Justin Tsao

# Data Science: League of Legends



Table of Contents

## I. INRTODUCTION

### Problem to Solve

This project will be focus on Esports, specifically the game League of Legends. League of Legends is an online multi-player game where 10 players compete in 5v5 matches. The problem this project aims to solve is predicting which team will win based on decisions a player can make before a match starts. This includes champion picks, summoner spells, and runes. Currently, there are a lot of tools that use post-match data to provide match statistics and determine best combinations of items for a given champion or calculate win rates based on past match results. The goal is to take a data science approach and use pre-match data to turn this into a classification problem. Before the actual prediction and machine learning aspects of this project, this project will explore some of other aspect of data science such as wrangling the data, using an API to extract data, and also exploring the data.

### Client

The client would be the developer of League of Legends – Riot Games, professional teams, and competitive ranked players of the game. Riot Games invests a lot of time and money into broadcasting/shout casting big events like World Championship tournaments. Today, it rivals the professionalism and quality of events like the NBA. During a broadcast, professional teams make their picks for team composition. Shout casters make this a huge talking point to their audience. Adding a new talking point could add more depth and engagement for the viewers. The match prediction can be this talking point.

For professional teams and ranked players, the ability to predict wins can be built into some form of an app which an individual can input player choices for their team and the opposing team to see the results of a certain team composition versus another. This could lead to more insight on strategies, and help to win games in ranked matches. A different model will most likely be needed for each player level tier and region. Another model will most likely be needed for professional level plays as strategy and skill level play a huge role in predicting match outcomes.

**Data**

The data I will be using will come from Riot Games API found at https://developer.riotgames.com/. I will use python coded to pull ranked (competitive) match data from the API based on popular players, streamers, and professional players. All match data will be from high ranked games where the meta game is usually defined and imitated at lower ranks. The data includes game stats like match duration, game creation date, characters picks and bans. The data also includes about 100 game stats each for the 10 players.

**Problem Solving**

Based on looking at the data, we can get the games of every player, how many matches a player has played, when the match was created, and every game stat at the end of the game. This information will be good for the initial data analysis. However, I am more interested in extracting the data which players have direct control over. The idea is build a model to predict games based on high tier matches. For this project, the matches will come from 'Diamond' Tier and above for ranked matches in North America.

At this level of play, the players are assumed to have in depth game play knowledge and are highly skilled. With this high level of gameplay the features of the model should be more relevant to predict the outcome of a match. My intuition is that team composition and choice of summoner spells/runes will give good predictive insight given a set level of skill. A model to predict win outcomes may prove difficult at low levels of play due to the vast difference in player skills, and casual players playing for fun. If the model proves to be successful at high levels of play, I assume that the model could have similar success when using data from professional matches as well.

## II. DATA WRANGLING

### Acquire Data Source

Code for the data wrangling can be found below:
https://github.com/jltsao88/Capstone_Project_1/blob/master/Data_Wrangling_API.ipynb

This project relies on data taken from competitive ranked matches. This was done directly through Riot Games API which can be found at https://developer.riotgames.com/. Anyone is welcome to create a free account and receive a API key.

The first thing to do is create an account with Riot Games, which I already had from playing the game. The next step was to acquire an API key and study the API reference documentation to get exactly what I was looking for. Take note that there is a rate limit of 20 requests every 1 second and 100 requests every 2 minutes. The Riot Games API contains many API methods which return data in JSON. The methods I

used were SUMMONER-V4 and MATCH-V4 which I will describe below. Keep in mind, that if trying to follow these steps, there is possibility that API methods could be updated or deprecated.

The goal was to obtain at least 100,000 matches worth of data and store the data as CSV files to be easily turned into Pandas Data Frames. Each row of the Data Frame would be a single match and columns would each represent statistics and key information of the game such as stats of each individual player and game id. To get matches of individual games would require getting an encrypted account id based on in game player names and the individual match id of every game.

The initial process to get at this data goes as follows. First, I needed to use the API method SUMMONER-V4 to get an encrypted account. I wrote a function called get_acct_id() which would take 3 arguments, the players in game name aka summoner name, region where the game is player, in this case 'na1', and the API key itself. I used the Python ***requests*** library to return the JSON in every case. In every method I would check the response code. Code 200 meant a successful request. If the response code returned was 429 that meant I was over the limit and would have Python sleep for 2 minutes before attempting any more calls to the API. A successful call to the API returned the encrypted player ID

Next, I needed the match ids for 100,000 games or more. I searched well known professional players and popular streamers, all who play the game at a high and competitive level. All the players are considered to know the game inside and out and would lead to the most promising match data. The API MATCH-V4 method was used to

return a list of all ranked games played in the 2018 season for each summoner name using the function get_match_history().

Next, I used the MATCH-V4 method again in a function named get_match_stats() to return the data of a match in JSON.

The main function I created to get 100,000+ match ids is called get_many_matches(). This function could take a list of in-game player names and either return a list of match ids as a new pickled file or add on to an existing pickled file. Due to the API rate limit it would currently take over 50+ hours to write all the data into CSV files. I decided to split the ids into 10 files of approximately 10000 match ids in each file. This was partly due to my fear of encountering bugs and being able to monitor printed debugging outputs. I am glad I did because I caught many bugs and silent fails which led to me refining my code multiple times over and add new exception handling. I was eventually able to pull over 146,000 matches.

The final step was to create a function called create_master_data() that would both clean/wrangle the data and write the data into a CSV file.

**Data Wrangling and Cleaning**

The way I cleaned and wrangled the data was first converting the JSON to a data frame. I saw the size of the data and decided to break them down into many CSV files. As mentioned above, create_master_data() would clean and wrangle the data. Many helper functions were created to achieve this. But first, I looked at the output of the JSON. What I noticed was the champions, items, and spells were returned as id numbers. I now needed to find a mapping of the id numbers. I found this data in JSON

format by asking around on Riot Games API Discord channel found here. I was directed to other APIs which I could extract the mapping of ids to items, champs, and spells. I created a dictionary in Python for all those ids and wrote functions which would replace those id numbers. All the cleaning functions are add_player_name(), add_champ_pick(), add_sumpell_pick(), add_stats(), add_team(), and add_team_stats(). Another function create_row() would use all those cleaning functions and create a dictionary which would later be used to create rows for the function create_master_data().

The API never returned missing values which was great. However, the game is updated on almost a bi-weekly basis meaning items are removed and changed. I ran into the case that id numbers for items and champions were missing due to different match ids occurring in different patches or updates to the game. In the case that the ids could not be matched, I allowed the id numbers to stay in place rather than remove them or create null values. I originally ran into KeyError exceptions during the initial runs and did more defensive programming to my functions.

Each player had close to 100 in game stats each. Further string manipulation was needed as the stats for each player were values in key-value pairs. Therefore, I added a ‘p’ + ‘number’ before any stat name to indicate which stat belonged to each player. For example, p3 would indicate player 3. Other names for certain columns were cleaned in this fashion.

In the context of the game, there are certain outliers for stats like kills. But this is due to the nature of the game. A player can ‘snowball’ if getting first kills very early in the game. This creates a lead which other players can not keep up with. This happens

from time to time. It is more common in low rank matches, but does happen in the competitive scene as well. These outliers do not need to be removed at all. They are often the high lights of a players skill in high ranked matches. In a 5v5 game with competent players, come backs can come at any instance and even an extremely high kill to death ratio is sometimes meaningless without looking at other stats like amount of 'vision', which can be calculated using how many vision wards are placed.

Overall I believe the data set is very reliable as I picked players in the highest ranked matches at the level of Master, Challenger, Grandmaster, and diamond. Below those tiers are platinum and gold rank. And, the lowest being silver and bronze ranked. Of course there will be differences in the data if matches were gathered form different tiers, but I was interested in the level of play of professionals and those who stream the game online for hours at a time for a living. These are the matches that are watched and analyzed by the rest of the player base and it is usually the top ranked matches which determine the meta play of the game at any given time.

## III. EXPLORATORY DATA ANALYSIS (EDA)

Code for EDA can be found below:
https://github.com/jltsao88/Capstone_Project_1/blob/master/EDA.ipynb

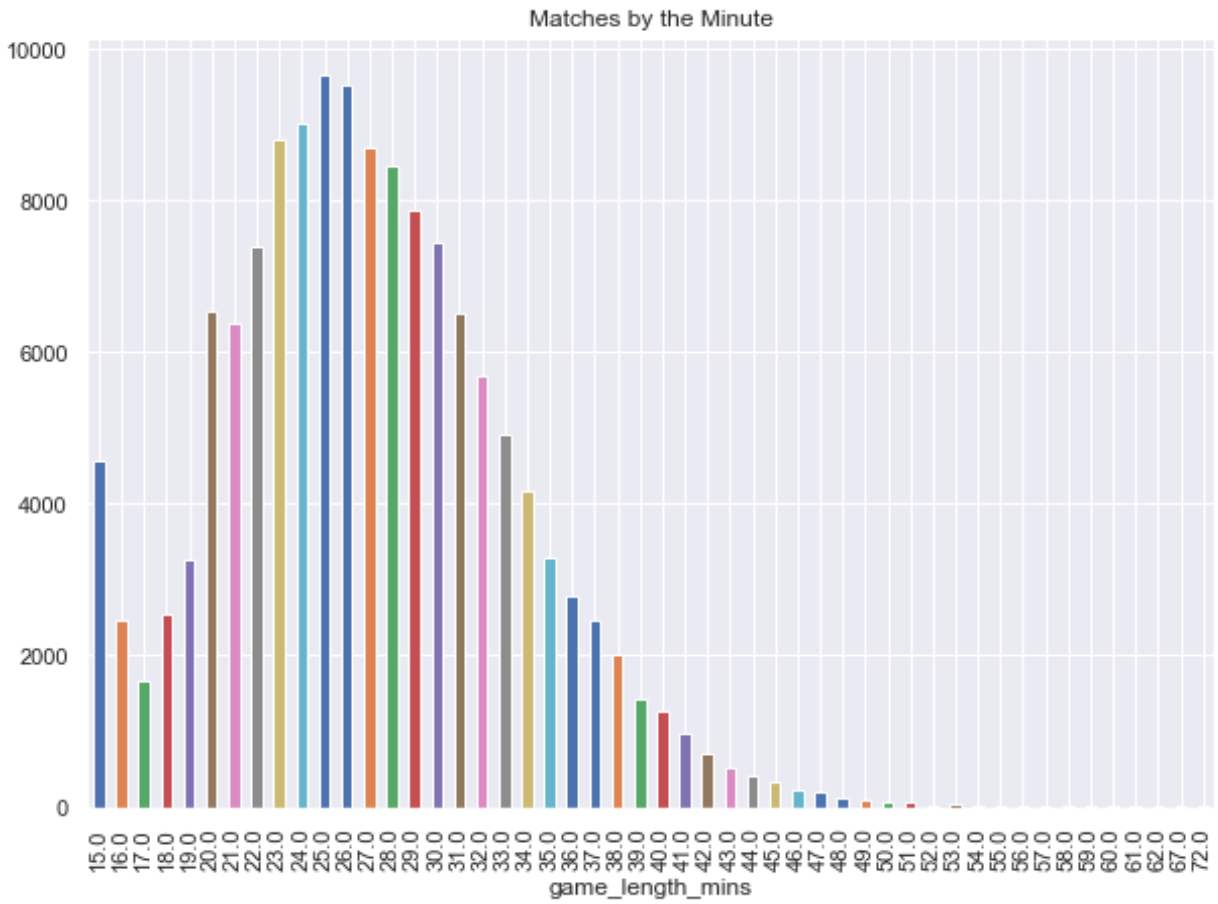
EDA is an important step in data science. For me, this was the most enjoyable part of the project. I got to explore the data in depth visualize many of statistics I was interested in. Though a lot of the analysis done for the EDA is data coming from post-match data, it did help me build some intuition for the final results of the machine learning part of this project, which is detailed later.

If you are following the code for the EDA, you will see there was still some cleaning that had to be done. Columns that had data from deprecated modes were removed. In 2018, a new rune system was implemented by the developers. The rune ids had to be mapped to the rune names. I also decided to create my own features, or new columns of data. These new features were the champion's primary and secondary roles such as tank, fighter, etc. I was interested in seeing the affects of champion roles on game play.

**Visual EDA**

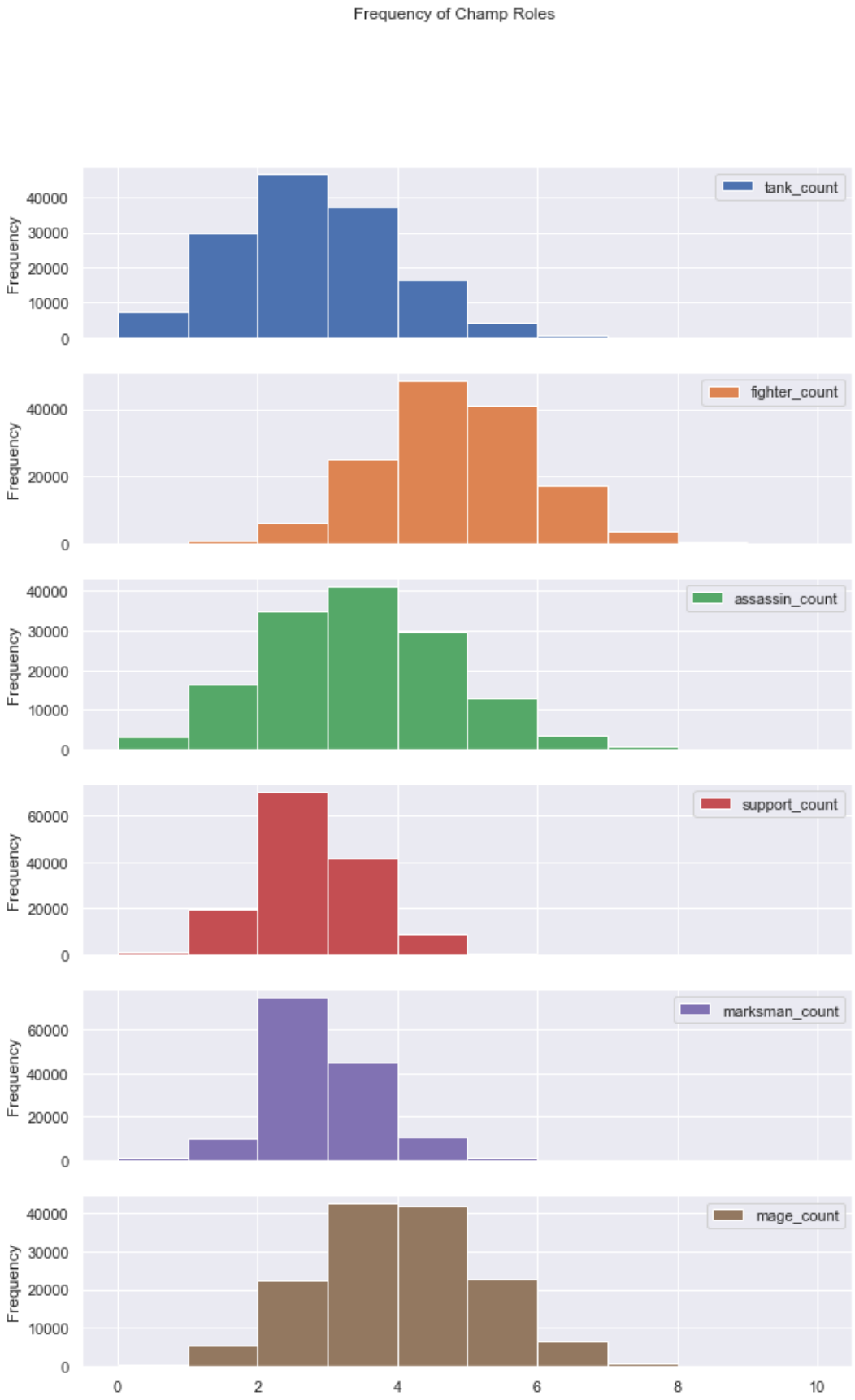
Initially, I was interested to see if the composition of champion archetypes had any affect on the length on matches. To get better results from the data, matches under 15 minutes were removed from the data set. Matches under 15 minutes meant early surrenders, the possibility that certain players did not connect to games, or players going AFK. This removed 3931 rows of data (matches), and we are left with 142,773 matches.

To help in visualizing the data for match times, game times were rounded to whole minutes.



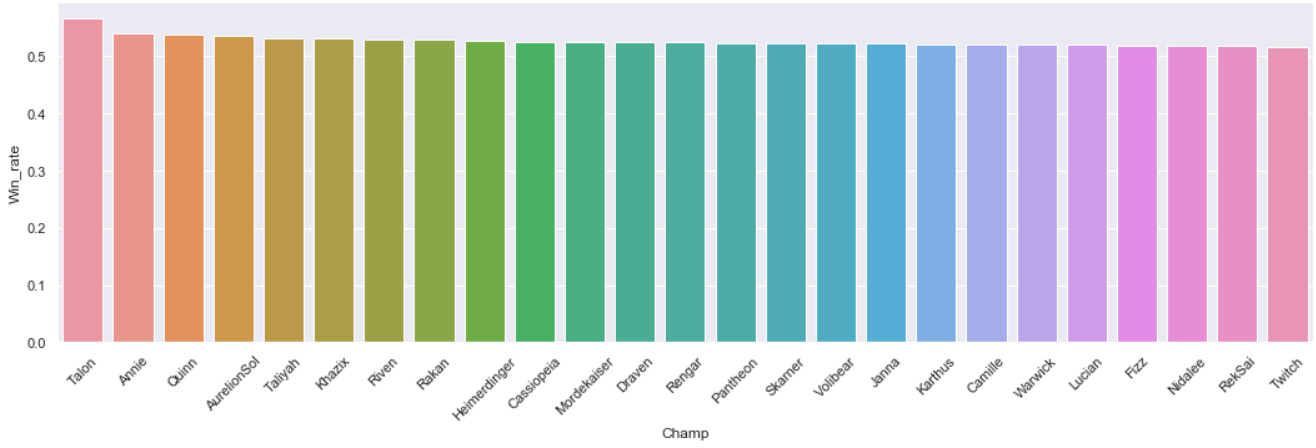

```
count    142773.000000
mean         27.471626
std           6.372879
min          15.016667
25%          23.100000
50%          26.950000
75%          31.400000
max          72.966667
Name: game_length_mins, dtype: float64
```

If looking at only the left side of the plot from 15 to 26 minutes, you will notice it looks less normal. This is most likely due to players decisions about surrendering a game early. 20 minutes is the normal surrender time, and after that the plot looks more normal for the right side.


Frequency of Champ Roles
tank_count
fighter_count
assassin_count
support_count
marksman_count
mage_count
Frequency
0
10000
20000
30000
40000
60000
0
2
4
6
8
10

```
supports: 325173, 12.81%
marksmen: 343258, 13.52%
mage: 504429, 19.87%
tank: 326048, 12.84%
assassin: 419309, 16.52%
fighter: 619549, 24.41%
The count and percentage of the champions played from the data.
```

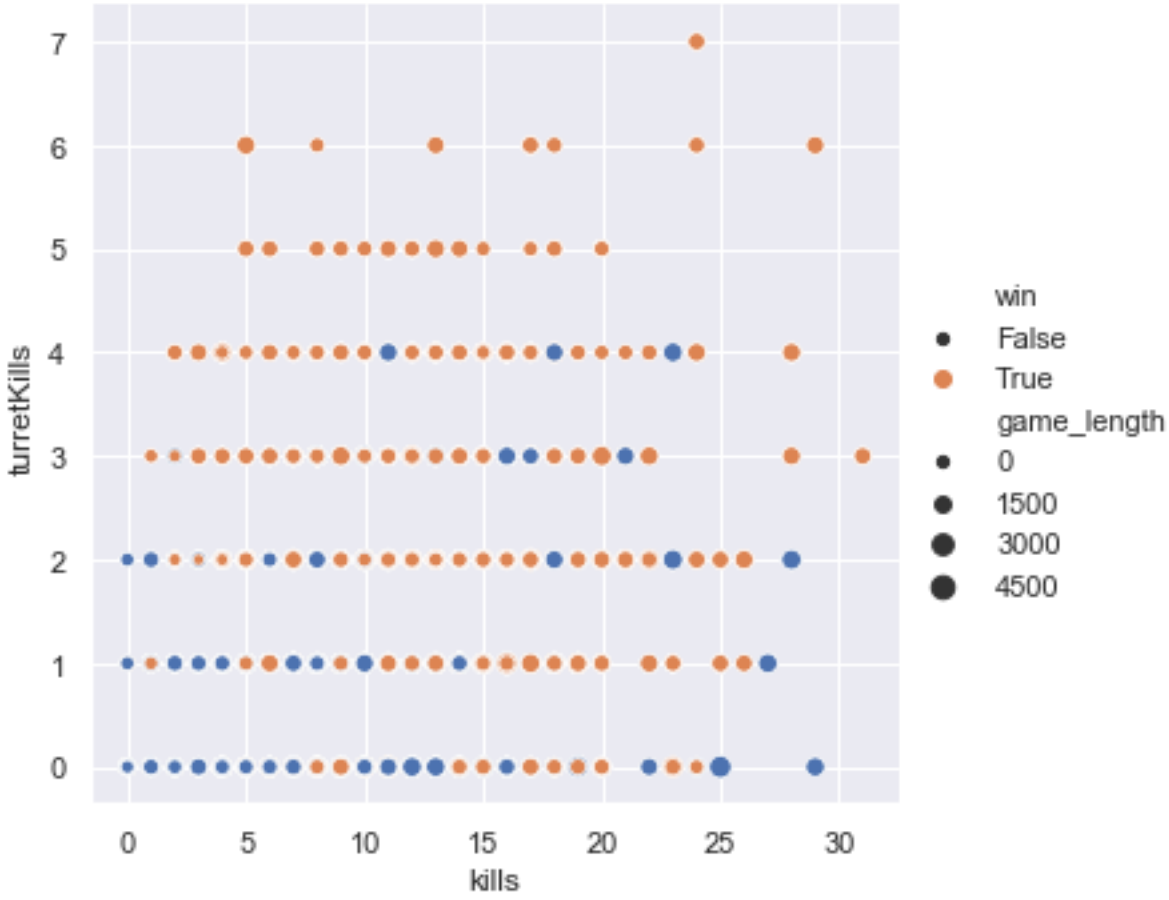
The plot above are histograms of the champion types picked in games. If you know the meta, the plot describes the champion composition of most teams pretty well. Supports and marksmen plots mirror each other pretty closely as a support and marksmen combination is on most standard teams. You can also see that mages and fighters have the highest picks. This makes sense since fighters and mages and very well rounded and versatile in play style. Please check out the GitHub link to the EDA code if you want to see how having a certain number of champion roles in a game affect average game lengths.



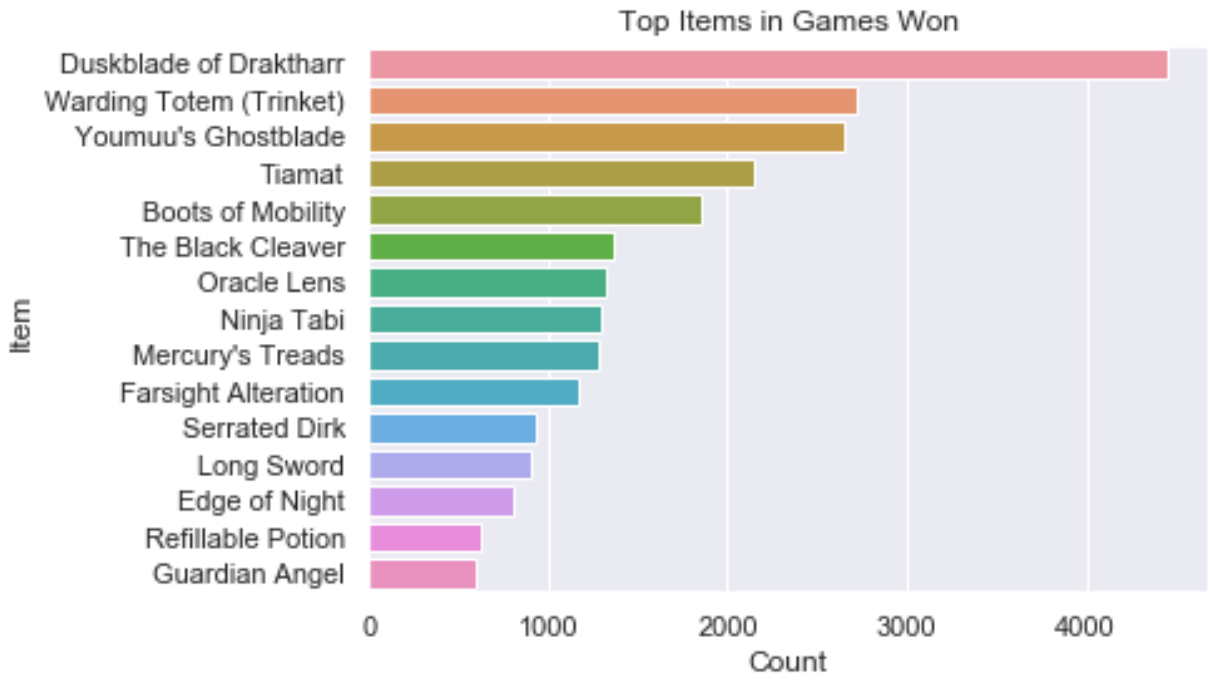

The plot above shows the 25 champions with the highest win rate. They all have above a 50% win rate. However, Talon stand out visually having a win rate of 56.54%. Exploring the data about Talon in depth, we find out he has two champion roles, a

primary role of an ‘assassin. and secondary role as a ‘fighter’. Out of the 146704 games in the data, Talon was played in 6.35% of the matches.


The plots graph total damage dealt to other players versus damage dealt to map objectives. The plot on the represents matches in which Talon got the first kill of the game which is rewarded with extra gold. Talon gets first blood in 21.93% of the games he is in. We see that Talon players get first blood more often than not. Even though Talon is an assassin who excels at killing champions 1v1, the plots show a trend that Talon wins more games by focusing on map objectives. This makes sense as League of Legends is a team based game.




The plot above shoes Talon's champion kills versus turret take downs in a game. Turrets need to be taken down to advance into the enemy's base. Though it may be a little difficult to see, Talon's win condition drops as the game length drags on, even with a high number of champion kills. But the main take away is that Talon wins more games with more turret take downs.

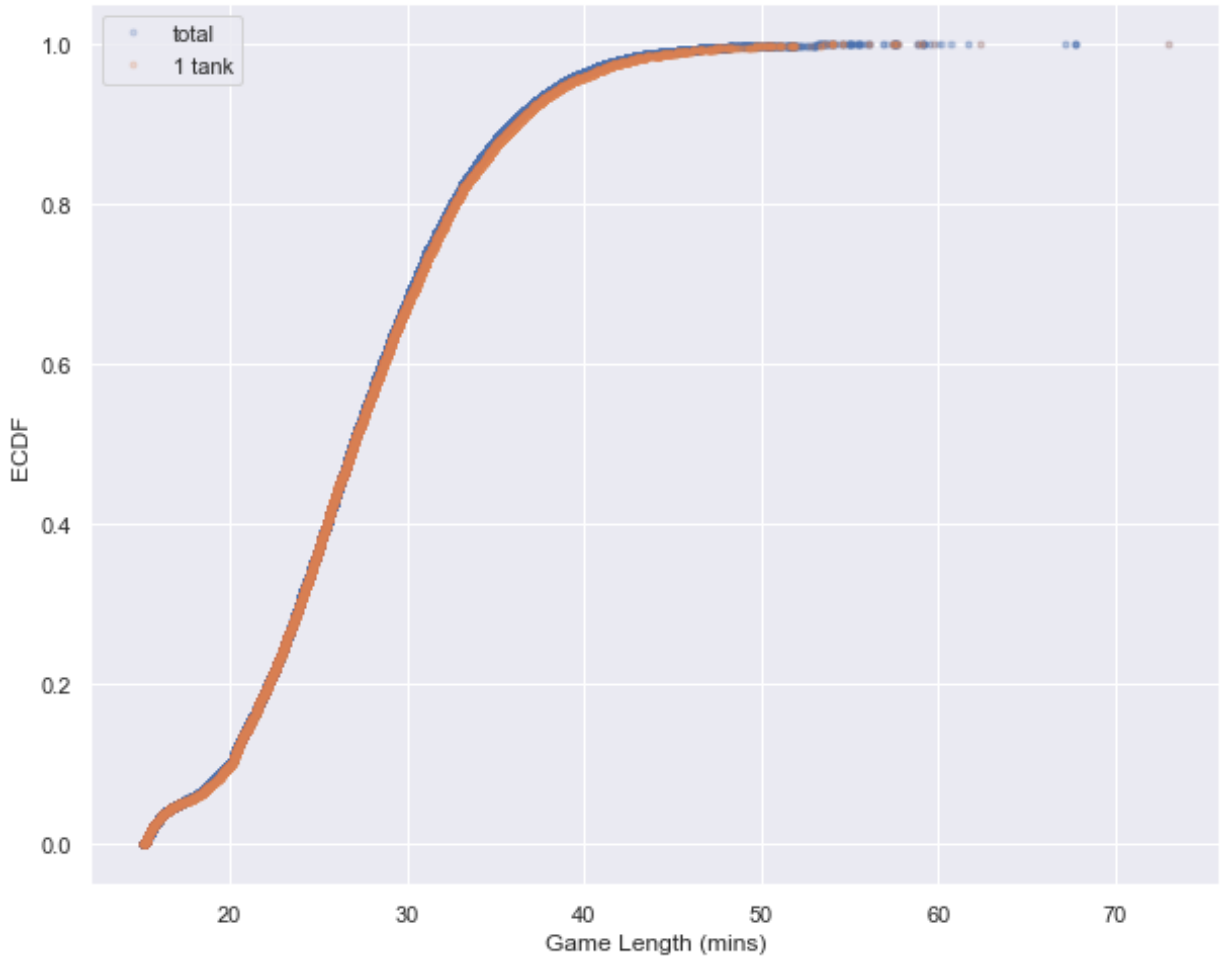



The plot above is pretty self-explanatory. Duskblade is clearly a strong item choice on Talon.


This plot is less interpretable in regards to Talon's win rate. Wards placed represent Talon's vision score in creating vision on the map. Wards killed is Talon score in denying vision to the enemy.

**Inferential Statistics**

Again, in the beginning of the project I was very interested in how number of champion roles in a game affected game play time. One would assume longer game times with more tanks and faster game times with more assassins. I wanted to test the hypothesis that there is no difference between average game lengths regardless of champion roles picked in a game. The alternative hypothesis would be that there is a difference in game time based on champion picks.



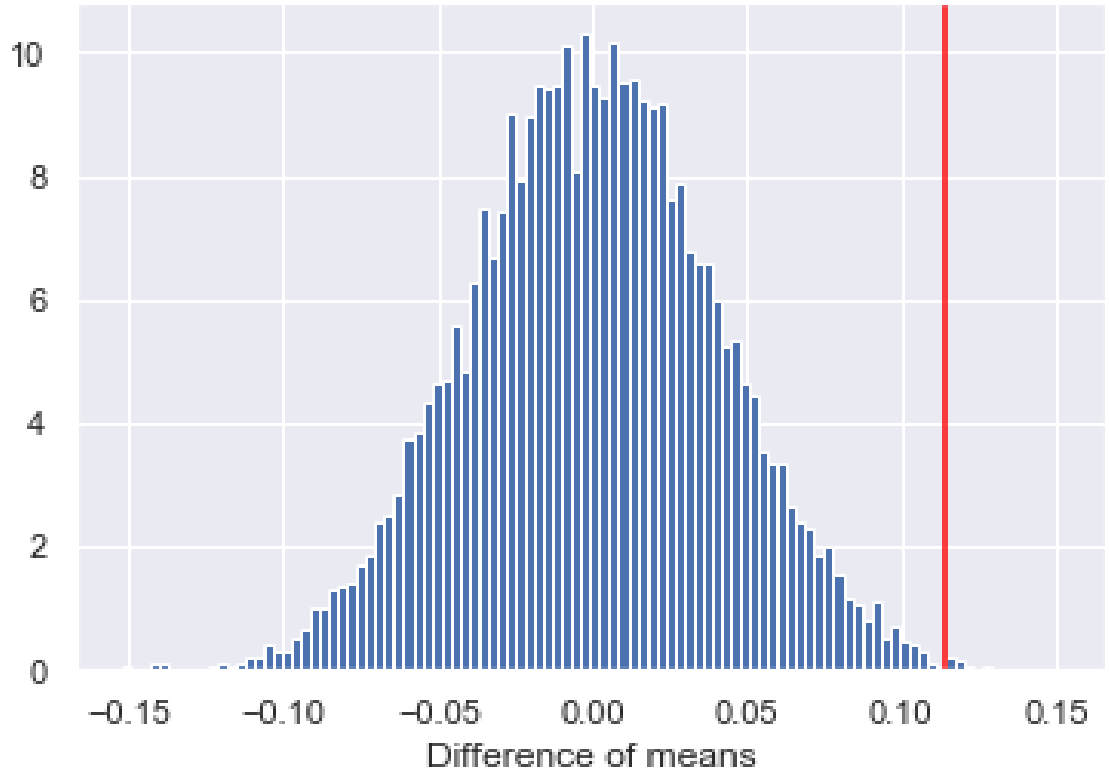

I used Python code to generate an Empirical Cumulative Distribution Function (ECDF) which graphically displays the bootstrapped samples for match duration in games with 1 tank. As an example, I used bootstrapping to obtain bootstrapped samples of the difference of means of game lengths between games with one tank and total games:

Null Hypothesis: The difference of means for game times is 0 for games with 1 tank

Alternate Hypothesis: The difference in means is greater than 0 for games with 1 tank

Significance level: 0.01

After doing some calculation, I found the difference of mean game times between total games and games with one tank to be 0.1144 minutes. I then calculated the p-value to of using Python to find the number of games in 10000 bootstrapped samples that had a difference of means greater than 0.1144 minutes. The p-value ended up being 0.0021 which was below the significance value. This means we can reject the null hypothesis that the difference in means is the same.




Although there is statistical significance there is not much practical significance in the outcome. Most players will not even notice the average game times varying by that much. I repeated this process using Python and the frequentist approach for varying number of champion types in the game. The final result was that there was no practical

significance in any of the tests as the game times did not vary much based on champion composition. My ultimate conclusion taken directly from the GitHub link:

"Although there are results that are statistically significant, none of the results have much practical significance. The game times barely vary more than 1 minute based on champion role composition. This may be the result of player knowledge at the very highest level of the game. They know what champions counter and are good against other champions. For those unfamiliar with the game, there is a draft phase before the start of matches. Each team can banned 5 champions for a total of up to 10 champions banned. players also pick champions in turns, allowing counter picking and formation of strategic team compositions. These findings could also suggest that Riot Games does a very good job at balancing champion power in overall game play. Riot Games also release constant patches to adjust power levels if certain champions are out of control.

For future analysis, it may be more interesting to see the same test done in lower level of play with more casual players. At the highest level of play, you can exact more normalized results."

Overall, the EDA for the data was interesting and insightful. Much of the data from the API comes from post-match statistics and therefore the EDA allowed me to explore less common player performance metrics based on current matches. Another benefit of doing EDA was developing intuition on what can affect match outcome in game. EDA also laid to rest my curiosity on finding a connection between champion roles and match times. However, although interesting it does not help me solve my proposed problem of predicting the outcome of matches based on pre-match decisions. I ended cutting out all features that a player could not directly choose at the champion select screen pre-match for the next step of machine learning.

## IV. MACHINE LEARNING

### Introduction

As a recap, the capstone project will utilize Riot Game's API to pull data on player matches from the 2018 season. The matches come from high ranking players in Diamond tier and above. When initially cleaning the data and forming a Pandas Data

Frame, each row of data represents one match. Each row, initially comprised of over a 1000 features, including post game stats and player choices (champion picks, runes, summoner spells, etc). I decided I wanted to predict the outcome of a match just based on player choices before a game. There were two target predictions, red side and blue side win. The outcome for either prediction would be 'Win' or 'Fail'. There are many tools online already online that utilize post game statistics to help players create the most optimal build or sites that display many game statistics and analytics. Predicting which team will win based on pre-match choices is something I have not seen online. Initially, I thought it would prove challenging because of the variance in player skill levels. Therefore, I ended up at the idea of creating a predictive model for high level matches only. This made more intuitive sense because players at the highest level all had a very high understanding of game play and game mechanics. Players at a high level are also more determined to win a game versus playing casually to have fun. With that assumption, outcomes could be better predicted as a result of team composition with the optimal summoner spells and runes.

The result of this model could develop into an application where players can analyze match ups and team combinations. League of Legends is a very popular game in Esports right now. This model could develop into a tool used by professional teams and their coaches to gain insight on developing new team composition strategies versus common picks in professional play. This could also be a useful and fun tool for shout casters to add to the repertoire of pre-game match statistics.

The GitHub link to this project:
https://github.com/jltsao88/Capstone_Project_1/blob/master/Machine_Learning.ipynb

## The Machine Learning Problem

The problem is a classification problem where the model is trying to classify matches as either a 'Win' or 'Fail' for a specific team. The Data Frame being used initially contained these features: Champion pick for each player, Summoner Spell of each player, the primary and secondary runes (labeled as perks in the data), the role of the champions picked, number of that type of role in game, red team win, and blue team win. The features 'blue_team_win' and 'red_team_win' were removed and stored as the target to predict. You will see I pulled out game length as well, but did not end up using it. The data was further split up into two feature sets. Feature Set 1 included the champion roles and number of champions. Feature Set 2 had roles and role counts removed. The intuition for 2 feature sets was that champion role features may end up just being noise in the model and I wanted to test that intuition. The features were all in categorical form and pre-processing was needed to create dummy variables for every feature. Pandas' .get_dummies() function was used to achieve this.

## Picking the Machine Learning Models

Since this is a classification problem, I decided to test 4 different classification models. They were Logistic Regression, SGD Classifier, Support Vector Machine Classifier, and Random Forest Classifier. This project utilizes Python's ***sklearn*** package to implement machine learning.

First, I created a training and test sets for Feature Set 1 and 2. Next, for all 4 models I initially used the default implementation without setting any hyper parameters to see how each model performed out of the box to predict Red Side wins. Feature Set 1 was

used as the initial test for these models. Each model was fitted to the training data and tested on the test data. The default accuracy score was used to get an idea of model performance. The out of the box scores were as followed: SGD: 0.536, SVC: 0.505, Logistic Regression: 0.566, Random Forest: 0.807. There was already a clear winner in performance based on accuracy. I also looked at the classification report for each model, looking at precision, recall, and f1-scores. Still, Random Forest came out on top. Just to make sure Random Forest was the best, I tried hyper parameter tuning and cross validation on the models. I also looked at AUC score for the ROC. Random Forest was still the best performer. With that, I went more in depth with fine tuning the hyper parameters of the Random Forest Model.

Because Random Forest performed so well I decided to tune the hyper parameters: 'n_estimators', 'max_features', 'max_depth', 'min_samples_split', 'min_samples_leaf', and 'bootstrap'. Look at the GitHub link for the details. I used a randomized grid search with 3 fold validation due to the training time and computational limits of my personal computer. I found optimal hyper parameters which slightly improved accuracy but saw great improvements in precision, recall, and f1-score. I decided to plot the ROC curve to get a visually sense of where the model was at. I trained the model again using the second feature set and saw slight improvements and decided the champion roles added noise to the model. From there on, I used the Feature Set 2 for the models. With Feature Set 2, the AUC score was 0.93. I repeated the steps again with predicting Blue Side win with similar results. The AUC score for predicting blue side win was slightly better at 0.932. Based on the results, Random Forest does a good job at classifying

wins for pre-game match features. I tried using a smaller test and training set to speed up the process, but the results were far worse by doing so.

Random Forest is an ensemble model and aggregates a bunch of decision trees together and uses that to make predictions. There are around 140 champions, 10 commonly used summoner spell, and a variety of rune choices. This makes the number of team compositions enormous. Intuitively, Random Forest would be the best in finding these choices to lead to a win or fail. Also, most of the features when split into dummy variables are binary choices. The other models used some sort of regression or draw lines/hyperplanes to predict the data. With the way the features are chosen and currently set up, it would seem the other models have a harder time at classifying. In the case of SVM, the run time to fit the model was extremely long and I could not test different hyper parameters with my current machine.

**Final Thoughts and Intuition**

I want to reinforce the idea that a successful outcome using the Random Forest model would most likely be applicable only in high level games or professional play. I would assume that the model would perform far worse in games where players are playing for fun, have inexperience, and high variation of skill levels. Overall, I am happy with how the model performed. I developed some intuition afterwards about why the model could not make better predictions. For one, the game is 10 players and not all player can control their emotions in a match. A common occurrence in games are ‘rage quitting’ or ‘intentionally feeding’. When that happens, one or a few players will get upset and basically throw the match. Unfortunately, it was difficult to find those matches were players intentionally threw the game.

Another afterthought was taking into account ‘one tricks’ in matches. These are players who predominantly play one champion and therefore have higher win rates and play rates of a certain champion. This could also skew certain features of the data. In hindsight, it would have been better to remove known ‘one trick’ players, but compiling a list would be a daunting task. Including those ‘one tricks’ in the current data may have led to some over fitting in the model. Making a generalized model would be difficult due to various skill levels over the different tiers of players. Ideally, a separate model would need to be trained for each player tier and every region. I do not believe a generalized model would be practical and lead to poor performance. If making a tool for shout casters at professional events, one could perform the steps in the model but used the data from matches of professional play only.