ECE421 – Introduction to Machine Learning

Assignment 3

Unsupervised Learning and Probabilistic Models

Hard Copy Due: Friday, March 27 @ 3:00 PM, at BA3128
Code Submission Due: Friday, March 27 @ 5:00 PM, on Quercus

General Notes:

- Attach this cover page to your hard copy submission
- Please post assignment related questions on Piazza.

Please check section to which you would like the assignment returned.

Tutorial Sections:

☐ Tutorial 1: Thursdays 3-5pm (SF2202)
☐ Tutorial 2: Thursdays 3-5pm (GB304)
☐ Tutorial 3: Tuesdays 10-12 (SF2202)
☐ Tutorial 4: Fridays 9-11 (BA1230)

Group Members	
Name	Student ID
Tianqi Huang	1004029712
Yifang Pan	1004192759
Jason Li	1004372813

ECE421

Assignment 3

Tianqi Huang, Yifang Pan, Jason Li
Student Numbers: 1004029712, 1004192759, 1004372813

March 2020

1 K-Means

Notation

Data Vector x:

$$\overline{\mathbf{x}} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_D \end{pmatrix}$$

Data Matrix X:

$$\mathbf{X} = \begin{bmatrix} — & \overline{x_1}^T & — \\ — & \overline{x_2}^T & — \\ \vdots & & \\ — & \overline{x_N}^T & — \end{bmatrix}$$

Data Center $\overline{\mu}$:

$$\overline{\mu} = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_D \end{pmatrix}$$

Matrix of Data Centers μ:

$$\mu = \begin{bmatrix} — & \overline{\mu_1}^T & — \\ — & \overline{\mu_2}^T & — \\ \vdots & & \\ — & \overline{\mu_N}^T & — \end{bmatrix}$$

1.1 Learning K-means

Part 1: Training

Since this assignment is done through PyTorch, the code would be different from the TensorFlow implementation. The gradient descent code used to find k-means is shown below

```
  data = x_matrix
  num_of_epochs = 100

  # to initialize MU
  avg = torch.zeros((k, data.shape[1]))
  std = torch.ones(k, data.shape[1])
  MU = torch.normal(avg, std)
  MU.requires_grad = True

  # change data to a tensor
  data = torch.tensor(data)

  # set up optimizer
  optimizer = torch.optim.Adam([{'params': MU, 'lr': 0.1, 'betas': (0.9,
  ↪  0.99), 'eps': 1 * 10 ** -5}])

  losses = []
  for epoch in range(0, num_of_epochs):
    optimizer.zero_grad() # elimiate the gradient from last iteration
    loss = distanceLossFunction(data, MU) # calculate loss
    loss.backward() # backprop gradient
    optimizer.step() # update MU
    losses.append(loss.item())
```

The helper functions used in this algorithm is shown below. They are the distanceFunction that was asked to implemented in the assignemnt, and a function that uses it to calculate loss. Note the functions are implemented with the "tensor" object in pytorch, as the training loop is implemented in pytorch.

```
def distanceLossFunction(X, MU):
  pair_dist = distance_func_torch(X, MU)
  min = pair_dist.min(dim = 1).values
  return min.sum()

# distance Function, but change into tensors first
def distance_func_torch(X, MU):
  pair_dist = torch.zeros((X.size()[0], MU.size()[0]))
  ones = torch.ones((X.size()[0], 1))
  for i in range(0, MU.size()[0]):
```

```
    # Calculate the difference between one center mu with all sample
    col = X - torch.mm(ones, MU[i, :].reshape((MU[i, :].size()[0], 1)).T)
    # find the norm
    pair_dist[:, i] = col.norm(dim = 1)
  return pair_dist
```

The resultant loss curve for K = 3 is showed below. It can be seen that the loss stabilizes after around 50 updates

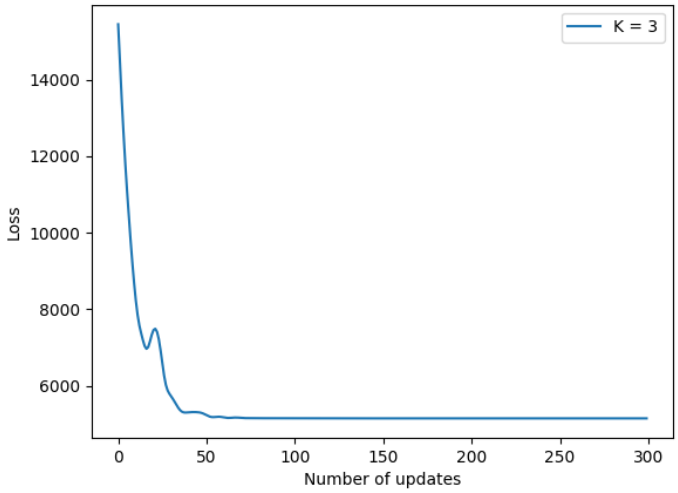


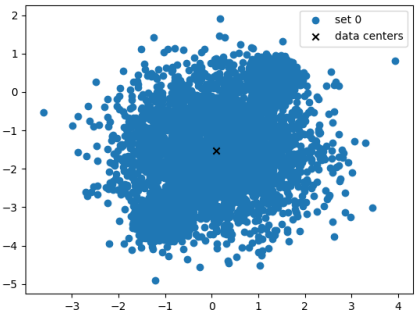
Figure 1: Loss vs updates

Part 2: Different Values of K

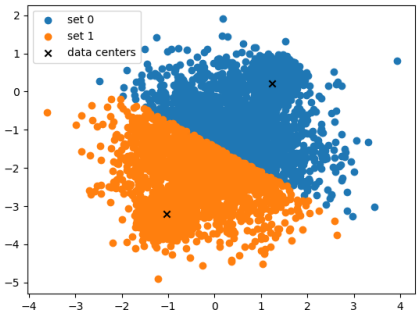
After running the algorithm through $K = 1, 2, 3, 4, 5$. The percentage of data point belonging to cluster k is shown in the table below:

	Percentage for Cluster K				
Number of Clusters	1	2	3	4	5
1	1	0	0	0	0
2	0.4968	0.5032	0	0	0
3	0.2428	0.3793	0.3779	0	0
4	0.3717	0.1159	0.37	0.1424	0
5	0.1346	0.1673	0.3687	0.211	0.1184

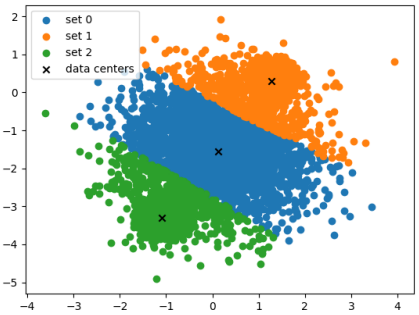
The scatter plots for the 5 different values of K is also shown below.



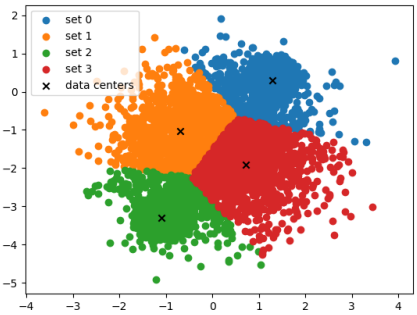
(a) K = 1



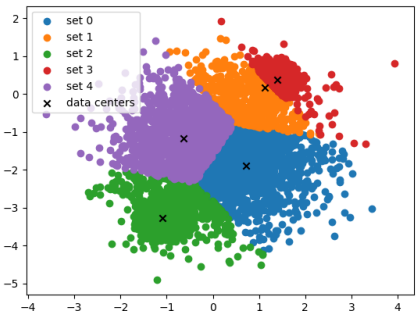
(b) K = 2



(c) K = 3



(d) K = 4



(e) K = 5

Figure 2: Scatter Plot of the 2-D data with different K values

From the percentages, we can see that for K=2, we have the most evenly distributed data points (it is approximately 50 percent each). From the Scatter graph, we can see that the data centers for K = 2 are also data centers for all results with $K \geq 2$. This might indicate that the data is best separated into 2 groups.

Part 3: Validation Set

In this section, μ matrix is trained only on the training set, which constitutes 2/3 of all data. The validation data has the following distributions across different clusters.

	Percentage for Cluster K					
Number of Clusters	1	2	3	4	5	Loss
1	1	0	0	0	0	6251.5674
2	0.5169	0.4830	0	0	0	2340.7021
3	0.3945	0.3714	0.2340	0	0	1681.7271
4	0.1212	0.1281	0.3651	0.3855	0	1452.8062
5	0.0864	0.0903	0.0954	0.3561	0.3717	1359.6371

From the table we can see that for 2 clusters, it still have the most evenly distributed data points. In addition, we can see the most significant reduction of loss when the number of clusters is increased from 1 to 2. Therefore we can conclude that 2 Clusters best separates the data set without overfitting.

2 Mixtures of Gaussians

2.1 The Gaussian Cluster Model

2.1.1

The probability density function for an arbitrary cluster is a multivariate Gaussian distribution represented by the following formula:

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \sigma^2) = \frac{1}{(2\pi)^{D/2}\,\sigma} \exp\left[-\frac{1}{2\sigma^2}(\mathbf{x}-\boldsymbol{\mu})^\top(\mathbf{x}-\boldsymbol{\mu})\right]$$

The code to derive the log probability given an input data matrix across all K clusters is provided below:

```
def distance_func_torch(X, MU):
  pair_dist = torch.zeros((X.size()[0], MU.size()[0]))
  ones = torch.ones((X.size()[0], 1))
  for i in range(0, MU.size()[0]):
    col = X - torch.mm(ones, MU[i, :].reshape((MU[i, :].size()[0], 1)).T)
    pair_dist[:, i] = col.norm(dim = 1)
  return pair_dist


def log_GaussPDF(X, mu, sigma):
    # Inputs
    # X: N X D
    # mu: K X D
    # sigma: K X 1
    # Outputs:
    # log Gaussian PDF N X K
    k = mu.shape[0]
```

```
    z = distance_func_torch(X, mu)
    log_gauss = torch.add(-X.shape[1] / 2 * np.log(2 * np.pi), -k / 2 *
    ↪  sigma)
    log_gauss = torch.add(log_gauss.T, -0.5 * torch.mul(z * z, (1 /
    ↪  torch.exp(sigma)).T))
    return log_gauss
```

2.1.2

The posterior probability is found by multiplying the multivariate Gaussian by a prior probability π for each individual k cluster.

$$P(z|\mathbf{x}) \propto \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_z, \sigma_z^2)\, \pi_z$$

To find the log posterior probability, we can take the sum across the log of the factors. Note that the denominator is not needed as it is constant for all values of k, and thus not useful for minimization during training.

It is also important to note that the prior probabilities are required to be positive constants between 0 and 1. Thus, we instead train the model on an unconstrained parameter and use its log-softmax in the probability formulas to satisfy this constraint. This is the reason we need to use the log-softmax function is to prevent rounding error from the exponent of large negative numbers producing a result of zero.
The code is shown below.

```
def log_posterior(log_PDF, log_pi):
    # Input
    # log_PDF: log Gaussian PDF N X K
    # log_pi: K X 1
    # Outputs
    # log_post: N X K
    real_log_pi = logsoftmax(torch.exp(log_pi))
    post = torch.add(log_PDF, real_log_pi.T)
    return post
```

Because PyTorch was used to produce the results presented in this report, the helper function for log-softmax was not used. Instead, the direct PyTorch implementation for used.

2.2 Learning the MoG

2.2.1

The epoch vs loss graph indicates that the MoG model converges rather quickly within around 50 epochs.

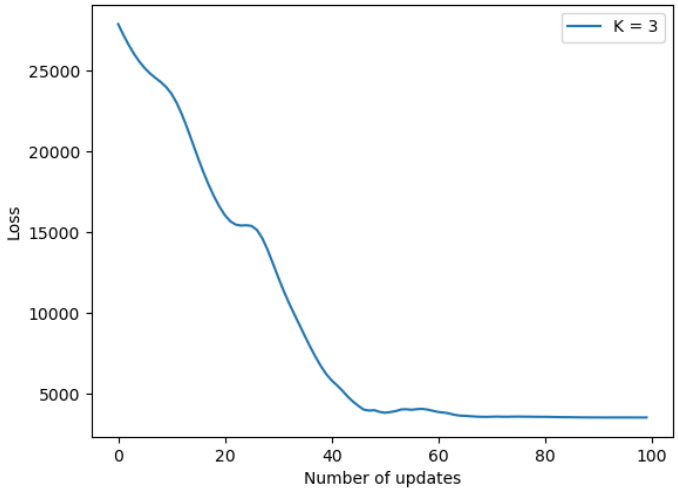


Figure 3: Loss vs updates for MoG model with K = 3

The final trained parameters are:

$$\boldsymbol{\mu}_1 = (1.30, 0.31)$$
$$\boldsymbol{\mu}_2 = (0.12, -1.52)$$
$$\boldsymbol{\mu}_3 = (-1.10, -3.31)$$

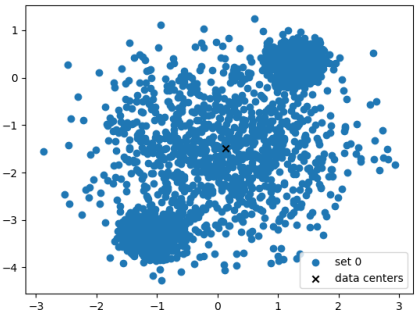
$$\sigma_1 = 0.027$$
$$\sigma_2 = 0.645$$
$$\sigma_3 = 0.026$$

$$\pi_1 = 0.336$$
$$\pi_2 = 0.332$$
$$\pi_3 = 0.332$$

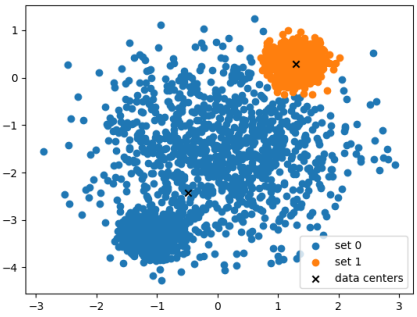
2.2.2

The loss function for used to train the model is the negative log loss of the posterior distribution across all input samples.

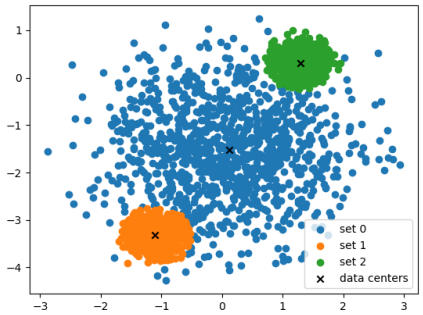
$$\mathcal{L} = -\sum_{n=1}^{N} \log \sum_{k}^{K} \mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_k, \sigma_k^2)\ \pi_k$$



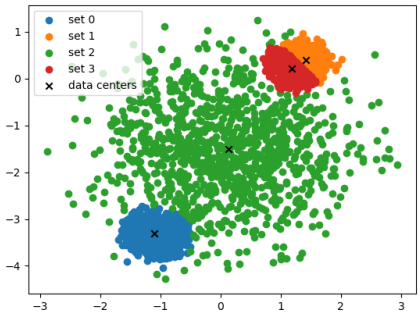
(a) K = 1



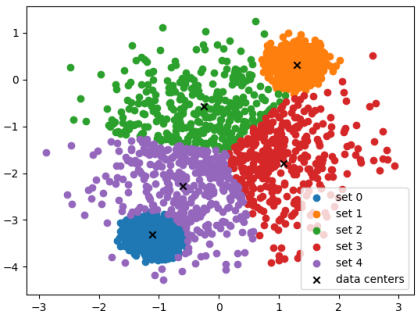
(b) K = 2



(c) K = 3



(d) K = 4



(e) K = 5

Figure 4: Scatter Plot of the 2-D Data for the MoG model across different K values.

	Percentage for Cluster K					
Number of Clusters	1	2	3	4	5	Loss
1	1	0	0	0	0	20070.2344
2	0.6615	0.3384	0	0	0	16158.5762
3	0.3138	0.3504	0.3357	0	0	3549.7183
4	0.3525	0.1713	0.3117	0.1644	0	-6164.3374
5	0.3423	0.3369	0.0993	0.1050	0.1164	-17763.5645

2.2.3

For the 100-D dataset, the validation results are summarized for both K-means and MoG in the two tables below:

K-means Algorithm Validation Loss	
Number of Clusters	Loss
5	41263.2461
10	28450.3105
15	28162.8535
20	28205.9238
30	27944.4023

MoG Algorithm Validation Loss	
Number of Clusters	Loss
5	664737.1250
10	674688.8125
15	673052.6250
20	670499.2500
30	653572.6250

For the K-means algorithm, the loss stops decreasing as $K \geq 10$. This indicates that the K-means algorithm predicts that there are 10 clusters in the 100-D dataset. The loss for the MoG algorithm stops decreasing at $K = 5$, meaning that the Gaussian model predicts there to be only 5 clusters in the dataset. From this, we can conclude that the original dataset is likely to be clustered within 5 to 10 clusters.