CS156 Final Project - Predicting the Number of MLB Wins

December 17, 2021

1. Problem Definition

Baseball can be much more than simply watching the game and rooting for your favorite team. Baseball is one of the most statistics-based sports, even having its own empirical and statistical analysis called Sabermetrics mainly conducted by the Society for American Baseball Research (SABR). Especially with the long history of Major League Baseball (MLB) that began in April 22, 1876 with the foundation of the National League (Noble, 2011), meaningful statistical inferences and predictions can be made with the vast amount of available data.

In this project, I use Lahman's Baseball Database to build and compare different machine learning models to **predict the number of wins of a MLB team** for a particular season, based on the team's statistics for that season. Lahman's Baseball Database consists of various MLB statistics from 1871 to 2020, including those of each team and player (batter and pitcher) per season.

Predicting the number of wins is a basic yet significant analysis because of its many implications. The prediction can be further used to estimate the final rankings and understand the dynamics among different teams. This helps teams better strategize their play and companies better evaluate potential sponsorships. The analysis is also interesting and easy to understand for the general fans as well, especially during time periods such as when a new season or postseason is about to begin.

2. Solution Specification

Predicting the number of wins of a team is a supervised regression problem. Thus, the main aim is to implement and compare various regression algorithms, using the mean absolute error (MAE) and coefficient of determination scores as the two main metrics for model evaluation. The project is executed in the following stage:

1. Data Preprocessing

The data preprocessing stage includes first manually excluding features that seem irrelevant and unnecessary to predicting the number of wins, such as the team's division, home ballpark name, number of home attendance, etc. Then, the missing values (null values) were identified and taken care of, such as removing features with too many missing values or filling some of the missing values with the feature's mean values.

2. Feature Engineering

The feature engineering stage includes transforming some features for better modeling purposes (e.g., grouping the years data into groups of 10 years) and creating meaningful new features from the raw data. For example, I created a feature named `LastYearLgWin`, which represents whether a particular team won the League Championship in the preceding season (1 if it did and 0 if it did not), assuming that a team's past performance is a relevant feature in predicting its current performance.

3. Data Analysis

The data analysis stage includes exploring and understanding the data, mainly to evaluate the predictive power of the features. For the categorical features, I visualized the density plots of the number of wins of each categorical feature. If each category has different patterns in relation with the number of wins, it has predictive power. For example, Figure 1 shows that the categories (0 or 1) in the `LastYearLgWin` feature (left) have different patterns whereas the patterns of those in the `yearGroup` feature (right) are difficult to distinguish.

[1]:
```python
from IPython import display
display.Image("/Users/soomi/Downloads/baseball_1.png")
```

[1]:

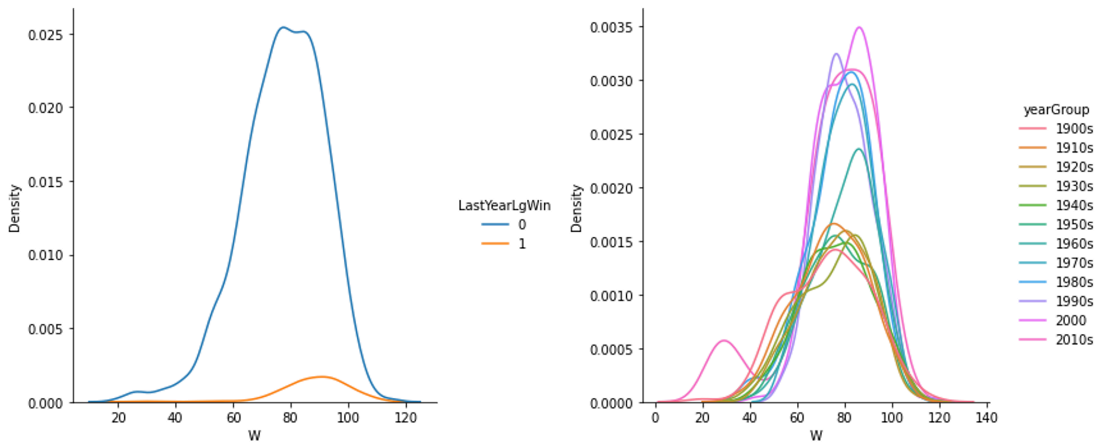


Figure 1. Density plots of number of wins based on LastYearLgWin values (left) and yearGroup values (right)

For the numerical features, I visualized and computed the correlation (Pearson's r) between each feature variable and the number of wins (output variable).

I used these results to remove the categorical and numerical features that have weak predictive power in predicting the number of wins. To avoid multicollinearity, I also checked the correlation among the feature variables and removed some features that are highly correlated with and are overlapping with other features.

The final 20 features I used to build the models include:

Variable Name	Description
`Rank`	Position in final standings
`G`	Games played
`R`	Runs scored
`H`	Hits by batters
`2B`	Doubles

Variable Name	Description
3B	Triples
HR	Homeruns by batters
BB	Walks by batters
SB	Stolen bases
RA	Opponents runs scored
ERA	Earned run average
SHO	Shutouts
SV	Saves
HA	Hits allowed
HRA	Homeruns allowed
SOA	Strikeouts by pitchers
E	Errors
DP	Double Plays
FP	Fielding percentage
LastYearLgWin	Previous year's League Champion (Y/N)

4. Modeling

Using the features listed above, I implemented the following mix of linear and non-linear regression models:

1. Linear Regression
2. Ridge Regression
3. Lasso Regression
4. Support Vector Regression (RBF kernel)
5. Support Vector Regression (linear kernel)
6. Support Vector Regression (polynomial kernel)
7. Decision Tree Regression
8. Random Forest Regression

3. Testing and Analysis

As mentioned above, the two main metrics used to evaluate and compare the models are the mean absolute error (MAE) and coefficient of determination scores (for both training and test dataset). The results are as follows:

	MAE	Training Score	Test Score
Linear	2.64431	0.94714	0.94389
Ridge	2.64432	0.94714	0.94389
Lasso	2.64918	0.94710	0.94379
RBF SVR	2.91589	0.95704	0.92704
Linear SVR	2.64738	0.94674	0.94348
Polynomial SVR	4.31301	0.87951	0.83758
Decision Tree	4.55357	0.83549	0.82285
Random Forest	3.14298	0.98764	0.91818

As expected, the **linear regression models** (linear regression, ridge regression, lasso regression, linear SVR) have similar results with an average MAE of 2.65, training score of 0.95, and test score of 0.94. Since the average number of wins in the entire dataset is 77.88, the linear models predict the number of wins to be approximately 77.88+2.65=80.53. Considering that the total number of games played is at least 160 games every year, this error is very small (although such "small" differences could highly affect the team rankings, especially during the end of the season). The linear models also do not show indications of overfitting as can be seen from the similar coefficient of determination scores for both the training and test set.

The table below also shows the feature variables with the highest coefficient values of the linear models: `R` (runs scored), `G` (games played), `Rank` (position in final standings), and `SV` (saves). It makes sense for these features to be highly correlated with the number of wins because a team is likely to have more wins if the team scores more runs (`R`), plays more games (`G`), and has higher rankings (`Rank`), and "saves" (`SV`) is only recorded for teams that won.

Features	Linear Coefficients	Ridge Coefficients	Lasso Coefficients
`R`	8.786274	8.785281	8.690665
`G`	5.911753	5.909876	5.686357
`Rank`	3.773018	3.773082	3.759902
`SV`	2.065450	2.065402	1.975719

The **non-linear regression models** (RBF SVR, polynomial SVR, decision tree, random forest) surprisingly do not show strong signs of overfitting, which may have been likely due to the flexible nature of non-linear models. The random forest model has the strongest indication of overfitting as it has the biggest difference between the training vs test scores (0.99 vs 0.92). Nonetheless, the random forest model has high coefficient of determination scores in general and has a low MAE of 3.14, which is only 0.49 higher than the average MAE of the linear regression models. The support vector regression model using the RBF kernel also has a low MAE of 2.92, which is only 0.22 higher than the average MAE of the linear regression models, and high coefficient of determination scores.

Therefore, I conclude that the RBF support vector model and the random forest model performed the best among the non-linear regression models and all the linear regression models performed very similarly and well. For this particular database and objective, the linear models performed better than non-linear models (lower MAE and higher coefficient of determination scores), implying a linear relationship is sufficient to explain the output variable (number of wins) using the extracted/pruned feature variables.

The model can be extended by using player-specific data in addition to this team-specific data. Some examples include computing and creating new features related to the average age of the players, the number of years since the players' debut (rookies vs veterans), the number of Hall of Fame nominated players per team, the number of injured players, etc.

4. References

- Lahman, S. (2020). Lahman's Baseball Database, 1871-2019, Main page. Retrieved from http://www.seanlahman.com/baseball-archive/statistics/
- MLB. (n.d.). Stolen Base (SB). Retrieved from https://www.mlb.com/glossary/standard-stats/stolen-base

- Noble, M. (2011, September 23). MLB carries on strong, 200,000 games later. Retrieved from https://www.mlb.com/news/c-25060814
- Pietro, M. (2020, May 18). Machine Learning with Python: Regression (complete tutorial). Retrieved from https://towardsdatascience.com/machine-learning-with-python-regression-complete-tutorial-47268e546cea

5. Appendices

The dataset can be downloaded here. The gist of the codes can be found here.

Environmental Setup

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats as sts
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score
from sklearn import linear_model
from sklearn.svm import SVR
from sklearn import tree
from sklearn.ensemble import RandomForestRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV
```

Data Preprocessing

Import the Data

[2]:
```
teams = pd.read_csv('/Users/soomi/Downloads/baseballdatabank-master/core/Teams.
↪csv')
teams.head(5)
```

[2]:
```
   yearID lgID teamID franchID divID  Rank   G  Ghome   W   L  …  DP     FP  \
0    1871  NaN    BS1      BNA   NaN     3  31    NaN  20  10  …  24  0.834
1    1871  NaN    CH1      CNA   NaN     2  28    NaN  19   9  …  16  0.829
2    1871  NaN    CL1      CFC   NaN     8  29    NaN  10  19  …  15  0.818
3    1871  NaN    FW1      KEK   NaN     7  19    NaN   7  12  …   8  0.803
4    1871  NaN    NY2      NNA   NaN     5  33    NaN  16  17  …  14  0.840

                      name                          park  attendance  BPF  \
0     Boston Red Stockings           South End Grounds I         NaN  103
```

```
1  Chicago White Stockings       Union Base-Ball Grounds         NaN  104
2   Cleveland Forest Citys  National Association Grounds         NaN   96
3     Fort Wayne Kekiongas                Hamilton Field         NaN  101
4         New York Mutuals      Union Grounds (Brooklyn)         NaN   90

   PPF  teamIDBR  teamIDlahman45  teamIDretro
0   98       BOS             BS1          BS1
1  102       CHI             CH1          CH1
2  100       CLE             CL1          CL1
3  107       KEK             FW1          FW1
4   88       NYU             NY2          NY2

[5 rows x 48 columns]
```

Exclude Unnecessary Features

First, I performed manual feature selection by dropping columns that seemed irrelevant in predicting the number of wins. The columns I dropped include:

- `lgID`: league
- `franchID`: franchise
- `divID`: team's division
- `Ghome`: games played at home
- `L`: number of losses
- `name`: team's full name
- `park`: name of team's home ballpark
- `attendance`: home attendance total
- `BPF`: three-year park factor for batters
- `PPF`: three-year park factor for pitchers
- `teamIDBR`: team ID used by Baseball Reference website
- `teamIDlahman45`: team ID used in Lahman database version 4.5
- `teamIDretro`: team ID used by Retrosheet

[3]:
```
# drop unnecessary features
cols_to_drop = ['lgID', 'franchID', 'divID', 'Ghome', 'L', 'name', 'park',␣
 ↪'attendance',
                'BPF', 'PPF', 'teamIDBR', 'teamIDlahman45', 'teamIDretro']

teams.drop(cols_to_drop, inplace=True, axis=1)
```

Examine Categorical Variables

The main categorical variables are `DivWin`, `WCWin`, `LgWin`, and `WSWin`, which represent the following:

- `DivWin`: Division Winner (Y or N)
- `WCWin`: Wild Card Winner (Y or N)
- `LgWin`: League Champion(Y or N)
- `WSWin`: World Series Winner (Y or N)

Let's first examine whether any of these columns include null values (especially since the MLB rules of championships and series are likely to have changed over time).

[4]:
```
print("How many null values are there?")
print("- DivWin: %.f (%.2f%%)" % (teams['DivWin'].isnull().sum(axis=0),
                                  (teams['DivWin'].isnull().sum(axis=0)␣
 ↪/ len(teams))*100))
print("- WCWin: %.f (%.2f%%)" % (teams['WCWin'].isnull().sum(axis=0),
                                  (teams['WCWin'].isnull().sum(axis=0) /
 ↪ len(teams))*100))
print("- LgWin: %.f (%.2f%%)" % (teams['LgWin'].isnull().sum(axis=0),
                                  (teams['LgWin'].isnull().sum(axis=0) /
 ↪ len(teams))*100))
print("- WSWin: %.f (%.2f%%)" % (teams['WSWin'].isnull().sum(axis=0),
                                  (teams['WSWin'].isnull().sum(axis=0) /
 ↪ len(teams))*100))
print("\n Total number of data:", len(teams))
```

```
How many null values are there?
- DivWin: 1545 (52.28%)
- WCWin: 2181 (73.81%)
- LgWin: 28 (0.95%)
- WSWin: 357 (12.08%)

 Total number of data: 2955
```

[5]:
```
# check which season has null values for 'LgWin'
set(teams[teams['LgWin'].isnull()]['yearID'])
```

[5]:
```
{1994}
```

The `LgWin` column has very few null values, which seem to be only from the 1994 season. Thus, I will keep the `LgWin` column while removing the 1994 season entirely. I will also remove the remaining winner-related columns (`DivWin`, `WCWin`, `WSWin`). Instead of filling the null values with zeros or other values (e.g., median, mean), I decided to remove them entirely because I want the data to be in the same condition as much as possible. In a sports context, I do not want an inconsistent game environment such that, for example, some seasons have played the World Series whereas others have not.

[6]:
```
# remove 'DivWin', 'WCWin', 'WSWin' because they have too many null values
teams.drop(['DivWin', 'WCWin', 'WSWin'], inplace=True, axis=1)

# remove the 1994 season because its 'LgWin' data is null
teams = teams[teams.yearID != 1994]
```

[7]:
```
# confirm that the 'LgWin' column does not have null values
print(teams['LgWin'].isnull().sum(axis=0))
```

0

Manage Null Values

[8]:
```
# which columns contain null values?
teams.columns[teams.isnull().any()].tolist()
```

[8]: ['BB', 'SO', 'SB', 'CS', 'HBP', 'SF']

[9]:
```
print("How many null values are there?")
print("- BB (walks by batters): %.f (%.2f%%)" % (teams['BB'].isnull().sum(),
                                                   (teams['BB'].isnull().
 ↪sum(axis=0) / len(teams))*100))
print("- SO (strikeouts by batters): %.f (%.2f%%)" % (teams['SO'].isnull().
 ↪sum(),
                                                   (teams['SO'].isnull().
 ↪sum(axis=0) / len(teams))*100))
print("- SB (stolen bases): %.f (%.2f%%)" % (teams['SB'].isnull().sum(),
                                                   (teams['SB'].isnull().
 ↪sum(axis=0) / len(teams))*100))
print("- CS (caught stealing): %.f (%.2f%%)" % (teams['CS'].isnull().sum(),
                                                   (teams['CS'].isnull().
 ↪sum(axis=0) / len(teams))*100))
print("- HBP (batters hit by pitch): %.f (%.2f%%)" % (teams['HBP'].isnull().
 ↪sum(),
                                                   (teams['HBP'].isnull().
 ↪sum(axis=0) / len(teams))*100))
print("- SF (sacrifice flies): %.f (%.2f%%)" % (teams['SF'].isnull().sum(),
                                                   (teams['SF'].isnull().
 ↪sum(axis=0) / len(teams))*100))
print("\n Total number of data:", len(teams))
```

```
How many null values are there?
- BB (walks by batters): 1 (0.03%)
- SO (strikeouts by batters): 16 (0.55%)
- SB (stolen bases): 126 (4.30%)
- CS (caught stealing): 832 (28.43%)
- HBP (batters hit by pitch): 1158 (39.56%)
- SF (sacrifice flies): 1541 (52.65%)

 Total number of data: 2927
```

Considering that there are 2927 rows in total, the CS, HBP, and SF columns have too many null values to be covered by other values, such as median, mean, or zeros. Thus, I decided to entirely remove these features.

[10]:
```
# drop columns with too many null values
teams.drop(['CS', 'HBP', 'SF'], inplace=True, axis=1)
```

For the remaining columns with null values, I first checked the years that contain null values. The SB (Stolen Bases) column especially had a lot of null values from 1872 to 1885. After research, I found that the modern steal MLB rule was implemented in 1898, and hence, the many null values during this time period. Therefore, I decided to eliminate the data before 1898 to fully take into account the SB feature. This also takes care of the null value in the BB column because its null value was from 1873.

The SO column has very few null values, so I will fill the missing values with the mean value of the corresponding season. For example, the missing SO values (strikeouts by batters) of 1911 will be replaced with the mean SO value of 1911.

[11]:
```
# check the years that contain null values for 'BB', 'SO', 'SB'
print("Years that contain null values for BB:", set(teams[teams['BB'].
 ↪isnull()]['yearID']))
print("Years that contain null values for SO:", set(teams[teams['SO'].
 ↪isnull()]['yearID']))
print("Years that contain null values for SB:", set(teams[teams['SB'].
 ↪isnull()]['yearID']))
```

```
Years that contain null values for BB: {1873}
Years that contain null values for SO: {1912, 1911}
Years that contain null values for SB: {1872, 1873, 1876, 1877, 1878, 1879,
1880, 1881, 1882, 1883, 1884, 1885}
```

[12]:
```
# filter rows for which the year is greater than or equal to 1898
teams = teams[teams['yearID'] >= 1898]
```

[13]:
```
# compute the mean SO value of 1911 and 1912 (years that contained null values)
SO_1911_mean = np.nanmean(teams[teams['yearID']==1911]['SO'])
SO_1912_mean = np.nanmean(teams[teams['yearID']==1912]['SO'])

print("Mean SO value in 1911:", SO_1911_mean)
print("Mean SO value in 1912:", SO_1912_mean)
```

```
Mean SO value in 1911: 599.75
Mean SO value in 1912: 578.375
```

[14]:
```
# replace 'SO' null values with the mean 'SO' value of the corresponding year
teams.loc[(teams.yearID == 1911) & (teams.SO.isna()), 'SO'] = SO_1911_mean
teams.loc[(teams.yearID == 1912) & (teams.SO.isna()), 'SO'] = SO_1912_mean
```

[15]:
```
# check that all null values have been handled
teams.columns[teams.isnull().any()].tolist()
```

[15]:
```
[]
```

Feature Engineering

I transformed and added some features that seem relevant to predicting the number of wins.

Transform `Rank` Feature

I converted the `Rank` feature (position in final standings) into negative values so that the correlation between the rank and number of wins is more intuitive (the bigger the rank value, the better the team performed).

[16]:
```
# the bigger the rank, the better
teams.loc[:, 'Rank'] = [-1*rank for rank in teams.loc[:, 'Rank']]
```

Transform `yearID` Feature

I categorized the `yearID` feature values into groups of 10 years and added them as a new column named `yearGroup`. The smallest year value is 1898 and the largest year value is 2020. The years 1898 and 1899 are grouped as '1900s' and the year 2020 is grouped as '2010s'.

[17]:
```
print(min(teams['yearID']))
print(max(teams['yearID']))
```

```
1898
2020
```

[18]:
```
# group every 10 years
year_group = []
for year in teams['yearID']:
    if year < 1910:
        year_group.append('1900s')
    elif year >= 1910 and year < 1920:
        year_group.append('1910s')
    elif year >= 1920 and year < 1930:
        year_group.append('1920s')
    elif year >= 1930 and year < 1940:
        year_group.append('1930s')
    elif year >= 1940 and year < 1950:
        year_group.append('1940s')
    elif year >= 1950 and year < 1960:
        year_group.append('1950s')
    elif year >= 1960 and year < 1970:
        year_group.append('1960s')
    elif year >= 1970 and year < 1980:
        year_group.append('1970s')
    elif year >= 1980 and year < 1990:
        year_group.append('1980s')
    elif year >= 1990 and year < 2000:
        year_group.append('1990s')
    elif year >= 2000 and year < 2010:
```

```
        year_group.append('2000')
    elif year >= 2010 and year <= 2020:
        year_group.append('2010s')
```

[19]:
```
teams['yearGroup'] = year_group
```

Add LastYearLgWin Feature

An important factor in predicting a team's current performance is its recent past performance and record. I will use the `LgWin` feature to create a new attribute named `LastYearLgWin` that represents whether the team was the League Champion in the preceding season (1 if it was and 0 if it was not).

[20]:
```
# find who won the League Championship each year
yearly_LgWinners = teams.loc[teams['LgWin'] == 'Y']
yearly_LgWinners = yearly_LgWinners[['yearID', 'teamID']]
yearly_LgWinners.head(5)
```

[20]:

	yearID	teamID
353	1898	BSN
364	1899	BRO
375	1900	BRO
387	1901	CHA
396	1901	PIT

[21]:
```
years = list(set(teams['yearID']))

# the first year is filled with 0s because it does not have former year to␣
 ↪compare with
first_year_teams = len(teams.loc[teams['yearID'] == years[0]])
total_LgWin = [[0]*first_year_teams]

# go through each team that played each year
# add 1 if team's name is in last year's League Championship list; else, add 0
for i in range(len(years)-1):
    current_year_teams = teams.loc[teams['yearID'] == years[i+1]]
    last_year_LgWinners = yearly_LgWinners.loc[yearly_LgWinners['yearID'] ==␣
 ↪years[i]]

    yearly_LgWin = []

    for t in range(len(current_year_teams['teamID'])):
        if current_year_teams['teamID'].iloc[t] ==␣
 ↪last_year_LgWinners['teamID'].iloc[0]:
            yearly_LgWin.append(1)
        else:
            yearly_LgWin.append(0)
```

```
    total_LgWin.append(yearly_LgWin)

print(len(total_LgWin))
print(len(total_LgWin)==len(years))
```

```
122
True
```

[22]:

```
# unpack lists in list as one big list to be added as a new column
single_total_LgWin = []
for lst in total_LgWin:
    single_total_LgWin += lst
```

[23]:

```
# add the new 'LastYearLgWin' column and remove the 'LgWin' column
teams['LastYearLgWin'] = single_total_LgWin
teams.drop('LgWin', inplace=True, axis=1)

teams.head(5)
```

[23]:

```
     yearID teamID  Rank    G    W    R    AB     H   2B   3B  …  IPouts  \
351    1898    BLN    -2  153   96  933  5242  1584  154   77  …    3969
352    1898    BRO   -10  149   54  638  5126  1314  156   66  …    3896
353    1898    BSN    -1  152  102  872  5276  1531  190   55  …    4020
354    1898    CHN    -4  152   85  828  5219  1431  175   84  …    4028
355    1898    CIN    -3  157   92  831  5334  1448  207  101  …    4156

       HA  HRA  BBA  SOA    E   DP     FP  yearGroup  LastYearLgWin
351  1236   17  400  422  326  105  0.947      1900s              0
352  1446   34  476  294  334  125  0.947      1900s              0
353  1186   37  470  432  310  102  0.950      1900s              0
354  1357   17  364  323  412  149  0.936      1900s              0
355  1484   16  449  294  325  128  0.950      1900s              0

[5 rows x 30 columns]
```

Data Analysis

Examine the Output Variable: Number of Wins

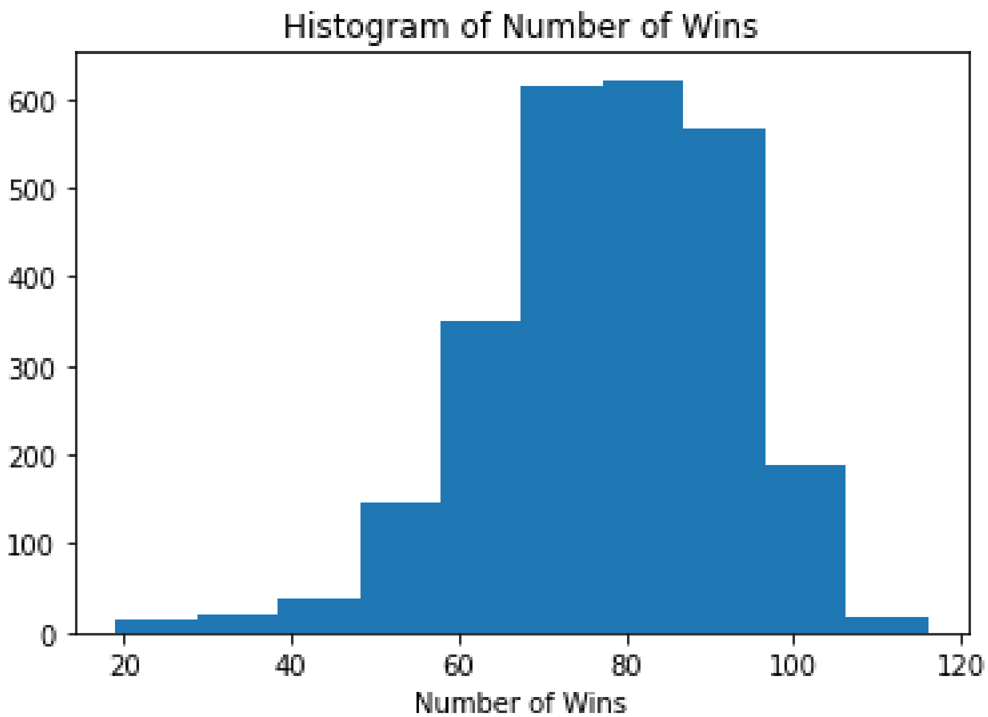
The `W` column (number of wins) is the output variable of interest. Let's visualize and examine its distribution:

[24]:

```
plt.hist(teams['W'])
plt.xlabel('Number of Wins')
plt.title('Histogram of Number of Wins')
```

```python
print("Average number of wins: %.3f" % np.mean(teams['W']))
print("Median number of wins: %.3f" % np.median(teams['W']))
print("Minimum number of wins: %.3f" % np.min(teams['W']))
print("Maximum number of wins: %.3f" % np.max(teams['W']))
```

```
Average number of wins: 77.880
Median number of wins: 79.000
Minimum number of wins: 19.000
Maximum number of wins: 116.000
```



Evaluate the Categorical Features

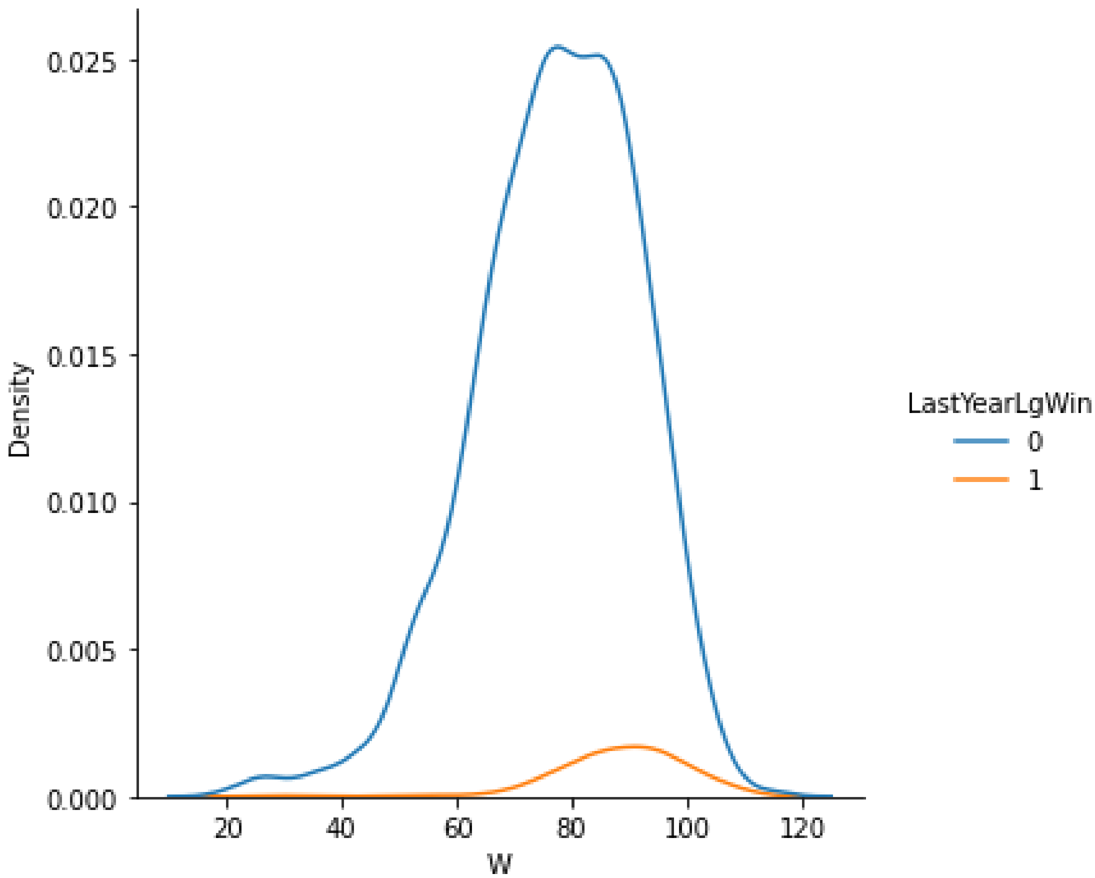
To evaluate how predictive each categorical feature is in predicting the number of wins, I plotted the `W` densities for each categorical feature. The two main categorical features of the dataset are `LastYearLgWin` and `yearGroup`. From these density plots, I made the following conclusions:

- `LastYearLgWin`: The difference between whether or not a team won the previous year's League Championship indeed has differing results in the number of wins. Thus, I will keep the `LastYearLgWin` feature as it seems to have predictive power in predicting the number of wins.
- `yearGroup`: The `W` density plots of each year group turned out to look quite similar. Although the earlier years (around 1900s to 1950s) and the later years (around 1960s to 2010s) look like they form two groups with similar densities, I verified that this is simply due to the difference in the number of total games played in the earlier years versus later years. Thus, I will remove

the `yearGroup` feature as it does not seem to have significant predictive power in predicting the number of wins.

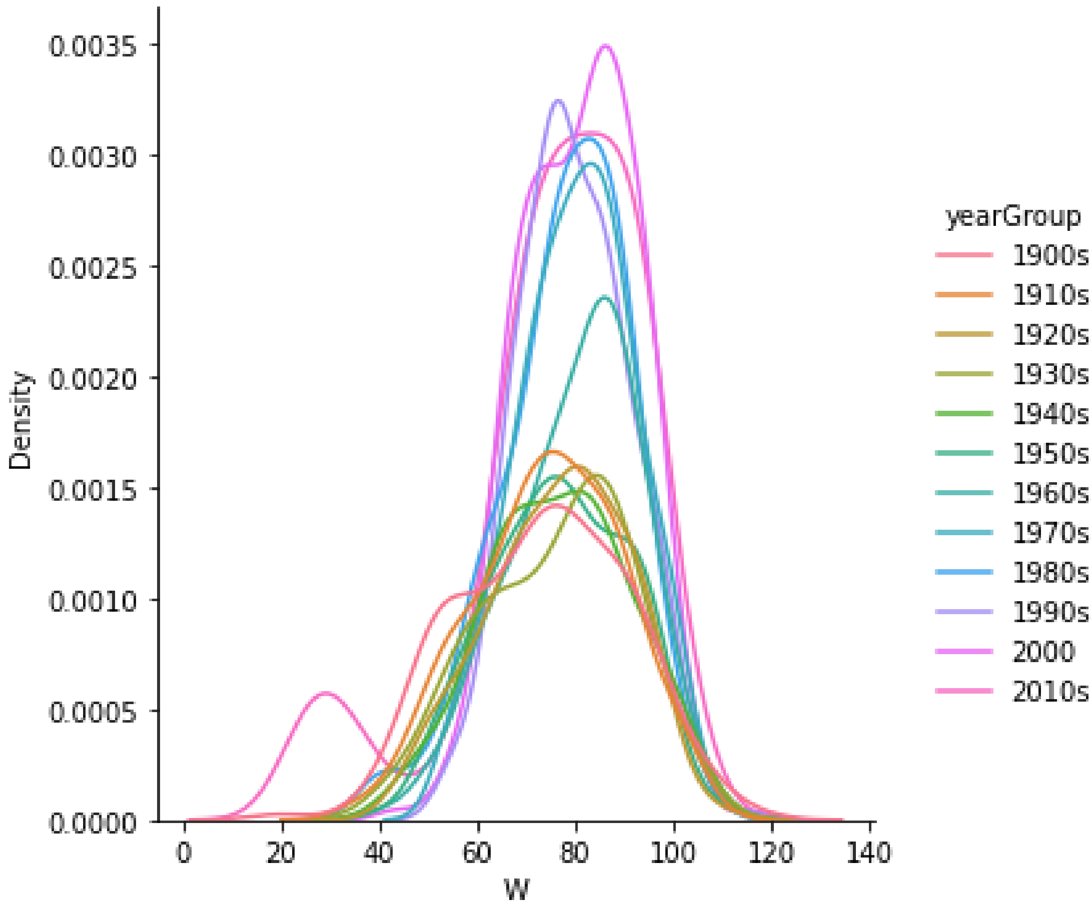
[25]:
```python
# density plot for 'LastYearLgWin'
sns.displot(data=teams, x='W', hue="LastYearLgWin", kind="kde")
```

[25]: <seaborn.axisgrid.FacetGrid at 0x7f9fa7765160>



[26]:
```python
# density plot for 'yearGroup'
sns.displot(data=teams, x='W', hue="yearGroup", kind="kde")
```

[26]: <seaborn.axisgrid.FacetGrid at 0x7f9fa9791908>



[27]:
```python
# verified that there are more games played in later years than earlier years
teams.groupby(['yearGroup'])['G'].count()
```

[27]:
```
yearGroup
1900s    176
1910s    176
1920s    160
1930s    160
1940s    160
1950s    160
1960s    198
1970s    246
1980s    260
1990s    250
2000     300
2010s    330
Name: G, dtype: int64
```

[28]:
```
# remove the 'yearGroup' feature
teams.drop(['yearGroup'], inplace=True, axis=1)
```

Evaluate the Numerical Features

To evaluate how predictive each numerical feature is in predicting the number of wins, I visualized and computed the correlation (Pearson's r) between each feature variable and the output variable (`W`). The computed Pearson correlation coefficient and p-value will be used to keep features with significant predictive power and possibly drop features with non-significant predictive power. The following two functions are derived from this tutorial.

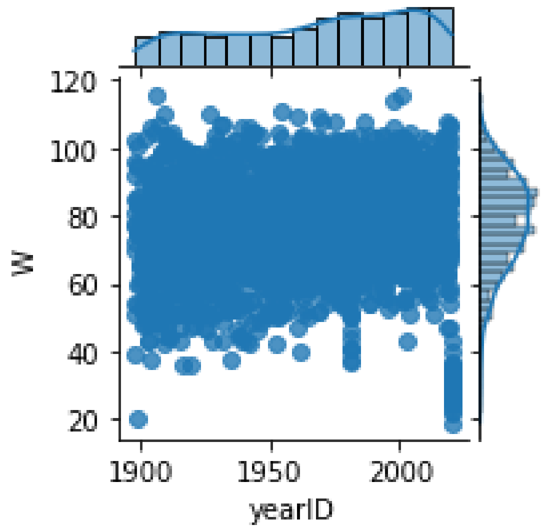
[29]:
```
def pearson_coeff(dt, feature, output):
    coeff, p = sts.pearsonr(dt[feature], dt[output])
    coeff, p = coeff, p
    conclusion = "Significant" if p < 0.05 else "Non-Significant"
    print("Pearson's r for %s: %.3f | %s (p-value: %.3f)" % (feature, coeff,␣
 ↪conclusion, p))
```

[30]:
```
def numerical_scatterplot(dt, feature, output):
    figsize=(3, 3)
    sns.jointplot(x=feature, y=output, data=dt, kind='reg',␣
 ↪height=int((figsize[0]+figsize[1])/2) )
    plt.show()
    pearson_coeff(dt, feature, output)
```

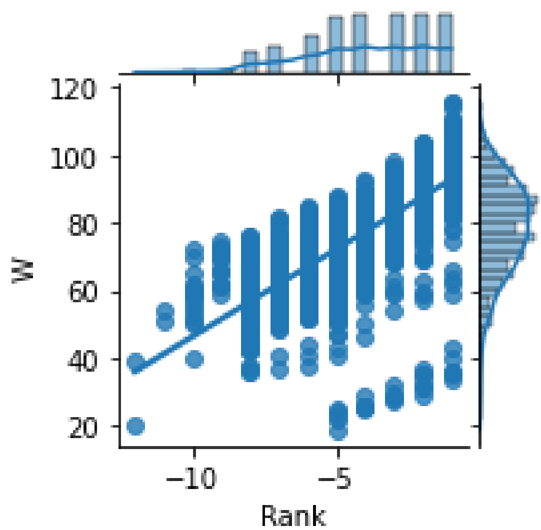
[31]:
```
numerical_feats = list(teams.columns)
numerical_feats.remove('teamID') # remove because it is categorical
numerical_feats.remove('LastYearLgWin') # remove because it is categorical
numerical_feats.remove('W') # remove because it is the output variable itself
print(numerical_feats)
```

```
['yearID', 'Rank', 'G', 'R', 'AB', 'H', '2B', '3B', 'HR', 'BB', 'SO', 'SB',
'RA', 'ER', 'ERA', 'CG', 'SHO', 'SV', 'IPouts', 'HA', 'HRA', 'BBA', 'SOA', 'E',
'DP', 'FP']
```

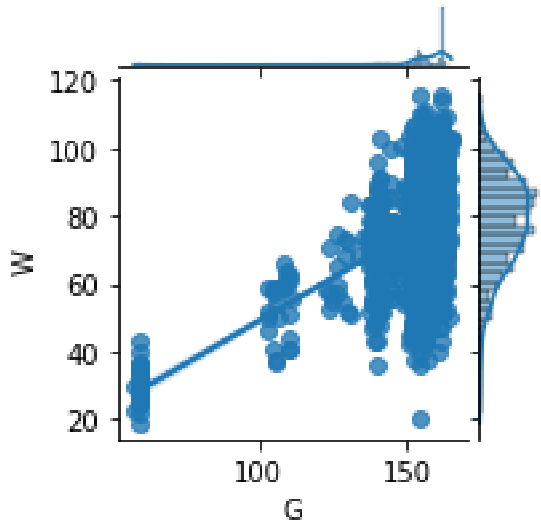
[32]:
```
for f in numerical_feats:
    numerical_scatterplot(teams, f, 'W')
```



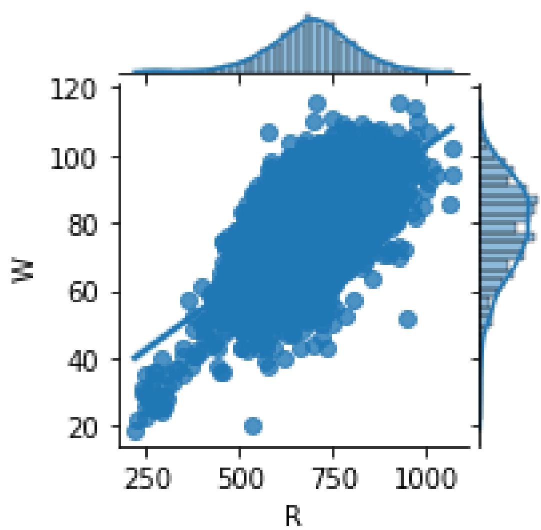
Pearson's r for yearID: 0.096 | Significant (p-value: 0.000)



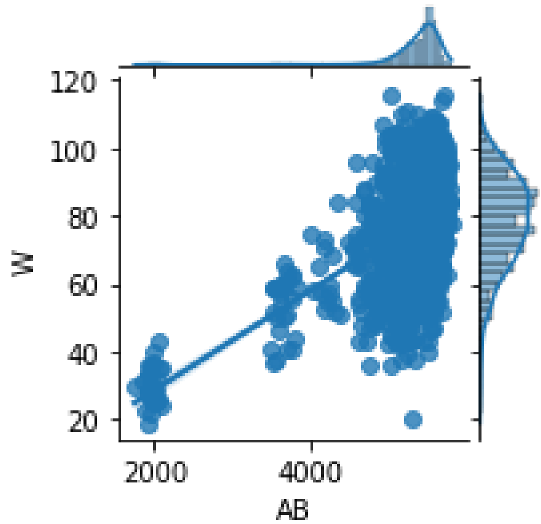
Pearson's r for Rank: 0.773 | Significant (p-value: 0.000)



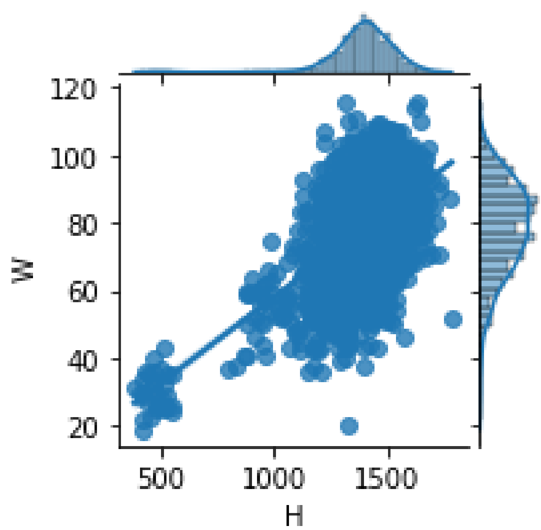

```
Pearson's r for G: 0.458 | Significant (p-value: 0.000)
```



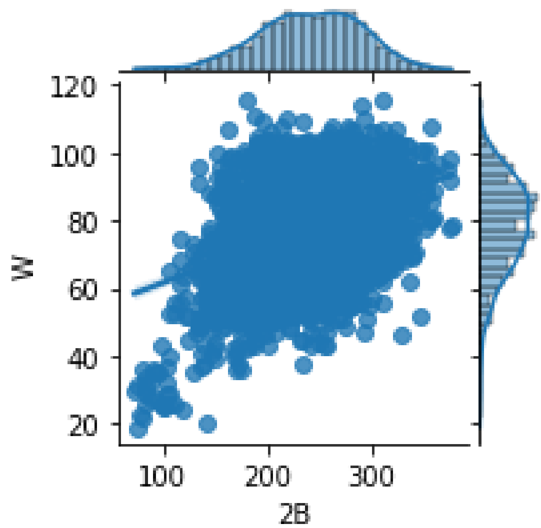

```
Pearson's r for R: 0.644 | Significant (p-value: 0.000)
```



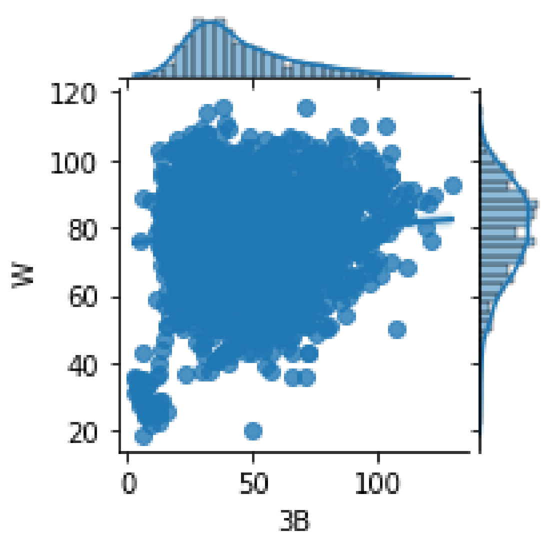
Pearson's r for AB: 0.476 | Significant (p-value: 0.000)



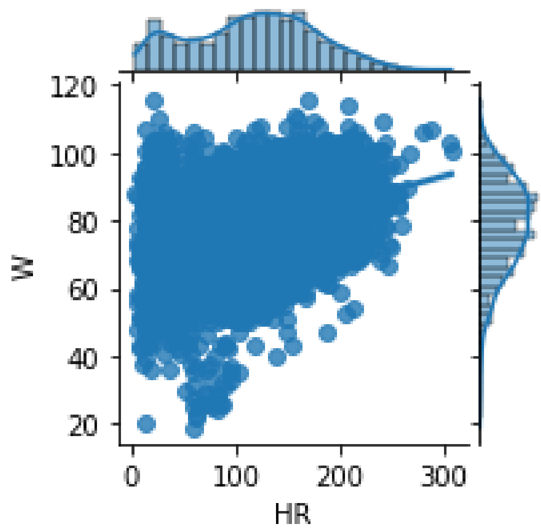
Pearson's r for H: 0.550 | Significant (p-value: 0.000)



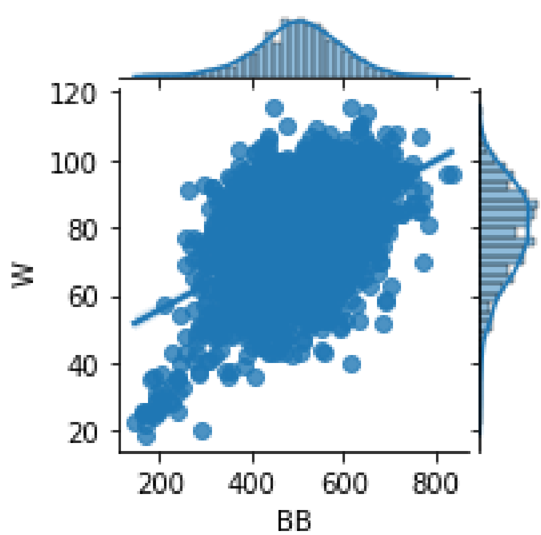

```
Pearson's r for 2B: 0.400 | Significant (p-value: 0.000)
```



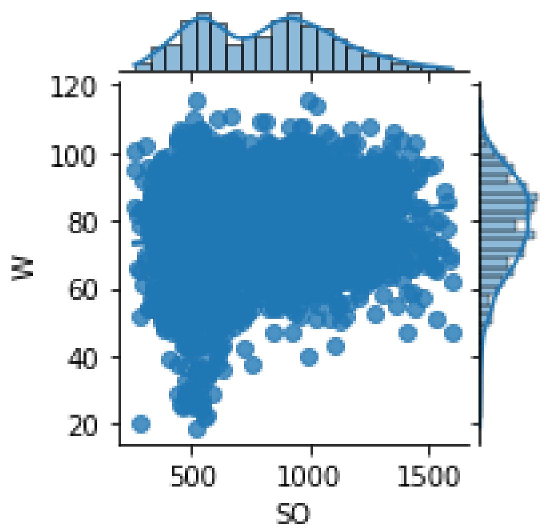

```
Pearson's r for 3B: 0.078 | Significant (p-value: 0.000)
```



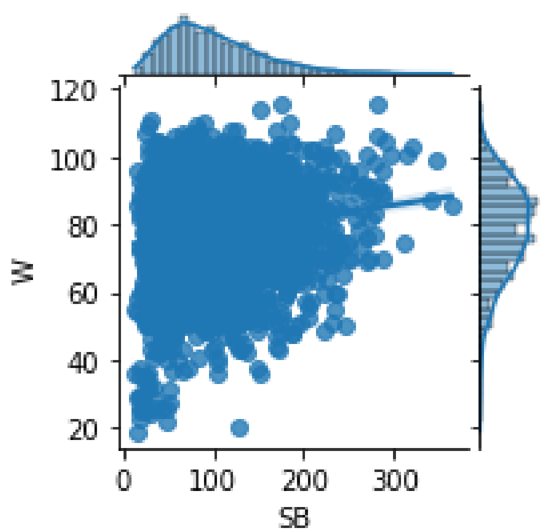
Pearson's r for HR: 0.344 | Significant (p-value: 0.000)



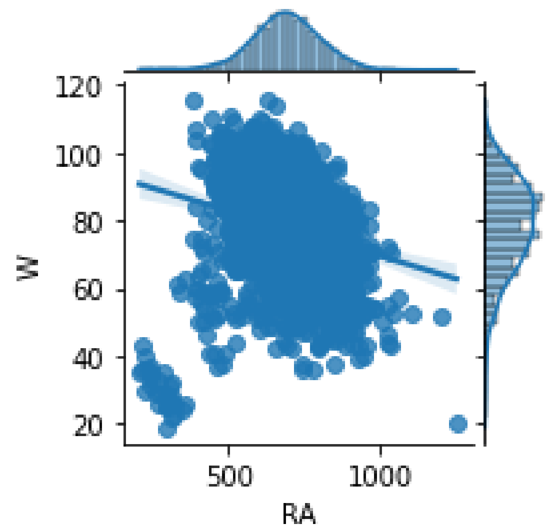
Pearson's r for BB: 0.481 | Significant (p-value: 0.000)



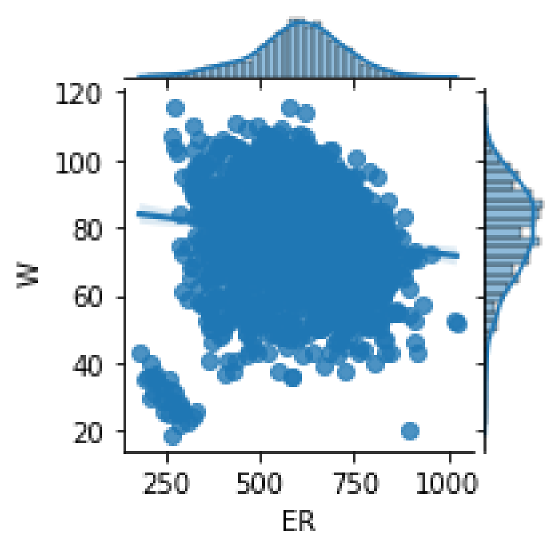
Pearson's r for SO: 0.161 | Significant (p-value: 0.000)



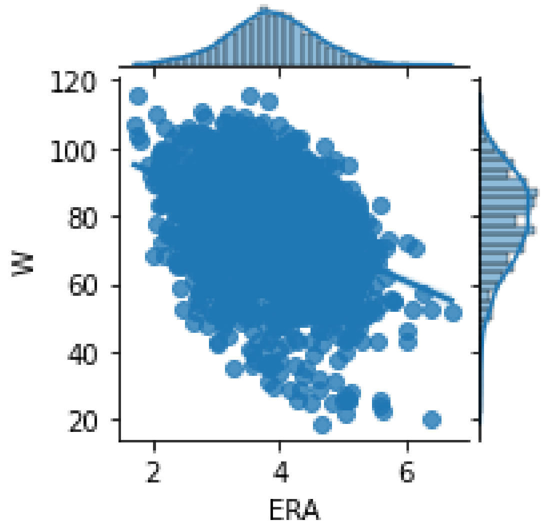
Pearson's r for SB: 0.146 | Significant (p-value: 0.000)



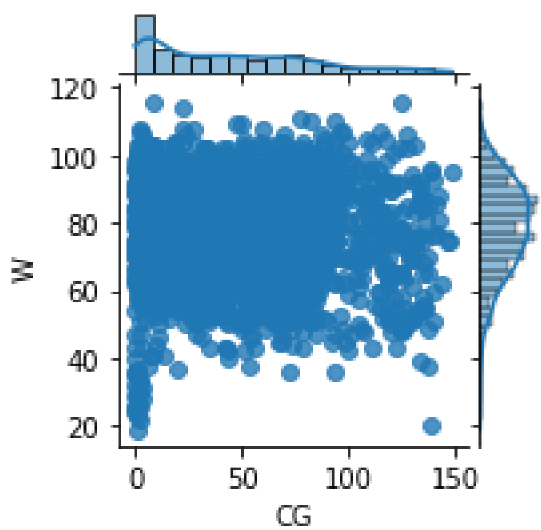
Pearson's r for RA: -0.219 | Significant (p-value: 0.000)



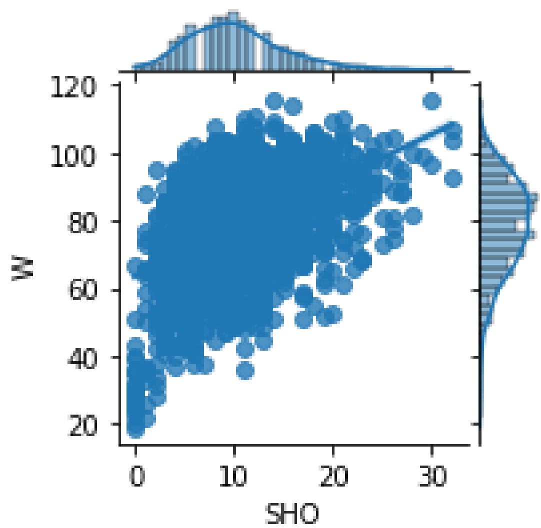
Pearson's r for ER: -0.122 | Significant (p-value: 0.000)



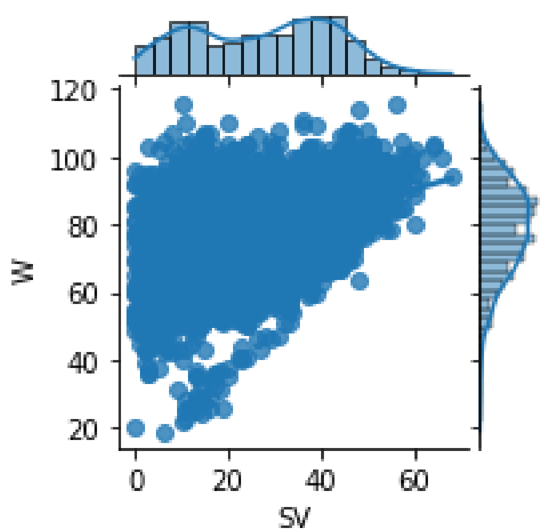

```
Pearson's r for ERA: -0.390 | Significant (p-value: 0.000)
```



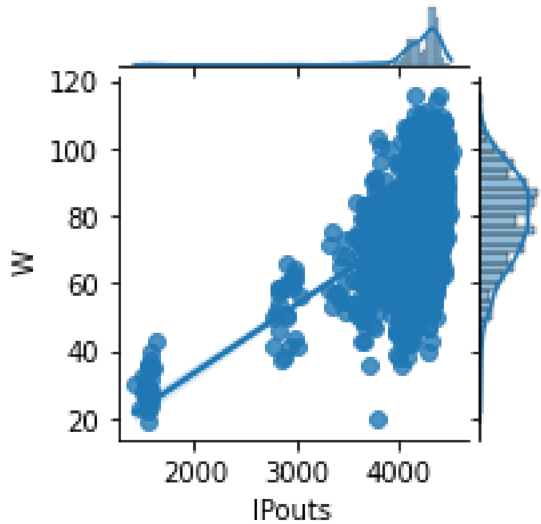

```
Pearson's r for CG: -0.027 | Non-Significant (p-value: 0.163)
```



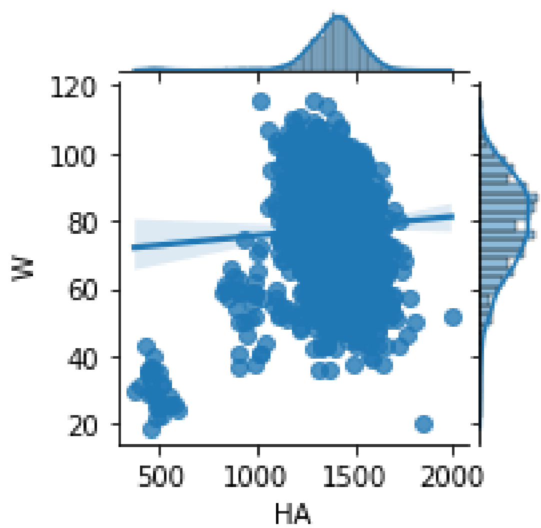
Pearson's r for SHO: 0.476 | Significant (p-value: 0.000)



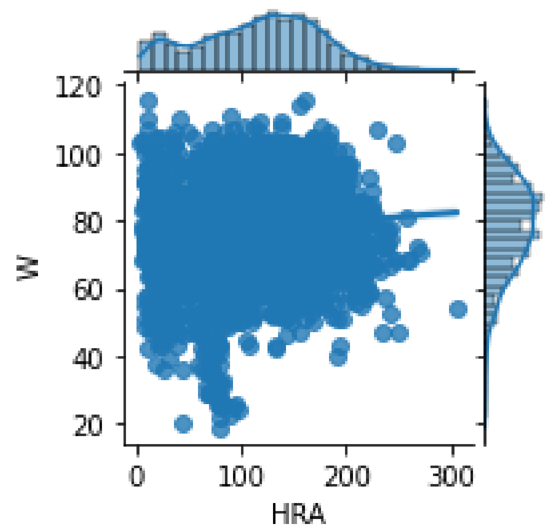
Pearson's r for SV: 0.401 | Significant (p-value: 0.000)



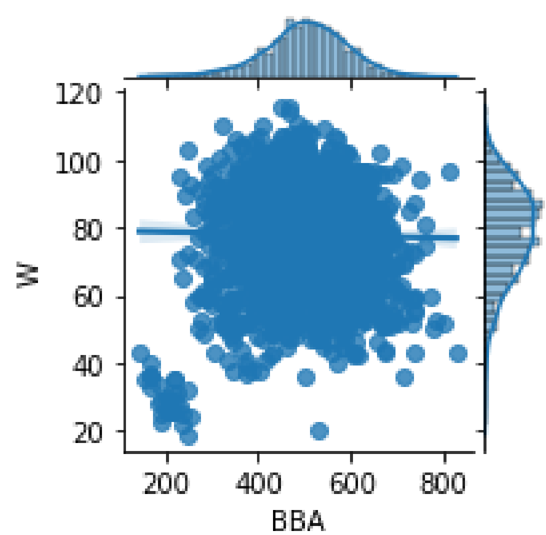
Pearson's r for IPouts: 0.502 | Significant (p-value: 0.000)



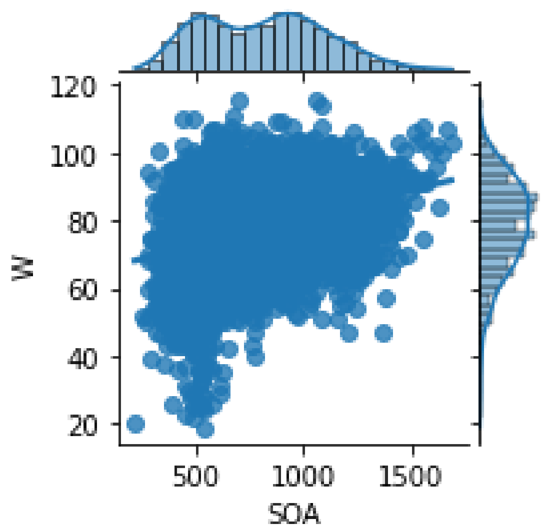
Pearson's r for HA: 0.061 | Significant (p-value: 0.002)



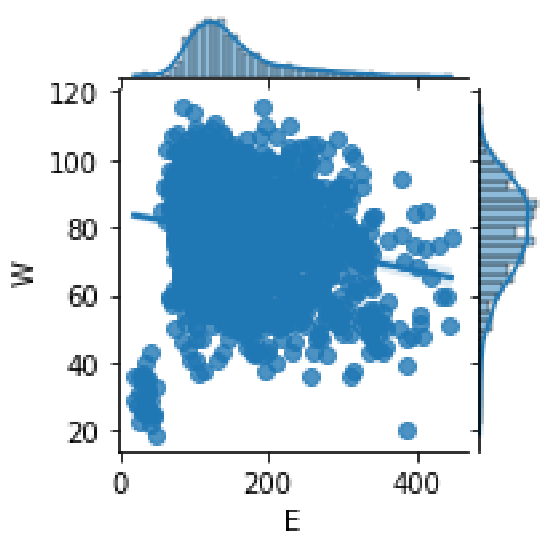
Pearson's r for HRA: 0.094 | Significant (p-value: 0.000)



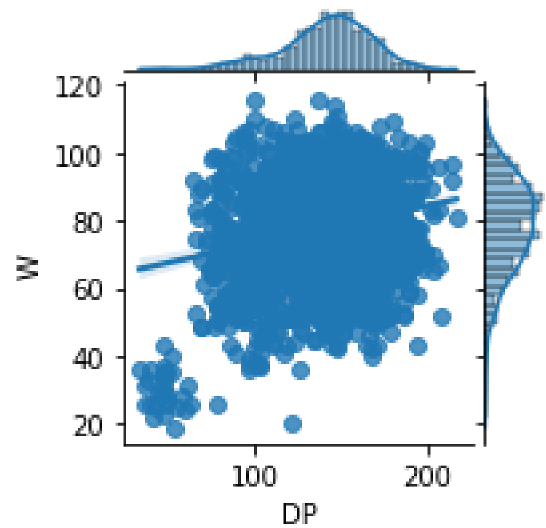
Pearson's r for BBA: -0.019 | Non-Significant (p-value: 0.337)



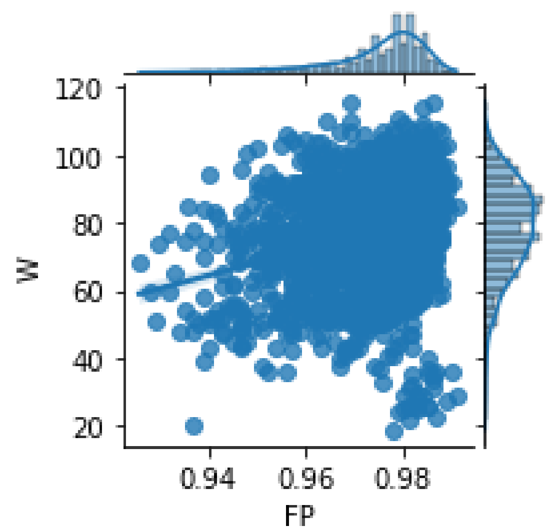

```
Pearson's r for SOA: 0.317 | Significant (p-value: 0.000)
```




```
Pearson's r for E: -0.188 | Significant (p-value: 0.000)
```




```
Pearson's r for DP: 0.211 | Significant (p-value: 0.000)
```




```
Pearson's r for FP: 0.260 | Significant (p-value: 0.000)
```

From these visualizations and results, I made the following conclusions:

- `yearID`: Even though this feature was concluded "Significant," I will remove it for the same reason I removed the `yearGroup` categorical variable; the correlation plot indicates that there is lack of predictive power in predicting the number of wins.
- `CG`, `BBA`: I will remove these features because they have been concluded "Non-Significant."
- The rest of the features will be kept because they have been concluded "Significant."

[33]:
```
# remove features with weak predictive power
teams.drop(['yearID', 'CG', 'BBA'], inplace=True, axis=1)
```

Modeling

Scale and Split the Data

Before beginning the modeling, I checked whether there is correlation among the feature variables, which is something we want to avoid in case of multicollinearity. To be more precise in computing the correlations, I first scaled the feature variable data using `StandardScaler()` and then plotted a correlation heatmap.

[34]:
```
# remove 'teamID' and 'W'
x_attributes = list(teams.columns)
x_attributes.remove('teamID')
x_attributes.remove('W')

print(x_attributes)
print("\nNumber of features to check:", len(x_attributes))
```

```
['Rank', 'G', 'R', 'AB', 'H', '2B', '3B', 'HR', 'BB', 'SO', 'SB', 'RA', 'ER',
'ERA', 'SHO', 'SV', 'IPouts', 'HA', 'HRA', 'SOA', 'E', 'DP', 'FP',
'LastYearLgWin']

Number of features to check: 24
```

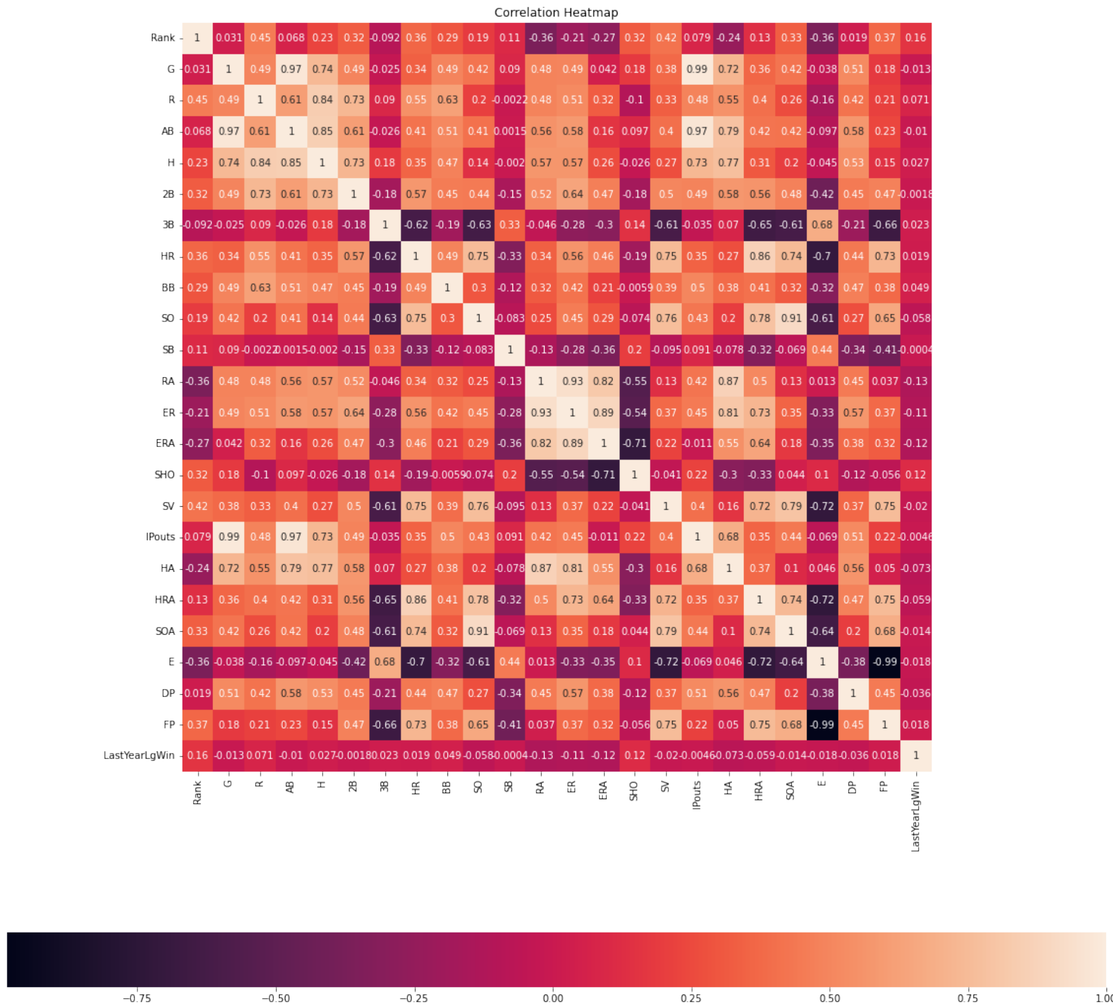
[35]:
```
# scale the feature values
scaler = StandardScaler()

X = teams[x_attributes]
X = scaler.fit_transform(X)
```

[36]:
```
check_corr = pd.DataFrame(X, columns = x_attributes)

plt.figure(figsize=(20, 20))
sns.heatmap(check_corr.corr(), cbar_kws= {'orientation': 'horizontal'},␣
↪annot=True, square=True)
plt.title("Correlation Heatmap")
```

[36]:
```
Text(0.5, 1.0, 'Correlation Heatmap')
```



The following are pairs of feature variables with high correlation (>=0.9):

- `G` (games played) & `AB` (at bats) = 0.97
- `G` (games played) & `IPOuts` (innings pitched x 3) = 0.99
- `AB` (at bats) & `IPOuts` (outs pitched) = 0.97
- `SO` (strikeouts by batters) & `SOA` (strikeouts by pitchers) = 0.91
- `RA` (opponents runs scored) & `ER` (earned runs allowed) = 0.93

There seems to be high correlation between `G`, `AB`, and `IPOuts` altogether, which makes sense because there are obviously more battings and pitchings if there are more games (vice versa). Thus, I will remove `AB` and `IPOuts` while keeping `G` because `G` seems to more general than `AB` and `IPOuts`, which are respectively batter-specific and pitcher-specific.

There is also high correlation between `SO` and `SOA`. However, I do not completely understand why these two are correlated as I interpreted `SO` as the number of times the batters got striked out by the opponent pitchers and `SOA` as the number of times the pitcher striked out the opponent batters. I could not find further explanations regarding these two variables to check whether my interpretation is correct or not. Nonetheless, the two are correlated, meaning keeping both can result in multicollinearity. Since the reason behind their high correlation is obscure, I will keep the feature that has a higher Pearson coefficient with the `W` output variable. Thus, I will remove `SO` (Pearson's r = 0.161) and keep `SOA` (Pearson's r = 0.317).

The high correlation between `RA` and `ER` is understandable because they represent very similar statistics. Thus, I will keep `RA` and remove `ER` because `RA` seems to be a more general statistics.

[37]:
```
# filtered feature variables after checking correlation
new_x_attributes = x_attributes
new_x_attributes.remove('AB')
new_x_attributes.remove('IPouts')
new_x_attributes.remove('SO')
new_x_attributes.remove('ER')
```

[38]:
```
print(new_x_attributes)
print("\nNumber of features to use:", len(new_x_attributes))
```

```
['Rank', 'G', 'R', 'H', '2B', '3B', 'HR', 'BB', 'SB', 'RA', 'ERA', 'SHO', 'SV',
'HA', 'HRA', 'SOA', 'E', 'DP', 'FP', 'LastYearLgWin']

Number of features to use: 20
```

After scaling the feature data, I split the data into training vs test sets with a 70/30 ratio.

[39]:
```
X = teams[new_x_attributes]
X = scaler.fit_transform(X)
y = teams['W']

# training: 70% vs test: 30%
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
↪random_state=42)
```

[40]:
```
print(len(X_train))
print(len(X_test))
print(len(y_train))
print(len(y_test))
```

```
1803
773
1803
773
```

Model Design

1. Linear Regression

[41]:
```
lr = LinearRegression().fit(X_train, y_train)
lr_y_pred = lr.predict(X_test)
```

[42]:
```
# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test, lr_y_pred))
# The coefficient of determination score
print("Training set score: %.5f" % lr.score(X_train,y_train))
print("Test set score: %.5f" % lr.score(X_test,y_test))
```

```
Mean absolute error: 2.64431
Training set score: 0.94714
Test set score: 0.94389
```

2. Ridge Regression

i) alpha = 0.01

[44]:
```
rr = Ridge(alpha=0.01).fit(X_train, y_train)
rr_y_pred = rr.predict(X_test)
```

[45]:
```
# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test, rr_y_pred))
# The coefficient of determination score
print("Training set score: %.5f" % rr.score(X_train,y_train))
print("Test set score: %.5f" % rr.score(X_test,y_test))
```

```
Mean absolute error: 2.64432
Training set score: 0.94714
Test set score: 0.94389
```

ii) alpha = 0.1

[46]:
```
rr01 = Ridge(alpha=0.1).fit(X_train, y_train)
rr01_y_pred = rr01.predict(X_test)
```

[47]:
```
# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test, rr01_y_pred))
# The coefficient of determination score
print("Training set score: %.5f" % rr01.score(X_train,y_train))
print("Test set score: %.5f" % rr01.score(X_test,y_test))
```

```
Mean absolute error: 2.64436
Training set score: 0.94714
Test set score: 0.94388
```

iii) alpha = 1

[48]:
```
rr1 = Ridge(alpha=1).fit(X_train, y_train)
rr1_y_pred = rr1.predict(X_test)
```

[49]:
```
# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test, rr1_y_pred))
```

```python
# The coefficient of determination score
print("Training set score: %.5f" % rr1.score(X_train,y_train))
print("Test set score: %.5f" % rr1.score(X_test,y_test))
```

```
Mean absolute error: 2.64498
Training set score: 0.94713
Test set score: 0.94387
```

iv) alpha = 10

[50]:
```python
rr10 = Ridge(alpha=10).fit(X_train, y_train)
rr10_y_pred = rr10.predict(X_test)
```

[51]:
```python
# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test, rr10_y_pred))
# The coefficient of determination score
print("Training set score: %.5f" % rr10.score(X_train,y_train))
print("Test set score: %.5f" % rr10.score(X_test,y_test))
```

```
Mean absolute error: 2.66011
Training set score: 0.94663
Test set score: 0.94340
```

3. Lasso Regression

i) alpha = 0.01

[52]:
```python
lasso = linear_model.Lasso(alpha=0.01).fit(X_train, y_train)
lasso_y_pred = lasso.predict(X_test)
```

[53]:
```python
# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test, lasso_y_pred))
# The coefficient of determination score
print("Training set score: %.5f" % lasso.score(X_train,y_train))
print("Test set score: %.5f" % lasso.score(X_test,y_test))
# Coefficients used
print("Number of features used: %.f" % np.sum(lasso.coef_!=0))
```

```
Mean absolute error: 2.64918
Training set score: 0.94710
Test set score: 0.94379
Number of features used: 18
```

ii) alpha = 0.1

[54]:
```python
lasso01 = linear_model.Lasso(alpha=0.1).fit(X_train, y_train)
lasso01_y_pred = lasso01.predict(X_test)
```

[55]:
```python
# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test, lasso01_y_pred))
# The coefficient of determination score
print("Training set score: %.5f" % lasso01.score(X_train,y_train))
print("Test set score: %.5f" % lasso01.score(X_test,y_test))
# Coefficients used
print("Number of features used: %.f" % np.sum(lasso01.coef_!=0))
```

```
Mean absolute error: 2.69353
Training set score: 0.94541
Test set score: 0.94196
Number of features used: 15
```

iii) alpha = 1

[56]:
```python
lasso1 = linear_model.Lasso(alpha=1).fit(X_train, y_train)
lasso1_y_pred = lasso1.predict(X_test)
```

[57]:
```python
# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test, lasso1_y_pred))
# The coefficient of determination score
print("Training set score: %.5f" % lasso1.score(X_train,y_train))
print("Test set score: %.5f" % lasso1.score(X_test,y_test))
# Coefficients used
print("Number of features used: %.f" % np.sum(lasso1.coef_!=0))
```

```
Mean absolute error: 3.28416
Training set score: 0.91199
Test set score: 0.91000
Number of features used: 9
```

iv) alpha = 10

[58]:
```python
lasso10 = linear_model.Lasso(alpha=10).fit(X_train, y_train)
lasso10_y_pred = lasso10.predict(X_test)
```

[59]:
```python
# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test, lasso10_y_pred))
# The coefficient of determination score
print("Training set score: %.5f" % lasso10.score(X_train,y_train))
print("Test set score: %.5f" % lasso10.score(X_test,y_test))
# Coefficients used
print("Number of features used: %.f" % np.sum(lasso10.coef_!=0))
```

```
Mean absolute error: 10.10929
Training set score: 0.12939
Test set score: 0.13297
Number of features used: 1
```

4. Support Vector Regression (SVR)

To find and used the best parameters for each kernel function (RBF, linear, polynomial), I used `GridSearchCV` on

- C: [0.1, 1, 10, 100]
- gamma: [0.0001, 0.001, 0.1, 1]

i) RBF Kernel

[60]:
```
# find the best parameters
svr_rbf_param = {'kernel':['rbf'], 'C':[0.1, 1, 10, 100], 'gamma':[0.1, 1, 10]}
svr_rbf_GridSearch = GridSearchCV(SVR(), svr_rbf_param).fit(X_train, y_train)
print(svr_rbf_GridSearch.best_params_)
```

```
{'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}
```

[61]:
```
# use the parameters derived from GridSearchCV
svr_rbf = SVR(kernel="rbf", C=10, gamma=0.1).fit(X_train, y_train)
svr_rbf_y_pred = svr_rbf.predict(X_test)
```

[62]:
```
# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test, svr_rbf_y_pred))
# The coefficient of determination score
print("Training set score: %.5f" % svr_rbf.score(X_train,y_train))
print("Test set score: %.5f" % svr_rbf.score(X_test,y_test))
```

```
Mean absolute error: 2.91589
Training set score: 0.95704
Test set score: 0.92704
```

ii) Linear Kernel

[63]:
```
# find the best parameters
svr_lin_param = {'kernel':['linear'], 'C':[0.1, 1, 10, 100], 'gamma':[0.1, 1,␣
↪10]}
svr_lin_GridSearch = GridSearchCV(SVR(), svr_lin_param).fit(X_train, y_train)
print(svr_lin_GridSearch.best_params_)
```

```
{'C': 100, 'gamma': 0.1, 'kernel': 'linear'}
```

[64]:
```
# use the parameters derived from GridSearchCV
svr_lin = SVR(kernel="linear", C=100, gamma=0.1).fit(X_train, y_train)
svr_lin_y_pred = svr_lin.predict(X_test)
```

[65]:
```
# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test, svr_lin_y_pred))
# The coefficient of determination score
print("Training set score: %.5f" % svr_lin.score(X_train,y_train))
print("Test set score: %.5f" % svr_lin.score(X_test,y_test))
```

```
Mean absolute error: 2.64738
Training set score: 0.94674
Test set score: 0.94348
```

iii) Polynomial Kernel

I could not finish running `GridSearchCV` on the polynomial kernel even after hours, so I manually plugged in and compared a few combinations of parameters.

[66]:
```
# parameter trial - 1 (C=1, gamma=0.1)
svr_poly_1 = SVR(kernel="poly", C=1, gamma=0.1).fit(X_train, y_train)
svr_poly_1_y_pred = svr_poly_1.predict(X_test)

# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test,␣
 ↪svr_poly_1_y_pred))
# The coefficient of determination score
print("Training set score: %.5f" % svr_poly_1.score(X_train,y_train))
print("Test set score: %.5f" % svr_poly_1.score(X_test,y_test))
```

```
Mean absolute error: 4.31301
Training set score: 0.87951
Test set score: 0.83758
```

[67]:
```
# parameter trial - 2 (C=10, gamma=0.1)
svr_poly_2 = SVR(kernel="poly", C=10, gamma=0.1).fit(X_train, y_train)
svr_poly_2_y_pred = svr_poly_2.predict(X_test)

# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test,␣
 ↪svr_poly_2_y_pred))
# The coefficient of determination score
print("Training set score: %.5f" % svr_poly_2.score(X_train,y_train))
print("Test set score: %.5f" % svr_poly_2.score(X_test,y_test))
```

```
Mean absolute error: 4.33899
Training set score: 0.91981
Test set score: 0.82363
```

[68]:
```
# parameter trial - 3 (C=100, gamma=0.1)
svr_poly_3 = SVR(kernel="poly", C=100, gamma=0.1).fit(X_train, y_train)
svr_poly_3_y_pred = svr_poly_3.predict(X_test)

# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test,␣
 ↪svr_poly_3_y_pred))
# The coefficient of determination score
print("Training set score: %.5f" % svr_poly_3.score(X_train,y_train))
print("Test set score: %.5f" % svr_poly_3.score(X_test,y_test))
```

```
Mean absolute error: 5.45244
Training set score: 0.94749
Test set score: 0.54277
```

[69]:
```
# parameter trial - 4 (C=1, gamma=1)
svr_poly_4 = SVR(kernel="poly", C=1, gamma=1).fit(X_train, y_train)
svr_poly_4_y_pred = svr_poly_4.predict(X_test)

# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test,␣
↪svr_poly_4_y_pred))
# The coefficient of determination score
print("Training set score: %.5f" % svr_poly_4.score(X_train,y_train))
print("Test set score: %.5f" % svr_poly_4.score(X_test,y_test))
```

```
Mean absolute error: 7.41426
Training set score: 0.96319
Test set score: -0.10572
```

[70]:
```
# use the parameters with a low MAE and high training/test R2 score
svr_poly = SVR(kernel="poly", C=1, gamma=0.1).fit(X_train, y_train)
svr_poly_y_pred = svr_poly.predict(X_test)
```

[71]:
```
# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test,␣
↪svr_poly_y_pred))
# The coefficient of determination score
print("Training set score: %.5f" % svr_poly.score(X_train,y_train))
print("Test set score: %.5f" % svr_poly.score(X_test,y_test))
```

```
Mean absolute error: 4.31301
Training set score: 0.87951
Test set score: 0.83758
```

5. Decision Tree

[85]:
```
# find the best parameters
dtree_param = {"max_depth":[5, 6, 7, 8, 9, 10],
               "min_samples_split":[0.1, 0.2, 0.3]}
dtree_GridSearch = GridSearchCV(tree.DecisionTreeRegressor(), dtree_param).
↪fit(X_train, y_train)
print(dtree_GridSearch.best_params_)
```

```
{'max_depth': 8, 'min_samples_split': 0.1}
```

[86]:
```
# use the parameters derived from GridSearchCV
dtree = tree.DecisionTreeRegressor(max_depth=8, min_samples_split=0.1).
↪fit(X_train, y_train)
```

```
dtree_y_pred = dtree.predict(X_test)
```

[87]:
```
# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test, dtree_y_pred))
# The coefficient of determination score
print("Training set score: %.5f" % dtree.score(X_train,y_train))
print("Test set score: %.5f" % dtree.score(X_test,y_test))
```

```
Mean absolute error: 4.55357
Training set score: 0.83549
Test set score: 0.82285
```

6. Random Forest

[92]:
```
# find the best parameters
rforest_param = {"n_estimators":[10, 20, 30, 40, 50],
                 "min_samples_leaf":[1, 5, 10, 15],
                 "max_features":["auto","sqrt",None]}
rforest_GridSearch = GridSearchCV(RandomForestRegressor(), rforest_param).
 ↪fit(X_train, y_train)
print(rforest_GridSearch.best_params_)
```

```
{'max_features': 'sqrt', 'min_samples_leaf': 1, 'n_estimators': 50}
```

[93]:
```
# use the parameters derived from GridSearchCV
rforest = RandomForestRegressor(n_estimators=50, min_samples_leaf=1,␣
 ↪max_features='auto').fit(X_train, y_train)
rforest_y_pred = rforest.predict(X_test)
```

[94]:
```
# The mean absolute error
print("Mean absolute error: %.5f" % mean_absolute_error(y_test, rforest_y_pred))
# The coefficient of determination score
print("Training set score: %.5f" % rforest.score(X_train,y_train))
print("Test set score: %.5f" % rforest.score(X_test,y_test))
```

```
Mean absolute error: 3.14298
Training set score: 0.98764
Test set score: 0.91818
```

Model Evaluation

The following table summarizes the results of the different models implemented. For the models that tried different parameter values (e.g., ridge, lasso), only the parameters with the lowest MAE are shown:

	MAE	Training Score	Test Score
Linear	2.64431	0.94714	0.94389
Ridge	2.64432	0.94714	0.94389

	MAE	Training Score	Test Score
Lasso	2.64918	0.94710	0.94379
RBF SVR	2.91589	0.95704	0.92704
Linear SVR	2.64738	0.94674	0.94348
Polynomial SVR	4.31301	0.87951	0.83758
Decision Tree	4.55357	0.83549	0.82285
Random Forest	3.14298	0.98764	0.91818

As expected, the **linear regression models** (linear regression, ridge regression, lasso regression, linear SVR) have similar results with an average MAE of 2.65, training score of 0.95, and test score of 0.94. Since the average number of wins in the entire dataset is 77.88, the linear models predict the number of wins to be approximately 77.88+2.65=80.53. Considering that the total number of games played is at least 160 games every year, this error is very small (although such "small" differences could highly affect the team rankings, especially during the end of the season). The linear models also do not show indications of overfitting as can be seen from the similar coefficient of determination scores for both the training and test set.

The table below also shows the feature variables with the highest coefficient values of the linear models. `R` (runs scored), `G` (games played), `Rank` (position in final standings), and `SV` (saves) especially have a strong positive correlation with the number of wins. It makes sense for these features to be highly correlated because a team is likely to have more wins if the team scores more runs (`R`), plays more games (`G`), and has higher rankings (`Rank`), and "saves" (`SV`) is only recorded for teams that won.

[110]:
```
coef_df = pd.DataFrame()
coef_df['Features'] = new_x_attributes
coef_df['Linear'] = lr.coef_
coef_df['Ridge'] = rr.coef_
coef_df['Lasso'] = lasso.coef_

coef_df.sort_values(by='Linear', ascending = False).head(10)
```

[110]:

	Features	Linear	Ridge	Lasso
2	R	8.786274	8.785281	8.690665
1	G	5.911753	5.909876	5.686357
0	Rank	3.773018	3.773082	3.759902
12	SV	2.065450	2.065402	1.975719
11	SHO	0.941617	0.941731	0.945232
5	3B	0.416630	0.416742	0.423813
13	HA	0.108416	0.108151	0.000000
19	LastYearLgWin	0.102461	0.102478	0.095497
3	H	0.102163	0.103002	0.149044
7	BB	0.048718	0.049005	0.050451

The **non-linear regression models** (RBF SVR, polynomial SVR, decision tree, random forest) surprisingly do not show strong signs of overfitting, which may have been likely due to the flexible nature of non-linear models. The random forest model has the strongest indication of overfitting as it has the biggest difference between the training vs test scores (0.99 vs 0.92). Nonetheless, the random forest model has high coefficient of determination scores in general and has a low MAE of 3.14, which is only 0.49 higher than the average MAE of the linear regression models. The support vector regression model using the RBF kernel also has a low MAE of 2.92, which is only 0.22 higher than the average MAE of the linear regression models, and high coefficient of determination scores.

Therefore, I conclude that the RBF support vector model and the random forest model performed the best among the non-linear regression models and all the linear regression models performed very similarly and well. For this particular database and objective, the linear models performed better than non-linear models (lower MAE and higher coefficient of determination scores), implying a linear relationship is sufficient to explain the output variable (number of wins) using the extracted/pruned feature variables.