SPR Assignment 2

Jayanth S (201081003), Praveen Kumar N (201082001), Rishabh Roy (201082002)

March 17, 2021

[2]:
```
from mnist import MNIST
import os
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stat
from sklearn.metrics import confusion_matrix
import seaborn as sn
import pandas as pd
import cv2
```

0.0.1 RANDOM INITIALISATION OF PARAMETERS

[3]:
```
def initialisation_parameters(K_init,img_size):
    """
    Initialise the parameters for Bernoulli mixture model

    Input : K_init -- No. of mixture components that density is assumed to be␣
 ↪made up of

    P_init -- Create a dictonary consisting of the inital probabilities and␣
 ↪weightage of
    each mixture component corresponding to each class

    Return : P_init

    """
    K=K_init   #no. of mixture components
    P_init={}



    #-------------------------------------------------------------------------
    #to randomly assign the inital probabilities

    pi=np.ones((1,K))*(1/K)   #np.random.dirichlet(alpha)  #to genereate lamda␣
 ↪values such that they sum to 1
    P_init["lamda"]=pi[0,:]
```

```
    for k in range(K):
        #alpha=np.random.randint(1,5,size=(K,))
        #alpha=np.reshape(alpha,(3,))


        #np.random.rand(1,img_size)
        p=np.random.uniform(0.25,0.75,(1,img_size)) #create random prob. values␣
↪for all mixture components
        p=p/np.sum(p)
        #p=(pi@p).T

        P_init["prob"+str(k)]=p

    return P_init
```

0.0.2 EM ALGORITHM

[4]:
```
def em_algorithm(iterations,P_em,K,clas):

    """EM ALGORITHM
    Input :
    iterations -- Maximum no. of iterations
    P_em       -- dictionary that contains probabilities and weightages of all␣
↪mixture components
    K          -- no. of mixture components
    res_coeff  -- dictionary containing the responsibility coefficient of all␣
↪images w.r.t each mixture component
    log_likey  -- contains log likelihood of each class for each iteration

    Return:
    P_em -- Updated dictionary that contains probabilities and weightages of all␣
↪mixture components of each class
    res_coeff -- Updated responsibility coefficients
    log_likely --  Updated log likelihood
    """


    ll_old=0
    ll_new=0
    log_likely=np.zeros((iterations,1))

    for itr in range(iterations):
        #responsibility coefficient of size(no. of images in a class x mixture␣
↪components)
        res_coeff=np.zeros((len(clas),K))
```

```
        for n in range(len(clas)):
            prob_pk=np.zeros((1,K))              #probability of each mixture␣
↪component
            res_coeff_nr=np.zeros((1,K))        #to store respob. coeff of each␣
↪mixture component

            for k in range(K):
                xn=clas[n][:]
                pk=P_em["prob"+str(k)]  #stores the probability of kth component

                prob_xp1=np.power(pk,xn)             #to find pi^xi
                #prob_xp1=np.clip(prob_xp1,1e-50,1)  #clip the values that are␣
↪lesser than 1e-50
                prob_xp2=np.power(1-pk,1-xn)         #to find (1-pi)^(1-xi)
                #prob_xp2=np.clip(prob_xp2,1e-50,1)  #clip the values that are␣
↪lesser than 1e-50

                """when we muliply pi^xi*(1-pi)^(1-xi) the values would be very␣
↪low
                ex: if all pi=0.5 then  pi^xi*(1-pi)^(1-xi)=0.5^784=o(1e^-237),␣
↪which will be close to zero.
                Hence we multiply it by 1e250. As we are doing this for each␣
↪mixture component. It doesn't affect
                in the calculations of responsibility coefficient.
                We have done this to avoid res_coefficient becoming undefined␣
↪value or zero for all features of a image"""
                prob_pk[0,k]=np.product(prob_xp1)*np.product(prob_xp2)*(1e250) ␣
↪#probability of x given prob. of kth mixture


                #numerator of respob. coeff of nth image with kth mixture
                res_coeff_nr[0,k]=P_em["lamda"][k]*prob_pk[0,k]
                #print(f"respb coeff nr {res_coeff_nr[0,k]}")

            #print(f"resolution coeff {np.sum(res_coeff_nr)}")

            # respob. coeff of nth image with kth mixture
            #added 1e-50 to the denominator so that responsibility coefficient␣
↪doesn't become undefined value (0/0 form)
            res_coeff[n,:]=res_coeff_nr/(np.sum(res_coeff_nr)+1e-50)

        #EXPECTATION - STEP
        N_k=np.sum(res_coeff,axis=0)
        #print(f"Value of {N_k}")

        P_em["lamda"]=N_k/len(clas)
```

```
        k=0
        for k in range(K):

            res_coeff_k=res_coeff[:,k]
            res_coeff_k=np.reshape(res_coeff_k,(1,len(clas)))

            #MAXIMISATION - STEP
            #added 1e-50 to the denominator so that responsibility coefficient␣
 ↪doesn't become undefined value (0/0 form)

            P_em["prob"+str(k)]=(res_coeff_k@clas[:][:])/(N_k[k]+1e-50)

        """Below code is find the log likelihood for the parameters obtained in␣
 ↪the current iteration"""




        ll=0
        for n in range(len(clas)):
            xi=clas[n][:]
            xi=np.reshape(xi,(1,784))
            for k in range(K):
                p_i=P_em["prob"+str(k)]
                p_i=np.clip(p_i,1e-50,1)
                p_i_minus=np.log(1-p_i)
                p_i=np.log(p_i)
                ll_update=res_coeff[n,k]*(xi@p_i.T+(1-xi)@(p_i_minus).T+np.
 ↪log(P_em["lamda"][k]))
                ll=ll+ll_update

        ll_old=ll_new
        ll_new=ll
#       print(f"log likelihood : {ll_new},old : {ll_old}")

        log_likely[itr]=ll_new



        if (np.abs(ll_new-ll_old)<1 and itr>=1):
            #log_likely[itr+1:iterations]=log_likely[itr]
            print(f"For iteration {itr} the difference between new log␣
 ↪likelihood and old log likelihood is less than 1")
            break

    return log_likely,P_em,res_coeff,itr
```

0.0.3 Initialisation of parameters for EM algorithm using K-means

[5]:
```
def init_using_kmeans(K,iter_kmeans):
    K=K
    iterations=iter_kmeans   #no. of iterations
    P_init={}
    pi=np.ones((1,K))*(1/K)   #np.random.dirichlet(alpha)  #to genereate lamda␣
↪values such that they sum to 1
    P_init["lamda"]=pi[0,:]

    #random initialisation of latent variables
    #for i in range(np.max(train_labels)+1):
    z=np.random.randint(0,K,size=(classes.shape[0],1))
    #print(z)
    #print(np.max(z))
    #mu={}     #dictionary to store the mean value for each mixture component
    tot_err=np.zeros((iterations,1))

    for itr in range(iterations):
        mu=np.zeros((K,784))
        xi_s=classes
        xi_s=np.array(xi_s)

        #print(np.shape(xi_s))
        err=np.zeros((K,1))
        for k in range(K):
            z_d=np.where(z==k,1,0)
            #print(z_d)
            #print(f"sum : {np.sum(z_d)}")
            z_d=z_d.T

            #find the mean value of kth mixture component using the images that␣
↪belong to the kth mixture
            mu_k=(z_d@xi_s)/(np.sum(z_d))
            mu_k=np.reshape(mu_k,(1,784))
            #print(np.max(mu_k))
            mu[k,:]=mu_k

            error=z_d@(xi_s-mu_k)
            err[k]=err[k]+np.linalg.norm(error)


        for n in range(len(xi_s)):
            xn=xi_s[n][:]-mu
            #print(np.shape(xn))
            min_dist=np.linalg.norm(xn,axis=1)
```

```
            z[n]=np.argmin(min_dist)  #store the distance  that is minimum among␣
 ↪all the mixture components

        #print(min_dist)

        z[n]=np.argmin(min_dist)
        #print(min_dist)

        total_err=np.sum(err)
        #print(f"total error : {total_err}")


#         if total_err<1e-20:
#           print(f"The iteration no. at which the total_err is less than 1e-20␣
 ↪is {total_err}: corresponding iteration is {itr}")
#           break

    #plot the images for each class using the avearge mean obtained by taking␣
 ↪mean over all mixture components
    plt.figure(figsize=(64,64))    #to specify the figure size
    print("Images after initialistion of parameters using K-means")
    for i in range(K):
        plt.subplot(1,K,i+1)
        train_image=mu[i,:]
        #train_imag1=np.where(train_imag1>0.5,1, 0)
        train_image=np.reshape(train_image,(28,28))
        #train_imag1=train_imag1.reshape(int(np.sqrt(len(train_imag1))),int(np.
 ↪sqrt(len(train_imag1))))
        plt.imshow(train_image,cmap='gray')
        plt.axis('off')
    for k in range(K):
        P_init["prob"+str(k)]=mu[k,:]
    return P_init
```

0.0.4 LOAD THE MNIST DATA AND CLASSIFFY THE DATA TO THEIR RESPECTIVE CLASSES

[6]:
```
"""Extract MNIST data from the specified folder. mndata has images of numbers␣
 ↪0-9 with their respective labels.

    train_images -- training set represented by a numpy array of shape (no of␣
 ↪images=60000,size of each image=784 )
    train_labels -- training labels represented by a numpy array (vector) of␣
 ↪shape (no of images=60000,1)
    test_images  -- test set represented by a numpy array of shape (no of␣
 ↪images=10000,size of each image=784)
```

```
    test_labels  -- test labels represented by a numpy array (vector) of shape
 ↪(no of images=10000,1)
"""

mndata=MNIST(r"Dataset_spr_2")

train_images, train_labels=mndata.load_training()      #loads training images
 ↪and their labels
test_images, test_labels=mndata.load_testing()     #loads testing images and
 ↪their labels
test_images=np.array(test_images)
#print(np.shape(test_images))

"""to convert gray scale image to binary image"""
train_images=np.array(train_images)
train_images=np.where(train_images > 127, 1,0 )
test_images=np.array(test_images)
test_images=np.where(test_images > 127, 1,0 )

"""classes -- list with each item containing images of that particular class """

train_size=len(train_labels)     #no. of images in training set
test_size=len(test_labels)       #no. of images in testing set
img_size=train_images.shape[1]

N_class=np.max(train_labels)+1   #total no. of classes
all_classes=[[] for i in range(N_class)]
for i in range(train_size):
      all_classes[train_labels[i]].append(train_images[i]) # the images with
 ↪label k is added to the class k.

#actual question starts from here
K=3  #no. of mixture components
n_img=200 #no. of images from each set
class_no=np.array([2,3,4])
classes=np.array(all_classes[class_no[0]][:n_img][:])
for i in range(len(class_no)-1):
    classes=np.append(classes,all_classes[class_no[i+1]][:n_img][:],axis=0)
# classes=np.append(classes,all_classes[4][:200][:],axis=0)
classes=np.random.permutation(classes)
```

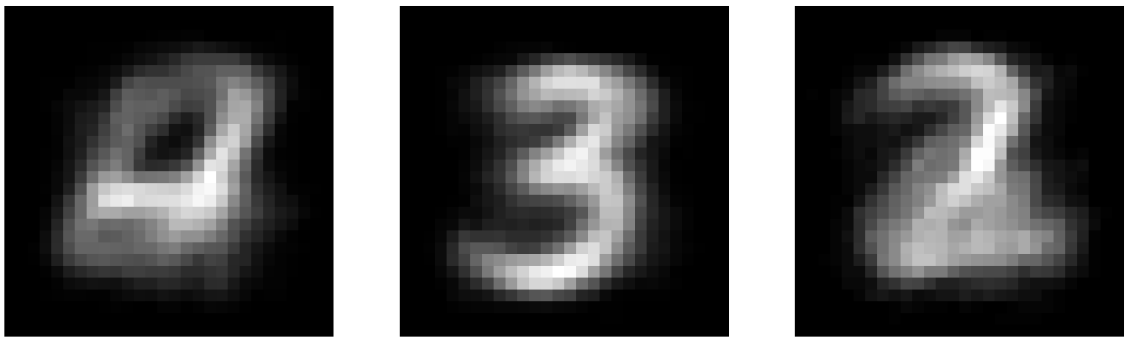
0.0.5 RUN THE BELOW CELL TO IMPLEMENT EM ALGORITHM

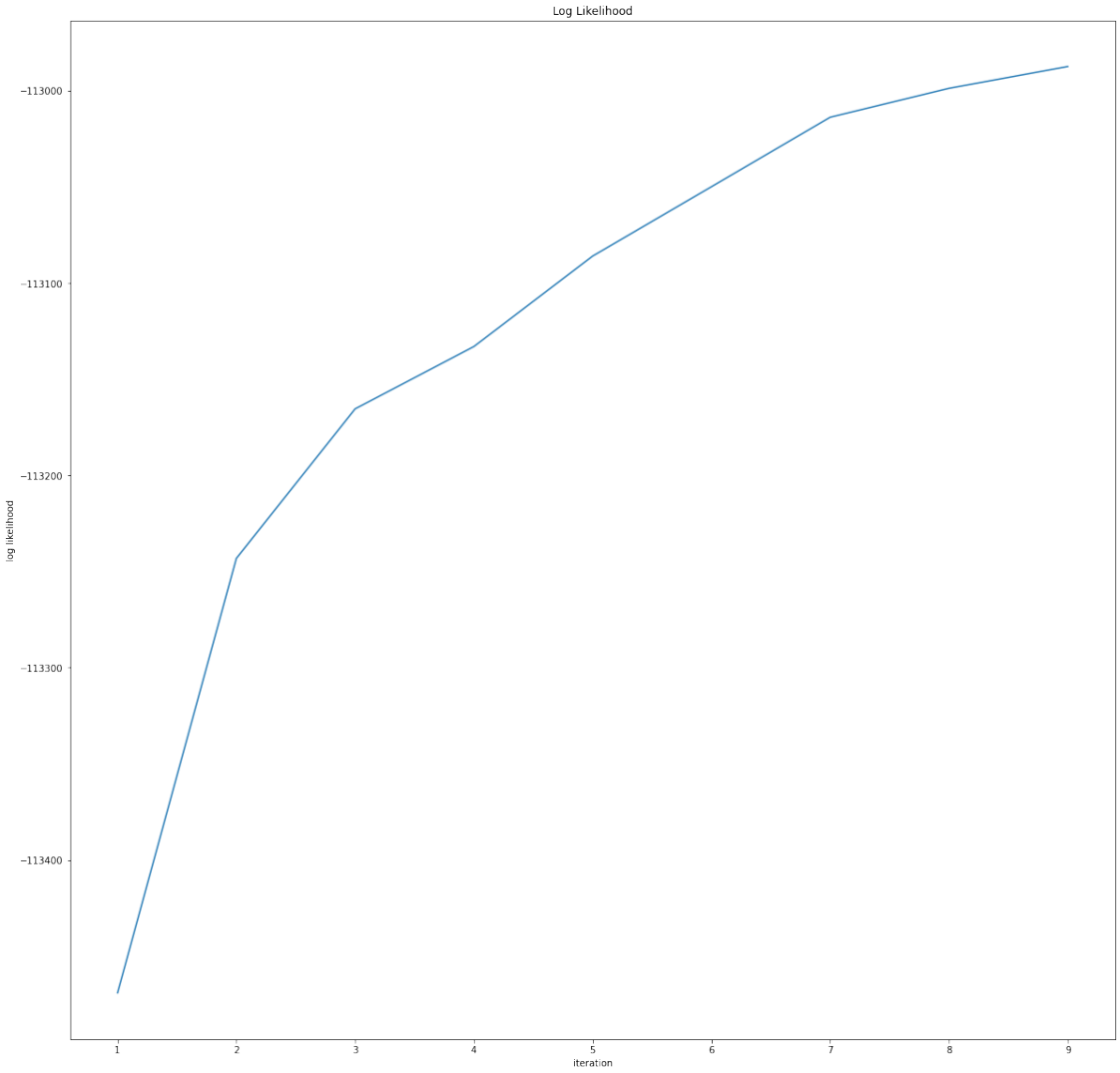
[7]:
```
K=3  #no. of mixture components
iterations=10 #max. no. of iterations
img_size=784
```

```
P=init_using_kmeans(K,iter_kmeans=2) #initialisation_parameters(K,img_size) ␣
 ↪#random initialisation of parameters
#print(np.max(P["class_prob"+str(0)]))



log_likehood=np.zeros((iterations,1))
N_k=np.zeros((1,K))

#run EM algorithm
log_likehood,P,res_coeff,iter_convg=em_algorithm(iterations,P,K,classes)
    #print(f"log_likelihood of class {i} is {log_likehood[i,:]}")

#print(log_likehood)
# dictonary={"class":range(0,10)}
# for i in range(10):
#     dictonary["L-L at iteration "+str(i)]=log_likehood[:,i]
# dataframe = pd.DataFrame(dictonary)

# display(dataframe)

#plot the log likelihood for each class
plt.figure(figsize=(20,20))
plt.title("Log Likelihood")
#plt.axis('off')

plt.plot(np.linspace(1,iter_convg,iter_convg),log_likehood[:iter_convg])  #np.
 ↪linspace(1,iterations,iterations),
plt.xlabel("iteration")
plt.ylabel("log likelihood")
plt.show()
x=plt.figure(figsize=(64,64))    #to specify the figure size
for i in range(K):
    plt.subplot(1,K,i+1)
    train_image=P["prob"+str(i)]
    #train_image=np.where(train_image>0.5,1, 0)
    train_image=np.reshape(train_image,(28,28))
    #train_imag1=train_imag1.reshape(int(np.sqrt(len(train_imag1))),int(np.
 ↪sqrt(len(train_imag1))))
    plt.imshow(train_image,cmap='gray')
    plt.axis('off')

print("Images obtained after estimating the paramters of the mixture components")

    #plt.title("Class "+str(i+2))
# x=plt.figure(figsize=(64,64))    #to specify the figure size
```

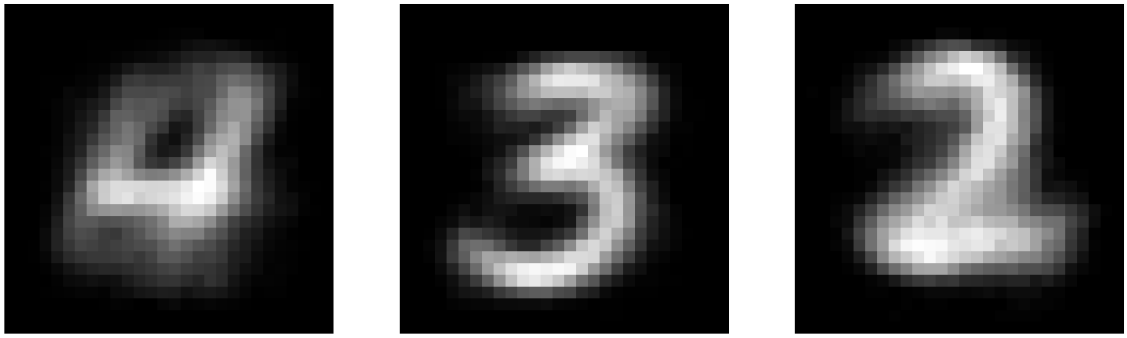
```
# x.suptitle("Images obtained after converting the feature values to binary␣
 ↪values")
# for i in range(K):
#     plt.subplot(1,K,i+1)
#     train_image=P["prob"+str(i)]
#     train_image=np.where(train_image>0.5,1, 0)
#     train_image=np.reshape(train_image,(28,28))
#     #train_imag1=train_imag1.reshape(int(np.sqrt(len(train_imag1))),int(np.
 ↪sqrt(len(train_imag1))))
#     plt.imshow(train_image,cmap='gray')
#     plt.axis('off')
#     #plt.title("Class "+str(i+2))
```

Images after initialistion of parameters using K-means





Images obtained after estimating the paramters of the mixture components



1 K-Means for MNIST Dataset

[75]:
```
from mnist import MNIST
import os
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stat
from sklearn.metrics import confusion_matrix
import seaborn as sn
import pandas as pd
import cv2
```

[77]:
```
def k_means_classifier(test_images,test_labels,mean,K):
  """
  Input:
  test_images, test labels -- Images and their corresponding labels for testing␣
 ↪the model
  mean -- contains the mean values of mixture components
  K    -- no of mixture components
  """
  K=K
  mis_clf=0
  #print(np.shape(test_images))
  predicted_label=np.zeros((len(test_images)))    #to store the predicted label␣
 ↪for each image
  #print(np.shape(predicted_label))

  for i in range(len(test_labels)):
    temp_image=np.array(test_images[i][:])   #load the feature vector of␣
 ↪particular image that needs to be classified
    temp_image=np.reshape(temp_image,(1,784))
    min_dist=np.zeros((np.max(test_labels)+1,1))
    for j in range(np.max(test_labels)+1):
        min_dist_class=np.zeros((K,1))
        for k in range(K):
            temp_image_dist=temp_image-mean["class"+str(j)][k,:]
            #print(np.max(mean_avg["class"+str(j)]))

            #find the distance of the image from each mixture component of each␣
 ↪class
            min_dist_class[k]=np.linalg.norm(temp_image_dist)

        min_dist[j]=np.min(min_dist_class)  #store the distance  that is minimum␣
 ↪among all the mixture components

    #print(min_dist)
```

```
    #the image belongs to the class such that the image distance is min. from␣
↪one of the mixture component of a class
    predicted_label[i]=np.argmin(min_dist)

    if predicted_label[i]!=test_labels[i]:
        mis_clf+=1
        #print(i,mis_clf,predicted_label[i],test_labels[i])                #to␣
↪print the predicted class and ture class of a mis-classified image

  #print(np.shape(predicted_label))
  test_labels=np.array(test_labels)
  print(f"No. of miss classifications is :{mis_clf}")
  miss_classf_error=(mis_clf/len(test_labels))*100
  print(f"% miss classification error is : {str(round(miss_classf_error,2))}%")
  #to create the confusion matrix
  conf_matrix=confusion_matrix(test_labels,predicted_label)       #to create the␣
↪confusion matrix

  return conf_matrix,mis_clf
```

1.0.1 LOAD THE MNIST DATA AND CLASSIFFY THE DATA TO THEIR RESPECTIVE CLASSES

[78]:
```
"""Extract MNIST data from the specified folder. mndata has images of numbers␣
↪0-9 with their respective labels.

    train_images -- training set represented by a numpy array of shape (no of␣
↪images=60000,size of each image=784 )
    train_labels -- training labels represented by a numpy array (vector) of␣
↪shape (no of images=60000,1)
    test_images  -- test set represented by a numpy array of shape (no of␣
↪images=10000,size of each image=784)
    test_labels  -- test labels represented by a numpy array (vector) of shape␣
↪(no of images=10000,1)
"""

mndata=MNIST(r"Dataset_spr_2")

train_images, train_labels=mndata.load_training()      #loads training images␣
↪and their labels
```

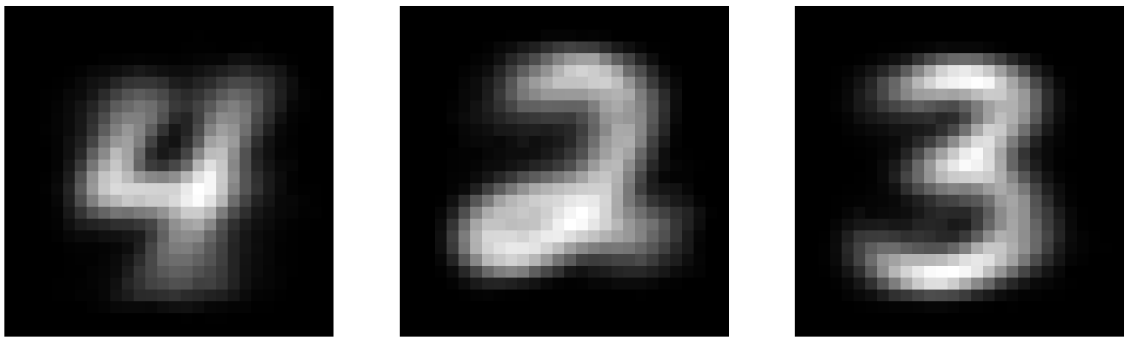
```
test_images, test_labels=mndata.load_testing()     #loads testing images and␣
 ↪their labels
test_images=np.array(test_images)
#print(np.shape(test_images))

"""to convert gray scale image to binary image"""
train_images=np.array(train_images)
train_images=np.where(train_images > 127, 1,0 )
test_images=np.array(test_images)
test_images=np.where(test_images > 127, 1,0 )

"""classes -- list with each item containing images of that particular class """

train_size=len(train_labels)      #no. of images in training set
test_size=len(test_labels)        #no. of images in testing set
img_size=train_images.shape[1]

N_class=np.max(train_labels)+1    #total no. of classes
all_classes=[[] for i in range(N_class)]
for i in range(train_size):
      all_classes[train_labels[i]].append(train_images[i]) # the images with␣
 ↪label k is added to the class k.

K=3  #no. of mixture components
n_img=500 #no. of images from each set
class_no=np.array([2,3,4])
classes=np.array(all_classes[class_no[0]][:n_img][:])
for i in range(len(class_no)-1):
    classes=np.append(classes,all_classes[class_no[i+1]][:n_img][:],axis=0)
# classes=np.append(classes,all_classes[4][:200][:],axis=0)
classes=np.random.permutation(classes)


#to check whether the images are shuffled or not
# train_image=classes[10,:]
# #train_image=np.where(train_image>0.5,1, 0)
# train_image=np.reshape(train_image,(28,28))
# #train_imag1=train_imag1.reshape(int(np.sqrt(len(train_imag1))),int(np.
 ↪sqrt(len(train_imag1))))
# plt.imshow(train_image)
# plt.axis('off')

# classes_labels=2*np.ones(200)
# classes_labels=np.append(classes_labels,3*np.ones(200))
# classes_labels=np.append(classes_labels,4*np.ones(200))

#print(np.shape(classes))
```

1.0.2 Run the below code to implement k-means clustering

[84]:
```
iterations=10   #no. of iterations
#random initialisation of latent variables
#for i in range(np.max(train_labels)+1):

#random initialisation of images to any one of the k-bins
z=np.random.randint(0,K,size=(classes.shape[0],1))
tot_err=np.zeros((iterations,1))

for itr in range(iterations):
    mu=np.zeros((K,784))
    xi_s=classes
    xi_s=np.array(xi_s)

    #print(np.shape(xi_s))
    err=np.zeros((K,1))
    for k in range(K):
        z_d=np.where(z==k,1,0)
        z_d=z_d.T

        #find the mean value of kth mixture component using the images that␣
↪belong to the kth mixture
        mu_k=(z_d@xi_s)/(np.sum(z_d))
        mu_k=np.reshape(mu_k,(1,784))

        mu[k,:]=mu_k

        error=z_d@(xi_s-mu_k)
        err[k]=err[k]+np.linalg.norm(error)


    for n in range(len(xi_s)):
        xn=xi_s[n][:]-mu
        #print(np.shape(xn))
        min_dist=np.linalg.norm(xn,axis=1)

        z[n]=np.argmin(min_dist)  #store the distance  that is minimum among all␣
↪the mixture components

    #print(min_dist)

    #reassignment of images to different bins based on the min. distance of the␣
↪image from the mean
    z[n]=np.argmin(min_dist)
    #print(min_dist)
```

```
    total_err=np.sum(err)
    #print(f"total error : {total_err}")

#     for n in range(len(xi_s)):
#         temp_image=np.array(xi_s[i][:])   #load the feature vector of␣
 ↪particular image that needs to be classified
#         temp_image=np.reshape(temp_image,(1,784))
#         min_dist=np.zeros((K,1))

#         for k in range(K):
#             temp_image_dist=temp_image-mu[k,:]
#             #print(np.max(mean_avg["class"+str(j)]))

#             #find the distance of the image from each mixture component
#             min_dist[k]=np.linalg.norm(temp_image_dist)

#         z[n]=np.argmin(min_dist)  #store the distance  that is minimum among␣
 ↪all the mixture components

    if total_err<1e-20:
      print(f"The iteration no. at which the total_err is less than 1e-20 is␣
 ↪{total_err}: corresponding iteration is {itr}")
      break

#plot the images for each class using the avearge mean obtained by taking mean␣
 ↪over all mixture components
plt.figure(figsize=(64,64))
print("Images constructed using mean values of each bin")
for i in range(K):
    plt.subplot(1,K,i+1)
    train_image=mu[i,:]
    #train_imag1=np.where(train_imag1>0.5,1, 0)
    train_image=np.reshape(train_image,(28,28))
    #train_imag1=train_imag1.reshape(int(np.sqrt(len(train_imag1))),int(np.
 ↪sqrt(len(train_imag1))))
    plt.imshow(train_image,cmap='gray')
    plt.axis('off')



# #k-means classifier using the mean values obtained from k-means clustering
#␣
 ↪confuse_matrix,mis_classification=k_means_classifier(test_images,test_labels,mu,K)
# plot_cnf_matrix(confuse_matrix)
```

```
Images constructed using mean values of each bin
```



2 Case : GMM Univariate

[59]:
```
from mnist import MNIST
import os
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stat
from sklearn.metrics import confusion_matrix
import seaborn as sns
import pandas as pd
import cv2
import sklearn.datasets as dataset
from sklearn.mixture import GaussianMixture
from scipy.stats import multivariate_normal
from mpl_toolkits.mplot3d import Axes3D

from itertools import product, combinations
import os
import scipy.stats as stat

import sklearn
import sklearn.datasets as dataset

from matplotlib import cm
# import plotly.express as px
%matplotlib inline
```

2.0.1 GMM Algorithm

[10]:
```
def gmm_algorithm_univariate(iterations,P_em,K,clas):

    """GMM ALGORITHM
    Input :
```

```
   iterations -- Maximum no. of iterations
   P_em       -- dictionary that contains mean, variances and weightages of all␣
↪mixture components
   K          -- no. of mixture components
   clas       -- input data
   Return:
   P_em -- Updated dictionary that contains mean, variances and weightages of␣
↪all mixture components
   res_coeff  -- contains the responsibility coefficient of all images w.r.t␣
↪each mixture
   log_likey  -- contains log likelihood of each iteration

   """


   ll_old=0
   ll_new=0
   log_likely= np.zeros((iterations,1))

   for itr in range(iterations):
       #responsibility coefficient of size(no. of images in a clas x mixture␣
↪components)
       res_coeff=np.zeros((len(clas),K))

       for n in range(len(clas)):
           res_coeff_nr=np.zeros((1,K))       #to store respob. coeff of each␣
↪mixture component
           xn=clas[n]
           for k in range(K):

               #store the mean and variance of kth mixture component
               mu=P_em["mean"+str(k)]
               var=P_em["variance"+str(k)]
               var=np.abs(var)
               var_inv=1/(var+0.01) #added 0.01 to avoid 'nan'(undefined)  np.
↪linalg.inv(var)

               #(1/(2*pi*var)^(1/2))*exp(-x^2/(2*var))
               fx_pi=1/(np.sqrt(2*np.pi*(var+0.01)))
               fx_var=(1/(np.sqrt(var+0.01)))
               muSmu=((xn-mu)*(xn-mu)*var_inv)/2
               f_x_k=fx_pi*np.exp(-muSmu)

               #gamma_nk numerator=lamda_k*gaussian(x_n,theta_k)

               res_coeff_nr[0,k]=P_em["lamda"][k]*f_x_k
```

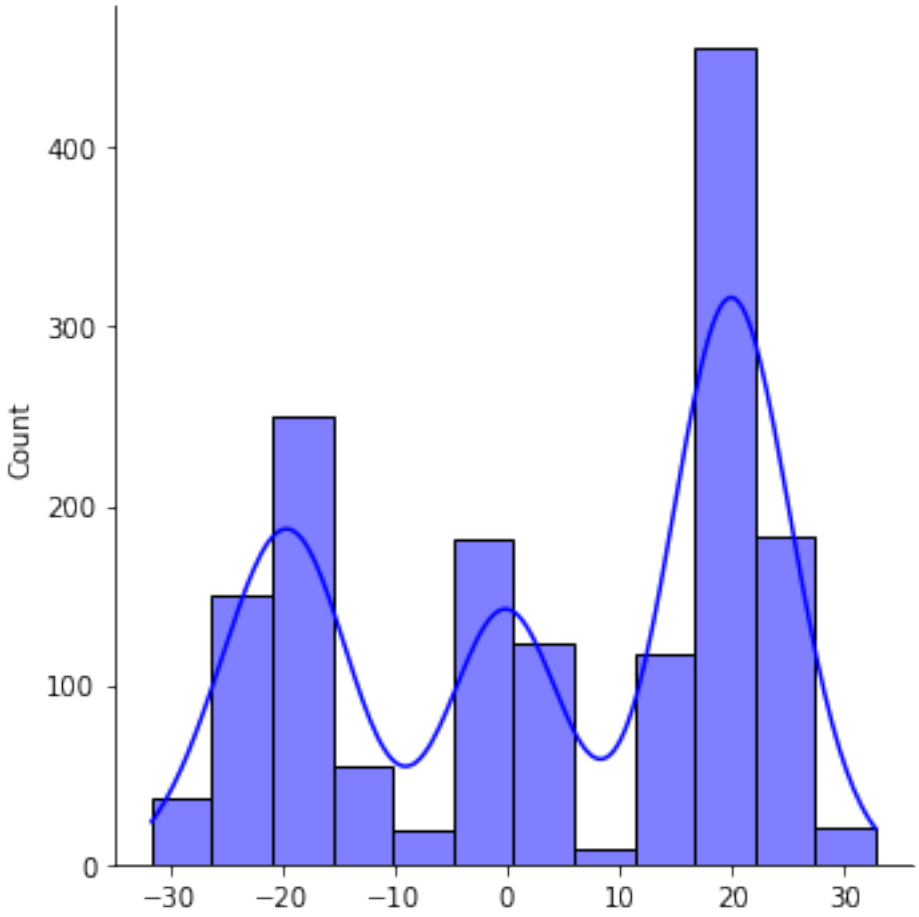
```
            # respob. coeff of nth input w.r.t kth mixture
            #added 1e-50 to the denominator so that responsibility coefficient␣
↪doesn't become undefined value (0/0 form)
            res_coeff[n,:]=res_coeff_nr/(np.sum(res_coeff_nr)+1e-50)

        #EXPECTATION - STEP

        #sum of gamma_nk over-all n for each k
        N_k=np.sum(res_coeff,axis=0)

        for k in range(K):

            res_coeff_k=res_coeff[:,k]
            res_coeff_k=np.reshape(res_coeff_k,(1,len(clas)))

            #MAXIMISATION - STEP
            #added 1e-50 to the denominator so that responsibility coefficient␣
↪doesn't become undefined value (0/0 form)

            #mean=(sum(gamma_nk*x_n))(over all n)/sum(gamma_nk) (over all n)
            P_em["mean"+str(k)]=(res_coeff_k@clas[:])/(N_k[k]+1e-50)

            #var= (sum(gamma_nk*(x_n-mean_k)^2)(over all n)/sum(gamma_nk) (over␣
↪all n)
            var_temp=clas[:]-P_em["mean"+str(k)]
            var_temp=np.array(var_temp)
            var_temp=np.abs(var_temp*var_temp)

            #update variance of kth mixture component
            P_em["variance"+str(k)]=(res_coeff_k@(var_temp))/(N_k[k]+1e-10)



        #update lamda
        P_em["lamda"]=N_k/len(clas)


        """Below code is find the log likelihood for the parameters obtained in␣
↪the current iteration"""


        ll=0
        for n in range(len(clas)):
            xi=clas[n]
            lh=0
            for k in range(K):
```

```
                #store the mean and variance of kth mixture component
                mu=P_em["mean"+str(k)]
                var=P_em["variance"+str(k)]
                var=np.abs(var)
                var_inv=1/(var+0.01) #added 0.01 to avoid 'nan'(undefined)  np.
 ↪linalg.inv(var)

                #(1/(2*pi*var)^(1/2))*exp(-x^2/(2*var))
                fx_pi=1/(np.sqrt(2*np.pi*(var+0.01)))
                fx_var=(1/(np.sqrt(var+0.01)))
                muSmu=((xi-mu)*(xi-mu)*var_inv)/2
                f_x_k=fx_pi*np.exp(-muSmu)

                #gamma_nk numerator=lamda_k*gaussian(x_n,theta_k)

                lh=lh+P_em["lamda"][k]*f_x_k

#                 mu=P_em["mean"+str(k)]  #stores the mean of kth component
#                 var=P_em["variance"+str(k)]
#                 var=np.abs(var)
#                 var_inv=1/(var+1e-50) #np.linalg.inv(var)
#                 fx_pi=np.log((2*np.pi)*var+0.99)*0.5
#                 #fx_var=(1/(np.sqrt(var+1e-50)))
#                 muSmu=((xi-mu)*var_inv*(xi-mu))/2

#                 f_x_k=fx_pi-muSmu
                #ll_update=res_coeff[n,k]*f_x_k
                #ll_update=res_coeff[n,k]*(f_x_k+np.log(P_em["lamda"][k]+0.01))␣
 ↪#(xi@p_i.T+(1-xi)@(p_i_minus).T
                #ll=ll+ll_update
            #print(lh)
            ll=ll+np.log(lh)
#         print(ll)
        ll_old=ll_new
        ll_new=ll
        log_likely[itr]=ll_new



        if (np.abs(ll_new-ll_old)<0.1 and itr>=1):
            #log_likely[itr+1:iterations]=log_likely[itr]
            print(f"For iteration {itr} the difference between new log␣
 ↪likelihood and old log likelihood is less than 0.1")
            break

    return log_likely,P_em,res_coeff,itr
```

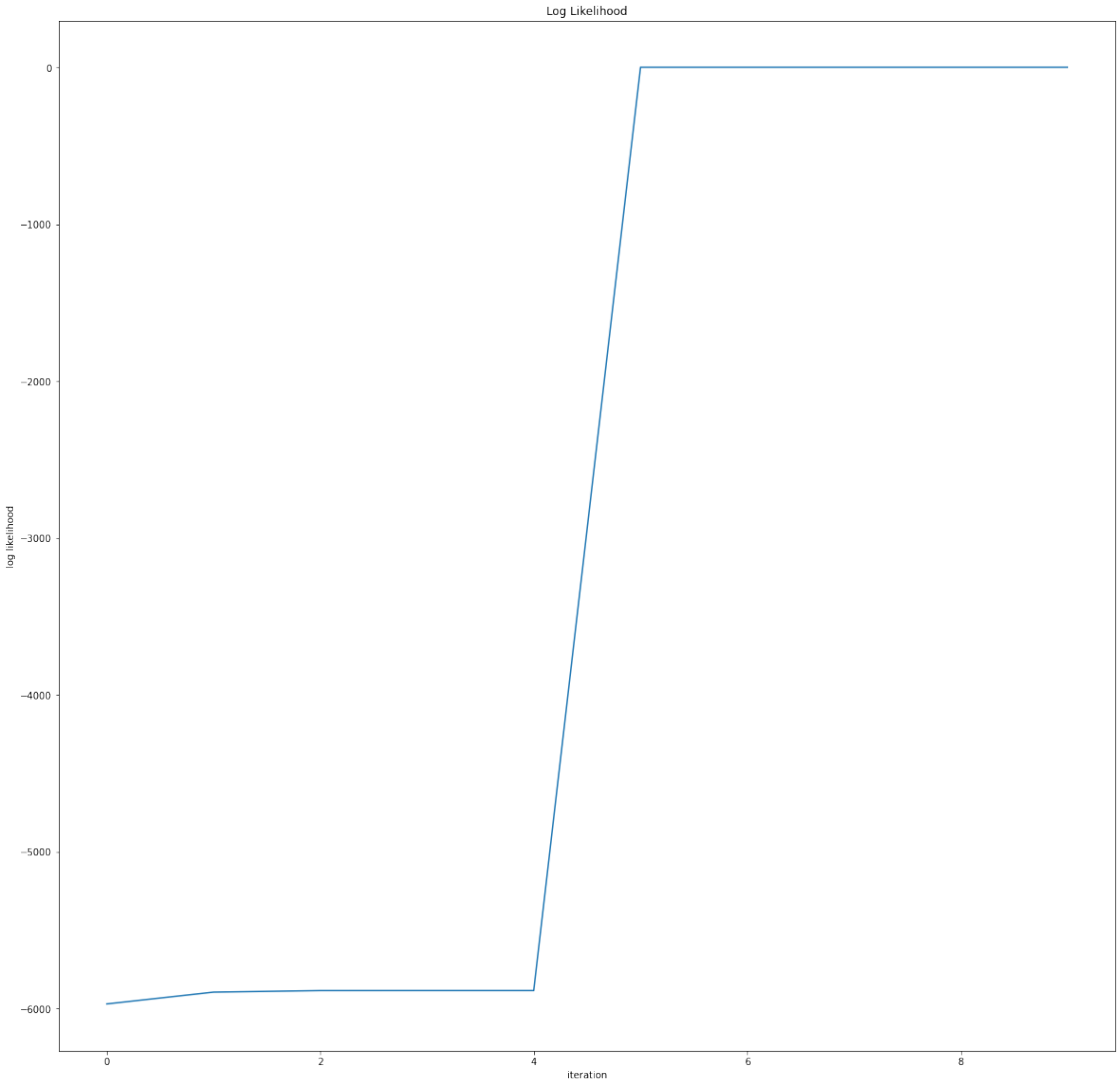
2.0.2 GMM for Univariate Gaussian

[80]:
```
K = 3      # Number of mixture components in univariate GMM
N = 1600 # Number of samples to be drawn from univariate GMM
iterations=10
means     = np.linspace(-20, 20, num=K)
variances = np.array([15,8,12])#np.linspace(3,6, num=K)
lambdas   = [0.3,0.2,0.5]

samples_lst=0
for n in range(N):
    mix_no=np.random.choice(3,p=lambdas)
    sample= np.random.normal(loc = means[mix_no],scale = np.
 ↪sqrt(variances[mix_no]))
    samples_lst=np.append(samples_lst,sample)
#     samples_lst= [np.random.normal(loc = means[k],scale = np.
 ↪sqrt(variances[k]),size = int(N//2)) for k in range(K)]
#     samples     = np.expand_dims(np.array([item for sublist in samples_lst for␣
 ↪item in sublist]),axis=1)
sns.displot(samples_lst, color='b',kde=True)   #kde=True to get the desnity␣
 ↪estimate
plt.hist(samples_lst, bins =10,density=True)
plt.show()

P=dict()

#initialise lamda's=1/K
phi=np.ones((1,K))*(1/K)  #np.random.dirichlet(alpha)  #to genereate lamda␣
 ↪values such that they sum to 1
P["lamda"]=phi [0,:]       #pi[0,:]

#random initialisation of mean and variance
for k in range(K):
    P["mean"+str(k)]= -20+40*np.random.rand()
    P["variance"+str(k)]= 5+ 5*np.random.rand()

# print(samples.shape)
# print(means.shape)
# print(variances.shape)
log_likely,P,res_coeff,itera=gmm_algorithm_univariate(iterations,P,K,samples_lst)
plt.figure(figsize=(20,20))
plt.title("Log Likelihood")
#plt.axis('off')

plt.plot(log_likely)#plt.plot(np.linspace(1,iter_convg,iter_convg),log_likehood[:
 ↪iter_convg])  #np.linspace(1,iterations,iterations),
plt.xlabel("iteration")
```

```
plt.ylabel("log likelihood")
plt.show()

for i in range(K):
    print(f"For component {i}  true mean is {means[i]}: and variance is␣
 ↪{variances[i]}")
#     print(np.squeeze(P["mean"+str(i)]),end=' , variance is : ')
#     print(np.squeeze(P["variance"+str(i)]),end=' and the weight is : ')
#     print(np.squeeze(P["lamda"][i]))


for i in range(K):
    print(f"For component {i} mean is : ",end='')
    print(np.squeeze(P["mean"+str(i)]),end=' , variance is : ')
    print(np.squeeze(P["variance"+str(i)]),end=' and the weight is : ')
    print(np.squeeze(P["lamda"][i]))

print("")
print("Plot Gaussian Distribution using the estimated parameters")

N=600
x=np.linspace(np.min(means)-10,np.max(means)+10,N)
for i in range(K):
    var_k=P["variance"+str(i)]
    std_dev=np.sqrt(var_k)
    loc1=P["mean"+str(i)]
#     scale=std_dev
    z=stat.norm.pdf(x,loc1,std_dev)
    plt.plot(x,z)
plt.show()
```

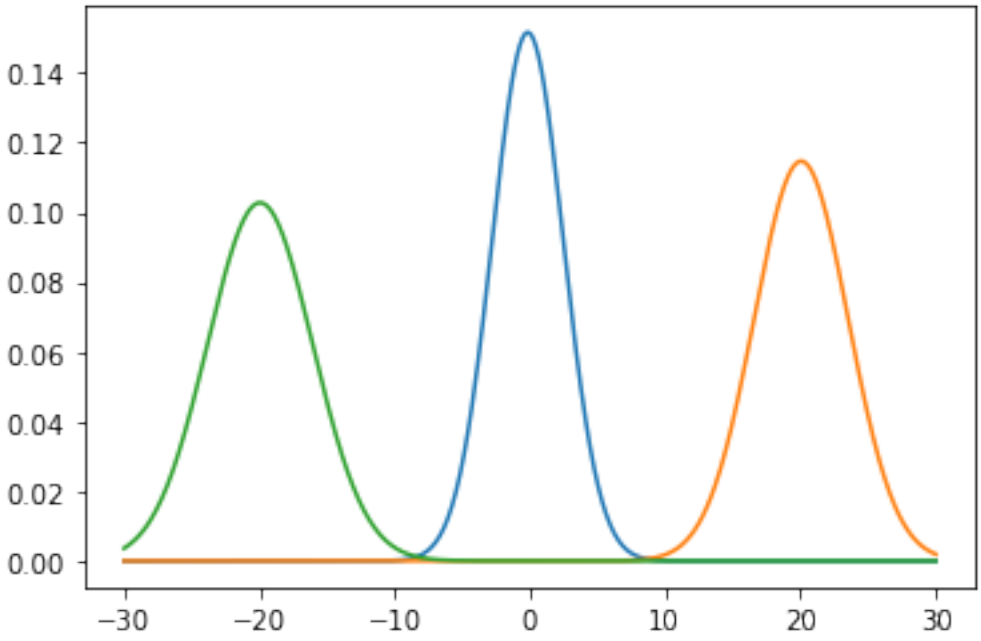



```
For iteration 4 the difference between new log likelihood and old log likelihood
is less than 0.1
```




```
For component 0  true mean is -20.0: and variance is 15
For component 1  true mean is 0.0: and variance is 8
For component 2  true mean is 20.0: and variance is 12
For component 0 mean is : -0.1618937738202576 , variance is : 6.92240040558849
and the weight is : 0.20172138283413588
For component 1 mean is : 20.04687944949306 , variance is : 12.098499257304349
and the weight is : 0.489625835583871
For component 2 mean is : -19.95485630773358 , variance is : 15.071574481853876
and the weight is : 0.3086527815819941
```

Plot Gaussian Distribution using the estimated parameters



[12]:
```
# gm=GaussianMixture(n_components=K, random_state=0).fit(samples)
# print(gm.covariances_,gm.means_,gm.weights_,gm.n_iter_)
# dim = 2  # dimension of random vector
# K = 4    # Number of components in univariate GMM
# N = 1000 # Number of samples from multivariate GMM
```

3 Case : GMM Multivariate

[13]:
```
def gmm_algorithm_mulivariate(iterations,P_em,K,clas):

    """GMM ALGORITHM
    Input :
    iterations -- Maximum no. of iterations
    P_em       -- dictionary that contains parameters and weightages of all␣
 ↪mixture components
    K          -- no. of mixture components

    Return:
    P_em -- Updated dictionary that contains parameters and weightages of all␣
 ↪mixture components
    res_coeff  -- contains the responsibility coefficient of all images w.r.t␣
 ↪each mixture
    log_likey  -- contains log likelihood of each iteration
```

```
    """


    ll_old=0
    ll_new=0
    log_likely= np.zeros((iterations,1))
    d=clas.shape[1]
    N=clas.shape[0]

    for itr in range(iterations):
        #responsibility coefficient of size(no. of samples in clas x mixture␣
↪components)
        res_coeff=np.zeros((N,K))

        for n in range(N):
            res_coeff_nr=np.zeros((1,K))       #to store respob. coeff of each␣
↪mixture component
            xn=clas[n][:]
            xn=np.reshape(xn,(d,1))
            #print(xn.shape)
            for k in range(K):

                mu=P_em["mean"+str(k)]  #stores the mean of kth component
                mu=np.reshape(mu,(d,1))
                var=P_em["variance"+str(k)]
                var=np.abs(var)
                #below code is to find Gaussian(x_n,theta_k)

                var_inv=np.abs(np.linalg.inv(var+np.identity(d)*0.01))
                fx_pi=1/(np.sqrt((2*np.pi)))**(d)
                fx_var=np.abs(np.linalg.det(var+np.identity(d)*0.01))

                fx_var=np.abs(1/np.sqrt(fx_var))
                muSmu=((xn-mu).T@var_inv@(xn-mu))/2


                f_x_k=fx_pi*fx_var*np.exp(-muSmu)

                #update the responsibility coefficient for kth component
                res_coeff_nr[0,k]=P_em["lamda"][k]*f_x_k
                #print(f"value of res : {res_coeff_nr[0,:]}, {itr}")


            # respob. coeff of nth sample with kth mixture
            #added 1e-50 to the denominator so that responsibility coefficient␣
↪doesn't become undefined value (0/0 form)
            res_coeff[n,:]=res_coeff_nr/(np.sum(res_coeff_nr)+1e-50)
```

```
        #EXPECTATION - STEP
        N_k=np.sum(res_coeff,axis=0)

        #update the lamda parameter for all mixture components
        P_em["lamda"]=N_k/N #len(clas)
#         P_em["lamda"]=P_em["lamda"]/np.sum(P_em["lamda"])
        #print(np.sum(P_em["lamda"]))

        #print(f"Value of {np.sum(N_k)},{itr}")
        #below code is to update the mean and variances for the mixture␣
 ↪components
        for k in range(K):
            mean_temp=np.zeros((1,d))
            for n in range(N):
                xi=clas[n,:]
                xi=np.reshape(xi,(1,d))

                res_coeff_k=res_coeff[n,k]
                #MAXIMISATION - STEP
                mean_temp= mean_temp+(res_coeff_k*xi)

            P_em["mean"+str(k)]= mean_temp/(N_k[k]+1e-50)

        for k in range(K):
            var_temp=np.zeros((d,d))
            mu_new=P_em["mean"+str(k)]
            for n in range(N):
                xi=clas[n,:]
                xi=np.reshape(xi,(1,d))
                res_coeff_k=res_coeff[n,k]
                #res_coeff_k=np.reshape(res_coeff_k,(1,len(clas)))

                #MAXIMISATION - STEP
                #added 1e-50 to the denominator so that responsibility␣
 ↪coefficient doesn't become undefined value (0/0 form)
                xi_mu=xi-mu_new
                xi_mu=np.reshape(xi_mu,(d,1))
                sigma_new=np.outer(xi_mu,xi_mu)

                var_temp= var_temp+np.abs((res_coeff_k*(sigma_new)))

            #update the mean and variances of kth class


            P_em["variance"+str(k)]= np.abs(var_temp)/(N_k[k]+1e-50)
```

```
        """Below code is find the log likelihood for the parameters obtained in␣
 ↪the current iteration"""
        ll=0
        for n in range(N):
            x=clas[n][:]
            x=np.reshape(x,(d,1))
            lh=0
            for k in range(K):

                mu=P_em["mean"+str(k)]  #stores the mean of kth component
                mu=np.reshape(mu,(d,1))
                var=np.abs(P_em["variance"+str(k)])

                var_inv=np.abs(np.linalg.inv(var+np.identity(d)*1e-5)) #to avoid␣
 ↪singularity
                fx_pi=1/(np.sqrt((2*np.pi)))**(d)
                fx_var=np.abs(1/(np.linalg.det(var+np.identity(d)*1e-5)))
                fx_var=np.sqrt(fx_var)
                muSmu=((x-mu).T@var_inv@(x-mu))/2

                f_x_k=fx_pi*fx_var*np.exp(-muSmu)

                #f_x_k=np.log(fx_pi)+np.log(fx_var)-muSmu

                #ll_update=P_em["lamda"][k]*(f_x_k) #np.log(P_em["lamda"][k]+0.
 ↪01)) #(xi@p_i.T+(1-xi)@(p_i_minus).T
                lh=lh+P_em["lamda"][k]*f_x_k
            ll=ll+np.log(lh+0.01)

        ll_old=ll_new
        ll_new=ll
#       print(f"log likelihood : {ll_new},old : {ll_old}")

        log_likely[itr]=ll_new



#         if (abs(ll_new-ll_old)<1 and itr>=1):
#             #log_likely[itr+1:iterations]=log_likely[itr]
#             print(f"For iteration {itr} the difference between new log␣
 ↪likelihood and old log likelihood is less than 1")
#             break
```

```
    return log_likely,P_em,res_coeff,itr
```

[14]:
```
def k_means_init(K,iteration_max,samples):
    iterations=iteration_max   #no. of iterations
    K=K
    z=np.random.randint(0,K,size=(samples.shape[0],1))
    tot_err=np.zeros((iterations,1))
    d=samples.shape[1]   #dimesnion of the data
    N=samples.shape[0]   #size of the data

    for itr in range(iterations):
        mu=np.zeros((K,d))
        covariance=[np.zeros((d,d)) for i in range(K)]

        #print(np.shape(covariance))
        xi_s=samples
        xi_s=np.array(xi_s)

        #print(np.shape(xi_s))
        err=np.zeros((K,1))
        for k in range(K):
            mean_temp=np.zeros((1,d))
            var_temp=np.zeros((d,d))

            zd=np.where(z==k,1,0)
            for n in range(N):

                #find the mean value of kth mixture component using the samples␣
↪that belong to the kth mixture in ith class
                mean_temp=mean_temp+(zd[n]*xi_s[n,:])
                mean_temp=np.reshape(mean_temp,(1,d))
    #             x_mu=xi_s[n,:]-mu[k,:]
    #             #print(x_mu.shape)
    #             var_temp=(x_mu@x_mu.T)
    #             #var_temp=var_temp.reshape(2,600,2,600).sum(axis=(1,3))
    #             var_temp=var_temp+z_d[n]*var_temp
    #             #var_temp

            #print(np.sum(z_d),k)
            #update the mean values
            mu[k,:]=mean_temp/(np.sum(zd))

            #below code is to find the covaraince and total erro
            for n in range(N):
                x_mu=xi_s[n,:]-mu[k,:]
                #print(x_mu.shape)
                var_temp=(x_mu@x_mu.T)
```

```
                #var_temp=var_temp.reshape(2,600,2,600).sum(axis=(1,3))
                var_temp=var_temp+zd[n]*var_temp
                #var_temp
            covariance[k][:,:]=var_temp/np.sum(zd)


            error=zd.T@(xi_s-mu[k,:])
            err[k]=err[k]+np.linalg.norm(error)

        #total error for each iteration
        tot_err[itr]=np.sum(err)

        #below code is to re-assignments of samples to different bins(mixture components)
        n=0
        for n in range(len(xi_s)):
            xn=xi_s[n][:]-mu
            #print(np.shape(xn))
            min_dist=np.linalg.norm(xn,axis=1)

            z[n]=np.argmin(min_dist)  #store the distance  that is minimum among all the mixture components

        #print(min_dist)

        z[n]=np.argmin(min_dist)
        #print(min_dist)



        if total_err<1e-20:
          #print(f"The iteration no. at which the total_err is less than 1e-20 is {total_err}: corresponding iteration is {itr}")
            break
    return mu,covariance
```

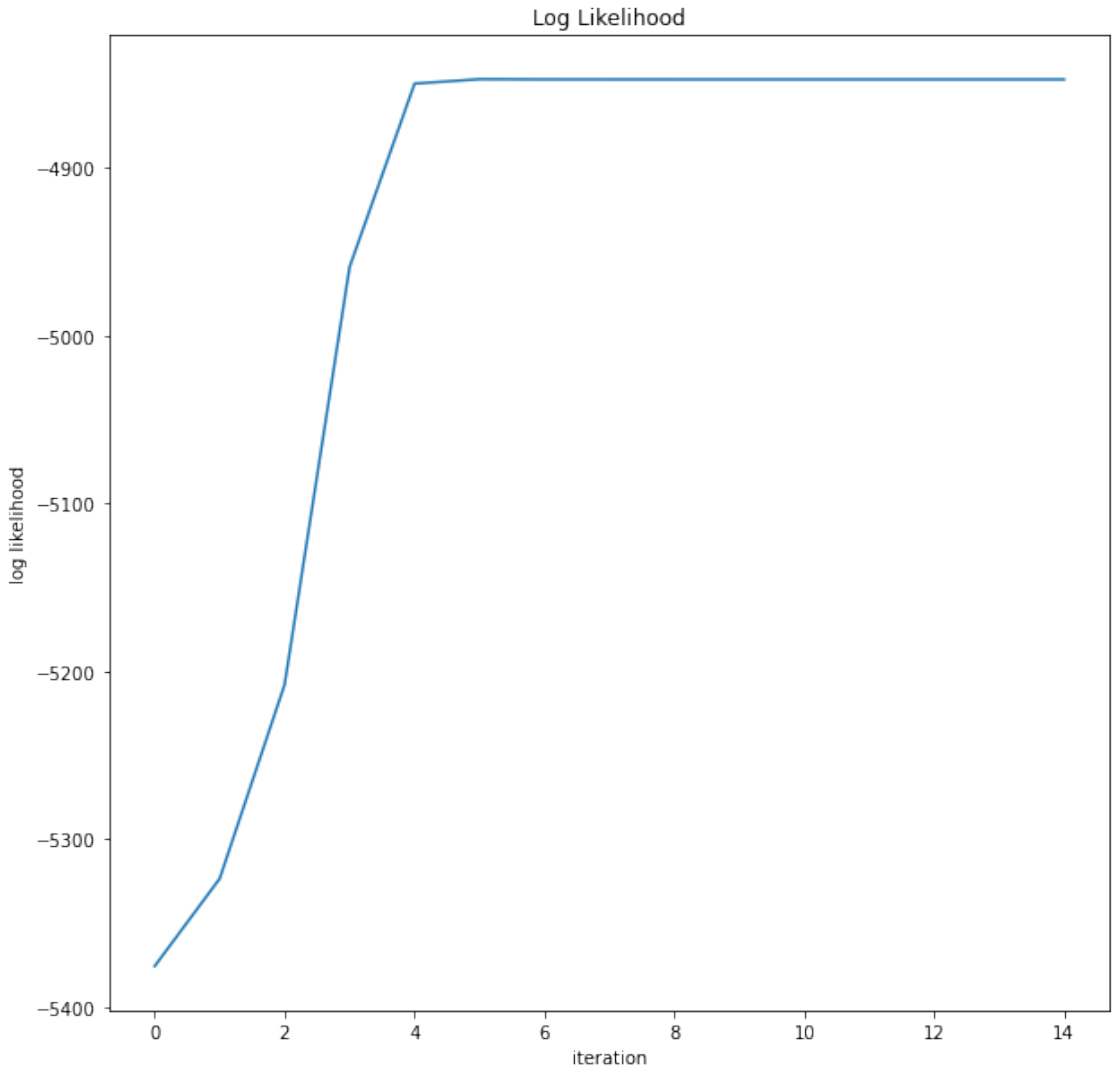
3.0.1 Run Below cell to run GMM for Multivariate case

[88]:
```
K = 2    # Number of mixture components in univariate GMM
N = 1200 # Number of samples to be drawn from each mixture of mulitvariate Gaussian to construct the mixture density
iterations=15
d=2     #dimenion of the data
mean=np.zeros((K,d))

covariance=[]
```

```
#"""Below code is to generate the mean vector and covariance matrix for the␣
 ↪mixture components"""
#mean[0,:]= 0
# covariance_temp= np.ones((d,d))*5
# covariance.append(covariance_temp) #np.linspace(3,6, num=K)

# samples_temp=np.random.multivariate_normal(mean[0,:],covariance_temp)
# samples=np.array(samples_temp)
mean=np.array([[2,2],[10,10]])

covariance_temp=np.array([[5,0],[0,4]])
covariance.append(covariance_temp)

covariance_temp=np.array([[4,0],[0,6]])
covariance.append(covariance_temp)


# for k in range(K):
#    mean[k,:]= np.array(mean[k,:])  # -20+40*np.array()#np.random.rand(1,d)
#     covariance_temp= dataset.make_spd_matrix(d)
#     covariacne_temp=10+5*np.abs(covariance_temp)
    #np.linspace(3,6, num=K)

true_covar=covariance
lambdas   = [0.4,0.6]


samples_lst=np.random.multivariate_normal(mean[0,:],covariance[0])
samples_lst=np.reshape(samples_lst,(1,d))
for n in range(N-1):
    mix_no=np.random.choice(2,p=lambdas)
    sample= np.random.multivariate_normal(mean[mix_no,:],covariance[mix_no])
    sample=np.reshape(sample,(1,d))
    samples_lst=np.append(samples_lst,sample,axis=0)


P={}
#random initialisation of parameters
phi=np.ones((1,K))*(1/K)  #np.random.dirichlet(alpha)  #to genereate lamda␣
 ↪values such that they sum to 1
P["lamda"]=phi [0,:]       #pi[0,:]

#means,covarinace= k_means_init(K,2,samples)
for k in range(K):
    P["mean"+str(k)]= 1+6*np.random.rand(1,d)
```

```
    P["variance"+str(k)]=2+np.random.rand(1)*np.identity(d)

# print(samples.shape)
# print(means.shape)
# print(variances.shape)
#print(samples_lst.shape)
log_likely,P,res_coeff,itera=gmm_algorithm_mulivariate(iterations,P,K,samples_lst)
plt.figure(figsize=(10,10))
plt.title("Log Likelihood")
#plt.axis('off')

plt.plot(log_likely)#plt.plot(np.linspace(1,iter_convg,iter_convg),log_likehood[:
 ↪iter_convg])  #np.linspace(1,iterations,iterations),
plt.xlabel("iteration")
plt.ylabel("log likelihood")
plt.show()

# print(mean)
# print(covariance)

for i in range(K):
    print(f"For component {i}  true mean is {mean[i]}: and covariance is␣
 ↪{covariance[i]}")

for i in range(K):
    print(f"For some component mean is : ",end='')
    print(np.squeeze(P["mean"+str(i)]),end=' , variance is : ')
    print(np.squeeze(P["variance"+str(i)]),end=' and the weight is : ')
    print(np.squeeze(P["lamda"][i]))



N=1000
samples_test=np.zeros((N,d))
x= np.linspace(np.min(mean)-10,np.max(mean)+10,N)
y= np.linspace(np.min(mean)-10,np.max(mean)+10,N)


X, Y = np.meshgrid(x,y)
#print(samples_test.shape)
pos = np.empty(X.shape + (2,))
pos[:, :, 0] = X
pos[:, :, 1] = Y

mu_k=mean[0,:].ravel()
print(mu_k)
Z = multivariate_normal.pdf(pos, mu_k ,covariance[0])
```

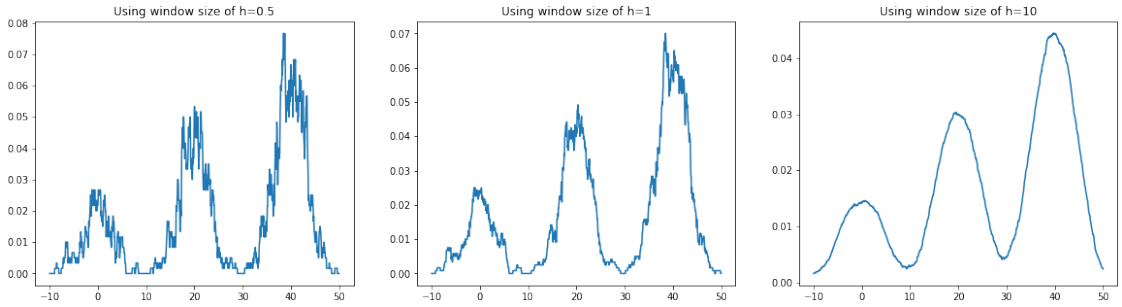
```
for k in range(K-1):

    mu_k=mean[k+1,:]
    mu_k=mu_k.ravel()
    Z =Z+ multivariate_normal.pdf(pos, mu_k, covariance[k+1])

# Create a surface plot and projected filled contour plot under it.
fig = plt.figure(figsize=(30,30))

plt.title("Actual Mixture Model",fontsize=20)
ax = fig.gca(projection='3d')

ax.plot_surface(X, Y, Z, rstride=3, cstride=3, linewidth=1,cmap=cm.viridis)
ax.view_init(25,-30)
plt.show()

N=1000
samples_test=np.zeros((N,d))
x= np.linspace(np.min(mean)-10,np.max(mean)+10,N)
y= np.linspace(np.min(mean)-10,np.max(mean)+10,N)


X, Y = np.meshgrid(x,y)
#print(samples_test.shape)
pos = np.empty(X.shape + (2,))
pos[:, :, 0] = X
pos[:, :, 1] = Y

mu_k=P["mean"+str(0)].ravel()
mu_k=mu_k.ravel()
print(mu_k)
Z = multivariate_normal.pdf(pos, mu_k ,P["variance"+str(0)])
for k in range(K-1):

    mu_k=P["mean"+str(k+1)]
    mu_k=mu_k.ravel()
    Z =Z+ multivariate_normal.pdf(pos, mu_k, P["variance"+str(k+1)])

# Create a surface plot and projected filled contour plot under it.
fig = plt.figure(figsize=(30,30))

plt.title("Estimated Mixture Model using GMM",fontsize=20)
ax = fig.gca(projection='3d')

ax.plot_surface(X, Y, Z, rstride=3, cstride=3, linewidth=1,cmap=cm.viridis)
ax.view_init(25,-30)
plt.show()
```



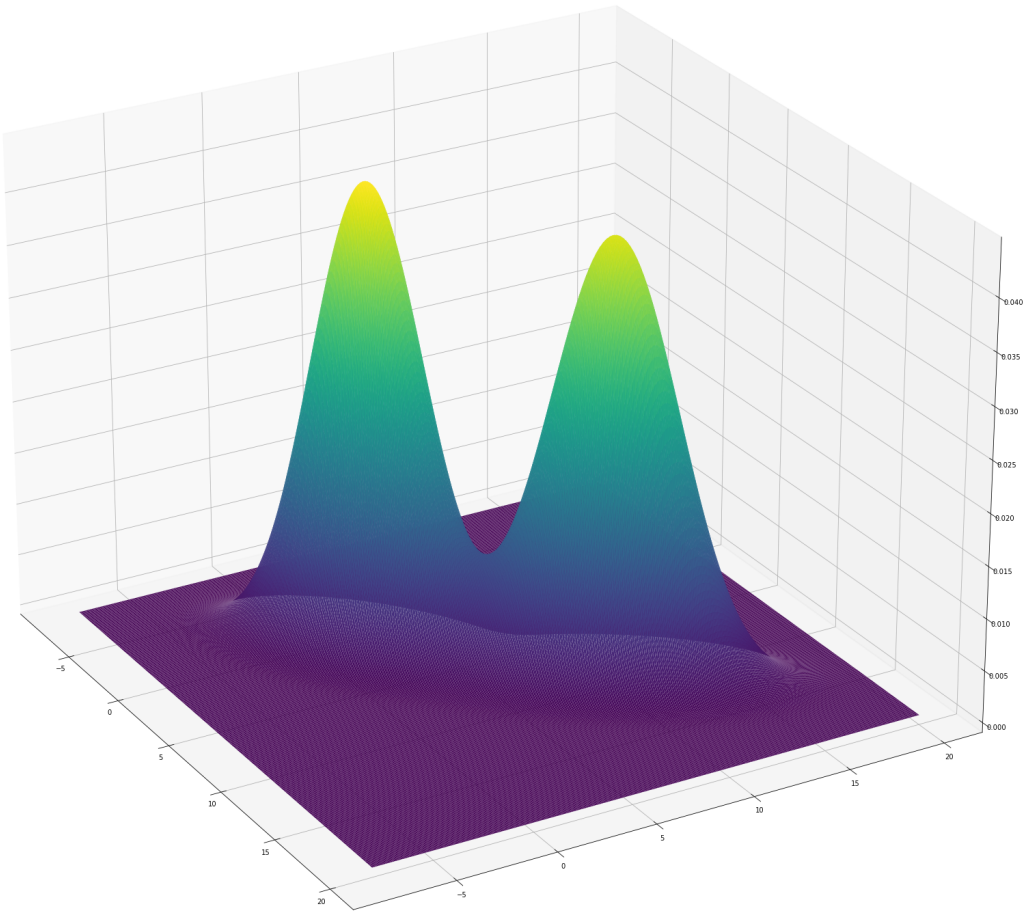

```
For component 0  true mean is [2 2]: and covariance is [[5 0]
 [0 4]]
For component 1  true mean is [10 10]: and covariance is [[4 0]
 [0 6]]
For some component mean is : [9.99657183 9.94757115] , variance is :
[[3.90200743 3.25208087]
 [3.25208087 6.37863146]] and the weight is : 0.5991562068561024
For some component mean is : [1.89381842 1.9342065 ] , variance is : [[5.5229091
2.89153448]
 [2.89153448 3.82260309]] and the weight is : 0.400843793143898
[2 2]
```

```
<ipython-input-88-5e4d52f473ae>:108: UserWarning: Requested projection is
different from current axis projection, creating new axis with requested
projection.
```

```
  ax = fig.gca(projection='3d')
<ipython-input-88-5e4d52f473ae>:108: MatplotlibDeprecationWarning: Adding an
axes using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and
returned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  ax = fig.gca(projection='3d')
```




```
[9.99657183 9.94757115]

<ipython-input-88-5e4d52f473ae>:140: UserWarning: Requested projection is
different from current axis projection, creating new axis with requested
projection.
  ax = fig.gca(projection='3d')
<ipython-input-88-5e4d52f473ae>:140: MatplotlibDeprecationWarning: Adding an
axes using the same arguments as a previous axes currently reuses the earlier
instance.  In a future version, a new instance will always be created and
returned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  ax = fig.gca(projection='3d')
```



3.0.2 K-Means for Question 2

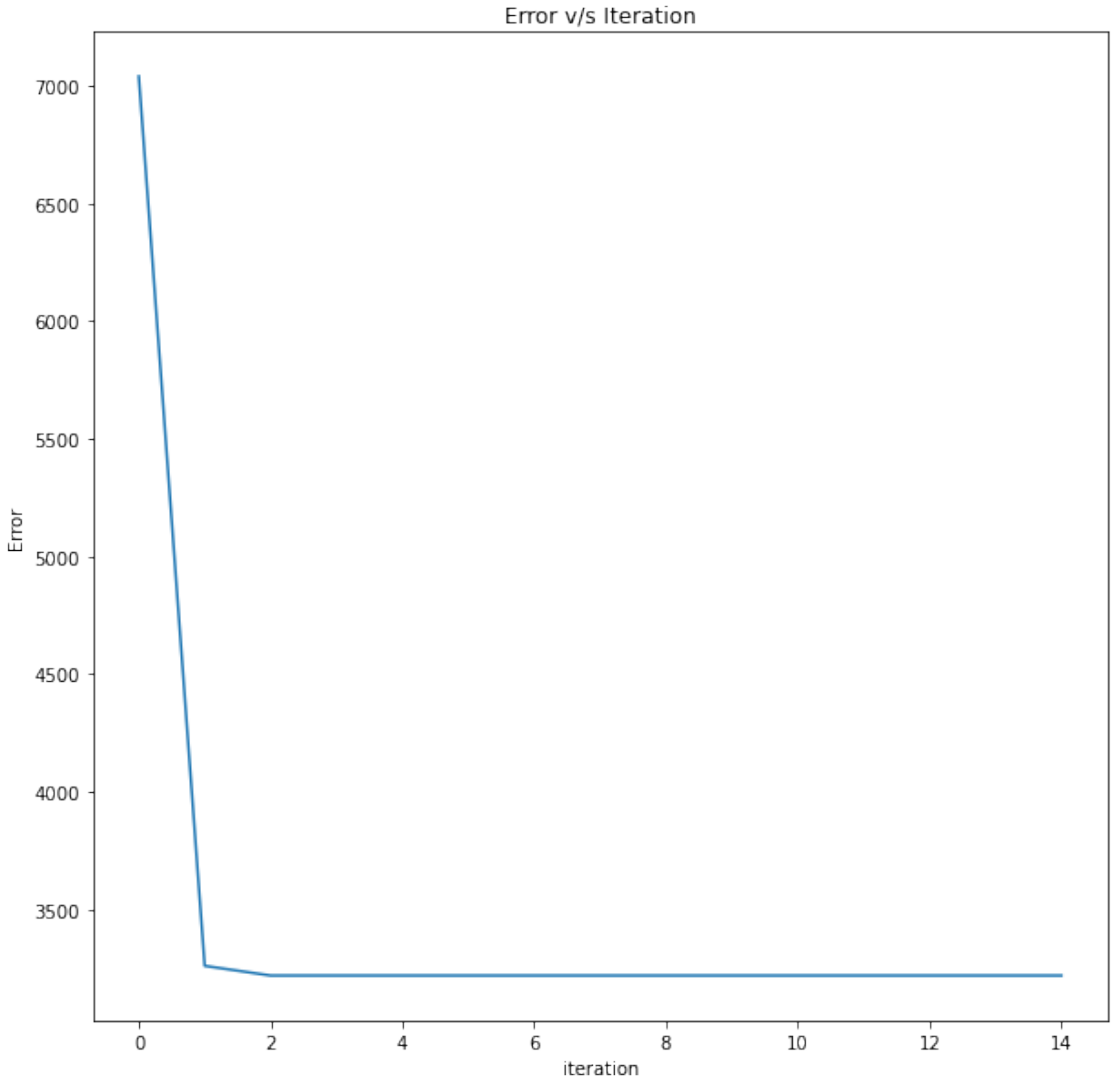
[87]:
```
K = 2     # Number of mixture components in univariate GMM
N = 1200 # Number of samples to be drawn from each mixture of mulitvariate␣
 ↪Gaussian to construct the mixture density
iterations=15
d=2      #dimenion of the data
mean=np.zeros((K,d))

covariance=[]
```

```
#"""Below code is to generate the mean vector and covariance matrix for the␣
 ↪mixture components"""
#mean[0,:]= 0
# covariance_temp= np.ones((d,d))*5
# covariance.append(covariance_temp) #np.linspace(3,6, num=K)

# samples_temp=np.random.multivariate_normal(mean[0,:],covariance_temp)
# samples=np.array(samples_temp)
mean=np.array([[2,2],[10,10]])

covariance_temp=np.array([[5,0],[0,4]])
covariance.append(covariance_temp)

covariance_temp=np.array([[4,0],[0,6]])
covariance.append(covariance_temp)


# for k in range(K):
#    mean[k,:]= np.array(mean[k,:])  # -20+40*np.array()#np.random.rand(1,d)
#     covariance_temp= dataset.make_spd_matrix(d)
#     covariacne_temp=10+5*np.abs(covariance_temp)
    #np.linspace(3,6, num=K)

true_covar=covariance
lambdas   = [0.4,0.6]


samples_lst=np.random.multivariate_normal(mean[0,:],covariance[0])
samples_lst=np.reshape(samples_lst,(1,d))
for n in range(N-1):
    mix_no=np.random.choice(2,p=lambdas)
    sample= np.random.multivariate_normal(mean[mix_no,:],covariance[mix_no])
    sample=np.reshape(sample,(1,d))
    samples_lst=np.append(samples_lst,sample,axis=0)


true_covar=covariance


z=np.random.randint(0,K,size=(samples_lst.shape[0],1))
tot_err=np.zeros((iterations,1))
d=samples_lst.shape[1]   #dimesnion of the data
N=samples_lst.shape[0]   #size of the data

#true_covar=covariance
for itr in range(iterations):
    mu=np.zeros((K,d))
```

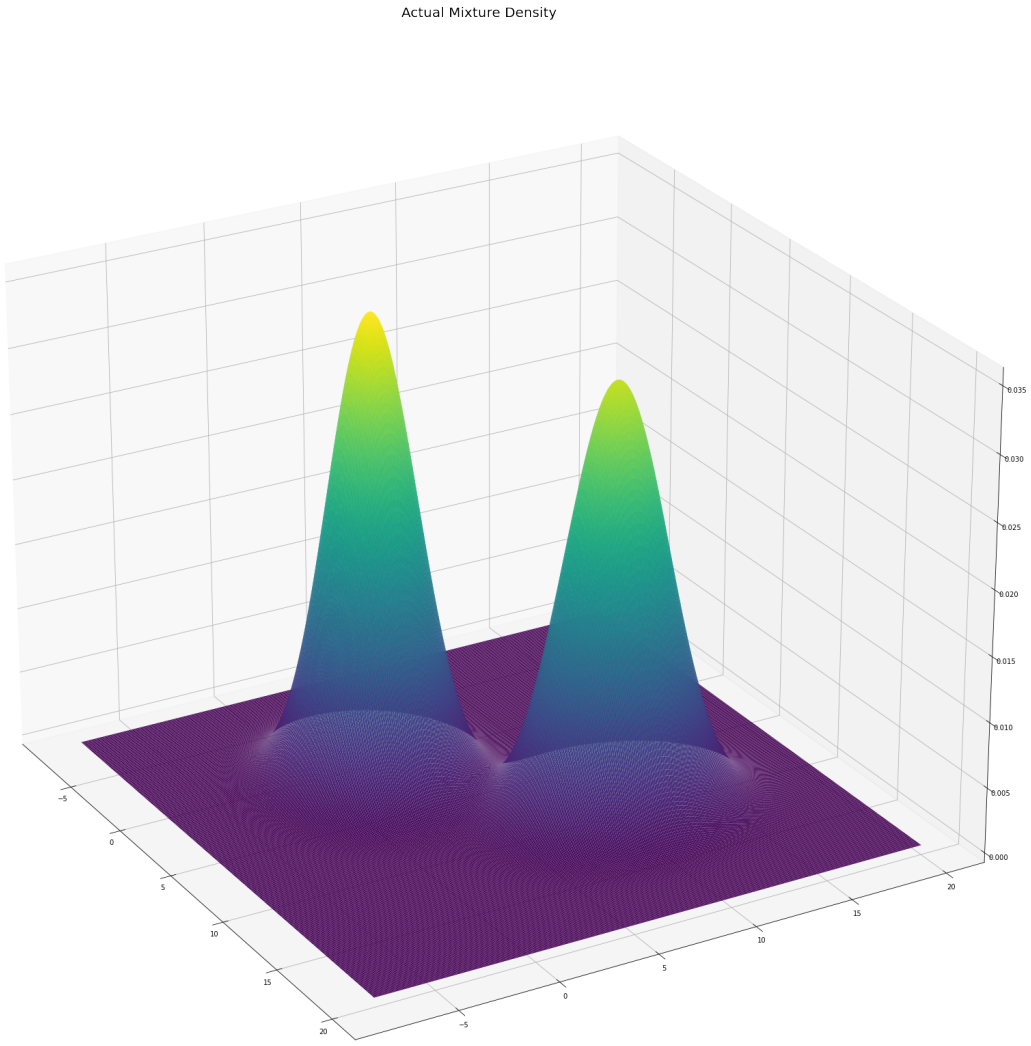
```
    covariance=np.zeros((K,d,d))

    err=np.zeros((K,1))
    for k in range(K):
        mean_temp=np.zeros((1,d))
        var_temp=np.zeros((d,d))

        zd=np.where(z==k,1,0)
        for n in range(N):
            xn=samples_lst[n,:]
            #find the mean value of kth mixture component using the samples that␣
↪belong to the kth mixture in ith class
            mean_temp=mean_temp+(zd[n]*xn)
            mean_temp=np.reshape(mean_temp,(1,d))

        mu[k,:]=mean_temp/(np.sum(zd))
        for n in range(N):
            xn=samples_lst[n,:]
            #find the mean value of kth mixture component using the samples that␣
↪belong to the kth mixture in ith class
            x_mu=xn-mu[k,:]
            x_mu=np.reshape(x_mu,(d,1))

            var_temp=var_temp+zd[n]*np.abs(np.outer(x_mu,x_mu))
            #var_temp
        covariance[k,:,:]=var_temp/np.sum(zd)

    err_itr=0
    for n in range(N):
        xn=samples_lst[n,:]

        for k in range(K):
            zd=np.where(z==k,1,0)
            norm_val=np.linalg.norm(xn-mu[k,:])

            err_itr=err_itr + zd[n]*norm_val

    #total error for each iteration
    tot_err[itr]=err_itr

    #below code is to re-assignments of samples to different bins(mixture␣
↪components)
    n=0
    for n in range(N):
        xn=samples_lst[n][:]-mu
        #print(np.shape(xn))
        min_dist=np.linalg.norm(xn,axis=1)
```

```
        z[n]=np.argmin(min_dist)  #store the distance  that is minimum among all␣
 ↪the mixture components

    #print(min_dist)

    z[n]=np.argmin(min_dist)
    #print(min_dist)

#     if err_itr<1:
#       print(f"The iteration no. at which the total_err is less than 1e-20 is␣
 ↪{total_err}: corresponding iteration is {itr}")
#       break

plt.figure(figsize=(10,10))
plt.title("Error v/s Iteration")
#plt.axis('off')

plt.plot(tot_err)#plt.plot(np.linspace(1,iter_convg,iter_convg),log_likehood[:
 ↪iter_convg])  #np.linspace(1,iterations,iterations),
plt.xlabel("iteration")
plt.ylabel("Error")
plt.show()


for i in range(K):
    print(f"For component {i}  true mean is {mean[i]}: and covariance is␣
 ↪{true_covar[i]}")
for i in range(K):
    print(f"For some component predicted mean is {mu[i]}: and covariance is␣
 ↪{covariance[i]}")

N=1000
samples_test=np.zeros((N,d))
x= np.linspace(np.min(mean)-10,np.max(mean)+10,N)
y= np.linspace(np.min(mean)-10,np.max(mean)+10,N)


X, Y = np.meshgrid(x,y)
#print(samples_test.shape)
pos = np.empty(X.shape + (2,))
pos[:, :, 0] = X
pos[:, :, 1] = Y

mu_k=mean[0,:].ravel()
print(mu_k)
Z = multivariate_normal.pdf(pos, mu_k ,true_covar[0])
```

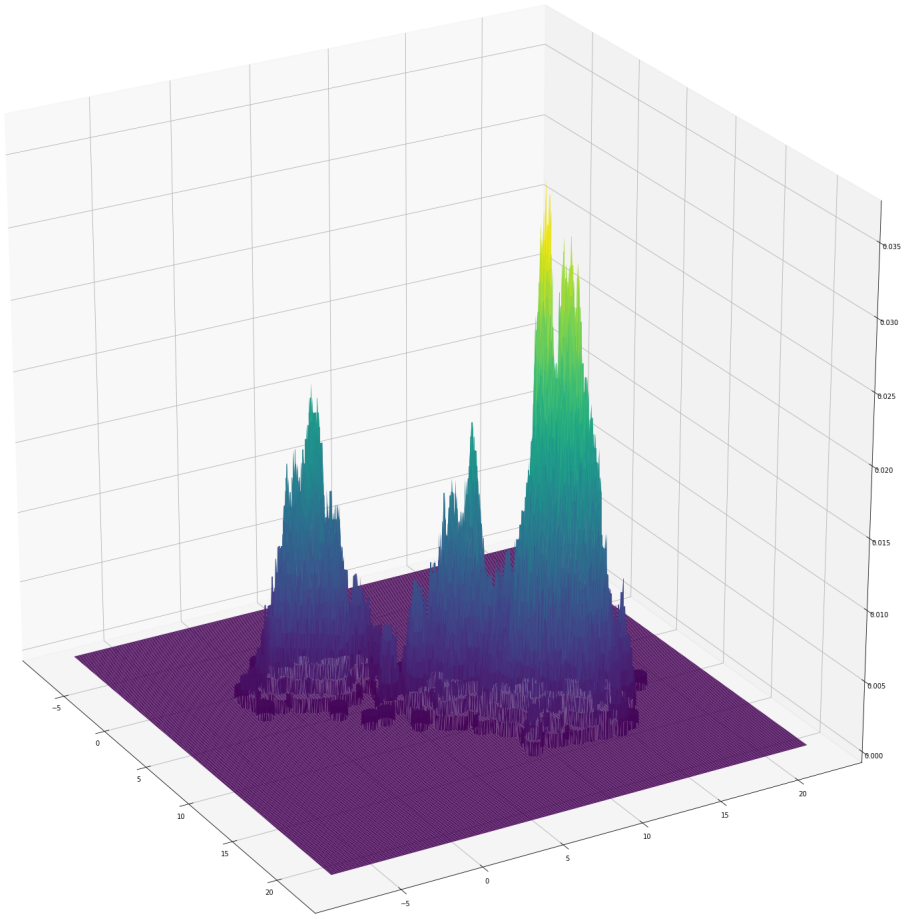
```
for k in range(K-1):

    mu_k=mean[k+1,:]
    mu_k=mu_k.ravel()
    Z =Z+ multivariate_normal.pdf(pos, mu_k, true_covar[k+1])


# Create a surface plot and projected filled contour plot under it.
fig = plt.figure(figsize=(30,30))

ax = fig.gca(projection='3d')
plt.title("Actual Mixture Density",fontsize=20)
ax.plot_surface(X, Y, Z, rstride=3, cstride=3, linewidth=1,cmap=cm.viridis)
ax.view_init(25,-30)
plt.show()

N=1000
samples_test=np.zeros((N,d))
x= np.linspace(np.min(mean)-10,np.max(mean)+10,N)
y= np.linspace(np.min(mean)-10,np.max(mean)+10,N)


X, Y = np.meshgrid(x,y)
#print(samples_test.shape)
pos = np.empty(X.shape + (2,))
pos[:, :, 0] = X
pos[:, :, 1] = Y

mu_k=mu[0,:].ravel()

print(mu_k)
Z = multivariate_normal.pdf(pos, mu_k ,covariance[0])
for k in range(K-1):

    mu_k=mu[k+1,:]
    mu_k=mu_k.ravel()
    Z =Z+ multivariate_normal.pdf(pos, mu_k, covariance[k+1])


# Create a surface plot and projected filled contour plot under it.
fig = plt.figure(figsize=(30,30))
ax = fig.gca(projection='3d')
plt.title("Estimated Mixture density using K-means",fontsize=20)
ax.plot_surface(X, Y, Z, rstride=3, cstride=3, linewidth=1,cmap=cm.viridis)
ax.view_init(25,-30)
plt.show()
```




```
For component 0  true mean is [2 2]: and covariance is [[5 0]
 [0 4]]
For component 1  true mean is [10 10]: and covariance is [[4 0]
 [0 6]]
For some component predicted mean is [2.00260106 1.98103974]: and covariance is
[[5.07580097 2.6612505 ]
 [2.6612505  3.48479035]]
For some component predicted mean is [10.04160992 10.02830998]: and covariance
is [[3.78677132 3.05521335]
 [3.05521335 6.09037959]]
[2 2]
```



[2.00260106 1.98103974]



4 Kernel Density Estimate

4.1 Multivariate gaussian mixture model:

[ ]:
```
!pip install plotly
```

[2]:
```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
from itertools import product, combinations
import os
import scipy.stats as stat
from sklearn.metrics import confusion_matrix
```

```
from sklearn.mixture import GMM
import seaborn as sns
import pandas as pd
import cv2
import sklearn
import sklearn.datasets as dataset
from sklearn.mixture import GaussianMixture
from scipy.stats import multivariate_normal
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import plotly.express as px
```

[ ]:
```
def gmm_algorithm_mulivariate(iterations,P_em,K,clas):

    """GMM ALGORITHM
    Input :
    iterations -- Maximum no. of iterations
    P_em       -- dictionary that contains parameters and weightages of all␣
 ↪mixture components
    K          -- no. of mixture components

    Return:
    P_em -- Updated dictionary that contains parameters and weightages of all␣
 ↪mixture components
    res_coeff  -- contains the responsibility coefficient of all images w.r.t␣
 ↪each mixture
    log_likey  -- contains log likelihood of each iteration

    """



    ll_old=0
    ll_new=0
    log_likely= np.zeros((iterations,1))
    d=clas.shape[1]
    N=clas.shape[0]

    for itr in range(iterations):
        #responsibility coefficient of size(no. of samples in clas x mixture␣
 ↪components)
        res_coeff=np.zeros((len(clas),K))

        for n in range(len(clas)):
            res_coeff_nr=np.zeros((1,K))       #to store respob. coeff of each␣
 ↪mixture component
            xn=clas[n][:]
            xn=np.reshape(xn,(d,1))
```

```
            #print(xn.shape)
            for k in range(K):

                mu=P_em["mean"+str(k)]  #stores the mean of kth component
                mu=np.reshape(mu,(d,1))
                var=P_em["variance"+str(k)]
                var=np.abs(var)
                #below code is to find Gaussian(x_n,theta_k)

                var_inv=np.linalg.inv(var+np.identity(d)*0.01)
                fx_pi=(1/np.sqrt((2*np.pi)))**(d)
                fx_var=np.abs(np.linalg.det(var))

                fx_var=np.abs(1/np.sqrt(fx_var))
                muSmu=((xn-mu).T@var_inv@(xn-mu))*0.5


                f_x_k=fx_pi*fx_var*np.exp(-muSmu)

                #update the responsibility coefficient for kth component
                res_coeff_nr[0,k]=P_em["lamda"][k]*f_x_k
                #print(f"value of res : {res_coeff_nr[0,:]}, {itr}")


            # respob. coeff of nth sample with kth mixture
            #added 1e-50 to the denominator so that responsibility coefficient␣
↪doesn't become undefined value (0/0 form)
            res_coeff[n,:]=res_coeff_nr[0,:]/(np.sum(res_coeff_nr)+1e-50)


        #EXPECTATION - STEP
        N_k=np.sum(res_coeff,axis=0)
        #print(np.sum(N_k))
        #update the lamda parameter for all mixture components
        P_em["lamda"]=N_k/N #len(clas)
        #P_em["lamda"]=P_em["lamda"]/np.sum(P_em["lamda"])
        #print(np.sum(P_em["lamda"]))

        #print(f"Value of {np.sum(N_k)},{itr}")
        #below code is to update the mean and variances for the mixture␣
↪components
        #xi=clas
        for k in range(K):
            mean_temp=np.zeros((d,1))
            #var_temp=np.zeros((d,d))
            for n in range(len(clas)):
                xi=clas[n,:]
```

```
                xi=np.reshape(xi,(d,1))

                res_coeff_k=res_coeff[n,k]
                #res_coeff_k=np.reshape(res_coeff_k,(1,len(clas)))

                #MAXIMISATION - STEP
                #added 1e-50 to the denominator so that responsibility␣
 ↪coefficient doesn't become undefined value (0/0 form)
                temp_mu=P_em["mean"+str(k)]
                temp_mu=np.reshape(temp_mu,(d,1))
                xi_mu=xi-temp_mu
                sigma=np.outer(xi_mu,xi_mu)
#                 print(f"xi_mu {np.shape(xi_mu)}")
#                 print(f"sigma {np.shape(sigma)}")
#                 var_temp=np.linalg.inv(sigma)
#                 print(np.shape(var_temp),np.shape(res_coeff_k))
                mean_temp= mean_temp+(res_coeff_k*xi)
                #var_temp= var_temp+np.abs((res_coeff_k*(sigma)))

            #update the mean and variances of kth class

            P_em["mean"+str(k)]= mean_temp/(N_k[k]+1e-50)
            for k in range(K):
              #mean_temp=np.zeros((d,1))
              var_temp=np.zeros((d,d))
              for n in range(len(clas)):
                  xi=clas[n,:]
                  xi=np.reshape(xi,(d,1))

                  res_coeff_k=res_coeff[n,k]
                  #res_coeff_k=np.reshape(res_coeff_k,(1,len(clas)))

                  #MAXIMISATION - STEP
                  #added 1e-50 to the denominator so that responsibility␣
 ↪coefficient doesn't become undefined value (0/0 form)
                  temp_mu=P_em["mean"+str(k)]
                  temp_mu=np.reshape(temp_mu,(d,1))
                  xi_mu=xi-temp_mu
                  sigma=np.outer(xi_mu,xi_mu)
 #                 print(f"xi_mu {np.shape(xi_mu)}")
 #                 print(f"sigma {np.shape(sigma)}")
 #                 var_temp=np.linalg.inv(sigma)
 #                 print(np.shape(var_temp),np.shape(res_coeff_k))
                  #mean_temp= mean_temp+(res_coeff_k*xi)
                  var_temp= var_temp+np.abs((res_coeff_k*(sigma)))

              P_em["variance"+str(k)]= np.abs(var_temp)/(N_k[k]+1e-50)
```

```
        """Below code is find the log likelihood for the parameters obtained in␣
 ↪the current iteration"""
        ll=0
        for n in range(len(clas)):
            x=clas[n][:]
            x=np.reshape(x,(d,1))
            for k in range(K):

                mu=P_em["mean"+str(k)]  #stores the mean of kth component
                mu=np.reshape(mu,(d,1))
                var=np.abs(P_em["variance"+str(k)])

                var_inv=np.abs(np.linalg.inv(var+np.identity(d)*1e-5)) #to avoid␣
 ↪singularity
                fx_pi=(1/np.sqrt((2*np.pi)))**(d)
                fx_var=np.abs(1/(np.linalg.det(var+np.identity(d)*1e-5)))
                #fx_var=np.sqrt(fx_var)
                muSmu=((x-mu).T@var_inv@(x-mu))*0.5


                f_x_k=np.log(fx_pi)+0.5*np.log(fx_var)-muSmu

                ll_update=res_coeff[n,k]*(f_x_k+np.log(P_em["lamda"][k]+0.01))␣
 ↪#(xi@p_i.T+(1-xi)@(p_i_minus).T
                ll=ll+ll_update

        ll_old=ll_new
        ll_new=ll
#       print(f"log likelihood : {ll_new},old : {ll_old}")

        log_likely[itr]=ll_new



#         if (abs(ll_new-ll_old)<1 and itr>=1):
#             #log_likely[itr+1:iterations]=log_likely[itr]
#             print(f"For iteration {itr} the difference between new log␣
 ↪likelihood and old log likelihood is less than 1")
#             break

    return log_likely,P_em,res_coeff,itr
```

```
[ ]: def k_means_init(K,iteration_max,samples):
        iterations=iteration_max   #no. of iterations
        K=K
        z=np.random.randint(0,K,size=(samples.shape[0],1))
        tot_err=np.zeros((iterations,1))
        d=samples.shape[1]   #dimesnion of the data
        N=samples.shape[0]   #size of the data

        for itr in range(iterations):
            total_err=0
            mu=np.zeros((K,d))


            #print(np.shape(covariance))
            xi_s=samples
            xi_s=np.array(xi_s)

            #print(np.shape(xi_s))
            err=np.zeros((K,1))
            for k in range(K):
                mean_temp=np.zeros((1,d))
                var_temp=np.zeros((d,d))

                zd=np.where(z==k,1,0)
                for n in range(N):

                    #find the mean value of kth mixture component using the samples␣
    ↪that belong to the kth mixture in ith class
                    mean_temp=mean_temp+(zd[n]*xi_s[n,:])
                    mean_temp=np.reshape(mean_temp,(1,d))
        #             x_mu=xi_s[n,:]-mu[k,:]
        #             #print(x_mu.shape)
        #             var_temp=(x_mu@x_mu.T)
        #             #var_temp=var_temp.reshape(2,600,2,600).sum(axis=(1,3))
        #             var_temp=var_temp+z_d[n]*var_temp
        #             #var_temp

                #print(np.sum(z_d),k)
                #update the mean values
                mu[k,:]=mean_temp/(np.sum(zd))
                covariance=np.zeros((K,d,d))

                #below code is to find the covaraince and total erro
                for n in range(N):
                    x_mu=xi_s[n,:]-mu[k,:]
                    x_mu=np.reshape(x_mu,(d,1))
                    var_temp=np.abs(zd[n]*x_mu@x_mu.T)
```

```
                #var_temp=var_temp.reshape(2,600,2,600).sum(axis=(1,3))
                var_temp=var_temp+var_temp
                #var_temp
            covariance[k][:,:]=var_temp/np.sum(zd)


            error=zd.T@(xi_s-mu[k,:])
            err[k]=err[k]+np.linalg.norm(error)

        #total error for each iteration
        tot_err[itr]=np.sum(err)

        #below code is to re-assignments of samples to different bins(mixture␣
↪components)
        n=0
        for n in range(len(xi_s)):
            xn=xi_s[n][:]-mu
            #print(np.shape(xn))
            min_dist=np.linalg.norm(xn,axis=1)

            z[n]=np.argmin(min_dist)  #store the distance  that is minimum among␣
↪all the mixture components

        #print(min_dist)

        z[n]=np.argmin(min_dist)
        #print(min_dist)



        if total_err<1e-20:
          #print(f"The iteration no. at which the total_err is less than 1e-20␣
↪is {total_err}: corresponding iteration is {itr}")
          break
    return mu,covariance
```

[ ]:
```
K =3      # Number of mixture components in univariate GMM
N = 200 # Number of samples to be drawn from each mixture of mulitvariate␣
 ↪Gaussian to construct the mixture density
iterations=10
d=4     #dimenion of the data
mean=np.zeros((K,d))

covariance=[]
```

```
#"""Below code is to generate the mean vector and covariance matrix for the␣
 ↪mixture components"""
#mean[0,:]= 0
# covariance_temp= np.ones((d,d))*5
# covariance.append(covariance_temp) #np.linspace(3,6, num=K)
samples=[]
#samples_temp=np.random.multivariate_normal(mean[0,:],covariance_temp,200)
#samples=np.array(samples_temp)
for k in range(K):
    mean[k,:]= -20+40*np.random.rand(1,d)
    covariance_temp= sklearn.datasets.make_spd_matrix(d)
    covariance_temp= np.identity(d)*(k+1) #np.
 ↪abs(covariance_temp@covariance_temp.T)
    covariance.append(covariance_temp) #np.linspace(3,6, num=K)
    samples_temp=np.random.multivariate_normal(mean[k,:],covariance_temp,N*(k+1))
    samples=np.append(samples,samples_temp)
    print(covariance_temp)

true_covar=covariance
samples=np.array(samples)
samples=np.reshape(samples,(6*N,d))


#print(samples[750:799,:])
#print(np.shape(samples),np.shape(covariance[2]),np.shape(mean))
    #lambdas   = [0.5,0.5,0.5]

# samples_lst = [np.random.normal(loc = means[k],scale = np.
 ↪sqrt(variances[k]),size = int(2500)) for k in range(K)]
# samples     = np.expand_dims(np.array([item for sublist in samples_lst for␣
 ↪item in sublist]),axis=1)
# sns.displot(samples, color='b')
# plt.hist(samples, bins =10,density=True)
# plt.show()

P_init={}
#random initialisation of parameters
pi=np.ones((1,K))*(1/K)  #np.random.dirichlet(alpha)  #to genereate lamda values␣
 ↪such that they sum to 1
P_init["lamda"]=np.array([0.1,0.85,0.05])       #pi[0,:]

#means,covariance_k= k_means_init(K,2,samples)
#print(covariance_k)
for k in range(K):
    P_init["mean"+str(k)]=8*np.random.rand(1,d) # means[k,:] #
    P_init["variance"+str(k)]=np.identity(d)#covariance_k[k][:,:]
```

```
# print(samples.shape)
# print(means.shape)
#print(covariance)
log_likely,P,res_coeff,itera=gmm_algorithm_mulivariate(iterations,P_init,K,samples)
plt.figure(figsize=(10,10))
plt.title("Log Likelihood")
#plt.axis('off')

plt.plot(log_likely)#plt.plot(np.linspace(1,iter_convg,iter_convg),log_likehood[:
 ↪iter_convg])  #np.linspace(1,iterations,iterations),
plt.xlabel("iteration")
plt.ylabel("log likelihood")
plt.show()

# print(mean)
# print(covariance)

for i in range(K):
    print(f"For component {i}  true mean is {mean[i]}: and covariance is␣
 ↪{covariance[i]}")

for i in range(K):
    print(f"For some component mean is : ",end='')
    print(np.squeeze(P["mean"+str(i)]),end=' , variance is : ')
    print(np.squeeze(P["variance"+str(i)]),end=' and the weight is : ')
    print(np.squeeze(P["lamda"][i]))
```

4.1.1 Parzen window for Univariate Gaussian Mixture Model

[8]:
```
def rectangular(sample_test,samples,h):

  d=samples.shape[1]  #dimension of the samples
  k=0
  N=samples.shape[0]  #total no. of samples
  for x in samples:
    if np.linalg.norm(sample_test-x)<h/2:    #sample_test : The point at which␣
 ↪the density needs to be estimated
      k=k+1
  parzen_est=k/(N*(h**d))      #this is the parzen window estimate using␣
 ↪rectangular window
  return parzen_est
```

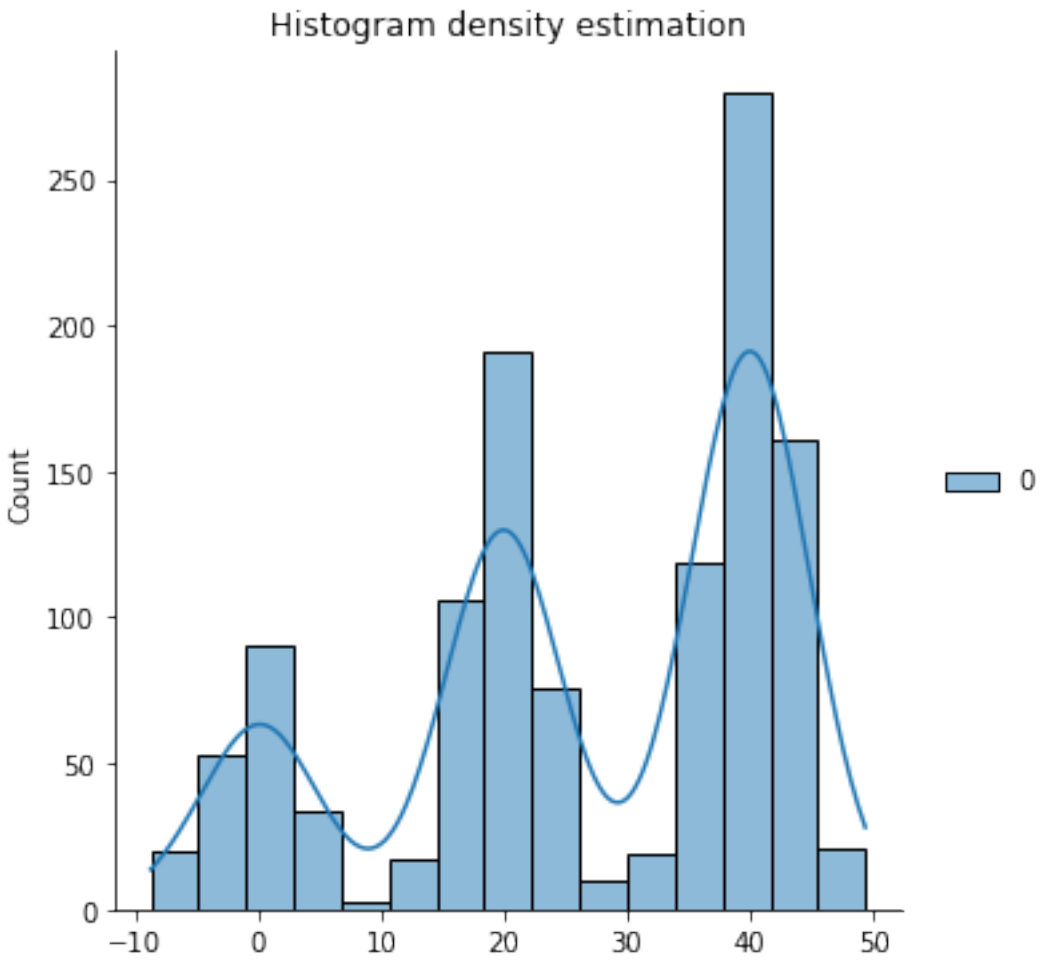
[7]:
```
def gaussian(sample_test,samples,h):

  d=samples.shape[1]  #dimension of the samples
```

```
  k=0
  N=samples.shape[0] #total no. of samples
  f_x=0
  for x in samples:   #sample_test : The point at which the density needs to be␣
 ↪estimated
    f_xi=np.exp(-(np.linalg.norm(sample_test-x)/(2*(h**2))))    #estimate the␣
 ↪density at sample_test using gaussian density function as the kernel function
    f_x=f_x+f_xi
  f_x=((1/np.sqrt(2*np.pi))**d)*(1/(N*(h**d)))*f_x   #this is the parzen window␣
 ↪estimate using gaussian window
  return f_x
```

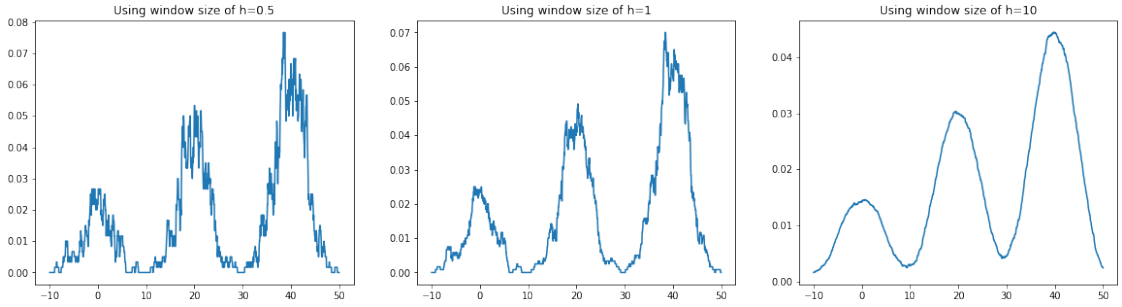
[9]:
```
K = 3     # Number of mixture components in univariate GMM
N = 200 # Number of samples to be drawn from each mixture of mulitvariate␣
 ↪Gaussian to construct the mixture density
iterations=10
d=1     #dimenion of the data
mean=np.zeros((K,d))

covariance=[]


#"""Below code is to generate the mean vector and covariance matrix for the␣
 ↪mixture components"""
#mean[0,:]= 0
# covariance_temp= np.ones((d,d))*5
# covariance.append(covariance_temp) #np.linspace(3,6, num=K)
samples=[]
#samples_temp=np.random.multivariate_normal(mean[0,:],covariance_temp,200)
#samples=np.array(samples_temp)

#below code is to generate samples using which density estimation is done
for k in range(K):
    mean[k,:]= k*20#-20+80*np.random.rand(1,d)
    covariance_temp= sklearn.datasets.make_spd_matrix(d)
    covariance_temp= np.identity(d)*(10) #np.abs(covariance_temp@covariance_temp.
 ↪T)
    covariance.append(covariance_temp) #np.linspace(3,6, num=K)
    samples_temp=np.random.multivariate_normal(mean[k,:],covariance_temp,N*(k+1))
    samples=np.append(samples,samples_temp)
    #print(covariance_temp)

true_covar=covariance
samples=np.array(samples)
samples=np.reshape(samples,(int((K+1)*K*N*0.5),d))
```

```
#generate the testing samples at which density is to be estimated
samples_test=np.linspace(-10,50,1200)
#print(samples_test.shape)

fig = plt.figure(figsize=(10,10))
#ax = fig.gca(projection='3d')


#plot histogram density estimation of the samples
sns.displot(samples, color='b',kde=True)
plt.title(f"Histogram density estimation")
plt.show()
dens_est=np.zeros((len(samples_test),1))


h=[0.5,1,10]  #for selecting different window size
kernel_sel=["rectangular", "gaussian"]
for kf in kernel_sel:
  print(f"Density estimation using Kernel fuctions being {kf} window")
  fig = plt.figure(figsize=(20,5))
  for window in h:
    for i in range(len(samples_test)):
      if kf=="rectangular":
        dens_est[i]=rectangular(samples_test[i],samples,window) #estimate the␣
 ↪density at a given point using rectangular  window kernel function
      else :
        dens_est[i]=gaussian(samples_test[i],samples,window)   #estimate the␣
 ↪density at a given point using gaussian  window kernel function
      #print(f"At {sample} the estimated density is {dens_est[i]}")
    #print(dens_est)
    #plot the estimated density using the parzen window
    plt.subplot(1,len(h),h.index(window)+1)
    plt.plot(samples_test,dens_est)
    plt.title(f"Using window size of h={window}")
  plt.show()
```

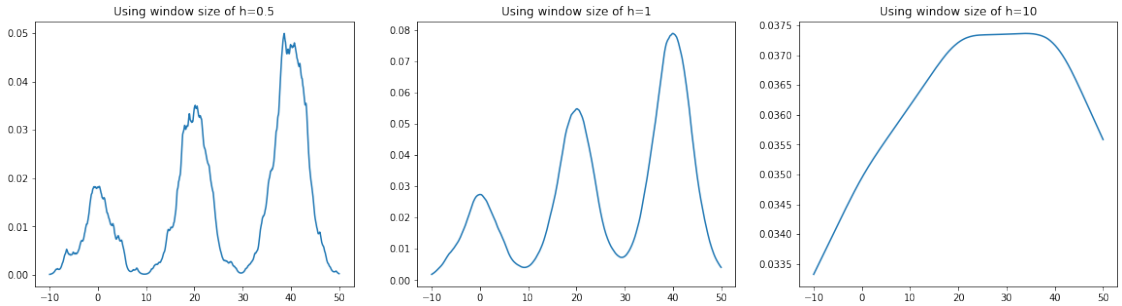
```
<Figure size 720x720 with 0 Axes>
```



Density estimation using Kernel fuctions being rectangular window



Density estimation using Kernel fuctions being gaussian window



4.1.2 Observations :

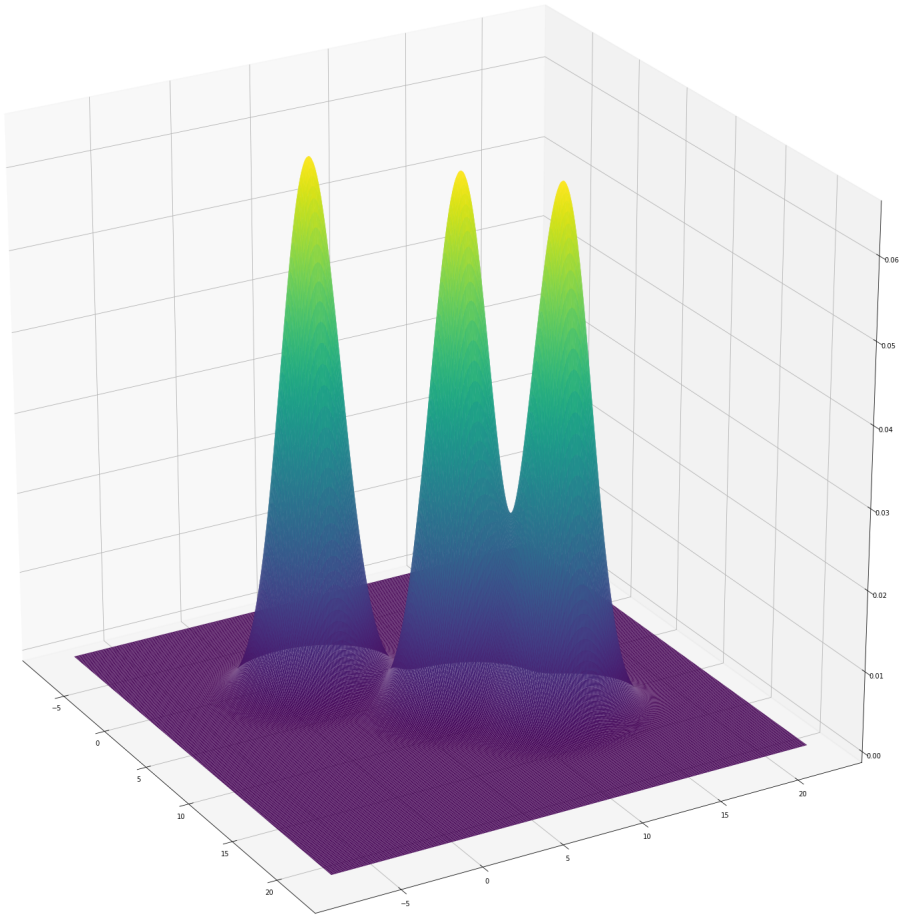
- In density estimation using rectangular window, the estimate is not smoother for small window size.
- In density estimation using gaussian window, the estimate is not able to distinguish different mixtures for larger window size.
- Also, if the mixture components have distributions close, then the non-parametric estimates fail.
- When compared with EM, the computation time is less for non-parametric density estimation.

[ ]:
```
K =3      # Number of mixture components in GMM
N = 1000 # Number of samples to be drawn from each mixture of mulitvariate␣
 ↪Gaussian to construct the mixture density
iterations=15
d=2     #dimenion of the data
mean=np.zeros((K,d))

covariance=[]


#"""Below code is to generate the mean vector and covariance matrix for the␣
 ↪mixture components"""
#mean[0,:]= 0
# covariance_temp= np.ones((d,d))*5
# covariance.append(covariance_temp) #np.linspace(3,6, num=K)

# samples_temp=np.random.multivariate_normal(mean[0,:],covariance_temp)
# samples=np.array(samples_temp)
mean=np.array([[2,2],[8,8],[12,12]])
covariance_temp=[[2,0],[0,3]]

for k in range(K):
#    mean[k,:]= np.array(mean[k,:])  # -20+40*np.array()#np.random.rand(1,d)
#     covariance_temp= dataset.make_spd_matrix(d)
```

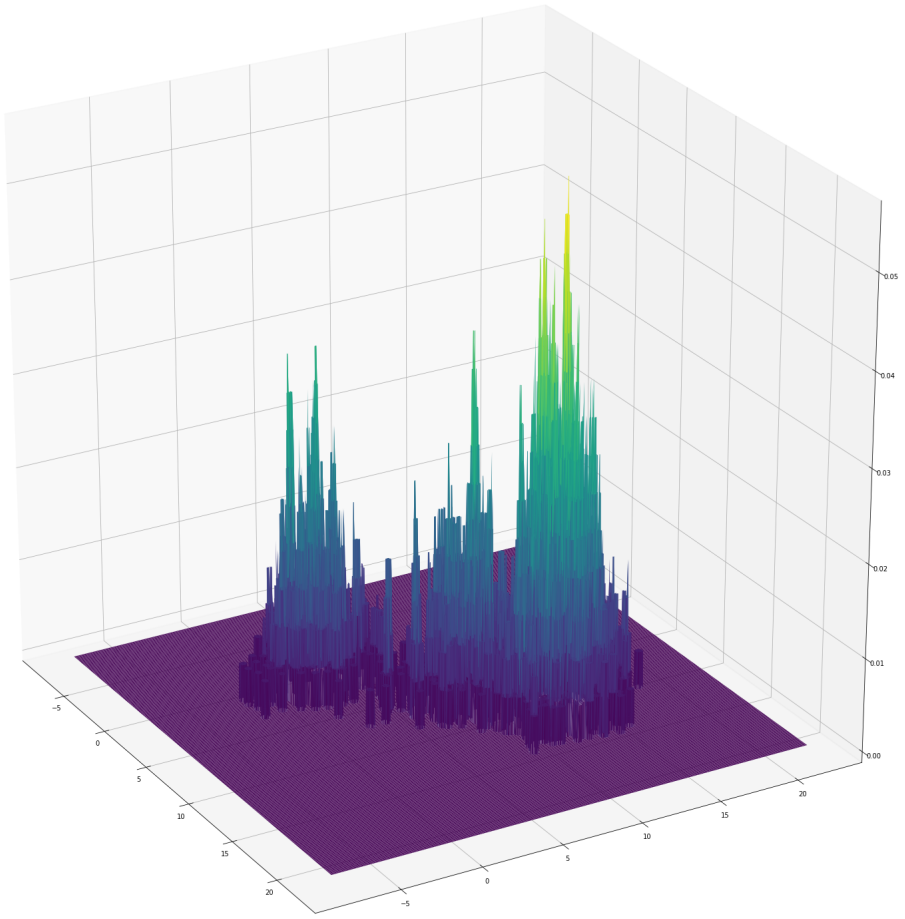
```
#     covariacne_temp=10+5*np.abs(covariance_temp)
    covariance.append(covariance_temp) #np.linspace(3,6, num=K)

true_covar=covariance
lambdas   = [0.3,0.2,0.5]

samples_lst=np.random.multivariate_normal(mean[1,:],covariance[1])
samples_lst=np.reshape(samples_lst,(1,d))
for n in range(N-1):
    mix_no=np.random.choice(3,p=lambdas)
    sample= np.random.multivariate_normal(mean[mix_no,:],covariance[mix_no])
    sample=np.reshape(sample,(1,d))
    samples_lst=np.append(samples_lst,sample,axis=0)

true_covar=covariance
# samples=np.array(samples)
# samples=np.reshape(samples,((int((K+1)*K*N*0.5),d)))

N=1000
samples_test=np.zeros((N,d))
x= np.linspace(np.min(mean)-10,np.max(mean)+10,N)
y= np.linspace(np.min(mean)-10,np.max(mean)+10,N)

samples_test=np.array([x,y])
print(np.shape(samples_test))
samples_test=samples_test.T
X, Y = np.meshgrid(x,y)
#print(samples_test.shape)
pos = np.empty(X.shape + (2,))
pos[:, :, 0] = X
pos[:, :, 1] = Y    #pos contains the points at which we estimate the density␣
 ↪for plotting

Z = multivariate_normal.pdf(pos, mean[0,:], covariance[0])
for k in range(K-1):
  Z =Z+ multivariate_normal.pdf(pos, mean[k+1,:], covariance[k+1])


# Create a surface plot and projected filled contour plot under it.
fig = plt.figure(figsize=(30,30))
ax = fig.gca(projection='3d')

ax.plot_surface(X, Y, Z, rstride=3, cstride=3, linewidth=1,cmap=cm.viridis)
ax.view_init(25,-30)
plt.show()
```

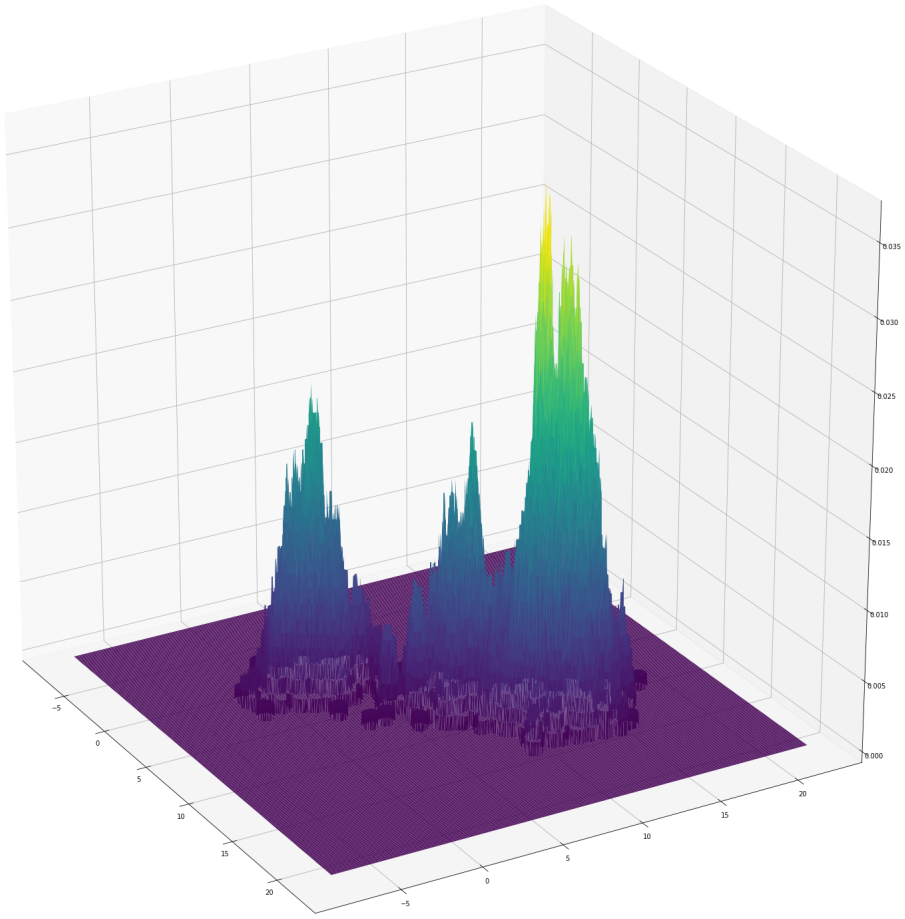
```
#dens_est=np.zeros((pos.shape[0]*pos.shape[1],1))
pos=np.reshape(pos,(pos.shape[0]*pos.shape[1],pos.shape[2]))
print(pos.shape)
h=[0.5,1,10]
print("Density estimation using rectangular window")
for width in h:
  dens_est=np.zeros((pos.shape[0],1))

  for i in range(len(pos)):
      dens_est[i]=rectangular(pos[i],samples_lst,width)

  size_n=int(np.sqrt(dens_est.shape[0]))

  dens_est=np.reshape(dens_est,(size_n,size_n))

  fig = plt.figure(figsize=(30,30))
  ax = fig.gca(projection='3d')

  ax.plot_surface(X, Y, dens_est, rstride=3, cstride=3, linewidth=1,cmap=cm.
 ↪viridis)
  ax.view_init(25,-30)
  plt.show()

print("Density estimation using Gaussian as Kernel window")
for width in h:
  dens_est=np.zeros((pos.shape[0],1))

  for i in range(len(pos)):
      dens_est[i]=gaussian(pos[i],samples_lst,width)

  dens_est=np.reshape(dens_est,(dens_est.shape[0],dens_est[0]))

  fig = plt.figure(figsize=(30,30))
  ax = fig.gca(projection='3d')

  ax.plot_surface(X, Y, dens_est, rstride=3, cstride=3, linewidth=1,cmap=cm.
 ↪viridis)
  ax.view_init(25,-30)
  plt.show()
```

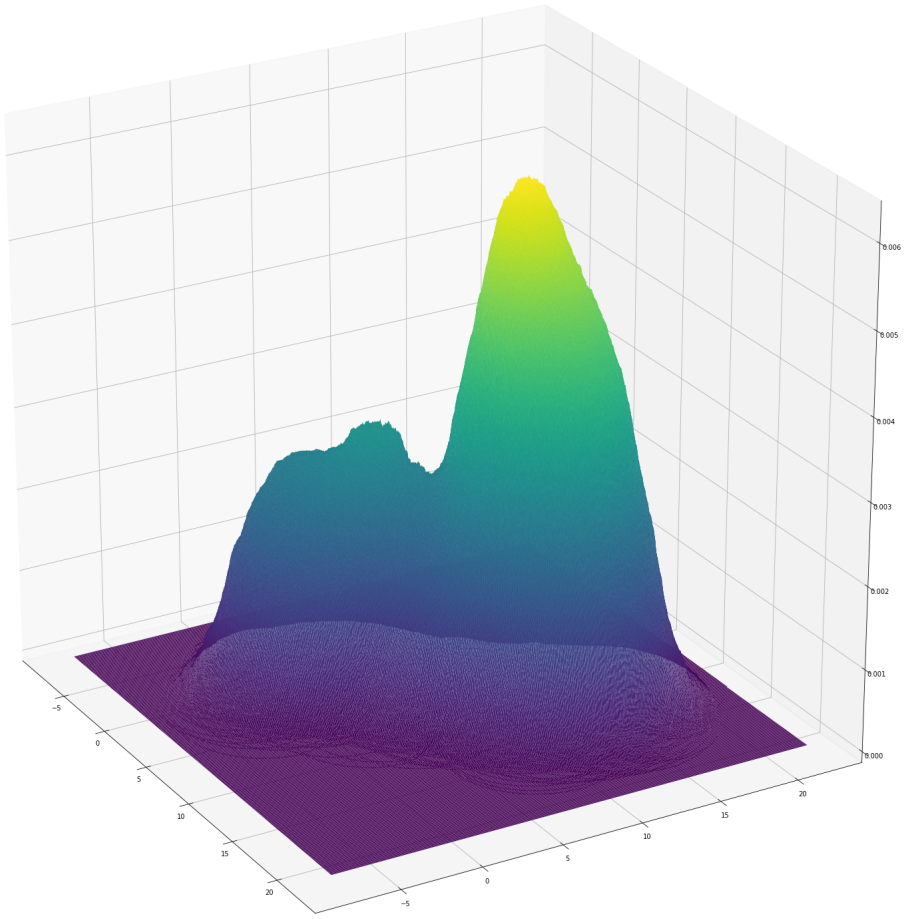
```
(2, 1000)
```



(1000000, 2)
Density estimation using rectangular window


0.05
0.04
0.03
0.02
0.01
0.00
−5
0
5
10
15
20
−5
0
5
10
15
20


0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
−5
0
5
10
15
20
−5
0
5
10
15
20



Density estimation using Gaussian as Kernel window

4.1.3 GMM Estimation using Kernel density estimation with kernel function as Gaussian

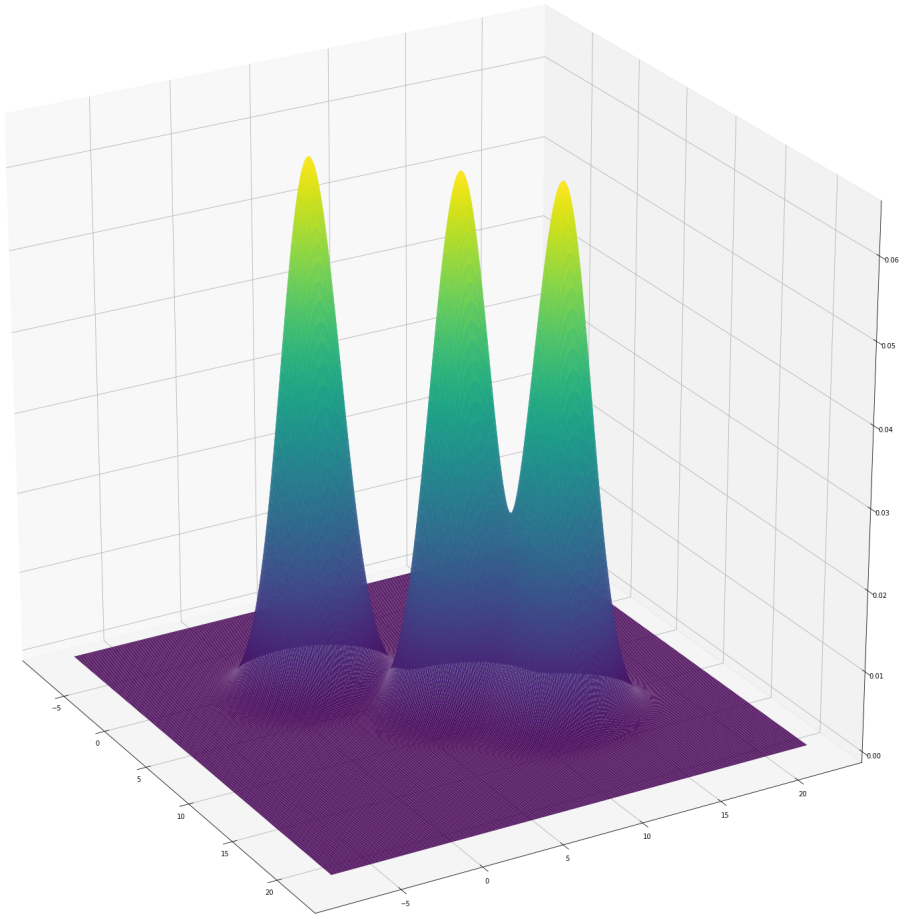
[ ]:
```
K =3      # Number of mixture components in GMM
N = 1000 # Number of samples to be drawn from each mixture of mulitvariate␣
 ↪Gaussian to construct the mixture density
iterations=15
d=2      #dimenion of the data
mean=np.zeros((K,d))

covariance=[]
```

```
#"""Below code is to generate the mean vector and covariance matrix for the␣
 ↪mixture components"""
#mean[0,:]= 0
# covariance_temp= np.ones((d,d))*5
# covariance.append(covariance_temp) #np.linspace(3,6, num=K)

# samples_temp=np.random.multivariate_normal(mean[0,:],covariance_temp)
# samples=np.array(samples_temp)
mean=np.array([[2,2],[8,8],[12,12]])
covariance_temp=[[2,0],[0,3]]

for k in range(K):
#    mean[k,:]= np.array(mean[k,:])  # -20+40*np.array()#np.random.rand(1,d)
#     covariance_temp= dataset.make_spd_matrix(d)
#     covariacne_temp=10+5*np.abs(covariance_temp)
    covariance.append(covariance_temp) #np.linspace(3,6, num=K)

true_covar=covariance
lambdas   = [0.3,0.2,0.5]

samples_lst=np.random.multivariate_normal(mean[1,:],covariance[1])
samples_lst=np.reshape(samples_lst,(1,d))
for n in range(N-1):
    mix_no=np.random.choice(3,p=lambdas)
    sample= np.random.multivariate_normal(mean[mix_no,:],covariance[mix_no])
    sample=np.reshape(sample,(1,d))
    samples_lst=np.append(samples_lst,sample,axis=0)

true_covar=covariance
# samples=np.array(samples)
# samples=np.reshape(samples,((int((K+1)*K*N*0.5),d)))

N=1000
samples_test=np.zeros((N,d))
x= np.linspace(np.min(mean)-10,np.max(mean)+10,N)
y= np.linspace(np.min(mean)-10,np.max(mean)+10,N)

samples_test=np.array([x,y])
print(np.shape(samples_test))
samples_test=samples_test.T
X, Y = np.meshgrid(x,y)
#print(samples_test.shape)
pos = np.empty(X.shape + (2,))
pos[:, :, 0] = X
pos[:, :, 1] = Y

Z = multivariate_normal.pdf(pos, mean[0,:], covariance[0])
```

```
for k in range(K-1):
  Z =Z+ multivariate_normal.pdf(pos, mean[k+1,:], covariance[k+1])


# Create a surface plot and projected filled contour plot under it.
fig = plt.figure(figsize=(30,30))
ax = fig.gca(projection='3d')

ax.plot_surface(X, Y, Z, rstride=3, cstride=3, linewidth=1,cmap=cm.viridis)
ax.view_init(25,-30)
plt.show()
# #cset = ax.contourf(X, Y, Z, zdir='z', offset=-0.15)
# plt.show()
# # fig = plt.figure(figsize=(10,10))
# # ax = fig.gca(projection='3d')

# #plt.hist(samples_test,bins=50)


# # for row in samples:
# #     ax.scatter(row[0], row[1], color="r", s=5, marker='^')

# # for row in samples_test:
# #     ax.scatter(row[0], row[1], color="k", s=5)


#dens_est=np.zeros((pos.shape[0]*pos.shape[1],1))
pos=np.reshape(pos,(pos.shape[0]*pos.shape[1],pos.shape[2]))
print(pos.shape)
h=[0.5,1,10]

print("Density estimation using Gaussian as Kernel window")
for width in h:
  dens_est=np.zeros((pos.shape[0],1))

  for i in range(len(pos)):
      dens_est[i]=gaussian(pos[i],samples_lst,width)

  size_n=int(np.sqrt(dens_est.shape[0]))

  dens_est=np.reshape(dens_est,(size_n,size_n))
  fig = plt.figure(figsize=(30,30))
  ax = fig.gca(projection='3d')

  ax.plot_surface(X, Y, dens_est, rstride=3, cstride=3, linewidth=1,cmap=cm.
 ↪viridis)
  ax.view_init(25,-30)
```

```
  plt.show()
#   #print(f"At {sample} the estimated density is {dens_est[i]}")
# print(dens_est)
# plt.show()
# plt.plot(samples_test,dens_est)
# #Create grid and multivariate normal
# # x = np.linspace(np.min(),np.max(samples[:,0]),1200)
# # y = np.linspace(np.min(samples[:,1]),np.max(),1200)
# X, Y = np.meshgrid(samples_test[:,0],samples[:,1])
# print(np.shape(X))
# fig=plt.figure(figsize=(15,15))
# #for k in range(1):
#     #rv = multivariate_normal(mean[k,:],covariance[k])
# #Make a 3D plot
# # %matplotlib qt
# ax = fig.gca(projection='3d')
# ax.plot_surface(X, Y, dens_est,cmap='viridis',linewidth=0)
# ax.set_xlabel('X axis')
# ax.set_ylabel('Y axis')
# ax.set_zlabel('Z axis')
# plt.show()
```

```
(2, 1000)
```



(1000000, 2)
Density estimation using Gaussian as Kernel window