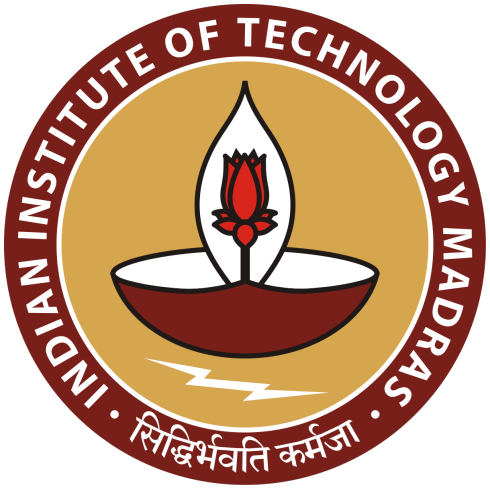
Assignment 1

CS5691: Pattern Recognition and Machine Learning



Nikhil Anand
BE20B022
Indian Institute of Technology Madras
February 2023

Contents

1 Question 1: PCA and Kernel PCA (MNIST dataset)

1.1 Part (i)

The entire code for this part is below:

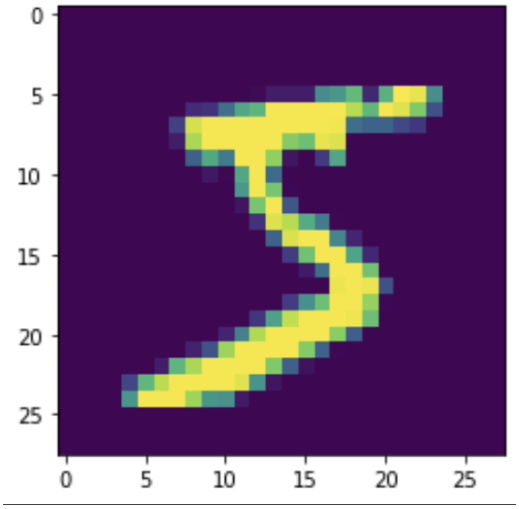
```
import numpy as np
import matplotlib.pyplot as plt
import random
import pandas as pd
from PIL import Image

def getEvalsEvecs(C):
  evals, evecs = np.linalg.eigh(C)

  evecs = np.transpose(evecs)

  lenn = len(evals)
  f_evals = []
  f_evecs = []

  for i in range(lenn):
    if(evals[lenn-1-i]>0):
      f_evals.append(evals[lenn-1-i])
      f_evecs.append(evecs[lenn-1-i])
    else:
      break

  return f_evals, f_evecs

df = pd.read_csv("mnist_train.csv")

n_samples = 3000
n_features = 784
data = np.array(df.iloc[:n_samples,:])

labels = np.zeros((n_samples,))
images = np.zeros((n_samples,n_features))
for i in range(len(data)):
    labels[i] = data[i][0]
    images[i] = data[i][1:]

print("Visualising the first image: ")
img = np.array(images[0].reshape(28,28))
plt.imshow(img)
plt.show(block=False)

X = images
mu = np.mean(X, axis=0)
centX = X - mu
```

```
C = (1/n_samples)*np.matmul(np.transpose(centX),centX)

f_evals, f_evecs = getEvalsEvecs(C)

print("Showing the top 5 principal components visualised: ")
for i in range(5):
  img = f_evecs[i].reshape(28,28)
  plt.imshow(img)
  plt.show(block=False)

print("The variance of the data along the top 10 principal components: ")
print(f_evals[:10])
```

Explanation of code:

We first define a function called 'getEvalsEvecs' which takes in C as a parameter (the Covariance matrix). The purpose of the function is to get a list of eigenvalues and eigenvectors for the given covariance matrix, and return a final list of eigenvectors and eigenvalues which satisfy the criterion that the eigenvalue corresponding to that term is greater than zero. This ensures all the eigenvalues that are zero are not counted in the final list, and only the principal components accounting for reasonably significant variance are counted. The code first reads the MNIST dataset csv file. Then, it is truncated upto 3000 samples, and the 'labels' and 'images' arrays are created storing the labels and the image array data respectively. Image data is a flattened array of shape (784,1) representing the image. The first image is visualised using 'plt.imshow', after reshaping it into a (28,28) image.

We can see how it looks below:



Now, we store the images array into a variable called X, resembling the notation for the dataset that was covered in class. Except in this case, the rows are different datapoints and the columns are features. It is an $(n \times d)$ matrix. We next center the data, by calculating the mean along the 0th axis and then subtracting it from X. Next, we calculate the covariance matrix on the centered dataset. It is calculated using the formula:

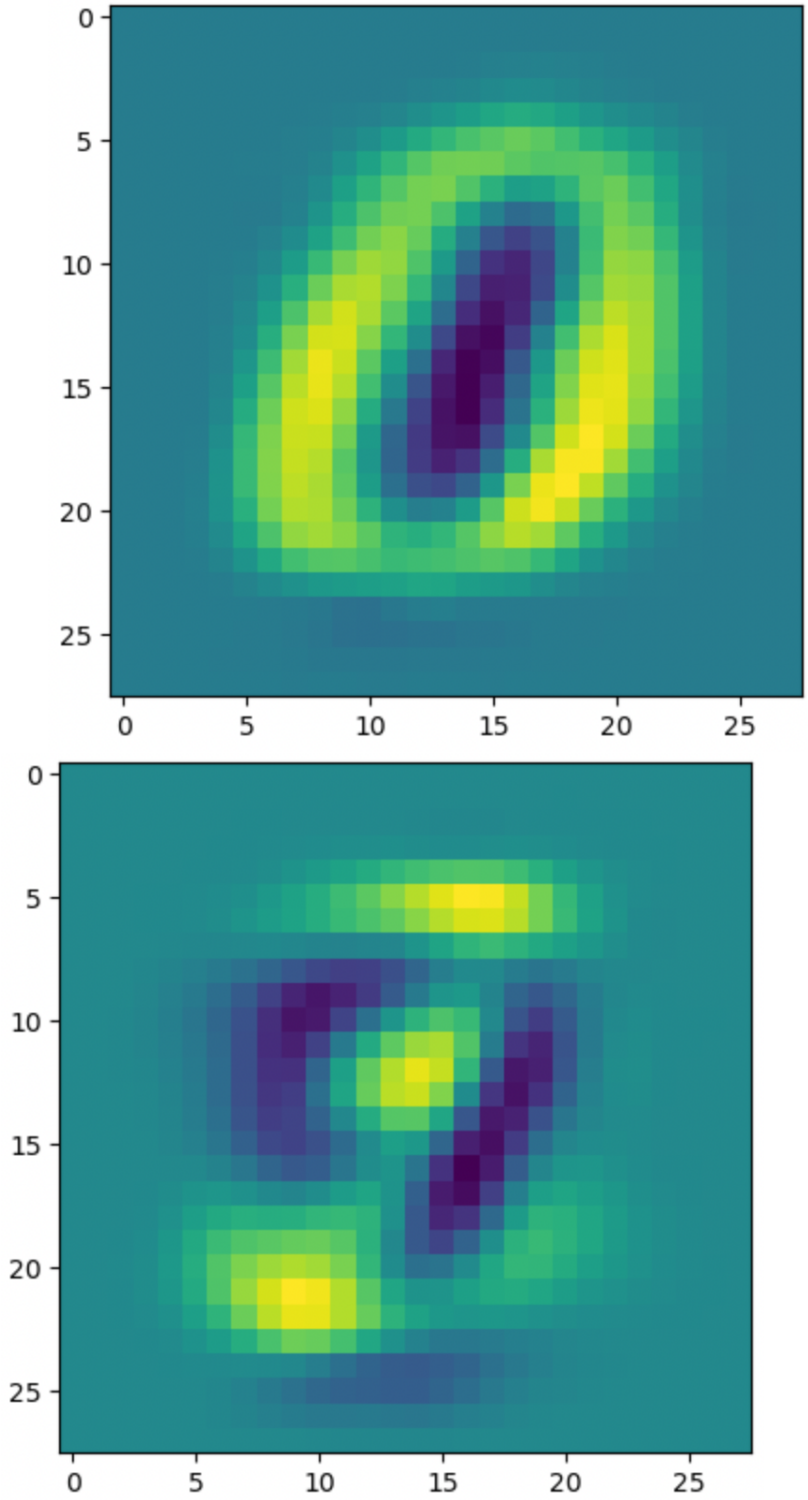
$$C = \frac{1}{n} X^T X$$

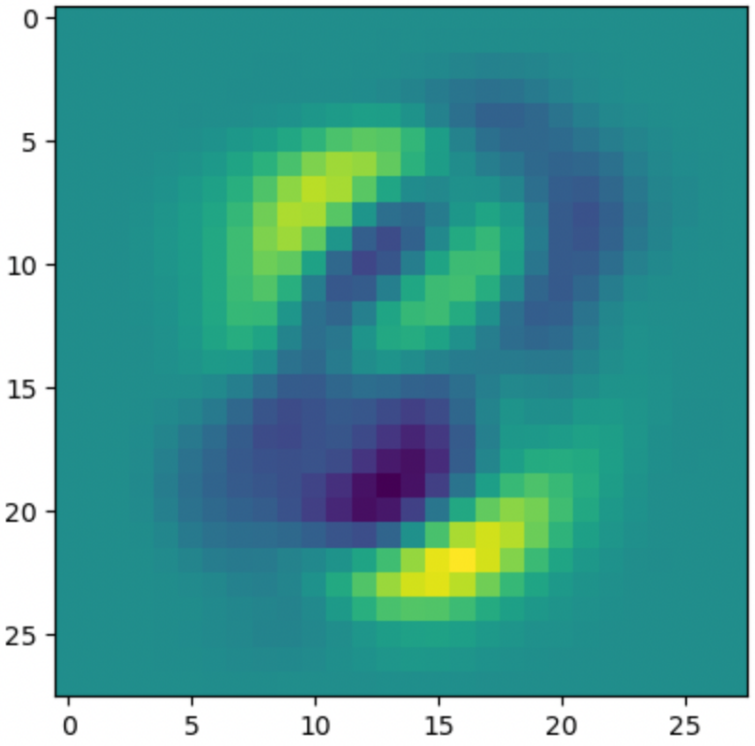
Where X is an $(n \times d)$ matrix. Clearly, C is $(d \times d)$.

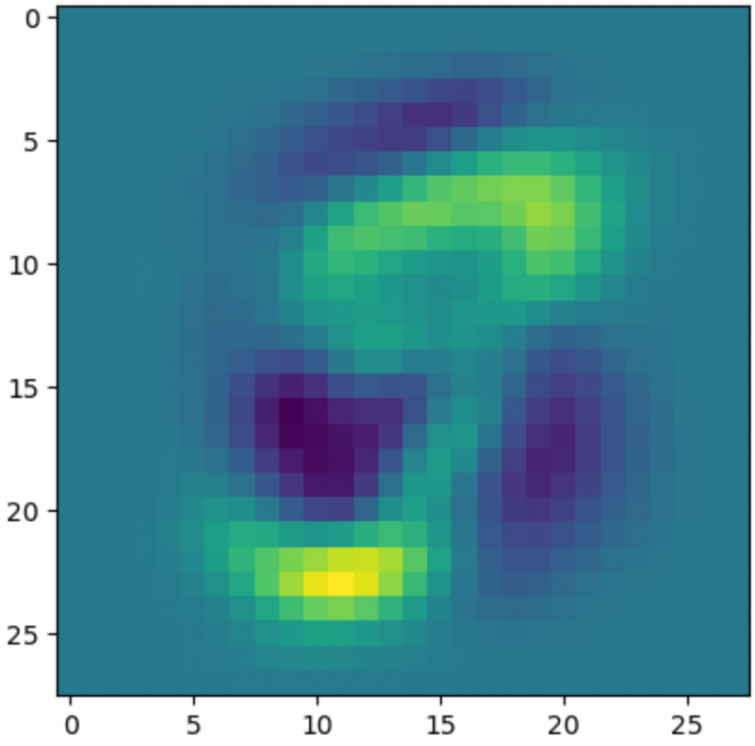
Now, we use the function 'getEvalsEvecs' and compute the eigenvalues and eigenvectors of the

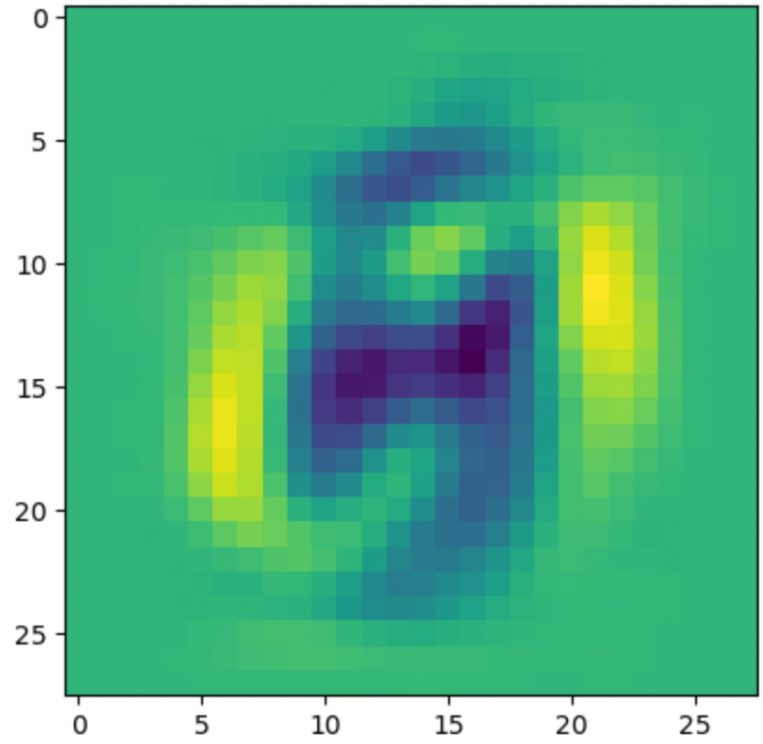
matrix C. Finally, the eigenvectors of C are themselves the top principal components of the dataset, so we have got our answer. We now visualise the top 5 principal components for the dataset. The output is below:

(Top row: PC1, PC2; Bottom row: PC3, PC4)




0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25



Now, to get the variance of the entire data along each of the principal components, we get:

$$var_{w_l}(X) = \frac{1}{n}\Sigma_1^n (x^T w_l)^2$$
$$= \frac{1}{n}\Sigma_1^n w_l^T x_i x_i^T w_l$$
$$= w_l^T C w_l = \lambda_l$$

Thus, the variance of the data along the top 10 principal components $w_l \forall l \in 1, 2, ..., 10$ is simply the top 10 eigenvalues of C. This has been printed in the code from the array in which we stored the eigenvalues. The output comes out to be:

[337127.28946546203, 249954.557913256, 219688.56723807723, 187483.52039853332, 165498.9239832566, 154108.93690834357, 114411.19904430327, 100628.2990793741, 97375.1431373693, 77795.57503974717]

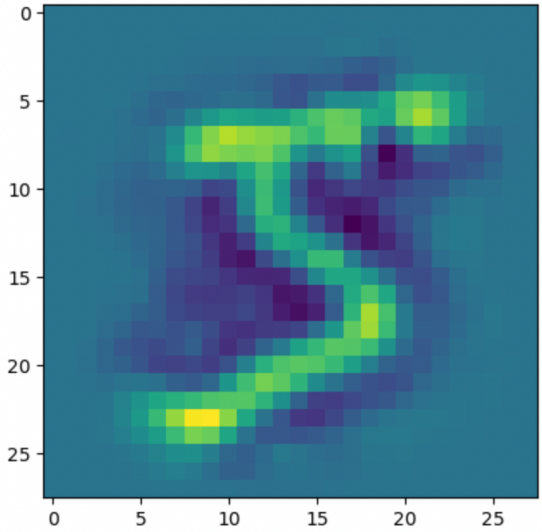
1.2 Part (ii)

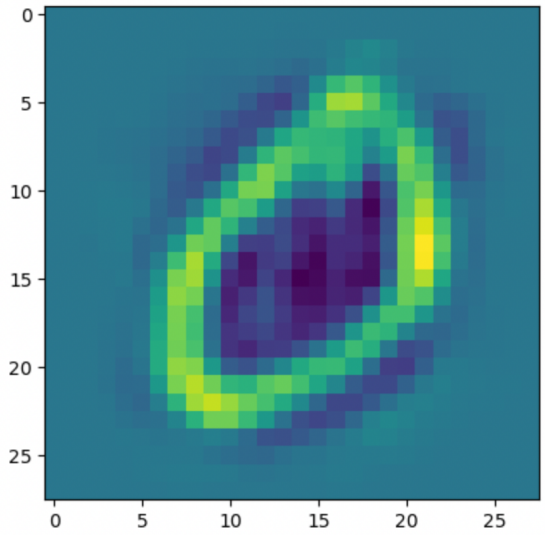
The entire code for this part is below:

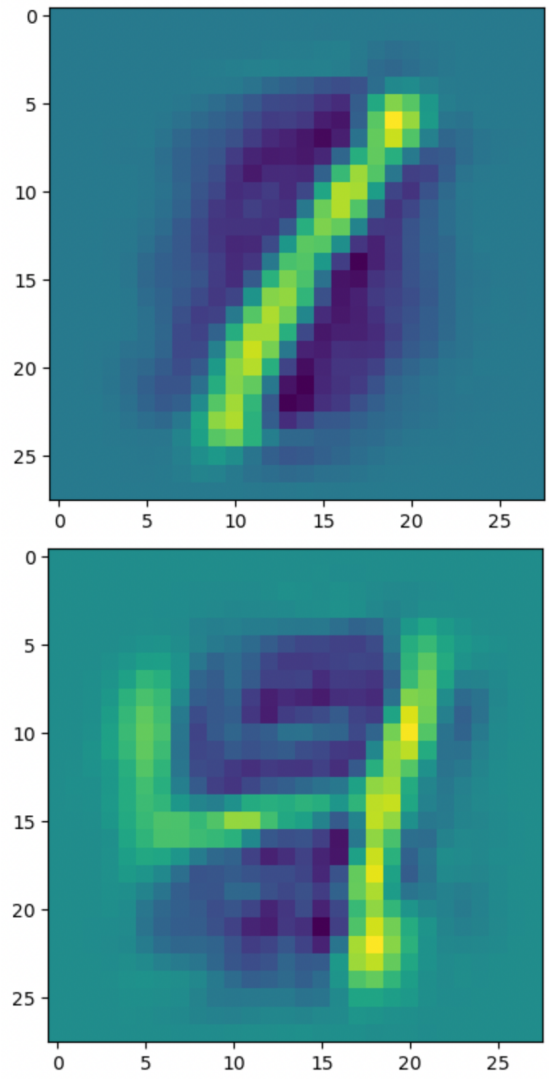
```
print("Showing the first 10 datapoints represented using only 70 principal com

d = 70
for i in range(5):
    xi_est = np.zeros((784,))
    for j in range(d):
        xi_proxy = np.dot(centX[i],f_evecs[j])
        xi_est += xi_proxy*f_evecs[j]
    xi_est = xi_est.reshape(28,28)
    plt.imshow(xi_est)
    plt.show(block=False)
```

Explanation of code:

We now have to reconstruct the data that we have using different dimensional representations. The code first assumes a value of d, representing the number of principal components that are used to approximate the data point. We loop over the first 5 data points, where we compute the projection of each data point along the first d eigenvectors and generate a vector approximation for the data point that way. We can vary d, and we see that as d increases, the approximation comes closer to the true value of the data point. At around $d = 70$, the numbers are quite legible, although not extremely sharp. I believe that this value would be reasonable to be able to classify the numbers in a downstream task, while also taking up only 70 dimensions per data point rather than 784 (saving up 90% of the space.







1.3 Part (iii)

The entire code for this part is below:

```
import math
def polyKernel(x,p):
  n = len(x)
  K = np.zeros((n,n))
  for i in range(n):
    for j in range(n):
      K[i][j] = math.pow((1+np.dot(x[i],x[j])),p)

  return K

def rbfKernel(x,s):
  n = len(x)
  K = np.zeros((n,n))
  for i in range(n):
    for j in range(n):
      K[i][j] = math.exp(-1*np.dot(x[i]-x[j],x[i]-x[j])/(2*s**2))
```

```
  return K

def plotTop2Components(x,labels,K):
  k_vals, k_vecs = getEvalsEvecs(K)
  print("Done getting evals, evecs")
  b1 = k_vecs[0]
  b2 = k_vecs[1]
  a1 = b1/math.sqrt(len(K)*k_vals[0])
  a2 = b2/math.sqrt(len(K)*k_vals[1])
  w1 = np.matmul(np.transpose(x),a1)
  w2 = np.matmul(np.transpose(x),a2)

  x_sc = np.zeros((len(K),))
  y_sc = np.zeros((len(K),))

  for i in range(len(K)):
    x_sc[i] = np.dot(x[i],w1)
    y_sc[i] = np.dot(x[i],w2)

  plt.rcParams['axes.facecolor'] = '#272b30'
  plt.rcParams['figure.facecolor'] = '#272b30'
  plt.rcParams['savefig.facecolor'] = '#272b30'
  plt.rcParams['xtick.color'] = 'white'
  plt.rcParams['ytick.color'] = 'white'
  plt.rcParams['axes.labelcolor'] = 'white'
  plt.rcParams['text.color'] = 'white'

  cmap = plt.cm.get_cmap('RdPu', 10)

  print("Populated x_sc and y_sc, getting ready to plot...")

  plt.scatter(x_sc,y_sc,c=labels,cmap=cmap)
  plt.xlabel("Component along w1")
  plt.ylabel("Component along w2")
  plt.title("Scatter plot")

  plt.show(block=False)

  for j in range(3):
    img = x_sc[j]*w1 + y_sc[j]*w2
    img = img.reshape(28,28)
    plt.imshow(img)
    plt.show(block=False)

for i in range(2,5):
    print("Showing scatterplot of top two principal components for polynomial
    plotTop2Components(centX,labels,polyKernel(centX,i))

for i in range(1,11):
```

```
print("Showing scatterplot of top two principal components for RBF kernel
plotTop2Components(centX, labels, rbfKernel(centX, 0.1*i))
```

Explanation of code:

For this part, I first define two functions (polyKernel and rbfKernel). Both these functions take the dataset and a parameter p as arguments, and they return the kernel matrix corresponding to that kernel, dataset and parameter.

Next, I create a function called 'plotTop2Components' which takes the dataset x, the labels associated with each datapoint, and the kernel matrix K (for any kernel) as arguments. The function then calculates the eigenvalues and eigenvectors of the kernel matrix, gets the top two eigenvectors β_1 and β_2 (b1 and b2 in the code), and calculates α_1 and α_2 (a1 and a2) according to the formula:

$$\alpha_i := \frac{\beta_i}{\sqrt{n\lambda_i}}$$

Finally, w_1 and w_2 are calculated according to:

$$w_i = X\alpha_i$$

The scatterplot arrays are then populated with the projections of x_i onto w_1 and w_2.

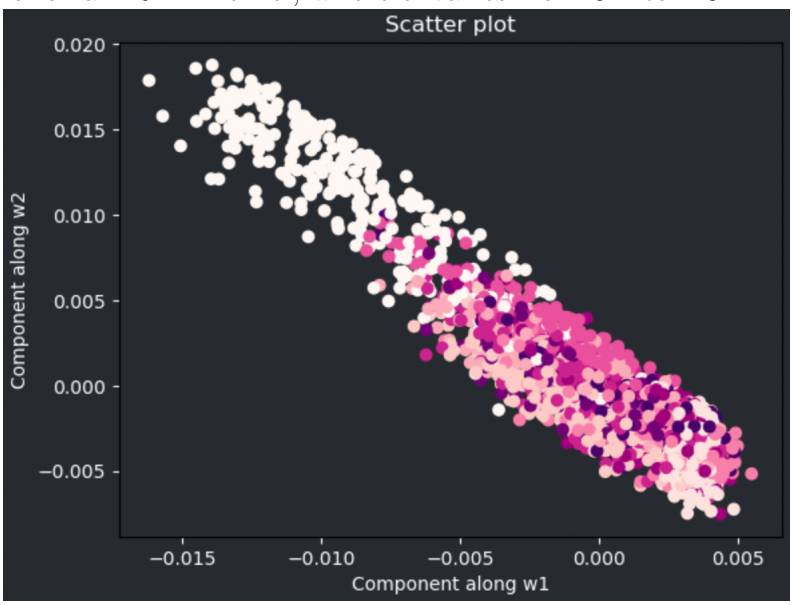
$$x_{sc,i} = x_i^T w_1$$
$$y_{sc,i} = x_i^T w_2$$

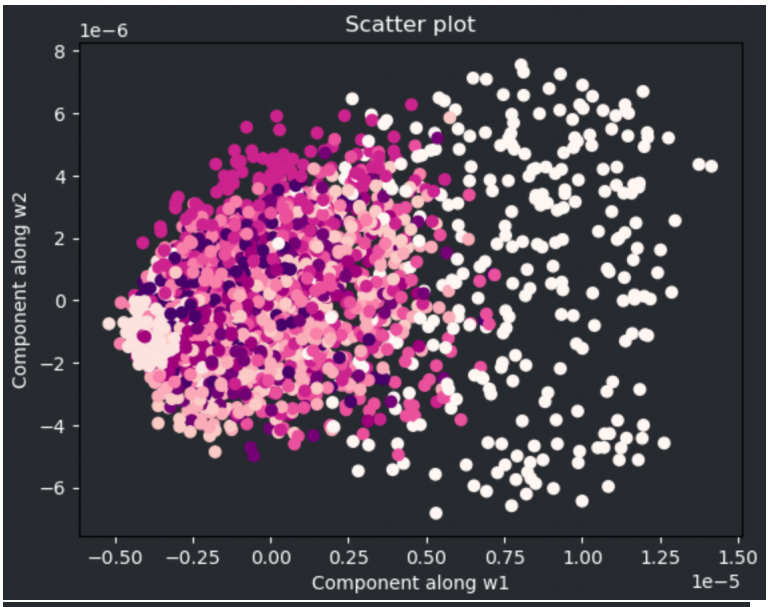
Next, we set some parameters of the scatterplot to make the background black and the axes white, to be able to visualise the colours of the datapoints better.

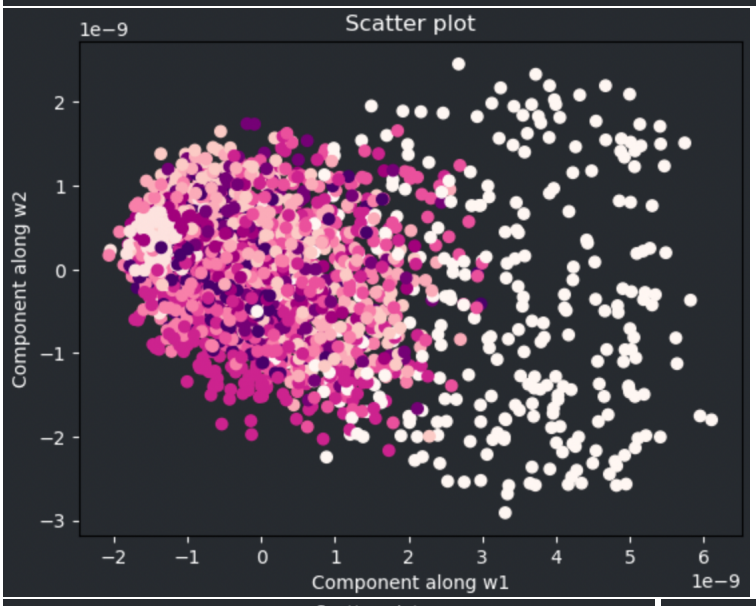
Also, we provide a colour map which is a gradient of a light pink to a dark red from the value of 0 to 9. This means that datapoints labelled as 0 will appear as light pink while those labelled as 9 will appear as dark red. Other values take up intermediate colours.

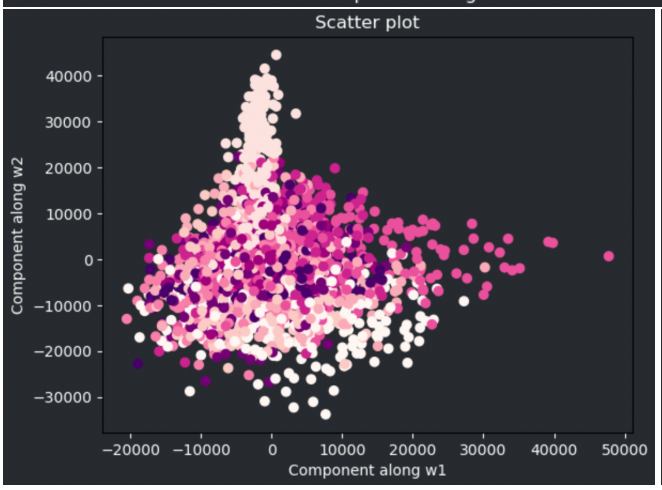
Finally, $x_{sc,i}$ and $y_{sc,i}$ are plotted on the scatterplot, and the projections of the first 3 datapoints along the first two principal components are found and visualised as images.

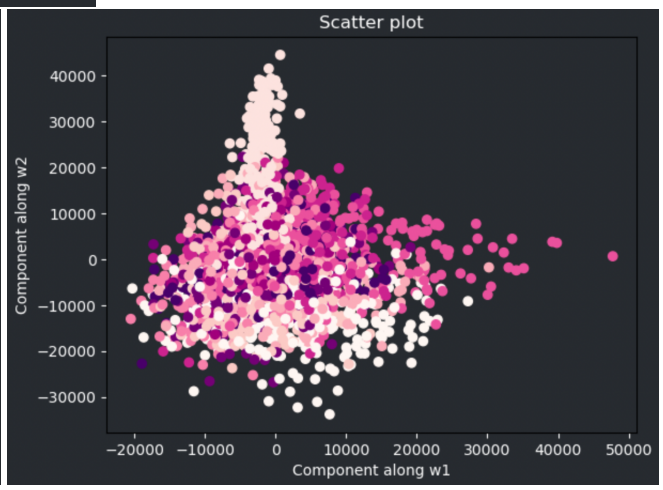
Now we simply have to run the 'plotTop2Components' function multiple times to visualise the scatterplots of the top 2 components in different scenarios. We first plot the top 2 components of the dataset after passing it through a polynomial kernel of degree 2, 3, 4, and 5. We then do the same for an RBF Kernel, where σ varies from 0.1 to 1.0. All the results are shown below:

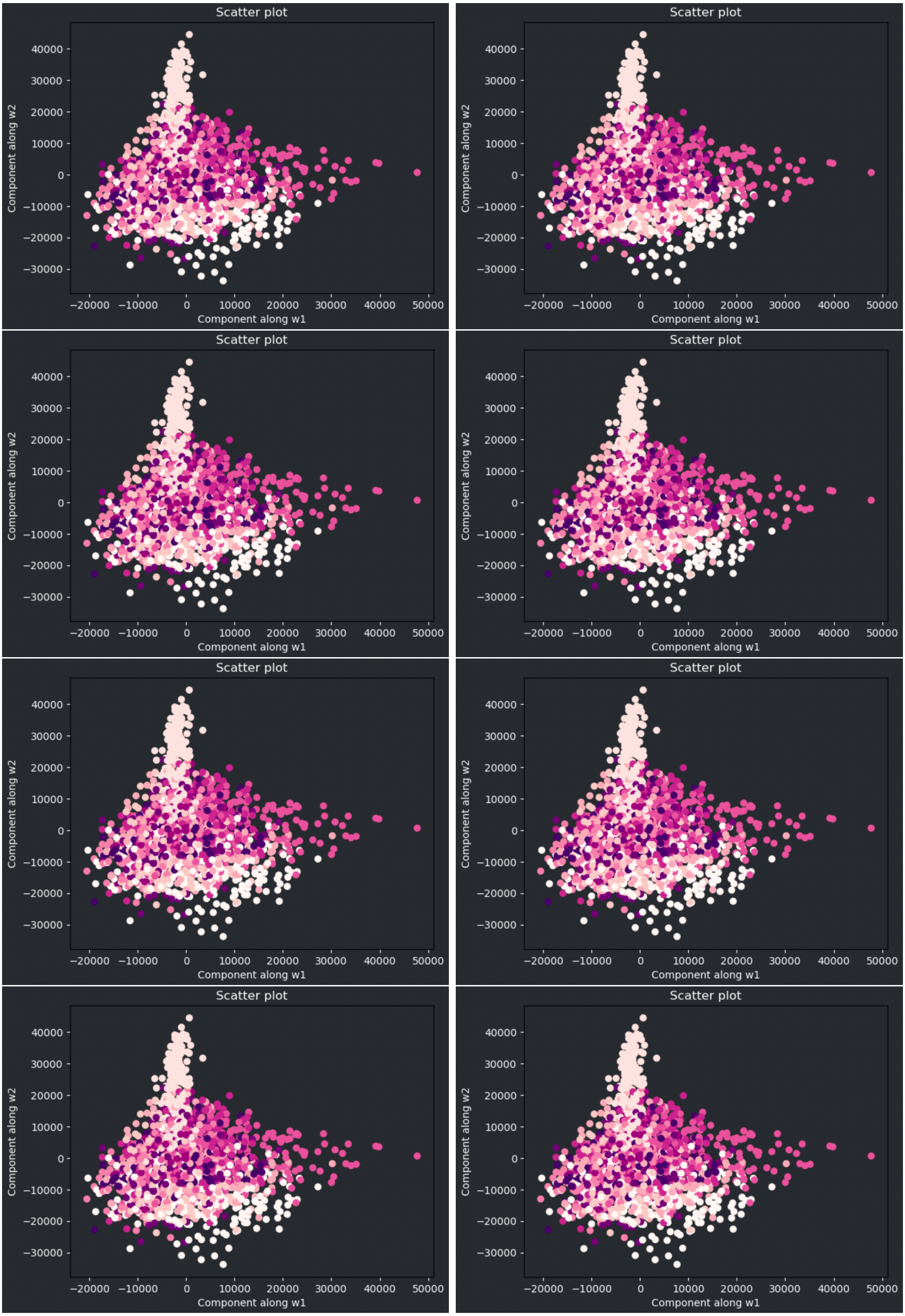



Scatter plot
1e−6
8
6
4
2
0
−2
−4
−6
Component along w2
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
1.25
1.50
Component along w1
1e−5


Scatter plot
1e−9
2
1
0
−1
−2
−3
Component along w2
−2
−1
0
1
2
3
4
5
6
Component along w1
1e−9


Scatter plot
40000
30000
20000
10000
0
−10000
−20000
−30000
Component along w2
−20000
−10000
0
10000
20000
30000
40000
50000
Component along w1


Scatter plot
40000
30000
20000
10000
0
−10000
−20000
−30000
Component along w2
−20000
−10000
0
10000
20000
30000
40000
50000
Component along w1


Scatter plot
Component along w2
Component along w1
40000
30000
20000
10000
0
−10000
−20000
−30000
−20000
−10000
0
10000
20000
30000
40000
50000

1.4 Part (iv)

How do we determine which kernel is best suited for this dataset? We would probably have to look at the two-component projection of the dataset in various kernels, and see if the images are distinguishable enough to be able to facilitate the downstream task of classification.

One way of doing this is to observe the scatterplots and analyse whether the points are forming clusters or are easy to classify into 10 classes. This is why we coloured the points according to the labels that were assigned to each datapoint.

If we observe the scatterplots for the polynomial kernels, for most of them, images corresponding to the number zero tend to have a higher value of a certain principal component, due to which they appear quite separated from the other numbers. Thus, the polynomial kernel is quite accurate in identifying zeros with only 2 principal components.

However, the RBF kernel also has some regions where a majority of points are a specific number. In all of the RBF Kernel scatterplots, the number 5 has a much higher value along the second principal component direction, and the number 1 has a higher value along the first principal component direction. Thus, these two numbers are easily identifiable compared to other numbers.

Thus, the RBF Kernel can be considered better, because a larger number of classes are easily distinguishable in this case.

2 Question 2: K-Means and Kernel K-Means

2.1 Part (i)

Code:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random
import matplotlib.colors
import math
import scipy

def plotPoints(x,y,c):
  plt.scatter(x,y,c=c)
  plt.xlabel('X')
  plt.ylabel('Y')
  plt.show(block=False)

def computeMeans(x,y,z):
  sum0 = np.zeros((2,))
  n0 = 0
  sum1 = np.zeros((2,))
  n1 = 0

  for i in range(n):
    if(z[i]==0):
      sum0+=np.array([x[i],y[i]])
```

```
        n0+=1
      elif(z[i]==1):
        sum1+=np.array([x[i],y[i]])
        n1+=1

  return (sum0/n0,sum1/n1)

def plotPointsMeans(x,y,m0,m1,c):
  plt.scatter(x,y,c=c,cmap=matplotlib.colors.ListedColormap(['red', 'blue']))
  plt.scatter(m0[0], m0[1], marker='+', s=200, color='red')
  plt.scatter(m1[0], m1[1], marker='+', s=200, color='blue')
  plt.xlabel('X')
  plt.ylabel('Y')
  plt.show(block=False)

def reassignClusters(x,y,m0,m1):
  z = np.zeros((n,))
  for i in range(n):
    pt = np.array([x[i],y[i]])
    dist0 = np.dot(pt-m0,pt-m0)
    dist1 = np.dot(pt-m1,pt-m1)
    if(dist0>dist1):
      z[i] = 1
    else:
      z[i] = 0

  return z

def computeObjective(x,y,z,mean0,mean1):
  obj = 0
  for i in range(n):
    pt = np.array([x[i],y[i]])
    if(z[i]==0):
      obj+=np.dot(pt-mean0,pt-mean0)
    elif(z[i]==1):
      obj+=np.dot(pt-mean1,pt-mean1)

  return obj



df = pd.read_csv("cm_dataset_2.csv",names=['x','y'])
x = df['x'].to_numpy()
y = df['y'].to_numpy()
n = 1000

print("Showing a scatterplot of the given data now: ")
plt.scatter(x,y)
plt.xlabel('X')
plt.ylabel('Y')
```

```
plt.show()

for k in range(5):
    print("RANDOM INITIALISATION NUMBER ",k+1)

    z = np.zeros((n,))
    objlist = []

    for i in range(n):
        z[i] = random.choice([0,1])

    print("Showing scatterplot of data coloured according to the initial random
    plotPoints(x,y,z)

    mean0 = np.array(computeMeans(x,y,z)[0])
    mean1 = np.array(computeMeans(x,y,z)[1])

    print("Plotting the points on a scatterplot with the calculated means shown
    plotPointsMeans(x,y,mean0,mean1,z)

    z_new = reassignClusters(x,y,mean0,mean1)
    objlist.append(computeObjective(x,y,z_new,mean0,mean1))

    print("After the first iteration of reassigning, the new plot is shown: ")
    plotPointsMeans(x,y,mean0,mean1,z_new)

    print("Now we iterate and after we are done, we plot the final clustering

    iter = 1
    while not(np.array_equal(z_new,z)):
        z = z_new

        mean0 = np.array(computeMeans(x,y,z)[0])
        mean1 = np.array(computeMeans(x,y,z)[1])
        z_new = reassignClusters(x,y,mean0,mean1)
        iter+=1

        objlist.append(computeObjective(x,y,z_new,mean0,mean1))

    plotPointsMeans(x,y,mean0,mean1,z)
    print(iter)

    print("Now we show the variation of the objective function with iterations
    plt.figure()
    plt.plot(objlist)
    plt.xlabel('$Iterations')
    plt.ylabel('Objective')
    plt.title('Objective vs iterations')

    plt.show()
```

Explanation of code:

We first import the necessary libraries, and we define a few functions.

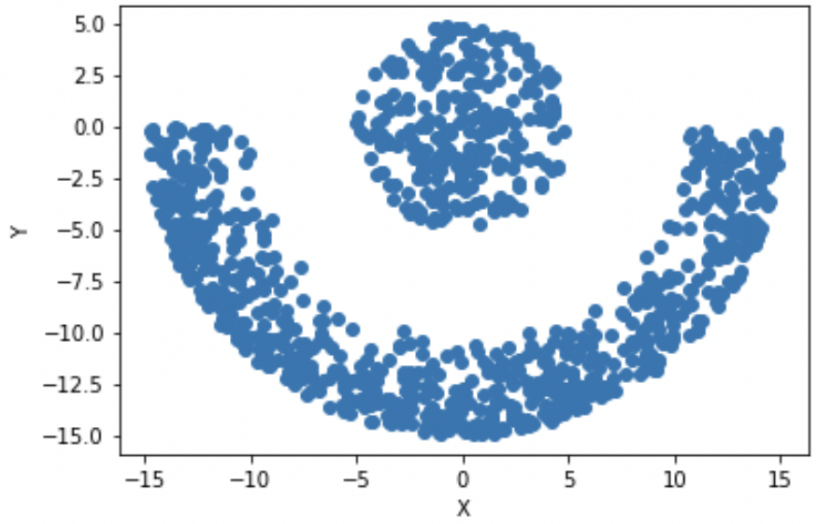
The plotPoints function takes in parameters x, y and c and plots the points given by (x,y) using the colour labels given by c. The computeMeans function takes in x, y and cluster assignments z, and computes and returns the centroids of the clusters. Next, the plotPointsMeans function takes in the points given by (x,y) as well as the means and colour labels, and plots all the plots along with their means.

The reassignClusters function computes a new reassignment of clusters to the datapoints according to the distance minimisation step and returns the array z.

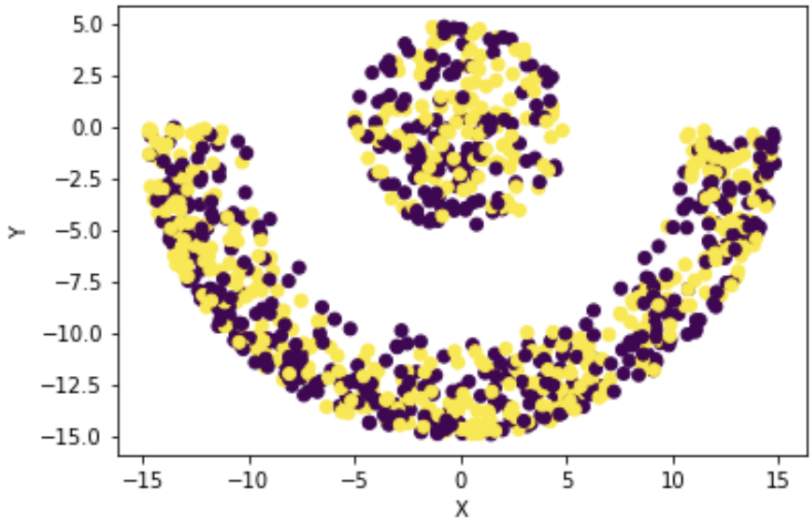
The computeObjective function computes the objective given the assignment and the means. The objective is given by:

$$F(z_1, z_2, ..., z_n) = \Sigma_1^n |x_i - \mu_{z_i}|^2$$

In the code, we first read the data from the file give, and we create two numpy arrays for the x and y values. We then plot the data on a scatterplot. The plot shows up like this:

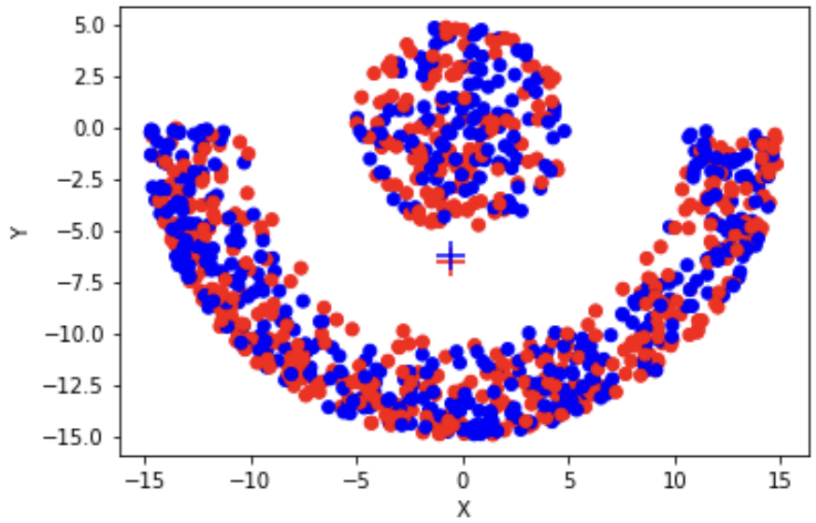


Next, we start a loop which runs 5 times, corresponding to the 5 initialisations as asked in the question. In the loop, we initialise the assignment matrix and give each element in it random values (0 or 1). Then, we plot the points:

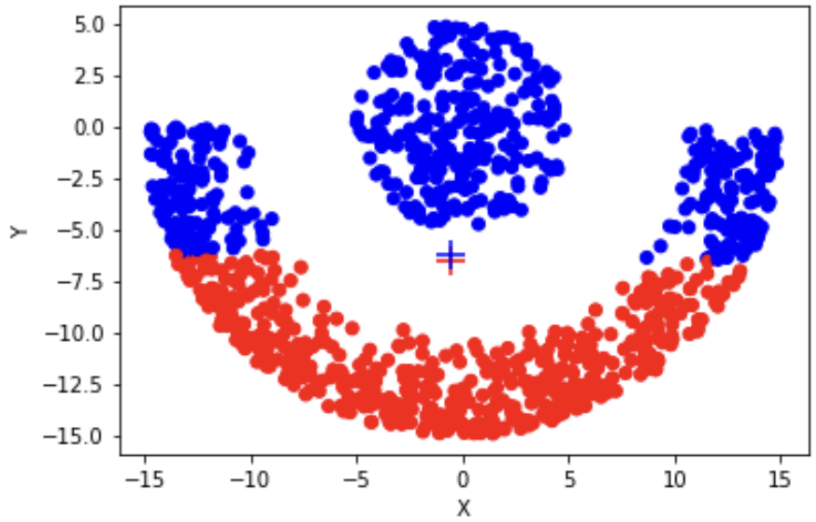


Now we plot the points along with their means, after computing the means.

The output is below, where the means are represented by 'plus' signs:

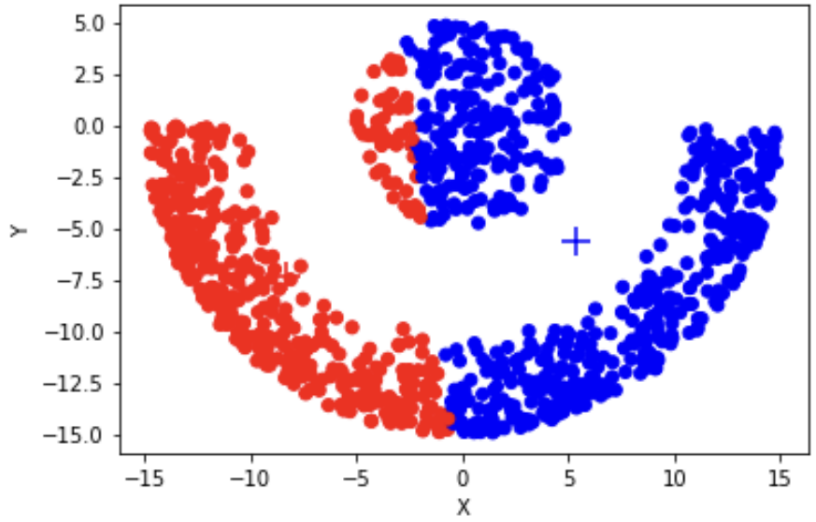


Now we reassign clusters, add the objective to the objlist, and plot the result.
The newly reassigned clusters are visualised:

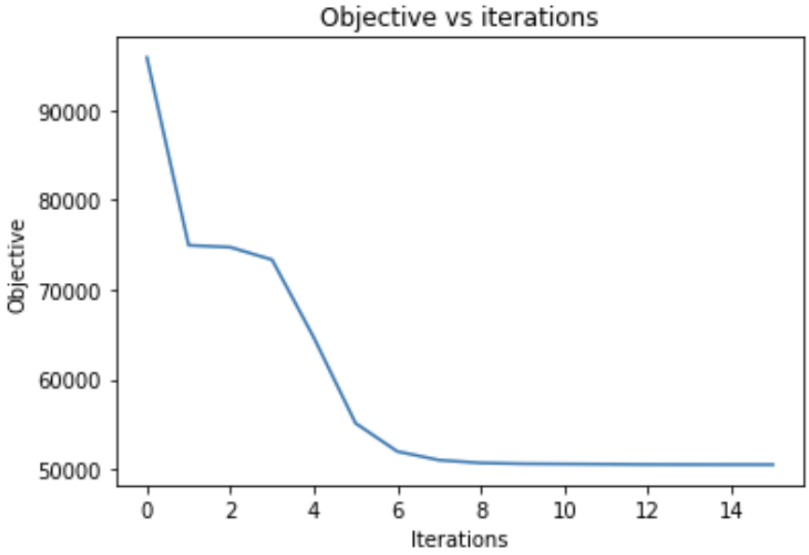


Now we have completed one iteration of K-means. We continue to the other iterations now, and perform iterations until convergence through the loop defined below. Note the iter is initialised to 1, as 1 iteration is done.

The final clustering is shown.



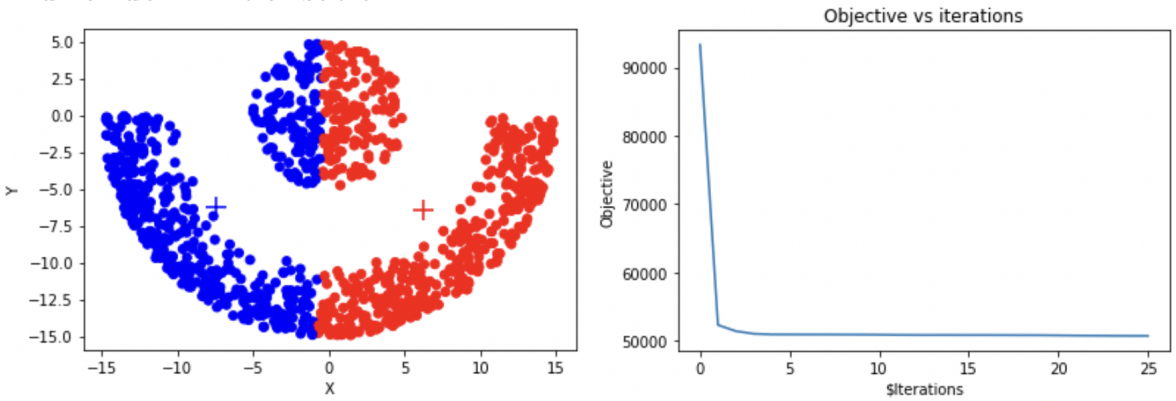
We then plot the objective functions/error functions vs number of iterations.



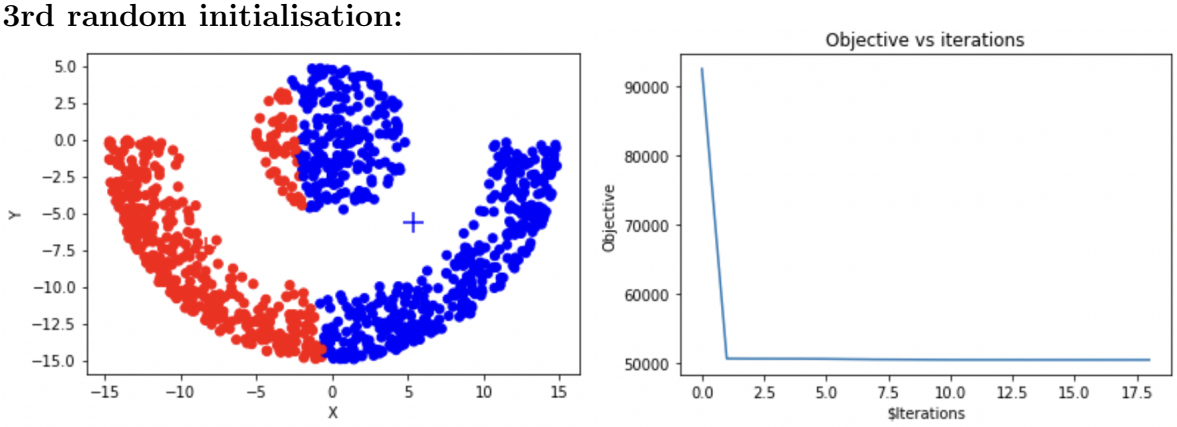
Now we repeat the above procedure for 4 more random initialisations and show the final clustering result, as well as the objective function graph.

Results are shown below:

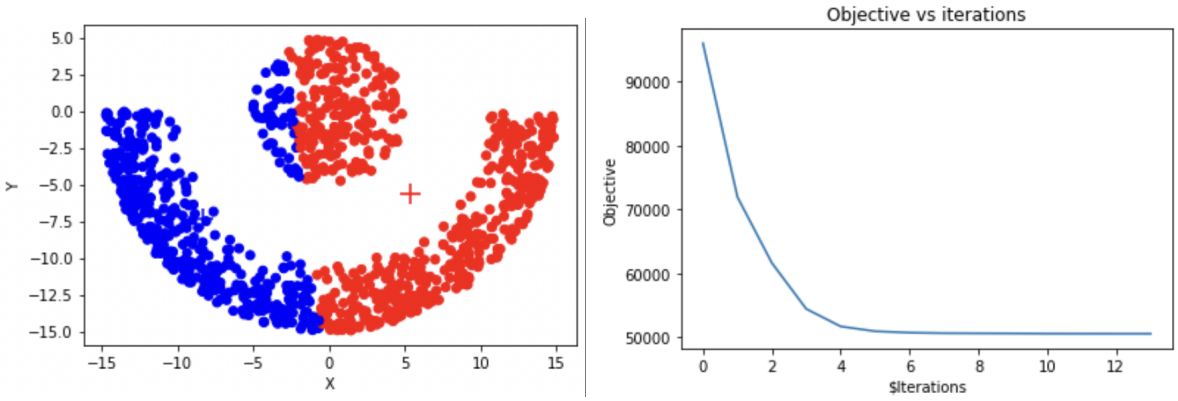
2nd random initialisation:



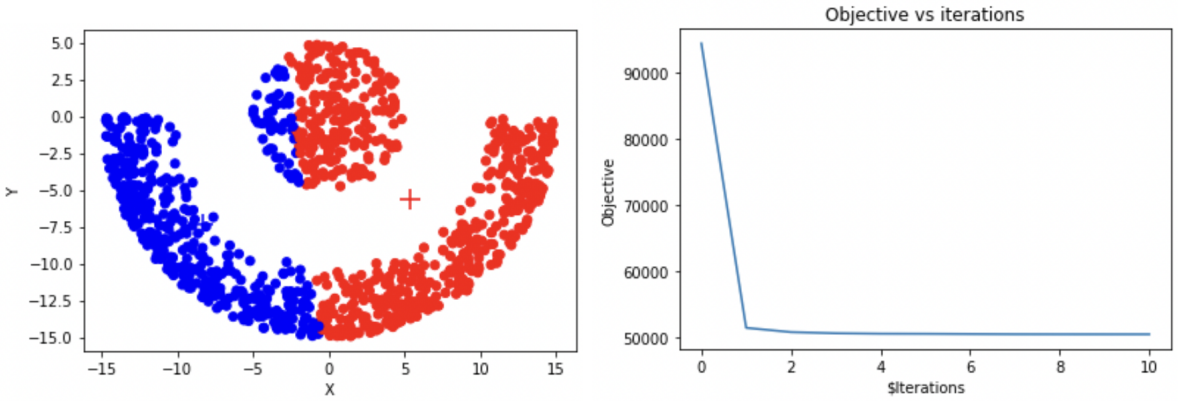
3rd random initialisation:



4th random initialisation:



5th random initialisation:



2.2 Part (ii)

Code:

```
from scipy.spatial import Voronoi, voronoi_plot_2d

def computeMeansK(x,y,z,k):
  sum = np.zeros((k,2))
  n = np.zeros((k,))

  for j in range(k):
    nj=0
    for i in range(len(x)):
      if(z[i]==j):
        sum[j]+=np.array([x[i],y[i]])
        nj+=1
    if(nj==0):
      pass
    else:
      sum[j]=sum[j]/nj

  means = sum
  # print(means)
  return means

def plotPointsMeansK(x,y,means,c,k):
  plt.scatter(x,y,c=c)
```

```
  for i in range(k):
    plt.scatter(means[i][0], means[i][1], marker='+', s=200, color='red')
  plt.show(block=False)

def plotPointsMeansVoronoiK(x,y,means,c,k):
  plt.scatter(x,y,c=c)
  for i in range(k):
    plt.scatter(means[i][0], means[i][1], marker='+', s=200, color='red')

  if(k==2):
    pass
  else:
    vor = Voronoi(means)
    fig = voronoi_plot_2d(vor)
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.title("Voronoi partition of clusters")

  plt.show(block=False)

def reassignClustersK(x,y,means,k):
  n = len(x)
  z = np.zeros((n,))
  for i in range(n):
    pt = np.array([x[i],y[i]])
    dist = np.zeros((k,))
    for j in range(k):
      dist[j] = np.dot(pt-means[j],pt-means[j])

    z[i] = np.argmin(dist)

  # print(z)
  return z

def computeObjectiveK(x,y,z,means,k):
  obj = 0
  n = len(x)
  for i in range(n):
    pt = np.array([x[i],y[i]])
    for j in range(k):
      if(z[i]==j):
        obj+=np.dot(pt-means[j],pt-means[j])

  return obj

def KMeans(x,y,z0,k):
  n = len(x)
  objlist = []

  print("PLOTTING INITIALISED POINTS")
```

```
plotPoints(x,y,z0)

print("COMPUTING MEANS")

means = np.array(computeMeansK(x,y,z0,k))

print("PLOTTING MEANS")

plotPointsMeansK(x,y,means,z0,k)

print("REASSIGNING CLUSTERS")

z_new = reassignClustersK(x,y,means,k)
objlist.append(computeObjectiveK(x,y,z_new,means,k))

print("PLOTTING NEW POINTS")

plotPointsMeansK(x,y,means,z_new,k)

print("STARTING ITERATIVE PROCEDURE")

iter = 1
z = z0

while not(np.array_equal(z_new,z)):
  z = z_new

  print("COMPUTING MEANS")

  means = np.array(computeMeansK(x,y,z,k))

  print("REASSIGNING CLUSTERS")

  z_new = reassignClustersK(x,y,means,k)
  iter+=1

  objlist.append(computeObjectiveK(x,y,z_new,means,k))

print("PLOTTING FINAL MEANS")

plotPointsMeansVoronoiK(x,y,means,z_new,k)
print(iter)

plt.figure()
plt.plot(objlist)
plt.xlabel('Iterations')
plt.ylabel('Objective')
plt.title('Objective vs iterations')
```

```
    plt.show(block=False)

    return z_new

def randInit(k,n):
    arr = [0,1,2,3,4,5]
    z = np.zeros((n,))
    for i in range(n):
        zi = random.choice(arr[:k])
        z[i]=zi
    return z

print("Running K-means with a random initialisation for K = 2 clusters")
KMeans(x,y,randInit(2,len(x)),2)

print("Running K-means with a random initialisation for K = 3 clusters")
KMeans(x,y,randInit(3,len(x)),3)

print("Running K-means with a random initialisation for K = 4 clusters")
KMeans(x,y,randInit(4,len(x)),4)

print("Running K-means with a random initialisation for K = 5 clusters")
KMeans(x,y,randInit(5,len(x)),5)
```

Explanation of code:

First we redefine all the functions computeMeansK, plotPointsMeansK, reassignClustersK, computeObjectiveK which are the same as the previous functions except they now incorporate a parameter k, for the number of clusters.

Now I define a KMeans function to encapsulate the entire K-means algorithm process that we discussed in the previous part. The difference here is that we can also take a parameter 'k' to define how many clusters we want to cluster the points into. The function takes the points, the number of clusters, and the initialisation of the assignment to clusters (z0).

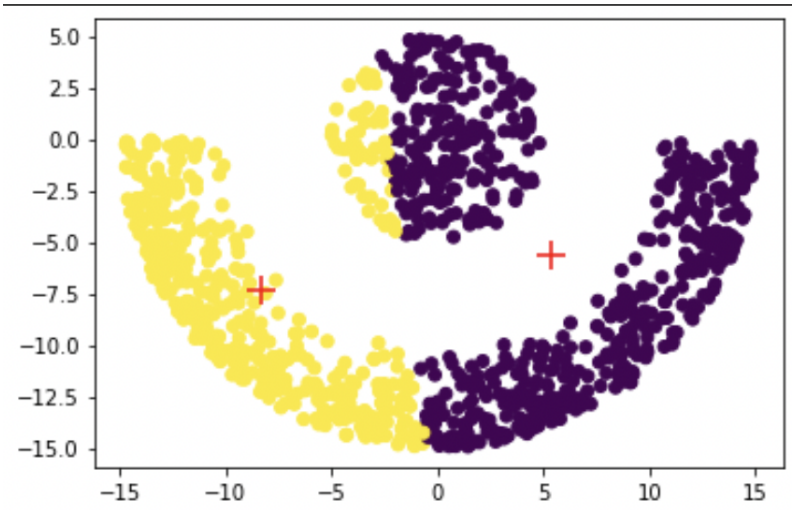
The steps involved are:

1. Points are assigned clusters according to z0, and these are then plotted.
2. The means are computed and put in an array called 'means'.
3. Points and means are then plotted.
4. Points are reassigned to clusters and the objective is appended to objlist.
5. The new assignment is then plotted along with the means.
6. Now we iteratively compute the means and reassign until convergence.
7. Finally, the cluster figure as well as the objective functions are plotted.

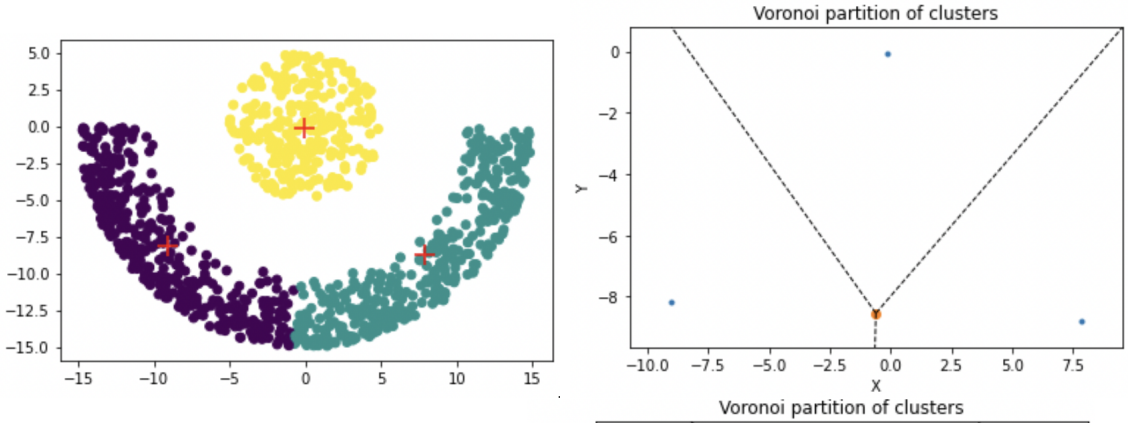
We now also create a function that allows us to randomly initialise assignments based on some number of clusters k and some number of points n. I also write a function to plot Voronoi regions.

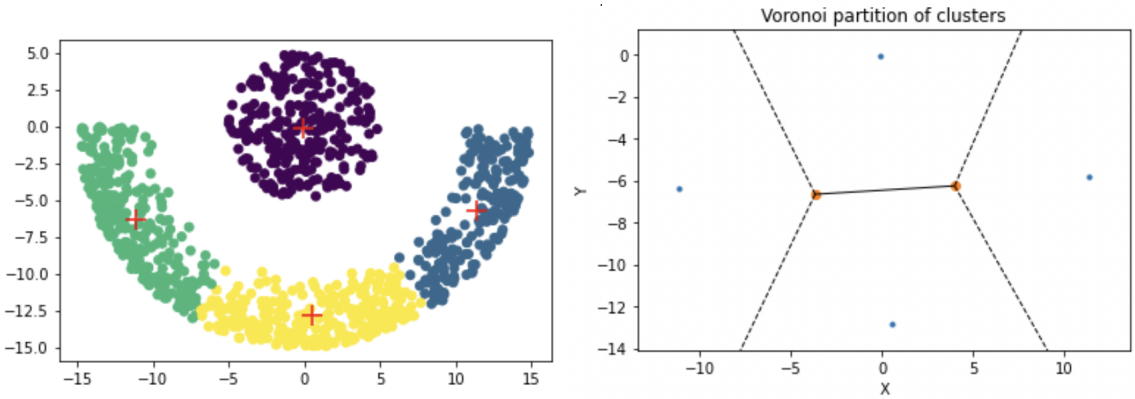
Now, we can put everything together and run the KMeans function for the cases specified in the question. We run:

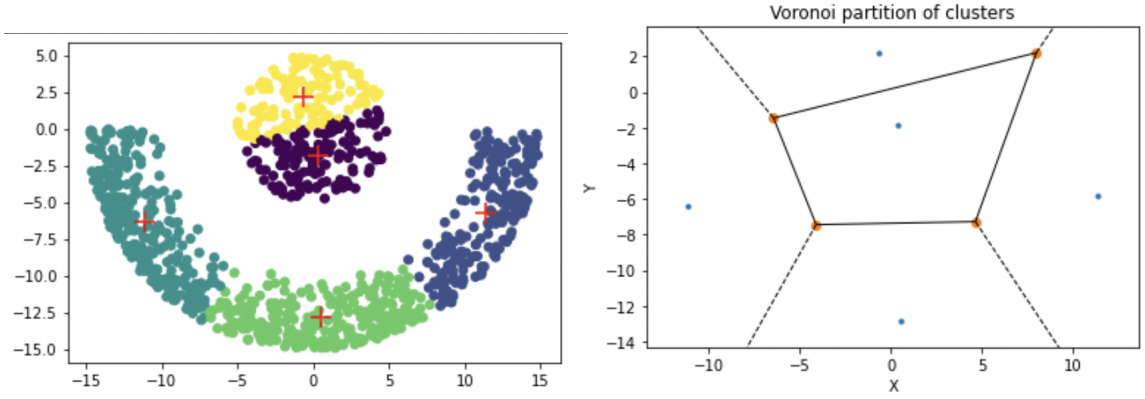
The result for the clustering is as follows:



For k = 3, 4, 5, we have shown the clustering results as well as the Voronoi regions obtained:







2.3 Part (iii)

We will do this by 2 methods.

Method 1:

```
from scipy.linalg import fractional_matrix_power

def rbfKernel(x,s):
  n = len(x)
  K = np.zeros((n,n))
  for i in range(n):
    for j in range(n):
      K[i][j] = math.exp(-1*np.dot(x[i]-x[j],x[i]-x[j])/(2*s**2))

  return K

df = pd.read_csv("cm_dataset_2.csv",names=['x','y'])
x = df['x'].to_numpy()
y = df['y'].to_numpy()
n_samples = 1000

print("Plotting the original data on a scatterplot: ")
plt.scatter(x,y)
plt.xlabel('X')
plt.ylabel('Y')
plt.show()

pts = np.zeros((n_samples,2)) # n points of degree 2
for i in range(n_samples):
  pts[i] = np.array([x[i],y[i]])

A = rbfKernel(pts,0.28)

D = np.zeros((n_samples,n_samples))
for i in range(n_samples):
  sumi = 0
  for j in range(n_samples):
    sumi += A[i][j]
  D[i][i] = sumi

Dh = fractional_matrix_power(D, -0.5)
L = np.matmul(np.matmul(Dh,A),Dh)

evals,evecs = np.linalg.eigh(L)
x1 = evecs[:,-1]
x2 = evecs[:,-2]

X = np.array([x1,x2])
X = np.transpose(X)

Y = np.zeros((n_samples,2))
for i in range(n_samples):
  norm = 0
```

```
    for j in range(2):
        norm+=X[i][j]**2
    norm = math.sqrt(norm)
    Y[i] = X[i]/norm

T = np.transpose(Y)

print("Running K-means on the rows of Y matrix: ")
clusters = KMeans(T[0],T[1],randInit(2,n_samples),2)

print("Plotting the final clusters obtained: ")
plotPoints(x,y,clusters)
```

Explanation:

We will attempt to run the specific form of the spectral clustering algorithm that was given by Andrew Ng in his paper, with a hope that we would be able to arrive at a good clustering of the data.

Here, we are given k = 2.

We follow the algorithm as mentioned here:

1. Set A to a RBF Kernel.
2. D is set to a diagonal matrix where $D_{i,i} = \Sigma_{j=1}^{n} A_{ij}$
3. Construct $L = D^{-1/2} A D^{-1/2}$
4. Find the 2 largest eigenvectors of L (k=2), and construct matrix X as -

$$X = [x_1 x_2]$$

where the columns of X are the 2 most important eigenvectors.

5. Create a matrix Y which is X normalised row-wise.
6. Cluster each row of Y according to K-means as if they were datapoints.
7. Assign clusters to the original samples according to the results of clustering.

We first define the rbfKernel function.

Then we fetch the data and define the x and y arrays, as well as n_samples, the number of samples (1000). Now we plot the original data on a scatterplot, which has been shown in the previous question.

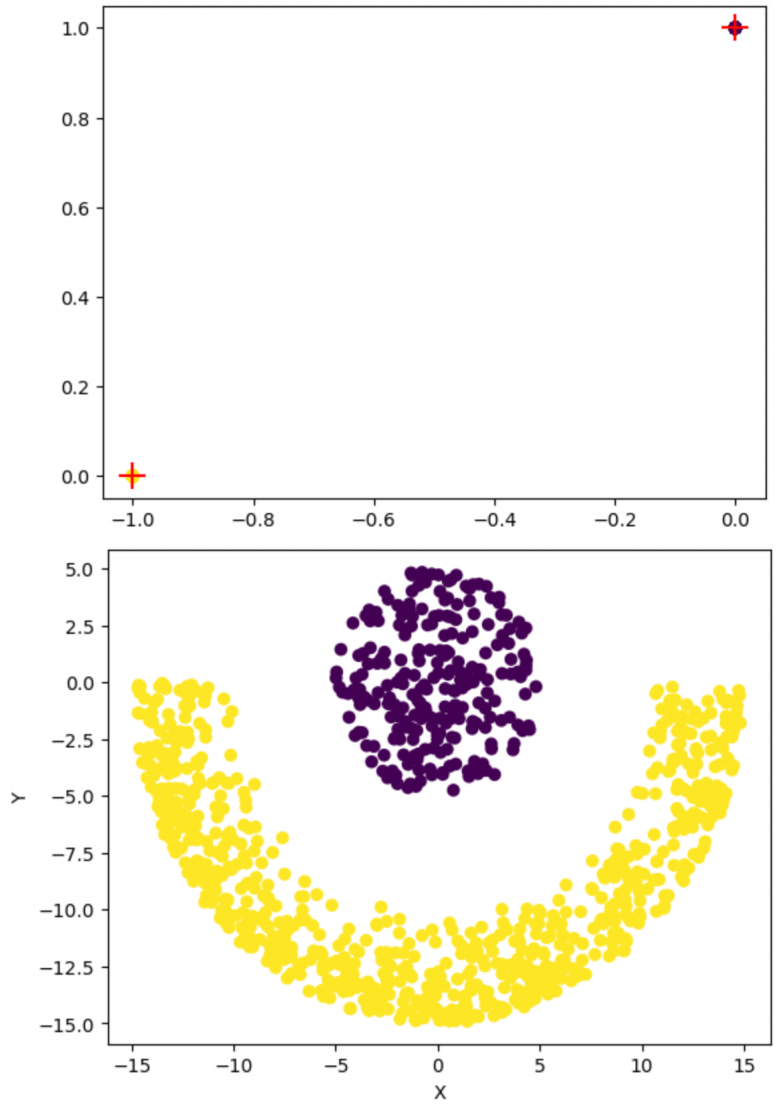
We then define D, the diagonal matrix as mentioned in the algorithm. We then define Dh as $D^{1/2}$.

Then, we calculate L, and find its top 2 eigenvectors and store in matrix X. We further calculate Y and then we define a matrix T as the transpose of matrix Y.

Matrix T is useful because its elements can be fed directly into the KMeans clustering function.

Now, we perform KMeans on the rows of T and plot the original points (x,y) according to the clustering formed.

The plots are ready.

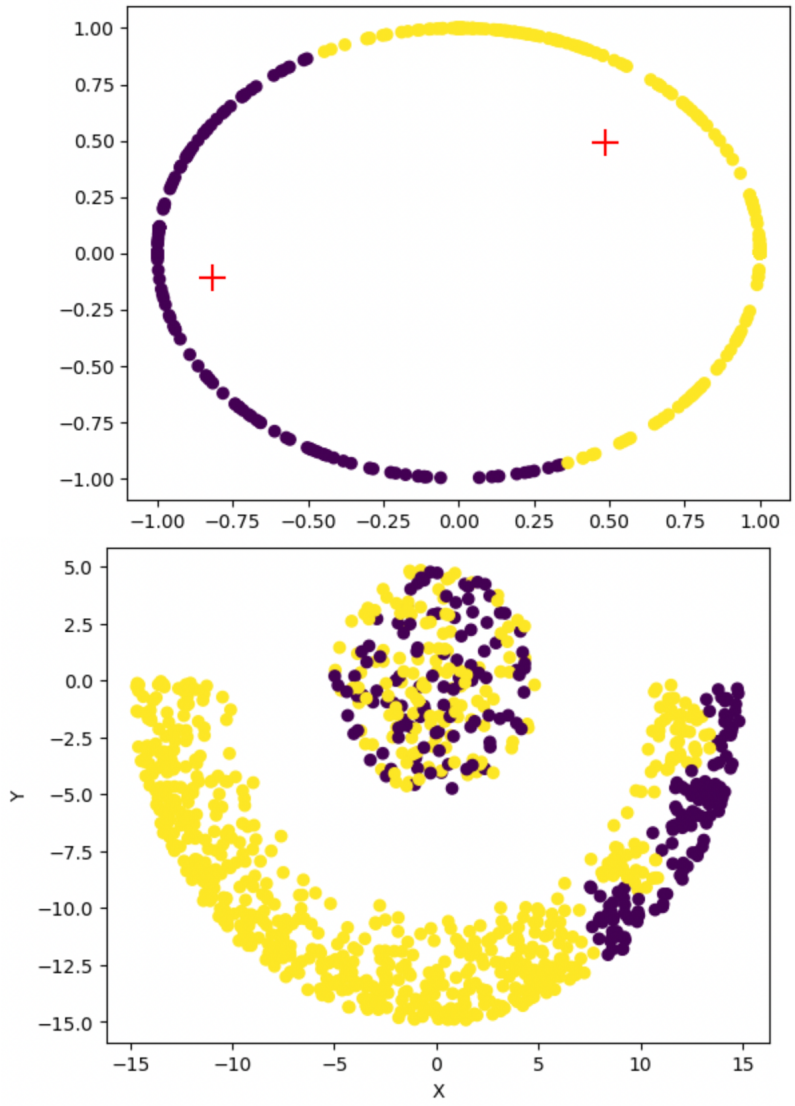


Method 2:

```
evals,evecs = np.linalg.eigh(A)

H = np.array([evecs[:,-1],evecs[:,-2]])

Hn = np.zeros((n_samples,2))
for i in range(n_samples):
    norm = 0
    for j in range(2):
        norm+=H[j][i]**2
    norm = math.sqrt(norm)
    Hn[i] = H[:,i]/norm

print("Running K-means on the rows of the normalised H matrix: ")
clusters = KMeans(Hn[:,0],Hn[:,1],randInit(2,n_samples),2)

print("Plotting the final clusters obtained: ")
plotPoints(x,y,clusters)
```

Explanation:

Here, we simply use the original kernel matrix, and without any of the intermediate steps, we we calculate the eigenvectors of the matrix, normalise the rows of H, and apply KMeans to it. The final clusters are then plotted.



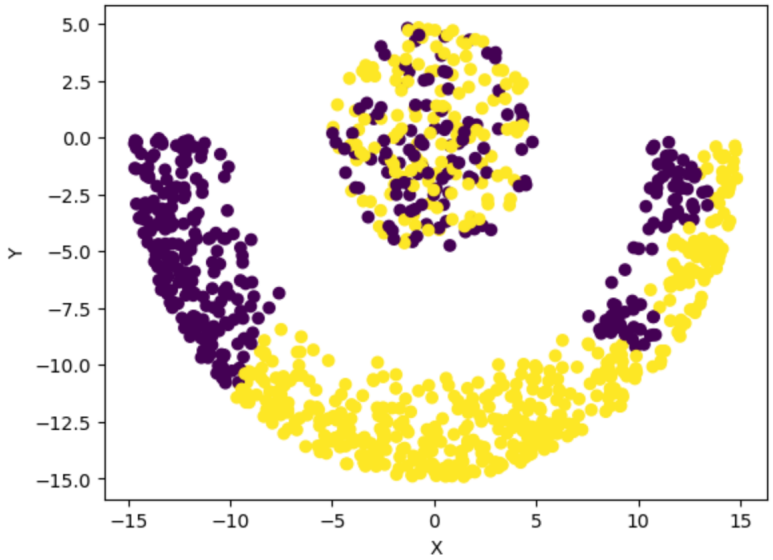
2.4 Part (iv)

For this part, the code below sums up what needs to be done.

```
evals, evecs = np.linalg.eigh(A)

H = np.array([evecs[:,-1], evecs[:,-2]])
T = np.transpose(H)

clusters = np.zeros((n_samples,))
for i in range(n_samples):
  if(T[i][1]>T[i][0]):
    clusters[i] = 1

print("Plotting the final clusters obtained: ")
plotPoints(x,y,clusters)
```

We first calculate the eigenvectors of the kernel matrix A. Then, we store the best two eigenvectors into a matrix H. Then, we create a clusters array that is 0 if the 0th index of T[i] is larger, else stores 1. This follows the clustering rule specified in the question.

We can then plot the points with the clusters array.



As we can see, this mapping performs almost identically to the second method of Part (iii). The reason for this is that taking the maximum of every row is equivalent to finding the row that maximises a specific feature (or a vector that is along a particular direction). Now, when we cluster the normalised rows, we are basically eliminating the effect of distance from the origin, and we are clustering based solely on the direction. This is the same as clustering those points which have a higher value of a certain feature than others. Thus, they perform quite similarly.