Supervised Machine Learning: Classification

Understanding Credit card fraud detection

Matthias Bonvin

November 2022

1 Introduction

For this exercise we will try to explore some data coming from banking transaction. The goal is to determine which are fraudulent and which are true. The economical incentives for such an algorithm are huge. One of the problem is that fraudulent transaction are more or less rare. According to https://legaldictionary.net/credit-card-fraud, in 2012 credit or debit card fraud gross losses amount to 5.22 cents per 100 dollars. Also https://www.nerdwallet.com/article/credit-cards/credit-card-theft-fraud-serious-crime-penalty in a year approximately 3.5% of user have credit fraud.

2 Data set description

The data set comes from https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud. The data set represents transaction by credit cards in September 2013 by EU cardholder. There are 284,807 transaction of which 492 are fraud. So the probability of a random transaction being a fraud is $0,17\%$.
The data set has 31 features. The feature "Time" represents the time passed since the first transaction in the data set. The feature "Amount" represents the sum of money that was contained in the transaction. The feature "Class" represents if a transaction is a fraud or not. If the transaction is a fraud, the class is 1, else it is 0. The 28 other features are for privacy reason confidential. The only thing we know is that they come from a PCA of some real financial data. We denote those features $V1, ..., V28$.
The goal of this work is to try to predict whether a transaction is a fraud or not. The more realistic goal will be to do better than assigning at random if a transaction is fraudulent or not using as a prior distribution, the one on the data set.

3 Data cleaning and feature engineering

First we take out the time variable as the start and end was arbitrary and therefore could induce a bias in the learning algorithm. Also some of the data are duplicates, so we delete those.

The main challenge with this machine learning classification is that the classes are extremely unbalanced. So the first question to ask is : Is that a property of the data or something that comes from a bad data collection?

I have decided that the data collection was well done and that the unbalanced classes are a defining property of our data set.

From this point there are two possible choices. Either we work with the original data set or we use techniques to over sample it. In this work, I have chosen to only used the original data set. This for two reasons. The first as this project has no practical implication and my computer is not very powerful, I prefer to work with a smallest data set to reduce the computation time. The second reason is more business oriented. Assuming a bank use the results of this report to decide whether a credit card transaction is a fraud or not. Then of course it is important for them to actually find the credit card fraud. I have done some rough numerical experimental and even if oversampling the data does actually provide a bit better proportion of rightly classified fraud than classify with the original data set, but it gives a huge proportion of miss classified none fraudulent transaction. This would mean for a bank a lot of work to reject some of those transactions, which would mean some expenses. The experiment about which I speak are available in the Jupyter Notebook coming with this report. To see an excellent way of how we can deal with imbalanced data the reader can look at https://www.kaggle.com/code/janiobachmann/credit-fraud-dealing-with-imbalanced-datasets/notebook.

This approach has an important problem. As there is a very low probability of a given transaction to be a fraud, most of the algorithms are not well designed to rightly classify such transaction. This is due to sparsity of the data in an high dimensional features space. So one thing we can try to do is to reduce the number of features to reduce the dimension of the features space. In general, this would make the data set less sparse and could lead to better predictions.

So we want to select some features among $V1, ..., V28$. To do that I have look at the pair plots of features against features and the distribution of each features, taking care of highlighting were the fraud are located 1 .

This analysis made me choose the following features to drop :

$$V13, V15, V19, V21, V22, V23, V24, V25, V26, V27, V28.$$

To assert visually how good this, I have done some PCA to reduce the dimension of the data, so it can be visualized. It gives the pictures 2. The fraudulent transactions are colorized in yellow. It seems visually clear that the data the yellow dots are more visible in 2b.

4 Classifiers

I will use the Python library sklearn. The classifiers used are SGDClassifier, DecisionTreeClassifier, neighbors, MLPClassifier and RandomForestClassifier.

The first implemented was SGDClassifier. It was done so as it is quick and due to its properties it should be pretty robust even if there are unbalanced classes as most of the data are not used to do the prediction. During the tunning it was clear that the three best option for the loss function are log, modified huber and perceptron and the penality to be the norm l1.

The second was DecisionTreeClassifier. This algorithm seems to be the best to deal with unbalanced classes, as it does not really care how much elements there are in each classes in order to classify them.

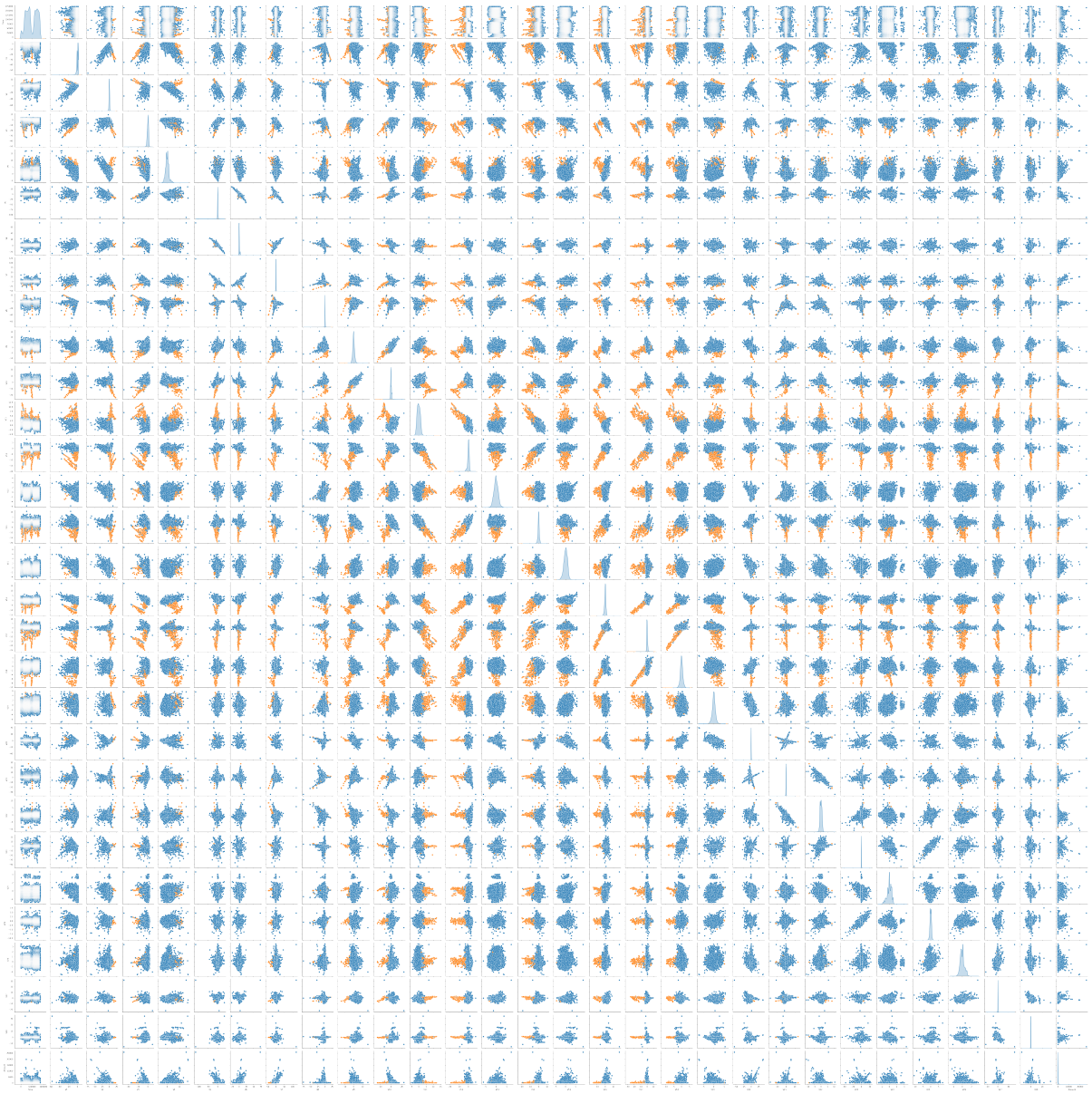
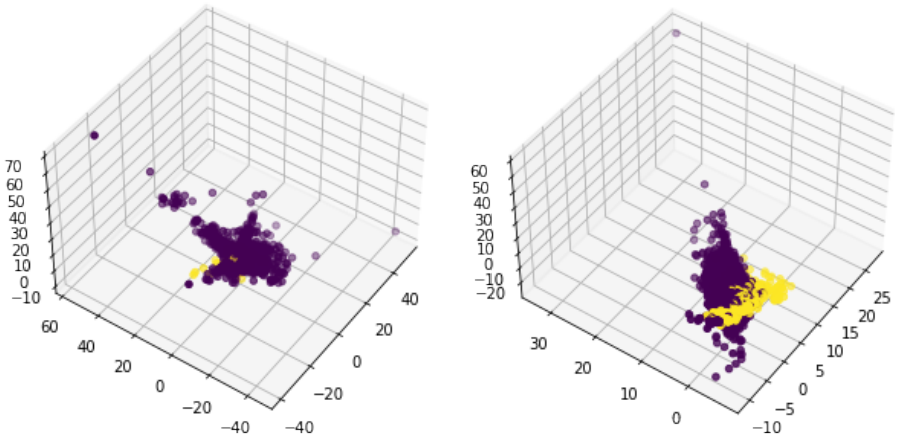


Figure 1: Pair plot comparison



(a) Visualized PCA for all features (b) Visualized PCA for selected features

Figure 2: PCA comparison

The third was neighbors. The idea is that according to the graphical analysis above, we have seen that the fraud tends to be in the same region. So this is a give away that kkn could be worth trying. The best number of neighbours is around 3 or 4.

The fourth was MLPClassifier. This is due to the fact that neural network are usually very efficient. I have not made it as efficient as possible as I didn't took the time to do it.

The fifth was RandomForestClassifier. This is following the same idea as DecisionTreeClassifier, but adding randomness can both make the computation faster and almost not change the tree, as the classes are so unbalance that it is unlikely to change much. This is in a way a escaping local optimum, faster version of DecisionTreeClassifier, with this data set.

I have used three different model selection metrics. There are the accuracy score, the confusion matrix and the true negative rate.

The accuracy score is not really suitied for this problem but it gives a good idea of how well the model is actually performing in general on the data.

The confusion matrix is a very useful tool to see how the algorithm classify each classes. This gives a good general view of how good the model is.

The true negative rate is the way to assert how better our model is than the random model.

5 Results

After training the models the results are the following using a mean of each metric over ten train test splits, with a test set of 15% of the size of the actual data set.

The following table 1 resume the accuracy score of each models. The SGDClassifier is taken with loss function log and the number of neighbors in knn to be 3.

We see that in term of accuracy our model are not so good, given that a random model is expected to have an accuracy score of 0.998333. So this is probably not a good way to model how good the model have learn from the data.

SGDClassifier	DecisionTreeClassifier	neighbors	MLPClassifier	RandomForestClassifier
0.99921050776569	0.9992903968608285	0.9994031814657299	0.999360887238892	0.9995958551657699

Table 1: Accuracy score comparison

SGDClassifier	DecisionTreeClassifier	neighbors	MLPClassifier	RandomForestClassifier
$\begin{bmatrix} 4.2e+04 & 19 \\ 15 & 48 \end{bmatrix}$	$\begin{bmatrix} 4.2e+04 & 16 \\ 14 & 48 \end{bmatrix}$	$\begin{bmatrix} 4.2e+04 & 2.6 \\ 23 & 39 \end{bmatrix}$	$\begin{bmatrix} 4.2e+04 & 12 \\ 15 & 47 \end{bmatrix}$	$\begin{bmatrix} 4.2e+04 & 2.4 \\ 15 & 47 \end{bmatrix}$

Table 2: Confusion matrix comparison

The following table 2 resumes the confusion matrix of each models. The SGDClassifier is taken with loss function log and the number of neighbors in knn to be 3.

What we can observe is that knn is very good to not miss-classified the non fraud but is not so good to classify the fraud. This may hint that we could use different models to make different predictions, as we could trust The following table 3 resumes the true negative rate of each models. The SGDClassifier is taken with loss function log and the number of neighbors in knn to be 3.

6 Conclusion

Given the above results, I would recommend to choose RandomForestClassifier as an algorithm.

The whole procedure is far from perfect, in particular it would be good to find a more appealing way to deal with imbalanced class. An idea is to find a better geometric way to understand the structure of our data set.

SGDClassifier	DecisionTreeClassifier	neighbors	MLPClassifier	RandomForestClassifier
0.7933333333333333	0.8033333333333333	0.6566666666666666	0.78	0.79

Table 3: True negative rate comparison

Untitled2

December 3, 2022

This Jupyter notebook is a notebook about fraude detection

The dataset is obtained from kaggle https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud

dataset = transaction by credit cards in September 2013 by eu cardholder There are 284,807 transaction of which 492 are fraude (Pr[fraude] = 0,0017)

Numerical output are result of a PCA transformation

We can not access the name of the feature due to confidentiality

We have 28 features V1, … V 28

Features 'Time' and 'Amount' are left unchanged by PCA

'Time' = seconds between each transactions and first transaction

'Amount' = transaction amount

'Class' = if fraud 1 else 0 which is the label

Some of the code was inspired by a post on kaggle https://www.kaggle.com/code/uvinir/credit-card-fraud-detection-decision-trees. To give the precise credit, it has informed me that there exists a way to deal with inbalanced data, my code to handle that is completely copied from the post (as I have choosen not to use it, it is commented). It has also made me attentive that there are some repetitive data in the dataset. Furthermore I have also reuse the idea to look at the confusion matrix, which was not part of the project before I had look at the post.

3,5% of user have credit fraud in 1 year source: https://www.nerdwallet.com/article/credit-cards/credit-card-theft-fraud-serious-crime-penalty

2012 credit/debit card fraud gross losses amount to 5.22 cents per 100 dollars https://legaldictionary.net/credit-card-fraud/

We begin with the import

[1]:
```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns

# To handle class inbalanced data
#from imblearn import under_sampling, over_sampling
```

```
#from imblearn.over_sampling import SMOTE
```

Then we import the data and define X and Y

We get ride of time for X as it only gives information about the time passed since the first transaction without any additional data. The start is therefore arbitrary and should not give us information about the fact that some transactions are fraude or not. This assumption can be certainly challenged but as we will see we will do troughout the project some effort to reduce the number of dimension of X, so we might begin with a simple suppression of arguably not very useful data.

[2]:
```
data = pd.read_csv ('C://Users//PC//Desktop//useful_stuff//fraude detection//
 ↪creditcard.csv')

data = data.drop_duplicates(keep='last')#as there are some duplicate in the␣
 ↪dataset

X = data.drop(['Class','Time'], axis = 1)
Y = data[['Class']]

#X_spld, Y_spld = SMOTE().fit_resample(X, Y) #use as the sampling is really poor
```

We look at some values of the data

[3]:
```
X.describe()
```

[3]:

	V1	V2	V3	V4	\
count	283726.000000	283726.000000	283726.000000	283726.000000	
mean	0.005917	-0.004135	0.001613	-0.002966	
std	1.948026	1.646703	1.508682	1.414184	
min	-56.407510	-72.715728	-48.325589	-5.683171	
25%	-0.915951	-0.600321	-0.889682	-0.850134	
50%	0.020384	0.063949	0.179963	-0.022248	
75%	1.316068	0.800283	1.026960	0.739647	
max	2.454930	22.057729	9.382558	16.875344	

	V5	V6	V7	V8	\
count	283726.000000	283726.000000	283726.000000	283726.000000	
mean	0.001828	-0.001139	0.001801	-0.000854	
std	1.377008	1.331931	1.227664	1.179054	
min	-113.743307	-26.160506	-43.557242	-73.216718	
25%	-0.689830	-0.769031	-0.552509	-0.208828	
50%	-0.053468	-0.275168	0.040859	0.021898	
75%	0.612218	0.396792	0.570474	0.325704	
max	34.801666	73.301626	120.589494	20.007208	

	V9	V10	…	V20	V21	\
count	283726.000000	283726.000000	…	283726.000000	283726.000000	
mean	-0.001596	-0.001441	…	0.000187	-0.000371	

```
std          1.095492      1.076407  …      0.769984      0.723909
min        -13.434066    -24.588262  …    -54.497720    -34.830382
25%         -0.644221     -0.535578  …     -0.211469     -0.228305
50%         -0.052596     -0.093237  …     -0.062353     -0.029441
75%          0.595977      0.453619  …      0.133207      0.186194
max         15.594995     23.745136  …     39.420904     27.202839

                 V22           V23           V24           V25  \
count  283726.000000  283726.000000  283726.000000  283726.000000
mean       -0.000015       0.000198       0.000214      -0.000232
std         0.724550       0.623702       0.605627       0.521220
min       -10.933144     -44.807735      -2.836627     -10.295397
25%        -0.542700      -0.161703      -0.354453      -0.317485
50%         0.006675      -0.011159       0.041016       0.016278
75%         0.528245       0.147748       0.439738       0.350667
max        10.503090      22.528412       4.584549       7.519589

                 V26           V27           V28         Amount
count  283726.000000  283726.000000  283726.000000  283726.000000
mean        0.000149       0.001763       0.000547      88.472687
std         0.482053       0.395744       0.328027     250.399437
min        -2.604551     -22.565679     -15.430084       0.000000
25%        -0.326763      -0.070641      -0.052818       5.600000
50%        -0.052172       0.001479       0.011288      22.000000
75%         0.240261       0.091208       0.078276      77.510000
max         3.517346      31.612198      33.847808   25691.160000

[8 rows x 29 columns]
```

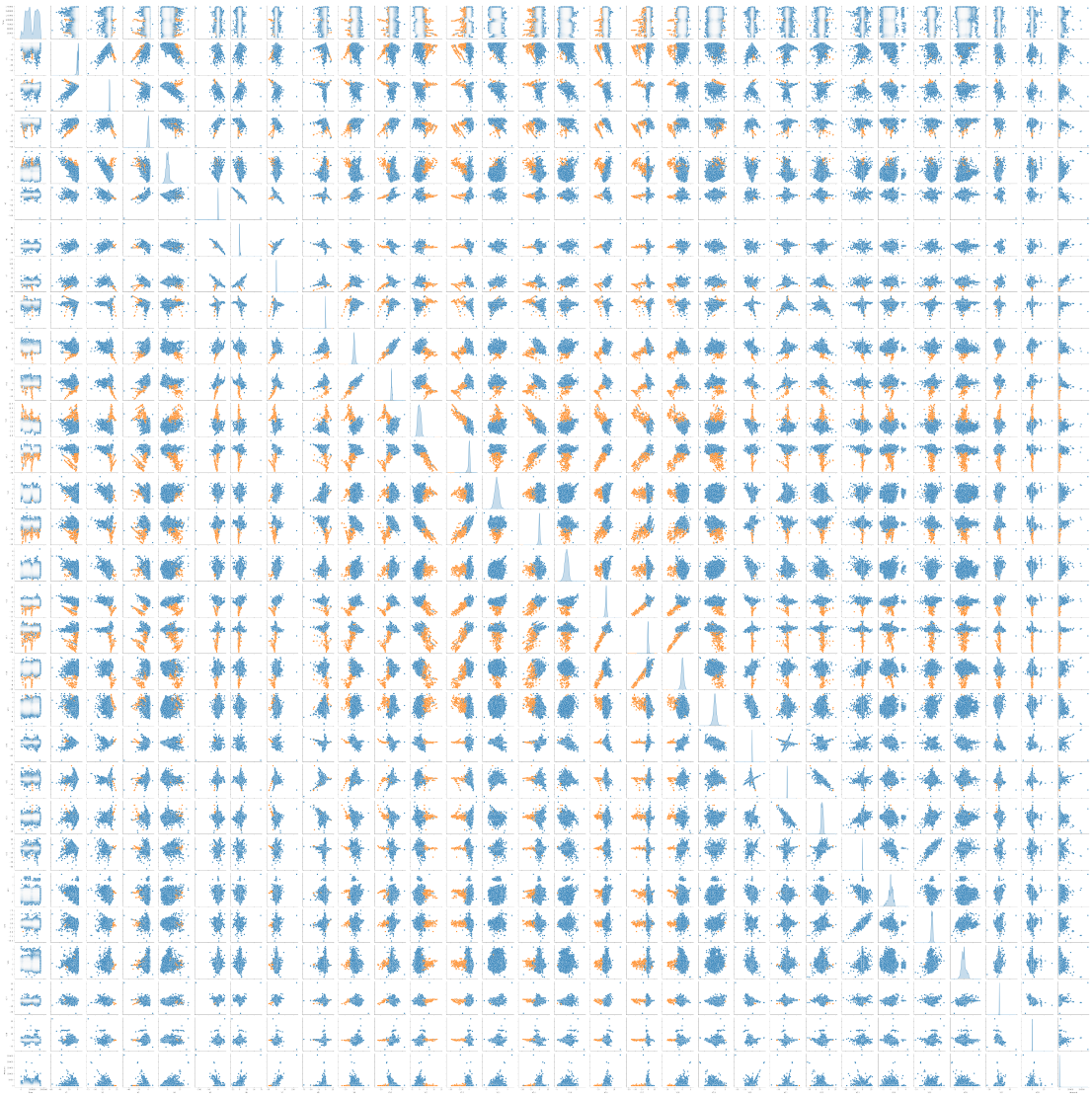
The percentage of fraudulant transactions

[4]:
```
1 - Y.mean()
```

[4]:
```
Class    0.998333
dtype: float64
```

[4]:
```
sns.pairplot(data=data, hue="Class")
```

[4]:
```
<seaborn.axisgrid.PairGrid at 0x1d788bbc280>
```



We do now our feature analysis, in order to take out some of the features.

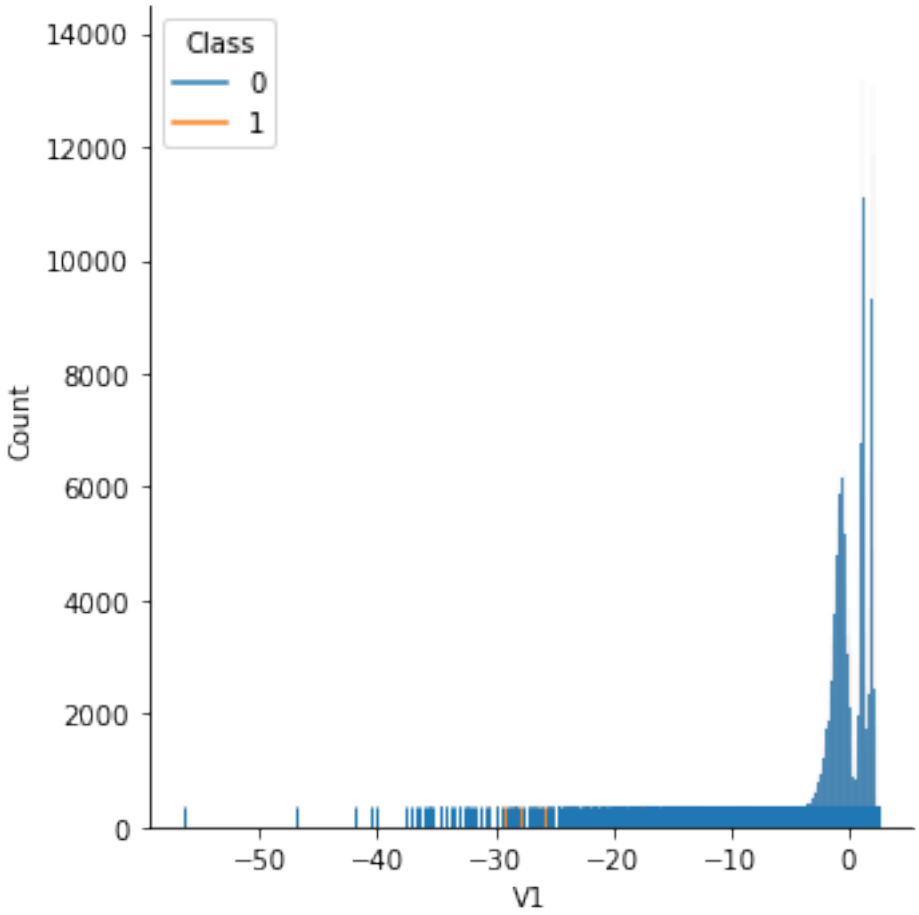
In the following we look at the empirical distribution of each features. This may bring us some indications.

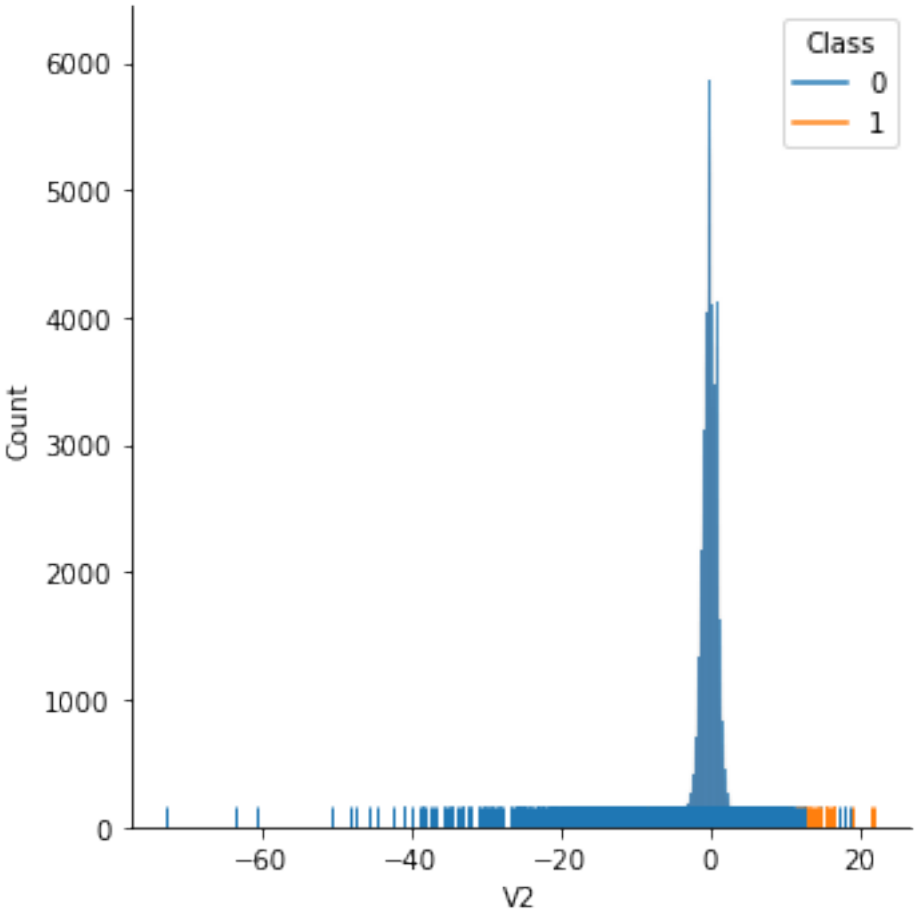
[9]:
```
for i in range(1,X.shape[1]): # empirical distribution of each features
    sns.displot(X['V{}'.format(i)])
    sns.rugplot(data = data, x = 'V{}'.format(i), hue = "Class")
sns.displot(X['Amount'],rug=True)
```

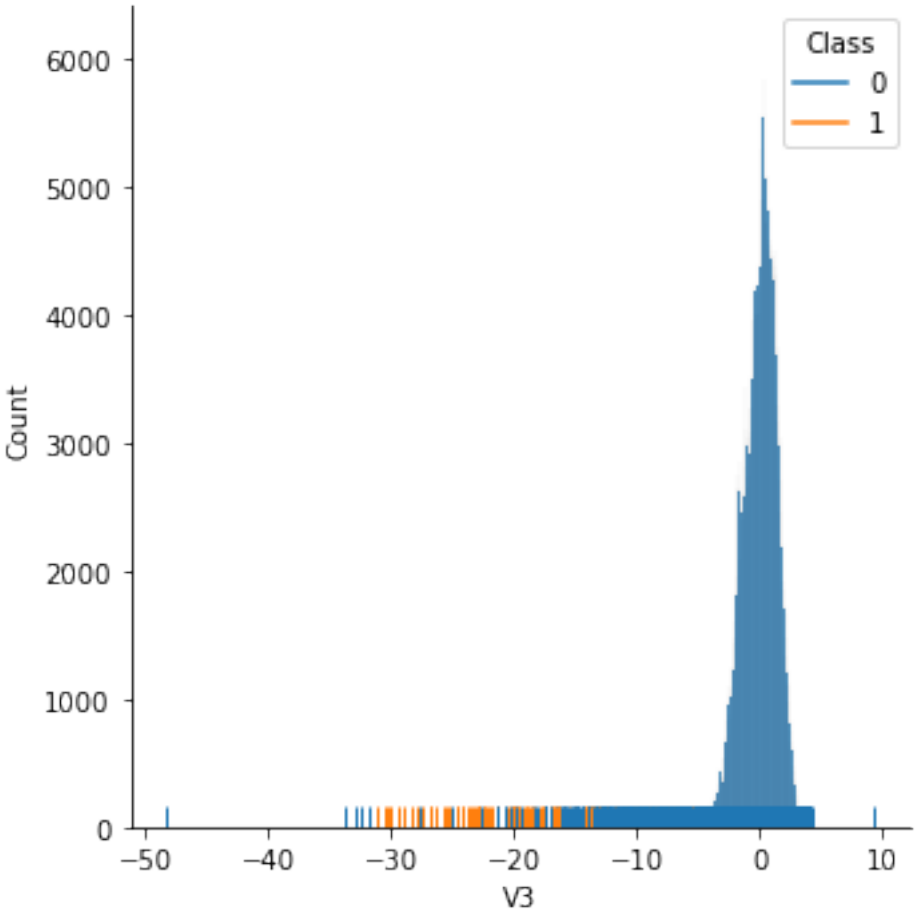
```
C:\Users\PC\anaconda3\lib\site-packages\seaborn\axisgrid.py:409: RuntimeWarning:
More than 20 figures have been opened. Figures created through the pyplot
interface (`matplotlib.pyplot.figure`) are retained until explicitly closed and
may consume too much memory. (To control this warning, see the rcParam
`figure.max_open_warning`).
```

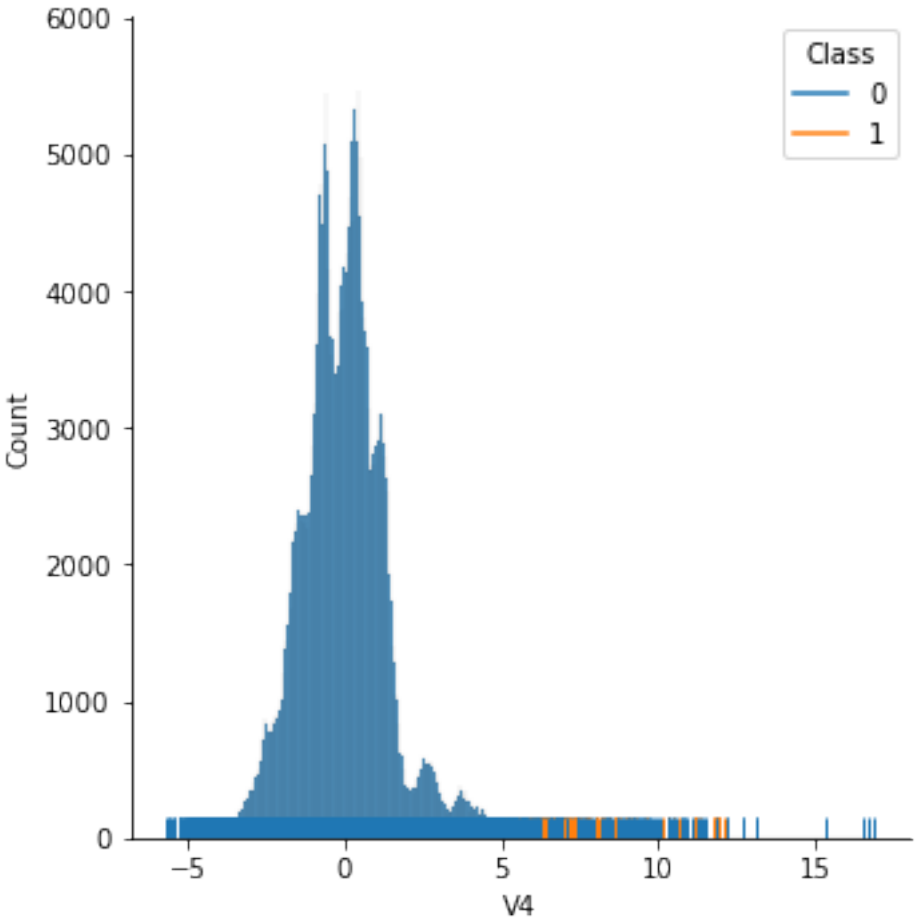
```
    fig = plt.figure(figsize=figsize)
```

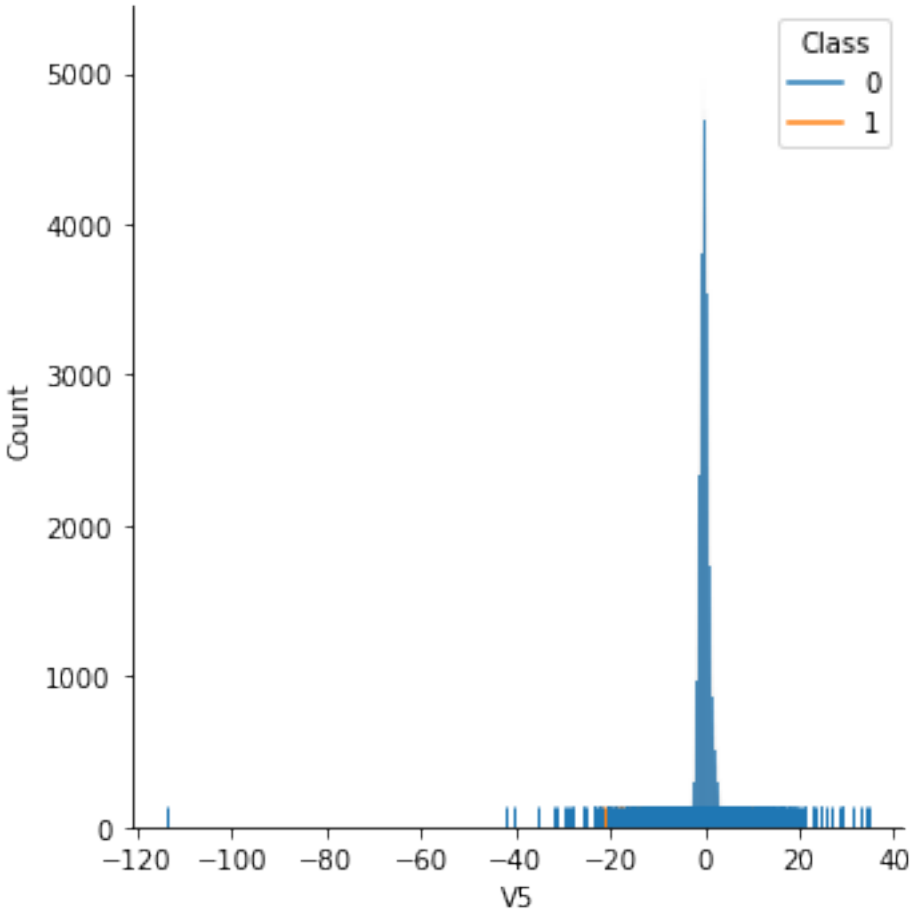
[9]: <seaborn.axisgrid.FacetGrid at 0x1dc33d88340>

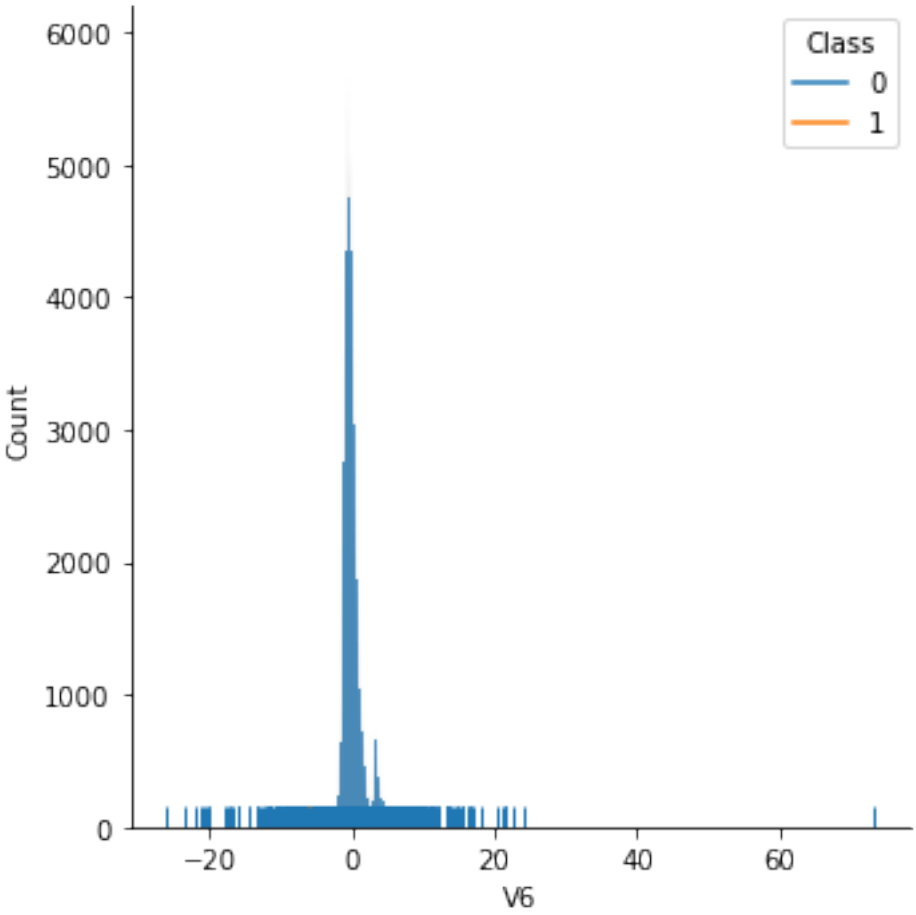


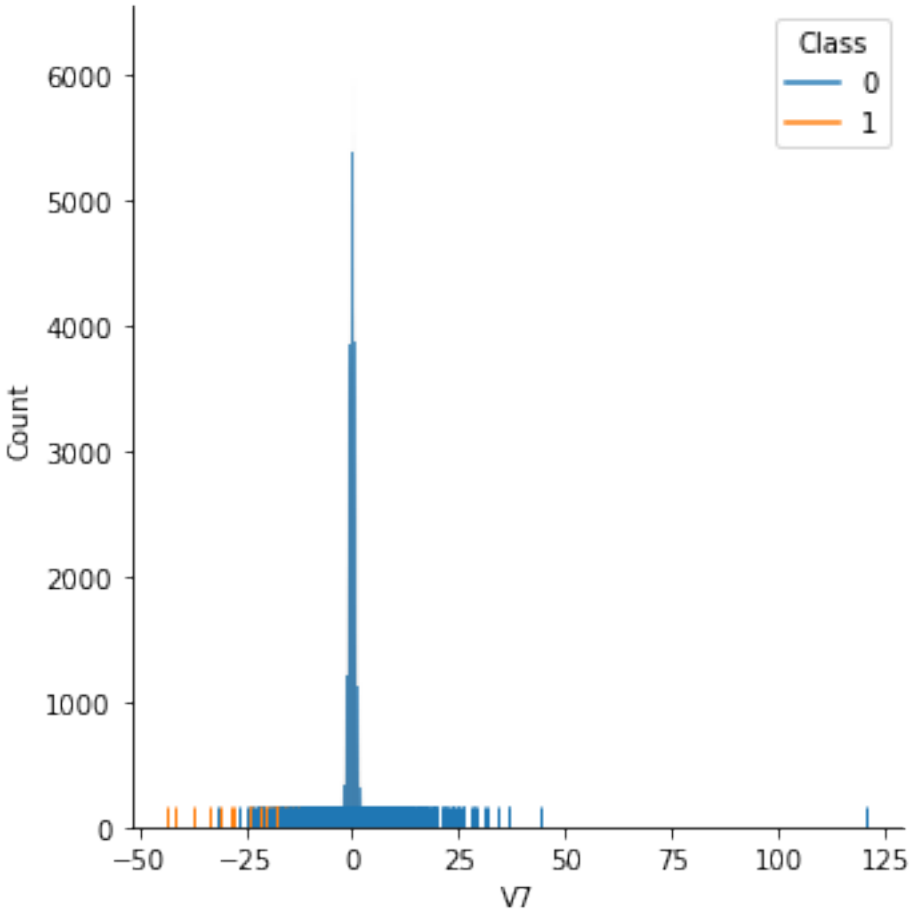

Class
0
1
6000
5000
4000
3000
2000
1000
0
Count
−60
−40
−20
0
20
V2

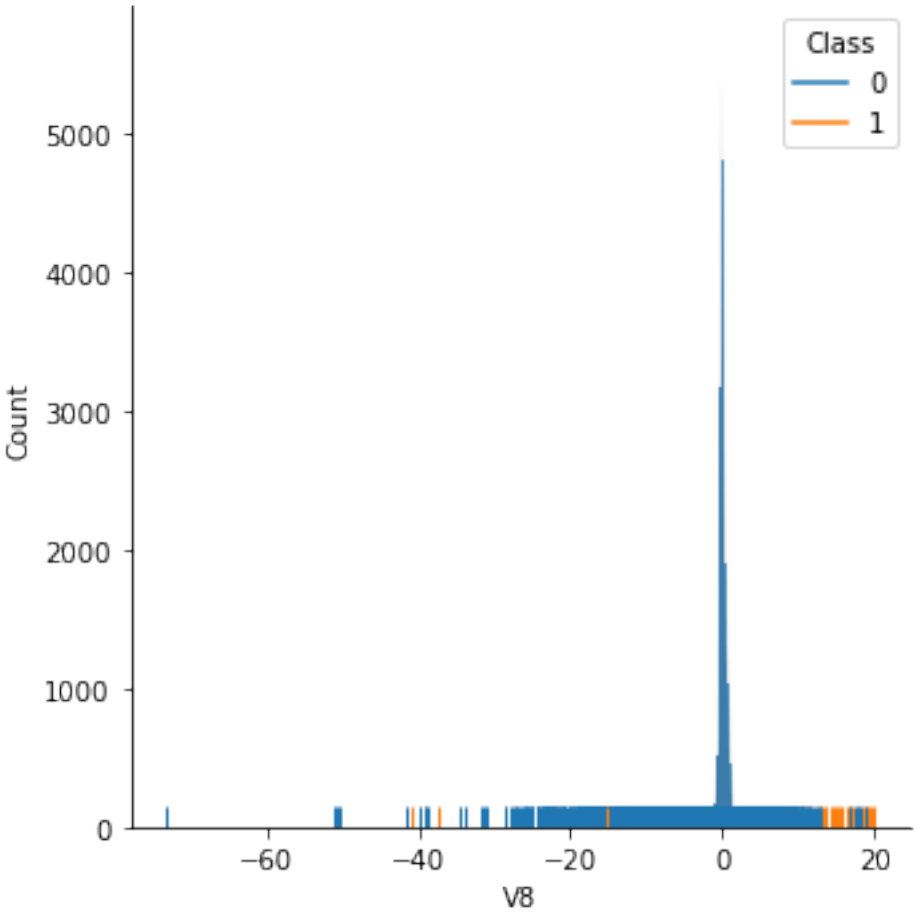

Class
0
1
6000
5000
4000
3000
2000
1000
0
Count
−50
−40
−30
−20
−10
0
10
V3

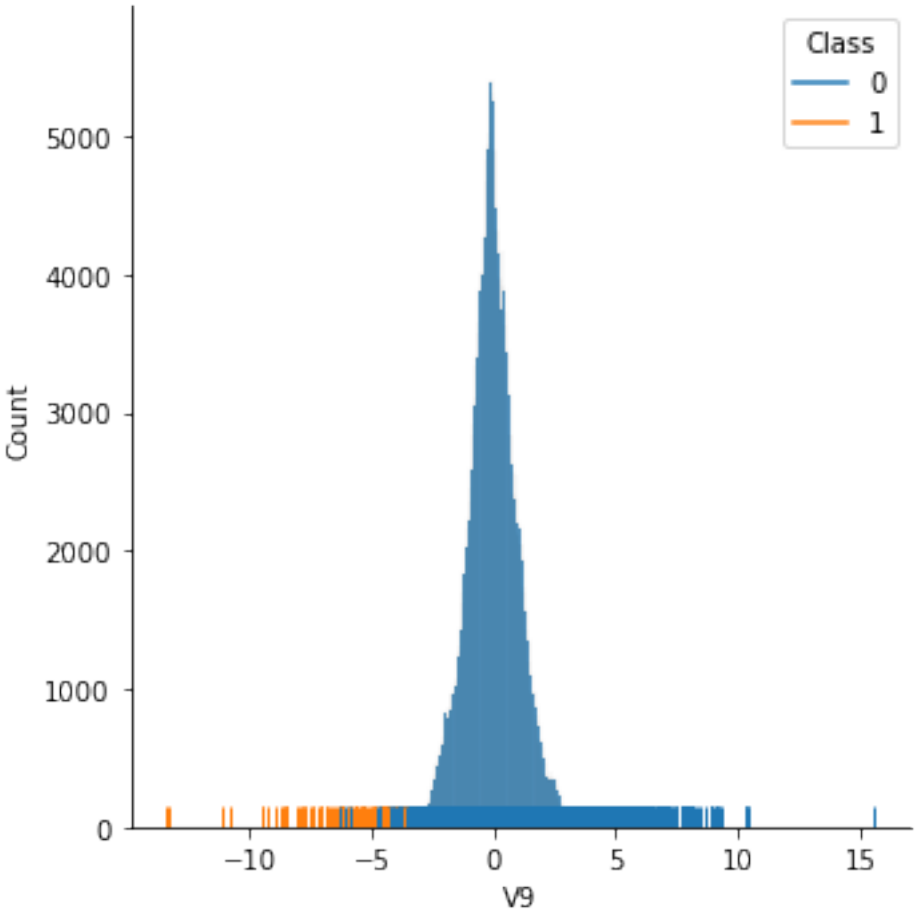

Class
0
1
6000
5000
4000
3000
2000
1000
0
Count
−5
0
5
10
15
V4

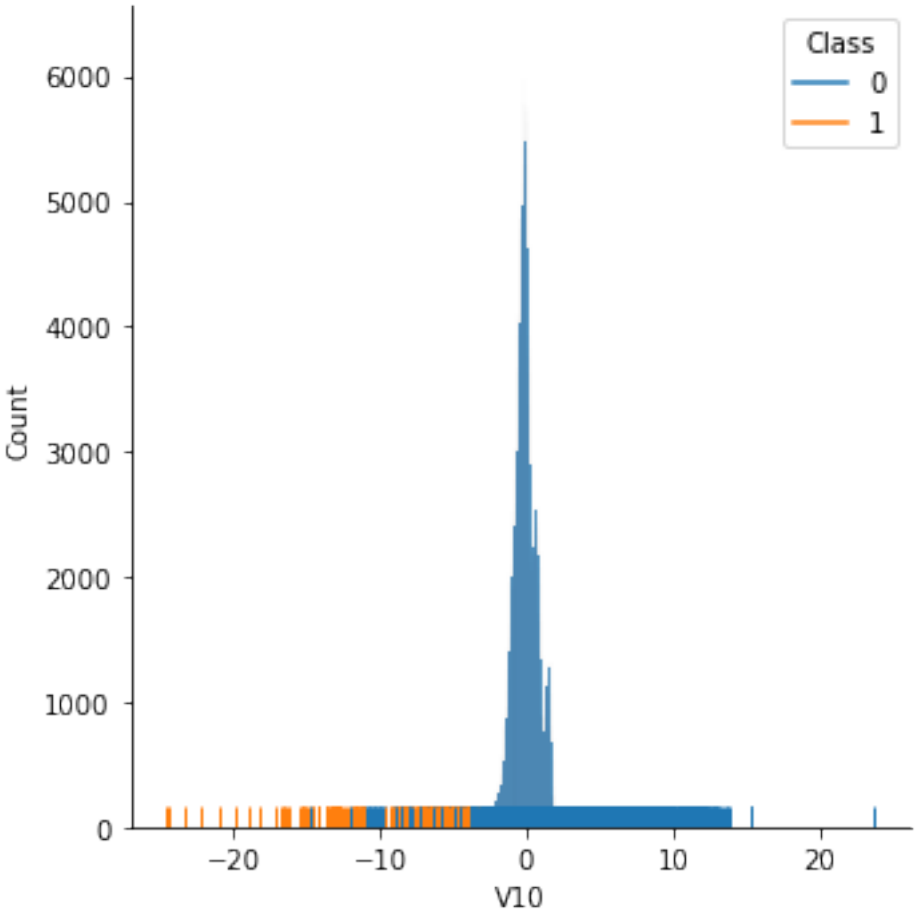

Class
0
1
5000
4000
3000
2000
1000
0
Count
−120
−100
−80
−60
−40
−20
0
20
40
V5

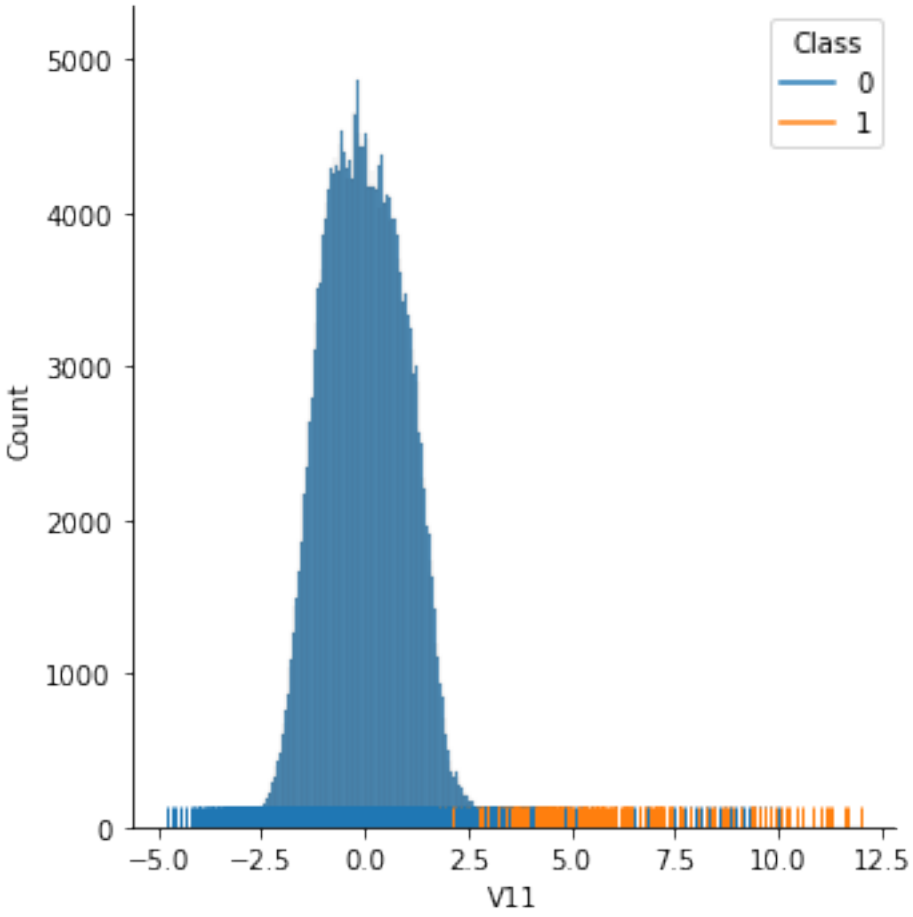

Class
0
1
6000
5000
4000
3000
2000
1000
0
Count
−20
0
20
40
60
V6

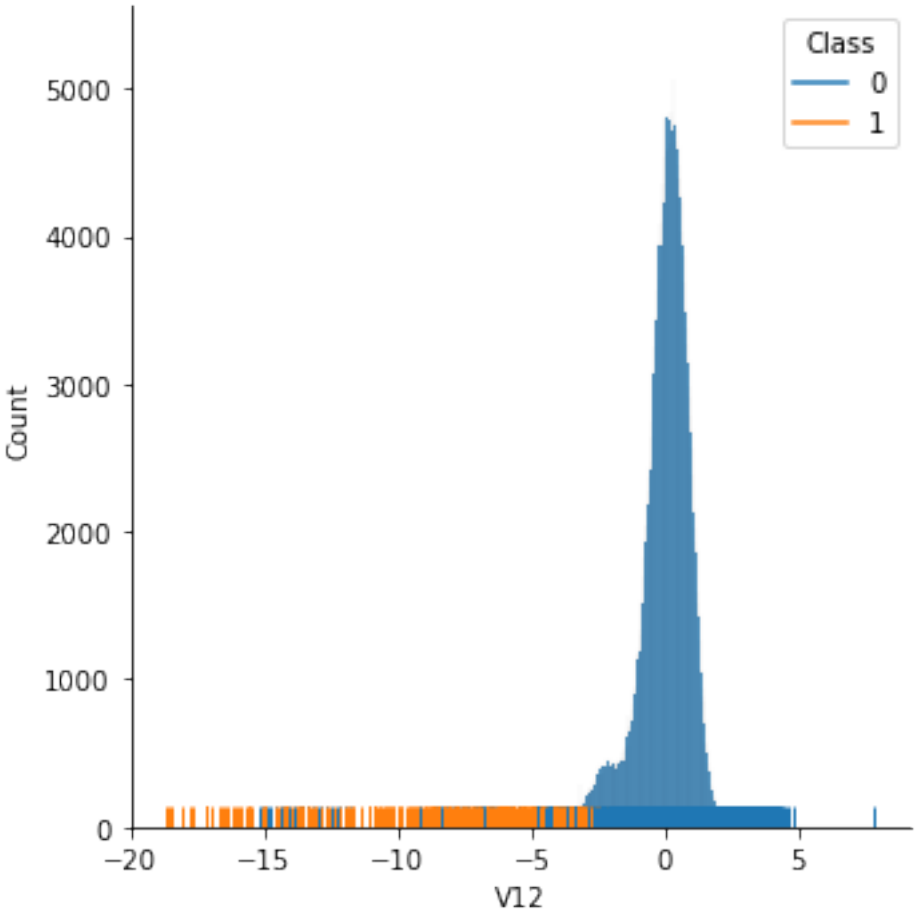

Class
0
1
6000
5000
4000
3000
2000
1000
0
Count
−50
−25
0
25
50
75
100
125
V7

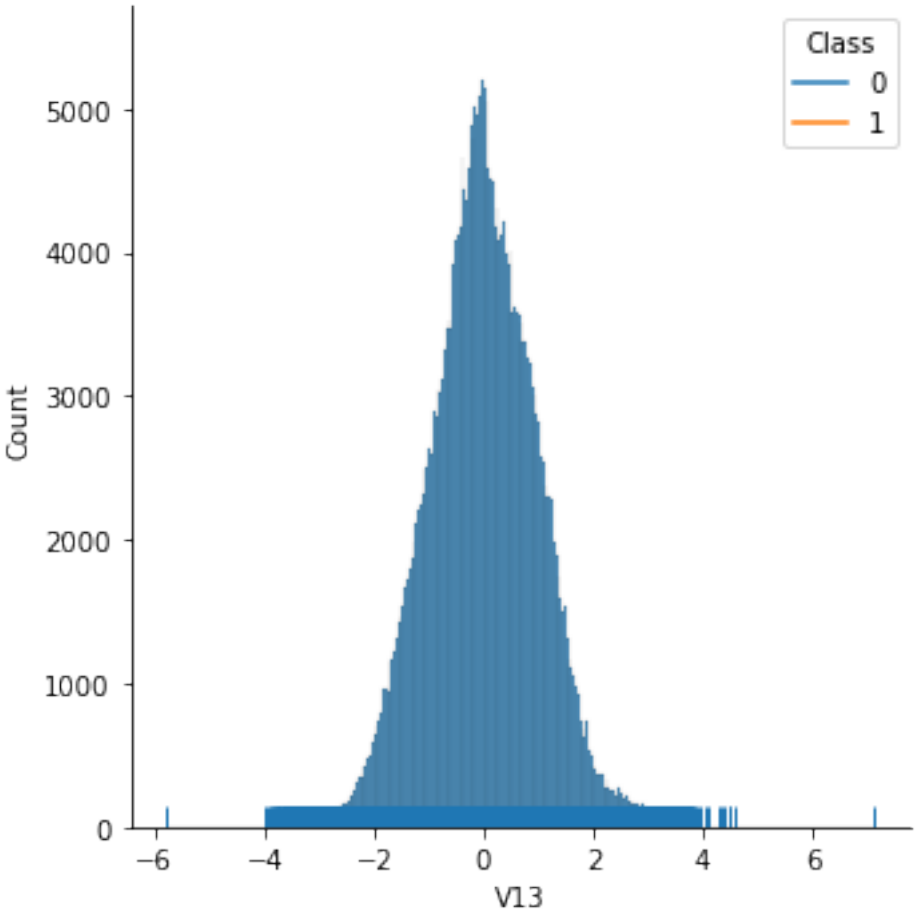

Class
0
1
5000
4000
3000
2000
1000
0
Count
−60
−40
−20
0
20
V8

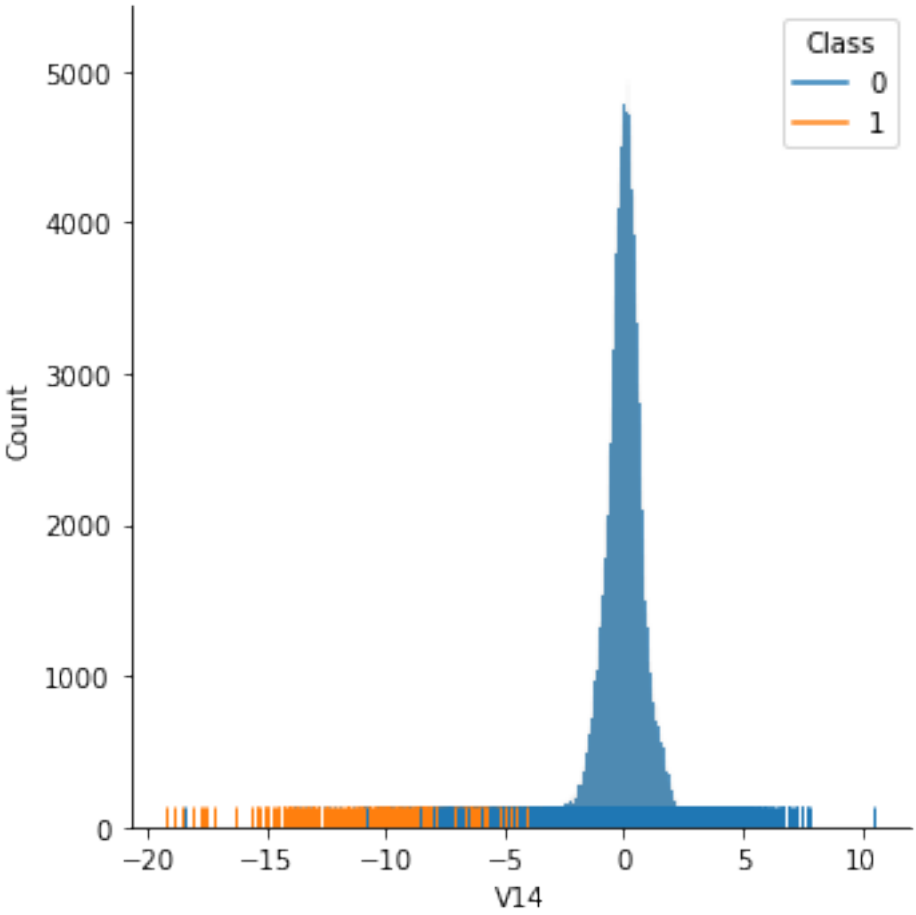

Class
0
1
5000
4000
3000
2000
1000
0
Count
−10
−5
0
5
10
15
V9

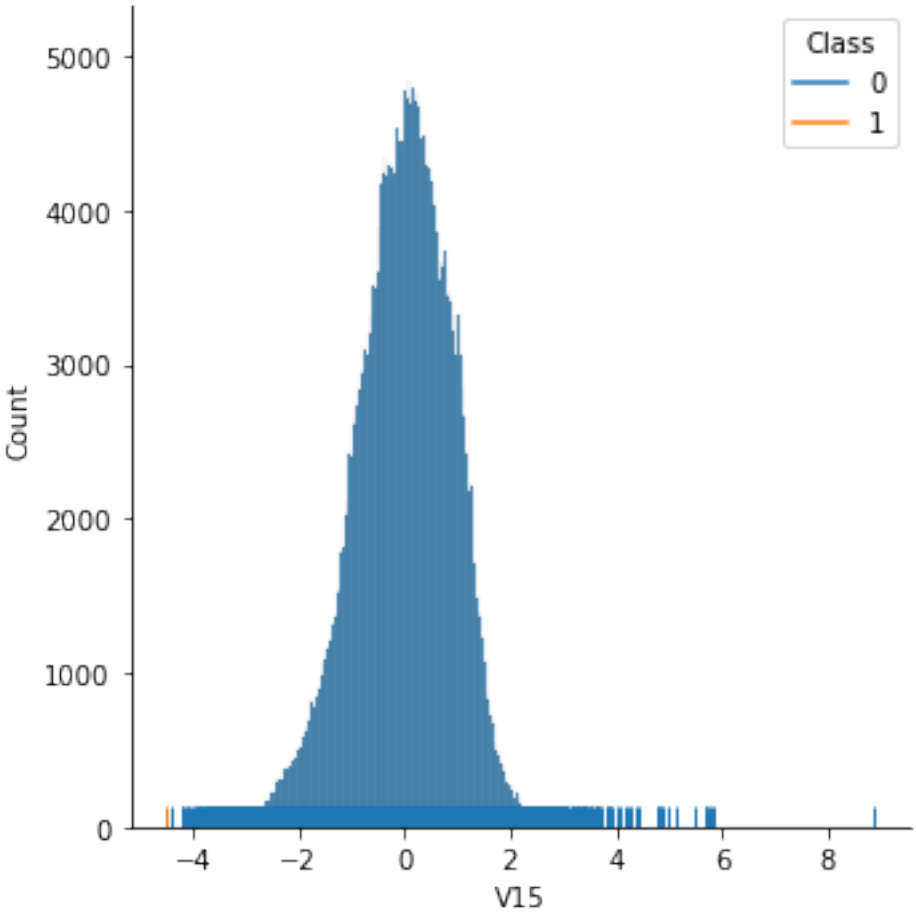

Class
0
1
6000
5000
4000
3000
2000
1000
0
Count
−20
−10
0
10
20
V10

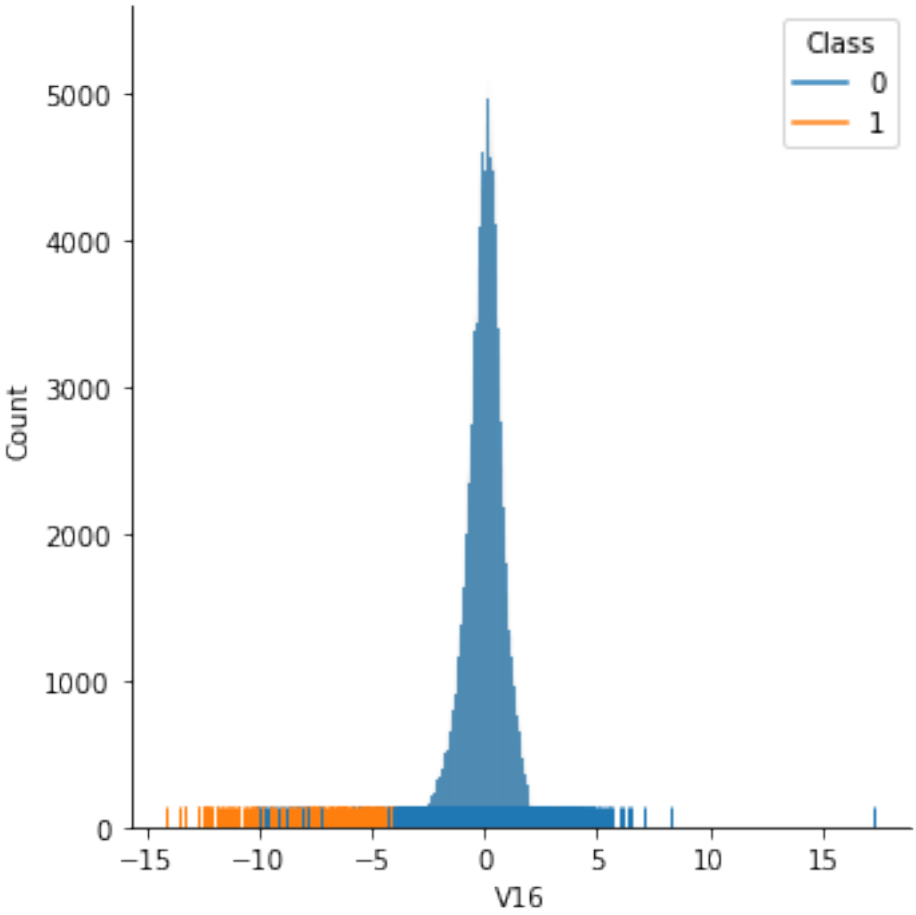

Class
0
1
5000
4000
3000
2000
1000
0
Count
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5
V11

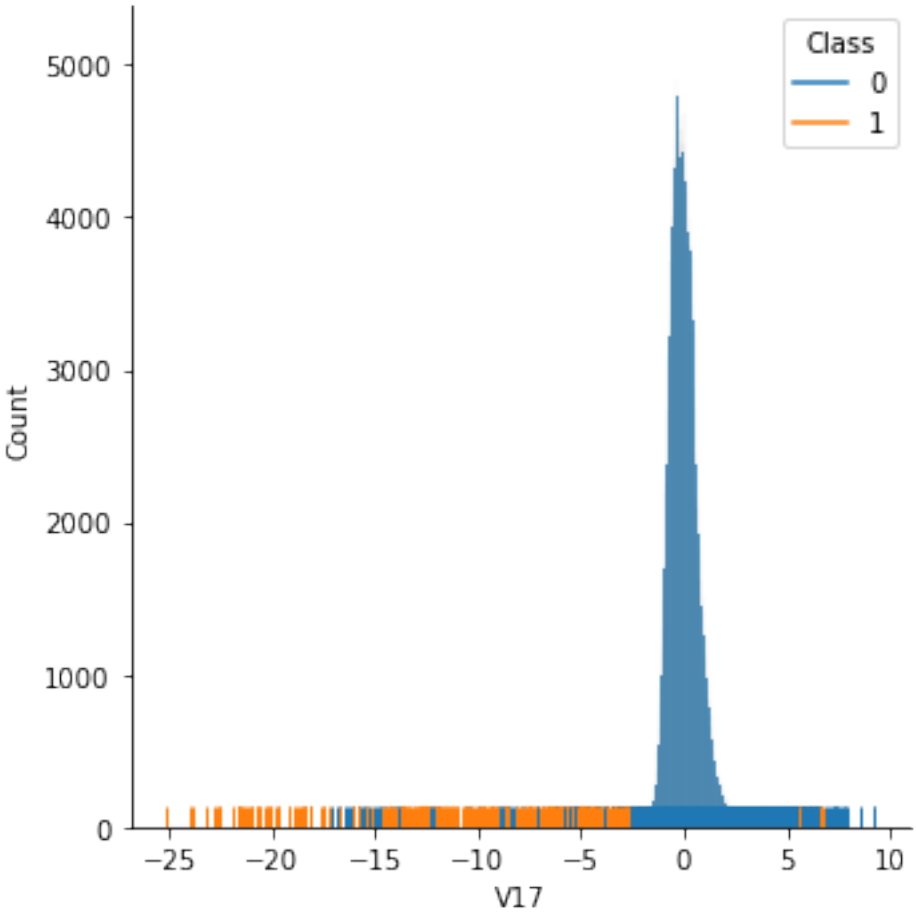

Class
0
1
5000
4000
3000
2000
1000
0
Count
−20
−15
−10
−5
0
5
V12

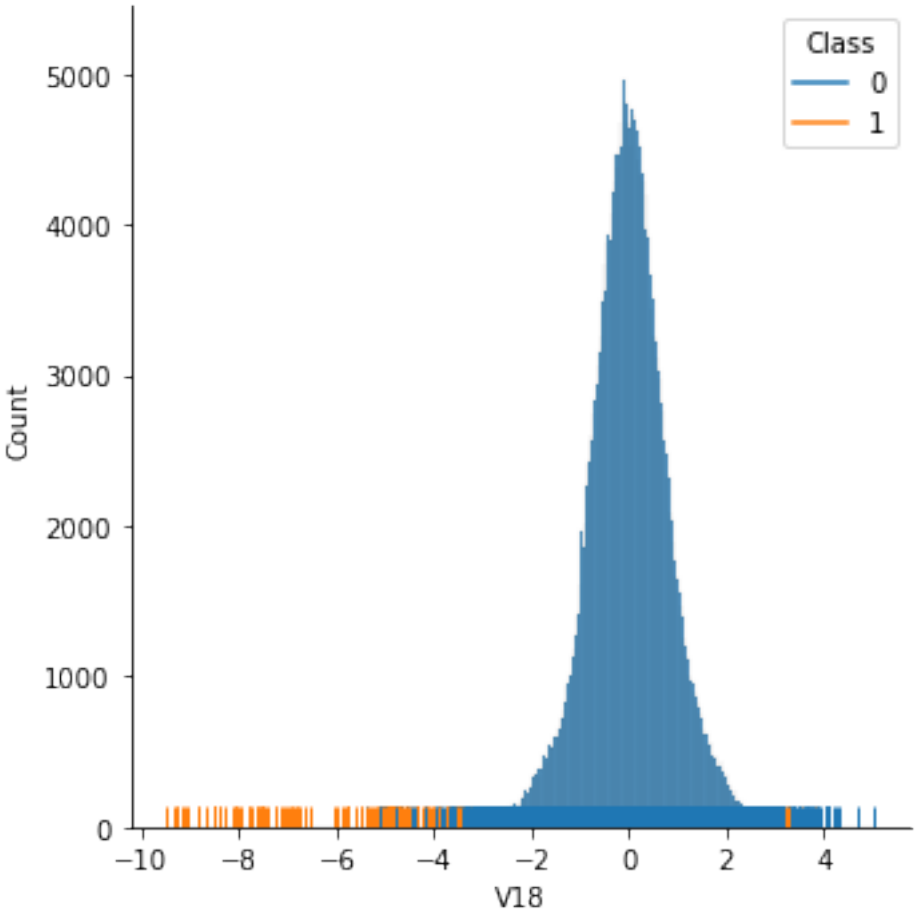

Class
0
1
5000
4000
3000
2000
1000
0
Count
−6
−4
−2
0
2
4
6
V13

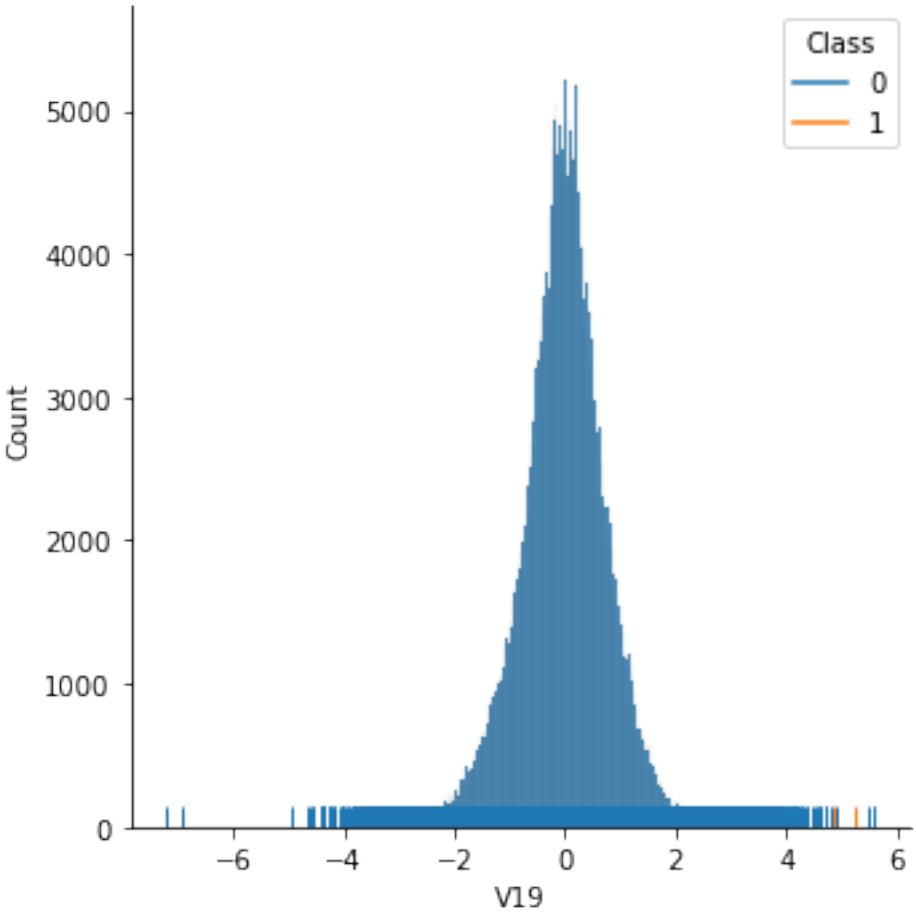

Class
0
1
5000
4000
3000
2000
1000
0
Count
−20
−15
−10
−5
0
5
10
V14

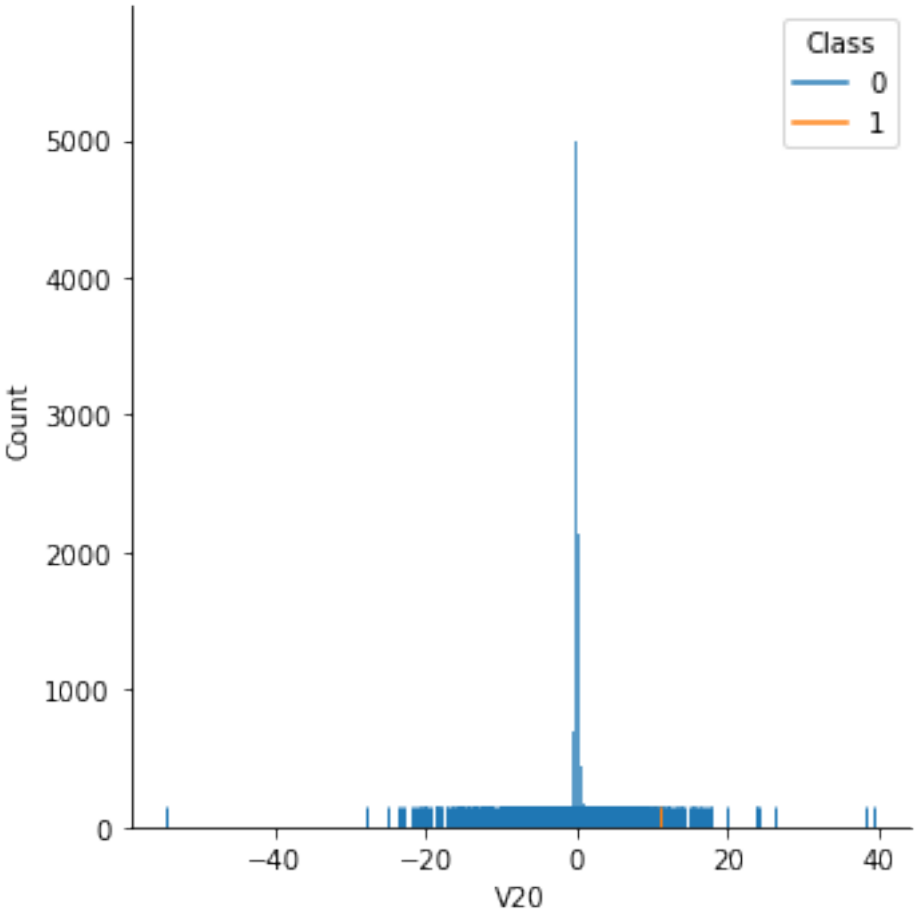

Class
0
1
5000
4000
3000
2000
1000
0
Count
−4
−2
0
2
4
6
8
V15

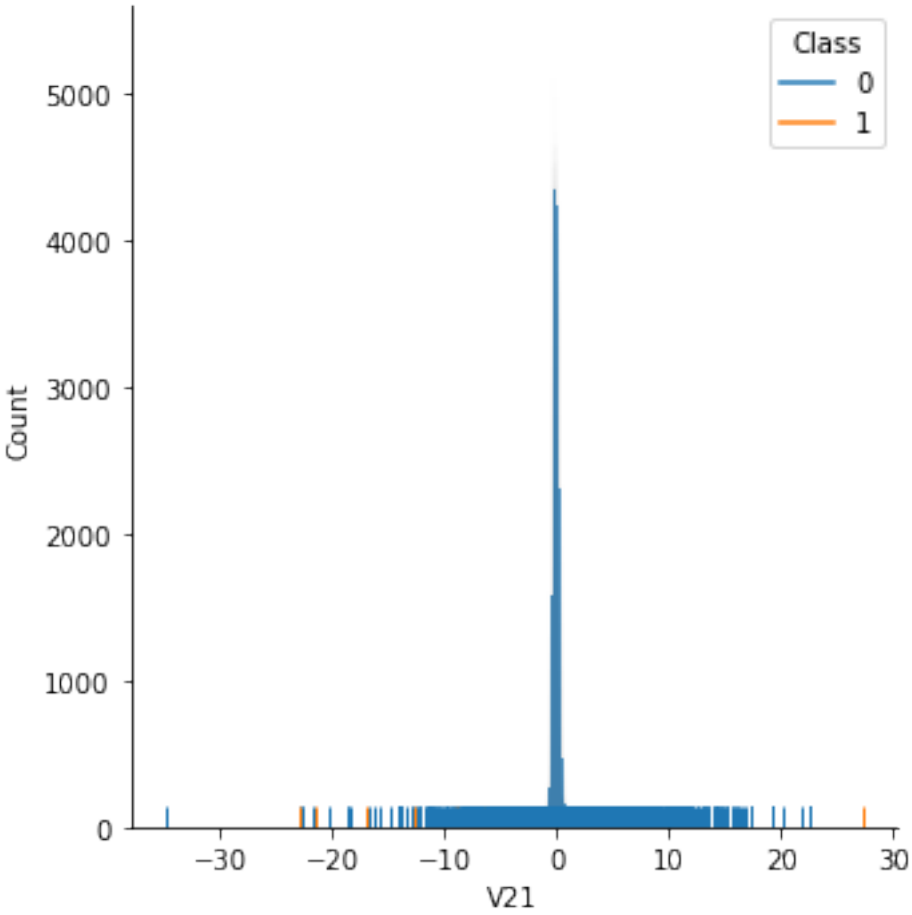

Class
0
1
5000
4000
3000
2000
1000
0
Count
−15
−10
−5
0
5
10
15
V16

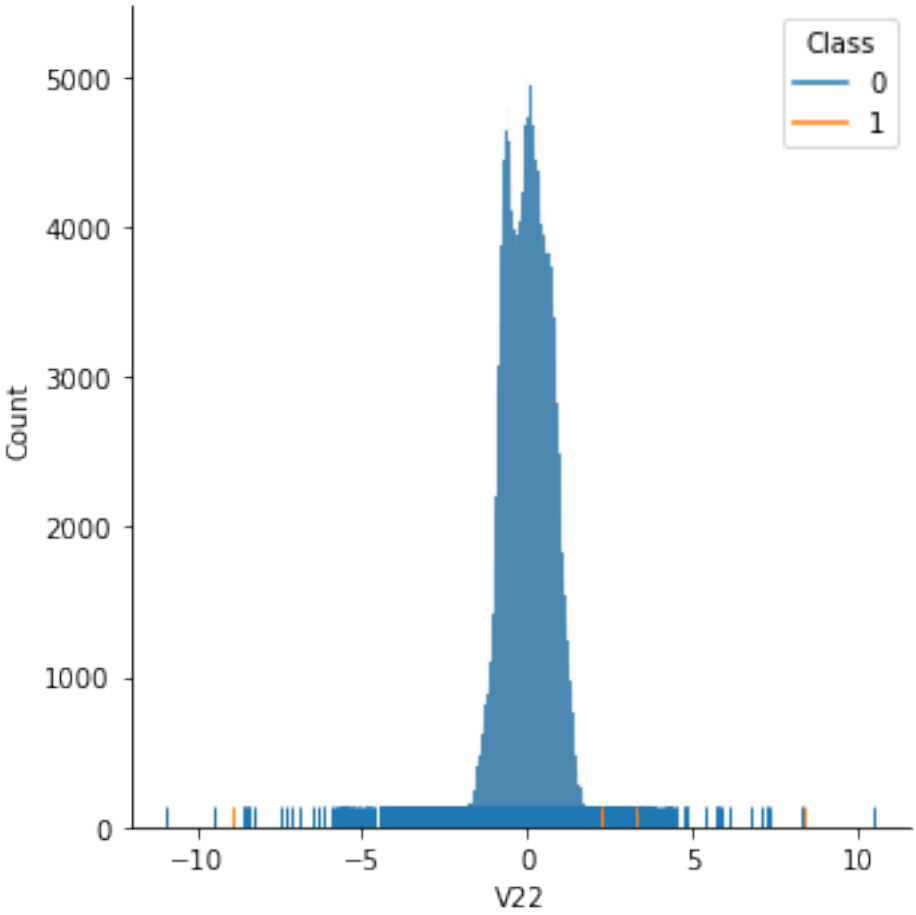

Class
0
1
5000
4000
3000
2000
1000
0
Count
−25
−20
−15
−10
−5
0
5
10
V17

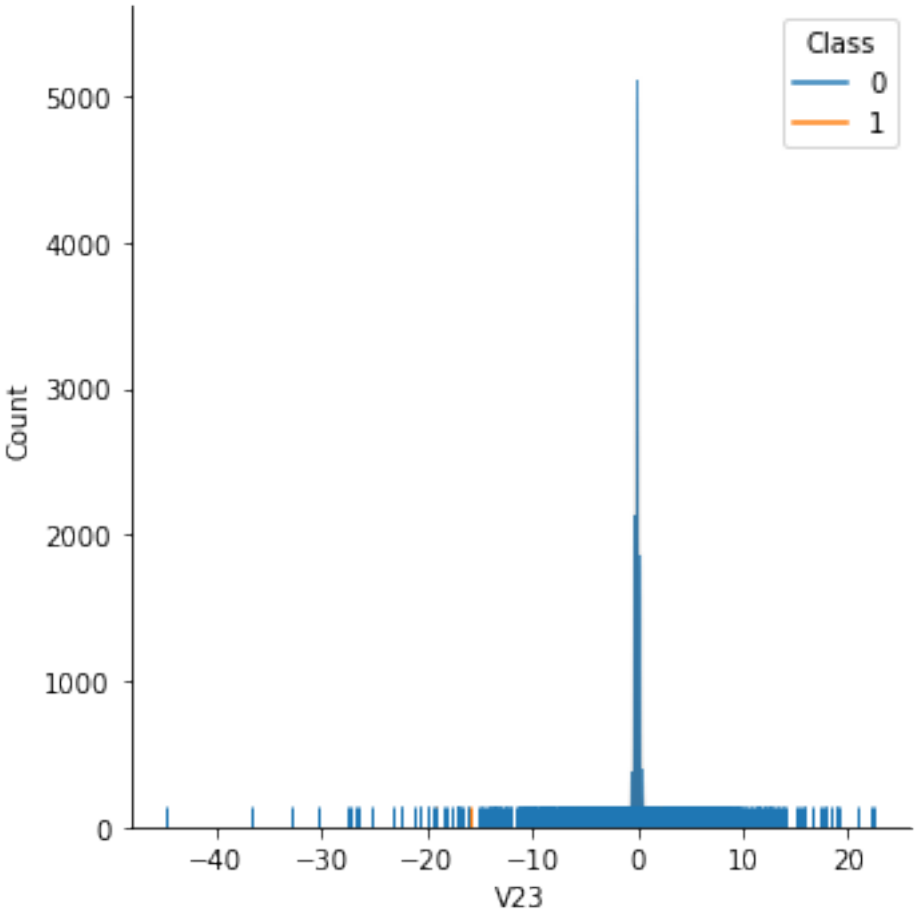

Class
0
1
5000
4000
3000
2000
1000
0
Count
−10
−8
−6
−4
−2
0
2
4
V18

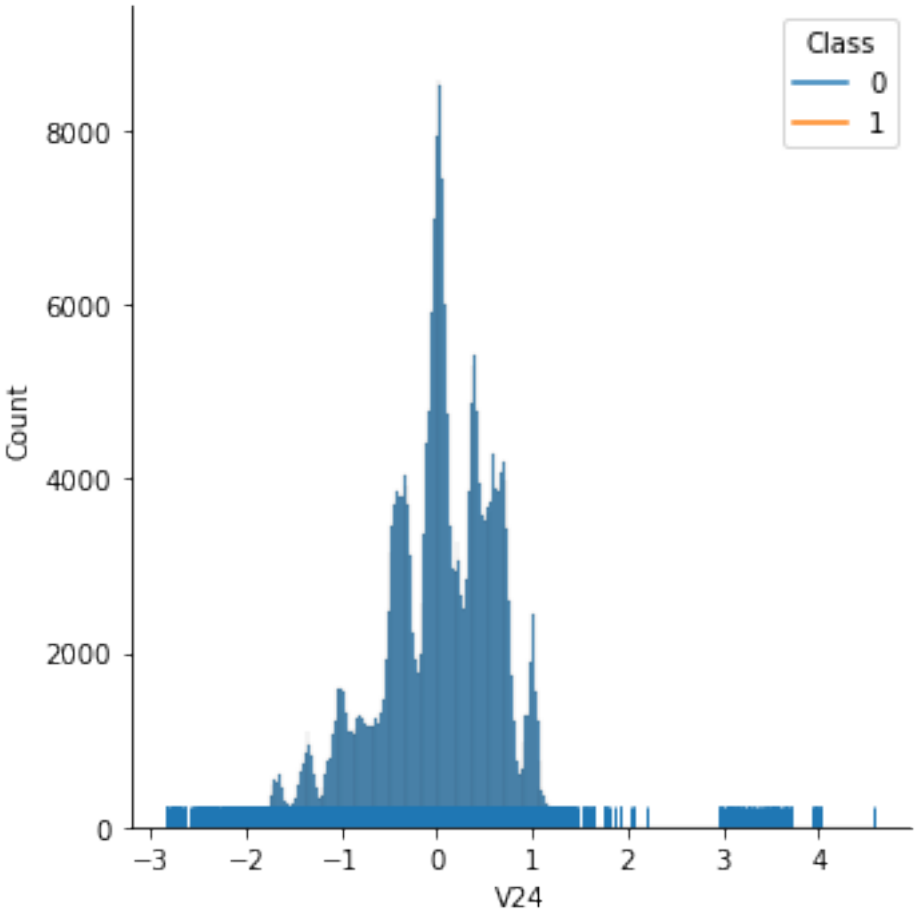

Class
0
1
5000
4000
3000
2000
1000
0
Count
−6
−4
−2
0
2
4
6
V19

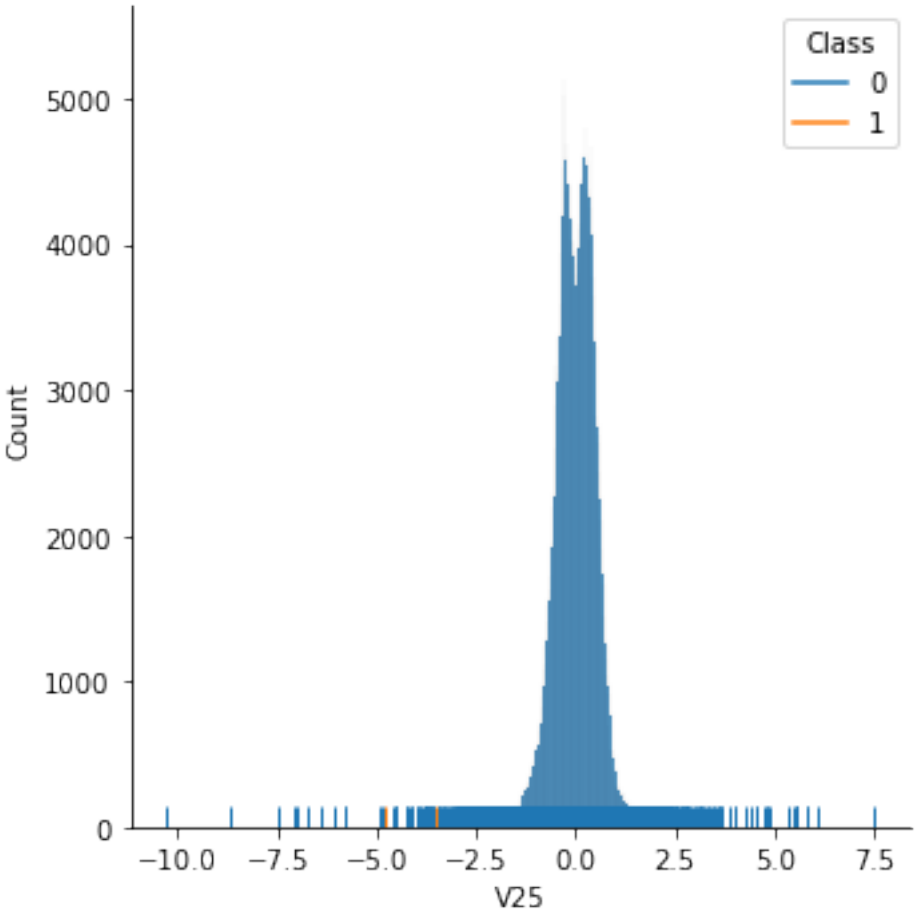

Class
0
1
5000
4000
3000
2000
1000
0
Count
−40
−20
0
20
40
V20

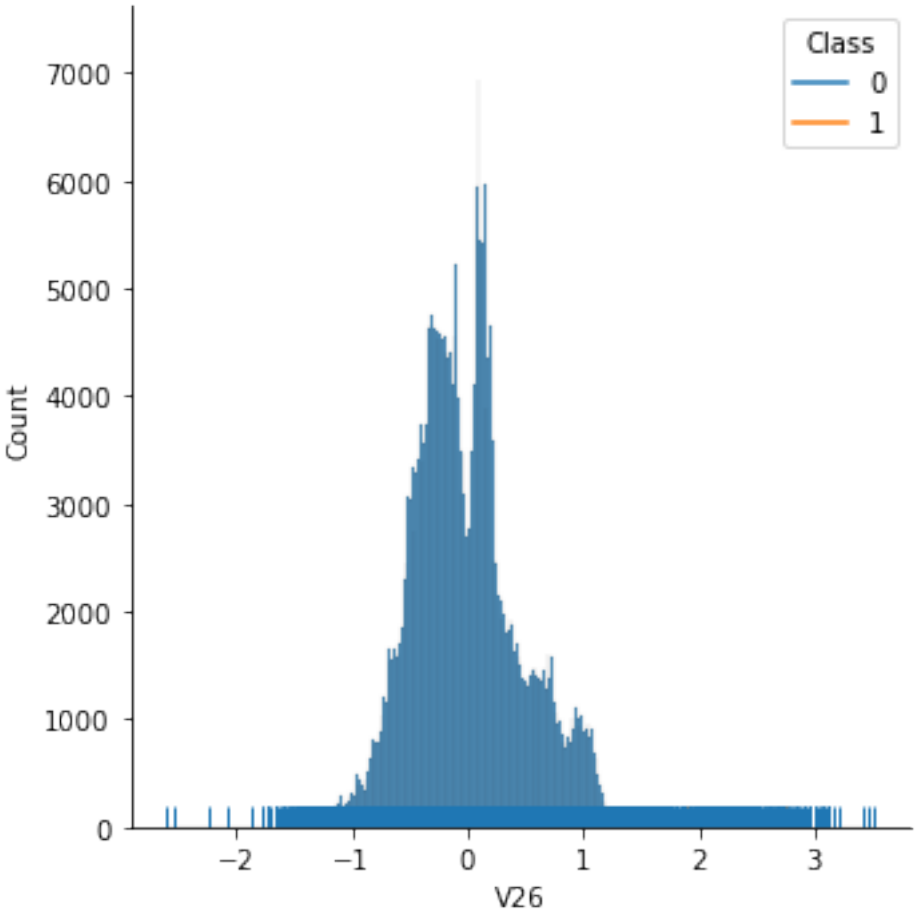

Class
0
1
5000
4000
3000
2000
1000
0
Count
−30
−20
−10
0
10
20
30
V21

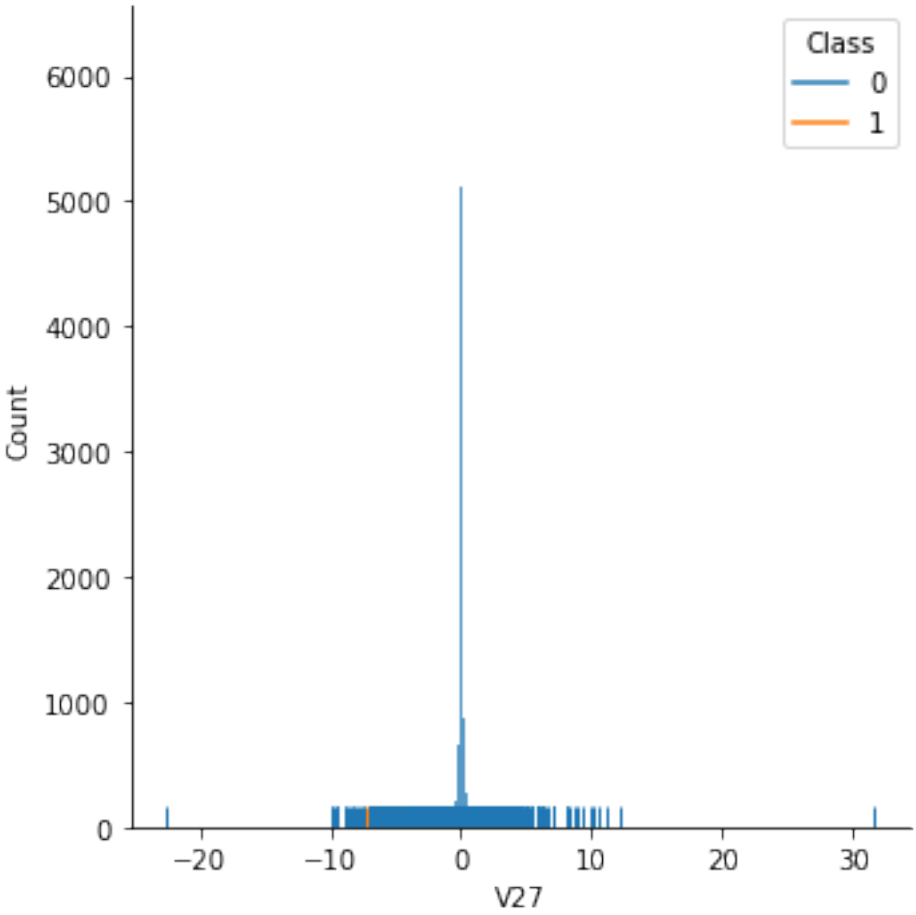

Class
0
1
5000
4000
3000
2000
1000
0
Count
−10
−5
0
5
10
V22

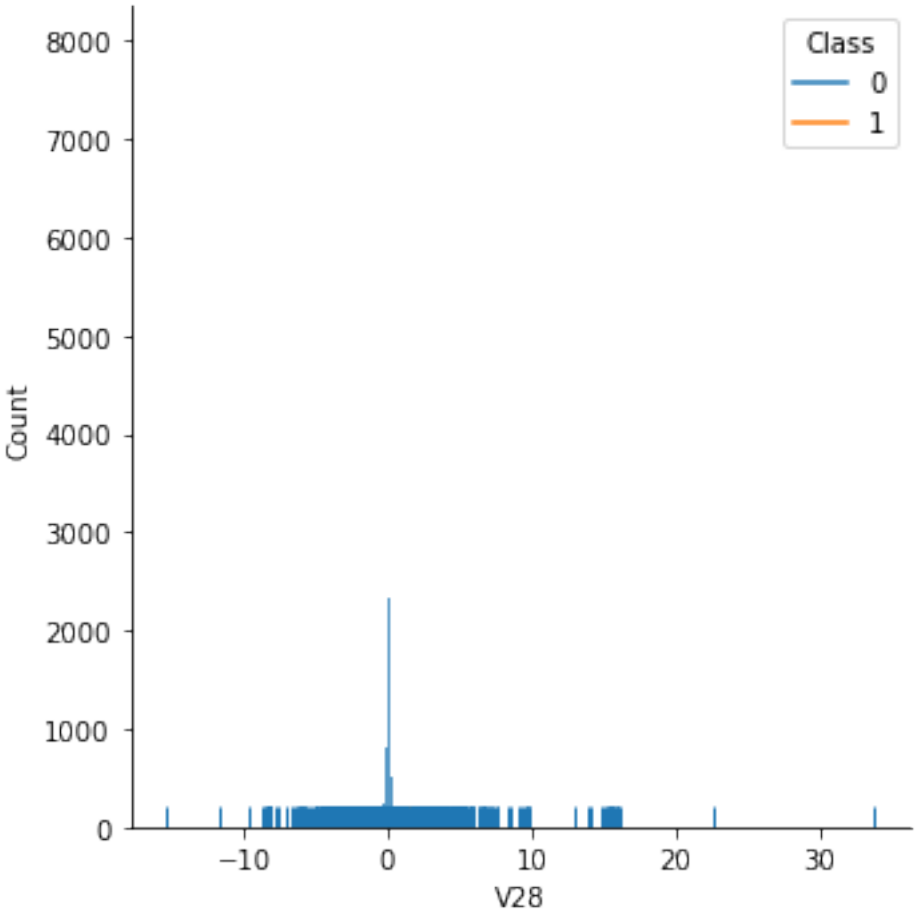

Class
0
1
5000
4000
3000
2000
1000
0
Count
−40
−30
−20
−10
0
10
20
V23

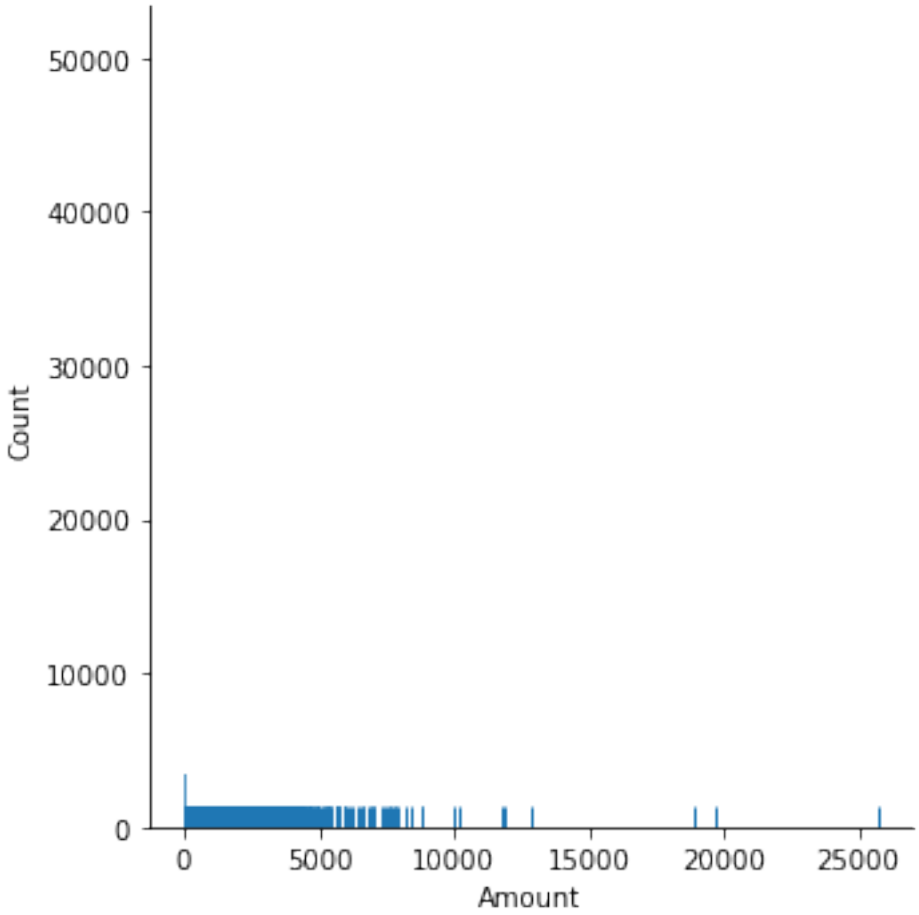

Class
0
1
8000
6000
4000
2000
0
Count
−3
−2
−1
0
1
2
3
4
V24


Class
0
1
5000
4000
3000
2000
1000
0
Count
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
V25


Class
0
1
7000
6000
5000
4000
3000
2000
1000
0
Count
−2
−1
0
1
2
3
V26


Class
0
1
6000
5000
4000
3000
2000
1000
0
Count
−20
−10
0
10
20
30
V27

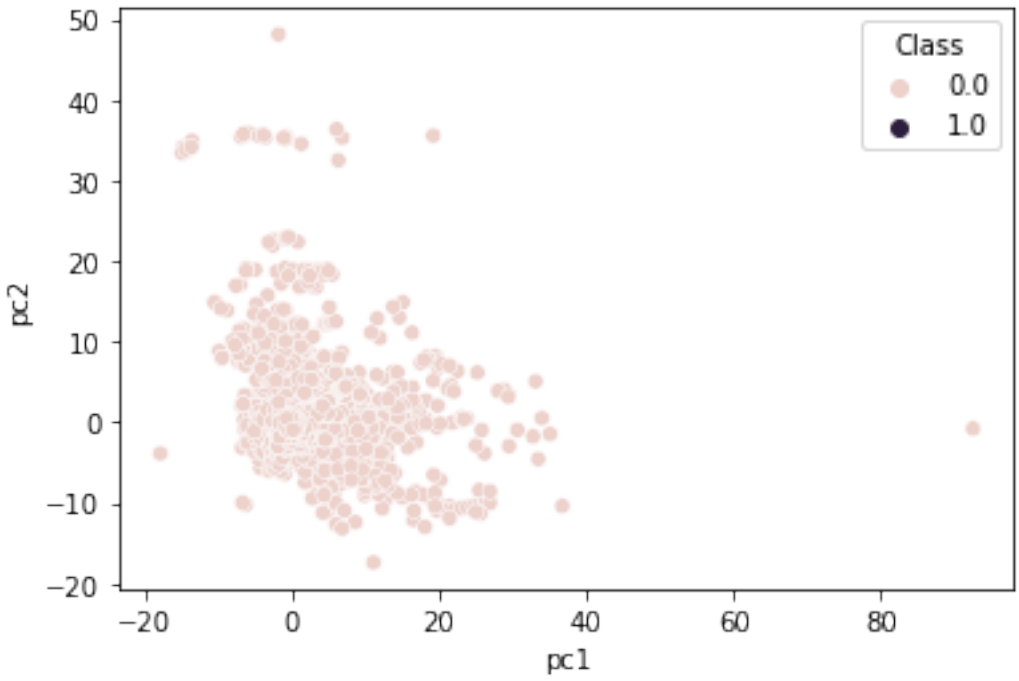

Class
0
1
8000
7000
6000
5000
4000
3000
2000
1000
0
Count
−10
0
10
20
30
V28

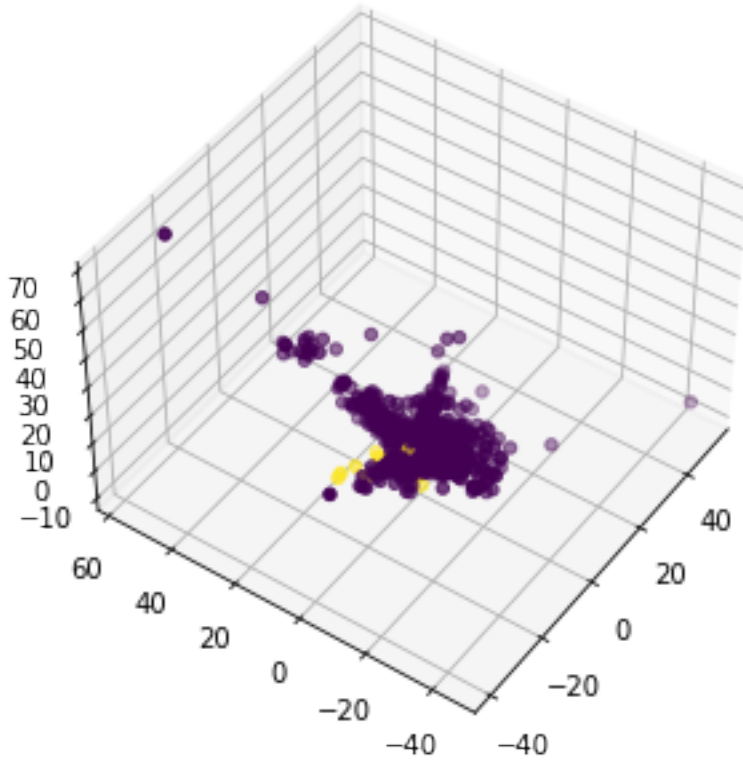


Then in an effort to visualise differently the data we do two PCA, one bringing the data in 2 dimension and the other one in 3 dimension.

[20]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

features = ['V{}'.format(i) for i in range(1, X.shape[1])]
x = X.loc[:, features].values
x = StandardScaler().fit_transform(x)

pca = PCA(n_components=3)
pCa = pca.fit_transform(x)
X_3d = pd.DataFrame(data = pCa, columns = ['pc1', 'pc2', 'pc3'])
data_3d = pd.concat ([X_3d,Y], axis=1)

pca = PCA(n_components=2)
pCa = pca.fit_transform(x)
X_2d = pd.DataFrame(data = pCa, columns = ['pc1', 'pc2'])
```

```
data_2d = pd.concat ([X_2d,Y], axis=1)


sns.scatterplot(x = 'pc1', y = 'pc2', data = data_2d, hue = "Class")


plt.figure(figsize=(6,5))
axes = plt.axes(projection='3d')
axes.scatter3D(X_3d['pc1'], X_3d['pc2'],X_3d['pc3'], c= Y)

axes.view_init(45, 215)
plt.show()
```





According to the above features analysis we define a new dataset.

[3]:
```
X_new = data.drop(['V13','V15','V19', 'V21', 'V22', 'V23', 'V24', 'V25', 'V26',␣
 ↪'V27', 'V28', 'Class','Time'], axis = 1)

#X_new_spld, Y_new_spld = SMOTE().fit_resample(X_new, Y)

X_new.describe()
```

[3]:

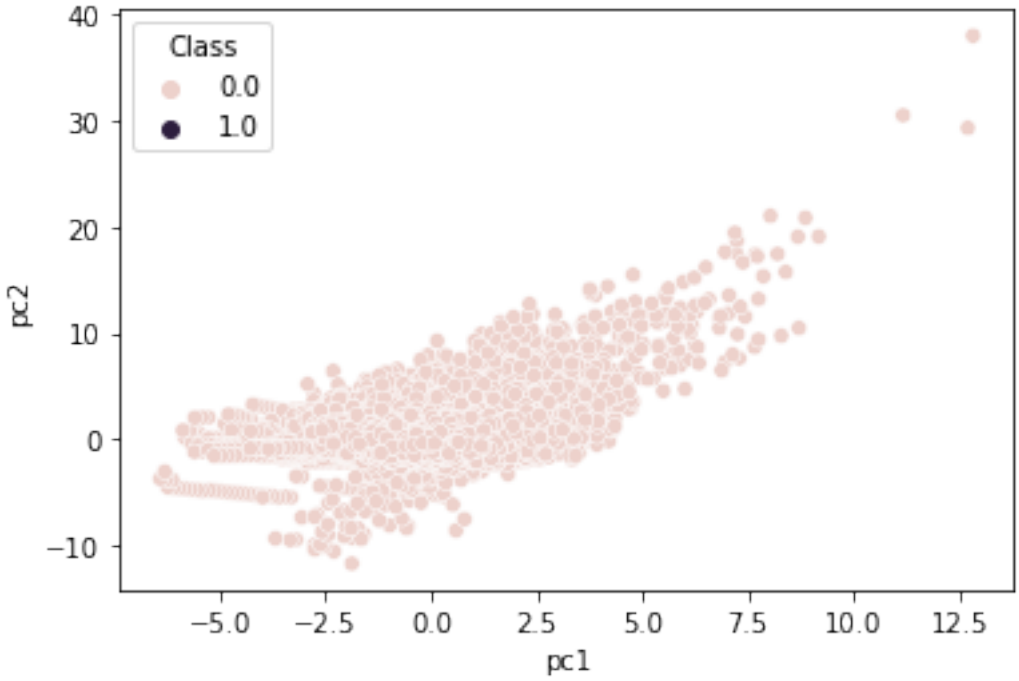
	V1	V2	V3	V4	\
count	283726.000000	283726.000000	283726.000000	283726.000000	
mean	0.005917	-0.004135	0.001613	-0.002966	
std	1.948026	1.646703	1.508682	1.414184	
min	-56.407510	-72.715728	-48.325589	-5.683171	
25%	-0.915951	-0.600321	-0.889682	-0.850134	
50%	0.020384	0.063949	0.179963	-0.022248	
75%	1.316068	0.800283	1.026960	0.739647	
max	2.454930	22.057729	9.382558	16.875344	

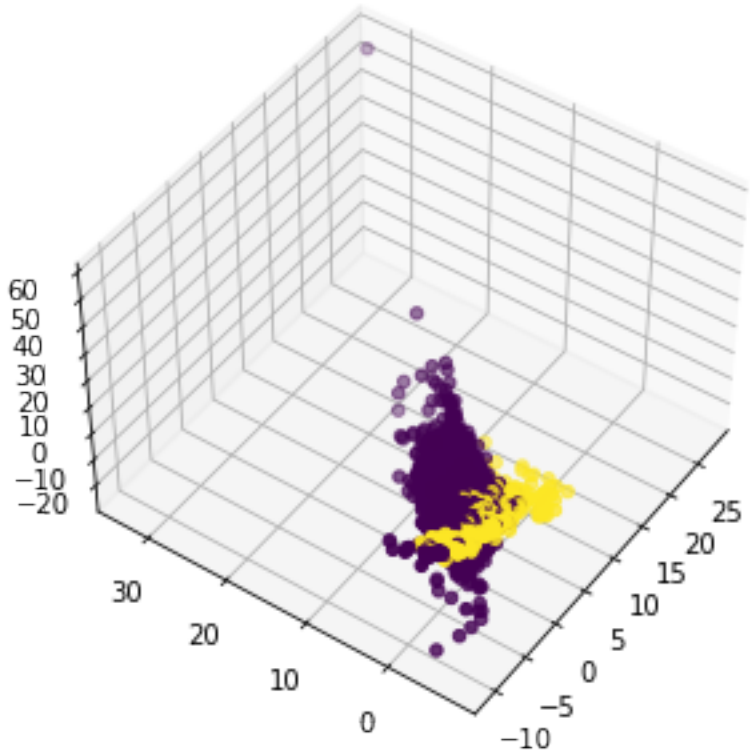
	V5	V6	V7	V8	\
count	283726.000000	283726.000000	283726.000000	283726.000000	
mean	0.001828	-0.001139	0.001801	-0.000854	

```
std          1.377008       1.331931       1.227664       1.179054
min       -113.743307     -26.160506     -43.557242     -73.216718
25%         -0.689830      -0.769031      -0.552509      -0.208828
50%         -0.053468      -0.275168       0.040859       0.021898
75%          0.612218       0.396792       0.570474       0.325704
max         34.801666      73.301626     120.589494      20.007208

                 V9             V10            V11            V12  \
count  283726.000000  283726.000000  283726.000000  283726.000000
mean       -0.001596      -0.001441       0.000202      -0.000715
std         1.095492       1.076407       1.018720       0.994674
min       -13.434066     -24.588262      -4.797473     -18.683715
25%        -0.644221      -0.535578      -0.761649      -0.406198
50%        -0.052596      -0.093237      -0.032306       0.139072
75%         0.595977       0.453619       0.739579       0.616976
max        15.594995      23.745136      12.018913       7.848392

                 V14            V16            V17            V18  \
count  283726.000000  283726.000000  283726.000000  283726.000000
mean        0.000252       0.001162       0.000170       0.001515
std         0.952215       0.873696       0.842507       0.837378
min       -19.214325     -14.129855     -25.162799      -9.498746
25%        -0.425732      -0.466860      -0.483928      -0.498014
50%         0.050209       0.067119      -0.065867      -0.002142
75%         0.492336       0.523512       0.398972       0.501956
max        10.526766      17.315112       9.253526       5.041069

                 V20         Amount
count  283726.000000  283726.000000
mean        0.000187      88.472687
std         0.769984     250.399437
min       -54.497720       0.000000
25%        -0.211469       5.600000
50%        -0.062353      22.000000
75%         0.133207      77.510000
max        39.420904   25691.160000
```

[19]:
```
features = ['V1','V2','V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10', 'V11',␣
 ↪'V12', 'V14', 'V16', 'V17', 'V18', 'V20']
x = X_new.loc[:, features].values
x = StandardScaler().fit_transform(x)

pca = PCA(n_components=3)
pCa = pca.fit_transform(x)
X_3d = pd.DataFrame(data = pCa, columns = ['pc1', 'pc2', 'pc3'])
data_3d = pd.concat ([X_3d,Y], axis=1)
```

```
pca = PCA(n_components=2)
pCa = pca.fit_transform(x)
X_2d = pd.DataFrame(data = pCa, columns = ['pc1', 'pc2'])
data_2d = pd.concat ([X_2d,Y], axis=1)


sns.scatterplot(x = 'pc1', y = 'pc2', data = data_2d, hue = "Class")


plt.figure(figsize=(6,5))
axes = plt.axes(projection='3d')
axes.scatter3D(X_3d['pc1'], X_3d['pc2'],X_3d['pc3'], c= Y)

axes.view_init(45, 215)
plt.show()
```





On a next step I would like to do some PCA but only with features that have good prediction value for the labels of our model

Also it would be good to have some way of counting the outliers in the above empirical distribution

We could also be able to do some features selection using it specifically for random forest and trees

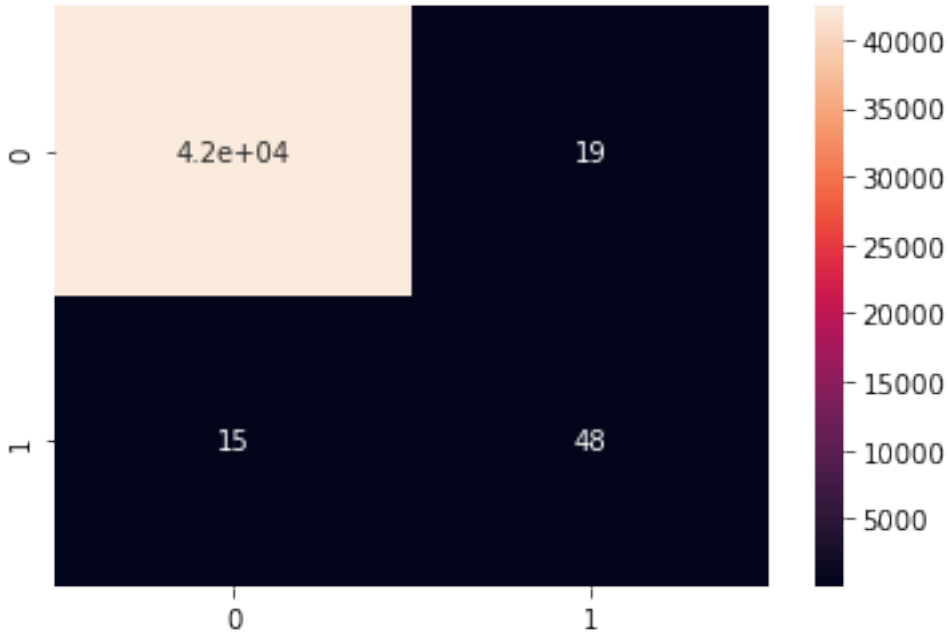
[4]:
```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, accuracy_score, confusion_matrix
def error_comp_new(method):
    n = 10
    accuracy= np.zeros(n)
    matrix_con = np.zeros((2,2))
    for i in range(n):
        X_train, X_test, y_train, y_test = train_test_split(X_new, Y,␣
↪test_size=0.15, random_state=i)
        method.fit(X_train, np.ravel(y_train))
        accuracy[i] = accuracy_score(np.ravel(y_test),method.predict(X_test))
        matrix_con += confusion_matrix(np.ravel(y_test), method.
↪predict(X_test)) # [[TN, FP], [FN, TP]]
    print('Accuracy selected features', accuracy.mean())
    print('Confusion matrix for all features')
```

```
    sns.heatmap(matrix_con/n, annot=True)
    print('Percentage of correct fraud detected', matrix_con[1,1]/(n*y_test.
↪value_counts()[1]))
    plt.show()
```

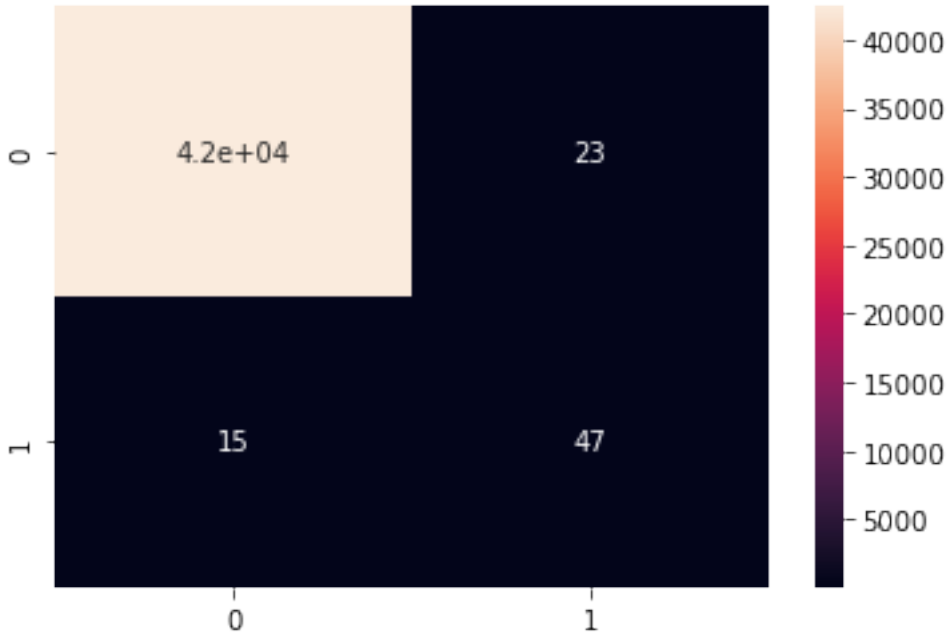
We test svm but as the dataset is very large we need the following SDG algo. We test it for multiple loss function.

[5]:
```
from sklearn.linear_model import SGDClassifier
loss_list = ['log', 'modified_huber', 'perceptron']
penalty = "l1"  #["l2", "l1", "elasticnet"]
for loss in loss_list:
    print('Loss function: ',loss)
    sdg = SGDClassifier(loss=loss, penalty="l1")
    error_comp_new(sdg)
```

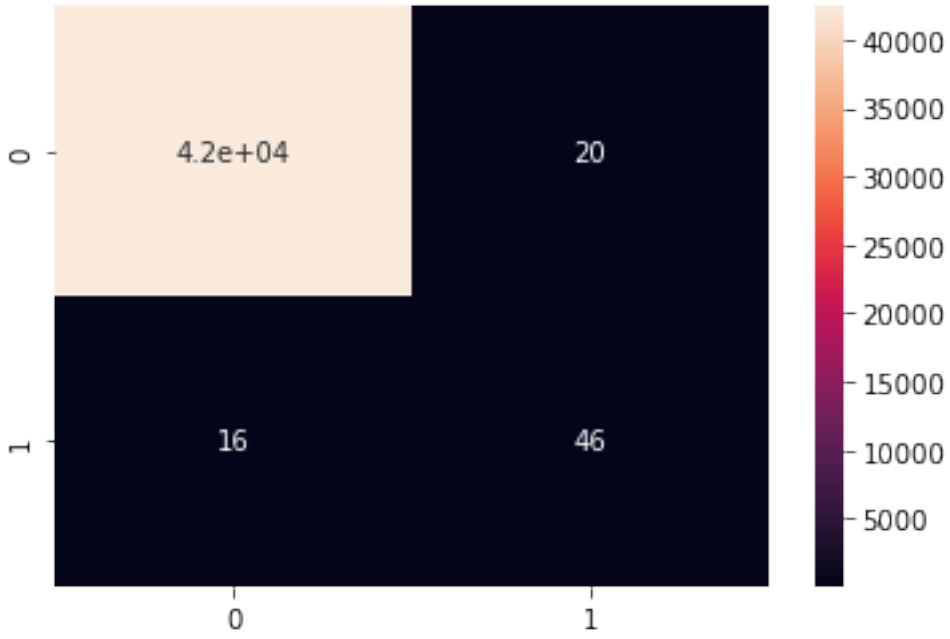
```
Loss function:  log
Accuracy selected features 0.99921050776569
Confusion matrix for all features
Percentage of correct fraud detected 0.7933333333333333
```




```
Loss function:  modified_huber
Accuracy selected features 0.9990977231607886
Confusion matrix for all features
Percentage of correct fraud detected 0.7833333333333333
```

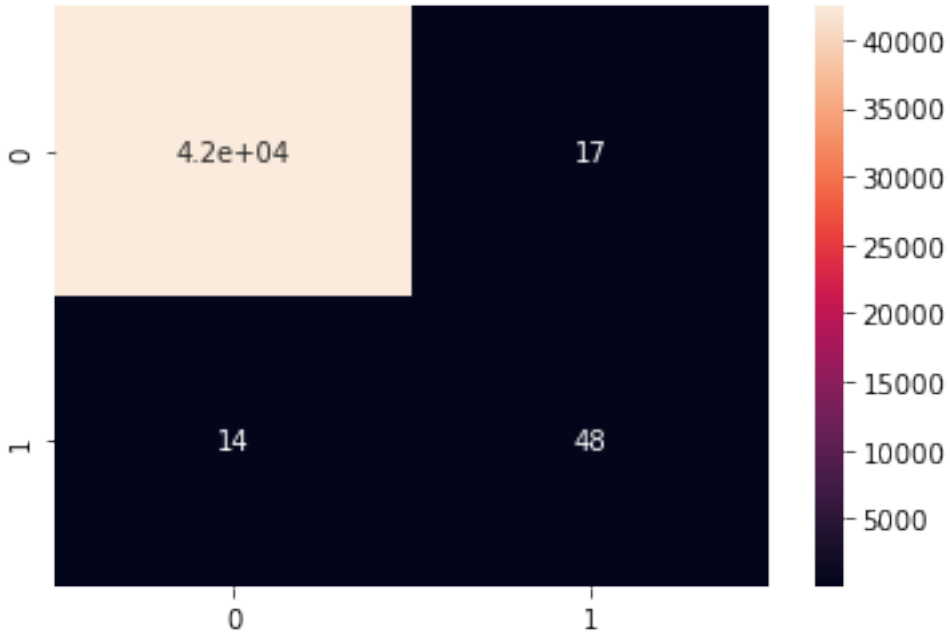



```
Loss function:  perceptron
Accuracy selected features 0.9991494161047016
Confusion matrix for all features
Percentage of correct fraud detected 0.77
```



[18]:
```
from sklearn.tree import DecisionTreeClassifier, export_graphviz, plot_tree
criterion_list = ['entropy']
splitter_list = ['best']
for criterion in criterion_list:
    print('Criterion: ',criterion)
    for splitter in splitter_list:
        print('Splitter: ',splitter)
        dtc = DecisionTreeClassifier(criterion = criterion, splitter = splitter)
        error_comp_new(dtc)
#tree_draw = export_graphviz(dtc)
plot_tree(dtc)
```

```
Criterion:  entropy
Splitter:  best
Accuracy selected features 0.9992669000681407
Confusion matrix for all features
Percentage of correct fraud detected 0.8
```



[18]:
```
[Text(0.2894251054852321, 0.9761904761904762, 'X[14] <= -2.677\nentropy =
0.018\nsamples = 241167\nvalue = [240754, 413]'),
 Text(0.15005274261603377, 0.9285714285714286, 'X[11] <= -2.18\nentropy =
0.768\nsamples = 370\nvalue = [83, 287]'),
 Text(0.1060126582278481, 0.8809523809523809, 'X[12] <= -3.43\nentropy =
0.633\nsamples = 339\nvalue = [54, 285]'),
 Text(0.043248945147679324, 0.8333333333333334, 'X[16] <= 0.55\nentropy =
0.493\nsamples = 306\nvalue = [33, 273]'),
```

```
 Text(0.016877637130801686, 0.7857142857142857, 'X[13] <= -0.132\nentropy =
0.096\nsamples = 162\nvalue = [2, 160]'),
 Text(0.008438818565400843, 0.7380952380952381, 'entropy = 0.0\nsamples =
160\nvalue = [0, 160]'),
 Text(0.025316455696202531, 0.7380952380952381, 'entropy = 0.0\nsamples =
2\nvalue = [2, 0]'),
 Text(0.06962025316455696, 0.7857142857142857, 'X[8] <= -3.41\nentropy =
0.751\nsamples = 144\nvalue = [31, 113]'),
 Text(0.04219409282700422, 0.7380952380952381, 'X[5] <= -4.473\nentropy =
0.247\nsamples = 73\nvalue = [3, 70]'),
 Text(0.03375527426160337, 0.6904761904761905, 'X[17] <= 94.99\nentropy =
0.722\nsamples = 15\nvalue = [3, 12]'),
 Text(0.025316455696202531, 0.6428571428571429, 'entropy = 0.0\nsamples =
3\nvalue = [3, 0]'),
 Text(0.04219409282700422, 0.6428571428571429, 'entropy = 0.0\nsamples =
12\nvalue = [0, 12]'),
 Text(0.05063291139240506, 0.6904761904761905, 'entropy = 0.0\nsamples =
58\nvalue = [0, 58]'),
 Text(0.0970464135021097, 0.7380952380952381, 'X[15] <= -2.566\nentropy =
0.968\nsamples = 71\nvalue = [28, 43]'),
 Text(0.0759493670886076, 0.6904761904761905, 'X[5] <= -2.579\nentropy =
0.822\nsamples = 35\nvalue = [26, 9]'),
 Text(0.05907172995780591, 0.6428571428571429, 'X[17] <= 94.99\nentropy =
0.529\nsamples = 25\nvalue = [22, 3]'),
 Text(0.05063291139240506, 0.5952380952380952, 'X[9] <= -12.374\nentropy =
0.258\nsamples = 23\nvalue = [22, 1]'),
 Text(0.04219409282700422, 0.5476190476190477, 'entropy = 0.0\nsamples =
1\nvalue = [0, 1]'),
 Text(0.05907172995780591, 0.5476190476190477, 'entropy = 0.0\nsamples =
22\nvalue = [22, 0]'),
 Text(0.06751054852320675, 0.5952380952380952, 'entropy = 0.0\nsamples =
2\nvalue = [0, 2]'),
 Text(0.09282700421940929, 0.6428571428571429, 'X[9] <= -4.878\nentropy =
0.971\nsamples = 10\nvalue = [4, 6]'),
 Text(0.08438818565400844, 0.5952380952380952, 'entropy = 0.0\nsamples =
5\nvalue = [0, 5]'),
 Text(0.10126582278481013, 0.5952380952380952, 'X[16] <= 0.574\nentropy =
0.722\nsamples = 5\nvalue = [4, 1]'),
 Text(0.09282700421940929, 0.5476190476190477, 'entropy = 0.0\nsamples =
1\nvalue = [0, 1]'),
 Text(0.10970464135021098, 0.5476190476190477, 'entropy = 0.0\nsamples =
4\nvalue = [4, 0]'),
 Text(0.11814345991561181, 0.6904761904761905, 'X[4] <= -0.129\nentropy =
0.31\nsamples = 36\nvalue = [2, 34]'),
 Text(0.10970464135021098, 0.6428571428571429, 'entropy = 0.0\nsamples =
32\nvalue = [0, 32]'),
 Text(0.12658227848101267, 0.6428571428571429, 'X[7] <= 0.912\nentropy =
```

```
1.0\nsamples = 4\nvalue = [2, 2]'),
 Text(0.11814345991561181, 0.5952380952380952, 'entropy = 0.0\nsamples =
2\nvalue = [2, 0]'),
 Text(0.1350210970464135, 0.5952380952380952, 'entropy = 0.0\nsamples = 2\nvalue
= [0, 2]'),
 Text(0.16877637130801687, 0.8333333333333334, 'X[13] <= -2.982\nentropy =
0.946\nsamples = 33\nvalue = [21, 12]'),
 Text(0.1518987341772152, 0.7857142857142857, 'X[5] <= -0.956\nentropy =
0.779\nsamples = 13\nvalue = [3, 10]'),
 Text(0.14345991561181434, 0.7380952380952381, 'X[6] <= -1.607\nentropy =
0.971\nsamples = 5\nvalue = [3, 2]'),
 Text(0.1350210970464135, 0.6904761904761905, 'entropy = 0.0\nsamples = 3\nvalue
= [3, 0]'),
 Text(0.1518987341772152, 0.6904761904761905, 'entropy = 0.0\nsamples = 2\nvalue
= [0, 2]'),
 Text(0.16033755274261605, 0.7380952380952381, 'entropy = 0.0\nsamples =
8\nvalue = [0, 8]'),
 Text(0.18565400843881857, 0.7857142857142857, 'X[4] <= -0.827\nentropy =
0.469\nsamples = 20\nvalue = [18, 2]'),
 Text(0.17721518987341772, 0.7380952380952381, 'entropy = 0.0\nsamples =
15\nvalue = [15, 0]'),
 Text(0.1940928270042194, 0.7380952380952381, 'X[9] <= -1.799\nentropy =
0.971\nsamples = 5\nvalue = [3, 2]'),
 Text(0.18565400843881857, 0.6904761904761905, 'entropy = 0.0\nsamples =
2\nvalue = [0, 2]'),
 Text(0.20253164556962025, 0.6904761904761905, 'entropy = 0.0\nsamples =
3\nvalue = [3, 0]'),
 Text(0.1940928270042194, 0.8809523809523809, 'X[6] <= -4.179\nentropy =
0.345\nsamples = 31\nvalue = [29, 2]'),
 Text(0.18565400843881857, 0.8333333333333334, 'entropy = 0.0\nsamples =
2\nvalue = [0, 2]'),
 Text(0.20253164556962025, 0.8333333333333334, 'entropy = 0.0\nsamples =
29\nvalue = [29, 0]'),
 Text(0.4287974683544304, 0.9285714285714286, 'X[12] <= -4.219\nentropy =
0.006\nsamples = 240797\nvalue = [240671, 126]'),
 Text(0.24472573839662448, 0.8809523809523809, 'X[9] <= -1.859\nentropy =
0.708\nsamples = 316\nvalue = [255, 61]'),
 Text(0.22784810126582278, 0.8333333333333334, 'X[13] <= 2.872\nentropy =
0.778\nsamples = 74\nvalue = [17, 57]'),
 Text(0.21940928270042195, 0.7857142857142857, 'X[11] <= -1.899\nentropy =
0.498\nsamples = 64\nvalue = [7, 57]'),
 Text(0.2109704641350211, 0.7380952380952381, 'entropy = 0.0\nsamples =
44\nvalue = [0, 44]'),
 Text(0.22784810126582278, 0.7380952380952381, 'X[2] <= -1.701\nentropy =
0.934\nsamples = 20\nvalue = [7, 13]'),
 Text(0.21940928270042195, 0.6904761904761905, 'X[10] <= 2.156\nentropy =
0.946\nsamples = 11\nvalue = [7, 4]'),
```

```
 Text(0.2109704641350211, 0.6428571428571429, 'entropy = 0.0\nsamples = 7\nvalue
= [7, 0]'),
 Text(0.22784810126582278, 0.6428571428571429, 'entropy = 0.0\nsamples =
4\nvalue = [0, 4]'),
 Text(0.23628691983122363, 0.6904761904761905, 'entropy = 0.0\nsamples =
9\nvalue = [0, 9]'),
 Text(0.23628691983122363, 0.7857142857142857, 'entropy = 0.0\nsamples =
10\nvalue = [10, 0]'),
 Text(0.2616033755274262, 0.8333333333333334, 'X[3] <= 4.501\nentropy =
0.121\nsamples = 242\nvalue = [238, 4]'),
 Text(0.25316455696202533, 0.7857142857142857, 'entropy = 0.0\nsamples =
209\nvalue = [209, 0]'),
 Text(0.270042194092827, 0.7857142857142857, 'X[3] <= 4.644\nentropy =
0.533\nsamples = 33\nvalue = [29, 4]'),
 Text(0.2616033755274262, 0.7380952380952381, 'entropy = 0.0\nsamples = 3\nvalue
= [0, 3]'),
 Text(0.27848101265822783, 0.7380952380952381, 'X[13] <= 0.562\nentropy =
0.211\nsamples = 30\nvalue = [29, 1]'),
 Text(0.270042194092827, 0.6904761904761905, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(0.2869198312236287, 0.6904761904761905, 'entropy = 0.0\nsamples =
29\nvalue = [29, 0]'),
 Text(0.6128691983122363, 0.8809523809523809, 'X[3] <= 1.661\nentropy =
0.004\nsamples = 240481\nvalue = [240416, 65]'),
 Text(0.43723628691983124, 0.8333333333333334, 'X[12] <= -0.819\nentropy =
0.002\nsamples = 221126\nvalue = [221094, 32]'),
 Text(0.34177215189873417, 0.7857142857142857, 'X[12] <= -0.82\nentropy =
0.007\nsamples = 29240\nvalue = [29222, 18]'),
 Text(0.3206751054852321, 0.7380952380952381, 'X[14] <= 0.716\nentropy =
0.007\nsamples = 29218\nvalue = [29202, 16]'),
 Text(0.3037974683544304, 0.6904761904761905, 'X[16] <= -0.187\nentropy =
0.002\nsamples = 17402\nvalue = [17400, 2]'),
 Text(0.29535864978902954, 0.6428571428571429, 'X[16] <= -0.187\nentropy =
0.007\nsamples = 3773\nvalue = [3771, 2]'),
 Text(0.2869198312236287, 0.5952380952380952, 'X[5] <= -1.578\nentropy =
0.004\nsamples = 3772\nvalue = [3771, 1]'),
 Text(0.27848101265822783, 0.5476190476190477, 'X[5] <= -1.582\nentropy =
0.063\nsamples = 135\nvalue = [134, 1]'),
 Text(0.270042194092827, 0.5, 'entropy = 0.0\nsamples = 134\nvalue = [134, 0]'),
 Text(0.2869198312236287, 0.5, 'entropy = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(0.29535864978902954, 0.5476190476190477, 'entropy = 0.0\nsamples =
3637\nvalue = [3637, 0]'),
 Text(0.3037974683544304, 0.5952380952380952, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(0.31223628691983124, 0.6428571428571429, 'entropy = 0.0\nsamples =
13629\nvalue = [13629, 0]'),
 Text(0.33755274261603374, 0.6904761904761905, 'X[14] <= 0.717\nentropy =
```

```
0.013\nsamples = 11816\nvalue = [11802, 14]'),
 Text(0.3291139240506329, 0.6428571428571429, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(0.3459915611814346, 0.6428571428571429, 'X[7] <= -3.141\nentropy =
0.012\nsamples = 11815\nvalue = [11802, 13]'),
 Text(0.32489451476793246, 0.5952380952380952, 'X[11] <= -0.336\nentropy =
0.17\nsamples = 119\nvalue = [116, 3]'),
 Text(0.31645569620253167, 0.5476190476190477, 'X[16] <= -0.969\nentropy =
0.544\nsamples = 24\nvalue = [21, 3]'),
 Text(0.3080168776371308, 0.5, 'X[17] <= 27.5\nentropy = 0.954\nsamples =
8\nvalue = [5, 3]'),
 Text(0.29957805907172996, 0.4523809523809524, 'entropy = 0.0\nsamples =
3\nvalue = [0, 3]'),
 Text(0.31645569620253167, 0.4523809523809524, 'entropy = 0.0\nsamples =
5\nvalue = [5, 0]'),
 Text(0.32489451476793246, 0.5, 'entropy = 0.0\nsamples = 16\nvalue = [16, 0]'),
 Text(0.3333333333333333, 0.5476190476190477, 'entropy = 0.0\nsamples =
95\nvalue = [95, 0]'),
 Text(0.3670886075949367, 0.5952380952380952, 'X[6] <= 2.778\nentropy =
0.01\nsamples = 11696\nvalue = [11686, 10]'),
 Text(0.350210970464135, 0.5476190476190477, 'X[7] <= 0.147\nentropy =
0.008\nsamples = 11633\nvalue = [11625, 8]'),
 Text(0.34177215189873417, 0.5, 'X[7] <= 0.147\nentropy = 0.012\nsamples =
7561\nvalue = [7553, 8]'),
 Text(0.3333333333333333, 0.4523809523809524, 'X[2] <= -2.581\nentropy =
0.011\nsamples = 7560\nvalue = [7553, 7]'),
 Text(0.31645569620253167, 0.40476190476190477, 'X[2] <= -2.584\nentropy =
0.037\nsamples = 767\nvalue = [764, 3]'),
 Text(0.3080168776371308, 0.35714285714285715, 'X[13] <= -0.383\nentropy =
0.014\nsamples = 765\nvalue = [764, 1]'),
 Text(0.29957805907172996, 0.30952380952380953, 'X[13] <= -0.4\nentropy =
0.211\nsamples = 30\nvalue = [29, 1]'),
 Text(0.2911392405063291, 0.2619047619047619, 'entropy = 0.0\nsamples =
29\nvalue = [29, 0]'),
 Text(0.3080168776371308, 0.2619047619047619, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(0.31645569620253167, 0.30952380952380953, 'entropy = 0.0\nsamples =
735\nvalue = [735, 0]'),
 Text(0.32489451476793246, 0.35714285714285715, 'entropy = 0.0\nsamples =
2\nvalue = [0, 2]'),
 Text(0.350210970464135, 0.40476190476190477, 'X[15] <= 0.377\nentropy =
0.007\nsamples = 6793\nvalue = [6789, 4]'),
 Text(0.34177215189873417, 0.35714285714285715, 'X[15] <= 0.376\nentropy =
0.012\nsamples = 3687\nvalue = [3683, 4]'),
 Text(0.3333333333333333, 0.30952380952380953, 'X[5] <= -0.675\nentropy =
0.01\nsamples = 3686\nvalue = [3683, 3]'),
 Text(0.32489451476793246, 0.2619047619047619, 'entropy = 0.0\nsamples =
```

```
1939\nvalue = [1939, 0]'),
 Text(0.34177215189873417, 0.2619047619047619, 'X[5] <= -0.674\nentropy =
0.018\nsamples = 1747\nvalue = [1744, 3]'),
 Text(0.3333333333333333, 0.21428571428571427, 'entropy = 0.0\nsamples =
1\nvalue = [0, 1]'),
 Text(0.350210970464135, 0.21428571428571427, 'X[13] <= 1.533\nentropy =
0.013\nsamples = 1746\nvalue = [1744, 2]'),
 Text(0.34177215189873417, 0.16666666666666666, 'entropy = 0.0\nsamples =
1344\nvalue = [1344, 0]'),
 Text(0.35864978902953587, 0.16666666666666666, 'X[13] <= 1.534\nentropy =
0.045\nsamples = 402\nvalue = [400, 2]'),
 Text(0.350210970464135, 0.11904761904761904, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(0.3670886075949367, 0.11904761904761904, 'X[6] <= -1.081\nentropy =
0.025\nsamples = 401\nvalue = [400, 1]'),
 Text(0.35864978902953587, 0.07142857142857142, 'X[6] <= -1.083\nentropy =
0.191\nsamples = 34\nvalue = [33, 1]'),
 Text(0.350210970464135, 0.023809523809523808, 'entropy = 0.0\nsamples =
33\nvalue = [33, 0]'),
 Text(0.3670886075949367, 0.023809523809523808, 'entropy = 0.0\nsamples =
1\nvalue = [0, 1]'),
 Text(0.3755274261603376, 0.07142857142857142, 'entropy = 0.0\nsamples =
367\nvalue = [367, 0]'),
 Text(0.350210970464135, 0.30952380952380953, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(0.35864978902953587, 0.35714285714285715, 'entropy = 0.0\nsamples =
3106\nvalue = [3106, 0]'),
 Text(0.350210970464135, 0.4523809523809524, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(0.35864978902953587, 0.5, 'entropy = 0.0\nsamples = 4072\nvalue = [4072,
0]'),
 Text(0.38396624472573837, 0.5476190476190477, 'X[6] <= 2.792\nentropy =
0.203\nsamples = 63\nvalue = [61, 2]'),
 Text(0.3755274261603376, 0.5, 'entropy = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(0.3924050632911392, 0.5, 'X[0] <= -0.367\nentropy = 0.119\nsamples =
62\nvalue = [61, 1]'),
 Text(0.38396624472573837, 0.4523809523809524, 'entropy = 0.0\nsamples =
57\nvalue = [57, 0]'),
 Text(0.4008438818565401, 0.4523809523809524, 'X[1] <= 1.096\nentropy =
0.722\nsamples = 5\nvalue = [4, 1]'),
 Text(0.3924050632911392, 0.40476190476190477, 'entropy = 0.0\nsamples =
4\nvalue = [4, 0]'),
 Text(0.4092827004219409, 0.40476190476190477, 'entropy = 0.0\nsamples =
1\nvalue = [0, 1]'),
 Text(0.3628691983122363, 0.7380952380952381, 'X[7] <= 0.145\nentropy =
0.439\nsamples = 22\nvalue = [20, 2]'),
 Text(0.35443037974683544, 0.6904761904761905, 'entropy = 0.0\nsamples =
```

```
17\nvalue = [17, 0]'),
 Text(0.37130801687763715, 0.6904761904761905, 'X[7] <= 0.32\nentropy =
0.971\nsamples = 5\nvalue = [3, 2]'),
 Text(0.3628691983122363, 0.6428571428571429, 'entropy = 0.0\nsamples = 2\nvalue
= [0, 2]'),
 Text(0.379746835443038, 0.6428571428571429, 'entropy = 0.0\nsamples = 3\nvalue
= [3, 0]'),
 Text(0.5327004219409283, 0.7857142857142857, 'X[6] <= 1.302\nentropy =
0.001\nsamples = 191886\nvalue = [191872, 14]'),
 Text(0.4957805907172996, 0.7380952380952381, 'X[16] <= -0.511\nentropy =
0.001\nsamples = 182261\nvalue = [182255, 6]'),
 Text(0.4641350210970464, 0.6904761904761905, 'X[16] <= -0.511\nentropy =
0.005\nsamples = 9983\nvalue = [9979, 4]'),
 Text(0.45569620253164556, 0.6428571428571429, 'X[4] <= 11.007\nentropy =
0.004\nsamples = 9982\nvalue = [9979, 3]'),
 Text(0.4345991561181435, 0.5952380952380952, 'X[17] <= 0.705\nentropy =
0.003\nsamples = 9980\nvalue = [9978, 2]'),
 Text(0.4177215189873418, 0.5476190476190477, 'X[17] <= 0.685\nentropy =
0.089\nsamples = 89\nvalue = [88, 1]'),
 Text(0.4092827004219409, 0.5, 'entropy = 0.0\nsamples = 85\nvalue = [85, 0]'),
 Text(0.42616033755274263, 0.5, 'X[13] <= -0.797\nentropy = 0.811\nsamples =
4\nvalue = [3, 1]'),
 Text(0.4177215189873418, 0.4523809523809524, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(0.4345991561181435, 0.4523809523809524, 'entropy = 0.0\nsamples = 3\nvalue
= [3, 0]'),
 Text(0.45147679324894513, 0.5476190476190477, 'X[3] <= 0.894\nentropy =
0.001\nsamples = 9891\nvalue = [9890, 1]'),
 Text(0.4430379746835443, 0.5, 'entropy = 0.0\nsamples = 9062\nvalue = [9062,
0]'),
 Text(0.459915611814346, 0.5, 'X[3] <= 0.894\nentropy = 0.013\nsamples =
829\nvalue = [828, 1]'),
 Text(0.45147679324894513, 0.4523809523809524, 'entropy = 0.0\nsamples =
1\nvalue = [0, 1]'),
 Text(0.46835443037974683, 0.4523809523809524, 'entropy = 0.0\nsamples =
828\nvalue = [828, 0]'),
 Text(0.4767932489451477, 0.5952380952380952, 'X[5] <= -7.071\nentropy =
1.0\nsamples = 2\nvalue = [1, 1]'),
 Text(0.46835443037974683, 0.5476190476190477, 'entropy = 0.0\nsamples =
1\nvalue = [1, 0]'),
 Text(0.48523206751054854, 0.5476190476190477, 'entropy = 0.0\nsamples =
1\nvalue = [0, 1]'),
 Text(0.47257383966244726, 0.6428571428571429, 'entropy = 0.0\nsamples =
1\nvalue = [0, 1]'),
 Text(0.5274261603375527, 0.6904761904761905, 'X[12] <= -0.658\nentropy =
0.0\nsamples = 172278\nvalue = [172276, 2]'),
 Text(0.5189873417721519, 0.6428571428571429, 'X[12] <= -0.658\nentropy =
```

```
0.003\nsamples = 7958\nvalue = [7956, 2]'),
 Text(0.510548523206751, 0.5952380952380952, 'X[12] <= -0.661\nentropy =
0.002\nsamples = 7957\nvalue = [7956, 1]'),
 Text(0.5021097046413502, 0.5476190476190477, 'entropy = 0.0\nsamples =
7809\nvalue = [7809, 0]'),
 Text(0.5189873417721519, 0.5476190476190477, 'X[12] <= -0.661\nentropy =
0.058\nsamples = 148\nvalue = [147, 1]'),
 Text(0.510548523206751, 0.5, 'entropy = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(0.5274261603375527, 0.5, 'entropy = 0.0\nsamples = 147\nvalue = [147,
0]'),
 Text(0.5274261603375527, 0.5952380952380952, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(0.5358649789029536, 0.6428571428571429, 'entropy = 0.0\nsamples =
164320\nvalue = [164320, 0]'),
 Text(0.569620253164557, 0.7380952380952381, 'X[1] <= -0.045\nentropy =
0.01\nsamples = 9625\nvalue = [9617, 8]'),
 Text(0.5611814345991561, 0.6904761904761905, 'X[6] <= 1.302\nentropy =
0.023\nsamples = 3576\nvalue = [3568, 8]'),
 Text(0.5527426160337553, 0.6428571428571429, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(0.569620253164557, 0.6428571428571429, 'X[1] <= -0.046\nentropy =
0.02\nsamples = 3575\nvalue = [3568, 7]'),
 Text(0.5611814345991561, 0.5952380952380952, 'X[3] <= 0.162\nentropy =
0.018\nsamples = 3574\nvalue = [3568, 6]'),
 Text(0.5527426160337553, 0.5476190476190477, 'entropy = 0.0\nsamples =
2248\nvalue = [2248, 0]'),
 Text(0.569620253164557, 0.5476190476190477, 'X[3] <= 0.164\nentropy =
0.042\nsamples = 1326\nvalue = [1320, 6]'),
 Text(0.5611814345991561, 0.5, 'entropy = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(0.5780590717299579, 0.5, 'X[11] <= 0.024\nentropy = 0.036\nsamples =
1325\nvalue = [1320, 5]'),
 Text(0.569620253164557, 0.4523809523809524, 'X[11] <= 0.024\nentropy =
0.065\nsamples = 654\nvalue = [649, 5]'),
 Text(0.5611814345991561, 0.40476190476190477, 'X[8] <= -0.137\nentropy =
0.054\nsamples = 653\nvalue = [649, 4]'),
 Text(0.5527426160337553, 0.35714285714285715, 'entropy = 0.0\nsamples =
464\nvalue = [464, 0]'),
 Text(0.569620253164557, 0.35714285714285715, 'X[8] <= -0.135\nentropy =
0.148\nsamples = 189\nvalue = [185, 4]'),
 Text(0.5611814345991561, 0.30952380952380953, 'entropy = 0.0\nsamples =
1\nvalue = [0, 1]'),
 Text(0.5780590717299579, 0.30952380952380953, 'X[6] <= 2.872\nentropy =
0.118\nsamples = 188\nvalue = [185, 3]'),
 Text(0.569620253164557, 0.2619047619047619, 'entropy = 0.0\nsamples =
139\nvalue = [139, 0]'),
 Text(0.5864978902953587, 0.2619047619047619, 'X[5] <= 0.515\nentropy =
0.332\nsamples = 49\nvalue = [46, 3]'),
```

```
 Text(0.5780590717299579, 0.21428571428571427, 'X[14] <= 0.11\nentropy =
0.985\nsamples = 7\nvalue = [4, 3]'),
 Text(0.569620253164557, 0.16666666666666666, 'X[11] <= -0.727\nentropy =
0.811\nsamples = 4\nvalue = [1, 3]'),
 Text(0.5611814345991561, 0.11904761904761904, 'entropy = 0.0\nsamples =
1\nvalue = [1, 0]'),
 Text(0.5780590717299579, 0.11904761904761904, 'entropy = 0.0\nsamples =
3\nvalue = [0, 3]'),
 Text(0.5864978902953587, 0.16666666666666666, 'entropy = 0.0\nsamples =
3\nvalue = [3, 0]'),
 Text(0.5949367088607594, 0.21428571428571427, 'entropy = 0.0\nsamples =
42\nvalue = [42, 0]'),
 Text(0.5780590717299579, 0.40476190476190477, 'entropy = 0.0\nsamples =
1\nvalue = [0, 1]'),
 Text(0.5864978902953587, 0.4523809523809524, 'entropy = 0.0\nsamples =
671\nvalue = [671, 0]'),
 Text(0.5780590717299579, 0.5952380952380952, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(0.5780590717299579, 0.6904761904761905, 'entropy = 0.0\nsamples =
6049\nvalue = [6049, 0]'),
 Text(0.7885021097046413, 0.8333333333333334, 'X[12] <= -1.017\nentropy =
0.018\nsamples = 19355\nvalue = [19322, 33]'),
 Text(0.6708860759493671, 0.7857142857142857, 'X[10] <= -0.531\nentropy =
0.075\nsamples = 1868\nvalue = [1851, 17]'),
 Text(0.6624472573839663, 0.7380952380952381, 'entropy = 0.0\nsamples =
698\nvalue = [698, 0]'),
 Text(0.679324894514768, 0.7380952380952381, 'X[14] <= 2.845\nentropy =
0.11\nsamples = 1170\nvalue = [1153, 17]'),
 Text(0.6708860759493671, 0.6904761904761905, 'X[8] <= 0.469\nentropy =
0.144\nsamples = 830\nvalue = [813, 17]'),
 Text(0.6624472573839663, 0.6428571428571429, 'X[14] <= -1.693\nentropy =
0.188\nsamples = 592\nvalue = [575, 17]'),
 Text(0.6455696202531646, 0.5952380952380952, 'X[4] <= 0.849\nentropy =
1.0\nsamples = 4\nvalue = [2, 2]'),
 Text(0.6371308016877637, 0.5476190476190477, 'entropy = 0.0\nsamples = 2\nvalue
= [0, 2]'),
 Text(0.6540084388185654, 0.5476190476190477, 'entropy = 0.0\nsamples = 2\nvalue
= [2, 0]'),
 Text(0.679324894514768, 0.5952380952380952, 'X[1] <= 1.393\nentropy =
0.171\nsamples = 588\nvalue = [573, 15]'),
 Text(0.6708860759493671, 0.5476190476190477, 'X[15] <= 1.026\nentropy =
0.232\nsamples = 397\nvalue = [382, 15]'),
 Text(0.620253164556962, 0.5, 'X[16] <= 0.923\nentropy = 0.107\nsamples =
284\nvalue = [280, 4]'),
 Text(0.6033755274261603, 0.4523809523809524, 'X[3] <= 1.662\nentropy =
0.035\nsamples = 271\nvalue = [270, 1]'),
 Text(0.5949367088607594, 0.40476190476190477, 'entropy = 0.0\nsamples =
```

```
1\nvalue = [0, 1]'),
 Text(0.6118143459915611, 0.40476190476190477, 'entropy = 0.0\nsamples =
270\nvalue = [270, 0]'),
 Text(0.6371308016877637, 0.4523809523809524, 'X[1] <= -1.384\nentropy =
0.779\nsamples = 13\nvalue = [10, 3]'),
 Text(0.6286919831223629, 0.40476190476190477, 'entropy = 0.0\nsamples =
9\nvalue = [9, 0]'),
 Text(0.6455696202531646, 0.40476190476190477, 'X[7] <= -0.122\nentropy =
0.811\nsamples = 4\nvalue = [1, 3]'),
 Text(0.6371308016877637, 0.35714285714285715, 'entropy = 0.0\nsamples =
3\nvalue = [0, 3]'),
 Text(0.6540084388185654, 0.35714285714285715, 'entropy = 0.0\nsamples =
1\nvalue = [1, 0]'),
 Text(0.7215189873417721, 0.5, 'X[17] <= 17.775\nentropy = 0.461\nsamples =
113\nvalue = [102, 11]'),
 Text(0.6877637130801688, 0.4523809523809524, 'X[9] <= -0.42\nentropy =
0.738\nsamples = 48\nvalue = [38, 10]'),
 Text(0.679324894514768, 0.40476190476190477, 'entropy = 0.0\nsamples =
18\nvalue = [18, 0]'),
 Text(0.6962025316455697, 0.40476190476190477, 'X[14] <= 1.507\nentropy =
0.918\nsamples = 30\nvalue = [20, 10]'),
 Text(0.6708860759493671, 0.35714285714285715, 'X[14] <= 0.828\nentropy =
0.503\nsamples = 18\nvalue = [16, 2]'),
 Text(0.6624472573839663, 0.30952380952380953, 'X[15] <= 1.151\nentropy =
0.971\nsamples = 5\nvalue = [3, 2]'),
 Text(0.6540084388185654, 0.2619047619047619, 'entropy = 0.0\nsamples = 2\nvalue
= [2, 0]'),
 Text(0.6708860759493671, 0.2619047619047619, 'X[4] <= 2.159\nentropy =
0.918\nsamples = 3\nvalue = [1, 2]'),
 Text(0.6624472573839663, 0.21428571428571427, 'entropy = 0.0\nsamples =
2\nvalue = [0, 2]'),
 Text(0.679324894514768, 0.21428571428571427, 'entropy = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(0.679324894514768, 0.30952380952380953, 'entropy = 0.0\nsamples =
13\nvalue = [13, 0]'),
 Text(0.7215189873417721, 0.35714285714285715, 'X[9] <= 0.266\nentropy =
0.918\nsamples = 12\nvalue = [4, 8]'),
 Text(0.7130801687763713, 0.30952380952380953, 'X[15] <= 1.096\nentropy =
0.722\nsamples = 10\nvalue = [2, 8]'),
 Text(0.7046413502109705, 0.2619047619047619, 'X[8] <= -0.752\nentropy =
0.918\nsamples = 3\nvalue = [2, 1]'),
 Text(0.6962025316455697, 0.21428571428571427, 'entropy = 0.0\nsamples =
1\nvalue = [0, 1]'),
 Text(0.7130801687763713, 0.21428571428571427, 'entropy = 0.0\nsamples =
2\nvalue = [2, 0]'),
 Text(0.7215189873417721, 0.2619047619047619, 'entropy = 0.0\nsamples = 7\nvalue
= [0, 7]'),
```

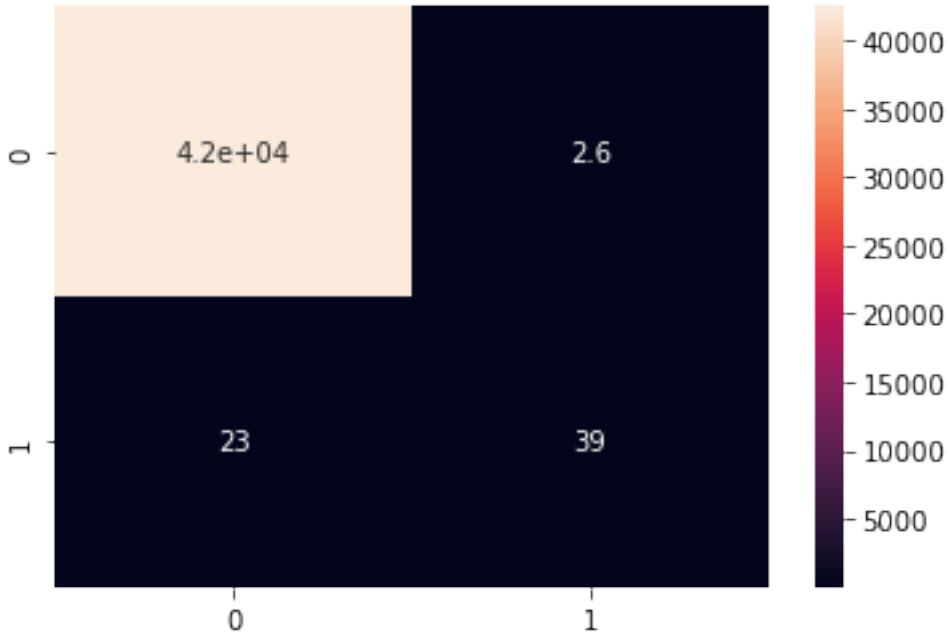
```
 Text(0.729957805907173, 0.30952380952380953, 'entropy = 0.0\nsamples = 2\nvalue
= [2, 0]'),
 Text(0.7552742616033755, 0.4523809523809524, 'X[10] <= -0.412\nentropy =
0.115\nsamples = 65\nvalue = [64, 1]'),
 Text(0.7468354430379747, 0.40476190476190477, 'X[8] <= -0.432\nentropy =
0.918\nsamples = 3\nvalue = [2, 1]'),
 Text(0.7383966244725738, 0.35714285714285715, 'entropy = 0.0\nsamples =
2\nvalue = [2, 0]'),
 Text(0.7552742616033755, 0.35714285714285715, 'entropy = 0.0\nsamples =
1\nvalue = [0, 1]'),
 Text(0.7637130801687764, 0.40476190476190477, 'entropy = 0.0\nsamples =
62\nvalue = [62, 0]'),
 Text(0.6877637130801688, 0.5476190476190477, 'entropy = 0.0\nsamples =
191\nvalue = [191, 0]'),
 Text(0.679324894514768, 0.6428571428571429, 'entropy = 0.0\nsamples =
238\nvalue = [238, 0]'),
 Text(0.6877637130801688, 0.6904761904761905, 'entropy = 0.0\nsamples =
340\nvalue = [340, 0]'),
 Text(0.9061181434599156, 0.7857142857142857, 'X[7] <= -0.11\nentropy =
0.011\nsamples = 17487\nvalue = [17471, 16]'),
 Text(0.8459915611814346, 0.7380952380952381, 'X[7] <= -0.11\nentropy =
0.029\nsamples = 4371\nvalue = [4358, 13]'),
 Text(0.8375527426160337, 0.6904761904761905, 'X[4] <= 1.394\nentropy =
0.027\nsamples = 4370\nvalue = [4358, 12]'),
 Text(0.7890295358649789, 0.6428571428571429, 'X[14] <= -0.122\nentropy =
0.017\nsamples = 3686\nvalue = [3680, 6]'),
 Text(0.7805907172995781, 0.5952380952380952, 'entropy = 0.0\nsamples =
1872\nvalue = [1872, 0]'),
 Text(0.7974683544303798, 0.5952380952380952, 'X[14] <= -0.122\nentropy =
0.032\nsamples = 1814\nvalue = [1808, 6]'),
 Text(0.7890295358649789, 0.5476190476190477, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(0.8059071729957806, 0.5476190476190477, 'X[17] <= 0.765\nentropy =
0.027\nsamples = 1813\nvalue = [1808, 5]'),
 Text(0.7805907172995781, 0.5, 'X[0] <= -0.137\nentropy = 0.15\nsamples =
93\nvalue = [91, 2]'),
 Text(0.7721518987341772, 0.4523809523809524, 'entropy = 0.0\nsamples =
74\nvalue = [74, 0]'),
 Text(0.7890295358649789, 0.4523809523809524, 'X[0] <= 0.207\nentropy =
0.485\nsamples = 19\nvalue = [17, 2]'),
 Text(0.7805907172995781, 0.40476190476190477, 'X[14] <= 0.375\nentropy =
1.0\nsamples = 4\nvalue = [2, 2]'),
 Text(0.7721518987341772, 0.35714285714285715, 'entropy = 0.0\nsamples =
2\nvalue = [0, 2]'),
 Text(0.7890295358649789, 0.35714285714285715, 'entropy = 0.0\nsamples =
2\nvalue = [2, 0]'),
 Text(0.7974683544303798, 0.40476190476190477, 'entropy = 0.0\nsamples =
```

```
15\nvalue = [15, 0]'),
 Text(0.8312236286919831, 0.5, 'X[1] <= -1.144\nentropy = 0.018\nsamples =
1720\nvalue = [1717, 3]'),
 Text(0.8227848101265823, 0.4523809523809524, 'X[1] <= -1.676\nentropy =
0.063\nsamples = 409\nvalue = [406, 3]'),
 Text(0.8143459915611815, 0.40476190476190477, 'entropy = 0.0\nsamples =
357\nvalue = [357, 0]'),
 Text(0.8312236286919831, 0.40476190476190477, 'X[4] <= -0.165\nentropy =
0.318\nsamples = 52\nvalue = [49, 3]'),
 Text(0.8227848101265823, 0.35714285714285715, 'entropy = 0.0\nsamples =
39\nvalue = [39, 0]'),
 Text(0.8396624472573839, 0.35714285714285715, 'X[6] <= 1.405\nentropy =
0.779\nsamples = 13\nvalue = [10, 3]'),
 Text(0.8312236286919831, 0.30952380952380953, 'X[15] <= -0.01\nentropy =
0.439\nsamples = 11\nvalue = [10, 1]'),
 Text(0.8227848101265823, 0.2619047619047619, 'entropy = 0.0\nsamples =
10\nvalue = [10, 0]'),
 Text(0.8396624472573839, 0.2619047619047619, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(0.8481012658227848, 0.30952380952380953, 'entropy = 0.0\nsamples =
2\nvalue = [0, 2]'),
 Text(0.8396624472573839, 0.4523809523809524, 'entropy = 0.0\nsamples =
1311\nvalue = [1311, 0]'),
 Text(0.8860759493670886, 0.6428571428571429, 'X[4] <= 1.408\nentropy =
0.073\nsamples = 684\nvalue = [678, 6]'),
 Text(0.8649789029535865, 0.5952380952380952, 'X[17] <= 8.91\nentropy =
0.881\nsamples = 10\nvalue = [7, 3]'),
 Text(0.8565400843881856, 0.5476190476190477, 'X[2] <= 1.552\nentropy =
0.811\nsamples = 4\nvalue = [1, 3]'),
 Text(0.8481012658227848, 0.5, 'entropy = 0.0\nsamples = 3\nvalue = [0, 3]'),
 Text(0.8649789029535865, 0.5, 'entropy = 0.0\nsamples = 1\nvalue = [1, 0]'),
 Text(0.8734177215189873, 0.5476190476190477, 'entropy = 0.0\nsamples = 6\nvalue
= [6, 0]'),
 Text(0.9071729957805907, 0.5952380952380952, 'X[12] <= -0.729\nentropy =
0.041\nsamples = 674\nvalue = [671, 3]'),
 Text(0.890295358649789, 0.5476190476190477, 'X[12] <= -0.736\nentropy =
0.286\nsamples = 40\nvalue = [38, 2]'),
 Text(0.8818565400843882, 0.5, 'X[3] <= 1.727\nentropy = 0.172\nsamples =
39\nvalue = [38, 1]'),
 Text(0.8734177215189873, 0.4523809523809524, 'X[1] <= -2.439\nentropy =
1.0\nsamples = 2\nvalue = [1, 1]'),
 Text(0.8649789029535865, 0.40476190476190477, 'entropy = 0.0\nsamples =
1\nvalue = [0, 1]'),
 Text(0.8818565400843882, 0.40476190476190477, 'entropy = 0.0\nsamples =
1\nvalue = [1, 0]'),
 Text(0.890295358649789, 0.4523809523809524, 'entropy = 0.0\nsamples = 37\nvalue
= [37, 0]'),
```

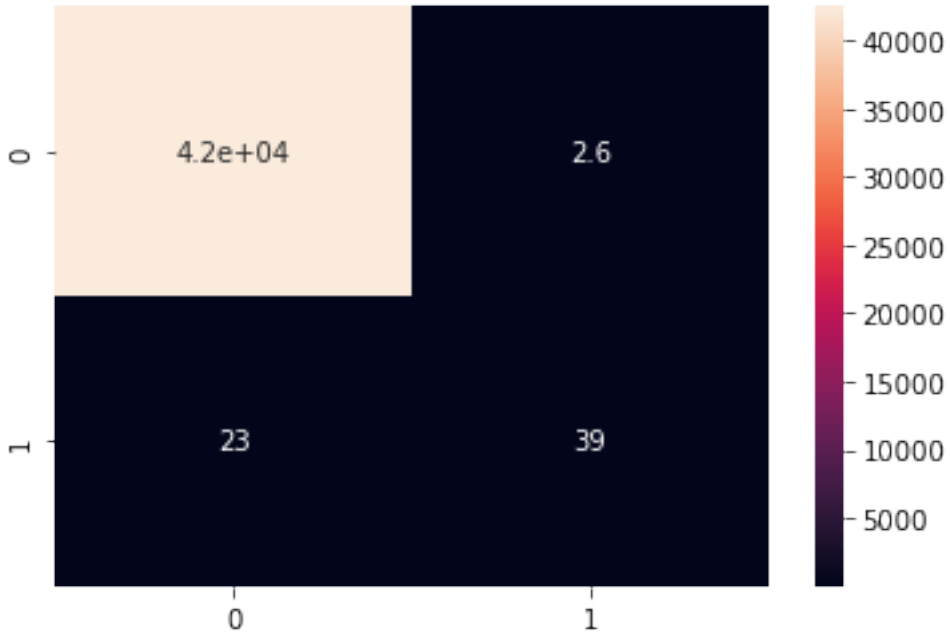
```
 Text(0.8987341772151899, 0.5, 'entropy = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(0.9240506329113924, 0.5476190476190477, 'X[17] <= 0.005\nentropy =
0.017\nsamples = 634\nvalue = [633, 1]'),
 Text(0.9156118143459916, 0.5, 'X[7] <= -0.188\nentropy = 0.276\nsamples =
21\nvalue = [20, 1]'),
 Text(0.9071729957805907, 0.4523809523809524, 'entropy = 0.0\nsamples =
19\nvalue = [19, 0]'),
 Text(0.9240506329113924, 0.4523809523809524, 'X[14] <= -1.049\nentropy =
1.0\nsamples = 2\nvalue = [1, 1]'),
 Text(0.9156118143459916, 0.40476190476190477, 'entropy = 0.0\nsamples =
1\nvalue = [1, 0]'),
 Text(0.9324894514767933, 0.40476190476190477, 'entropy = 0.0\nsamples =
1\nvalue = [0, 1]'),
 Text(0.9324894514767933, 0.5, 'entropy = 0.0\nsamples = 613\nvalue = [613,
0]'),
 Text(0.8544303797468354, 0.6904761904761905, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(0.9662447257383966, 0.7380952380952381, 'X[17] <= 3.935\nentropy =
0.003\nsamples = 13116\nvalue = [13113, 3]'),
 Text(0.9578059071729957, 0.6904761904761905, 'X[11] <= -2.542\nentropy =
0.015\nsamples = 2201\nvalue = [2198, 3]'),
 Text(0.9409282700421941, 0.6428571428571429, 'X[17] <= 3.785\nentropy =
0.156\nsamples = 88\nvalue = [86, 2]'),
 Text(0.9324894514767933, 0.5952380952380952, 'entropy = 0.0\nsamples =
77\nvalue = [77, 0]'),
 Text(0.9493670886075949, 0.5952380952380952, 'X[5] <= -0.002\nentropy =
0.684\nsamples = 11\nvalue = [9, 2]'),
 Text(0.9409282700421941, 0.5476190476190477, 'entropy = 0.0\nsamples = 2\nvalue
= [0, 2]'),
 Text(0.9578059071729957, 0.5476190476190477, 'entropy = 0.0\nsamples = 9\nvalue
= [9, 0]'),
 Text(0.9746835443037974, 0.6428571428571429, 'X[11] <= 1.237\nentropy =
0.006\nsamples = 2113\nvalue = [2112, 1]'),
 Text(0.9662447257383966, 0.5952380952380952, 'entropy = 0.0\nsamples =
2045\nvalue = [2045, 0]'),
 Text(0.9831223628691983, 0.5952380952380952, 'X[11] <= 1.243\nentropy =
0.111\nsamples = 68\nvalue = [67, 1]'),
 Text(0.9746835443037974, 0.5476190476190477, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(0.9915611814345991, 0.5476190476190477, 'entropy = 0.0\nsamples =
67\nvalue = [67, 0]'),
 Text(0.9746835443037974, 0.6904761904761905, 'entropy = 0.0\nsamples =
10915\nvalue = [10915, 0]')]
```

[7]:
```
from sklearn import neighbors
k = 5
accuracy_nb_neighbour = np.zeros(k)
accuracy_nb_neighbour_new = np.zeros(k)
for i in range(3,k):
    print("For ", i, "neighbours")
    knc = neighbors.KNeighborsClassifier(i, weights='distance')
    accuracy_nb_neighbour_new = error_comp_new(knc)
accuracy_nb_neighbour_new
```

```
For  3 neighbours
Accuracy selected features 0.9994031814657299
Confusion matrix for all features
Percentage of correct fraud detected 0.6566666666666666
```

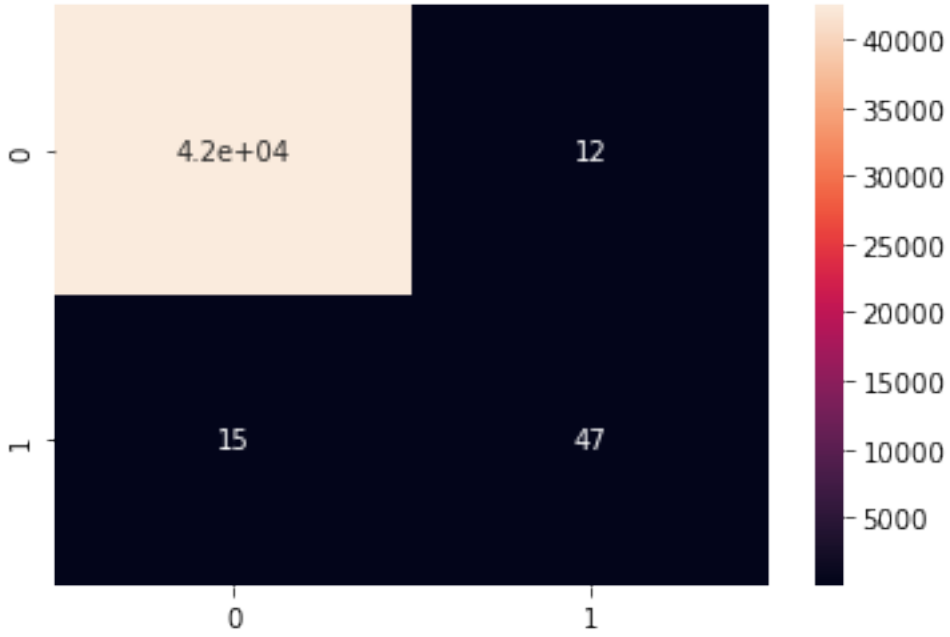


For 4 neighbours
Accuracy selected features 0.9993937827486548
Confusion matrix for all features
Percentage of correct fraud detected 0.65



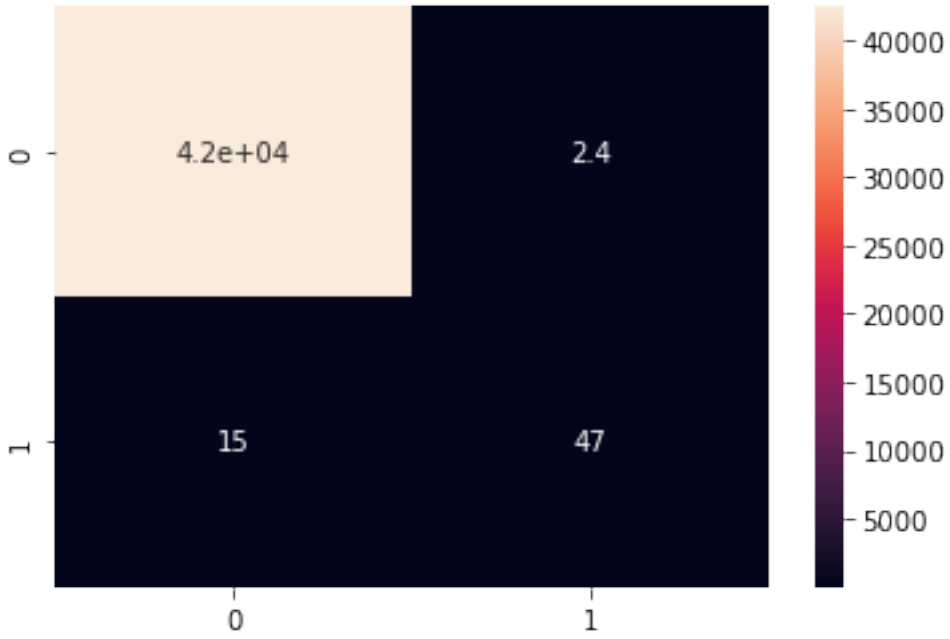
[8]:
```
from sklearn.neural_network import MLPClassifier

activation_list =['logistic']#['identity', 'logistic', 'tanh', 'relu']
for activation in activation_list:
    print("Activation function: ", activation)
    mlp = MLPClassifier(activation = activation, random_state=1, max_iter=300)
    error_comp_new(mlp)
```

```
Activation function:  logistic
Accuracy selected features 0.999360887238892
Confusion matrix for all features
Percentage of correct fraud detected 0.78
```



[9]:
```
from sklearn.ensemble import RandomForestClassifier
criterion_list = ['gini']
for criterion in criterion_list:
    print("Criterion : ", criterion)
    rfc = RandomForestClassifier(criterion = criterion)
    error_comp_new(rfc)
```

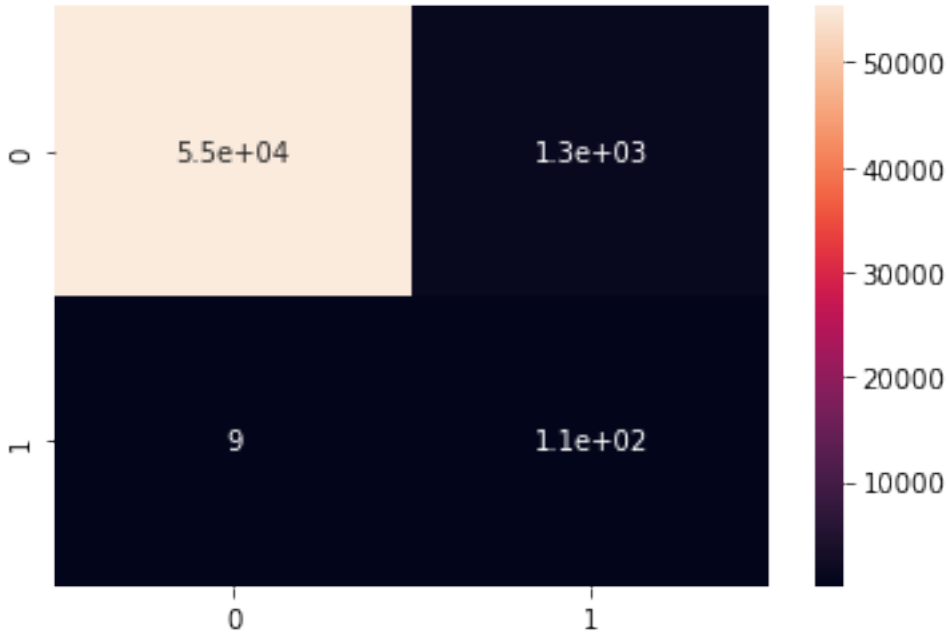
```
Criterion :  gini
Accuracy selected features 0.9995958551657699
Confusion matrix for all features
Percentage of correct fraud detected 0.79
```



[26]:
```
# To handle class inbalanced data
from imblearn import under_sampling, over_sampling
from imblearn.over_sampling import SMOTE

from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, accuracy_score, confusion_matrix

from sklearn.linear_model import SGDClassifier
from sklearn.tree import DecisionTreeClassifier

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2)

X_spld, Y_spld = SMOTE().fit_resample(X_train, y_train) #use as the sampling is␣
 ↪really poor


sdg = SGDClassifier(loss='log', penalty="l2")
sdg.fit(X_spld, np.ravel(Y_spld))

dtc = DecisionTreeClassifier(criterion = 'entropy')
dtc.fit(X_spld, np.ravel(Y_spld))

print(y_test.value_counts())

print(accuracy_score(np.ravel(y_test),sdg.predict(X_test)))
```

```
matrix_con = confusion_matrix(np.ravel(y_test), sdg.predict(X_test))
sns.heatmap(matrix_con, annot=True)

print('Percentage of correct fraud detected', matrix_con[1,1]/(y_test.
 ↪value_counts()[1]))

#print(accuracy_score(np.ravel(y_test),dtc.predict(X_test)))
#sns.heatmap(confusion_matrix(np.ravel(y_test), dtc.predict(X_test)),␣
 ↪annot=True)
```

```
Class
0        56628
1          118
dtype: int64
0.9777429246114263
Percentage of correct fraud detected 0.923728813559322
```



[32]:

[32]: 91

[ ]: