Dealing with Imbalanced Data Sets by Jeff Gross

January 10, 2018

1 Dealing with Imbalanced Data Sets

Author: Jeff Gross

The objective is to explore different approaches to tackling imbalanced data sets, in this case for fraud detection. This notebook will walk you through six classification algorithm with two over-sampling and three under-sampling techniques of working with unbalanced data, This data was retrieved from the kaggle website with the pre-processing step of the data already complete.

Regular algorithms are often biased towards the majority class because of their loss functions attempt to optimize error rate, without taking the data distribution into consideration. In the worst case, minority examples are treated as outliers of the majority class and ignored.

2 Data Dictionary

The dataset contains transactions made by credit cards in the month of September 2013 by european cardholders. All of the observations occur in a two day span, where we there were 492 frauds out of 284,807 transactions. The dataset was collected and analysed by the Machine Learning Group of ULB (Université Libre de Bruxelles) on big data mining and fraud detection. The data only contains numerical variables which are the result of a PCA transformation. The PCA transformation was for security and confidentiality reasons.

- Features V1, V2, ... V28 are the principal components obtained with PCA
- 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset
- 'Amount' is the transaction Amount, this feature can be used for example-dependant cost-senstive learning.
- 'Class' is the response variable and it takes value **1 in case of fraud and 0 otherwise.**

2.1 Results

The six different algorithms that were used in this study were logistic regression, random forest classifer, support vector classifier, XGBoost, and Neural Networks. The oversampling techniques were random over sampling and synthetic minority oversampling techinque (SMOTE). The undersampling techniques were random under sampling, edited nearest neighbor, and condensed nearest neighbors. I focused on the f1 score in my analysis. It is also a valid measure of an accurate model. It is the harmonic mean of precision and recall, and will be more insensitive to imbalanced data.

Of the six different algorithms that were used to predict this imbalanced data seet, the best algorithm was random forest classifier, without under or oversampling, with an average F1 score of .89. Second place went to XGBoost without under or oversampling with an average F1 score of .84. There was a tie for third place. Logistic regression utilizing an L2 regularization penalty, Lasso regression, with an average F1 score of .75, and logistic regression with an L2 regularization penalty and re-balanced sampling weights with an average F1 score of .76. Fourth place went to Logistic regression utilizing L2 regularization penalty and under sampling utilizing edited nearest neighbors with an average F1 score of .73

1st place: Random Forest F1 score .89

2nd place: XGBoost F1 Score .84

3rd place: Logistic Regression w.L2 F1 Score=.75, w.re-balance & L2 F1 Score=.76

4th place: Logistic Regression w.ENN & L2 F1 Score .73

Contents: * Section **??** * Section **??** * Section **??**

- Section **??**
- Section **??**
  - Section **??**
    - Section **??**
    - Section **??**
  - Section **??**
    - Section **??**
    - Section **??**
  - Section **??**
    - Section **??**
    - Section **??**
  - Section **??**
    - Section **??**
    - Section **??**
  - Section **??**
- Section **??**

Exploratory Data Analysis In this first section of the notebook I will go through and explore some of the features. I will look at their structure in the dataset, look to validate the pre-processing steps, and visualize the data to get a better understanding.

The dataset can be found here and downloaded for interactive use with this notebook: https://www.kaggle.com/dalpozz/creditcardfraud

```
In [3]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        import seaborn as sns
        from collections import Counter
        import warnings
        warnings.filterwarnings('ignore')

        %matplotlib inline
```

```
In [4]: # pandas function to read in a csv file
        df = pd.read_csv('creditcard.csv')
```

```
In [3]: print(df.shape)
        df.head()
```

```
(284807, 31)
```

```
Out[3]:    Time        V1        V2        V3        V4        V5        V6        V7  \
        0   0.0 -1.359807 -0.072781  2.536347  1.378155 -0.338321  0.462388  0.239599
        1   0.0  1.191857  0.266151  0.166480  0.448154  0.060018 -0.082361 -0.078803
        2   1.0 -1.358354 -1.340163  1.773209  0.379780 -0.503198  1.800499  0.791461
        3   1.0 -0.966272 -0.185226  1.792993 -0.863291 -0.010309  1.247203  0.237609
        4   2.0 -1.158233  0.877737  1.548718  0.403034 -0.407193  0.095921  0.592941

                 V8        V9  ...       V21       V22       V23       V24  \
        0  0.098698  0.363787  ... -0.018307  0.277838 -0.110474  0.066928
        1  0.085102 -0.255425  ... -0.225775 -0.638672  0.101288 -0.339846
        2  0.247676 -1.514654  ...  0.247998  0.771679  0.909412 -0.689281
        3  0.377436 -1.387024  ... -0.108300  0.005274 -0.190321 -1.175575
        4 -0.270533  0.817739  ... -0.009431  0.798278 -0.137458  0.141267

                V25       V26       V27       V28  Amount  Class
        0  0.128539 -0.189115  0.133558 -0.021053  149.62      0
        1  0.167170  0.125895 -0.008983  0.014724    2.69      0
        2 -0.327642 -0.139097 -0.055353 -0.059752  378.66      0
        3  0.647376 -0.221929  0.062723  0.061458  123.50      0
        4 -0.206010  0.502292  0.219422  0.215153   69.99      0

        [5 rows x 31 columns]
```

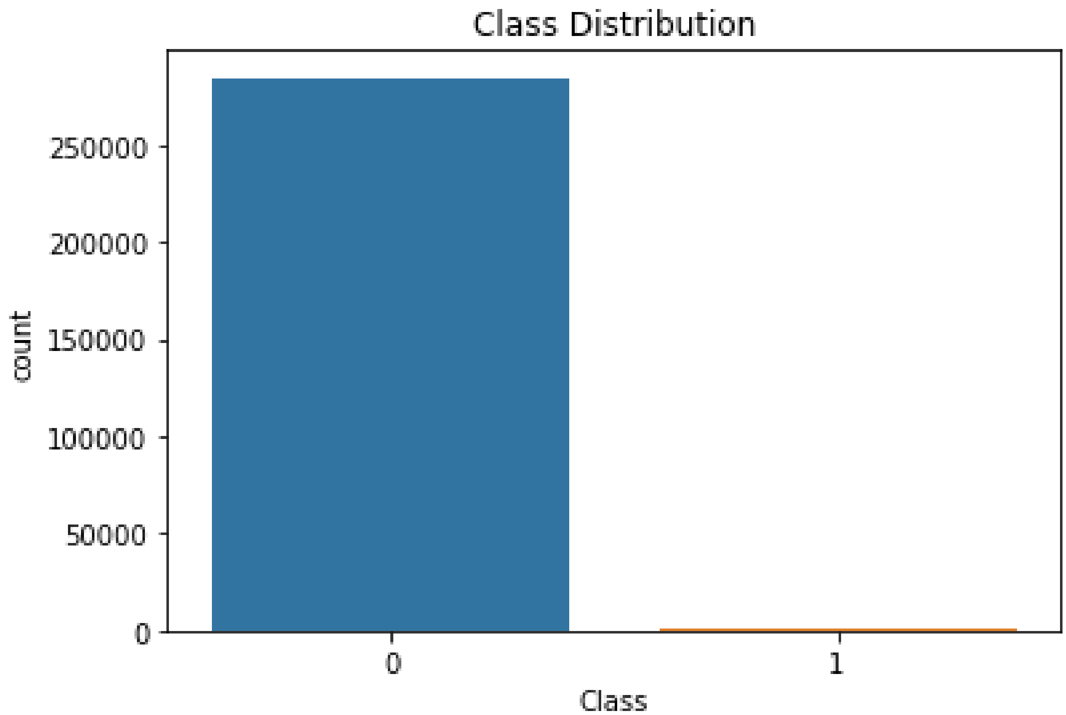
```
In [4]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
Time      284807 non-null float64
V1        284807 non-null float64
V2        284807 non-null float64
V3        284807 non-null float64
V4        284807 non-null float64
V5        284807 non-null float64
V6        284807 non-null float64
V7        284807 non-null float64
V8        284807 non-null float64
V9        284807 non-null float64
V10       284807 non-null float64
V11       284807 non-null float64
```

```
V12        284807 non-null float64
V13        284807 non-null float64
V14        284807 non-null float64
V15        284807 non-null float64
V16        284807 non-null float64
V17        284807 non-null float64
V18        284807 non-null float64
V19        284807 non-null float64
V20        284807 non-null float64
V21        284807 non-null float64
V22        284807 non-null float64
V23        284807 non-null float64
V24        284807 non-null float64
V25        284807 non-null float64
V26        284807 non-null float64
V27        284807 non-null float64
V28        284807 non-null float64
Amount     284807 non-null float64
Class      284807 non-null int64
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
```

Response Variable / Dependent Variable After getting an understanding for the structure of the data we can dive into different inquiries about certain features of the data. Like mentioned in the introduction the class has a very imbalanced data set where majority of the observations are non-fraudulant. There are only 492 fraud observations as it is noted below.

```
In [4]: Counter(df['Class'])

Out[4]: Counter({0: 284315, 1: 492})

In [5]: sns.countplot(df['Class']);
        plt.title('Class Distribution');
```



Below is an interactive map that incorporates a box plot for any selected feature with the x-axis represented by Class. This is very useful because you can see how each of the features are represented by the class. You might have a few features that are majority for one class or another.

```
In [6]: # this is an interactive map that allows you to look at the boxplot between the respon
        from ipywidgets import interact
        %matplotlib inline

        column_name = list(df.columns)

        @interact
        def box_plot(Feature= column_name):
            sns.boxplot(df['Class'], df[Feature])
```

```
interactive(children=(Dropdown(description='Feature', options=('Time', 'V1', 'V2', 'V3', 'V4',
```

We also know from the introduction that part of the pre-processing step was already completed, which we can see from zero missing values.

```
In [7]: df.isnull().sum() # There is no missing or null values in this dataset

Out[7]: 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
```

Okay, so what does this dataset consist of? The describe function allows us to get a few of the descriptive statistics of each of the variables, which may be useful for the pca components since we do not necessarily know what they are.

In [8]:
```python
# select any index value to get the summary statistics
df.describe().transpose()
```

Out[8]:
```
            count          mean           std         min           25%  \
Time     284807.0  9.481386e+04  47488.145955    0.000000  54201.500000
V1       284807.0  1.165980e-15      1.958696  -56.407510     -0.920373
V2       284807.0  3.416908e-16      1.651309  -72.715728     -0.598550
V3       284807.0 -1.373150e-15      1.516255  -48.325589     -0.890365
V4       284807.0  2.086869e-15      1.415869   -5.683171     -0.848640
V5       284807.0  9.604066e-16      1.380247 -113.743307     -0.691597
V6       284807.0  1.490107e-15      1.332271  -26.160506     -0.768296
V7       284807.0 -5.556467e-16      1.237094  -43.557242     -0.554076
V8       284807.0  1.177556e-16      1.194353  -73.216718     -0.208630
V9       284807.0 -2.406455e-15      1.098632  -13.434066     -0.643098
```

```
V10     284807.0  2.239751e-15      1.088850  -24.588262     -0.535426
V11     284807.0  1.673327e-15      1.020713   -4.797473     -0.762494
V12     284807.0 -1.254995e-15      0.999201  -18.683715     -0.405571
V13     284807.0  8.176030e-16      0.995274   -5.791881     -0.648539
V14     284807.0  1.206296e-15      0.958596  -19.214325     -0.425574
V15     284807.0  4.913003e-15      0.915316   -4.498945     -0.582884
V16     284807.0  1.437666e-15      0.876253  -14.129855     -0.468037
V17     284807.0 -3.800113e-16      0.849337  -25.162799     -0.483748
V18     284807.0  9.572133e-16      0.838176   -9.498746     -0.498850
V19     284807.0  1.039817e-15      0.814041   -7.213527     -0.456299
V20     284807.0  6.406703e-16      0.770925  -54.497720     -0.211721
V21     284807.0  1.656562e-16      0.734524  -34.830382     -0.228395
V22     284807.0 -3.444850e-16      0.725702  -10.933144     -0.542350
V23     284807.0  2.578648e-16      0.624460  -44.807735     -0.161846
V24     284807.0  4.471968e-15      0.605647   -2.836627     -0.354586
V25     284807.0  5.340915e-16      0.521278  -10.295397     -0.317145
V26     284807.0  1.687098e-15      0.482227   -2.604551     -0.326984
V27     284807.0 -3.666453e-16      0.403632  -22.565679     -0.070840
V28     284807.0 -1.220404e-16      0.330083  -15.430084     -0.052960
Amount  284807.0  8.834962e+01    250.120109    0.000000      5.600000
Class   284807.0  1.727486e-03      0.041527    0.000000      0.000000

                 50%            75%            max
Time    84692.000000  139320.500000  172792.000000
V1          0.018109       1.315642       2.454930
V2          0.065486       0.803724      22.057729
V3          0.179846       1.027196       9.382558
V4         -0.019847       0.743341      16.875344
V5         -0.054336       0.611926      34.801666
V6         -0.274187       0.398565      73.301626
V7          0.040103       0.570436     120.589494
V8          0.022358       0.327346      20.007208
V9         -0.051429       0.597139      15.594995
V10        -0.092917       0.453923      23.745136
V11        -0.032757       0.739593      12.018913
V12         0.140033       0.618238       7.848392
V13        -0.013568       0.662505       7.126883
V14         0.050601       0.493150      10.526766
V15         0.048072       0.648821       8.877742
V16         0.066413       0.523296      17.315112
V17        -0.065676       0.399675       9.253526
V18        -0.003636       0.500807       5.041069
V19         0.003735       0.458949       5.591971
V20        -0.062481       0.133041      39.420904
V21        -0.029450       0.186377      27.202839
V22         0.006782       0.528554      10.503090
V23        -0.011193       0.147642      22.528412
V24         0.040976       0.439527       4.584549
```

```
V25          0.016594       0.350716       7.519589
V26         -0.052139       0.240952       3.517346
V27          0.001342       0.091045      31.612198
V28          0.011244       0.078280      33.847808
Amount      22.000000      77.165000   25691.160000
Class        0.000000       0.000000       1.000000
```

Something to give us a better concept of what each PCA component entails below is an interactive scatterplot matrix of the class variable and any other specified feature in the data. The plot also gives the distribution for each of the features picked where you can determine if any transformations should be applied.

```
In [9]: %matplotlib inline

        @interact
        def sns_scatter(Feature_x=column_name, Feature_y=column_name):

            scatter_list = [Feature_x, Feature_y]

            sns.set(color_codes=True)
            sns.pairplot(df[scatter_list])
```

```
interactive(children=(Dropdown(description='Feature_x', options=('Time', 'V1', 'V2', 'V3', 'V4
```

Imbalanced Data:

- **Accuracy paradox**: which is the case where we get a higher accuracy percentage because it is reflecting the underlying class distribution. The dataset is highly imbalanced, the positive class (frauds) account for 0.172% of all transactions.
  - Conventional algorithms are often biased towards the majority class because their loss functions attempt to optimize quantities such as error rate, not taking the data distribution into consideration.
  - In some cases, minority examples may even be treated as outliers of the majority class and ignored, or the learning algorithm generates a classifier that classifies every example as the majority class.

List of Techniques 1. **Collect more data**, which is not plausible in this case. 2. **Use a different scoring method**. Accuracy will be biased towards the majority class, and the F1 or ROC_AUC score will be a better estimator for true positives. A few key terms for classification and accuracy:

```
__Accuracy__ = TP+TN/Total

__Precison__ = TP/(TP+FP)

__Recall__ = TP/(TP+FN)

__F1__ = (Precison * Recall) / (Precison + Recall)
```

```
__TP__ = True positive, number of cases that were positive and predicted positive

__TN__ = True negative, number of cases that were negative and predicted negative

__FP__ = False possitve, number of cases that were negative and predicted positive

__FN__= False Negative, number of cases that were positive and predicted negative

It is always a trade off for which one will affect you more FP or FN, it comes down to the scop

Using accuracy yeilds a much higher result compared to the average_precision score (area under

The f1 score is also a valid measure of an accurate model. This is the harmonic mean of precis
```

3. **Resample the dataset** so that the sample you use to build the model is more balanced.

   imblearn.under_sampling deletes instances from the over-represented class.

   imblearn.over_sampling adds copies of instances from the under-represented class (sampling with replacement).

   Over-sampling followed by under-sampling

4. **Try different algorithms**. For example decision trees use the decision boundary to split the data by looking at the class variable, and will allow both classes to be addressed.

5. **Try penalizing the model**. There are different algorithms that are specific to penalizing class and weights.

```
In [5]: def logistic_model(X_trn, y_trn, X_tst, y_tst):
            """create a function for logistic regression"""

            logreg = LogisticRegression(penalty='l1')
            logreg.fit(X_trn, y_trn)
            y_pred = logreg.predict(X_tst)

            return get_scores(y_tst, y_pred)

        def logistic_model_w(X_trn, y_trn, X_tst, y_tst):
            """create a function for logistic regression"""
            logreg = LogisticRegression(penalty='l1', class_weight={0:.1, 1:.9})
            logreg.fit(X_trn, y_trn)
            y_pred = logreg.predict(X_tst)

            return get_scores(y_tst, y_pred)

        def get_scores(y_tst, pred):
            print('Accuracy Score: {}\n'.format(accuracy_score(y_tst, pred)))
            print('Average Precision Score: {}\n'.format(average_precision_score(y_tst, pred))
            print('Average Recall Score: {}\n'.format(recall_score(y_tst, pred)))
```

```
    print('Average F1 Score: {}'.format(f1_score(y_tst, pred)))

def model_scores(y_tst, pred):
    print('Accuracy Score: {}\n'.format(accuracy_score(y_tst, pred)))
    print('Average Precision Score: {}\n'.format(average_precision_score(y_tst, pred)))
    print('Average Recall: {}\n'.format(recall_score(y_tst, pred)))
    print('Average F1 Score: {}'.format(f1_score(y_tst, pred)))

    cnf_matrix=confusion_matrix(y_tst, pred)

    fig= plt.figure(figsize=(6,3))

    sns.heatmap(cnf_matrix, cmap="coolwarm_r", annot=True, linewidths=0.5)
    plt.title("Confusion_matrix")
    plt.xlabel("Predicted_class")
    plt.ylabel("Real class")
    plt.show()
    print("\n----------Classification Report------------------------------------")
    print(classification_report(y_tst,pred))

def make_roc_curve(estimator, X_trn, y_trn, X_tst, y_tst):
    # ROC_AUC score
    y_pred_score = estimator.fit(X_trn, y_trn.values.ravel()).decision_function(X_tst.v

    fp, tp, thresholds = roc_curve(y_tst.values.ravel(), y_pred_score)
    roc_auc = auc(fp,tp)

    # Plot ROC
    plt.title('ROC_CURVE')
    plt.plot(fp, tp, 'b',label='AUC = %0.2f'% roc_auc)
    plt.legend(loc='lower right')
    plt.plot([0,1],[0,1],'r--')
    plt.xlim([-0.1,1.0])
    plt.ylim([-0.1,1.01])
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
    plt.show()
```

Models
Logistic Regression with the imbalanced classes
I use logistic regression in this case to show the different methods of working with imbalanced data. The first model will start with the original train and test, then we will use under_sample and over_sampling methods to see which works best.

```
In [6]: from sklearn.model_selection import train_test_split
        from sklearn.preprocessing import minmax_scale
        from sklearn.linear_model import LogisticRegression
        from sklearn.metrics import classification_report, average_precision_score, precision_
```

```
from sklearn.metrics import roc_curve, auc, f1_score, confusion_matrix
from sklearn.model_selection import cross_val_score
from collections import Counter
from imblearn.pipeline import make_pipeline
import winsound
```

I split the data into test and train, and print the class imbalance for each dataset with the initial imbalance. Normally, my next step would be to standardize the independent variables and apply dimensional reduction. If the data is linear then PCA or LDA would be applied. If the data is nonlinear, KernelPCA would be applied. In this case, PCA has already been applied. The PCA output would also be helpful to know how much of the variance is accounted for by the first two PCA variables.

```
In [7]: y = df['Class']
        X = df.iloc[:,:-1]

        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=
```

```
In [14]: # one was to see the counts per class
         print('Original Data {}'.format(Counter(df['Class'])))
         print('Train Data {}'.format(Counter(y_train)))
         print('Test Data {}'.format(Counter(y_test)))
```

```
Original Data Counter({0: 284315, 1: 492})
Train Data Counter({0: 227447, 1: 398})
Test Data Counter({0: 56868, 1: 94})
```

First, I will perform logistic regression with only default values.

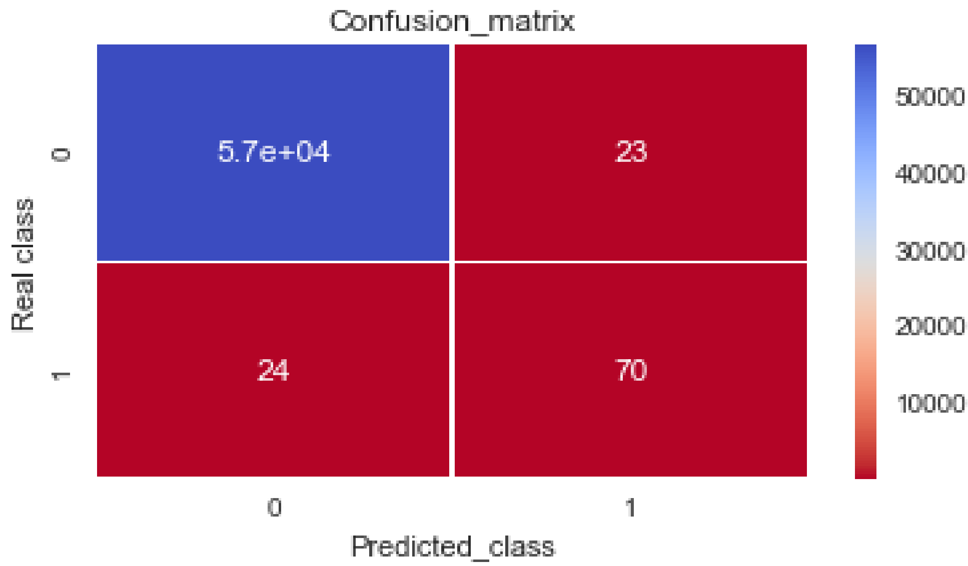
2.2 Logistic Regression (with default values)

```
In [15]: log = LogisticRegression(random_state=613)
         log.fit(X_train, y_train)
```

```
Out[15]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                   penalty='l2', random_state=613, solver='liblinear', tol=0.0001,
                   verbose=0, warm_start=False)
```

```
In [16]: # Predicting the Test set results
         log_pred = log.predict(X_test)
         model_scores(y_test, log_pred)
```

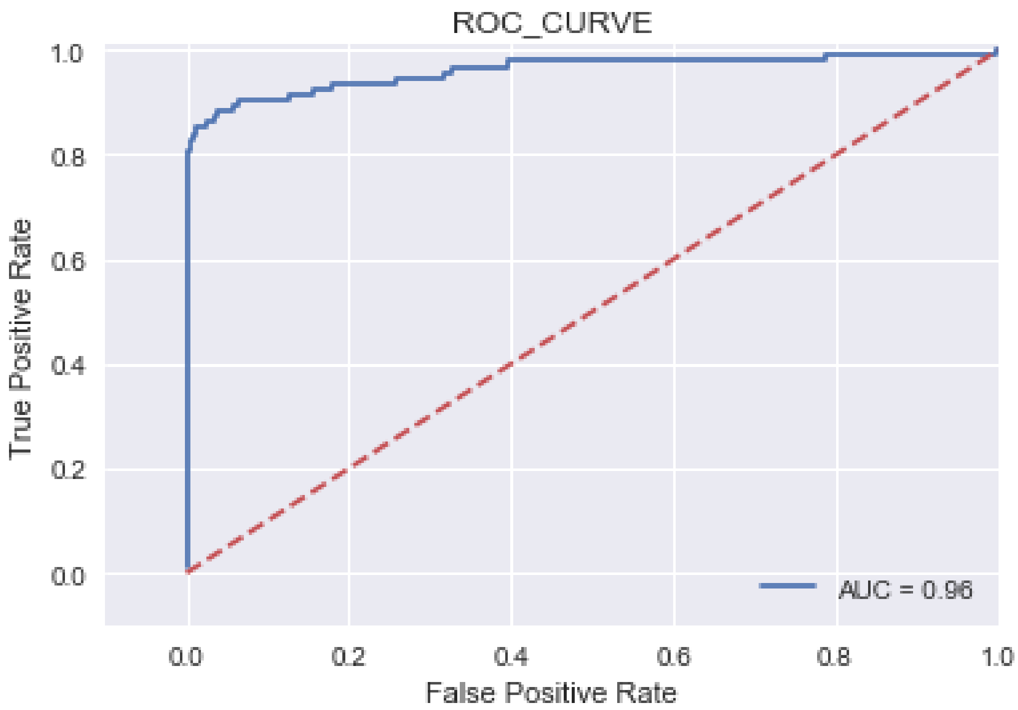
```
Accuracy Score: 0.9991748885221726

Average Precision Score: 0.5609338020632603

Average Recall: 0.7446808510638298

Average F1 Score: 0.7486631016042781
```




```
----------Classification Report------------------------------------
             precision    recall  f1-score   support

          0       1.00      1.00      1.00     56868
          1       0.75      0.74      0.75        94

avg / total       1.00      1.00      1.00     56962
```

The ROC curve represents how the classifier is performing. The x-axis is the false positive rate and the y-axis is the true positive rate or recall. We want the ROC curve to be as close to the upper left hand corner as possible, which shows that we have classified all instances correctly. The area under the curve is the percentage of tradeoff between sensitivity (true positives) and specificity (1-false positives).

In [17]:
```
#ROC curve
make_roc_curve(log, X_train, y_train, X_test, y_test)
```



In [18]:
```
# Applying k-Fold Cross Validation
from sklearn.model_selection import cross_val_score
f1 = cross_val_score(estimator = log, X = X_train, y = y_train, cv = 10, scoring='f1'
print("10-fold CV F1 Average: {}".format(np.mean(f1)))
print("10-fold CV F1 Std Dev: {} ".format(f1.std()))
```

```
10-fold CV F1 Average: 0.6674504089348291
10-fold CV F1 Std Dev: 0.06150873780722817 4
```

In [19]:
```
#Only use first two PCA variables for plot
y_g = df['Class']
X_g = df.iloc[:,1:3]

X_train_g, X_test_g, y_train_g, y_test_g = train_test_split(X_g, y_g, test_size=0.2,
```

In [20]:
```
#logisti regression with only first two PCA variables for plot
log_g = LogisticRegression(random_state=613)
log_g.fit(X_train_g, y_train_g)
```

Out[20]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=613, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

```
In [21]: #convert from dataframe to arrays
         X_train_g = X_train_g.values
         X_test_g = X_test_g.values
         y_train_g = y_train_g.values
         y_test_g = y_test_g.values
```

```
In [22]: # Visualising the Training set results
         from matplotlib.colors import ListedColormap
         X_set, y_set = X_train_g, y_train_g
         X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max(
                              np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max(
         plt.contourf(X1, X2, log_g.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.sh
                      alpha = 0.55, cmap = ListedColormap(('red', 'green')))
         plt.xlim(X1.min(), X1.max())
         plt.ylim(X2.min(), X2.max())
         for i, j in enumerate(np.unique(y_set)):
             plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                         c = ListedColormap(('red', 'green'))(i), label = j)
         plt.title('Logistic Regression (Training set)')
         plt.xlabel('PCA1')
         plt.ylabel('PCA2')
         plt.legend()
         plt.show()
```

```
In [23]: # Visualising the Test set results
         from matplotlib.colors import ListedColormap
         X_set, y_set = X_test_g, y_test_g
         X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max(
                              np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max(
         plt.contourf(X1, X2, log_g.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.sh
                      alpha = 0.55, cmap = ListedColormap(('red', 'green')))
         plt.xlim(X1.min(), X1.max())
         plt.ylim(X2.min(), X2.max())
         for i, j in enumerate(np.unique(y_set)):
             plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                         c = ListedColormap(('red', 'green'))(i), label = j)
         plt.title('Logistic Regression (Testing set)')
         plt.xlabel('PCA1')
         plt.ylabel('PCA2')
         plt.legend()
         plt.show()
```

2.3 Result Logistic Regression with default values: Average F1 Score: 0.75

2.4 Logistic Regression (with rebalancing)

```
In [24]: y = df['Class']
         X = df.iloc[:,:-1]
```

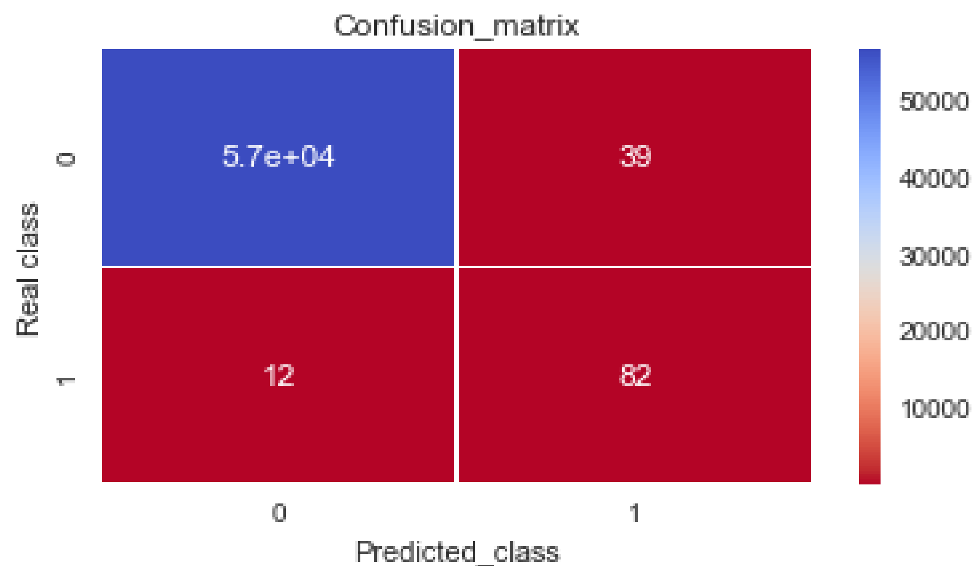
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=
```

```
In [25]: log_w = LogisticRegression(class_weight={0:.1, 1:.9}, random_state=613)
         log_w.fit(X_train, y_train)
```

```
Out[25]: LogisticRegression(C=1.0, class_weight={0: 0.1, 1: 0.9}, dual=False,
                   fit_intercept=True, intercept_scaling=1, max_iter=100,
                   multi_class='ovr', n_jobs=1, penalty='l2', random_state=613,
                   solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

```
In [26]: # Predicting the Test set results
         log_pred_w = log_w.predict(X_test)
         model_scores(y_test, log_pred_w)
```

```
Accuracy Score: 0.9991046662687406

Average Precision Score: 0.5913835171207676

Average Recall: 0.8723404255319149

Average F1 Score: 0.7627906976744186
```

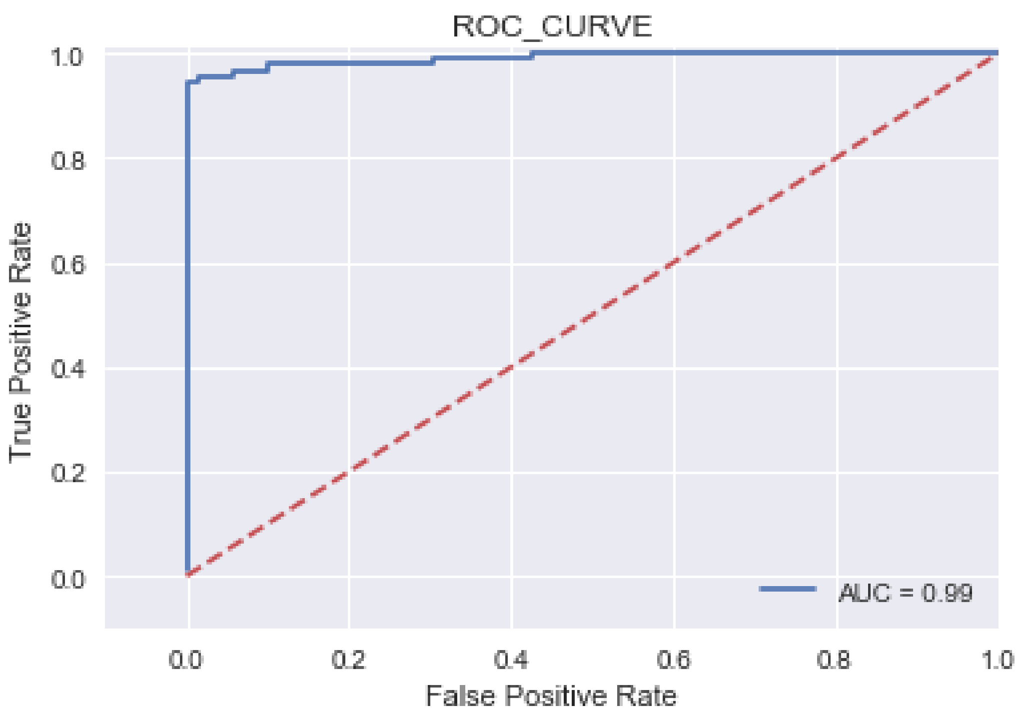



```
----------Classification Report------------------------------------
             precision    recall  f1-score   support


          0       1.00      1.00      1.00     56868
```

```
          1       0.68      0.87      0.76        94

avg / total       1.00      1.00      1.00     56962
```

The ROC curve represents how the classifier is performing. The x-axis is the false positive rate and the y-axis is the true positive rate or recall. We want the ROC curve to be as close to the upper left hand corner as possible, which shows that we have classified all instances correctly. The area under the curve is the percentage of tradeoff between sensitivity (true positives) and specificity (1-false positives).

```
In [27]: #ROC curve
         make_roc_curve(log_w, X_train, y_train, X_test, y_test)
```




```
In [8]: #Applying k-Fold Cross Validation
        log_w_pipe = make_pipeline(LogisticRegression(class_weight={0:.1, 1:.9}, random_state=(
        scores_f1 = cross_val_score(log_w_pipe, X_train, y_train, cv=10, scoring='f1')

        print("10-fold CV F1 Average: {}".format(np.mean(scores_f1)))
        print("10-fold CV F1 Std Dev: {} ".format(scores_f1.std()))

10-fold CV F1 Average: 0.7314527057974125
10-fold CV F1 Std Dev: 0.04013801485873621
```

2.5 Graph

```
In [29]: #Only use first two PCA variables for plot
         y_g = df['Class']
         X_g = df.iloc[:,1:3]

         X_train_g, X_test_g, y_train_g, y_test_g = train_test_split(X_g, y_g, test_size=0.2, r
```

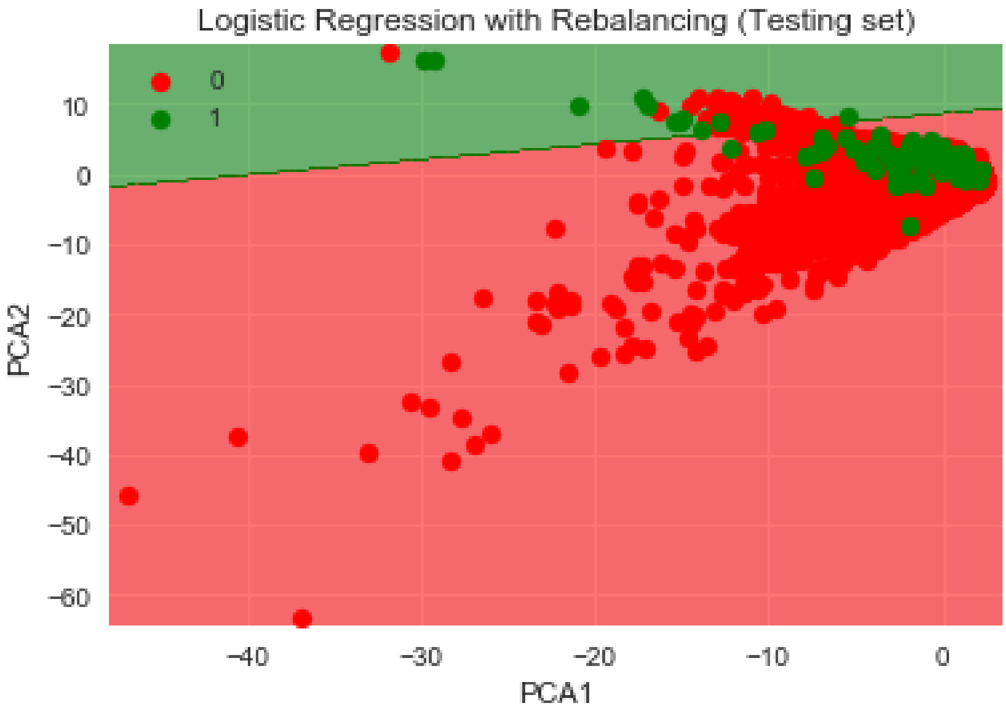
```
In [30]: #logisti regression with only first two PCA variables for plot
         log_w_g = LogisticRegression(class_weight={0:.1, 1:.9}, random_state=613)
         log_w_g.fit(X_train_g, y_train_g)
```

```
Out[30]: LogisticRegression(C=1.0, class_weight={0: 0.1, 1: 0.9}, dual=False,
                   fit_intercept=True, intercept_scaling=1, max_iter=100,
                   multi_class='ovr', n_jobs=1, penalty='l2', random_state=613,
                   solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

```
In [33]: #convert from dataframe to arrays
         X_train_g = X_train_g.values
         X_test_g = X_test_g.values
         y_train_g = y_train_g.values
         y_test_g = y_test_g.values
```

```
In [34]: # Visualising the Training set results
         from matplotlib.colors import ListedColormap
         X_set, y_set = X_train_g, y_train_g
         X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max(
                              np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max(
         plt.contourf(X1, X2, log_w_g.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1
                      alpha = 0.55, cmap = ListedColormap(('red', 'green')))
         plt.xlim(X1.min(), X1.max())
         plt.ylim(X2.min(), X2.max())
         for i, j in enumerate(np.unique(y_set)):
             plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                         c = ListedColormap(('red', 'green'))(i), label = j)
         plt.title('Logistic Regression with Rebalancing (Training set)')
         plt.xlabel('PCA1')
         plt.ylabel('PCA2')
         plt.legend()
         plt.show()
```

In [35]:
```
# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test_g, y_test_g
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max(
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max(
plt.contourf(X1, X2, log_w_g.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1
             alpha = 0.55, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Logistic Regression with Rebalancing (Testing set)')
plt.xlabel('PCA1')
plt.ylabel('PCA2')
plt.legend()
plt.show()
```



2.6 Result Logistic Regression with rebalancing: Average F1 Score: 0.76

Under_sampling Data

Under-sampling will downsample the majority class. Some people view that the disadvantage to under-sampling is that valuable data is being discarded, and is making the independent variables look like they have a higher variance between features.

One article that argues for undersampling with a mathematical foundation is called *Class Imbalance* (by Wallace, Small, Bradley, and Trikalinos4). Their argument is that two classes must be distinguishable in the tail of some distribution of an explanatory variable.

Random Under Sampling:

Drops data from the majority class at random, usually until response is balanced.

```
In [5]: from imblearn.under_sampling import RandomUnderSampler
        from imblearn.pipeline import make_pipeline
```

```
In [37]: rus = RandomUnderSampler(replacement=False, random_state = 1)
         X_train_rus, y_train_rus = rus.fit_sample(X_train, y_train)

         print('Original train set was {} and Random Under train set was {}.'.format(len(y_trai
```

```
Original train set was 227845 and Random Under train set was 796.
```

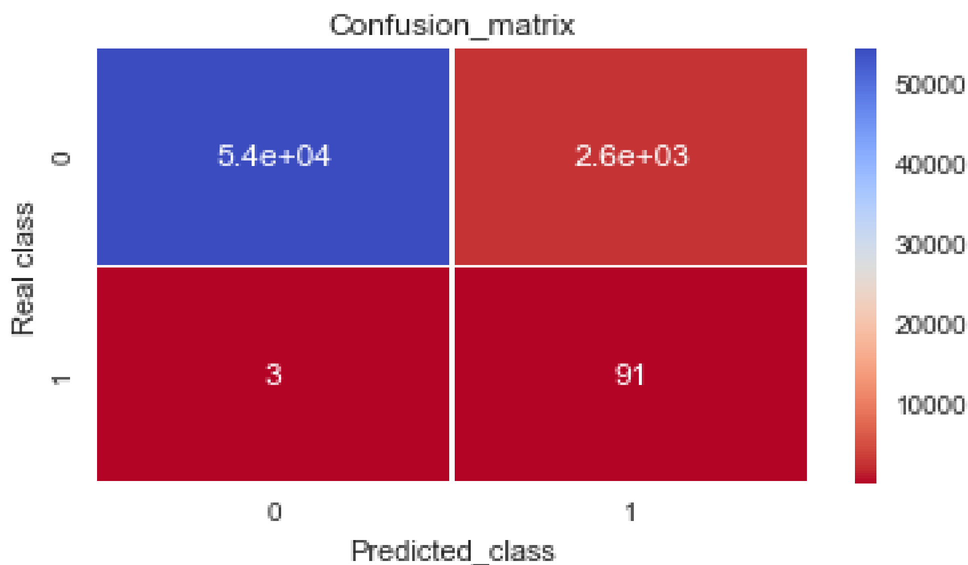
Now lets apply it to the Logistic Regression model and see its performance.

```
In [38]: log_rus = LogisticRegression(random_state=613)
         log_rus.fit(X_train_rus, y_train_rus)
```

```
Out[38]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                   penalty='l2', random_state=613, solver='liblinear', tol=0.0001,
                   verbose=0, warm_start=False)
```

```
In [39]: # Predicting the Test set results
         y_pred_rus = log_rus.predict(X_test)
         model_scores(y_test, y_pred_rus)
```

```
Accuracy Score: 0.9548470910431516

Average Precision Score: 0.03317136769791283

Average Recall: 0.9680851063829787

Average F1 Score: 0.06608569353667394
```




```
----------Classification Report------------------------------------
             precision    recall  f1-score   support

          0       1.00      0.95      0.98     56868
          1       0.03      0.97      0.07        94

avg / total       1.00      0.95      0.98     56962
```

The pipeline runs the two in parallel and then fits the model on the train and test.

As an additional step we will later go through and see how a gridsearch is then applied to run a cross validation itteration over the model.

```
In [40]: #Applying k-Fold Cross Validation
         rus_log_pipe = make_pipeline(RandomUnderSampler(replacement=False, random_state=613),
         scores_f1 = cross_val_score(rus_log_pipe, X_train, y_train, cv=10, scoring='f1')

         print("10-fold CV F1 Average: {}".format(np.mean(scores_f1)))
         print("10-fold CV F1 Std Dev: {} ".format(scores_f1.std()))
```

```
10-fold CV F1 Average: 0.09553835503234881
10-fold CV F1 Std Dev: 0.013824559872636609
```

2.7 Result Logistic Regression with Randum Undersampling: Average F1 Score: 0.10

2.8 Edited Nearest Neighbors:

Removes all instances that are misclassified nearest neighbors from the training set.

In a sense it take cares of outliers or boundary issues, because removes nearest neighbors in the classified 'all' section. That means if there are a few data points that are not being classified towards a specific class, and tend to be classified as anything. Below we will do a similar logistic regression model using this function to see the results.

```
In [6]: from imblearn.under_sampling import EditedNearestNeighbours
```

```
In [42]: enn5 = EditedNearestNeighbours(n_neighbors=5, random_state = 1)
         X_train_enn5, y_train_enn5 = enn5.fit_sample(X_train, y_train)

         print('Original Training set & Edited Nearesr Neighbors Sample set:', [len(y_train),
```

```
Original Training set & Edited Nearesr Neighbors Sample set: [227845, 227103]
```

```
In [43]: enn5_pipe = make_pipeline(EditedNearestNeighbours(n_neighbors=5, random_state=1), Log
         scores_f1 = cross_val_score(enn5_pipe, X_train, y_train, cv=10, scoring='f1', n_jobs=

         print('Average Cross Validated F1 Score:',np.mean(scores_f1))
         print("10-fold CV F1 Std Dev: {} ".format(scores_f1.std()))
```

```
Average Cross Validated F1 Score: 0.663612800219
10-fold CV F1 Std Dev: 0.042630136444790884
```

Unlike the random sampling, this method barely shrunk the data. If you noticed the parameters that I have been using are the default parameters, which can be adjusted. Lets try the same model again with an increased n_neighbors at 10.

```
In [44]: enn10 = EditedNearestNeighbours(n_neighbors=10, random_state = 1)
         X_train_enn10, y_train_enn10 = enn10.fit_sample(X_train, y_train)

         print('Original Training set & Edited Nearesr Neighbors Sample set:', [len(y_train), l
```

```
Original Training set & Edited Nearesr Neighbors Sample set: [227845, 227103]
```

Even with ten we are still only 1000 parameters off. Lets see what the results show then adjust the parameters one more time.

```
In [45]: enn10_pipe = make_pipeline(EditedNearestNeighbours(n_neighbors=10, random_state=1), Lo
         scores_f1 = cross_val_score(enn10_pipe, X_train, y_train, cv=10, scoring='f1')

         print('Average Cross Validated F1 Score:',np.mean(scores_f1))
         print("10-fold CV F1 Std Dev: {} ".format(scores_f1.std()))
```

```
Average Cross Validated F1 Score: 0.646926903345
10-fold CV F1 Std Dev: 0.057533519888639986
```

```
In [46]: enn_pipe15 = make_pipeline(EditedNearestNeighbours(n_neighbors=15, random_state=1), Lo
         scores_f1 = cross_val_score(enn_pipe15, X_train, y_train, cv=10, scoring='f1',n_jobs=-

         print('Average Cross Validated F1 Score:',np.mean(scores_f1))
         print("10-fold CV F1 Std Dev: {} ".format(scores_f1.std()))
```

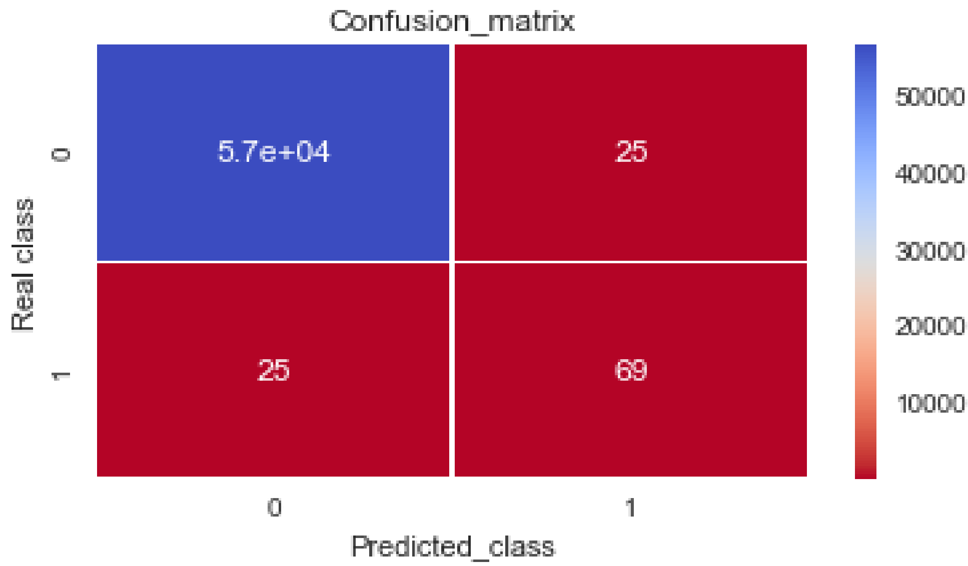
```
Average Cross Validated F1 Score: 0.661612619319
10-fold CV F1 Std Dev: 0.06393111984493369
```

```
In [47]: log_enn10 = LogisticRegression(random_state=613)
         log_enn10.fit(X_train_enn10, y_train_enn10)
```

```
Out[47]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                   penalty='l2', random_state=613, solver='liblinear', tol=0.0001,
                   verbose=0, warm_start=False)
```

```
In [48]: # Predicting the Test set results
         y_pred_enn10 = log_enn10.predict(X_test)
         model_scores(y_test, y_pred_enn10)
```

```
Accuracy Score: 0.9991222218320986

Average Precision Score: 0.5392573589798312

Average Recall: 0.7340425531914894

Average F1 Score: 0.7340425531914893
```



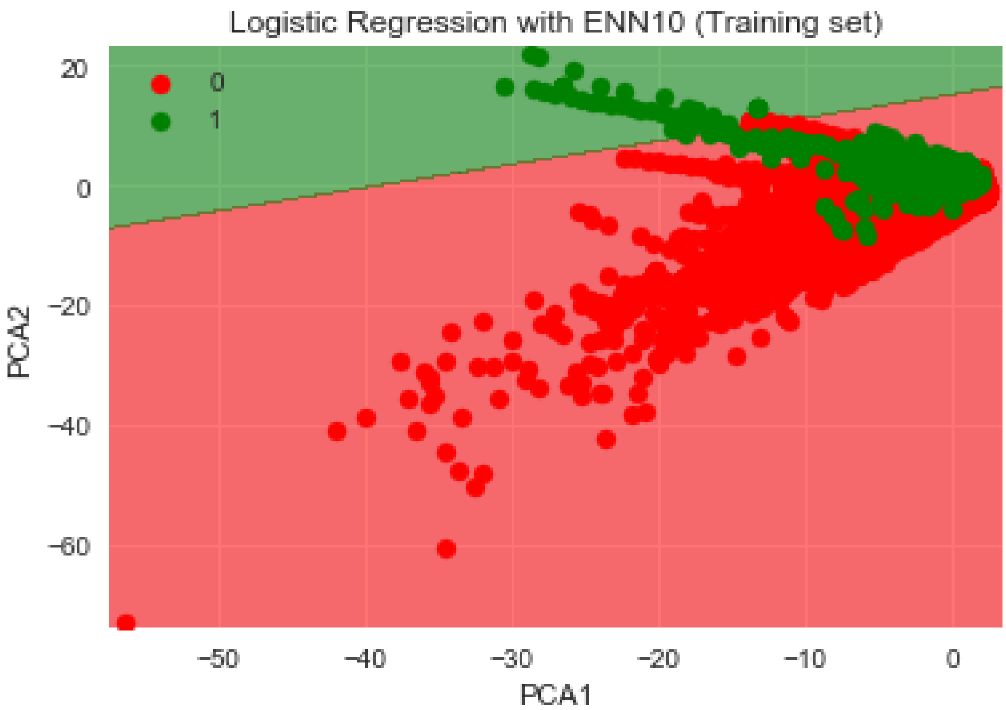

```
----------Classification Report------------------------------------
             precision    recall  f1-score   support

          0       1.00      1.00      1.00     56868
          1       0.73      0.73      0.73        94

avg / total       1.00      1.00      1.00     56962
```

```
In [49]: enn10 = EditedNearestNeighbours(n_neighbors=10, random_state = 1)
         X_train_enn10_g, y_train_enn10_g = enn10.fit_sample(X_train_g, y_train_g)

In [50]: log_enn10_g = LogisticRegression(random_state=613)
         log_enn10_g.fit(X_train_enn10_g, y_train_enn10_g)

Out[50]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                   penalty='l2', random_state=613, solver='liblinear', tol=0.0001,
                   verbose=0, warm_start=False)
```

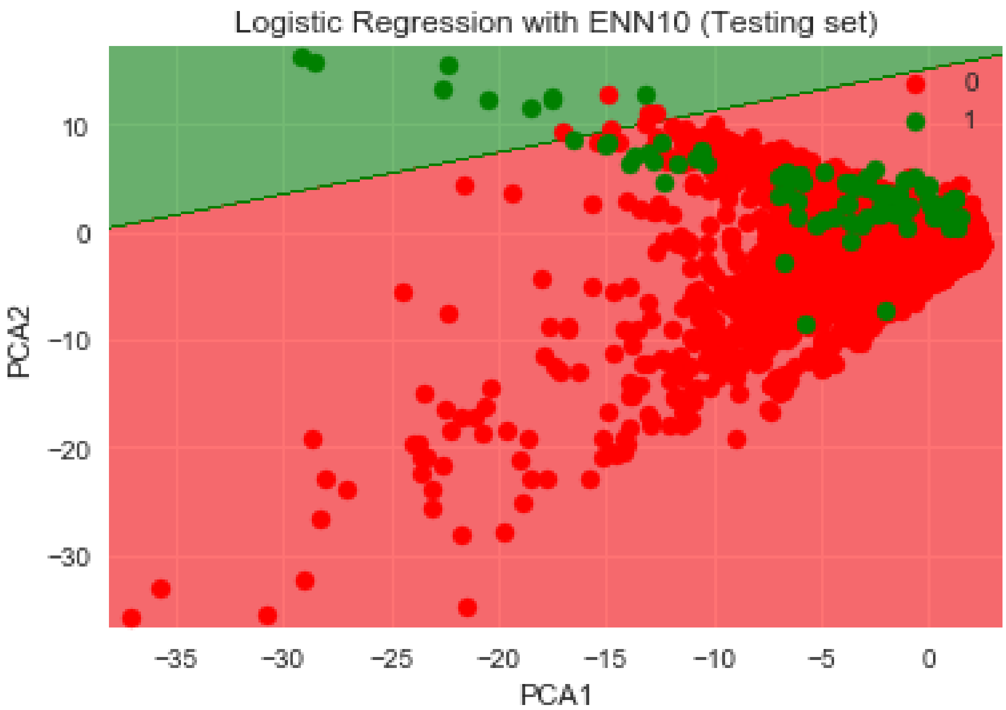
2.9 Graph

```
In [51]: #Only use first two PCA variables for plot
         y_g = df['Class']
         X_g = df.iloc[:,1:3]

         X_train_g, X_test_g, y_train_g, y_test_g = train_test_split(X_g, y_g, test_size=0.2, r
```

```
In [52]: #logisti regression with only first two PCA variables for plot
         log_w_g = LogisticRegression(class_weight={0:.1, 1:.9}, random_state=613)
         log_w_g.fit(X_train_g, y_train_g)

Out[52]: LogisticRegression(C=1.0, class_weight={0: 0.1, 1: 0.9}, dual=False,
                   fit_intercept=True, intercept_scaling=1, max_iter=100,
                   multi_class='ovr', n_jobs=1, penalty='l2', random_state=613,
                   solver='liblinear', tol=0.0001, verbose=0, warm_start=False)

In [54]: #convert from dataframe to arrays
         X_train_g = X_train_g.values
         X_test_g = X_test_g.values
         y_train_g = y_train_g.values
         y_test_g = y_test_g.values

In [55]: # Visualising the Training set results
         from matplotlib.colors import ListedColormap
         X_set, y_set = X_train_enn10_g, y_train_enn10_g
         X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max(
                              np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max(
         plt.contourf(X1, X2, log_enn10_g.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape
                      alpha = 0.55, cmap = ListedColormap(('red', 'green')))
         plt.xlim(X1.min(), X1.max())
         plt.ylim(X2.min(), X2.max())
         for i, j in enumerate(np.unique(y_set)):
             plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                         c = ListedColormap(('red', 'green'))(i), label = j)
         plt.title('Logistic Regression with ENN10 (Training set)')
         plt.xlabel('PCA1')
         plt.ylabel('PCA2')
         plt.legend()
         plt.show()
```



In [56]:
```
# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test_g, y_test_g
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max(
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max(
plt.contourf(X1, X2, log_enn10_g.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape
             alpha = 0.55, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Logistic Regression with ENN10 (Testing set)')
plt.xlabel('PCA1')
plt.ylabel('PCA2')
plt.legend()
plt.show()
```



2.10 Result Using Edited Nearest Neighbours(n_neighbors=10):

2.11 Average Cross Validated F1 Score: 0.73

Another one similar to edited nearest neighbors is condensed nearest neighbors.

2.12 Condensed Nearest Neighbors:

Opposite of edited nearest neighbors it will itteratively add points to data misclassified by K-nearest neighbors. Generally will remove a lot of points from majority class.

```
In [29]: from imblearn.under_sampling import CondensedNearestNeighbour
```

```
In [30]: cnn_pipe5 = make_pipeline(CondensedNearestNeighbour(n_neighbors=5, random_state=1), Lo
         scores_f1 = cross_val_score(cnn_pipe5, X_train, y_train, cv=10, scoring='f1',n_jobs=-

         winsound.Beep(500,10000)
```

```
In [31]: print('Average Cross Validated F1 Score:',np.mean(scores_f1))
         print("10-fold CV F1 Std Dev: {} ".format(scores_f1.std()))
```

```
Average Cross Validated F1 Score: 0.30254918247054835
10-fold CV F1 Std Dev: 0.060395570396020844
```

```
In [ ]: # cnn5 = CondensedNearestNeighbour(random_state =613, n_neighbors=5,n_jobs=-1)
        # X_train_cnn5, y_train_cnn5 = cnn5.fit_sample(X_train, y_train)

        # log = LogisticRegression()
        # log_cnn5 = log.fit(X_train_cnn5, y_train_cnn5)

        # log_pred_cnn5 = log_cnn5.predict(X_test)

        # model_scores(y_test, log_pred_cnn5)
```

```
In [32]: cnn_pipe10 = make_pipeline(CondensedNearestNeighbour(n_neighbors=10, random_state=1),
         scores_f1 = cross_val_score(cnn_pipe10, X_train, y_train, cv=10, scoring='f1',n_jobs=
```

```
In [33]: print('Average Cross Validated F1 Score:',np.mean(scores_f1))
         print("10-fold CV F1 Std Dev: {} ".format(scores_f1.std()))
```

```
Average Cross Validated F1 Score: 0.21625446079591465
10-fold CV F1 Std Dev: 0.03889599766012031
```

```
In [ ]: # cnn_pipe15 = make_pipeline(CondensedNearestNeighbour(n_neighbors=15, random_state=1)
        # scores_f1 = cross_val_score(cnn_pipe15, X_train, y_train, cv=10, scoring='f1',n_jobs
```

```
In [ ]: # print('Average Cross Validated F1 Score:',np.mean(scores_f1))
        # print("10-fold CV F1 Std Dev: {} ".format(scores_f1.std()))
```

Result Using Condensed Nearest Neighbour(n_neighbors=5): ## Average Cross Validated F1 Score: 0.35

Over_sampling Data

Over-sampling will randomly replicate minority class values to increase the sample size. Since it is replicating instances, we have to keep in mind that variables will now appear to have lower variance. However, because we are replicating instances it also means we are replicating the number of errors. So when a classifier makes a false negative error, the new sampled dataset will not make new errors for that replicated point.

Random Over_sampling:

This method is similar to the way random under_sampling works, however in this case it duplicates instances in the minority class at random.

```
In [8]: from imblearn.over_sampling import RandomOverSampler
```

```
In [7]: ros = RandomOverSampler(random_state = 1)
        X_train_ros, y_train_ros = ros.fit_sample(X_train, y_train)
```

```
In [8]: print('Original Training set & Random Over Sample set:', [len(y_train), len(y_train_ros
```

```
Original Training set & Random Over Sample set: [227845, 454894]
```

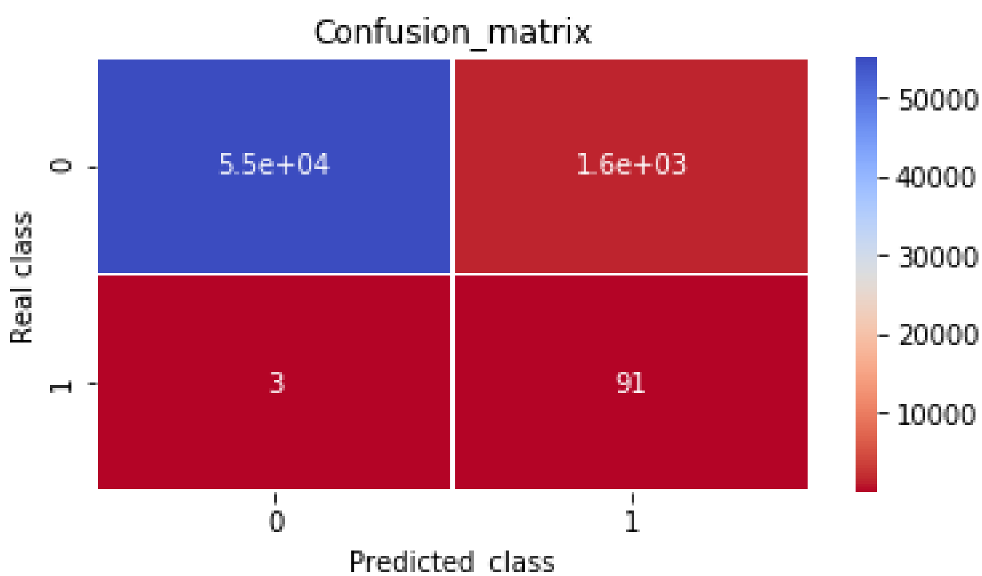
```
In [9]: ros_pipe = make_pipeline(RandomOverSampler(random_state=1), LogisticRegression())
        scores_f1 = cross_val_score(ros_pipe, X_train, y_train, cv=10, scoring='f1')
```

```
In [10]: print('Average Cross Validated F1 Score:',np.mean(scores_f1))
         print("10-fold CV F1 Std Dev: {} ".format(scores_f1.std()))
```

```
Average Cross Validated F1 Score: 0.098937605187
10-fold CV F1 Std Dev: 0.012757269748356859
```

```
In [11]: ros = RandomOverSampler(random_state = 1)
         X_train_ros, y_train_ros = ros.fit_sample(X_train, y_train)

         log = LogisticRegression()
         log_ros = log.fit(X_train_ros, y_train_ros)

         log_pred_ros = log_ros.predict(X_test)

         model_scores(y_test, log_pred_ros)
```

```
Accuracy Score: 0.9715950984867104

Average Precision Score: 0.05169143848424235

Average Recall: 0.9680851063829787

Average F1 Score: 0.10111111111111112
```




```
----------Classification Report------------------------------------
```

```
             precision    recall  f1-score   support

          0       1.00      0.97      0.99     56868
          1       0.05      0.97      0.10        94

avg / total       1.00      0.97      0.98     56962
```

Result Using Random Over Samplinig: Average Cross Validated F1 Score: 0.11

2.13 SMOTE (Synthetic Minority Oversampling TEchnique):

This techniques creates new data points from the minority class by interpolating between existing ones. Only used for continuous data. 1. It ignores the majority class 2. Then for every minority instance it selects K-nearest neighbors 3. Then creates new data points from the first instance to each of its k-nearest neighbors

```
In [9]: from imblearn.over_sampling import SMOTE
```

```
In [14]: sm = SMOTE()
         X_train_sm, y_train_sm = sm.fit_sample(X_train, y_train)
```

```
In [15]: print('Resampled dataset shape {}'.format(Counter(y_train_sm)))
         print(y_train_sm.shape, X_train_sm.shape)
```

```
Resampled dataset shape Counter({0: 227447, 1: 227447})
(454894,) (454894, 30)
```

```
In [16]: logistic_model(X_train_sm, y_train_sm, X_test, y_test)
```

```
Accuracy Score: 0.9905199957866648

Average Precision Score: 0.1377223193236517

Average Recall Score: 0.9574468085106383

Average F1 Score: 0.25
```

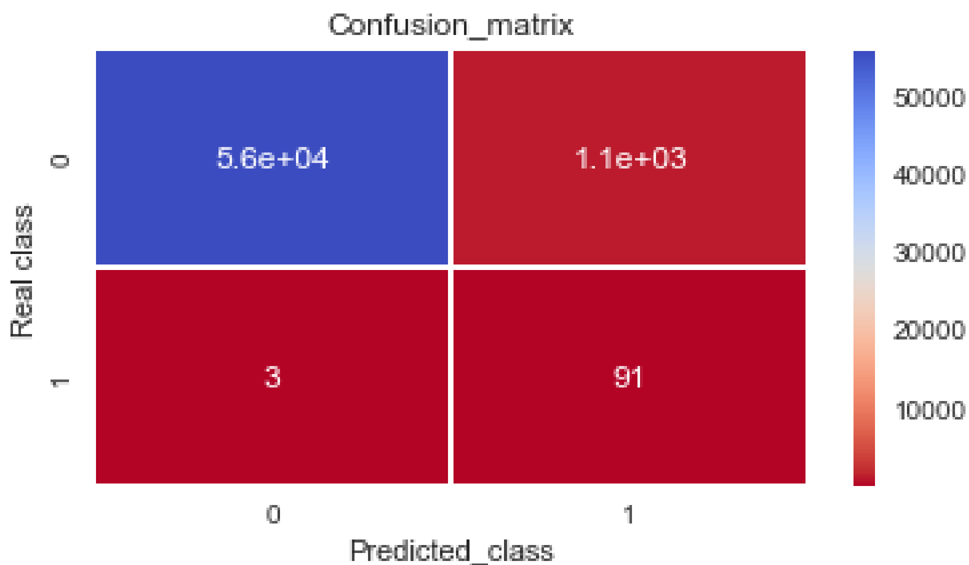
```
In [17]: sm_pipe = make_pipeline(SMOTE(), LogisticRegression())
         scores_f1 = cross_val_score(sm_pipe, X_train, y_train, cv=10, scoring='f1')
```

```
In [18]: print('Average Cross Validated F1 Score:',np.mean(scores_f1))
         print("10-fold CV F1 Std Dev: {} ".format(scores_f1.std()))
```

```
Average Cross Validated F1 Score: 0.14628456192
10-fold CV F1 Std Dev: 0.018457904135908917
```

```
In [19]: sm = SMOTE()
         X_train_sm, y_train_sm = sm.fit_sample(X_train, y_train)

         log = LogisticRegression()
         log_sm = log.fit(X_train_sm, y_train_sm)

         log_pred_sm = log_sm.predict(X_test)

         model_scores(y_test, log_pred_sm)
```

```
Accuracy Score: 0.9801095467153541

Average Precision Score: 0.07220315373417815

Average Recall: 0.9680851063829787

Average F1 Score: 0.13840304182509505
```




```
----------Classification Report------------------------------------
             precision    recall  f1-score   support

          0       1.00      0.98      0.99     56868
          1       0.07      0.97      0.14        94

avg / total       1.00      0.98      0.99     56962
```

Result Using SMOTE (Synthetic Minority Oversampling TEchnique: Average Cross Validated F1 Score: 0.14
Random Forest Classifier
Random Forest is an ensemble method that will do both classification and regression. Random forest takes a subsample of the data set using a technique called bootstrapping. What bootstrap does is it keeps the same length of the data but replaces on observation with a random observation from that sample. With this technique you may get multiples of the same observation, but the idea is that you are covering the entire population. From this sample it then generates a set of decisions based on a random sample of features. It decides the threshold of the feature, and makes a split. Each of the trees MSE value is averaged together to output the accuracy.

Random Forest is a part of the CART (Classification and Regression Trees). The tree series stems off the basic idea of a decision tree with rules that split the data into different nodes.

First, the Random Forest will be run without undersample or oversample. Then it will be run with undersample and oversample.

```
In [27]: from sklearn.ensemble import RandomForestClassifier
         from sklearn.model_selection import GridSearchCV
         from sklearn.metrics import confusion_matrix
```

2.13.1 RF without undersample/oversample

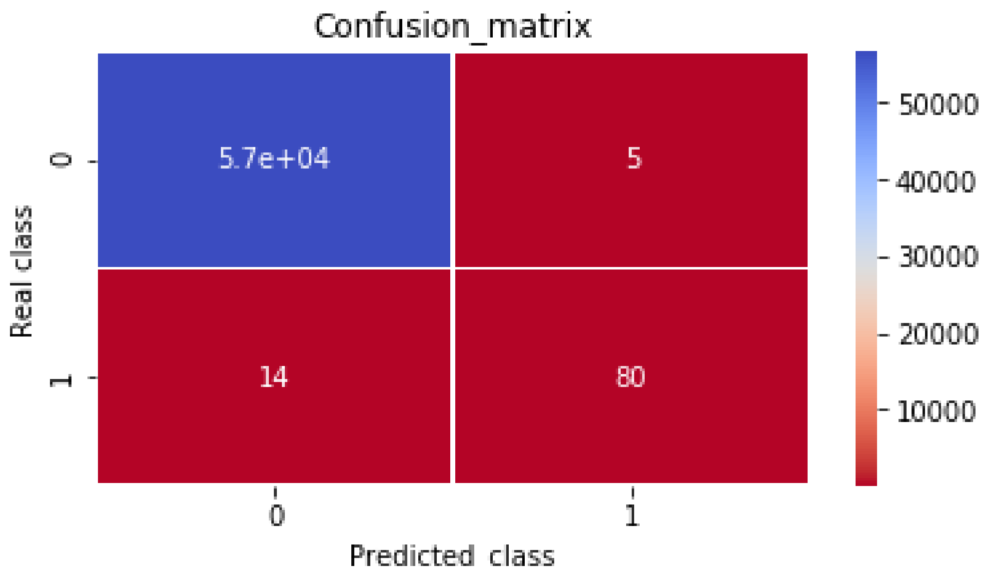
As general model for comparison I want to show how well random forest performs without oversampling and undersampling.

```
In [14]: rf_params = {
             'n_estimators' : [100, 150],
             'max_depth': [5, 7],
             'min_samples_split' : [2, 3],
         }
```

```
In [ ]: #rf_params = {
        #    'n_estimators' : np.arange(100, 300, 50),
        #    'criterion' : ['gini','entropy'],
        #    'max_features' : ['auto', 'sqrt', 'log2'],
        #    'max_depth': np.arange(1, 10, 1),
        #    'min_samples_split' : np.arange(2, 10, 1),
        #    'class_weight' : [{0: .1, 1: .9}]
        #}
```

```
In [15]: rf = RandomForestClassifier()
         grid_search = GridSearchCV(estimator = rf,
                                    param_grid = rf_params,
                                    scoring = 'f1',
                                    cv = 10,
                                    n_jobs = -1)
         rf_grid = GridSearchCV(rf, param_grid=rf_params)
         rf_model = rf_grid.fit(X_train, y_train)

         rf_pred = rf_model.predict(X_test)
```

```
In [16]: model_scores(y_test, rf_pred)

Accuracy Score: 0.9996664442961974

Average Precision Score: 0.801247029451468

Average Recall: 0.851063829787234

Average F1 Score: 0.8938547486033519
```




```
----------Classification Report------------------------------------
             precision    recall  f1-score   support

          0       1.00      1.00      1.00     56868
          1       0.94      0.85      0.89        94

avg / total       1.00      1.00      1.00     56962
```

Result Using Random Forest: Average Cross Validated F1 Score: 0.89

2.14 Graph

```
In [17]: #Only use first two PCA variables for plot
         y_g = df['Class']
```

```
X_g = df.iloc[:,1:3]

X_train_g, X_test_g, y_train_g, y_test_g = train_test_split(X_g, y_g, test_size=0.2, r
```

In [18]:
```
#Random Forest with only first two PCA variables for plot

rf_params_g = {
    'n_estimators' : [100],
    'max_depth': [5],
    'min_samples_split' : [2],
}

rf_g = RandomForestClassifier()
grid_search_g = GridSearchCV(estimator = rf_g,
                           param_grid = rf_params_g,
                           scoring = 'f1',
                           cv = 10,
                           n_jobs = -1)

rf_grid_g = GridSearchCV(rf_g, param_grid=rf_params_g)
rf_model_g = rf_grid_g.fit(X_train_g, y_train_g)
```

In [19]:
```
#convert from dataframe to arrays
X_train_g = X_train_g.values
X_test_g = X_test_g.values
y_train_g = y_train_g.values
y_test_g = y_test_g.values
```

In [20]:
```
type(X_train_g)
```

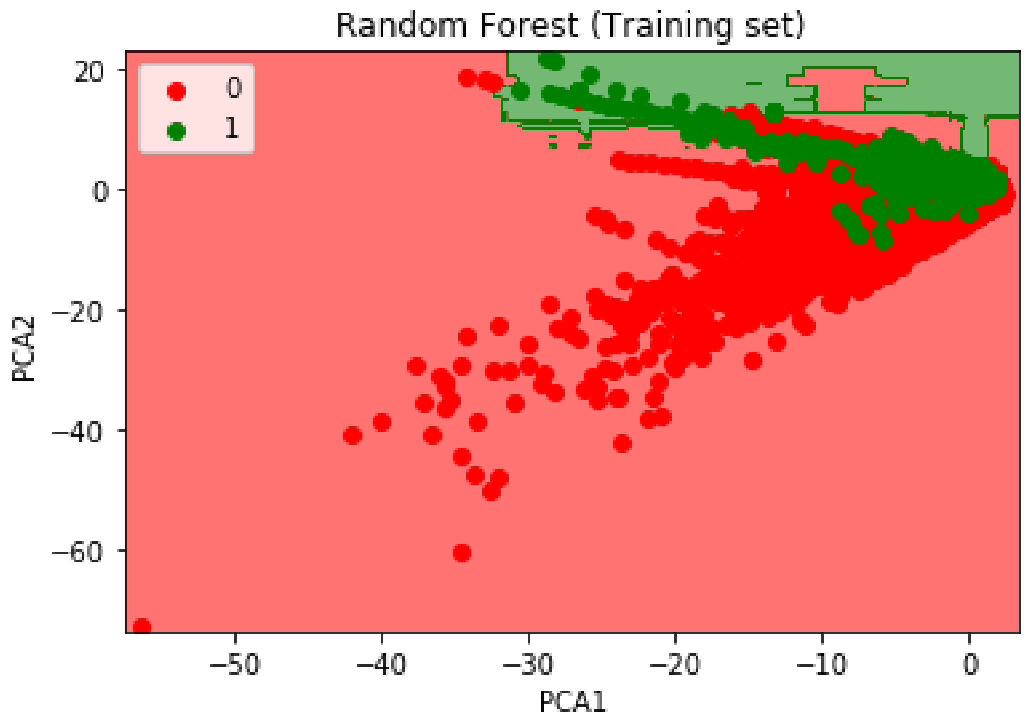
Out[20]:
```
numpy.ndarray
```

In [21]:
```
type(y_train_g)
```

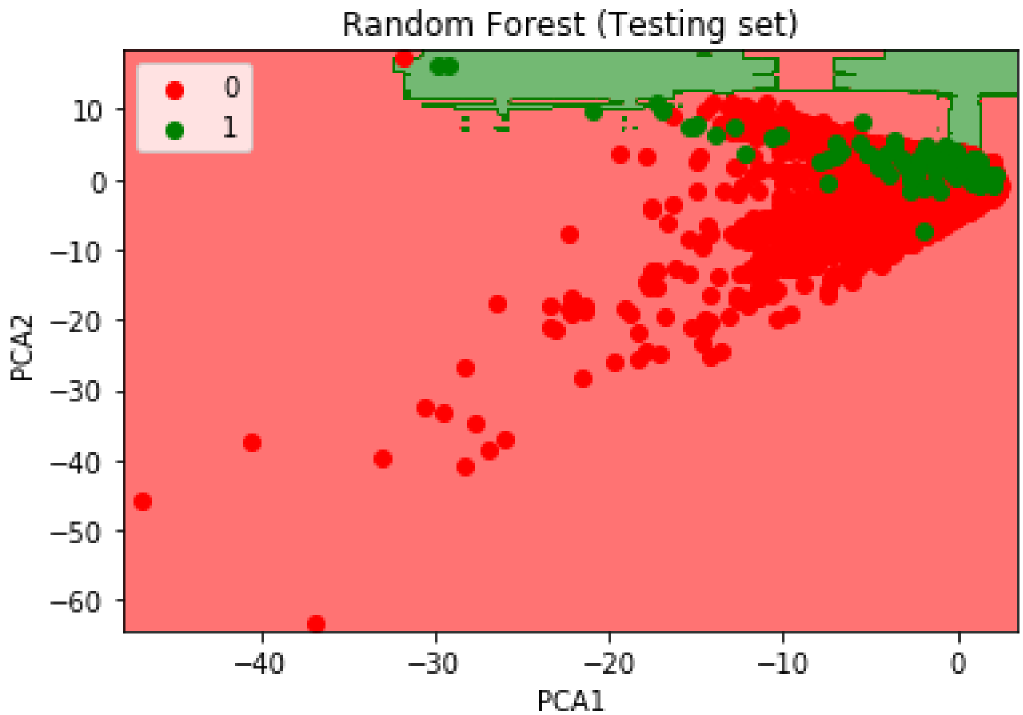
Out[21]:
```
numpy.ndarray
```

In [22]:
```
# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train_g, y_train_g
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max(
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max(
plt.contourf(X1, X2, rf_model_g.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape
             alpha = 0.55, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Random Forest (Training set)')
```

```
plt.xlabel('PCA1')
plt.ylabel('PCA2')
plt.legend()
plt.show()
```




```
In [23]: # Visualising the Test set results
         from matplotlib.colors import ListedColormap
         X_set, y_set = X_test_g, y_test_g
         X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max(
                              np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max(
         plt.contourf(X1, X2, rf_model_g.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape
                      alpha = 0.55, cmap = ListedColormap(('red', 'green')))
         plt.xlim(X1.min(), X1.max())
         plt.ylim(X2.min(), X2.max())
         for i, j in enumerate(np.unique(y_set)):
             plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                         c = ListedColormap(('red', 'green'))(i), label = j)
         plt.title('Random Forest (Testing set)')
         plt.xlabel('PCA1')
         plt.ylabel('PCA2')
         plt.legend()
         plt.show()
```



RF with CondensedNearestNeighbour - undersampling

```
In [21]: from imblearn.under_sampling import CondensedNearestNeighbour
```

```
In [22]: #fit cnn
         cnn = CondensedNearestNeighbour(n_neighbors=5, random_state = 1)
         X_train_cnn5, y_train_cnn5 = cnn.fit_sample(X_train, y_train)
```

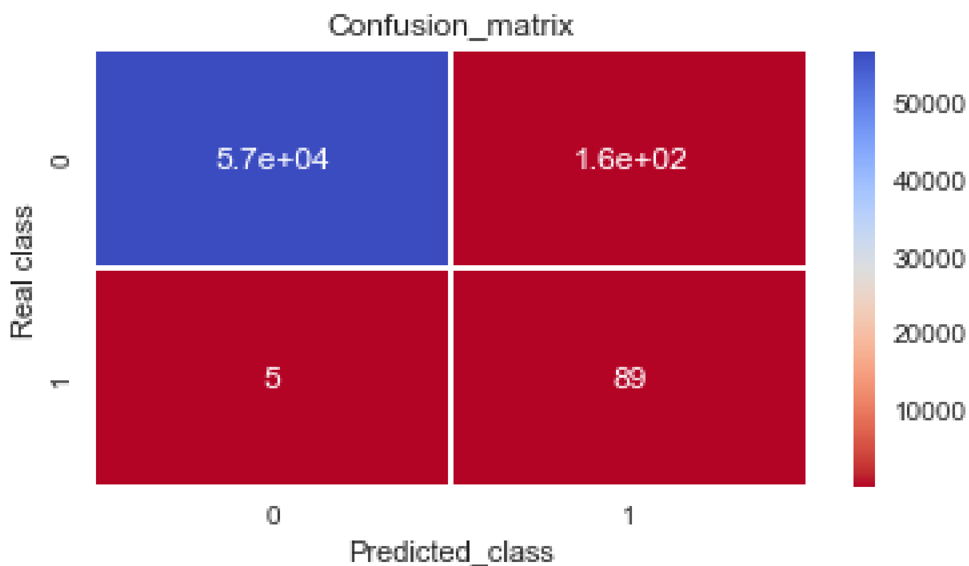
```
In [23]: print('Original Training set & Condensed Nearest Neighbor Sample set:', [len(y_train)
```

```
Original Training set & Condensed Nearest Neighbor Sample set: [227845, 1396]
```

```
In [24]: rf_params = {
             'n_estimators' : np.arange(50, 150, 10),
             'max_depth': np.arange(5, 7, 1),
             'min_samples_split' : [2, 3],
         }
```

```
In [25]: rf_1 = RandomForestClassifier(n_jobs=-1)
         rf_grid_1 = GridSearchCV(rf_1, param_grid=rf_params)
         rf_model_1 = rf_grid_1.fit(X_train_cnn5, y_train_cnn5)

         rf_pred_1 = rf_model_1.predict(X_test)

         model_scores(y_test, rf_pred_1)
```

```
Accuracy Score: 0.9970857764825674

Average Precision Score: 0.3371516076040242

Average Recall: 0.9468085106382979

Average F1 Score: 0.5174418604651163
```




```
----------Classification Report------------------------------------
             precision    recall  f1-score   support

          0       1.00      1.00      1.00     56868
          1       0.36      0.95      0.52        94

avg / total       1.00      1.00      1.00     56962
```

RF with SMOTE - oversampling

```
In [11]: from imblearn.over_sampling import SMOTE
         import winsound

In [27]: sm = SMOTE(random_state=1)
         X_train_sm, y_train_sm = sm.fit_sample(X_train, y_train)

In [28]: print('Original Training set & SMOTE set:', [len(y_train), len(y_train_sm)])
```

```
Original Training set & SMOTE set: [227845, 454894]
```

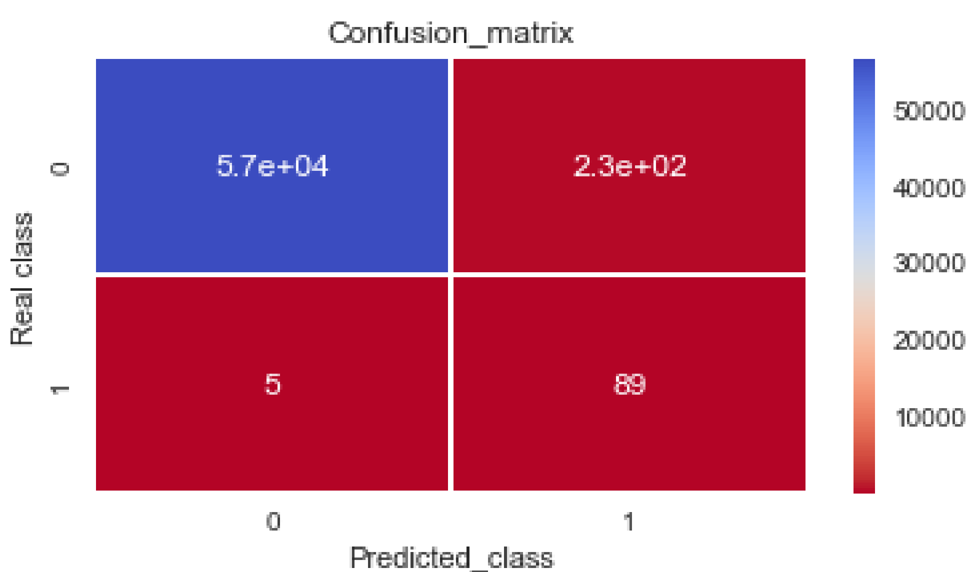
```
In [29]: #rf_params = {
         #    'n_estimators' : np.arange(100, 150, 10),
         #    'max_depth': np.arange(5, 10, 1),
         #    'min_samples_split' : [2, 3],
         #}
```

```
In [33]: rf_params = {
             'n_estimators' : [100, 150],
             'max_depth': [5, 7],
             'min_samples_split' : [2, 3],
         }
```

```
In [34]: rfs = RandomForestClassifier()
         rfs_grid = GridSearchCV(rfs, param_grid=rf_params, n_jobs=-1)
         rfs_model = rfs_grid.fit(X_train_sm, y_train_sm)

         rfs_pred = rfs_model.predict(X_test)
         #winsound.Beep(500,1000)
```

```
In [36]: model_scores(y_test, rfs_pred)
         #winsound.Beep(500,10000)
```

```
Accuracy Score: 0.9958919981742214

Average Precision Score: 0.26507506538537035

Average Recall: 0.9468085106382979

Average F1 Score: 0.4320388349514563
```



```
----------Classification Report------------------------------------
             precision    recall  f1-score   support

          0       1.00      1.00      1.00     56868
          1       0.28      0.95      0.43        94

avg / total       1.00      1.00      1.00     56962
```

Support Vector Classifier

Is a form of support vectore machines, which are very effective in high dimensional spaces and is memory effecient by using a subset of training points in the decision function. The classifier uses the same type of kernel function.

If it is radial basis function then it uses an activation function to project n-dimensions of feature space. It then tries to optimize the boundaries on either side of the decision line with an expected error value. It can also include gradient descent where it attempts to fit coefficient weights by the finding the optimum local minimum.

The linear function does not transorm the data into n-dimesions but it attempts to create a linear line through the data. If there are multiple classes then it becomes a one-verse-all method.

Parameters:

C: is the penalty parameter, which trades off misclassification. A low C makes the decision surface smooth, white high C aims at classifying all training examples correctly. This allows the model to select more or less samples as support vectors.

kernel: is the type of kernel function or algorthim that would be applied to svc. For instance if it is linear there will be no activation function applied, which in other words it would not bring the feature space to n-dimension transformation.

gamma: shows how far the influence of the training data point reaches. Low values mean far and high values mean close.

step_out: step size when extrapolating, used with kind:svm. Extrapolate means to estimate something by assuming that the current method will remain applicable for further instances outside of data scope.

kind: the type of SMOTE algorithm, which would be 'regular', 'svc', 'borderline1', or 'borderline2'.

```
In [25]: #imports
         from sklearn.svm import SVC
```

2.14.1 SVC without undersampling/ oversampling

```
In [25]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=
```

```
In [26]: svc_params = {
              'C': np.arange(0.1, 5, 0.5),
              'kernel': ['rbf'],
```

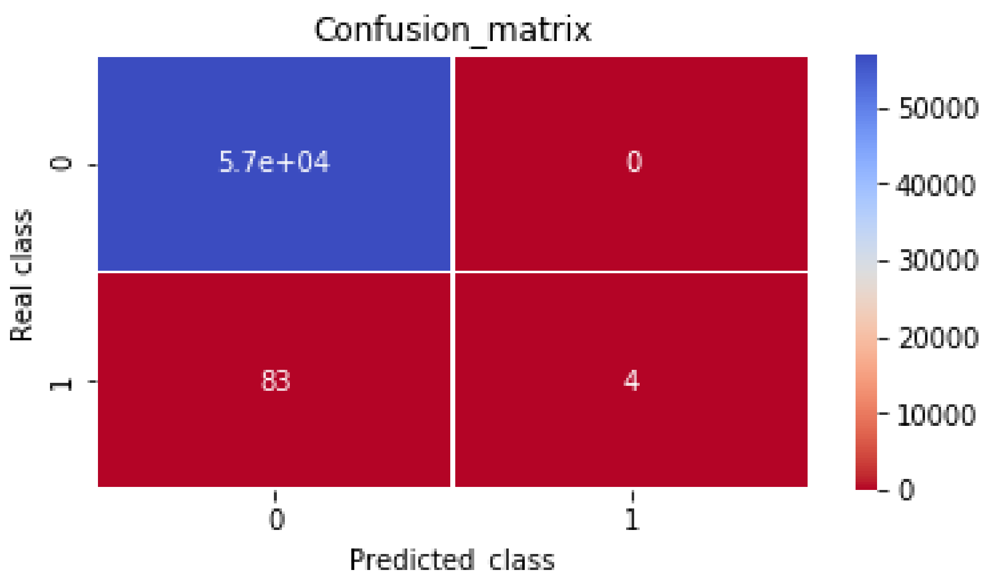
```
            'gamma': np.arange(0.1, 5, 0.5),
            'max_iter':[1000]
        }
```

In [27]:
```
svc_clf = SVC()
svc_grid = GridSearchCV(svc_clf, param_grid=svc_params)
svc_model = svc_grid.fit(X_train, y_train)
```

In [28]:
```
svc_pred = svc_model.predict(X_test)
```

In [29]:
```
model_scores(y_test, svc_pred)
```

```
Accuracy Score: 0.9985428882412837

Average Precision Score: 0.04743412325296921

Average Recall: 0.04597701149425287

Average F1 Score: 0.08791208791208792
```




```
----------Classification Report------------------------------------
             precision    recall  f1-score   support

          0       1.00      1.00      1.00     56875
          1       1.00      0.05      0.09        87

avg / total       1.00      1.00      1.00     56962
```

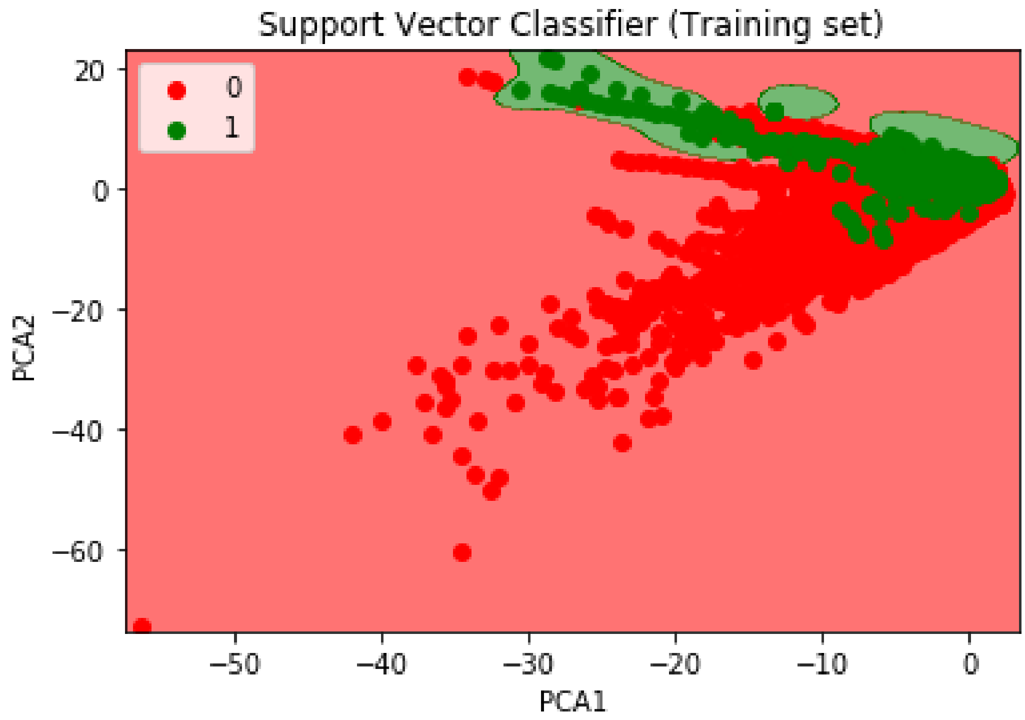
2.14.2 Graph

```
In [30]: #Only use first two PCA variables for plot
         y_g = df['Class']
         X_g = df.iloc[:,1:3]

         X_train_g, X_test_g, y_train_g, y_test_g = train_test_split(X_g, y_g, test_size=0.2, r
```

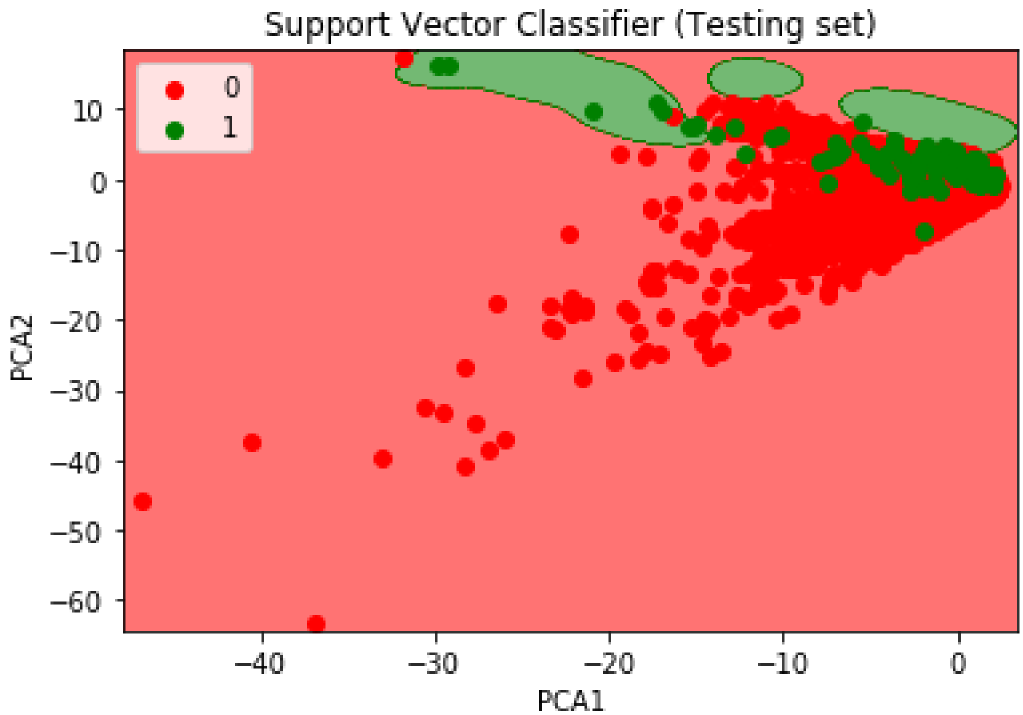
```
In [31]: #SVR with only first two PCA variables for plot
         svc_params = {
             'C': np.arange(0.1, 5, 0.5),
             'kernel': ['rbf'],
             'gamma': np.arange(0.1, 5, 0.5),
             'max_iter':[1000]
         }

         svc_g = SVC()
         svc_grid_g = GridSearchCV(svc_g, param_grid=svc_params)
         svc_model_g = svc_grid_g.fit(X_train_g, y_train_g)
```

```
In [32]: #convert from dataframe to arrays
         X_train_g = X_train_g.values
         X_test_g = X_test_g.values
         y_train_g = y_train_g.values
         y_test_g = y_test_g.values
```

```
In [33]: # Visualising the Training set results
         from matplotlib.colors import ListedColormap
         X_set, y_set = X_train_g, y_train_g
         X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max(
                              np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max(
         plt.contourf(X1, X2, svc_model_g.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape
                      alpha = 0.55, cmap = ListedColormap(('red', 'green')))
         plt.xlim(X1.min(), X1.max())
         plt.ylim(X2.min(), X2.max())
         for i, j in enumerate(np.unique(y_set)):
             plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                         c = ListedColormap(('red', 'green'))(i), label = j)
         plt.title('Support Vector Classifier (Training set)')
         plt.xlabel('PCA1')
         plt.ylabel('PCA2')
         plt.legend()
         plt.show()
```



In [34]:
```
# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test_g, y_test_g
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max(
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max(
plt.contourf(X1, X2, svc_model_g.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape
             alpha = 0.55, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Support Vector Classifier (Testing set)')
plt.xlabel('PCA1')
plt.ylabel('PCA2')
plt.legend()
plt.show()
```



SVC with CondensedNearestNeighbour - undersampling

```
In [8]: from imblearn.under_sampling import CondensedNearestNeighbour
        cnn = CondensedNearestNeighbour(n_neighbors=10, random_state = 613, n_jobs=-1)
        X_train_cnn10, y_train_cnn10 = cnn.fit_sample(X_train, y_train)

        #winsound.Beep(500,10000)
```

```
In [9]: print('Original Training set & Condensed Nearest Neighbor Sample set:', [len(y_train),
```

```
Original Training set & Condensed Nearest Neighbor Sample set: [227845, 1045]
```

```
In [10]: X_train_cnn10=pd.DataFrame(data=X_train_cnn10)
```

```
In [11]: X_train_cnn10.head()
```

```
Out[11]:         0         1         2         3         4         5         6  \
         0  151384.0 -0.098670  1.118022 -1.419248 -0.280080  0.667973 -0.824589
         1  120121.0  1.877828  0.421871 -0.631872  3.728578  0.536681  0.589198
         2  113513.0  0.025692  0.423046 -1.231143 -1.893868  3.320716  3.262109
         3   57232.0  1.163919  0.228664  0.139728  0.484538 -0.220504 -1.031357
         4   67608.0 -0.295570 -0.484728  2.835961 -0.336874 -1.442787  0.998449

                 7         8         9  ...        20        21        22        23  \
```

```
0  0.658272  0.177229  0.054906  ...  -0.296783  0.266887  0.708125 -0.172176
1 -0.021479  0.126885 -0.914571  ...  -0.270228  0.123737  0.470545  0.110416
2  0.671243  0.355486  0.528778  ...   0.097613  0.218512  0.980139 -0.307353
3  0.401498 -0.227066 -0.539325  ...  -0.001633 -0.293428 -0.967329  0.125161
4 -1.271316  0.601446 -0.194068  ...   0.162613  0.345339  1.331306 -0.149416

         24        25        26        27        28    29
0 -1.110173 -0.235229 -0.096557 -0.123233  0.010421  35.0
1  0.760153  0.118625  0.041352 -0.034675 -0.053506   5.3
2  0.714692  0.128959 -0.537229 -0.291759 -0.376752   1.0
3  0.538190  0.233182  0.103396 -0.070322  0.008016  38.9
4  0.288751 -0.486431  0.027122  0.241931  0.142747   2.0

[5 rows x 30 columns]
```

```
In [12]: X_train_cnn10.columns = ['Time', 'V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9'
```

```
In [ ]: #X_train_cnn10_1 = pd.DataFrame(X_train_cnn10,columns='Time V1 V2 V3 V4 V5 V6 V7 V8 V9
```

```
In [13]: X_train_cnn10.head()
```

```
Out[13]:       Time        V1        V2        V3        V4        V5        V6  \
0  151384.0 -0.098670  1.118022 -1.419248 -0.280080  0.667973 -0.824589
1  120121.0  1.877828  0.421871 -0.631872  3.728578  0.536681  0.589198
2  113513.0  0.025692  0.423046 -1.231143 -1.893868  3.320716  3.262109
3   57232.0  1.163919  0.228664  0.139728  0.484538 -0.220504 -1.031357
4   67608.0 -0.295570 -0.484728  2.835961 -0.336874 -1.442787  0.998449

         V7        V8        V9  ...        V20       V21       V22  \
0  0.658272  0.177229  0.054906  ...  -0.296783  0.266887  0.708125
1 -0.021479  0.126885 -0.914571  ...  -0.270228  0.123737  0.470545
2  0.671243  0.355486  0.528778  ...   0.097613  0.218512  0.980139
3  0.401498 -0.227066 -0.539325  ...  -0.001633 -0.293428 -0.967329
4 -1.271316  0.601446 -0.194068  ...   0.162613  0.345339  1.331306

        V23       V24       V25       V26       V27       V28  Amount
0 -0.172176 -1.110173 -0.235229 -0.096557 -0.123233  0.010421    35.0
1  0.110416  0.760153  0.118625  0.041352 -0.034675 -0.053506     5.3
2 -0.307353  0.714692  0.128959 -0.537229 -0.291759 -0.376752     1.0
3  0.125161  0.538190  0.233182  0.103396 -0.070322  0.008016    38.9
4 -0.149416  0.288751 -0.486431  0.027122  0.241931  0.142747     2.0

[5 rows x 30 columns]
```

```
In [14]: X_train_cnn10.set_index('Time', inplace=True)
```

```
In [15]: X_train_cnn10.head()
```

```
Out[15]:             V1        V2        V3        V4        V5        V6  \
Time
```

```
151384.0 -0.098670  1.118022 -1.419248 -0.280080  0.667973 -0.824589
120121.0  1.877828  0.421871 -0.631872  3.728578  0.536681  0.589198
113513.0  0.025692  0.423046 -1.231143 -1.893868  3.320716  3.262109
57232.0   1.163919  0.228664  0.139728  0.484538 -0.220504 -1.031357
67608.0  -0.295570 -0.484728  2.835961 -0.336874 -1.442787  0.998449

                V7        V8        V9       V10   ...          V20       V21  \
Time                                               ...
151384.0  0.658272  0.177229  0.054906 -1.090686   ...    -0.296783  0.266887
120121.0 -0.021479  0.126885 -0.914571  1.541791   ...    -0.270228  0.123737
113513.0  0.671243  0.355486  0.528778  0.129648   ...     0.097613  0.218512
57232.0   0.401498 -0.227066 -0.539325  0.083866   ...    -0.001633 -0.293428
67608.0  -1.271316  0.601446 -0.194068  0.231689   ...     0.162613  0.345339

               V22       V23       V24       V25       V26       V27  \
Time
151384.0  0.708125 -0.172176 -1.110173 -0.235229 -0.096557 -0.123233
120121.0  0.470545  0.110416  0.760153  0.118625  0.041352 -0.034675
113513.0  0.980139 -0.307353  0.714692  0.128959 -0.537229 -0.291759
57232.0  -0.967329  0.125161  0.538190  0.233182  0.103396 -0.070322
67608.0   1.331306 -0.149416  0.288751 -0.486431  0.027122  0.241931

               V28  Amount
Time
151384.0  0.010421    35.0
120121.0 -0.053506     5.3
113513.0 -0.376752     1.0
57232.0   0.008016    38.9
67608.0   0.142747     2.0

[5 rows x 29 columns]
```

In [16]:
```
del X_train_cnn10.index.name
```

In [17]:
```
X_train_cnn10.head()
```

Out[17]:
```
                V1        V2        V3        V4        V5        V6  \
151384.0 -0.098670  1.118022 -1.419248 -0.280080  0.667973 -0.824589
120121.0  1.877828  0.421871 -0.631872  3.728578  0.536681  0.589198
113513.0  0.025692  0.423046 -1.231143 -1.893868  3.320716  3.262109
57232.0   1.163919  0.228664  0.139728  0.484538 -0.220504 -1.031357
67608.0  -0.295570 -0.484728  2.835961 -0.336874 -1.442787  0.998449

                V7        V8        V9       V10   ...          V20       V21  \
151384.0  0.658272  0.177229  0.054906 -1.090686   ...    -0.296783  0.266887
120121.0 -0.021479  0.126885 -0.914571  1.541791   ...    -0.270228  0.123737
113513.0  0.671243  0.355486  0.528778  0.129648   ...     0.097613  0.218512
57232.0   0.401498 -0.227066 -0.539325  0.083866   ...    -0.001633 -0.293428
```

```
67608.0  -1.271316  0.601446 -0.194068  0.231689   ...    0.162613  0.345339

               V22       V23       V24       V25       V26       V27  \
151384.0  0.708125 -0.172176 -1.110173 -0.235229 -0.096557 -0.123233
120121.0  0.470545  0.110416  0.760153  0.118625  0.041352 -0.034675
113513.0  0.980139 -0.307353  0.714692  0.128959 -0.537229 -0.291759
57232.0  -0.967329  0.125161  0.538190  0.233182  0.103396 -0.070322
67608.0   1.331306 -0.149416  0.288751 -0.486431  0.027122  0.241931

               V28  Amount
151384.0  0.010421    35.0
120121.0 -0.053506     5.3
113513.0 -0.376752     1.0
57232.0   0.008016    38.9
67608.0   0.142747     2.0

[5 rows x 29 columns]
```

In [18]:
```
y_train_cnn10
```

Out[18]:
```
array([0, 0, 0, ..., 1, 1, 1], dtype=int64)
```

In [19]:
```
y_train_cnn10.shape
```

Out[19]:
```
(1045,)
```

In [ ]:
```
#y_train_cnn5_1 = np.delete(y_train_cnn5, [0])
```

In [20]:
```
X_test.shape
```

Out[20]:
```
(56962, 30)
```

In [21]:
```
X_test.drop('Time', axis=1, inplace=True)
```

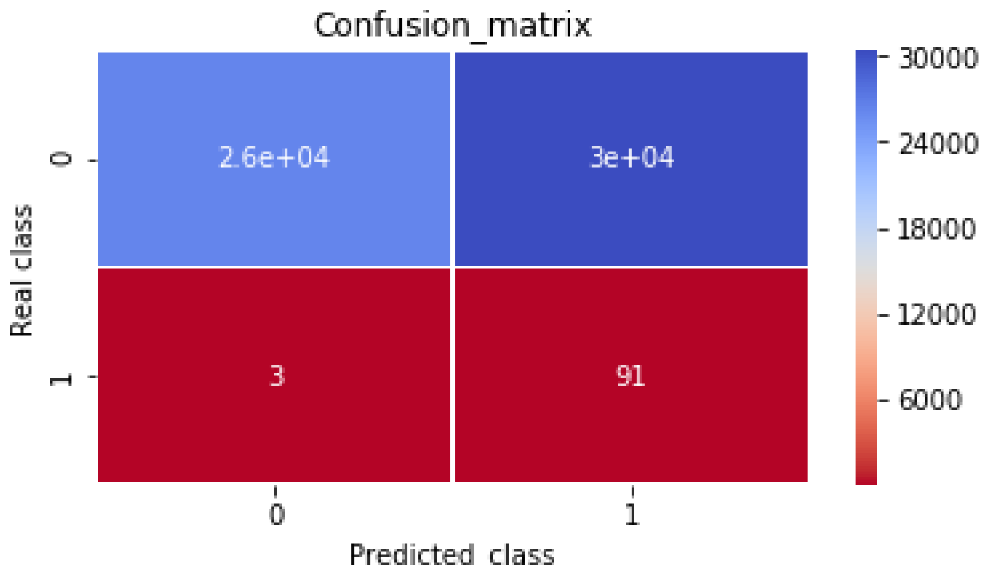
In [22]:
```
X_test.shape
```

Out[22]:
```
(56962, 29)
```

In [23]:
```
svc_params = {
    'C': [0.5, 1, 1.5, 2],
    'kernel': ['rbf', 'linear'],
    'gamma': [1, 3, 5],
    'max_iter':[1000]
}
```

In [28]:
```
svc = SVC()
svc_grid = GridSearchCV(svc, param_grid=svc_params, n_jobs=-1)
svc_model_cnn10 = svc_grid.fit(X_train_cnn10, y_train_cnn10)

svc_pred_cnn10 = svc_model_cnn10.predict(X_test)

model_scores(y_test, svc_pred_cnn10)
#winsound.Beep(500,10000)
```

```
Accuracy Score: 0.4648888732839437

Average Precision Score: 0.0029345321315622282

Average Recall: 0.9680851063829787

Average F1 Score: 0.0059354922871212865
```




```
----------Classification Report------------------------------------
             precision    recall  f1-score   support

          0       1.00      0.46      0.63     56868
          1       0.00      0.97      0.01        94

avg / total       1.00      0.46      0.63     56962
```

SVC with SMOTE - oversampling

```
In [13]: from imblearn.over_sampling import SMOTE
         from sklearn.svm import SVC

In [38]: sm = SMOTE(random_state=1)
         X_train_smote, y_train_smote = sm.fit_sample(X_train, y_train)
```

In [39]:
```
svc_params_2 = {
    'C': [0.5, 1, 1.5, 2],
    'kernel': ['rbf', 'linear'],
    'gamma': [1, 3, 5],
    'max_iter':[1000]
}
```

In [41]:
```
type(X_train_smote)
```

Out[41]:
```
numpy.ndarray
```

In [ ]:
```
#X_train_smote_1 = X_train_smote.set_index(0)
```

In [ ]:
```
#X_train_smote_1.head()
```

In [ ]:
```
#X_train_smote_1.columns = ['V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10
```

In [ ]:
```
#del X_train_smote_1.index.name
```

In [ ]:
```
#X_train_smote_1.head()
```

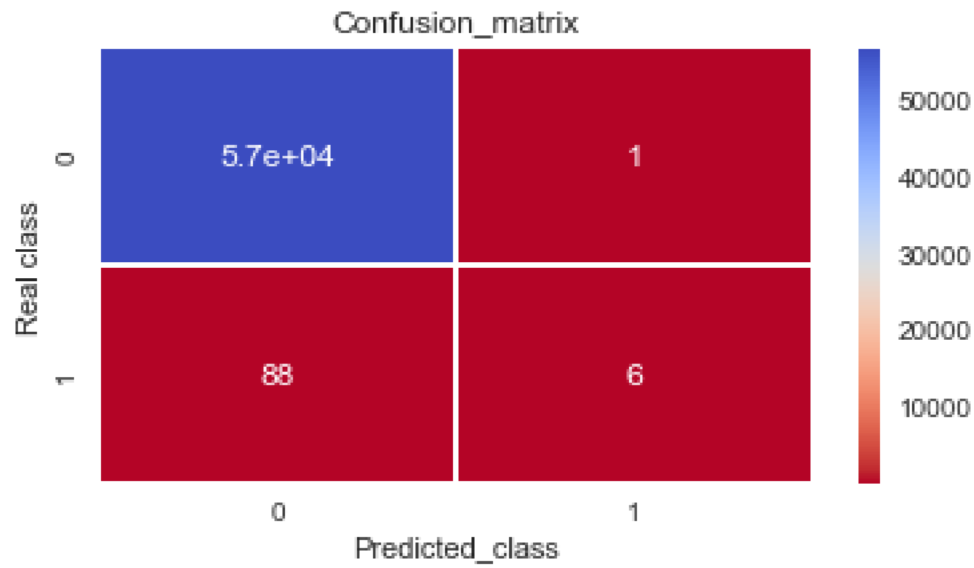
In [ ]:
```
#X_train_smote_1.shape
```

In [ ]:
```
#X_test.shape
```

In [45]:
```
svc = SVC()
svc_grid_smote = GridSearchCV(svc, param_grid=svc_params_2, n_jobs=-1)
svc_model_smote = svc_grid_smote.fit(X_train_smote, y_train_smote)

svc_pred_smote = svc_model_smote.predict(X_test)

#winsound.Beep(500,10000)
```

In [46]:
```
model_scores(y_test, svc_pred_smote)
```

```
Accuracy Score: 0.9984375548611355

Average Precision Score: 0.05625613577611437

Average Recall: 0.06382978723404255

Average F1 Score: 0.1188118811881188
```




```
----------Classification Report------------------------------------
             precision    recall  f1-score   support

          0       1.00      1.00      1.00     56868
          1       0.86      0.06      0.12        94

avg / total       1.00      1.00      1.00     56962
```

XGBoost

'Extreme Gradient Boosting' is another ensemble method that can handle both regression and classification. XGBoost is known for its speed and model performance. New models are added to the original to correct errors made by the original. Gradient boosting creates new models that predict the errors of the previous model and add them together for the final prediction. XGBoost uses gradient descent algorithm to minimize the loss when adding the new models. Gradient Descent is an itterative optimization algorithm that uses learning rate to find the optimal local minimum.

```
In [14]: from xgboost.sklearn import XGBClassifier
```

```
C:\Users\Y\Anaconda3\lib\site-packages\sklearn\cross_validation.py:41: DeprecationWarning: Thi
  "This module will be removed in 0.20.", DeprecationWarning)
```

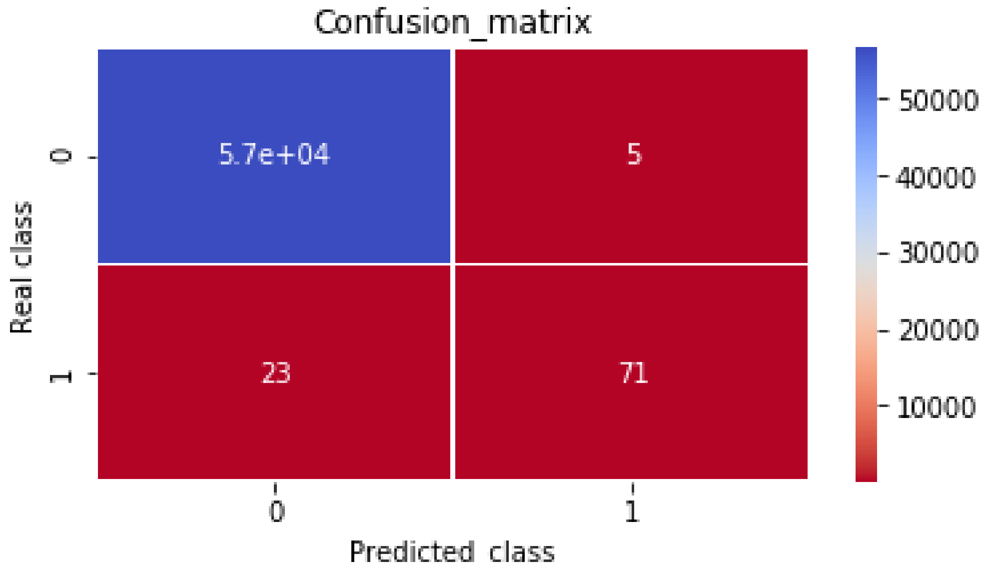
```
In [15]: import numpy as np
```

```
In [19]: y = df['Class']
         X = df.iloc[:,:-1]

         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=
```

XGBoost without undersampling/ oversampling

```
In [16]: #xg_params = {
         #    'colsample_bytree': [0.2],
         #    'gamma':[0.01],
         #    'learning_rate':[0.001, 0.01],
         #    'max_depth':np.arange(3,7,1),
         #    'n_estimators':[4000, 5000],
         #    'reg_alpha':[0.5,0.9],
         #    'reg_lambda':[0.3, 0.4, 0.5],
         #    'subsample': [0.2]
         #}

In [32]: xg_params = {
             'colsample_bytree': [0.2],
             'learning_rate':[0.001, 0.01],
             'gamma':[0.01],
             'max_depth':np.arange(3,4,1),
             'n_estimators':[100,200],
             'reg_alpha':[0.75],
             'reg_lambda':[0.4],
         }

In [33]: xgb_clf = XGBClassifier()

In [34]: xgb_clf = XGBClassifier()
         xgb_grid = GridSearchCV(xgb_clf, param_grid=xg_params, n_jobs = -1)
         xgb_model = xgb_grid.fit(X_train, y_train)

         xgb_pred = xgb_model.predict(X_test)

         model_scores(y_test, xgb_pred)

         #winsound.Beep(500,10000)

Accuracy Score: 0.9995084442259752

Average Precision Score: 0.7060308776212885

Average Recall: 0.7553191489361702

Average F1 Score: 0.8352941176470589
```

```
C:\Users\Y\Anaconda3\lib\site-packages\sklearn\preprocessing\label.py:151: DeprecationWarning:
  if diff:
```




```
----------Classification Report------------------------------------
             precision    recall  f1-score   support

          0       1.00      1.00      1.00     56868
          1       0.93      0.76      0.84        94

avg / total       1.00      1.00      1.00     56962
```

XGBoost with CondensedNearestNeighbour - undersampling

In [35]:
```
y = df['Class']
X = df.iloc[:,:-1]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=
```

In [36]:
```
X_train.head()
```

Out[36]:
```
            Time        V1        V2        V3        V4        V5        V6  \
170318  120121.0  1.877828  0.421871 -0.631872  3.728578  0.536681  0.589198
160639  113513.0  0.025692  0.423046 -1.231143 -1.893868  3.320716  3.262109
77827    57232.0  1.163919  0.228664  0.139728  0.484538 -0.220504 -1.031357
```

```
100739   67608.0 -0.295570 -0.484728  2.835961 -0.336874 -1.442787  0.998449
19537    30364.0  1.109071  0.061282  0.622337  1.452091 -0.178357  0.351704

              V7        V8        V9    ...          V20       V21       V22  \
170318 -0.021479  0.126885 -0.914571    ...    -0.270228  0.123737  0.470545
160639  0.671243  0.355486  0.528778    ...     0.097613  0.218512  0.980139
77827   0.401498 -0.227066 -0.539325    ...    -0.001633 -0.293428 -0.967329
100739 -1.271316  0.601446 -0.194068    ...     0.162613  0.345339  1.331306
19537  -0.285974  0.091020 -0.677818    ...     0.124986  0.119724  0.202824

             V23       V24       V25       V26       V27       V28  Amount
170318  0.110416  0.760153  0.118625  0.041352 -0.034675 -0.053506    5.30
160639 -0.307353  0.714692  0.128959 -0.537229 -0.291759 -0.376752    1.00
77827   0.125161  0.538190  0.233182  0.103396 -0.070322  0.008016   38.90
100739 -0.149416  0.288751 -0.486431  0.027122  0.241931  0.142747    2.00
19537  -0.174770 -0.421437  0.301579  1.024982 -0.068117  0.009277   62.94

[5 rows x 30 columns]
```

In [39]:
```
cnn = CondensedNearestNeighbour(n_neighbors=10, random_state = 613, n_jobs=-1)
X_train_cnn10, y_train_cnn10 = ckk.fit_sample(X_train, y_train)

winsound.Beep(500,10000)
```

In [68]:
```
X_train_cnn10_1 = pd.DataFrame(X_train_cnn10,columns='Time V1 V2 V3 V4 V5 V6 V7 V8 V9
```

In [69]:
```
X_train_cnn10_1.head()
```

Out[69]:
```
       Time        V1        V2        V3        V4        V5        V6  \
0  151384.0 -0.098670  1.118022 -1.419248 -0.280080  0.667973 -0.824589
1  120121.0  1.877828  0.421871 -0.631872  3.728578  0.536681  0.589198
2  113513.0  0.025692  0.423046 -1.231143 -1.893868  3.320716  3.262109
3   57232.0  1.163919  0.228664  0.139728  0.484538 -0.220504 -1.031357
4   67608.0 -0.295570 -0.484728  2.835961 -0.336874 -1.442787  0.998449

         V7        V8        V9    ...          V20       V21       V22  \
0  0.658272  0.177229  0.054906    ...    -0.296783  0.266887  0.708125
1 -0.021479  0.126885 -0.914571    ...    -0.270228  0.123737  0.470545
2  0.671243  0.355486  0.528778    ...     0.097613  0.218512  0.980139
3  0.401498 -0.227066 -0.539325    ...    -0.001633 -0.293428 -0.967329
4 -1.271316  0.601446 -0.194068    ...     0.162613  0.345339  1.331306

        V23       V24       V25       V26       V27       V28  Amount
0 -0.172176 -1.110173 -0.235229 -0.096557 -0.123233  0.010421    35.0
1  0.110416  0.760153  0.118625  0.041352 -0.034675 -0.053506     5.3
2 -0.307353  0.714692  0.128959 -0.537229 -0.291759 -0.376752     1.0
3  0.125161  0.538190  0.233182  0.103396 -0.070322  0.008016    38.9
4 -0.149416  0.288751 -0.486431  0.027122  0.241931  0.142747     2.0
```

```
[5 rows x 30 columns]
```

```
In [74]: X_train_cnn10_2 = X_train_cnn10_1.set_index(['Time'])

In [75]: X_train_cnn10_2.head()
```

```
Out[75]:                 V1        V2        V3        V4        V5        V6  \
         Time
         151384.0 -0.098670  1.118022 -1.419248 -0.280080  0.667973 -0.824589
         120121.0  1.877828  0.421871 -0.631872  3.728578  0.536681  0.589198
         113513.0  0.025692  0.423046 -1.231143 -1.893868  3.320716  3.262109
         57232.0   1.163919  0.228664  0.139728  0.484538 -0.220504 -1.031357
         67608.0  -0.295570 -0.484728  2.835961 -0.336874 -1.442787  0.998449

                        V7        V8        V9       V10    ...          V20       V21  \
         Time                                               ...
         151384.0  0.658272  0.177229  0.054906 -1.090686   ...    -0.296783  0.266887
         120121.0 -0.021479  0.126885 -0.914571  1.541791   ...    -0.270228  0.123737
         113513.0  0.671243  0.355486  0.528778  0.129648   ...     0.097613  0.218512
         57232.0   0.401498 -0.227066 -0.539325  0.083866   ...    -0.001633 -0.293428
         67608.0  -1.271316  0.601446 -0.194068  0.231689   ...     0.162613  0.345339

                       V22       V23       V24       V25       V26       V27  \
         Time
         151384.0  0.708125 -0.172176 -1.110173 -0.235229 -0.096557 -0.123233
         120121.0  0.470545  0.110416  0.760153  0.118625  0.041352 -0.034675
         113513.0  0.980139 -0.307353  0.714692  0.128959 -0.537229 -0.291759
         57232.0  -0.967329  0.125161  0.538190  0.233182  0.103396 -0.070322
         67608.0   1.331306 -0.149416  0.288751 -0.486431  0.027122  0.241931

                       V28  Amount
         Time
         151384.0  0.010421    35.0
         120121.0 -0.053506     5.3
         113513.0 -0.376752     1.0
         57232.0   0.008016    38.9
         67608.0   0.142747     2.0

         [5 rows x 29 columns]
```

```
In [76]: del X_train_cnn10_2.index.name

In [77]: X_train_cnn10_2.head()
```

```
Out[77]:                 V1        V2        V3        V4        V5        V6  \
         151384.0 -0.098670  1.118022 -1.419248 -0.280080  0.667973 -0.824589
         120121.0  1.877828  0.421871 -0.631872  3.728578  0.536681  0.589198
         113513.0  0.025692  0.423046 -1.231143 -1.893868  3.320716  3.262109
```

```
57232.0    1.163919  0.228664  0.139728  0.484538 -0.220504 -1.031357
67608.0   -0.295570 -0.484728  2.835961 -0.336874 -1.442787  0.998449

                V7        V8        V9       V10   ...       V20       V21  \
151384.0  0.658272  0.177229  0.054906 -1.090686   ...  -0.296783  0.266887
120121.0 -0.021479  0.126885 -0.914571  1.541791   ...  -0.270228  0.123737
113513.0  0.671243  0.355486  0.528778  0.129648   ...   0.097613  0.218512
57232.0   0.401498 -0.227066 -0.539325  0.083866   ...  -0.001633 -0.293428
67608.0  -1.271316  0.601446 -0.194068  0.231689   ...   0.162613  0.345339

               V22       V23       V24       V25       V26       V27  \
151384.0  0.708125 -0.172176 -1.110173 -0.235229 -0.096557 -0.123233
120121.0  0.470545  0.110416  0.760153  0.118625  0.041352 -0.034675
113513.0  0.980139 -0.307353  0.714692  0.128959 -0.537229 -0.291759
57232.0  -0.967329  0.125161  0.538190  0.233182  0.103396 -0.070322
67608.0   1.331306 -0.149416  0.288751 -0.486431  0.027122  0.241931

               V28  Amount
151384.0  0.010421    35.0
120121.0 -0.053506     5.3
113513.0 -0.376752     1.0
57232.0   0.008016    38.9
67608.0   0.142747     2.0

[5 rows x 29 columns]
```

In [78]:
```
X_train_cnn10_2.shape
```

Out[78]:
```
(1045, 29)
```

In [ ]:
```
# X_train_cnn10 = pd.DataFrame(data=X_train_cnn10)
```

In [ ]:
```
X_train_cnn10.head()
```

In [ ]:
```
# xg_params = {
#     'colsample_bytree': [0.2],
#     'gamma':[0.01],
#     'learning_rate':[0.001, 0.01],
#     'max_depth':np.arange(3,7,1),
#     'n_estimators':[4000, 5000],
#     'reg_alpha':[0.5,0.9],
#     'reg_lambda':[0.3, 0.4, 0.5],
#     'subsample': [0.2]
# }
```

In [42]:
```
xg_params = {
     'colsample_bytree': [0.2],
     'learning_rate':[0.001, 0.01],
     'gamma':[0.01],
```

```
        'max_depth':np.arange(3,4,1),
        'n_estimators':[100,200],
        'reg_alpha':[0.75],
        'reg_lambda':[0.4],
    }
```

In [48]:
```
X_test.head()
```

Out[48]:
```
                Time        V1        V2        V3        V4        V5        V6  \
19366     30217.0  0.312730 -1.579302  0.037925  0.974769 -1.016948 -0.186730
280869   169799.0  1.563200 -0.624672 -2.234148  0.291392  0.916992  0.812909
119038    75307.0 -0.790818  1.500153  0.150621  0.557415  0.307555 -0.672485
222132   142840.0  2.228171 -1.438665 -0.887874 -1.637291 -1.206383 -0.426009
134028    80615.0 -1.501183  1.586508 -1.212181  0.171627  1.297362  4.061726

              V7        V8        V9    ...          V20       V21       V22  \
19366   0.369821 -0.183192  0.510461    ...     0.928395  0.217170 -0.335091
280869  0.121979  0.168880  0.919114    ...     0.185794 -0.082234 -0.302545
119038  0.369654  0.459281 -1.079754    ...    -0.134711  0.112008  0.152055
222132 -1.153598 -0.105519 -1.443694    ...    -0.375517 -0.144229  0.059692
134028 -0.120086  1.661070 -0.453209    ...     0.056840 -0.075352 -0.192637

             V23       V24       V25       V26       V27       V28  Amount
19366  -0.517359 -0.036104  0.274476  0.456141 -0.108858  0.097648  497.60
280869  0.086185 -0.666717 -0.262277 -0.032024  0.007327  0.008732  198.86
119038 -0.100873  0.096955 -0.454014  0.366937 -0.125195  0.085907    0.76
222132  0.197388 -0.410414 -0.237783 -0.198362  0.013074 -0.058514   32.95
134028 -0.126501  1.020786  0.213743 -0.249811  0.089311  0.089326  121.40

[5 rows x 30 columns]
```

In [63]:
```
y_train_cnn10.shape
```

Out[63]:
```
(1045,)
```

In [58]:
```
# y_train_cnn10_1 = np.delete(y_train_cnn10, [0])
```

In [60]:
```
# y_train_cnn10_1.shape
```

Out[60]:
```
(1044,)
```

In [62]:
```
X_train_cnn10_1.shape
```

Out[62]:
```
(1045, 30)
```

In [52]:
```
X_test.head()
```

Out[52]:
```
                Time        V1        V2        V3        V4        V5        V6  \
19366     30217.0  0.312730 -1.579302  0.037925  0.974769 -1.016948 -0.186730
280869   169799.0  1.563200 -0.624672 -2.234148  0.291392  0.916992  0.812909
```

```
119038   75307.0 -0.790818  1.500153  0.150621  0.557415  0.307555 -0.672485
222132  142840.0  2.228171 -1.438665 -0.887874 -1.637291 -1.206383 -0.426009
134028   80615.0 -1.501183  1.586508 -1.212181  0.171627  1.297362  4.061726

              V7        V8        V9    ...          V20       V21       V22  \
19366   0.369821 -0.183192  0.510461    ...     0.928395  0.217170 -0.335091
280869  0.121979  0.168880  0.919114    ...     0.185794 -0.082234 -0.302545
119038  0.369654  0.459281 -1.079754    ...    -0.134711  0.112008  0.152055
222132 -1.153598 -0.105519 -1.443694    ...    -0.375517 -0.144229  0.059692
134028 -0.120086  1.661070 -0.453209    ...     0.056840 -0.075352 -0.192637

             V23       V24       V25       V26       V27       V28  Amount
19366  -0.517359 -0.036104  0.274476  0.456141 -0.108858  0.097648  497.60
280869  0.086185 -0.666717 -0.262277 -0.032024  0.007327  0.008732  198.86
119038 -0.100873  0.096955 -0.454014  0.366937 -0.125195  0.085907    0.76
222132  0.197388 -0.410414 -0.237783 -0.198362  0.013074 -0.058514   32.95
134028 -0.126501  1.020786  0.213743 -0.249811  0.089311  0.089326  121.40

[5 rows x 30 columns]
```

In [53]:
```
X_test.drop('Time', axis=1, inplace=True)
```

In [54]:
```
X_test.head()
```

Out[54]:
```
              V1        V2        V3        V4        V5        V6        V7  \
19366   0.312730 -1.579302  0.037925  0.974769 -1.016948 -0.186730  0.369821
280869  1.563200 -0.624672 -2.234148  0.291392  0.916992  0.812909  0.121979
119038 -0.790818  1.500153  0.150621  0.557415  0.307555 -0.672485  0.369654
222132  2.228171 -1.438665 -0.887874 -1.637291 -1.206383 -0.426009 -1.153598
134028 -1.501183  1.586508 -1.212181  0.171627  1.297362  4.061726 -0.120086

              V8        V9       V10    ...          V20       V21       V22  \
19366  -0.183192  0.510461 -0.402411    ...     0.928395  0.217170 -0.335091
280869  0.168880  0.919114 -1.139037    ...     0.185794 -0.082234 -0.302545
119038  0.459281 -1.079754 -0.920195    ...    -0.134711  0.112008  0.152055
222132 -0.105519 -1.443694  1.711884    ...    -0.375517 -0.144229  0.059692
134028  1.661070 -0.453209  0.005655    ...     0.056840 -0.075352 -0.192637

             V23       V24       V25       V26       V27       V28  Amount
19366  -0.517359 -0.036104  0.274476  0.456141 -0.108858  0.097648  497.60
280869  0.086185 -0.666717 -0.262277 -0.032024  0.007327  0.008732  198.86
119038 -0.100873  0.096955 -0.454014  0.366937 -0.125195  0.085907    0.76
222132  0.197388 -0.410414 -0.237783 -0.198362  0.013074 -0.058514   32.95
134028 -0.126501  1.020786  0.213743 -0.249811  0.089311  0.089326  121.40

[5 rows x 29 columns]
```

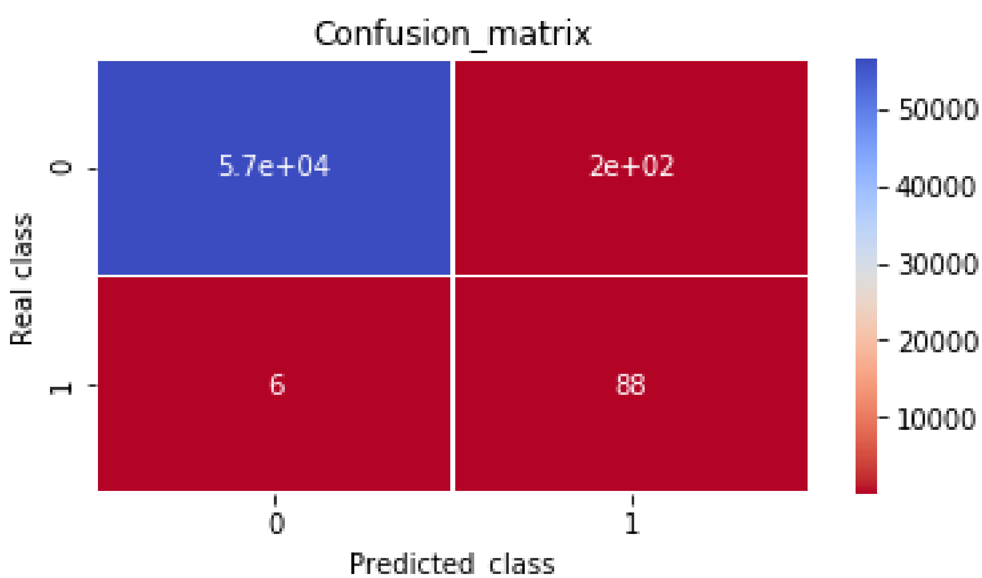
In [55]:
```
y_test.head()
```

```
Out[55]: 19366     0
         280869    0
         119038    0
         222132    0
         134028    0
         Name: Class, dtype: int64
```

```
In [66]: X_train_cnn10_2 = X_train_cnn10_1.drop('Time', axis=1, inplace=True)
```

```
In [79]: xgb = XGBClassifier()
         xgb_grid_cnn10 = GridSearchCV(xgb, param_grid=xg_params, n_jobs=-1)
         xgb_model_cnn10 = xgb_grid_cnn10.fit(X_train_cnn10_2, y_train_cnn10)

         xgb_pred_cnn10 = xgb_model_cnn10.predict(X_test)

         model_scores(y_test, xgb_pred_cnn10)

         #winsound.Beep(500,10000)
```

```
C:\Users\Y\Anaconda3\lib\site-packages\sklearn\preprocessing\label.py:151: DeprecationWarning:
  if diff:


Accuracy Score: 0.9963835539482462

Average Precision Score: 0.286157342836413

Average Recall: 0.9361702127659575

Average F1 Score: 0.46073298429319376
```



```
----------Classification Report------------------------------------
             precision    recall  f1-score   support

          0       1.00      1.00      1.00     56868
          1       0.31      0.94      0.46        94

avg / total       1.00      1.00      1.00     56962
```

XGBoost with SMOTE- oversampling

In [80]:
```
from imblearn.over_sampling import SMOTE
```

In [83]:
```
sm = SMOTE(random_state=613)
X_train_smote, y_train_smote = sm.fit_sample(X_train, y_train)

# winsound.Beep(500,10000)
```

In [84]:
```
X_train_smote = pd.DataFrame(data=X_train_smote)
```

In [85]:
```
X_train_smote.head()
```

Out[85]:
```
          0         1         2         3         4         5         6  \
0  120121.0  1.877828  0.421871 -0.631872  3.728578  0.536681  0.589198
1  113513.0  0.025692  0.423046 -1.231143 -1.893868  3.320716  3.262109
2   57232.0  1.163919  0.228664  0.139728  0.484538 -0.220504 -1.031357
3   67608.0 -0.295570 -0.484728  2.835961 -0.336874 -1.442787  0.998449
4   30364.0  1.109071  0.061282  0.622337  1.452091 -0.178357  0.351704

          7         8         9  ...         20        21        22  \
0 -0.021479  0.126885 -0.914571  ...  -0.270228  0.123737  0.470545
1  0.671243  0.355486  0.528778  ...   0.097613  0.218512  0.980139
2  0.401498 -0.227066 -0.539325  ...  -0.001633 -0.293428 -0.967329
3 -1.271316  0.601446 -0.194068  ...   0.162613  0.345339  1.331306
4 -0.285974  0.091020 -0.677818  ...   0.124986  0.119724  0.202824

         23        24        25        26        27        28     29
0  0.110416  0.760153  0.118625  0.041352 -0.034675 -0.053506   5.30
1 -0.307353  0.714692  0.128959 -0.537229 -0.291759 -0.376752   1.00
2  0.125161  0.538190  0.233182  0.103396 -0.070322  0.008016  38.90
3 -0.149416  0.288751 -0.486431  0.027122  0.241931  0.142747   2.00
4 -0.174770 -0.421437  0.301579  1.024982 -0.068117  0.009277  62.94

[5 rows x 30 columns]
```

In [86]:
```
X_train_smote_1 = X_train_smote.set_index(0)
```

In [87]:
```
X_train_smote_1.head()
```

Out[87]:
```
                 1         2         3         4         5         6   \
0
120121.0  1.877828  0.421871 -0.631872  3.728578  0.536681  0.589198
113513.0  0.025692  0.423046 -1.231143 -1.893868  3.320716  3.262109
57232.0   1.163919  0.228664  0.139728  0.484538 -0.220504 -1.031357
67608.0  -0.295570 -0.484728  2.835961 -0.336874 -1.442787  0.998449
30364.0   1.109071  0.061282  0.622337  1.452091 -0.178357  0.351704

                 7         8         9         10  ...        20        21  \
0                                                  ...
120121.0 -0.021479  0.126885 -0.914571  1.541791  ... -0.270228  0.123737
113513.0  0.671243  0.355486  0.528778  0.129648  ...  0.097613  0.218512
57232.0   0.401498 -0.227066 -0.539325  0.083866  ... -0.001633 -0.293428
67608.0  -1.271316  0.601446 -0.194068  0.231689  ...  0.162613  0.345339
30364.0  -0.285974  0.091020 -0.677818  0.662103  ...  0.124986  0.119724

                 22        23        24        25        26        27  \
0
120121.0  0.470545  0.110416  0.760153  0.118625  0.041352 -0.034675
113513.0  0.980139 -0.307353  0.714692  0.128959 -0.537229 -0.291759
57232.0  -0.967329  0.125161  0.538190  0.233182  0.103396 -0.070322
67608.0   1.331306 -0.149416  0.288751 -0.486431  0.027122  0.241931
30364.0   0.202824 -0.174770 -0.421437  0.301579  1.024982 -0.068117

                 28     29
0
120121.0 -0.053506   5.30
113513.0 -0.376752   1.00
57232.0   0.008016  38.90
67608.0   0.142747   2.00
30364.0   0.009277  62.94

[5 rows x 29 columns]
```

In [88]:
```
X_train_smote_1.columns = ['V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10
```

In [89]:
```
del X_train_smote_1.index.name
```

In [90]:
```
X_train_smote_1.head()
```

Out[90]:
```
                V1        V2        V3        V4        V5        V6  \
120121.0  1.877828  0.421871 -0.631872  3.728578  0.536681  0.589198
113513.0  0.025692  0.423046 -1.231143 -1.893868  3.320716  3.262109
57232.0   1.163919  0.228664  0.139728  0.484538 -0.220504 -1.031357
67608.0  -0.295570 -0.484728  2.835961 -0.336874 -1.442787  0.998449
```

```
30364.0   1.109071  0.061282  0.622337  1.452091 -0.178357  0.351704

                V7        V8        V9       V10  ...       V20       V21  \
120121.0 -0.021479  0.126885 -0.914571  1.541791  ... -0.270228  0.123737
113513.0  0.671243  0.355486  0.528778  0.129648  ...  0.097613  0.218512
57232.0   0.401498 -0.227066 -0.539325  0.083866  ... -0.001633 -0.293428
67608.0  -1.271316  0.601446 -0.194068  0.231689  ...  0.162613  0.345339
30364.0  -0.285974  0.091020 -0.677818  0.662103  ...  0.124986  0.119724

               V22       V23       V24       V25       V26       V27  \
120121.0  0.470545  0.110416  0.760153  0.118625  0.041352 -0.034675
113513.0  0.980139 -0.307353  0.714692  0.128959 -0.537229 -0.291759
57232.0  -0.967329  0.125161  0.538190  0.233182  0.103396 -0.070322
67608.0   1.331306 -0.149416  0.288751 -0.486431  0.027122  0.241931
30364.0   0.202824 -0.174770 -0.421437  0.301579  1.024982 -0.068117

               V28  Amount
120121.0 -0.053506    5.30
113513.0 -0.376752    1.00
57232.0   0.008016   38.90
67608.0   0.142747    2.00
30364.0   0.009277   62.94

[5 rows x 29 columns]
```

In [91]:
```
X_train_smote_1.shape
```

Out[91]:
```
(454894, 29)
```

In [92]:
```
y_train_smote.shape
```

Out[92]:
```
(454894,)
```

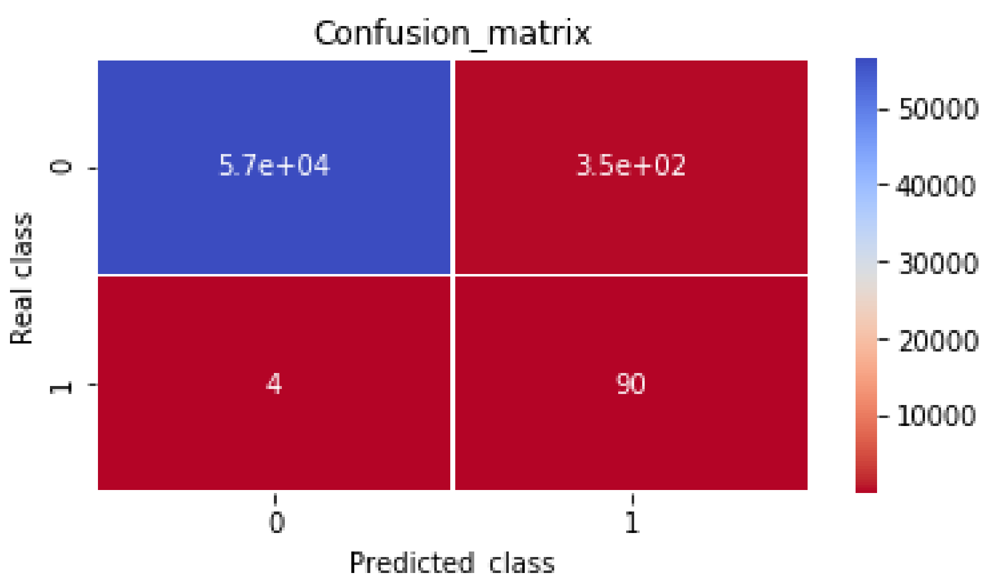
In [ ]:
```
# xg_params = {
#     'colsample_bytree': [0.2],
#     'gamma':[0.01],
#     'learning_rate':[0.001, 0.01],
#     'max_depth':np.arange(3,7,1),
#     'n_estimators':[4000, 5000],
#     'reg_alpha':[0.5,0.9],
#     'reg_lambda':[0.3, 0.4, 0.5],
#     'subsample': [0.2]
# }
```

In [93]:
```
xg_params = {
    'colsample_bytree': [0.2],
    'learning_rate':[0.001, 0.01],
    'gamma':[0.01],
    'max_depth':np.arange(3,4,1),
```

```
            'n_estimators':[100,200],
            'reg_alpha':[0.75],
            'reg_lambda':[0.4],
        }
```

```
In [94]: xgb = XGBClassifier()
         xgb_grid_smote = GridSearchCV(xgb, param_grid=xg_params, n_jobs=-1)
         xgb_model_smote = xgb_grid_smote.fit(X_train_smote_1, y_train_smote)

         xgb_pred_smote = xgb_model_smote.predict(X_test)

         model_scores(y_test, xgb_pred_smote)

         winsound.Beep(500,10000)
```

```
Accuracy Score: 0.993732663881184

Average Precision Score: 0.1945853752239907

Average Recall: 0.9574468085106383

Average F1 Score: 0.33519553072625696


C:\Users\Y\Anaconda3\lib\site-packages\sklearn\preprocessing\label.py:151: DeprecationWarning:
  if diff:
```



```
----------Classification Report------------------------------------
             precision    recall  f1-score   support

          0       1.00      0.99      1.00     56868
          1       0.20      0.96      0.34        94

avg / total       1.00      0.99      1.00     56962
```

Neural Network

Neural networks are modeled after biological neural networks and attempt to allow computers to learn in asimilar manner to humans. This is called reinforcement learning. Use cases for neural networks include pattern recognition, time series predictions, signal processing, image recognition, and anomaly detection.

The basic structure of a Artificial Neural Networks (ANN) is an input layer, hidden layers, and an output layer. Real values from the data go in the input layer. The hidden layer is the Layers in between input and output. Three or more hidden layers is a deep network. Final estimate of the output is in the output layer. The ReLu and tanh activation functions tend to have the best performance.

Cost functions, which will allow us to measure how well these neurons are performing. It measures how far off we are from the expected value. We can use our neurons and the measurement of error (our cost function) and then attempt to correct our prediction using Gradient Descent. Gradient descent is an optimization algorithm for finding the minimum of a function. To find a local minimum, we take steps proportional to the negative of the gradient. Using gradient descent we can figure out the best parameters for minimizing our cost, for example, finding the best values for the weights of the neuron inputs.

We use back propagation to quickly adjust the optimal parameters or weights across our entire network. Backpropagation works by calculating the error at the output and then distributes back through the network layers. It relies heavily on the chain rule to go back through the network and calculate these errors.

```
In [1]: # Import necessary modules
        import os
        os.environ['KERAS_BACKEND']='tensorflow'
        import keras
        from keras.layers import Dense
        from keras.models import Sequential
        import pandas as pd
        import numpy as np
        from sklearn.model_selection import train_test_split
        import matplotlib.pyplot as plt
        import seaborn as sns
        from sklearn.metrics import classification_report, average_precision_score, precision_r
        from sklearn.metrics import roc_curve, auc, f1_score, confusion_matrix
        %matplotlib inline
```

```
Using TensorFlow backend.
```

```
In [13]: def get_scores(y_tst, pred):
             print('Accuracy Score: {}\n'.format(accuracy_score(y_tst, pred)))
             print('Average Precision Score: {}\n'.format(average_precision_score(y_tst, pred)
             print('Average Recall Score: {}\n'.format(recall_score(y_tst, pred)))
             print('Average F1 Score: {}'.format(f1_score(y_tst, pred)))

         def model_scores(y_tst, pred):
             print('Accuracy Score: {}\n'.format(accuracy_score(y_tst, pred)))
             print('Average Precision Score: {}\n'.format(average_precision_score(y_tst, pred)
             print('Average Recall: {}\n'.format(recall_score(y_tst, pred)))
             print('Average F1 Score: {}'.format(f1_score(y_tst, pred)))

             cnf_matrix=confusion_matrix(y_tst, pred)

             fig= plt.figure(figsize=(6,3))

             sns.heatmap(cnf_matrix, cmap="coolwarm_r", annot=True, linewidths=0.5)
             plt.title("Confusion_matrix")
             plt.xlabel("Predicted_class")
             plt.ylabel("Real class")
             plt.show()
             print("\n----------Classification Report------------------------------------")
             print(classification_report(y_tst,pred))
```

```
In [14]: # pandas function to read in a csv file
         df = pd.read_csv('creditcard.csv')
```

```
In [15]: y = df['Class']
         X = df.iloc[:,:-1]
```

```
In [16]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=
```

```
In [17]: # Save the number of columns in predictors: n_cols
         n_cols = X.shape[1]
```

```
In [18]: n_cols
```

```
Out[18]: 30
```

```
In [19]: # Save the number of columns in predictors: n_cols
         input_shape = (30,)
```

```
In [20]: X_train.values.shape
```

```
Out[20]: (227845, 30)
```

```
In [21]: y_train.values.shape
```

Out[21]: (227845,)

In [22]: y_train.values

Out[22]: array([0, 0, 0, ..., 0, 0, 0], dtype=int64)

In [25]:
```
def get_new_model(input_shape = input_shape):
    model = Sequential()
    # Add the first layer
    model.add(Dense(activation="relu", input_shape=input_shape, units=100, kernel_ini

    # Add the second layer
    model.add(Dense(activation="relu", units=100, kernel_initializer='glorot_uniform')

    # Add the third layer
    model.add(Dense(activation="relu", units=100, kernel_initializer='glorot_uniform')

    # Add the output layer
    model.add(Dense(activation="sigmoid", units=1, kernel_initializer='glorot_uniform

    return(model)
```

2.15 Baseline Neural Network Model with 3 Learning Rates, Epoch=10, early_stopping(patience=2)

In [30]:
```
# Import the SGD optimizer
from keras.optimizers import SGD

# Import EarlyStopping
from keras.callbacks import EarlyStopping

# Create list of learning rates: lr_to_test
lr_to_test = [1, 0.01, .000001]
#lr_to_test = [1, 0.9]

# Loop over learning rates
for lr in lr_to_test:
    print('\n\nTesting model with learning rate: %f\n'%lr )

    # Build new model to test, unaffected by previous models
    model = get_new_model()

    # Create SGD optimizer with specified learning rate: my_optimizer
    my_optimizer = SGD(lr=lr)

    # Compile the model
    model.compile(optimizer=my_optimizer, loss='binary_crossentropy',  metrics=['accu

    # Define early_stopping_monitor
```

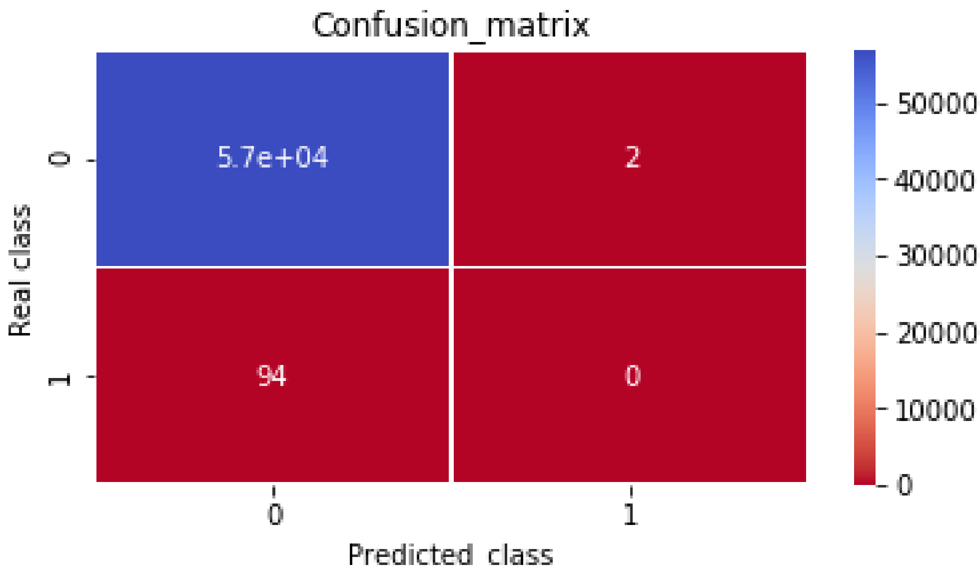
```
early_stopping_monitor = EarlyStopping(monitor='val_loss', min_delta=0, patience=
#'val_loss','acc','loss',val_acc'

keras.callbacks.History()

# Fit the model
#model.fit(X_train.values,y_train.values, epochs=1, batch_size = 10)

# Fit the model
model_1 = model.fit(X_train.values, y_train.values, batch_size = 10, epochs=10, va
```

```
Testing model with learning rate: 1.000000

Train on 182276 samples, validate on 45569 samples
Epoch 1/10
182276/182276 [==============================] - 25s 135us/step - loss: 0.2131 - acc: 0.9867 -
Epoch 2/10
182276/182276 [==============================] - 24s 134us/step - loss: 0.0282 - acc: 0.9982 -
Epoch 3/10
182276/182276 [==============================] - 26s 145us/step - loss: 0.0282 - acc: 0.9982 -
Epoch 00003: early stopping


Testing model with learning rate: 0.010000

Train on 182276 samples, validate on 45569 samples
Epoch 1/10
182276/182276 [==============================] - 26s 140us/step - loss: 0.0491 - acc: 0.9969 -
Epoch 2/10
182276/182276 [==============================] - 24s 134us/step - loss: 0.0282 - acc: 0.9982 -
Epoch 3/10
182276/182276 [==============================] - 24s 132us/step - loss: 0.0282 - acc: 0.9982 -
Epoch 4/10
182276/182276 [==============================] - 25s 136us/step - loss: 0.0282 - acc: 0.9982 -
Epoch 5/10
182276/182276 [==============================] - 24s 134us/step - loss: 0.0282 - acc: 0.9982 -
Epoch 6/10
182276/182276 [==============================] - 26s 144us/step - loss: 0.0282 - acc: 0.9982 -
Epoch 7/10
182276/182276 [==============================] - 27s 147us/step - loss: 0.0282 - acc: 0.9982 -
Epoch 8/10
182276/182276 [==============================] - 28s 151us/step - loss: 0.0282 - acc: 0.9982 -
Epoch 9/10
182276/182276 [==============================] - 28s 156us/step - loss: 0.0282 - acc: 0.9982 -
Epoch 10/10
182276/182276 [==============================] - 28s 151us/step - loss: 0.0282 - acc: 0.9982 -
```

```
Testing model with learning rate: 0.000001

Train on 182276 samples, validate on 45569 samples
Epoch 1/10
182276/182276 [==============================] - 29s 157us/step - loss: 0.1347 - acc: 0.9913 -
Epoch 2/10
182276/182276 [==============================] - 27s 150us/step - loss: 0.0322 - acc: 0.9978 -
Epoch 3/10
182276/182276 [==============================] - 27s 149us/step - loss: 0.0321 - acc: 0.9978 -
Epoch 4/10
182276/182276 [==============================] - 27s 151us/step - loss: 0.0319 - acc: 0.9978 -
Epoch 5/10
182276/182276 [==============================] - 27s 149us/step - loss: 0.0318 - acc: 0.9978 -
Epoch 6/10
182276/182276 [==============================] - 28s 153us/step - loss: 0.0316 - acc: 0.9978 -
Epoch 7/10
182276/182276 [==============================] - 28s 151us/step - loss: 0.0314 - acc: 0.9978 -
Epoch 8/10
182276/182276 [==============================] - 27s 149us/step - loss: 0.0311 - acc: 0.9978 -
Epoch 9/10
182276/182276 [==============================] - 27s 149us/step - loss: 0.0309 - acc: 0.9979 -
Epoch 10/10
182276/182276 [==============================] - 28s 155us/step - loss: 0.0307 - acc: 0.9979 -
```

```
In [32]: y_pred = model.predict(X_test.values)
         y_pred

Out[32]: array([[ 0.],
                [ 0.],
                [ 0.],
                ...,
                [ 0.],
                [ 0.],
                [ 0.]], dtype=float32)

In [33]: y_pred_int = y_pred.astype(int)

In [34]: cm = confusion_matrix(y_test, y_pred_int)
         cm

Out[34]: array([[56866,     2],
                [   94,     0]], dtype=int64)

In [35]: model_scores(y_test, y_pred_int)

Accuracy Score: 0.9983146659176293
```

```
Average Precision Score: 0.001650222955654647

Average Recall: 0.0

Average F1 Score: 0.0
```




```
----------Classification Report------------------------------------
             precision    recall  f1-score   support

          0       1.00      1.00      1.00     56868
          1       0.00      0.00      0.00        94

avg / total       1.00      1.00      1.00     56962
```

2.16 Result: Neural Network Predicted all X_test to be non-Fraud cases.

2.17 Experiment with Rebalancing

In [36]:
```
# Set up the model: model
model_w = Sequential()

# Add the first layer
model_w.add(Dense(activation="relu", input_dim=30, units=100, kernel_initializer='glo
```

```
    # Add the second layer
    model_w.add(Dense(activation="relu", units=100, kernel_initializer='glorot_uniform'))

    # Add the third layer
    model_w.add(Dense(activation="relu", units=100, kernel_initializer='glorot_uniform'))

    # Add the output layer
    model_w.add(Dense(activation="sigmoid", units=1, kernel_initializer='glorot_uniform'))
```

```
In [38]: # Import the SGD optimizer
         from keras.optimizers import SGD

         # Import EarlyStopping
         from keras.callbacks import EarlyStopping

         # Create list of learning rates: lr_to_test
         lr_to_test = [1, .01, .000001]

         # Loop over learning rates
         for lr in lr_to_test:
             print('\n\nTesting model with learning rate: %f\n'%lr )

             # Create SGD optimizer with specified learning rate: my_optimizer
             my_optimizer = SGD(lr=lr)

             # Compile the model
             model_w.compile(optimizer=my_optimizer, loss='binary_crossentropy', metrics=['accu

             # Define early_stopping_monitor
             #early_stopping_monitor = EarlyStopping(monitor='val_loss', min_delta=0, patience

             early_stopping_monitor = EarlyStopping(patience=2, verbose=2)

             #Define class weight
             class_weight = {0 : .1, 1: .9}

             # Fit the model
             model_training_w = model_w.fit(X_train.values, y_train.values, batch_size = 10,val
                                             class_weight = class_weight)
```

```
Testing model with learning rate: 1.000000

Train on 182276 samples, validate on 45569 samples
Epoch 1/10
182276/182276 [==============================] - 25s 134us/step - loss: 0.0268 - acc: 0.9974 -
Epoch 2/10
182276/182276 [==============================] - 25s 135us/step - loss: 0.0254 - acc: 0.9982 -
```

```
Epoch 3/10
182276/182276 [==============================] - 25s 139us/step - loss: 0.0254 - acc: 0.9982 -


Testing model with learning rate: 0.010000

Train on 182276 samples, validate on 45569 samples
Epoch 1/10
182276/182276 [==============================] - 25s 139us/step - loss: 0.0254 - acc: 0.9982 -
Epoch 2/10
182276/182276 [==============================] - 24s 134us/step - loss: 0.0254 - acc: 0.9982 -
Epoch 3/10
182276/182276 [==============================] - 25s 135us/step - loss: 0.0254 - acc: 0.9982 -


Testing model with learning rate: 0.000001

Train on 182276 samples, validate on 45569 samples
Epoch 1/10
182276/182276 [==============================] - 25s 140us/step - loss: 0.0254 - acc: 0.9982 -
Epoch 2/10
182276/182276 [==============================] - 24s 134us/step - loss: 0.0254 - acc: 0.9982 -
Epoch 3/10
182276/182276 [==============================] - 26s 145us/step - loss: 0.0254 - acc: 0.9982 -
```

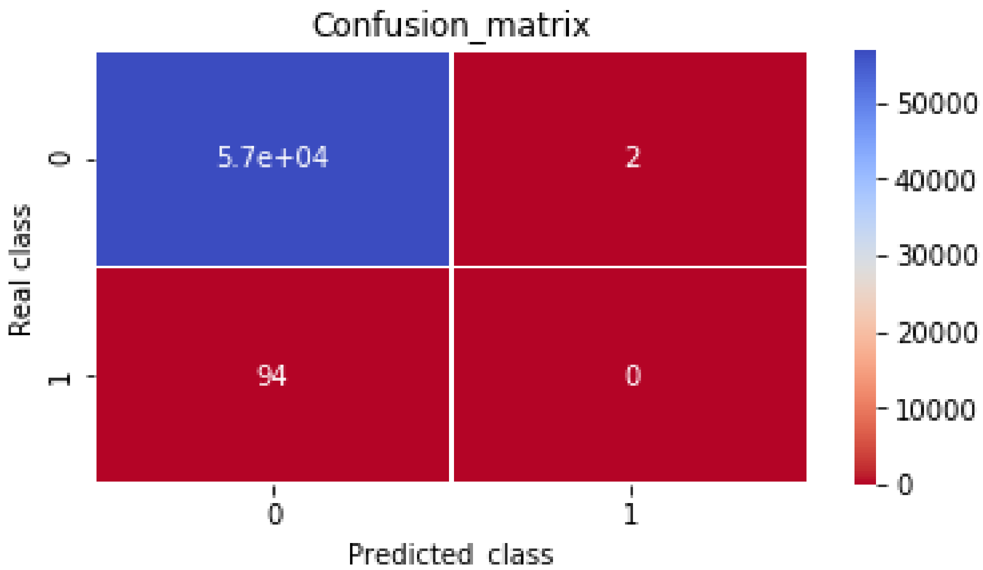
```
In [39]: y_pred_w = model_w.predict(X_test.values)
         y_pred_w
```

```
Out[39]: array([[ 0.],
                [ 0.],
                [ 0.],
                ...,
                [ 0.],
                [ 0.],
                [ 0.]], dtype=float32)
```

```
In [40]: y_pred_W = y_pred_w.astype(int)
```

```
In [41]: import seaborn as sns
         model_scores(y_test, y_pred_int)
```

```
Accuracy Score: 0.9983146659176293

Average Precision Score: 0.001650222955654647

Average Recall: 0.0

Average F1 Score: 0.0
```




```
----------Classification Report------------------------------------
             precision    recall  f1-score   support

          0       1.00      1.00      1.00     56868
          1       0.00      0.00      0.00        94

avg / total       1.00      1.00      1.00     56962
```

2.18 Result: No difference before and after re-balancing.

2.19 Experimenting with different optimizer: 'adam'

In [42]:
```
# Set up the model: model
model_adam = Sequential()

# Add the first layer
model_adam.add(Dense(activation="relu", input_dim=30, units=100, kernel_initializer='g

# Add the second layer
model_adam.add(Dense(activation="relu", units=100, kernel_initializer='glorot_uniform

# Add the third layer
model_adam.add(Dense(activation="relu", units=100, kernel_initializer='glorot_uniform

# Add the output layer
model_adam.add(Dense(activation="sigmoid", units=1, kernel_initializer='glorot_unifor
```

```
In [43]: # Import EarlyStopping
         from keras.callbacks import EarlyStopping

         # Compile the model
         model_adam.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy']

         # Define early_stopping_monitor
         early_stopping_monitor = EarlyStopping(patience=2, verbose=2)

         # Fit the model
         model_adam_training = model_adam.fit(X_train.values, y_train.values, validation_split
```

```
Train on 159491 samples, validate on 68354 samples
Epoch 1/10
159491/159491 [==============================] - 11s 70us/step - loss: 0.0272 - acc: 0.9983 - v
Epoch 2/10
159491/159491 [==============================] - 10s 64us/step - loss: 0.0272 - acc: 0.9983 - v
Epoch 3/10
159491/159491 [==============================] - 11s 68us/step - loss: 0.0272 - acc: 0.9983 - v
```

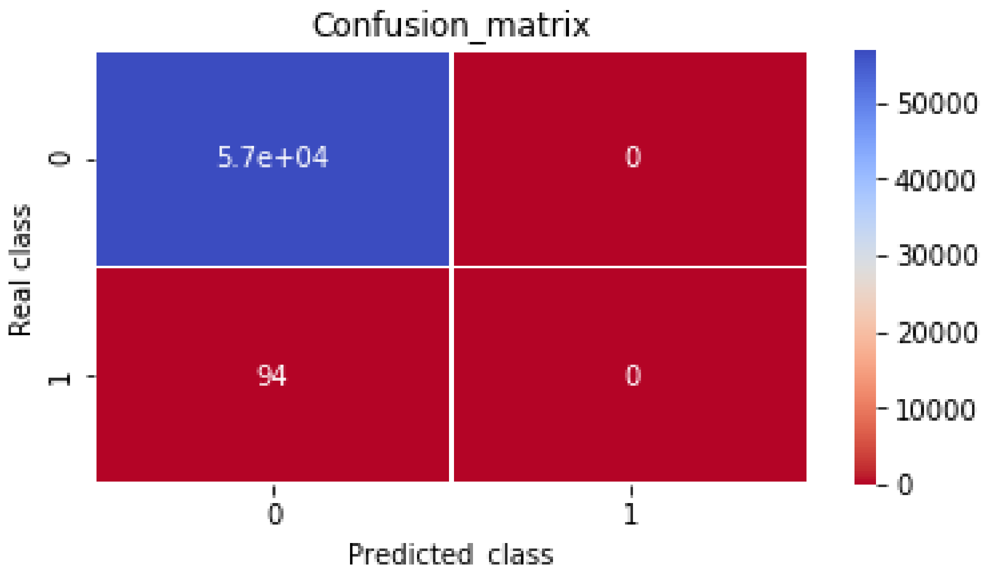
```
In [44]: y_pred_adam = model_adam.predict(X_test.values)
         y_pred_adam
```

```
Out[44]: array([[ 0.],
                [ 0.],
                [ 0.],
                ...,
                [ 0.],
                [ 0.],
                [ 0.]], dtype=float32)
```

```
In [45]: y_pred_adam = y_pred_adam.astype(int)
```

```
In [46]: import seaborn as sns
         model_scores(y_test, y_pred_adam)
```

```
Accuracy Score: 0.9983497770443454

Average Precision Score: 0.001650222955654647

Average Recall: 0.0

Average F1 Score: 0.0


C:\Users\Y\.conda\envs\tensorflow\lib\site-packages\sklearn\metrics\classification.py:1135: Und
  'precision', 'predicted', average, warn_for)
```




```
----------Classification Report------------------------------------
             precision    recall  f1-score   support

          0       1.00      1.00      1.00     56868
          1       0.00      0.00      0.00        94

avg / total       1.00      1.00      1.00     56962


C:\Users\Y\.conda\envs\tensorflow\lib\site-packages\sklearn\metrics\classification.py:1135: Und
  'precision', 'predicted', average, warn_for)
```

2.20 Result: No difference before and after using Adam optimizer.

2.21 Experimenting with different optimizer: 'rmsprop' (rec for recurrent NN)

In [47]:
```
# Set up the model: model
model_rms = Sequential()

# Add the first layer
model_rms.add(Dense(activation="relu", input_dim=30, units=100, kernel_initializer='gl

# Add the second layer
model_rms.add(Dense(activation="relu", units=100, kernel_initializer='glorot_uniform')
```

```
# Add the third layer
model_rms.add(Dense(activation="relu", units=100, kernel_initializer='glorot_uniform')

# Add the output layer
model_rms.add(Dense(activation="sigmoid", units=1, kernel_initializer='glorot_uniform
```

In [48]:
```
model_rms
```

Out[48]:
```
<keras.models.Sequential at 0x21d2aad4908>
```

In [49]:
```
# Import EarlyStopping
from keras.callbacks import EarlyStopping

# Compile the model
model_rms.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy

# Define early_stopping_monitor
early_stopping_monitor = EarlyStopping(patience=2, verbose=2)

# Fit the model
model_rms_training = model_rms.fit(X_train.values, y_train.values, validation_split=.3
```

```
Train on 159491 samples, validate on 68354 samples
Epoch 1/10
159491/159491 [==============================] - 10s 66us/step - loss: 0.0272 - acc: 0.9983 - v
Epoch 2/10
159491/159491 [==============================] - 10s 60us/step - loss: 0.0272 - acc: 0.9983 - v
Epoch 3/10
159491/159491 [==============================] - 10s 64us/step - loss: 0.0272 - acc: 0.9983 - v
Epoch 00003: early stopping
```

In [50]:
```
y_pred_rms = model_rms.predict(X_test.values)
y_pred_rms
```

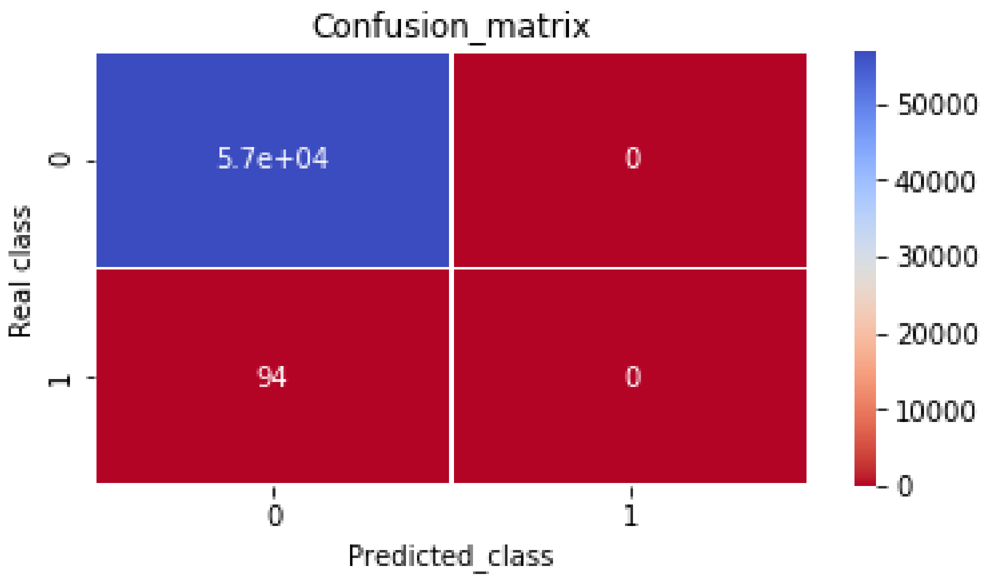
Out[50]:
```
array([[ 0.],
       [ 0.],
       [ 0.],
       ...,
       [ 0.],
       [ 0.],
       [ 0.]], dtype=float32)
```

In [51]:
```
y_pred_rms = y_pred_rms.astype(int)
```

In [52]:
```
import seaborn as sns
model_scores(y_test, y_pred_rms)
```

```
Accuracy Score: 0.9983497770443454
```

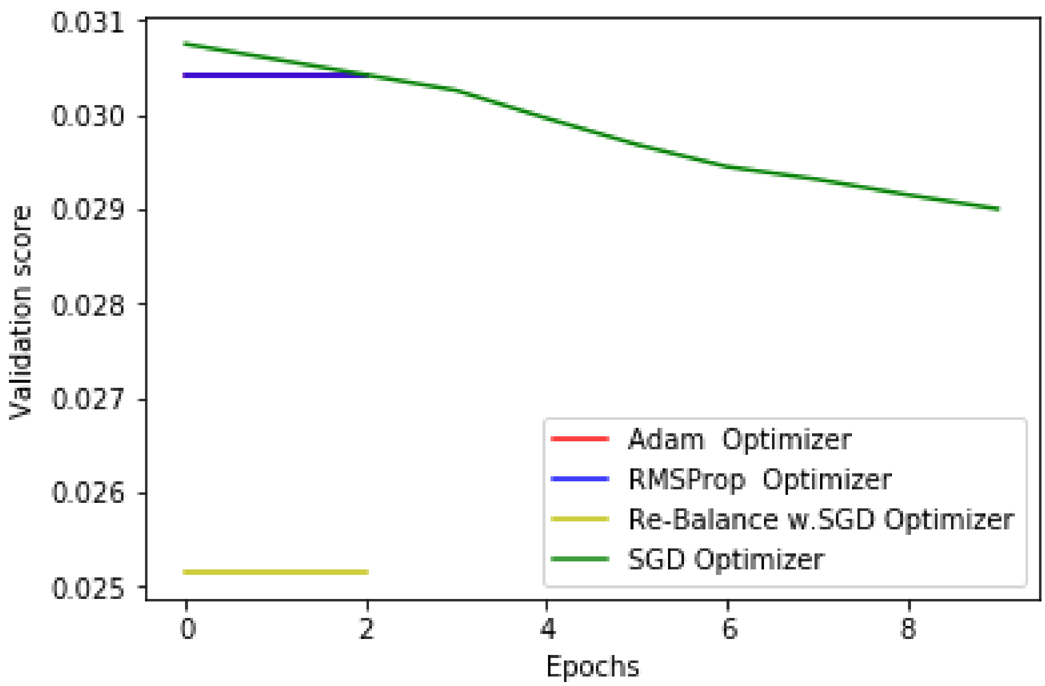
```
Average Precision Score: 0.001650222955654647

Average Recall: 0.0

Average F1 Score: 0.0

C:\Users\Y\.conda\envs\tensorflow\lib\site-packages\sklearn\metrics\classification.py:1135: Und
  'precision', 'predicted', average, warn_for)
```




```
----------Classification Report------------------------------------
             precision    recall  f1-score   support

          0       1.00      1.00      1.00     56868
          1       0.00      0.00      0.00        94

avg / total       1.00      1.00      1.00     56962

C:\Users\Y\.conda\envs\tensorflow\lib\site-packages\sklearn\metrics\classification.py:1135: Und
  'precision', 'predicted', average, warn_for)
```

In [57]:
```
# Create the plot
plt.plot(model_adam_training.history['val_loss'], 'r', label='Adam  Optimizer')
```

```
plt.plot(model_rms_training.history['val_loss'], 'b', label='RMSProp  Optimizer')
plt.plot(model_training_w.history['val_loss'], 'y', label='Re-Balance w.SGD Optimizer
plt.plot(model_1.history['val_loss'], 'g', label='SGD Optimizer')
plt.xlabel('Epochs')
plt.ylabel('Validation score')
plt.legend()
plt.show()
```



Results

The six different algorithms that were used in this study were logistic regression, random forest classifer, support vector classifier, XGBoost, and Neural Networks. The oversampling techniques were random over sampling and synthetic minority oversampling techinque (SMOTE). The undersampling techniques were random under sampling, edited nearest neighbor, and condensed nearest neighbors. I focused on the f1 score in my analysis. It is also a valid measure of an accurate model. It is the harmonic mean of precision and recall, and will be more insensitive to imbalanced data.

Of the six different algorithms that were used to predict this imbalanced data seet, the best algorithm was random forest classifier, without under or oversampling, with an average F1 score of .89. Second place went to XGBoost without under or oversampling with an average F1 score of .84. There was a tie for third place. Logistic regression utilizing an L2 regularization penalty, Lasso regression, with an average F1 score of .75, and logistic regression with an L2 regularization penalty and re-balanced sampling weights with an average F1 score of .76. Fourth place went to Logistic regression utilizing L2 regularization penalty and under sampling utilizing edited nearest neighbors with an average F1 score of .73

1st place: Random Forest F1 score .89

2nd place: XGBoost F1 Score .84
3rd place: Logistic Regression w.L2 F1 Score=.75, w.re-balance & L2 F1 Score=.76
4th place: Logistic Regression w.ENN & L2 F1 Score .73