ECE421 Assignment 3

Saad Jameel (1003365878)

Mohammad Mustafa Arif (1003116736)

May 25, 2019

1 K-Means

This section involved learning the clustering algorithm and investigating the effect of the number of clusters on the final cluster assignment. For K clusters, D dimensional data and N data points, the loss function was defined to be the sum of the square distance of each point to the nearest cluster center, as given by the following equation:

$$L(\mu) = \sum_{n=1}^{N} min_{k=1}^{K} \mid\mid x_n - \mu_k \mid\mid_2^2 \tag{1}$$

where μ is a $K \times D$ matrix such that the k^{th} row (written μ_k) denotes the centroid of the k^{th} cluster and x_n is the n^{th} data point represented as a $1 \times D$ row vector.

Using this definition of loss and the Adam Optimizer, the following series of investigations were performed by varying the hyperparameters. For the optimizer, the following parameters were used:

Learning Rate: 0.01
β_1: 0.9
β_2: 0.99
ϵ: 1e-5

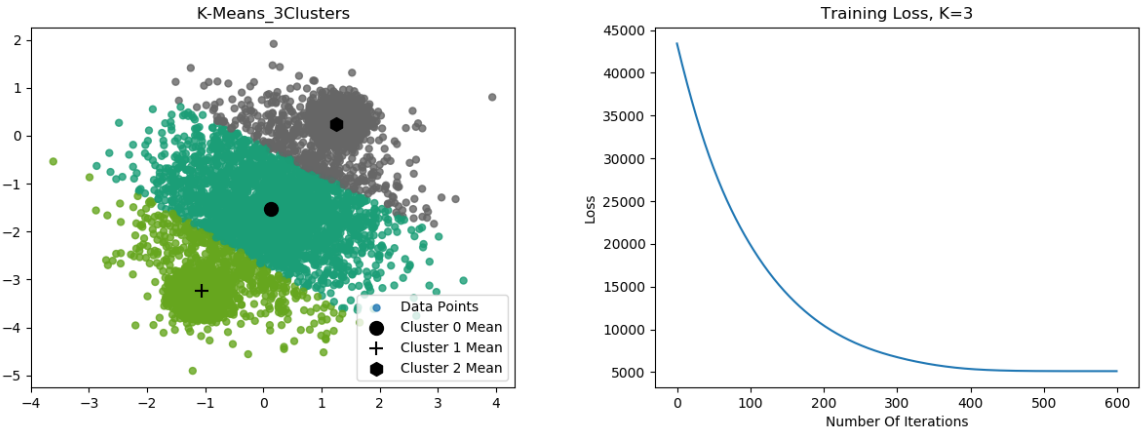
Each of the investigations was done by training the model for 600 epochs, and initially, no data was filtered out to be used as a validation set.

1.1 Learning K-Means

1.1.1 K=3

The first investigation was done with three clusters on the all the available data. A scatter plot showing the assignment of each data point, along with the cluster means is given in the figure below. Also given is the plot of training loss against the number of epochs. Here, the value of the training loss is obtained by equation (1), getting the sum of square distances between points and the nearest cluster means. The table below summarizes the optimal x,y coordinates of the final μ values for each cluster.

ClusterID	μ_x	μ_y
0	0.14	-1.52
1	-1.06	-3.24
2	1.24	0.25



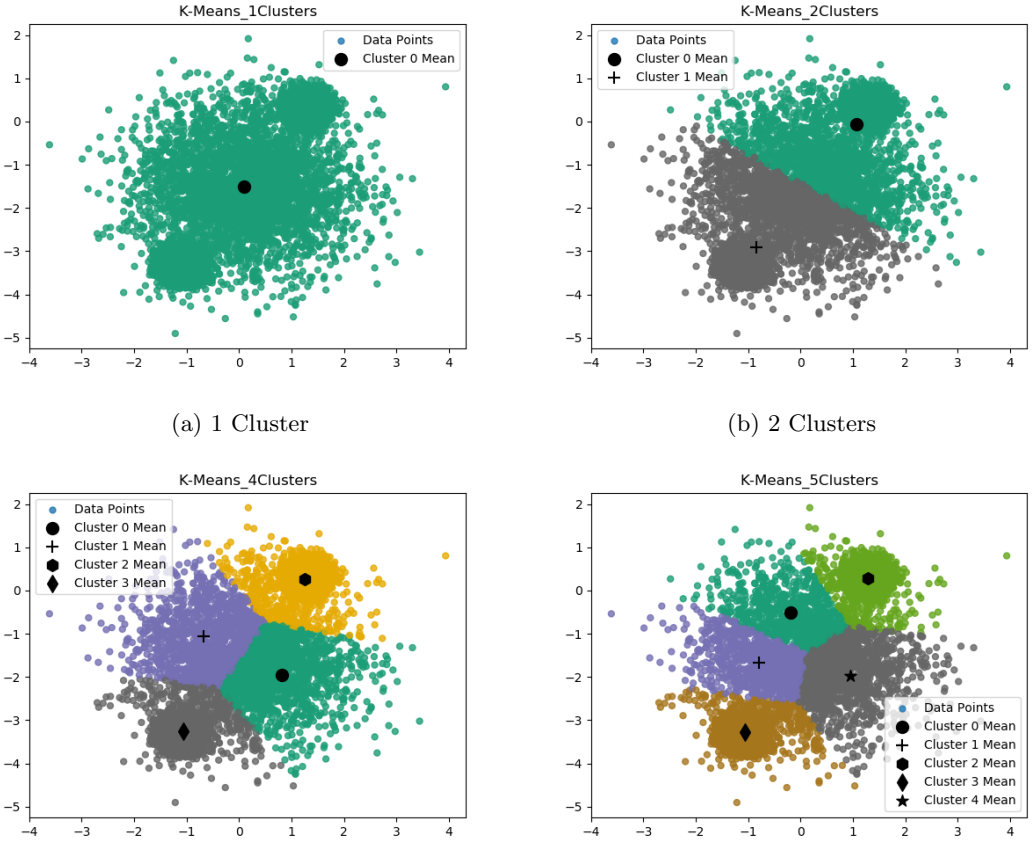
(a) Scatter Plot

(b) Training Loss vs Epochs

Figure 1: Scatter Plot and Training Loss for 3 Clusters

1.1.2 Repeating the Investigation of K=1,2,4,5

The scatter plots obtained are presented below:



(a) 1 Cluster

(b) 2 Clusters

(c) 4 Clusters

(d) 5 Clusters

Figure 2: Scatter Plots for K = 1,2,4,5

For each value of K, the percentage of points belonging to each cluster was also noted. The results are summarized in the table below:

K	Cluster0	Cluster1	Cluster2	Cluster3	Cluster4
1	100%	n/a	n/a	n/a	n/a
2	49.53%	50.47%	n/a	n/a	n/a
3	23.82%	38.21%	37.97%	n/a	n/a
4	13.53%	12.03%	37.31%	37.13%	n/a
5	7.47%	8.57%	35.89%	36.35%	11.72%

Based on the scatter plots, it appears that K = 3 is the optimal number of clusters. It seems like there's a dense cluster at the top right, a sparse one in the middle and a dense one at the bottom left. It appears however that K-means is suboptimal for this data as a lot of the data assigned to the corner clusters actually appear to be a part of the middle cluster.

1.1.3 Holding Out Validation Data

The investigation was repeated for as before but 1/3 of the available data-set was separated at the beginning and used as a validation set. Therefore, the model was trained on the other 2/3 of the available data. By clustering the training data and computing the loss on the validation set, the final training and validation losses for each value of K were obtained. Results are summarized in the table below:

K	Training Loss	Validation Loss
1	25588.99	12870.10
2	6243.33	2960.66
3	3489.18	1629.42
4	2320.15	1054.53
5	1961.30	918.95

K = 5 clusters minimizes the loss, however this is expected as more cluster centers decreases the average distance of each point to a cluster. If we evaluate only based on loss, then K = ∞ clusters would be optimal.

In reality the loss does not decrease as much from K =3 to K=4 clusters and K =4 to K =5 clusters. This means that adding more cluster centers doesn't decrease the loss as much. Thus K =3/K=4 clusters would be ideal. Visually it would be K = 3 as there only appears to be 3 clusters.

2 Mixture of Gaussians

2.1 The Gaussian Cluster Mode

2.1.1 Log Probability Density Function

The first function implemented for this section was the log-GaussPDF function that computed the logarithm of the normal distribution of the points x, given the means and variances. The function was designed with the following input arguments:

X: an N $\times$ D matrix representing the data points
μ: a $K \times D$ matrix representing the centroids for all K clusters
σ^2: a $K \times 1$ matrix representing the scalar variances for all K clusters

It was assumed that the variances in each of the D dimensions are equal. A snippet of the implemented python function is given below:

```
def log_GaussPDF(X, mu, sigma):
    # Inputs
    # X: N X D
    # mu: K X D
    # sigma: K X 1
    # log_pi: K X 1

    # Outputs:
    # log Gaussian PDF N X K

    D = X.shape[1]
    distance = distanceFunc(X, mu)
    invsigma = tf.math.reciprocal(sigma)
    invsigma = -1/2 * tf.square(invsigma)
    mat = tf.transpose(invsigma) *  distance
    logsigma = -tf.cast((tf.shape(X)[1]), dtype=tf.float64) * tf.log(sigma)

    coeff = logsigma - tf.cast((tf.shape(X)[1]/2), dtype=tf.float64) *
    tf.log(2*tf.cast(np.pi, dtype=tf.float64))

    mat = mat + tf.transpose(coeff)

    return mat
```

As seen above, this function uses distanceFunc, a function defined while computing the kmeans in section 1. A snippet of distanceFunc is shown below:

```
def distanceFunc(X, MU):
    # Inputs
    # X: is an NxD matrix (N observations and D dimensions)
    # MU: is an KxD matrix (K means and D dimensions)
    # Outputs
    # pair_dist: is the pairwise distance matrix (NxK)
    # TODO
    x2 = tf.reduce_sum(tf.square(X), 1)
    y2 = tf.reduce_sum(tf.square(MU), 1)
    x2 = tf.reshape(x2, [-1, 1])
    y2 = tf.reshape(y2, [1, -1])
    diff = x2 - 2 * X @ tf.transpose(MU) + y2
    return diff
```

2.1.2 Log Posterior Probability Function

This section involved defining the posterior log function, which takes in the logarithm of the distribution of x given cluster parameters, as defined by the log-GaussPDF function, and the logarithm of the prior distribution of the clusters. It returns the posterior distribution by using bayes rule, as shown:

$$P(Z|X) = \frac{P(X|Z) \times P(Z)}{P(X)} \tag{2}$$

where Z is a cluster variable, and X is the given data vector. In the log domain, this equation translates to:

$$log(P(Z|X)) = log(P(X|Z)) + log(P(Z)) - log(P(X)) \tag{3}$$

Using the expression in (3), the log posterior function was defined as follows:

```
def log_posterior(log_PDF, log_pi):
    # Input
    # log_PDF: log Gaussian PDF N X K
    # log_pi: K X 1

    # Outputs
    # log_post: N X K

    coeff = tf.squeeze(log_pi) + log_PDF
    return coeff - tf.expand_dims(hlp.reduce_logsumexp(coeff), 1)
```

As seen above, this function uses the given helper function reduce-logsumexp(). logsumexp() is important for avoiding numerical underflow and overflow. In this case, small probability values could underflow, so logsumexp is better than reduce sum. Here, logsumexp() performs the following operation:

$$logsumexp(z_1 ... z_N) = log(\sum_{n=1}^{N} e^{z_n}) \tag{4}$$

Thus, using logsumexp() as shown in the code snippet above yields log(P(X)) by computing:

$$log(P(X)) = log(\sum_{z} e^{log(P(X|Z)) + log(P(Z))}) \tag{5}$$

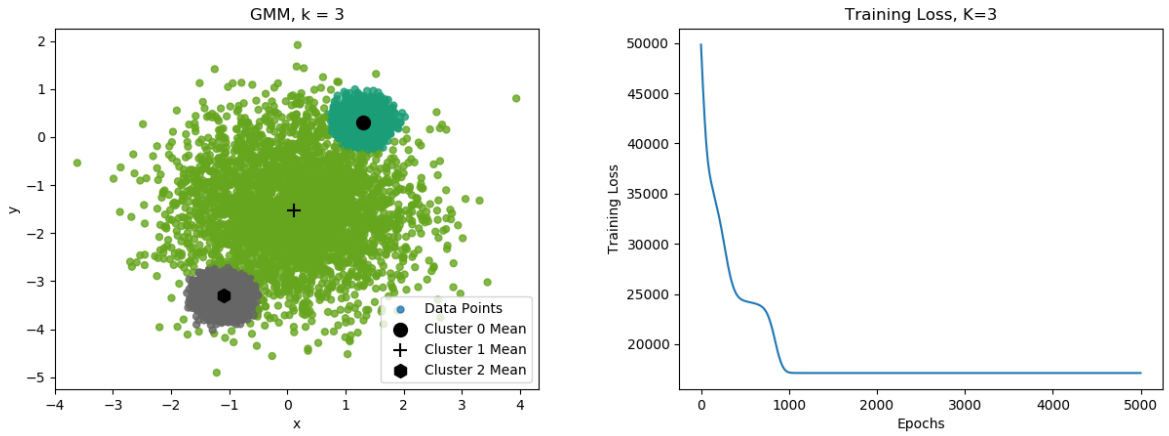
2.2 Learning the MoG

This section involved computing the maximum likelihood estimate of the model parameters μ, σ and π, where π represents the prior probability for each cluster. The loss function used was -log(P(x)), the negative log-likelihood function. In additional, the priors π_k were assumed to be unconstrained tensorflow variables. Therefore, they were normalized via the logsoftmax helper function to ensure that $\sum_k \pi_k = 1$. Details of the tensorflow implementation are given below:

```
#Defining placeholders and variables
x = tf.placeholder(name = 'x', dtype = tf.float64, shape = (None, data.shape[1]))
mu = tf.get_variable(name = 'mean', dtype=tf.float64, shape=(K, data.shape[1]),
                     initializer=tf.initializers.random_normal(seed=18786708))
stdvec = tf.get_variable(name = 'std', dtype=tf.float64, shape=(K, 1),
                     initializer=tf.initializers.random_normal(seed=366901768))
prior = tf.get_variable(name = 'pi', dtype=tf.float64, shape=(K, 1),
                     initializer=tf.initializers.random_normal(seed=1566557))
sigma = tf.exp(stdvec)
log_gauss_pdf = log_GaussPDF(x, mu, sigma)
log_prior = hlp.logsoftmax(prior) #To ensure that priors are normalized
log_post = log_posterior(log_gauss_pdf, log_prior)

#Defining loss function
temp = hlp.reduce_logsumexp(tf.squeeze(log_prior) + log_gauss_pdf)
loss = -tf.reduce_sum(temp)
```

2.2.1 K= 3: Loss, Scatter Plot and Model Parameters



(a) Scatter Plot

(b) Training Loss vs Epochs

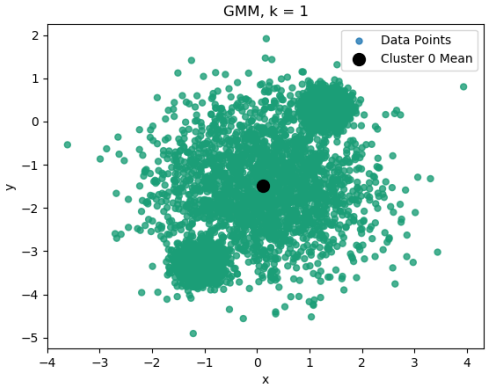
Figure 3: Scatter Plot and Training Loss for 3 Clusters

Final parameters are given below. Each row corresponds to to each cluster.

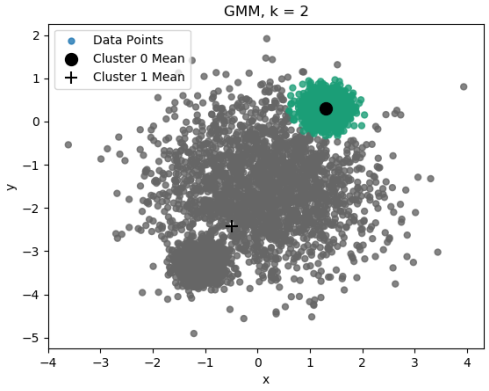
$$\text{Final } \mu = \begin{pmatrix} 1.30 & 0.31 \\ 0.106 & -1.53 \\ -1.10 & -3.31 \end{pmatrix} \quad \text{Final } \sigma = \begin{pmatrix} 0.197 \\ 0.994 \\ 0.1977 \end{pmatrix}$$

$$\text{Final } \pi = \begin{pmatrix} 0.334 \\ 0.333 \\ 0.333 \end{pmatrix}$$

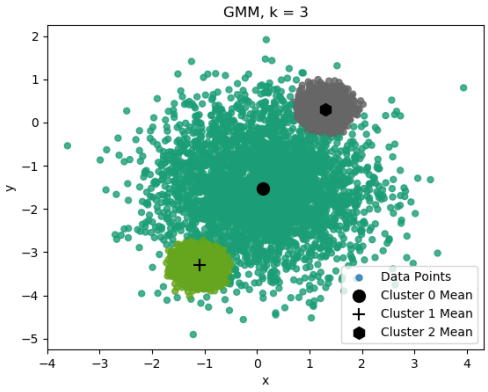
2.2.2 Holding Out Validation Data



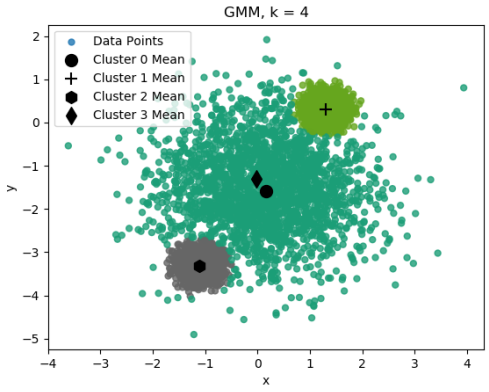
(a) 1 Cluster



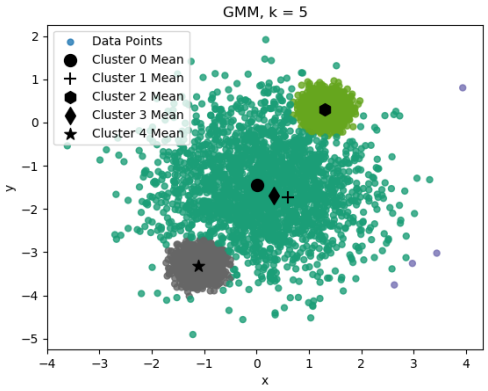
(b) 2 Clusters



(c) 3 Clusters



(d) 4 Clusters



(e) 5 Clusters

Figure 4: GMM Scatter Plots for K = 1,2,4,5

K	Training Loss	Validation Loss
1	23266.10	11652.24
2	16155.74	7987.79
3	11505.93	5629.62
4	11505.83	5629.67
5	11505.44	5630.74

As the number of clusters increase, the loss appears to decrease. 3 clusters appears to optimal as increasing the number of clusters after this point only decreases loss by a small amount. This agrees with what's visually intuitive as there only appear to be three clusters, one at the top right, one at the bottom left, and the large one in the middle.

2.2.3 Comparing GMM and K-Means for 100 Dimensional Data

Both GMM and the K-Means algorithms were run on 100 dimensional data with varying values of K. The purpose of this investigation was to find the number of clusters within the dataset by observing the loss values at different values of K. Since there is no direct way to visualize 100 dimensional data, scatter plots could not be obtained, and a qualitative analysis was not possible. Thus, the number of clusters in the dataset could not be visually determined. This was unlike the 2 dimesnional case because it was apparent from the scatter plots in the previous section that the data had 3 clusters (two dense clusters on the top right and bottom left, and a sparse one in the center). This can be verified via Figure 4.

Thus, a quantitative approach was deemed necessary. Both the algorithms for this investigation were run with a learning rate 0f 0.01 and training was done for 600 epochs. Results from this investigation are summarized in the table below:

K	K-Means Loss	GMM Loss
5	587385.33	1303322.47
10	367979.14	1091619.93
15	212825.50	860637.04
20	211265.98	860674.38
30	209504.07	859795.97

As seen in the table above, the loss value for both K-Means and GMM algorithms decreases with an increase in the value of K. This is expected since the more clusters there are, the lower the average distance will be from of an arbitrary point from a cluster center. However, after K = 15, both the loss values seem to level off because the decrease in loss is very small as K is increased from 15 to 20, and then to 30. This seems to suggest that the 100 dimensional dataset actually has 15 clusters. Beyond 15, most of the clusters are likely empty causing the loss to level off.

3 Appendices - Python Code

3.1 starter-kmeans.py

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import helper as hlp


# Distance function for K-means
def distanceFunc(X, MU):
    # Inputs
    # X: is an NxD matrix (N observations and D dimensions)
    # MU: is an KxD matrix (K means and D dimensions)
    # Outputs
    # pair_dist: is the pairwise distance matrix (NxK)
    # TODO
    # sum((x-y)^2) = sum(x^2 - 2xy + y^2)
    x2 = tf.reduce_sum(tf.square(X), 1)
    y2 = tf.reduce_sum(tf.square(MU), 1)
    x2 = tf.reshape(x2, [-1, 1])
    y2 = tf.reshape(y2, [1, -1])
    diff = x2 - 2 * X @ tf.transpose(MU) + y2
    return diff


def kmeans(epochs, clusters, step, data, val_data):
    D = data.shape[1]
    N = data.shape[0]
    K = clusters

    #Initializing Placeholders & Variables
    centers = tf.get_variable(name='MU', dtype=tf.float64, shape=(K, D),
                              initializer=tf.initializers.random_normal(seed=1))
    inputs = tf.placeholder(name='inputs', dtype=tf.float64, shape=(None, D))

    training_losses = []
    valid_losses = []
    loss = tf.reduce_sum(tf.reduce_min(distanceFunc(inputs, centers), axis=1))
    optimizer = tf.train.AdamOptimizer(learning_rate=step, beta1=0.9, beta2=0.99,
    epsilon=1e-5).minimize(loss)
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for epoch in range(epochs):
            #Why do we feed in the same data twice?

            training_loss = sess.run(loss, feed_dict={inputs: data})
            valid_loss = sess.run(loss, feed_dict={inputs: val_data})
            sess.run(optimizer, feed_dict={inputs: data})
            training_losses.append(training_loss)
            valid_losses.append(valid_loss)
            print('Training loss = {} | Validation loss = {}'.format(training_loss, valid_loss))
        centroids = sess.run(centers)
```

```
plt.plot(training_losses)
plt.xlabel("Number Of Iterations")
plt.ylabel("Loss")
plt.title("Training Loss, K={}".format(K))
plt.savefig("Loss{}Clusters".format(K))
plt.show()

#print(type(centroids))
#print(centroids.shape)
#print(data.shape)

dist = distanceFunc(data, centroids)
cluster_class = tf.argmin(dist, axis=1)
cluster_class = cluster_class.eval()

clusterID, count = np.unique(cluster_class, return_counts=True)
counts = dict(zip(clusterID, count))

print(clusterID,count)

for i in range(K):
    p = counts[i]/N * 100
    print("Percentage of points in cluster {}: {}%".format(i, p))



#print(cluster_class.shape)
#print(cluster_class)

datanp = data

x = datanp[:, 0]
y = datanp[:, 1]
centroids_x = centroids[:, 0]
centroids_y = centroids[:, 1]


plt.scatter(x, y, c=cluster_class, label='Data Points', s=25, alpha=0.8, cmap = 'Dark2')

mark = ['o', '+', 'h', 'd', '*', 'P']
for i in range(centroids_x.shape[0]):
    plt.scatter(centroids_x[i], centroids_y[i], marker=mark[i], label='Cluster {} Mean'
    .format(i), s=100, c='k')



#plt.scatter(centroids_x, centroids_y, marker='x', label='Cluster Means', c='k', s=100)
plt.legend()
plt.title("K-Means_{}Clusters".format(K))
plt.savefig("K-Means_{}Clusters".format(K))
plt.show()
```

```
if __name__ == "__main__":

    # Loading data
    data = np.load('data2D.npy')
    #data = np.load('data100D.npy')
    [num_pts, dim] = np.shape(data)

    # For Validation set
    is_valid = False
    if is_valid:
      valid_batch = int(num_pts / 3.0)
      np.random.seed(45689)
      rnd_idx = np.arange(num_pts)
      np.random.shuffle(rnd_idx)
      val_data = data[rnd_idx[:valid_batch]]
      data = data[rnd_idx[valid_batch:]]

    epochs = 600
    lr = 0.01
    K = 2

    kmeans(epochs = epochs, clusters = K, step = lr, data = data, val_data = data)
```

3.2 starter-gmm.py

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import helper as hlp


# Distance function for GMM
def distanceFunc(X, MU):
    # Inputs
    # X: is an NxD matrix (N observations and D dimensions)
    # MU: is an KxD matrix (K means and D dimensions)
    # Outputs
    # pair_dist: is the pairwise distance matrix (NxK)
    # TODO
    x2 = tf.reduce_sum(tf.square(X), 1)
    y2 = tf.reduce_sum(tf.square(MU), 1)
    x2 = tf.reshape(x2, [-1, 1])
    y2 = tf.reshape(y2, [1, -1])
    diff = x2 - 2 * X @ tf.transpose(MU) + y2
    return diff


def log_GaussPDF(X, mu, sigma):
    # Inputs
    # X: N X D
    # mu: K X D
    # sigma: K X 1
    # log_pi: K X 1

    # Outputs:
    # log Gaussian PDF N X K

    # TODO
    D = X.shape[1]

    #distance = distanceFunc(X, mu)
    #sigma = tf.transpose(sigma)
    #m = tf.cast(D, dtype=tf.float64) * tf.log(sigma*tf.sqrt(2*tf.cast(np.pi, dtype=tf.float64))) + 1/(
    #return -m
    distance = distanceFunc(X, mu)

    invsigma = tf.math.reciprocal(sigma)

    invsigma = -1/2 * tf.square(invsigma)

    mat = tf.transpose(invsigma) *  distance

    logsigma = -tf.cast((tf.shape(X)[1]), dtype=tf.float64) * tf.log(sigma)

    coeff = logsigma - tf.cast((tf.shape(X)[1]/2), dtype=tf.float64) * tf.log(2*tf.cast(np.pi,
    dtype=tf.float64))
```

```
    mat = mat + tf.transpose(coeff)

    return mat


def log_posterior(log_PDF, log_pi):
    # Input
    # log_PDF: log Gaussian PDF N X K
    # log_pi: K X 1

    # Outputs
    # log_post: N X K

    # TODO
    coeff = tf.squeeze(log_pi) + log_PDF
    return coeff - tf.expand_dims(hlp.reduce_logsumexp(coeff), 1)



if __name__ == "__main__":

    # Loading data
    data = np.load('data100D.npy')
    # data = np.load('data2D.npy')
    [num_pts, dim] = np.shape(data)
    # print("hello")

    val_data = data

    # For Validation set
    is_valid = False

    if is_valid:
        valid_batch = int(num_pts / 3.0)
        np.random.seed(45689)
        rnd_idx = np.arange(num_pts)
        np.random.shuffle(rnd_idx)
        val_data = data[rnd_idx[:valid_batch]]
        data = data[rnd_idx[valid_batch:]]

    epochs = 1000 #5000
    lr = 0.005
    K = 5

    x = tf.placeholder(name = 'x', dtype = tf.float64, shape = (None, data.shape[1]))
    mu = tf.get_variable(name = 'mean', dtype=tf.float64, shape=(K, data.shape[1]),
                         initializer=tf.initializers.random_normal(seed=18786708))

    stdvec = tf.get_variable(name = 'std', dtype=tf.float64, shape=(K, 1),
                         initializer=tf.initializers.random_normal(seed=366901768))

    prior = tf.get_variable(name = 'pi', dtype=tf.float64, shape=(K, 1),
                         initializer=tf.initializers.random_normal(seed=1566557))
```

```
sigma = tf.exp(stdvec)
log_gauss_pdf = log_GaussPDF(x, mu, sigma)
log_prior = hlp.logsoftmax(prior) #To ensure that priors are normalized
log_post = log_posterior(log_gauss_pdf, log_prior)

#Defining loss function
temp = hlp.reduce_logsumexp(tf.squeeze(log_prior) + log_gauss_pdf)
loss = -tf.reduce_sum(temp)

optimizer = tf.train.AdamOptimizer(learning_rate=lr).minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    train_loss = []

    for epoch in range(epochs):
        sess.run(optimizer, feed_dict={x: data})
        trainingloss = sess.run(loss, feed_dict={x: data})
        validloss = sess.run(loss, feed_dict={x: val_data})
        train_loss.append(trainingloss)

        print('Training loss = {} | Validation loss = {}'.format(trainingloss, validloss))

    mu_after_training = sess.run(mu)
    sigma_after_training = sess.run(sigma)
    posterior_after_training = sess.run(log_post, feed_dict={x: data})

    print("Final mu: {}".format(mu_after_training))
    print("Final stdev: {}".format(sigma_after_training))

    plt.plot(train_loss)
    plt.xlabel('Epochs')
    plt.ylabel('Training Loss')

    plt.title("Training Loss, K={}".format(K))
    plt.savefig("LossGMM{}Clusters100D".format(K))
    plt.show()

    #####CLUSTERS - CHANGE THIS

    final_mu = mu_after_training
    final_posterior = posterior_after_training

    labels = np.argmax(final_posterior, axis=1)
    unique, counts = np.unique(labels, return_counts=True)
    dictionary = dict(zip(unique, counts))


    for clusters in range(K):

        try:
            p = dictionary[clusters]* 100 / data.shape[0]
            print('% of points belonging to cluster {} is: {}% '.format(clusters, p))
```

```
            except:
                print('% of points belonging to cluster {} is: 0% '.format(clusters))



'''
        x_mu, y_mu = final_mu.T
        plt.scatter(data[:, 0], data[:, 1], c=labels,label='Data Points', s=25, alpha=0.8,cmap='Dark2')
        mark = ['o', '+', 'h', 'd', '*', 'P']
        for i in range(K):
            plt.scatter(final_mu[i, 0], final_mu[i, 1], marker=mark[i], label='Cluster {} Mean'
            .format(i),s=100, c='k')
        #plt.scatter(data[:, 0], data[:, 1], c=labels, label='data', s=10, alpha=0.8)
        #plt.scatter(final_mu[:, 0], final_mu[:, 1], cmap='r', marker='X', label='means', c='r')
        plt.xlabel('x')
        plt.ylabel('y')
        plt.title('GMM, k = {}'.format(K))
        plt.legend()
        plt.savefig("GMM_{}Clusters".format(K))
        plt.show()
'''
```