def predict_game(clf, Team_A, Team_B):
             champion_list = pd.read_csv('data/champion_list.csv')
             champion_list.head()

             champions = dict(zip(list(champion_list['Champion_Name'])
                                  ,list(champion_list['Champion_Id'])))

             input_header = list(data)
             input_header.remove('TeamA_Victory')


             game = {}
             for x in range(len(Team_A)):
                 key1 = "TeamA_"+str(champions[Team_A[x]])
                 key2 = "TeamB_"+str(champions[Team_B[x]])
                 game[key1] = 1
                 game[key2] = 1

             app_input = []
             for each in input_header:
                 if each not in game:
                     app_input.append(0)
                 else:
                     app_input.append(1)
             app_input = [tuple(app_input)]

             predict_input = pd.DataFrame.from_records(app_input, columns=input_header)

             outcome = clf.predict(predict_input)

             if (outcome[0]==1):
                 print("Predicted outcome: Team_A Victory")
             else:
                 print("Predicted outcome: Team_A Defeat")
```

```
            #return outcome

In [15]: # Input Team A and Team B

         # Three examples of recently played games

         # Final game in the 2018 Spring Split Finals -
         # Professional Players in a Tournament
         # Team_A (Win) - Maokai, Skarner, Azir, Jhin, Braum
         # Team_B (Lose) - Gnar, Sejuani, Ryze, Caitlyn, Thresh

         Team_A = ['Maokai','Skarner','Azir','Jhin','Braum']
         Team_B = ['Gnar','Sejuani','Ryze','Caitlyn','Thresh']
         predict_game(clf, Team_A, Team_B)


         #Competitive Ranked 5v5 Solo Queue
         # Team_A (Lose) - Malphite, LeeSin, Fizz, Jhin, Rakan
         # Team_B (Win) - Darius, Shaco, Diana, Draven, Karma

         Team_A = ['Malphite', 'Lee Sin', 'Fizz', 'Jhin', 'Rakan']
         Team_B = ['Darius', 'Shaco', 'Diana', 'Draven', 'Karma']
         predict_game(clf, Team_A, Team_B)

         # Team_A Lose
         # TTeam_B Win
         Team_A = ['LeBlanc', 'Leona', 'Mordekaiser','Rek\'Sai','Kai\'Sa']
         Team_B = ['Jayce', 'Caitlyn', 'Graves', 'Alistar', 'Karma']
         predict_game(clf, Team_A, Team_B)

Predicted outcome: Team_A Victory
Predicted outcome: Team_A Defeat
Predicted outcome: Team_A Victory
```

In this small sample size of recent games, the model was successful in predicting the outcome of two out of three games, including the final game in the Finals match of the 2018 Spring Split.

9 Related Works

K. A. Willoughby, Winning Games in Canadian Football: A Logistic Regression Analysis, The College Mathematics Journal 33 (2002), #3.

This paper details a very standard application of Logistic Regression to predict one of two outcomes: Team A wins or Team B wins, similar to problem my project tackles. Willoughby's approach, however, features quantitative data that are relative to a persistent professional team, like yardage and number of interceptions in previous games. For my project, we cannot calculate similar metrics because players do not play with the same players on the same team as they do in football, so the difference in the performance of our final models reflect greatly on the significance each respective's training data. In sports like football, many more variables between games are held constant. Each player plays on the same team, and each player more or less plays in the same position. The largest variance between the outcomes of these games can be accounted for by how the players on different teams play against each other. It helps the performance of models for these types of problems that this difference can directly be translated into one team's victory over the other.

After coming to my results, I looked to other projects to see if more effective methods were found. I found a project by Jihan Yan (https://github.com/arilato/ranked_prediction) that was moderately more successful than mine. Yan was able to achieve 60% accuracy as compared to my model's 53%, but his model considered more than just champions selected. His model not only directly analyzed each champion's matchup measured by each champion's winrate versus its specific opponent, he also considered each player's personal stats. For my project, I wanted to avoid any player-specific data and only examine raw aspects of the game, so it is interesting to see that even with consideration of player skill, the best model can only accurately predict 60% of games.

More about his project can be found here: https://hackernoon.com/league-of-legends-predicting-wins-in-champion-select-with-machine-learning-6496523a7ea7.

DeepLeague (https://github.com/farzaa/DeepLeague) is an interesting project that extends beyond the reach of my project and looks into how games are played during them. In DeepLeague, computer vision is used to precisely track where each of the ten players are at all times. Each league game averages between 20-30 minutes, so there is an enormous amount of data available that can be collected pertaining to how successful teams exactly, down to the very specific pixel. If players are interested in learning how to actually play games well, instead of simply picking the correct champions to play, more advanced projects will apply these technologies to analyze winning playstyles and specific play movements. I imagine these types of projects to be extremely complex, yet solving them would also be very rewarding. Projects like these remind me of projects like AlphaGo. Go is much simpler than League of Legends in terms of the complexity space that contains each possible set of moves for each player, and it still was not until 2015 that computers were able to "solve" Go. I do not know if a game like League will ever be "solvable", but training a computer to be good at the game will definitely need to be able to analyze the movements of each player as done in DeepLeague.

This project goes more into detail here: https://medium.com/@farzatv/deepleague-leveraging-computer-vision-and-deep-learning-on-the-league-of-legends-mini-map-giving-d275fd17c4e0.

10 Conclusion

Although I was not able to train a model that could reliably predict the outcome of games based off of champions alone, the overall results of the project provide valubable insight to the intricacies of the game. From my results, I come to the conclusion that the pure champions that make up each team have a smaller impact on each game than I had previously thought. Whereas there is a significant emphasis on the champion select phase of League of Legends, the performance of the champions picked are largely determined by the individual player that picks them. While there are stronger and weaker champions as seen by the disparity in win rates, it is ultimately how these champions are utilized in-game that influence the outcome of games, rather than the raw strengths of each champion.

Despite the shortcomings of this project, I do still believe that it is possible to train a model that is reliable to a much higher degree. Future adaptations of this project will need to consider player-specific data, such as player rank, champion mastery, and other performance-specific data that provides more information as to how the player will execute the strengths and weakenesses of their champion. If I am to take this approach, I may need to adjust the goal of the project as the specific players on the enemy team are not revealed before the game starts. Thus, the future application will either need to only consider one team's players and employ more advance probabilistic models, or attempt to predict the outcome after the game starts and this information becomes available. While the former would be more useful, I imagine that design to be more difficult to accurately model.

Furthermore, despite attempting to simply the problem as possible by considering only champions selected, due to the compelxity of the game, the data still ended up being high-dimensional with 280 features. I originally wanted to train a SVM, but even a linear SVM required too much time and computing power to train. In the future, I will need to employ dimension-reduction techniques like Principal Component Analysis or t-SNE. It also helps to use quantitative data. Despite the data for this project was extremely large (241 x 64,000), much of that data was empty 0s. Only 10 cells in each row were relevant to predicting the outcome that game, so dimension reduction techniques or more efficient data representations definitely could have been used.

11 Acknowledgments

External libraries used:

pandas: (https://pandas.pydata.org/) open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

sklearn: (http://scikit-learn.org/stable/) Split data into Training/Test sets, Logistic Regression, Gaussian Naive Bayes model, GridSearchCV, F1 score,

xgboost: (https://github.com/dmlc/xgboost) an open-source software library which provides the gradient boosting Python, Random Forest Model

Riot API tutorial: (https://www.youtube.com/watch?v=0NycEiHOeX8) I used this tutorial to get me started with the Riot API. The backbone of my request method is modeled after what is done in this tutorial. However, I have implemented my own methods of crawling and tracking which matches have already been recorded for my data collection process.

Ratelimit 1.4.1 (https://pypi.python.org/pypi/ratelimit) I used this external library to limit my requests to comply with Riot's rate limit of 100 requests per 120 seconds.

Predicting the Winning Football Team:

(https://github.com/llSourcell/Predicting_Winning_Teams/blob/master/Prediction.ipynb) I organized my project similarly to this one, especially in terms of training the model. This project had a similar goal to my project, however this one used originally quantitative data.