cc_fraud_detection

November 10, 2021

[1]:
```
!pip install -q scikit_plot
```

[2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import scikitplot as skplt

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from imblearn.under_sampling import RandomUnderSampler

from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier

from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus

from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_auc_score

#setting style to seaborn
sns.set_style("dark")
```

[3]:
```
# data https://www.kaggle.com/mlg-ulb/creditcardfraud?select=creditcard.csv
# alternate https://www.dropbox.com/s/b44o3t3ehmnx2b7/creditcard.csv?dl=1

#importing the data to be used in the analysis and split it into training and␣
↪test data.
file_path = "https://www.dropbox.com/s/b44o3t3ehmnx2b7/creditcard.csv?dl=1"

# importing the dataset to a dataframe
df = pd.read_csv(file_path)
print("Original Dataset dimensions:", df.shape)
```

```
#splitting test data
test = df.sample(frac=0.15, random_state=0)
df = df.drop(test.index)

print("Train data dimensions: ", df.shape)
print("Test data dimensions: ", test.shape)
```

```
Original Dataset dimensions: (284807, 31)
Train data dimensions:  (242086, 31)
Test data dimensions:  (42721, 31)
```

[4]:
```
#Checking the first entries for the dataset
df.head()
```

[4]:
```
   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       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

        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 rows x 31 columns]
```

[5]:
```
#checking the statistical summary for the dataset

"""
Note:
Time = Number of seconds elapsed between this transaction and the first␣
 ↪transaction in the dataset
Amount = Transaction amount

Class:
- Grouped into two segments:
0 and 1 as transaction type indicators.\
```

```
0 - Normal Transaction
1 - Fraudulent Transacti
"""
df.describe()
```

[5]:

```
                Time             V1             V2             V3  \
count  242086.000000  242086.000000  242086.000000  242086.000000
mean    94857.597379       0.001456      -0.000800      -0.001004
std     47490.660832       1.956513       1.654883       1.514580
min         0.000000     -56.407510     -72.715728     -33.680984
25%     54234.000000      -0.920406      -0.598659      -0.893374
50%     84747.000000       0.019230       0.064567       0.177607
75%    139362.000000       1.316034       0.803174       1.026561
max    172792.000000       2.451888      22.057729       9.382558

                  V4             V5             V6             V7  \
count  242086.000000  242086.000000  242086.000000  242086.000000
mean        0.000647      -0.000986      -0.001196      -0.000033
std         1.417228       1.366284       1.326879       1.223095
min        -5.683171     -42.147898     -26.160506     -43.557242
25%        -0.848236      -0.693615      -0.769025      -0.553805
50%        -0.018959      -0.054544      -0.274310       0.040344
75%         0.743691       0.611455       0.397688       0.570104
max        16.875344      34.801666      23.917837      44.054461

                  V8             V9  …            V21            V22  \
count  242086.000000  242086.000000  …  242086.000000  242086.000000
mean       -0.000416      -0.000725  …      -0.000015       0.000606
std         1.199718       1.099350  …       0.734189       0.726284
min       -73.216718     -13.434066  …     -34.830382     -10.933144
25%        -0.208857      -0.645058  …      -0.228492      -0.542962
50%         0.022160      -0.051370  …      -0.028987       0.007162
75%         0.327186       0.597195  …       0.187064       0.529814
max        20.007208      15.594995  …      27.202839      10.503090

                 V23            V24            V25            V26  \
count  242086.000000  242086.000000  242086.000000  242086.000000
mean       -0.000482       0.000199       0.000048      -0.000269
std         0.629651       0.605150       0.521574       0.482084
min       -44.807735      -2.822684     -10.295397      -2.604551
25%        -0.161941      -0.354494      -0.316853      -0.327387
50%        -0.011175       0.040764       0.017175      -0.052337
75%         0.147358       0.439320       0.350795       0.241214
max        22.528412       4.022866       7.519589       3.517346

                 V27            V28         Amount          Class
```

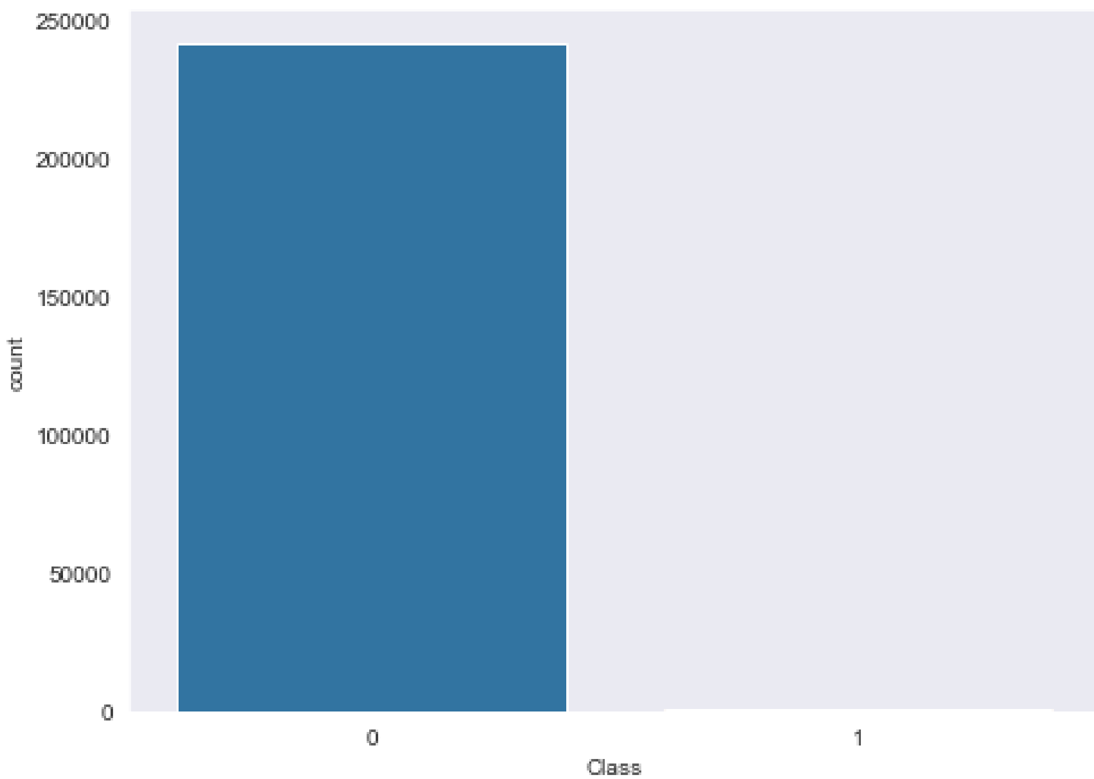
```
count  242086.000000  242086.000000  242086.000000  242086.000000
mean       -0.000727       0.000065      88.612429       0.001727
std         0.401490       0.327734     247.655020       0.041517
min       -22.565679     -11.710896       0.000000       0.000000
25%        -0.070744      -0.052903       5.662500       0.000000
50%         0.001131       0.011209      22.000000       0.000000
75%         0.090776       0.078234      77.580000       0.000000
max        12.152401      33.847808   19656.530000       1.000000

[8 rows x 31 columns]
```

[6]:
```python
#Calculating the column with the most null entries
df.isnull().sum().max()
```

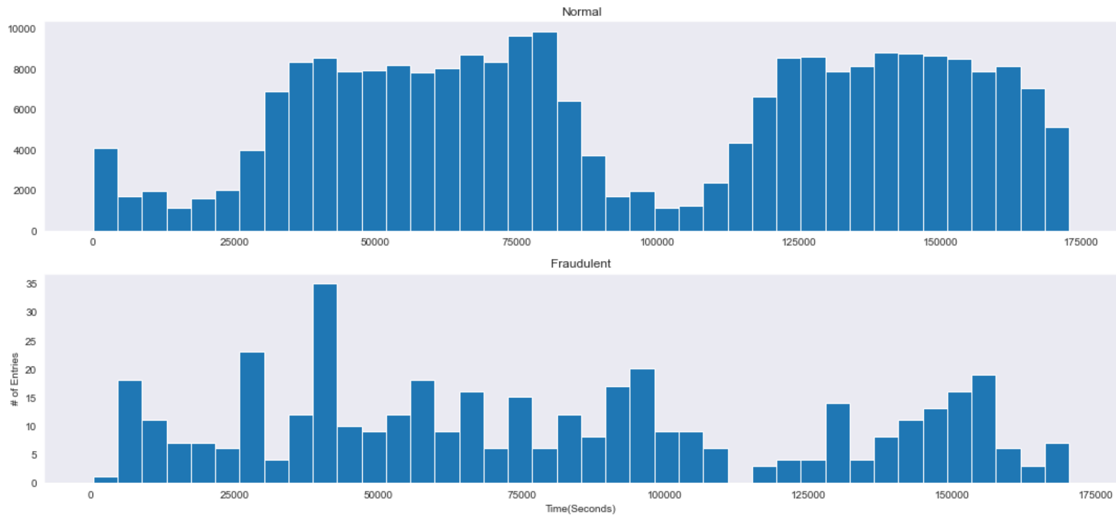
[6]: 0

[7]:
```python
#Checking the number of entries per different classes
print(df.Class.value_counts())

print("\nFrauds represents {}% of the dataset \n".format(round(df[df.Class ==␣
↪1].shape[0]*100/df.shape[0],2)))

fig, ax = plt.subplots(figsize=(7,5))
sns.countplot(x="Class", data=df, ax=ax)
plt.tight_layout();
```

```
0    241668
1       418
Name: Class, dtype: int64

Frauds represents 0.17% of the dataset
```

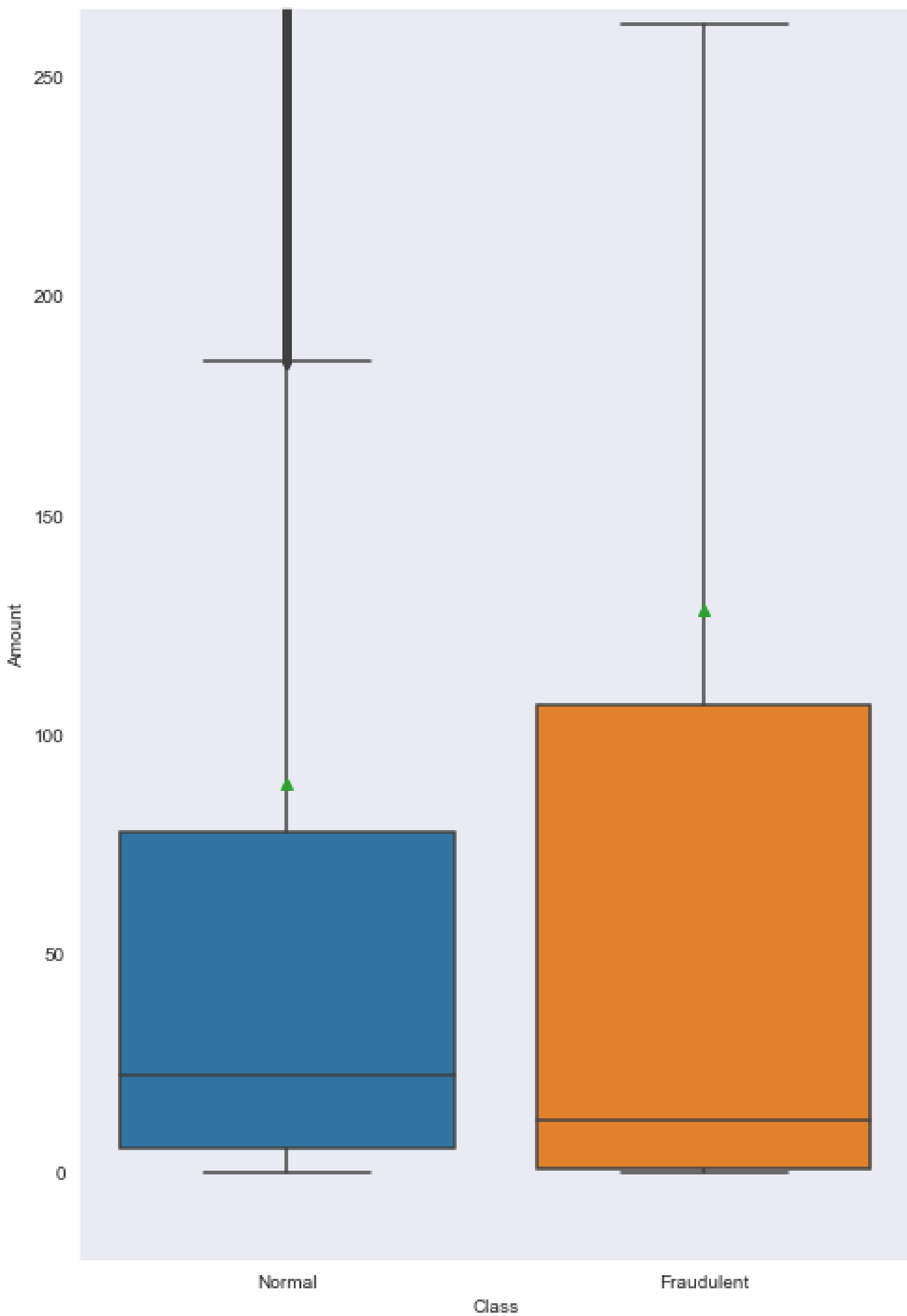



```
[8]: # try to enhance the unbalanced dataset(enhancment)
     # checking the distribution for the variable Time in normal and fraudulent␣
      ↪transactions:


     #Distirbution for "Time" per class

     n_bins=40

     fig, (ax1, ax2) = plt.subplots(nrows=2, figsize=(15,7))

     df[df.Class == 0].Time.hist(ax=ax1, bins=n_bins, grid=False)
     ax1.set_title("Normal")

     df[df.Class == 1].Time.hist(ax=ax2, bins=n_bins, grid=False)
     ax2.set_title("Fraudulent")
     ax2.set_xlabel("Time(Seconds)")
     ax2.set_ylabel("# of Entries")

     plt.tight_layout()
```

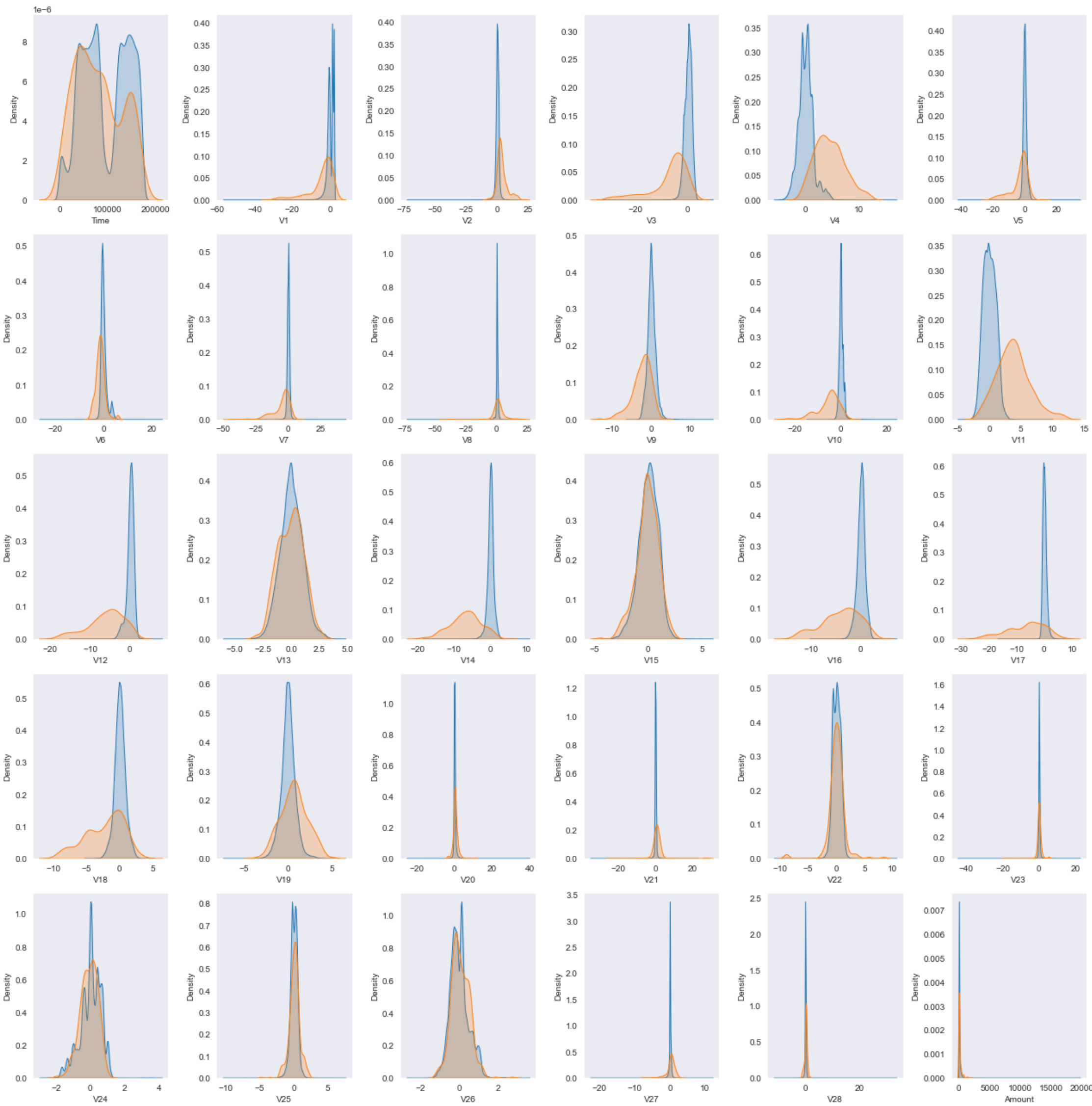



```
[9]: """
The difference in the number of entries may be explained by the different␣
↪periods of the day, as day and night,
when the number of transactions vastly differs.

Plotting a boxplot for the Amount variable in normal and fraudulent␣
↪transactions:
"""

#Calculating the superior limit for Amount Variable
q3 = df[df.Class == 1].Amount.quantile(.75)
q1 = df[df.Class == 1].Amount.quantile(.25)
IQR = q3 - q1

sup_limit = q3 + 1.5*IQR
```

```
[10]: #plottig the boxplot for the the normal and fraudulent distibution
fig, ax = plt.subplots(figsize=(7,10))
sns.boxplot(x="Class", y="Amount", data=df, ax=ax, showmeans=True)
ax.set_ylim(-20, sup_limit)
ax.set_xticklabels(["Normal", "Fraudulent"])

plt.tight_layout()
```


250
200
150
100
50
0
Amount
Normal
Fraudulent
Class

```
[11]: """

      Although the median is lower for the fraudulent transactions (represented by␣
       ↪the black line inside each box),
      the mean (represented by the green triangle) is higher for fraudulent␣
       ↪transactions than for normal ones.

      We can also plot a density plot for each variable, separating fraudulent and␣
       ↪normal transactions.
      Here, we are searching for variables that are significantly different for␣
       ↪normal and fraudulent transactions:

      """




      #plotting the density plot

      columns_names = df.drop(labels=["Class"], axis=1).columns

      df_normal = df[df.Class == 0]
      df_fraud = df[df.Class == 1]

      fig, ax = plt.subplots(nrows=5,ncols=6, figsize=(18,18))
      fig.subplots_adjust(hspace=1, wspace=1)

      idx = 0

      for col in columns_names:
        idx+=1
        plt.subplot(5,6,idx)
        sns.kdeplot(df_normal[col], label = "Normal", shade=True)
        sns.kdeplot(df_fraud[col], label = "Fraud", shade=True)

      plt.tight_layout()
```



[12]:

```
"""

Some variables as 'V14' and 'V4' have pretty different behavior for the two␣
↪classes.

After the initial exploratory analysis we can state that:

    (1)The variables Time and Amount are not normalized and will need to be␣
↪transformed before training the model.
    (2)The dataset is extremely unbalanced, representing a challenge for the␣
↪analysis.
```

```
    (3)The mean amount for fraudulent transactions is higher than the normal␣
↪transaction mean amount.
    (4)Some variables, as V14 and V4, have a clear different behavior for␣
↪normal and fraudulent transactions.

Based on that, we can now prepare the data before training the model.

==> Preparing the data <==

First, we will normalize the Time and Amount variables. Since their dimensions␣
↪are different from all the other variables, our model will be biased by␣
↪these columns if we don't normalize them.

"""

#normalizing "Amount" and "Time" variables
df_copy = df.copy()


std_scaler = StandardScaler()
df_copy["std_amount"] = std_scaler.fit_transform(df_copy.Amount.values.
↪reshape(-1,1))
df_copy["std_time"] = std_scaler.fit_transform(df_copy.Time.values.
↪reshape(-1,1))

df_copy.drop(["Time","Amount"], axis=1, inplace=True)
```

[13]:
```
#checking the first entries
df_copy.head()
```

[13]:
```
         V1        V2        V3        V4        V5        V6        V7  \
0 -1.359807 -0.072781  2.536347  1.378155 -0.338321  0.462388  0.239599
1  1.191857  0.266151  0.166480  0.448154  0.060018 -0.082361 -0.078803
2 -1.358354 -1.340163  1.773209  0.379780 -0.503198  1.800499  0.791461
3 -0.966272 -0.185226  1.792993 -0.863291 -0.010309  1.247203  0.237609
4 -1.158233  0.877737  1.548718  0.403034 -0.407193  0.095921  0.592941

         V8        V9       V10  …       V22       V23       V24       V25  \
0  0.098698  0.363787  0.090794  …  0.277838 -0.110474  0.066928  0.128539
1  0.085102 -0.255425 -0.166974  … -0.638672  0.101288 -0.339846  0.167170
2  0.247676 -1.514654  0.207643  …  0.771679  0.909412 -0.689281 -0.327642
3  0.377436 -1.387024 -0.054952  …  0.005274 -0.190321 -1.175575  0.647376
4 -0.270533  0.817739  0.753074  …  0.798278 -0.137458  0.141267 -0.206010

        V26       V27       V28  Class  std_amount  std_time
0 -0.189115  0.133558 -0.021053      0    0.246341 -1.997399
1  0.125895 -0.008983  0.014724      0   -0.346945 -1.997399
```

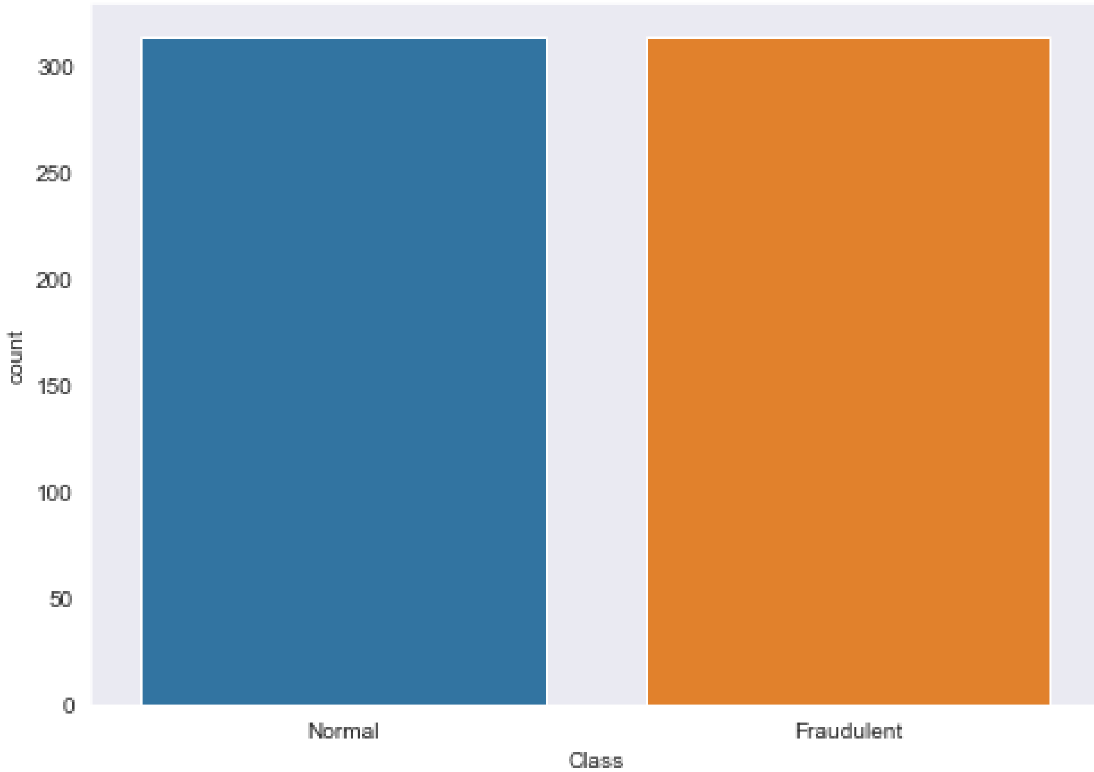
```
2 -0.139097 -0.055353 -0.059752      0    1.171178 -1.997378
3 -0.221929  0.062723  0.061458      0    0.140872 -1.997378
4  0.502292  0.219422  0.215153      0   -0.075195 -1.997357

[5 rows x 31 columns]
```

[14]:
```python
#splitting the dataset into train and validation
np.random.seed(2)
X = df_copy.drop("Class", axis=1)
y = df_copy["Class"]

X_train, X_val, y_train, y_val = train_test_split(X,y, shuffle=True, stratify=y)
```

[19]:
```python
"""

Last but not least, since the fraudulent transaction only accounts for 0,17% of␣
↪the dataset,
we should balance the dataset to have better results with our models.

Among others, there are two ways in which we can solve this problem:

    1. Over Sampling - Creates new entries for the minority class based on the␣
↪existing samples.
    2. Under Sampling - Randomly deletes entries for the majority class.

Here we will choose the under sampling method and apply it to the data:

"""

#Balancing the dataset
rus = RandomUnderSampler()

X_rus, y_rus = rus.fit_resample(X_train, y_train)
```

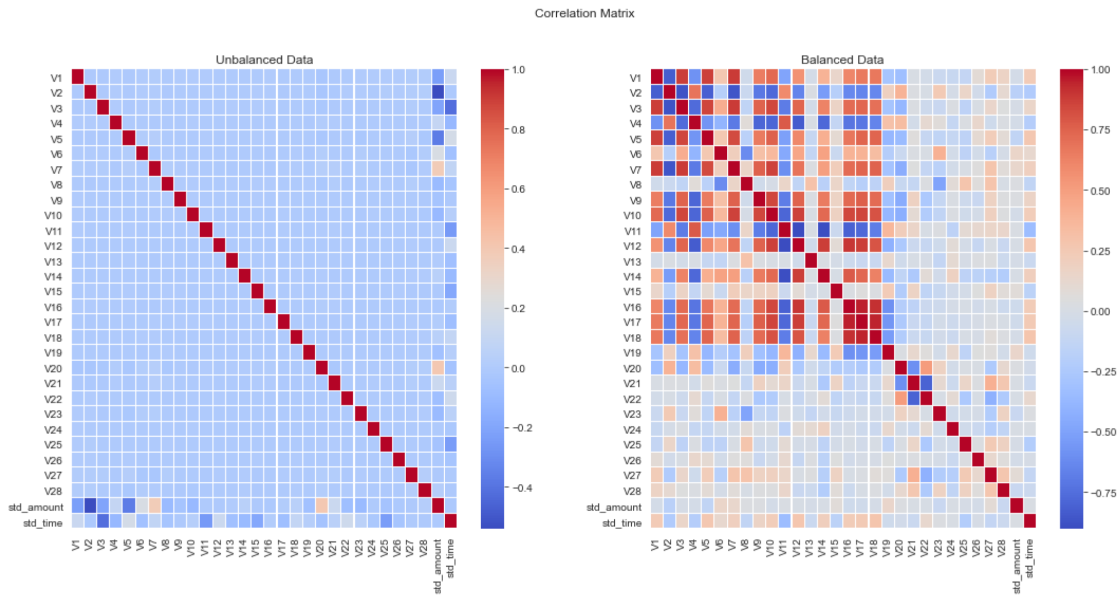
[17]:
```python
#Plotting balanced values
print(pd.Series(y_rus).value_counts())

fig, ax = plt.subplots(figsize=(7,5))

sns.countplot(x=y_rus, ax=ax)
ax.set_xticklabels(labels=["Normal","Fraudulent"])

plt.tight_layout()
```

```
0    313
1    313
Name: Class, dtype: int64
```



[20]:
```
"""

Now we have the same number for fraudulent and normal transactions. To better␣
↪understand
the influence of unbalanced data, let´s plot a correlation matrix for the␣
↪balanced and unbalanced dataset:

"""

#plotting the correlation matrix for unbalanced and balanced data

imb_corr = X_train.corr()
corr = pd.DataFrame(X_rus).corr()

fig, ax = plt.subplots(nrows=1,ncols=2,figsize=(18,8))
fig.suptitle("Correlation Matrix")

sns.heatmap(corr, ax=ax[1], cmap="coolwarm", linewidths=.1,
            xticklabels=imb_corr.columns, yticklabels=imb_corr.columns)
ax[1].set_title("Balanced Data")
```

```
sns.heatmap(imb_corr, ax=ax[0], cmap="coolwarm", linewidth=.1)
ax[0].set_title("Unbalanced Data")

plt.show()
```

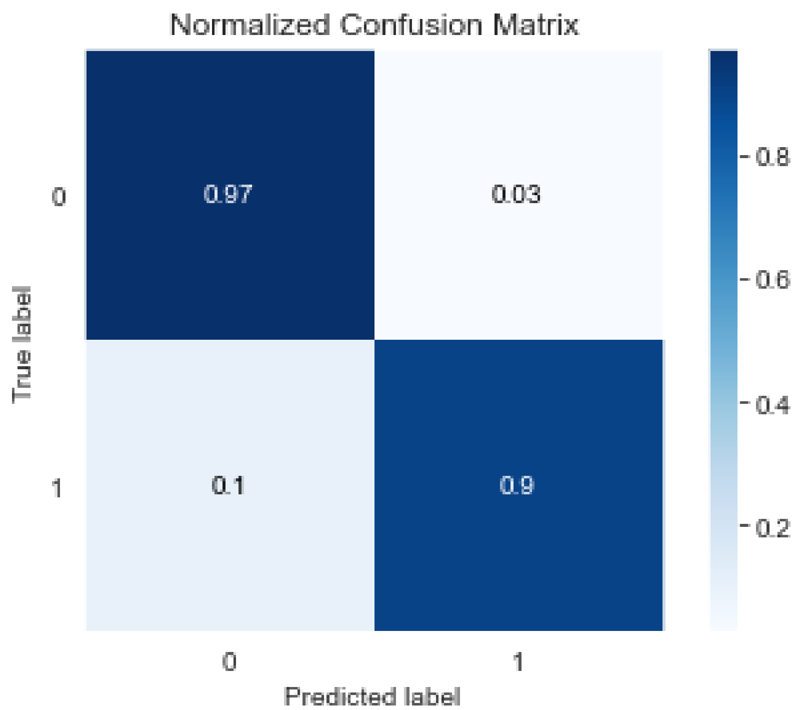


[21]:
```
"""

Training two models that use different algorithms to classify
the data: Logistic Regression and Decision Tree Classifier. Afterward, trying
 ↪to analyze which
model better classifies news transactions.

"""

#Building the first model using Logisitic Regression
lr_model = LogisticRegression()

lr_model.fit(X_rus, y_rus)

y_pred = lr_model.predict(X_val)
```

[22]:
```
"""

Metrics for evaluating classification models are:

->Precision - the proportion of predicted Positives that are truly Positive
->Recall - the proportion of actual positives correctly classified
```

```
->f1-score - the harmonic mean of precision and recall
->accuracy - the proportion of true results among the total number of cases␣
↪examined
->ROC score - indicates how well the probabilities from the positive classes␣
↪are separated
  from the negative classes

"""

#Checking the metrics for the first model
print("Classification Report for Logisitic Regression Model: \n\n",␣
↪classification_report(y_val, y_pred, digits=4))

#ROC
print("ROC Curve: \n\n", round(roc_auc_score(y_val, y_pred),4), "\n")

#plotting the confusion matrix
skplt.metrics.plot_confusion_matrix(y_val, y_pred, normalize=True);
```

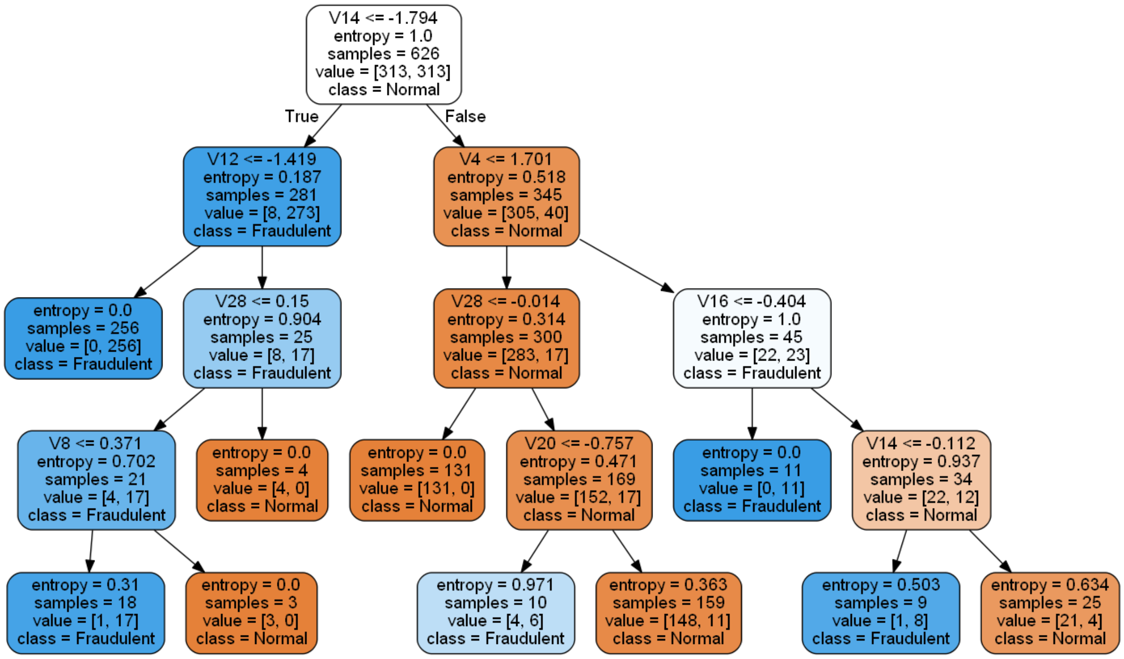
```
Classification Report for Logisitic Regression Model:

              precision    recall  f1-score   support

           0     0.9998    0.9659    0.9826     60417
           1     0.0441    0.9048    0.0841       105

    accuracy                         0.9658     60522
   macro avg     0.5220    0.9353    0.5333     60522
weighted avg     0.9982    0.9658    0.9810     60522

ROC Curve:

 0.9353
```



[23]:
```
#Building other model using Decision Tree Classifier

tree_depth = 4

dt_model = DecisionTreeClassifier(criterion="entropy", max_depth=tree_depth)

dt_model.fit(X_rus, y_rus)

y_pred = dt_model.predict(X_val)
```

[24]:
```
#plotting the Decision Tree

# creating the dot
dot = export_graphviz(dt_model, filled=True, rounded=True, feature_names=X.
 ↪columns, class_names=["Normal", "Fraudulent"])

#plotting
graph = pydotplus.graph_from_dot_data(dot)
Image(graph.create_png())
```

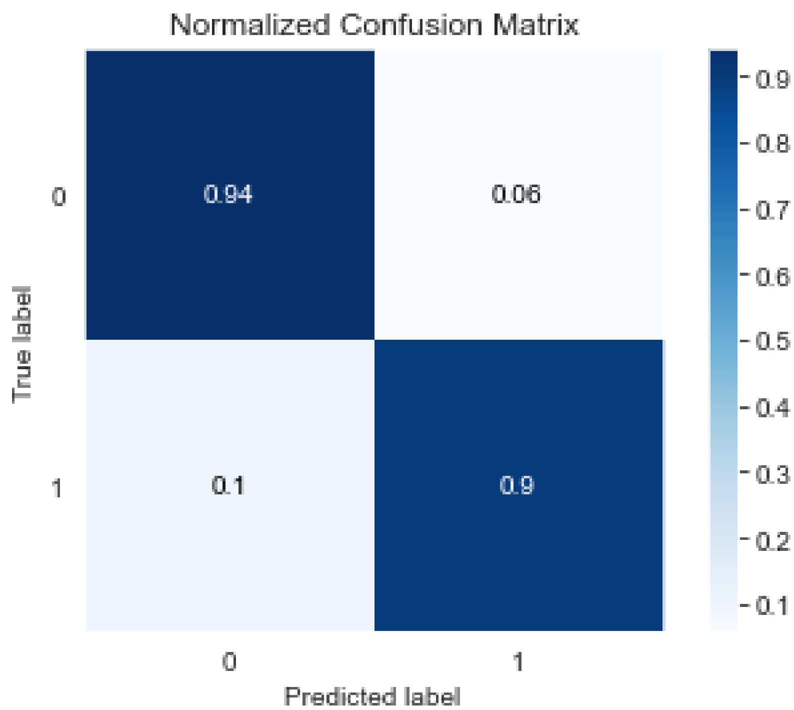
[24]:



[25]:

```python
#Checking the metrics for the second model
print("Classification Report for the Decision Tree Classifier: \n\n",
 ↪classification_report(y_val, y_pred, digits=4))

#ROC
print("ROC Curve: \n\n", round(roc_auc_score(y_val, y_pred),4), "\n")

#plotting the confusion matrix
skplt.metrics.plot_confusion_matrix(y_val, y_pred, normalize=True);
```

```
Classification Report for the Decision Tree Classifier:

              precision    recall  f1-score   support

           0     0.9998    0.9350    0.9663     60417
           1     0.0234    0.8952    0.0456       105

    accuracy                         0.9350     60522
   macro avg     0.5116    0.9151    0.5060     60522
weighted avg     0.9981    0.9350    0.9647     60522

ROC Curve:

 0.9151
```



[26]:
```
"""

Model validation:

We should check the metrics against data that the model has not seen before to␣
↪validate it.
We will do that using the test data.

Let's first normalize the variables Time and Amount at the test data:

"""

# normalizing test data

test_copy = test.copy()

std_scaler = StandardScaler()
test_copy["std_amount"] = std_scaler.fit_transform(test_copy.Amount.values.
↪reshape(-1,1))
test_copy["std_time"] = std_scaler.fit_transform(test_copy.Time.values.
↪reshape(-1,1))

test_copy.drop(["Amount", "Time"], axis=1, inplace=True)
```

```
test_copy.head()
```

[26]:

```
               V1        V2        V3        V4        V5        V6        V7  \
183484 -0.323334  1.057455 -0.048341 -0.607204  1.259821 -0.091761  1.159101
255448 -0.349718  0.932619  0.142992 -0.657071  1.169784 -0.733369  1.009985
244749 -1.614711 -2.406570  0.326194  0.665520  2.369268 -1.775367 -1.139049
63919  -2.477184  0.860613  1.441850  1.051019 -1.856621  2.078384  0.510828
11475   1.338831 -0.547264  0.737389 -0.212383 -1.110039 -0.525744 -0.801403

               V8        V9       V10  …       V22       V23       V24  \
183484 -0.124335 -0.174640 -1.644401  … -0.433890 -0.261613 -0.046651
255448 -0.071069 -0.302083 -1.192404  … -0.833209 -0.030360  0.490035
244749  0.329904  0.903813 -0.219013  …  1.134489  0.965054  0.640981
63919  -0.243399 -0.260691  0.133040  …  0.692245  0.150121 -0.260777
11475  -0.063672  0.997276  0.113386  … -0.074719  0.067055  0.333122

              V25       V26       V27       V28  Class  std_amount  std_time
183484  0.211512  0.008297  0.108494  0.161139      0   -0.177738  0.658372
255448 -0.404816  0.134350  0.076830  0.175562      0   -0.321945  1.320094
244749 -1.801998 -1.041114  0.286285  0.437322      0    0.034666  1.219742
63919   0.005183 -0.177847 -0.510060 -0.660533      0    0.838765 -0.919236
11475   0.379087 -0.268706 -0.002769  0.003272      0   -0.310490 -1.572827

[5 rows x 31 columns]
```

[27]:

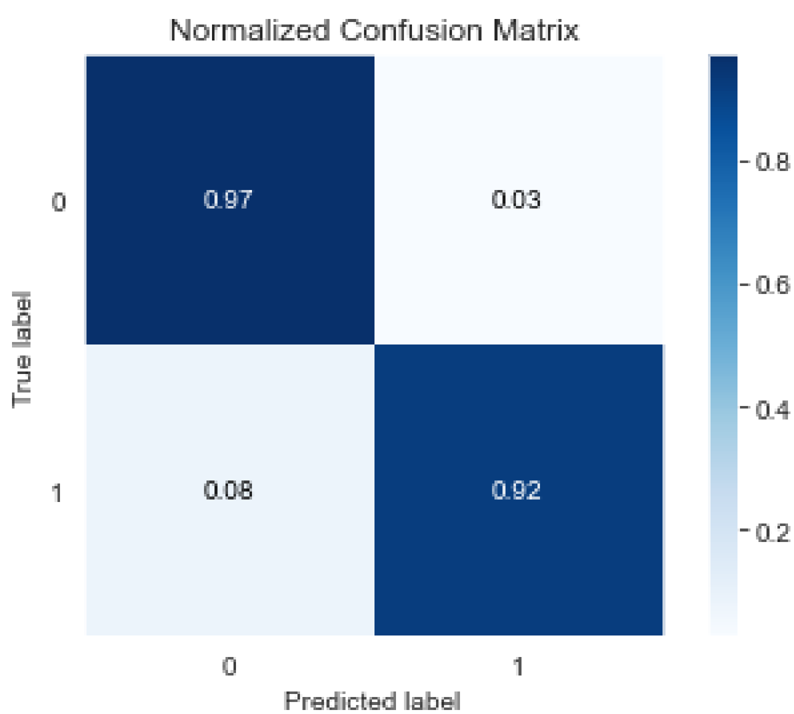
```
#Splitting the data in X and y
X_test = test_copy.drop(["Class"],axis=1)
y_test = test_copy["Class"]

#Prediciting test data for Logisitic Regression model
y_pred = lr_model.predict(X_test)
```

[28]:

```
"""

Finally, let's check the accuracy and confusion matrix for the logistic␣
↪regression model:

"""

#Checking the metrics for the first model
print("Classification Report for Logisitic Regression Model: \n\n",␣
↪classification_report(y_test, y_pred, digits=4))

#ROC
print("ROC Curve: \n\n", round(roc_auc_score(y_test, y_pred),4), "\n")
```

```
#plotting the confusion matrix
skplt.metrics.plot_confusion_matrix(y_test, y_pred, normalize=True);
```

```
Classification Report for Logisitic Regression Model:

              precision    recall  f1-score   support

           0     0.9999    0.9666    0.9830     42647
           1     0.0456    0.9189    0.0868        74

    accuracy                         0.9665     42721
   macro avg     0.5227    0.9428    0.5349     42721
weighted avg     0.9982    0.9665    0.9814     42721

ROC Curve:

 0.9428
```



[29]:
```
#prediciting data for the Decision Tree Model
y_pred = dt_model.predict(X_test)
```

[30]:
```
#Checking the metrics for the second model
```

```python
print("Classification Report for the Decision Tree Classifier: \n\n",␣
↪classification_report(y_test, y_pred, digits=4))

#ROC
print("ROC Curve: \n\n", round(roc_auc_score(y_test, y_pred),4), "\n")

#plotting the confusion matrix
skplt.metrics.plot_confusion_matrix(y_test, y_pred, normalize=True);
```

```
Classification Report for the Decision Tree Classifier:

              precision    recall  f1-score   support

           0     0.9998    0.9365    0.9671     42647
           1     0.0238    0.8919    0.0464        74

    accuracy                         0.9365     42721
   macro avg     0.5118    0.9142    0.5068     42721
weighted avg     0.9981    0.9365    0.9655     42721

ROC Curve:

 0.9142
```