ECE521 Assignment 3 Report

# Row Division:

For this assignment, both Wei Cui and Leon Chen have been working closely, collaborating on deriving proofs, coding, and testing results. Lots of work has been done parallely. Thus, both team members contributed 50%​ of the assignment.

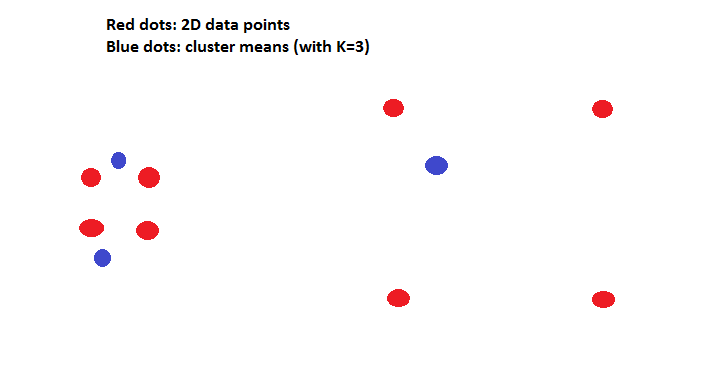
# 1 K-means

## 1.1 Learning K-means

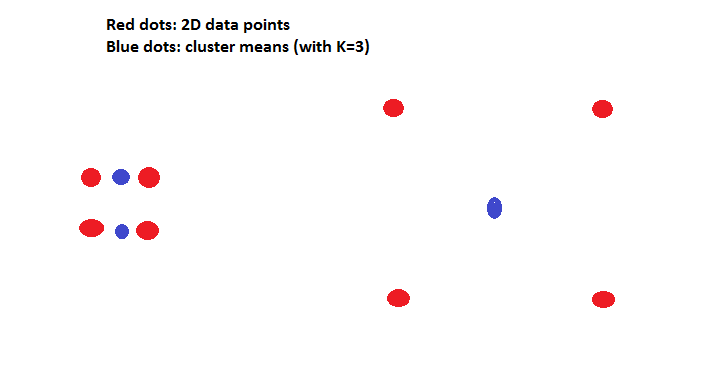
### Part 1

The loss function is not convex in 𝝁. To prove that it is not convex in general, it suffices to consider a special case.

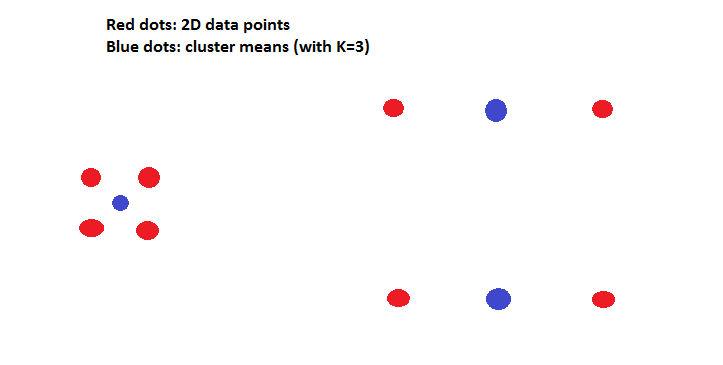
Consider the 2D cases with points located as following. Assume the initialization points for the cluster means are shown below:



After several optimization steps, the given initialization would converge to the below stable state:



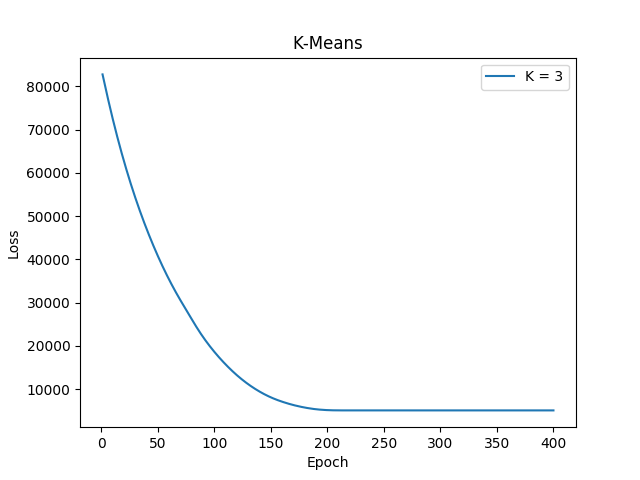
However, the global optimum for the above 2D data distribution would be shown as following:



Prove by contradiction:

If the loss function is convex, then despite of the initialization, the converging solution would always be a global optimum. However, as above, the case shows an initialization of 𝝁 converging to a non-global optimum. Thus, the loss function is not convex on 𝝁.

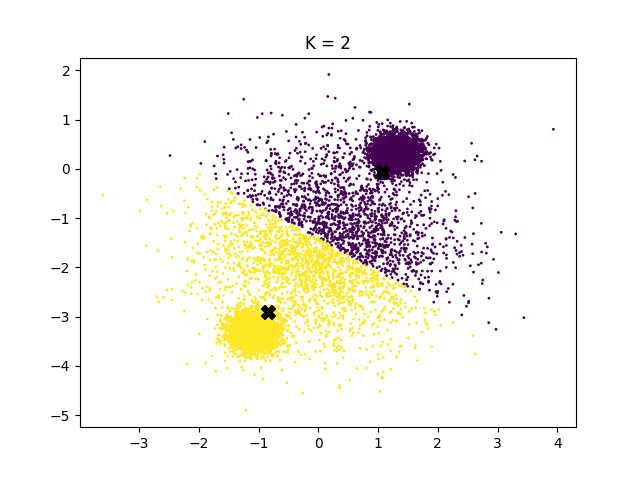
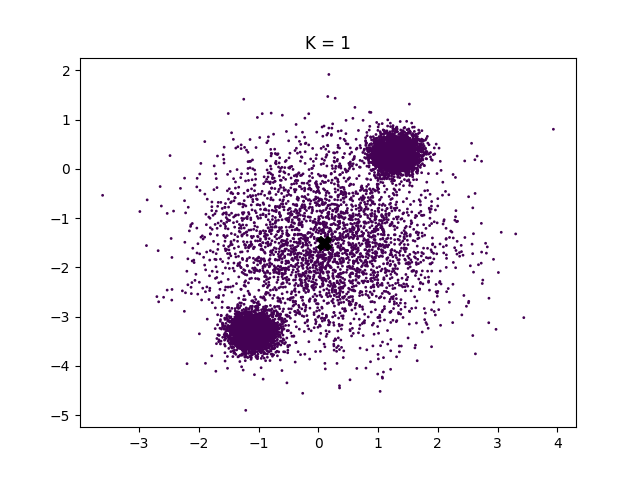
### Part 2

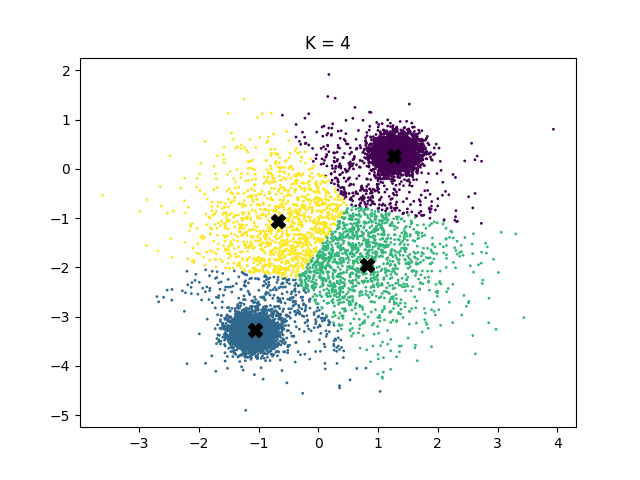
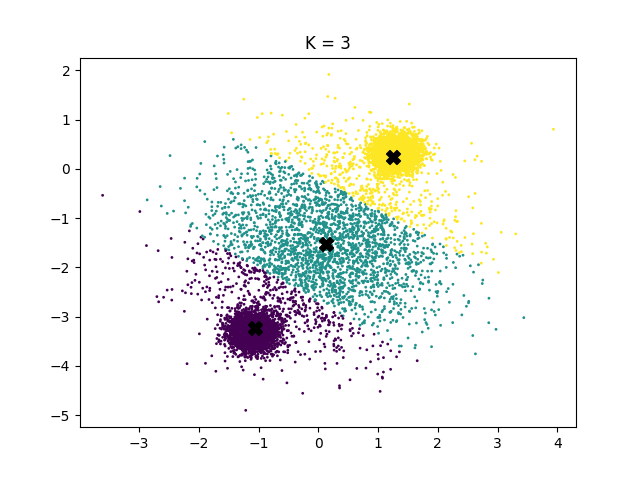


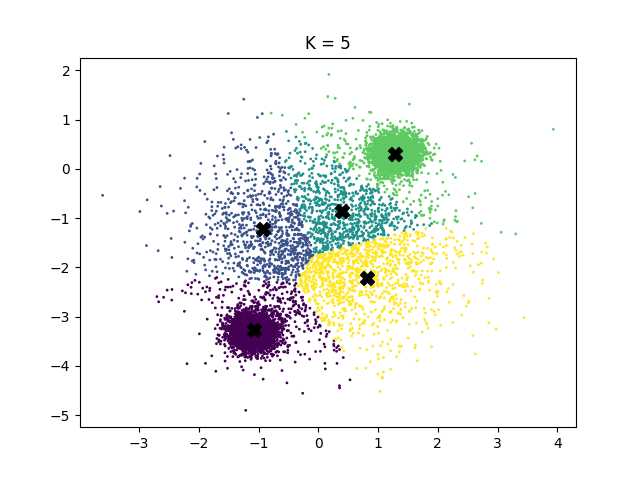
*Loss vs. the number of updates*

As it can be seen, the model converges after around 200 updates. This is likely the limit of a three cluster model; further training does not appear to yield better results.

### Part 3







*Cluster assignments for K = 1, 2, 3, 4, 5*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **K = 1** | **K = 2** | **K = 3** | **K = 4** | **K = 5** |
| **Cluster 1** | 100% | 50.45% | 38.15% | 37.13% | 7.47% |
| **Cluster 2** |  | 49.55% | 38.06% | 12.09% | 35.90\* |
| **Cluster 3** |  |  | 23.79% | 13.49% | 8.56% |
| **Cluster 4** |  |  |  | 37.29% | 36.34% |
| **Cluster 5** |  |  |  |  | 11.73% |

*Percentage of data assigned to each cluster*

Judging from the cluster plots, the best K would either be 3 or 4. At 3 or 4 clusters, each cluster is responsible for a sizable portion of the data. If we go higher and use five clusters, it appears that several clusters are assigned a very small percentage of the data, making it redundant.

### Part 4

For K = {1, 2, 3, 4, 5}

* K=1: Validation Loss is: 12969.3
* K=2: Validation Loss is: 3065.86
* K=3: Validation Loss is: 1693.27
* K=4: Validation Loss is: 1100.42
* K=5: Validation Loss is: 935.744

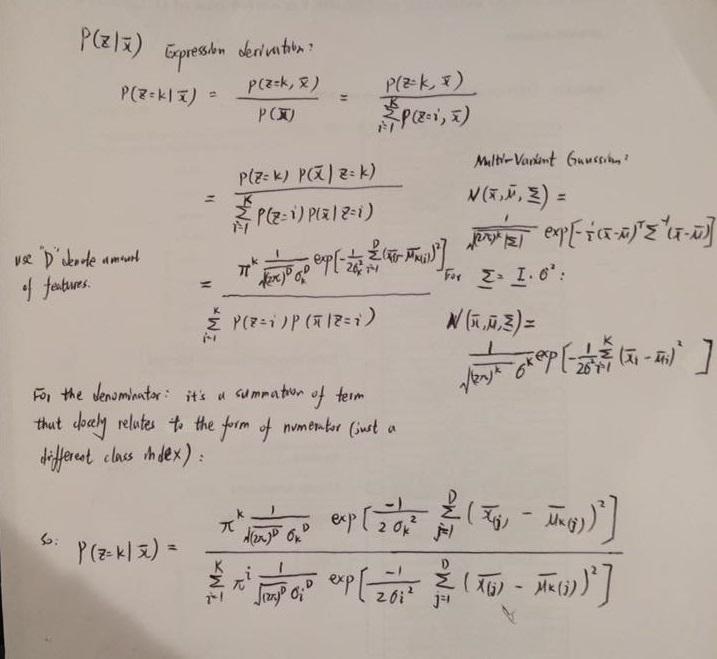
Looking at the validation loss, it is appears 4 and 5 are the best values to use. Though the cluster plot for K=5 seemed like it was overfitting, the validation showed that it performed much better than K=3. In fact K=5 is the best K to use for this data set.

# 2 Mixtures of Gaussians

## 2.1 The Gaussian cluster model

### Part 1

The expression for the posterior distribution is derived and shown in the following proof:



### 

### Part 2

The code snippet is shown as following:

# constants  
PI = np.pi  
def log\_likelihood(data, mean, stddev):  
 # data: BXD; mean: KXD; variance: K (given uni-standard deviation assumption)  
 K, D = np.shape(mean)  
 log\_prob = []  
 for i in range(K):  
 dists = data - mean[i] # through broadcasting, shape is BXD  
 gaussian\_coefficient = -D/2\*np.log(2\*PI) - D\*np.log(stddev[i])  
 # Taking the log should turn into summation terms; result is with dimension: BX1  
 log\_prob\_per\_class = gaussian\_coefficient \  
 + (-1/(2\*np.power(stddev[i],2))\*np.sum(np.power(dists,2),axis=1))  
 log\_prob.append(log\_prob) # shape: KXB  
 # result takes on the shape: KXB, transform it for more natural representation  
 log\_prob = np.transpose(log\_prob, [1,0]) # BXK  
 return log\_prob

### Part 3

The code snippet is shown as following:

Note: the code snippet below didn’t assume on that the pi vector is already computed in log scale (which is the case with utils.py provided).

def log\_prob\_cluster(data, mean, stddev, cluster, pi\_vector):  
 # assume pi\_vector: 1XK  
 # compute the log probability for "cluster" specified by the parameter  
 K, D = np.shape(mean)  
 # compute term for each cluster (p(x|z) added with a prior)  
 log\_joint\_prob = []  
 log\_likelihood\_prob = log\_likelihood(data, mean, stddev) # BXK  
 # compute the log prior   
 log\_prob\_prior = np.log(pi\_vector)   
 log\_prob\_posterior\_unnormalized = log\_prob\_prior + log\_prob\_likelihood # BXK  
 # now ready to compute the probability on posterior for each cluster  
 log\_prob\_posterior = log\_prob\_posterior\_unnormalized \  
 / reduce\_logsumexp(log\_prob\_posterior\_unnormalized, axis=1) # BXK  
 return log\_prob\_posterior

It is important to use the log-sum-exp function as if we simply take the log of the exponential, it is numerically unstable. This is because the data can be arbitrarily large, so the exponential could cause an overflow. The LSE function takes care of such cases, using a numerically stable approach that does not take the exponentials directly, so the overflow does not happen.

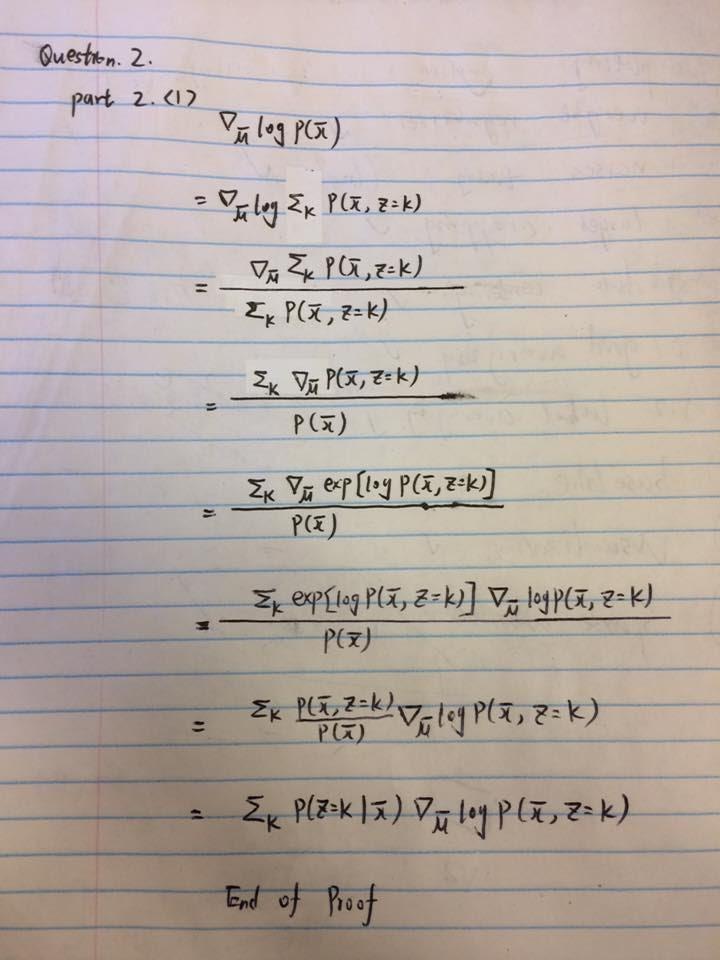
## 

## 

## 2.2 Learning the MoG

### Part 1

The proof is shown as following:



### Part 2

The logsoftmax() function provided by utils.py returns a log probability so, unlike in 2.1, we assume a log prior (pi vector) for this portion.

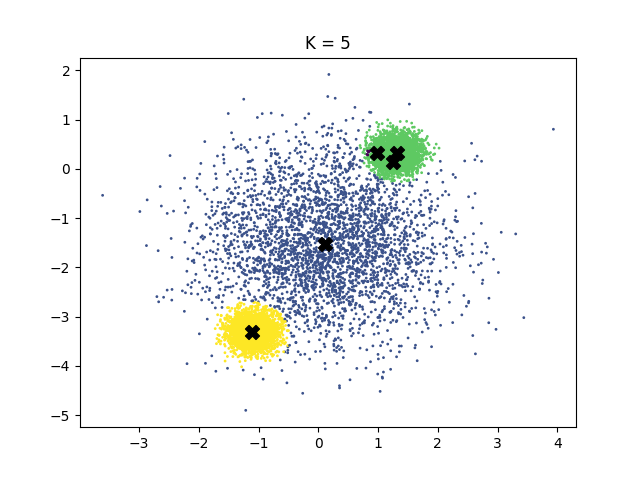
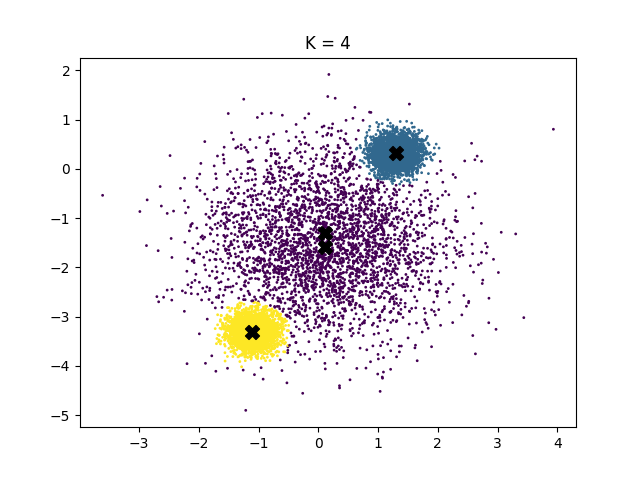
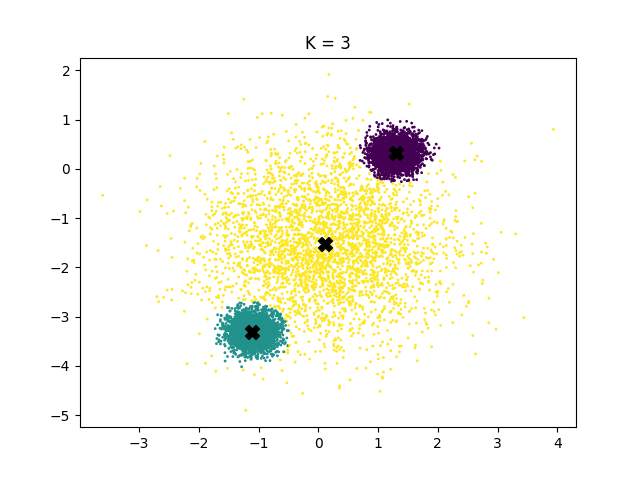
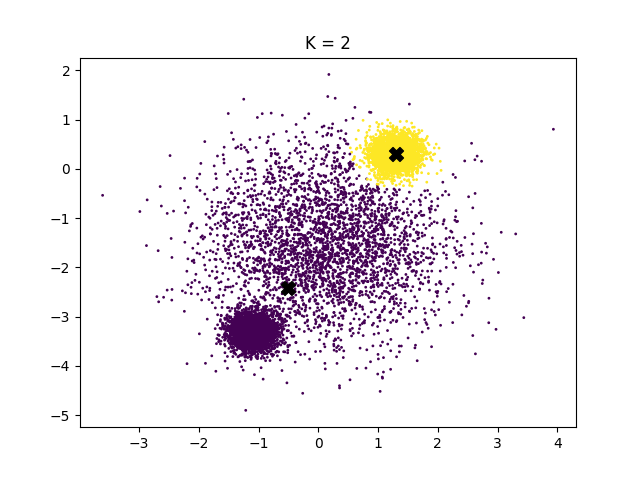
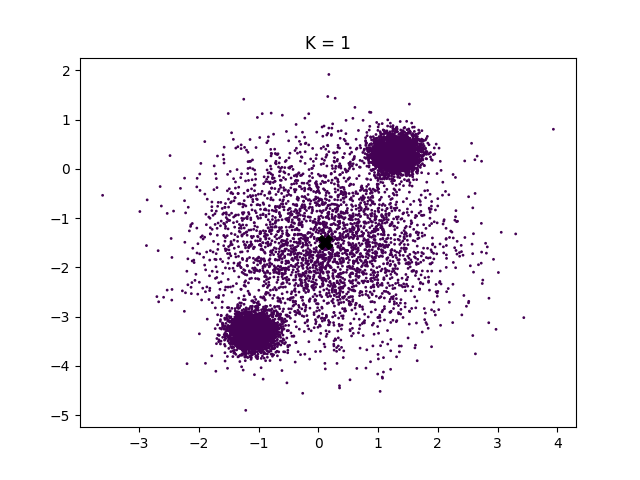
### 

*Loss vs. the number of updates*

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Mean** | **Pi** | **Sigma** |
| **Cluster 1** | 1.29872644, 0.30861631 | 0.33347192 | 0.19713925 |
| **Cluster 2** | 0.10580099, -1.52707195 | 0.33485365 | 0.99359304 |
| **Cluster 3** | -1.10194325, -3.30602694 | 0.33167449 | 0.19773863 |

Looking at these results, it can be seen that the 3 cluster are responsible for around the same amount of data point. However, the middle cluster has a much higher variance than the other two clusters, which matches our intuition.

### Part 3



*Cluster assignments for K = 1, 2, 3, 4, 5*

For K = {1, 2, 3, 4, 5}

* K=1: Validation Loss is: 11679.9
* K=2: Validation Loss is: 8051.4
* K=3: Validation Loss is: 5584.26
* K=4: Validation Loss is: 5584.37
* K=5: Validation Loss is: 5586.01

It appears that K values higher than 3 all give the same loss. This can be explained by the 2D cluster plots which shows that for K=4 and K=5 three of the cluster are responsible for most of the data points. Hence, extra cluster are rendered redundant.

### Part 4

For K-means, K = {5, 10, 15, 20, 25}

|  |  |  |
| --- | --- | --- |
|  | **Validation Loss** | **Dominate Clusters** |
| **K=5** | 71849.0 | 5 |
| **K=10** | 71849.0 | 5 |
| **K=15** | 71848.6 | 5 |
| **K=20** | 71849.0 | 5 |
| **K=25** | 71808.7 | 5 |

K-means is converging to 5 clusters, the loss was the same across different K values.

For Mixture of Gaussians, K = {5, 10, 15, 20, 25}

|  |  |  |
| --- | --- | --- |
|  | **Validation Loss** | **Dominate Clusters** |
| **K=5** | 277512.0 | 4 |
| **K=10** | 161356.0 | 5 |
| **K=15** | 161356.0 | 5 |
| **K=20** | 161358.0 | 5 |
| **K=25** | 161359.0 | 5 |

It appears the data was generated by 5 clusters. However, setting K=5 gave the worst results.

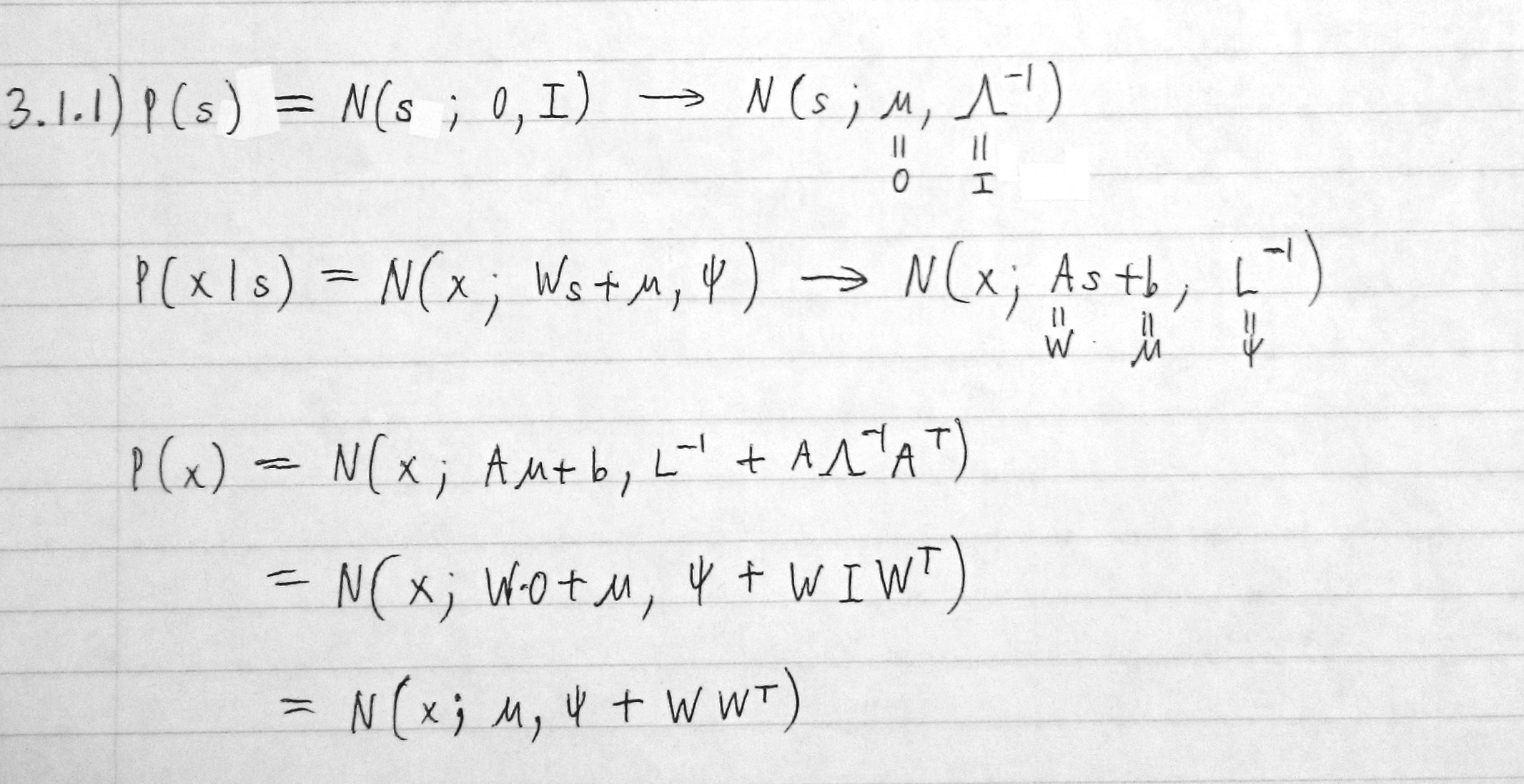
Running the K=5 multiple times reveals the issue. The model with five clusters would often converge to a state with 3 or 4 dominate clusters, leaving the model with less flexibility. With the higher K values, 5 clusters consistently dominate most of the data.

With fewer clusters, it is very easy for the cluster get stuck in a local minimum. Perhaps this shows that instead of limiting the model capabilities, one should let the model decide if there are redundant parameters.

# 3 Discover Latent Dimensions

## 3.1 Factor Analysis

### Part 1



As , .

### 

### 

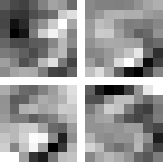
### Part 2

The final negative log likelihoods are:

* Training: -8656.86
* Validation: -1266.68
* Testing: -1229.76

Clearly the training set has much lower error than validation or testing.

After training, the plots for weights are as following:



The bottom two weight looks like 3 and 5, but the top two appears to the the overlap between 3 and 5. It is likely 3 and 5 share certain correlations in their features and the factor analysis learned these similar features instead of the numbers themselves. This is expected as this is unsupervised learning so the 3 and 5 digit are not explicitly separated by labels.

In terms of the latent dimensions, notice that both 5 and 3 have a curved at the lower right corner. The top left corner is also highly weighted, as both 3 and 5 overlap there. However, as they approach the top left corner from different direction, the latent dimension looks more like a corner than a curve.

### Part 3

The PCA returns a vector of the value (0.01310373, 0.01324098, 0.99982647). As expected the dimension is the largest value as it is the dimension of maximum variance.

The Factor Analysis returns a vector of value (-3.02042055, -3.03099427, 0.05865546). The and components of the vector much larger in magnitude than , as that is the direction of highest correlation. Since the and direction are both similar in magnitude, it shows the maximum correlation is in the direction.

# Appendix A: K-means

## 1.1.2

# ECE521 A3  
# March 18th 2017  
# Wei Cui Leon Chen  
# Question 1.1 part (2)  
  
import matplotlib.pyplot as plt  
import numpy as np  
import tensorflow as tf  
  
# parameters  
K = 3  
feature\_amount = 2  
epoch\_amount = 400  
batch\_size = 500  
# hyper-parameters  
learning\_rate = 0.001  
  
# Tensorflow graph  
data = tf.placeholder(tf.float32, shape=[None, feature\_amount], name="data\_placeholder")  
with tf.name\_scope("2D\_mean"):  
 mean = tf.Variable(tf.random\_normal([K, feature\_amount]), name="mean")  
# compute the loss given the mean  
sub\_array = []  
for i in range(K):  
 sub\_array\_per\_class = tf.reduce\_sum(tf.pow((data - mean[i]), 2), axis=1)  
 sub\_array.append(sub\_array\_per\_class)  
sub\_array = tf.reduce\_min(sub\_array, axis=0)  
loss = tf.reduce\_sum(sub\_array)  
optimizer = tf.train.AdamOptimizer(learning\_rate, beta1=0.9, beta2=0.99, epsilon=1e-5)  
train\_step = optimizer.minimize(loss)  
  
  
if(\_\_name\_\_ == '\_\_main\_\_'):  
 # load dataset from data2D.npy  
 data2D = np.load('data2D.npy') #should be a 2D array with size 10000 X 2  
 # parse data into minibatches  
 # assuming batch size is divisible by the amount of data  
 data2D = np.reshape(data2D,[-1, batch\_size, feature\_amount])  
  
 batch\_amount = np.shape(data2D)[0]  
  
 total\_loss = np.zeros(epoch\_amount)  
 saver = tf.train.Saver()  
 with tf.Session() as sess:  
 sess.run(tf.global\_variables\_initializer())  
 for i in range(epoch\_amount):  
 epoch\_loss = 0  
 for j in range(batch\_amount):  
 batch\_loss, \_ = sess.run([loss, train\_step], feed\_dict={data: data2D[j]})  
 epoch\_loss += batch\_loss  
 total\_loss[i] = epoch\_loss  
 if((i+1) % 20 == 0):  
 print("At epoch{}, loss across all data: {}".format(i, epoch\_loss))  
 print("Model training complete!")  
 save\_path = saver.save(sess, "trained\_model/model\_1\_1\_2.ckpt")  
 print("Trained model saved in file: {}".format(save\_path))  
  
 print("Script completed successfully!")  
 # Plot the data  
 plt.figure(1)  
 plt.plot(range(1, epoch\_amount+1), total\_loss, label='K = {}'.format(K))  
 plt.ylabel('Loss')  
 plt.xlabel('Epoch')  
 plt.title('K-Means')  
 plt.legend(loc='upper right')  
 plt.show()

## 1.1.3

# ECE521 A3  
# March 18th 2017  
# Wei Cui Leon Chen  
# Question 1.1 part (3)  
  
import matplotlib.pyplot as plt  
import numpy as np  
import tensorflow as tf  
  
# parameters  
K = 5  
feature\_amount = 2  
epoch\_amount = 400  
batch\_size = 500  
# hyper-parameters  
learning\_rate = 0.001  
  
# Tensorflow graph  
data = tf.placeholder(tf.float32, shape=[None, feature\_amount], name="data\_placeholder")  
with tf.name\_scope("2D\_mean"):  
 mean = tf.Variable(tf.random\_normal([K, feature\_amount]), name="mean")  
# compute the loss given the mean  
sub\_array = []  
for i in range(K):  
 sub\_array\_per\_class = tf.reduce\_sum(tf.pow((data - mean[i]), 2), axis=1)  
 sub\_array.append(sub\_array\_per\_class)  
assignments = tf.argmin(sub\_array, axis=0)  
sub\_array = tf.reduce\_min(sub\_array, axis=0)  
loss = tf.reduce\_sum(sub\_array)  
optimizer = tf.train.AdamOptimizer(learning\_rate, beta1=0.9, beta2=0.99, epsilon=1e-5)  
train\_step = optimizer.minimize(loss)  
  
  
if(\_\_name\_\_ == '\_\_main\_\_'):  
 # load dataset from data2D.npy  
 data2D = np.load('data2D.npy') #should be a 2D array with size 10000 X 2  
 orig\_data = np.copy(data2D)  
 n\_val = data2D.shape[0]  
 # parse data into minibatches  
 # assuming batch size is divisible by the amount of data  
 data2D = np.reshape(data2D,[-1, batch\_size, feature\_amount])  
  
 batch\_amount = np.shape(data2D)[0]  
  
 total\_loss = np.zeros(epoch\_amount)  
 centroids = np.zeros(K)  
 saver = tf.train.Saver()  
 with tf.Session() as sess:  
 sess.run(tf.global\_variables\_initializer())  
 for i in range(epoch\_amount):  
 epoch\_loss = 0  
 assign = []  
 for j in range(batch\_amount):  
 centroids, batch\_loss, assign\_batch, \_ = sess.run([mean, loss, assignments, train\_step], feed\_dict={data: data2D[j]})  
 epoch\_loss += batch\_loss  
 total\_loss[i] = epoch\_loss  
 assign.append(assign\_batch)  
 if((i+1) % 20 == 0):  
 print("At epoch{}, loss across all data: {}".format(i, epoch\_loss))  
 print("Model training complete!")  
 save\_path = saver.save(sess, "trained\_model/model\_1\_1\_3.ckpt")  
 print("Trained model saved in file: {}".format(save\_path))  
  
 print("Script completed successfully!")  
  
 # Percentage assigned to each cluster  
 for cluster in range(K):  
 print('{}% assigned to cluster {}'.format(round(np.mean(np.array(assign) == cluster)\*100, 2), cluster+1))  
  
 # Plot the data  
 plt.figure(1)  
 plt.scatter(orig\_data[:, 0], orig\_data[:, 1], c=assign, s=1)  
 plt.scatter(centroids[:, 0], centroids[:, 1], marker='X', color='black', s=100)  
 plt.title('K = {}'.format(K))  
 plt.show()

## 1.1.4

# ECE521 A3  
# March 18th 2017  
# Wei Cui Leon Chen  
# Question 1.1 part (4)  
  
import matplotlib.pyplot as plt  
import numpy as np  
import tensorflow as tf  
  
# parameters  
K = 5  
feature\_amount = 2  
epoch\_amount = 400  
batch\_size = 303  
# hyper-parameters  
learning\_rate = 0.001  
  
# Tensorflow graph  
data = tf.placeholder(tf.float32, shape=[None, feature\_amount], name="data\_placeholder")  
with tf.name\_scope("2D\_mean"):  
 mean = tf.Variable(tf.random\_normal([K, feature\_amount]), name="mean")  
# compute the loss given the mean  
sub\_array = []  
for i in range(K):  
 sub\_array\_per\_class = tf.reduce\_sum(tf.pow((data - mean[i]), 2), axis=1)  
 sub\_array.append(sub\_array\_per\_class)  
assignments = tf.argmin(sub\_array, axis=0)  
sub\_array = tf.reduce\_min(sub\_array, axis=0)  
loss = tf.reduce\_sum(sub\_array)  
optimizer = tf.train.AdamOptimizer(learning\_rate, beta1=0.9, beta2=0.99, epsilon=1e-5)  
train\_step = optimizer.minimize(loss)  
  
if(\_\_name\_\_ == '\_\_main\_\_'):  
 # load dataset from data2D.npy  
 data2D = np.load('data2D.npy') #should be a 2D array with size 10000 X 2  
 tr\_data = data2D[:int(data2D.shape[0]\*2/3)]  
 va\_data = data2D[int(data2D.shape[0]\*2/3):]  
 n\_tr = tr\_data.shape[0]  
 n\_va = va\_data.shape[0]  
 # parse data into minibatches  
 # assuming batch size is divisible by the amount of data  
 tr\_data = np.reshape(tr\_data, [-1, batch\_size, feature\_amount])  
 batch\_amount = np.shape(tr\_data)[0]  
  
 train\_loss = np.zeros(epoch\_amount)  
 centroids = np.zeros(K)  
 saver = tf.train.Saver()  
 with tf.Session() as sess:  
 sess.run(tf.global\_variables\_initializer())  
 for i in range(epoch\_amount):  
 epoch\_loss = 0  
 assign = []  
 for j in range(batch\_amount):  
 centroids, batch\_loss, assign\_batch, \_ = sess.run([mean, loss, assignments, train\_step], feed\_dict={data: tr\_data[j]})  
 epoch\_loss += batch\_loss  
 train\_loss[i] = epoch\_loss  
 assign.append(assign\_batch)  
 if((i+1) % 20 == 0):  
 print("At epoch{}, loss across all data: {}".format(i, epoch\_loss))  
 print("Model training complete!")  
 save\_path = saver.save(sess, "trained\_model/model\_1\_1\_4.ckpt")  
 print("Trained model saved in file: {}".format(save\_path))  
  
 # Validation  
 valid\_loss = sess.run(loss, feed\_dict={data: va\_data})  
 print('Validation Loss is: ', valid\_loss)  
  
 print("Script completed successfully!")

# Appendix B: Mixture of Gaussians

## 2.1

# ECE521 A3  
# March 18th 2017  
# Wei Cui Leon Chen  
# Question 2.1 Part (2)  
  
import numpy as np  
import tensorflow as tf   
from utils import \*   
  
# parameters  
# constants  
PI = np.pi  
def log\_likelihood(data, mean, stddev):  
 # data: BXD; mean: KXD; variance: K (given uni-standard deviation assumption)  
 K, D = np.shape(mean)  
 log\_prob = []  
 for i in range(K):  
 dists = data - mean[i] # through broadcasting, shape is BXD  
 gaussian\_coefficient = -D/2\*np.log(2\*PI) - D\*np.log(stddev[i])  
 # Taking the log should turn into summation terms; result is with dimension: BX1  
 log\_prob\_per\_class = gaussian\_coefficient + (-1/(2\*np.power(stddev[i],2))\*np.sum(np.power(dists,2),axis=1))  
 log\_prob.append(log\_prob) # shape: KXB  
 # result takes on the shape: KXB, transform it for more natural representation  
 log\_prob = np.transpose(log\_prob, [1,0]) # BXK  
 return log\_prob  
  
def log\_prob\_cluster(data, mean, stddev, cluster, pi\_vector):  
 # assume pi\_vector: 1XK  
 # compute the log probability for "cluster" specified by the parameter  
 K, D = np.shape(mean)  
 # compute term for each cluster (p(x|z) added with a prior)  
 log\_joint\_prob = []  
 log\_likelihood\_prob = log\_likelihood(data, mean, stddev) # BXK  
 # compute the log prior   
 log\_prob\_prior = np.log(pi\_vector)   
 log\_prob\_posterior\_unnormalized = log\_prob\_prior + log\_prob\_likelihood # BXK  
 # now ready to compute the probability on posterior for each cluster  
 log\_prob\_posterior = log\_prob\_posterior\_unnormalized / reduce\_logsumexp(log\_prob\_posterior\_unnormalized, axis=1) # BXK  
 return log\_prob\_posterior  
  
if(\_\_name\_\_ == '\_\_main\_\_'):  
 # load dataset from data2D.npy  
 data2D = np.load('data2D.npy') #should be a 2D array with size 10000 X 2

## 2.2.2

# ECE521 A3  
# March 23th 2017  
# Wei Cui Leon Chen  
# Question 2.2.2  
  
import numpy as np  
import tensorflow as tf  
import matplotlib.pyplot as plt  
from utils import \*  
  
# parameters  
MINIBATCH\_SIZE = 500  
# hyper-parameters  
LEARNING\_RATE = 0.001  
# constants  
PI = np.pi  
K = 3  
FEATURE\_AMOUNT = 2  
EPOCH\_AMOUNT = 500  
  
# For 2.1  
def log\_likelihood(data, mean, stddev\_vector):  
 # data: BXD; mean: KXD; variance: K (given uni-standard deviation assumption)  
 log\_probs = []  
 for i in range(K):  
 dists = data - mean[i] # through broadcasting, shape is BXD  
 gaussian\_coefficient = -FEATURE\_AMOUNT/2\*tf.log(2\*PI) - FEATURE\_AMOUNT\*tf.log(stddev\_vector[0,i])  
 # Taking the log should turn into summation terms; result is with dimension: BX1  
 log\_prob\_per\_class = gaussian\_coefficient + (-1/(2\*tf.pow(stddev\_vector[0,i],2))\*tf.reduce\_sum(tf.pow(dists,2),axis=1))  
 log\_probs.append(log\_prob\_per\_class) # shape: KXB  
 # result takes on the shape: KXB, transform it for more natural representation  
 log\_probs = tf.transpose(log\_probs, [1,0]) # BXK  
 return log\_probs  
  
# For 2.1  
def log\_posterior(data, mean, stddev\_vector, pi\_vector):  
 # assume pi\_vector: 1XK  
 # compute the log probability for "cluster" specified by the parameter  
 # compute term for each cluster (p(x|z) added with a prior)  
 log\_joint\_prob = []  
 log\_prob\_likelihood = log\_likelihood(data, mean, stddev\_vector) # BXK  
 # compute the log prior  
 log\_prob\_prior = pi\_vector # tf.log(pi\_vector)  
 log\_prob\_posterior\_unnormalized = log\_prob\_prior + log\_prob\_likelihood # BXK  
 # now ready to compute the probability on posterior for each cluster  
 log\_prob\_posterior = log\_prob\_posterior\_unnormalized / reduce\_logsumexp(log\_prob\_posterior\_unnormalized, reduction\_indices=1) # BXK  
 return log\_prob\_posterior  
  
# For 2.2  
def log\_marginal\_likelihood(data, mean, stddev\_vector, pi\_vector):  
 # compute term for each cluster (p(x|z) added with a prior)  
 log\_joint\_prob = []  
 log\_prob\_likelihood = log\_likelihood(data, mean, stddev\_vector) # BXK  
 # compute the log prior  
 log\_prob\_prior = pi\_vector # tf.log(pi\_vector)  
 log\_prob\_posterior\_unnormalized = log\_prob\_prior + log\_prob\_likelihood # BXK  
 log\_marginal\_likelihood\_per\_point = reduce\_logsumexp(log\_prob\_posterior\_unnormalized, reduction\_indices=1) # BX1  
 log\_marginal\_likelihood\_all = tf.reduce\_sum(log\_marginal\_likelihood\_per\_point)  
 return log\_marginal\_likelihood\_all  
  
# placeholders  
data = tf.placeholder(tf.float32, shape=[None, FEATURE\_AMOUNT], name="data\_placeholder")  
with tf.name\_scope("MoG\_parameters"):  
 mean = tf.Variable(tf.random\_normal([K, FEATURE\_AMOUNT]), name="mean")  
 # for parametrizing standard deviation  
 var\_parametrize = tf.Variable(tf.random\_normal([1,K]), name='var\_parametrize')  
 # for parametrizing pi  
 pi\_parametrize = tf.Variable(tf.random\_normal([1,K]), name='pi\_parametrize')  
  
# get variance vector and pi vector  
var\_vector = tf.exp(var\_parametrize); std\_vector = tf.sqrt(var\_vector)  
pi\_vector = logsoftmax(pi\_parametrize)  
  
loss = -log\_marginal\_likelihood(data, mean, std\_vector, pi\_vector)  
optimizer = tf.train.AdamOptimizer(LEARNING\_RATE, beta1=0.9, beta2=0.99, epsilon=1e-5)  
train\_step = optimizer.minimize(loss)  
  
if(\_\_name\_\_ == '\_\_main\_\_'):  
 # load dataset from data2D.npy  
 data2D = np.load('data2D.npy') #should be a 2D array with size 10000 X 2  
 # parse data into minibatches  
 # assuming batch size is divisible by the amount of data  
 data2D = np.reshape(data2D,[-1, MINIBATCH\_SIZE, FEATURE\_AMOUNT])  
  
 batch\_amount = np.shape(data2D)[0]  
 saver = tf.train.Saver()  
 loss\_epoches = []  
 print("Starting tensorflow session...")  
 with tf.Session() as sess:  
 sess.run(tf.global\_variables\_initializer())  
 for i in range(EPOCH\_AMOUNT):  
 epoch\_loss = 0  
 for j in range(batch\_amount):  
 batch\_loss, centroids, pi\_log, sigma, \_ = sess.run([loss, mean, pi\_vector, std\_vector, train\_step], feed\_dict={data: data2D[j]})  
 epoch\_loss += batch\_loss  
 loss\_epoches.append(epoch\_loss)  
 if((i+1) % 25 == 0):  
 print("At epoch {}, loss across all data: {}".format(i+1, epoch\_loss))  
 print("Model training complete!")  
 save\_path = saver.save(sess, "trained\_model/model\_2\_2.ckpt")  
 print("Trained model saved in file: {}".format(save\_path))  
  
 print("Mean: {}".format(centroids))  
 print("Pi: {}".format(np.exp(pi\_log)))  
 print("Sigma: {}".format(sigma))  
  
 print("Plotting losses vs number of updates:")  
 plt.title("Loss VS Number of Updates")  
 plt.xlabel("Epoch")  
 plt.ylabel("Negative Log Marginal Likelihood")  
 plt.plot(loss\_epoches)  
 plt.show()  
 print("Script completed successfully!")

## 2.2.3

# ECE521 A3  
# March 23th 2017  
# Wei Cui Leon Chen  
# Question 2.2.3  
  
import numpy as np  
import tensorflow as tf  
import matplotlib.pyplot as plt  
from utils import \*  
  
# parameters  
MINIBATCH\_SIZE = 303  
# hyper-parameters  
LEARNING\_RATE = 0.001  
# constants  
PI = np.pi  
K = 1  
FEATURE\_AMOUNT = 2  
EPOCH\_AMOUNT = 500  
  
# For 2.1  
def log\_likelihood(data, mean, stddev\_vector):  
 # data: BXD; mean: KXD; variance: K (given uni-standard deviation assumption)  
 log\_probs = []  
 for i in range(K):  
 dists = data - mean[i] # through broadcasting, shape is BXD  
 gaussian\_coefficient = -FEATURE\_AMOUNT/2\*tf.log(2\*PI) - FEATURE\_AMOUNT\*tf.log(stddev\_vector[0,i])  
 # Taking the log should turn into summation terms; result is with dimension: BX1  
 log\_prob\_per\_class = gaussian\_coefficient + (-1/(2\*tf.pow(stddev\_vector[0,i],2))\*tf.reduce\_sum(tf.pow(dists,2),axis=1))  
 log\_probs.append(log\_prob\_per\_class) # shape: KXB  
 # result takes on the shape: KXB, transform it for more natural representation  
 log\_probs = tf.transpose(log\_probs, [1,0]) # BXK  
 return log\_probs  
  
# For 2.1  
def log\_posterior(data, mean, stddev\_vector, pi\_vector):  
 # assume pi\_vector: 1XK  
 # compute the log probability for "cluster" specified by the parameter  
 # compute term for each cluster (p(x|z) added with a prior)  
 log\_joint\_prob = []  
 log\_prob\_likelihood = log\_likelihood(data, mean, stddev\_vector) # BXK  
 # compute the log prior  
 log\_prob\_prior = pi\_vector # tf.log(pi\_vector)  
 log\_prob\_posterior\_unnormalized = log\_prob\_prior + log\_prob\_likelihood # BXK  
 # now ready to compute the probability on posterior for each cluster  
 log\_prob\_posterior = log\_prob\_posterior\_unnormalized / reduce\_logsumexp(log\_prob\_posterior\_unnormalized, reduction\_indices=1) # BXK  
 return log\_prob\_posterior  
  
# For 2.2  
def log\_marginal\_likelihood(data, mean, stddev\_vector, pi\_vector):  
 # compute term for each cluster (p(x|z) added with a prior)  
 log\_joint\_prob = []  
 log\_prob\_likelihood = log\_likelihood(data, mean, stddev\_vector) # BXK  
 # compute the log prior  
 log\_prob\_prior = pi\_vector # tf.log(pi\_vector)  
 log\_prob\_posterior\_unnormalized = log\_prob\_prior + log\_prob\_likelihood # BXK  
 log\_marginal\_likelihood\_per\_point = reduce\_logsumexp(log\_prob\_posterior\_unnormalized, reduction\_indices=1) # BX1  
 log\_marginal\_likelihood\_all = tf.reduce\_sum(log\_marginal\_likelihood\_per\_point)  
 return log\_marginal\_likelihood\_all, log\_prob\_posterior\_unnormalized  
  
# placeholders  
data = tf.placeholder(tf.float32, shape=[None, FEATURE\_AMOUNT], name="data\_placeholder")  
with tf.name\_scope("MoG\_parameters"):  
 mean = tf.Variable(tf.random\_normal([K, FEATURE\_AMOUNT]), name="mean")  
 # for parametrizing standard deviation  
 var\_parametrize = tf.Variable(tf.random\_normal([1,K]), name='var\_parametrize')  
 # for parametrizing pi  
 pi\_parametrize = tf.Variable(tf.random\_normal([1,K]), name='pi\_parametrize')  
  
# get variance vector and pi vector  
var\_vector = tf.exp(var\_parametrize); std\_vector = tf.sqrt(var\_vector)  
pi\_vector = logsoftmax(pi\_parametrize)  
  
loss, indiv\_loss = log\_marginal\_likelihood(data, mean, std\_vector, pi\_vector)  
loss = -loss  
indiv\_loss = -indiv\_loss  
assignments = tf.argmin(indiv\_loss, axis=1)  
optimizer = tf.train.AdamOptimizer(LEARNING\_RATE, beta1=0.9, beta2=0.99, epsilon=1e-5)  
train\_step = optimizer.minimize(loss)  
  
if(\_\_name\_\_ == '\_\_main\_\_'):  
 # load dataset from data2D.npy  
 data2D = np.load('data2D.npy') #should be a 2D array with size 10000 X 2  
 tr\_data = data2D[:int(data2D.shape[0]\*2/3)]  
 va\_data = data2D[int(data2D.shape[0]\*2/3):]  
 n\_tr = tr\_data.shape[0]  
 n\_va = va\_data.shape[0]  
 # parse data into minibatches  
 # assuming batch size is divisible by the amount of data  
 tr\_data = np.reshape(tr\_data, [-1, MINIBATCH\_SIZE, FEATURE\_AMOUNT])  
 batch\_amount = np.shape(tr\_data)[0]  
  
 saver = tf.train.Saver()  
 loss\_epoches = []  
 print("Starting tensorflow session...")  
 with tf.Session() as sess:  
 sess.run(tf.global\_variables\_initializer())  
 for i in range(EPOCH\_AMOUNT):  
 epoch\_loss = 0  
 for j in range(batch\_amount):  
 batch\_loss, \_ = sess.run([loss, train\_step], feed\_dict={data: tr\_data[j]})  
 epoch\_loss += batch\_loss  
 loss\_epoches.append(epoch\_loss)  
 if((i+1) % 25 == 0):  
 print("At epoch {}, loss across all data: {}".format(i+1, epoch\_loss))  
 print("Model training complete!")  
 save\_path = saver.save(sess, "trained\_model/model\_2\_2.ckpt")  
 print("Trained model saved in file: {}".format(save\_path))  
  
 # Validation  
 valid\_loss = sess.run(loss, feed\_dict={data: va\_data})  
 print('Validation Loss is: ', valid\_loss)  
  
 # Final cluster  
 centroids, assign = sess.run([mean, assignments], feed\_dict={data: data2D})  
  
 # Percentage assigned to each cluster  
 for cluster in range(K):  
 print('{}% assigned to cluster {}'.format(round(np.mean(np.array(assign) == cluster)\*100, 2), cluster+1))  
  
 print("Plotting losses vs number of updates:")  
 # Plot the data  
 plt.figure(1)  
 plt.scatter(data2D[:, 0], data2D[:, 1], c=assign, s=1)  
 plt.scatter(centroids[:, 0], centroids[:, 1], marker='X', color='black', s=100)  
 plt.title('K = {}'.format(K))  
 plt.show()  
 print("Script completed successfully!")

## 2.2.4: K-means

# ECE521 A3  
# March 18th 2017  
# Wei Cui Leon Chen  
# Question 2.2.4 - K-means  
  
import matplotlib.pyplot as plt  
import numpy as np  
import tensorflow as tf  
  
# parameters  
K = 25  
feature\_amount = 100  
epoch\_amount = 800  
batch\_size = 3333  
# hyper-parameters  
learning\_rate = 0.005  
  
# Tensorflow graph  
data = tf.placeholder(tf.float32, shape=[None, feature\_amount], name="data\_placeholder")  
with tf.name\_scope("2D\_mean"):  
 mean = tf.Variable(tf.random\_normal([K, feature\_amount]), name="mean")  
# compute the loss given the mean  
sub\_array = []  
for i in range(K):  
 sub\_array\_per\_class = tf.reduce\_sum(tf.pow((data - mean[i]), 2), axis=1)  
 sub\_array.append(sub\_array\_per\_class)  
assignments = tf.argmin(sub\_array, axis=0)  
sub\_array = tf.reduce\_min(sub\_array, axis=0)  
loss = tf.reduce\_sum(sub\_array)  
optimizer = tf.train.AdamOptimizer(learning\_rate, beta1=0.9, beta2=0.99, epsilon=1e-5)  
train\_step = optimizer.minimize(loss)  
  
  
if(\_\_name\_\_ == '\_\_main\_\_'):  
 # load dataset from data2D.npy  
 data100D = np.load('data100D.npy') #should be a 2D array with size 10000 X 2  
 tr\_data = data100D[:int(data100D.shape[0]\*2/3)]  
 va\_data = data100D[int(data100D.shape[0]\*2/3):]  
 n\_tr = tr\_data.shape[0]  
 n\_va = va\_data.shape[0]  
 # parse data into minibatches  
 # assuming batch size is divisible by the amount of data  
 tr\_data = np.reshape(tr\_data, [-1, batch\_size, feature\_amount])  
 batch\_amount = np.shape(tr\_data)[0]  
  
 train\_loss = np.zeros(epoch\_amount)  
 centroids = np.zeros(K)  
 saver = tf.train.Saver()  
 with tf.Session() as sess:  
 sess.run(tf.global\_variables\_initializer())  
 for i in range(epoch\_amount):  
 epoch\_loss = 0  
 for j in range(batch\_amount):  
 centroids, batch\_loss, assign\_batch, \_ = sess.run([mean, loss, assignments, train\_step], feed\_dict={data: tr\_data[j]})  
 epoch\_loss += batch\_loss  
 train\_loss[i] = epoch\_loss  
 if((i+1) % 20 == 0):  
 print("At epoch {}, loss across all data: {}".format(i+1, epoch\_loss))  
 print("Model training complete!")  
 save\_path = saver.save(sess, "trained\_model/model\_1\_1\_4.ckpt")  
 print("Trained model saved in file: {}".format(save\_path))  
  
 # Validation  
 valid\_loss, assign = sess.run([loss, assignments], feed\_dict={data: va\_data})  
 print('Validation Loss is: ', valid\_loss)  
  
 # Percentage assigned to each cluster  
 for cluster in range(K):  
 print('{}% assigned to cluster {}'.format(round(np.mean(np.array(assign) == cluster)\*100, 2), cluster+1))  
  
 print("Script completed successfully!")

2.2.4: MoG

# ECE521 A3  
# March 23th 2017  
# Wei Cui Leon Chen  
# Question 2.2.4 - MoG  
  
import numpy as np  
import tensorflow as tf  
import matplotlib.pyplot as plt  
from utils import \*  
  
# parameters  
MINIBATCH\_SIZE = 3333  
# hyper-parameters  
LEARNING\_RATE = 0.005  
# constants  
PI = np.pi  
K = 25  
FEATURE\_AMOUNT = 100  
EPOCH\_AMOUNT = 800  
  
# For 2.1  
def log\_likelihood(data, mean, stddev\_vector):  
 # data: BXD; mean: KXD; variance: K (given uni-standard deviation assumption)  
 log\_probs = []  
 for i in range(K):  
 dists = data - mean[i] # through broadcasting, shape is BXD  
 gaussian\_coefficient = -FEATURE\_AMOUNT/2\*tf.log(2\*PI) - FEATURE\_AMOUNT\*tf.log(stddev\_vector[0,i])  
 # Taking the log should turn into summation terms; result is with dimension: BX1  
 log\_prob\_per\_class = gaussian\_coefficient + (-1/(2\*tf.pow(stddev\_vector[0,i],2))\*tf.reduce\_sum(tf.pow(dists,2),axis=1))  
 log\_probs.append(log\_prob\_per\_class) # shape: KXB  
 # result takes on the shape: KXB, transform it for more natural representation  
 log\_probs = tf.transpose(log\_probs, [1,0]) # BXK  
 return log\_probs  
  
# For 2.1  
def log\_posterior(data, mean, stddev\_vector, pi\_vector):  
 # assume pi\_vector: 1XK  
 # compute the log probability for "cluster" specified by the parameter  
 # compute term for each cluster (p(x|z) added with a prior)  
 log\_joint\_prob = []  
 log\_prob\_likelihood = log\_likelihood(data, mean, stddev\_vector) # BXK  
 # compute the log prior  
 log\_prob\_prior = pi\_vector # tf.log(pi\_vector)  
 log\_prob\_posterior\_unnormalized = log\_prob\_prior + log\_prob\_likelihood # BXK  
 # now ready to compute the probability on posterior for each cluster  
 log\_prob\_posterior = log\_prob\_posterior\_unnormalized / reduce\_logsumexp(log\_prob\_posterior\_unnormalized, reduction\_indices=1) # BXK  
 return log\_prob\_posterior  
  
# For 2.2  
def log\_marginal\_likelihood(data, mean, stddev\_vector, pi\_vector):  
 # compute term for each cluster (p(x|z) added with a prior)  
 log\_joint\_prob = []  
 log\_prob\_likelihood = log\_likelihood(data, mean, stddev\_vector) # BXK  
 # compute the log prior  
 log\_prob\_prior = pi\_vector # tf.log(pi\_vector)  
 log\_prob\_posterior\_unnormalized = log\_prob\_prior + log\_prob\_likelihood # BXK  
 log\_marginal\_likelihood\_per\_point = reduce\_logsumexp(log\_prob\_posterior\_unnormalized, reduction\_indices=1) # BX1  
 log\_marginal\_likelihood\_all = tf.reduce\_sum(log\_marginal\_likelihood\_per\_point)  
 return log\_marginal\_likelihood\_all, log\_prob\_posterior\_unnormalized  
  
# placeholders  
data = tf.placeholder(tf.float32, shape=[None, FEATURE\_AMOUNT], name="data\_placeholder")  
with tf.name\_scope("MoG\_parameters"):  
 mean = tf.Variable(tf.random\_normal([K, FEATURE\_AMOUNT]), name="mean")  
 # for parametrizing standard deviation  
 var\_parametrize = tf.Variable(tf.random\_normal([1,K]), name='var\_parametrize')  
 # for parametrizing pi  
 pi\_parametrize = tf.Variable(tf.random\_normal([1,K]), name='pi\_parametrize')  
  
# get variance vector and pi vector  
var\_vector = tf.exp(var\_parametrize); std\_vector = tf.sqrt(var\_vector)  
pi\_vector = logsoftmax(pi\_parametrize)  
  
loss, indiv\_loss = log\_marginal\_likelihood(data, mean, std\_vector, pi\_vector)  
loss = -loss  
indiv\_loss = -indiv\_loss  
assignments = tf.argmin(indiv\_loss, axis=1)  
optimizer = tf.train.AdamOptimizer(LEARNING\_RATE, beta1=0.9, beta2=0.99, epsilon=1e-5)  
train\_step = optimizer.minimize(loss)  
  
if(\_\_name\_\_ == '\_\_main\_\_'):  
 # load dataset from data2D.npy  
 data100D = np.load('data100D.npy') #should be a 2D array with size 10000 X 2  
 tr\_data = data100D[:int(data100D.shape[0]\*2/3)]  
 va\_data = data100D[int(data100D.shape[0]\*2/3):]  
 n\_tr = tr\_data.shape[0]  
 n\_va = va\_data.shape[0]  
 # parse data into minibatches  
 # assuming batch size is divisible by the amount of data  
 tr\_data = np.reshape(tr\_data, [-1, MINIBATCH\_SIZE, FEATURE\_AMOUNT])  
 batch\_amount = np.shape(tr\_data)[0]  
  
 saver = tf.train.Saver()  
 loss\_epoches = []  
 print("Starting tensorflow session...")  
 with tf.Session() as sess:  
 sess.run(tf.global\_variables\_initializer())  
 for i in range(EPOCH\_AMOUNT):  
 epoch\_loss = 0  
 for j in range(batch\_amount):  
 batch\_loss, \_ = sess.run([loss, train\_step], feed\_dict={data: tr\_data[j]})  
 epoch\_loss += batch\_loss  
 loss\_epoches.append(epoch\_loss)  
 if((i+1) % 25 == 0):  
 print("At epoch {}, loss across all data: {}".format(i+1, epoch\_loss))  
 print("Model training complete!")  
 save\_path = saver.save(sess, "trained\_model/model\_2\_2.ckpt")  
 print("Trained model saved in file: {}".format(save\_path))  
  
 # Validation  
 valid\_loss, assign = sess.run([loss, assignments], feed\_dict={data: va\_data})  
 print('Validation Loss is: ', valid\_loss)  
  
 # Percentage assigned to each cluster  
 for cluster in range(K):  
 print('{}% assigned to cluster {}'.format(round(np.mean(np.array(assign) == cluster)\*100, 2), cluster+1))  
  
 print("Script completed successfully!")

# Appendix C: Discover Latent Dimensions

## 3.1.2

# Wei Cui Leon Chen  
# March 24th 2017  
# ECE521 A3 Question 3.1.2  
  
import numpy as np  
import tensorflow as tf  
import matplotlib.pyplot as plt  
  
# hyper-parameters  
LEARNING\_RATE = 0.001  
EPOCH\_AMOUNT = 1000  
# parameters  
minibatch\_size = 100  
D = 64  
K = 4  
# constant  
PI = np.pi  
  
# helper function to get the marginal log likelihood which we want to maximize  
def marginal\_log\_likelihood(data, mean, weight, cov):  
 cov\_mat = tf.matmul(weight, tf.transpose(weight, [1,0])) + tf.diag(cov)  
 log\_coefficient = -D/2\*np.log(2\*PI)  
 log\_det\_term = -tf.reduce\_sum(tf.log(tf.diag\_part(tf.cholesky(cov\_mat))))  
 log\_likelihood\_set = 0  
 # loop over images. Don't think there's a vectorized approach of coding  
 for i in range(minibatch\_size):  
 diff\_mat = tf.expand\_dims(data[i]-mean, axis=0)  
 log\_exp\_term = -1/2\*tf.matmul(tf.matmul(diff\_mat, tf.matrix\_inverse(cov\_mat)), tf.transpose(diff\_mat, [1,0]))  
 log\_likelihood\_point = log\_coefficient + log\_det\_term + log\_exp\_term  
 log\_likelihood\_set += log\_likelihood\_point  
 return log\_likelihood\_set  
  
# placeholders  
x = tf.placeholder(tf.float32, [None, 64], "data\_placeholder")  
  
with tf.name\_scope("variables"):  
 weight = tf.Variable(tf.truncated\_normal(shape=[D, K], stddev=0.01, dtype=tf.float32, name='weight'))  
 cov = tf.Variable(tf.truncated\_normal(shape=[D], stddev=0.01, dtype=tf.float32, name="covariance"))  
# mean  
mean = tf.reduce\_mean(x, axis=0) # 1XD  
marginal\_log\_likelihood\_value = marginal\_log\_likelihood(x, mean, weight, tf.exp(cov))  
loss = -marginal\_log\_likelihood\_value  
optimizer = tf.train.AdamOptimizer(LEARNING\_RATE, beta1=0.9, beta2=0.99, epsilon=1e-5)  
train\_step = optimizer.minimize(loss)  
  
if(\_\_name\_\_=='\_\_main\_\_'):  
 print("Loading the data")  
 with np.load("tinymnist.npz") as data\_dict:  
 # image data pixel values are already normalized between 0 and 1  
 train\_data = data\_dict["x"] # train\_data: 700X64  
 valid\_data = data\_dict["x\_valid"] # valid\_data: 100X64  
 test\_data = data\_dict["x\_test"] # test\_data: 400X64  
 np.random.seed(521)  
 randIndx = np.arange(len(train\_data))  
 np.random.shuffle(randIndx)  
 train\_data = train\_data[randIndx]  
 print("Done loading the data!")  
  
 # parse the train data into minibatches  
 train\_data\_batches = np.reshape(train\_data, [-1, minibatch\_size, 64])  
 train\_batch\_amount = np.shape(train\_data\_batches)[0]  
  
 saver = tf.train.Saver()  
 loss\_epoches = []  
 print("Starting tensorflow session...")  
 with tf.Session() as sess:  
 sess.run(tf.global\_variables\_initializer())  
 for i in range(EPOCH\_AMOUNT):  
 epoch\_loss = 0  
 for j in range(train\_batch\_amount):  
 batch\_loss, \_ = sess.run([loss, train\_step], feed\_dict={x: train\_data\_batches[j]})  
 epoch\_loss += batch\_loss  
 loss\_epoches.append(epoch\_loss)  
 if((i+1) % 20 == 0):  
 print("At epoch {}, loss across all training data: {}".format(i+1, epoch\_loss[0][0]))  
 print("Model training complete!")  
 save\_path = saver.save(sess, "trained\_model/model\_3\_2.ckpt")  
 print("Trained model saved in file: {}".format(save\_path))  
 # getting weights  
 weights = sess.run(weight)  
  
 # Validation  
 valid\_loss = sess.run(loss, feed\_dict={x: valid\_data})  
 print('Validation Loss is: ', valid\_loss[0][0])  
  
 # Testing  
 test\_loss = sess.run(loss, feed\_dict={x: test\_data})  
 print('Test Loss is: ', test\_loss[0][0])  
  
 print("saving trained weights...")  
 np.save("trained\_model/weights\_3\_2.npy", weights)  
 print("Plotting weights:")  
 weights = np.transpose(weights, [1,0])  
 weights\_plot = np.reshape(weights, [-1, 8, 8])  
 fig = plt.figure(figsize=(2,2))  
 for img in range(np.shape(weights\_plot)[0]):  
 ax = fig.add\_subplot(2,2,img+1)  
 ax.imshow(weights\_plot[img], cmap='gray')  
 ax.axis('off')  
 fig.subplots\_adjust(wspace = 0, hspace = 0)  
 plt.show()

## 3.1.3

# ECE521 A3  
# March 23th 2017  
# Wei Cui Leon Chen  
# Question 3.1.3  
  
import numpy as np  
import tensorflow as tf  
import matplotlib.pyplot as plt  
  
# PCA and FA from Scikit-Learn  
from sklearn.decomposition import PCA  
from sklearn.decomposition import FactorAnalysis  
  
if(\_\_name\_\_ == '\_\_main\_\_'):  
 mean = [0, 0, 0]  
 covar = np.eye(3)  
 latent = np.random.multivariate\_normal(mean, covar, 200) # s = N(s; 0, I)  
 data = np.matmul([[1, 0, 0], [1, 0.1, 0], [0, 0, 10]], latent.T) # x = [s1, s1 + 0.1\*s2, s3]  
 data = data.T  
  
 # PCA  
 pca = PCA(n\_components=1).fit(data)  
 print('PCA: ', pca.components\_)  
  
 # Factor Analysis  
 factor = FactorAnalysis(n\_components=1).fit(data)  
  
 # Get W and Psi inverse  
 W = factor.components\_.T  
 info = np.linalg.inv(factor.get\_covariance())  
  
 # Wproj = ((I + W^T \* Psi^-1 \* W)^-1 \* W)^T \* Psi^-1  
 sigma = covar + np.matmul(np.matmul(W.T, info), W)  
 sigma = np.linalg.inv(sigma)  
 Wproj = np.matmul(np.matmul(sigma, W).T, info)  
 print('FA: ', Wproj)