In order to begin processing the League of Legends dataset, all of the csv files were imported into a jupyter notebook in order to determine what preprocessing steps would be needed to convert the data into a useable format. This was done using the package pandas. Upon looking at the head() method of each generated dataframe, It became apparent that only the "LeagueofLegends.csv" file would be needed for the purpose of creating a predictor model.

First, the `.head()`, `.describe()`, and `.info()` methods were called on the main dataframe. It was observed that some columns contained missing values. Upon further inspection, this was a result of the format of the specific tournament the data was referencing. Values such as a team name was missing. Because this dataset was fairly large (~8000 samples) and the number of missing values was quite small (~40), it was decided that the rows containing missing values could be dropped without much consequence. Afterwards, each column of dataframe was inspected in order to determine whether each column would need to be processed and how to process them. Figure 1 below describes how each column of the dataframe would be processed.

```
#1 League_df.League // dummy encode
#2 League_df.Year // dummy encode
#3 League_df.Season // dummy encode
#4 League_df.Type // dummy encode
#5 League_df.blueTeamTag // dummy encode
#6 League_df.bResult // will be used as label
#7 League_df.rResult // dropped because contains same information as bResult
#8 League_df.redTeamTag // dummy encode
#9 League_df.gameLength
#10 League_df.golddiff // calculate var
#11 League_df.goldblue // calculate var
#12 League_df.bKills // retrieve number of kills made by team, will use coordinate data later for plotting where kills occurred
#13 League_df.bTowers // going to convert to column for every turret, value will be time taken, if not taken, value = 0
#14 League_df.bInhibs // convert into number of inhibs taken, will use time for plots later on
#15 League_df.bDragons // create a score metric, based on number of dragons taken and type of dragon taken
#16 League_df.bBarons // create a score metric, similar to dragons
#17 League_df.bHeralds // take lenght to determine if herald was taken
#18 League_df.goldred // going to take total gold at specific time points based on max length of game
#19 League_df.rKills // retrieve number of kills made by team, will use coordinate data later for plotting where kills occurred
#20 League_df.rTowers // going to convert to column for every turret, value will be time taken, if not taken, value = 0
#21 League_df.rInhibs // convert into number of inhibs taken, will use time for plots later on
#22 League_df.rDragons // create a score metric, based on number of dragons taken and type of dragon taken
#23 League_df.rBarons // create a score metric, similar to dragons
#24 League_df.rHeralds // only one herald per game but there are games where herald isnt taken by anyone
#25 League_df.blueTop // dummy encode
#26 League_df.blueTopChamp // dummy encode
#27 League_df.goldblueTop // calculate var
#28 League_df.blueJungle // dummy encode
#29 League_df.blueJungleChamp // dummy encode
#30 League_df.goldblueJungle // calculate var
#31 League_df.blueMiddle // dummy encode
#32 League_df.blueMiddleChamp // dummy encode
#33 League_df.goldblueMiddle // calculate var
#34 League_df.blueADC // dummy encode
#35 League_df.blueADCChamp // dummy encode
#36 League_df.goldblueADC // calculate var
#37 League_df.blueSupport // dummy encode
#38 League_df.blueSupportChamp // dummy encode
#39 League_df.goldblueSupport // calculate var
#40 League_df.blueBans // dummy encode plus fill in empty values as 'empty'
#41 League_df.redTop // dummy encode
#42 League_df.redTopChamp // dummy encode
#43 League_df.goldredTop // calculate var
#44 League_df.redJungle // dummy encode
#45 League_df.redJungleChamp // dummy encodeb
#46 League_df.goldredJungle // calculate var
#47 League_df.redMiddle // dummy encode
#48 League_df.redMiddleChamp // dummy encode
#49 League_df.goldredMiddle // calculate var
#50 League_df.redADC // dummy encode
#51 League_df.redADCChamp // dummy encode
#52 League_df.goldredADC // calculate var
#53 League_df.redSupport // dummy encode
#54 League_df.redSupportChamp // dummy encode
#55 League_df.goldredSupport // calculate var
#56 League_df.redBans // dummy encode
```

Figure 1. Description of how each column in the League of Legends dataframe were to be processed.

Ultimately, due to the structures of the data of the columns, a total of eight functions were needed to completely process all 56 columns of the dataframe.

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

    return df
```

Figure 2. Function used to convert ban columns into separate columns.

The first function takes in a dataframe, columns, and column names as arguments. This function was responsible for converting the “blueBans” and “redBans” columns into five columns, one for each possible ban within the game. This was done so that these columns could be encoded.

	blueBans	redBans
0	['Rumble', 'Kassadin', 'Lissandra']	['Tristana', 'Leblanc', 'Nidalee']
1	['Kassadin', 'Sivir', 'Lissandra']	['RekSai', 'Janna', 'Leblanc']
2	['JarvanIV', 'Lissandra', 'Kassadin']	['Leblanc', 'Zed', 'RekSai']
3	['Annie', 'Lissandra', 'Kassadin']	['RekSai', 'Rumble', 'LeeSin']
4	['Irelia', 'Pantheon', 'Kassadin']	['Rumble', 'Sivir', 'Rengar']

(a)

	blueban1	blueban2	blueban3	blueban4	blueban5	redban1	redban2	redban3	redban4	redban5
0	Rumble	Kassadin	Lissandra	None	None	Tristana	Leblanc	Nidalee	None	None
1	Kassadin	Sivir	Lissandra	None	None	RekSai	Janna	Leblanc	None	None
2	JarvanIV	Lissandra	Kassadin	None	None	Leblanc	Zed	RekSai	None	None
3	Annie	Lissandra	Kassadin	None	None	RekSai	Rumble	LeeSin	None	None
4	Irelia	Pantheon	Kassadin	None	None	Rumble	Sivir	Rengar	None	None

(b)

Figure 3. (a) Bans column prior to processing. (b) Bans columns after processing.

```python
def delete_col_get_dummies(df, cols):

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

    return df
```

Figure 4. Function used to dummy encode columns.

The second function is responsible for processing the columns needed to be converted into dummy variables. If-statements were used to add specific prefixes to the names of each column.

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

    return df
```

Figure 5. Function used to calculate variance of columns.

The third function was responsible for calculating the variance of the numerical columns. This function was most often used on columns with gold values.

```python
def count_num(df, cols):

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

    return df
```

Figure 6. Function used to count the number of objectives each team took.

The fourth function was used to count the number of objectives each team retrieved. This was done by converting the column data type from a string to a literal and then applying the length function on each list.

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

def process_dragons(df, cols, col_names):

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

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

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

    return df
```

Figure 7. Functions used to process dragon columns.

The fifth function and sixth function are used to process the dragon columns. First the columns are converted from a string into a list, and then a list of the dragon types are generated. Another column of how many dragons were killed by each team was generated by taking the length of this list. Finally, the Counter class was used to retrieve a count of how many of each type of dragon was taken, and these counter objects were converted into separate columns.

	bDragons	rDragons
0	[[37.267, None]]	[[17.14, None], [30.934, None], [24.641, None]]
1	[[32.545, None], [26.177, None], [19.119, None]]	[[12.264, None]]
2	[[24.577, None], [37.867, None], [30.87, None]...	[]
3	[]	[[26.274, None], [10.153, None], [18.515, None...
4	[[14.589, None], [30.307, None]]	[[21.901, None]]

(a)

	bNone	bEarth_drag	bWater_drag	bAir_drag	bFire_drag	bElder_drag	rNone	rEarth_drag	rWater_drag	rAir_drag	rFire_drag	rElder_drag
7577	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	2.0
7578	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0
7579	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0
7580	0.0	0.0	1.0	1.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
7581	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	1.0	0.0	1.0	0.0

(b)

Figure 8. (a) Dragon columns before processing. (b) Dragon columns after processing.

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

def convert_to_tower_list(tower_str):

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

    return tower_list


def process_towers(df, cols, col_names):

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

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

    return df
```

Figure 9. Functions and dictionary used to process tower columns.

The seventh and eight functions were used to process the tower columns. The purpose of these functions were to generate a set of columns containing information on when each teams' towers were destroyed. This was done by first creating a dictionary containing information on what index each tower would be located at, then creating a function which would take the list within each cell of each column and assign the times each tower was taken to a specific index using that dictionary. Finally, this was used on the tower columns using the `.apply()` method.

	bTowers	rTowers
0	[[27.542, 'MID_LANE', 'BASE_TURRET'], [39.269,...	[[39.23, 'TOP_LANE', 'INNER_TURRET'], [20.681,...
1	[[23.239, 'BOT_LANE', 'OUTER_TURRET'], [33.018...	[[19.257, 'MID_LANE', 'OUTER_TURRET'], [15.206...
2	[[15.045, 'BOT_LANE', 'OUTER_TURRET'], [39.566...	[[24.62, 'MID_LANE', 'OUTER_TURRET'], [30.493,...
3	[[19.941, 'BOT_LANE', 'OUTER_TURRET'], [38.77,...	[[36.384, 'MID_LANE', 'NEXUS_TURRET'], [31.665...
4	[[22.594, 'MID_LANE', 'OUTER_TURRET'], [34.213...	[[11.644, 'MID_LANE', 'OUTER_TURRET'], [12.438...

(a)

	btop_outer	btop_inner	btop_base	bmiddle_outer	bmiddle_inner	bbmiddle_base	bbot_outer	bbot_inner	bbot_base	rtop_outer	rtop_inner	rtop_base
0	17.856	23.038	0.000	15.014	15.217	27.542	16.556	33.583	0.000	15.288	39.23	0.0
1	15.306	34.766	0.000	25.564	0.000	0.000	23.239	33.018	0.000	23.409	0.00	0.0
2	17.340	35.463	36.946	25.481	32.833	33.135	15.045	30.158	37.109	30.493	0.00	0.0
3	22.879	38.770	0.000	18.541	0.000	0.000	19.941	0.000	0.000	20.760	0.00	0.0
4	18.487	0.000	0.000	22.594	26.963	0.000	14.976	33.717	33.925	15.252	0.00	0.0

(b)

Figure 10. (a) Tower columns before preprocessing. (b) Tower columns after preprocessing.

In terms of outliers, the only true outliers present were certain games whose game lengths were almost two times the average length. It is difficult to determine whether these sets of data will heavily affect the results. Games are often determined by hundreds of different factors, so these samples were kept. Should the results of the model be heavily affected by outliers, then they shall be removed.