project

December 16, 2022

1 Analyzing High-Ranked League Of Legends Matches

Khawaja Arsalan Khalid

CMSC320

Final Tutorial

December 2022

1.1 Introduction

League of Legends, also known as LoL, is one of the most popular MOBA's (Multiplayer Online Battle Arena) in the world. It is played by over 100 million active users every month, actively developed by Riot Games, and has a popular esports industry composed of 12 leagues. It is also reknown for its tie-ins with music videos, comic books, short stories, and an animated series, Arcane.

LoL is a free-to-play team-based strategy multiplayer game where two teams of five "champions" battle in player versus player combat, each team occupying and defending one half of the map (Blue vs Red). Each player controls one character known as a "champion", with a unique playstyle and multiple abilities. There are 162 champions in the game currently, leading to a vast number of combinations of possible games.

Games usually last between 20-35 minutes, but can go on for even longer depending on team matchups and player skill.

Currently, Riot Games has an API that allows developers to obtain game information and statistics for every point in the game. However, there are premade datasets of game data that we are going to be using in this tutorial. The reason for this choice is we want to analyze games at the highest level to get rid of any underlying player skill differences. Getting data of this level from the API would take months to collect and aggregate, so we use this **dataset** instead.

The goal of this tutorial is to see if we can find a way to analyze champions in the game and gain some insight into how a match is won or lost. Since each champion is different, it might be difficult to see how this affects their win rate, but we might want to see how the champion roster performs overall and if some in-game information can allow us to predict a team's chance of winning.

1.1.1 League Terminology

LoL is played on a map that looks like this:

In this picture, we can see that there are towers located across the map with each team having 3 in each lane and 2 at the end for a total of 11 towers. As well, behind the third tower, there is a crystal which is called an inhibitor and can be destroyed after the tower in front of it is destroyed.

During a match, champions become more powerful by collecting experience points, earning gold, and purchasing items to defeat the opposing team. The ultimate goal of League is to destroy the other team's base, known as "Nexus", but it's not easy. Your enemies will do everything they can to kill you and destroy your base. Each base has a series of turrets and waves of minions that constantly spawn, as well as neutral monsters within the Jungle that can grant points or power ups. Your team needs to clear at least one lane to get to the enemy Nexus.

By the river in the middle, we see that there are two holes which can contain monsters that either team can slay for temporary or permanent power ups. In the lower half, the hole contains the dragon which provides permanent power ups. In the upper half, before 20 minutes, the hole will contain a Rift Herald which can help with destroying towers by ramming its head into them. However, after 20 minutes, it will contain a Baron which, when slain, provides a benefit to your team and your minions allowing you to destroy towers quickly.

1.2 Importing Data

We will be using Python 3 along with a few libraries: **pandas**, **numpy**, **matplotlib**, **scikit-learn**, **searbon**, **re**, and **random**.

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import recall_score, balanced_accuracy_score,␣
 ↪confusion_matrix, ConfusionMatrixDisplay, precision_score
from sklearn.neighbors import KNeighborsClassifier
```

```
import re
from sklearn.linear_model import LinearRegression
import numpy as np
from random import choice


# Type hint dataframe to ensure we get the right type
winners_df : pd.DataFrame = pd.read_csv("./winners.csv")
losers_df : pd.DataFrame = pd.read_csv("./losers.csv")
matches_df : pd.DataFrame = pd.read_pickle("./matches.pickle")
```

The datasets are divided into the dataset of winning players, losing players, and the match information. The winner and loser datasets come in the form of a (Comma Separated Value) CSV file, while the match information comes in the form of a **pickle** file.

1.3 Looking at Player Data

First, we're gonna look at the player data stored in the two winners and losers dataset. We're dealing with almost 108000 games where we have 5 winners and 5 losers. However, for us, it only matters which side won and which side lost.

So, we combine the two datasets to give us one dataframe containing both winners and losers. Then, we replace the "Fail" in the win column with "Lose" since that matches the theme of winning and losing. Also, before we start looking at and parsing the data, we want to make sure that all fields are defined and there are no empty cells in the dataframe. For that, we drop any NA values in the dataframe.

This ends up being useful in our case, but some datasets might not have any missing values, so this could be redundant for them.

1.3.1 Tidying Player Data

[2]:
```
total_matches_df = pd.concat([winners_df, losers_df])
total_matches_df["win"].replace("Fail", "Lose", inplace = True)
total_matches_df.dropna(axis = 0, how = "any", inplace = True)
```

[3]:
```
total_matches_df.head()
```

[3]:

	Unnamed: 0	teamId	win	firstBlood	firstTower	firstInhibitor	firstBaron	firstDragon	firstRiftHerald	towerKills	inhibitorKills
0	0	200	Win	False	True	True	False	True	True	9	1
1	1	100	Win	False	False	False	False	True	True	4	0
2	2	200	Win	True	True	True	False	True	True	5	1
3	3	200	Win	True	True	False					
4	4	100	Win	True	True	True					

```
3       False        False             True           6               0
4        True         True             True          11               3

   baronKills  dragonKills  vilemawKills  riftHeraldKills  \
0           0            3             0                2
1           0            2             0                2
2           0            2             0                2
3           1            3             0                1
4           2            2             0                2

   dominionVictoryScore                                              bans  \
0                     0  [{'championId': 523, 'pickTurn': 6}, {'champio…
1                     0  [{'championId': 523, 'pickTurn': 1}, {'champio…
2                     0  [{'championId': 350, 'pickTurn': 6}, {'champio…
3                     0  [{'championId': 81, 'pickTurn': 6}, {'champion…
4                     0  [{'championId': 30, 'pickTurn': 1}, {'champion…

         gameId
0  4.247263e+09
1  4.247156e+09
2  4.243963e+09
3  4.241678e+09
4  4.241539e+09
```

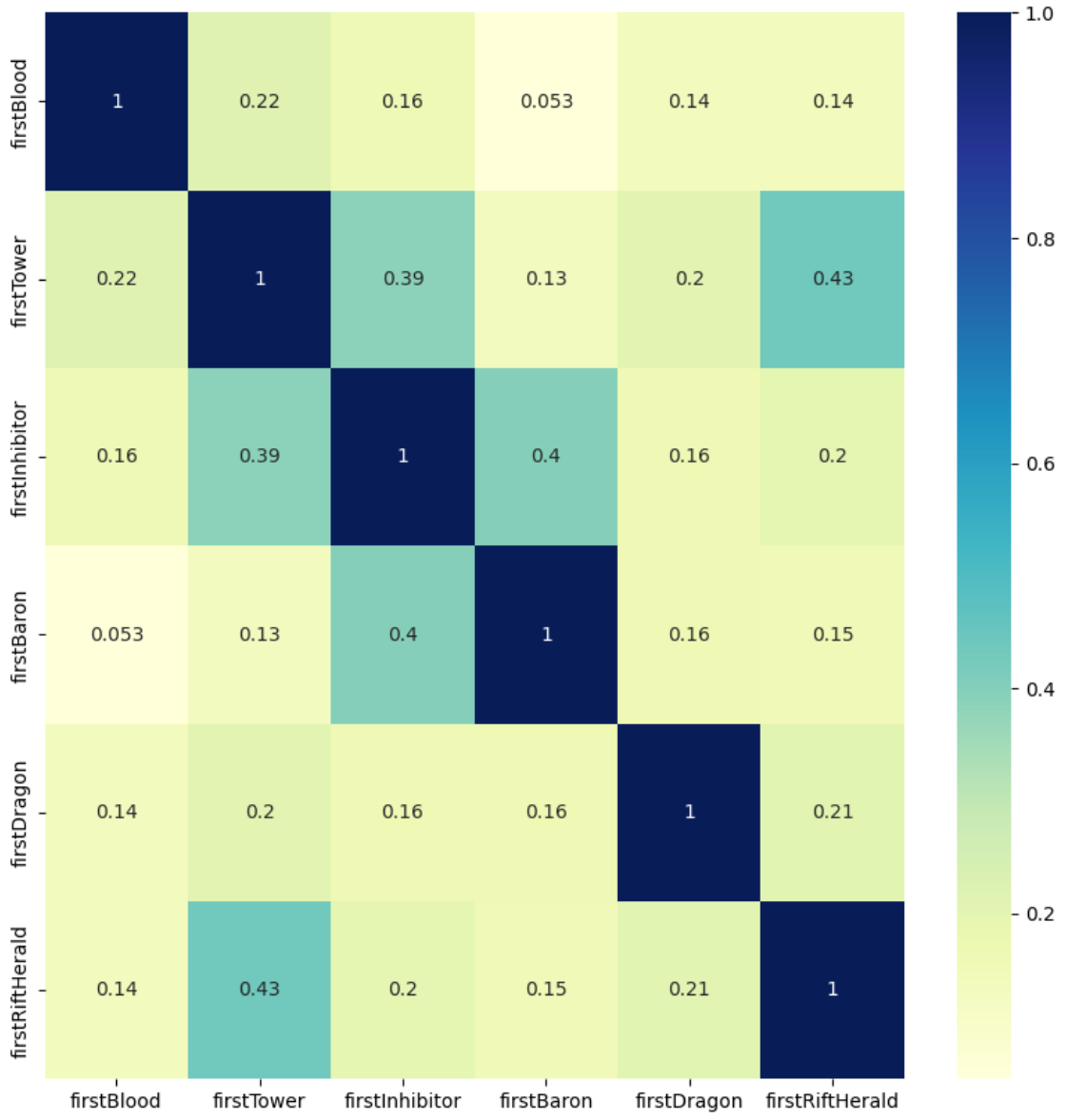
In the above game dataset, we have 17 columns. We have: 1. teamId => contains information on which team won (100 for Blue, 200 for Red) 1. win => whether the team won or lost, currently incorrect information 1. firstBlood => whether the winning team got the first elimination in the game 1. firstTower => whether the winning team were the first team to destroy a tower 1. firstInhibitor => whether the winning team were the first team to destroy an inhibitor 1. firstBaron => whether the winning team were the first team to slay the Baron 1. firstDragon => whether the winning team were the first team to slay the first Dragon 1. firstRiftHerald => whether the winning team were the first team to slay the Rift Herald 1. towerKills => number of towers destroyed by the team specified in teamId 1. inhibitorKills => number of inhibitors destroyed by the team specified in teamId 1. baronKills => number of Barons killed by the team specified in teamId 1. dragonKills => number of Dragons killed by the team specified in teamId 1. vilemawKills => this column is for a limited time game mode which isn't available anymore 1. riftHeraldKills => number of Rift Heralds killed by the team specified in teamId 1. dominionVictoryScore => victory score (0 in all cases) 1. bans => list of bans by both teams as a json 1. gameId => ID of the game used as a foreign key for matches dataset

1.3.2 Visualizing the Player Data

We don't need to do any more tidying of the data since we have all the information we need. First, we want to get all the first type columns (firstBlood, firstTower, …). We can then use a heatmap and see whether any of these columns have any correlation.

[4]:

```
# Only get the needed columns
total_matches_df = total_matches_df[["win", "firstBlood", "firstTower",␣
 ↪"firstInhibitor", "firstBaron", "firstDragon", "firstRiftHerald"]]
plt.figure(figsize = (10, 10))
sns.heatmap(total_matches_df.corr(numeric_only = True), cmap = "YlGnBu", annot␣
 ↪= True)
plt.show()
```



Most of the columns seem to be unrelated except for the firstTower and firstRiftHerald which seem to be moderately related. This can be explained by the fact that a team that slays and obtains the Rift Herald can do massive damage to a tower and can more easily destroy the first tower.

Isolating effect of features on Win Percentage Now, we want to see how each feature individually affects the chance that a team wins. A feature's importance can be defined as the percentage of times it agrees with the win variable. In this case, agreeing means that when the feature is false, the team lost and when the feature is true, the team won. So, the importance of a variable can be defined as the number of times it agrees with "win" divided by the number of games.

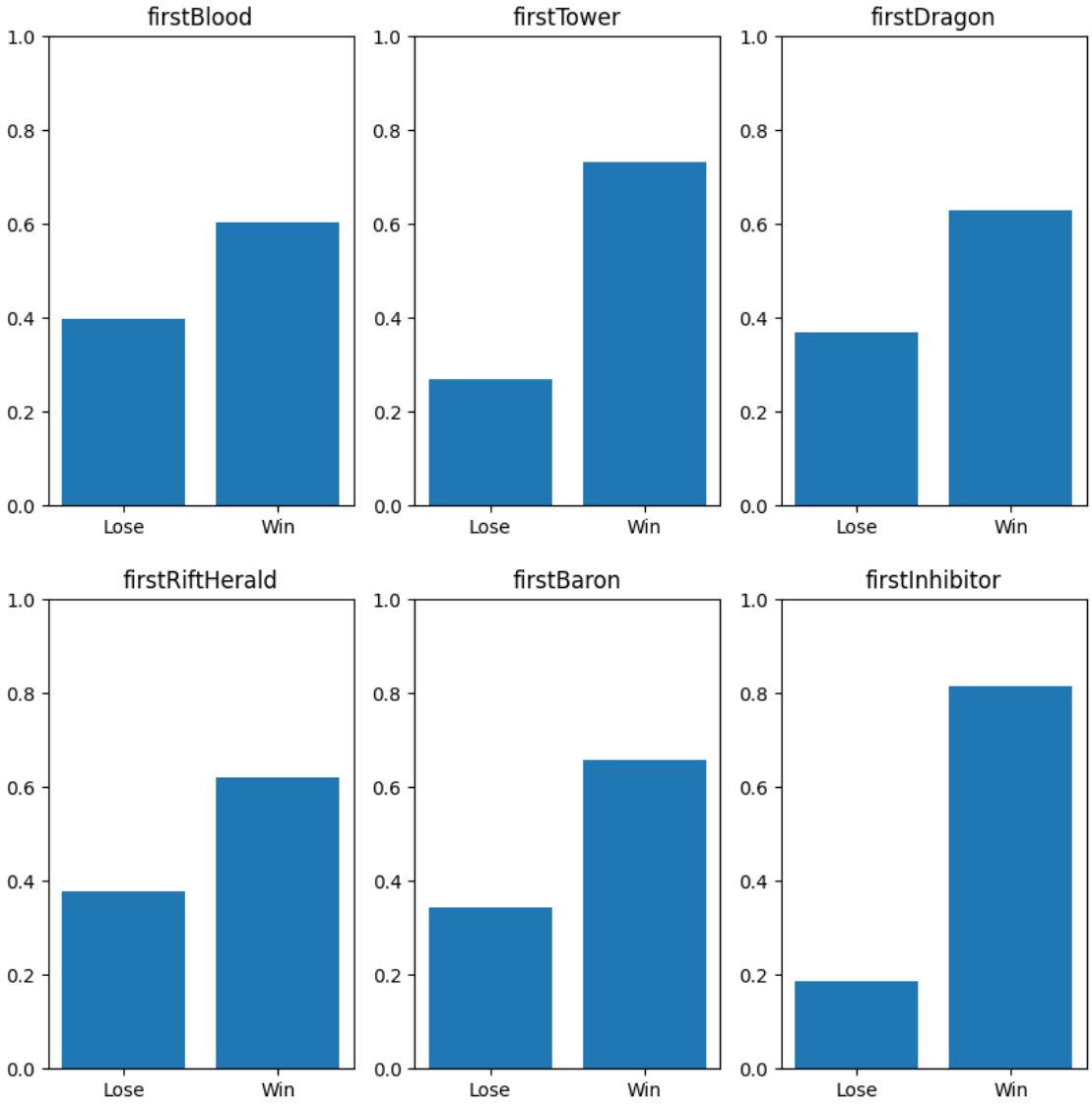
Importance = # of agreeing observations / total number of observations

Using this formula, we can see how well a feature can predict a game's end result and thus, define a sort of hierarchy of importance in a game.

[5]:
```
fig, axes = plt.subplots(2, 3, squeeze = False, figsize = (10, 10))
features = ["firstBlood", "firstTower", "firstDragon", "firstRiftHerald",␣
 ↪"firstBaron", "firstInhibitor"]

for i in range(len(features)):
    d = total_matches_df.apply(lambda x : (x["win"] == "Win" and x[features[i]]␣
 ↪== True or x["win"] != "Win" and x[features[i]] == False), axis = 1)
    v = d.sum() / len(d)
    h = [1 - v, v]
    # b is the row, a is the column
    a = i % 3
    b = i // 3
    axes[b][a].bar(x = ["Lose", "Win"], height = h)
    axes[b][a].set_title(features[i])
    axes[b][a].set_ylim(0, 1)

plt.show()
```



From the graphs above, we see that most features, when true, do provide a benefit to the team and increase their chances of winning. However, two features stand out amongst the group, firstInhibitor and firstTower. Why?

This can be seen from the effects of destroying an inhibitor or tower. Destroying an enemy inhibitor leads to your minions in that lane getting buffs and becoming temporarily stronger. This leads to more pressure on the enemy team which can allow you to turn that into a bigger advantage. The same cannot be said for the first tower. However, usually getting the first tower allows you to leave your assigned lane and go harass people in other lanes allowing your teammates to get an advantage.

The advantages gained from them more easily convert into winning games than the other 4 conditions.

1.3.3 Modeling and Predicting Player Data

Now that we know how important some of these features can be, we might want to try and predict a game's outcome based on these variables. The first thought that comes to mind is the **Decision Tree** model from scikit-learn. The reason we use this model first is it follows from the information we just got. We were checking how important a feature is for each game. The decision tree works similar to that and can assign importances to specific features allowing us to see their effects.

To test our model, we will use 3 metrics, precision, recall, and balanced accuracy. Precision is defined as the number of positive true predictions divided by the total number of positive predictions. Recall (hit rate, true positive rate) is defined as the number of true positives divided by the number of total positives. Balanced accuracy is defined as the average of the true positive rate and the true negative rate. For more information, click **here**.

[6]:
```
y = total_matches_df["win"]
x = total_matches_df.drop(["win"], axis = 1)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3,␣
 ↪stratify=y)


dc = DecisionTreeClassifier().fit(x_train, y_train)
pred = dc.predict(x_test)
prec = precision_score(y_true = y_test, y_pred = pred, pos_label = "Win")
rec = recall_score(y_true = y_test, y_pred = pred, pos_label = "Win")
bac = balanced_accuracy_score(y_true = y_test, y_pred = pred)

print(f"Precision: {prec}")
print(f"Recall: {rec}")
print(f"Balanced Accuracy: {bac}")
fn = ["firstBlood", "firstTower", "firstInhibitor", "firstBaron",␣
 ↪"firstDragon", "firstRiftHerald"]
# Pad the strings with spaces to make them look better when the user is reading␣
 ↪them
max_string_len = max([len(f"Importance of {i} :") for i in fn])
print("Feature importances:")
for (x, y) in zip(fn, dc.feature_importances_):
    s = f"Importance of {x} :"
    pos = s.find(":")
    s = s + " " * (max_string_len - len(s)) + f" {round(y, 5)}"
    print(s)
```

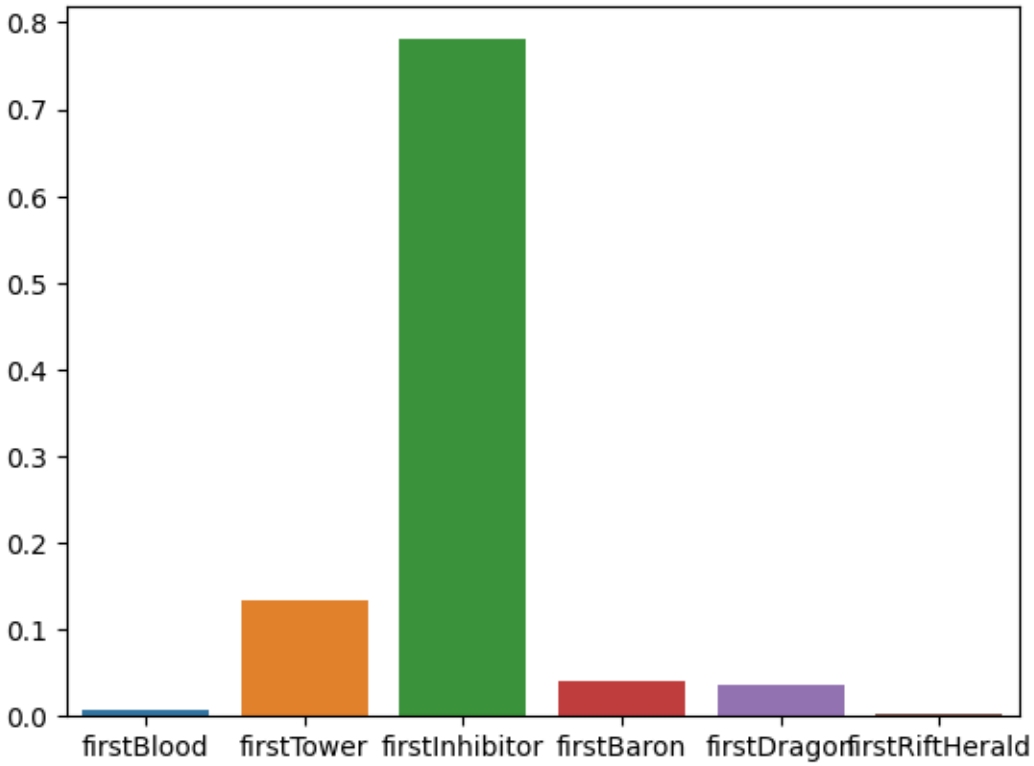
```
Precision: 0.8334524660471766
Recall: 0.8571166038776072
Balanced Accuracy: 0.8429096574296904
Feature importances:
Importance of firstBlood :      0.00806
Importance of firstTower :      0.13359
Importance of firstInhibitor :  0.78049
Importance of firstBaron :      0.04066
```

```
Importance of firstDragon :     0.0351
Importance of firstRiftHerald : 0.00209
```

Looking at our model, we see that the precision, recall, and balanced accuracy all hover around 83-85%. This means our models are reasonably able to predict a game's outcome from these features. As well, if we look at our decision tree's importances, we see that, as expected, it highly values the firstInhibitor with an importance of 0.78. We also see that features like firstRiftHerald and firstBlood have near no significance in the model.

Visualizing the Decision Tree Model We can visualize these importances and understand how the decision tree weighs them

[7]:
```
sns.barplot(x = fn, y = dc.feature_importances_)
plt.show()
```



It is also possible to graph the model and see the tree itself, but in our case, the tree splits 64 times and is very difficult to read. However, the functionality does **exist**.

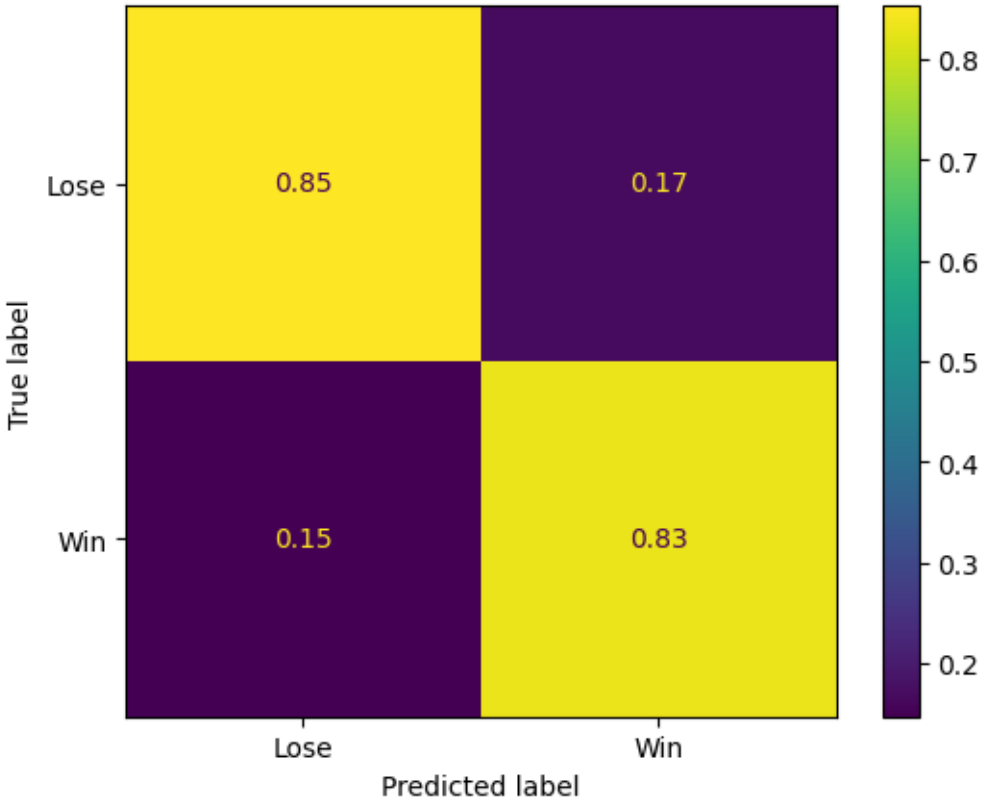
Looking at Confusion Matrix Another method of evaluating a decision tree which combines precision, recall, accuracy, and all other metrics is the confusion matrix. The confusion matrix shows us what the algorithm predicted and the real value and shows us how this changes in the dataset.

[8]:
```
# Normalize the confusion matrix to display as a fraction of the predicted␣
↪values
cfm = confusion_matrix(y_test, pred, normalize = "pred")

disp = ConfusionMatrixDisplay(cfm, display_labels = dc.classes_)

disp.plot()

plt.show()
```



As we can see from our confusion matrix, approximately 83-85% of the time, the algorithm predicts correctly since the predicted and true labels agree in those cases.

1.3.4 Creating a K-Neighbors model

Let's try to look at a different learning model.

A K-Neighbors model looks at the K neighbors around that data point and checks their "win" value. In essence, we want to classify our data based on what our neighbors classify it as. If our current situation closely matches others, and they have won their game, we are likely to win ours as well.

However, in practice, we might not know how many neighbors we should look at. Only one, nearest

5, nearest 50? We can test these values out and see where the computing cost outweighs the benefits gained by increasing the number of neighbors. Essentially, we're looking for when the slope approaches 0 and our increased precision/recall/accuracy is not worth the training/computation time.

Picking an optimal number of neighbors

[9]:
```
y = total_matches_df["win"]
x = total_matches_df.drop(["win"], axis = 1)
indices = []
precisions = []
recalls = []
accuracies = []
# Loop over the number of neighbors we want to test with
for i in range(5, 401, 5):
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3,␣
 ↪stratify=y)

    # Model with i neighbors and weight it by distance
    knn = KNeighborsClassifier(n_neighbors = i, weights = "distance").
 ↪fit(x_train, y_train)
    pred = knn.predict(x_test)
    indices.append(i)
    precisions.append(precision_score(y_true = y_test, y_pred = pred, pos_label␣
 ↪= "Win"))
    recalls.append(recall_score(y_true = y_test, y_pred = pred, pos_label =␣
 ↪"Win"))
    accuracies.append(balanced_accuracy_score(y_true = y_test, y_pred = pred))
```

This takes a few minutes to train since we're training ~80 models with increasing complexity and tracking their values. However, we can then graph these values and look at which number of neighbors we should try to look at.

Visualizing K-Neighbors Model

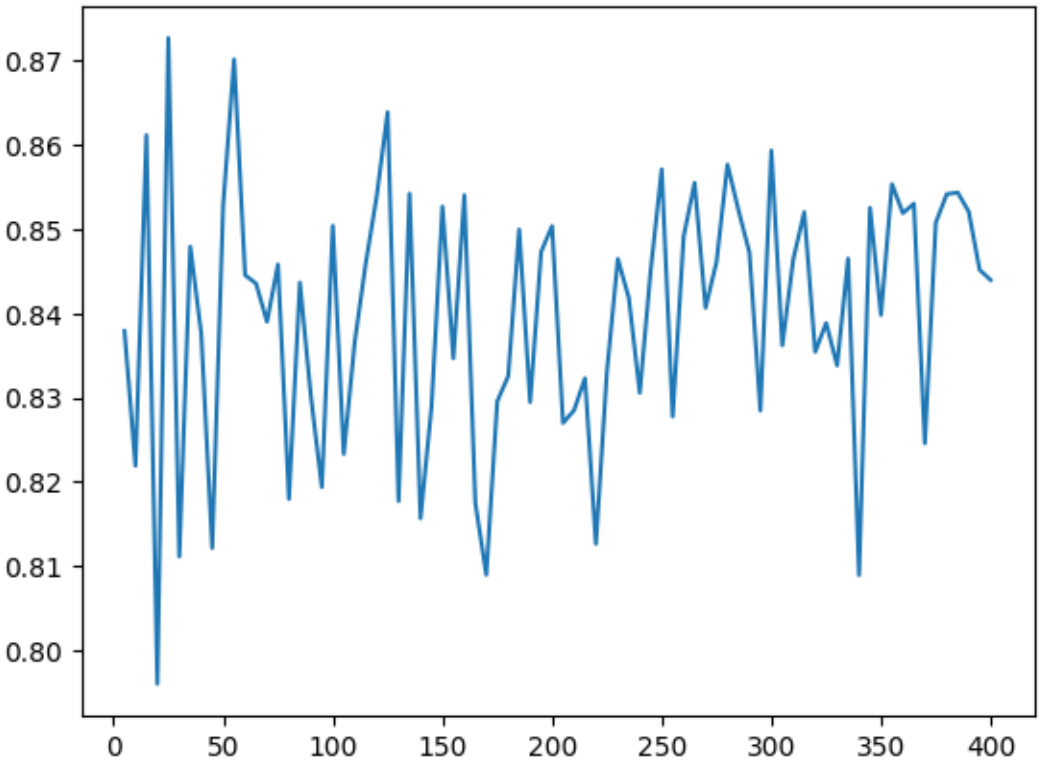
[10]:
```
print(np.median(recalls))
print(np.median(accuracies))
print(np.median(precisions))
```

```
0.8438237005727587
0.8406588837215224
0.839613898695619
```

Looking at the median values, we see that the k-neighbors model has metrics comparable to the decision tree with 83-85% for all 3 metrics. We can plot these to get a better idea of what k values to pick.

[11]:
```
sns.lineplot(x = indices, y = recalls)
```

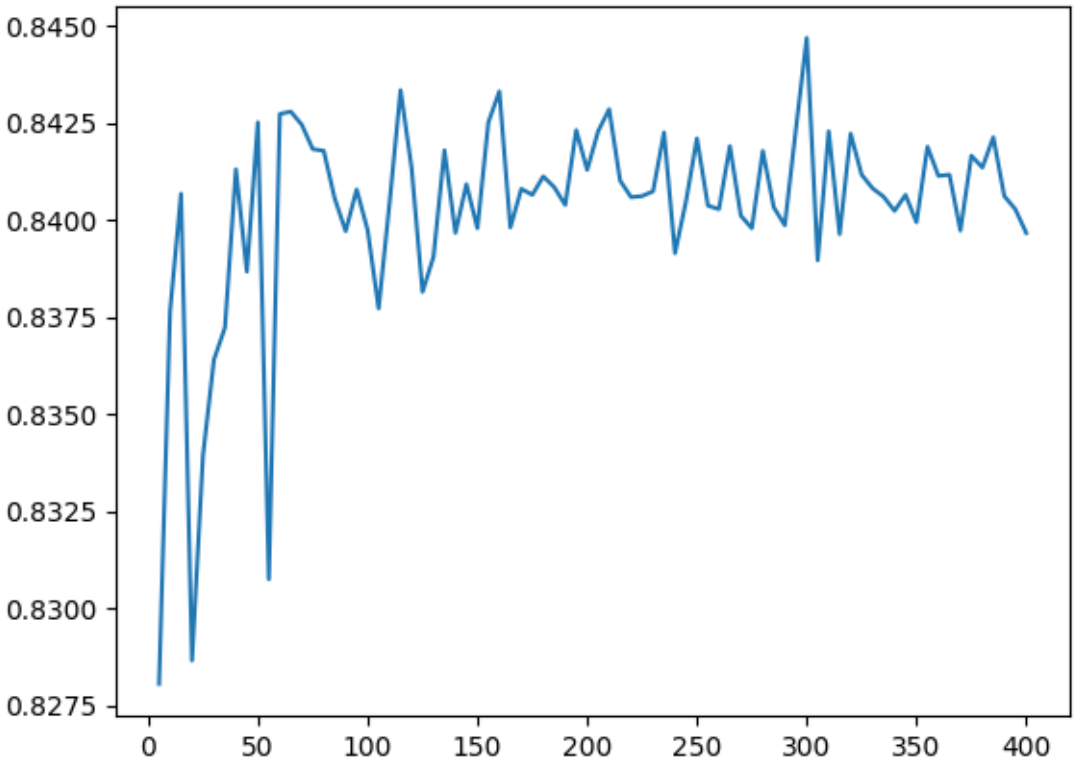
[11]:
```
<AxesSubplot: >
```



Looking at the graph for recalls, it doesn't seem to stabilize as we increase k, but still hovers around 85% for high values of k. So, as long as we pick a value > 50, it's just a matter of testing with a slight change and seeing which is good.

[12]:
```
sns.lineplot(x = indices, y = accuracies)
```

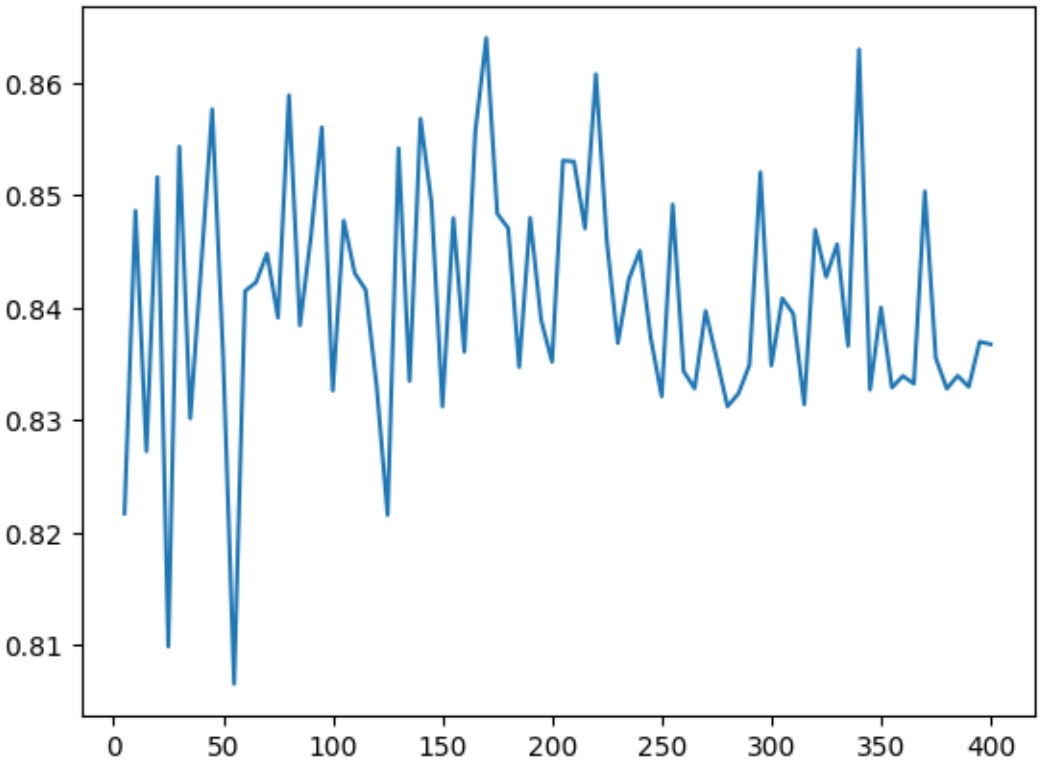
[12]:
```
<AxesSubplot: >
```



Looking at accuracy, we see a similar theme with it dramatically increasing after picking k to be greater than 50 and stabilizing afterwards.

[13]:
```
sns.lineplot(x = indices, y = precisions)
```

[13]:
```
<AxesSubplot: >
```



Finally, precision doesn't seem to have a pattern for minimum values, but we can see that the graph seems to narrow the further right we go. This could mean that doing more iterations on larger values will make us stabilize at 84-85%.

Overall, for the values of k to choose, it seems that as long as we pick a $k > 50$, our results won't vary much from changing k to be slightly more or less. Picking higher values of k is good, but I would recommend picking values around 250.

1.4 Looking at Match Data

We've looked at the specifics of the game, but now we want to look at the game in a broader sense. We know that there are 162 champions in the game, but which ones are the best? Are newer champions better? Players often complain about the nature of the balance of the game. Usually, this devolves into complaining about specific champions having high win rates, specific roles such as DUO having high win rates, or even specific lanes having high win rates. Is any of this true? We can use the dataset to get this information and see whether any of these statements are true.

[14]:
```
matches_df.head()
```

[14]:

	gameCreation	gameDuration	gameId	gameMode	gameType	\
0	1.585155e+12	1323.0	4.247263e+09	CLASSIC	MATCHED_GAME	
1	1.585152e+12	1317.0	4.247156e+09	CLASSIC	MATCHED_GAME	
2	1.585059e+12	932.0	4.243963e+09	CLASSIC	MATCHED_GAME	

```
3  1.584978e+12        2098.0  4.241678e+09  CLASSIC  MATCHED_GAME
4  1.584973e+12        2344.0  4.241539e+09  CLASSIC  MATCHED_GAME

     gameVersion  mapId                              participantIdentities  \
0  10.6.314.4405   11.0  [{'participantId': 1, 'player': {'platformId':…
1  10.6.314.4405   11.0  [{'participantId': 1, 'player': {'platformId':…
2  10.6.313.8894   11.0  [{'participantId': 1, 'player': {'platformId':…
3  10.6.313.8894   11.0  [{'participantId': 1, 'player': {'platformId':…
4  10.6.313.8894   11.0  [{'participantId': 1, 'player': {'platformId':…

                                        participants platformId  queueId  \
0  [{'participantId': 1, 'teamId': 100, 'champion…         KR    420.0
1  [{'participantId': 1, 'teamId': 100, 'champion…         KR    420.0
2  [{'participantId': 1, 'teamId': 100, 'champion…         KR    420.0
3  [{'participantId': 1, 'teamId': 100, 'champion…         KR    420.0
4  [{'participantId': 1, 'teamId': 100, 'champion…         KR    420.0

   seasonId status.message  status.status_code
0      13.0            NaN                NaN
1      13.0            NaN                NaN
2      13.0            NaN                NaN
3      13.0            NaN                NaN
4      13.0            NaN                NaN
```

In the above match dataset, we have 14 columns. We have: 1. gameCreation => unix timestamp of when the game was created 1. gameDuration => time for game to end in seconds 1. gameId => unique ID used to join this dataset and winners/losers 1. gameMode => different game modes picked for each game 1. gameType => specifies whether the game went through (matched) or was cancelled 1. gameVersion => self-explanatory 1. mapId => version of the map being used 1. participantIdentities => identities of the people playing in the match 1. participants => data about the participants (champion, win/loss, in game metrics) 1. platformId => which region the players were on (only KR in our case) 1. **queueId** => what kind of match was played 1. **seasonId** => which season the match was played in 1. status.message && status.status_code => NaN variables that might hold information on why a match was cancelled?

1.4.1 Tidying Match Data

First, we drop all duplicates from the set of matches since it's possible there are multiple games with differing "win" variables depending on which side we're looking at. Second, we only want to look at CLASSIC matches since those are the competitive ones where we can assume players have equal skill levels and are trying their best. Other game modes may be casual and or limited time opportunities which we don't want skewing our metrics.

[15]:
```
matches_df.drop_duplicates(subset = "gameId", keep = "first", inplace = True,␣
↪ignore_index = True)
matches_df = matches_df[matches_df["gameMode"] == "CLASSIC"]
```

Here, we need to establish a set of principles for a new "win" column. It will state the side that won a match based on the stats of the first person in the participants field. The first person is

always the one with a teamId of 100 (Blue side), so if they won, their whole team won and so the win goes to Blue. Similarly, if they lost, the win goes to Red.

[16]:
```
matches_df["wins"] = matches_df.apply(lambda x : "Blue" if␣
↪x["participants"][0]["stats"]["win"] else "Red", axis = 1)
```

1.4.2 Getting information from match data

Since each match contains a lot of information, we can go through each one and get all the names of every champion played by each side and the roles each player picked and which lanes they picked.

[17]:
```
champs = []

for index, match in matches_df.iterrows():
    dur = match["gameDuration"]
    p = match["participants"]
    roles = []
    lanes = []
    blue_champs = []
    red_champs = []
    for i in p:
        # Add to blue or red depending on teamID
        if i["teamId"] == 100:
            blue_champs.append(i["championId"])
        else:
            red_champs.append(i["championId"])

        # Get current players selected role
        roles.append(i["timeline"]["role"])
        # Get current players lane
        lanes.append(i["timeline"]["lane"])

    # If a team is missing a player, ignore this match
    if len(blue_champs) != 5 or len(red_champs) != 5:
        continue

    # Add to list of matches
    for i in range(len(blue_champs)):
        champs.append([dur, blue_champs[i], "Win" if match["wins"] == "Blue"␣
↪else "Lose", roles[i], lanes[i]])
    for i in range(len(red_champs)):
        champs.append([dur, red_champs[i], "Win" if match["wins"] == "Red" else␣
↪"Lose", roles[i + 5], lanes[i + 5]])
```

[18]:
```
champs_df = pd.DataFrame(champs, columns = ["Match Duration", "Champion ID",␣
↪"Win/Loss", "Role", "Lane"])
champs_df.head()
```

[18]:

	Match Duration	Champion ID	Win/Loss	Role	Lane
0	1323.0	7	Lose	DUO_CARRY	MIDDLE
1	1323.0	350	Lose	DUO_SUPPORT	MIDDLE
2	1323.0	266	Lose	SOLO	TOP
3	1323.0	517	Lose	NONE	JUNGLE
4	1323.0	110	Lose	SOLO	BOTTOM

Now, we can isolate these games by either role or champion ID or lane. Let's check whether a player's role (SOLO, DUO, DUO_CARRY, DUO_SUPPORT) affects their win rate.

[19]:
```
average_wr = len(champs_df[champs_df["Win/Loss"] == "Win"]) / len(champs_df)
print(f"Average Win Rate : {round(average_wr * 100, 3)}")
roles = champs_df["Role"].unique()
roles = np.delete(roles, np.where(roles == "NONE"))
for i in roles:
    df = champs_df[champs_df["Role"] == i]
    d = len(df[df["Win/Loss"] == "Win"])
    v = d / len(df)
    print(f"{i}'s win rate is {round(v * 100, 3)}%")
```

```
Average Win Rate : 50.0
DUO_CARRY's win rate is 49.865%
DUO_SUPPORT's win rate is 50.574%
SOLO's win rate is 49.421%
DUO's win rate is 49.114%
```

Well, those numbers seem to tell us something slightly different. It seems that a player's win rate doesn't change depending on their role unless the role is DUO_SUPPORT. Any other role you pick has a ~49% win rate. We removed the NONE role from our list since we don't care about players whose role wasn't recorded.

We can try to understand why DUO_CARRY and DUO_SUPPORT have a higher win rate than SOLO or DUO. Playing DUO in different lanes seems to be the worst option since the impact one player has doesn't translate well into an impact on the other player's lane. In other words, gaining an advantage doesn't translate to an advantage for your duo partner which means you're basically playing SOLO. DUO_CARRY and DUO_SUPPORT on the other hand, play in the same lane where one good play by either player translates to an advantage for both of them. Since they can impact each other's plays directly, they can snowball much quicker and affect a game's outcome more drastically. There's also a difference between playing CARRY and SUPPORT since the SUPPORT can also leave the lane and impact the rest of the map whereas most CARRY champions are unable to get around the map as quick.

Now, let's look at how a player's lane affects their win rate. The code will look almost the same, but this time we will look at unique lanes instead of roles.

[20]:
```
average_wr = len(champs_df[champs_df["Win/Loss"] == "Win"]) / len(champs_df)
print(f"Average Win Rate : {round(average_wr * 100, 3)}")
lanes = champs_df["Lane"].unique()
lanes = np.delete(lanes, np.where(lanes == "NONE"))
for i in lanes:
```

```
    df = champs_df[champs_df["Lane"] == i]
    d = len(df[df["Win/Loss"] == "Win"])
    v = d / len(df)
    print(f"{i}'s win rate is {round(v * 100, 3)}%")
```

```
Average Win Rate : 50.0
MIDDLE's win rate is 50.219%
TOP's win rate is 49.757%
JUNGLE's win rate is 50.249%
BOTTOM's win rate is 49.851%
```

It seems that there's not much to learn from lanes. All of their win rates hover around 50% and barely change. Middle and Jungle have a higher win rate than the other 2 since usually players who get ahead in those lanes can snowball much harder than those in the top or bottom lane.

1.4.3 Reading champion names

Before we look at winrates by champion, you might want to learn more details about a specific champion. Unfortunately, the champions that a person plays are stored as a key, not as the name of the champion. Riot does, however, provide a **json** file that gives us all the information we need about all the champions and their names and keys. However, we only require two pieces of information from this json, so we will use regex to parse it instead.

Champion names are stored as "id" : "{name}" and keys are stored as "key" : "{key}".

Once we're done, we can map each champion's key to their name and vice-versa.

[21]:
```
with open("./champion.json", "r") as f:
    data = f.read()

# Match "id" : "{w}" where w is any number of letters
p = re.compile(r"\"id\":\"(\w+)\"")
names = p.findall(data)
# Match "key" : "{d}" where d is any number of digits
q = re.compile(r"\"key\":\"(\d+)\"")
keys = q.findall(data)

champs_dict = dict(zip(keys, names)) | dict(zip(names, keys))

k = [int(i) for i in keys]
keys = sorted(k)
names = [champs_dict[str(i)] for i in keys]
```

1.4.4 Which champions are being played

As a player, you might want to know what champions the top players are picking to see if you can use that to climb up the ladder. Let's try to calculate the play rate for each champion and see which champion is played the most

[22]:
```
play_rates = pd.DataFrame(columns = ["Champion Name", "Play Rate"])

for name in names:
    id = int(champs_dict[name])
    pr = len(champs_df[champs_df["Champion ID"] == id]) * 100 / len(champs_df)
    play_rates = pd.concat([play_rates, pd.DataFrame([[name, pr]], columns =␣
 ↪play_rates.columns)], ignore_index = True)

# Sort by second value which is the play rate, reverse so it's in descending␣
 ↪order
play_rates.sort_values(by = ["Play Rate"], ascending = False, inplace = True)
play_rates.head()
```

[22]:

	Champion Name	Play Rate
59	LeeSin	3.665464
71	Ezreal	3.044659
20	MissFortune	2.897592
140	Thresh	2.494101
129	Lucian	2.406292

It seems that Lee Sin is the most played champion by a solid margin. We can speculate why this is the case, but another way of knowing is to look at his win rate in all lanes. For that we're going to need information specific to him.

If you want information on a specific champion, it'll be pretty easy from this dataframe. In my case, I picked two champions that I know are played very often in the highest level of the game. The first one is Ezreal, a traditional Bottom Lane Carry who can, in certain situations, be played in other lanes as well. The second is Lee Sin, a high skill common Jungler with the highest play rate of all champions.

Let's see what kind of win rates these two champions have.

[23]:
```
champ1_id = int(champs_dict["LeeSin"])
champ2_id = int(champs_dict["Ezreal"])

champ1_df = champs_df[champs_df["Champion ID"] == champ1_id]
champ1_wr = len(champ1_df[champ1_df["Win/Loss"] == "Win"]) / len(champ1_df)
print(f"{champs_dict[str(champ1_id)]}'s win rate is {round(champ1_wr * 100,␣
 ↪3)}")

champ2_df = champs_df[champs_df["Champion ID"] == champ2_id]
champ2_wr = len(champ2_df[champ2_df["Win/Loss"] == "Win"]) / len(champ2_df)
print(f"{champs_dict[str(champ2_id)]}'s win rate is {round(champ2_wr * 100,␣
 ↪3)}")
```

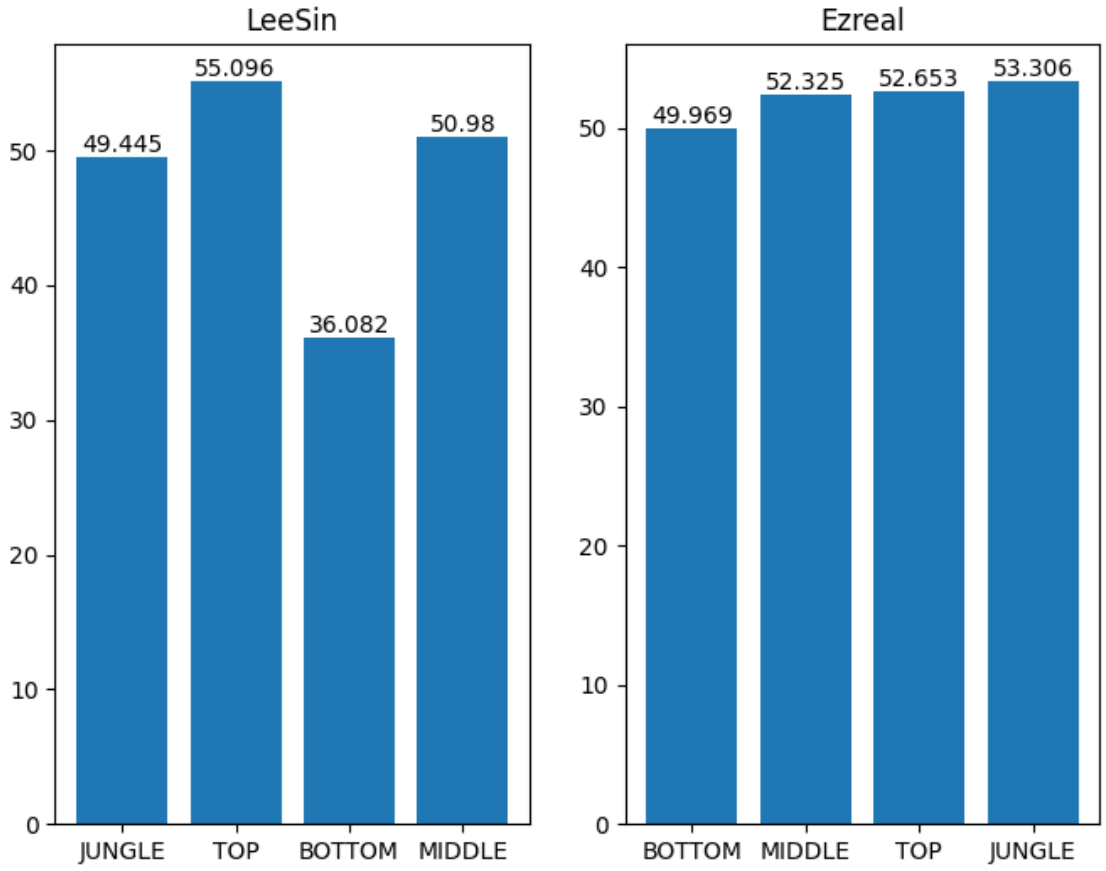
```
LeeSin's win rate is 49.255
Ezreal's win rate is 48.618
```

So, we notice that these two highly played champions seem to have low win rates. Surely, if they're the most played champions, they should have higher win rates that indicate that. Let's try to

account for the different lanes both of these champions play.

[24]:
```
fig, axes = plt.subplots(1, 2, squeeze = False, figsize = (8, 6))
for i in [champ1_id, champ2_id]:
    champ_df = champs_df[champs_df["Champion ID"] == i]
    lanes = champ_df["Lane"].unique()
    lanes = np.delete(lanes, np.where(lanes == "NONE"))
    lane_wrs = []
    for lane in lanes:
        lane_df = champ_df[champ_df["Lane"] == lane]
        w = len(lane_df[lane_df["Win/Loss"] == "Win"]) / len(lane_df)
        lane_wrs.append(round(w * 100, 3))

    a = 0 if i == champ1_id else 1
    bar_champ = axes[0][a].bar(x = lanes, height = lane_wrs)
    axes[0][a].set_title(champs_dict[str(i)])
    axes[0][a].bar_label(bar_champ)
```



There you go! Both champions are played in multiple roles and that seems to affect their win rates a lot. For example, when played in the top lane, Lee Sin has a 55% win rate making him one of

the strongest champions in the game currently, but he has a 36% win rate in the bottom lane. For Ezreal, it seems that his win rate increases as we pick him in unconventional lanes. This high win rate, however, could be due to a smaller number of games, so we should check the number of games to see whether this is the case.

[25]:
```
for i in [champ1_id, champ2_id]:
    print(f"Champion : {champs_dict[str(i)]}")
    champ_df = champs_df[champs_df["Champion ID"] == i]
    lanes = champ_df["Lane"].unique()
    lanes = np.delete(lanes, np.where(lanes == "NONE"))
    for lane in lanes:
        lane_df = champ_df[champ_df["Lane"] == lane]
        print(f"{lane} : {len(lane_df)}")
```

```
Champion : LeeSin
JUNGLE : 24874
TOP : 314
BOTTOM : 97
MIDDLE : 51
Champion : Ezreal
BOTTOM : 19204
MIDDLE : 1269
TOP : 245
JUNGLE : 242
```

Yea, that explains it. For Lee Sin, only around 300 games in top lane leads to an overly inflated win rate. For Ezreal, however, he has more than 1000 games in Middle lane, but has kept his above average win percentage at 52%. Could this mean that playing him in Middle is a good unconventional strategy? It's very likely this win rate will drop a bit if he gets played there more, since people will learn to play against him then. For now, though, it seems to be a good choice to throw off your opponents.

There are more champions that you could do this analysis on to figure out who has a good unconventional swappable lane. However, that would take a lot more work, and you would need to standardize how many games you're looking for before you consider a champion playable in another lane. For now, this is beyond the scope of this tutorial.

1.4.5 Visualizing champion data

Now that we've looked at individual champions, let's look at the win rates for each champion on a graph. For this, we isolate every game where the Champion ID matches our specific champions key. Then we just convert the Win/Loss column to 1's and 0's and divide the sum by the length. This gives us the champion's win rate over the total number of games it has been played in. Since each team has a unique set of 5 champions that can't be picked by the other team, we can be sure that these values are accurate.

[26]:
```
champs_wr = []
for name, key in zip(names, keys):
    champ_key = int(key)
```

```
    current_champ_df = champs_df[champs_df["Champion ID"] == key]
    # Convert our True/False column into a 1/0 column
    current_champ_df["Win/Loss"] = (current_champ_df["Win/Loss"] == "Win").
↪astype(int)
    l = len(current_champ_df)
    # Only add the win rate if the number of games is > 0
    if l != 0:
        champ_wr = current_champ_df["Win/Loss"].sum() / l
        champs_wr.append((key, champ_wr))
```

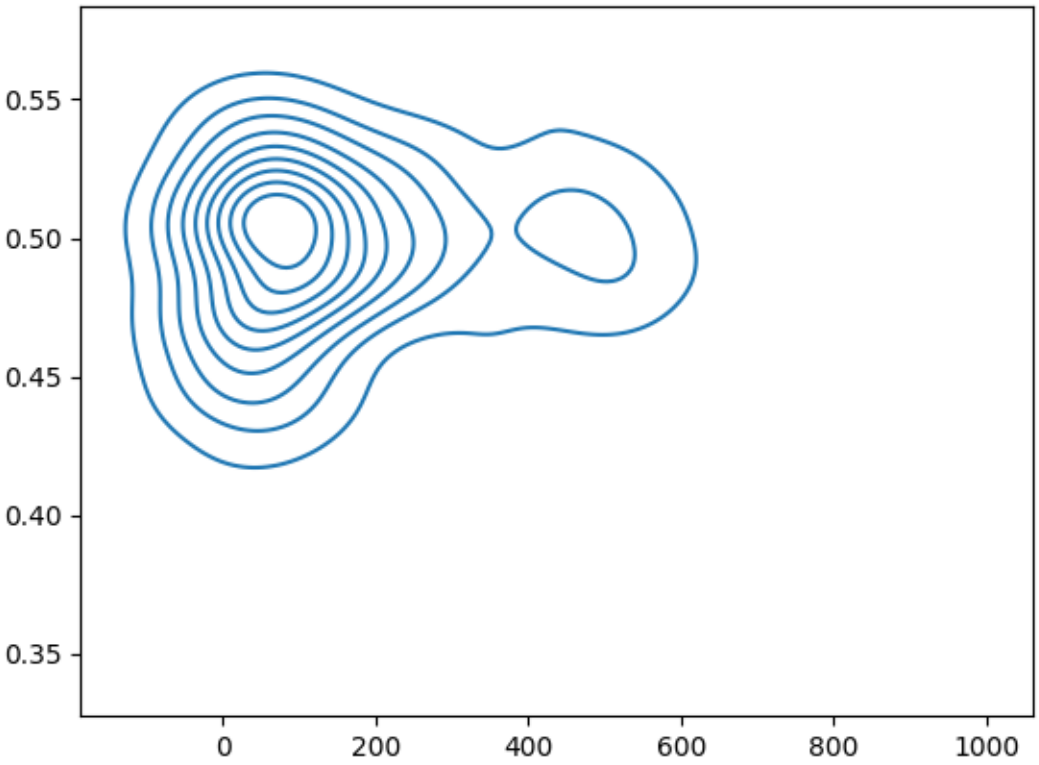
```
/tmp/ipykernel_37915/2971242163.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  current_champ_df["Win/Loss"] = (current_champ_df["Win/Loss"] ==
"Win").astype(int)
```

We can plot these win rates to maybe notice some trends in the champions' win rates. Graphing this with the kdeplot in seaborn gives a pretty satisfying plot that explains everything pretty well.

[27]:
```
sns.kdeplot(data = None, x = [i[0] for i in champs_wr], y = [i[1] for i in␣
↪champs_wr])
```

[27]:
```
<AxesSubplot: >
```

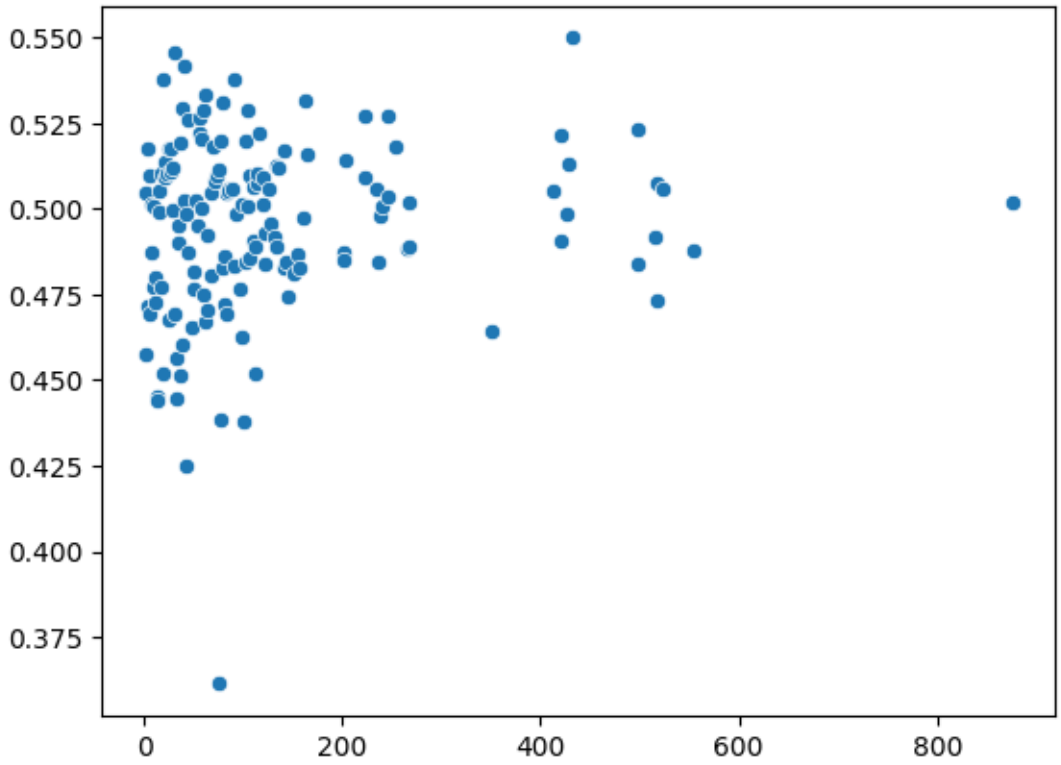


We see that there are some rings around the 50% mark and they seem concentrated around 45-55% as well. There's no discernable pattern in the data and no link between the x and y variables.

However, we can't just look at the graph and state this definitively. We can use a scatter plot and regression line to show this empirically. We will want to look at the residuals for the regression line and the R-squared value for it. We will use this to show that most champions in the game hover at around a 50% win rate.

[28]:
```
sns.scatterplot(data = None, x = [i[0] for i in champs_wr], y = [i[1] for i in␣
↪champs_wr])
```

[28]:
```
<AxesSubplot: >
```



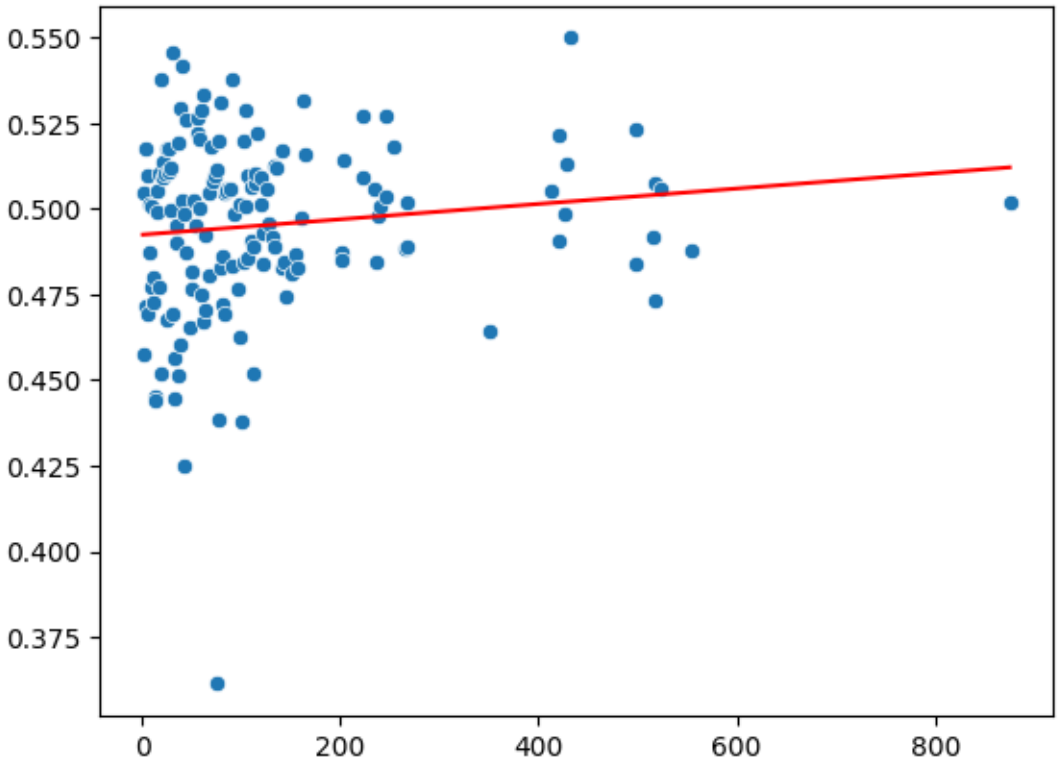
It seems that most champions hover at around 47-53% win rate with some outliers out there. Since more champions are always being introduced and old champions are being changed, the state of these win rates is only accurate for less than a season. These numbers could change the very next day and a 55% win rate champion could become terrible depending on the changes.

Let us, however, test the theory that newer champions have higher win rates than older champions. We can try to create a linear regression and see if the champions Id and win rate are related in any sense.

[29]:
```
x_val = [i[0] for i in champs_wr]
y_val = [i[1] for i in champs_wr]

reg_model = LinearRegression().fit(X = np.reshape(x_val, (-1, 1)), y = y_val)
y_pred = reg_model.predict(np.reshape(x_val, (-1, 1)))
rsquare = reg_model.score(X = np.reshape(x_val, (-1, 1)), y = y_val)

fig, ax = plt.subplots()
# Plot scatter of win rates
p1 = sns.scatterplot(x = x_val, y = y_val, ax = ax)
# Plot regression line on top
p2 = sns.lineplot(x = x_val, y = y_pred, ax = ax, color = "r")
plt.show()
```



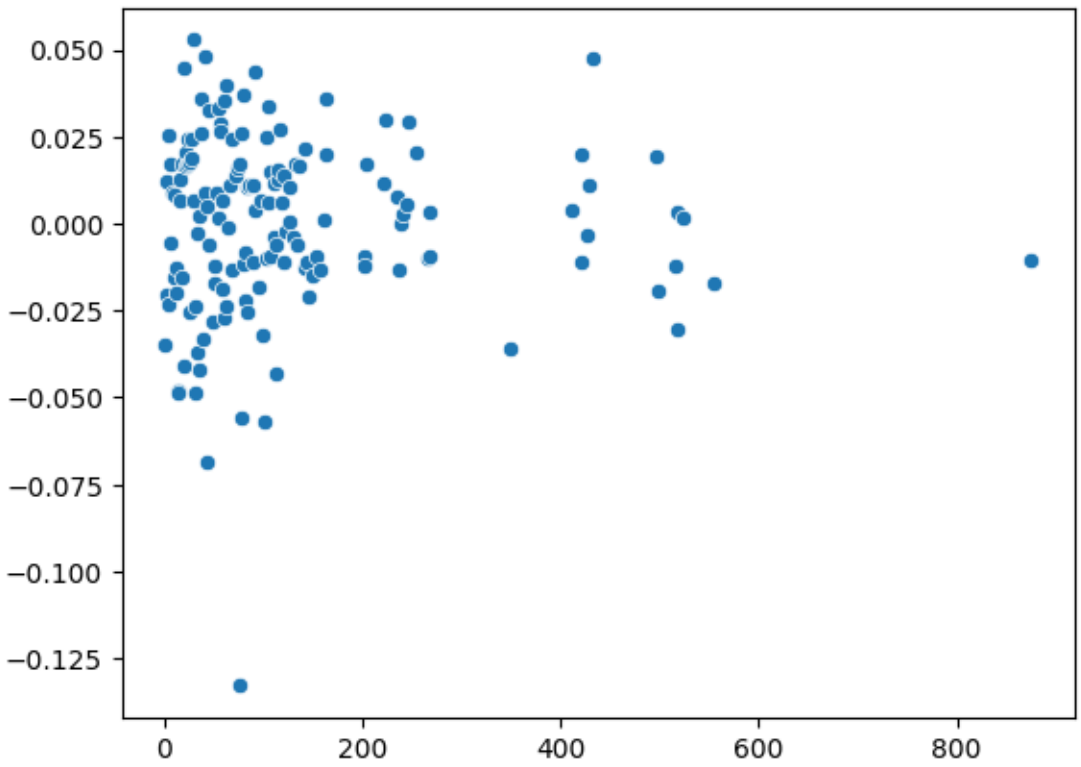
After plotting that regression line on top of our existing graph, we see that it has a near 0 slope. This indicates that the win rates of champions hover around 49-51%. This matches our predictions that most champions hover at around a 50% win rate and there are only a few outliers that are really good or really bad choices.

We can also score this line to see how well it can predict by checking it's residuals and R-squared value.

[30]:
```
residuals = y_val - y_pred

sns.scatterplot(x = x_val, y = residuals)
```

[30]:
```
<AxesSubplot: >
```



Note that the residuals look very similar to the original graph and show that a horizontal line is still the best we can get. Even then, we can get the R-square value of the regression.

[31]:
```
print(f"R-Squared: {rsquare}")
```

```
R-Squared: 0.015363244670800191
```

That low value of R-Squared tells us that there is little to no relation between champion win rates and their keys.

1.5 Conclusion

The game might seem very complex to a newcomer, but hopefully this tutorial explained the relations between some basic in-game concepts and how they affect a match's outcome. For the experienced players, I hope that this has provided you with some more insight on information that might seem like second-nature to you, but that might confuse others. Even if you're not interested in MOBAs, they offer a very good platform for data analysis due to the vast amount of information they store in each game and play. This massive amount of information allows us to see a lot of relations that wouldn't be possible without years of playing.

However, even though we have gained some insights from this data, there's a lot more information left in these datasets. They contain item choices, rune trees, in-game timed statistics, and more. These statistics could also be combined to reveal more information on the effects of these individual features on the game's eventual outcome. As well, we only looked at the top echelon of the ranked

leaderboard. We could look to see lower ranked games and see how their decisions differ from those in the higher ranks. Perhaps, they value objectives differently and have skewed champion win rates.

Even so, we can look at which champions players hate playing against from their bans, which two champion combinations are the best, which rune trees get the most use, and even more. Those are just a few ideas that you could implement from this dataset and ones that I may add soon in another tutorial.