1. K-means

1.1.

The distanceFunc() was implemented as follows:

```
# Distance function for K-means
def distanceFunc(X, MU):
    # Inputs
    # X: is an NxD matrix (N observations and D dimensions)
    # MU: is an KxD matrix (K means and D dimensions)
    # Outputs
    # pair_dist: is the squared pairwise distance matrix (NxK)
    # TODO
    newX = tf.expand_dims(X,0)
    newMU = tf.expand_dims(MU, 1)
    dis = tf.reduce_sum(tf.square(tf.subtract(newX,newMU)),2)
    output = tf.transpose(dis)
    return output
```

Figure 1: Python Implementation of distanceFunc()

This function calculates and returns the squared pairwise distance between the all N data points and K cluster centre points. The output matrix is NxK size.

After using the Adam Optimizer with the recommended hyperparameters, the following output was produced:

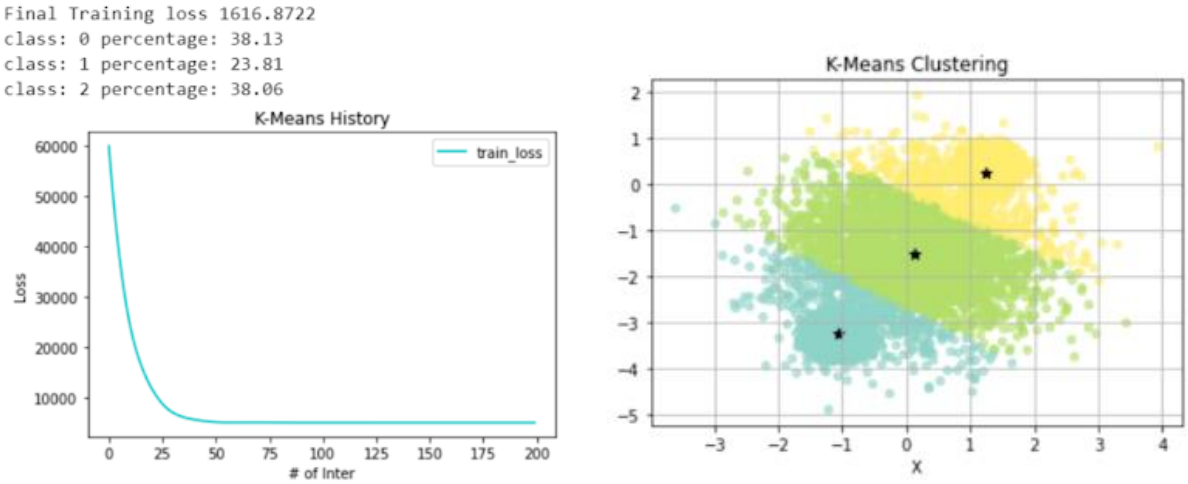


Figure 2: The Loss Graph and Data Clusters for Part 1.1

Below is the Python implementation of K-means using Adam optimizer:

```
#PART 1.1

#starter_kmeans.py

import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
import numpy as np
import matplotlib.pyplot as plt
import helper as hlp

def dataAssign(X, MU):
  dists= distanceFunc(X,MU)
  min_dist = tf.argmin(dists,1)
  return min_dist

# Distance function for K-means
def distanceFunc(X, MU):
    # Inputs
    # X: is an NxD matrix (N observations and D dimensions)
    # MU: is an KxD matrix (K means and D dimensions)
    # Outputs
    # pair_dist: is the squared pairwise distance matrix (NxK)
    # TODO
    newX = tf.expand_dims(X,0)
    newMU = tf.expand_dims(MU, 1)
    dis = tf.reduce_sum(tf.square(tf.subtract(newX,newMU)),2)
    output = tf.transpose(dis)
    return output

data = np.load('/content/drive/My Drive/data2D.npy')
[NumberOfPoints, dimension] = np.shape(data)

loss_history = np.empty(shape=[0],dtype=float)
K = 3
D = dimension
N = NumberOfPoints
```

```
iterations = 200

X = tf.placeholder("float", shape=[None,D])
MU_initial = tf.truncated_normal([K,D],stddev=0.05)
MU = tf.Variable(MU_initial)

distance = distanceFunc(X,MU)
loss = tf.reduce_sum(tf.reduce_min(distance,axis = 1))
optimizer = tf.train.AdamOptimizer(learning_rate=0.1, beta1=0.9, beta2= 0.999, epsilon=1e-5).minimize(loss)

init_g = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init_g)

for step in range(iterations):
  centVal,lossVal,_ = sess.run([MU,loss,optimizer], feed_dict={X:data})
  loss_history = np.append(loss_history,lossVal)

clustering = sess.run(dataAssign(X, MU),feed_dict={X: data, MU:centVal})
percentages = np.zeros(K)

for i in range(K):
  percentages[i] = np.sum(np.equal(i, clustering))*100.0/len(clustering)
  print("Percentage of class", i, ":", percentages[i])

plt.scatter(data[:, 0], data[:, 1], c=clustering,
            cmap=plt.get_cmap('Set3'), s=25, alpha=0.6)
plt.scatter(centVal[:, 0], centVal[:, 1], marker='*', c="black",
            cmap=plt.get_cmap('Set1'), s=50, linewidths=1)
plt.title('K Means Clustering')
plt.xlabel('X')
plt.ylabel('Y')
plt.grid()
plt.show()

plt.figure(1)
plt.plot(range(len(loss_history)), loss_history, c="c", label="train_loss")
plt.legend(loc = "best")
plt.xlabel('Number of Iterations')
plt.ylabel('Loss')
plt.show()
```

Figure 3: Python Implementation of K-means using Adam Optimizer

1.2.

Based on the figures below for K=1 to K=5, it can be said that K=3 is the best option. This is because if the percentages of the data point distribution are noticed for K=4 and K=5, it can be seen that there is an uneven distribution. K=3 provides the most balanced distribution.

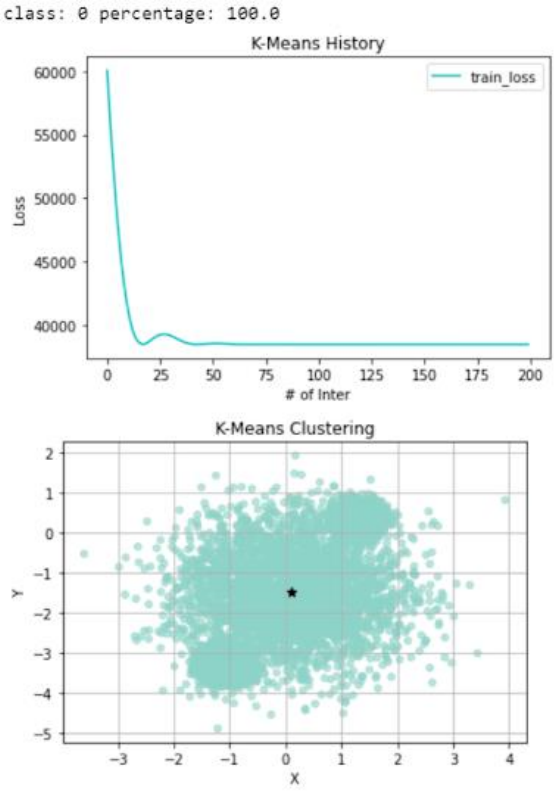


Figure 4: Loss and Cluster data with K=1

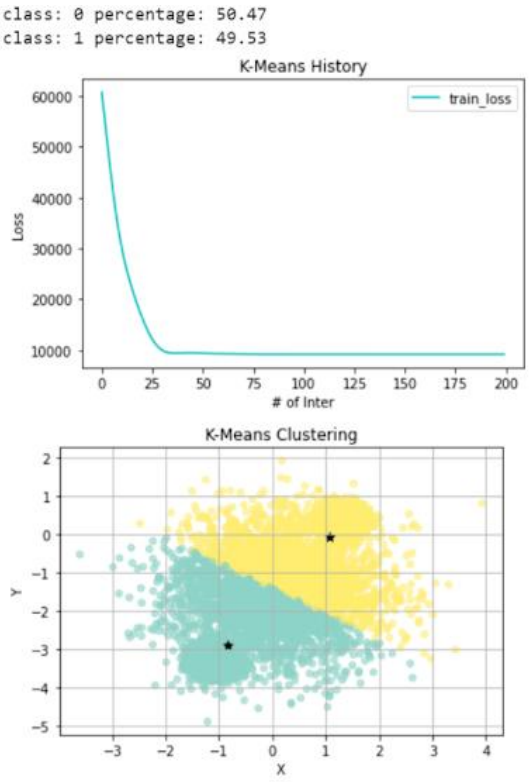
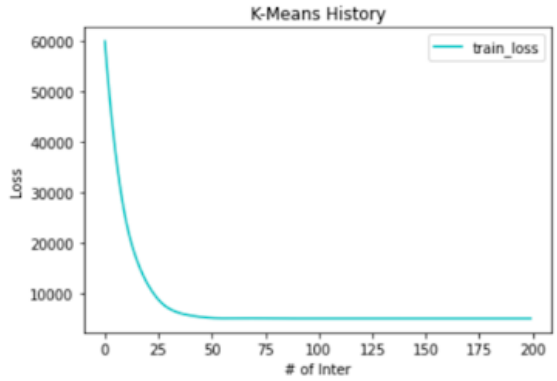


Figure 5: Loss and Cluster data with K=2

```
Final Training loss 1616.8722
class: 0 percentage: 38.13
class: 1 percentage: 23.81
class: 2 percentage: 38.06
```



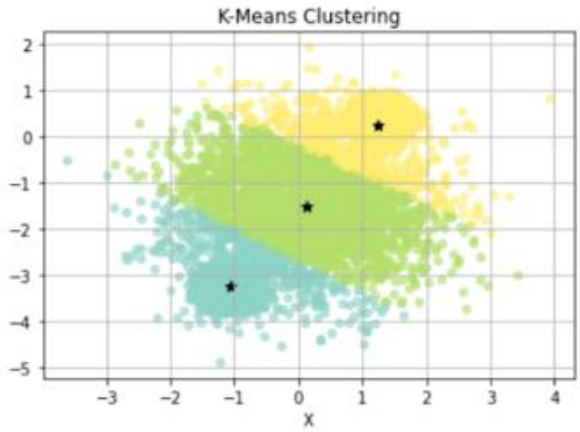


Figure 6: Loss and Cluster data with K=3

```
class: 0 percentage: 37.28
class: 1 percentage: 12.1
class: 2 percentage: 13.49
class: 3 percentage: 37.13
```

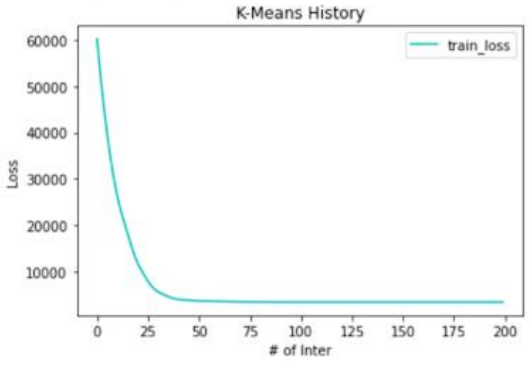
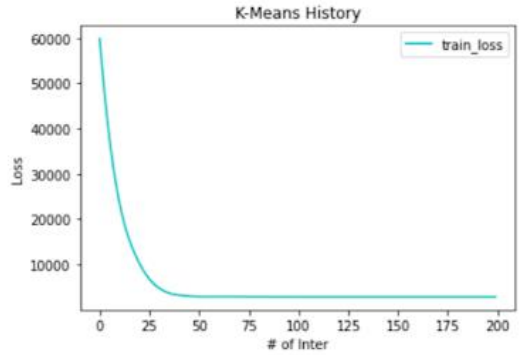


Figure 7: Loss and Cluster data with K=4

```
class: 0 percentage: 11.08
class: 1 percentage: 37.02
class: 2 percentage: 7.55
class: 3 percentage: 36.69
class: 4 percentage: 7.66
```



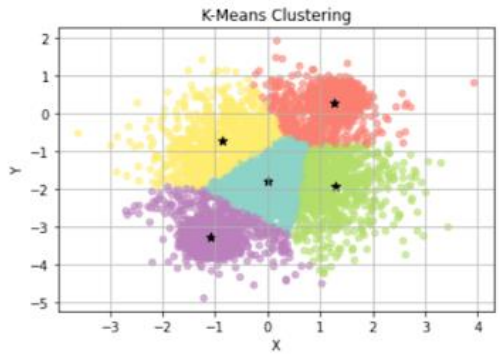


Figure 8: Loss and Cluster data with K=5

1.3.

The final validation loss for each of the K values tested is shown below:

K Value	Final Validation Loss
1	12860.77
2	2971.56
3	1677.40
4	1106.39
5	946.094

Table 1: Values for Validation Loss for Various K-Values Ranging from 1 to 5

The best k-value cannot be solely decided based upon the Final Validation Loss. This is because the higher the K-value, the lower the loss will be. When the K-value is equal to the number of points (K=N), then the loss is zero. However, that does not mean that K=N is the best K-value. In terms of the decrease in loss, it can be seen that the loss valid in Table 1 rapidly decreases until K=3, but then the decrease becomes slower. This means that K=3 is the best K-value.

2. Mixtures of Gaussians

2.1. The Gaussian Cluster Mode

2.1.1.

Below is the Python implementation of Log Probability Density Function.

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

    # Outputs:
    # log Gaussian PDF N X K

    StraightDist = distanceFunc(X, mu)
    Dist = -1* tf.div(StraightDist, tf.transpose(2*sigma))
    x = -1*tf.log((2*math.pi)**(dim/2)*sigma)
    return (tf.transpose(x)+ Dist)
```

Figure 9: Python Implementation of Log Probability Density Function

2.1.2.

Below is the Python implementation of Log Probability of Clusters.

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

    # Outputs
    # log_post: N X K
    prob = tf.add(log_PDF, log_pi)
    sum = reduce_logsumexp(prob + log_pi , keep_dims = True)
    return prob - sum
```

Figure 10: Python Implementation of Log Probability of Clusters

2.2. Learning the MoG

2.2.1.

For this part, we initialized learning rate to 0.001, iteration to 1000, and the other parameters as follow:

```
# initialize parameters
iteration = 1000
standard_deviation = 0.01

X = tf.placeholder("float", [None, D], "X")
mu = tf.Variable(tf.random_normal([K, D], stddev = standard_deviation))
sigma = tf.Variable(tf.random_normal([K, 1], stddev = standard_deviation))
exp_sigma = tf.exp(sigma)
pi = tf.Variable(tf.random_normal([K, 1], stddev = standard_deviation))
log_pi = tf.squeeze(logsoftmax(pi))
```

Figure 11: Initialization of Parameters

By setting K to 3, we plot 2D cluster and loss with respect to number of updates using helper functions. The training result is shown below.

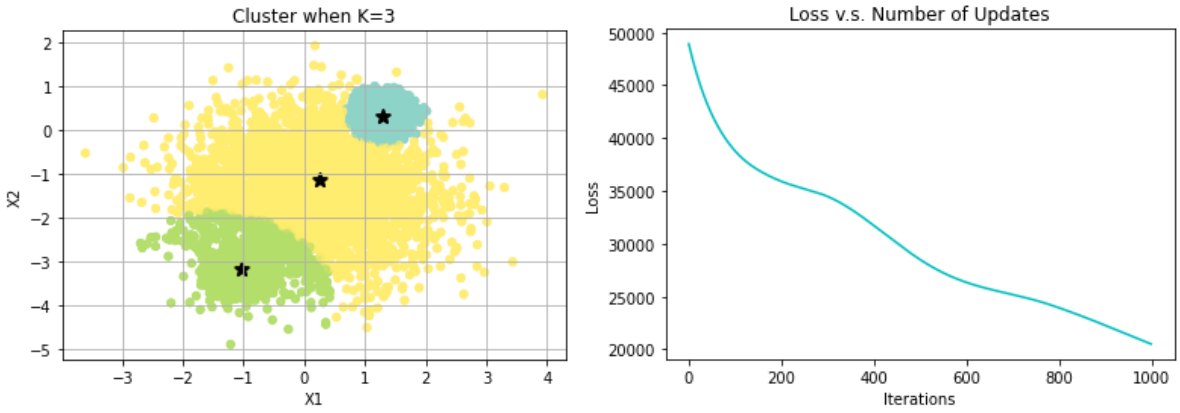


Figure 12: 2D Cluster Plot and Loss vs. Number of Updates Plot with K=3

From the training result, the best model parameters learnt is shown in the following table.

Cluster	π	μ	σ
1	-1.1131778	[0.00378427 0.0207892]	0.9953059
2	-1.082175	[-0.01064107 -0.01662607]	0.9888336
3	-1.1007279	[0.01392248 -0.00137027]	0.99996436

Table 2: Best Model Parameters for Gaussian Clusters

2.2.2.

Noticeable from the graph below, the loss decreases quickly as K goes from 1 to 3. However, the changes slow down after K=3. This means that for this case as well, K=3 is the optimal number of clusters to classify the data.

K Value	Loss
1	11648.988
2	7980.297
3	5623.892
4	5623.197
5	5623.87

Table 3: Final Loss Values with Different Values of K

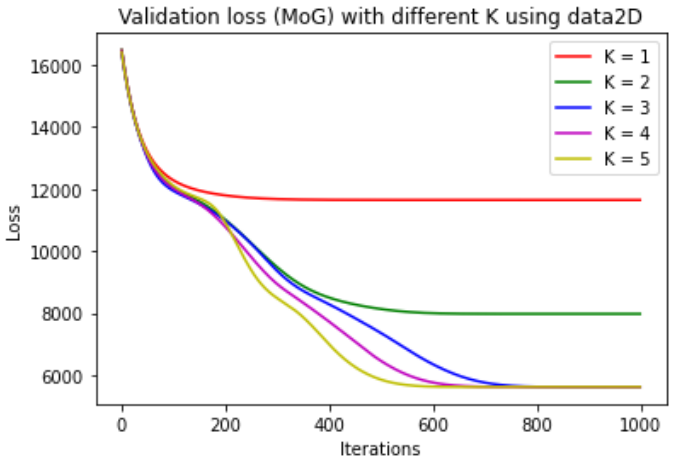


Figure 13: Validation Loss Values with Different Values of K

Below is the 2D scatter plot of different values of K.

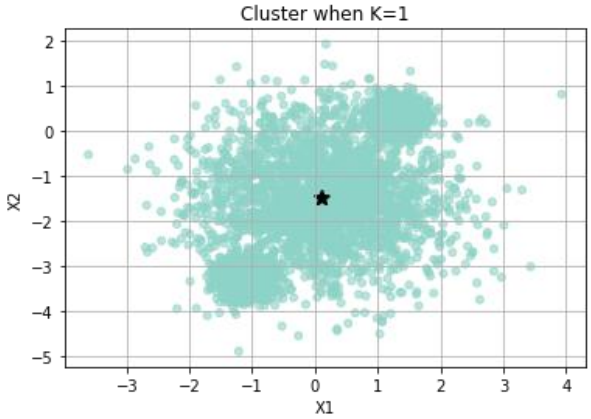


Figure 14: 2D Scatter Plot with K=1

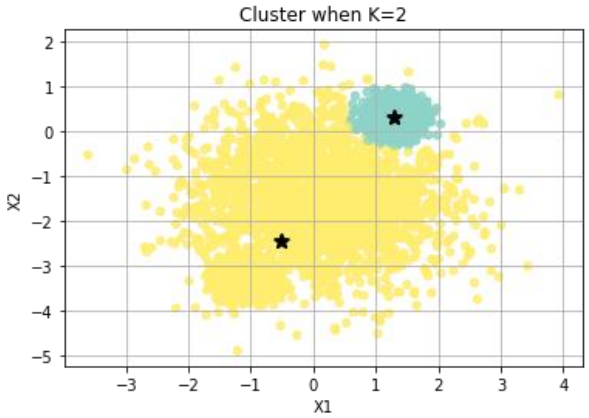


Figure 15: 2D Scatter Plot with K=2

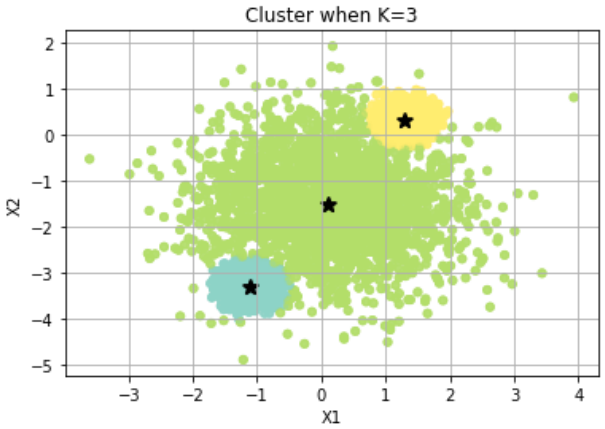


Figure 16: 2D Scatter Plot with K=3

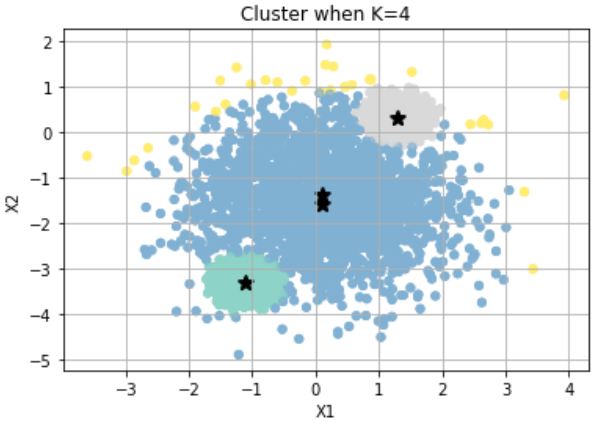


Figure 17: 2D Scatter Plot with K=4

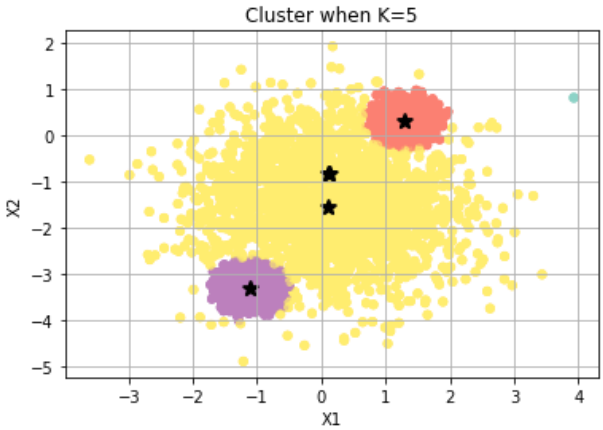


Figure 18: 2D Scatter Plot with K=5

2.2.3.

Using MoG and K means on data100D data points, the loss graph of various K values are shown below.

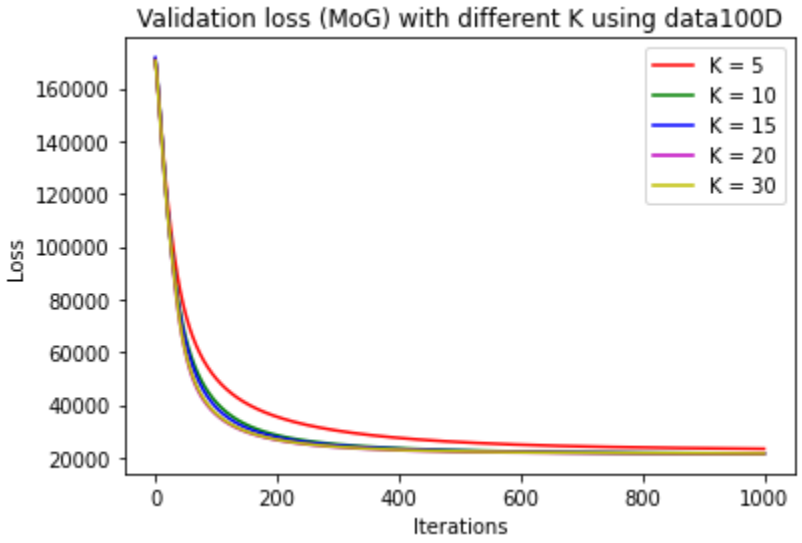


Figure 19: Validation Loss Using MoG

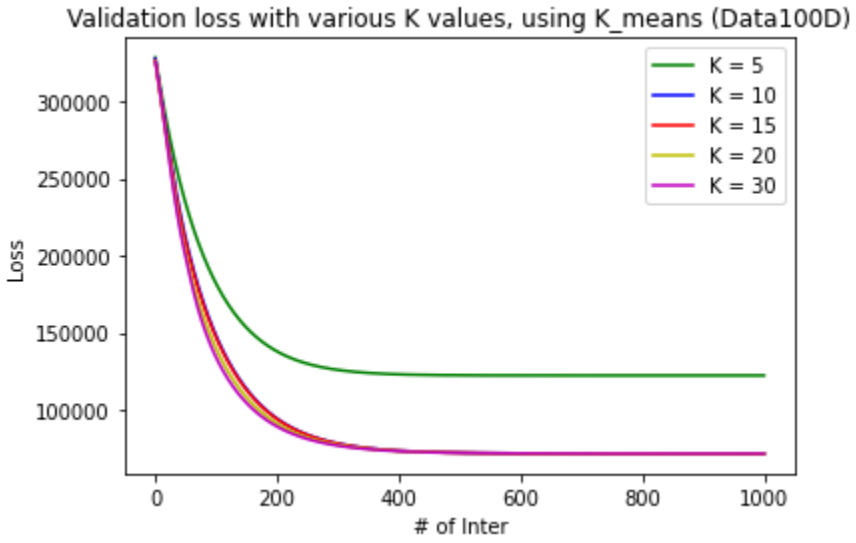


Figure 20: Validation Loss Using K Means

From figure 19 and 20, we observe that the loss values are much higher in K means compared to MoG method with Data100D dataset. From K >= 10, it can be seen that the plot converges to a similar loss value after 1000 iterations. As a result, K = 10 is the reasonable amount of clusters for training using MoG.