ECE 521 Inference Algorithms and Machine Learning

Assignment #3

Student Name: Shengze Gao(ID: 1002935942)
Student Name: Jingxiong Luo(ID: 1002772332)

Mar 24th, 2017

Percentage of Contribution:
Jingxiong Luo: 50% Shengze Gao: 50%

Table of Content

1. K-means

1.1 Learning K-means

1.1.1

For D dimension dataset:

$$\mathcal{L}(\mu)=\sum_{n=1}^{B}\min_{k=1}^{K}\|x_n-\mu_k\|_2^2=\sum_{n=1}^{B}\min_{k=1}^{K}\sum_{d=1}^{D}(x_{nd}-\mu_{kd})^2$$

$$t\mathcal{L}(\mu_a)+(1-t)\mathcal{L}(\mu_b)-\mathcal{L}(t\mu_a+(1-t)\mu_b)=\sum_{n=1}^{B}\min_{k=1}^{k}t\|x_n-\mu_{ka}\|_2^2+\sum_{n=1}^{B}\min_{k=1}^{K}(1-t)\|x_n-\mu_{kb}\|_2^2$$
$$-\sum_{n=1}^{B}\min_{k=1}^{K}\|x_n-t\mu_{ka}-(1-t)\mu_{kb}\|_2^2$$
$$=\sum_{n=1}^{B}\min_{k=1}^{K}\sum_{d=1}^{D}\underbrace{t(x_{nd}-\mu_{kad})^2}_{①}+\sum_{n=1}^{B}\min_{k=1}^{K}\sum_{d=1}^{D}\underbrace{(1-t)(x_{nd}-\mu_{kbd})^2}_{②}$$
$$-\sum_{n=1}^{B}\min_{k=1}^{K}\sum_{d=1}^{D}(x_{nd}-\underbrace{t\mu_{kad}}_{③}-(1-t)\mu_{kbd})^2$$

In order to prove $t\mathcal{L}(\mu_a)+(1-t)\mathcal{L}(\mu_b)-\mathcal{L}(t\mu_a+(1-t)\mu_b)$ is not always larger than 0, let's make an example:

For ALL dimensions (ALL data), we have: $x_n=0$, $\mu_{1a}=0.1$, $\mu_{2a}=0.9$ and other μ_{ka} are larger than 0.9

$\mu_{1b}=2.75$, $\mu_{2b}=2$ and other μ_{kb} are larger than 2.75

Assume $t=0.5$.

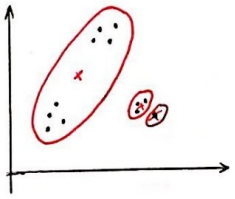
Thus: ① $=\sum_{n=1}^{B}\sum_{d=1}^{D}0.5\cdot(0-0.1)^2$ ② $=\sum_{n=1}^{B}\sum_{d=1}^{D}0.5\cdot(0-2)^2$ ③ $=\sum_{n=1}^{B}\sum_{d=1}^{D}(0-0.5\times0.1-0.5\times2.75)^2$

$$\Rightarrow ①+②-③=\sum_{n=1}^{B}\sum_{d=1}^{D}(0.005+2-2.030625)=\sum_{n=1}^{B}\sum_{d=1}^{D}(-0.25625)<0$$

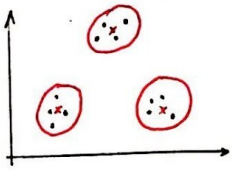
Thus, in this specific case, $t\mathcal{L}(\mu_a)+(1-t)\mathcal{L}(\mu_b)-\mathcal{L}(t\mu_a+(1-t)\mu_b)<0$

$\Rightarrow$ The Loss function is not convex.

Intutively, the loss function could get stuck at the local optima as follows:



The global optima should be:



1.1.2
the K-means clusters μ :

```
('mu:', array([[ 0.12183344, -1.52304184],
       [-1.0559268 , -3.24319768],
       [ 1.25175285,  0.2465685 ]]))
```

As indicated, the plot of the loss vs the number of updates included as follows:

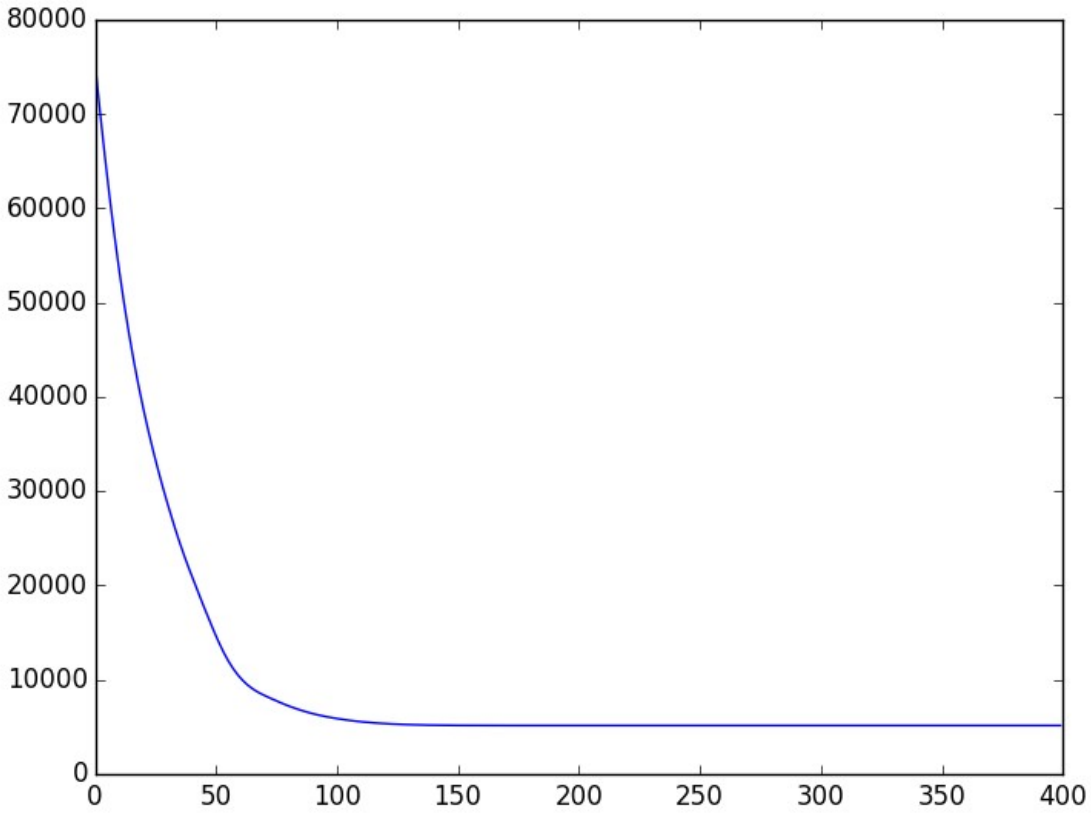


Figure 1.1.2-1: The loss vs the number of updates of learning of K-means when K = 3

Deriving from the above plot, the learning converged within 150 updates.

1.1.3

The following discusses the cluster assignments for data points with different values of K (K = 1, 2, 3, 4, 5).

For K = 1, the percentage of the data points belonging to each K clusters is reported as follows:

```
('percentage:', array([[ 1.]]))
```

It shows that all the data points are assigned to a cluster (the percentage of this cluster is 100% as displayed).

Intuitively, the 2D scatter plot of data points colored by their cluster assignments is shown as follows:

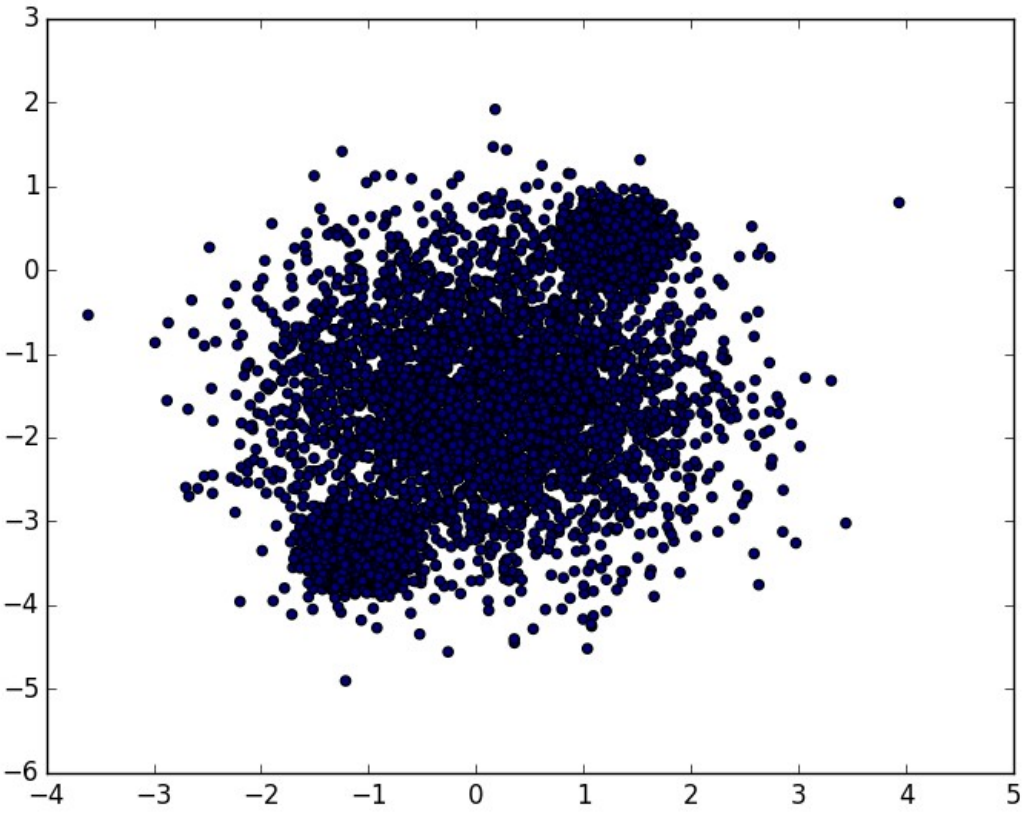


Figure 1.1.3-1: The 2D scatter plot of data point colored by their cluster assignments (K = 1)

For K = 2, the percentage of the data points belonging to each K clusters is reported as follows:

```
('percentage:', array([[ 0.4954],
       [ 0.5046]]))
```

It shows that the percentage of data points assigned to two clusters: 49.54% and 50.46%.

Intuitively, the 2D scatter plot of data points colored by their cluster assignments is shown as follows:

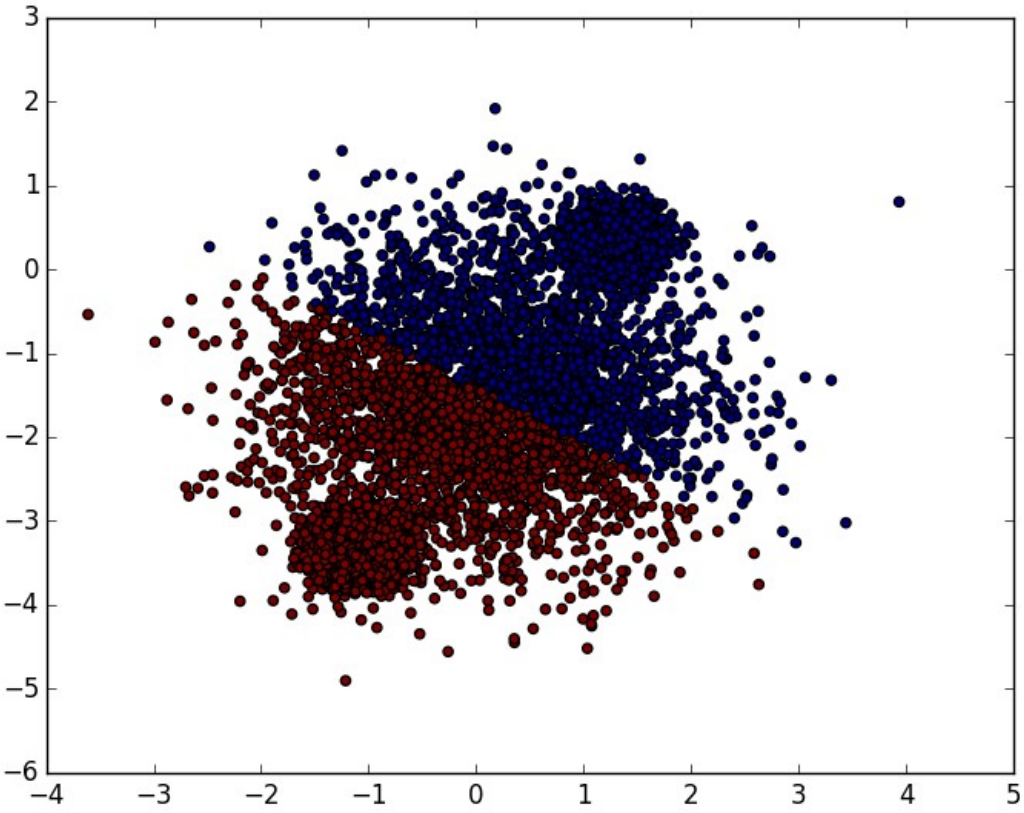


Figure 1.1.3-2: The 2D scatter plot of data point colored by their cluster assignments (K = 2)

For K = 3, the percentage of the data points belonging to each K clusters is reported as follows:

```
('percentage:', array([[ 0.3806],
       [ 0.3813],
       [ 0.2381]]))
```

It shows that the percentage of data points assigned to three clusters: 38.06%, 38.13% and 23.81%.

Intuitively, the 2D scatter plot of data points colored by their cluster assignments is shown as follows:

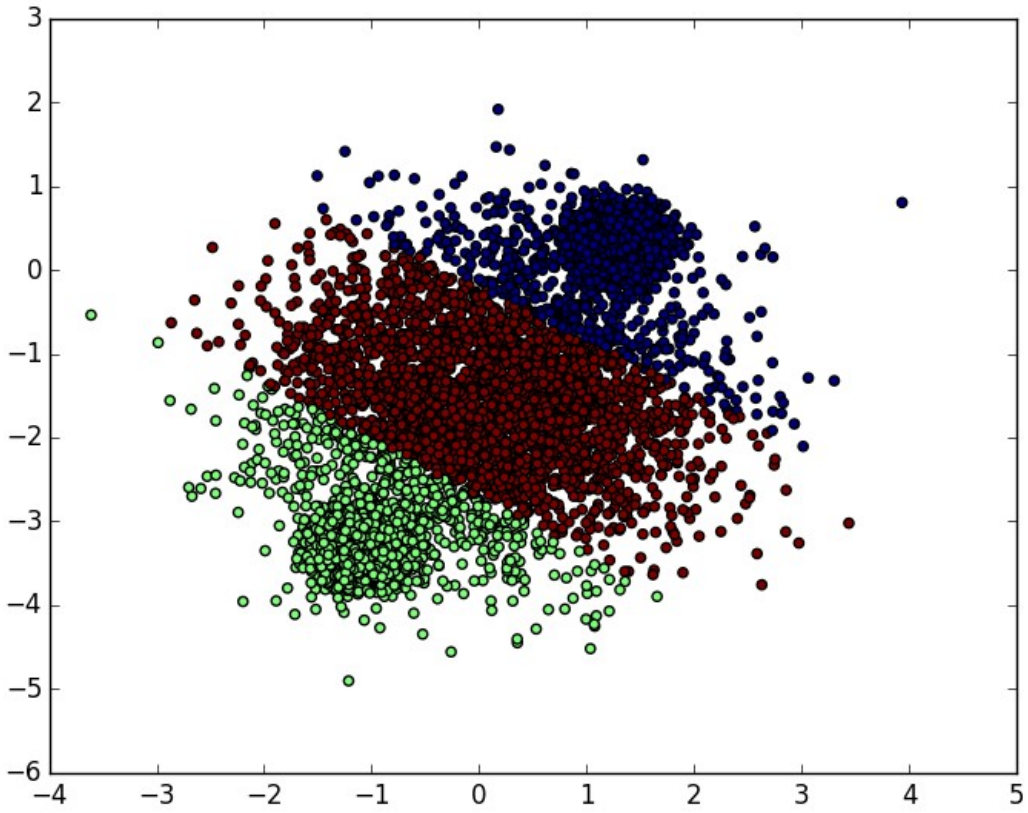


Figure 1.1.3-3: The 2D scatter plot of data point colored by their cluster assignments (K = 3)

For K = 4, the percentage of the data points belonging to each K clusters is reported as follows:

```
('percentage:', array([[ 0.1204],
       [ 0.1353],
       [ 0.373 ],
       [ 0.3713]]))
```

It shows that the percentage of data points assigned to three clusters: 12.04%, 13.53%, 37.3% and 37.13%.

Intuitively, the 2D scatter plot of data points colored by their cluster assignments is shown as follows:

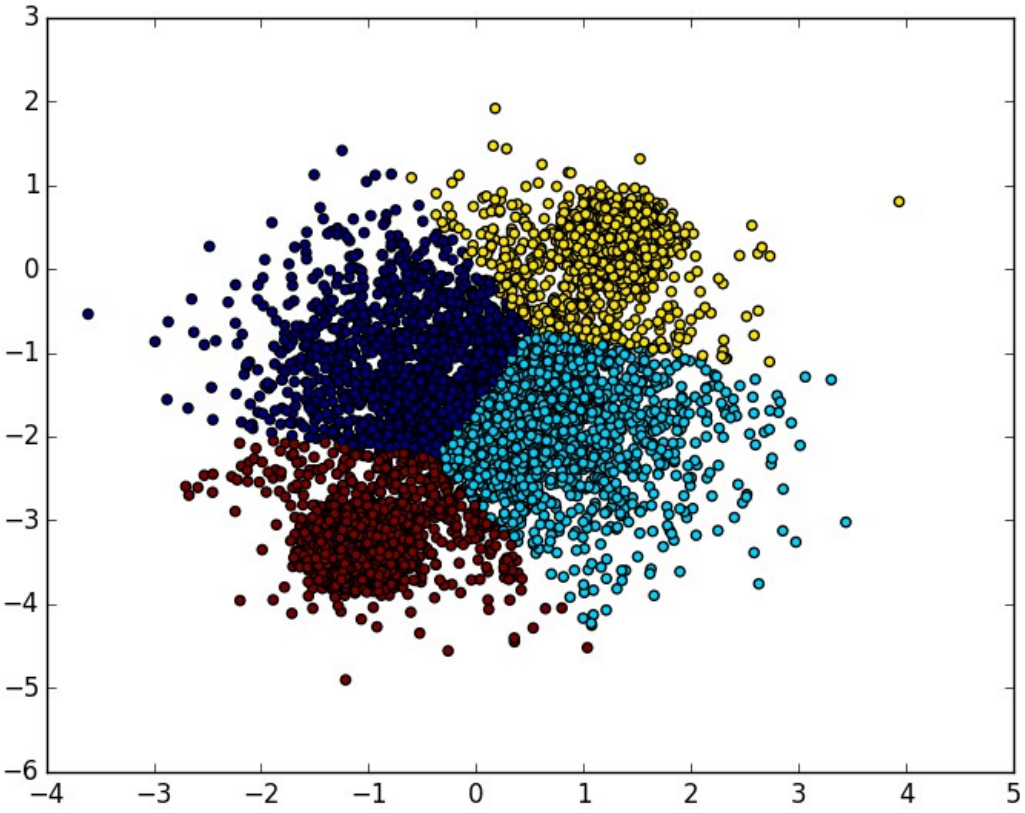


Figure 1.1.3-4: The 2D scatter plot of data point colored by their cluster assignments (K = 4)

For K = 5, the percentage of the data points belonging to each K clusters is reported as follows:

```
('percentage:', array([[ 0.1076],
       [ 0.0882],
       [ 0.0842],
       [ 0.3576],
       [ 0.3624]]))
```

It shows that the percentage of data points assigned to three clusters: 8.82%, 8.42%, 10.76%, 35.76% and 36.24%.

Intuitively, the 2D scatter plot of data points colored by their cluster assignments is shown as follows:

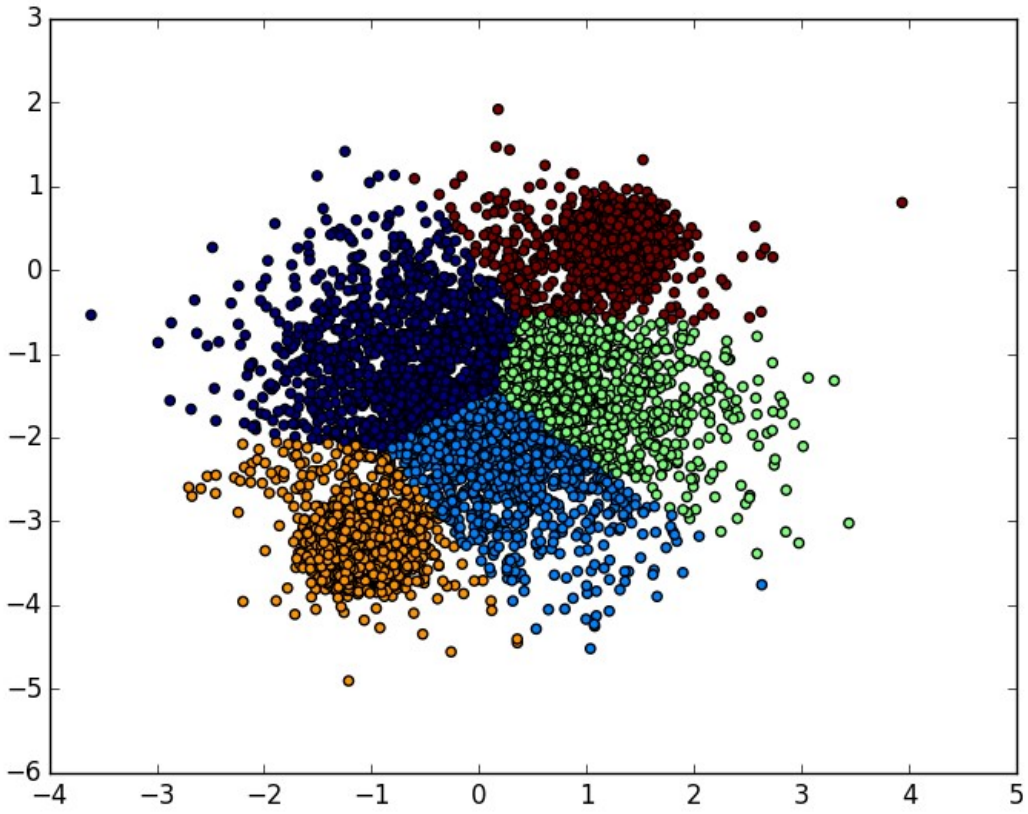


Figure 1.1.3-5: The 2D scatter plot of data point colored by their cluster assignments (K = 5)

Based on the percentage of data points belonging to each cluster and the 2D scatter plots, it is more likely to be evenly divided for all data points when K = 3. And according to the original data distribution (figure 1.1.3-1), we can see that the data points are approximately made by three clusters. Thus, 3 clusters may be the best choice since it can evenly divide the data points.

1.1.4

The following displays the loss for the validation data for each value of K = 1, 2, 3, 4, 5.

When K = 1:

```
'TRAIN_loss:', 25577.15173425692)
'VALID_loss:', 12876.880576846093)
```

The loss for validation data = 12876.9

When K = 2:

```
'TRAIN_loss:', 6065.4186671720972)
'VALID_loss:', 3138.5000419890521)
```

The loss for validation data = 3138.5

When K = 3:

```
'TRAIN_loss:', 3369.562566088106)
'VALID_loss:', 1744.2125116328218)
```

The loss for validation data = 1744.2

When K = 4:

```
'TRAIN_loss:', 2244.8716578896288)
'VALID_loss:', 1132.3907098571003)
```

The loss for validation data = 1132.4

When K = 5:

```
'TRAIN_loss:', 1916.4639700280479)
'VALID_loss:', 968.95213048348774)
```

The loss for validation data = 969.0

Two clusters is the best choice. By viewing all the validation loss for K = 1, 2, 3, 4, 5, the loss decreases dramatically from the case of K = 1 to K = 2. The loss decreases much slower after K becomes larger than 2, which means the increasing of K after K is larger than 2 does not that effective to the decreasing of validation loss. Thus, K = 2 is considered as the best choice.

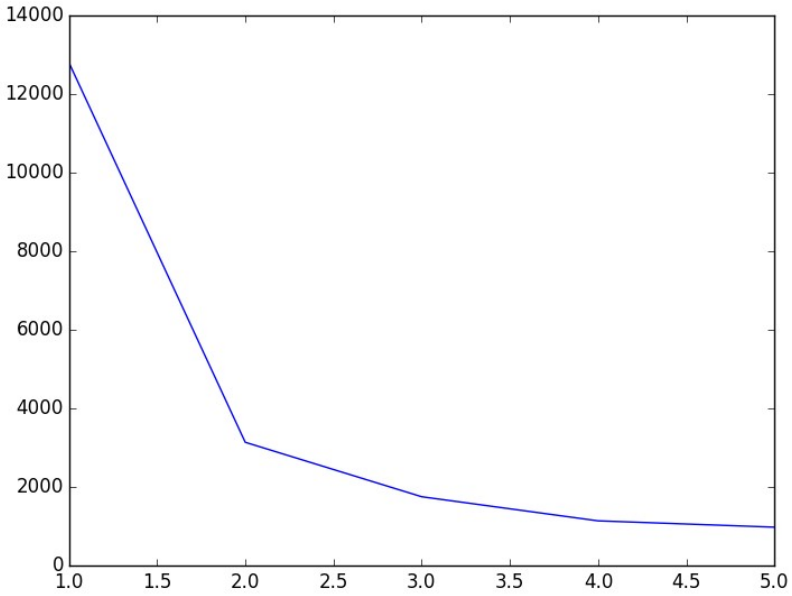


Figure 1.1.4 cluster number vs. validation loss

2. Mixtures of Gaussians

2.1 The Gaussian cluster model

2.1.1

$$P(z=k) = \pi^k$$

$$P(x|z=k) = \mathcal{N}(x|\mu^k, \sigma^{k^2})$$

$$P(x, z=k) = \mathcal{N}(x|\mu^k, \sigma^{k^2}) \cdot \pi^k$$

$$\Rightarrow P(z=k|x) = \frac{P(x,z=k)}{\sum_{k=1}^{K} P(x,z=k)} = \frac{\pi^k \mathcal{N}(x|\mu^k, \sigma^{k^2})}{\sum_{k=1}^{K} \pi^k \mathcal{N}(x|\mu^k, \sigma^{k^2})}$$

2.1.2

The snippets of the Python code for obtaining the log probability density function for cluster k: log N (x ; μ k , σk 2) is shown as follows:

```
def Euclidean_dis(X, Y):
    #return the matrix containing the pairwise Euclidean distances

    X_b = tf.expand_dims(X, 1)
    result = X_b - Y
    result_square = tf.square(result)
    Euclidean_dis = tf.reduce_sum(result_square, 2)

    return Euclidean_dis

def Gaussian_prob(X,mu,Vsigma):

    Dims = tf.cast(tf.shape(X)[1],tf.float64)
    pi = tf.constant(3.14159265359,tf.float64)
    dis = Euclidean_dis(X,mu)

    first_term = -Dims/2.*tf.log(2.*pi*tf.square(Vsigma))
    second_term = tf.div(-dis,2.*tf.square(Vsigma))
    log_gauss = first_term + second_term

    return log_gauss
```

In order to verify the implementation of this part of code, a set of input data, mu and sigma are provided as follows:

```
x = np.array([[0.,0.],[10.,1.]])
mu = np.array([[0.,0.],[2.,2.]])
sigma=np.array([[0.5,0.1]])
```

The output of log_gauss is:

```
[[ -4.51582705e-01  -3.97232707e+02]
 [ -2.02451583e+02  -3.24723271e+03]]
```

It is able to successfully obtain the log probability density function without any infinite values or nan values.

2.1.3

The snippets of Python code is shown as follows:

```
def Euclidean_dis(X, Y):
    X_b = tf.expand_dims(X, 1)
    result = X_b - Y
    result_square = tf.square(result)
    Euclidean_dis = tf.reduce_sum(result_square, 2)

    return Euclidean_dis

def Gaussian_prob(X,mu,Vsigma):

    Dims = tf.cast(tf.shape(X)[1],tf.float64)
    pi = tf.constant(3.14159265359,tf.float64)
    dis = Euclidean_dis(X,mu)
    first_term = -Dims/2.*tf.log(2.*pi*Vsigma*Vsigma)
    second_term = tf.div(-dis,2.*tf.square(Vsigma))
    log_gauss = first_term + second_term

    return log_gauss

def jointprob(X,mu,prior,sigma):
    log_gauss = Gaussian_prob(X,mu,sigma)
    joint = log_gauss+prior
    return joint

def log_prob(x,mean,prior,sigma):
   log_post = jointprob(x,mean,prior,sigma)
   posteriori = logsoftmax(log_post)
   return posteriori

def reduce_logsumexp(input_tensor, reduction_indices=1, keep_dims=False):...
def logsoftmax(input_tensor):...
```

It is important to use the log-sum-exp function because in this project we need to compute the sum of log-likelihood of mixture gaussian. However if we run codes like this:

```
In[15]: a=np.exp(-1000)
In[16]: b=np.exp(-1001)
In[17]: np.log(a+b)
```

a and b are similar to gaussian pdf. If we directly run this in computer, it will return -inf which is not desirable. So we should implement log-sum-exp:

```
In[19]: -1000+np.log(1+np.exp(-1))
Out[19]: -999.68673831248179
```

Instead, it does the same calculation by pulling out the maximum firstly, thus avoiding the exp(-a) being 0.0. So we can get the right answer.

2.2 Learning the MoG

2.2.1

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

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

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

$$= \sum_{k} P(z=k \mid x) \nabla_\mu \log P(x, z=k)$$

2.2.2

Note that the Python code for all problems are included in Appendix.

When K = 3, the following capture displays the best model parameters (mu, π, sigma) it has learnt.

```
('mean:', array([[ 0.10620466, -1.5273211 ],
       [-1.1018883 , -3.30592465],
       [ 1.29837322,  0.30911011]]))
('prior:', array([[ 0.33465932,  0.33190734,  0.33343334]]))
('sigma:', array([ 0.9935203 ,  0.19764431,  0.19709408]))
```

Note that the sum of all π s equals to 1 and sigma is converted to exp(sigma).

The following plot displays the loss vs the number of updates.

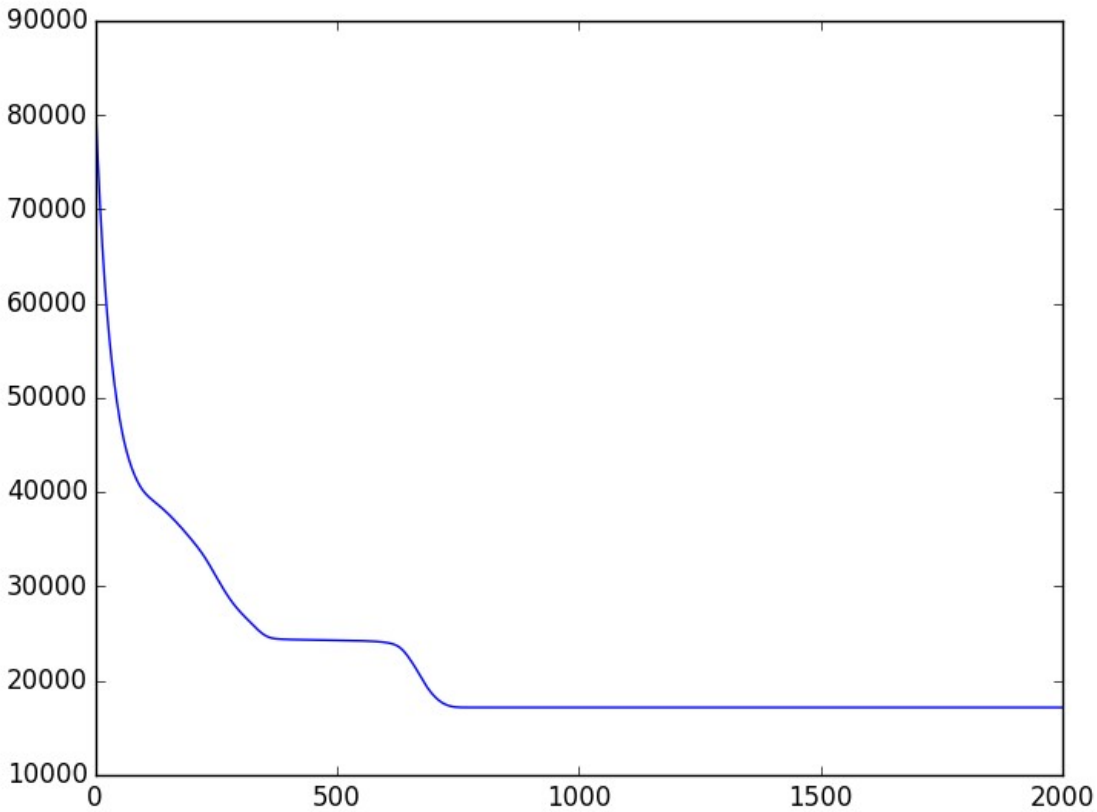


Figure 2.2.2-1: The loss vs the number of updates for the learning of MoG when K = 3

2.2.3

The following displays the loss for the validation data and 2D scatter for each value of K = 1, 2, 3, 4, 5.

When K = 1, the loss of validation data is:

```
('update_times:', 999, 'TRAIN_loss:', 23309.193734817411)
('update_times:', 999, 'VALID_loss:', 11607.877431670622)
```

Loss of validation data = 11607.9

The 2D scatter plot of data points colored by their cluster assignments shown as follows:

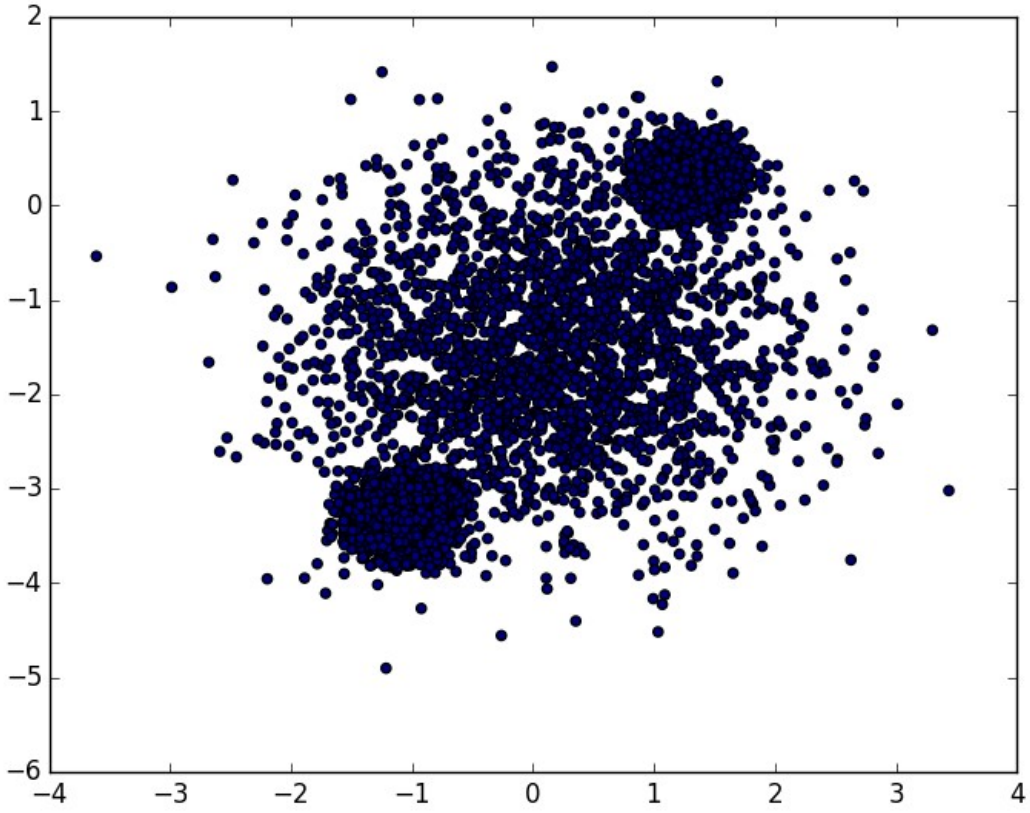


Figure 2.2.3-1: 2D scatter plot of data points colored by their cluster assignments when K = 1

When K = 2, the loss of validation data is:

```
('update_times:', 1999, 'TRAIN_loss:', 16141.105124180418)
('update_times:', 1999, 'VALID_loss:', 7998.6322423649799)
```

Loss of validation data = 7998.6

The 2D scatter plot of data points colored by their cluster assignments shown as follows:

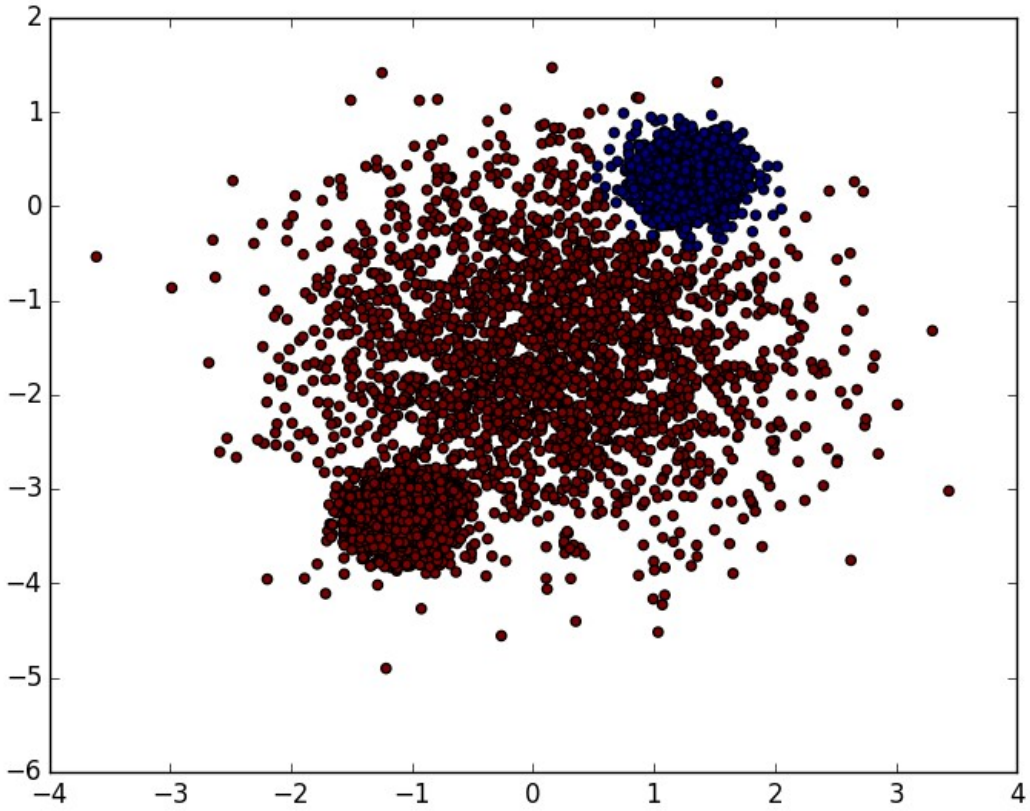


Figure 2.2.3-2: 2D scatter plot of data points colored by their cluster assignments when K = 2

When K = 3, the loss of validation data is:

```
('update_times:', 1999, 'TRAIN_loss:', 11394.524421578455)
('update_times:', 1999, 'VALID_loss:', 5741.6473403292975)
```

Loss of validation data = 5741.6

The 2D scatter plot of data points colored by their cluster assignments shown as follows:

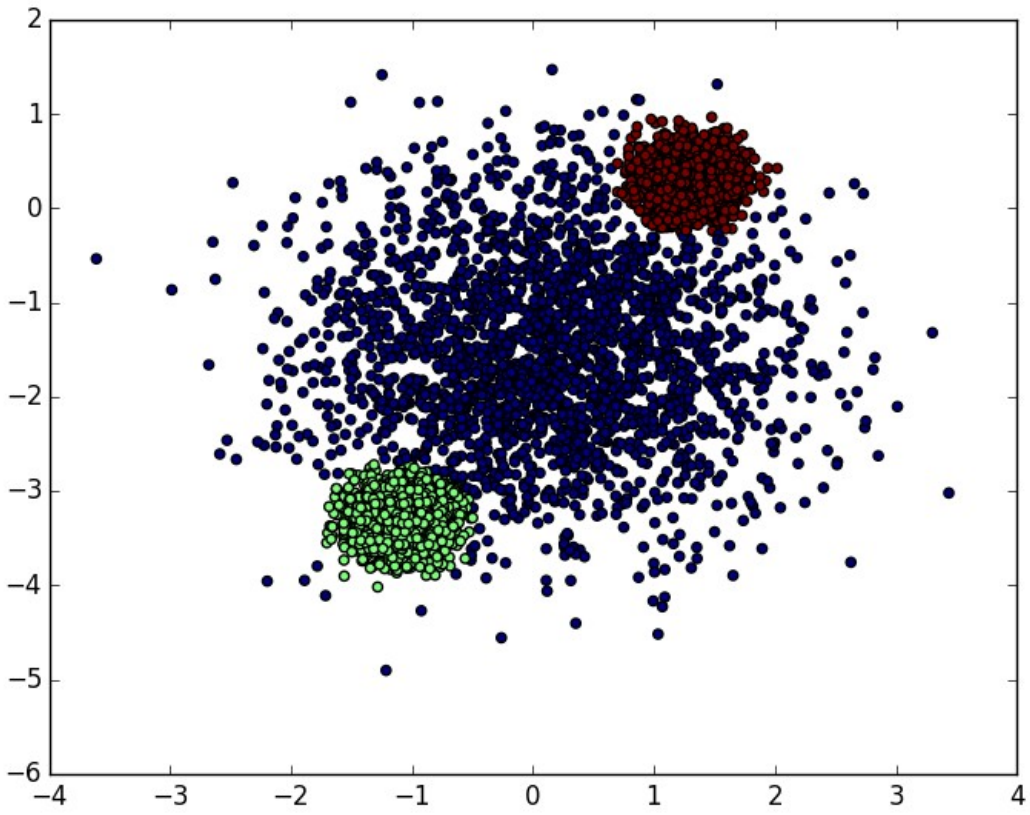


Figure 2.2.3-3: 2D scatter plot of data points colored by their cluster assignments when K = 3

When K = 4, the loss of validation data is:

```
('update_times:', 1999, 'TRAIN_loss:', 11390.297632495069)
('update_times:', 1999, 'VALID_loss:', 5742.7757431865793)
```

Loss of validation data = 5742.8

The 2D scatter plot of data points colored by their cluster assignments shown as follows:

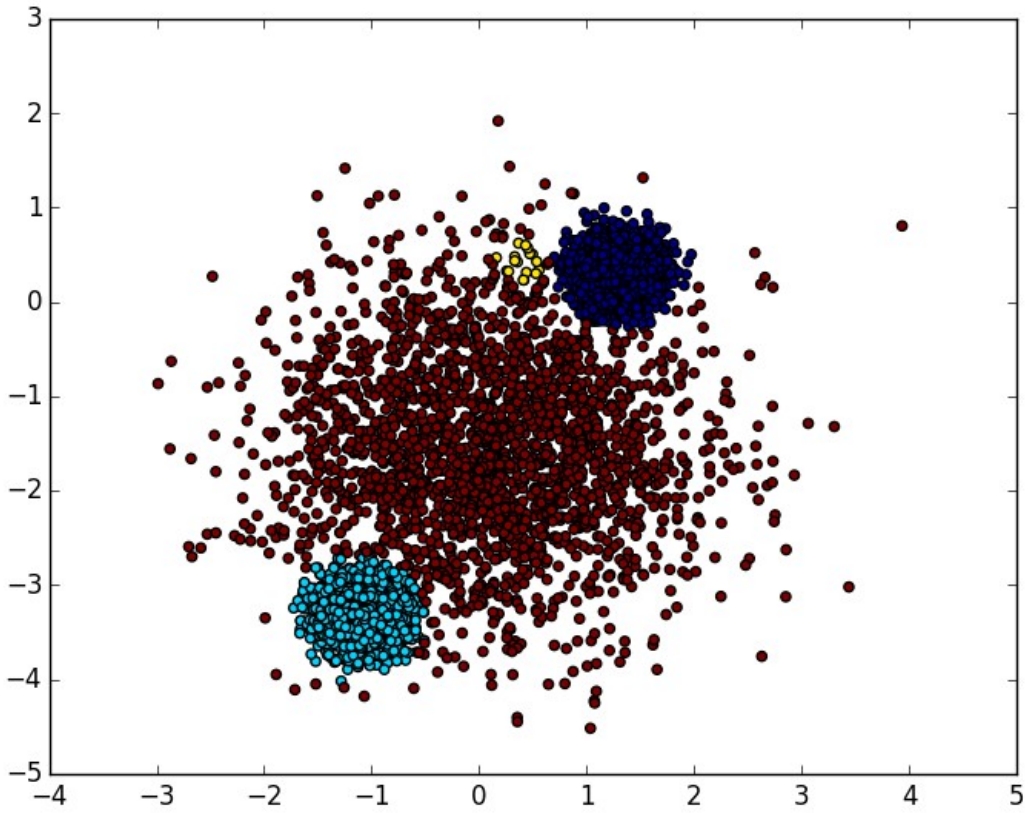


Figure 2.2.3-4: 2D scatter plot of data points colored by their cluster assignments when K = 4

When K = 5, the loss of validation data is:

```
('update_times:', 1999, 'TRAIN_loss:', 11388.660275056844)
('update_times:', 1999, 'VALID_loss:', 5741.4045208268108)
```

Loss of validation data = 5741.4

The 2D scatter plot of data points colored by their cluster assignments shown as follows:

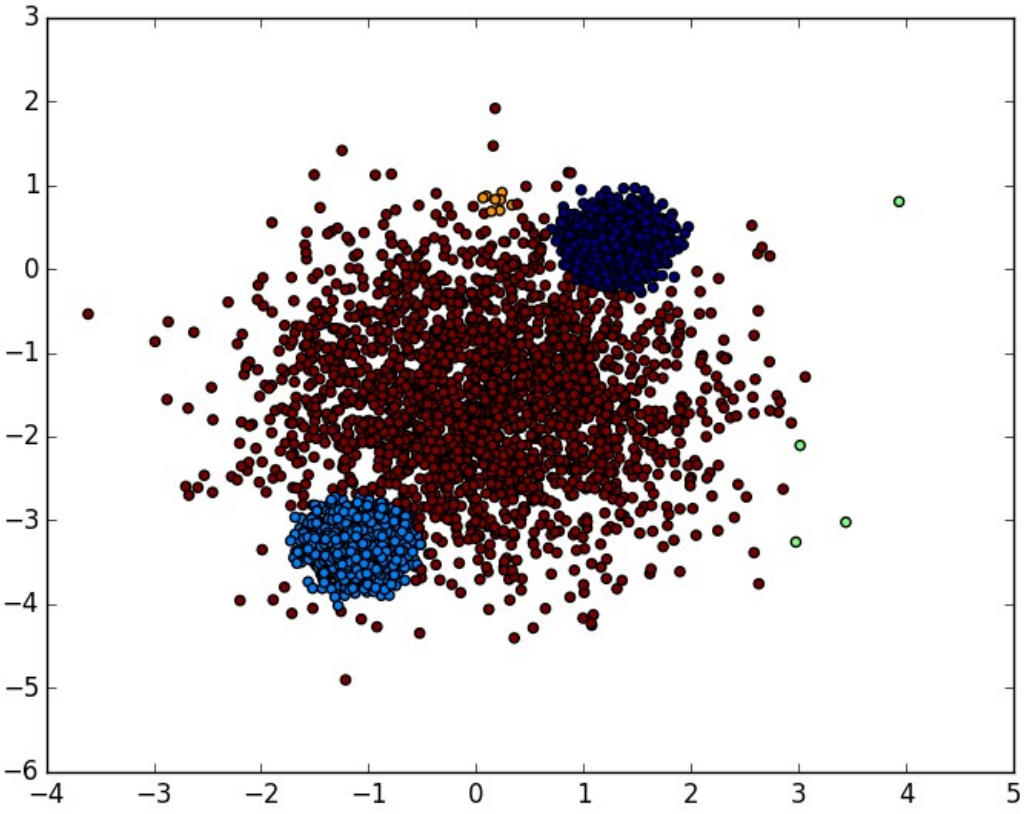


Figure 2.2.3-5: 2D scatter plot of data points colored by their cluster assignments when K = 5

K	1	2	3	4	5
Loss of validation data	11607.9	7998.6	5741.6	5742.8	5741.4

Based on the loss of validation data and the 2D scatter plots, 3 clusters is the best choice.

By viewing all the validation loss for K = 1, 2, 3, 4, 5, the loss decreases dramatically from the case of K = 1 to K = 2 and from K = 2 to K = 3, while the loss of validation data almost does not change when K is larger than 3. When K is larger than 3, the increasing of cluster numbers basically does not affect the loss of validation data.

2.2.4

For 100D data points, the following displays the loss for validation data and percentage of each cluster for both K-means model and Mixtures of Gaussians model when K = 1, 2, 3, 4, 5 separately.

K-means model:

Number of cluster	Validation loss	Percentage of each cluster
1	332901	1
2	265282	0.79, 0.21
3	200084	0.59, 0.20, 0.21
4	189731	0.50, 0.30, 0.09, 0.11
5	71826	0.29, 0.21, 0.10, 0.20, 0.20
6	71825	0, 0.20, 0.10, 0.30, 0.20 ,0.20
7	70619	0.07, 0.20, 0.30, 0.06, 0.10, 0.20, 0.07
8	70616	0.20, 0.06, 0.07, 0.07, 0.20, 0.30, 0.10, 0
9	70695	0.07, 0, 0.07, 0.20, 0.10, 0, 0.20, 0.06, 0.30
10	70628	0.07, 0.10, 0.06, 0, 0.07, 0.30, 0, 0.20, 0.20, 0

By viewing the result of K-means model, the validation loss almost does not decrease when the number of cluster is larger than 5. Basically, there is an “elbow” (similarly to *Figure 1.1.4 cluster number vs. validation loss*) when K = 5 in this case. Meanwhile, it happens that there are some clusters that do not have any data points assigned in when number of clusters is larger than 5. Basically, that means that some of the clusters are useless when number of clusters is really large. Thus, 5 is the best number of clusters for K-means model.

Mixtures of Gaussians model:

Number of cluster	Validation loss	Sigma	Percentage of each cluster
1	472993	1	1
2	435526	0.94, 0.21	0.79,0.21
3	351064	0.30, 0.82, 0.88	0.20, 0.50, 0.30
4	267493	0.75,0.69,0.30,0.35	0.41, 0.19, 0.19, 0.21
5	247134	0.35,3.0,0.35,0.49,0.88	0.20, 0.0003, 0.20, 0.31, 0.29
6	367359	5.7,0.21,0.88,0.52,0.80,0.49	0, 0, 0.30, 0, 0.40, 0.30
7	347121	4.22, 0.62, 10.13, 0.62, 0.90, 0.75, 0.30	0, 0, 0, 0, 0.40, 0.41, 0.19
8	161909	0.11, 0.35, 0.33, 0.87, 0.41, 0.69, 0.30, 0.49	0.10, 0.20, 0, 0, 0, 0.20, 0.20, 0.30
9	288135	0.35, 0.69, 0.61, 0.35, 0.49, 5.2, 0.52, 2.5, 0.76	0.10, 0.20, 0, 0.10, 0.30, 0, 0, 0, 0.30
10	161900	0.55, 0.30, 0.29, 1.5, 0.355, 0.55, 0.49, 0.155, 0.70, 0.112	0, 0.19, 0, 0, 0.20, 0, 0.31, 0, 0.20, 0.10

By reviewing the result of MoG model, when K=8 and K=10, we get the minimum validation loss. However, when K=8 and K=10, some clusters' percentages equal to 0. There is only 5 effective clusters in both K=8 and K=10. On the other hand, there is also an "elbow" (similarly to *Figure 1.1.4 cluster number vs. validation loss*) when K = 5. Basically, that means that some of the clusters are useless when number of clusters is really large. Thus, 5 is the best number of clusters for MoG model.

Comparing the two models, we can summarize that there always exist an "elbow" on the curve of validation loss vs number of clusters and we usually select the best number of clusters at the "elbow" point. When the number of clusters is larger than the "elbow", the loss of validation does not decrease as fast as before. Meanwhile, when the number of clusters is larger than "elbow" point, it is more frequently to see that some of the clusters do not have data points assigned in. The difference between these two models is that K-means is a type of hard assignment while MoG assigned data depending on probability. Thus, MoG is a more accurate and stable model if we have a dataset with Gaussian distribution.

3. Discover Latent Dimensions

3.1 Factor Analysis

3.1.1

$$\begin{cases} P(x) = \mathcal{N}(x; \mu, \Lambda^{-1}) \\ P(y|x) = \mathcal{N}(y; Ax+b; L^{-1}) \\ P(y) = \mathcal{N}(y; A\mu+b, L^{-1} + A\Lambda^{-1}A^T) \end{cases}$$

Given $P(s) = \mathcal{N}(s; 0, I)$, $P(x|s) = \mathcal{N}(x; Ws+\mu, \Psi)$

$$\Rightarrow P(x) = \int_s P(x|s)P(s)ds = \mathcal{N}(x; W\cdot 0+\mu, \Psi+WIW^T)$$

$$= \mathcal{N}(x; \mu, \Psi+WW^T)$$

$$\Rightarrow \log P(x) = \log \int_s P(x|s)P(s)ds = \log \mathcal{N}(x; \mu, \Psi+WW^T)$$

3.1.2

After training a factor analysis model setting the number of latent dimension K = 4 with the tiny hand-written digits dataset, the training, validation and test marginal log likelihood is reported as follows:

```
('train_err:', 8284.4261544935016)
('valid_err:', 1219.9187309161912)
('test_err:', 4545.997875407098)
```

Training marginal log likelihood	8284.4
Validation marginal log likelihood	1219.9
Test marginal log likelihood	4546.0

For each of the four rows of the learnt weight matrix, four 8x8 images can be plotted as follows:

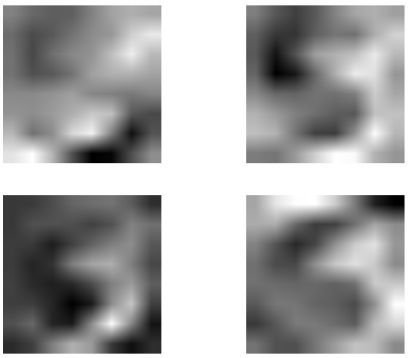


Figure 3.1.2: Plot of each row of the learnt weight matrix as four 8x8 images

As shown on the visualization graph, the top two images of the four 8x8 images should be class “5” while the bottom two images of the four images should be class “3”. This means that the first two latent dimensions represent class “5” while the last two latent dimensions represent class “3”. In another word, we might use only two latent factors to represent these two classes, one for representing class “3”, another one for representing class “5”.

3.1.3

In order to present on what directions that PCA and FA learnt, we can observe the first component of PCA and the weight matrix of FA respectively.

For the first component of PCA on all x1, x2, x3 directions:

```
[x1 x2 x3] =  [ 0.00678037  0.00677852  0.99995404]
```

Thus, PCA learns the maximum variance direction which is the x3 direction.

For the weight matrix of FA on all x1, x2, x3 directions:

```
[x1 x2 x3] =  [[-1.03498876 -1.03370559  0.06448168]]
```

Thus, FA learns the maximum correlation direction which is the x1 and x2 direction.

The following 3D plots can intuitively have the idea of on which directions PCA and FA learns:

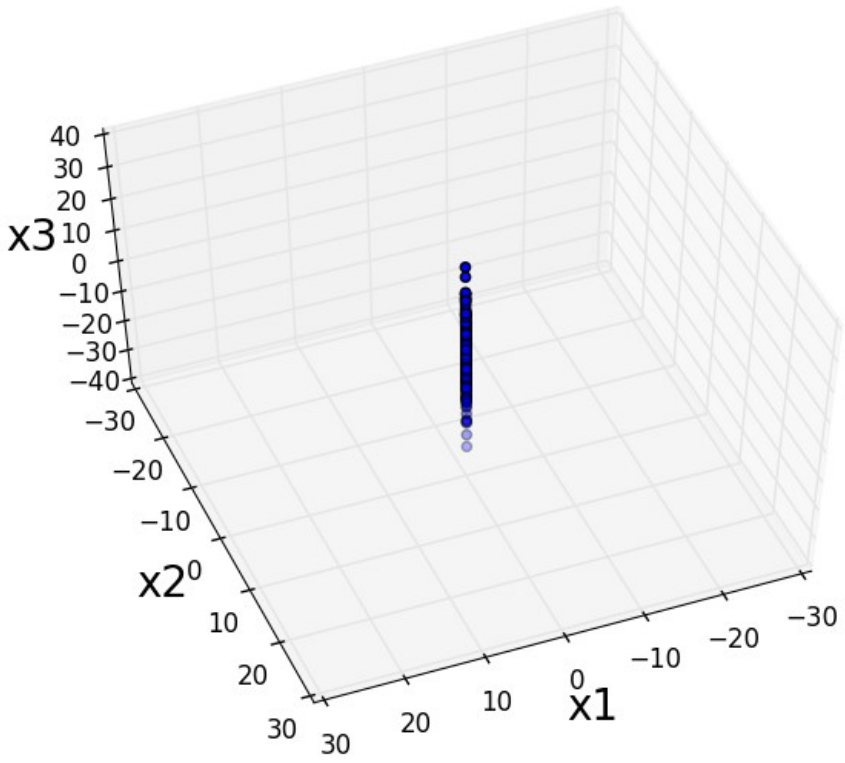


Figure 3.1.3-1: 3D plot for showing the direction PCA learns

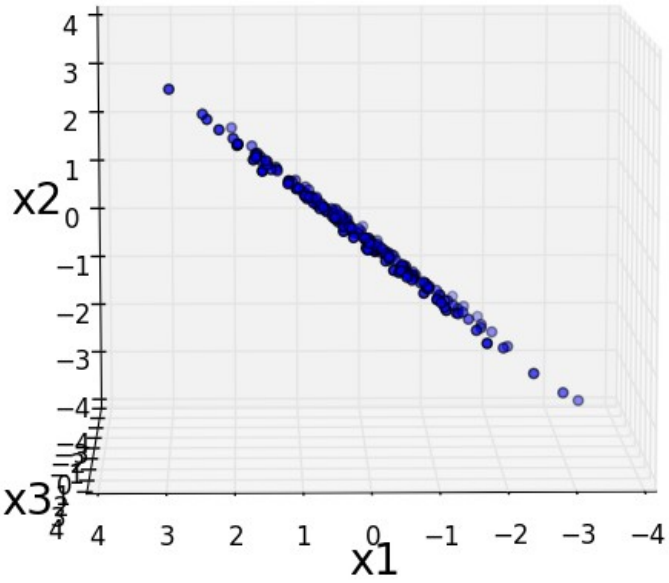


Figure 3.1.3-2: 3D plot for showing the direction FA learns

Appendix

1.K-means cluster:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

input_data = np.load('data2D.npy')
#x = input_data[:, 0]
#y = input_data[:, 1]

# data features
data_num = np.shape(input_data)[0]
data_dim = np.shape(input_data)[1]
np.random.seed(521)
tf.set_random_seed(521)
#print(data_num,data_dim)
#plt.plot(x,y,'*')
#plt.show()
K=3

sess = tf.InteractiveSession()

def Euclidean_dis(X, Y):
  '''
  return the matrix containing the pairwise Euclidean distances
  '''

  X_b = tf.expand_dims(X, 1)
  result = X_b - Y
  result_square = tf.square(result)
  Euclidean_dis = tf.reduce_sum(result_square, 2)

  return Euclidean_dis



def graph(K,n):


  #define variables
  data = tf.placeholder(tf.float64, [None,data_dim], name='input_x')
  centorid =  tf.cast(tf.Variable(tf.random_normal([K, data_dim])),tf.float64)

  #calculate the distance, then assignment each point to the nearest centeroid
  Dis = Euclidean_dis(data,centorid)
  assignment = tf.arg_min(Dis,1)
  assignment = tf.cast(tf.one_hot(assignment,K),tf.float64)

  #recalculate the centeroid
  #point_num = tf.expand_dims(tf.reduce_sum(assignment,0),1)
  #U_total = tf.matmul(tf.transpose(assignment),data)
  #U = U_total / point_num

  #difine Loss funtion
  U = tf.matmul(assignment,centorid)
  distance = data - U
  Loss = tf.reduce_sum(tf.square(distance))
```

```
    #define train algorithm
    optimizer = tf.train.AdamOptimizer(learning_rate=n,beta1=0.9,beta2=0.99,epsilon=1e-5)
    train = optimizer.minimize(loss=Loss)

    #print('center:',sess.run(centorid))
    #print('data',sess.run(data))
    #print('center',sess.run(centorid))
    #print('assignment:',sess.run(assignment))
    #print('dis_center:',sess.run(distance))
    return train,Loss,centorid,data

def runMult(K,n):
    train_err = []
    # Build computation graph
    train,loss,centorid,data = graph(K,n)

    # Initialize session
    init = tf.initialize_all_variables()
    sess.run(init)

    #train model

    for i in np.arange(400):
        _, err, center= sess.run([train, loss, centorid], feed_dict={data:input_data})
        train_err.append(err)
        print('update_times:',i,'loss:',err)

    #x = center[:, 0]
    #y = center[:, 1]
    #plt.plot(x,y,'*')
    #plt.show()
    print('mu:',center)
    plt.plot(train_err)
    plt.show()
runMult(K,0.05)
```

2.Mixture of Gaussians:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
tf.set_random_seed(521)
input_data = np.load('data2D.npy')
np.random.seed(521)
sess = tf.InteractiveSession()
data_num = np.shape(input_data)[0]
data_dim = np.shape(input_data)[1]
randIndx = np.arange(len(input_data))
np.random.shuffle(randIndx)
input_data = input_data[randIndx]
trainData = input_data[:6667]
validData = input_data[6667:]
K=3
n=0.1

def Euclidean_dis(X, Y):
  #return the matrix containing the pairwise Euclidean distances

  X_b = tf.expand_dims(X, 1)
  result = X_b - Y
  result_square = tf.square(result)
  Euclidean_dis = tf.reduce_sum(result_square, 2)

  return Euclidean_dis

def Gaussian_prob(X,mu,Vsigma):

  Dims = tf.shape(X)[1]
  pi = tf.constant(3.14159265359,tf.float64)
  dis = Euclidean_dis(X,mu)
  first_term = tf.cast(-Dims,tf.float64)/2.*tf.log(2.*pi*Vsigma*Vsigma)
  second_term = tf.div(-dis,2.*tf.square(Vsigma))
  log_gauss = first_term + second_term
  assignment = tf.arg_max(log_gauss, 1)

  return log_gauss, assignment



def jointprob(X,mu,prior,sigma):
  log_gauss, assignment = Gaussian_prob(X,mu,sigma)
  posterior = log_gauss+prior
  return posterior, assignment

def reduce_logsumexp(input_tensor, reduction_indices=1, keep_dims=False):

 max_input_tensor1 = tf.reduce_max(input_tensor, reduction_indices, keep_dims=keep_dims)
 max_input_tensor2 = max_input_tensor1
 if not keep_dims:
   max_input_tensor2 = tf.expand_dims(max_input_tensor2,
                                      reduction_indices)
 return tf.log(tf.reduce_sum(tf.exp(input_tensor - max_input_tensor2),
                             reduction_indices, keep_dims=keep_dims)) + max_input_tensor1

def logsoftmax(input_tensor):
 return input_tensor - reduce_logsumexp(input_tensor, keep_dims=True)
```

```
def log_prob(x,mean,prior,sigma):
  log_post = jointprob(x,mean,prior,sigma)
  posteriori = logsoftmax(log_post)
  return posteriori

def graph():
    data = tf.placeholder(tf.float64, [None,data_dim], name='input_x')
    mean =  tf.cast(tf.Variable(tf.random_normal([K, data_dim])),tf.float64)
    sigma = tf.cast(tf.Variable(tf.random_normal([K,])),tf.float64)
    sigma_exp = tf.exp(sigma)
    prior = tf.cast(tf.Variable(tf.random_normal([1,K])),tf.float64)
    prior_softmax = logsoftmax(prior)

    #define the graph
    log_post, assignment = jointprob(data,mean,prior_softmax,sigma_exp)
    marginprob = reduce_logsumexp(log_post,keep_dims = True)
    loss_f =-1* tf.reduce_sum (marginprob)

    #define train algorithm
    optimizer = tf.train.AdamOptimizer(learning_rate=n,beta1=0.9,beta2=0.99,epsilon=1e-5)
    train = optimizer.minimize(loss=loss_f)

    return train,data,mean,loss_f, assignment,sigma_exp,prior_softmax

def runMult():
    train_err = []
    # Build computation graph
    train,data,mean,loss, assignment ,a,b= graph()

    # Initialize session
    init = tf.initialize_all_variables()
    sess.run(init)

    #train model
    for i in np.arange(800):
        _, err, center, assign,si,pri= sess.run([train, loss, mean, assignment,a,b], feed_dict={data:trainData})
        train_err.append(err)
        print('update_times:',i,'loss:',err)
        #print('mean:',center)

    #plt.plot(train_err)
    assign = np.int32(assign)
    #print('class:', assign)
    x = input_data[:, 0]
    y = input_data[:, 1]
    #print('mean:', center)
    #print('prior:', np.exp(pri))
    #print('sigma:', si)
    plt.scatter(x,y,c=assign)
    plt.show()


runMult()
```

3.Factor Analysis:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import pylab
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

np.random.seed(510)
s1=np.random.normal(0,1,200)
s2=np.random.normal(0,1,200)
s3=np.random.normal(0,1,200)
x1=s1
x2=s1+0.001*s2
x3=10*s3

trainData = np.transpose(np.array([x1,x2,x3]))

sess = tf.InteractiveSession()
K = 1
n = 0.01

def Gaussian_prob(X,mu,var):
  Dims = tf.cast(tf.shape(X)[1], tf.float64)
  pi = tf.constant(3.14159265359,tf.float64)
  log_det = tf.reduce_sum(tf.log(tf.square(tf.diag_part(tf.cholesky(var)))))
  first_term = - (Dims/2.)*tf.log(2.*pi) - 0.5 * log_det
  second_term = -0.5 * tf.diag_part(tf.matmul(tf.matmul((X-mu), tf.matrix_inverse(var)), tf.transpose(X-mu)))
  log_gauss1 = first_term + second_term
  log_gauss = tf.reduce_sum(log_gauss1)

  return log_gauss, first_term

def graph():
  X = tf.placeholder(tf.float64, [None,3], name='input_x')
  Mu =  tf.Variable(tf.truncated_normal(shape=[1,3]), name='Mu')
  W = tf.Variable(tf.truncated_normal(shape=[K,3]), name='weights')
  Phi_ =  tf.Variable(tf.truncated_normal(shape=[3,]), name='phi')

  Mu = tf.cast(Mu, tf.float64)
  W = tf.cast(W, tf.float64)
  Phi_ = tf.cast(tf.exp(Phi_), tf.float64)

  #a = tf.cast(tf.zeros([3,3]),tf.float64)
  phi = tf.cast(tf.diag(Phi_), tf.float64)
  #phi = tf.cast(tf.matrix_set_diag(tf.cast(tf.zeros([64,64]),tf.float64), tf.abs(Phi_)), tf.float64)
  var = phi + tf.matmul(tf.transpose(W),W)


  log_px, f = Gaussian_prob(X, Mu, var)
  #logpxsum = tf.reduce_sum(log_px)
  loss = -log_px

  optimizer = tf.train.AdamOptimizer(learning_rate=n,beta1=0.9,beta2=0.99,epsilon=1e-5)
  train = optimizer.minimize(loss)

  return train, loss, X, Mu, W, log_px, f
```

```
def runMult():
    train_err = []
    #valid_err = []
    logpxsum_train = []
    #logpxsum_valid = []
    # Build computation graph
    train, loss, data, Mu, W, var, f = graph()

    # Initialize session
    init = tf.initialize_all_variables()
    sess.run(init)

    #train model
    for i in np.arange(5000):
        _, err, mean, weight, variance, first = sess.run([train, loss, Mu, W, var, f], feed_dict={data:trainData})
        train_err.append(err)
        logpxsum_train.append(variance)
        print('update_times:',i)
        #print('update_times:',i,'TRAIN_v:',variance)
        #print('mean:',mean)
        print('[x1 x2 x3] = ',weight)
        #print('var:',mean)
        #print(sess.run(tf.shape(variance)))

#       errValid, varvalid = sess.run([loss, var], feed_dict= {data: validData})
#       valid_err.append (errValid)
#       logpxsum_valid.append(varvalid)
        #print('update_times:',i,'VALID_loss:',errValid)
        #print('update_times:',i,'valid_v:',varvalid)

    kk = trainData * weight
    ax.scatter(kk[:, 0], kk[:, 2], kk[:, 1])
    ax.set_xlabel("x1", fontsize=20)
    ax.set_ylabel("x3", fontsize=20)
    ax.set_zlabel("x2", fontsize=20)

    plt.ylim([-4, 4])
    #plt.plot(train_err)

    #plt.plot(valid_err)
    plt.show()
    #plt.plot(logpxsum_train)
    #plt.plot(logpxsum_valid)

#   assign = np.int32(assign)
#   print('class:', assign)
#   x = trainData[:, 0]
#   y = trainData[:, 1]
#
#   plt.scatter(x,y,c=assign)


runMult()
```

4.PCA:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
import pylab
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

s1=np.random.normal(0,1,200)
s2=np.random.normal(0,1,200)
s3=np.random.normal(0,1,200)
x1=s1
x2=s1+0.001*s2
x3=10*s3

data = np.transpose(np.array([x1,x2,x3]))
#plt.scatter(data[:,0],data[:,1],data[:,2])
#ax.scatter(data[:,0],data[:,1],data[:,2])
#data = np.array([[1.,3.],[4.,1.],[3.,3.],[7.,9.],[4.,9.],[7.,8.],[11.,2.],[1.,0.],[5.,8.]])

def doPCA():
  pca = PCA(n_components=1)
  pca.fit(data)
  return pca

pca = doPCA()
print(pca.explained_variance_ratio_)
first_pc = pca.components_[0]
#second_pc = pca.components_[1]

transformed_data = pca.transform(data)
kk = transformed_data * first_pc
ax.scatter(kk[:, 0], kk[:, 1], kk[:, 2])
#for i, j in zip(transformed_data, data):
  #plt.scatter(first_pc[0]*i[0], first_pc[1]*i[0], color = 'r')
  #plt.scatter(second_pc[0]*i[1], second_pc[1]*i[1], color = 'g')
  #plt.scatter(j[0], j[1], color = 'b')

print("[x1 x2 x3] = ", first_pc)
print("[x1 x2 x3] = ", kk)
#print(np.matmul(data,np.expand_dims(first_pc,1)))
ax.set_xlabel("x1",fontsize = 20)
ax.set_ylabel("x2",fontsize = 20)
ax.set_zlabel("x3",fontsize = 20)
pylab.xlim([-30,30])
pylab.ylim([-30,30])

plt.show()
```