Programming Assignment II

CS5691: Pattern Recognition and Machine Learning

Team 17 CCS Section Spring 2021

September 23, 2022

Aanand Krishnan
BE17B001

Manoranjan J
NA17B112

Reneeth Krishna MG
BS17B025

Contents

I Pattern classification on linearly separable data

I.1 Python Code

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from mlxtend.plotting import plot_confusion_matrix
from sklearn.metrics import accuracy_score

################################################################################
# train data
f = open('datasets/Dataset_1a/train.csv', 'r')

length = 0
for line in f:

    d = [float(i) for i in line.split(',')]

    if length == 0:
        data = np.array(d)
        data = data[np.newaxis,:]
    else:
        d = np.array(d)
        d = d[np.newaxis,:]
        data = np.append(data,d,axis=0)
    length = length+1

f.close()

train_data = pd.DataFrame(data)

# shuffle dataset
# train_data = train_data.sample(frac=1)

# get train data
train_data = np.array(train_data)

# test data
f = open('datasets/Dataset_1a/dev.csv', 'r')

length = 0
for line in f:

    d = [float(i) for i in line.split(',')]

    if length == 0:
        data = np.array(d)
        data = data[np.newaxis,:]
```

```python
    else:
        d = np.array(d)
        d = d[np.newaxis,:]
        data = np.append(data,d,axis=0)
    length = length+1

f.close()

test_data = pd.DataFrame(data)

# shuffle dataset
# test_data = test_data.sample(frac=1)

# get test data
test_data = np.array(test_data)
################################################################################

# Split training data
# length of data for fit
train_len = int(np.shape(train_data)[0])
val_len = int(np.shape(test_data)[0]*0.5)
test_len = int(np.shape(test_data)[0]*0.5)

X_train = train_data[:,0:2]
X_val = test_data[0:val_len,0:2]
X_test = test_data[val_len:val_len+test_len,0:2]

y_train = train_data[:,2]
y_val = test_data[0:val_len,2]
y_test = test_data[val_len:val_len+test_len,2]

# ################################################################################
# Build KNN classifier
y_pred = []
def KNN_classifier(K, x):
    """
    Parameters
    ----------
    K : value of nearest neighbours
    x : feature vector

    Returns
    -------
    None.
    """
    # find distance between feature vector and training data
    dist = np.linalg.norm(x-X_train,axis=1)

    # get the top index for the minimum distance
    min_dist_index = np.argsort(dist)
```

```
    topk = min_dist_index[0:K]

    # get class of corresponding class
    K_class = y_train[topk]

    # get count of each class
    unique_class, counts = np.unique(K_class, return_counts=1)

    # get the index of max counts
    max_count = np.argmax(counts)

    # choose that as the class
    y_pred.append(unique_class[max_count])

K = 15
# test on training data
for i in range(0, train_len):
    KNN_classifier(K, X_train[i])

y_pred = np.array(y_pred)

train_accuracy = accuracy_score(y_train,y_pred)*100
print("training accuracy: " , train_accuracy)

# plot confusion matrix for training data
c_matrix  = confusion_matrix(y_train, y_pred)
fig, ax = plot_confusion_matrix(conf_mat=c_matrix,figsize=(7,7),cmap=plt.cm.RdYlBu_r)
ax.set(title = "Confusion Matrix")
plt.show()

y_pred = []
# test on validation data
for i in range(0, val_len):
    KNN_classifier(K, X_val[i])

y_pred = np.array(y_pred)

val_accuracy = accuracy_score(y_val,y_pred)*100
print("validation accuracy: " , val_accuracy)

y_pred = []
# test on test data
for i in range(0, test_len):
    KNN_classifier(K, X_test[i])

y_pred = np.array(y_pred)

test_accuracy = accuracy_score(y_test,y_pred)*100
print("validation accuracy: " , test_accuracy)
```

```python
# plot confusion matrix for test data
c_matrix  = confusion_matrix(y_test, y_pred)
fig, ax = plot_confusion_matrix(conf_mat=c_matrix,figsize=(7,7),cmap=plt.cm.RdYlBu_r)
ax.set(title = "Confusion Matrix")
plt.show()

# Decision Region Plots ######################################################
# define bounds of the domain
min1, max1 = X_train[:, 0].min()-1, X_train[:, 0].max()+1
min2, max2 = X_train[:, 1].min()-1, X_train[:, 1].max()+1

# define the x and y scale
x1_grid = np.arange(min1, max1, 0.1)
x2_grid = np.arange(min2, max2, 0.1)

x1_grid, x2_grid = np.meshgrid(x1_grid, x2_grid)

c1, c2 = x1_grid.flatten(), x1_grid.flatten()
c1, c2 = x1_grid.reshape((len(c1), 1)), x2_grid.reshape((len(c2), 1))

x = np.hstack((c1,c2))

y_pred = []
for i in range(0, np.shape(x)[0]):
    KNN_classifier(K, x[i,:])

y_pred = np.array(y_pred)

x3_grid = y_pred.reshape(x1_grid.shape)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.contourf(x1_grid, x2_grid, x3_grid, cmap='Paired')
ax.scatter(X_train[:,0],X_train[:,1],marker='x')
ax.set_xlabel('x1',fontsize=20)
ax.set_ylabel('x2',fontsize=20)
ax.set_title('KNN model with K = 15', fontsize=20)


################################################################################

# Naive Bayes classifier
# P(x/y=yi) = N(x/mui,ci)
unique_class, class_index, counts = np.unique(y_train, return_inverse=1,return_counts

# Compute mean and variance for each class
mu = np.zeros((np.shape(unique_class)[0],2))
variance = np.zeros((np.shape(unique_class)[0],2))
for i in range(0,np.size(unique_class)):
    index = np.where(class_index==i)
```

```
    mu[i,:] = np.mean(X_train[index,:],axis=1)
    variance[i,:] = np.var(X_train[index,:],axis=1)

# Gaussian function
N = lambda mu,C,x : ((1/(((2*np.pi)**(np.shape(unique_class)[0]/2))
*np.linalg.det(C)**(1/2)))*np.exp(-0.5*(((x-mu).T)@np.linalg.inv(C)@(x-mu))))
################################################################################
# Comment all other case when testing 1 case

# case 1: when covariance matrix is sigma**2[I] ##############################
var_avg = (np.mean(np.sum(variance,axis=1)/2.0))

covar = np.eye(np.shape(X_train)[1])*var_avg
################################################################################
# case 2: when covariance matrix is same but has different diaganol elements
covar = np.zeros((np.shape(unique_class)[0],
np.shape(X_train)[1],np.shape(X_train)[1]))
for i in range(0,np.shape(unique_class)[0]):
    covar[i,:,:] = np.diag(variance[i,:])
covar = np.mean(covar,axis=0)
################################################################################
# case 3: when covariance matrix is different but has diaganol elements
covar = np.zeros((np.shape(unique_class)[0],
np.shape(X_train)[1],np.shape(X_train)[1]))
for i in range(0,np.shape(unique_class)[0]):
    covar[i,:,:] = np.diag(variance[i,:])


prior = counts/train_len

# test on train data
y_pred = np.zeros((train_len,))
for i in range(train_len):
    decision = []
    for j in range(0,np.shape(unique_class)[0]):
        decision.append(N(mu[j],covar[j,:,:],X_train[i,:])*prior[j])
    y_pred[i] = np.argmax(decision)

train_accuracy = accuracy_score(y_train,y_pred)*100
print("training accuracy: " , train_accuracy)

# plot confusion matrix for training data
c_matrix  = confusion_matrix(y_train, y_pred)
fig, ax = plot_confusion_matrix(conf_mat=c_matrix,figsize=(7,7),cmap=plt.cm.RdYlBu_r)
ax.set(title = "Confusion Matrix")
plt.show()

# test on validation data
y_pred = np.zeros((val_len,))
for i in range(val_len):
```

```
    decision = []
    for j in range(0,np.shape(unique_class)[0]):
        decision.append(N(mu[j],covar[j,:,:],X_val[i,:])*prior[j])
    y_pred[i] = np.argmax(decision)

val_accuracy = accuracy_score(y_val,y_pred)*100
print("validation accuracy: " , val_accuracy)

# test on test data
y_pred = np.zeros((test_len,))
for i in range(test_len):
    decision = []
    for j in range(0,np.shape(unique_class)[0]):
        decision.append(N(mu[j],covar[j,:,:],X_test[i,:])*prior[j])
    y_pred[i] = np.argmax(decision)

y_test = np.array(y_test)
test_accuracy = accuracy_score(y_test,y_pred)*100
print("test accuracy: " , test_accuracy)

# plot confusion matrix for test data
c_matrix  = confusion_matrix(y_test, y_pred)
fig, ax = plot_confusion_matrix(conf_mat=c_matrix,figsize=(7,7),cmap=plt.cm.RdYlBu_r)
ax.set(title = "Confusion Matrix")
plt.show()

# Decision Region Plots ######################################################
# define bounds of the domain
min1, max1 = X_train[:, 0].min()-1, X_train[:, 0].max()+1
min2, max2 = X_train[:, 1].min()-1, X_train[:, 1].max()+1

# define the x and y scale
x1_grid = np.arange(min1, max1, 0.1)
x2_grid = np.arange(min2, max2, 0.1)

x1_grid, x2_grid = np.meshgrid(x1_grid, x2_grid)

c1, c2 = x1_grid.flatten(), x1_grid.flatten()
c1, c2 = x1_grid.reshape((len(c1), 1)), x2_grid.reshape((len(c2), 1))

x = np.hstack((c1,c2))

y_pred = np.zeros((np.shape(x)[0],))
for i in range(np.shape(x)[0]):
    decision = []
    for j in range(0,np.shape(unique_class)[0]):
        decision.append(N(mu[j],covar[j,:,:],x[i,:])*prior[j])
    y_pred[i] = np.argmax(decision)

x3_grid = y_pred.reshape(x1_grid.shape)
```

```

fig = plt.figure()
ax = fig.add_subplot(111)
ax.contourf(x1_grid, x2_grid, x3_grid, cmap='Paired')
ax.scatter(X_train[:,0],X_train[:,1],marker='x')
ax.set_xlabel('x1',fontsize=20)
ax.set_ylabel('x2',fontsize=20)
ax.set_title('Naiyve Bayes Classifier', fontsize=20)

# plot levl of curves of the gaussian functions
for i in range(0,np.shape(unique_class)[0]):
    x3_grid = []
    for j in range(np.shape(x)[0]):
        x3_grid.append(N(mu[i],covar[i,:,:],x[j,:]))
    x3_grid = np.array(x3_grid)
    x3_grid = x3_grid.reshape(x1_grid.shape)
    contours = ax.contour(x1_grid, x2_grid, x3_grid, cmap='tab20b')
    ax.clabel(contours, inline=1, fontsize=5)
```

II Pattern classification on non-linearly separable data

II.1 K nearest Neighbours Method and Bayes classifier with KNN for density estimation

II.1.1 Python Code

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from mlxtend.plotting import plot_confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.cluster import KMeans
from scipy.stats import multivariate_normal
###############################################################################
# train data
f = open('datasets/Dataset_1b/train.csv', 'r')

length = 0
for line in f:

    d = [float(i) for i in line.split(',')]

    if length == 0:
        data = np.array(d)
        data = data[np.newaxis,:]
    else:
        d = np.array(d)
        d = d[np.newaxis,:]
        data = np.append(data,d,axis=0)
    length = length+1

f.close()

train_data = pd.DataFrame(data)

# shuffle dataset
# train_data = train_data.sample(frac=1)

# get train data
train_data = np.array(train_data)

# test data
f = open('datasets/Dataset_1b/dev.csv', 'r')

length = 0
for line in f:

    d = [float(i) for i in line.split(',')]

```

```
    if length == 0:
        data = np.array(d)
        data = data[np.newaxis,:]
    else:
        d = np.array(d)
        d = d[np.newaxis,:]
        data = np.append(data,d,axis=0)
    length = length+1

f.close()

test_data = pd.DataFrame(data)

# shuffle dataset
test_data = test_data.sample(frac=1)

# get test data
test_data = np.array(test_data)
################################################################################

# Split training data
# length of data for fit
train_len = int(np.shape(train_data)[0])
val_len = int(np.shape(test_data)[0]*0.5)
test_len = int(np.shape(test_data)[0]*0.5)

X_train = train_data[:,0:2]
X_val = test_data[0:val_len,0:2]
X_test = test_data[val_len:val_len+test_len,0:2]

y_train = train_data[:,2]
y_val = test_data[0:val_len,2]
y_test = test_data[val_len:val_len+test_len,2]

KNN classifier

# Build KNN classifier
y_pred = []
def KNN_classifier(K, x):
    """
    Parameters
    ----------
    K : value of nearest neighbours
    x : feature vector

    Returns
    -------
    None.
    """
    # find distance between feature vector and training data
```

```
    dist = np.linalg.norm(x-X_train,axis=1)

    # get the top index for the minimum distance
    min_dist_index = np.argsort(dist)
    topk = min_dist_index[0:K]

    # get class of corresponding class
    K_class = y_train[topk]

    # get count of each class
    unique_class, counts = np.unique(K_class, return_counts=1)

    # get the index of max counts
    max_count = np.argmax(counts)

    # choose that as the class
    y_pred.append(unique_class[max_count])

K = 15
# test on training data
for i in range(0, train_len):
    KNN_classifier(K, X_train[i])

y_pred = np.array(y_pred)

train_accuracy = accuracy_score(y_train,y_pred)*100
print("training accuracy: " , train_accuracy)

# plot confusion matrix for training data
c_matrix  = confusion_matrix(y_train, y_pred)
fig, ax = plot_confusion_matrix(conf_mat=c_matrix,figsize=(7,7),cmap=plt.cm.RdYlBu_r)
ax.set(title = "Confusion Matrix")
plt.show()

y_pred = []
# test on validation data
for i in range(0, val_len):
    KNN_classifier(K, X_val[i])

y_pred = np.array(y_pred)

val_accuracy = accuracy_score(y_val,y_pred)*100
print("validation accuracy: " , val_accuracy)

y_pred = []
# test on test data
for i in range(0, test_len):
    KNN_classifier(K, X_test[i])

y_pred = np.array(y_pred)
```

```python

test_accuracy = accuracy_score(y_test,y_pred)*100
print("test accuracy: " , test_accuracy)

# plot confusion matrix for test data
c_matrix  = confusion_matrix(y_test, y_pred)
fig, ax = plot_confusion_matrix(conf_mat=c_matrix,figsize=(7,7),cmap=plt.cm.RdYlBu_r)
ax.set(title = "Confusion Matrix")
plt.show()

# Decision Region Plots ######################################################
# define bounds of the domain
min1, max1 = X_train[:, 0].min()-1, X_train[:, 0].max()+1
min2, max2 = X_train[:, 1].min()-1, X_train[:, 1].max()+1

# define the x and y scale
x1_grid = np.arange(min1, max1, 0.1)
x2_grid = np.arange(min2, max2, 0.1)

x1_grid, x2_grid = np.meshgrid(x1_grid, x2_grid)

c1, c2 = x1_grid.flatten(), x1_grid.flatten()
c1, c2 = x1_grid.reshape((len(c1), 1)), x2_grid.reshape((len(c2), 1))

x = np.hstack((c1,c2))

y_pred = []
for i in range(0, np.shape(x)[0]):
    KNN_classifier(K, x[i,:])

y_pred = np.array(y_pred)

x3_grid = y_pred.reshape(x1_grid.shape)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.contourf(x1_grid, x2_grid, x3_grid, cmap='Paired')
ax.scatter(X_train[:,0],X_train[:,1],marker='x')
ax.scatter(X_test[:,0],X_test[:,1],marker='x')
ax.set_xlabel('x1',fontsize=20)
ax.set_ylabel('x2',fontsize=20)
ax.set_title('KNN model with K = 15', fontsize=20)

##############################################################
# Bayes classifier with KNN for density estimation

# Build KNN for density estimation
y_pred = []
def KNN(K, x, X):
    """
```

```
    Parameters
    ----------
    K : value of nearest neighbours
    x : feature vector
    X : training data related to particular class
    Returns
    -------
    None.
    """
    # find distance between feature vector and training data
    dist = np.linalg.norm(x-X,axis=1)

    # get the top k index for the minimum distance
    min_dist_index = np.argsort(dist)
    topk = min_dist_index[0:K]

    # radius is distance of kth neearest neighbour
    R = dist[topk[-1]]

    return R

# split into different classes
unique_class,counts = np.unique(y_train,return_counts=1)
total_class = len(unique_class)

class_data = []

for i in range(0,total_class):
    class_data.append(X_train[y_train==i])

K=20
# bayes classifier -> this is just the min value of R for all the classes
# upon simplification of the actual bayes theorem
# test on train data
y_pred = np.zeros((train_len,))
for i in range(train_len):
    decision = []
    for j in range(0,total_class):
        decision.append(KNN(K,X_train[i],class_data[j]))
    y_pred[i] = np.argmin(decision)

train_accuracy = accuracy_score(y_train,y_pred)*100
print("training accuracy: " , train_accuracy)

# plot confusion matrix for training data
c_matrix  = confusion_matrix(y_train, y_pred)
fig, ax = plot_confusion_matrix(conf_mat=c_matrix,figsize=(7,7),cmap=plt.cm.RdYlBu_r)
ax.set(title = "Confusion Matrix")
plt.show()
```

```python
# test on validation data
y_pred = np.zeros((val_len,))
for i in range(val_len):
    decision = []
    for j in range(0,total_class):
        decision.append(KNN(K,X_val[i],class_data[j]))
    y_pred[i] = np.argmin(decision)

val_accuracy = accuracy_score(y_val,y_pred)*100
print("validation accuracy: " , val_accuracy)

# test on test data
y_pred = np.zeros((test_len,))
for i in range(test_len):
    decision = []
    for j in range(0,total_class):
        decision.append(KNN(K,X_test[i],class_data[j]))
    y_pred[i] = np.argmin(decision)

y_test = np.array(y_test)
test_accuracy = accuracy_score(y_test,y_pred)*100
print("test accuracy: " , test_accuracy)

# plot confusion matrix for test data
c_matrix  = confusion_matrix(y_test, y_pred)
fig, ax = plot_confusion_matrix(conf_mat=c_matrix,figsize=(7,7),cmap=plt.cm.RdYlBu_r)
ax.set(title = "Confusion Matrix")
plt.show()


# Decision Region Plots ######################################################
# define bounds of the domain
min1, max1 = X_train[:, 0].min()-1, X_train[:, 0].max()+1
min2, max2 = X_train[:, 1].min()-1, X_train[:, 1].max()+1

# define the x and y scale
x1_grid = np.arange(min1, max1, 0.1)
x2_grid = np.arange(min2, max2, 0.1)

x1_grid, x2_grid = np.meshgrid(x1_grid, x2_grid)

c1, c2 = x1_grid.flatten(), x1_grid.flatten()
c1, c2 = x1_grid.reshape((len(c1), 1)), x2_grid.reshape((len(c2), 1))

x = np.hstack((c1,c2))

y_pred = np.zeros((np.shape(x)[0],))
for i in range(np.shape(x)[0]):
    decision = []
    for j in range(0,total_class):
```

```python
            decision.append(KNN(K,x[i],class_data[j]))
        y_pred[i] = np.argmin(decision)

x3_grid = y_pred.reshape(x1_grid.shape)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.contourf(x1_grid, x2_grid, x3_grid, cmap='Paired')
ax.scatter(X_train[:,0],X_train[:,1],marker='x')
ax.set_xlabel('x1',fontsize=20)
ax.set_ylabel('x2',fontsize=20)
ax.set_title('Bayes Classifier with KNN density estimation', fontsize=20)
```

II.2 Bayes Classifier with GMM

II.2.1 Python Code

```python
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns  #A statistical plotting library
from sklearn.cluster import KMeans
from kneed import KneeLocator  #A function that helps in optimization of
                               #number of clusters from an error curve
from scipy.stats import multivariate_normal as mvn
from mlxtend.plotting import plot_confusion_matrix
from sklearn.metrics import confusion_matrix


# In[2]:


header_names = ['x1', 'x2' , 'Class']
D = pd.read_csv('datasets/Dataset_1b/train.csv', header = None, names = header_names)
D.head()


# In[3]:


L_df = D.loc[:,['x1','x2']]
Unlab_Data = L_df.to_numpy()

lab_df = D.loc[:,'Class']
```

```
labels = lab_df.to_numpy()

#Training Dataset for Class 0
L0 = (D['Class'] == 0.0)
L0_df = D.loc[L0 , ['x1', 'x2'] ]
Class0 = L0_df.to_numpy()

#Training Dataset for Class 1
L1 = (D['Class'] == 1.0)
L1_df = D.loc[L1 , ['x1', 'x2']]
Class1 = L1_df.to_numpy()

#Training Dataset for Class 2
L2 = (D['Class'] == 2.0)
L2_df = D.loc[L2, ['x1', 'x2']]
Class2 = L2_df.to_numpy()
labels.shape


# In[4]:


#KMeans implementation for initialization and optimization of the number of
#clusters.
#Number of clusters for each class eqauls the number of gaussian componenets
#to be fitted for that class.
def K_Clustering(Class,M):
    #Dictionary of the arguments for scikit.KMeans
    KMeans_args = {
        "init" :"random",
        "n_init" : 10,
        "max_iter" : 300,
        "random_state" : 0,
        }
    #Estimation of the optimum number of clusters using elbow method
    std_error = []
    for cluster in range(1,11):
        kmeans = KMeans(n_clusters = cluster , **KMeans_args)
        kmeans.fit(Class)
        std_error.append(kmeans.inertia_)
    if M==0:
        #detecting the elbow point of the curve of 's_err vs K' using kneed, which
        #gives the optimum number of clusters
        curve = KneeLocator(range(1,11), std_error, curve="convex",
        direction = "decreasing")
        K_opt = curve.elbow
    else:
        #Using Manually entered value for K_opt
        K_opt = M
    #clustering the class in to K_opt clusters
```

```
    kmeans =  KMeans(n_clusters = K_opt , **KMeans_args)
    kmeans.fit(Class)
    labels = kmeans.labels_
    centers = kmeans.cluster_centers_
    return K_opt,labels,centers


# In[5]:

#initialization of the parameters using K-Clusters

def Parameters_old(Class,M):
    #Will return a mean(mu)-(K,d) array;
    N,d = np.shape(Class)
    K,lab,mu = K_Clustering(Class,M)
    #gamma contains initial responsibilty values for an example w.r.t
    #each clusters as columns
    gamma = np.array([ [0]*K for i in range(N)])
    for example in range(N):
        for cluster in range(K):
            if lab[example] == cluster:
                gamma[example][cluster]= 1
    return K,mu,gamma


# In[6]:



#Defining the Gaussian Mixture Model as a class

class Gaussian_Mixture_Model:
    #Class - Examples of the class to which the Gaussian Componenets
    #need to be fitted
    #Class - N x d matrix, where N is the number of examples and
    #d is the number of features for each example
    #K - Number of Gaussian Components that needs to be fitted

    def __init__(self,Class,K,MU,GAMMA,f):
        self.Class = Class
        self.K = K   #Attribute for Number of clusters
        self.GAMMA = GAMMA          #Attribute for NxK array of posterior
                                    #prob. / responsibity term.
        self.MU = MU                #Attribute for the mean values. An Kxd array.
        self.SIGMA = None           #Attribute for (K,d,d) array of covariances
        self.W = None               #Attribute for prior probabilty,
                                    #an array of length K
        #self.max_iter = max_iter    #Attribute for the number of iterations
        self.N = len(self.Class)    #Attribute for number of examples available
        self.d = len(self.Class[0]) #Attribute for the number of features
```

```
                                    #in each example
        self.f = f                  #Attribute that acts as switch between
                                    #diagonal and full covariance matrices
        self.mean_shift = np.reshape(self.Class, (self.N, 1, self.d) ) -
                                     np.reshape(self.MU, (1, self.K, self.d) )

    def Prior_Probability(self):
        #A function to estimate the (K,) array of prior prob.
        self.W = np.einsum("ij -> j",self.GAMMA) / self.N

    def Mean(self):
      # A function to calculate mean
      self.MU =  ((self.GAMMA).T) @ (self.Class) / np.reshape((self.W*self.N),
                  (self.K, 1))

    def Covariance_Matrix_Array(self):
        # A function to calculate covariances of the features of the examples

        Nk = np.einsum("ij -> j",self.GAMMA)
        self.mean_shift = np.reshape(self.Class, (self.N, 1, self.d) ) -
                                    np.reshape(self.MU, (1, self.K, self.d) )

        sigma = np.einsum("nki,nkj->kij", np.einsum("nk,nki->nki", self.GAMMA,
                          self.mean_shift), self.mean_shift) / np.reshape(Nk,
                                                                     (self.K, 1, 1))

        if self.f==1: #Case where we use full diagonal covariance matrix
            self.SIGMA = sigma

        if self.f==0: #Case where we use a diagonal covariance matrix
            I = np.identity(self.d,dtype=int) #An identity matrix of the size
            #equal to number of feature

            self.SIGMA = np.einsum("kij,ij -> kij",sigma,I)


    def Gaussian_Prob(self):
        #This function accounts for our assumption that the conditional
        #distribution of an example is a Gaussian.

        self.Covariance_Matrix_Array()           #SIGMA gets updated to the
                                                  #full covariance matrix
        SIGMA_inv = np.linalg.inv(self.SIGMA)     #Inverse of the covariance matrix

        #Normalisation term of the Gaussian dist.
        norm = np.sqrt(((2*np.pi)**self.d)*np.linalg.det(self.SIGMA))


        #Exponential term of the Gaussian
        expo = np.exp(-0.5*(np.einsum("nkj,nkj->nk", np.einsum("nki,kij->nkj",
```

```
                    self.mean_shift, SIGMA_inv),self.mean_shift)))

        #Prob_mat is an (NxK)-array that contains Gaussian Prob. of the
        #various examples to belong to respective clusters
        Prob_mat =  expo / norm
        return Prob_mat

    def Expectation_Step(self):
        #In this step we update the values of the responsibilty term

        N = self.Gaussian_Prob()
        #Prior probability array
        self.W =  np.einsum("ij -> j",self.GAMMA) / self.N
        Num =  N * self.W
        Den = np.reshape(np.sum(Num, axis=1), (self.N, 1) )
        self.GAMMA = Num/Den

    def Maximization_Step(self):
        #In this step we updtae the various parameters

        #Updation of GAMMA
        self.Expectation_Step()

        #Updation of W
        self.Prior_Probability()

        #Updation of Mean MU
        self.Mean()

        #Updation of Covariance Matrix SIGMA
        self.Covariance_Matrix_Array()


    def Log_Likelihood(self):

      llhd = np.sum(np.log(self.Gaussian_Prob() @ self.W))

      return llhd


    def fit(self,max_iter,threshold):

        log_likelihoods = []  #Attribute for 1D array that contains Log_Likelihood
                              # values.
                              #Size depends on the number iterations required
                              # to converge


        for i in range(max_iter):
            self.Expectation_Step()   #Updates Gamma
```

```
            self.Maximization_Step()  #Updates all the other parameters
            log_likelihoods.append(self.Log_Likelihood())
            #An if conditional for the requirement of convergence
            if (i!=0) & ((log_likelihoods[i] - log_likelihoods[i-1]) < threshold):
                    break

        print("Number of iterations to convegre:" ,i)

    def plot(self,ax,x1_grid,x2_grid):
        # #Plotting log_likelihood vs iterations, comment out if not needed
        # sns.set_style("darkgrid")          #setting the plot style
        # fig = plt.figure(figsize=(10,10))
        # ax0 = fig.add_subplot(111)
        # ax0.set_title('Log-Likelihood')
        # ax0.plot(range(i+1),log_likelihoods)

        #Plot of the fitted Gaussians for each class
        XY = np.array([x1_grid.flatten(),x2_grid.flatten()]).T

        for mu,sigma in zip(self.MU,self.SIGMA):
            multi_normal = mvn(mean=mu,cov=sigma)
            contours = ax.contour(x1_grid, x2_grid, multi_normal.pdf(XY).reshape(
            len(x1_grid),len(x1_grid)),cmap='hsv',levels=4,extend='min')
            ax.clabel(contours, inline=1, fontsize=5)

    def Class_Prob(self,Y):
            #A function that returns Prob.
            # for a unlabelled vector Y to belong to a class
            #Pred_Prob = []
            Multi_Gauss = []
            for mu,sigma in zip(self.MU,self.SIGMA):
                Multi_Gauss.append(mvn(mean=mu,cov=sigma).pdf(Y))
                #An array of Multi-Variate Gaussian Prob of various clusters
            Wt_Gauss = np.einsum("i,i->i",self.W,Multi_Gauss)
            #An array of weighted probabilities
            Pred_Prob =np.sum(Wt_Gauss)
            return Pred_Prob

# In[7]:


#Fitting gaussian mixtures for Class0
K,MU,GAMMA = Parameters_old(Class0,10)
#0 as the second argument chooses by default K_opt estimated using elbow method.
#If not pass the number of clusters needed

gmm0 = Gaussian_Mixture_Model(Class0,K,MU,GAMMA,1)
# 0 as the last argument -> diagonal covariance matrix.
# 1-> full covariance matix.
gmm0.fit(max_iter=100,threshold = 1e-10)
```

```
# In[8]:


#Fitting gaussian mixtures for Class1
K,MU,GAMMA = Parameters_old(Class1,10)
gmm1 = Gaussian_Mixture_Model(Class1,K,MU,GAMMA,1)
gmm1.fit(max_iter=100,threshold = 1e-10)


# In[9]:


#Fitting gaussian mixtures for Class2
K,MU,GAMMA = Parameters_old(Class2,10)
gmm2 = Gaussian_Mixture_Model(Class2,K,MU,GAMMA,1)
gmm2.fit(max_iter=100,threshold = 1e-10)


# In[10]:
l1 = len(Class0)
l2 = len(Class1)
l3 = len(Class2)
total = l1+l2+l3

prior = []
prior.append(l1/total)
prior.append(l2/total)
prior.append(l3/total)

# We have fitted gaussians to each class and now we would like to make prediction
# for unlabelled points
def Class_Prediction(Y):
    # gmm0,gmm1,gmm2 are the instances of class 0, class 1 and class 2 respectively
    n = len(Y) #number of unlabelled examples
    prediction = []
    for example in range(n):
        Prob=[]
        Prob = [gmm0.Class_Prob(Y[example,:])*prior[0], gmm1.Class_Prob(Y[example,:])
        prediction.append(np.argmax(Prob))
    # print("Labels for the given dataset:", prediction)
    return prediction

# In[11]:

header_names = ['x1', 'x2' , 'Class']
D = pd.read_csv('datasets/Dataset_1b/dev.csv', header = None, names = header_names)
D.head()
```

```python
# In[12]:

L_df = D.loc[:,['x1','x2']]
X_dev = L_df.to_numpy()

lab_df = D.loc[:,'Class']
y_dev = lab_df.to_numpy()

# In[12]:

# Divide into test and validation set
from sklearn.model_selection import train_test_split
X_val,X_test,y_val,y_test = train_test_split(X_dev,y_dev, test_size=0.5)

# In[14]:
from sklearn.metrics import accuracy_score
y_train = labels
X_train = np.concatenate((Class0,Class1,Class2))

predictions = Class_Prediction(X_train)

train_accuracy = accuracy_score(y_train,predictions)*100
print("train accuracy: " , train_accuracy)

# plot confusion matrix for training data
c_matrix  = confusion_matrix(y_train, predictions)
fig, ax = plot_confusion_matrix(conf_mat=c_matrix,figsize=(7,7),cmap=plt.cm.RdYlBu_r)
ax.set(title = "Confusion Matrix")
plt.show()

# In[15]:
predictions = Class_Prediction(X_val)

val_accuracy = accuracy_score(y_val,predictions)*100
print("val accuracy: " , val_accuracy)

# In[16]:
predictions = Class_Prediction(X_test)

test_accuracy = accuracy_score(y_test,predictions)*100
print("test accuracy: " , test_accuracy)

# plot confusion matrix for test data
c_matrix  = confusion_matrix(y_test, predictions)
fig, ax = plot_confusion_matrix(conf_mat=c_matrix,figsize=(7,7),cmap=plt.cm.RdYlBu_r)
ax.set(title = "Confusion Matrix")
plt.show()

# In[16]:

```

```python
# Plot decision surface with training data and GMM superimposed
# define bounds of the domain
min1, max1 = X_train[:, 0].min()-1, X_train[:, 0].max()+1
min2, max2 = X_train[:, 1].min()-1, X_train[:, 1].max()+1

# define the x and y scale
x1_grid = np.arange(min1, max1, 0.1)
x2_grid = np.arange(min2, max2, 0.1)

x1_grid, x2_grid = np.meshgrid(x1_grid, x2_grid)

c1, c2 = x1_grid.flatten(), x1_grid.flatten()
c1, c2 = x1_grid.reshape((len(c1), 1)), x2_grid.reshape((len(c2), 1))

x = np.hstack((c1,c2))

# predict
predictions = np.array(Class_Prediction(x))

x3_grid = predictions.reshape(x1_grid.shape)

# plot decision surfaxe
fig = plt.figure(figsize=[13,13])
ax = fig.add_subplot(111)
ax.contourf(x1_grid, x2_grid, x3_grid, cmap='Pastel1')
ax.scatter(X_train[:,0],X_train[:,1],marker='x')
gmm0.plot(ax,x1_grid,x2_grid) # call to plot gaussian functions of class 0
gmm1.plot(ax,x1_grid,x2_grid) # call to plot gaussian functions of class 1
gmm2.plot(ax,x1_grid,x2_grid) # call to plot gaussian functions of class 2
ax.set_xlabel('x1',fontsize=20)
ax.set_ylabel('x2',fontsize=20)
ax.set_title('Bayes Classifier with GMM', fontsize=20)
```

III Static Pattern Classification on Real World Dataset 2A

III.1 Python Code

```
import numpy as np
import pandas as pd
from sklearn import preprocessing
from sklearn.cluster import KMeans
from matplotlib import pyplot as plt
import math

#scaling factor
scalingfac = 1

#extracting and parsing data
#preprocessing : quantile transformation
f = pd.read_csv('Dataset_2A/coast/train.csv')
f1 = f.to_numpy()
data1 = f1[:,1:]
data1 = data1.astype('float')
data1 = np.apply_along_axis(lambda x:np.append(x,np.array([1,0,0,0,0])),1,data1)
X1 = data1[:,:-5]
X1 = scalingfac*X1
X1 = (preprocessing.QuantileTransformer(random_state=0)).fit_transform(X1)
f = pd.read_csv('Dataset_2A/forest/train.csv')
f1 = f.to_numpy()
data2 = f1[:,1:]
data2 = data2.astype('float')
data2 = np.apply_along_axis(lambda x:np.append(x,np.array([0,1,0,0,0])),1,data2)
X2 = data2[:,:-5]
X2 = scalingfac*X2
X2 = (preprocessing.QuantileTransformer(random_state=0)).fit_transform(X2)
f = pd.read_csv('Dataset_2A/mountain/train.csv')
f1 = f.to_numpy()
data3 = f1[:,1:]
data3 = data3.astype('float')
data3 = np.apply_along_axis(lambda x:np.append(x,np.array([0,0,1,0,0])),1,data3)
X3 = data3[:,:-5]
X3 = scalingfac*X3
X3 = (preprocessing.QuantileTransformer(random_state=0)).fit_transform(X3)
f = pd.read_csv('Dataset_2A/opencountry/train.csv')
f1 = f.to_numpy()
data4 = f1[:,1:]
data4 = data4.astype('float')
data4 = np.apply_along_axis(lambda x:np.append(x,np.array([0,0,0,1,0])),1,data4)
X4 = data4[:,:-5]
X4 = scalingfac*X4
X4 = (preprocessing.QuantileTransformer(random_state=0)).fit_transform(X4)
f = pd.read_csv('Dataset_2A/street/train.csv')
f1 = f.to_numpy()
```

```python
data5 = f1[:,1:]
data5 = data5.astype('float')
data5 = np.apply_along_axis(lambda x:np.append(x,np.array([0,0,0,0,1])),1,data5)
X5 = data5[:,:-5]
X5 = scalingfac*X5
X5 = (preprocessing.QuantileTransformer(random_state=0)).fit_transform(X5)

data = np.concatenate((data1,data2,data3,data4,data5),axis=0)

#prior probabilities
py1 = len(data1)/len(data)
py2 = len(data2)/len(data)
py3 = len(data3)/len(data)
py4 = len(data4)/len(data)
py5 = len(data5)/len(data)
ppy = np.array([py1,py2,py3,py4,py5])

#hyperparameters
threshold = 1e-12
nclasses = 5
#option for diagonal(opt=0) or full(opt=1) covariance matrix
opt = int(input('covariance'))
#number of clusters in each class
q = list(map(int,input('clusters').split(' ')))

# initialisation of parameters
fweights = {}
fmeans = {}
fvariances = {}

#GMM for each class
def GMMperclass(X,q,threshold,opt):  #X = data q = cluster #hyperparameter
    dim = X.shape[1]
    N = len(X)

    #kmeans clustering
    kmeans = KMeans(init='random',n_clusters=q,n_init=20,max_iter=800)
    kmeans.fit(X)
    response = np.zeros((N,q))
    labl = kmeans.labels_

    #gamma matrix
    for i in range(len(labl)):
        response[i,labl[i]] = 1

    #gaussian for individual datapoint
    def gauss(x,u,v):
        dim = len(x)
        num = ((np.reshape((x-u),(1,dim)))@((np.linalg.inv(v))@
                                    (np.reshape((x-u),(dim,1)))))
```

```
        num = -num/2
        den = np.sqrt(((2*np.pi)**dim)*(np.linalg.det(v)))
        return ((np.exp(num))/den)

    #gaussian for whole datapoints
    def gaussmat(X,q,means,variances):
        N,dim = X.shape
        #Inverse of the covariance matrix
        sigma_inv = np.linalg.inv(variances)
        mean_shift = np.reshape(X, (N, 1, dim) ) - np.reshape(means, (1,q, dim) )
        #Normalisation term of the Gaussian dist
        norm = np.sqrt(((2*np.pi)**dim)*np.linalg.det(variances))
        #Exponential term of the Gaussian
        expo = np.exp(-0.5*(np.einsum("nkj,nkj->nk", np.einsum("nki,kij->nkj",
                      mean_shift, sigma_inv),mean_shift)))
        return expo/norm

    #updating gamma=response update
    def responseupdate(weights,X,q,means,variances):
        N,dim = X.shape
        No = gaussmat(X,q,means,variances)
        Num =  No * weights
        Den = np.reshape(np.sum(Num, axis=1), (N, 1) )
        return Num/Den

    #estimation of log likelihoods
    def llhd(weights,X,q,means,variances):
        return np.sum(np.log(gaussmat(X,q,means,variances) @ weights))

    ##initialisation
    weights = np.einsum("ij -> j",response) / N
    means = (((response).T) @ X)/ np.reshape((weights*N), (q, 1))
    Nki = N*weights
    mean_shifti = np.reshape(X, (N, 1, dim) ) - np.reshape(means, (1, q, dim) )
    sigmai = np.einsum("nki,nkj->kij", np.einsum("nk,nki->nki", response,
                 mean_shifti), mean_shifti) / np.reshape(Nki, (q, 1, 1))

    #full covariance matrix
    if opt == 1:
        variances = sigmai
    #diagonal covariance matrix
    if opt == 0:
        I = np.identity(dim)
        variances = np.einsum("kij,ij -> kij",sigmai,I)

    #initial log likelihood
    NLL = llhd(weights,X,q,means,variances)
    nweights = weights
    nmeans = means
    nvariances = variances
```

```
    OLL = NLL+10
    ite = 0

    #EM maximisation
    while abs(NLL-OLL)>=threshold:
        OLL = NLL
        #gamma calculation
        nresponse = responseupdate(nweights,X,q,nmeans,nvariances)
        #updation
        nweights = np.einsum("ij -> j",nresponse)/N
        nmeans = (((nresponse).T) @X) / np.reshape((nweights*N), (q, 1))
        Nk = N*nweights
        mean_shift = np.reshape(X, (N, 1, dim) ) - np.reshape(nmeans, (1, q, dim) )
        sigma = np.einsum("nki,nkj->kij", np.einsum("nk,nki->nki",
                             nresponse, mean_shift), mean_shift) / np.reshape(Nk, (q, 1, 1))

        #full covariance matrix
        if opt == 1:
            nvariances = sigma

        #full diagonal matrix
        if opt == 0:
            I = np.identity(dim)
            nvariances = np.einsum("kij,ij -> kij",sigma,I)


        NLL = llhd(nweights,X,q,nmeans,nvariances)
        ite+=1
    print("iterations=%f"%ite)

    #final parameters
    fweights = nweights
    fmeans = nmeans
    fvariances = nvariances
    return fweights,fmeans,fvariances

##training

fweights[0],fmeans[0],fvariances[0] = GMMperclass(X1,q[0],threshold,opt)
fweights[1],fmeans[1],fvariances[1] = GMMperclass(X2,q[1],threshold,opt)
fweights[2],fmeans[2],fvariances[2] = GMMperclass(X3,q[2],threshold,opt)
fweights[3],fmeans[3],fvariances[3] = GMMperclass(X4,q[3],threshold,opt)
fweights[4],fmeans[4],fvariances[4] = GMMperclass(X5,q[4],threshold,opt)

#modeloutput

def bayesclf(x,ppy,nclasses,q,fweights,fmeans,fvariances):
    def gauss(x,u,v):
        dim = len(x)
        num = ((np.reshape((x-u),(1,dim)))@((np.linalg.inv(v))@
```

```
                                            (np.reshape((x-u),(dim,1)))))
            num = -num/2
            den = np.sqrt(((2*np.pi)**dim)*(np.linalg.det(v)))
            return ((np.exp(num))/den)
        def gaussmat(X,q,means,variances):
            N,dim = X.shape
            sigma_inv = np.linalg.inv(variances)
            mean_shift = np.reshape(X, (N, 1, dim) ) - np.reshape(means, (1,q, dim) )
            norm = np.sqrt(((2*np.pi)**dim)*np.linalg.det(variances))
            expo = np.exp(-0.5*(np.einsum("nkj,nkj->nk", np.einsum("nki,kij->nkj",
                                                    mean_shift, sigma_inv),mean_shift)))
            return expo/norm
        pxy = np.zeros((nclasses,1))
        pyx = np.zeros((nclasses,1))
        res = np.zeros((nclasses,1))
        for i in range(nclasses):
            pxy[i] = np.sum((gaussmat(np.reshape(x,(1,len(x))),q[i],fmeans[i],
                                                    fvariances[i]))*(fweights[i]))
        for i in range(nclasses):
            pyx[i] = pxy[i]*ppy[i]
        pyx = pyx/np.sum(pyx)
        res[np.argmax(pyx)] = 1
        return np.transpose(res)

#train labels
ytr = data[:,-5:]
#predicted train labels
ytrp = np.apply_along_axis(lambda x:bayesclf(x,ppy,nclasses,q,fweights,fmeans,
                        fvariances),1,np.concatenate((X1,X2,X3,X4,X5),axis=0))
c = 0
for i in range(len(ytr)):
    if np.linalg.norm((ytrp[i,:]-ytr[i,:])) < 1:
        c +=1
print('training accuracy = %f'%(c/len(ytr)))

###########test

#extracting test data
#preprocessing : quantile transformation
f = pd.read_csv('Dataset_2A/coast/dev.csv')
f1 = f.to_numpy()
datat1 = f1[:,1:]
datat1 = datat1.astype('float')
datat1 = np.apply_along_axis(lambda x:np.append(x,np.array([1,0,0,0,0])),1,datat1)
Xt1 = datat1[:,:-5]
Xt1 = scalingfac*Xt1
Xt1 = ((preprocessing.QuantileTransformer(random_state=0)).fit(Xt1)).transform(Xt1)
f = pd.read_csv('Dataset_2A/forest/dev.csv')
f1 = f.to_numpy()
datat2 = f1[:,1:]
```

```
datat2 = datat2.astype('float')
datat2 = np.apply_along_axis(lambda x:np.append(x,np.array([0,1,0,0,0])),1,datat2)
Xt2 = datat2[:,:-5]
Xt2 = scalingfac*Xt2
Xt2 = ((preprocessing.QuantileTransformer(random_state=0)).fit(Xt2)).transform(Xt2)
f = pd.read_csv('Dataset_2A/mountain/dev.csv')
f1 = f.to_numpy()
datat3 = f1[:,1:]
datat3 = datat3.astype('float')
datat3 = np.apply_along_axis(lambda x:np.append(x,np.array([0,0,1,0,0])),1,datat3)
Xt3 = datat3[:,:-5]
Xt3 = scalingfac*Xt3
Xt3 = ((preprocessing.QuantileTransformer(random_state=0)).fit(Xt3)).transform(Xt3)
f = pd.read_csv('Dataset_2A/opencountry/dev.csv')
f1 = f.to_numpy()
datat4 = f1[:,1:]
datat4 = datat4.astype('float')
datat4 = np.apply_along_axis(lambda x:np.append(x,np.array([0,0,0,1,0])),1,datat4)
Xt4 = datat4[:,:-5]
Xt4 = scalingfac*Xt4
Xt4 = ((preprocessing.QuantileTransformer(random_state=0)).fit(Xt4)).transform(Xt4)
f = pd.read_csv('Dataset_2A/street/dev.csv')
f1 = f.to_numpy()
datat5 = f1[:,1:]
datat5 = datat5.astype('float')
datat5 = np.apply_along_axis(lambda x:np.append(x,np.array([0,0,0,0,1])),1,datat5)
Xt5 = datat5[:,:-5]
Xt5 = scalingfac*Xt5
Xt5 = ((preprocessing.QuantileTransformer(random_state=0)).fit(Xt5)).transform(Xt5)

datat = np.concatenate((datat1,datat2,datat3,datat4,datat5),axis=0)
Xt = np.concatenate((Xt1,Xt2,Xt3,Xt4,Xt5),axis=0)
datat = np.concatenate((Xt,datat[:,-5:]),axis=1)

#test labels
yte = datat[:,-5:]

#predicted test labels
ytep = np.apply_along_axis(lambda x:bayesclf(x,ppy,nclasses,q,fweights,fmeans,
                           fvariances),1,datat[:,:-5])

cte = 0

for i in range(len(ytep)):
    if np.linalg.norm((ytep[i,:]-yte[i,:])) < 1:
        cte +=1

print('test accuracy = %f'%(cte/len(ytep)))
```

```python
#plotting confusion matrix
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

def clf_label(res):
    if np.argmax(res) == 0:
        return 'coast'
    if np.argmax(res) == 1:
        return 'forest'
    if np.argmax(res) == 2:
        return 'mountain'
    if np.argmax(res) == 3:
        return 'opencountry'
    if np.argmax(res) == 4:
        return 'street'

train_confusion = confusion_matrix([clf_label(ytr[i,:]) for i in range(len(ytr))],
                    [clf_label(ytrp[i,:]) for i in range(len(ytrp))],labels =
                    ['coast','forest','mountain','opencountry','street'])

test_confusion = confusion_matrix([clf_label(yte[i,:]) for i in range(len(yte))],
                        [clf_label(ytep[i,:]) for i in range(len(ytep))],labels =
                        ['coast','forest','mountain','opencountry','street'])

train_tab = ConfusionMatrixDisplay(train_confusion,display_labels =
['coast','forest','mountain','opencountry','street'] )
plt.figure(1)
train_tab.plot()
plt.title('Training Data')
test_tab = ConfusionMatrixDisplay(test_confusion,display_labels =
['coast','forest','mountain','opencountry','street'] )
plt.figure(2)
test_tab.plot()
plt.title('Test Data')
```

IV Static Pattern Classification on Real World Dataset 2B

IV.1 Python Code

```
import numpy as np
import pandas as pd
import os
from sklearn import preprocessing
from sklearn.cluster import KMeans
from matplotlib import pyplot as plt

#extracting and parsing files tr
#preprocessing : quantile transformation
directory = 'Dataset_2B/coast/train'
dataX = []
for filename in os.listdir(directory):
    f = open(directory+'/'+filename)
    data =[]
    for line in f:
        data.append([float(x) for x in line.strip().split(' ')])
    dataX+=(data)
data1 = np.array(dataX)
data1 = np.apply_along_axis(lambda x:np.append(x,np.array([1,0,0,0,0])),1,data1)
X1 = data1[:,:-5]
X1 = (preprocessing.QuantileTransformer(random_state=0)).fit_transform(X1)
directory = 'Dataset_2B/forest/train'
dataX = []
for filename in os.listdir(directory):
    f = open(directory+'/'+filename)
    data =[]
    for line in f:
        data.append([float(x) for x in line.strip().split(' ')])
    dataX+=(data)
data2 = np.array(dataX)
data2 = np.apply_along_axis(lambda x:np.append(x,np.array([0,1,0,0,0])),1,data2)
X2 = data2[:,:-5]
X2 = (preprocessing.QuantileTransformer(random_state=0)).fit_transform(X2)
directory = 'Dataset_2B/mountain/train'
dataX = []
for filename in os.listdir(directory):
    f = open(directory+'/'+filename)
    data =[]
    for line in f:
        data.append([float(x) for x in line.strip().split(' ')])
    dataX+=(data)
data3 = np.array(dataX)
data3 = np.apply_along_axis(lambda x:np.append(x,np.array([0,0,1,0,0])),1,data3)
X3 = data3[:,:-5]
X3 = (preprocessing.QuantileTransformer(random_state=0)).fit_transform(X3)
directory = 'Dataset_2B/opencountry/train'
```

```
dataX = []
for filename in os.listdir(directory):
    f = open(directory+'/'+filename)
    data =[]
    for line in f:
        data.append([float(x) for x in line.strip().split(' ')])
    dataX+=(data)
data4 = np.array(dataX)
data4 = np.apply_along_axis(lambda x:np.append(x,np.array([0,0,0,1,0])),1,data4)
X4 = data4[:,:-5]
X4 = (preprocessing.QuantileTransformer(random_state=0)).fit_transform(X4)
directory = 'Dataset_2B/street/train'
dataX = []
for filename in os.listdir(directory):
    f = open(directory+'/'+filename)
    data =[]
    for line in f:
        data.append([float(x) for x in line.strip().split(' ')])
    dataX+=(data)
data5 = np.array(dataX)
data5 = np.apply_along_axis(lambda x:np.append(x,np.array([0,0,0,0,1])),1,data5)
X5 = data5[:,:-5]
X5 = (preprocessing.QuantileTransformer(random_state=0)).fit_transform(X5)

data = np.concatenate((data1,data2,data3,data4,data5),axis=0)

#prior probabilities
py1 = len(data1)/len(data)
py2 = len(data2)/len(data)
py3 = len(data3)/len(data)
py4 = len(data4)/len(data)
py5 = len(data5)/len(data)
ppy = np.array([py1,py2,py3,py4,py5])

#hyperparameters
threshold = 1e-10
nclasses = 5
#option for diagonal(opt=0) or full(opt=1) covariance matrix
opt = int(input('covariance'))
#number of clusters in each class
q = list(map(int,input('clusters').split(' ')))

# initialisation of parameters
fweights = {}
fmeans = {}
fvariances = {}
#GMM for each class
def GMMperclass(X,q,threshold,opt):  #X = data q = cluster #hyperparameter
    dim = X.shape[1]
    N = len(X)
```

```
    #kmeans clustering
    kmeans = KMeans(init='random',n_clusters=q,n_init=20,max_iter=800)
    kmeans.fit(X)
    response = np.zeros((N,q))
    labl = kmeans.labels_

    #gamma matrix
    for i in range(len(labl)):
        response[i,labl[i]] = 1

    #gaussian for individual datapoint
    def gauss(x,u,v):
        dim = len(x)
        num = ((np.reshape((x-u),(1,dim)))@((np.linalg.inv(v))@
                                   (np.reshape((x-u),(dim,1)))))
        num = -num/2
        den = np.sqrt(((2*np.pi)**dim)*(np.linalg.det(v)))
        return ((np.exp(num))/den)

    #gaussian for whole datapoints
    def gaussmat(X,q,means,variances):
        N,dim = X.shape
        #Inverse of the covariance matrix
        sigma_inv = np.linalg.inv(variances)
        mean_shift = np.reshape(X, (N, 1, dim) ) - np.reshape(means, (1,q, dim) )
        #Normalisation term of the Gaussian dist
        norm = np.sqrt(((2*np.pi)**dim)*np.linalg.det(variances))
        #Exponential term of the Gaussian
        expo = np.exp(-0.5*(np.einsum("nkj,nkj->nk", np.einsum("nki,kij->nkj",
                                         mean_shift, sigma_inv),mean_shift)))
        return expo/norm

    #updating gamma=response update
    def responseupdate(weights,X,q,means,variances):
        N,dim = X.shape
        No = gaussmat(X,q,means,variances)
        Num =  No * weights
        Den = np.reshape(np.sum(Num, axis=1), (N, 1) )
        return Num/Den

    #estimation of log likelihoods
    def llhd(weights,X,q,means,variances):
        return np.sum(np.log(gaussmat(X,q,means,variances) @ weights))

    ##initialisation
    weights = np.einsum("ij -> j",response) / N
    means = (((response).T) @ X)/ np.reshape((weights*N), (q, 1))
    Nki = N*weights
    mean_shifti = np.reshape(X, (N, 1, dim) ) - np.reshape(means, (1, q, dim) )
```

```
    sigmai = np.einsum("nki,nkj->kij", np.einsum("nk,nki->nki", response,
                     mean_shifti), mean_shifti) / np.reshape(Nki, (q, 1, 1))

    #full covariance matrix
    if opt == 1:
        variances = sigmai
    #diagonal covariance matrix
    if opt == 0:
        I = np.identity(dim)
        variances = np.einsum("kij,ij -> kij",sigmai,I)

    #initial log likelihood
    NLL = llhd(weights,X,q,means,variances)
    nweights = weights
    nmeans = means
    nvariances = variances
    OLL = NLL+10
    ite = 0

    #EM maximisation
    while abs(NLL-OLL)>=threshold:
        OLL = NLL
        #gamma calculation
        nresponse = responseupdate(nweights,X,q,nmeans,nvariances)
        #updation
        nweights = np.einsum("ij -> j",nresponse)/N
        nmeans = (((nresponse).T) @X) / np.reshape((nweights*N), (q, 1))
        Nk = N*nweights
        mean_shift = np.reshape(X, (N, 1, dim) ) - np.reshape(nmeans, (1, q, dim) )
        sigma = np.einsum("nki,nkj->kij", np.einsum("nk,nki->nki",
                nresponse, mean_shift),  mean_shift)/ np.reshape(Nk, (q, 1, 1))

        #full covariance matrix
        if opt == 1:
            nvariances = sigma

        #full diagonal matrix
        if opt == 0:
            I = np.identity(dim)
            nvariances = np.einsum("kij,ij -> kij",sigma,I)


        NLL = llhd(nweights,X,q,nmeans,nvariances)
        ite+=1
    print("iterations=%f"%ite)

    #final parameters
    fweights = nweights
    fmeans = nmeans
    fvariances = nvariances
```

```
    return fweights,fmeans,fvariances

##training

fweights[0],fmeans[0],fvariances[0] = GMMperclass(X1,q[0],threshold,opt)
fweights[1],fmeans[1],fvariances[1] = GMMperclass(X2,q[1],threshold,opt)
fweights[2],fmeans[2],fvariances[2] = GMMperclass(X3,q[2],threshold,opt)
fweights[3],fmeans[3],fvariances[3] = GMMperclass(X4,q[3],threshold,opt)
fweights[4],fmeans[4],fvariances[4] = GMMperclass(X5,q[4],threshold,opt)

# classification for variable length features

def bayesclfvarlength(X,ppy,nclasses,q,fweights,fmeans,fvariances):
    def gauss(x,u,v):
        dim = len(x)
        num = ((np.reshape((x-u),(1,dim)))@((np.linalg.inv(v))@
                                    (np.reshape((x-u),(dim,1)))))
        num = -num/2
        den = np.sqrt(((2*np.pi)**dim)*(np.linalg.det(v)))
        return ((np.exp(num))/den)
    def gaussmat(X,q,means,variances):
        N,dim = X.shape
        sigma_inv = np.linalg.inv(variances)      #Inverse of the covariance matrix
        mean_shift = np.reshape(X, (N, 1, dim) ) - np.reshape(means, (1,q, dim) )
        norm = np.sqrt(((2*np.pi)**dim)*np.linalg.det(variances))
        #Normalisation term of the Gaussian dist.
        #Exponential term of the Gaussian
        expo = np.exp(-0.5*(np.einsum("nkj,nkj->nk", np.einsum("nki,kij->nkj",
                                            mean_shift, sigma_inv),mean_shift)))
        return expo/norm
    pxy = np.zeros((nclasses,1))
    pyx = np.zeros((nclasses,1))
    res = np.zeros((nclasses,1))
    for i in range(nclasses):
        pxy[i] = np.prod(np.sum(gaussmat(X,q[i],fmeans[i],fvariances[i])*
                                fweights[i],axis=1))
    for i in range(nclasses):
        pyx[i] = pxy[i]*ppy[i]
    pyx = pyx/np.sum(pyx)
    #res[np.argmax(pyx)] = 1
    #return np.transpose(res)
    return np.argmax(pyx)

#prediction of training labels
#reorganising of data points to images
Xn1 = []
for i in range(int(X1.shape[0]/36)):
    Xn1.append(X1[(36*i):(36*(i+1)),:])
Xn1 = np.array(Xn1)
Xn2 = []
```

```
for i in range(int(X2.shape[0]/36)):
    Xn2.append(X2[(36*i):(36*(i+1)),:])
Xn2 = np.array(Xn2)
Xn3 = []
for i in range(int(X3.shape[0]/36)):
    Xn3.append(X3[(36*i):(36*(i+1)),:])
Xn3 = np.array(Xn3)
Xn4 = []
for i in range(int(X4.shape[0]/36)):
    Xn4.append(X4[(36*i):(36*(i+1)),:])
Xn4 = np.array(Xn4)
Xn5 = []
for i in range(int(X5.shape[0]/36)):
    Xn5.append(X5[(36*i):(36*(i+1)),:])
Xn5 = np.array(Xn5)



c = 0
for i in range(Xn1.shape[0]):
    if bayesclfvarlength(Xn1[i,:,:],ppy,nclasses,q,fweights,fmeans,fvariances) == 0:
        c += 1
for i in range(Xn2.shape[0]):
    if bayesclfvarlength(Xn2[i,:,:],ppy,nclasses,q,fweights,fmeans,fvariances) == 1:
        c += 1
for i in range(Xn3.shape[0]):
    if bayesclfvarlength(Xn3[i,:,:],ppy,nclasses,q,fweights,fmeans,fvariances) == 2:
        c += 1
for i in range(Xn4.shape[0]):
    if bayesclfvarlength(Xn4[i,:,:],ppy,nclasses,q,fweights,fmeans,fvariances) == 3:
        c += 1
for i in range(Xn5.shape[0]):
    if bayesclfvarlength(Xn5[i,:,:],ppy,nclasses,q,fweights,fmeans,fvariances) == 4:
        c += 1

print('training accuracy =
  %f'%(c/((X1.shape[0]+X2.shape[0]+X3.shape[0]+X4.shape[0]+X5.shape[0])/36)))



#####testing

directory = 'Dataset_2B/coast/dev'
dataX = []
for filename in os.listdir(directory):
    f = open(directory+'/'+filename)
    data =[]
    for line in f:
        data.append([float(x) for x in line.strip().split(' ')])
    dataX+=(data)
```

```
datat1 = np.array(dataX)
datat1 = np.apply_along_axis(lambda x:np.append(x,np.array([1,0,0,0,0])),1,datat1)
Xt1 = datat1[:,:-5]
Xt1 = ((preprocessing.QuantileTransformer(random_state=0)).fit(Xt1)).transform(Xt1)
directory = 'Dataset_2B/forest/dev'
dataX = []
for filename in os.listdir(directory):
    f = open(directory+'/'+filename)
    data =[]
    for line in f:
        data.append([float(x) for x in line.strip().split(' ')])
    dataX+=(data)
datat2 = np.array(dataX)
datat2 = np.apply_along_axis(lambda x:np.append(x,np.array([0,1,0,0,0])),1,datat2)
Xt2 = datat2[:,:-5]
Xt2= ((preprocessing.QuantileTransformer(random_state=0)).fit(Xt2)).transform(Xt2)
directory = 'Dataset_2B/mountain/dev'
dataX = []
for filename in os.listdir(directory):
    f = open(directory+'/'+filename)
    data =[]
    for line in f:
        data.append([float(x) for x in line.strip().split(' ')])
    dataX+=(data)
datat3 = np.array(dataX)
datat3 = np.apply_along_axis(lambda x:np.append(x,np.array([0,0,1,0,0])),1,datat3)
Xt3 = datat3[:,:-5]
Xt3= ((preprocessing.QuantileTransformer(random_state=0)).fit(Xt3)).transform(Xt3)
directory = 'Dataset_2B/opencountry/dev'
dataX = []
for filename in os.listdir(directory):
    f = open(directory+'/'+filename)
    data =[]
    for line in f:
        data.append([float(x) for x in line.strip().split(' ')])
    dataX+=(data)
datat4 = np.array(dataX)
datat4 = np.apply_along_axis(lambda x:np.append(x,np.array([0,0,0,1,0])),1,datat4)
Xt4 = datat4[:,:-5]
Xt4= ((preprocessing.QuantileTransformer(random_state=0)).fit(Xt4)).transform(Xt4)
directory = 'Dataset_2B/street/dev'
dataX = []
for filename in os.listdir(directory):
    f = open(directory+'/'+filename)
    data =[]
    for line in f:
        data.append([float(x) for x in line.strip().split(' ')])
    dataX+=(data)
datat5 = np.array(dataX)
datat5 = np.apply_along_axis(lambda x:np.append(x,np.array([0,0,0,0,1])),1,datat5)
```

```
Xt5 = datat5[:,:-5]
Xt5= ((preprocessing.QuantileTransformer(random_state=0)).fit(Xt5)).transform(Xt5)


#prediction of test labels
#reorganising of data points to images
Xtn1 = []
for i in range(int(Xt1.shape[0]/36)):
    Xtn1.append(Xt1[(36*i):(36*(i+1)),:])
Xtn1 = np.array(Xtn1)
Xtn2 = []
for i in range(int(Xt2.shape[0]/36)):
    Xtn2.append(Xt2[(36*i):(36*(i+1)),:])
Xtn2 = np.array(Xtn2)
Xtn3 = []
for i in range(int(Xt3.shape[0]/36)):
    Xtn3.append(Xt3[(36*i):(36*(i+1)),:])
Xtn3 = np.array(Xtn3)
Xtn4 = []
for i in range(int(Xt4.shape[0]/36)):
    Xtn4.append(Xt4[(36*i):(36*(i+1)),:])
Xtn4 = np.array(Xtn4)
Xtn5 = []
for i in range(int(Xt5.shape[0]/36)):
    Xtn5.append(Xt5[(36*i):(36*(i+1)),:])
Xtn5 = np.array(Xtn5)



ct = 0
for i in range(Xtn1.shape[0]):
    if bayesclfvarlength(Xtn1[i,:,:],ppy,nclasses,q,fweights,fmeans,fvariances) == 0:
        ct += 1
for i in range(Xtn2.shape[0]):
    if bayesclfvarlength(Xtn2[i,:,:],ppy,nclasses,q,fweights,fmeans,fvariances) == 1:
        ct += 1
for i in range(Xtn3.shape[0]):
    if bayesclfvarlength(Xtn3[i,:,:],ppy,nclasses,q,fweights,fmeans,fvariances) == 2:
        ct += 1
for i in range(Xtn4.shape[0]):
    if bayesclfvarlength(Xtn4[i,:,:],ppy,nclasses,q,fweights,fmeans,fvariances) == 3:
        ct += 1
for i in range(Xtn5.shape[0]):
    if bayesclfvarlength(Xtn5[i,:,:],ppy,nclasses,q,fweights,fmeans,fvariances) == 4:
        ct += 1

print('test accuracy
= %f'%(ct/((Xt1.shape[0]+Xt2.shape[0]+Xt3.shape[0]+Xt4.shape[0]+Xt5.shape[0])/36)))

##confusion matrix
```

```python
img_labels = ['coast','forest','mountain','opencountry','street']
yt = []
ytp = []
for i in range(Xn1.shape[0]):
    yt.append(0)
    ytp.append(bayesclfvarlength(Xn1[i,:,:],ppy,nclasses,q,fweights,fmeans,
                fvariances))
for i in range(Xn2.shape[0]):
    yt.append(1)
    ytp.append(bayesclfvarlength(Xn2[i,:,:],ppy,nclasses,q,fweights,fmeans,
               fvariances))
for i in range(Xn3.shape[0]):
    yt.append(2)
    ytp.append(bayesclfvarlength(Xn3[i,:,:],ppy,nclasses,q,fweights,fmeans,
              fvariances))
for i in range(Xn4.shape[0]):
    yt.append(3)
    ytp.append(bayesclfvarlength(Xn4[i,:,:],ppy,nclasses,q,fweights,fmeans,
              fvariances))
for i in range(Xn5.shape[0]):
    yt.append(4)
    ytp.append(bayesclfvarlength(Xn5[i,:,:],ppy,nclasses,q,fweights,fmeans,
               fvariances))

yte = []
ytep = []
for i in range(Xtn1.shape[0]):
    yte.append(0)
    ytep.append(bayesclfvarlength(Xtn1[i,:,:],ppy,nclasses,q,fweights,fmeans,
               fvariances))
for i in range(Xtn2.shape[0]):
    yte.append(1)
    ytep.append(bayesclfvarlength(Xtn2[i,:,:],ppy,nclasses,q,fweights,fmeans,
               fvariances))
for i in range(Xtn3.shape[0]):
    yte.append(2)
    ytep.append(bayesclfvarlength(Xtn3[i,:,:],ppy,nclasses,q,fweights,fmeans,
                 fvariances))
for i in range(Xtn4.shape[0]):
    yte.append(3)
    ytep.append(bayesclfvarlength(Xtn4[i,:,:],ppy,nclasses,q,fweights,fmeans,
                fvariances))
for i in range(Xtn5.shape[0]):
    yte.append(4)
    ytep.append(bayesclfvarlength(Xtn5[i,:,:],ppy,nclasses,q,fweights,fmeans,
                fvariances))

from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

train_confusion = confusion_matrix(yt,ytp)
```

```
test_confusion = confusion_matrix(yte,ytep)

train_tab = ConfusionMatrixDisplay(train_confusion,display_labels =
['coast','forest','mountain','opencountry','street'] )
plt.figure(1)
train_tab.plot()
plt.title('Training Data')
test_tab = ConfusionMatrixDisplay(test_confusion,display_labels =
['coast','forest','mountain','opencountry','street'] )
plt.figure(2)
test_tab.plot()
plt.title('Test Data')
```