Bank Marketing Campaign Subscriptions

Purpose of This Project:

As a Data Scientist on a software marketing team, a model predicting whether or not a prospective lead would be an appropriate target for a marketing campaign could be invaluable to sales teams in multiple industries. While this project uses data from a Portuguese bank campaign from about 10 years ago, so it is rather old, a number of the elements of in this data are reminiscent of the very immediate problem facing my team. It blends both socio-economic factors and demographic information about the clients and it also suffers from massive class imbalance problems. I will be using three approaches to address class imbalance:

- No class balancing
- Undersampling the majority class
- Data generation in the minority class using Synthetic Minority Oversampling Technique or SMOTE

My intention is to practice techniques and the different approaches requirement to modeling class imbalanced datasets for application to current real-world datasets.

Objective:

To determine the most important factors that affect whether or not a prospective client of a marketing campaign will purchase a subscription for a Portuguese bank. The data includes a massive amount of information, both regarding the client and the state of the economy at the time of the solicitation.

The Data:

A detailed description of the data and its source can be found at https://www.kaggle.com/pankajbhowmik/bank-marketing-campaign-subscriptions (https://www.kaggle.com/pankajbhowmik/bank-marketing-campaign-subscriptions). Here are the

column descriptions as presented by the Kaggle project:

1. Variables that describing attributes related directly to the client:

- age
- job: type of job (e.g. 'admin', 'technician', 'unemployed', etc)
- marital: marital status ('married', 'single', 'divorced', 'unknown')
- education: level of education ('basic.4y', 'high.school', 'basic.6y', 'basic.9y','professional.course', 'unknown','university.degree','illiterate')
- default: if the client has credit in default ('no', 'unknown', 'yes')
- housing: if the client has housing a loan ('no', 'unknown', 'yes')
- loan: if the client has a personal loan ? ('no', 'unknown', 'yes')

2. Variables related to the last contact of the current campaign:

- contact: type of communication ('telephone', 'cellular')
- month: month of last contact
- dayofweek: day of last contact
- duration: call duration (in seconds)

3. Other variables related to the campaign(s):

- campaign: number of contacts performed during this campaign and for this client
- pdays: number of days passed by after the client was last contacted from a previous campaign
- previous: number of contacts performed before this campaign and for this client
- poutcome: outcome of previous marketing campaign ('nonexistent', 'failure', 'success')

4. Socioeconomic variables:

- emp.var.rate: employement variation rate - quarterly indicator
- cons.price.idx: consumer price index - monthly indicator
- cons.conf.idx: consumer confidence index - monthly indicator
- euribor3m: euribor 3 month rate - daily indicator
- nr.employed: number of employees - quarterly indicator

The target variable is 'subscribed_yes', which is true is the prospect purchased the subscription or false if they did not.

Method:

I will be explorating multiple classification models using accuracy and the f1_score to evaluate their effectiveness. Knowing the most important factors, and even probabilities of whether or not a prospect will purchase a subscription, will help future marketing campaigns allocate resources most effectively. In this case, both False Negatives and False Positives can be problematic, as the marketing campaign might focus too much energy on False Positives if our Precision is low and they might miss out on solid prospectives if the Recall score is too low. I will be examining all of the above, but the overall effectiveness of the model will primarily focus on a blend of Accuracy, Precision, Recall, F1-Score and ROC-AUC. For my GridSearch model tuning, I will maximizing the model's F1-Score to strike the best possible balance between Precision and Recall.

I will use the following models to investigate the data:

- Logistic Regression
- K Nearest Neighbors
- Gaussian Naive Bayes
- Basic Decision Tree
- Ada Boosted Random Forest
- XGBoosted Random Forest

I will create two versions of each model: a baseline with no hyperparameter tuning and a tuned model using Sci Kit Learn's GridSearch module.

Functions

For source on every function, please visit my Utils.py file in this same directory. I created the class *Model_Analysis* for quick comparisons across all of my various scoring metrics. The code of the class object can be found in the Utils.py file, but it will provide 3 important functions in my analysis:

```
1. Fit the model in question against all three data sets
2. Provide a table of Train and Test scores for all three sets
3. Directly compare all metrics of the testing scores and highlight
the best value in each category
4. Plot confusion matricies of the three test scores for side-by-si
de comparison
```

This was my most used set of functions in this project and I will use them again in future endeavors.

In [1]:
```
import pandas as pd
import numpy as np
import pandas as pd
import numpy as np
import warnings
import matplotlib.pyplot as plt
from utils import Model_Analysis, scores
%matplotlib inline
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler, StandardScaler, RobustScale
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import precision_score, recall_score, accuracy_score,
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier
```

Data Exploration

In [2]:
```
df = pd.read_csv('Bank_Campaign.csv',sep=';')
df.head()
```

Out[2]:

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	..
0	56	housemaid	married	basic.4y	no	no	no	telephone	may	mon	..
1	57	services	married	high.school	unknown	no	no	telephone	may	mon	..
2	37	services	married	high.school	no	yes	no	telephone	may	mon	..
3	40	admin.	married	basic.6y	no	no	no	telephone	may	mon	..
4	56	services	married	high.school	no	no	yes	telephone	may	mon	..

5 rows × 21 columns

In [3]:
```
df.describe()
```

Out[3]:

	age	duration	campaign	pdays	previous	emp.var.rate	cons.pr
count	41188.00000	41188.000000	41188.000000	41188.000000	41188.000000	41188.000000	41188.(
mean	40.02406	258.285010	2.567593	962.475454	0.172963	0.081886	93.5
std	10.42125	259.279249	2.770014	186.910907	0.494901	1.570960	0.5
min	17.00000	0.000000	1.000000	0.000000	0.000000	-3.400000	92.2
25%	32.00000	102.000000	1.000000	999.000000	0.000000	-1.800000	93.(
50%	38.00000	180.000000	2.000000	999.000000	0.000000	1.100000	93.7
75%	47.00000	319.000000	3.000000	999.000000	0.000000	1.400000	93.9
max	98.00000	4918.000000	56.000000	999.000000	7.000000	1.400000	94.7

In [4]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 21 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   age             41188 non-null  int64  
 1   job             41188 non-null  object 
 2   marital         41188 non-null  object 
 3   education       41188 non-null  object 
 4   default         41188 non-null  object 
 5   housing         41188 non-null  object 
 6   loan            41188 non-null  object 
 7   contact         41188 non-null  object 
 8   month           41188 non-null  object 
 9   day_of_week     41188 non-null  object 
 10  duration        41188 non-null  int64  
 11  campaign        41188 non-null  int64  
 12  pdays           41188 non-null  int64  
 13  previous        41188 non-null  int64  
 14  poutcome        41188 non-null  object 
 15  emp.var.rate    41188 non-null  float64
 16  cons.price.idx  41188 non-null  float64
 17  cons.conf.idx   41188 non-null  float64
 18  euribor3m       41188 non-null  float64
 19  nr.employed     41188 non-null  float64
 20  subscribed      41188 non-null  object 
dtypes: float64(5), int64(5), object(11)
memory usage: 6.6+ MB
```

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

Out[5]:
```
(41188, 21)
```

Data Quirks

There isn't a huge amount that's problematic about this data though there are a few quirks that I've listed below:

- Eldest age is 98, youngest is 17, but beyond that, there are no obvious outliers (no children below 17 were solicited for, well, obvious reasons).
- Duration - 4 are 0 seconds and the max is 81 minutes long.
- Job - 330 unknown
- Marital - 80 unknown
- Education - 1731 unknown - those affected have a yes/no rate of 14.5%/85.5%, similar to larger dataset
- Default - only 3 'yes' values. I will drop this column.
- Housing - 'unknown' has a yes/no of 10.8%/89.2%, similar to larger dataset. I will include unknowns and see if it has any affect on the final outcome.
- Loan - same values as Housing, probably both on the same survey
- Very few calls in December, most in May

- Day of the week is ONLY weekdays, pretty evenly balanced
- pdays - 39673 people at 999. Dropping column would be best choice
- previous - lots of 0 values, but not unusable.
- All economic indicators are sensible.
- No obvious null values

In [6]:
```
df.isna().sum()
```

Out[6]:
```
age               0
job               0
marital           0
education         0
default           0
housing           0
loan              0
contact           0
month             0
day_of_week       0
duration          0
campaign          0
pdays             0
previous          0
poutcome          0
emp.var.rate      0
cons.price.idx    0
cons.conf.idx     0
euribor3m         0
nr.employed       0
subscribed        0
dtype: int64
```

In [7]:
```
df.duration.replace(0, np.nan, inplace=True)
df.dropna(inplace=True)
```

In [8]:
```
df.loc[df.loan=='unknown'].subscribed.value_counts(normalize=True)
```

Out[8]:
```
no     0.891919
yes    0.108081
Name: subscribed, dtype: float64
```

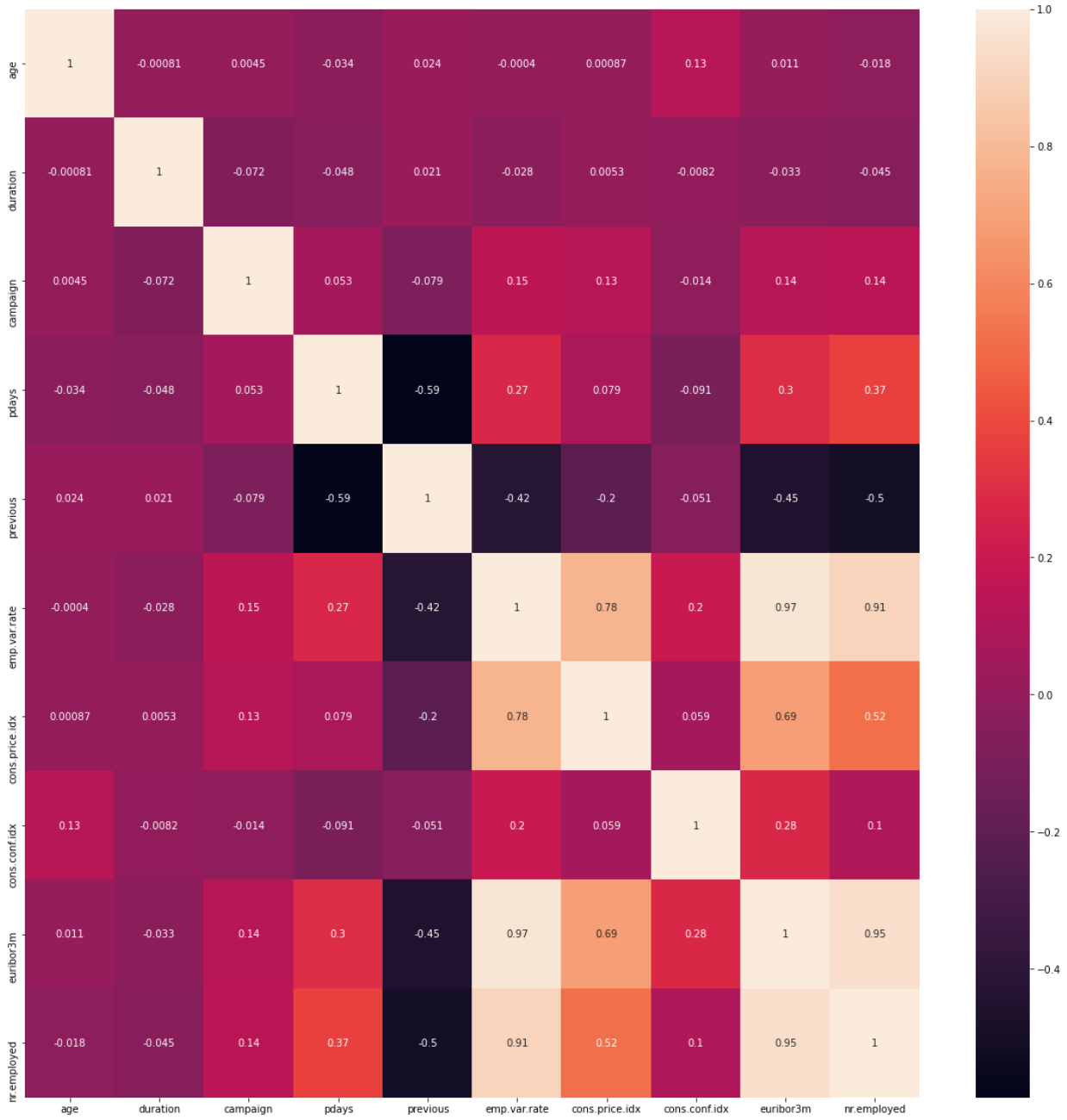
In [9]:
```
df.subscribed.value_counts(normalize=True)
```

Out[9]:
```
no     0.887335
yes    0.112665
Name: subscribed, dtype: float64
```

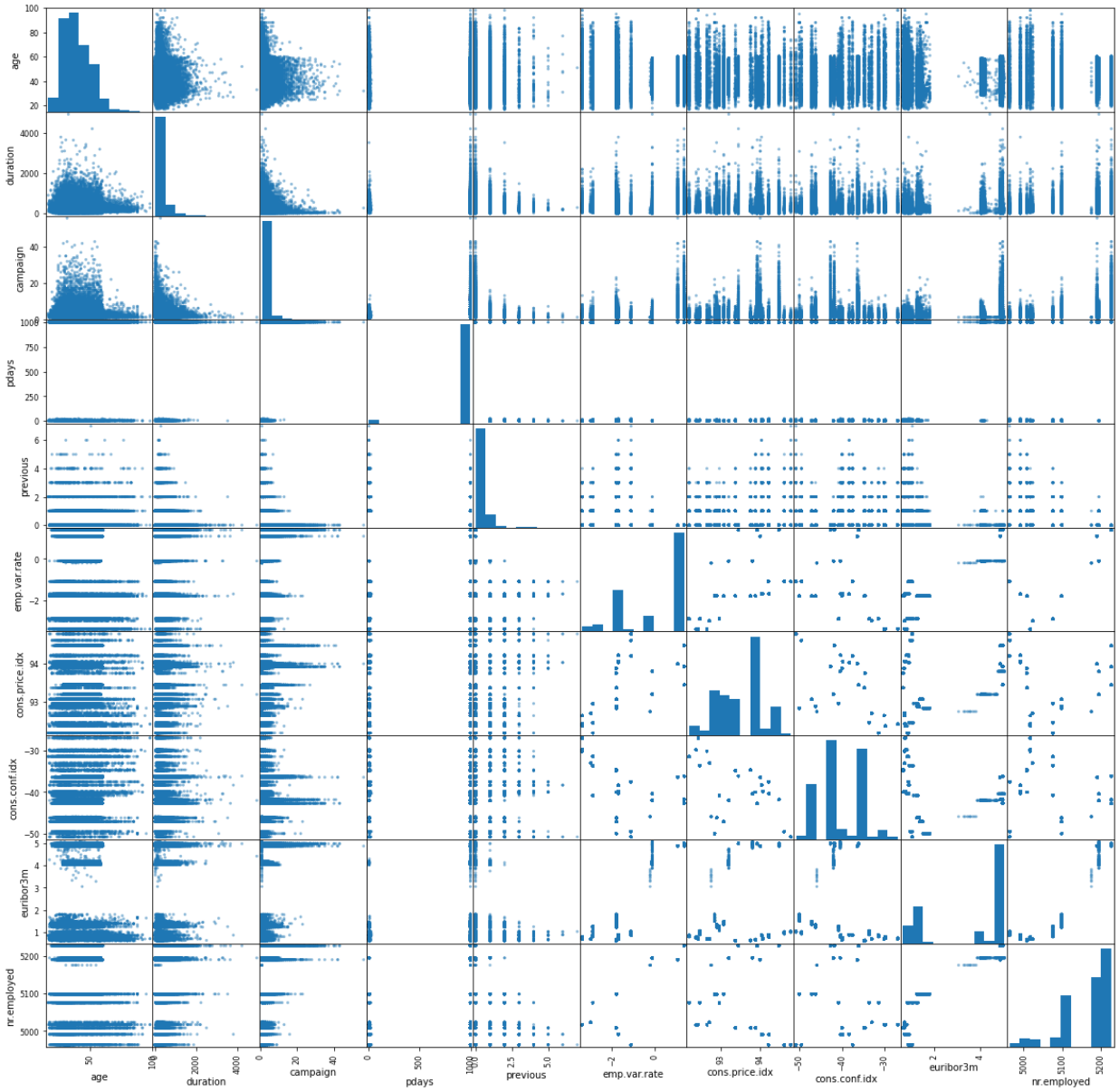
Correlations

Some of the columns have obvious correlations, such as several of the economic factors. Since I will be building a logistic regression model, I will remove columns with strong correlations. As you can see from my heatmap of the Pearson's Correlation Coefficient below, there are a few obvious candidates.

In [10]:
```
plt.figure(figsize=(20,20))
sns.heatmap(df.corr(),annot=True);
```



In [11]:
```
pd.plotting.scatter_matrix(df, figsize=(20,20));
```



Observations

High correlation between:

- emp.var.rate and euribor3m - .97
- nr.employed and emp.var.rate - .95
- euribor3m and nr.employed - .91
- cons.price.indx and emp.var.rate - .78

This makes sense because these factors should be correlated if they are similar indicators of economic health. From here I know that my continuous predictors are:

Age
Call Duration
Number of Campaign Contacts
Number of Previous Campaign Calls
National Employment Rate
Consumer Price Index
Consumer Confidence Index

And my categorical variables are:

Job
Marital Status
Highest Level of Education
If they have a housing loan
If they have a non-housing loan
Contact Method
Month of Contact
Weekday of Contact
If they have subscribed before

Data Preparation

To prepare the data for modeling, I'll first one hot encode the categorical variables and then I will split the data into 3 parts for testing.

In [12]:
```
continuous = ['age','duration','campaign','previous','emp.var.rate','cons.p
categorical = ['job','marital','education','housing','loan','contact','mont
```

In [13]:
```
ohe_df = pd.get_dummies(pd.concat([df[categorical],df[continuous],df['subsc
```

In [14]:
```
ohe_df.head()
```

Out[14]:

	age	duration	campaign	previous	emp.var.rate	cons.price.idx	cons.conf.idx	job_blue-collar	job_ent
0	56	261.0	1	0	1.1	93.994	-36.4	0	
1	57	149.0	1	0	1.1	93.994	-36.4	0	
2	37	226.0	1	0	1.1	93.994	-36.4	0	
3	40	151.0	1	0	1.1	93.994	-36.4	0	
4	56	307.0	1	0	1.1	93.994	-36.4	0	

5 rows × 49 columns

In [15]:
```
ohe_df.shape
```

Out[15]:
```
(41184, 49)
```

In [16]:
```
ohe_df.subscribed_yes.value_counts(normalize=True)
```

Out[16]:
```
0    0.887335
1    0.112665
Name: subscribed_yes, dtype: float64
```

Create Test/Train and Holdout Datasets

To create my datasets, I first will randomize the dataframe and select the first 10% of the rows to create a **Holdout Dataset**. Since I will be testing numerous models on both the main training and testing datasets, I don't want to make my selection of a model bias to one random partition of the dataset while attempting to describe the true underlying pattern. This holdout dataset will be the final dataset I'll use to determine the most appropriate model.

In [17]:
```
shuffled = ohe_df.sample(frac=1, random_state=42)
```

In [18]:
```
holdout = shuffled.iloc[:4118]
```

Train/Test Split

I'm going to then split the data with a 75/25 ratio for training and testing.

In [19]:
```
sample_df = shuffled.iloc[4118:]
```

In [20]:
```
y = sample_df['subscribed_yes']
X = sample_df.drop(columns=['subscribed_yes'])
```

In [21]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.25, ra
```

The Three Datasets

One of the big questions I'm trying to answer regards the methodology behind dealing with class imbalance. Because the underlying classes are failure/success with a rate of 88%/12%, so any model that guesses no on every result will be correct 88% of the time. With each method, I will be using the same training data, simply with either more or fewer rows. Here are the three methods I will explore:

Leave the Dataset Alone

I will keep the training dataset exactly the same with no alteration. This set will be denoted by the variables X_train and y_train, and whenever it is referenced by name I will call it the 'None' dataset.

SMOTE

I will use the Synthetic Minority Oversampling Technique, or SMOTE, to create new rows to increase the prevalence of the minority class without (hopefully) influencing the overall model quality. This is denoted by X_train_SMOTE and y_train_SMOTE, and I'll label it SMOTE.

Undersample the Majority Class

For the final method, I will instead be reducing the number of rows in the dataset from the majority class. I did this using the sample DataFrame method. The main downside of this method is that it will be reducing our training data significantly. This dataset will be denoted by X_train_sampled and y_train_sampled and it's labeled Sampled.

In [22]:
```
from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=42)
X_train_SMOTE, y_train_SMOTE = smote.fit_sample(X_train, y_train)
```

In [23]:
```
train = pd.concat([X_train,y_train], axis=1)
train.subscribed_yes.value_counts()
```

Out[23]:
```
0    24630
1     3169
Name: subscribed_yes, dtype: int64
```

In [24]:
```
non_subscribe = train.loc[train.subscribed_yes==0]
subscribe = train.loc[train.subscribed_yes==1]
```

In [25]:
```
sampled_non_subscribe = non_subscribe.sample(3169, replace=False, random_st
```

In [26]:
```
sampled_df = pd.concat([subscribe, sampled_non_subscribe], axis=0)
```

In [27]:
```
X_train_sampled = sampled_df.drop(columns=['subscribed_yes'])
y_train_sampled = sampled_df['subscribed_yes']
```

In [28]:
```
y_train_sampled.value_counts()
```

Out[28]:
```
1    3169
0    3169
Name: subscribed_yes, dtype: int64
```

From here on, I will grade each model separately to see what transformation performs better on the test data.

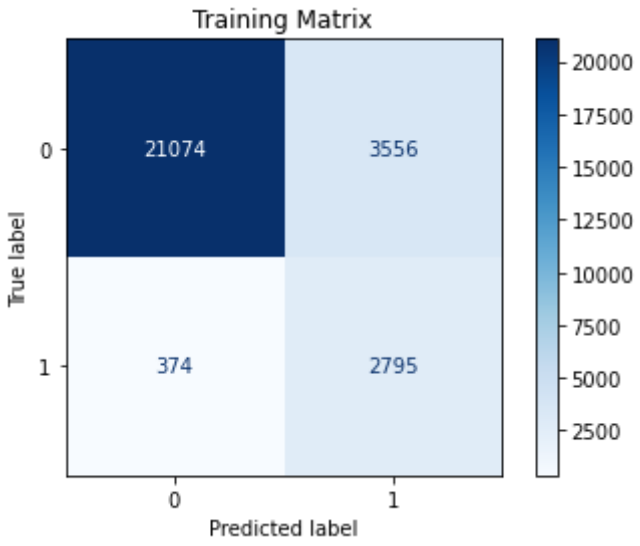
Model 1a: Logistic Regression - No Transformations

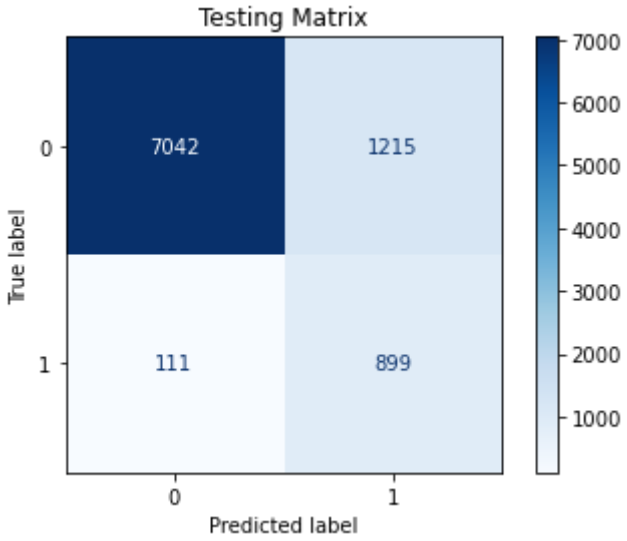
We'll begin with a basic Logistic Regression model. Before I move forward, I will create a dictionary for use in my Model_Analysis object, with each X_train, y_train dataset present and labeled. I will be testing the models based on the same X_test, y_test dataset.

In [29]:
```
data_dict = {
    'None' : {'x': X_train, 'y': y_train},
    'SMOTE': {'x': X_train_SMOTE, 'y': y_train_SMOTE},
    'Sampled': {'x': X_train_sampled, 'y': y_train_sampled}
}
```

In [30]:
```
logreg = LogisticRegression(fit_intercept=False, solver='liblinear', class_
```

In [31]:
```
scores(logreg, X_train, X_test, y_train, y_test)
```

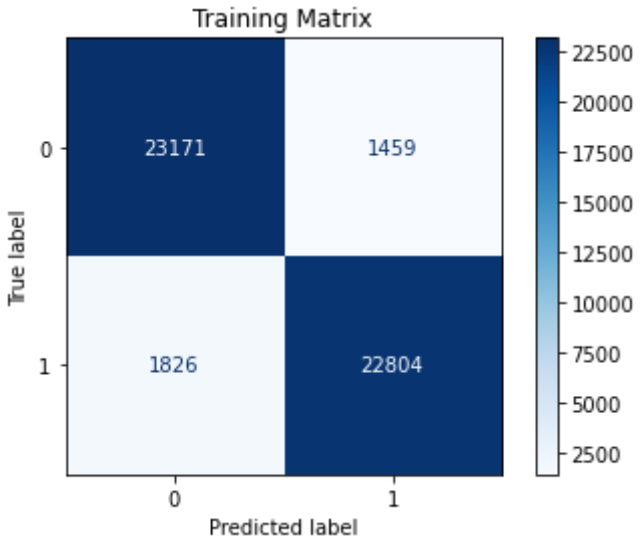


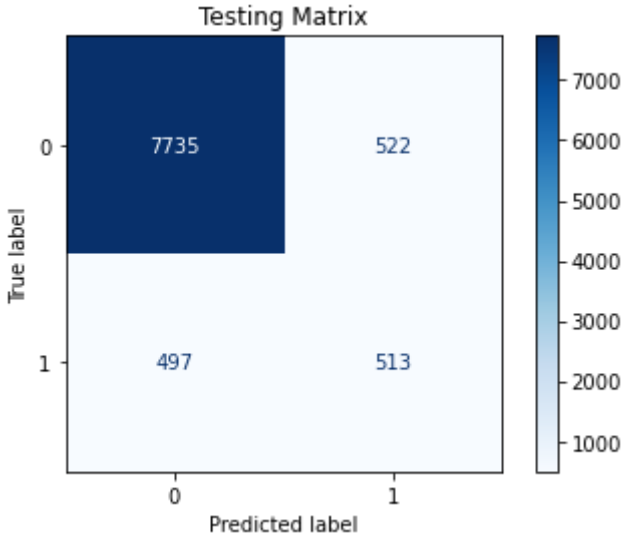


Out[31]:

	Train	Test
Accuracy	0.858628	0.856912
Precision	0.440088	0.425260
Recall	0.881982	0.890099
F1 Score	0.587185	0.575544
ROC-AUC Score	0.868802	0.871476

In [32]:
```
scores(logreg, X_train_SMOTE, X_test, y_train_SMOTE, y_test)
```

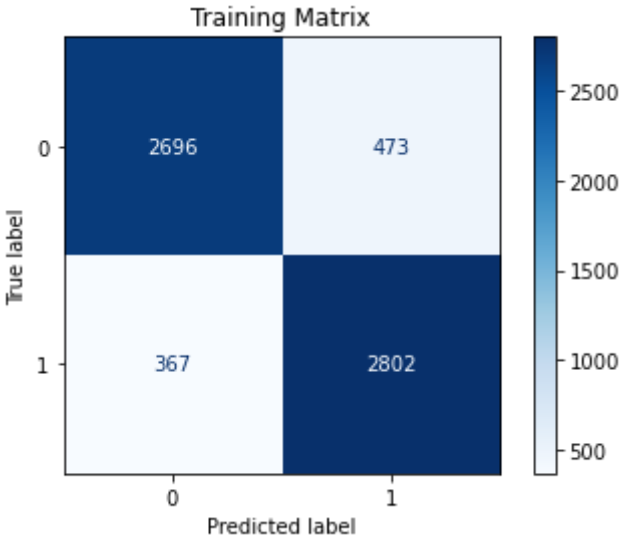


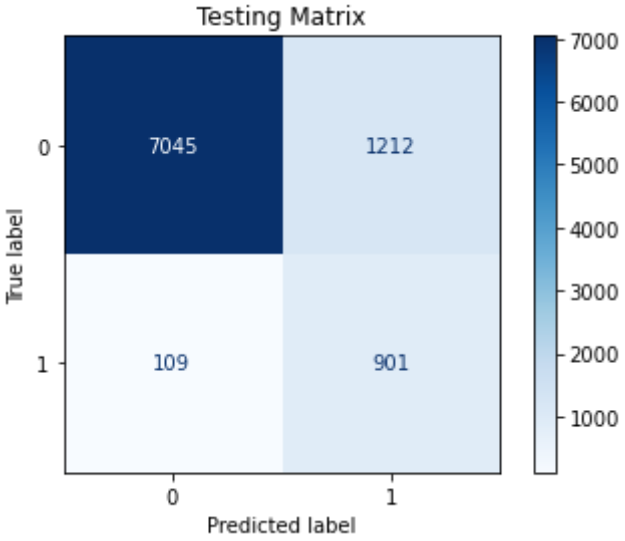


Out[32]:

	Train	Test
Accuracy	0.933313	0.890040
Precision	0.939867	0.495652
Recall	0.925863	0.507921
F1 Score	0.932812	0.501711
ROC-AUC Score	0.933313	0.722351

In [33]:
```
scores(logreg, X_train_sampled, X_test, y_train_sampled, y_test)
```





Out[33]:

	Train	Test
Accuracy	0.867466	0.857451

	Train	Test
Precision	0.855573	0.426408
Recall	0.884191	0.892079
F1 Score	0.869646	0.577009
ROC-AUC Score	0.867466	0.872647

In [34]:
```
log_reg_analysis = Model_Analysis(logreg, data_dict, X_test, y_test)
```

In [35]:
```
log_reg_analysis.calc_scores()
```

In [36]:
```
log_reg_analysis.score_table
```

Out[36]:

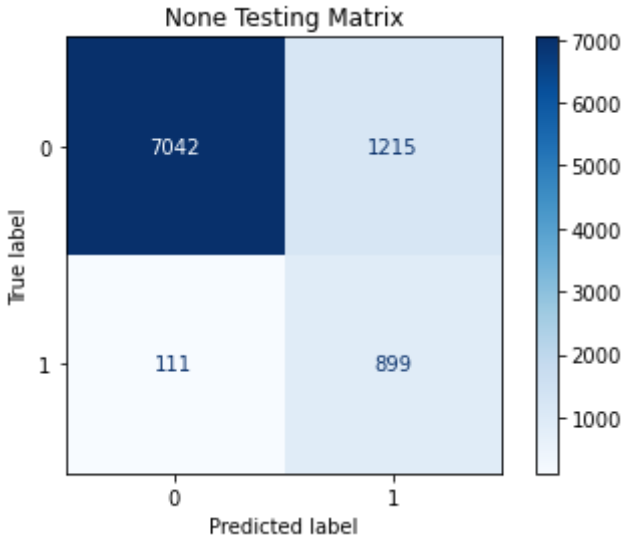
	None Train	None Test	SMOTE Train	SMOTE Test	Sampled Train	Sampled Test
Accuracy	0.858628	0.856912	0.933313	0.890040	0.867466	0.857451
Precision	0.440088	0.425260	0.939867	0.495652	0.855573	0.426408
Recall	0.881982	0.890099	0.925863	0.507921	0.884191	0.892079
F1 Score	0.587185	0.575544	0.932812	0.501711	0.869646	0.577009
ROC-AUC Score	0.868802	0.871476	0.933313	0.722351	0.867466	0.872647

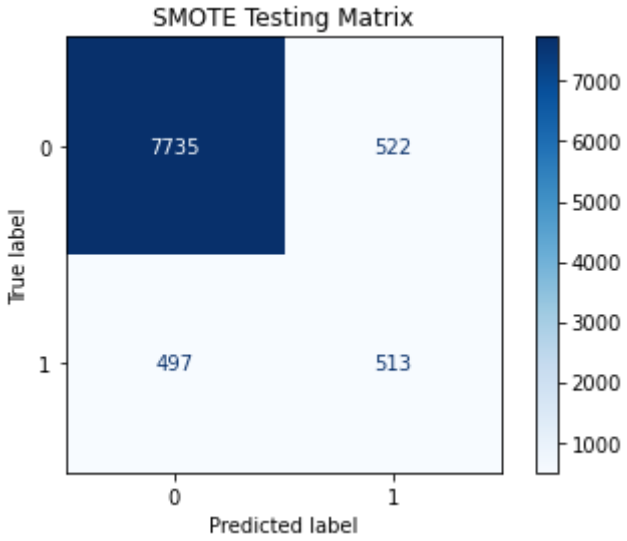
In [37]:
```
log_reg_analysis.test_scores
```

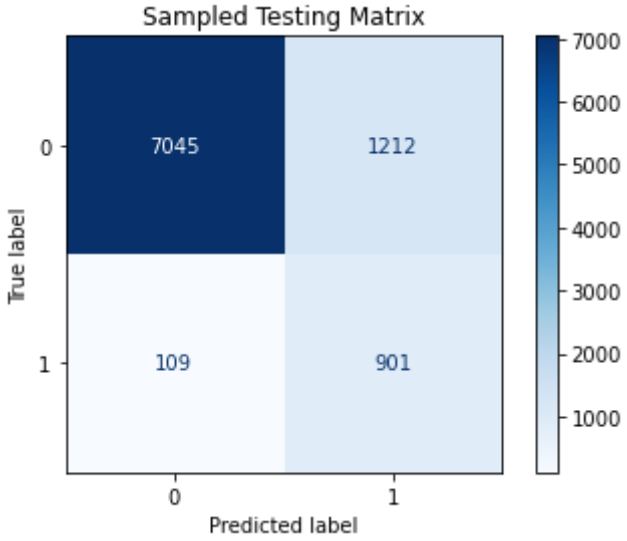
Out[37]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.856912	0.890040	0.857451
Precision	0.425260	0.495652	0.426408
Recall	0.890099	0.507921	0.892079
F1 Score	0.575544	0.501711	0.577009
ROC-AUC Score	0.871476	0.722351	0.872647

In [38]:
```
log_reg_analysis.confusion_compare()
```





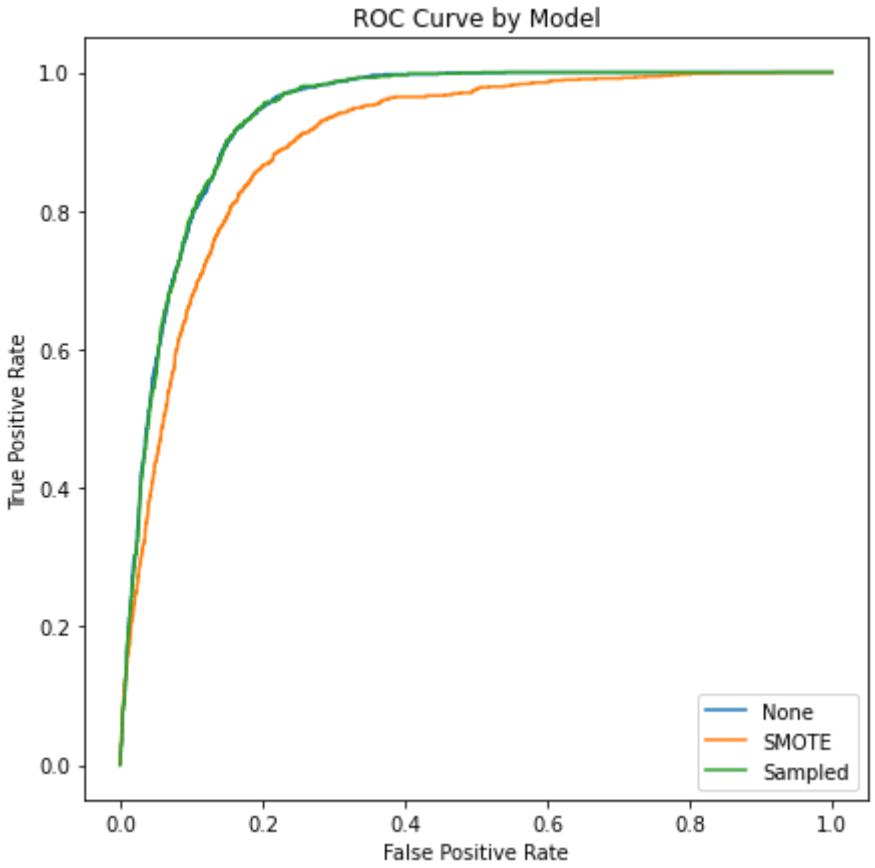


Out[38]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.856912	0.890040	0.857451

	None Test	SMOTE Test	Sampled Test
Precision	0.425260	0.495652	0.426408
Recall	0.890099	0.507921	0.892079
F1 Score	0.575544	0.501711	0.577009
ROC-AUC Score	0.871476	0.722351	0.872647

In [39]:
```
log_reg_analysis.plot_roc()
```



Observations

It's pretty clear that this model doesn't do much better than a blind guess based on the testing data. While SMOTE has the best accuracy, it is clearly overfit. It's interesting to note that the scores for the untampered data and the sampled datasets are almost exactly the same. Let's do some tuning to see what we can improve.

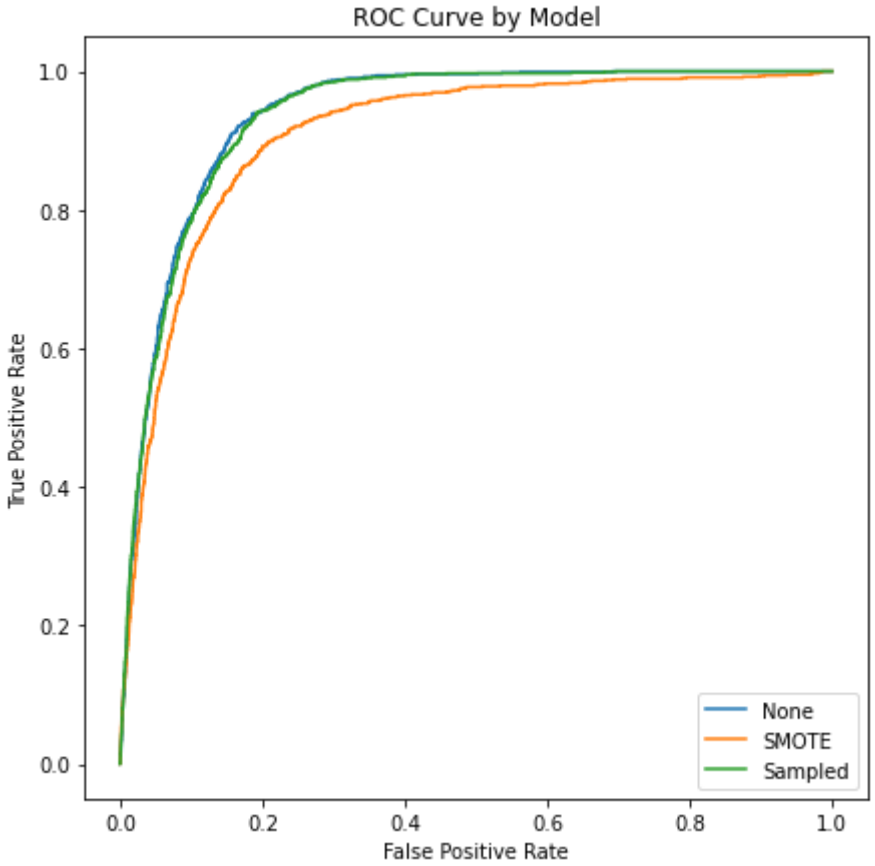
Model 1b: Logistic Regression with Tuning and Transformations

All of these steps will be taken using all three data set: 1) no class tuning, 2) majority sampling, 3) SMOTE.

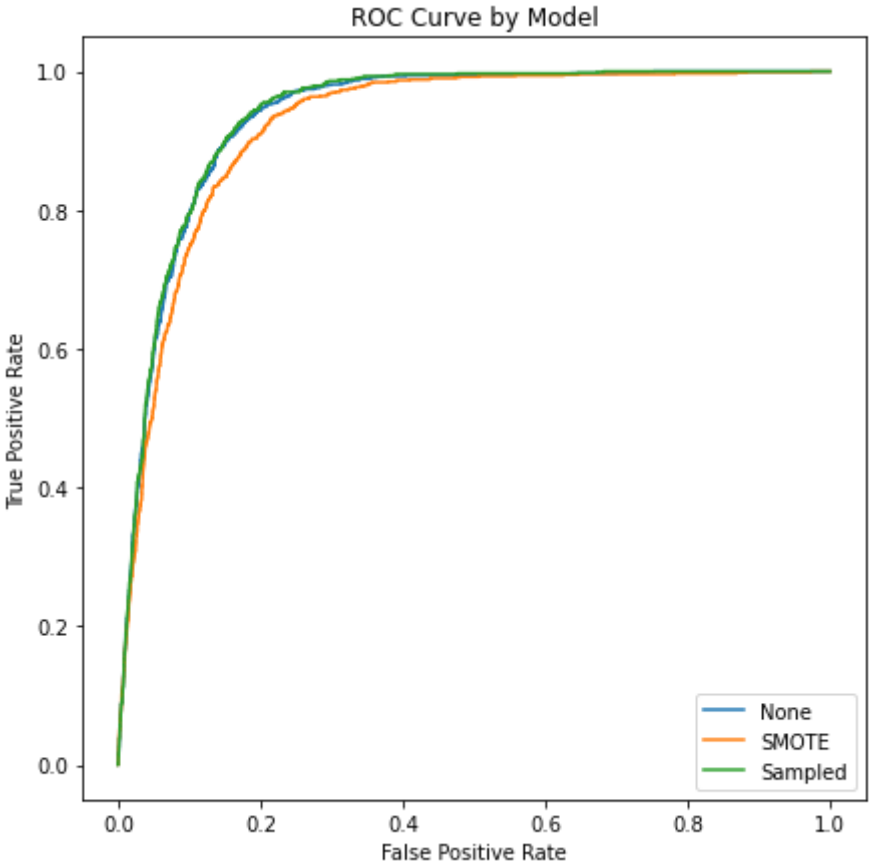
- Data Scaling - MinMaxScaler, StandardScaler, RobustScaler
- L1, L2 penalties
- Tune C

First I will create pipelines with different scalers to see which perform the best on the datasets.

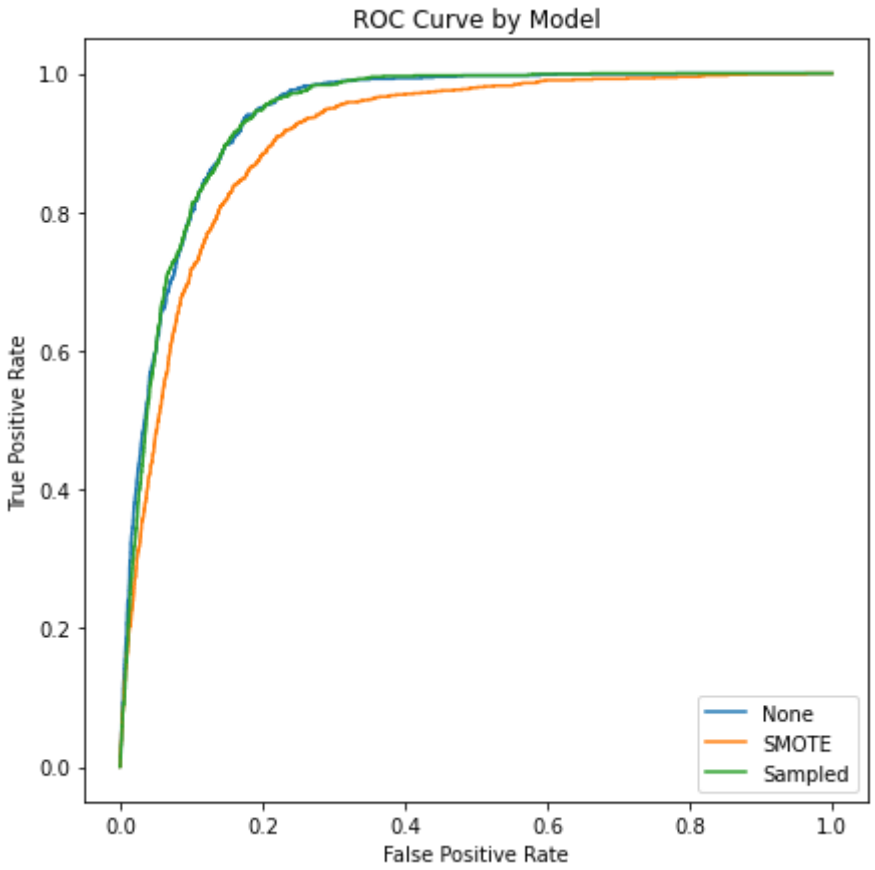
In [40]:
```
from sklearn.pipeline import Pipeline

min_max = MinMaxScaler()
robust = RobustScaler()
standard = StandardScaler()

pipe1 = Pipeline([('scale', min_max), ('logreg', logreg)])
pipe_log_1 = Model_Analysis(pipe1, data_dict, X_test, y_test)
pipe_log_1.calc_scores()
pipe_log_1.plot_roc()
```



In [41]:
```
pipe2 = Pipeline([('scale', robust), ('logreg', logreg)])
pipe_log_2 = Model_Analysis(pipe2, data_dict, X_test, y_test)
pipe_log_2.calc_scores()
pipe_log_2.plot_roc()
```



In [42]:
```
pipe3 = Pipeline([('scale', standard), ('logreg', logreg)])
pipe_log_3 = Model_Analysis(pipe3, data_dict, X_test, y_test)
pipe_log_3.calc_scores()
pipe_log_3.plot_roc()
```



In [43]:
```
pipe_log_1.test_scores
```

Out[43]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.855293	0.900183	0.856048
Precision	0.422627	0.540515	0.422562
Recall	0.895050	0.561386	0.875248
F1 Score	0.574151	0.550753	0.569955
ROC-AUC Score	0.872740	0.751506	0.864474

In [44]:
```
pipe_log_2.test_scores
```

Out[44]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.853566	0.897917	0.855293
Precision	0.419787	0.526490	0.422916
Recall	0.899010	0.629703	0.899010
F1 Score	0.572329	0.573490	0.575230
ROC-AUC Score	0.873509	0.780214	0.874478

In [45]:
```
pipe_log_3.test_scores
```

Out[45]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.757527	0.897594	0.863926
Precision	0.308098	0.531937	0.437469
Recall	0.983168	0.502970	0.869307
F1 Score	0.469171	0.517048	0.582035
ROC-AUC Score	0.856547	0.724417	0.866287

Observations

The Min-Max Scaler seems to have the best performance on the data, with a max accuracy score of 90% on the SMOTE dataset, so we will keep that moving forward. Now, I will change the solver to 'lbjgs' and see if we have more luck.

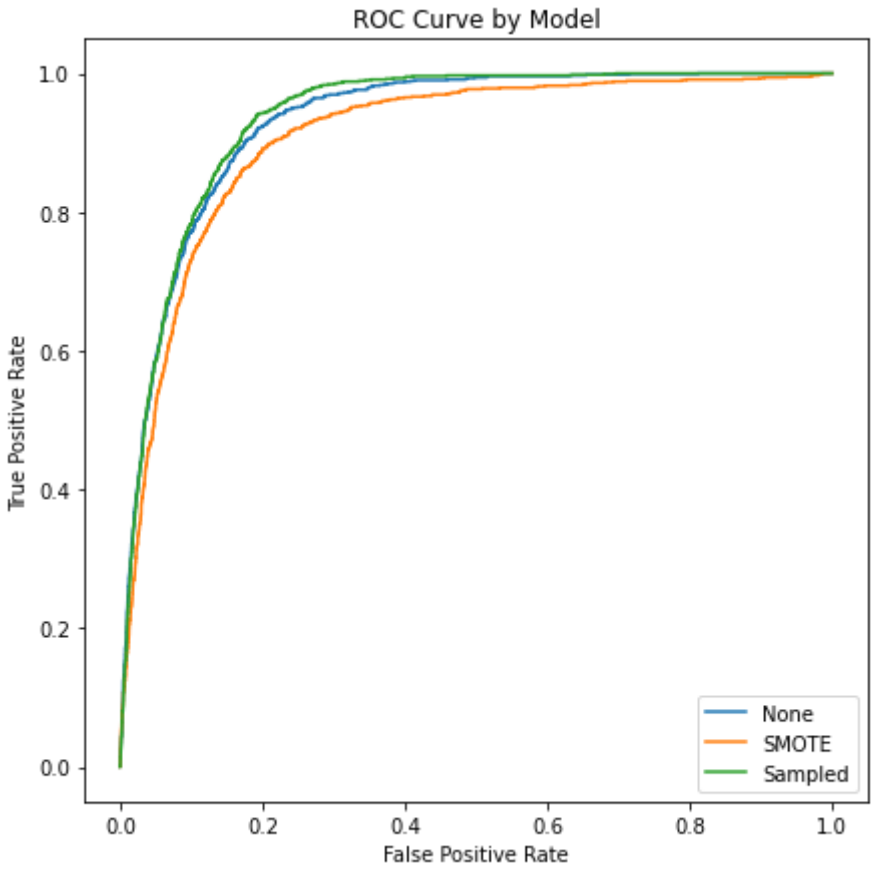
In [46]:
```
logreg_new = LogisticRegression(fit_intercept=False, solver='lbfgs', random
pipe4 = Pipeline([('ss', MinMaxScaler()), ('logreg', logreg_new)])
```

In [47]:
```
pipe_log_4 = Model_Analysis(pipe4, data_dict, X_test, y_test)
pipe_log_4.calc_scores()
pipe_log_4.test_scores
```

Out[47]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.913241	0.900183	0.856048
Precision	0.662975	0.540515	0.422562
Recall	0.414851	0.561386	0.875248
F1 Score	0.510353	0.550753	0.569955
ROC-AUC Score	0.694528	0.751506	0.864474

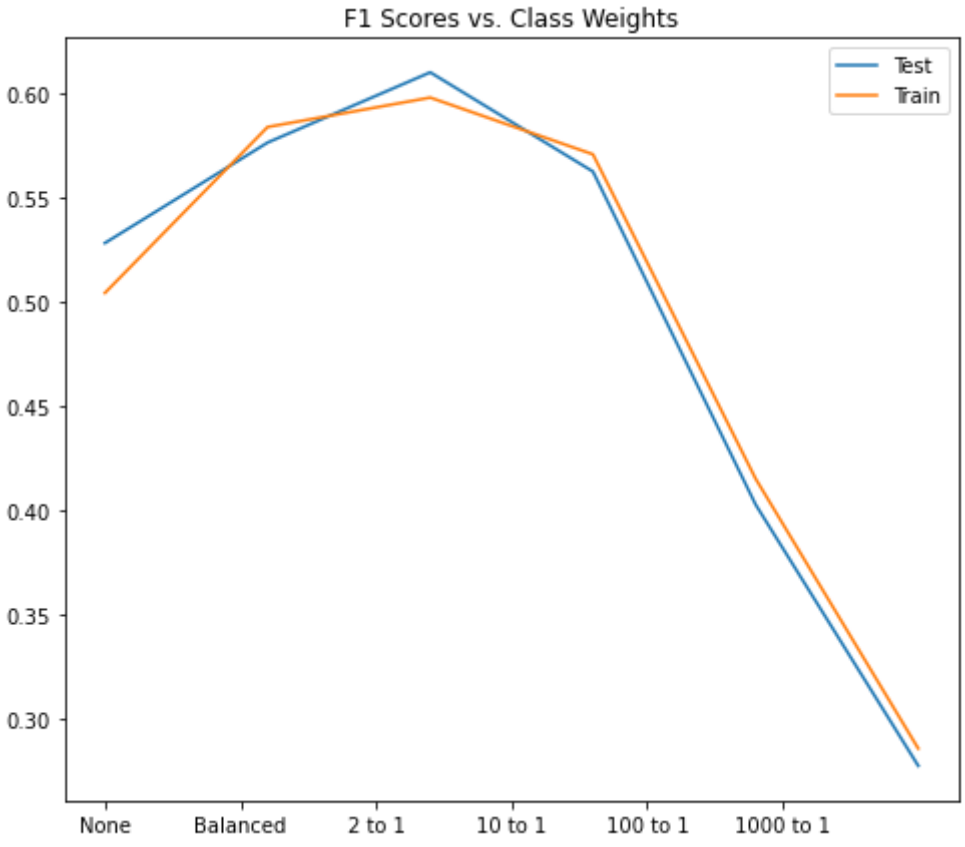
In [48]:
```
pipe_log_4.plot_roc()
```



Weight Tuning

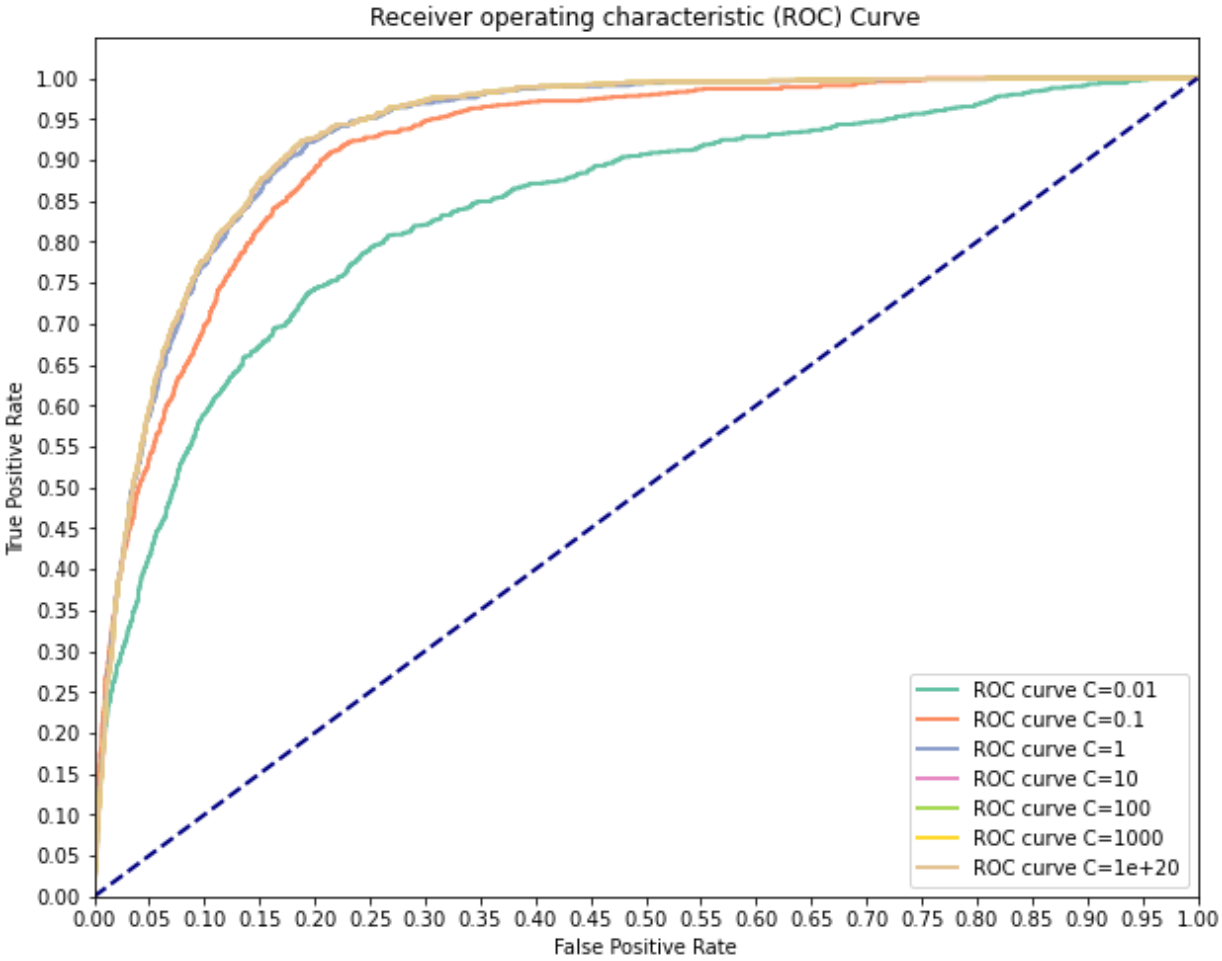
Next, we'll examine the affect of class weight on the None dataset to see what effect it has on f1_score (our main comparison metric).

In [49]:
```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")


    weights = [None, 'balanced', {1:2, 0:1}, {1:10, 0:1}, {1:100, 0:1}, {1:
    names = ['None', 'Balanced', '2 to 1', '10 to 1', '100 to 1', '1000 to
    colors = sns.color_palette('Set2')

    f1_scores_test = []
    f1_scores_train = []

    plt.figure(figsize=(8,7))

    for n, weight in enumerate(weights):
        pipe = Pipeline([('ss', MinMaxScaler()),('logreg', LogisticRegressi
        model_log = pipe.fit(X_train, y_train)

        y_hat_train = pipe.predict(X_train)
        y_hat_test = pipe.predict(X_test)

        f1_scores_train.append(f1_score(y_train, y_hat_train))
        f1_scores_test.append(f1_score(y_test, y_hat_test))

    plt.plot(np.linspace(0, 6, num=6), f1_scores_test, label='Test')
    plt.plot(np.linspace(0, 6, num=6), f1_scores_train, label='Train')
    plt.title('F1 Scores vs. Class Weights')
    plt.xticks(np.arange(len(weights)), labels=names)
    plt.legend()
    plt.show()
```

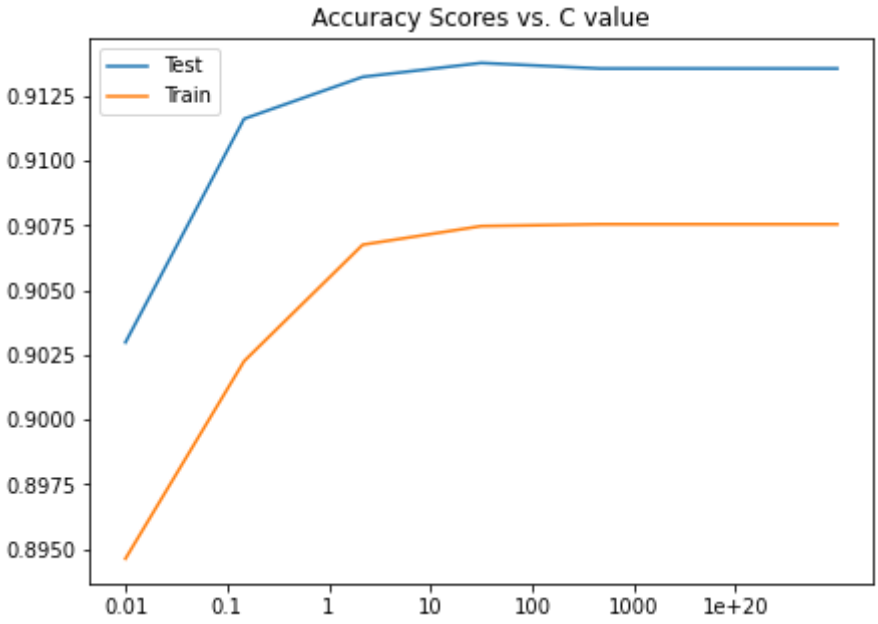


It seems like between 2:1 and 10:1 has the best f1_score.

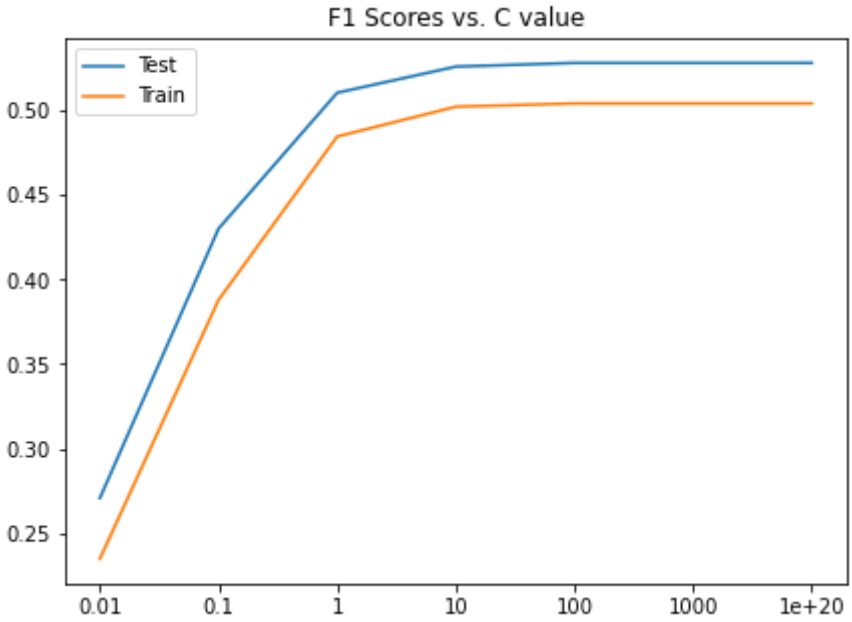
In [50]:
```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")

    C = [0.01, 0.1, 1, 10, 100, 1000, 1e20]
    colors = sns.color_palette('Set2')

    plt.figure(figsize=(10,8))

    for n, c in enumerate(C):
        pipe = Pipeline([('ss', MinMaxScaler()),('logreg', LogisticRegressi
        model_log = pipe.fit(X_train, y_train)

        y_hat_test = pipe.predict(X_test)

        y_score = pipe.fit(X_train, y_train).decision_function(X_test)

        fpr, tpr, thresholds = roc_curve(y_test, y_score)

        plt.plot(fpr, tpr, color=colors[n],
                 lw=2, label=f'ROC curve C={c}')

    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])

    plt.yticks([i/20.0 for i in range(21)])
    plt.xticks([i/20.0 for i in range(21)])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic (ROC) Curve')
    plt.legend(loc='lower right')
    plt.show()
```


Receiver operating characteristic (ROC) Curve
True Positive Rate
False Positive Rate
0.00 0.05 0.10 0.15 0.20 0.25 0.30 0.35 0.40 0.45 0.50 0.55 0.60 0.65 0.70 0.75 0.80 0.85 0.90 0.95 1.00
ROC curve C=0.01
ROC curve C=0.1
ROC curve C=1
ROC curve C=10
ROC curve C=100
ROC curve C=1000
ROC curve C=1e+20

In [51]:
```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")

    C = [0.01, 0.1, 1, 10, 100, 1000, 1e20]
    colors = sns.color_palette('Set2')

    plt.figure(figsize=(7,5))

    acc_score_test = []
    acc_score_train = []

    for n, c in enumerate(C):
        pipe = Pipeline([('ss', MinMaxScaler()),('logreg', LogisticRegressi
        model_log = pipe.fit(X_train, y_train)

        y_hat_train = pipe.predict(X_train)
        y_hat_test = pipe.predict(X_test)

        acc_score_train.append(accuracy_score(y_train, y_hat_train))
        acc_score_test.append(accuracy_score(y_test, y_hat_test))

    plt.plot(np.linspace(0, 7, num=7), acc_score_test, label='Test')
    plt.plot(np.linspace(0, 7, num=7), acc_score_train, label='Train')
    plt.title('Accuracy Scores vs. C value')
    plt.xticks(np.arange(len(C)), labels=C)
    plt.legend()
    plt.show()
```



In [52]:
```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")

    C = [0.01, 0.1, 1, 10, 100, 1000, 1e20]
    colors = sns.color_palette('Set2')

    plt.figure(figsize=(7,5))

    f1_scores_test = []
    f1_scores_train = []

    for n, c in enumerate(C):
        pipe = Pipeline([('ss', MinMaxScaler()),('logreg', LogisticRegressi
        model_log = pipe.fit(X_train, y_train)

        y_hat_train = pipe.predict(X_train)
        y_hat_test = pipe.predict(X_test)

        f1_scores_train.append(f1_score(y_train, y_hat_train))
        f1_scores_test.append(f1_score(y_test, y_hat_test))

    plt.plot(range(0,len(C)), f1_scores_test, label='Test')
    plt.plot(range(0,len(C)), f1_scores_train, label='Train')
    plt.title('F1 Scores vs. C value')
    plt.xticks(np.arange(len(C)), labels=C)
    plt.legend()
    plt.show()
```



From both the accuracy and the F1 score we can see the larger the C, the better our model does. Using this info, I'm going to create a GridSearch object that will explore various combinations of parameters. As with all GridSearches in this project, I will be using F1 as the main comparison metric.

In [53]:
```
pipe = Pipeline([('ss', MinMaxScaler()), ('logreg', LogisticRegression(rand

params = [{
    'logreg__fit_intercept': [True, False],
    'logreg__C' : [0.1, 1, 100, 1e20],
    'logreg__solver' : ['lbfgs', 'liblinear'],
    'logreg__penalty' : ['none','l1','l2','elasticnet'],
    'logreg__class_weight' : [None, 'balanced', {1:2, 0:1}, {1:10, 0:1}]
}]

grid = GridSearchCV(pipe, param_grid=params, scoring='f1', cv=5)
```

In [54]:
```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    grid_log = Model_Analysis(grid, data_dict, X_test, y_test)
    grid_log.calc_scores()
```

In [55]:
```
grid_log.test_scores
```

Out[55]:

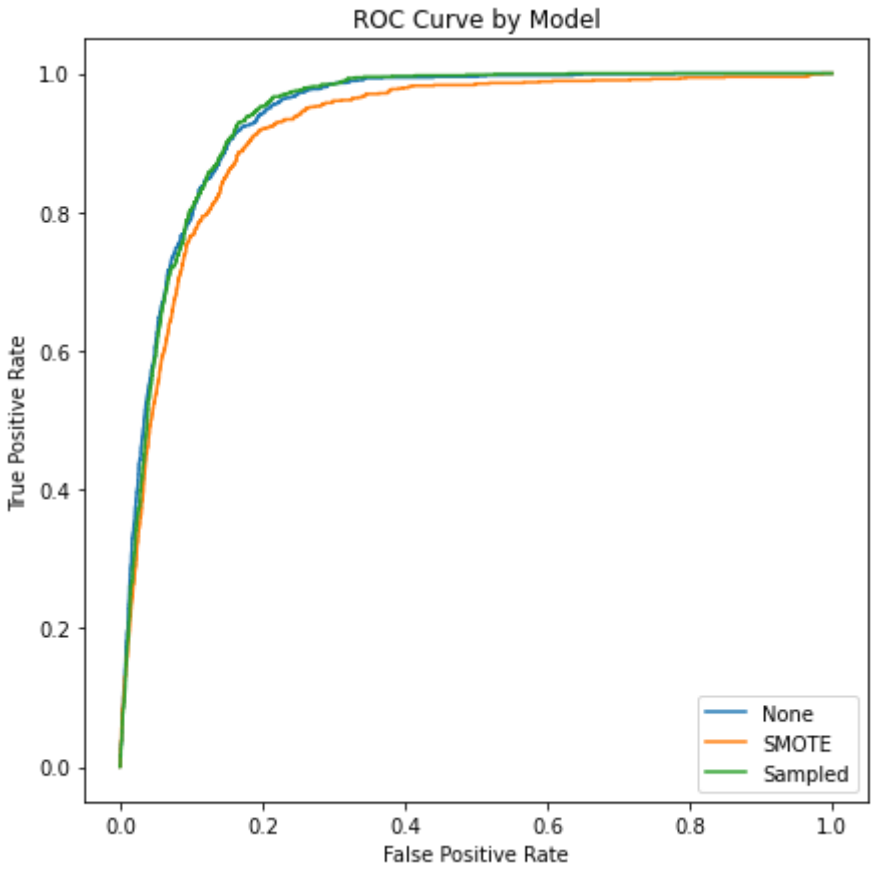
	None Test	SMOTE Test	Sampled Test
Accuracy	0.913564	0.890148	0.818712
Precision	0.595434	0.497365	0.370856
Recall	0.645545	0.747525	0.952475
F1 Score	0.619477	0.597310	0.533851
ROC-AUC Score	0.795947	0.827559	0.877412

In [56]:
```
grid_log.score_table
```

Out[56]:

	None Train	None Test	SMOTE Train	SMOTE Test	Sampled Train	Sampled Test
Accuracy	0.907982	0.913564	0.929781	0.890148	0.875039	0.818712
Precision	0.593454	0.595434	0.911167	0.497365	0.828587	0.370856
Recall	0.612180	0.645545	0.952416	0.747525	0.945724	0.952475
F1 Score	0.602672	0.619477	0.931335	0.597310	0.883289	0.533851
ROC-AUC Score	0.779111	0.795947	0.929781	0.827559	0.875039	0.877412

In [57]:
```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")

    grid_log.plot_roc()
```



In [58]:
```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")

    grid.fit(X_train, y_train)
```

In [59]:
```
log_reg_best = grid.best_estimator_ #NONE dataset
```

In [60]:
```
grid.best_params_
```

Out[60]:
```
{'logreg__C': 100,
 'logreg__class_weight': {1: 2, 0: 1},
 'logreg__fit_intercept': True,
 'logreg__penalty': 'l1',
 'logreg__solver': 'liblinear'}
```

Observations:

For the logistic regression, the best we can do on the testing score is 91.4%. This was achieved on the untampered data (no SMOTE or sampling), and with the following parameters.

```
{'logreg__C': 100,
 'logreg__class_weight': {1: 2, 0: 1},
 'logreg__fit_intercept': True,
 'logreg__penalty': 'l1',
 'logreg__solver': 'liblinear'}
```

Model 2: K Nearest Neighbors

This dataset is not ideal for KNN simply because of how large it is and how many columns. At (41000, 51) the BigO is simply too large to be feasible. I did build a GridSearch for KNN and I have left my code in the cells below commented out. I do not recommend running it because it will take exceptionally long and the results are underwhelming.

In [61]:
```
# pipe_knn = Pipeline([('ss', MinMaxScaler()), ('knn', KNeighborsClassifier

# knn_analysis = Model_Analysis(pipe_knn, data_dict, X_test, y_test)
# knn_analysis.calc_scores()
```

In [62]:
```
# knn_analysis.score_table
```

Out[62]:

	None Train	None Test	SMOTE Train	SMOTE Test	Sampled Train	Sampled Test
Accuracy	0.913162	0.893709	0.941657	0.868782	0.813979	0.782238
Precision	0.771387	0.531486	0.952045	0.387555	0.841221	0.285166
Recall	0.338593	0.208911	0.930166	0.351485	0.774061	0.662376
F1 Score	0.470614	0.299929	0.940978	0.368640	0.806245	0.398689
ROC-AUC Score	0.662841	0.593192	0.941657	0.641771	0.813979	0.729638

In [63]:
```
# knn_analysis.test_scores
```

Out[63]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.893709	0.868782	0.782238
Precision	0.531486	0.387555	0.285166
Recall	0.208911	0.351485	0.662376
F1 Score	0.299929	0.368640	0.398689
ROC-AUC Score	0.593192	0.641771	0.729638

In [64]:
```
# grid = [{
#     'knn__n_neighbors': [3, 4, 5],
# }]

# search = GridSearchCV(pipe_knn, param_grid=grid, scoring='f1', cv=5)

# knn_analysis2 = Model_Analysis(search, data_dict, X_test, y_test)
# knn_analysis2.calc_scores()
```

In [65]:
```
# knn_analysis2.score_table
```

Out[65]:

	None Train	None Test	SMOTE Train	SMOTE Test	Sampled Train	Sampled Test
Accuracy	0.927012	0.891335	0.941657	0.868782	0.813979	0.782238
Precision	0.801268	0.502935	0.952045	0.387555	0.841221	0.285166
Recall	0.478384	0.254455	0.930166	0.351485	0.774061	0.662376
F1 Score	0.599091	0.337936	0.940978	0.368640	0.806245	0.398689
ROC-AUC Score	0.731559	0.611847	0.941657	0.641771	0.813979	0.729638

In [66]:
```
# knn_analysis2.test_scores
```

Out[66]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.891335	0.868782	0.782238
Precision	0.502935	0.387555	0.285166
Recall	0.254455	0.351485	0.662376
F1 Score	0.337936	0.368640	0.398689
ROC-AUC Score	0.611847	0.641771	0.729638

In [67]:
```
# knn_analysis2.model.best_params_
```

Out[67]:
```
{'knn__n_neighbors': 5}
```

In [68]:
```
# grid2 = [{
#     'knn__p': [1, 2, 3],
# }]
# pipe_knn2 = Pipeline([('ss', MinMaxScaler()), ('knn', KNeighborsClassifie

# search2 = GridSearchCV(pipe_knn2, param_grid=grid2, scoring='f1', cv=5)

# knn_analysis3 = Model_Analysis(search2, data_dict, X_test, y_test)
# knn_analysis3.calc_scores()
```

In [69]:
```
# knn_analysis3.score_table
```

Out[69]:

	None Train	None Test	SMOTE Train	SMOTE Test	Sampled Train	Sampled Test
Accuracy	0.913702	0.893601	0.943402	0.873206	0.817135	0.792705
Precision	0.771127	0.528436	0.954058	0.408435	0.846313	0.301179
Recall	0.345535	0.220792	0.931669	0.364356	0.775008	0.683168
F1 Score	0.477228	0.311453	0.942730	0.385139	0.809092	0.418055
ROC-AUC Score	0.666170	0.598346	0.943402	0.649903	0.817135	0.744636

In [70]:
```
# knn_analysis3.test_scores
```

Out[70]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.893601	0.873206	0.792705
Precision	0.528436	0.408435	0.301179
Recall	0.220792	0.364356	0.683168
F1 Score	0.311453	0.385139	0.418055
ROC-AUC Score	0.598346	0.649903	0.744636

In [71]:
```
# knn_analysis3.model.best_params_
```

Out[71]:
```
{'knn__p': 1}
```

In [72]:
```
# grid3 = [{
#     'knn__weights': ['uniform','distance'],
# }]
# pipe_knn3 = Pipeline([('ss', MinMaxScaler()), ('knn', KNeighborsClassifie

# search3 = GridSearchCV(pipe_knn3, param_grid=grid3, scoring='f1', cv=5)

# knn_analysis4 = Model_Analysis(search3, data_dict, X_test, y_test)
# knn_analysis4.calc_scores()
```

In [73]:
```
# knn_analysis4.score_table
```

Out[73]:

	None Train	None Test	SMOTE Train	SMOTE Test	Sampled Train	Sampled Test
Accuracy	1.0	0.890687	0.943402	0.873206	0.817135	0.792705
Precision	1.0	0.497018	0.954058	0.408435	0.846313	0.301179
Recall	1.0	0.247525	0.931669	0.364356	0.775008	0.683168
F1 Score	1.0	0.330469	0.942730	0.385139	0.809092	0.418055
ROC-AUC Score	1.0	0.608442	0.943402	0.649903	0.817135	0.744636

In [74]:
```
# knn_analysis4.test_scores
```

Out[74]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.890687	0.873206	0.792705
Precision	0.497018	0.408435	0.301179
Recall	0.247525	0.364356	0.683168
F1 Score	0.330469	0.385139	0.418055
ROC-AUC Score	0.608442	0.649903	0.744636

In [75]:
```
# knn_analysis4.model.best_params_
```

Out[75]:
```
{'knn__weights': 'uniform'}
```

In [76]:
```
# grid = [{
#     'knn__n_neighbors': [3, 4, 5],
#     'knn__p': [1, 2, 3],
#     'knn__weights': ['uniform','distance']
# }]

# pipe_knn = Pipeline([('ss', MinMaxScaler()), ('knn', KNeighborsClassifier

# search = GridSearchCV(pipe_knn, param_grid=grid, scoring='f1', cv=5)

# knn_analysis_final = Model_Analysis(search, data_dict, X_test, y_test)
# knn_analysis_final.calc_scores()
```

In [77]:
```
# knn_analysis_final.score_table
```

Out[77]:

	None Train	None Test	SMOTE Train	SMOTE Test	Sampled Train	Sampled Test
Accuracy	1.0	0.885939	0.943402	0.873206	0.817135	0.792705
Precision	1.0	0.462036	0.954058	0.408435	0.846313	0.301179
Recall	1.0	0.283168	0.931669	0.364356	0.775008	0.683168
F1 Score	1.0	0.351136	0.942730	0.385139	0.809092	0.418055
ROC-AUC Score	1.0	0.621419	0.943402	0.649903	0.817135	0.744636

In [78]:
```
# knn_analysis_final.test_scores
```

Out[78]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.885939	0.873206	0.792705
Precision	0.462036	0.408435	0.301179
Recall	0.283168	0.364356	0.683168
F1 Score	0.351136	0.385139	0.418055
ROC-AUC Score	0.621419	0.649903	0.744636

In [79]:
```
# knn_analysis_final.model.best_params_
```

Out[79]:
```
{'knn__n_neighbors': 5, 'knn__p': 1, 'knn__weights': 'uniform'}
```

Observatons

Overall the None dataset had the best blend of scores with the parameters:

- n_neighbors: 5
- p : 1
- weights: uniform

However, overall the scores are actually slightly worse than a of 'no', so we will not include this model in the final comparison.

Model 3: Gaussian Naive Bayes

I am not exceptionally hopeful that a Gaussian Naive Bayes would be an effective method to model a class imbalance problem like this, though I decided to explore it for practice purposes. GNB models are better suited for similarity comparisons, but let's see how it does.

In [55]:
```
from sklearn.naive_bayes import GaussianNB
```

In [80]:
```
from sklearn.naive_bayes import GaussianNB

pipe_bayes = Pipeline([('scaler', MinMaxScaler()), ('bayes', GaussianNB())]

bayes_analysis = Model_Analysis(pipe_bayes, data_dict, X_test, y_test)
bayes_analysis.calc_scores()
```

In [81]:
```
bayes_analysis.score_table
```

Out[81]:

	None Train	None Test	SMOTE Train	SMOTE Test	Sampled Train	Sampled Test
Accuracy	0.871794	0.879249	0.863886	0.786986	0.673241	0.883350
Precision	0.442936	0.451469	0.835059	0.259241	0.864058	0.463136
Recall	0.483749	0.501980	0.906902	0.513861	0.411171	0.441584
F1 Score	0.462443	0.475387	0.869499	0.344622	0.557195	0.452103
ROC-AUC Score	0.702735	0.713688	0.863886	0.667128	0.673241	0.689485

In [82]:
```
bayes_analysis.test_scores
```

Out[82]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.879249	0.786986	0.883350
Precision	0.451469	0.259241	0.463136
Recall	0.501980	0.513861	0.441584
F1 Score	0.475387	0.344622	0.452103
ROC-AUC Score	0.713688	0.667128	0.689485

Observations

This time the model did best on the highly sampled (and smallest) dataset, which makes sense for Naive Bayes. Also, hyperparameter tuning wouldn't do much. The final model's max accuracy is less than a blind guess. I will leave it out of the final comparison for these reasons.

Model 4: Decision Tree

I have a lot more hope for decision trees. While with trees it is not neccesary to transform the data, since I already have, I will keep the data the same. I will examine four types of trees:

- Regular Decision Tree
- Gradient Boosted Tree
- Ada Boosted Tree
- XGBoosted Tree

First, let's have a look at a decision tree.

In [83]:
```
from sklearn.tree import DecisionTreeClassifier

pipe_tree = Pipeline([('mm', MinMaxScaler()), ('tree', DecisionTreeClassifi

tree_analysis = Model_Analysis(pipe_tree, data_dict, X_test, y_test)
tree_analysis.calc_scores()
tree_analysis.score_table
```

Out[83]:

	None Train	None Test	SMOTE Train	SMOTE Test	Sampled Train	Sampled Test
Accuracy	1.0	0.886911	1.0	0.883134	1.0	0.824431
Precision	1.0	0.482975	1.0	0.468616	1.0	0.364515
Recall	1.0	0.533663	1.0	0.539604	1.0	0.821782
F1 Score	1.0	0.507056	1.0	0.501611	1.0	0.505020
ROC-AUC Score	1.0	0.731892	1.0	0.732379	1.0	0.823268

In [84]:
```
tree_analysis.test_scores
```

Out[84]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.886911	0.883134	0.824431
Precision	0.482975	0.468616	0.364515
Recall	0.533663	0.539604	0.821782
F1 Score	0.507056	0.501611	0.505020
ROC-AUC Score	0.731892	0.732379	0.823268

Without any hyperparameter tuning, the model is horrendously overfit, and the max accuracy score, again, barely clears the minimum requirement. Next, I will build a gridsearch with the following hyperparameters:

In [85]:
```
grid = {
    'tree__criterion': ['gini', 'entropy'],
    'tree__splitter': ['best', 'random'],
    'tree__max_depth': [5, 7, 9],
    'tree__max_features': [5, 8, None],
    'tree__min_samples_leaf': [3, 5, 10]
}

tree_search = GridSearchCV(pipe_tree, param_grid=grid, scoring='f1', cv=5)
tree_analysis2 = Model_Analysis(tree_search, data_dict, X_test, y_test)
tree_analysis2.calc_scores()
tree_analysis2.score_table
```

Out[85]:

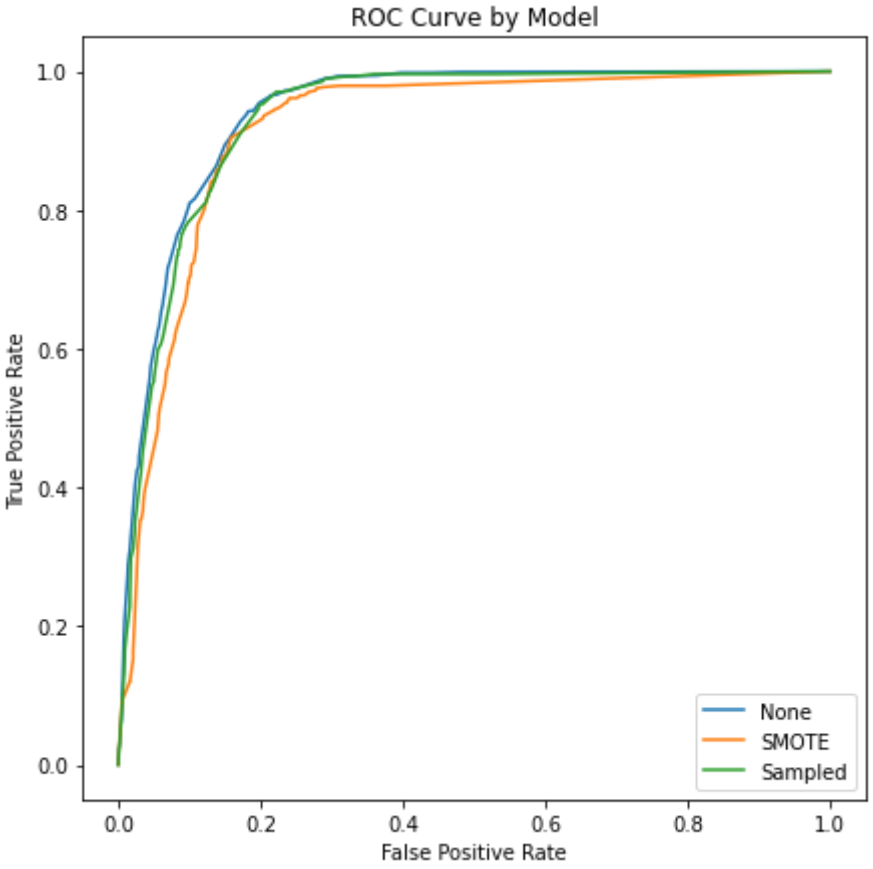
	None Train	None Test	SMOTE Train	SMOTE Test	Sampled Train	Sampled Test
Accuracy	0.916688	0.912917	0.929436	0.875904	0.881666	0.818496
Precision	0.644625	0.606061	0.905560	0.458482	0.837190	0.369868
Recall	0.599874	0.574257	0.958871	0.765347	0.947618	0.945545
F1 Score	0.621445	0.589731	0.931453	0.573442	0.888988	0.531737
ROC-AUC Score	0.778662	0.764300	0.929436	0.827387	0.881666	0.874250

In [86]:
```
tree_analysis2.test_scores
```

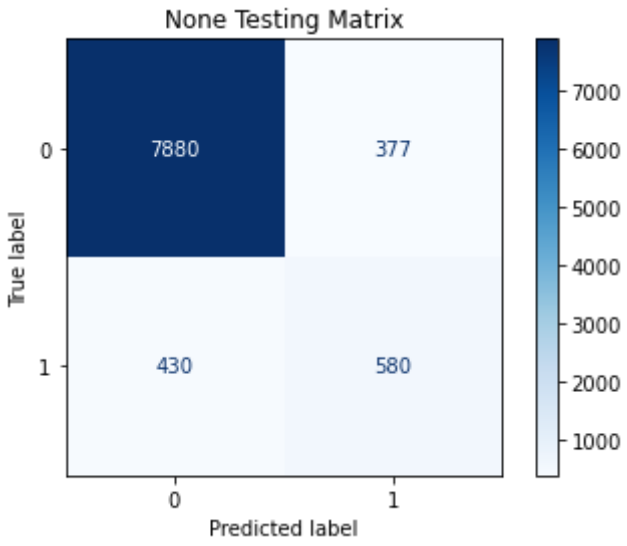
Out[86]:

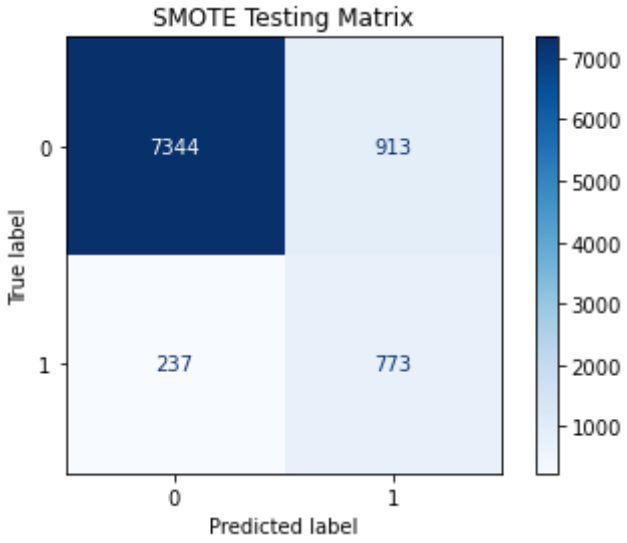
	None Test	SMOTE Test	Sampled Test
Accuracy	0.912917	0.875904	0.818496
Precision	0.606061	0.458482	0.369868
Recall	0.574257	0.765347	0.945545
F1 Score	0.589731	0.573442	0.531737
ROC-AUC Score	0.764300	0.827387	0.874250

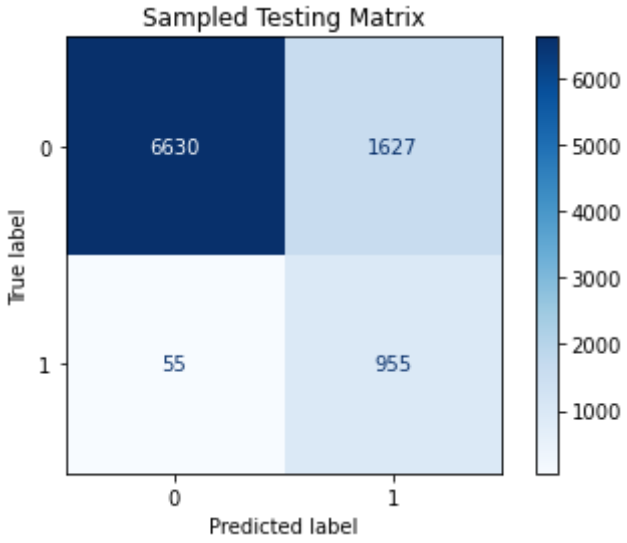
In [87]:
```
tree_analysis2.plot_roc()
```



In [88]:
```
tree_analysis2.confusion_compare()
```







Out[88]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.912917	0.875904	0.818496
Precision	0.606061	0.458482	0.369868
Recall	0.574257	0.765347	0.945545
F1 Score	0.589731	0.573442	0.531737
ROC-AUC Score	0.764300	0.827387	0.874250

In [89]:
```
tree_search.fit(X_train, y_train)
tree_search.best_params_
```

Out[89]:
```
{'tree__criterion': 'entropy',
 'tree__max_depth': 7,
 'tree__max_features': None,
 'tree__min_samples_leaf': 10,
 'tree__splitter': 'best'}
```

Observations

Once again, the None dataset had the best overall scores on the Decision Tree model with an accuracy of 91% and an F1 of 59%. Let's see how it does on the Ada and Gradient Boosted models.

Model 5: Ada Boost and Gradient Boost

In [90]:
```
# Instantiate an AdaBoostClassifier
adaboost_clf = AdaBoostClassifier(random_state=42)

# Instantiate an GradientBoostingClassifier
gbt_clf = GradientBoostingClassifier(random_state=42)
```

In [91]:
```
ada_analysis = Model_Analysis(adaboost_clf, data_dict, X_test, y_test)
gbt_analysis = Model_Analysis(gbt_clf, data_dict, X_test, y_test)
```

In [92]:
```
ada_analysis.calc_scores()
ada_analysis.score_table
```

Out[92]:

	None Train	None Test	SMOTE Train	SMOTE Test	Sampled Train	Sampled Test
Accuracy	0.907047	0.912377	0.932562	0.893169	0.869675	0.863494
Precision	0.657852	0.659677	0.936711	0.509107	0.866437	0.437161
Recall	0.384664	0.404950	0.927812	0.553465	0.874093	0.878218
F1 Score	0.485464	0.501840	0.932240	0.530361	0.870248	0.583745
ROC-AUC Score	0.679461	0.689698	0.932562	0.744094	0.869675	0.869955

In [93]:
```
ada_analysis.test_scores
```

Out[93]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.912377	0.893169	0.863494
Precision	0.659677	0.509107	0.437161
Recall	0.404950	0.553465	0.878218
F1 Score	0.501840	0.530361	0.583745
ROC-AUC Score	0.689698	0.744094	0.869955

In [94]:
```
gbt_analysis.calc_scores()
```

In [95]:
```
gbt_analysis.score_table
```

Out[95]:

	None Train	None Test	SMOTE Train	SMOTE Test	Sampled Train	Sampled Test
Accuracy	0.919386	0.917989	0.938063	0.894033	0.891291	0.846768
Precision	0.693172	0.661499	0.925101	0.510116	0.862573	0.409930
Recall	0.525402	0.506931	0.953309	0.699010	0.930893	0.923762
F1 Score	0.597738	0.573991	0.938993	0.589808	0.895432	0.567864
ROC-AUC Score	0.747740	0.737600	0.938063	0.808449	0.891291	0.880556

In [96]:
```
gbt_analysis.test_scores
```

Out[96]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.917989	0.894033	0.846768
Precision	0.661499	0.510116	0.409930
Recall	0.506931	0.699010	0.923762
F1 Score	0.573991	0.589808	0.567864
ROC-AUC Score	0.737600	0.808449	0.880556

Both models did fairly well on the test sets, with the None dataset performing best with both. Now, let's change up the hyperparameters to see if we can increase the overall accuracy.

In [97]:
```
ada_pipe = Pipeline([('mm', MinMaxScaler()),('ada', AdaBoostClassifier(rand

params = {
    'ada__learning_rate': [0.1, 0.2, 0.3],
    'ada__n_estimators': [25, 50, 75]
}

ada_grid = GridSearchCV(ada_pipe, params, scoring='f1', cv=5)
ada_analysis2 = Model_Analysis(ada_grid, data_dict, X_test, y_test)
ada_analysis2.calc_scores()
ada_analysis2.score_table
```

Out[97]:

	None Train	None Test	SMOTE Train	SMOTE Test	Sampled Train	Sampled Test
Accuracy	0.905464	0.911406	0.921153	0.881623	0.868728	0.856156
Precision	0.657359	0.656198	0.907680	0.471249	0.854198	0.424779
Recall	0.356579	0.393069	0.937678	0.705941	0.889240	0.902970
F1 Score	0.462357	0.491641	0.922435	0.565200	0.871367	0.577764
ROC-AUC Score	0.666333	0.683939	0.921153	0.804527	0.868728	0.876700

In [98]:
```
ada_analysis2.test_scores
```

Out[98]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.911406	0.881623	0.856156
Precision	0.656198	0.471249	0.424779
Recall	0.393069	0.705941	0.902970
F1 Score	0.491641	0.565200	0.577764
ROC-AUC Score	0.683939	0.804527	0.876700

In [99]:
```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    gbt_pipe = Pipeline([('mm', MinMaxScaler()),('gbt', GradientBoostingCla

    params = {
    'gbt__learning_rate': [0.5, 0.1],
    'gbt__n_estimators': [100, 125],
    'gbt__min_samples_split': [3],
    'gbt__max_depth': [4, 5]
    }

    gbt_grid = GridSearchCV(gbt_pipe, params, scoring='f1', cv=5)
    gbt_analysis2 = Model_Analysis(gbt_grid, data_dict, X_test, y_test)
    gbt_analysis2.calc_scores()
    gbt_analysis2.score_table
```

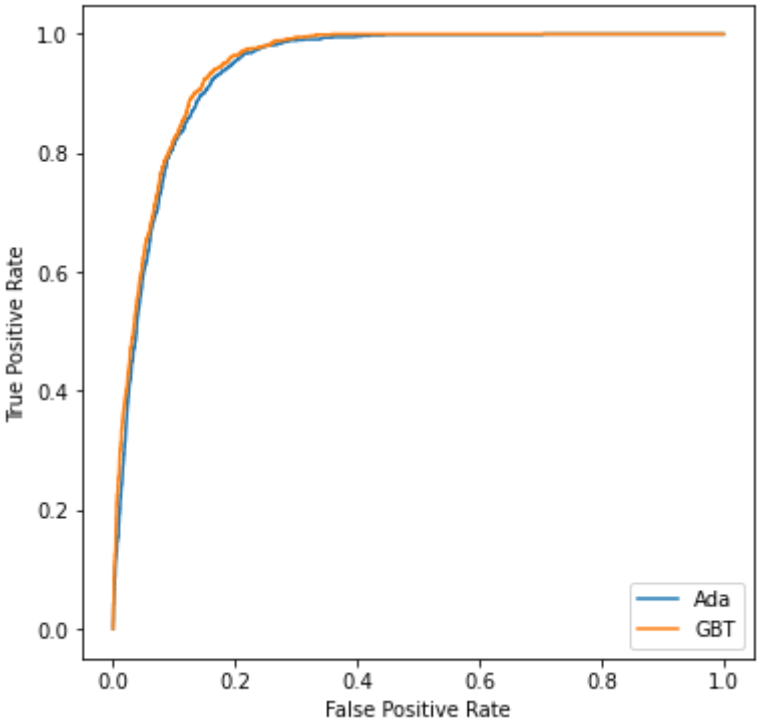
In [100]:
```
gbt_analysis2.test_scores
```

Out[100]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.916370	0.898673	0.846336
Precision	0.633979	0.527713	0.409607
Recall	0.550495	0.669307	0.928713
F1 Score	0.589295	0.590135	0.568485
ROC-AUC Score	0.755809	0.798018	0.882487

In [101]:
```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")

    ada_grid.fit(X_train, y_train)
    gbt_grid.fit(X_train, y_train)
```

In [102]:
```
plt.figure(figsize=(6,6))
classifiers = [ada_grid, gbt_grid]
names = ['Ada', 'GBT']

ax = plt.gca()
for n, i in enumerate(classifiers):
    plot_roc_curve(i, X_test, y_test, ax=ax, label=names[n]);
```



Model 6: XGBoost

My final model is the XGBoosted Tree. Tyically, this scores better than the basic gradient boosting of Ada and GBT, so I'm hopeful we will get a solid option for final comparison.

In [103]:
```
xgb = Pipeline([('mm', MinMaxScaler()), ('xgb', XGBClassifier(random_state=

xgb_analysis = Model_Analysis(xgb, data_dict, X_test, y_test)
xgb_analysis.calc_scores()
xgb_analysis.score_table
```

Out[103]:

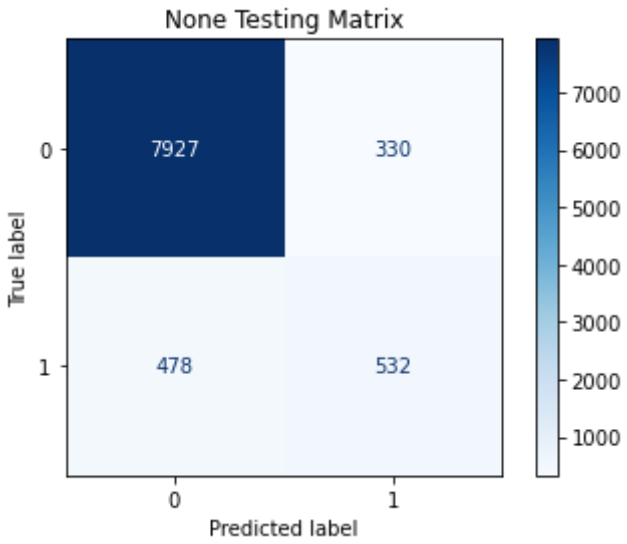
	None Train	None Test	SMOTE Train	SMOTE Test	Sampled Train	Sampled Test
Accuracy	0.958596	0.912809	0.974300	0.906874	0.970811	0.850005
Precision	0.875093	0.617169	0.976933	0.570878	0.959360	0.414875
Recall	0.742821	0.526733	0.971539	0.586139	0.983275	0.916832
F1 Score	0.803550	0.568376	0.974228	0.578407	0.971170	0.571252
ROC-AUC Score	0.864590	0.743383	0.974300	0.766122	0.970811	0.879331

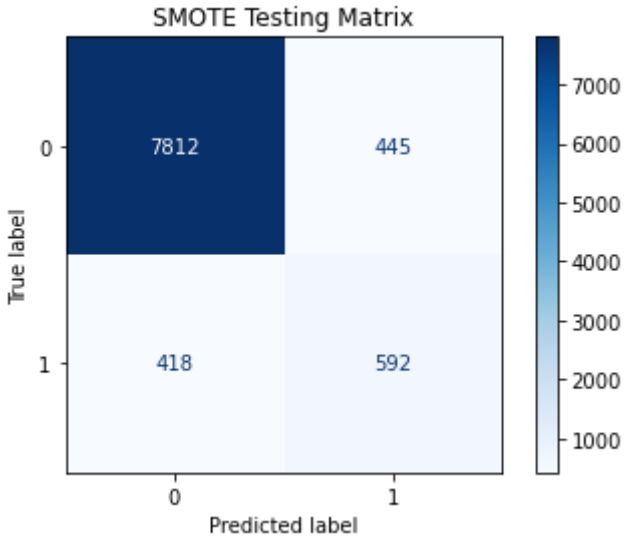
In [104]:
```
xgb_analysis.test_scores
```

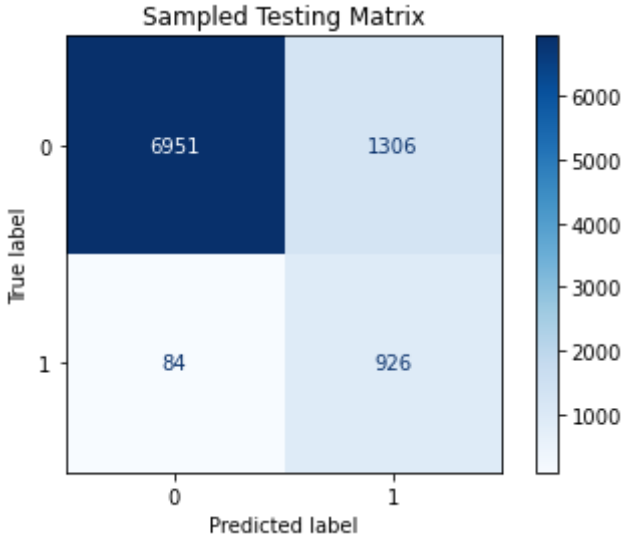
Out[104]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.912809	0.906874	0.850005
Precision	0.617169	0.570878	0.414875
Recall	0.526733	0.586139	0.916832
F1 Score	0.568376	0.578407	0.571252
ROC-AUC Score	0.743383	0.766122	0.879331

In [105]:
```
xgb_analysis.confusion_compare()
```



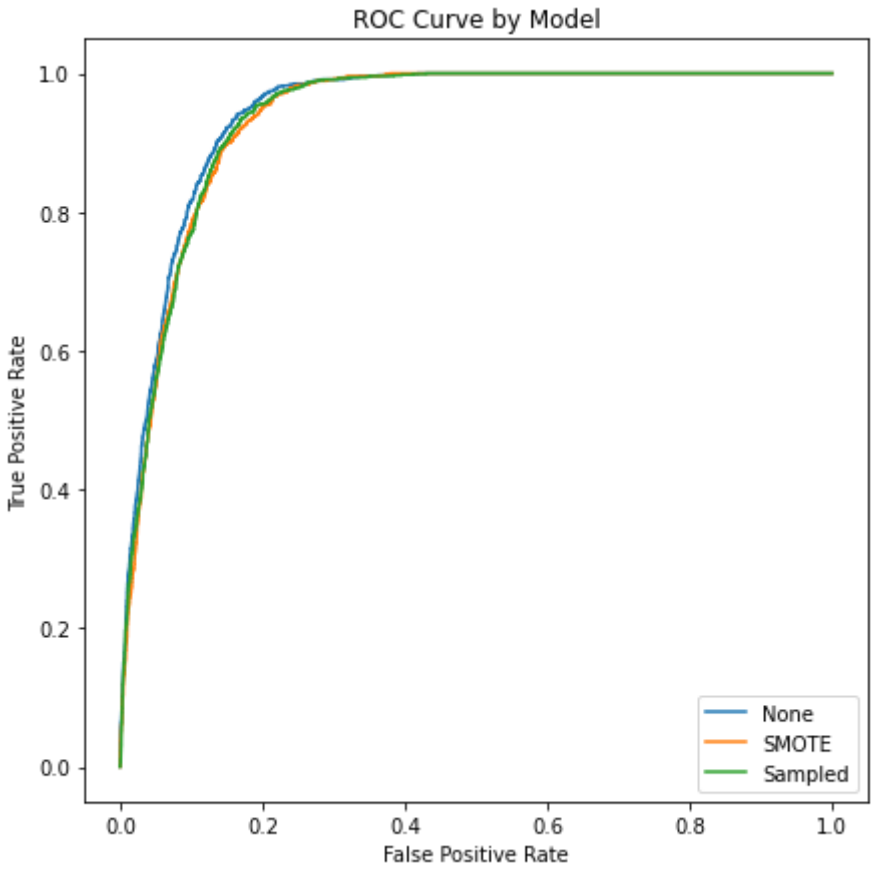




Out[105]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.912809	0.906874	0.850005
Precision	0.617169	0.570878	0.414875
Recall	0.526733	0.586139	0.916832
F1 Score	0.568376	0.578407	0.571252
ROC-AUC Score	0.743383	0.766122	0.879331

In [106]:
```
xgb_analysis.plot_roc()
```



Observations

Already, we can see a significant difference with the accuracy scores on both the SMOTE and the None test sets. Let's tune some hyperparameters.

Please keep in mind the next few cells take a long time to run

In [107]:
```
param_xgb = {
    'xgb__n_estimators': [75, 100, 125],
    'xgb__learning_rate': [0.1, 0.2],
    'xgb__max_depth': [5, 6, 7],
    'xgb__min_child_weight': [1, 2],
    'xgb__subsample': [0.5, 0.7]
}

xgb_grid = GridSearchCV(xgb, param_xgb, scoring='f1', cv=5)
xgb_analysis2 = Model_Analysis(xgb_grid, data_dict, X_test, y_test)
xgb_analysis2.calc_scores()
xgb_analysis2.score_table
```

Out[107]:

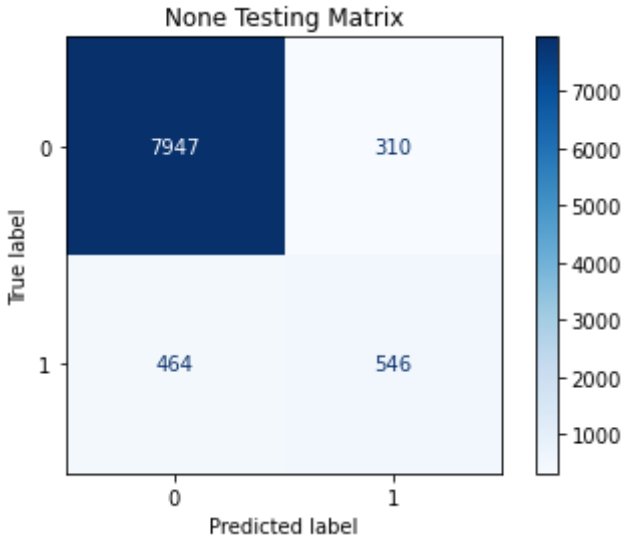
	None Train	None Test	SMOTE Train	SMOTE Test	Sampled Train	Sampled Test
Accuracy	0.931400	0.916478	0.947036	0.900939	0.910697	0.851300
Precision	0.746870	0.637850	0.937358	0.535826	0.884718	0.417930
Recall	0.602398	0.540594	0.958100	0.681188	0.944462	0.927723
F1 Score	0.666900	0.585209	0.947616	0.599826	0.913614	0.576261
ROC-AUC Score	0.788065	0.751525	0.947036	0.804503	0.910697	0.884838

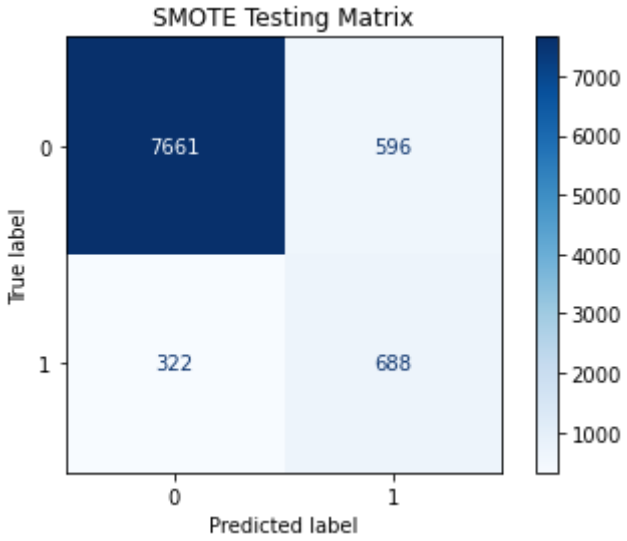
In [108]:
```
xgb_analysis2.test_scores
```

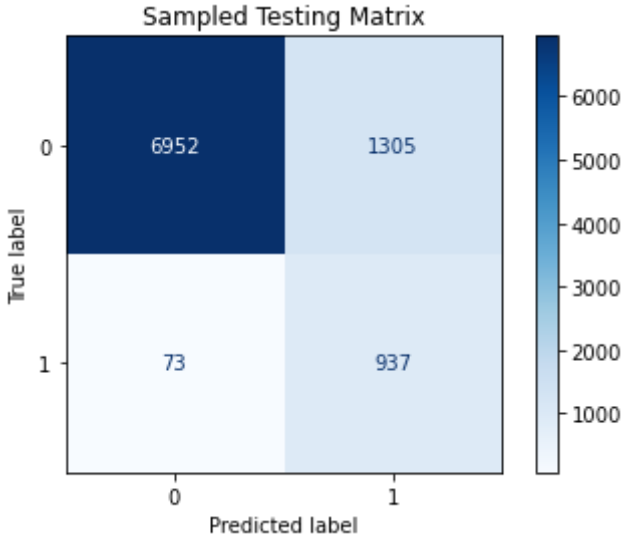
Out[108]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.916478	0.900939	0.851300
Precision	0.637850	0.535826	0.417930
Recall	0.540594	0.681188	0.927723
F1 Score	0.585209	0.599826	0.576261
ROC-AUC Score	0.751525	0.804503	0.884838

In [109]:
```
xgb_analysis2.confusion_compare()
```





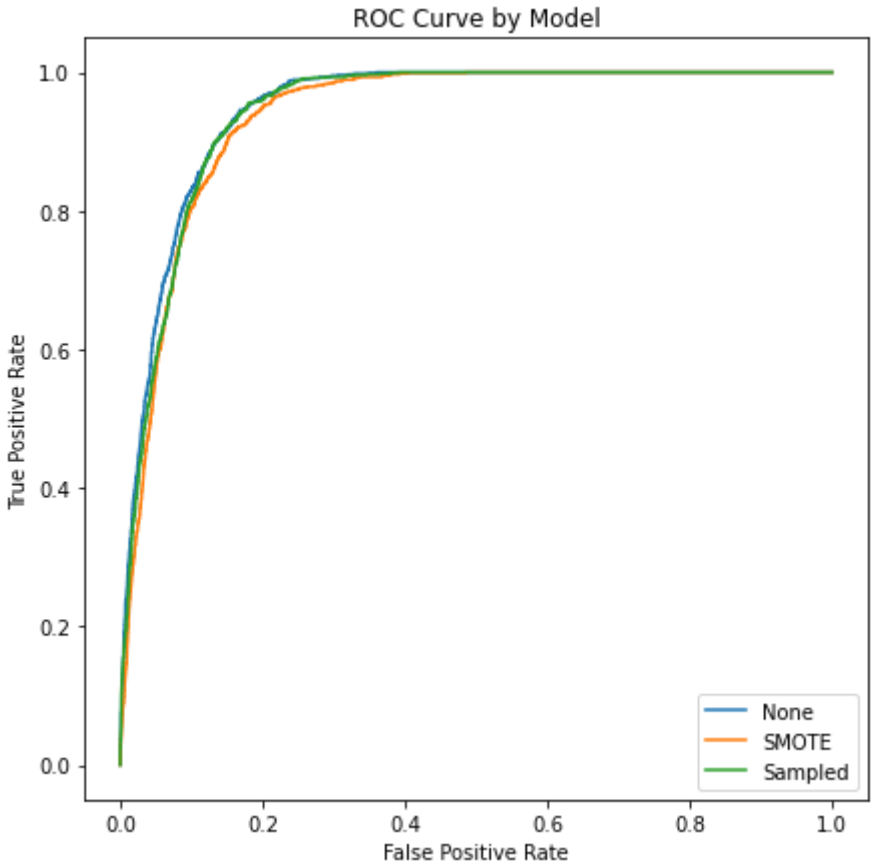


Out[109]:

	None Test	SMOTE Test	Sampled Test
Accuracy	0.916478	0.900939	0.851300

	None Test	SMOTE Test	Sampled Test
Precision	0.637850	0.535826	0.417930
Recall	0.540594	0.681188	0.927723
F1 Score	0.585209	0.599826	0.576261
ROC-AUC Score	0.751525	0.804503	0.884838

In [110]:
```
xgb_analysis2.plot_roc()
```



In [111]:
```
xgb_grid.fit(X_train, y_train)
```

Out[111]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('mm', MinMaxScaler()),
                                       ('xgb',
                                        XGBClassifier(base_score=0.5,
                                                      booster='gbtree',
                                                      colsample_bylevel=
1,
                                                      colsample_bynode=1,
                                                      colsample_bytree=1,
                                                      gamma=0, gpu_id=-1,
                                                      importance_type='ga
in',
                                                      interaction_constra
ints='',
                                                      learning_rate=0.300
000012,
                                                      max_delta_step=0,
                                                      max_depth=6,
                                                      min_child_weight=1,
                                                      missing=nan,
                                                      monotone_constrain
t...
                                                      n_estimators=100,
                                                      n_jobs=0,
                                                      num_parallel_tree=
1,
                                                      random_state=42,
                                                      reg_alpha=0, reg_la
mbda=1,
                                                      scale_pos_weight=1,
                                                      subsample=1,
                                                      tree_method='exac
t',
                                                      validate_parameters
=1,
                                                      verbosity=None))]),
             param_grid={'xgb__learning_rate': [0.1, 0.2],
                         'xgb__max_depth': [5, 6, 7],
                         'xgb__min_child_weight': [1, 2],
                         'xgb__n_estimators': [75, 100, 125],
                         'xgb__subsample': [0.5, 0.7]},
             scoring='f1')
```

In [112]:
```
xgb_grid_SMOTE = GridSearchCV(xgb, param_xgb, scoring='f1', cv=5)
xgb_grid_SMOTE.fit(X_train_SMOTE, y_train_SMOTE)
```

Out[112]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('mm', MinMaxScaler()),
                                       ('xgb',
                                        XGBClassifier(base_score=0.5,
                                                      booster='gbtree',
                                                      colsample_bylevel=
1,
                                                      colsample_bynode=1,
                                                      colsample_bytree=1,
                                                      gamma=0, gpu_id=-1,
                                                      importance_type='ga
in',
                                                      interaction_constra
ints='',
                                                      learning_rate=0.300
000012,
                                                      max_delta_step=0,
                                                      max_depth=6,
                                                      min_child_weight=1,
                                                      missing=nan,
                                                      monotone_constrain
t...
                                                      n_estimators=100,
                                                      n_jobs=0,
                                                      num_parallel_tree=
1,
                                                      random_state=42,
                                                      reg_alpha=0, reg_la
mbda=1,
                                                      scale_pos_weight=1,
                                                      subsample=1,
                                                      tree_method='exac
t',
                                                      validate_parameters
=1,
                                                      verbosity=None))]),
             param_grid={'xgb__learning_rate': [0.1, 0.2],
                         'xgb__max_depth': [5, 6, 7],
                         'xgb__min_child_weight': [1, 2],
                         'xgb__n_estimators': [75, 100, 125],
                         'xgb__subsample': [0.5, 0.7]},
             scoring='f1')
```

Observations

Overall the None dataset did the best for the XGBoost, where the sampled set still remained worse than a blind guess, and the SMOTE barely provides any improvement over that. I will carry over the SMOTE model to the final comparison just for sake of variety.

Final Comparison

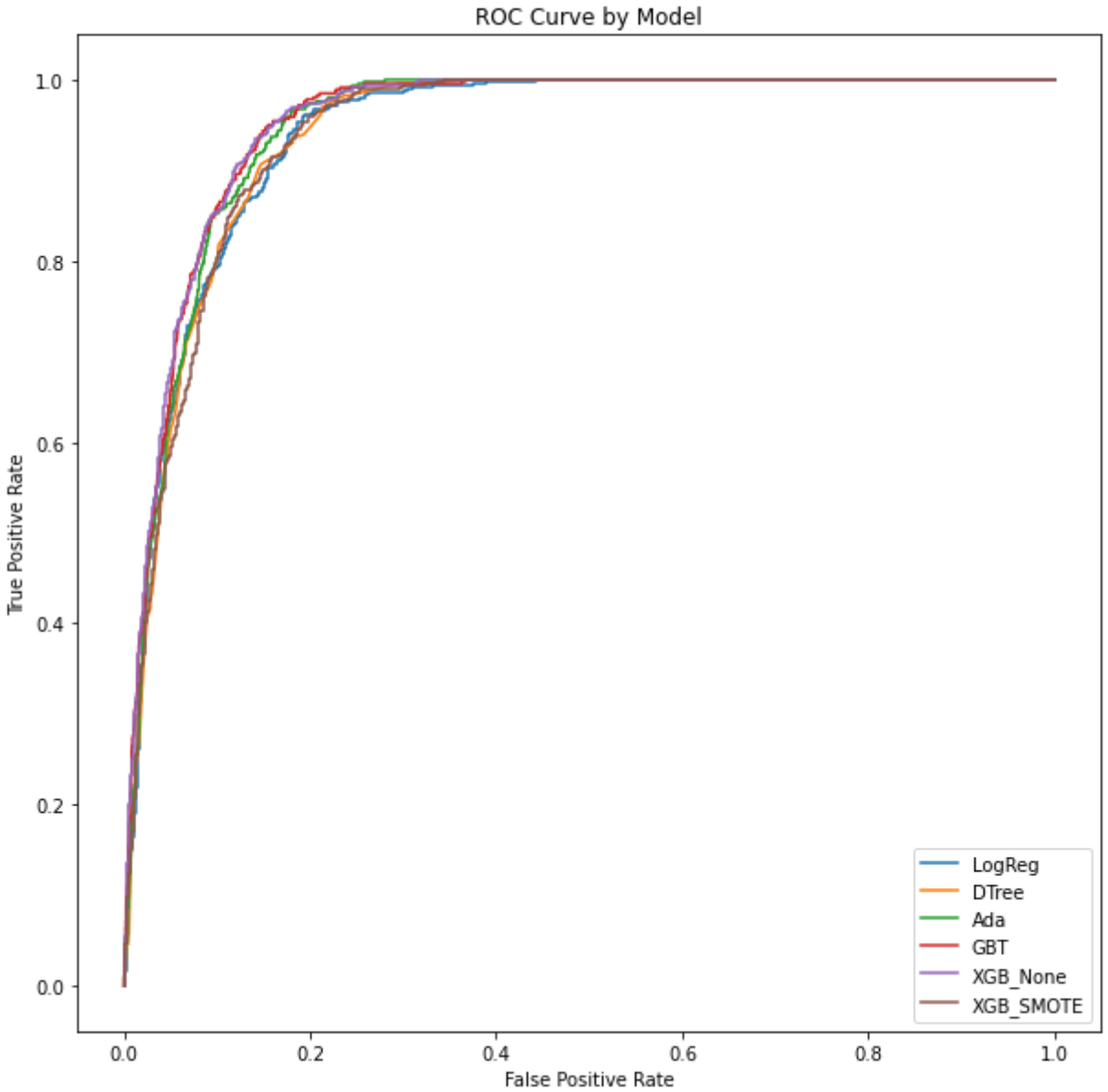
Let's see how everything stacks up comparatively. Just a reminder, we will be comparing the scores of the following models:

- Logistic Regression Gridsearch with the None Dataset
- Decision Tree Gridsearch with the None Dataset
- Ada Boosted Tree with the None Dataset
- Gradient Boosted Tree with the None Dataset
- XGBoosted Tree with the None Dataset
- XGBoosted Tree with the SMOTE Dataset

First, let's prep our holdout Dataset

In [113]:
```
X_holdout = holdout.drop(columns=['subscribed_yes'])
y_holdout = holdout['subscribed_yes']
```

In [114]:
```
#Logreg NONE
#Decision Tree - NONE
#Ada Boost - None
#GBT Boost - None
#XGBOOST - None
#XGBoost - SMOTE

plt.figure(figsize=(10,10))
classifiers = [log_reg_best, tree_search, ada_grid, gbt_grid, xgb_grid, xgb
names = ['LogReg', 'DTree', 'Ada', 'GBT', 'XGB_None', 'XGB_SMOTE']
ax = plt.gca()
for n, i in enumerate(classifiers):
    plot_roc_curve(i, X_holdout, y_holdout, ax=ax, label=names[n]);
plt.title('ROC Curve by Model')
plt.show()
```



In [115]:
```
df_list = []

for n, i in enumerate(classifiers):
    y_pred = i.predict(X_holdout)

    dictionary = {
        'Accuracy': accuracy_score(y_holdout, y_pred),
        'Precision': precision_score(y_holdout, y_pred),
        'Recall': recall_score(y_holdout, y_pred),
        'F1 Score': f1_score(y_holdout, y_pred),
        'ROC-AUC Score': roc_auc_score(y_holdout, y_pred)
    }

    df = pd.DataFrame.from_dict(dictionary, orient='index', columns=[names[
    df_list.append(df)

final_scores = pd.concat(df_list, axis=1)
final_scores.style.highlight_max(color='lightgreen', axis=1)
```

Out[115]:

	LogReg	DTree	Ada	GBT	XGB_None	XGB_SMOTE
Accuracy	0.914279	0.913550	0.911608	0.918164	0.917921	0.902623
Precision	0.617391	0.624703	0.699588	0.664894	0.664879	0.555970
Recall	0.616052	0.570499	0.368764	0.542299	0.537961	0.646421
F1 Score	0.616721	0.596372	0.482955	0.597372	0.594724	0.597793
ROC-AUC Score	0.783963	0.763647	0.674401	0.753922	0.751890	0.790670

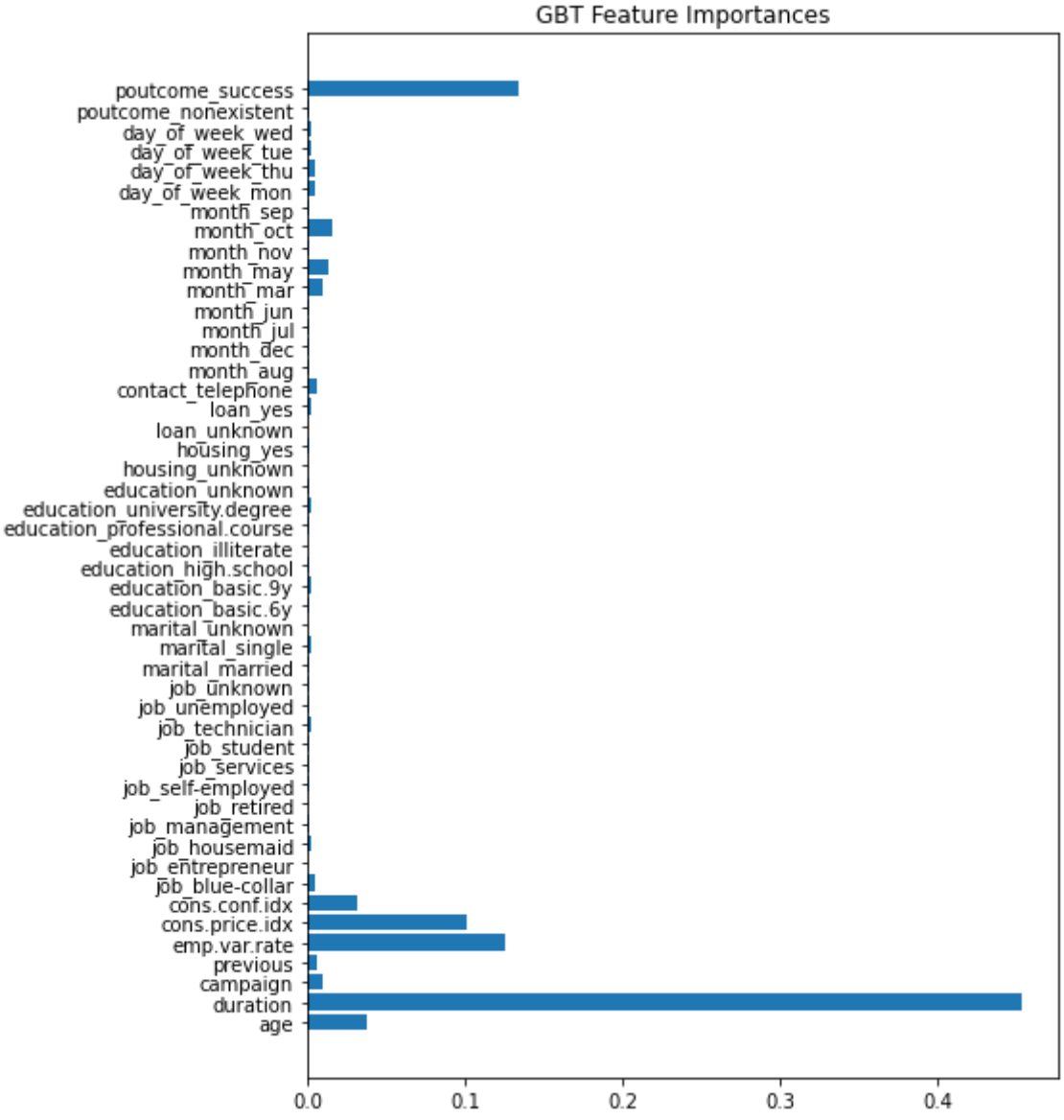
Observations

Based on these numbers, both the **Gradient Boosted Random Forest** and the **XGBoosted Forest** had the best accuracy scores and, in fact, had almost exactly the same metrics across the board. While the Gradient Boosted model had the best score on the holdout, I will consider both viable for analysis.

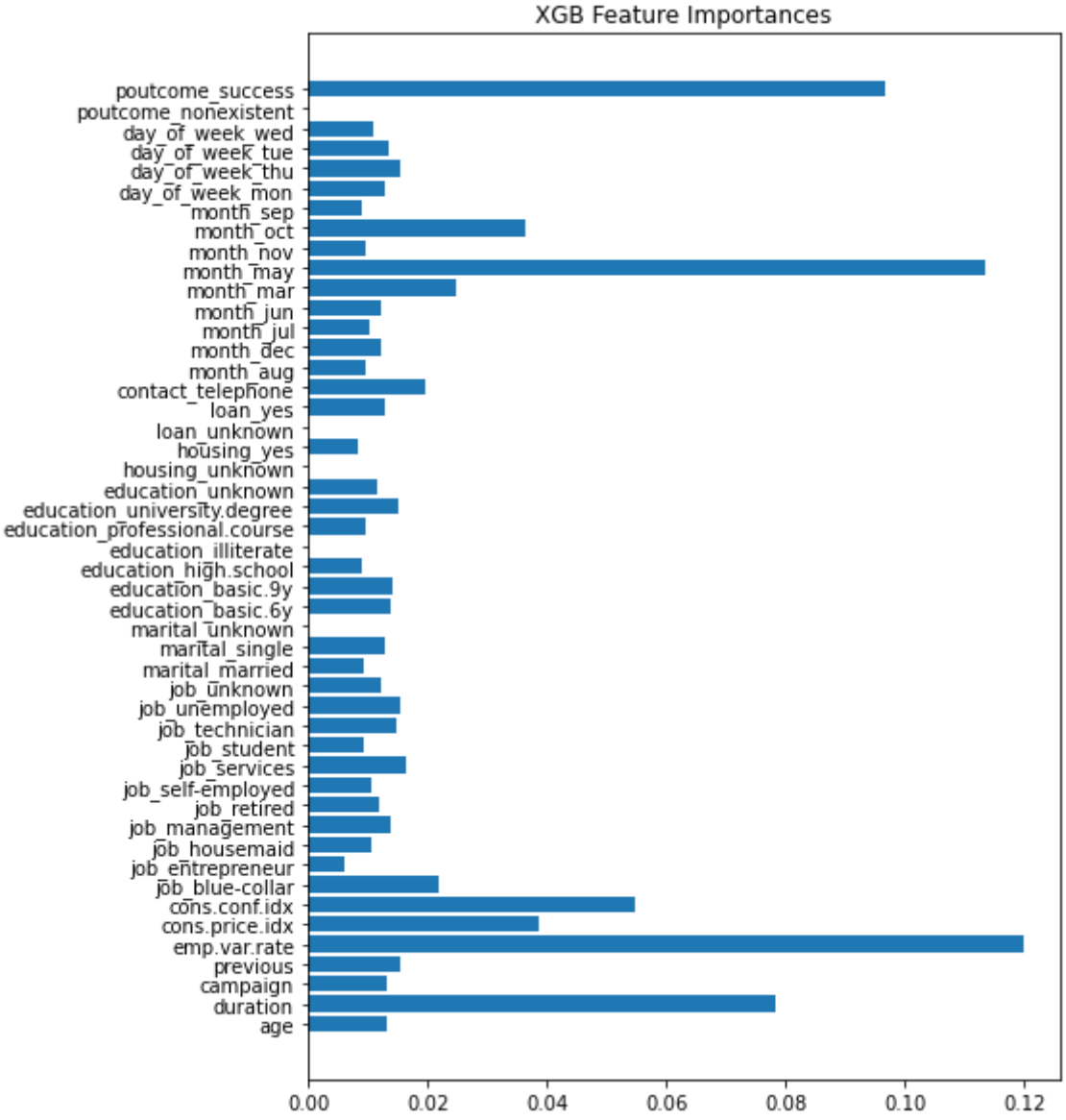
In [123]:
```
best_model = gbt_grid.best_estimator_.named_steps['gbt']
```

In [125]:
```
second_best_model = xgb_grid.best_estimator_.named_steps['xgb']
```

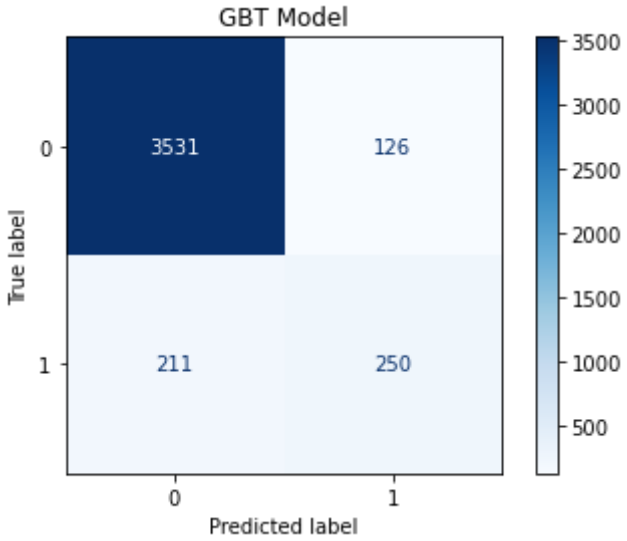
In [127]:
```
plt.figure(figsize=(7,10))
plt.barh(range(len(best_model.feature_importances_)), best_model.feature_im
plt.yticks(np.arange(len(best_model.feature_importances_)), X_holdout.colum
plt.title('GBT Feature Importances')
plt.show()
```



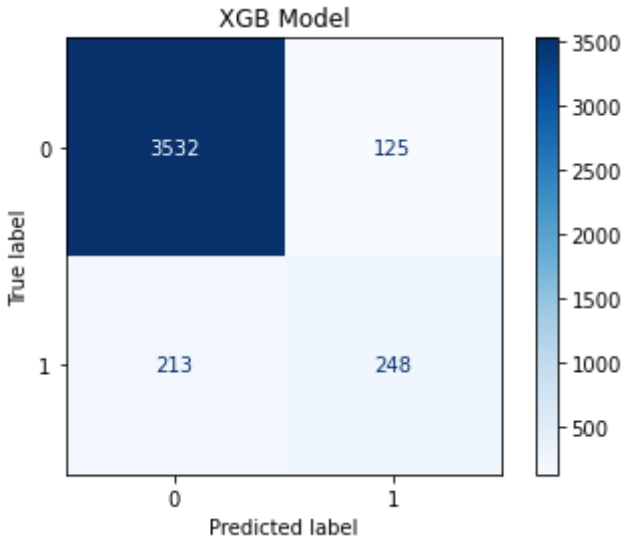
In [128]:
```
plt.figure(figsize=(7,10))
plt.barh(range(len(second_best_model.feature_importances_)), second_best_mo
plt.yticks(np.arange(len(best_model.feature_importances_)), X_holdout.colum
plt.title('XGB Feature Importances')
plt.show()
```



In [129]:
```
plot_confusion_matrix(gbt_grid, X_holdout, y_holdout, cmap=plt.cm.Blues)
plt.title('GBT Model');
```



In [130]:
```
plot_confusion_matrix(xgb_grid, X_holdout, y_holdout, cmap=plt.cm.Blues)
plt.title('XGB Model');
```



Conclusions

In the end, the XGBoosted Model and the Gradient Boosted Models were the most successful on the Holdout data. Surprisingly, transforming the data to account for class imbalance wasn't as helpful as first assumed. It was a good practice to see what models performed better and, depending on the measure we wanted to optimize, each data set had different strengths. While both models had almost exactly the same scores, they had different feature importances.

Model Quality

It is important to note that, with a baseline model, predicting that a sale would be unsuccessful would be correct ~89% of the time. Our final model had an accuracy of 91.8% on the holdout data set, so it is not much more accurate than the baseline. These models do highlight more True Positives than False Positives, which is good in a sales environment, so you don't dedicate resources to the wrong prospects. They did however produce almost as many False Negatives as False Positives, so they will miss a number of good prospects while providing a solid number of successful sales. If your business strategy is less dependent on volume and more focused on closing better deals, this trade off is the best of two evils.

Data Set

For precision and accuracy, the None dataset almost always outperformed the others. While the sampled (and smaller) dataset had the worst precision and accuracy, it usually had the best ROC-AUC, F1 and Recall scores. Finally, SMOTE tended to be a split between the two.

Model Selection

Using the XGBoosted and Gradient Boosted models had the best performance overall on the untampered holdout data set. It's clear that KNN is simply not feasible on a dataset with 40k rows and, even when it did complete, it was inaccurate compared to the others. Logistic Regression worked well, comparatively, but a gradient boosted decision tree seems to be the most successful option.

Demographics or Firmographics?

Based on the feature importance of the model, it was surprising to see which features were the most important and which were the least.

In [134]:
```
gbt_importances = pd.DataFrame.from_dict(dict(zip(X_holdout.columns, best_m
gbt_importances.sort_values(by='GBT Importance',ascending=False).head()
```

Out[134]:

	GBT Importance
duration	0.453804
poutcome_success	0.134174
emp.var.rate	0.125078
cons.price.idx	0.100826
age	0.037259

In [135]:
```
 pd.DataFrame.from_dict(dict(zip(X_holdout.columns, second_best_model.featu
ort_values(by='XGB Importance',ascending=False).head()
```

Out[135]:

	XGB Importance
emp.var.rate	0.119879
month_may	0.113280
poutcome_success	0.096654
duration	0.078137
cons.conf.idx	0.054868

Most Important:

Interestingly, the two models had several similarities in which features were important, the weight of each is very different. Here are a few take-aways

```
1. Duration is a key factor in both models but drastically more imp
ortant in GBT
2. When the person was called mattered with the XGB model, but not
 the GBT
3. If they previously subscribed to a campaign was essential to bot
h
4. The employment rate was important for both
5. Consumer Confidex Index/price were important for both
6. Age was a factor in GBT but not XGB
```

Let us break each of these down.

1) Duration of the call is more of an indicator that the sale will go through, rather than the basis of a calling strategy. The fact that the call is a success is probably the deciding factor in the length of the call rather than the longer the call the more likely it will be a successful sale. One should not

train sales people to just drone on in order to sell a subscription. That said, if a call is going a long time, there's a good chance it will end in a sale.

2) the May factor might be a case of confirmation bias. 33% of the calls were made in May, so it would make sense that a higher number of successful calls occurred then as well. This might be because the company carrying out this campaign might have already noticed an uptick in subscriptions in the month of May and therefore decided to allocate resources to emphasize that, or it might simply be due to the fact that the company had more staff during that month in order to increase volume. Perhaps there is an underlying link between the beginning of Summer and consumer purchasing habits.

3) It makes perfect sense that someone who previously subscribed to a similar campaign would subscribe to another one. This means that pursuing your current client list for growth opportunities will almost always be a good use of resources.

4) When the employment rate is higher, you sell more subscriptions. This makes sense but it means that larger economic metrics might be the most influential factors when it comes to sales success. If more people are fully employed, they'll have more disposable income for a subscription. It's also telling that the fifth most important factor is the consumer confidence index for XGB and fourth for GBT.

5) While the consumer confidence index and consumer price index are not necessarily the same thing, they are somewhat linked. The confidence index is a measure of optimism of the consumer with regards to the economy and the price index is a measure of the fluctation of pricing on consumer goods. Once again, this encourages me to explore macro-economic factors when exploring future models of this kind.

6) Age seems like a sensible measure when it comes to determining a sales success, though I don't think that means the older the client, the more likely the sale. Rather, identifying consumers in certain age ranges is helpful information when in a sales environment.

Now, let's look at the opposite list.

In [136]:
```
gbt_importances.sort_values(by='GBT Importance',ascending=True).head(5)
```

Out[136]:

	GBT Importance
loan_unknown	0.000079
housing_unknown	0.000160
marital_unknown	0.000333
education_illiterate	0.000333
job_retired	0.000358

In [139]:
```
xgb_importances.sort_values(by='XGB Importance',ascending=True).head(5)
```

Out[139]:

	XGB Importance
poutcome_nonexistent	0.0
housing_unknown	0.0
marital_unknown	0.0
loan_unknown	0.0
education_illiterate	0.0

Least Important:

Both models have massive similarities in terms of what is least important. In no particular order, they are:

1. Housing status unknown
2. Unknown Marital Status
3. Unknown Loan Status
4. Illiterate

Other than illeteracy, the majority of these factors are based on the fact that we didn't have information for those prospects. All we can glean from this is that the more data we have about our prospect, the better we can predict their interest in a subscription, which is no surprise. Lastly, it makes sense that someone who is illiterate is unlikely to purchase a financial product.

Applying This Elsewhere

While this dataset doesn't directly apply to business to business marketing and sales, there are a number of lessons that can be gleaned from this model. First and foremost, I explored a methodology of approaching a heavily imbalanced classification problem. The key takeaways are:

```
1. When considering data transformations, both SMOTE and leaving th
e data imbalance alone are preferable to sampling the data to a sma
ller dataset.
2. More data is better than less
3. Macro Economic Factors can be a helpful indicator of purchase in
tent
4. Past purchase history with your company is a good indicator of p
urchase intent.
5. The demographics of the actual individual are not nearly as impo
rtant as the timing of the sales call
6. F1 Score is a much more preferable to accuracy when scoring a cl
ass imbalance model.
7. Gradient Boosting is a powerful modeling tool and should be cons
idered first or second (behind Logarithmic Regression) whenever bui
lding a new model of this kind. Logistic Regression also performed
 very well.
8. If possible, create a holdout set of data to test a final model
 to ensure you don't accidentally train to the testing data.
```

This project is useful for helping direct resources when data gathering as well. Since the customers with the least information were the least predictable, it would be useful to gather information. Most importantly, however, we must be wary of correlation vs. causation when selecting variables. Creating KPIs based on only making sales calls in May and making the calls as long as possible would be a poor managerial choice. That said, when the employment rate goes up, and the consumer confidence is high, it might be a good time to get calling clients.

Areas of Further Study

The next step would be to build a dataset of this size for business to business interactions. While demographics for the individual prospect might not have been as important as macro-economic factors, firmographics for the companies purchasing new products might be very useful information. Future models should not include factors that are affected by the success of the call, it should only include factors that might affect the outcome itself. That way, the model will be able to highlight best sales practices, rather than best indicators or a sale that has already occurred.

This project will be invaluable in determining data collection next steps and the tools used are appropriate for evaluating and building models for similar problems in the future.