Machine Learning Engineer Nanodegree

Capstone Project

Adam Bulow
April 13, 2019

I. Definition

Project Overview

Domain Background:

In this project, we will attempt to build a model that uses an MLB hitter's career statistics to predict whether or not he is a National Baseball Hall of Famer. The process for a player being inducted to the National Baseball Hall of Fame (the Hall) is a very complicated one, but at a high level it is as follows:

- All players who played for ten seasons or more are eligible to be inducted starting five years after their final game.
- Eligible players appear on a ballot, which is handed out to a committee composed of all Baseball Writers Association of America (BBWAA) members who have been an active member for at least ten years.
- If a player receives a vote on 75% of ballots or more, he is inducted into the Hall.

Notably, the criteria on which the committee votes is ill-defined, and highly subjective. According to the BBWAA Election Rules (https://baseballhall.org/hall-of-famers/rules/bbwaa-rules-for-election), the only voting criteria given is "Voting shall be based upon the player's record, playing ability, integrity, sportsmanship, character, and contributions to the team(s) on which the player played."

With such limited criteria, there are constant arguments and discussions amongst fans, voters and baseball men and women as to whether or not certain players belong in the Hall.

While integrity, sportsmanship and character are nearly impossible to ascertain from looking at box scores, it seems that a player's record, playing ability and contributions (which are by far the most crucial categories) can be analyzed quantitatively.

Datasets:

In order to accomplish this, we will utilize several publicly available datasets:

1: Fangraphs' career hitting data:
https://www.fangraphs.com/leaders.aspx

This dataset will contain the full career statistics of all MLB players. This will contain baseball statistics such as PA, HR, OBP, etc. These will form the majority of the features we will feed into our model.

2: Lahman's Baseball Database
http://www.seanlahman.com/baseball-archive/statistics/

This dataset contains more information about each player, such as which position they primarily played, which years they played in and how many All-Star game appearances each player had. These will also be features for our model.

3: Baseball Reference's Hall of Fame Inductees Table
https://www.baseball-reference.com/awards/hof.shtml

This is a dataset containing all the players who have been inducted into the Hall. All players in this table will receive a label of 'Inducted' or 1, while all other players will receive a label of 'Not Inducted' or 0.

4: MLB Rosetta
https://github.com/geoffharcourt/mlb_rosetta

This table is simply a mapping between player id's across these different data sources (in addition to other baseball sites). We will use this table as a reference so that we can join information all of our datasets together.

Problem Statement

Problem:

The goal of this project is to see if it is possible to build a model that will take player career statistics as inputs and predict which players are Hall of Famers. For simplicity, we will restrict our analysis for this project to hitters (that is, we will exclude players who were primarily pitchers). We can accomplish this by training a model on players who's Hall of Fame fate has already been decided, with their career statistics as the features and Hall induction status as the labels. The model will be a classification model that takes as input a player's career statistics and returns a simple Yes/No (1/0) as predicted Hall of Fame induction status. Since we will have labels for all of the players, it will be straightforward to calculate the accuracy of our model.

Solution:

The solution to the problem will be creating a classifier that takes each player's career batting statistics as features and outputs a predicted Hall of Fame status: Yes/No (1/0). To do this, we will use the datasets listed above to creating a dataset that we can use to build our model. We will first start by filtering down to only Hall eligible players who have already been determined to be in the

Hall or not be in the Hall. Then we will split this data into training, testing and cross-validation sets. We will then attempt to train several different types of classification models on this data, tuning hyperparameters as necessary, to achieve a model that has a high accuracy as well as F1-score.

Project Design:

In order to complete this project we will first start by constructing a dataset which we will use to train, validate and test our classification model. To do this, we will take the Fangraphs career hitting database, which will provide the players and their career hitting statistics, which will be the main features for the model. We will then join in information from the Lahman database to get each player's years they played in as well as number of All Star appearances, which will also be features we can use for our model. We will then filter to only Hall eligible players who played their last game well enough into the past such that their Hall of Fame fate has already been decided. Finally, we will use the Baseball Reference Hall of Fame Inductees table to create labels for the players in our dataset, with all players in this table receiving a label of 1, and all other players receiving a label of 0.

Once we have this dataset, we can conduct some exploratory analysis of this dataset, including finding out what percentage of these players are in the Hall of Fame. Since we are employing a simple benchmark model of classifying each player as negative, this number will give us 1 - accuracy for our benchmark model. We can also try to identify some trends in the data, looking for things such as "All players with over 3000 hits are in the Hall of Fame" or "Every player in the Hall of Fame have at least 100 HR".

After this exploratory analysis, we can begin to construct our model. We will first split our data into a training set, a cross-validation set, and a testing set. We will then try using several different classification algorithms, tuning hyperparameters, and try to come up with the best model we can that maximizes accuracy. The algorithms we will consider are 1) Decision Trees/Random Forests, 2) Logistic Regression, 3) Support Vector Machines, and 4) Neural Networks. Once we have constructed models using each of these algorithms, we will weigh accuracy, F1-score, training time, and interpretability when selecting which model is the best for solving this problem. We will then conduct and in-depth analysis of our selected model, detailing its performance, evaluating its strengths and weaknesses, and attempting to reason about why it gets predictions for certain players right and others wrong if there are any identifiable patterns.

Metrics

Two evaluation metrics we can use to evaluate our model are accuracy and F1-score. This will allow us to compare our classification results to our simple benchmark model. Ultimately, what we want to know is how often our model makes the correct Hall of Fame induction prediction, and accuracy will let us know that. The F1-score will also be useful to balance the Precision and Recall of our model, and will distinguish our model as hopefully being far better than our simple benchmark model.

II. Analysis

Data Exploration

In [29]:
```
features = [x for x in fg.columns if x not in ['Name', 'Team', 'bis_id', 'lahm
an_id', 'baseball_reference_id', 'playerID_x', 'playerID_y', 'HOF']]
print("Features:")
print(features)
print()
print("Number of features: " + str(len(features)))
```

```
Features:
['G', 'PA', 'HR', 'R', 'RBI', 'SB', 'ISO', 'AVG', 'OBP', 'SLG', 'wOBA', 'wRC
+', 'BsR', 'Off', 'Def', 'WAR', 'AB', 'H', '1B', '2B', '3B', 'BB', 'HBP', 'wR
AA', 'wRC', 'debut_year', 'final_year', 'seasons', 'all_star_apps']

Number of features: 29
```

These are baseball statistics. For more information, go to the glossary at https://library.fangraphs.com/fangraphs-library-glossary/ (https://library.fangraphs.com/fangraphs-library-glossary/)

To do some initial data exploration, we can focus on a few important features: Games (G), Hits (H) and Home Runs (HR).

In [30]:
```
some_features = ['G', 'H', 'HR']
```

In [31]:
```
fg_non_hof = fg[fg['HOF'] == 0][some_features]
fg_non_hof_describe = fg_non_hof.describe()
print("Non-HOF distributions:")
fg_non_hof_describe
```

```
Non-HOF distributions:
```

Out[31]:

	G	H	HR
count	443.000000	443.000000	443.000000
mean	1599.753950	1508.480813	118.300226
std	318.586152	403.416806	100.815353
min	1200.000000	466.000000	2.000000
25%	1336.500000	1215.000000	39.000000
50%	1516.000000	1447.000000	91.000000
75%	1813.000000	1758.000000	168.000000
max	2700.000000	2715.000000	583.000000

In [32]:
```
fg_hof = fg[fg['HOF'] == 1][some_features]
fg_hof_describe = fg_hof.describe()[1:]
print("HOF distributions:")
fg_hof_describe
```

```
HOF distributions:
```

Out[32]:

	G	H	HR
mean	2069.288660	2316.391753	199.164948
std	496.269478	637.695045	171.122622
min	1215.000000	1161.000000	9.000000
25%	1673.000000	1779.000000	69.000000
50%	2120.000000	2299.000000	138.000000
75%	2422.000000	2735.000000	301.000000
max	3298.000000	4189.000000	755.000000

In [35]:
```
comparison_df = pd.merge(fg_non_hof_describe, fg_hof_describe, left_index=True
, right_index=True, suffixes=('_non_hof', '_hof'))
comparison_df = comparison_df.reindex(columns=sorted(comparison_df.columns))
print("Comparison:")
comparison_df
```

```
Comparison:
```

Out[35]:

	G_hof	G_non_hof	HR_hof	HR_non_hof	H_hof	H_non_hof
mean	2069.288660	1599.753950	199.164948	118.300226	2316.391753	1508.480813
std	496.269478	318.586152	171.122622	100.815353	637.695045	403.416806
min	1215.000000	1200.000000	9.000000	2.000000	1161.000000	466.000000
25%	1673.000000	1336.500000	69.000000	39.000000	1779.000000	1215.000000
50%	2120.000000	1516.000000	138.000000	91.000000	2299.000000	1447.000000
75%	2422.000000	1813.000000	301.000000	168.000000	2735.000000	1758.000000
max	3298.000000	2700.000000	755.000000	583.000000	4189.000000	2715.000000

Medians:

Non-HOF:
G: 1516
HR: 91
H: 1447

HOF:
G: 2120
HR: 128
H: 2299

This shows that these three metrics likely have a strong positive relationship with HOF status

We also see from the table that every HOF player has at least 1215 games, 9 home runs and 1161 hits.

In [36]:
```
def make_hof_pct_table(df1):
    result = df1.groupby('HOF').size().to_frame('count').reset_index()
    result['pct'] = round(100*(result['count'] / len(df1)), 1)
    return result
```

In [37]:
```
games_club = fg[fg['G'] >= 2800][['G', 'HOF']]
games_pct = make_hof_pct_table(games_club)
games_pct
```

Out[37]:

	HOF	count	pct
0	1.0	9	100.0

100% of players (9/9) with 2800 games are in the HOF

In [38]:
```
five_hundred_club = fg[fg['HR'] >= 500][['HR', 'HOF']]
five_hundred_pct = make_hof_pct_table(five_hundred_club)
five_hundred_pct
```

Out[38]:

	HOF	count	pct
0	0.0	1	10.0
1	1.0	9	90.0

90% of players (9/10) with 500 HR are in the HOF

In [39]:
```
threek_club = fg[fg['H'] >= 3000][['H', 'HOF']]
threek_pct = make_hof_pct_table(threek_club)
threek_pct
```

Out[39]:

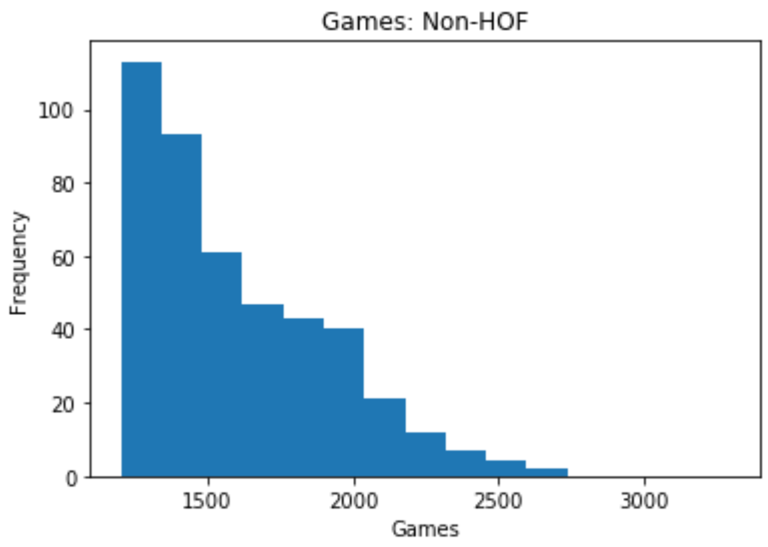
	HOF	count	pct
0	1.0	15	100.0

100% of players (15/15) with 3000 hits are in the HOF

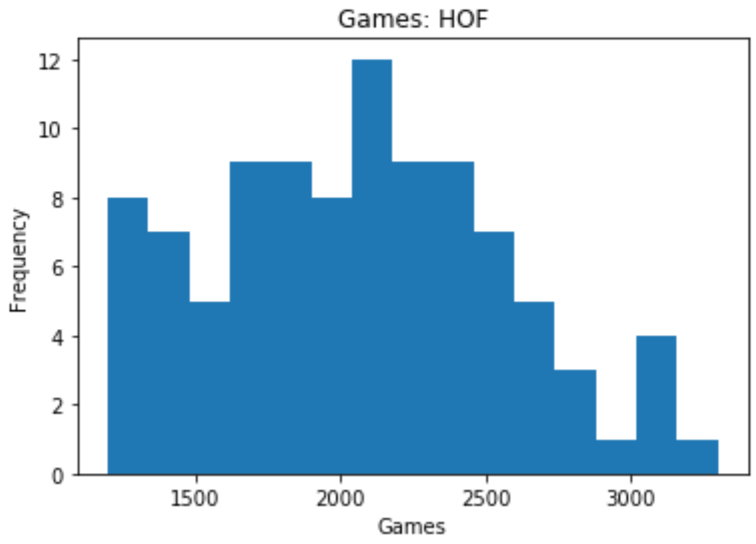
Exploratory Visualization

To visualize how the distributions of the features differ for HOF players and Non-HOF players, we can plot their histograms

In [40]:
```
plt.title('Games: Non-HOF')
plt.hist(fg_non_hof['G'], bins=15, range=(fg['G'].min(), fg['G'].max()))
plt.ylabel('Frequency')
plt.xlabel('Games')
plt.show()
```

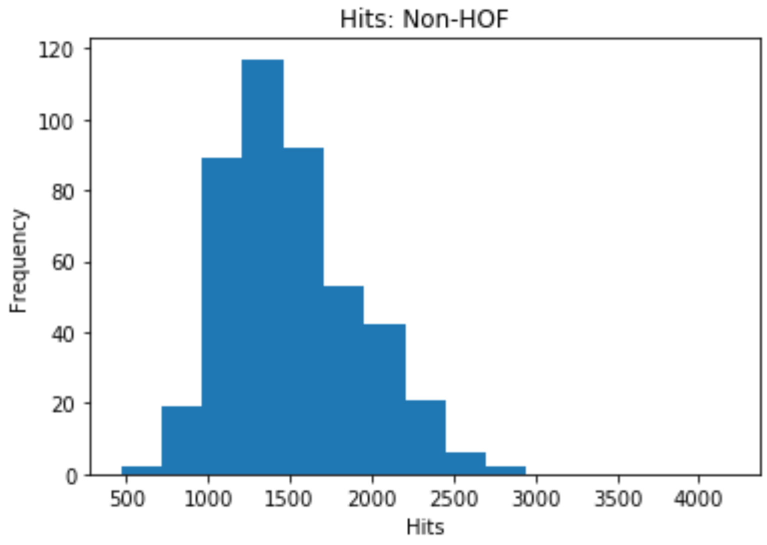


In [41]:
```
plt.title('Games: HOF')
plt.hist(fg_hof['G'], bins=15, range=(fg['G'].min(), fg['G'].max()))
plt.ylabel('Frequency')
plt.xlabel('Games')
plt.show()
```

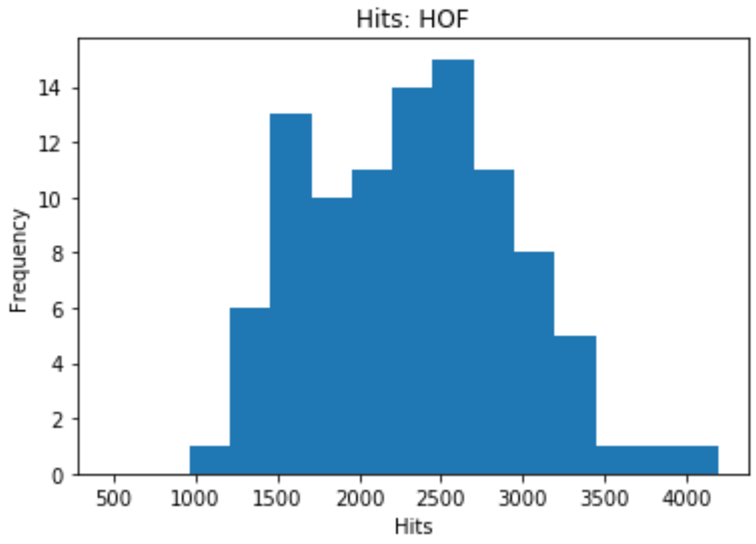


We see that the HOF distribution for Games is more symmetric and has a much higher center.

In [42]:
```
plt.title('Hits: Non-HOF')
plt.hist(fg_non_hof['H'], bins=15, range=(fg['H'].min(), fg['H'].max()))
plt.ylabel('Frequency')
plt.xlabel('Hits')
plt.show()
```

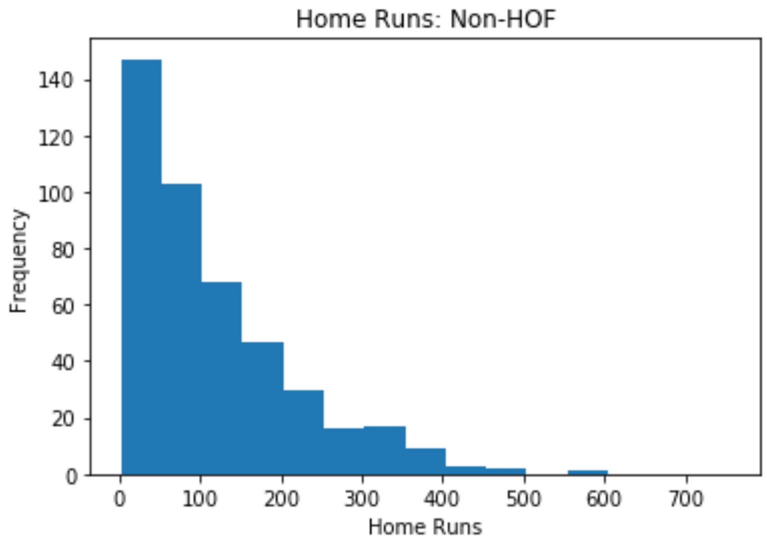


In [43]:
```
plt.title('Hits: HOF')
plt.hist(fg_hof['H'], bins=15, range=(fg['H'].min(), fg['H'].max()))
plt.ylabel('Frequency')
plt.xlabel('Hits')
plt.show()
```

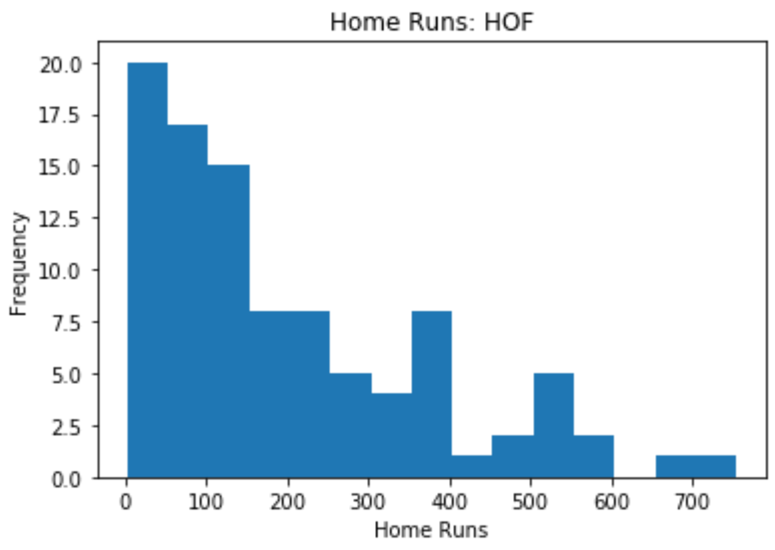


We again see that the HOF distribution for Hits is more symmetric and has a much higher center.

In [44]:
```
plt.title('Home Runs: Non-HOF')
plt.hist(fg_non_hof['HR'], bins=15, range=(fg['HR'].min(), fg['HR'].max()))
plt.ylabel('Frequency')
plt.xlabel('Home Runs')
plt.show()
```



```
In [45]: plt.title('Home Runs: HOF')
         plt.hist(fg_hof['HR'], bins=15, range=(fg['HR'].min(), fg['HR'].max()))
         plt.ylabel('Frequency')
         plt.xlabel('Home Runs')
         plt.show()
```



Both distributions for Home Runs are very right-skewed, however the HOF distribution has many more high valued observations.

It is clear from these graphs that these features have a positive relationship with HOF status.

Algorithms and Techniques

The algorithms we will use for this project are 1) Random Forests, 2) Logistic Regression, 3) Support Vector Machines, and 4) Neural Networks (MLP). All of these algorithms are suitable for classification problems such as the one we are trying to solve. While these algorithms all have their strengths and weaknesses, we will attempt to solve the problem using each of them and compare their performances to see which is best for this case.

Random Forests is an ensemble method of another type of model called Decision Trees. Decision Trees is a model that works by taking features of the data and creating a tree, which maps out explicit rules for what prediction to make given each observation. The model works by identifying which features of the data gives us the most information towards making a prediction, splitting the data on those features first. The one weakness of Decision Trees is that they tend to "overfit" (especially when the data has a lot of features), which means that they make predictions based too exactly on the specific data we use to train the model, and may not generalize well. To solve this problem, we can use Random Forests. Random Forests essentially creates multiple Decision Trees that each use only a subset of the features in the dataset, and combines them to create a single, robust prediction. When making a prediction using Random Forests, we simply let each of the Decision Trees make a prediction on each observation, and choose the prediction that occurs in the majority of the Decision Trees.

Logistic Regression classifies points by taking the output of a linear function and applying the sigmoid function to it, making the final output a predicted probability p_hat in between 0 and 1. We train the model by using a cost function that applies a large negative penalty to positive points that have a low p_hat, and a large negative penalty to negative points that have a high p_hat.

Support Vector Machines is a classification algorithm that finds the hyperplane with the maximum margin that distinctly classifies the data points. We do this by constructing a line with a margin, and penalizing points that are in the margin or are misclassified by their distance from the margin in the misclassified region. We call this classification error. We also apply a margin error by giving a large penalty if the margin is small and a small penalty if the margin is large. The total error function is C * Classification Error + Margin Error, where C is a parameter that can be tuned. A large C focuses more on correctly classifying points and a small C focuses more on having a large margin.

Multilayer Perceptrons are another algorithm that can be used for classification. They work by having an input layer, hidden layer(s), and an output layer. Each layer contains a set of nodes where every node in a layer is connected to every node in the next layer. The input layer contains one node for each feature we use to train the model, and the output layer in a binary classification problem such as ours will be a single node, which ultimately be a probability. If this number is greater than 0.5 we classify the point as positive, if it is less than 0.5 we classify the point as negative. The model works by first assigning random weights and biases to each set of connections between the nodes. For each observation, we feed in the values for each feature into the input layer, and after being passed through the entire architecture, a final output is produced. By applying a sigmoid activation function to the output layer, the output becomes a probability as desired. The model is trained by passing observations into the model and getting the final output, comparing it to the true value, and then using the error to slightly adjust the weights and biases in the model to improve them (this process is called backpropogation). This process is repeated many times, ultimately resulting in a model whose weights and biases should lead to good predictions.

Benchmark

As a benchmark model, we can compare our results to a simple/naive model that classifies every player as not a hall of famer (0).

In [46]:
```
# simple model / baseline:
simple = make_hof_pct_table(fg)
simple
```

Out[46]:

	HOF	count	pct
0	0.0	443	82.0
1	1.0	97	18.0

Since the bar for induction is very high, and only roughly 15% of eligible players are ultimately inducted, this model has an accuracy of 82%. However, this model would have a Precision of 100%, a Recall of 0%, and thus a an F1-score of 0, since it simply classifies all players as negative.

III. Methodology

Data Preprocessing:

First, we will clean up our data and join together the tables to create a finished table with all the features we need to build our models.

In [1]:

```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

In [2]:

```
# Start with the fangraphs table
fg = pd.read_csv('fangraphs/fg_career_data.csv')
fg.head()
```

Out[2]:

	Name	Team	G	PA	HR	R	RBI	SB	ISO	BABIP	...	HBP	SF	SH	GDP	B
0	Eduardo Rodriguez	Brewers	30	1	0	1	0.0	0.0	2.0	1.0	...	0.0	0.0	0.0	0.0	
1	Scott Munninghoff	Phillies	4	1	0	1	0.0	0.0	2.0	1.0	...	0.0	0.0	0.0	0.0	
2	Eric Cammack	Mets	8	1	0	0	1.0	0.0	2.0	1.0	...	0.0	0.0	0.0	0.0	
3	Frank O'Connor	Phillies	3	2	1	1	3.0	0.0	1.5	1.0	...	0.0	NaN	NaN	NaN	
4	Hub Knolls	Superbas	2	2	0	0	0.0	0.0	1.0	NaN	...	0.0	NaN	1.0	NaN	N

5 rows × 37 columns

In [3]:
```python
# Filter to just batters who played at least 1200 games
fg = fg[fg['G'] >= 1200]
fg.head()
```

Out[3]:

	Name	Team	G	PA	HR	R	RBI	SB	ISO	BABIP	...	HBP	SF	
249	Babe Ruth	- - -	2503	10616	714	2174	2217.0	123.0	0.348	0.340	...	43.0	NaN	1
265	Ted Williams	Red Sox	2292	9791	521	1798	1839.0	24.0	0.289	0.328	...	39.0	20.0	
310	Lou Gehrig	Yankees	2164	9660	493	1888	1995.0	102.0	0.292	0.332	...	45.0	NaN	1(
354	Jimmie Foxx	- - -	2317	9670	534	1751	1922.0	87.0	0.284	0.336	...	13.0	NaN	7
357	Rogers Hornsby	- - -	2259	9475	301	1579	1584.0	135.0	0.218	0.365	...	48.0	NaN	2

5 rows × 37 columns

In [4]:
```python
# id mapping table
id_map = pd.read_csv('mlb_rosetta-master/mlb_rosetta.csv', dtype={'retrosheet_
id': object, 'lahman_id': object, 'baseball_reference_id': object})
id_map.head()
```

Out[4]:

	id	first	last	current	bis_id	bis_milb_id	retrosheet_id	stats_inc_id	base
0	110001	Hank	Aaron	NaN	1000001.0	NaN	aaroh101	NaN	
1	110002	Tommie	Aaron	NaN	1000002.0	NaN	aarot101	NaN	
2	110003	Don	Aase	NaN	1000003.0	NaN	aased001	NaN	
3	110004	John	Abadie	NaN	1000004.0	NaN	abadj101	NaN	
4	110005	Ed	Abbaticchio	NaN	1000005.0	NaN	abbae101	NaN	

In [5]:
```python
# Just keep lahman_id and bis_id (fangraphs id)
id_map = id_map[['bis_id', 'lahman_id', 'baseball_reference_id']]
# Remove rows where there is no lahman_id
id_map = id_map.dropna(subset=['lahman_id'])
# Remove rows where there is no baseball_reference_id
id_map = id_map.dropna(subset=['baseball_reference_id'])
id_map.head()
```

Out[5]:

	bis_id	lahman_id	baseball_reference_id
0	1000001.0	aaronha01	aaronha01
1	1000002.0	aaronto01	aaronto01
2	1000003.0	aasedo01	aasedo01
3	1000004.0	abadijo01	abadijo01
4	1000005.0	abbated01	abbated01

In [6]:
```python
# Rename fangraphs playerid column to be bis_id
fg = fg.rename(columns={'playerid': 'bis_id'})
fg.head()
```

Out[6]:

	Name	Team	G	PA	HR	R	RBI	SB	ISO	BABIP	...	HBP	SF	
249	Babe Ruth	- - -	2503	10616	714	2174	2217.0	123.0	0.348	0.340	...	43.0	NaN	1
265	Ted Williams	Red Sox	2292	9791	521	1798	1839.0	24.0	0.289	0.328	...	39.0	20.0	
310	Lou Gehrig	Yankees	2164	9660	493	1888	1995.0	102.0	0.292	0.332	...	45.0	NaN	1
354	Jimmie Foxx	- - -	2317	9670	534	1751	1922.0	87.0	0.284	0.336	...	13.0	NaN	7
357	Rogers Hornsby	- - -	2259	9475	301	1579	1584.0	135.0	0.218	0.365	...	48.0	NaN	2

5 rows × 37 columns

In [7]:
```python
# Join in lahman_id into fangraphs df
fg = fg.merge(id_map, how='inner', left_on='bis_id', right_on='bis_id')
fg.head()
```

Out[7]:

	Name	Team	G	PA	HR	R	RBI	SB	ISO	BABIP	...	SH	GDP	E
0	Lou Gehrig	Yankees	2164	9660	493	1888	1995.0	102.0	0.292	0.332	...	106.0	2.0	
1	Jimmie Foxx	- - -	2317	9670	534	1751	1922.0	87.0	0.284	0.336	...	71.0	69.0	
2	Rogers Hornsby	- - -	2259	9475	301	1579	1584.0	135.0	0.218	0.365	...	216.0	3.0	
3	Hank Greenberg	- - -	1394	6096	331	1051	1276.0	58.0	0.292	0.323	...	35.0	66.0	
4	Ty Cobb	- - -	3035	13072	117	2246	1937.0	892.0	0.146	0.378	...	295.0	NaN	

5 rows × 39 columns

In [8]:
```python
# People table
people = pd.read_csv('lahman/People.csv')
people.head()
```

Out[8]:

	playerID	birthYear	birthMonth	birthDay	birthCountry	birthState	birthCity	deathYear	deat
0	aardsda01	1981.0	12.0	27.0	USA	CO	Denver	NaN	
1	aaronha01	1934.0	2.0	5.0	USA	AL	Mobile	NaN	
2	aaronto01	1939.0	8.0	5.0	USA	AL	Mobile	1984.0	
3	aasedo01	1954.0	9.0	8.0	USA	CA	Orange	NaN	
4	abadan01	1972.0	8.0	25.0	USA	FL	Palm Beach	NaN	

5 rows × 24 columns

In [9]:
```python
# Just keep debut and finalGame
people = people[['playerID', 'debut', 'finalGame']]
people.head()
```

Out[9]:

	playerID	debut	finalGame
0	aardsda01	2004-04-06	2015-08-23
1	aaronha01	1954-04-13	1976-10-03
2	aaronto01	1962-04-10	1971-09-26
3	aasedo01	1977-07-26	1990-10-03
4	abadan01	2001-09-10	2006-04-13

In [10]:
```python
# Get debut year, final year and number of seasons
people['debut_year'] = pd.DatetimeIndex(people['debut']).year
people['final_year'] = pd.DatetimeIndex(people['finalGame']).year
people['seasons'] = people['final_year'] - people['debut_year'] + 1
people = people[['playerID', 'debut_year', 'final_year', 'seasons']]
people.head()
```

Out[10]:

	playerID	debut_year	final_year	seasons
0	aardsda01	2004.0	2015.0	12.0
1	aaronha01	1954.0	1976.0	23.0
2	aaronto01	1962.0	1971.0	10.0
3	aasedo01	1977.0	1990.0	14.0
4	abadan01	2001.0	2006.0	6.0

In [11]:
```python
# filter to final_year <= 2006 and seasons >= 10 to get only a list of HOF eli
gible players who are confident have already had their HOF fate determined
people = people[(people['final_year'] <= 2006) & (people['seasons'] >= 10)]
people.head()
```

Out[11]:

	playerID	debut_year	final_year	seasons
1	aaronha01	1954.0	1976.0	23.0
2	aaronto01	1962.0	1971.0	10.0
3	aasedo01	1977.0	1990.0	14.0
7	abbated01	1897.0	1910.0	14.0
12	abbotgl01	1973.0	1984.0	12.0

In [12]:
```
# Join debut year and final year into fangraphs df
fg = fg.merge(people, how='inner', left_on='lahman_id', right_on='playerID')
fg.head()
```

Out[12]:

	Name	Team	G	PA	HR	R	RBI	SB	ISO	BABIP	...	wRC	WPA	R
0	Lou Gehrig	Yankees	2164	9660	493	1888	1995.0	102.0	0.292	0.332	...	2265	NaN	
1	Jimmie Foxx	- - -	2317	9670	534	1751	1922.0	87.0	0.284	0.336	...	2136	NaN	
2	Rogers Hornsby	- - -	2259	9475	301	1579	1584.0	135.0	0.218	0.365	...	2018	NaN	
3	Hank Greenberg	- - -	1394	6096	331	1051	1276.0	58.0	0.292	0.323	...	1287	NaN	
4	Ty Cobb	- - -	3035	13072	117	2246	1937.0	892.0	0.146	0.378	...	2534	NaN	

5 rows × 43 columns

In [13]:
```
# Allstar table
allstar = pd.read_csv('lahman/AllstarFull.csv')
allstar.head()
```

Out[13]:

	playerID	yearID	gameNum	gameID	teamID	lgID	GP	startingPos
0	gomezle01	1933	0	ALS193307060	NYA	AL	1.0	1.0
1	ferreri01	1933	0	ALS193307060	BOS	AL	1.0	2.0
2	gehrilo01	1933	0	ALS193307060	NYA	AL	1.0	3.0
3	gehrich01	1933	0	ALS193307060	DET	AL	1.0	4.0
4	dykesji01	1933	0	ALS193307060	CHA	AL	1.0	5.0

In [14]:
```
# Get number of allstar appearances per player
allstar = allstar[allstar['gameNum'].isin([0, 1])]
allstar = allstar.groupby('playerID').size().to_frame('all_star_apps').reset_i
ndex()
allstar.head()
```

Out[14]:

	playerID	all_star_apps
0	aaronha01	21
1	aasedo01	1
2	abreubo01	2
3	abreujo02	2
4	adamsac01	1

In [15]:

```python
# Join raw_all_star_apps into fangraphs df
fg = fg.merge(allstar, how='left', left_on='lahman_id', right_on='playerID')
fg['all_star_apps'].fillna(0, inplace=True)
fg.head()
```

Out[15]:

	Name	Team	G	PA	HR	R	RBI	SB	ISO	BABIP	...	RE24	bis_i
0	Lou Gehrig	Yankees	2164	9660	493	1888	1995.0	102.0	0.292	0.332	...	NaN	100459
1	Jimmie Foxx	- - -	2317	9670	534	1751	1922.0	87.0	0.284	0.336	...	NaN	100428
2	Rogers Hornsby	- - -	2259	9475	301	1579	1584.0	135.0	0.218	0.365	...	NaN	100603
3	Hank Greenberg	- - -	1394	6096	331	1051	1276.0	58.0	0.292	0.323	...	NaN	100499
4	Ty Cobb	- - -	3035	13072	117	2246	1937.0	892.0	0.146	0.378	...	NaN	100237

5 rows × 45 columns

In [16]:

```python
# HOF labels table
hof = pd.read_csv('baseball_ref/hof.csv')
hof.head()
```

Out[16]:

	Year	Name	Unnamed: 2	Voted By	Inducted As	Votes	% of Ballots
0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	2019.0	Harold Baines\baineha01	1959-Living	Veterans	Player	NaN	NaN
2	2019.0	Roy Halladay\hallaro01	1977-2017	BBWAA	Player	363.0	85.4%
3	2019.0	Edgar Martinez\martied01	1963-Living	BBWAA	Player	363.0	85.4%
4	2019.0	Mike Mussina\mussimi01	1968-Living	BBWAA	Player	326.0	76.7%

In [17]:

```python
# Clean HOF table
hof[['Name', 'baseball_reference_id']] = hof.Name.str.split('\\',expand=True,)
hof['HOF'] = 1
hof = hof[~hof['baseball_reference_id'].isnull()]
hof = hof[['baseball_reference_id', 'HOF']]
hof.head()
```

Out[17]:

	baseball_reference_id	HOF
1	baineha01	1
2	hallaro01	1
3	martied01	1
4	mussimi01	1
5	riverma01	1

In [18]:
```python
# Join HOF labels into fangraphs df
fg = fg.merge(hof, how='left', left_on='baseball_reference_id', right_on='base
ball_reference_id')
fg.head()
```

Out[18]:

	Name	Team	G	PA	HR	R	RBI	SB	ISO	BABIP	...	bis_id	lahm
0	Lou Gehrig	Yankees	2164	9660	493	1888	1995.0	102.0	0.292	0.332	...	1004598	geh
1	Jimmie Foxx	- - -	2317	9670	534	1751	1922.0	87.0	0.284	0.336	...	1004285	fc
2	Rogers Hornsby	- - -	2259	9475	301	1579	1584.0	135.0	0.218	0.365	...	1006030	horr
3	Hank Greenberg	- - -	1394	6096	331	1051	1276.0	58.0	0.292	0.323	...	1004996	gree
4	Ty Cobb	- - -	3035	13072	117	2246	1937.0	892.0	0.146	0.378	...	1002378	cob

5 rows × 46 columns

In [19]:
```python
fg['HOF'].fillna(0, inplace=True)
fg.head()
```

Out[19]:

	Name	Team	G	PA	HR	R	RBI	SB	ISO	BABIP	...	bis_id	lahm
0	Lou Gehrig	Yankees	2164	9660	493	1888	1995.0	102.0	0.292	0.332	...	1004598	geh
1	Jimmie Foxx	- - -	2317	9670	534	1751	1922.0	87.0	0.284	0.336	...	1004285	fc
2	Rogers Hornsby	- - -	2259	9475	301	1579	1584.0	135.0	0.218	0.365	...	1006030	horr
3	Hank Greenberg	- - -	1394	6096	331	1051	1276.0	58.0	0.292	0.323	...	1004996	gree
4	Ty Cobb	- - -	3035	13072	117	2246	1937.0	892.0	0.146	0.378	...	1002378	cob

5 rows × 46 columns

```
In [20]: # Remove features with NaN values
         fg = fg[[x for x in fg.columns if x not in [x for x in fg.columns if fg[x].isn
         ull().values.any()]]]
         fg.head()
```

Out[20]:

	Name	Team	G	PA	HR	R	RBI	SB	ISO	AVG	...	wRC	bis_id
0	Lou Gehrig	Yankees	2164	9660	493	1888	1995.0	102.0	0.292	0.340	...	2265	1004598
1	Jimmie Foxx	- - -	2317	9670	534	1751	1922.0	87.0	0.284	0.325	...	2136	1004285
2	Rogers Hornsby	- - -	2259	9475	301	1579	1584.0	135.0	0.218	0.358	...	2018	1006030
3	Hank Greenberg	- - -	1394	6096	331	1051	1276.0	58.0	0.292	0.313	...	1287	1004996
4	Ty Cobb	- - -	3035	13072	117	2246	1937.0	892.0	0.146	0.366	...	2534	1002378

5 rows × 36 columns

Implementation And Refinement

First we will scale our features using the StandardScaler. Then, we will split our data into a training set and a testing set (15%). We will make accuracy our scorer.

```
In [21]: from sklearn.metrics import make_scorer
         from sklearn.metrics import accuracy_score
         from sklearn.metrics import f1_score
         from sklearn.preprocessing import StandardScaler
         from sklearn.model_selection import train_test_split
         from sklearn.model_selection import GridSearchCV

         import warnings
         from sklearn.exceptions import DataConversionWarning
         warnings.filterwarnings("ignore", category=DeprecationWarning)
         warnings.filterwarnings("ignore", category=DataConversionWarning)

         scorer = make_scorer(accuracy_score)

         features = ['G', 'PA', 'HR', 'R', 'RBI', 'SB', 'ISO', 'AVG', 'OBP', 'SLG', 'wO
         BA', 'wRC+', 'BsR', 'Off', 'Def', 'WAR', 'AB', 'H', '1B', '2B', '3B', 'BB', 'H
         BP', 'wRAA', 'wRC', 'debut_year', 'final_year', 'seasons', 'all_star_apps']
         X = StandardScaler().fit_transform(fg[features])
         y = fg['HOF']
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, rand
         om_state=13)
```

Random Forests

We will use Random Forests with Grid Search to choose the optimal Random Forests model:

```
In [22]: from sklearn.ensemble import RandomForestClassifier

         param_grid = {'max_depth': [3, None],
                       'max_features': [1, 3, 10],
                       'min_samples_split': [2, 3, 10],
                       'bootstrap': [True, False],
                       'criterion': ['gini', 'entropy'],
                       'n_estimators': [10, 100]
                      }

         rf = RandomForestClassifier(random_state=13)
         rf_grid_obj = GridSearchCV(rf, param_grid, scoring=scorer, cv=5)
         rf_grid_fit = rf_grid_obj.fit(X_train, y_train)
         best_rf = rf_grid_fit.best_estimator_

         y_pred = best_rf.predict(X_test)
         acc = accuracy_score(y_test, y_pred)
         f1 = f1_score(y_test, y_pred)
         print("Random Forest Testing Set Accuracy: " + str(round(100*acc, 2)) + "%")
         print("Random Forest Testing Set F1-score: " + str(round(f1, 4)))
```

```
Random Forest Testing Set Accuracy: 85.19%
Random Forest Testing Set F1-score: 0.5
```

Logistic Regression

We will use Logistic Regression with L1 Regularization to choose the optimal Logistic Regression model (with built-in feature selection):

```
In [23]: from sklearn.linear_model import LogisticRegression

         lr = LogisticRegression(penalty='l1', solver='liblinear', random_state=13)
         lr.fit(X_train,y_train)

         y_pred = lr.predict(X_test)
         acc = accuracy_score(y_test, y_pred)
         f1 = f1_score(y_test, y_pred)
         print("Logistic Regression Testing Set Accuracy: " + str(round(100*acc, 2)) +
         "%")
         print("Logistic Regression Testing Set F1-score: " + str(round(f1, 4)))
```

```
Logistic Regression Testing Set Accuracy: 86.42%
Logistic Regression Testing Set F1-score: 0.5926
```

Support Vector Machines

We will use Support Vector Machines with Grid Search to choose the optimal Support Vector Machines model:

In [24]:
```
from sklearn.svm import SVC

param_grid = {'kernel':('linear', 'rbf'),
              'C':(1,0.25,0.5,0.75),
              'gamma': (1,2,3,'auto'),
              'decision_function_shape':('ovo','ovr'),
              'shrinking':(True,False)
             }

svm = SVC(random_state=13)
svm_grid_obj = GridSearchCV(svm, param_grid, scoring=scorer, cv=5)
svm_grid_fit = svm_grid_obj.fit(X_train, y_train)
best_svm = svm_grid_fit.best_estimator_

y_pred = best_svm.predict(X_test)
acc = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
print("SVM Testing Set Accuracy: " + str(round(100*acc, 2)) + "%")
print("SVM Testing Set F1-score: " + str(round(f1, 4)))
```

```
SVM Testing Set Accuracy: 86.42%
SVM Testing Set F1-score: 0.56
```

Neural Networks (MLP)

After experimenting with various architechtures, epochs and batch size, this MLP performed the best:

```
In [25]: from keras.models import Sequential
         from keras.layers.core import Dense, Dropout

         model = Sequential()
         model.add(Dense(32, input_dim=len(features), kernel_initializer='normal', acti
         vation='relu'))
         model.add(Dense(16, kernel_initializer='normal', activation='relu'))
         model.add(Dense(1, kernel_initializer='normal', activation='sigmoid'))
         model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accurac
         y'])

         model.fit(X_train, y_train, epochs=40, batch_size=10, verbose=0)

         acc = model.evaluate(X_test, y_test)[1]
         f1 = f1_score(y_test, y_pred)
         print("MLP Testing Set Accuracy: " + str(round(100*acc, 2)) + "%")
         print("MLP Testing Set F1-score: " + str(round(f1, 4)))
```

```
Using TensorFlow backend.

81/81 [==============================] - 0s 412us/ste
p□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
MLP Testing Set Accuracy: 86.42%
MLP Testing Set F1-score: 0.56
```

The process of training these models was fairly straightforward, and I did not face many challenges. The hardest part was finding a suitable Neural Network architecture, and the one I decided on seemed to perform better than architectures with more layers/nodes. It was also very important to start by scaling the features, since neglecting to do this resulted in worse accuracy across the different models.

IV. Results

Model Evaluation and Validation

Choosing and evaluating a final model

After reviewing the metrics for all the models, we will choose the Logistic Regression model we created as our final model. This model was tied for the highest accuracy at 86.42% and it had the highest F1-score of 0.5926. Additionally, this model was the fastest to train. Since this model performed well on the test set, we conclude that it generalizes well to unseen data.

Let's examine the coefficients for our logistic regression model to see which features had the highest influence:

```
In [26]: coefficients_dict = dict(zip(features, lr.coef_[0]))
         coefficients_dict
```

```
Out[26]: {'1B': 0.0,
          '2B': 0.0,
          '3B': 0.42381017592336356,
          'AB': 0.0,
          'AVG': 0.7838007926940174,
          'BB': 0.0,
          'BsR': 0.5749675116834414,
          'Def': 0.2139754992952219,
          'G': 0.0,
          'H': 0.0,
          'HBP': -0.007101296195415733,
          'HR': 0.5320802441639144,
          'ISO': 0.0,
          'OBP': 0.7047394117926162,
          'Off': 0.0,
          'PA': 0.0,
          'R': 0.0,
          'RBI': 0.33324835740618186,
          'SB': -0.20912837628006892,
          'SLG': 0.0,
          'WAR': 1.340148220583923,
          'all_star_apps': 0.8874638722733965,
          'debut_year': -1.13493930335757,
          'final_year': 0.0,
          'seasons': 0.131239657297181,
          'wOBA': 0.0,
          'wRAA': 0.0,
          'wRC': 0.0,
          'wRC+': -0.8942864667647683}
```

```
In [27]: top_5_coefficients = sorted(list(abs(lr.coef_[0])))[-5:]
         top_coefficients_dict = {key: coefficients_dict[key] for key in coefficients_d
         ict if abs(coefficients_dict[key]) in top_5_coefficients}
         top_coefficients_dict
```

```
Out[27]: {'AVG': 0.7838007926940174,
          'WAR': 1.340148220583923,
          'all_star_apps': 0.8874638722733965,
          'debut_year': -1.13493930335757,
          'wRC+': -0.8942864667647683}
```

The features with the highest positive effect on HOF status: AVG, wRC+, WAR and All-star appearances
The feature with the highest negative effect on HOF status: Debut year

This tells us that performing well in AVG, wRC+ and WAR, and making All-star appearances are key factors for being inducted into the HOF. This also shows that it was easier for players who debuted further in the past to be inducted.

Justification

Comparing the final model to benchmark

Our model performed significantly better than the benchmark model. It's accuracy was 86.42% compared to 82% for the benchmark model, and it's F1-score was 0.5926 compared to 0 for the benchmark model.

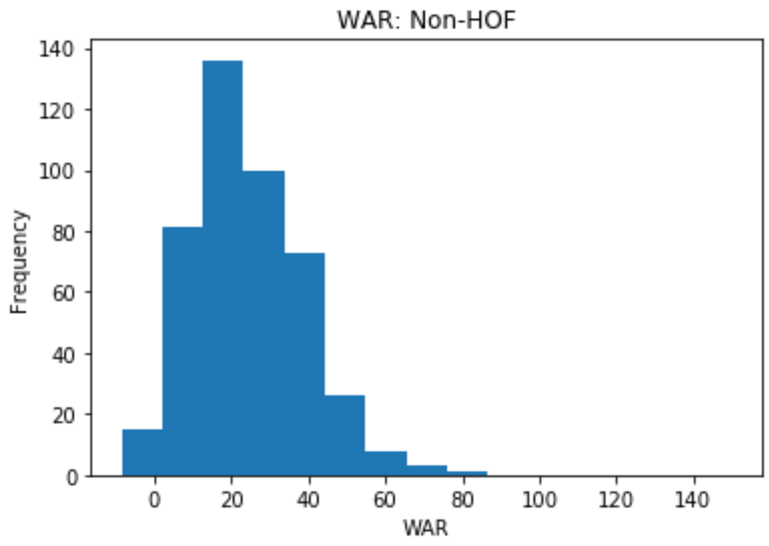
V. Conclusion

Free-Form Visualization

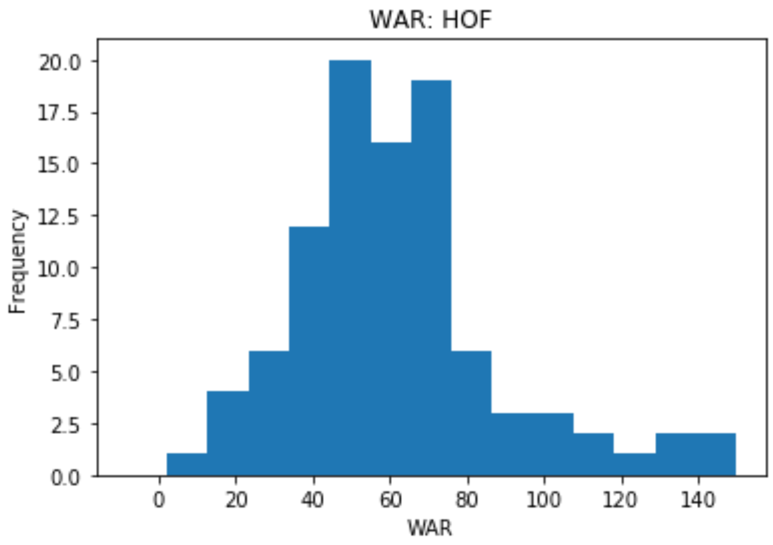
In examining the Logistic Regression coefficients, we saw that WAR was the feature with the most influence, and it had a positive relationship with HOF status. We can see this has merit by plotting the distributions of WAR for HOF players and non-HOF players:

In [56]:
```
fg_non_hof = fg[fg['HOF'] == 0][['HOF','WAR']]
fg_hof = fg[fg['HOF'] == 1][['HOF','WAR']]
```

In [57]:
```
plt.title('WAR: Non-HOF')
plt.hist(fg_non_hof['WAR'], bins=15, range=(fg['WAR'].min(), fg['WAR'].m
ax()))
plt.ylabel('Frequency')
plt.xlabel('WAR')
plt.show()
```



In [58]:
```
plt.title('WAR: HOF')
plt.hist(fg_hof['WAR'], bins=15, range=(fg['WAR'].min(), fg['WAR'].max
()))
plt.ylabel('Frequency')
plt.xlabel('WAR')
plt.show()
```



We see that the HOF distribution contains much higher values in general. It appears that almost all of the Non-HOF players have a WAR less than the mean WAR for HOF players. This makes sense given our findings.

Reflection

Summary

We started by constructing a dataset which we used to train our classification model. To do this, we took the Fangraphs career hitting database, which gave us the main features we could use to train the model. We then joined in information from the Lahman database to get each player's years they played in as well as number of All Star appearances, which we also used in training our model. We then filtered to only Hall eligible players who played their last game well enough into the past such that their Hall of Fame fate has already been decided. Then, we used the Baseball Reference Hall of Fame Inductees table to create labels for the players in our dataset, with all players in this table receiving a label of 1, and all other players receiving a label of 0.

Once we had this dataset, we conducted some exploratory analysis of this dataset, including finding out what percentage of these players are in the Hall of Fame, which was 18%. We employed a simple benchmark model of classifying each player as negative, which had an accuracy of 100% - 18% = 82%. This benchmark model has an F1-score of 0.

Then, we began to construct classification models. After splitting our data in training and testing sets, we constructed models using Random Forests, Logistic Regression, Support Vector Machines and Neural Networks (MLP). After reviewing the metrics for all the models, we chose the Logistic Regression model we created as our final model. This model was tied for the highest accuracy at 86.42% and it had the highest F1-score of 0.5926. Additionally, this model was the fastest to train. Our model performed significantly better than the benchmark model. It's accuracy was 86.42% compared to 82% for the benchmark model, and it's F1-score was 0.5926 compared to 0 for the benchmark model.

Other Reflections

The most interesting aspect of the entire process for me was the process of acquiring several raw datasets and cleaning and joining them to create a dataset that was ready for use to train machine learning models. Surprisingly, the challenge of finding the right datasets and preparing them for analysis was actually the hardest part of this project.

I believe that the model we created is suitable to solve our main problem. It has a relatively high accuracy (higher than our benchmark model), is fast to train and has interpretable coefficients. It also has a significantly higher F1-score than our benchmark model. While I was hoping to achieve an even higher accuracy of above 90%, I am still satisfied with the results.

Improvement

One possible improvement could be to experiment with additional classification algorithms. It is possible that another classifier would have performed even better than the ones we used. Another way the results could be improved would be by experimenting with more Neural Network architectures. It is possible a better architecture exists to solve this problem that I did not find in my experimentation. Potentially the biggest improvement could be made by focusing more on feature engineering. Doing more research into the problem and finding out how to construct better features for the model could help immensely. We also could have used dimensionality reduction to combine our features and reduce the number used. For all of these reasons, I think it is certainly possible that one could construct a better model than the one we constructed.