Assignment 2

Prakhar Dogra

1. **Generate 500 data points drawn from each of 3 (three) Gaussians: N1 (1, 0.1), N2 (1.5., 0.1) and N3 (2, 0.2) whose drawing probability on each iteration are P (1) = 0.25, P(2) = 0.50, and P(3) = 0.25.**

500 Data points are generated using the following code snippet.

```
size = 500
set1 = np.random.normal(loc = 1, scale = 0.1, size = size)
set2 = np.random.normal(loc = 1.5, scale = 0.1, size = size)
set3 = np.random.normal(loc = 2, scale = 0.2, size = size)

p1 = 0.25
p2 = 0.5
p3 = 0.25

dset = np.array(random.sample(list(set1), int(p1*size)) +
random.sample(list(set2), int(p2*size)) +
random.sample(list(set3), int(p3*size)))
```

2. **Derive the Gaussian Mixture Model (GMM) for data generated using 3 (three) Gaussians. And seeds of your choice for initialization.**

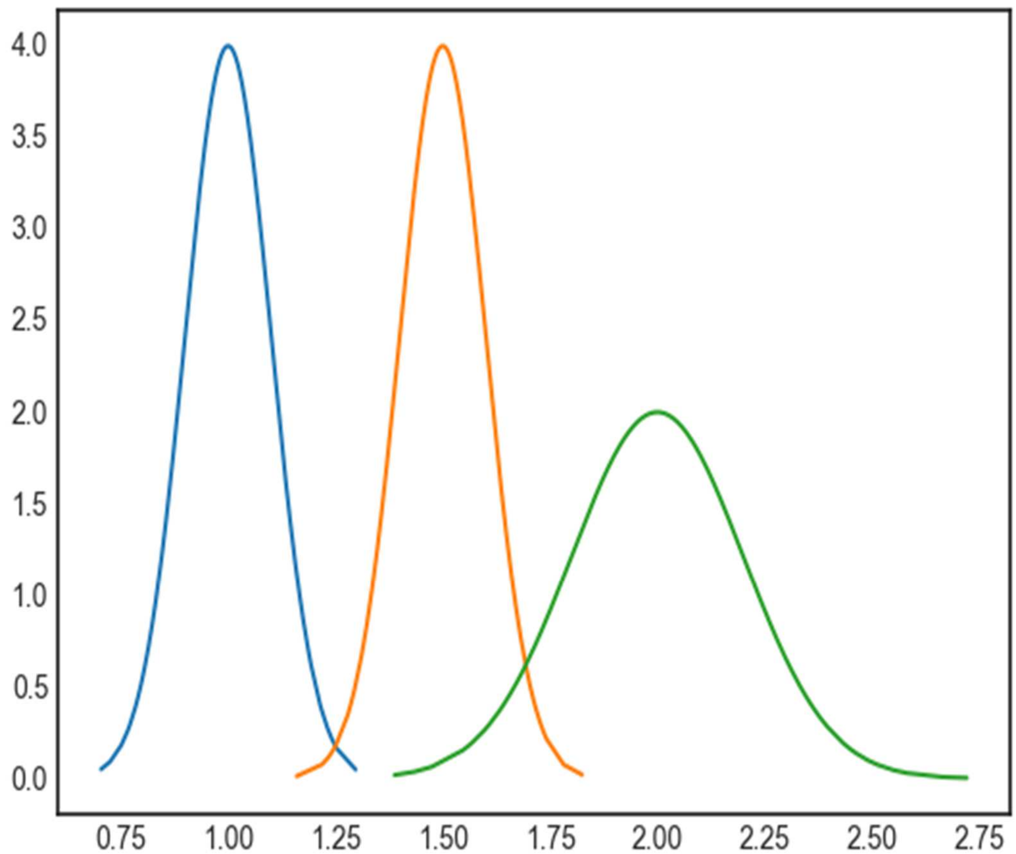
We derive the Gaussian mixture model of the given three gaussians with the help of Expectation Maximization algorithm. A Gaussian Mixture Model is represented as:

$$p(x) = \sum_{k=1}^{K} \pi_k N(x|\mu_k, \Sigma_k)$$

where p(x) is the probability density function. Above stated equation represents a mixture model consisting of K-Gaussians. For the purpose of our assignment, I have taken K =3.

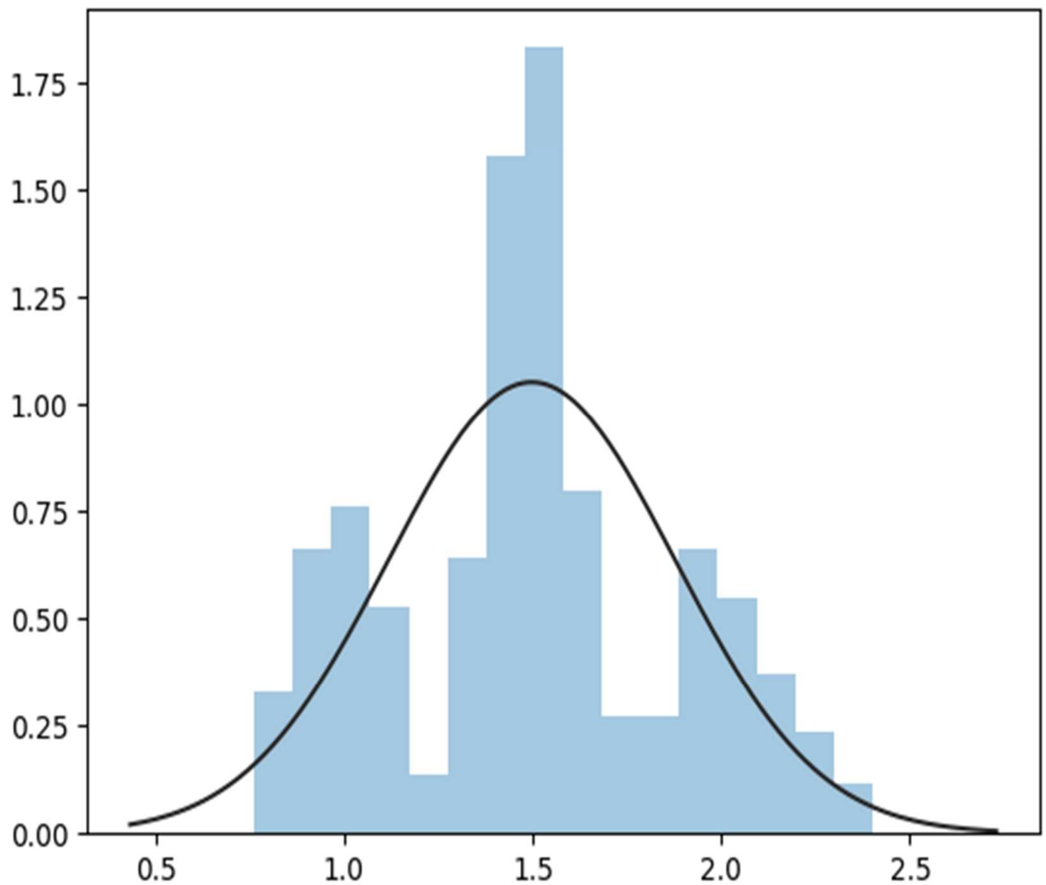
3a. Draw the graphs of
(a) original Gaussians;
(b) data drawn (500 points); and
(c) GMM.

(a)



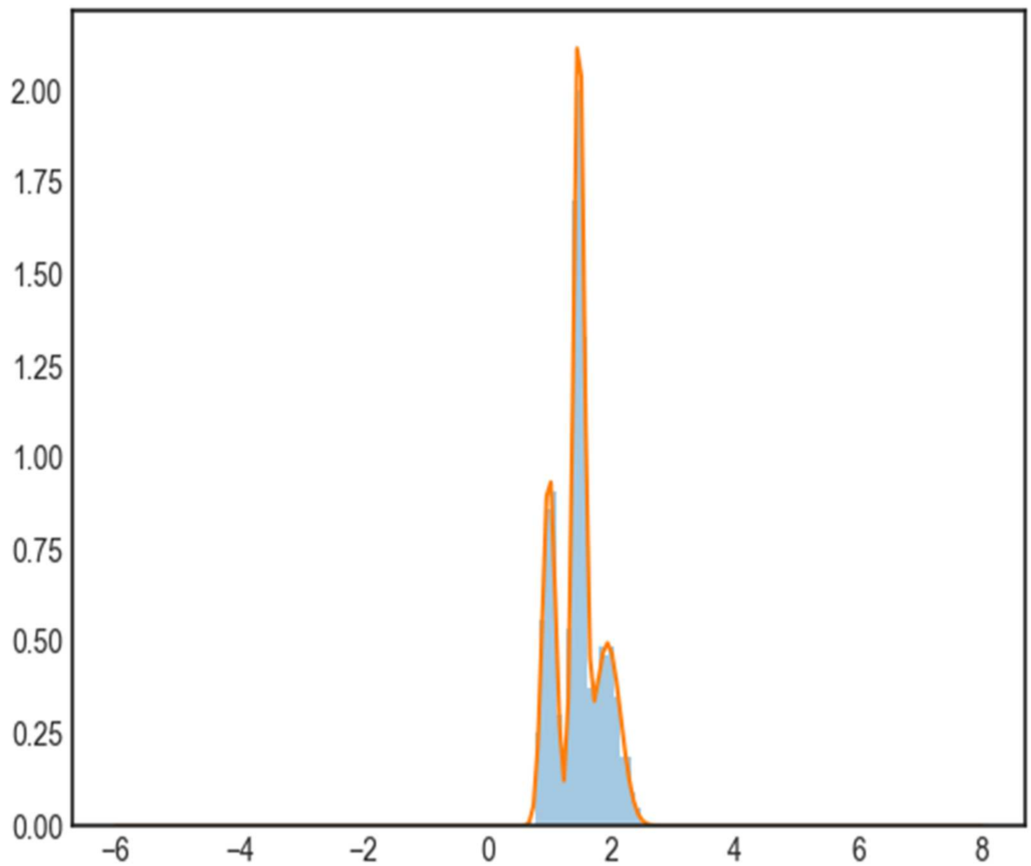
Graph 1: Graph of original gaussians (data points produced in 1.)

(b)



Graph 2: Graph of the 500 data points drawn from the original gaussians

(c)



Graph 3: Graph of the Gaussian Mixture Model

3b. Table comparing the true and estimated values for the parameters (mean, variance, mixing coefficients) describing the Gaussians.

Parameters	True Value	Estimated Value
Mean of first gaussian	1.0	0.993159
Mean of second gaussian	1.5	1.49083
Mean of third gaussian	2.0	1.99197
Standard Deviation of first gaussian	0.1	0.0983373
Standard Deviation of second gaussian	0.1	0.104083
Standard Deviation of third gaussian	0.2	0.212849
Mixing coefficient of first gaussian	0.25	0.254304803233
Mixing coefficient of second gaussian	0.5	0.480866737919
Mixing coefficient of third gaussian	0.25	0.264828458848

3c. How long it takes for EM to converge, i.e., stopping criteria and number of steps, and the log – likelihood (under the IID assumption) at convergence.

It takes considerable number of random restarts for the Expectation Maximization algorithm to converge. For the purpose for this assignment, I have taken 500 random restarts to find the mixture model with the best log likelihood.

For each such random restart, there are 40 iterations for the Expectation Maximization algorithm to find the maximum log likelihood.

3d. Accuracy for the points generated against derived GMM.

Accuracy for the points generated against derived GMM is calculated as 97.2%

REPEAT above for second scenario using new variances of 0.3, 0.4, and 0.3 for the three Gaussians. COMPARE and DISCUSS results for the two scenarios.

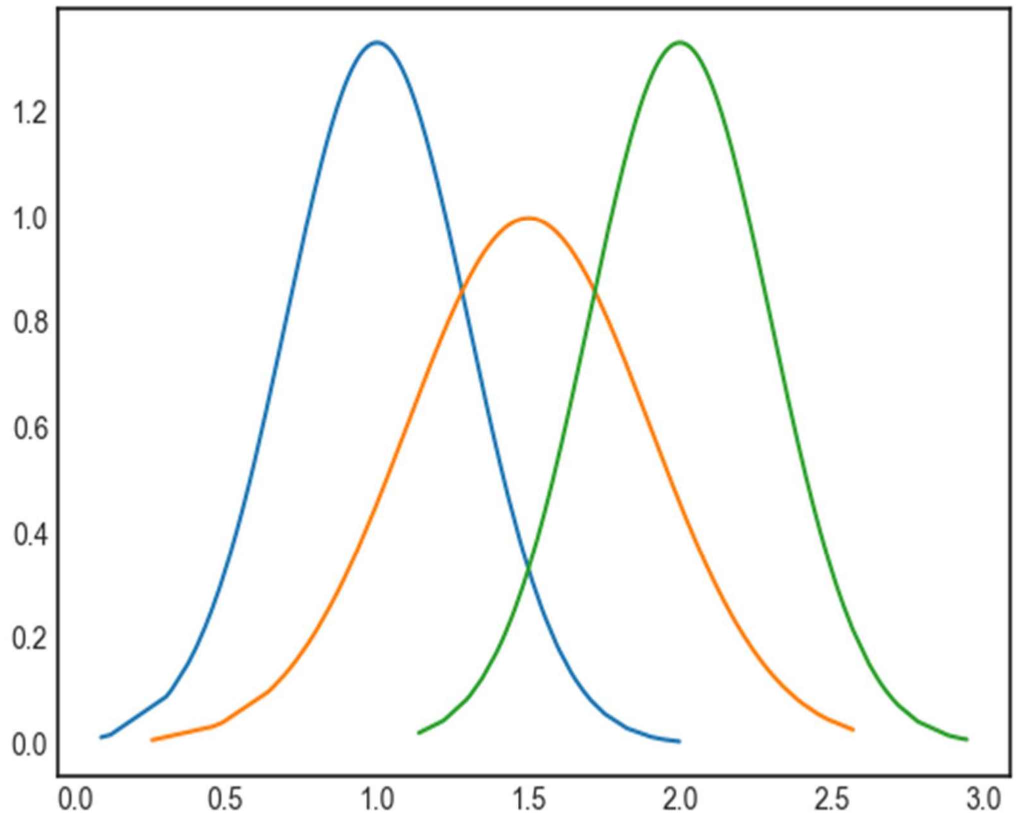
500 Data points are generated using the following code snippet.

```
size = 500

p1 = 0.25
p2 = 0.5
p3 = 0.25


set4 = np.random.normal(loc = 1, scale = 0.3, size = size)
set5 = np.random.normal(loc = 1.5, scale = 0.4, size = size)
set6 = np.random.normal(loc = 2, scale = 0.3, size = size)

dset2 = np.array(random.sample(list(set4), int(p1*size)) +
random.sample(list(set5), int(p2*size)) +
random.sample(list(set6), int(p3*size)))
```

3a. Draw the graphs of

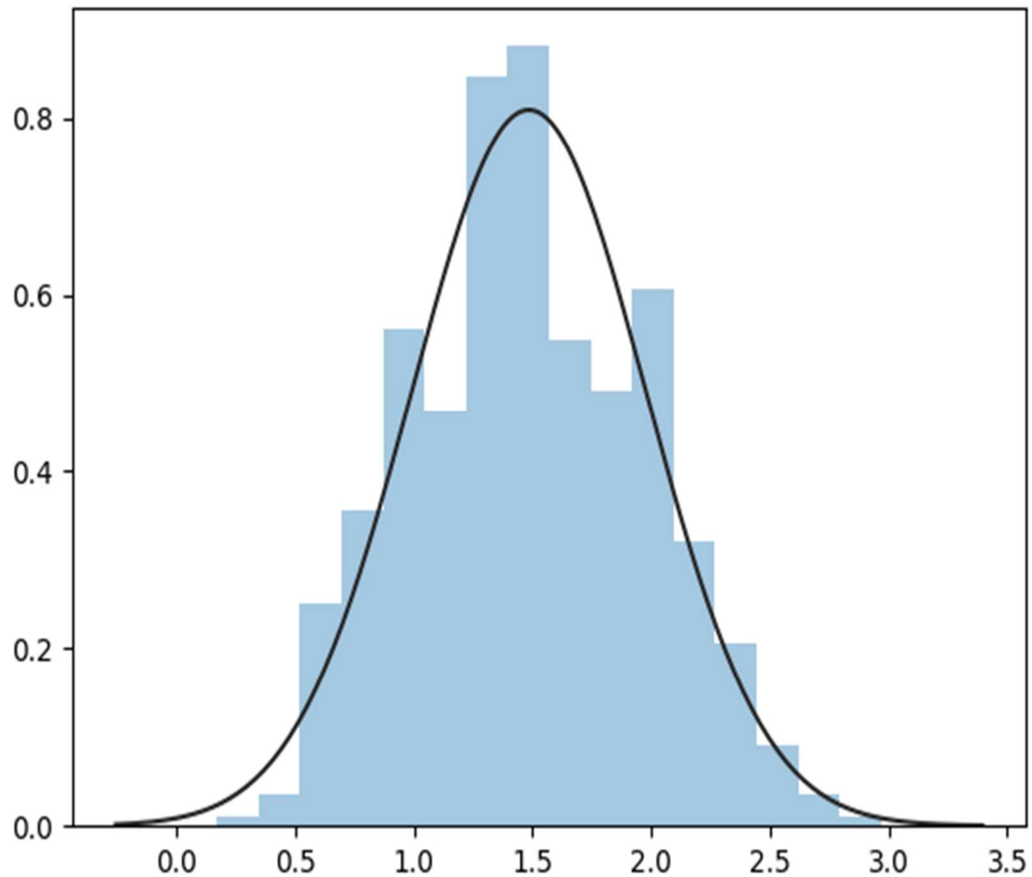
- **(a) original Gaussians;**
- **(b) data drawn (500 points); and**
- **(c) GMM.**

(a)



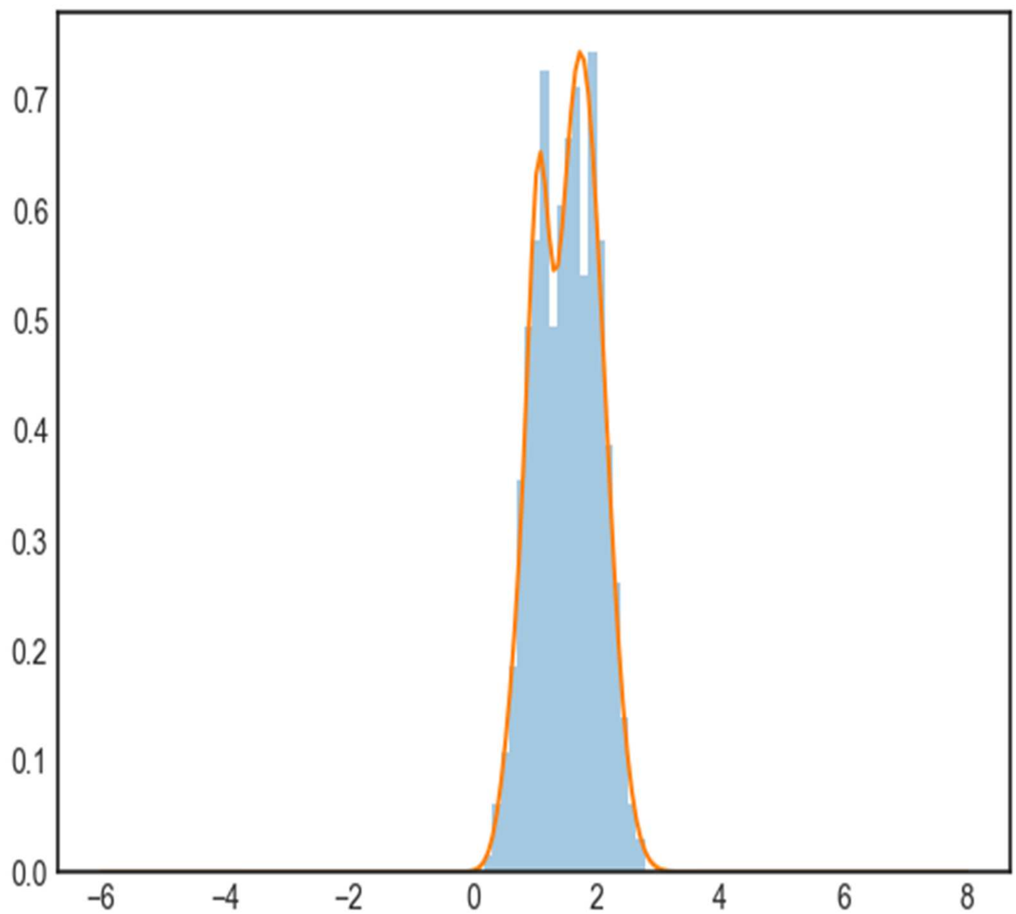
Graph 4: Graph of new gaussians (data points produced from above code snippet)

(b)



Graph 5: Graph of the 500 data points drawn from the original gaussians

(c)



Graph 6: Graph of the Gaussian Mixture Model

3b. Table comparing the true and estimated values for the parameters (mean, variance, mixing coefficients) describing the Gaussians.

Parameters	True Value	Estimated Value
Mean of first gaussian	1.0	1.14944
Mean of second gaussian	1.5	1.74943
Mean of third gaussian	2.0	2.15552
Standard Deviation of first gaussian	0.1	0.305958
Standard Deviation of second gaussian	0.1	0.171389
Standard Deviation of third gaussian	0.2	0.239127
Mixing coefficient of first gaussian	0.25	0.274547386569
Mixing coefficient of second gaussian	0.5	0.531130103572
Mixing coefficient of third gaussian	0.25	0.19432250986

3c. How long it takes for EM to converge, i.e., stopping criteria and number of steps, and the log – likelihood (under the IID assumption) at convergence.

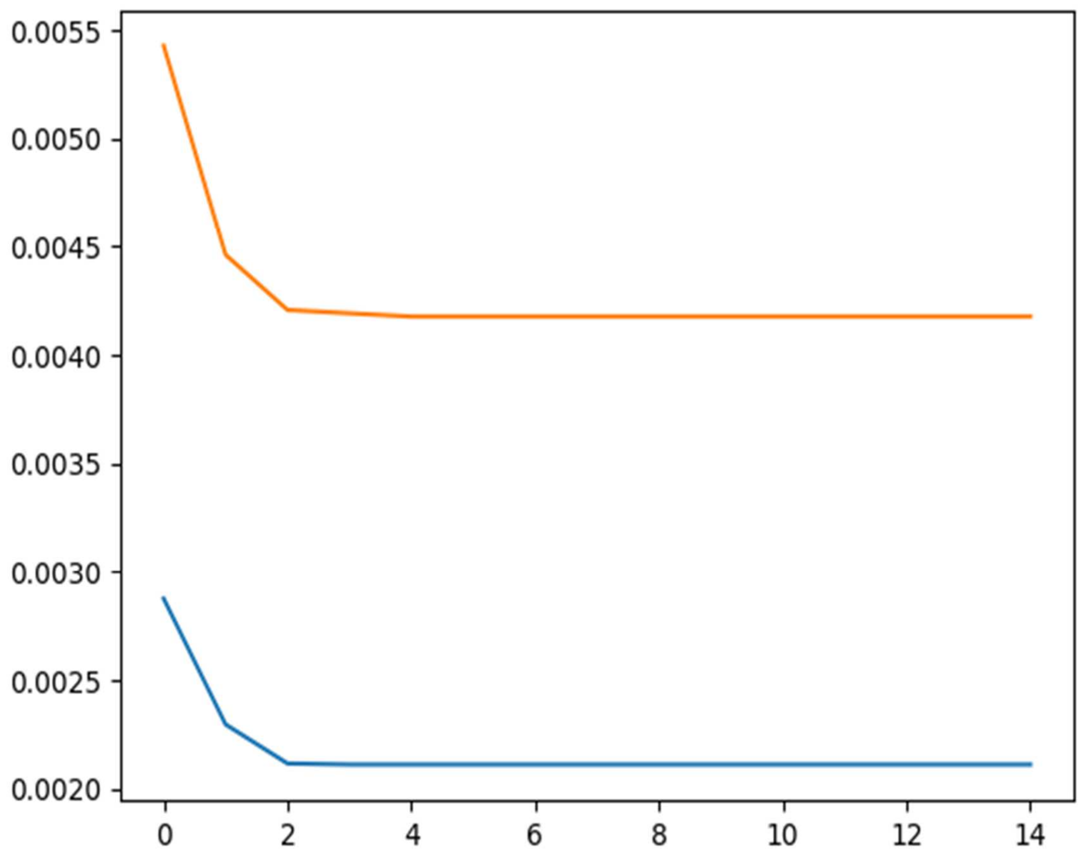
It takes considerable number of random restarts for the Expectation Maximization algorithm to converge. For the purpose for this assignment, I have taken 500 random restarts to find the mixture model with the best log likelihood.

For each such random restart, there are 40 iterations for the Expectation Maximization algorithm to find the maximum log likelihood.

3d. Accuracy for the points generated against derived GMM.

Accuracy for the points generated against derived GMM is calculated as 98.0%

4. Use K-Means to cluster the 500 points generated above under both scenarios and graph the mean – square error against step iteration. Indicate stopping criteria, and accuracy upon convergence. Estimate the normal distributions corresponding to each cluster using maximum likelihood. Assume K = 3.



Graph 7: Graph representing the Mean Square Error of each iteration for both sets of data points

Stopping criteria for the K-Means is the number of iterations. For the purpose of this assignment, we have taken the number of iterations to be 15. We can take a larger number but the fact that K-Means converges much before the maximum number of iterations.

Dataset	Dataset 1	Dataset 2
Accuracy	96%	66.6%

5. Compare performance against execution time for EM and K – Means.

Following times have been compared for the first dataset:

EM	**K-Means**
96%	97.2%

6. Implement Fuzzy ("soft") Clustering (e.g., Fuzzy C-Means Clustering; choose C = 3) on the original data and compare the results (in terms of accuracy and execution time) to EM and K – Means.

Results	**Fuzzy C-Means**	**K-Means**	**EM**
Accuracy	95.8%	96%	97.2%
Execution Time	3.12400007248 sec	0.430999994278 sec	0.184000015259 sec

7. Use K – Means to cluster the 500 point generated above and search for the optimal number of clusters 'K'. Use the silhouette to choose K.

K	**Silhouette Coefficient**
2	0.59965418025332318
3	0.72116561132548684
4	0.60761774214890096
5	0.60377937310399743
6	0.57040805266395556
7	0.56905640091609455
8	0.5739073323818239
9	0.56646900441751402
10	0.56383833990104593

From the above table it is very clear that for K=3, we get the maximum silhouette coefficient, making it the optimal value for K.

Source Code:

Gaussian Mixture Model and EM

```
import numpy as np
import matplotlib.pyplot as plt
import random
import seaborn as sns
sns.set_style("white")
from scipy.stats import norm
import time
from math import sqrt, log, exp, pi
from random import uniform

size = 500

set1 = np.random.normal(loc = 1, scale = 0.1, size = size)
set2 = np.random.normal(loc = 1.5, scale = 0.1, size = size)
set3 = np.random.normal(loc = 2, scale = 0.2, size = size)

p1 = 0.25
p2 = 0.5
p3 = 0.25

dset = np.array(random.sample(list(set1), int(p1*size)) +
random.sample(list(set2), int(p2*size)) + random.sample(list(set3),
int(p3*size)))

set4 = np.random.normal(loc = 1, scale = 0.3, size = size)
set5 = np.random.normal(loc = 1.5, scale = 0.4, size = size)
set6 = np.random.normal(loc = 2, scale = 0.3, size = size)
dset2 = np.array(random.sample(list(set4), int(p1*size)) +
random.sample(list(set5), int(p2*size)) + random.sample(list(set6),
int(p3*size)))

set4.sort()
plt.plot(set4, norm.pdf(set4,1,0.3))

set5.sort()
plt.plot(set5, norm.pdf(set5,1.5,0.4))

set6.sort()
plt.plot(set6, norm.pdf(set6,2,0.3))
plt.show()

sns.distplot(dset, fit=norm, kde=False)
plt.show()

data = dset2

class Gaussian:
    def __init__(self, mu, sigma):
        self.mu = mu
        self.sigma = sigma
```

```python
    def pdf(self, datum):
        u = (datum - self.mu) / abs(self.sigma)
        y = (1 / (sqrt(2 * pi) * abs(self.sigma))) * exp(-u * u / 2)
        return y

    def __repr__(self):
        return 'Gaussian({0:4.6}, {1:4.6})'.format(self.mu, self.sigma)

best_single = Gaussian(np.mean(data), np.std(data))
x = np.linspace(-6, 8, 200)
g_single = stats.norm(best_single.mu, best_single.sigma).pdf(x)

class GaussianMixture:
    def __init__(self, data, mu_min=min(data), mu_max=max(data),
sigma_min=.1, sigma_max=1, mix1=.25, mix2=.5):
        self.data = data
        self.one = Gaussian(uniform(mu_min, mu_max),
                            uniform(sigma_min, sigma_max))
        self.two = Gaussian(uniform(mu_min, mu_max),
                            uniform(sigma_min, sigma_max))
        self.three = Gaussian(uniform(mu_min, mu_max),
                            uniform(sigma_min, sigma_max))

        self.mix1 = mix1
        self.mix2 = mix2 #mix3 can be calculated as 1-(mix1+mix2)

    #calculates Expectation
    def Estep(self):
        self.loglike = 0.
        for datum in self.data:
            wp1 = self.one.pdf(datum) * (self.mix1)
            wp2 = self.two.pdf(datum) * (self.mix2)
            wp3 = self.three.pdf(datum) * (1 - self.mix1 - self.mix2)
            den = wp1 + wp2 + wp3
            wp1 /= den
            wp2 /= den
            wp3 /= den
            self.loglike += log(wp1 + wp2 + wp3)
            yield (wp1, wp2, wp3)

    #performs Maximization
    def Mstep(self, weights):
        (left, mid, rigt) = zip(*weights)
        one_den = sum(left)
        two_den = sum(mid)
        three_den = sum(rigt)
        self.one.mu = sum(w * d / one_den for (w, d) in zip(left, data))
        self.two.mu = sum(w * d / two_den for (w, d) in zip(mid, data))
        self.three.mu = sum(w * d / three_den for (w, d) in zip(rigt, data))
        self.one.sigma = sqrt(sum(w * ((d - self.one.mu) ** 2)
                                  for (w, d) in zip(left, data)) / one_den)
        self.two.sigma = sqrt(sum(w * ((d - self.two.mu) ** 2)
                                  for (w, d) in zip(mid, data)) / two_den)
        self.three.sigma = sqrt(sum(w * ((d - self.three.mu) ** 2)
                                  for (w, d) in zip(rigt, data)) / three_den)
        self.mix1 = one_den / len(data)
```

```
        self.mix2 = two_den / len(data)

    def iterate(self, N=1, verbose=False):
        mix.Mstep(mix.Estep())

    def pdf(self, x):
        return (self.mix1) * self.one.pdf(x) + (self.mix2) * self.two.pdf(x)
+ (1 - self.mix1 - self.mix2) * self.three.pdf(x)

    def __repr__(self):
        return 'GaussianMixture({0}, {1}, {2}, {3}, {4},
{5})'.format(self.one, self.two, self.three, self.mix1, self.mix2, 1 -
self.mix1 - self.mix2)

    def __str__(self):
        return 'Mixture: {0}, {1}, {2}, {3}, {4}, {5})'.format(self.one,
self.two, self.three, self.mix1, self.mix2, 1 - self.mix1 - self.mix2)


start_time = time.time()
n_iterations = 5
best_mix = None
best_loglike = float('-inf')
mix = GaussianMixture(data)

for _ in range(n_iterations):
    mix.iterate(verbose=True)
    if mix.loglike > best_loglike:
        best_loglike = mix.loglike
        best_mix = mix

n_iterations = 40
n_random_restarts = 500
best_mix = None
best_loglike = float('-inf')

for _ in range(n_random_restarts):
    mix = GaussianMixture(data)
    for _ in range(n_iterations):
        mix.iterate()
        if mix.loglike > best_loglike:
            best_loglike = mix.loglike
            best_mix = mix

print (time.time() - start_time)
print (best_loglike)
sns.distplot(data, bins=20, kde=False, norm_hist=True)
g_both = [best_mix.pdf(e) for e in x]
plt.plot(x, g_both, label='gaussian mixture');
plt.show()
print (best_mix)
```

K-Means

```
import numpy as np
from sklearn import mixture
import matplotlib.pyplot as plt
import random
from random import sample
from math import sqrt
from numpy import mean
import copy
from sklearn import metrics
import time
import seaborn as sns
from scipy.stats import norm

total_distances = []

size = 500

set1 = np.random.normal(loc = 1, scale = 0.1, size = size)
set2 = np.random.normal(loc = 1.5, scale = 0.1, size = size)
set3 = np.random.normal(loc = 2, scale = 0.2, size = size)

p1 = 0.25
p2 = 0.5
p3 = 0.25

dset = np.array(random.sample(list(set1), int(p1*size)) +
random.sample(list(set2), int(p2*size)) + random.sample(list(set3),
int(p3*size)))

set4 = np.random.normal(loc = 1, scale = 0.3, size = size)
set5 = np.random.normal(loc = 1.5, scale = 0.4, size = size)
set6 = np.random.normal(loc = 2, scale = 0.3, size = size)
dset2 = np.array(random.sample(list(set4), int(p1*size)) +
random.sample(list(set5), int(p2*size)) + random.sample(list(set6),
int(p3*size)))

start_time = time.time()

def initialize_centers(df, k):
    random_indices = sample(range(size), k)
    centers = []
    for id in random_indices:
        centers.append(df[id])
    print("Random Indices : " + str(random_indices))
    return centers

def compute_center(df, k, cluster_labels):
    cluster_centers = list()
    data_points = list()
    for i in range(k):
        for idx, val in enumerate(cluster_labels):
            if val == i:
```

```
                data_points.append([df[idx]])
        cluster_centers.append(map(mean, zip(*data_points)))
    return cluster_centers

def euclidean_distance(x, y):
    summ = 0
    for i in range(len(x)):
        term = (x[i] - y[i]) ** 2
        summ += term
    return sqrt(summ)

def assign_cluster(df, cluster_centers):
    cluster_assigned = list()
    for i in range(size):
        distances = []
        distances2 = []
        for center in cluster_centers:
            distance = euclidean_distance([df[i]], [center])
            distance2 = distance ** 2
            distances.append(distance)
            distances2.append(distance2)
        total_distance = sum(distances2)/size
        min_dist, idx = min((val, idx) for (idx, val) in
enumerate(distances))
        cluster_assigned.append(idx)
    total_distances.append(total_distance)
    return cluster_assigned


def kmeans(df, k):
    cluster_centers = initialize_centers(df, k)
    curr = 0
    while curr < MAX_ITER:
        cluster_labels = assign_cluster(df, cluster_centers)
        cluster_centers = compute_center(df, k, cluster_labels)
        curr += 1
    return cluster_labels, cluster_centers


k = 3
MAX_ITER = 15

labels, centers = kmeans(dset, k)
print (time.time() - start_time)

plt.plot(range(MAX_ITER), total_distances)
plt.show()
```

Fuzzy C-Means

```
import numpy as np
from sklearn import mixture
import matplotlib.pyplot as plt
import random
from random import sample
from math import sqrt
from numpy import mean
import operator
import math
from sklearn import metrics
import time

size = 500

set1 = np.random.normal(loc = 1, scale = 0.1, size = size)
set2 = np.random.normal(loc = 1.5, scale = 0.1, size = size)
set3 = np.random.normal(loc = 2, scale = 0.2, size = size)

p1 = 0.25
p2 = 0.5
p3 = 0.25

df = np.array(random.sample(list(set1), int(p1*size)) +
random.sample(list(set2), int(p2*size)) + random.sample(list(set3),
int(p3*size)))
df_labels = np.array([0]*int(p1*size) + [1]*int(p2*size) + [2]*int(p3*size))
#true cluster labels

set4 = np.random.normal(loc = 1, scale = 0.3, size = size)
set5 = np.random.normal(loc = 1.5, scale = 0.4, size = size)
set6 = np.random.normal(loc = 2, scale = 0.3, size = size)
df2 = np.array(random.sample(list(set4), int(p1*size)) +
random.sample(list(set5), int(p2*size)) + random.sample(list(set6),
int(p3*size)))

k = 3
MAX_ITER = 100
m = 2.00
start_time = time.time()

def initialize_membership_matrix():
    membership_mat = list()
    for i in range(size):
        random_num_list = [random.random() for i in range(k)]
        summation = sum(random_num_list)
        temp_list = [x/summation for x in random_num_list]
        membership_mat.append(temp_list)
    return membership_mat

def calculate_cluster_center(membership_mat):
    cluster_mem_val = zip(*membership_mat)
    cluster_centers = list()
```

```
    for j in range(k):
        x = list(cluster_mem_val[j])
        xraised = [e ** m for e in x]
        denominator = sum(xraised)
        temp_num = list()
        for i in range(size):
            data_point = [df[i]]
            prod = [xraised[i] * val for val in data_point]
            temp_num.append(prod)
        numerator = map(sum, zip(*temp_num))
        center = [z/denominator for z in numerator]
        cluster_centers.append(center)
    return cluster_centers


def update_membership_value(membership_mat, cluster_centers):
    p = float(2/(m-1))
    for i in range(size):
        x = [df[i]]
        distances = [np.linalg.norm(map(operator.sub, x, cluster_centers[j]))
for j in range(k)]
        for j in range(k):
            den = sum([math.pow(float(distances[j]/distances[c]), p) for c in
range(k)])
            membership_mat[i][j] = float(1/den)
    return membership_mat


def get_clusters(membership_mat):
    cluster_labels = list()
    for i in range(size):
        max_val, idx = max((val, idx) for (idx, val) in
enumerate(membership_mat[i]))
        cluster_labels.append(idx)
    return cluster_labels


def fuzzy_c_means_clustering():
    membership_mat = initialize_membership_matrix()
    curr = 0
    while curr <= MAX_ITER:
        cluster_centers = calculate_cluster_center(membership_mat)
        membership_mat = update_membership_value(membership_mat,
cluster_centers)
        cluster_labels = get_clusters(membership_mat)
        curr += 1
    return cluster_labels, cluster_centers

labels, centers = fuzzy_c_means_clustering()
print (time.time() - start_time)
print ("Silhouette Coefficient: %0.5f", metrics.silhouette_score(df.reshape(-
1, 1), labels))
print labels
print df_labels
print (metrics.accuracy_score(df_labels, labels))
```