Hrishikesh Hippalgaonkar
hippalga@usc.edu

EE 511 Simulation Methods for Stochastic Systems
Project #3

https://github.com/hrishi121/Simulation-methods-for-Stochastic-Systems

❖ K-means Clustering:

K-means clustering is one of the popular clustering methods. In *k-means clustering* the data is clustered into k partitions. The clustering is done in such a way that each data sample in a cluster if it has the smallest distance from that cluster centroid as compared to the distance of the data sample from other cluster centroid. *K-means clustering* is like the *expectation-maximization algorithm* for *Gaussian Mixture Models.* The algorithm for the *k-means clustering* is:
Assume some initial *k* centroids, each for one of the *k* clusters
Assignment Step: Assign each data sample to the cluster whose centroid has the least squared Euclidean distance, this is intuitively the "nearest" mean. It can be mathematically represented as

$$\boldsymbol{x_t \in S_i \ iff \ \|x_t - \mu_i\| > \|x_t - \mu_j\| \ \forall j \neq i, 0 \leq j \leq k}$$

where, x_t is the data sample, S_i is one of the clusters and μ_i represents its cluster centroid.
Update Step: Calculate the new centroids for each of the clusters based on the data samples assigned to each of the clusters.

$$\boldsymbol{\mu_i = \frac{1}{|S_i|} \sum_{x_k \in S_i} x_k}$$

The algorithm converges when the centroids no longer change, or a maximum number of pre-determined iterations have been achieved. There is no guarantee that the optimum will be found using this algorithm.

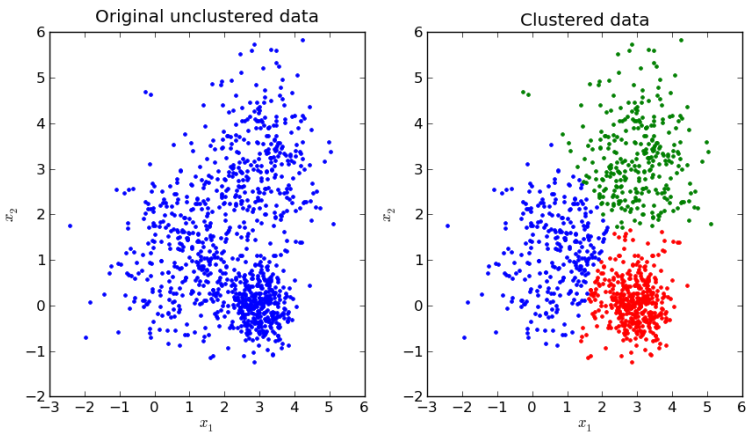


Figure 1. Example of *k-means clustering*

1. Problem 1

[Testing Faith]

Download the "old faithful" data set from blackboard. This contains samples of a 2-D random variable: the first dimension is the duration of the old faithful geyser eruptions. The second is the waiting time between eruptions. Generate a 2-D scatter plot of the data. Run a k-means clustering routine on the data for k=2. Show the two clusters in a scatterplot.

Approach: The *scikit-learn k-means clustering* function can be used for the above data-set. The function takes a number different arguments such as total number of clusters, maximum number of iterations, centroid initialization method etc.

Code:

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

df = pd.read_table('Old_Faithful_2.txt', delim_whitespace=True, names=('#', 'Eruption
Time', 'Waiting Time'))
data = df.values
data = data[:,(1,2)]

kmCluster = KMeans(n_clusters = 2, init= 'k-means++', random_state=10).fit(data)
cluster_labels = kmCluster.labels_
silhouette_avg = silhouette_score(data, cluster_labels, metric = 'euclidean')
print("Average silhouette score: ", silhouette_avg)

plt.figure(1)
plt.scatter(data[:,0], data[:, 1], c = cluster_labels, cmap='viridis')
plt.title("Scatter-plot for the 'Old_Faithful' data")
plt.xlabel("Eruption Time")
plt.ylabel("Waiting Time")
plt.grid(True, alpha = 0.3)
```

Result:

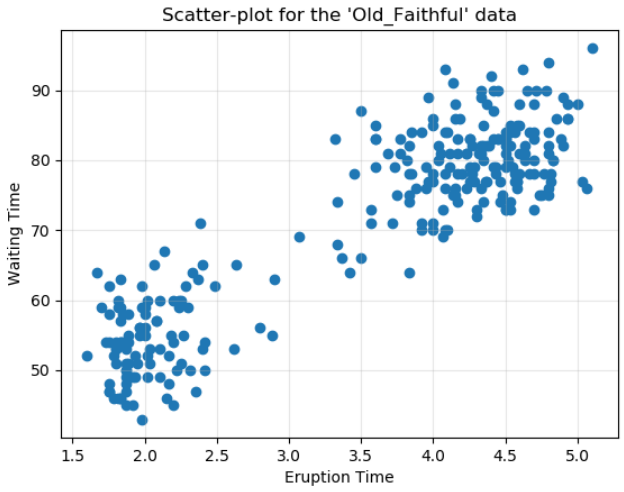


Figure 2. Scatter-plot of the 'Old_Faithful' data-set

Upon visual inspection, the data can be clustered into two clusters, with one cluster centered at the top right and another cluster centered at the bottom left. The clusters provided by the k-means clustering match our intuition, and this can be observed in the figure below.

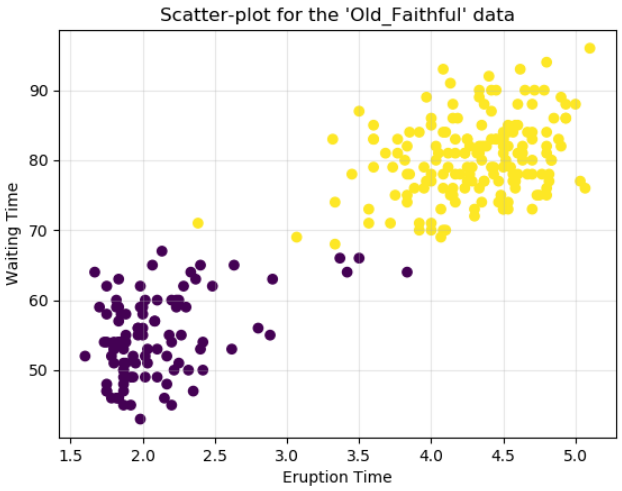


Figure 3. Scatter-plot for the clustered 'Old_Faithful' data-set using *k-means clustering*

❖ Gaussian Mixture Model:

A mixture model is a probability distribution model for representing clusters within an overall population, without requiring an observed data to identify these clusters. Gaussian Mixture Models (GMMs) are mixture models composed of different gaussian distributions. GMMs are among the most statistically mature methods for clustering. GMM is a parametric probability density function represented as a weighted sum of gaussian component densities. Mathematically, GMM can be represented as,

$$p(x|\lambda) = \sum_{i=1}^{M} w_i g(x_i|\mu_i, \Sigma_i)$$

Where, x_i is a d-dimensional data vector (feature vector), $w_i \;\; i = 0,1, \ldots . M$ are the mixture weights and $g(x_i|\mu_i, \Sigma_i)\; i = 0,1, \ldots M$ are the component gaussian densities. The component density is a d-variate Gaussian function of the form

$$g(x_i|\mu_i, \Sigma_i) = \frac{1}{(2\pi)^{d/2}|\Sigma_i|^{1/2}} exp\left\{-\frac{1}{2}(x-\mu_i)'{\Sigma_i}^{-1}(x-\mu_i)\right\}$$

❖ Expectation – Maximization:

In statistics, an expectation–maximization (EM) algorithm is an iterative method to find maximum likelihood or maximum a posteriori (MAP) estimates of parameters in statistical models, where the model depends on unobserved latent variables. The EM iteration alternates between performing an expectation (E) step, which creates a function for the expectation of the log-likelihood evaluated using the current estimate for the parameters, and a maximization (M) step, which computes parameters maximizing the expected log-likelihood found on the *E* step. These parameter-estimates are then used to determine the distribution of the latent variables in the next E step.

2. Problem 2

[EM]

- Write a 2-dimensional RNG for a Gaussian mixture model (GMM) pdf with 2 subpopulations. Use any function/sub-routine available in your language of choice.
- Implement the expectation maximization (EM) algorithm for estimating the pdf parameters of 2-D GMMs from samples (Refer to the Noisy Clustering Paper linked on blackboard for the relevant update equations).
- Compare the quality and speed your GMM-EM estimation on 300 samples of GMM distributions featuring each of the following: a) spherical covariance matrices, b) ellipsoidal covariance matrices, and c) poorly-separated subpopulations.
- Apply your GMM-EM algorithm to fit the "old faithful" data set to a GMM pdf with two components.

Approach: For the first part of the question I initialized two separate bivariate Gaussian distributions, each with a different mean and covariance matrix. Then I used the *scikit-learn* Function `mixture.GaussianMixture.fit` function to find the probability distribution of a single bivariate gaussian mixture which fits both the gaussian distributions. Then I used the `mixture.sample` function to draw sample from the bivariate gaussian mixture

For the second part, I created two functions, one each for the Expectation step and Maximization step. The algorithm for both the steps is as following:

Start with some randomly initialized cluster parameters for each of the 'C' clusters:

$$Mean: \mu_c, \quad Covariance: \Sigma_c, \quad Mixture\ weight: \pi_c$$

Expectation Step (E-Step):

For each data sample x_i,

Compute p_{ic} , the probability that it belongs to a cluster 'C', and normalize it to sum to one (over all clusters 'C')

$$p_{ic} = \frac{\pi_C g(x_i|\mu_C, \Sigma_C)}{\sum_{C'} \pi_{C'} g(x_i|\mu_{C'}, \Sigma_{C'})}$$

Maximization Step (M-Step):

Once we have p_{ic} update the parameters *mean, covariance and mixture weight* for each of the clusters

$$m_c = \sum_i p_{ic}$$

$$\pi_c = \frac{m_c}{\sum_{C'} m_{C'}}$$

$$\mu_C = \frac{1}{m_C} \sum_i p_{ic}\, x_i$$

$$\Sigma_C = \frac{1}{m_C} \sum_i p_{ic} (x_i - \mu_C)'(x_i - \mu_C)$$

Now, pass these updated parameters to the E-step and run the both steps in a loop until the change is parameters is very low.

Code: (Part a, c)

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import mixture
import EM_algorithm as em
from scipy.stats import multivariate_normal
from matplotlib.colors import LogNorm

def main():

    # generate spherical data centered on (10, 10)
    shifted_gaussian = []
    mu1 = np.array([10, 10])
    sigma1 = np.array([[3, 0], [0, 3]])
    for i in range(0,300):
        shifted_gaussian.append(multivariate_normal.rvs(mu1, sigma1))


    # generate [-5, -5] centered stretched Gaussian data (ellipsoidal)
    stretched_gaussian = []
```

```
mu2 = np.array([-5, -5])
sigma2 = np.array([[3, 0], [0, 1]])
for i in range(0,300):
    stretched_gaussian.append(multivariate_normal.rvs(mu2, sigma2))

# generated poorly distributed data
poorly_distributed_gaussian = []
mu2 = np.array([-5, -5])
sigma2 = np.array([[6, 3], [2, 4]])
for i in range(0,300):
    poorly_distributed_gaussian.append(multivariate_normal.rvs(mu2, sigma2))

# concatenate the two datasets into the final training set
X_train = np.vstack([shifted_gaussian, poorly_distributed_gaussian])

# fit a Gaussian Mixture Model with two components
clf = mixture.GaussianMixture(n_components=2, covariance_type='full')
clf.fit(X_train)

# display predicted scores by the model as a contour plot
x = np.linspace(-20., 30.)
y = np.linspace(-20., 40.)
X, Y = np.meshgrid(x, y)
XX = np.array([X.ravel(), Y.ravel()]).T
Z = -clf.score_samples(XX)
Z = Z.reshape(X.shape)

plt.figure(4)
CS = plt.contour(X, Y, Z, norm=LogNorm(vmin=1.0, vmax=1000.0),
                 levels=np.logspace(0, 3, 10))
CB = plt.colorbar(CS, shrink=0.8, extend='both')
plt.scatter(X_train[:, 0], X_train[:, 1], .8)

plt.title('Mixture of two Bivariate Gaussian distributions ')
plt.axis('tight')
plt.show()

data, label = clf.sample((300))

data = np.asarray(data)
numSamples = np.size(data, axis = 0)

wt1 = np.random.rand(np.size(data, axis = 0), 1)
wt2 = np.empty([numSamples, 1])
for i in range(0, np.size(data, axis = 0)):
    wt2[i] = 1 - wt1[i]
weights = (wt1, wt2)
parameters = em.Mstep(data, weights)

loglikelihood = np.empty([500])
for i in range(0, 500):
    #print("Iteration: ", i)
    (weights, loglike) = em.Estep(data, parameters)
    loglikelihood[i] = loglike
    parameters = em.Mstep(data, weights)

param1, param2, param3, param4, param5 = parameters
print("Final parameters: ")
print("\nMean 1: ",param1)
print("Covariance 1: ",param3)
```

```
        print("Mean 2: ",param2)
        print("Covariance 2: ",param4)
        print("Mixture weight: ", param5)


        label1 = np.empty([300])
        weights = np.asarray(weights)
        for i in range(0, 300):
            if (weights[0, i] > 0.5):
                label1[i] = 0
            else:
                label1[i] = 1
        #print(label1)

        plt.figure(3)
        plt.scatter(data[:, 0], data[:, 1], c= label1.reshape(300))
        plt.title(" Clustered mixture of bivariate Gaussians usin EM algorithm")
        plt.xlabel("Feaeture 1")
        plt.ylabel("Feaeture 2")

    if __name__ == "__main__":
main()
```

Code: (Part b: EM Algorithm)

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from math import log
from matplotlib.colors import LogNorm


"Calculate probability of a data point given the current parameters"
def MultivariateGaussianPDF(datum, mu1, sigma1):

    sigma = np.asarray(sigma1)
    mu = np.asarray(mu1).reshape(2)
    data = np.asarray(datum).reshape(2)
    y = multivariate_normal.pdf(data, mean = mu, cov = sigma)
    return y

"The Estimation Step"
def Estep(data, parameters):
    mu1, mu2, sigma1, sigma2, mixWeight = parameters
    i = 0
    loglike = 0
    wt1 = np.empty([np.size(data, axis = 0), 1])
    wt2 = np.empty([np.size(data, axis = 0), 1])
    for dataSample in data:
        # unnormalized weights (or "responsibility")
        wt1[i] = MultivariateGaussianPDF(dataSample, mu1, sigma1) * mixWeight
        wt2[i] = MultivariateGaussianPDF(dataSample, mu2, sigma2) * (1. -
mixWeight)
        # compute denominator
        den = wt1[i] + wt2[i]
        # normalize
        wt1[i] = wt1[i] / den
        wt2[i] = wt2[i] / den
        loglike = loglike + log(wt1[i] + wt2[i])
        i = i + 1
```

```
    # Return the weight tuple
    weights = (wt1, wt2)
    #print("E Step done!")
    return (weights, loglike)

"The Maximation Step"
def Mstep(data1, weights):
    wt11, wt22 = weights
    wt1 = np.asarray(wt11)
    wt2 = np.asarray(wt22)
    data = np.asarray(data1)
    totalDataSamples = np.size(data, axis = 0)
    totalWeight1 = sum(wt1)      #This is the total responsibility allocated to
cluster 1
    totalWeight2 = sum(wt2)      #This is the total responsibility allocated to
cluster 2
    mixWeight = totalWeight1 / totalDataSamples

    # Update the means mu1 and mu2
    feature1 = np.reshape(data[:, 0], (totalDataSamples, 1))
    feature2 = np.reshape(data[:, 1], (totalDataSamples, 1))

    t1 = (sum(wt1 * feature1)) / totalWeight1
    t2 = (sum(wt1 * feature2)) / totalWeight1
    t3 = (sum(wt2 * feature1)) / totalWeight2
    t4 = (sum(wt2 * feature2)) / totalWeight2

    mu1 = np.array([t1, t2])
    mu2 = np.array([t3, t4])


    # Update the covariance matrices sigma1 an sigma2
    sigma1 = np.zeros([2, 2])
    sigma2 = np.zeros([2, 2])
    i = 0
    for datum in data:
        dataSample = np.asarray(datum).reshape(2,1)
        y1 = dataSample - mu1
        y2 = dataSample - mu2
        sigma1 = sigma1 + wt1[i] * np.outer(np.transpose(y1), y1)
        sigma2 = sigma2 + wt2[i] * np.outer(np.transpose(y2), y2)
        i = i + 1

    sigma1 = sigma1 / totalWeight1
    sigma2 = sigma2 / totalWeight2

    parameters = (mu1, mu2, sigma1, sigma2, mixWeight)
    #print("M Step done!")
    return parameters

"Plot the Decision Region using the parameters from EM computation"
def plotDecisionRegion(data, parameters):

    param1, param2, param3, param4, param5 = parameters

    # Make the plot
    u1 = np.linspace(param1[0] - 3, param1[0] + 3, 1000)
    v1 = np.linspace(param1[1] - 30, param1[1] + 30, 1000)
    u2 = np.linspace(param2[0] - 3, param2[0] + 3, 1000)
```

```
        v2 = np.linspace(param2[1] - 30, param2[1] + 30, 1000)

        x1, y1 = np.meshgrid(u1, v1)
        data1 = np.dstack((x1, y1))

        x2, y2 = np.meshgrid(u2, v2)
        data2 = np.dstack((x2, y2))

        sigma1 = np.asarray(param3)
        mu1 = np.asarray(param1).reshape(2)
        sigma2 = np.asarray(param4)
        mu2 = np.asarray(param2).reshape(2)

        z1 = multivariate_normal.pdf(data1, mean = mu1, cov = sigma1)
        z2 = multivariate_normal.pdf(data2, mean = mu2, cov = sigma2)

        plt.figure(1)
        plt.plot(data[:, 0], data[:, 1], "r+")
        plt.contour( x1, y1, z1 )
        plt.contour( x2, y2, z2 )
        plt.title("Clustering using Gaussian Mixture Model and EM algorithm", alpha =
    0.75)
  plt.legend('Data Samples', loc = 4)
```

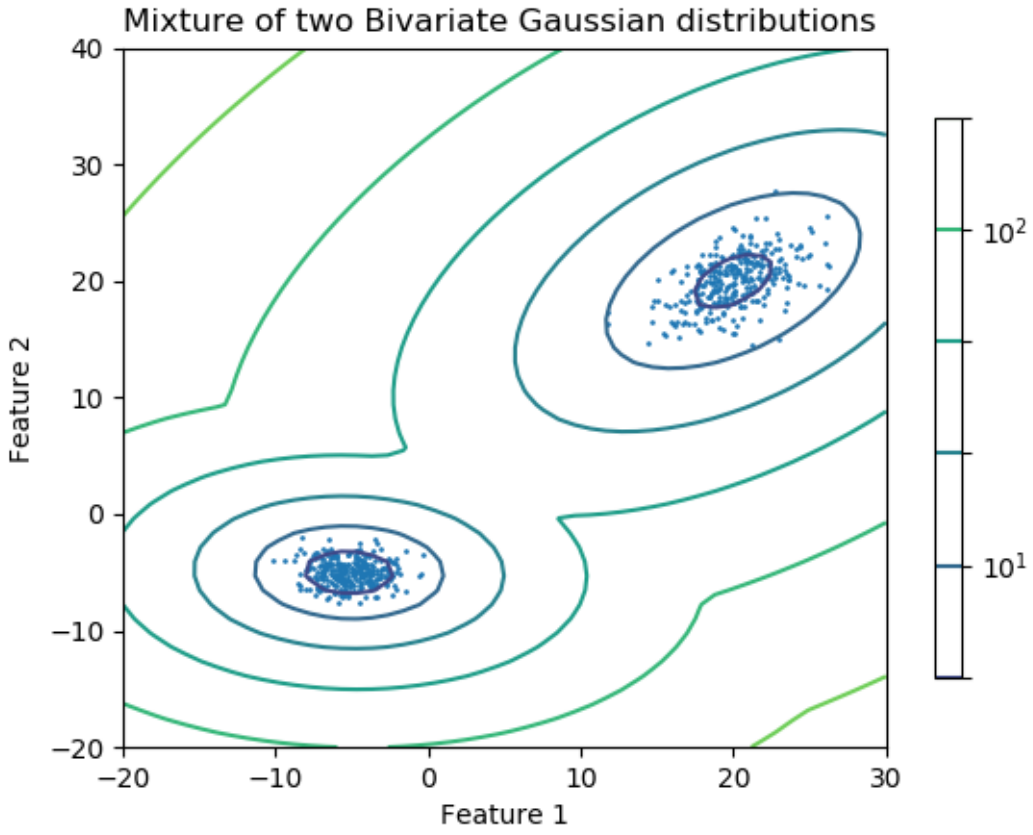
Code: (Part d)

```
    import pandas as pd
    import numpy as np
    import EM_algorithm as em
    import matplotlib.pyplot as plt

    def main():

        df = pd.read_table('Old_Faithful_2.txt', delim_whitespace=True, names=('#',
    'Eruption Time', 'Waiting Time'))
        data = df.values
        data = data[:,(1,2)]
        numSamples = np.size(data, axis = 0)

        wt1 = np.random.rand(np.size(data, axis = 0), 1)
        wt2 = np.empty([numSamples, 1])
        for i in range(0, np.size(data, axis = 0)):
            wt2[i] = 1 - wt1[i]
        weights = (wt1, wt2)
        parameters = em.Mstep(data, weights)

        loglikelihood = np.empty([1000])
        for i in range(0, 1000):
            #print("Iteration: ", i)
            (weights, loglike) = em.Estep(data, parameters)
            loglikelihood[i] = loglike
            parameters = em.Mstep(data, weights)

        param1, param2, param3, param4, param5 = parameters
        print("Final parameters: ")
        print("\nMean 1: ",param1)
        print("Covariance 1: ",param3)
        print("Mean 2: ",param2)
```

```
        print("Covariance 2: ",param4)
        print("Mixture weight: ", param5)

        em.plotDecisionRegion(data, parameters)
        label1 = np.empty([numSamples])
        weights = np.asarray(weights)
        for i in range(0, numSamples):
            if (weights[0, i] > 0.5):
                label1[i] = 0
            else:
                label1[i] = 1
        #print(label1)
        plt.figure(3)
        plt.scatter(data[:, 0], data[:, 1], c= label1.reshape(numSamples))

  if __name__ == "__main__":
main()
```

Result:

A. Mixture of Gaussian distributions with poorly-distributed covariance matrix and ellipsoidal covariance matrix



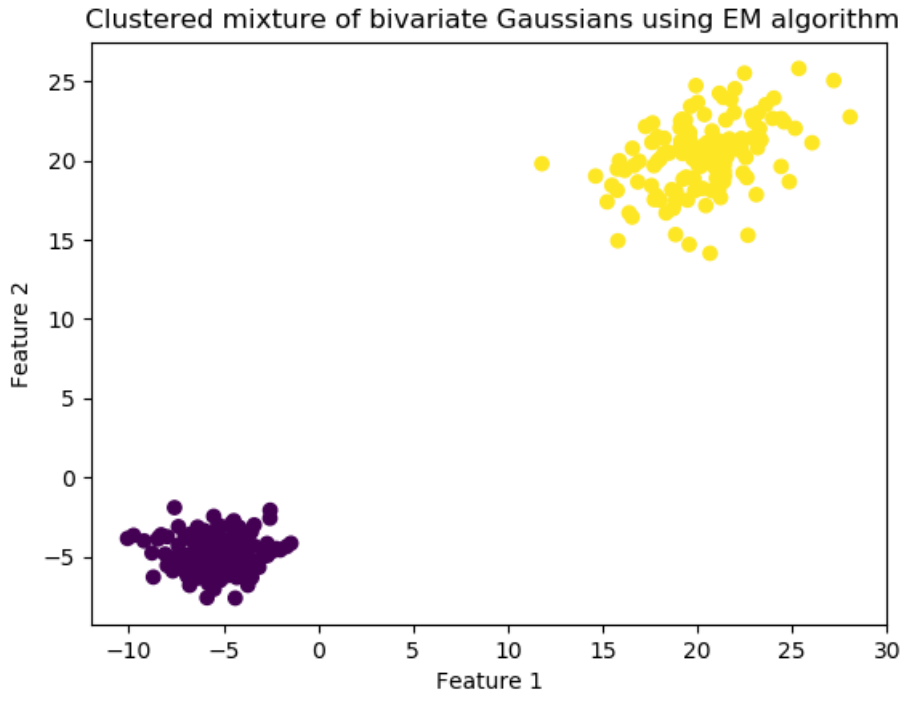
(a) Mixture of two gaussian distributions, one having ellipsoidal covariance matrix and other having a poorly distributed covariance matrix

```
IPython console
Console 1/A

In [14]: runfile('F:/USC/Notes/EE 511/Project 3/Question_2a.py', wdir='F:/USC/Notes/EE
511/Project 3')
Reloaded modules: EM_algorithm
Parameters of the both the Gaussian distributions:

Mean 1:  [20 20]
Covariance 1:  [[6 3]
 [2 4]]
Mean 2:  [-5 -5]
Covariance 2:  [[3 0]
 [0 1]]
```

(b) Parameters of each of the separate Gaussian distributions in the mixture shown above

```
IPython console
Console 1/A

Final parameters of the mixture(calculated by the EM algorihtm):

Mean 1:  [[ 20.4019066 ]
 [ 19.96650195]]
Covariance 1:  [[ 6.00646103  3.01562862]
 [ 3.01562862  3.69964512]]
Mean 2:  [[-4.95184152]
 [-5.15125339]]
Covariance 2:  [[ 3.66008702 -0.14048933]
 [-0.14048933  1.03463397]]
Mixture weight:  [ 0.48]
```

(c) Parameters of each of the separate Gaussian distributions in the mixture calculated by the EM algorithm

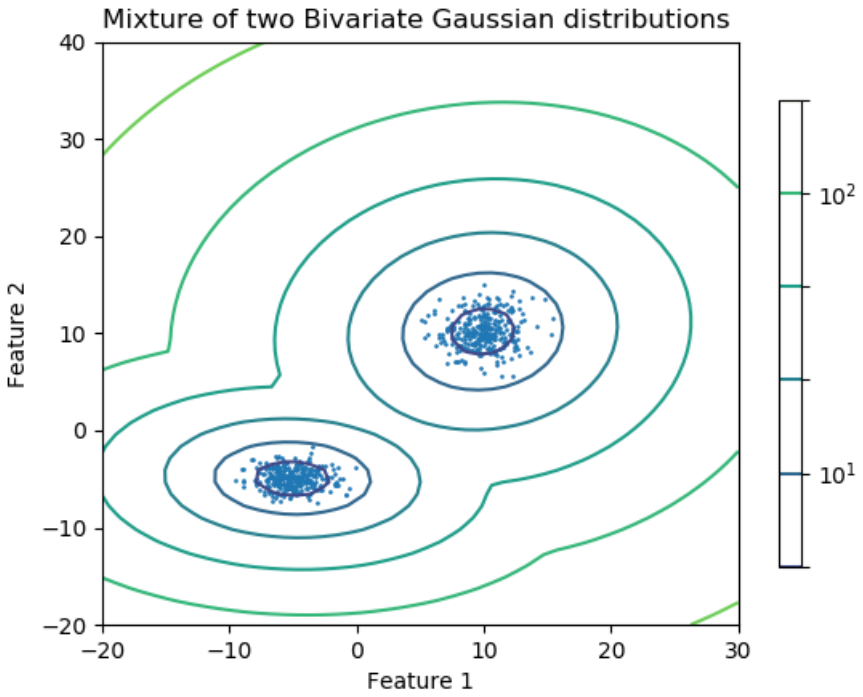


(d) The above gaussian mixture clustered using EM algorithm

```
IPython console
Console 1/A
Total time taken for 500 iterations of EM (in seconds):  43.172631581442374
```

(e) Time required by the EM algorithm

B. Mixture of Gaussian distributions with spherical covariance matrix and ellipsoidal covariance matrix



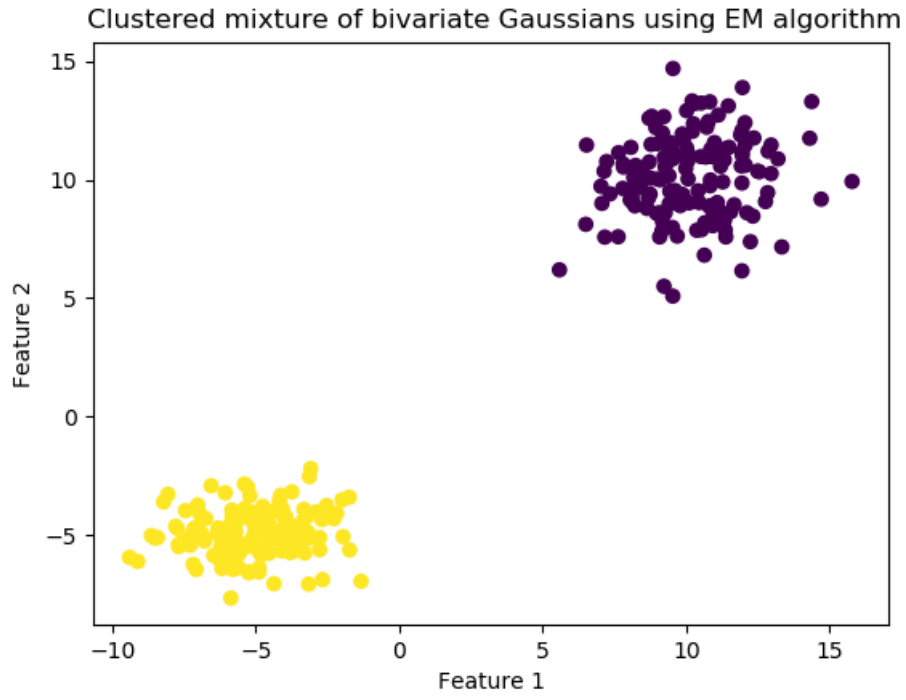
(a) Mixture of two gaussian distributions, one having ellipsoidal covariance matrix and other having a spherical covariance matrix

```
IPython console
Console 1/A
In [18]: runfile('F:/USC/Notes/EE 511/Project 3/Question_2a.py', wdir='F:/USC/Notes/EE
511/Project 3')
Reloaded modules: EM_algorithm
Parameters of the both the Gaussian distributions:

Mean 1:  [10 10]
Covariance 1:  [[3 0]
 [0 3]]
Mean 2:  [-5 -5]
Covariance 2:  [[3 0]
 [0 1]]
```

(b) Parameters of each of the separate Gaussian distributions in the mixture shown above

```
IPython console
Console 1/A
Final parameters of the mixture(calculated by the EM algorihtm):

Mean 1:  [[ 10.17291194]
 [ 10.13340207]]
Covariance 1:  [[ 3.04803189  0.39243863]
 [ 0.39243863  3.01449393]]
Mean 2:  [[-5.13310597]
 [-4.9035519 ]]
Covariance 2:  [[ 2.63836819  0.15349842]
 [ 0.15349842  0.93504997]]
Mixture weight:  [ 0.49666667]
```

(c)Parameters of each of the separate Gaussian distributions in the mixture calculated by the EM algorithm

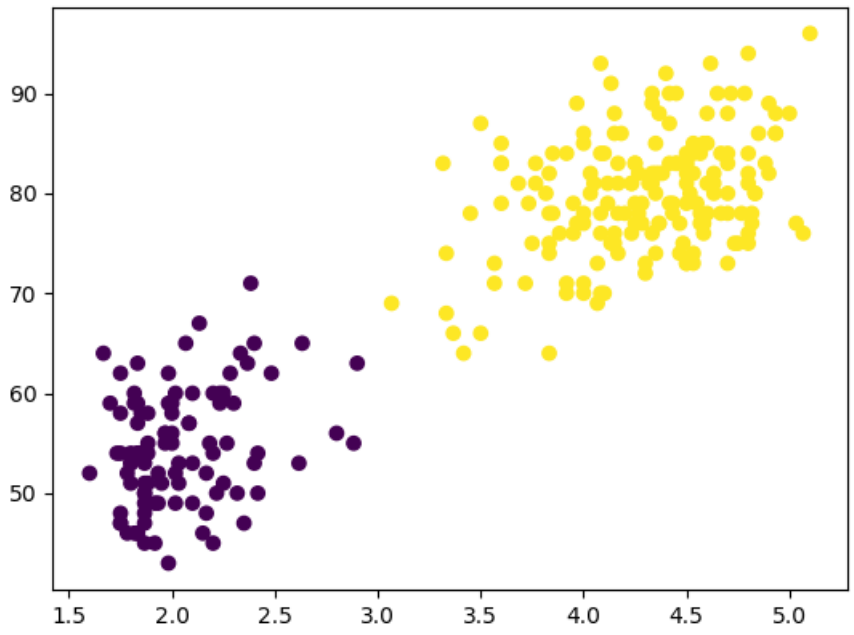


(d) The above gaussian mixture clustered using EM algorithm

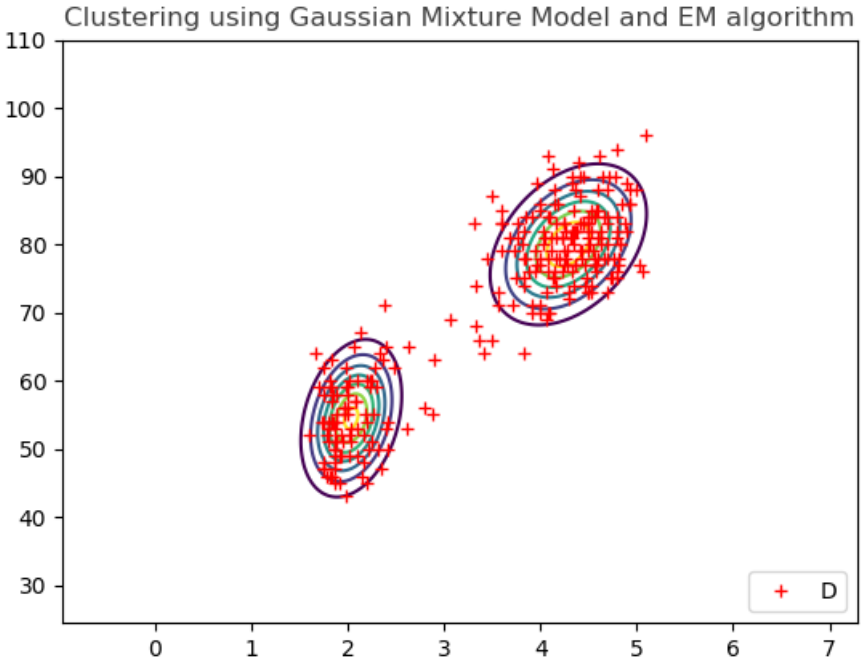
```
IPython console
Console 1/A
Total time taken for 500 iterations of EM (in seconds):  76.67003878722244
```

(e) Time required by the EM algorithm

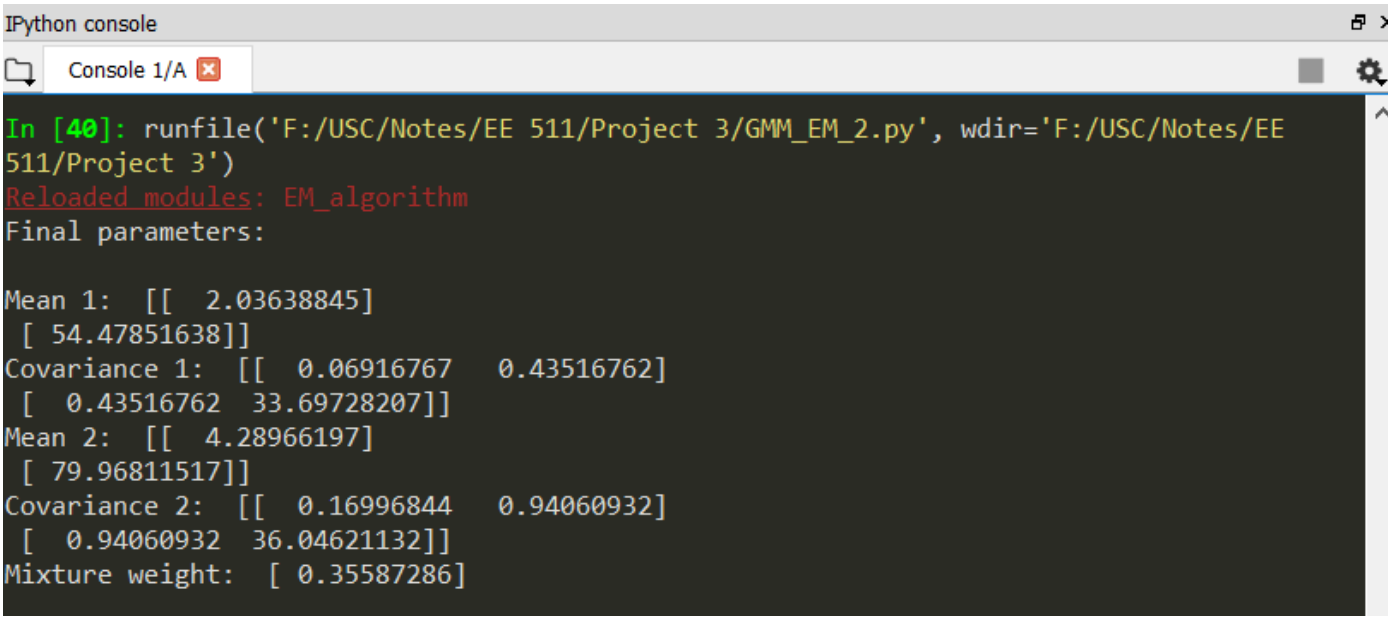
D. Clustering the 'Old_Faithful' data using EM algorithm:



(a) The 'Old_Faithful' data clustered using EM



(b) The 'Old_Faithful' data clustered using EM (contour plot)



(c) Cluster parameters determined by the EM algorithm

3. Problem 3

[Clusters of Text]

Download the "nips-87-92" data set from blackboard. This contains a *bag-of-words* data set for NIPS papers from 1987-1992. Columns in this bag-of-words model represent the (scaled) number of times a specified word appears in the different documents. The first column specifies a document id for each paper. Each row has over 11000 dimensions. So, scatterplots are not useful.

- Run your k-means clustering method to cluster the row vectors of the data set. Try different values of k and pick the best value

- Report your clusters by listing the document ids for each cluster.

Approach: This problem can be considered as an example of *Clustering Text Documents using k-means.* This can also be interpreted as a *Latent Semantic Analysis* example. *Latent Semantic Analysis* is a natural language processing technique wherein relationships are analysed between a set of documents and the terms they contain. LSA assumes that words that are similar in meaning will occur in similar pieces of texts.

The input data consists of over 700 documents and each document is represented by around 11000 features. Each feature represents the normalized frequency of a particular word in that document.

Since the dataset is very sparse and high dimensional, I have used Singular Value Decomposition to extract 100 new features from the data. Then I have used Mini-batch k-means and classical k-means to cluster the data for different number of clusters.

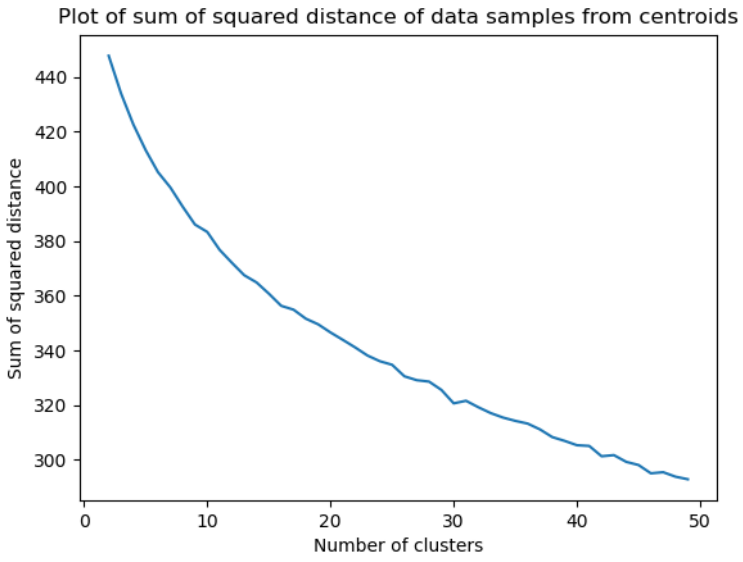
Furthermore, I have calculated *silhouette score* for each of each of the different number of clusters. The Silhouette Coefficient is calculated using the mean intra-cluster distance **(a)** and the mean nearest-cluster distance (**b)** for each sample. The Silhouette Coefficient for a sample is **(b – a)/ max(a , b)**.

Code:

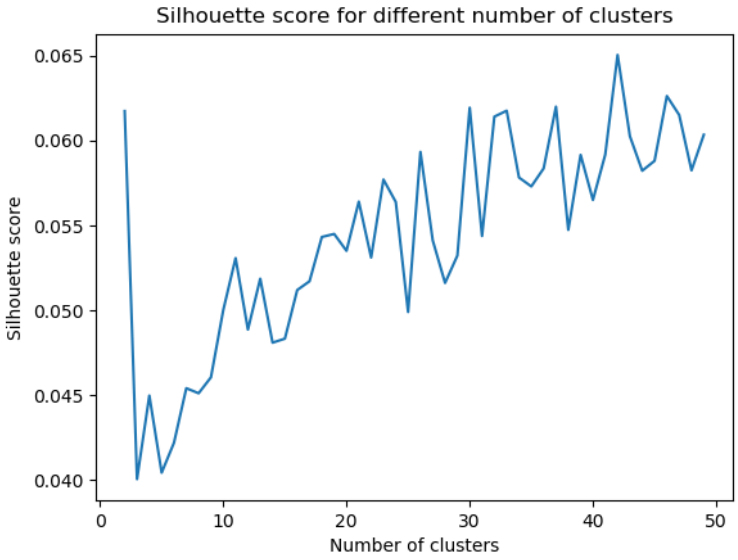
```
from sklearn.cluster import MiniBatchKMeans,KMeans
from sklearn.metrics import silhouette_score
import matplotlib.pyplot as plt
import numpy as np
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import Normalizer
from sklearn.decomposition import TruncatedSVD

dataset = np.genfromtxt("nips-87-92.csv", delimiter = ",")
data = dataset[1:, 2:]
data_label = dataset[1:, 1]

svd = TruncatedSVD(n_components = 100)
 #For latent semantic analysis, value of 100 is recommended
normalizer = Normalizer(copy=False)
lsa = make_pipeline(svd, normalizer)
data = lsa.fit_transform(data)

dist = []
x = []
silhouette_avg = []
```

```
for i in range(2, 50):

    #kmCluster = KMeans(n_clusters= i, init='k-means++', max_iter=500).fit(data)
    kmCluster = MiniBatchKMeans(n_clusters = i, init='k-means++', n_init=1,
                        batch_size=100, compute_labels = True).fit(data)
    cluster_labels = kmCluster.labels_
    dist.append(kmCluster.inertia_)
    x.append(i)
    temp = silhouette_score(data, cluster_labels, metric = 'euclidean')
    silhouette_avg.append(temp)
    print("For n_clusters =", i,
      "The average silhouette_score is :", temp)

silhouette_avg = np.asarray(silhouette_avg)
index = np.argmax(silhouette_avg)
print("The best silhouette score is: ",silhouette_avg[index], " for n_cluster :",
index + 2)
i = range(2, 50)
plt.figure(1)
plt.plot(i, silhouette_avg)
plt.title("Silhouette score(cosine distance metric) for different number of clusters")
plt.ylabel("Silhouette score")
plt.ylabel("Number of clusters")

plt.figure(2)
plt.plot(x, dist)
plt.title("Plot of sum of squared distance od data samples from centroids")
plt.ylabel("Sum of squared distance")
plt.ylabel("Number of clusters")
```

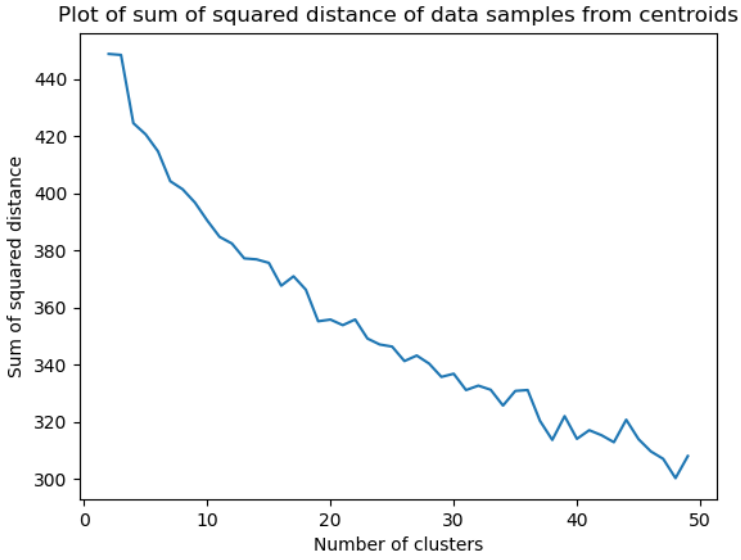
Result:



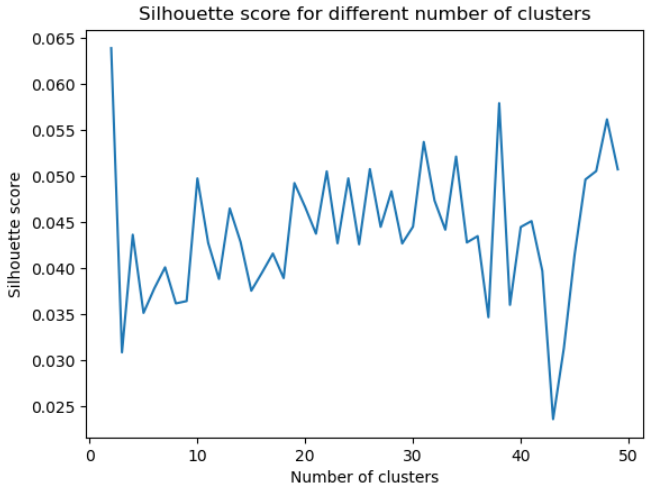
(a) Plot of sum of squared distance for *k-means algorithm*



(b) Plot of the silhouette score for *k-means algorithm*



(c) Plot of sum of squared distance for *Mini Batch k-means algorithm*



(d) Plot of the silhouette score for *Mini Batch k-means algorithm*

A silhouette score of 1 or close to 1 indicates very good clustering. A score of 0 or close to 0 indicates overlapping between clusters and a score of -1 or close to -1 indicates very bad clustering. The silhouette score that I got for both the *k-means algorithm* and *Mini-batch k-means algorithm* for different cluster numbers was always between 0 and 0.1, even after applying SVD Thus, it can be inferred form the scores that the dataset cannot be clustered perfectly and an overlapping will always be present between clusters. Furthermore, we could attribute this to the fact that a document might contain topics and text which could enable it to be a part of two or more clusters. Thus, there is no best value for the number of clusters

I tried clustering the data into 2 clusters. Some of the documents that were clustered together were

Cluster 1: 1992_48 , 1989_44 , 1989_15 ,1987_10 , 1989_44 , 1987_86 , 1991_9 , 1987_51 , 1988_54 1988_52 , 1987_42

Cluster 2: 1987_38 , 1988_76 , 1987_65 , 1991_89 , 1990_37 , 1991_133 , 1988_19 , 1991_14 , 1992_10 , 1991_69 , 1988_29 , 1992_21 , 1992_56 , 1990_21 , 1988_87 , 1991_120 , 1992_73 , 1991_136 , 1992_62