Credit Card Fraud Detection and Energy Efficiency Analysis

By Seneca Anderson

I. Introduction

The purpose of this paper is to describe the process and results of applying machine learning to two analysis problems: credit card fraud detection and energy efficiency in buildings. Various types of data pre-processing and machine learning algorithms were applied to these problems, and their performance was evaluated using various relevant metrics..

The credit card fraud detection problem was a classification problem. Out of the three classifiers and seven sampling methods, two combinations—the random forest classifier with random oversampling and the random forest classifier with SMOTE—had the best balance of precision, recall, and accuracy. Which one is superior depends on one's priorities.

The energy analysis problem was a regression problem. Five regression models were applied; each was trained and evaluated twice, once with unscaled data and once with scaled data. Overall, a polynomial regressor with no scaling performed best, with a mean squared error of 1.0826 and an explained variance score of 0.9884.

II. Credit Card Fraud Detection

Summary

The purpose of the credit card fraud detection problem was to build a classifier that could accurately detect instances of credit card fraud. Due to the large number of samples and the severe class imbalance of the dataset, over- and under-sampling methods were used in order to improve performance. Different sampling methods were combined with three different classifiers—a logistic regressor, a neural network, and a random forest classifier—to determine which combination led to the best results. Due to the inverse relationship between precision and recall, determining the best classifier depends on whether it is better to catch more instances of fraud while also flagging more non-fraud instances, or to miss instances of fraud while reducing the number of accidental accusations. The conclusions of this report are made on the assumption that recall should be prioritized over precision, while still retaining a reasonable balance.

Program Structure

The program was organized in a way that reduced duplicate code and prevented output from being influenced by differences in how sampling methods and models were executed.

First, general data-preprocessing was performed; the results applied to every sampling method and classifier.

Each classifier was run via a function that set up the classifier, trained it, and returned evaluation metrics, such as precision and recall. This ensured that the only difference between each execution of the model was the sampling method used.

The sampling methods were also run via a single function; it accepted the classifier's run_model function and a (potentially empty) list of samplers to run, then returned lists of the evaluation metrics returned by the run_model function. This ensured that each sampling method was applied in exactly the same way, avoiding the influence of other variables.

Additionally, all evaluation metrics were generated and printed by the same functions.

Data Pre-Processing

The same data pre-processing methods applied to all the classifiers and sampling methods. After finding some preliminary information about the dataset (ensuring there were no null values and checking the severity of the class imbalance), the dataset was divided into X and y sets. Scikit-learn's train_test_split function was used to divide these sets into train and test data, with 33% going to the test set. Afterwards, the sklearn StandardScaler was used to transform both the X_train and X_test data sets. This step was essential; without it, the neural network achieved an approximately 99.83% accuracy simply by ruling every instance as "not fraud".

Sampling Methods

Multiple sampling methods were tested to see how each affected the results. First, each classifier was run without sampling to get baseline metrics. Then the same classifier was run again with four oversampling methods from imbalance-learn: Random Over Sampling, SMOTE, ADASYN, and Border Line SMOTE. This process was repeated with two undersampling methods: Random Under Sampling and Near Miss.

Originally, several other over- and under-sampling methods were applied. However, due to the severity of the class imbalance, methods that used clustering failed to complete and had to be eliminated.

Classification Methods

Three classification methods were used. Where relevant, the random_state was set to 42 in order to ensure that results could be replicated.

The first model was a simple logistic regression classifier. Its evaluation metrics were accuracy, precision, recall, f1, confusion matrix, and ROC curve.

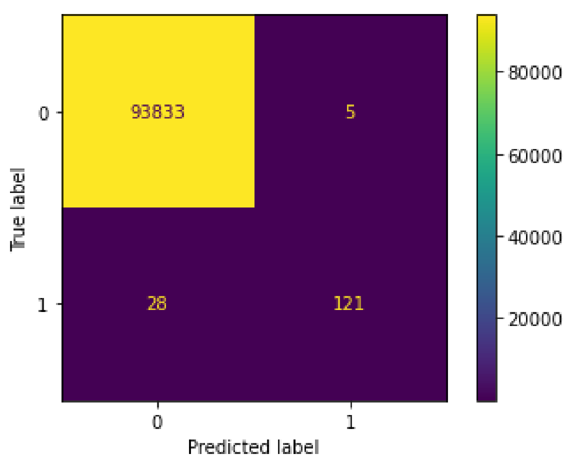
The second model was a keras Sequential neural network with four Dense layers. All the layers used a uniform distribution to initialize the weights and a tanh activation function, except for the last, which used a sigmoid function. The number of neurons in each layer, from first to last, was as follows: 60, 30, 10, 1. The optimizer used was Adam, the loss function was binary cross entropy loss, the number of epochs was twenty, and the batch size was ten. Its evaluation metrics were accuracy, precision, recall, f1, and confusion matrix. Due to incompatibility between keras and a function required to generate the ROC curve, no ROC curve was generated.

The third model was a sklearn's random forest classifier. Its evaluation metrics were accuracy, precision, recall, f1, confusion matrix, and ROC curve.

Results

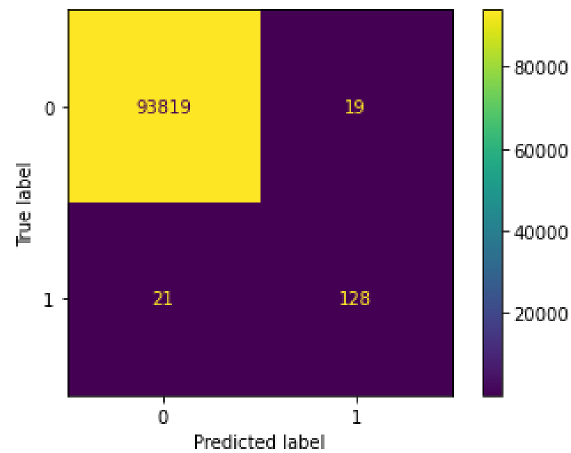
The random forest classifier combined with random oversampling had the highest f1 score (0.8800) and precision (0.9603). It achieved the highest accuracy, 0.9996, which was also achieved by several other models. The high precision shows that this model is the best if one's priority is to reduce the number of accidental accusations of fraud. However, its recall was only 0.8121, lower than the recalls achieved by fifteen other models, indicating that it is not the best model for catching as many instances of fraud as possible.

Confusion Matrix for Random Forest Classifier with Random Oversampling



The random forest classifier with SMOTE oversampling achieved the highest recall with a reasonable precision. Its recall was 0.8591, its precision 0.8707, its f1 score 0.8649, and its overall accuracy 0.9996. This constitutes an 0.0480 increase in recall and an 0.0904 loss of precision.

Confusion Matrix for Random Forest Classifier with SMOTE Oversampling



Higher recalls were achieved by models with much lower precision. In particular, models with over 0.9 recall had abysmal precision scores, less than 0.1.

Which of these models is considered superior depends on one's priorities. If the goal is to find a good balance between high precision and high recall, using the random forest classifier with either random or SMOTE oversampling is best.

Logistic Regression

Sampler	Accuracy	Precision	Recall	F1
None	0.9993	0.8716	0.6376	0.7364
Random Over	0.9729	0.0509	0.9128	0.0964
SMOTE	0.9721	0.0499	0.9195	0.0947
ADASYN	0.9054	0.0156	0.9463	0.0307
Border Line SMOTE	0.9910	0.1340	0.8591	0.2319
Random Under	0.9585	0.0342	0.9262	0.0660
Near Miss	0.5852	0.0035	0.9128	0.0069

Neural Network Classifier

Sampler	Accuracy	Precision	Recall	F1
None	0.9994	0.8039	0.8255	0.8146
Random Over	0.9992	0.7151	0.8255	0.7664
SMOTE	0.9985	0.5079	0.8658	0.6402
ADASYN	0.9989	0.6316	0.8054	0.7080
Border Line SMOTE	0.9993	0.7755	0.7651	0.7703
Random Under	0.9411	0.0244	0.9262	0.0475
Near Miss	0.5532	0.0033	0.9195	0.0065

Random Forest Classifier

Sampler	Accuracy	Precision	Recall	F1
None	0.9996	0.9370	0.7987	0.8623
Random Over	0.9996	0.9603	0.8121	0.8800
SMOTE	0.9996	0.8707	0.8591	0.8649
ADASYN	0.9995	0.8552	0.8322	0.8435
Border Line SMOTE	0.9996	0.9302	0.8054	0.8633
Random Under	0.9676	0.0437	0.9329	0.0836
Near Miss	0.0607	0.0017	0.9866	0.0033

III. Energy Efficiency Analysis

Summary

The purpose of the energy efficiency analysis was to build a regressor that could predict the heating and cooling load requirements of a building based on a variety of factors. Five different regressors were trained and evaluated. This process was performed twice each—once with the original data and once with scaled data. The regressors were evaluated using the mean squared error (MSE) and explained variance score (EVS). The model with the lowest MSE and highest EVS was a polynomial regressor of degree four run on unscaled data.

Data Pre-Processing

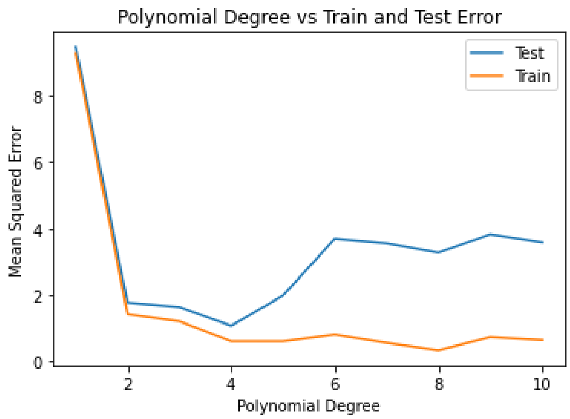
The same general pre-processing steps were applied to the data for all the regressors. First, general attributes of the data, including whether there were any null values, were investigated. Once it was determined that there were no null values, the data was split into X and y. Then it was divided into train and test sets, with 33% of the data used for testing. Scaled versions of X_train and X_test were created using sklearn's StandardScaler.

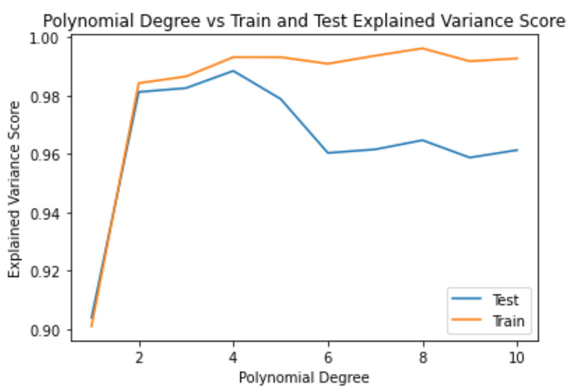
Regression Methods

Five regressors were trained: linear, polynomial, SVM, decision tree, and random forest. Each was trained and evaluated on both scaled and unscaled data. To prevent code differences from affecting the output for scaled and unscaled data, each regressor was built, trained, and evaluated via a function, which was called twice.

Results

The most accurate model was a polynomial regressor with degree four trained on unscaled data. Linear and polynomial regressors with a degree lower degree than four had a higher MSE and lower EVS. With degrees five and higher, the training error and EVS continued to improve while the testing error and EVS worsened, indicating overfitting.





When run on scaled data, the linear and polynomial regressors had bizarre MSE and EVS values, indicating that scaling was not an effective pre-processing method for these regressors.

The second-best model was the random forest regressor, and the third-best was the decision tree. Both had marginally better performance on unscaled data than on scaled data.

The SVM regressor performed the worst out of all the regressors for both scaled and unscaled data. Additionally, running it on unscaled data lead to warnings about failing to converge, indicating a potential source of the extremely high MSE (33.4386) The SVM was the only regressor that performed better on scaled data than on unscaled, with an MSE of 10.8701 and a higher EVS.

	Without Scaling		**With Scaling**	
Model	**MSE**	**EVS**	**MSE**	**EVS**
Polynomial (Degree 4)	1.0826	0.9884	1.1630	0.9876
SVM	33.4386	0.8251	10.8701	0.8948
Decision Tree	2.8324	0.9696	2.8792	0.9690
Random Forest	1.7338	0.9814	1.7540	0.9812

IV. Conclusion

In this project, machine learning models were developed and trained for two datasets. The first program, in which classification models were applied to a credit card fraud detection problem, evaluated four classifiers and seven sampling options to determine the optimal combination. The best combinations were a random forest classifier with random oversampling and a random forest classifier with SMOTE oversampling; which should be used on one's priorities regarding precision vs. recall. The second problem, in which regressors were used to predict the heating and cooling loads of buildings, involved evaluating five regressors and the effects of scaling on accuracy. The best model was a polynomial regressor with degree four trained on unscaled data.

Credit Card Fraud Detection (Classification)

In [1]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from imblearn.over_sampling import RandomOverSampler, SMOTE, ADASYN, BorderlineSMOTE
from imblearn.under_sampling import RandomUnderSampler, NearMiss
```

In [2]:
```
from sklearn.model_selection import cross_val_predict
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score
from sklearn.metrics import precision_recall_curve, roc_curve, confusion_matrix, ConfusionMatrixDisplay
```

In [3]:
```
credit = pd.read_csv('creditcard.csv')
credit.head()
```

Out[3]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.

5 rows × 31 columns

In [4]:
```
credit.isna().sum()
```

Out[4]:
```
Time      0
V1        0
V2        0
V3        0
V4        0
V5        0
V6        0
V7        0
V8        0
V9        0
V10       0
V11       0
V12       0
V13       0
V14       0
V15       0
V16       0
V17       0
V18       0
V19       0
V20       0
V21       0
V22       0
V23       0
V24       0
V25       0
V26       0
V27       0
V28       0
Amount    0
Class     0
dtype: int64
```

In [5]:
```
credit['Class'].value_counts()
```

Out[5]:
```
0    284315
1       492
Name: Class, dtype: int64
```

In [6]:
```
X = credit.drop(['Class'], 1).to_numpy()
y = credit['Class'].to_numpy()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
```

```
C:\Users\Seneca\AppData\Local\Temp/ipykernel_18096/4172792951.py:1: FutureWarning: In a future version of pandas all arguments of
DataFrame.drop except for the argument 'labels' will be keyword-only
  X = credit.drop(['Class'], 1).to_numpy()
```

In [7]:

```
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.fit_transform(X_test)
```

In [8]:

```
def run_sampler_batch(run_model, samplers = [], incROC = False):
    metrics = []
    conf_disps = []
    rocs = []

    # make sure there are samplers to run
    if(len(samplers) > 0):
        for sampler, name in samplers:
            # get samples

            X_train_re, y_train_re = sampler.fit_resample(X_train_s, y_train)

            # run model
            if(incROC):
                mets, conf_disp, roc = run_model(X_train_re, y_train_re, name)
                rocs.append(roc)
            else:
                mets, conf_disp = run_model(X_train_re, y_train_re, name)

            # store performance stats in lists
            metrics.append(mets)
            conf_disps.append(conf_disp)

    else: # run model with original scaled training data
        if(incROC):
            mets, conf_disp, roc = run_model(X_train_s, y_train, "No Sampling Method")
            rocs.append(roc)
        else:
            mets, conf_disp = run_model(X_train_s, y_train, "No Sampling Method")

        # store performance stats in lists
        metrics.append(mets)
        conf_disps.append(conf_disp)

    # return performance stats
    if(incROC):
        return metrics, conf_disps, rocs
    return metrics, conf_disps
```

In [9]:

```
def print_metrics(metrics):
    for item in metrics:
        print("*****************************************")
        print(item["name"])
        print("Accuracy Score: ", item["accuracy"])
        print("Precision: ", item["precision"])
        print("Recall: ", item["recall"])
        print("F1: ", item["f1"])
        print()

def plot_conf_disps(conf_disps):
    for item in conf_disps:
        print(item["name"])
        item["disp"].plot()
        plt.show()

def plot_rocs(rocs):
    for item in rocs:
        plt.plot(item["fpr"], item["tpr"])
        plt.title(item["name"])
        plt.xlabel("False Positive Rate")
        plt.ylabel("True Positive Rate (Recall)")
        plt.show()
```

Logistic Regression

In [10]:

```
def run_log_reg(X_tr_nn, y_tr_nn, name):
    print("RUNNING LOGISTIC REGRESSION CLASSIFIER ON DATA: ", name)
    log_reg = LogisticRegression(solver='lbfgs', random_state=42, max_iter=1000)

    log_reg = log_reg.fit(X_tr_nn, y_tr_nn)

    y_pred = log_reg.predict(X_test_s)
    # y_pred = np.where(y_pred > 0.5, 1, 0)

    # get metrics to be printed later
    metrics_rf = {"name": name,
               "accuracy": accuracy_score(y_test, y_pred),
               "precision": precision_score(y_test, y_pred, zero_division=False),
               "recall": recall_score(y_test, y_pred),
               "f1": f1_score(y_test, y_pred),
              }

    # generate confusion matrix visualizations
```

```
    conf_matrix = confusion_matrix(y_test, y_pred)
    conf_disp_rf = {"name": name,
                    "disp": ConfusionMatrixDisplay(conf_matrix)
                    }

    # get false and true positive rates to graph later

    y_scores = cross_val_predict(log_reg, X_test_s, y_test, cv=3, method="decision_function")
    fpr, tpr, thresholds_roc = roc_curve(y_test, y_scores)
    roc_rf = {"name": name,
              "fpr": fpr,
              "tpr": tpr
             }

    return metrics_rf, conf_disp_rf, roc_rf
```

Logistic regression without accounting for class imbalances

In [11]:
```
metrics, conf_disps, rocs = run_sampler_batch(run_model=run_log_reg, incROC=True)
```

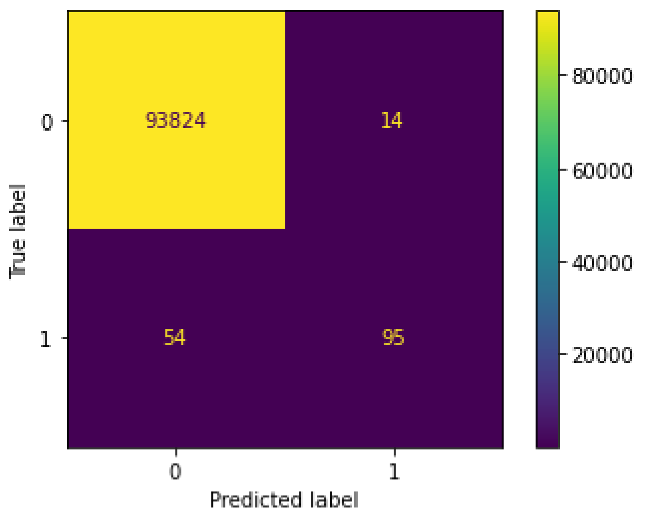
```
RUNNING LOGISTIC REGRESSION CLASSIFIER ON DATA:  No Sampling Method
```

In [12]:
```
print_metrics(metrics)
```

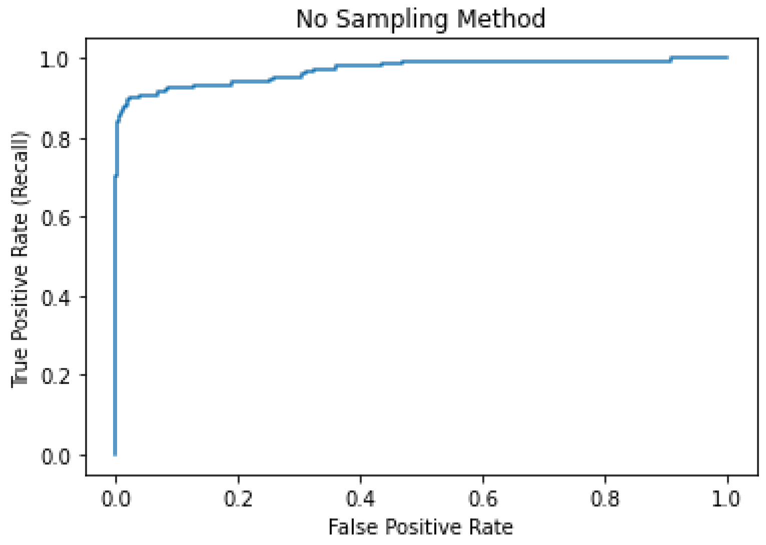
```
*****************************************
No Sampling Method
Accuracy Score:  0.9992764956855735
Precision:  0.8715596330275229
Recall:  0.6375838926174496
F1:  0.7364341085271318
```

In [13]:
```
plot_conf_disps(conf_disps)
```

```
No Sampling Method
```



In [14]:
```
plot_rocs(rocs)
```



Logistic Regression with Oversampling

In [15]:
```
over_samplers = [[RandomOverSampler(random_state=42), "RandomOverSampler"],
                 [SMOTE(random_state=42), "SMOTE"],
                 [ADASYN(random_state=42), "ADASYN"],
                 [BorderlineSMOTE(random_state=42), "BorderLineSMOTE"]
                ]
```

In [16]:
```
metrics, conf_disps, rocs = run_sampler_batch(run_model=run_log_reg, samplers=over_samplers, incROC=True)
```

```
RUNNING LOGISTIC REGRESSION CLASSIFIER ON DATA:  RandomOverSampler
RUNNING LOGISTIC REGRESSION CLASSIFIER ON DATA:  SMOTE
```

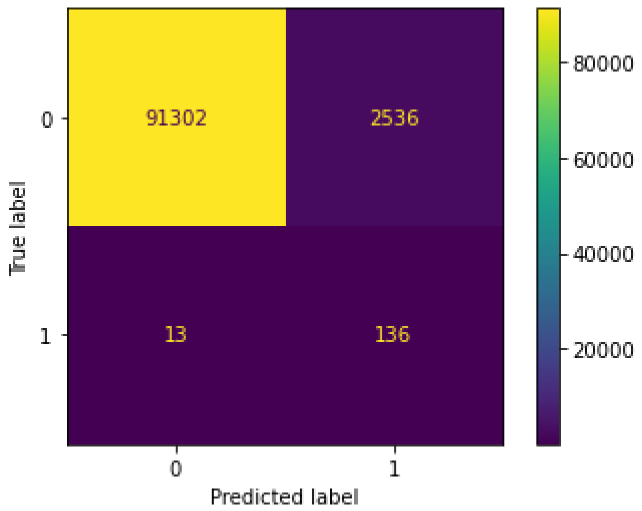
```
RUNNING LOGISTIC REGRESSION CLASSIFIER ON DATA:  ADASYN
RUNNING LOGISTIC REGRESSION CLASSIFIER ON DATA:  BorderLineSMOTE
```

In [17]:
```
print_metrics(metrics)
```

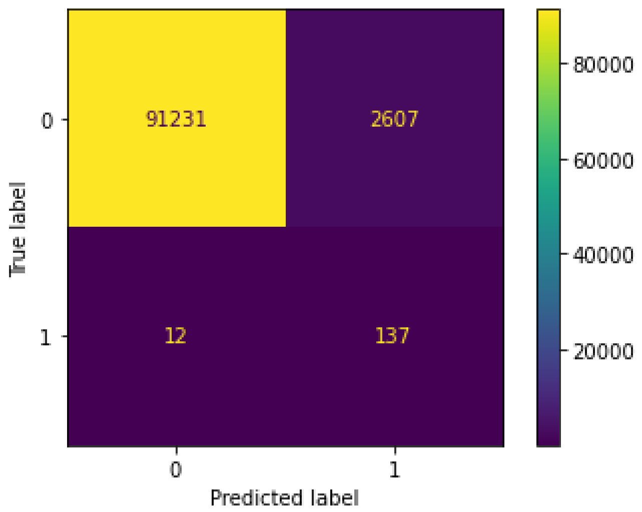
```
*****************************************
RandomOverSampler
Accuracy Score:  0.9728792279783375
Precision:  0.05089820359281437
Recall:  0.912751677852349
F1:  0.09641970932293513

*****************************************
SMOTE
Accuracy Score:  0.9721344441252514
Precision:  0.049927113702623906
Recall:  0.9194630872483222
F1:  0.0947113722779122

*****************************************
ADASYN
Accuracy Score:  0.9053805313500803
Precision:  0.015621537779747396
Recall:  0.9463087248322147
F1:  0.030735694822888286

*****************************************
BorderLineSMOTE
Accuracy Score:  0.9909774756083288
Precision:  0.13403141361256546
Recall:  0.8590604026845637
F1:  0.23188405797101452
```

In [18]:
```
plot_conf_disps(conf_disps)
```

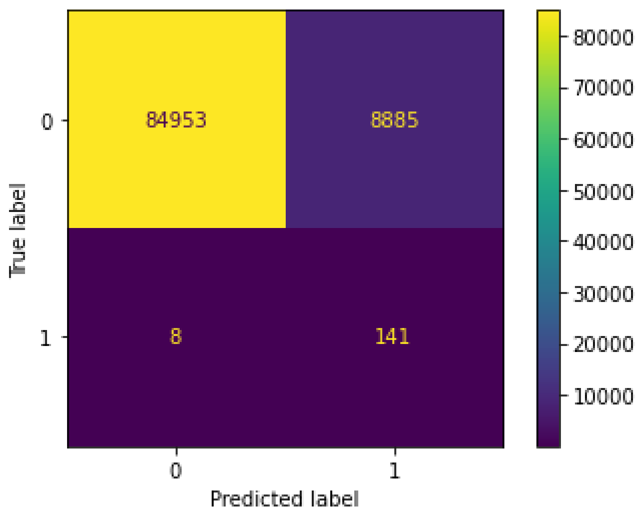
RandomOverSampler



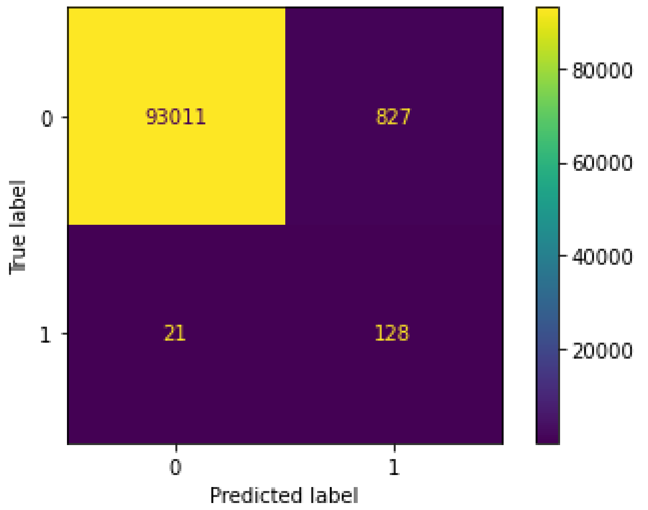
SMOTE



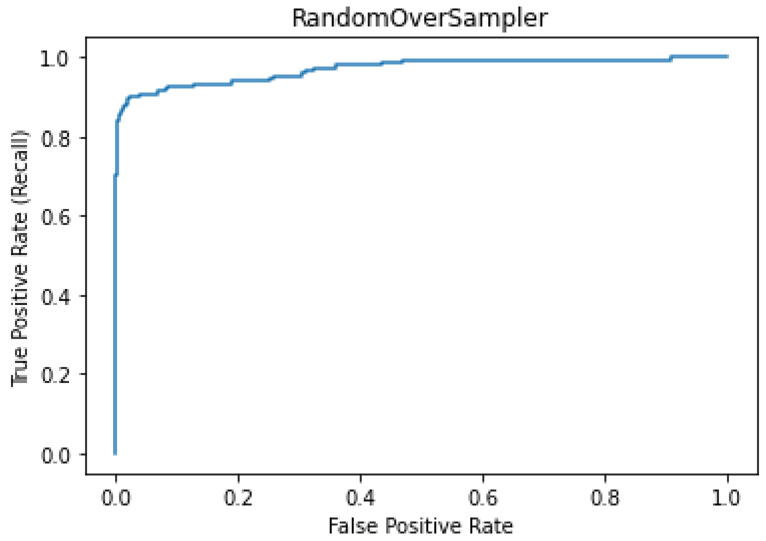
ADASYN

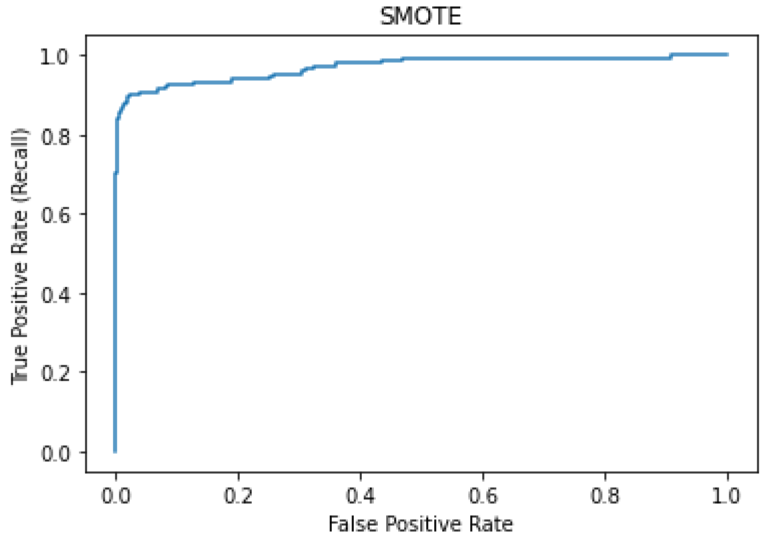


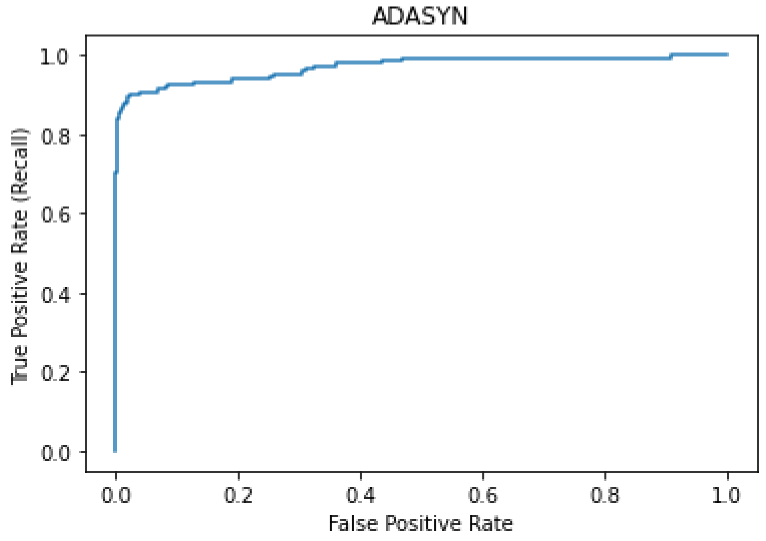
BorderLineSMOTE

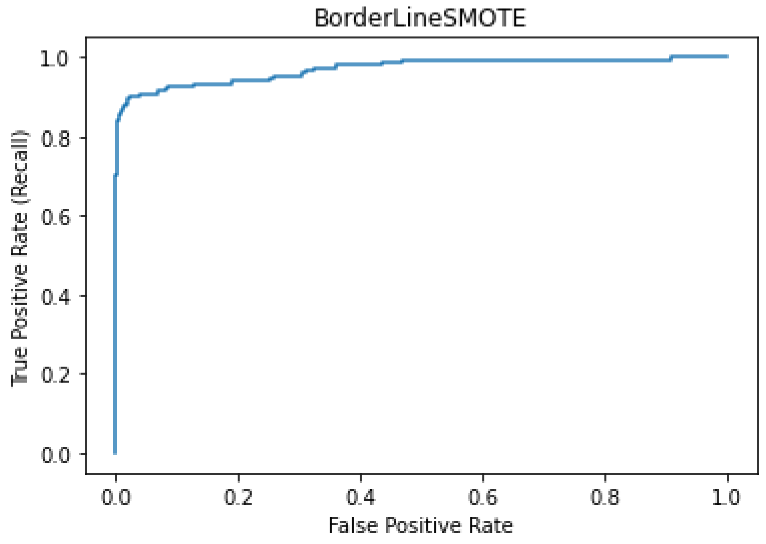


In [19]:
```
plot_rocs(rocs)
```









Logistic Regression with Undersampling

In [20]:
```
under_samplers = [[RandomUnderSampler(random_state=42), "RandomUnderSampler"],
                  [NearMiss(), "NearMiss"]
                 ]
```

In [21]:
```
metrics, conf_disps, rocs = run_sampler_batch(run_model=run_log_reg, samplers=under_samplers, incROC=True)
```

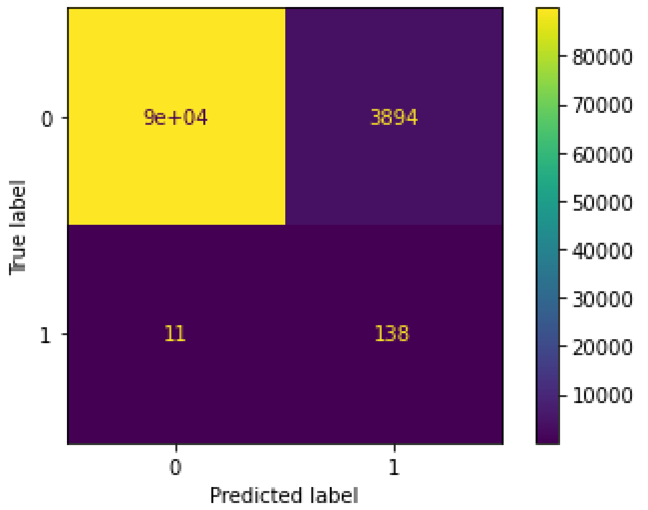
```
RUNNING LOGISTIC REGRESSION CLASSIFIER ON DATA:  RandomUnderSampler
RUNNING LOGISTIC REGRESSION CLASSIFIER ON DATA:  NearMiss
```

In [22]:
```
print_metrics(metrics)
```

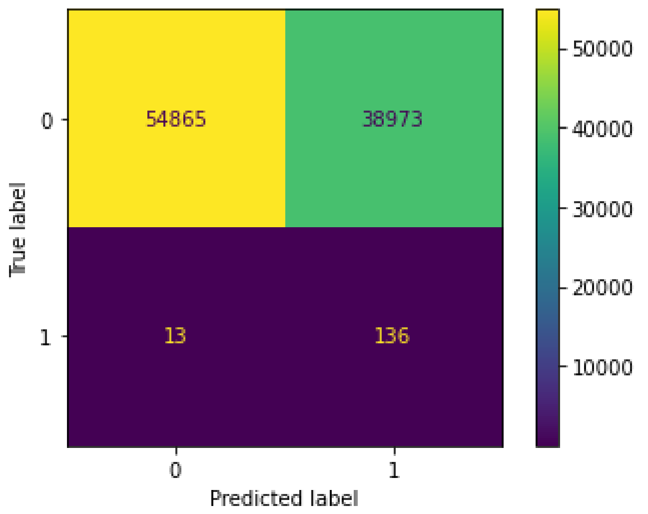
```
*****************************************
RandomUnderSampler
Accuracy Score:  0.9584517007671274
Precision:  0.03422619047619048
Recall:  0.9261744966442953
F1:  0.06601291557043769

*****************************************
NearMiss
Accuracy Score:  0.585197952908381
Precision:  0.0034774604311028153
Recall:  0.912751677852349
F1:  0.006928524122471852
```

In [23]:
```
plot_conf_disps(conf_disps)
```

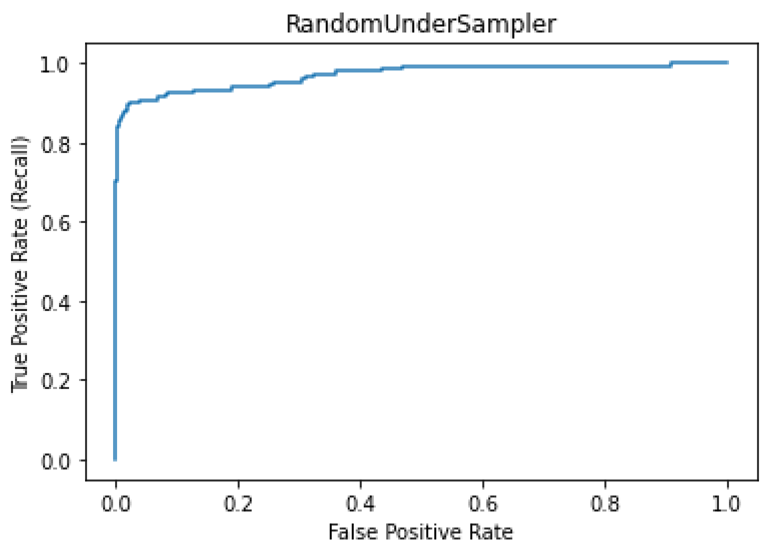
RandomUnderSampler

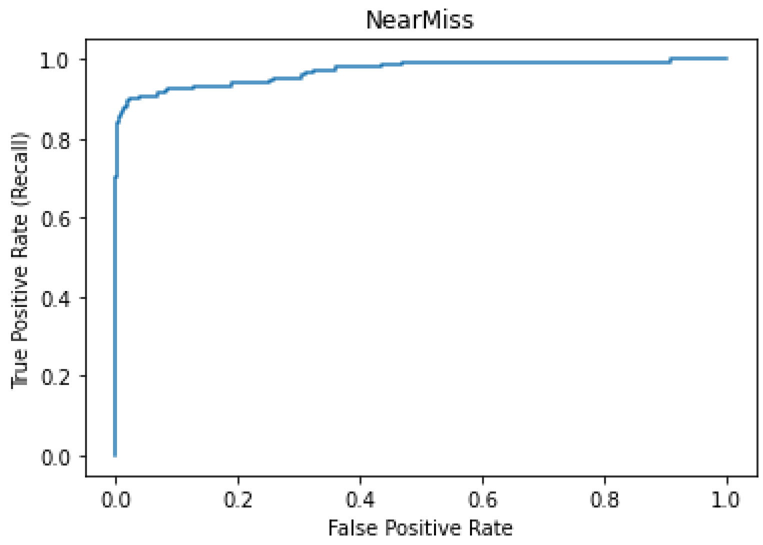


NearMiss



In [24]:
```
plot_rocs(rocs)
```





Neural Network

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

In [26]:
```
np.shape(X_train_s[0])
```

Out[26]:
```
(30,)
```

In [27]:
```
def run_nn(X_tr_nn, y_tr_nn, name):
    print("RUNNING NEURAL NETWORK ON DATA: ", name)
    classifier = Sequential()
    classifier.add(keras.layers.Dense(units=60, kernel_initializer='uniform', activation='tanh', input_dim=30))
    classifier.add(keras.layers.Dense(units=30, kernel_initializer='uniform', activation='tanh'))
    classifier.add(keras.layers.Dense(units=10, kernel_initializer='uniform', activation='tanh'))
    classifier.add(keras.layers.Dense(units=1, kernel_initializer='uniform', activation='sigmoid'))

    classifier.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

    history = classifier.fit(X_tr_nn, y_tr_nn, epochs=20, batch_size=10)

    y_pred = classifier.predict(X_test_s)
    y_pred = np.where(y_pred > 0.5, 1, 0)

    # get metrics to be printed later
    metrics_nn = {"name": name,
                "accuracy": accuracy_score(y_test, y_pred),
                "precision": precision_score(y_test, y_pred, zero_division=False),
                "recall": recall_score(y_test, y_pred),
                "f1": f1_score(y_test, y_pred),
              }

    # generate confusion matrix visualizations
    conf_matrix = confusion_matrix(y_test, y_pred)
    conf_disp_nn = {"name": name,
                  "disp": ConfusionMatrixDisplay(conf_matrix)
                   }

    return metrics_nn, conf_disp_nn
```

Neural Network without accounting for class imbalance

In [28]:
```
metrics, conf_disps = run_sampler_batch(run_nn)
```

```
RUNNING NEURAL NETWORK ON DATA:  No Sampling Method
Epoch 1/20
19082/19082 [==============================] - 20s 1ms/step - loss: 0.0091 - accuracy: 0.9991
Epoch 2/20
19082/19082 [==============================] - 20s 1ms/step - loss: 0.0040 - accuracy: 0.9993
Epoch 3/20
19082/19082 [==============================] - 21s 1ms/step - loss: 0.0035 - accuracy: 0.9994
Epoch 4/20
19082/19082 [==============================] - 21s 1ms/step - loss: 0.0034 - accuracy: 0.9993
Epoch 5/20
19082/19082 [==============================] - 22s 1ms/step - loss: 0.0032 - accuracy: 0.9993
Epoch 6/20
19082/19082 [==============================] - 23s 1ms/step - loss: 0.0031 - accuracy: 0.9993
Epoch 7/20
19082/19082 [==============================] - 24s 1ms/step - loss: 0.0030 - accuracy: 0.9993
Epoch 8/20
19082/19082 [==============================] - 23s 1ms/step - loss: 0.0028 - accuracy: 0.9994
Epoch 9/20
19082/19082 [==============================] - 23s 1ms/step - loss: 0.0027 - accuracy: 0.9994
Epoch 10/20
19082/19082 [==============================] - 27s 1ms/step - loss: 0.0026 - accuracy: 0.9994
Epoch 11/20
19082/19082 [==============================] - 21s 1ms/step - loss: 0.0025 - accuracy: 0.9994
```

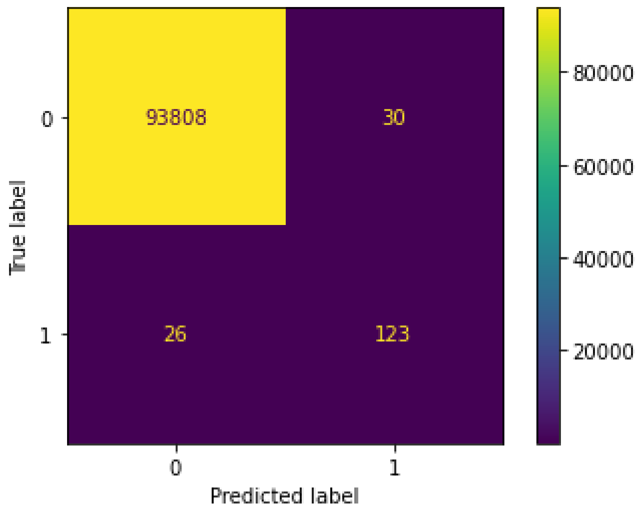
```
Epoch 12/20
19082/19082 [==============================] - 22s 1ms/step - loss: 0.0025 - accuracy: 0.9994
Epoch 13/20
19082/19082 [==============================] - 21s 1ms/step - loss: 0.0022 - accuracy: 0.9995
Epoch 14/20
19082/19082 [==============================] - 21s 1ms/step - loss: 0.0023 - accuracy: 0.9995
Epoch 15/20
19082/19082 [==============================] - 21s 1ms/step - loss: 0.0021 - accuracy: 0.9995
Epoch 16/20
19082/19082 [==============================] - 21s 1ms/step - loss: 0.0022 - accuracy: 0.9995
Epoch 17/20
19082/19082 [==============================] - 21s 1ms/step - loss: 0.0021 - accuracy: 0.9996
Epoch 18/20
19082/19082 [==============================] - 21s 1ms/step - loss: 0.0019 - accuracy: 0.9995
Epoch 19/20
19082/19082 [==============================] - 21s 1ms/step - loss: 0.0018 - accuracy: 0.9995
Epoch 20/20
19082/19082 [==============================] - 21s 1ms/step - loss: 0.0018 - accuracy: 0.9995
```

In [29]:
```
print_metrics(metrics)
```

```
******************************************
No Sampling Method
Accuracy Score:  0.9994041729175311
Precision:  0.803921568627451
Recall:  0.825503355704698
F1:  0.8145695364238411
```

In [30]:
```
plot_conf_disps(conf_disps)
```

```
No Sampling Method
```



Neural Network with Oversampling

In [31]:
```
over_samplers = [[RandomOverSampler(random_state=42), "RandomOverSampler"],
                 [SMOTE(random_state=42), "SMOTE"],
                 [ADASYN(random_state=42), "ADASYN"],
                 [BorderlineSMOTE(random_state=42), "BorderLineSMOTE"]
                ]
```

In [32]:
```
metrics, conf_disps = run_sampler_batch(run_nn, over_samplers)
```

```
RUNNING NEURAL NETWORK ON DATA:  RandomOverSampler
Epoch 1/20
38096/38096 [==============================] - 52s 1ms/step - loss: 0.0476 - accuracy: 0.9835
Epoch 2/20
38096/38096 [==============================] - 57s 1ms/step - loss: 0.0080 - accuracy: 0.9978
Epoch 3/20
38096/38096 [==============================] - 49s 1ms/step - loss: 0.0051 - accuracy: 0.9987
Epoch 4/20
38096/38096 [==============================] - 42s 1ms/step - loss: 0.0037 - accuracy: 0.9992
Epoch 5/20
38096/38096 [==============================] - 42s 1ms/step - loss: 0.0030 - accuracy: 0.9994
Epoch 6/20
38096/38096 [==============================] - 43s 1ms/step - loss: 0.0027 - accuracy: 0.9995
Epoch 7/20
38096/38096 [==============================] - 44s 1ms/step - loss: 0.0027 - accuracy: 0.9995
Epoch 8/20
38096/38096 [==============================] - 44s 1ms/step - loss: 0.0021 - accuracy: 0.9996
Epoch 9/20
38096/38096 [==============================] - 45s 1ms/step - loss: 0.0020 - accuracy: 0.9996
Epoch 10/20
38096/38096 [==============================] - 45s 1ms/step - loss: 0.0018 - accuracy: 0.9997
Epoch 11/20
38096/38096 [==============================] - 44s 1ms/step - loss: 0.0017 - accuracy: 0.9997
Epoch 12/20
38096/38096 [==============================] - 42s 1ms/step - loss: 0.0016 - accuracy: 0.9997
Epoch 13/20
38096/38096 [==============================] - 42s 1ms/step - loss: 0.0018 - accuracy: 0.9997
Epoch 14/20
38096/38096 [==============================] - 43s 1ms/step - loss: 0.0015 - accuracy: 0.9997
```

```
Epoch 15/20
38096/38096 [==============================] - 44s 1ms/step - loss: 0.0014 - accuracy: 0.9997
Epoch 16/20
38096/38096 [==============================] - 45s 1ms/step - loss: 0.0014 - accuracy: 0.9998
Epoch 17/20
38096/38096 [==============================] - 45s 1ms/step - loss: 0.0015 - accuracy: 0.9997
Epoch 18/20
38096/38096 [==============================] - 45s 1ms/step - loss: 0.0015 - accuracy: 0.9997
Epoch 19/20
38096/38096 [==============================] - 45s 1ms/step - loss: 0.0014 - accuracy: 0.9997
Epoch 20/20
38096/38096 [==============================] - 47s 1ms/step - loss: 0.0017 - accuracy: 0.9997
RUNNING NEURAL NETWORK ON DATA:  SMOTE
Epoch 1/20
38096/38096 [==============================] - 45s 1ms/step - loss: 0.0507 - accuracy: 0.9819
Epoch 2/20
38096/38096 [==============================] - 45s 1ms/step - loss: 0.0109 - accuracy: 0.9968
Epoch 3/20
38096/38096 [==============================] - 44s 1ms/step - loss: 0.0067 - accuracy: 0.9981
Epoch 4/20
38096/38096 [==============================] - 45s 1ms/step - loss: 0.0048 - accuracy: 0.9988
Epoch 5/20
38096/38096 [==============================] - 44s 1ms/step - loss: 0.0038 - accuracy: 0.9990
Epoch 6/20
38096/38096 [==============================] - 45s 1ms/step - loss: 0.0032 - accuracy: 0.9992
Epoch 7/20
38096/38096 [==============================] - 44s 1ms/step - loss: 0.0030 - accuracy: 0.9993
Epoch 8/20
38096/38096 [==============================] - 45s 1ms/step - loss: 0.0024 - accuracy: 0.9994
Epoch 9/20
38096/38096 [==============================] - 45s 1ms/step - loss: 0.0021 - accuracy: 0.9995
Epoch 10/20
38096/38096 [==============================] - 44s 1ms/step - loss: 0.0020 - accuracy: 0.9995
Epoch 11/20
38096/38096 [==============================] - 42s 1ms/step - loss: 0.0020 - accuracy: 0.9995
Epoch 12/20
38096/38096 [==============================] - 42s 1ms/step - loss: 0.0017 - accuracy: 0.9996
Epoch 13/20
38096/38096 [==============================] - 41s 1ms/step - loss: 0.0015 - accuracy: 0.9996
Epoch 14/20
38096/38096 [==============================] - 41s 1ms/step - loss: 0.0015 - accuracy: 0.9997
Epoch 15/20
38096/38096 [==============================] - 41s 1ms/step - loss: 0.0015 - accuracy: 0.9996
Epoch 16/20
38096/38096 [==============================] - 42s 1ms/step - loss: 0.0013 - accuracy: 0.9997
Epoch 17/20
38096/38096 [==============================] - 42s 1ms/step - loss: 0.0013 - accuracy: 0.9997
Epoch 18/20
38096/38096 [==============================] - 42s 1ms/step - loss: 0.0012 - accuracy: 0.9997
Epoch 19/20
38096/38096 [==============================] - 41s 1ms/step - loss: 0.0013 - accuracy: 0.9997
Epoch 20/20
38096/38096 [==============================] - 40s 1ms/step - loss: 0.0012 - accuracy: 0.9997
RUNNING NEURAL NETWORK ON DATA:  ADASYN
Epoch 1/20
38096/38096 [==============================] - 39s 1ms/step - loss: 0.0588 - accuracy: 0.9797
Epoch 2/20
38096/38096 [==============================] - 42s 1ms/step - loss: 0.0099 - accuracy: 0.9975
Epoch 3/20
38096/38096 [==============================] - 42s 1ms/step - loss: 0.0066 - accuracy: 0.9984
Epoch 4/20
38096/38096 [==============================] - 40s 1ms/step - loss: 0.0052 - accuracy: 0.9988
Epoch 5/20
38096/38096 [==============================] - 41s 1ms/step - loss: 0.0041 - accuracy: 0.9991
Epoch 6/20
38096/38096 [==============================] - 41s 1ms/step - loss: 0.0031 - accuracy: 0.9993
Epoch 7/20
38096/38096 [==============================] - 40s 1ms/step - loss: 0.0030 - accuracy: 0.9994
Epoch 8/20
38096/38096 [==============================] - 40s 1ms/step - loss: 0.0027 - accuracy: 0.9995
Epoch 9/20
38096/38096 [==============================] - 40s 1ms/step - loss: 0.0022 - accuracy: 0.9995
Epoch 10/20
38096/38096 [==============================] - 39s 1ms/step - loss: 0.0021 - accuracy: 0.9995
Epoch 11/20
38096/38096 [==============================] - 38s 1ms/step - loss: 0.0020 - accuracy: 0.9996
Epoch 12/20
38096/38096 [==============================] - 38s 990us/step - loss: 0.0022 - accuracy: 0.9996
Epoch 13/20
38096/38096 [==============================] - 38s 995us/step - loss: 0.0018 - accuracy: 0.9996
Epoch 14/20
38096/38096 [==============================] - 45s 1ms/step - loss: 0.0016 - accuracy: 0.9997
Epoch 15/20
38096/38096 [==============================] - 47s 1ms/step - loss: 0.0016 - accuracy: 0.9997
Epoch 16/20
38096/38096 [==============================] - 48s 1ms/step - loss: 0.0016 - accuracy: 0.9997
Epoch 17/20
38096/38096 [==============================] - 46s 1ms/step - loss: 0.0016 - accuracy: 0.9997
Epoch 18/20
38096/38096 [==============================] - 46s 1ms/step - loss: 0.0017 - accuracy: 0.9997
Epoch 19/20
38096/38096 [==============================] - 47s 1ms/step - loss: 0.0016 - accuracy: 0.9997
```

```
Epoch 20/20
38096/38096 [==============================] - 48s 1ms/step - loss: 0.0015 - accuracy: 0.9997
RUNNING NEURAL NETWORK ON DATA:  BorderLineSMOTE
Epoch 1/20
38096/38096 [==============================] - 51s 1ms/step - loss: 0.0170 - accuracy: 0.9957
Epoch 2/20
38096/38096 [==============================] - 51s 1ms/step - loss: 0.0052 - accuracy: 0.9989
Epoch 3/20
38096/38096 [==============================] - 53s 1ms/step - loss: 0.0038 - accuracy: 0.9993
Epoch 4/20
38096/38096 [==============================] - 51s 1ms/step - loss: 0.0032 - accuracy: 0.9994
Epoch 5/20
38096/38096 [==============================] - 51s 1ms/step - loss: 0.0028 - accuracy: 0.9995
Epoch 6/20
38096/38096 [==============================] - 51s 1ms/step - loss: 0.0024 - accuracy: 0.9996
Epoch 7/20
38096/38096 [==============================] - 51s 1ms/step - loss: 0.0024 - accuracy: 0.9996
Epoch 8/20
38096/38096 [==============================] - 53s 1ms/step - loss: 0.0022 - accuracy: 0.9997
Epoch 9/20
38096/38096 [==============================] - 51s 1ms/step - loss: 0.0020 - accuracy: 0.9997
Epoch 10/20
38096/38096 [==============================] - 51s 1ms/step - loss: 0.0018 - accuracy: 0.9997
Epoch 11/20
38096/38096 [==============================] - 52s 1ms/step - loss: 0.0017 - accuracy: 0.9997
Epoch 12/20
38096/38096 [==============================] - 49s 1ms/step - loss: 0.0016 - accuracy: 0.9997
Epoch 13/20
38096/38096 [==============================] - 51s 1ms/step - loss: 0.0017 - accuracy: 0.9997
Epoch 14/20
38096/38096 [==============================] - 49s 1ms/step - loss: 0.0016 - accuracy: 0.9998
Epoch 15/20
38096/38096 [==============================] - 45s 1ms/step - loss: 0.0017 - accuracy: 0.9997
Epoch 16/20
38096/38096 [==============================] - 47s 1ms/step - loss: 0.0014 - accuracy: 0.9998
Epoch 17/20
38096/38096 [==============================] - 47s 1ms/step - loss: 0.0013 - accuracy: 0.9998
Epoch 18/20
38096/38096 [==============================] - 46s 1ms/step - loss: 0.0015 - accuracy: 0.9998
Epoch 19/20
38096/38096 [==============================] - 47s 1ms/step - loss: 0.0013 - accuracy: 0.9998
Epoch 20/20
38096/38096 [==============================] - 46s 1ms/step - loss: 0.0012 - accuracy: 0.9998
```

In [33]:

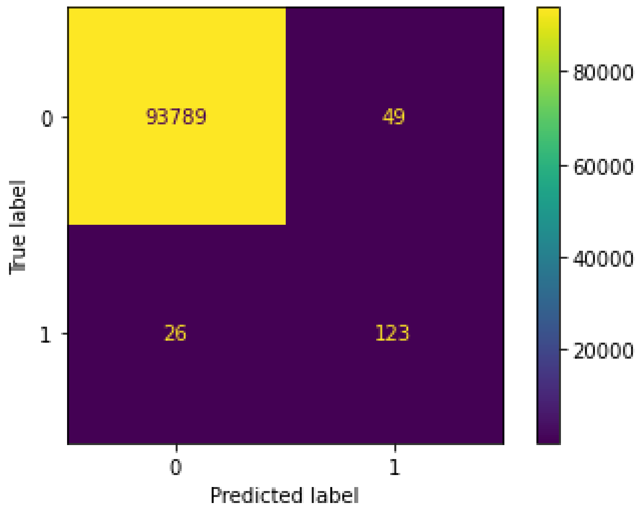
```
print_metrics(metrics)
```

```
*****************************************
RandomOverSampler
Accuracy Score:  0.9992020173002649
Precision:  0.7151162790697675
Recall:  0.825503355704698
F1:  0.7663551401869161

*****************************************
SMOTE
Accuracy Score:  0.9984572334471788
Precision:  0.5078740157480315
Recall:  0.8657718120805369
F1:  0.6401985111662531

*****************************************
ADASYN
Accuracy Score:  0.9989466628363497
Precision:  0.631578947368421
Recall:  0.8053691275167785
F1:  0.7079646017699115

*****************************************
BorderLineSMOTE
Accuracy Score:  0.9992764956855735
Precision:  0.7755102040816326
Recall:  0.7651006711409396
F1:  0.7702702702702703
```

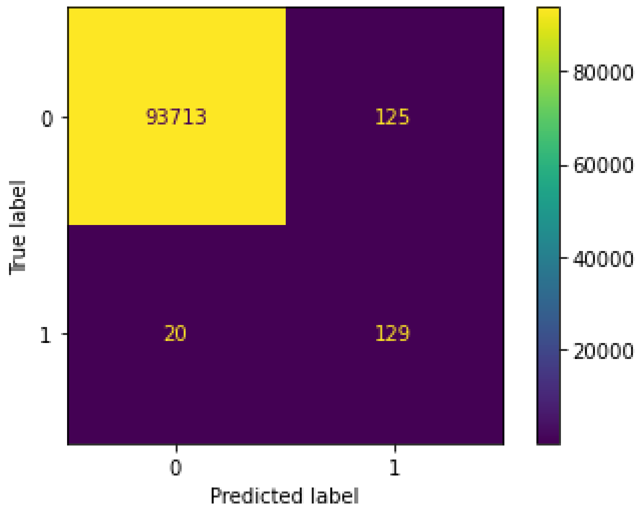
In [34]:

```
plot_conf_disps(conf_disps)
```

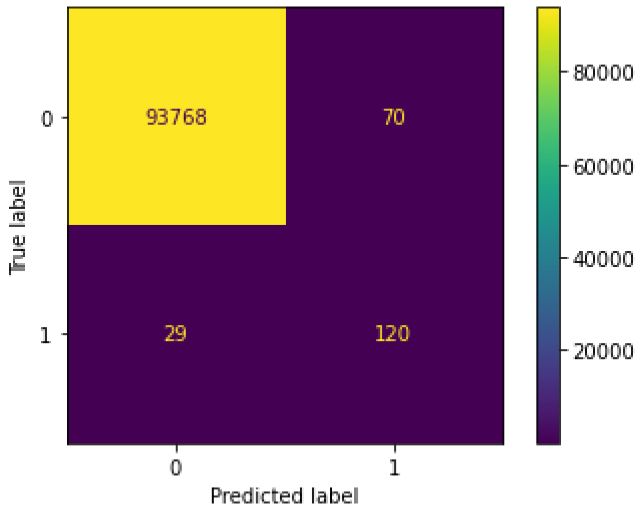
```
RandomOverSampler
```



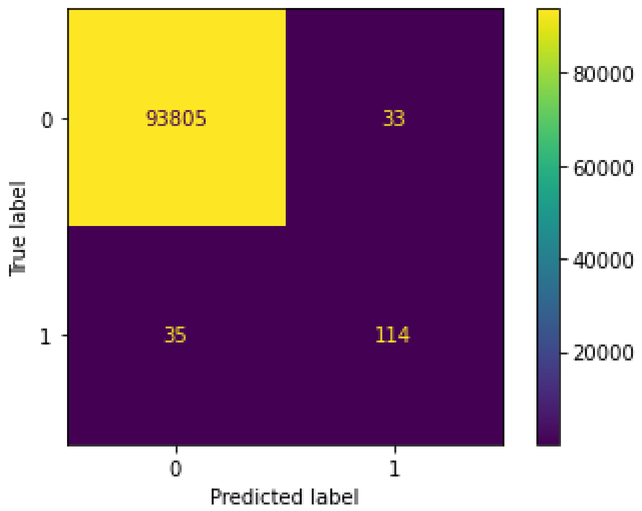
SMOTE



ADASYN



BorderLineSMOTE



Neural Network with Undersampling

In [35]:

```
under_samplers = [[RandomUnderSampler(random_state=42), "RandomUnderSampler"],
                  [NearMiss(), "NearMiss"]
                 ]
```

In [36]:

```
metrics, conf_disps = run_sampler_batch(run_nn, under_samplers)
```

```
RUNNING NEURAL NETWORK ON DATA:  RandomUnderSampler
Epoch 1/20
69/69 [==============================] - 0s 854us/step - loss: 0.5887 - accuracy: 0.8630
Epoch 2/20
69/69 [==============================] - 0s 1ms/step - loss: 0.3145 - accuracy: 0.9344
Epoch 3/20
69/69 [==============================] - 0s 1ms/step - loss: 0.2301 - accuracy: 0.9402
Epoch 4/20
69/69 [==============================] - 0s 1ms/step - loss: 0.1945 - accuracy: 0.9461
Epoch 5/20
69/69 [==============================] - 0s 1ms/step - loss: 0.1725 - accuracy: 0.9577
Epoch 6/20
69/69 [==============================] - 0s 1ms/step - loss: 0.1654 - accuracy: 0.9548
```

```
Epoch 7/20
69/69 [==============================] - 0s 1ms/step - loss: 0.1586 - accuracy: 0.9563
Epoch 8/20
69/69 [==============================] - 0s 1ms/step - loss: 0.1538 - accuracy: 0.9548
Epoch 9/20
69/69 [==============================] - 0s 1ms/step - loss: 0.1458 - accuracy: 0.9534
Epoch 10/20
69/69 [==============================] - 0s 1ms/step - loss: 0.1455 - accuracy: 0.9563
Epoch 11/20
69/69 [==============================] - 0s 1ms/step - loss: 0.1430 - accuracy: 0.9548
Epoch 12/20
69/69 [==============================] - 0s 1ms/step - loss: 0.1408 - accuracy: 0.9548
Epoch 13/20
69/69 [==============================] - 0s 1ms/step - loss: 0.1411 - accuracy: 0.9548
Epoch 14/20
69/69 [==============================] - 0s 2ms/step - loss: 0.1332 - accuracy: 0.9563
Epoch 15/20
69/69 [==============================] - 0s 2ms/step - loss: 0.1341 - accuracy: 0.9592
Epoch 16/20
69/69 [==============================] - 0s 2ms/step - loss: 0.1315 - accuracy: 0.9577
Epoch 17/20
69/69 [==============================] - 0s 1ms/step - loss: 0.1284 - accuracy: 0.9577
Epoch 18/20
69/69 [==============================] - 0s 2ms/step - loss: 0.1249 - accuracy: 0.9636
Epoch 19/20
69/69 [==============================] - 0s 2ms/step - loss: 0.1289 - accuracy: 0.9548
Epoch 20/20
69/69 [==============================] - 0s 2ms/step - loss: 0.1266 - accuracy: 0.9592
RUNNING NEURAL NETWORK ON DATA:  NearMiss
Epoch 1/20
69/69 [==============================] - 1s 968us/step - loss: 0.5603 - accuracy: 0.8717
Epoch 2/20
69/69 [==============================] - 0s 1ms/step - loss: 0.3267 - accuracy: 0.9227
Epoch 3/20
69/69 [==============================] - 0s 1ms/step - loss: 0.2194 - accuracy: 0.9577
Epoch 4/20
69/69 [==============================] - 0s 1ms/step - loss: 0.1703 - accuracy: 0.9636
Epoch 5/20
69/69 [==============================] - 0s 1ms/step - loss: 0.1475 - accuracy: 0.9679
Epoch 6/20
69/69 [==============================] - 0s 1ms/step - loss: 0.1270 - accuracy: 0.9708
Epoch 7/20
69/69 [==============================] - 0s 1ms/step - loss: 0.1239 - accuracy: 0.9752
Epoch 8/20
69/69 [==============================] - 0s 1ms/step - loss: 0.1099 - accuracy: 0.9767
Epoch 9/20
69/69 [==============================] - 0s 1ms/step - loss: 0.1073 - accuracy: 0.9767
Epoch 10/20
69/69 [==============================] - 0s 2ms/step - loss: 0.1085 - accuracy: 0.9738
Epoch 11/20
69/69 [==============================] - 0s 2ms/step - loss: 0.1109 - accuracy: 0.9723
Epoch 12/20
69/69 [==============================] - 0s 2ms/step - loss: 0.1048 - accuracy: 0.9738
Epoch 13/20
69/69 [==============================] - 0s 1ms/step - loss: 0.0956 - accuracy: 0.9796
Epoch 14/20
69/69 [==============================] - 0s 1ms/step - loss: 0.0994 - accuracy: 0.9767
Epoch 15/20
69/69 [==============================] - 0s 1ms/step - loss: 0.0968 - accuracy: 0.9767
Epoch 16/20
69/69 [==============================] - 0s 2ms/step - loss: 0.1016 - accuracy: 0.9767
Epoch 17/20
69/69 [==============================] - 0s 2ms/step - loss: 0.1079 - accuracy: 0.9694
Epoch 18/20
69/69 [==============================] - 0s 2ms/step - loss: 0.1004 - accuracy: 0.9752
Epoch 19/20
69/69 [==============================] - 0s 1ms/step - loss: 0.0944 - accuracy: 0.9781
Epoch 20/20
69/69 [==============================] - 0s 2ms/step - loss: 0.0944 - accuracy: 0.9781
```

In [37]:

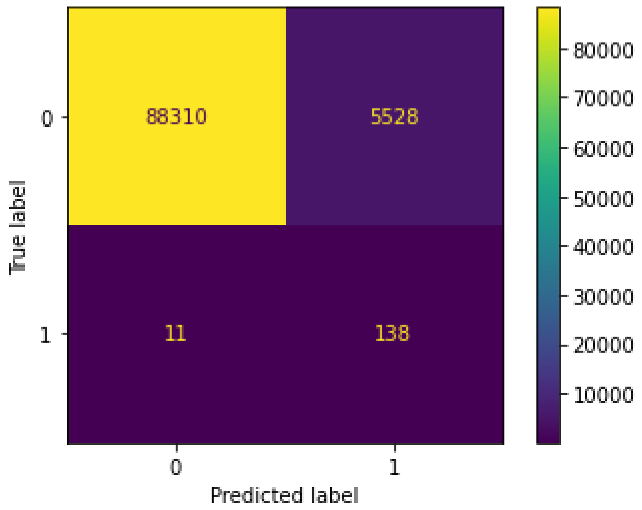
```
print_metrics(metrics)
```

```
****************************************
RandomUnderSampler
Accuracy Score:  0.9410663176822327
Precision:  0.02435580656547829
Recall:  0.9261744966442953
F1:  0.047463456577815984

****************************************
NearMiss
Accuracy Score:  0.5532254460723292
Precision:  0.003252920505271156
Recall:  0.9194630872483222
F1:  0.006482905477345322
```

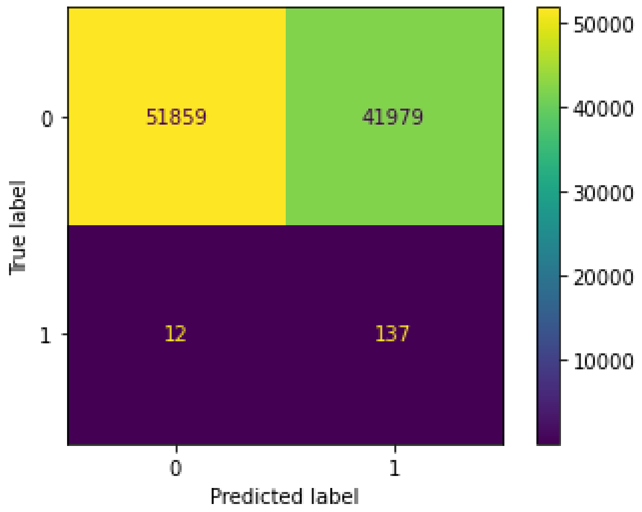
In [38]:

```
plot_conf_disps(conf_disps)
```

```
RandomUnderSampler
```



NearMiss



Random Forest Classifier

In [39]:
```
from sklearn.ensemble import RandomForestClassifier
```

In [40]:
```
def run_rfc(X_tr_nn, y_tr_nn, name):
    print("RUNNING RANDOM FOREST CLASSIFIER ON DATA: ", name)
    rfc = RandomForestClassifier(random_state=42)

    rfc.fit(X_tr_nn, y_tr_nn)

    y_pred = rfc.predict(X_test_s)
    # y_pred = np.where(y_pred > 0.5, 1, 0)

    # get metrics to be printed later
    metrics_rf = {"name": name,
                  "accuracy": accuracy_score(y_test, y_pred),
                  "precision": precision_score(y_test, y_pred, zero_division=False),
                  "recall": recall_score(y_test, y_pred),
                  "f1": f1_score(y_test, y_pred),
                 }

    # generate confusion matrix visualizations
    conf_matrix = confusion_matrix(y_test, y_pred)
    conf_disp_rf = {"name": name,
                    "disp": ConfusionMatrixDisplay(conf_matrix)
                   }

    # get false and true positive rates to graph later

    y_scores = cross_val_predict(rfc, X_test_s, y_test, cv=3)
    fpr, tpr, thresholds_roc = roc_curve(y_test, y_scores)
    roc_rf = {"name": name,
              "fpr": fpr,
              "tpr": tpr
             }

    return metrics_rf, conf_disp_rf, roc_rf
```

In [41]:
```
metrics, conf_disps, rocs = run_sampler_batch(run_model=run_rfc, incROC=True)
```

```
RUNNING RANDOM FOREST CLASSIFIER ON DATA:  No Sampling Method
```

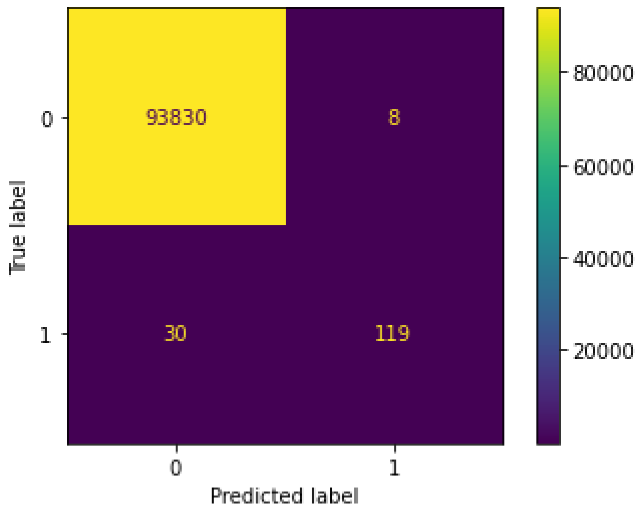
In [42]:
```
print_metrics(metrics)
```

```
*****************************************
No Sampling Method
Accuracy Score:  0.9995956887654676
Precision:  0.937007874015748
Recall:  0.7986577181208053
F1:  0.8623188405797101
```

In [43]:

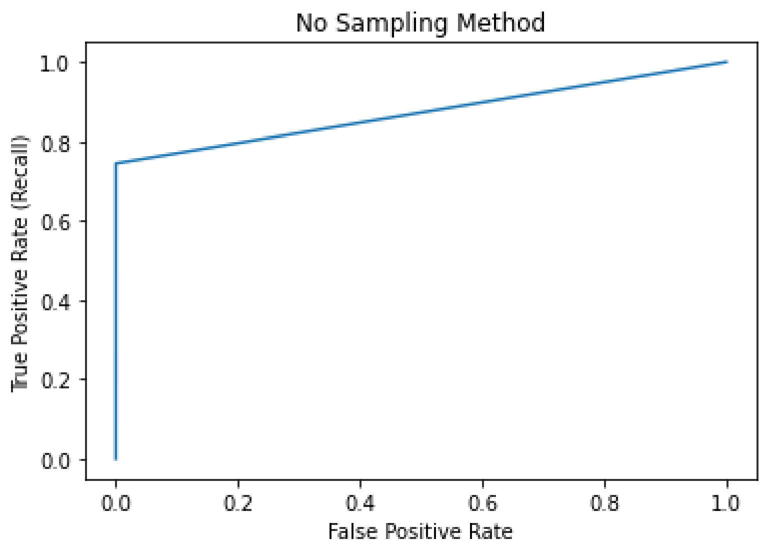
```
plot_conf_disps(conf_disps)
```

No Sampling Method



In [44]:

```
plot_rocs(rocs)
```



Random Forest Classifier with Oversampling

In [45]:

```
over_samplers = [[RandomOverSampler(random_state=42), "RandomOverSampler"],
                 [SMOTE(random_state=42), "SMOTE"],
                 [ADASYN(random_state=42), "ADASYN"],
                 [BorderlineSMOTE(random_state=42), "BorderLineSMOTE"]
                ]
```

In [46]:

```
metrics, conf_disps, rocs = run_sampler_batch(run_model=run_rfc, samplers=over_samplers, incROC=True)
```

```
RUNNING RANDOM FOREST CLASSIFIER ON DATA:  RandomOverSampler
RUNNING RANDOM FOREST CLASSIFIER ON DATA:  SMOTE
RUNNING RANDOM FOREST CLASSIFIER ON DATA:  ADASYN
RUNNING RANDOM FOREST CLASSIFIER ON DATA:  BorderLineSMOTE
```

In [47]:

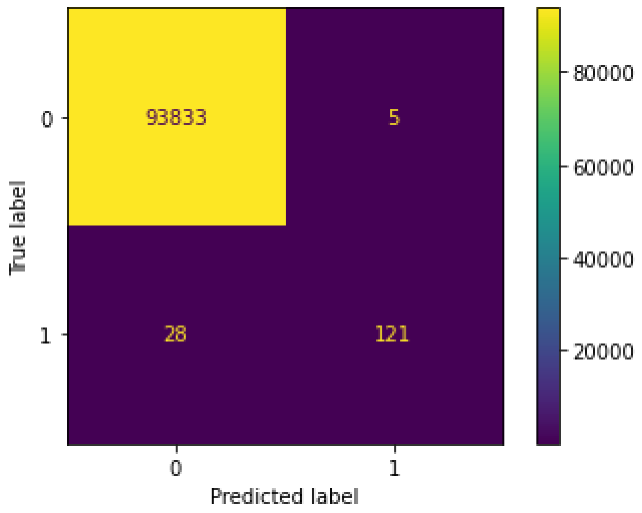
```
print_metrics(metrics)
```

```
*****************************************
RandomOverSampler
Accuracy Score:  0.9996488876121166
Precision:  0.9603174603174603
Recall:  0.8120805369127517
F1:  0.8800000000000001

*****************************************
SMOTE
Accuracy Score:  0.999574409226808
Precision:  0.8707482993197279
Recall:  0.8590604026845637
F1:  0.8648648648648649

*****************************************
ADASYN
Accuracy Score:  0.9995105706108292
Precision:  0.8551724137931035
Recall:  0.8322147651006712
F1:  0.8435374149659864

*****************************************
BorderLineSMOTE
Accuracy Score:  0.9995956887654676
Precision:  0.9302325581395349
Recall:  0.8053691275167785
F1:  0.8633093525179855
```

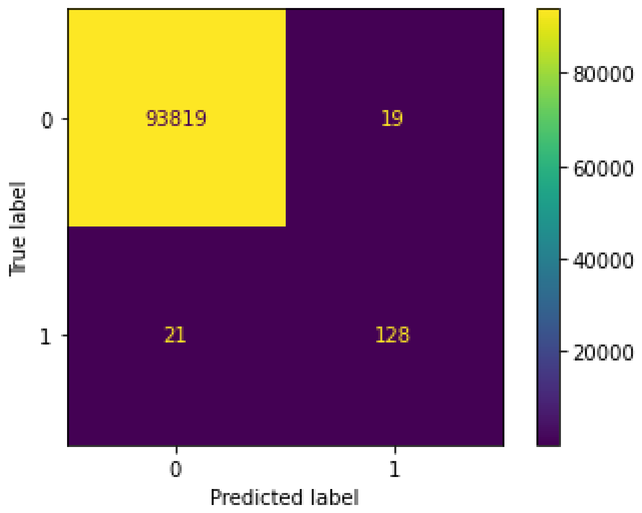
In [48]:

```
plot_conf_disps(conf_disps)
```

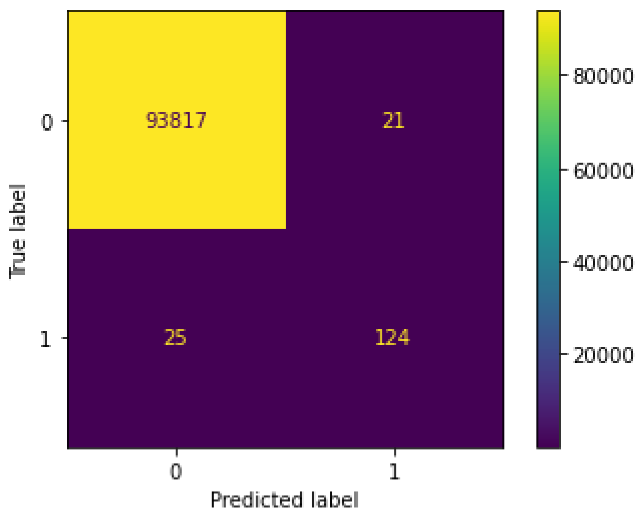
RandomOverSampler



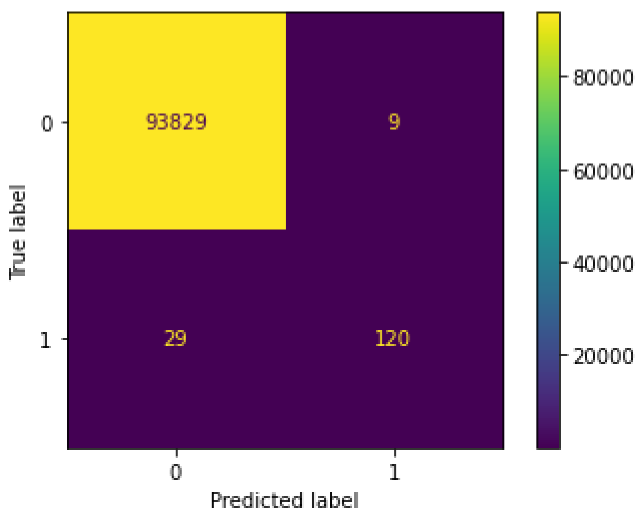
SMOTE



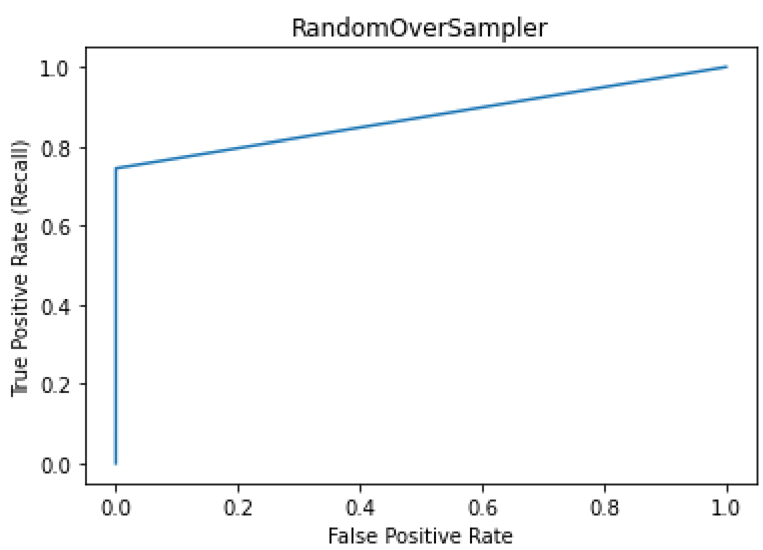
ADASYN

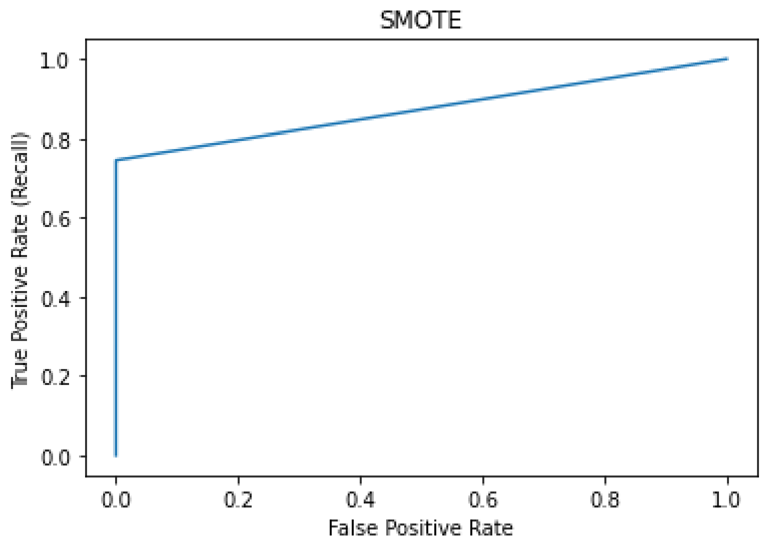


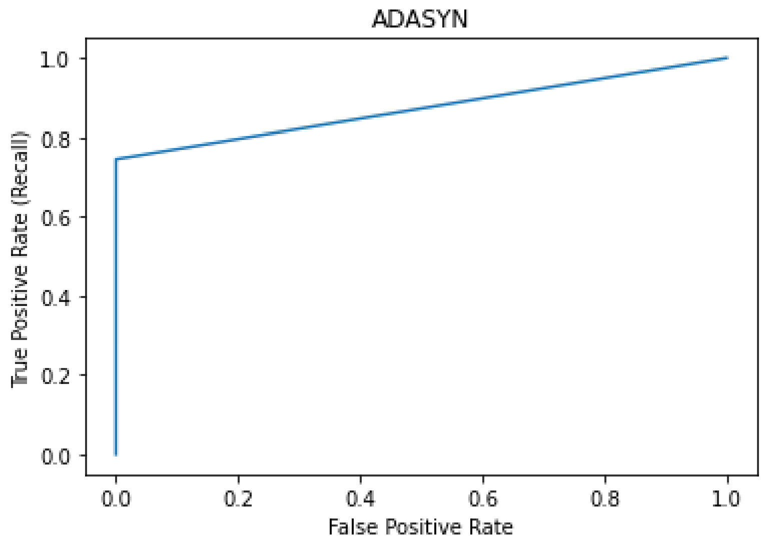
BorderLineSMOTE

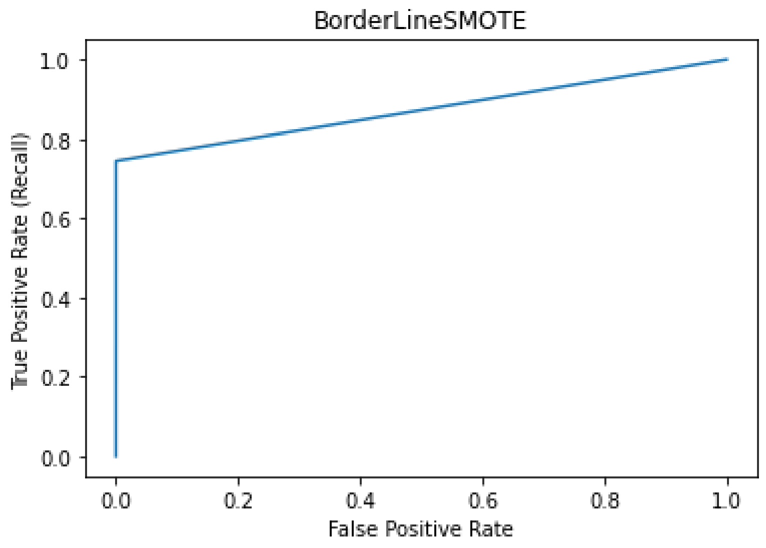


In [49]:
```
plot_rocs(rocs)
```









Random Forest Classifier with Undersampling

In [50]:
```
under_samplers = [[RandomUnderSampler(random_state=42), "RandomUnderSampler"],
                  [NearMiss(), "NearMiss"]
                 ]
```

In [51]:
```
metrics, conf_disps, rocs = run_sampler_batch(run_model=run_rfc, samplers=under_samplers, incROC=True)
```

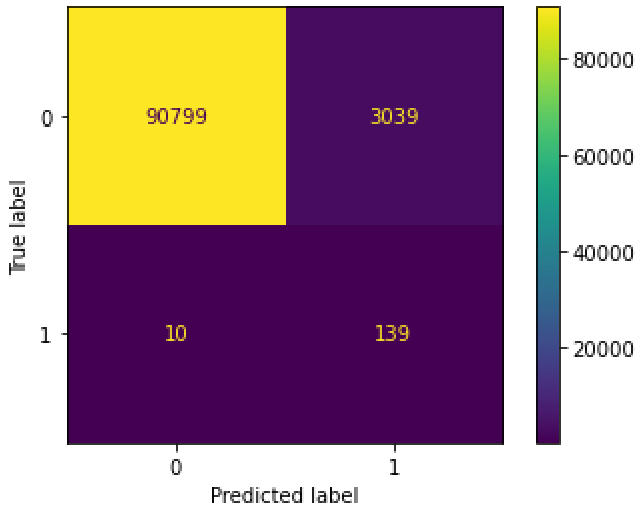
```
RUNNING RANDOM FOREST CLASSIFIER ON DATA:  RandomUnderSampler
RUNNING RANDOM FOREST CLASSIFIER ON DATA:  NearMiss
```

In [52]:
```
print_metrics(metrics)
```

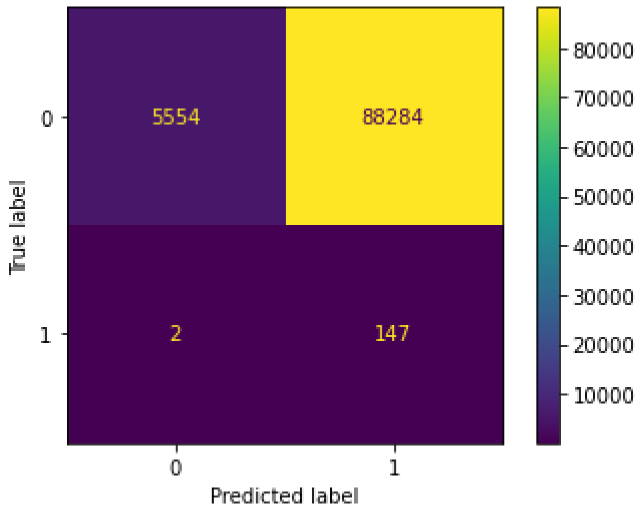
```
*****************************************
RandomUnderSampler
Accuracy Score:  0.967559343313437
Precision:  0.04373820012586532
Recall:  0.9328859060402684
F1:  0.08355876164712954

*****************************************
NearMiss
Accuracy Score:  0.0606573249491951
Precision:  0.0016623129897886488
Recall:  0.9865771812080537
F1:  0.0033190336419056223
```

In [53]:
```
plot_conf_disps(conf_disps)
```

```
RandomUnderSampler
```

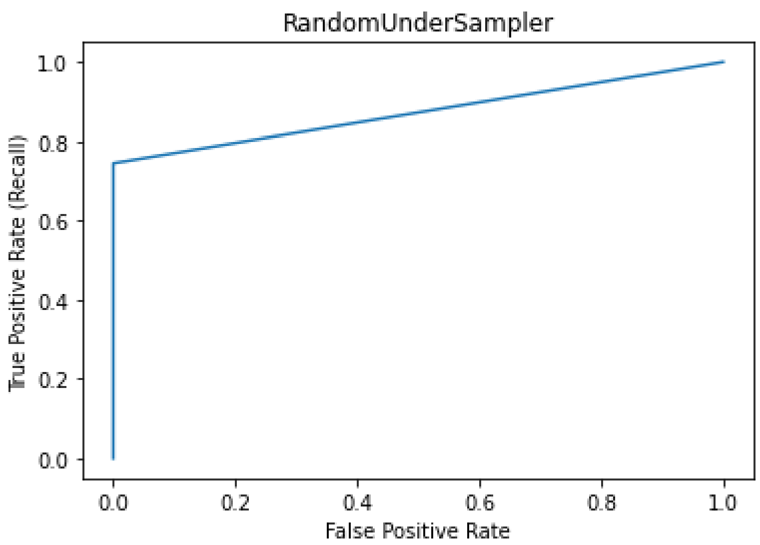


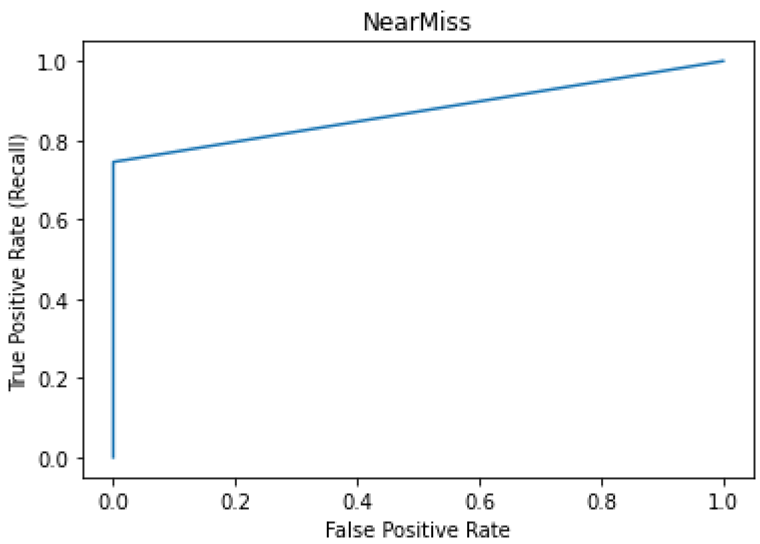
NearMiss



In [54]:

```
plot_rocs(rocs)
```





Energy Efficiency (Regression)

In [1]:

```
import pandas as pd
```

In [2]:

```
energy = pd.read_excel("ENB2012_data.xlsx")
energy.head()
```

Out[2]:

	X1	X2	X3	X4	X5	X6	X7	X8	Y1	Y2
0	0.98	514.5	294.0	110.25	7.0	2	0.0	0	15.55	21.33
1	0.98	514.5	294.0	110.25	7.0	3	0.0	0	15.55	21.33
2	0.98	514.5	294.0	110.25	7.0	4	0.0	0	15.55	21.33
3	0.98	514.5	294.0	110.25	7.0	5	0.0	0	15.55	21.33
4	0.90	563.5	318.5	122.50	7.0	2	0.0	0	20.84	28.28

In [3]:

```
energy.describe()
```

Out[3]:

	X1	X2	X3	X4	X5	X6	X7	X8	Y1	Y2
count	768.000000	768.000000	768.000000	768.000000	768.00000	768.000000	768.000000	768.00000	768.000000	768.000000
mean	0.764167	671.708333	318.500000	176.604167	5.25000	3.500000	0.234375	2.81250	22.307195	24.587760
std	0.105777	88.086116	43.626481	45.165950	1.75114	1.118763	0.133221	1.55096	10.090204	9.513306
min	0.620000	514.500000	245.000000	110.250000	3.50000	2.000000	0.000000	0.00000	6.010000	10.900000
25%	0.682500	606.375000	294.000000	140.875000	3.50000	2.750000	0.100000	1.75000	12.992500	15.620000
50%	0.750000	673.750000	318.500000	183.750000	5.25000	3.500000	0.250000	3.00000	18.950000	22.080000
75%	0.830000	741.125000	343.000000	220.500000	7.00000	4.250000	0.400000	4.00000	31.667500	33.132500
max	0.980000	808.500000	416.500000	220.500000	7.00000	5.000000	0.400000	5.00000	43.100000	48.030000

In [4]:

```
for col in energy.columns:
    print(col)
    print("********************")
    print(energy[col].value_counts())
    print()
```

```
X1
********************
0.98    64
0.90    64
0.86    64
0.82    64
0.79    64
0.76    64
0.74    64
0.71    64
0.69    64
0.66    64
0.64    64
0.62    64
Name: X1, dtype: int64

X2
********************
514.5    64
563.5    64
588.0    64
612.5    64
637.0    64
661.5    64
686.0    64
710.5    64
735.0    64
759.5    64
784.0    64
808.5    64
Name: X2, dtype: int64

X3
********************
294.0    192
318.5    192
343.0    128
416.5     64
245.0     64
269.5     64
367.5     64
Name: X3, dtype: int64
```

```
X4
********************
220.50    384
147.00    192
122.50    128
110.25     64
Name: X4, dtype: int64

X5
********************
7.0    384
3.5    384
Name: X5, dtype: int64

X6
********************
2    192
3    192
4    192
5    192
Name: X6, dtype: int64

X7
********************
0.10    240
0.25    240
0.40    240
0.00     48
Name: X7, dtype: int64

X8
********************
1    144
2    144
3    144
4    144
5    144
0     48
Name: X8, dtype: int64

Y1
********************
15.16    6
13.00    5
15.23    4
28.15    4
14.60    4
        ..
33.21    1
36.77    1
36.71    1
37.03    1
16.64    1
Name: Y1, Length: 587, dtype: int64

Y2
********************
21.33    4
29.79    4
14.27    4
17.20    4
14.28    4
        ..
14.65    1
14.54    1
14.39    1
14.46    1
17.11    1
Name: Y2, Length: 636, dtype: int64
```

In [5]:

```
energy.isna().sum()
```

Out[5]:

```
X1    0
X2    0
X3    0
X4    0
X5    0
X6    0
X7    0
X8    0
Y1    0
Y2    0
dtype: int64
```

In [6]:

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

In [7]:

```
X = energy.drop(['Y1', 'Y2'], 1).to_numpy()
```

```
y = energy.drop(['X1', 'X2', 'X3', 'X4', 'X5', 'X6', 'X7', 'X8'], 1).to_numpy()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)

scaler = StandardScaler()
scaler.fit(X)
X_train_s = scaler.transform(X_train)
X_test_s = scaler.transform(X_test)
```

```
C:\Users\Seneca\AppData\Local\Temp/ipykernel_6068/2382515809.py:1: FutureWarning: In a future version of pandas all arguments of D
ataFrame.drop except for the argument 'labels' will be keyword-only
  X = energy.drop(['Y1', 'Y2'], 1).to_numpy()
C:\Users\Seneca\AppData\Local\Temp/ipykernel_6068/2382515809.py:2: FutureWarning: In a future version of pandas all arguments of D
ataFrame.drop except for the argument 'labels' will be keyword-only
  y = energy.drop(['X1', 'X2', 'X3', 'X4', 'X5', 'X6', 'X7', 'X8'], 1).to_numpy()
```

Linear & Polynomial Regression

In [8]:

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, explained_variance_score
from sklearn.preprocessing import PolynomialFeatures
import matplotlib.pyplot as plt
```

Find best polynomial degree

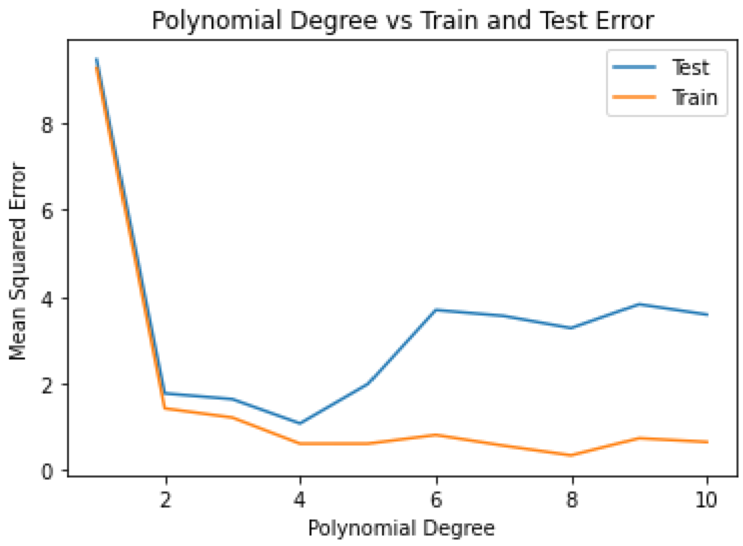
In [30]:

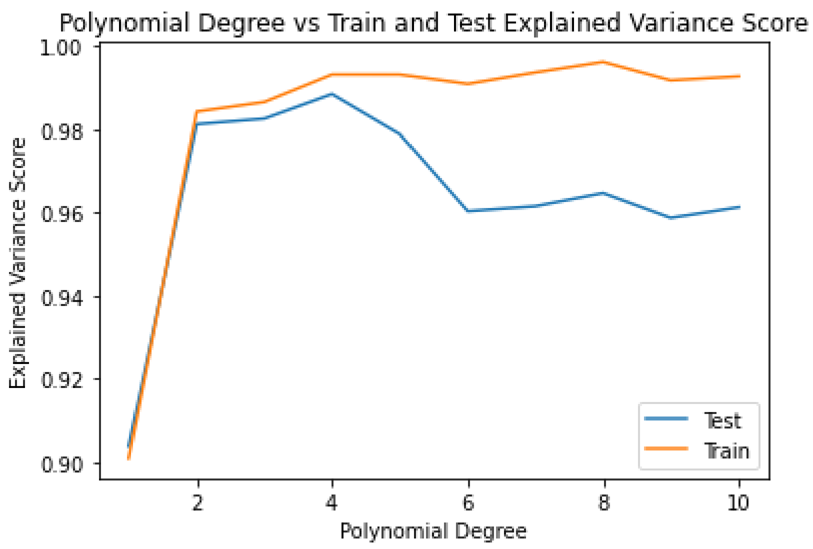
```
def run_polynomial_reg(X_tr, X_te):
    test_error = []
    test_var = []
    train_error = []
    train_var = []
    degrees = list(range(1, 11))

    # linear regression
    print("Degree: 1")
    lin_reg = LinearRegression()
    lin_reg = lin_reg.fit(X_tr, y_train)
    y_pred = lin_reg.predict(X_te)
    test_error.append(mean_squared_error(y_test, y_pred))
    test_var.append(explained_variance_score(y_test, y_pred))
    print("Test MSE: ", mean_squared_error(y_test, y_pred))
    print("Test Explained Variance: ", explained_variance_score(y_test, y_pred))

    y_pred = lin_reg.predict(X_tr)
    train_error.append(mean_squared_error(y_train, y_pred))
    train_var.append(explained_variance_score(y_train, y_pred))

    # polynomial regression
    for k in range (2, 11):
        print("Degree: ", k)
        pol = PolynomialFeatures(k)
        X_train_pol = pol.fit_transform(X_tr)
        X_test_pol = pol.fit_transform(X_te)

        # Train
        lin_reg = LinearRegression()
        lin_reg = lin_reg.fit(X_train_pol, y_train)

        # Test error
        y_pred = lin_reg.predict(X_test_pol)
        test_error.append(mean_squared_error(y_test, y_pred))
        test_var.append(explained_variance_score(y_test, y_pred))
        print("Test MSE: ", mean_squared_error(y_test, y_pred))
        print("Test Explained Variance: ", explained_variance_score(y_test, y_pred))

        # Train error
        y_pred = lin_reg.predict(X_train_pol)
        train_error.append(mean_squared_error(y_train, y_pred))
        train_var.append(explained_variance_score(y_train, y_pred))

    # make error plot
    plt.plot(degrees, test_error, label="Test")
    plt.plot(degrees, train_error, label="Train")
    plt.title("Polynomial Degree vs Train and Test Error")
    plt.xlabel("Polynomial Degree")
    plt.ylabel("Mean Squared Error")
    plt.legend()
    plt.show()

    # make explained variance plot
    plt.plot(degrees, test_var, label="Test")
    plt.plot(degrees, train_var, label="Train")
    plt.title("Polynomial Degree vs Train and Test Explained Variance Score")
    plt.xlabel("Polynomial Degree")
    plt.ylabel("Explained Variance Score")
    plt.legend()
    plt.show()
```

Linear and Polynomial Regression Without Scaling

In [31]:
```
run_polynomial_reg(X_train, X_test)
```

```
Degree: 1
Test MSE:  9.473928887107558
Test Explained Variance:  0.9039367329631596
Degree:  2
Test MSE:  1.776666517087284
Test Explained Variance:  0.9812194450573242
Degree:  3
Test MSE:  1.6434598349914515
Test Explained Variance:  0.982552314074946
Degree:  4
Test MSE:  1.082644402129154
Test Explained Variance:  0.9884413310144932
Degree:  5
Test MSE:  1.9903321686464723
Test Explained Variance:  0.9788519320756142
Degree:  6
Test MSE:  3.7020169584381586
Test Explained Variance:  0.9603563709879839
Degree:  7
Test MSE:  3.5652840333426123
Test Explained Variance:  0.9615661401947476
Degree:  8
Test MSE:  3.283734404865018
Test Explained Variance:  0.9647114917427451
Degree:  9
Test MSE:  3.8333183965856357
Test Explained Variance:  0.9587680677234443
Degree:  10
Test MSE:  3.5961897774523846
Test Explained Variance:  0.9613200920313656
```



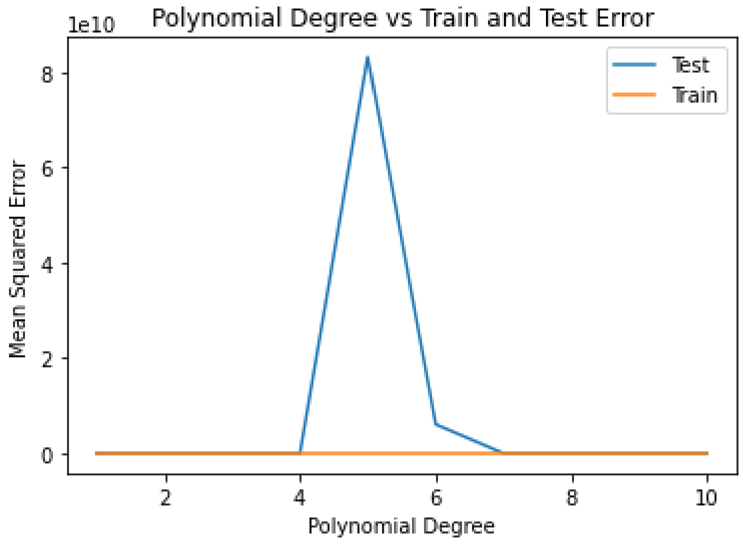


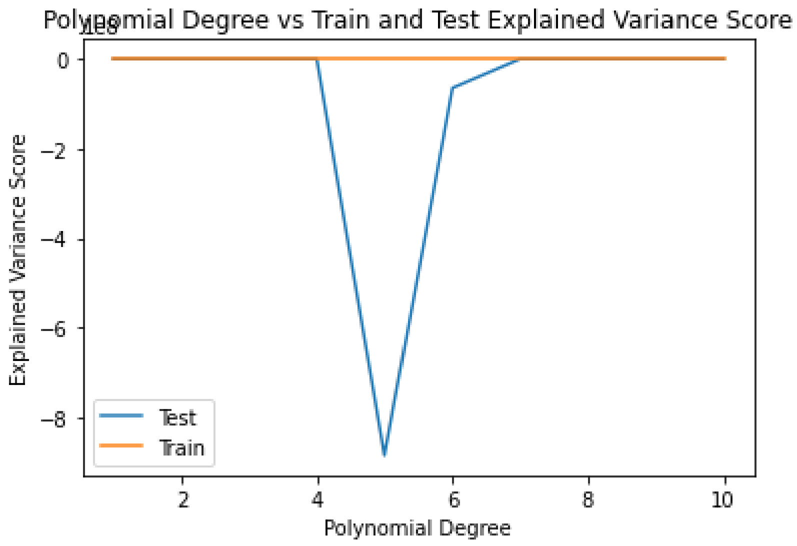
Linear and Polynomial Regression With Scaling

In [32]:
```
run_polynomial_reg(X_train_s, X_test_s)
```

```
Degree: 1
Test MSE:  9.494604383689417
Test Explained Variance:  0.903859607073465
Degree:  2
Test MSE:  1.7803196001673411
Test Explained Variance:  0.9811703714264455
Degree:  3
Test MSE:  1.8048407585068111
Test Explained Variance:  0.9807061614823389
Degree:  4
Test MSE:  1.163010164847321
Test Explained Variance:  0.9876269648293268
Degree:  5
Test MSE:  83188428505.0631
Test Explained Variance:  -884615746.5392985
Degree:  6
Test MSE:  6074869586.2866335
Test Explained Variance:  -64765644.66045452
```

```
Degree:  7
Test MSE:  519.0339146365108
Test Explained Variance:  -4.535582710096245
Degree:  8
Test MSE:  291.23574444429005
Test Explained Variance:  -2.1219878788028774
Degree:  9
Test MSE:  73.56289493848388
Test Explained Variance:  0.21533847054273098
Degree:  10
Test MSE:  151.6375396264335
Test Explained Variance:  -0.6204944250252705
```





Support Vector Machine

In [12]:
```
from sklearn.svm import LinearSVR
from sklearn.multioutput import MultiOutputRegressor
```

In [13]:
```
def run_svm(X_tr, X_te):
    svr = LinearSVR()
    multi_svr = MultiOutputRegressor(svr)
    multi_svr.fit(X_tr, y_train)
    y_pred = multi_svr.predict(X_te)
    print("Mean Squared Error: ", mean_squared_error(y_test, y_pred))
    print("Explained Variance Score: ", explained_variance_score(y_test, y_pred))
```

SVM Without Scaling

In [14]:
```
run_svm(X_train, X_test)
```

```
Mean Squared Error:  33.438587215816675
Explained Variance Score:  0.8251060837734372
```

```
C:\Users\Seneca\anaconda3\lib\site-packages\sklearn\svm\_base.py:985: ConvergenceWarning: Liblinear failed to converge, increase t
he number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
C:\Users\Seneca\anaconda3\lib\site-packages\sklearn\svm\_base.py:985: ConvergenceWarning: Liblinear failed to converge, increase t
he number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
```

SVM With Scaling

In [15]:
```
run_svm(X_train_s, X_test_s)
```

```
Mean Squared Error:  10.870072827441948
Explained Variance Score:  0.8948133892000907
```

Decision Tree

In [22]:
```
from sklearn.tree import DecisionTreeRegressor
```

In [23]:
```
def run_dec_tree(X_tr, X_te):
    tree = DecisionTreeRegressor(random_state=42)
    tree.fit(X_tr, y_train)
    y_pred = tree.predict(X_te)
    print("Mean Squared Error: ", mean_squared_error(y_test, y_pred))
    print("Explained Variance Score: ", explained_variance_score(y_test, y_pred))
```

Decision Tree Without Scaling

In [24]:
```
run_dec_tree(X_train, X_test)
```

```
Mean Squared Error:  2.8324252677165345
Explained Variance Score:  0.9696445644623223
```

Decision Tree With Scaling

In [25]:
```
run_dec_tree(X_train_s, X_test_s)
```

```
Mean Squared Error:  2.879196094488188
Explained Variance Score:  0.9690379388604704
```

Random Forest Regression

In [26]:
```
from sklearn.ensemble import RandomForestRegressor
```

In [27]:
```
def run_rand_forest(X_tr, X_te):
    rfr = RandomForestRegressor(random_state=42)
    rfr.fit(X_tr, y_train)
    y_pred = rfr.predict(X_te)
    print("Mean Squared Error: ", mean_squared_error(y_test, y_pred))
    print("Explained Variance Score: ", explained_variance_score(y_test, y_pred))
```

Random Forest Regression Without Scaling

In [28]:
```
run_rand_forest(X_train, X_test)
```

```
Mean Squared Error:  1.7338181172637772
Explained Variance Score:  0.981435199815003
```

Random Forest Regression With Scaling

In [29]:
```
run_rand_forest(X_train_s, X_test_s)
```

```
Mean Squared Error:  1.7540353284251966
Explained Variance Score:  0.9811876497266685
```