A MACHINE LEARNING APPROACH TO PREDICTING NBA GAMES

WEDNESDAY, MAY 12, 2021

BENNETT COHEN
CHRISTOPHER LANDGREBE
ARMAN VAZIRI
LEONARDO BIRAL
CALVIN SHUESTER

Introduction to Machine Learning
Department of Industrial Engineering & Operations Research
University of California, Berkeley

Contents

1 Introduction **3**

2 Data Collection and Feature Engineering **4**

i Data Collection 4

ii Feature Engineering 5

2.1 F1: Cumulative Team Statistics 5

2.2 F2: Momentum & Recency 6

2.3 F3: Cumulative Player Statistics 7

2.4 F4: Head to Head Statistics 7

2.5 F5: X-Factor Features 8

2.5.1 NBA Awards 8

2.5.2 The "LeBron Effect" 8

2.5.3 All-NBA Teams Selections 9

2.5.4 Home Court Advantage 9

2.5.5 Altitude and Timezone Difference for Away Team 9

2.5.6 Vegas Betting Odds 10

3 Building Models to Predict Individual NBA Games **11**

i Approach #1: Modified 5-Fold Cross-Validation 11

1.1 Model #0: Baseline Model 11

1.2 Model #1: Logistic Regression 12

1.3 Model #2: Random Forest 12

1.4 Model #3: XGBoost with a Sequential 5-Fold Cross-Validation Process 14

1.5 Model #4: Neural Network (Multilayer Perceptron Classifier) 16

1.6 Final Model Outputs 17

ii Approach #2: Hold-Out Validation Set and Blending 17

4 Conclusion and Extensions **19**

5 Appendix #1: Cumulative Statistic Engineering **20**

6 Appendix #2: X-Factor Feature Engineering **38**

7 Appendix #3: Approach #1 (Modified Cross-Validation) Modeling and Analysis **52**

8 Appendix #4: Approach #2 (Hold-Out Validation Blending Modeling and Analysis **72**

Introduction

NBA betting is a big business, accounting for upwards of $500 million in legal bets placed on every month games throughout its 82 game season. With this in mind, we wanted to explore how we could maximize our betting profits by making more accurate predictions on the outcomes of games through machine learning. We searched for the most thorough and reliable data available, eventually choosing to use the NBA's own highly trusted statistical database website https://www.nba.com/stats/, along with a sports betting research website called from https://www.sportsbookreviewsonline.com). As is discussed in later sections, the raw data lends no feasible value to our task because game data only is available after a game outcome is known, so we spent considerable time cleaning and processing the data to curate statistical features. Using these features, we began implementing machine learning techniques (Logistic Regression, Random Forest, Gradient Boosting, and Neural Networks) to compare against a baseline model of predicting the home team to win in recent NBA seasons. We first implemented a modified 5-Fold Cross-Validation approach on all of our models, followed by an exploration into using a hold-out validation set and blending approach on the predictions from our models to see if we could improve our accuracy (primary performance metric). Overall, we found success with our modeling techniques, and our best performing models performed significantly higher than a baseline model and many pre-existing models. As will be discussed, our largest area for improvement is in feature engineering, which could take this model from exploration into true practical feasibility as a betting engine. The following sections outline our processes of data collection , feature engineering, and modeling, and the appendices that follow provide the codebase and technical explanations of these techniques in greater detail

Data Collection and Feature Engineering

i Data Collection

The data used in this project has been extracted from the NBA's own statistics website (Link: https://www.nba.com/stats/) along with https://www.sportsbookreviewsonline.com. There are ten main data sets, which we later combine in our feature engineering section. There are 10 data sets (some of which created by us for ease), which are described below:

1. GAMES ⟶ the box score of all games between the 2007 and 2018 season, along with the winner of each game.
2. DETAILS ⟶ the box score statistics for each player in each game.
3. PLAYERS ⟶ every NBA player between 2007 and 2018, and what team they played for each season.
4. TEAMS ⟶ statistics about the team itself, including location, arena capacity, founding data, and coach.
5. ODDS ⟶ the winning odds for the home and away team for each game.
6. AWARDS ⟶ a data set we created ourselves for the MVP, DPOY, ROY, and 6MOY in each season of data in the provided sets.
7. FIRST TEAM ⟶ a data set we created ourselves that lists the 5 players on the All-NBA 1st Team in each season of data in the provided sets.
8. SECOND TEAM ⟶ a data set we created ourselves that lists the 5 players on the All-NBA 2nd Team in each season of data in the provided sets.

9. THIRD TEAM ⟶ a data set we created ourselves that lists the 5 players on the All-NBA 3rd Team in each season of data in the provided sets.

10. ALTITUDE-TIMEZONE ⟶ a data set we created ourselves that lists the altitude and timezone of each NBA team's home arena.

For clarity and, we won't show what each of these data sets looks like in their raw form, but instead will show how we use these data sets in conjunction to construct our features to build models off of.

ii Feature Engineering

Our models are based off the following five feature types (F2, F2, F3, F4, and F5), which are the various factors that we believe might influence which team wins a game. We begin with the most common features and slowly add more unique features to create a more robust and potentially accurate model. We don't do feature selection here, and save analyzing feature significance for later sections.

2.1 F1: Cumulative Team Statistics

First, we look at the GAMES data. The primary issue here is that these features don't actually help make predictions because the box score only is known *after* the game is over. If we were to use these as features, our models would have 100% accuracy but would be infeasible. However, what probably does matter to predicting which team will win is the cumulative box score, or the average statistics for a team in every game up until the game we try to predict. This is built off of the assumption that if a hypothetical home team has performed better than the away team across all games earlier in the season, they should be more likely to win this game, and vice versa. For instance, using the average points-per-game (abbreviated to "PPG"), if PPG_1 = 110.6 and PPG_2 = 96.6, Team #1 is likely to score more points than Team #2, and would win the game. Our GAMES data includes the statistics for the home and away team separately, which leads us to make a key assumption: teams tend to perform differently when they play at home than when they play away. This is touched upon in F5 and can be due to various factors (time zone, traveling, altitude, etc). As a result, in a given example above, PPG_1 = 110.6 means that

when Team #1 is the home team, they average 110.6 points per game. It does not mean Team #1 average 110.6 points across all games.The box-score statistics we implement are below (the same features are used for the away team):

Statistic	Meaning
WIN_PCT_home	Percentage of Games Won at Home
PTS_home	Avg. Points-per-Game
FG_PCT_home	Avg. Percentage of Field Goals/Baskets Made
FT_PCT_home	Avg. Percentage of Free Throws Made
FG3_PCT_home	Avg. Percentage of 3 Pointers Made
AST_home	Avg. Number of Assists
REB_home	Avg. Number of Rebounds

2.2 F2: Momentum & Recency

Beyond simple season-long cumulative statistics, we know that hot-streaks and cold-streaks are extremely common in sports. A team can get "hot" at the right time, winning many games in a row, so if a team has a bad start to the season, but is very "hot" at the time of a game we are trying to predict, simply using the season statistics wouldn't be reflective of their current level of play (the same is of course true for a cold streak). This is where the idea of momentum and recency comes from; we need to create features to capture teams most recent performances. To do this, we simple create the same exact features as we did in F1, but not include the entire season. We aren't sure exactly what length of time is considered significant to capture a team's momentum, so we create features for the previous five and previous ten games separately. It's possible one of these is shown to be more significant later on, but because this is rather simple to implement, we do both time frames. Again, we use the seven statistics from the table above. This increases our feature count to $2 \times (7 + 7 + 7) = 42$ features. There is potential collinearity between these because the information for the last five games is captured in the last ten statistics, which is also captured in the season-long statistics, but we don't need to address this just yet until we see how the models perform.

2.3 F3: Cumulative Player Statistics

Delving deeper yet again, we also want to consider how each player performs in each game to potentially unveil certain trends and increase our model performance. For F1 and F2, we only looked the the data from the GAMES (and TEAMS in order to turn the TEAM_ID values into team names), but now we consider the more complicated DETAILS data, which consists of the box-score statistics for each player on each team in each game. This data is much larger than the other two data sets. Another assumption we make is that the majority of the most popular and predictive box score statistics are performed by the starting five of a team (2 Guards, 2 Forwards, 1 Center) and that the contributions of the bench players aren't as important. Further, this makes some intuitive sense in that if a "superstar" player such as Kevin Durant scores 60 points in a game, his team is more likely to win and the bench players probably did less in that game because there is limited time to score that many points. This allows us to simplify our data set and only consider five players from each team. Also, we don't actually use the names of the starters, but their position instead, which we simplify further because the data doesn't distinguish between the two types of guards and forwards, leaving us with only three positions to consider. We then use the same functions from earlier to calculate cumulative season-long and last-five team box score stats, but for each type of player.These statistic definitions are shown below:

2.4 F4: Head to Head Statistics

Another factor of NBA games to consider is the notion that certain teams perform differently against different teams regardless of their cumulative statistics due to the different styles of play. For instance, if Team #1 tends to shoot many perimeter shots (long range shots such as three pointers) and if their opponent's (Team #2) defense doesn't defend well against these types of shot attempts, there should be some inherent advantage to Team #1, but this isn't currently reflected in our features. It's very difficult to identify a team's style of play because it can vary based on their opponent, so we get around this by simple creating statistics for the exact combination of home and away teams; in other words, the cumulative statistics in these head-to-head match ups throughout the season.

2.5 F5: X-Factor Features

For our final feature type, we will consider factors that don't translate into quantitative measures as easily. The features detailed below are ones that are less conventional than the features created by the previous 4 types, but they are ones that we believe have the possibility of predictive power as well.

2.5.1 NBA Awards

Each season the National Basketball Association hands out various awards to deserving league players. Four of the most recognized awards are

1. Regular Season Most Valuable Player (MVP)
2. Defensive Player of the Year (DPOY)
3. Rookie of the Year (ROY)
4. Sixth Man of the Year (6MOY)

Players who win these awards typically exhibit a level of skill, leadership, and overall player quality beyond what statistical metrics may contain. If a player won one of these awards in recent years, we can make the assumption they are providing a similar level of skill and quality for their team now, though this benefit decreases over time so we decide to only look at the last four years (i.e. the "prime" of an NBA player's career is short enough such that they probably are out of it in five years). We use the AWARDS data set we created to calculate the number of players on the starting five of the home/away roster that won one of these awards in the last four years, creating eight new features (four awards for both home and away).

2.5.2 The "LeBron Effect"

Over the course of the last 15+ years, LeBron James has proven that he is a transcendent player, providing a level of skill and leadership beyond any of the awards above (he has often lost awards simply because voters don't want to give it to him even if he is the obvious choice). To capture his value to a team, we create a binary variable for both home and away to reflect if he is on the starting roster at the time of the game.

2.5.3 All-NBA Teams Selections

At the end of each season the NBA chooses the All-NBA First, Second, and Third Teams. Each of these "teams" is made up of five players, where the All-NBA First Team is the Top 5 players in the league according to the NBA (position included), the All-NBA Second Team is the next best 5 players, and so on. We calculate the same counts for these teams as we did above with awards in the last four years though this feature expands the range of values because we are looking at a five player roster instead of one player. We make the same assumptions as earlier.

2.5.4 Home Court Advantage

Of the four major U.S. sports leagues (NBA, NFL, MLB, NHL), home court advantage is most meaningful and significant in the NBA. As we look at our baseline model, this will become clear, but the home team wins in the NBA in our data set around 59% of the time, a number much higher than in other leagues. That being said, home court advantage varies by team based on many factors and in wha ttime period. As such, we quantify a team's home court advantage as the winning percentage of the home team (when they play at home) over the last four years.

2.5.5 Altitude and Timezone Difference for Away Team

Throughout the NBA season, teams trave all over the country (and out) to play at various arenas, in various timezones and at different altitudes. A significant change in timezone (like for an east coast team playing on the west coast or vice versa) could *potentially* handicap the away team. This could be due to adverse effects of traveling or maybe a negative effect on the away team's sleep schedule. Additionally a drastic change (particularly a drastic *increase*) in elevation could also have a significant effect on the away team's ability to perform as well as they typically do. Many Studies have examined elevation and found that physical activities tend to be more difficult at higher elevations, Meaning that the home team may be acclimated to the elevation. We believe that our current data doesn't sufficiently represent the effects of these two factors so this in mind, we designed two features to hopefully accurately represent their effects. These features are the change in elevation and change in timezone for the away team from their own arena. For instance, if the Knicks travel to Los Angeles to play the Lakers, these features would be the change in elevation between the arenas, along with -3 because Los Angeles is three hours behind New York.

2.5.6 Vegas Betting Odds

Our project is motivated by sports betting, which relies on Vegas bookmakers to set the odds off each team winning in order for bettors to make bets. We believe that these bookmakers use large data sets and robust models to set these odds to maximize their profits (as they are businesses), and their "predictions" about the game are summarized by the odds they make. We can't access their data, but we can access the odds, so we add a feature for Vegas betting odds for the home and away team based off of the ODDS data set.

Building Models to Predict Individual NBA Games

Before building models, we need to actually create training and testing data. Because this model will hopefully be used to predict future games (i.e. the time component of the data is significant). Our training data will be data from the 2007 Season to the 2016 Season, and our testing data will be from the 2017 and 2018 Season. This corresponds to 83% training and 17% testing. One thing we want to note here is that we tried the same approach but with randomly split training and testing data, and achieved accuracy values ≈ 2.5-3% higher than time-based splitting for the best models, but it doesn't make sense to simply split randomly if our goal is to predict future games (i.e. this accuracy increase isn't relevant).

i Approach #1: Modified 5-Fold Cross-Validation

1.1 Model #0: Baseline Model

For our baseline model, we introduce a Dummy Classifier to predict the most common label in the training dataset, which is *HOME_TEAM_WINS* = 1. In the training set, the home team wins ≈ 58.8% of games. An alternative would be to pick the team with a higher win percentage but we choose to follow the metric from class. We also create encoded variables from our categorical variables here.

Baseline Model Confusion Matrix and Performance.

	HOME WIN	AWAY WIN
PRED. HOME WIN	0	1081
PRED. AWAY WIN	0	1545

Table 3.1: Confusion Matrix

	Test Set Statistic
Accuracy	0.5883
TPR	1.0000
FPR	1.0000

Table 3.2: Performance Metrics

1.2 Model #1: Logistic Regression

For our first model after the baseline, we implement regularized Logistic Regression using Sci-Kit Learn. Because there are 832 features used in our model (many of which are likely highly correlated, such as Field Goals Made and Points Per Game), it's possible we will be overfitting the noise of these features. We will be doing feature selection with our Random Forest and XGBoost Models later, but beacuse we aren't sure which have the most predictive power yet, we will use regularization to prevent overfitting.

Logistic Regression Confusion Matrix and Performance.

	HOME WIN	AWAY WIN
PRED. HOME WIN	458	623
PRED. AWAY WIN	275	1270

Table 3.3: Confusion Matrix

	Test Set Statistic
Accuracy	0.6580
TPR	0.8220
FPR	0.5763

Table 3.4: Performance Metrics

1.3 Model #2: Random Forest

As we move to tree based models, we first recognize that our data set and computation power limits us both for cross-validation and using a hold-out validation set. On one hand, our data is too large (12,835 x 832) to do cross-validation on a large grid of values in a reasonable time frame. On the other side, however, we aren't sure there is enough data to feel comfortable enough using a hold-out validation set to tune hyperparameters and reduce overfitting. Instead, we will first use reasonable values for hyperparameters to run a Random Forest on the entire data

set. Then, we will look at Feature Importances, and select the features with the most predictive power under this model (where predictive power is defined as the mean and standard deviation of accumulation of the impurity decrease within each tree for each feature). The motivation behind this dimensionality reduction is that there are many features we've created that are highly correlated, so our model is overfitting to the noise of our training set. We then perform a grid search on the *max_features* hyperparameter (the number of features to consider at each split), which runs approximately 100x faster because we never consider a high number of features. In an ideal world, we will see our performance actually increase from our "Reasonable Model." Our "reasonable" outputs the following test set performance metrics, along with the a bar plot of the Top 30 Feature Importances are shown below.

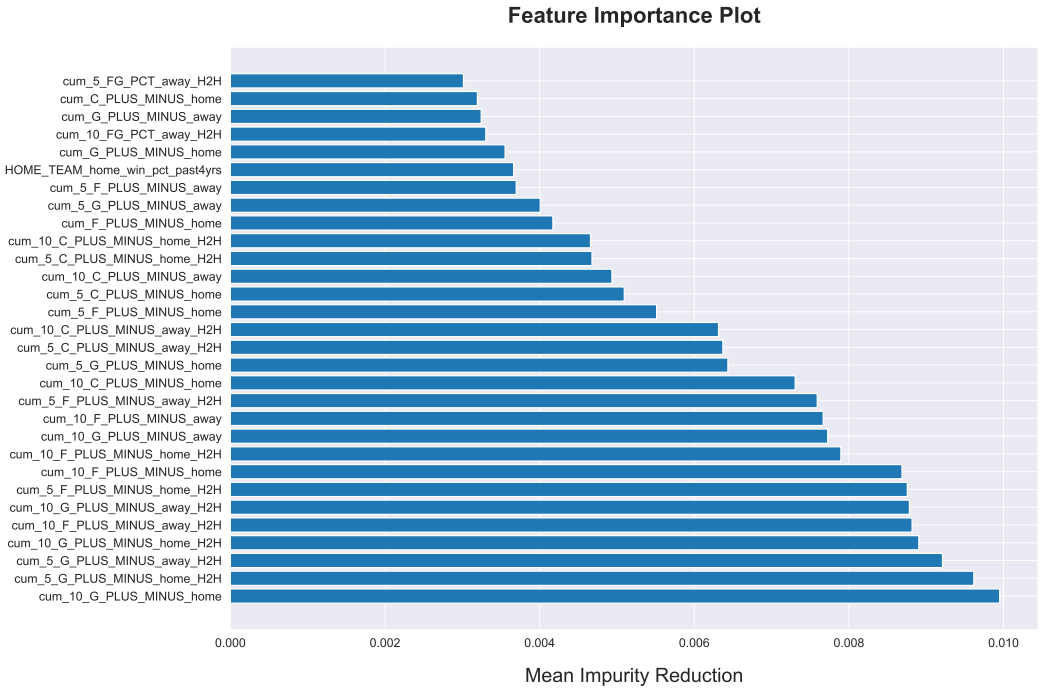
Reasonable Random Forest Confusion Matrix and Performance.

	HOME WIN	AWAY WIN
PRED. HOME WIN	430	651
PRED. AWAY WIN	267	1278

Table 3.5: Confusion Matrix

	Test Set Statistic
Accuracy	0.6504
TPR	0.8272
FPR	0.6022

Table 3.6: Performance Metrics



After using this subset of features, we using GridSearchCV and find out optimal *max_features* = 1. We run our best estimator model on the test set and output the following results. Appendix #3 has the code for cross-validation.

Cross-Validated Random Forest Confusion Matrix and Performance.

	HOME WIN	AWAY WIN
PRED. HOME WIN	481	600
PRED. AWAY WIN	302	1243

Table 3.7: Confusion Matrix

	Test Set Statistic
Accuracy	0.6565
TPR	0.8045
FPR	0.5550

Table 3.8: Performance Metrics

1.4 Model #3: XGBoost with a Sequential 5-Fold Cross-Validation Process

For our boosting model, we use the XGBoost library, and perform a series of grid searches to perform cross-validation to reduce overfitting and improve test set performance. This greedy process and code base for these steps were taken from former Columbia University Masters

Student and Spotify ML Engineer Aarshay Jain (Link: https://www.analyticsvidhya.com). Appendix #3 outlines the sequential cross-validation process, and it drastically reduces our run time by first picking an optimal $n\_estimators$ for learning rate = 0.1, then cross-validating other hyperparameters one by one and picking the optimal at each step, finally decreasing our learning rate to 0.005 at the end. These optimal parameters are shown below. Note that the optimal regularization parameter was found to be very high. This hyperparameter penalizes features which increase the cost and reduce overfitting, indicating that feature selection is probably also useful for this model to reduce is further.

Hyperparameter	Optimal Value
learning_rate	0.005
n_estimators	1142
max_depth	2
min_child_weight	4
gamma	0.0
subsample	0.8
colsample_bytree	0.8
reg_alpha	100

Figure 3.1: Optimal hyperparameters for XGBoost

As with the Random Forest model, we run our best estimator model, and calculate the test set performance statistics. We also use the Feature Importance score to find the Top 30 most importance features in this model.

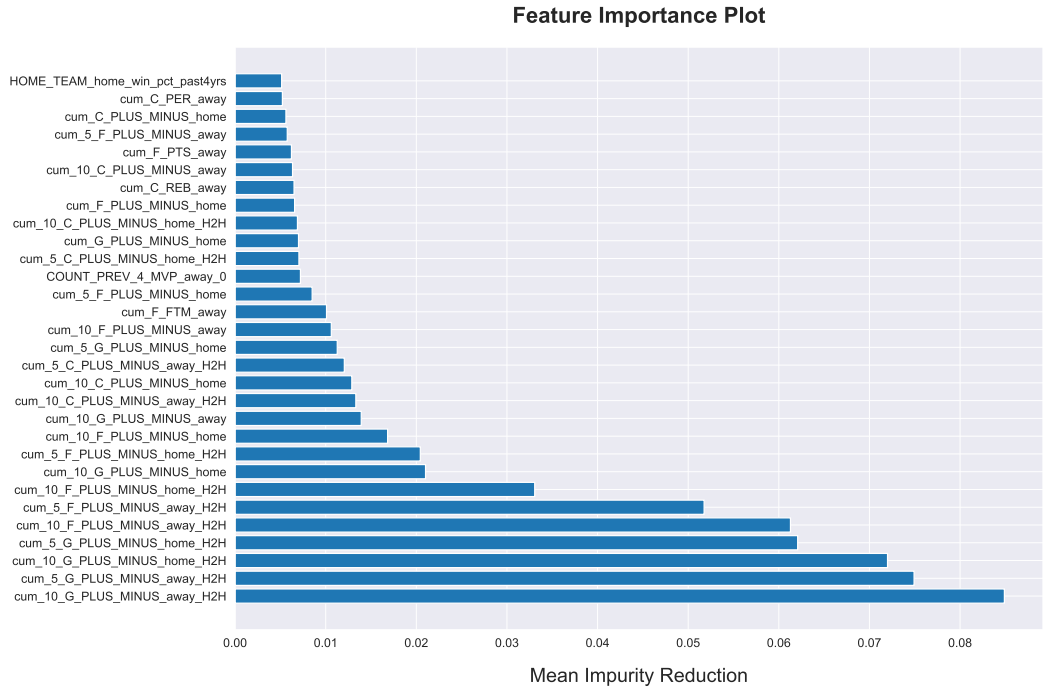
Sequentially Cross-Validation XGBoost Confusion Matrix and Performance

	HOME WIN	AWAY WIN
PRED. HOME WIN	498	583
PRED. AWAY WIN	324	1221

Table 3.9: Confusion Matrix

	Test Set Statistic
Accuracy	0.6546
TPR	0.7903
FPR	0.5393

Table 3.10: Performance Metrics



To see if we dealt with overfitting properly, we re-run our XGBoost model with only the Top 30 features. Below, we see the results compared to using all features. From these tables of outputs, we see that using all the features is slightly better than using very few, likely due to the high regularization hyperparameter value.

	HOME WIN	AWAY WIN
PRED. HOME WIN	491	590
PRED. AWAY WIN	321	1224

Table 3.11: Confusion Matrix

	Test Set Statistic
Accuracy	0.6531
TPR	0.7922
FPR	0.5458

Table 3.12: Performance Metrics

1.5 Model #4: Neural Network (Multilayer Perceptron Classifier)

For our final model, we explore the implementation of a Neural Network using Sci-Kit Learn's MLPClassifier. We perform no cross-validation on the hidden layer sizes or node counts, and create a network with two hidden layers with 25 nodes in each. We attempted to use TensorFlow

but couldn't quite tune it properly, so this is more of a toe-dip into the water of deep learning. Regardless, our best outputs are shown below.

Neural Network Confusion Matrix and Performance.

	HOME WIN	AWAY WIN
PRED. HOME WIN	430	651
PRED. AWAY WIN	267	1278

Table 3.13: Confusion Matrix

	Test Set Statistic
Accuracy	0.6504
TPR	0.8272
FPR	0.6022

Table 3.14: Performance Metrics

1.6 Final Model Outputs

We aggregate our results in the table below. We see that

	Accuracy	TPR	FPR
Baseline Model	0.5883	1.0000	1.0000
Logistic Regression	0.6580	0.8220	0.5763
Random Forest	0.6565	0.8045	0.5550
XGBoost	0.6546	0.7903	0.5393
Neural Network	0.6645	0.8149	0.5504

ii Approach #2: Hold-Out Validation Set and Blending

Recall that our limiting factor in building our most robust model (from a modeling standpoint—not in regards to data) is our lack of computational power. That was why we removed potentially unhelpful features before cross-validation. However, we now try the alternative approach, which is to split our data into (1) Training Set, (2) Validation A Set, (3) Validation B Set, and (4) Testing Set. To split this, we first set our Testing Data to be the 2018 Season (8.5%). We then split the remaining data into Train, Validation A, and Validation B randomly such that Validation A and B are each 9.2% of the total data, and Training is 73.2%. Instead of cross-validation, we tune hyperparameters by iterating through a search space and choosing

the value that maximizes the accuracy on Validation A. After doing this for all of our models, we use the predicted probabilities for the Validation B Set to train a Logistic Regression model to evaluate on on our test set (effectively creating a blended prediction). The results are shown below. Our main goal of this exploration was to see if a Blended model could outperform our best model from earlier, and it does (albeit by 0.0019)! Interestingly, our Logistic Regression is the second best performing model in terms of accuracy, outperforming our Neural Network.

	Accuracy	TPR	FPR
Baseline Model	0.5883	1.0000	1.0000
Logistic Regression	0.6611	0.8162	0.5606
Random Forest	0.6527	0.8311	0.6022
XGBoost	0.6531	0.7961	0.5513
Neural Network	0.6596	0.8634	0.6318
Blended	0.6664	0.8239	0.5587

Figure 3.2: Final results using a hold-out validation set approach and blending.

Conclusion and Extensions

In this project, we went through an entire detailed process of collecting data, constructing features with predictive power, and implementing various supervised learning techniques to achieve our best accuracy of X% with a XYZ model. There are various routes we could take with this project. On a modeling and computation standpoint, our best next steps would be to pursue using higher-powered computers to let us do more cross-validation and improve our model performance. Further, having a greater understanding of neural network architecture would lend itself nicely to building a more accurate model. We've seen this model to perform very well, and there is likely untapped potential in its implementation. More importantly, we believe we need to construct better features, specifically ones that aggregate statistics we've already considered. From our feature importance scores, we see that nearly all of the Top 30 for Random Forest and XGBoost are PLUS/MINUS statistics. These statistics reflect how a team performs when a given player (or players are playing) as how many points teams score relative to each other (e.g. a *cum_10_G_PLUS_MINUS_home* of +10 means that in the last 10 games, that home team outscores their opponents by an average of 10 points when the starting guards are playing. There are many other aggregate statistics such as Win Shares, Possession or Pace based statistics, True Shooting Percentage, Usage Rate, etc. Access to many of these on a player-by-player basis lie behind a paywall (or a time wall in our case as well), but implementing this undoubtedly would allow us to reduce model complexity and increase our performance. Regardless, we are very pleased with our ability to curate this many predictive features, build models with an appreciable increase in accuracy over the baseline model, and look forward to continue in our efforts to reach closer to 75% accuracy.

Appendix #1: Cumulative Statistic Engineering

FP1_CUM_STATS

May 9, 2021

[1]:
```
# Importing Libraries & Functions
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import time
import re
import warnings
warnings.filterwarnings('ignore')
```

First, we load in our GAMES and TEAMS datasets and perform some simple cleaning to make them more usable.

[2]:
```
# Loading + Cleaning Data
games = pd.read_csv('FP1_DATA/games.csv')
timestamp = pd.to_datetime(games['GAME_DATE_EST'])
games.insert(0, 'TIMESTAMP', timestamp)
season_list = np.arange(2007,2019)
games = games[games['SEASON'].isin(season_list)].sort_values('TIMESTAMP')

teams = pd.read_csv('FP1_DATA/teams.csv')
teams = teams[['TEAM_ID','NICKNAME']]
team_dict = teams.set_index('TEAM_ID').T.to_dict('list')
games['HOME_TEAM_NAME'] = games['HOME_TEAM_ID'].map(team_dict)
games['AWAY_TEAM_NAME'] = games['VISITOR_TEAM_ID'].map(team_dict)


drop_cols = ['GAME_DATE_EST', 'GAME_STATUS_TEXT', 'HOME_TEAM_ID',
             'VISITOR_TEAM_ID', 'TEAM_ID_home', 'TEAM_ID_away']
games = games.drop(drop_cols, axis = 1).reset_index(drop = True)
front_cols = ['TIMESTAMP', 'GAME_ID', 'HOME_TEAM_NAME', 'AWAY_TEAM_NAME']
temp_games = games[front_cols]
games = pd.concat([temp_games, games.drop(front_cols, axis = 1)], axis = 1)
games['HOME_TEAM_NAME'] = games['HOME_TEAM_NAME'].apply(lambda x: x[0])
games['AWAY_TEAM_NAME'] = games['AWAY_TEAM_NAME'].apply(lambda x: x[0])
```

0.0.1 Player-Position Statistics

Because we want to consider player/position contributions to the overall statistics, we use the DETAILS dataframe to convert players to positions, and then calculate the statistics in each game, grouped by each position, Guards, Forwards, and Center. We also then calculate Player Efficiency Rating (PER) using these statistics. Note that a statistic like *ppg_G_home* is the sum of the points per game by the starting guards.

[33]:
```
# Defining the Position Stat Function
def pos_stat(stat_list):


    ### Takes in a list of statistics to consider from the DETAILS data frame␣
↪and calculates
    ### the stats by position in each home for the home and away team.
    ### NOTE: THESE ARE NOT CUMULATIVE STATS!!!!!


    # Removing non-starters, getting rid of unnecessary columns
    details = pd.read_csv('FP1_DATA/games_details.csv')
    season_dict = games[['GAME_ID', 'SEASON']].set_index('GAME_ID').iloc[:,0].T.
↪to_dict()
    details['SEASON'] = details['GAME_ID'].map(season_dict)
    details = details[details.notna()]
    details = details[~details['START_POSITION'].isna()]
    details['TEAM_NAME'] = details['TEAM_ID'].map(team_dict).apply(lambda x:␣
↪x[0])
    details['MINS'] = details['MIN'].str.split(':').apply(lambda x: float(x[0])␣
↪+ (float(x[1])/60))
    details = details.drop(['TEAM_ABBREVIATION', 'TEAM_CITY', 'COMMENT',␣
↪'MIN'], axis = 1)
    temp_cols = ['GAME_ID','START_POSITION', 'TEAM_NAME']
    final_cols = np.append(temp_cols, stat_list).flatten()
    details = details[final_cols]

    output_df = pd.DataFrame()
    for stat in stat_list:


        # Groupby the GAME, POSITION and TEAM and add up all stats by each␣
↪position
        df = details.copy()
        df = df.groupby(['GAME_ID', 'START_POSITION', 'TEAM_NAME']).sum().
↪reset_index()
        df = pd.merge(df, games[['GAME_ID', 'HOME_TEAM_NAME',␣
↪'AWAY_TEAM_NAME']], on = 'GAME_ID')
```

```
        # Need to Re-Order the DF s.t. home and away are known
        home_condition = (df['TEAM_NAME'] == df['HOME_TEAM_NAME'])
        away_condition = (df['TEAM_NAME'] != df['HOME_TEAM_NAME'])

        # Adding HOME Columns Manually
        df['C_' + str(stat) + '_home'] = df[ home_condition &␣
↪(df['START_POSITION'] == 'C')][stat]
        df['F_' + str(stat) + '_home'] = df[ home_condition &␣
↪(df['START_POSITION'] == 'F')][stat]
        df['G_' + str(stat) + '_home'] = df[ home_condition &␣
↪(df['START_POSITION'] == 'G')][stat]

        # Adding AWAY Columns Manually
        df['C_' + str(stat) + '_away'] = df[ away_condition &␣
↪(df['START_POSITION'] == 'C')][stat]
        df['F_' + str(stat) + '_away'] = df[ away_condition &␣
↪(df['START_POSITION'] == 'F')][stat]
        df['G_' + str(stat) + '_away'] = df[ away_condition &␣
↪(df['START_POSITION'] == 'G')][stat]

        # Grouping Again to Get Rid of NaN
        df = df.groupby('GAME_ID', as_index = False).sum()
        if stat_list.index(stat) > 0:
            df = df.drop('GAME_ID', axis = 1)
        # Append to final data frame
        output_df = pd.concat([output_df, df], axis = 1)

    output_df = output_df.drop(stat_list, axis = 1)
    return output_df
```

[34]:
```
# Adding Columns for Position Stats using pos_stat Function
stat_list = ['FGM', 'FGA','FG_PCT', 'FG3M', 'FG3A',
             'FG3_PCT', 'FTM', 'FTA', 'FT_PCT', 'OREB',
             'DREB', 'REB', 'AST', 'STL', 'BLK', 'TO',
             'PF', 'PTS', 'PLUS_MINUS']
pos_df = pos_stat(stat_list)
```

[35]:
```
# Calculing Player Efficiency Rating
# Formula: (PTS + REB + AST + STL + BLK - Missed FG - Missed FT - TO) / GP
p1 = pos_df['C_PTS_home'] + pos_df['C_REB_home'] + pos_df['C_AST_home'] +␣
 ↪pos_df['C_STL_home']
missed_fg = (pos_df['C_FGA_home'] - pos_df['C_FGM_home'])
missed_ft = pos_df['C_FTA_home'] - pos_df['C_FTM_home']
p2 = (missed_fg + missed_ft + pos_df['C_TO_home'])
per = (p1 - p2)/82
```

```
for pos in ['C', 'F', 'G']:
    p1 = pos_df[pos + '_PTS_home'] + pos_df[pos + '_REB_home'] + pos_df[pos +␣
 ↪'_AST_home'] + pos_df[pos + '_STL_home']
    missed_fg = (pos_df[pos + '_FGA_home'] - pos_df[pos + '_FGM_home'])
    missed_ft = pos_df[pos + '_FTA_home'] - pos_df[pos + '_FTM_home']
    p2 = (missed_fg + missed_ft + pos_df[pos + '_TO_home'])
    per = (p1-p2)/82
    pos_df[pos + '_PER_home'] = per

for pos in ['C', 'F', 'G']:
    p1 = pos_df[pos + '_PTS_away'] + pos_df[pos + '_REB_away'] + pos_df[pos +␣
 ↪'_AST_away'] + pos_df[pos + '_STL_away']
    missed_fg = (pos_df[pos + '_FGA_away'] - pos_df[pos + '_FGM_away'])
    missed_ft = pos_df[pos + '_FTA_away'] - pos_df[pos + '_FTM_away']
    p2 = (missed_fg + missed_ft + pos_df[pos + '_TO_away'])
    per = (p1-p2)/82
    pos_df[pos + '_PER_away'] = per
```

0.0.2 Calculating the Cumulative Statistics

The bulk of our cumulative sum and average calculations are shown below. We first define functions to calculate the cumulative average statistics for the (1) Season, (2) Last 5 Games, and (3) Last 10 Games. Because a team's win or loss count is calculated as a sum, we also create functions to calculate those sums for the Season, Last 5 and Last 10 Games based off of the 'HOME_TEAM_WINS' column.

[36]:
```
# Defining the Cumulative Stat Functions
def cum_avg(arr):
    temp_list = [np.mean(arr[:i]) for i in np.arange(len(arr)+1)][1:]
    return np.append(np.nan, temp_list)[:-1]

def cum_5_avg(arr):
    means = []
    for i in np.arange(len(arr)):
        if i < 5:
            means = np.append(means, np.nan)
        else:
            means = np.append(means, np.mean(arr[i-5: i+1]))
    return means

def cum_10_avg(arr):
    means = []
    for i in np.arange(len(arr)):
        if i < 10:
            means = np.append(means, np.nan)
        else:
            means = np.append(means, np.mean(arr[i-10: i+1]))
```

```
    return means

def cum_wins(arr):
    temp_list = [np.sum(arr[:i]) for i in np.arange(len(arr)+1)][1:]
    return np.append(np.nan, temp_list)[:-1]

def cum_5_wins(arr):
    wins = []
    for i in np.arange(len(arr)):
        if i < 5:
            wins = np.append(wins, np.nan)
        else:
            wins = np.append(wins, np.sum(arr[i-5: i+1]))
    return wins

def cum_10_wins(arr):
    wins = []
    for i in np.arange(len(arr)):
        if i < 10:
            wins = np.append(wins, np.nan)
        else:
            wins = np.append(wins, np.sum(arr[i-10: i+1]))
    return wins

def cum_losses(arr):
    temp_list = [np.count_nonzero(arr[:i]==0) for i in np.arange(len(arr)+1)][1:
 ↪]
    return np.append(np.nan, temp_list)[:-1]

def cum_5_losses(arr):
    losses = []
    for i in np.arange(len(arr)):
        if i < 5:
            losses = np.append(losses, np.nan)
        else:
            losses = np.append(losses, np.count_nonzero(arr[i-5: i+1]==0))
    return losses

def cum_10_losses(arr):
    losses = []
    for i in np.arange(len(arr)):
        if i < 10:
            losses = np.append(losses, np.nan)
        else:
            losses = np.append(losses, np.count_nonzero(arr[i-10: i+1]==0))
    return losses
```

The next two cells do the same thing but for the HOME and AWAY statistics, respectively. For each HOME and AWAY stat, we do the following:

(1) Calculate the Cumulative Season Average

(2) Calculate the Cumulative Last 5 Game Average

(3) Calculate the Cumulative Last 10 Game Average

(4) Calculate the Cumulative Season Average for this specific combination of HOME and AWAY TEAM (Head-to-Head)

(5) Calculate the Cumulative Last 5 Game Average for this specific combination of HOME and AWAY TEAM (Head-to-Head)

(6) Calculate the Cumulative Last 10 Game Average for this specific combination of HOME and AWAY TEAM (Head-to-Head)

We had to go about this in a rather unorthodox way using two grouped dataframes because using a nested for loop increased the run time by over 50x. We then combine these two dataframes together, using 'GAME_ID' to join. This turns our initial dataframe of 132 columns to 792 features.

[37]:
```
# Cumulative Statistics: HOME
df = games.copy()
df = df.merge(pos_df, on = 'GAME_ID')
df['TEAM_LIST'] = df['HOME_TEAM_NAME'] + ', ' + df['AWAY_TEAM_NAME']
df_home = df.drop('AWAY_TEAM_NAME', axis = 1)
df_home = df.sort_values('TIMESTAMP', ascending = True)
column_list = df_home.columns[4:]
home_stat_list = [stat for stat in column_list if 'home' in stat]

t = time.time()
for stat in home_stat_list:

    keep_cols = np.append(['TIMESTAMP', 'TEAM_LIST','GAME_ID',␣
↪'HOME_TEAM_NAME', 'AWAY_TEAM_NAME'], stat)
    df_home_temp = df_home[keep_cols]

    # Two Grouped DFs
    df_1 = df_home_temp.groupby(['HOME_TEAM_NAME', 'GAME_ID']).mean()
    df_2 = df_home_temp.groupby(['HOME_TEAM_NAME']).agg(list)

    # Using Functions above to calulate the cumulative values as lists
    cum_lists = df_2.apply(lambda x: cum_avg(x[stat]), axis = 1)
    cum_5_lists = df_2.apply(lambda x: cum_5_avg(x[stat]), axis = 1)
    cum_10_lists = df_2.apply(lambda x: cum_10_avg(x[stat]), axis = 1)

    # Converting these nested lists to single lists
    cum_vals = cum_lists.to_frame().explode(0)[0].to_numpy()
    cum_5_vals = cum_5_lists.to_frame().explode(0)[0].to_numpy()
    cum_10_vals = cum_10_lists.to_frame().explode(0)[0].to_numpy()
```

```
df_1 = df_1.reset_index()
df_1['cum_' + str(stat)] = cum_vals
df_1['cum_5_' + str(stat)] = cum_5_vals
df_1['cum_10_' + str(stat)] = cum_10_vals

# Converting to Dictionaries and Adding to Large DF
cum_dict = df_1.set_index('GAME_ID').iloc[:, -3].T.to_dict()
cum_dict_5 = df_1.set_index('GAME_ID').iloc[:, -2].T.to_dict()
cum_dict_10 = df_1.set_index('GAME_ID').iloc[:, -1].T.to_dict()

df_home['cum_' + str(stat)] = df['GAME_ID'].map(cum_dict)
df_home['cum_5_' + str(stat)] = df['GAME_ID'].map(cum_dict_5)
df_home['cum_10_' + str(stat)] = df['GAME_ID'].map(cum_dict_10)


# HEAD TO HEAD STATS ==> SAME THING BASICALLY

# Two Grouped DFs
df_1_H2H = df_home.groupby(['TEAM_LIST', 'GAME_ID']).mean()
df_2_H2H = df_home.groupby('TEAM_LIST').agg(list)

# Using Functions above to calulate the cumulative values as lists
cum_lists_H2H = df_2_H2H.apply(lambda x: cum_avg(x[stat]), axis = 1)
cum_5_lists_H2H = df_2_H2H.apply(lambda x: cum_5_avg(x[stat]), axis = 1)
cum_10_lists_H2H = df_2_H2H.apply(lambda x: cum_10_avg(x[stat]), axis = 1)

# Converting these nested lists to single lists
cum_vals_H2H = cum_lists_H2H.to_frame().explode(0)[0].to_numpy()
cum_5_vals_H2H = cum_5_lists_H2H.to_frame().explode(0)[0].to_numpy()
cum_10_vals_H2H = cum_5_lists_H2H.to_frame().explode(0)[0].to_numpy()

#df_1_H2H = df_1_H2H.reset_index()
df_1_H2H['cum_' + str(stat) + '_H2H'] = cum_vals_H2H
df_1_H2H['cum_5_' + str(stat) + '_H2H'] = cum_5_vals_H2H
df_1_H2H['cum_10_' + str(stat) + '_H2H'] = cum_10_vals_H2H


# Converting to Dictionaries and Adding to Large DF
cum_dict_H2H = df_1_H2H.droplevel(0).iloc[:, -3].T.to_dict()
cum_5_dict_H2H = df_1_H2H.droplevel(0).iloc[:, -2].T.to_dict()
cum_10_dict_H2H = df_1_H2H.droplevel(0).iloc[:, -1].T.to_dict()

df_home['cum_' + str(stat) + '_H2H'] = df['GAME_ID'].map(cum_dict_H2H)
df_home['cum_5_' + str(stat) + '_H2H'] = df['GAME_ID'].map(cum_5_dict_H2H)
df_home['cum_10_' + str(stat) + '_H2H'] = df['GAME_ID'].map(cum_10_dict_H2H)
```

```
   print('# ' + str(home_stat_list.index(stat)+1) + ' of ' +␣
↪str(len(home_stat_list)) + ' COMPLETE: ', str(stat), '-- TIME: ', time.
↪time()-t)
```

```
# 1 of 66 COMPLETE:  PTS_home -- TIME:  4.732546091079712
# 2 of 66 COMPLETE:  FG_PCT_home -- TIME:  9.695076942443848
# 3 of 66 COMPLETE:  FT_PCT_home -- TIME:  14.761112928390503
# 4 of 66 COMPLETE:  FG3_PCT_home -- TIME:  19.93681502342224
# 5 of 66 COMPLETE:  AST_home -- TIME:  25.567740201950073
# 6 of 66 COMPLETE:  REB_home -- TIME:  30.941532135009766
# 7 of 66 COMPLETE:  C_FGM_home -- TIME:  36.46212697029114
# 8 of 66 COMPLETE:  F_FGM_home -- TIME:  42.01122498512268
# 9 of 66 COMPLETE:  G_FGM_home -- TIME:  47.68910884857178
# 10 of 66 COMPLETE:  C_FGA_home -- TIME:  53.532379150390625
# 11 of 66 COMPLETE:  F_FGA_home -- TIME:  59.75461792945862
# 12 of 66 COMPLETE:  G_FGA_home -- TIME:  65.97572493553162
# 13 of 66 COMPLETE:  C_FG_PCT_home -- TIME:  72.31214499473572
# 14 of 66 COMPLETE:  F_FG_PCT_home -- TIME:  78.80450510978699
# 15 of 66 COMPLETE:  G_FG_PCT_home -- TIME:  85.44951796531677
# 16 of 66 COMPLETE:  C_FG3M_home -- TIME:  92.15903091430664
# 17 of 66 COMPLETE:  F_FG3M_home -- TIME:  99.03179001808167
# 18 of 66 COMPLETE:  G_FG3M_home -- TIME:  106.04615688323975
# 19 of 66 COMPLETE:  C_FG3A_home -- TIME:  113.1100070476532
# 20 of 66 COMPLETE:  F_FG3A_home -- TIME:  120.63993501663208
# 21 of 66 COMPLETE:  G_FG3A_home -- TIME:  128.50653100013733
# 22 of 66 COMPLETE:  C_FG3_PCT_home -- TIME:  136.20415687561035
# 23 of 66 COMPLETE:  F_FG3_PCT_home -- TIME:  143.93483114242554
# 24 of 66 COMPLETE:  G_FG3_PCT_home -- TIME:  151.8971450328827
# 25 of 66 COMPLETE:  C_FTM_home -- TIME:  159.69006609916687
# 26 of 66 COMPLETE:  F_FTM_home -- TIME:  167.07637405395508
# 27 of 66 COMPLETE:  G_FTM_home -- TIME:  174.62059688568115
# 28 of 66 COMPLETE:  C_FTA_home -- TIME:  182.10178399085999
# 29 of 66 COMPLETE:  F_FTA_home -- TIME:  189.88045191764832
# 30 of 66 COMPLETE:  G_FTA_home -- TIME:  197.57915687561035
# 31 of 66 COMPLETE:  C_FT_PCT_home -- TIME:  205.346195936203
# 32 of 66 COMPLETE:  F_FT_PCT_home -- TIME:  213.21816396713257
# 33 of 66 COMPLETE:  G_FT_PCT_home -- TIME:  221.12918281555176
# 34 of 66 COMPLETE:  C_OREB_home -- TIME:  229.35324597358704
# 35 of 66 COMPLETE:  F_OREB_home -- TIME:  237.59066581726074
# 36 of 66 COMPLETE:  G_OREB_home -- TIME:  245.87727618217468
# 37 of 66 COMPLETE:  C_DREB_home -- TIME:  254.53458881378174
# 38 of 66 COMPLETE:  F_DREB_home -- TIME:  263.1116750240326
# 39 of 66 COMPLETE:  G_DREB_home -- TIME:  272.0000171661377
```

```
# 40 of 66 COMPLETE:  C_REB_home -- TIME:  280.865033864975
# 41 of 66 COMPLETE:  F_REB_home -- TIME:  289.7501118183136
# 42 of 66 COMPLETE:  G_REB_home -- TIME:  299.03335189819336
# 43 of 66 COMPLETE:  C_AST_home -- TIME:  308.22829699516296
# 44 of 66 COMPLETE:  F_AST_home -- TIME:  317.475172996521
# 45 of 66 COMPLETE:  G_AST_home -- TIME:  326.8412981033325
# 46 of 66 COMPLETE:  C_STL_home -- TIME:  336.6581370830536
# 47 of 66 COMPLETE:  F_STL_home -- TIME:  346.2609899044037
# 48 of 66 COMPLETE:  G_STL_home -- TIME:  355.999685049057
# 49 of 66 COMPLETE:  C_BLK_home -- TIME:  366.0432939529419
# 50 of 66 COMPLETE:  F_BLK_home -- TIME:  376.02791690826416
# 51 of 66 COMPLETE:  G_BLK_home -- TIME:  386.0702030658722
# 52 of 66 COMPLETE:  C_TO_home -- TIME:  396.69300293922424
# 53 of 66 COMPLETE:  F_TO_home -- TIME:  407.72757291793823
# 54 of 66 COMPLETE:  G_TO_home -- TIME:  418.47975397109985
# 55 of 66 COMPLETE:  C_PF_home -- TIME:  429.37751603126526
# 56 of 66 COMPLETE:  F_PF_home -- TIME:  440.09311509132385
# 57 of 66 COMPLETE:  G_PF_home -- TIME:  450.8636300563812
# 58 of 66 COMPLETE:  C_PTS_home -- TIME:  462.70912289619446
# 59 of 66 COMPLETE:  F_PTS_home -- TIME:  474.4053599834442
# 60 of 66 COMPLETE:  G_PTS_home -- TIME:  487.0005979537964
# 61 of 66 COMPLETE:  C_PLUS_MINUS_home -- TIME:  499.56221413612366
# 62 of 66 COMPLETE:  F_PLUS_MINUS_home -- TIME:  512.1199169158936
# 63 of 66 COMPLETE:  G_PLUS_MINUS_home -- TIME:  525.0752880573273
# 64 of 66 COMPLETE:  C_PER_home -- TIME:  537.8398959636688
# 65 of 66 COMPLETE:  F_PER_home -- TIME:  551.0608150959015
# 66 of 66 COMPLETE:  G_PER_home -- TIME:  564.3962941169739
```

[38]:
```
# Cumulative Statistics: AWAY
df_away = df.drop('HOME_TEAM_NAME', axis = 1)
df_away = df.sort_values('TIMESTAMP', ascending = True)
column_list = df_away.columns[4:]
away_stat_list = [stat for stat in column_list if 'away' in stat]

t = time.time()
for stat in away_stat_list:

    keep_cols = np.append(['TIMESTAMP', 'TEAM_LIST','GAME_ID',␣
↪'HOME_TEAM_NAME', 'AWAY_TEAM_NAME'], stat)
    df_away_temp = df_away[keep_cols]

    # Two Grouped DFs
    df_1 = df_away_temp.groupby(['AWAY_TEAM_NAME', 'GAME_ID']).mean()
    df_2 = df_away_temp.groupby(['AWAY_TEAM_NAME']).agg(list)

    # Using Functions above to calulate the cumulative values as lists
    cum_lists = df_2.apply(lambda x: cum_avg(x[stat]), axis = 1)
```

```
    cum_5_lists = df_2.apply(lambda x: cum_5_avg(x[stat]), axis = 1)
    cum_10_lists = df_2.apply(lambda x: cum_10_avg(x[stat]), axis = 1)

    # Converting these nested lists to single lists
    cum_vals = cum_lists.to_frame().explode(0)[0].to_numpy()
    cum_5_vals = cum_5_lists.to_frame().explode(0)[0].to_numpy()
    cum_10_vals = cum_10_lists.to_frame().explode(0)[0].to_numpy()

    df_1 = df_1.reset_index()
    df_1['cum_' + str(stat)] = cum_vals
    df_1['cum_5_' + str(stat)] = cum_5_vals
    df_1['cum_10_' + str(stat)] = cum_10_vals

    # Converting to Dictionaries and Adding to Large DF
    cum_dict = df_1.set_index('GAME_ID').iloc[:, -3].T.to_dict()
    cum_dict_5 = df_1.set_index('GAME_ID').iloc[:, -2].T.to_dict()
    cum_dict_10 = df_1.set_index('GAME_ID').iloc[:, -1].T.to_dict()

    df_away['cum_' + str(stat)] = df['GAME_ID'].map(cum_dict)
    df_away['cum_5_' + str(stat)] = df['GAME_ID'].map(cum_dict_5)
    df_away['cum_10_' + str(stat)] = df['GAME_ID'].map(cum_dict_10)




    # HEAD TO HEAD STATS ==> SAME THING BASICALLY



    # Two Grouped DFs
    df_1_H2H = df_away.groupby(['TEAM_LIST', 'GAME_ID']).mean()
    df_2_H2H = df_away.groupby('TEAM_LIST').agg(list)

    # Using Functions above to calulate the cumulative values as lists
    cum_lists_H2H = df_2_H2H.apply(lambda x: cum_avg(x[stat]), axis = 1)
    cum_5_lists_H2H = df_2_H2H.apply(lambda x: cum_5_avg(x[stat]), axis = 1)
    cum_10_lists_H2H = df_2_H2H.apply(lambda x: cum_10_avg(x[stat]), axis = 1)

    # Converting these nested lists to single lists
    cum_vals_H2H = cum_lists_H2H.to_frame().explode(0)[0].to_numpy()
    cum_5_vals_H2H = cum_5_lists_H2H.to_frame().explode(0)[0].to_numpy()
    cum_10_vals_H2H = cum_5_lists_H2H.to_frame().explode(0)[0].to_numpy()

    #df_1_H2H = df_1_H2H.reset_index()
    df_1_H2H['cum_' + str(stat) + '_H2H'] = cum_vals_H2H
    df_1_H2H['cum_5_' + str(stat) + '_H2H'] = cum_5_vals_H2H
    df_1_H2H['cum_10_' + str(stat) + '_H2H'] = cum_10_vals_H2H
```

```
    # Converting to Dictionaries and Adding to Large DF
    cum_dict_H2H = df_1_H2H.droplevel(0).iloc[:, -3].T.to_dict()
    cum_5_dict_H2H = df_1_H2H.droplevel(0).iloc[:, -2].T.to_dict()
    cum_10_dict_H2H = df_1_H2H.droplevel(0).iloc[:, -1].T.to_dict()

    df_away['cum_' + str(stat) + '_H2H'] = df['GAME_ID'].map(cum_dict_H2H)
    df_away['cum_5_' + str(stat) + '_H2H'] = df['GAME_ID'].map(cum_5_dict_H2H)
    df_away['cum_10_' + str(stat) + '_H2H'] = df['GAME_ID'].map(cum_10_dict_H2H)


    print('# ' + str(away_stat_list.index(stat)+1) + ' of ' +␣
↪str(len(away_stat_list)) + ' COMPLETE: ', str(stat), '-- TIME: ', time.
↪time()-t)
```

```
# 1 of 66 COMPLETE:  PTS_away -- TIME:  4.816655158996582
# 2 of 66 COMPLETE:  FG_PCT_away -- TIME:  9.50899600982666
# 3 of 66 COMPLETE:  FT_PCT_away -- TIME:  14.484194040298462
# 4 of 66 COMPLETE:  FG3_PCT_away -- TIME:  19.607362985610962
# 5 of 66 COMPLETE:  AST_away -- TIME:  24.847387075424194
# 6 of 66 COMPLETE:  REB_away -- TIME:  30.26409387588501
# 7 of 66 COMPLETE:  C_FGM_away -- TIME:  35.743788957595825
# 8 of 66 COMPLETE:  F_FGM_away -- TIME:  41.35866117477417
# 9 of 66 COMPLETE:  G_FGM_away -- TIME:  47.139140129089355
# 10 of 66 COMPLETE:  C_FGA_away -- TIME:  52.970470905303955
# 11 of 66 COMPLETE:  F_FGA_away -- TIME:  58.956490993499756
# 12 of 66 COMPLETE:  G_FGA_away -- TIME:  65.04403519630432
# 13 of 66 COMPLETE:  C_FG_PCT_away -- TIME:  71.5125629901886
# 14 of 66 COMPLETE:  F_FG_PCT_away -- TIME:  78.20596814155579
# 15 of 66 COMPLETE:  G_FG_PCT_away -- TIME:  84.69293427467346
# 16 of 66 COMPLETE:  C_FG3M_away -- TIME:  91.42034006118774
# 17 of 66 COMPLETE:  F_FG3M_away -- TIME:  98.34503698348999
# 18 of 66 COMPLETE:  G_FG3M_away -- TIME:  106.01250886917114
# 19 of 66 COMPLETE:  C_FG3A_away -- TIME:  114.5474910736084
# 20 of 66 COMPLETE:  F_FG3A_away -- TIME:  123.3448371887207
# 21 of 66 COMPLETE:  G_FG3A_away -- TIME:  131.8079149723053
# 22 of 66 COMPLETE:  C_FG3_PCT_away -- TIME:  140.44215083122253
# 23 of 66 COMPLETE:  F_FG3_PCT_away -- TIME:  149.15179109573364
# 24 of 66 COMPLETE:  G_FG3_PCT_away -- TIME:  158.22675395011902
# 25 of 66 COMPLETE:  C_FTM_away -- TIME:  167.22784519195557
# 26 of 66 COMPLETE:  F_FTM_away -- TIME:  176.34949803352356
# 27 of 66 COMPLETE:  G_FTM_away -- TIME:  185.65676498413086
# 28 of 66 COMPLETE:  C_FTA_away -- TIME:  194.9421420097351
# 29 of 66 COMPLETE:  F_FTA_away -- TIME:  204.45041728019714
# 30 of 66 COMPLETE:  G_FTA_away -- TIME:  214.49129915237427
# 31 of 66 COMPLETE:  C_FT_PCT_away -- TIME:  224.46165084838867
```

```
# 32 of 66 COMPLETE:  F_FT_PCT_away -- TIME:  234.3953721523285
# 33 of 66 COMPLETE:  G_FT_PCT_away -- TIME:  244.97989106178284
# 34 of 66 COMPLETE:  C_OREB_away -- TIME:  255.18730998039246
# 35 of 66 COMPLETE:  F_OREB_away -- TIME:  264.47126817703247
# 36 of 66 COMPLETE:  G_OREB_away -- TIME:  273.49217891693115
# 37 of 66 COMPLETE:  C_DREB_away -- TIME:  282.9281189441681
# 38 of 66 COMPLETE:  F_DREB_away -- TIME:  292.20760798454285
# 39 of 66 COMPLETE:  G_DREB_away -- TIME:  301.4861340522766
# 40 of 66 COMPLETE:  C_REB_away -- TIME:  311.6607029438019
# 41 of 66 COMPLETE:  F_REB_away -- TIME:  321.65348196029663
# 42 of 66 COMPLETE:  G_REB_away -- TIME:  332.44315910339355
# 43 of 66 COMPLETE:  C_AST_away -- TIME:  342.8448979854584
# 44 of 66 COMPLETE:  F_AST_away -- TIME:  352.99427604675293
# 45 of 66 COMPLETE:  G_AST_away -- TIME:  363.7647511959076
# 46 of 66 COMPLETE:  C_STL_away -- TIME:  374.22685408592224
# 47 of 66 COMPLETE:  F_STL_away -- TIME:  385.01499009132385
# 48 of 66 COMPLETE:  G_STL_away -- TIME:  395.6952350139618
# 49 of 66 COMPLETE:  C_BLK_away -- TIME:  406.950119972229
# 50 of 66 COMPLETE:  F_BLK_away -- TIME:  418.1836130619049
# 51 of 66 COMPLETE:  G_BLK_away -- TIME:  430.0547981262207
# 52 of 66 COMPLETE:  C_TO_away -- TIME:  442.0038981437683
# 53 of 66 COMPLETE:  F_TO_away -- TIME:  454.32734513282776
# 54 of 66 COMPLETE:  G_TO_away -- TIME:  466.3609290122986
# 55 of 66 COMPLETE:  C_PF_away -- TIME:  478.45548486709595
# 56 of 66 COMPLETE:  F_PF_away -- TIME:  490.4913680553436
# 57 of 66 COMPLETE:  G_PF_away -- TIME:  502.55992698669434
# 58 of 66 COMPLETE:  C_PTS_away -- TIME:  514.6256859302521
# 59 of 66 COMPLETE:  F_PTS_away -- TIME:  526.9215860366821
# 60 of 66 COMPLETE:  G_PTS_away -- TIME:  539.7647070884705
# 61 of 66 COMPLETE:  C_PLUS_MINUS_away -- TIME:  553.0157041549683
# 62 of 66 COMPLETE:  F_PLUS_MINUS_away -- TIME:  566.2632670402527
# 63 of 66 COMPLETE:  G_PLUS_MINUS_away -- TIME:  579.8383350372314
# 64 of 66 COMPLETE:  C_PER_away -- TIME:  593.600998878479
# 65 of 66 COMPLETE:  F_PER_away -- TIME:  607.1849961280823
# 66 of 66 COMPLETE:  G_PER_away -- TIME:  620.3899221420288
```

[39]:
```
# Combining the Home and Away DataFrames
df_home_2 = df_home.copy()
df_away_2 = df_away.copy()
df_home_2 = df_home_2.drop(np.append(home_stat_list, away_stat_list,), axis = 1)
df_away_2 = df_away_2.drop(np.append(home_stat_list, away_stat_list), axis = 1)

df_clean = pd.merge(df_home_2, df_away_2, on = 'GAME_ID')
xcol = [col for col in df_clean.columns if '_x' in col]
ycol = [col for col in df_clean.columns if '_y' in col]
df_clean = df_clean.drop(ycol, axis = 1)
df_clean.columns = df_clean.columns.str.replace(r'_x$', '')
```

0.0.3 Win & Loss Counters

We then do the same thing but for the Win and Loss counter. We complete this separately because it is a sum—not an average.

[40]:
```
# Win-Loss Counter: HOME
df_clean2 = df_clean.copy()
season_dict = games[['GAME_ID', 'SEASON']].set_index('GAME_ID').iloc[:,0].T.
 ↪to_dict()
df_clean2['SEASON'] = df_clean2['GAME_ID'].map(season_dict)

keep_cols = ['TIMESTAMP', 'SEASON', 'HOME_TEAM_WINS','GAME_ID',␣
 ↪'HOME_TEAM_NAME', 'AWAY_TEAM_NAME']
df_clean2_temp = df_clean2[keep_cols]
df_clean2_temp['AWAY_TEAM_WINS'] = df_clean2_temp['HOME_TEAM_WINS'].map({0:1, 1:
 ↪0})

# Two Grouped DFs
df_g1 = df_clean2_temp.groupby(['HOME_TEAM_NAME', 'SEASON','GAME_ID']).mean()
df_g2 = df_clean2_temp.groupby(['HOME_TEAM_NAME']).agg(list)

# Using Functions above to calulate the cumulative values as lists

    # HOME WINS!
cum_wins_home = df_g2['HOME_TEAM_WINS'].apply(cum_wins).to_frame().
 ↪explode('HOME_TEAM_WINS')['HOME_TEAM_WINS'].to_numpy()
cum_5_wins_home = df_g2['HOME_TEAM_WINS'].apply(cum_5_wins).to_frame().
 ↪explode('HOME_TEAM_WINS')['HOME_TEAM_WINS'].to_numpy()
cum_10_wins_home = df_g2['HOME_TEAM_WINS'].apply(cum_10_wins).to_frame().
 ↪explode('HOME_TEAM_WINS')['HOME_TEAM_WINS'].to_numpy()

    # HOME WINS!
cum_losses_home = df_g2['HOME_TEAM_WINS'].apply(cum_losses).to_frame().
 ↪explode('HOME_TEAM_WINS')['HOME_TEAM_WINS'].to_numpy()
cum_5_losses_home = df_g2['HOME_TEAM_WINS'].apply(cum_5_losses).to_frame().
 ↪explode('HOME_TEAM_WINS')['HOME_TEAM_WINS'].to_numpy()
cum_10_losses_home = df_g2['HOME_TEAM_WINS'].apply(cum_10_losses).to_frame().
 ↪explode('HOME_TEAM_WINS')['HOME_TEAM_WINS'].to_numpy()


# #df_clean = df_clean.reset_index()
df_clean2['cum_WINS_home'] = cum_wins_home
df_clean2['cum_LOSSES_home'] = cum_losses_home
df_clean2['cum_5_WINS_home'] = cum_5_wins_home
df_clean2['cum_10_WINS_home'] = cum_10_wins_home
df_clean2['cum_5_LOSSES_home'] = cum_5_losses_home
df_clean2['cum_10_LOSSES_home'] = cum_10_losses_home
```

[41]:
```
# Win-Loss Counter: AWAY


df_clean2_temp['AWAY_TEAM_WINS'] = df_clean2_temp['HOME_TEAM_WINS'].map({0:1, 1:
 ↪0})

# Two Grouped DFs
df_g1 = df_clean2_temp.groupby(['AWAY_TEAM_NAME', 'SEASON','GAME_ID']).mean()
df_g2 = df_clean2_temp.groupby(['AWAY_TEAM_NAME']).agg(list)

# Using Functions above to calulate the cumulative values as lists

    # AWAY WINS!
cum_wins_away = df_g2['AWAY_TEAM_WINS'].apply(cum_wins).to_frame().
 ↪explode('AWAY_TEAM_WINS')['AWAY_TEAM_WINS'].to_numpy()
cum_5_wins_away = df_g2['AWAY_TEAM_WINS'].apply(cum_5_wins).to_frame().
 ↪explode('AWAY_TEAM_WINS')['AWAY_TEAM_WINS'].to_numpy()
cum_10_wins_away = df_g2['AWAY_TEAM_WINS'].apply(cum_10_wins).to_frame().
 ↪explode('AWAY_TEAM_WINS')['AWAY_TEAM_WINS'].to_numpy()

    # AWAY LOSSES
cum_losses_away = df_g2['AWAY_TEAM_WINS'].apply(cum_losses).to_frame().
 ↪explode('AWAY_TEAM_WINS')['AWAY_TEAM_WINS'].to_numpy()
cum_5_losses_away = df_g2['AWAY_TEAM_WINS'].apply(cum_5_losses).to_frame().
 ↪explode('AWAY_TEAM_WINS')['AWAY_TEAM_WINS'].to_numpy()
cum_10_losses_away = df_g2['AWAY_TEAM_WINS'].apply(cum_10_losses).to_frame().
 ↪explode('AWAY_TEAM_WINS')['AWAY_TEAM_WINS'].to_numpy()


# #df_clean = df_clean.reset_index()
df_clean2['cum_WINS_away'] = cum_wins_away
df_clean2['cum_LOSSES_away'] = cum_losses_away
df_clean2['cum_5_WINS_away'] = cum_5_wins_away
df_clean2['cum_10_WINS_away'] = cum_10_wins_away
df_clean2['cum_5_LOSSES_away'] = cum_5_losses_away
df_clean2['cum_10_LOSSES_away'] = cum_10_losses_away
```

[42]:
```
# Win-Loss Counter: HOME HEAD-TO-HEAD

#HOME HEAD TO HEAD Win/Loss Counter

df_clean2_temp['AWAY_TEAM_WINS'] = df_clean2_temp['HOME_TEAM_WINS'].map({0:1, 1:
 ↪0})
df_clean2_temp['TEAM_LIST'] = df_clean2_temp['HOME_TEAM_NAME'] + ', ' +␣
 ↪df_clean2_temp['AWAY_TEAM_NAME']
```

```
# Two Grouped DFs
df_g1 = df_clean2_temp.groupby(['TEAM_LIST', 'SEASON','GAME_ID']).mean()
df_g2 = df_clean2_temp.groupby(['TEAM_LIST']).agg(list)

# Using Functions above to calulate the cumulative values as lists

    # HOME WINS!
cum_wins_home_H2H = df_g2['HOME_TEAM_WINS'].apply(cum_wins).to_frame().
 ↪explode('HOME_TEAM_WINS')['HOME_TEAM_WINS'].to_numpy()
cum_5_wins_home_H2H = df_g2['HOME_TEAM_WINS'].apply(cum_5_wins).to_frame().
 ↪explode('HOME_TEAM_WINS')['HOME_TEAM_WINS'].to_numpy()
cum_10_wins_home_H2H = df_g2['HOME_TEAM_WINS'].apply(cum_10_wins).to_frame().
 ↪explode('HOME_TEAM_WINS')['HOME_TEAM_WINS'].to_numpy()

    # HOME Losses!
cum_losses_home_H2H = df_g2['HOME_TEAM_WINS'].apply(cum_losses).to_frame().
 ↪explode('HOME_TEAM_WINS')['HOME_TEAM_WINS'].to_numpy()
cum_5_losses_home_H2H = df_g2['HOME_TEAM_WINS'].apply(cum_5_losses).to_frame().
 ↪explode('HOME_TEAM_WINS')['HOME_TEAM_WINS'].to_numpy()
cum_10_losses_home_H2H = df_g2['HOME_TEAM_WINS'].apply(cum_10_losses).
 ↪to_frame().explode('HOME_TEAM_WINS')['HOME_TEAM_WINS'].to_numpy()


# #df_clean = df_clean.reset_index()
df_clean2['cum_WINS_home_H2H'] = cum_wins_home_H2H
df_clean2['cum_LOSSES_home_H2H'] = cum_losses_home_H2H
df_clean2['cum_5_WINS_home_H2H'] = cum_5_wins_home_H2H
df_clean2['cum_10_WINS_home_H2H'] = cum_10_wins_home_H2H
df_clean2['cum_5_LOSSES_home_H2H'] = cum_5_losses_home_H2H
df_clean2['cum_10_LOSSES_home_H2H'] = cum_10_losses_home_H2H
```

[43]:
```
# Win-Loss Counter: AWAY HEAD-TO-HEAD

# Two Grouped DFs
df_g1 = df_clean2_temp.groupby(['TEAM_LIST', 'SEASON','GAME_ID']).mean()
df_g2 = df_clean2_temp.groupby(['TEAM_LIST']).agg(list)

# Using Functions above to calulate the cumulative values as lists

    # HOME WINS!
cum_wins_away_H2H = df_g2['AWAY_TEAM_WINS'].apply(cum_wins).to_frame().
 ↪explode('AWAY_TEAM_WINS')['AWAY_TEAM_WINS'].to_numpy()
cum_5_wins_away_H2H = df_g2['AWAY_TEAM_WINS'].apply(cum_5_wins).to_frame().
 ↪explode('AWAY_TEAM_WINS')['AWAY_TEAM_WINS'].to_numpy()
```

```
cum_10_wins_away_H2H = df_g2['AWAY_TEAM_WINS'].apply(cum_10_wins).to_frame().
 ↪explode('AWAY_TEAM_WINS')['AWAY_TEAM_WINS'].to_numpy()

    # HOME Losses!
cum_losses_away_H2H = df_g2['AWAY_TEAM_WINS'].apply(cum_losses).to_frame().
 ↪explode('AWAY_TEAM_WINS')['AWAY_TEAM_WINS'].to_numpy()
cum_5_losses_away_H2H = df_g2['AWAY_TEAM_WINS'].apply(cum_5_losses).to_frame().
 ↪explode('AWAY_TEAM_WINS')['AWAY_TEAM_WINS'].to_numpy()
cum_10_losses_away_H2H = df_g2['AWAY_TEAM_WINS'].apply(cum_10_losses).
 ↪to_frame().explode('AWAY_TEAM_WINS')['AWAY_TEAM_WINS'].to_numpy()


# #df_clean = df_clean.reset_index()
df_clean2['cum_WINS_away_H2H'] = cum_wins_away_H2H
df_clean2['cum_LOSSES_away_H2H'] = cum_losses_away_H2H
df_clean2['cum_5_WINS_away_H2H'] = cum_5_wins_away_H2H
df_clean2['cum_10_WINS_away_H2H'] = cum_10_wins_away_H2H
df_clean2['cum_5_LOSSES_away_H2H'] = cum_5_losses_away_H2H
df_clean2['cum_10_LOSSES_away_H2H'] = cum_10_losses_away_H2H
```

0.0.4 Vegas Betting Odds

We then load in our ODDS datasets for the required seasons. This dataset required cleaning for the names and dates to make it usable to join with the rest of the cumulative statistics. We then merge DF_CLEAN2 with ODDS using the TIMESTAMP and HOME_TEAM_NAME as keys, losing a few rows along the way due to missing data.

[44]:
```
# Cleaning the Odds DataFrame

cols = [col for col in df_clean2.columns if np.any(['5' in col, '10' in col])]
for col in cols:
    new_col = re.sub(r'_\d{1,2}_', '_', col)
    df_clean2[col] = df_clean2[col].fillna(df_clean2[new_col])


odds = pd.DataFrame()
for season in season_list:
    df = pd.read_excel('ODDS_DATA/odds_' + str(season) + '.xlsx', usecols =␣
 ↪['Date', 'Team', 'ML'])
    df['Date'] = df['Date'].apply(lambda x: '{0:0>4}'.format(x))

    new_year_index = df[df['Date'].str.contains(r'01\d{2}')].index[0]
    year1 = np.repeat(str(season), new_year_index-1)
    year2 = np.repeat(str(season+1), len(df) - len(year1))
    df['Year'] = np.append(year1, year2)
    df['TIMESTAMP'] = df['Year'] + df['Date']
    df = df.drop('Date', axis = 1)
```

```
    odds = odds.append(df)
home_teams, home_odds = odds.iloc[1::2]['Team'], odds.iloc[1::2]['ML']
away_teams, away_odds = odds.iloc[::2]['Team'], odds.iloc[::2]['ML']
dates = odds.iloc[1::2]['TIMESTAMP']
odds = pd.DataFrame({'TIMESTAMP' : list(dates),
                     'HOME_TEAM_NAME' : list(home_teams),
                     'AWAY_TEAM_NAME' : list(away_teams),
                     'HOME_TEAM_ODDS' : list(home_odds),
                     'AWAY_TEAM_ODDS' : list(away_odds)})
odds['HOME_TEAM_NAME'] = odds['HOME_TEAM_NAME'].str.replace(r'([a-z])([A-Z])',␣
 ↪r'\1 \2')
odds['AWAY_TEAM_NAME'] = odds['AWAY_TEAM_NAME'].str.replace(r'([a-z])([A-Z])',␣
 ↪r'\1 \2')

odds['TIMESTAMP'] = pd.to_datetime(odds['TIMESTAMP'], format = '%Y%m%d', errors␣
 ↪= 'coerce')
teams = pd.read_csv('FP1_DATA/teams.csv', usecols = ['CITY', 'NICKNAME'])
teams['CITY'][7] = 'LAClippers'
teams['CITY'][8] = 'LALakers'

odds['HOME_TEAM_NAME'] = odds['HOME_TEAM_NAME'].str.replace('New Jersey',␣
 ↪'Brooklyn')
odds['AWAY_TEAM_NAME'] = odds['AWAY_TEAM_NAME'].str.replace('New Jersey',␣
 ↪'Brooklyn')

team_city_dict = teams.set_index('CITY')['NICKNAME'].T.to_dict()
odds['HOME_TEAM_NAME'] = odds['HOME_TEAM_NAME'].map(team_city_dict)
odds['AWAY_TEAM_NAME'] = odds['AWAY_TEAM_NAME'].map(team_city_dict)

new_df = pd.merge(df_clean2, odds, on = ['TIMESTAMP', 'HOME_TEAM_NAME'])

cols = [col for col in new_df.columns if np.any(['5' in col, '10' in col])]
for col in cols:
    new_col = re.sub(r'_\d{1,2}_', '_', col)
    new_df[col] = new_df[col].fillna(new_df[new_col])

new_df = new_df.fillna(0)
new_df['SEASON'] = new_df['GAME_ID'].map(season_dict)

cum_col_list = np.append([col for col in list(new_df.columns) if 'cum' in col],␣
 ↪['HOME_TEAM_WINS', 'SEASON', 'GAME_ID'])
new_df = new_df[cum_col_list]
```

Lastly, we convert this final dataframe to a CSV to load into our FP3_MODELS notebook.

[45]:
```
new_df.to_csv('FP1.csv', index = False)
```

Appendix #2: X-Factor Feature Engineering

FP2_XFACTOR

May 9, 2021

[1]:
```
# Importing Libraries & Functions

#Normal Stuff
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import time
import re
import copy
from datetime import datetime, timedelta
import warnings
warnings.filterwarnings('ignore')
```

[3]:
```
# Loading + Cleaning Data
games = pd.read_csv('FP1_DATA/games.csv')
timestamp = pd.to_datetime(games['GAME_DATE_EST'])
games.insert(0, 'TIMESTAMP', timestamp)
games = games[games['SEASON'].
 ↪isin([2005,2006,2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,␣
 ↪2019])].sort_values('TIMESTAMP')

teams = pd.read_csv('FP1_DATA/teams.csv')
teams = teams[['TEAM_ID','NICKNAME']]
team_dict = teams.set_index('TEAM_ID').T.to_dict('list')
games['HOME_TEAM_NAME'] = games['HOME_TEAM_ID'].map(team_dict)
games['AWAY_TEAM_NAME'] = games['VISITOR_TEAM_ID'].map(team_dict)


drop_cols = ['GAME_DATE_EST', 'GAME_STATUS_TEXT', 'HOME_TEAM_ID',
             'VISITOR_TEAM_ID', 'TEAM_ID_home', 'TEAM_ID_away', 'SEASON']
games = games.drop(drop_cols, axis = 1).reset_index(drop = True)
front_cols = ['TIMESTAMP', 'GAME_ID', 'HOME_TEAM_NAME', 'AWAY_TEAM_NAME']
temp_games = games[front_cols]
games = pd.concat([temp_games, games.drop(front_cols, axis = 1)], axis = 1)
games['HOME_TEAM_NAME'] = games['HOME_TEAM_NAME'].apply(lambda x: x[0])
games['AWAY_TEAM_NAME'] = games['AWAY_TEAM_NAME'].apply(lambda x: x[0])
```

```
game_szn_dict = pd.read_csv('FP1_DATA/games.csv')[['GAME_ID','SEASON']].
 ↪set_index('GAME_ID').iloc[:, 0].T.to_dict()
games['SEASON'] = games['GAME_ID'].map(game_szn_dict)
games.head(5)
```

[3]:

	TIMESTAMP	GAME_ID	HOME_TEAM_NAME	AWAY_TEAM_NAME	PTS_home	FG_PCT_home	\
0	2005-10-10	10500120	Heat	Spurs	103.0	0.522	
1	2005-10-10	10500001	Wizards	Cavaliers	94.0	0.432	
2	2005-10-11	10500006	Lakers	Warriors	101.0	0.400	
3	2005-10-11	10500004	Pistons	Bulls	87.0	0.447	
4	2005-10-11	10500008	Cavaliers	Celtics	96.0	0.435	

	FT_PCT_home	FG3_PCT_home	AST_home	REB_home	PTS_away	FG_PCT_away	\
0	0.692	0.286	19.0	38.0	101.0	0.506	
1	0.610	0.333	18.0	41.0	116.0	0.507	
2	0.707	0.083	14.0	47.0	93.0	0.357	
3	0.652	0.444	27.0	35.0	76.0	0.338	
4	0.725	0.318	17.0	34.0	86.0	0.400	

	FT_PCT_away	FG3_PCT_away	AST_away	REB_away	HOME_TEAM_WINS	SEASON
0	0.652	0.400	23.0	37.0	1	2005
1	0.841	0.357	19.0	38.0	0	2005
2	0.729	0.296	16.0	44.0	1	2005
3	0.719	0.100	13.0	49.0	1	2005
4	0.783	0.286	20.0	45.0	1	2005

[13]:
```
# Creating Roster Data Frame
details = pd.read_csv('FP1_DATA/games_details.csv')
details = details[~details['START_POSITION'].isna()]
details['TEAM_NAME'] = details['TEAM_ID'].map(team_dict).apply(lambda x: x[0])

details = details[['GAME_ID', 'TEAM_NAME', 'PLAYER_NAME']]
details_grouped = details.groupby(['GAME_ID', 'TEAM_NAME'], as_index = False).
 ↪agg(list)

roster = pd.merge(details_grouped, games[['TIMESTAMP','GAME_ID',␣
 ↪'HOME_TEAM_NAME', 'AWAY_TEAM_NAME']], on = 'GAME_ID')
home_condition = (roster['TEAM_NAME'] == roster['HOME_TEAM_NAME'])
away_condition = (roster['TEAM_NAME'] != roster['HOME_TEAM_NAME'])

roster_home = roster[home_condition]
roster_home['ROSTER_home'] = roster_home['PLAYER_NAME']
roster_home = roster_home[['TIMESTAMP','GAME_ID', 'ROSTER_home']]

roster_away = roster[away_condition]
roster_away['ROSTER_away'] = roster_away['PLAYER_NAME']
```

```
roster_away = roster_away[['TIMESTAMP','GAME_ID', 'ROSTER_away']]

roster = pd.merge(roster_home, roster_away, on = 'GAME_ID')
roster['SEASON'] = roster['GAME_ID'].map(game_szn_dict)
roster = roster[roster['SEASON']<=2018]
roster = roster[roster['SEASON']>=2007]
roster = roster.drop(['TIMESTAMP_x', 'TIMESTAMP_y'], axis = 1)
roster = roster.sort_values(['SEASON','GAME_ID'])
roster
```

[13]:

	GAME_ID	ROSTER_home	ROSTER_away	SEASON
2997	20700001	[Bruce Bowen, Tim Duncan, Fabricio Oberto, Mic…	[Martell Webster, LaMarcus Aldridge, Joel Przy…	2007
2998	20700002	[Luke Walton, Ronny Turiaf, Kwame Brown, Kobe …	[Shane Battier, Chuck Hayes, Yao Ming, Tracy M…	2007
2999	20700003	[Kelenna Azubuike, Mickael Pietrus, Andris Bie…	[Andrei Kirilenko, Carlos Boozer, Mehmet Okur,…	2007
3000	20700004	[Jason Kapono, Chris Bosh, Andrea Bargnani, An…	[Andre Iguodala, Reggie Evans, Samuel Dalember…	2007
3001	20700005	[Hedo Turkoglu, Rashard Lewis, Dwight Howard, …	[Desmond Mason, Yi Jianlian, Andrew Bogut, Mic…	2007
…	…	…	…	…
19740	41800402	[Kawhi Leonard, Pascal Siakam, Marc Gasol, Dan…	[Andre Iguodala, Draymond Green, DeMarcus Cous…	2018
19741	41800403	[Andre Iguodala, Draymond Green, DeMarcus Cous…	[Kawhi Leonard, Pascal Siakam, Marc Gasol, Dan…	2018
19742	41800404	[Andre Iguodala, Draymond Green, DeMarcus Cous…	[Kawhi Leonard, Pascal Siakam, Marc Gasol, Dan…	2018
19743	41800405	[Kawhi Leonard, Pascal Siakam, Marc Gasol, Dan…	[Andre Iguodala, Kevin Durant, Draymond Green,…	2018
19744	41800406	[Andre Iguodala, Draymond Green, Kevon Looney,…	[Kawhi Leonard, Pascal Siakam, Marc Gasol, Dan…	2018

[15960 rows x 4 columns]

[8]:
```
# Games that we will use, from the 2007 season thru the 2018 season
games1 = games[games['GAME_ID'].isin(roster['GAME_ID'].to_list())]
games1 = games1.sort_values(['TIMESTAMP'])
games1
```

[8]:

	TIMESTAMP	GAME_ID	HOME_TEAM_NAME	AWAY_TEAM_NAME	PTS_home
2946	2007-10-30	20700003	Warriors	Jazz	96.0
2947	2007-10-30	20700002	Lakers	Rockets	93.0
2948	2007-10-30	20700001	Spurs	Trail Blazers	106.0

```
2956  2007-10-31  20700007           Nets          Bulls     112.0
2954  2007-10-31  20700010       Pelicans          Kings     104.0
…          …       …              …             …       …
19478 2019-06-02  41800402        Raptors       Warriors     104.0
19479 2019-06-05  41800403       Warriors        Raptors     109.0
19480 2019-06-07  41800404       Warriors        Raptors      92.0
19481 2019-06-10  41800405        Raptors       Warriors     105.0
19482 2019-06-13  41800406       Warriors        Raptors     110.0

       FG_PCT_home  FT_PCT_home  FG3_PCT_home  AST_home  REB_home  PTS_away  \
2946         0.416        0.684         0.261      19.0      37.0     117.0
2947         0.421        0.600         0.250      18.0      37.0      95.0
2948         0.471        0.692         0.250      21.0      40.0      97.0
2956         0.402        0.902         0.375      24.0      48.0     103.0
2954         0.506        0.762         0.476      23.0      44.0      90.0
…            …            …             …          …         …         …
19478        0.372        0.885         0.289      17.0      49.0     109.0
19479        0.396        0.833         0.333      25.0      41.0     123.0
19480        0.449        0.667         0.296      26.0      42.0     105.0
19481        0.447        0.778         0.250      19.0      43.0     106.0
19482        0.488        0.700         0.355      28.0      42.0     114.0

       FG_PCT_away  FT_PCT_away  FG3_PCT_away  AST_away  REB_away  \
2946         0.456        0.833         0.455      24.0      56.0
2947         0.459        0.677         0.273      23.0      49.0
2948         0.500        0.765         0.462      15.0      40.0
2956         0.396        0.731         0.348      23.0      45.0
2954         0.444        0.833         0.300      21.0      34.0
…            …            …             …          …         …
19478        0.463        0.870         0.382      34.0      42.0
19479        0.524        0.952         0.447      30.0      40.0
19480        0.419        0.958         0.313      22.0      39.0
19481        0.463        0.714         0.476      27.0      37.0
19482        0.476        0.793         0.394      25.0      39.0

       HOME_TEAM_WINS  SEASON
2946                0    2007
2947                0    2007
2948                1    2007
2956                1    2007
2954                1    2007
…                 …      …
19478               0    2018
19479               0    2018
19480               0    2018
19481               0    2018
19482               0    2018
```

```
[15960 rows x 18 columns]
```

0.1 Feature #1 : NBA Awards

The cell below loads in the AWARDS dataframe, and calculates the number of players on the HOME/AWAY team's rosters who received awards for Most-Valuable Player (MVP), Defensive Player of the Year (DPOY), Rookie of the Year (ROY), and Sixth Man of the Year (6MOY) in the last four years.

[16]:
```
pd.read_csv('FP2_DATA/awards.csv')
```

[16]:

	SEASON	MVP	DPOY	ROY	6MOY
0	2019	Giannis Antetokounmpo	Giannis Antetokounmpo	Ja Morant	Montrezl Harrell
1	2018	Giannis Antetokounmpo	Rudy Gobert	Luka Dončić	Lou Williams
2	2017	James Harden	Rudy Gobert	Ben Simmons	Lou Williams
3	2016	Russell Westbrook	Draymond Green	Malcolm Brogdon	Eric Gordon
4	2015	Stephen Curry	Kawhi Leonard	Karl-Anthony Towns	Jamal Crawford
5	2014	Stephen Curry	Kawhi Leonard	Andrew Wiggins	Lou Williams
6	2013	Kevin Durant	Joakim Noah	Michael Carter-Williams	Jamal Crawford
7	2012	LeBron James	Marc Gasol	Damian Lillard	J.R. Smith
8	2011	LeBron James	Tyson Chandler	Kyrie Irving	James Harden
9	2010	Derrick Rose	Dwight Howard	Blake Griffin	Lamar Odom
10	2009	LeBron James	Dwight Howard	Tyreke Evans	Jamal Crawford
11	2008	LeBron James	Dwight Howard	Derrick Rose	Jason Terry
12	2007	Kobe Bryant	Kevin Garnett	Kevin Durant	Manu Ginóbili
13	2006	Dirk Nowitzki	Marcus Camby	Brandon Roy	Leandro Barbosa
14	2005	Steve Nash	Ben Wallace		
15	2004	Steve Nash	Ben Wallace		
16	2003	Kevin Garnett	Ron Artest		
17	2002	Tim Duncan	Ben Wallace		

```
14              Chris Paul     Bobby Jackson
15            Emeka Okafor        Ben Gordon
16            LeBron James    Antawn Jamison
17       Amar'e Stoudemire     Bobby Jackson
```

[17]:
```
pd.read_csv('FP2_DATA/awards.csv').iloc[:15, :5]
```

[17]:
```
    SEASON                   MVP                  DPOY  \
0     2019  Giannis Antetokounmpo  Giannis Antetokounmpo
1     2018  Giannis Antetokounmpo            Rudy Gobert
2     2017           James Harden            Rudy Gobert
3     2016      Russell Westbrook         Draymond Green
4     2015          Stephen Curry          Kawhi Leonard
5     2014          Stephen Curry          Kawhi Leonard
6     2013           Kevin Durant            Joakim Noah
7     2012           LeBron James             Marc Gasol
8     2011           LeBron James         Tyson Chandler
9     2010           Derrick Rose          Dwight Howard
10    2009           LeBron James          Dwight Howard
11    2008           LeBron James          Dwight Howard
12    2007            Kobe Bryant          Kevin Garnett
13    2006          Dirk Nowitzki           Marcus Camby
14    2005             Steve Nash            Ben Wallace

                       ROY              6MOY
0                Ja Morant  Montrezl Harrell
1              Luka Dončić      Lou Williams
2              Ben Simmons      Lou Williams
3           Malcolm Brogdon       Eric Gordon
4        Karl-Anthony Towns    Jamal Crawford
5             Andrew Wiggins      Lou Williams
6   Michael Carter-Williams    Jamal Crawford
7             Damian Lillard        J.R. Smith
8               Kyrie Irving      James Harden
9              Blake Griffin        Lamar Odom
10              Tyreke Evans    Jamal Crawford
11              Derrick Rose       Jason Terry
12              Kevin Durant     Manu Ginóbili
13               Brandon Roy   Leandro Barbosa
14                Chris Paul     Bobby Jackson
```

[22]:
```
# Feature #1: NBA Awards
def prev_4(award, awards):
    award_list = np.array(list(awards[award]))
    prev_4_list = []
    for i in np.arange(13):
        prev_4_list.append(award_list[np.arange(i+1, i+5)])
```

```
    return prev_4_list

awards = pd.read_csv('FP2_DATA/awards.csv')
awards['SEASON'] = awards['SEASON'].astype(int)
full_list = []
award_names = ['MVP', 'DPOY', 'ROY', '6MOY']
for award in award_names:
    temp_4_list = prev_4(award, awards)
    full_list.append(temp_4_list)
awards = awards.iloc[:13]
for i in np.arange(len(full_list)):
    awards['PREV_4_' + str(award_names[i])] = full_list[i]
    awards = awards.drop(award_names[i], axis = 1)

df_X1 = pd.merge(roster, awards, on = 'SEASON')
def count_award_home(row, award_name):
    return sum(item in row['ROSTER_home'] for item in row['PREV_4_' +␣
 ↪str(award_name)])
def count_award_away(row, award_name):
    return sum(item in row['ROSTER_away'] for item in row['PREV_4_' +␣
 ↪str(award_name)])


award_name = '6MOY'
for award_name in award_names:
    df_X1['COUNT_PREV_4_' + award_name + '_home'] = df_X1.apply(lambda row:␣
 ↪count_award_home(row, award_name), axis = 1)
    df_X1['COUNT_PREV_4_' + award_name + '_away'] = df_X1.apply(lambda row:␣
 ↪count_award_away(row, award_name), axis = 1)
    df_X1 = df_X1.drop('PREV_4_' + str(award_name), axis = 1)
award_df = df_X1.drop(['ROSTER_home', 'ROSTER_away', 'SEASON'], axis = 1)
award_df
```

[22]:

	GAME_ID	COUNT_PREV_4_MVP_home	COUNT_PREV_4_MVP_away	\
0	20700001	0	0	
1	20700002	0	0	
2	20700003	0	0	
3	20700004	0	0	
4	20700005	0	0	
…	…	…	…	
15955	41800402	0	2	
15956	41800403	2	0	
15957	41800404	2	0	
15958	41800405	0	2	
15959	41800406	2	0	

	COUNT_PREV_4_DPOY_home	COUNT_PREV_4_DPOY_away	COUNT_PREV_4_ROY_home	\

```
0                          0                          0                          0
1                          0                          0                          0
2                          0                          0                          0
3                          0                          0                          0
4                          0                          0                          0
…                          …                          …                          …
15955                      2                          1                          0
15956                      1                          2                          0
15957                      1                          2                          0
15958                      2                          1                          0
15959                      1                          2                          0

       COUNT_PREV_4_ROY_away  COUNT_PREV_4_6MOY_home  COUNT_PREV_4_6MOY_away
0                          1                          0                          0
1                          0                          0                          0
2                          0                          0                          0
3                          0                          0                          0
4                          0                          0                          0
…                          …                          …                          …
15955                      0                          0                          0
15956                      0                          0                          0
15957                      0                          0                          0
15958                      0                          0                          0
15959                      0                          0                          0

[15960 rows x 9 columns]
```

0.2 Feature #2 : The Lebron Effect

The cell below outputs a binary column for whether Lebron James is on the roster for the HOME or AWAY team.

[24]:
```python
# Feature #2: The Lebron Effect
df_X2 = copy.copy(roster)
def lebron_home(row):
    home_roster = row['ROSTER_home']
    if 'LeBron James' in home_roster:
        return 1
    return 0

def lebron_away(row):
    away_roster = row['ROSTER_away']
    if 'LeBron James' in away_roster:
        return 1
    return 0

df_X2['LEBRON_home'] = df_X2.apply(lambda row: lebron_home(row), axis = 1)
```

```
df_X2['LEBRON_away'] = df_X2.apply(lambda row: lebron_away(row), axis = 1)

lebron_df = df_X2.drop(['ROSTER_home', 'ROSTER_away', 'SEASON'], axis = 1)
lebron_df
```

[24]:

	GAME_ID	LEBRON_home	LEBRON_away
2997	20700001	0	0
2998	20700002	0	0
2999	20700003	0	0
3000	20700004	0	0
3001	20700005	0	0
…	…	…	…
19740	41800402	0	0
19741	41800403	0	0
19742	41800404	0	0
19743	41800405	0	0
19744	41800406	0	0

[15960 rows x 3 columns]

0.3 Feature #3 : All-NBA Appearances

The cell below calculate the number of players on the HOME/AWAY roster who were voted on the NBA First-Team, Second-Team, and Third-Team in the last four years.

[30]:
```
# Feature #3 : All-NBA Appearances
first_team = pd.read_csv('FP2_DATA/first_team.csv')
second_team = pd.read_csv('FP2_DATA/second_team.csv')
third_team = pd.read_csv('FP2_DATA/third_team.csv')

first_team['SEASON'] = first_team['SEASON'].astype(int)
second_team['SEASON'] = second_team['SEASON'].astype(int)
third_team['SEASON'] = third_team['SEASON'].astype(int)


full_ls1, full_ls2, full_ls3 = [], [], []
for i in range(len(first_team)-4):
    temp_ls1, temp_ls2, temp_ls3 = [], [], []
    for row in first_team.values[i+1:i+5]:
        temp_ls1 = np.append(temp_ls1, row[1:])
    for row in second_team.values[i+1:i+5]:
        temp_ls2 = np.append(temp_ls2, row[1:])
    for row in third_team.values[i+1:i+5]:
        temp_ls3 = np.append(temp_ls3, row[1:])
    full_ls1.append(temp_ls1)
    full_ls2.append(temp_ls2)
    full_ls3.append(temp_ls3)
```

```
seasons =␣
 ↪[2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020]
all_nba = pd.DataFrame({'SEASON':seasons[:],
                        'PREV_first_team': full_ls1,
                        'PREV_second_team': full_ls2,
                        'PREV_third_team': full_ls3})
df_X3 = pd.merge(roster, all_nba, on='SEASON')

def count_all_nba_home(row, team_name):
    return sum(item in row['ROSTER_home'] for item in row['PREV_' +␣
 ↪str(team_name)])
def count_all_nba_away(row, team_name):
    return sum(item in row['ROSTER_away'] for item in row['PREV_' +␣
 ↪str(team_name)])

team_names = ['first_team', 'second_team', 'third_team']
for team_name in team_names:
    df_X3['COUNT_PREV_4_' + team_name] = df_X3.apply(lambda row:␣
 ↪count_all_nba_home(row, team_name), axis = 1)
    df_X3['COUNT_PREV_4_' + team_name] = df_X3.apply(lambda row:␣
 ↪count_all_nba_away(row, team_name), axis = 1)
    df_X3 = df_X3.drop('PREV_' + str(team_name), axis = 1)

all_nba_df = df_X3.drop(['ROSTER_home', 'ROSTER_away', 'SEASON'], axis = 1)
all_nba_df
```

[30]:

	GAME_ID	COUNT_PREV_4_first_team	COUNT_PREV_4_second_team	\
0	20700001	0	1	
1	20700002	0	0	
2	20700003	0	0	
3	20700004	0	0	
4	20700005	0	0	
…	…	…	…	
15955	41800402	0	0	
15956	41800403	0	0	
15957	41800404	0	0	
15958	41800405	0	0	
15959	41800406	0	0	

	COUNT_PREV_4_third_team
0	1
1	0
2	0
3	0
4	0
…	…

```
15955                          0
15956                          0
15957                          0
15958                          0
15959                          0

[15960 rows x 4 columns]
```

0.4 Feature #4 : Home-Court Advantage

The cell below contains the winning percentage at home for each HOME team over the last five years to approximate the home-court advantage.

[32]:
```
# Feature #4 : Home-Court Advantage
games2 = copy.copy(games)
games2 = games2[['TIMESTAMP', 'GAME_ID', 'SEASON', 'HOME_TEAM_NAME',␣
↪'HOME_TEAM_WINS']]

df_X4 = copy.copy(games1[['TIMESTAMP', 'GAME_ID', 'SEASON', 'HOME_TEAM_NAME']])

def home_win_pct_past4(row):
    curr_date = row['TIMESTAMP']
    curr_season = row['SEASON']
    home_team = row['HOME_TEAM_NAME']
    temp_games = games2[games2['TIMESTAMP']<curr_date]
    temp_games = temp_games[curr_season-temp_games['SEASON'] < 4]
    temp_games = temp_games[temp_games['HOME_TEAM_NAME']==home_team]

    home_wins_past4 = sum(temp_games['HOME_TEAM_WINS'])
    home_games_past4 = len(temp_games)
    return np.round(home_wins_past4/home_games_past4,3)

df_X4['HOME_TEAM_home_win_pct_past4yrs'] = df_X4.apply(lambda row:␣
↪home_win_pct_past4(row), axis = 1)
home_advantage_df = df_X4#.drop(['TIMESTAMP', 'SEASON', 'HOME_TEAM_NAME'],␣
↪axis=1)
home_advantage_df
```

[32]:
```
       TIMESTAMP   GAME_ID  SEASON HOME_TEAM_NAME  \
2946  2007-10-30  20700003    2007       Warriors
2947  2007-10-30  20700002    2007         Lakers
2948  2007-10-30  20700001    2007          Spurs
2956  2007-10-31  20700007    2007           Nets
2954  2007-10-31  20700010    2007       Pelicans
…            …         …       …              …
19478 2019-06-02  41800402    2018        Raptors
19479 2019-06-05  41800403    2018       Warriors
```

```
19480 2019-06-07  41800404    2018       Warriors
19481 2019-06-10  41800405    2018        Raptors
19482 2019-06-13  41800406    2018       Warriors

       HOME_TEAM_home_win_pct_past4yrs
2946                             0.624
2947                             0.615
2948                             0.755
2956                             0.650
2954                             0.560
…                                …
19478                            0.762
19479                            0.808
19480                            0.805
19481                            0.758
19482                            0.801

[15960 rows x 5 columns]
```

0.5 Feature #5 / #6 : Altitude and Timezone Difference

The cell below contains the difference in altitude between the HOME and AWAY team.

[34]:
```
# Feature #5 / #6 : Altitude and Timezone Difference

# Game Season Dictionary
game_szn_dict = pd.read_csv('FP1_DATA/games.csv')[['GAME_ID','SEASON']].
 ↪set_index('GAME_ID').iloc[:, 0].T.to_dict()
games['SEASON'] = games['GAME_ID'].map(game_szn_dict)


altitude_timezone = pd.read_csv('FP2_DATA/altitude_timezone.csv')
alt_team_dict = {}
for row in altitude_timezone.values:
    alt_team_dict[row[0],row[1],row[2]] = row[3]
timezone_team_dict = {}
for row in altitude_timezone.values:
    timezone_team_dict[row[0],row[1],row[2]] = row[4]

df_X5 = copy.copy(games1[['GAME_ID', 'SEASON', 'HOME_TEAM_NAME',␣
 ↪'AWAY_TEAM_NAME']])

def home_team_alt(row):
    for key in alt_team_dict:
        if row['HOME_TEAM_NAME']==key[0] and np.all([row['SEASON']+1>=key[1],␣
 ↪row['SEASON']+1<=key[2]]):
            return alt_team_dict[key]
```

```
def away_team_alt(row):
    for key in alt_team_dict:
        if row['AWAY_TEAM_NAME']==key[0] and np.all([row['SEASON']+1>=key[1],␣
↪row['SEASON']+1<=key[2]]):
            return alt_team_dict[key]

def home_team_timezone(row):
    for key in timezone_team_dict:
        if row['HOME_TEAM_NAME']==key[0] and np.all([row['SEASON']+1>=key[1],␣
↪row['SEASON']+1<=key[2]]):
            return timezone_team_dict[key]
def away_team_timezone(row):
    for key in timezone_team_dict:
        if row['AWAY_TEAM_NAME']==key[0] and np.all([row['SEASON']+1>=key[1],␣
↪row['SEASON']+1<=key[2]]):
            return timezone_team_dict[key]

df_X5['HOME_TEAM_altitude'] = df_X5.apply(lambda row: home_team_alt(row), axis␣
↪= 1)
df_X5['AWAY_TEAM_altitude'] = df_X5.apply(lambda row: away_team_alt(row), axis␣
↪= 1)
df_X5['AWAY_TEAM_delta_altitude'] =␣
↪df_X5['HOME_TEAM_altitude']-df_X5['AWAY_TEAM_altitude']

df_X5['HOME_TEAM_timezone'] = df_X5.apply(lambda row: home_team_timezone(row),␣
↪axis = 1)
df_X5['AWAY_TEAM_timezone'] = df_X5.apply(lambda row: away_team_timezone(row),␣
↪axis = 1)
df_X5['AWAY_TEAM_delta_timezone'] =␣
↪df_X5['HOME_TEAM_timezone']-df_X5['AWAY_TEAM_timezone']

altitude_timezone_df = copy.
↪copy(df_X5[['GAME_ID','AWAY_TEAM_delta_altitude','AWAY_TEAM_delta_timezone']])
```

Finally, we merge all of these dataframes together using 'GAME_ID' as the key, and output this dataframe as a CSV to be loaded into our FP3_MODELS notebook.

[37]:
```
XFactor_df = pd.merge(award_df, lebron_df, on='GAME_ID').merge(all_nba_df,␣
↪on='GAME_ID').merge(home_advantage_df, on='GAME_ID').
↪merge(altitude_timezone_df, on='GAME_ID')
XFactor_df.to_csv('FP2.csv', index=False)
```

[ ]:
```

```

Appendix #3: Approach #1 (Modified Cross-Validation) Modeling and Analysis

FP3_MODELS

May 9, 2021

[145]:
```
# Importing Libraries & Functions
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import time
import re
# import warnings
# warnings.filterwarnings('ignore')
from IPython.lib.display import Audio
framerate = 4410

#SciKit Learn
from sklearn.model_selection import GridSearchCV, KFold, train_test_split
from sklearn.metrics import r2_score, mean_squared_error, confusion_matrix,␣
 ↪mean_absolute_error
from sklearn.metrics import accuracy_score, auc, roc_curve, roc_auc_score,␣
 ↪accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.dummy import DummyClassifier
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
import statsmodels.formula.api as smf

import xgboost as xgb
from xgboost import XGBClassifier
from sklearn.neural_network import MLPClassifier


def outputs(cm):
    acc = np.round((cm.ravel()[0]+cm.ravel()[3])/sum(cm.ravel()),4)
    tpr = np.round(cm.ravel()[3] / (cm.ravel()[3] + cm.ravel()[2]),4)
    fpr = np.round(cm.ravel()[1] / (cm.ravel()[1] + cm.ravel()[0]),4)
    outputs = [acc, tpr, fpr]
    return outputs
```

0.1 Loading Data + Train/Test Split

First, we load in the two CSVs from the FP1_CUM_STATS and FP2_XFACTOR notebooks and join them on the 'GAME_ID' as the key. We then set our training data to be all data from the 2010 Season to the 2016 Season, and our testing data to be all data from the 2017 Season to the 2018 Season. The cell below performs this task, and outputs the Counts and Percentages of Training and Testing data. We also create categorical encoded variables for the X-Factor features from FP2.

[159]:
```
# Splitting Data
FP1 = pd.read_csv('FP1.csv')
FP2 = pd.read_csv('FP2.csv')
df = pd.merge(FP1, FP2, on = 'GAME_ID').rename(columns={'SEASON_y':'SEASON'})
df = df.drop(['TIMESTAMP','HOME_TEAM_NAME', 'SEASON_x'], axis=1)

matchers = ['COUNT_PREV_4','Lebron']
cat_cols = [col for col in df.columns if any(xs in col for xs in matchers)]
for cat_col in cat_cols:
    df[cat_col] = df[cat_col].astype(str)
cat_df = pd.get_dummies(df[cat_cols], drop_first = False)
cat_df['GAME_ID'] = df['GAME_ID']
df = df.merge(cat_df, on = 'GAME_ID')

train_df = df[df['SEASON'].between(2007,2016)]
test_df = df[df['SEASON'].between(2017,2018)]

train_df, test_df = train_df.drop(['SEASON', 'GAME_ID'], axis = 1), test_df.
 ↪drop(['SEASON', 'GAME_ID'], axis = 1)
X_train, y_train = train_df.drop('HOME_TEAM_WINS', axis = 1),␣
 ↪train_df['HOME_TEAM_WINS']
X_test, y_test = test_df.drop('HOME_TEAM_WINS', axis = 1),␣
 ↪test_df['HOME_TEAM_WINS']

train_prop = np.round(len(X_train) / (len(df)), 3)
test_prop = np.round(len(X_test) / (len(df)), 3)

print('Training Data: ' + str(len(X_train)) + ' rows -- ' + str(np.
 ↪round(train_prop*100,2)) + '%')
print('Testing Data: ' + str(len(X_test)) + ' rows -- ' + str(np.
 ↪round(test_prop*100,2)) + '%')
```

```
Training Data: 12849 rows -- 83.0%
Testing Data: 2626 rows -- 17.0%
```

0.2 Model #0: Baseline (Dummy) Model

Our dummy model predicts the most frequent label in the training set, which is HOME_TEAM_WINS = 1. This is the same as always predicting the home team to win.

[8]:
```
# Baseline Model
model_0 = DummyClassifier(strategy = "most_frequent")
model_0.fit(X_train, y_train)
y_pred_0 = model_0.predict(X_test)

# Confusion Matrix & Outputs
cm_0 = confusion_matrix(y_test, y_pred_0)
outputs_0 = outputs(cm_0)
print ("\nConfusion Matrix : \n", cm_0)
print ("\nAccuracy : ", outputs_0[0])
```

```
Confusion Matrix :
 [[   0 1081]
 [   0 1545]]

Accuracy :  0.5883
```

0.3 Model #1: Logistic Regression

The cell below cretes a Logistic Regression model and calculates the outputs to be presented later. We tested various L2 Penalty values and found 0.5 to be sufficient.

[19]:
```
# Logistic Regression

model_1 = LogisticRegression(random_state = 69, tol = 0.001, max_iter = 500,
                             verbose = 3, n_jobs = -1, penalty = 'l2',
                             fit_intercept = True, C = 0.5)
model_1.fit(X_train, y_train)
y_prob_1 = model_1.predict_proba(X_test)
y_pred_1 = pd.Series([1 if x > 0.5 else 0 for x in y_prob_1[:,1]], index =␣
 ↪y_test.index)

# Confusion Matrix & Outputs
cm_1 = confusion_matrix(y_test, y_pred_1)
outputs_1 = outputs(cm_1)
print ("\nConfusion Matrix : \n", cm_1)
print ("\nAccuracy : ", outputs_1[0])
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done   1 out of   1 | elapsed:  1.7min finished


Confusion Matrix :
 [[ 458  623]
 [ 275 1270]]

Accuracy :  0.658
```

0.4 Model #2: Random Forest

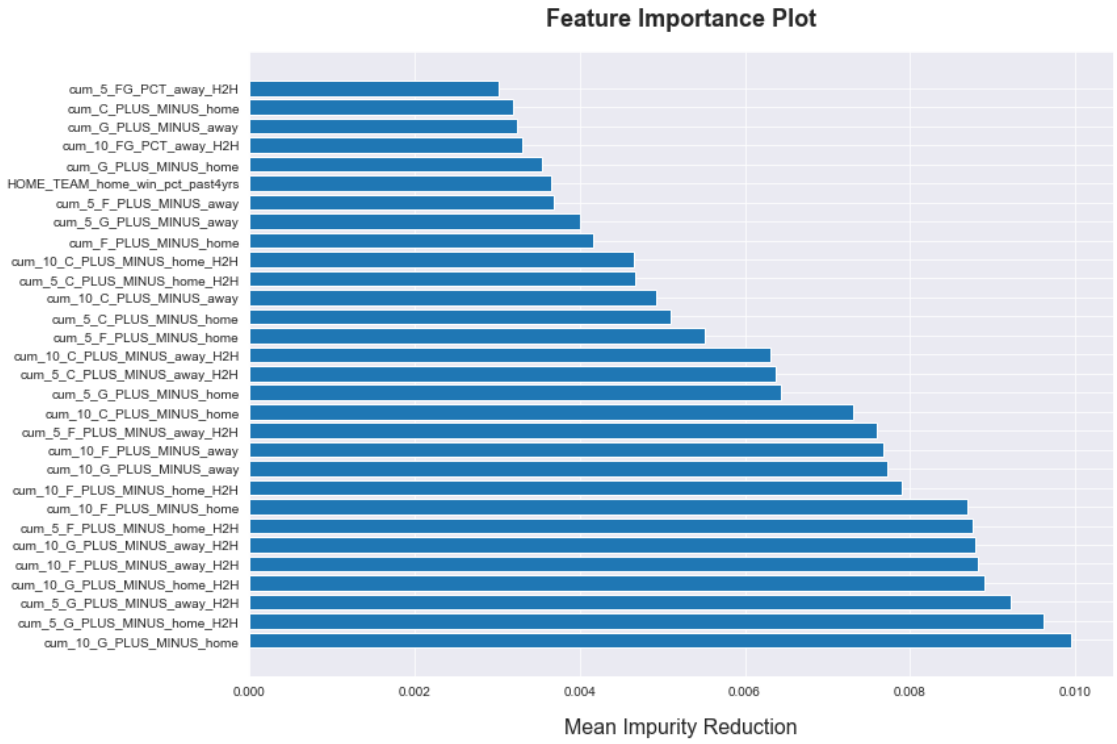
This cell below is a Random Forest model with cross-validation on various parameters.

[23]:
```
# First Model w Reasonable Values

model_2 = RandomForestClassifier(n_estimators = 1000, random_state = 69, n_jobs␣
↪= -1,
                                 min_samples_split = 5, min_samples_leaf = 5,␣
↪verbose = 1)
#model_2 = RandomForestClassifier(n_estimators = 1000, random_state = 69,␣
↪n_jobs = -1, verbose = 3)
model_2.fit(X_train, y_train)

y_prob_2 = model_2.predict_proba(X_test)
y_pred_2 = pd.Series([1 if x > 0.5 else 0 for x in y_prob_2[:,1]], index =␣
↪y_test.index)
#
# Confusion Matrix & Outputs
cm_2 = confusion_matrix(y_test, y_pred_2)
outputs_2 = outputs(cm_2)
print ("\nConfusion Matrix : \n", cm_2)
print ("\nAccuracy : ", outputs_2[0])
```

```
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:    3.6s
[Parallel(n_jobs=-1)]: Done 192 tasks      | elapsed:   15.7s
[Parallel(n_jobs=-1)]: Done 442 tasks      | elapsed:   35.3s
[Parallel(n_jobs=-1)]: Done 792 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 1000 out of 1000 | elapsed:  1.4min finished
[Parallel(n_jobs=4)]: Using backend ThreadingBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  42 tasks      | elapsed:    0.0s
[Parallel(n_jobs=4)]: Done 192 tasks      | elapsed:    0.1s
[Parallel(n_jobs=4)]: Done 442 tasks      | elapsed:    0.3s


Confusion Matrix :
 [[ 430  651]
 [ 267 1278]]

Accuracy :  0.6504

[Parallel(n_jobs=4)]: Done 792 tasks      | elapsed:    0.5s
[Parallel(n_jobs=4)]: Done 1000 out of 1000 | elapsed:    0.6s finished
```

[28]:
```
# Selecting top features
out = pd.DataFrame({'Feature' : list(X_train.columns),
                    'Importance Score': model_2.feature_importances_}).
↪sort_values('Importance Score', ascending = False)
```

```
top = out.iloc[:30,:]['Feature'].values
# top = out[out['Importance Score'] > 0.0018]['Feature'].values
out_30 = out.iloc[:30,:]
sns.set_style('darkgrid')
fig, ax = plt.subplots(figsize = (12,8))
plt.barh(width = out_30['Importance Score'], y = out_30['Feature'])
plt.title('Feature Importance Plot', fontsize = 18, fontweight = 'bold', pad =␣
↪20)
plt.xlabel('Mean Impurity Reduction', fontsize = 16, labelpad = 15)
fig.tight_layout()
plt.savefig('fig1.pdf')
```



[29]:
```
# GRID SEARCH ON TOP 30
X_train_top = X_train[top]

grid = {'max_features': np.arange(1, len(X_train_top.columns)+1)}


model_2 = RandomForestClassifier(n_estimators = 200, random_state = 69, n_jobs␣
↪= -1,
                                 min_samples_split = 5, min_samples_leaf = 5)

model_2 = GridSearchCV(model_2, param_grid=grid, scoring='accuracy',
```

```
                        cv=5, verbose=2, n_jobs = -1).fit(X_train_top, y_train)

print('Optimal Max Features: ', model_2.best_params_)
y_prob_2 = model_2.best_estimator_.predict_proba(X_test[top])
y_pred_2 = pd.Series([1 if x > 0.5 else 0 for x in y_prob_2[:,1]], index =␣
 ↪y_test.index)

# Confusion Matrix & Outputs
cm_2 = confusion_matrix(y_test, y_pred_2)
outputs_2 = outputs(cm_2)
print ("\nConfusion Matrix : \n", cm_2)
print ("\nAccuracy : ", outputs_2[0])
```

```
Fitting 5 folds for each of 30 candidates, totalling 150 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  33 tasks      | elapsed:  1.5min
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed: 20.8min finished

Optimal Max Features:  {'max_features': 1}

Confusion Matrix :
 [[ 475  606]
 [ 307 1238]]

Accuracy :  0.6523
```

[63]:
```
# Run our Best Model with Top 30 Features and Optimal Max Features

model_2 = RandomForestClassifier(max_features = 1, n_estimators = 500,␣
 ↪random_state = 69,
                                 n_jobs = -1, min_samples_split = 8,␣
 ↪min_samples_leaf = 2, verbose = 1)
model_2.fit(X_train_top, y_train)

y_prob_2 = model_2.predict_proba(X_test[top])
y_pred_2 = pd.Series([1 if x > 0.5 else 0 for x in y_prob_2[:,1]], index =␣
 ↪y_test.index)
#
# Confusion Matrix & Outputs
cm_2 = confusion_matrix(y_test, y_pred_2)
outputs_2 = outputs(cm_2)
print ("\nConfusion Matrix : \n", cm_2)
print ("\nAccuracy : ", outputs_2[0])
```

```
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:    0.3s
[Parallel(n_jobs=-1)]: Done 192 tasks      | elapsed:    1.2s
```

```
[Parallel(n_jobs=-1)]: Done 442 tasks      | elapsed:    2.7s
[Parallel(n_jobs=-1)]: Done 500 out of 500 | elapsed:    3.1s finished
[Parallel(n_jobs=4)]: Using backend ThreadingBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  42 tasks      | elapsed:    0.0s
[Parallel(n_jobs=4)]: Done 192 tasks      | elapsed:    0.1s


Confusion Matrix :
 [[ 481  600]
 [ 302 1243]]

Accuracy :  0.6565

[Parallel(n_jobs=4)]: Done 442 tasks      | elapsed:    0.3s
[Parallel(n_jobs=4)]: Done 500 out of 500 | elapsed:    0.3s finished
```

0.5 Model #3: XGBoost

The cells below go through an entire process of creating and cross-validating an XGBoost model. First, we set default parameters and using the XGB Cross-Validation function to determine the optimal number of trees for the specified learning rate. We then use this number of estimators to cross-validate and select an approximate values for *max_depth* and *min_child_weight*, which we then find more optimally by shrinking the search space.

[66]:
```
# Defining the Cross-Validation Function
def modelfit(alg, train, predictors, useTrainCV = True, cv_folds = 5,␣
↪early_stopping_rounds = 50):

    if useTrainCV:
        xgb_param = alg.get_xgb_params()
        xgtrain = xgb.DMatrix(train[predictors].values, label=train[target].
↪values)
        cvresult = xgb.cv(xgb_param, xgtrain, num_boost_round=alg.
↪get_params()['n_estimators'], nfold = cv_folds,
                          metrics = 'error', early_stopping_rounds =␣
↪early_stopping_rounds, verbose_eval = True)
        alg.set_params(n_estimators = cvresult.shape[0])

    #Fit the algorithm on the data
    alg.fit(train[predictors], train['HOME_TEAM_WINS'], eval_metric = 'error')

    #Predict training set:
    train_predictions = alg.predict(train[predictors])
    train_predprob = alg.predict_proba(train[predictors])[:,1]

    #Print model report:
    print("\nModel Report")
    print("Accuracy : %.4g" % accuracy_score(train['HOME_TEAM_WINS'].values,␣
↪train_predictions))
```

```
    print("AUC Score (Train): %f" % roc_auc_score(train['HOME_TEAM_WINS'],␣
 ↪train_predprob))
```

Run the first cell below and the final number it outputs in square brackets is the n_estimators that you plug into gridsearchcv object below.

STEP #1 : Calculate n_estimators for learning_rate = 0.1 (VERY HIGH)

STEP #2 : Use Cross-Validation to tune the other hyperparameters

STEP #3 : Now reduce the learning rate to 0.01 or something smaller and calculate the new n_estimators for the hyperparameters found above. Obviously, this will mean much more trees are used.

STEP #4 : Run our final model using the learning rate set above, the hyperparameters found using cross-validation, and the corresponding (now higher) n_estimators.

[67]:
```
# Determining Optimal Number of Estimators
X_train[cat_cols] = X_train[cat_cols].astype(int).iloc[0,4]
target = 'HOME_TEAM_WINS'
train = pd.concat([X_train, y_train], axis = 1)
predictors = [x for x in train.columns if x not in [target]]

model_3 = XGBClassifier(learning_rate = 0.1, n_estimators = 1000, max_depth =␣
 ↪5, min_child_weight = 1,
                      gamma = 0, subsample = 0.8, colsample_bytree = 0.8,␣
 ↪objective = 'binary:logistic',
                      nthread = 4, scale_pos_weight = 1, seed = 69)
modelfit(model_3, train, X_train.columns, useTrainCV = True)
```

```
[0]     train-error:0.309246+0.00302314 test-error:0.341351+0.0142631
[1]     train-error:0.292708+0.00222886 test-error:0.33637+0.0131123
[2]     train-error:0.288077+0.00273134 test-error:0.331778+0.013126
[3]     train-error:0.283543+0.00195502 test-error:0.327186+0.0100671
[4]     train-error:0.280586+0.00189596 test-error:0.324696+0.0142833
[5]     train-error:0.278057+0.00283106 test-error:0.322205+0.0122311
[6]     train-error:0.275333+0.00235825 test-error:0.320883+0.0134514
[7]     train-error:0.273757+0.00229698 test-error:0.319092+0.0136011
[8]     train-error:0.270721+0.00241979 test-error:0.319793+0.0135794
[9]     train-error:0.269126+0.00278041 test-error:0.319326+0.0132778
[10]    train-error:0.267608+0.00315513 test-error:0.318003+0.0140194
[11]    train-error:0.26578+0.00319178  test-error:0.317769+0.0145812
[12]    train-error:0.264067+0.00220918 test-error:0.319482+0.0159085
[13]    train-error:0.261577+0.00286451 test-error:0.317147+0.0147729
[14]    train-error:0.25897+0.00252027  test-error:0.315668+0.0141513
[15]    train-error:0.257491+0.00181287 test-error:0.315201+0.0140676
[16]    train-error:0.255642+0.00257715 test-error:0.315979+0.0126304
[17]    train-error:0.253774+0.00186687 test-error:0.315279+0.0133808
[18]    train-error:0.252705+0.00183907 test-error:0.315357+0.014047
```

```
[67]    train-error:0.172075+0.00194407 test-error:0.311465+0.012059
[68]    train-error:0.170383+0.00195392 test-error:0.312788+0.0124392
[69]    train-error:0.16904+0.00221166  test-error:0.312866+0.0119389
[70]    train-error:0.167425+0.00250209 test-error:0.312944+0.0118411
[71]    train-error:0.165869+0.00253674 test-error:0.313878+0.0124513
[72]    train-error:0.164799+0.0031096  test-error:0.314345+0.0129608
[73]    train-error:0.164157+0.00346184 test-error:0.314345+0.0133951
[74]    train-error:0.162269+0.00401186 test-error:0.313099+0.0122255
[75]    train-error:0.161355+0.0043109  test-error:0.313722+0.0114529
[76]    train-error:0.160129+0.00439078 test-error:0.313644+0.0106728
[77]    train-error:0.158631+0.00406842 test-error:0.313333+0.0111183
[78]    train-error:0.157249+0.00380047 test-error:0.3131+0.0112765
[79]    train-error:0.155635+0.00364119 test-error:0.314189+0.0108496
[80]    train-error:0.154195+0.00318813 test-error:0.314578+0.0114155
[81]    train-error:0.152677+0.00314208 test-error:0.315123+0.0113525
[82]    train-error:0.151763+0.00329103 test-error:0.314968+0.0112465
[83]    train-error:0.150343+0.00359871 test-error:0.314656+0.011299
[84]    train-error:0.149623+0.00336215 test-error:0.314734+0.0112428
[85]    train-error:0.14865+0.0033986   test-error:0.314968+0.0118385
[86]    train-error:0.147541+0.0038851  test-error:0.314578+0.0119869
[87]    train-error:0.145789+0.00350429 test-error:0.31489+0.0119306
[88]    train-error:0.144564+0.00343713 test-error:0.314423+0.0128451
[89]    train-error:0.143221+0.00338309 test-error:0.3138+0.0124841
[90]    train-error:0.141917+0.00352677 test-error:0.314579+0.0134962
[91]    train-error:0.140964+0.00335707 test-error:0.314734+0.0129112
[92]    train-error:0.139057+0.00403841 test-error:0.314812+0.0132796
[93]    train-error:0.137949+0.00411283 test-error:0.315123+0.0128654
[94]    train-error:0.136372+0.00356211 test-error:0.314657+0.0139575
[95]    train-error:0.135438+0.00368865 test-error:0.315669+0.0141339
[96]    train-error:0.134213+0.00438309 test-error:0.315123+0.0135128
[97]    train-error:0.132831+0.00434987 test-error:0.315512+0.0135648
[98]    train-error:0.13145+0.0039917   test-error:0.315279+0.0135823
[99]    train-error:0.129835+0.00356583 test-error:0.315357+0.0127973
[100]   train-error:0.128298+0.00333591 test-error:0.314423+0.0128104
[101]   train-error:0.127422+0.00354557 test-error:0.314579+0.0129548
[102]   train-error:0.12569+0.00323086  test-error:0.314579+0.0127081
[103]   train-error:0.124893+0.00328372 test-error:0.315279+0.0143903
[104]   train-error:0.123434+0.0036919  test-error:0.314345+0.0124415
[105]   train-error:0.122286+0.00340742 test-error:0.315201+0.0125614
[106]   train-error:0.121352+0.00324969 test-error:0.314267+0.0123532
[107]   train-error:0.120457+0.00325277 test-error:0.314579+0.0126786
[108]   train-error:0.119017+0.00328681 test-error:0.314267+0.0133274
[109]   train-error:0.1182+0.00330641   test-error:0.313956+0.0141952
[110]   train-error:0.116779+0.00269758 test-error:0.314501+0.014107
[111]   train-error:0.115554+0.00274051 test-error:0.314423+0.014377
[112]   train-error:0.114542+0.00276094 test-error:0.314812+0.0137077
[113]   train-error:0.113491+0.00238824 test-error:0.315591+0.0139373
[114]   train-error:0.112635+0.00287366 test-error:0.314812+0.0146195
```

61

```
[115]   train-error:0.111585+0.002942   test-error:0.31419+0.0143762
[116]   train-error:0.110145+0.00323045 test-error:0.3131+0.01468

Model Report
Accuracy : 0.807
AUC Score (Train): 0.892198
```

We create a search space of max_depth and min_child_weight, then because we are only looking at odd numbers, create a search space to look at the optimal from the cell below, but also plus and minus 1 to account for the even numbers.

[68]:
```
# Cross-Validating MAX_DEPTH and MIN_CHILD_WEIGHT

param_test1 = {
 'max_depth':range(3,8,2),
 'min_child_weight':range(1,6,2)
}
gsearch1 = GridSearchCV(estimator = XGBClassifier(learning_rate =0.1,␣
 ↪n_estimators=116, gamma=0,
                                                   subsample=0.8,␣
 ↪colsample_bytree=0.8,
                                                   objective= 'binary:logistic',␣
 ↪nthread=4,
                                                   scale_pos_weight=1, seed=27),
                        param_grid = param_test1,
                        scoring = 'accuracy', n_jobs = -1, cv=5, verbose = 3)
gsearch1.fit(train[predictors],train[target])
print('Best Approx. Parameters: \n', gsearch1.best_params_, gsearch1.
 ↪best_score_)
```

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed: 15.3min
[Parallel(n_jobs=-1)]: Done  45 out of  45 | elapsed: 34.3min finished

Best Approx. Parameters:
 {'max_depth': 3, 'min_child_weight': 5} 0.6747605163631627
```

[69]:
```
# Reducing Search Space
param_test2 = {
 'max_depth':[2,3,4],
 'min_child_weight':[4,5,6]
}
gsearch2 = GridSearchCV(estimator = XGBClassifier(learning_rate=0.1,␣
 ↪n_estimators=106, gamma=0, subsample=0.8,
                                                   colsample_bytree=0.8,␣
 ↪objective= 'binary:logistic',
```

```
                                                    nthread=4,␣
↪scale_pos_weight=1,seed=27),
                        param_grid = param_test2, scoring='accuracy',␣
↪n_jobs=-1, iid=False, cv=5, verbose = 3)
gsearch2.fit(train[predictors],train[target])
print('Best Parameters: \n', gsearch2.best_params_, gsearch2.best_score_)
```

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed: 10.3min
[Parallel(n_jobs=-1)]: Done  45 out of  45 | elapsed: 20.2min finished
/Users/bennettcohen/opt/anaconda3/lib/python3.7/site-
packages/sklearn/model_selection/_search.py:823: FutureWarning: The parameter
'iid' is deprecated in 0.22 and will be removed in 0.24.
  "removed in 0.24.", FutureWarning
```

```
Best Parameters:
 {'max_depth': 2, 'min_child_weight': 4} 0.6797414852029511
```

We then tune *gamma* in the same way and then calculate the new optimal number of parameters for the new set of hyperparameters. We then tune *subsample*, *colsample_bytree*, and *reg_alpha* in the same way as the rest with the new optimal estimators.

[70]:
```
# Cross-Validating GAMMA
param_test3 = {
    'gamma':[i/10.0 for i in range(0,4)]
}
gsearch3 = GridSearchCV(estimator = XGBClassifier(learning_rate =0.1,␣
↪n_estimators=106, max_depth=2,
                                                    min_child_weight=4,␣
↪subsample=0.8, colsample_bytree=0.8,
                                                    objective= 'binary:logistic',␣
↪nthread=4,
                                                    scale_pos_weight=1,seed=27),
                        param_grid = param_test3,␣
↪scoring='accuracy',n_jobs=-1,iid=False, cv=5, verbose = 3)
gsearch3.fit(train[predictors],train[target])
gsearch3.best_params_, gsearch3.best_score_
```

```
Fitting 5 folds for each of 4 candidates, totalling 20 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  20 out of  20 | elapsed:  7.6min remaining:    0.0s
[Parallel(n_jobs=-1)]: Done  20 out of  20 | elapsed:  7.6min finished
/Users/bennettcohen/opt/anaconda3/lib/python3.7/site-
packages/sklearn/model_selection/_search.py:823: FutureWarning: The parameter
'iid' is deprecated in 0.22 and will be removed in 0.24.
  "removed in 0.24.", FutureWarning
```

[70]: ({'gamma': 0.0}, 0.6797414852029511)

[71]:
```
# Determining NEW Optimal Number of Estimators
xgb2 = XGBClassifier(learning_rate =0.1, n_estimators=1000, max_depth=2,␣
 ↪min_child_weight=4,
                     gamma=0.0, subsample=0.8, colsample_bytree=0.8, objective=␣
 ↪'binary:logistic',
                     nthread=4, scale_pos_weight=1,seed=27)
modelfit(xgb2, train, predictors, useTrainCV = True)
```

```
[0]     train-error:0.341525+0.00283615 test-error:0.34672+0.0128637
[1]     train-error:0.338995+0.00409822 test-error:0.342751+0.0107766
[2]     train-error:0.335007+0.00282147 test-error:0.341118+0.0134404
[3]     train-error:0.332399+0.00303904 test-error:0.338627+0.0133228
[4]     train-error:0.32808+0.00465129  test-error:0.33419+0.0108018
[5]     train-error:0.327302+0.00341702 test-error:0.334736+0.0125145
[6]     train-error:0.326076+0.0034436  test-error:0.33388+0.0135716
[7]     train-error:0.325103+0.00359584 test-error:0.332478+0.0106554
[8]     train-error:0.323722+0.0026166  test-error:0.331077+0.0111449
[9]     train-error:0.322243+0.00377748 test-error:0.330766+0.0115357
[10]    train-error:0.321815+0.00328032 test-error:0.329832+0.0105127
[11]    train-error:0.320881+0.002277   test-error:0.32921+0.0108061
[12]    train-error:0.320803+0.00272104 test-error:0.330222+0.0105617
[13]    train-error:0.320511+0.00272567 test-error:0.32742+0.0124625
[14]    train-error:0.319402+0.00274731 test-error:0.326019+0.0139266
[15]    train-error:0.31843+0.00281763  test-error:0.326175+0.0134086
[16]    train-error:0.318099+0.00252308 test-error:0.326642+0.0131819
[17]    train-error:0.317651+0.00245588 test-error:0.325397+0.0140718
[18]    train-error:0.316951+0.00300396 test-error:0.324774+0.0131944
[19]    train-error:0.315725+0.00290292 test-error:0.324229+0.0132418
[20]    train-error:0.315102+0.00235106 test-error:0.323062+0.0115174
[21]    train-error:0.313916+0.00248042 test-error:0.322284+0.0126481
[22]    train-error:0.314052+0.00304143 test-error:0.321817+0.0120403
[23]    train-error:0.313312+0.00305319 test-error:0.321816+0.0105551
[24]    train-error:0.312281+0.00330422 test-error:0.322594+0.0107472
[25]    train-error:0.311522+0.0027968  test-error:0.320727+0.011593
[26]    train-error:0.311347+0.00229116 test-error:0.319793+0.0123588
[27]    train-error:0.310511+0.00256729 test-error:0.319948+0.0114676
[28]    train-error:0.309596+0.00277727 test-error:0.320415+0.011543
[29]    train-error:0.309032+0.00250885 test-error:0.320182+0.0125164
[30]    train-error:0.308779+0.00263207 test-error:0.31917+0.012491
[31]    train-error:0.30765+0.00329369  test-error:0.318859+0.0112952
[32]    train-error:0.30767+0.00229664  test-error:0.319015+0.0115892
[33]    train-error:0.306755+0.00194511 test-error:0.318703+0.011122
[34]    train-error:0.306833+0.00177436 test-error:0.317536+0.0110944
[35]    train-error:0.305705+0.00183526 test-error:0.317536+0.010416
[36]    train-error:0.305958+0.00186279 test-error:0.316835+0.0110952
```

```
[85]    train-error:0.289342+0.002165   test-error:0.312322+0.0124527
[86]    train-error:0.288894+0.00214957 test-error:0.312788+0.0127438
[87]    train-error:0.288369+0.0022613  test-error:0.312711+0.0125997
[88]    train-error:0.287882+0.00225907 test-error:0.312088+0.0128061
[89]    train-error:0.288408+0.00200088 test-error:0.312555+0.0128675
[90]    train-error:0.287766+0.00199406 test-error:0.313177+0.0114584
[91]    train-error:0.287532+0.00205299 test-error:0.311932+0.0122164
[92]    train-error:0.287357+0.00214805 test-error:0.31201+0.0115973
[93]    train-error:0.28691+0.00185414  test-error:0.312321+0.0114809
[94]    train-error:0.286579+0.00183222 test-error:0.312477+0.0112991
[95]    train-error:0.286462+0.0014833  test-error:0.312243+0.0111415
[96]    train-error:0.286073+0.0019406  test-error:0.312866+0.0105701
[97]    train-error:0.285606+0.0018514  test-error:0.313022+0.0117396
[98]    train-error:0.285158+0.00206015 test-error:0.312866+0.0110846
[99]    train-error:0.285081+0.00211103 test-error:0.313177+0.0107907
[100]   train-error:0.284341+0.00210994 test-error:0.313177+0.0113835
[101]   train-error:0.284516+0.00227254 test-error:0.313333+0.0107124
[102]   train-error:0.283855+0.00217871 test-error:0.313566+0.0106257
[103]   train-error:0.283407+0.00245097 test-error:0.313411+0.0109609
[104]   train-error:0.283037+0.00234303 test-error:0.312555+0.0112015
[105]   train-error:0.282668+0.00254689 test-error:0.3138+0.0113626
[106]   train-error:0.282571+0.00230379 test-error:0.314345+0.0115526
[107]   train-error:0.282318+0.0019223  test-error:0.314501+0.0117069
[108]   train-error:0.282162+0.00214566 test-error:0.314579+0.0114563
[109]   train-error:0.282104+0.0019556  test-error:0.314189+0.0109166
[110]   train-error:0.281948+0.00189235 test-error:0.314656+0.0106828
[111]   train-error:0.282045+0.0018707  test-error:0.3145+0.0104106
[112]   train-error:0.281753+0.00187057 test-error:0.314578+0.0105554
[113]   train-error:0.28154+0.00141868  test-error:0.314423+0.0114509
[114]   train-error:0.281481+0.0013847  test-error:0.314345+0.0112589
[115]   train-error:0.281345+0.00146551 test-error:0.313333+0.0115085
[116]   train-error:0.281014+0.00153902 test-error:0.313411+0.0120011
[117]   train-error:0.280975+0.00155211 test-error:0.313878+0.0116616
[118]   train-error:0.280353+0.00163464 test-error:0.314267+0.0116545
[119]   train-error:0.280275+0.00203596 test-error:0.314189+0.0114459
[120]   train-error:0.279633+0.00174654 test-error:0.313878+0.0115729
[121]   train-error:0.279555+0.00164392 test-error:0.313956+0.0113615
[122]   train-error:0.279127+0.00183223 test-error:0.313567+0.0113001
[123]   train-error:0.278874+0.00194563 test-error:0.313956+0.011301
[124]   train-error:0.278854+0.00206021 test-error:0.313255+0.0111467
[125]   train-error:0.278076+0.0019797  test-error:0.313255+0.0112981
[126]   train-error:0.277765+0.0018105  test-error:0.312633+0.0105071
[127]   train-error:0.277434+0.00194278 test-error:0.312555+0.010449
[128]   train-error:0.277765+0.0014103  test-error:0.313177+0.0102596

Model Report
Accuracy : 0.7056
AUC Score (Train): 0.763459
```

[72]:
```
# Cross-Validating SUBSAMPLE and COLSAMPLE_BYTREE
param_test4 = {
 'subsample':[i/10.0 for i in range(7,9)],
 'colsample_bytree':[i/10.0 for i in range(7,9)]
}
gsearch4 = GridSearchCV(estimator = XGBClassifier(learning_rate =0.1,␣
 ↪n_estimators=128, max_depth=2,
                                                   min_child_weight=4, gamma=0.
 ↪0, objective= 'binary:logistic',
                                                   nthread=4,␣
 ↪scale_pos_weight=1,seed=27),
                        param_grid = param_test4,␣
 ↪scoring='accuracy',n_jobs=-1,iid=False, cv=5, verbose = 3)
gsearch4.fit(train[predictors],train[target])
gsearch4.best_params_, gsearch4.best_score_
```

```
Fitting 5 folds for each of 4 candidates, totalling 20 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  20 out of  20 | elapsed:  8.4min remaining:    0.0s
[Parallel(n_jobs=-1)]: Done  20 out of  20 | elapsed:  8.4min finished
/Users/bennettcohen/opt/anaconda3/lib/python3.7/site-
packages/sklearn/model_selection/_search.py:823: FutureWarning: The parameter
'iid' is deprecated in 0.22 and will be removed in 0.24.
  "removed in 0.24.", FutureWarning
```

[72]:
```
({'colsample_bytree': 0.8, 'subsample': 0.8}, 0.6777958387417776)
```

[73]:
```
#Cross-Validating REG_ALPHA
param_test6 = {
 'reg_alpha':[0.1, 1, 10,100]
}
gsearch6 = GridSearchCV(estimator = XGBClassifier(learning_rate = 0.1,␣
 ↪n_estimators = 128, max_depth = 2,
                                                   min_child_weight = 4, gamma =␣
 ↪0.0, subsample = 0.8,
                                                   colsample_bytree = 0.8,␣
 ↪objective = 'binary:logistic',
                                                   scale_pos_weight = 1, seed =␣
 ↪27),
                        param_grid = param_test6, scoring = 'accuracy', n_jobs␣
 ↪= -1, iid = False, cv = 5, verbose = 3)
gsearch6.fit(train[predictors],train[target])
gsearch6.best_params_, gsearch6.best_score_
```

```
Fitting 5 folds for each of 4 candidates, totalling 20 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
```

```
[Parallel(n_jobs=-1)]: Done  20 out of  20 | elapsed:  9.1min remaining:    0.0s
[Parallel(n_jobs=-1)]: Done  20 out of  20 | elapsed:  9.1min finished
/Users/bennettcohen/opt/anaconda3/lib/python3.7/site-
packages/sklearn/model_selection/_search.py:823: FutureWarning: The parameter
'iid' is deprecated in 0.22 and will be removed in 0.24.
  "removed in 0.24.", FutureWarning
```

[73]:
```
({'reg_alpha': 0.1}, 0.6769396258593557)
```

We then calculate the optimal number of trees for our final set of hyperparameters, but decrease the learning rate to 0.01 to get our final hyperparameters and run this model.

[75]:
```
# Determining Optimal Number of Trees w/ New (LOWER) Learning Rate

xgb3 = XGBClassifier(learning_rate = 0.005, n_estimators = 5000, max_depth = 2,␣
 ↪min_child_weight = 4, gamma = 0.0,
                     subsample = 0.8, colsample_bytree = 0.8, reg_alpha = 0.1,␣
 ↪objective = 'binary:logistic',
                     nthread=4, scale_pos_weight = 1, seed = 69)
modelfit(xgb3, train, X_train.columns, useTrainCV = True, cv_folds = 5,␣
 ↪early_stopping_rounds = 100)
```

```
[0]     train-error:0.342439+0.00683494 test-error:0.347265+0.012247
[1]     train-error:0.341116+0.00781443 test-error:0.346254+0.0130183
[2]     train-error:0.340377+0.00541607 test-error:0.343997+0.0139761
[3]     train-error:0.341097+0.00429897 test-error:0.345242+0.0131253
[4]     train-error:0.340591+0.00414606 test-error:0.344541+0.0138959
[5]     train-error:0.33917+0.00187307  test-error:0.345164+0.0137555
[6]     train-error:0.33884+0.00279843  test-error:0.344697+0.0130332
[7]     train-error:0.337244+0.00398054 test-error:0.344074+0.0133438
[8]     train-error:0.338003+0.0031799  test-error:0.344541+0.0133094
[9]     train-error:0.338159+0.00332641 test-error:0.344463+0.0133088
[10]    train-error:0.337108+0.00420864 test-error:0.344308+0.0133692
[11]    train-error:0.336836+0.0041686  test-error:0.344541+0.0132745
[12]    train-error:0.337166+0.00392312 test-error:0.345164+0.0128678
[13]    train-error:0.337225+0.0042229  test-error:0.344074+0.0132625
[14]    train-error:0.336427+0.0037595  test-error:0.343607+0.0136187
[15]    train-error:0.336349+0.00373618 test-error:0.343452+0.0140151
[16]    train-error:0.335824+0.00352966 test-error:0.34353+0.0141755
[17]    train-error:0.335921+0.00358052 test-error:0.345164+0.0151571
[18]    train-error:0.336077+0.00292335 test-error:0.345476+0.0149327
[19]    train-error:0.335474+0.00349787 test-error:0.345787+0.0145228
[20]    train-error:0.33631+0.00340686  test-error:0.344697+0.0148593
[21]    train-error:0.335882+0.00335133 test-error:0.343997+0.0150647
[22]    train-error:0.336096+0.00328381 test-error:0.344541+0.0149992
[23]    train-error:0.335941+0.00331348 test-error:0.344386+0.0147544
[24]    train-error:0.335591+0.00372033 test-error:0.345087+0.0149736
[25]    train-error:0.335766+0.00333158 test-error:0.345009+0.0153295
```

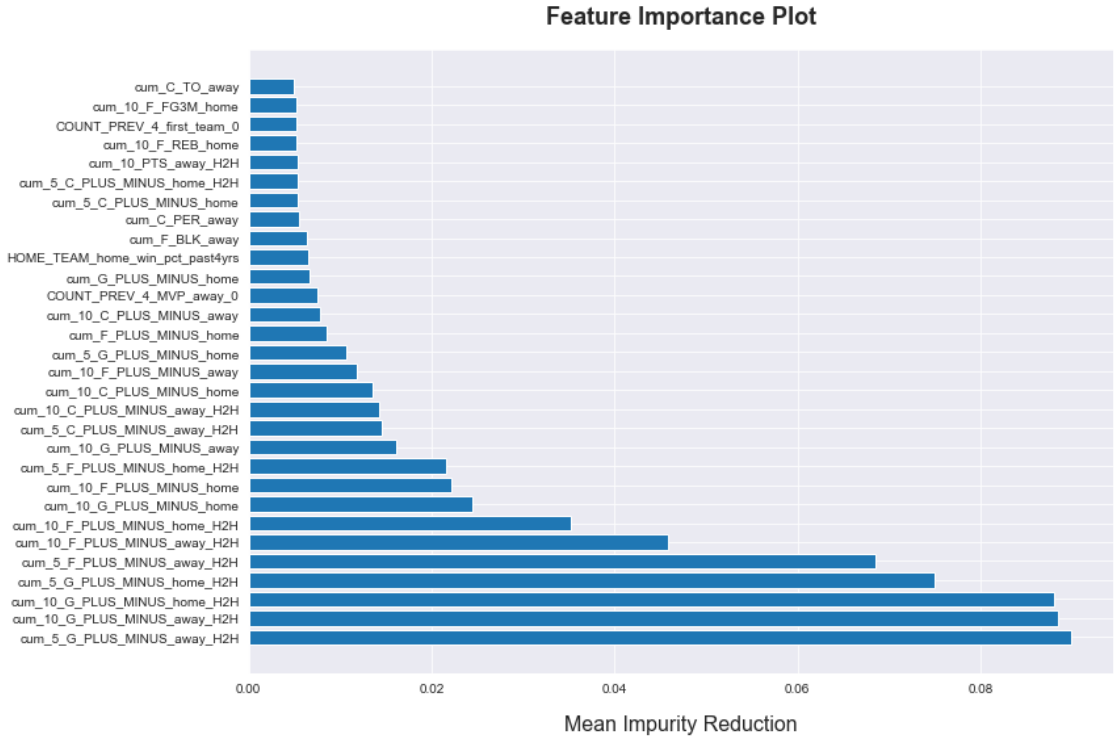
```
[1130]  train-error:0.297397+0.00238715 test-error:0.313878+0.012152
[1131]  train-error:0.297416+0.00243878 test-error:0.313878+0.012152
[1132]  train-error:0.297397+0.00239209 test-error:0.313722+0.0121643
[1133]  train-error:0.297377+0.00241122 test-error:0.313644+0.0122429
[1134]  train-error:0.297397+0.00242912 test-error:0.313489+0.0123652
[1135]  train-error:0.297358+0.00252492 test-error:0.313411+0.0124459
[1136]  train-error:0.297241+0.0025541  test-error:0.313333+0.012528
[1137]  train-error:0.297241+0.00246437 test-error:0.313489+0.0123652
[1138]  train-error:0.297202+0.00244397 test-error:0.313567+0.0123229
[1139]  train-error:0.297299+0.00247323 test-error:0.313644+0.0122626
[1140]  train-error:0.297241+0.00240224 test-error:0.313489+0.0120796
[1141]  train-error:0.297241+0.00250053 test-error:0.313489+0.0122042
[1142]  train-error:0.297183+0.00257095 test-error:0.313567+0.0121439

Model Report
Accuracy : 0.6995
AUC Score (Train): 0.752191
```

We use our optimal hyperparameters in our final model below.

[107]:
```
# Running our Final XgBoost Model
xgb5 = XGBClassifier(learning_rate = 0.005, n_estimators = 1142, max_depth = 2,␣
 ↪min_child_weight = 4, gamma = 0.0,
                     subsample = 0.8, colsample_bytree = 0.8, reg_alpha = 100,␣
 ↪objective = 'binary:logistic',
                     nthread = 4, scale_pos_weight = 1, seed = 69)

model_3 = xgb5
X_test[cat_cols] = X_test[cat_cols].astype(int).iloc[0,4]
model_3.fit(X_train, y_train)
y_prob_3 = model_3.predict_proba(X_test)
y_pred_3 = pd.Series([1 if x > 0.5 else 0 for x in y_prob_3[:,1]], index =␣
 ↪y_test.index)

# Confusion Matrix & Outputs
cm_3 = confusion_matrix(y_test, y_pred_3)
outputs_3 = outputs(cm_3)
print ("\nConfusion Matrix : \n", cm_3)
print ("\nAccuracy : ", outputs_3[0])
```

```
Confusion Matrix :
 [[ 498  583]
 [ 324 1221]]

Accuracy :  0.6546
```

[108]:
```
# Selecting top features
out = pd.DataFrame({'Feature' : list(X_train.columns),
                    'Importance Score': model_3.feature_importances_}).
 ↪sort_values('Importance Score', ascending = False)
top_3 = out.iloc[:30,:]['Feature'].values

out_303 = out.iloc[:30,:]
sns.set_style('darkgrid')
fig, ax = plt.subplots(figsize = (12,8))
plt.barh(width = out_303['Importance Score'], y = out_303['Feature'])
plt.title('Feature Importance Plot', fontsize = 18, fontweight = 'bold', pad =␣
 ↪20)
plt.xlabel('Mean Impurity Reduction', fontsize = 16, labelpad = 15)
fig.tight_layout()
plt.savefig('fig2.pdf')
```



This cell tries to run XGBoost with fewer features to see if it performs better than using all features. On some level, this is a test of how resistant to overfitting our old model is.

[109]:
```
# Running our Final XgBoost Model with only 30 Features
xgb5 = XGBClassifier(learning_rate = 0.005, n_estimators = 1142, max_depth = 2,␣
 ↪min_child_weight = 4, gamma = 0.0,
```

```
                    subsample = 0.7, colsample_bytree = 0.7, reg_alpha = 100,␣
 ↪objective = 'binary:logistic',
                    nthread = 4, scale_pos_weight = 1, seed = 69)

model_3T = xgb5
X_test[cat_cols] = X_test[cat_cols].astype(int).iloc[0,4]
model_3T.fit(X_train[top_3], y_train)
y_prob_3T = model_3T.predict_proba(X_test[top_3])
y_pred_3T = pd.Series([1 if x > 0.5 else 0 for x in y_prob_3T[:,1]], index =␣
 ↪y_test.index)

# Confusion Matrix & Outputs
cm_3T = confusion_matrix(y_test, y_pred_3T)
outputs_3T = outputs(cm_3T)
print ("\nConfusion Matrix : \n", cm_3T)
print ("\nAccuracy : ", outputs_3T[0])
```

```
Confusion Matrix :
 [[ 491  590]
 [ 321 1224]]

Accuracy :  0.6531
```

0.6 Model #4: Neural Network

The cell below creates a neural network using Sci-Kit Learn's MLP Classifier. NOTE: We also tried to implement Keras, but it actually couldn't outperform MLPClassifier, even with Dropout regularization.

[112]:
```
# Neural Network using MLPClassifier
model_4 = MLPClassifier(max_iter=1000,
                        alpha=10,
                        activation='relu',
                        hidden_layer_sizes=(25,25),
                        solver='adam',
                        learning_rate_init=0.001,
                        random_state=69,
                        tol=0.00000000000001)
model_4.fit(X_train, y_train)

y_prob_4 = model_4.predict_proba(X_test)
y_pred_4 = pd.Series([1 if x > 0.5 else 0 for x in y_prob_4[:, 1]],
                     index=y_test.index)

# Confusion Matrix & Outputs
cm_4 = confusion_matrix(y_test, y_pred_4)
```

```
outputs_4 = outputs(cm_4)
print ("\nConfusion Matrix : \n", cm_4)
print ("\nAccuracy : ", outputs_4[0])
```

```
Confusion Matrix :
 [[ 486  595]
 [ 286 1259]]

Accuracy :  0.6645
```

0.7 Final Model Results

The cell below calculates the Accuracy (our primary performance metric), TPR, and FPR for each of the four models above.

[114]:
```
# Generating the Output Table
df = pd.DataFrame({'Baseline Model': np.round(outputs_0,4),
                   'Logistic Regression': np.round(outputs_1,4),
                   'Random Forest': np.round(outputs_2,4),
                   'XGBoost': np.round(outputs_3,4),
                   'Neural Network': np.round(outputs_4,4)}, index =␣
 ↪['Accuracy', 'TPR', 'FPR'], ).T
df
```

[114]:

	Accuracy	TPR	FPR
Baseline Model	0.5883	1.0000	1.0000
Logistic Regression	0.6580	0.8220	0.5763
Random Forest	0.6565	0.8045	0.5550
XGBoost	0.6546	0.7903	0.5393
Neural Network	0.6645	0.8149	0.5504

[ ]:
```

```

Appendix #4: Approach #2 (Hold-Out Validation Blending Modeling and Analysis

FP4_VALIDATION

May 9, 2021

[7]:
```
# Importing Libraries & Functions
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import time
import re
# import warnings
# warnings.filterwarnings('ignore')
from IPython.lib.display import Audio
framerate = 4410
# play_time_seconds = 3
# t = np.linspace(0, play_time_seconds, framerate*play_time_seconds)
# audio_data = np.sin(2*np.pi*300*t) + np.sin(2*np.pi*240*t)
# Audio(audio_data, rate=framerate, autoplay=True)

#SciKit Learn
from sklearn.model_selection import GridSearchCV, KFold, train_test_split
from sklearn.metrics import r2_score, mean_squared_error, confusion_matrix,␣
 ↪mean_absolute_error
from sklearn.metrics import accuracy_score, auc, roc_curve, roc_auc_score,␣
 ↪accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.dummy import DummyClassifier
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
import statsmodels.formula.api as smf

import xgboost as xgb
from xgboost import XGBClassifier
from sklearn.neural_network import MLPClassifier

def outputs(cm):
    acc = np.round((cm.ravel()[0]+cm.ravel()[3])/sum(cm.ravel()),4)
    tpr = np.round(cm.ravel()[3] / (cm.ravel()[3] + cm.ravel()[2]),4)
```

```
    fpr = np.round(cm.ravel()[1] / (cm.ravel()[1] + cm.ravel()[0]),4)
    outputs = [acc, tpr, fpr]
    return outputs
```

0.1 Loading Data + Train/Test Split

First, we load in the two CSVs from the FP1_CUM_STATS and FP2_XFACTOR notebooks and join them on the 'GAME_ID' as the key. We then set our training data to be all data from the 2010 Season to the 2016 Season, and our testing data to be all data from the 2017 Season to the 2018 Season. The cell below performs this task, and outputs the Counts and Percentages of Training and Testing data.

[8]:
```
# Splitting Data
FP1 = pd.read_csv('FP1.csv')
FP2 = pd.read_csv('FP2.csv')
df = pd.merge(FP1, FP2, on = 'GAME_ID').rename(columns={'SEASON_y':'SEASON'})
df = df.drop(['TIMESTAMP','HOME_TEAM_NAME', 'SEASON_x'], axis=1)

matchers = ['COUNT_PREV_4','Lebron']
cat_cols = [col for col in df.columns if any(xs in col for xs in matchers)]
for cat_col in cat_cols:
    df[cat_col] = df[cat_col].astype(str)
cat_df = pd.get_dummies(df[cat_cols], drop_first = False)
cat_df['GAME_ID'] = df['GAME_ID']
df = df.merge(cat_df, on = 'GAME_ID')

test_df = df[df['SEASON'].between(2017,2018)]
rest_df = df[df['SEASON'].between(2007,2016)]

val = rest_df.sample(frac = 0.2)
val = rest_df.sample(frac = 0.2)
train_df = rest_df[~rest_df.index.isin(val.index)]
shuffled = val.sample(frac=1)
result = np.array_split(shuffled, 2)
val_a_df, val_b_df = result[0], result[1]

train_df = train_df.drop(['SEASON', 'GAME_ID'], axis = 1)
test_df = test_df.drop(['SEASON', 'GAME_ID'], axis = 1)
val_a_df = val_a_df.drop(['SEASON', 'GAME_ID'], axis = 1)
val_b_df = val_b_df.drop(['SEASON', 'GAME_ID'], axis = 1)

X_train, y_train = train_df.drop('HOME_TEAM_WINS', axis = 1),␣
 ↪train_df['HOME_TEAM_WINS']
X_test, y_test = test_df.drop('HOME_TEAM_WINS', axis = 1),␣
 ↪test_df['HOME_TEAM_WINS']
X_val_a, y_val_a = val_a_df.drop('HOME_TEAM_WINS', axis = 1),␣
 ↪val_a_df['HOME_TEAM_WINS']
```

```
X_val_b, y_val_b = val_b_df.drop('HOME_TEAM_WINS', axis = 1),␣
 ↪val_b_df['HOME_TEAM_WINS']

X_train[cat_cols] = X_train[cat_cols].astype(int).iloc[0,4]
X_val_a[cat_cols] = X_val_a[cat_cols].astype(int).iloc[0,4]
X_val_b[cat_cols] = X_val_b[cat_cols].astype(int).iloc[0,4]
X_test[cat_cols] = X_test[cat_cols].astype(int).iloc[0,4]



train_prop = np.round(len(X_train) / (len(df)), 3)
val_a_prop = np.round(len(X_val_a) / (len(df)), 3)
val_b_prop = np.round(len(X_val_b) / (len(df)), 3)
test_prop = np.round(len(X_test) / (len(df)), 3)

print('Training Data: ' + str(len(X_train)) + ' rows -- ' + str(np.
 ↪round(train_prop*100,2)) + '%')
print('Val A: ' + str(len(X_val_a)) + ' rows -- ' + str(np.
 ↪round(val_a_prop*100,2)) + '%')
print('Val B: ' + str(len(X_val_b)) + ' rows -- ' + str(np.
 ↪round(val_b_prop*100,2)) + '%')
print('Testing Data: ' + str(len(X_test)) + ' rows -- ' + str(np.
 ↪round(test_prop*100,2)) + '%')
```

```
Training Data: 10279 rows -- 66.4%
Val A: 1285 rows -- 8.3%
Val B: 1285 rows -- 8.3%
Testing Data: 2626 rows -- 17.0%
```

0.2 Model #0: Baseline (Dummy) Model

Our dummy model predicts the most frequent label in the training set, which is HOME_TEAM_WINS = 1. This is the same as always predicting the home team to win.

[9]:
```
# Baseline Model
model_0 = DummyClassifier(strategy = "most_frequent")
model_0.fit(X_train, y_train)
y_pred_0 = model_0.predict(X_test)

# Confusion Matrix & Outputs
cm_0 = confusion_matrix(y_test, y_pred_0)
outputs_0 = outputs(cm_0)
print ("\nConfusion Matrix : \n", cm_0)
print ("\nAccuracy : ", outputs_0[0])
```

```
Confusion Matrix :
 [[   0 1081]
```

```
 [   0 1545]]

Accuracy :  0.5883
```

0.3 Model #1: Logistic Regression

The cell below uses a hold-out validation set to calculate an optimal regularizaiton penalty for logistic regression.

[10]:
```
# Logistic Regression

tic = time.time()

model_1_acc_list = []
param_list = [0.01, 0.1, 0.25, 0.5, 0.75, 1, 1.5]

for i in param_list:
    model_1 = LogisticRegression(random_state = 69, tol = 0.001, max_iter = 100,
                                 verbose = 0, n_jobs = -1, penalty = 'l2',
                                 fit_intercept = True, C = i)
    model_1.fit(X_train, y_train)

    y_prob_1 = model_1.predict_proba(X_val_a)
    y_pred_1 = pd.Series([1 if x > 0.5 else 0 for x in y_prob_1[:,1]], index =␣
 ↪y_val_a.index)

    # Confusion Matrix & Outputs
    cm_1 = confusion_matrix(y_val_a, y_pred_1)
    accuracy = outputs(cm_1)[0]

    model_1_acc_list.append(accuracy)
    print(f'Parameter Value = {str(i)} -- {time.time()-tic}')

opt_param = param_list[np.argmax(model_1_acc_list)]

print('Optimal C: ', opt_param)
print('Optimal Validation Accuracy: ', np.max(model_1_acc_list))

toc = time.time()
minutes = np.floor((toc-tic)/60)
seconds = np.round((toc-tic) - (60*minutes),3)
print(f'Elapsed Time:  {int(minutes)} min  {seconds} sec\n')
```

```
Parameter Value = 0.01 -- 2.8885838985443115
Parameter Value = 0.1 -- 4.804497003555298
Parameter Value = 0.25 -- 6.7163121700286865
Parameter Value = 0.5 -- 8.549582242965698
Parameter Value = 0.75 -- 10.422482967376709
```

```
Parameter Value = 1 -- 12.251121044158936
Parameter Value = 1.5 -- 14.078026056289673
Optimal C:  1.5
Optimal Validation Accuracy:  0.6848
Elapsed Time:  0 min  14.079 sec
```

Note how high the optimal value is above!

[11]:
```
# Logistic Regression Best Estimator

model_1 = LogisticRegression(random_state = 69, tol = 0.001, max_iter = 100,
                                verbose = 3, n_jobs = -1, penalty = 'l2',
                                fit_intercept = True, C = opt_param)
model_1.fit(X_train, y_train)

y_prob_1 = model_1.predict_proba(X_test)
y_pred_1 = pd.Series([1 if x > 0.5 else 0 for x in y_prob_1[:,1]], index =␣
 ↪y_test.index)

# Confusion Matrix & Outputs
cm_1 = confusion_matrix(y_test, y_pred_1)
outputs_1 = outputs(cm_1)
print ("\nConfusion Matrix : \n", cm_1)
print ("\nAccuracy : ", outputs_1[0])
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.


Confusion Matrix :
 [[ 475  606]
 [ 284 1261]]

Accuracy :  0.6611

[Parallel(n_jobs=-1)]: Done   1 out of   1 | elapsed:    1.7s finished
```

0.4 Model #2: Random Forest

This cell below is a Random Forest model with hold-out cross-validation on MAX_FEATURES, MIN_SAMPLES_SPLIT, and MIN_SAMPLES_LEAF.

[12]:
```
# Random Forest

tic = time.time()

model_2_acc_list = []
param_list = np.linspace(1, len(X_train.columns),12, dtype = 'int32')
param_list2 = [2,4,6,8]
param_list3 = [2,4,6,8]
```

```
big_param_list = [(x, y, z) for x in param_list for y in param_list2 for z in␣
 ↪param_list3]
for i in big_param_list:
    model_2 = RandomForestClassifier(n_estimators = 100, max_features = i[0],␣
 ↪random_state = 69, n_jobs = -1,
                                     min_samples_split = i[1], min_samples_leaf␣
 ↪= i[2], verbose = 0)
    model_2.fit(X_train, y_train)

    y_prob_2 = model_2.predict_proba(X_val_a)
    y_pred_2 = pd.Series([1 if x > 0.5 else 0 for x in y_prob_2[:,1]], index =␣
 ↪y_val_a.index)

    # Confusion Matrix & Outputs
    cm_2 = confusion_matrix(y_val_a, y_pred_2)
    accuracy = outputs(cm_2)[0]

    model_2_acc_list.append(accuracy)
    print(f'Parameter Values = {i[0], i[1], i[2]} -- {time.time()-tic}')

opt_param = big_param_list[np.argmax(model_2_acc_list)][0]
opt_param2 = big_param_list[np.argmax(model_2_acc_list)][1]
opt_param3 = big_param_list[np.argmax(model_2_acc_list)][2]


print('Optimal Max Features: ', opt_param)
print('Optimal Min Samples Split: ', opt_param2)
print('Optimal Min Sampes Leaf: ', opt_param3)
print('Optimal Validation Accuracy: ', np.max(model_2_acc_list))

toc = time.time()
minutes = np.floor((toc-tic)/60)
seconds = np.round((toc-tic) - (60*minutes),3)
print(f'Elapsed Time:  {int(minutes)} min  {seconds} sec\n')
```

```
Parameter Values = (1, 2, 2) -- 0.39290904998779297
Parameter Values = (1, 2, 4) -- 0.7889249324798584
Parameter Values = (1, 2, 6) -- 1.1659808158874512
Parameter Values = (1, 2, 8) -- 1.5340547561645508
Parameter Values = (1, 4, 2) -- 1.948197841644287
Parameter Values = (1, 4, 4) -- 2.3396878242492676
Parameter Values = (1, 4, 6) -- 2.7084858417510986
Parameter Values = (1, 4, 8) -- 3.075427770614624
Parameter Values = (1, 6, 2) -- 3.4883108139038086
Parameter Values = (1, 6, 4) -- 3.8911826610565186
Parameter Values = (1, 6, 6) -- 4.266714811325073
Parameter Values = (1, 6, 8) -- 4.628662824630737
```

```
Parameter Values = (716, 8, 2) -- 5032.657346963882
Parameter Values = (716, 8, 4) -- 5097.323113918304
Parameter Values = (716, 8, 6) -- 5156.036525964737
Parameter Values = (716, 8, 8) -- 5211.602543830872
Parameter Values = (796, 2, 2) -- 5295.2686767578125
Parameter Values = (796, 2, 4) -- 5367.187859773636
Parameter Values = (796, 2, 6) -- 5432.312650918961
Parameter Values = (796, 2, 8) -- 5493.108404874802
Parameter Values = (796, 4, 2) -- 5577.020538806915
Parameter Values = (796, 4, 4) -- 5649.252530813217
Parameter Values = (796, 4, 6) -- 5714.559775829315
Parameter Values = (796, 4, 8) -- 5775.292544841766
Parameter Values = (796, 6, 2) -- 5858.117245912552
Parameter Values = (796, 6, 4) -- 5930.100612640381
Parameter Values = (796, 6, 6) -- 5995.21789264679
Parameter Values = (796, 6, 8) -- 6055.985009908676
Parameter Values = (796, 8, 2) -- 6138.572227716446
Parameter Values = (796, 8, 4) -- 6210.665695905685
Parameter Values = (796, 8, 6) -- 6275.921432733536
Parameter Values = (796, 8, 8) -- 6336.594791889191
Parameter Values = (876, 2, 2) -- 6427.831973791122
Parameter Values = (876, 2, 4) -- 6507.707731962204
Parameter Values = (876, 2, 6) -- 6586.124364852905
Parameter Values = (876, 2, 8) -- 6652.594261884689
Parameter Values = (876, 4, 2) -- 6743.336278915405
Parameter Values = (876, 4, 4) -- 6822.268302679062
Parameter Values = (876, 4, 6) -- 6893.563651800156
Parameter Values = (876, 4, 8) -- 6959.401465654373
Parameter Values = (876, 6, 2) -- 7049.351226806641
Parameter Values = (876, 6, 4) -- 7128.859305858612
Parameter Values = (876, 6, 6) -- 7201.591723918915
Parameter Values = (876, 6, 8) -- 7271.615322828293
Parameter Values = (876, 8, 2) -- 7366.754317045212
Parameter Values = (876, 8, 4) -- 7449.037947893143
Parameter Values = (876, 8, 6) -- 7520.966126680374
Parameter Values = (876, 8, 8) -- 7586.561902046204
Optimal Max Features:  398
Optimal Min Samples Split:  2
Optimal Min Sampes Leaf:  8
Optimal Validation Accuracy:  0.6856
Elapsed Time:  126 min  26.563 sec
```

[13]:
```
# Random Forest Best Estimator

model_2 = RandomForestClassifier(max_features = opt_param, n_estimators = 500,␣
 ↪random_state = 69, n_jobs = -1,
```

```
                                    min_samples_split = opt_param2,␣
 ↪min_samples_leaf = opt_param3, verbose = 1)
model_2.fit(X_train, y_train)

y_prob_2 = model_2.predict_proba(X_test)
y_pred_2 = pd.Series([1 if x > 0.5 else 0 for x in y_prob_2[:,1]], index =␣
 ↪y_test.index)

# Confusion Matrix & Outputs
cm_2 = confusion_matrix(y_test, y_pred_2)
outputs_2 = outputs(cm_2)
print ("\nConfusion Matrix : \n", cm_2)
print ("\nAccuracy : ", outputs_2[0])
```

```
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:   11.0s
[Parallel(n_jobs=-1)]: Done 184 tasks      | elapsed:   53.6s
[Parallel(n_jobs=-1)]: Done 434 tasks      | elapsed:  2.1min


Confusion Matrix :
 [[ 430  651]
 [ 261 1284]]

Accuracy :  0.6527
[Parallel(n_jobs=-1)]: Done 500 out of 500 | elapsed:  2.4min finished
[Parallel(n_jobs=8)]: Using backend ThreadingBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 184 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 434 tasks      | elapsed:    0.1s
[Parallel(n_jobs=8)]: Done 500 out of 500 | elapsed:    0.1s finished
```

0.5 Model #3: XGBoost

This cell below is a XGBoost model with hold-out cross-validation on MIN_CHILD_WEIGHT and REG_ALPHA. We use the learning rate and n_estimator combination that minimized error from FP3.

[20]:
```
# XGBoost

tic = time.time()

model_3_acc_list = []
param_list = [2,4,6,8]
param_list2 = [0.1, 1, 10, 100]
big_list = [(x, y) for x in param_list for y in param_list2]
for i in big_list:
```

```
    model_3 = XGBClassifier(learning_rate = 0.005, n_estimators = 1142,␣
↪max_depth = 2, min_child_weight = i[0], gamma = 0.0,
                            subsample = 0.8, colsample_bytree = 0.8, reg_alpha␣
↪= i[1],
                            objective = 'binary:logistic', nthread = 4,␣
↪scale_pos_weight = 1, seed = 69, eval_metric = 'error')
    model_3.fit(X_train, y_train)

    y_prob_3 = model_3.predict_proba(X_val_a)
    y_pred_3 = pd.Series([1 if x > 0.5 else 0 for x in y_prob_3[:,1]], index =␣
↪y_val_a.index)

    # Confusion Matrix & Outputs
    cm_3 = confusion_matrix(y_val_a, y_pred_3)
    accuracy = outputs(cm_3)[0]

    model_3_acc_list.append(accuracy)
    print(f'Parameter Value = {str(i)} -- {time.time()-tic}')

opt_param = big_list[np.argmax(model_3_acc_list)][0]
opt_param2 = big_list[np.argmax(model_3_acc_list)][1]

print('Optimal Min Child Weight: ', opt_param)
print('Optimal Regularization Penalty: ', opt_param2)
print('Optimal Validation Accuracy: ', np.max(model_3_acc_list))

toc = time.time()
minutes = np.floor((toc-tic)/60)
seconds = np.round((toc-tic) - (60*minutes),3)
print(f'Elapsed Time:  {int(minutes)} min  {seconds} sec\n')
```

```
Parameter Value = (2, 0.1) -- 57.70359921455383
Parameter Value = (2, 1) -- 114.19874715805054
Parameter Value = (2, 10) -- 170.51963710784912
Parameter Value = (2, 100) -- 227.14792323112488
Parameter Value = (4, 0.1) -- 283.5574572086334
Parameter Value = (4, 1) -- 340.0915231704712
Parameter Value = (4, 10) -- 396.81933212280273
Parameter Value = (4, 100) -- 454.6802990436554
Parameter Value = (6, 0.1) -- 511.12441396713257
Parameter Value = (6, 1) -- 567.6020023822784
Parameter Value = (6, 10) -- 626.3314392566681
Parameter Value = (6, 100) -- 683.0906391143799
Parameter Value = (8, 0.1) -- 739.1422650814056
Parameter Value = (8, 1) -- 795.0162613391876
Parameter Value = (8, 10) -- 851.4480702877045
Parameter Value = (8, 100) -- 907.8744382858276
```

```
Optimal Number of Estimators:  8
Optimal Regularization Penalty:  0.1
Optimal Validation Accuracy:  0.6934
Elapsed Time:  15 min  7.875 sec
```

[25]:
```
# XGBoost Best Estimator

model_3 = XGBClassifier(learning_rate = 0.005, max_depth = 2, min_child_weight␣
 ↪= opt_param, gamma = 0.0,
                         subsample = 0.8, colsample_bytree = 0.8, reg_alpha =␣
 ↪opt_param2, n_estimators = 1142,
                         objective = 'binary:logistic', nthread = 4,␣
 ↪scale_pos_weight = 1, seed = 69, eval_metric = 'error')
model_3.fit(X_train, y_train)

y_prob_3 = model_3.predict_proba(X_test)
y_pred_3 = pd.Series([1 if x > 0.5 else 0 for x in y_prob_3[:,1]], index =␣
 ↪y_test.index)

# Confusion Matrix & Outputs
cm_3 = confusion_matrix(y_test, y_pred_3)
outputs_3 = outputs(cm_3)
print ("\nConfusion Matrix : \n", cm_3)
print ("\nAccuracy : ", outputs_3[0])
```

```
Confusion Matrix :
 [[ 485  596]
 [ 315 1230]]

Accuracy :  0.6531
```

0.6 Model #4: MLP Classifier

This cell below is an MLPClassifier model with hold-out cross-validation on hidden layer size, node count, and regulariziation parameter. Again, we are tuning this parameter because we are very skeptical as to whether the models are overfitting.

[26]:
```
# Neural Network
tic = time.time()

model_4_acc_list = []
param_list = [(25, ), (25, 25), (50, )]
param_list2 = [0.1, 1, 10, 50, 100]
big_list = [(x, y) for x in param_list for y in param_list2]
for i in big_list:
    model_4 = MLPClassifier(max_iter=500,
```

```
                            alpha=i[1],
                            activation='relu',
                            hidden_layer_sizes=i[0],
                            solver='adam',
                            learning_rate_init=0.001,
                            random_state=69,
                            tol=0.00000000000001)
    model_4.fit(X_train, y_train)

    y_prob_4 = model_4.predict_proba(X_test)
    y_pred_4 = pd.Series([1 if x > 0.5 else 0 for x in y_prob_4[:, 1]],
                         index=y_test.index)

    # Confusion Matrix & Outputs
    cm_4 = confusion_matrix(y_test, y_pred_4)
    accuracy = outputs(cm_4)[0]

    model_4_acc_list.append(accuracy)
    print(f'Parameter Values = {str(i[0]), str(i[1])} -- {time.time()-tic}')

opt_param1 = big_list[np.argmax(model_4_acc_list)][0]
opt_param2 = big_list[np.argmax(model_4_acc_list)][1]

print('Optimal Hidden Layers: ', opt_param1)
print('Optimal Alpha: ', opt_param2)
print('Optimal Validation Accuracy: ', np.max(model_4_acc_list))

toc = time.time()
minutes = np.floor((toc - tic) / 60)
seconds = np.round((toc - tic) - (60 * minutes), 3)
print(f'Elapsed Time:  {int(minutes)} min  {seconds} sec\n')
```

```
Parameter Values = ('(25,)', '0.1') -- 8.652297019958496
Parameter Values = ('(25,)', '1') -- 19.761035919189453
Parameter Values = ('(25,)', '10') -- 30.899311780929565
Parameter Values = ('(25,)', '50') -- 53.827632904052734
Parameter Values = ('(25,)', '100') -- 78.09302806854248
Parameter Values = ('(25, 25)', '0.1') -- 90.39153575897217
Parameter Values = ('(25, 25)', '1') -- 102.95964479446411
Parameter Values = ('(25, 25)', '10') -- 132.5773470401764
Parameter Values = ('(25, 25)', '50') -- 151.4269459247589
Parameter Values = ('(25, 25)', '100') -- 170.90269994735718
Parameter Values = ('(50,)', '0.1') -- 180.02133798599243
Parameter Values = ('(50,)', '1') -- 190.63888096809387
Parameter Values = ('(50,)', '10') -- 208.72383999824524
Parameter Values = ('(50,)', '50') -- 226.07587599754333
Parameter Values = ('(50,)', '100') -- 252.29841089248657
```

```
Optimal Hidden Layers:  (25, 25)
Optimal Alpha:  10
Optimal Validation Accuracy:  0.6596
Elapsed Time:  4 min  12.299 sec
```

[27]:
```
# Neural Network Best Estimator

model_4 = MLPClassifier(max_iter = 1000, alpha = opt_param2, activation =␣
↪'relu', hidden_layer_sizes = opt_param1,
                        solver = 'adam', learning_rate_init = 0.001,␣
↪random_state = 69, tol = 0.00000000000001)
model_4.fit(X_train, y_train)

y_prob_4 = model_4.predict_proba(X_test)
y_pred_4 = pd.Series([1 if x > 0.5 else 0 for x in y_prob_4[:,1]], index =␣
↪y_test.index)

# Confusion Matrix & Outputs
cm_4 = confusion_matrix(y_test, y_pred_4)
outputs_4 = outputs(cm_4)
print ("\nConfusion Matrix : \n", cm_4)
print ("\nAccuracy : ", outputs_4[0])
```

```
Confusion Matrix :
 [[ 398  683]
 [ 211 1334]]

Accuracy :  0.6596
```

0.7 Model #5: Blending

We then use our models to predict probabilities on the Validation B dataset, and build a training set from this. We also build a testing blended dataframe. We combine these predictions using Logistic Regression (analagous to using Linear Regression to blend regressors).

[28]:
```
# Creating blend_df
y_prob_1_val_b = model_1.predict_proba(X_val_b)
y_prob_2_val_b = model_2.predict_proba(X_val_b)
y_prob_3_val_b = model_3.predict_proba(X_val_b)
y_prob_4_val_b = model_4.predict_proba(X_val_b)



val_b_df_blend = pd.DataFrame({'Logistic Regression': list(y_prob_1_val_b[:,1]),
                               'Random Forest': list(y_prob_2_val_b[:,1]),
                               'XGBoost': list(y_prob_3_val_b[:,1]),
```

```
                                'MLP': list(y_prob_4_val_b[:,1]),
                               'HOME_TEAM_WINS': y_val_b})
test_df_blend = pd.DataFrame({'Logistic Regression': y_prob_1[:,1],
                              'Random Forest': y_prob_2[:,1],
                              'XGBoost': y_prob_3[:,1],
                               'MLP': y_prob_4[:,1],
                              'HOME_TEAM_WINS': y_test})
X_val_b_blend, y_val_b_blend = val_b_df_blend.drop('HOME_TEAM_WINS', axis=1),␣
 ↪val_b_df_blend['HOME_TEAM_WINS']
X_test_blend, y_test_blend = test_df_blend.drop('HOME_TEAM_WINS', axis=1),␣
 ↪test_df_blend['HOME_TEAM_WINS']
```

```
[Parallel(n_jobs=8)]: Using backend ThreadingBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 184 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 434 tasks      | elapsed:    0.1s
[Parallel(n_jobs=8)]: Done 500 out of 500 | elapsed:    0.1s finished
```

[29]:
```
# Blended Logistic Regression

model_5 = LogisticRegression(random_state = 69, tol = 0.00001, max_iter = 10000,
                             verbose = 3, n_jobs = -1, fit_intercept = True)
model_5.fit(X_val_b_blend, y_val_b_blend)
y_prob_5 = model_5.predict_proba(X_test_blend)
y_pred_5 = pd.Series([1 if x > 0.5 else 0 for x in y_prob_5[:,1]], index =␣
 ↪y_test_blend.index)

# Confusion Matrix & Outputs
cm_5 = confusion_matrix(y_test_blend, y_pred_5)
outputs_5 = outputs(cm_5)
print ("\nConfusion Matrix : \n", cm_5)
print ("\nAccuracy : ", outputs_5[0])
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.


Confusion Matrix :
 [[ 477  604]
 [ 272 1273]]

Accuracy :  0.6664

[Parallel(n_jobs=-1)]: Done   1 out of   1 | elapsed:    2.2s finished
```

[30]:
```
# Generating the Output Table
df = pd.DataFrame({'Baseline Model': np.round(outputs_0,4),
                   'Logistic Regression': np.round(outputs_1,4),
                   'Random Forest': np.round(outputs_2,4),
```

```
                  'XGBoost': np.round(outputs_3,4),
                  'Neural Network': np.round(outputs_4,4),
                  'Blended': np.round(outputs_5,4)}, index = ['Accuracy',␣
 ↪'TPR', 'FPR'], ).T
df
```

[30]:

	Accuracy	TPR	FPR
Baseline Model	0.5883	1.0000	1.0000
Logistic Regression	0.6611	0.8162	0.5606
Random Forest	0.6527	0.8311	0.6022
XGBoost	0.6531	0.7961	0.5513
Neural Network	0.6596	0.8634	0.6318
Blended	0.6664	0.8239	0.5587

[ ]: