Introduction to Machine Learning Winter 2020

Lab 3: Unsupervised Learning & Probabilistic Models

Team member	Contribution	Contribution percentage
Yunyi Zhu	1 (1.1.1-1.1.3), 2(2.1.1, 2.1.2)	50%
Zibo Wen	2(2.1.1 - 2.2.3)	50%

0 Introduction

Two types of unsupervised learning models are implemented in this assignment, where the task is to summarize the simulated datasets into different clusters. In the first part, a K-means learning model is developed, by alternating between assigning data points in clusters and updating the cluster center; in the second part, a probabilistic version of K-means clustering, the Mixtures of Gaussians (MoG) is implemented. The effect of choosing different numbers of cluster centers in each model is also investigated.

1 K-means

1.1 Learning K-means

1.1.1 Implementation and Tensorflow

The distanceFunc() function is implemented as follows. The data matrix and the MU matrix is expanded with tf.repeat and tf.tile so the distance is calculated for each data point with respect to every data centers.

```
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)
      N = X.get_shape().as_list()[0];
      K = MU.get_shape().as_list()[0];
      X_repeat = tf.repeat(X,repeats=K,axis=0);
      MU_tile = tf.tile(MU, [N,1]);
      reducedSum = tf.reduce_sum(tf.square(X_repeat-MU_tile),1);
      pair_dist = tf.reshape(reducedSum,[-1,K]);
      return pair_dist;
```

The tensorflow training is done as follows. As a result of the a few hundred updates mentioned in the handout, the clustering results are plotted after 500 epochs. The exact same Adam optimizer parameters are used according to the handout.

```
def K_means(numClusters):
      MU = tf.Variable(tf.truncated_normal(shape=(numClusters, dim),
stddev=0.5, dtype=tf.float32))
      X = tf.placeholder(tf.float32, shape=(num_pts, dim))

      pair_dist = distanceFunc(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)

      init_op = tf.global_variables_initializer()

      trainingError = []
      validationError = []
      colors = []
      epoch = 500
      cluster1 = 0;
      cluster2 = 0;
      cluster3 = 0;
      cluster4 = 0;
      cluster5 = 0;
      distanceMatrix = 0;

      with tf.Session() as sess:
      sess.run(init_op)
      feed_dict={X: data}
      for i in range(epoch):
            distanceMatrix, trainingE, _ = sess.run([pair_dist, loss,
optimizer], feed_dict=feed_dict);
            if (is_valid):
                  validationError.append(validationLoss(MU.eval()));

            if 1:
                  trainingError.append(trainingE);

      plt.figure(1, figsize=(10, 10))
      plt.plot(trainingError)
      plt.ylabel('The loss')
      plt.xlabel('The number of updates')
      plt.title('The loss vs the number of updates')
      argminDistance = np.argmin(distanceMatrix, axis=1);
      for i in range(num_pts):
            if (argminDistance[i]==0):
                  colors.append('blue');
                  cluster1+=1;
            elif (argminDistance[i]==1):
                  colors.append('green');
                  cluster2+=1;
            elif (argminDistance[i]==2):
                  colors.append('red');
                  cluster3+=1;
            elif (argminDistance[i]==3):
                  colors.append('yellow');
                  cluster4+=1;
            elif (argminDistance[i]==4):
                  colors.append('orange');
                  cluster5+=1;

      plt.figure(2, figsize=(10, 10))
      plt.scatter(data[:,0], data[:,1], c=colors);
      plt.scatter(MU.eval()[:,0], MU.eval()[:,1], c='black');

      if (is_valid):
            plt.figure(3, figsize=(10, 10))
            plt.plot(validationError)
            plt.ylabel('The validation loss')
            plt.xlabel('The number of updates')
            plt.title('The validation loss vs the number of updates')

      plt.show()

      print('trainDataError',trainingError[-1]);
      if (is_valid):
            print('validationDataError',validationError[-1]);
            print('trainDataError to validationDataError ratio: ',
validationError[-1]/trainingError[-1])
      print('cluster 1 contains',cluster1/num_pts*100,'% of the points');
      print('cluster 2 contains',cluster2/num_pts*100,'% of the points');
      print('cluster 3 contains',cluster3/num_pts*100,'% of the points');
      print('cluster 4 contains',cluster4/num_pts*100,'% of the points');
      print('cluster 5 contains',cluster5/num_pts*100,'% of the points');
```

```
K_means(1);
K_means(2);
K_means(3);
K_means(4);
K_means(5);
```

With K=3, the following plot of “the loss vs the number of updates” is generated:

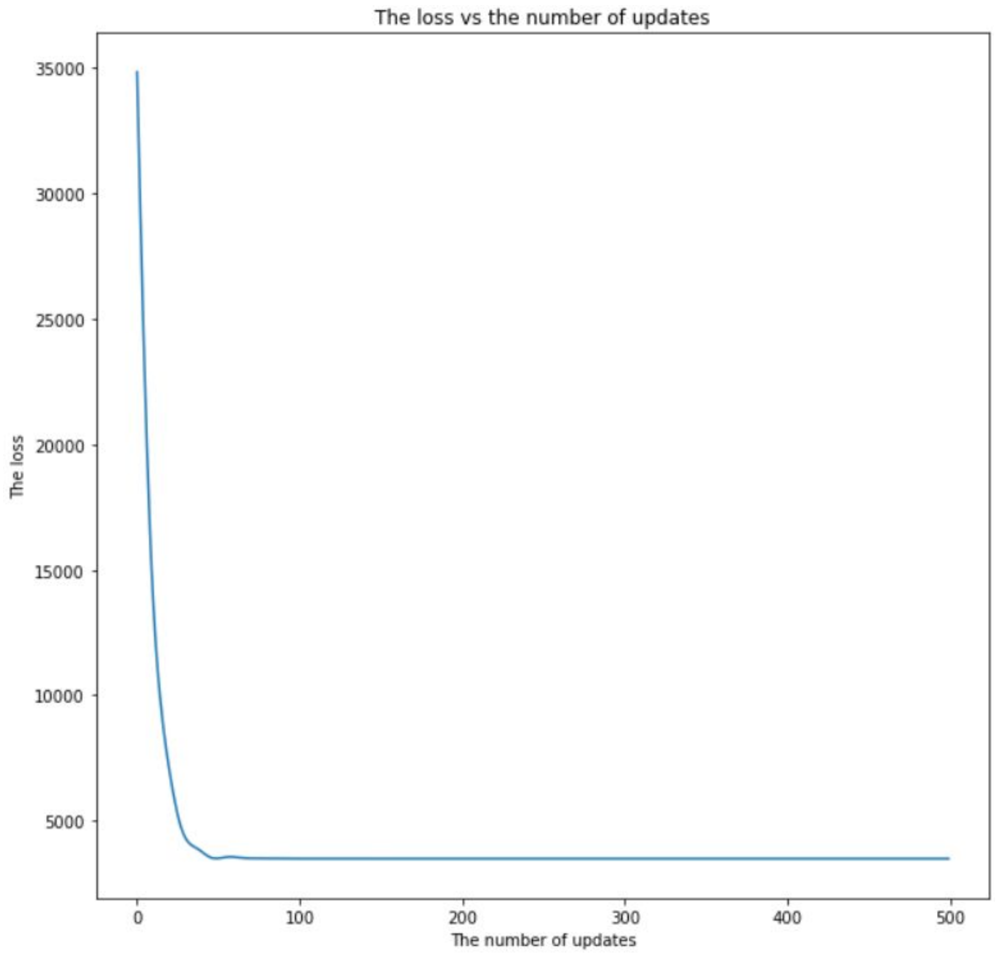


Figure 1.1.1.A - The Loss vs the Number of Updates with K =3

1.1.2 Run with Different Values of K

With the data scatter plot listed below, we can see that there are two obviously dense circles of points on top right and bottom left, followed by one scattered circle of points in the middle which is also worth noting. Thus visually, the percentage data of K=3 follows this observation well. Additionally, from the percentage data alone, we can see that the balancing weakens when K increases even more. Therefore, K=3 gives the best clustering.

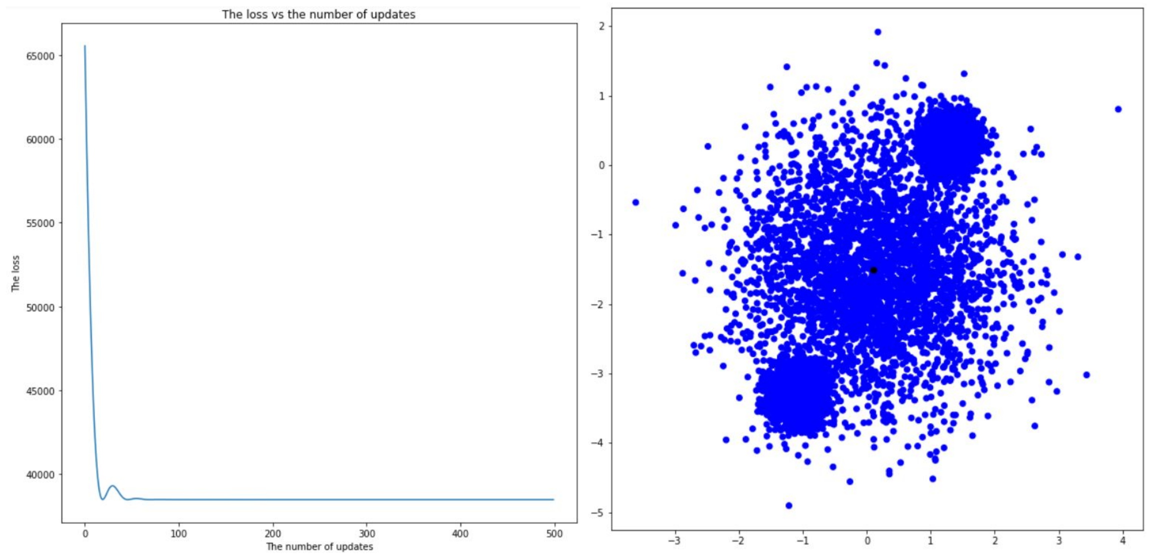


Figure 1.1.2.A - The Loss vs the number of Updates & Learning Results with K =1

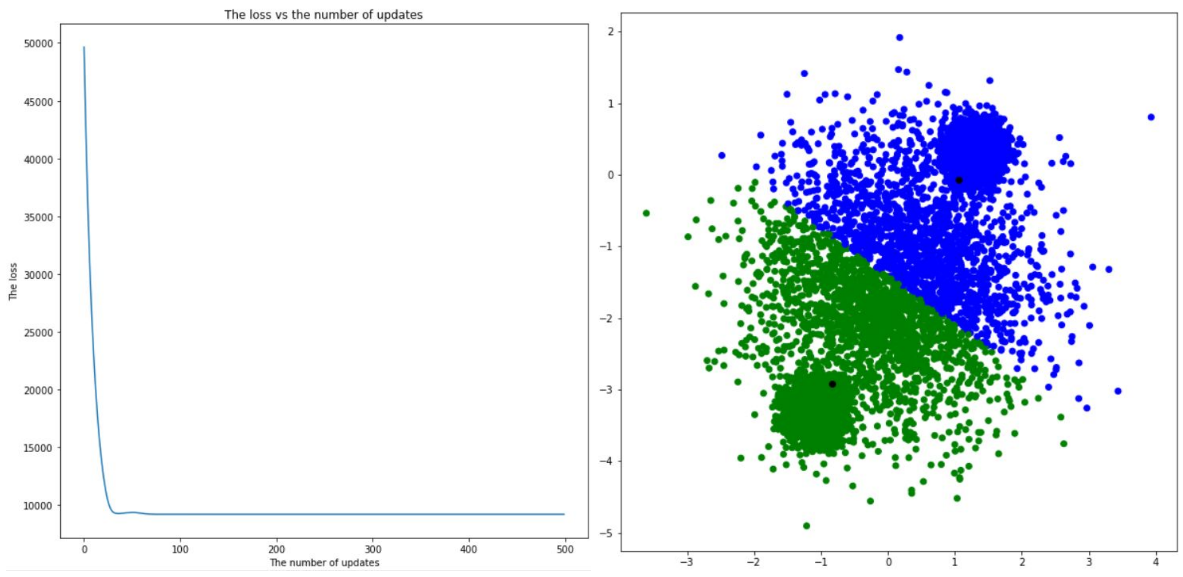


Figure 1.1.2.B - The Loss vs the number of Updates & Learning Results with K =2

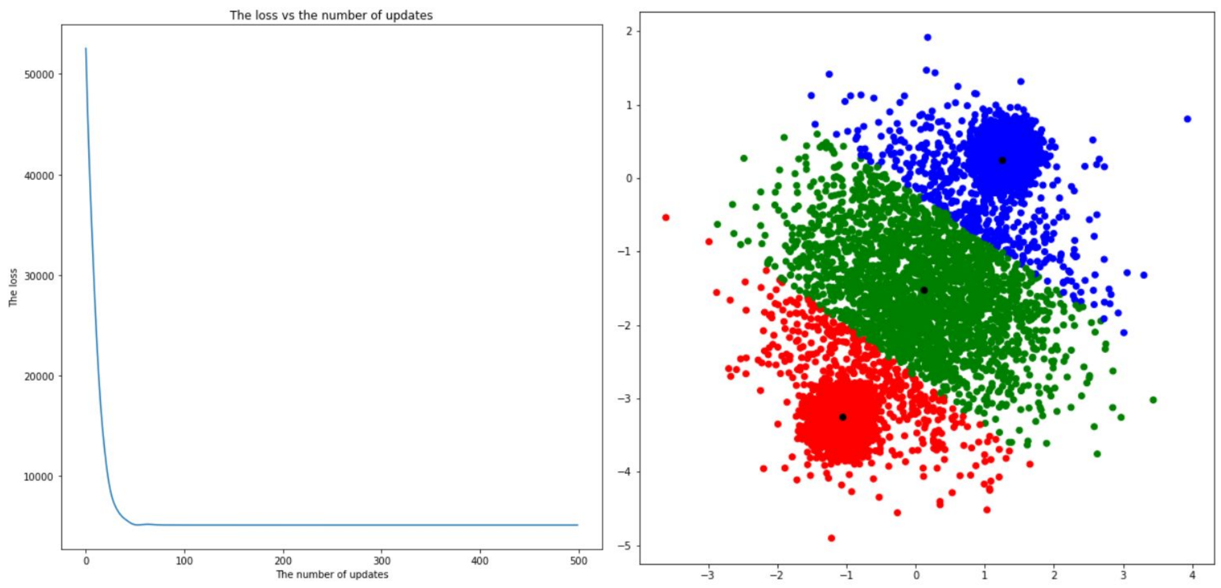


Figure 1.1.2.C - The Loss vs the number of Updates & Learning Results with K =3

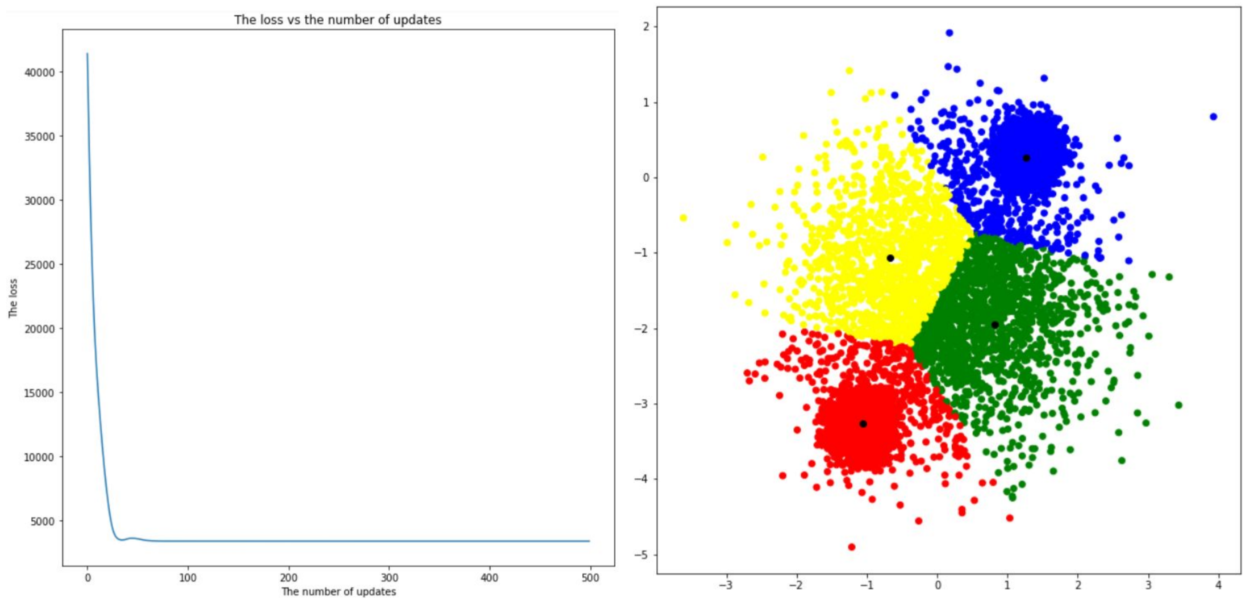
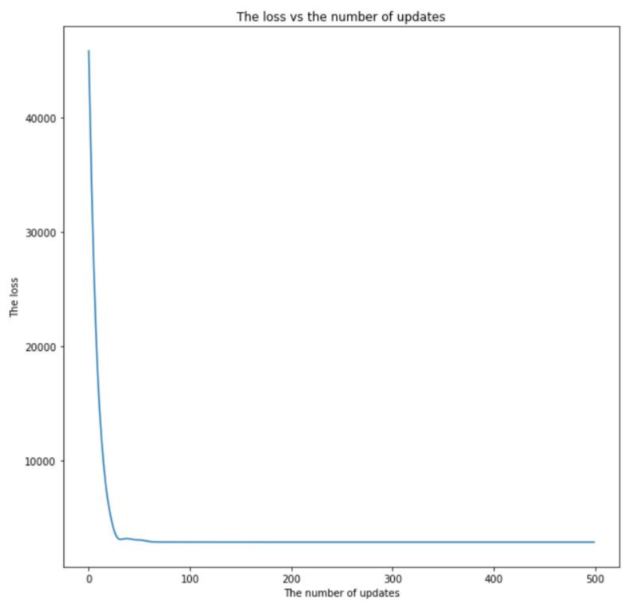


Figure 1.1.2.D - The Loss vs the number of Updates & Learning Results with K =4



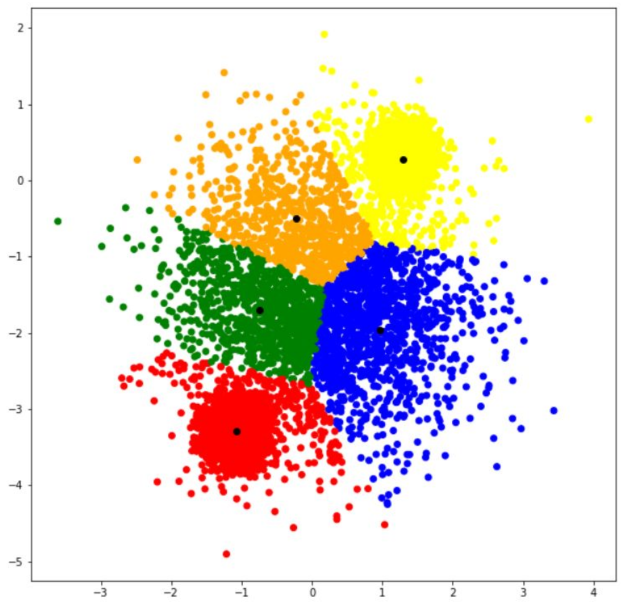


Figure 1.1.2.E - The Loss vs the number of Updates & Learning Results with K =5

Putting the percentage data into tabular form:

	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
K=1	100.0 %				
K=2	49.55 %	50.45 %			
K=3	38.06 %	23.81 %	38.13 %		
K=4	37.29 %	13.49 %	37.13 %	12.09 %	
K=5	11.36 %	8.87 %	36.3 %	35.92 %	7.55 %

Figure 1.1.2.E - Cluster Assignment in Percentage with Different Number of Clusters

1.1.3 Run with Validation data held

The following code snippets are added to utilize the validation data and help calculate the validation loss without touching the tensorflow training implementation.

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

is_valid = 1;

# 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:]]

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

```
def validationLoss(MU):
      N = val_data.shape[0];
      K = MU.shape[0];
      val_data_repeat = np.repeat(val_data,repeats=K,axis=0);
      MU_tile = np.tile(MU, [N,1]);
      reducedSum = np.sum(np.square(val_data_repeat-MU_tile),1);
      pair_dist = np.reshape(reducedSum,[-1,K]);
      validLoss = np.sum(np.amin(pair_dist, axis=1));
      return validLoss;
```

To compare the ending training loss and validation loss, the following table is generated.

	Training loss	Validation loss	Ratio
K=1	25588.998	12870.103913659343	0.5029545857982929
K=2	6243.3345	2960.6728850765244	0.47421340279674223
K=3	3489.1765	1629.3094087808217	0.4669610157000052
K=4	2320.153	1054.5379253634553	0.4545122199882539
K=5	1970.1632	900.9628782440909	0.45730367645791414

It can be observed that the training loss decreases steadily as K increases. This is not surprising, as the training loss can be eventually 0 when K=number of data points. On the other hand, we can see that the ratio between the training loss is about two times of the validation loss. This can also be explained because ⅓ of the data points are held for validation purposes, and that is half of the training data set. The ratio is seen to be increasing slightly when K=5 compared to K=4, making K=4 the local minima. By considering this ratio data alone, I would say that K=4 provides the best clustering. However, comparing this ratio when K=4 to the case where K=3, the advantage is not overwhelming. If all the aspects are to be weighed together, I would still say that K=3 best classifies the data sets, because visually it makes more sense as described in 1.1.2 part of the question.

2 Mixtures of Gaussians

2.1 The Guassian cluster mode

2.1.1 Log probability density function

$$logN(x; \mu_k, \sigma_k^2) = log(\frac{1}{(2\pi\sigma_k^2)^{\frac{d}{2}}} e^{-\frac{(x-\mu_k)^2}{2\sigma_k^2}})$$

$$= \ -\frac{d}{2}log(2\pi\ \sigma_k^2)\ -\ \frac{(x-\mu_k)^2}{2\sigma_k^2}$$

The Python Implementation of this equation is presented below:

```
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

      # TODO
      N = x.get_shape().as_list()[0];
      pair_dist = distanceFunc(x,mu)
      sigma_repeat = tf.repeat(tf.transpose(sigma),repeats = N, axis=0);
      result = -(dim/2)*tf.log(2*np.pi*sigma_repeat) -
                                   (pair_dist)/(2*sigma_repeat)

      return result;
```

2.1.2 Vectorized log probability

$$log(P(z = k \mid x))\ =\ log(\frac{P(x, z{=}k)}{\sum_{j=1}^{K} P(x, z{=}j)})$$

$$= log(\frac{P(z=k)\,P(x|z=k)}{\sum_{j=1}^{K} P(z=j)P(x|z=j)})$$

$$= log(P(z=k)) + log(P(x|z=k)) - log(\sum_{j=1}^{K} P(z=j)P(x|z=j))$$

$$= log(P(z=k)) + log(P(x|z=k))$$
$$- log(\sum_{j=1}^{K} exp(log(P(z=j)) + log(P(x|z=j))))$$

From the above equation we can see that we need to use the *reduce_logsumexp* function in the helper file. Since the the probabilities $P(z=k))$ and $(P(x|z=k)$ are both calculated in the log domain, it is much easier if we use *reduce_logsumexp* to calculate the summation in the third term in the above equation in the log domain as well, instead of using *tf.reduce_sum*.

Here is the python implementation:

```
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

    log_pi = tf.squeeze(log_pi)
    return log_PDF + log_pi - hlp.reduce_logsumexp(log_PDF + log_pi, 1,
                                                                True)
```

2.2 Learning the MoG

2.2.1 Implementation with data2D.npy K=3

A MoG model is trained for the dataset data2D.npy, setting K = 3. The loss vs number of updates plot is shown in the figure below.

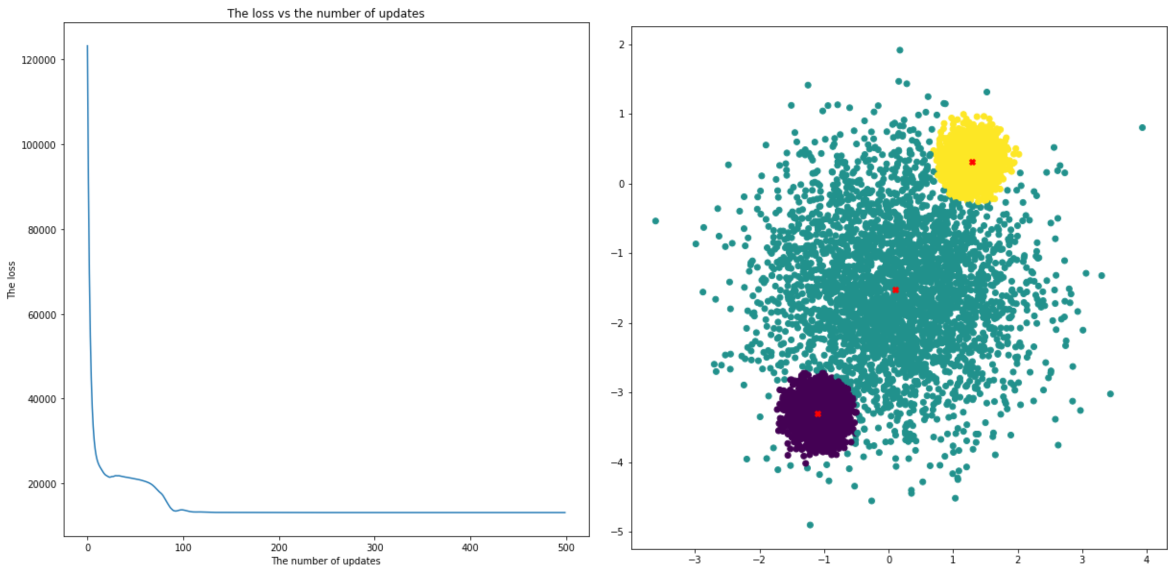


Figure 2.2.1.A - The Loss vs the Number of Updates & Learning Results

Mean	□	σ
`0.10610254 -1.5273092`	`-1.0946859`	`0.987211`
`-1.1030831  -3.3059728`	`-1.1029606`	`0.03907813`
`1.2933022   0.30241516`	`-1.0982077`	`0.03893004`

Table 2.2.1.A - The Parameters Learnt by MoG Model after Training

2.2.2 Running with Different K

1/3 of the data was held out for validation and for each value of K in {1 2 3 4 5}, a MoG model was trained. For each value of K, the loss function for the validation data was presented in Figure 2.2.2.A.The 2D scatter plots of data points clusters are presented in Figure 2.2.2.B to 2.2.2.F.

From the 2D plots and the validation loss graph we can see that **K = 3** is the best value. For similar reasons from Section 1, we can see the two dense clusters of dots in the two corners. Also, the validation loss decreases by a considerable amount K increases from 1 to 2 and 3, but it stops decreasing when K keeps increases to 4 and 5.

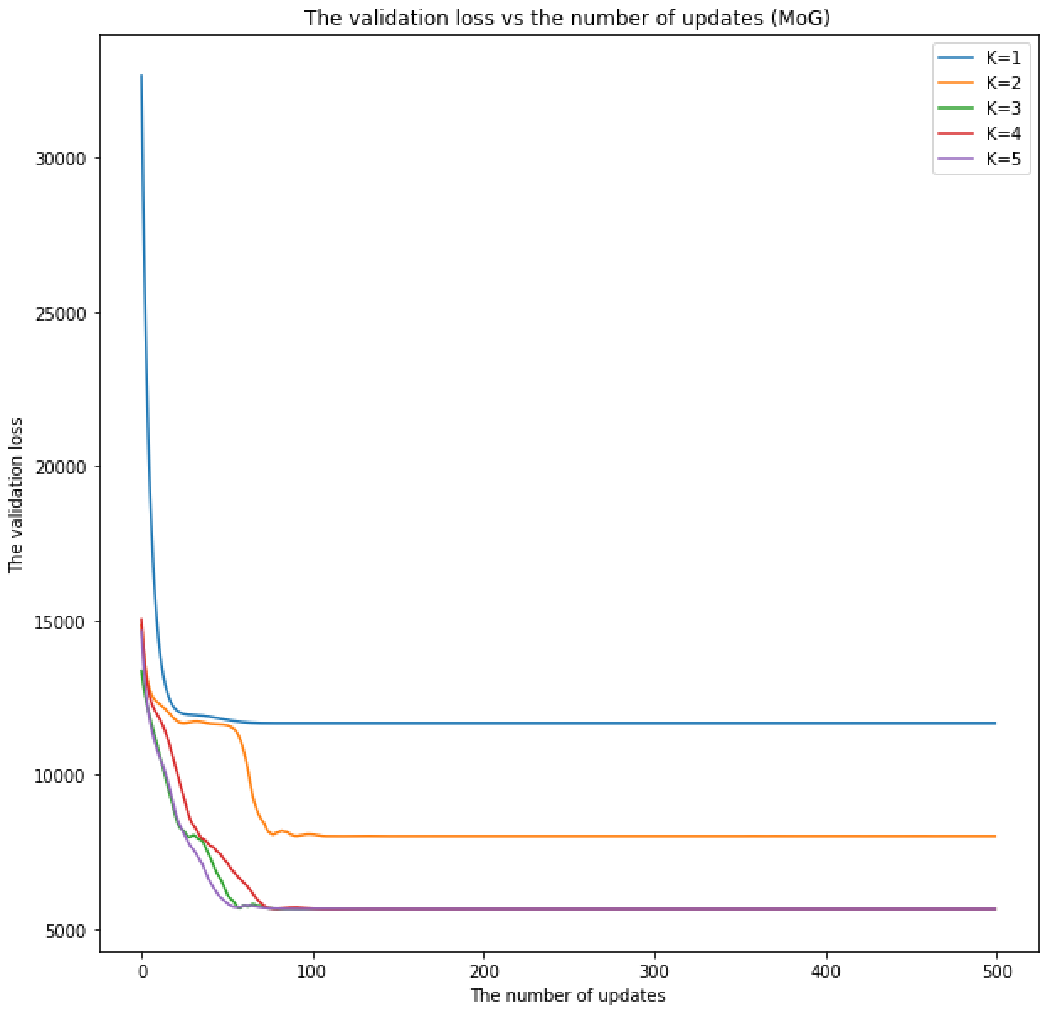


Figure 2.2.2.A - The Validation Loss curve for different K values

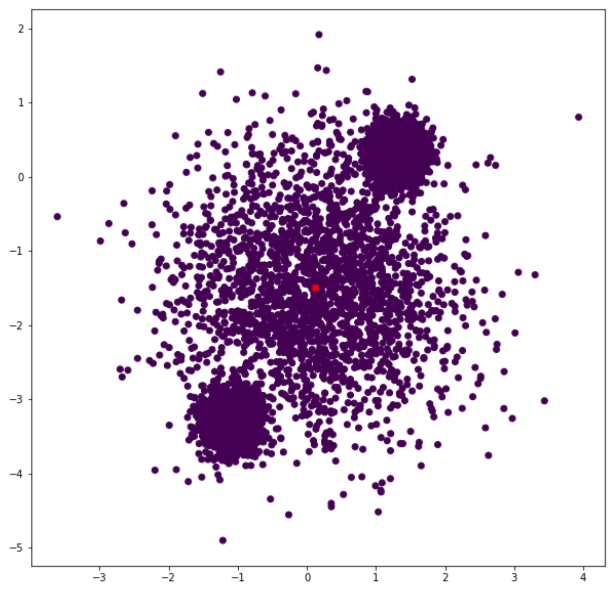


Figure 2.2.2.B - Data Clusters for K=1

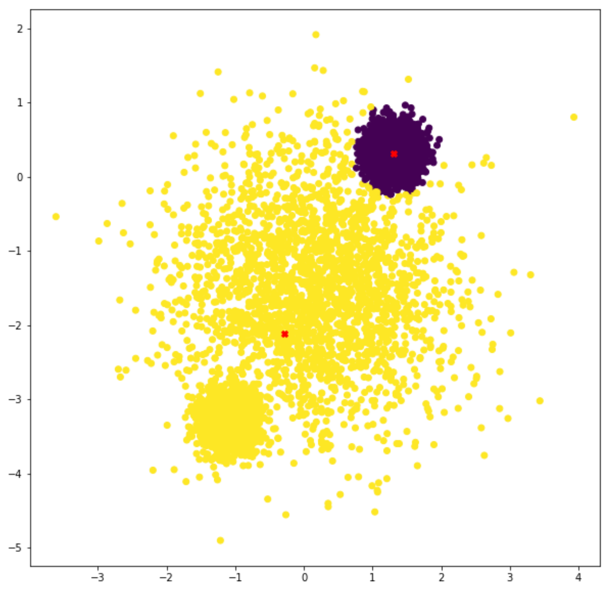


Figure 2.2.2.B - Data Clusters for K=2

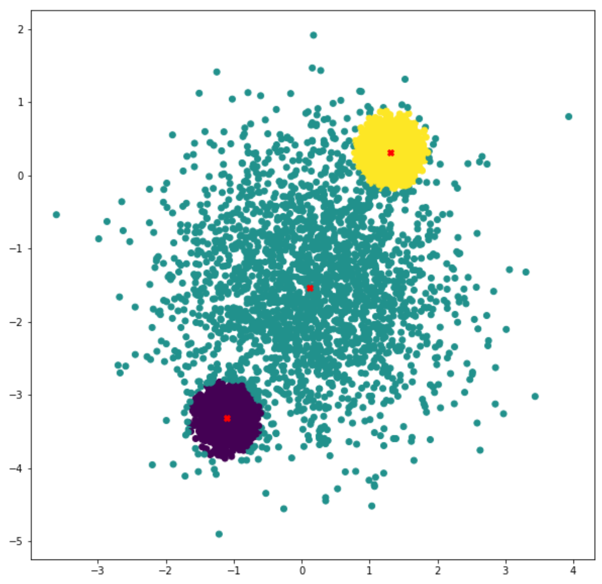


Figure 2.2.2.B - Data Clusters for K=3

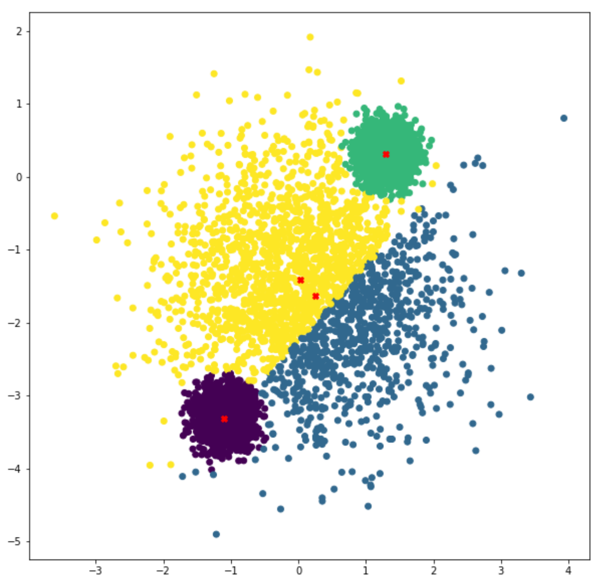


Figure 2.2.2.B - Data Clusters for K=4

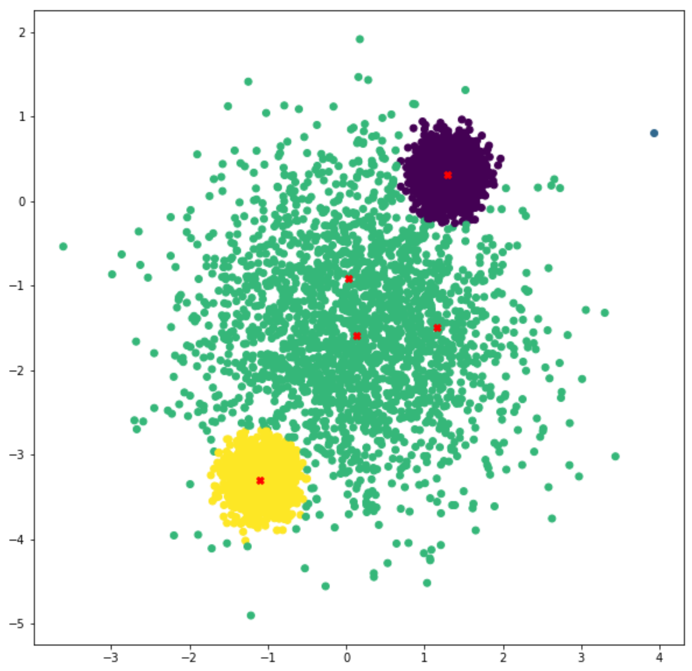


Figure 2.2.2.B - Data Clusters for K=5

2.2.3 Comparison of K-means and MoG with Higher Dimension Data Set

K=10 is confirmed to be the best value by both running results. The running results of the K-means learning algorithm on data100D.npy for K = {5, 10, 15, 20, 30} with 1/3 held for validation are shown in Figure 2.2.3.A. The value K=10 appears to be the best value, based on the observations from the validation loss vs number of updates plot. The final validation loss drops a lot from K=5 to K=10 and remains almost the same for any larger values of K, and from the experience from the last sections, this means K=10 is the best value.

The running results of the MoG learning algorithm on data100D.npy for the Ks are shown in Figure 2.2.3.B. Similarly, the value K=10 also appears to be the best value, for the same reason. By looking into the final validation loss, we see that K-means provides the smaller validation loss, yet the loss function for MoG and K-means are different, the numerical value cannot be directly compared.

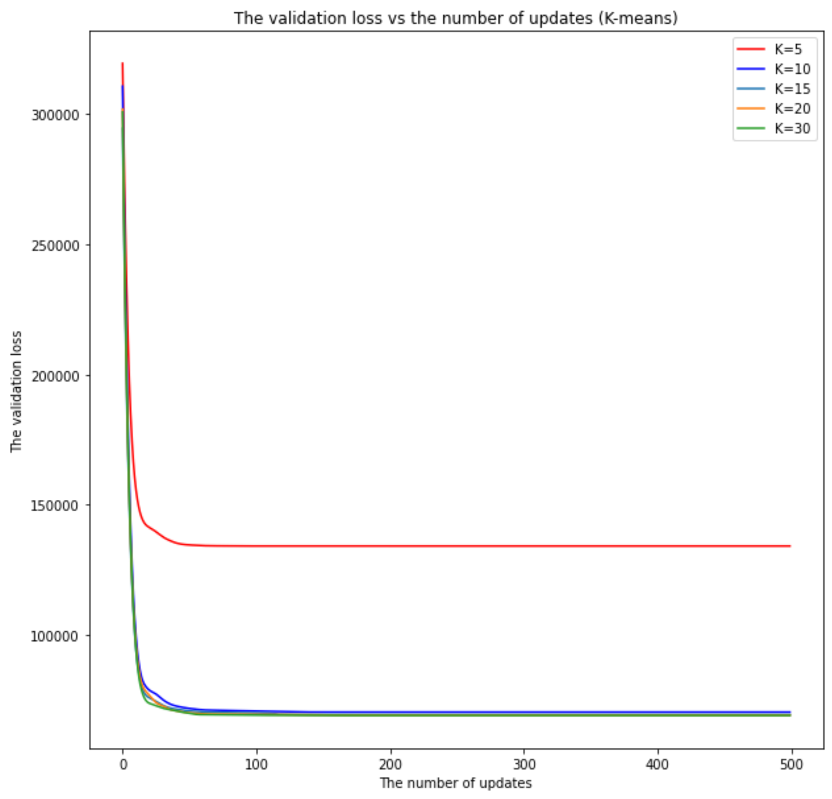


Figure 2.2.3.A - The K-means Validation Loss vs the Number of Updates

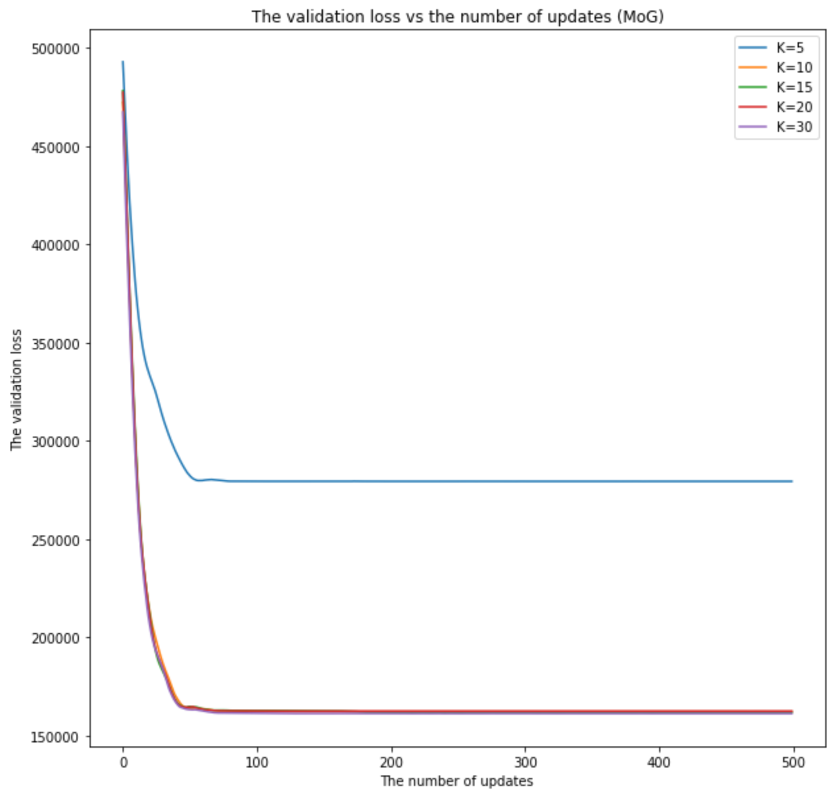


Figure 2.2.3.B - The MoG Validation Loss vs the Number of Updates

data100D MoG Running Results:

Number of Clusters	Final Training Error	Final Validation Error
5	563307.8	279439.44
10	321005.6	162338.42
15	320354.3	161915.56
20	321909.53	162598.36
30	317760.97	161380.77

Table 2.2.3.A - The MoG Training and Validation Loss vs the Number of Updates

data100D K-means Running Results:

Number of Clusters	Final Training Error	Final Validation Error
5	291564.31	145982.56
10	142206.19	71172.09
15	137829.03	69450.24
20	137412.06	69320.05
30	134691.31	68399.23

Table 2.2.3.B - The K-means Training and Validation Loss vs the Number of Updates