ECE521 Winter 2017: Assignment 3

FuYuan Tee, (999295837) * Chee Loong Soon, (999295793) †

March 24th, 2017

Contents

*Equal Contribution (50%), fuyuan.tee@mail.utoronto.ca

†Equal Contribution (50%), cheeloong.soon@mail.utoronto.ca

1 K-means

1.1 Learning K-means

1.1.1 Convexity of $\mathcal{L}(\mu)$

$$L(\mu) = \sum_{n=1}^{B} min_{k=1}^{K} ||x_n - \mu_k||_2^2 \tag{1}$$

B is the number of training instances, K is the number of clusters. μ_k is the cluster mean for cluster k. x_n is the data points for the n^{th} input data. D is the dimension of the input data.

For the function 1 to convex, it must satisfy the Jensen's Inequality as shown in equation 2.

$$L(\alpha\mu^1 + (1-\alpha)\mu^2) \leq \alpha L(\mu^1) + (1-\alpha)L(\mu^2) \tag{2}$$

where $\alpha \in [0, 1]$

We will prove that the $L(\mu)$ function in 1 is not Convex by a proof by Counter Example. We will show there exist an example that does not satisfy the Jensen Inequality, hence proving that the function $L(\mu)$ is not convex.

Proof By Counter-Example:

Let $B = 1$, $D = 1$ represent a one single dimension input data, x_1. Let $K = 2$ represent two different number of clusters we are supposed to train and let μ_1 and μ_2 be their respective means.

The first configuration of cluster, μ^1 is shown in equation 3.

$$\begin{aligned} \mu^1 = (\mu_1^1, \mu_2^1) = [1, 3] \\ x_1 = (1) \\ L(\mu^1) = min((x_1 - \mu_1^1)^2, (x_1 - \mu_2^1)^2) \\ = min((1-1)^2, (1-3)^2) = min(0, 4) = 0 \end{aligned} \tag{3}$$

The second configuration of cluster, μ^2 is shown in equation 4

$$\begin{aligned} \mu^2 = (\mu_1^2, \mu_2^2) = [3, 1] \\ x_1 = (1) \\ L(\mu^2) = min((x_1 - \mu_1^2)^2, (x_1 - \mu_2^2)^2) \\ = min((1-3)^2, (1-1)^2) = min(4, 0) = 0 \end{aligned} \tag{4}$$

Therefore, from equation 2, set $\alpha = 0.5$.

The Right Hand side of 2 would be 0 as shown in equation 5.

$$\begin{aligned} \alpha L(\mu^1) + (1-\alpha)L(\mu^2) = \\ 0.5 * 0 + (1 - 0.5) * 0 = 0 \end{aligned} \tag{5}$$

However, the Left hand side of 2 would be 1 as shown in equation 6.

$$\begin{aligned} L(\alpha\mu^1 + (1-\alpha)\mu^2) = \\ L(0.5[1, 3] + (1 - 0.5)[3, 1]) = \\ L([2, 2]) = min((1-2)^2, (1-2)^2) \\ = min(1, 1) = 1 \end{aligned} \tag{6}$$

Since equation 6 is $>$ equation 5, this violates Jensen's Inequality as shown in 2. This proves that the Loss function, $L(\mu)$ is NOT convex.

1.1.2 K-means without validation, $(K = 3)$

The K-means algorithm was employed to cluster *data2D.npy* using $K = 3$ clusters. An Adam optimizer was used in the training of the K-means cluster centers. The cluster centers, $\boldsymbol{\mu}$, learnt from this training are as shown in Table 1.

Table 1: Cluster centers using K-means ($K = 3$) on *data2D.npy* without validation

Cluster	Center
1	$(-1.1, -3.2)$
2	$(0.1, -1.5)$
3	$(1.2, 0.3)$

A graph of the loss function, $\mathcal{L}(\boldsymbol{\mu}) = \sum_{n=1}^{B} \min_{k=1}^{K} ||\mathbf{x}_n - \boldsymbol{\mu}_k||_2^2$, as training progressed is depicted in Figure 1.

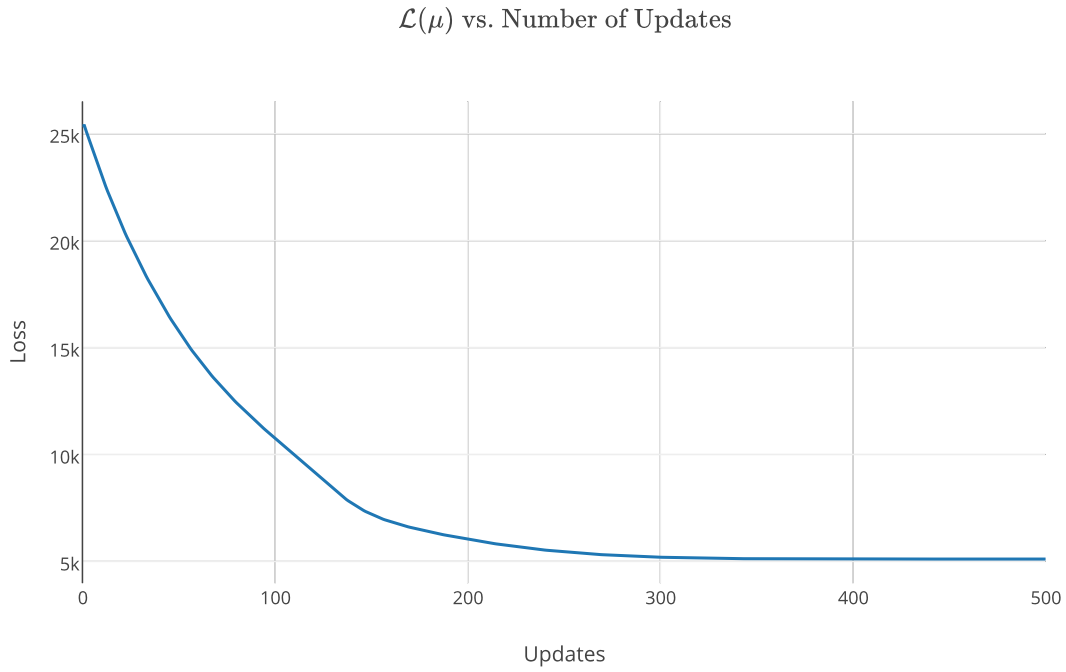

Figure 1: Loss function with respect to training updates.

1.1.3 K-means without validation, $(K = 1, 2, 3, 4, 5)$

The training procedure from the previous section was repeated for values of K ranging from 1 to 5. The training results are summarised in Table 2. The percentage of data points assigned to each cluster is also illustrated in Figure 2.

Table 2: K-means training results on *data2D.npy* without validation

K	Training Loss	Cluster	Center	% Data Points
1	38454	1	$(0.1, -1.5)$	100
2	9203	1	$(1.1, -0.1)$	49.5
		2	$(-0.8, -2.9)$	50.5
3	5111	1	$(-1.1, -3.2)$	38.2
		2	$(0.1, -1.5)$	23.8
		3	$(1.2, 0.3)$	38.0
4	3374	1	$(0.8, -2.0)$	13.5
		2	$(-0.7, -1.1)$	12.1
		3	$(1.3, 0.3)$	37.3
		4	$(-1.1, -3.3)$	37.1
5	2848	1	$(0.3, -2.4)$	8.7
		2	$(-0.7, -1.0)$	10.7
		3	$(1.3, 0.3)$	36.1
		4	$(-1.1, -3.3)$	35.8
		5	$(1.1, -1.4)$	8.6

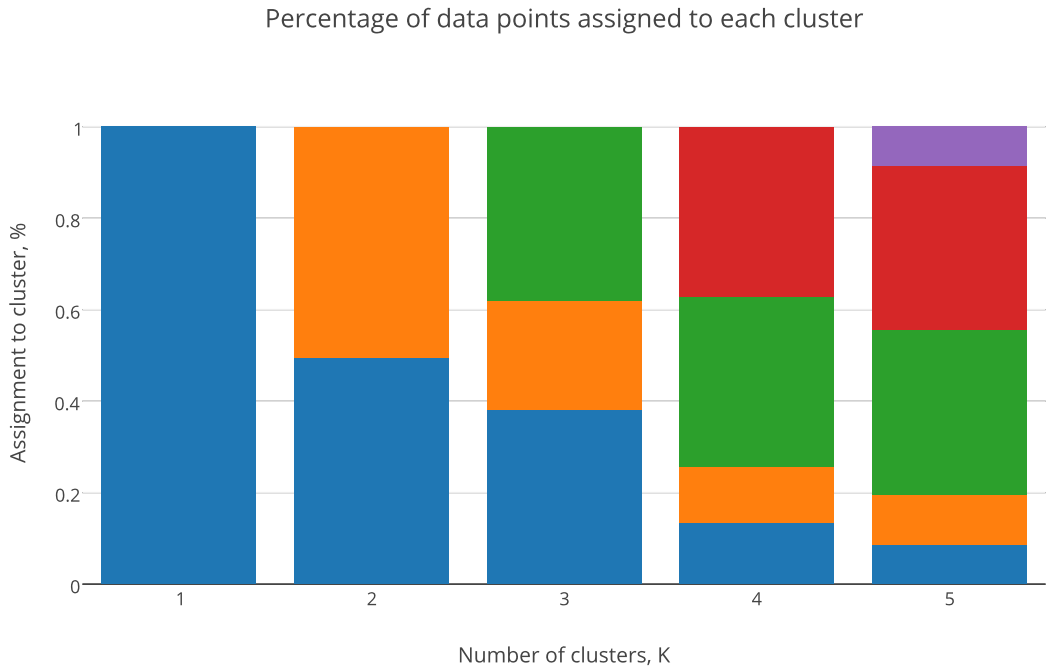

Figure 2: Percentage of data points assigned to the first, second, third, fourth and fifth clusters for $K = 1, 2, 3, 4, 5$.

To understand how many clusters to best cluster the dataset, we provided plots visualising the assignments of data points to each cluster. These illustrations can be found in Figures 3 and 4.

From inspecting these figures, the data points can be seen to comprise of 3 clusters. However, the hard K-means clustering of $K = 3$ leads to non-ideal clustering since portions of the data points that visually belong to the middle cluster were assigned to the the top-most and bottom-most clusters (Figure 3, $K = 3$). Hence, a clustering assignment of $K = 4$ looks to be an appropriate and parsimonious choice.

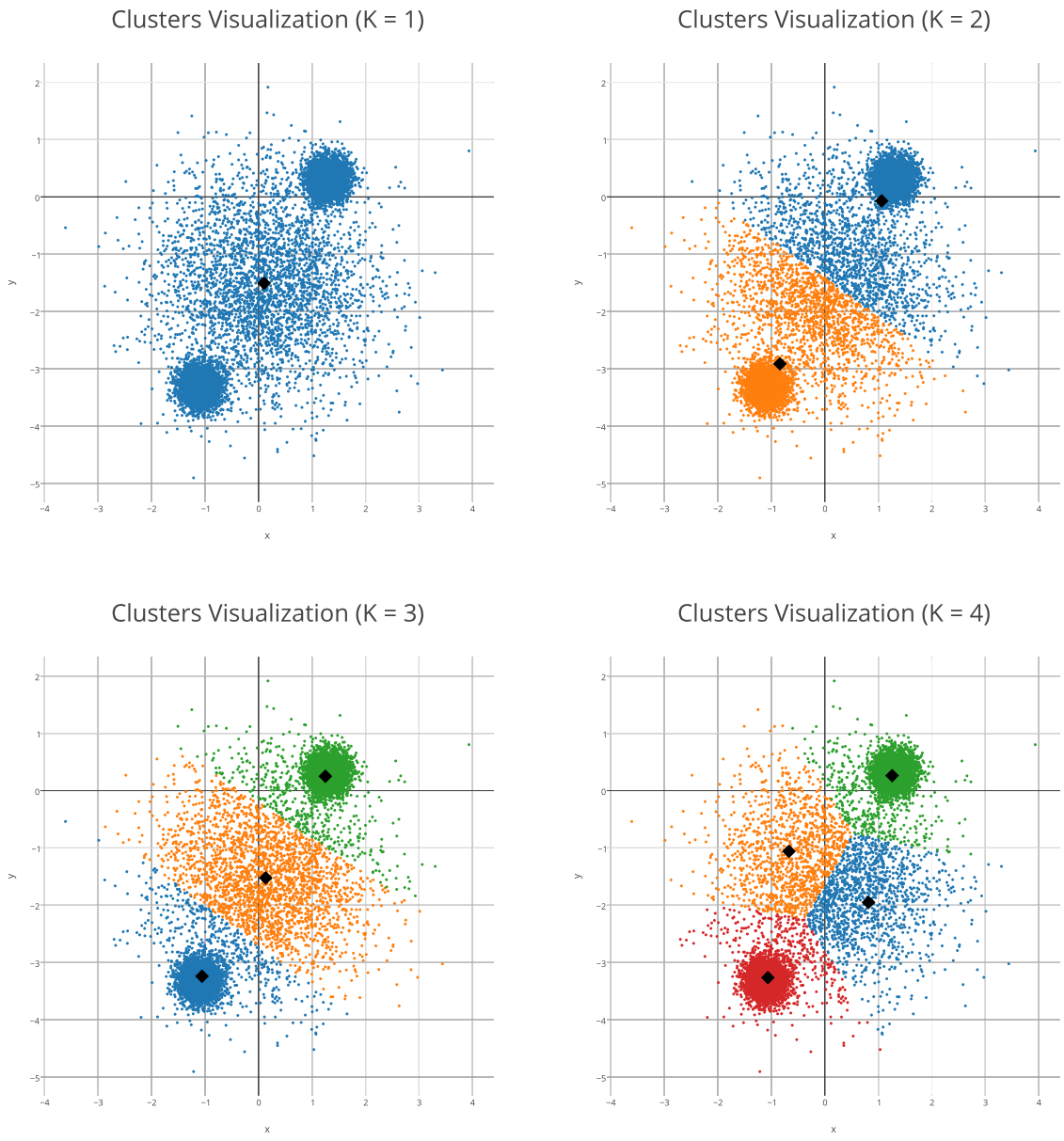

Figure 3: Assignment of data points to nearest K-means clusters, for $K = 1, 2, 3, 4$. For each run of K, data points are coloured to their corresponding cluster assignments: { 1, 2, 3, 4 }. The black diamonds represent the cluster centers.

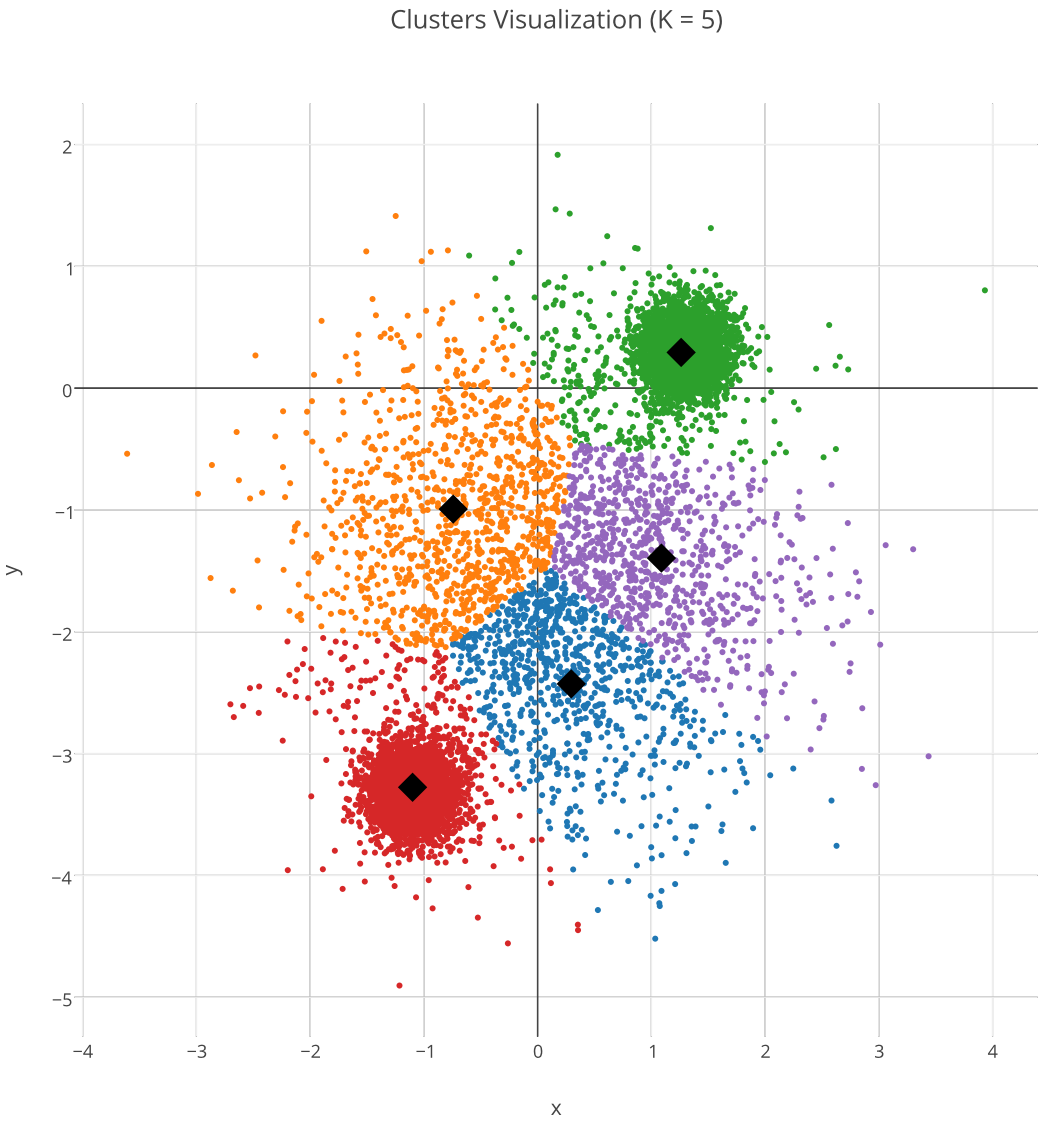

Figure 4: Assignment of data points to nearest K-means clusters, for $K = 5$. Data points are coloured to their corresponding cluster assignments: { 1, 2, 3, 4, 5 }. The black diamonds represent the cluster centers.

1.1.4 K-means with validation

Similarly, the training procedure from the previous section was repeated, with 1/3 of the data held out for validation.

The training results are summarised in Table 3. The percentage of data points assigned to each cluster has not changed significantly compared to the previous section. As such, one can refer to Figure 2 for the proportion of data points to each cluster.

From the table, $K = 5$ clusters seems to be the best, based on the lowest validation loss.

Table 3: K-means training results on *data2D.npy* with validation

K	Validation Loss	Cluster	Center	% Data Points
1	12877	1	$(0.1, -1.5)$	100
2	3140	1	$(1.1, -0.1)$	49.6
		2	$(-0.8, -2.9)$	50.4
3	1745	1	$(-1.1, -3.2)$	38.6
		2	$(0.2, -1.5)$	23.4
		3	$(1.2, 0.2)$	38.1
4	1133	1	$(0.8, -2.0)$	13.0
		2	$(-0.6, -1.1)$	12.2
		3	$(1.3, 0.3)$	37.3
		4	$(-1.1, -3.3)$	37.4
5	948	1	$(0.3, -2.4)$	8.9
		2	$(-0.7, -1.0)$	10.6
		3	$(1.3, 0.3)$	36.0
		4	$(-1.1, -3.3)$	36.1
		5	$(1.1, -1.4)$	8.5

2 Mixture of Gaussians (MoG)

2.1 The Gaussian cluster model

2.1.1 Deriving the latent variable posterior distribution for a data point, $P(z \mid \mathbf{x})$

Given a total of K clusters, let k be the k-th cluster the data points may assigned to: $k \in \{1, ..., K\}$.

From Bayes' Rule,

$$P(z = k \mid \mathbf{x}) = \frac{P(\mathbf{x} \mid z = k)P(z = k)}{P(\mathbf{x})} \tag{7}$$

Given the prior $P(z = k) = \pi_k$, and the probability density function being a multivariate Gaussian distribution,

$$\begin{aligned} P(z = k \mid \mathbf{x}) &= \frac{\pi_k P(\mathbf{x} \mid z = k)}{\sum_{j=1}^{K} P(\mathbf{x} \mid z = j)P(z = j)} \\ &= \frac{\pi_k \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} P(\mathbf{x} \mid z = j)P(z = j)} \end{aligned} \tag{8}$$

Assuming that all D dimensions are independent and of equal variance, $\boldsymbol{\Sigma}_i = \sigma_i^2 \mathbf{I}_D \in \mathbb{R}^{D \times D}, \forall i = 1, \ldots, K$ where $\sigma_i^2 \in \mathbb{R}$.

$$P(z = k \mid \mathbf{x}) = \frac{\pi_k \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_k, \sigma_k^2 \mathbf{I}_D)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_j, \sigma_j^2 \mathbf{I}_D)} \tag{9}$$

2.1.2 Computing the log probability density function, $\log \mathcal{N}(\mathbf{x} \mid \mu_k, \sigma_k^2 \mathbf{I}_D)$

Taking the $\log$ of the probability density function,

$$
\begin{aligned}
\log \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_k, \sigma_k^2 \mathbf{I}_D) &= \log \prod_{i=1}^{D} \mathcal{N}(x_i \mid \mu_{ki}, \sigma_k^2) \\
&= \sum_{i=1}^{D} \log \mathcal{N}(x_i \mid \mu_{ki}, \sigma_k^2) \\
&= \sum_{i=1}^{D} \log \left[\frac{1}{\sqrt{2\pi\sigma_k^2}} \exp\left\{ -\frac{(x_i - \mu_{ki})^2}{2\sigma_k^2} \right\} \right] \\
&= \sum_{i=1}^{D} \left[-\frac{1}{2} \log\left(2\pi\sigma_k^2\right) - \frac{(x_i - \mu_{ki})^2}{2\sigma_k^2} \right] \\
&= -\sum_{i=1}^{D} \frac{(x_i - \mu_{ki})^2}{2\sigma_k^2} - \frac{D}{2} \log\left(2\pi\sigma_k^2\right)
\end{aligned}
\tag{10}
$$

The Python code snippet for implementing the Equation 10 is attached.

```
'''
Calculate log probability density function for all pairs of B data points and K clusters

Assumptions:
Dimensions are independent and have the same standard deviation, sigma
Output:
log PDF function (N x K)
'''
def calc_log_gaussian_cluster_k(X, Mu, sigma_sq):
    with tf.name_scope('log_gaussian_cluster'):
        # Infer dimension of data
        D = tf.shape(X)[1]

        # Calculate Mahalanobis distance
        ### Expand dim(X) to (N x 1 x D)
        ### Expand dim(Mu) to (1 x K x D)
        ### Reduce sum along the D-axis
        with tf.name_scope('mahalanobis_dist'):
            dist = - tf.divide(tf.reduce_sum(tf.square(tf.expand_dims(X, axis=1) - tf.expand_dims(Mu, axis=0)), axis=2), 2 * tf.transpose(sigma_sq), name='mahalanobis_dist')

        # Calculate log of gaussian constant term
        ### Transpose sigma_sq to (1 x K)
        with tf.name_scope('log_gauss_const'):
            log_gauss_const = - tf.multiply(tf.cast(D, tf.float32) / 2, tf.log(2 * np.pi * tf.transpose(sigma_sq)), name='log_gauss_const')

        # Sum results
        log_gaussian_cluster = tf.add(dist, log_gauss_const, name='log_gauss_cluster')

    return log_gaussian_cluster
```

2.1.3 Computing the conditional responsibilities, $\gamma(\mathbf{x}) = \log P(z \mid \mathbf{x})$

$$
\begin{aligned}
&\log P(z \mid \mathbf{x}) \\
&= \log \frac{\pi_k \prod_{i=1}^{D} \mathcal{N}(x_i \mid \mu_{ki}, \sigma_k^2)}{\sum_{i=1}^{K} \pi_j \prod_{i=1}^{D} \mathcal{N}(x_i \mid \mu_{ji}, \sigma_j^2)} \\
&= \log \left[\pi_k \prod_{i=1}^{D} \mathcal{N}(x_i \mid \mu_{ki}, \sigma_k^2) \right] - \log \left[\sum_{j=1}^{K} \pi_j \prod_{i=1}^{D} \mathcal{N}(x_i \mid \mu_{ji}, \sigma_j^2) \right] \\
&= \log \pi_k + \log \prod_{i=1}^{D} \mathcal{N}(x_i \mid \mu_{ki}, \sigma_k^2) - \log \left\{ \sum_{j=1}^{K} \pi_j \frac{1}{(2\pi\sigma_j^2)^{\frac{D}{2}}} \prod_{i=1}^{D} \exp \left[-\frac{(x_i - \mu_{ji})^2}{2\sigma_j^2} \right] \right\} \\
&= \log \pi_k - \sum_{i=1}^{D} \frac{(x_i - \mu_{ki})^2}{2\sigma_k^2} - \frac{D}{2} \log\left(2\pi\sigma_k^2\right) - \log \left\{ \sum_{j=1}^{K} \frac{\pi_j}{(2\pi\sigma_j^2)^{\frac{D}{2}}} \exp \left[-\sum_{i=1}^{D} \frac{(x_i - \mu_{ji})^2}{2\sigma_j^2} \right] \right\} \\
&= -\log \left\{ \sum_{i=1}^{K} \exp \left[\log \left(\frac{\pi_j}{(2\pi\sigma_j^2)^{\frac{D}{2}}} \right) - \sum_{i=1}^{D} \frac{(x_i - \mu_{ji}^2)}{2\sigma_j^2} \right] \right\} - \sum_{i=1}^{D} \frac{(x_i - \mu_{ki})^2}{2\sigma_k^2} - \frac{D}{2} \log\left(2\pi\sigma_k^2\right) + \log \pi_k \\
&= -\log \left\{ \sum_{i=1}^{K} \exp \left[-\sum_{i=1}^{D} \frac{(x_i - \mu_{ji})^2}{2\sigma_j^2} - \frac{D}{2} \log\left(2\pi\sigma_j^2\right) + \log \pi_j \right] \right\} \\
&\qquad\qquad - \sum_{i=1}^{D} \frac{(x_i - \mu_{ki})^2}{2\sigma_k^2} - \frac{D}{2} \log\left(2\pi\sigma_k^2\right) + \log \pi_k
\end{aligned}
\tag{11}
$$

The Python code snippet for implementing the Equation 11 is attached.

```
'''
Calculate log probability cluster variable z given x, a.k.a. conditional responsibilities, gamma

Output:
    conditional responsibilities (N x K)
'''
def calc_log_conditional_responsibilities(X, Mu, sigma_sq, log_pi):
    with tf.name_scope('log_conditional_responsibilities'):
        # Calculate unnormalised_log_posterior P(z|x)
        with tf.name_scope('unnormalised_log_posterior'):
            unnormalised_log_posterior = calc_log_gaussian_cluster_k(X, Mu, sigma_sq) + tf.transpose(log_pi)

```

```
        # Return log normalised posterior / conditional responsibilities
        with tf.name_scope('log_gamma_z'):
            cond_resp = tf.add(-
              ↪  tf.reduce_logsumexp(unnormalised_log_posterior, axis=1,
              ↪  keep_dims=True), unnormalised_log_posterior,
              ↪  name='log_gamma_z')
    return cond_resp
```

The function *tf.reduce_logsumexp* allows for more numerically stable computation. As described in TensorFlow r1.0 documentation, *tf.reduce_logsumexp* prevents both numerical overflow and underflow, from taking the exponent of large inputs and from taking the log of small inputs respectively. This is mitigated by first subtracting the maximum input to the function, evaluating the log-sum-exp expression before adding the maximum value back to obtain the final result. Also, we take the log as adding a bunch of log probabilities is more stable than multiplying a bunch of probabilities with values below 1.

2.2 Learning the MoG

2.2.1 Implicit cluster assignment variable

Proving $\nabla_{\boldsymbol{\mu}} \log P(\mathbf{x}) = \sum_{k=1}^{K} P(z = k \mid \mathbf{x}) \nabla_{\boldsymbol{\mu}} \log P(\mathbf{x}, z = k)$.

Starting from the Left-Hand Side (LHS):

$$
\begin{aligned}
LHS &\equiv \nabla_{\boldsymbol{\mu}} \log P(\mathbf{x}) \\
&= \frac{1}{P(\mathbf{x})} \nabla_{\boldsymbol{\mu}} P(\mathbf{x}) \\
&= \frac{1}{P(\mathbf{x})} \nabla_{\boldsymbol{\mu}} \left[\sum_{k=1}^{K} P(\mathbf{x}, z = k) \right] \\
&= \frac{1}{P(\mathbf{x})} \sum_{k=1}^{K} \nabla_{\boldsymbol{\mu}} P(\mathbf{x}, z = k) \\
&= \sum_{k=1}^{K} \frac{P(z = k \mid \mathbf{x})}{P(z = k \mid \mathbf{x}) P(\mathbf{x})} \nabla_{\boldsymbol{\mu}} P(\mathbf{x}, z = k) \\
&= \sum_{k=1}^{K} P(z = k \mid \mathbf{x}) \left[\frac{1}{P(\mathbf{x}, z = k)} \nabla_{\boldsymbol{\mu}} P(\mathbf{x}, z = k) \right] \\
&= \sum_{k=1}^{K} P(z = k \mid \mathbf{x}) \nabla_{\boldsymbol{\mu}} \log P(\mathbf{x}, z = k) \\
&\equiv RHS
\end{aligned}
\tag{12}
$$

Hence, it is shown that the cluster assignment variable, $P(z = k \mid \mathbf{x})$, is implicit in the maximisation of the marginal log-likelihood, $\log P(\mathbf{X})$.

2.2.2 MoG without validation on *data2D.npy*

In this section, the data points from *data2D.npy* were clustered using Mixture of Gaussian with number of clusters, $K = 3$. The loss function to be minimized in this approach is the negative marginal log-likelihood:

$$\begin{aligned} -\log P(\mathbf{X}) &= -\log \prod_{n=1}^{B} \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_k, \sigma_k^2) \\ &= -\sum_{n=1}^{B} \log \sum_{k=1}^{K} \frac{\pi_k}{(2\pi\sigma_k^2)^{\frac{D}{2}}} \exp\left[-\frac{(\mathbf{x}-\boldsymbol{\mu}_k)^T(\mathbf{x}-\boldsymbol{\mu}_k)}{2\sigma_k^2}\right] \\ &= -\sum_{n=1}^{B} \log \sum_{k=1}^{K} \exp\left[-\frac{1}{2\sigma_k^2}(\mathbf{x}-\boldsymbol{\mu}_k)^T(\mathbf{x}-\boldsymbol{\mu}_k) - \frac{D}{2}\log\big(2\pi\sigma_k^2\big) + \log \pi_k\right] \end{aligned} \tag{13}$$

where B is the number of data points.

Given the constraints $\sigma_k^2 \geq 0 \quad \forall k \in 1, ..., K$ and $\sum_{k=1}^{K} \pi_k = 1$, σ_k^2 and π_k are generated from unconstrained parameters ϕ_k and ψ_k respectively as follow:

$$\sigma_k^2 = \exp(\phi_k) \tag{14}$$

$$\pi_k = \frac{\exp(\psi_k)}{\sum_{k'=1}^{K} \exp(\psi_{k'})} \tag{15}$$

In implementing the MoG training procedure, an Adam Optimizer was used. No validation data points were used — all data points were used to train the model. $-P(\mathbf{X})$ and $\log \pi_k$ were calculated using TensorFlow functions *tf.log_softmax* and *tf.reduce_logsumexp* respectively.

The parameters learnt by the MoG model are summarised in Table 4. A graph of the loss function over number of updates is also provided in Figure 5.

Table 4: Trained MoG parameters for $K = 3$ on *data2D.npy* without validation

Cluster k	Center, $\boldsymbol{\mu}_k$	Variance, σ_k^2	Latent prior, $P(z = k) = \pi_k$
1	$(-1.1, -3.3)$	0.039	0.33
2	$(0.11, -1.5)$	0.99	0.33
3	$(1.3, 0.31)$	0.039	0.33

The loss is converges around 17132 as shown in Figure 5.

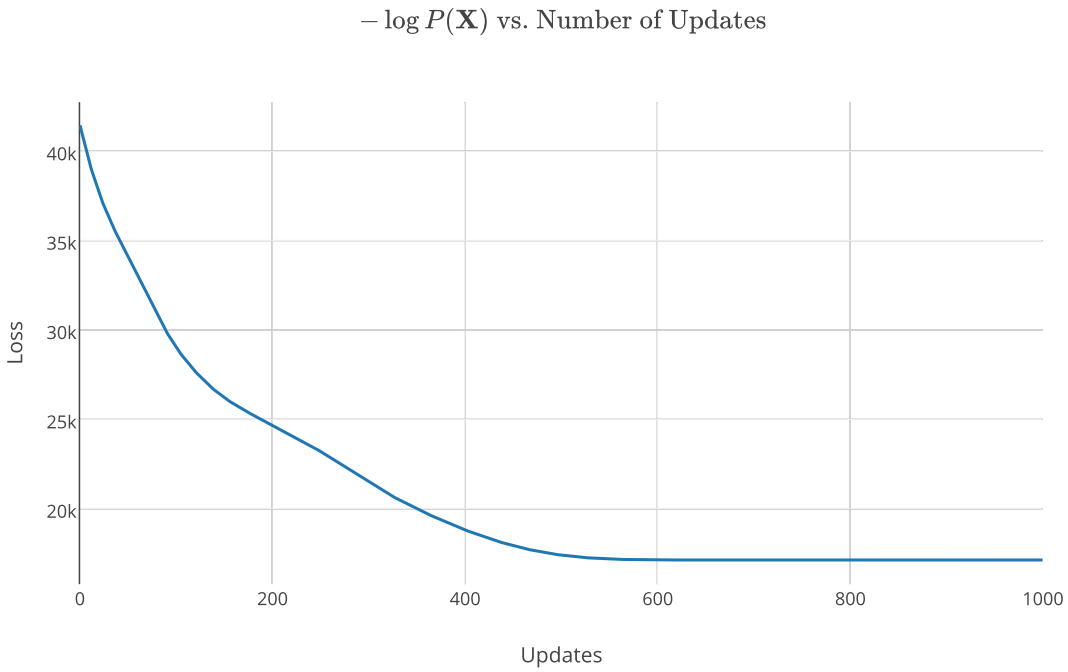

Figure 5: Loss function with respect to training updates.

2.2.3 MoG with validation

The training procedure from the previous section is repeated, this time with 1/3 of the *data2D.npy* data points held out for validation. MoG models for $K = 1, 2, 3, 4, 5$ were trained.

The training results are summarised in Table 5. A comparison of validation loss between MoG models and the percentage of data points assigned to the clusters in each MoG models are provided in Figures 6 and 7 respectively. Coloured scatters plots are provided in Figures 8 and 9, showing how data are assigned to each Gaussian cluster.

Table 5: MoG training results on *data2D.npy* with validation

K	Validation Loss	Cluster k	Center, $\boldsymbol{\mu}_k$	Variance, σ_k^2	Prior, π_k
1	11655	1	$(0.10, -1.5)$	1.92	1.00
2	8001	1	$(-0.51, -2.45)$	1.05	0.66
		2	$(1.29, 0.30)$	0.04	0.34
3	5743	1	$(-1.10, -3.30)$	0.04	0.34
		2	$(0.13, -1.52)$	0.98	0.33
		3	$(1.30, 0.31)$	0.04	0.33
4	5744	1	$(0.20, -1.64)$	0.96	0.16
		2	$(-0.07, -1.41)$	0.98	0.17
		3	$(-1.10, -3.30)$	0.04	0.34
		4	$(1.30, 0.31)$	0.04	0.33
5	5744	1	$(-0.34, -1.96)$	0.77	0.10
		2	$(0.72, -1.82)$	0.78	0.09
		3	$(-1.10, -3.30)$	0.04	0.34
		4	$(1.30, 0.31)$	0.04	0.33
		5	$(0.07, -1.06)$	0.87	0.15

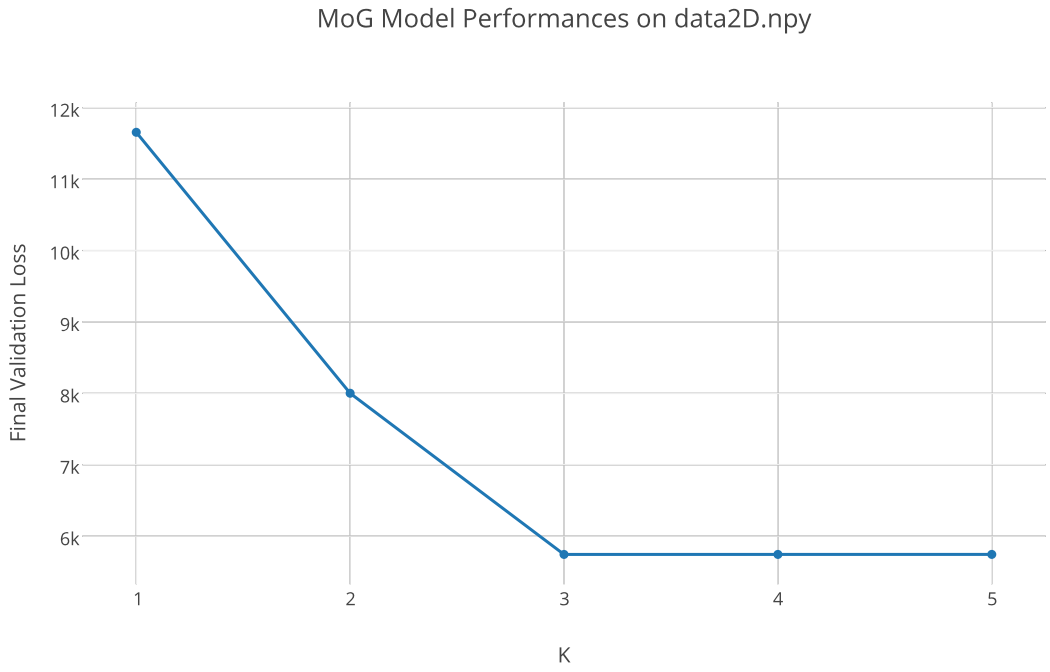

Figure 6: Comparison of model performance for various clusters ($K = 1, 2, 3, 4, 5$).

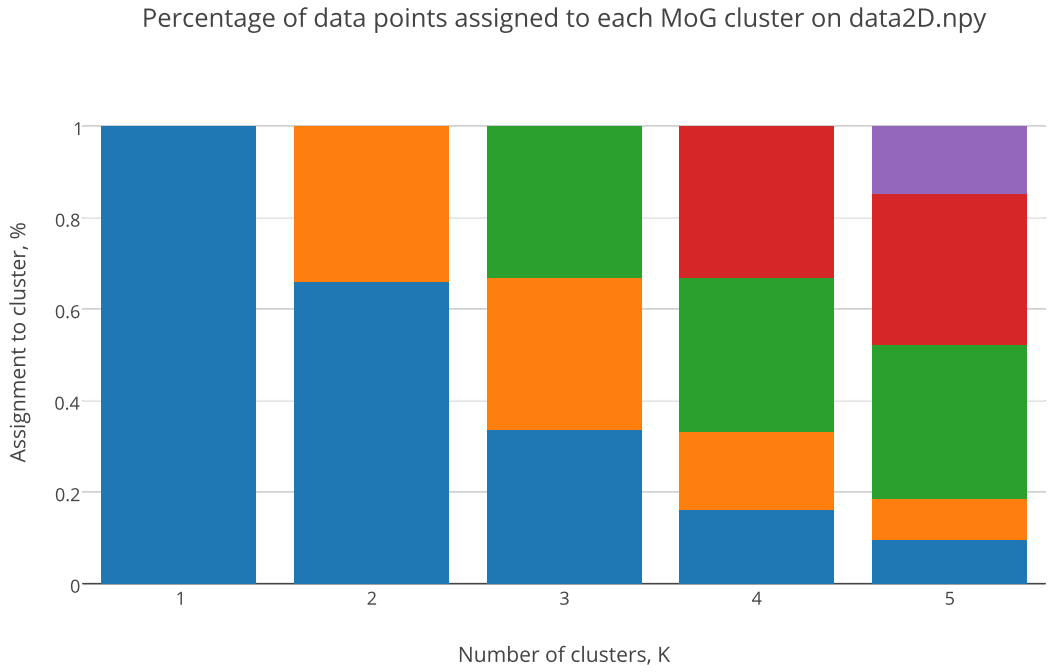

Figure 7: Loss function with respect to training updates.

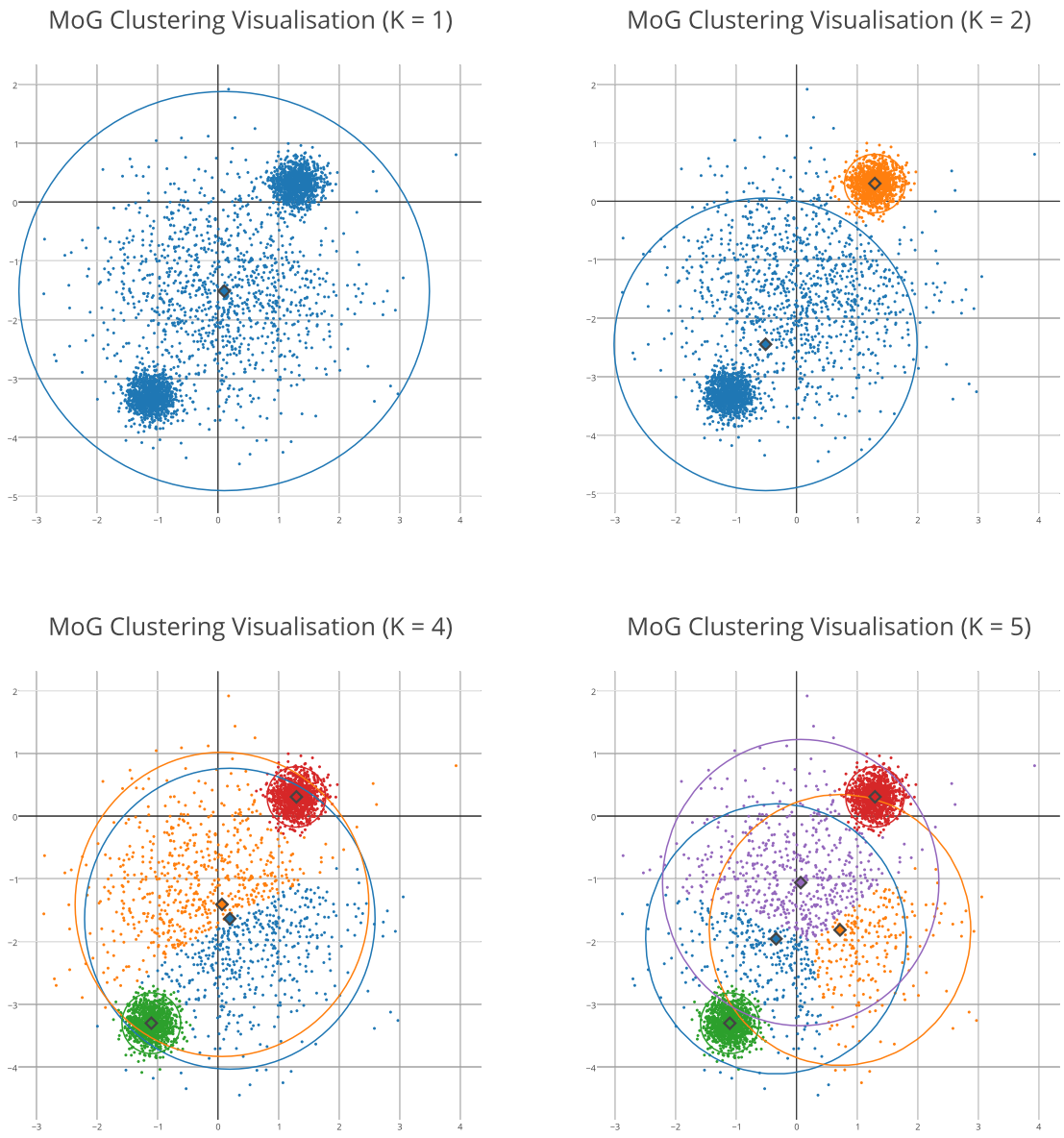

Figure 8: Assignment of validation data points to MoG clusters, for $K = 1, 2, 4, 5$. For each run of K, data points are coloured to their corresponding cluster assignments: { 1, 2, 3, 4, 5 }. The coloured diamonds represent the cluster centers, while the coloured circles represent 95% of the volume under the surfaces of the corresponding multivariate Gaussian distributions.

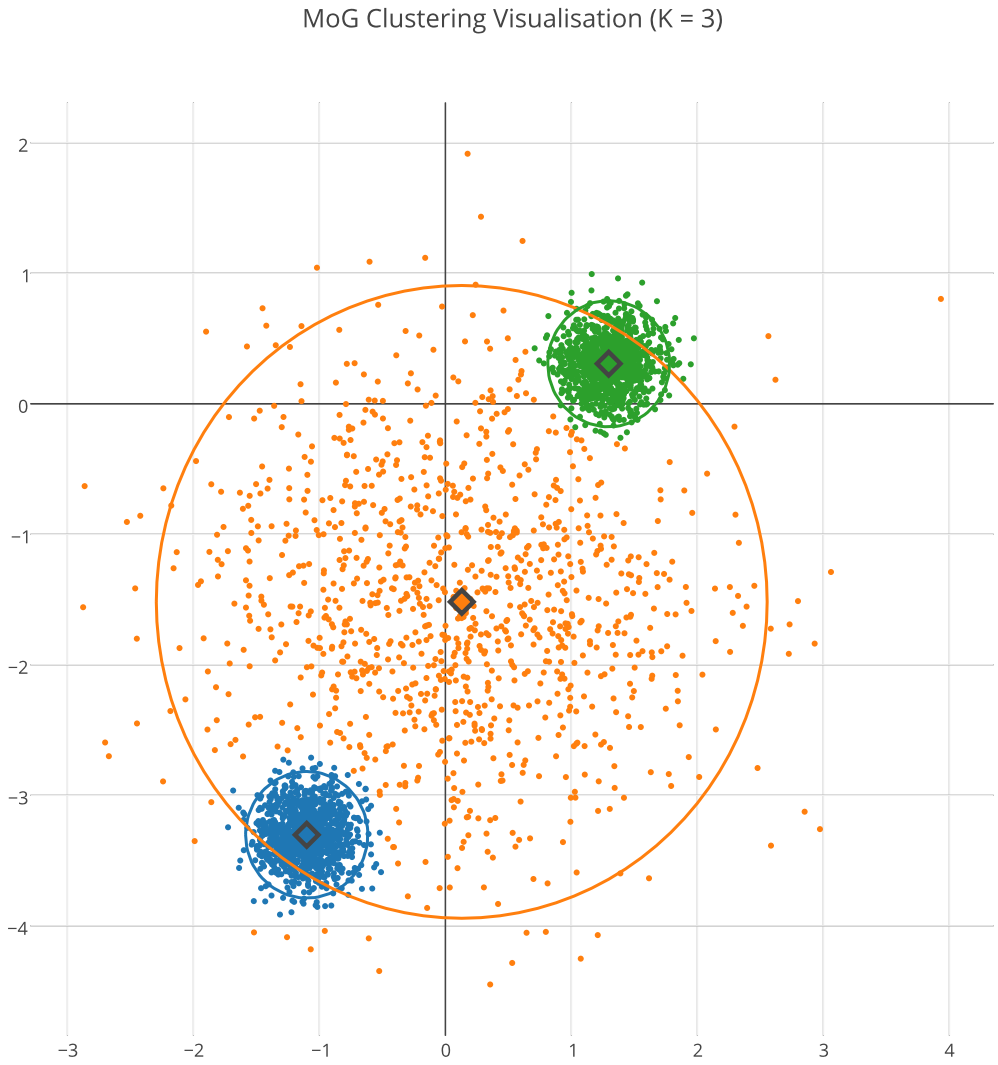

Figure 9: Assignment of validation data points to MoG clusters, for $K = 3$. Data points are coloured to their corresponding cluster assignments: { 1, 2, 3 }. The coloured diamonds represent the cluster centers, while the coloured circles represent 95% of the volume under the surfaces of the corresponding multivariate Gaussian distributions.

Based on the figures and the validation losses, it is clear that the best number of clusters, K, is 3. The scatter plots visually show the data points to be comprised of three clusters. Hence, it would make the most sense to assign 3 *soft* clusters for this clustering assignment. Similarly, from Table 5 and Figure 6, the loss plateaus from $K = 3$ onwards. As such, the most parsimonious model should be chosen to be $K = 3$.

2.2.4 K-means and MoG on *data100D.npy*

K-means and MoG were used to cluster a 100-dimensional data set, *data100D.npy*. Both these algorithms were run with $K = 1, ..., 15$. The validation losses for both these clustering techniques were calculated and summarised in Table 6 and Figure 11.

Table 6: Validation loss (in thousands) of K-means and MoG on *data100.npy*

K	K-means Loss ('000s)	MoG Loss ('000s)
	$\mathcal{L}(\boldsymbol{\mu})$	$-\log P(\mathbf{X})$
1	337	475
2	269	475
3	220	475
4	220	407
5	152	367
6	73	367
7	73	367
8	73	367
9	73	367
10	72	367
11	72	367
12	72	367
13	72	290
14	71	290
15	72	290

zazxx

From Figure 11, the validation loss begins plateauing at $K = 6$ for K-means and $K = 5$ and 13 for MoG. In the interest of model simplicity and accuracy, any one of these three values are potentially suitable as to cluster the data. However, the MoG algorithm does depend on initialization. For example, referring to Figure 10, that came from Soon's implementation, we had a validation loss convergint at around 160K for the 100D data. Based on the intution explained later below, we pick K = 5 to be our best cluster as we may be able to achieve that same validation loss of 160K with K = 5 if we ran enough times with different randomization of initial values for the variables.

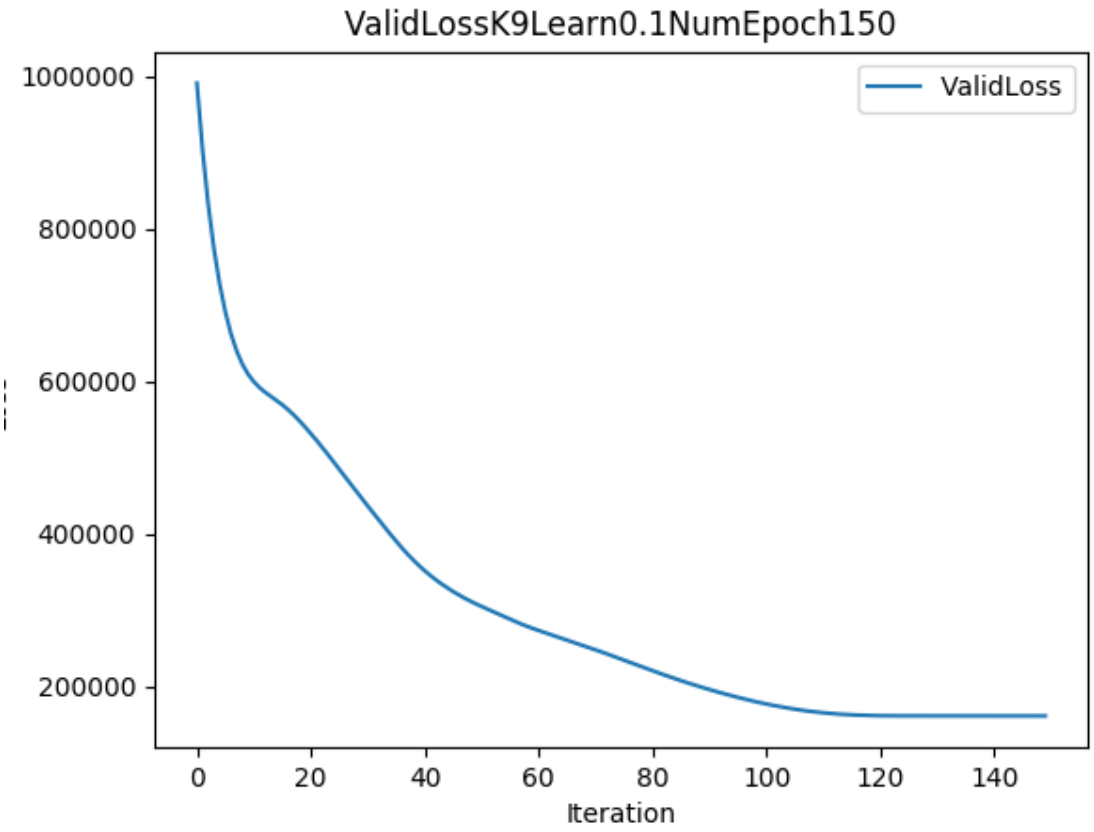

Figure 10: Validation Loss for MOG for converges around 160k.

To gain more intuition of the data set as in the coloured 2D scatter plots from Sections 1.1.2 and 2.2.3, the spatial distribution of *data100D.npy* data points was inspected by projecting the data points to 3 dimensions using the top 3 principal components, using Principal Component Analysis (PCA). The top 3 principal components explained 68.4% of the variance. These projections were done using TensorBoard's Embedding

tab. Snapshots of the spatial distributions are shown in Figure 12.

The projected dataset indicate that there are 5 distinct clusters within the dataset. Using this intuition, and from the choices of $K = 5, 6,$ or 13 from Figure 11, the best number of clusters is chosen to be $K = 5$.

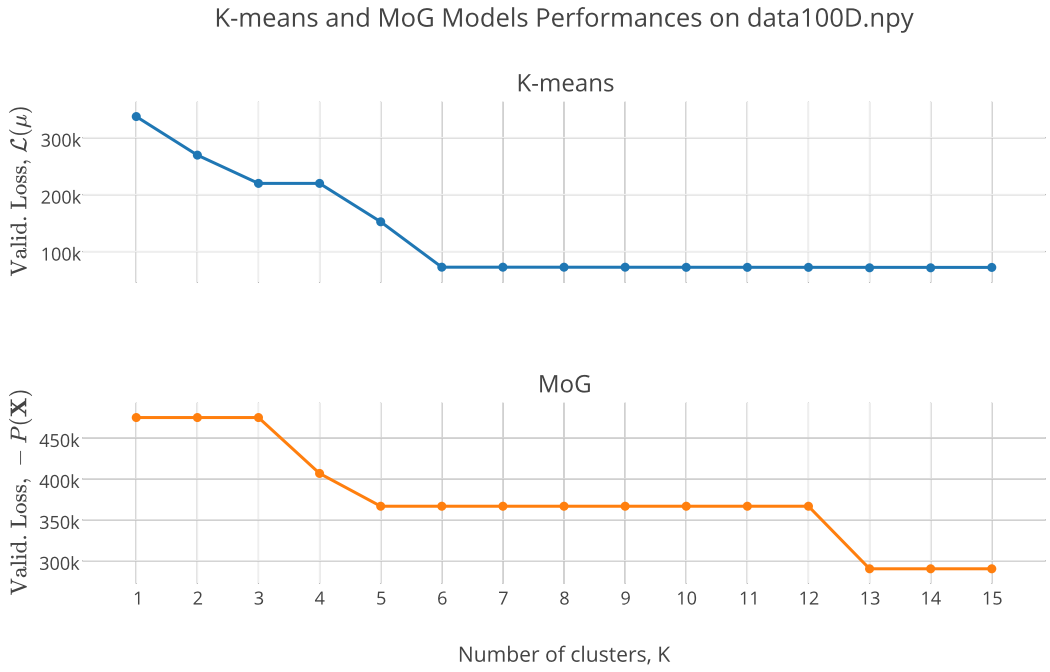

Figure 11: Comparison between K-means and MoG performance on *data100D.npy* for $K = 1, ..., 15$.

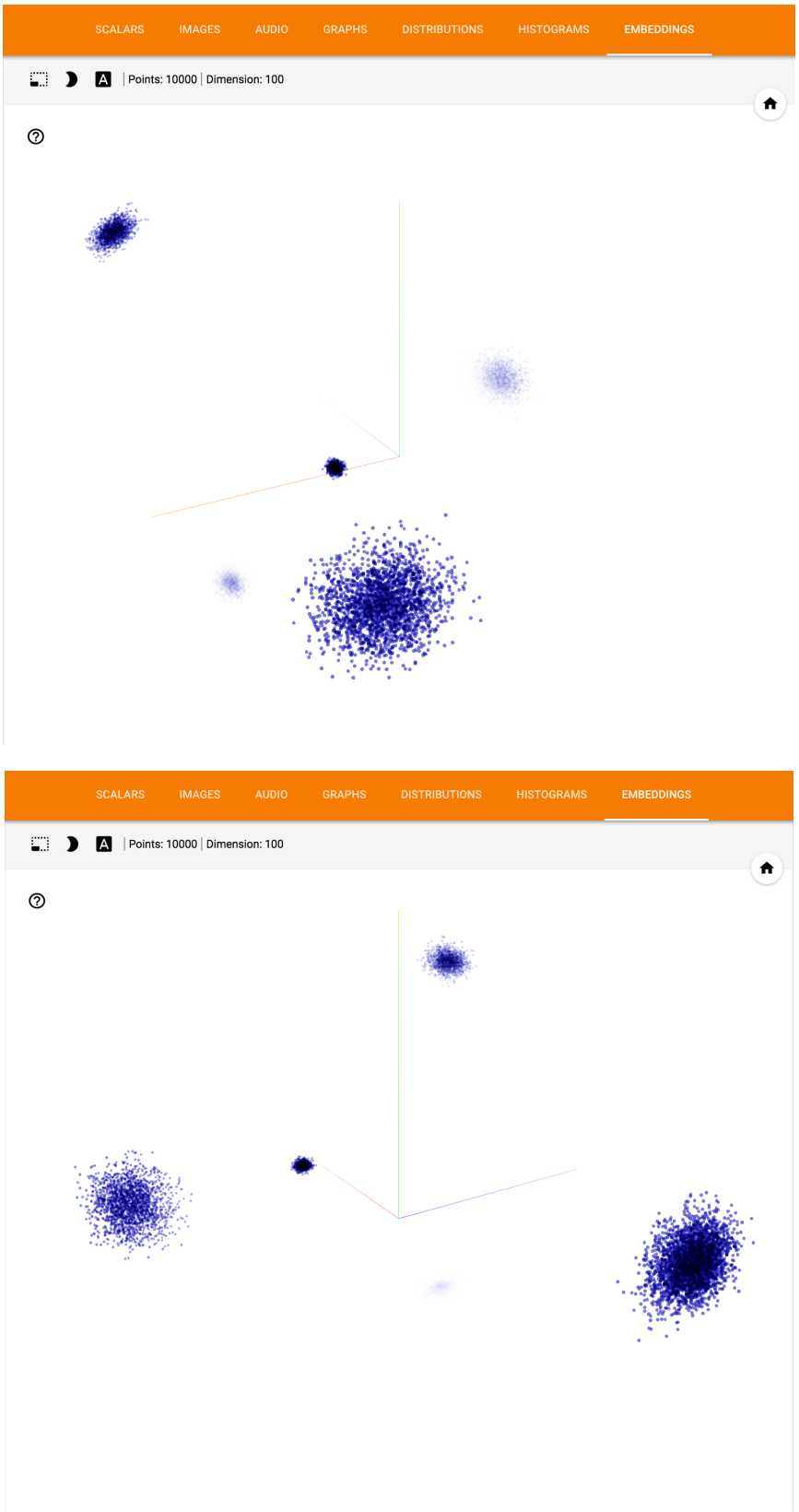

Figure 12: Projected 3D spatial distributions of *data100D.npy* data points using PCA from TensorBoard Embeddings.

3 Discover Latent Dimensions

3.1 Factor Analysis

3.1.1 Deriving the marginal log likelihood for a single training example

Given

$$P(\mathbf{s}_n) = \mathcal{N}(\mathbf{s}_n; \mathbf{0}, \mathbf{I}_K) \tag{16}$$

$$P(\mathbf{x}_n \mid \mathbf{s}_n) = \mathcal{N}(\mathbf{x}_n; \mathbf{W}\mathbf{s}_n + \boldsymbol{\mu}, \boldsymbol{\Psi}) \tag{17}$$

where

$$\mathbf{s}_n \in \mathbb{R}^K \tag{18}$$

$$\mathbf{x}_n, \boldsymbol{\mu} \in \mathbb{R}^D \tag{19}$$

$$\mathbf{W} \in \mathbb{R}^{D \times K} \tag{20}$$

$$\boldsymbol{\Psi} = \begin{bmatrix} \psi_1 & 0 & \cdots & 0 \\ 0 & \psi_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \psi_k \end{bmatrix} \in \mathbb{R}^{D \times D} \tag{21}$$

Quoting Equation 3 from the handout of Assignment 3,

$$P(\mathbf{y}) = \mathcal{N}(\mathbf{y}; \mathbf{A}\boldsymbol{\mu} + \mathbf{b}, \mathbf{L}^{-1} + \mathbf{A}\boldsymbol{\Lambda}^{-1}\mathbf{A}^T) \tag{22}$$

for a single training example,

$$\begin{aligned} \log P(\mathbf{X}) &= \log P(\mathbf{x}) \\ &= \log \mathcal{N}(\mathbf{x}; \mathbf{W} \cdot \mathbf{0} + \boldsymbol{\mu}, \boldsymbol{\Psi} + \mathbf{W}\mathbf{I}_K + \mathbf{W}^T) \\ &= \log \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Psi} + \mathbf{W}\mathbf{W}^T) \end{aligned} \tag{23}$$

Hence, it is proven that the marginal log likelihood of the factor analysis model also follows a Gaussian distribution.

Our handwritten proof is shown in the figure 13. In green is the formulas that are given with their notation, in blue is the corresponding notation that is used in this question.

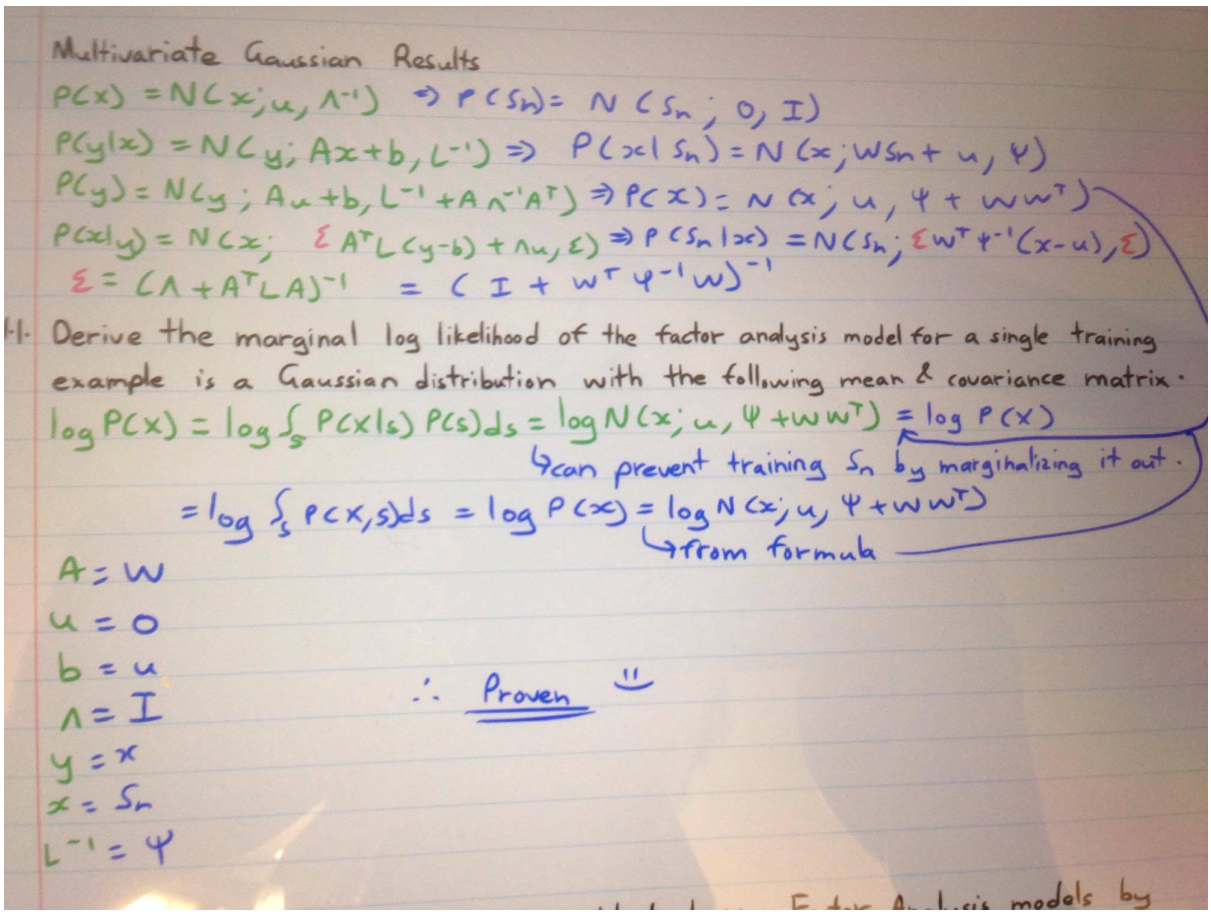

Figure 13: Simple Math Proof using given Formula

3.1.2 Factor Analysis on *tinymnist.npz*

Factor Analysis is carried out on *tinymnist.npz* which contains 8x8 grayscale images of the digits *3* and *5*, using $K = 4$. The parameters trained were $\mathbf{\Psi}$ and $\mathbf{W}$. $\boldsymbol{\mu}$ was not trained as it is by definition the average of each feature in the design matrix, $\mathbf{X}$.

The parameters were trained by maximizing the marginal log-likelihood function $P(\mathbf{X}) = \sum_{n=1}^{B} P(\mathbf{x}_n)$. $\psi_i \quad \forall i = 1, ..., D$ are constrained to be more than zero. Hence, ψ_i were initialised using unconstrained variable ϕ_i as such:

$$\psi_i = \exp(\phi_i) \tag{24}$$

Since the covariance matrix of the probability density function ($\Psi + \mathbf{W}\mathbf{W}^T$) has a determinant not equal to 1, the matrix determinant is implemented in TensorFlow using Cholesky decomposition.

The marginal log likelihood post-training can be found in Table 7.

Table 7: Marginal log likelihoods of factor analysis model post-training on *tinymnist.npz*

Data	$\log P(\mathbf{X})$
Training	8453
Validation	1267
Test	4729

The each trained weights vector in the latent matrix, $\mathbf{W}$ is visualised as 8x8 grayscale heatmap, as depicted in Figure 14. From the figure, Factor Analysis has learnt the following latent dimensions:

Table 8: Latent dimensions found from Factor Analysis of *tinymnist.npz*

Latent Dimensions	Comments
s_1	Captures the variability in the top-right and bottom-half corners of the image. Modelling the variability in the top-right corner is key in distinguishing between *3*s and *5*s.
s_2	Captures the variability in the top-left and lower-right of the image. This image has an *S* shape and can be used to identify *5*s.
s_3	Captures the variability in the top-half of the image. The constrast between the black and white pixels help to identify the upper half of the digit *3*.
s_4	Captures the variability in the bottom-half of the image.

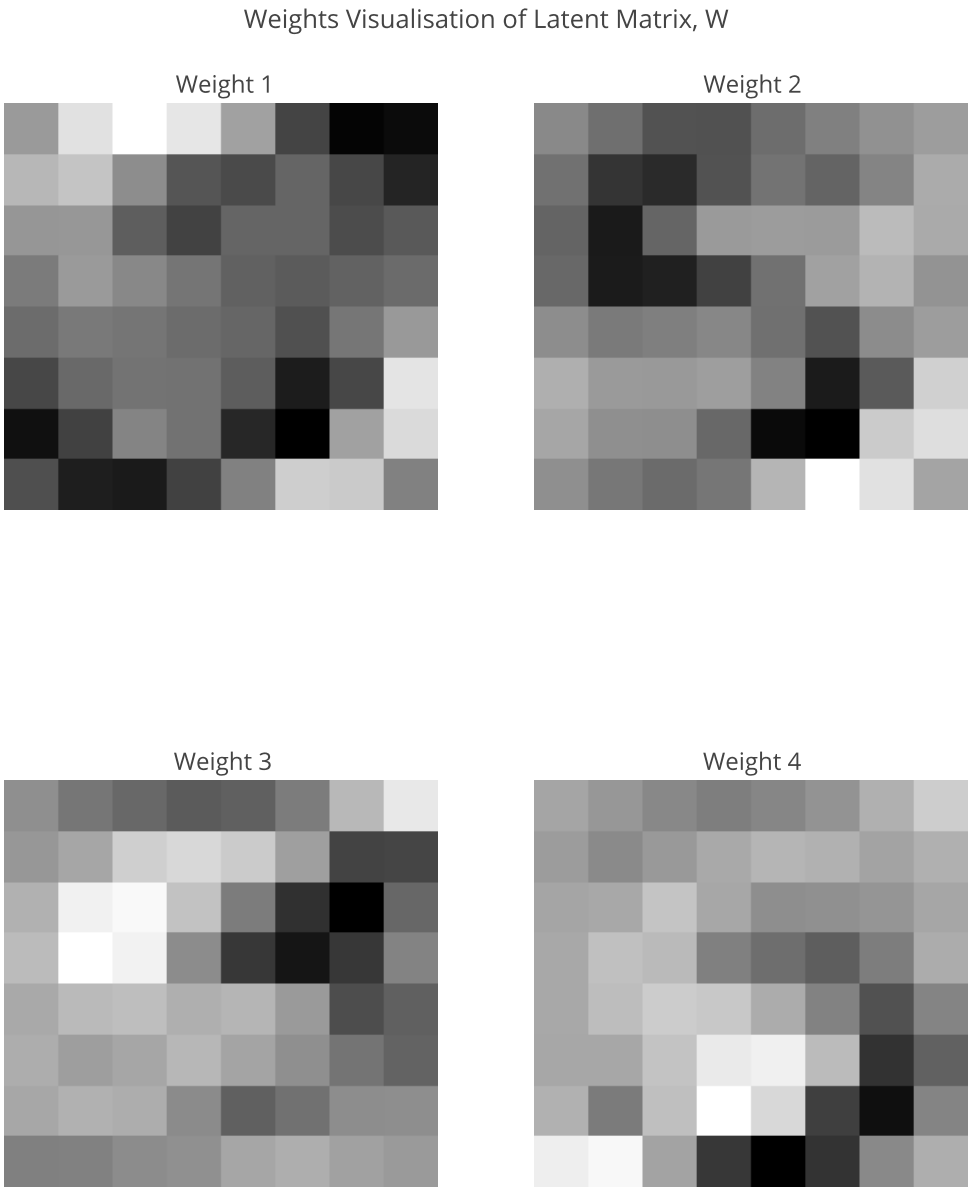

Figure 14: Visualisations of latent matrix, W, for the $K = 4$ latent variables.

3.1.3 Comparison between PCA and FA

A 3D latent variable, $\mathbf{s}$, was sampled from a multivariate Gaussian, $\mathcal{N}(\mathbf{0}, \mathbf{I}_3)$. The variable is then used to generate a data point, $\mathbf{x}$, using the following relations:

$$\begin{aligned} x_1 &= s_1 \\ x_2 &= s_1 + 0.001 s_2 \\ x_3 &= 10 s_3 \end{aligned} \tag{25}$$

This was repeated 200 times to generate a toy dataset consisting of 200 3D points.

For PCA, the largest principal component of this toy dataset was obtained by finding the eigenvector, $\boldsymbol{e}_n$, corresponding to the largest eigenvalue of the dataset covariance matrix, Σ.

$$\boldsymbol{e}_n \textit{ where } n = \arg\max_i \lambda_i \quad \forall \lambda_i \in \sigma(\Sigma) \tag{26}$$

where $\sigma(\Sigma)$ is the spectrum of data covariance matrix.

The largest principal component was found to be $\boldsymbol{e}_n = \begin{bmatrix} 1.21 \times 10^{-6} \\ 1.37 \times 10^{-3} \\ 1.00 \times 10^{0} \end{bmatrix}$.

From this result, it can be seen that the main principal component found using PCA is approximately in the x_3 direction:

$$\begin{aligned} \boldsymbol{e}_n^T \mathbf{x} &= x_3 + 1.212 \times 10^{-6} x_1 + 1.373 \times 10^{-3} x_2 \\ &\approx x_3 \end{aligned} \tag{27}$$

Meanwhile, assuming $K = 1$, Factor Analysis learns the direction of maximum corre-

lation $(x_1 + x_2)$ as shown from the trained $\mathbf{W} = \begin{bmatrix} 0.4798 & 0.4616 & 3.145 \times 10^{-9} \end{bmatrix}$.

$$\begin{aligned} \mathbf{Wx} &= 0.4798x_1 + 0.4616x_2 - 3.145 \times 10^{-9}x_3 \\ &\approx 0.4798x_1 + 0.4616x_2 \\ &= \begin{bmatrix} 0.4798 & 0.4616 \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= \mathbf{k} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \end{aligned} \tag{28}$$

4 Appendices

Both of us implemented the assignment separately to maximise our learnings. The separate implementations can be found below. The graphs come from both separate implementations.

4.1 FuYuan Tee's Implementation

4.1.1 Base Code for K-means

```
# Import relevant packages
import tensorflow as tf
import numpy as np

import time
import datetime

# Non-interactive plotting
import matplotlib.pyplot as plt

# Interactive plotting
from plotly import tools
import plotly.plotly as py
import plotly.graph_objs as go
import plotly.grid_objs as gro
import plotly.offline as pyo
from plotly.offline import download_plotlyjs

# Configure environment
%config InlineBackend.figure_format = 'retina'
np.set_printoptions(precision=3)

# Global Variables
CURRENT_DIR =
 ↪  '/Users/christophertee/Dropbox/University/MASc/Courses/Winter 2017' +
 ↪  \
```

```
              '/ECE521 (Inference Algorithms & Machine
               ↪ Learning)/Assignment 3'
LOG_DIR = '/Logs'

# Activate Plotly Offline for Jupyter
pyo.init_notebook_mode(connected=True)

# Define global variable SEED
SEED = 521

# Load data into memory
"""
data2D.npy contains 10,000 data points of dimension 2
data100D.npy contains 10,000 data points of dimension 100
"""
# Load data
data2D = np.load("./Data/data2D.npy")
data100D = np.load("./Data/data100D.npy")

# Set random seed
np.random.seed(521)

# Generate random index
randIdx2D = np.arange(len(data2D))
randIdx100D = np.arange(len(data100D))

# Randomise data2D
np.random.shuffle(randIdx2D)
data2D = data2D[randIdx2D]

# Randomise data100D
np.random.shuffle(randIdx100D)
data100D = data100D[randIdx100D]

'''
Creates a graph for K-means based on the loss function above:
```

```
60
61  Inputs
62      K:        Number of classifiers
63      data_dim: dimension of datapoint
64
65  X:  data placeholder (N x data_dim)
66  mu: cluster centres (K x data_dim)
67  '''
68  def build_k_means(K, data_dim, device='cpu'):
69
70      # Set TF graph seed
71      tf.set_random_seed(SEED)
72
73      # Define computation device
74      try:
75          assert device == 'cpu' or device == 'gpu'
76      except AssertionError:
77          print 'Invalid device chosen. Please use \'cpu\' or \'gpu\''
78          quit()
79      device = '/' + device + ':0'
80
81      with tf.device('/cpu:0'):
82          # Create placeholder
83          with tf.name_scope('placeholders'):
84              X = tf.placeholder(tf.float32, shape=[None, data_dim],
                    ↪  name='inputs')
85          # Define parameters
86          with tf.variable_scope('parameters'):
87              mu = tf.get_variable('cluster_centres', shape=[K, data_dim],
                    ↪  initializer=tf.random_normal_initializer(seed=SEED))
88
89      with tf.device(device):
90          # Calculate distance matrix (N x K)
91          # by subtracting expanded X (N x D x 1) with expanded mu (1 x D x
                ↪  K) using broadcasting
92          with tf.name_scope('distances'):
```

```
93              dist = tf.reduce_sum(tf.square(tf.expand_dims(X, axis=2) -
                ↪  tf.expand_dims(tf.transpose(mu), axis=0)), \
94                                   axis=1, name='distances')
95
96          # Create responsibility indices to track which datapoint belongs
            ↪  to which cluster
97          with tf.name_scope('responsibility'):
98              _, resp = tf.nn.top_k(-dist, name='responsibility_indices')
99              resp = tf.cast(resp + 1, tf.int64)
100
101         # Calculate loss
102         with tf.name_scope('loss'):
103             loss = tf.reduce_sum(tf.reduce_min(dist, axis=1), name='loss')
104             tf.summary.scalar('loss', loss)
105
106         # Create Adam optimizer
107         with tf.name_scope('optimizer'):
108             optimizer = tf.train.AdamOptimizer(learning_rate=0.01,
                ↪  beta1=0.9, beta2=0.99, epsilon=1e-5).minimize(loss)
109
110     with tf.device('/cpu:0'):
111         # Merge all summaries
112         merged = tf.summary.merge_all()
113
114     return X, mu, resp, loss, optimizer, merged
115
116 '''
117 Run k-means clustering algorithm to cluster datapoints
118 '''
119 def run_k_means(K_list, data_dim, has_valid=False, device='cpu'):
120     '''
121     If has_valid is true, subsets:
122         first 2/3 of data as training data
123         remaining 1/3 of data as validation data
124     '''
125     def subset_data(D):
```

```
126         if D == 2:
127             data = data2D
128         elif D == 100:
129             data = data100D
130         divider = data.shape[0] * 2 / 3
131         return data[:divider], data[divider:]
132 
133     '''
134     Calculate percentage of points belonging to each of K clusters
135     '''
136     def calculate_composition(K, resp_idx):
137         composition = np.array([])
138         for k in range(K):
139             composition = np.append(composition,
                ↪  np.true_divide(np.sum(resp_idx == k + 1),
                ↪  resp_idx.shape[0]))
140         return composition
141 
142     ########################
143     ##  Function begins  ##
144     ########################
145     '''
146     cluster_centres: 11 x K x D
147     resp_idx:        N x 11
148     '''
149 
150     # Assert correct value for data_dim
151     assert data_dim == 2 or data_dim == 100
152 
153     # Define locally global function
154     MAX_ITER = 500
155     CURR_TIME = '{:%b%d %H_%M_%S}'.format(datetime.datetime.now())
156     SUMMARY_DIR = CURRENT_DIR + LOG_DIR + '/K-means/' + CURR_TIME
157 
158     # Create list to store run results
159     results = []
```

```
160
161     for K in K_list:
162         # Clear any pre-defined graph
163         tf.reset_default_graph()
164
165         # Build TensorFlow graph
166         X, mu, resp, loss, optimizer, merged = build_k_means(K, data_dim,
              ↪  device)
167
168         # Select appropriate input_data
169         if has_valid:
170             input_data, valid_data = subset_data(data_dim)
171         else:
172             input_data = data2D if D == 2 else data100D
173
174         # Create arrays to log session losses, cluster_centres and
              ↪  responsbility indices
175         train_loss = np.array([])[:, np.newaxis]
176         if has_valid:
177             valid_loss = np.array([])[:, np.newaxis]
178         cluster_centres = np.array([])[:, np.newaxis,
              ↪  np.newaxis].reshape(0, K, data_dim)
179         resp_idx = np.array([])[:,
              ↪  np.newaxis].reshape(input_data.shape[0], 0)
180
181         # Begin session
182         with tf.Session(config=tf.ConfigProto(allow_soft_placement=True,
              ↪  log_device_placement=False)) as sess:
183             # Log start time
184             start_time = time.time()
185
186             # Create sub-directory title
187             sub_dir = '/K={},dim={},valid={}'.format(K, data_dim,
                  ↪  has_valid)
188
189             # Create summary writers
```

```
190             train_writer = tf.summary.FileWriter(SUMMARY_DIR + sub_dir +
                 ↪  '/train', graph=sess.graph)
191             if has_valid:
192                 valid_writer = tf.summary.FileWriter(SUMMARY_DIR + sub_dir
                     ↪  + '/valid')
193
194             # Initialise all TensorFlow variables
195             tf.global_variables_initializer().run()
196
197             # Define iterator
198             currIter = 0
199
200             # Calculate training (and validation) loss,
201             # cluster centres and responsibility indices before any
                 ↪  training
202             err, summaries, clusters, indices = sess.run([loss, merged,
                 ↪  mu, resp], feed_dict={X:input_data})
203             train_loss = np.append(train_loss, err)
204             train_writer.add_summary(summaries, currIter)
205
206             if has_valid:
207                 err, summaries = sess.run([loss, merged],
                     ↪  feed_dict={X:valid_data})
208                 valid_loss = np.append(valid_loss, err)
209                 valid_writer.add_summary(summaries, currIter)
210
211             if data_dim == 2:
212                 cluster_centres = np.append(cluster_centres,
                     ↪  clusters[np.newaxis, :, :], axis=0)
213                 resp_idx = np.append(resp_idx, indices, axis=1)
214
215             # Begin training
216             while currIter < MAX_ITER:
217                 # Train graph
218                 _, err, summaries = sess.run([optimizer, loss, merged],
                     ↪  feed_dict={X:input_data})
```

```
219
220             # Add training loss
221             train_writer.add_summary(summaries, currIter + 1)
222             train_loss = np.append(train_loss, err)
223
224             # Log validation loss
225             if has_valid:
226                 err, summaries = sess.run([loss, merged],
                      ↪  feed_dict={X:valid_data})
227                 valid_loss = np.append(valid_loss, err)
228                 valid_writer.add_summary(summaries, currIter)
229
230             # Log responsibility indices and cluster centres every 10%
                  ↪  of maximum iteration
231             if ((float(currIter) + 1) * 100 / MAX_ITER) % 10 == 0:
232                 clusters, indices = sess.run([mu, resp],
                      ↪  feed_dict={X:input_data})
233
234                 cluster_centres = np.append(cluster_centres,
                      ↪  clusters[np.newaxis, :, :], axis=0)
235                 resp_idx = np.append(resp_idx, indices, axis=1)
236
237             # Post training progress to user, every 100 iterations
238             if currIter % 100 == 99:
239                 if not has_valid:
240                     print 'iter: {:3d}, train_loss:
                          ↪  {:3.1f}'.format(currIter,
                          ↪  train_loss[currIter])
241                 else:
242                     print 'iter: {:3d}, train_loss: {:3.1f},
                          ↪  valid_loss: {:3.1f}'\
243                           .format(currIter + 1,
                                ↪  train_loss[currIter],
                                ↪  valid_loss[currIter])
244
245             currIter += 1
```

```
246
247             # End of while loop
248             print 'Max iteration reached'
249             train_writer.close()
250             if has_valid:
251                 valid_writer.close()
252
253             # Calculate composition of points belonging to each cluster
254             composition = calculate_composition(K, resp_idx[:, -1])
255
256             if not has_valid:
257                 results.append(
258                     {
259                         'K': K,
260                         'train_loss': train_loss,
261                         'cluster_centres': cluster_centres,
262                         'responsibility_indices': resp_idx.astype(int),
263                         'composition': composition,
264                         'time_of_run': '{:%b%d
                              ↪  %H_%M_%S}'.format(datetime.datetime.now())
265                     }
266                 )
267             else:
268                 results.append(
269                 {
270                     'K': K,
271                     'train_loss': train_loss,
272                     'valid_loss': valid_loss,
273                     'cluster_centres': cluster_centres,
274                     'responsibility_indices': resp_idx.astype(int),
275                     'composition': composition,
276                     'time_of_run': '{:%b%d
                          ↪  %H_%M_%S}'.format(datetime.datetime.now())
277                 }
278             )
279
```

```
            # TODO calculate convergence
            if not has_valid:
                print 'K: {:3d}, train loss: {:3.1f}, duration:
                  ↪  {:3.1f}s\n'\
                        .format(K, train_loss[-1], time.time() -
                          ↪  start_time)
            else:
                print 'K: {:3d}, train loss: {:3.1f}, valid loss: {:3.1f},
                  ↪  duration: {:3.1f}s\n'\
                        .format(K, train_loss[-1], valid_loss[-1],
                          ↪  time.time() - start_time)

    print 'RUN COMPLETED'
    return results
```

4.1.2 K-means - Q1.1.2

```
# Run K-means
results_1_1_2 = run_k_means(K_list=[3], data_dim=2)
```

4.1.3 K-means - Q1.1.3

```
# Run K-means
results_1_1_3 = run_k_means(K_list=[1, 2, 3, 4, 5], data_dim=2)

# Save results
np.save('./Results/K-means/1_1_3.npy', results_1_1_3)

# Plot losses
def loss_IGraph(loss):
    # Define data to plot
    trace = go.Scatter(
        x = range(loss.shape[0]),
        y = loss
    )
    data = go.Data([trace])

    # Define layout
    layout = go.Layout(
        title = '$\\mathcal{L}({\\mathbf{\\mu}}) \\text{ vs. Number of
        ↪ Updates}$',
        xaxis = {'title': 'Updates'},
        yaxis = {'title': 'Loss'}
    )

    # Define figure
    figure = go.Figure(data=data, layout=layout)

    # Generate plot
    py.iplot(figure, filename='/ECE521: A3/Q1: K-means/Q1.2_k_means_loss',
    ↪ sharing='private')
    return pyo.iplot(figure)

# Generate loss function graph
figure = loss_IGraph(results_1_1_3[2]['train_loss'])

```

```
# Cluster Visualisation
'''
Colour data points by clusters generated by K-means algorithm
Input:
    cluster_centres: coordinates of cluster centres (K x D)
    resp_idxes:      final responsibility indices for each run of K (N x
     ↪  num_subplots)
'''
def visualise_k_means_clusters(result):
    # Store cluster centres and responsibility indices

    cluster_centres = result['cluster_centres'][-1,:,:]
    resp_idx = result['responsibility_indices'][:,-1]

    # Define colour list as per Plotly's default colour list
    colour_list = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd',
     ↪  '#8c564b']

    # Create traces for each cluster
    traces = []
    for k in range(np.amax(resp_idx)):
        # Create trace for data points in cluster k
        traces.append(go.Scatter(
            x = data2D[resp_idx == k + 1][:,0],
            y = data2D[resp_idx == k + 1][:,1],
            hoverinfo = 'none',
            mode = 'markers',
            marker = {
                'size': 4,
                'color': colour_list[k],
            }
        ))

        # Create trace for cluster centre k
        traces.append(go.Scatter(
            x = [cluster_centres[k][0]],
```

```
            y = [cluster_centres[k][1]],
            name = 'Cluster {}'.format(k + 1),
            mode = 'markers',
            marker = {
                'size': 15,
                'symbol': 'diamond',
                'color': '#000000'
            }
        ))

    # Add traces
    traces = go.Data(traces)

    # Generate figure layout
    layout = go.Layout(
        height = 800,
        showlegend = False,
        title = 'Clusters Visualization (K = {})'.format(result['K']),
        xaxis = {'title': 'x'},
        yaxis = {'title': 'y'}
    )

    # Generate figure
    figure = go.Figure(data=traces, layout=layout)

    py.iplot(figure, filename='/ECE521: A3/Q1:
      ↪  K-means/Q1.3_cluster_viz_K={}'.format(result['K']),
      ↪  sharing='private')
    return pyo.iplot(figure)
```

4.1.4 K-means - Q1.1.4

```
# Run K-means
results_1_1_4 = run_k_means(K_list=[1, 2, 3, 4, 5], data_dim=2,
 ↪  has_valid=True)

# Save results
np.save('./Results/K-means/1_1_4.npy', results_1_1_4)

# Generate bar chart
'''
Generate a bar chart for each model showing percentage of data points
 ↪  belong to each cluster
'''
def cluster_assignment_IGraph(results):
    # Define colour list as per Plotly's default colour list
    colour_list = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd',
     ↪  '#8c564b']

    # Define empty figure
    figure = {
        'data': [],
        'layout': {}
    }

    # Define data to plot
    for i, result in enumerate(results):
        for k in range(result['K']):
            trace = go.Bar(
                x = [i + 1],
                y = [result['composition'][k]],
                marker = {'color': colour_list[k]},
                name = 'Cluster {}'.format(k + 1)
            )
            figure['data'].append(trace)

```

```
    # Define layout
    figure['layout'] = {
        'title': 'Percentage of data points assigned to each cluster',
        'xaxis': {'title': 'Number of clusters, K'},
        'yaxis': {'title': 'Assignment to cluster, %'},
        'barmode': 'stack',
        'showlegend': False
    }

    # Generate plot
    py.iplot(figure, filename='/ECE521: A3/Q1:
      ↪ K-means/Q1.3_assignment_bar_chart', sharing='private')
    return pyo.iplot(figure)

# Generate loss function graph
figure = cluster_assignment_IGraph(results_1_1_3)
```

4.1.5 Base Code for MoG

```
# Import relevant packages
import tensorflow as tf
import numpy as np

import time
import datetime

# Non-interactive plotting
import matplotlib.pyplot as plt

# Interactive plotting
from plotly import tools
import plotly.plotly as py
import plotly.graph_objs as go
import plotly.grid_objs as gro
import plotly.offline as pyo
from plotly.offline import download_plotlyjs

# Configure environment
%config InlineBackend.figure_format = 'retina'
np.set_printoptions(precision=3)

# Global Variables
CURRENT_DIR =
 ↪  '/Users/christophertee/Dropbox/University/MASc/Courses/Winter 2017' +
 ↪  \
              '/ECE521 (Inference Algorithms & Machine
                ↪  Learning)/Assignment 3'
LOG_DIR = '/Logs'

# Activate Plotly Offline for Jupyter
pyo.init_notebook_mode(connected=True)

# Define global variable SEED
```

```
SEED = 521

# Load data into memory
"""
data2D.npy contains 10,000 data points of dimension 2
data100D.npy contains 10,000 data points of dimension 100
"""
# Load data
data2D = np.load("./Data/data2D.npy")
data100D = np.load("./Data/data100D.npy")

# Set random seed
np.random.seed(521)

# Generate random index
randIdx2D = np.arange(len(data2D))
randIdx100D = np.arange(len(data100D))

# Randomise data2D
np.random.shuffle(randIdx2D)
data2D = data2D[randIdx2D]

# Randomise data100D
np.random.shuffle(randIdx100D)
data100D = data100D[randIdx100D]

'''
Builds TensorFlow graph for MoG

Input:
    K: number of clusters
    D: dimension of data (only 2 or 100 allowed)
Internal variables:
    X: input data matrix (N x D)
    Mu: cluster centres (K x D)
    sigma_sq: cluster variance (K x 1)
```

```
    log_pi: log of latent cluster variables (K x 1)
'''
def build_MoG(K, D, device='cpu'):
    '''
    Calculate log probability density function for all pairs of B data points and K clusters

    Assumptions:
        Dimensions are independent and have the same standard deviation, sigma
    Output:
        log PDF function (N x K)
    '''
    def calc_log_gaussian_cluster_k(X, Mu, sigma_sq):
        with tf.name_scope('log_gaussian_cluster'):
            # Infer dimension of data
            D = tf.shape(X)[1]

            # Calculate Mahalanobis distance
            ### Expand dim(X) to (N x 1 x D)
            ### Expand dim(Mu) to (1 x K x D)
            ### Reduce sum along the D-axis
            with tf.name_scope('mahalanobis_dist'):
                dist = -
                    tf.divide(tf.reduce_sum(tf.square(tf.expand_dims(X,
                    axis=1) - tf.expand_dims(Mu, axis=0)), axis=2), 2 *
                    tf.transpose(sigma_sq), name='mahalanobis_dist')

            # Calculate log of gaussian constant term
            ### Transpose sigma_sq to (1 x K)
            with tf.name_scope('log_gauss_const'):
                log_gauss_const = - tf.multiply(tf.cast(D, tf.float32) /
                    2, tf.log(2 * np.pi * tf.transpose(sigma_sq)),
                    name='log_gauss_const')

            # Sum results
```

```
            log_gaussian_cluster = tf.add(dist, log_gauss_const,
             ↪  name='log_gauss_cluster')

        return log_gaussian_cluster

    '''
    Calculate log probability cluster variable z given x, a.k.a.
↪   conditional responsibilities, gamma

    Output:
        conditional responsibilities (N x K)
    '''
    def calc_log_conditional_responsibilities(X, Mu, sigma_sq, log_pi):
        with tf.name_scope('log_conditional_responsibilities'):
            # Calculate unnormalised_log_posterior P(z|x)
            with tf.name_scope('unnormalised_log_posterior'):
                unnormalised_log_posterior =
                 ↪  calc_log_gaussian_cluster_k(X, Mu, sigma_sq) +
                 ↪  tf.transpose(log_pi)

            # Return log normalised posterior / conditional
             ↪  responsibilities
            with tf.name_scope('log_gamma_z'):
                cond_resp = tf.add(-
                 ↪  tf.reduce_logsumexp(unnormalised_log_posterior,
                 ↪  axis=1, keep_dims=True),\
                                    unnormalised_log_posterior, \
                                    name='log_gamma_z')
        return cond_resp

    '''
    Calculates the negative log marginal probability, -log P(X), aka the
↪   loss function for MoG

    Output:
        - log P(X)  (scalar)
```

```
    '''
    def calc_neg_log_marg_prob(X, Mu, sigma_sq, log_pi):
        with tf.name_scope('loss'):
            loss =
              ↪  tf.negative(tf.reduce_sum(tf.reduce_logsumexp(calc_log_gaussian_cluste
              ↪  Mu, sigma_sq) + tf.transpose(log_pi), axis=1), axis=0),
              ↪  name='-log_P_X')
        return loss

    '''
    Helper function to add histogram tag to variables
    Input:
        var: variable to be tagged with histogram summary
    '''
    def _add_histogram(vars_):
        for var in vars_:
            tf.summary.histogram(var.op.name, var)

    ########################
    ##  Function begins  ##
    ########################

    # Fix TF graph seed
    tf.set_random_seed(SEED)

    # Define computation device
    try:
        assert device == 'cpu' or device == 'gpu'
    except AssertionError:
        print 'Invalid device chosen. Please use \'cpu\' or \'gpu\''
        quit()
    device = '/' + device + ':0'

    with tf.device('/cpu:0'):
        # Define placeholder
        with tf.name_scope('placeholder'):
```

```
158             X = tf.placeholder(tf.float32, shape=[None, D], name='inputs')
159
160         # Define parameters
161         with tf.variable_scope('parameters'):
162             Mu = tf.get_variable('cluster_centres', shape=[K, D],
                  ↪  initializer=tf.truncated_normal_initializer(seed=SEED))
163             phi = tf.get_variable('latent_for_sigma_sq', shape=[K, 1],
                  ↪  initializer=tf.truncated_normal_initializer(seed=SEED))
164             psi = tf.get_variable('latent_for_pi', shape=[K, 1],
                  ↪  initializer=tf.truncated_normal_initializer(seed=SEED +
                  ↪  1))
165
166             # Calculate bounded variables sigma_sq and pi
167             sigma_sq = tf.exp(phi, name='sigma_sq')
168
169             with tf.name_scope('log_pi'):
170                 log_pi =
                      ↪  tf.transpose(tf.nn.log_softmax(tf.transpose(psi)),
                      ↪  name='log_pi')
171
172     with tf.device(device):
173         # Calculate conditional responsibilities
174         log_resp = calc_log_conditional_responsibilities(X, Mu, sigma_sq,
              ↪  log_pi)
175
176         # Calculate loss
177         loss = calc_neg_log_marg_prob(X, Mu, sigma_sq, log_pi)
178         tf.summary.scalar('loss', loss)
179
180         # Define optimizer
181         optimizer = tf.train.AdamOptimizer(learning_rate=0.01, \
182                                            beta1=0.9, beta2=0.99,
                                                ↪  epsilon=1e-5).minimize(loss)
183
184     with tf.device('/cpu:0'):
185         # Add histogram summaries for variables of interest
```

```
        _add_histogram([Mu, phi, psi, sigma_sq, log_pi, log_resp])

        # Merge all summaries
        merged = tf.summary.merge_all()

    return X, Mu, sigma_sq, log_pi, log_resp, loss, optimizer, merged

'''
Runs MoG training algorithm more efficiently by not saving loss values.
    Tensorboard embedding enabled
'''
def run_MoG_v2(K_list, D, QUES_DIR, has_valid=False, device='cpu'):
    '''
    If has_valid is true, subsets:
        first 2/3 of data as training data
        remaining 1/3 of data as validation data
    '''
    def subset_data(D):
        if D == 2:
            data = data2D
        elif D == 100:
            data = data100D
        divider = data.shape[0] * 2 / 3
        return data[:divider], data[divider:]

    '''
    Embed data for visualization purposes
    '''
    def embed_data(D, train_writer):
        # Define input data
        input_data = data2D if D == 2 else data100D
        input_data_name = 'data{}D.npy'.format(D)

        # Create variable to embed
        data_to_embed = tf.Variable(input_data, name=input_data_name,
         ↪  trainable=False, collections=[])
```

```
221
222         # Define projector configurations
223         config = projector.ProjectorConfig()
224
225         # Add embedding
226         embedding = config.embeddings.add()
227
228         # Connect tf.Variable to embedding
229         embedding.tensor_name = data_to_embed.name
230
231         # Evaluate tf.Variable
232         sess.run(data_to_embed.initializer)
233
234         # Create save checkpoint
235         saver = tf.train.Saver([data_to_embed])
236         saver.save(sess, SUMMARY_DIR + sub_dir + '/train/model.ckpt',
              ↪  MAX_ITER)
237
238         # Write projector_config.pbtxt in LOG_DIR
239         projector.visualize_embeddings(train_writer, config)
240
241     ######################
242     ##  Function begins  ##
243     ######################
244     '''
245     cluster_centres: 11 x K x D
246     train_resp:        11 x N x K
247     '''
248
249     # Assert correct value for D
250     assert D == 2 or D == 100
251
252     # Define locally global function
253     MAX_ITER = 1500
254     CURR_TIME = '{:%b%d %H_%M_%S}'.format(datetime.datetime.now())
255     SUMMARY_DIR = CURRENT_DIR + LOG_DIR + '/MoG' + QUES_DIR + '/' +
          ↪  CURR_TIME
```

```
256
257     # Create list to store run results
258     results = []
259
260     for K in K_list:
261         # Clear any pre-defined graph
262         tf.reset_default_graph()
263
264         # Build TensorFlow graph
265         X, Mu, sigma_sq, log_pi, log_resp, loss, optimizer, merged =
             ↪  build_MoG(K, D, device)
266
267         # Select appropriate input_data
268         if has_valid:
269             input_data, valid_data = subset_data(D)
270         else:
271             input_data = data2D if D == 2 else data100D
272
273         # Create arrays to log cluster_centres, cluster_variances, pi's,
             ↪  and responsbility indices
274         train_loss = np.array([])[:, np.newaxis]
275         if has_valid:
276             valid_loss = np.array([])[:, np.newaxis]
277             valid_resp = np.array([])[:, np.newaxis,
                 ↪  np.newaxis].reshape(0, valid_data.shape[0], K)
278         cluster_centres = np.array([])[:, np.newaxis,
             ↪  np.newaxis].reshape(0, K, D)
279         cluster_variances = np.array([])[:, np.newaxis].reshape(0, K)
280         cluster_pi = np.array([])[:, np.newaxis].reshape(0, K)
281         train_resp = np.array([])[:, np.newaxis, np.newaxis].reshape(0,
             ↪  input_data.shape[0], K)
282
283         # Begin session
284         with tf.Session(config=tf.ConfigProto(allow_soft_placement=True,
             ↪  log_device_placement=False)) as sess:
285             # Log start time
```

```
286             start_time = time.time()
287
288             # Create sub-directory title
289             sub_dir = '/K={},D={},valid={}'.format(K, D, has_valid)
290
291             # Create summary writers
292             train_writer = tf.summary.FileWriter(SUMMARY_DIR + sub_dir +
                 ↪  '/train', graph=sess.graph)
293             if has_valid:
294                 valid_writer = tf.summary.FileWriter(SUMMARY_DIR + sub_dir
                     ↪  + '/valid')
295
296             # Initialise all TensorFlow variables
297             tf.global_variables_initializer().run()
298
299             # Define iterator
300             curr_iter = 0
301
302             # Calculate training (and validation) loss,
303             # cluster centres and responsibility indices before any
                 ↪  training
304             err, summaries, clusters, variances, log_prior_pi,
                 ↪  log_train_indices = sess.run([loss, merged, Mu, sigma_sq,
                 ↪  log_pi, log_resp], feed_dict={X:input_data})
305             train_loss = np.append(train_loss, err)
306             train_writer.add_summary(summaries, curr_iter)
307
308             # Log clusters and responsibility indices
309             cluster_centres = np.append(cluster_centres,
                 ↪  clusters[np.newaxis,:,:], axis=0)
310             cluster_variances = np.append(cluster_variances,
                 ↪  np.transpose(variances), axis=0)
311             cluster_pi = np.append(cluster_pi,
                 ↪  np.exp(np.transpose(log_prior_pi)), axis=0)
312
313             train_resp = np.append(train_resp,
                 ↪  np.exp(log_train_indices)[np.newaxis,:,:], axis=0)
```

```
314
315             # Log validation data
316             if has_valid:
317                 err, log_valid_indices, summaries  = sess.run([loss,
                      ↪  log_resp, merged], feed_dict={X:valid_data})
318
319                 valid_loss = np.append(valid_loss, err)
320                 valid_resp = np.append(valid_resp,
                      ↪  np.exp(log_valid_indices)[np.newaxis, :, :], axis=0)
321                 valid_writer.add_summary(summaries, curr_iter)
322
323             # Begin training
324             while curr_iter < MAX_ITER:
325                 # Train graph
326                 _, summaries, err = sess.run([optimizer, merged, loss],
                      ↪  feed_dict={X:input_data})
327
328                 # Add training loss
329                 train_loss = np.append(train_loss, err)
330                 train_writer.add_summary(summaries, curr_iter + 1)
331
332                 # Log validation loss
333                 if has_valid:
334                     summaries, err = sess.run([merged, loss],
                          ↪  feed_dict={X:valid_data})
335
336                     valid_loss = np.append(valid_loss, err)
337                     valid_writer.add_summary(summaries, curr_iter)
338
339                 # Log responsibility indices and cluster centres every 10%
                      ↪  of maximum iteration
340                 if ((float(curr_iter) + 1) * 100 / MAX_ITER) % 10 == 0:
341                     clusters, variances, log_prior_pi, log_train_indices =
                          ↪  sess.run([Mu, sigma_sq, log_pi, log_resp],
                          ↪  feed_dict={X:input_data})
342
```

```
343                     cluster_centres = np.append(cluster_centres,
                          ↪  clusters[np.newaxis, :, :], axis=0)
344                     cluster_variances = np.append(cluster_variances,
                          ↪  np.transpose(variances), axis=0)
345                     cluster_pi = np.append(cluster_pi,
                          ↪  np.exp(np.transpose(log_prior_pi)), axis=0)
346
347                     train_resp = np.append(train_resp,
                          ↪  np.exp(log_train_indices)[np.newaxis,:,:],
                          ↪  axis=0)
348
349                     if has_valid:
350                         log_valid_indices = sess.run(log_resp,
                              ↪  feed_dict={X:valid_data})
351                         valid_resp = np.append(valid_resp,
                              ↪  np.exp(log_valid_indices)[np.newaxis, :, :],
                              ↪  axis=0)
352
353                 # Post training progress to user, every 100 iterations
354                 if curr_iter % 100 == 99:
355                     print 'iter: {:3d}'.format(curr_iter + 1)
356
357                 curr_iter += 1
358
359             # End of while loop
360             print 'Max iteration reached'
361
362             # Embed data
363             embed_data(D, train_writer)
364
365             # Close writers
366             train_writer.close()
367             if has_valid:
368                 valid_writer.close()
369
370             if not has_valid:
```

```
                results.append(
                    {
                        'K': K,
                        'train_loss': train_loss,
                        'cluster_centres': cluster_centres,
                        'cluster_variances': cluster_variances,
                        'cluster_pi': cluster_pi,
                        'train_resp': train_resp,
                        'time_of_run': '{:%b%d
                          ↪  %H_%M_%S}'.format(datetime.datetime.now())
                    }
                )
            else:
                results.append(
                {
                    'K': K,
                    'train_loss': train_loss,
                    'valid_loss': valid_loss,
                    'cluster_centres': cluster_centres,
                    'cluster_variances': cluster_variances,
                    'cluster_pi': cluster_pi,
                    'train_resp': train_resp,
                    'valid_resp': valid_resp,
                    'time_of_run': '{:%b%d
                      ↪  %H_%M_%S}'.format(datetime.datetime.now())
                }
            )

            # TODO calculate convergence
            print 'K: {:3d}, duration: {:3.1f}s\n'.format(K, time.time() -
              ↪  start_time)

    print 'RUN COMPLETED'
    return results
```

4.1.6 MoG - Q2.2.2

```
# Run MoG
results_2_2_2 = run_MoG_v2(K_list=[3], D=2, QUES_DIR='/Q2.2.2')

# Save results
np.save('./Results/MoG/2_2_2.npy', results_2_2_2)

# Plot loss graph
def loss_IGraph(loss):
    # Define data to plot
    trace = go.Scatter(
        x = range(loss.shape[0]),
        y = loss
    )
    data = go.Data([trace])

    # Define layout
    layout = go.Layout(
        title = '$-\\log P(\mathbf{X}) \\text{ vs. Number of Updates}$',
        xaxis = {'title': 'Updates'},
        yaxis = {'title': 'Loss'}
    )

    # Define figure
    figure = go.Figure(data=data, layout=layout)

    # Generate plot
    py.iplot(figure, filename='/ECE521: A3/Q2: Mixture of
     ↪ Gaussians/Q2.2_MoG_loss', sharing='private')
    return pyo.iplot(figure)

# Generate loss function graph
figure = loss_IGraph(results_2_2_2[0]['train_loss'])
```

4.1.7 MoG - Q2.2.3

```
# Run MoG
results_2_2_3 = run_MoG_v2(K_list=[1, 2, 3, 4, 5], D=2,
 ↪  QUES_DIR='/Q2.2.3', has_valid=True)

# Save results
np.save('./Results/MoG/2_2_3.npy', results_2_2_3)

# Plot graph of performance comparison
def IGraph_2_2_3(results):
    valid_loss = [result['valid_loss'][-1] for result in results]

    figure = {
        'data': [],
        'layout': {}
    }

    figure['data'].append({
        'x': [k + 1 for k in range(5)],
        'y': valid_loss,
    })

    figure['layout'] = {
        'title': 'MoG Model Performances on data2D.npy',
        'showlegend': False,
        'xaxis': {'title': 'K', 'dtick': 1},
        'yaxis': {'title': 'Final Validation Loss'}
    }

    py.iplot(figure, \
             filename='/ECE521: A3/Q2: Mixture of
              ↪  Gaussians/Q2.3_compare_MoG_clusters',\
             sharing='private')
    return pyo.iplot(figure)

```

```
IGraph_2_2_3(results_2_2_3)

# Generate cluster assignment bar chart
'''
Generate a bar chart for each model showing percentage of data points belong to each cluster
'''
def cluster_assignment_IGraph(results, is_MoG, D, question_name):
    assert D == 2 or D == 100

    # Define colour list as per Plotly's default colour list
    colour_list = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b']

    # Define empty figure
    figure = {
        'data': [],
        'layout': {}
    }

    # Define data to plot
    for i, result in enumerate(results):
        for k in range(result['K']):
            trace = go.Bar(
                x = [i + 1],
                y = [result['cluster_pi'][-1][k]] if is_MoG == True else [result['composition'][k]],
                marker = {'color': colour_list[k]},
                name = 'Cluster {}'.format(k + 1)
            )
            figure['data'].append(trace)

    # Define layout
    figure['layout'] = {
        'title': 'Percentage of data points assigned to each {} cluster on data{}D.npy'\
```

```
            .format('MoG' if is_MoG == True else 'K-means', D),
        'xaxis': {'title': 'Number of clusters, K'},
        'yaxis': {'title': 'Assignment to cluster, %'},
        'barmode': 'stack',
        'showlegend': False
    }

    # Generate plot
    py.iplot(figure, \
             filename='/ECE521: A3/Q2: Mixture of Gaussians/Q{}_assignment_bar_chart_{}D_{}'\
                .format(question_name, D, 'MoG' if is_MoG is True else 'K-means'),\
             sharing='private')

    return pyo.iplot(figure)

cluster_assignment_IGraph(results_2_2_3, is_MoG=True, D=2, question_name='2.3')

# Visualising MoG clusters
'''
Final result by colouring data points by clusters generated by Mixture of Gaussian algorithm
Input:
    result:           MoG training result with validation
Notes:
    cluster_centres:  coordinates of cluster centres (K x D)
    cluter_variances: cluster variances (K)
    train_resp:       training responsibility indices for each run of K ((N*2/3) x K)
    valid_resp:       validation responsibility indices for each run of K ((N/3) x K)
'''
def visualise_MoG_clusters(result):
    '''
```

```
    '''
    Convert hex values of type string to RGB of type int
    Input:
        colour_list: numpy array of type string (numColour x 1)
    Output:
        RGB: RGB component of type int (numColour x 3)
    '''
    def _hex_to_rgb(colour_list):
        RGB = np.array([])[np.newaxis,:].reshape(0,3)
        # Split hex values into R, G, B components
        # Convert components to int and store in RGB array
        for colour in colour_list:
            RGB = np.append(RGB, np.array([int(colour[1:3], 16), \
                                           int(colour[3:5], 16), \
                                           int(colour[5:7],
                                            ↪  16)]).reshape(1, 3),
                                            ↪  axis=0)
        return RGB

    '''
    Convert RGB of type int to hex string of format '#xxxxxx'
    Input:
        RGB: RGB component of type int (N x 3)
    Output:
        hex_colours: (N x 1)
    '''
    def _rgb_to_hex(RGB):
        hex_colours = np.array([])
        # Convert RGB ints to a single hex string
        RGB = RGB.astype(int)
        for colour in RGB:
            hex_colours = np.append(hex_colours,
              ↪  '#{:02X}{:02X}{:02X}'.format(colour[0], colour[1],
              ↪  colour[2]))
        return hex_colours

    '''
```

```
    Return the 'average' colour based on Plotly's default colour list and responsibility index
    Input:
        idx: responsibility index (N x K)
    Output:
        average_colour (N x 1)
    '''
    def get_colour_gradient(resp):
        # Assert error if there are more colours than available colours
        N = resp.shape[0]
        K = resp.shape[1]
        try:
            assert K <= colour_list.shape
        except AssertionError:
            print 'Not enough colours to colour all K clusters. Consider increasing number of colours in colour_list.'

        # Matrix multiply resp (N x K) and RGB-ed colour_list (K x 3) to obtain 'average' colour
        # Multiply max resp to whiten less certain data points
        # assigned_colour = np.matmul(resp, _hex_to_rgb(colour_list[:K]))
        assigned_colour = np.matmul(np.eye(K, dtype='int')[np.argmax(resp, axis=1)], _hex_to_rgb(colour_list[:K]))
        white_layer = np.repeat(255, N * 3).reshape(N, 3)

        # Append white_layer to assigned_colour on axis=2
        # pre_whitened (N x K x 2)
        pre_whitened = np.append(assigned_colour[:,:,np.newaxis], white_layer[:,:,np.newaxis], axis=2)

        # Create weights (N x 2)
        # Second layer takes the converse of the maximum responsibility (N x 1)
        weights = np.append(np.ones(N)[:,np.newaxis], 1 - np.amax(resp, axis=1)[:, np.newaxis], axis=1)

```

```
156         # Conform shape of weights to shape of pre_whitened
157         weights = np.transpose(np.tile(weights, (3, 1, 1)), (1, 0, 2))
158 
159         # Perform weighted-average to colours
160         whitened_colour = np.average(pre_whitened, weights=weights,
              ↪  axis=2)
161 
162         # Return matrix of colour in hex form
163         return _rgb_to_hex(whitened_colour)
164 
165     '''
166     Create x- and y-coordinates for ellipses for each cluster
167     Assummptions:
168         Joint independence and equal marginal variances
169         Dimension of data point is 2
170     Returns:
171         ellipse: x- and y-coordinates for K ellipses (N x K x D)
172     '''
173     def calc_ellipse_coordinates(centres, variances):
174         # Create trace for region to encompass 95% of the points (using
              ↪  Chi-squared critical value)
175         # Assuming joint independence and equal marginal variances
176 
177         # Chi-squared with df 2 and alpha=5%
178         crit_val = 5.991
179 
180         # Calculate axes length
181         axis_lengths = np.sqrt(variances * crit_val)
182 
183         # Calculate coordinates to trace ellipse
184         t = np.arange(-np.pi, np.pi + np.pi / 50, np.pi / 50) # Parameter
185         x = np.transpose(centres[:,0][:, np.newaxis]) + axis_lengths *
              ↪  np.cos(t)[:, np.newaxis]
186         y = np.transpose(centres[:,1][:, np.newaxis]) + axis_lengths *
              ↪  np.sin(t)[:, np.newaxis]
187 
```

```
        # Stack x- and y-coordinates along axis=2
        ellipse = np.stack([x, y], axis=2)

        return ellipse

    ######################
    ##  Function begins  ##
    ######################

    # Define K and divider between training and validation data
    K = result['K']
    divider = data2D.shape[0] * 2 / 3 # Anything before K is part of the
     ↪  training data. Anything after is part of validation data

    # Store cluster parameters and responsibility indices
    centres = result['cluster_centres'][-1]
    variances = result['cluster_variances'][-1]
    train_resp = result['train_resp'][-1]
    valid_resp = result['valid_resp'][-1]

    # Create ellipse coordinates
    ellipse = calc_ellipse_coordinates(centres, variances)

    # Define colour list as per Plotly's default colour list
    colour_list = np.array(['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728',
     ↪  '#9467bd', '#8c564b'])

    # Define blank figure
    figure = {
        'data': [],
        'layout': {}
    }

    # Create trace for training data points
    # Create trace for validation data points
    valid_data_trace = {
```

```
        'x': data2D[divider:][:,0],
        'y': data2D[divider:][:,1],
        'mode': 'markers',
        'hoverinfo': 'none',
        'marker': {
            'size': 4,
            'color': colour_list[np.argmax(valid_resp, axis=1)]
             ↪  #get_colour_gradient(valid_resp)
        }
    }

    # Append data traces
    figure['data'].append(valid_data_trace)

    for k in range(K):
        # Create trace for cluster centres
        centre_trace = {
            'x': np.round([centres[k][0]], 3),
            'y': np.round([centres[k][1]], 3),
            'name': 'Cluster {}'.format(k + 1),
            'mode': 'markers',
            'marker': {
                    'size': 12,
                    'symbol': 'diamond',
                    'color': colour_list[k],
                    'line': {'width': 3}
                }
        }

        # Create trace for region encompassing 95% of data points
        variance_trace = {
            'x': ellipse[:,k,:][:,0],
            'y': ellipse[:,k,:][:,1],
            'hoverinfo': 'none',
            'mode': 'lines',
            'name': 'Cluster {}'.format(k + 1),
```

```
            'marker': {
                'color': colour_list[k]
            }
        }

        # Add cluster trace
        for trace in [centre_trace, variance_trace]:
            figure['data'].append(trace)

    # Generate figure layout
    figure['layout'] = go.Layout(
        width = 900,
        height = 900,
        showlegend = False,
        title = 'MoG Clustering Visualisation (K = {})'.format(K),
        xaxis = {'range': [-4, 4], 'autorange': False},
        yaxis = {'range': [-5, 2], 'autorange': False}
    )

    # Upload graph to cloud
    py.iplot(figure, \
             filename='/ECE521: A3/Q2: Mixture of
              ↪  Gaussians/Q2.3_MoG_clusters_K={}'.format(K), \
             sharing='private')

    return pyo.iplot(figure)

fig2_2_3 = []
for result in results_2_2_3:
    fig2_2_3.append(visualise_MoG_clusters(result))
```

4.1.8 MoG - Q2.2.4

```
# Run K-means
results_2_2_2_4_K_means = run_k_means(K_list=[1, 2, 3, 4, 5, 6, 7, 8, 9,
 ↪  10, 11, 12, 13, 14, 15], data_dim=100, device='cpu', has_valid=True)

# Run MoG
results_2_2_4_MoG = run_MoG_v2(K_list=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
 ↪  12, 13, 14, 15], D=100, has_valid=True, device='gpu',
 ↪  QUES_DIR='/Q2.2.4')

# Save results
np.save('./Results/MoG/2_2_4_MoG_x15.npy', results_2_2_4_MoG)
np.save('./Results/MoG/2_2_4_K-means.npy', results_2_2_2_4_K_means)

# Plot validation losses
def IGraph_2_2_4(K_means, MoG):
    valid_loss_K_means = [result['valid_loss'][-1] for result in K_means]
    valid_loss_MoG = [result['valid_loss'][-1] for result in MoG]

    figure = tools.make_subplots(rows=2, cols=1, shared_xaxes=True,
     ↪  subplot_titles=('K-means', 'MoG'))

    K_means_trace = {
        'x': [k + 1 for k in range(15)],
        'y': valid_loss_K_means,
        'name': 'K-means'
    }

    MoG_trace = {
        'x': [k + 1 for k in range(15)],
        'y': valid_loss_MoG,
        'name': 'MoG'
    }

    figure.append_trace(K_means_trace, 1, 1)
```

```
    figure.append_trace(MoG_trace, 2, 1)

    figure['layout'].update({
        'height': 500,
        'width': 800,
        'title': 'K-means and MoG Models Performances on data100D.npy',
        'xaxis1': {'title': 'Number of clusters, K', 'dtick': 1},
        'yaxis1': {'title': '$\\text{Valid. Loss, } \\mathcal{L}(\\mathbf{\\mu})$'},
        'yaxis2': {'title': '$\\text{Valid. Loss, } - P(\\mathbf{X})$'},
        'showlegend': False
    })



    py.iplot(figure, filename='/ECE521: A3/Q2: Mixture of Gaussians/Q2.4_K_means_vs_MoG', sharing='private')

    return pyo.iplot(figure)

IGraph_2_2_4(results_2_2_4_K_means, results_2_2_4_MoG)
```

4.1.9 Base code for FA

```
# Import relevant packages
import tensorflow as tf
import numpy as np

import time
import datetime

# Non-interactive plotting
import matplotlib.pyplot as plt

# Interactive plotting
from plotly import tools
import plotly.plotly as py
import plotly.graph_objs as go
import plotly.grid_objs as gro
import plotly.offline as pyo
from plotly.offline import download_plotlyjs

# Configure environment
%config InlineBackend.figure_format = 'retina'
np.set_printoptions(precision=3)

# Global Variables
CURRENT_DIR =
 ↪  '/Users/christophertee/Dropbox/University/MASc/Courses/Winter 2017' +
 ↪  \
              '/ECE521 (Inference Algorithms & Machine
                ↪  Learning)/Assignment 3'
LOG_DIR = '/Logs'

# Activate Plotly Offline for Jupyter
pyo.init_notebook_mode(connected=True)

# Define global variable SEED
```

```
SEED = 521

# Load data into memory

"""
tinymnist.npz consists of images of '3's and '5's with dimensions (8 x 8)

train_data: 700 data points
valid_data: 100 data points
test_data: 400 data points
"""
with np.load ("./Data/tinymnist.npz") as data :
    # Generate _data
    train_data, train_target = data ["x"], data["y"]
    valid_data, valid_target = data ["x_valid"], data ["y_valid"]
    test_data, test_target = data ["x_test"], data ["y_test"]

'''
Builds TensorFlow graph for FA

Input:
    K: number of latent variables
    D: dataset dimension
    device: CPU or GPU to perform computation-heavy ops
Internal variables:
    X: input data matrix (N x D)
    mu: mean of each input (D x 1)
    psi_vector: variance of x_n given s_n for each dimension (D x 1)
    W: latent_matrix that projects s_n from K-dimensions to D-dimensions
 ↪  (K x D)
    Sigma: Marginal covariance matrix (D x D)
'''
def build_FA(K, D, device='cpu'):
    '''
    Helper function to add histogram tag to variables
    Input:
```

```
67          var: variable to be tagged with histogram summary
68      '''
69      def _add_histogram(vars_):
70          for var in vars_:
71              tf.summary.histogram(var.op.name, var)
72
73      ######################
74      ##  Function begins  ##
75      ######################
76
77      # Define computation device
78      try:
79          assert device == 'cpu' or device == 'gpu'
80      except AssertionError:
81          print 'Invalid device chosen. Please use \'cpu\' or \'gpu\''
82          quit()
83      device = '/' + device + ':0'
84
85      # Fix TF graph seed
86      tf.set_random_seed(SEED)
87
88      with tf.device('/cpu:0'):
89          # Define placeholder
90          with tf.name_scope('placeholder'):
91              X = tf.placeholder(tf.float32, shape=[None, D], name='inputs')
92
93          # Define parameters
94          with tf.variable_scope('generated_parameters'):
95              phi = tf.get_variable('latent_for_psi', shape=[D, 1], \
96
                                      ↪  initializer=tf.truncated_normal_initializer(seed=SI
97              W = tf.get_variable('latent_matrix_W', shape=[D, K], \
98
                                           ↪  initializer=tf.truncated_normal_initializer
99              mu = tf.get_variable('mean', shape=[D, 1],
                 ↪  initializer=tf.truncated_normal_initializer(seed=SEED))
```

```
100
101     with tf.device(device):
102         # Calculate gaussian parameters
103         with tf.name_scope('gaussian_parameters'):
104             psi_vector = tf.exp(phi, name='variance_vector_psi')
105             Psi = tf.multiply(tf.eye(tf.shape(psi_vector)[0]), psi_vector,
                  ↪  name='Psi_matrix')
106             Sigma = tf.add(Psi, tf.matmul(W, tf.transpose(W)),
                  ↪  name='Sigma')
107
108         # Calculate projection matrix to infer latent variable s
109         with tf.name_scope('projection_matrix'):
110             Sigma_s_posterior = tf.matrix_inverse(tf.eye(K) \
111                                         + tf.matmul(tf.transpose(W), \
112
                                                  ↪  tf.matmul(tf.matrix_inverse(Psi
                                                  ↪  W)))
113             W_proj = tf.matmul(Sigma_s_posterior,
                  ↪  tf.matmul(tf.transpose(W), tf.matrix_inverse(Psi)))
114
115         # Calculate loss function
116         with tf.name_scope('marginal_log_likelihood'):
117             with tf.name_scope('Mahalanobis_dist'):
118                 # Expand variables for tensor multiplication
119                 X_expanded = tf.expand_dims(X, axis=2)
120                 mu_expanded = tf.expand_dims(mu, axis=0)
121
122                 # Calculate mahalanobis distance
123                 Sigma_inv_tiled =
                      ↪  tf.tile(tf.expand_dims(tf.matrix_inverse(Sigma),
                      ↪  axis=0), \
124                                                 multiples=[tf.shape(X)[0],1,1],
                                                      ↪  \
125                                                 name='Sigma_inv_tiled')
126
127                 dist = tf.reduce_sum(- 1. / 2 * \
```

```
                                    tf.matmul(tf.transpose(X_expanded -
                                     ↪  mu_expanded, perm=[0,2,1]), \
                                            tf.matmul(Sigma_inv_tiled, \
                                                      X_expanded -
                                                       ↪  mu_expanded)),
                                                       ↪  \
                                    name='Mahalanobis_dist')

            # Calculate log gaussian constant
            with tf.name_scope('log_gauss_const'):
                log_gauss_const = tf.negative(tf.cast(tf.shape(X)[0],
                 ↪  tf.float32) * tf.cast(D, tf.float32)\
                                              * tf.log(2. * np.pi) / 2,\
                                              name='log_gauss_const')

            # Calculate log_determinant
            with tf.name_scope('log_det'):
                test = tf.cholesky(Sigma)
                log_det = tf.negative(tf.cast(tf.shape(X)[0], tf.float32)
                 ↪  \
                                      *
                                       ↪  tf.reduce_sum(tf.log(tf.diag_part(tf.cholesky
                                      name='log_det')

            # Calculate loss function
            loss = tf.negative(tf.add_n([dist, log_gauss_const, log_det]),
             ↪  name='loss')
            tf.summary.scalar('loss', loss)

        # Define optimizer
        with tf.name_scope('Adam_optimizer'):
            optimizer = tf.train.AdamOptimizer(learning_rate=0.01, \
                                               beta1=0.9, beta2=0.99,
                                                ↪  epsilon=1e-5).minimize(loss)

    with tf.device('/cpu:0'):
```

```
156         # Add histogram summaries for variables of interest
157         _add_histogram([psi_vector, W, Sigma])
158 
159         # Merge all summaries
160         merged = tf.summary.merge_all()
161 
162     return X, mu, psi_vector, W, Sigma, W_proj, loss, optimizer, merged,
      ↪  test
163 
164 '''
165 Runs FA training algorithm. Tensorboard embedding enabled
166 '''
167 def run_FA(K_list, QUES_DIR, D=64, device='cpu'):
168     '''
169     Embed data for visualization purposes
170     '''
171     def embed_data(D, train_writer):
172         # Define input data
173         if D == 64:
174             input_data = train_data
175             input_data_name = 'tinymnist_training'
176         elif D == 3:
177             input_data = toy_data
178             input_data_name = 'toy_data'
179 
180         # Create variable to embed
181         data_to_embed = tf.Variable(input_data, name=input_data_name,
          ↪  trainable=False, collections=[])
182 
183         # Define projector configurations
184         config = projector.ProjectorConfig()
185 
186         # Add embedding
187         embedding = config.embeddings.add()
188 
189         # Connect tf.Variable to embedding
```

```
        embedding.tensor_name = data_to_embed.name

        # Evaluate tf.Variable
        sess.run(data_to_embed.initializer)

        # Create save checkpoint
        saver = tf.train.Saver([data_to_embed])
        saver.save(sess, SUMMARY_DIR + sub_dir + '/train/model.ckpt',
          ↪  MAX_ITER)

        # Write projector_config.pbtxt in LOG_DIR
        projector.visualize_embeddings(train_writer, config)

    ######################
    ##  Function begins  ##
    ######################

    # Check compatibility of dataset
    try:
        assert D == 64 or D == 3
    except AssertionError:
        print 'Incompatible dataset dimension, D. Please use 64 or 3\
            for tinymnist or toy dataset respectively'
        quit()

    # Define locally global function
    MAX_ITER = 1200 if D == 64 else 5000
    CURR_TIME = '{:%b%d %H_%M_%S}'.format(datetime.datetime.now())
    SUMMARY_DIR = CURRENT_DIR + LOG_DIR + '/FA/' + QUES_DIR + '/' +
      ↪  CURR_TIME

    # Create list to store run results
    results = []

    for K in K_list:
        # Clear any pre-defined graph
```

```
        tf.reset_default_graph()

        # Build TensorFlow graph
        X, mu, psi_vector, W, Sigma, W_proj, loss, optimizer, merged, test
          ↪  = build_FA(K, D, device)

        # Select appropriate input_data
        if D == 64:
            input_data = train_data
        elif D == 3:
            input_data = toy_data

        # Create arrays to log losses, psi and W
        train_loss = np.array([])[:, np.newaxis]
        if D == 64:
            valid_loss = np.array([])[:, np.newaxis]
            test_loss = np.array([])[:, np.newaxis]

        mean = np.array([])[:, np.newaxis].reshape(0, D)
        psi = np.array([])[:, np.newaxis].reshape(0, D)
        Ws = np.array([])[:, np.newaxis, np.newaxis].reshape(0, D, K)
        Wproj = np.array([])[:, np.newaxis, np.newaxis].reshape(0, K, D)

        # Begin session
        with tf.Session(config=tf.ConfigProto(allow_soft_placement=True,
          ↪  log_device_placement=False)) as sess:
            # Log start time
            start_time = time.time()

            # Create sub-directory title
            sub_dir = '/K={},D={}'.format(K, D)

            # Create summary writers
            train_writer = tf.summary.FileWriter(SUMMARY_DIR + sub_dir +
              ↪  '/train', graph=sess.graph)
            if D == 64:
```

```
257             valid_writer = tf.summary.FileWriter(SUMMARY_DIR + sub_dir
                    ↪  + '/valid')
258             test_writer = tf.summary.FileWriter(SUMMARY_DIR + sub_dir
                    ↪  + '/test')
259
260         # Initialise all TensorFlow variables
261         tf.global_variables_initializer().run()
262
263         # Define iterator
264         curr_iter = 0
265
266         # Calculate training (and validation) loss,
267         # cluster centres and responsibility indices before any
                ↪  training
268         err, summaries, curr_mu, curr_psi, curr_W, curr_W_proj = \
269             sess.run([loss, merged, mu, psi_vector, W, W_proj],
                    ↪  feed_dict={X: input_data})
270         train_loss = np.append(train_loss, err)
271         train_writer.add_summary(summaries, curr_iter)
272
273         # Log psi and W
274         mean = np.append(mean, np.transpose(curr_mu), axis=0)
275         psi = np.append(psi, np.transpose(curr_psi), axis=0)
276         Ws = np.append(Ws, curr_W[np.newaxis, :, :], axis=0)
277         Wproj = np.append(Wproj, curr_W_proj[np.newaxis, :, :],
                ↪  axis=0)
278
279         if D == 64:
280             # Log validation loss
281             err, summaries  = sess.run([loss, merged],
                    ↪  feed_dict={X:valid_data})
282             valid_loss = np.append(valid_loss, err)
283             valid_writer.add_summary(summaries, curr_iter)
284
285             # Log test loss
286             err, summaries  = sess.run([loss, merged],
                    ↪  feed_dict={X:test_data})
```

```
            test_loss = np.append(test_loss, err)
            test_writer.add_summary(summaries, curr_iter)

        # Begin training
        while curr_iter < MAX_ITER:
            # Train graph
            _, summaries, err = sess.run([optimizer, merged, loss],
              ↪  feed_dict={X:input_data})

            # Add training loss
            train_loss = np.append(train_loss, err)
            train_writer.add_summary(summaries, curr_iter + 1)

            if D == 64:
                # Log validation loss
                summaries, err = sess.run([merged, loss],
                  ↪  feed_dict={X:valid_data})
                valid_loss = np.append(valid_loss, err)
                valid_writer.add_summary(summaries, curr_iter)

                # Log test loss
                err, summaries  = sess.run([loss, merged],
                  ↪  feed_dict={X:test_data})
                test_loss = np.append(test_loss, err)
                test_writer.add_summary(summaries, curr_iter)

            # Log responsibility indices and cluster centres every 10%
              ↪  of maximum iteration
            if ((float(curr_iter) + 1) * 100 / MAX_ITER) % 10 == 0:
                curr_mu, curr_psi, curr_W, curr_W_proj = sess.run([mu,
                  ↪  psi_vector, W, W_proj], feed_dict={X:input_data})

                mean = np.append(mean, np.transpose(curr_mu), axis=0)
                psi = np.append(psi, np.transpose(curr_psi), axis=0)
                Ws = np.append(Ws, curr_W[np.newaxis, :, :], axis=0)
                Wproj = np.append(Wproj, curr_W_proj[np.newaxis, :,
                  ↪  :], axis=0)
```

```
318
319             # Post training progress to user, every 100 iterations
320             if curr_iter % 100 == 99:
321                 print 'iter: {:3d}, train_loss:
                      ↪  {:5.0f}'.format(curr_iter + 1, train_loss[-1])
322
323             curr_iter += 1
324
325         # End of while loop
326         print 'Max iteration reached'
327
328         # Embed data
329         embed_data(D, train_writer)
330
331         # Close writers
332         train_writer.close()
333         if D == 64:
334             valid_writer.close()
335             test_writer.close()
336
337         if D == 64:
338             results.append(
339                 {
340                     'K': K,
341                     'train_loss': train_loss,
342                     'valid_loss': valid_loss,
343                     'test_loss': test_loss,
344                     'psi': psi,
345                     'mean': mean,
346                     'W': Ws,
347                     'W_proj': Wproj,
348                     'time_of_run': '{:%b%d
                          ↪  %H_%M_%S}'.format(datetime.datetime.now())
349                 }
350             )
351         elif D == 3:
```

```
352                 results.append(
353                     {
354                         'K': K,
355                         'train_loss': train_loss,
356                         'psi': psi,
357                         'mean': mean,
358                         'W': Ws,
359                         'W_proj': Wproj,
360                         'time_of_run': '{:%b%d
                              ↪  %H_%M_%S}'.format(datetime.datetime.now())
361                     }
362                 )
363 
364             # TODO calculate convergence
365             print 'K: {:3d}, duration: {:3.1f}s\n'\
366                 .format(K, time.time() - start_time)
367 
368     print 'RUN COMPLETED'
369     return results
```

4.1.10 FA - Q3.1.2

```
# Run FA
results_3_1_2 = run_FA(K_list=[4], QUES_DIR='/Q3.1.2', device='gpu')

# Save results
np.save('./Results/FA/3_1_2.npy', results_3_1_2)

# Visualise weights
'''
Creates 2x2 subplot of weights
Each subplot consists of an 8x8 heatmap for the weight of each latent variable
Input:
    weights: final trained weights (D x K)
'''
def visualise_weights(weights):
    # Define colour list as per Plotly's default colour list
    colour_list = np.array(['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b'])

    # Define empty figure
    figure = tools.make_subplots(rows=2, cols=2, subplot_titles=('Weight 1', 'Weight 2', 'Weight 3', 'Weight 4'))

    # Define subplot traces
    for i, weight in enumerate(np.transpose(weights)):
        trace = go.Heatmap(
            z = np.reshape(weight, (8,8)), # TODO Reverse order
            colorscale = [[0, '#000000'], [1, '#FFFFFF']],
            showscale = False
        )
        figure.append_trace(trace, i / 2 + 1, i % 2 + 1)

        figure['layout']['xaxis{}'.format(i + 1)].update(showticklabels = False, ticks = '')
```

```
        figure['layout']['yaxis{}'.format(i + 1)].update(showticklabels =
          ↪  False, ticks = '', autorange='reversed')

    figure['layout'].update(
        height = 900,
        width = 800,
        showlegend = False,
        title = 'Weights Visualisation of Latent Matrix, W'
    )

    py.iplot(figure, filename='/ECE521: A3/Q3: Factor
      ↪  Analysis/Q1.2_weights_viz', sharing='private')
    return pyo.iplot(figure)

visualise_weights(results_3_1_2[0]['W'][-1])
```

4.1.11 FA - Q3.1.3

```
# Generate toy dataset
# Reset random seed
np.random.seed(SEED)

# Define variable to store toy data
toy_data = np.array([])[np.newaxis, :].reshape(0, 3)

for i in range(5000):
    # Sample s from normal distribution
    s = np.random.randn(3,1)

    # Define conversion matrix from s to x
    A = np.array([[1, 0, 0],
                  [1, 0.001, 0],
                  [0, 0, 10]])

    toy_data = np.append(toy_data, np.transpose(np.matmul(A, s)), axis=0)

# Train using PCA
'''
Finds the largest principal component by finding the normalised
 ↪  eigenvector corresponding
to the largest eigenvalue of the data covariance matrix
'''
def largest_component_PCA(data):
    # Obtain covariance matrix
    sigma = np.cov(np.transpose(data))

    # Calculate eigenvalues and eigenvectors
    e_value, e_vector = np.linalg.eig(sigma)

    # Return the largest principle component
    PC = e_vector[np.argmax(e_value)][:, np.newaxis]

```

```
    # Save data
    np.save('./Results/FA/3_1_3_PCA_x{}.npy'.format(data.shape[0]), PC)

    return PC

print 'Principle component: \n{}'.format(largest_component_PCA(toy_data))

# Train using FA
results_3_1_3 = run_FA(K_list=[1], D=3, QUES_DIR='Q3.1.3', device='cpu')
```

4.2 Soon Chee Loong's Implementation

4.2.1 K-Means

```
import tensorflow as tf
import numpy as np
from dataInitializer import DataInitializer
import datetime
import sys

class KMeans(object):
    def __init__(self, questionTitle, K, trainData, validData, hasValid,
      ↪  dataType, numEpoch = 50, learningRate = 0.1):
        """
        Constructor
        """
        self.K = K
        self.dataType = dataType
        self.trainData = trainData
        self.validData = validData
        self.D = self.trainData[0].size # Dimension of each data
        self.hasValid = hasValid
        self.learningRate = learningRate
        self.numEpoch = numEpoch
```

```
20          self.miniBatchSize = self.trainData.shape[0] # miniBatchSize is
             ↪  entire data size
21          self.questionTitle = questionTitle
22          self.optimizer = tf.train.AdamOptimizer(learning_rate =
             ↪  self.learningRate, beta1=0.9, beta2=0.99, epsilon=1e-5)
23          # Execute KMeans
24          self.KMeansMethod()
25
26      def printPlotResults(self, xAxis, yTrainErr, yValidErr, numUpdate,
         ↪  minAssignTrain, currTrainData, centers, minAssignValid):
27          figureCount = 0 # TODO: Make global
28          import matplotlib.pyplot as plt
29
30          print "K: ", self.K
31          print "Iter: ", numUpdate
32          print str(self.K) + "Lowest TrainLoss", np.min(yTrainErr)
33          print str(self.K) + "Lowest ValidLoss", np.min(yValidErr)
34          # Count how many assigned to each class
35          numTrainAssignEachClass = np.bincount(minAssignTrain)
36          numValidAssignEachClass = np.bincount(minAssignValid)
37          print "Train Percentage Assignment To Classes:",
             ↪  percentageTrainAssignEachClass
38          print "Train Assignments To Classes:", numTrainAssignEachClass
39          percentageTrainAssignEachClass =
             ↪  numTrainAssignEachClass/float(sum(numTrainAssignEachClass))
40
41          percentageValidAssignEachClass = percentageTrainAssignEachClass #
             ↪  Initialize
42
43          if self.hasValid:
44              print "Valid Assignments To Classes:", numValidAssignEachClass
45              percentageValidAssignEachClass =
                 ↪  numValidAssignEachClass/float(sum(numValidAssignEachClass))
46              print "Valid Percentage Assignment To Classes:",
                 ↪  percentageValidAssignEachClass
47
```

```
48          trainStr = "Train"
49          validStr = "Valid"
50          typeLossStr = "Loss"
51          typeScatterStr = "Assignments"
52          trainLossStr = trainStr + typeLossStr
53          validLossStr = validStr + typeLossStr
54          iterationStr = "Iteration"
55          dimensionOneStr = "D1"
56          dimensionTwoStr = "D2"
57          paramStr = "K" + str(self.K) + "Learn" + str(self.learningRate) +
             ↪  "NumEpoch" + str(self.numEpoch)
58
59          # Train Loss
60          figureCount = figureCount + 1
61          plt.figure(figureCount)
62          title = trainStr + typeLossStr + paramStr
63          plt.title(title)
64          plt.xlabel(iterationStr)
65          plt.ylabel(typeLossStr)
66          plt.plot(np.array(xAxis), np.array(yTrainErr), label =
             ↪  trainLossStr)
67          plt.legend()
68          plt.savefig(self.questionTitle + title + ".png")
69          plt.close()
70          plt.clf()
71
72          # Valid Loss
73          if self.hasValid:
74              figureCount = figureCount + 1
75              plt.figure(figureCount)
76              title = validStr + typeLossStr + paramStr
77              plt.title(title)
78              plt.xlabel(iterationStr)
79              plt.ylabel(typeLossStr)
80              plt.plot(np.array(xAxis), np.array(yValidErr), label =
                 ↪  validLossStr)
```

```
81              plt.legend()
82              plt.savefig(self.questionTitle + title + ".png")
83              plt.close()
84              plt.clf()
85
86          if self.dataType != "2D":
87              return
88
89          # Plot percentage in each different classes as well
90          # Scatter plot based on assignment colors
91          # Including percentage as the label
92          figureCount = figureCount + 1
93          plt.figure(figureCount)
94          title = trainStr + typeScatterStr + paramStr
95          plt.title(title)
96          plt.xlabel(dimensionOneStr)
97          plt.ylabel(dimensionTwoStr)
98          colors = ['blue', 'red', 'green', 'black', 'yellow', 'magenta',
             ↪  'cyan', 'brown', 'orange',
99                    'aqua']
100         colors = colors[:self.K]
101         plt.scatter(currTrainData[:, 0], currTrainData[:, 1],
             ↪  c=minAssignTrain, s=10, alpha=0.5)
102         for i, j, k in zip(centers, percentageTrainAssignEachClass,
             ↪  colors):
103             plt.plot(i[0], i[1], 'kx', markersize=15, label=j, c=k)
104         plt.legend()
105         plt.savefig(self.questionTitle + title + ".png")
106         plt.close()
107         plt.clf()
108
109         if self.hasValid:
110             # Valid Assignments
111             figureCount = figureCount + 1
112             plt.figure(figureCount)
113             title = validStr + typeScatterStr + paramStr
```

```
114             plt.title(title)
115             plt.xlabel(dimensionOneStr)
116             plt.ylabel(dimensionTwoStr)
117             colors = ['blue', 'red', 'green', 'black', 'yellow',
                ↪  'magenta', 'cyan', 'brown', 'orange',
118                     'aqua']
119             colors = colors[:self.K]
120             plt.scatter(self.validData[:, 0], self.validData[:, 1],
                ↪  c=minAssignValid, s=10, alpha=0.5)
121             for i, j, k in zip(centers, percentageValidAssignEachClass,
                ↪  colors):
122                 plt.plot(i[0], i[1], 'kx', markersize=15, label=j, c=k)
123             plt.legend()
124             plt.savefig(self.questionTitle + title + ".png")
125             plt.close()
126             plt.clf()
127
128     def PairwiseDistances(self, X, U):
129         """
130         input:
131             X is a matrix of size (B x D)
132             U is a matrix of size (K x D)
133         output:
134             Distances = matrix of size (B x D) containing the pairwise
↪  Euclidean distances
135         """
136         batchSize = tf.shape(X)[0]
137         dimensionSize = tf.shape(X)[1]
138         numClusters = tf.shape(U)[0]
139         X_broadcast = tf.reshape(X, (batchSize, 1, dimensionSize))
140         sumOfSquareDistances =
                ↪  tf.reduce_sum(tf.square(tf.subtract(X_broadcast, U)), 2)
141         return sumOfSquareDistances
142
143     def KMeansMethod(self):
144         '''
```

```
        Build Graph and execute in here
        so don't have to pass variables one by one
        Bad Coding Style but higher programmer productivity
        '''
        # Build Graph
        U = tf.Variable(tf.truncated_normal([self.K, self.D]))
        train_data = tf.placeholder(tf.float32, shape=[None, self.D],
          ↪  name="trainingData")
        sumOfSquare = self.PairwiseDistances(train_data, U)
        minSquare = tf.reduce_min(sumOfSquare, 1)
        minAssignments = tf.argmin(sumOfSquare,1)
        loss = tf.reduce_sum(minSquare)
        validLoss = loss
        minValidAssignments = minAssignments

        if self.hasValid:
            valid_data = tf.placeholder(tf.float32, shape=[None, self.D],
              ↪  name="validationData")
            validSumOfSquare = self.PairwiseDistances(valid_data, U)
            validLoss = tf.reduce_sum(tf.reduce_min(validSumOfSquare, 1))
            minValidAssignments = tf.argmin(validSumOfSquare, 1)

        train = self.optimizer.minimize(loss)

        # Session
        init = tf.global_variables_initializer()
        sess = tf.InteractiveSession()
        sess.run(init)
        currEpoch = 0
        minAssign = 0
        centers = 0
        xAxis = []
        yTrainErr = []
        yValidErr = []
        numUpdate = 0
        step = 0
```

```
179         currTrainDataShuffle = self.trainData
180         while currEpoch < self.numEpoch:
181             np.random.shuffle(self.trainData) # Shuffle Batches
182             currTrainDataShuffle = self.trainData
183             step = 0
184             while step*self.miniBatchSize < self.trainData.shape[0]:
185                 feedDicts = {train_data:
                      ↪  self.trainData[step*self.miniBatchSize:(step+1)*self.miniBatchSize
186                 if self.hasValid:
187                     feedDicts = {train_data:
                          ↪  self.trainData[step*self.miniBatchSize:(step+1)*self.miniBatchS
                          ↪  valid_data:self.validData}
188                 _, minAssignTrain, minAssignValid, centers, errTrain,
                      ↪  errValid = sess.run([train, minAssignments,
                      ↪  minValidAssignments, U, loss, validLoss], feed_dict =
                      ↪  feedDicts)
189                 xAxis.append(numUpdate)
190                 yTrainErr.append(errTrain)
191                 yValidErr.append(errValid)
192                 step += 1
193                 numUpdate += 1
194             currEpoch += 1
195             if currEpoch%50 == 0:
196                 logStdOut("e: " + str(currEpoch))
197         if self.dataType == "2D":
198             print "Center Values", centers
199         self.printPlotResults(xAxis, yTrainErr, yValidErr, numUpdate,
              ↪  minAssignTrain, currTrainDataShuffle, centers,
              ↪  minAssignValid)
200
201 def executeKMeans(questionTitle, K, dataType, hasValid):
202     """
203     Re-loads the data and re-randomize it with same seed anytime to ensure
      ↪  replicable results
204     """
205     logStdOut(questionTitle)
```

```
206     print questionTitle
207     trainData = 0
208     validData = 0
209     # Load data with seeded randomization
210     dataInitializer = DataInitializer()
211     if hasValid:
212         trainData, validData = dataInitializer.getData(dataType, hasValid)
213     else:
214         trainData = dataInitializer.getData(dataType, hasValid)
215     # Execute algorithm
216     kObject = KMeans(questionTitle, K, trainData, validData, hasValid,
        ↪ dataType)
217     logElapsedTime(questionTitle + "K" + str(K))
218
219
220 # Global for logging
221 questionTitle = "" # Need to be global for logging to work
222 startTime = datetime.datetime.now()
223 figureCount = 1 # To not overwrite existing pictures
224
225 def logStdOut(message):
226     # Temporary print to std out
227     sys.stdout = sys.__stdout__
228     print message
229     # Continue editing same file
230     sys.stdout = open("result" + questionTitle + ".txt", "a")
231
232 def logElapsedTime(message):
233     ''' Logs the elapsedTime with a given message '''
234     global startTime
235     endTime = datetime.datetime.now()
236     elapsedTime = endTime - startTime
237     hours, remainder = divmod(elapsedTime.seconds, 3600)
238     minutes, seconds = divmod(remainder, 60)
239     totalDays = elapsedTime.days
240     timeStr = str(message) + ': Days: ' + str(totalDays) +  " hours: " +
        ↪ str(hours) + ' minutes: ' + str(minutes) +  ' seconds: ' +
        ↪ str(seconds)
```

```
241     logStdOut(timeStr)
242     startTime = datetime.datetime.now()
243 
244 if __name__ == "__main__":
245     print "ECE521 Assignment 3: Unsupervised Learning: K Means"
246 
247     # Unsupervised => Data has no label or target
248     '''
249     questionTitle = "1.1.2"
250     dataType = "2D"
251     hasValid = False # No validation data
252     K = 3
253     executeKMeans(questionTitle, K, dataType, hasValid)
254     # '''
255 
256     '''
257     questionTitle = "1.1.3"
258     diffK = [1, 2, 3, 4, 5]
259     dataType = "2D"
260     hasValid = False
261     for K in diffK:
262         executeKMeans(questionTitle, K, dataType, hasValid)
263     # '''
264 
265     '''
266     questionTitle = "1.1.4"
267     diffK = [1, 2, 3, 4, 5]
268     dataType = "2D"
269     hasValid = True
270     for K in diffK:
271         executeKMeans(questionTitle, K, dataType, hasValid)
272     # '''
273 
274     # Run using 100D data
275     questionTitle = "2.2.4.1"
276     diffK = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
    dataType = "100D"
    hasValid = True
    for K in diffK:
        executeKMeans(questionTitle, K, dataType, hasValid)
    # '''
```

4.2.2 Mixture of Gaussians

```
import tensorflow as tf
import numpy as np
import sys
from dataInitializer import DataInitializer
from utils import *
import datetime
import sys

class MixtureOfGaussians(object):
    def __init__(self, questionTitle, K, trainData, validData, hasValid,
     ↪  dataType, numEpoch = 500, learningRate = 0.001):
        """
        Constructor
        """
        self.K = K # number of clusters
        self.dataType = dataType
        self.trainData = trainData
        self.validData = validData
        self.D = self.trainData[0].size # Dimension of each data
        self.hasValid = hasValid
        self.learningRate = learningRate
        self.numEpoch = numEpoch
        self.miniBatchSize = self.trainData.shape[0] # miniBatchSize is
         ↪  entire data size
        self.questionTitle = questionTitle
        self.optimizer = tf.train.AdamOptimizer(learning_rate =
         ↪  self.learningRate, beta1=0.9, beta2=0.99, epsilon=1e-5)
```

```
25          # Execute Mixture of Gaussians
26          self.MixtureOfGaussiansMethod()
27
28      def printPlotResults(self, xAxis, yTrainErr, yValidErr, numUpdate,
          ↪  minAssignTrain, currTrainData, clusterMean, clusterStdDeviation,
          ↪  clusterPrior,  minAssignValid):
29          figureCount = 0 # TODO: Make global
30          import matplotlib.pyplot as plt
31          if self.dataType == "2D":
32              print "mean", clusterMean
33          print "K: ", self.K
34          print "Iter: ", numUpdate
35          numTrainAssignEachClass = np.bincount(minAssignTrain)
36          numValidAssignEachClass = np.bincount(minAssignValid)
37          print "Train Assignments To Classes:", numTrainAssignEachClass
38          percentageTrainAssignEachClass =
              ↪  numTrainAssignEachClass/float(sum(numTrainAssignEachClass))
39          print "Train Percentage Assignment To Classes:",
              ↪  percentageTrainAssignEachClass
40          percentageValidAssignEachClass = percentageTrainAssignEachClass #
              ↪  Initialize
41          if self.hasValid:
42              print "Valid Assignments To Classes:", numValidAssignEachClass
43              percentageValidAssignEachClass =
                  ↪  numValidAssignEachClass/float(sum(numValidAssignEachClass))
44              print "Valid Percentage Assignment To Classes:",
                  ↪  percentageValidAssignEachClass
45          print "prior", clusterPrior
46          print "prior.shape", clusterPrior.shape
47          print "prior Sum", np.sum(clusterPrior)
48          print "stdDeviation", clusterStdDeviation
49          print "stdDeviationShape", clusterStdDeviation.shape
50          print str(self.K) + "Lowest TrainLoss", np.min(yTrainErr)
51          print str(self.K) + "Lowest ValidLoss", np.min(yValidErr)
52
53          trainStr = "Train"
```

```
54          validStr = "Valid"
55          typeLossStr = "Loss"
56          typeScatterStr = "Assignments"
57          trainLossStr = trainStr + typeLossStr
58          validLossStr = validStr + typeLossStr
59          iterationStr = "Iteration"
60          dimensionOneStr = "D1"
61          dimensionTwoStr = "D2"
62          paramStr = "K" + str(self.K) + "Learn" + str(self.learningRate) +
             ↪  "NumEpoch" + str(self.numEpoch)
63
64          # Train Loss
65          figureCount = figureCount + 1
66          plt.figure(figureCount)
67          title = trainStr + typeLossStr + paramStr
68          plt.title(title)
69          plt.xlabel(iterationStr)
70          plt.ylabel(typeLossStr)
71          plt.plot(np.array(xAxis), np.array(yTrainErr), label =
             ↪  trainLossStr)
72          plt.legend()
73          plt.savefig(self.questionTitle + title + ".png")
74          plt.close()
75          plt.clf()
76
77          # Valid Loss
78          if self.hasValid:
79              figureCount = figureCount + 1
80              plt.figure(figureCount)
81              title = validStr + typeLossStr + paramStr
82              plt.title(title)
83              plt.xlabel(iterationStr)
84              plt.ylabel(typeLossStr)
85              plt.plot(np.array(xAxis), np.array(yValidErr), label =
                 ↪  validLossStr)
86              plt.legend()
```

```
87              plt.savefig(self.questionTitle + title + ".png")
88              plt.close()
89              plt.clf()
90
91          if self.dataType != "2D":
92              return
93          # Plot percentage in each different classes as well
94          # Scatter plot based on assignment colors
95          # Including percentage as the label
96          # Train Scatter Plot
97          figureCount = figureCount + 1
98          plt.figure(figureCount)
99          plt.axes()
100         title = trainStr + typeScatterStr + paramStr
101         plt.title(title)
102         plt.xlabel(dimensionOneStr)
103         plt.ylabel(dimensionTwoStr)
104         colors = ['blue', 'red', 'green', 'black', 'yellow']
105         plt.scatter(currTrainData[:, 0], currTrainData[:, 1],
             ↪  c=minAssignTrain, s=10, alpha=0.5)
106         colors = colors[:self.K]
107         for i, j, k, l in zip(clusterMean, percentageTrainAssignEachClass,
             ↪  colors, clusterStdDeviation[0]):
108             plt.plot(i[0], i[1], 'kx', markersize=15, label=j, c=k)
109             circle = plt.Circle((i[0], i[1]), radius=2*l, color=k,
                 ↪  fill=False)
110             plt.gca().add_patch(circle)
111         plt.legend()
112         plt.savefig(self.questionTitle + title + ".png")
113         plt.close()
114         plt.clf()
115
116         if self.hasValid:
117             # Valid Scatter Plot
118             figureCount = figureCount + 1
119             plt.figure(figureCount)
```

```
            plt.axes()
            title = validStr + typeScatterStr + paramStr
            plt.title(title)
            plt.xlabel(dimensionOneStr)
            plt.ylabel(dimensionTwoStr)
            colors = ['blue', 'red', 'green', 'black', 'yellow']
            plt.scatter(self.validData[:, 0], self.validData[:, 1],
             ↪  c=minAssignValid, s=10, alpha=0.5)
            colors = colors[:self.K]
            for i, j, k, l in zip(clusterMean,
             ↪  percentageValidAssignEachClass, colors,
             ↪  clusterStdDeviation[0]):
                plt.plot(i[0], i[1], 'kx', markersize=15, label=j, c=k)
                circle = plt.Circle((i[0], i[1]), radius=2*l, color=k,
                 ↪  fill=False)
                plt.gca().add_patch(circle)
            plt.legend()
            plt.savefig(self.questionTitle + title + ".png")
            plt.close()
            plt.clf()

    def PairwiseDistances(self, X, U):
        """
        input:
            X is a matrix of size (B x D)
            U is a matrix of size (K x D)
        output:
            Distances = matrix of size (B x K) containing the pairwise
↪  Euclidean distances
        """
        batchSize = tf.shape(X)[0]
        dimensionSize = tf.shape(X)[1]
        numClusters = tf.shape(U)[0]
        X_broadcast = tf.reshape(X, (batchSize, 1, dimensionSize))
        sumOfSquareDistances =
         ↪  tf.reduce_sum(tf.square(tf.subtract(X_broadcast, U)), 2)
```

```
150         return sumOfSquareDistances
151
152     def LnProbabilityXGivenZ(self, data, mean, variance):
153         sumOfSquare = self.PairwiseDistances(data, mean)
154         logLikelihoodDataGivenCluster =
              ↪  tf.add(-tf.multiply(tf.cast(self.D,
              ↪  tf.float32)/2.0,tf.log(tf.constant(2.0*np.pi)*variance)),
              ↪  -tf.divide(sumOfSquare, 2.0*variance))
155         return logLikelihoodDataGivenCluster
156
157     def LnProbabilityZGivenX(self, data, mean, variance, lnPriorBroad):
158         lnProbabilityXGivenZ = self.LnProbabilityXGivenZ(data, mean,
              ↪  variance)
159         # lnPriorBroad = tf.log(tf.reshape(prior, (1, self.K)))
160         numerator = lnPriorBroad + lnProbabilityXGivenZ
161         lnProbabilityX = tf.reshape(reduce_logsumexp(numerator, 1),
              ↪  (tf.shape(data)[0], 1))
162         lnProbabilityZGivenX = numerator - lnProbabilityX
163         return lnProbabilityZGivenX
164         # Monotonically increasing, others doesnt matter ??
165         # return numerator
166
167     def LnProbabilityX(self, data, mean, variance, lnPriorBroad):
168         lnProbabilityXGivenZ = self.LnProbabilityXGivenZ(data, mean,
              ↪  variance)
169         # lnPriorBroad = tf.log(tf.reshape(prior, (1, self.K)))
170         numerator = lnPriorBroad + lnProbabilityXGivenZ
171         lnProbabilityX = tf.reshape(reduce_logsumexp(numerator, 1),
              ↪  (tf.shape(data)[0], 1))
172         return lnProbabilityX
173
174     def MixtureOfGaussiansMethod(self):
175         '''
176         Build Graph and execute in here
177         so don't have to pass variables one by one
178         Bad Coding Style but higher programmer productivity
```

```
        '''
        # Build Graph
        # Mean location matters a lot in convergence
        clusterMean = tf.Variable(tf.truncated_normal([self.K, self.D],
         ↪  mean=-1, stddev=2.0)) # cluster centers
        clusterStdDeviationConstraint =
         ↪  tf.Variable(tf.truncated_normal([1, self.K], mean=0,
         ↪  stddev=0.1))
        clusterVariance = tf.exp(clusterStdDeviationConstraint)
        clusterStdDeviation = tf.sqrt(clusterVariance)
        # Uniform intialization
        clusterPriorConstraint = tf.Variable(tf.ones([1, self.K]))
        logClusterConstraint = logsoftmax(clusterPriorConstraint)
        clusterPrior = tf.exp(logClusterConstraint)

        trainData = tf.placeholder(tf.float32, shape=[None, self.D],
         ↪  name="trainingData")

        sumOfSquare = self.PairwiseDistances(trainData, clusterMean)
        lnProbabilityXGivenZ = self.LnProbabilityXGivenZ(trainData,
         ↪  clusterMean, clusterVariance)
        lnProbabilityX = self.LnProbabilityX(trainData, clusterMean,
         ↪  clusterVariance, logClusterConstraint)
        loss = (tf.reduce_sum(-1.0 * lnProbabilityX))
        # This is needed to decide which assignment it is
        lnProbabilityZGivenX = self.LnProbabilityZGivenX(trainData,
         ↪  clusterMean, clusterVariance, logClusterConstraint)
        probabilityZGivenX = tf.exp(lnProbabilityZGivenX)
        check = tf.reduce_sum(probabilityZGivenX, 1) # Check probabilities
         ↪  sum to 1
        # Assign classes based on maximum posterior probability for each
         ↪  data point
        minAssignments = tf.argmax(lnProbabilityXGivenZ, 1) # No prior
         ↪  contribution during assignment
        minAssignments = tf.argmax(lnProbabilityZGivenX, 1) # Prior
         ↪  contributes during assignment
```

```
204
205         #
              ↪ ----------------------------------------------------------------------------
206         #logLikelihoodDataGivenCluster =
              ↪ self.LnProbabilityZGivenX(trainData, clusterMean,
              ↪ clusterStdDeviation, clusterPrior)
207         validLoss = loss # initialization
208         minValidAssignments = minAssignments #Initialization
209         if self.hasValid:
210             valid_data = tf.placeholder(tf.float32, shape=[None, self.D],
                  ↪ name="validationData")
211             validLoss = tf.reduce_sum(-1.0 *
                  ↪ self.LnProbabilityX(valid_data,
                  ↪ clusterMean,clusterVariance,logClusterConstraint))
212             validLnProbabilityZGivenX =
                  ↪ self.LnProbabilityZGivenX(valid_data, clusterMean,
                  ↪ clusterVariance, logClusterConstraint)
213             minValidAssignments = tf.argmax(validLnProbabilityZGivenX, 1)
                  ↪ # Prior contributes during assignment
214
215         train = self.optimizer.minimize(loss)
216
217         # Session
218         init = tf.global_variables_initializer()
219         sess = tf.InteractiveSession()
220         sess.run(init)
221         currEpoch = 0
222         minAssignTrain = 0
223         minAssignValid = 0
224         centers = 0
225         xAxis = []
226         yTrainErr = []
227         yValidErr = []
228         numUpdate = 0
229         step = 0
230         currTrainDataShuffle = self.trainData
```

```
231         while currEpoch < self.numEpoch:
232             np.random.shuffle(self.trainData) # Shuffle Batches
233             step = 0
234             while step*self.miniBatchSize < self.trainData.shape[0]:
235                 feedDicts = {trainData:
                      ↪  self.trainData[step*self.miniBatchSize:(step+1)*self.miniBatchSize
236                 if self.hasValid:
237                     feedDicts = {trainData:
                          ↪  self.trainData[step*self.miniBatchSize:(step+1)*self.miniBatchS
                          ↪  valid_data: self.validData}
238                 _, minAssignTrain, paramClusterMean, paramClusterPrior,
                      ↪  paramClusterStdDeviation, zGivenX, checkZGivenX,
                      ↪  errTrain, errValid, minAssignValid = sess.run([train,
                      ↪  minAssignments, clusterMean, clusterPrior,
                      ↪  clusterStdDeviation, lnProbabilityZGivenX, check,
                      ↪  loss, validLoss, minValidAssignments], feed_dict =
                      ↪  feedDicts)
239                 xAxis.append(numUpdate)
240                 yTrainErr.append(errTrain)
241                 yValidErr.append(errValid)
242                 step += 1
243                 numUpdate += 1
244             currEpoch += 1
245 
246             if currEpoch%100 == 0:
247                 logStdOut("e: " + str(currEpoch))
248         # Calculate everything again without training
249         feedDicts = {trainData: self.trainData}
250         # No training, just gather data for valid assignments
251         if self.hasValid:
252             feedDicts = {trainData: self.trainData, valid_data:
                  ↪  self.validData}
253         minAssignTrain, paramClusterMean, paramClusterPrior,
              ↪  paramClusterStdDeviation, zGivenX, checkZGivenX, errTrain,
              ↪  errValid, minAssignValid = sess.run([minAssignments,
              ↪  clusterMean, clusterPrior, clusterStdDeviation,
              ↪  lnProbabilityZGivenX, check, loss, validLoss,
              ↪  minValidAssignments], feed_dict = feedDicts)
```

```
254         # Count how many assigned to each class
255         currTrainDataShuffle = self.trainData
256         self.printPlotResults(xAxis, yTrainErr, yValidErr, numUpdate,
              ↪  minAssignTrain, currTrainDataShuffle, paramClusterMean,
              ↪  paramClusterStdDeviation, paramClusterPrior, minAssignValid)
257
258 def executeMixtureOfGaussians(questionTitle, K, dataType, hasValid,
     ↪  numEpoch, learningRate):
259     """
260     Re-loads the data and re-randomize it with same seed anytime to ensure
     ↪  replicable results
261     """
262     logStdOut(questionTitle)
263     print questionTitle
264     trainData = 0
265     validData = 0
266     # Load data with seeded randomization
267     dataInitializer = DataInitializer()
268     if hasValid:
269         trainData, validData = dataInitializer.getData(dataType, hasValid)
270     else:
271         trainData = dataInitializer.getData(dataType, hasValid)
272
273     # Execute algorithm
274     kObject = MixtureOfGaussians(questionTitle, K, trainData, validData,
          ↪  hasValid, dataType, numEpoch, learningRate)
275     logElapsedTime(questionTitle + "K" + str(K) + "NumEpoch" +
          ↪  str(numEpoch))
276
277 # Global for logging
278 questionTitle = "" # Need to be global for logging to work
279 startTime = datetime.datetime.now()
280 figureCount = 1 # To not overwrite existing pictures
281
282 def logStdOut(message):
283     # Temporary print to std out
```

```
284     sys.stdout = sys.__stdout__
285     print message
286     # Continue editing same file
287     sys.stdout = open("result" + questionTitle + ".txt", "a")
288 
289 def logElapsedTime(message):
290     ''' Logs the elapsedTime with a given message '''
291     global startTime
292     endTime = datetime.datetime.now()
293     elapsedTime = endTime - startTime
294     hours, remainder = divmod(elapsedTime.seconds, 3600)
295     minutes, seconds = divmod(remainder, 60)
296     totalDays = elapsedTime.days
297     timeStr = str(message) + ': Days: ' + str(totalDays) +  " hours: " +
      ↪  str(hours) + ' minutes: ' + str(minutes) +  ' seconds: ' +
      ↪  str(seconds)
298     logStdOut(timeStr)
299     startTime = datetime.datetime.now()
300 
301 if __name__ == "__main__":
302     print "ECE521 Assignment 3: Unsupervised Learning: GaussianCluster"
303     '''
304     # Gaussian Cluster Model
305     questionTitle = "2.1.2" # Implemented function
306     questionTitle = "2.1.3" # Implemented FUnction
307     print "ECE521 Assignment 3: Unsupervised Learning: Mixture of
  ↪  Gaussian"
308     questionTitle = "2.2.2"
309     dataType = "2D"
310     hasValid = False # No validation data
311     K = 3
312     numEpoch = 200
313     learningRate = 0.1
314     # Note: Loss will be higher since no validation data
315     executeMixtureOfGaussians(questionTitle, K, dataType, hasValid,
  ↪  numEpoch, learningRate)
```

```
    # '''

    '''
    questionTitle = "2.2.3"
    dataType = "2D"
    hasValid = True
    diffK = [1, 2, 3, 4, 5]
    numEpoch = 200
    learningRate = 0.1
    for K in diffK:
        executeMixtureOfGaussians(questionTitle, K, dataType, hasValid,
 ↪  numEpoch, learningRate)
    # '''

    questionTitle = "2.2.4.2"
    dataType = "100D"
    hasValid = True
    diffK = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    numEpoch = 150
    learningRate = 0.1
    for K in diffK:
        executeMixtureOfGaussians(questionTitle, K, dataType, hasValid,
          ↪  numEpoch, learningRate)
    # '''
```

4.2.3 Helper Functions

```
import numpy as np
import tensorflow as tf

class DataInitializer(object):
    def __init__(self):
        """
        Data Initializer code
```

```
        """
        self.data2D = np.load("data2D.npy")
        self.data100D = np.load("data100D.npy")
        self.tinyData = np.load("tinymnist.npz")

    def getData(self, dataType, hasValid):
        """
        Returns the train, validation, and test data for 2D dataset
        that has already been randomized
        """
        data = 0
        if dataType == "2D":
            data = self.data2D
        elif dataType == "100D":
            data = self.data100D
        if hasValid:
            trainData, validData = self.splitDataRandom(data, hasValid)
            '''
            print 'data' + str(dataType) + ' All: (number of data, data
↪  dimension):', data.shape
            print 'data' + str(dataType) + ' Train: (number of data, data
↪  dimension):', trainData.shape
            print 'data' + str(dataType) + ' Valid: (number of data, data
↪  dimension):', validData.shape
            '''
            return trainData, validData
        trainData = self.splitDataRandom(data, hasValid)
        '''
        print 'data' + str(dataType) + ' All: (number of data, data
↪  dimension):', data.shape
        print 'data' + str(dataType) + ' Train: (number of data, data
↪  dimension):', trainData.shape
        '''
        return trainData

    def getTinyData(self):
```

```
        """
        Returns the train, validation, and test data for tinyMnist dataset
        that has already been randomized.
        """
        trainData, trainTarget = self.tinyData["x"], self.tinyData["y"]
        validData, validTarget = self.tinyData["x_valid"], self.tinyData
          ↪ ["y_valid"]
        testData, testTarget = self.tinyData["x_test"],
          ↪ self.tinyData["y_test"]
        print trainData.shape
        print trainTarget.shape
        print validData.shape
        print validTarget.shape
        print testData.shape
        print testTarget.shape
        # TODO: Randomize data?
        return trainData, trainTarget, validData, validTarget, testData,
          ↪ testTarget

    def splitDataRandom(self, data, hasValid):
        """
        Splits data using the ratio:
        i) If it hasValid
            66.6% training data
            33.3% validation data
        ii) No validation data
            100.0% training data
        """
        np.random.seed(521)
        randIdx = np.arange(len(data))
        np.random.shuffle(randIdx)
        if hasValid:
            trainStopPoint = int(np.ceil((2.0/3.0)*len(data)))
            # print trainStopPoint
            trainData, validData = data[randIdx[:trainStopPoint]],
              ↪ data[randIdx[trainStopPoint:]]
```

```
            return trainData, validData
        else:
            trainData = data[randIdx[:]]
            return trainData
```

4.2.4 FactorAnalysis

```
import tensorflow as tf
import numpy as np
import sys
from dataInitializer import DataInitializer
from utils import *
import datetime
import sys
import matplotlib.pyplot as plt

class FactorAnalysis(object):
    def __init__(self, questionTitle, K, trainData, trainTarget,
     ↪  validData, validTarget, testData, testTarget, numEpoch = 500,
     ↪  learningRate = 0.001):
        """
        Constructor
        """
        self.K = K # number of factors
        self.trainData = trainData
        self.trainTarget = trainTarget
        self.validData = validData
        self.validTarget = validTarget
        self.testData = testData
        self.testTarget = testTarget
        self.D = self.trainData[0].size # Dimension of each data
        self.learningRate = learningRate
        self.numEpoch = numEpoch
        self.miniBatchSize = self.trainData.shape[0] # miniBatchSize is
         ↪  entire data size
```

```
26          self.questionTitle = questionTitle
27          self.optimizer = tf.train.AdamOptimizer(learning_rate =
             ↪  self.learningRate, beta1=0.9, beta2=0.99, epsilon=1e-5)
28          # Execute Factor Analysis
29          self.FactorAnalysisMethod()
30
31      def saveGrayscaleImage(self, image, width=8, height=8, imageName=""):
32          """ This plots an image given its width and height
33          image is the image to plot
34          imageName is the name of the image to save as.
35          """
36          figureCount = 0 # TODO: Make global
37          plt.figure(figureCount)
38          # Draw each figures (8, 8)
39          currImage = image[:]
40          currImage = np.reshape(currImage, (width,height))
41          plt.imshow(currImage, interpolation="nearest", cmap="gray")
42          plt.savefig(str(imageName) + ".png")
43
44      def printTensor(self, tensorToPrint, trainData, message=""):
45          init = tf.global_variables_initializer()
46          sess = tf.InteractiveSession()
47          sess.run(init)
48          printDict = {trainData: self.trainData}
49          valueToPrint = sess.run([tensorToPrint], feed_dict = printDict)
50          print message, valueToPrint
51          print "shape", np.array(valueToPrint).shape
52          plt.close()
53          plt.clf()
54
55      def printPlotResults(self, xAxis, yTrainErr, yValidErr, numUpdate,
         ↪  currTrainDataShuffle, factorMean, factorCovariance,
         ↪  factorWeights):
56          figureCount = 0 # TODO: Make global
57          import matplotlib.pyplot as plt
58          print "mean", factorMean
```

```
59          print "K: ", self.K
60          print "Iter: ", numUpdate
61          print "mean", factorMean
62          print "meanShape", factorMean.shape
63          print "CoVariance", factorCovariance
64          print "CoVarianceShape", factorCovariance.shape
65          print "Lowest TrainLoss", np.min(yTrainErr)
66          print "Lowest ValidLoss", np.min(yValidErr)
67
68          trainStr = "Train"
69          validStr = "Valid"
70          typeLossStr = "Loss"
71          typeScatterStr = "Assignments"
72          trainLossStr = trainStr + typeLossStr
73          validLossStr = validStr + typeLossStr
74          iterationStr = "Iteration"
75          paramStr = "K" + str(self.K) + "Learn" + str(self.learningRate) +
             ↪  "NumEpoch" + str(self.numEpoch)
76
77          # Train Loss
78          figureCount = figureCount + 1
79          plt.figure(figureCount)
80          title = trainStr + typeLossStr + paramStr
81          plt.title(title)
82          plt.xlabel(iterationStr)
83          plt.ylabel(typeLossStr)
84          plt.plot(np.array(xAxis), np.array(yTrainErr), label =
             ↪  trainLossStr)
85          plt.legend()
86          plt.savefig(self.questionTitle + title + ".png")
87          plt.close()
88          plt.clf()
89
90          # Valid Loss
91          figureCount = figureCount + 1
92          plt.figure(figureCount)
```

```
93          title = validStr + typeLossStr + paramStr
94          plt.title(title)
95          plt.xlabel(iterationStr)
96          plt.ylabel(typeLossStr)
97          plt.plot(np.array(xAxis), np.array(yValidErr), label =
             ↪  validLossStr)
98          plt.legend()
99          plt.savefig(self.questionTitle + title + ".png")
100         plt.close()
101         plt.clf()
102
103         # Weight Images
104         for i in xrange(self.K):
105             imageTitle = self.questionTitle + "WeightDim" + str(i) + "K" +
                 ↪  str(self.K) +  "NumEpoch" + str(self.numEpoch)
106             self.saveGrayscaleImage(factorWeights[:, i], 8, 8, imageTitle)
107
108     def FactorAnalysisMethod(self):
109         '''
110         Build Graph and execute in here
111         so don't have to pass variables one by one
112         Bad Coding Style but higher programmer productivity
113         '''
114         trainData = tf.placeholder(tf.float32, shape=[None, self.D],
             ↪  name="trainingData")
115         validData = tf.placeholder(tf.float32, shape=[None, self.D],
             ↪  name="validationData")
116         # Build Graph
117         print "trainShape", self.trainData.shape
118         print "validShape", self.validData.shape
119         print "testShape", self.testData.shape
120         factorMean = tf.Variable(tf.truncated_normal([self.D]))
121         factorWeights = tf.Variable(tf.truncated_normal([self.D, self.K]))
122         factorStdDeviationConstraint =
             ↪  tf.Variable(tf.truncated_normal([self.D]))
123         factorTraceCoVariance = tf.exp(factorStdDeviationConstraint)
```

```
        factorCovariance = tf.diag(factorTraceCoVariance) +
         ↪  tf.matmul(factorWeights, tf.transpose(factorWeights))
        factorCovarianceInv = tf.matrix_inverse(factorCovariance)
        logDeterminantCovariance = 2.0 *
         ↪  tf.reduce_sum(tf.log(tf.diag_part(tf.cholesky(factorCovariance))))

        # Train Loss
        xDeductU = tf.subtract(trainData, factorMean)
        xDeductUTranspose = tf.transpose(xDeductU)
        total = tf.trace(tf.matmul(tf.matmul(xDeductU,
         ↪  factorCovarianceInv), xDeductUTranspose))
        logProbability = -0.5 * (total + self.D * tf.log(2.0 * np.pi) +
         ↪  logDeterminantCovariance)
        loss = tf.negative(logProbability)

        validDeductU = tf.subtract(validData, factorMean)
        validDeductUTranspose = tf.transpose(validDeductU)
        validTotal = tf.trace(tf.matmul(tf.matmul(validDeductU,
         ↪  factorCovarianceInv), validDeductUTranspose))
        validLogProbability = -0.5 * (validTotal + self.D * tf.log(2.0 *
         ↪  np.pi) + logDeterminantCovariance)
        validLoss = tf.negative(validLogProbability)

        train = self.optimizer.minimize(loss)
        # Session
        init = tf.global_variables_initializer()
        sess = tf.InteractiveSession()
        sess.run(init)
        currEpoch = 0
        minAssignTrain = 0
        minAssignValid = 0
        centers = 0
        xAxis = []
        yTrainErr = []
        yValidErr = []
        numUpdate = 0
```

```
154         step = 0
155         currTrainDataShuffle = self.trainData
156         while currEpoch < self.numEpoch:
157             np.random.shuffle(self.trainData) # Shuffle Batches
158             step = 0
159             while step*self.miniBatchSize < self.trainData.shape[0]:
160                 feedDicts = {trainData:
                      ↪  self.trainData[step*self.miniBatchSize:(step+1)*self.miniBatchSize
                      ↪  validData: self.validData}
161                 _, errTrain, errValid, paramFactorMean,
                      ↪  paramFactorCovariance, paramFactorWeights =
                      ↪  sess.run([train, loss, validLoss, factorMean,
                      ↪  factorCovariance, factorWeights], feed_dict =
                      ↪  feedDicts)
162                 xAxis.append(numUpdate)
163                 yTrainErr.append(errTrain)
164                 yValidErr.append(errValid)
165                 step += 1
166                 numUpdate += 1
167             currEpoch += 1
168 
169             if currEpoch%100 == 0:
170                 logStdOut("e: " + str(currEpoch))
171         # Calculate everything again without training
172         feedDicts = {trainData: self.trainData, validData: self.validData}
173         errTrain, errValid, paramFactorMean, paramFactorCovariance,
              ↪  paramFactorWeights = sess.run([loss, validLoss, factorMean,
              ↪  factorCovariance, factorWeights], feed_dict = feedDicts)
174         # Count how many assigned to each class
175         currTrainDataShuffle = self.trainData
176         self.printPlotResults(xAxis, yTrainErr, yValidErr, numUpdate,
              ↪  currTrainDataShuffle, paramFactorMean, paramFactorCovariance,
              ↪  paramFactorWeights)
177 
178 def executeFactorAnalysis(questionTitle, K, numEpoch, learningRate):
179     """
```

```
180     Re-loads the data and re-randomize it with same seed anytime to ensure
 ↪  replicable results
181     """
182     logStdOut(questionTitle)
183     print questionTitle
184     trainData = 0
185     validData = 0
186     # Load data with seeded randomization
187     dataInitializer = DataInitializer()
188     trainData, trainTarget, validData, validTarget, testData, testTarget =
 ↪  dataInitializer.getTinyData()
189 
190     # Execute algorithm
191     kObject = FactorAnalysis(questionTitle, K, trainData, trainTarget,
 ↪  validData, validTarget, testData, testTarget, numEpoch,
 ↪  learningRate)
192     logElapsedTime(questionTitle + "K" + str(K) + "NumEpoch" +
 ↪  str(numEpoch))
193 
194 # Global for logging
195 questionTitle = "" # Need to be global for logging to work
196 startTime = datetime.datetime.now()
197 figureCount = 1 # To not overwrite existing pictures
198 
199 def logStdOut(message):
200     # Temporary print to std out
201     # sys.stdout = sys.__stdout__ # TODO: Uncomment this
202     print message
203     # Continue editing same file
204     # sys.stdout = open("result" + questionTitle + ".txt", "a") #TODO:
 ↪  Uncomment this
205 
206 def logElapsedTime(message):
207     ''' Logs the elapsedTime with a given message '''
208     global startTime
209     endTime = datetime.datetime.now()
```

```
210     elapsedTime = endTime - startTime
211     hours, remainder = divmod(elapsedTime.seconds, 3600)
212     minutes, seconds = divmod(remainder, 60)
213     totalDays = elapsedTime.days
214     timeStr = str(message) + ': Days: ' + str(totalDays) +  " hours: " +
          ↪  str(hours) + ' minutes: ' + str(minutes) +  ' seconds: ' +
          ↪  str(seconds)
215     logStdOut(timeStr)
216     startTime = datetime.datetime.now()
217
218 if __name__ == "__main__":
219     print "ECE521 Assignment 3: Unsupervised Learning: Factor Analysis"
220     questionTitle = "3.1.2"
221     numEpoch = 250
222     learningRate = 0.1
223     K = 4
224     executeFactorAnalysis(questionTitle, K, numEpoch, learningRate)
225     # '''
226
227     '''
228     questionTitle = "3.1.3"
229     diffK = [1, 2, 3, 4, 5]
230     # for K in diffK:
231         executeFactorAnalysis(questionTitle, K, numEpoch, learningRate)
232     # TODO:
233     # '''
```