ECE421 - Winter 2022
Assignment 3:
Unsupervised Learning and Probabilistic Models

Faranak Dayyani, Student Number: 1002373674

Part 1: K-Means

1.1: Learning K-Means

1. 200 iterations were used to train the data for dataset data2D.npy. The cluster centers were initialized with Gaussian distribution with standard deviation of 1.0. The graph below shows the loss vs. number of updates with K = 3:

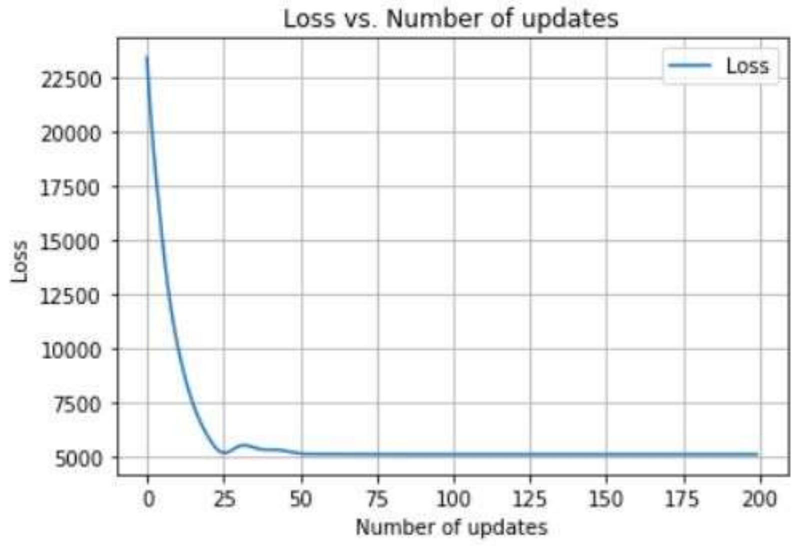
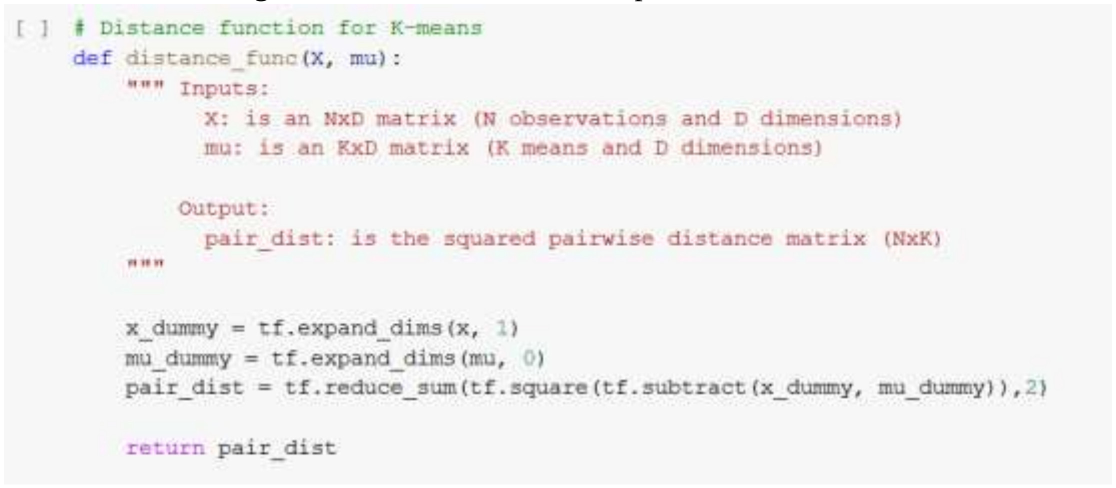
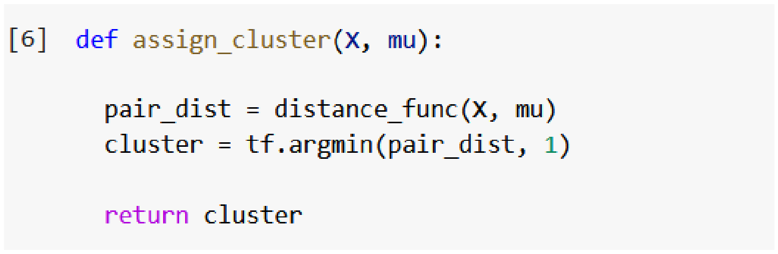


Figure 1: Loss vs. Number of Updates for K-Means



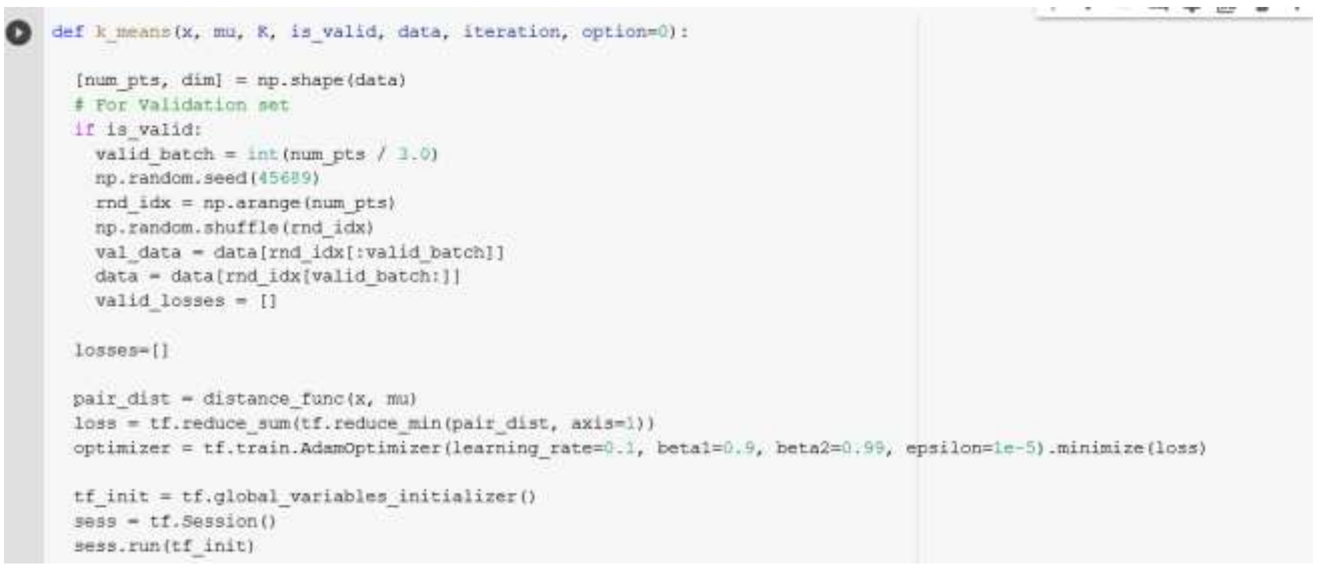
```
[ ] # Distance function for K-means
    def distance_func(X, mu):
        """ Inputs:
              X: is an NxD matrix (N observations and D dimensions)
              mu: is an KxD matrix (K means and D dimensions)

            Output:
              pair_dist: is the squared pairwise distance matrix (NxK)
        """

        x_dummy = tf.expand_dims(x, 1)
        mu_dummy = tf.expand_dims(mu, 0)
        pair_dist = tf.reduce_sum(tf.square(tf.subtract(x_dummy, mu_dummy)),2)

        return pair_dist
```

Figure 2: Code snippet of the distance_func(X, mu)



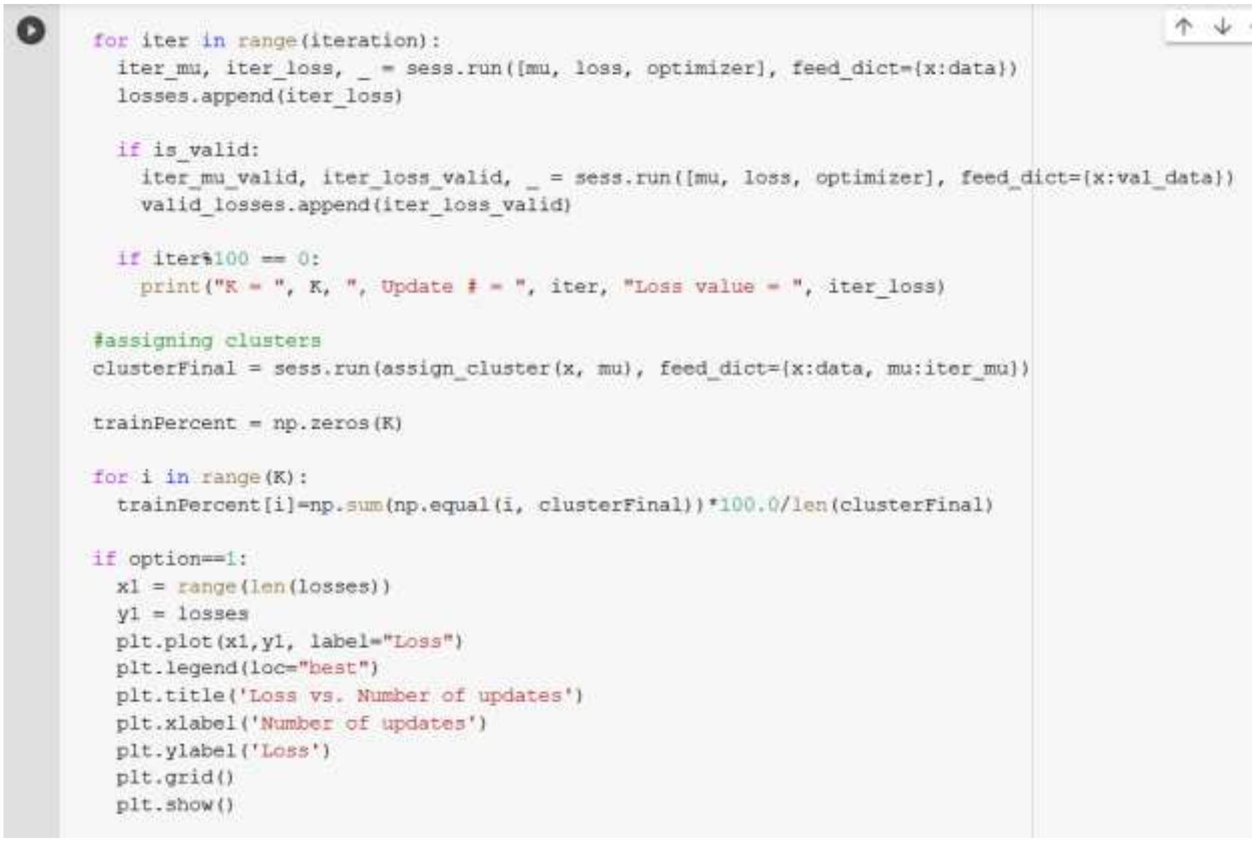
```
[6] def assign_cluster(X, mu):

      pair_dist = distance_func(X, mu)
      cluster = tf.argmin(pair_dist, 1)

      return cluster
```

Figure 3: Code snippet of assign_cluster (X, mu) function



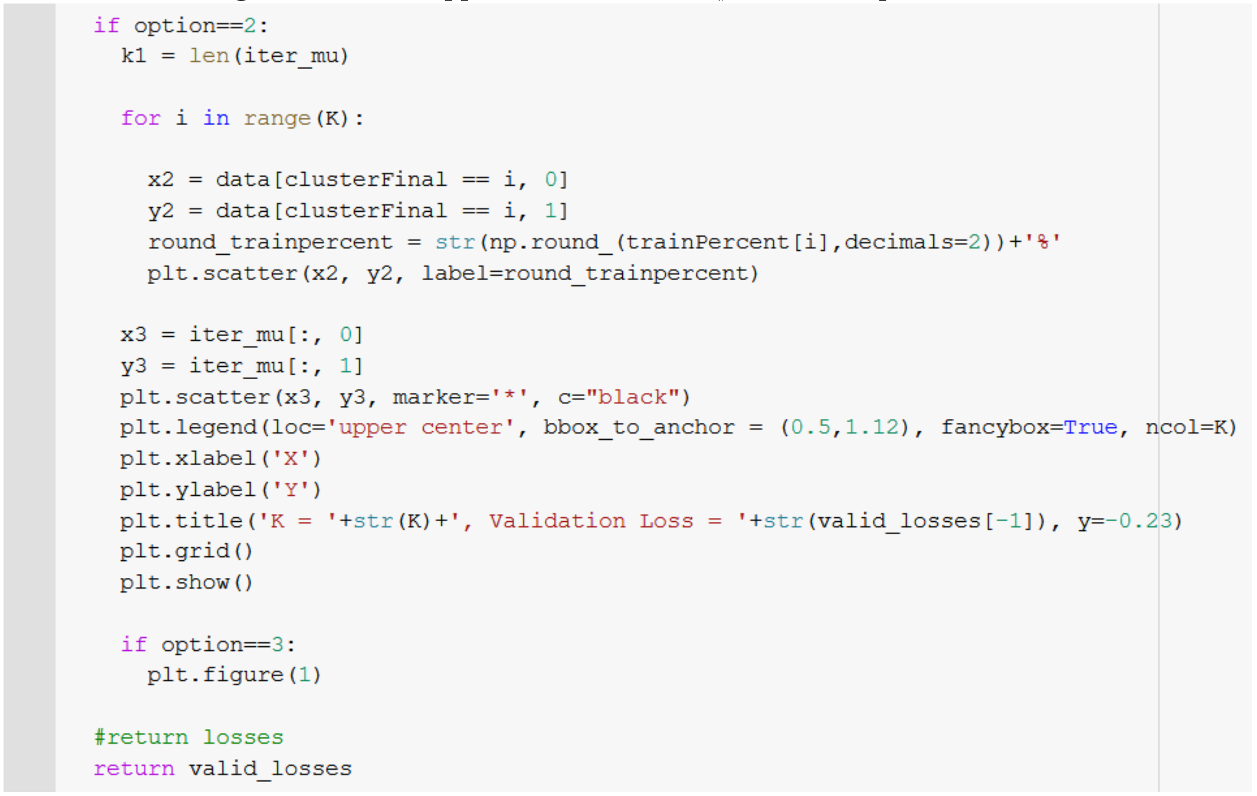
```
def k_means(x, mu, K, is_valid, data, iteration, option=0):

  [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:]]
    valid_losses = []

  losses=[]

  pair_dist = distance_func(x, 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)

  tf_init = tf.global_variables_initializer()
  sess = tf.Session()
  sess.run(tf_init)
```

Figure 4: Code snippet of the K_means() function – part 1



```
for iter in range(iteration):
  iter_mu, iter_loss, _ = sess.run([mu, loss, optimizer], feed_dict={x:data})
  losses.append(iter_loss)

  if is_valid:
    iter_mu_valid, iter_loss_valid, _ = sess.run([mu, loss, optimizer], feed_dict={x:val_data})
    valid_losses.append(iter_loss_valid)

  if iter%100 == 0:
    print("K = ", K, ", Update # = ", iter, "Loss value = ", iter_loss)

#assigning clusters
clusterFinal = sess.run(assign_cluster(x, mu), feed_dict={x:data, mu:iter_mu})

trainPercent = np.zeros(K)

for i in range(K):
  trainPercent[i]=np.sum(np.equal(i, clusterFinal))*100.0/len(clusterFinal)

if option==1:
  x1 = range(len(losses))
  y1 = losses
  plt.plot(x1,y1, label="Loss")
  plt.legend(loc="best")
  plt.title('Loss vs. Number of updates')
  plt.xlabel('Number of updates')
  plt.ylabel('Loss')
  plt.grid()
  plt.show()
```

Figure 5: Code snippet of the K_means() function – part 2



```
if option==2:
  k1 = len(iter_mu)

  for i in range(K):

    x2 = data[clusterFinal == i, 0]
    y2 = data[clusterFinal == i, 1]
    round_trainpercent = str(np.round_(trainPercent[i],decimals=2))+'%'
    plt.scatter(x2, y2, label=round_trainpercent)

  x3 = iter_mu[:, 0]
  y3 = iter_mu[:, 1]
  plt.scatter(x3, y3, marker='*', c="black")
  plt.legend(loc='upper center', bbox_to_anchor = (0.5,1.12), fancybox=True, ncol=K)
  plt.xlabel('X')
  plt.ylabel('Y')
  plt.title('K = '+str(K)+', Validation Loss = '+str(valid_losses[-1]), y=-0.23)
  plt.grid()
  plt.show()

  if option==3:
    plt.figure(1)

#return losses
return valid_losses
```

Figure 6: Code snippet of the K_means() function – part 3

2. In this section, the scatter plots for K values of 1,2,3,4 and 5 are shown below for the K-Means model:

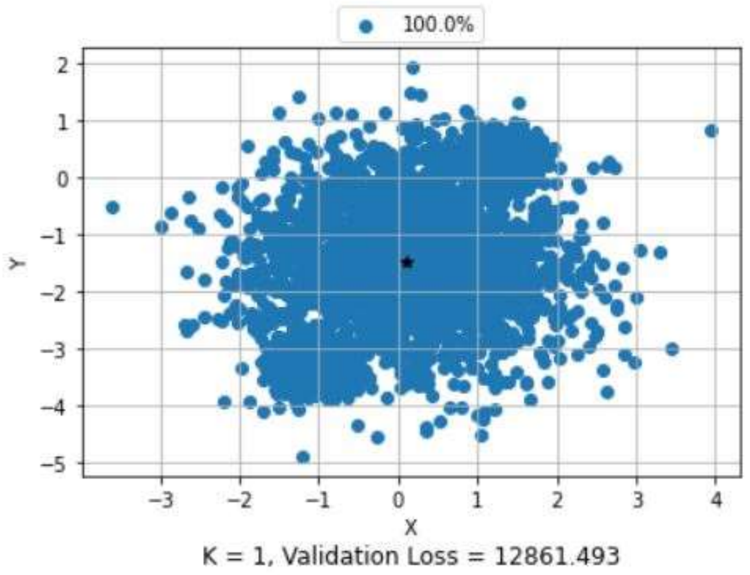


Figure 7: Scatter plot for K = 1

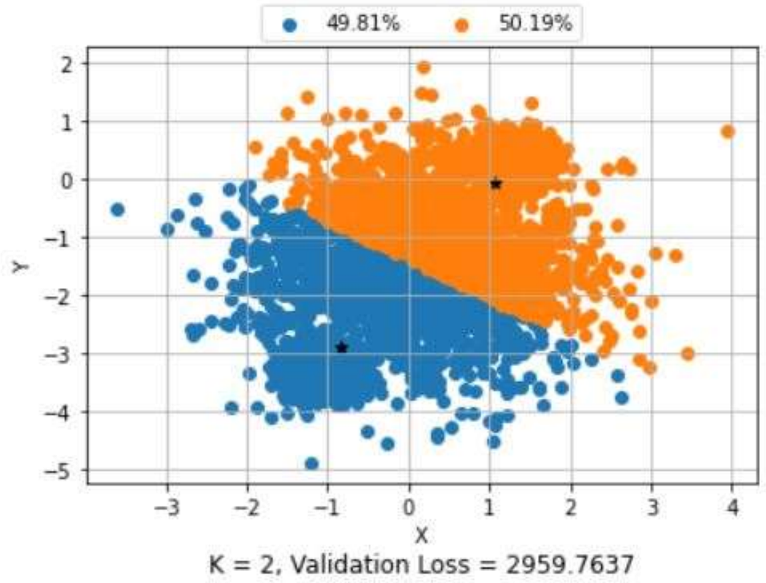


Figure 8: Scatter plot for K = 2

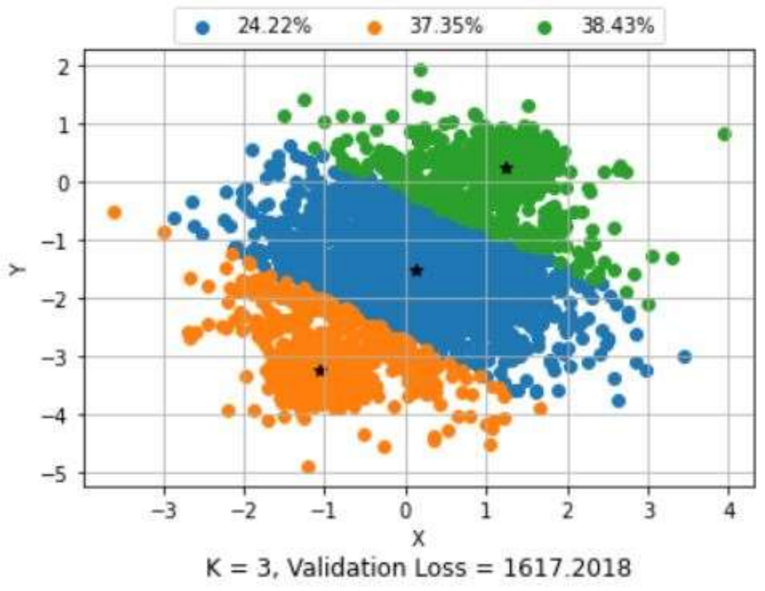


Figure 9: Scatter plot for K = 3

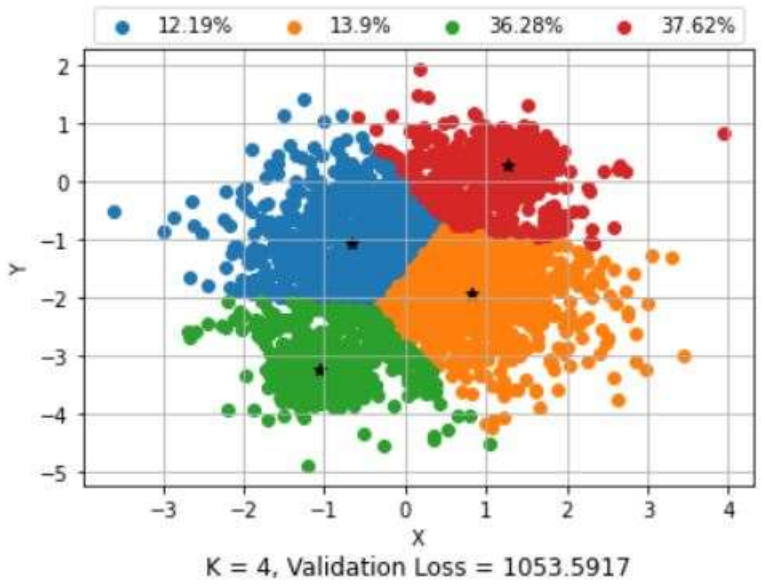


Figure 10: Scatter plot for K = 4

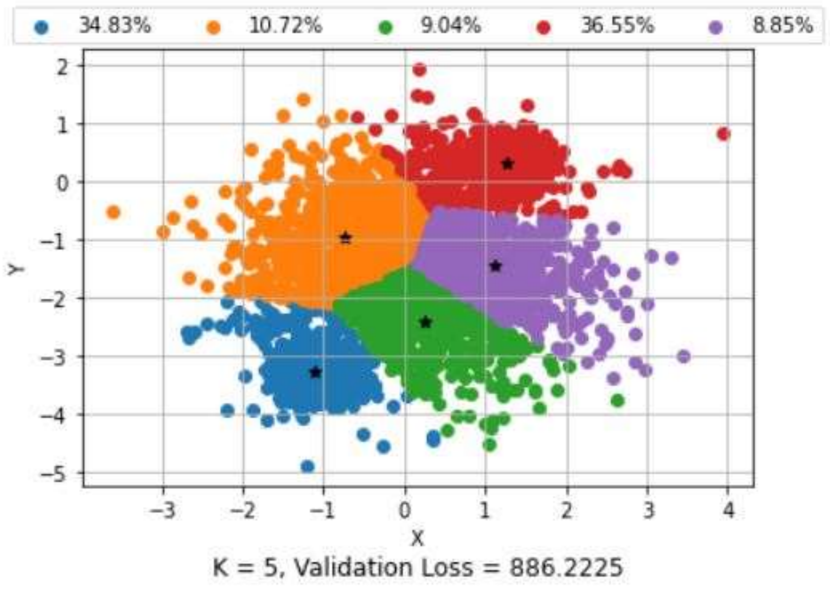


Figure 11: Scatter plot for K = 5

Figures 7-11 above illustrate the 2D scatter plots for different values of K with training data points colored by their cluster assignments. The percentage of the training data points that belong to each cluster assignment are provided in the legend of each plot located at the upper center. The validation loss for each scatter plot is also displayed in the caption and below each plot.

By observing the distribution data for each cluster in each of the scatter plots for different values of K, it is evident that the distribution of data (in percentage) is mostly balanced till K=3. After increasing the K value to 4 and 5, it can be seen that some of the clusters have more data than the others.

As a general rule, more number of clusters always gives smaller loss values. When K equals the number of data, the loss can be equal to 0. Therefore, the decision on the best number of clusters to use cannot be made solely by looking at the lowest loss value.

The trend of decrease of loss with increasing value of K can be used to decide the best number of clusters (K). It can be seen that the amount of decline in loss from when K=2 to K=3 and onwards (figures 8-11), significantly drops. Thus, from the observations above, K = 3 is the best choice for the number of clusters to use.

Part 2: Mixtures of Gaussians

2.1: The Gaussian cluster mode

1. Log probability of density function for cluster k:

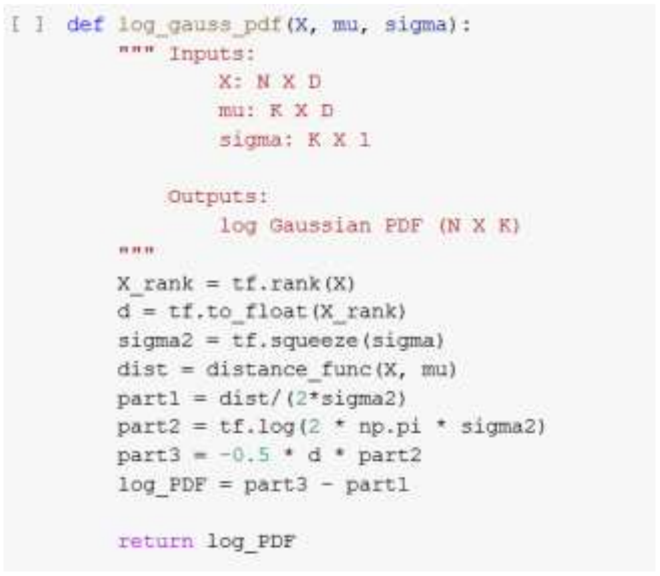
$$P(x|\mu_k, \sigma_k^2) = N(x;\ \mu_k, \sigma_k^2)$$

$$\log P(x|\mu_k, \sigma_k^2) = logN(x;\ \mu_k, \sigma_k^2)$$

$$\log P(x|\mu_k, \sigma_k^2) = \log\left(\frac{1}{\sqrt{(2\pi)^d\sigma_k^2}} e^{\left(\frac{-||x-\mu_k|\ \ |_2^2}{2\sigma_k^2}\right)}\right)$$

$$\log P(x|\mu_k, \sigma_k^2) = \log\left(\frac{1}{\sqrt{(2\pi)^d\sigma_k^2}}\right) - \frac{||x-\mu_k||_2^2}{2\sigma_k^2}$$

$$\log P(x|\mu_k, \sigma_k^2) = -\frac{1}{2}\log((2\pi)^d\sigma_k^2) - \frac{||x-\mu_k||_2^2}{2\sigma_k^2}$$



```
[ ] 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)
        """
        X_rank = tf.rank(X)
        d = tf.to_float(X_rank)
        sigma2 = tf.squeeze(sigma)
        dist = distance_func(X, mu)
        part1 = dist/(2*sigma2)
        part2 = tf.log(2 * np.pi * sigma2)
        part3 = -0.5 * d * part2
        log_PDF = part3 - part1

        return log_PDF
```

Figure 12: Code snippet of the log_gauss_pdf(X, mu, sigma) function

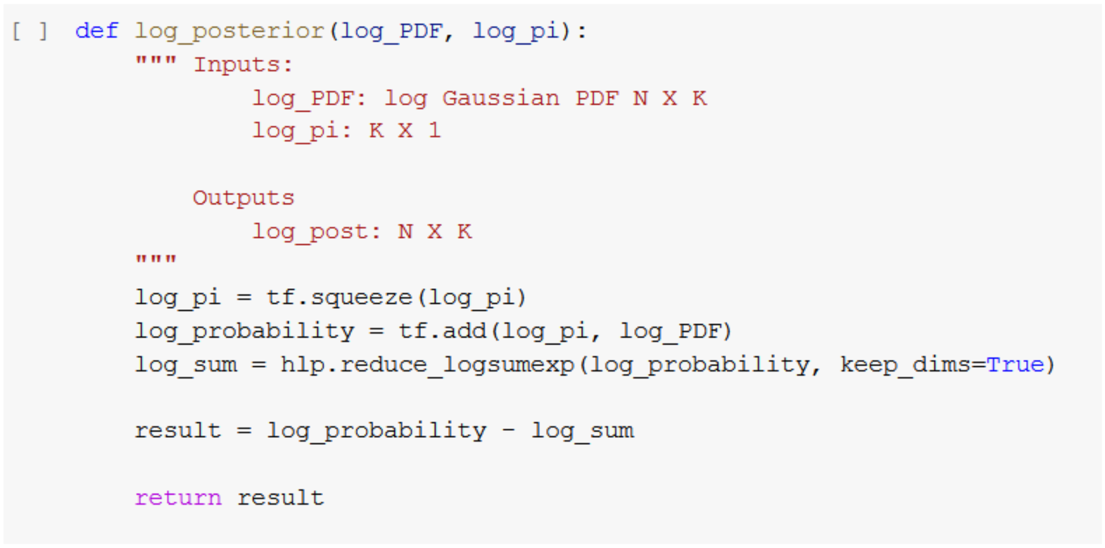
2. Log probability of the cluster variable z given the data vector x: log P (x|z):
   Applying Bayes' rule:

$$P(z=k|x) = \frac{P(x|z=k)P(z=k)}{P(x)}$$

$$P(z=k|x) = \frac{P(x|z=k)P(z=k)}{\sum_i P(x|z=i)\,P(z=i)}$$

$$logP(z=k|x) = \log\left(\frac{P(x|z=k)P(z=k)}{\sum_i P(x|z=i)\,P(z=i)}\right)$$

$$logP(z=k|x) = \log P(x|z=k) + logP(z=k) - log\sum_i e^{logP(x|z\,=\,i)+\log P(z=i)}$$



```
[ ] 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
        """
        log_pi = tf.squeeze(log_pi)
        log_probability = tf.add(log_pi, log_PDF)
        log_sum = hlp.reduce_logsumexp(log_probability, keep_dims=True)

        result = log_probability - log_sum

        return result
```

Figure 13: Code snippet of the log_posterior(log_PDF, log_pi) function

The log_gauss_pdf function defined in part 2.1.1 was used in order to calculate the log posterior matrix (log_posterior function above). In order to use the log_gauss_pdf function for implementing the log_posterior function, it is important to use the log-sum-exp function instead of tf.reduce_sum function. The reason behind this is that the exponential function in log-sum-exp function will nullify the log term from the log gaussian matrix. If the function in part 2.1.1 was not used in order to compute the log_posterior function, then it would not be necessary to use the log-sum-exp function.

2.2: Learning the MoG

1. 250 iterations were used to train the data for dataset data2D.npy. All parameters were initialized by sample from standard normal distribution with standard deviation of 1.0. The graph below shows the loss vs. number of updates for MoG with K = 3:

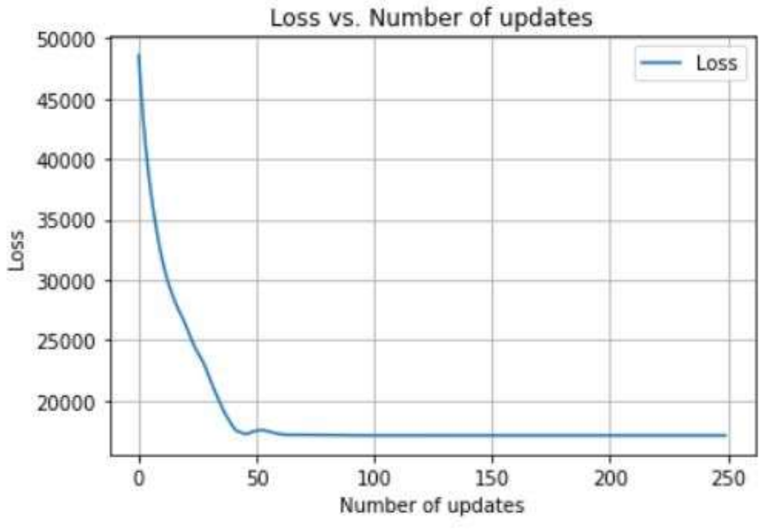
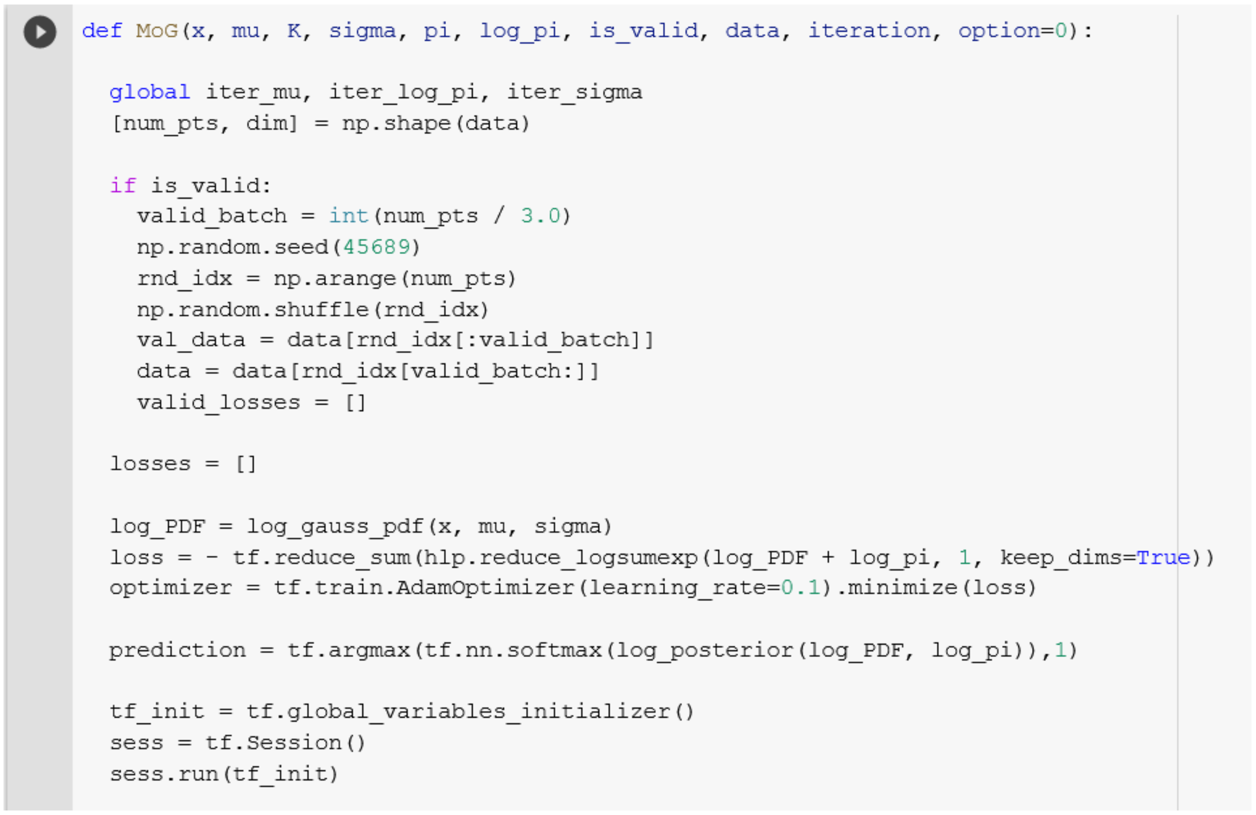
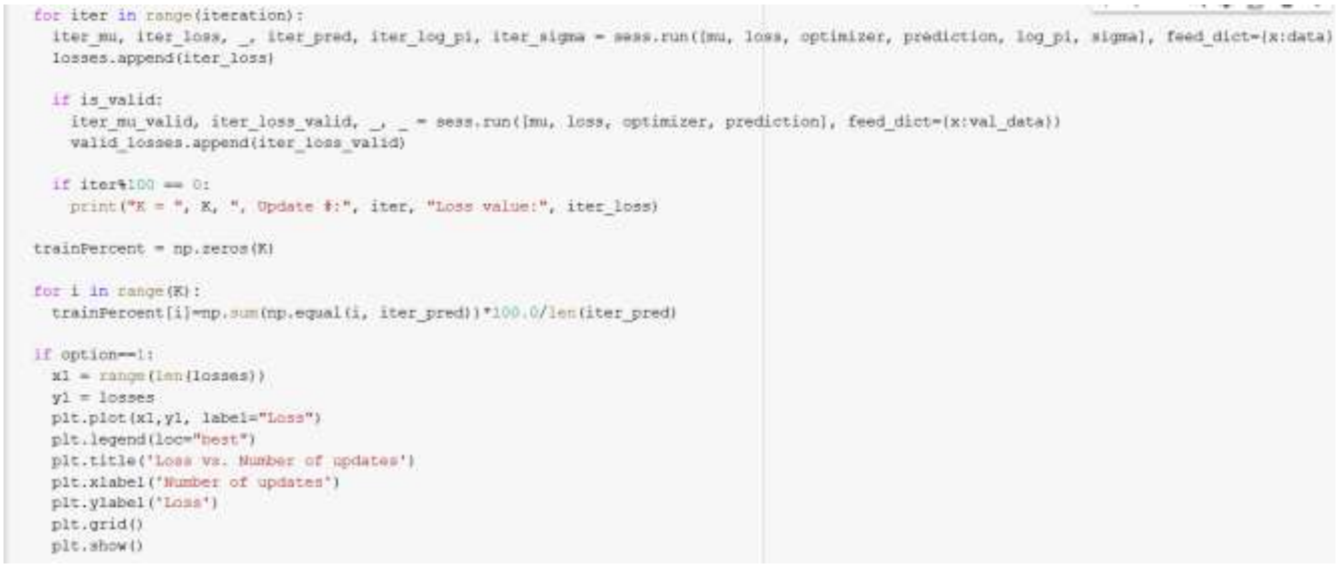


Figure 14: Loss vs. Number of Updates for MoG



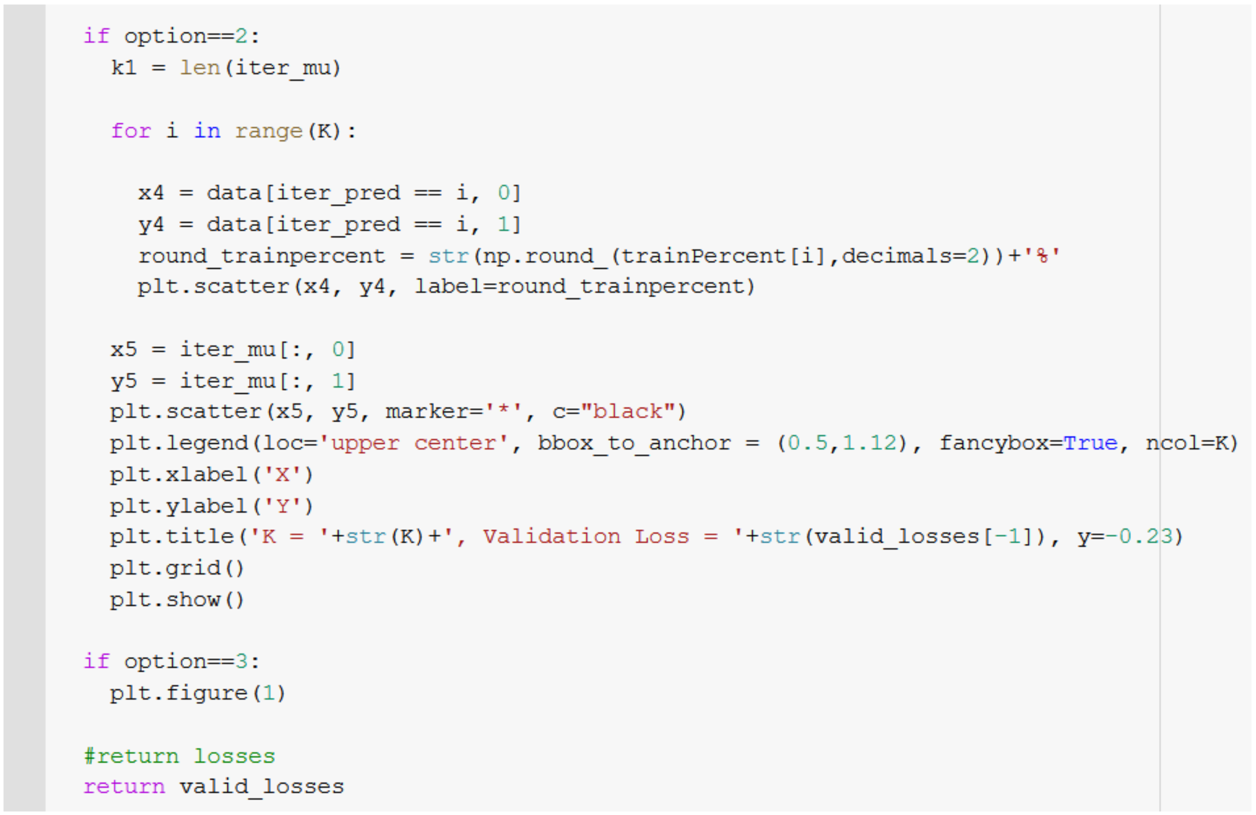
```python
def MoG(x, mu, K, sigma, pi, log_pi, is_valid, data, iteration, option=0):

  global iter_mu, iter_log_pi, iter_sigma
  [num_pts, dim] = np.shape(data)

  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:]]
    valid_losses = []

  losses = []

  log_PDF = log_gauss_pdf(x, mu, sigma)
  loss = - tf.reduce_sum(hlp.reduce_logsumexp(log_PDF + log_pi, 1, keep_dims=True))
  optimizer = tf.train.AdamOptimizer(learning_rate=0.1).minimize(loss)

  prediction = tf.argmax(tf.nn.softmax(log_posterior(log_PDF, log_pi)),1)

  tf_init = tf.global_variables_initializer()
  sess = tf.Session()
  sess.run(tf_init)
```

Figure 15: Code snippet of MoG() function – part 1



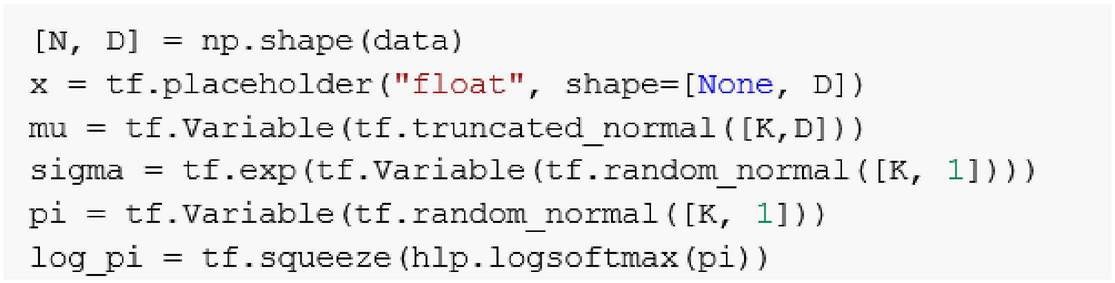
```python
for iter in range(iteration):
  iter_mu, iter_loss, _, iter_pred, iter_log_pi, iter_sigma = sess.run([mu, loss, optimizer, prediction, log_pi, sigma], feed_dict={x:data}
  losses.append(iter_loss)

  if is_valid:
    iter_mu_valid, iter_loss_valid, _, _ = sess.run([mu, loss, optimizer, prediction], feed_dict={x:val_data})
    valid_losses.append(iter_loss_valid)

  if iter%100 == 0:
    print("K = ", K, ", Update #:", iter, "Loss value:", iter_loss)

trainPercent = np.zeros(K)

for i in range(K):
  trainPercent[i]=np.sum(np.equal(i, iter_pred))*100.0/len(iter_pred)

if option==1:
  x1 = range(len(losses))
  y1 = losses
  plt.plot(x1,y1, label="Loss")
  plt.legend(loc="best")
  plt.title('Loss vs. Number of updates')
  plt.xlabel('Number of updates')
  plt.ylabel('Loss')
  plt.grid()
  plt.show()
```

Figure 16: Code snippet of MoG() function – part 2



```
if option==2:
  k1 = len(iter_mu)

  for i in range(K):

    x4 = data[iter_pred == i, 0]
    y4 = data[iter_pred == i, 1]
    round_trainpercent = str(np.round_(trainPercent[i],decimals=2))+'%'
    plt.scatter(x4, y4, label=round_trainpercent)

  x5 = iter_mu[:, 0]
  y5 = iter_mu[:, 1]
  plt.scatter(x5, y5, marker='*', c="black")
  plt.legend(loc='upper center', bbox_to_anchor = (0.5,1.12), fancybox=True, ncol=K)
  plt.xlabel('X')
  plt.ylabel('Y')
  plt.title('K = '+str(K)+', Validation Loss = '+str(valid_losses[-1]), y=-0.23)
  plt.grid()
  plt.show()

if option==3:
  plt.figure(1)

#return losses
return valid_losses
```

Figure 17: Code snippet of MoG() function – part 3



```
[N, D] = np.shape(data)
x = tf.placeholder("float", shape=[None, D])
mu = tf.Variable(tf.truncated_normal([K,D]))
sigma = tf.exp(tf.Variable(tf.random_normal([K, 1])))
pi = tf.Variable(tf.random_normal([K, 1]))
log_pi = tf.squeeze(hlp.logsoftmax(pi))
```

Figure 18: Code snippet of parameter initialization

Table 1: Best learned model parameters:

Cluster	$\boldsymbol{\mu}$	$\boldsymbol{\pi}$	$\boldsymbol{\sigma^2}$
1	[0.10597776 -1.5274241]	-1.0946786	0.9871902
2	[-1.1017281 -3.3061693]	-1.1028792	0.03884992
3	[1.2986314 0.30916542]	-1.098296	0.03908239

The initialization of parameters in figure 17 above show that the constraint for π is enforced by using logofsoftmax() function from the helper.py code. Also, σ^2 is replaced with exp (σ^2).

2. In this section, the scatter plots for K values of 1,2,3,4 and 5 are shown below for the MoG model:

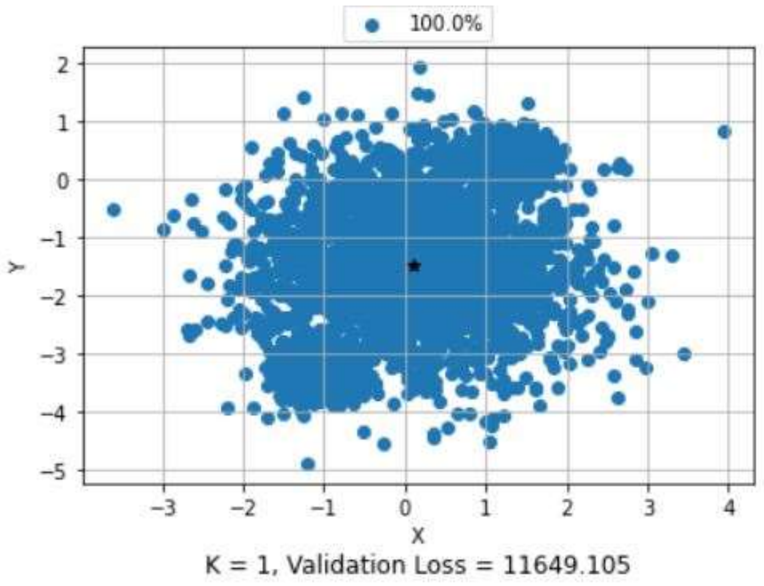


Figure 19: Scatter plot for K = 1

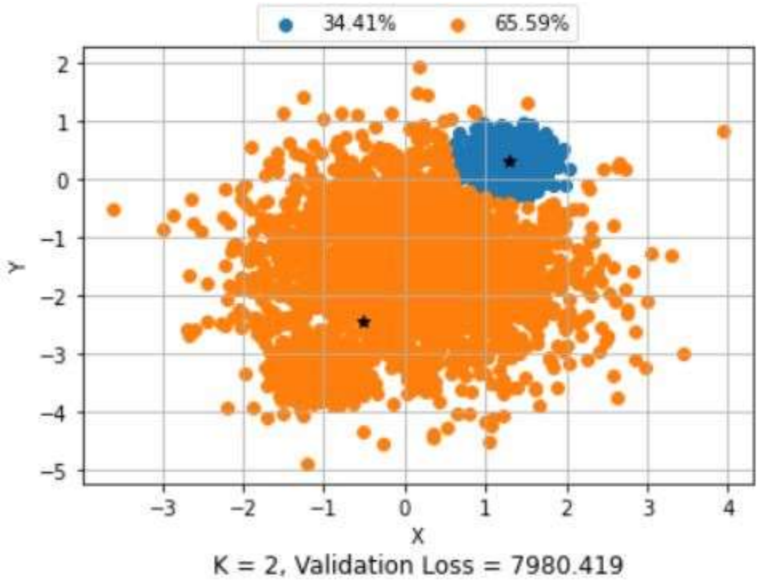


Figure 20: Scatter plot for K = 2

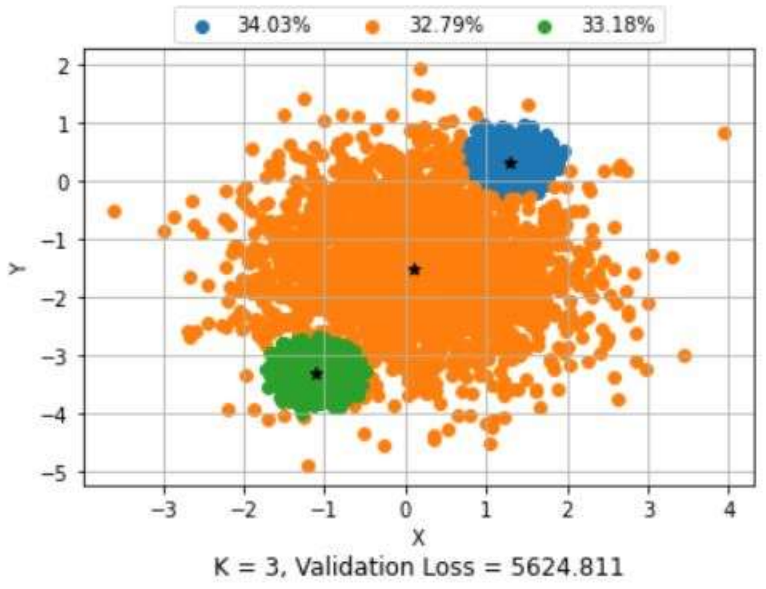


Figure 21: Scatter plot for K = 3

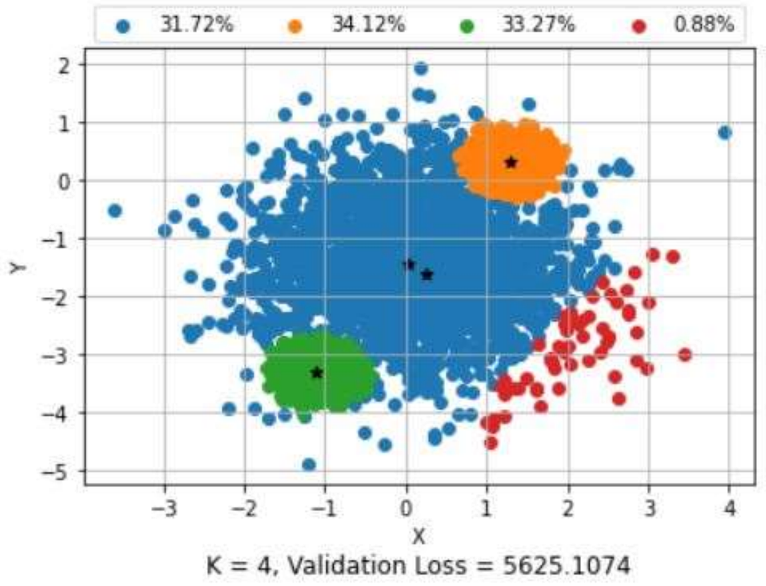


Figure 22: Scatter plot for K = 4

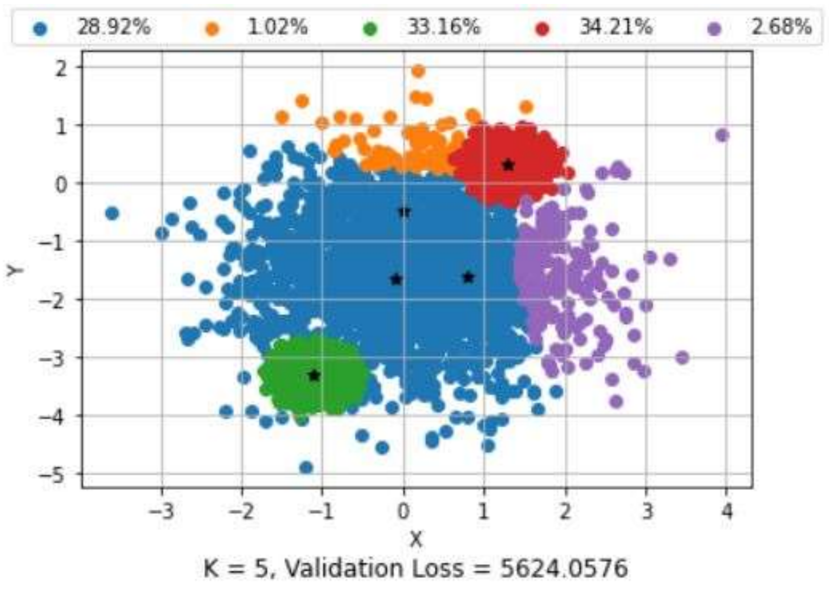


Figure 23: Scatter plot for K = 5

By looking at figures 19-23 above, it is evident that the validation loss decreases as the value of K increases. However, according to figures 20-22, as K increases from 3 to 5, the decrease in validation loss is very small and quite negligible. For instance, the validation loss for K values of 3 to 5 is 5624.811, 5625.1074 and 5624.0576, respectively. Therefore, the best value of K is 3.

3. The plots below shows the loss vs. number of updates for K-Means and MoG models with K values of 5, 10, 15, 20 and 30 for data100D dataset.

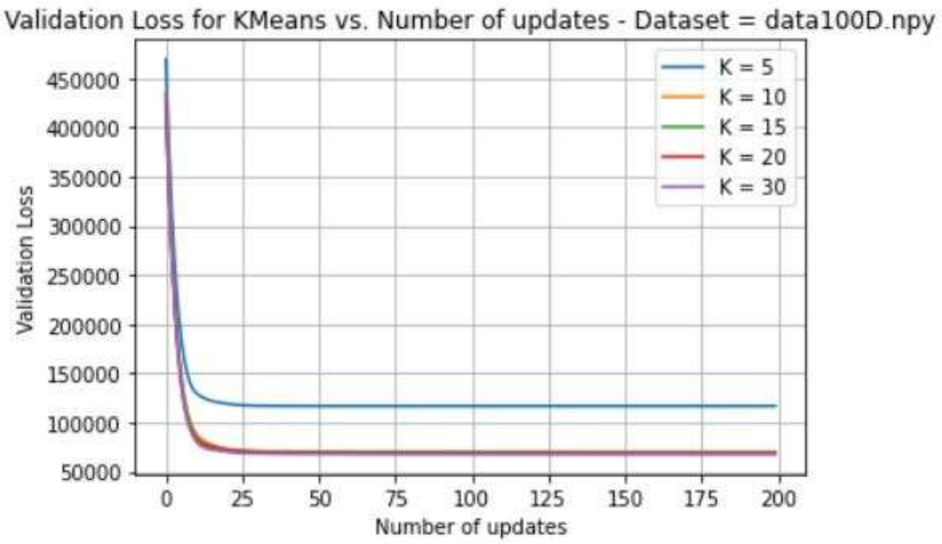


Figure 24: Loss vs. Number of Updates for K-Means

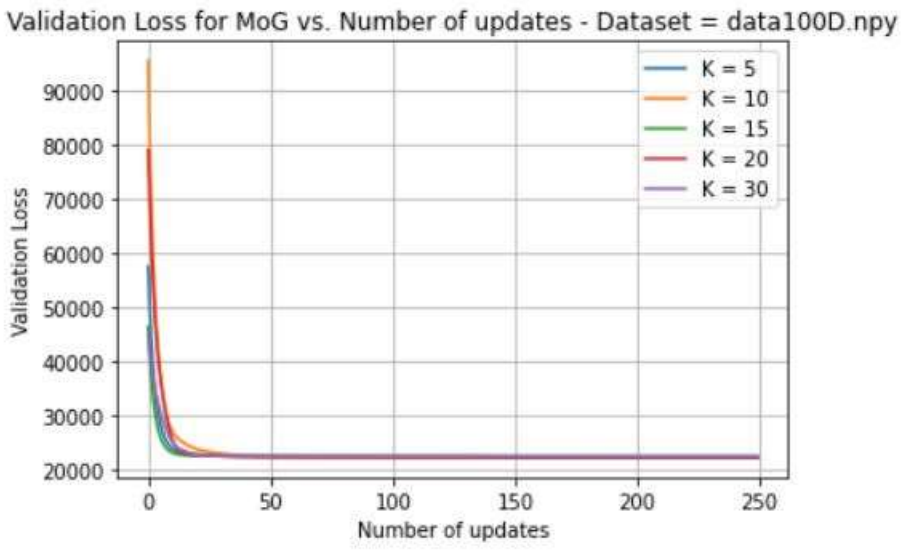


Figure 25: Loss vs. Number of Updates for MoG

Table 2: Final validation loss values for different values of K for K-Means and MoG models:

	K = 5	**K = 10**	**K = 15**	**K = 20**	**K = 30**
K-Means	122162.89	69758.24	69648.99	68435.9	68389.68
MoG	22488.361	22373.994	22313.521	22315.8	22486.451

Figure 24 above illustrates the validation loss for K-Means for cluster (K) values of 5, 10, 15, 20 and 30 over 200 iterations. Figure 25 above shows the validation loss for MoG model for K values of 5, 10, 15, 20 and 30 over 250 iterations.

For K-Means (figure 24), it can be seen that there was a large drop in loss value with increasing the value of K after K = 10. This is also illustrated in table 2 where the final validation losses for each model is displayed for different values of K. Therefore, it can be said that K=10 is the amount of clusters within the dataset by looking at both the validation loss values in table 2 as well as figure 23.

For MoG (figure 25), it can be seen that there was not any large drop in loss with increasing value of K after K = 5. However, the only drop can be seen in table 2 where the value of validation loss dropped from 22488.361 to 22373.994 for K = 5 and K = 10, respectively and remained somewhat consistent after K = 10. Therefore, by looking at the validation loss in table 2, it can be said that the amount of clusters within the dataset for MoG model is K = 10.

By looking at figures 25 and 25 and table 2 above for the validation loss for K-Means and MoG models, it is evident that the amount of decrease in loss after K = 10 is negligible and insignificant. However, the amount of decrease in loss in K-Means prior to K=10 is much higher. This is due to the K-Means loss being a sum of the distances from a point to its center. In conclusion, it can be said that K = 10 is the reasonable amount of clusters within the dataset for both K-Means and MoG models by looking at validation losses.