Assignment 3 — Unsupervised Learning & Probabilistic Models

Part 0: Changes and Assumptions

For gradient descent for both the K-Means and MoG I have used the following parameters
- learning rate = 0.01
- number epochs = 1000

For K-Means initializations
- MU ~ N(0, 1)

For MoG initializations
- MU ~ N(0, 1)
- sigma ~ N(0, 1)
- weights ~ N(0, 1)

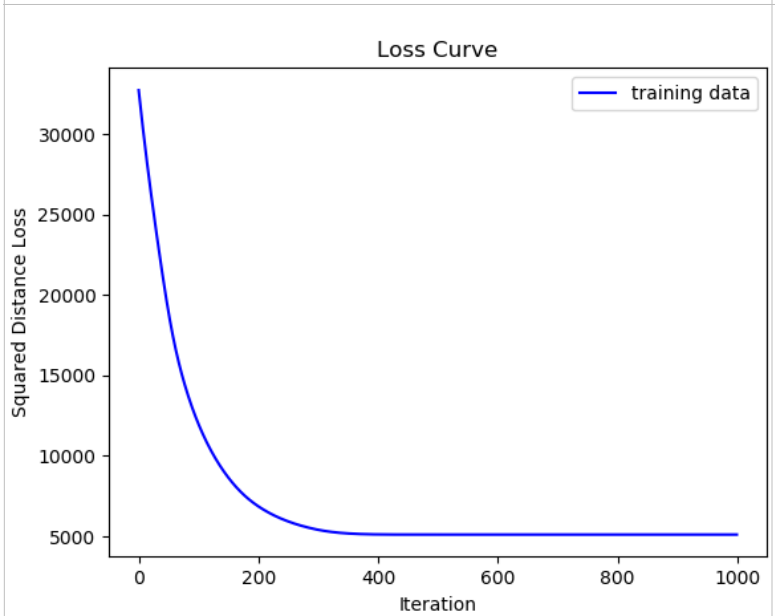
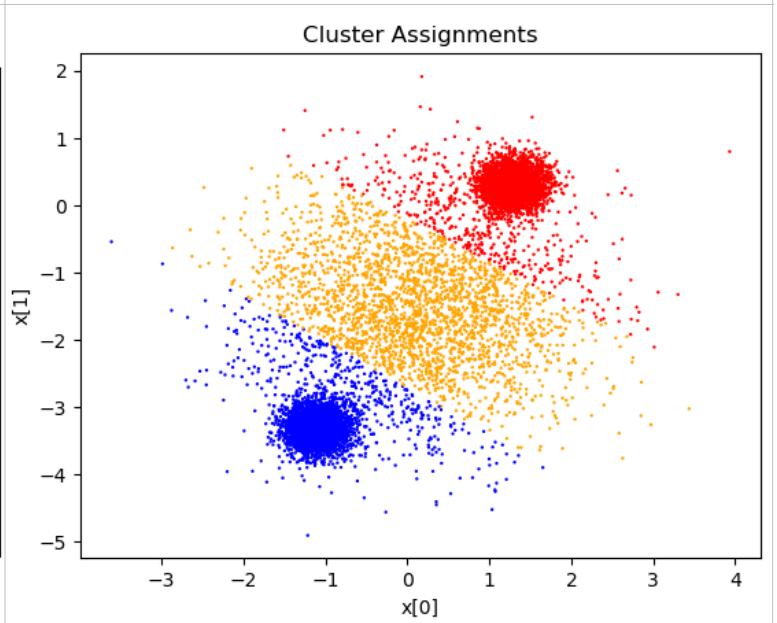
I have also changed the dimensions for the weights (π) and standard deviations (σ) from Kx1 to 1xK for convenience. This means that I have also modified the logsoftmax function in helper.py to use reduction_indices=1 instead of 0.

Part 1: K-Means

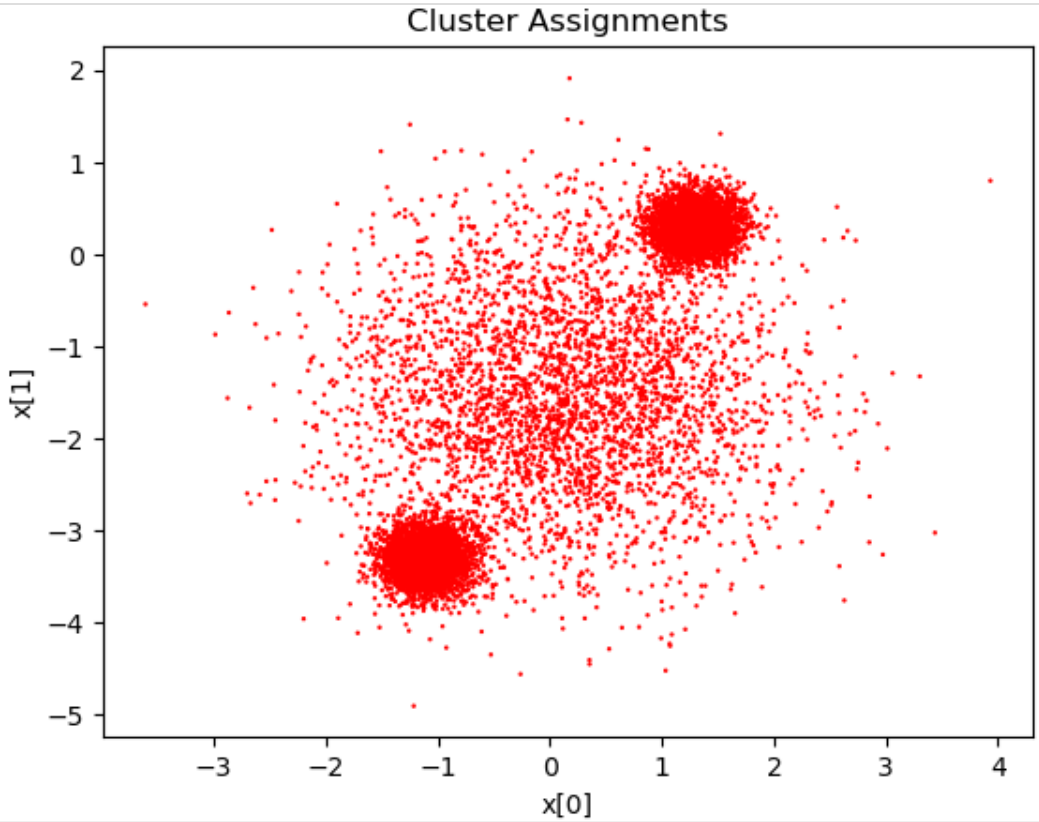
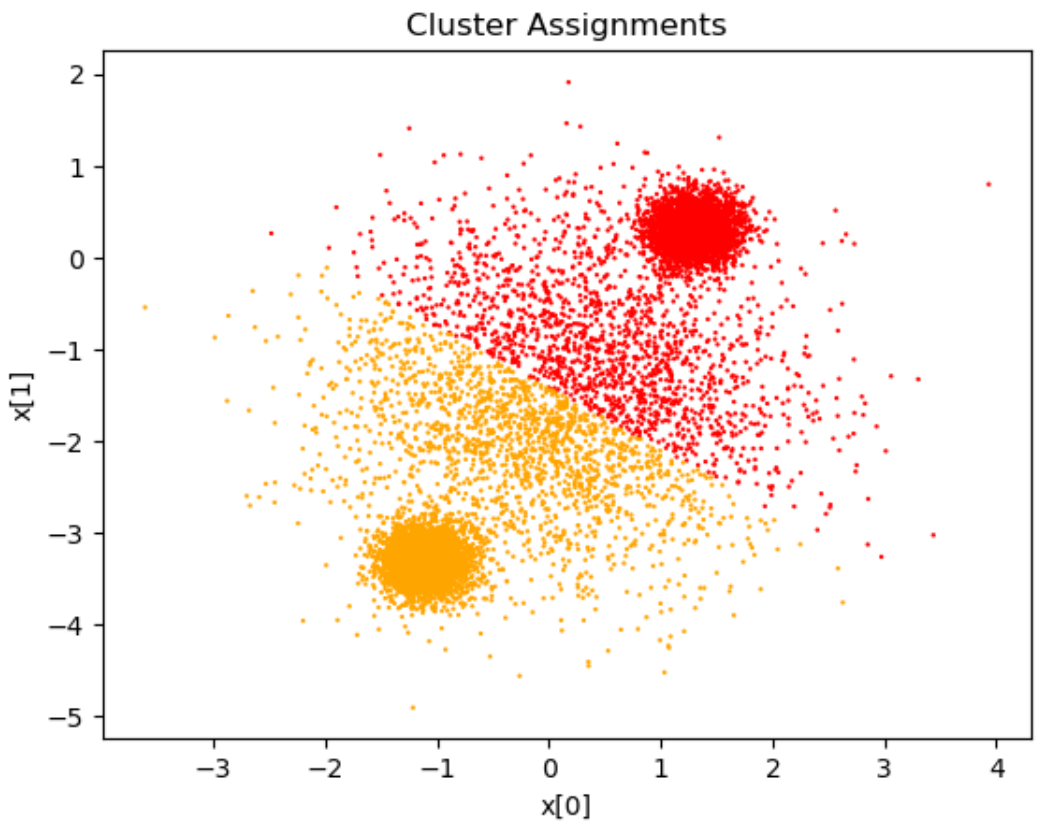
* All mathematical work related to the vectorization of the K-Means loss can be found in **Appendix A - K-Means**
* All code related to learning the K-Means model can be found in **Appendix B — kmeans.py** & **starter_kmeans.py**

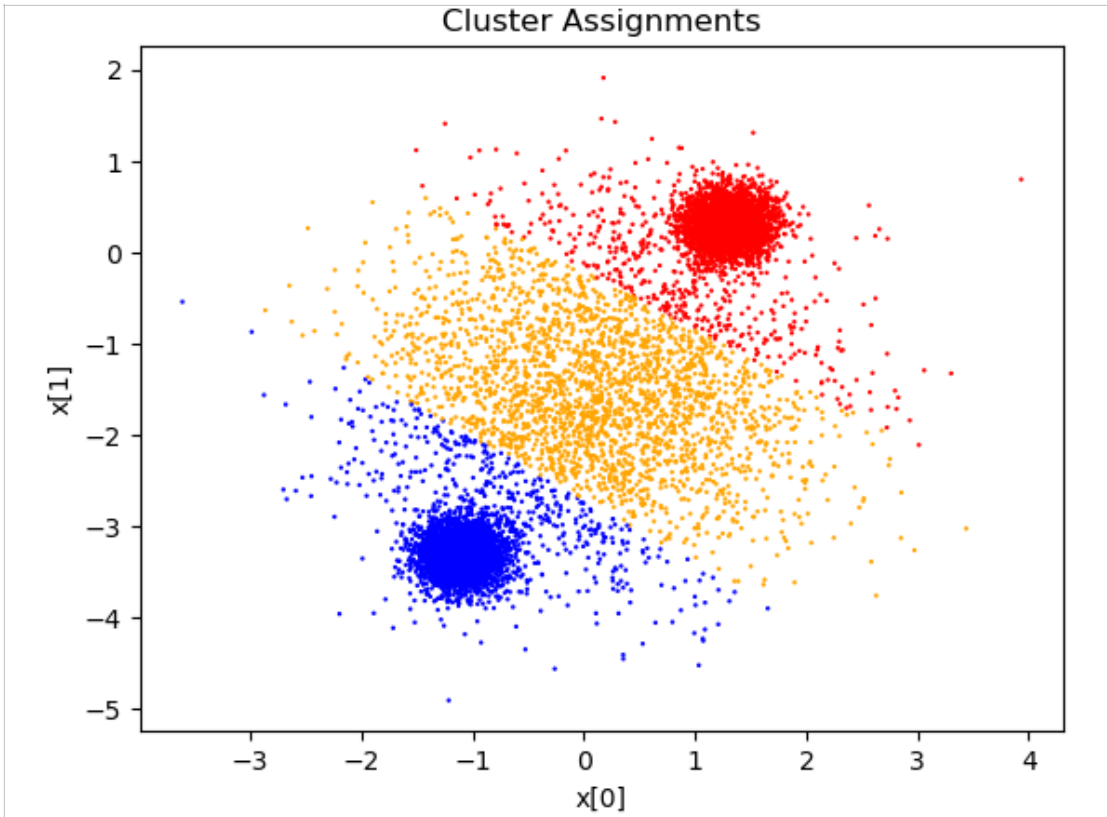
1. K-Means with K=3 clusters

μ = [[0.13439235 -1.5228275]
[-1.0565008 -3.2404728]
[1.2492981 0.25105822]]

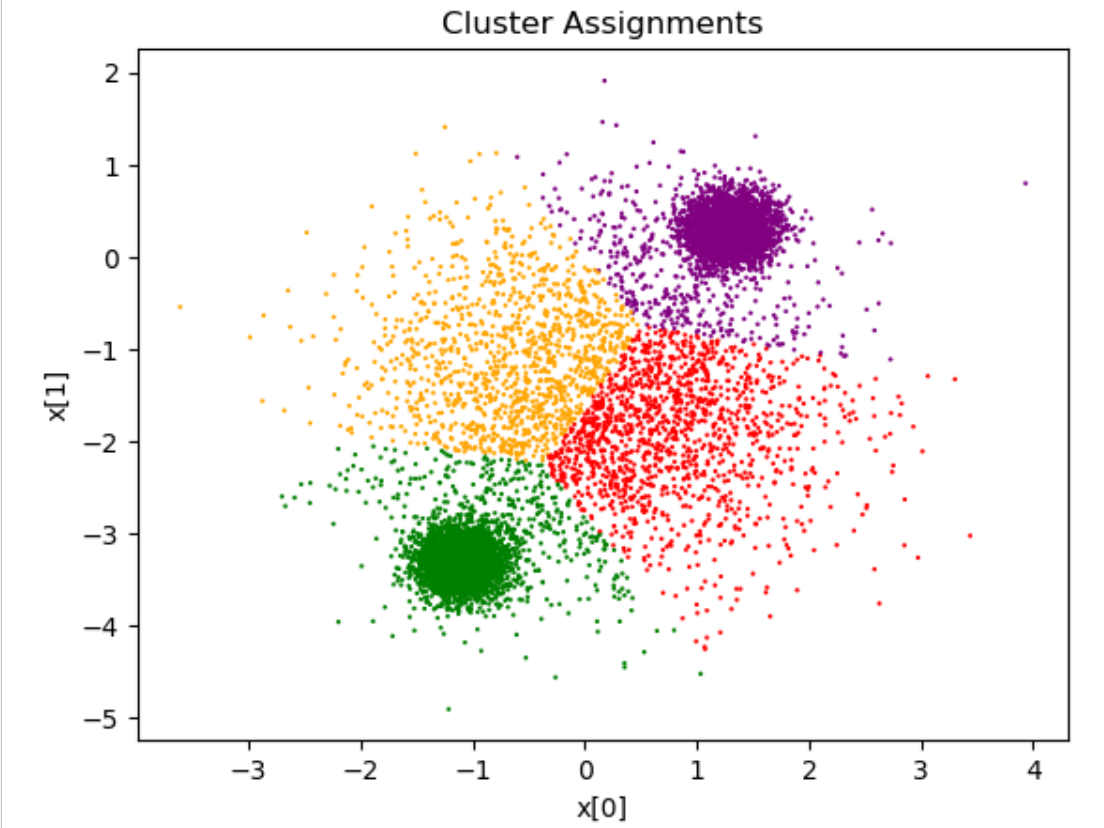
Loss Curve	**Cluster Partitions**
	

2. K-Means with K={1,2,3,4,5} clusters

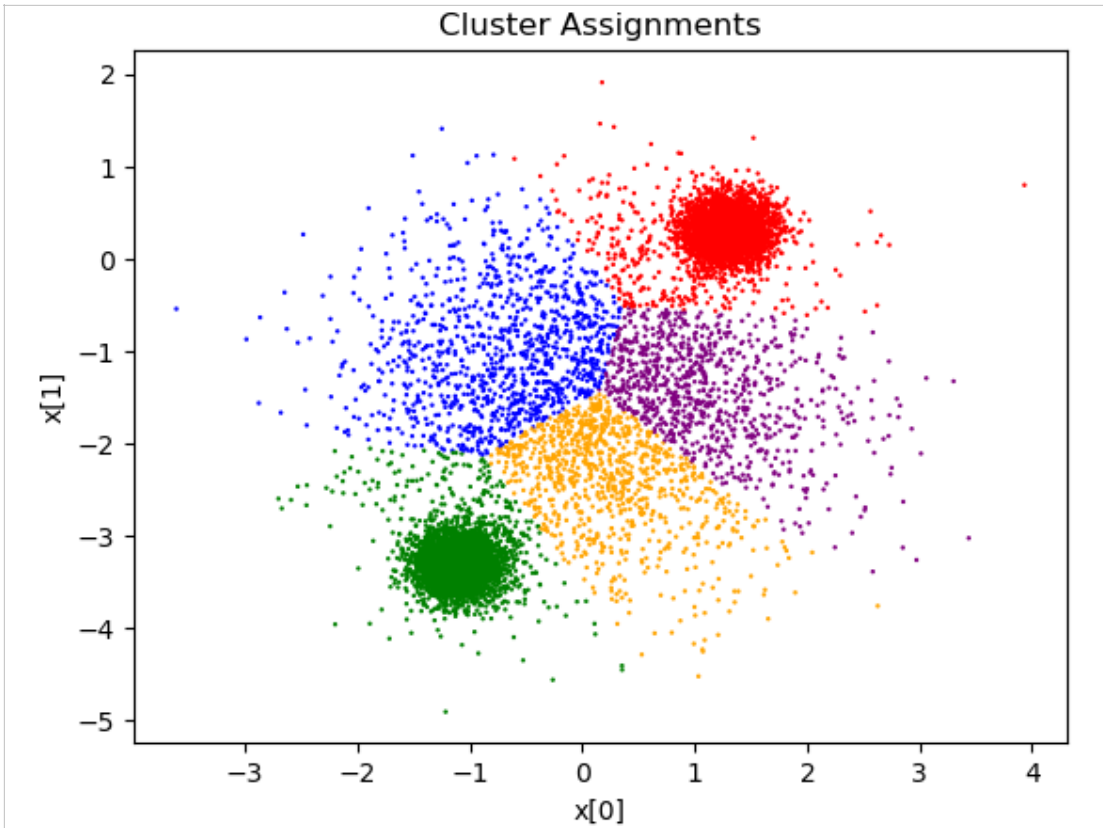
Cluster Partitions	Training Data Distribution
	Cluster 0: 1.0
	Cluster 0: 0.4954 Cluster 1: 0.5046



Cluster 0: 0.2383
Cluster 1: 0.3820
Cluster 2: 0.3797



Cluster 0: 0.1349
Cluster 1: 0.3713
Cluster 2: 0.3729
Cluster 3: 0.1209

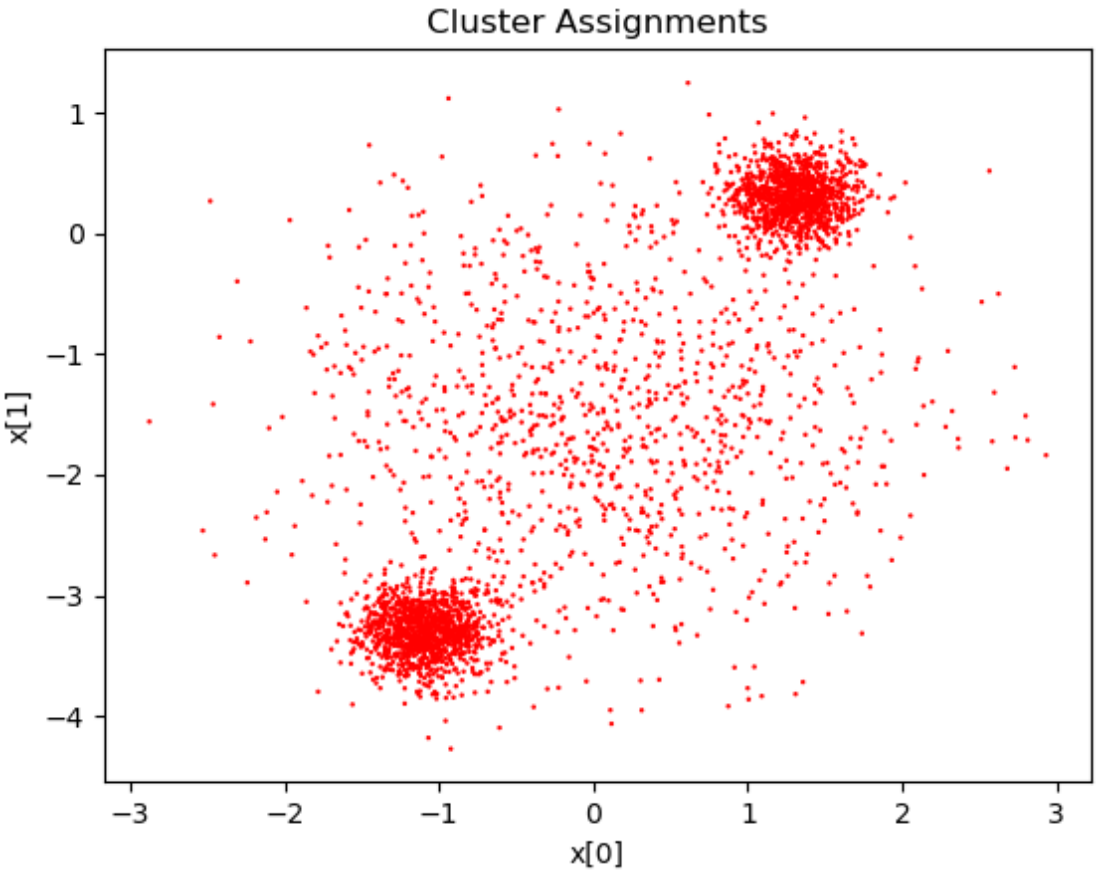
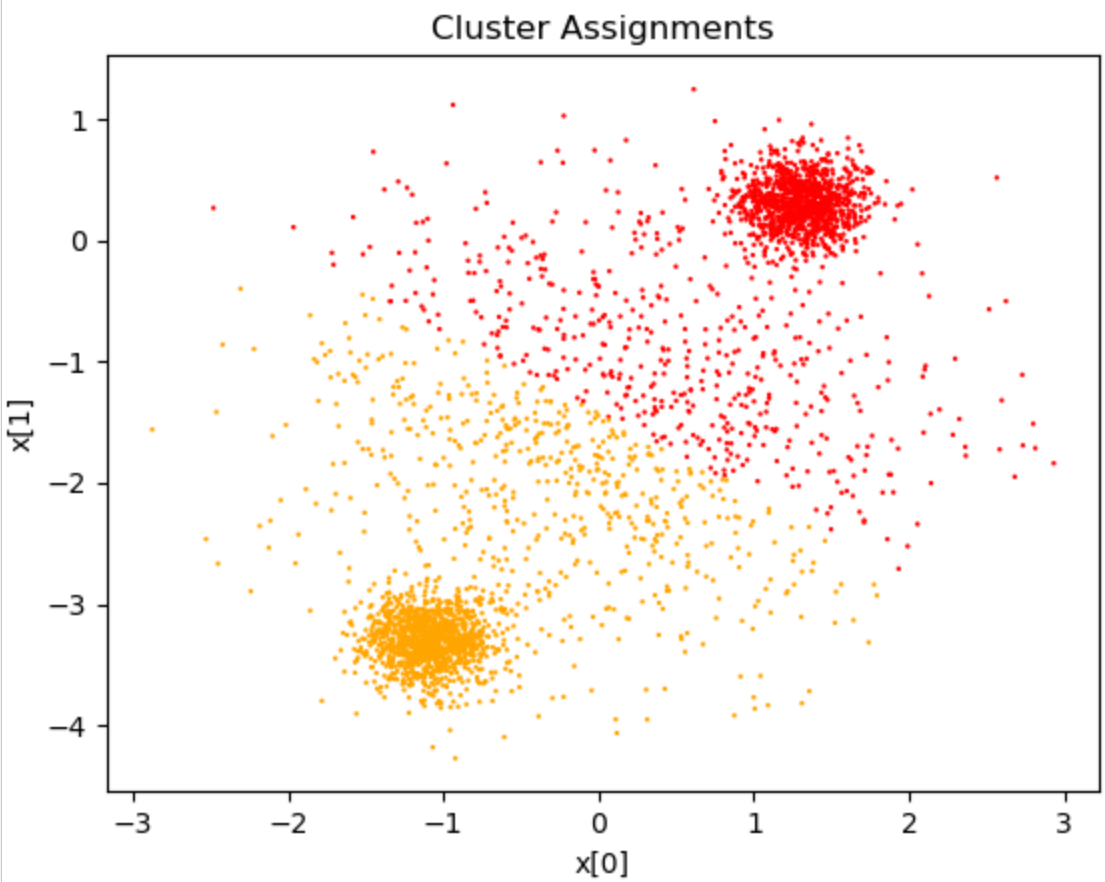


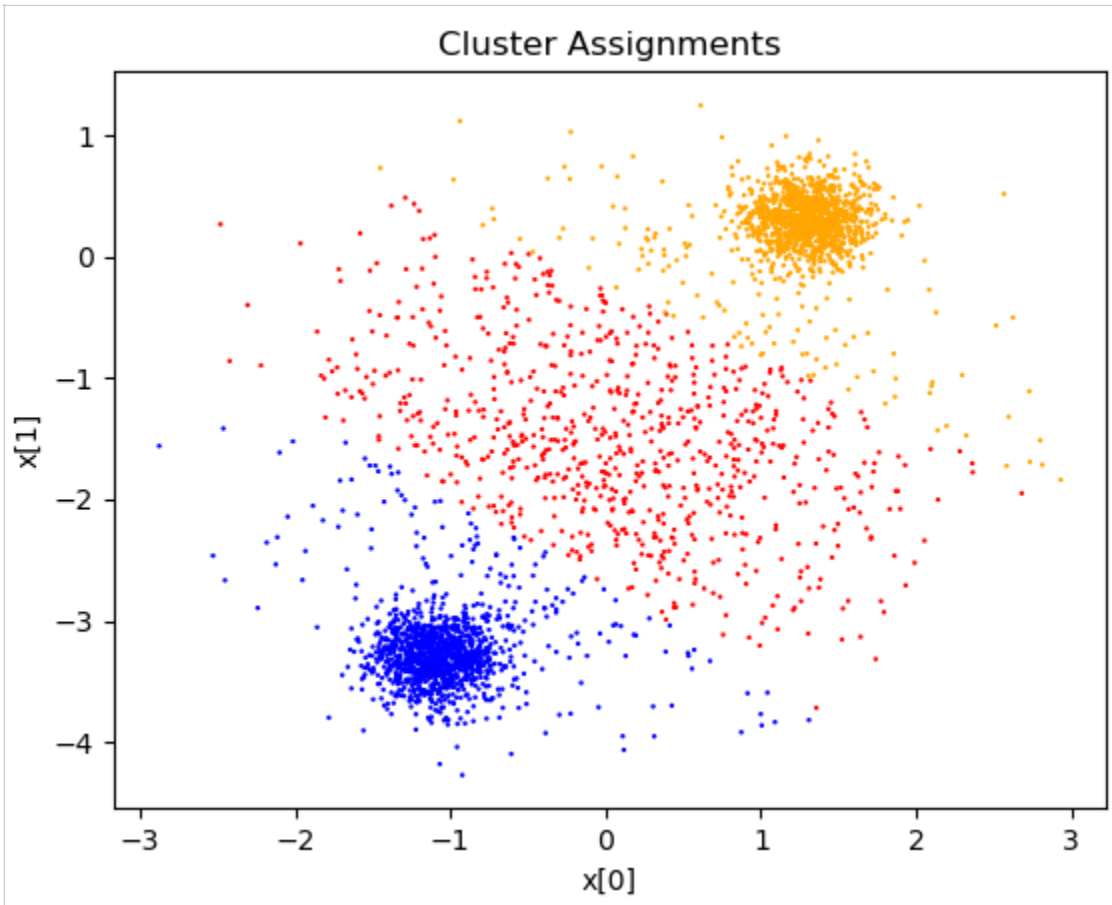
Cluster 0: 0.0842
Cluster 1: 0.3576
Cluster 2: 0.3624
Cluster 3: 0.1077
Cluster 4: 0.0881

How many clusters is best?

Since we can visualize the data set, we can immediately see that there are 3 clusters. There is a large wide cluster in the centre of the data plane, and two compact clusters to the top right and bottom left of the plane. However, K-Means is a hard clustering algorithm based on the distance between data points, and so there is no K that is able to perfectly segment the data into the 3 clusters we want. The best K-Means can do is when K=3, which still incorrectly distributes many points. One alternative is to have K=5, and then group several clusters into one (blue, yellow, purple) to try to approximate the correct cluster assignment.

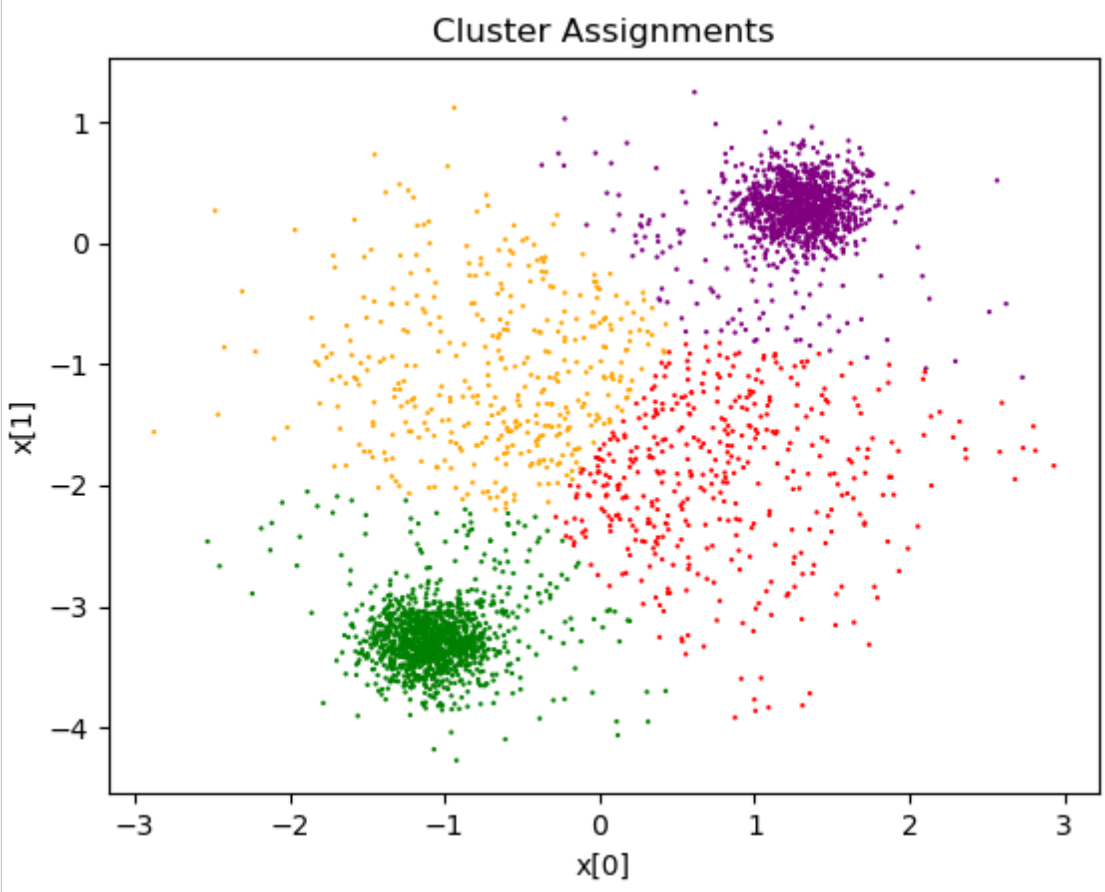
3. K-Means with K={1,2,3,4,5} clusters on Validation Data

Cluster Partitions	Validation Data Distribution
	Validation Loss: 12870.1045 Cluster 0: 1.0
	Validation Loss: 2960.6733 Cluster 0: 0.4818 Cluster 1: 0.5182



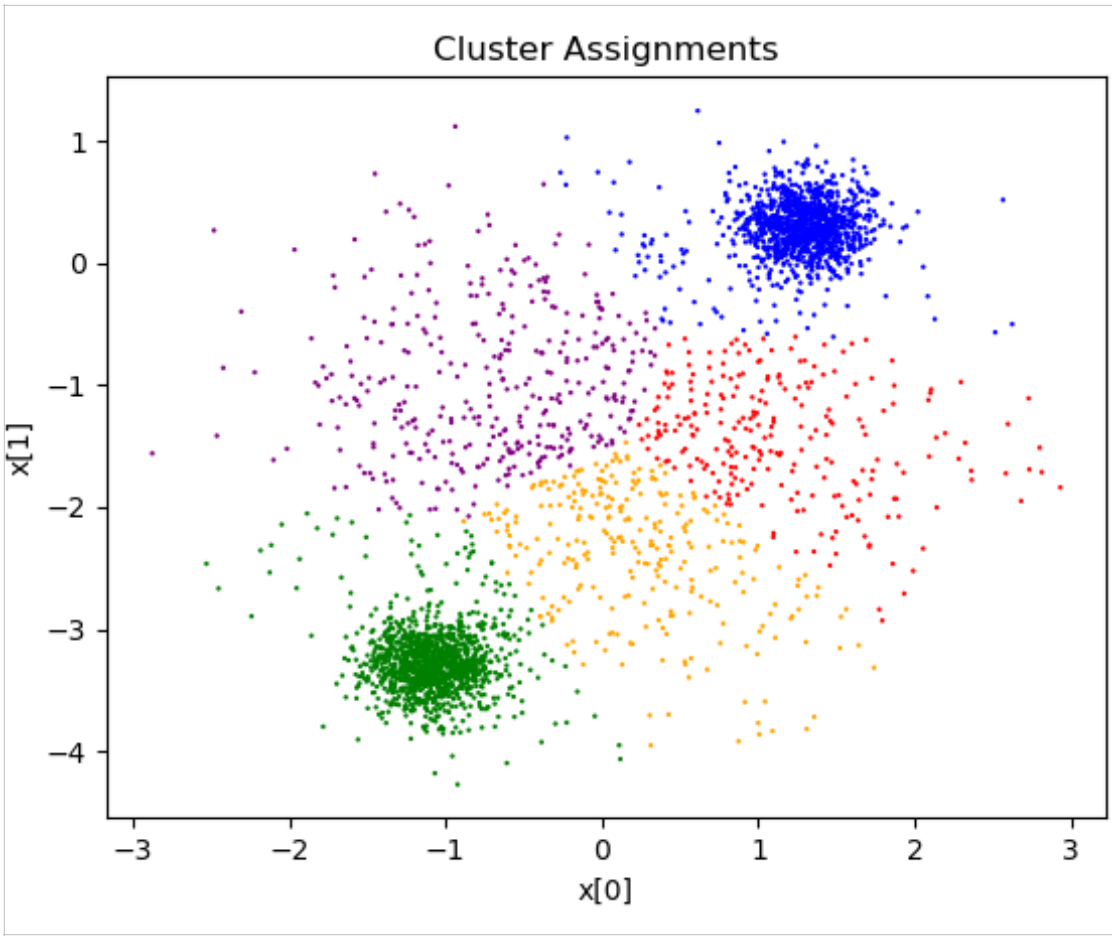
Validation Loss: 1629.2084

Cluster 0: 0.2286
Cluster 1: 0.3978
Cluster 2: 0.3735



Validation Loss: 1054.5376

Cluster 0: 0.1266
Cluster 1: 0.3879
Cluster 2: 0.3657
Cluster 3: 0.1197



Validation Loss: 887.24884

Cluster 0: 0.0750
Cluster 1: 0.3759
Cluster 2: 0.3567
Cluster 3: 0.1080
Cluster 4: 0.0843

How many clusters is best?

Since the validation data follows the same distribution as the training data, we can infer there will also be 3 clusters. We can also confirm this by visualizing the plots of the data, like in part 1. The loss for K-Means tells us nothing as it will always decrease as K (number of clusters) increases. This is because the loss is the sum of squared distances, which means that with more clusters the distances to clusters is naturally lower.

Part 2: Mixture of Gaussians

* All mathematical work related to the vectorization of the MoG loss function can be found in **Appendix A - GMM**
* All code related to learning the MoG model can be found in **Appendix B — gmm.py** & **starter_gmm.py**

Why is it important to use the log-sum-exp function instead of using tf.reduce_sum?

If we want to use the log gaussian matrix (log_GaussPDF) calculated in part 1 to help calculate the log posterior matrix (log_posterior), then it is necessary to use the log-sum-exp function. This is because the exp call will negate the log term from the log gaussian matrix. If we decide not to use the log gaussian matrix from part 1 then it is not necessary to use the log-sum-exp function; it would suffice to use a log-sum function. Please refer to the mathematical derivation of the vectorized log posterior matrix in Appendix A for more details.

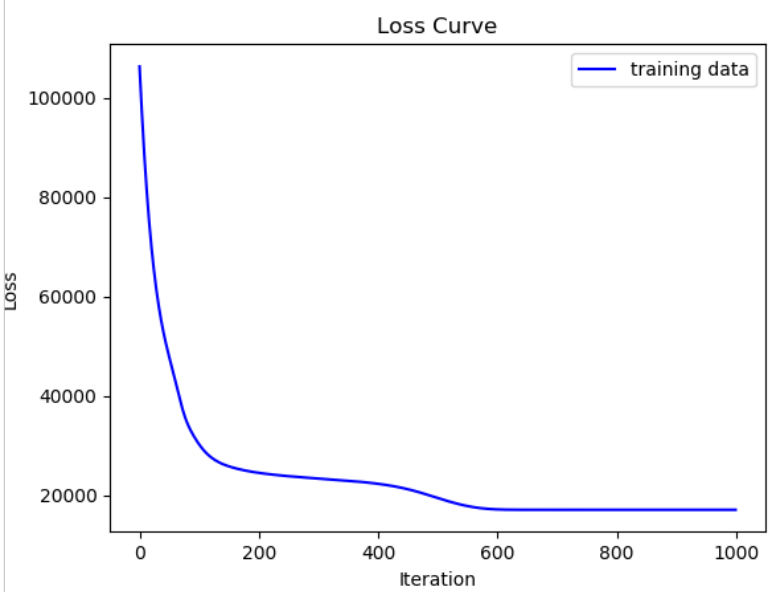
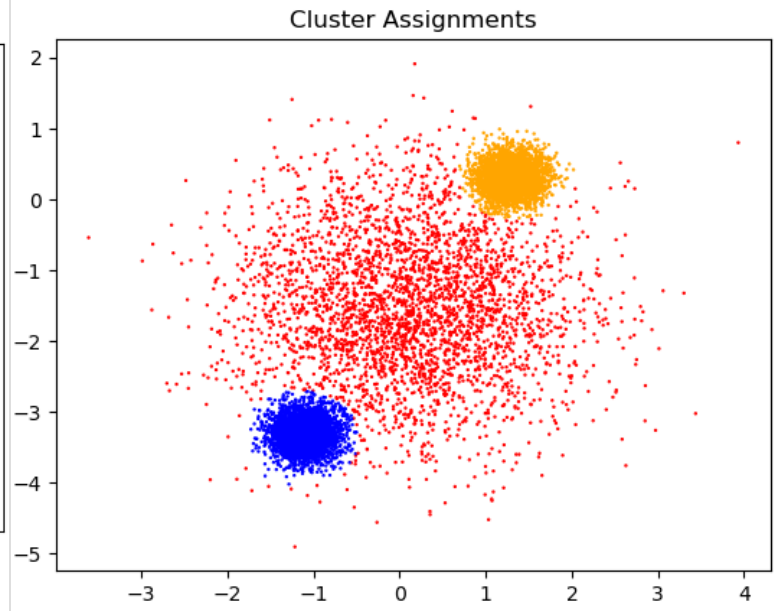
1. GMM with K=3 clusters

The learned gaussian cluster parameters are:

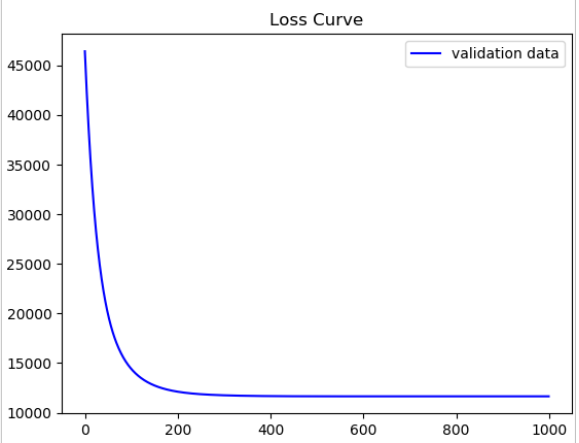
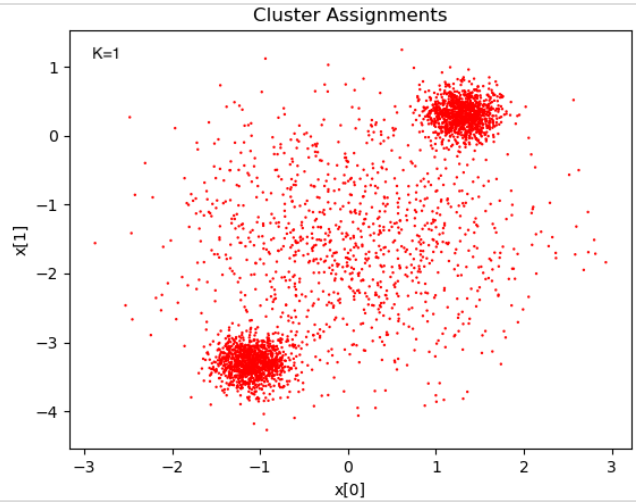
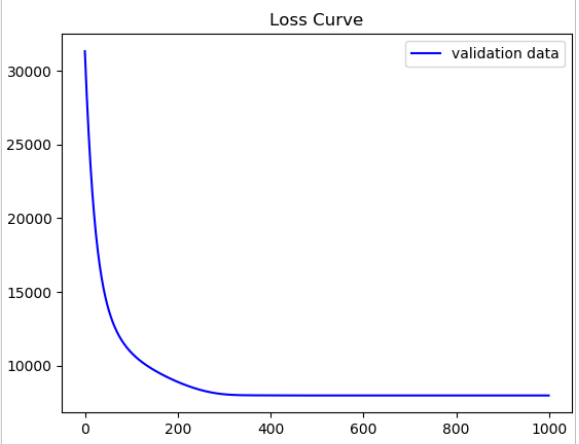
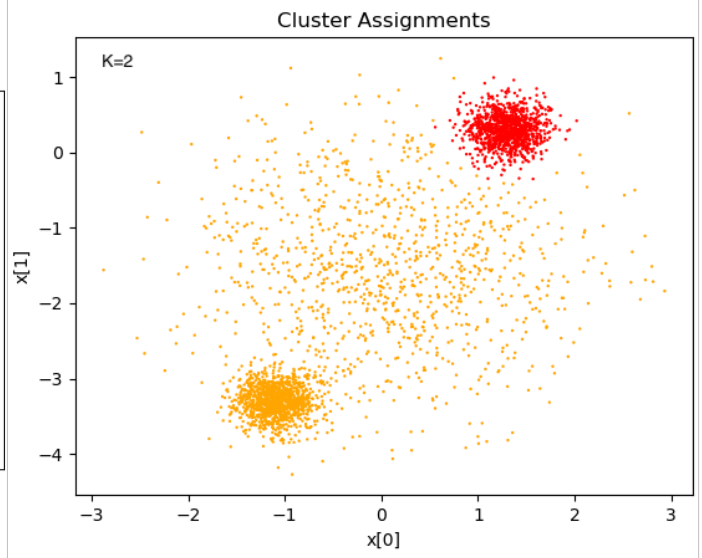
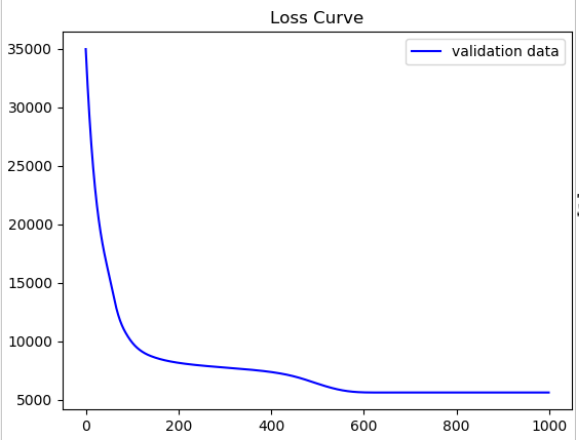
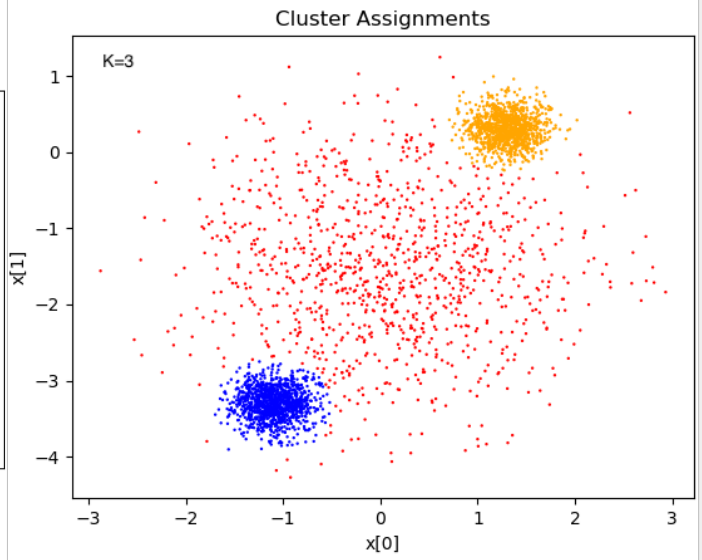
μ = [[0.10597418 -1.5274261]
[-1.1017274 -3.3061705]
[1.2986319 0.30903977]]

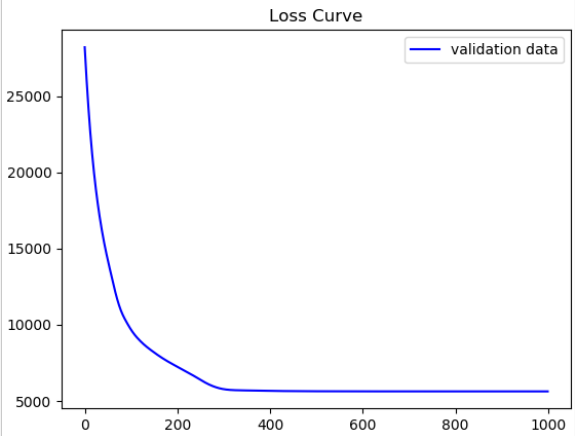
σ = [0.9935754 0.1976884 0.19710355]

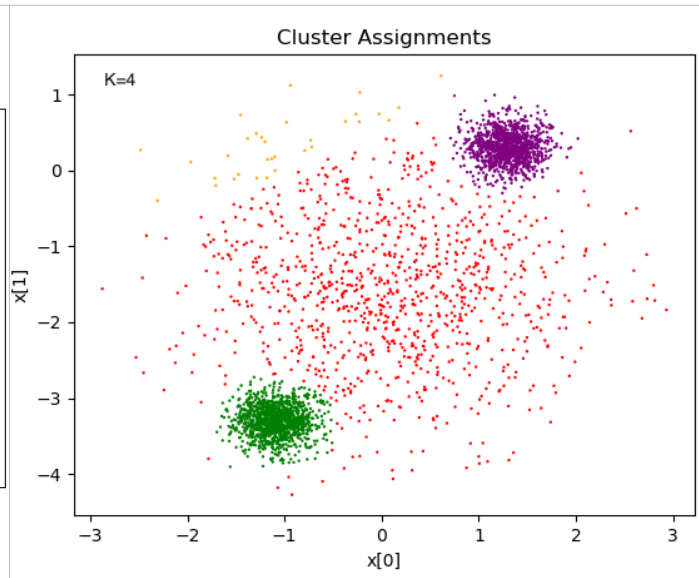
w = [0.33464736 0.33191344 0.33343923]

Loss Curve	**Cluster Partitions**
	

2. GMM with K={1,2,3,4,5} clusters on Validation Data

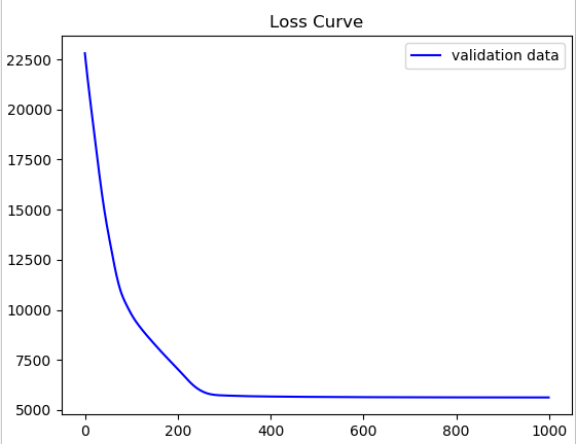
Loss Curve	Cluster Partition	Data Distributions
		Training Loss: [23265.98] Validation Loss: [11651.443] Cluster 0: 1.0
		Training Loss: [16155.764] Validation Loss: [7987.795] Cluster 0: 0.3381 Cluster 1: 0.6619
		Training Loss: [11505.926] Validation Loss: [5629.6353] Cluster 0: 0.3132 Cluster 1: 0.3507 Cluster 2: 0.3360

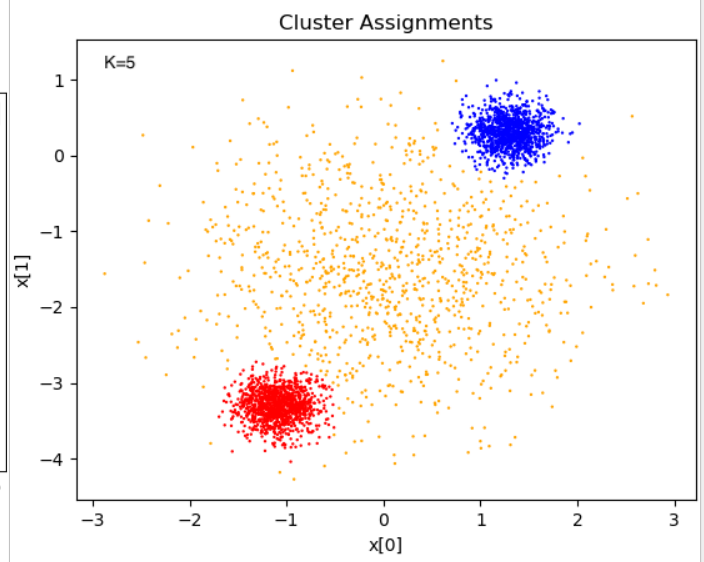




Training Loss: [11505.871]
Validation Loss: [5629.859]

Cluster 0: 0.3012
Cluster 1: 0.3522
Cluster 2: 0.3369
Cluster 3: 0.0096





Training Loss: [11506.596]
Validation Loss: [5631.4565]

Cluster 0: 0.0
Cluster 1: 0.3525
Cluster 2: 0.3366
Cluster 3: 0.3108
Cluster 4: 0.0

Which value of K is best?

Since the loss is defined as the negative log likelihood L $= -$ Log $P(\mathbf{X})$, the model with the lower loss will have the higher likelihood. Therefore, the value of K that produces the lowest loss should theoretically be the best choice of K. While this may not always be the case, as probabilistic models are never conclusive, it does provide a simple way to try and determine the best K value. Another characteristic we can look at is the actual cluster assignments of the data. Models with empty clusters are extraneous and may suggest a K value that is larger than the actual number of clusters.

By examining the validation data, K=3 produces the smallest validation loss, and therefore has the greatest probability. By examining the cluster assignments, K=4 has a cluster that only contain a handful of points, its validation loss is also slightly higher than that of K=3. Finally since we can *visualize* the data, we can see that there are 3 clusters and K=3 should produce the best results.

3. K-Means vs GMM

Similar to previous questions, I have held out 1/3 of the of the data for validation. The results below are based on the validation data

* Only *non-empty* clusters are listed in the data below

K	K-Means	MoG
5	Training Loss: 303200.56 **Validation Loss**: 154032.69 **Cluster 0:** # Points: 332 Percent of Points: 0.0996 **Cluster 2:** # Points: 643 Percent of Points: 0.1929 **Cluster 3:** # Points: 1710 Percent of Points: 0.5131 **Cluster 4:** # Points: 648 Percent of Points: 0.1944	Training Loss: [946597.8] **Validation Loss**: [472394.2] **Cluster 3:** # Points: 3333 Percent of Points: 1.0
10	Training Loss: 247106.12 **Validation Loss**: 122440.04 **Cluster 0:** # Points: 664 Percent of Points: 0.1992 **Cluster 5:** # Points: 975 Percent of Points: 0.2925 **Cluster 8:** # Points: 1046 Percent of Points: 0.3138 **Cluster 9:** # Points: 648 Percent of Points: 0.1944	Training Loss: [729600.4] **Validation Loss**: [361662.6] **Cluster 3:** # Points: 1046 Percent of Points: 0.3138 **Cluster 5:** # Points: 1639 Percent of Points: 0.4917 **Cluster 8:** # Points: 648 Percent of Points: 0.1944
15	Training Loss: 143512.4 **Validation Loss**: 71794.86 **Cluster 2:** # Points: 1046 Percent of Points: 0.3138 **Cluster 5:** # Points: 643 Percent of Points: 0.1929 **Cluster 12:** # Points: 664 Percent of Points: 0.1992 **Cluster 13:** # Points: 648 Percent of Points: 0.1944 **Cluster 14:** # Points: 332 Percent of Points: 0.0996	Training Loss: [322521.5] **Validation Loss**: [162851.06] **Cluster 3:** # Points: 1046 Percent of Points: 0.3138 **Cluster 5:** # Points: 643 Percent of Points: 0.1929 **Cluster 8:** # Points: 354 Percent of Points: 0.1062 **Cluster 11:** # Points: 664 Percent of Points: 0.1992 **Cluster 13:** # Points: 294 Percent of Points: 0.0882 **Cluster 14:** # Points: 332 Percent of Points: 0.0996

20	Training Loss: 141873.12 **Validation Loss**: 71077.17 **Cluster 3:** # Points: 340 Percent of Points: 0.1020 **Cluster 5:** # Points: 643 Percent of Points: 0.1929 **Cluster 10:** # Points: 1046 Percent of Points: 0.3138 **Cluster 12:** # Points: 664 Percent of Points: 0.1992 **Cluster 13:** # Points: 308 Percent of Points: 0.0924 **Cluster 17:** # Points: 332 Percent of Points: 0.0996	Training Loss: [322435.8] **Validation Loss**: [162847.56] **Cluster 3:** # Points: 1046 Percent of Points: 0.3138 **Cluster 5:** # Points: 643 Percent of Points: 0.1929 **Cluster 8:** # Points: 353 Percent of Points: 0.1059 **Cluster 11:** # Points: 664 Percent of Points: 0.1992 **Cluster 13:** # Points: 295 Percent of Points: 0.0885 **Cluster 14:** # Points: 332 Percent of Points: 0.0996
30	Training Loss: 140521.47 **Validation Loss**: 70445.234 **Cluster 2:** # Points: 643 Percent of Points: 0.1929 **Cluster 3:** # Points: 508 Percent of Points: 0.1524 **Cluster 6:** # Points: 351 Percent of Points: 0.1053 **Cluster 10:** # Points: 538 Percent of Points: 0.1614 **Cluster 13:** # Points: 297 Percent of Points: 0.0891 **Cluster 17:** # Points: 332 Percent of Points: 0.0996 **Cluster 28:** # Points: 664 Percent of Points: 0.1992	Training Loss: [573231.75] **Validation Loss**: [285103.66] **Cluster 3:** # Points: 1046 Percent of Points: 0.3138 **Cluster 5:** # Points: 975 Percent of Points: 0.2925 **Cluster 8:** # Points: 354 Percent of Points: 0.1062 **Cluster 13:** # Points: 294 Percent of Points: 0.0882 **Cluster 20:** # Points: 664 Percent of Points: 0.1992

Comment on how many clusters you think are within the dataset and compare the learnt results of K-Means and MoG

Since we cannot visualize this dataset, we must rely on the loss values and cluster distributions to determine the number of clusters. Unfortunately, since K-Means loss is just a sum of the distances from a point to its centre, this loss does not tell us anything. With a higher K value, the loss/distances will always decrease as the distances become smaller. Therefore it is best to analyze the negative log likelihood of the MoG model.

One way we can do this is by setting aside 1/3 of the data for validation. After we have learned the clusters based on the training data we then compute the negative log likelihood (loss) for the validation data. The K value that produces the lowest validation loss is most likely to be the best K value.

The smallest loss for the MoG model is when K=20 (loss=162847.56), with the next smallest loss at K=15 (loss=162851.06). Next, if we analyze the cluster distributions for K=20 and K=15 we see that they are both extremely similar. Both have 6 populated clusters and the number of points in each of these clusters is almost exactly the same. Therefore, we can estimate that there are about K=20 clusters in this dataset.

Appendix A — Vectorization

The following pages show the mathematical vectorization for the K-Means, and GMM. A dot overtop a variable indicates broadcasting.

K-Means

$$X = \begin{bmatrix} - \underline{x}_1^T - \\ \vdots \\ - \underline{x}_N^T - \end{bmatrix} \qquad \mu = \begin{bmatrix} - \underline{\mu}_1 - \\ \vdots \\ - \underline{\mu}_K - \end{bmatrix}$$

Let the distance matrix be

$$D = \left[\| \underline{x}_i - \underline{\mu}_j \|^2 \right] = \begin{bmatrix} \overline{\underline{x}_1 \underline{\mu}_1} & \cdots & \overline{\underline{x}_1 \underline{\mu}_K} \\ \vdots & & \\ \overline{\underline{x}_N \underline{\mu}_1} & \cdots & \overline{\underline{x}_N \underline{\mu}_K} \end{bmatrix}$$

where $\overline{\underline{x}_i \underline{\mu}_j} = \| \underline{x}_i - \underline{\mu}_j \|^2$

$$\Rightarrow d_{ij} = \| \underline{x}_i - \underline{\mu}_j \|^2$$

$$= \sum_d (x_{id} - \mu_{jd})^2 = \sum_d (x_{id}^2 - 2 x_{id} \mu_{jd} + \mu_{jd}^2)$$

$$= \sum_d x_{id}^2 \cdot 1_{dj} - 2 x_{id} \mu_{dj}^T + 1_{id} (\mu_{dj}^T)^2$$

$$\Rightarrow \boxed{D = X^2 \cdot 1 - 2 X \mu^T + 1 (\mu^T)^2}$$

GMM

$$X = \begin{bmatrix} \underline{x}_1^T \\ \vdots \\ \underline{x}_N^T \end{bmatrix} \quad \mu = \begin{bmatrix} \underline{\mu}_1^T \\ \vdots \\ \underline{\mu}_K^T \end{bmatrix} \qquad \underline{\sigma} = [\sigma_1 \cdots \sigma_K] \qquad \underline{\pi} = [P(1) \cdots P(K)]$$

$$P = [\ln P(\underline{x}_i | j)] = \ln \begin{bmatrix} P(\underline{x}_1|1) & \cdots & P(\underline{x}_1|K) \\ \vdots & & \vdots \\ P(\underline{x}_N|1) & \cdots & P(\underline{x}_N|K) \end{bmatrix}$$

where $P(\underline{x}_i|j) = \mathcal{N}(\underline{x}_i; \underline{\mu}_j, \sigma_j^2)$

Let P be the log gaussian matrix

$$\Rightarrow \quad p_{ij} = \ln \mathcal{N}(\underline{x}_i; \underline{\mu}_j, \sigma_j^2)$$

$$= \ln \left[\frac{1}{(2\pi)^{d/2} |\Sigma_j|^{1/2}} \cdot e^{-\frac{1}{2}(\underline{x}_i - \underline{\mu}_j)^T \Sigma_j^{-1} (\underline{x}_i - \underline{\mu}_j)} \right]$$

where $\begin{cases} \Sigma_j = \sigma_j^2 \cdot I \\ |\Sigma_j| = \sigma_j^{2d} \\ \Sigma_j^{-1} = \frac{1}{\sigma_j^2} \cdot I \end{cases}$ (due to independence of RVs and same standard deviation)

$$= \ln \left[\frac{1}{(\sigma_j \sqrt{2\pi})^d} \cdot e^{-\frac{1}{2\sigma_j^2}(\underline{x}_i - \underline{\mu}_j)^2} \right]$$

$$= -d \ln(\sigma_j \sqrt{2\pi}) - \frac{1}{2\sigma_j^2}(\underline{x}_i - \underline{\mu}_j)^2 = -d \ln(\sigma_j \sqrt{2\pi}) - \frac{1}{2\sigma_j^2} \sum_d (x_{id} - \mu_{jd})^2$$

$$= -d \ln(\sigma_j \sqrt{2\pi}) - \frac{1}{2\sigma_j^2} \sum_d \left[x_{id}^2 \cdot 1_{dj} - 2 x_{id} \mu_{dj}^T + 1_{id} (\mu_{dj}^T)^2 \right]$$

$$\Rightarrow \boxed{P = -d \ln(\dot{\underline{\sigma}} \sqrt{2\pi}) - \frac{1}{2\dot{\underline{\sigma}}^2} \otimes \left[X^2 \cdot 1 - 2X\mu^T + 1(\mu^T)^2 \right]}$$

Similarly let Q be the log posterior matrix

$$Q = [\ln P(j | \underline{x}_i)]$$

$$\Rightarrow q_{ij} = \ln \left[\frac{P(\underline{x}_i|j) \cdot P(j)}{\sum_1^K P(\underline{x}_i|k) \cdot P(k)} \right] = \ln P(\underline{x}_i|j) + \ln P(j) - \ln \left[\sum_1^K P(\underline{x}_i|k) \cdot P(k) \right]$$

$$= \ln P(\underline{x}_i|j) + \ln P(j) - \ln \left[\sum_{k=1}^K e^{\ln P(\underline{x}_i|k) \cdot P(k)} \right]$$

$$= \ln P(x_i|j) + \ln P(j) - \ln \left[\sum e^{\ln P(\underline{x}_i|k) + \ln P(k)} \right]$$

$$\Rightarrow \boxed{Q = P + \ln \dot{\underline{\pi}} - \underbrace{\ln \sum_{\text{rows}} e^{(P + \ln \dot{\underline{\pi}})}}_{\text{log-sum-exp } \otimes}}$$

Maximum Likelihood Estimate

$$L(\mu, \underline{\sigma}, \underline{\pi}) = -\ln P(X)$$

$$= -\ln \prod_n \sum_k P(\underline{x}_n | k) \cdot P(k)$$

$$= -\sum_n \left(\ln \sum_k P(\underline{x}_n | k) \cdot P(k) \right)$$

$$= -\sum_n \left(\ln \sum_k e^{\ln P(x_n | k) \cdot P(k)} \right)$$

$$= -\sum_n \left(\ln \sum_k e^{\ln P(x_n | k) + \ln P(k)} \right)$$

$$= -\sum_{\text{cols}} \left(\underbrace{\ln \sum_{\text{rows}} e^{(P + \ln \underline{\pi})}}_{\text{log-sum-exp}} \right)$$

Appendix B — Code

helper.py

```
import tensorflow as tf

def reduce_logsumexp(input_tensor, reduction_indices=1, keep_dims=False):
  """Computes the sum of elements across dimensions of a tensor in log domain.

     It uses a similar API to tf.reduce_sum.

  Args:
    input_tensor: The tensor to reduce. Should have numeric type.
    reduction_indices: The dimensions to reduce.
    keep_dims: If true, retains reduced dimensions with length 1.
  Returns:
    The reduced tensor.
  """
  max_input_tensor1 = tf.reduce_max(
      input_tensor, reduction_indices, keep_dims=keep_dims)
  max_input_tensor2 = max_input_tensor1
  if not keep_dims:
    max_input_tensor2 = tf.expand_dims(max_input_tensor2, reduction_indices)
  return tf.log(
      tf.reduce_sum(
          tf.exp(input_tensor - max_input_tensor2),
          reduction_indices,
          keep_dims=keep_dims)) + max_input_tensor1


def logsoftmax(input_tensor):
  """Computes normal softmax nonlinearity in log domain.

     It can be used to normalize log probability.
     The softmax is always computed along the second dimension of the input Tensor.

  Args:
    input_tensor: Unnormalized log probability.
  Returns:
    normalized log probability.
  """
  return input_tensor - reduce_logsumexp(input_tensor, reduction_indices=1, keep_dims=True)
```

starter_kmeans.py

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

# Loading data
#data = np.load('data100D.npy')
data = np.load('data2D.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:]]

# Distance function for K-means
# 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)
#   pair_dist = np.sum(X**2, axis=1) - 2*(X @ MU.T) + np.sum((MU.T)**2, axis=0)
def distanceFunc(X, MU):
    pair_dist = tf.reduce_sum(tf.square(X), axis=1, keepdims=True) \
                - 2 * tf.matmul(X, tf.transpose(MU)) \
                + tf.reduce_sum(tf.square(tf.transpose(MU)), axis=0, keepdims=True)

    return pair_dist

# Squared Distance Loss for K-Means
def calculate_loss(X, MU):
    D = distanceFunc(X, MU)
    e = tf.reduce_min(D, axis=1)
    L = tf.reduce_sum(e)
    return L

# Partitions the data into K clusters based on MU
# returns a Nx1 vector of cluster assignments for x1 - xN
# the clusters are numbered from 0 to K-1
def cluster_assignments(X, MU):
    D = distanceFunc(X, MU)
    s = tf.argmin(D, axis=1)
    return s
```

kmeans.py

```
from starter_kmeans import *
import time

# K: number of clusters
def build_graph(K, learning_rate):
    tf.set_random_seed(421)

    # DEFINE INPUT PLACEHOLDERS
    X = tf.placeholder(tf.float32, shape=[None, dim], name="X")

    # DEFINE VARIABLES TO LEARN
    MU = tf.get_variable('MU',
        shape=[K, dim],
        initializer=tf.initializers.random_normal(mean=0, stddev=1))

    # DEFINE LOSS FUNCTIONS
    loss = calculate_loss(X, MU)

    # DETERMINE CLUSTER ASSIGNMENTS
    s = cluster_assignments(X, MU)

    # INITIALIZE GRADIENT DESCENT OPTIMIZER
    optimizer = tf.train.AdamOptimizer(
        learning_rate=learning_rate,
        beta1=0.9,
        beta2=0.99,
        epsilon=1e-5
    ).minimize(loss)

    return optimizer, X, MU, s, loss

def train_clusters(K, learning_rate, n_epochs):
    optimizer, X, MU, s, loss = build_graph(K, learning_rate)
    global_init = tf.global_variables_initializer()

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

        loss_curves = {'train': [], 'valid': []}
        cluster_assignments = {}

        for iter in range(n_epochs):
            # GRADIENT DESCENT STEP on data set
            feed_dict_batch = {X: data}
            [_opt, _loss] = sess.run([optimizer, loss], feed_dict=feed_dict_batch)
            loss_curves['train'].append(_loss)

            # GET VALIDATION LOSS
            if is_valid:
                feed_dict_batch = {X: val_data}
                [_loss] = sess.run([loss], feed_dict=feed_dict_batch)
                loss_curves['valid'].append(_loss)

        # GET CLUSTER ASSIGNMENTS
        feed_dict_batch = {X: data}
        [cluster_assignments['train']] = sess.run([s], feed_dict=feed_dict_batch)
        if is_valid:
            feed_dict_batch = {X: val_data}
            [cluster_assignments['valid']] = sess.run([s], feed_dict=feed_dict_batch)
```

```
        # GET LEARNED K-MEANS CLUSTERS
        [MU] = sess.run([MU], feed_dict={})

    return MU, loss_curves, cluster_assignments


def main():
    start_time = time.time()
    K = 5
    MU, loss, cluster_assignments = train_clusters(
        K=K,
        learning_rate=0.01,
        n_epochs=1000
    )
    end_time = time.time()
    print("--- %s seconds ---" % (time.time() - start_time))

    # REPORT FINAL TRAINING AND VALIDATION LOSS
    print("Training Loss:", loss['train'][-1])
    type = 'train'
    if is_valid:
        print("Validation Loss:", loss['valid'][-1])
        type = 'valid'

    # CALCULATE CLUSTER DISTRIBUTIONS
    for cluster in range(K):
        print("Cluster {}:\n\t# Points: {}\n\tPercent of Points: {}".format(
            cluster,
            np.sum(cluster_assignments[type]==cluster),
            np.mean(cluster_assignments[type]==cluster))
        )
    print("MU:\n", MU)

    # PLOT LOSS CURVE
    plt.plot(loss[type], color='blue', label='training data' if not is_valid else 'validation data')
    plt.legend()
    plt.title('Loss Curve')
    plt.ylabel('Squared Distance Loss')
    plt.xlabel('Iteration')
    plt.show()

    # PLOT CLUSTER ASSIGNMENTS
    colors = ['red','green','blue','purple', 'orange']
    plt.scatter(
        data[:,0] if not is_valid else val_data[:,0],
        data[:,1] if not is_valid else val_data[:,1],
        s=0.5,
        c=cluster_assignments[type],
        cmap=matplotlib.colors.ListedColormap(colors)
    )
    plt.title('Cluster Assignments')
    plt.xlabel('x[0]')
    plt.ylabel('x[1]')
    plt.show()


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

starter_gmm.py

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

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

# Constants
pi = 3.141592654

# 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:]]

# Distance function for GMM
# 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)
def distanceFunc(X, MU):
   pair_dist = tf.reduce_sum(tf.square(X), axis=1, keepdims=True) \
              - 2 * tf.matmul(X, tf.transpose(MU)) \
              + tf.reduce_sum(tf.square(tf.transpose(MU)), axis=0, keepdims=True)

    return pair_dist

# Inputs
#   X: N X D
#   MU: K X D
#   sigma: 1 X K
# Outputs:
#   log Gaussian PDF N X K
def log_GaussPDF(X, MU, sigma):
    # HEADS UP:  I define sigma to be 1xK NOT Kx1
    pair_dist = distanceFunc(X, MU)
    log_PDF = - dim * tf.log(sigma * np.sqrt(2*pi)) - pair_dist / (2 * tf.square(sigma))
    return log_PDF
```

```
# Input
#   log_PDF: log Gaussian PDF N X K
#   log_pi: 1 X K
# Outputs
#   log_post: N X K
def log_posterior(log_PDF, log_pi):
    # HEADS UP:  I define log_pi to be 1xK NOT Kx1
    Z = log_PDF + log_pi
    log_post = Z - reduce_logsumexp(Z, reduction_indices=1, keep_dims=True)
    return log_post

# Input
#   X: N X D
#   MU: K X D
#   sigma: 1 X K
#   w: 1 X K (weights aka. P(k))
# Outputs
#   loss: constant
def calculate_loss(X, MU, sigma, w):
    P = log_GaussPDF(X, MU, sigma)
    Q = tf.reduce_logsumexp(P + tf.log(w), reduction_indices=1, keep_dims=True)
    loss = - tf.reduce_sum(Q, reduction_indices=0, keep_dims=False)
    return loss

# Returns a Nx1 vector of cluster assignments
def cluster_assignments(X, MU, sigma, w):
    log_PDF = log_GaussPDF(X, MU, sigma)
    P_j_x = log_posterior(log_PDF, tf.log(w))
    s = tf.argmax(P_j_x, axis=1)
    return s
```

gmm.py

```
from starter_gmm import *
import time

def build_graph(K, learning_rate):
    tf.set_random_seed(421)

    # DEFINE INPUT PLACEHOLDERS
    X = tf.placeholder(tf.float32, shape=[None, dim], name="X")

    # DEFINE VARIABLES TO LEARN
    MU = tf.get_variable('MU',
        shape=[K, dim],
        initializer=tf.initializers.random_normal(mean=0, stddev=1))

    sigma_unconstrained = tf.get_variable('sigma_unconstrained',
        shape=[1, K],
        initializer=tf.initializers.random_normal(mean=0, stddev=1))
    sigma = tf.exp(sigma_unconstrained, name='sigma')

    w_unconstrained = tf.get_variable('weight_unconstrained',
        shape=[1, K],
        initializer=tf.initializers.random_normal(mean=0, stddev=1))
    ln_w = logsoftmax(w_unconstrained)  # Note: I have modified logsoftmax to axis=1
    w = tf.exp(ln_w, name='weight')

    # DEFINE LOSS FUNCTIONS
    loss = calculate_loss(X, MU, sigma, w)

    # DETERMINE CLUSTER ASSIGNMENTS
    s = cluster_assignments(X, MU, sigma, w)

    # INITIALIZE GRADIENT DESCENT OPTIMIZER
    optimizer = tf.train.AdamOptimizer(
        learning_rate=learning_rate,
        beta1=0.9,
        beta2=0.99,
        epsilon=1e-5
    ).minimize(loss)

    return optimizer, X, MU, sigma, w, s, loss



def train_clusters(K, learning_rate, n_epochs):
    optimizer, X, MU, sigma, w, s, loss = build_graph(K, learning_rate)
    global_init = tf.global_variables_initializer()

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

        loss_curves = {'train': [], 'valid': []}
        cluster_assignments = {}

        for iter in range(n_epochs):
            # GRADIENT DESCENT STEP on data set
            feed_dict_batch = {X: data}
            [_opt, _loss] = sess.run([optimizer, loss], feed_dict=feed_dict_batch)
            loss_curves['train'].append(_loss)
```

```
        # GET VALIDATION LOSS
        if is_valid:
            feed_dict_batch = {X: val_data}
            [_loss] = sess.run([loss], feed_dict=feed_dict_batch)
            loss_curves['valid'].append(_loss)

    # GET CLUSTER ASSIGNMENTS
    feed_dict_batch = {X: data}
    [cluster_assignments['train']] = sess.run([s], feed_dict=feed_dict_batch)
    if is_valid:
        feed_dict_batch = {X: val_data}
        [cluster_assignments['valid']] = sess.run([s], feed_dict=feed_dict_batch)

    # GET LEARNED GMM CLUSTERS
    [MU, sigma, w] = sess.run([MU, sigma, w], feed_dict={})

return MU, sigma, w, cluster_assignments, loss_curves


def main():
    start_time = time.time()
    K = 3
    MU, sigma, w, cluster_assignments, loss = train_clusters(
        K=K,
        learning_rate=0.01,
        n_epochs=1000
    )
    end_time = time.time()
    print("--- %s seconds ---" % (time.time() - start_time))

    # REPORT FINAL TRAINING AND VALIDATION LOSS
    print("Training Loss:", loss['train'][-1])
    type = 'train'
    if is_valid:
        print("Validation Loss:", loss['valid'][-1])
        type = 'valid'

    # CALCULATE CLUSTER DISTRIBUTIONS
    for cluster in range(K):
        print("Cluster {}:\n\t# Points: {}\n\tPercent of Points: {}".format(
            cluster,
            np.sum(cluster_assignments[type]==cluster),
            np.mean(cluster_assignments[type]==cluster))
        )
    print("MU:\n", MU)
    print("sigma:\n", sigma[0])
    print("weights:\n", w[0])

    # PLOT LOSS CURVE
    plt.plot(loss[type], color='blue', label='training data' if not is_valid else 'validation data')
    plt.legend()
    plt.title('Loss Curve')
    plt.ylabel('Loss')
    plt.xlabel('Iteration')
    plt.show()
```

```
    # PLOT CLUSTER ASSIGNMENTS
    colors = ['red','green','blue','purple', 'orange']
    plt.scatter(
        data[:,0] if not is_valid else val_data[:,0],
        data[:,1] if not is_valid else val_data[:,1],
        s=0.5,
        c=cluster_assignments[type],
        cmap=matplotlib.colors.ListedColormap(colors)
    )
    plt.title('Cluster Assignments')
    plt.xlabel('x[0]')
    plt.ylabel('x[1]')
    plt.show()

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