Assignment 3

Wednesday, April 3rd, 2019 | ECE1513
University of Toronto | Introduction to Machine Learning

1 Unsupervised Learning Using K-Means

1.1 Learning K-Means

The first data set that will be used to create a K-means model has $D = 2$ input features, and will be fitted with $K = 1, 2, 3, 4, 5$ cluster centers. The center of these K clusters, denoted by the cluster center tensor $\boldsymbol{\mu}$ where the kth row of the tensor denotes the kth cluster center $\boldsymbol{\mu}_k$, will be used in the loss function in order to optimize their location. The loss function is given by:

$$\boldsymbol{\mathcal{L}}(\boldsymbol{\mu}) = \sum_{n=1}^{N} \min_{k=1}^{K} ||\boldsymbol{x}_n - \boldsymbol{\mu}_k||_2^2 \tag{1}$$

The $D = 2$ data set has $N = 10000$ data points. Initially, we will use all of these as training samples, however in section 1.1.3 one third of this data set will be withheld to be used as validation points.

The Tensorflow graph is built as follows. After initializing centroid locations, and defining the distance squared function, the output of the distance squared function is input into the loss function which will be minimized using the Adam optimizer. After the optimization is completed, the data points in the data set are assigned to their nearest centroid, and the data is plotted. Code snippets for the build graph function and the optimization step are presented below, along with the distance squared function which is input into the loss.

```python
def distanceFunc(X, mu): # Returns distance squared
    expandPoints = tf.expand_dims(X, 0)
    expandCentroid = tf.expand_dims(mu, 1)
    return tf.reduce_sum(tf.square(tf.subtract(expandPoints, expandCentroid)), 2)
```

Figure 1: Code snippet of the distance squared function used to calculate the loss per epoch.

```python
def buildGraph():
    tf.reset_default_graph() # Clear any previous junk
    tf.set_random_seed(45689)

    trainingInput = tf.placeholder(tf.float32, shape=(None, dim))  # Data placeholder
    centroid = tf.get_variable('mean', shape=(k,dim), initializer=tf.initializers.random_normal())  # Mean placeholder
    distanceSquared = distanceFunc(trainingInput,centroid) # Finds the euclidean norm
    loss = tf.math.reduce_sum(tf.math.reduce_min(distanceSquared,0)) # Choose the smallest distance for each point and sum them
    optimizer =tf.train.AdamOptimizer(learning_rate= 0.01, beta1=0.9, beta2=0.99,epsilon=1e-5).minimize(loss) # Optimize
    return optimizer, loss,  distanceSquared, centroid, trainingInput
```

Figure 2: Code snippet of the Build Graph function used to create the graph to be optimized by the Adam optimizer in Tensorflow.

```python
def kMeans(valid=False):
    loadData(valid) # Load the Data
    optimizer, loss,  distanceSquared, centroid, trainingInput = buildGraph() # Build the graph
    init = tf.global_variables_initializer()
    with tf.Session() as sess:
        sess.run(init)
        for i in range(0,epochs):
            _, trainLoss[i], dist, mU = sess.run([optimizer, loss,  distanceSquared, centroid], feed_dict = {trainingInput:trainData})

            if valid: # Find validation loss and distance if this is True
                validLoss[i],distV = sess.run([loss, distanceSquared], feed_dict = {trainingInput: validData})

        assign = np.argmin(dist,0) # Assign the point to the nearest cluster
        inCluster = np.mean(np.eye(k)[assign],0)  # Find the average number of points in each cluster
        plotter(valid) # Plot the Loss vs Epochs graph
        scatter(trainData, assign, mU) # Draw a 2D scatter plot. For dim = 2 only

        if valid:
            assignV = np.argmin(distV,0) # If we are validating, use the distances of the validation points to assign the cluster
            inClusterV = np.mean(np.eye(k)[assignV],0) # Find the average number of points in each cluster
            scatter(validData, assignV, mU) # Draw a 2D scatter plot for the validation points. For dim = 2 only
            return mU, inCluster, inClusterV
    return mU, inCluster, None
```

Figure 3: Code snippet of the optimization function which determines the optimized location for each centroid, and determines which centroid each data point belongs to.

1.1.1 Optimization Using Adam Optimizer

The loss function given in equation 1 will be minimized using a gradient descent algorithm that employs the Adam optimizer. The initial cluster center values will be selected by sampling a standard normal distribution, and the optimizer will use the following hyperparameters: $\beta 1 = 0.9$, $\beta 2 = 0.99$, $\epsilon = 10^{-5}$.

The loss value per epoch of the optimization is plotted in figure 4 below. A learning rate of $\alpha = 0.01$ was used during this optimization.

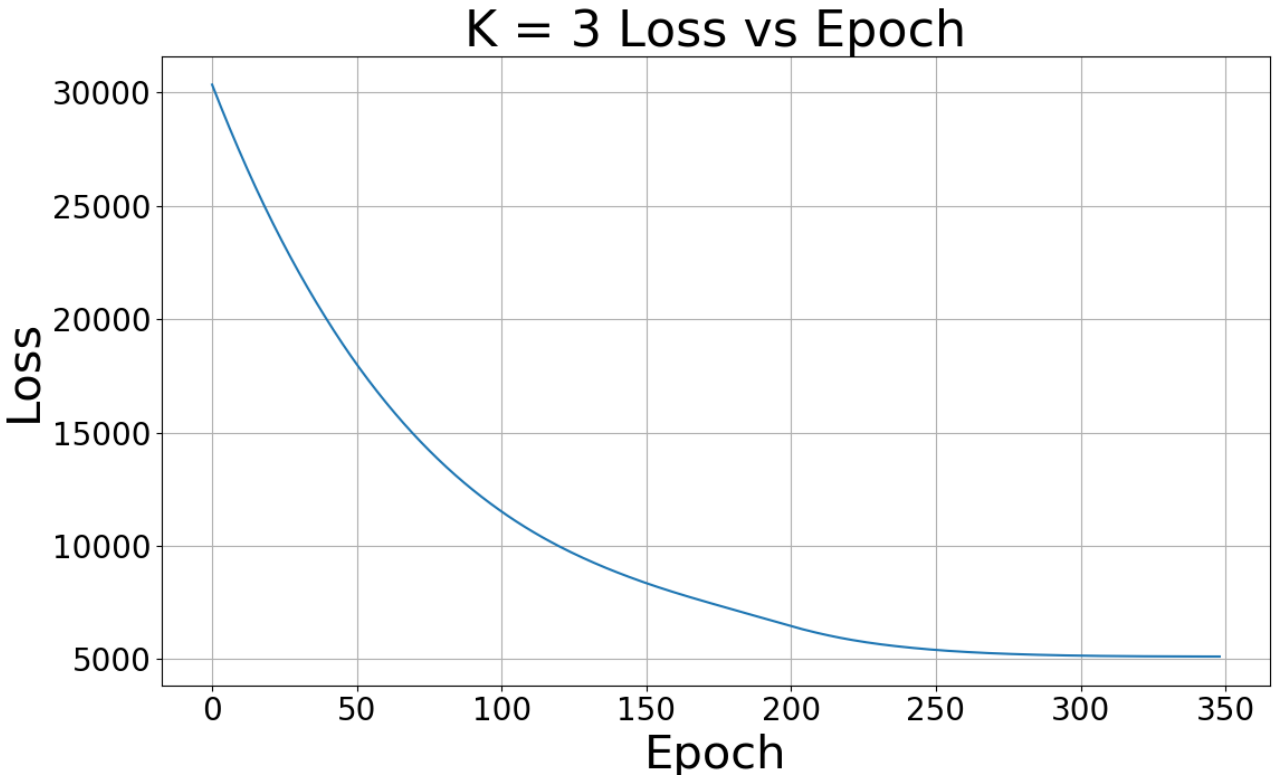

Figure 4: Loss versus epoch for k-means clustering optimization on the 2 dimensional dataset using 3 clusters.

1.1.2 Optimizing Number of Clusters

The loss function given above in equation 1 is optimized again using 1, 2, 4 and 5 clusters. Table 1 below details the percentage of the data points belonging to each cluster.

Number of Clusters	% Data	% Data	% Data	% Data	% Data
K = 1	100				
K = 2	50.5	49.5			
K = 3	38.2	23.8	38.0		
K = 4	37.1	12.1	37.3	13.5	
K = 5	36.8	11.1	37.0	7.6	7.5

Table 1: Percentage of data points belonging to each cluster for K $= 1, 2, 3, 4, 5$.

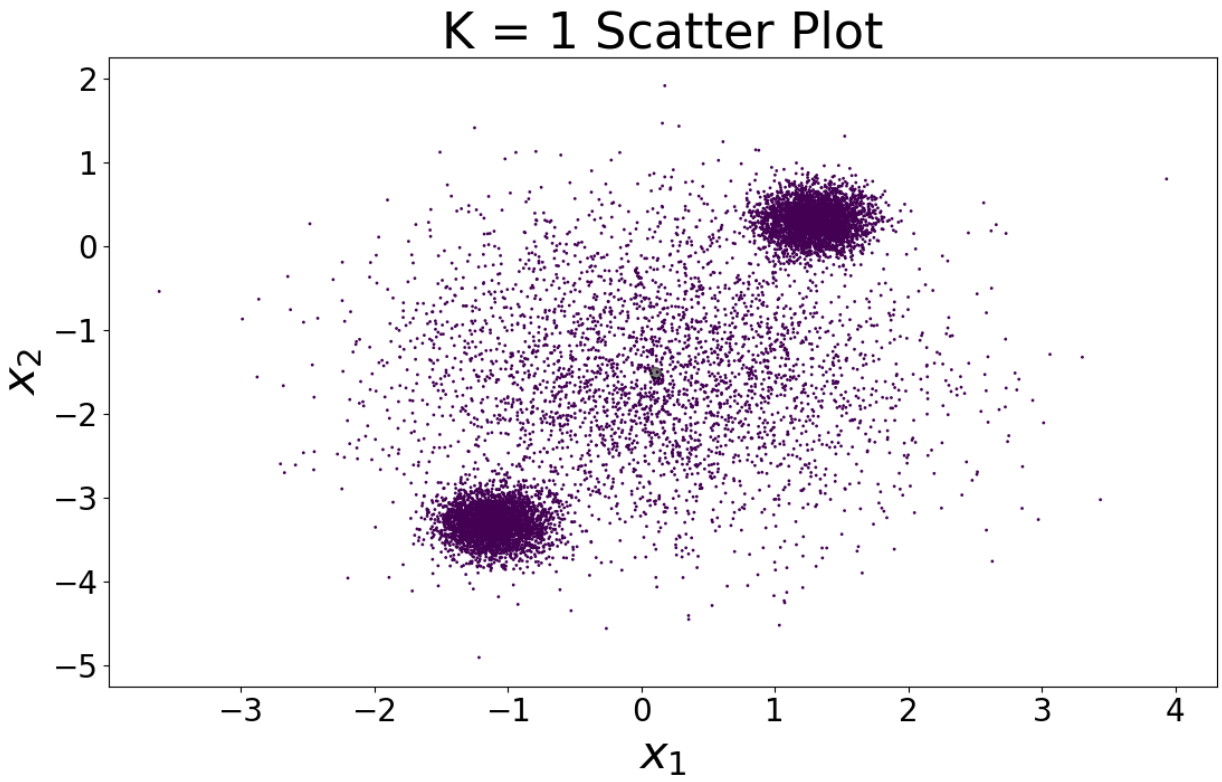

Figure 5: 2-D scatter plot of $D = 2$ data set for $K = 1$. Data points are colour labeled according to which cluster they belong to.

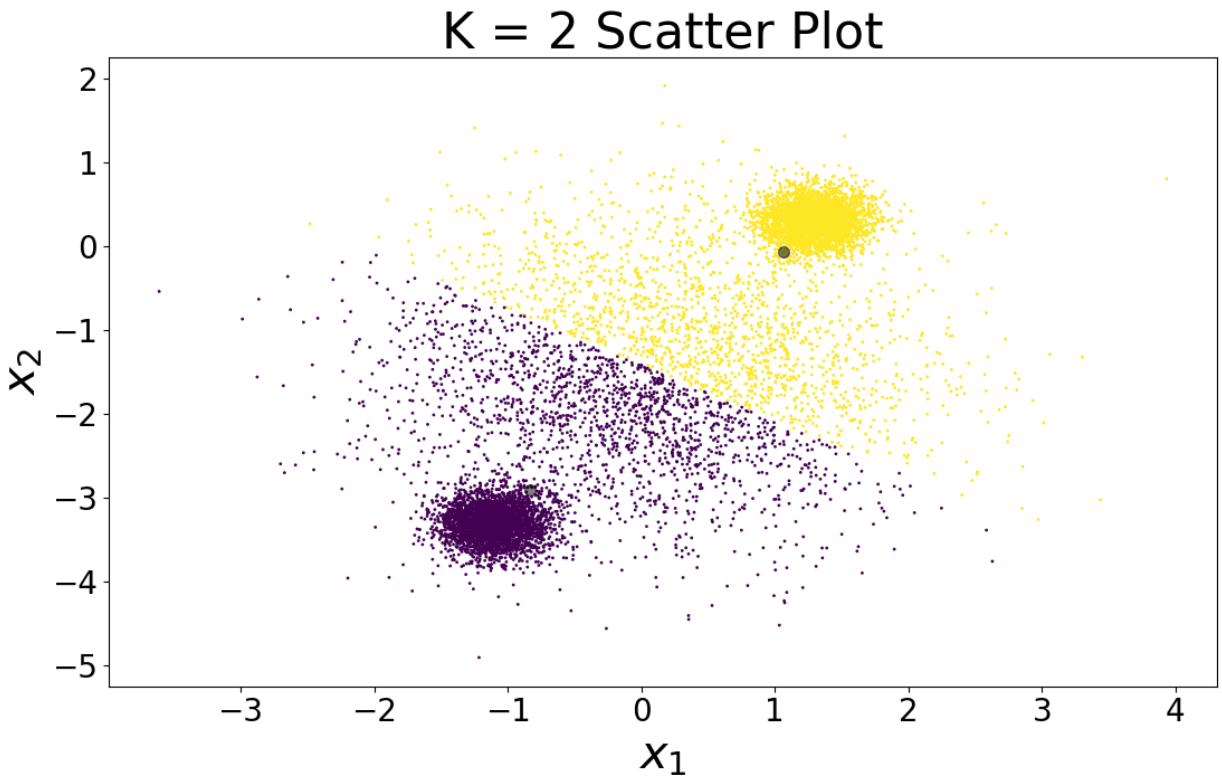

Figure 6: 2-D scatter plot of $D = 2$ data set for $K = 2$. Data points are colour labeled according to which cluster they belong to.

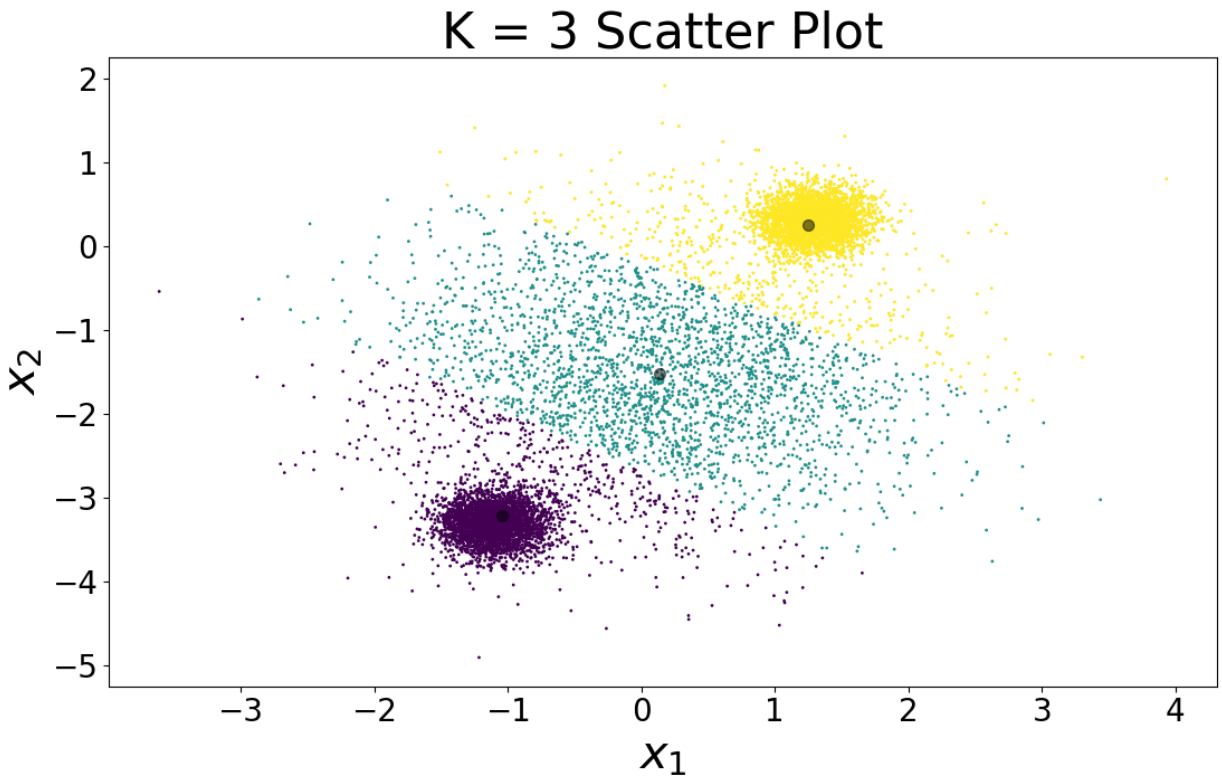

Figure 7: 2-D scatter plot of $D = 2$ data set for $K = 3$. Data points are colour labeled according to which cluster they belong to.

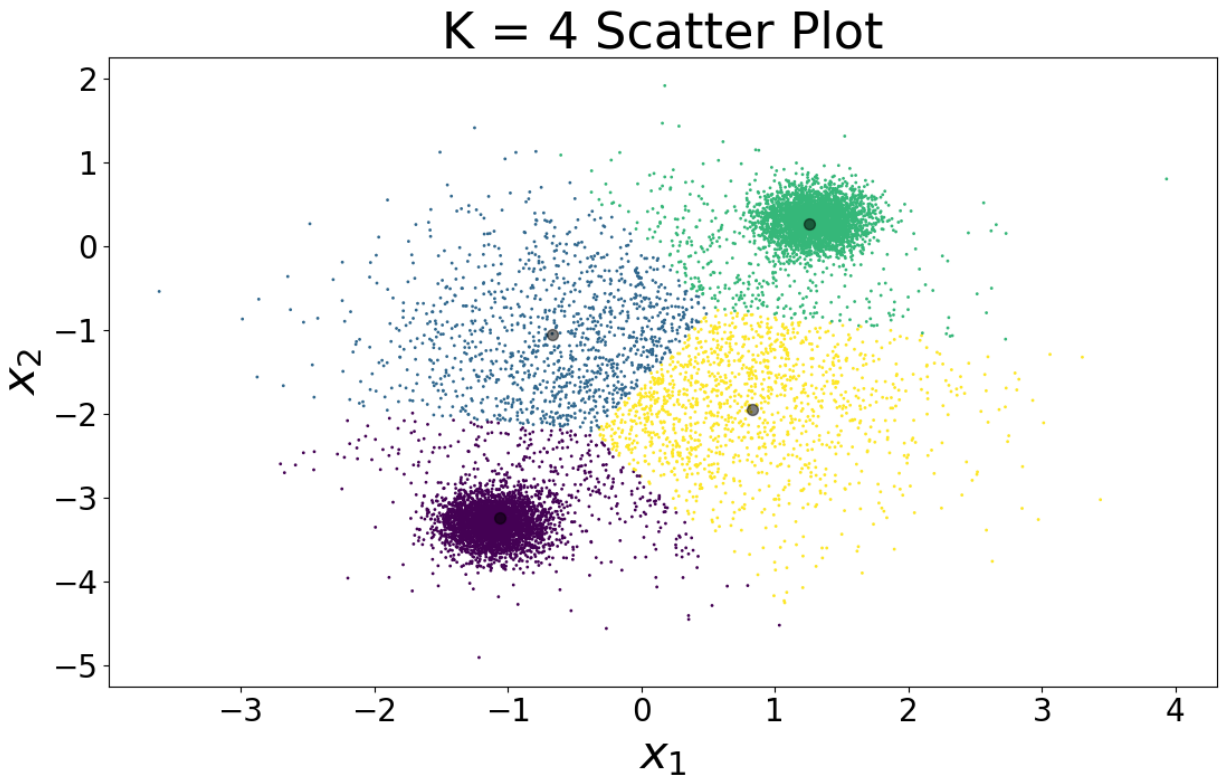

Figure 8: 2-D scatter plot of $D = 2$ data set for $K = 4$. Data points are colour labeled according to which cluster they belong to.

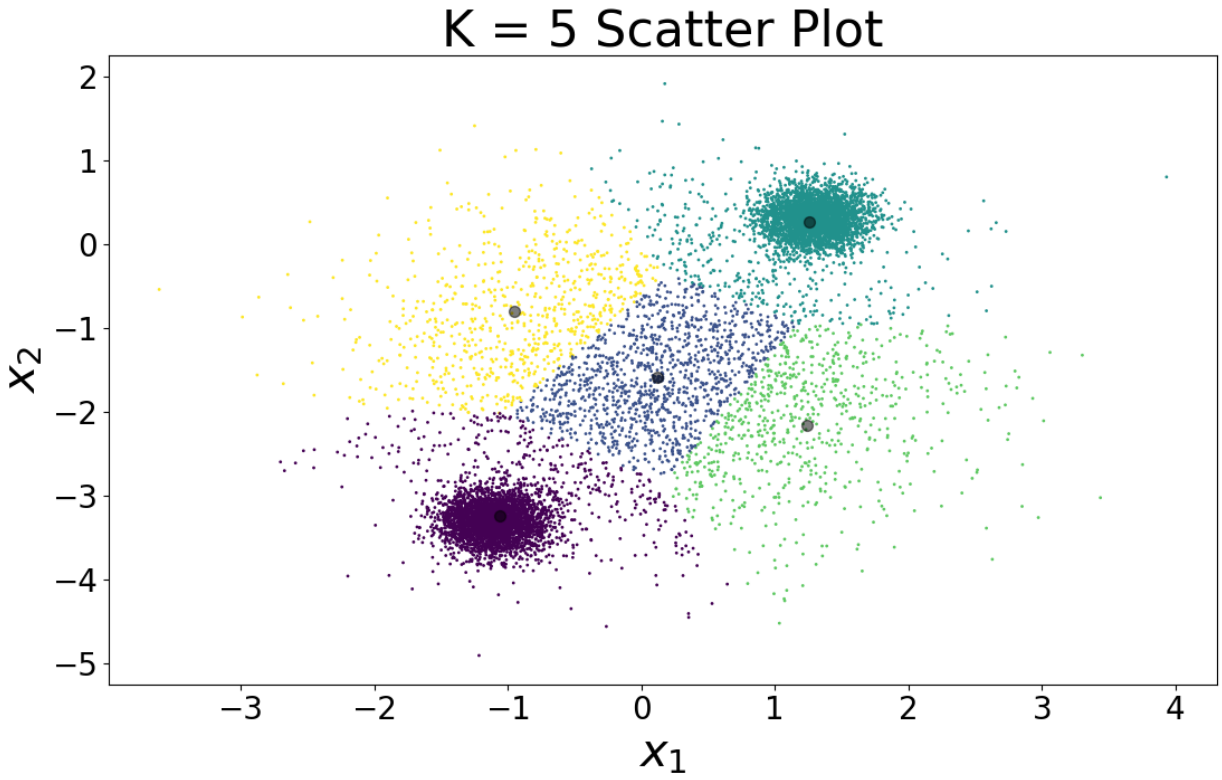

Figure 9: 2-D scatter plot of $D = 2$ data set for $K = 5$. Data points are colour labeled according to which cluster they belong to.

In terms of an optimal number of clusters, it was decided that 3 was the optimal number for the $D = 2$ data set. When looking at the plots of which data points belong to which cluster, it is clear that there are 2 distinct clusters that have data points much closer together. These two cluster centers are separated by a less dense group of data points that do not obviously belong to either group. When the value of K is increased to 4, this middle group is almost arbitrarily divided down the middle without any obvious reason why. Similarly, when K is increased to 5, this central group is split into thirds. Since the boundaries between the central group for $K = 4$ and $K = 5$ do not appear to make any distinctions between the middle group, they appear to be over fitting the data, and thus, 3 was the optimal number of clusters for this data set.

1.1.3 Implementing Training and Validation Segmentation

The same optimization as section 1.1.2 is performed, varying the same number of clusters K $= \{1, 2, 3, 4, 5\}$, however now one third of the training data is held out as a validation set. Table 2 below details the loss on the validation set for each optimization.

Number of Clusters	Validation Loss
K $= 1$	12,870.1
K $= 2$	2,960.67
K $= 3$	1,629.21
K $= 4$	1,054.54
K $= 5$	907.21

Table 2: Loss of the K-means model on the validation set for optimizations where K $= 1, 2, 3, 4, 5$.

Based on the loss values on the validation set, it appears that $K = 3$ or $K = 4$ is the optimal number of clusters for this data set. When increasing the number of clusters from 1 to 2, a decrease in loss of approximately 80% is observed. When the number of clusters is again increased from 2 to 3, a reduction in loss of approximately 40% is observed. Finally, when the number of clusters is increased from 3 to 4, a reduction in loss of approximately 30% is observed. Increasing the number of clusters from 4 to 5 only decreases the loss by approximately 10%, and thus does not seem to be deepening the model's understanding.

Clearly, a reduction in loss of 80% is desirable, so K $= 1$ is clearly not an optimal number of clusters. Similarly, when we increase the cluster number from 2 to 3, we observe nearly a 50% reduction in loss, so 2 is clearly not an optimal number. The decrease in loss obtained by increasing the number of clusters to 5 is just above 10%, which is not as significant a jump. Therefore, based on changes in loss value, it was determined that 3 or 4 was the optimal number of clusters. When combining this information with the figures from section 1.1.2, it was decided that 3 was the optimal number of clusters.

2 Unsupervised Learning Using Gaussian Mixture Model

2.1 The Gaussian Mixture Model

The same $D = 2$ data set is optimized using a Gaussian Mixture Model. In this model, each data point has a probability that it is associated with one of the Gaussian distributions, with each of the K mixture components occuring with probability $\pi^k = \mathrm{P}(z = k)$. The center of each cluster is still denoted by the cluster mean tensor $\boldsymbol{\mu}$, with the center of the k$^{\text{th}}$ cluster denoted by $\boldsymbol{\mu}^k$. For simplicity, it is assumed that the different input features are all independent for each cluster k and have the same standard deviation σ^k.

2.1.1 Modifying the Distance Function

Due to the assumption made about the standard deviations, the covariance matrix for each cluster is just a diagonal with the same value for variance along the diagonals. This means we can represent each cluster's covariance as one constant, meaning we only need a k by 1 vector to represent the k-clusters, with each element being the variance of the kth cluster. The determinant for each cluster will be the the k by 1 vector raised to the power of D. Because it's square-rooted, the determinant is to the power of $\frac{D}{2}$, the same as 2π. This let's us write the following:

$$\mathcal{D}^2 = ||\boldsymbol{x}_n - \boldsymbol{\mu}_k||_2^2 \tag{2}$$

$$P(\boldsymbol{X}|\boldsymbol{Z}) = -\frac{1}{2}\frac{\mathcal{D}^2}{\sigma^2} - \frac{D}{2}\log(2\pi\sigma^2) \tag{3}$$

$$\tag{4}$$

The code snippet implementing this function is presented in the code snippets below, where the distance function is the same as in Figure 1.

```
def log_GaussPDF(X, mu, variance):
    distanceSquared = distanceFunc(X,mu)
    return -1.0*tf.divide(tf.transpose(distanceSquared),2*variance) - dim/2*tf.log(2*math.pi*variance)
```

Figure 10: Function calculating the log probability of the cluster variable z, implemented in the Gaussian Mixture Model.

2.1.2 Computing the Posterior

In order to reduce training time, a vectorized Tensorflow implementation is presented below which calculates the log probability of the cluster latent variable z given the data vector $\boldsymbol{x}$. It is computed as $\log P(z|\boldsymbol{x})$. In this implementation, the helper functions given in the assignment are used, specifically $logsumexp$.

```
def log_posterior(log_PDF, log_Pi):
    lagrange = tf.add(log_PDF, log_Pi) # Couldn't think of a name, so we named it after Lagrange
    return tf.subtract(lagrange,hlp.reduce_logsumexp(lagrange,keep_dims=True))
```

Figure 11: Bayesian posterior function, implemented in the Gaussian Mixture Model.

It is important to use the log-sum-exp function in this instance since we cannot add the probabilities in the log domain The log-sum-exp function returns the probabilities from the logarithmic domain to the standard domain, sums them, and then returns them back to the logarithmic domain.

2.2 Learning the Gaussian Mixture Model

The marginal data liklihood for the GMM is given by the following equation:

$$P(\boldsymbol{X}) = \prod_{n=1}^{N} P(\boldsymbol{x}_n) = \prod_{n=1}^{N}\sum_{k=1}^{K} P(z_n = k)P(\boldsymbol{x}_n|z_n = k) \tag{5}$$

$$= \prod_{n=1}^{N}\sum_{k=1}^{K} \pi^k \boldsymbol{\mathcal{N}}(\boldsymbol{x}_n; \boldsymbol{\mu}^k, (\sigma^k)^2) \tag{6}$$

In the GMM implementation, we will seek to minimize the negative logarithmic likelihood as our loss function.

$$\boldsymbol{\mathcal{L}}(\boldsymbol{\mu}, \sigma, \pi) = -\log P(\boldsymbol{X}) \tag{7}$$

The maximum likelihood estimate for this loss function will be a set of parameters $\boldsymbol{\mu}, \sigma, \pi$ that minimize this loss function.

2.2.1 Implementing the Loss Function

The above loss function is minimized using the Adam optimizer in Tensorflow. In order to optimize this loss function, two substitutions will need to be made, in order to ensure that the standard deviations σ and the mixture component probabilities π^k are constrained.

In order to constrain the standard deviations, an unconstrained variable ϕ is initialized from a random normal distribution. The variance of the Gaussian distributions is then set equal to the exponent of this variable, namely $\sigma^2 = exp(\phi)$. This restricts the values of the standard deviations σ to lie in the interval $\sigma \in [0, \infty)$.

Next the mixture component probabilities π^k are constrained so that $\sum_k \pi^k = 1$. This constraint can be satisfied by feeding another unconstrained variable ψ through the softmax function, namely $\pi^k = \frac{\exp(\psi^k)}{\sum_{k'} \exp(\psi^{k'})}$. This ensures that the mixture component probabilities sum to 1, satisfying the simplex constraint. The code implementations of both constraint substitutions are given below.

```
def initializeVars():
    centroid = tf.get_variable('mean', shape=(k,dim), initializer = tf.initializers.random_normal())
    phi = tf.get_variable('phi', shape=(1, k), initializer = tf.initializers.random_normal()) # Unconstrained form of variance
    variance = tf.math.exp(phi) # Constrained form of variance
    gamma =  tf.get_variable('gamma', shape=(k,1), initializer = tf.initializers.random_normal()) # Unconstrained form of weights
    logPi =  tf.transpose(hlp.logsoftmax(gamma)) # Constrained form of weights

    return centroid, variance, logPi
```

Figure 12: Initialization of unconstrained variables ϕ and ψ.

Finally, to tie in all these functions, we implement the graph below, and then use the graph to optimize our variables:

```
def buildGraph():
    tf.reset_default_graph() # Clear any previous junk
    tf.set_random_seed(45689)

    trainingInput = tf.placeholder(tf.float32, shape=(None, dim))  # Data placeholder
    centroid, variance, logPi = initializeVars() # Initialize mean, variance, and weights
    logPDF = log_GaussPDF(trainingInput,centroid,variance) # Calculate P(X|Z)
    logPosterior  = log_posterior(logPDF, logPi)  # Calculates the Bayesian P(Z|X)
    loss = -1.0*tf.reduce_sum(hlp.reduce_logsumexp(logPDF + logPi,keep_dims=True)) # Calculate the loss
    optimizer =tf.train.AdamOptimizer(learning_rate= 0.01, beta1=0.9, beta2=0.99,epsilon=1e-5).minimize(loss)

    # The tf.exp term returns the probabilites and weights in a form humans can understand
    return optimizer, loss, tf.exp(logPosterior), centroid, variance, tf.exp(logPi), trainingInput
```

Figure 13: Code snippet of the Build Graph function used to create the graph to be optimized by the Adam optimizer in Tensorflow.

```
def GMM(valid = False):
    loadData(valid) # Load the Data
    optimizer, loss, logPosterior, centroid, variance, weight, trainingInput = buildGraph()
    init = tf.global_variables_initializer()
    with tf.Session() as sess:
        sess.run(init)
        for i in range(0,epochs):
            _,  trainLoss[i], mU, var, weights, probs= sess.run([optimizer, loss, centroid, variance, weight, logPosterior], feed_dict = {trainingInput: t

            if valid: # Find validation loss and probability if this is True
                validLoss[i], probsV = sess.run([loss, logPosterior], feed_dict = {trainingInput: validData})

        assign = np.argmax(probs,1)  # Assign the point to the highest probability cluster
        inCluster = np.mean(np.eye(k)[assign],0) # Find the average number of points in each cluster
        plotter(valid) # Plot the Loss vs Epochs graph
        scatter(trainData, assign, mU, np.transpose(var), weights) # Draw a 2D scatter plot. For dim = 2 only

        if valid:
            assignV = np.argmax(probsV,1) # If we are validating, use the distances of the validation points to assign the cluster
            inClusterV = np.mean(np.eye(k)[assignV],0) # Find the average number of points in each cluster
            scatter(validData, assignV, mU, np.transpose(var), weights) # Draw a 2D scatter plot for the validation points. For dim = 2 only
            return mU, var, weights, inCluster, inClusterV

    # Return nothing if we are not validating for probsV and inClusterV
    return  mU, var, weights,inCluster, None
```

Figure 14: Code snippet of the optimization function which determines the optimized location for each centroid, and determines which centroid each data point belongs to.

The optimization is then performed for the $D = 2$ data set, and the loss per epoch is plotted below.

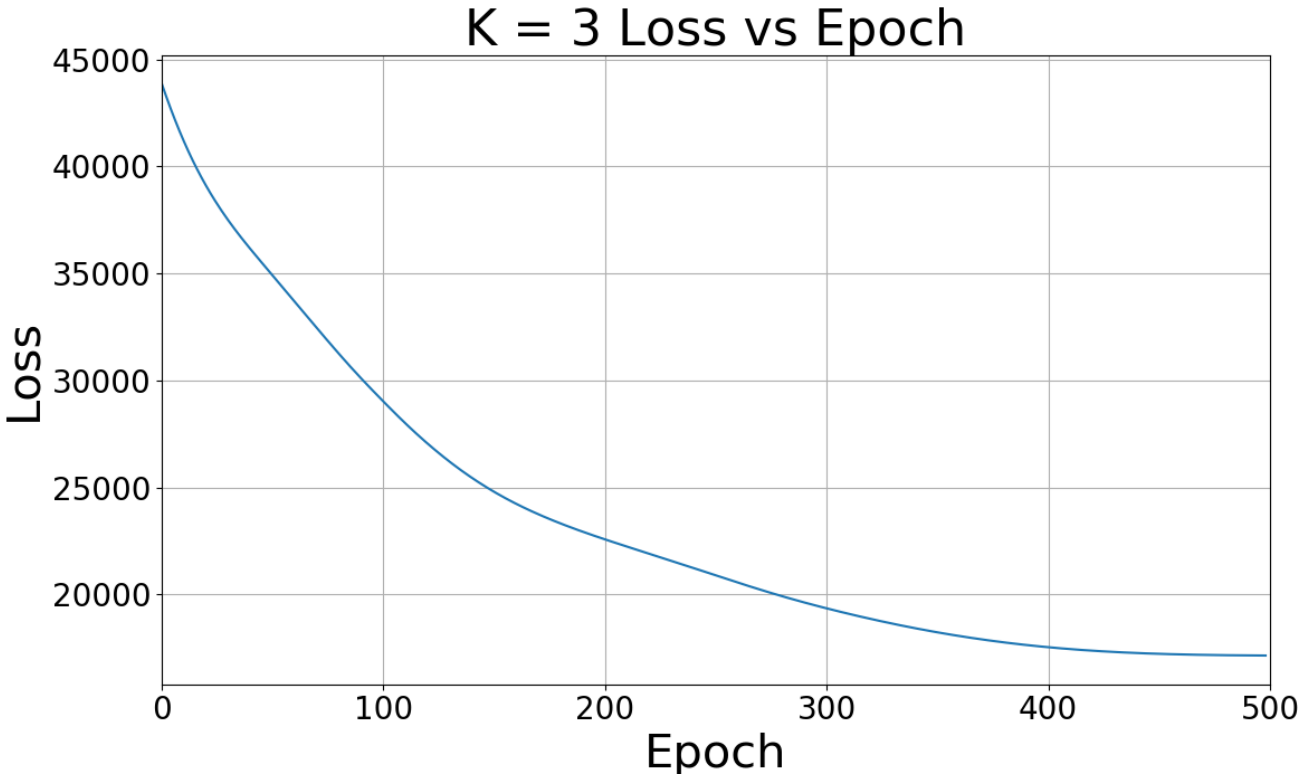

Figure 15: Loss per epoch when training the GMM using $K = 3$ for the $D = 2$ data set.

The optimal parameter models for this optimization are presented below.

	$\boldsymbol{\mu}$		σ^2		π	
Optimized Value	$\begin{bmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \\ \boldsymbol{\mu}_3 \end{bmatrix} = \begin{bmatrix} -1.10 & -3.31 \\ 1.30 & 0.31 \\ 0.11 & -1.52 \end{bmatrix}$		$\begin{bmatrix} \sigma_1^2 \\ \sigma_2^2 \\ \sigma_3^2 \end{bmatrix} = \begin{bmatrix} 0.0427 \\ 0.0388 \\ 0.9818 \end{bmatrix}$		$\begin{bmatrix} \pi_1 \\ \pi_2 \\ \pi_3 \end{bmatrix} = \begin{bmatrix} 0.334 \\ 0.333 \\ 0.333 \end{bmatrix}$	

Table 3: Optimal model parameters for the GMM implementing 3 Gaussians on the $D = 2$ data set.

2.2.2 Implementing Training and Validation Segmentation

Next, $\frac{1}{3}$ of the data set is held out of the training to be used as a validation set. The model is then trained for $K = \{1, 2, 3, 4, 5\}$, and the final loss value on the validation set is given in the table below.

Number of Clusters	Validation Loss
K = 1	11651.40
K = 2	7987.79
K = 3	5629.62
K = 4	5629.64
K = 5	5629.58

Table 4: Loss of the GMM on the validation set for optimizations where K $= 1, 2, 3, 4, 5$.

The following 5 plots display the cluster association of each data point for each of the 5 models that were optimized.

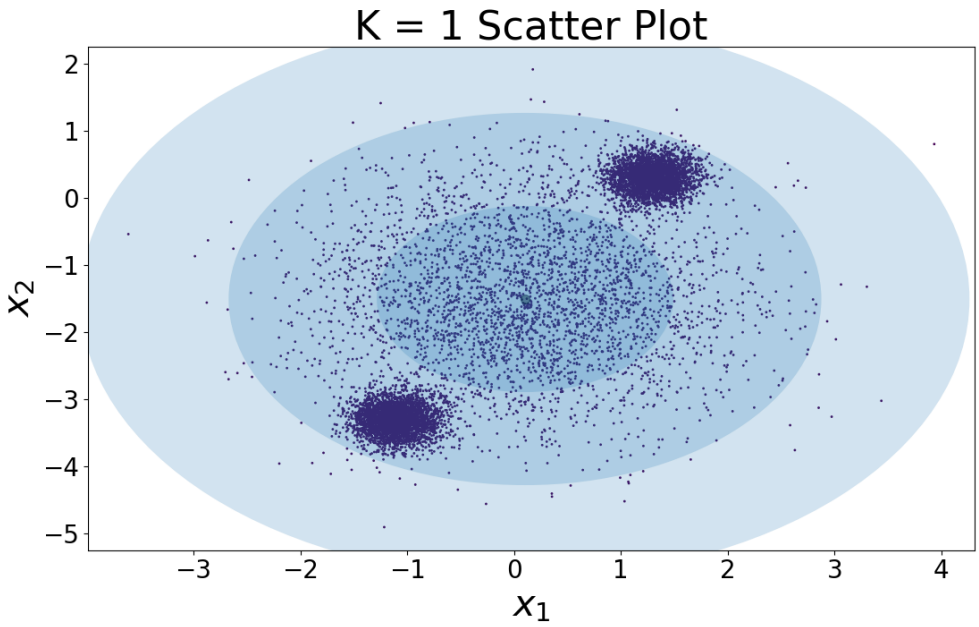

Figure 16: 2 dimensional scatter plot of the $D = 2$ data set for $K = 1$ with colour coded data point - Gaussian assignments.

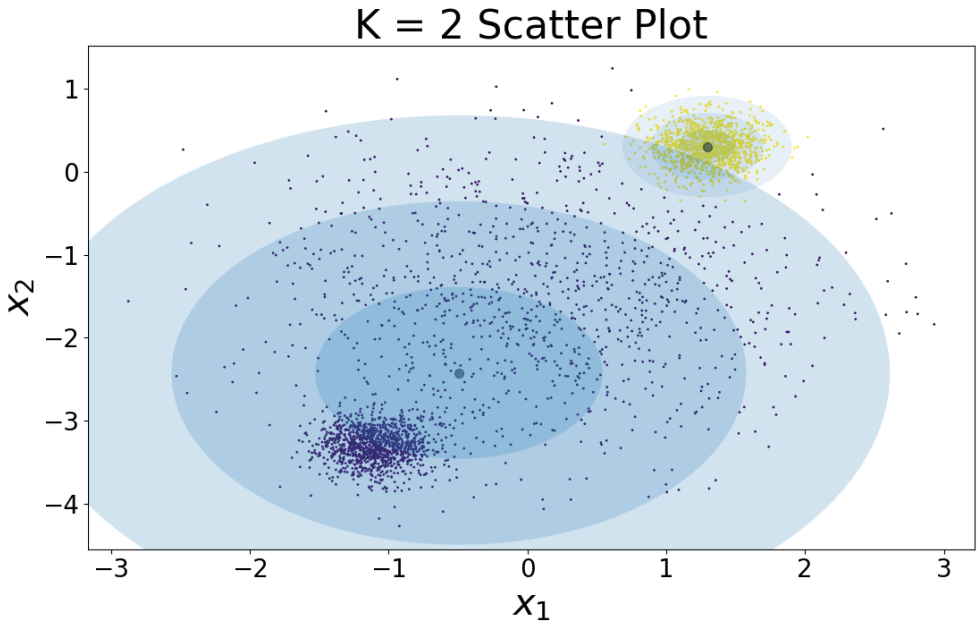

Figure 17: 2 dimensional scatter plot of the $D = 2$ data set for $K = 2$ with colour coded data point - Gaussian assignments.

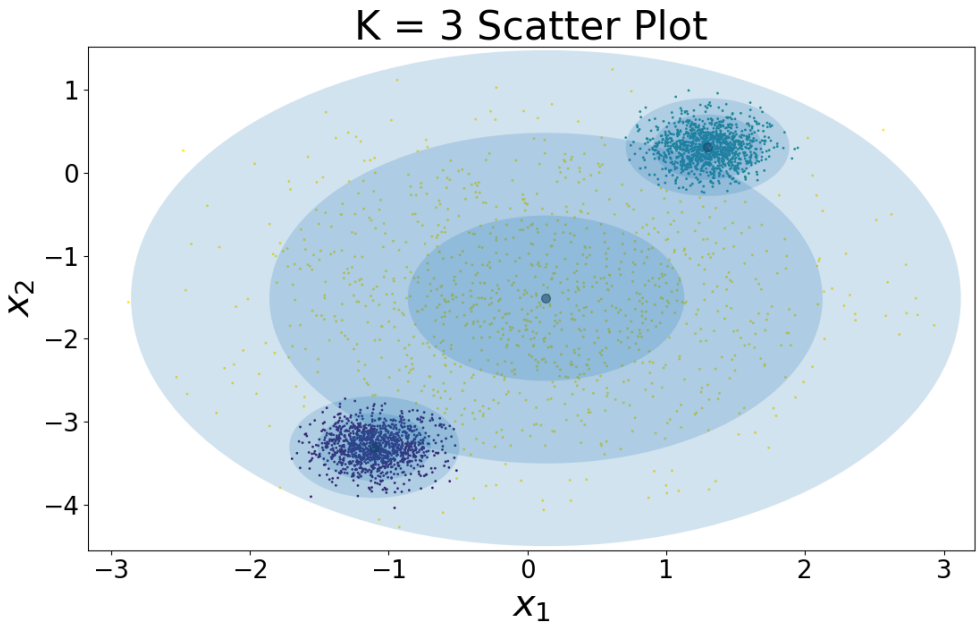

Figure 18: 2 dimensional scatter plot of the $D = 2$ data set for $K = 3$ with colour coded data point - Gaussian assignments.

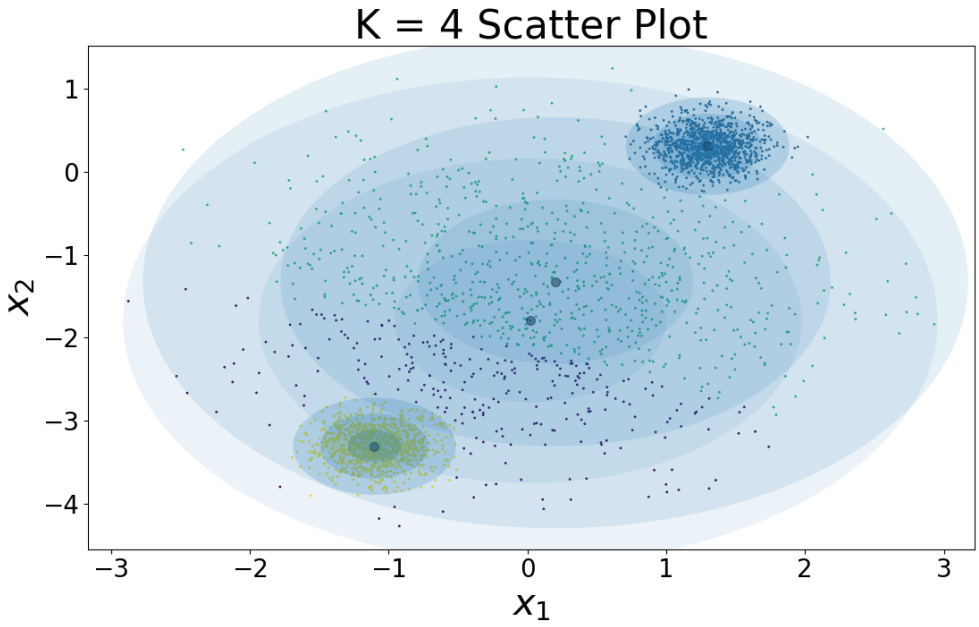

Figure 19: 2 dimensional scatter plot of the $D = 2$ data set for $K = 4$ with colour coded data point - Gaussian assignments.

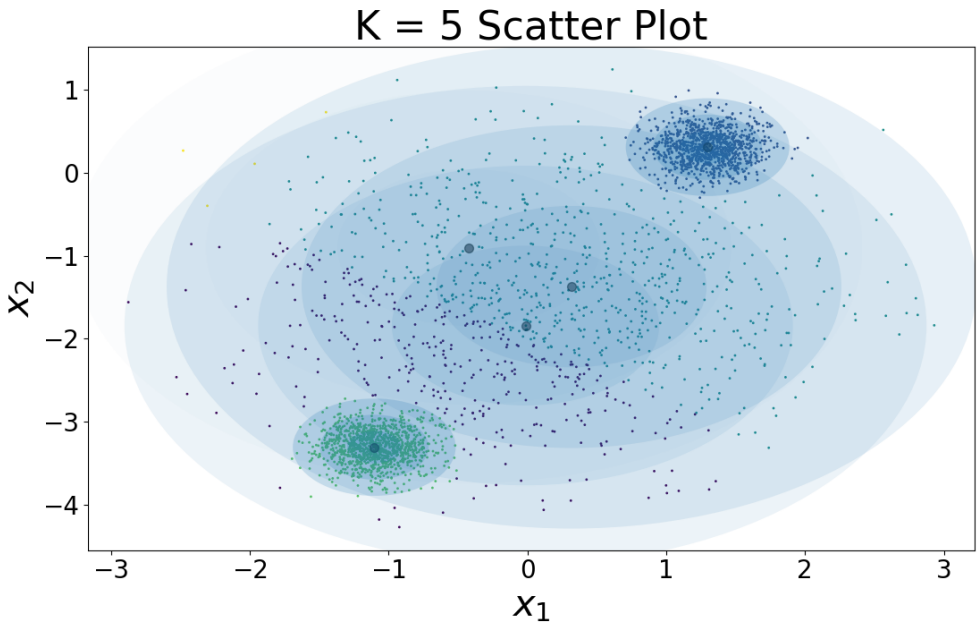

Figure 20: 2 dimensional scatter plot of the $D = 2$ data set for $K = 5$ with colour coded data point - Gaussian assignments.

Based on the above 5 plots, 3 seems to be the optimal number of Gaussian clusters. For $K = 2$, the data points laying in between the two dense clusters are assigned to one of the dense Gaussians. This assignment seems arbitrary, and therefore $K = 2$ does not seem appropriate. When the number of Gaussian clusters is increased to 4 or 5, the data points not belonging to the two dense clusters are represented by 2 or 3 Gaussians located very near to one another. This seems to indicate that there isn't much discrepancy between these data points, and therefore they can be represented by just one Gaussian. Therefore, along with the 2 densely packed areas represented by their own Gaussian, $K = 3$ seems to be the correct number of clusters for this data set.

2.2.3 Modelling the 100D Data Set

Finally, both models are run on the $D = 100$ data set for $K = \{5, 10, 15, 20, 30\}$.

Number of Clusters	Loss
K = 5	215,509
K = 10	215,268
K = 15	215,361
K = 20	212,945
K = 30	211,200

Table 5: Loss of the KMeans K $= 5, 10, 15, 20, 30$.

Number of Clusters	Loss
K = 5	1,091,210
K = 10	834,024
K = 15	834,038
K = 20	486,583
K = 30	484,477

Table 6: Loss of the GMM K $= 5, 10, 15, 20, 30$.

Based on the loss values provided from the Gaussian Mixture model, it appears as though, of the 5 values of K chosen, $K = 20$ best represents the number of clusters in the data set. We can see this by observing that increasing the number of clusters from 20 to 30 has minimal impact on the loss of the validation set, however increasing the number of clusters from 15 to 20 nearly halves the loss attained by the model. Clearly, we want to minimize loss in this situation, and $K = 20$ appears to have best loss improvement per additional clusters out of all the trials.

In a similar vein, when comparing the losses provided from the K-means clustering algorithm, $K = 20$ seems to have a drastic impact on the loss of the model. While $K = 5, 10, 15$ all result in roughly the same loss, using 20 clusters results in a loss reduction of about 2500. Increasing the number of clusters to 30 results in a further reduction of about 2000, but this improvement may

not be as advantageous due to the differences in the GMM and Kmeans models.

In K-means, more clusters is nearly always better for improving the loss achieved by the model; since the loss function merely uses the minimum of the distances from the point to each cluster, having more clusters results in a smaller total loss. Therefore, reductions in loss do not always correspond to better models, it can also represent overfitting of the data.

With GMMs, the log probability that a point belongs to each cluster is used. Clearly, more clusters will still result in lower total loss, however, since GMM determines the probability that a point belongs to each cluster, increasing cluster number does not have as drastic an impact on the loss, since these additional clusters are occasionally placed directly beside one another, as can be seen in figure 20.

Thus, when the data cannot be visualized, the loss values determined from the GMM are a better indication of how many clusters are actually present within the dataset when compared to the loss values calculated from the Kmeans model.