Preparing Environment

Note: Project was completed in Jupyter Notebooks. It can be run through Jupyter and results will be displayed within notebook.

The goal of this project is to build predictive models to accurately predict whether a student will drop out or not based on a variety of different characteristics. The first step is an Exploratory Data Analysis, in order to better understand the datasets provided and prepare them for modeling. The next step is coding the actual predictive models in an effort to determine which students will drop out. The final step requires performing some tests to conclude which model is most accurate.

The first step is to import any libraries I may need throughout the course of this project.

In [1]:
```
# import numpy for numerical computing
import numpy as np

# import pandas for dataframe use, set display option
import pandas as pd
pd.set_option('display.max_columns', 100)

# import matplotlib for visualization of data
from matplotlib import pyplot as plt
# display plots in given notebook
%matplotlib inline

# import seaborn for easier visualization of graphs
import seaborn as sns
sns.set_style('darkgrid')

# (optional) to supress future warnings
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

Next, I'll read in 3 datasets to start: the 3 most recent datasets saved under Student Static data, Student Progress Data, and Student Financial Aid Data, respectively. I have ran and fit the model on two years of student data, given the time crunch.

In [3]:
```
# Load data from CSV into 3 variables

# student static data (collect Fall + Spring for total 2016 records)
StaticSpring2016 = pd.read_csv('Spring_2016.csv')
StaticFall2015 = pd.read_csv('Fall 2015.csv')

# student progress data (collect Fall + Spring for total 2016 records)
ProgressFall2015 = pd.read_csv('Fall 2015_SP.csv')
ProgressSpring2016 = pd.read_csv('Spring_2016_SP.csv')

# student financial aid data
fa_df = pd.read_csv('2011-2017_Cohorts_Financial_Aid_and_Fafsa_Data.csv'
)
```

In [4]:
```
# display dimensions of data
StaticSpring2016.shape
```

Out[4]: (503, 35)

In [5]:
```
StaticFall2015.shape
```

Out[5]: (1848, 35)

In [6]:
```
ProgressFall2015.shape
```

Out[6]: (5510, 17)

In [7]:
```
ProgressSpring2016.shape
```

Out[7]: (5271, 17)

In [8]:
```
fa_df.shape
```

Out[8]: (13769, 33)

With the data loaded in, the first step will be to merge all the dataframes so we have one centralized dataset.

In [9]:
```
# merge all static data from 2016
StaticTotal = StaticFall2015.merge(StaticSpring2016, how='outer')
```

In [10]:
```
StaticTotal.shape
```

Out[10]: (2351, 35)

In [11]:
```
# merge all progress data from 2016
ProgressTotal = ProgressFall2015.merge(ProgressSpring2016, how='outer')
```

In [12]:
```
ProgressTotal.shape
```

Out[12]: (10781, 17)

In [13]:
```
# merge financial aid data with student info based on StudentID
fa_df = fa_df.rename(columns={'ID with leading': 'StudentID'})
student_FinAid_2016 = fa_df.merge(StaticTotal, how='inner', on=['Student
ID'])
```

In [14]:
```
student_FinAid_2016.shape
```

Out[14]: (2351, 67)

In [15]:
```
student_FinAid_2016.head()
```

Out[15]:

	StudentID	cohort	cohort term	Marital Status	Adjusted Gross Income	Parent Adjusted Gross Income	Father's Highest Grade Level	Mother's Highest Grade Level	Housing	2012 Loan	S
0	341292	2015-16	1	Single	0.0	21623.0	High School	Middle School	On Campus Housing	NaN	
1	348791	2015-16	1	Single	22143.0	0.0	Unknown	Unknown	Off Campus	NaN	
2	347807	2015-16	1	Single	0.0	39975.0	College	College	With Parent	NaN	
3	343175	2015-16	1	Single	0.0	203000.0	College	High School	With Parent	NaN	
4	347137	2015-16	1	Single	4347.0	16788.0	Middle School	Middle School	With Parent	NaN	

In [16]:
```
# drop duplicates before merging final datasets
ProgressTotal.drop_duplicates(subset='StudentID', inplace=True)
```

In [17]:
```
# merge dataframes based on StudentID
student_FinAid_progress_2016 = student_FinAid_2016.merge(ProgressTotal,
how='inner', on=['StudentID'])
```

In [18]:
```
student_FinAid_progress_2016.shape
```

Out[18]: (2351, 83)

In [19]:
```
student_FinAid_progress_2016.head()
```

Out[19]:

	StudentID	cohort	cohort term	Marital Status	Adjusted Gross Income	Parent Adjusted Gross Income	Father's Highest Grade Level	Mother's Highest Grade Level	Housing	2012 Loan	S
0	341292	2015-16	1	Single	0.0	21623.0	High School	Middle School	On Campus Housing	NaN	
1	348791	2015-16	1	Single	22143.0	0.0	Unknown	Unknown	Off Campus	NaN	
2	347807	2015-16	1	Single	0.0	39975.0	College	College	With Parent	NaN	
3	343175	2015-16	1	Single	0.0	203000.0	College	High School	With Parent	NaN	
4	347137	2015-16	1	Single	4347.0	16788.0	Middle School	Middle School	With Parent	NaN	

In [20]:
```
# merge dropout indicator with dataframe based on StudentIDs
dropouts_df = pd.read_csv('DropoutTrainLabels.csv')
student_FinAid_progress_2016 = student_FinAid_progress_2016.merge(dropou
ts_df, how='inner', on=['StudentID'])
```

In [21]:
```
student_FinAid_progress_2016.head()
```

Out[21]:

	StudentID	cohort	cohort term	Marital Status	Adjusted Gross Income	Parent Adjusted Gross Income	Father's Highest Grade Level	Mother's Highest Grade Level	Housing	2012 Loan	S
0	341292	2015-16	1	Single	0.0	21623.0	High School	Middle School	On Campus Housing	NaN	
1	348791	2015-16	1	Single	22143.0	0.0	Unknown	Unknown	Off Campus	NaN	
2	343175	2015-16	1	Single	0.0	203000.0	College	High School	With Parent	NaN	
3	347137	2015-16	1	Single	4347.0	16788.0	Middle School	Middle School	With Parent	NaN	
4	326392	2015-16	1	Married	61811.0	0.0	High School	Middle School	Off Campus	NaN	

In [22]:
```
# fill in all NaN values left in dataframe with 0
# (no info given, can't assume values)
student_FinAid_progress_2016.fillna(0, inplace=True)
student_FinAid_progress_2016.head()
```

Out[22]:

	StudentID	cohort	cohort term	Marital Status	Adjusted Gross Income	Parent Adjusted Gross Income	Father's Highest Grade Level	Mother's Highest Grade Level	Housing	2012 Loan	S
0	341292	2015-16	1	Single	0.0	21623.0	High School	Middle School	On Campus Housing	0.0	
1	348791	2015-16	1	Single	22143.0	0.0	Unknown	Unknown	Off Campus	0.0	
2	343175	2015-16	1	Single	0.0	203000.0	College	High School	With Parent	0.0	
3	347137	2015-16	1	Single	4347.0	16788.0	Middle School	Middle School	With Parent	0.0	
4	326392	2015-16	1	Married	61811.0	0.0	High School	Middle School	Off Campus	0.0	

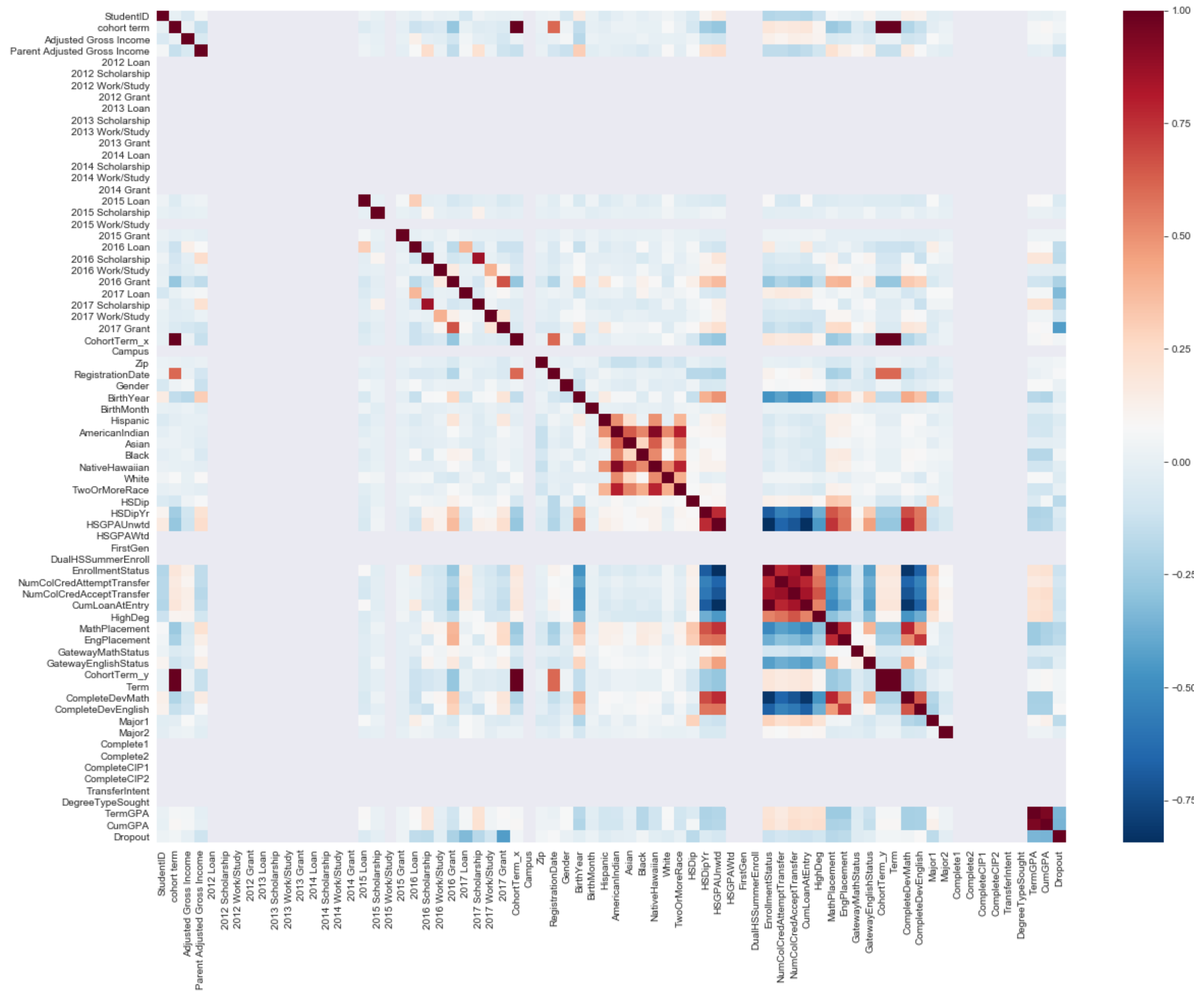
EDA

With the datasets merged and clean, the next step is to begin the Exploratory Data Analysis.

I'll begin by analyzing relationships between all the variables. This can be done initially with a correlation map.

In [23]:
```
# plot revised heat map with less features and numeric values
correlations = student_FinAid_progress_2016.corr()
plt.figure(figsize=(20,15))
sns.heatmap(correlations, cmap='RdBu_r')
```

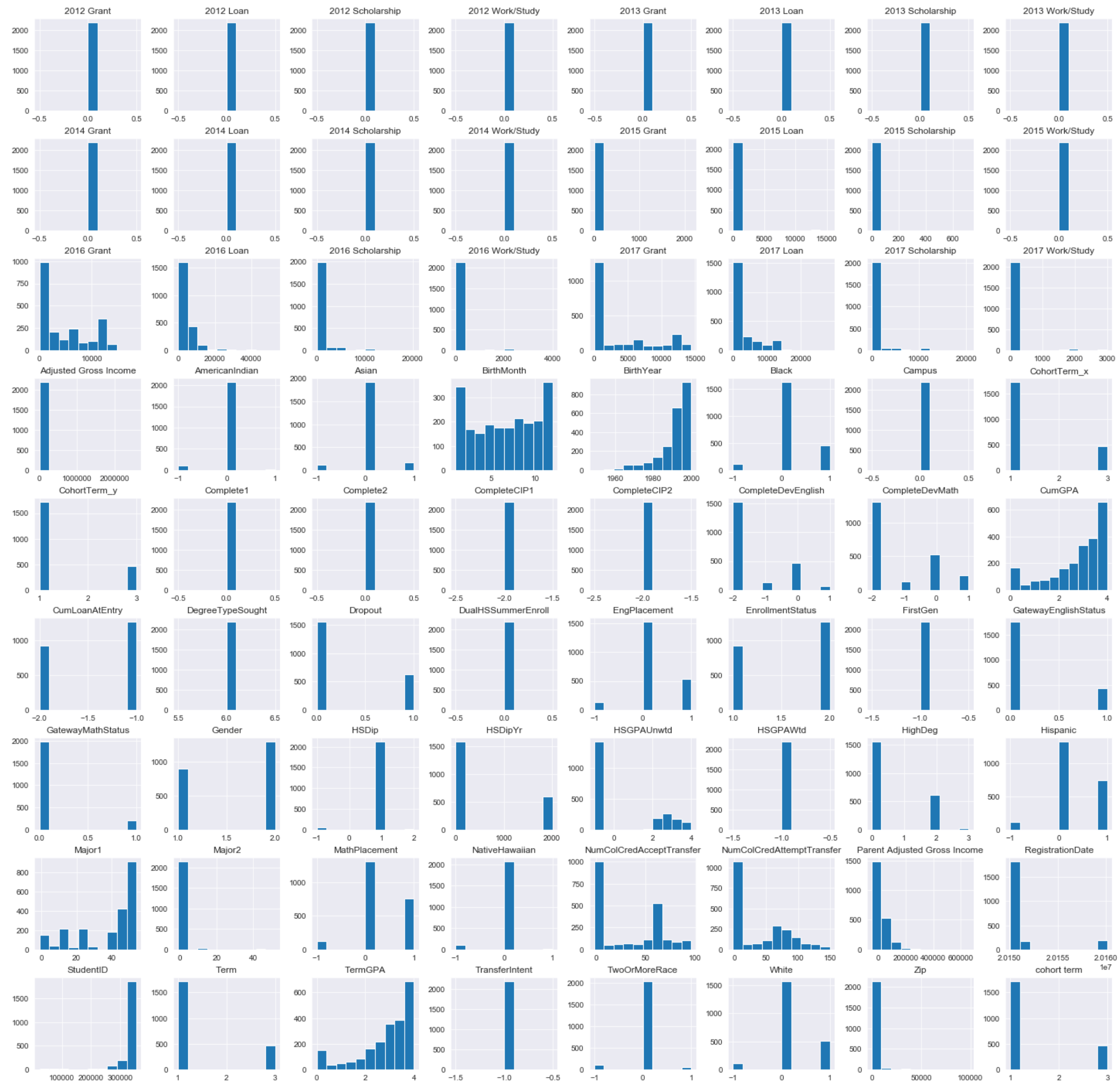
Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x7fee8a05e850>



After looking at the correlation map, we can see a few variables with slight correlation to drop out rates. For example, being in Developmental English, the major, and race can all play small roles in determining if a student will drop out.

Another way to analyze relationships between variables is by plotting distributions of the different features using histograms.

In [24]:
```
# plot histogram grid
student_FinAid_progress_2016.hist(figsize=(25,25))

# remove text for clarity
plt.show()
```



After reviewing all the histograms, nothing appears to be out of the ordinary. Thus, I can move on to reviewing the formal summary statistics for the numerical features of our dataset.

In [25]:
```
# summarize numerical features
student_FinAid_progress_2016.describe()
```

Out[25]:

	StudentID	cohort term	Adjusted Gross Income	Parent Adjusted Gross Income	2012 Loan	2012 Scholarship	201 Work/Stuc
count	2184.000000	2184.000000	2.184000e+03	2184.000000	2184.0	2184.0	2184
mean	334661.211996	1.430403	1.213499e+04	23673.847527	0.0	0.0	0
std	37408.834521	0.822112	6.606193e+04	43708.208893	0.0	0.0	0
min	23606.000000	1.000000	-8.250000e+02	-49406.000000	0.0	0.0	0
25%	342031.750000	1.000000	0.000000e+00	0.000000	0.0	0.0	0
50%	345628.000000	1.000000	0.000000e+00	0.000000	0.0	0.0	0
75%	348010.750000	1.000000	1.246475e+04	31974.500000	0.0	0.0	0
max	355200.000000	3.000000	2.576425e+06	657631.000000	0.0	0.0	0

After reviewing all the numerical data, I can go onto displaying summary statistics for categorial features of the datasets. The numerical summary stats all look normal, so we can move on.

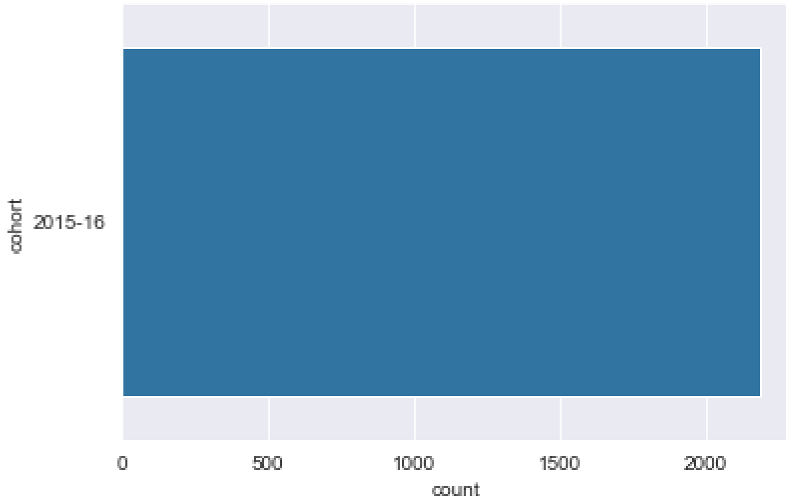
In [26]:
```
# summarize categorical features
student_FinAid_progress_2016.describe(include=['object'])
```

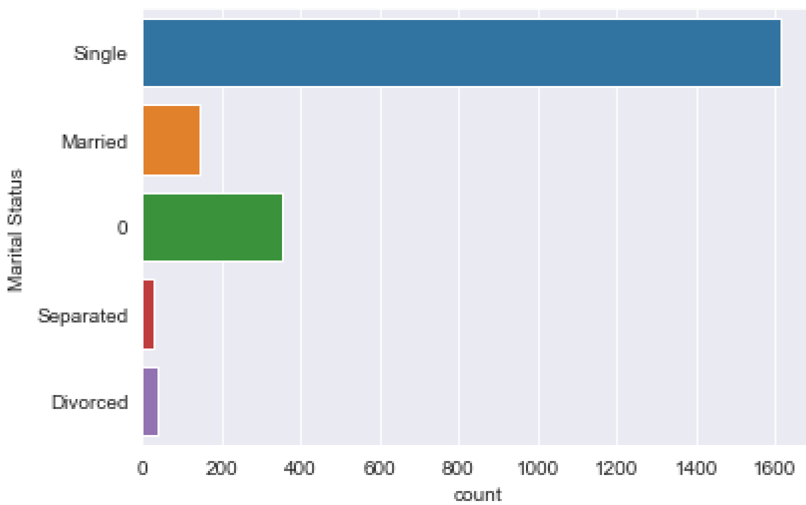
Out[26]:

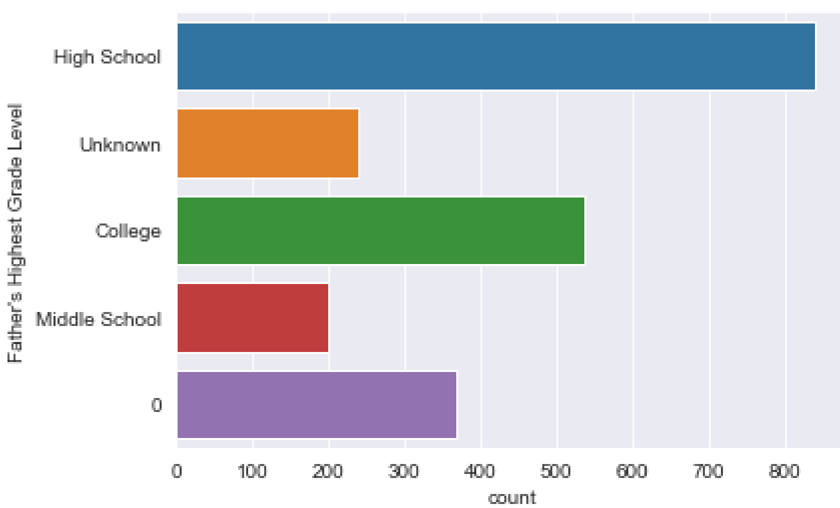
	cohort	Marital Status	Father's Highest Grade Level	Mother's Highest Grade Level	Housing	Cohort_x	Address1	Address2	City	Sta
count	2184	2184	2184	2184	2184	2184	2184	2184	2184	218
unique	1	5	5	5	4	1	2136	74	280	
top	2015-16	Single	High School	High School	Off Campus	2015-16	0	0	Jersey City	N
freq	2184	1617	839	761	848	2184	33	2095	619	21(

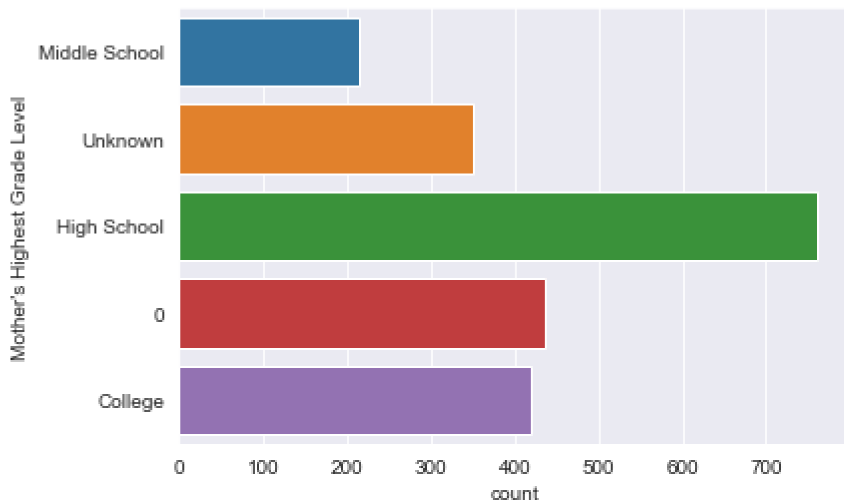
Next, I'll visualize this information, as I did with numerical features. Using Seaborn's countplot function, I can create bar plots to visualize categorical features of the datasets.

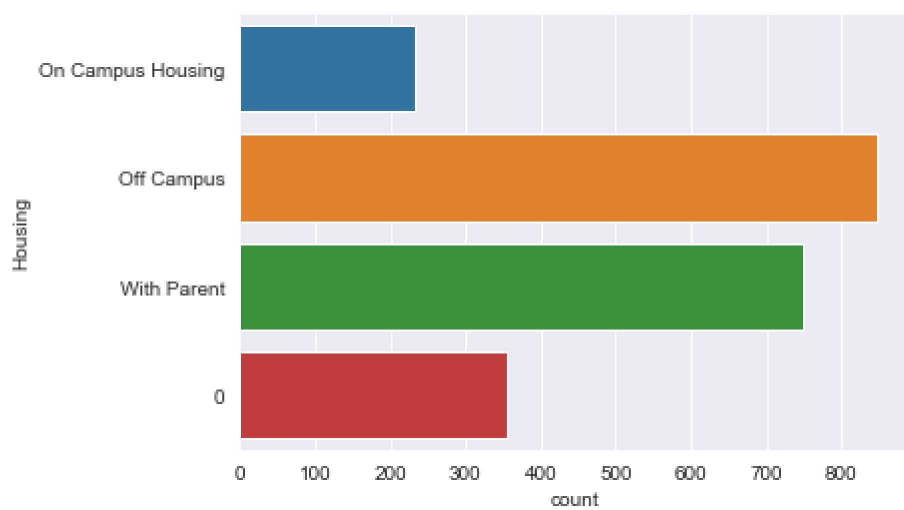
In [29]:
```
# plot bar plots for each categorical feature in each dataset
for feature in student_FinAid_progress_2016.dtypes[student_FinAid_progre
ss_2016.dtypes=='object'].index:
    sns.countplot(y=feature, data=student_FinAid_progress_2016)
    plt.show()
```

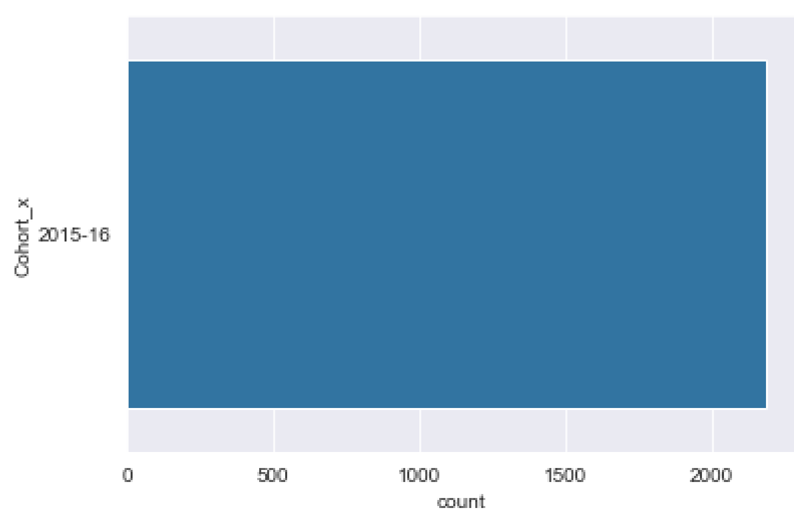

cohort
2015-16
0
500
1000
1500
2000
count

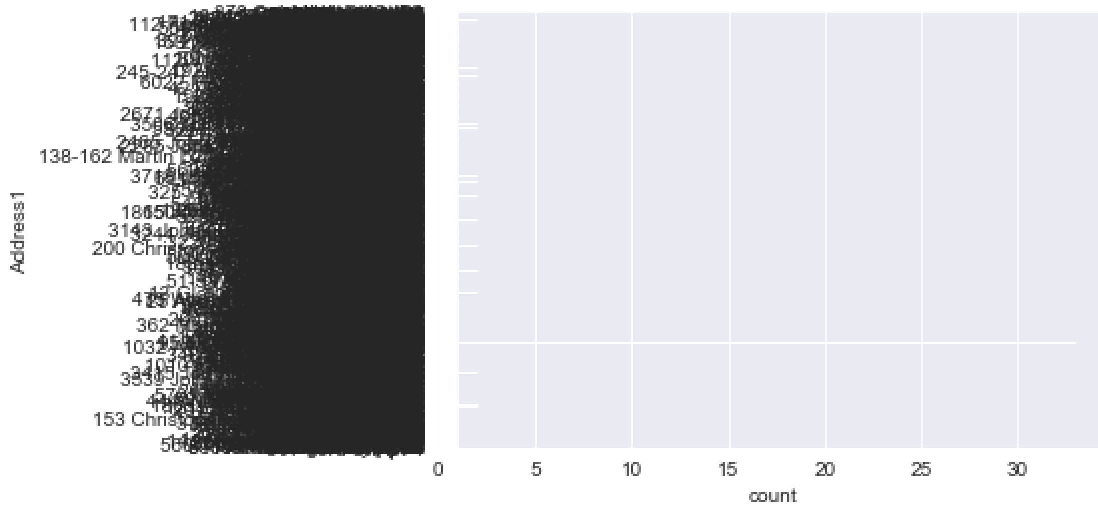

Marital Status
Single
Married
0
Separated
Divorced
0
200
400
600
800
1000
1200
1400
1600
count

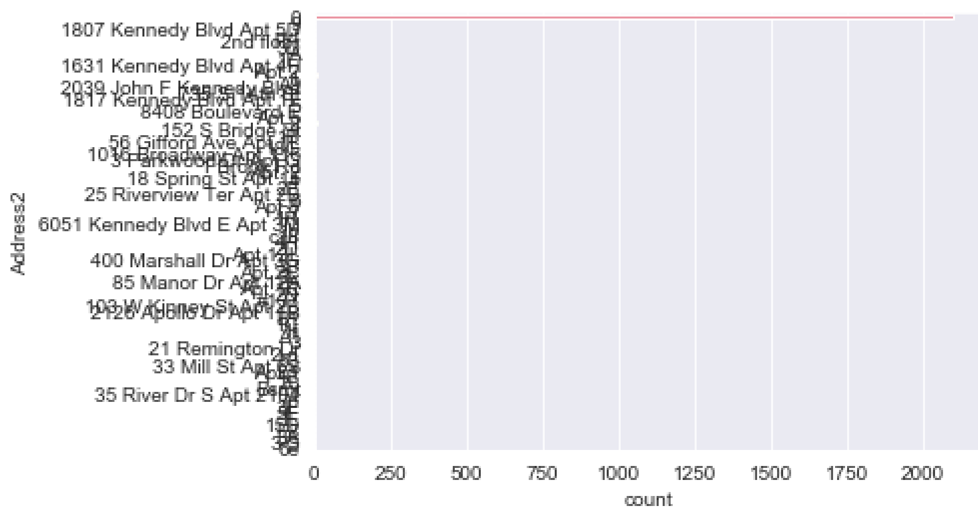

Father's Highest Grade Level
High School
Unknown
College
Middle School
0
0
100
200
300
400
500
600
700
800
count

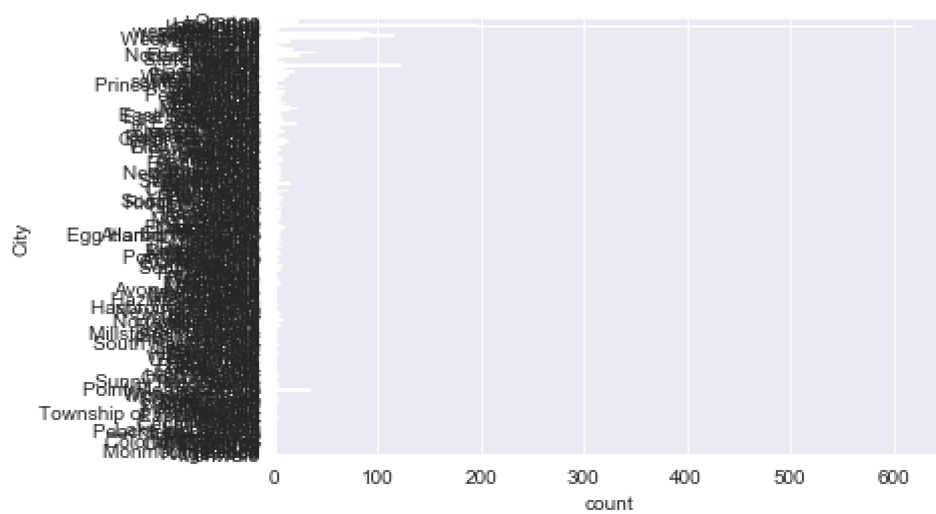

Mother's Highest Grade Level
Middle School
Unknown
High School
0
College
0
100
200
300
400
500
600
700
count

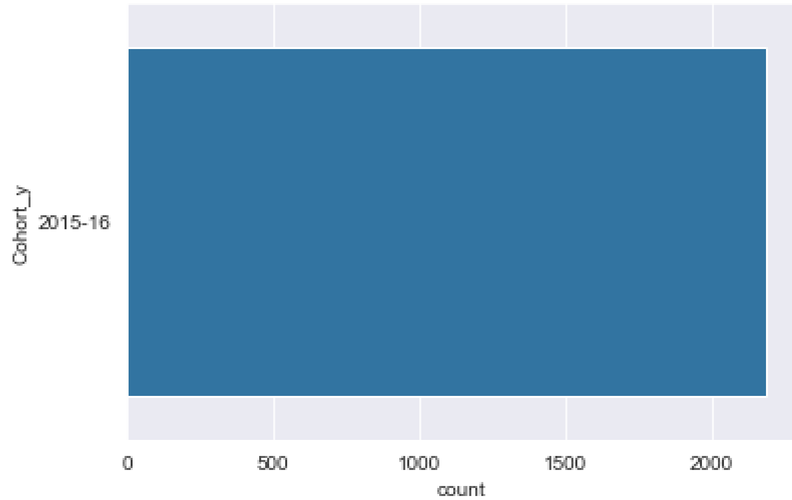

Housing
On Campus Housing
Off Campus
With Parent
0
0
100
200
300
400
500
600
700
800
count

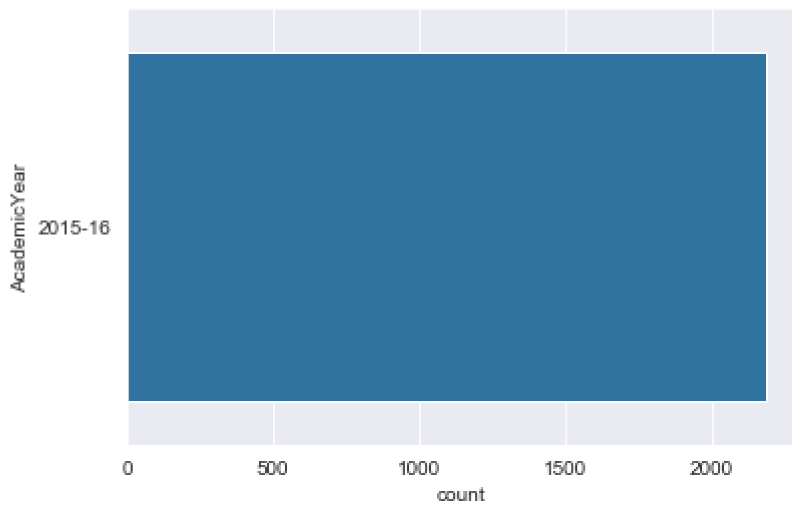

Cohort_x
2015-16
0
500
1000
1500
2000
count


Address1
0
5
10
15
20
25
30
count


Address2
0
250
500
750
1000
1250
1500
1750
2000
count


City
0
100
200
300
400
500
600
count





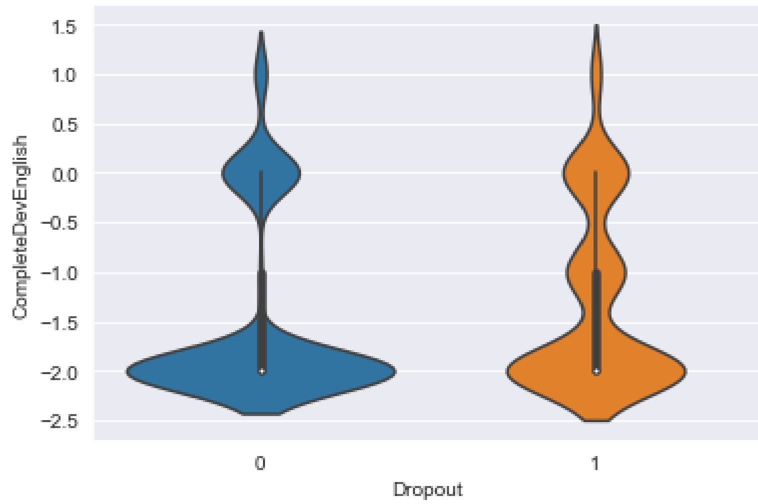
Realizing the state information does not tell us much, it can be removed from the dataset.

In [28]:
```
student_FinAid_progress_2016.drop(['State'], axis=1, inplace=True)
```

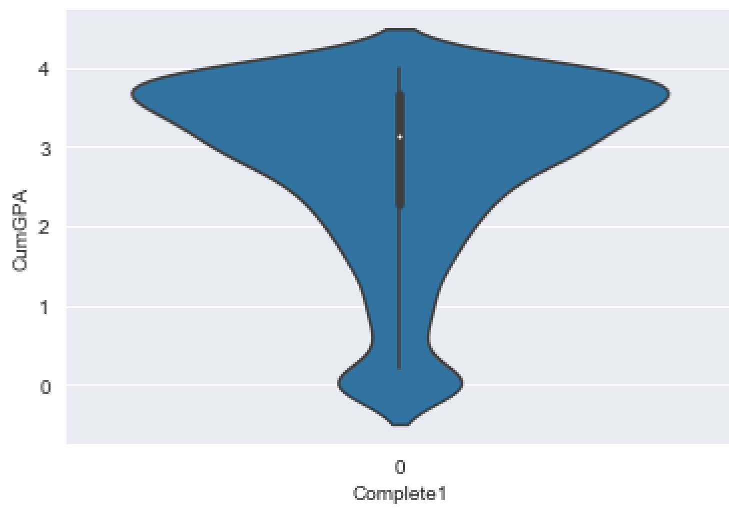
Plotting these bar plots allow for better understanding of how the data is broken down according to different variables, such as housing placement, parental education level, and parental marital status.

Next, I'll create some segementations, as these are powerful ways to cut the data and observe relationships between variables.

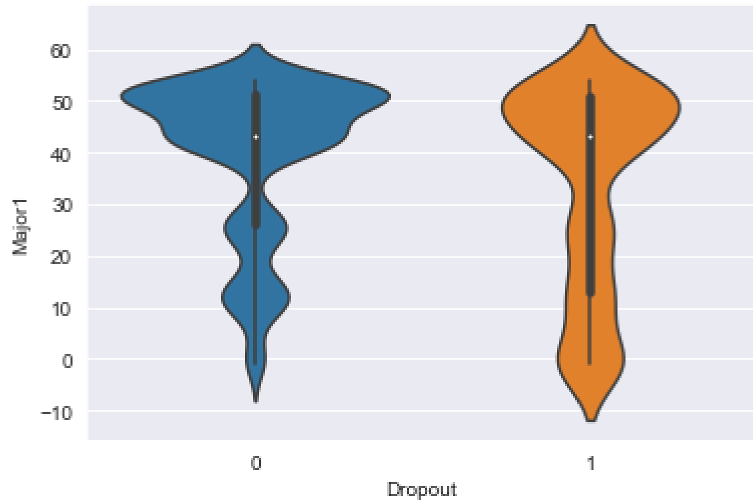
In [30]:
```
sns.violinplot(y='CompleteDevEnglish', x='Dropout', data=student_FinAid_
progress_2016)
plt.show()
```



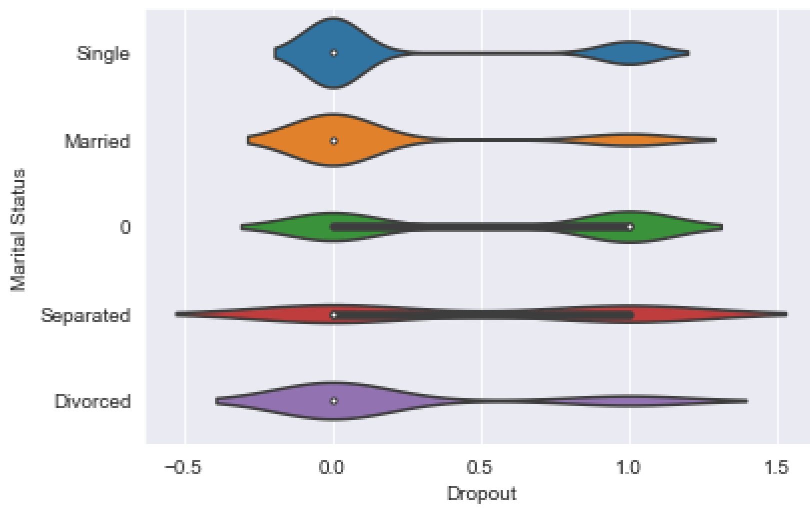
In [31]:
```
sns.violinplot(y='CumGPA', x='Complete1', data=student_FinAid_progress_2
016)
plt.show()
```



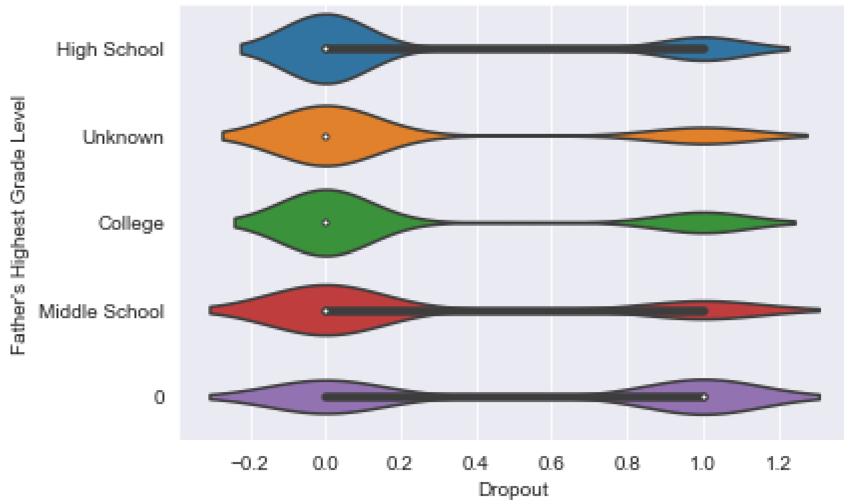
In [32]:
```
sns.violinplot(y='Major1', x='Dropout', data=student_FinAid_progress_201
6)
plt.show()
```



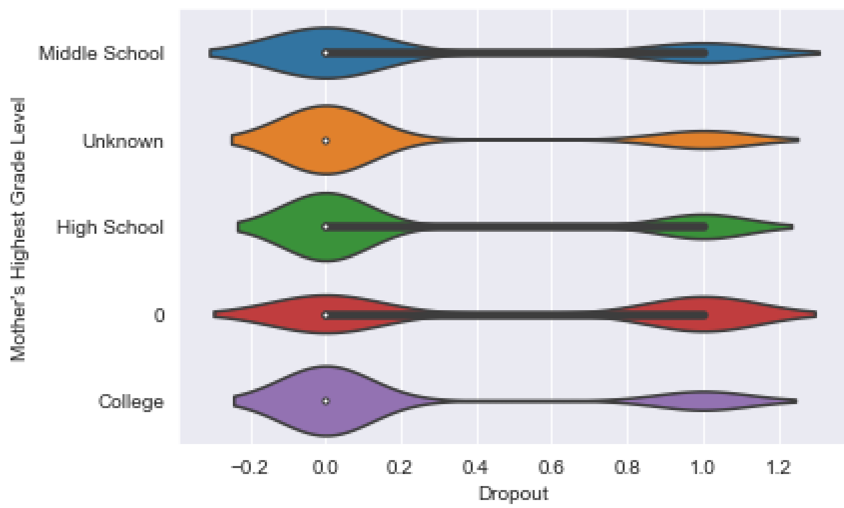
In [33]:
```
sns.violinplot(y='Marital Status', x='Dropout', data=student_FinAid_prog
ress_2016)
plt.show()
```



In [34]:
```
sns.violinplot(y='Father\'s Highest Grade Level', x='Dropout', data=student_FinAid_progress_2016)
plt.show()
```



In [35]:
```
sns.violinplot(y='Mother\'s Highest Grade Level', x='Dropout', data=student_FinAid_progress_2016)
plt.show()
```



The EDA shows us which features are most important when determining who will drop out. It's evident that there is a large amount of students dropping out when their major code falls in the range of 40-50, showing Major1 affects drop out. Additionally, it's clear that quite a lot of drop outs come from Single family homes.

Feature Engineering

Feature engineering consists of engineering appropriate features that allow for further, in-depth analysis. Additionally, I'll remove any features that have none or negative correlations to dropout. This will allow for the model to predict based on only the most relevant features, namely: Marital Status, Major1, and Parents' Education. Finally, I'll drop null values that still may exist in the dataset.

In [36]:
```
# all these variables have no effect on dropout, variables can be droppe
d
student_FinAid_progress_2016.drop(['Campus','Cohort_x', 'CohortTerm_x',
'Cohort_y', 'CohortTerm_y', 'City', 'Zip'], axis=1, inplace=True)

# drop address 1 and 2 (specific address not necessary)
student_FinAid_progress_2016.drop(['Address1', 'Address2'], axis=1, inpl
ace=True)
```

In [37]:
```
# change birth date to age, more intuitive and insightful
student_FinAid_progress_2016['Age'] = 2016 - (student_FinAid_progress_20
16.BirthYear)

# drop registration date --> doesn't tell us much info
student_FinAid_progress_2016.drop(['RegistrationDate'], axis=1, inplace=
True)
```

In [38]:
```
# view changes
student_FinAid_progress_2016.head()
```

Out[38]:

	StudentID	cohort	cohort term	Marital Status	Adjusted Gross Income	Parent Adjusted Gross Income	Father's Highest Grade Level	Mother's Highest Grade Level	Housing	2012 Loan
0	341292	2015-16	1	Single	0.0	21623.0	High School	Middle School	On Campus Housing	0.0
1	348791	2015-16	1	Single	22143.0	0.0	Unknown	Unknown	Off Campus	0.0
2	343175	2015-16	1	Single	0.0	203000.0	College	High School	With Parent	0.0
3	347137	2015-16	1	Single	4347.0	16788.0	Middle School	Middle School	With Parent	0.0
4	326392	2015-16	1	Married	61811.0	0.0	High School	Middle School	Off Campus	0.0

Checking for any null values before moving on to model building.

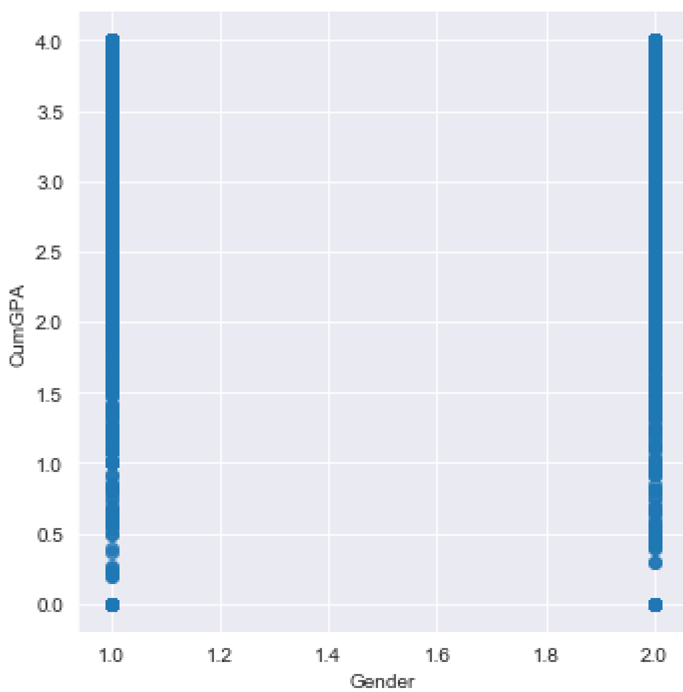
In [39]:
```
student_FinAid_progress_2016.isnull().sum()
```

Out[39]:
```
StudentID               0
cohort                  0
cohort term             0
Marital Status          0
Adjusted Gross Income   0
                       ..
DegreeTypeSought        0
TermGPA                 0
CumGPA                  0
Dropout                 0
Age                     0
Length: 74, dtype: int64
```

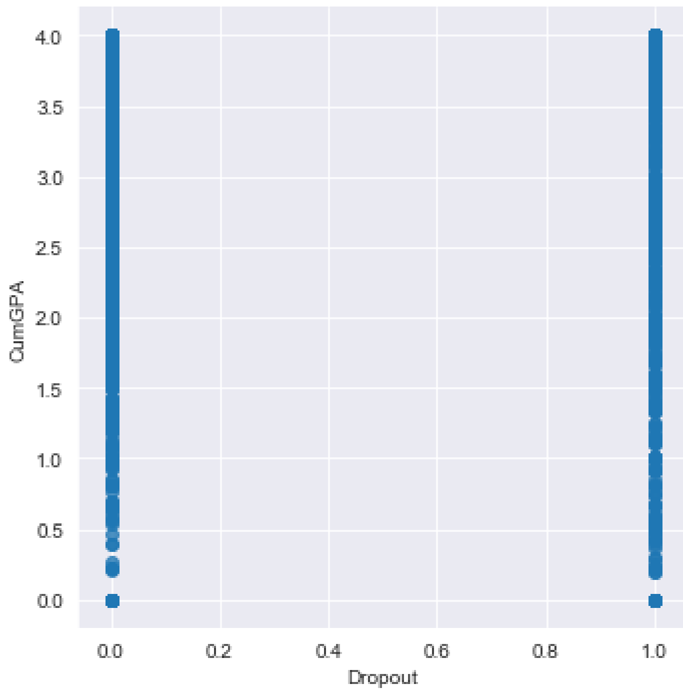
All good there!

With the dataframe prepared, I can conduct analysis on numerous variables all while ensuring I am comparing the correct students to one another.

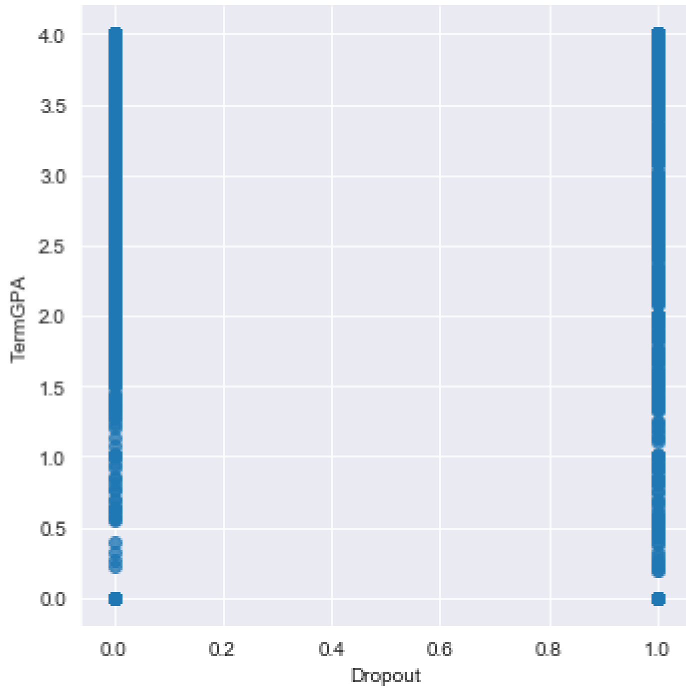
In [40]:
```
# comparing gender and cumulative GPA
sns.lmplot(x='Gender',
           y='CumGPA',
           data=student_FinAid_progress_2016,
           fit_reg=False)
plt.show()
```



In [41]:
```
# comparing gender and cumulative GPA
sns.lmplot(x='Dropout',
           y='CumGPA',
           data=student_FinAid_progress_2016,
           fit_reg=False)
plt.show()
```



In [42]:
```
# comparing majors and dropout rate
sns.lmplot(x='Dropout',
           y='TermGPA',
           data=student_FinAid_progress_2016,
           fit_reg=False)
plt.show()
```



With a complete EDA, a far more thorough understanding of the dataset, and features prime for analysis, I can move on to model building. The first step of this and final step of feature engineering is to ensure all variables are numeric, as the model cannot handle categorical variables.

In [43]:
```
student_FinAid_progress_2016.dtypes[student_FinAid_progress_2016.dtypes=
='object']
```

Out[43]:
```
cohort                          object
Marital Status                  object
Father's Highest Grade Level    object
Mother's Highest Grade Level    object
Housing                         object
AcademicYear                    object
dtype: object
```

In [44]:
```
# use get_dummies to assign categorical variables as numeric
final_df = pd.get_dummies(student_FinAid_progress_2016, columns=['cohor
t','Marital Status', 'Father\'s Highest Grade Level', 'Mother\'s Highest
Grade Level', 'Housing','AcademicYear'])
final_df.head()
```

Out[44]:

	StudentID	cohort term	Adjusted Gross Income	Parent Adjusted Gross Income	2012 Loan	2012 Scholarship	2012 Work/Study	2012 Grant	2013 Loan	2(Scholars
0	341292	1	0.0	21623.0	0.0	0.0	0.0	0.0	0.0	
1	348791	1	22143.0	0.0	0.0	0.0	0.0	0.0	0.0	
2	343175	1	0.0	203000.0	0.0	0.0	0.0	0.0	0.0	
3	347137	1	4347.0	16788.0	0.0	0.0	0.0	0.0	0.0	
4	326392	1	61811.0	0.0	0.0	0.0	0.0	0.0	0.0	

In [45]:
```
# check to make sure no object types remain
final_df.dtypes[final_df.dtypes=='object']
```

Out[45]: `Series([], dtype: object)`

In [46]:
```
# save new dataframe for later use
final_df.to_csv('ABT_REVISED.csv', index=None)
```

Model Building

Now that the dataframe has been cleaned, I have selected relevant features, and categorical variables are ready to be analyzed, I can move on to the actual predictive model building portion of the project.

I will first have to import a few more libraries specific to model building.

In [47]:
```
# import Logistic Regression predictive model
# (+2 extra models are RF and GB)
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier, GradientBoostingCla
ssifier
```

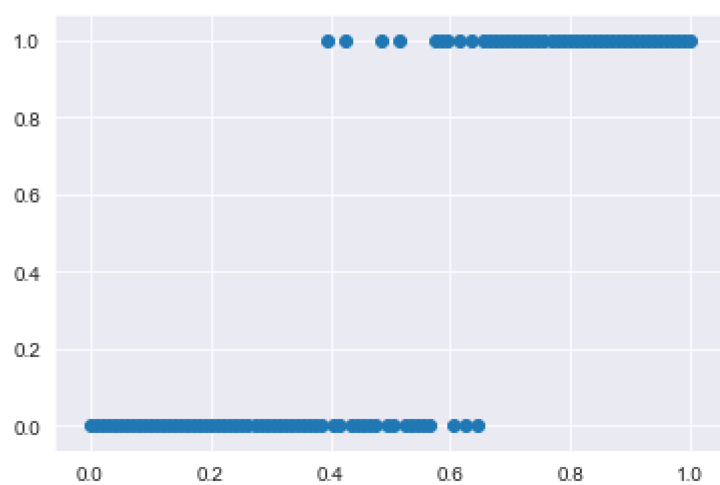
The first step is to initialize the logistic regression model with a random seed.

In [48]:
```
# Input features
x = np.linspace(0, 1, 100)

# Add noise to regression
# (noise is common in all datasets, needs to be represented)
np.random.seed(555)
noise = np.random.uniform(-0.2, 0.2, 100)

# Create target variable, reshape X variable
y = ((x + noise) > 0.5).astype(int)
X = x.reshape(100, 1)
```

In [49]:
```
# define function to fit Logistic Regression model to data
def fit_and_plot_classifier(clf):
    # fit model
    clf.fit(X, y)

    # Predict and take second value of each prediction
    pred = clf.predict_proba(X)
    pred = [p[1] for p in pred]

    # Plot
    plt.scatter(X, y)
    plt.plot(X, pred, 'k--')
    plt.show()

    # Return fitted model and predictions
    return clf, pred
```

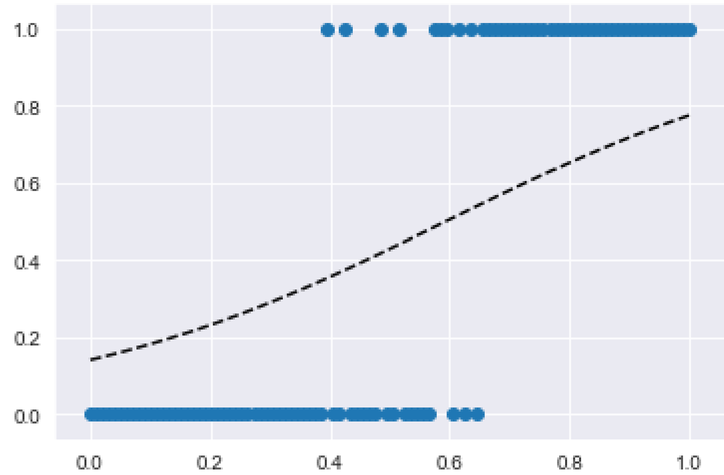
First, I'll graph a scatter plot to ensure that the simulated noisy dataset has been created.

In [50]:
```
plt.scatter(x, y)
plt.show()
```



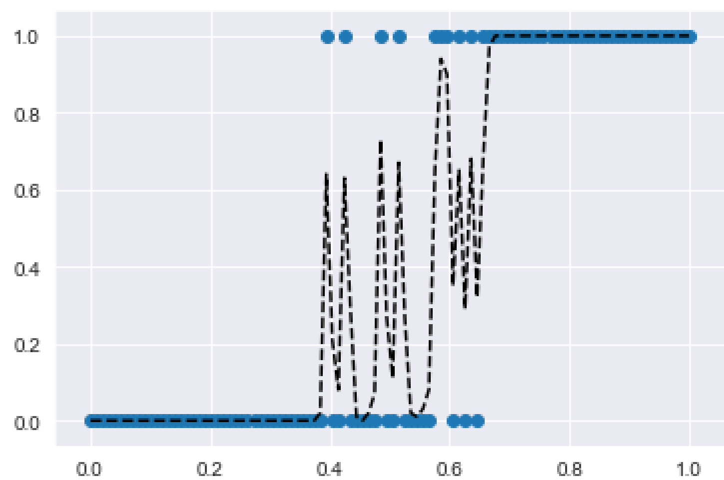
Finally, I'll confirm that a correct helper function has been defined to help me fit and plot classifiers. I will fit the L1-regularized logistic regression on the noisy dataset with C = 0.5 as a test value.

In [51]:
```
clf, pred = fit_and_plot_classifier(LogisticRegression(C=0.5))
```



Next, I'll move on to looking at the tree ensemble model of Random Forest. Using a number of "strong" decision trees and combining their predictions, I can create a predictive model that may be better than our L1 logistic regression.

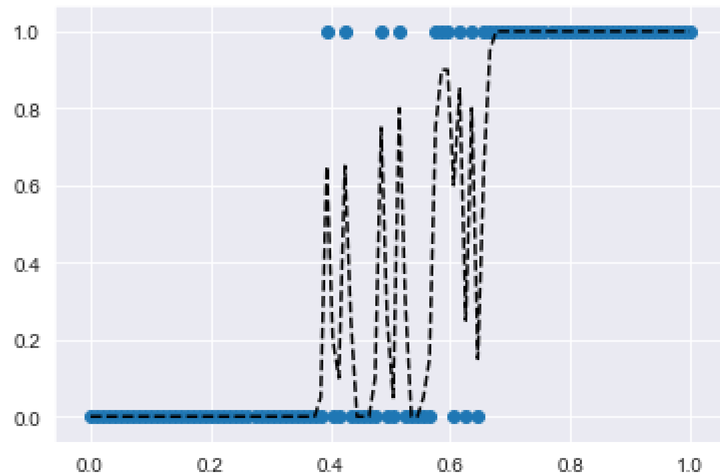
In [52]:
```
# fitting and plotting RF with default hyperparameter values
# and random state = 123
clf, pred = fit_and_plot_classifier(RandomForestClassifier(random_state=
123))
```



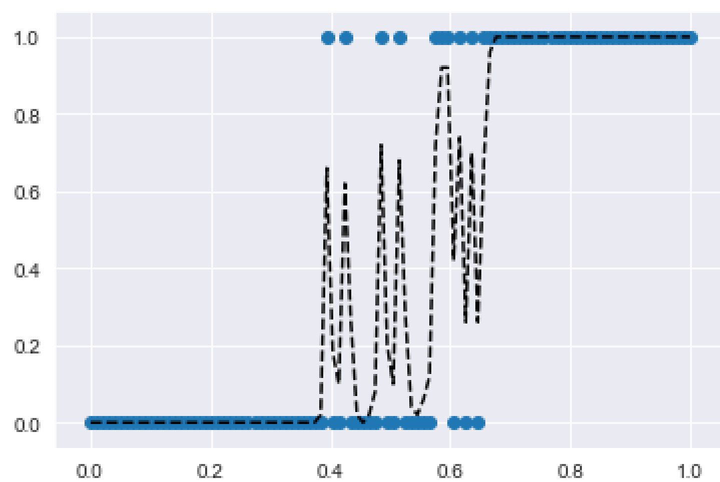
For our given simulated dataset, it's quite evident that the default random forest model suffers from overfitting. I will now try tuning the hyperparameters and overall model complexity to see if I can arrive at a better Random Forest.

In [53]:
```
# tuning number of estimators (# of trees in forest)
for n_trees in [20, 50, 100, 200]:
    print('Number of Trees:', n_trees)
    fit_and_plot_classifier(RandomForestClassifier(random_state=123, n_e
stimators=n_trees))
```

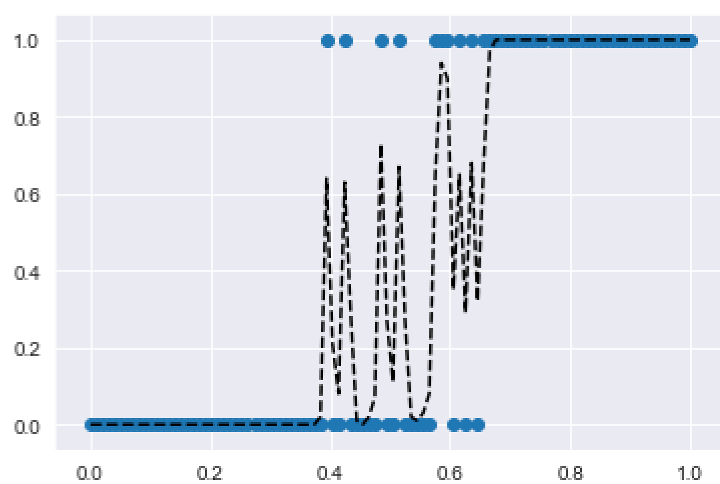
Number of Trees: 20



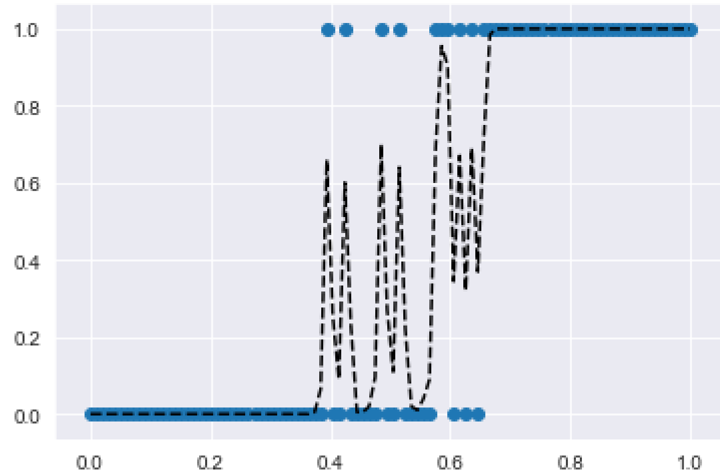
Number of Trees: 50



Number of Trees: 100



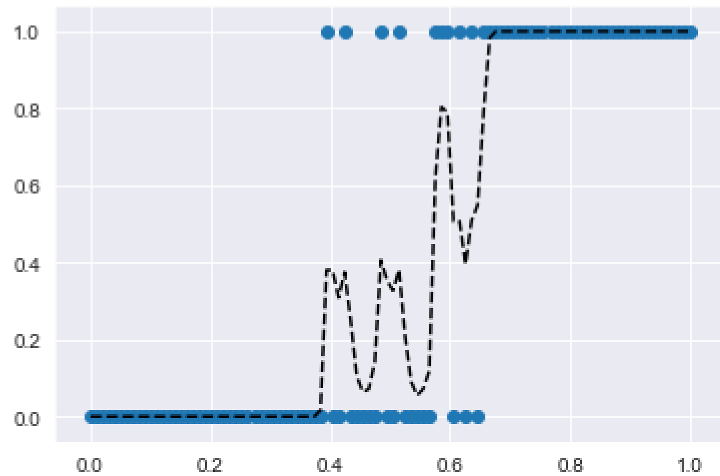
Number of Trees: 200



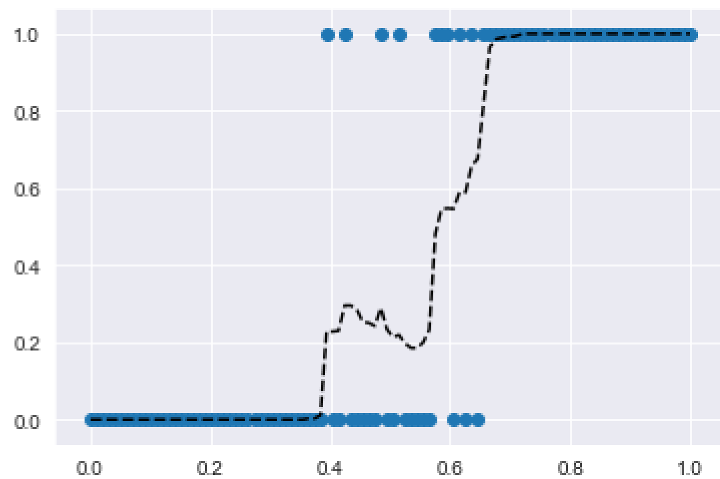
It appears that increasing the number of decision trees does not help the model much. Next, I'll try adjusting leaf size to see if I can further improve the model.

In [54]:
```
for leaf_size in [2, 5, 10, 20]:
    print('Minimum Leaf Size:', leaf_size)
    fit_and_plot_classifier(RandomForestClassifier(random_state=123, min
_samples_leaf=leaf_size))
```

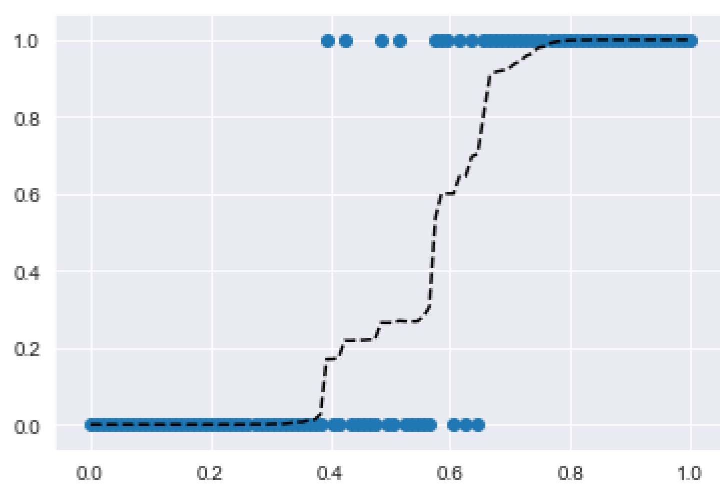
```
Minimum Leaf Size: 2
```



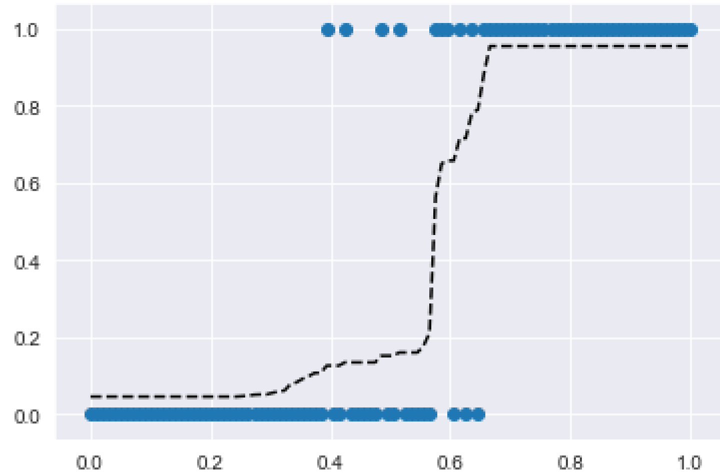

```
Minimum Leaf Size: 5
```




```
Minimum Leaf Size: 10
```



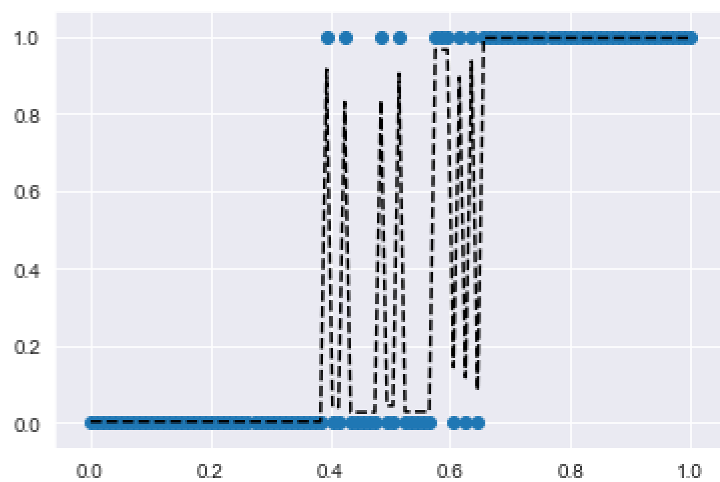

```
Minimum Leaf Size: 20
```



For this dataset, increasing the minimum leaf size in the RF helps the model. Increasing the min. leaf size helps reduce model complexity, addressing the overfitting problem seen earlier.

Finally, I can repeat the same steps of fitting and plotting the model using Gradient Boosted trees. Additionally, tuning hyperparameters in the GB model can assist in finding the best model fit possible.

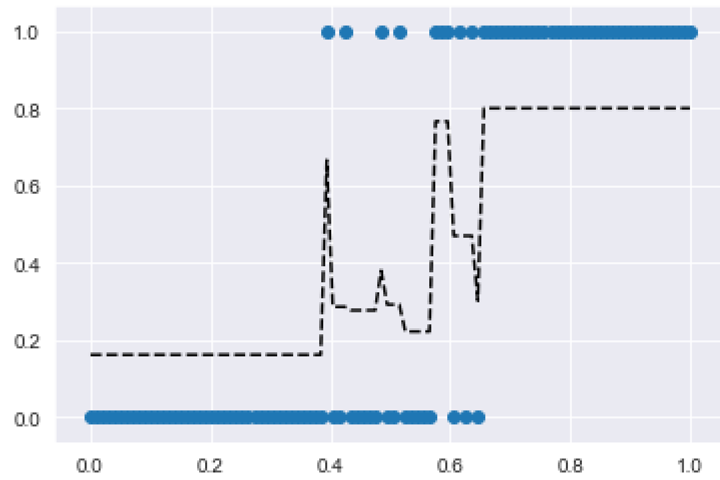
In [55]:
```
# fit and plot GB tree with default hyperparameters and random state = 1
23
clf, pred = fit_and_plot_classifier(GradientBoostingClassifier(random_st
ate=123))
```



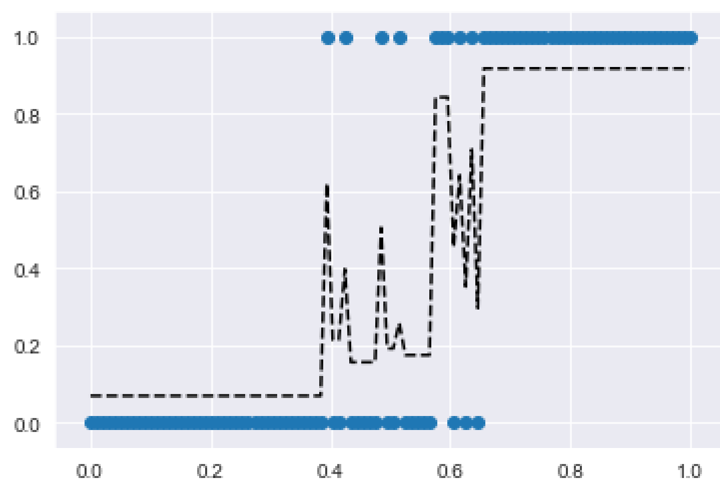
Again, it is evident that the Gradient Boosted tree model suffers from great amounts of overfitting. Tuning the number of estimators (number of trees) and max depth values can again help improve the fit of the model.

In [56]:
```
# changing number of trees
for n_trees in [10, 20, 50, 200]:
    print('Number of Trees:', n_trees)
    fit_and_plot_classifier(GradientBoostingClassifier(random_state=123,
n_estimators=n_trees))
```

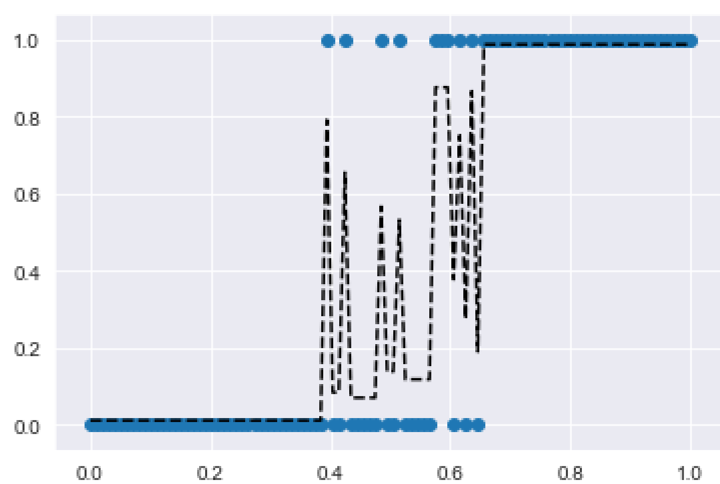
```
Number of Trees: 10
```



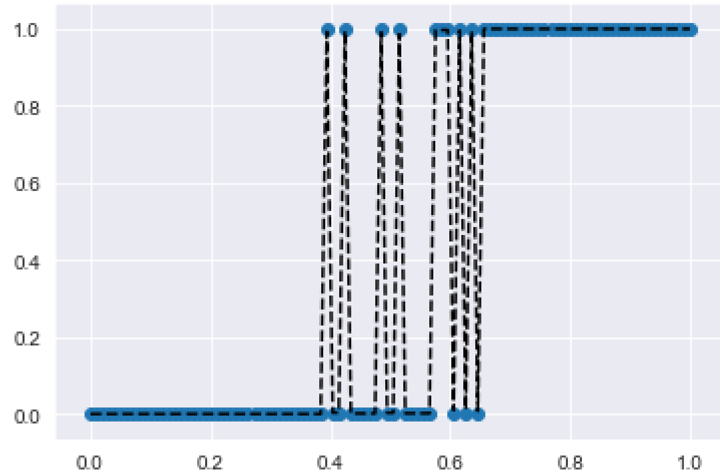

```
Number of Trees: 20
```




```
Number of Trees: 50
```

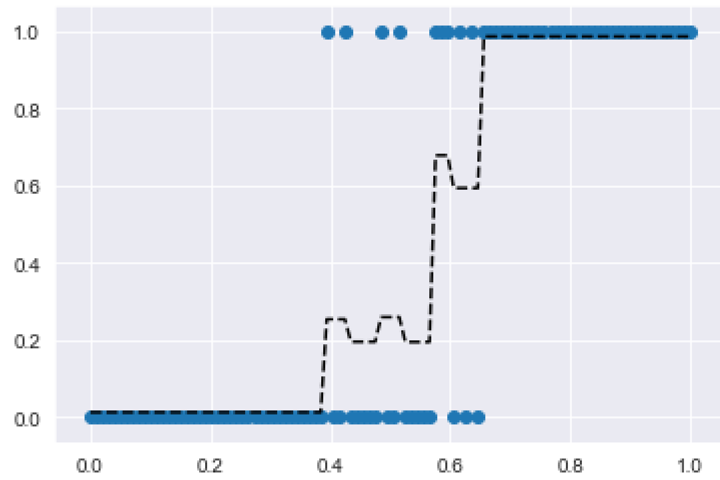



```
Number of Trees: 200
```

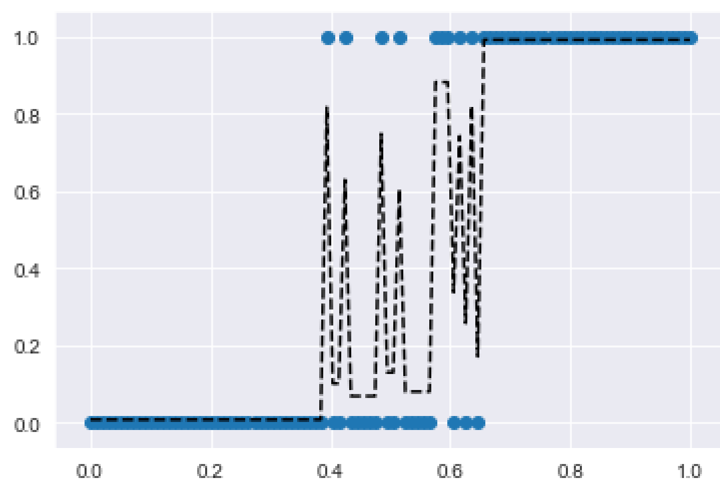

1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

In [57]:
```
# changing max depth value
for depth in [1, 2, 5, 10]:
    print('Max Depth:', depth)
    fit_and_plot_classifier(GradientBoostingClassifier(random_state=123,
max_depth = depth))
```

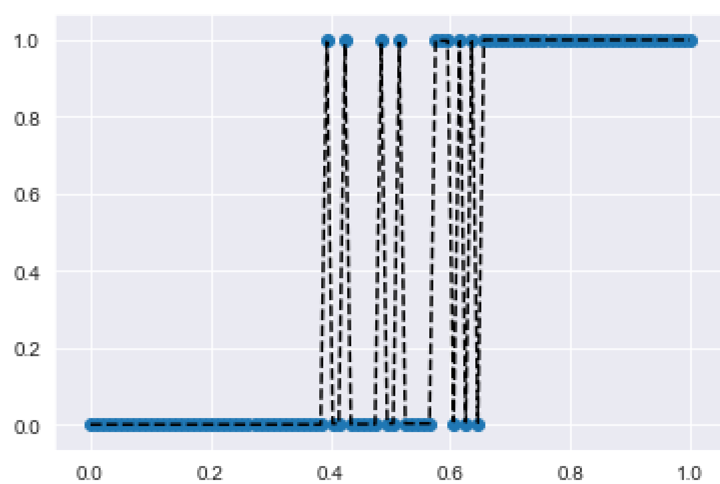
Max Depth: 1



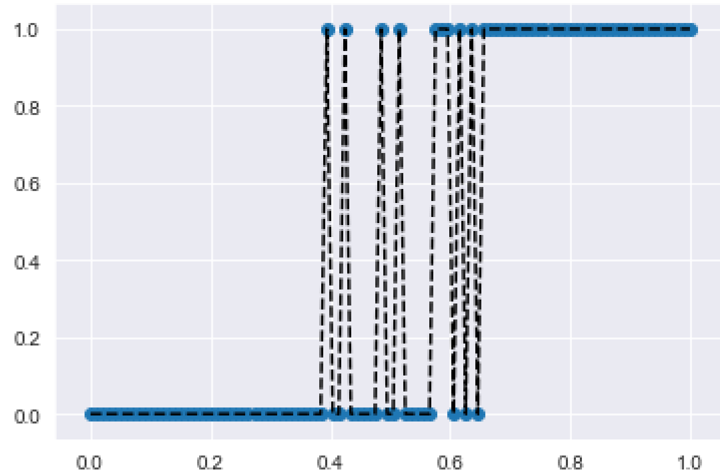
Max Depth: 2



Max Depth: 5



Max Depth: 10



For this dataset, reducing the max depth definitely helps the model. Now that I have created and fitted three different predictive models, I will go on to see which one performs the best!

Model Evaluation

I will first import some necessary libraries required to train and test my models.

In [58]:
```
# to split train and test set
from sklearn.model_selection import train_test_split

# to create model pipelines
from sklearn.pipeline import make_pipeline

# StandardScaler
from sklearn.preprocessing import StandardScaler

# GridSearchCV
from sklearn.model_selection import GridSearchCV

# Classification metrics required
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_curve, roc_auc_score
```

In [59]:
```
# load in ABT saved previously
abt = pd.read_csv('ABT_REVISED.csv')
abt.head()
```

Out[59]:

	StudentID	cohort term	Adjusted Gross Income	Parent Adjusted Gross Income	2012 Loan	2012 Scholarship	2012 Work/Study	2012 Grant	2013 Loan	2(Scholars
0	341292	1	0.0	21623.0	0.0	0.0	0.0	0.0	0.0	
1	348791	1	22143.0	0.0	0.0	0.0	0.0	0.0	0.0	
2	343175	1	0.0	203000.0	0.0	0.0	0.0	0.0	0.0	
3	347137	1	4347.0	16788.0	0.0	0.0	0.0	0.0	0.0	
4	326392	1	61811.0	0.0	0.0	0.0	0.0	0.0	0.0	

In [60]:
```
# check for categorical variables (would cause error in model)
abt.dtypes[abt.dtypes=='object']
```

Out[60]:
```
Series([], dtype: object)
```

Now, I'll split my dataset into separate training and test sets.

In [61]:
```
# separate dataframe into objects for target var 'y', input features 'x'
# target variable
y = abt['Dropout']

# input features
x = abt.drop('Dropout', axis=1)
```

With input features and a target variable defined, I can now split them into training and test sets. I will have a test_size = 0.2, 20% of the dataset is for testing. I'll have a random_state = 1234 and stratify my splits in order to ensure balance across subsets of data.

In [62]:
```
# split x and y into train and test sets
x_train, x_test, y_train, y_test = train_test_split(x, y,
                                                    test_size=0.2,
                                                    random_state=1234,
                                                    stratify=abt['Dropou
t'])

# print number of observations in each split
print(len(x_train), len(x_test), len(y_train), len(y_test))
```

```
1747 437 1747 437
```

Now that I have training and testing datasets, I can begin setting up my preprocessing pipelines for each algorithm written earlier. Standardizing our features across splits (or bringing them to scale) is useful to ensure accurate predictions.

My pipeline dictionary is defined as below:

1. 'l1' for L1-regularized logistic regression
2. 'l2' for L2-regularized logistic regression
3. 'rf' for Random Forest
4. 'gb' for Gradient Boosted Tree

In [63]:
```
# Pipeline dictionary
pipelines = {
    'l1' : make_pipeline(StandardScaler(),
                         LogisticRegression(penalty='l1' , random_state=
123 , solver='liblinear')),
    'l2' : make_pipeline(StandardScaler(),
                         LogisticRegression(penalty='l2' , random_state=
123)),
    'rf' : make_pipeline(StandardScaler(), RandomForestClassifier(random
_state=123)),
    'gb' : make_pipeline(StandardScaler(), GradientBoostingClassifier(ra
ndom_state=123))
}
```

With pipelines ready for each model, I can move on to declaring hyperparameters to tune.

In [64]:
```
# declare hyperparameter grids for L1 regression and L2 regression

# Logistic Regression hyperparameters
l1_hyperparameters = {
    'logisticregression__C' : [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5,
10, 50, 100, 500, 1000],
}

l2_hyperparameters = {
    'logisticregression__C' : [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5,
10, 50, 100, 500, 1000],
}

# Random Forest hyperparameters
rf_hyperparameters = {
    'randomforestclassifier__n_estimators': [100, 200],
    'randomforestclassifier__max_features': ['auto', 'sqrt', 0.33],
    'randomforestclassifier__min_samples_leaf': [1, 3, 5, 10]
}

# Boosted Tree hyperparameters
gb_hyperparameters = {
    'gradientboostingclassifier__n_estimators': [100, 200],
    'gradientboostingclassifier__learning_rate': [0.05, 0.1, 0.2],
    'gradientboostingclassifier__max_depth': [1, 3, 5]
}
```

I can now create a hyperparameters dictionary to store the above parameters.

In [65]:
```
# Create hyperparameters dictionary
hyperparameters = {
    'l1' : l1_hyperparameters,
    'l2' : l2_hyperparameters,
    'rf' : rf_hyperparameters,
    'gb' : gb_hyperparameters
}
```

Finally, I am ready to begin fitting and tuning my models using cross-validation. I will create a fitted_models dictionary to store all my fitted models that have been tuned using cross-validation.

In [66]:
```
# create fitted models dictionary
fitted_models = {}

# Loop through model pipelines, tuning each one and saving to fitted_mod
el
for name, pipeline in pipelines.items():
    # Create cross-validation object from pipeline and hyperparameters
    model = GridSearchCV(pipeline, hyperparameters[name], cv=10, n_jobs=
-1)

    # Fit model on X_train, y_train
    model.fit(x_train, y_train)

    # Store model in fitted_models[name]
    fitted_models[name] = model

    # Print '{name} has been fitted'
    print(name, 'has been fitted.')
```

```
l1 has been fitted.
l2 has been fitted.
rf has been fitted.
gb has been fitted.
```

Now that all my models have been successfully fitted to the dataset, I can finally move on to evaluating my models and picking the best one. L1 and L2 Logistic Regression, Lasso and Ridge regression respectively, are not as complex as the Random Forest and Gradient Boosted Tree models, but they may fit the data better.

First, I'll display the best score attribute for each fitted model.

In [67]:
```
# displaying best_score_ for each model
for name, model in fitted_models.items():
    print(name, model.best_score_)
```

```
l1 0.8775172413793102
l2 0.8649096880131364
rf 0.8843809523809524
gb 0.8826765188834156
```

Now that I've seen a general estimate of the model's performance, I can calculate the AUROC performance of each model on the test set and determine the best overall predictive model! The AUROC score helps determine how good the model is at determining whether a student will drop out (0 for non-dropout, 1 for dropout). AUROC shows the how capable the model is at predicting the proper label.

In [68]:
```
for name, model in fitted_models.items():
    pred = model.predict_proba(x_test)
    pred = [p[1] for p in pred]

    print(name, roc_auc_score(y_test, pred))
```

```
l1 0.9531210126065432
l2 0.9478129944367886
rf 0.9529168580615526
gb 0.9523299137447047
```

The random forest model has the highest test AUROC and the highest cross-validated score.

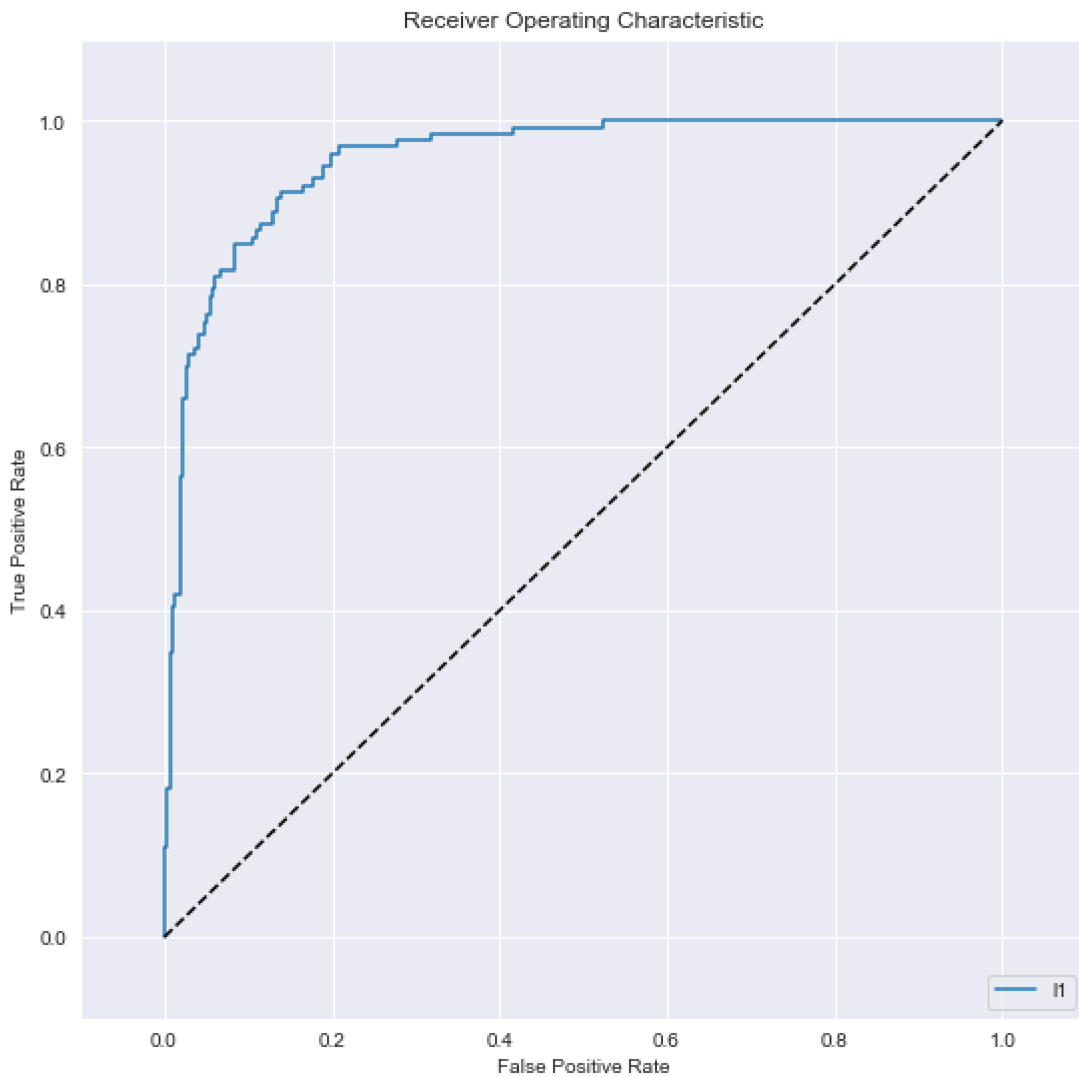
For visualization, plotting the ROC curve for the RF model can help better understand the score.

In [69]:
```
# Predict PROBABILITIES using Random Forest regression
pred = fitted_models['rf'].predict_proba(x_test)

# Get just the prediction for the positive class (1)
pred = [p[1] for p in pred]

# Display first 10 predictions
print(np.round(pred[:10], 2))
```

```
[0.36 0.15 0.08 0.04 0.92 0.54 0.08 0.8  0.3  0.1 ]
```

In [70]:
```
# Calculate ROC curve from y_test and pred
fpr, tpr, thresholds = roc_curve(y_test, pred)
```

In [71]:
```
# Initialize figure
fig = plt.figure(figsize=(9,9))
plt.title('Receiver Operating Characteristic')

# Plot ROC curve
plt.plot(fpr, tpr, label='l1')
plt.legend(loc='lower right')

# Diagonal 45 degree line
plt.plot([0,1],[0,1],'k--')

# Axes limits and labels
plt.xlim([-0.1,1.1])
plt.ylim([-0.1,1.1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



In [72]:
```
# Calculate AUROC
print(roc_auc_score(y_test, pred))
```

```
0.9529168580615526
```

So, all in all, the random forest model has a 95.29% chance of distinguishing between a positive class observation and a negative class observation (chance of determining whether a student will drop out or not).

With a solid accuracy of our predictive model, the final step of the project is to run the model on the complete dataset and store the results to a final CSV file.

In [73]:
```
# getting prediction and storing
pred = fitted_models['rf'].predict(x_test)
studentIDs = x_test.loc[:, ['StudentID']]
studentIDs['Dropout'] = pred
```

In [74]:
```
studentIDs.style.hide_index()
studentIDs.to_csv('Final_Predictions.csv')
```

In [75]:
```
# view final product
studentIDs
```

Out[75]:

	StudentID	Dropout
805	347361	0
622	349072	0
1094	321455	0
590	342920	0
2167	354910	1
...	...	...
1771	307540	0
732	347308	0
995	275869	1
1773	309022	0
1343	347652	0

437 rows × 2 columns

Final Comments: Given more time, I would have added a few more blocks of code to feature engineering, in order to:

1. Aggregate students' financial aid values
2. Impute demographics information based on Zip codes for students missing this information
3. Pulled the last term GPA of students for a more comprehensive analysis.