Term Project_draft_2_v1

November 20, 2021

[1]:
```
# Course       : DSC630 - Predictive Analytics
# Project      : Credit Card Fraud Detection
# Name         : Vasanthakumar Kalaikkovan, Ganeshkumar Muthusamy and Venkata␣
 ↪Kanaparthi
```

0.1 Problem Statement

Credit card fraud is a major problem in financial services and costs billions of dollars every year. Credit card fraud continues to increase due to the rise and acceleration of Phone Order / Mail Order / E-Commerce. There has been tremendous use of credit cards for online shopping which led to a high amount of fraud related to credit cards. Financial institutions like Visa, MasterCard, Amex, and all debit networks have mandated that banks and merchants introduce EMV card technology to counter the fraud. In 2018, a total of $24.26 Billion was lost due to payment card fraud across the globe, and the USA is the most fraud-prone country. Credit card fraud was ranked the number one type of identity theft fraud. Credit card fraud increased by 18.4% in 2018 and is still climbing. There can be two kinds of card fraud, card-present fraud, and card-not-present fraud.

[2]:
```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import scikitplot as skplt
from sklearn.model_selection import train_test_split,GridSearchCV
from imblearn.over_sampling import SMOTE
from sklearn.feature_selection import VarianceThreshold
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif
from sklearn.dummy import DummyClassifier
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.metrics import␣
 ↪classification_report,confusion_matrix,ConfusionMatrixDisplay,roc_auc_score
from sklearn.metrics import auc,make_scorer,precision_recall_curve,log_loss
from sklearn.model_selection import cross_val_score
from numpy import mean, std
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
```

```python
from sklearn.linear_model import SGDClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
import scikitplot as skplt
```

[3]:
```python
# Load data into a dataframe
df = pd.read_csv("creditcard.csv")
df.head(10)
```

[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
5   2.0 -0.425966  0.960523  1.141109 -0.168252  0.420987 -0.029728  0.476201
6   4.0  1.229658  0.141004  0.045371  1.202613  0.191881  0.272708 -0.005159
7   7.0 -0.644269  1.417964  1.074380 -0.492199  0.948934  0.428118  1.120631
8   7.0 -0.894286  0.286157 -0.113192 -0.271526  2.669599  3.721818  0.370145
9   9.0 -0.338262  1.119593  1.044367 -0.222187  0.499361 -0.246761  0.651583

         V8        V9  …       V21       V22       V23       V24       V25  \
0  0.098698  0.363787  … -0.018307  0.277838 -0.110474  0.066928  0.128539
1  0.085102 -0.255425  … -0.225775 -0.638672  0.101288 -0.339846  0.167170
2  0.247676 -1.514654  …  0.247998  0.771679  0.909412 -0.689281 -0.327642
3  0.377436 -1.387024  … -0.108300  0.005274 -0.190321 -1.175575  0.647376
4 -0.270533  0.817739  … -0.009431  0.798278 -0.137458  0.141267 -0.206010
5  0.260314 -0.568671  … -0.208254 -0.559825 -0.026398 -0.371427 -0.232794
6  0.081213  0.464960  … -0.167716 -0.270710 -0.154104 -0.780055  0.750137
7 -3.807864  0.615375  …  1.943465 -1.015455  0.057504 -0.649709 -0.415267
8  0.851084 -0.392048  … -0.073425 -0.268092 -0.204233  1.011592  0.373205
9  0.069539 -0.736727  … -0.246914 -0.633753 -0.120794 -0.385050 -0.069733

        V26       V27       V28  Amount  Class
0 -0.189115  0.133558 -0.021053  149.62      0
1  0.125895 -0.008983  0.014724    2.69      0
2 -0.139097 -0.055353 -0.059752  378.66      0
3 -0.221929  0.062723  0.061458  123.50      0
4  0.502292  0.219422  0.215153   69.99      0
5  0.105915  0.253844  0.081080    3.67      0
6 -0.257237  0.034507  0.005168    4.99      0
7 -0.051634 -1.206921 -1.085339   40.80      0
8 -0.384157  0.011747  0.142404   93.20      0
9  0.094199  0.246219  0.083076    3.68      0

[10 rows x 31 columns]
```

[4]:
```python
# Check the dimension of the table
print("The dimension of the table is: ", df.shape)
```

```
The dimension of the table is:  (284807, 31)
```

[5]:
```python
# What type of variables are in the table
print("Describe Data")
print(df.describe())
```

```
Describe Data
                Time            V1            V2            V3            V4  \
count  284807.000000  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean    94813.859575  3.919560e-15  5.688174e-16 -8.769071e-15  2.782312e-15
std     47488.145955  1.958696e+00  1.651309e+00  1.516255e+00  1.415869e+00
min         0.000000 -5.640751e+01 -7.271573e+01 -4.832559e+01 -5.683171e+00
25%     54201.500000 -9.203734e-01 -5.985499e-01 -8.903648e-01 -8.486401e-01
50%     84692.000000  1.810880e-02  6.548556e-02  1.798463e-01 -1.984653e-02
75%    139320.500000  1.315642e+00  8.037239e-01  1.027196e+00  7.433413e-01
max    172792.000000  2.454930e+00  2.205773e+01  9.382558e+00  1.687534e+01

                 V5            V6            V7            V8            V9  \
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean  -1.552563e-15  2.010663e-15 -1.694249e-15 -1.927028e-16 -3.137024e-15
std    1.380247e+00  1.332271e+00  1.237094e+00  1.194353e+00  1.098632e+00
min   -1.137433e+02 -2.616051e+01 -4.355724e+01 -7.321672e+01 -1.343407e+01
25%   -6.915971e-01 -7.682956e-01 -5.540759e-01 -2.086297e-01 -6.430976e-01
50%   -5.433583e-02 -2.741871e-01  4.010308e-02  2.235804e-02 -5.142873e-02
75%    6.119264e-01  3.985649e-01  5.704361e-01  3.273459e-01  5.971390e-01
max    3.480167e+01  7.330163e+01  1.205895e+02  2.000721e+01  1.559499e+01

       …           V21           V22           V23           V24  \
count  …  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean   …  1.537294e-16  7.959909e-16  5.367590e-16  4.458112e-15
std    …  7.345240e-01  7.257016e-01  6.244603e-01  6.056471e-01
min    … -3.483038e+01 -1.093314e+01 -4.480774e+01 -2.836627e+00
25%    … -2.283949e-01 -5.423504e-01 -1.618463e-01 -3.545861e-01
50%    … -2.945017e-02  6.781943e-03 -1.119293e-02  4.097606e-02
75%    …  1.863772e-01  5.285536e-01  1.476421e-01  4.395266e-01
max    …  2.720284e+01  1.050309e+01  2.252841e+01  4.584549e+00

                V25           V26           V27           V28         Amount  \
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05  284807.000000
mean   1.453003e-15  1.699104e-15 -3.660161e-16 -1.206049e-16      88.349619
std    5.212781e-01  4.822270e-01  4.036325e-01  3.300833e-01     250.120109
min   -1.029540e+01 -2.604551e+00 -2.256568e+01 -1.543008e+01       0.000000
25%   -3.171451e-01 -3.269839e-01 -7.083953e-02 -5.295979e-02       5.600000
50%    1.659350e-02 -5.213911e-02  1.342146e-03  1.124383e-02      22.000000
75%    3.507156e-01  2.409522e-01  9.104512e-02  7.827995e-02      77.165000
```

```
max    7.519589e+00  3.517346e+00  3.161220e+01  3.384781e+01   25691.160000

               Class
count  284807.000000
mean        0.001727
std         0.041527
min         0.000000
25%         0.000000
50%         0.000000
75%         0.000000
max         1.000000

[8 rows x 31 columns]
```

[6]:
```python
# Check if any missing values
df.isnull().sum()
```

[6]:
```
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
```

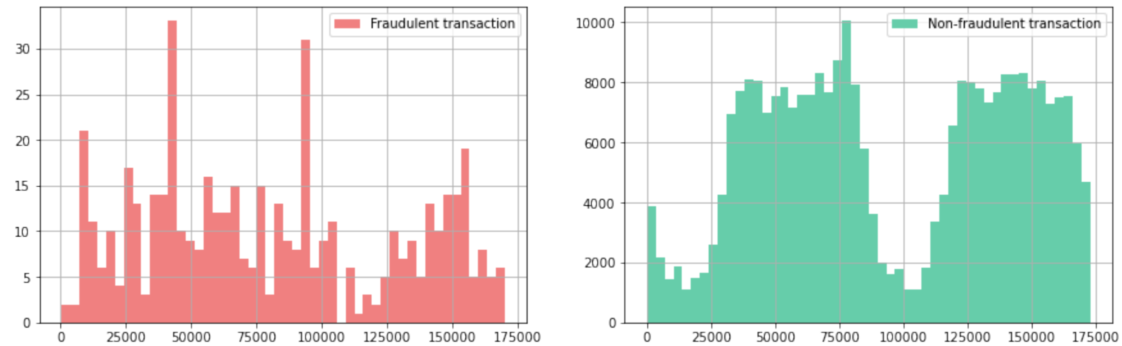
[7]:
```
# Check the types of each feature
df.dtypes
```

[7]:
```
Time      float64
V1        float64
V2        float64
V3        float64
V4        float64
V5        float64
V6        float64
V7        float64
V8        float64
V9        float64
V10       float64
V11       float64
V12       float64
V13       float64
V14       float64
V15       float64
V16       float64
V17       float64
V18       float64
V19       float64
V20       float64
V21       float64
V22       float64
V23       float64
V24       float64
V25       float64
V26       float64
V27       float64
V28       float64
Amount    float64
Class       int64
dtype: object
```

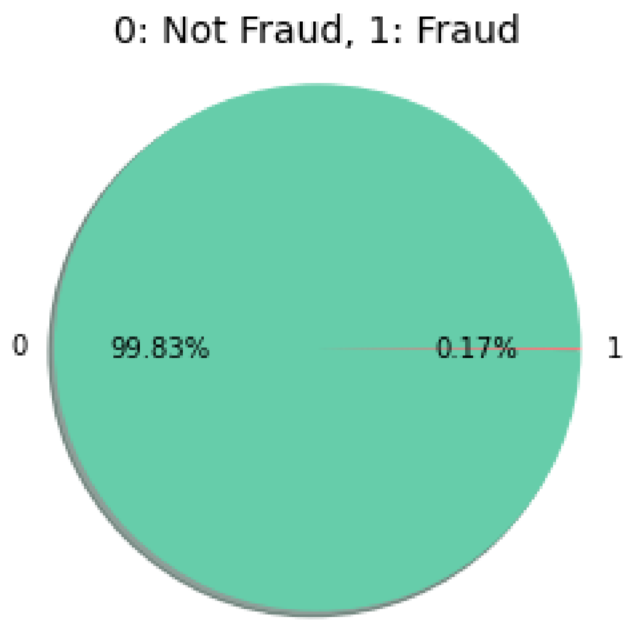
[8]:
```
# Histograms fraudulent and non-fraudulent transactions
fraud = df[df.Class == 1]
non_fraud = df[df.Class == 0]
plt.figure(figsize=(15, 10))

plt.subplot(2, 2, 1)
fraud.Time.hist(color='#F08080', bins=50, label="Fraudulent transaction")
plt.legend()
```

```
plt.subplot(2, 2, 2)
non_fraud.Time.hist(color='#66CDAA', bins=50, label="Non-fraudulent␣
↪transaction")
plt.legend()
```

[8]: <matplotlib.legend.Legend at 0x7f8929110b20>



[9]:
```
# Pie Chart fradulent % vs non-fradulent %
my_data = "Class"
grouped = df[my_data].value_counts().reset_index()
grouped = grouped.rename(columns = {my_data : "count", "index" : my_data})

labels = grouped[my_data]
sizes = grouped['count']

fig, ax = plt.subplots()
ax.pie(sizes, labels=labels, autopct='%1.2f%%', shadow=True, startangle=0,␣
↪colors = ['#66CDAA','#F08080'])
ax.axis('equal')
plt.title('0: Not Fraud, 1: Fraud', fontsize=14)
plt.show()
```

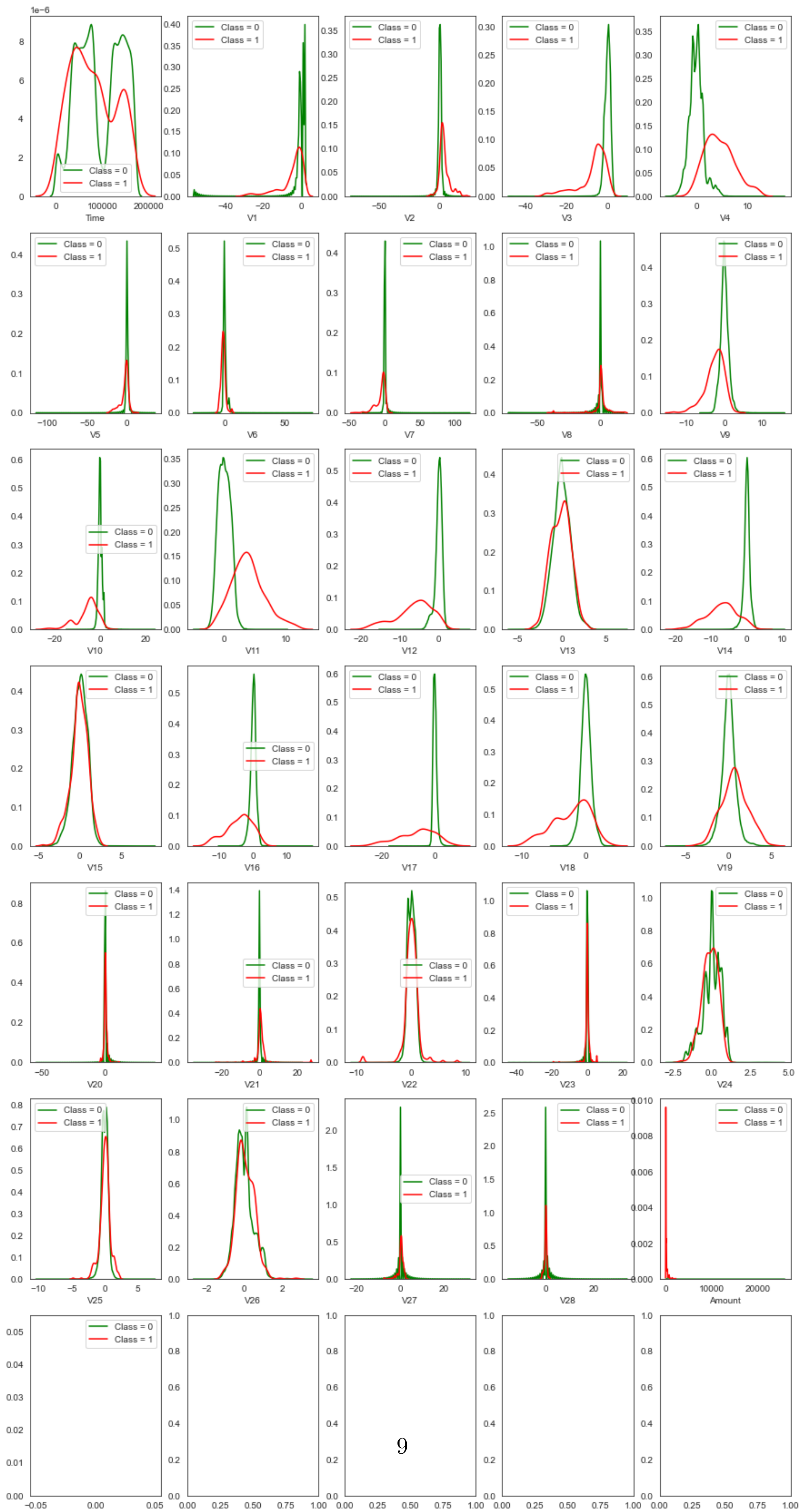


[10]:
```
# Density plot of the features
var = df.columns.values

x = 0
non_fraud = df.loc[df['Class'] == 0]
fraud = df.loc[df['Class'] == 1]

sns.set_style('white')
plt.figure()
fig, ax = plt.subplots(7,5,figsize=(15,30))

for feature in var:
    x += 1
    plt.subplot(7,5,x)
    sns.kdeplot(non_fraud[feature],label="Class = 0", color='green')
    sns.kdeplot(fraud[feature],label="Class = 1", color='red')
    plt.xlabel(feature, fontsize=10)
    locs, labels = plt.xticks()
    plt.tick_params(axis='both', which='major', labelsize=10)
plt.show();
```

```
/Users/ganeshkumar/opt/anaconda3/lib/python3.8/site-
packages/seaborn/distributions.py:283: UserWarning: Data must have variance to
compute a kernel density estimate.
  warnings.warn(msg, UserWarning)
/Users/ganeshkumar/opt/anaconda3/lib/python3.8/site-
```

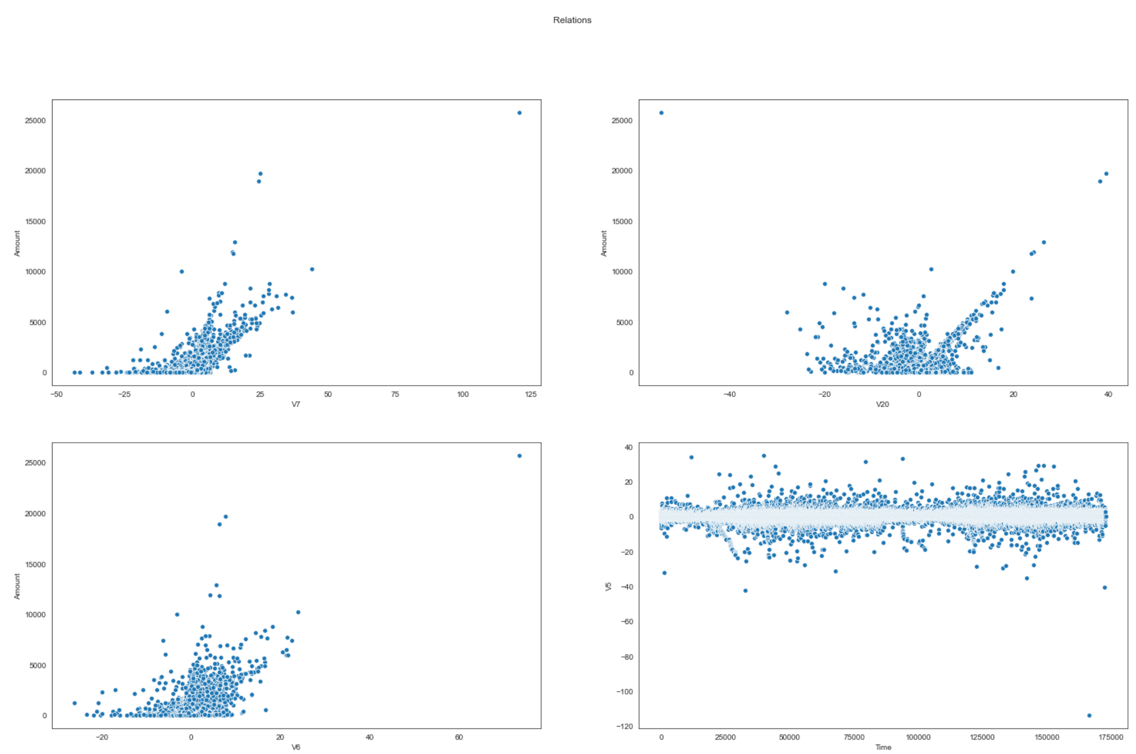
```
packages/seaborn/distributions.py:283: UserWarning: Data must have variance to
compute a kernel density estimate.
  warnings.warn(msg, UserWarning)
```

```
<Figure size 432x288 with 0 Axes>
```

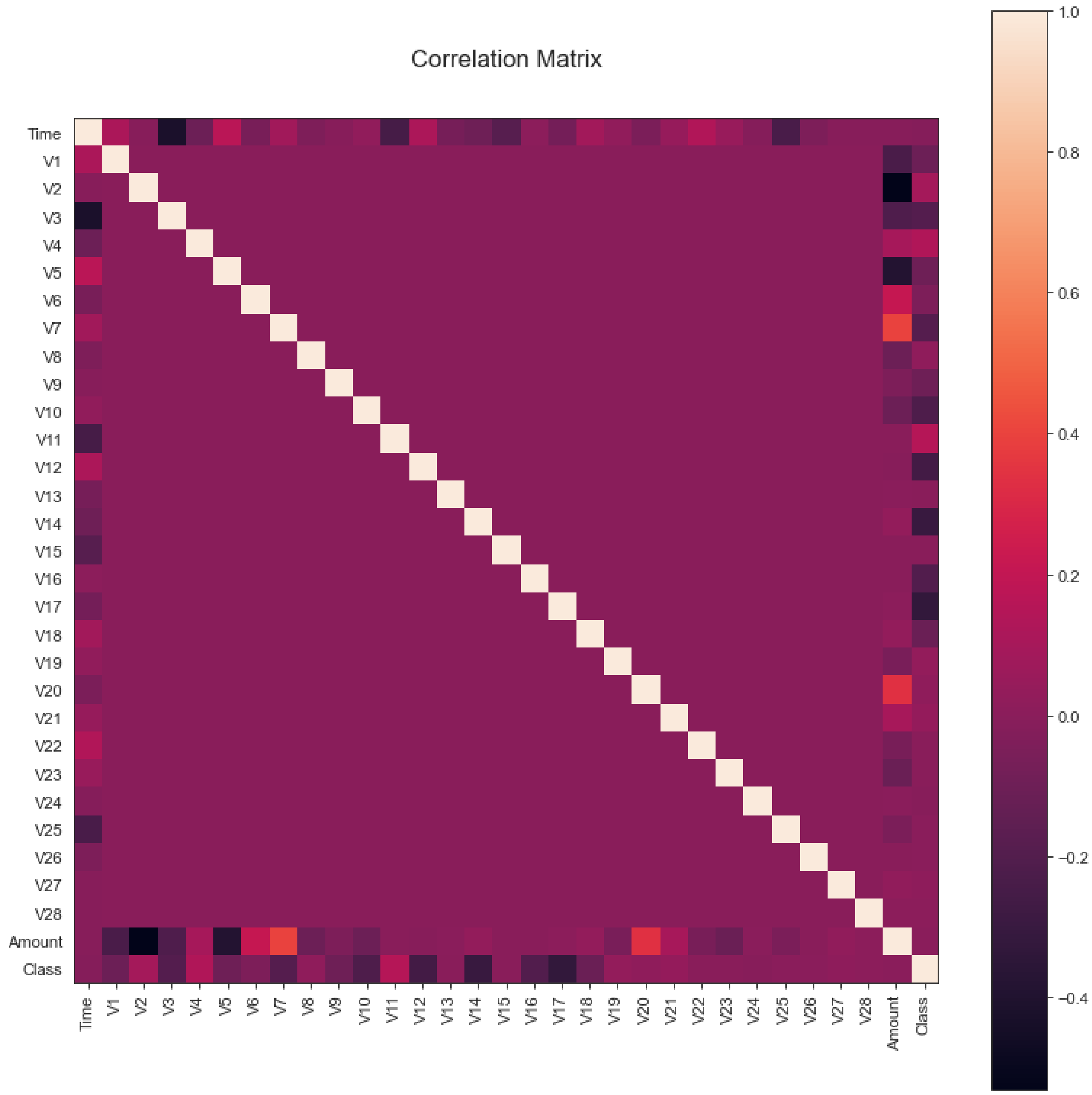

Class = 0
Class = 1
Time
V1
V2
V3
V4
V5
V6
V7
V8
V9
V10
V11
V12
V13
V14
V15
V16
V17
V18
V19
V20
V21
V22
V23
V24
V25
V26
V27
V28
Amount
Class

[11]:
```
def topn(df,n):
    npa = df.values

    npa = np.tril(npa, -1)
    topn_ind = np.argpartition(npa,-n,None)[-n:] #flatend ind, unsorted
    topn_ind = topn_ind[np.argsort(npa.flat[topn_ind])][::-1] #arg sort in␣
 ↪descending order
    cols,indx = np.unravel_index(topn_ind,npa.shape,'F') #unflatten, using␣
 ↪column-major ordering

    return ([df.columns[c] for c in cols],[df.index[i] for i in indx])
```

[12]:
```
max_corr_x , max_corr_y = topn(df.corr(),4)

fig, axes = plt.subplots(2, 2, figsize=(25, 15))
fig.suptitle('Relations')

axes = axes.reshape(4,)

for i in range(len(max_corr_x)):
    sns.scatterplot(ax = axes[i],data = df, x= max_corr_x[i],y=max_corr_y[i])
```



[13]:
```
#Correlation Matrix
df = df[[col for col in df if df[col].nunique() > 1]] # keep columns where␣
↪there are more than 1 unique values
corr = df.corr()
plt.figure(num=None, figsize=(12, 12), dpi=80, facecolor='w', edgecolor='k')
corrMat = plt.matshow(corr, fignum = 1)
plt.xticks(range(len(corr.columns)), corr.columns, rotation=90)
plt.yticks(range(len(corr.columns)), corr.columns)
plt.gca().xaxis.tick_bottom()
plt.colorbar(corrMat)
plt.title(f'Correlation Matrix', fontsize=15)
plt.show()
```



[14]:
```
df['Class'].value_counts()
```

[14]:
```
0    284315
1       492
Name: Class, dtype: int64
```

0.1.1 Train and Test

[15]:
```
# Train and test data
x=df.drop(columns=["Time","Class"],axis="columns")
y=df.Class
```

[16]:
```
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=.
↪25,random_state=42)
```

[17]:
```
# Details of training dataset
print("Shape of x_train dataset: ", x_train.shape)
print("Shape of y_train dataset: ", y_train.shape)
print("Shape of x_test dataset: ", x_test.shape)
print("Shape of y_test dataset: ", y_test.shape)
print("Before OverSampling, counts of label '1': {}".format(sum(y_train==1)))
print("Before OverSampling, counts of label '0': {} \n".format(sum(y_train==0)))
```

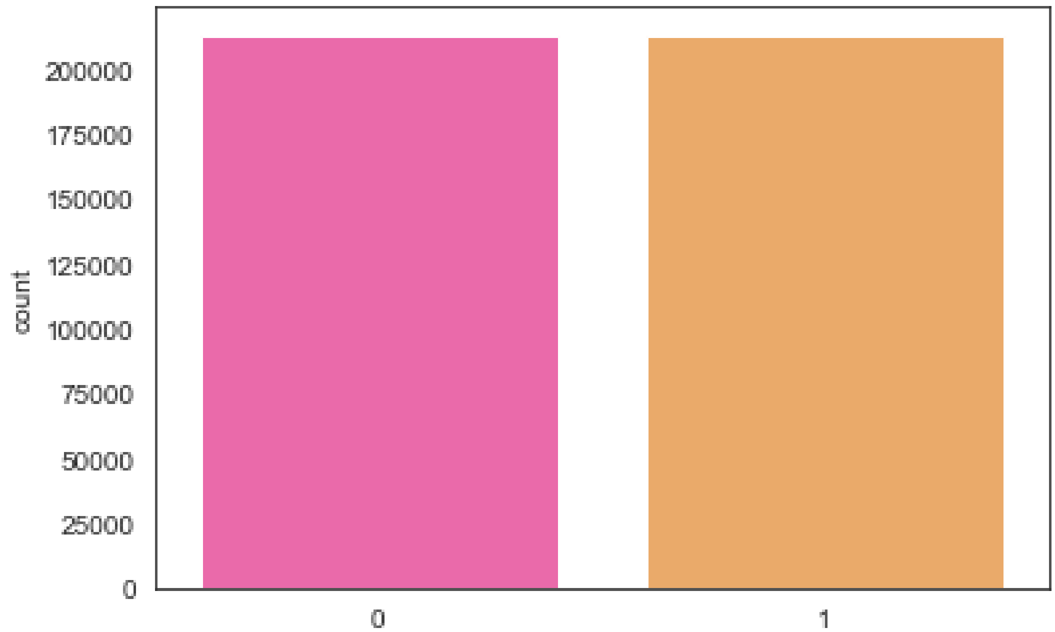
```
Shape of x_train dataset:  (213605, 29)
Shape of y_train dataset:  (213605,)
Shape of x_test dataset:  (71202, 29)
Shape of y_test dataset:  (71202,)
Before OverSampling, counts of label '1': 379
Before OverSampling, counts of label '0': 213226
```

[18]:
```
# Oversample the training dataset
sm = SMOTE(random_state=2)
x_train_s, y_train_s = sm.fit_resample(x_train, y_train.ravel())

print('After OverSampling, the shape of train_x: {}'.format(x_train_s.shape))
print('After OverSampling, the shape of train_y: {} \n'.format(y_train_s.shape))

print("After OverSampling, counts of label '1', %: {}".format(sum(y_train_s==1)/
↪len(y_train_s)*100.0,2))
print("After OverSampling, counts of label '0', %: {}".format(sum(y_train_s==0)/
↪len(y_train_s)*100.0,2))

sns.countplot(x=y_train_s, data=df, palette='spring')
```

```
After OverSampling, the shape of train_x: (426452, 29)
After OverSampling, the shape of train_y: (426452,)
```

```
After OverSampling, counts of label '1', %: 50.0
After OverSampling, counts of label '0', %: 50.0
```

[18]: <matplotlib.axes._subplots.AxesSubplot at 0x7f8919687ca0>



[19]:
```
# Feature selection using Variance Threshold with threshold of 0.5
var = VarianceThreshold(threshold=.5)
var.fit(x_train_s,y_train_s)
x_train_var=var.transform(x_train_s)
x_test_var=var.transform(x_test)
x_train_var.shape
```

[19]: (426452, 25)

[20]:
```
# Alternate way to perform feature selection and display the features
def variance_threshold_selector(data, threshold=0.5):
    selector = VarianceThreshold(threshold)
    selector.fit(data)
    return data[data.columns[selector.get_support(indices=True)]]
variance_threshold_selector(x_train_s, 0.5)
```

[20]:
```
          V1        V2        V3        V4        V5        V6        V7  \
0  -1.648591  1.228130  1.370169 -1.735542 -0.029455 -0.484129  0.918645
1  -0.234775 -0.493269  1.236728 -2.338793 -1.176733  0.885733 -1.960981
2   1.134626 -0.774460 -0.163390 -0.533358 -0.604555 -0.244482 -0.212682
```

```
3        0.069514  1.017753  1.033117  1.384376  0.223233 -0.310845  0.597287
4       -0.199441  0.610092 -0.114437  0.256565  2.290752  4.008475 -0.123530
…             …         …         …         …         …         …         …
426447   1.065197  0.639987  0.203695  3.005300 -0.066426  0.058743 -0.365901
426448  -6.689924  2.108646 -7.070325  5.133202 -2.313166 -2.003876 -8.549248
426449  -0.415163 -0.973361 -2.162616  2.541888  0.707195 -0.199787  1.793974
426450   0.172346  0.879474  1.678842  3.136607 -0.697444  1.213229 -1.449331
426451  -2.160564  1.424873 -0.282192  2.005992 -0.628338  0.148190 -1.810695

              V8        V9       V10  …       V16        V17       V18  \
0      -0.438750  0.982144  1.241635  …  0.664548  -1.280961  0.184568
1      -2.363412 -2.694774  0.360215  … -0.163459   0.562423 -0.577032
2       0.040782 -1.136627  0.792009  … -1.371503   0.020165  0.796223
3      -0.127658 -0.701533  0.070739  … -0.507737  -0.024208  0.371960
4       1.038374 -0.075846  0.030453  … -0.541172  -0.174950  0.355749
…             …         …         …  …        …          …         …
426447  0.318517 -0.139786 -0.311478  …  1.110343   1.651054  1.223993
426448  0.674817 -3.590065 -8.093407  … -7.755543 -12.704684 -4.443473
426449 -0.182592 -0.794665 -0.302585  … -0.843149   1.686313  1.296810
426450 -1.552406 -0.848937  0.131740  …  0.278056   0.626466  0.512756
426451  0.490887 -0.818383 -1.778571  … -2.974153  -4.531439 -1.914134

             V19       V20       V21       V22       V23       V27      Amount
0      -0.331603  0.384201 -0.218076 -0.203458 -0.213015 -0.262968   38.420000
1      -1.635634  0.364679 -1.495358 -0.083066  0.074612  0.089293   61.200000
2      -0.519459 -0.396476 -0.684454 -1.855269  0.171997 -0.061178  110.950000
3       1.561447  0.148760  0.097023  0.369957 -0.219266  0.114440   10.000000
4       1.375281  0.292972 -0.019733  0.165463 -0.080978  0.481769   22.000000
…             …         …         …         …         …         …         …
426447 -0.872929 -0.193495 -0.087659 -0.084135 -0.126067  0.046251    1.725917
426448  2.498435  0.070287  0.067269  0.467306 -0.642386  1.339039    0.864189
426449  2.017400  1.171567  0.451141  0.535937  1.149598 -0.019759  453.619063
426450  0.001894  0.495637 -0.788119  0.929233 -0.117767  0.112793    0.339493
426451  0.606703  0.222745  0.425152  0.368008 -0.397128 -0.335962   17.827416

[426452 rows x 25 columns]
```

[21]:
```
varth_features=var.get_support()
varth_features
```

[21]:
```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True, False, False, False,  True,
       False,  True])
```

[22]:
```
# Feature selection using SelectKBest feature selection
skbest = SelectKBest(k=10)
```

```
skbest.fit(x_train_s,y_train_s)
x_train_skbest=skbest.transform(x_train_s)
x_test_skbest=skbest.transform(x_test)
x_train_skbest.shape
```

[22]: (426452, 10)

[23]:
```
kbest_features=skbest.get_support()
kbest_features
```

[23]:
```
array([False,  True,  True,  True, False, False, False, False,  True,
        True,  True,  True, False,  True, False,  True,  True, False,
       False, False, False, False, False, False, False, False, False,
       False, False])
```

[24]:
```
# 10 best features using SelectKBest
best_features = SelectKBest(score_func=f_classif, k=10)
fit = best_features.fit(x_train_s,y_train_s)
df_scores = pd.DataFrame(fit.scores_)
df_columns = pd.DataFrame(x_train_s.columns)
feature_scores = pd.concat([df_columns, df_scores],axis=1)
feature_scores.columns = ['Feature_Name','Score']  # name output columns
print(feature_scores.nlargest(10,'Score'))         # print 10 best features
```

```
   Feature_Name          Score
13          V14  634408.764309
3            V4  477637.117795
10          V11  422096.275092
11          V12  415541.515783
9           V10  307081.858759
15          V16  240824.011003
8            V9  219463.452793
2            V3  203528.966849
16          V17  202096.481017
1            V2  151952.766954
```

[25]:
```
# calculate precision recall area under curve
def preci_auc(y_true, pred_prob):
   # calculate precision-recall curve
    p, r, _ = precision_recall_curve(y_true, pred_prob)
    # calculate area under curve
    return auc(r, p)
```

[26]:
```
# Evaluate a model
def evaluate_model(x, y, model):
    # Define evaluation procedure
    CV = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # Define the model evaluation the metric
```

```
    metric = make_scorer(preci_auc, needs_proba=True)
    # Evaluate model
    scores = cross_val_score(model, x, y, scoring='roc_auc', cv=CV, n_jobs=-1)
    return scores
```

[27]:
```
# define the reference model
model = DummyClassifier(strategy='constant', constant=1)
# Evaluate the model
scores = evaluate_model(x_train_skbest, y_train_s, model)
# summarize performance
print('Mean area under curve: %.3f (%.3f)' % (mean(scores), std(scores)))
```

```
Mean area under curve: 0.500 (0.000)
```

[28]:
```
# Normalize the input
scaler = StandardScaler()
scaler.fit(x_train_skbest)
x_train_norm = scaler.transform(x_train_skbest)
x_test_norm = scaler.transform(x_test_skbest)
```

[29]:
```
def model_val(x, y, classifier, scor, show):
  x = np.array(x)
  y = np.array(y)

  scores = cross_val_score(classifier, x, y, scoring=scor)

  if show == True:
    print("Score: {:.2f} (+/- {:.2f})".format(scores.mean(), scores.std()))

  return scores.mean()
```

[30]:
```
# List of models
rfc = RandomForestClassifier()
ctc = DecisionTreeClassifier()
sglc = SGDClassifier()
lr = LogisticRegression()

model = []
score = []

# Check model score
for classifier in (rfc, ctc, sglc, lr):
    model.append(classifier.__class__.__name__)
    score.append(model_val(x_train_norm, y_train_s, classifier, scor='roc_auc',␣
 ↪show=True))

pd.DataFrame(data=score, index=model, columns=['roc_auc'])
```

```
Score: 1.00 (+/- 0.00)
Score: 1.00 (+/- 0.00)
Score: 0.99 (+/- 0.00)
Score: 0.99 (+/- 0.00)
```

[30]:

	roc_auc
RandomForestClassifier	0.999975
DecisionTreeClassifier	0.997339
SGDClassifier	0.990387
LogisticRegression	0.990506

0.1.2 Random Forest Model Evaluation

[31]:
```
pipeline_rf = Pipeline([
    ('model', RandomForestClassifier(n_jobs=-1, random_state=1))
])
parm_gridscv_rf = {'model__n_estimators': [75]}
grid_rf = GridSearchCV(estimator=pipeline_rf, param_grid=parm_gridscv_rf,␣
 ↪scoring='roc_auc', n_jobs=-1,
                       pre_dispatch='2*n_jobs', cv=5, verbose=1,␣
 ↪return_train_score=False)
grid_rf.fit(x_train_norm, y_train_s)
```

```
Fitting 5 folds for each of 1 candidates, totalling 5 fits
```

[31]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('model',
                                        RandomForestClassifier(n_jobs=-1,
random_state=1))]),
             n_jobs=-1, param_grid={'model__n_estimators': [75]},
             scoring='roc_auc', verbose=1)
```

[32]:
```
pd.DataFrame(grid_rf.cv_results_)
```

[32]:
```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0     122.154728      0.336013          0.27644        0.058226

  param_model__n_estimators                           params  split0_test_score  \
0                        75  {'model__n_estimators': 75}           0.999974

   split1_test_score  split2_test_score  split3_test_score  split4_test_score  \
0           0.999987           0.999925           0.999961           0.999959

   mean_test_score  std_test_score  rank_test_score
0         0.999961        0.000021                1
```

[33]:
```
grid_rf.best_score_, grid_rf.best_params_
```

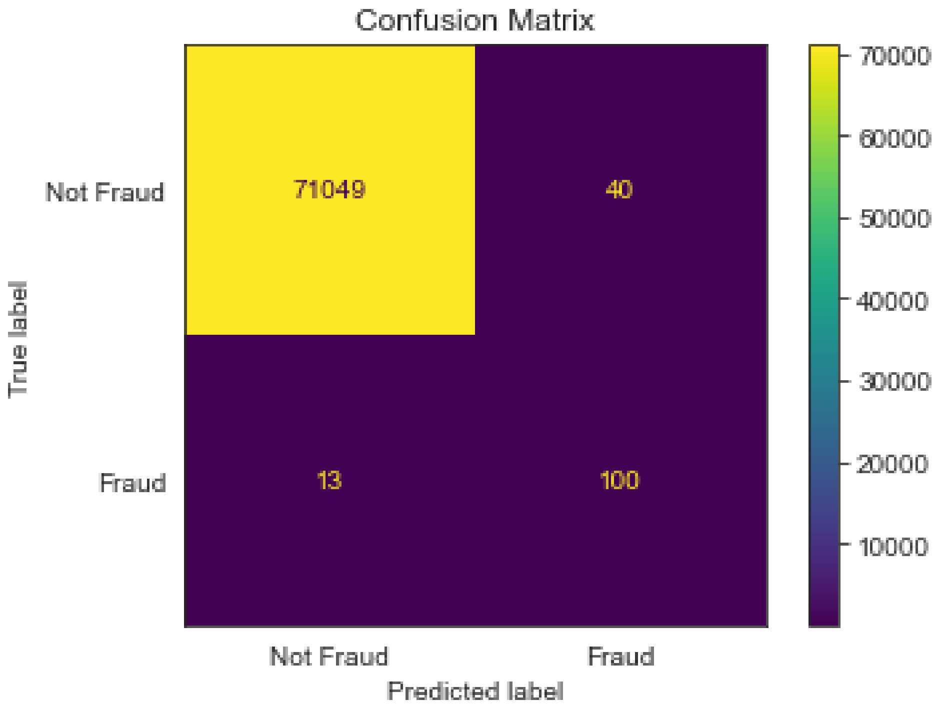
[33]: (0.9999611455402032, {'model__n_estimators': 75})

0.1.3 Test Random Forest model

[34]:
```python
# Prediction for the test set
y_pred_test = grid_rf.predict(x_test_norm)
# Decimal places based on number of samples
dec = np.int64(np.ceil(np.log10(len(y_test))))

print('Confusion Matrix')
print(confusion_matrix(y_test, y_pred_test), '\n')

print('Classification report')
print(classification_report(y_test, y_pred_test, digits=dec))

print('Scalar Metrics')
format_str = '%%13s = %%.%if' % dec
if y_test.nunique() <= 2: # metrics for binary classification
    try:
        y_score = grid_rf.predict_proba(x_test_norm)[:,1]
    except:
        y_score = grid_rf.decision_function(x_test_norm)
    print(format_str % ('AUROC', roc_auc_score(y_test, y_score)))
```

```
Confusion Matrix
[[71049    40]
 [   13   100]]

Classification report
              precision    recall  f1-score   support

           0    0.99982   0.99944   0.99963     71089
           1    0.71429   0.88496   0.79051       113

    accuracy                        0.99926     71202
   macro avg    0.85705   0.94220   0.89507     71202
weighted avg    0.99936   0.99926   0.99930     71202

Scalar Metrics
        AUROC = 0.98299
```

[35]:
```python
# Plot confusion matrix
con_mat=confusion_matrix(y_test,y_pred_test,labels=[0,1])
cmatrix=ConfusionMatrixDisplay(confusion_matrix=con_mat,display_labels=["Not␣
↪Fraud","Fraud"])
cmatrix.plot()
plt.title("Confusion Matrix")
plt.show()
```



[36]:
```
log_loss(y_test, y_pred_test)
```

[36]: 0.025709771254003325

0.1.4 Logistic Regression Model Evaluation

[37]:
```
# Logistic regression model with different C values
parameters = {
    'tol': [0.00001, 0.0001, 0.001],
    'C': [1, 50, 100]
}

lgr = GridSearchCV(LogisticRegression(random_state=101, n_jobs=1,␣
 ↪max_iter=1000),
                     param_grid=parameters,
                     cv=3,
                     n_jobs=1,
                     scoring='roc_auc'
                    )
lgr.fit(x_train_norm, y_train_s)
clf = lgr.best_estimator_

print(lgr.best_estimator_)
print("The best classifier score:",lgr.best_score_)
```

```
LogisticRegression(C=1, max_iter=1000, n_jobs=1, random_state=101, tol=1e-05)
The best classifier score: 0.9905075847918005
```

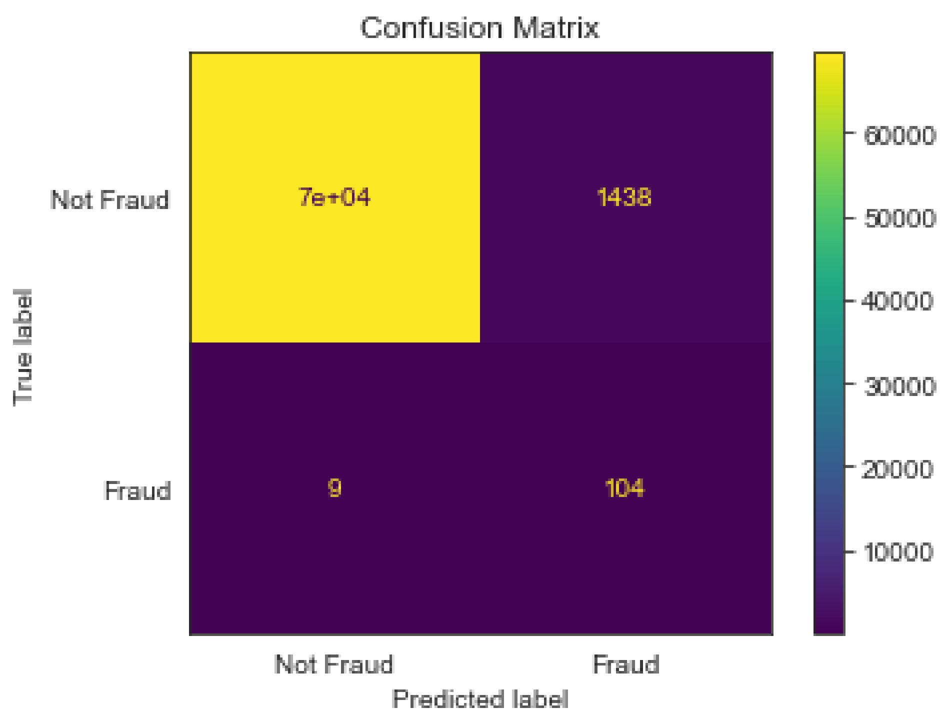
0.1.5 Test Logistic Regression Model

[38]:
```
y_pred_test1 = clf.predict(x_test_norm)
# Decimal places based on number of samples
dec = np.int64(np.ceil(np.log10(len(y_test))))

print('Confusion Matrix')
print(confusion_matrix(y_test, y_pred_test1), '\n')

print('Classification report')
print(classification_report(y_test, y_pred_test1, digits=dec))

print('Scalar Metrics')
format_str = '%%13s = %%.%if' % dec
if y_test.nunique() <= 2: # metrics for binary classification
    try:
        y_score1 = clf.predict_proba(x_test_norm)[:,1]
    except:
        y_score1 = clf.decision_function(X_test_norm)
    print(format_str % ('AUROC', roc_auc_score(y_test, y_score1)))
```

```
Confusion Matrix
[[69651  1438]
 [    9   104]]

Classification report
```

```
              precision    recall  f1-score   support

           0    0.99987   0.97977   0.98972     71089
           1    0.06744   0.92035   0.12568       113

    accuracy                        0.97968     71202
   macro avg    0.53366   0.95006   0.55770     71202
weighted avg    0.99839   0.97968   0.98835     71202

Scalar Metrics
        AUROC = 0.97698
```

[39]:
```
# Plot confusion matrix
con_mat1=confusion_matrix(y_test,y_pred_test1,labels=[0,1])
cmatrix1=ConfusionMatrixDisplay(confusion_matrix=con_mat1,display_labels=["Not␣
 ↪Fraud","Fraud"])
cmatrix1.plot()
plt.title("Confusion Matrix")
plt.show()
```



[40]:
```
log_loss(y_test, y_pred_test1)
```

[40]: 0.7019291489640804

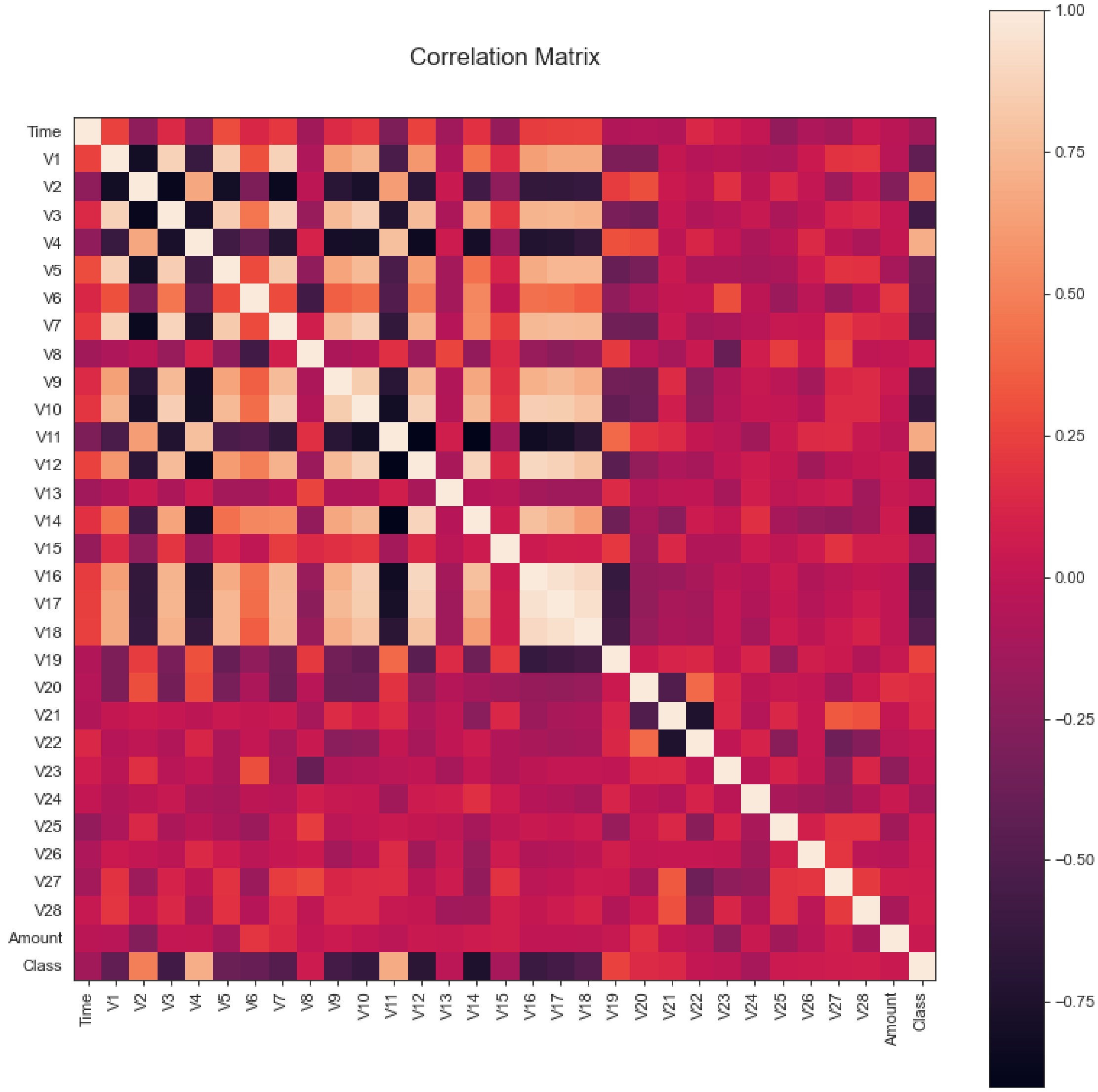
0.1.6 Train and Test after balancing

[138]:
```
df = df.sample(frac=1)

# Taking only few sample cases where length of sample is equal to number of␣
↪fraud cases.
fraud_df = df.loc[df['Class'] == 1]
non_fraud_df = df.loc[df['Class'] == 0].sample(len(fraud_df))

normal_distributed_df = pd.concat([fraud_df, non_fraud_df])

# Shuffle dataframe rows
new_df = normal_distributed_df.sample(frac=1, random_state=42)

new_df.head()
```

[138]:
```
            Time        V1        V2        V3        V4        V5        V6  \
180249  124450.0  2.109715 -0.458784 -1.436015 -1.452778 -0.364377 -1.586309
154697  102625.0 -4.221221  2.871121 -5.888716  6.890952 -3.404894 -1.154394
35009    37917.0  1.384650 -1.376614  0.417809 -1.567336 -1.471850  0.004499
230076  146179.0 -0.067672  4.251181 -6.540388  7.283657  0.513541 -2.635066
197586  132086.0 -0.361428  1.133472 -2.971360 -0.283073  0.371452 -0.574680

              V7        V8        V9  …       V21       V22       V23  \
180249  0.121582 -0.366016  1.753300  … -0.260543 -0.541017  0.275271
154697 -7.739928  2.851363 -2.507569  …  1.620591  1.567947 -0.578007
35009  -1.299660  0.053219 -1.772205  … -0.025284  0.148147 -0.171506
230076 -1.865911  0.780272 -3.868248  …  0.415437 -0.469938  0.007128
197586  4.031513 -0.934398 -0.768255  …  0.110815  0.563861 -0.408436

             V24       V25       V26       V27       V28  Amount  Class
180249 -0.098500 -0.117366 -0.484246 -0.014693 -0.058168    4.69      0
154697 -0.059045 -1.829169 -0.072429  0.136734 -0.599848    7.59      1
35009  -0.507789  0.388308 -0.075662  0.032970  0.019813   79.00      0
230076 -0.388147 -0.493398  0.466468  0.566370  0.262990    0.77      1
197586 -0.880079  1.408392 -0.137402 -0.001250 -0.182751  480.72      1

[5 rows x 31 columns]
```

[139]:
```
#Correlation Matrix
new_df = new_df[[col for col in new_df if new_df[col].nunique() > 1]] # keep␣
↪columns where there are more than 1 unique values
corr = new_df.corr()
plt.figure(num=None, figsize=(12, 12), dpi=80, facecolor='w', edgecolor='k')
corrMat = plt.matshow(corr, fignum = 1)
plt.xticks(range(len(corr.columns)), corr.columns, rotation=90)
plt.yticks(range(len(corr.columns)), corr.columns)
```

```
plt.gca().xaxis.tick_bottom()
plt.colorbar(corrMat)
plt.title(f'Correlation Matrix', fontsize=15)
plt.show()
```



[140]:
```
# Train and test data
x=new_df.drop(columns=["Time","Class"],axis="columns")
y=new_df.Class
```

[141]:
```
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=.
↪25,random_state=42)
```

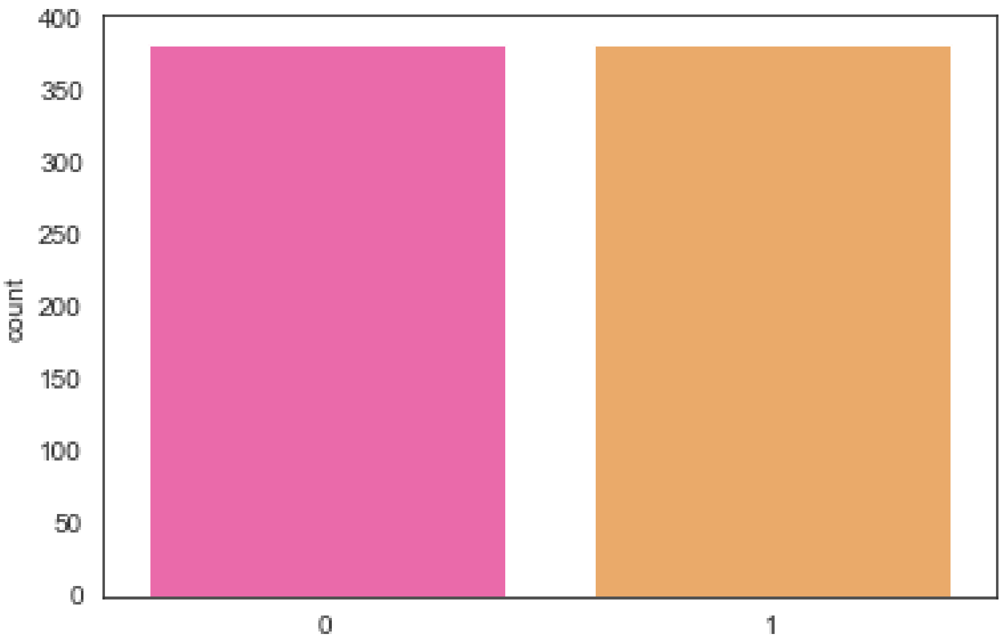
[142]:
```
# Details of training dataset
print("Shape of x_train dataset: ", x_train.shape)
print("Shape of y_train dataset: ", y_train.shape)
print("Shape of x_test dataset: ", x_test.shape)
print("Shape of y_test dataset: ", y_test.shape)
print("Before OverSampling, counts of label '1': {}".format(sum(y_train==1)))
print("Before OverSampling, counts of label '0': {} \n".format(sum(y_train==0)))
```

```
Shape of x_train dataset:  (738, 29)
Shape of y_train dataset:  (738,)
Shape of x_test dataset:  (246, 29)
Shape of y_test dataset:  (246,)
Before OverSampling, counts of label '1': 355
Before OverSampling, counts of label '0': 383
```

[143]:
```
# Oversample the training dataset
sm = SMOTE(random_state=2)
x_train_s, y_train_s = sm.fit_resample(x_train, y_train.ravel())

print('After OverSampling, the shape of train_x: {}'.format(x_train_s.shape))
print('After OverSampling, the shape of train_y: {} \n'.format(y_train_s.shape))

print("After OverSampling, counts of label '1', %: {}".format(sum(y_train_s==1)/
 ↪len(y_train_s)*100.0,2))
print("After OverSampling, counts of label '0', %: {}".format(sum(y_train_s==0)/
 ↪len(y_train_s)*100.0,2))

sns.countplot(x=y_train_s, data=new_df, palette='spring')
```

```
After OverSampling, the shape of train_x: (766, 29)
After OverSampling, the shape of train_y: (766,)

After OverSampling, counts of label '1', %: 50.0
After OverSampling, counts of label '0', %: 50.0
```

[143]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f88e9793220>
```



[144]:
```
# Feature selection using Variance Threshold with threshold of 0.5
var = VarianceThreshold(threshold=.5)
var.fit(x_train_s,y_train_s)
x_train_var=var.transform(x_train_s)
x_test_var=var.transform(x_test)
x_train_var.shape
```

[144]:
```
(766, 25)
```

[145]:
```
# Alternate way to perform feature selection and display the features
def variance_threshold_selector(data, threshold=0.5):
    selector = VarianceThreshold(threshold)
    selector.fit(data)
    return data[data.columns[selector.get_support(indices=True)]]
variance_threshold_selector(x_train_s, 0.5)
```

[145]:
```
            V1        V2         V3        V4        V5        V6        V7  \
0    -1.396204  2.618584  -6.036770  3.552454  1.030091 -2.950358 -1.528506
1    -0.518521  1.629458   1.998444  2.897556  0.275262  0.357826  0.409205
2   -11.205461  7.914633 -13.987752  4.333341 -8.484970 -3.506561 -8.935243
3    -3.821939  5.667247  -9.244963  8.246147 -4.368286 -3.450735 -8.427378
4    -6.618211  3.835943  -6.316453  1.844111 -2.476892 -1.886718 -3.817495
..         …         …          …        …         …         …         …
761  -0.794559  5.026167  -8.053515  7.465753  0.114331 -2.344652 -3.116944
762   1.113945  2.989537  -4.925187  4.266583  2.868245 -1.387594  0.927837
```

```
763  -5.101215  8.938518 -15.821726  10.301614 -4.601440 -3.373624 -11.049532
764  -2.801712 -0.263848  -5.731595   2.295171 -1.876356  0.590591  -1.160552
765  -3.878726  2.531532  -2.946649   4.093946 -1.516382 -0.855728  -4.386300

           V8        V9        V10  …        V16        V17       V18  \
0    0.189319 -1.433554  -5.569142  …  -2.497341  -1.588336  0.120289
1    0.044234 -0.102798   0.540645  …  -0.314980   0.979470 -0.289904
2    7.704449 -2.336584  -5.927359  …  -3.847293  -6.700637 -2.492616
3    2.305609 -5.338079 -12.011161  … -12.105602 -21.338195 -8.045436
4    0.613470 -1.482121  -4.868747  …  -3.939384  -7.164430 -2.434672
..        …         …          …  …        …          …         …
761  1.487198 -4.550283  -5.570927  …  -2.391081  -2.354116  0.327073
762 -1.555537 -1.560760  -1.588009  …   2.931121   5.723689  3.748270
763  5.368419 -5.673940 -11.652628  … -10.152260 -13.687070 -4.993079
764  1.445675 -1.869726  -6.197320  …  -4.095083  -5.457750 -1.690061
765 -0.063074 -1.199092  -5.433314  …  -6.866600  -9.750700 -4.469437

          V19       V20       V21       V22       V23       V27      Amount
0    0.170144  0.031795  0.143177 -0.390176  0.356029  0.062655    1.000000
1    0.293955  0.179944 -0.300826 -0.550231  0.043216  0.235407    6.480000
2    0.469554  0.860912  0.942593 -0.987848 -0.279446  1.084023   99.990000
3    0.156015  1.115247  1.990520  0.083353 -0.062264  1.869570   75.860000
4    0.235227 -0.953827  1.636622  0.038727  0.278218 -2.042403   57.730000
..        …         …         …         …         …         …         …
761  0.793628  0.825869  0.579042 -0.469174 -0.005520  0.657755    0.770000
762 -1.793996 -0.356969  1.099571 -0.795293  0.292327  0.146284    0.807329
763  1.137330  1.449403  2.000051  0.200852  0.602423  1.591059    1.000000
764 -0.267847  2.196937  1.401291  0.840844  1.323173  0.696839  723.210000
765  0.434441  0.019369  1.467101 -0.073473 -0.036148 -0.112660    0.929997

[766 rows x 25 columns]
```

[146]:
```
varth_features=var.get_support()
varth_features
```

[146]:
```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True, False, False, False,  True,
       False,  True])
```

[147]:
```
# Feature selection using SelectKBest feature selection
skbest = SelectKBest(k=10)
skbest.fit(x_train_s,y_train_s)
x_train_skbest=skbest.transform(x_train_s)
x_test_skbest=skbest.transform(x_test)
x_train_skbest.shape
```

[147]: (766, 10)

[148]:
```
kbest_features=skbest.get_support()
kbest_features
```

[148]:
```
array([False,  True,  True,  True, False, False, False, False,  True,
        True,  True,  True, False,  True, False,  True,  True, False,
       False, False, False, False, False, False, False, False, False,
       False, False])
```

[149]:
```
# 10 best features using SelectKBest
best_features = SelectKBest(score_func=f_classif, k=10)
fit = best_features.fit(x_train_s,y_train_s)
df_scores = pd.DataFrame(fit.scores_)
df_columns = pd.DataFrame(x_train_s.columns)
feature_scores = pd.concat([df_columns, df_scores],axis=1)
feature_scores.columns = ['Feature_Name','Score']  # name output columns
print(feature_scores.nlargest(10,'Score'))         # print 10 best features
```

```
   Feature_Name       Score
13          V14  961.878974
3            V4  768.888297
10          V11  703.241720
11          V12  661.244182
9           V10  534.163705
15          V16  453.489625
2            V3  390.362289
16          V17  384.380278
8            V9  369.572827
1            V2  247.774452
```

[150]:
```
# calculate precision recall area under curve
def preci_auc(y_true, pred_prob):
   # calculate precision-recall curve
    p, r, _ = precision_recall_curve(y_true, pred_prob)
    # calculate area under curve
    return auc(r, p)
```

[151]:
```
# Evaluate a model
def evaluate_model(x, y, model):
    # Define evaluation procedure
    CV = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # Define the model evaluation the metric
    metric = make_scorer(preci_auc, needs_proba=True)
    # Evaluate model
    scores = cross_val_score(model, x, y, scoring='roc_auc', cv=CV, n_jobs=-1)
    return scores
```

[152]:
```
# define the reference model
model = DummyClassifier(strategy='constant', constant=1)
# Evaluate the model
scores = evaluate_model(x_train_skbest, y_train_s, model)
# summarize performance
print('Mean area under curve: %.3f (%.3f)' % (mean(scores), std(scores)))
```

```
Mean area under curve: 0.500 (0.000)
```

[153]:
```
# Normalize the input
scaler = StandardScaler()
scaler.fit(x_train_skbest)
x_train_norm = scaler.transform(x_train_skbest)
x_test_norm = scaler.transform(x_test_skbest)
```

[154]:
```
def model_val(x, y, classifier, scor, show):
  x = np.array(x)
  y = np.array(y)

  scores = cross_val_score(classifier, x, y, scoring=scor)

  if show == True:
    print("Score: {:.2f} (+/- {:.2f})".format(scores.mean(), scores.std()))

  return scores.mean()
```

[155]:
```
# List of models
rfc = RandomForestClassifier()
ctc = DecisionTreeClassifier()
sglc = SGDClassifier()
lr = LogisticRegression()

model = []
score = []

# Check model score
for classifier in (rfc, ctc, sglc, lr):
    model.append(classifier.__class__.__name__)
    score.append(model_val(x_train_norm, y_train_s, classifier, scor='roc_auc',␣
 ↪show=True))

pd.DataFrame(data=score, index=model, columns=['roc_auc'])
```

```
Score: 0.97 (+/- 0.01)
Score: 0.89 (+/- 0.02)
Score: 0.98 (+/- 0.01)
Score: 0.98 (+/- 0.01)
```

```
[155]:                             roc_auc
       RandomForestClassifier    0.971959
       DecisionTreeClassifier    0.889183
       SGDClassifier             0.975939
       LogisticRegression        0.978982
```

0.1.7 Random Forest Model Evaluation after balancing

```
[156]: pipeline_rf = Pipeline([
           ('model', RandomForestClassifier(n_jobs=-1, random_state=1))
       ])
       parm_gridscv_rf = {'model__n_estimators': [75]}
       grid_rf = GridSearchCV(estimator=pipeline_rf, param_grid=parm_gridscv_rf,␣
        ↪scoring='roc_auc', n_jobs=-1,
                              pre_dispatch='2*n_jobs', cv=5, verbose=1,␣
        ↪return_train_score=False)
       grid_rf.fit(x_train_norm, y_train_s)
```

```
Fitting 5 folds for each of 1 candidates, totalling 5 fits
```

```
[156]: GridSearchCV(cv=5,
                    estimator=Pipeline(steps=[('model',
                                               RandomForestClassifier(n_jobs=-1,
       random_state=1))]),
                    n_jobs=-1, param_grid={'model__n_estimators': [75]},
                    scoring='roc_auc', verbose=1)
```

```
[157]: pd.DataFrame(grid_rf.cv_results_)
```

```
[157]:    mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
       0         0.2282      0.016534         0.022987        0.001361

         param_model__n_estimators                       params  split0_test_score  \
       0                        75  {'model__n_estimators': 75}           0.980098

          split1_test_score  split2_test_score  split3_test_score  split4_test_score  \
       0           0.965311           0.969754           0.983424           0.968301

          mean_test_score  std_test_score  rank_test_score
       0         0.973378        0.007072                1
```

```
[158]: grid_rf.best_score_, grid_rf.best_params_
```

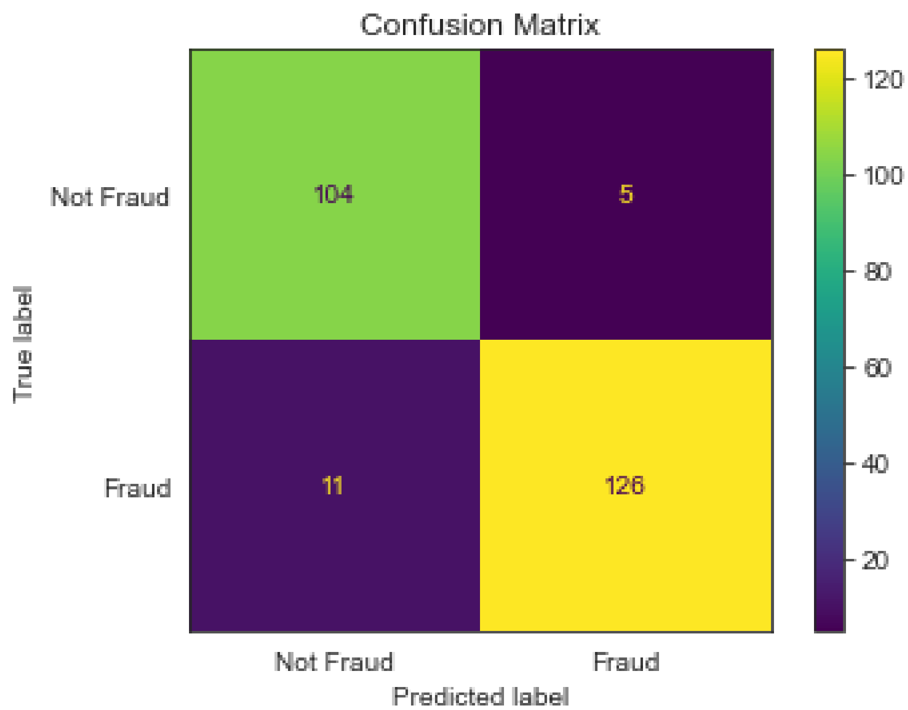
```
[158]: (0.9733777329983756, {'model__n_estimators': 75})
```

0.1.8 Test Random Forest model after balancing

[159]:
```
# Prediction for the test set
y_pred_test = grid_rf.predict(x_test_norm)
# Decimal places based on number of samples
dec = np.int64(np.ceil(np.log10(len(y_test))))

print('Confusion Matrix')
print(confusion_matrix(y_test, y_pred_test), '\n')

print('Classification report')
print(classification_report(y_test, y_pred_test, digits=dec))

print('Scalar Metrics')
format_str = '%%13s = %%.%if' % dec
if y_test.nunique() <= 2: # metrics for binary classification
    try:
        y_score = grid_rf.predict_proba(x_test_norm)[:,1]
    except:
        y_score = grid_rf.decision_function(x_test_norm)
    print(format_str % ('AUROC', roc_auc_score(y_test, y_score)))
```

```
Confusion Matrix
[[104   5]
 [ 11 126]]

Classification report
              precision    recall  f1-score   support

           0      0.904     0.954     0.929       109
           1      0.962     0.920     0.940       137

    accuracy                          0.935       246
   macro avg      0.933     0.937     0.934       246
weighted avg      0.936     0.935     0.935       246

Scalar Metrics
        AUROC = 0.973
```

[160]:
```
# Plot confusion matrix
con_mat=confusion_matrix(y_test,y_pred_test,labels=[0,1])
cmatrix=ConfusionMatrixDisplay(confusion_matrix=con_mat,display_labels=["Not␣
 ↪Fraud","Fraud"])
cmatrix.plot()
plt.title("Confusion Matrix")
plt.show()
```



[161]:
```
log_loss(y_test, y_pred_test)
```

[161]: 2.2464407329500897

0.1.9 Logistic Regression Model Evaluation after balancing

[162]:
```python
# Logistic regression model with different C values
parameters = {
    'tol': [0.00001, 0.0001, 0.001],
    'C': [1, 50, 100]
}

lgr = GridSearchCV(LogisticRegression(random_state=101, n_jobs=1,␣
↪max_iter=1000),
                     param_grid=parameters,
                     cv=3,
                     n_jobs=1,
                     scoring='roc_auc'
                    )
lgr.fit(x_train_norm, y_train_s)
clf = lgr.best_estimator_

print(lgr.best_estimator_)
print("The best classifier score:",lgr.best_score_)
```

```
LogisticRegression(C=1, max_iter=1000, n_jobs=1, random_state=101, tol=1e-05)
The best classifier score: 0.9782445712352362
```

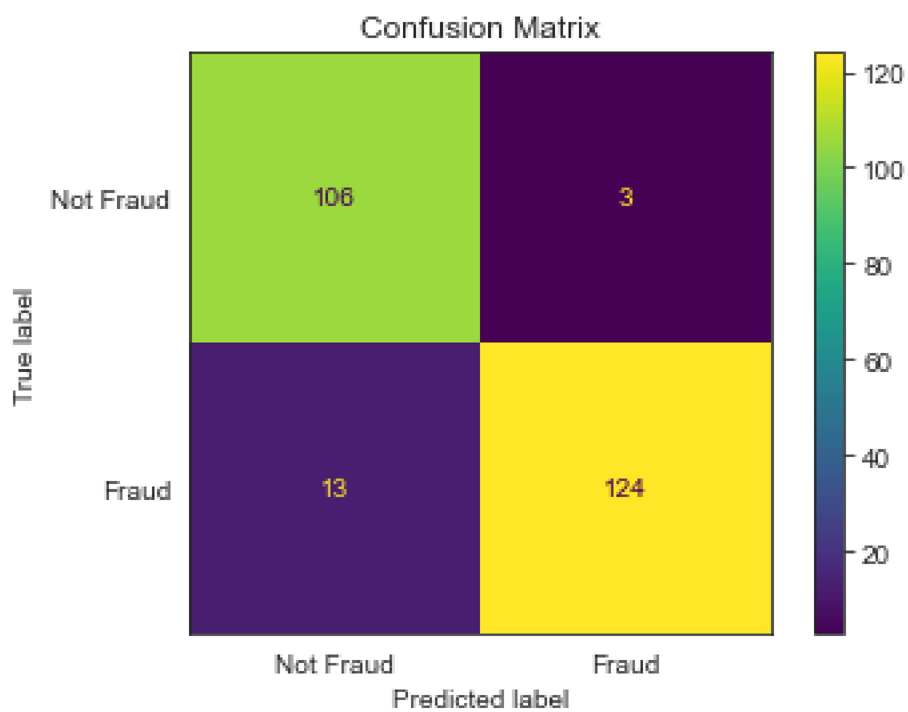
0.1.10 Test Logistic Regression Model after balancing

[163]:
```python
y_pred_test1 = clf.predict(x_test_norm)
# Decimal places based on number of samples
dec = np.int64(np.ceil(np.log10(len(y_test))))

print('Confusion Matrix')
print(confusion_matrix(y_test, y_pred_test1), '\n')

print('Classification report')
print(classification_report(y_test, y_pred_test1, digits=dec))

print('Scalar Metrics')
format_str = '%%13s = %%.%if' % dec
if y_test.nunique() <= 2: # metrics for binary classification
    try:
        y_score1 = clf.predict_proba(x_test_norm)[:,1]
    except:
        y_score1 = clf.decision_function(X_test_norm)
    print(format_str % ('AUROC', roc_auc_score(y_test, y_score1)))
```

```
Confusion Matrix
[[106   3]
 [ 13 124]]

Classification report
```

```
              precision    recall  f1-score   support

           0      0.891     0.972     0.930       109
           1      0.976     0.905     0.939       137

    accuracy                          0.935       246
   macro avg      0.934     0.939     0.935       246
weighted avg      0.938     0.935     0.935       246

Scalar Metrics
        AUROC = 0.985
```

[164]:
```python
# Plot confusion matrix
con_mat1=confusion_matrix(y_test,y_pred_test1,labels=[0,1])
cmatrix1=ConfusionMatrixDisplay(confusion_matrix=con_mat1,display_labels=["Not␣
↪Fraud","Fraud"])
cmatrix1.plot()
plt.title("Confusion Matrix")
plt.show()
```



[165]:
```python
log_loss(y_test, y_pred_test1)
```

[165]: 2.246434232157974

```
[ ]:
```