

Student name: David Schenck

Student pace: Flex

Scheduled project review date/time:

Instructor Name: Morgan Jones

Blog post URL: <https://daviddat24.wordpress.com/2023/08/01/why-the-rise-of-ai-is-concerning/>
[\(https://daviddat24.wordpress.com/2023/08/01/why-the-rise-of-ai-is-concerning/\)](https://daviddat24.wordpress.com/2023/08/01/why-the-rise-of-ai-is-concerning/)

Project Description

Every year, the NFL hosts the NFL Combine which seeks to evaluate prospective players ahead of the Draft. The players who are invited participate in several tests of their athleticism as well as completing some position specific drills. NFL teams use the Combine and other methods to evaluate players and decide who they want to draft or potentially sign after the draft is complete.

In this project, I utilize the results of the NFL combine from 2009 to 2019 to try to answer the following questions:

1. How accurately can one predict whether a player will be drafted based on their NFL Combine performance?
2. Which tests carry more weight in determining which players get drafted?
3. Does the weight given to certain tests depend on the player a position plays?

An NFL team could use the model created below to estimate how likely it is that a given player will be drafted. This could be useful because it can give the team a sense of how much they should prioritize a player in the Draft. If a player is going to be highly coveted by other teams, it will be necessary to select them with an early draft pick. If the player is unlikely to be drafted (but the team still wants them), they can use a later draft pick or even try to sign them as an undrafted player.

Exploring the Data

```
In [1]: # import pandas as pd
# import numpy as np
# import matplotlib.pyplot as plt
# %matplotlib inline
# import seaborn as sns
# import scipy.stats as st
```

The main dataset contains information about the performance of players in the NFL Combine from 2009 to 2019.

Source: [\(https://www.kaggle.com/datasets/redlineracer/nfl-combine-performance-data-2009-2019\)](https://www.kaggle.com/datasets/redlineracer/nfl-combine-performance-data-2009-2019).

Columns in the dataset: Year: Year player attended combine

Player: Name of player and a unique code used by pro-football-reference.com.

Age: Age of player when they attended the combine.

School: College the player last attended.

Height: Height of player in meters.

Weight: Weight of player in kg.

Sprint_40yd: Time in the 40-yard sprint.

Vertical_Jump: Height of vertical jump in cm.

Bench_Press_Reps: Number of times player could bench press 225 pounds consecutively.

Broad_Jump: Distance player could jump from a standing position in cm.

Agility_3cone: Time to complete 3-cone drill in seconds.

Shuttle: Time to complete the 20-yard shuttle drill in seconds.

Drafted..tm.rnd.yr.: If drafted, it gives the team, round, pick, and year.

BMI: Body mass index.

Player_Type: Offense, defense, or special teams.

Position_Type: Puts positions into categories.
Position: The specific position the player plays.
Drafted: Yes or No.

Fixes to Original Data

The original data had 550 players whose age was missing. This created a problem because only 2% of those players were drafted, but 77% of players with listed ages got drafted. This was a very strong systematic effect that biased the models.

I went through all of the players with missing ages and searched for them online to get the missing data. I still wasn't able to find the ages for 35 of the players, but that is way better than missing 550. I also found that there were other issues with the data:

1. There were 30 players who were listed as going undrafted, but were actually drafted. The vast majority of these were recent (from 2018 and 2019). I fixed these.
2. Some players had the wrong school listed. These were also fixed.

I also went through the data to make sure it wasn't missing players who were drafted after attending the combine. From 2009-2017, there were about a dozen players like this who were missing. I manually added them to the data. However, in 2018 and 2019, it was clear that whoever compiled the data made some sort of mistake which led to missing players. In 2018, all of the linebackers were missing. In 2019, linebackers, defensive ends, defensive tackles, offensive guards, and centers were missing. I added all of these manually. In all, 98 players were added to the data.

Another issue with the 2018 and 2019 data is that the records now list all safeties as just 'S' rather than separating free safeties and strong safeties. There were 56 of these players. I manually fixed these myself.

In [2]: # Read in data

```
df_combine = pd.read_csv('./Data/NFL.csv')
df_combine
```

Out[2]:

	Year	Player	Age	School	Height	Weight	Sprint_40yd	Vertical_Jump	Bench_Press_Reps
0	2009	Beanie Wells\WellCh00	20.0	Ohio St.	1.854200	106.594207	4.38	85.09	25.0
1	2009	Will Davis\DavidWi99	22.0	Illinois	1.879600	118.387609	4.84	83.82	27.0
2	2009	Herman Johnson\JohnHe23	24.0	LSU	2.006600	165.107623	5.50	NaN	21.0
3	2009	Rashad Johnson\JohnRa98	23.0	Alabama	1.803400	92.079251	4.49	93.98	15.0
4	2009	Cody Brown\BrowCo96	22.0	Connecticut	1.879600	110.676538	4.76	92.71	26.0
...
3570	2019	Jalen Jelks\JelkJa00	22.0	Oregon	1.965325	116.119647	4.92	82.55	19.0
3571	2019	Chris Slayton\SlayCh00	22.0	Syracuse	1.930400	139.252858	5.13	73.66	28.0
3572	2019	Javon Patterson\PattJa00	21.0	Mississippi	1.905000	139.252858	5.13	69.85	27.0
3573	2019	Dakota Allen\AlleDa01	23.0	Texas Tech	1.838325	105.233430	4.77	80.01	23.0
3574	2019	Jordan Brailford\BraiJo00	23.0	Oklahoma St.	1.905000	114.305277	4.65	95.25	22.0

3575 rows × 18 columns

The data below lists each player that made the Pro Bowl from 2009 to 2022. The Pro Bowl is the NFL All-Star team; the best few players at each position are chosen each year. This is a good measure of whether or not a player is succeeding as an NFL player.

The only column in this data that I actually care about is Player-additional. This is a unique code for each player that is also included in the "Player" column in the Combine data. I can use this column to count how many Pro Bowls each player in the combine data has been in.

```
In [3]: # Read in Pro Bowl data  
df_probowl = pd.read_excel('./Data/NFLProBowls.xlsx')  
df_probowl
```

Out[3]:

	Pos	Player	Conf	Tm	Age	Yrs	Player-additional	Year
0	QB	Lamar Jackson	AFC	BAL	22	1	JackLa00	2019
1	QB	Russell Wilson	NFC	SEA	31	7	WilsRu00	2019
2	QB	Aaron Rodgers	NFC	GNB	36	14	RodgAa00	2019
3	QB	Deshawn Watson	AFC	HOU	24	2	WatsDe00	2019
4	QB	Patrick Mahomes	AFC	KAN	24	2	MahoPa00	2019
...
1625	KR	DeSean Jackson	NFC	PHI	23	1	JackDe00	2009
1626	KR	Percy Harvin	NFC	MIN	21	Rook	HarvPe00	2009
1627	KR	Johnny Knox	NFC	CHI	23	Rook	KnoxJo00	2009
1628	SP	Kassim Osgood	AFC	SDG	29	6	OsgoKa00	2009
1629	SP	Heath Farwell	NFC	MIN	28	4	FarwHe20	2009

1630 rows × 8 columns

Below, I split the Player column in the Combine data into the player name and the reference code.

The Player column will now only include the player name. A new column called player_code will hold the reference code.

```
In [4]: df_combine['player_code'] = [x[1] if len(x)==2 else None for x in df_combine['Player'].str.split()]
df_combine['Player'] = [x[0] for x in df_combine['Player'].str.split(pat="\\")]
df_combine
```

Out[4]:

	Year	Player	Age	School	Height	Weight	Sprint_40yd	Vertical_Jump	Bench_Press_Reps	Broad_J
0	2009	Beanie Wells	20.0	Ohio St.	1.854200	106.594207	4.38	85.09	25.0	32
1	2009	Will Davis	22.0	Illinois	1.879600	118.387609	4.84	83.82	27.0	29
2	2009	Herman Johnson	24.0	LSU	2.006600	165.107623	5.50	NaN	21.0	
3	2009	Rashad Johnson	23.0	Alabama	1.803400	92.079251	4.49	93.98	15.0	30
4	2009	Cody Brown	22.0	Connecticut	1.879600	110.676538	4.76	92.71	26.0	30
...
3570	2019	Jalen Jelks	22.0	Oregon	1.965325	116.119647	4.92	82.55	19.0	28
3571	2019	Chris Slayton	22.0	Syracuse	1.930400	139.252858	5.13	73.66	28.0	25
3572	2019	Javon Patterson	21.0	Mississippi	1.905000	139.252858	5.13	69.85	27.0	24
3573	2019	Dakota Allen	23.0	Texas Tech	1.838325	105.233430	4.77	80.01	23.0	29
3574	2019	Jordan Brailford	23.0	Oklahoma St.	1.905000	114.305277	4.65	95.25	22.0	32

3575 rows × 19 columns



Next, I create another column that counts the number of Pro Bowls for each player.

In [5]:

```
df_combine['pro_bowls'] = df_combine['player_code'].apply(lambda x: sum(df_probowl['Player-addit:'))
df_combine['pro_bowlYN'] = df_combine['pro_bowls'].apply(lambda x: 0 if x==0 else 1)
df_combine
```

Out[5]:

	Year	Player	Age	School	Height	Weight	Sprint_40yd	Vertical_Jump	Bench_Press_Reps	Broad_J
0	2009	Beanie Wells	20.0	Ohio St.	1.854200	106.594207	4.38	85.09	25.0	32
1	2009	Will Davis	22.0	Illinois	1.879600	118.387609	4.84	83.82	27.0	29
2	2009	Herman Johnson	24.0	LSU	2.006600	165.107623	5.50	NaN	21.0	
3	2009	Rashad Johnson	23.0	Alabama	1.803400	92.079251	4.49	93.98	15.0	30
4	2009	Cody Brown	22.0	Connecticut	1.879600	110.676538	4.76	92.71	26.0	30
...
3570	2019	Jalen Jelks	22.0	Oregon	1.965325	116.119647	4.92	82.55	19.0	28
3571	2019	Chris Slayton	22.0	Syracuse	1.930400	139.252858	5.13	73.66	28.0	25
3572	2019	Javon Patterson	21.0	Mississippi	1.905000	139.252858	5.13	69.85	27.0	24
3573	2019	Dakota Allen	23.0	Texas Tech	1.838325	105.233430	4.77	80.01	23.0	29
3574	2019	Jordan Brailford	23.0	Oklahoma St.	1.905000	114.305277	4.65	95.25	22.0	32

3575 rows × 21 columns



Below, I change the Drafted column from "Yes" and "No" to 1 and 0.

In [6]:

```
df_combine['Drafted'].replace({'No':0, 'Yes':1}, inplace=True)
```

I also split up the draft info in Drafted..tm.rnd.yr. into three columns:

1. Team
2. Round
3. Pick

```
In [7]: draft_info_split = df_combine['Drafted..tm.rnd.yr.'].str.split('/')
df_combine['Team'] = [x[0] if type(x)==list else x for x in draft_info_split]
df_combine['Round'] = [int(x[1][1]) if type(x)==list else x for x in draft_info_split]
df_combine['Pick'] = [int(x[2][1:-8]) if type(x)==list else x for x in draft_info_split]
df_combine.drop(columns='Drafted..tm.rnd.yr.', inplace=True)
df_combine[['Team', 'Round', 'Pick']]
```

Out[7]:

	Team	Round	Pick
0	Arizona Cardinals	1.0	31.0
1	Arizona Cardinals	6.0	204.0
2	Arizona Cardinals	5.0	167.0
3	Arizona Cardinals	3.0	95.0
4	Arizona Cardinals	2.0	63.0
...
3570	Dallas Cowboys	7.0	241.0
3571	New York Giants	7.0	245.0
3572	Indianapolis Colts	7.0	246.0
3573	Los Angeles Rams	7.0	251.0
3574	Washington Redskins	7.0	253.0

3575 rows × 3 columns

DUPLICATES:

Below, I look to see if any players are repeated in the data.

```
In [8]: df_combine['player_code'].value_counts()
```

```
Out[8]: BullRa00    1
McBaDa99    1
WingBr00    1
FaleDa00    1
FeenTr00    1
..
FuncDe00    1
RichWe00    1
ColeKu99    1
CrosTy00    1
BrowJa02    1
Name: player_code, Length: 3193, dtype: int64
```

Each player code appears once, so there are no duplicates in the data.

Data Info and Basic Stats

Look at the data columns and basic statistics for numerical data.

In [9]: df_combine.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3575 entries, 0 to 3574
Data columns (total 23 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   Year              3575 non-null    int64  
 1   Player             3575 non-null    object  
 2   Age                3540 non-null    float64 
 3   School             3575 non-null    object  
 4   Height              3575 non-null    float64 
 5   Weight              3575 non-null    float64 
 6   Sprint_40yd        3380 non-null    float64 
 7   Vertical_Jump       2862 non-null    float64 
 8   Bench_Press_Reps   2647 non-null    float64 
 9   Broad_Jump          2832 non-null    float64 
 10  Agility_3cone      2324 non-null    float64 
 11  Shuttle             2406 non-null    float64 
 12  BMI                 3575 non-null    float64 
 13  Player_Type         3575 non-null    object  
 14  Position_Type       3575 non-null    object  
 15  Position             3575 non-null    object  
 16  Drafted             3575 non-null    int64  
 17  player_code          3193 non-null    object  
 18  pro_bowls            3575 non-null    int64  
 19  pro_bowlYN           3575 non-null    int64  
 20  Team                2395 non-null    object  
 21  Round               2395 non-null    float64 
 22  Pick                2395 non-null    float64 
dtypes: float64(12), int64(4), object(7)
memory usage: 642.5+ KB
```

There are 3575 players listed in the data.

There is some missing data.

35 players do not have an age listed.

195 players do not have a 40-yard sprint time.

713 players do not have a vertical jump recorded.

928 players do not have a number of bench press reps recorded.

743 players do not have a broad jump recorded.

1251 players do not have a time for the 3-cone drill.

1169 players do not have a time for the 20-yard shuttle.

1180 players have NaN for the Drafted..tm.rnd.yr column. This is not actually missing data. It just means they were not drafted.

382 players do not have a player code. This is not surprising because players who never made it to the NFL will not be given a player code by pro-football-reference.com. This also shows that some players (in fact, a significant number of them) made it to the NFL despite not being drafted.

In [10]: df_combine.describe()

Out[10]:

	Year	Age	Height	Weight	Sprint_40yd	Vertical_Jump	Bench_Press_Reps	Broad_Jump
count	3575.000000	3540.000000	3575.000000	3575.000000	3380.000000	2862.000000	2647.000000	2832.000000
mean	2013.960000	22.022599	1.874916	110.087566	4.769885	83.399773	20.360786	291.676667
std	3.140671	0.983071	0.067152	20.433895	0.300367	10.646721	6.485650	23.848590
min	2009.000000	18.000000	1.651000	67.585263	4.220000	44.450000	2.000000	198.120000
25%	2011.000000	21.000000	1.828800	93.213232	4.530000	76.200000	16.000000	276.860000
50%	2014.000000	22.000000	1.879600	105.233430	4.690000	83.820000	20.000000	294.640000
75%	2017.000000	23.000000	1.930400	127.005864	4.960000	90.170000	25.000000	307.340000
max	2019.000000	28.000000	2.082800	167.375585	6.000000	114.300000	49.000000	373.380000



Things I noticed from the table above:

67.0% of players who attend the combine* end up being drafted.

*There may be players who attended who are not in the data for one reason or another

The scale is quite different for different drills. For example, the slowest 40-yard sprint was 6 seconds, but the shortest broad jump was 198.12 cm. For certain types of classification techniques, rescaling will be needed.

The means and medians for the 6 drills do not differ that much. I will look at the actual distributions, but this indicates that the distributions are somewhat symmetric. This could be important if I choose to do a Naive Bayes classifier using a Gaussian probability density.

Ideally, every player in the data would have participated in all 6 drills. However, that is not the case here. Let's see what happens if I cut out all players that have at least one drill missing.

First, I create a column called Missing_Athletic_Tests that lists how many of the 6 drills have NaN.

In [11]: df_combine['Missing_Athletic_Tests'] = np.sum(df_combine[['Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Agility_3cone', 'Shuttle']].isna(), axis=1)

Next, I look at the basic stats for players who completed all 6 drills.

In [12]: df_combine[df_combine['Missing_Athletic_Tests'] == 0].describe()

Out[12]:

	Year	Age	Height	Weight	Sprint_40yd	Vertical_Jump	Bench_Press_Reps	Broad_Jun
count	1780.000000	1767.000000	1780.000000	1780.000000	1780.000000	1780.000000	1780.000000	1780.000000
mean	2013.712921	22.013016	1.877140	112.768669	4.777876	83.207831	20.585955	290.38336
std	3.111413	0.947877	0.068408	21.067575	0.315292	10.831459	6.495289	24.48131
min	2009.000000	19.000000	1.651000	75.296333	4.260000	49.530000	2.000000	208.28000
25%	2011.000000	21.000000	1.828800	93.893621	4.530000	76.200000	16.000000	274.32000
50%	2014.000000	22.000000	1.879600	109.769353	4.680000	83.820000	20.000000	294.64000
75%	2016.000000	23.000000	1.930400	135.624119	5.010000	91.440000	25.000000	307.34000
max	2019.000000	26.000000	2.057400	167.375585	6.000000	114.300000	44.000000	355.60000

In [13]: for n in range(7):
 print(f"sum(df_combine['Missing_Athletic_Tests'] == n) players missed exactly {n} athletic

```
1780 players missed exactly 0 athletic test(s).
579 players missed exactly 1 athletic test(s).
363 players missed exactly 2 athletic test(s).
171 players missed exactly 3 athletic test(s).
301 players missed exactly 4 athletic test(s).
309 players missed exactly 5 athletic test(s).
72 players missed exactly 6 athletic test(s).
```

Out of 3575 players, 1780 completed all 6 drills (this is 49.8%). While this sample is still large enough to gain some insights, it feels a bit too strict to cut that much data. There is still information to be gained from those who completed some, but not all of the drills. Also, there are players who did not complete any of the drills who might still be relevant. An example of this is Kyler Murray, currently quarterback for the Arizona Cardinals. He did not participate in the Combine, but was still drafted with the first overall pick. This shows that missing the Combine or missing some tests does not preclude a player from being drafted.

Rather than cutting players who have missing drills, it might be more appropriate to impute values. Rather than imputing values now, I think it would be more appropriate to handle the imputing on a case-by-case basis for each classifier I use. The reason for this is that some classifiers can handle missing data on their own (like XGBoost) while others cannot work if missing data is included (like K Nearest Neighbors).

When I do impute values, I don't just want to use the mean or median for that column. The data includes different football positions with different body types. The average 40-yard sprint time would probably be too slow for the average cornerback, but way too fast for the average offensive lineman. A more accurate way to impute values is to take the mean or median for each position separately.

School

The last college each player attended is listed in the School column. Let's look at whether or not players from different schools tend to get drafted.

```
In [14]: # for i,school in enumerate(df_combine['School'].value_counts().index):
#     ydrafted = sum((df_combine['School']==school) & (df_combine['Drafted']==1))
#     ndrafted = sum((df_combine['School']==school) & (df_combine['Drafted']==0))
#     print(f'{i} {ydrafted} {ndrafted} {np.round(ydrafted / (ydrafted+ndrafted),3)}')

```

0	83	23	0.783	Alabama
1	66	32	0.673	LSU
2	65	20	0.765	Ohio St.
3	57	24	0.704	USC
4	62	16	0.795	Florida
5	53	21	0.716	Florida St.
6	56	15	0.789	Georgia
7	54	16	0.771	Clemson
8	53	17	0.757	Oklahoma
9	49	17	0.742	Miami (FL)
10	44	16	0.733	Notre Dame
11	41	17	0.707	Stanford
12	39	17	0.696	Iowa
13	41	14	0.745	Penn St.
14	38	17	0.691	Wisconsin
15	33	19	0.635	Texas A&M
16	33	18	0.647	Oregon
17	35	16	0.686	North Carolina
18	30	20	0.6	Auburn
19	???	???	???	???

There are 255 unique schools listed. I do not think it is wise to actually treat each one of these as a separate category when fitting classifiers. Instead, it would make more sense to divide the schools into groups depending on how big their football program is. The easiest way to define these groups is to simply see how many players they send to the combine. The heavy-hitters in college football like Alabama, Ohio State, Georgia, and Notre Dame send lots of players. Let's see if the more well-represented schools have a higher rate of being drafted compared to the less-well represented schools.

I define three groups:

- Group 1: The top 21 most represented schools. Includes 1387 players. Each school has at least 50 players.
- Group 2: The next 30 most represented schools. Includes 1099 players. Each school has at least 25 players.
- Group 3: The rest of the schools (204 total). Includes 1089 players. Each school has at least 1 player.

```
In [62]: # ydrafted = [sum((df_combine['School']==sch) & (df_combine['Drafted']==1)) for sch in df_combine['School'].value_counts().index]
# ndrafted = [sum((df_combine['School']==sch) & (df_combine['Drafted']==0)) for sch in df_combine['School'].value_counts().index]
#
# print('Group 1:',sum(ydrafted[0:21])/(sum(ydrafted[0:21])+sum(ndrafted[0:21])))
# print('Group 2:',sum(ydrafted[21:51])/(sum(ydrafted[21:51])+sum(ndrafted[21:51])))
# print('Group 3:',sum(ydrafted[51:])/(sum(ydrafted[51:])+sum(ndrafted[51:])))
```

```
Group 1: 0.7302533532041728
Group 2: 0.6864966949952785
Group 3: 0.62
```

It does look like the bigger college football programs have a higher conversion rate for getting their players drafted. 72.0% of the players from Group 1 schools got drafted.
 66.4% of the players from Group 2 schools got drafted.
 62.0% of the players from Group 3 schools got drafted.

Below, I create a new column to indicate the group the player's school is in. These can be used as categories for a classifier.

```
In [16]: df_combine['school_group'] = [1 if df_combine['School'].value_counts()[x] > 49  
else 2 if df_combine['School'].value_counts()[x] > 24  
else 3 for x in df_combine['School']]
```

Trends in the data

Below, I look at how the different columns in the data relate to whether a player was drafted.



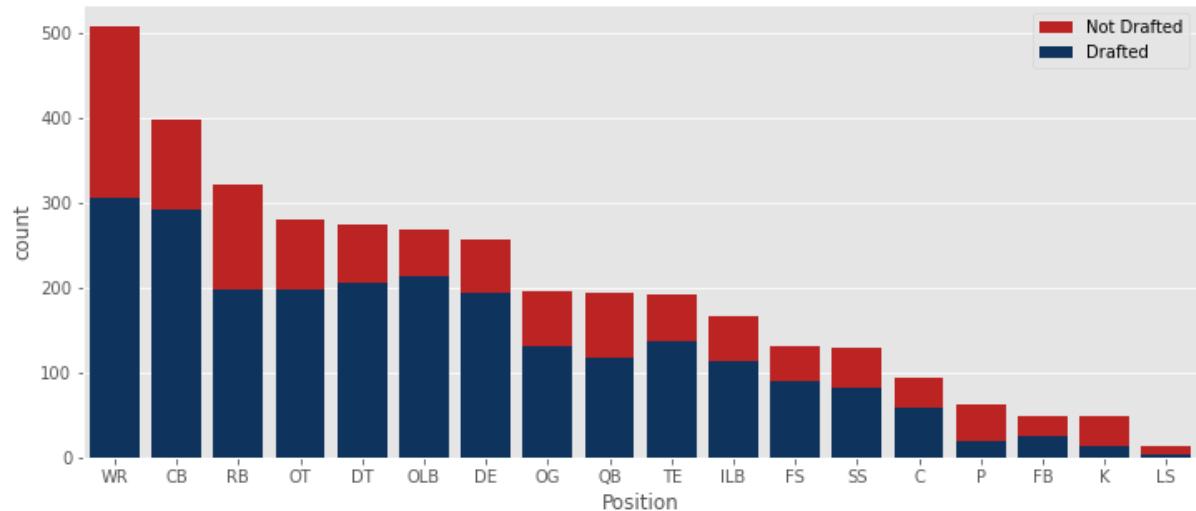
The distributions above show some very subtle differences, but it can be hard to see. Part of the reason for this might be that all the different positions are being mixed together. It might be easier to see differences if we separate the players by position.

```
In [18]: # Number of players at each position
plt.style.use('ggplot')
fig, ax = plt.subplots(figsize=(12,5))

sns.countplot(data=df_combine,
               x='Position',
               ax=ax,
               color="#D50A0A",
               label='Not Drafted',
               alpha=1.0,
               order=df_combine['Position'].value_counts().index)

sns.countplot(data=df_combine[df_combine['Drafted']==1],
               x='Position',
               ax=ax,
               color="#013369",
               label='Drafted',
               alpha=1.0,
               order=df_combine['Position'].value_counts().index)
ax.legend()
```

Out[18]: <matplotlib.legend.Legend at 0x281fbe33ee0>



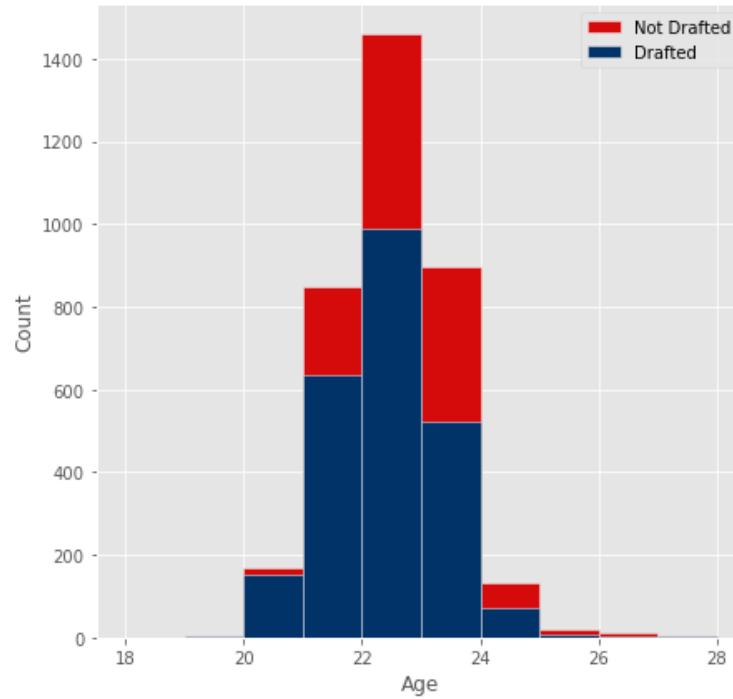
Below is a histogram of players by age. It appears that the youngest players (19 and 20 years old), get drafted at a very high rate.

```
In [19]: fig, ax = plt.subplots(figsize=(7,7))

sns.histplot(data=df_combine,
              x='Age',
              ax=ax,
              color='#D50A0A',
              binwidth=1,
              label='Not Drafted',
              alpha=1.0)

sns.histplot(data=df_combine[df_combine['Drafted']==1],
              x='Age',
              ax=ax,
              color='#013369',
              binwidth=1,
              label='Drafted',
              alpha=1.0)
ax.legend();
```

Out[19]: <matplotlib.legend.Legend at 0x281fc0510a0>



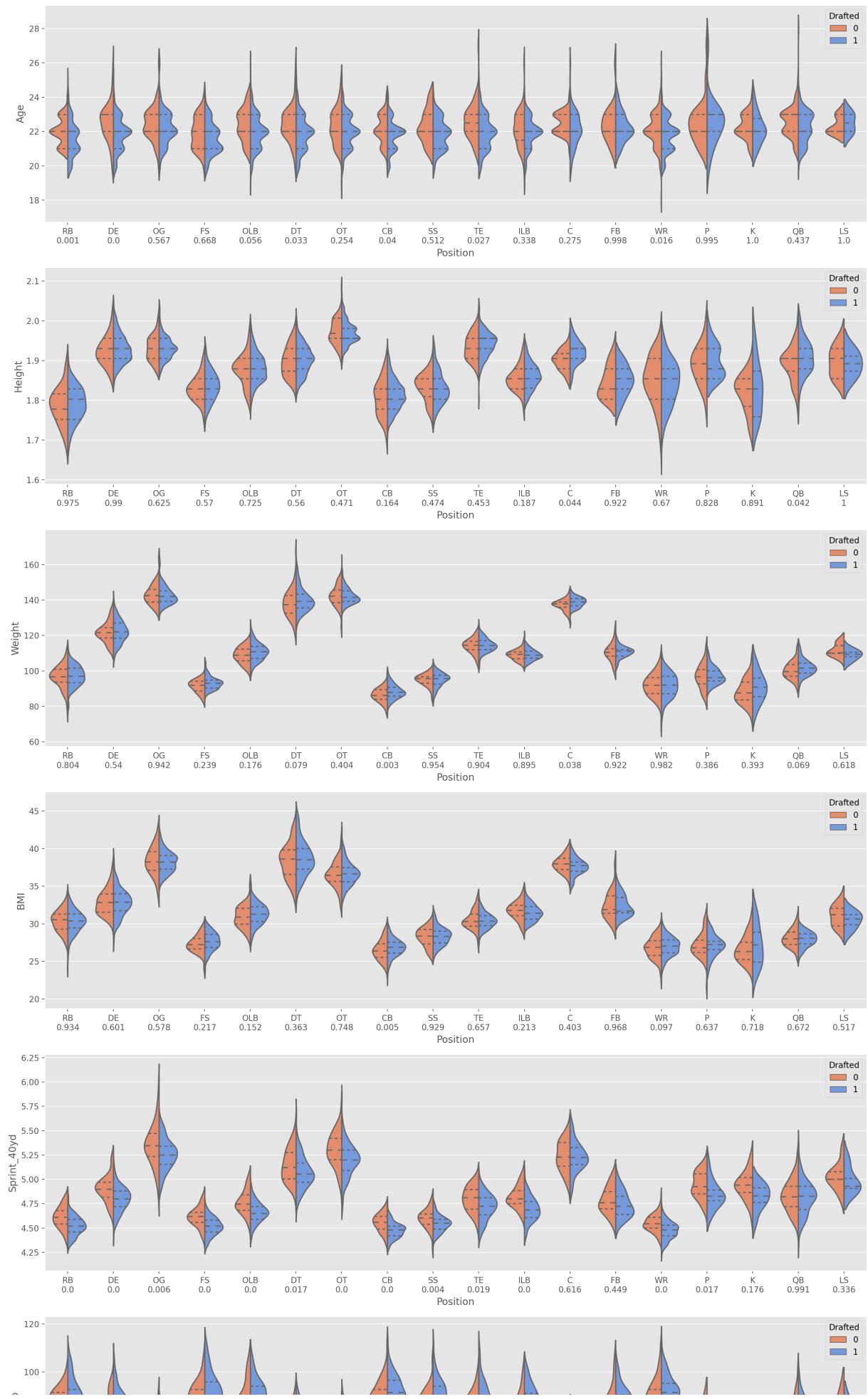
Below, I plot split violin plots for each numerical column separated by position. The violins are split so that the distribution of undrafted players is on the left and the distribution of drafted players is on the right. I also obtain the p-value for a Kolmogorov-Smirnov test to help determine which distributions are different.

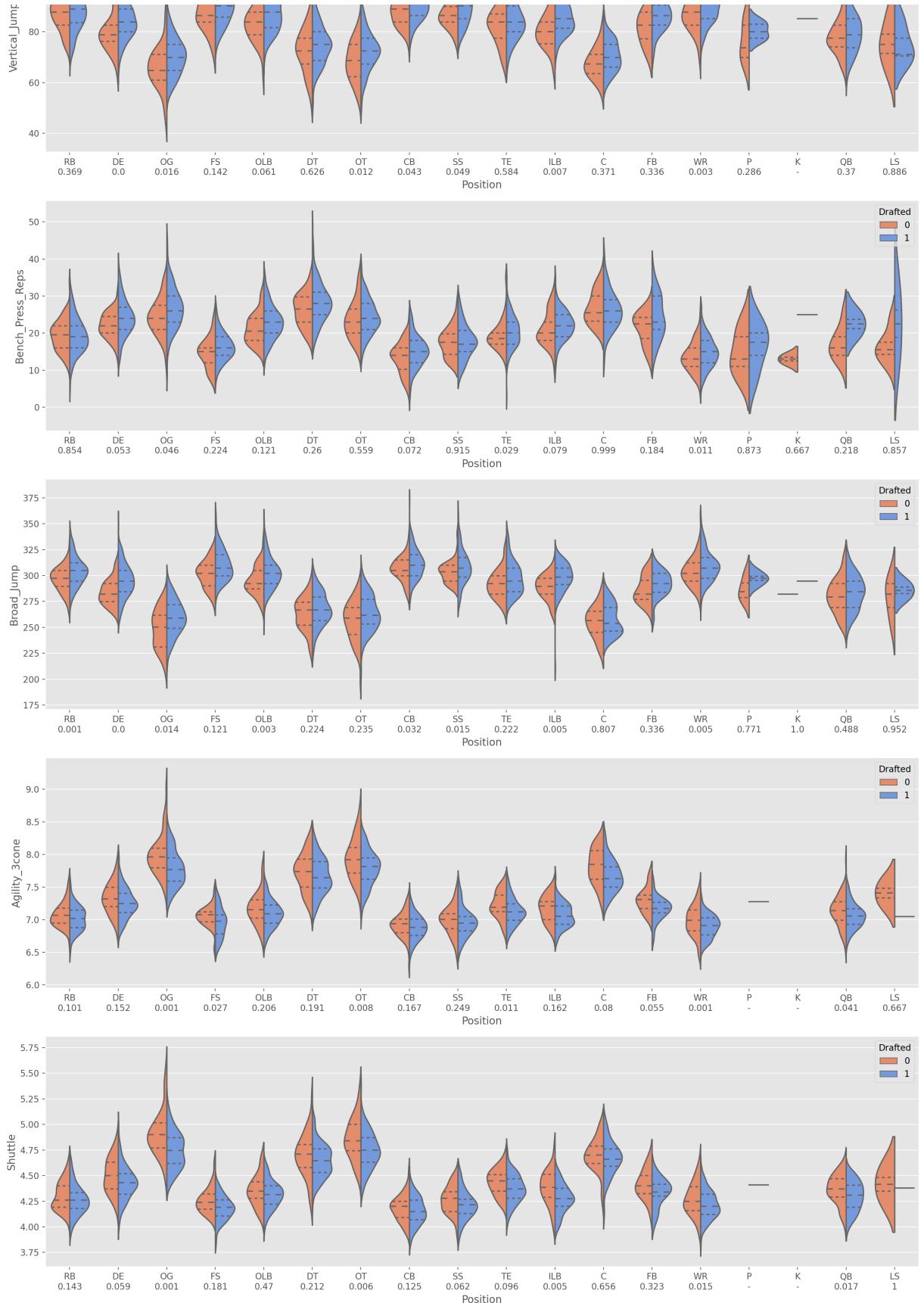
```
In [20]: # Violin plots for each athletic test and each position
fig, ax = plt.subplots(nrows=10, figsize=(15,45), dpi=200)
plt.style.use('ggplot')

cols = ['Age', 'Height', 'Weight', 'BMI', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jun']

for i, col in enumerate(cols):
    sns.violinplot(data=df_combine, x='Position', y=col, hue='Drafted', split=True, inner='quart',
                    palette=['#f88458', '#6495ed'], ax=ax[i])
    ks_results = []
    for j, pos in enumerate(df_combine['Position'].unique()):
        try:
            ks_result = st.ks_2samp(df_combine[(df_combine['Drafted']==1) &
                                                (df_combine[col].isna()==False) &
                                                (df_combine['Position']==pos)][col],
                                      df_combine[(df_combine['Drafted']==0) &
                                                (df_combine[col].isna()==False) &
                                                (df_combine['Position']==pos)][col])
            ks_results.append(ks_result[1])
        except:
            ks_results.append("-")
        ax[i].set_xticklabels([f"\n{pos}\n{ks}" if ks=="-" else f"\n{pos}\n{round(ks,3)}" for pos, ks in
                               zip(df_combine['Position'].unique(), ks_results)])
fig.tight_layout()
```







First, plotting the distributions for each position separately shows the drastic differences in performance on certain tests. The classifiers will be much more accurate if the positions are taken into account.

Second, there are visible differences between the drafted and undrafted players in many cases. For example, for positions where speed is considered important (wide receiver, running back, safety, cornerback), the drafted players ran the 40-yard sprint about a tenth of a second faster, a significant difference given the spread in the values is on the order of a tenth of a

second.

KS tests:

Height: The only positions for which the KS-test p-value was below 0.05 were center (C) and quarterback (QB). In both cases, taller players were more likely to get drafted.

Weight: Significant p-values are seen for cornerbacks (CB) and centers (C). In both cases, heavier players were more likely to get drafted.

BMI: The only position with a significant p-value is cornerbacks (CB). Larger BMIs were more likely to be drafted.

Sprint_40yd: Now we are talking. Most players do the 40-yard sprint, regardless of position. The only positions that do NOT have significant p-values from the KS-test centers (C), fullbacks (FB), kickers (K), quarterbacks (QB), and long snappers (LS). These are mostly not a surprise as most of these positions do not put great emphasis on speed (I guess the age of the running quarterback is over).

Vertical_Jump: Significant KS-test results: defensive ends (DE), offensive guards (OG), offensive tackle (OT), strong safeties (SS), inside linebackers (ILB), and wide receivers (WR). In every case, players with larger vertical jumps had a better chance of being drafted.

Bench_Press_Reps: Significant KS-test results: offensive guards (OG), tight ends (TE), and wide receivers (WR). More reps (stronger) means a better chance of being drafted.

Broad_Jump: Significant KS-test results: running backs (RB), defensive ends (DE), offensive guards (OG), outside linebackers (OLB), cornerbacks (CB), strong safeties (SS), inside linebackers (ILB), and wide receivers (WR). Larger broad jumps lead to a better chance of being drafted.

Agility_3cone: Significant KS-test results: offensive guards (OG), free safeties (FS), offensive tackles (OT), tight ends (TE), wide receivers (WR), and quarterbacks (QB). Several other positions had low, but not significant p-values. Faster times lead to higher chances of being drafted.

Shuttle: Significant KS-test results: offensive guards (OG), offensive tackles (OT), inside linebackers (ILB), wide receivers (WR), and quarterbacks (QB). Faster shuttle times lead to higher chances of being drafted.

There weren't any columns that led to a significant p-value for kickers, punters, or long snappers. This is not particularly surprising; those positions have a very specific set of skills needed that are not really tested by the drills at the combine. Fullbacks also didn't have any p-values below 0.05. This is probably just because of small sample sizes (who still uses fullbacks?).

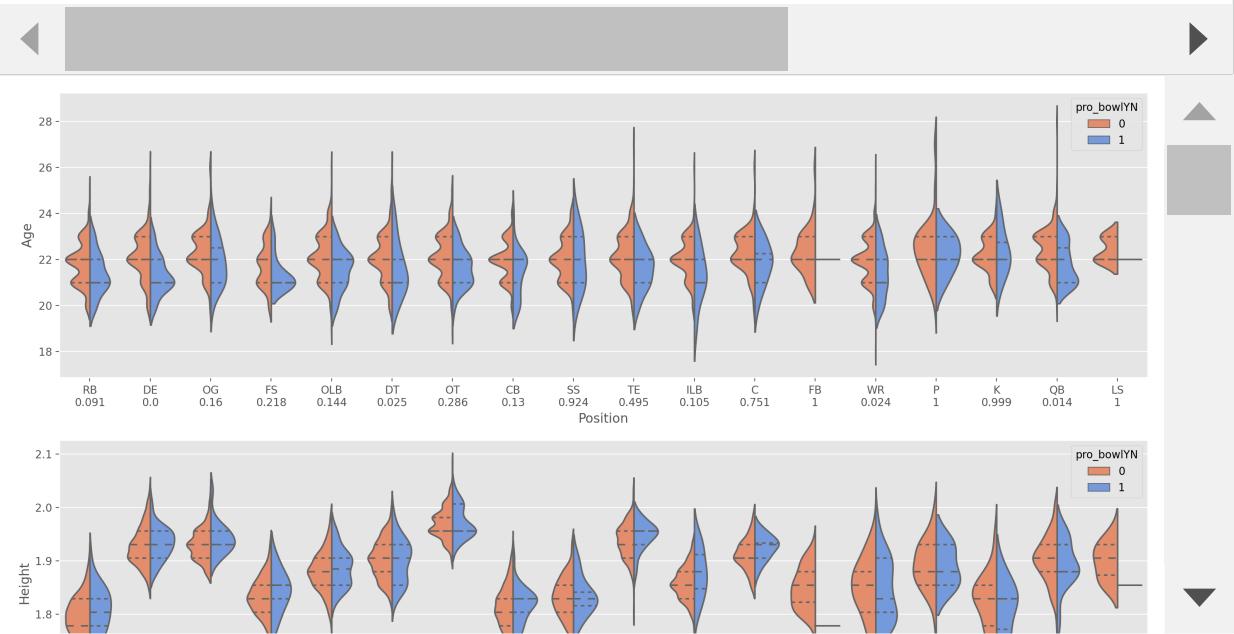
The plots and KS tests indicate that the results of the combine are going to have predictive power when trying to tell who will get drafted.

Below are more violin plots, but with the players separated by whether or not they ever made a Pro Bowl.

```
In [21]: # Violin plots for each athletic test and each position
fig, ax = plt.subplots(nrows=10, figsize=(15,45), dpi=200)
plt.style.use('ggplot')

cols = ['Age', 'Height', 'Weight', 'BMI', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Junp']

for i, col in enumerate(cols):
    sns.violinplot(data=df_combine, x='Position', y=col, hue='pro_bowlYN', split=True, inner='quart')
    palette=['#f88458', '#6495ed'], ax=ax[i])
    ks_results = []
    for j, pos in enumerate(df_combine['Position'].unique()):
        try:
            ks_result = st.ks_2samp(df_combine[(df_combine['pro_bowlYN']==1) &
                                                (df_combine[col].isna()==False) &
                                                (df_combine['Position']==pos)][col],
                                      df_combine[(df_combine['pro_bowlYN']==0) &
                                                (df_combine[col].isna()==False) &
                                                (df_combine['Position']==pos)][col])
            ks_results.append(ks_result[1])
        except:
            ks_results.append("-")
        ax[i].set_xticklabels([f"\n{pos}\n{ks}" if ks=="-" else f"\n{pos}\n{round(ks,3)}" for pos, ks in
                               zip(df_combine['Position'].unique(), ks_results)])
    fig.tight_layout()
```

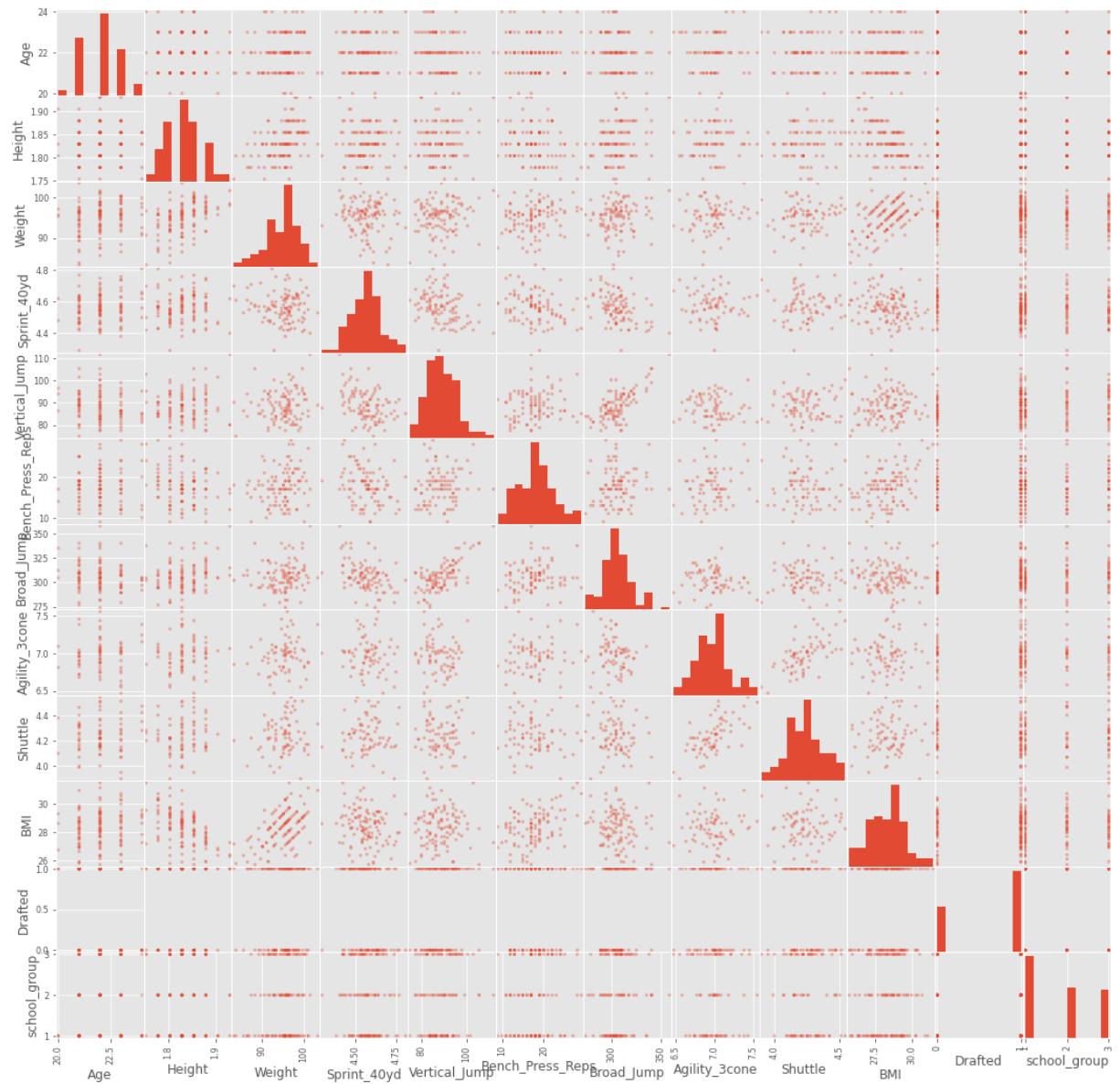


Similar to the previous violin plots, it looks like certain drills have some ability to predict whether someone will make a Pro Bowl, but it also depends on the position they play.

There were not enough players at fullback, punter, or kicker who have made a Pro Bowl to calculate anything.

Below, I look at the scatter matrix of the numerical data. When considering all of the players at once, it looks like there are strong correlations between different features. However, this is mostly due to the fact that different positions have different levels of performance. For example, wide receivers will typically run faster AND jump higher than centers. If the scatter matrix only considers one position at a time, then the correlations tend to be much lower.

```
In [23]: cols = ['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Age', 'BMI', 'Drafted', 'school_group']
pd.plotting.scatter_matrix(df_combine[df_combine['Position']=='SS'][cols], figsize=(18,18), alpha=0.5)
```



Missing Values

For certain classifiers, it is important to fill missing values. As discussed before, it would not make much sense to just take the mean or median of a given column since different football positions have such different performance on the drills. Instead, I should impute using just players of the same position.

Below, I wrote two separate classes (ImputeByPosition and ScaleByPosition). These are written with the methods `fit`, `transform`, and `fit_transform` to match sklearn's transformers.

Note: I originally wrote a function rather than a class to do this. The function worked, except that it meant the sklearn Pipeline technically did not do the right thing because it would use the test data to define the mean and standard deviation for rescaling the test data when it should use the training data for that.


```
In [22]: class ImputeByPosition:
    def __init__(self, split_by_position='default'):
        """
        This is a method for imputing missing values in the Combine data.

        Options:
        split_by_position: A list of columns that should be split by position.
        The 'default' option will split all numerical columns except Age.
        Other options are None and 'all'.
        """
        self.split_by_position = split_by_position
        self.fitted = False
        if split_by_position == 'default':
            # Columns where position IS NOT taken into account
            self.cols1 = ['Age']

            # Columns where position IS taken into account
            self.cols2 = ['Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Bench_Press_Rep', 'Shuttle', 'BMI']

        if split_by_position == 'all':
            # Columns where position IS NOT taken into account
            self.cols1 = []

            # Columns where position IS taken into account
            self.cols2 = ['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Bench_Press_Rep', 'Agility_3cone', 'Shuttle', 'BMI']

        if split_by_position == None:
            # Columns where position IS NOT taken into account
            self.cols1 = ['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Bench_Press_Rep', 'Agility_3cone', 'Shuttle', 'BMI']

            # Columns where position IS taken into account
            self.cols2 = []

    def fit(self, X, y=None):
        means = {}
        self.positions = list(X['Position'].unique())
        for col in X.columns:
            if col in self.cols1: means[col] = X[col].mean()
            if col in self.cols2: means[col] = [X[X['Position']==pos][col].mean() for pos in self.positions]

        self.fitted = True
        self.means = means

    def transform(self, X):
        if self.fitted:
            Xnew = X.copy()

            for col in X.columns:
                if col in self.cols1: Xnew.loc[Xnew[col].isna()==True,col] = self.means[col]
                if col in self.cols2:
                    for i, pos in enumerate(self.positions):
                        Xnew.loc[(Xnew[col].isna()==True) & (Xnew['Position']==pos),col] = self.means[pos]

            return Xnew

        else:
            raise Exception('Need to fit before transforming')

    def fit_transform(self, X, y=None):
        self.fit(X)
        return self.transform(X)

class ScaleByPosition:
    def __init__(self, split_by_position='default'):
        """
```

This is a method for rescaling columns in the Combine data.

```
Options:  
split_by_position: A list of columns that should be split by position.  
The 'default' option will split all numerical columns except Age.  
Other options are None and 'all'.  
"""  
self.split_by_position = split_by_position  
self.fitted = False  
if split_by_position == 'default':  
    # Columns where position IS NOT taken into account  
    self.cols1 = ['Age']  
  
    # Columns where position IS taken into account  
    self.cols2 = ['Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Bro  
        'Shuttle', 'BMI']  
  
if split_by_position == 'all':  
    # Columns where position IS NOT taken into account  
    self.cols1 = []  
  
    # Columns where position IS taken into account  
    self.cols2 = ['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps',  
        'Agility_3cone', 'Shuttle', 'BMI']  
  
if split_by_position == None:  
    # Columns where position IS NOT taken into account  
    self.cols1 = ['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps',  
        'Agility_3cone', 'Shuttle', 'BMI']  
  
    # Columns where position IS taken into account  
    self.cols2 = []  
  
def fit(self, X, y=None):  
    means = {}  
    stds = {}  
    self.positions = list(X['Position'].unique())  
    for col in X.columns:  
        if col in self.cols1:  
            means[col] = X[col].mean()  
            stds[col] = X[col].std()  
        if col in self.cols2:  
            means[col] = [X[X['Position']==pos][col].mean() for pos in self.positions]  
            stds[col] = [X[X['Position']==pos][col].std() for pos in self.positions]  
  
    self.fitted = True  
    self.means = means  
    self.stds = stds  
  
def transform(self, X):  
    if self.fitted:  
        Xnew = X.copy()  
  
        for col in X.columns:  
            if col in self.cols1:  
                Xnew[col] -= self.means[col]  
                Xnew[col] /= self.stds[col]  
            if col in self.cols2:  
                for i, pos in enumerate(self.positions):  
                    Xnew.loc[Xnew['Position']==pos,col] -= self.means[col][i]  
                    Xnew.loc[Xnew['Position']==pos,col] /= self.stds[col][i]  
  
    return Xnew  
  
else:  
    raise Exception('Need to fit before transforming')  
  
def fit_transform(self, X, y=None):  
    self.fit(X)
```

```
return self.transform(X)
```

In [23]: # Try out scaler

```
scaler = ScaleByPosition()
scaler.fit(df_combine)
df_fill = scaler.transform(df_combine)
df_fill
```

Out[23]:

	Year	Player	Age	School	Height	Weight	Sprint_40yd	Vertical_Jump	Bench_Press_Reps	Broad_Jump
0	2009	Beanie Wells	-2.057429	Ohio St.	1.462241	1.589628	-1.585033	-0.349712	1.239326	-
1	2009	Will Davis	-0.022988	Illinois	-1.372217	-0.604464	0.127388	0.065980	0.722462	-
2	2009	Herman Johnson	2.011453	LSU	2.519135	4.064537	1.148935	NaN	-0.902263	-
3	2009	Rashad Johnson	0.994232	Alabama	-0.900537	-0.156677	-0.584485	0.471376	-0.231025	-
4	2009	Cody Brown	-0.022988	Connecticut	-0.045629	0.056673	0.661650	0.663379	0.724782	-
...
3570	2019	Jalen Jelks	-0.022988	Oregon	1.037763	-0.969300	0.709398	-0.100910	-0.999512	-
3571	2019	Chris Slayton	-0.022988	Syracuse	0.782574	-0.046170	0.221823	-0.060600	0.031218	-
3572	2019	Javon Patterson	-1.040208	Mississippi	-0.299587	0.361851	-0.676696	0.071673	0.125524	-
3573	2019	Dakota Allen	0.994232	Texas Tech	-0.675117	-1.299107	0.269931	-0.647072	0.267538	-
3574	2019	Jordan Brailford	0.994232	Oklahoma St.	0.633663	0.797571	-0.288813	0.977159	-0.150822	-

3575 rows × 25 columns



Now the values have been rescaled so that the values are more like z-scores specific to that player's position.

In [24]:

```
# Try out imputer
imputer = ImputeByPosition()
imputer.fit(df_fill)
df_fill = imputer.transform(df_fill)
df_fill
```

Out[24]:

	Year	Player	Age	School	Height	Weight	Sprint_40yd	Vertical_Jump	Bench_Press_Reps	Broad_Jump
0	2009	Beanie Wells	-2.057429	Ohio St.	1.462241	1.589628	-1.585033	-3.497123e-01	1.239326	1.72
1	2009	Will Davis	-0.022988	Illinois	-1.372217	-0.604464	0.127388	6.597976e-02	0.722462	-2
2	2009	Herman Johnson	2.011453	LSU	2.519135	4.064537	1.148935	-2.262875e-17	-0.902263	5.18
3	2009	Rashad Johnson	0.994232	Alabama	-0.900537	-0.156677	-0.584485	4.713763e-01	-0.231025	-1
4	2009	Cody Brown	-0.022988	Connecticut	-0.045629	0.056673	0.661650	6.633794e-01	0.724782	2.65
...
3570	2019	Jalen Jelks	-0.022988	Oregon	1.037763	-0.969300	0.709398	-1.009102e-01	-0.999512	-3
3571	2019	Chris Slayton	-0.022988	Syracuse	0.782574	-0.046170	0.221823	-6.059972e-02	0.031218	-6
3572	2019	Javon Patterson	-1.040208	Mississippi	-0.299587	0.361851	-0.676696	7.167333e-02	0.125524	-5
3573	2019	Dakota Allen	0.994232	Texas Tech	-0.675117	-1.299107	0.269931	-6.470723e-01	0.267538	-4
3574	2019	Jordan Brailford	0.994232	Oklahoma St.	0.633663	0.797571	-0.288813	9.771593e-01	-0.150822	1.25

3575 rows × 25 columns

The values that used to be missing are now set to the average value for a given player's position. Since this was run on rescaled data, the means should be 0 (or very close to it, notice the Vertical_Jump for the third row is -2e-17). If this had been run on data that had not been rescaled, the imputed values would just be the averages per position.

Below, we can see that out of 48 kickers, none did the Agility_3cone or Shuttle drills, and 3 or fewer did Vertical_Jump, Bench_Press_Reps, and Broad_Jump. A similar story is seen with punters. I am going to cut both from the dataset because both positions put great emphasis on a specific skill (kicking) and that skill is not captured in the Combine.

I am going to cut long snappers from the dataset because there are so few of them.

There will still be 3451 players in the dataset.

In [25]: df_combine[df_combine['Position']=='K'].describe()

Out[25]:

	Year	Age	Height	Weight	Sprint_40yd	Vertical_Jump	Bench_Press_Reps	Broad_Jump	Agi
count	48.000000	47.000000	48.000000	48.000000	35.000000	1.00		3.0	2.000000
mean	2013.875000	22.234043	1.822979	88.979703	4.910000	85.09		17.0	288.290000
std	3.077890	0.757937	0.051933	7.379699	0.146348	NaN		7.0	8.980256
min	2009.000000	21.000000	1.727200	74.389149	4.570000	85.09		12.0	281.940000
25%	2011.000000	22.000000	1.778000	83.914588	4.805000	85.09		13.0	285.115000
50%	2014.000000	22.000000	1.828800	87.770124	4.930000	85.09		14.0	288.290000
75%	2016.000000	23.000000	1.854200	94.460611	4.960000	85.09		19.5	291.465000
max	2019.000000	24.000000	1.955800	107.047799	5.250000	85.09		25.0	294.640000

In [26]: # Cut kickers, punters, and long snappers
df_combine.drop(df_combine[df_combine['Position'].isin(['K', 'P', 'LS'])].index, inplace=True)
df_combine

Out[26]:

	Year	Player	Age	School	Height	Weight	Sprint_40yd	Vertical_Jump	Bench_Press_Reps	Broad_J
0	2009	Beanie Wells	20.0	Ohio St.	1.854200	106.594207	4.38	85.09		25.0
1	2009	Will Davis	22.0	Illinois	1.879600	118.387609	4.84	83.82		27.0
2	2009	Herman Johnson	24.0	LSU	2.006600	165.107623	5.50	NaN		21.0
3	2009	Rashad Johnson	23.0	Alabama	1.803400	92.079251	4.49	93.98		15.0
4	2009	Cody Brown	22.0	Connecticut	1.879600	110.676538	4.76	92.71		26.0
...
3570	2019	Jalen Jelks	22.0	Oregon	1.965325	116.119647	4.92	82.55		19.0
3571	2019	Chris Slayton	22.0	Syracuse	1.930400	139.252858	5.13	73.66		28.0
3572	2019	Javon Patterson	21.0	Mississippi	1.905000	139.252858	5.13	69.85		27.0
3573	2019	Dakota Allen	23.0	Texas Tech	1.838325	105.233430	4.77	80.01		23.0
3574	2019	Jordan Brailford	23.0	Oklahoma St.	1.905000	114.305277	4.65	95.25		32

3451 rows × 25 columns

Before I actually start fitting any models, I want to see how much each individual attribute does at separating drafted and undrafted players at each position. To do this, I consider each combination of attribute and position, I loop through different cutoff values of the attribute, and I calculate which cutoff gives the highest accuracy.

This should give me a sense of which attributes matter the most, which attributes are more important for each position, and for which positions I should expect to make more accurate predictions.

```
In [27]: cols = ['Height','Weight','BMI','Sprint_40yd','Vertical_Jump','Bench_Press_Reps','Broad_Jump','Age']

n_vals = 100

cutoffs = np.zeros((len(cols),len(df_combine['Position'].unique())))
accuracy = np.zeros((len(cols),len(df_combine['Position'].unique())))
recall0 = np.zeros((len(cols),len(df_combine['Position'].unique())))
recall1 = np.zeros((len(cols),len(df_combine['Position'].unique())))
precision0 = np.zeros((len(cols),len(df_combine['Position'].unique())))
precision1 = np.zeros((len(cols),len(df_combine['Position'].unique())))

for i, col in enumerate(cols):
    for j, pos in enumerate(df_combine['Position'].unique()):
        min_val = np.min(df_combine[df_combine['Position']==pos][col])
        max_val = np.max(df_combine[df_combine['Position']==pos][col])
        delta = (max_val - min_val) / n_vals
        val = min_val
        mean0 = np.mean(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==0)][col])
        mean1 = np.mean(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==1)][col])
        best_val = min_val
        best_acc = 0.0
        best_recall_precision = [0.0, 0.0, 0.0, 0.0]
        while val <= max_val:
            if mean1 > mean0:
                TN = sum(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==0)][col])
                FP = sum(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==0)][col])
                FN = sum(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==1)][col])
                TP = sum(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==1)][col])
                # If on the cutoff, split it evenly
                TN += sum(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==0)][col])
                FP += sum(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==0)][col])
                FN += sum(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==1)][col])
                TP += sum(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==1)][col])
            else:
                TN = sum(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==0)][col])
                FP = sum(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==0)][col])
                FN = sum(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==1)][col])
                TP = sum(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==1)][col])
                # If on the cutoff, split it evenly
                TN += sum(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==0)][col])
                FP += sum(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==0)][col])
                FN += sum(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==1)][col])
                TP += sum(df_combine[(df_combine['Position']==pos) & (df_combine['Drafted']==1)][col])

            # Normal accuracy
            acc = (TN + TP) / (TN + FP + FN + TP)
            # Balanced accuracy
            #acc = TN / (TN + FP) / 2 + TP / (TP + FN) / 2

            if acc > best_acc:
                best_acc = acc
                best_val = val
                best_recall_precision = [TN/(TN+FP), TP/(TP+FN), TN/(TN+FN), TP/(TP+FP)]

            val += delta

        cutoffs[i,j] = best_val
        accuracy[i,j] = best_acc
        recall0[i,j] = best_recall_precision[0]
        recall1[i,j] = best_recall_precision[1]
        precision0[i,j] = best_recall_precision[2]
        precision1[i,j] = best_recall_precision[3]
```

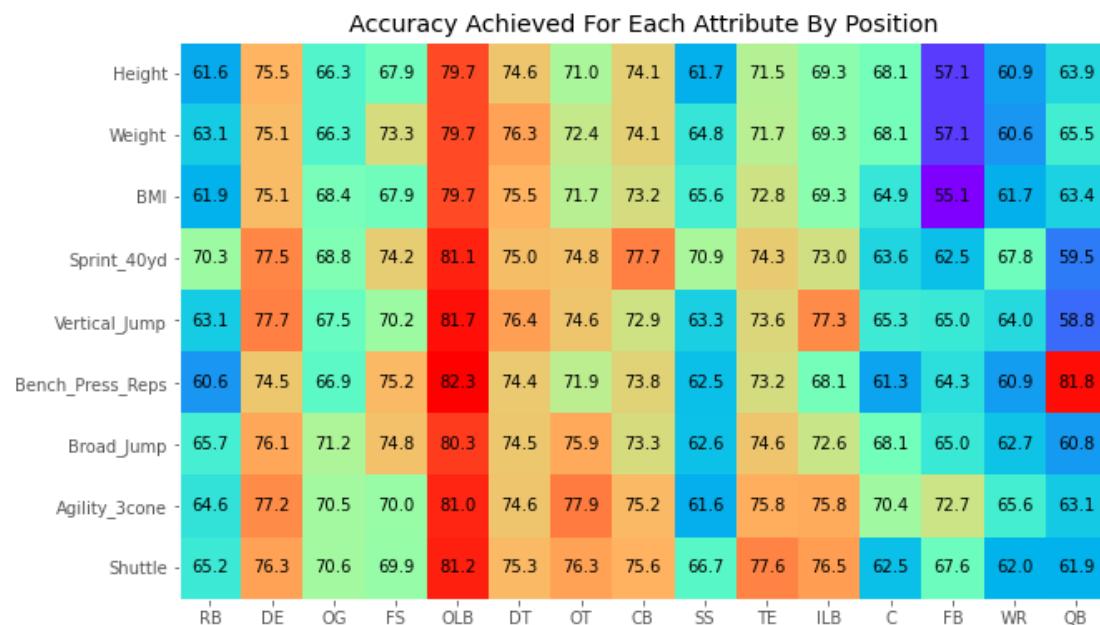
The figure below represents the accuracy I would achieve just by using one attribute for each position.

For example, The pixel corresponding to Sprint_40yd and RB is equal to 70.3. This means that if I only used Sprint_40yd to predict which running backs were drafted, I would expect an accuracy of about 70.3%.

One caveat is that these numbers do not take into account missing data. I could potentially achieve lower accuracy just because the model struggles to make predictions for players who did not complete all of the athletic drills.

```
In [29]: fig, ax = plt.subplots(figsize=(10,10))

ax.grid(False)
ax.imshow(accuracy, cmap='rainbow')
ax.set_yticks(np.arange(len(cols)))
ax.set_yticklabels(cols)
ax.set_xticks(np.arange(len(df_combine['Position'].unique())))
ax.set_xticklabels(df_combine['Position'].unique())
ax.set_title('Accuracy Achieved For Each Attribute By Position')
for i in np.arange(len(cols)):
    for j in np.arange(len(df_combine['Position'].unique())):
        ax.text(j,i,round(100*accuracy[i,j],1), horizontalalignment='center', verticalalignment=
```



The figure indicates that it will be easier to make predictions for certain positions. For example, all of the attributes achieved an accuracy above 79% for outside linebackers (OLB), so I should expect at least that level of accuracy for that position. However, wide receivers had accuracies between 60% and 68%, much lower than for most of the other positions.

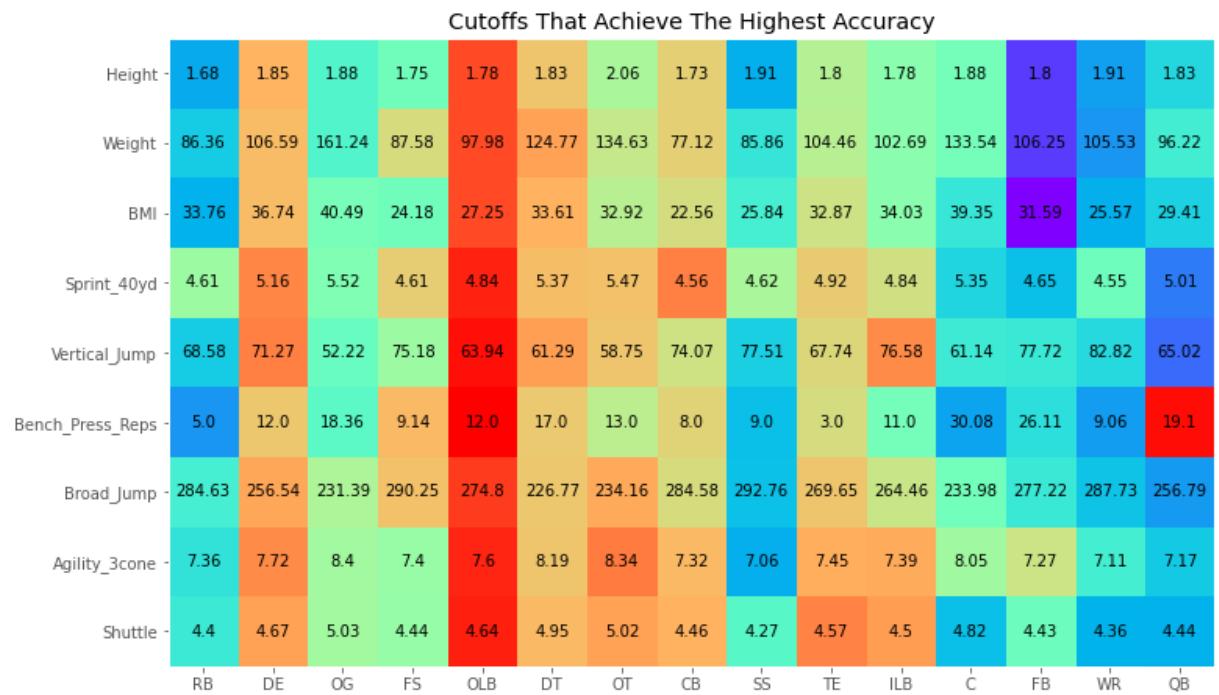
Hypothetically, it might still be possible to achieve an accuracy higher than 68% by combining the attributes together, which the models will do, but it is still reasonable to think that it will be difficult to make accurate predictions for wide receivers.

Overall, the positions that are considered "skill positions" appear to have lower accuracies. These include RB, WR, and QB. This is not really a surprise. These positions are called "skill positions" because there are specific skills like throwing and catching that are not reflected in the Combine data. This means that there is other information that scouts use to evaluate these players other than how fast and athletic they are.

The plot below is similar to the one above, except that the numbers represent the cutoffs that achieve the highest accuracy. The pixels are still color-coded based on accuracy.

```
In [28]: fig, ax = plt.subplots(figsize=(12,12))

ax.grid(False)
ax.imshow(accuracy, cmap='rainbow')
ax.set_yticks(np.arange(len(cols)))
ax.set_yticklabels(cols)
ax.set_xticks(np.arange(len(df_combine['Position'].unique())))
ax.set_xlabels(df_combine['Position'].unique())
ax.set_title('Cutoffs That Achieve The Highest Accuracy')
for i in np.arange(len(cols)):
    for j in np.arange(len(df_combine['Position'].unique())):
        ax.text(j,i,round(cutoffs[i,j],2), horizontalalignment='center', verticalalignment='center')
```



Some of the numbers above actually correspond to the maximum or minimum value recorded for that combination. This means that the highest accuracy was achieved by simply assuming all players were drafted.

Classification Models

Below, I iteratively create models using each of the following types of classifiers:

1. Logistic Regression
2. Decision Tree
3. K Nearest Neighbors
4. Naive Bayes
5. Random Forest
6. XGBoost

```
In [27]: ┌─ from sklearn.linear_model import LogisticRegression
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.neighbors import KNeighborsClassifier
    from sklearn.ensemble import RandomForestClassifier
    from sklearn.model_selection import train_test_split, GridSearchCV
    from sklearn.metrics import classification_report, accuracy_score
    from sklearn.preprocessing import StandardScaler, OneHotEncoder
    from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, roc_curve, auc, RocCurveDisplay
    from xgboost import XGBClassifier
    from sklearn.preprocessing import FunctionTransformer
    from sklearn.pipeline import Pipeline
    from imblearn.over_sampling import SMOTE
    from imblearn.pipeline import Pipeline as Pipe
```

Below, I set a random seed that will be used for the train_test_split for every model

```
In [28]: ┌─ rand_seed = 23
```

I am going to be iteratively fitting models. Each time I fit a model, I want to look at the classification report for both the training and test data, the confusion matrix, and the accuracy for each position (for the test data). I write a function below to do that.

```
In [29]: ┌─ # Function for displaying results of a model
  def display_model_results(y_train, y_train_pred, y_test, y_test_pred, df):
      """
      Function for displaying the results of fitting a model.
      The function will display the following:
      1. Classification report for both training and test data.
      2. Confusion matrix for the test data.
      3. Accuracy for each position for the test data.

      Input:
      y_train: Actual class for training data
      y_train_pred: Predicted class for training data
      y_test: Actual class for test data
      y_test_pred: Predicted class for test data
      df: Original dataframe
      """

      print('Training:')
      print(classification_report(y_train, y_train_pred))
      print('Test:')
      print(classification_report(y_test, y_test_pred))

      df_test = df.loc[y_test.index]
      accuracy_by_pos = np.zeros(len(df['Position'].unique()))
      for i, pos in enumerate(df['Position'].unique()):
          accuracy_by_pos[i] = sum(df_test[df_test['Position']==pos]['Drafted']==y_test_pred[df_test.index==df_test['Position']==pos])/sum(df_test['Position']==pos)

      fig, ax = plt.subplots(ncols=2, figsize=(15,5))
      ax[0].grid(False)
      cf = confusion_matrix(y_test, y_test_pred)
      ConfusionMatrixDisplay(cf, display_labels=['Not Drafted', 'Drafted']).plot(ax=ax[0])

      ax[1].bar(np.arange(len(df['Position'].unique())),accuracy_by_pos)
      ax[1].set_xticks(np.arange(len(df['Position'].unique())))
      ax[1].set_xticklabels(df['Position'].unique())
      ax[1].set_ylabel('Accuracy');
```

Logistic Regression

A logistic regression model cannot handle missing values, therefore, I will use the `ImputeAndScaleByPosition` function that I wrote to fill missing values. Technically, a logistic regression model does not require rescaling (if all the values in a column doubled, then you could compensate for that by halving the corresponding coefficient). However, if the logistic regression uses regularization to punish large parameter values, then scaling values to a common scale becomes important. Therefore, I also rescale the numerical values.

Model 1: Baseline Logistic Regression

Below, I fit a logistic regression model WITHOUT treating the positions separately when imputing and scaling. The model still takes into account position by one-hot-encoding the positions.

```
In [463]: # Define X and y
X = df_combine[['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Shuttle', 'BMI', 'Position', 'school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

# Apply scaler
scaler = ScaleByPosition(split_by_position=None)
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

# Apply imputer
imputer = ImputeByPosition(split_by_position=None)
imputer.fit(X_train)
X_train = imputer.transform(X_train)
X_test = imputer.transform(X_test)

# Apply get_dummies to separate categorical data into columns
X_train = pd.get_dummies(X_train, columns=['Position', 'school_group'], drop_first=True)
X_test = pd.get_dummies(X_test, columns=['Position', 'school_group'], drop_first=True)
```

```
In [464]: # Instantiate LogisticRegressionModel
logreg = LogisticRegression(C=1e12, random_state=42, solver='lbfgs', max_iter=5000)

# Fit model
logreg.fit(X_train, y_train)

# Get predictions
y_train_pred = logreg.predict(X_train)
y_test_pred = logreg.predict(X_test)

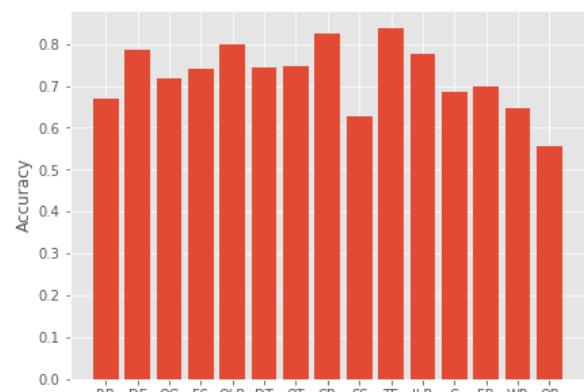
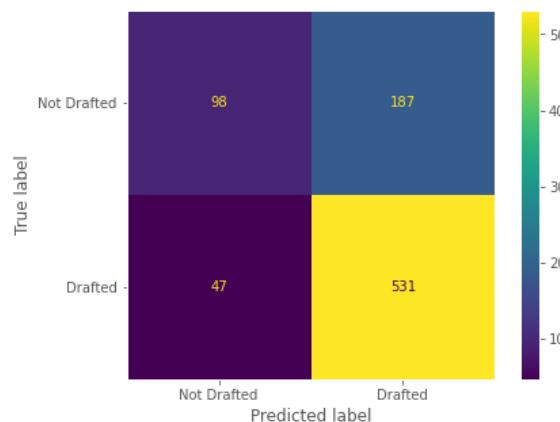
display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

Training:

	precision	recall	f1-score	support
0	0.62	0.31	0.41	808
1	0.74	0.91	0.82	1780
accuracy			0.72	2588
macro avg	0.68	0.61	0.62	2588
weighted avg	0.71	0.72	0.69	2588

Test:

	precision	recall	f1-score	support
0	0.68	0.34	0.46	285
1	0.74	0.92	0.82	578
accuracy			0.73	863
macro avg	0.71	0.63	0.64	863
weighted avg	0.72	0.73	0.70	863



First, the model does not appear to be overfitting since the test data had a very similar accuracy as the training data.

The model is good at identifying players who will be drafted (92% recall), but struggles with players who will not be drafted (34% recall).

The model performed the worst on quarterbacks (55% accuracy). I am not surprised by this; I expect skill positions to be harder to predict based solely on Combine performance.

I am going to look at the ROC curve and AUC score for each model I make. I will use these to help evaluate whether the models are getting better as I iterate.

```
In [465]: # Plot ROC curve
y_test_pred_prob = logreg.predict_proba(X_test)
fig_roc, ax_roc = plt.subplots(figsize=(8,8))

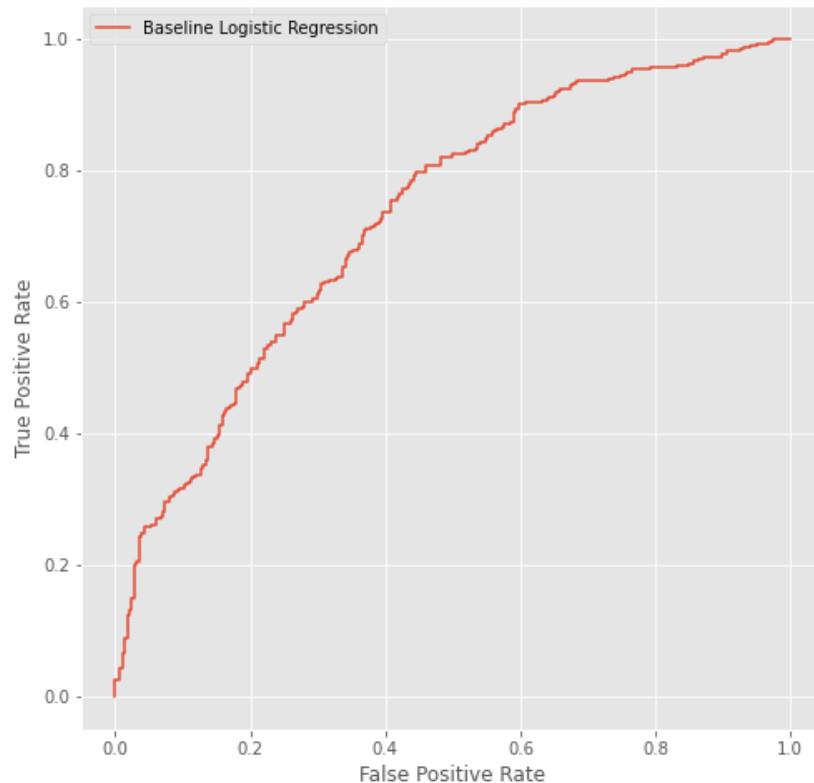
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])
print('AUC:',auc(fpr, tpr))

ax_roc.plot(fpr, tpr, label='Baseline Logistic Regression')

ax_roc.set_xlabel('False Positive Rate')
ax_roc.set_ylabel('True Positive Rate')
ax_roc.legend()
```

AUC: 0.7299277605779153

Out[465]: <matplotlib.legend.Legend at 0x13cb7cd8ee0>



Model 2: Impute and Scale Each Position Separately

Next, I run the same model, but with imputing and scaling done separately for each column taking into account position.
Note: The Age column still does not take into account position, but Height, Weight, and BMI will since I expect those to be different for different positions.

The logistic regression model will not fit an intercept now since each position effectively has its own intercept.

```
In [31]: # Define X and y
X = df_combine[['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Shuttle', 'BMI', 'Position', 'school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

# Apply scaler
scaler = ScaleByPosition()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

# Apply imputer
imputer = ImputeByPosition()
imputer.fit(X_train)
X_train = imputer.transform(X_train)
X_test = imputer.transform(X_test)

# Apply get_dummies to separate categorical data into columns
X_train = pd.get_dummies(X_train, columns=['Position', 'school_group'], drop_first=True)
X_test = pd.get_dummies(X_test, columns=['Position', 'school_group'], drop_first=True)
```

```
In [32]: # Instantiate LogisticRegressionModel
logreg = LogisticRegression(C=1e12, random_state=42, solver='lbfgs', max_iter=5000)

# Fit model
logreg.fit(X_train, y_train)

# Get predictions
y_train_pred = logreg.predict(X_train)
y_test_pred = logreg.predict(X_test)

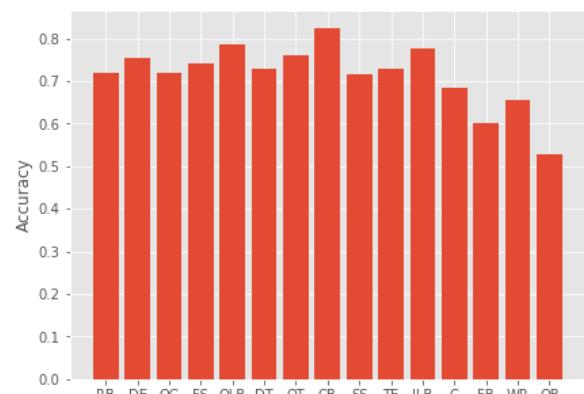
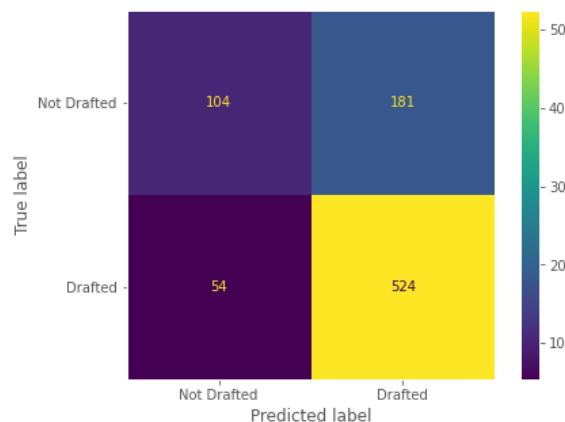
display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

Training:

	precision	recall	f1-score	support
0	0.63	0.36	0.46	808
1	0.76	0.90	0.82	1780
accuracy			0.73	2588
macro avg	0.69	0.63	0.64	2588
weighted avg	0.72	0.73	0.71	2588

Test:

	precision	recall	f1-score	support
0	0.66	0.36	0.47	285
1	0.74	0.91	0.82	578
accuracy			0.73	863
macro avg	0.70	0.64	0.64	863
weighted avg	0.72	0.73	0.70	863

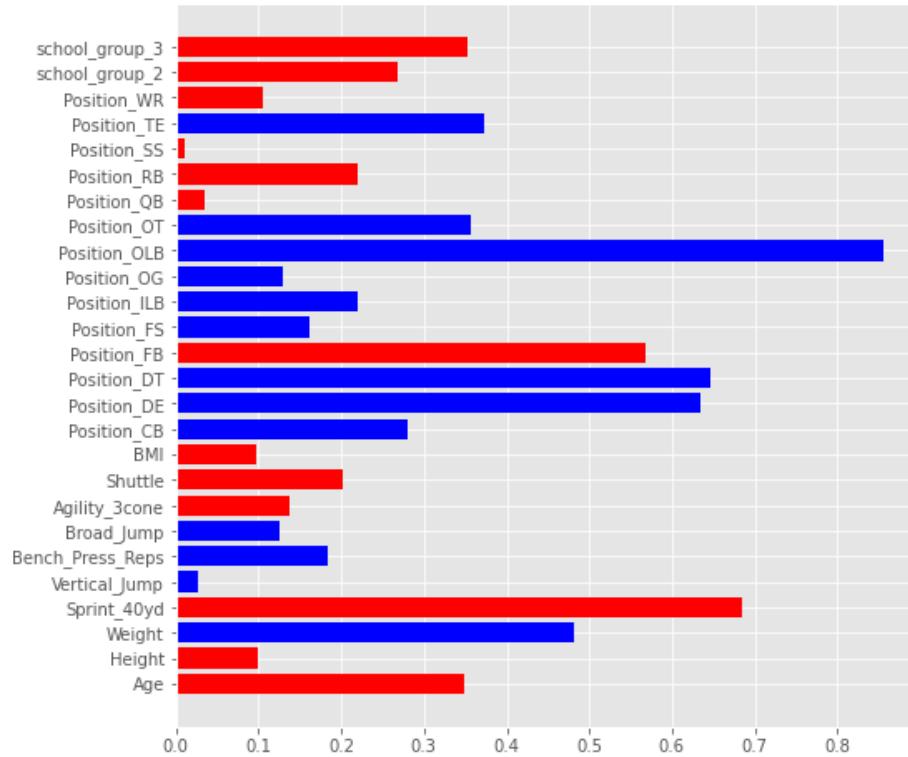


The fit got slightly better for the training data which achieved 1% higher accuracy. The overall performance is not that much better, though.

The model had similar performance on quarterbacks.

```
In [33]: # Look at the coefficients for the model
fig, ax = plt.subplots(figsize=(8,8))

ax.barh(np.arange(len(X_train.columns)), np.abs(logreg.coef_[0]), color=['blue' if x>0 else 'red']
ax.set_yticks(np.arange(len(X_train.columns)))
ax.set_yticklabels(X_train.columns);
```



The largest coefficients are school_group_1, Position_OLB, Position_FB, and Sprint_40yd.

The fact that both school_group_2 and school_group_3 are red means that players from school_group_1 (the 21 schools that send the most players to the Combine) are drafted more often.

The other large coefficients indicate that OLB get drafted at a higher rate than other positions, FB get drafted the least often, and the test that carries the most weight is the 40-yard sprint.

```
In [469]: # Update ROC curve
y_test_pred_prob = logreg.predict_proba(X_test)

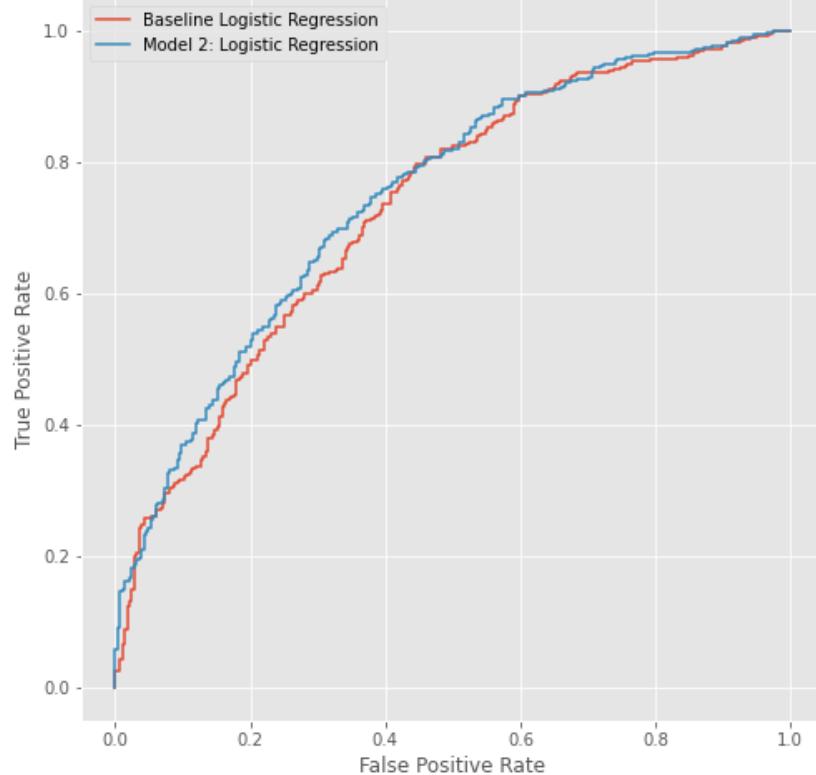
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])
print('AUC:', auc(fpr, tpr))

ax_roc.plot(fpr, tpr, label='Model 2: Logistic Regression')
ax_roc.legend()

fig_roc
```

AUC: 0.7470284708310568

Out[469]:



Below, I look at histograms of the probability calculated from the model. This will help give me a sense of how well the model is separating drafted and undrafted players.

```
In [36]: # y_test_pred_prob = logreg.predict_proba(X_test)

fig, ax = plt.subplots(figsize=(7,7))

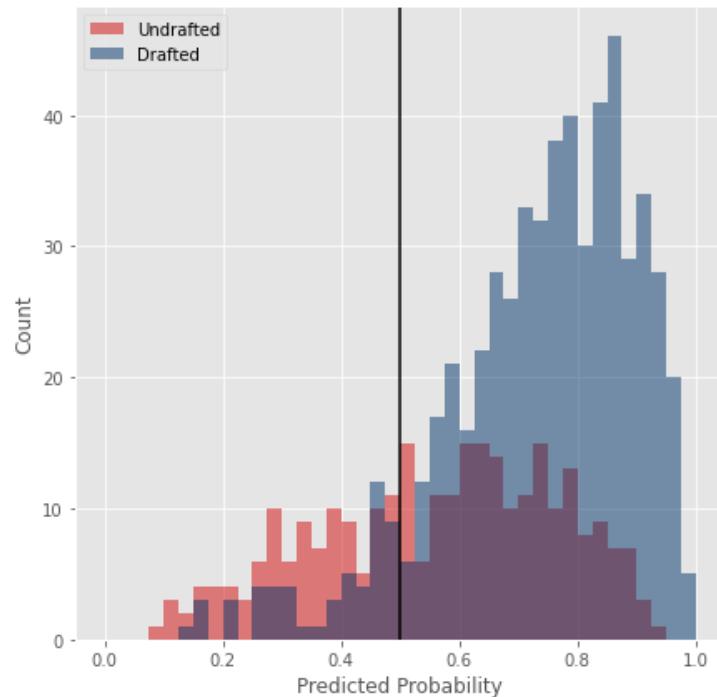
keep = y_test==0
ax.hist(y_test_pred_prob[keep][:,1],bins=np.arange(41)/40,color="#D50A0A",alpha=0.5,label='Undrafted')

keep = y_test==1
ax.hist(y_test_pred_prob[keep][:,1],bins=np.arange(41)/40,color="#013369",alpha=0.5,label='Drafted')

ax.axvline(0.5, color='black')

ax.set_xlabel('Predicted Probability')
ax.set_ylabel('Count')

ax.legend();
```



The black vertical line indicates where the model sees the cutoff between drafted and undrafted players.

The precision could be improved by moving the cutoff to the right.

Model 3: Interaction Terms The two previous models handled position by creating one-hot-encoded columns, one for each position. These are useful if certain positions are drafted more often than others. However, this does not allow the model to determine which columns matter more for which position. For example, this does not allow the model to emphasize the 40-yard sprint for running backs while emphasizing the number of bench press reps for offensive guards.

I address this in this model by creating interaction terms between position and numerical values. Now instead of having one column per drill and one column for each position, there will be columns like "Sprint_40ydxPosition_RB" and "Bench_Press_RepsxPosition_OG". The upside is that the model can now determine what matters to each position separately. The downside is that it creates lots of columns, so there is a higher risk of overfitting.

The function below creates the interaction terms.

```
In [42]: def create_interaction_terms(X, list1):
    """
        This function creates interaction terms between position and other values.

    Input:
    X: The DataFrame that has the independent variables

    list1: A list of attributes to incorporate into interaction terms.
    """
    list2 = list(df_combine['Position'].unique())

    len1 = len(list1)
    len2 = len(list2)

    n = 0
    while n < len1*len2:
        i = n // len2
        j = n % len2
        X[f'{list1[i]}xPosition_{list2[j]}'] = X[list1[i]]*X['Position'].\
            apply(lambda x: 1 if x==list2[j] else 0)

        n += 1

    # Drop individual columns used to make the interaction terms
    X.drop(columns=list1,inplace=True)

    return X
```

```
In [472]: # Define X and y
X = df_combine[['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump',
                 'Shuttle', 'BMI', 'Position', 'school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

# Apply scaler
scaler = ScaleByPosition()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

# Apply imputer
imputer = ImputeByPosition()
imputer.fit(X_train)
X_train = imputer.transform(X_train)
X_test = imputer.transform(X_test)

# Create interaction terms
list1 = ['Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Agility']

X_train = create_interaction_terms(X_train, list1)
X_test = create_interaction_terms(X_test, list1)

# Apply get_dummies to separate categorical data into columns
X_train = pd.get_dummies(X_train, columns=['Position', 'school_group'], drop_first=True)
X_test = pd.get_dummies(X_test, columns=['Position', 'school_group'], drop_first=True)
```

```
In [473]: # Instantiate LogisticRegressionModel
logreg = LogisticRegression(C=1e12, random_state=42, solver='lbfgs', max_iter=5000)

# Fit model
logreg.fit(X_train, y_train)

# Get predictions
y_train_pred = logreg.predict(X_train)
y_test_pred = logreg.predict(X_test)

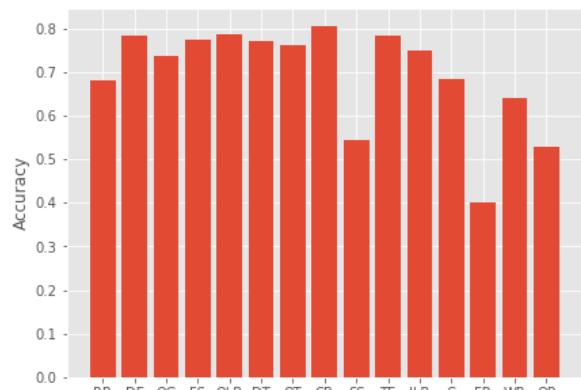
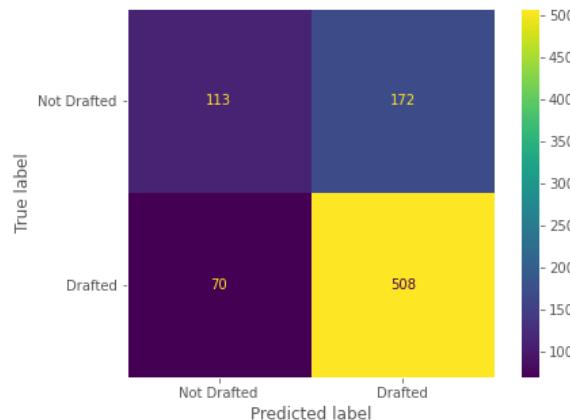
display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

Training:

	precision	recall	f1-score	support
0	0.66	0.46	0.54	808
1	0.78	0.89	0.84	1780
accuracy			0.76	2588
macro avg	0.72	0.68	0.69	2588
weighted avg	0.75	0.76	0.74	2588

Test:

	precision	recall	f1-score	support
0	0.62	0.40	0.48	285
1	0.75	0.88	0.81	578
accuracy			0.72	863
macro avg	0.68	0.64	0.65	863
weighted avg	0.70	0.72	0.70	863

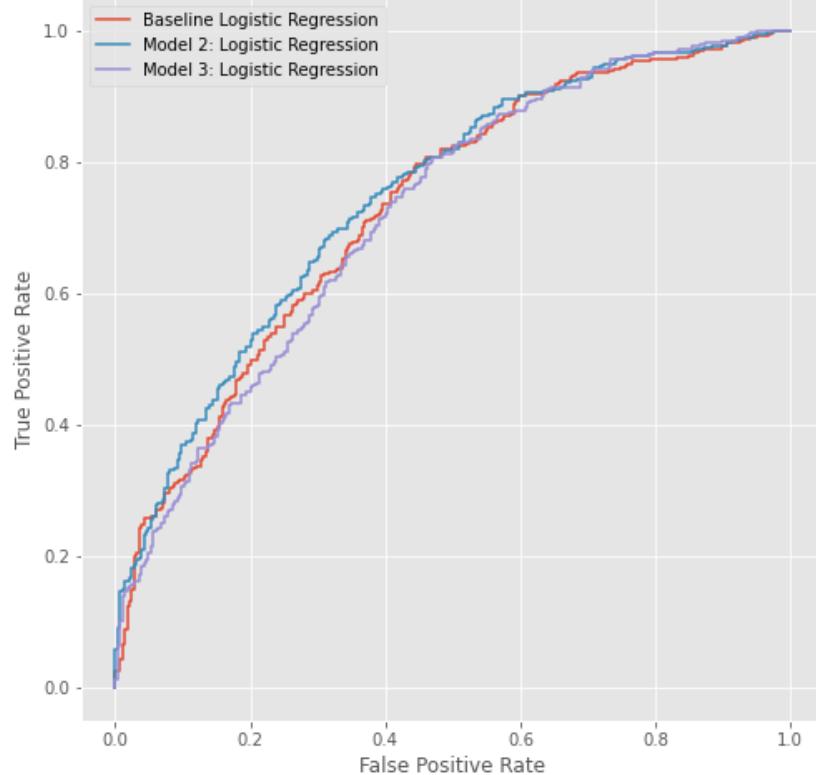


I knew that creating lots of interaction terms might lead to overfitting and it feels like that did happen a little. The accuracy on the test data was 4 percentage points lower than the training accuracy (72% vs 76%). The overall accuracy is a little lower than the previous model.

```
In [474]: # Update ROC Curve  
y_test_pred_prob = logreg.predict_proba(X_test)  
  
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])  
print('AUC:', auc(fpr, tpr))  
  
ax_roc.plot(fpr, tpr, label='Model 3: Logistic Regression')  
ax_roc.legend()  
  
fig_roc
```

AUC: 0.7214350755782188

Out[474]:



Model 4: Regularization, No Interaction Terms One way to reduce overfitting is by using regularization. I have been using the lbfgs solver which uses Lasso regularization, but I have been using a very high C value so that there is essentially no regularization. Below, I will loop through some different values of C to see which works best.

First, I will try it without interaction terms. In the next model, I will try it with interaction terms.

I will start using GridSearchCV with 3 cross validations.

I will try different values of C (regularization constant) and class_weight (which changes relative weight of the two classes)

```
In [47]: # Logistic regression pipeline with GridSearchCV
logreg = LogisticRegression(random_state=42, solver='lbfgs', max_iter=5000)

param_grid_logreg = {
    'logreg__C': [0.01, 0.1, 1, 10, 100, 1000, 1e4, 1e6, 1e8, 1e10],
    'logreg__class_weight': [None, {0:3, 1:2}, 'balanced']
}

pipe = Pipeline(steps=[
    ('scaler', ScaleByPosition()),
    ('imputer', ImputeByPosition()),
    ('dummies', FunctionTransformer(pd.get_dummies, kw_args={'columns': ['Position', 'school_group']})),
    ('logreg', logreg)
])

logreg_gridcv = GridSearchCV(pipe, param_grid=param_grid_logreg, cv=3)

X = df_combine[['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Shuttle', 'BMI', 'Position', 'school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

logreg_gridcv.fit(X_train, y_train)

best_parameters = logreg_gridcv.best_params_

print('Grid Search found the following optimal parameters: ')
for param_name in sorted(best_parameters.keys()):
    print('%s: %r' % (param_name, best_parameters[param_name]))

y_train_pred = logreg_gridcv.predict(X_train)
y_test_pred = logreg_gridcv.predict(X_test)

display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

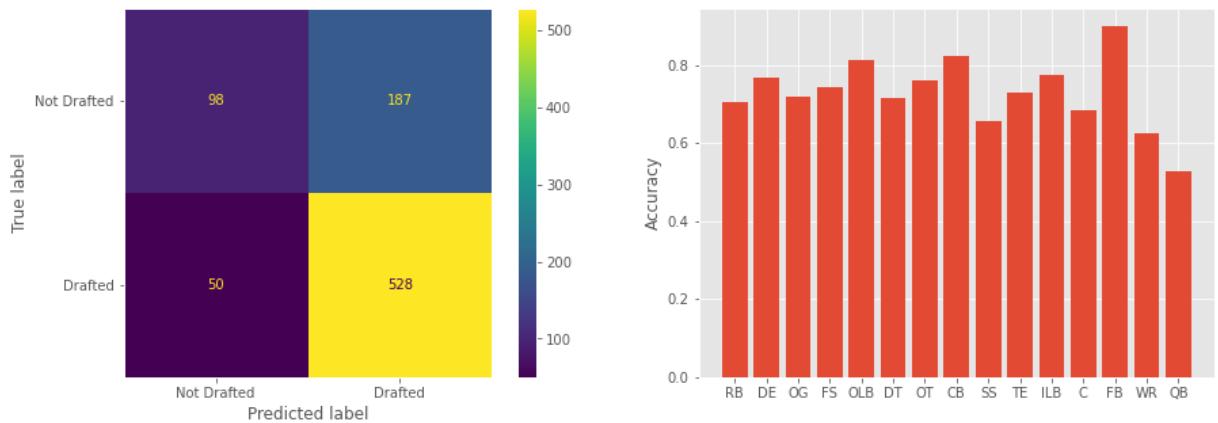
Grid Search found the following optimal parameters:

logreg__C: 0.1
logreg__class_weight: None
Training:

	precision	recall	f1-score	support
0	0.63	0.36	0.45	808
1	0.76	0.91	0.82	1780
accuracy			0.73	2588
macro avg	0.69	0.63	0.64	2588
weighted avg	0.72	0.73	0.71	2588

Test:

	precision	recall	f1-score	support
0	0.66	0.34	0.45	285
1	0.74	0.91	0.82	578
accuracy			0.73	863
macro avg	0.70	0.63	0.63	863
weighted avg	0.71	0.73	0.70	863



The best fit parameters were $C=0.1$ and equal class weights. The performance of the model did not change much from the version without regularization.

```
In [477]: # Update ROC Curve
y_test_pred_prob = logreg_gridcv.predict_proba(X_test)

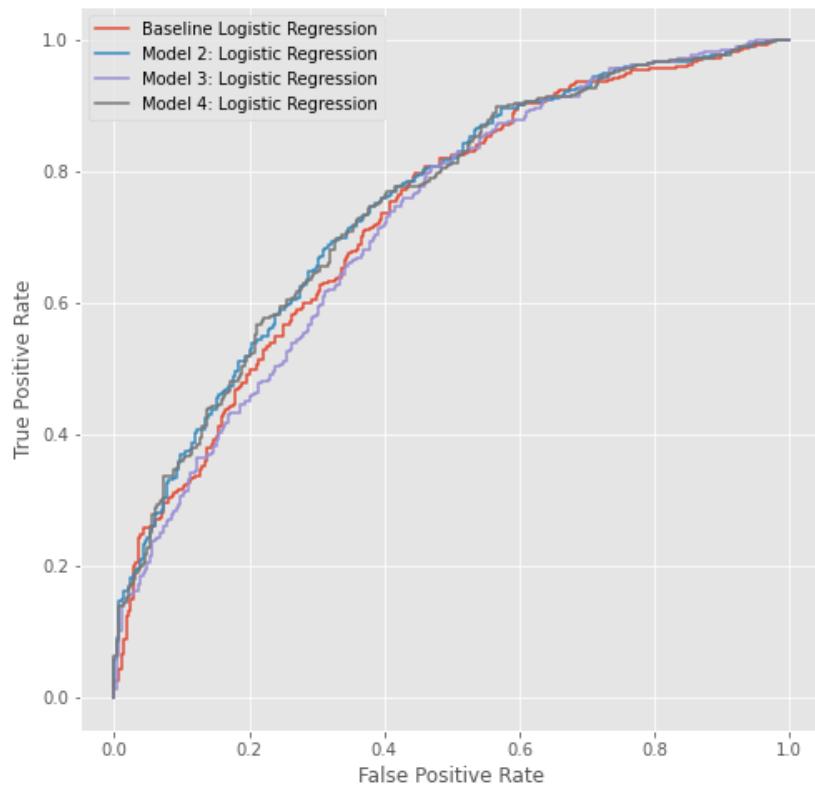
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])
print('AUC:',auc(fpr, tpr))

ax_roc.plot(fpr, tpr, label='Model 4: Logistic Regression')
ax_roc.legend()

fig_roc
```

AUC: 0.7450130516602926

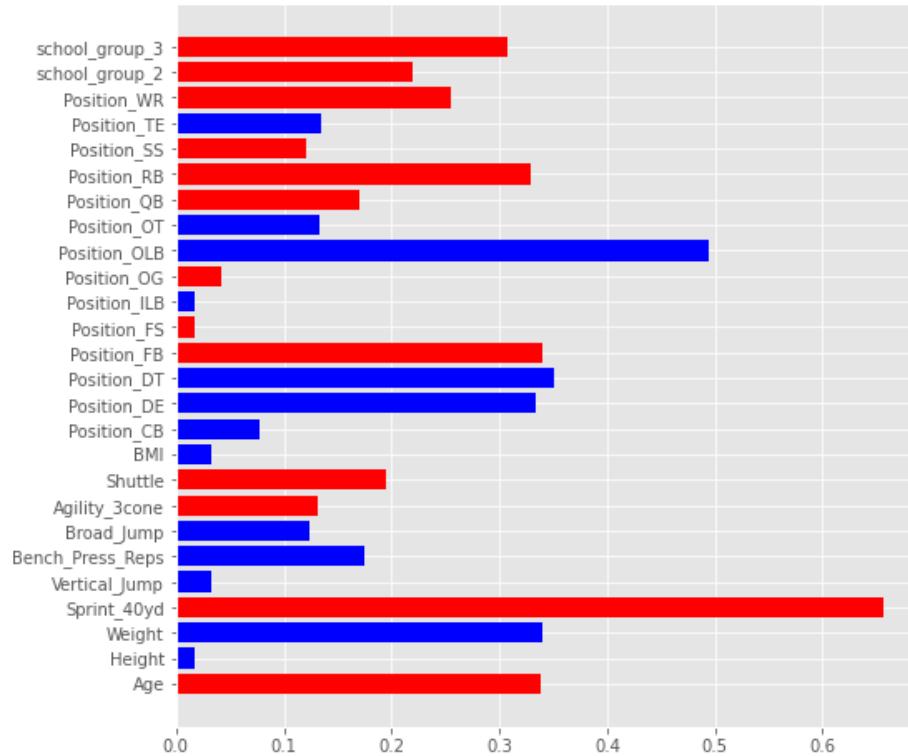
Out[477]:



```
In [38]: # Plot coefficient values for last model
coefs = logreg_gridcv.best_estimator_['logreg'].coef_[0]
cols = pd.get_dummies(X_train, columns=['Position','school_group'], drop_first=True).columns

fig, ax = plt.subplots(figsize=(8,8))

ax.barh(np.arange(len(cols)), np.abs(coefs), color=['blue' if x>0 else 'red' for x in coefs])
ax.set_yticks(np.arange(len(cols)))
ax.set_yticklabels(cols);
```



```
In [54]: # y_test_pred_prob = logreg_gridcv.predict_proba(X_test)

fig, ax = plt.subplots(figsize=(7,7))

keep = y_test==0
ax.hist(y_test_pred_prob[keep][:,1],bins=np.arange(41)/40,color="#D50A0A",alpha=0.5,label='Real: Undrafted')

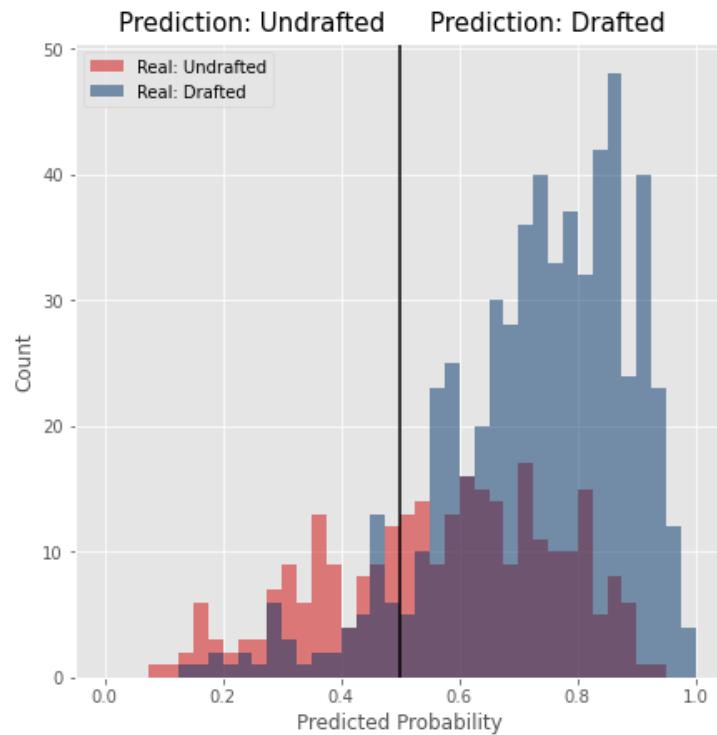
keep = y_test==1
ax.hist(y_test_pred_prob[keep][:,1],bins=np.arange(41)/40,color="#013369",alpha=0.5,label='Real: Drafted')

ax.axvline(0.5, color='black')

ax.set_xlabel('Predicted Probability')
ax.set_ylabel('Count')

ax.text(0.25,52,'Prediction: Undrafted',verticalalignment='center',horizontalalignment='center',color='red')
ax.text(0.75,52,'Prediction: Drafted',verticalalignment='center',horizontalalignment='center',color='blue')

ax.legend();
```



Model 5: Regularization and Interaction Terms

Now I try the same as above, but with interaction terms.

```
In [43]: # Logistic regression pipeline with GridSearchCV
logreg = LogisticRegression(random_state=42, solver='lbfgs', max_iter=5000)

param_grid_logreg = {
    'logreg__C': [0.01, 0.1, 1, 10, 100, 1000, 1e4, 1e6, 1e8, 1e10],
    'logreg__class_weight': [None, {0:3, 1:2}, 'balanced']
}

list1 = ['Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Agility']

pipe = Pipeline(steps=[
    ('scaler', ScaleByPosition()),
    ('imputer', ImputeByPosition()),
    ('interterms', FunctionTransformer(create_interaction_terms, kw_args={'list1':list1})),
    ('dummies', FunctionTransformer(pd.get_dummies, kw_args={'columns':['Position', 'school_group']})),
    ('logreg', logreg)
])

logreg_gridcv = GridSearchCV(pipe, param_grid=param_grid_logreg, cv=3)

X = df_combine[['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump',
                 'Shuttle', 'BMI', 'Position', 'school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

logreg_gridcv.fit(X_train, y_train)

best_parameters = logreg_gridcv.best_params_

print('Grid Search found the following optimal parameters: ')
for param_name in sorted(best_parameters.keys()):
    print('%s: %r' % (param_name, best_parameters[param_name]))

y_train_pred = logreg_gridcv.predict(X_train)
y_test_pred = logreg_gridcv.predict(X_test)

display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

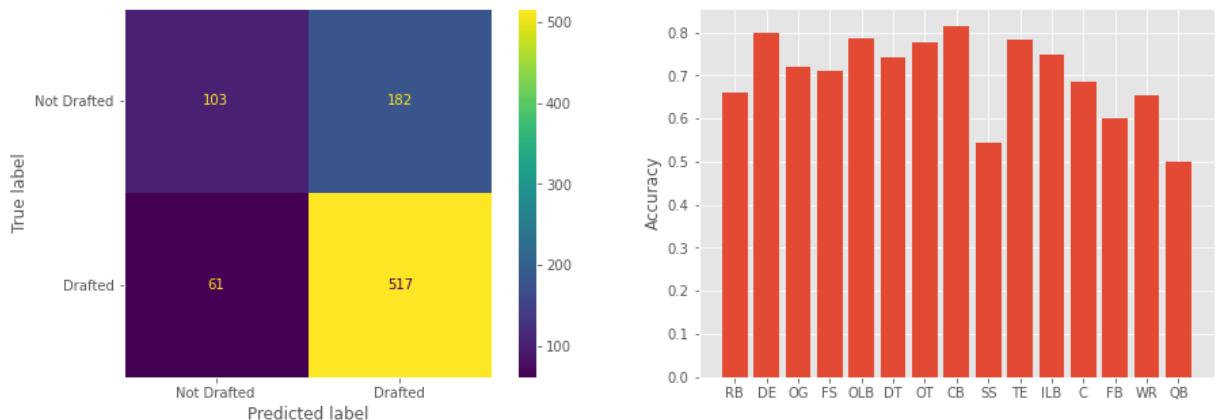
Grid Search found the following optimal parameters:

logreg__C: 1
 logreg__class_weight: None
 Training:

	precision	recall	f1-score	support
0	0.67	0.43	0.52	808
1	0.78	0.90	0.84	1780
accuracy			0.75	2588
macro avg	0.72	0.67	0.68	2588
weighted avg	0.74	0.75	0.74	2588

Test:

	precision	recall	f1-score	support
0	0.63	0.36	0.46	285
1	0.74	0.89	0.81	578
accuracy			0.72	863
macro avg	0.68	0.63	0.63	863
weighted avg	0.70	0.72	0.69	863



For this model, the best C value was 1. The accuracy on the test data and the AUC (see below) are a little lower than the model without interaction terms.

```
In [479]: # Update ROC Curve
y_test_pred_prob = logreg_gridcv.predict_proba(X_test)

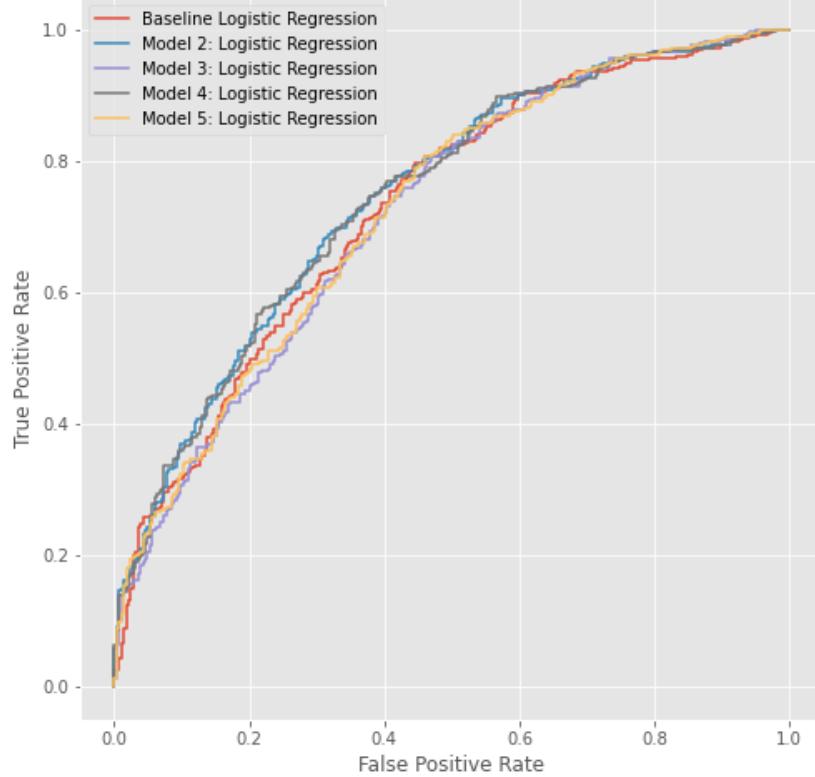
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])
print('AUC:',auc(fpr, tpr))

ax_roc.plot(fpr, tpr, label='Model 5: Logistic Regression')
ax_roc.legend()

fig_roc
```

AUC: 0.7271353123292662

Out[479]:

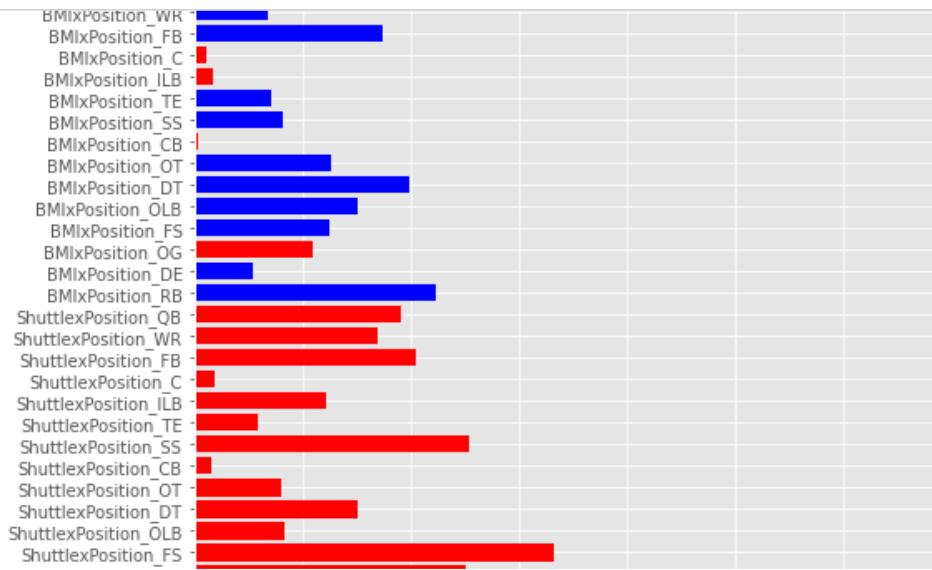


```
In [44]: # Plot coefficient values for last model
coefs = logreg_gridcv.best_estimator_['logreg'].coef_[0]

cols = pd.get_dummies(create_interaction_terms(X_train,list1), columns=['Position','school_group'])

fig, ax = plt.subplots(figsize=(8,40))

ax.barh(np.arange(len(cols)), np.abs(coefs), color=['blue' if x>0 else 'red' for x in coefs])
ax.set_yticks(np.arange(len(cols)))
ax.set_yticklabels(cols);
```



```
In [45]: # y_test_pred_prob = logreg_gridcv.predict_proba(X_test)

fig, ax = plt.subplots(figsize=(7,7))

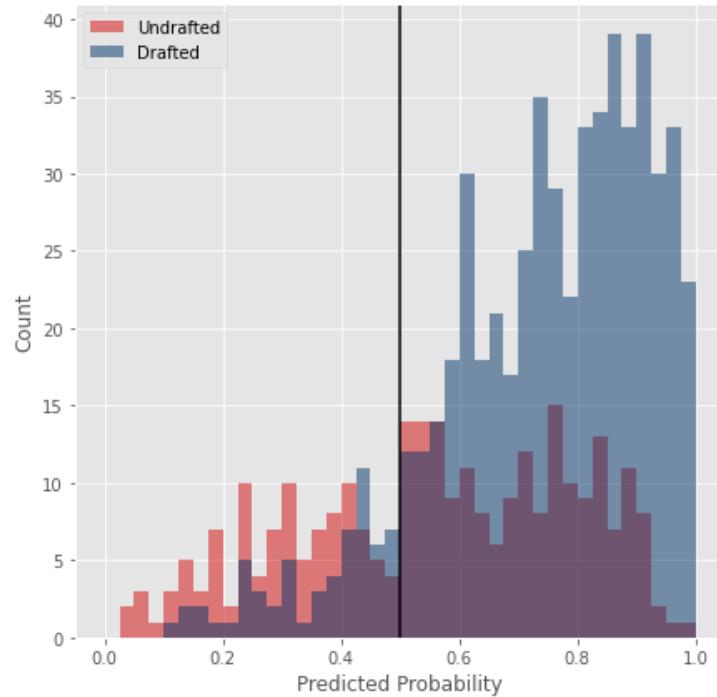
keep = y_test==0
ax.hist(y_test_pred_prob[keep][:,1],bins=np.arange(41)/40,color="#D50A0A",alpha=0.5,label='Undrafted')

keep = y_test==1
ax.hist(y_test_pred_prob[keep][:,1],bins=np.arange(41)/40,color="#013369",alpha=0.5,label='Drafted')

ax.axvline(0.5, color='black')

ax.set_xlabel('Predicted Probability')
ax.set_ylabel('Count')

ax.legend();
```



The last model has a lot of parameters, but the number of parameters that applies to a given player is much lower. For example, a QB will only use Age, the school_group parameters, and any parameter that ends with Position_QB.

There is some evidence for overfitting since certain values are expected to be positive (whenever bigger values are preferred) and others are expected to be negative (whenever smaller values are preferred), but the coefficients don't always reflect this. One example is the 40-yard sprint where lower times should be preferred, so we expect these to be red in the plot above. However, the values are actually blue for QB and C. What this means to me is not that teams actually value having slower quarterbacks and centers, but that random noise or possible correlations with other features might lead to these values in the fit. Maybe the model thinks that slower centers are more likely to get drafted because larger, stronger centers are preferred and they just happen to be slower on average. This does harm the model slightly because the model would predict that a slower quarterback or center would be drafted over a faster one if everything else is equal, but that would not be true in reality.

Evaluating the Logistic Regression Models

The best performing model had C=0.1 and no interaction terms. However, the model with interaction terms was very close in performance.

Description	C	Training Accuracy	Test Accuracy	ROC AUC
Baseline, preprocessing ignores position	1e12	72%	73%	0.730
Preprocessing takes into account position	1e12	73%	73%	0.747
Uses Interaction Terms	1e12	76%	72%	0.721
Stricter regularization, no inter terms	0.1	73%	73%	0.745
Stricter regularization, yes inter terms	1	75%	72%	0.727

Comparing the last two models

The last two models differed in that model 4 did not use interaction terms and model 5 did. The one without interaction terms performed slightly better. I want to compare the predictions more closely, so below I plot the predictions of each model against one another separated by position (the final plot has all of the positions together).

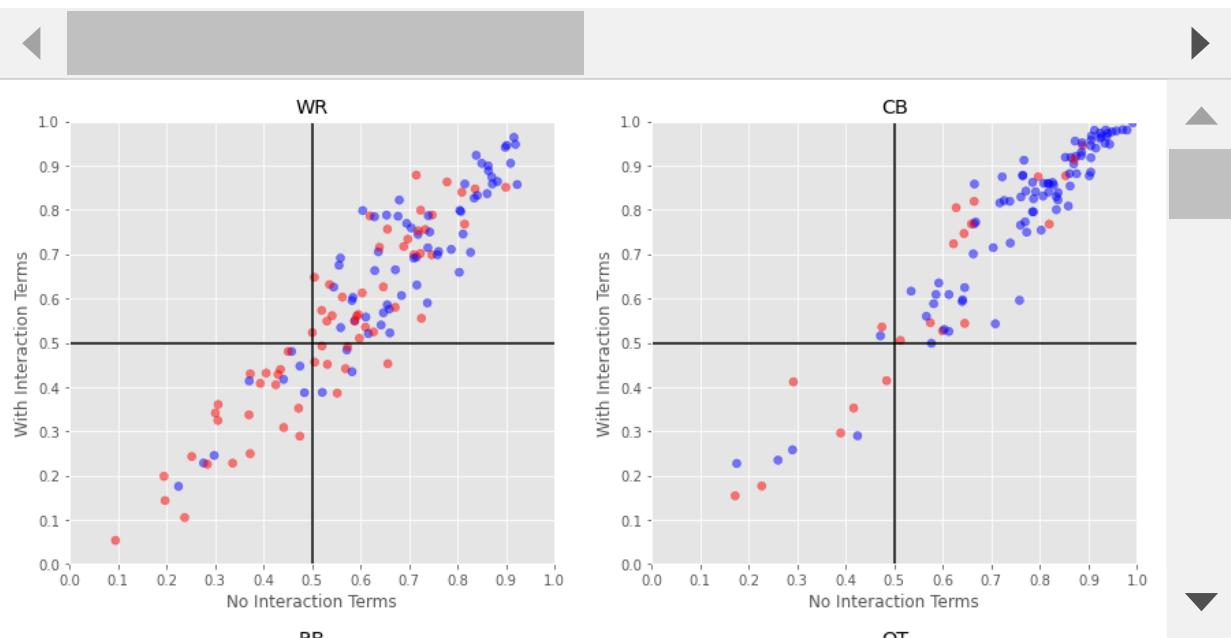
```
In [594]: # Run after model 4  
y_test_pred_0 = logreg_gridcv.predict_proba(X_test)
```

```
In [ ]: # Run after model 5  
y_test_pred_1 = logreg_gridcv.predict_proba(X_test)
```

```
In [648]: fig, ax = plt.subplots(ncols=2, nrows=8, figsize=(14,56))

for i, pos in enumerate(df_combine['Position'].value_counts().index):
    r = i // 2
    c = i % 2
    ax[r,c].scatter(y_test_pred_0[:,1], y_test_pred_1[:,1],
                      color = [(0,0,1,0.5) if (x==1)&(y==pos) else (1,0,0,0.5) if (x==0)&(y==pos) else
                                (0,0,0,0.5)])
    ax[r,c].axvline(0.5, color='black')
    ax[r,c].axhline(0.5, color='black')
    ax[r,c].set_title(pos)
    ax[r,c].set_xticks(np.arange(0,1.1,0.1))
    ax[r,c].set_yticks(np.arange(0,1.1,0.1))
    ax[r,c].set_xbound([0,1])
    ax[r,c].set_ybound([0,1])
    ax[r,c].set_xlabel('No Interaction Terms')
    ax[r,c].set_ylabel('With Interaction Terms')

ax[7,1].scatter(y_test_pred_0[:,1], y_test_pred_1[:,1],
                  color = [(0,0,1,0.5) if x==1 else (1,0,0,0.5) for x in df_combine.loc[y_test.index]])
ax[7,1].axvline(0.5, color='black')
ax[7,1].axhline(0.5, color='black')
ax[7,1].set_title('All')
ax[7,1].set_xticks(np.arange(0,1.1,0.1))
ax[7,1].set_yticks(np.arange(0,1.1,0.1))
ax[7,1].set_xbound([0,1])
ax[7,1].set_ybound([0,1])
ax[7,1].set_xlabel('No Interaction Terms')
ax[7,1].set_ylabel('With Interaction Terms');
```



The positions that have the most players are plotted first.

The models are in tighter agreement for the positions with a large sample size. For instance, the models agree pretty well for wide receivers (WR) and cornerbacks (CB). Note that close agreement between the models does not necessarily correspond to higher accuracy. There are quite a few wide receivers who are predicted to get drafted according to both models, but were not.

The two positions with the lower correlation between the two models' predictions are quarterbacks (QB) and centers (C). Quarterbacks in particular have some of the biggest disagreements between the two models (like one player that is over 90% to get drafted according to one model, but about 30% to get drafted according to the other). The disagreement is partly because they have a lower sample size, but I also think that the quarterbacks are being judged on other criteria other than their combine performance.

Keep Interaction Terms or No?

The performance of the models with and without interaction terms were pretty close. I suspect that the model without those terms are underfitting because they are not able to determine how certain positions care about certain tests more than others. On the other hand, I suspect that the model with the interaction terms is overfitting because the sample size for certain positions is not quite enough to tightly constrain the parameters. Hypothetically, I would expect the model with interaction terms to be better IF the was a very large sample at each position. However, for this project, I think it makes more sense to go with the simpler model if the performance is comparable (and probably slightly better).

I will still try out interaction terms with some of the other classifiers below to see if it makes a difference.

Decision Tree Classifier

For the decision tree classifier, it is necessary to replace missing values. It is also important to rescale the data based on position.

One limitation of the decision tree classifier is that it might struggle to treat different positions separately. Hypothetically, it could start by splitting based on position, but since it is a greedy algorithm and each position gets drafted at a similar rate, it probably will not do that.

Model 1: Default hyperparameters

First, I will create a decision tree model using the default hyperparameters.

The model will impute and scale positions separately.

```
In [354]: # Define X and y
X = df_combine[['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Shuttle', 'BMI', 'Position', 'school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

# Apply scaler
scaler = ScaleByPosition()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

# Apply imputer
imputer = ImputeByPosition()
imputer.fit(X_train)
X_train = imputer.transform(X_train)
X_test = imputer.transform(X_test)

# Apply get_dummies to separate categorical data into columns
X_train = pd.get_dummies(X_train, columns=['Position', 'school_group'])
X_test = pd.get_dummies(X_test, columns=['Position', 'school_group'])
```

In [355]:

```
# Instantiate DecisionTreeClassifier
dtc = DecisionTreeClassifier(random_state=42)

# Fit model
dtc.fit(X_train, y_train)

# Get predictions
y_train_pred = dtc.predict(X_train)
y_test_pred = dtc.predict(X_test)

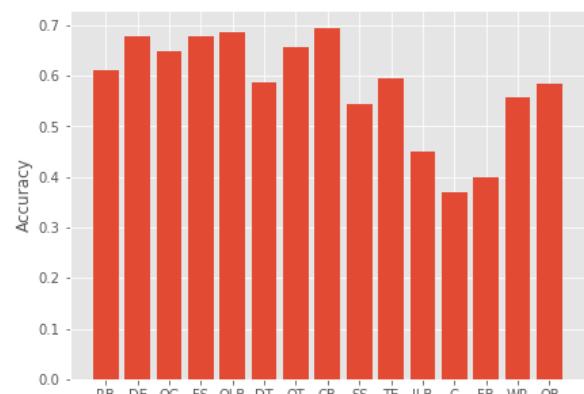
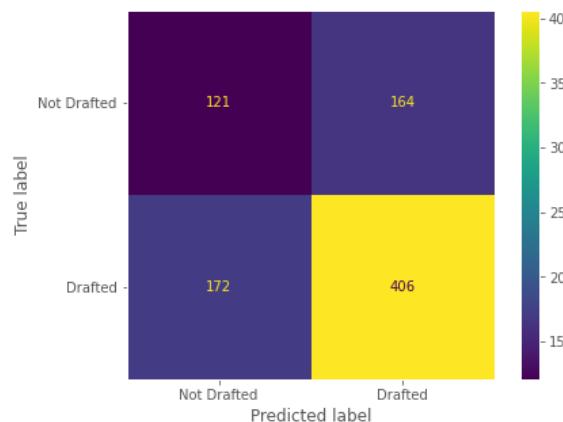
display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

Training:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	808
1	1.00	1.00	1.00	1780
accuracy			1.00	2588
macro avg	1.00	1.00	1.00	2588
weighted avg	1.00	1.00	1.00	2588

Test:

	precision	recall	f1-score	support
0	0.41	0.42	0.42	285
1	0.71	0.70	0.71	578
accuracy			0.61	863
macro avg	0.56	0.56	0.56	863
weighted avg	0.61	0.61	0.61	863



This model is severely overfitting. This is not surprising because the default parameters don't limit the tree from splitting as many times as it needs to in order to get a "perfect" fit. The model needs some hyperparameters to help keep it from overfitting.

```
In [356]: # Plot ROC curve
y_test_pred_prob = dtc.predict_proba(X_test)
fig_roc, ax_roc = plt.subplots(figsize=(8,8))

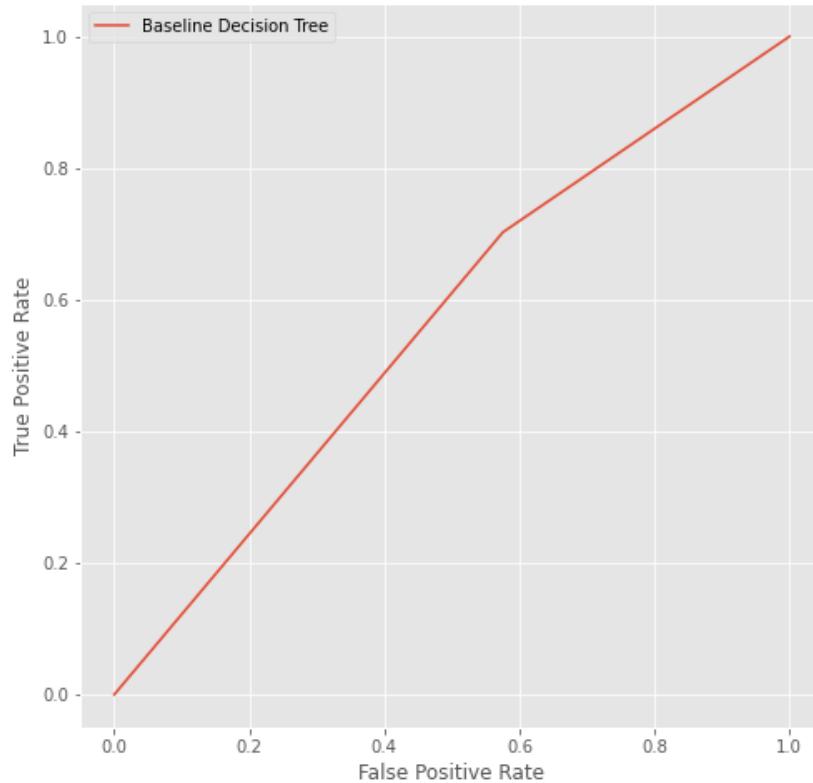
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])
print('AUC:',auc(fpr, tpr))

ax_roc.plot(fpr, tpr, label='Baseline Decision Tree')

ax_roc.set_xlabel('False Positive Rate')
ax_roc.set_ylabel('True Positive Rate')
ax_roc.legend()
```

AUC: 0.5634917744187458

Out[356]: <matplotlib.legend.Legend at 0x13cad776b20>



The ROC curve for the baseline model is not very interesting because of the overfitting. The effect of overfitting the model is that the model will always produce either a 0 or 1 for the probability and never anything in between.

Model 2: Grid Search Cross Validation The second model makes the following changes compared to the first:

1. The model will use some hyperparameters built in to the `DecisionTreeClassifier` class: `criterion`, `max_depth`, `min_samples_leaf`, and `class_weight`.
2. `GridSearchCV` is used to both test out different values of these parameters, but do 3-way cross validation at the same time.
3. In order to avoid data leakage, I use a pipeline that performs all the preprocessing prior to each cross validation. This does make the process take quite a bit longer because every step of the process is done for every model.

```
In [357]: # Decision tree pipeline with GridSearchCV
dtc = DecisionTreeClassifier()

param_grid_dtc = {
    'dtc__criterion': ['gini', 'entropy'],
    'dtc__max_depth': [3, 4, 5, None],
    'dtc__min_samples_leaf': [1, 2, 3, 4],
    'dtc__class_weight': [None, {0: 2, 1: 1}, {0: 3, 1: 2}]
}

pipe = Pipeline(steps=[
    ('scaler', ScaleByPosition()),
    ('imputer', ImputeByPosition()),
    ('dummies', FunctionTransformer(pd.get_dummies, kw_args={'columns': ['Position', 'school_group']})),
    ('dtc', dtc)
])

dtc_gridcv = GridSearchCV(pipe, param_grid=param_grid_dtc, cv=3)

X = df_combine[['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Shuttle', 'BMI', 'Position', 'school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

dtc_gridcv.fit(X_train, y_train)

best_parameters = dtc_gridcv.best_params_

print('Grid Search found the following optimal parameters: ')
for param_name in sorted(best_parameters.keys()):
    print('%s: %r' % (param_name, best_parameters[param_name]))

y_train_pred = dtc_gridcv.predict(X_train)
y_test_pred = dtc_gridcv.predict(X_test)

display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

Grid Search found the following optimal parameters:

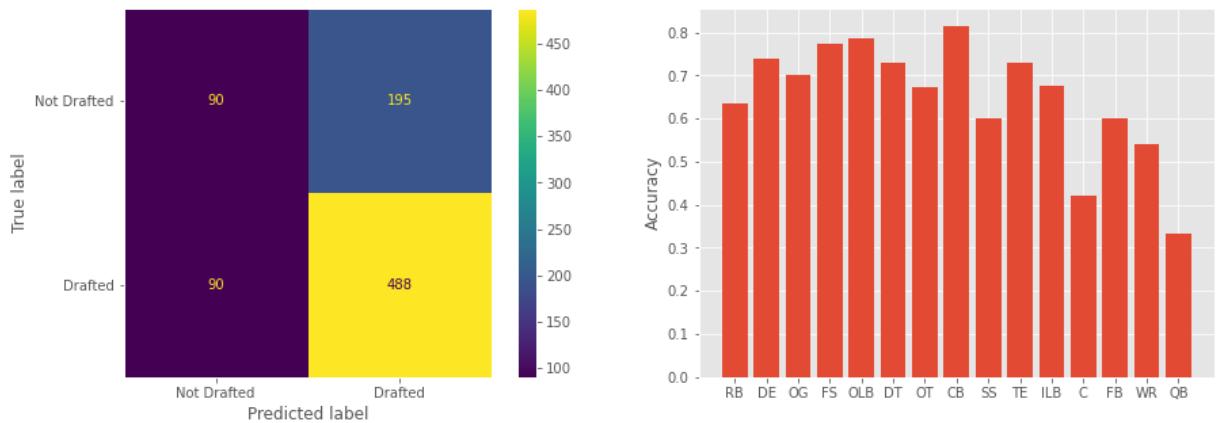
dtc__class_weight: None
dtc__criterion: 'entropy'
dtc__max_depth: 3
dtc__min_samples_leaf: 1

Training:

	precision	recall	f1-score	support
0	0.58	0.38	0.46	808
1	0.76	0.88	0.81	1780
accuracy			0.72	2588
macro avg	0.67	0.63	0.64	2588
weighted avg	0.70	0.72	0.70	2588

Test:

	precision	recall	f1-score	support
0	0.50	0.32	0.39	285
1	0.71	0.84	0.77	578
accuracy			0.67	863
macro avg	0.61	0.58	0.58	863
weighted avg	0.64	0.67	0.65	863



The overfitting issue has been greatly alleviated, though not entirely eliminated.

The overall accuracy on the test data was only 67%, not quite as good as the other types of models I have tried. It also struggles with recall of undrafted players, just like the other models.

```
In [358]: # Update ROC Curve
y_test_pred_proba = dtc_gridcv.predict_proba(X_test)

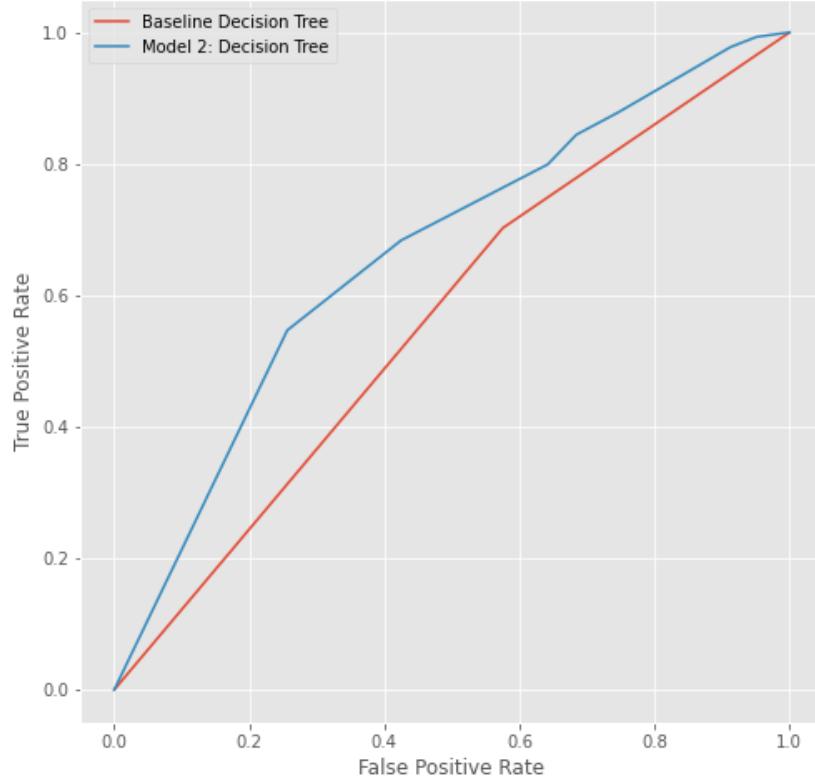
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_proba[:,1])
print('AUC:',auc(fpr, tpr))

ax_roc.plot(fpr, tpr, label='Model 2: Decision Tree')
ax_roc.legend()

fig_roc
```

AUC: 0.66395313543374

Out[358]:



Model 3: Interaction Terms

Next, I try using the interaction terms like I did with logistic regression.

```
In [359]: # Decision tree pipeline with GridSearchCV
dtc = DecisionTreeClassifier()

param_grid_dtc = {
    'dtc_criterion': ['gini', 'entropy'],
    'dtc_max_depth': [3, 4, 5, None],
    'dtc_min_samples_leaf': [1, 2, 3, 4],
    'dtc_class_weight': [None, {0: 2, 1: 1}, {0: 3, 1: 2}]
}

list1 = ['Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Agility']

pipe = Pipeline(steps=[
    ('scaler', ScaleByPosition()),
    ('imputer', ImputeByPosition()),
    ('interterms', FunctionTransformer(create_interaction_terms, kw_args={'list1':list1})),
    ('dummies', FunctionTransformer(pd.get_dummies, kw_args={'columns': ['Position', 'school_group']})),
    ('dtc', dtc)
])

dtc_gridcv = GridSearchCV(pipe, param_grid=param_grid_dtc, cv=3)

X = df_combine[['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump',
                 'Shuttle', 'BMI', 'Position', 'school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

dtc_gridcv.fit(X_train, y_train)

best_parameters = dtc_gridcv.best_params_

print('Grid Search found the following optimal parameters: ')
for param_name in sorted(best_parameters.keys()):
    print('%s: %r' % (param_name, best_parameters[param_name]))

y_train_pred = dtc_gridcv.predict(X_train)
y_test_pred = dtc_gridcv.predict(X_test)

display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

Grid Search found the following optimal parameters:

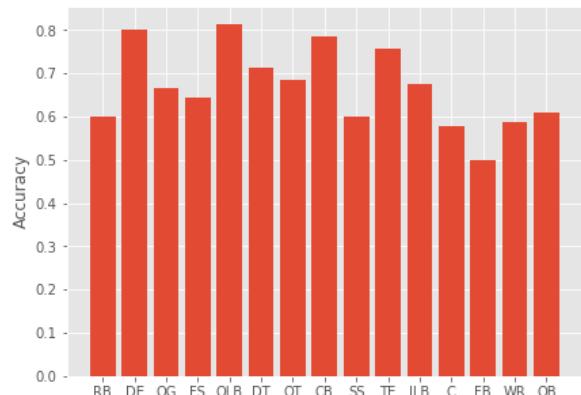
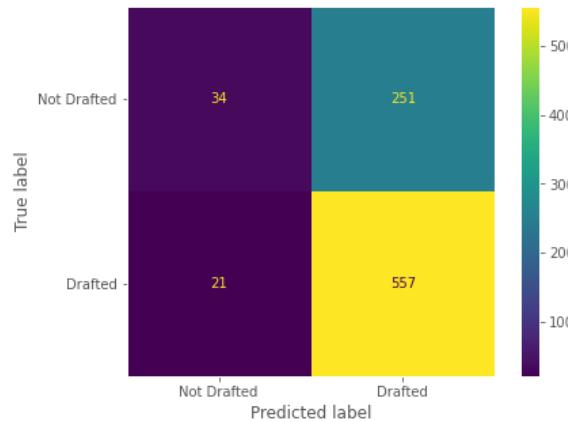
```
dtc_class_weight: None
dtc_criterion: 'gini'
dtc_max_depth: 4
dtc_min_samples_leaf: 4
```

Training:

	precision	recall	f1-score	support
0	0.71	0.15	0.25	808
1	0.72	0.97	0.82	1780
accuracy			0.72	2588
macro avg	0.71	0.56	0.54	2588
weighted avg	0.71	0.72	0.64	2588

Test:

	precision	recall	f1-score	support
0	0.62	0.12	0.20	285
1	0.69	0.96	0.80	578
accuracy			0.68	863
macro avg	0.65	0.54	0.50	863
weighted avg	0.67	0.68	0.60	863



This time around, adding interaction terms seems to have improved the fit and reduced overfitting. However, the performance is still worse than the best logistic regression model.

```
In [360]: # Update ROC Curve
y_test_pred_prob = dtc_gridcv.predict_proba(X_test)

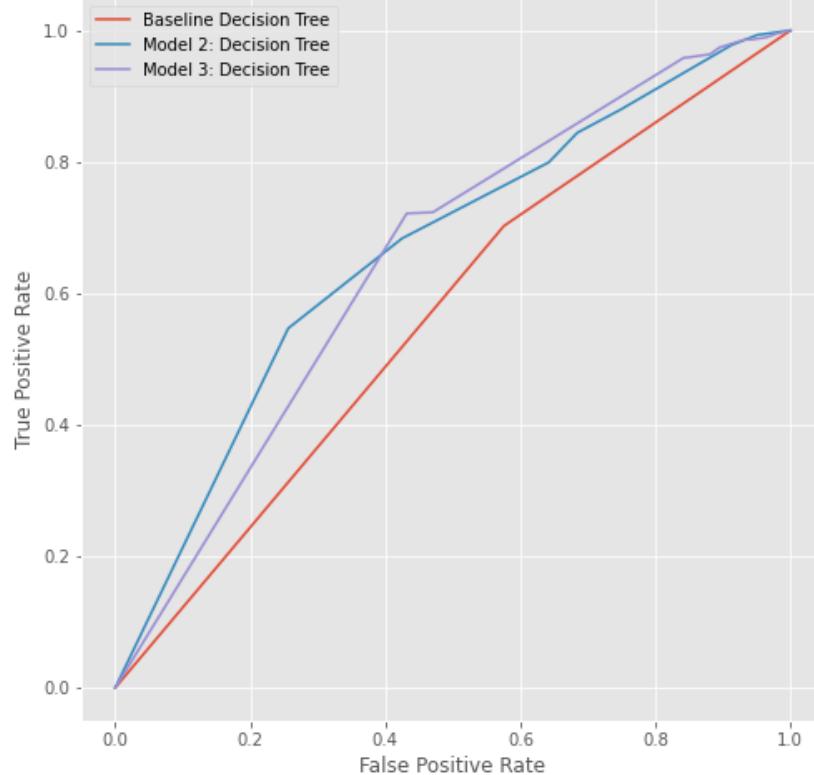
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])
print('AUC:',auc(fpr, tpr))

ax_roc.plot(fpr, tpr, label='Model 3: Decision Tree')
ax_roc.legend()

fig_roc
```

AUC: 0.6509045104109755

Out[360]:



Evaluating the Decision Tree Classifiers

These models did not achieve the same level of accuracy as the logistic regression models.

The models with and without interaction terms had very similar performance, both in terms of accuracy and AUC.

	Description	Training Accuracy	Test Accuracy	ROC AUC
	Default Hyperparameters	100% (Overfitting)	61%	0.563
	No interaction terms, optimal hyperparameters		72%	0.664
	Uses Interaction terms, optimal hyperparameters		72%	0.651

K Nearest Neighbors Classifier

Model 1: Baseline K Nearest Neighbors Classifier

K nearest neighbors needs missing values replaced and scaling of parameters.

The first model uses the default hyperparameters.

```
In [361]: # Define X and y
X = df_combine[['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Shuttle', 'BMI', 'Position', 'school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

# Apply scaler
scaler = ScaleByPosition()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

# Apply imputer
imputer = ImputeByPosition()
imputer.fit(X_train)
X_train = imputer.transform(X_train)
X_test = imputer.transform(X_test)

# Apply get_dummies to separate categorical data into columns
X_train = pd.get_dummies(X_train, columns=['Position', 'school_group'])
X_test = pd.get_dummies(X_test, columns=['Position', 'school_group'])
```

```
In [362]: # Instantiate KNeighborsClassifier
knn = KNeighborsClassifier()

# Fit model
knn.fit(X_train, y_train)

# Get predictions
y_train_pred = knn.predict(X_train)
y_test_pred = knn.predict(X_test)

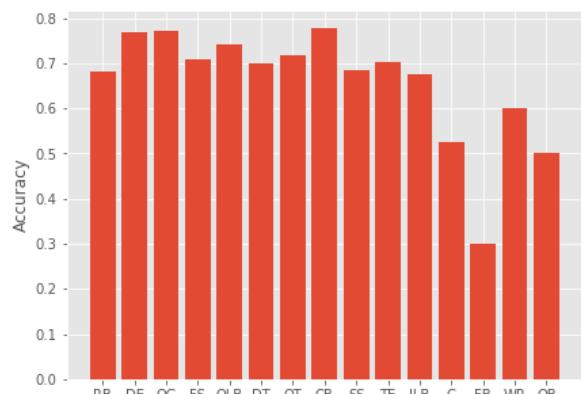
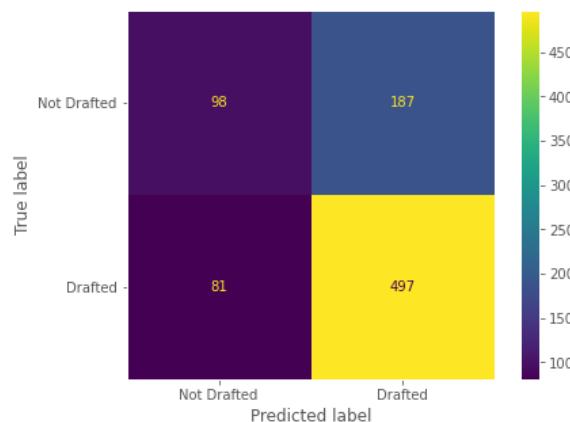
display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

Training:

	precision	recall	f1-score	support
0	0.72	0.49	0.58	808
1	0.80	0.91	0.85	1780
accuracy			0.78	2588
macro avg	0.76	0.70	0.72	2588
weighted avg	0.77	0.78	0.77	2588

Test:

	precision	recall	f1-score	support
0	0.55	0.34	0.42	285
1	0.73	0.86	0.79	578
accuracy			0.69	863
macro avg	0.64	0.60	0.61	863
weighted avg	0.67	0.69	0.67	863



The baseline model achieves a 69% accuracy on the test data.

```
In [363]: # Plot ROC curve
y_test_pred_prob = knn.predict_proba(X_test)
fig_roc, ax_roc = plt.subplots(figsize=(8,8))

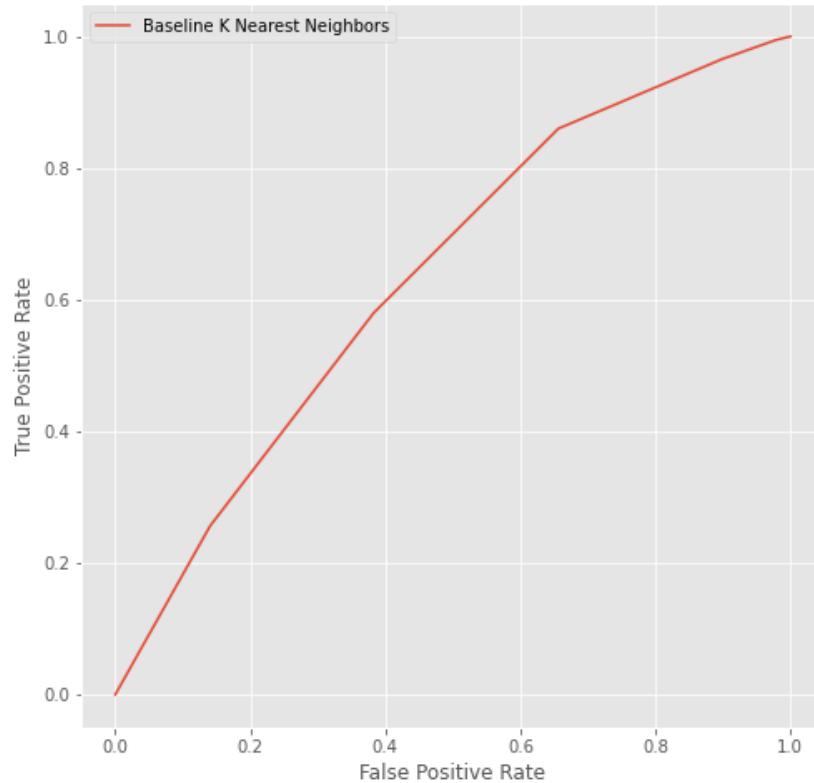
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])
print('AUC:',auc(fpr, tpr))

ax_roc.plot(fpr, tpr, label='Baseline K Nearest Neighbors')

ax_roc.set_xlabel('False Positive Rate')
ax_roc.set_ylabel('True Positive Rate')
ax_roc.legend()
```

AUC: 0.6371486675165421

Out[363]: <matplotlib.legend.Legend at 0x13cb65f7d30>



The baseline model is not very impressive. Let's try tweaking some things about the model.

Model 2: Separating the Positions

One of the things I have tried to achieve with other models is separating the different positions. I can do that with a K nearest neighbors model by increasing the values of the one-hot-encoded position columns. Below, I multiply those columns by 10 so that the model should not consider players of different positions to be neighbors (unless there are fewer than n_neighbors of a given position). I do not actually expect this to make a huge difference, but it is worth trying.

```
In [364]: # Define X and y
X = df_combine[['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Shuttle', 'BMI', 'Position', 'school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

# Apply scaler
scaler = ScaleByPosition()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

# Apply imputer
imputer = ImputeByPosition()
imputer.fit(X_train)
X_train = imputer.transform(X_train)
X_test = imputer.transform(X_test)

# Apply get_dummies to separate categorical data into columns
X_train = pd.get_dummies(X_train, columns=['Position', 'school_group'])
X_test = pd.get_dummies(X_test, columns=['Position', 'school_group'])

# Separate positions further so they won't be neighbors
for col in X_train.columns:
    if 'Position' in col:
        X_train[col] = X_train[col].apply(lambda x: 10*x)
        X_test[col] = X_test[col].apply(lambda x: 10*x)
```

```
In [365]: # Instantiate KNeighborsClassifier
knn = KNeighborsClassifier()

# Fit model
knn.fit(X_train, y_train)

# Get predictions
y_train_pred = knn.predict(X_train)
y_test_pred = knn.predict(X_test)

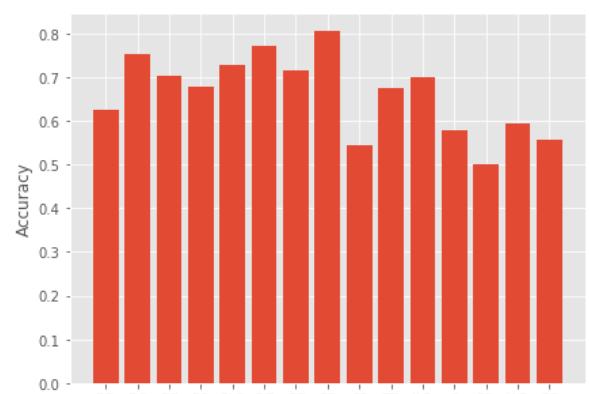
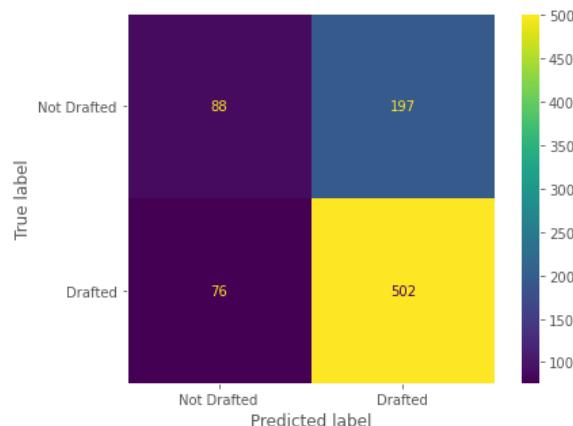
display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

Training:

	precision	recall	f1-score	support
0	0.72	0.47	0.57	808
1	0.79	0.92	0.85	1780
accuracy			0.78	2588
macro avg	0.76	0.69	0.71	2588
weighted avg	0.77	0.78	0.76	2588

Test:

	precision	recall	f1-score	support
0	0.54	0.31	0.39	285
1	0.72	0.87	0.79	578
accuracy			0.68	863
macro avg	0.63	0.59	0.59	863
weighted avg	0.66	0.68	0.66	863

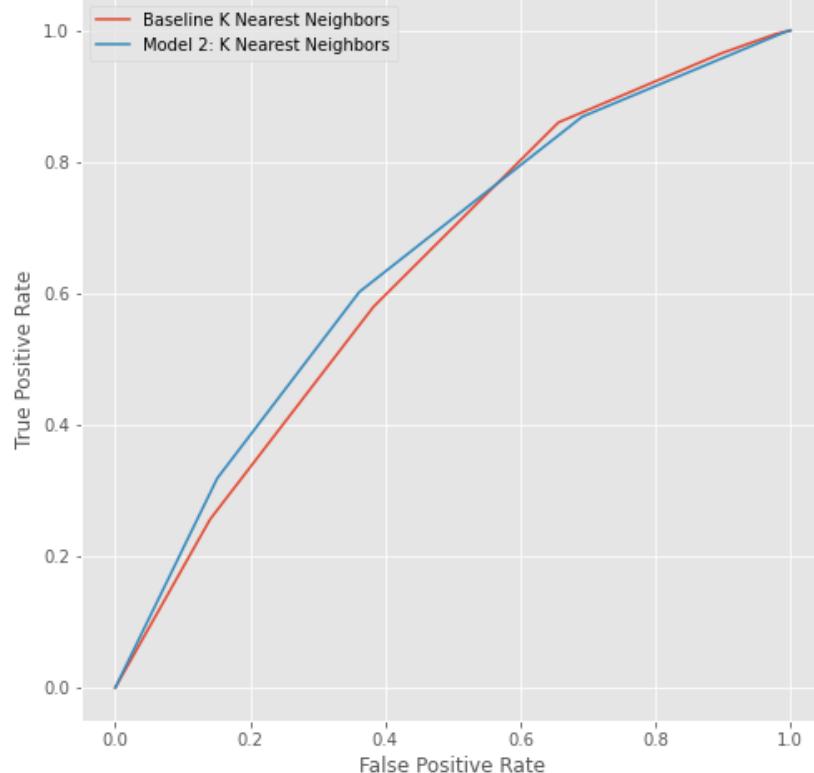


This model is slightly worse than the first. It seems that separating the positions did not improve the fit.

```
In [366]: # Update ROC Curve  
y_test_pred_prob = knn.predict_proba(X_test)  
  
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])  
print('AUC:', auc(fpr, tpr))  
  
ax_roc.plot(fpr, tpr, label='Model 2: K Nearest Neighbors')  
ax_roc.legend()  
  
fig_roc
```

AUC: 0.6520548776786256

Out[366]:



Model 3: GridSearchCV

The model below adjusts two hyperparameters:

1. n_neighbors: number of neighbors to use when making predictions
2. p: exponent of the Minkowski metric to use.

Once again, I use a pipeline to make sure that there is no data leakage.

```
In [367]: # K Nearest neighbors classifier pipeline with GridSearchCV
knn = KNeighborsClassifier()

param_grid_knn = {
    'knn__n_neighbors': [9, 25, 49, 81],
    'knn__p': [1, 1.5, 2, 2.5, 3]
}

pipe = Pipeline(steps=[
    ('scaler', ScaleByPosition()),
    ('imputer', ImputeByPosition()),
    ('dummies', FunctionTransformer(pd.get_dummies, kw_args={'columns': ['Position', 'school_group']}),
    ('knn', knn)
])

knn_gridcv = GridSearchCV(pipe, param_grid=param_grid_knn, cv=3)

X = df_combine[['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Shuttle', 'BMI', 'Position', 'school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

knn_gridcv.fit(X_train, y_train)

best_parameters = knn_gridcv.best_params_

print('Grid Search found the following optimal parameters: ')
for param_name in sorted(best_parameters.keys()):
    print('%s: %r' % (param_name, best_parameters[param_name]))

y_train_pred = knn_gridcv.predict(X_train)
y_test_pred = knn_gridcv.predict(X_test)

display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

Grid Search found the following optimal parameters:

knn_n_neighbors: 49

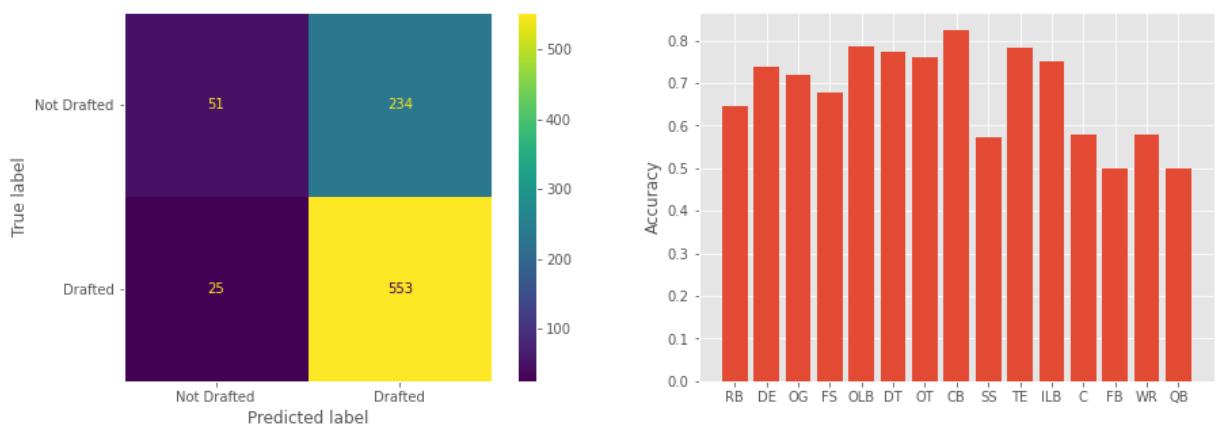
knn_p: 2.5

Training:

	precision	recall	f1-score	support
0	0.74	0.21	0.32	808
1	0.73	0.97	0.83	1780
accuracy			0.73	2588
macro avg	0.74	0.59	0.58	2588
weighted avg	0.73	0.73	0.67	2588

Test:

	precision	recall	f1-score	support
0	0.67	0.18	0.28	285
1	0.70	0.96	0.81	578
accuracy			0.70	863
macro avg	0.69	0.57	0.55	863
weighted avg	0.69	0.70	0.64	863



The best result used the 49 nearest neighbors and p=2.5.

The result was not as good as previous models and really struggled with recall of undrafted players.

```
In [368]: # Update ROC Curve
y_test_pred_prob = knn_gridcv.predict_proba(X_test)

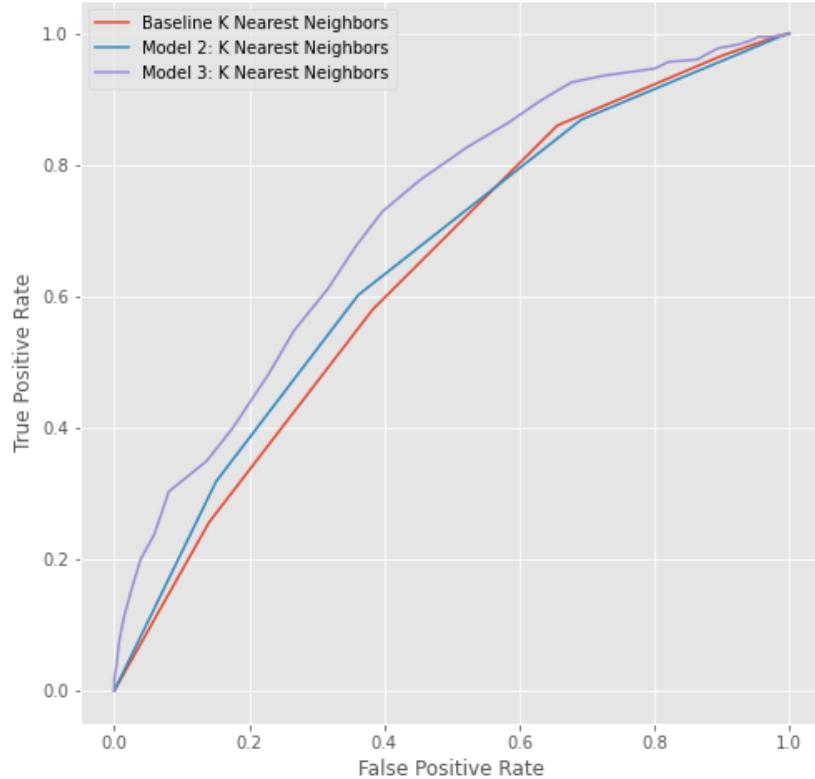
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])
print('AUC:',auc(fpr, tpr))

ax_roc.plot(fpr, tpr, label='Model 3: K Nearest Neighbors')
ax_roc.legend()

fig_roc
```

AUC: 0.7175651065379711

Out[368]:



This model was a clear improvement over the other K Nearest Neighbors models, but it still did not perform as well as the logistic regression models.

Evaluating the K Nearest Neighbors Models

The best performing model used k=49 and p=2.5. The performance was still a bit below that of the best logistic regression model and there is still some evidence of overfitting, even in the best model.

Description	k	p	Training Accuracy	Test Accuracy	ROC AUC
Default Hyperparameters	5	2	78%	69%	0.637
Positions separated by 10x	5	2	78%	68%	0.652
Uses Interaction terms, optimal hyperparameters	49	2.5	73%	70%	0.717

Naive Bayes Classifier

Scikit Learn does have a class for Naive Bayes models, but I decided to write my own from scratch for three reasons:

1. It would be significantly easier getting the model to treat positions the way I want it to.
2. Technically, a Naive Bayes model shouldn't mind missing data, but online searches indicate that sklearn's Naive Bayes does.
3. It is good practice.

The "Naive" in Naive Bayes refers to the assumption that all of the independent columns are independent of one another. In reality, that tends to not actually be true and, realistically, it is not true in this case either. Players with faster 40-yard sprints tend to also have higher vertical jumps, but a lower number of bench press reps. However, the model should still be able to work reasonably well without taking into account these correlations.

The one way in which I do not want the model to be naive is by assuming everything is independent of a player's position. I have written my classifier so that the distributions depend on the position. This is part of the reason why I chose to write my own classifier since I wanted it to incorporate dependence on something other than the target variable.

Below is the class that I wrote. It can actually be run without taking position into account (if you want) and you can specify which columns are categorical (those columns don't even need to be turned into numbers).

This model does not require imputing or scaling of data. It is fine with missing values (it just likely won't perform as well on input that is missing data).

The predict method has one parameter called "bias". This bias indicates how much the model will lean towards predicting 1 rather than 0 (the model only works with binary classes as written). A bias of 1 means there is no bias, above 1 means it will predict 1 more often, and below 1 means it will predict 1 less often. It has an option called 'proportional' which automatically accounts for the class imbalance in the training data.


```
In [369]: class NaiveBayes:
    def __init__(self, split_by_position=None, categorical_cols=None):
        """
        Arguments:
        split_by_position: List of column names where the probabilities should be calculated
        for each position separately.

        categorical_thresh: Indicates which columns are categorical.
        """
        self.split_by_position = split_by_position
        self.categorical_cols = categorical_cols

    def fit(self, X, y):
        """
        Inputs:
        X: pandas dataframe of independent variables
        y: target variable
        """
        self.X = X
        self.y = y

    def predict_proba(self, X):
        """
        Calculates the probability that a player will be drafted (or not).
        Input:
        X: pandas DataFrame that matches the format of DataFrame used for fitting.
        """

        proba = []
        for i in X.index:
            logprob0 = np.log(sum(self.y==0)/len(self.y))
            logprob1 = np.log(sum(self.y==1)/len(self.y))

            for col in self.X.columns:

                if pd.isna(X.loc[i,col])==False:

                    if col in self.categorical_cols:

                        if col in self.split_by_position:
                            count0 = sum(self.y[(self.X[col]==X.loc[i,col]) & (self.X['Position']==0)])
                            count1 = sum(self.y[(self.X[col]==X.loc[i,col]) & (self.X['Position']==1)])
                            logprob0 += np.log(count0/(count0+count1))
                            logprob1 += np.log(count1/(count0+count1))

                        else:
                            count0 = sum(self.y[self.X[col]==X.loc[i,col]]==0)
                            count1 = sum(self.y[self.X[col]==X.loc[i,col]]==1)
                            logprob0 += np.log(count0/(count0+count1))
                            logprob1 += np.log(count1/(count0+count1))

                    else:

                        if col in self.split_by_position:
                            mean0 = np.mean(self.X[(self.X['Position']==0)==X.loc[i,'Position']])
                            mean1 = np.mean(self.X[(self.X['Position']==1)==X.loc[i,'Position']])
                            std0 = np.std(self.X[(self.X['Position']==0)==X.loc[i,'Position']])
                            std1 = np.std(self.X[(self.X['Position']==1)==X.loc[i,'Position']])

                            mean0 = np.mean(self.X[self.y==0][col])
                            mean1 = np.mean(self.X[self.y==1][col])
                            std0 = np.std(self.X[self.y==0][col], ddof=1)
                            std1 = np.std(self.X[self.y==1][col], ddof=1)

                            logprob0 += np.log(st.norm.pdf(X.loc[i,col], mean0, std0))
                            logprob1 += np.log(st.norm.pdf(X.loc[i,col], mean1, std1))

                        else:
```

```
proba.append([np.exp(logprob0)/(np.exp(logprob0)+np.exp(logprob1)),  
             np.exp(logprob1)/(np.exp(logprob0)+np.exp(logprob1))])  
  
return np.array(proba)  
  
def predict(self, X, bias=1.0):  
    """  
        Predict whether a player will be drafted or not.  
        Input:  
        X: pandas DataFrame that matches the format of DataFrame used for fitting.  
        bias: Number that indicates by how much to bias towards a positive prediction. Default is  
        If bias is set to 'proportional', then the relative proportion of the target class is used  
    """  
    if bias == 'proportional': bias = sum(self.y==0)/sum(self.y==1)  
  
    proba = self.predict_proba(X)  
  
    pred = [1 if x[0]/x[1] < bias else 0 for x in proba]  
  
    return pd.Series(pred, index=X.index)
```

Model 1: Baseline Model with no bias The data fed to the Naive Bayes will not be imputed (missing values will remain missing) and it will not be rescaled (it does not need to be because it accounts for position).

The baseline model will use the default bias (1, meaning no bias).

In [370]:

```
# Define X and y
X = df_combine[['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Shuttle', 'BMI', 'Position', 'school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

split_cols = ['Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Age', 'Shuttle', 'BMI']
cat_cols = ['Position', 'school_group']

nb = NaiveBayes(split_by_position=split_cols, categorical_cols=cat_cols)

nb.fit(X_train, y_train)

# Bias is 1.0 by default
y_train_pred = nb.predict(X_train)
y_test_pred = nb.predict(X_test)

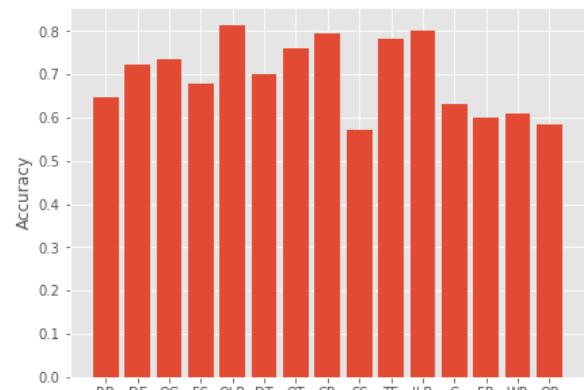
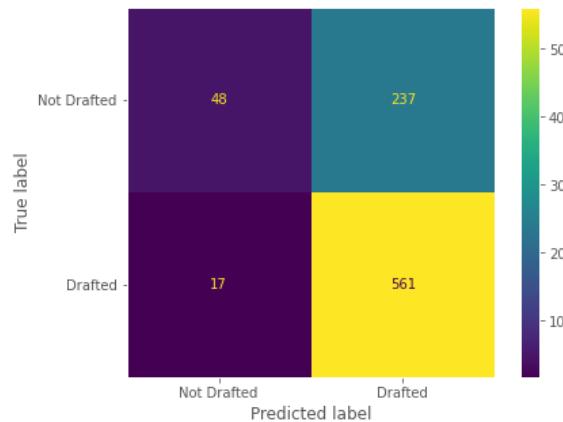
display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

Training:

	precision	recall	f1-score	support
0	0.77	0.22	0.34	808
1	0.73	0.97	0.83	1780
accuracy			0.74	2588
macro avg	0.75	0.59	0.59	2588
weighted avg	0.74	0.74	0.68	2588

Test:

	precision	recall	f1-score	support
0	0.74	0.17	0.27	285
1	0.70	0.97	0.82	578
accuracy			0.71	863
macro avg	0.72	0.57	0.54	863
weighted avg	0.71	0.71	0.64	863



So far, all of the models have overpredicted the number of drafted players, leading to low values for the recall of undrafted players. This model is especially bad in this sense with just 17% recall of undrafted players. The next model should address this to some extent.

```
In [371]: # Plot ROC curve
y_test_pred_prob = nb.predict_proba(X_test)
fig_roc, ax_roc = plt.subplots(figsize=(8,8))

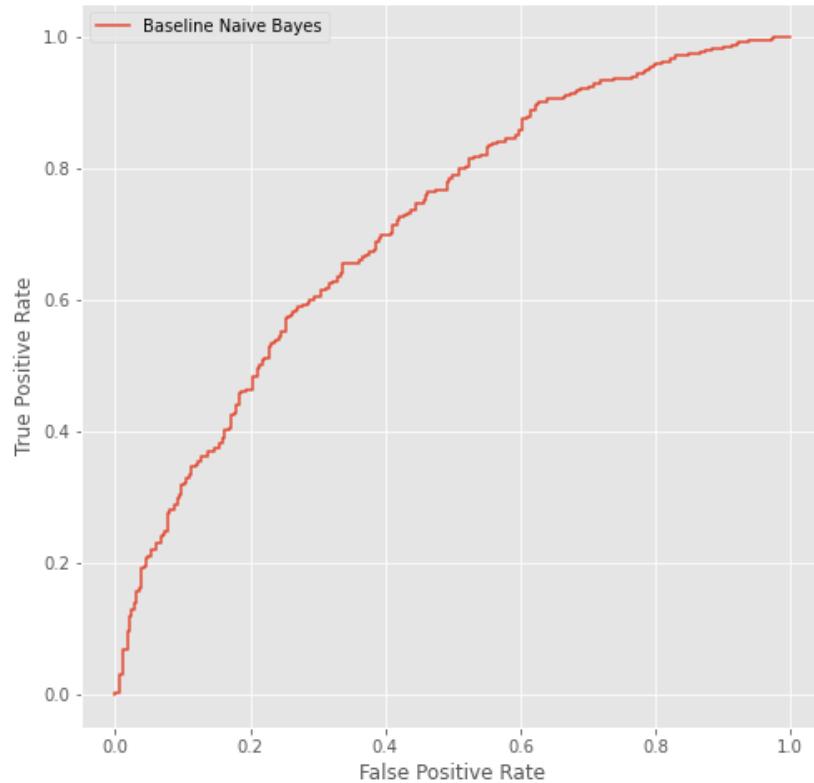
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])
print('AUC:',auc(fpr, tpr))

ax_roc.plot(fpr, tpr, label='Baseline Naive Bayes')

ax_roc.set_xlabel('False Positive Rate')
ax_roc.set_ylabel('True Positive Rate')
ax_roc.legend()
```

AUC: 0.714884963273235

Out[371]: <matplotlib.legend.Legend at 0x13cba030490>



I was curious to see what the histogram of predicted probabilities were for this model. Those histograms are below.

The model tends to predict a lot of players will be drafted with high probability. Even the distribution of undrafted players is skewed towards very high probabilities.

```
In [372]: # y_test_pred_prob = nb.predict_proba(X_test)

fig, ax = plt.subplots(figsize=(7,7))

keep = y_test==0

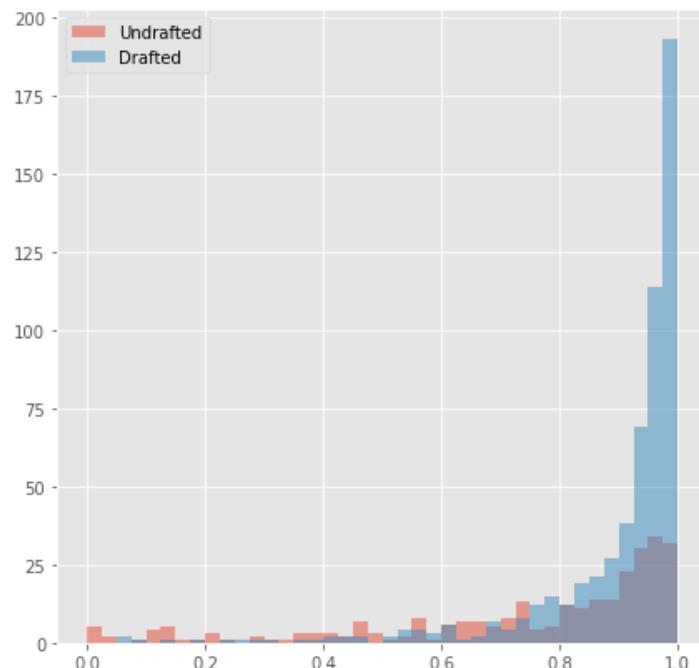
ax.hist(np.sort(y_test_pred_prob[keep][:,1]),bins=np.arange(41)/40,alpha=0.5,label='Undrafted',density=True)

keep = y_test==1

ax.hist(np.sort(y_test_pred_prob[keep][:,1]),bins=np.arange(41)/40,alpha=0.5,label='Drafted',density=True)

ax.legend()
```

Out[372]: <matplotlib.legend.Legend at 0x13cb730f8b0>



Model 2: Proportional bias

The second model will be the same as the first, except now it will use the option of making the bias proportional. Because about 1/3 of players did not get drafted, the bias will be around $(1/3)/(2/3) = 1/2$. This should fix the problem of the model predicting so many players getting drafted.

In [373]:

```
# Define X and y
X = df_combine[['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Shuttle', 'BMI', 'Position', 'school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

split_cols = ['Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Age', 'Shuttle', 'BMI']
cat_cols = ['Position', 'school_group']

nb = NaiveBayes(split_by_position=split_cols, categorical_cols=cat_cols)

nb.fit(X_train, y_train)

# Using proportional bias
y_train_pred = nb.predict(X_train, bias='proportional')
y_test_pred = nb.predict(X_test, bias='proportional')

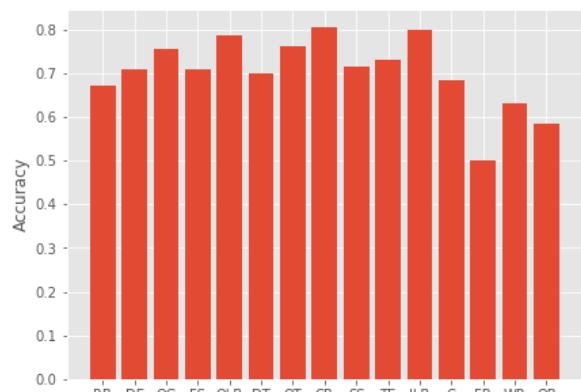
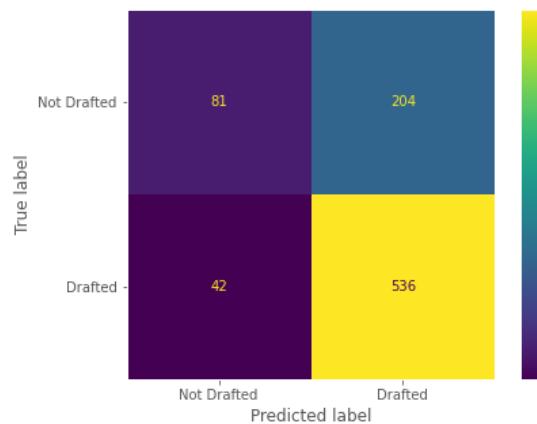
display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

Training:

	precision	recall	f1-score	support
0	0.69	0.33	0.45	808
1	0.76	0.93	0.83	1780
accuracy			0.75	2588
macro avg	0.72	0.63	0.64	2588
weighted avg	0.74	0.75	0.72	2588

Test:

	precision	recall	f1-score	support
0	0.66	0.28	0.40	285
1	0.72	0.93	0.81	578
accuracy			0.71	863
macro avg	0.69	0.61	0.61	863
weighted avg	0.70	0.71	0.68	863



```
In [374]: # Update ROC Curve
y_test_pred_prob = nb.predict_proba(X_test)

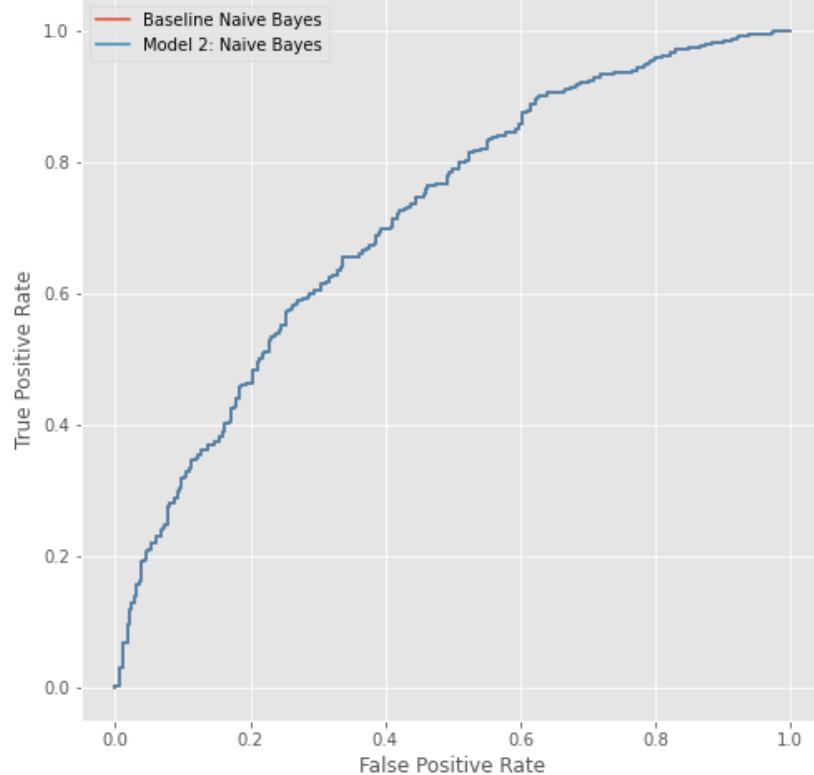
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])
print('AUC:', auc(fpr, tpr))

ax_roc.plot(fpr, tpr, label='Model 2: Naive Bayes')
ax_roc.legend()

fig_roc
```

AUC: 0.714884963273235

Out[374]:



The curve is precisely the same as the last model. This is because all I changed is the value that I use to make the cutoff between predicting 0 or 1, but the actual calculation of the probabilities was not changed at all.

As predicted, the recall of undrafted players did improve (17% → 28%), bringing it closer to the best logistic regression models (though still worse). The overall accuracy is actually the same as the previous Naive Bayes model.

Naive Bayes result I could continue tweaking the bias parameter to see if it helps, but the precision and recall values are all at or below those of the best logistic regression model. This means that the Naive Bayes model will not outperform the logistic regression no matter what bias I use.

Evaluating the Naive Bayes Models

The Naive Bayes models each achieved accuracies of 71% on the test data. The performance was not as good as the best model so far.

Description	bias	Training Accuracy	Test Accuracy	ROC AUC
Default Hyperparameters	default	74%	71%	0.715
Use balanced bias	balanced	75%	71%	0.715

Random Forest Classifier

Model 1: Baseline Model The baseline random forest classifier will use the default hyperparameter values and it will use the same preprocessing as the decision tree models.

```
In [375]: # Define X and y
X = df_combine[['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Shuttle', 'BMI', 'Position', 'school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

# Apply scaler
scaler = ScaleByPosition()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

# Apply imputer
imputer = ImputeByPosition()
imputer.fit(X_train)
X_train = imputer.transform(X_train)
X_test = imputer.transform(X_test)

# Apply get_dummies to separate categorical data into columns
X_train = pd.get_dummies(X_train, columns=['Position', 'school_group'])
X_test = pd.get_dummies(X_test, columns=['Position', 'school_group'])
```

In [376]:

```
# Instantiate RandomForestClassifier
rfc = RandomForestClassifier(random_state=42)

# Fit model
rfc.fit(X_train, y_train)

# Get predictions
y_train_pred = rfc.predict(X_train)
y_test_pred = rfc.predict(X_test)

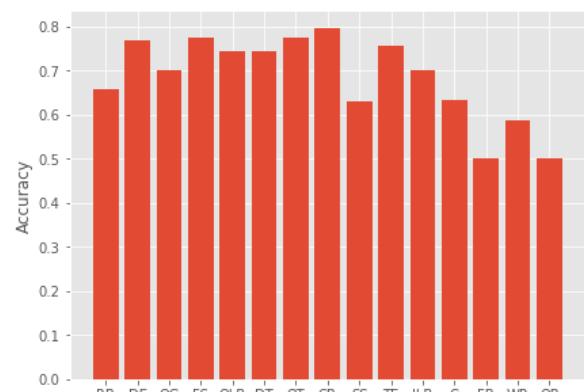
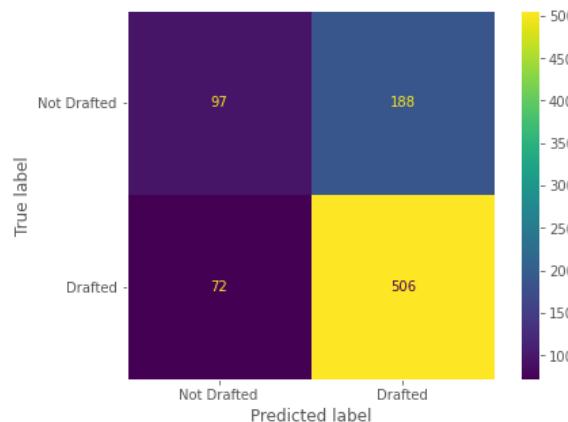
display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

Training:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	808
1	1.00	1.00	1.00	1780
accuracy			1.00	2588
macro avg	1.00	1.00	1.00	2588
weighted avg	1.00	1.00	1.00	2588

Test:

	precision	recall	f1-score	support
0	0.57	0.34	0.43	285
1	0.73	0.88	0.80	578
accuracy			0.70	863
macro avg	0.65	0.61	0.61	863
weighted avg	0.68	0.70	0.67	863



The fit to the training data is "perfect", indicating overfitting. We need to adjust some hyperparameters.

The fit to the test data was not that bad considering the overfitting. Let's see if it improves if we adjust the model.

```
In [377]: # Plot ROC curve
y_test_pred_prob = rfc.predict_proba(X_test)
fig_roc, ax_roc = plt.subplots(figsize=(8,8))

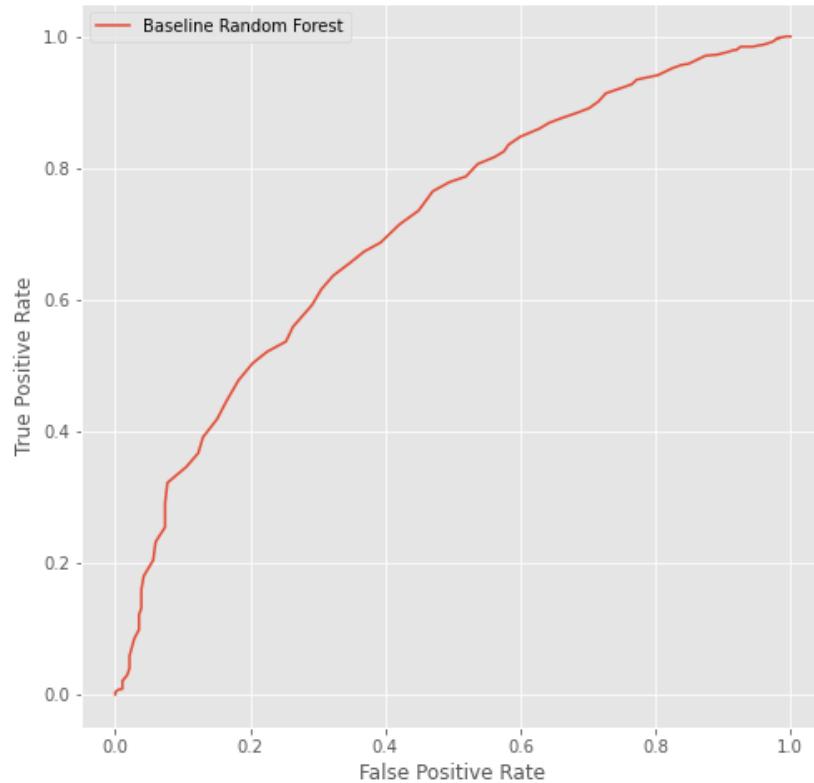
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])
print('AUC:',auc(fpr, tpr))

ax_roc.plot(fpr, tpr, label='Baseline Random Forest')

ax_roc.set_xlabel('False Positive Rate')
ax_roc.set_ylabel('True Positive Rate')
ax_roc.legend()
```

AUC: 0.709142232744491

Out[377]: <matplotlib.legend.Legend at 0x13ca97bf9a0>



Model 2: GridSearchCV

The second model varies some hyperparameters to reduce the overfitting.

```
In [384]: # Random forest classifier pipeline with GridSearchCV
rfc = RandomForestClassifier(random_state=42)

param_grid_rfc = {
    'rfc__n_estimators': [150, 200],
    'rfc__max_depth':[4,6,8],
    'rfc__criterion': ['gini','entropy'],
    'rfc__min_samples_leaf': [1,2,3,4],
    'rfc__class_weight': [None, {0:3, 1:2}, {0:2, 1:1}]
}

pipe = Pipeline(steps=[
    ('scaler',ScaleByPosition()),
    ('imputer',ImputeByPosition()),
    ('dummies',FunctionTransformer(pd.get_dummies,kw_args={'columns':['Position','school_group']}),
    ('rfc',rfc)
])

rfc_gridcv = GridSearchCV(pipe, param_grid=param_grid_rfc, cv=3)

X = df_combine[['Age','Height','Weight','Sprint_40yd','Vertical_Jump','Bench_Press_Reps','Broad_Jump','Shuttle','BMI','Position','school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

rfc_gridcv.fit(X_train, y_train)

best_parameters = rfc_gridcv.best_params_

print('Grid Search found the following optimal parameters: ')
for param_name in sorted(best_parameters.keys()):
    print('%s: %r' % (param_name, best_parameters[param_name]))

y_train_pred = rfc_gridcv.predict(X_train)
y_test_pred = rfc_gridcv.predict(X_test)

display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

Grid Search found the following optimal parameters:

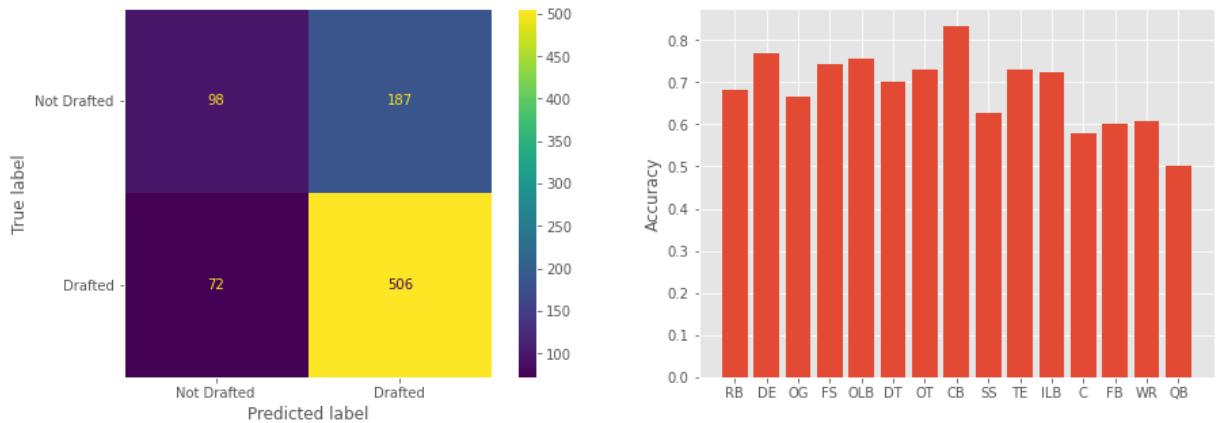
rfc__class_weight: {0: 3, 1: 2}
rfc__criterion: 'entropy'
rfc__max_depth: 6
rfc__min_samples_leaf: 2
rfc__n_estimators: 150

Training:

	precision	recall	f1-score	support
0	0.73	0.50	0.59	808
1	0.80	0.92	0.86	1780
accuracy			0.79	2588
macro avg	0.77	0.71	0.72	2588
weighted avg	0.78	0.79	0.77	2588

Test:

	precision	recall	f1-score	support
0	0.58	0.34	0.43	285
1	0.73	0.88	0.80	578
accuracy			0.70	863
macro avg	0.65	0.61	0.61	863
weighted avg	0.68	0.70	0.68	863



The model achieves an accuracy of 70%. The overfitting has been reduced, but not eliminated.

```
In [385]: # Update ROC Curve
y_test_pred_prob = rfc_gridcv.predict_proba(X_test)

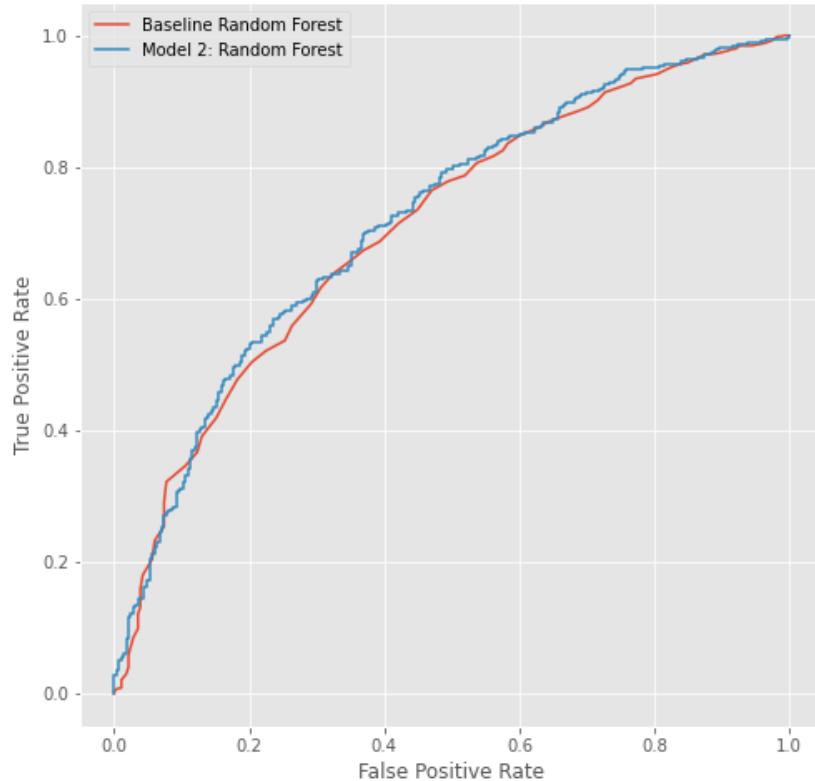
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])
print('AUC:',auc(fpr, tpr))

ax_roc.plot(fpr, tpr, label='Model 2: Random Forest')
ax_roc.legend()

fig_roc
```

AUC: 0.7197231833910035

Out[385]:



Model 3: Interaction Terms


```
In [395]: # Random forest classifier pipeline with GridSearchCV
rfc = RandomForestClassifier(random_state=42)

param_grid_rfc = {
    'rfc__n_estimators': [100, 150, 200],
    'rfc__max_depth':[5,7,9],
    'rfc__criterion': ['gini','entropy'],
    'rfc__min_samples_leaf': [4,6,8],
    'rfc__class_weight': [None, {0:3, 1:2}, {0:2, 1:1}]
}

list1 = ['Height','Weight','Sprint_40yd','Vertical_Jump','Bench_Press_Reps','Broad_Jump','Agility']

pipe = Pipeline(steps=[
    ('scaler',ScaleByPosition()),
    ('imputer',ImputeByPosition()),
    ('interterms',FunctionTransformer(create_interaction_terms, kw_args={'list1':list1})),
    ('dummies',FunctionTransformer(pd.get_dummies,kw_args={'columns':['Position','school_group']}),
    ('rfc',rfc)
])

rfc_gridcv = GridSearchCV(pipe, param_grid=param_grid_rfc, cv=3)

X = df_combine[['Age','Height','Weight','Sprint_40yd','Vertical_Jump','Bench_Press_Reps','Broad_Jump',
                 'Shuttle','BMI','Position','school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

rfc_gridcv.fit(X_train, y_train)

best_parameters = rfc_gridcv.best_params_

print('Grid Search found the following optimal parameters: ')
for param_name in sorted(best_parameters.keys()):
    print('%s: %r' % (param_name, best_parameters[param_name]))

y_train_pred = rfc_gridcv.predict(X_train)
y_test_pred = rfc_gridcv.predict(X_test)

display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

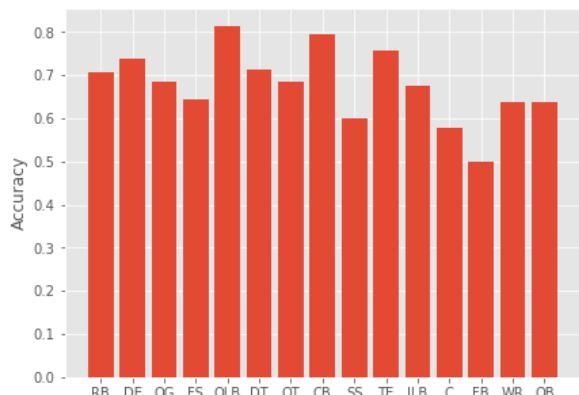
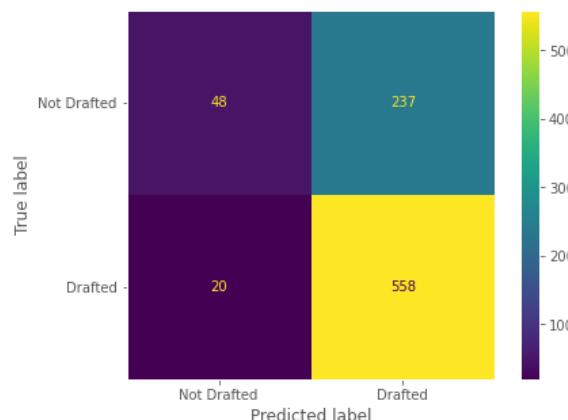
Grid Search found the following optimal parameters:

```
rfc_class_weight: {0: 3, 1: 2}
rfc_criterion: 'gini'
rfc_max_depth: 7
rfc_min_samples_leaf: 6
rfc_n_estimators: 150
Training:
```

	precision	recall	f1-score	support
0	0.81	0.17	0.28	808
1	0.72	0.98	0.83	1780
accuracy			0.73	2588
macro avg	0.77	0.57	0.55	2588
weighted avg	0.75	0.73	0.66	2588

Test:

	precision	recall	f1-score	support
0	0.71	0.17	0.27	285
1	0.70	0.97	0.81	578
accuracy			0.70	863
macro avg	0.70	0.57	0.54	863
weighted avg	0.70	0.70	0.63	863



The introduction of interaction terms did not significantly change the success of the model. It still does not match the logistic regression models.

```
In [396]: # Update ROC Curve
y_test_pred_prob = rfc_gridcv.predict_proba(X_test)

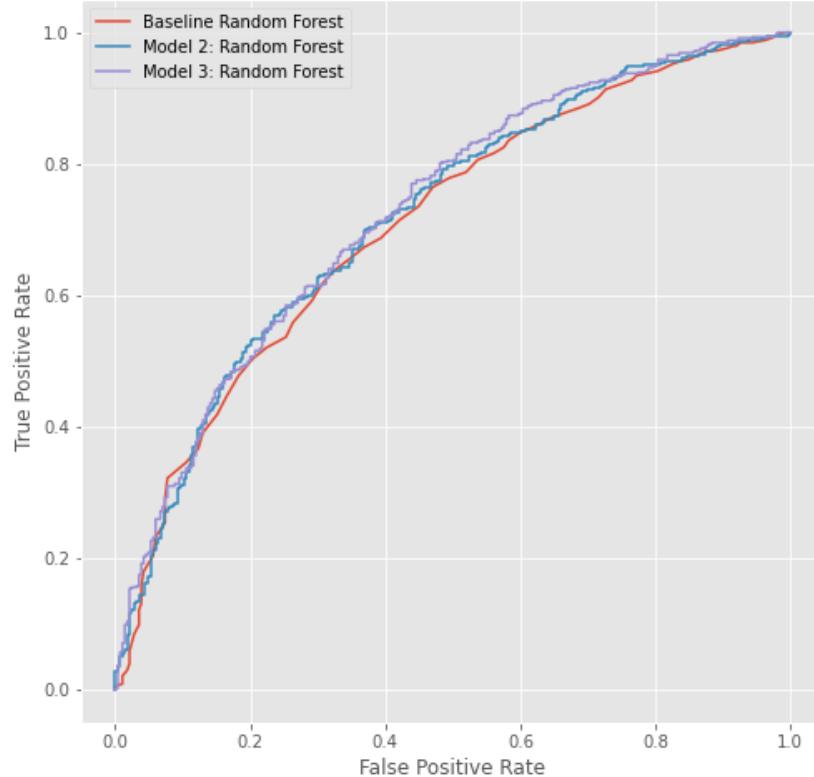
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])
print('AUC:',auc(fpr, tpr))

ax_roc.plot(fpr, tpr, label='Model 3: Random Forest')
ax_roc.legend()

fig_roc
```

AUC: 0.7285042190250712

Out[396]:



Evaluating the Random Forest Classifiers

Each version of the random forest classifier achieved a 70% total accuracy on the test data. None were able to match the performance of the best model so far.

Description	Training Accuracy	Test Accuracy	ROC AUC
Default Hyperparameters	100% (Overfitting)	70%	0.709
No inter terms, optimal hyperparameters		79%	0.720
Used inter terms, optimal hyperparameters		73%	0.729

XGBoost Classifier

Model 1: Baseline Since XGBoost can handle missing data, I will not impute anything for the missing data. I will still rescale values for each position separately.

```
In [397]: # Define X and y
X = df_combine[['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Shuttle', 'BMI', 'Position', 'school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

# Apply scaler (no imputing)
scaler = ScaleByPosition()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

# Apply get_dummies to separate categorical data into columns
X_train = pd.get_dummies(X_train, columns=['Position', 'school_group'])
X_test = pd.get_dummies(X_test, columns=['Position', 'school_group'])
```

```
In [398]: # Instantiate XGBCClassifier
xgb = XGBCClassifier()

# Fit model
xgb.fit(X_train, y_train)

# Get predictions
y_train_pred = xgb.predict(X_train)
y_test_pred = xgb.predict(X_test)

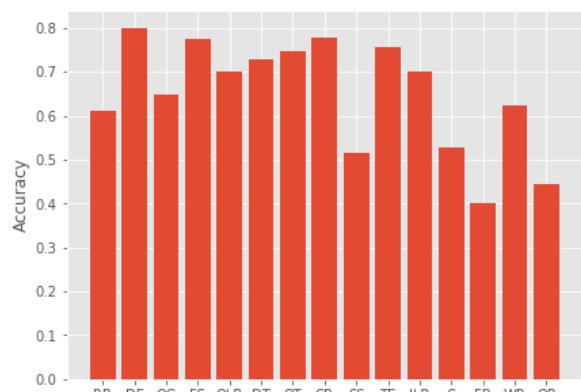
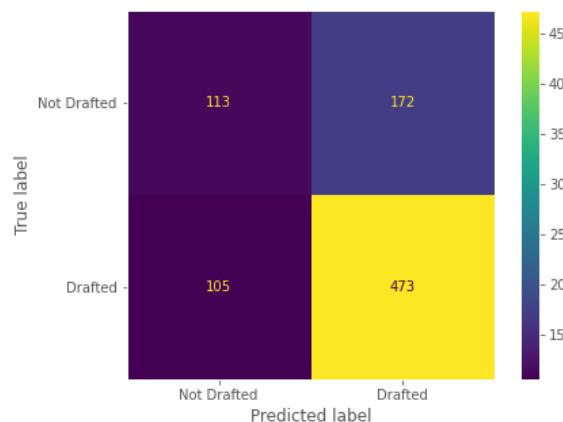
display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

Training:

	precision	recall	f1-score	support
0	1.00	0.98	0.99	808
1	0.99	1.00	1.00	1780
accuracy			0.99	2588
macro avg	1.00	0.99	0.99	2588
weighted avg	0.99	0.99	0.99	2588

Test:

	precision	recall	f1-score	support
0	0.52	0.40	0.45	285
1	0.73	0.82	0.77	578
accuracy			0.68	863
macro avg	0.63	0.61	0.61	863
weighted avg	0.66	0.68	0.67	863



The baseline model is severely overfitting. Below, I do a grid search to tune hyperparameters which should help fix this problem.

In [399]:

```
# Plot ROC curve
y_test_pred_prob = xgb.predict_proba(X_test)
fig_roc, ax_roc = plt.subplots(figsize=(8,8))

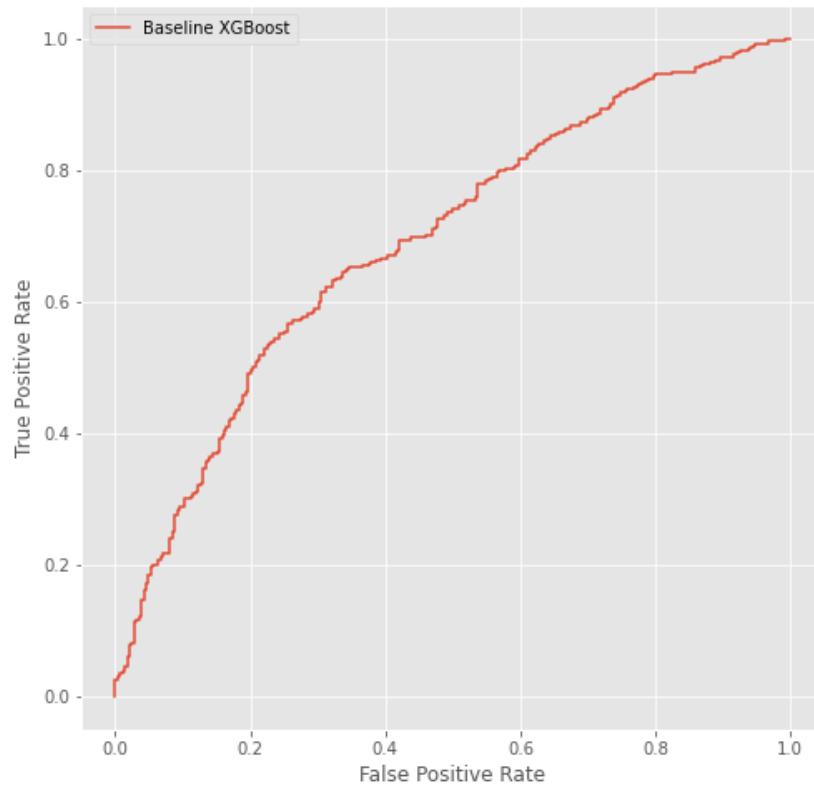
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])
print('AUC:',auc(fpr, tpr))

ax_roc.plot(fpr, tpr, label='Baseline XGBoost')

ax_roc.set_xlabel('False Positive Rate')
ax_roc.set_ylabel('True Positive Rate')
ax_roc.legend()
```

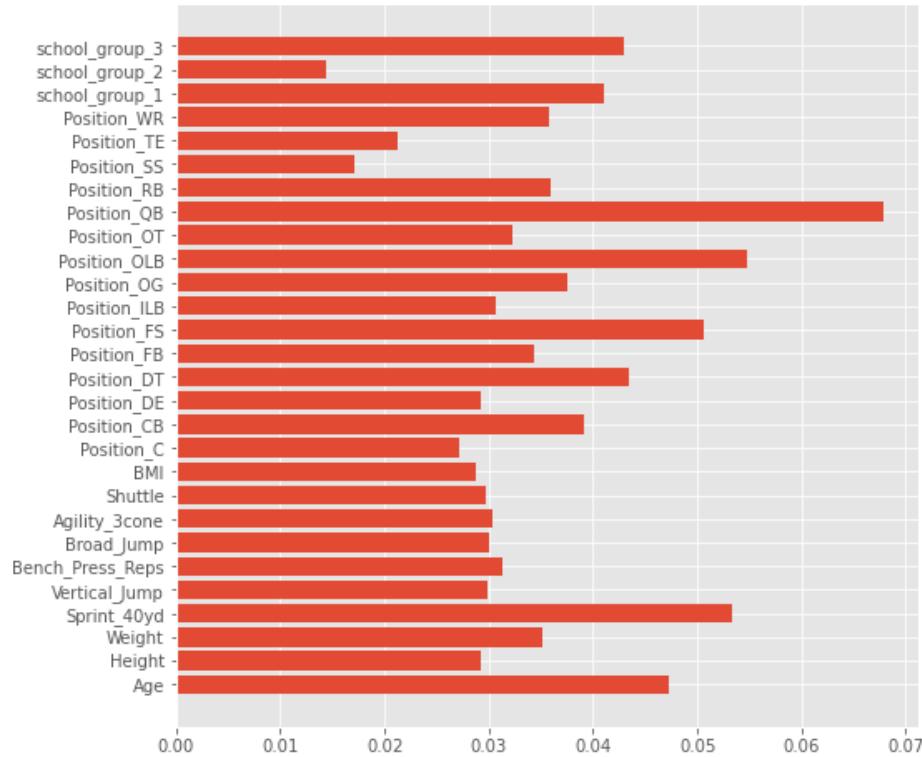
AUC: 0.691410186365568

Out[399]: <matplotlib.legend.Legend at 0x13cad7cd3d0>



```
In [400]: # Plot feature importances
fig, ax = plt.subplots(figsize=(8,8))

ax.barh(np.arange(len(xgb.feature_importances_)),xgb.feature_importances_)
ax.set_yticks(np.arange(len(xgb.feature_importances_)))
ax.set_yticklabels(X_train.columns);
```



Model 2: GridSearchCV

This model varies some of the hyperparameters to find the optimal combination.

I utilize a pipeline to avoid data leakage.

```
In [407]: # XGBClassifier pipeline with GridSearchCV
xgb = XGBClassifier()

param_grid_xgb = {
    'xgb_learning_rate': [0.07, 0.05, 0.03],
    'xgb_max_depth': [3,4,5],
    'xgb_min_child_weight': [1,2,3],
    'xgb_subsample': [0.5, 0.7, 0.9],
    'xgb_n_estimators': [150, 200, 250],
    'xgb_scale_pos_weight':[0.6, 0.8, 1],
    'xgb_booster': ['gbtree']
}

pipe = Pipeline(steps=[
    ('scaler',ScaleByPosition()),
    ('dummies',FunctionTransformer(pd.get_dummies,kw_args={'columns':['Position','school_group']}),
    ('xgb',xgb)
])

xgb_gridcv = GridSearchCV(pipe, param_grid=param_grid_xgb, cv=3)

X = df_combine[['Age','Height','Weight','Sprint_40yd','Vertical_Jump','Bench_Press_Reps','Broad_Jump','Shuttle','BMI','Position','school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

xgb_gridcv.fit(X_train, y_train)

best_parameters = xgb_gridcv.best_params_

print('Grid Search found the following optimal parameters: ')
for param_name in sorted(best_parameters.keys()):
    print('%s: %r' % (param_name, best_parameters[param_name]))

y_train_pred = xgb_gridcv.predict(X_train)
y_test_pred = xgb_gridcv.predict(X_test)

display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

Grid Search found the following optimal parameters:

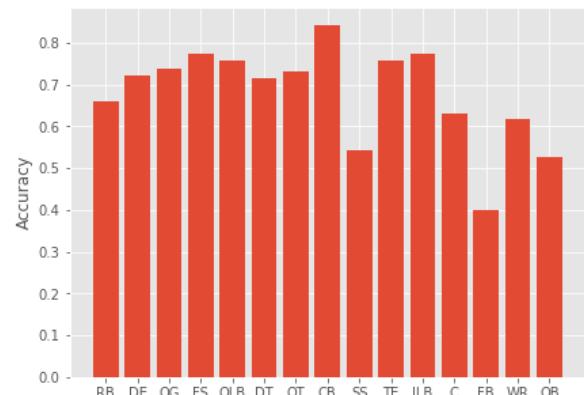
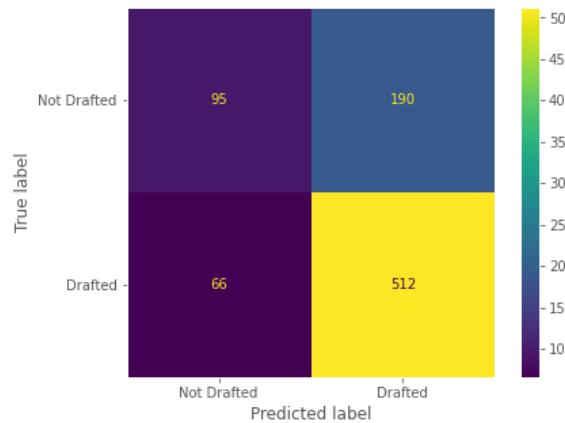
```
xgb_booster: 'gbtree'
xgb_learning_rate: 0.05
xgb_max_depth: 3
xgb_min_child_weight: 1
xgb_n_estimators: 200
xgb_scale_pos_weight: 1
xgb_subsample: 0.9
```

Training:

	precision	recall	f1-score	support
0	0.80	0.47	0.59	808
1	0.80	0.95	0.86	1780
accuracy			0.80	2588
macro avg	0.80	0.71	0.73	2588
weighted avg	0.80	0.80	0.78	2588

Test:

	precision	recall	f1-score	support
0	0.59	0.33	0.43	285
1	0.73	0.89	0.80	578
accuracy			0.70	863
macro avg	0.66	0.61	0.61	863
weighted avg	0.68	0.70	0.68	863

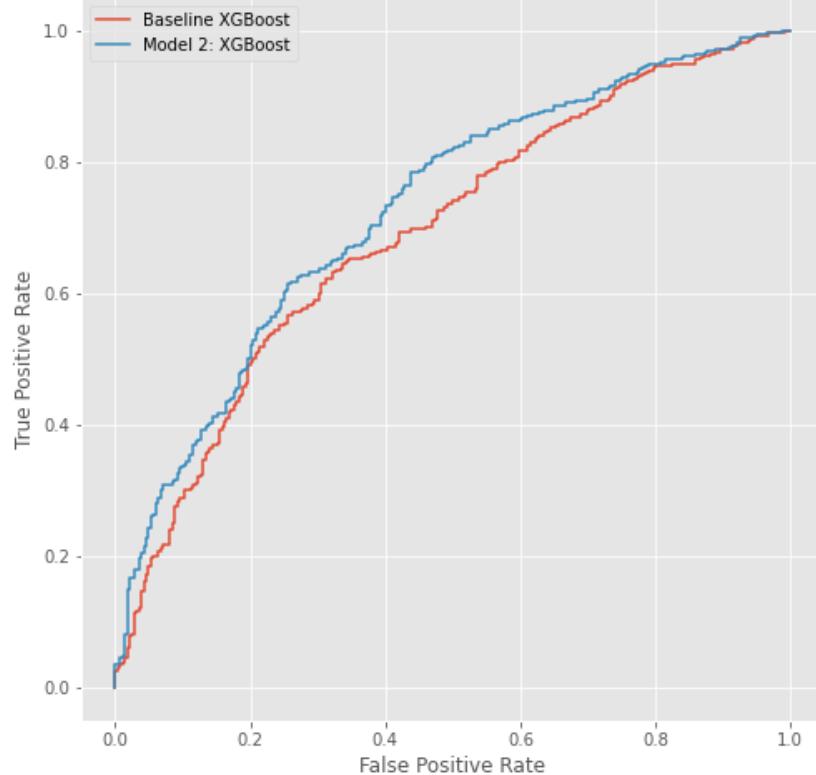


The model still appears to be overfitting slightly. The accuracy of the test data did improve, but it is still a bit lower than the best logistic regression model.

```
In [408]: # Update ROC Curve  
y_test_pred_prob = xgb_gridcv.predict_proba(X_test)  
  
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])  
print('AUC:',auc(fpr, tpr))  
  
ax_roc.plot(fpr, tpr, label='Model 2: XGBoost')  
ax_roc.legend()  
  
fig_roc
```

AUC: 0.7275420384872215

Out[408]:





Model 3: Interaction terms

Like the last model, I use GridSearchCV to test out different combinations of hyperparameters, but I also create interaction terms.

```
In [419]: # XGBClassifier pipeline with GridSearchCV
xgb = XGBClassifier()

param_grid_xgb = {
    'xgb_learning_rate': [0.08, 0.06, 0.04],
    'xgb_max_depth': [4, 5, 6],
    'xgb_min_child_weight': [5, 6, 7],
    'xgb_subsample': [0.6, 0.7, 0.8, 0.9],
    'xgb_n_estimators': [150, 200, 250],
    'xgb_scale_pos_weight': [0.6, 0.8, 1],
    'xgb_booster': ['gbtree']
}

list1 = ['Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Agility']

pipe = Pipeline(steps=[
    ('scaler', ScaleByPosition()),
    ('interterms', FunctionTransformer(create_interaction_terms, kw_args={'list1': list1})),
    ('dummies', FunctionTransformer(pd.get_dummies, kw_args={'columns': ['Position', 'school_group']})),
    ('xgb', xgb)
])

xgb_gridcv = GridSearchCV(pipe, param_grid=param_grid_xgb, cv=3)

X = df_combine[['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump',
                 'Shuttle', 'BMI', 'Position', 'school_group']]
y = df_combine['Drafted']

# Split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rand_seed, test_size=0.25)

xgb_gridcv.fit(X_train, y_train)

best_parameters = xgb_gridcv.best_params_

print('Grid Search found the following optimal parameters: ')
for param_name in sorted(best_parameters.keys()):
    print('%s: %r' % (param_name, best_parameters[param_name]))

y_train_pred = xgb_gridcv.predict(X_train)
y_test_pred = xgb_gridcv.predict(X_test)

display_model_results(y_train, y_train_pred, y_test, y_test_pred, df_combine)
```

Grid Search found the following optimal parameters:

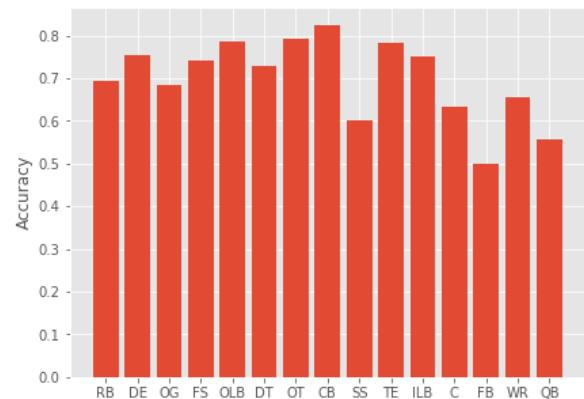
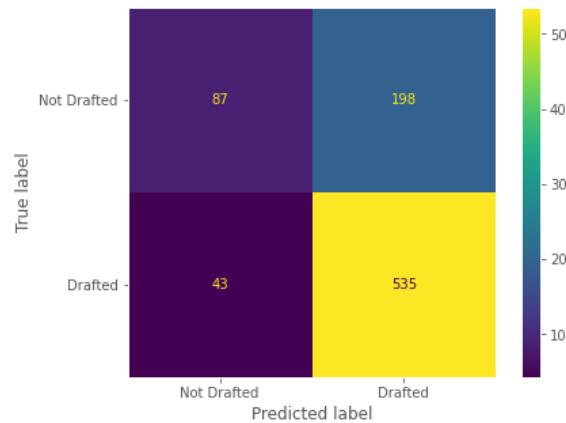
```
xgb_booster: 'gbtree'
xgb_learning_rate: 0.04
xgb_max_depth: 4
xgb_min_child_weight: 5
xgb_n_estimators: 250
xgb_scale_pos_weight: 1
xgb_subsample: 0.6
```

Training:

	precision	recall	f1-score	support
0	0.73	0.33	0.45	808
1	0.76	0.95	0.84	1780
accuracy			0.75	2588
macro avg	0.75	0.64	0.65	2588
weighted avg	0.75	0.75	0.72	2588

Test:

	precision	recall	f1-score	support
0	0.67	0.31	0.42	285
1	0.73	0.93	0.82	578
accuracy			0.72	863
macro avg	0.70	0.62	0.62	863
weighted avg	0.71	0.72	0.69	863

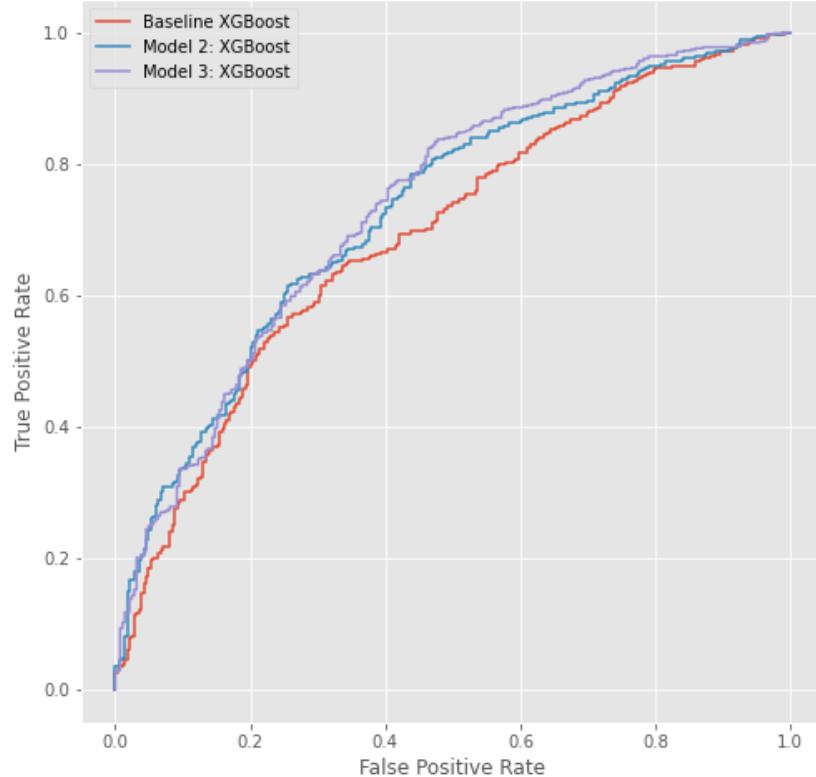


The model is barely achieving a better accuracy than the last one. It also achieved a higher AUC (as seen below). This is the best XGBoost Classifier model, but it still is outperformed by the logistic regression.

```
In [420]: # Update ROC Curve  
y_test_pred_prob = xgb_gridcv.predict_proba(X_test)  
  
fpr, tpr, thresh = roc_curve(y_test, y_test_pred_prob[:,1])  
print('AUC:',auc(fpr, tpr))  
  
ax_roc.plot(fpr, tpr, label='Model 3: XGBoost')  
ax_roc.legend()  
  
fig_roc
```

AUC: 0.735536939233898

Out[420]:



```
In [421]: # Plot feature importance for last model
fig, ax = plt.subplots(figsize=(8,36))

ax.barh(np.arange(len(xgb_gridcv.best_estimator_['xgb'].feature_importances_)),xgb_gridcv.best_e:
ax.set_yticks(np.arange(len(xgb_gridcv.best_estimator_['xgb'].feature_importances_)))
ax.set_yticklabels(pd.get_dummies(create_interaction_terms(X_train,list1),columns=['Position','s
    <ipython-input-337-fdf88b5a8623>:19: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy)
X[f"{list1[i]}xPosition_{list2[j]}"] = X[list1[i]]*X['Position'].\
C:\Users\david\anaconda3\envs\learn-env\lib\site-packages\pandas\core\frame.py:4163: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy)
return super().drop(
```

```
In [422]: # y_test_pred_prob = xgb_gridcv.predict_proba(X_test)

fig, ax = plt.subplots(figsize=(7,7))

keep = y_test==0

ax.hist(np.sort(y_test_pred_prob[keep][:,1]),bins=np.arange(41)/40,alpha=0.5,label='Undrafted',density=True)

keep = y_test==1

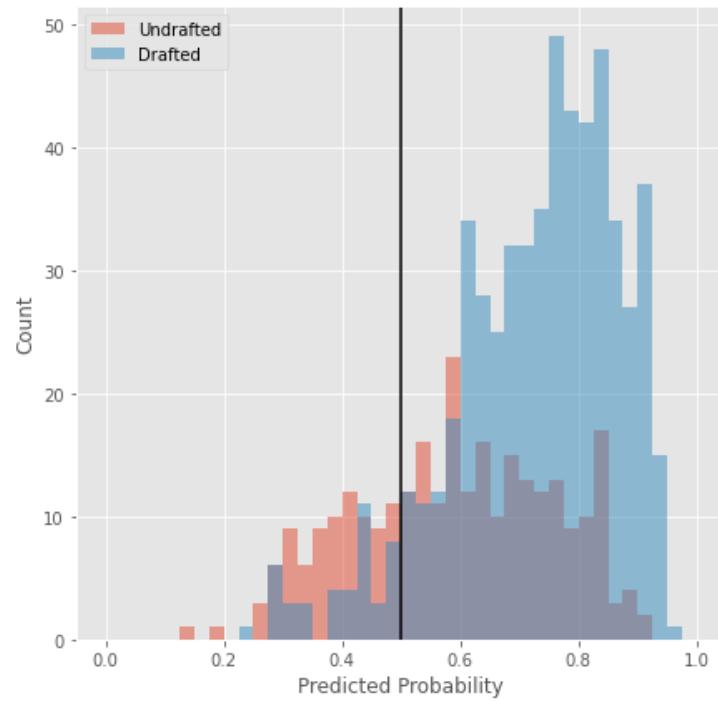
ax.hist(np.sort(y_test_pred_prob[keep][:,1]),bins=np.arange(41)/40,alpha=0.5,label='Drafted',density=True)

ax.axvline(0.5, color='black')

ax.set_xlabel('Predicted Probability')
ax.set_ylabel('Count')

ax.legend()
```

Out[422]: <matplotlib.legend.Legend at 0x13cad8661f0>



Evaluating the XGBoost Classifiers

In this case, interaction terms did make a positive difference in the fit. Adding them reduced the overfitting and increased the accuracy on the test data by 2%. The AUC also improved slightly. However, the logistic regression model still performed the best.

Description	Training Accuracy	Test Accuracy	ROC AUC
Default Hyperparameters	99% (Overfitting)	68%	0.691
No inter terms, optimal hyperparameters		80%	0.728
Used inter terms, optimal hyperparameters		75%	0.736

Summary of Model Fitting Results

How good are the models?

First, the overall performance of the models was not that strong. The accuracy achieved by the models ranged from 67% to 73%. Also, the models tended to overpredict the number of drafted players and had values below 35% for the recall of undrafted players. The 73% accuracy is even less impressive given that 67% of players in the dataset were drafted, so a 67% accuracy could be achieved by simply predicting that all players will get drafted.

It is not surprising that the model was not achieving very high accuracy. The data used here only told us about a player's athleticism, such as how fast they run or how high they jump, and their basic physical characteristics, like age, height, weight, and BMI. While athleticism and physicality are important in the sport of football, they do not reflect any skills specific to football in any way. NFL teams need to use other means to evaluate a player's football ability and understanding such as watching their college game film, talking to them about strategy, or running more football specific drills. The NFL Combine does actually have position-specific drills, but those drills are not in the data, mostly because they do not boil down to a few simple numbers.

So are the models useless?

Even if the models do not achieve a high accuracy, they are still useful. For one, the best models can be used to identify the players who have a very strong chance of being drafted. Looking at the probability of being drafted rather than just the binary prediction, we can see that the players with very high probabilities are drafted at a very high rate.

Also, we can see some other trends in the data, such as which positions tend to get drafted more often and which criteria tend to get used when evaluating Combine performance. It is pretty clear that the most important thing that happens at the Combine is the 40-yard sprint. Not every position cares about top speed, but for the ones that do, the 40-yard sprint time is a major factor in determining who will get drafted.

It also appears that a player's age and the school they attended also matters. Players that attended the bigger football programs tended to get drafted more often and younger players also tended to get drafted more often.

Which type of model worked best?

The best performing model was a logistic regression model that did not use interaction terms and used a value of C=0.1 for regularization.

One thing that I tested for several of the classifiers was whether or not the inclusion of interaction terms improved the fit. Theoretically, the interaction terms are important because the tests are given different weights depending on a player's position. A model without the interaction terms cannot take this into account which should lead to underfitting. However, the models with interaction terms just have tons of parameters and some of the parameters only depend on a small portion of the data, leading to extra variance in the model (overfitting).

A problem for the tree-based classifiers:

One issue that I think the tree-based classifiers (decision tree, random forest, XGBoost) had was that it was difficult for them to take into account position, whether or not I used interaction terms. One reason is because a decision tree only makes binary splits at each node, so when it tries to split based on position, it can only separate one position from the rest rather than efficiently grouping similar positions together. Another issue is that the decision trees use a greedy algorithm; they always want to make the split that separates drafted and undrafted players best. While some positions are drafted more often than others, splitting based on position probably isn't going to be the best, especially in the first couple layers. This makes it harder for the decision trees to isolate certain positions.

Overpredicting drafted players

All of the models that were tested, including the best logistic regression model, overpredicted the number of drafted players. While only 67% of the players in the dataset were drafted, the model predicts that 84.7% of them will get drafted. Hypothetically, I could fix this by changing the cutoff at which the model will predict a player to be drafted. The model produces a single number between 0 and 1 that essentially represents the probability that the player will be drafted. By default, the model will predict 1 for anything above 0.5 and 0 for anything below 0.5. If I choose a larger cutoff, I can bring down the number of players that are predicted to be drafted...

HOWEVER, I do not actually want to do this because it will increase the number of false negatives and I believe that false negatives are worse than false positives. Here is why:

False Positives vs. False Negatives

Here is what a false positive and a false negative mean for this model:

1. False Positive: The model predicts that a player will be drafted, but they turn out not to be drafted.
2. False Negative: The model predicts that a player will not be drafted, but it turns out they are.

The way I would expect an NFL team to use this model is to first identify a player that they would like to have on their team. For the sake of an example, let's say the Arizona Cardinals want to select John Doe. The Cardinals first see whether the model says that John is predicted to be drafted or not. This is useful information because it tells the Cardinals how much they should prioritize John in the draft. If John is predicted to be drafted with high confidence, the Cardinals will need to dedicate an earlier draft pick to select them. If John is not predicted to be drafted, the Cardinals can use a later pick or even try to sign John as an undrafted free agent.

So why are false negatives worse than false positives? In the case of a false negative, the Cardinals are led to believe that John won't get drafted. As a result, they might think they can pick John up late, but then John actually gets drafted by another team in an early round, so the Cardinals miss out on getting John on their team.

In the case of a false positive, the Cardinals think that John is going to get drafted, so they dedicate an early pick to them. The downside is that, since the prediction was wrong and John was not going to get drafted by another team, the Cardinals could have used that pick to select another sought after player and still gotten John later. This can also effect other picks they make later in the draft. While there is this downside to a false positive, the Cardinals still got the player they wanted.

For this reason, a model that emphasizes recall is good because it will reduce false negatives. This is why the overpredicting of drafted players is not a downside. That said, I would still urge a team using the model to consider the predicted probabilities rather than just the binary predictions. If the model calculates a probability of 0.95 for a player, they are almost certainly going to be drafted, but if it calculates a probability of 0.6, then it is more of a toss-up.

Digging Deeper Into The Model

The best performing model was a logistic regression model. Below, I look into what the model tells us about how teams use the NFL Combine.

Below, I run the model again so that I can use it in later cells. This time, I use all the data.

```
In [649]: # Define X and y
X = df_combine[['Age', 'Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Shuttle', 'BMI', 'Position', 'school_group']]
y = df_combine['Drafted']

# Apply scaler
scaler = ScaleByPosition()
scaler.fit(X)
X = scaler.transform(X)

# Apply imputer
imputer = ImputeByPosition()
imputer.fit(X)
X = imputer.transform(X)

# Create interaction terms
list1 = ['Height', 'Weight', 'Sprint_40yd', 'Vertical_Jump', 'Bench_Press_Reps', 'Broad_Jump', 'Agility', 'Position', 'school_group']

#X = create_interaction_terms(X, list1)

# Apply get_dummies to separate categorical data into columns
X = pd.get_dummies(X, columns=['Position', 'school_group'], drop_first=True)

# Instantiate LogisticRegressionModel
logreg = LogisticRegression(C=0.1, random_state=42, solver='lbfgs', max_iter=5000)

# Fit model
logreg.fit(X, y)

# Get predictions
y_pred = logreg.predict(X)

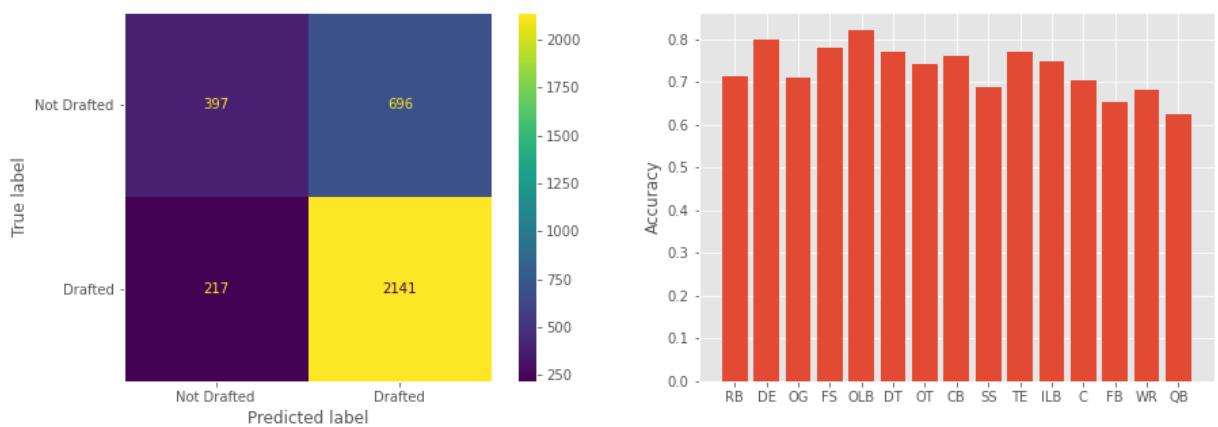
# I feed the data twice because display_model expects training and test data
display_model_results(y, y_pred, y, y_pred, df_combine)
```

Training:

	precision	recall	f1-score	support
0	0.65	0.36	0.47	1093
1	0.75	0.91	0.82	2358
accuracy			0.74	3451
macro avg	0.70	0.64	0.64	3451
weighted avg	0.72	0.74	0.71	3451

Test:

	precision	recall	f1-score	support
0	0.65	0.36	0.47	1093
1	0.75	0.91	0.82	2358
accuracy			0.74	3451
macro avg	0.70	0.64	0.64	3451
weighted avg	0.72	0.74	0.71	3451



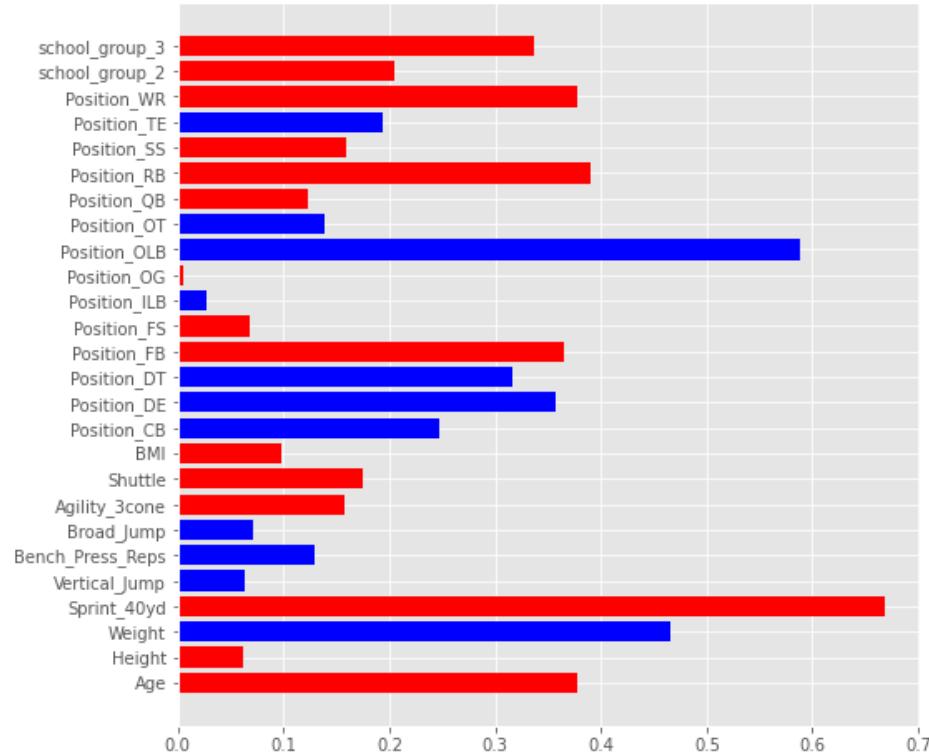
Below, I look at the coefficients for the model. The positive coefficients are in blue while negative coefficients are in red.

The constant coefficient is not shown. The constant coefficient corresponds to a player from school_group_1 and is a center. This means that the other two school_group coefficients are relative to school_group_1 and all of the Position coefficients are relative to centers.

All of the numerical data has been rescaled so that the coefficients can be compared to one another.

```
In [652]: # Look at the coefficients for the model
fig, ax = plt.subplots(figsize=(8,8))

ax.barh(np.arange(len(X.columns)), np.abs(logreg.coef_[0]), color=['blue' if x>0 else 'red' for x in logreg.coef_[0]])
ax.set_yticks(np.arange(len(X.columns)))
ax.set_yticklabels(X.columns);
```



School Group

As expected, players from school_group_1 get drafted the most often and school_group_3 the least often. This means that teams are prioritizing players from larger college football programs.

Position

At least as far as this data is concerned, outside linebackers (OLB) had the highest percentage of being drafted while running backs (RB) were the least likely to get drafted. This is timely because there is currently a lot of coverage about how elite running backs are not being valued.

Athletic Tests

By far the most important of the 6 athletic tests is the 40-yard sprint. Being faster is very important to several positions in football and that is reflected here.

Weight and Age

Both the weight and age coefficients had larger absolute values than the athletic tests (save for the sprint). Teams tend to want heavier players and younger players. I think the preference for younger players makes a lot of sense. Most players who could make an NFL roster at age 25 probably already made one when they were 23 or younger.

Improved Plots

Below, I change up the plots so they will be better for my presentation. I will improve them by sorting them and changing the colors.

```
In [730]: # Create dictionary of coefficients
coeffs = {x: y for x, y in zip(X.columns, logreg.coef_[0])}
coeffs['school_group_1'] = 0.0
coeffs['Position_C'] = 0.0

list_tests = ['Vertical_Jump', 'Broad_Jump', 'Bench_Press_Reps', 'Agility_3cone', 'Shuttle', 'Sprint_40_yard', '3_Cone_Drill']
labels_tests = ['Vertical Jump', 'Broad Jump', 'Bench Press Reps', '3-Cone Drill', 'Shuttle', '40-yard Sprint', '3-Cone Drill']

list_phys = ['Height', 'BMI', 'Age', 'Weight']

# Look at the coefficients for the model
fig, ax = plt.subplots(nrows=2, figsize=(6,5), gridspec_kw = {'height_ratios':[6,4]})

# Combine tests
ax[0].set_title('NFL Combine Tests')

ax[0].barh(np.arange(len(list_tests)), [np.abs(coeffs[x]) for x in list_tests],
           color=['#013369' if coeffs[x]>0 else '#D50A0A' for x in list_tests])

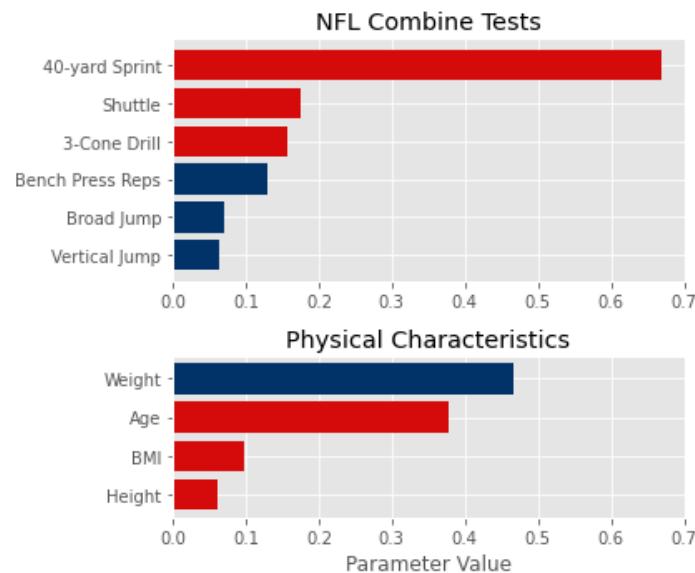
ax[0].set_yticks(np.arange(len(list_tests)))
ax[0].set_yticklabels(labels_tests)
ax[0].set_xbound([0,0.7])

# Physical Characteristics
ax[1].set_title('Physical Characteristics')

ax[1].barh(np.arange(len(list_phys)), [np.abs(coeffs[x]) for x in list_phys],
           color=['#013369' if coeffs[x]>0 else '#D50A0A' for x in list_phys])

ax[1].set_yticks(np.arange(len(list_phys)))
ax[1].set_yticklabels(list_phys)
ax[1].set_xbound([0,0.7])
ax[1].set_xlabel('Parameter Value')

fig.tight_layout();
```



```
In [731]: list_pos = ['Position_RB', 'Position_WR', 'Position_FB', 'Position_SS', 'Position_QB', 'Position_FS', 'Position_ILB', 'Position_OT', 'Position_TE', 'Position_CB', 'Position_DT', 'Position_DE']
labels_pos = ['Running Backs', 'Wide Receivers', 'Fullbacks', 'Strong Safeties', 'Quarterbacks', 'Free Safeties', 'Offensive Guards', 'Centers', 'Inside Linebackers', 'Offensive Tackles', 'Tight Ends', 'Defensive Tackles', 'Defensive Ends', 'Outside Linebackers']

list_school = ['school_group_3', 'school_group_2', 'school_group_1']
labels_school = ['Schools 52-255', 'Schools 22-51', 'Schools 1-21']

coeffs['school_group_1'] == coeffs['school_group_2']
coeffs['school_group_3'] == coeffs['school_group_2']
coeffs['school_group_2'] == coeffs['school_group_2']

# Look at the coefficients for the model
fig, ax = plt.subplots(nrows=2, figsize=(6,6), gridspec_kw = {'height_ratios':[15,3]})

# Positions
ax[0].set_title('Positions')

ax[0].barh(np.arange(len(list_pos)), [np.abs(coeffs[x]) for x in list_pos],
           color=['#013369' if coeffs[x]>0 else '#D50A0A' for x in list_pos])

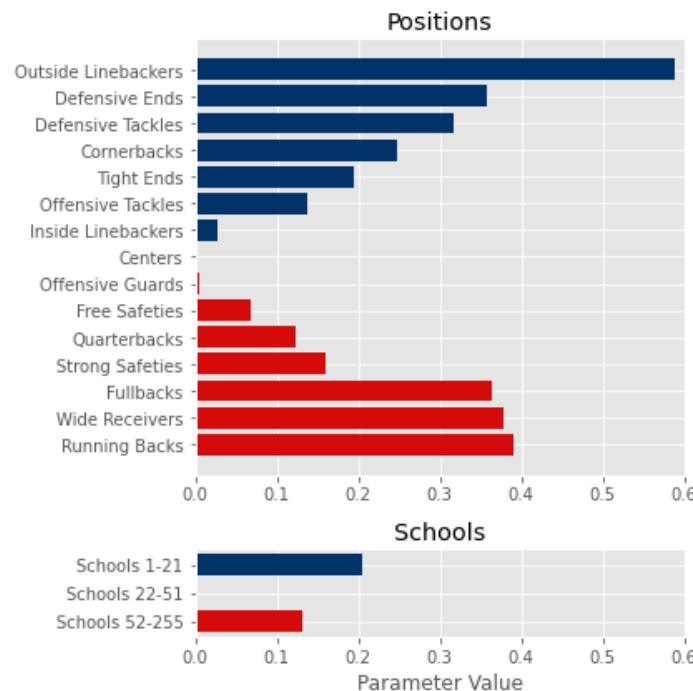
ax[0].set_yticks(np.arange(len(list_pos)))
ax[0].set_yticklabels(labels_pos)
ax[0].set_xbound([0,0.6])

# School Groups
ax[1].set_title('Schools')

ax[1].barh(np.arange(len(list_school)), [np.abs(coeffs[x]) for x in list_school],
           color=['#013369' if coeffs[x]>0 else '#D50A0A' for x in list_school])

ax[1].set_yticks(np.arange(len(list_school)))
ax[1].set_yticklabels(labels_school)
ax[1].set_xbound([0,0.6])
ax[1].set_xlabel('Parameter Value')

fig.tight_layout();
```



How Important are the Combine Tests?

Below, I estimate the probability that a player will get drafted given a certain performance on one of the Combine tests.

Here is the math behind how this works:

The logistic regression uses a sigmoid function of this form:

$$P = \frac{1}{1 + e^{-(C+B_1 \cdot x_1 + B_2 \cdot x_2 + B_3 \cdot x_3 \dots)}}$$

where C and the B_n are the parameters in the fit.

The model predicts 1 if $P > 0.5$ and 0 if $P < 0.5$. The probability P equals exactly 0.5 when the linear terms in the exponent all combine to equal 0 (because $e^0 = 1$). This means that the model predicts 1 when the exponent is positive and 0 when it is negative.

The exponent is just a linear combination of features. We can estimate the distribution of the exponent by propagating the uncertainty in the x_n . Since the numerical values were transformed using a standard scaler prior to fitting, all the uncertainties are equal to 1. This means that we can estimate the variance in the exponent by adding the squares of the parameters (not including the categorical variables).

```
In [578]: # Choose test, position, and school_group
combine_test = 'Sprint_40yd'
position = 'RB'
school = 'school_group_1'

# Create a dictionary that holds the coefficients
coeffs = {x: y for x,y in zip(X.columns,logreg.coef_[0])}

cols = ['Age','Height','Weight','Sprint_40yd','Vertical_Jump','Bench_Press_Reps','Broad_Jump','Age','BMI']

# Estimate variance of exponential term
var_exp = 0.0
for col in cols:
    if col != combine_test: var_exp += coeffs[col]*coeffs[col]

# Calculate offset from categorical variables
offset = 0.0
if position != 'C': offset += coeffs['Position_'+position]
if school != 'school_group_1': offset += coeffs[school]

# Range of values to use
start_z = -3
end_z = 3

z = np.arange(start_z, end_z, 0.1)
probs = 1-st.norm.cdf(0, z*coeffs[combine_test] + offset + logreg.intercept_[0], np.sqrt(var_exp))

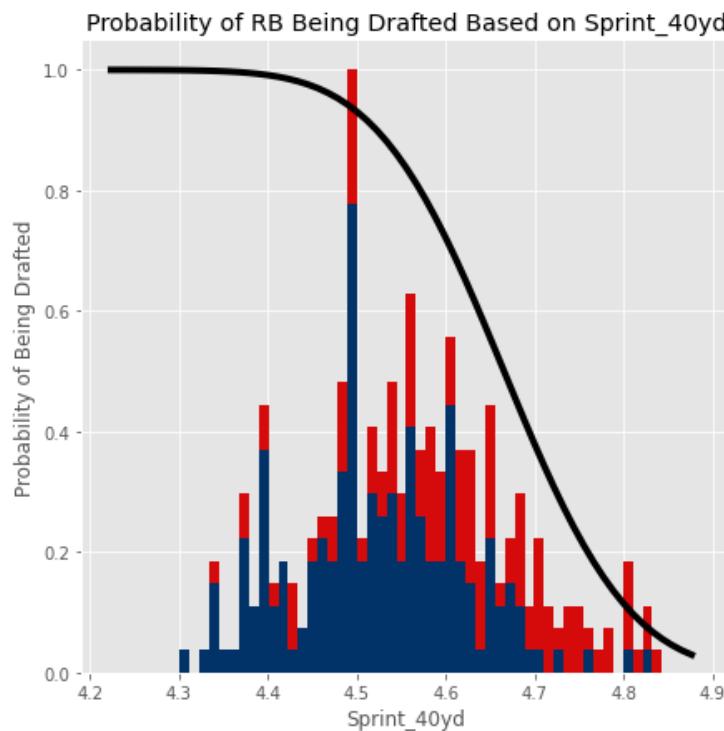
fig, ax = plt.subplots(figsize=(7,7))

bins = np.arange(start_z, end_z+0.1, 0.1)*df_combine[df_combine['Position']==position][combine_test].std() + df_combine[df_combine['Position']==position][combine_test].mean()

histo = np.histogram(df_combine[df_combine['Position']==position][combine_test], bins=bins)
histo_yes = np.histogram(df_combine[(df_combine['Position']==position) & (df_combine['Drafted']==1)][combine_test], bins=bins)

ax.bar(histo[1][:-1], histo[0]/np.max(histo[0]), width=histo[1][1]-histo[1][0], align='edge', color='black')
ax.bar(histo_yes[1][:-1], histo_yes[0]/np.max(histo[0]), width=histo_yes[1][1]-histo_yes[1][0], align='edge', color="#013369")

ax.plot(z*df_combine[df_combine['Position']==position][combine_test].std()+
        df_combine[df_combine['Position']==position][combine_test].mean(), probs, color='black',
        marker='x', markersize=10)
ax.set_xlabel(combine_test)
ax.set_ylabel('Probability of Being Drafted')
ax.set_title(f"Probability of {position} Being Drafted Based on {combine_test}");
```



I think the plot above can be misleading. Really, it shows the probability that the model will predict 1. This means that the actual probability output by the sigmoid will be 0.51, meaning the model will predict 1, but not with much confidence.

Alternative: Calculate the actual probability that the sigmoid will produce and show error with shading based on uncertainty in other variables. This means passing the variance in the exponent through the sigmoid.

```
In [587]: # Choose test, position, and school_group
combine_test = 'Weight'
position = 'QB'
school = 'school_group_1'

# Create a dictionary that holds the coefficients
coeffs = {x: y for x,y in zip(X.columns,logreg.coef_[0])}

cols = ['Age','Height','Weight','Sprint_40yd','Vertical_Jump','Bench_Press_Reps','Broad_Jump','Age','BMI']

# Estimate variance of exponential term
var_exp = 0.0
for col in cols:
    if col != combine_test: var_exp += coeffs[col]*coeffs[col]

# Calculate offset from categorical variables
offset = 0.0
if position != 'C': offset += coeffs['Position']+position
if school != 'school_group_1': offset += coeffs[school]

# Range of values to use
start_z = -3
end_z = 3

z = np.arange(start_z, end_z, 0.1)
probs = 1/(1+np.exp(-(z*coeffs[combine_test] + offset + logreg.intercept_[0])))
probs_up1 = 1/(1+np.exp(-(z*coeffs[combine_test] + offset + logreg.intercept_[0] + np.sqrt(var_exp))))
probs_dn1 = 1/(1+np.exp(-(z*coeffs[combine_test] + offset + logreg.intercept_[0] - np.sqrt(var_exp)))

fig, ax = plt.subplots(figsize=(7,7))

bins = np.arange(start_z, end_z+0.1, 0.1)*df_combine[df_combine['Position']==position][combine_test].std() + df_combine[df_combine['Position']==position][combine_test].mean()

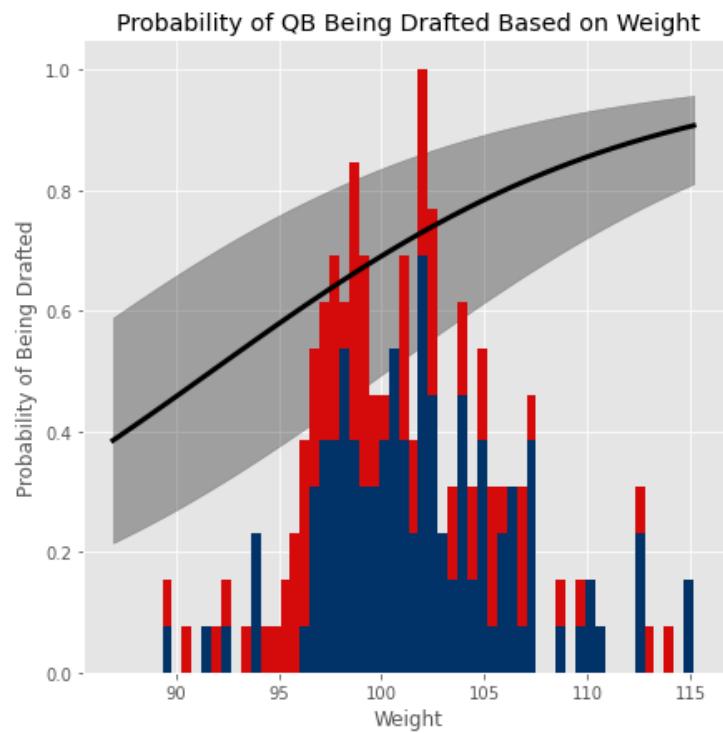
histo = np.histogram(df_combine[df_combine['Position']==position][combine_test], bins=bins)
histo_yes = np.histogram(df_combine[(df_combine['Position']==position) & (df_combine['Drafted']==1)], bins=bins)

ax.bar(histo[1][:-1], histo[0]/np.max(histo[0]), width=histo[1][1]-histo[1][0], align='edge', color='black')
ax.bar(histo_yes[1][:-1], histo_yes[0]/np.max(histo[0]), width=histo_yes[1][1]-histo_yes[1][0], align='edge', color="#013369")

ax.fill_between(z*df_combine[df_combine['Position']==position][combine_test].std() + df_combine[df_combine['Position']==position][combine_test].mean(), probs_up1, probs_dn1, color='black', alpha=0.3)

ax.plot(z*df_combine[df_combine['Position']==position][combine_test].std() + df_combine[df_combine['Position']==position][combine_test].mean(), probs, color='black', linewidth=2)

ax.set_xlabel(combine_test)
ax.set_ylabel('Probability of Being Drafted')
ax.set_title(f"Probability of {position} Being Drafted Based on {combine_test}");
```



```
In [110]: # Accuracy as a function of predicted probability
y_test_pred_prob = logreg.predict_proba(X_test)

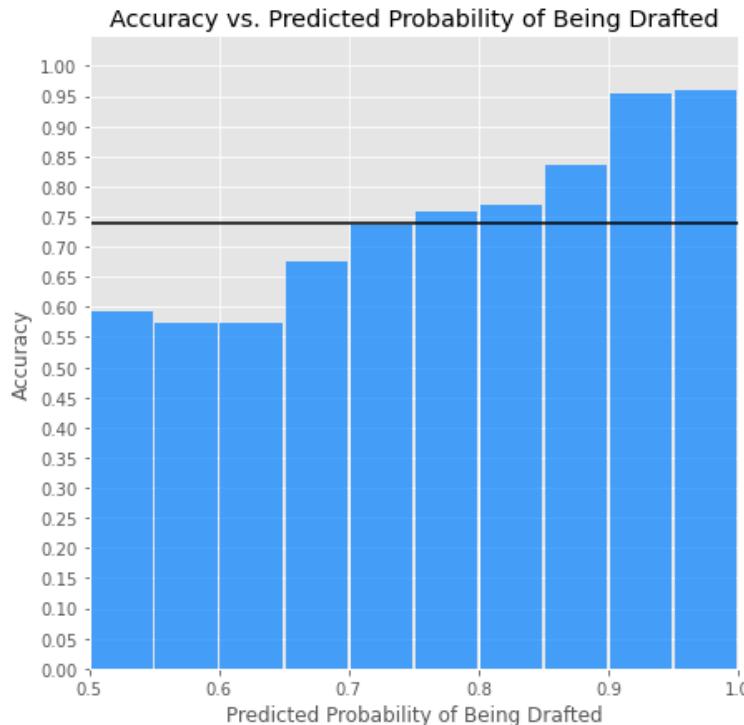
acc = np.zeros(20)
for i, prob in enumerate(np.arange(20)/20):
    keep = (y_test_pred_prob[:,1] >= prob) & (y_test_pred_prob[:,1] < prob+0.05)
    acc[i] = sum(y_test[keep] == y_test_pred[keep]) / sum(keep)

fig, ax = plt.subplots(figsize=(7,7))

ax.bar(np.arange(20)/20+0.025, acc, width=0.048, align='center', color=(0.0,0.5,1.0,0.7))
ax.axhline(0.74, color = 'black')
ax.set_xbound([0.5,1])
ax.set_yticks(np.arange(21)/20)
ax.set_title('Accuracy vs. Predicted Probability of Being Drafted')
ax.set_xlabel('Predicted Probability of Being Drafted')
ax.set_ylabel('Accuracy')
```

```
<ipython-input-110-2543d0ef1a00>:7: RuntimeWarning: invalid value encountered in long_scalars
acc[i] = sum(y_test[keep] == y_test_pred[keep]) / sum(keep)
<ipython-input-110-2543d0ef1a00>:7: RuntimeWarning: invalid value encountered in long_scalars
acc[i] = sum(y_test[keep] == y_test_pred[keep]) / sum(keep)
```

Out[110]: Text(0, 0.5, 'Accuracy')



While the overall accuracy of the model is just 74%, when the model assigns a probability of 90% or more to a player, they get drafted over 95% of the time. There are 92 such players out of 863 in the training data.