BankNote_Authentication

November 20, 2022

[ ]:
```
!jupyter nbconvert --to pdf
```

Principles of Data Science

Submitted By:

Name: **Prateek Kumar**

Register Number: **2248013**

Class: **1 MSc Data Science**

Overview

The Banknote Dataset involves predicting whether a given banknote is authentic given a number of measures taken from a photograph.

It is a binary (2-class) classification problem. The number of observations for each class is not balanced. There are 1,372 observations with 4 input variables and 1 output variable.

The variable names are as follows:Variance of Wavelet Transformed image (continuous).Skewness of Wavelet Transformed image (continuous).Kurtosis of Wavelet Transformed image (continuous).Entropy of image (continuous).Class (0 for authentic, 1 for inauthentic).

About this Data:

Data were extracted from images that were taken from genuine and forged banknote-like specimens. For digitization, an industrial camera usually used for print inspection was used. The final images have 400x 400 pixels. Due to the object lens and distance to the investigated object gray-scale pictures with a resolution of about 660 dpi were gained. Wavelet Transform tool were used to extract features from images.

Attribute Information:

- variance of Wavelet Transformed image (continuous)
- skewness of Wavelet Transformed image (continuous)
- skewness of Wavelet Transformed image (continuous)
- curtosis of Wavelet Transformed image (continuous)
- entropy of image (continuous)
- class (integer)

Problem Definition

- Understand the Dataset & Features
- Perform Data Preprocessing Technique to Get Balanced Structured Data

- Perform Statistical Data Analysis and Derive Valuable Inferences
- Perform Exploratory Data Analysis and Derive Valuable Insights
- Perform Regression and Classification via. different Models

Approach

This is an extension to the Problem Defnintion. Mention the process/appraoch that you have followed in order to reach out the above problem defintion.

Step 1: Know the dataset thoroughly.

Step 2: Perform preprocessing on data.

Step 3: Import needfull libraries as an when you try to plot different graphs and evaluate the model.

Step 4: Perform Statistical Data Analysis and Derive Valuable Inferences.

Step 5: Perform Exploratory Data Analysis and Derive Valuable Insights.

Step 6: Perform Regression and Classification via. different Models

Sections Here, mentioned sections are defined in the below code. For this lab, the sections are - 1. Lab Overview 2. Dataset Overview 3. Data Analyst Process 4. Implementation and Evaluation of Regression and Classification Models. 5. Conclusion

References 1. https://pandas.pydata.org/ 2. https://matplotlib.org/ 3. https://seaborn.pydata.org/ 4. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticR
5. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
6. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
7. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.manhattan_distances.html
8. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html#:~:text=%27weighted%27%2

Part A : Pre Processing and Exploratory Data Analysis

Importing Neccessary Libraries

[1]:
```python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from sklearn import metrics
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings("ignore")
```

Uploading the Dataset in Google Colab

[2]:
```python
from google.colab import files
uploaded = files.upload()
```

```
<IPython.core.display.HTML object>
```

```
Saving banknote.txt to banknote.txt
```

Reading the text file dataset

[3]:
```
B = np.loadtxt("banknote.txt", delimiter=',')
Total_data=len(B)
C = B[:,0:4]
Variance = C[:,0]
Skewness = C[:,1]
Curtosis = C[:,2]
Entropy = C[:,3]
Class = B[:,4]
```

Creating the Dataframe

[4]:
```
BN = pd.DataFrame(B,␣
↪columns=['Variance','Skewness','Curtosis','Entropy','Class'])
```

[5]:
```
BN
```

[5]:
```
      Variance  Skewness  Curtosis  Entropy  Class
0      3.62160   8.66610   -2.8073 -0.44699    0.0
1      4.54590   8.16740   -2.4586 -1.46210    0.0
2      3.86600  -2.63830    1.9242  0.10645    0.0
3      3.45660   9.52280   -4.0112 -3.59440    0.0
4      0.32924  -4.45520    4.5718 -0.98880    0.0
…          …         …        …        …      …
1367   0.40614   1.34920   -1.4501 -0.55949    1.0
1368  -1.38870  -4.87730    6.4774  0.34179    1.0
1369  -3.75030 -13.45860   17.5932 -2.77710    1.0
1370  -3.56370  -8.38270   12.3930 -1.28230    1.0
1371  -2.54190  -0.65804    2.6842  1.19520    1.0

[1372 rows x 5 columns]
```

Using Shape function to check rows and columns

[8]:
```
BN.shape
```

[8]:
```
(1372, 5)
```

Columns function to read column index

[9]:
```
BN.columns
```

[9]:
```
Index(['Variance', 'Skewness', 'Curtosis', 'Entropy', 'Class'], dtype='object')
```

Read first 5 rows defaults

[10]:
```
BN.head()
```

[10]:

	Variance	Skewness	Curtosis	Entropy	Class
0	3.62160	8.6661	-2.8073	-0.44699	0.0
1	4.54590	8.1674	-2.4586	-1.46210	0.0
2	3.86600	-2.6383	1.9242	0.10645	0.0
3	3.45660	9.5228	-4.0112	-3.59440	0.0
4	0.32924	-4.4552	4.5718	-0.98880	0.0

Read last 5 rows default

[11]:
```
BN.tail()
```

[11]:

	Variance	Skewness	Curtosis	Entropy	Class
1367	0.40614	1.34920	-1.4501	-0.55949	1.0
1368	-1.38870	-4.87730	6.4774	0.34179	1.0
1369	-3.75030	-13.45860	17.5932	-2.77710	1.0
1370	-3.56370	-8.38270	12.3930	-1.28230	1.0
1371	-2.54190	-0.65804	2.6842	1.19520	1.0

Checking the datatypes

[12]:
```
BN.dtypes
```

[12]:
```
Variance    float64
Skewness    float64
Curtosis    float64
Entropy     float64
Class       float64
dtype: object
```

Checking info of dataframe

[13]:
```
BN.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1372 entries, 0 to 1371
Data columns (total 5 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Variance  1372 non-null   float64
 1   Skewness  1372 non-null   float64
 2   Curtosis  1372 non-null   float64
 3   Entropy   1372 non-null   float64
 4   Class     1372 non-null   float64
dtypes: float64(5)
memory usage: 53.7 KB
```

Statistical summary of data.

[14]:
```
BN.describe()
```

[14]:

	Variance	Skewness	Curtosis	Entropy	Class
count	1372.000000	1372.000000	1372.000000	1372.000000	1372.000000
mean	0.433735	1.922353	1.397627	-1.191657	0.444606
std	2.842763	5.869047	4.310030	2.101013	0.497103
min	-7.042100	-13.773100	-5.286100	-8.548200	0.000000
25%	-1.773000	-1.708200	-1.574975	-2.413450	0.000000
50%	0.496180	2.319650	0.616630	-0.586650	0.000000
75%	2.821475	6.814625	3.179250	0.394810	1.000000
max	6.824800	12.951600	17.927400	2.449500	1.000000

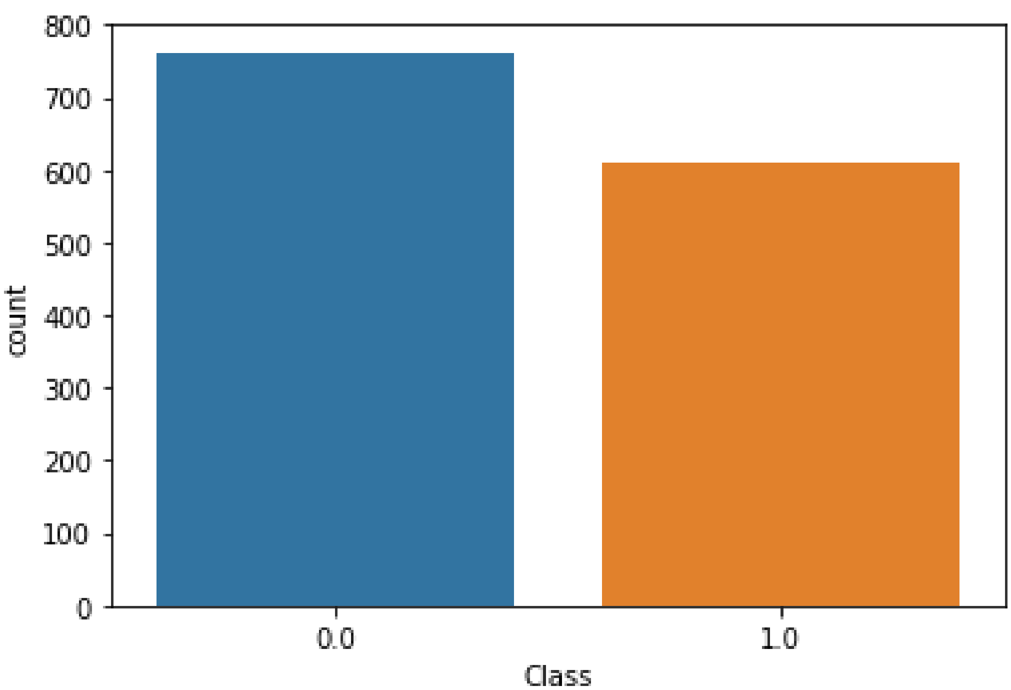
Checking the missing values in data

[15]:
```
BN.isnull().sum()
```

[15]:
```
Variance    0
Skewness    0
Curtosis    0
Entropy     0
Class       0
dtype: int64
```

Seaborn plot to count the class label 0 and 1

[17]:
```
sns.countplot(x=BN["Class"])
plt.show()
```



Correlation function to find the corr of data

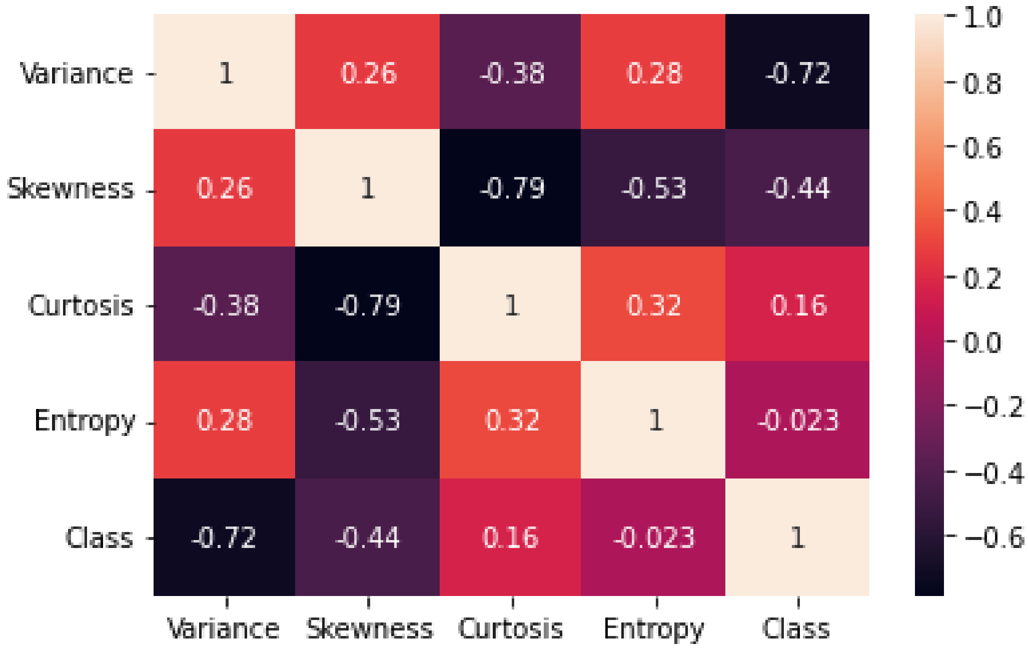
[18]:
```
BN.corr()
```

[18]:

	Variance	Skewness	Curtosis	Entropy	Class
Variance	1.000000	0.264026	-0.380850	0.276817	-0.724843
Skewness	0.264026	1.000000	-0.786895	-0.526321	-0.444688
Curtosis	-0.380850	-0.786895	1.000000	0.318841	0.155883
Entropy	0.276817	-0.526321	0.318841	1.000000	-0.023424
Class	-0.724843	-0.444688	0.155883	-0.023424	1.000000

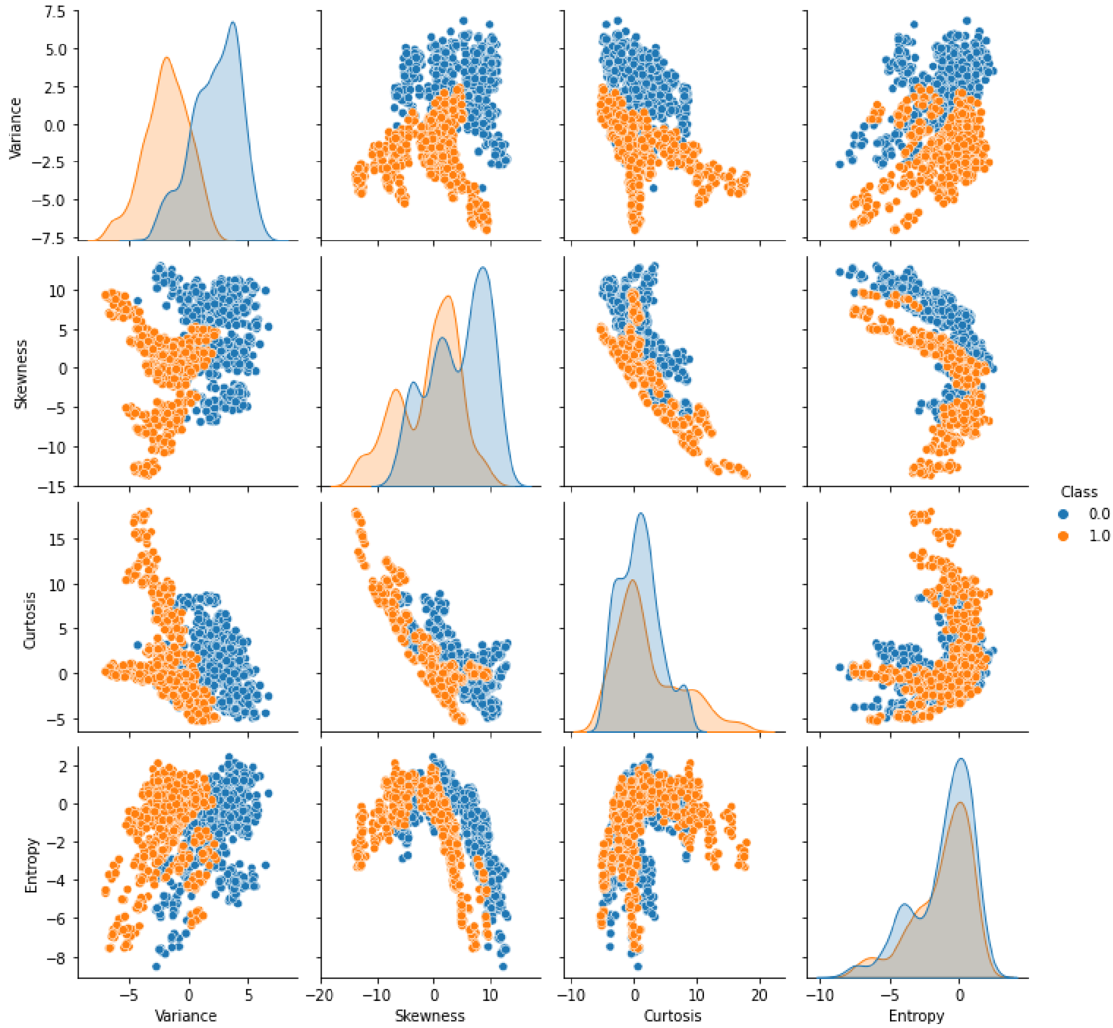
Heatmap of correlation plot

[19]:
```
sns.heatmap(BN.corr(), annot=True)
plt.show()
```



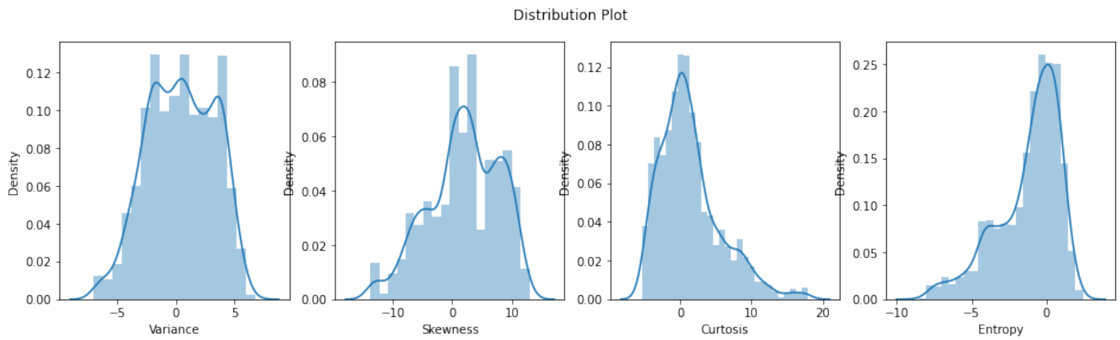
Pairplot of class label

[20]:
```
sns.pairplot(BN, hue = "Class")
plt.show()
```



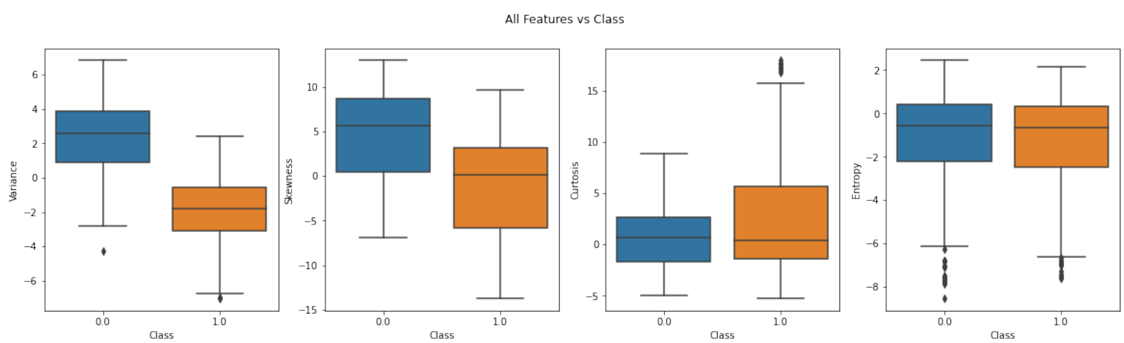
Distribution Plot of Class labels and density

[21]:
```
columns = list(BN.columns)
columns.remove('Class')
fig, ax = plt.subplots(ncols = 4, figsize=(16, 4))
fig.suptitle("Distribution Plot")
for index, column in enumerate(columns):
 sns.distplot(BN[column], ax=ax[index])

plt.show()
```



Subplots of All features vs Class label

[22]:
```python
fig, ax = plt.subplots(ncols=4, figsize=(20, 5))
fig.suptitle("All Features vs Class")
for index, column in enumerate(columns):
 sns.boxplot(x="Class", y=column, data=BN, ax=ax[index])

plt.show()
```



Drop class label

[23]:
```python
Y = BN['Class']
X = BN.drop(['Class'], axis=1)
```

[24]:
```python
X.sample(5)
```

[24]:

	Variance	Skewness	Curtosis	Entropy
437	0.54150	6.03190	1.68250	-0.46122
46	2.08430	6.62580	0.48382	-2.21340
1331	0.22432	-0.52147	-0.40386	1.20170
1365	-4.50460	-5.81260	10.88670	-0.52846
429	2.55030	-4.95180	6.37290	-0.41596

[25]:
```
Y.sample(5)
```

[25]:
```
875     1.0
125     0.0
283     0.0
566     0.0
772     1.0
Name: Class, dtype: float64
```

Train Test Spilt of the Data

[26]:
```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.30,␣
↪random_state = 8)
```

[27]:
```
X_train.sample(5)
```

[27]:
```
      Variance  Skewness  Curtosis  Entropy
266  -0.016103    9.7484   0.15394 -1.61340
112   3.235100    9.6470  -3.20740 -2.59480
793  -2.286000   -5.4484   5.80390  0.88231
1293 -3.969800    3.6812  -0.60008 -4.01330
315   0.329200   -4.4552   4.57180 -0.98880
```

[28]:
```
X_test.sample(5)
```

[28]:
```
      Variance  Skewness  Curtosis  Entropy
466    1.14720    3.5985   1.93870 -0.43406
1243  -5.06760   -5.1877  10.42660 -0.86725
129    3.46630    1.1112   1.74250  1.33880
96     2.95430    1.0760   0.64577  0.89394
810   -0.64326    2.4748  -2.94520 -1.02760
```

[29]:
```
Y_train.sample(5)
```

[29]:
```
172      0.0
1140     1.0
33       0.0
1145     1.0
1254     1.0
Name: Class, dtype: float64
```

[30]:
```
Y_test.sample(5)
```

[30]:
```
589     0.0
337     0.0
844     1.0
282     0.0
```

```
592    0.0
Name: Class, dtype: float64
```

Logistic Regression Prediction

[31]:
```python
from sklearn.linear_model import LogisticRegression
logistic_regressor = LogisticRegression()
logistic_regressor.fit(X_train, Y_train)
```

[31]:
```
LogisticRegression()
```

[32]:
```python
y_pred = logistic_regressor.predict(X_test)
y_prob = logistic_regressor.predict_proba(X_test)
```

[33]:
```python
y_pred[0:5]
y_prob[0:5]
```

[33]:
```
array([[9.99998554e-01, 1.44574492e-06],
       [9.99995849e-01, 4.15114512e-06],
       [9.99999999e-01, 6.40792509e-10],
       [9.99982486e-01, 1.75142303e-05],
       [9.99842008e-01, 1.57992362e-04]])
```

Classification Report

[34]:
```python
from sklearn.metrics import classification_report
print(classification_report(Y_test, y_pred))
```

```
              precision    recall  f1-score   support

         0.0       1.00      0.98      0.99       242
         1.0       0.97      0.99      0.98       170

    accuracy                           0.99       412
   macro avg       0.98      0.99      0.99       412
weighted avg       0.99      0.99      0.99       412
```

Confusion Matrix

[35]:
```python
from sklearn.metrics import confusion_matrix
```

[36]:
```python
CM = confusion_matrix(Y_test, y_pred)
CM
```

[36]:
```
array([[237,   5],
       [  1, 169]])
```

[37]:
```python
Accuracy = (CM[0][0] + CM[1][1]) / (CM[0][0] + CM[1][1] + CM[0][1] + CM[1][0])
Accuracy
```

[37]: 0.9854368932038835

[38]:
```python
ErrorRate = (CM[0][1] + CM[1][0]) / (CM[0][0] + CM[1][1] + CM[0][1] + CM[1][0])
ErrorRate
```

[38]: 0.014563106796116505

[39]:
```python
Sensitivity = CM[0][0]/(CM[0][0] + CM[1][0])
Sensitivity
```

[39]: 0.9957983193277311

[40]:
```python
Specificity = CM[1][1]/(CM[1][1] + CM[0][1])
Specificity
```

[40]: 0.9712643678160919

[41]:
```python
Recall = CM[0][0]/(CM[0][0] + CM[1][0])
Recall
```

[41]: 0.9957983193277311

[42]:
```python
Precision = CM[0][0]/(CM[0][0] + CM[0][1])
Precision
```

[42]: 0.9793388429752066

[43]:
```python
F1Score = (2*(Precision*Recall))/(Precision + Recall)
F1Score
```

[43]: 0.9875

Evaluate The Effect of Parameters

[46]:
```python
from sklearn.metrics import accuracy_score
#BN = pd.read_csv("cat_1_a_dataset.csv")
y = BN['Class']
x = BN.drop(['Class'], axis=1)
```

[47]:
```python
def doLogisticRegression(x, y, test_size = 0.20, random_state = 42,␣
↪penalty='l2', solver='lbfgs'):
 x_train, x_test, y_train, y_test = train_test_split(x, y, test_size =␣
↪test_size, random_state = random_state)

 logistic_regressor = LogisticRegression(penalty = penalty, solver = solver)
```

```
    logistic_regressor.fit(x_train, y_train)
    y_pred = logistic_regressor.predict(x_test)

    acc_score = accuracy_score(y_test, y_pred)

    return acc_score
```

[49]:
```
penalties = ['none', 'l2']
test_size = [0.30, 0.25, 0.20]
random_states = [21, 42, 84]
solvers = ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']
for t_size in test_size:
  for r_state in random_states:
    for penalty in penalties:
      for solver in solvers:
        accuracy = doLogisticRegression(x, y, t_size, r_state, penalty)
        print("Test: {} | Random State: {} | Penalty: {} | Solver: {} |␣
 ↪Accuracy : {}".format(t_size, r_state, penalty, solver, accuracy))
```

```
Test: 0.3 | Random State: 21 | Penalty: none | Solver: newton-cg | Accuracy :
0.9878640776699029
Test: 0.3 | Random State: 21 | Penalty: none | Solver: lbfgs | Accuracy :
0.9878640776699029
Test: 0.3 | Random State: 21 | Penalty: none | Solver: liblinear | Accuracy :
0.9878640776699029
Test: 0.3 | Random State: 21 | Penalty: none | Solver: sag | Accuracy :
0.9878640776699029
Test: 0.3 | Random State: 21 | Penalty: none | Solver: saga | Accuracy :
0.9878640776699029
Test: 0.3 | Random State: 21 | Penalty: l2 | Solver: newton-cg | Accuracy :
0.9902912621359223
Test: 0.3 | Random State: 21 | Penalty: l2 | Solver: lbfgs | Accuracy :
0.9902912621359223
Test: 0.3 | Random State: 21 | Penalty: l2 | Solver: liblinear | Accuracy :
0.9902912621359223
Test: 0.3 | Random State: 21 | Penalty: l2 | Solver: sag | Accuracy :
0.9902912621359223
Test: 0.3 | Random State: 21 | Penalty: l2 | Solver: saga | Accuracy :
0.9902912621359223
Test: 0.3 | Random State: 42 | Penalty: none | Solver: newton-cg | Accuracy :
0.9902912621359223
Test: 0.3 | Random State: 42 | Penalty: none | Solver: lbfgs | Accuracy :
0.9902912621359223
Test: 0.3 | Random State: 42 | Penalty: none | Solver: liblinear | Accuracy :
0.9902912621359223
Test: 0.3 | Random State: 42 | Penalty: none | Solver: sag | Accuracy :
0.9902912621359223
```

```
Test: 0.3 | Random State: 42 | Penalty: none | Solver: saga | Accuracy :
0.9902912621359223
Test: 0.3 | Random State: 42 | Penalty: l2 | Solver: newton-cg | Accuracy :
0.9878640776699029
Test: 0.3 | Random State: 42 | Penalty: l2 | Solver: lbfgs | Accuracy :
0.9878640776699029
Test: 0.3 | Random State: 42 | Penalty: l2 | Solver: liblinear | Accuracy :
0.9878640776699029
Test: 0.3 | Random State: 42 | Penalty: l2 | Solver: sag | Accuracy :
0.9878640776699029
Test: 0.3 | Random State: 42 | Penalty: l2 | Solver: saga | Accuracy :
0.9878640776699029
Test: 0.3 | Random State: 84 | Penalty: none | Solver: newton-cg | Accuracy :
0.9830097087378641
Test: 0.3 | Random State: 84 | Penalty: none | Solver: lbfgs | Accuracy :
0.9830097087378641
Test: 0.3 | Random State: 84 | Penalty: none | Solver: liblinear | Accuracy :
0.9830097087378641
Test: 0.3 | Random State: 84 | Penalty: none | Solver: sag | Accuracy :
0.9830097087378641
Test: 0.3 | Random State: 84 | Penalty: none | Solver: saga | Accuracy :
0.9830097087378641
Test: 0.3 | Random State: 84 | Penalty: l2 | Solver: newton-cg | Accuracy :
0.9830097087378641
Test: 0.3 | Random State: 84 | Penalty: l2 | Solver: lbfgs | Accuracy :
0.9830097087378641
Test: 0.3 | Random State: 84 | Penalty: l2 | Solver: liblinear | Accuracy :
0.9830097087378641
Test: 0.3 | Random State: 84 | Penalty: l2 | Solver: sag | Accuracy :
0.9830097087378641
Test: 0.3 | Random State: 84 | Penalty: l2 | Solver: saga | Accuracy :
0.9830097087378641
Test: 0.25 | Random State: 21 | Penalty: none | Solver: newton-cg | Accuracy :
0.9854227405247813
Test: 0.25 | Random State: 21 | Penalty: none | Solver: lbfgs | Accuracy :
0.9854227405247813
Test: 0.25 | Random State: 21 | Penalty: none | Solver: liblinear | Accuracy :
0.9854227405247813
Test: 0.25 | Random State: 21 | Penalty: none | Solver: sag | Accuracy :
0.9854227405247813
Test: 0.25 | Random State: 21 | Penalty: none | Solver: saga | Accuracy :
0.9854227405247813
Test: 0.25 | Random State: 21 | Penalty: l2 | Solver: newton-cg | Accuracy :
0.9883381924198251
Test: 0.25 | Random State: 21 | Penalty: l2 | Solver: lbfgs | Accuracy :
0.9883381924198251
Test: 0.25 | Random State: 21 | Penalty: l2 | Solver: liblinear | Accuracy :
0.9883381924198251
```

Test: 0.25 | Random State: 21 | Penalty: l2 | Solver: sag | Accuracy :
0.9883381924198251
Test: 0.25 | Random State: 21 | Penalty: l2 | Solver: saga | Accuracy :
0.9883381924198251
Test: 0.25 | Random State: 42 | Penalty: none | Solver: newton-cg | Accuracy :
0.9883381924198251
Test: 0.25 | Random State: 42 | Penalty: none | Solver: lbfgs | Accuracy :
0.9883381924198251
Test: 0.25 | Random State: 42 | Penalty: none | Solver: liblinear | Accuracy :
0.9883381924198251
Test: 0.25 | Random State: 42 | Penalty: none | Solver: sag | Accuracy :
0.9883381924198251
Test: 0.25 | Random State: 42 | Penalty: none | Solver: saga | Accuracy :
0.9883381924198251
Test: 0.25 | Random State: 42 | Penalty: l2 | Solver: newton-cg | Accuracy :
0.9883381924198251
Test: 0.25 | Random State: 42 | Penalty: l2 | Solver: lbfgs | Accuracy :
0.9883381924198251
Test: 0.25 | Random State: 42 | Penalty: l2 | Solver: liblinear | Accuracy :
0.9883381924198251
Test: 0.25 | Random State: 42 | Penalty: l2 | Solver: sag | Accuracy :
0.9883381924198251
Test: 0.25 | Random State: 42 | Penalty: l2 | Solver: saga | Accuracy :
0.9883381924198251
Test: 0.25 | Random State: 84 | Penalty: none | Solver: newton-cg | Accuracy :
0.9825072886297376
Test: 0.25 | Random State: 84 | Penalty: none | Solver: lbfgs | Accuracy :
0.9825072886297376
Test: 0.25 | Random State: 84 | Penalty: none | Solver: liblinear | Accuracy :
0.9825072886297376
Test: 0.25 | Random State: 84 | Penalty: none | Solver: sag | Accuracy :
0.9825072886297376
Test: 0.25 | Random State: 84 | Penalty: none | Solver: saga | Accuracy :
0.9825072886297376
Test: 0.25 | Random State: 84 | Penalty: l2 | Solver: newton-cg | Accuracy :
0.9854227405247813
Test: 0.25 | Random State: 84 | Penalty: l2 | Solver: lbfgs | Accuracy :
0.9854227405247813
Test: 0.25 | Random State: 84 | Penalty: l2 | Solver: liblinear | Accuracy :
0.9854227405247813
Test: 0.25 | Random State: 84 | Penalty: l2 | Solver: sag | Accuracy :
0.9854227405247813
Test: 0.25 | Random State: 84 | Penalty: l2 | Solver: saga | Accuracy :
0.9854227405247813
Test: 0.2 | Random State: 21 | Penalty: none | Solver: newton-cg | Accuracy :
0.9818181818181818
Test: 0.2 | Random State: 21 | Penalty: none | Solver: lbfgs | Accuracy :
0.9818181818181818

```
Test: 0.2 | Random State: 21 | Penalty: none | Solver: liblinear | Accuracy :
0.9818181818181818
Test: 0.2 | Random State: 21 | Penalty: none | Solver: sag | Accuracy :
0.9818181818181818
Test: 0.2 | Random State: 21 | Penalty: none | Solver: saga | Accuracy :
0.9818181818181818
Test: 0.2 | Random State: 21 | Penalty: l2 | Solver: newton-cg | Accuracy :
0.9854545454545455
Test: 0.2 | Random State: 21 | Penalty: l2 | Solver: lbfgs | Accuracy :
0.9854545454545455
Test: 0.2 | Random State: 21 | Penalty: l2 | Solver: liblinear | Accuracy :
0.9854545454545455
Test: 0.2 | Random State: 21 | Penalty: l2 | Solver: sag | Accuracy :
0.9854545454545455
Test: 0.2 | Random State: 21 | Penalty: l2 | Solver: saga | Accuracy :
0.9854545454545455
Test: 0.2 | Random State: 42 | Penalty: none | Solver: newton-cg | Accuracy :
0.9854545454545455
Test: 0.2 | Random State: 42 | Penalty: none | Solver: lbfgs | Accuracy :
0.9854545454545455
Test: 0.2 | Random State: 42 | Penalty: none | Solver: liblinear | Accuracy :
0.9854545454545455
Test: 0.2 | Random State: 42 | Penalty: none | Solver: sag | Accuracy :
0.9854545454545455
Test: 0.2 | Random State: 42 | Penalty: none | Solver: saga | Accuracy :
0.9854545454545455
Test: 0.2 | Random State: 42 | Penalty: l2 | Solver: newton-cg | Accuracy :
0.9854545454545455
Test: 0.2 | Random State: 42 | Penalty: l2 | Solver: lbfgs | Accuracy :
0.9854545454545455
Test: 0.2 | Random State: 42 | Penalty: l2 | Solver: liblinear | Accuracy :
0.9854545454545455
Test: 0.2 | Random State: 42 | Penalty: l2 | Solver: sag | Accuracy :
0.9854545454545455
Test: 0.2 | Random State: 42 | Penalty: l2 | Solver: saga | Accuracy :
0.9854545454545455
Test: 0.2 | Random State: 84 | Penalty: none | Solver: newton-cg | Accuracy :
0.9818181818181818
Test: 0.2 | Random State: 84 | Penalty: none | Solver: lbfgs | Accuracy :
0.9818181818181818
Test: 0.2 | Random State: 84 | Penalty: none | Solver: liblinear | Accuracy :
0.9818181818181818
Test: 0.2 | Random State: 84 | Penalty: none | Solver: sag | Accuracy :
0.9818181818181818
Test: 0.2 | Random State: 84 | Penalty: none | Solver: saga | Accuracy :
0.9818181818181818
Test: 0.2 | Random State: 84 | Penalty: l2 | Solver: newton-cg | Accuracy :
0.9818181818181818
```

```
Test: 0.2 | Random State: 84 | Penalty: l2 | Solver: lbfgs | Accuracy :
0.9818181818181818
Test: 0.2 | Random State: 84 | Penalty: l2 | Solver: liblinear | Accuracy :
0.9818181818181818
Test: 0.2 | Random State: 84 | Penalty: l2 | Solver: sag | Accuracy :
0.9818181818181818
Test: 0.2 | Random State: 84 | Penalty: l2 | Solver: saga | Accuracy :
0.9818181818181818
```

[50]:
```
BN1 = pd.DataFrame(columns = ['Test Size', 'Random States', 'Penalty',␣
↪'Solvers', 'Accuracy'])
BN1
```

[50]:
```
Empty DataFrame
Columns: [Test Size, Random States, Penalty, Solvers, Accuracy]
Index: []
```

[53]:
```
penalties = ['none', 'l2']
test_size = [0.30, 0.25, 0.20]
random_states = [21, 42, 84]
solvers = ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']
for t_size in test_size:
  for r_state in random_states:
    for penalty in penalties:
      for solver in solvers:
        accuracy = doLogisticRegression(x, y, t_size, r_state, penalty)

        BankNoteAuthentication = {}
        BankNoteAuthentication['Test Size'] = t_size
        BankNoteAuthentication['Random States'] = r_state
        BankNoteAuthentication['Penalty'] = penalty
        BankNoteAuthentication['Solvers'] = solver
        BankNoteAuthentication['Accuracy'] = accuracy

        BN1 = BN1.append(BankNoteAuthentication, ignore_index = True)
```

[54]:
```
BN1.head()
```

[54]:

	Test Size	Random States	Penalty	Solvers	Accuracy
0	0.3	21	none	newton-cg	0.987864
1	0.3	21	none	lbfgs	0.987864
2	0.3	21	none	liblinear	0.987864
3	0.3	21	none	sag	0.987864
4	0.3	21	none	saga	0.987864

[55]:
```
BN1.tail()
```

[55]:
```
    Test Size Random States Penalty    Solvers  Accuracy
85        0.2             84      l2  newton-cg  0.981818
86        0.2             84      l2      lbfgs  0.981818
87        0.2             84      l2  liblinear  0.981818
88        0.2             84      l2        sag  0.981818
89        0.2             84      l2       saga  0.981818
```

KNN ALGORITHM

[56]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics
from collections import OrderedDict
```

[57]:
```
A = np.loadtxt("banknote.txt", delimiter=',')
Total_data=len(A)
X = A[:,0:4]
Variance = X[:,0]
Skewness = X[:,1]
Curtosis = X[:,2]
Entropy = X[:,3]
Class = A[:,4]
```

[64]:
```
df = pd.
 ↪DataFrame(A,columns=['Variance','Skewness','Curtosis','Entropy','Class'])
df
```

[64]:
```
      Variance  Skewness  Curtosis  Entropy  Class
0      3.62160   8.66610   -2.8073 -0.44699    0.0
1      4.54590   8.16740   -2.4586 -1.46210    0.0
2      3.86600  -2.63830    1.9242  0.10645    0.0
3      3.45660   9.52280   -4.0112 -3.59440    0.0
4      0.32924  -4.45520    4.5718 -0.98880    0.0
…          …         …       …        …      …
1367   0.40614   1.34920   -1.4501 -0.55949    1.0
1368  -1.38870  -4.87730    6.4774  0.34179    1.0
1369  -3.75030 -13.45860   17.5932 -2.77710    1.0
1370  -3.56370  -8.38270   12.3930 -1.28230    1.0
1371  -2.54190  -0.65804    2.6842  1.19520    1.0

[1372 rows x 5 columns]
```

[65]:
```
def splitData(df, headSize):
  """
  This function splits the data based on the head size .
  """
  hd = df.head(headSize)
  tl = df.tail(len(df)-headSize)
```

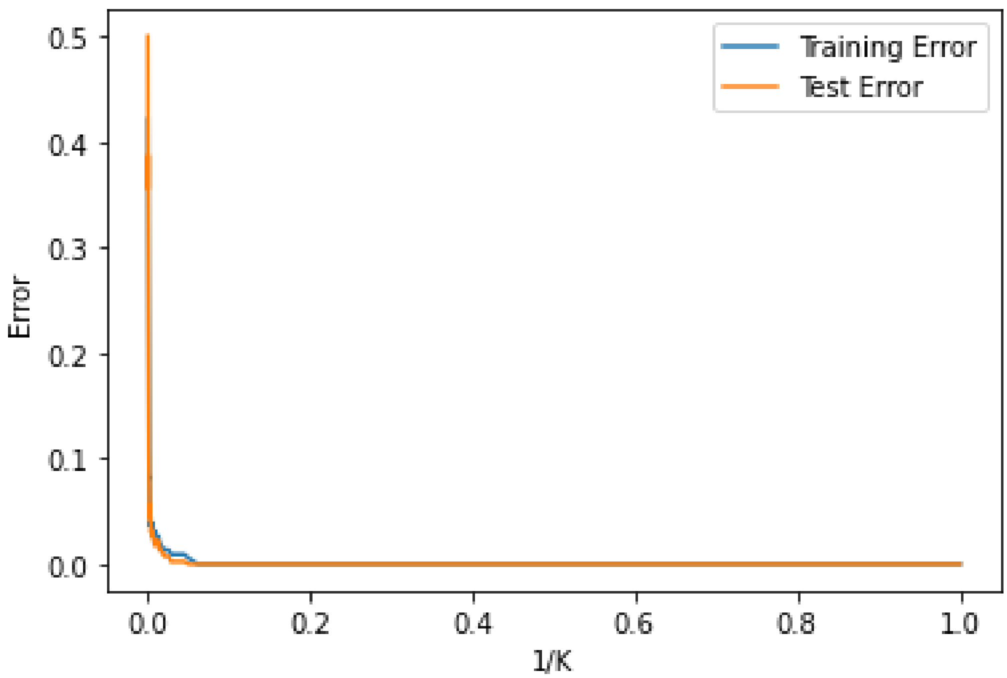
```
    return hd, tl

def getData(a,b):
    """
    This function combines 2 dataframes.
    """
    x = pd.concat([a, b], sort=False)
    y = x['Class']
    return x,y

data_0 = df.loc[df['Class']==0]
test_0, train_0 = splitData(data_0, 200)
data_1 = df.loc[df['Class']==1]
test_1, train_1 = splitData(data_1, 200)
X_tr, Y_train = getData(train_0, train_1)
X_test, Y_test = getData(test_0, test_1)
```

[66]:
```
neighbors = list(range(1,901,3))
kinv = []
k2=[]
training_error = []
test_error = []
best_error = 1
X_train = X_tr.drop(columns = ['Class'])
X_test = X_test.drop(columns = ['Class'])
```

[67]:
```
for k in neighbors:
  knn = KNeighborsClassifier(n_neighbors=k, p=2)
  knn.fit(np.array(X_train), np.array(Y_train))
  test_pred = knn.predict(X_test)
  train_pred = knn.predict(X_train)
  tr_error = 1 - metrics.accuracy_score(Y_train, train_pred)
  t_error = 1 - metrics.accuracy_score(Y_test, test_pred)
  training_error.append(tr_error)
  test_error.append(t_error)

  if t_error <= best_error:
    best_error = t_error
    kstar = k

  kinv.append(1/k)
  k2.append(k)
```

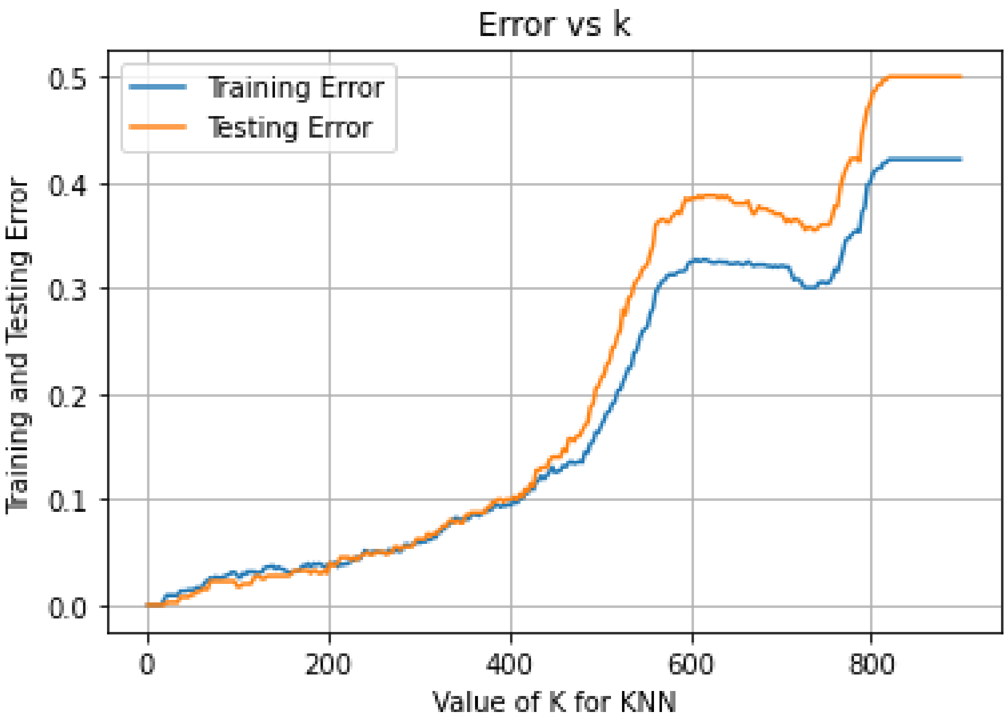
[68]:
```
plt.plot(kinv, training_error, label= 'Training Error')
plt.plot(kinv, test_error, label= 'Test Error')
plt.xlabel('1/K')
plt.ylabel('Error')
```

```
plt.legend()
plt.show()

print("The optimal value of k (highest) is %d" % kstar)
#plt.figure(figsize=(15,10))
plt.plot(k2, training_error, label='Training Error')
plt.plot(k2, test_error, label='Testing Error')
plt.grid()
plt.legend(loc='best')
plt.xlabel('Value of K for KNN')
plt.ylabel('Training and Testing Error')
plt.title('Error vs k')
```



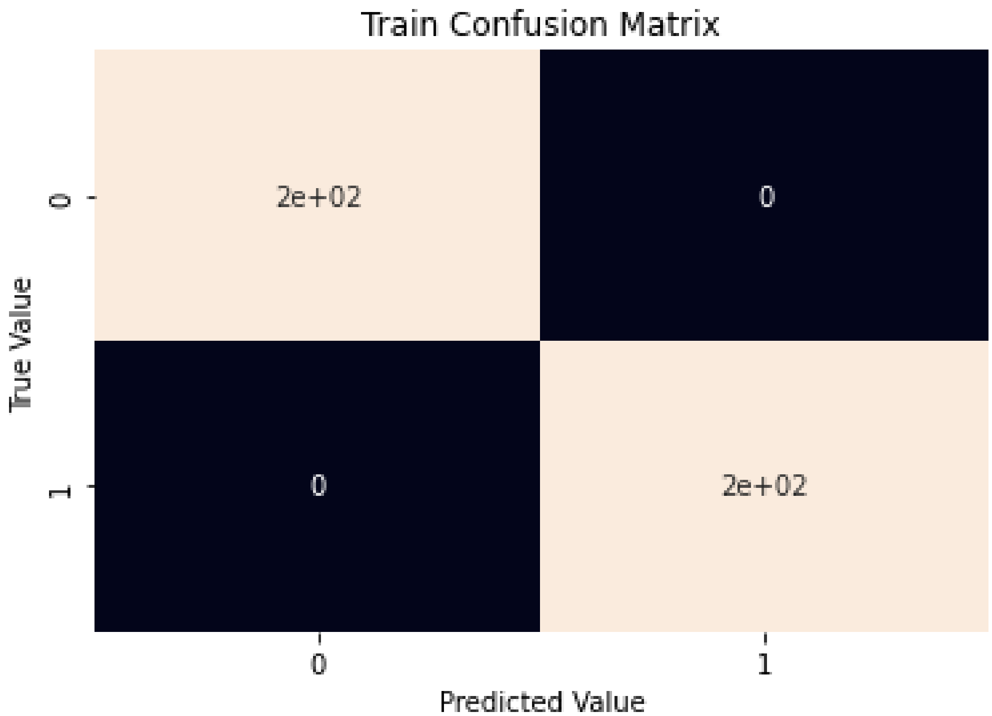

```
The optimal value of k (highest) is 19
```

```
[68]: Text(0.5, 1.0, 'Error vs k')
```



Confusion Matrix and Classification Report

[69]:
```
from sklearn.metrics import classification_report, confusion_matrix
knn = KNeighborsClassifier(n_neighbors=19)
knn.fit(np.array(X_train), np.array(Y_train))
pred = knn.predict(X_test)
cm = confusion_matrix(Y_test, pred)
ax= plt.subplot()
sns.heatmap(cm, annot=True, cbar= False, ax = ax);
plt.title('Train Confusion Matrix')
plt.xlabel('Predicted Value')
plt.ylabel('True Value')
plt.show()
plt.figure()
plt.show()
```




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

[70]:
```
print('Classification report: \n',classification_report(Y_test, pred))
print('TN - True Negative :{}'.format(cm[0,0]))
print('FP - False Positive :{}'.format(cm[0,1]))
print('FN - False Negative :{}'.format(cm[1,0]))
print('TP - True Positive :{}'.format(cm[1,1]))
print('Accuracy Rate: {}'.format(np.divide(np.sum([cm[0,0],cm[1,1]]),np.
↪sum(cm))))
print('Misclassification Rate: {}'.format(np.divide(np.
↪sum([cm[0,1],cm[1,0]]),np.sum(cm))))
```

```
Classification report:
               precision    recall  f1-score   support

         0.0       1.00      1.00      1.00       200
         1.0       1.00      1.00      1.00       200

    accuracy                           1.00       400
   macro avg       1.00      1.00      1.00       400
weighted avg       1.00      1.00      1.00       400

TN - True Negative :200
```

```
FP - False Positive :0
FN - False Negative :0
TP - True Positive :200
Accuracy Rate: 1.0
Misclassification Rate: 0.0
```

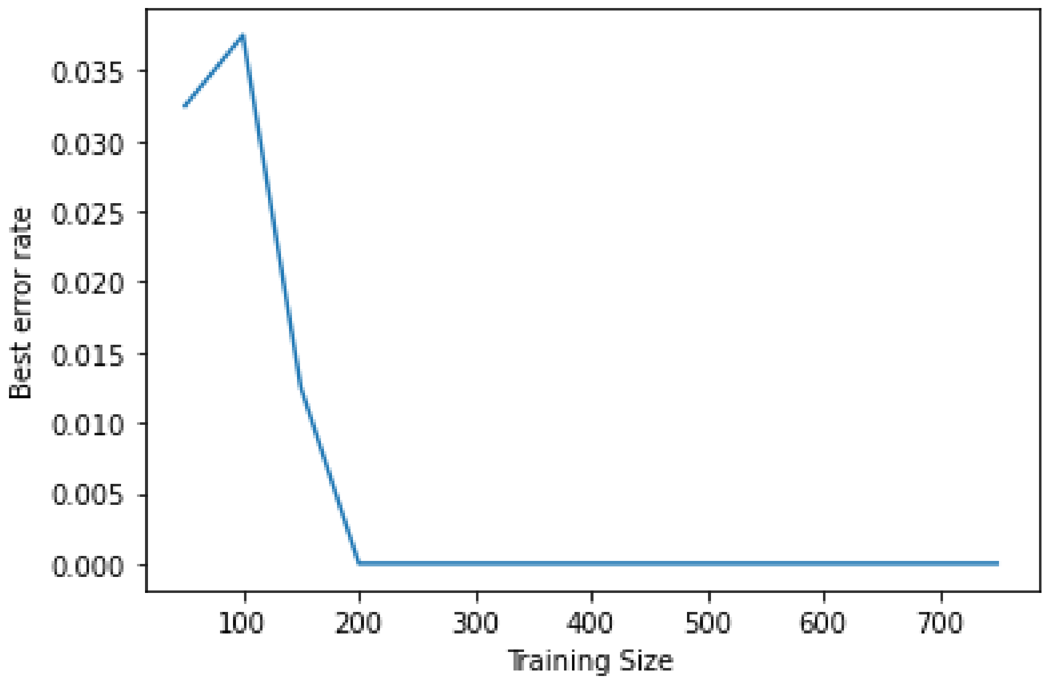
[71]:
```python
TN = cm[0][0]
FN = cm[1][0]
TP = cm[1][1]
FP = cm[0][1]
TPR = TP/(TP+FN)
TNR = TN/(TN+FP)
Precision = TP/(TP+FP)
Fscore = 2*TP/(2*TP+FP+FN)
print('True Negative Rate:', TNR)
print('True Positive Rate:', TPR)
print('Precision:', Precision)
print('F-score:', Fscore)
```

```
True Negative Rate: 1.0
True Positive Rate: 1.0
Precision: 1.0
F-score: 1.0
```

Learning curves

[77]:
```python
N = list(range(50, 800, 50))
min_test_error = []
for n in range(50, 800, 50):
  #split the train data from a(iii) into N/2
  train_data = X_tr.loc[X_tr['Class']==0].head(n//2)
  test_data = X_tr.loc[X_tr['Class']==1].head(n//2)
  x_train= pd.concat([train_data,test_data], sort=False)
  y_train = x_train['Class']
  x_train = x_train.drop(columns = ['Class'])

  neighbors = list(range(1,n,40))
  test_scores = []
  optimal_k = []

  for k in range(1,n,40):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(np.array(x_train), np.array(y_train))
    test_pred = knn.predict(X_test)
    test_scores.append(metrics.accuracy_score(Y_test, test_pred))
  test_error= [1 - t for t in test_scores]
  min_test_error.append(min(test_error))
```

```
plt.plot(N, min_test_error)
plt.xlabel('Training Size')
plt.ylabel('Best error rate')
plt.show()
```



Manhattan distance with logarithmetic distance

[79]:
```
def logDist(x, y,**kwargs):
    """
    This function gives a user defined minkowiski distance metric for logp values.
    """
    p = kwargs["t"]
    return np.sum(abs(np.subtract(x,y))**p)**(1/p)

logminkowiski_test_error= []
best_error = 1
best_logp = 0
for p in np.arange(0.1, 1.1, 0.1):
    knn_log_minkowiski= KNeighborsClassifier(n_neighbors=11, metric = logDist␣
 ↪,metric_params={'t': 10**p })
    knn_log_minkowiski.fit(np.array(X_train), np.array(Y_train))
    log_minkowiski_test_pred = knn_log_minkowiski.predict(X_test)
    logmtest_error = 1 - metrics.accuracy_score(Y_test, log_minkowiski_test_pred)
    logminkowiski_test_error.append(logmtest_error)
    if logmtest_error <= best_error:
```

```
    best_error = logmtest_error
    best_logp = p

print("The best log10p: ", best_logp)
```

```
The best log10p:  1.0
```

Manhattan distance metric

[80]:
```
neighbors = list(range(1,901,10))
best_test_error =[]
manhattan_test_error = []
optimal_k = []
best_error = 1

for k in neighbors:
  knn_manhattan = KNeighborsClassifier(n_neighbors=k, p=1)
  knn_manhattan.fit(np.array(X_train), np.array(Y_train))
  manhattan_test_pred = knn_manhattan.predict(X_test)
  mtest_error = 1 - metrics.accuracy_score(Y_test, manhattan_test_pred)
  manhattan_test_error.append(mtest_error)
  if mtest_error <= best_error:
    best_error = mtest_error
    kstar = k

best_test_error.append(best_error)
optimal_k.append(kstar)
print("The optimal value of k(manhattan) is %d" % kstar)
```

```
The optimal value of k(manhattan) is 11
```

Chebyshev metric

[81]:
```
neighbors = list(range(1,901,10))
best_test_error =[]
manhattan_test_error = []
optimal_k = []
best_error = 1

for k in neighbors:
  knn_manhattan = KNeighborsClassifier(n_neighbors=k, metric= 'chebyshev')
  knn_manhattan.fit(np.array(X_train), np.array(Y_train))
  manhattan_test_pred = knn_manhattan.predict(X_test)
  mtest_error = 1 - metrics.accuracy_score(Y_test, manhattan_test_pred)
  manhattan_test_error.append(mtest_error)
  if mtest_error <= best_error:
    best_error = mtest_error
    kstar = k
```

```
best_test_error.append(best_error)
optimal_k.append(kstar)
print("The optimal value of k(chebyshev) is %d" % kstar)
```

```
The optimal value of k(chebyshev) is 11
```

Weighted metric

[85]:
```
neighbors = list(range(1,901,10))
best_test_error =[]
manhattan_test_error = []
optimal_k = []
best_error = 1

for k in neighbors:
  knn_manhattan = KNeighborsClassifier(n_neighbors=k, metric=␣
 ↪'manhattan',weights='distance')
  knn_manhattan.fit(np.array(X_train), np.array(Y_train))
  manhattan_test_pred = knn_manhattan.predict(X_test)
  mtest_error = 1 - metrics.accuracy_score(Y_test, manhattan_test_pred)
  manhattan_test_error.append(mtest_error)
  if mtest_error <= best_error:
    best_error = mtest_error
    kstar = k

best_test_error.append(best_error)
optimal_k.append(kstar)
print("The optimal value of k(manhattan) is %d" % kstar)

#best_test_error =[]
chebyshev_test_error = []
#optimal_k = []
#best_error = 1
for k in neighbors:
  knn_chebyshev = KNeighborsClassifier(n_neighbors=k, metric=␣
 ↪'chebyshev',weights='distance')
  knn_chebyshev.fit(np.array(X_train), np.array(Y_train))
  chebyshev_test_pred = knn_chebyshev.predict(X_test)
  mtest_error = 1 - metrics.accuracy_score(Y_test, chebyshev_test_pred)
  chebyshev_test_error.append(mtest_error)
  if mtest_error <= best_error:
    best_error = mtest_error
    kstar = k

best_test_error.append(best_error)
optimal_k.append(kstar)
print("The optimal value of k(chebyshev) is %d" % kstar)
```

```
euclidean_test_error = []
#best_test_error =[]
#best_error = 1

for k in neighbors:
  knn_euclidean= KNeighborsClassifier(n_neighbors=k,weights='distance')
  knn_euclidean.fit(np.array(X_train), np.array(Y_train))
  euclidean_test_pred = knn_euclidean.predict(X_test)
  eubtest_error = 1 - metrics.accuracy_score(Y_test, euclidean_test_pred)
  euclidean_test_error.append(eubtest_error)
  if eubtest_error <= best_error:
    best_error = eubtest_error
    kstar = k

best_test_error.append(best_error)
optimal_k.append(kstar)
print("The optimal value of k(Euclidean) is %d" % kstar)
```

```
The optimal value of k(manhattan) is 81
The optimal value of k(chebyshev) is 481
The optimal value of k(Euclidean) is 51
```

[86]:
```
weighted_metric = OrderedDict({'weighted_Metric': ['Manhattan', 'Chebyshev',␣
↪'Euclidean'],'Optimal_k (Highest)': optimal_k,'Best_error': best_test_error})
metric_table = pd.DataFrame.from_dict(weighted_metric)
metric_table
```

[86]:

	weighted_Metric	Optimal_k (Highest)	Best_error
0	Manhattan	81	0.0
1	Chebyshev	481	0.0
2	Euclidean	51	0.0

Conclusions

Banknote authentication is an important task. It is difficult to manually detect fake banknotes. Machine learning algorithms can help in this regard. In this problem, we explained how we solved the problem of banknote authentication using machine learning techniques. We compared two different algorithms in terms of performance and concluded that the KNN & Logistics algorithms are the best algorithms for banknote authentication with an accuracy of 100% & 99.83%.