Predicting the Winner of the NBA Finals

MIE368 - Analytics in Action

December 5th, 2019

Chris Overvelde - 1003912741
Maria Papadimitriou - 1003913624
Toufic Constantin - 1003985949

Abstract

In the 2019 offseason of the National Basketball Association (NBA), there were multiple shocking trades, free agency signings, and incoming college talent that shuffled many team rosters, and as a result caused the league to become more balanced. This change of events caused difficulty for bettors to predict a winning team from a playoff bracket due to the recent widespread of exceptional talent. For this reason, this project will predict the probability of any playoff team winning the NBA Finals. In turn, NBA sports bettors will be able to use this model in their decision making process when choosing which team to bet on from the start of the playoffs to increase their confidence. In order to determine which playoff team is most likely to win the NBA Finals in a given year, multiple machine learning methods were used. Firstly, a logistic regression model was trained on historical playoff games data to predict the game outcome -- whether the home or away team won. A bracket algorithm was created which performs the following operations throughout all rounds of the playoffs. The logistic regression model was used to calculate the probability of a team winning against their opponent by using a dataset that contained regular season data. A new probability was calculated every matchup of the playoffs, and was used as a threshold to then simulate a best of seven-game series. Once a team reached four games, they were declared the winner of the series and were brought forth to the next round. This was done until a champion was declared. The simulation was repeated 300-500 times using Monte Carlo simulation to determine the most likely winner. Ten different years of finals winners were analyzed to determine how accurate the prediction and simulation models were. The overall simulation model correctly predicted the most likely winner in 4 of the 10 years, meaning there is a 40% accuracy. In conclusion, it was determined that predicting playoff basketball outcomes can be complex, as there are factors that are extremely difficult to be accounted for in a model. These factors include player injuries, and how certain players perform better in more crucial games. Further investigation into incorporating these difficult factors may provide a greater insight into the prediction and increase the accuracy of the model.

Table of Contents

1.0 Introduction

In the 2019 offseason, many shocking trades, free agency signings, and new college talent shuffled team rosters which as a result, caused the league to become more balanced compared to previous years. This has caused major excitement for the upcoming season, however it is now more challenging for bettors to predict a winning team from a playoff bracket due to the recent widespread of exceptional talent.

This purpose of this project is to predict the probability of any playoff team winning the NBA finals, given their regular season performance. After rework of the original project scope and methodologies, it was decided that the main focus of the project is to determine which NBA team is most likely to win the finals given a playoff bracket, rather than focusing on maximizing expected earnings. The scope was rephrased such that given the final prediction of the model, bettors can feel confident in selecting a team at the start of the playoffs at winning the NBA Finals. In addition, if some slight changes were made, the model could also be used by normal fans of the sport to assist them in creating playoff brackets, as the model can be used to determine the winner of each round.

The machine learning model utilizes regular season statistics which will be elaborated further in the *Data* section below. This data will assist in the understanding as to how likely a team is to succeed in the playoffs. The following sections will discuss the data utilized in this project, the machine learning methodologies, the final results, and a discussion of the next steps in developing the model.

2.0 Data

There are four significant datasets that are used throughout the entirety of the project. The first dataset, NBA Finals Team Statistics [1], contains the information of the team that won the finals in a given year, as well as their regular season data. This dataset is used to determine the accuracy of the final model in terms of how accurately it predicts the winning team of the NBA finals compared to the actual winners in a given season. The second dataset, NBA Data [2] contains aggregate regular season data for each NBA team for each season. Some of the key features in this dataset include the total points scored, points scored against, total offensive rebounds, and total defensive rebounds for each team in the NBA. The third dataset, NBA Games All [3], contains all NBA games (preseason, regular seasons, all star, and playoff) from the years 1950-2018. This dataset includes basic game data, game dates, the home team, the away team, and the result of the match (win or loss) in respect to the home team. The final dataset, NBA Teams All [3] contains the key features of team ID numbers, and the respective team name abbreviation. For the purpose of consistency, each dataset was converted to include the team abbreviations instead of team names in order to join the datasets more efficiently.

3.0 Data Cleaning and Exploratory Data Analysis (EDA)

The initial component of this project included data cleaning and EDA in order to construct the working dataset that would be used to train the classification model. This component was crucial for the base classification model that would be used to predict the outcome of an individual playoff game. Data cleaning can be found in Appendix A.

After the data cleaning was performed, a thorough analysis of the datasets followed. From this, it was determined that the desired working dataset structure was to include select regular season totals from both the home and away teams in a playoff matchup, where the totals are used as feature variables to train the model. The target variable is a binary variable indicating the outcome of the game with respect to the home team (either being a 1 for if the home team wins or a 0 if the home team loses). A visual description of the working dataframe can be seen below in Table 1. To see a full description of the dataset and variables used, see Appendix B.

Table 1: Initial Working Dataframe used to Train the Initial Prediction Model

Feature Variables	H_Wins, A_Wins, H_oppPTS, A_oppPTS, H_ORB, A_ORB, H_DRB, A_DRB
Target Variable	homeTeamWins

The proceeding steps included populating the data frame with the different playoff matchups and the outcome of the game. This approach can be referenced in Appendix C.

After finally constructing the working dataframe, each value was converted to an integer and feature imbalances were checked for by analyzing the table produced by the built in Python function 'df.describe()'. It was determined that there were no apparent imbalances as there were no large skews in the distribution of values (Appendix D, Figure D1).

To further verify the claim that none of the data was skewed, a general plot for each of the columns in the final dataframe was done to see what type of distribution was produced. It was then found that each of the distributions were approximately normal for each feature, as there was no evident Poisson tail, as seen in Appendix D, Figure D2. This indicated that the data was again, not skewed, and normalization of the data was not required. Through plotting, it was also determined that there were some values inputted as zeros. For example, some of the data showed that the number of H_Wins and A_Wins (which represent regular season wins) were 0 for some teams, which is not actually the case. Therefore the rows which included these zeros were voided from the working dataset as it would potentially train the model incorrectly, with regular season statistics that would never occur (Appendix D, Figure D3).

4.0 Methods

The methodologies of focus for this project include a combination of both prediction and simulation machine learning concepts. The prediction component includes a classification model that uses aggregate regular season data of two playoff teams as feature variables to predict the outcome of the playoff game matchup with respect to the home team (where either the home team wins or loses). Given the feature variables, the model was trained such that it classifies the outcome of the game through a binary variable, as either the home team won the match, with a value of 1, or lost, with a value of 0. The prediction model was used to predict the probability of each team winning, and these outputs were then used to simulate a best of 7 series. The team in the matchup which won 4 games first was declared the winner of the

matchup and was brought forth to the next round of the playoffs. The aforementioned analysis was conducted on each round of the playoffs, until a champion team was determined. This bracket was then simulated 300-500 times using monte carlo simulation. Using trial and error on the 2016 playoffs, it was found that in order to simulate a playoff bracket for a given year 500 times, it took over 15 minutes. For this reason, it was decided that the number of simulations was to be 300 for the remaining years, for the purpose of runtime. To ensure that 300 replications would be enough to get the same results as running for 500 simulations, the 2016 playoffs were run for 100 simulations and then for 300 simulations and the final results were the same each time, confirming that 300 simulations of the bracket would be suffice. The simulation outputs a distribution of the number of times each team was determined to be the winner of the playoffs, where the team names are found on the x-axis, and the amount of wins are found on the y-axis. The initial findings and results can be further seen in both the *Initial Findings* and *Results* sections of this report.

5.0 Initial Findings

Before a classification model was created, the data frame was split into a training set and test set, using a 70:30 ratio (Appendix E, Figure E1). Both a logistic regression model and a CART model were created and fitted with the data. It was determined that the logistic regression model returned better scores for both the testing and training data, so it was decided that the logistic regression model would be used (Appendix E). After creating and fitting the logistic regression model, the outputted scores were 0.616 for the training data and 0.622 for the testing data (Appendix E1). The scores for both the train and test set were relatively similar, indicating that the data was not overfitted, as the score for the test set is also greater than the score for the train set.

6.0 Rework of the Logistic Regression Model

Though the logistic regression model score produced a value of 0.616 for the train set and 0.622 for the test set, further approaches were taken in order to potentially increase the score and in turn, increase the prediction model accuracy. These approaches included incorporating more regular season aggregates into the dataset, computing head to head data, and performing feature engineering.

As mentioned, more regular season totals were added to the final dataframe in order to recognize whether or not these features had a higher correlation to the target variable compared to the regular season data used in the initial prediction model. These regular season totals included home and away field goals made (FG), home and away field goals attempted (FGA), home and away 2 points made (2P), home and away 2 points attempted (2PA), home and away 3 points made (3P), home and away 3 points attempted (3PA), home and away free throws made (FT), home and away three points attempted (FTA), and home and away assists (AST).

Table 2: Working Dataframe

Feature Variables	H_Wins, A_Wins, H_oppPTS, A_oppPTS, H_ORB, A_ORB, H_DRB, A_DRB, H_FG, A_FG, H_FGA, A_FGA, H_2P, A_2P, H_2PA, A_2PA, H_3P, A_3P, H_3PA, A_3PA, H_FT, H_FTA, A_FT, A_FTA, H_AST, A_AST
Target Variable	homeTeamWins

An additional approach to improve the score for the prediction model was to include head to head data based on the regular season matchups. This data was calculated by creating a function in Python that inputs a given season year, iterates through all the games in which occured in that season and created a dictionary within a dictionary, where the key of the first dictionary was the hometeam, and the value of the first dictionary was an inner dictionary containing the away teams (as keys) and the respective ratio of games the home team won in respect to the away team (as values). This value was then extracted and inputted into the final dataframe, under the feature name of h2hWinP, in order to test whether the correlation to the target variable was significant. The head to head function can be seen in Figure F1 of Appendix F.

In a final attempt to improve the score, the features that were appended to the columns of the working dataframe were feature engineered to determine whether or not the operations between two current feature variables provided a larger correlation to the target variable than the features themselves. Each feature ran through a function which looped through every pair of features, applied the operation, and returned the feature pairings if the engineered feature had a correlation higher than that of its original components. The feature engineering function and its respective outputs can be found in Appendix G, Table G1. Given these outputs, the original variables in the dataframe were dropped and replaced with the new engineered features (Appendix G, Figure G3). The finalized engineered features were incorporated into the working dataframe, which can be seen more clearly in Table 3 below.

Table 3: Working Dataframe used to Train the Prediction Model

Feature Variables	H_Wins, A_Wins, H_ORB, A_ORB, H_DRB, A_DRB, h2hWinP, hfgP, afgP, h2pP, a2pP, h2pP, a3pP, diffPTS, diffoppPTS, hFTP, aFTP
Target Variable	homeTeamWins

Though the score only had an increase of 1.2% for the train set and 0.7% for the test set, it was decided that this logistic regression model would be used in the bracket simulation component of the project as this was the maximized improvement that was able to be made after all the iterations of feature engineering and creation.

7.0 Results

As mentioned, the logistic regression model was revisited as the working dataframe used to train the model was adjusted to improve the score. The final model resulted in the prediction model to have a training score of 0.627 and testing score of 0.628. Again, these values indicate that the data is not overfit due to similar values, and the test score being larger than the train score (Appendix H, Figure H1).

The bracket algorithm was created in order to effectively simulate an NBA playoffs. This component involved creating three different functions in Python. The first function, *bracket*, inputs two dataframes, the first containing regular season statistics for the 16 playoff teams that year, and a dataframe containing the regular season head to head win percentages from that year. The function also takes in a logistic regression model, and the year in which it is going to predict. The purpose of this function is to create 2 dictionaries for each conference, each containing 4 teams and their seed. This was used for the next function *playoffs*, but only for the first two rounds of the playoffs. After these rounds were completed, this function also deals with the conference finals, and the NBA Finals by directly calling the method *matchup* for those rounds. *Bracket* returns the winner of the playoffs. This function can be seen in Appendix I.

Playoffs takes in the two previously mentioned dataframes, a logistic regression model, and two dictionaries (either both of the eastern conference dictionaries, or both of the western conference dictionaries). This function is used for the first 2 rounds of the playoffs, as it pairs up the highest and lowest seeded teams in each dictionary. This function calls the function *matchup* to determine the winner of a given matchup, and then keeps track of the winning team. This function can be seen in Appendix J.

The function *matchup* takes in the two previously mentioned dataframes, a logistic regression model, and the two teams playing each other. This function creates a new dataframe that has all of the necessary feature variables that are needed to use the previously trained logistic regression model. The probability that the first inputted team (higher seeded team) wins is then calculated by using the 'mdl.predict_proba(df)[0,1]' method. To then determine the winner of an individual game, a random number that's uniformly distributed between 0 and 1 is generated. If the random number is between 0 and the probability that the first team would win a game, then the amount of games won by the first team is incremented by 1. If the random number is greater than the probability of the first team winning a game, then the amount of games won by the second team is incremented by 1. A best of 7 game series is generated by doing this, and the function returns both the winner and loser. This function can be seen in Appendix K.

After testing to make sure these functions all worked together, each year's playoffs were simulated 300 or 500 times. A dictionary kept track of how many times each team in that year's playoffs won, and then displayed a bar chart, which visually showed how many times each team won the playoffs. This histogram is displayed with the name of the teams on the x-axis and the amount of wins on the y-axis. The team that the model predicts to have the greatest chance of winning the playoffs that year is determined by the team which won the most number of simulations. The code that was used to do this can be seen in Appendix L.

After inputting the data from the regular season of the years 2006 - 2016 (excluding the year 2012 as the data was insufficient), the bracket simulation correctly predicted a total of 4 of 10 winners. This accounted for a total accuracy of 40% of the simulation model. The comparison between what the model predicted and the actual winners can be seen below in Table 4.

Table 4: Simulation Model Predictions versus Actual Winners

Season	Our Prediction	Actual Winner
2016	GSW	CLE
2015	GSW	GSW
2014	SAS	SAS
2013	MIA	MIA
2011	CHI	DAL
2010	CLE	LAL
2009	CLE	LAL
2008	BOS	BOS
2007	DAL	SAS
2006	DET	MIA

Correct Predictions
Incorrect Predictions

After analyzing the years in which the model predicted correctly and incorrectly, further investigation was done to understand why the prediction model produced outputs as such, which will be further elaborated upon in the *Discussion* section. An example output of the bracket where the model predicted correctly can be seen in Figure 1.

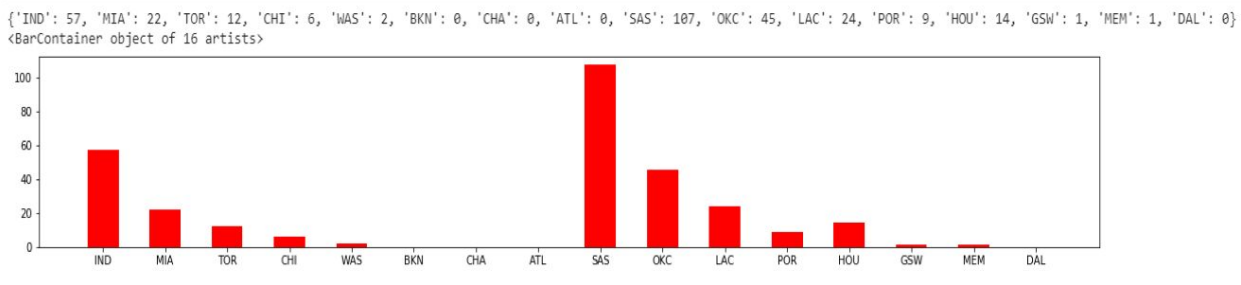


Figure 1: 2014 Bracket Simulation of the NBA Playoffs

Figure 1 encompasses the final distribution generated from the bracket algorithm. Of the 300 simulations performed, the San Antonio Spurs won 107 simulations. In other words, they won the most number of simulations, and were determined to be the most likely winners of the 2014 season according to the model. In actuality, the San Antonio Spurs were the winners of the 2014 NBA playoffs, which indicates that the model was able to correctly predict the outcome of this playoffs.

Conversely, a year in which the model did not predict correctly is 2016, as seen in Figure 2 below.

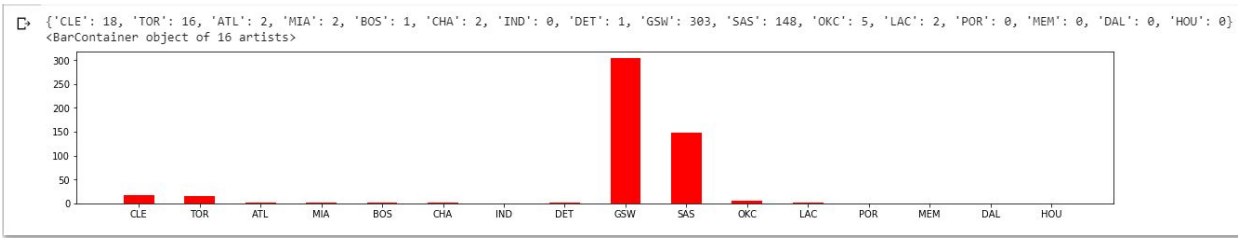


Figure 2: 2016 Bracket Simulation of the NBA Playoffs

The output of the model shows the Golden State Warriors as the most likely winners of the finals. More specifically, of the 500 simulations performed for 2016, the Warriors won 303 (the majority). The actual winners of the NBA Finals in 2016 were the Cleveland Cavaliers. After analyzing the Cavaliers further, it can be seen that they only win 18 simulations, third to Golden State and San Antonio Spurs, so they were not expected to win. For the other outcomes, please see Appendix M.

8.0 Discussion

Initially, it was concerning that the model did not predict the Cleveland Cavaliers as the winners for the 2016 NBA finals. Further investigation was done in order to determine why the predicted results were inaccurate. In 2016, the Golden State Warriors had the NBA's best ever regular season record of 73-9 (73 wins, 9 losses) and were considered heavy favourites to win the title. As the logistic regression model is trained only on data from the regular season, the number of wins in the season, as well as other impressive regular season statistics from Golden State heavily impacted the probability of them winning over any other team when matched up against them in the playoffs. In doing so, the number of simulations where the Warriors were declared the winner was a majority, and therefore it was inevitable that the model predicted them as the winner. However, the events that actually transpired were quite shocking. In the finals, the Golden State Warriors held a 3-1 lead over the Cleveland Cavaliers in the finals, but lost their lead and ended up losing 4-3. In NBA history, a team in the NBA finals has never lost a 3-1 lead (apart from this case). This was a rare situation that was almost impossible for the model to predict.

For the other playoffs' where the model inaccurately predicted the winner, it was noticed that the model almost always predicted one of the top seeds to win. This makes logical sense as the playoff seeds are determined on how well the team does in the regular season, and the prediction model was only trained on regular season data.

9.0 Conclusion and Future Directions

In conclusion, predicting the outcome of a sports matchup is extremely difficult. There are many factors that need to be accounted for, some of which are unpredictable or extremely unlikely, which makes them very difficult to incorporate into a prediction model. Examples of this include serious player injuries that occur during the playoffs, or accounting for the disparity between how certain players perform during the regular season versus the playoffs.

In order to increase the accuracy of the model, future approaches include looking to incorporate playoff data on an ongoing basis (i.e. updating the head to head data as the playoff games progress), looking into the series scores of previous playoff matchups, factoring in player performance in the playoffs in comparison to the regular season, looking into the coaches on each team and their playoff experience, and looking into how prediction models would be able to account for player injuries. Therefore people betting on the outcome of sports games are always taking a risk, but with a well trained model, bettors can increase their confidence when choosing to risk their money. However, no matter what model changes and improvements are made, no model will ever be perfect, and it is this unpredictability that makes the NBA exciting and popular to watch.

10.0 References

[1] Carter, A. (2018). NBA Championship Data. Retrieved from: https://www.kaggle.com/carterallen/nba-championship-data

[2] Chan, T. (2019). MIE268: Lab Week 2. [Excel File].

[3] Hallmark, E. (2018). NBA Historic Stats and Betting Data. Retrieved from: https://www.kaggle.com/ehallmar/nba-historical-stats-and-betting-data

11.0 Appendices

Appendix A: Data Cleaning

```
df1 = pd.read_csv("{}/champs_series_averages.csv".format(data_dir), header=None)
df2 = pd.read_csv("{}/NBA_data.csv".format(data_dir), header=None)
df3 = pd.read_csv("{}/nba_games_all.csv".format(data_dir), header=None)
df4 = pd.read_csv("{}/nba_teams_all.csv".format(data_dir), header=None)
dfFinal = pd.read_csv("{}/dfFinal.csv".format(data_dir), header=None)

df1 = df1.replace(np.nan, "ID")
newHeader = df1.iloc[0]
df1 = df1[1:]
df1.columns = newHeader
df1 = df1.drop(columns=['ID'])

newHeader = df2.iloc[0]
df2 = df2[1:]
df2.columns = newHeader

df2 = df2.replace(to_replace = 'Atlanta Hawks', value = 'ATL')
df2 = df2.replace(to_replace = 'Boston Celtics', value = 'BOS')
df2 = df2.replace(to_replace = 'Cleveland Cavaliers', value = 'CLE')
df2 = df2.replace(to_replace = 'New Orleans Hornets', value = 'NOP')
df2 = df2.replace(to_replace = 'Chicago Bulls', value = 'CHI')
df2 = df2.replace(to_replace = 'Dallas Mavericks', value = 'DAL')
df2 = df2.replace(to_replace = 'Denver Nuggets', value = 'DEN')
df2 = df2.replace(to_replace = 'Golden State Warriors', value = 'GSW')
df2 = df2.replace(to_replace = 'Houston Rockets', value = 'HOU')
df2 = df2.replace(to_replace = 'Los Angeles Clippers', value = 'LAC')
df2 = df2.replace(to_replace = 'Los Angeles Lakers', value = 'LAL')
df2 = df2.replace(to_replace = 'Miami Heat', value = 'MIA')
df2 = df2.replace(to_replace = 'Milwaukee Bucks', value = 'MIL')
df2 = df2.replace(to_replace = 'Minnesota Timberwolves', value = 'MIN')
df2 = df2.replace(to_replace = 'Brooklyn Nets', value = 'BKN')
df2 = df2.replace(to_replace = 'New York Knicks', value = 'NYK')
df2 = df2.replace(to_replace = 'Orlando Magic', value = 'ORL')
df2 = df2.replace(to_replace = 'Indiana Pacers', value = 'IND')
df2 = df2.replace(to_replace = 'Philadelphia 76ers', value = 'PHI')
df2 = df2.replace(to_replace = 'Phoenix Suns', value = 'PHX')
df2 = df2.replace(to_replace = 'Portland Trail Blazers', value = 'POR')
df2 = df2.replace(to_replace = 'Sacramento Kings', value = 'SAC')
df2 = df2.replace(to_replace = 'San Antonio Spurs', value = 'SAS')
df2 = df2.replace(to_replace = 'Oklahoma City Thunder', value = 'OKC')
df2 = df2.replace(to_replace = 'Toronto Raptors', value = 'TOR')
df2 = df2.replace(to_replace = 'Utah Jazz', value = 'UTA')
df2 = df2.replace(to_replace = 'Memphis Grizzlies', value = 'MEM')
df2 = df2.replace(to_replace = 'Washington Wizards', value = 'WAS')
df2 = df2.replace(to_replace = 'Detroit Pistons', value = 'DET')
df2 = df2.replace(to_replace = 'Charlotte Bobcats', value = 'CHA')
#Go over some of the below team names, to make sure they are being set to the correctly
associated team abbreviation
df2 = df2.replace(to_replace = 'Charlotte Hornets', value = 'CHA')
df2 = df2.replace(to_replace = 'Washington Bullets', value = 'WAS')
df2 = df2.replace(to_replace = 'Vancouver Grizzlies', value = 'MEM')
df2 = df2.replace(to_replace = 'Kansas City Kings', value = 'SAC')
df2 = df2.replace(to_replace = 'San Diego Clippers', value = 'LAC')
df2 = df2.replace(to_replace = 'New Jersey Nets', value = 'BKN')
df2 = df2.replace(to_replace = 'Seattle SuperSonics', value = 'OKC')
df2 = df2.replace(to_replace = 'New Orleans Pelicans', value = 'NOP')
df2 = df2.replace(to_replace = 'New Orleans/Oklahoma City Hornets', value = 'NOP')

df2 = df2.sort_values('SeasonEnd', ascending=False)
```

```
newHeader = df3.iloc[0]
df3 = df3[1:]
df3.columns = newHeader
newHeader = df4.iloc[0]
df4 = df4[1:]
df4.columns = newHeader

df4 = df4.replace(np.nan, "MISSING") # replacing all of the NaN's with the string 'MISSING'
df4 = df4[df4.abbreviation != 'MISSING'] # all irrelevant rows have 'MISSING' as the team
abbreviation, so we are deleting all of these rows

df4 = df4.drop(columns = 'league_id') # dropping this column, as it provides no useful
information (it's just '00' for every row)

df4.index = range(1,31) # changing the index range because it gets messed up after dropping
the rows
df3 = df3[df3.season_type != 'Pre Season']
df3 = df3[df3.season_type != 'All Star']

#WHY DOESN'T THIS WORK FOR ALL ROWS OF DATA???
for i in range(0,30):
  df3 = df3.replace(to_replace = df4.team_id.iloc[i], value = df4.abbreviation.iloc[i])


dfFinal = df3.copy()
dfFinal = dfFinal[['game_id','game_date','team_id','wl','a_team_id','season_type','season']]
dfFinal = dfFinal[dfFinal.season_type != 'Regular Season']
dfFinal = dfFinal.drop(columns = 'season_type')

# Note:
# team_id is the home team and a_team_id is the away team

#Sorting the data from oldest to newest
dfFinal = dfFinal.sort_values(['season','game_date'])
dfFinal = dfFinal.rename(columns={"team_id": "homeTeam", "a_team_id": "awayTeam", "wl":
"homeTeamWins"})
dfFinal['homeTeamWins'] = dfFinal.homeTeamWins.apply(lambda x: 1 if x == 'W' else 0)
dfFinal = dfFinal[1900:7366]
dfFinal = dfFinal.replace(to_replace = '1979-80', value = 1980)
dfFinal = dfFinal.replace(to_replace = '1980-81', value = 1981)
dfFinal = dfFinal.replace(to_replace = '1981-82', value = 1982)
dfFinal = dfFinal.replace(to_replace = '1982-83', value = 1983)
dfFinal = dfFinal.replace(to_replace = '1983-84', value = 1984)
dfFinal = dfFinal.replace(to_replace = '1984-85', value = 1985)
dfFinal = dfFinal.replace(to_replace = '1985-86', value = 1986)
dfFinal = dfFinal.replace(to_replace = '1986-87', value = 1987)
dfFinal = dfFinal.replace(to_replace = '1987-88', value = 1988)
dfFinal = dfFinal.replace(to_replace = '1988-89', value = 1989)
dfFinal = dfFinal.replace(to_replace = '1989-90', value = 1990)
dfFinal = dfFinal.replace(to_replace = '1990-91', value = 1991)
dfFinal = dfFinal.replace(to_replace = '1991-92', value = 1992)
dfFinal = dfFinal.replace(to_replace = '1992-93', value = 1993)
dfFinal = dfFinal.replace(to_replace = '1993-94', value = 1994)
dfFinal = dfFinal.replace(to_replace = '1994-95', value = 1995)
dfFinal = dfFinal.replace(to_replace = '1995-96', value = 1996)
dfFinal = dfFinal.replace(to_replace = '1996-97', value = 1997)
dfFinal = dfFinal.replace(to_replace = '1997-98', value = 1998)
dfFinal = dfFinal.replace(to_replace = '1998-99', value = 1999)
dfFinal = dfFinal.replace(to_replace = '1999-00', value = 2000)
dfFinal = dfFinal.replace(to_replace = '2000-01', value = 2001)
dfFinal = dfFinal.replace(to_replace = '2001-02', value = 2002)
dfFinal = dfFinal.replace(to_replace = '2002-03', value = 2003)
dfFinal = dfFinal.replace(to_replace = '2003-04', value = 2004)
dfFinal = dfFinal.replace(to_replace = '2004-05', value = 2005)
dfFinal = dfFinal.replace(to_replace = '2005-06', value = 2006)
dfFinal = dfFinal.replace(to_replace = '2006-07', value = 2007)
```

```
dfFinal = dfFinal.replace(to_replace = '2007-08', value = 2008)
dfFinal = dfFinal.replace(to_replace = '2008-09', value = 2009)
dfFinal = dfFinal.replace(to_replace = '2009-10', value = 2010)
dfFinal = dfFinal.replace(to_replace = '2010-11', value = 2011)
dfFinal = dfFinal.replace(to_replace = '2011-12', value = 2012)
dfFinal = dfFinal.replace(to_replace = '2012-13', value = 2013)
dfFinal = dfFinal.replace(to_replace = '2013-14', value = 2014)
dfFinal = dfFinal.replace(to_replace = '2014-15', value = 2015)
dfFinal = dfFinal.replace(to_replace = '2015-16', value = 2016)
```

Appendix B: Feature and Target Variable Descriptions

Feature Variables	**Description**
H_Wins	The number of regular season wins the home team got
A_Wins	The number of regular season wins the away team got
H_PTS	Total points scored by the home team during the regular season
A_PTS	Total points scored by the away team during the regular season
H_oppPTS	Total points scored against home team during the regular season
A_oppPTS	Total points scored against away team during the regular season
H_ORB	Total number of offensive rebounds during the regular season by the home team
A_ORB	Total number of offensive rebounds during the regular season by the away team
H_DRB	Total number of defensive rebounds during the regular season by the home team
A_DRB	Total number of defensive rebounds during the regular season by the away team

Target Variable	**Description**
homeTeamWins	Did the home team win? (Yes 1, no 0)

Appendix C: Populating the Final Data Frame

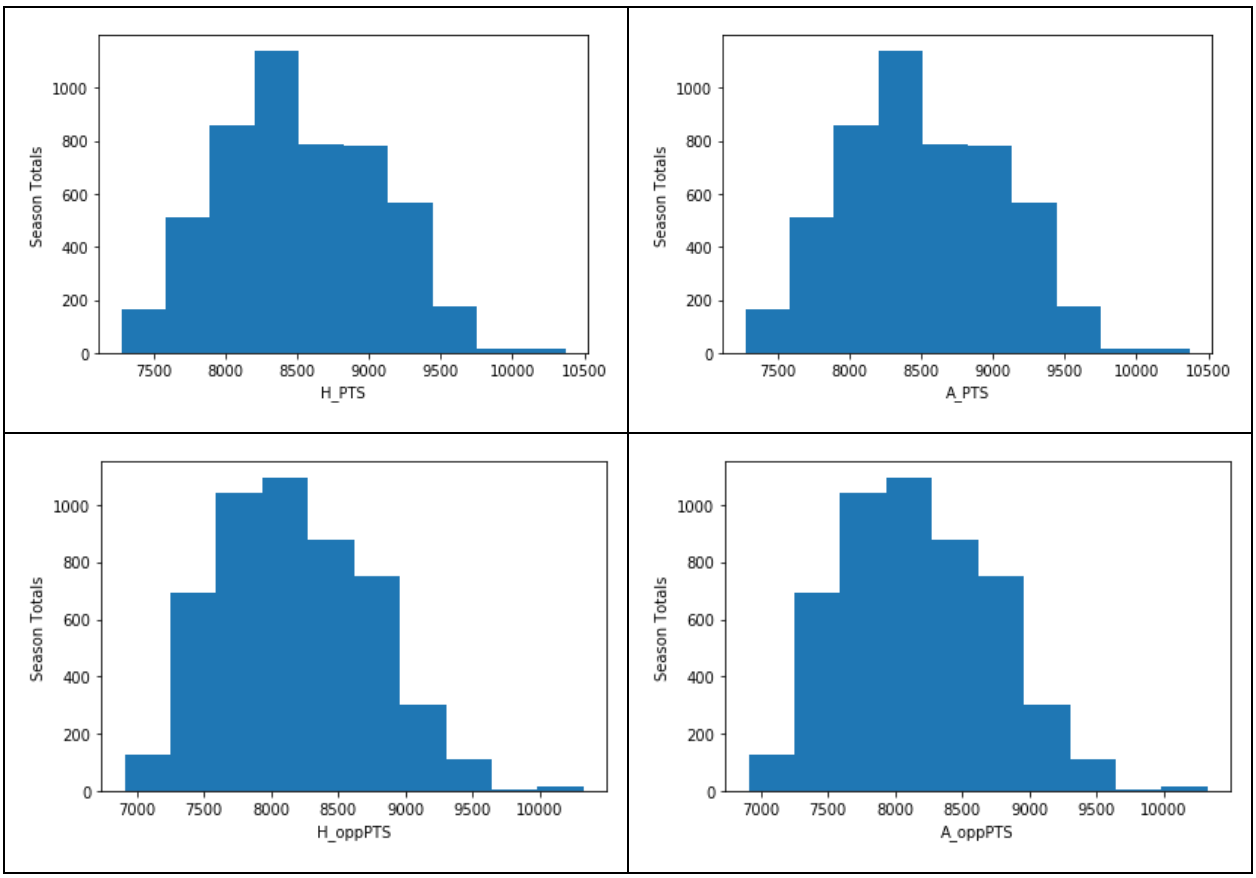
```
# Adding the feature columns
dfFinal['H_Wins'] = 0
dfFinal['A_Wins'] = 0
dfFinal['H_PTS'] = 0
dfFinal['A_PTS'] = 0
dfFinal['H_oppPTS'] = 0
dfFinal['A_oppPTS'] = 0
dfFinal['H_ORB'] = 0
dfFinal['A_ORB'] = 0
dfFinal['H_DRB'] = 0
dfFinal['A_DRB'] = 0

df2.SeasonEnd = df2.SeasonEnd.astype(int)
dfFinal.season = dfFinal.season.astype(int)
for i in range(0,955):
  for j in range(0,5466):
    if((dfFinal.homeTeam.iloc[j] == df2.Team.iloc[i]) and (dfFinal.season.iloc[j] ==
df2.SeasonEnd.iloc[i])):
      dfFinal.H_Wins.iloc[j] = df2.W.iloc[i]
      dfFinal.H_PTS.iloc[j] = df2.PTS.iloc[i]
      dfFinal.H_oppPTS.iloc[j] = df2.oppPTS.iloc[i]
      dfFinal.H_ORB.iloc[j] = df2.ORB.iloc[i]
      dfFinal.H_DRB.iloc[j] = df2.DRB.iloc[i]
    if((dfFinal.awayTeam.iloc[j] == df2.Team.iloc[i]) and (dfFinal.season.iloc[j] ==
df2.SeasonEnd.iloc[i])):
      dfFinal.A_Wins.iloc[j] = df2.W.iloc[i]
      dfFinal.A_PTS.iloc[j] = df2.PTS.iloc[i]
      dfFinal.A_oppPTS.iloc[j] = df2.oppPTS.iloc[i]
      dfFinal.A_ORB.iloc[j] = df2.ORB.iloc[i]
      dfFinal.A_DRB.iloc[j] = df2.DRB.iloc[i]

dfFinal.H_Wins = dfFinal.H_Wins.astype(int)
dfFinal.A_Wins = dfFinal.A_Wins.astype(int)
dfFinal.H_PTS = dfFinal.H_PTS.astype(int)
dfFinal.A_PTS = dfFinal.A_PTS.astype(int)
dfFinal.H_oppPTS = dfFinal.H_oppPTS.astype(int)
dfFinal.A_oppPTS = dfFinal.A_oppPTS.astype(int)
dfFinal.H_ORB = dfFinal.H_ORB.astype(int)
dfFinal.A_ORB = dfFinal.A_ORB.astype(int)
dfFinal.H_DRB = dfFinal.H_DRB.astype(int)
dfFinal.A_DRB = dfFinal.A_DRB.astype(int)
```

Appendix D: Exploratory Data Analysis and Checking for Skewness

```
# Checking for feature imbalances
dfFinal.describe()
```

	homeTeamWins	season	H_Wins	A_Wins	H_PTS	A_PTS	H_oppPTS	A_oppPTS	H_ORB	A_ORB	H_DRB	A_DRB
count	5466.000000	5466.000000	5466.000000	5466.000000	5466.000000	5466.000000	5466.000000	5466.000000	5466.000000	5466.000000	5466.000000	5466.000000
mean	0.500000	1999.308452	50.006220	50.006220	8038.666118	8038.666118	7713.570984	7713.570984	962.907794	962.907794	2376.769667	2376.769667
std	0.500046	10.452016	14.095465	14.095465	2011.092041	2011.092041	1936.486907	1936.486907	279.408541	279.408541	588.713246	588.713246
min	0.000000	1980.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	1990.000000	46.000000	46.000000	8002.000000	8002.000000	7683.000000	7683.000000	880.000000	880.000000	2406.000000	2406.000000
50%	0.500000	2000.000000	53.000000	53.000000	8404.000000	8404.000000	8047.000000	8047.000000	1001.000000	1001.000000	2494.000000	2494.000000
75%	1.000000	2008.000000	58.000000	58.000000	8897.000000	8897.000000	8539.000000	8539.000000	1118.000000	1118.000000	2601.000000	2601.000000
max	1.000000	2016.000000	73.000000	73.000000	10371.000000	10371.000000	10328.000000	10328.000000	1512.000000	1512.000000	2972.000000	2972.000000

Figure D1: df.describe() Used on Final Data Frame



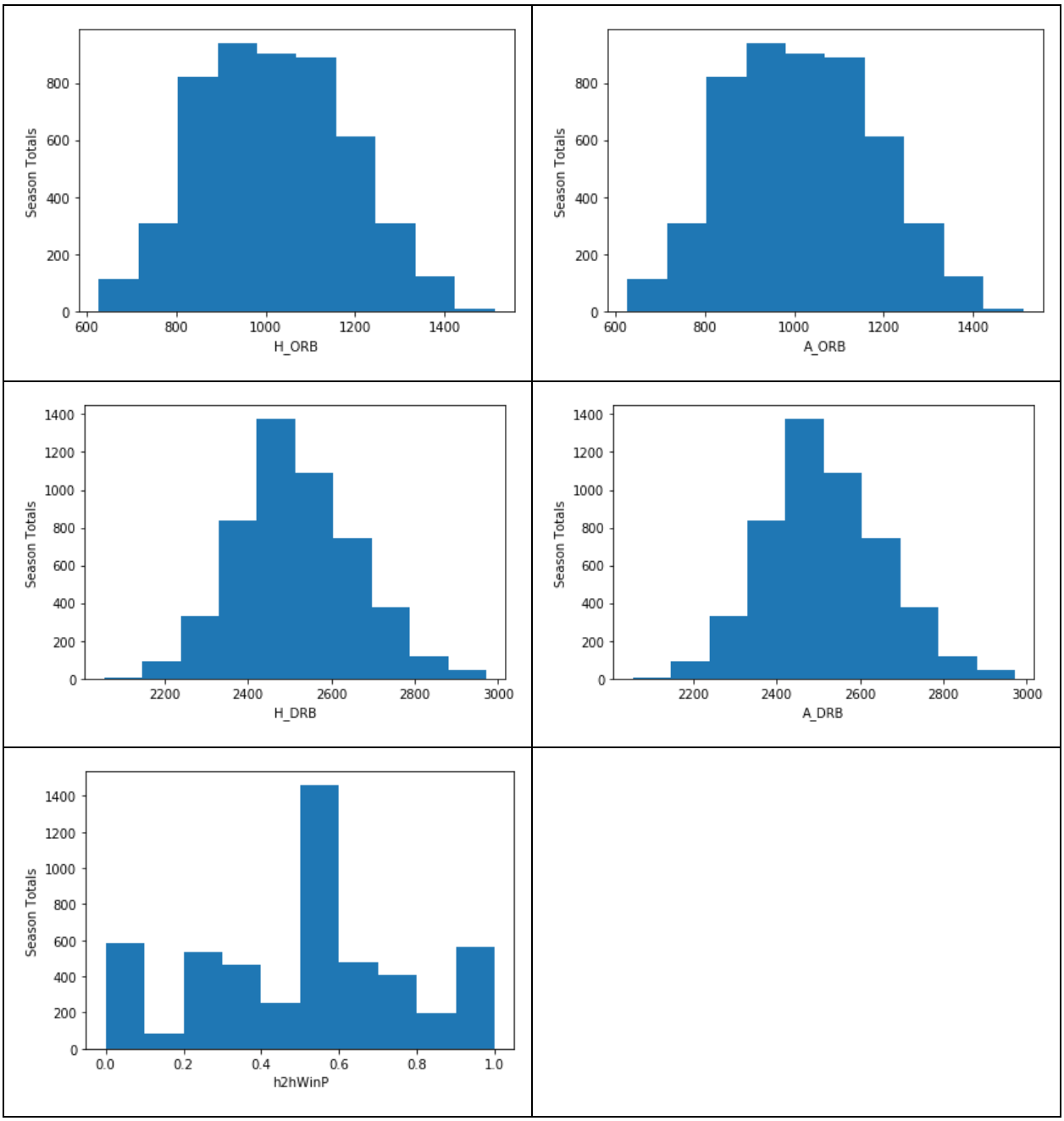
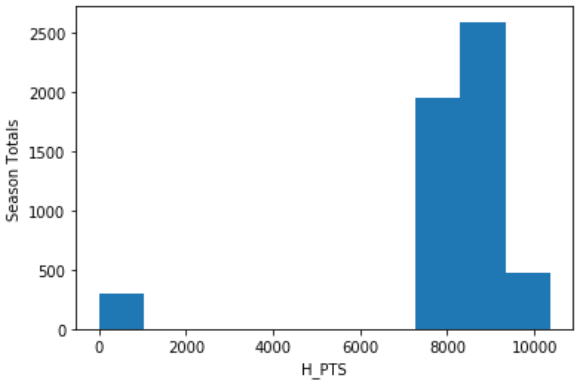
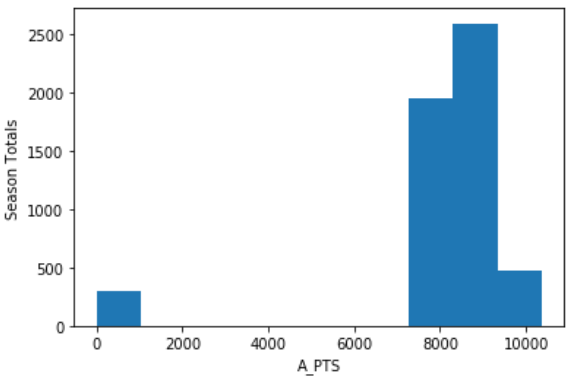


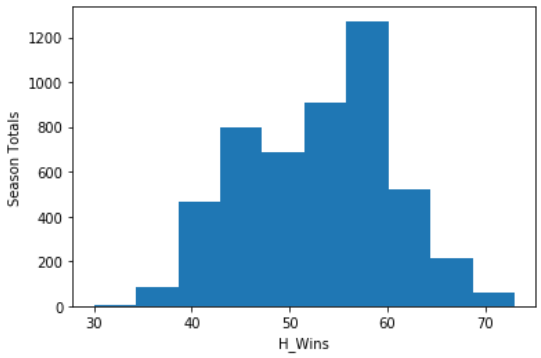
Figure D2: Regular Season Data Distributions (Excluding H_PTS and A_PTS)



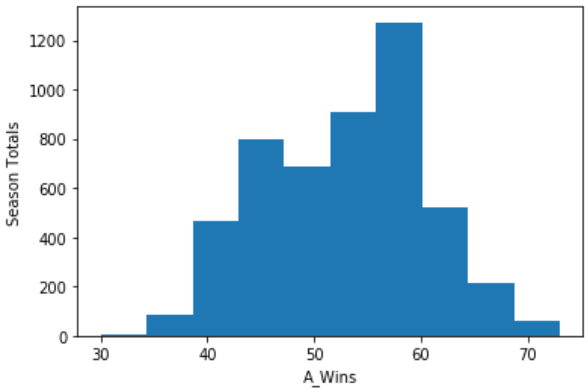
H_PTS with 0 values



A_PTS with 0 values



H_PTS with 0 values omitted



A_PTS with 0 values omitted

```
dfFinal = dfFinal[dfFinal.H_PTS != 0]
dfFinal = dfFinal[dfFinal.H_Wins != 0]
```

Figure D3: Regular Season Data Distributions of H_PTS and A_PTS With and Without 0 values

Appendix E: Initial Logistic Regression and CART Models

```
# Creating the x and y training and testing data
x_train, x_test, y_train, y_test =
train_test_split(dfFinal.drop(['game_id','game_date','homeTeam','homeTeamWins','awayTe
am','season'], 1), dfFinal['homeTeamWins'], test_size=0.30, random_state=5)
# Creating a logistic regression model
mdl = LogisticRegression(random_state=1)
mdl.fit(x_train, y_train)
train_score = mdl.score(x_train, y_train)
test_score = mdl.score(x_test, y_test)
print("The training score is: " + train_score.astype(str))
print("The testing score is: " + test_score.astype(str))
```

Output:

```
The training score is: 0.616
The testing score is: 0.622
```

Figure E1: Train Test Split and Initial Logistic Regression Model Creation

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import export_graphviz
```

```
# Train the CART model
for i in range(2,10):
  cart_model = DecisionTreeClassifier(random_state=3,max_depth=i)
  cart_model.fit(x_train, y_train)
  train_score = cart_model.score(x_train, y_train)
  test_score = cart_model.score(x_test, y_test)

  # Print out summary of model performance
  print('The score of this model over training data is {:.3f} and {:.3f} over the
testing data'.format(train_score, test_score))
```

Output:

```
The score of this model over training data is 0.620 and 0.604 over the testing data
The score of this model over training data is 0.629 and 0.620 over the testing data
The score of this model over training data is 0.644 and 0.596 over the testing data
The score of this model over training data is 0.658 and 0.590 over the testing data
The score of this model over training data is 0.668 and 0.595 over the testing data
The score of this model over training data is 0.681 and 0.590 over the testing data
The score of this model over training data is 0.692 and 0.569 over the testing data
The score of this model over training data is 0.705 and 0.563 over the testing data
```

Figure E2: Train Test Split and CART Model Creation

Appendix F: Head to Head Data Function

```
def head_to_head(season):
  head_to_head = {}

  # Only iterate through the third data frame which contains all games that have
occured
  for index, row in df3.iterrows():

     if row["season_year"] == season:

    # We only care about the season of interest that we input
     # If the team ID is not in the keys of the head to head dictionary
      if row.team_id not in head_to_head:

        head_to_head[row.team_id] = {row.a_team_id:[row.wl]}
        # Add the team as a key in head_to_head and an inner dictionary as the value
        # Add the a_team as the key of the inner dictionary
        # Add the "wl" attributes to a list as the value
      elif (row.team_id in head_to_head) & (row.a_team_id not in
head_to_head.get(row.team_id)):
        # Add the a_team_id as a key in the inner dictionary
        head_to_head[row.team_id][row.a_team_id] = [row.wl]

      elif (row.team_id in head_to_head) & (row.a_team_id in
head_to_head.get(row.team_id)):
        wl = head_to_head[row.team_id][row.a_team_id]
        wl.append(row.wl)
        head_to_head[row.team_id][row.a_team_id] = wl

  for hometeam in head_to_head:
    for awayteam in head_to_head[hometeam]:
      count = 0
      list_length = len(head_to_head[hometeam][awayteam])

      for i in range(list_length):
        if head_to_head[hometeam][awayteam][i] == "W":
          count += 1
      head_to_head[hometeam][awayteam] = count/list_length
  return head_to_head
```

Figure F1: Python Function used to calculate head to head data

Appendix G: Feature Engineering and Outputs

```
base_corrs = dfFinal.corr().homeTeamWins.drop(index='homeTeamWins')
for feature1 in base_corrs.index:
    for feature2 in base_corrs.index:
        if feature2 != feature1:
            # multiply the two features to create a new feature
            new_feature = x_train[feature1] / x_train[feature2]
            new_corr = np.abs(np.round(np.corrcoef(new_feature, y_train)[0,1], 3))
            corr1 = np.abs(np.round(base_corrs[feature1], 3))
            corr2 = np.abs(np.round(base_corrs[feature2], 3))
            # add a threshold of 0.02 to make sure that the improvement is meaningful
            if new_corr > max(corr1, corr2)+0.02:
                print('{} {} combine to get correlation {} compared to {} {}'.format(
                    feature1, feature2, new_corr, corr1, corr2))
```

Figure G1: Feature Engineering Using Division Operator

```
base_corrs = dfFinal.corr().homeTeamWins.drop(index='homeTeamWins')
for feature1 in base_corrs.index:
    for feature2 in base_corrs.index:
        if feature2 != feature1:
            # multiply the two features to create a new feature
            new_feature = x_train[feature1] - x_train[feature2]
            new_corr = np.abs(np.round(np.corrcoef(new_feature, y_train)[0,1], 3))
            corr1 = np.abs(np.round(base_corrs[feature1], 3))
            corr2 = np.abs(np.round(base_corrs[feature2], 3))
            # add a threshold of 0.02 to make sure that the improvement is meaningful
            if new_corr > max(corr1, corr2)+0.02:
                print('{} {} combine to get correlation {} compared to {} {}'.format(
                    feature1, feature2, new_corr, corr1, corr2))
```

Figure G2: Feature Engineering Using Subtraction Operator

Name of New Feature	Operation	Correlation to homeTeamWins (Target Variable)		
		Improved	Initial	Initial
hfgP	H_FG / H_FGA	0.088	0.04	0.015
afgP	A_FG / A_FGA	0.094	0.04	0.015
h2pP	H_2P / H_2PA	0.109	0.018	0.01
a2pP	A_2P / A_2PA	0.109	0.018	0.01
h3pP	H_3P / H_3PA	0.103	0.03	0.03
a3pP	A_3P / A_3PA	0.085	0.06	0.09
hFTP	H_FT / H_FTA	0.07	0.01	0.01
aFTP	A_FT / A_FTA	0.07	0.01	0.01

diffPTS	H_PTS - A_PTS	0.094	0.042	0.042
diffoppPTS	H_oppPTS - A_oppPTS	0.132	0.047	0.047

Table G1: New Feature and Feature Improvement

```
def new_f_div(f1, f2, name):

  dfFinal[name] = dfFinal[f1] / dfFinal[f2]

def new_f_sub(f1, f2, name):

  dfFinal[name] = dfFinal[f1] - dfFinal[f2]

new_f_div('H_FG', 'H_FGA', 'hfgP')
new_f_div('A_FG', 'A_FGA', 'afgP')
new_f_div('H_2P', 'H_2PA', 'h2pP')
new_f_div('A_2P', 'A_2PA', 'a2pP')
new_f_div('H_3P', 'H_3PA', 'h3pP')
new_f_div('A_3P', 'A_3PA', 'a3pP')
new_f_sub('H_PTS', "A_PTS", 'diffPTS')
new_f_sub('H_oppPTS', "A_oppPTS", 'diffoppPTS')
new_f_div('H_FT', 'H_FTA', 'hFTP')
new_f_div('A_FT', 'A_FTA', 'aFTP')
dfFinal = dfFinal.drop(columns=["H_FG", "H_FGA", 'A_FG', 'A_FGA', 'H_2P', 'H_2PA',
'A_2P', 'A_2PA', 'H_3P', 'H_3PA', 'A_3P', 'A_3PA', 'H_PTS', 'A_PTS', "H_oppPTS",
"A_oppPTS"])
dfFinal = dfFinal.drop(columns=["H_AST", "A_AST"])
dfFinal = dfFinal.drop(columns=['H_FT', 'A_FT', 'H_FTA', 'A_FTA'])
```

Figure G3: Adding Engineered Features to the Final Data Frame and Removing Original

Appendix H: Second Iteration of the Logistic Regression Model

```
x_train, x_test, y_train, y_test =
train_test_split(dfFinal.drop(['game_id','game_date','homeTeam','homeTeamWins','awayTe
am'], 1), dfFinal['homeTeamWins'], test_size = 0.30, random_state = 5)
# Creating a logistic regression model
mdl = LogisticRegression(random_state = 1)
mdl.fit(x_train, y_train)
train_score = mdl.score(x_train, y_train)
test_score = mdl.score(x_test, y_test)
print("The training score is: " + train_score.astype(str))
print("The testing score is: " + test_score.astype(str))
```

Figure H1: Train Test Split and Final Logistic Regression Model Creation

Appendix I: Bracket Function Code

```
[20] def bracket(df, df2, year, mdl):
       df2 = df2[df2.season == year]

       df = df[df.SeasonEnd == year]
       df = df[df.Rank < 9]

       ET1 = {}   #dictionary for the eastern playoff teams --> key is the team abbrv. and value is the rank of the team
       ET2 = {}
       WT1 = {}   #dictionary for the western playoff teams --> key is the team abbrv. and value is the rank of the team
       WT2 = {}

       #filling up the dictionaries
       for j in range (0,16):     # iterating through the top 8 teams of each conference (total of 16 teams)

         #eastern conference
         if(df.Conference.iloc[j] == 0):

           team = df.Team.iloc[j]
           rank = df.Rank.iloc[j]

           if(rank == 1 or rank == 8 or rank == 4 or rank == 5):
             ET1[team] = rank
           else:
             ET2[team] = rank

         #western conference
         if(df.Conference.iloc[j] == 1):

           team = df.Team.iloc[j]
           rank = df.Rank.iloc[j]

           if(rank == 1 or rank == 8 or rank == 4 or rank == 5):
             WT1[team] = rank
           else:
             WT2[team] = rank


       #Now that we have our dictionaries filled, we start playing teams against each other
       cnt = 0 #this variable is meant to keep track of which round of the playoffs we're looking at (in total 4 rounds)

       while(cnt < 2):
         ET1, ET2 = playoffs(ET1, ET2, df, df2, mdl)
         WT1, WT2 = playoffs(WT1, WT2, df, df2, mdl)
         cnt += 1

       newET = {}
       newET.update(ET1)
       newET.update(ET2)
       ECF = list(newET.keys())
```

```
newWT = {}
newWT.update(WT1)
newWT.update(WT2)
WCF = list(newWT.keys())

winner, loser, winnerW, loserW = matchup(ECF[0], ECF[1], df, df2, mdl)
#print(winner + " beat " + loser + " " + str(winnerW) + " to " + str(loserW))

del newET[loser]

winner, loser, winnerW, loserW = matchup(WCF[0], WCF[1], df, df2, mdl)
#print(winner + " beat " + loser + " " + str(winnerW) + " to " + str(loserW))

del newWT[loser]

finals_dict = {}
finals_dict.update(newET)
finals_dict.update(newWT)
Finals = list(finals_dict.keys())

winner, loser, winnerW, loserW = matchup(Finals[0], Finals[1], df, df2, mdl)
#print(winner + " beat " + loser + " " + str(winnerW) + " to " + str(loserW) + " to win the NBA Finals")



return winner
```

Appendix J: Playoffs Function Code

```
[21] def playoffs(dict1, dict2, df, df2, mdl):
     #this should return a dictionary with the remaining teams after the given round (i.e. if a team loses, remove them)

       newD1 = dict1.copy()
       newD2 = dict2.copy()

       cnt = len(dict1)/2       #dictionary should always be of a size that is a multiple of 2 (4,2,1) --> when it gets to 1, the dictionary returned will be the winner of that conference
       i = 0

       while(i < cnt):      #going through each matchup 1 by 1

         #Getting the maximum and minimum ranks left in the dictionary --> makes the 1st seed play the 8th, then 2nd play the 7th...
         highRank1 = min(dict1.keys(), key=(lambda k: dict1[k]))
         lowRank1 = max(dict1.keys(), key=(lambda k: dict1[k]))
         highRank2 = min(dict2.keys(), key=(lambda k: dict2[k]))
         lowRank2 = max(dict2.keys(), key=(lambda k: dict2[k]))

         winner1, loser1, winner1W, loser1W = matchup(highRank1, lowRank1, df, df2, mdl)
         winner2, loser2, winner2W, loser2W = matchup(highRank2, lowRank2, df, df2, mdl)

         #print(winner1 + " beat " + loser1 + " " + str(winner1W) + " to " + str(loser1W))
         #print(winner2 + " beat " + loser2 + " " + str(winner2W) + " to " + str(loser2W))

         #If I'm deleting both the winner and loser from the dictionaries, then I need to create a new dictionary to store the winners --> will return the new dictionaries
         del dict1[winner1]
         del dict1[loser1]
         del dict2[winner2]
         del dict2[loser2]

         #Keeping track of the winners by deleting just the losers --> these are the dictionaries that will be returned
         del newD1[loser1]
         del newD2[loser2]

         i += 1


       return newD1, newD2
```

Appendix K: Playoffs Function Code

```
def matchup(t1, t2, df, df2, mdl):
  #now we need to populate the dataframe before we get to predicting who's going to win
  #we may need a for loop to iterate through the rows of df
  for i in range(0,16):
    if df.iloc[i].Team == t1:
      H_Wins = df.iloc[i].W
      H_ORB = df.iloc[i].ORB
      H_DRB = df.iloc[i].DRB
      hfgP = df.iloc[i].FG / df.iloc[i].FGA
      h2pP = df.iloc[i]._2P / df.iloc[i]._2PA
      h3pP = df.iloc[i]._3P / df.iloc[i]._3PA
      hFTP = df.iloc[i].FT / df.iloc[i].FTA

    if df.iloc[i].Team == t2:
      A_Wins = df.iloc[i].W
      A_ORB = df.iloc[i].ORB
      A_DRB = df.iloc[i].DRB
      afgP = df.iloc[i].FG / df.iloc[i].FGA
      a2pP = df.iloc[i]._2P / df.iloc[i]._2PA
      a3pP = df.iloc[i]._3P / df.iloc[i]._3PA
      aFTP = df.iloc[i].FT / df.iloc[i].FTA

  for x in range(0,16):
    for y in range(0,16):
      if df.iloc[x].Team == t1 and df.iloc[y].Team == t2:
        diffPTS = df.iloc[x].PTS - df.iloc[y].PTS
        diffoppPTS = df.iloc[x].oppPTS - df.iloc[y].oppPTS

  h2hWinP = -1
  for j in range(0,df2.season.count()):
    if df2.Team1.iloc[j] == t1 and df2.Team2.iloc[j] == t2:
      h2hWinP = df2.h2hWinP.iloc[j]

  if h2hWinP == -1:
    h2hWinP = 0.5

  tempDF3 = pd.DataFrame({"H_Wins":[H_Wins], "A_Wins":[A_Wins], "H_ORB":[H_ORB], "A_ORB":[A_ORB], "H_DRB":[H_DRB], "A_DRB":[A_DRB], "h2hWinP":[h2hWinP], "hfgP":[hfgP],
                          "afgP":[afgP], "h2pP":[h2pP], "a2pP":[a2pP], "h3pP":[h3pP], "a3pP":[a3pP], "diffPTS":[diffPTS], "diffoppPTS":[diffoppPTS], "hFTP":[hFTP], "aFTP":[aFTP]})
  #now we can predict the probability each team has at winning a given game
  t1WinProb = mdl.predict_proba(tempDF3)[0,1]
  t2WinProb = mdl.predict_proba(tempDF3)[0,0]

  #Now we need to simulate a best of 7 series
  t1Wins = 0
  t2Wins = 0

  while(t1Wins < 4 and t2Wins < 4):
    threshold = np.random.random()
    if(threshold < t1WinProb):
      t1Wins += 1
    else:
      t2Wins += 1

  if(t1Wins == 4):
    return t1, t2, t1Wins, t2Wins
  else:
    return t2, t1, t2Wins, t1Wins
```

Appendix L: Histogram Simulation Code for the 2016 Playoffs

```
[23] winners = {}
     teams = []
     dP = dfPlayoffs.copy()

     dP = dP[dP.SeasonEnd == 2016]
     dP = dP[dP.Rank < 9]


     for k in range(0,16):
       teams.append(dP.Team.iloc[k])


     for x in range(0,16):
       winners.update({teams[x] : 0})


     for z in range(0,300):
       winner = bracket(dfPlayoffs, dfH2H, 2016, mdl)
       for y in range(0,16):
         team = teams[y]
         if(team == winner):
           winners.update({team : winners.get(team) + 1})
```

```
[24] print(winners)
     plt.figure(figsize=(20, 3))
     plt.bar(winners.keys(), winners.values(), width=0.5, color='r')
```

Appendix M: Playoff Winner Distributions

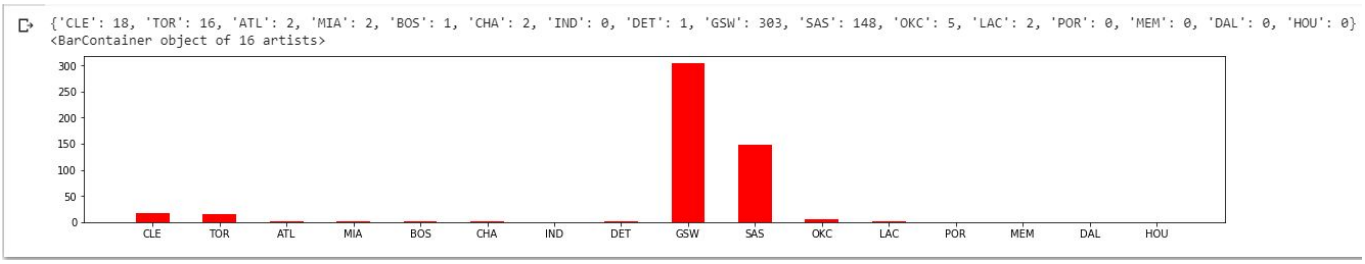


Figure M1. 2016 Playoff Simulation Output

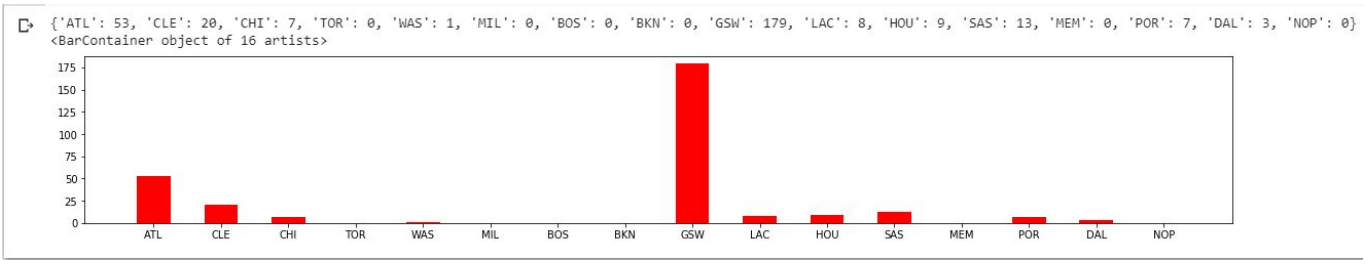


Figure M2. 2015 Playoff Simulation Output

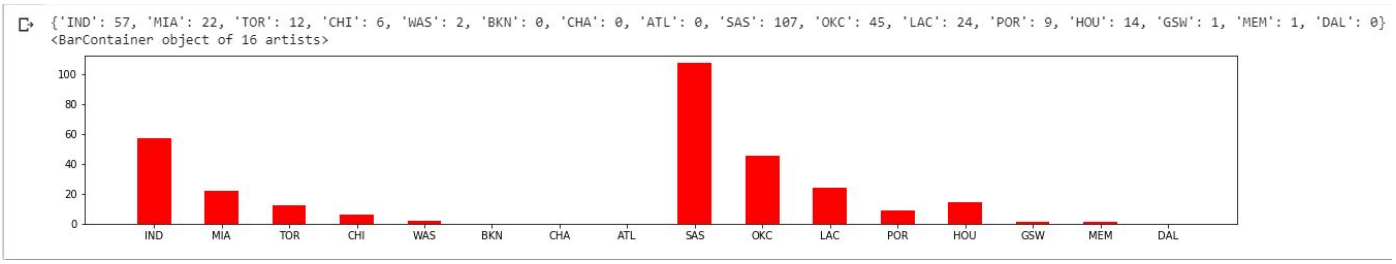


Figure M3. 2014 Playoff Simulation Output

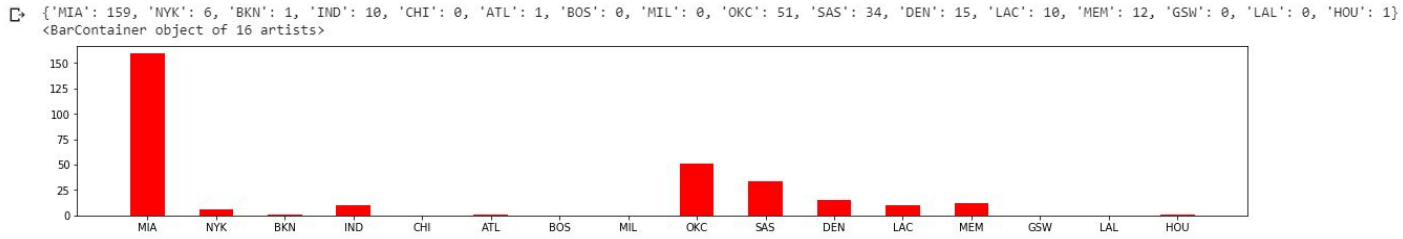


Figure M4. 2013 Playoff Simulation Output

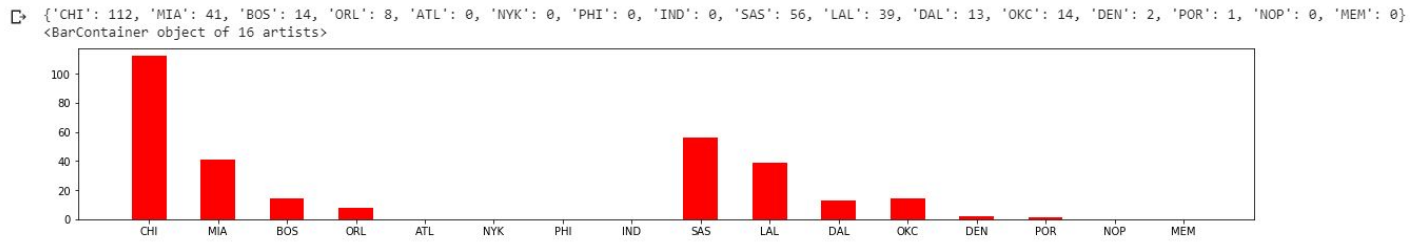


Figure M5. 2011 Playoff Simulation Output

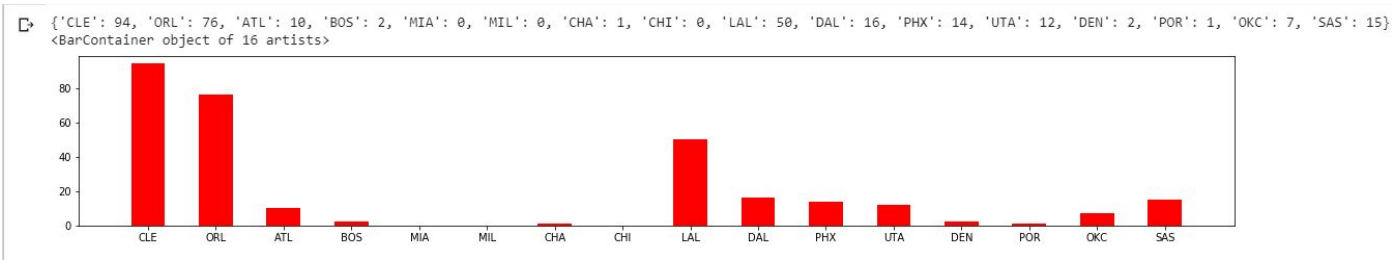


Figure M6. 2010 Playoff Simulation Output

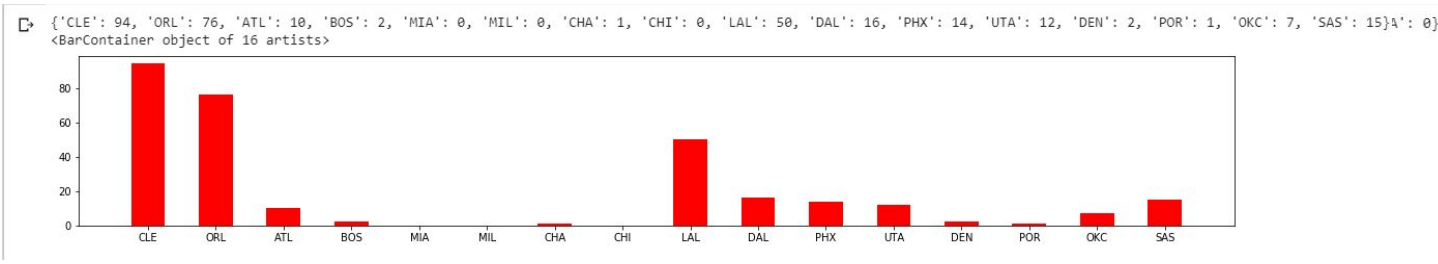


Figure M7. 2009 Playoff Simulation Output

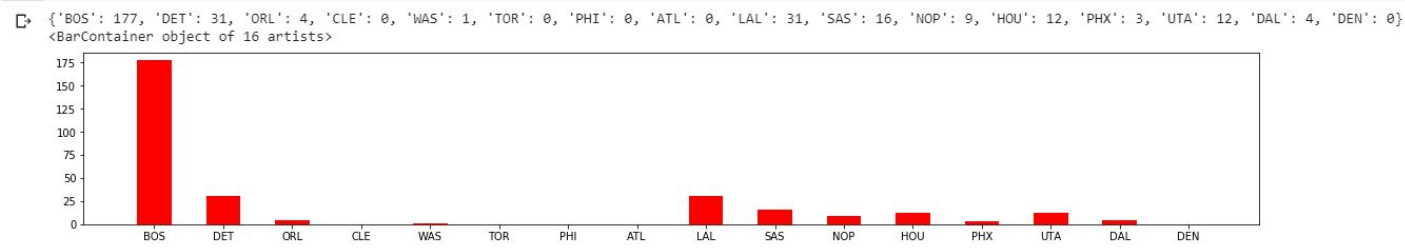


Figure M8. 2008 Playoff Simulation Output

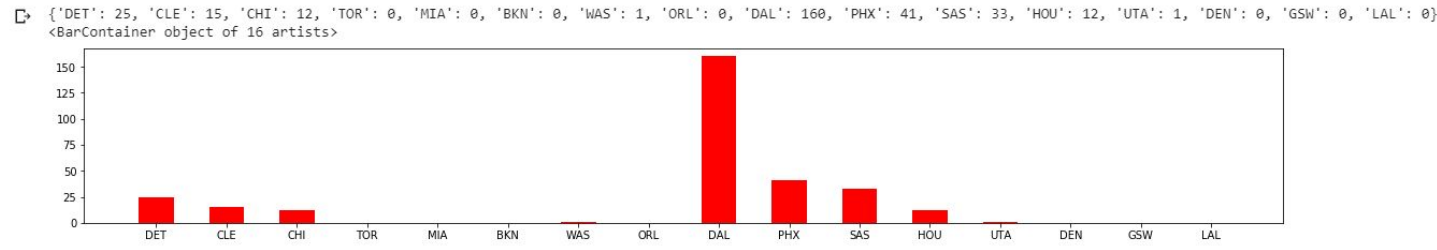


Figure M9. 2007 Playoff Simulation Output

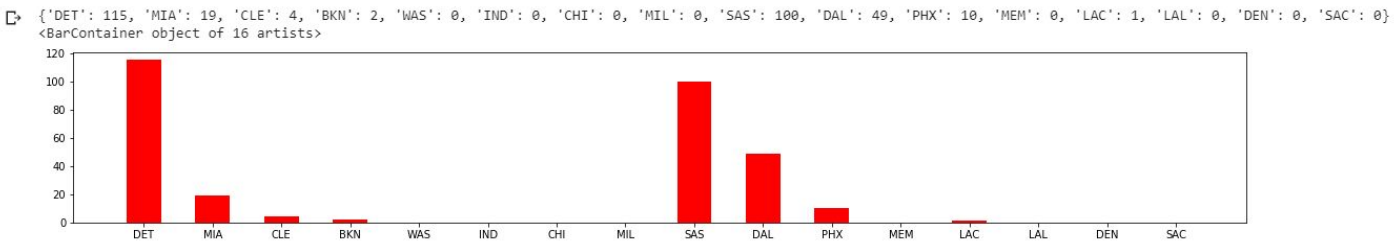


Figure M10. 2006 Playoff Simulation Output