Jia Ming (Carter) Huang (Student ID: 1003893245)

	P1	P2	Total
Mark			

Table 1: Marking Table

Submission Date: Monday, April 11, 2022, 11:59 AM
Number of Pages: 18

Contents

1 K-means [9 pt.]

The distance function implemented is $\sqrt{\sum_{i=1}^{D}(x_n^i - \mu_k^i)^2} \forall n, k$, as shown in Figure 1.

```
# Distance function for K-means and GMM
def distance_func(X, mu):
    """ Inputs:
          X: is an NxD matrix (N observations and D dimensions)
          mu: is an KxD matrix (K means and D dimensions)

        Output:
          pair_dist: is the squared pairwise distance matrix (NxK)
    """
    # TODO
    pair_dist = tf.expand_dims(X, -1) - tf.transpose(mu) # returns NxK, pairwise distance
    pair_dist = tf.reduce_sum(tf.square(pair_dist), axis=1) #computes squared pairwise distance
    return pair_dist
```

Figure 1: Distance Function

Given the suggested hyper-parameters over 500 epochs, the Adam optimizer trains the dataset data2D.npy, and produces the validation loss curve in Figure 2 at $K = 3$.

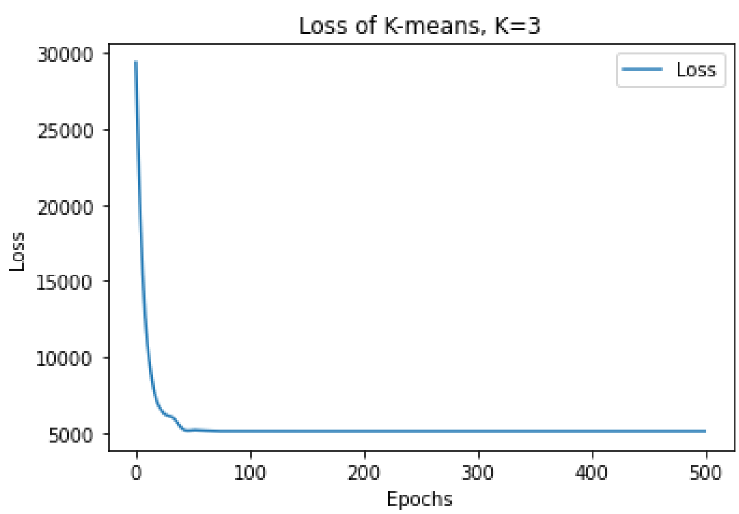

Figure 2: K-Means Validation Loss without Hold-out at K=3

Similarly, with hold-out, Figures 3-7 show the scatter plots of the data2D.npy training data points colored by their cluster assignments with their final validation loss and percentage share of training data points, for $K = 1$ to $K = 5$, inclusive.

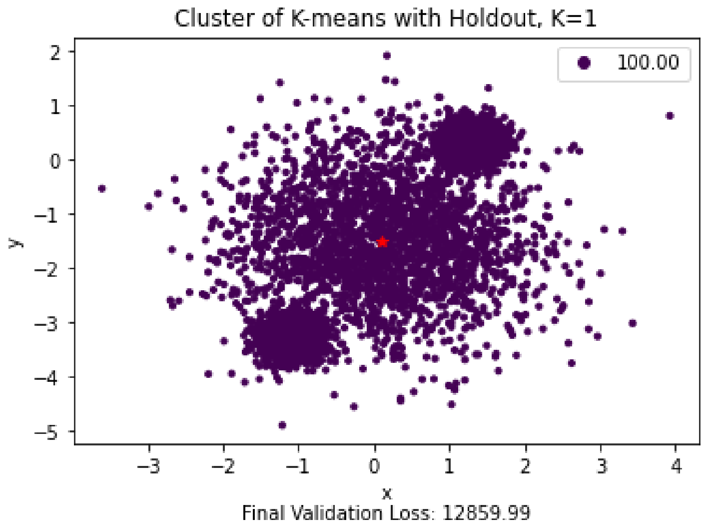

Figure 3: K-Means Validation Loss with Hold-out at K=1

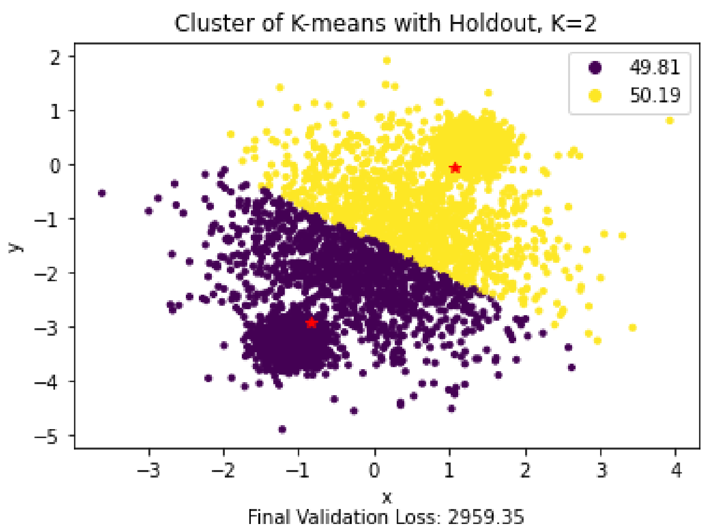

Figure 4: K-Means Validation Loss with Hold-out at K=2

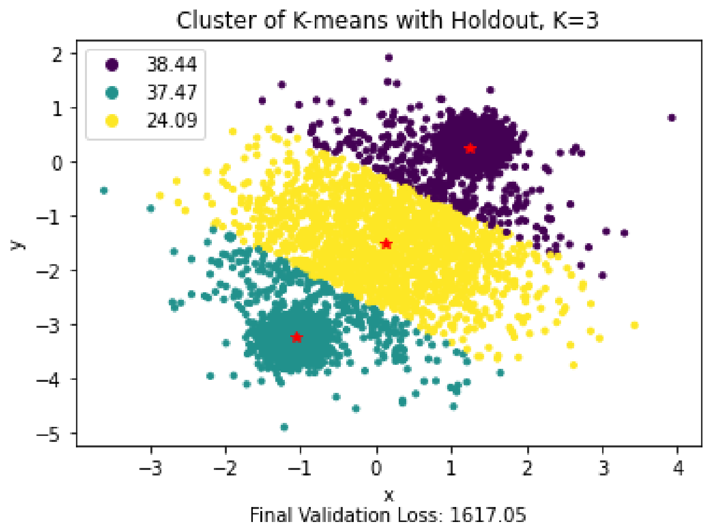

Figure 5: K-Means Validation Loss with Hold-out at K=3

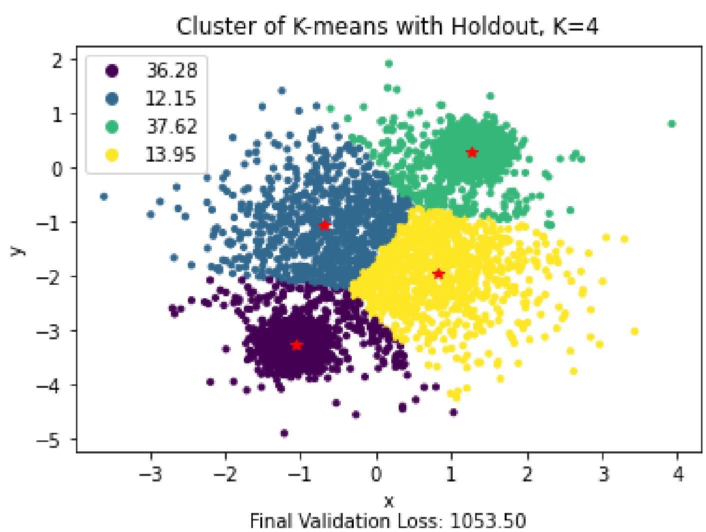

Figure 6: K-Means Validation Loss with Hold-out at K=4

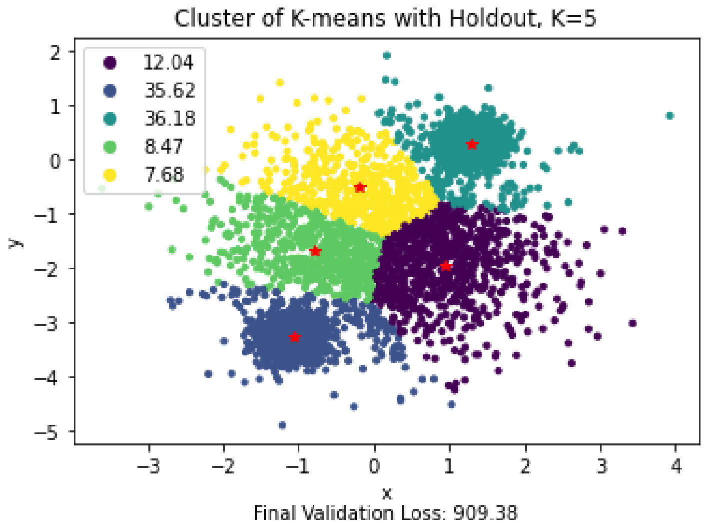

Figure 7: K-Means Validation Loss with Hold-out at K=5

Based on the clusters shown in Figures 3 to 7, $K = 3$ is the best number of clusters to use for this particular dataset. After $K = 3$, it seems that existing clusters are being partitioned into smaller ones. This suggests that the data can be adequately described by 3 clusters, and any excess are evidence of overfitting.

2 Mixtures of Gaussians [16 pt.]

Consider the multivariate normal distribution $\frac{1}{(2\pi)^{\frac{D}{2}}|\sum^{\frac{1}{2}}|}e^{(\frac{1}{2}(x-\mu)^T\sum^{-1}(x-\mu))}$ and by extension, the multivariate normal log distribution $log(\frac{1}{(2\pi)^{\frac{D}{2}}|\sum^{\frac{1}{2}}|}e^{(\frac{1}{2}(x-\mu)^T\sum^{-1}(x-\mu))})$. We simplify the multivariate normal log distribution by separating the two terms as follows: $log(\frac{1}{(2\pi)^{\frac{D}{2}}|\sum^{\frac{1}{2}}|})log(e^{(\frac{1}{2}(x-\mu)^T\sum^{-1}(x-\mu))})$. With some basic arithmetic, we implement $-\frac{D}{2}log(2\pi|\sum|)) - \frac{1}{2}\frac{(x-\mu)^2}{\sum}$, as shown in Figure 8.

```python
def log_gauss_pdf(X, mu, sigma):
    """ Inputs:
            X: N X D
            mu: K X D
            sigma: K X 1

        Outputs:
            log Gaussian PDF (N X K)
    """

    # TODO
    D = tf.cast(tf.rank(X), tf.float32)
    pair_dist = distance_func(X, mu)

    # factored 1/2 from log exponent
    left = -1 * D/2 * tf.log(2 * np.pi * tf.square(sigma))
    right = -1 * 1/2 * pair_dist / tf.square(sigma)

    return left + right
```

Figure 8: Log Gauss PDF Implementation

The log probability of the cluster variable z given the data vector x is $logP(z|x) = log(\frac{P(X|z)P(z)}{P(X)}) = log(P(X|z)) + log(P(z)) - log(P(x))$ by Baye's Theorem. The first term is our multivariate normal log distribution, the second term is $log(\pi)$, and the third term $log(P(x)) = log(\sum_{k=1}^{K}(log(P(X|z)) + log(P(z)))$. The latter is solved using the provided reduce_logsumexp function, which is required to prevent overflow and underflow issues for small logarithms instead of tf.reduce_sum. The aggregate of these terms is shown in Figure 9.

```python
def log_posterior(log_PDF, log_pi):
    """ Inputs:
            log_PDF: log Gaussian PDF N X K
            log_pi: K X 1

        Outputs
            log_post: N X K
    """

    # TODO
    return log_PDF + tf.transpose(log_pi) - reduce_logsumexp(log_PDF + tf.transpose(log_pi), reduction_indices=0, keep_dims=True)
```

Figure 9: Log Probability of the Cluster Variable z Implementation

Learning the data for data2D.npy, and $K = 3$, we can see the validation loss over 500 epochs in Figure 10. The best model parameters learned can be reviewed in Figure 11.

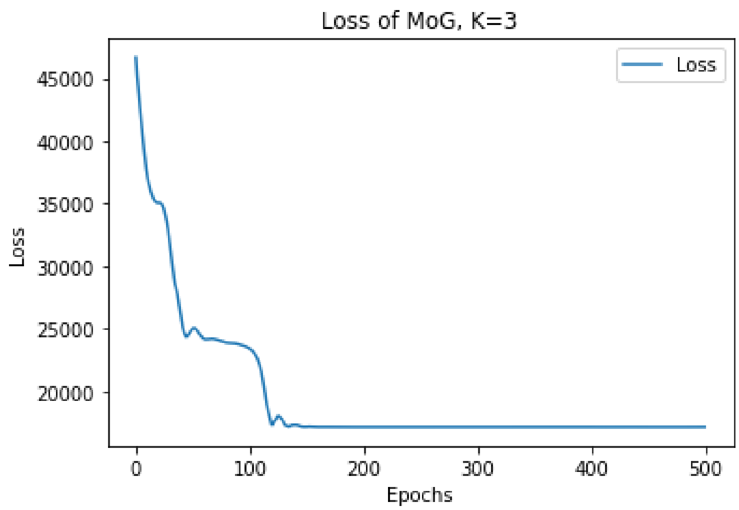

Figure 10: MoG Validation Loss without Hold-out at K=3

```
u =  [[ 0.10601281 -1.5273387 ]
[ 1.2985541   0.3090567 ]
[-1.1017283  -3.306177  ]]
sigma =  [0.9935729  0.1970875  0.19770154]
pi =  [[-1.0946906]
[-1.0982943]
[-1.1028692]]
```

Figure 11: Best Model Parameters Learned

Similarly, with hold-out, Figures 12-16 show the scatter plots of the data2D.npy training data points colored by their cluster assignments with their final validation loss and percentage share of training data points, for $K = 1$ to $K = 5$, inclusive.

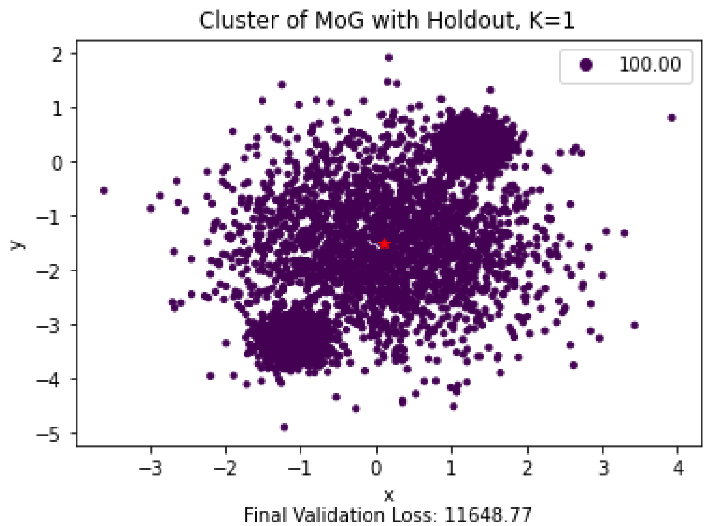

Figure 12: MoG Validation Loss with Hold-out at K=1

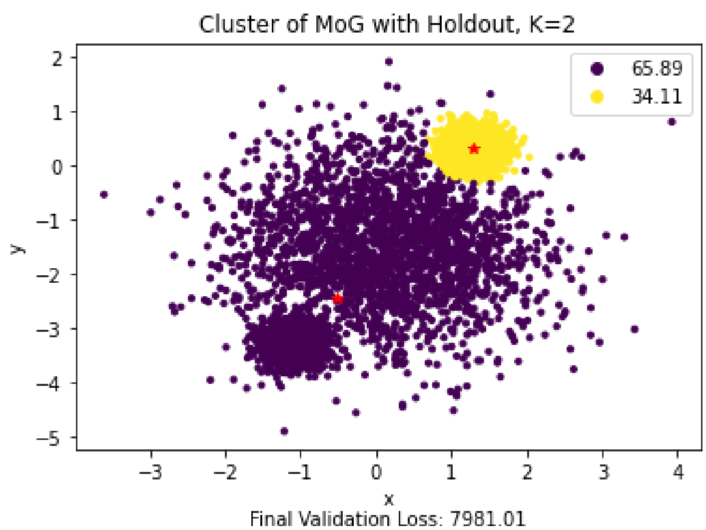

Figure 13: MoG Validation Loss with Hold-out at K=2

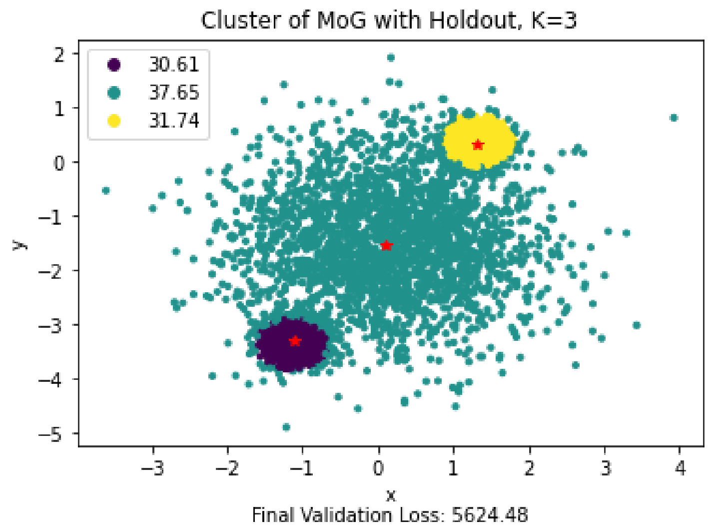

Figure 14: MoG Validation Loss with Hold-out at K=3

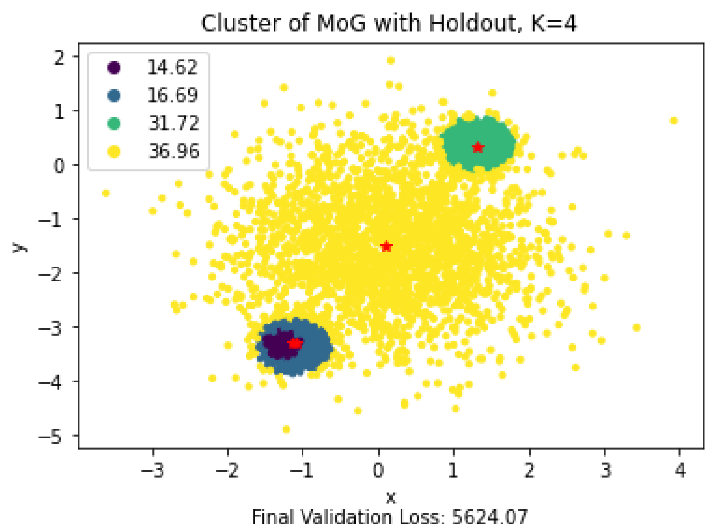

Figure 15: MoG Validation Loss with Hold-out at K=4

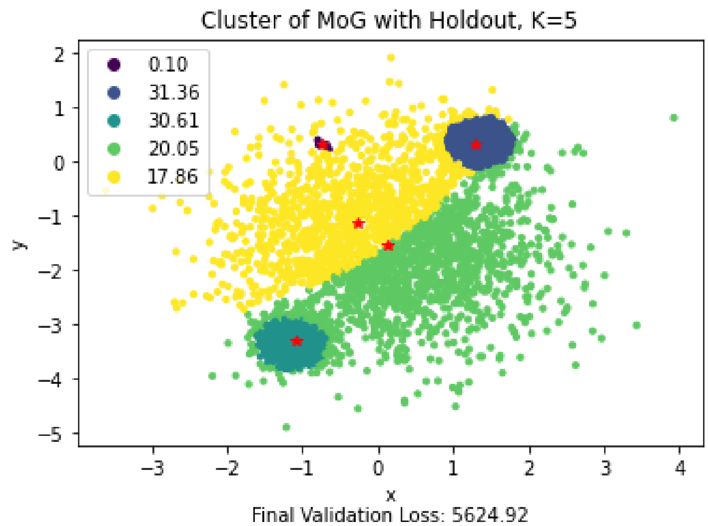

Figure 16: MoG Validation Loss with Hold-out at K=5

Again, $K = 3$ is the best number of clusters for two reasons. The first being that the validation loss plateaus after $K = 3$, and the second being that the dataset seems to be learned within 3 clusters. After 3 clusters, we have an uneven distribution of clusters, which are partitioned from existing ones. When we run the K-means and MoG learning algorithms again on data100D.npy for $K = 5, 10, 15, 20, 30$ over 500 epochs, we see results as shown in Figure 17.

```
K-means Final Validation Loss for K=5: 123543.875
K-means Final Validation Loss for K=10: 70459.0546875
K-means Final Validation Loss for K=15: 69626.1484375
K-means Final Validation Loss for K=20: 68620.015625
K-means Final Validation Loss for K=30: 68191.6796875
MoG Final Validation Loss for K=5: 22485.279296875
MoG Final Validation Loss for K=10: 22485.228515625
MoG Final Validation Loss for K=15: 22485.70703125
MoG Final Validation Loss for K=20: 22485.333984375
MoG Final Validation Loss for K=30: 22485.265625
```

Figure 17: K-Means and MoG Validation Loss for data100D

There is a significant drop in validation loss for K-Means between $K = 5$, and $K = 10$, which suggests that there are between 6 to 10 clusters in the data100D.npy dataset. On the other hand, because the validation loss changes negligibly after $K = 5$, the validation loss curves plateaus well before 500 epochs, and there is in fact a change in validation loss for some $K < 5$, we can conclude that there are < 5 clusters in the data100D.npy. This behavior is quite strange, but I believe the K-Means algorithm to be the most correctly implemented, so between 6 to 10 clusters would be my choice.

3 Code

```
# -*- coding: utf-8 -*-
"""ECE421 Assignment 3.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/18-E139AhTx_0pGrJwT14-
        vYYap0xhxuh
"""

#!pip install tensorflow==1.14.0
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)

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
```

```python
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=0, keep_dims=True)

# Distance function for K-means and GMM
def distance_func(X, mu):
    """ Inputs:
          X: is an NxD matrix (N observations and D dimensions)
          mu: is an KxD matrix (K means and D dimensions)

        Output:
          pair_dist: is the squared pairwise distance matrix (NxK)
    """
    # TODO
    pair_dist = tf.expand_dims(X, -1) - tf.transpose(mu) # returns NxK,
        pairwise distance
    pair_dist = tf.reduce_sum(tf.square(pair_dist), axis=1) #computes
        squared pairwise distance
    return pair_dist


def log_gauss_pdf(X, mu, sigma):
    """ Inputs:
            X: N X D
            mu: K X D
            sigma: K X 1

        Outputs:
            log Gaussian PDF (N X K)
    """

    # TODO
    D = tf.cast(tf.rank(X), tf.float32)
    pair_dist = distance_func(X, mu)
```

```
    # factored 1/2 from log exponent
    left = -1 * D/2 * tf.log(2 * np.pi * tf.square(sigma))
    right = -1 * 1/2 * pair_dist / tf.square(sigma)

    return left + right


def log_posterior(log_PDF, log_pi):
    """ Inputs:
            log_PDF: log Gaussian PDF N X K
            log_pi: K X 1

        Outputs
            log_post: N X K
    """

    # TODO
    return log_PDF + tf.transpose(log_pi) - reduce_logsumexp(log_PDF +
        tf.transpose(log_pi), reduction_indices=0, keep_dims=True)

def KNN(dataset, is_valid, K, epochs, plot):
    # Loading data
    if(dataset == 2):
        data = np.load('/content/gdrive/My_Drive/ECE421/data2D.npy')
    else:
        data = np.load('/content/gdrive/My_Drive/ECE421/data100D.npy')

    [num_pts, dim] = np.shape(data)

    # For Validation set
    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:]]

    tf.set_random_seed(421)
    opt_data = tf.placeholder(tf.float32, shape=(None, dim), name="
        train_data")
    mu = tf.Variable(tf.random.normal(shape=[K, dim]), name="mu")

    pair_dist = distance_func(opt_data, mu)
    loss = tf.reduce_sum(tf.reduce_min(pair_dist, axis=1))
```

```
optimizer = tf.train.AdamOptimizer(learning_rate=0.1, beta1=0.9,
    beta2=0.99, epsilon=1e-5).minimize(loss)

loss_arr = []
valid_loss = []
cluster_arr = []
init = tf.global_variables_initializer()

#inspiration from https://www.altoros.com/blog/using-k-means-
    clustering-in-tensorflow/

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

    for i in range(epochs):
        sess.run(optimizer, feed_dict={opt_data: data})
        loss_val = sess.run(loss, feed_dict={opt_data: data})
        loss_arr.append(loss_val)

        if is_valid:
            sess.run(optimizer, feed_dict={opt_data: val_data})
            val_loss = sess.run(loss, feed_dict={opt_data: val_data
                })
            valid_loss.append(val_loss)

    cluster_arr = mu.eval()
    dist = sess.run(pair_dist, feed_dict={opt_data: data})

#find cluster
cluster = np.argmin(dist, axis=1)

#find percentage of points
points = np.zeros(K)
for i in range(K):
    points[i] = np.sum(i == cluster)/len(cluster)*100.0

#plot loss
if is_valid and plot:
    plt.figure()
    plt.title('Loss of K-means with Holdout, K=%i' %K)
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.plot(loss_arr, label="Loss")
    plt.legend()
```

```
        fig = plt.figure()
        plt.title('Cluster of K-means with Holdout, K=%i' %K)
        percentage = [f'{points[i]:.2f}' for i in range(K)]
        scatter = plt.scatter(data[:,0], data[:,1], c=cluster, s=10)
        plt.plot(cluster_arr[:, 0], cluster_arr[:, 1], '*r')
        plt.legend(handles=scatter.legend_elements()[0], labels=
            percentage)
        plt.xlabel('x\nFinal Validation Loss: %.2f' %valid_loss[-1])
        plt.ylabel("y")

    elif plot:
        plt.figure()
        plt.title('Loss of K-means, K=%i' %K)
        plt.xlabel("Epochs")
        plt.ylabel("Loss")
        plt.plot(loss_arr, label="Loss")
        plt.legend()

    else:
        print("K-means Final Validation Loss for K={}: {}".format(K,
            valid_loss[-1]))
        plt.figure()
        plt.title('Loss of K-means with Holdout, K=%i' %K)
        plt.xlabel("Epochs")
        plt.ylabel("Loss")
        plt.plot(loss_arr, label="Loss")
        plt.legend()

def MoG(dataset, is_valid, K, epochs, plot):
    # Loading data
    if(dataset == 2):
        data = np.load('/content/gdrive/My Drive/ECE421/data2D.npy')
    else:
        data = np.load('/content/gdrive/My Drive/ECE421/data100D.npy')

    [num_pts, dim] = np.shape(data)

    # For Validation set
    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:]]
```

```
tf.set_random_seed(421)

opt_data = tf.placeholder(tf.float32, shape=(None, dim))
mu = tf.Variable(tf.random.normal(shape=[K, dim]))
sigma = tf.squeeze(tf.exp(tf.Variable(tf.random.normal(shape=[K,
    1]))))
log_pi = logsoftmax(tf.Variable(tf.random.normal(shape=[K, 1])))
log_PDF = log_gauss_pdf(opt_data, mu, sigma)
log_post = log_posterior(log_PDF, log_pi)
clusters = tf.argmax(log_post, axis=1)
loss = (-1) * tf.reduce_sum(reduce_logsumexp(log_PDF + tf.transpose
    (log_pi)))

optimizer = tf.train.AdamOptimizer(learning_rate=0.1, beta1=0.9,
    beta2=0.99, epsilon=1e-5).minimize(loss)

loss_arr = []
valid_loss = []
cluster_arr = []
init = tf.global_variables_initializer()

#inspiration from https://www.altoros.com/blog/using-k-means-
    clustering-in-tensorflow/



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

    for i in range(epochs):
        cluster, _ = sess.run([clusters, optimizer], feed_dict={
            opt_data: data})
        loss_val = sess.run(loss, feed_dict={opt_data: data})
        loss_arr.append(loss_val)

        if is_valid:
            sess.run(optimizer, feed_dict={opt_data: val_data})
            val_loss = sess.run(loss, feed_dict={opt_data: val_data
                })
            valid_loss.append(val_loss)

    cluster_arr = mu.eval()
    [mu, sigma, pi] = sess.run([mu, sigma, log_pi], feed_dict={})

#find percentage of points
points = np.zeros(K)
```

```
    for i in range(K):
        points[i] = np.sum(i == cluster)/len(cluster)*100.0

    #plot loss
    if is_valid and plot:
        plt.figure()
        plt.title('Loss of MoG with Holdout, K=%i' %K)
        plt.xlabel("Epochs")
        plt.ylabel("Loss")
        plt.plot(loss_arr, label="Loss")
        plt.legend()

        fig = plt.figure()
        plt.title('Cluster of MoG with Holdout, K=%i' %K)
        percentage = [f'{points[i]:.2f}' for i in range(K)]
        scatter = plt.scatter(data[:,0], data[:,1], c=cluster, s=10)
        plt.plot(cluster_arr[:, 0], cluster_arr[:, 1], '*r')
        plt.legend(handles=scatter.legend_elements()[0], labels=
            percentage)
        plt.xlabel('x\nFinal Validation Loss: %.2f' %valid_loss[-1])
        plt.ylabel("y")

    elif plot:
        print(" u = ", mu)
        print(" sigma = ", sigma)
        print(" pi = ", pi)

        plt.figure()
        plt.title('Loss of MoG, K=%i' %K)
        plt.xlabel("Epochs")
        plt.ylabel("Loss")
        plt.plot(loss_arr, label="Loss")
        plt.legend()

    else:
        print("MoG Final Validation Loss for K={}: {}".format(K,
            valid_loss[-1]))
        plt.figure()
        plt.title('Loss of MoG with Holdout, K=%i' %K)
        plt.xlabel("Epochs")
        plt.ylabel("Loss")
        plt.plot(loss_arr, label="Loss")
        plt.legend()

#K-means with K=3
KNN(2, False, 3, 500, True)
```

```
#K-means with K=1,2,3,4,5 and hold out 1/3 valid data
KNN(2, True, 1, 500, True)
KNN(2, True, 2, 500, True)
KNN(2, True, 3, 500, True)
KNN(2, True, 4, 500, True)
KNN(2, True, 5, 500, True)

plt.show()

#MoG with K=3
MoG(2, False, 3, 500, True)
#MoG with K=1,2,3,4,5 and hold out 1/3 valid data
MoG(2, True, 1, 500, True)
MoG(2, True, 2, 500, True)
MoG(2, True, 3, 500, True)
MoG(2, True, 4, 500, True)
MoG(2, True, 5, 500, True)

plt.show()

KNN(100, True, 5, 500, False)
KNN(100, True, 10, 500, False)
KNN(100, True, 15, 500, False)
KNN(100, True, 20, 500, False)
KNN(100, True, 30, 500, False)

MoG(100, True, 5, 500, False)
MoG(100, True, 10, 500, False)
MoG(100, True, 15, 500, False)
MoG(100, True, 20, 500, False)
MoG(100, True, 30, 500, False)

plt.show()
```