NFL Combine Draft Classification Analysis

Eric Au

Source: Getty Images (https://www.sportingnews.com/us/nfl/news/nfl-combine-drills-explained-40-yard-dash/lds11epxn7znufqyjkaphguq)

1. Business and Data Understanding

Stakeholders: The New York Giants front office (General Manager, President, Scouting Department).

Every year, the National Football League (NFL) holds a week long showcase where college football players, otherwise known as prospects, perform physical drills and tests in front of team coaches, scouts, and general managers. These drills are intended to measure a player's physical ability such as speed, quickness, strength, and overall athleticism.

- *But what can NFL teams learn from these workouts?*
- *What exactly do non-football athletic testing measurements contribute to prospect evaluation? Is there any value to the combine?*

These are questions that many fans ask to this day and NFL teams try to interpret to make the best decision possible when drafting their players.

For this analysis, we will be using player combine data scraped from Pro-Football Reference (https://www.pro-football-reference.com/) over the last 22 years (2000-2022).

Each record represents an individual player who was eligible to be drafted with information related to their combine measurements. Additionally, each record indicates whether that player was `Drafted` with 1 for "Yes" and 0 for "No". This will be further clarified in the preliminary cleaning of the data set.

The Task: Create a model that can determine whether a player was `Drafted` or `Undrafted` based on the available data and information provided in the NFL Combine.

We will attempt to find any significance between the combine and draft status as well as explore the following:

- How significant are the measurables in determining draft status? Which measurables are good indicators of draft status?

In [32]:
```
# import libraries
import pandas as pd
import numpy as np
import math
import xgboost as xgb

import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

from sklearn.preprocessing import OneHotEncoder, StandardScaler, MaxAbsScaler
from sklearn.impute import SimpleImputer

from sklearn.compose import ColumnTransformer

from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score, \
RandomizedSearchCV, StratifiedKFold, KFold
from sklearn.feature_selection import SelectFromModel

from sklearn.ensemble import BaggingClassifier, RandomForestClassifier, \
AdaBoostClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import precision_score, recall_score, f1_score, \
accuracy_score, classification_report, log_loss, plot_confusion_matrix, confusion_matrix, \
plot_roc_curve, make_scorer, roc_auc_score

from sklearn.base import clone

# ignore warnings
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
pd.options.mode.chained_assignment = None
```

In [33]:
```
# load in data
df = pd.read_csv('Data/combine_2000_2022.csv', index_col = 0)
df.head()
```

Out[33]:

	Player	Pos	School	College	Ht	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Drafted (tm/rnd/yr)	Year
0	John Abraham	OLB	South Carolina	NaN	6-4	252.0	4.55	NaN	NaN	NaN	NaN	NaN	New York Jets / 1st / 13th pick / 2000	2000
1	Shaun Alexander	RB	Alabama	College Stats	6-0	218.0	4.58	NaN	NaN	NaN	NaN	NaN	Seattle Seahawks / 1st / 19th pick / 2000	2000
2	Darnell Alford	OT	Boston Col.	NaN	6-4	334.0	5.56	25.0	23.0	94.0	8.48	4.98	Kansas City Chiefs / 6th / 188th pick / 2000	2000
3	Kyle Allamon	TE	Texas Tech	NaN	6-2	253.0	4.97	29.0	NaN	104.0	7.29	4.49	NaN	2000
4	Rashard Anderson	CB	Jackson State	NaN	6-2	206.0	4.55	34.0	NaN	123.0	7.18	4.15	Carolina Panthers / 1st / 23rd pick / 2000	2000

In [34]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 7680 entries, 0 to 7820
Data columns (total 14 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   Player                7680 non-null   object 
 1   Pos                   7680 non-null   object 
 2   School                7680 non-null   object 
 3   College               6240 non-null   object 
 4   Ht                    7651 non-null   object 
 5   Wt                    7656 non-null   float64
 6   40yd                  7206 non-null   float64
 7   Vertical              5932 non-null   float64
 8   Bench                 5096 non-null   float64
 9   Broad Jump            5859 non-null   float64
 10  3Cone                 4792 non-null   float64
 11  Shuttle               4895 non-null   float64
 12  Drafted (tm/rnd/yr)   4937 non-null   object 
 13  Year                  7680 non-null   int64  
dtypes: float64(7), int64(1), object(6)
memory usage: 900.0+ KB
```

Looks like we need to do some data cleaning based on the missing values, incorrect data types, and identifying whether someone was drafted or not as a binary value (0 or 1).

For further information on the drills/measurables, this [USA Today article (https://ftw.usatoday.com/lists/nfl-combine-drills-three-cone-shuttle-explained)](https://ftw.usatoday.com/lists/nfl-combine-drills-three-cone-shuttle-explained) provides an in depth explanation.

2. Initial Data Preparation

The following steps below are preliminary data cleaning just to get the dataframe in a working set condition for the `train_test_split`. Handling of missing values will be performed following the split.

Convert to float types in appropriate columns;

In [35]:
```
# create a list of columns that need to be floats
col_floats = ['Wt', '40yd', 'Vertical', 'Bench', 'Broad Jump', '3Cone', 'Shuttle']
```

In [36]:
```
# convert to type float for col_floats columns
df[col_floats].astype(float)
```

Out[36]:

	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle
0	252.0	4.55	NaN	NaN	NaN	NaN	NaN
1	218.0	4.58	NaN	NaN	NaN	NaN	NaN
2	334.0	5.56	25.0	23.0	94.0	8.48	4.98
3	253.0	4.97	29.0	NaN	104.0	7.29	4.49
4	206.0	4.55	34.0	NaN	123.0	7.18	4.15
...	...	...	...	...	...	...	...
7816	304.0	4.77	29.0	NaN	111.0	NaN	NaN
7817	255.0	NaN	NaN	NaN	NaN	NaN	NaN
7818	206.0	NaN	NaN	12.0	NaN	NaN	NaN
7819	316.0	5.13	28.5	27.0	110.0	7.75	4.71
7820	215.0	4.88	30.0	NaN	109.0	7.19	4.40

7680 rows × 7 columns

Remove missing data

There are 29 rows of missing height data. Closer inspection reveals that these values were from 2021 (likely due to logistical data collection due to pandemic rules). While we can do research and fill in each individual player's height, we will simply remove them from the data set for now.

In [37]:
```
# there are 29 missing heights in the dataframe
missing_heights = df[df['Ht'].isna()].index
len(missing_heights)
```

Out[37]: 29

In [38]:
```
# remove these rows from the df
df = df.drop(missing_heights)
```

In [39]:
```
# check shape (7680 - 29) = 7651
df.shape
```

Out[39]: (7651, 14)

Convert `Height` to appropriate float value

In [40]:
```
# create feet and inches columns to seperate the 'Ht'
df['feet'] = df['Ht'].str.split("-").str[0].astype(int)
df['inches'] = df['Ht'].str.split("-").str[1].astype(int)

# add new Height column that calculates height as a float
df['Height'] = df['feet'] + round((df['inches']/12),2)
```

In [41]:
```
df_model = df.drop(columns = ['Ht', 'College', 'feet', 'inches', 'Drafted (tm/rnd/yr)'])
df_model
```

Out[41]:

	Player	Pos	School	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Year	Height
0	John Abraham	OLB	South Carolina	252.0	4.55	NaN	NaN	NaN	NaN	NaN	2000	6.33
1	Shaun Alexander	RB	Alabama	218.0	4.58	NaN	NaN	NaN	NaN	NaN	2000	6.00
2	Darnell Alford	OT	Boston Col.	334.0	5.56	25.0	23.0	94.0	8.48	4.98	2000	6.33
3	Kyle Allamon	TE	Texas Tech	253.0	4.97	29.0	NaN	104.0	7.29	4.49	2000	6.17
4	Rashard Anderson	CB	Jackson State	206.0	4.55	34.0	NaN	123.0	7.18	4.15	2000	6.17
...	...	...	...	...	...	...	...	...	...	...	...	...
7816	Devonte Wyatt	DT	Georgia	304.0	4.77	29.0	NaN	111.0	NaN	NaN	2022	6.25
7817	Jalen Wydermyer	TE	Texas A&M	255.0	NaN	NaN	NaN	NaN	NaN	NaN	2022	6.33
7818	Cade York	K	LSU	206.0	NaN	NaN	12.0	NaN	NaN	NaN	2022	6.08
7819	Nick Zakelj	OT	Fordham	316.0	5.13	28.5	27.0	110.0	7.75	4.71	2022	6.50
7820	Bailey Zappe	QB	Western Kentucky	215.0	4.88	30.0	NaN	109.0	7.19	4.40	2022	6.08

7651 rows × 12 columns

In [42]:
```
# remove unneeded columns and extra columns created
df = df.drop(columns = ['Player', 'Ht', 'College', 'feet', 'inches', 'Year'])
```

In [43]:
```
# rename the columns
df = df[['Pos', 'School', 'Wt', '40yd', 'Vertical', 'Bench',
       'Broad Jump', '3Cone', 'Shuttle', 'Drafted (tm/rnd/yr)', 'Height']]
```

Create binary `Drafted` Column of 1 - Yes, 0 - No if player was drafted

In [44]:
```
# fill missing in drafted column with 0
df['Drafted (tm/rnd/yr)'] = df['Drafted (tm/rnd/yr)'].fillna(0)
```

In [45]:
```
# create new column Drafted that designates whether drafted (1) or not (0)
df['Drafted'] = np.where(df['Drafted (tm/rnd/yr)'] == 0, 0, 1)

# drop extra drafted column now
df.drop(columns = 'Drafted (tm/rnd/yr)', inplace = True)
```

Lets now inspect the dataframe.

In [46]:
```
df.head(5)
```

Out[46]:

	Pos	School	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Height	Drafted
0	OLB	South Carolina	252.0	4.55	NaN	NaN	NaN	NaN	NaN	6.33	1
1	RB	Alabama	218.0	4.58	NaN	NaN	NaN	NaN	NaN	6.00	1
2	OT	Boston Col.	334.0	5.56	25.0	23.0	94.0	8.48	4.98	6.33	1
3	TE	Texas Tech	253.0	4.97	29.0	NaN	104.0	7.29	4.49	6.17	0
4	CB	Jackson State	206.0	4.55	34.0	NaN	123.0	7.18	4.15	6.17	1

Note: The below code just a new df using the cleaning process from above to be utilized later in this notebook for analysis.

In [47]:
```
df_metrics = df
df_metrics.head(3)
```

Out[47]:

	Pos	School	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Height	Drafted
0	OLB	South Carolina	252.0	4.55	NaN	NaN	NaN	NaN	NaN	6.33	1
1	RB	Alabama	218.0	4.58	NaN	NaN	NaN	NaN	NaN	6.00	1
2	OT	Boston Col.	334.0	5.56	25.0	23.0	94.0	8.48	4.98	6.33	1

In [48]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 7651 entries, 0 to 7820
Data columns (total 11 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   Pos         7651 non-null   object 
 1   School      7651 non-null   object 
 2   Wt          7651 non-null   float64
 3   40yd        7191 non-null   float64
 4   Vertical    5919 non-null   float64
 5   Bench       5082 non-null   float64
 6   Broad Jump  5850 non-null   float64
 7   3Cone       4784 non-null   float64
 8   Shuttle     4888 non-null   float64
 9   Height      7651 non-null   float64
 10  Drafted     7651 non-null   int64  
dtypes: float64(8), int64(1), object(2)
memory usage: 717.3+ KB
```

In [49]:
```
df.describe()
```

Out[49]:

	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Height	Drafted
count	7651.000000	7191.000000	5919.000000	5082.000000	5850.000000	4784.000000	4888.000000	7651.000000	7651.000000
mean	242.735852	4.776540	32.892854	20.744392	114.590769	7.284933	4.399677	6.149007	0.643315
std	45.236972	0.304748	4.217085	6.378024	9.351811	0.417687	0.267133	0.220414	0.479052
min	144.000000	4.220000	17.500000	2.000000	74.000000	6.280000	3.730000	5.330000	0.000000
25%	205.000000	4.540000	30.000000	16.000000	109.000000	6.980000	4.200000	6.000000	0.000000
50%	232.000000	4.690000	33.000000	21.000000	116.000000	7.190000	4.360000	6.170000	1.000000
75%	280.000000	4.980000	36.000000	25.000000	121.000000	7.530000	4.560000	6.330000	1.000000
max	384.000000	6.050000	46.500000	49.000000	147.000000	9.120000	5.560000	6.830000	1.000000

In [50]:
```
# save as cleaned csv
df.to_csv('Data/cleaned_combine_2000_2022.csv')
```

3. Exploratory Data Analysis

Position Breakdown

There are 22 different positions available in the combine. Let's further explore the distribution of positions through a brief EDA.

In [51]:
```
# check positions count
sorted_positions = df['Pos'].value_counts()
sorted_positions
```

Out[51]:
```
WR      1058
CB       761
RB       671
S        559
OT       524
DE       519
DT       497
OLB      430
TE       425
QB       416
OG       403
ILB      276
C        184
LB       154
P        142
OL       139
FB       120
DL       116
K        103
EDGE      65
DB        59
LS        30
Name: Pos, dtype: int64
```

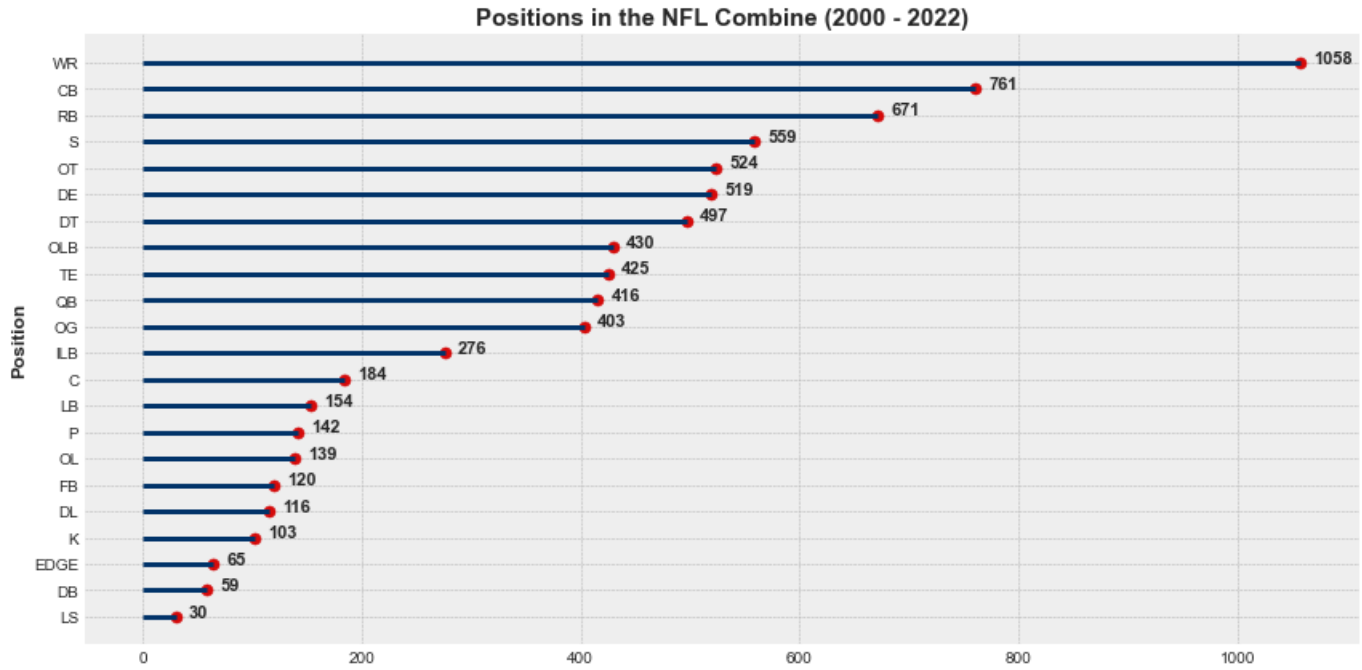
In [52]:
```
# create positions_df that orders positions by most frequent to least
positions = ['WR', 'CB', 'RB', 'S', 'OT', 'DE', 'DT', 'OLB', 'TE', 'QB', 'OG', 'ILB',
       'C', 'LB', 'P', 'OL', 'FB', 'DL', 'K', 'EDGE', 'DB', 'LS']

values = [1058,  761,  671,  559,  524,  519,  497,  430,  425,  416,  403,
        276,  184,  154,  142,  139,  120,  116,  103,   65,   59,   30]

data = {'Position': positions,
        'Count': values}

positions_df = pd.DataFrame(data,
                        columns=['Count'],
                        index = positions)
```

In [53]:
```
positions_df.head(3)
```

Out[53]:

	Count
WR	1058
CB	761
RB	671

In [54]:
```
# plot a lollipop plot of most frequent positions in the combine
fig, ax = plt.subplots(figsize=(12, 6))
plt.style.use('bmh')

# create dataframe of positions & number of drafted
ordered_df = positions_df.sort_values(by = 'Count')

# plots the lines
plt.hlines(y = ordered_df.index,  # positions
           xmin = 0,
           xmax = ordered_df['Count'],  # Count values
           color = '#013369', linewidth = 3)

plt.scatter(ordered_df['Count'], # Count values
            ordered_df.index,  # positions
            color='#d50a0a',
            alpha=1, s = 50)

# annotate the scatter plot values
for idx, row in ordered_df.iterrows():
    ax.annotate(round(row['Count'],4), (row['Count'], idx ),
                weight = "bold", fontsize = 11,
                xytext=(row['Count'] + 12, idx)) # offset annotate

plt.title('Positions in the NFL Combine (2000 - 2022)', weight = "bold", fontsize = 15)
plt.ylabel('Position', weight = "bold", fontsize = 12)

plt.tight_layout()
plt.show()
fig.savefig('Images/Plots/Combine_Positions.png');
```



Let's split these 22 positions into a more organized group based on Offensive, Defensive, and Special Teams positions.

Note: Later we'll create individual models for each positional group and see if we can create a less generalized model that doesn't use the entire dataset.

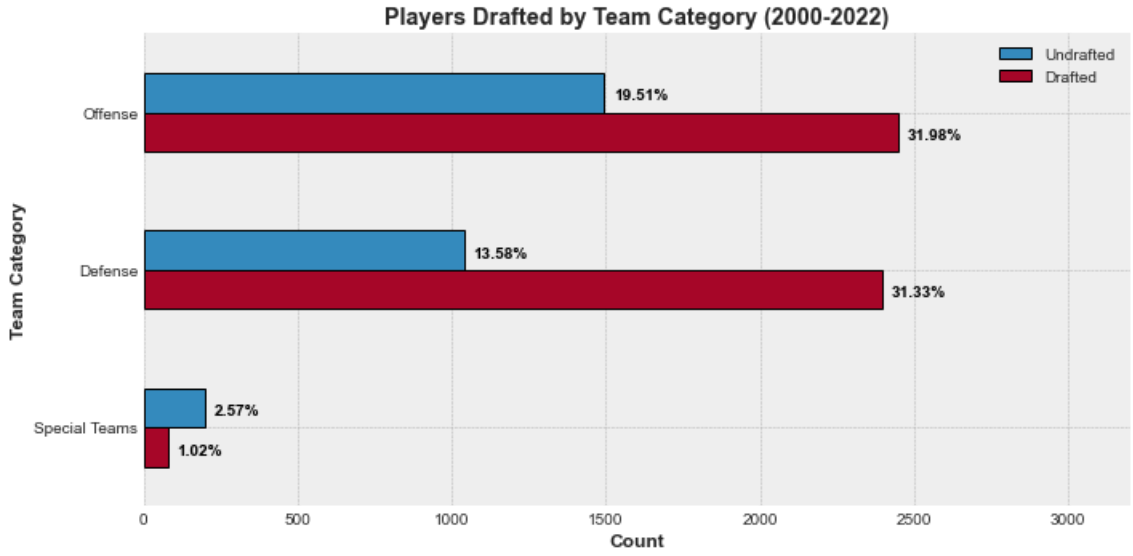
In [55]:
```
# list of offensive positions
offense_list = ['WR', 'RB', 'OT', 'TE', 'QB', 'OG', 'C', 'OL', 'FB']

# list of defensive positions
defense_list = ['CB', 'S', 'DE', 'DT', 'OLB', 'ILB', 'LB', 'DL', 'EDGE', 'DB']

# list of special team positions
special_teams_list = ['LS', 'K', 'P']
```

In [56]:
```
# split into offensive positions
offense = df.loc[df['Pos'].isin(offense_list)]

# split into defensive positions
defense = df.loc[df['Pos'].isin(defense_list)]

# split into special teams positions
special_teams = df.loc[df['Pos'].isin(special_teams_list)]
```

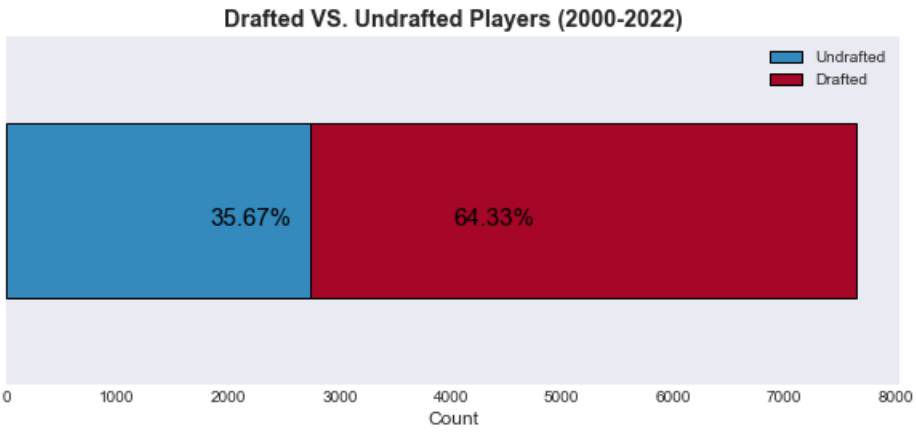
In [57]:
```
# percent and count breakdowns
# Offense
total_offense = len(offense)
total_offense_drafted = offense['Drafted'].value_counts()[1]
total_offense_undrafted = offense['Drafted'].value_counts()[0]
percent_off_drafted = round(offense['Drafted'].value_counts(normalize = True)[1], 2)
percent_off_undrafted = round(offense['Drafted'].value_counts(normalize = True)[0], 2)

print("Offense")
print(f"Total Players:      {total_offense}")
print(f"Total Drafted:      {total_offense_drafted}")
print(f"Total Undrafted:    {total_offense_undrafted}")
print(f"Percent Drafted:    {percent_off_drafted}")
print(f"Percent Undrafted:  {percent_off_undrafted}")
print("-"*25)

# Defense
total_defense = len(defense)
total_defense_drafted = defense['Drafted'].value_counts()[1]
total_defense_undrafted = defense['Drafted'].value_counts()[0]
percent_def_drafted = round(defense['Drafted'].value_counts(normalize = True)[1], 2)
percent_def_undrafted = round(defense['Drafted'].value_counts(normalize = True)[0], 2)

print("Defense")
print(f"Total Players:      {total_defense}")
print(f"Total Drafted:      {total_defense_drafted}")
print(f"Total Undrafted:    {total_defense_undrafted}")
print(f"Percent Drafted:    {percent_def_drafted}")
print(f"Percent Undrafted:  {percent_def_undrafted}")
print("-"*25)

# Special teams
total_st = len(special_teams)
total_st_drafted = special_teams['Drafted'].value_counts()[1]
total_st_undrafted = special_teams['Drafted'].value_counts()[0]
percent_st_drafted = round(special_teams['Drafted'].value_counts(normalize = True)[1], 2)
percent_st_undrafted = round(special_teams['Drafted'].value_counts(normalize = True)[0], 2)

print("Special Teams")
print(f"Total Players:      {total_st}")
print(f"Total Drafted:      {total_st_drafted}")
print(f"Total Undrafted:    {total_st_undrafted}")
print(f"Percent Drafted:    {percent_st_drafted}")
print(f"Percent Undrafted:  {percent_st_undrafted}")
```

```
Offense
Total Players:      3940
Total Drafted:      2447
Total Undrafted:    1493
Percent Drafted:    0.62
Percent Undrafted:  0.38
-------------------------
Defense
Total Players:      3436
Total Drafted:      2397
Total Undrafted:    1039
Percent Drafted:    0.7
Percent Undrafted:  0.3
-------------------------
Special Teams
Total Players:      275
Total Drafted:      78
Total Undrafted:    197
Percent Drafted:    0.28
Percent Undrafted:  0.72
```

In [58]:
```
# plot visual breakdown of drafted by team category
position_category = ['Offense', 'Defense', 'Special Teams']

data = {'Drafted': [total_offense_drafted, total_defense_drafted, total_st_drafted],
        'Undrafted': [total_offense_undrafted, total_defense_undrafted, total_st_undrafted]
       }

# create dataframe of positions & number of drafted
drafted = pd.DataFrame(data,
                       columns=['Undrafted', 'Drafted'],
                       index = position_category)

fig, ax = plt.subplots(figsize=(10, 5))
plt.style.use('bmh')

drafted.plot(kind = 'barh',
             edgecolor = "black",
             linewidth = 1,
             ax = ax, stacked = False,
             xlim = (0, 3200))

# create a list to collect the plt.patches data
totals = []

# find the values and append to list
for i in ax.patches:
    totals.append(i.get_width())

# set individual bar lables using above list
total = sum(totals)

# set individual bar lables using above list
for i in ax.patches:
    # get_width pulls left or right; get_y pushes up or down
    ax.text(i.get_width()+ 30, i.get_y() + 0.17, \
            str(round((i.get_width()/total)*100, 2))+'%',
            fontsize=10, weight = 'bold', color='black')

# invert for largest on top
ax.invert_yaxis()

plt.title('Players Drafted by Team Category (2000-2022)', weight = "bold")
plt.ylabel('Team Category', weight = "bold")
plt.xlabel('Count', weight = "bold")

plt.tight_layout()
plt.show()
fig.savefig('Images/Plots/Position_Category_Drafted.png');
```



In [59]:
```
# create dict of drafted/undrafted with sums of each category
total = ['']

data = {'Undrafted': [drafted['Undrafted'].sum()],
        'Drafted': [drafted['Drafted'].sum()]
        }

# new drafted_total df
drafted_total = pd.DataFrame(data,
                        columns=['Undrafted', 'Drafted'],
                        index = total)

plt.style.use('seaborn-dark')

fig, ax = plt.subplots(figsize=(10, 4))

drafted_total.plot(kind = 'barh',
             edgecolor = "black",
             linewidth = 1,
             ax = ax, stacked = True)

# create a list to collect the plt.patches data
totals = []

# find the values and append to list
for i in ax.patches:
    totals.append(i.get_width())

# set individual bar lables using above list
total = sum(totals)

# set individual bar lables using above list
for i in ax.patches:
    # get_width pulls left or right; get_y pushes up or down
    ax.text(i.get_width()-900, i.get_y()+.21, \
            str(round((i.get_width()/total)*100, 2))+'%',
            fontsize=15,
            color='black')

ax.set_xlabel('Count')

ax.set(ylabel=None)
ax.set_title('Drafted VS. Undrafted Players (2000-2022)', weight = "bold")
ax.legend()

plt.show()
```



In [60]:
```
# get a sense of average metrics of drafted vs undrafted players
aggs = df.groupby('Drafted').agg(['mean', 'std'])
aggs
```

Out[60]:

	Wt		40yd		Vertical		Bench		Broad Jump		3Cone		Shuttle
	mean	std	mean	std	mean	std	mean	std	mean	std	mean	std	mean
Drafted													
0	238.130451	44.166829	4.827518	0.304425	32.126826	4.183633	19.449483	6.088900	113.015686	9.188386	7.338688	0.437711	4.43802
1	245.289313	45.624052	4.748869	0.301366	33.299948	4.178520	21.419760	6.421818	115.434121	9.330657	7.256132	0.403689	4.37904

Generally, we see that players that were `Drafted` (classified as a 1), perform better on average, in the following categories:

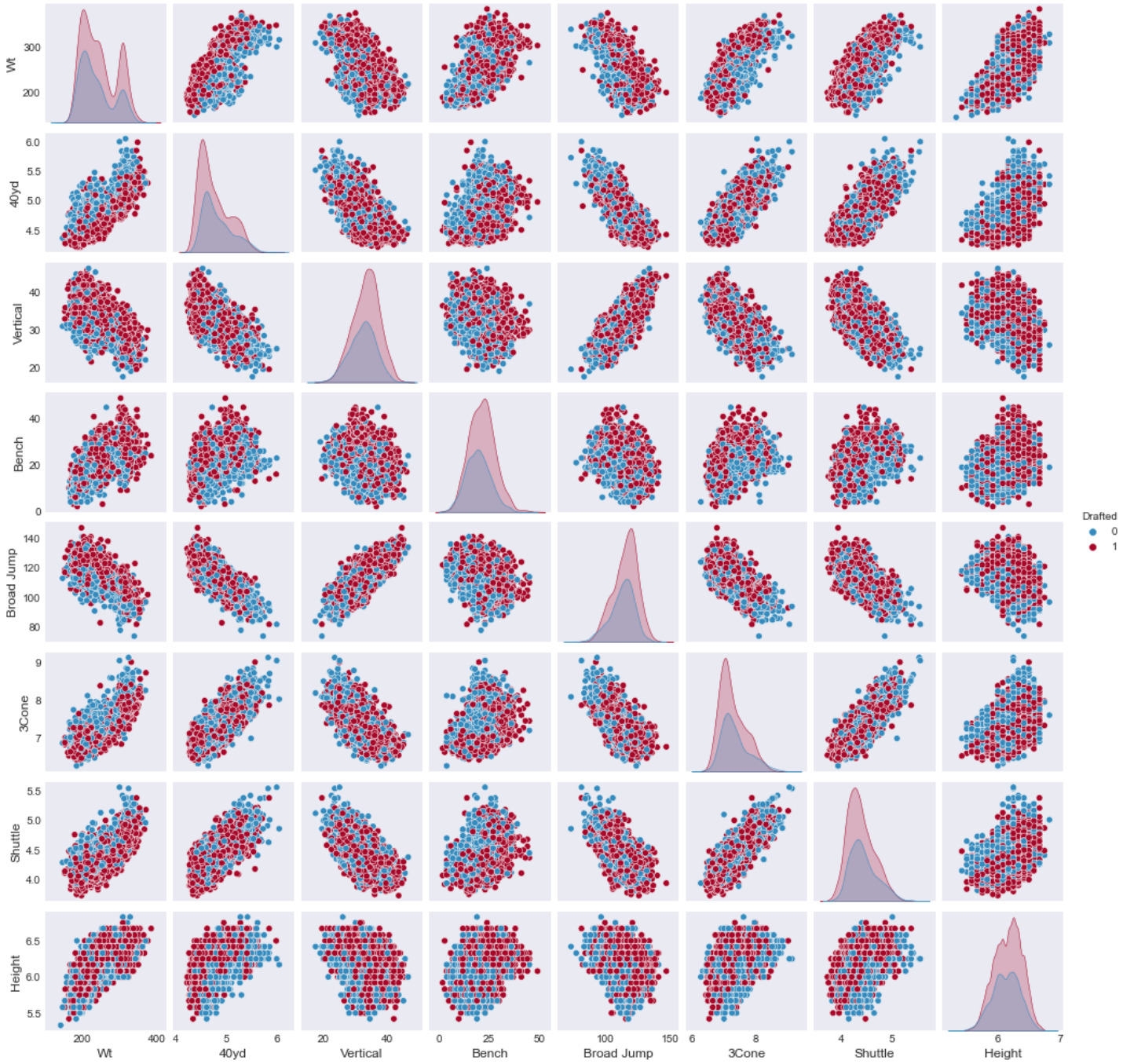
- 40 Yard, Vertical, Bench, Broad Jump, Shuttle

Interestingly, players who were `Drafted` do not perform better than those `Undrafted` in the 3 Cone drill.

We also notice that `Drafted` players are generally slightly taller and heavier as well.

In [61]:
```
# a prelim check to see classification of drafted vs undrafted for each predictor
sns.pairplot(hue = 'Drafted', data = df, height = 1.75)
```

Out[61]:
```
<seaborn.axisgrid.PairGrid at 0x7f97a8eac3a0>
```



Based on the pairplot above, we can see some clear correlations between the variables in our data. For example, `Height` and `Weight` look positively correlated as expected. `Vertical` and `Broad Jump` are also positively correlated which implies those that jump vertically higher can also jump horizontally further.

However, what it is more difficult to determine the delineation between those that were drafted and undrafted. There appears to be a lot of overlapping points and no clear grouping of the target variable `Drafted` amongst the variables.

There is also a similar histogram distribution between those who were drafted and undrafted.

4. Train-Test Split

Based on the early exploratory data analysis on the drafted vs. undrafted players, this dataset is relatively unbalanced at a 2:1 ratio. Below we will perform a Train-Test Split and specify a `stratify=y` to ensure an even balance of drafted and undrafted player in the split.

In [62]:
```
# feature selection, target variable = 'Drafted'
X = df.drop(['Drafted'], axis = 1)
y = df['Drafted']
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify = y, random_state= 42)
```

In [63]:
```
# feature selection, target variable = 'Drafted'
X_drop_school = df.drop(['Drafted', 'School'], axis = 1)
y_drop_school = df['Drafted']
X_train_drop_school, X_test_drop_school, y_train_drop_school, y_test_drop_school = train_test_split(X_drop_s
```

In [64]:
```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(5738, 10)
(1913, 10)
(5738,)
(1913,)
```

In [65]:
```
# check that percent in y_train and y_test for drafted is balanced
print("Train percent Drafted:", y_train.value_counts(normalize=True)[1])
print("Test percent Drafted: ", y_test.value_counts(normalize=True)[1])
```

```
Train percent Drafted: 0.6432554897176717
Test percent Drafted:  0.643491897543126
```

Handling Missing Values

In [66]:
```
X_train.isna().sum()
```

Out[66]:
```
Pos              0
School           0
Wt               0
40yd           344
Vertical      1330
Bench         1943
Broad Jump    1378
3Cone         2173
Shuttle       2097
Height           0
dtype: int64
```

In [67]:
```
f = (X_train['40yd'].isna().sum()/X_train.shape[0]).round(2) * 100
v = (X_train['Vertical'].isna().sum()/X_train.shape[0]).round(2) * 100
b = (X_train['Bench'].isna().sum()/X_train.shape[0]).round(2) * 100
bj = (X_train['Broad Jump'].isna().sum()/X_train.shape[0]).round(2) * 100
c = (X_train['3Cone'].isna().sum()/X_train.shape[0]).round(2) * 100
s = (X_train['Shuttle'].isna().sum()/X_train.shape[0]).round(2) * 100

print(f'{f}% of 40yd is missing.')
print(f'{v}% of Vertical is missing.')
print(f'{b}% of Bench is missing.')
print(f'{bj}% of Broad Jump is missing.')
print(f'{c}% of 3Cone is missing.')
print(f'{s}% of Shuttle is missing.')
```

```
6.0% of 40yd is missing.
23.0% of Vertical is missing.
34.0% of Bench is missing.
24.0% of Broad Jump is missing.
38.0% of 3Cone is missing.
37.0% of Shuttle is missing.
```

A large majority of values are missing in this data set as evidenced from the above. I can fill in the NaNs with the average (mean) grouped by position.

Instead of filling all missing values with the mean by each combine measurable, we will fill missing values by mean of each grouped position.

This makes sense since some players may be faster than others simply by the nature of their position and their physical build.

In [68]:
```
X_train.head()
```

Out[68]:

	Pos	School	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Height
6651	S	Kentucky	208.0	4.39	33.5	19.0	113.0	NaN	NaN	5.92
6261	DT	Fort Hays State	315.0	5.09	31.0	NaN	112.0	7.5	4.53	6.42
6174	QB	Nebraska	218.0	4.98	32.0	NaN	115.0	7.0	4.41	6.33
7742	EDGE	Cincinnati	228.0	4.67	33.0	NaN	120.0	NaN	4.37	6.42
7016	DL	Ohio State	264.0	NaN	NaN	NaN	NaN	NaN	NaN	6.42

In [69]:
```
# create a helper function for future use.
def fill_missing(df):
    """
    Helper function that takes in columns from a dataframe and fills in the NaNs
    with the means grouped by Position. Can be used for the training and test sets.
    """
    missing_cols = ['40yd', 'Vertical', 'Bench', 'Broad Jump', '3Cone', 'Shuttle']

    for col in missing_cols:
        df[col] = df[col].fillna(df.groupby('Pos')[col].transform('mean'))

    return df
```

In [70]:
```
# fill missing in X_train
fill_missing(X_train)
```

Out[70]:

	Pos	School	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Height
6651	S	Kentucky	208.0	4.3900	33.500000	19.000000	113.000000	6.994696	4.199388	5.92
6261	DT	Fort Hays State	315.0	5.0900	31.000000	27.557093	112.000000	7.500000	4.530000	6.42
6174	QB	Nebraska	218.0	4.9800	32.000000	18.647059	115.000000	7.000000	4.410000	6.33
7742	EDGE	Cincinnati	228.0	4.6700	33.000000	22.928571	120.000000	7.147778	4.370000	6.42
7016	DL	Ohio State	264.0	4.9875	30.450667	25.750000	111.291667	7.584068	4.610508	6.42
...	...	...	...	...	...	...	...	...	...	...
1599	RB	Northern Illinois	237.0	4.4900	31.000000	22.000000	114.000000	7.500000	4.150000	5.83
3257	DE	Wisconsin	266.0	4.8600	30.500000	24.000000	112.000000	7.680000	4.880000	6.42
4638	WR	Baylor	208.0	4.5200	32.500000	11.000000	119.000000	7.010000	4.320000	6.17
3229	TE	Fresno State	251.0	4.8900	26.500000	20.080645	107.000000	7.650000	4.550000	6.42
3207	CB	Hawaii	187.0	4.4700	39.500000	18.000000	129.000000	6.951101	4.165562	5.75

5738 rows × 10 columns

In [71]:
```
X_train.isna().sum()
```

Out[71]:
```
Pos             0
School          0
Wt              0
40yd            0
Vertical        0
Bench           0
Broad Jump      0
3Cone          76
Shuttle        76
Height          0
dtype: int64
```

After filling in the missing values with the means, we still have missing values for 3Cone and Shuttle drills.

Upon closer inspection, Kickers do not perform in the 3Cone and Shuttle drills. This makes sense because there really is no need for them to perform these drills for their particular skillsets.

In [72]:
```
# confirm that kickers do not perform 3Cone and Shuttle Drills
# isolate for dataframe with just Kickers
kickers = X_train[X_train['Pos'] == 'K']
display(kickers.head())
print("3Cone Mean for Kickers:", kickers['3Cone'].mean(), ", Count of NaN:", kickers['3Cone'].isna().sum())
print("Shuttle Mean for Kickers:", kickers['Shuttle'].mean(), ", Count of NaN:", kickers['Shuttle'].isna().s
```

	Pos	School	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Height
3021	K	USC	227.0	4.570000	33.5	25.0	113.5	NaN	NaN	6.17
884	K	North Dakota	190.0	4.893636	33.5	15.5	113.5	NaN	NaN	6.25
7431	K	Stanford	204.0	4.893636	33.5	15.5	113.5	NaN	NaN	6.17
6639	K	LSU	184.0	4.893636	33.5	15.5	113.5	NaN	NaN	5.83
6605	K	Oklahoma	213.0	4.893636	33.5	15.5	113.5	NaN	NaN	5.75

```
3Cone Mean for Kickers: nan , Count of NaN: 76
Shuttle Mean for Kickers: nan , Count of NaN: 76
```

There are a few options here; do we fill in the NaNs with 0? This would have a minor impact on the averages for 3Cone and Shuttle, and pull the mean closer to 0. Likely filling these values with 0s will not impact the decision whether a team would draft a kicker since they did not participate in these drills.

Another option is to drop kickers from the original `df` and just analyze all other positions. Using domain knowledge and a brief analysis on the percentage that kickers make up the dataset (as part of the Special Teams category), dropping these values will not likely impact whether a player is drafted or not. Generally speaking, very few kickers are ever drafted and are not a critical position of need in football. However, dropping kickers from the `X_train` will cause data leakage as rows will be removed from the X_train set.

We can always reanalyze the kickers and special teams category/sub-group and drop 3Cone and Shuttle columns later. To avoid data leakage, we will fill NaNs with 0s.

In [73]:
```
# fill NaNs with 0
X_train = X_train.fillna(0)
```

In [74]:
```
# check
X_train.isna().sum()
```

Out[74]:
```
Pos           0
School        0
Wt            0
40yd          0
Vertical      0
Bench         0
Broad Jump    0
3Cone         0
Shuttle       0
Height        0
dtype: int64
```

In [75]:
```
y_train.shape
```

Out[75]:
```
(5738,)
```

In [76]:
```
X_train.shape
```

Out[76]:
```
(5738, 10)
```

Apply the above cleaning steps to the X_test

In [77]:
```
# apply 'fill_missing' helper function on X_test set too
fill_missing(X_test)
```

Out[77]:

	Pos	School	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Height
1856	OT	North Carolina	291.0	5.30	27.711538	24.156250	102.475728	7.844737	4.763196	6.33
4490	OLB	UNLV	233.0	4.84	35.500000	25.000000	117.000000	6.910000	4.300000	5.92
5743	QB	Virginia Tech	232.0	4.80	26.500000	23.000000	112.000000	7.167262	4.410000	6.25
6981	WR	Miami	170.0	4.45	36.500000	14.446154	125.000000	6.973559	4.220857	5.75
5081	CB	Tulane	182.0	4.50	33.500000	9.000000	115.000000	7.200000	4.260000	5.83
...	...	...	...	...	...	...	...	...	...	...
5528	DE	Florida	239.0	4.80	34.500000	23.875000	128.000000	7.010000	4.000000	6.50
4182	OLB	Boise State	260.0	4.62	31.500000	19.000000	118.000000	7.070000	4.330000	6.25
1353	DT	Iowa	294.0	4.95	29.193182	23.000000	104.547619	7.675811	4.622632	6.42
5079	OT	Iowa	313.0	5.31	32.000000	17.000000	101.000000	7.840000	4.770000	6.50
5021	K	Notre Dame	236.0	5.17	NaN	14.000000	NaN	NaN	NaN	6.08

1913 rows × 10 columns

In [78]:
```
X_test.isna().sum()
```

Out[78]:
```
Pos            0
School         0
Wt             0
40yd           0
Vertical      27
Bench          0
Broad Jump    27
3Cone         27
Shuttle       27
Height         0
dtype: int64
```

In [79]:
```
# fill NaNs with 0
X_test = X_test.fillna(0)
X_test_drop_school = X_test_drop_school.fillna(0)
```

In [80]:
```
X_test.isna().sum()
```

Out[80]:
```
Pos           0
School        0
Wt            0
40yd          0
Vertical      0
Bench         0
Broad Jump    0
3Cone         0
Shuttle       0
Height        0
dtype: int64
```

In [81]:
```
# final describe check
# show all rows
pd.set_option('display.max_rows', None)
# check breakdown of summary statistics by Pos
X_train.groupby('Pos').describe().T
```

Out[81]:

	Pos	C	CB	DB	DE	DL	DT	EDGE	FB	ILB	K	...	
Wt	**count**	128.000000	574.000000	42.000000	384.000000	94.000000	382.000000	49.000000	92.000000	209.000000	76.000000	...	299
	mean	302.265625	192.721254	195.547619	268.828125	289.489362	307.295812	251.775510	244.445652	242.057416	197.671053	...	314
	std	8.578938	8.763262	11.555175	13.366950	23.394120	15.375234	8.708868	11.104265	7.465726	18.904947	...	12
	min	263.000000	169.000000	177.000000	234.000000	248.000000	268.000000	228.000000	220.000000	226.000000	144.000000	...	282
	25%	299.000000	187.000000	186.000000	260.000000	266.500000	298.000000	248.000000	238.750000	237.000000	184.750000	...	306
	50%	302.000000	192.000000	195.000000	268.000000	290.000000	306.000000	253.000000	245.000000	242.000000	193.000000	...	313
	75%	307.250000	198.000000	204.750000	277.000000	306.750000	314.750000	256.000000	250.250000	246.000000	210.250000	...	320
	max	327.000000	221.000000	226.000000	310.000000	342.000000	369.000000	271.000000	281.000000	273.000000	260.000000	...	364
40yd	**count**	128.000000	574.000000	42.000000	384.000000	94.000000	382.000000	49.000000	92.000000	209.000000	76.000000	...	299
	mean	5.224370	4.495394	4.521500	4.835416	4.987500	5.095973	4.701538	4.770220	4.756617	4.893636	...	5
	std	0.162605	0.093876	0.113687	0.143302	0.199617	0.165819	0.138196	0.143136	0.121914	0.104895	...	0

5. Pipeline

Now that the `X_train` and `X_test` sets are ready to be modeled, we'll create a pipeline that will streamline the preprocessing and modeling efforts.

In [82]:
```
from sklearn.pipeline import Pipeline

# this will allow us to visualize the pipeline (may not be available in learn-env)
from sklearn import set_config
set_config(display= 'diagram')
```

In [83]:
```
# reset display to normal
pd.set_option('display.max_rows', 500)
X_train.head()
```

Out[83]:

	Pos	School	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Height
6651	S	Kentucky	208.0	4.3900	33.500000	19.000000	113.000000	6.994696	4.199388	5.92
6261	DT	Fort Hays State	315.0	5.0900	31.000000	27.557093	112.000000	7.500000	4.530000	6.42
6174	QB	Nebraska	218.0	4.9800	32.000000	18.647059	115.000000	7.000000	4.410000	6.33
7742	EDGE	Cincinnati	228.0	4.6700	33.000000	22.928571	120.000000	7.147778	4.370000	6.42
7016	DL	Ohio State	264.0	4.9875	30.450667	25.750000	111.291667	7.584068	4.610508	6.42

First, we'll define which columns to treat as numeric and categorical.

In [84]:
```
# define numeric columns to be standard scaled
num_cols = X_train.select_dtypes(['int', 'float']).columns
```

In [85]:
```
# define categorical columns to be OHE
cat_cols = X_train.select_dtypes(['object']).columns
```

Define the seperate pipelines

We'll only need to create two seperate pipelines that will handle the numerical and categorical columns. We'll apply a standard scaler on the numerical columns and we'll one-hot encode and apply a MaxAbsScaler to the categorical values to normalize the dataset.

Numeric & Nominal Pipelines

In [86]:
```
# define numeric transformation pipeline that scales the numbers
numeric_pipeline = Pipeline([('numnorm', StandardScaler())]) # apply a standard scaler

# define a nominal transformation pipeline that OHE the cats, and MaxAbsScales the set
nominal_pipeline = Pipeline([('onehotenc', OneHotEncoder(categories="auto", # ohe the cat variables
                                  sparse = False,
                                  handle_unknown = 'ignore')), # keep as ignore since unbalanced 'School' valu
                             ('onehotnorm', MaxAbsScaler())]) # apply a MaxAbsScaler
```

Unite the numeric and nominal pipelines into the ColumnTransformer.

In [87]:
```
# transform the selected columns with nominal, and numeric pipelines
ct = ColumnTransformer(transformers =
    [("nominalpipe", nominal_pipeline, cat_cols), # ohe and MaxAbsScale the Pos and School variables
     ("numpipe", numeric_pipeline, num_cols)])
```

In [88]:
```
# inspect the transformed X_Train
pp_X_train = pd.DataFrame(ct.fit_transform(X_train))
pp_X_train.head()
```

Out[88]:

	0	1	2	3	4	5	6	7	8	9	...	340	341	342	343	344	345	346	347	348	
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	-0.768205	-1.293342	0.167570	-0.188488	-0.166420	-0.195421	-0.236988	-1.03
1	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	1.587333	1.031950	-0.473641	1.288138	-0.280893	0.358288	0.356745	1.23
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	-0.548061	0.666547	-0.217157	-0.249392	0.062525	-0.189610	0.141242	0.82
3	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	...	0.0	0.0	-0.327917	-0.363226	0.039327	0.489433	0.634888	-0.027675	0.069408	1.23
4	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.464600	0.691461	-0.614537	0.976303	-0.361978	0.450410	0.501327	1.23

5 rows × 350 columns

For later use, we'll need to know which columns or features are most important during the modeling. Below is code to get those feature names after OHE the categorical columns.

In [89]:
```
# confirm how many features we have
len(X_train['Pos'].unique()) + len(X_train['School'].unique())
```

Out[89]: 342

In [90]:
```
# get feature names after encoding
feature_names = list(nominal_pipeline.named_steps['onehotenc'].fit(X_train[cat_cols]).get_feature_names())

# confirm length of features
len(feature_names)
```

Out[90]: 342

In [91]:
```
# removes the OHE strings at front end of feature names
def clean_features(lst):
    new_list = []
    for value in lst:
        splitted = value.split('_')[1] # returns name of feature after '_'
        new_list.append(splitted)
        continue
    return new_list
```

In [92]:
```
# apply function to clean feature_names
cleaned_feature_names = clean_features(feature_names)
```

In [93]:
```
# preview cleaned feature names
cleaned_feature_names[:5]
```

Out[93]: ['C', 'CB', 'DB', 'DE', 'DL']

In [94]:
```
# preview first and last feature column names
print(cleaned_feature_names[:5])
print(cleaned_feature_names[-5:])
```

```
['C', 'CB', 'DB', 'DE', 'DL']
['Wisconsin', 'Wisconsin-Whitewater', 'Wyoming', 'Yale', 'Youngstown State']
```

In [95]:
```
# get list of numerical cols
num_features_names = list(num_cols)
num_features_names
```

Out[95]: ['Wt', '40yd', 'Vertical', 'Bench', 'Broad Jump', '3Cone', 'Shuttle', 'Height']

In [96]:
```
# put into a dataframe
feature_names_df = pd.DataFrame(cleaned_feature_names)
# add the numerical cols at end of dataframe
feature_names_df = feature_names_df.append(num_features_names)
```

In [97]:
```
# reset the index
feature_names_df = feature_names_df.reset_index().drop(columns = 'index')
```

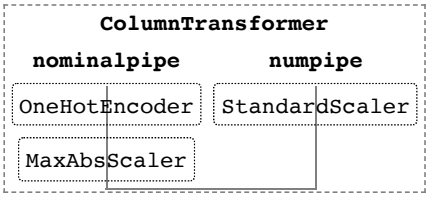
In [98]:
```
# check that numericals are appended to end of categorical df
feature_names_df.tail(10)
```

Out[98]:

	0
340	Yale
341	Youngstown State
342	Wt
343	40yd
344	Vertical
345	Bench
346	Broad Jump
347	3Cone
348	Shuttle
349	Height

In [99]:
```
# check columntransformer
ct
```

Out[99]:



6. Model Selection

1st Model - Baseline

We will now go through an iterative process to determine which model suits the dataset best when it comes to classification of those who were drafted or not. As a reminder, we want to have a better baseline accuracy score of about 63% (those who were Drafted) if our model was just simply randomly guessing.

However, the main metric we'll be focussed on here is the **F1-Score** which takes into consideration a balance between the False Positives and False Negatives into account. In other words, we want to have a harmonized balance between precision and recall and take into consideration the False Positives and False Negatives.

- False Positives are players who were labeled as 'Drafted' but they were actually 'Undrafted'
- False Negatives are players who were labeled as 'Undrafted' but they were actually 'Drafted.

In [100]:
```
# percentage of those drafted and undrafted
y_train.value_counts(normalize = True)
```

Out[100]:
```
1    0.643255
0    0.356745
Name: Drafted, dtype: float64
```

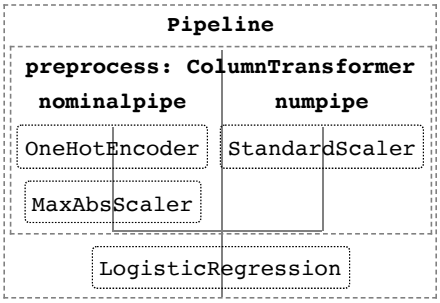
In [101]:
```
# to hide warnings
from sklearn.utils.testing import ignore_warnings
from sklearn.exceptions import ConvergenceWarning

# to set style for confusion matrices
plt.style.use("seaborn-dark")
```

Logistic Regression Model

In [102]:
```
# build baseline log reg pipepline
steps = [('preprocess', ct),
         ('model', LogisticRegression(random_state = 42))]

base_log_reg_pipeline = Pipeline(steps)

base_log_reg_pipeline.fit(X_train, y_train)
```

```
/Users/eric/opt/anaconda3/envs/learn-env/lib/python3.8/site-packages/sklearn/linear_model/_logistic.py:76
2: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-learn.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression)
  n_iter_i = _check_optimize_result(
```

Out[102]:

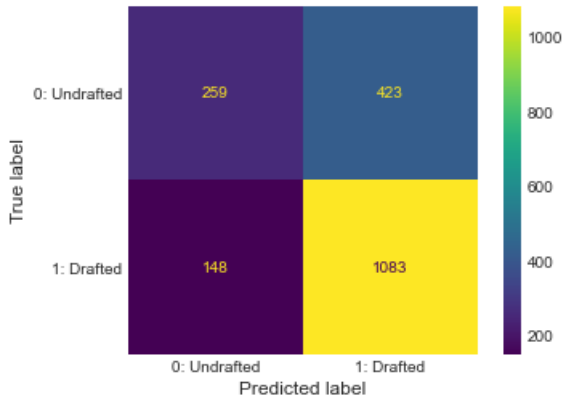


In [103]:
```
# helper function to get AUC score
def get_auc(model, X, y):
    # get y_prob
#     X_trans = ct.transform(X)
    y_prob = model.predict_proba(X)
    score = roc_auc_score(y, y_prob[:,1])
    return round(score,3)
```

In [104]:
```
# create a tracking list of f1 and AUC scores
f1_train_list = []
f1_test_list = []
auc_list = []
```

In [105]:
```
# define class_names for confusion matrix label moving forward
class_names = ['0: Undrafted', '1: Drafted']

# plot baseline log reg matrix on testing set
plot_confusion_matrix(base_log_reg_pipeline,
                      X_test, y_test,
                      display_labels = class_names)

# get f1 scores and AUC
f1_train1 = round(f1_score(y_train, base_log_reg_pipeline.predict(X_train)),4)
f1_test1 = round(f1_score(y_test, base_log_reg_pipeline.predict(X_test)),4)
auc1 = get_auc(base_log_reg_pipeline, X_test, y_test)

# append to scores list
f1_train_list.append(f1_train1)
f1_test_list.append(f1_test1)
auc_list.append(auc1)

print('1st Model - Baseline Logistic Regression F1 Score (Training):', f1_train1)
print('1st Model - Baseline Logistic Regression F1 Score (Test):', f1_test1)
print('AUC Test Score:', auc1)
print('-'*80)

# get classification report on the test set
y_pred = base_log_reg_pipeline.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
1st Model - Baseline Logistic Regression F1 Score (Training): 0.8106
1st Model - Baseline Logistic Regression F1 Score (Test): 0.7914
AUC Test Score: 0.724
--------------------------------------------------------------------------------
              precision    recall  f1-score   support

           0       0.64      0.38      0.48       682
           1       0.72      0.88      0.79      1231

    accuracy                           0.70      1913
   macro avg       0.68      0.63      0.63      1913
weighted avg       0.69      0.70      0.68      1913
```



Cross Validation

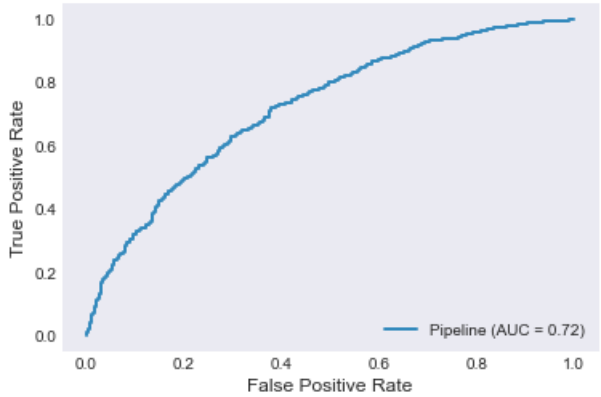
In [106]:
```
# define a simple function that returns cross validation score for a 5 fold
# scores with f1
def get_cv_score(model, X, y):
    cv_score = np.mean(cross_val_score(model, X, y, scoring = 'f1', cv = 5))
    return round(cv_score, 4)
```

In [107]:
```
get_cv_score(base_log_reg_pipeline, X_train, y_train)
```

```
/Users/eric/opt/anaconda3/envs/learn-env/lib/python3.8/site-packages/sklearn/linear_model/_logistic.py:76
2: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-learn.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression)
  n_iter_i = _check_optimize_result(
/Users/eric/opt/anaconda3/envs/learn-env/lib/python3.8/site-packages/sklearn/linear_model/_logistic.py:76
2: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-learn.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
```

In [108]:
```
# fit and plot roc curve
base_log_reg_pipeline.fit(X_train, y_train)
plot_roc_curve(base_log_reg_pipeline, X_test, y_test)
plt.show()
```

```
/Users/eric/opt/anaconda3/envs/learn-env/lib/python3.8/site-packages/sklearn/linear_model/_logistic.py:76
2: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-learn.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression)
  n_iter_i = _check_optimize_result(
```



2nd Model - Tuned Logistic Regression

Lets define a new model and try to improve on the baseline logistic regression model and address the convergence and max iterations warnings.

In [109]:
```
# build baseline log reg pipepline
steps = [('preprocess', ct),
         ('model', LogisticRegression(random_state = 42))]

base_log_reg_pipeline = Pipeline(steps)

# define new parameters in the grid
logreg_pipe_grid = {'model__penalty': ['l2', 'none'],
              'model__solver': ['lbfgs'],
              'model__C': [0.0001, 0.001, 0.01, 0.1, 1],
              'model__max_iter': [1e2, 1e3, 1e4, 1e5]}

# grid search best f1 score
gs_logreg_pipe2 = GridSearchCV(estimator = base_log_reg_pipeline,
                               param_grid = logreg_pipe_grid,
                               scoring = 'f1')
```

To save time, below is an example of the grid search in action. Just need to uncomment out the cells. Otherwise, the best parameters have been performed and extracted to fit the 2nd model using the `lbfgs` solver.

In [110]:
```
# run this to find best parameters based on gridsearch
# takes a while!
# gs_logreg_pipe2.fit(X_train, y_train)
```

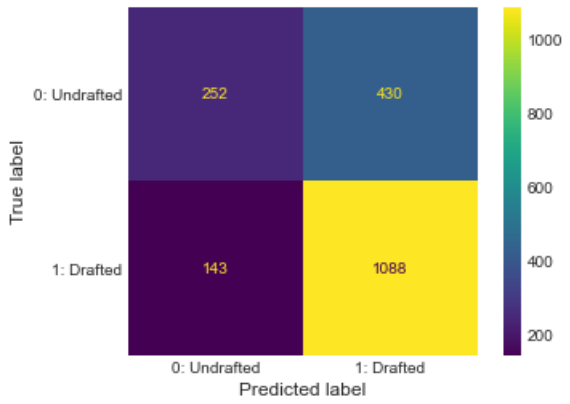
In [111]:
```
# find my best parameters in the log reg model 2, using lbfgs solver
# ---- to save time; below are best results ----- #
# {'model__C': 0.1,
#  'model__max_iter': 100.0,
#  'model__penalty': 'l2',
#  'model__solver': 'lbfgs'}

#gs_logreg_pipe2.best_params_
```

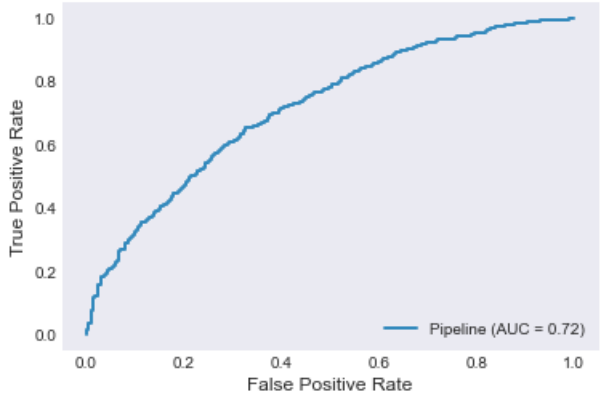
In [112]:
```
# whats the best score based on these parameters? using lbfgs solver
# gs_logreg_pipe2.best_score_
```

In [113]:
```
# reassign best params to the 2nd model
# gs_logreg_pipe2 = gs_logreg_pipe2.best_estimator_
# gs_logreg_pipe2
```

In [114]:
```
# take best params from above, and redefine in steps for pipeline
model_2_steps = [('preprocess', ct),
                 ('model', LogisticRegression(random_state = 42, # input best params from grid search
                                              C = 0.1,
                                              max_iter = 100,
                                              penalty = 'l2',
                                              solver = 'lbfgs'))]

gs_logreg_pipe2 = Pipeline(model_2_steps)

gs_logreg_pipe2.fit(X_train, y_train)

# plot confusion matrix
plot_confusion_matrix(gs_logreg_pipe2,
                      X_test, y_test,
                      display_labels = class_names)

# get f1 scores and AUC
f1_train2 = round(f1_score(y_train, gs_logreg_pipe2.predict(X_train)),4)
f1_test2 = round(f1_score(y_test, gs_logreg_pipe2.predict(X_test)),4)
auc2 = get_auc(gs_logreg_pipe2, X_test, y_test)

# append to scores list
f1_train_list.append(f1_train2)
f1_test_list.append(f1_test2)
auc_list.append(auc2)

print('2nd Model - Best Params (lbfgs) Logistic Regression F1 Score (Training):', f1_train2)
print('2nd Model - Best Params (lbfgs) Logistic Regression F1 Score (Test):', f1_test2)
print('AUC Test Score:', auc2)
print()
print('Cross Validation Score:', get_cv_score(gs_logreg_pipe2, X_train, y_train))
print('-'*80)

# get classification report on the test set
y_pred = gs_logreg_pipe2.predict(X_test)
print(classification_report(y_test, y_pred))

# plot roc curve
plot_roc_curve(gs_logreg_pipe2, X_test, y_test)
plt.show()
```

```
2nd Model - Best Params (lbfgs) Logistic Regression F1 Score (Training): 0.8065
2nd Model - Best Params (lbfgs) Logistic Regression F1 Score (Test): 0.7916
AUC Test Score: 0.718

Cross Validation Score: 0.7975
--------------------------------------------------------------------------------
              precision    recall  f1-score   support

           0       0.64      0.37      0.47       682
           1       0.72      0.88      0.79      1231

    accuracy                           0.70      1913
   macro avg       0.68      0.63      0.63      1913
weighted avg       0.69      0.70      0.68      1913
```





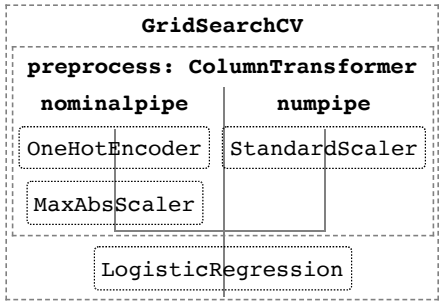
3rd Model - Try different Solvers

While we fine tuned the original basesline model, lets now run a grid search on the available logistic regression solvers and determine the optimum solver based on the `C` = 0.1, and `max_iter` = 100.

Run the Grid Search:

In [115]:
```
# build new pipeline
steps = [('preprocess', ct),
         ('model', LogisticRegression(random_state = 42))]

base_log_reg_pipeline = Pipeline(steps)

# define new parameters in the grid
solvers_pipe_grid = {'model__solver': ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'],
                     'model__C': [0.1],
                     'model__max_iter': [1e2]}

# scoring on F1
best_solver_pipe3 = GridSearchCV(estimator = base_log_reg_pipeline,
                                 param_grid = solvers_pipe_grid,
                                 scoring = 'f1')

# run this to find best parameters based on gridsearch
best_solver_pipe3.fit(X_train, y_train)
```

Out[115]:



In [116]:
```
display(best_solver_pipe3.best_params_)
display(best_solver_pipe3.best_score_)
```

```
{'model__C': 0.1, 'model__max_iter': 100.0, 'model__solver': 'liblinear'}
```

```
0.7977458684447366
```

So based on the grid search above, our best solver is the `liblinear`. Lets now apply this model with this solver.

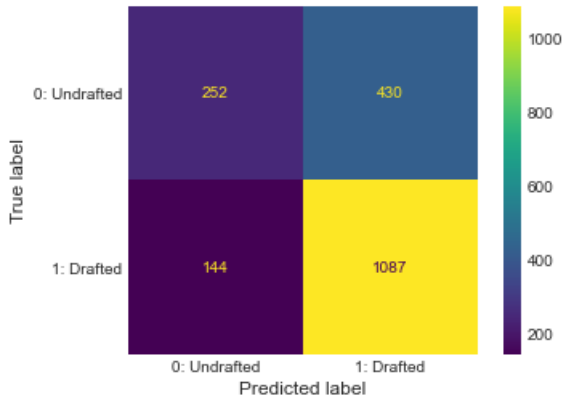
Using the `liblinear` model

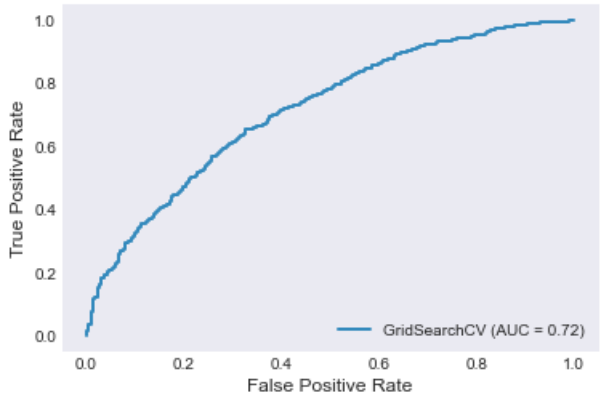
In [117]:
```
# build log reg liblinear pipeline
steps = [('preprocess', ct),
         ('model', LogisticRegression(random_state = 42))]

base_log_reg_pipeline = Pipeline(steps)

# define new parameters in the grid
best_solver_pipe_grid = {'model__solver': ['liblinear'],
                         'model__penalty': ['l1', 'l2'],
                         'model__C': [0.01],
                         'model__max_iter': [1e2]}

# run this to find best parameters based on gridsearch
best_solver_pipe3.fit(X_train, y_train)

display(best_solver_pipe3.best_params_)
display(best_solver_pipe3.best_score_)
```

```
{'model__C': 0.1, 'model__max_iter': 100.0, 'model__solver': 'liblinear'}

0.7977458684447366
```

In [118]:
```
# plot matrix
plot_confusion_matrix(best_solver_pipe3,
                      X_test, y_test,
                      display_labels = class_names)

f1_train3 = round(f1_score(y_train, best_solver_pipe3.predict(X_train)),4)
f1_test3 = round(f1_score(y_test, best_solver_pipe3.predict(X_test)),4)
auc3 = get_auc(best_solver_pipe3, X_test, y_test)

# get scores
print('3rd Model -  Liblinear F1 Score: (Training):', f1_train3)
print('3rd Model - Liblinear F1 Score (Test):', f1_test3)
print('AUC Test Score:', auc3)
print()
print('Cross Validation Score:', get_cv_score(best_solver_pipe3, X_train, y_train))
print('-'*80)

# append to scores list
f1_train_list.append(f1_train3)
f1_test_list.append(f1_test3)
auc_list.append(auc3)

# get classification report on the test set
y_pred = best_solver_pipe3.predict(X_test)
print(classification_report(y_test, y_pred))

# plot roc curve
plot_roc_curve(best_solver_pipe3, X_test, y_test)
plt.show()
```

```
3rd Model -  Liblinear F1 Score: (Training): 0.8067
3rd Model - Liblinear F1 Score (Test): 0.7911
AUC Test Score: 0.718

Cross Validation Score: 0.7973
--------------------------------------------------------------------------------
              precision    recall  f1-score   support

           0       0.64      0.37      0.47       682
           1       0.72      0.88      0.79      1231

    accuracy                           0.70      1913
   macro avg       0.68      0.63      0.63      1913
weighted avg       0.69      0.70      0.68      1913
```





Conclusions:

- After running a few logistic regression models, we arent really performing better than the baseline model. While we addressed the `max_iter` and `Convergence` issues, the baseline logistic regression model performed the best on the test set thus far.

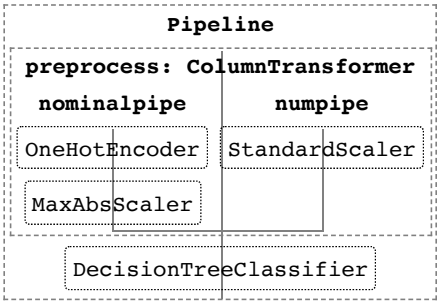
Lets now move on and try different classifier types.

4th Model - Decision Tree Classifier (DCT)

Note: While we do not need to scale or OHE the X_train for a Decision Tree Classifier, we'll just leave it in the pipeline since it doesn't affect the model's performance.

In [119]:
```
# build new DCT pipeline
steps = [('preprocess', ct),
         ('dt_clf', DecisionTreeClassifier (random_state = 42))]

dt_clf_pipe4 = Pipeline(steps)

dt_clf_pipe4.fit(X_train,y_train)
```

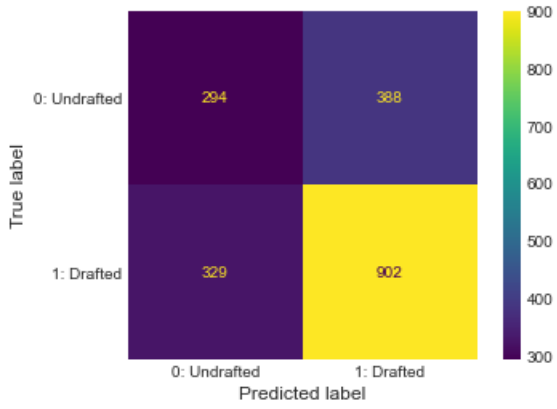
Out[119]:

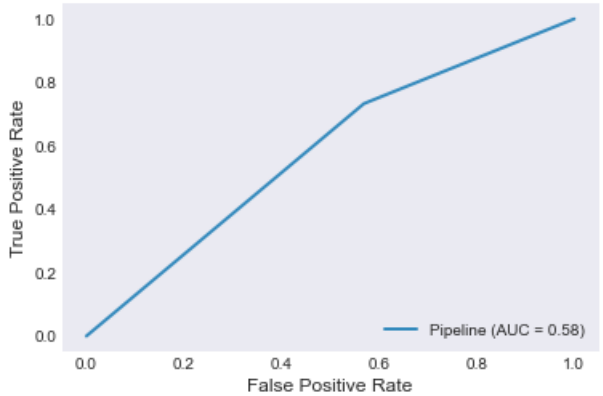


We know that decision tree classifiers naturally overfits, so lets check how the model will perform on the test size and cross validation.

In [120]:
```
# plot matrix
plot_confusion_matrix(dt_clf_pipe4,
                      X_test, y_test,
                      display_labels = class_names)

f1_train4 = round(f1_score(y_train, dt_clf_pipe4.predict(X_train)),4)
f1_test4 = round(f1_score(y_test, dt_clf_pipe4.predict(X_test)),4)
auc4 = get_auc(dt_clf_pipe4, X_test, y_test)

# get scores
print('4th Model - Decision Tree F1 Score (Training):', f1_train4)
print('4th Model - Decision Tree F1 Score (Test):', f1_test4)
print('AUC Test Score:', auc4)
print()
print('Cross Validation Score:', get_cv_score(dt_clf_pipe4, X_train, y_train))
print('-'*80)

# append to scores list
f1_train_list.append(f1_train4)
f1_test_list.append(f1_test4)
auc_list.append(auc4)

# get classification report on the test set
y_pred = dt_clf_pipe4.predict(X_test)
print(classification_report(y_test, y_pred))

# plot roc curve
plot_roc_curve(dt_clf_pipe4, X_test, y_test)
plt.show()
```

```
4th Model - Decision Tree F1 Score (Training): 1.0
4th Model - Decision Tree F1 Score (Test): 0.7156
AUC Test Score: 0.582

Cross Validation Score: 0.7233
--------------------------------------------------------------------------------
              precision    recall  f1-score   support

           0       0.47      0.43      0.45       682
           1       0.70      0.73      0.72      1231

    accuracy                           0.63      1913
   macro avg       0.59      0.58      0.58      1913
weighted avg       0.62      0.63      0.62      1913
```





4B Model - Pruning the Decision Tree Classifier (DCT)

Clearly the DCT needs to be worked with. Lets now run a grid search on the optimum hyperparameters for the DCT.

The below pipeline fills in the optimum score for the following grid search values:

```
dt_pipe_grid = {'dt_clf__criterion': ['gini', 'entropy'],
                'dt_clf__max_depth': [1, 10, 20],
                'dt_clf__min_samples_split': [ 2, 3, 5, 10],
                'dt_clf__min_samples_leaf': [ 2, 3, 5, 10]}
```

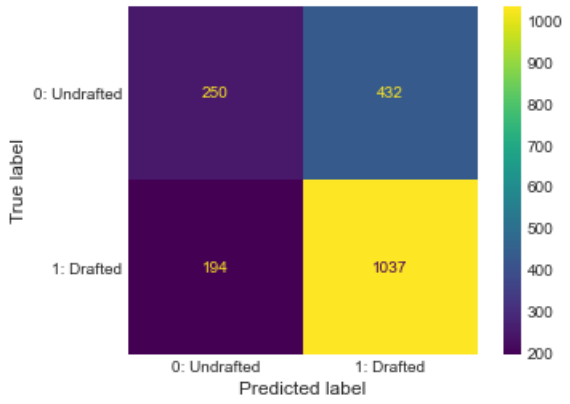
In [121]:
```
# build pipeline
steps = [('preprocess', ct),
         ('dt_clf', DecisionTreeClassifier(random_state = 42))]

dt_clf_pipe4b = Pipeline(steps)

dt_clf_pipe4b.fit(X_train, y_train)

# define new parameters in the grid
dt_pipe_grid = {'dt_clf__criterion': ['entropy'],
                'dt_clf__max_depth': [10],
                'dt_clf__min_samples_split': [10],
                'dt_clf__min_samples_leaf': [2]}

# grid search with default accuracy scoring, because error with f1 scoring
dt_clf_pipe4b = GridSearchCV(estimator = dt_clf_pipe4b,
                             param_grid = dt_pipe_grid)

# run this to find best parameters based on gridsearch
dt_clf_pipe4b.fit(X_train, y_train)

# get best scores and parameters
display(dt_clf_pipe4b.best_params_)
display(dt_clf_pipe4b.best_score_)
```

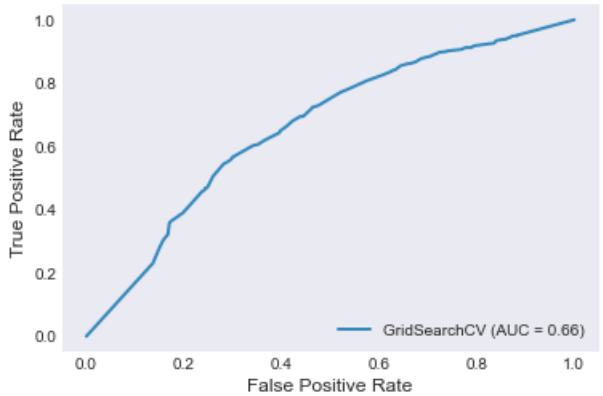
```
{'dt_clf__criterion': 'entropy',
 'dt_clf__max_depth': 10,
 'dt_clf__min_samples_leaf': 2,
 'dt_clf__min_samples_split': 10}
```

```
0.6695674825100474
```

In [122]:
```
# plot matrix
plot_confusion_matrix(dt_clf_pipe4b,
                      X_test, y_test,
                      display_labels = class_names)

f1_train4b = round(f1_score(y_train, dt_clf_pipe4b.predict(X_train)),4)
f1_test4b = round(f1_score(y_test, dt_clf_pipe4b.predict(X_test)),4)
auc4b = get_auc(dt_clf_pipe4b, X_test, y_test)

# append to scores list
f1_train_list.append(f1_train4b)
f1_test_list.append(f1_test4b)
auc_list.append(auc4b)

# get scores
print('4B Model - Decision Tree F1 Score (Training):', f1_train4b)
print('4B Model - Decision Tree F1 Score (Test):', f1_test4b)
print('AUC Test Score:', auc4b)
print()
print('Cross Validation Score:', get_cv_score(dt_clf_pipe4b, X_train, y_train))
print('-'*80)

# get classification report on the test set
y_pred = dt_clf_pipe4b.predict(X_test)
print(classification_report(y_test, y_pred))

# plot roc curve
plot_roc_curve(dt_clf_pipe4b, X_test, y_test)
plt.show()
```

```
4B Model - Decision Tree F1 Score (Training): 0.8411
4B Model - Decision Tree F1 Score (Test): 0.7681
AUC Test Score: 0.662

Cross Validation Score: 0.756
--------------------------------------------------------------------------------
              precision    recall  f1-score   support

           0       0.56      0.37      0.44       682
           1       0.71      0.84      0.77      1231

    accuracy                           0.67      1913
   macro avg       0.63      0.60      0.61      1913
weighted avg       0.65      0.67      0.65      1913
```





5th Model - KNN Classifier

The below pipeline fills in the optimum score for the following grid search values:

```
knn_pipe_grid = {'knn__n_neighbors': [21, 31, 41],
                 'knn__p': [1, 2, 3, 4]}
```

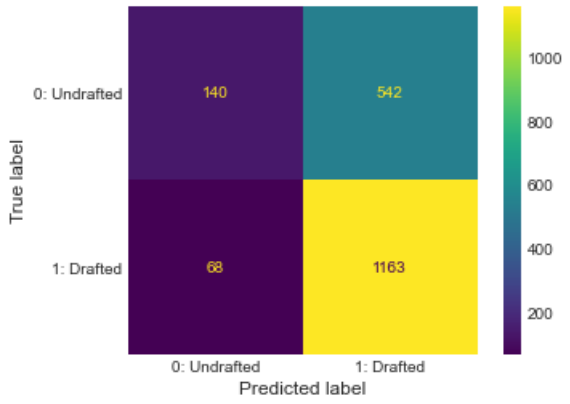
In [123]:
```
# build pipeline
steps = [('preprocess', ct),
         ('knn', KNeighborsClassifier())]

knn_pipe = Pipeline(steps)

knn_pipe.fit(X_train, y_train)

# define new parameters in the grid
# takes a while to run!
knn_pipe_grid = {'knn__n_neighbors': [41],
                 'knn__p': [1]}

# grid search with f1 scoring
knn_pipe = GridSearchCV(estimator = knn_pipe,
                        param_grid = knn_pipe_grid,
                        scoring = 'f1')

# run this to find best parameters based on gridsearch
knn_pipe.fit(X_train, y_train)

# get best scores and parameters
display(knn_pipe.best_params_)
display(knn_pipe.best_score_)
```

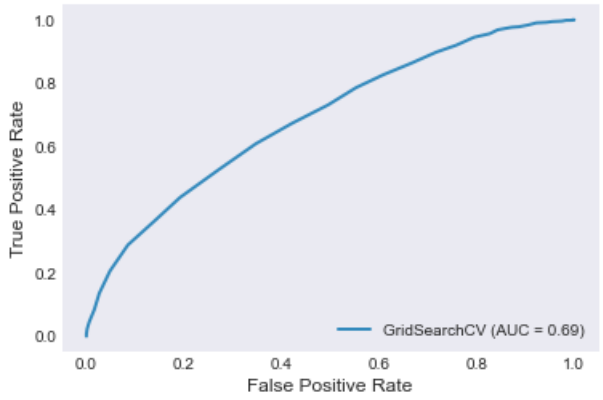
```
{'knn__n_neighbors': 41, 'knn__p': 1}
```

```
0.7947886998945844
```

In [124]:
```
# plot matrix
plot_confusion_matrix(knn_pipe,
                      X_test, y_test,
                      display_labels = class_names)

f1_train5 = round(f1_score(y_train, knn_pipe.predict(X_train)),4)
f1_test5 = round(f1_score(y_test, knn_pipe.predict(X_test)),4)
auc5 = get_auc(knn_pipe, X_test, y_test)

# append to scores list
f1_train_list.append(f1_train5)
f1_test_list.append(f1_test5)
auc_list.append(auc5)

# get scores
print('5th Model - KNN F1 Score (Training):', f1_train5)
print('5th Model - KNN F1 Score (Test):', f1_test5)
print('AUC Test Score:', auc5)
print()
print('Cross Validation Score:', get_cv_score(knn_pipe, X_train, y_train))
print('-'*80)

# get classification report on the test set
y_pred = knn_pipe.predict(X_test)
print(classification_report(y_test, y_pred))

# plot roc curve
plot_roc_curve(knn_pipe, X_test, y_test)
plt.show()
```

```
5th Model - KNN F1 Score (Training): 0.8042
5th Model - KNN F1 Score (Test): 0.7922
AUC Test Score: 0.689

Cross Validation Score: 0.7948
--------------------------------------------------------------------------------
              precision    recall  f1-score   support

           0       0.67      0.21      0.31       682
           1       0.68      0.94      0.79      1231

    accuracy                           0.68      1913
   macro avg       0.68      0.58      0.55      1913
weighted avg       0.68      0.68      0.62      1913
```





6th Model - Adaboost

The below pipeline fills in the optimum score for the following grid search values:

```
param_grid = {
    'ab_clf__n_estimators': [10, 50, 100],
    'ab_clf__base_estimator__max_depth': [2, 5, 10]
}
```

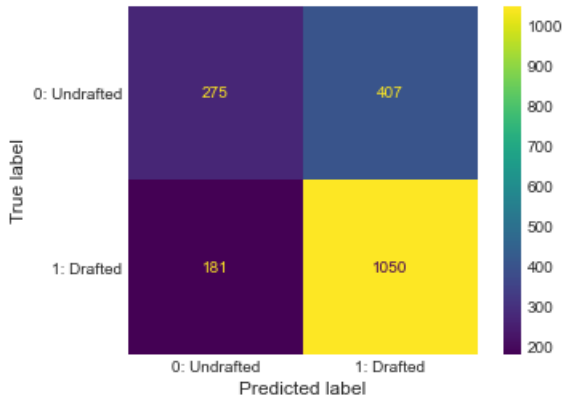
In [125]:
```
# Build pipeline
ab_steps = [('preprocess', ct),
         ('ab_clf', AdaBoostClassifier(base_estimator = DecisionTreeClassifier(),
                                       random_state = 42))]

ab_clf = Pipeline(ab_steps)

ab_clf.fit(X_train, y_train)

param_grid = {
    'ab_clf__n_estimators': [10],
    'ab_clf__base_estimator__max_depth': [2]
}

# find best param based on f1 score
ab_clf = GridSearchCV(estimator = ab_clf,
                      param_grid = param_grid,
                      scoring = 'f1') # goes through all parameters in the defined grid

# run this to find best parameters based on gridsearch
ab_clf.fit(X_train, y_train)

# get best scores and parameters
display(ab_clf.best_params_)
display(ab_clf.best_score_)
```

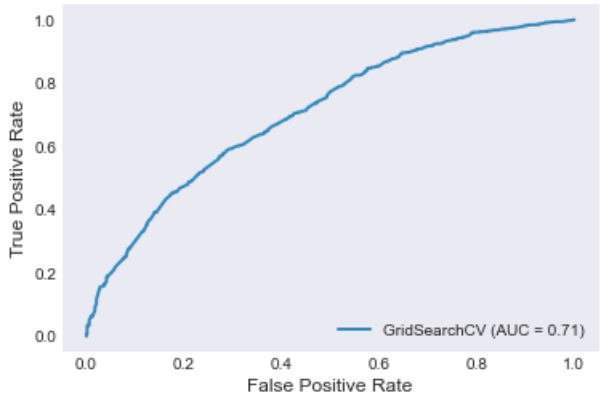
```
{'ab_clf__base_estimator__max_depth': 2, 'ab_clf__n_estimators': 10}
```

```
0.7827364303639507
```

In [126]:
```
# plot matrix
plot_confusion_matrix(ab_clf,
                      X_test, y_test,
                      display_labels = class_names)

f1_train6 = round(f1_score(y_train, ab_clf.predict(X_train)),4)
f1_test6 = round(f1_score(y_test, ab_clf.predict(X_test)),4)
auc6 = get_auc(ab_clf, X_test, y_test)

# append to scores list
f1_train_list.append(f1_train6)
f1_test_list.append(f1_test6)
auc_list.append(auc6)

# get scores
print('6th Model - Adaboost F1 Score (Training):', f1_train6)
print('6th Model - Adaboost F1 Score (Test):', f1_test6)
print('AUC Test Score:', auc6)
print()
print('Cross Validation Score:', get_cv_score(ab_clf, X_train, y_train))
print('-'*80)

# get classification report on the test set
y_pred = ab_clf.predict(X_test)
print(classification_report(y_test, y_pred))

# plot roc curve
plot_roc_curve(ab_clf, X_test, y_test)
plt.show()
```

```
6th Model - Adaboost F1 Score (Training): 0.7958
6th Model - Adaboost F1 Score (Test): 0.7813
AUC Test Score: 0.708

Cross Validation Score: 0.7827
--------------------------------------------------------------------------------
              precision    recall  f1-score   support

           0       0.60      0.40      0.48       682
           1       0.72      0.85      0.78      1231

    accuracy                           0.69      1913
   macro avg       0.66      0.63      0.63      1913
weighted avg       0.68      0.69      0.68      1913
```





7th Model - Gradient Boost

The below pipeline fills in the optimum score for the following grid search values:

```
param_grid = {
    'gbt_clf__learning_rate': [0.001, 0.01, 1],
    'gbt_clf__n_estimators': [50, 100, 200],
    'gbt_clf__max_depth': [1, 3, 5]
}
```

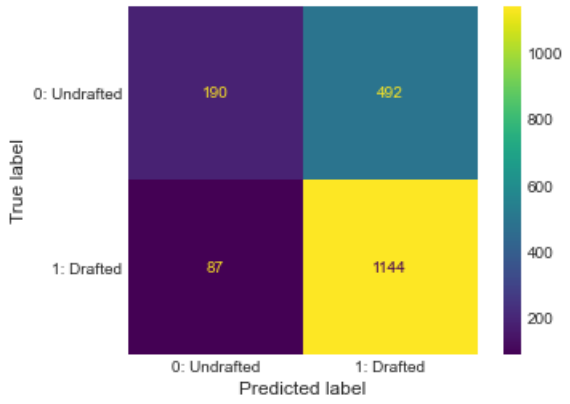
In [127]:
```
# Instantiate an XGB Classifier
gbt_steps = [('preprocess', ct),
         ('gbt_clf', GradientBoostingClassifier(random_state = 42))]

gbt_clf = Pipeline(gbt_steps)

gbt_clf.fit(X_train, y_train)

param_grid = {
    'gbt_clf__learning_rate': [0.01],
    'gbt_clf__n_estimators': [200],
    'gbt_clf__max_depth': [5]
}

# find best param based on f1 score
gbt_clf = GridSearchCV(estimator = gbt_clf,
                       param_grid = param_grid,
                       scoring = 'f1') # goes through all parameters in the defined grid

# run this to find best parameters based on gridsearch
gbt_clf.fit(X_train, y_train)

# get best scores and parameters
display(gbt_clf.best_params_)
display(gbt_clf.best_score_)
```

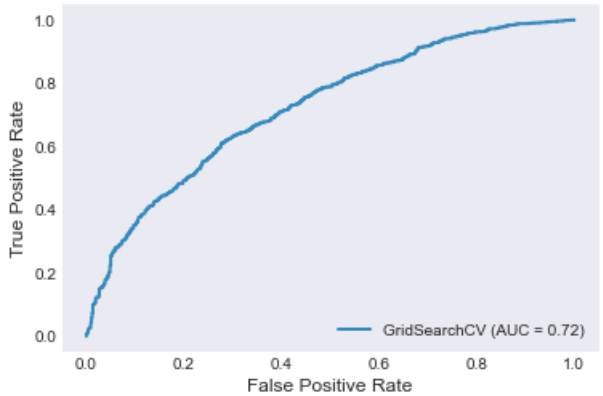
```
{'gbt_clf__learning_rate': 0.01,
 'gbt_clf__max_depth': 5,
 'gbt_clf__n_estimators': 200}
```

```
0.7969163423992287
```

In [128]:
```
# plot matrix
plot_confusion_matrix(gbt_clf,
                      X_test, y_test,
                      display_labels = class_names)

f1_train7 = round(f1_score(y_train, gbt_clf.predict(X_train)),4)
f1_test7 = round(f1_score(y_test, gbt_clf.predict(X_test)),4)
auc7 = get_auc(gbt_clf, X_test, y_test)

# append to scores list
f1_train_list.append(f1_train7)
f1_test_list.append(f1_test7)
auc_list.append(auc7)

# get scores
print('7th Model - Gradient Boost Score (Training):', f1_train7)
print('7th Model - Gradient Boost Score (Test):', f1_test7)
print('AUC Test Score:', auc7)
print()
print('Cross Validation Score:', get_cv_score(gbt_clf, X_train, y_train))
print('-'*80)

# get classification report on the test set
y_pred = gbt_clf.predict(X_test)
print(classification_report(y_test, y_pred))

# plot roc curve
plot_roc_curve(gbt_clf, X_test, y_test)
plt.show()
```

```
7th Model - Gradient Boost Score (Training): 0.8198
7th Model - Gradient Boost Score (Test): 0.798
AUC Test Score: 0.724

Cross Validation Score: 0.7969
--------------------------------------------------------------------------------
              precision    recall  f1-score   support

           0       0.69      0.28      0.40       682
           1       0.70      0.93      0.80      1231

    accuracy                           0.70      1913
   macro avg       0.69      0.60      0.60      1913
weighted avg       0.69      0.70      0.65      1913
```





8th Model - XGBoost

The below pipeline fills in the optimum score for the following grid search values:

```
param_grid = {
    'xgb_clf__learning_rate': [0.01, 0.1, 1],
    'xgb_clf__max_depth': [3, 4, 5],
    'xgb_clf__n_estimators': [50, 100, 150],
    'xgb_clf__gamma': [0, 5, 10]
}
```

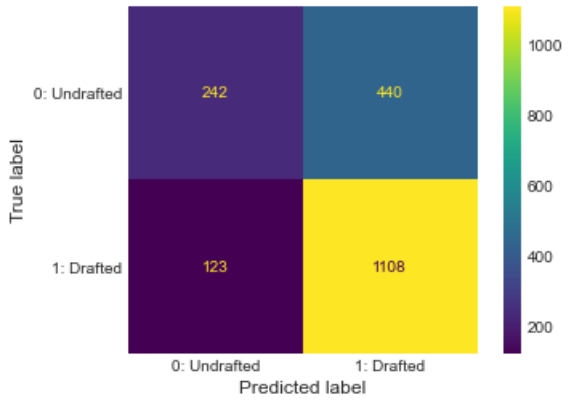
In [129]:
```
# Instantiate an XGB Classifier
steps = [('preprocess', ct),
         ('xgb_clf', XGBClassifier(random_state = 42))]

xgb_clf = Pipeline(steps)

xgb_clf.fit(X_train, y_train)

param_grid = {
    'xgb_clf__learning_rate': [0.1],
    'xgb_clf__max_depth': [4],
    'xgb_clf__n_estimators': [100],
    'xgb_clf__gamma': [0]
}

# find best param based on f1 score
xgb_clf = GridSearchCV(estimator = xgb_clf,
                       param_grid = param_grid,
                       scoring = 'f1') # goes through all parameters in the defined grid

# run this to find best parameters based on gridsearch
xgb_clf.fit(X_train, y_train)

# get best scores and parameters
display(xgb_clf.best_params_)
display(xgb_clf.best_score_)
```

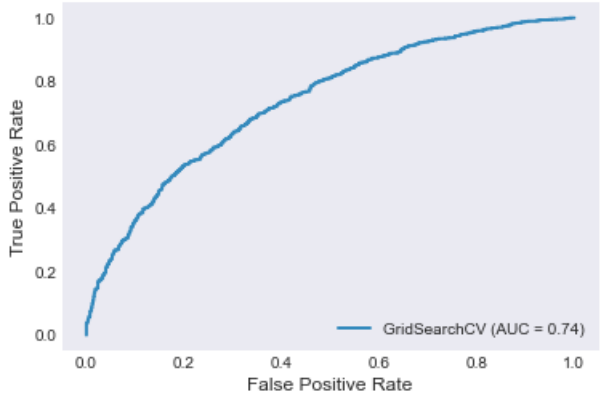
```
{'xgb_clf__gamma': 0,
 'xgb_clf__learning_rate': 0.1,
 'xgb_clf__max_depth': 4,
 'xgb_clf__n_estimators': 100}
```

```
0.7978795379640818
```

In [130]:
```
# plot matrix
plot_confusion_matrix(xgb_clf,
                      X_test, y_test,
                      display_labels = class_names)

f1_train8 = round(f1_score(y_train, xgb_clf.predict(X_train)),4)
f1_test8 = round(f1_score(y_test, xgb_clf.predict(X_test)),4)
auc8 = get_auc(xgb_clf, X_test, y_test)

# append to scores list
f1_train_list.append(f1_train8)
f1_test_list.append(f1_test8)
auc_list.append(auc8)

# get scores
print('8th Model - XGBoost F1 Score (Training):', f1_train8)
print('8th Model - XGBoost F1 Score (Test):', f1_test8)
print('AUC Test Score:', auc8)
print()
print('Cross Validation Score:', get_cv_score(xgb_clf, X_train, y_train))
print('-'*80)

# get classification report on the test set
y_pred = xgb_clf.predict(X_test)
print(classification_report(y_test, y_pred))

# plot roc curve
plot_roc_curve(xgb_clf, X_test, y_test)
plt.show()
```

```
8th Model - XGBoost F1 Score (Training): 0.8348
8th Model - XGBoost F1 Score (Test): 0.7974
AUC Test Score: 0.737

Cross Validation Score: 0.7979
--------------------------------------------------------------------------------
              precision    recall  f1-score   support

           0       0.66      0.35      0.46       682
           1       0.72      0.90      0.80      1231

    accuracy                           0.71      1913
   macro avg       0.69      0.63      0.63      1913
weighted avg       0.70      0.71      0.68      1913
```





Model # 9 - Bagging Classifier

The below pipeline fills in the optimum score for the following grid search values:

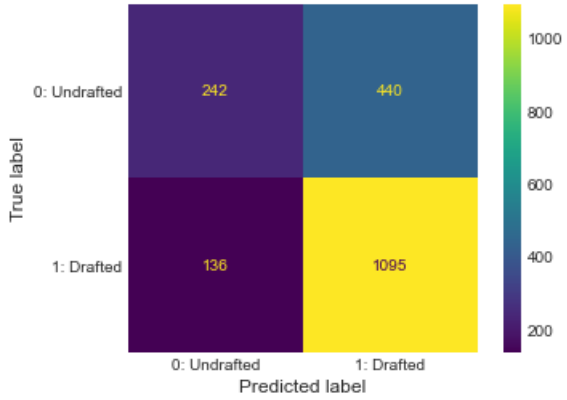
```
param_grid = {
    'bag_dtc_clf__base_estimator__max_depth': [4, 8, 12],
    'bag_dtc_clf__n_estimators': [50, 100, 150, 200]
}
```

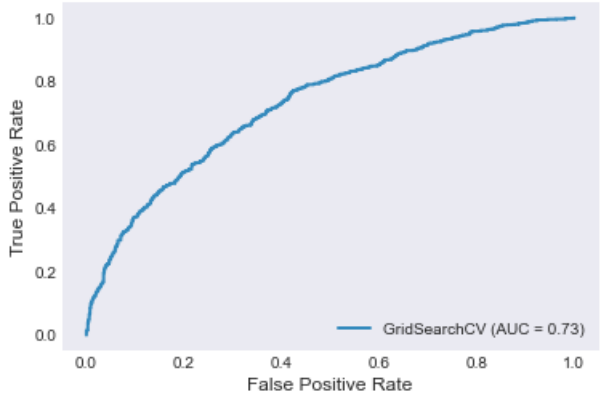
In [131]:
```
# Instantiate Classifier
steps = [('preprocess', ct),
         ('bag_dtc_clf', BaggingClassifier(base_estimator = DecisionTreeClassifier(),
                                           random_state = 42))]

bag_dtc_clf = Pipeline(steps)

bag_dtc_clf.fit(X_train, y_train)

param_grid = {
    'bag_dtc_clf__base_estimator__max_depth': [8],
    'bag_dtc_clf__n_estimators': [150]
}

# find best param based on f1 score
bag_dtc_clf = GridSearchCV(estimator = bag_dtc_clf,
                           param_grid = param_grid,
                           scoring = 'f1')

# run this to find best parameters based on gridsearch
bag_dtc_clf.fit(X_train, y_train)

# get best scores and parameters
display(bag_dtc_clf.best_params_)
display(bag_dtc_clf.best_score_)
```

```
{'bag_dtc_clf__base_estimator__max_depth': 8, 'bag_dtc_clf__n_estimators': 150}

0.7948097297402283
```

In [132]:
```
# plot matrix
plot_confusion_matrix(bag_dtc_clf,
                      X_test, y_test,
                      display_labels = class_names)

f1_train9 = round(f1_score(y_train, bag_dtc_clf.predict(X_train)),4)
f1_test9 = round(f1_score(y_test, bag_dtc_clf.predict(X_test)),4)
auc9 = get_auc(bag_dtc_clf, X_test, y_test)

# append to scores list
f1_train_list.append(f1_train9)
f1_test_list.append(f1_test9)
auc_list.append(auc9)

# get scores
print('9th Model - Bagging Decision Tree Classifier F1 Score (Training):', f1_train9)
print('9th Model - Bagging Decision Tree Classifier F1 Score (Test):', f1_test9)
print('AUC Test Score:', auc9)
print()
print('Cross Validation Score:', get_cv_score(bag_dtc_clf, X_train, y_train))
print('-'*80)

# get classification report on the test set
y_pred = bag_dtc_clf.predict(X_test)
print(classification_report(y_test, y_pred))

# plot roc curve
plot_roc_curve(bag_dtc_clf, X_test, y_test)
plt.show()
```

```
9th Model - Bagging Decision Tree Classifier F1 Score (Training): 0.8508
9th Model - Bagging Decision Tree Classifier F1 Score (Test): 0.7918
AUC Test Score: 0.733

Cross Validation Score: 0.7948
--------------------------------------------------------------------------------
              precision    recall  f1-score   support

           0       0.64      0.35      0.46       682
           1       0.71      0.89      0.79      1231

    accuracy                           0.70      1913
   macro avg       0.68      0.62      0.62      1913
weighted avg       0.69      0.70      0.67      1913
```





Model # 10 - Random Forest Classifier

The below pipeline fills in the optimum score for the following grid search values:

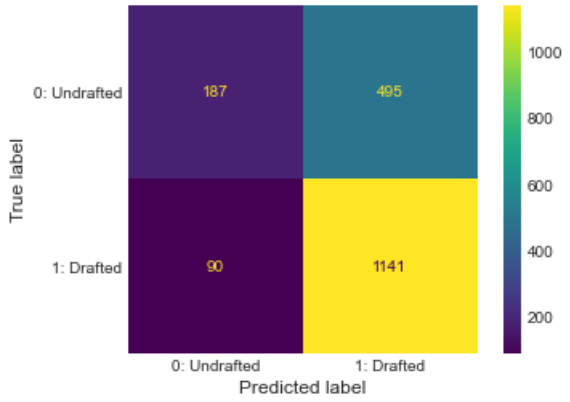
```
param_grid = {
    'rf_clf__criterion': ['gini', 'entropy'],
    'rf_clf__n_estimators': [50, 150, 200],
    'rf_clf__min_samples_leaf': [2, 5, 20, 50]
}
```

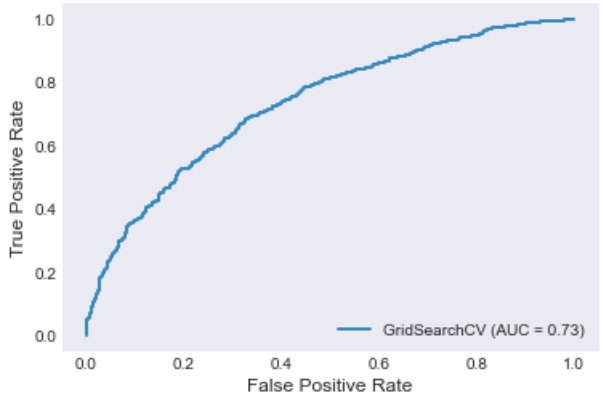
In [133]:
```
# Instantiate Classifier
steps = [('preprocess', ct),
         ('rf_clf', RandomForestClassifier(random_state = 42))]

rf_clf = Pipeline(steps)

rf_clf.fit(X_train, y_train)

param_grid = {
    'rf_clf__criterion': ['entropy'],
    'rf_clf__n_estimators': [150],
    'rf_clf__min_samples_leaf': [2]
}

# find best param based on f1 score
rf_clf = GridSearchCV(estimator = rf_clf,
                      param_grid = param_grid,
                      scoring = 'f1')

# run this to find best parameters based on gridsearch
rf_clf.fit(X_train, y_train)

# get best scores and parameters
display(rf_clf.best_params_)
display(rf_clf.best_score_)
```

```
{'rf_clf__criterion': 'entropy',
 'rf_clf__min_samples_leaf': 2,
 'rf_clf__n_estimators': 150}

0.8031392581177783
```

In [134]:
```
# plot matrix
plot_confusion_matrix(rf_clf,
                      X_test, y_test,
                      display_labels = class_names)

f1_train10 = round(f1_score(y_train, rf_clf.predict(X_train)),4)
f1_test10 = round(f1_score(y_test, rf_clf.predict(X_test)),4)
auc10 = get_auc(rf_clf, X_test, y_test)

# append to scores list
f1_train_list.append(f1_train10)
f1_test_list.append(f1_test10)
auc_list.append(auc10)

# get scores
print('10th Model - Random Forest Classifier F1 Score (Training):', f1_train10)
print('10th Model - Random Forest Classifier F1 Score (Test):', f1_test10)
print('AUC Test Score:', auc10)
print()
print('Cross Validation Score:', get_cv_score(rf_clf, X_train, y_train))
print('-'*80)

# get classification report on the test set
y_pred = rf_clf.predict(X_test)
print(classification_report(y_test, y_pred))

# plot roc curve
plot_roc_curve(rf_clf, X_test, y_test)
plt.show()
```

```
10th Model - Random Forest Classifier F1 Score (Training): 0.8582
10th Model - Random Forest Classifier F1 Score (Test): 0.796
AUC Test Score: 0.735

Cross Validation Score: 0.8031
--------------------------------------------------------------------------------
              precision    recall  f1-score   support

           0       0.68      0.27      0.39       682
           1       0.70      0.93      0.80      1231

    accuracy                           0.69      1913
   macro avg       0.69      0.60      0.59      1913
weighted avg       0.69      0.69      0.65      1913
```





7. Evaluation

After assessing 10 total models, lets now summarize the findings by creating a dataframe for all the results above.

In [135]:
```
# create a summary dataframe for all the models
columns = ['Model', 'Training - F1 Score', 'Testing - F1 Score', 'AUC']

models = ['1st Model - Baseline Logistic Regression',
          '2nd Model - Best Params (lbfgs) Logistic Regression',
          '3rd Model -  Liblinear',
          '4th Model - Decision Tree',
          '4B Model - Decision Tree',
          '5th Model - KNN',
          '6th Model - Adaboost',
          '7th Model - Gradient Boost',
          '8th Model - XGBoost',
          '9th Model - Bagging Decision Tree Classifier',
          '10th Model - Random Forest Classifier']

zipped = list(zip(models, f1_train_list, f1_test_list, auc_list))

summary_df = pd.DataFrame(zipped, columns=columns, index = None)
```

In [136]:
```
# sort the dataframe by Testing F1 Score
summary_df.sort_values(by = ['AUC', 'Testing - F1 Score'], ascending = False)
```

Out[136]:

	Model	Training - F1 Score	Testing - F1 Score	AUC
8	8th Model - XGBoost	0.8348	0.7974	0.737
10	10th Model - Random Forest Classifier	0.8582	0.7960	0.735
9	9th Model - Bagging Decision Tree Classifier	0.8508	0.7918	0.733
7	7th Model - Gradient Boost	0.8198	0.7980	0.724
0	1st Model - Baseline Logistic Regression	0.8106	0.7914	0.724
1	2nd Model - Best Params (lbfgs) Logistic Regre...	0.8065	0.7916	0.718
2	3rd Model - Liblinear	0.8067	0.7911	0.718
6	6th Model - Adaboost	0.7958	0.7813	0.708
5	5th Model - KNN	0.8042	0.7922	0.689
4	4B Model - Decision Tree	0.8411	0.7681	0.662
3	4th Model - Decision Tree	1.0000	0.7156	0.582

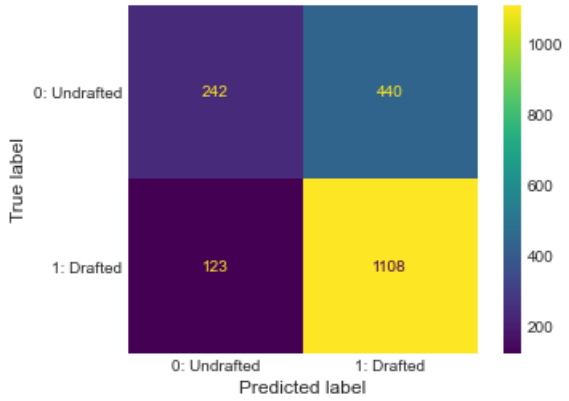
So our best model was the 8th Model - XGBoost based on AUC and is slightly higher than the Random Forst and Bagging Decision Tree Classifiers.

However, the 7th Model - Gradient Boost performed the best overall in terms of F1 Score, but only slightly better than the XGBoost model.

Based on the above, we will say that the XGBoost performed the best overall when it comes to classifying between players who are Drafted and Undrafted.

As a reminder, below is the performance of the XGBoost model.

In [137]:
```
plot_confusion_matrix(xgb_clf,
                      X_test, y_test,
                      display_labels = class_names)
plt.show()
```



In [138]:
```
# get classification report on the test set
y_pred = xgb_clf.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.66      0.35      0.46       682
           1       0.72      0.90      0.80      1231

    accuracy                           0.71      1913
   macro avg       0.69      0.63      0.63      1913
weighted avg       0.70      0.71      0.68      1913
```

While the XGBoost Model performed with an overall F1-Score of 80%, the model still struggles when it comes to classifying whether a player will be drafted or undrafted.

There is still a significant amount of False-Positives (or players classified as Drafted when they were not) amounting to a Precision Score of 72% correctly classified Drafted players. Consequently, there is a Precision Score of 66% when classifying Undrafted players.

Assessing the Important Features

Lets now assess which features in the XGBoost model are the most important when making a classification.

In [139]:
```
# assign xgb model as best model
best_model = xgb_clf.best_estimator_
```

In [140]:
```
# check feature importances from xgb model
feat_imp = best_model['xgb_clf'].feature_importances_

feat_imp_series = pd.Series(feat_imp,
          index = pp_X_train.columns)

# preview first 5
feat_imp_series[:5]
```

Out[140]:
```
0    0.000000
1    0.014012
2    0.018107
3    0.004642
4    0.000000
dtype: float32
```

We need to index these importance features values with their appropriate values to make sense of which Importance value is associated with which Feature. We'll merge the `feature_names_df` with the `feature_imp_df` .

In [141]:
```
# recall feature_names_df and names of features in the df
feature_names_df.tail(10)
```

Out[141]:

	0
340	Yale
341	Youngstown State
342	Wt
343	40yd
344	Vertical
345	Bench
346	Broad Jump
347	3Cone
348	Shuttle
349	Height

In [142]:
```
# convert feat_imp_series into a df
feat_imp_df = pd.DataFrame(feat_imp_series)
feat_imp_df.head()
```

Out[142]:

	0
0	0.000000
1	0.014012
2	0.018107
3	0.004642
4	0.000000

In [143]:
```
# merge the feature names and feature importance dataframes based on index
merged = pd.merge(feature_names_df, feat_imp_df, left_index=True, right_index=True)

# rename the new merged dataframe
merged = merged.rename(columns = {'0_x': 'Feature', '0_y': 'Importance'})
merged.head()
```

Out[143]:

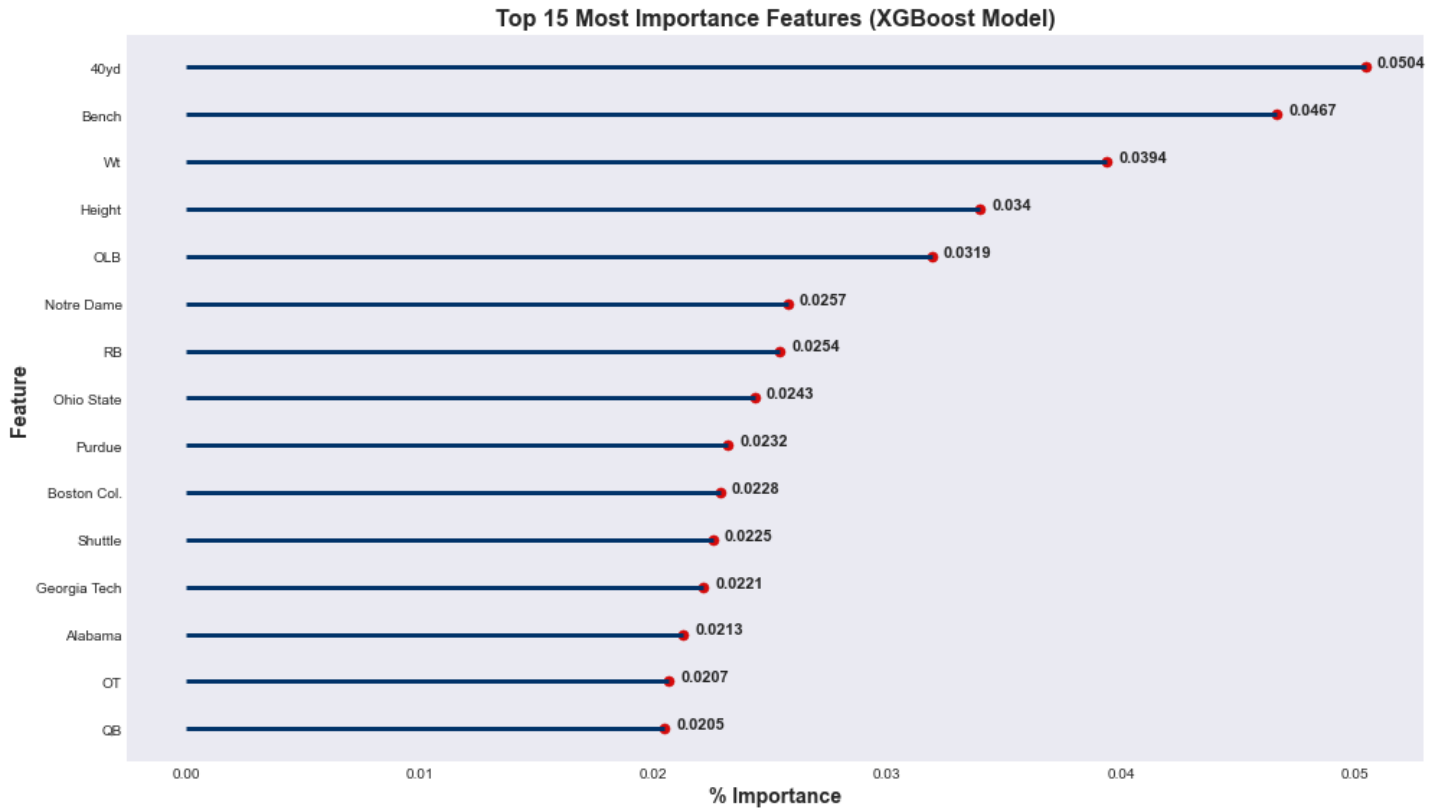
	Feature	Importance
0	C	0.000000
1	CB	0.014012
2	DB	0.018107
3	DE	0.004642
4	DL	0.000000

In [144]:
```
# sort by top 15 most important features
sorted_features = merged.sort_values('Importance', ascending = False)

# create a top 15 features df
top_15_features = sorted_features.head(15)
top_15_features
```

Out[144]:

	Feature	Importance
343	40yd	0.050430
345	Bench	0.046655
342	Wt	0.039378
349	Height	0.033950
14	OLB	0.031905
225	Notre Dame	0.025746
18	RB	0.025394
228	Ohio State	0.024315
244	Purdue	0.023174
50	Boston Col.	0.022843
348	Shuttle	0.022532
115	Georgia Tech	0.022148
26	Alabama	0.021284
15	OT	0.020655
17	QB	0.020483

In [145]:
```
# plot a lollipop plot of top 15 most important features
fig, ax = plt.subplots(figsize=(14, 8))
plt.style.use('bmh')

# sort by descending
top_15_features= top_15_features.sort_values(by = 'Importance')

# plots the lines
plt.hlines(y = top_15_features['Feature'],  # feature
           xmin = 0,
           xmax = top_15_features['Importance'],  # importance
           color = '#013369', linewidth = 3)

plt.scatter(top_15_features['Importance'], # Count values
            top_15_features['Feature'],  # positions
            color='#d50a0a',
            alpha=1, s = 50)

# annotate the scatter plot values
for idx, row in top_15_features.iterrows():
    ax.annotate(round(row['Importance'],4), (row['Importance'], row['Feature'] ),
                weight = "bold", fontsize = 11,
                xytext=(row['Importance'] + 0.0005, row['Feature'])) # offset annotate

plt.title('Top 15 Most Importance Features (XGBoost Model)', weight = "bold", fontsize = 16)
plt.ylabel('Feature', weight = "bold", fontsize = 14)
plt.xlabel('% Importance', weight = "bold", fontsize = 14)

plt.tight_layout()
plt.show()
fig.savefig('Images/Plots/top15_important_features.png');
```



As we can see from the graph, the relative scores highlight which features are most relevant to classification of the target of `Drafted`. The higher the feature importance, the more of an effect it has on the model's performance.

Based on the above, we can make the following observations:

- The model suggests that the most important combine measurables that classify `Draft` status are the 40 Yard and Bench. The Shuttle drill follows thereafter, though not as important of a feature.
- The schools with the most importance for `Draft` classification are Notre Dame, Ohio State, Purdue, and Boston College. Alabama is surprisingly behind these other schools even though they have been powerhouses recently when it comes to producing NFL talent.
- Positions with the most importance are surprisingly the OLB (Outside Linebacker), RB (Running Back), OT (Offensive Tackle), and followed by the QB (Quarterback). While the model does not suggest that these are the most important positions on the field, they are the most

important when it comes to classification of draft status alone.

Observations of the Model on the Original Dataset

In [146]:
```
# # reload the original dataset
df = pd.read_csv('Data/combine_2000_2022.csv', index_col = 0)
df.head()
```

Out[146]:

	Player	Pos	School	College	Ht	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Drafted (tm/rnd/yr)	Year
0	John Abraham	OLB	South Carolina	NaN	6-4	252.0	4.55	NaN	NaN	NaN	NaN	NaN	New York Jets / 1st / 13th pick / 2000	2000
1	Shaun Alexander	RB	Alabama	College Stats	6-0	218.0	4.58	NaN	NaN	NaN	NaN	NaN	Seattle Seahawks / 1st / 19th pick / 2000	2000
2	Darnell Alford	OT	Boston Col.	NaN	6-4	334.0	5.56	25.0	23.0	94.0	8.48	4.98	Kansas City Chiefs / 6th / 188th pick / 2000	2000
3	Kyle Allamon	TE	Texas Tech	NaN	6-2	253.0	4.97	29.0	NaN	104.0	7.29	4.49	NaN	2000
4	Rashard Anderson	CB	Jackson State	NaN	6-2	206.0	4.55	34.0	NaN	123.0	7.18	4.15	Carolina Panthers / 1st / 23rd pick / 2000	2000

In [147]:
```
# dataframe with only player names
df_names = df[['Player', 'Year']]
df_names.head()
```

Out[147]:

	Player	Year
0	John Abraham	2000
1	Shaun Alexander	2000
2	Darnell Alford	2000
3	Kyle Allamon	2000
4	Rashard Anderson	2000

In [148]:
```
# save original uncleaned metrics
# df_metrics will be used in the pipeline to make predictions
df_metrics_original = df_metrics.copy()
```

In [149]:
```
df_metrics_original.head(3)
```

Out[149]:

	Pos	School	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Height	Drafted
0	OLB	South Carolina	252.0	4.55	NaN	NaN	NaN	NaN	NaN	6.33	1
1	RB	Alabama	218.0	4.58	NaN	NaN	NaN	NaN	NaN	6.00	1
2	OT	Boston Col.	334.0	5.56	25.0	23.0	94.0	8.48	4.98	6.33	1

In [150]:
```
# apply helper function to fill in missing with means for each metric column
df_metrics = fill_missing(df_metrics)
```

In [151]:
```
# fill NAs with 0
df_metrics = df_metrics.fillna(0)
```

In [152]:
```
# check NAs
df_metrics.isna().sum()
```

Out[152]:
```
Pos           0
School        0
Wt            0
40yd          0
Vertical      0
Bench         0
Broad Jump    0
3Cone         0
Shuttle       0
Height        0
Drafted       0
dtype: int64
```

In [153]:
```
# create actual_drafted as target variable, treat as 'y_test'
actual_drafted = df_metrics['Drafted']
actual_drafted.head()
```

Out[153]:
```
0    1
1    1
2    1
3    0
4    1
Name: Drafted, dtype: int64
```

In [154]:
```
# drop the target
df_metrics = df_metrics.drop(columns = ['Drafted'])
df_metrics_original = df_metrics_original.drop(columns = ['Drafted'])
```

In [155]:
```
# treat df_metrics as my 'X_test'
df_metrics.head(3)
```

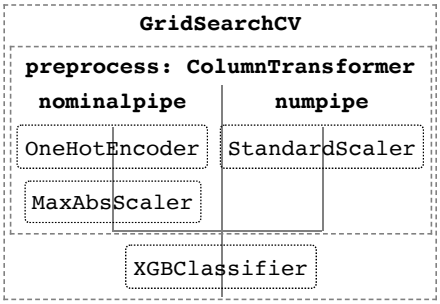
Out[155]:

	Pos	School	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Height
0	OLB	South Carolina	252.0	4.55	34.466667	22.532544	117.646893	7.135816	4.288433	6.33
1	RB	Alabama	218.0	4.58	34.357782	19.334661	118.345455	7.070724	4.274591	6.00
2	OT	Boston Col.	334.0	5.56	25.000000	23.000000	94.000000	8.480000	4.980000	6.33

Pass the dataframe through the XGBoost Model Pipeline

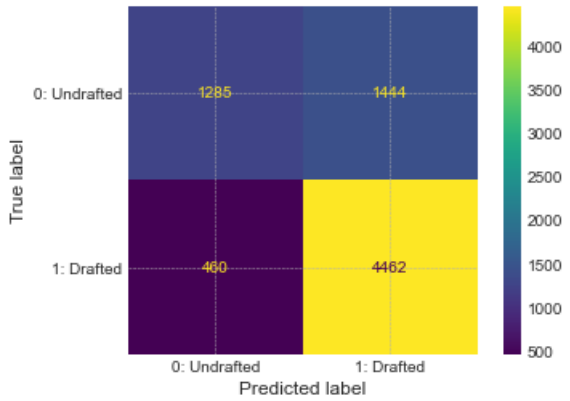
In [156]:
```
# Instantiate an XGB Classifier
steps = [('preprocess', ct),
         ('xgb_clf', XGBClassifier(random_state = 42))]

xgb_clf = Pipeline(steps)

xgb_clf.fit(X_train, y_train) # fit onto the original train set

param_grid = {
    'xgb_clf__learning_rate': [0.1],
    'xgb_clf__max_depth': [4],
    'xgb_clf__n_estimators': [100],
    'xgb_clf__gamma': [0]
}

# find best param based on f1 score
xgb_clf = GridSearchCV(estimator = xgb_clf,
                       param_grid = param_grid,
                       scoring = 'f1')

# run this to find best parameters based on gridsearch
xgb_clf.fit(X_train, y_train)
```

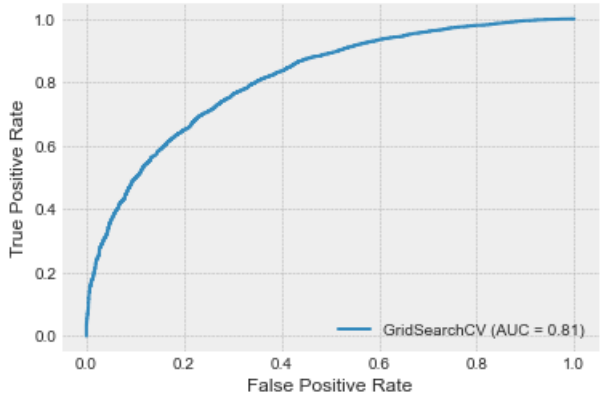
Out[156]:



In [157]:
```
# plot matrix
plot_confusion_matrix(xgb_clf,
                      df_metrics, actual_drafted, # substitute X_test and y_test here
                      display_labels = class_names)

# get f1 and AUC scores from the entire set
f1_score = round(f1_score(actual_drafted, xgb_clf.predict(df_metrics)),4)
auc_score = get_auc(xgb_clf, df_metrics, actual_drafted)

# get scores
print('XGBoost F1 Score (Entire Set):', f1_score)
print('AUC Score (Entire Set):', auc_score)
print('-'*80)

# get classification report on the entire set
entire_preds = xgb_clf.predict(df_metrics)  # apply prediction on the X_test ie. df_metrics
print(classification_report(actual_drafted, entire_preds))  # classif. report on actual drafted vs. preds

# plot roc curve
plot_roc_curve(xgb_clf, df_metrics, actual_drafted)
plt.show()
```

```
XGBoost F1 Score (Entire Set): 0.8242
AUC Score (Entire Set): 0.811
--------------------------------------------------------------------------------
              precision    recall  f1-score   support

           0       0.74      0.47      0.57      2729
           1       0.76      0.91      0.82      4922

    accuracy                           0.75      7651
   macro avg       0.75      0.69      0.70      7651
weighted avg       0.75      0.75      0.74      7651
```





Based on the F1-Score and AUC score from the XGBoost model predicting on the entire dataset, we are actually getting a slightly better score than on the `X_test` . This is expected since a majority of the `X_train` set is already fit into the model. Therefore, the model would expect to classify these familiar datapoints very well.

Combine the dataframe to compare predictions vs actual draft status results.

In [158]:
```
draft_preds = pd.DataFrame(entire_preds, columns = ['Predicted Draft Status'])
draft_preds.head(3)
```

Out[158]:

	Predicted Draft Status
0	1
1	1
2	0

In [159]:
```
# create dataframe of actually drafted
actual_drafted_df = pd.DataFrame(actual_drafted)

# merge to preds
combined_draft = pd.merge(actual_drafted_df, draft_preds, left_index=True, right_index=True)
combined_draft.head(3)
```

Out[159]:

	Drafted	Predicted Draft Status
0	1	1
1	1	1
2	1	0

In [160]:
```
# merge combined_draft to df_metrics_original
final_df = pd.merge(df_metrics_original, combined_draft, left_index=True, right_index=True)

# join with df_names
final_df = pd.merge(df_names, final_df, left_index=True, right_index=True)
final_df.tail()
```

Out[160]:

	Player	Year	Pos	School	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Height	Drafted	Predicted Draft Status
7646	Velus Jones	2022	WR	Tennessee	204.0	4.31	33.0	NaN	121.0	NaN	NaN	6.00	1	1
7647	Kerby Joseph	2022	S	Illinois	203.0	NaN	38.5	18.0	123.0	NaN	NaN	6.08	1	1
7648	Kevin Austin Jr.	2022	WR	Notre Dame	200.0	4.43	39.0	NaN	132.0	6.71	4.15	6.17	0	0
7649	Reggie Roberson Jr.	2022	WR	SMU	192.0	NaN	29.0	NaN	114.0	NaN	NaN	5.92	0	1
7650	Cameron Jurgens	2022	C	Nebraska	303.0	4.92	NaN	25.0	NaN	NaN	NaN	6.25	1	0

In [161]:
```
# percent predicted
final_df['Predicted Draft Status'].value_counts(normalize = True)
```

Out[161]:
```
1    0.772982
0    0.227018
Name: Predicted Draft Status, dtype: float64
```

In [162]:
```
# percent actual
final_df['Drafted'].value_counts(normalize = True)
```

Out[162]:
```
1    0.641635
0    0.358365
Name: Drafted, dtype: float64
```

Lets find some players who fit the description of the following:

- True Positive (Predicted as Drafted and was Actually Drafted)
- True Negative (Predicted as Undrafted and was Actually Undrafted)
- False Positive (Predicted as Drafted and was Actually Undrafted)
- False Negative (Predicted as Undrafted and was Actually Drafted)

Lets pick a few players at random out of the `X_test` set since this is what we originally predicted on. And then compare these results with the `final_df` .

In [163]:
```
X_test.head(3)
```

Out[163]:

	Pos	School	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Height
1856	OT	North Carolina	291.0	5.30	27.711538	24.15625	102.475728	7.844737	4.763196	6.33
4490	OLB	UNLV	233.0	4.84	35.500000	25.00000	117.000000	6.910000	4.300000	5.92
5743	QB	Virginia Tech	232.0	4.80	26.500000	23.00000	112.000000	7.167262	4.410000	6.25

In [164]:
```
# select a random player from X_test
def select_player(test, final):
    test_index = np.random.choice(list(test.index)) # select from list of indexes in the test set
    test_player_selected = test.loc[[test_index]]  # loc the index and the player values from test

    final_player_selected = final.loc[[test_index]] # find the player in the final_df set

    return final_player_selected
```

In [165]:
```
# return randomly selected player
# continuously run to see draft status
select_player(X_test, final_df)
```

Out[165]:

	Player	Year	Pos	School	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Height	Drafted	Predicted Draft Status
3757	Tori Gurley	2011	WR	South Carolina	216.0	4.53	33.5	15.0	118.0	7.05	4.25	6.33	0	1

Assessing the Offense, Defense, and Special Teams

As an additional analysis, recall that earlier we split up the overall dataset into offense, defense, and special teams groups.

We'll check how these subset groups perform and see if there are any additional takeaways we can make.

In [166]:
```
# preview example of the special teams group only
special_teams.head()
```

Out[166]:

	Pos	School	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Height	Drafted
5	K	Ala-Birmingham	202.0	NaN	NaN	NaN	NaN	NaN	NaN	5.83	0
8	K	Baylor	167.0	NaN	NaN	NaN	NaN	NaN	NaN	6.00	0
10	P	North Texas	227.0	NaN	NaN	NaN	NaN	NaN	NaN	6.25	0
15	P	East Carolina	200.0	NaN	NaN	NaN	NaN	NaN	NaN	6.17	0
138	P	Colorado State	234.0	NaN	NaN	NaN	NaN	NaN	NaN	6.17	0

Lets now apply the XGBoost model on the subgroups to see if the model can generalize and classify the sub-groups better.

Note: It was discovered earlier that Special Teams players (ie. Kickers) do not perform in the 3Cone and Shuttle drill. We will drop these columns when analyzing the special teams data subset.

In [167]:
```
# offense - create new X and y
offense_X = offense.drop(columns = 'Drafted')
offense_y = offense['Drafted']

# defense - create new X and y
defense_X = defense.drop(columns = 'Drafted')
defense_y = defense['Drafted']

# special teams - create new X and y
st_X = special_teams.drop(columns = ['Drafted', '3Cone', 'Shuttle'])
st_y = special_teams['Drafted']
```

In [168]:
```
# offense train test split
off_X_train, off_X_test, off_y_train, off_y_test = train_test_split(offense_X, offense_y, stratify = offense

# offense train test split
def_X_train, def_X_test, def_y_train, def_y_test = train_test_split(defense_X, defense_y, stratify = defense

# offense train test split
st_X_train, st_X_test, st_y_train, st_y_test = train_test_split(st_X, st_y, stratify = st_y, random_state=42
```

In [169]:
```
# preview
off_X_train.head()
```

Out[169]:

	Pos	School	Wt	40yd	Vertical	Bench	Broad Jump	3Cone	Shuttle	Height
754	OG	Nebraska	349.0	5.49	28.5	32.0	97.0	8.12	4.81	6.33
5211	OG	Missouri	305.0	5.14	31.0	36.0	112.0	7.60	4.50	6.42
5845	RB	North Carolina	196.0	4.37	33.5	17.0	121.0	NaN	NaN	5.75
2302	WR	Oregon	197.0	4.52	38.0	NaN	127.0	6.84	4.08	6.17
2011	OT	Cornell	316.0	5.41	31.5	23.0	98.0	8.36	5.07	6.42

In [170]:
```
# check missing
off_X_train.isna().sum()
```

Out[170]:
```
Pos              0
School           0
Wt               0
40yd           142
Vertical       565
Bench         1143
Broad Jump     608
3Cone          968
Shuttle        929
Height           0
dtype: int64
```

In [171]:
```
# create a helper function for future use.
def fill_missing(df):
    """
    Helper function that takes in columns from a dataframe and fills in the NaNs
    with the means grouped by Position. Can be used for the training and test sets.
    """
    missing_cols = ['40yd', 'Vertical', 'Bench', 'Broad Jump', '3Cone', 'Shuttle']

    for col in missing_cols:
        df[col] = df[col].fillna(df.groupby('Pos')[col].transform('mean'))

    return df
```

In [172]:
```
# apply helper function to fill in missing on train and test sets
fill_missing(off_X_train)
fill_missing(off_X_test)

fill_missing(def_X_train)
fill_missing(def_X_test)

# check
def_X_train.isna().sum()
```

Out[172]:
```
Pos           0
School        0
Wt            0
40yd          0
Vertical      0
Bench         0
Broad Jump    0
3Cone         0
Shuttle       0
Height        0
dtype: int64
```

In [173]:
```
# fill the missing in special teams subset with the median instead of mean
def fill_missing_subset(df):
    """
    Helper function that takes in columns from a dataframe and fills in the NaNs
    with the means grouped by Position. Can be used for the training and test sets.
    """
    st_missing_cols = ['40yd', 'Vertical', 'Bench', 'Broad Jump']

    for col in st_missing_cols:
            df[col] = df[col].fillna(value = df[col].median())

    return df
```

In [174]:
```
# fill the missing special teams values
fill_missing_subset(st_X_train)
fill_missing_subset(st_X_test)
# check
st_X_train.isna().sum()
```

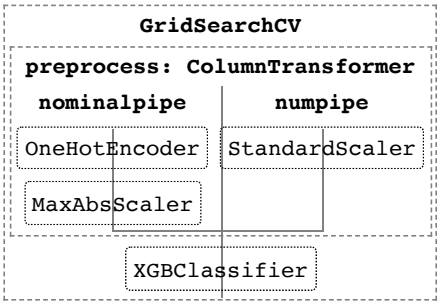
Out[174]:
```
Pos           0
School        0
Wt            0
40yd          0
Vertical      0
Bench         0
Broad Jump    0
Height        0
dtype: int64
```

Fitting Subsets to the XGBoost Model

Offense:

In [175]:
```
# Instantiate an XGB Classifier
steps = [('preprocess', ct),
         ('xgb_clf', XGBClassifier(random_state = 42))]

xgb_clf = Pipeline(steps)

xgb_clf.fit(off_X_train, off_y_train)

param_grid = {
    'xgb_clf__learning_rate': [0.1],
    'xgb_clf__max_depth': [4],
    'xgb_clf__n_estimators': [100],
    'xgb_clf__gamma': [0]
}

# find best param based on f1 score
xgb_clf = GridSearchCV(estimator = xgb_clf,
                       param_grid = param_grid,
                       scoring = 'f1')

# run this to find best parameters based on gridsearch
xgb_clf.fit(off_X_train, off_y_train)
```

Out[175]:

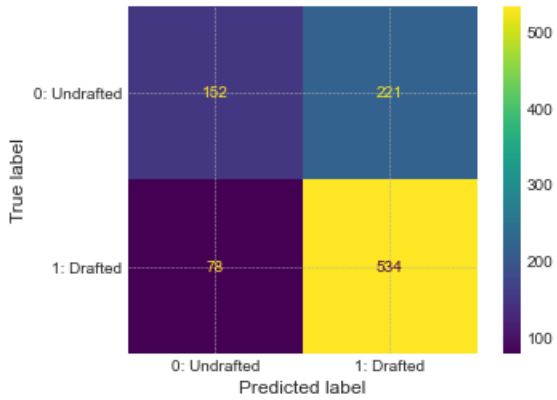


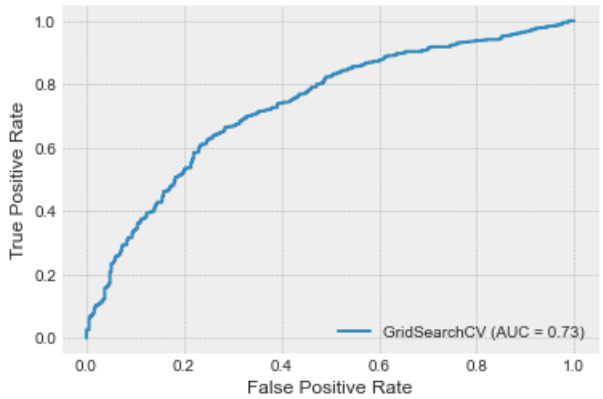
In [176]:
```
off_y_test
```

Out[176]:
```
94      0
6857    1
7171    0
5261    1
4181    0
       ..
3398    1
524     1
6467    1
6690    1
1743    1
Name: Drafted, Length: 985, dtype: int64
```

In [177]:
```
# plot matrix
plot_confusion_matrix(xgb_clf,
                      off_X_test, off_y_test,
                      display_labels = class_names)

# get classification report on the test set
off_y_pred = xgb_clf.predict(off_X_test)
print(classification_report(off_y_test, off_y_pred))

# plot roc curve
plot_roc_curve(xgb_clf, off_X_test, off_y_test)
plt.show()
```

```
              precision    recall  f1-score   support

           0       0.66      0.41      0.50       373
           1       0.71      0.87      0.78       612

    accuracy                           0.70       985
   macro avg       0.68      0.64      0.64       985
weighted avg       0.69      0.70      0.68       985
```



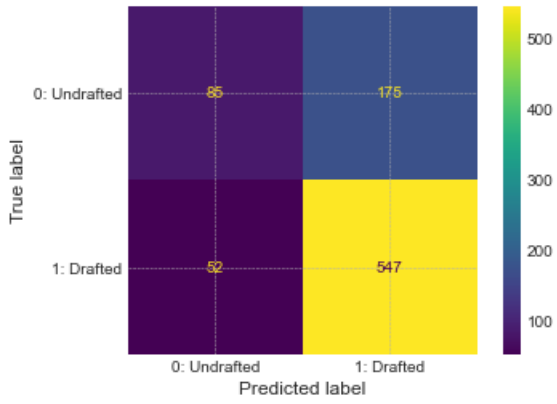


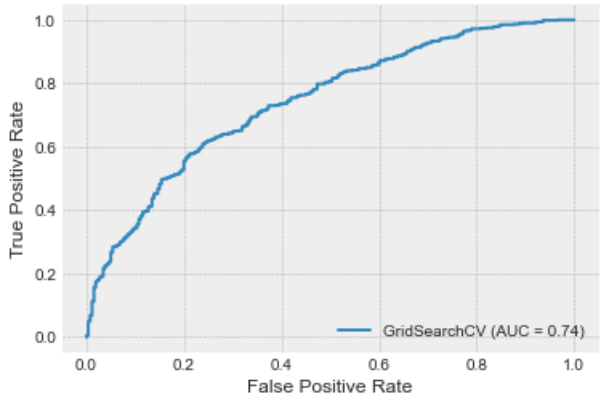
We see that the Offense subset performs slightly worse compared to the overall dataset. However, there is no significant difference in the F1-Score that we can see in this subset.

Defense:

In [178]:
```
# using same parameters on offense pipeline above
# fit onto the def train sets
xgb_clf.fit(def_X_train, def_y_train)

# plot matrix
plot_confusion_matrix(xgb_clf,
                      def_X_test, def_y_test,
                      display_labels = class_names)

# get classification report on the test set
def_y_pred = xgb_clf.predict(def_X_test)
print(classification_report(def_y_test, def_y_pred))

# plot roc curve
plot_roc_curve(xgb_clf, def_X_test, def_y_test)
plt.show()
```

```
              precision    recall  f1-score   support

           0       0.62      0.33      0.43       260
           1       0.76      0.91      0.83       599

    accuracy                           0.74       859
   macro avg       0.69      0.62      0.63       859
weighted avg       0.72      0.74      0.71       859
```





The Defense subset performs much better (F1-Score: 83%) compared to the overall dataset and Offensive subset. We know from the early EDA that there is a higher percentage of Undrafted players in the Offense than in the Defense. This could likely contribute to the model's ability to distinguish between drafted and undrafted players.

Special Teams:

Before proceeding with analysis, lets check the class balance in the Special Teams subset.

In [179]:
```
# check class balance
st_y_test.value_counts(normalize = True)
```

Out[179]:
```
0    0.710145
1    0.289855
Name: Drafted, dtype: float64
```

There is more of a class imbalance in the Special Teams subset. This is due to the nature of the positions in this subset as they are not necessarily high-priority positions in the draft. We also have far less data in this subset group as a result. However, lets still perform an analysis and see what we can observe.

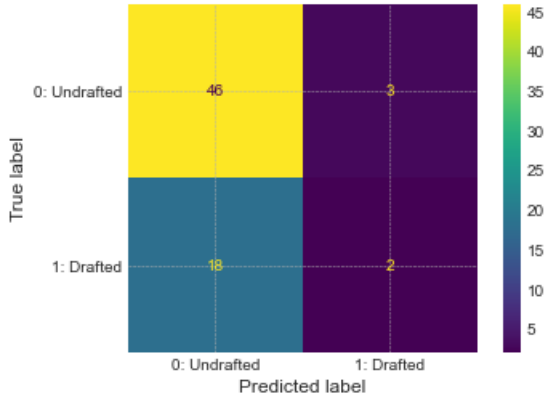
We'll need to redefine the pipeline for the special teams subset since we dropped two columns.

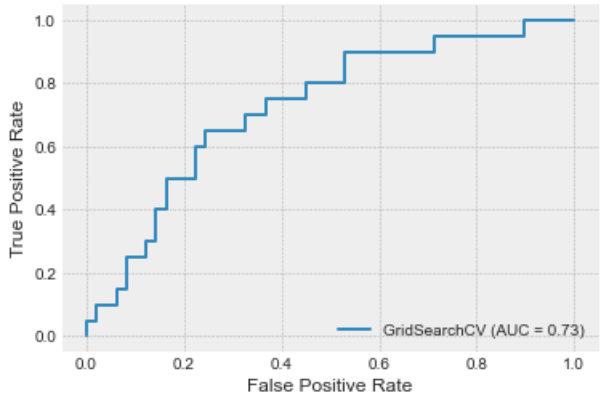
In [180]:
```
# define numeric columns to be standard scaled in special teams subset
st_num_cols = st_X_train.select_dtypes(['int', 'float']).columns

# define categorical columns to be OHE in special teams subset
st_cat_cols = st_X_train.select_dtypes(['object']).columns

# create new column transformer for special teams subset
st_ct = ColumnTransformer(transformers =
    [("nominalpipe", nominal_pipeline, st_cat_cols), # ohe and MaxAbsScale the Pos and School variables
     ("numpipe", numeric_pipeline, st_num_cols)])
```

In [181]:
```
# Instantiate an XGB Classifier
st_steps = [('preprocess', st_ct),  # use st_ct as column transformer for preprocessing
         ('xgb_clf', XGBClassifier(random_state = 42))]

xgb_clf = Pipeline(st_steps)

xgb_clf.fit(off_X_train, off_y_train)

param_grid = {
    'xgb_clf__learning_rate': [0.1],
    'xgb_clf__max_depth': [4],
    'xgb_clf__n_estimators': [100],
    'xgb_clf__gamma': [0]
}

# find best param based on f1 score
xgb_clf = GridSearchCV(estimator = xgb_clf,
                       param_grid = param_grid,
                       scoring = 'f1')
```

In [182]:
```
# using same parameters on offense pipeline above
# fit onto the st train sets
xgb_clf.fit(st_X_train, st_y_train)

# plot matrix
plot_confusion_matrix(xgb_clf,
                      st_X_test, st_y_test,
                      display_labels = class_names)

# get classification report on the test set
st_y_pred = xgb_clf.predict(st_X_test)
print(classification_report(st_y_test, st_y_pred))

# plot roc curve
plot_roc_curve(xgb_clf, st_X_test, st_y_test)
plt.show()
```

```
              precision    recall  f1-score   support

           0       0.72      0.94      0.81        49
           1       0.40      0.10      0.16        20

    accuracy                           0.70        69
   macro avg       0.56      0.52      0.49        69
weighted avg       0.63      0.70      0.62        69
```





Clearly, the special team subset does not perform well in this model likely due to the class imbalance and less data in this subset. There is a greater variability in the Special Teams subset since players that fall under this positional group are not typically drafted early or often.

We will call it here and finalize with some conclusions.

8. Conclusions & Recommendations

Model Performance

While the best model performed with an overall F1-Score of 80%, there is still evidence to suggest that draft combine metrics alone are not a strong indicator of whether a player will be drafted or undrafted.

With the XGBoost Model, there are still a significant amount of False-Positives (players classified as Drafted when they were not) amounting to a Precision Score of 72% of correctly classified Drafted players. Similarly, there is a Precision Score of 62% of correctly classifying Undrafted players.

It is clear that classification of draft status is not an exact science. Though, it is also important to note that there are potentially additional information and data that could be implimented into the model. For example, college statistics were not factored into the analysis. This would involve gathering all the data for every player in the combine and merging this information with the combine data. While this seems ideal, there is a lack of data when it comes to measuring each individual unique position in football. Further analysis would have to be split up into positional categories as not all statistics across positions are the same.

At the pro level, there are efforts to obtain more data to measure player's performance. NFL Next Gen Stats (https://operations.nfl.com/gameday/technology/nfl-next-gen-stats/) has been collecting data to assist in these efforts, though it has not been applied at the collegiate level yet.

Model Value & Limitations

Not withstanding, there is at least some value to the combine based on our model. The best indicators from the model suggest that the 40 Yard, Bench Press, and Shuttle, are the most important metrics when it comes to classification of draft status. We also observed that on average, that players who are drafted, tend to perform better at each combine drill than those who were not drafted.

We can confidently conclude that taking in consideration of combine metrics alone does not provide a sure-fire determination of whether someone should be drafted or not. However, we can still use the combine as a *guide* when it comes to predicting whether someone should be drafted or not with an 80% F1-Score accuracy using our best model. Additionally, there is still value in traditional scouting of players such as immeasurable interviews as well as accounting for a player's college career statistics.

Finally, it is also important to note the limitations of the model and that the model does not predict whether a player will be successful at the professional level. The model does not take into account any other 'intangible' measures such as the player's overall character, demeanor, or work ethic, all of which have value and factor into draft status.

Recommendations:

- Focus on targeting players who perform better than the average positional group for each combine drill. These players tend to be drafted players. However, due-dilligence should also be applied when assessing a player's overall value and not solely just on combine metrics.
- When drafting players, priority should be placed the 40 Yard, Bench, and Shuttle as these drills have the highest importance in determining draft status. Height and weight of the player are also significant and should be compared against the relative average weights and heights of the position.
- Seek to incoorporate more data collection at the collegiate level; potentially partner with the NCAA Football.