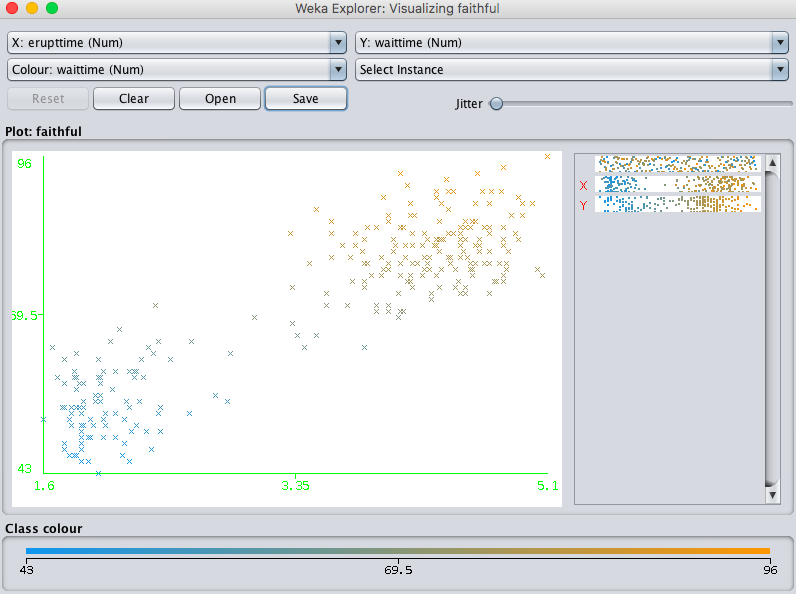
**Question 1:**



Can see 2 clusters.

1. **Steps for Hierarchical Clustering algorithm**

* Make each instance in the dataset into a trivial mini-cluster.
* Find two closet clusters and merge them. Keep repeating this until all the clusters are merged.
* When all the clusters are merged and only one cluster is left, the clustering stops.

As the number of records increase, the performance of hierarchical clustering decreases and time for execution increases. Hierarchical clustering is better for small datasets and hence it is perfect for the above question.

**Question 2**

1. K-means clustering Python code:

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.animation as animation

def load\_dataset(name):

return np.loadtxt(name)

def euclidian(a, b):

return np.linalg.norm(a-b)

def plot(dataset, history\_centroids, belongs\_to):

colors = ['r', 'g']

fig, ax = plt.subplots()

for index in range(dataset.shape[0]):

instances\_close = [i for i in range(len(belongs\_to)) if belongs\_to[i] == index]

for instance\_index in instances\_close:

ax.plot(dataset[instance\_index][0], dataset[instance\_index][1], (colors[index] + 'o'))

history\_points = []

for index, centroids in enumerate(history\_centroids):

for inner, item in enumerate(centroids):

if index == 0:

history\_points.append(ax.plot(item[0], item[1], 'bo')[0])

else:

history\_points[inner].set\_data(item[0], item[1])

print("centroids {} {}".format(index, item))

plt.pause(0.8)

def kmeans(k, epsilon=0, distance='euclidian'):

history\_centroids = []

if distance == 'euclidian':

dist\_method = euclidian

dataset = load\_dataset('faithful.txt')

# dataset = dataset[:, 0:dataset.shape[1] - 1]

num\_instances, num\_features = dataset.shape

prototypes = dataset[np.random.randint(0, num\_instances - 1, size=k)]

history\_centroids.append(prototypes)

prototypes\_old = np.zeros(prototypes.shape)

belongs\_to = np.zeros((num\_instances, 1))

norm = dist\_method(prototypes, prototypes\_old)

iteration = 0

while norm > epsilon:

iteration += 1

norm = dist\_method(prototypes, prototypes\_old)

prototypes\_old = prototypes

for index\_instance, instance in enumerate(dataset):

dist\_vec = np.zeros((k, 1))

for index\_prototype, prototype in enumerate(prototypes):

dist\_vec[index\_prototype] = dist\_method(prototype,

instance)

belongs\_to[index\_instance, 0] = np.argmin(dist\_vec)

tmp\_prototypes = np.zeros((k, num\_features))

for index in range(len(prototypes)):

instances\_close = [i for i in range(len(belongs\_to)) if belongs\_to[i] == index]

prototype = np.mean(dataset[instances\_close], axis=0)

# prototype = dataset[np.random.randint(0, num\_instances, size=1)[0]]

tmp\_prototypes[index, :] = prototype

prototypes = tmp\_prototypes

history\_centroids.append(tmp\_prototypes)

# plot(dataset, history\_centroids, belongs\_to)

return prototypes, history\_centroids, belongs\_to

#######Scatter Plot########

def execute():

dataset = load\_dataset('faithful.txt')

centroids, history\_centroids, belongs\_to = kmeans(2)

plot(dataset, history\_centroids, belongs\_to)

execute()

###Objective Function Plot####

def pick\_mu(k):

dataset = load\_dataset('faithful.txt')

x\_copy=dataset

np.random.shuffle(x\_copy)

mu=x\_copy[:k]

return mu

def obj(k):

L=[ ]

mu=pick\_mu(k)

c=np.zeros(500).astype(int)

c=c.astype(int)

x = load\_dataset('faithful.txt')

dataset= np.vstack((x))

for t in range(20):

for i in range(dataset.shape[0]):

dist=[]

for k1 in range(k):

dist.append(np.sqrt(np.sum((dataset[i]-mu[k1])\*\*2)))

c[i]=np.argmin(dist)+1

for k1 in range(k):

nk=0

for i in range(dataset.shape[0]):

if c[i]==k1+1:

nk=nk+1

val =np.zeros((1,2))

for i in range(dataset.shape[0]):

if c[i]==k1+1:

val=np.add(val,dataset[i])

mu[k1]=val/nk

sum1=0

for i in range(dataset.shape[0]):

for k1 in range(k):

if c[i]==k1+1:

sum1=sum1+(np.sum((dataset[i]-mu[k1])\*\*2))

L.append(sum1)

return L,mu,c

x\_axis = np.arange(1, 21)

plt.figure(figsize=(9,6))

l2,mu2,c2=obj(2)

plt.plot(x\_axis,l2, label="k=2")

plt.xticks(np.arange(min(x\_axis), max(x\_axis)+1, 1.0))

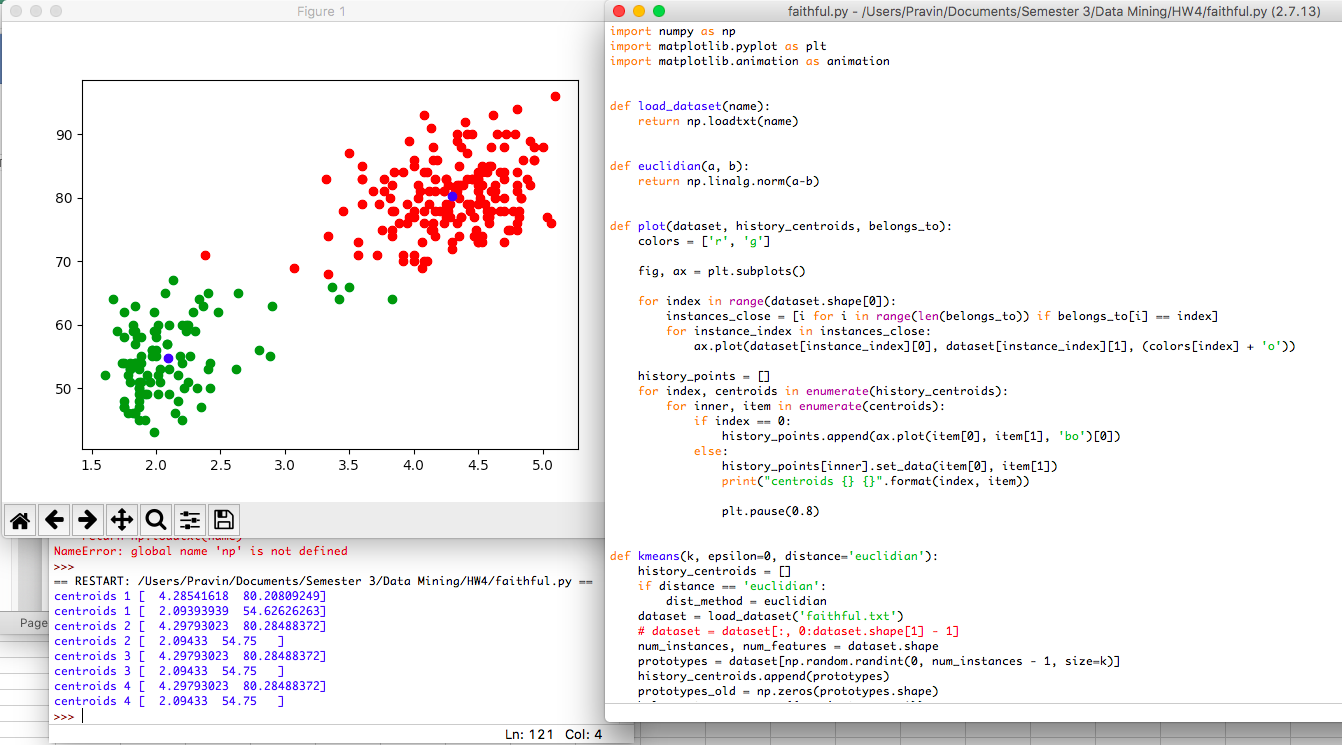
plt.title("Objective Training Function L per iteration --->")

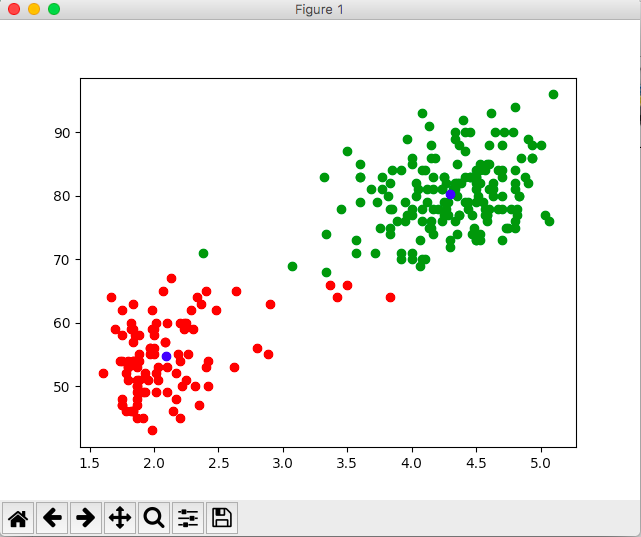
plt.xlabel("No of iterations --->")

plt.ylabel("Objective Training Function")

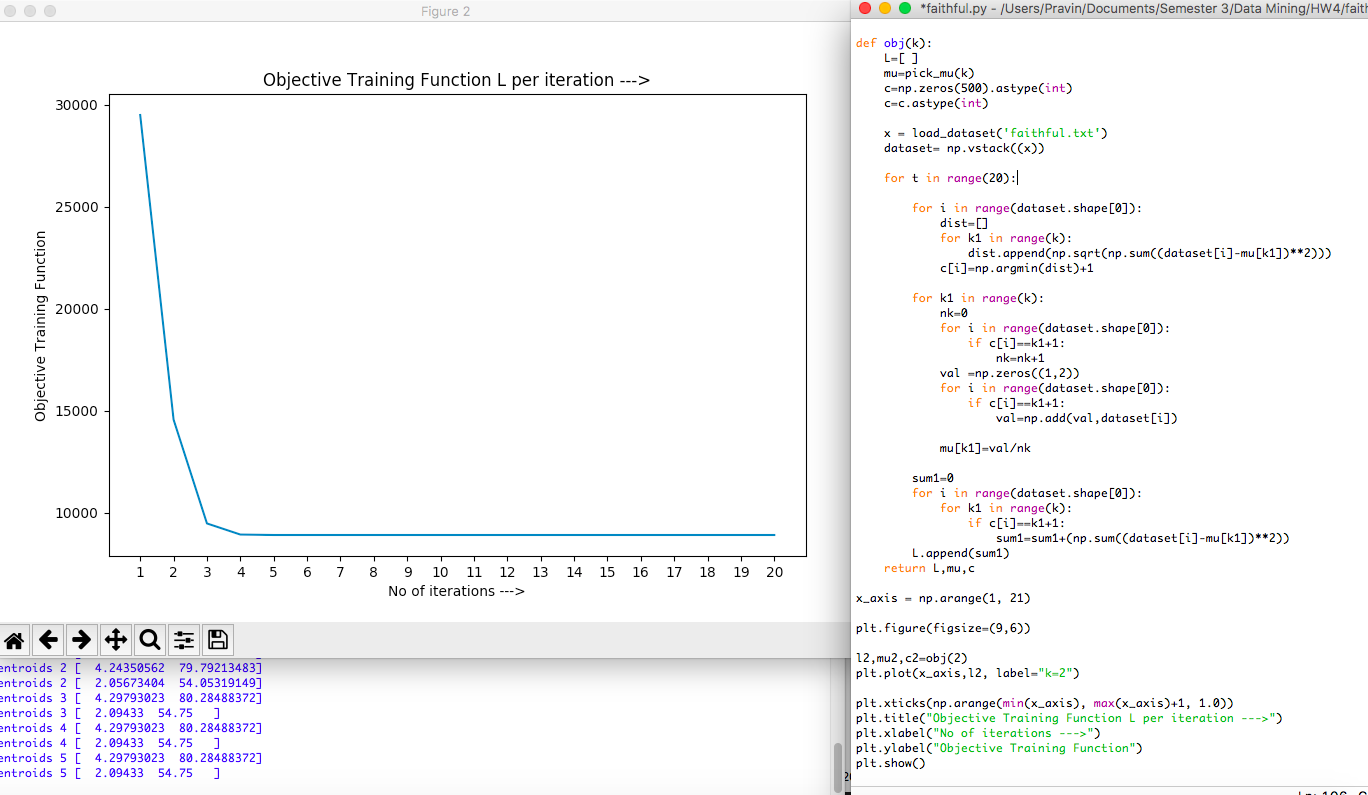
plt.show()

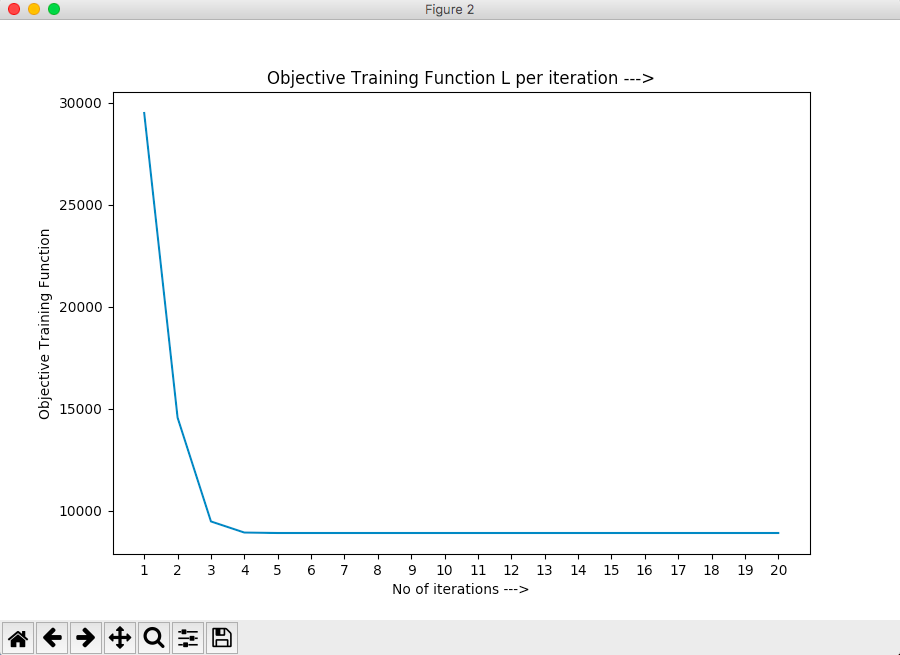
**b)**





**c)**





1. Yes, using K-means clustering helped.

**Question 3**

**a)**

|  |  |
| --- | --- |
| **K-means** | **EM** |
| Hard assign a data point to one particular cluster on convergence. | Soft assigns a data point to clusters. It gives a probability of any point belonging to any centroid. |
| It used L2 norm for optimization. | This doesn’t depend on the L2 norm but is based on the expectation means probability of a point belonging to one particular cluster. |
| Input: The number of k and database containing n objects.  Output: Set of k-clusters that minimize squared-error criterion. | Input: Cluster number k, database and stopping tolerance  Output: Set of k-clusters with weight that maximize log-likelihood function. |
| K-means is concerned with explicitly identifying distinct sub-populations | EM seeks to model the entire sampling density |

1. Density Estimation:

Let X1, X2,….Xn be the sample from distribution P with density p. The goal of density estimation is to estimate p with as few assumptions about p as possible.

It estimates the probabilistic distribution over instance space p(x).

**OR** It constructs an estimate of unobservable underlying PDF, based on a set of observed data.

Kernal Density Estimation:

KDE on the other hand avoids the problem of choice of number of components by using one component centered on each point of the dataset.

In Kernel density function for an instance in random sample *x*, it estimates the Probability density function P(X) from which the sample was derived. Kernel function is superposed on each observation enabling each observation to contribute for the PDF estimate.  Kernel Density function is a classic form of non-parametric estimation.

EM Algorithm for Gaussian Mixture Models:

It is an iterative algorithm that start some initial estimate Θ (e.g. random), and then proceeds to iteratively update Θ until convergence is detected. Each iteration consists of an E-step and M-step.

A Gaussian mixture model is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. Mixture models are used for generalizing k-means clustering to incorporate information about the covariance structure of the data as well as the centers of the latent Gaussians. They are referred to as semi-parametric methods.