ECE421 Assignment 3

Student	Ⓥ Valentina Manferrari
utorID	1004796615

1. K-means

1.1 Learning K-means

distanceFunc()

$$\mathbf{X} = \begin{bmatrix} \underline{x}_1^T \\ \underline{x}_2^T \\ \dots \\ \underline{x}_N^T \end{bmatrix} \qquad \boldsymbol{\mu} = \begin{bmatrix} \underline{\mu}_1 \\ \underline{\mu}_2 \\ \dots \\ \underline{\mu}_k \end{bmatrix} \qquad \mathbf{D} = \left[||\underline{x}_i - \mu_j||^2 \right] = \begin{bmatrix} \underline{x_1\mu_1} & \dots & \underline{x_1\mu_k} \\ \dots & \dots & \dots \\ \underline{x_N\mu_1} & \dots & \underline{x_N\mu_K} \end{bmatrix}$$

D is the distance matrix, where $\underline{x_i\mu_j} = ||\underline{x}_i - \underline{\mu}_j||^2$. Therefore

$d_{ij} = ||\underline{x}_i - \underline{\mu}_j||^2 = \sum_d (x_{id} - \mu_{jd})^2 = \sum_d (x_{id}^2 1_{id} - 2x_{id}\mu_{dj}^T + 1_{id}(\mu_{dj}^T)^2)$

$\rightarrow \mathbf{D} = \mathbf{X}^2 \cdot \mathbf{1} - 2\mathbf{X}\boldsymbol{\mu}^T + \mathbf{1}(\boldsymbol{\mu}^T)^2$

```
# 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)
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
```

This function is then used to calculate the squared distance loss and the partition of data into clusters used for training our model.

```
# 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
def cluster_assignments(X, MU):
    D = distanceFunc(X, MU)
    s = tf.argmin(D, axis=1)
    return s # returns a Nx1 vector of cluster assignments for x_1 - x_N
```

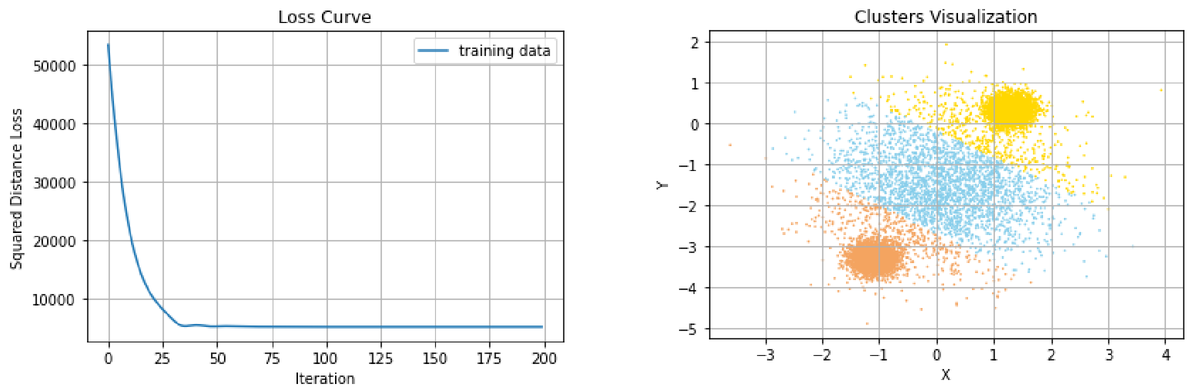
Training

Using the above implemented functions I constructed the below showed training function:

```
def train(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 dataset
            feed_dict_batch = {X: data}
            [_opt, _loss] = sess.run([optimizer, loss], feed_dict=feed_dict_batch)
            loss_curves['train'].append(_loss)
            #If dealing with validation dataset, get validation loss
            if validation:
                feed_dict_batch = {X: val_data}
                [_loss] = sess.run([loss], feed_dict=feed_dict_batch)
                loss_curves['valid'].append(_loss)

        # Getting assignment of clisters for training dataset
        feed_dict_batch = {X: data}
        [cluster_assignments['train']] = sess.run([s], feed_dict=feed_dict_batch)
        # Getting assignment of clisters for validation dataset
        if validation: # (only for validation == True)
            feed_dict_batch = {X: val_data}
            [cluster_assignments['valid']] = sess.run([s], feed_dict=feed_dict_batch)
        # Getting learned K-Means clusters
        [MU] = sess.run([MU], feed_dict={})

    return MU, loss_curves, cluster_assignments
```

```
### Helper function for train function
def build_graph(K, learning_rate): # K is used to define the number of clusters we want
    tf.set_random_seed(124)
    X = tf.placeholder(tf.float32, shape=[None, dim], name="X")

    # Defining variables to learn
    MU = tf.get_variable('MU', shape=[K, dim],
        initializer=tf.initializers.random_normal(mean=0, stddev=1))

    # Loss function
    loss = calculate_loss(X, MU)
    # Determining clusters assignment
    s = cluster_assignments(X, MU)
    # Initialization of GD optimizer (using Adam)
    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
```

Figure 1: K-means when K = 3. Loss curve and Clusters Visualization [Training Loss: 5110.9453]



$$\boldsymbol{\mu} = \begin{bmatrix} [1.2517773 & 0.24658293] \\ [0.12182339 & -1.5230445] \\ [-1.0559494 & -3.2431884] \end{bmatrix}$$

Results

Figure 2: K-means when K = 1. Loss curve and Clusters Visualization [Training Loss: 38453.49]

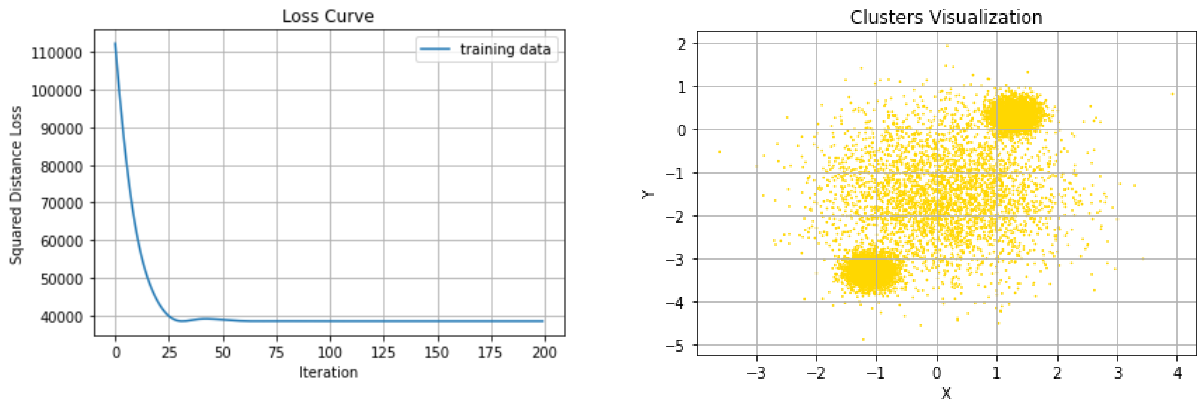


Figure 3: K-means when K = 2. Loss curve and Clusters Visualization [Training Loss: 9203.359]

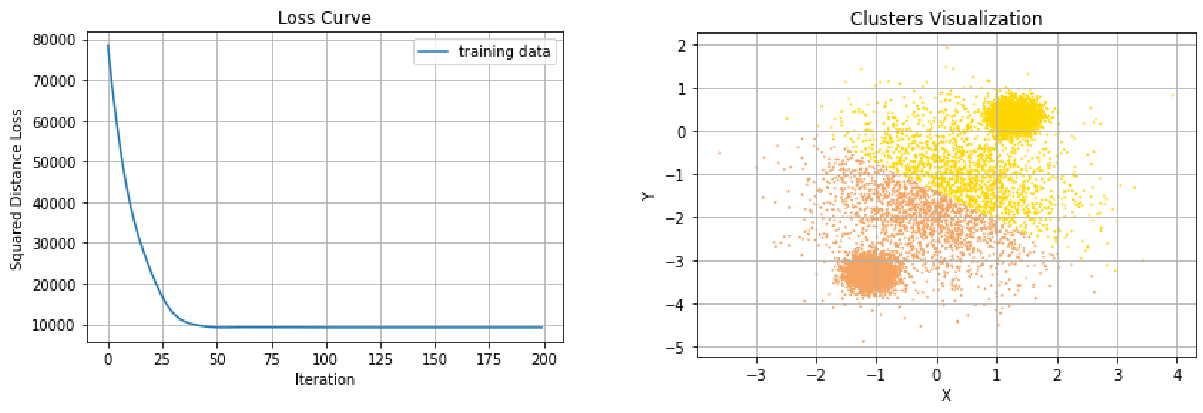


Figure 4: K-means when K = 3. Loss curve and Clusters Visualization [Training Loss: 5110.9453]

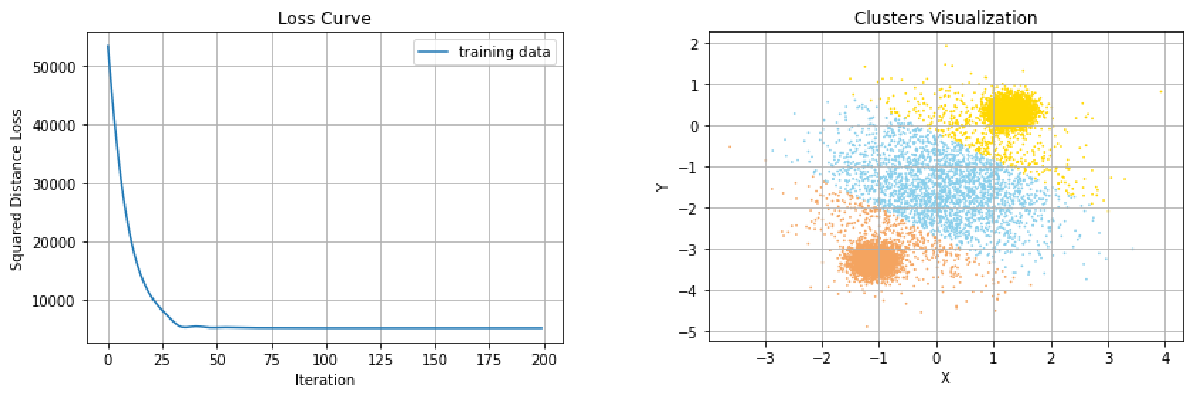


Figure 5: K-means when K = 4. Loss curve and Clusters Visualization [Training Loss: 3374.0356]

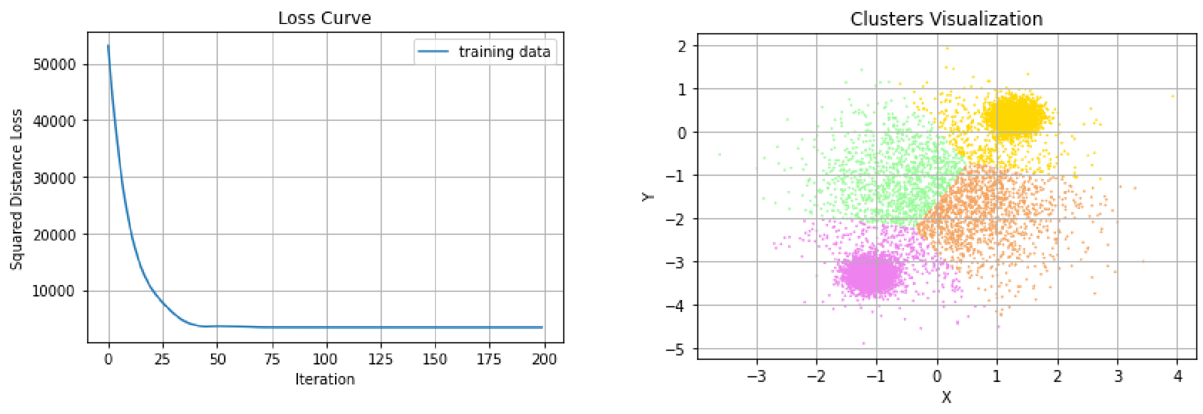


Figure 6: K-means when K = 5. Loss curve and Clusters Visualization [Training Loss: 2870.3918]

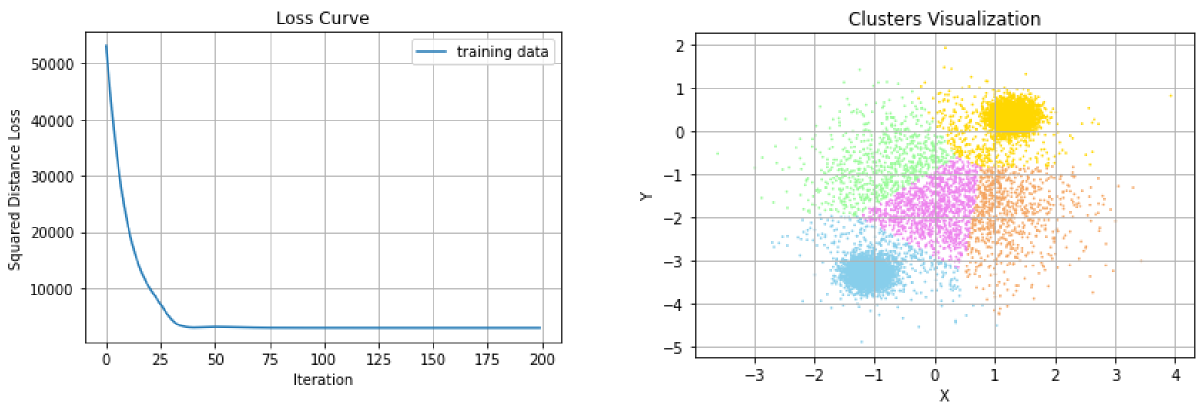


Table 1: Percentage of the data points belonging to each of the K clusters.

Name	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
K = 1	100				
K = 2	49.54	50.46			
K = 3	38.06	23.81	38.13		
K = 4	37.28	12.1	37.13	13.49	
K = 5	37.02	7.66	36.69	11.08	7.55

From the data reported above in [Table 1], we can observe that the percentage of data in each cluster is balanced up to K = 3. For K = 4 and K=5 indeed, some clusters have accumulated much more data than others, leading to more unbalanced cluster percentages. From this observation, we can therefore say that K = 3 seems to be the best cluster division choice, where 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. This choice still incorrectly distributes many points, but this is probably the best we can get since 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. One alternative could be to have K=5 and then group the three central clusters into one (green, purple, orange) to try to approximate the correct cluster assignment. This would result in a greater balance between the three clusters than with percentages of data points being 37.02 / 36.60 / 26.29 %

Figure 7: K-means when K = 1. Loss curve and Clusters Visualization on Validation Data

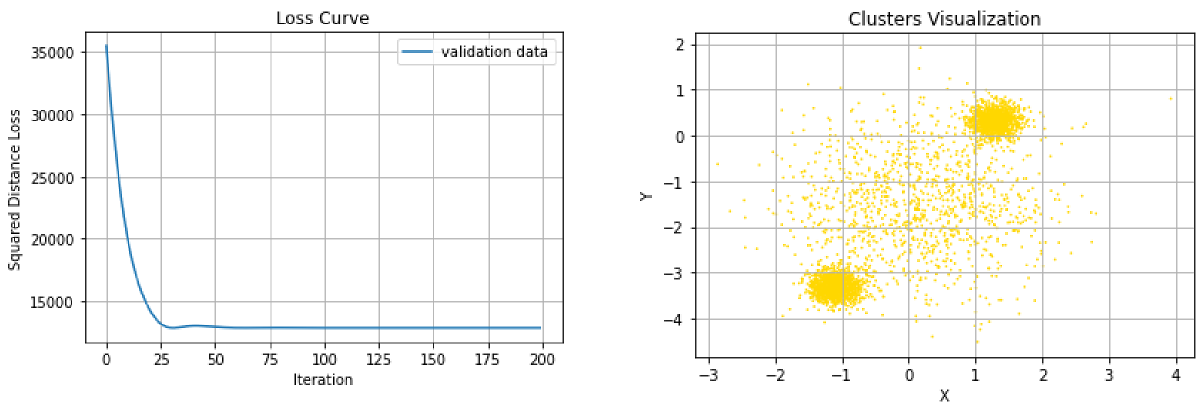


Figure 8: K-means when K = 2. Loss curve and Clusters Visualization on Validation Data

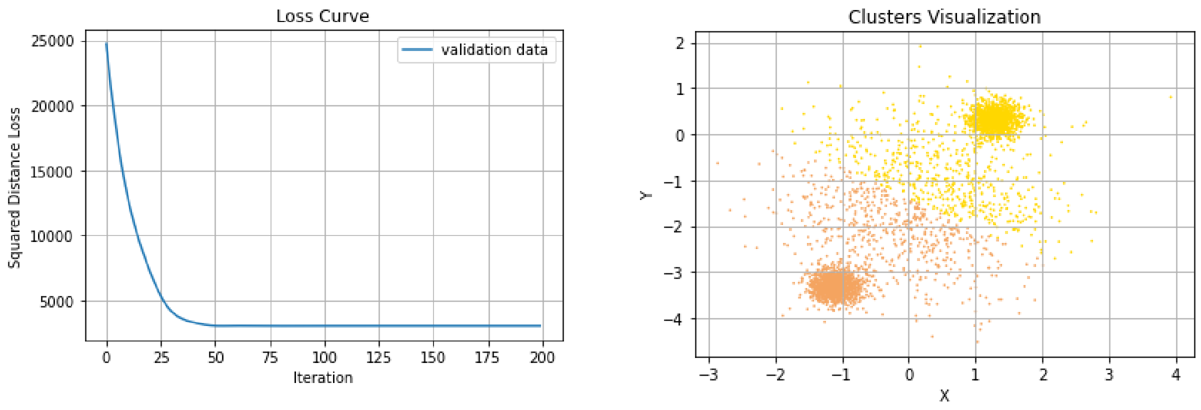


Figure 9: K-means when K = 3. Loss curve and Clusters Visualization on Validation Data

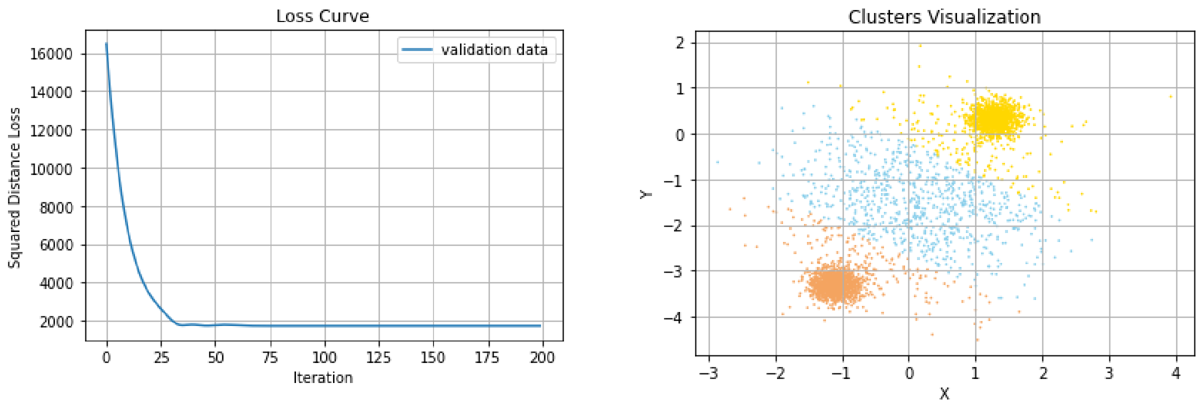


Figure 10: K-means when K = 4. Loss curve and Clusters Visualization on Validation Data

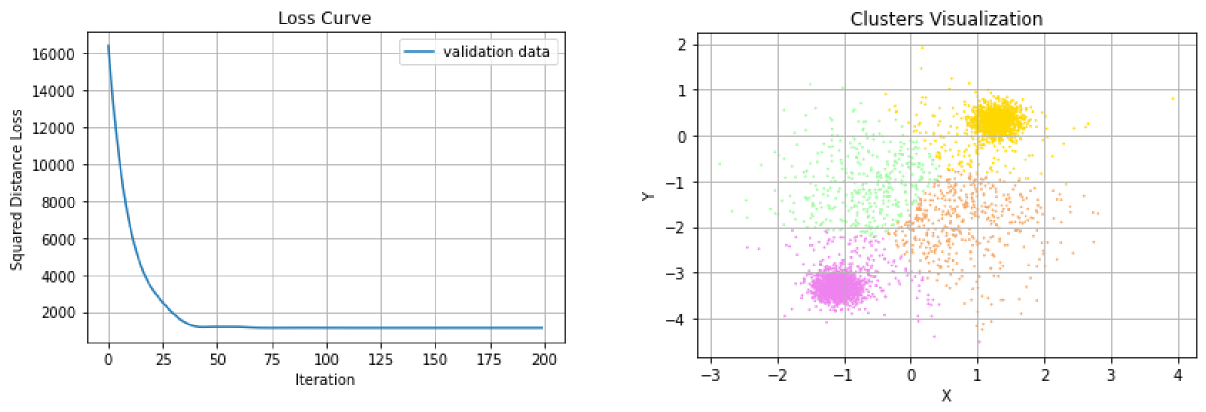


Figure 11: K-means when K = 5. Loss curve and Clusters Visualization on Validation Data

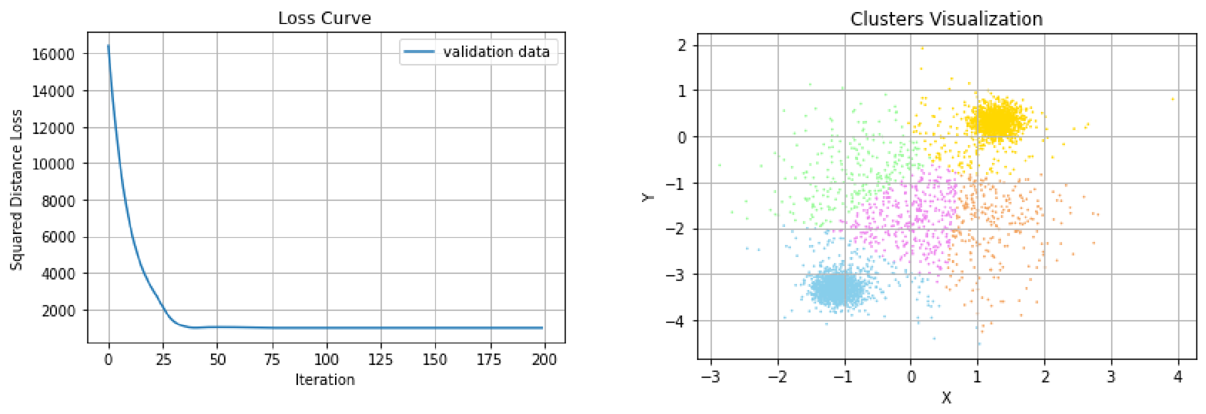
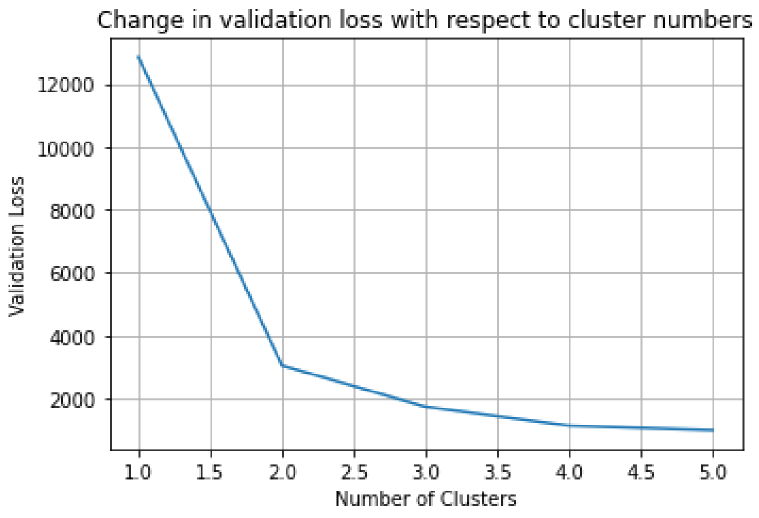


Table 2: Loss of validation data with respect to different cluster number.

Aa Number of clusters	# 1	# 2	# 3	# 4	# 5
Validation Loss	12856.141	3042.2302	1731.768	1129.8677	985.74207

Figure 12: Visualization of data contained in Table 2



It is pretty obvious that we can not draw a conclusion about which is the best number of clusters to use by only looking at the loss for different cluster sizes. Indeed, more clusters will always give us a smaller loss value because the loss is calculated as the sum of squared distances (in the most extreme case, if the number of clusters K equals the number of data we can even reach zero loss) but this does not mean at all that it is always better to use more clusters! Therefore, instead of looking at the actual loss values, we can look at the trend of change in loss values as shown in [Figure 12]. We can observe that there is not much change in the slope after K = 3 and therefore this should be the best choice. This result can also be confirmed by looking at the clusters graphs and percentages distributions as I did before.

2. Mixture of Gaussians

2.1 The Gaussian cluster mode

log_GaussPDF()

$$\mathbf{X} = \begin{bmatrix} \underline{x}_1^T \\ \underline{x}_2^T \\ ... \\ \underline{x}_N^T \end{bmatrix} \quad \boldsymbol{\mu} = \begin{bmatrix} \underline{\mu}_1 \\ \underline{\mu}_2 \\ ... \\ \underline{\mu}_k \end{bmatrix} \quad \boldsymbol{\sigma} = \begin{bmatrix} \sigma_1 & \sigma_2 & ... & \sigma_k \end{bmatrix} \quad \boldsymbol{\pi} = \begin{bmatrix} P(1) & P(2) & ... & P(k) \end{bmatrix}$$

$$\mathbf{P} = \left[lnP(\underline{x}_i|j)\right] = ln \begin{bmatrix} P(\underline{x}_1|1) & ... & P(\underline{x}_1|k) \\ ... & ... & ... \\ P(\underline{x}_N|1) & ... & P(\underline{x}_N|k) \end{bmatrix}$$

P is the log-gaussian matrix where $P(x_i|j) = \mathcal{N}(\underline{x}_i; \underline{\mu}_j, \sigma_j^2)$. Therefore:

$$p_{ij} = \mathcal{N}(\underline{x}_i; \underline{\mu}_j, \sigma_j^2) = ln\left[\frac{1}{(2\pi)^{d/2}|\sum_j|^{1/2}} e^{-\frac{1}{2}(\underline{x}_i - \underline{\mu}_j)^T \sum_j^{-1}(\underline{x}_i - \underline{\mu}_j)}\right]$$

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

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

$$\rightarrow \mathbf{P} = -d \cdot ln(\dot{\underline{\boldsymbol{\sigma}}}\sqrt{2\pi}) - \frac{1}{2\dot{\underline{\boldsymbol{\sigma}}}^2} \bigotimes [\mathbf{X}^2 \cdot \mathbf{1} - 2\mathbf{X}\boldsymbol{\mu}^T + \mathbf{1}(\boldsymbol{\mu}^T)^2]$$

```
# Inputs
#   X of sise N X D; MU of size K X D; Sigma of size 1 X K
# Outputs:
#   log Gaussian PDF of size N X K
def log_GaussPDF(X, MU, sigma):
    pair_dist = distanceFunc(X, MU) #same as defined in Part 1
    log_PDF = - dim * tf.log(sigma * np.sqrt(2*pi)) - pair_dist / (2 * tf.square(sigma))
    return log_PDF
```

log_posterior()

Similarly, let Q be the log-posterior matrix:

$$\mathbf{Q} = \left[lnP(j|\underline{x}_i)\right]$$

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

$$= lnP(\underline{x}_i|j) + lnP(j) - ln[\sum_1^k e^{lnP(\underline{x}_i|k)\cdot P(k)}]$$

$$= lnP(\underline{x}_i|j) + lnP(j) - ln[\sum_1^k e^{lnP(\underline{x}_i|k)+lnP(k)}]$$

$$\rightarrow \mathbf{Q} = \mathbf{P} + ln\dot{\underline{\boldsymbol{\pi}}} - ln\sum_{rows} e^{\mathbf{P}+ln\dot{\underline{\boldsymbol{\pi}}}}$$

```
# Input
#   log Gaussian PDF of size N X K; log_pi of size 1 X K
# Outputs
#   log_post of size N X K
def log_posterior(log_PDF, log_pi):
    Z = log_PDF + log_pi
    log_post = Z - reduce_logsumexp(Z, reduction_indices=1, keep_dims=True)
    return log_post
```

Since we want to use the log_GaussPDF function previously implemented to calculate the log posterior matrix, we need to use the reduce_logsumexp() function because only in this way the exp call will negate the logarithmic term from the log gaussian matrix. If instead, we had decided not to use the log Gaussian matrix previously calculated it would have been sufficient to use a log-sum function.

2.2 Learning the MoG

Maximum Likelihood Estimate computation:

$$L(\mu, \sigma, \pi) = -lnP(x) = -ln\prod_n \sum_k P(\underline{x}_n|k)P(k) =$$
$$-\sum_n [ln\sum_k P(\underline{x}_n|k)P(k)]$$
$$= -\sum_n [ln\sum_k e^{ln[P(\underline{x}_n|k)P(k)]}] = -\sum_n [ln\sum_k e^{lnP(\underline{x}_n|k)+lnP(k)}]$$
$$= -\sum_{cols} [ln\sum_{rows} e^{P+ln\dot{\underline{\Pi}}}]$$

```
# Input
#   X of size N X D; MU of K X D; sigma of size 1 X K; weights aka. P(k) of size 1 X 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
```

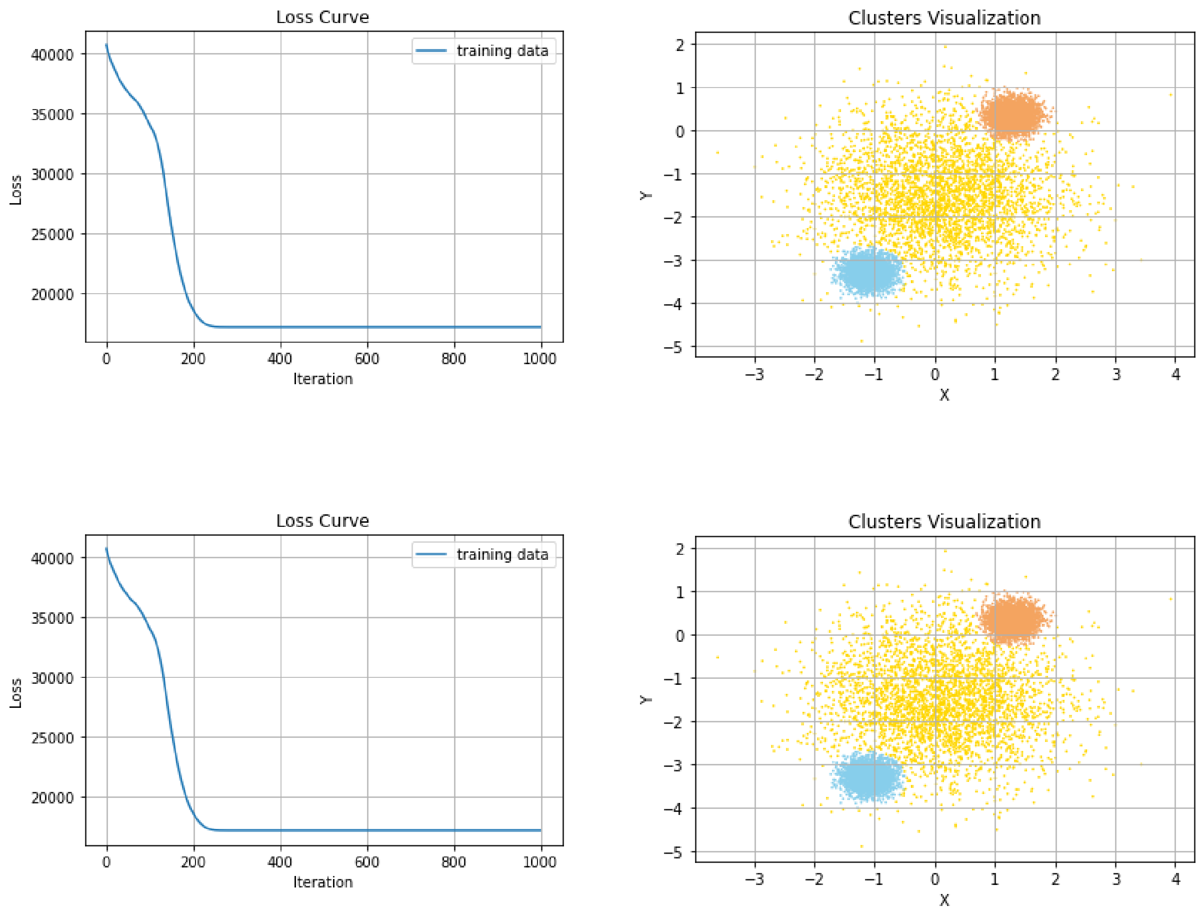
Training

The parameters used to get the best training results have been learning_rate=0.01 and number of epochs = 1000. All parameters were initialized by sampling from the standard normal distribution, with the mean = 0 and standard_deviation = 1.

```
def train(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 dataset
            feed_dict_batch = {X: data}
            [_opt, _loss] = sess.run([optimizer, loss], feed_dict=feed_dict_batch)
            loss_curves['train'].append(_loss)
            #If dealing with validation dataset, get validation loss
            if validation:
                feed_dict_batch = {X: val_data}
                [_loss] = sess.run([loss], feed_dict=feed_dict_batch)
                loss_curves['valid'].append(_loss)

        # Getting assignment of clisters for training dataset
        feed_dict_batch = {X: data}
        [cluster_assignments['train']] = sess.run([s], feed_dict=feed_dict_batch)
        # Getting assignment of clisters for validation dataset
        if validation: # (only for validation == True)
            feed_dict_batch = {X: val_data}
            [cluster_assignments['valid']] = sess.run([s], feed_dict=feed_dict_batch)

        # Getting learned K-Means clusters
        [MU, sigma, w] = sess.run([MU, sigma, w], feed_dict={})

    return MU, sigma, w, cluster_assignments, loss_curves
```

```
### Helper function for train function
# K is used to define the number of clusters we want
def build_graph(K, learning_rate):
    tf.set_random_seed(421)
    X = tf.placeholder(tf.float32, shape=[None, dim], name="X")

    # Defining variables to learn
    init = tf.initializers.random_normal(mean=0, stddev=1)
    MU = tf.get_variable('MU',
        shape=[K, dim],
        initializer=init)

    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)
    w = tf.exp(ln_w, name='weight')

    # Loss function
    loss = calculate_loss(X, MU, sigma, w)
    # Determining clusters assignment
    s = cluster_assignments(X, MU, sigma, w)
    # Initialization of GD optimizer (using Adam)
    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
```

Figure 13: MoG with K = 3. Loss curve and Clusters Visualization [Training Loss: 17132.29]



$$\boldsymbol{\mu} = \begin{bmatrix} [0.10601898 & -1.5274578] \\ [-1.1014141 & -3.3061883] \\ [1.2986271 & 0.30878878] \end{bmatrix}$$

$$\boldsymbol{\sigma} = \begin{bmatrix} 0.99356985 & 0.19768703 & 0.19710371 \end{bmatrix}$$

$$\boldsymbol{W} = \begin{bmatrix} 0.3346481 & 0.33191153 & 0.33344024 \end{bmatrix}$$

Results

Figure 14: MoG with K = 1. Loss curve and Clusters Visualization [Training Loss: 34915.94]

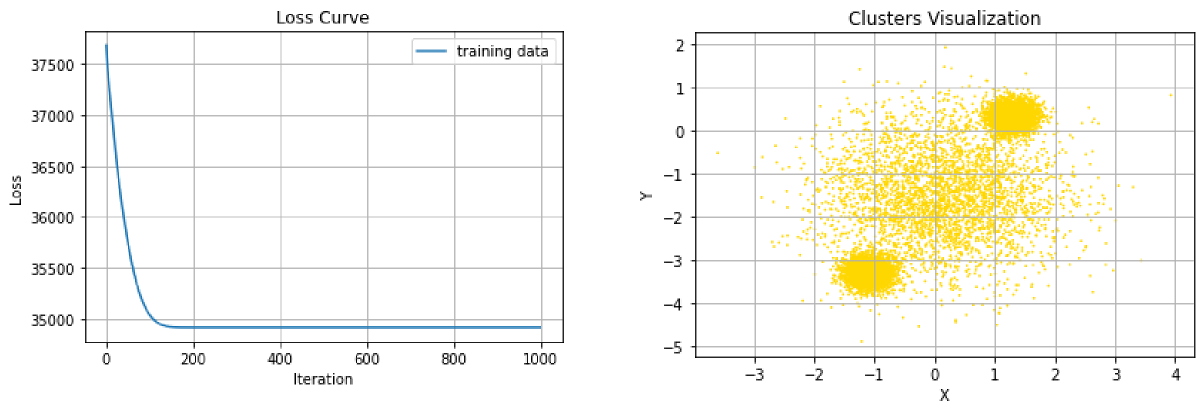
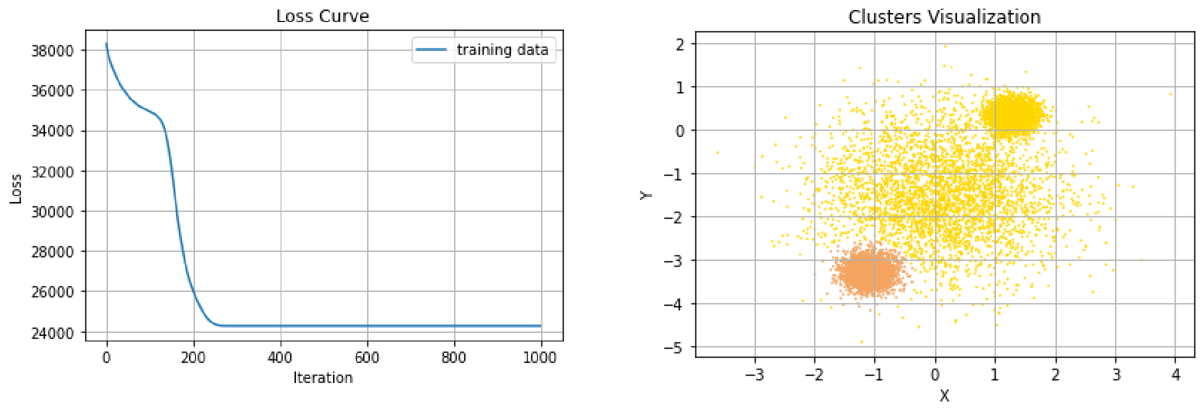


Figure 15: MoG with K = 2. Loss curve and Clusters Visualization [Training Loss: 24270.084]



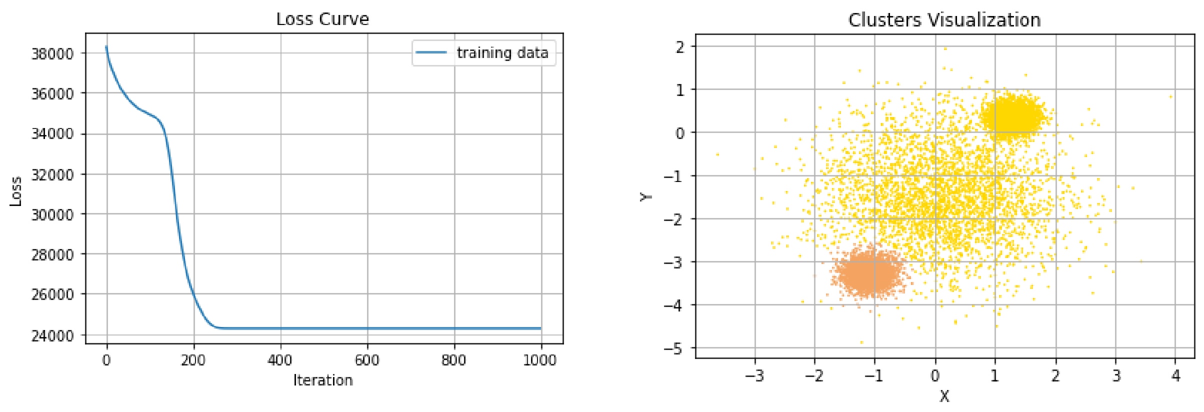


Figure 16: MoG with K = 3. Loss curve and Clusters Visualization [Training Loss: 17132.29]

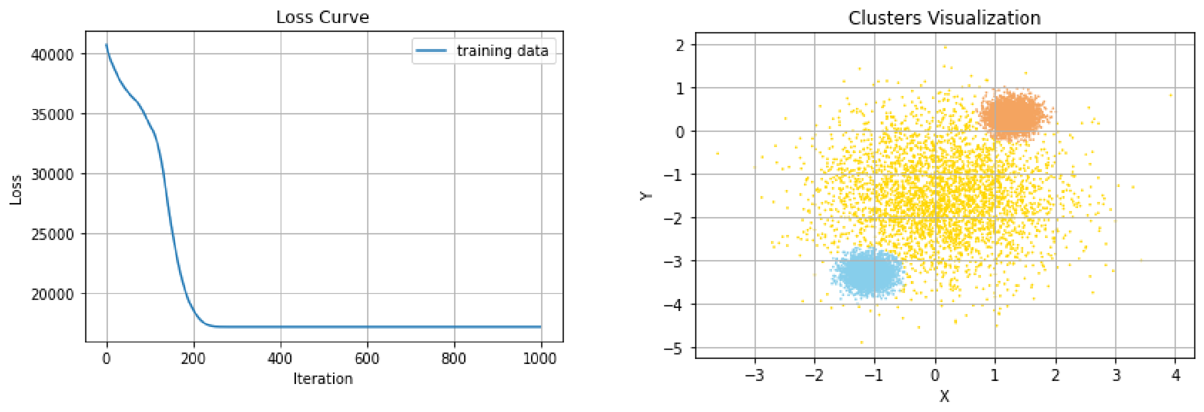


Figure 17: MoG with K = 4. Loss curve and Clusters Visualization [Training Loss: 17132.29]

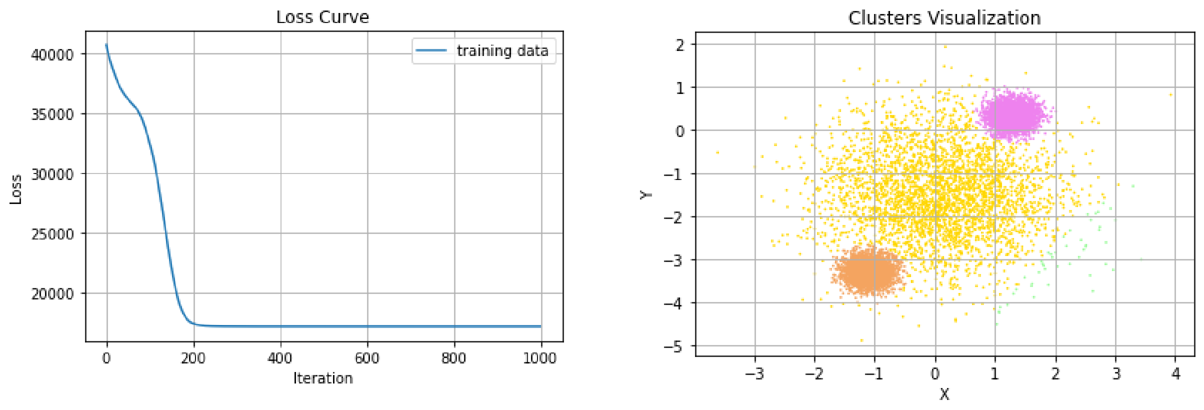


Figure 18: MoG with K = 5. Loss curve and Clusters Visualization [Training Loss: 17130.355]

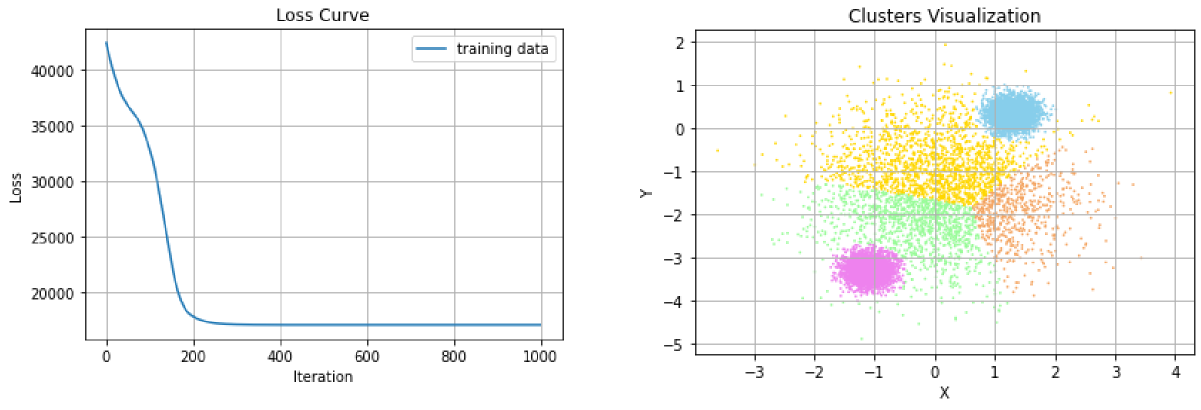


Table 3: Percentage of the data points belonging to each of the K clusters.

Name	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
K = 1	100				
K = 2	65.65	34.35			
K = 3	32.27	33.84	33.89		
K = 4	31.61	0.47	33.98	33.94	
K = 5	15.26	11.32	33.97	33.88	5.57

Figure 19: MoG with K = 1. Loss curve and Clusters Visualization on Validation Data

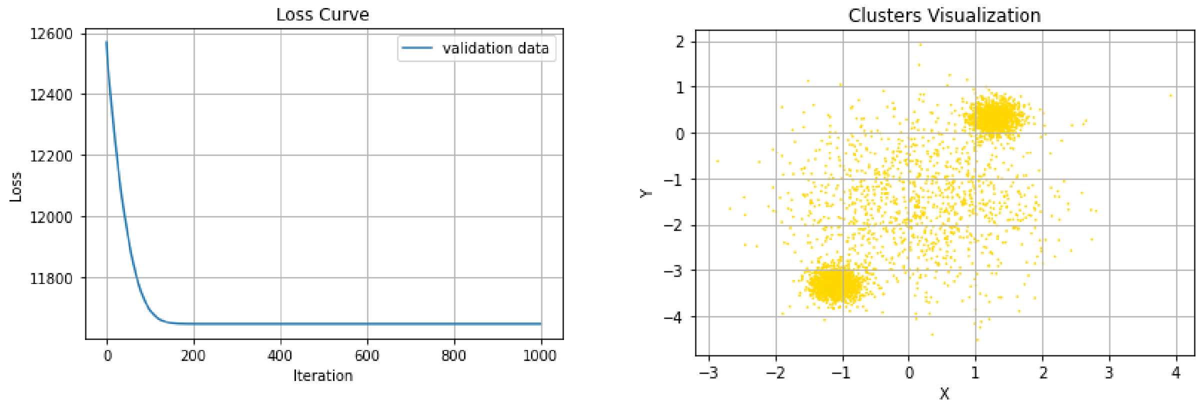


Figure 20: MoG with K = 2. Loss curve and Clusters Visualization on Validation Data

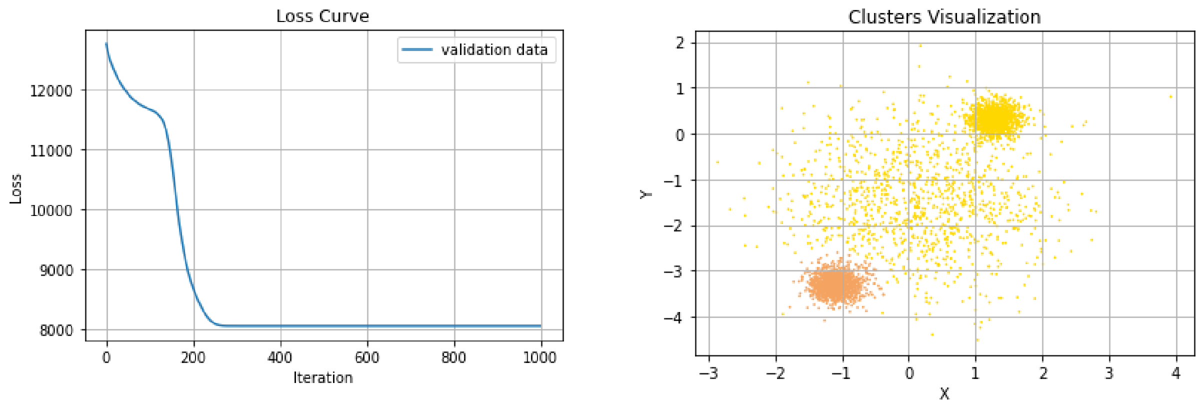


Figure 21: MoG with K = 3. Loss curve and Clusters Visualization on Validation Data

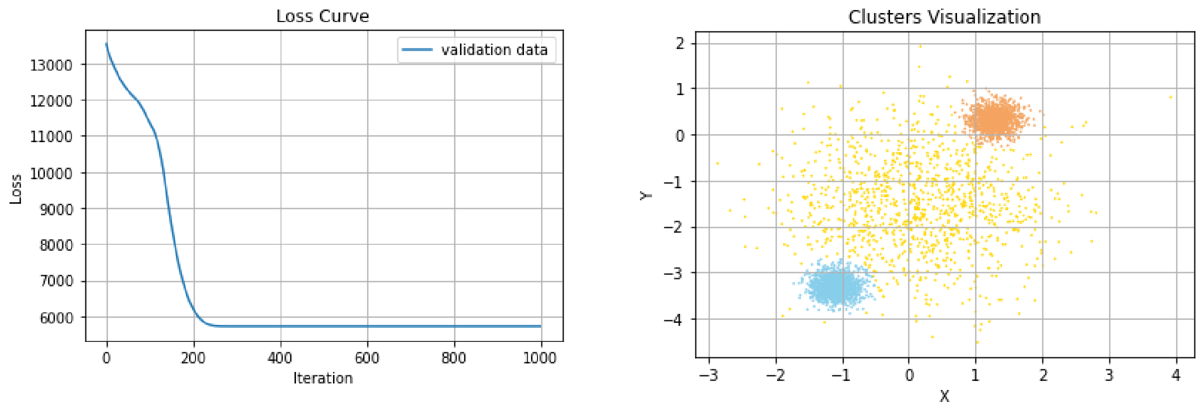


Figure 22: MoG with K = 4. Loss curve and Clusters Visualization on Validation Data

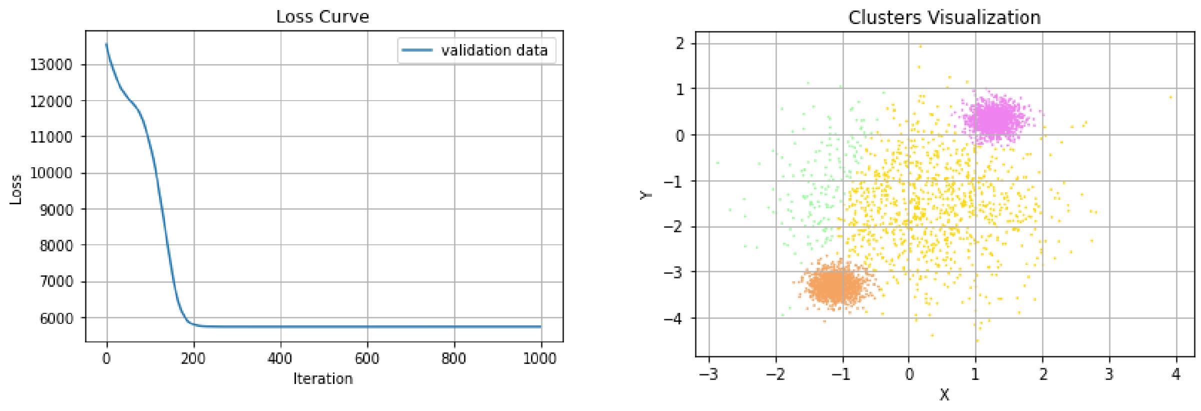


Figure 23: MoG with K = 5. Loss curve and Clusters Visualization on Validation Data

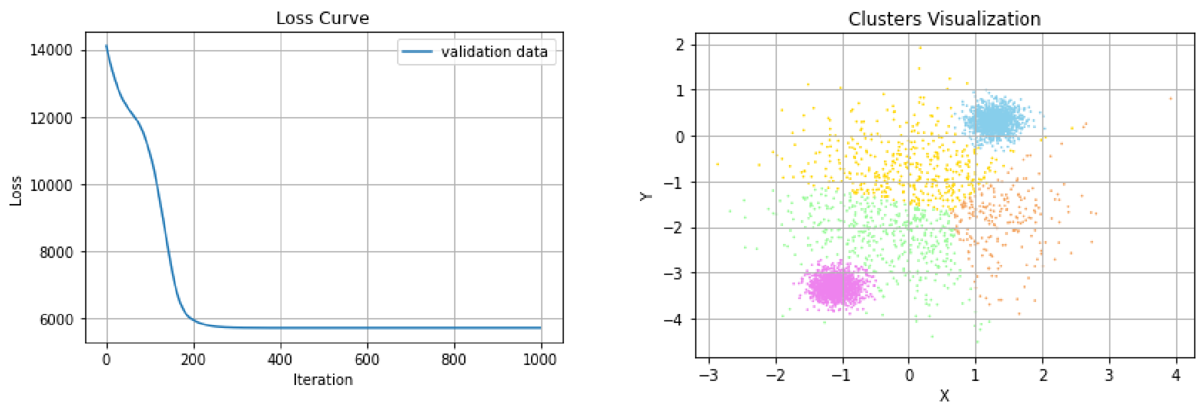


Table 4: Loss of validation data with respect to different cluster number.

Number of clusters	1	2	3	4	5
Validation Loss	11647.817	8047.223	5721.836	5723.0273	5724.0635

Since the loss is defined as the negative log-likelihood, $L = -logP(\mathbf{X})$, the model with the lower loss will have a higher likelihood. Therefore, the value of K that produces the lowest loss should theoretically be the best choice of K. By examining the validation losses reported in [Table 4] we observe that K=3 produces the smallest validation loss, and therefore has the greatest probability. Another characteristic we can look at is the actual cluster assignments of the data: models having empty clusters may indeed suggest a K value that is larger than the actual number of clusters.
By examining [Table 3], we can observe that even though validation loss for K=4 is very close to the loss of K=3, it has one cluster that contains very few points; this confirms our first claim that K=3 is the best model choice.

Note: since in this case we are dealing with only 2 dimensions, the choice of K=3 can also be confirmed by simply looking at the scatter graph visualization of the data and asserting that there are 3 clusters.

K-Means vs MoG

Figure 24: Validation losses using of K-Means and MoG with different K values

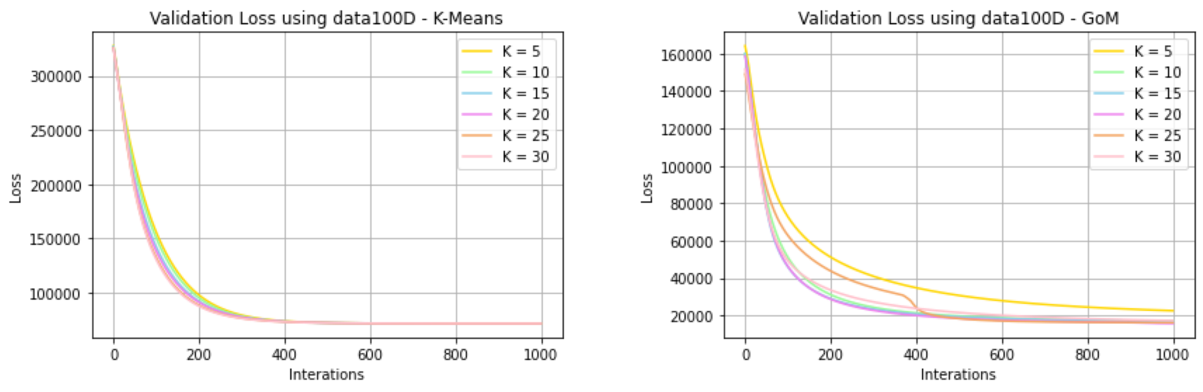


Table 5: Loss of validation data with respect to different cluster number.

Clusters	5	10	15	20	25	30
K-Means Loss	143545.29	143545.32	143544.89	143545.37	143544.95	143544.85
MoG Loss	45152.44	34106.18	32485.37	31468.57	33753.77	34436.78

Since this dataset has 100 dimensions we cannot visualize it and we must rely on the loss values and cluster distributions only to determine the right 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 much: as the K value increases, the loss will always decrease as the distances become smaller. Therefore it is best to analyze the negative log-likelihood of the MoG model. After learning the clusters based on the training data I then computed the loss for the validation data: the K value with the lowest validation loss is most likely to be the best K value.
The smallest loss for the MoG model reached is 31468.57 when K=20 with the next smallest loss being 32485.37 at K=15. Now, looking at the cluster distributions for K=20 and K=15 it is observable that they are both extremely similar: both have the same number of populated clusters and the number of points in each of these is almost the same. Therefore, we can estimate that there are about K=20 clusters in this dataset.