Assignment 3

ECE421
April 8, 2020

Maxwell Fingold 1003915882
Andy Xie 1004102650

1. Introduction

In this report, we will implement and test two different machine learning algorithms which operate by optimizing a loss function over clusters of data points from two different datasets. The two algorithms are K-Means and a Mixture of Gaussians Model. They both operate by minimizing distances from clusters centers to data points, but the Mixture of Gaussian Model operates using a probabilistic structure whereas K-Means operates more deterministically.

Implementations of the algorithms will be in Python Tensorflow, and theoretical derivations of what is implemented will also be shown. We also show the outputs of these algorithms after training them for different numbers of clusters on the two different datasets data2D.npy and data100D.npy.

2. K-Means

We will begin by implementing and examining the use of the K-Means learning algorithm. This algorithm is used to determine the centers, μ_K, of K clusters of data points from the data set D. It determines these centers by minimizing the following loss function:

$$\mathcal{L}(\mu) = \sum_{n=1}^{N} \min_{k=1}^{K} \|\mathbf{X}_n - \mu_k\|_2^2$$

The variables in the equation above are defined as follows:

$$\mu = [\mu_1, \mu_2, \ldots, \mu_K]$$
$$\mathcal{D} = [\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_N]$$

2.1.1 Implementation of K-Means Algorithm

Before implementing any sort of gradient descent process, we must first have a method to compute the loss function. The implementation used is shown below in Figure 1 with the distance_func() and loss_function() Python Tensorflow functions.

```python
# 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

    pair_dist = tf.transpose(tf.reduce_sum(tf.square(X - MU), axis=2))
    return pair_dist
```

```python
def loss_function(X, MU):
    dist = distanceFunc(X, MU) #get distances from data points to cluster means
    error = tf.reduce_min(dist, axis=1) #get smallest cluster mean-point distances
    loss = tf.reduce_sum(error) #calculate error
    return loss
```

Figure 1: distanceFunc() and loss_function() implementations for K-Means

The minimization process alluded to in the previous section is implemented using Tensorflow and the Adam Optimizer. All centers, $\boldsymbol{\mu}_k$, are initialized by sampling from a standard normal distribution, which has mean and standard deviation 0 and 1 respectively. These values are stored in matrix MU. Values for data points are imported from the file data2D.npy. The implementation of the training process can be seen below in Figure 2.

```python
with tf.Session() as training_loop:
  tf.initializers.global_variables().run()

  training_loss = []
  validation_loss = []
  for epoch in range(150):
    new_MU, train_loss, new_dist, _ = training_loop.run([MU, loss, dist, optimizer])

    if is_valid:
      v_loss = loss_function(val_data,new_MU)
      v_dist = distanceFunc(val_data, new_MU)
      with tf.Session() as get_loss:
        tf.initializers.global_variables().run()
        valid_loss, valid_dist = get_loss.run([v_loss,v_dist])
        validation_loss.append(valid_loss)

    training_loss.append(train_loss)
```

Figure 2: Tensorflow Implementation of K-Means Training

The implementation above was run for K=3 clusters, with the following parameters passed to the Adam Optimizer: a learning rate of 0.1; a beta_1 of 0.9; a beta_2 of 0.99; and an epsilon of 1e-

05. The results of this simulation are represented below in Figure 3.

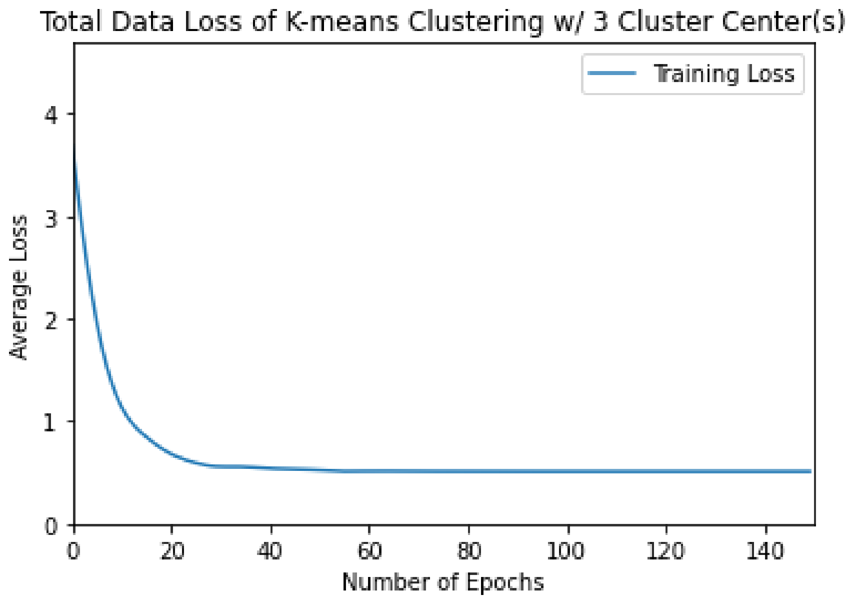


Figure 3: Training Loss for 3 Clusters, 100 Epochs

2.1.2 Varying Number of Clusters

We will now examine how varying the number of clusters changes how the data points from data2D.npy are grouped. In Figures 4 through 8 below, the training loss and scatter plots for the clustered data points with their respective centers are shown. The centers of each cluster are represented by a large X, and each cluster is shown by a separate color. As well, the percentages of data points which fall into each cluster for each trial are listed in Table 2.

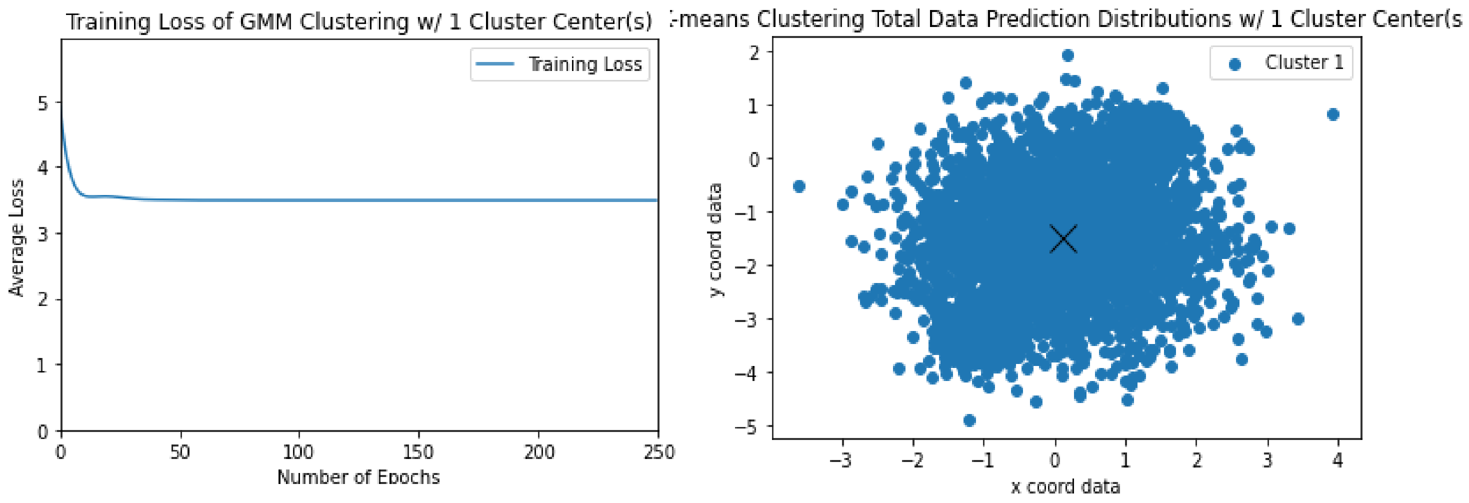


Figure 4: Training Loss and Clustering for K=1 Cluster

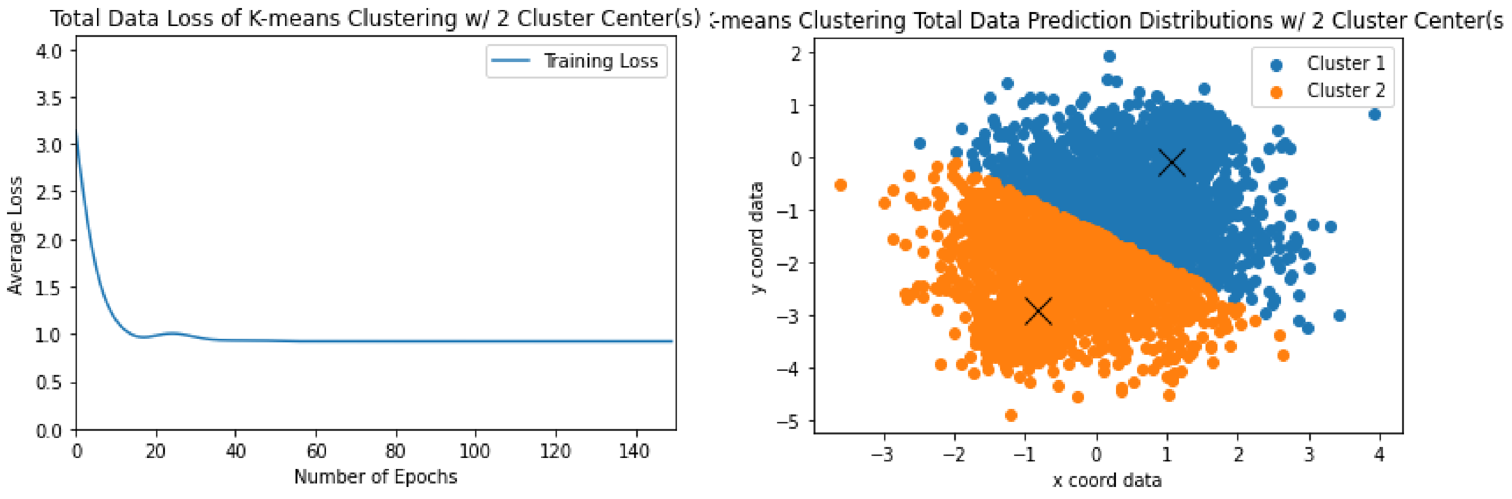


Figure 5: Training Loss and Clustering for K=2 Clusters

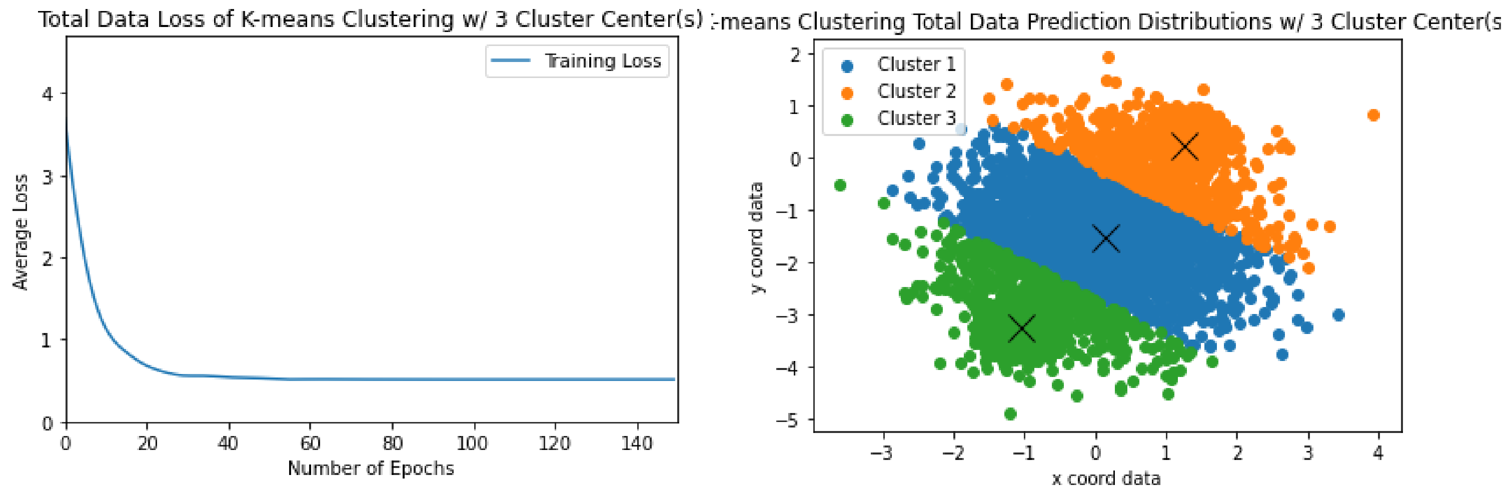


Figure 6: Training Loss and Clustering for K=3 Clusters

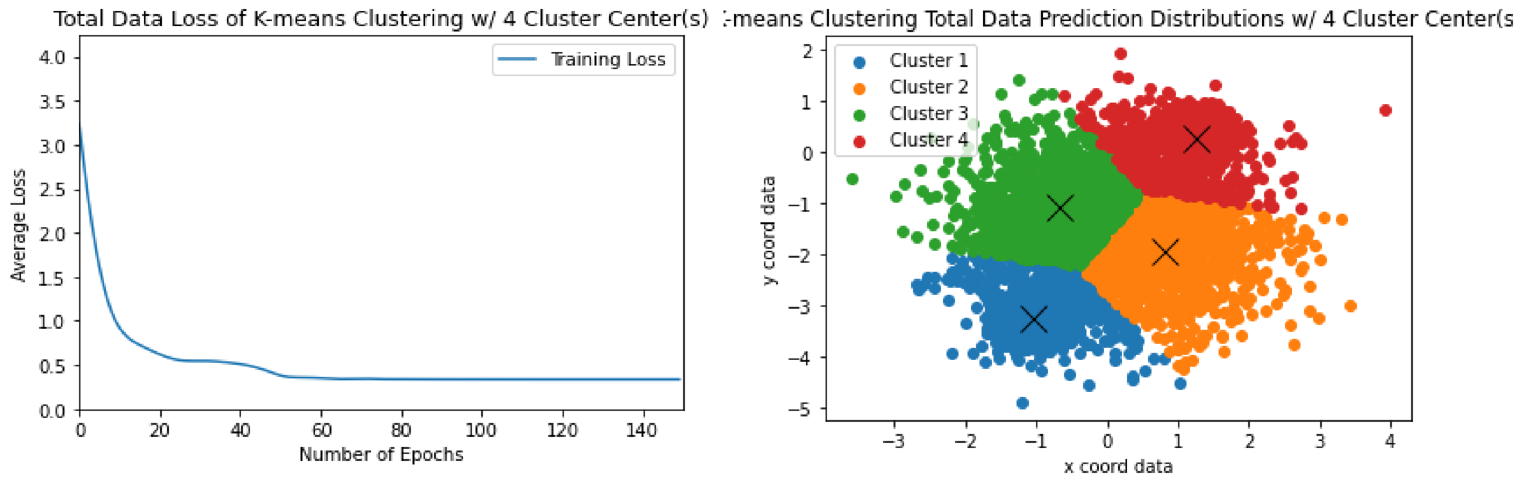
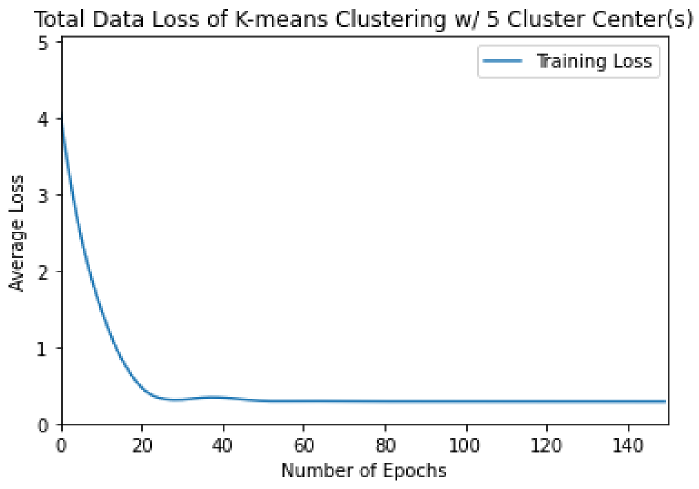


Figure 7: Training Loss and Clustering for K=4 Clusters



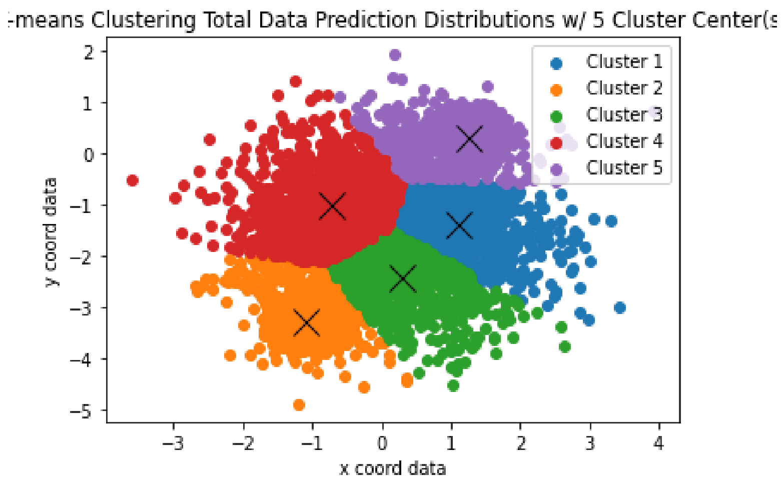


Figure 8: Training Loss and Clustering for K=5 Clusters

	Total Training Loss	Average Training Loss
K=1 Clusters	38453.489	3.849
K=2 Clusters	9203.382	0.920
K=3 Clusters	5110.954	0.511
K=4 Clusters	3374.079	0.337
K=5 Clusters	2848.408	0.086

Table 1: Total and Average Final Training Losses for K=1 to 5 clusters in K-Means Learning w/o Validation.

	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
K=1 Clusters	1	0	0	0	0
K=2 Clusters	0.505	0.495	0	0	0
K=3 Clusters	0.238	0.381	0.381	0	0
K=4 Clusters	0.371	0.135	0.120	0.373	0
K=5 Clusters	0.086	0.358	0.088	0.107	0.362

Table 2: Percentage of Data Points which fall into each Cluster for K=1 to 5 cluster trials w/o validation.

It is evident by observing all information in Figure 4, Table 1, and Table 2 that K=1 clusters is insufficient to properly cluster points from the data2D.npy data set. The losses are astronomical relative to the other trials – over four times as large as the next worst case – and clearly an oversimplification of the data if we examine the K=3 and greater trials.

By examining the graphs of the training loss in Figures 4 through 8 and the reported values in Table 1, it can easily be observed that by increasing the number of clusters, average training loss decreases. This is logical as if you were to introduce a new cluster center closer to any data point than any center was before, the Euclidean distance from that point to a center decreases, hence decreasing its training loss, and therefore the average training loss of the data set as a whole.

While this decrease in the loss function exists, by examining Table 2, we see that for K=5, there are some clusters which contain a very low percentage of the data points. This is indicative of overfitting, especially as for K=4 we see much more even distribution of data points between the clusters. While K=3 has the most equal distribution of points between clusters, if we look at the spatial distribution of the clusters in Figure 6, we can see that Cluster 1 occupies a much larger area than Clusters 2 or 3. This is especially noticeable when compared to the K=4 case in Figure 7 and the K=2 case in Figure 5 as those have clusters which appear to clusters with similar areas.

Although the K=2 case does have both an equal split of data points per cluster and clusters with similar areas, the divide appears too simple as there are points with large x and small y, and the converse which are not very well grouped into these clusters. This is seen in the relatively large training loss for K=2 in Figure 5. For these reasons, K=4 clusters appears to be most representative for this data set.

2.1.3 K-Means Training with a Validation Set

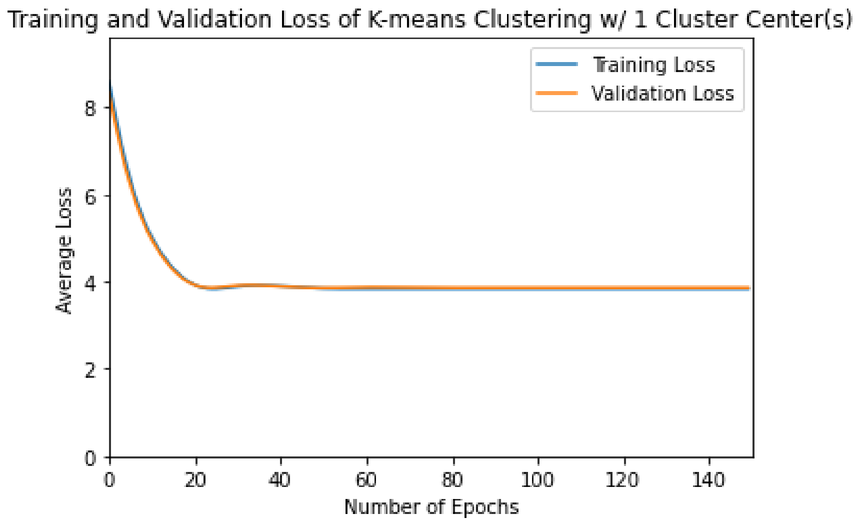
To verify if this method of training is useful, it is important to check if it can properly predict the location of points it has not seen yet. In other words: we must check the validation loss of our K-Means implementation for each of the cluster amounts seen above. To do this, we leave a third of all data points for validation, and the other two thirds for training. The output of these trials is seen below in Figures 9 through 13. In addition, losses and cluster data percentages are listed in Tables 3 and 4 below.

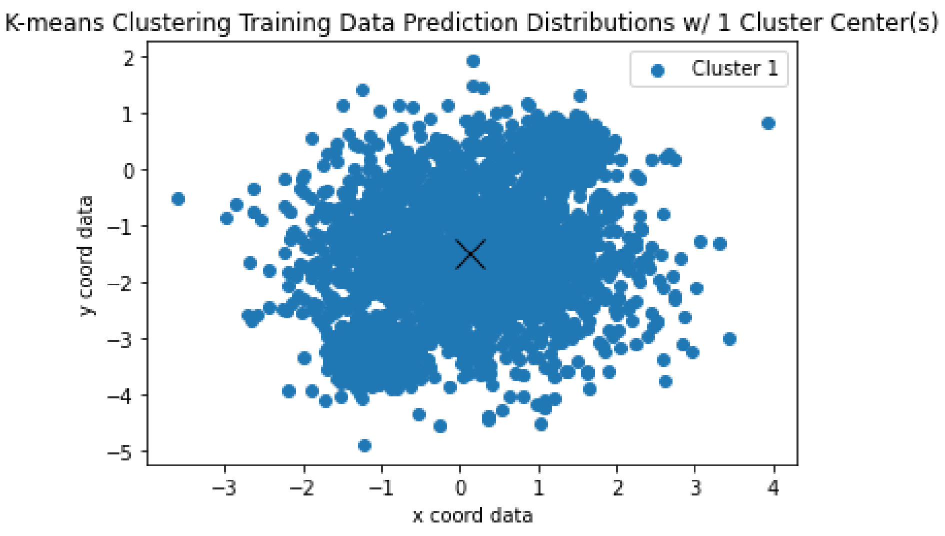
	Total Training Loss	Average Training Loss	Total Valid Loss	Average Valid Loss
K=1 Clusters	25589.049	3.838	12870.242	3.861
K=2 Clusters	6243.454	0.936	2960.510	0.888
K=3 Clusters	3489.996	0.523	1629.394	0.489
K=4 Clusters	2320.507	0.348	1054.241	0.316
K=5 Clusters	1960.491	0.294	901.725	0.271

Table 3: Training and Validation Losses for K=1 to 5 Clusters in K-Means Training with Validation

	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
K=1 Clusters	1	0	0	0	0
K=2 Clusters	0.502	0.498	0	0	0
K=3 Clusters	0.384	0.241	0.375	0	0
K=4 Clusters	0.137	0.123	0.377	0.363	0
K=5 Clusters	0.096	0.349	0.092	0.365	0.098

Table 4: Percent of Data Points which fall into each Cluster for K=1 to 5 Cluster Trials with Validation





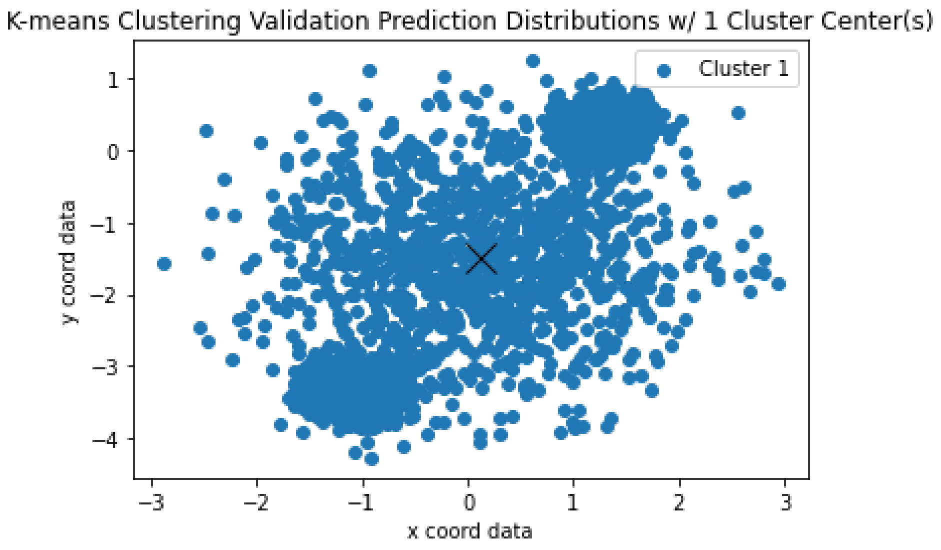
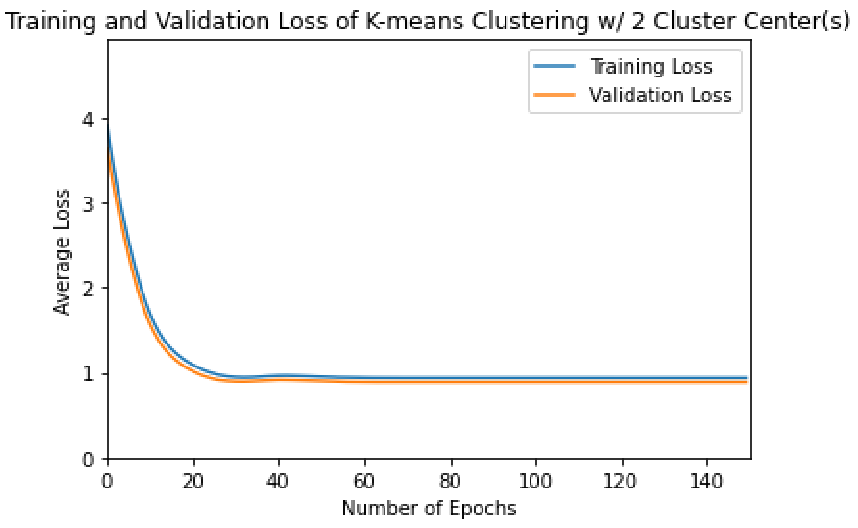


Figure 9: Results from K-Means Training and Validation on K=1 Cluster



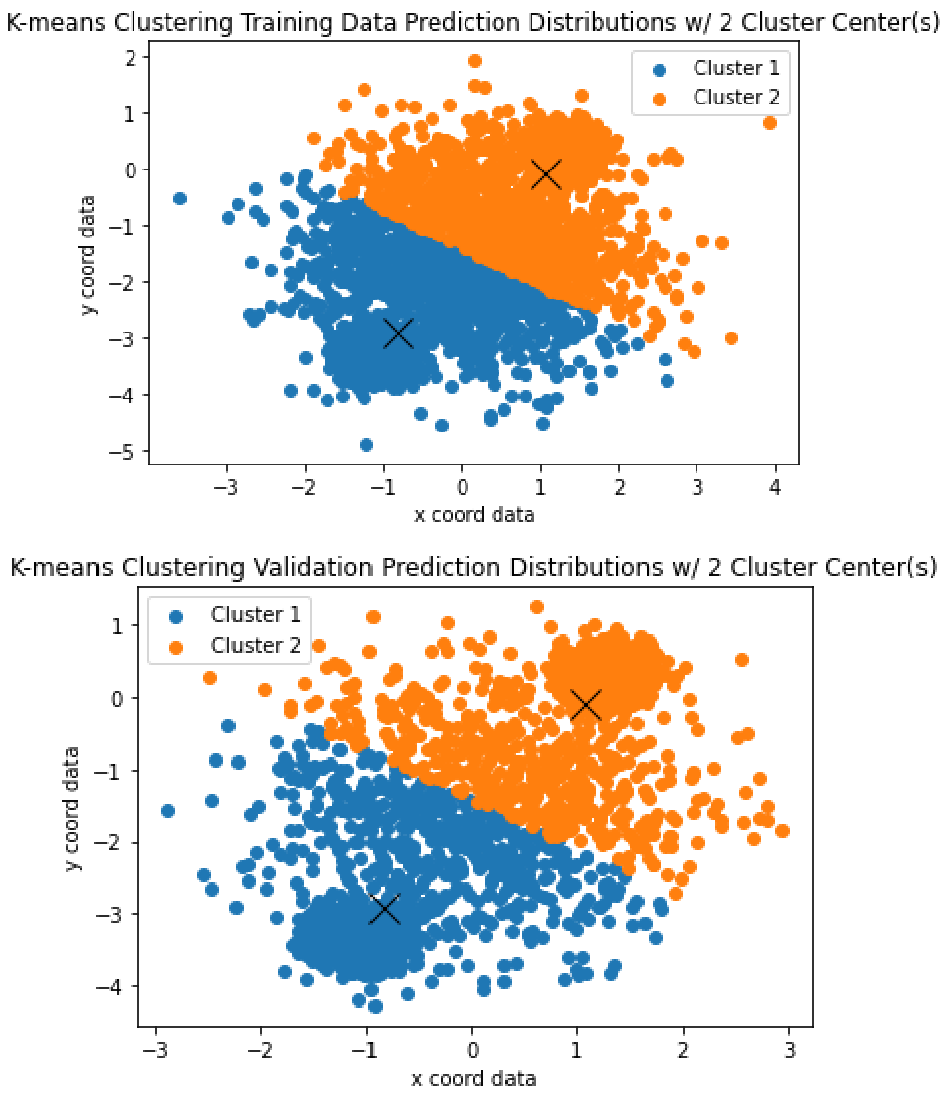
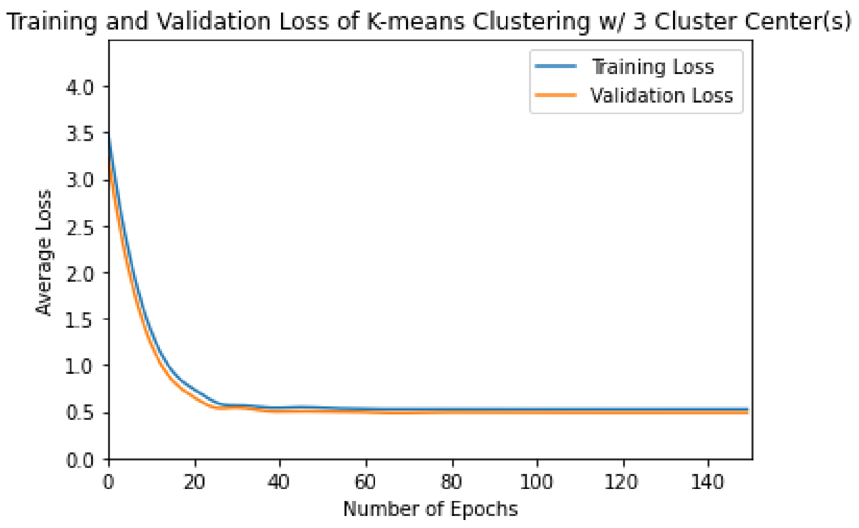
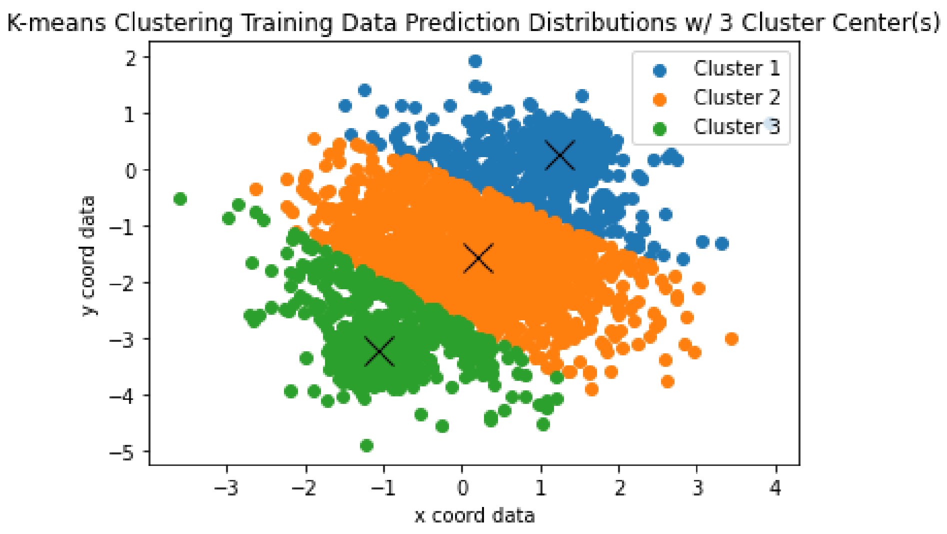


Figure 10: Results from K-Means Training and Validation on K=2 Clusters





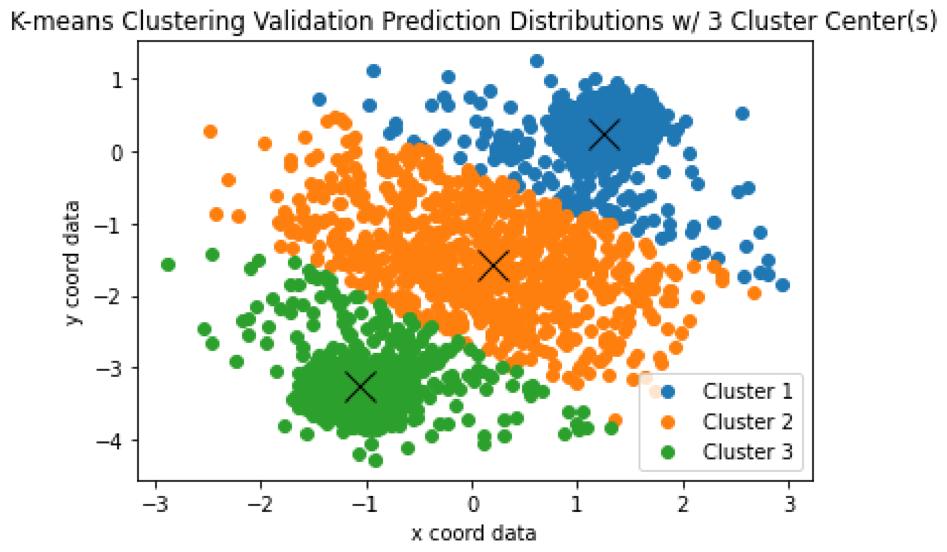
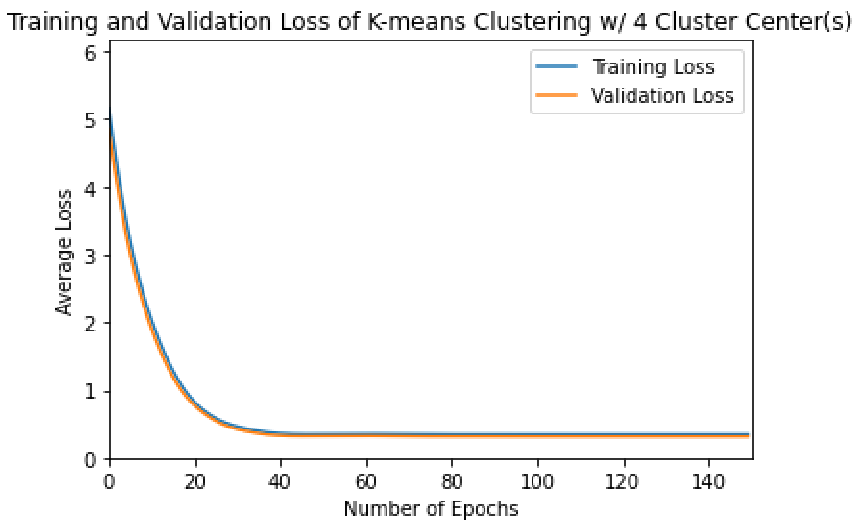
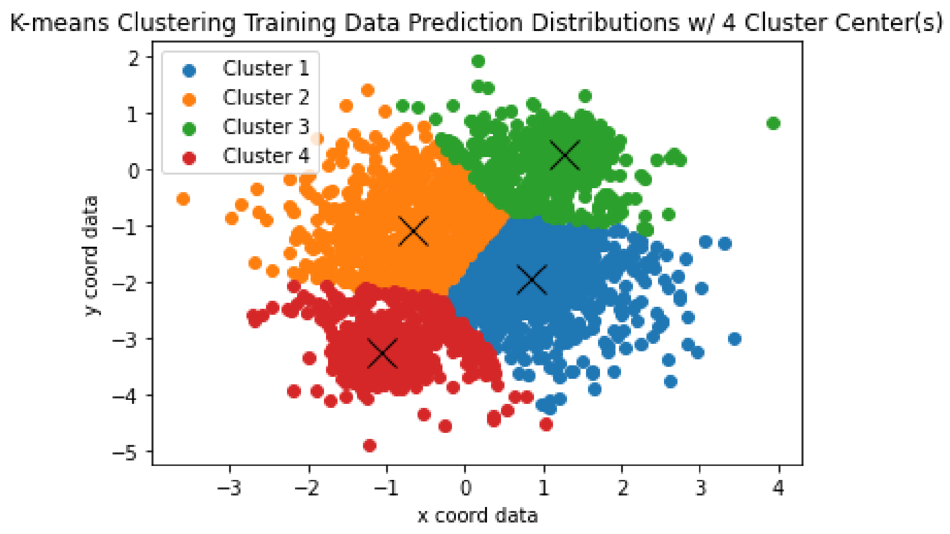


Figure 11: Results from K-Means Training and Validation on K=3 Clusters





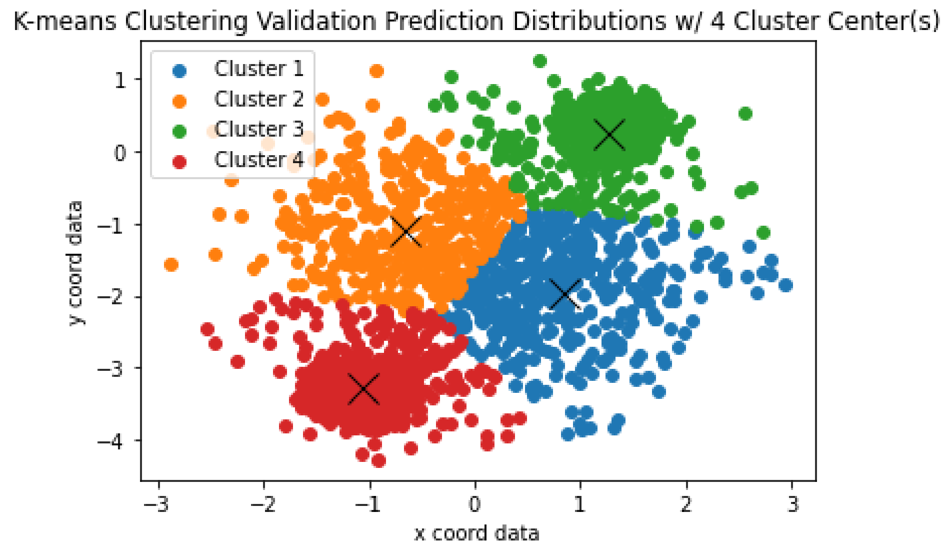
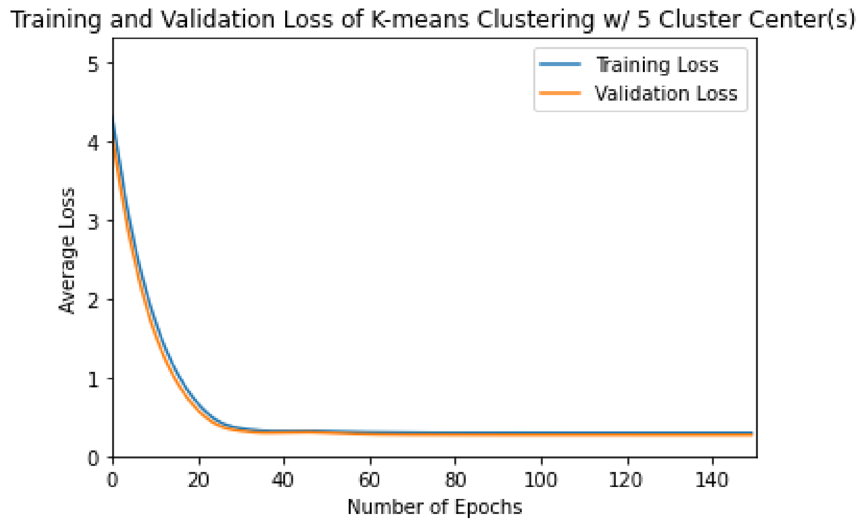
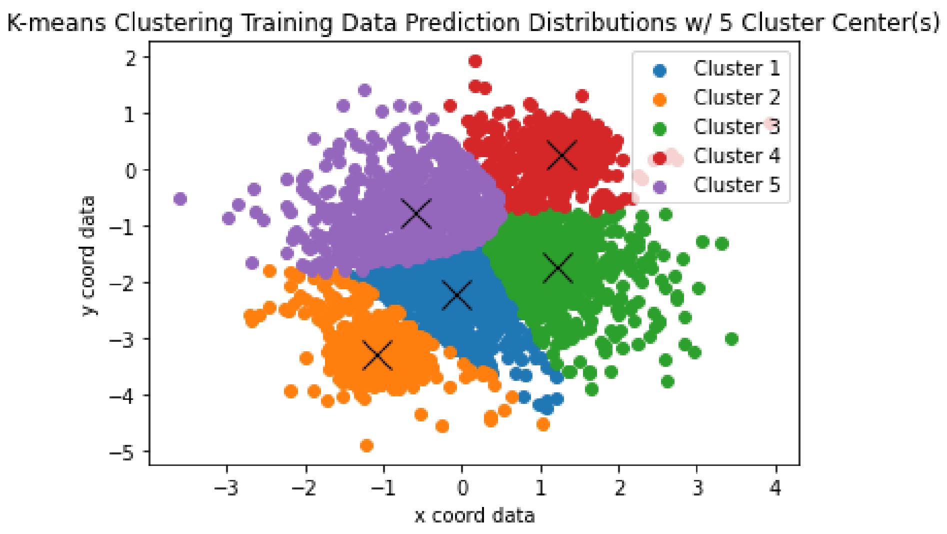


Figure 12: Results from K-Means Training and Validation on K=4 Clusters





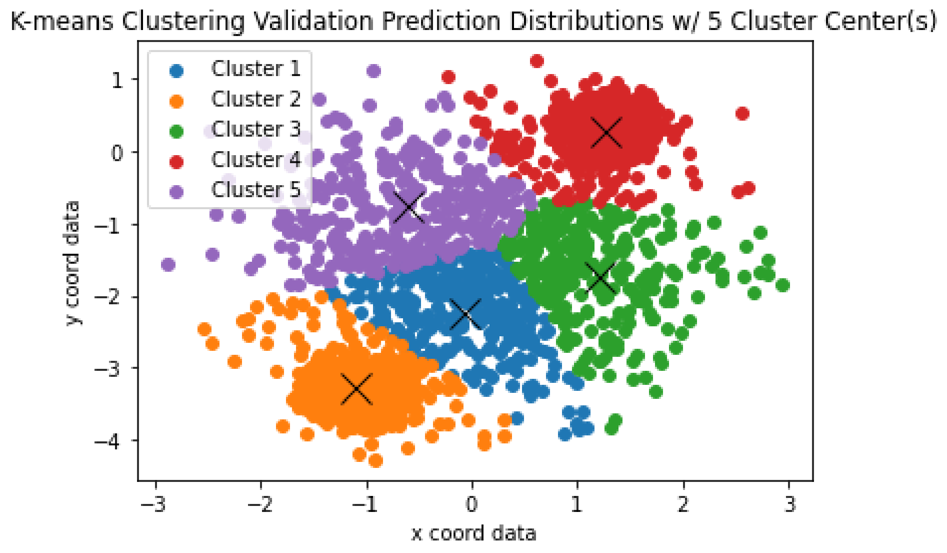


Figure 13: Results from K-Means Training and Validation on K=5 Clusters

For all cases but the K=1 case, average validation loss is lower than average training loss. This shows that K-Means is an effective model for the prediction of how points fit into these clusters. We also see a similar trend to before where losses decrease as the number of clusters increases. All trends that were exemplified in section 2.1.2 are actually all still present, but some have their effects exacerbated.

The most obvious change can be seen in the K=5 clusters case. In the Prediction Distribution graphs in Figure 13, we can see obvious, possibly unnatural distortions in the shape of clusters. Cluster 1 is especially odd, being elongated and somewhat triangular in shape. While average losses are low, this oddly shaped cluster could be indicative of some overfitting. It is also interesting to note how the removal of a third of the data set from training affected the shape of clusters across each case. We can see by comparing Figure 13 and Figure 8 that the cluster shapes greatly changed between each run. While it is possible that points were not removed evenly from across the data set, and that could be causing distortions, if we do a similar comparison for the K=3 and K=4 cases, we do not see that level of change in shape. This further supports the notion that the 5 cluster model is overfitting the data.

In this run, it is more difficult to decide which of the K=3 and K=4 cluster cases better fits the data set. This confusion comes from the placement of the cluster centers seen in the Validation Prediction Distribution for K=4 clusters in Figure 12. These cluster centers are over a much less dense area than was previously expected from examining the graph in Figure 7. This may also indicate some overfitting for the K=4 clusters case. However, the K=3 clusters case still continues to have the issue of containing points very far from the center of Cluster 1 as can be seen in Figure 11. This could still imply that the K=3 case slightly under fits the data set.

This tradeoff between K=3 and K=4 clusters means that it would likely be most beneficial if there was a model that could balance the benefits of both. This is of course impossible due to the discrete nature of the K-Means learning algorithm. Nevertheless, due to the K=4 clusters case having slightly less training and validation loss after converging, we consider the four clusters most accurately represents the distribution of data points in data2D.npy.

3. Mixtures of Gaussians

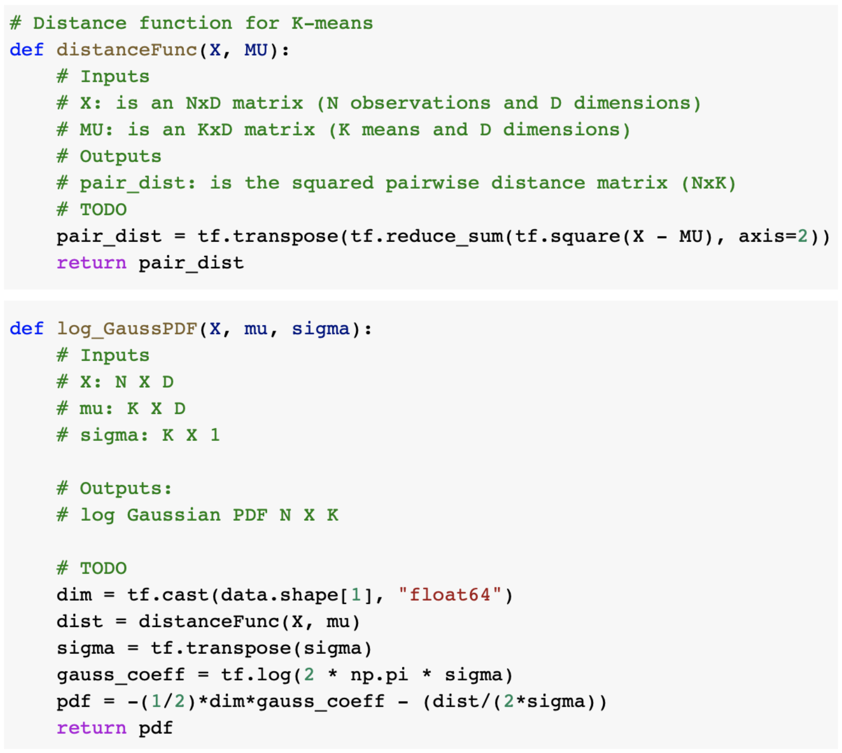
In this section, we will examine and implement a Mixtures of Gaussians (MoG) learning algorithm. While Expectation-Maximum (EM) is generally used for the learning algorithm in MoG models, we will implement a different model due to the slow convergence of EM. It will utilize gradient descent, and have many components based on the methods developed in the K-Means section of this report.

3.1 The Gaussian Cluster Mode

We will begin our MoG model by defining the model which is to be implemented. The model to be used is a multivariate Gaussian distribution centered at each cluster mean, $\boldsymbol{\mu}^k$. As well, all data dimensions are assumed to be independent of each other, and all have equivalent standard deviations, $\boldsymbol{\sigma}^k$. Finally, each of the K mixture components occur with probability π^k.

3.1.1 Implementation of Log Probability Density Function

To calculate the Log Probability Density Function for the kth cluster, the following code shown in Figure 14 was implemented. The distanceFunc() used in K-Means training was maintained in its form as is seen in Figure 1, and the MoG specific implementation can be seen in the log_GaussPDF() function below.



```
# 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
    pair_dist = tf.transpose(tf.reduce_sum(tf.square(X - MU), axis=2))
    return pair_dist
```

```
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
    dim = tf.cast(data.shape[1], "float64")
    dist = distanceFunc(X, mu)
    sigma = tf.transpose(sigma)
    gauss_coeff = tf.log(2 * np.pi * sigma)
    pdf = -(1/2)*dim*gauss_coeff - (dist/(2*sigma))
    return pdf
```

Figure 14: distanceFunc() and log_GaussPDF() Python Tensorflow Implementation

3.1.2 Implementation of Log Conditional Probability Function

Now that we have implemented the function required to compute the probability density function for a cluster, we would like to compute the probability a given data vector is contained within cluster k. In other words, we must compute the following probability:

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

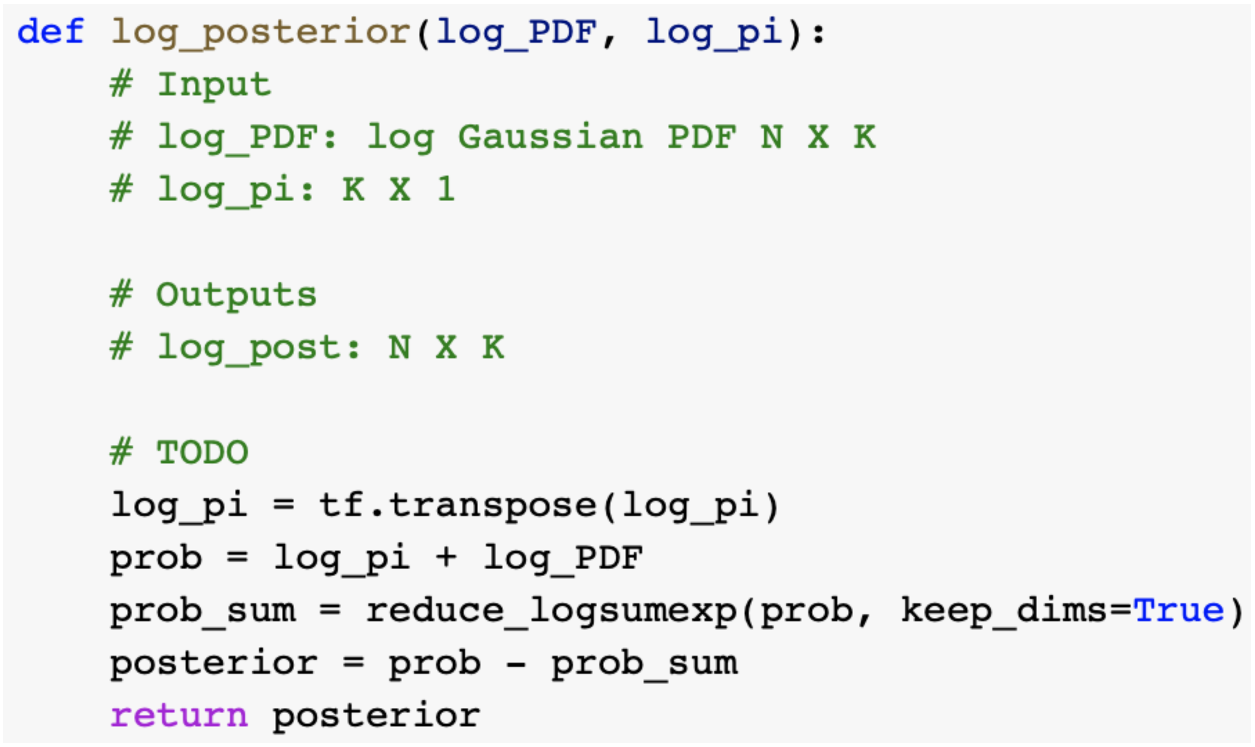
By then applying log rules to the equation above, it can be shown that the probability can also be expressed by the following equation below, which is what is to be implemented.

$$log(P(z=k|\mathbf{X})) = log(P(\mathbf{X}, z=k)) - log(\sum_{j=1}^{K} P(\mathbf{X}, z=j))$$

It is also important to note the following manipulation is invalid, and hence we cannot use tf.reduce_sum but are instead required to use the reduce_logsumexp () helper function.

$$log(\sum_{j=1}^{K} P(\mathbf{X}, z=j)) \neq \sum_{j=1}^{K} log(P(\mathbf{X}, z=j))$$

Our implementation of the Log Conditional Probability Function log_posterior() is shown below in Figure 15. The pure probability can be generated by calling log_posterior(pdf, 0), where 0 is a KX1 vector of zeros.



```python
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

    # TODO
    log_pi = tf.transpose(log_pi)
    prob = log_pi + log_PDF
    prob_sum = reduce_logsumexp(prob, keep_dims=True)
    posterior = prob - prob_sum
    return posterior
```

Figure 15: log_posterior() Python Tensorflow Implementation

3.2 Learning the MoG

Now that we have implemented the necessary helper functions, we will construct the Loss Function to be minimized in the training process. The loss function we will be minimizing is the negative log likelihood, shown in the equation below.

$$\mathcal{L}(\mu, \sigma, \pi) = -log(P(\mathbf{X}))$$

3.2.1 Implementing the Loss Function for MoG

As per the assignment handout [1], the equation for the loss function is equivalent to:

$$\mathcal{L}(\mu, \sigma, \pi) = -log(\prod_n \sum_k \pi^k \mathcal{N}(\mathbf{X}_n; \mu^k, \sigma^{k^2}))$$

By applying log rules to this equation, the loss function can also be expressed as:

$$\mathcal{L}(\mu,\sigma,\pi) = -\sum_n log(\sum_k \pi^k \mathcal{N}(\mathbf{X}_n; \mu^k, {\sigma^k}^2))$$

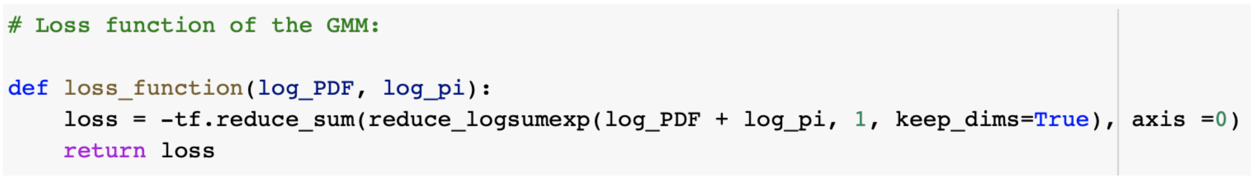
Then, by exponent rules, we receive that:

$$\mathcal{L}(\mu,\sigma,\pi) = -\sum_n log(\sum_k exp(log(\pi^k) + log(\mathcal{N}(\mathbf{X}_n; \mu^k, {\sigma^k}^2))))$$

Finally, we realize that this equation can be represented by the relation below.

$$\mathcal{L}(\mu,\sigma,\pi) = -\sum_n log(\sum_k exp(log(\pi^k) + log(P(\mathbf{X}_n, z = k))))$$

The final equation above is what is implemented as loss_function() below in Figure 16.

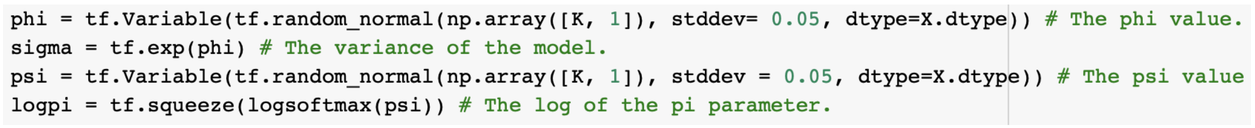


```
# Loss function of the GMM:

def loss_function(log_PDF, log_pi):
    loss = -tf.reduce_sum(reduce_logsumexp(log_PDF + log_pi, 1, keep_dims=True), axis =0)
    return loss
```

Figure 16: loss_function() Python Tensorflow Implementation

Along with the loss_function() implementation shown above, constant definitions in Python Tensorflow are shown below in Figure 17.



```
phi = tf.Variable(tf.random_normal(np.array([K, 1]), stddev= 0.05, dtype=X.dtype)) # The phi value.
sigma = tf.exp(phi) # The variance of the model.
psi = tf.Variable(tf.random_normal(np.array([K, 1]), stddev = 0.05, dtype=X.dtype)) # The psi value
logpi = tf.squeeze(logsoftmax(psi)) # The log of the pi parameter.
```

Figure 17: Constant Allocations in Python Tensorflow for MoG Training

We can now train our network as we have an implemented loss function. We use K = 3 clusters, and train the network on data2D.npy. The results of the simulation are shown below, with a plot of the loss function in Figure 18, and a list of the cluster centres with their corresponding cluster data percentages in Table 5.

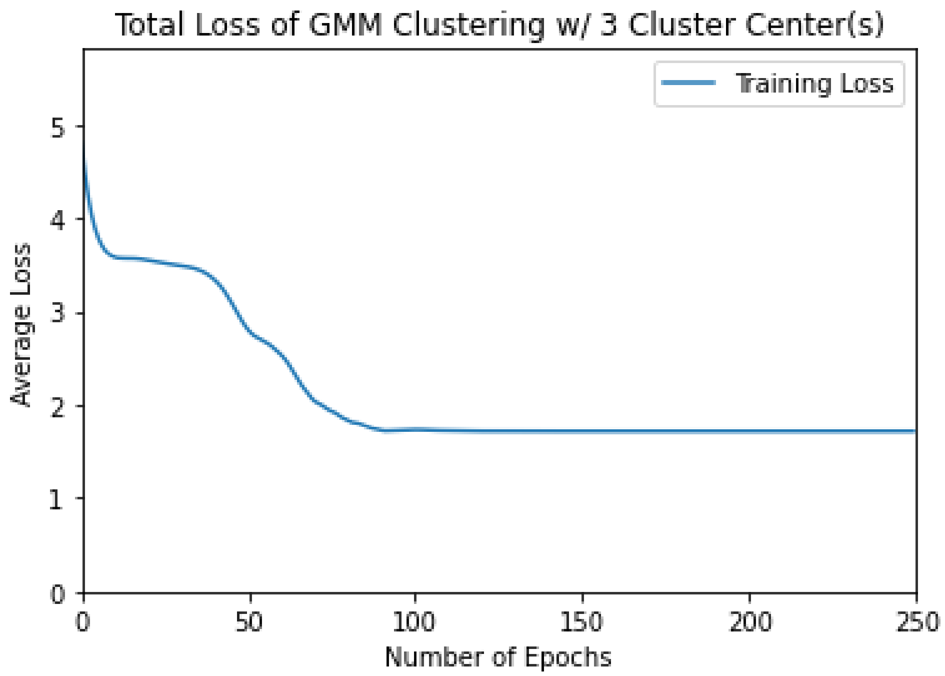


Figure 18: Training Loss over 250 Epochs for MoG Training on data2D.npy

Cluster #	Cluster Centre (x,y)	Phi Values	Psi Values	Percentage of Data in Cluster
Cluster 1	(-1.10315622,-3.30615770)	-3.24233087	-0.23739255	33.84%
Cluster 2	(0.10597514, -1.52734578)	-0.01283568	-0.22916092	32.27%
Cluster 3	(1.29600009, 0.30918072)	-3.24817470	-0.23282574	33.89%

Table 5: Model Parameters for MoG Training with K=3 Clusters on data2D.npy

3.2.2 Training the MoG with a Validation Set

Now that we have verified the functionality of this training method, we will now attempt to check how well it is able to predict the clustering of unforeseen data points. We will hold out a third of the data points in data2D.npy for validation, and we will run trials for K=1 to 5 cluster centers. The final losses in this training and validation process are listed in Table 6, and the percentages of data points within each cluster for each cluster number trial are listed in Table 7. The output graphs of each trial are shown in Figures 19 through 23.

	Total Training Loss	Average Training Loss	Total Valid Loss	Average Valid Loss
K=1 Clusters	23265.979	3.490	11651.443	3.496
K=2 Clusters	16155.743	2.423	7987.797	2.397
K=3 Clusters	11505.944	1.726	5629.354	1.689
K=4 Clusters	11504.034	1.726	5630.213	1.689
K=5 Clusters	11508.023	1.726	5629.824	1.689

Table 6: Training and Validation Losses for K=1 to 5 Clusters in GMM Training w/ Validation.

	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
K=1 Clusters	1	0	0	0	0
K=2 Clusters	0.656	0.344	0	0	0
K=3 Clusters	0.328	0.331	0.341	0	0
K=4 Clusters	0.331	0.329	0.340	0	0
K=5 Clusters	0.332	0.323	0.341	0.003	0.0004

Table 7: Percentage of Data Points which fall into each Cluster for K=1 to 5 cluster trials for GMM.

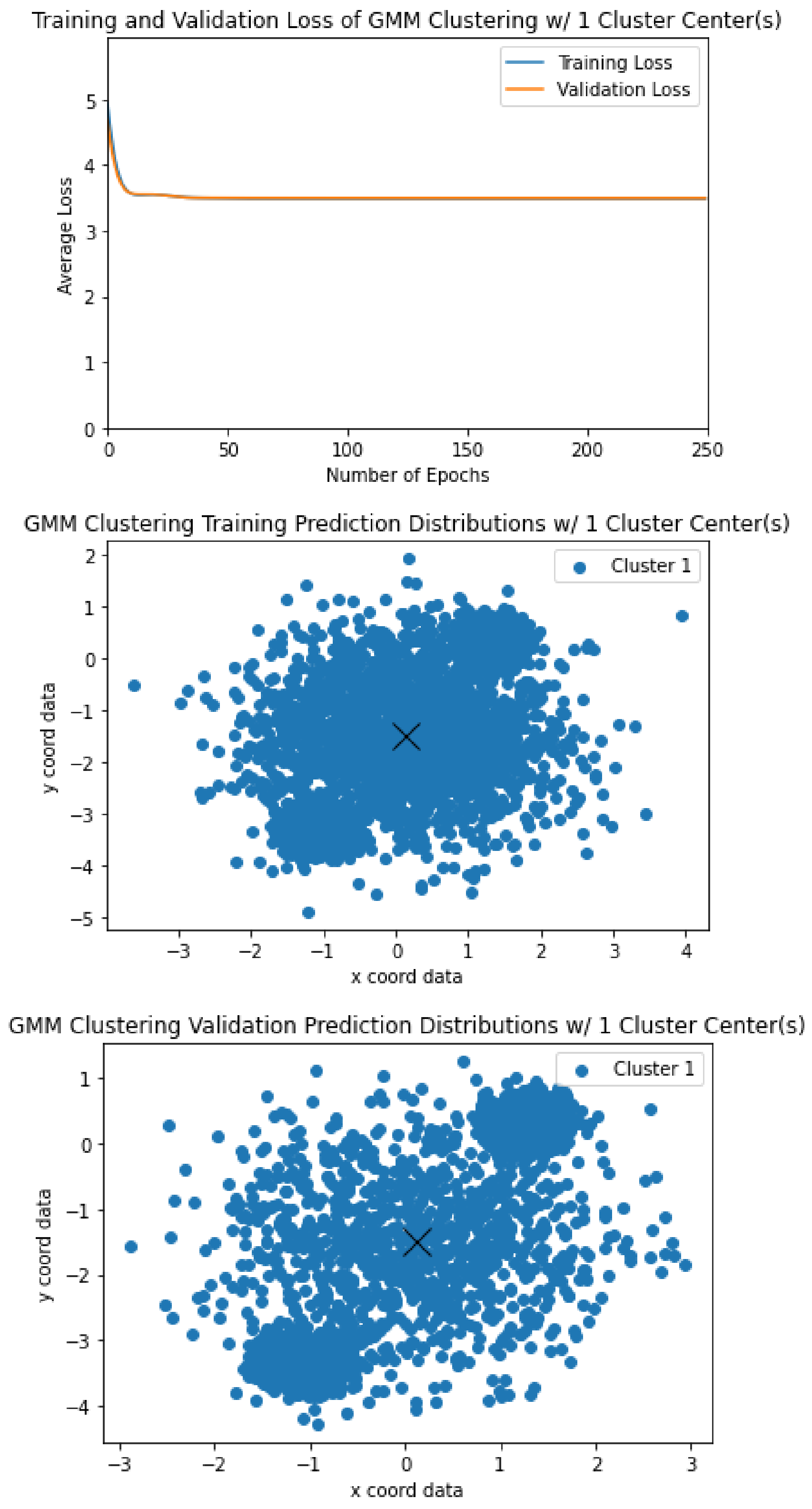


Figure 19: Results from MoG Training and Validation on K=1 Cluster

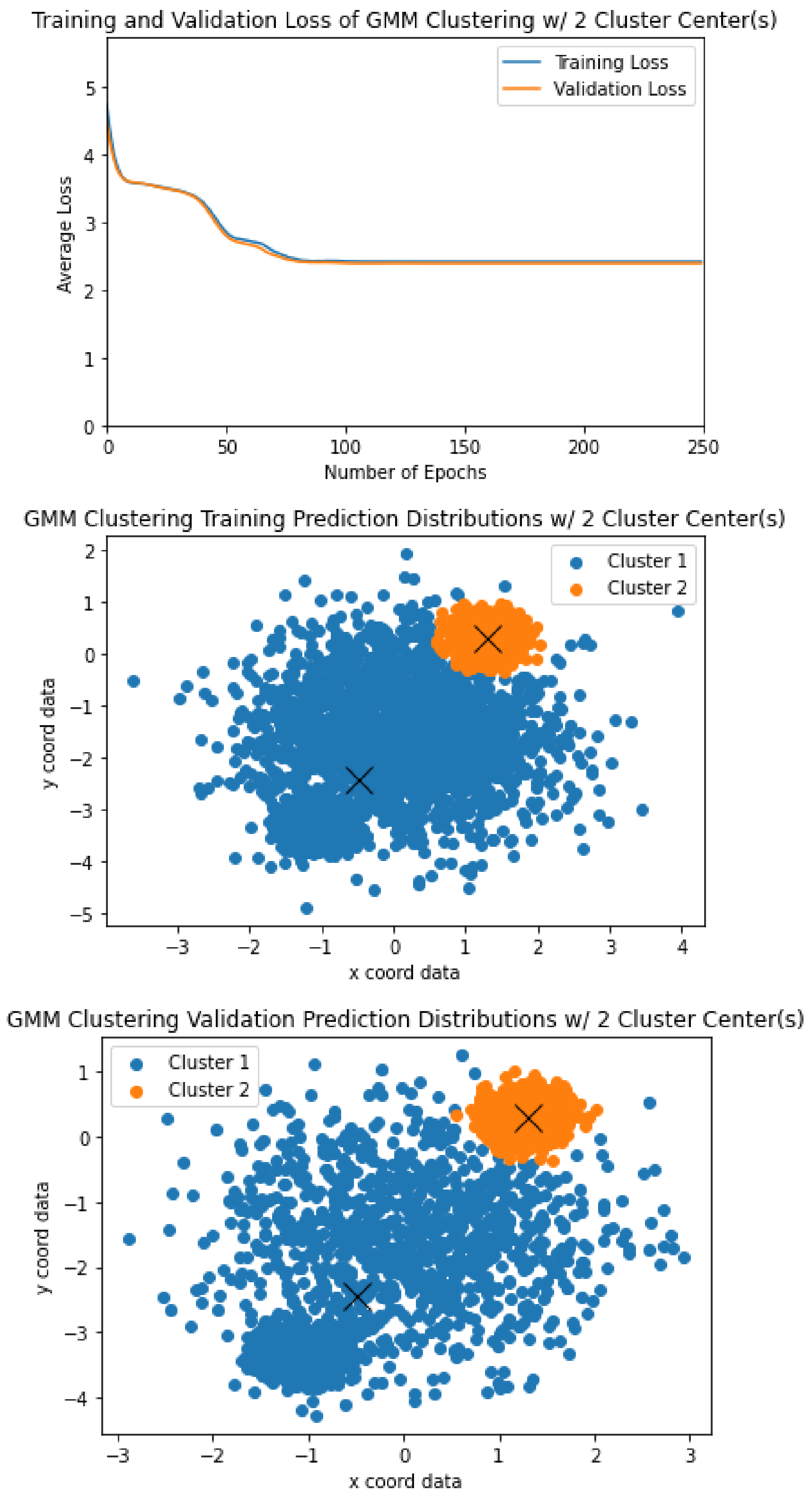


Figure 20: Results from MoG Training and Validation on K=2 Clusters

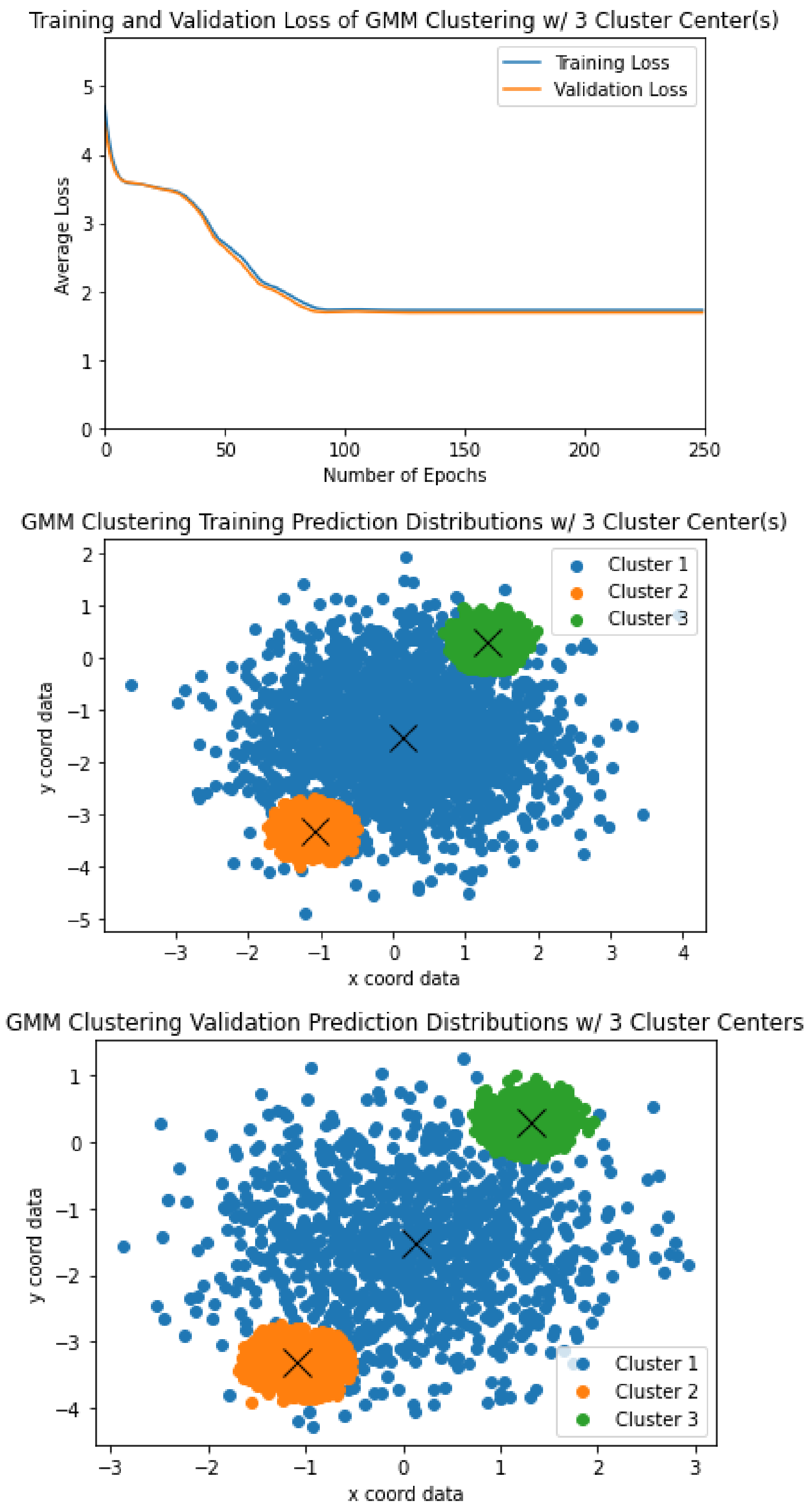
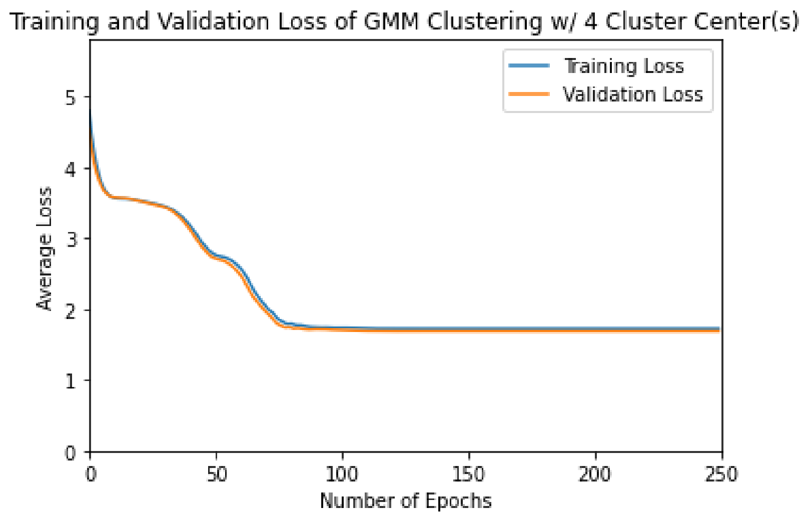
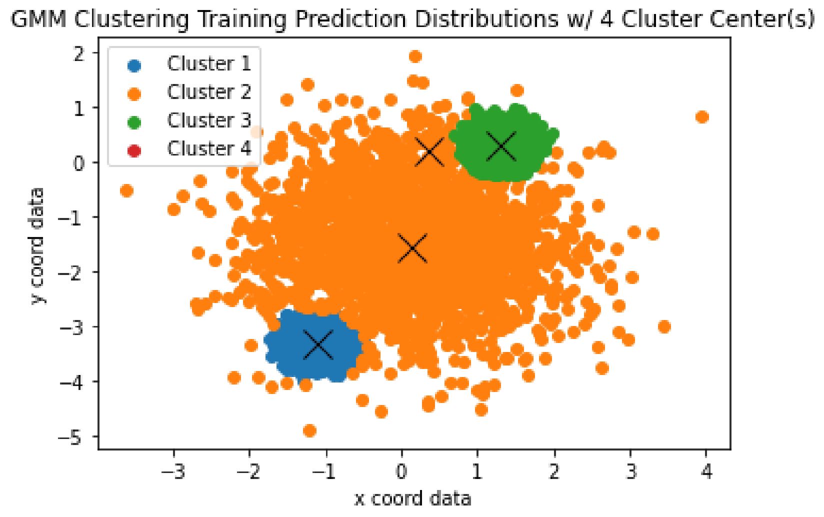


Figure 21: Results from MoG Training and Validation on K=3 Clusters





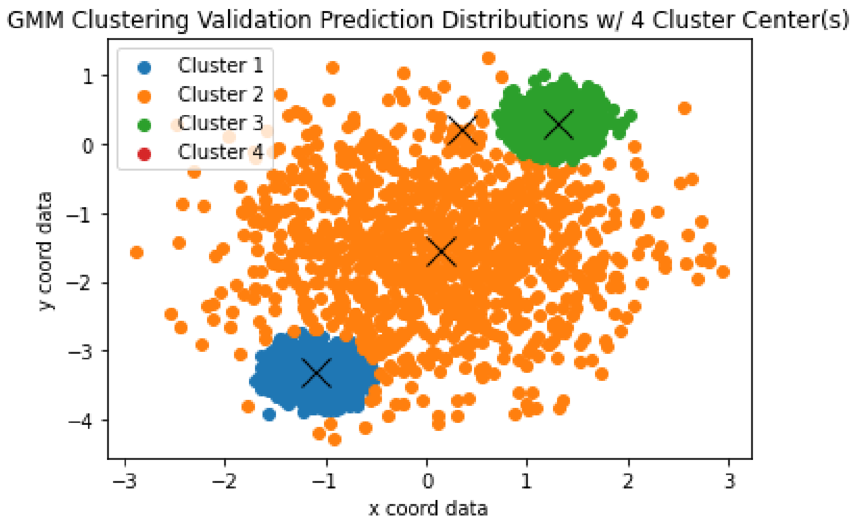


Figure 22: Results from MoG Training and Validation on K=4 Clusters

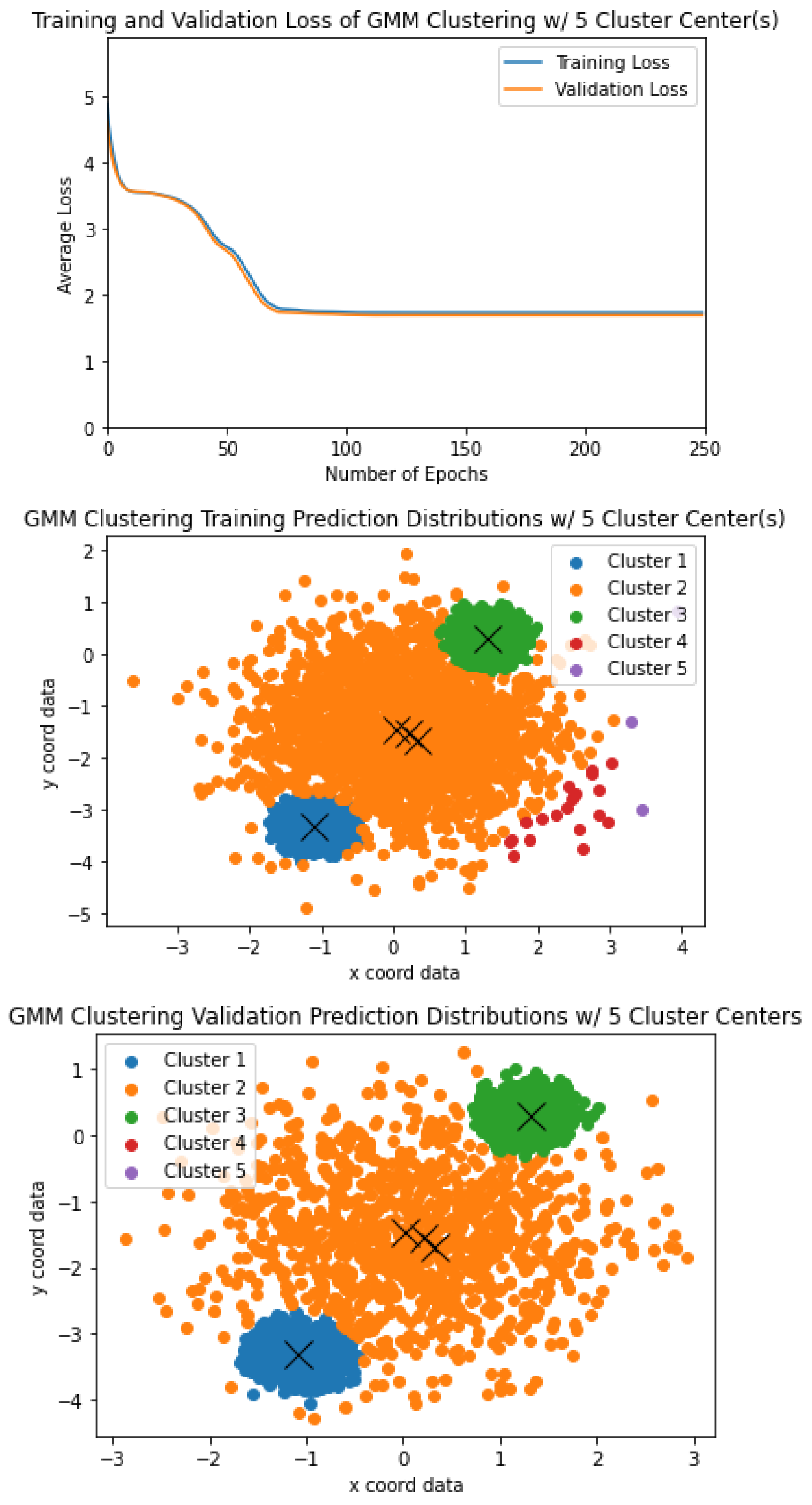


Figure 23: Results from MoG Training and Validation on K=5 Clusters

Contrary to the K-Means training, it is obvious which of the values of K is best for data2D.npy. If we simply look at the placement of cluster centers on the Training and Validation Prediction Distribution graphs in Figures 19 to 23, it is clear to see that K=3 is optimal. In the K=4 case, we see a redundant cluster in Cluster 4, where it literally contains 0% of the data points from data2D.npy as is seen in Table 7. Then if we look at the K=5 trial, we see that there are three cluster centers placed at the middle of Cluster 2, with both Clusters 4 and Cluster 5 containing less than 1% of data, as is shown in Table 7. Then, as the K=3 trial and all above show three clusters of equal mass, the K=1 and K=2 cases are clearly under fitting the data. For these reasons, it is clear to say that K=3 clusters best fits the data in data2D.npy.

3.2.3 Comparing K-Means and MoG Training

Up to this point, all training and validation for the K-Means and MoG training algorithms has been applied to the dataset contained within data2D.npy. Now, we will compare the two methods on a much larger dataset that is contained within data100D.npy. Training and Validation Losses for K-Means and MoG Training are shown in Table 8 and Table 9 respectively. Output graphs for K-Means and MoG Training are shown in Figures 24 through 28 and Figures 29 through 33 respectively.

	Total Training Loss	**Average Training Loss**	**Total Valid Loss**	**Average Valid Loss**
K=5 Clusters	245752.366	36.861	123745.901	37.127
K=10 Clusters	143460.894	21.518	71792.084	21.540
K=15 Clusters	140117.146	21.017	70350.841	21.107
K=20 Clusters	136762.602	20.513	69078.314	20.726
K=30 Clusters	137459.080	20.618	69733.959	20.922

Table 8: Training and Validation Losses for K=5 to 30 Clusters in K-Means Training at Dim = 100.

	Total Training Loss	**Average Training Loss**	**Total Valid Loss**	**Average Valid Loss**
K=5 Clusters	542426.443	81.360	271826.722	81.556
K=10 Clusters	321143.251	48.169	162156.895	48.652
K=15 Clusters	319167.942	47.873	161615.670	48.489
K=20 Clusters	318907.124	47.833	161489.330	48.452

K=30 Clusters	316811.783	47.519	161105.702	48.337

Table 9: Training and Validation Losses for K=5 to 30 Clusters in GMM Training at Dim = 100.

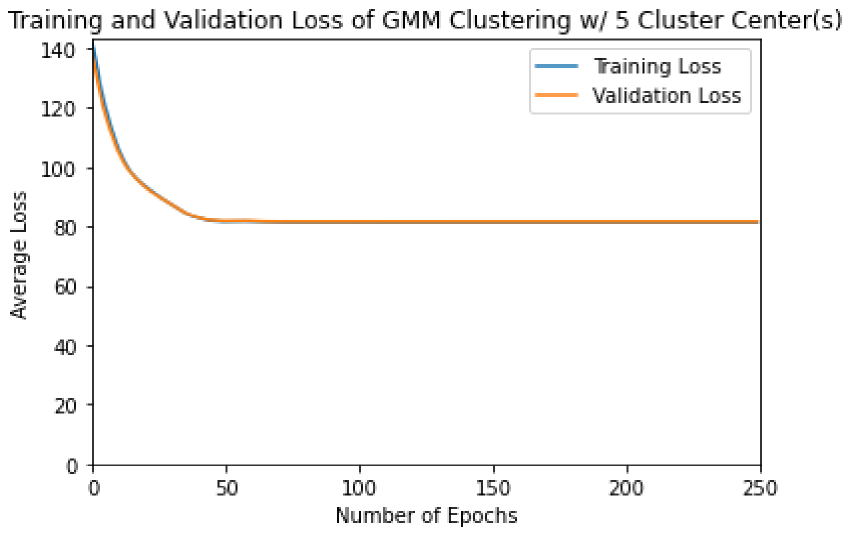


Figure 24: Training and Validation Loss for K-Means, K=5 Clusters

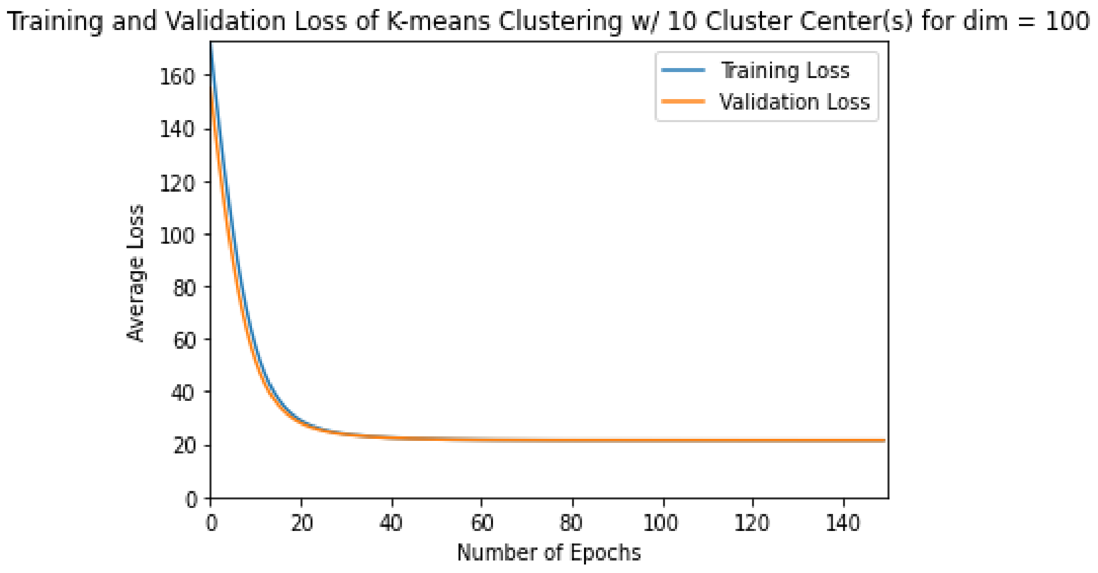


Figure 25: Training and Validation Loss for K-Means, K=10 Clusters

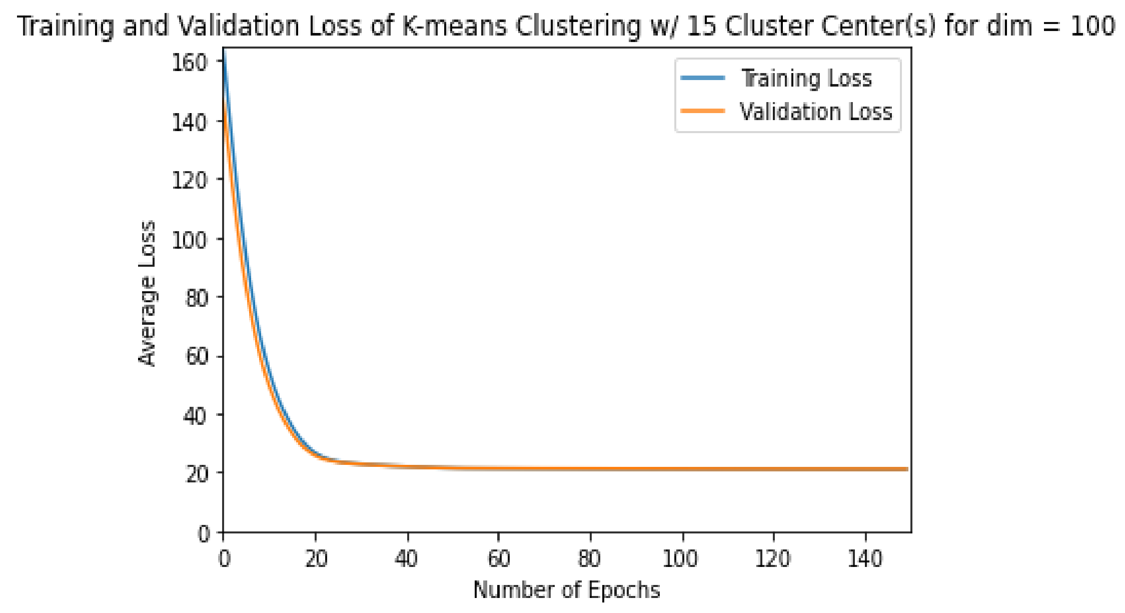


Figure 26: Training and Validation Loss for K-Means, K=15 Clusters

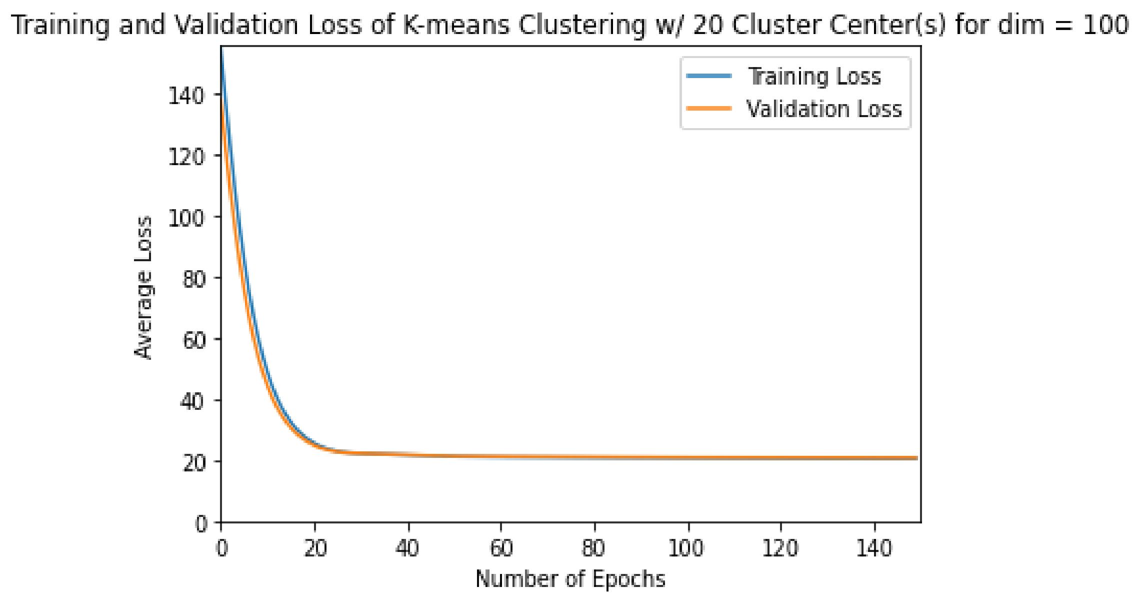


Figure 27: Training and Validation Loss for K-Means, K=20 Clusters

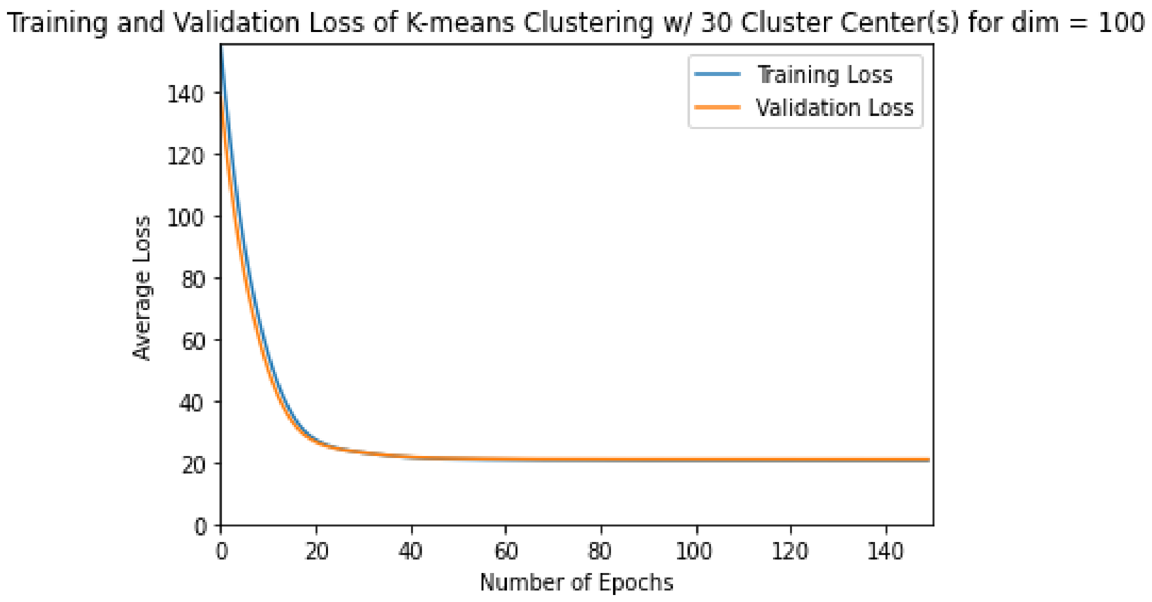


Figure 28: Training and Validation Loss for K-Means, K=30 Clusters

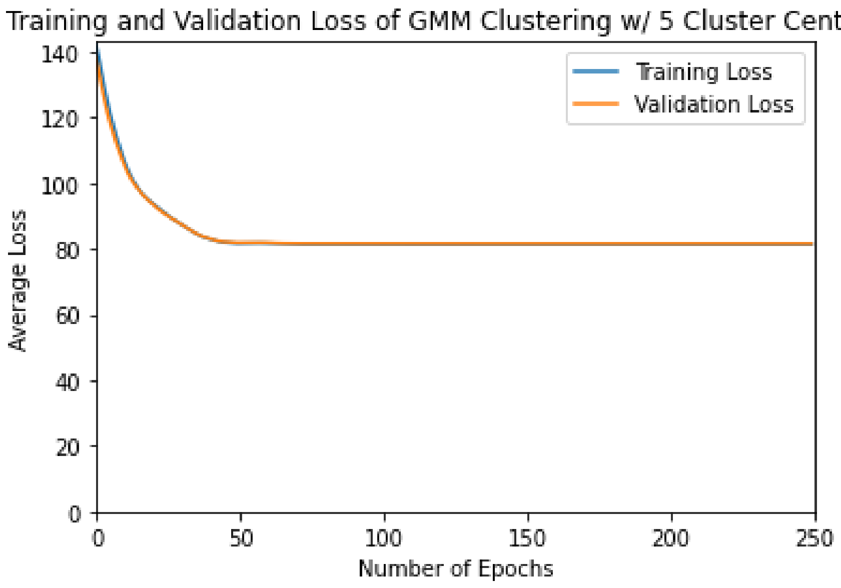


Figure 29: Training and Validation Loss for MoG, K=5 Clusters

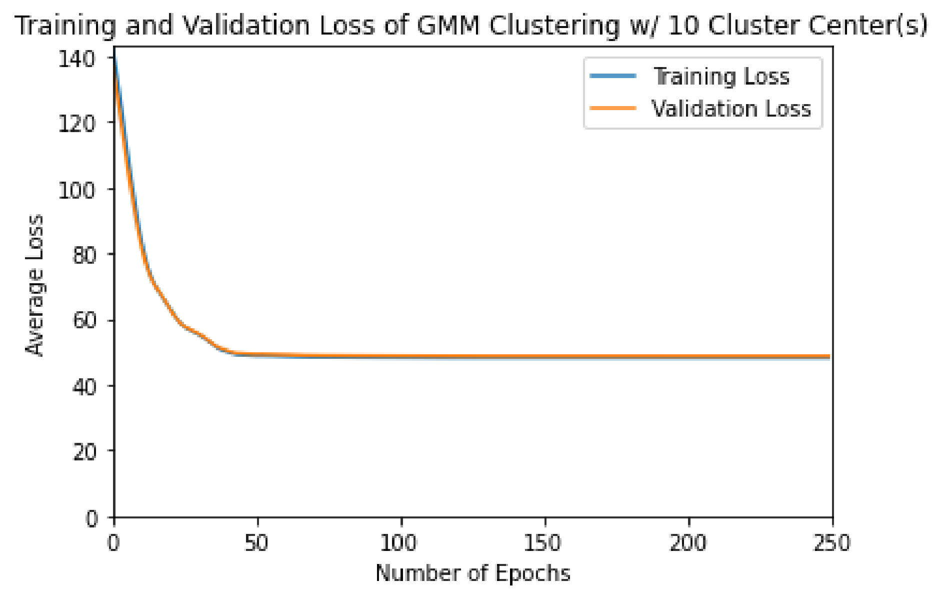


Figure 30: Training and Validation Loss for MoG, K=10 Clusters

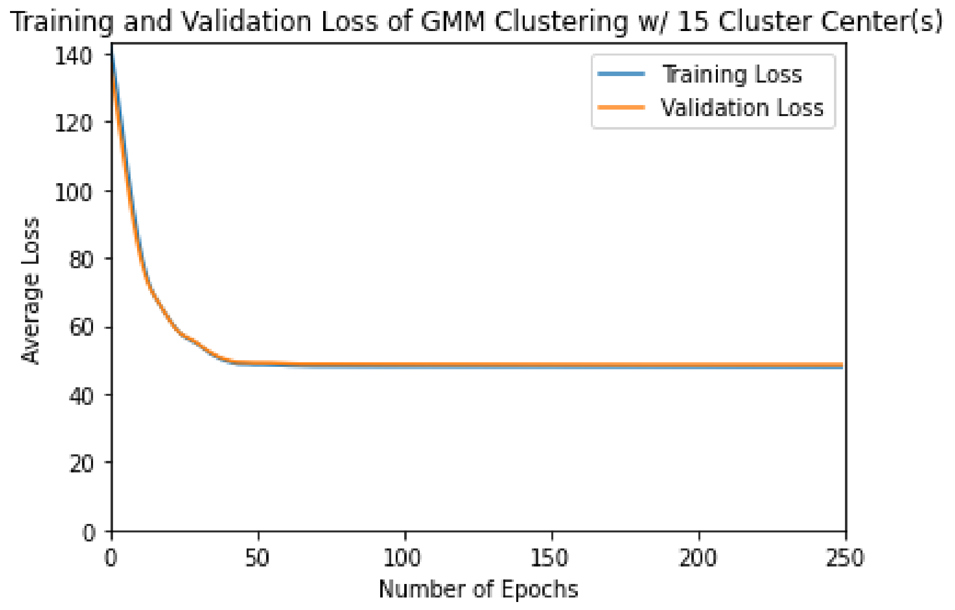


Figure 31: Training and Validation Loss for MoG, K=15 Clusters

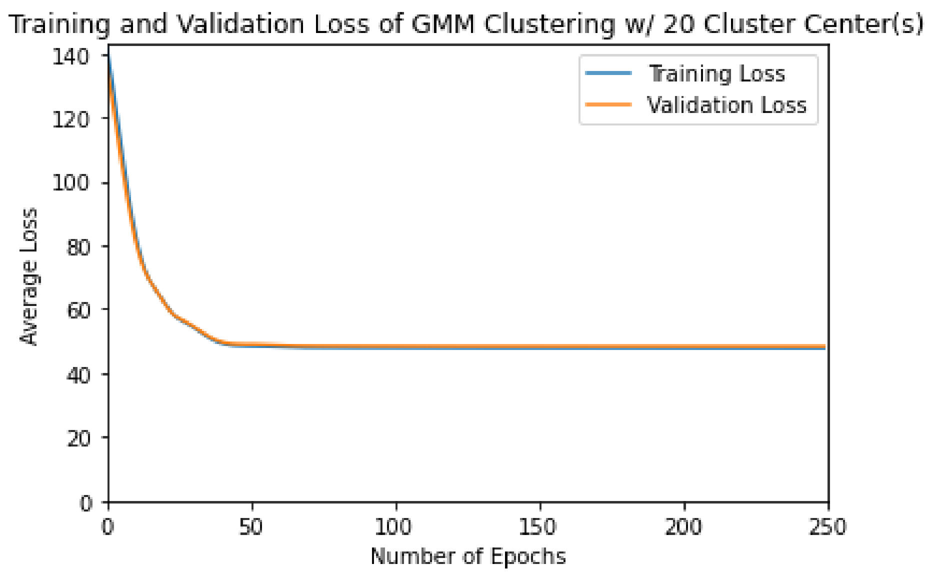


Figure 32: Training and Validation Loss for MoG, K=20 Clusters

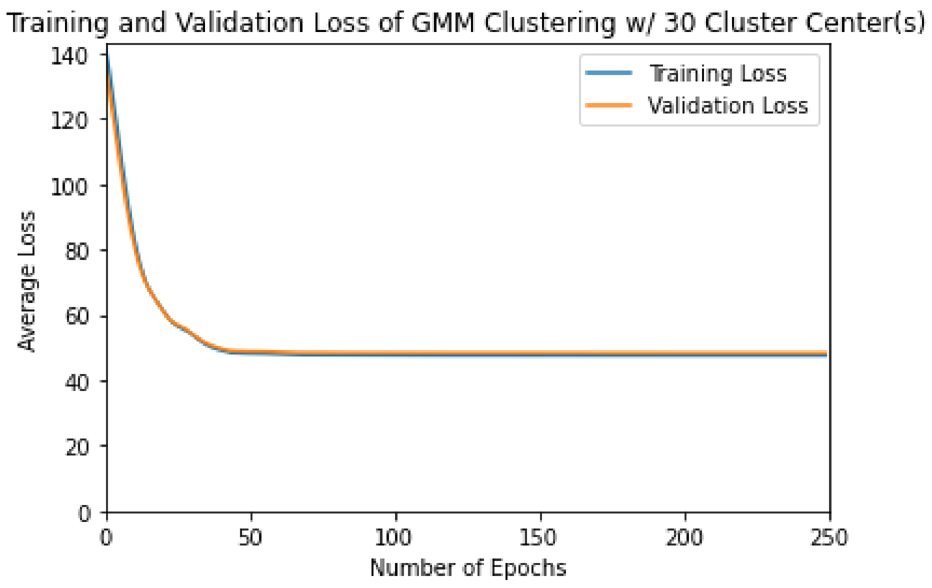


Figure 33: Training and Validation Loss for MoG, K=30 Clusters

It can clearly be seen by comparing Average and Total Validation Losses in Tables 8 and 9 that the Validation Loss stops decreasing when we reach K=10 clusters. This is also visible as we examine trends in final losses as we look down Figures 24-28 and Figures 29-33. This implies that there are about 10 clusters within the dataset data100D.npy.

Although we still see that losses are decreasing as we increase K, this is due to the fact that losses must decrease as K increases as long as the added cluster centers are contained within twice the maximum range of the dataset due to the Euclidean distance implementation of the loss function for both MoG and K-Means training.

However, we do notice that K-Means has much lower training and validation losses than MoG training for all trials. This is by over a factor of two, and demonstrates an interesting manner in which the two algorithms are different.

The K-Means algorithm is more effective at decreasing losses, but MoG is better at finding representative clusters. The lower losses from the K-Means algorithm are much simpler to see – just look at the entries of Tables 8 and 9. This trend occurs because K-Means attempts to solely decrease total Euclidean Distance from data points to cluster centers. This allows the algorithm to divide large clusters into smaller components. This cluster division is visible in the Prediction Distributions in Figures 12 and 13, where the large cluster in the center of the graph is divided into two and three smaller sections. We can contrast this cluster division to the output of the MoG algorithm in Figures 22 and 23 where the large cluster in the middle is not divided into smaller components.

While this consistency in MoG may be more representative of the dataset (so it will likely be better at classification than K-Means), its inability to subdivide large clusters will restrict its ability to decrease the loss compared to K-Means. This introduces a tradeoff between the two algorithms, and as such, allows both to be more effective than the other in different applications.

4. Conclusion

In this report, we examined the use of the K-Means Learning Algorithm, as well as a MoG Leaning Model on two datasets. First, we implemented K-Means in Python Tensorflow, which allowed us to train on the data2D.npy dataset with and without a validation subset. We varied the number of clusters that K-Means would use in the training process, and determined that four clusters would likely be more representative of the dataset. We then implemented a MoG Learning model in Python Tensorflow by extending our K-Means model. This allowed us to train the MoG model on data2D.npy with and without a validation subset. It was found that K=3 clusters is optimal for this dataset, contrary to the conclusion we made from the information given from K-Means. Finally, we compared the two algorithms directly by training them on a second dataset, data100D.npy. We discussed the trends seen, and concluded that while K-Means is better and minimizing losses, the MoG model was better at finding representative clusters. This meant that they could both be effective in different scenarios.

5. References

[1] Assignment 3: Unsupervised Learning and Probabilistic Models, ECE421.