Pattern Recognition Assignment - 1

Name: Arnab Mukherjee Roll No: MB1908

In this assignment we will use python as our programming language to address the problems.

Generating Data Set A

We will generate 500 observations each from the trivariate normal distribution with distinct mean vectors as well as distinct dispersion matrices.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

In [2]:
```
#for 3D plot
from mpl_toolkits import mplot3d
from mpl_toolkits.mplot3d import Axes3D
plt.rcParams["figure.figsize"] = (30,10)
```

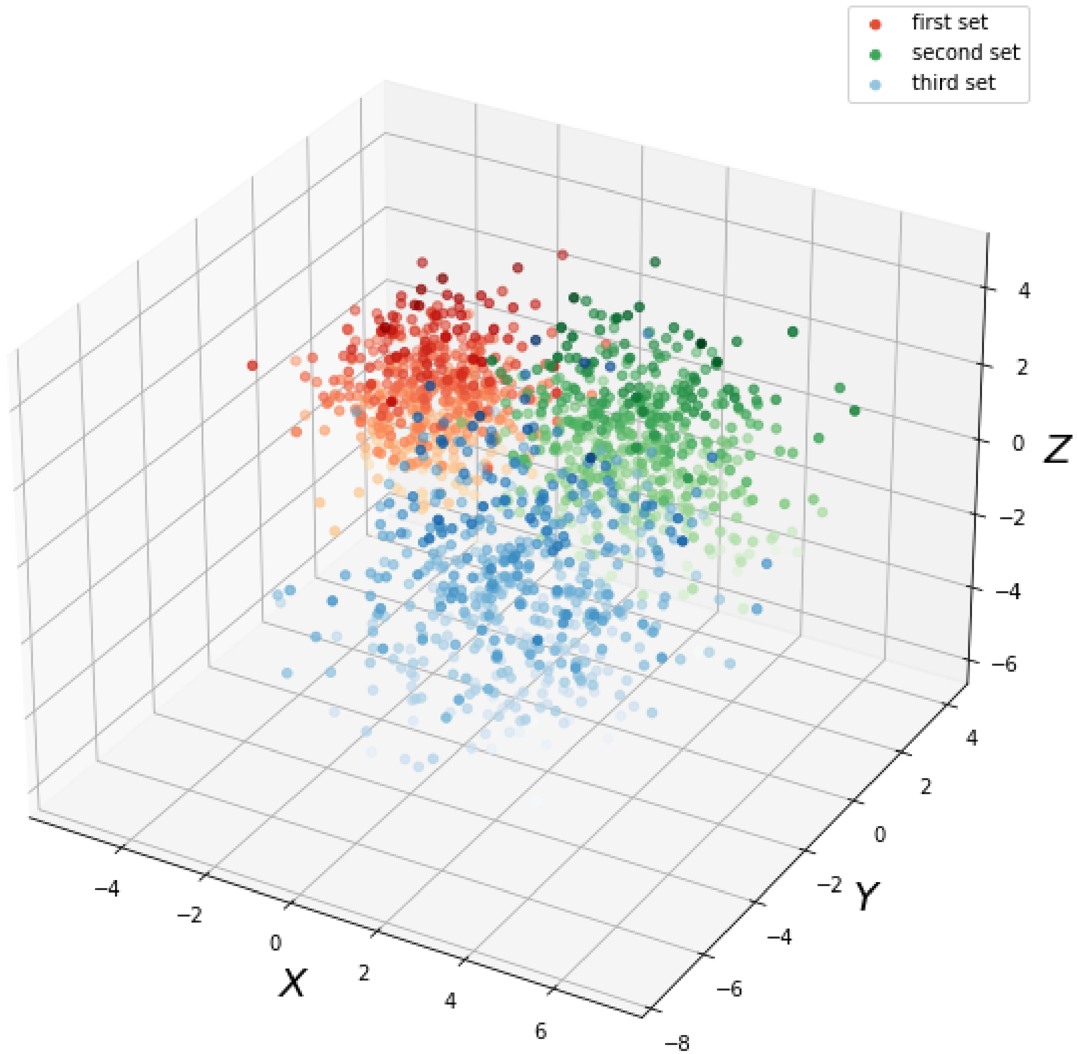
Question-1:

Here we will generate the dataset A and create 3D scatter plot for the generated dataset.

In [3]:
```
np.random.seed(1908)
ax = plt.axes(projection='3d')
mean1 = [-2,0,1]
mean2 = [2.1,1.3,0]
mean3 = [2,-3.02,-1]
var1 = [[1,0,0],[0,1,0],[0,0,1]]
var2 = [[2,0,0],[0,1,0],[0,0,2]]
```

```
var3 = [[3,0,0],[0,3,0],[0,0,3]]
data1_x,data1_y,data1_z = np.random.multivariate_normal(mean1,var1,500)
.T
data2_x,data2_y,data2_z = np.random.multivariate_normal(mean2,var2,500)
.T
data3_x,data3_y,data3_z = np.random.multivariate_normal(mean3,var3,500)
.T
ax.scatter3D(data1_x, data1_y, data1_z, c=data1_z, label = "first set",
cmap = "OrRd")
ax.scatter3D(data2_x, data2_y, data2_z, c=data2_z, label ='second set',
cmap = "Greens")
ax.scatter3D(data3_x, data3_y, data3_z, c=data3_z, label='third set',cm
ap = "Blues")
ax.set_xlabel('$X$', fontsize = 20)
ax.set_ylabel('$Y$',fontsize = 20)
ax.set_zlabel('$Z$',fontsize = 20)
ax.legend()
```

Out[3]:
```
<matplotlib.legend.Legend at 0x25b1c26cfd0>
```


first set
second set
third set
X
Y
Z
−4
−2
0
2
4
6
−8
−6
4

Next we convert this dataset to a dataframe for the sake of computational ease. Marker column below will represent the set from which the data comes from. marker values are 0,1,2.

In [4]:

```
#creating a data frame with these data
np.random.seed(1908)
df = pd.DataFrame(data1_x,columns=["X-axis"])
df_temp = pd.DataFrame(data1_y,columns = ["Y-axis"])
df = pd.concat([df,df_temp],axis = 1)
df_temp = pd.DataFrame(data1_z,columns = ["Z-axis"])
df= pd.concat([df,df_temp],axis = 1)
df_temp = pd.DataFrame(np.zeros(500),columns = ["Marker"])
df_1 = pd.concat([df,df_temp],axis = 1)

df = pd.DataFrame(data2_x,columns=["X-axis"])
df_temp = pd.DataFrame(data2_y,columns = ["Y-axis"])
df = pd.concat([df,df_temp],axis = 1)
df_temp = pd.DataFrame(data2_z,columns = ["Z-axis"])
df= pd.concat([df,df_temp],axis = 1)
df_temp = pd.DataFrame(np.ones(500),columns = ["Marker"])
df_2 = pd.concat([df,df_temp],axis = 1)

df = pd.DataFrame(data3_x,columns=["X-axis"])
df_temp = pd.DataFrame(data3_y,columns = ["Y-axis"])
df = pd.concat([df,df_temp],axis = 1)
df_temp = pd.DataFrame(data3_z,columns = ["Z-axis"])
df= pd.concat([df,df_temp],axis = 1)
df_temp = pd.DataFrame(np.repeat(2,500),columns = ["Marker"])
df_3 = pd.concat([df,df_temp],axis = 1)


df = pd.concat([df_1,df_2,df_3])

s = pd.Series(range(1500))
df.set_index([s])
df = df.sample(frac = 1)
df.head()
```

Out[4]:

	X-axis	Y-axis	Z-axis	Marker

	X-axis	Y-axis	Z-axis	Marker
385	1.950392	2.773708	-0.280443	1.0
444	-0.284602	-0.786112	0.954173	0.0
309	4.060312	1.442383	0.305969	1.0
262	1.174158	-5.471219	0.515854	2.0
97	2.686453	-3.871344	-1.980982	2.0

Generating Dataset B

First we import the complete csv leaf data set then we will do the following:

- We will randomly select 6 of 36 classes and use the original label
- We will select 4 out of 16 features. Now note that labels are itself a feature so we will essentially choose 4 out of 15 features.

In [5]:
```
cd "Downloads"
```

```
C:\Users\Arnab\Downloads
```

In [6]:
```
dfB = pd.read_csv("leaf.csv",header = None)
dfB.head()
```

Out[6]:

	0	1	2	3	4	5	6	7	8	9	10	
0	1	1	0.72694	1.4742	0.32396	0.98535	1.00000	0.83592	0.004657	0.003947	0.047790	0.127
1	1	2	0.74173	1.5257	0.36116	0.98152	0.99825	0.79867	0.005242	0.005002	0.024160	0.090
2	1	3	0.76722	1.5725	0.38998	0.97755	1.00000	0.80812	0.007457	0.010121	0.011897	0.057
3	1	4	0.73797	1.4597	0.35376	0.97566	1.00000	0.81697	0.006877	0.008607	0.015950	0.065
4	1	5	0.82301	1.7707	0.44462	0.97698	1.00000	0.75493	0.007428	0.010042	0.007938	0.045

In [7]:
```
#Getting unique label values:
dfB[0].unique()
```

Out[7]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 22,
23,
       24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36], dtype=int6
4)
```

Now interestingly we can see that we have unique values from 1-15 and then from 22-36. So we have to sample from this range and take care of the missing range and make sure that it doesn't hamper our sampling. Essentially we will be performing SRSWOR of size 6 from this set.

In [8]:
```
import random
unique = dfB[0].unique().tolist() #we want to convert in to a list
s = random.sample(unique, 6)
s
```

Out[8]:
```
[30, 5, 32, 7, 28, 15]
```

Now we will select the rows in which the label comes from the set s

In [9]:
```
listB = []
for i in range(len(dfB)):
    if dfB[0][i] in s:
        listB = listB + [i]
len(listB)
```

Out[9]:
```
67
```

Now we just select these rows

In [10]:
```
dfB = dfB.iloc[listB]
dfB.head()
```

Out[10]:

0	1	2	3	4	5	6	7	8	9	10

	0	1	2	3	4	5	6	7	8	9	10
40	5	1	0.87844	1.8096	0.63151	0.83923	0.83684	0.37688	0.043563	0.34539	0.045468	0.125
41	5	2	0.88075	1.7360	0.58345	0.83383	0.91754	0.41551	0.040582	0.29973	0.035786	0.100
42	5	3	0.86545	1.8803	0.62039	0.82443	0.85439	0.33077	0.047000	0.40204	0.039518	0.115
43	5	4	0.93671	2.4151	0.72980	0.81793	0.86491	0.33439	0.080539	1.18050	0.048722	0.120
44	5	5	0.92676	2.2220	0.65580	0.82432	0.89474	0.35618	0.038012	0.26298	0.059981	0.149

Now we select 4 out of 15 features (we have to select the first feature)

In [11]:
```
l = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
s = random.sample(l,4)
```

In [12]:
```
s_updated = [0] + s
s_updated
```

Out[12]: [0, 9, 7, 6, 14]

In [13]:
```
dfB = dfB.iloc[:,s_updated]
```

In [14]:
```
dfB.head()
```

Out[14]:

	0	9	7	6	14
40	5	0.34539	0.37688	0.83684	0.000228
41	5	0.29973	0.41551	0.91754	0.000258
42	5	0.40204	0.33077	0.85439	0.000201
43	5	1.18050	0.33439	0.86491	0.000372
44	5	0.26298	0.35618	0.89474	0.000284

So now we have the synthetic dataset , i.e. dataset A as dataframe df and the real dataset , i.e.

dataset B as dataframe dfB . An important point to remember in the dataset A the dependent variable is in the Marker column while for dataset B it is in the column 0

Partitioning Dataset into 50% Train and 50% Test

In [15]:
```
#for Dataset A
np.random.seed(1908)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(df.iloc[:,[0,1,2]],
 df["Marker"].values, test_size=0.50)
```

In [16]:
```
#for Dataset B
np.random.seed(1908)
XB_train,XB_test,yB_train,yB_test = train_test_split(dfB.iloc[:,[1,2,3,
4]], dfB[0].values, test_size=0.50)
```

Question-2:

Here we will have to calculate training and test error rates of the Bayes Classifier with respect to the classification problems corresponding to dataset A and dataset B.

Bayes Error:

In [17]:
```
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
```

Dataset-A

In [18]:
```
y_pred = gnb.fit(X_train, y_train).predict(X_test)
tst = ((y_test != y_pred).sum()/750)*100
y_pred_1 = gnb.fit(X_train, y_train).predict(X_train)
tr = ((y_train != y_pred_1).sum()/750)*100
print("Training error rate: " + str(tr) + "%")
print("Test error rate: "+ str(tst) + "%")
```

```
Training error rate: 6.800000000000001%
Test error rate: 6.4%
```

In [19]:
```
#Confusion matrix for test data
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
[[225   9   6]
 [  6 241  10]
 [  7  10 236]]
```

Dataset-B

In [20]:
```
from sklearn.metrics import accuracy_score
y_pred = gnb.fit(XB_train, yB_train).predict(XB_test)
tst = 1- accuracy_score(yB_test, y_pred)
y_pred_1 = gnb.fit(XB_train, yB_train).predict(XB_train)
tr = 1 - accuracy_score(yB_train, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
```

```
Training error rate: 3.0303030303030276%
Test error rate: 41.17647058823529%
```

In [21]:
```
#Confusion matrix for test data
cm = confusion_matrix(yB_test, y_pred)
print(cm)
```

```
[[3 0 0 0 0 0]
 [0 6 0 1 0 0]
 [2 0 4 0 0 0]
 [0 0 0 5 0 1]
 [0 1 0 9 0 0]
 [0 0 0 0 0 2]]
```

Question-3

Linear Discriminant Analysis

Dataset-A

In [22]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as
 LDA

lda = LDA(n_components=2)
lda.fit(X_train,y_train)
y_pred = lda.predict(X_test)
tst = 1 - accuracy_score(y_test, y_pred)
y_pred_1 = lda.predict(X_train)
tr = 1 - accuracy_score(y_train, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
```

```
Training error rate: 7.733333333333336%
Test error rate: 6.799999999999995%
```

In [23]:
```
#Confusion matrix for test data
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
[[233   5   2]
 [ 10 243   4]
 [ 13  17 223]]
```

Dataset-B

In [24]:
```
XB_train = sc.fit_transform(XB_train)
```

```
XB_test = sc.transform(XB_test)
lda = LDA(n_components=3)
lda.fit(XB_train,yB_train)
y_pred = lda.predict(XB_test)
tst = 1 - accuracy_score(yB_test, y_pred)
y_pred_1 = lda.predict(XB_train)
tr = 1 - accuracy_score(yB_train, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
```

```
Training error rate: 12.121212121212121%
Test error rate: 52.94117647058824%
```

In [25]:

```
#Confusion matrix for test data
cm = confusion_matrix(yB_test, y_pred)
print(cm)
```

```
[[3 0 0 0 0 0]
 [0 3 0 0 0 4]
 [1 0 5 0 0 0]
 [0 1 0 1 3 1]
 [0 0 0 6 2 2]
 [0 0 0 0 0 2]]
```

Quadratic Discriminant Analysis

Dataset-A

In [26]:

```
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
 as QDA
clf = QDA()
clf.fit(X_train, y_train)
QDA(priors=None, reg_param=0.0)
y_pred = clf.predict(X_test)
tst = 1 - accuracy_score(y_test, y_pred)
y_pred_1 = clf.predict(X_train)
tr = 1 - accuracy_score(y_train, y_pred_1)
```

```
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
```

```
Training error rate: 6.933333333333335%
Test error rate: 6.266666666666665%
```

In [27]:

```
#Confusion matrix for test data
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
[[225  10   5]
 [  5 243   9]
 [  7  11 235]]
```

Dataset-B

In [28]:

```
clf = QDA()
clf.fit(XB_train, yB_train)
QDA(priors=None, reg_param=0.0)
y_pred = clf.predict(XB_test)
tst = 1 - accuracy_score(yB_test, y_pred)
y_pred_1 = clf.predict(XB_train)
tr = 1 - accuracy_score(yB_train, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
```

```
Training error rate: 0.0%
Test error rate: 70.58823529411764%
```

```
C:\Users\Arnab\AppData\Roaming\Python\Python38\site-packages\sklearn\di
scriminant_analysis.py:715: UserWarning: Variables are collinear
  warnings.warn("Variables are collinear")
```

In [29]:

```
#Confusion matrix for test data
cm = confusion_matrix(yB_test, y_pred)
print(cm)
```

```
[[ 3  0  0  0  0  0]
```

```
 [ 5   0   0   2   0   0]
 [ 6   0   0   0   0   0]
 [ 1   0   0   5   0   0]
 [ 0   0   0  10   0   0]
 [ 0   0   0   0   0   2]]
```

Support Vector Machine - one vs one

We will use linear kernel, polynomial kernel and gaussian kernel

Dataset-A

In [30]:
```
from sklearn.svm import SVC
```

In [31]:
```
#Linear Kernel
linear_svc1 = SVC(
    C=10000, kernel='linear'
).fit(X_train, y_train)
y_pred = linear_svc1.predict(X_test)
tst = 1 - accuracy_score(y_test, y_pred)
y_pred_1 = linear_svc1.predict(X_train)
tr = 1 - accuracy_score(y_train, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
Training error rate: 7.733333333333336%
Test error rate: 6.266666666666665%
[[227   7   6]
 [  3 246   8]
 [ 10  13 230]]
```

In [32]:
```
#Ploynomial Kernel
poly_svc1 = SVC(
    C=10, kernel='poly', degree=4, coef0=7
).fit(X_train, y_train)
```

```
y_pred = poly_svc1.predict(X_test)
tst = 1 - accuracy_score(y_test, y_pred)
y_pred_1 = poly_svc1.predict(X_train)
tr = 1 - accuracy_score(y_train, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
Training error rate: 5.466666666666664%
Test error rate: 6.799999999999995%
[[223   9   8]
 [  6 246   5]
 [ 11  12 230]]
```

In [33]:

```
#Gaussian Kernel
rbf_svc1 = SVC(
    kernel='rbf', gamma=1
).fit(X_train, y_train)
y_pred = rbf_svc1.predict(X_test)
tst = 1 - accuracy_score(y_test, y_pred)
y_pred_1 = rbf_svc1.predict(X_train)
tr = 1 - accuracy_score(y_train, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
Training error rate: 5.8666666666666645%
Test error rate: 5.733333333333334%
[[226   9   5]
 [  4 250   3]
 [  7  15 231]]
```

Dataset-B

In [34]:

```
#Linear Kernel
linear_svc1 = SVC(
```

```
    C=10000, kernel='linear'
).fit(XB_train, yB_train)
y_pred = linear_svc1.predict(XB_test)
tst = 1 - accuracy_score(yB_test, y_pred)
y_pred_1 = linear_svc1.predict(XB_train)
tr = 1 - accuracy_score(yB_train, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
cm = confusion_matrix(yB_test, y_pred)
print(cm)
```

```
Training error rate: 0.0%
Test error rate: 32.35294117647059%
[[3 0 0 0 0 0]
 [0 6 0 0 0 1]
 [1 0 5 0 0 0]
 [0 1 0 2 3 0]
 [0 2 0 2 6 0]
 [0 1 0 0 0 1]]
```

In [35]:

```
#Ploynomial Kernel
poly_svc1 = SVC(
    C=10, kernel='poly', degree=4, coef0=7
).fit(XB_train, yB_train)
y_pred = poly_svc1.predict(XB_test)
tst = 1 - accuracy_score(yB_test, y_pred)
y_pred_1 = poly_svc1.predict(XB_train)
tr = 1 - accuracy_score(yB_train, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
cm = confusion_matrix(yB_test, y_pred)
print(cm)
```

```
Training error rate: 0.0%
Test error rate: 35.29411764705882%
[[3 0 0 0 0 0]
 [0 6 0 0 0 1]
 [1 0 5 0 0 0]
 [0 1 0 1 3 1]
```

 [0 2 0 2 6 0]
 [0 1 0 0 0 1]]

In [36]:
```
#Gaussian Kernel
rbf_svc1 = SVC(
    kernel='rbf', gamma=1
).fit(XB_train, yB_train)
y_pred = rbf_svc1.predict(XB_test)
tst = 1 - accuracy_score(yB_test, y_pred)
y_pred_1 = rbf_svc1.predict(XB_train)
tr = 1 - accuracy_score(yB_train, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
cm = confusion_matrix(yB_test, y_pred)
print(cm)
```

```
Training error rate: 12.121212121212121%
Test error rate: 58.82352941176471%
[[3 0 0 0 0 0]
 [0 2 0 0 0 5]
 [3 0 3 0 0 0]
 [0 0 0 4 0 2]
 [3 0 0 5 0 2]
 [0 0 0 0 0 2]]
```

Support Vector Machine: one vs many

Dataset-A

In [37]:
```
from sklearn.multiclass import OneVsRestClassifier
clf = OneVsRestClassifier(SVC()).fit(X_train, y_train)
y_pred = clf.predict(X_test)
y_pred = clf.predict(X_test)
tst = 1 - accuracy_score(y_test, y_pred)
y_pred_1 = clf.predict(X_train)
tr = 1 - accuracy_score(y_train, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
```

```
print("Test error rate: "+ str(tst*100) + "%")
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
Training error rate: 6.266666666666665%
Test error rate: 6.000000000000005%
[[228   9   3]
 [  3 248   6]
 [  7  17 229]]
```

Dataset-B

In [38]:
```
clf = OneVsRestClassifier(SVC()).fit(XB_train, yB_train)
y_pred = clf.predict(XB_test)
y_pred = clf.predict(XB_test)
tst = 1 - accuracy_score(yB_test, y_pred)
y_pred_1 = clf.predict(XB_train)
tr = 1 - accuracy_score(yB_train, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
cm = confusion_matrix(yB_test, y_pred)
print(cm)
```

```
Training error rate: 18.181818181818176%
Test error rate: 55.88235294117647%
[[3 0 0 0 0 0]
 [0 1 0 0 0 6]
 [0 0 6 0 0 0]
 [0 0 0 3 0 3]
 [0 0 0 7 0 3]
 [0 0 0 0 0 2]]
```

K-Nearest Neighbour Classification with K=1

Dataset-A

In [39]:
```
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors=1)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
tst = 1 - accuracy_score(y_test, y_pred)
y_pred_1 = classifier.predict(X_train)
tr = 1 - accuracy_score(y_train, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
Training error rate: 0.0%
Test error rate: 10.533333333333328%
[[217  11  12]
 [ 11 224  22]
 [  6  17 230]]
```

Dataset-B

In [40]:
```
classifier = KNeighborsClassifier(n_neighbors=1)
classifier.fit(XB_train, yB_train)
y_pred = classifier.predict(XB_test)
tst = 1 - accuracy_score(yB_test, y_pred)
y_pred_1 = classifier.predict(XB_train)
tr = 1 - accuracy_score(yB_train, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
cm = confusion_matrix(yB_test, y_pred)
print(cm)
```

```
Training error rate: 0.0%
Test error rate: 64.70588235294117%
[[3 0 0 0 0 0]
 [0 3 0 0 0 4]
 [4 0 2 0 0 0]
 [0 0 0 2 2 2]
```

```
 [0 1 0 8 0 1]
 [0 0 0 0 0 2]]
```

K-Nearest Neighbour Classification with K=3

Dataset-A

In [41]:
```
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors=3)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
tst = 1 - accuracy_score(y_test, y_pred)
y_pred_1 = classifier.predict(X_train)
tr = 1 - accuracy_score(y_train, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
Training error rate: 5.333333333333334%
Test error rate: 7.333333333333336%
[[224   9   7]
 [ 10 240   7]
 [  6  16 231]]
```

Dataset-B

In [42]:
```
classifier = KNeighborsClassifier(n_neighbors=3)
classifier.fit(XB_train, yB_train)
y_pred = classifier.predict(XB_test)
tst = 1 - accuracy_score(yB_test, y_pred)
y_pred_1 = classifier.predict(XB_train)
tr = 1 - accuracy_score(yB_train, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
```

```
cm = confusion_matrix(yB_test, y_pred)
print(cm)
```

```
Training error rate: 21.212121212121215%
Test error rate: 55.88235294117647%
[[3 0 0 0 0 0]
 [1 1 0 0 0 5]
 [0 0 6 0 0 0]
 [0 0 0 1 3 2]
 [0 0 0 6 2 2]
 [0 0 0 0 0 2]]
```

Question-4:

In this question we have to use multiedit and condensation method. Now unfortunately there aren't any packages in python that have functions which can perform these tasks. For this reason I wrote the function for multiedit and condensation on my own. I will use it here. Note that both the functions returns dataframe for X and array for y.

In [43]:
```
#Function to perform multiediting, with N = 3 partitions and I = 5 and
 assuming the dataset is already shuffled
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors=1) #Because we are workin
g with 1-NNR
def multiedit(X,y):
    df1 = X[:250]
    y1 = y[:250]
    df2 = X[250:500]
    y2 = y[250:500]
    df3 = X[500:]
    y3 = y[500:]
    I = 1
    count = len(X)
    print("Number of samples = " + str(count))
    while(I<5):
        #Reindexing phase
        df1.index = pd.RangeIndex(len(df1.index))
```

```
        df1.index = range(len(df1.index))
        df2.index = pd.RangeIndex(len(df2.index))
        df2.index = range(len(df2.index))
        df3.index = pd.RangeIndex(len(df3.index))
        df3.index = range(len(df3.index))
        #For partition-1 we test it using knn for partition-2
        classifier.fit(df2, y2)
        y_pred = classifier.predict(df1)
        list1 = []
        for i in range(len(df1)):
            if y_pred[i] != y1[i]:
                list1.append(i)
        #For partition-2 we test it using knn for partition-3
        classifier.fit(df3, y3)
        y_pred = classifier.predict(df2)
        list2 = []
        for i in range(len(df2)):
            if y_pred[i] != y2[i]:
                list2.append(i)
        #For partition-3 we test it using knn for partition-1
        classifier.fit(df1, y1)
        y_pred = classifier.predict(df3)
        list3 = []
        for i in range(len(df3)):
            if y_pred[i] != y3[i]:
                list3.append(i)
        #discarding misclassified samples
        df1 = df1.drop(list1)
        y1 = [i for j, i in enumerate(y1) if j not in list1]
        df2 = df2.drop(list2)
        y2 = [i for j, i in enumerate(y2) if j not in list2]
        df3 = df3.drop(index = list3)
        y3 = [i for j, i in enumerate(y3) if j not in list3]

        temp = len(df1) + len(df2) + len(df3)
        if count == temp:
            I = I+1
        else:
            I = 1
```

```
        count = temp
        print("Number of samples = " + str(count))
    df = pd.concat([df1,df2,df3],axis = 0)
    y_new = y1 + y2 + y3
    return df,y_new
```

In [44]:

```
#Function for the Basuc condensed NN (CNN) Rule
classifier = KNeighborsClassifier(n_neighbors=1)
def cnn(X,y):
    df1 = X[:2]
    y1 = y[:2]
    df2 = X[2:]
    y2 = y[2:]
    count = 1
    df1.index = pd.RangeIndex(len(df1.index))
    df1.index = range(len(df1.index))
    df2.index = pd.RangeIndex(len(df2.index))
    df2.index = range(len(df2.index))
    while count!=0 and len(y2)!=0:
        df1.index = pd.RangeIndex(len(df1.index))
        df1.index = range(len(df1.index))
        df2.index = pd.RangeIndex(len(df2.index))
        df2.index = range(len(df2.index))
        classifier.fit(df1,y1)
        y_pred = classifier.predict(df2)
        list1 = []
        for i in range(len(df2)):
            if y_pred[i] != y2[i]:
                list1.append(i)
        count = len(list1)
        df_temp = df2.iloc[list1]
        df1 = pd.concat([df1,df_temp],axis = 0)
        df2 = df2.drop(list1)
        temp = []
        for i in range(len(y2)):
            if i in list1:
                y1 = np.append(y1,y2[i])
            else:
                temp.append(y2[i])
```

```
        y2 = temp.copy()
        print("Number of misclassified points = "+ str(count))
    return df1,y1
```

Applying MultiEdit and then performing 1-NNR

In [45]:
```
X = df.iloc[:750,[0,1,2]]# 50% training set
X.head()
```

Out[45]:

	X-axis	Y-axis	Z-axis
385	1.950392	2.773708	-0.280443
444	-0.284602	-0.786112	0.954173
309	4.060312	1.442383	0.305969
262	1.174158	-5.471219	0.515854
97	2.686453	-3.871344	-1.980982

In [46]:
```
y = df["Marker"].values[:750]
```

In [47]:
```
df_test = df.iloc[750:,[0,1,2]]# 50% test set
y_test = df["Marker"].values[750:]
```

In [48]:
```
df_new,y_new = multiedit(X,y)
```

```
Number of samples = 750
Number of samples = 677
Number of samples = 670
Number of samples = 670
Number of samples = 670
Number of samples = 670
Number of samples = 670
```

In [49]:
```
classifier = KNeighborsClassifier(n_neighbors=1)
```

```
classifier.fit(df_new, y_new)
y_pred = classifier.predict(df_test)
tst = 1 - accuracy_score(y_test, y_pred)
y_pred_1 = classifier.predict(df_new)
tr = 1 - accuracy_score(y_new, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
Training error rate: 0.0%
Test error rate: 6.3999999999999995%
[[226   6   4]
 [  6 246   6]
 [  8  18 230]]
```

Applying Condensation and 1-NNR

In [50]:

```
df_new , y_new = cnn(X,y)
classifier = KNeighborsClassifier(n_neighbors=1)
classifier.fit(df_new, y_new)
y_pred = classifier.predict(df_test)
tst = 1 - accuracy_score(y_test, y_pred)
y_pred_1 = classifier.predict(df_new)
tr = 1 - accuracy_score(y_new, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
Number of misclassified points = 304
Number of misclassified points = 370
Number of misclassified points = 2
Number of misclassified points = 0
Training error rate: 0.0%
Test error rate: 9.733333333333338%
[[219  12   5]
```

```
 [ 10 228  20]
 [ 11  15 230]]
```

Applying MultiEdit and Condensation and then 1-NNR

In [51]:

```
df_new,y_new = multiedit(X,y)
df_new , y_new = cnn(df_new,y_new)
classifier = KNeighborsClassifier(n_neighbors=1)
classifier.fit(df_new, y_new)
y_pred = classifier.predict(df_test)
tst = 1 - accuracy_score(y_test, y_pred)
y_pred_1 = classifier.predict(df_new)
tr = 1 - accuracy_score(y_new, y_pred_1)
print("Training error rate: " + str(tr*100) + "%")
print("Test error rate: "+ str(tst*100) + "%")
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
Number of samples = 750
Number of samples = 677
Number of samples = 670
Number of samples = 670
Number of samples = 670
Number of samples = 670
Number of samples = 670
Number of misclassified points = 254
Number of misclassified points = 137
Number of misclassified points = 7
Number of misclassified points = 0
Training error rate: 0.0%
Test error rate: 6.533333333333335%
[[225   7   4]
 [  5 246   7]
 [  8  18 230]]
```

Comparison Tables:

Dataset-A

Name	Train Error	Test Error
Bayes(Naive)	6.8	6.4
LDA	7.733	6.7999
QDA	6.933	6.266
SVM one v one Kernel = linear	7.733	6.266
SVM one v one Kernel = polynomial	5.466	6.7999
SVM one v one Kernel = gaussian	5.866	5.7333
SVM one v many	6.2666	6.0000
KNN K=1	0.0	10.5333
KNN K=3	5.33	7.333
MultiEdit - 1NNR	0.0	6.399
Condensation-1NNR	0.0	9.733
MultiEdit-Condensation-1NNR	0.0	6.5333

As we can see from the above table that one v one svm with gaussian kernel is the best fit for this dataset. While it might strike surprising that the error rate is lower than bayes error rate but we have to remember that we are not evaluating the actual bayes error rate here , rather we are evaluating the naive bayes with normal probability model.

Dataset-B

Name	Train Error	Test Error
Bayes(Naive)	3.3	41.176
LDA	12.12	52.941
QDA	0.0	70.588
SVM one v one Kernel = linear	0.0	32.3529

Name	Train Error	Test Error
SVM one v one Kernel = polynomial	0.0	35.29411
SVM one v one Kernel = gaussian	12.1212	58.8235
SVM one v many	18.1818	55.88235
KNN K=1	0.0	64.7058
KNN K=3	21.2121	55.88235

Note that here typically the training error is very low and test error is extremely high this is primarily due to the fact that the data set is so small. In this setup SVM one v one with linear kernel performs best but still there is clear indication of over-fitting. Same logic as the previous explanation applies for naive bayes here as well. (i.e. it is not the true bayes error rate). So clearly in this particular setup we need more data before we can conclude anything.