ECE 421 ASSIGNMENT 3

By Valentina Manferrari

Student ID: 1004796615

Link to the Google Colab document: https://colab.research.google.com/drive/1VEkTgAclgi-fB-7fj5Hm19nkAvM9kFv4?usp=sharing (https://colab.research.google.com/drive/1VEkTgAclgi-fB-7fj5Hm19nkAvM9kFv4?usp=sharing)

In [ ]:

```
from google.colab import drive
drive.mount('/content/drive')
```

In [ ]:

```
%%shell
jupyter nbconvert --to html /content/drive/MyDrive/ECE421/A3/ECE421_A3_code_manferrari.
ipynb
```

In [1]:

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import time
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
```

```
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/
python/compat/v2_compat.py:96: disable_resource_variables (from tensorflo
w.python.ops.variable_scope) is deprecated and will be removed in a future
version.
Instructions for updating:
non-resource variables are not supported in the long term
```

1. K-means

In [2]:

```
# Loading data
data = np.load('data2D.npy')
[num_pts, dim] = np.shape(data)

# If dealing with validation set validation flag needs to be turned on
validation = True
if validation:
  valid_batch = int(num_pts / 3.0)
  np.random.seed(421)
  rnd_idx = np.arange(num_pts)
  np.random.shuffle(rnd_idx)
  val_data = data[rnd_idx[:valid_batch]]
  data = data[rnd_idx[valid_batch:]]

# Distance function for K-means
# Inputs
#    X: is an NxD matrix (N observations and D dimensions)
#    MU: is an KxD matrix (K means and D dimensions)
# Outputs
#   pair_dist: is the pairwise distance matrix (NxK)
def distanceFunc(X, MU):
    pair_dist = tf.reduce_sum(tf.square(X), axis=1, keepdims=True) \
                - 2 * tf.matmul(X, tf.transpose(MU)) \
                + tf.reduce_sum(tf.square(tf.transpose(MU)), axis=0, keepdims=True)
    return pair_dist

# Squared Distance Loss for K-Means
def calculate_loss(X, MU):
    D = distanceFunc(X, MU)
    e = tf.reduce_min(D, axis=1)
    L = tf.reduce_sum(e)
    return L

# Partitions the data into K clusters based on MU
def cluster_assignments(X, MU):
    D = distanceFunc(X, MU)
    s = tf.argmin(D, axis=1)
    return s # returns a Nx1 vector of cluster assignments for x_1 - x_N
```

In [3]:

```
# K is used to define the number of clusters we want
def build_graph(K, learning_rate):
    tf.set_random_seed(124)
    X = tf.placeholder(tf.float32, shape=[None, dim], name="X")

    # Defining variables to learn
    MU = tf.get_variable('MU', shape=[K, dim],
                         initializer=tf.initializers.random_normal(mean=0, stddev=1))

    # Loss function
    loss = calculate_loss(X, MU)
    # Determining clusters assignment
    s = cluster_assignments(X, MU)
    # Initialization of GD optimizer (using Adam)
    optimizer = tf.train.AdamOptimizer( learning_rate=learning_rate,
                                       beta1=0.9, beta2=0.99, epsilon=1e-5).minimize(lo
ss)

    return optimizer, X, MU, s, loss
```

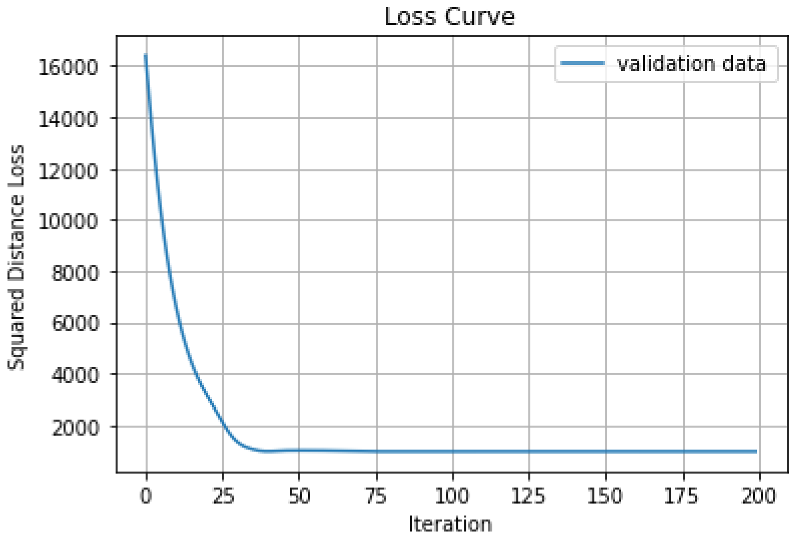
In [ ]:

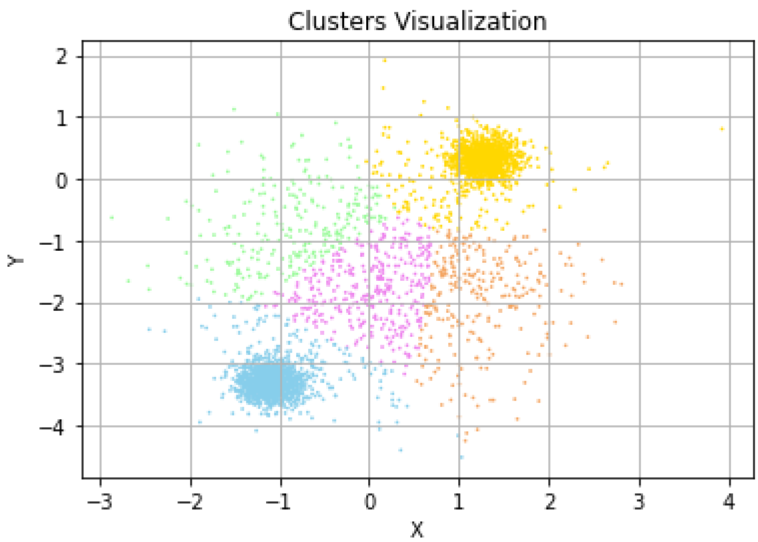
```
def train(K, learning_rate, n_epochs):
    optimizer, X, MU, s, loss = build_graph(K, learning_rate)
    global_init = tf.global_variables_initializer()

    with tf.Session() as sess:
        sess.run(global_init)
        loss_curves = {'train': [], 'valid': []}
        cluster_assignments = {}

        for iter in range(n_epochs):
            # Gradient descent step on dataset
            feed_dict_batch = {X: data}
            [_opt, _loss] = sess.run([optimizer, loss], feed_dict=feed_dict_batch)
            loss_curves['train'].append(_loss)
            #If dealing with validation dataset, get validation loss
            if validation:
                feed_dict_batch = {X: val_data}
                [_loss] = sess.run([loss], feed_dict=feed_dict_batch)
                loss_curves['valid'].append(_loss)

        # Getting assignment of clisters for training dataset
        feed_dict_batch = {X: data}
        [cluster_assignments['train']] = sess.run([s], feed_dict=feed_dict_batch)
        # Getting assignment of clisters for validation dataset
        if validation: # (only for validation == True)
            feed_dict_batch = {X: val_data}
            [cluster_assignments['valid']] = sess.run([s], feed_dict=feed_dict_batch)
        # Getting learned K-Means clusters
        [MU] = sess.run([MU], feed_dict={})

    return MU, loss_curves, cluster_assignments
```

In [4]:

```
def main():
    K = 5
    MU, loss, cluster_assignments = train(K=K, learning_rate=0.1, n_epochs=200)

    # Print out the final training & validation Loss
    print("Training Loss:", loss['train'][-1])
    type = 'train'
    if validation:
        print("Validation Loss:", loss['valid'][-1])
        type = 'valid'

    # Printing out final cluster distribution
    for cluster in range(K):
        print("Cluster {}:\n\t# Points: {}\n\tPercent of Points: {}".format(
            cluster,
            np.sum(cluster_assignments[type]==cluster),
            np.mean(cluster_assignments[type]==cluster))
        )
    print("MU:\n", MU)

    # Plotting loss curve
    plt.plot(loss[type], label='training data' if not validation else 'validation data'
)
    plt.legend()
    plt.title('Loss Curve')
    plt.ylabel('Squared Distance Loss')
    plt.xlabel('Iteration')
    plt.grid()
    plt.show()

    # Plotting cluster assignment
    colors = ['gold','palegreen','skyblue','violet', 'sandybrown']
    plt.scatter(
        data[:,0] if not validation else val_data[:,0],
        data[:,1] if not validation else val_data[:,1],
        s=0.5,
        c=cluster_assignments[type],
        cmap=matplotlib.colors.ListedColormap(colors)
    )
    plt.title('Clusters Visualization')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.grid()
    plt.show()

if __name__ == '__main__':
    main()
```

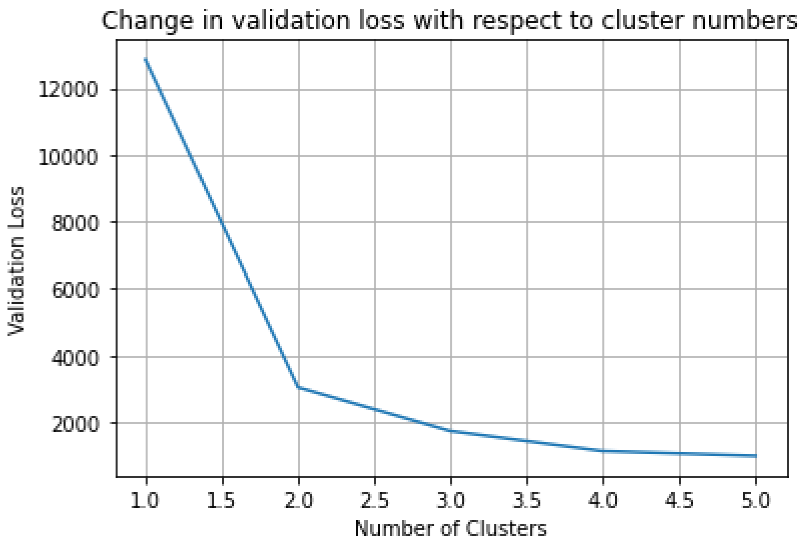
```
--- 0.7029011249542236 seconds ---
Training Loss: 1886.228
Validation Loss: 985.74207
Cluster 0:
	# Points: 1223
	Percent of Points: 0.36693669366936693
Cluster 1:
	# Points: 257
	Percent of Points: 0.0771077107710771
Cluster 2:
	# Points: 1244
	Percent of Points: 0.3732373237323732
Cluster 3:
	# Points: 339
	Percent of Points: 0.10171017101710171
Cluster 4:
	# Points: 270
	Percent of Points: 0.081008100810081
MU:
 [[ 1.2610028   0.2677277 ]
 [-0.9037934  -0.7265959 ]
 [-1.0784869  -3.271028  ]
 [-0.00714005 -1.778053  ]
 [ 1.3048029  -1.9214578 ]]
```





In [3]:

```
x = [1,2,3,4,5]
y = [12856.141, 3042.2302, 1731.768, 1129.8677, 985.74207]
plt.plot(x,y)
plt.title('Change in validation loss with respect to cluster numbers')
plt.ylabel('Validation Loss')
plt.xlabel('Number of Clusters')
plt.grid()
plt.show()
```



2. Mixture of Gaussians

2.1 The Gaussian cluster mode

In [2]:

```
data = np.load('data2D.npy')
#data = np.load('data100D.npy')
[num_pts, dim] = np.shape(data)

# Constants
pi = 3.141592654

# For Validation set
validation = True
if validation:
  valid_batch = int(num_pts / 3.0)
  np.random.seed(421)
  rnd_idx = np.arange(num_pts)
  np.random.shuffle(rnd_idx)
  val_data = data[rnd_idx[:valid_batch]]
  data = data[rnd_idx[valid_batch:]]

# Distance function for GMM
# Inputs
#   X: is an NxD matrix (N observations and D dimensions)
#   MU: is an KxD matrix (K means and D dimensions)
# Outputs
#   pair_dist: is the pairwise distance matrix (NxK)
def distanceFunc(X, MU):
    pair_dist = tf.reduce_sum(tf.square(X), axis=1, keepdims=True) \
                - 2 * tf.matmul(X, tf.transpose(MU)) \
                + tf.reduce_sum(tf.square(tf.transpose(MU)), axis=0, keepdims=True)

    return pair_dist

# Inputs
#   X of sise N X D; MU of size K X D; Sigma of size 1 X K
# Outputs:
#   log Gaussian PDF of size N X K
def log_GaussPDF(X, MU, sigma):
    pair_dist = distanceFunc(X, MU) #same as defined in Part 1
    log_PDF = - dim * tf.log(sigma * np.sqrt(2*pi)) - pair_dist / (2 * tf.square(sigma
))
    return log_PDF


# Input
#   log Gaussian PDF of size N X K; log_pi of size 1 X K
# Outputs
#   log_post of size N X K
def log_posterior(log_PDF, log_pi):
    Z = log_PDF + log_pi
    log_post = Z - reduce_logsumexp(Z, reduction_indices=1, keep_dims=True)
    return log_post

# Input
#   X of size N X D; MU of K X D; sigma of size 1 X K; weights aka. P(k) of size 1 X K
# Outputs
#   loss (constant)
def calculate_loss(X, MU, sigma, w):
    P = log_GaussPDF(X, MU, sigma)
    Q = tf.reduce_logsumexp(P + tf.log(w), reduction_indices=1, keep_dims=True)
    loss = - tf.reduce_sum(Q, reduction_indices=0, keep_dims=False)
    return loss
```

```python
# Returns a Nx1 vector of cluster assignments
def cluster_assignments(X, MU, sigma, w):
    log_PDF = log_GaussPDF(X, MU, sigma)
    P_j_x = log_posterior(log_PDF, tf.log(w))
    s = tf.argmax(P_j_x, axis=1)
    return s
```

2.2 Learning the MoG

In [3]:

```python
# functions taken from the helper file provided

def reduce_logsumexp(input_tensor, reduction_indices=1, keep_dims=False):
  """Computes the sum of elements across dimensions of a tensor in log domain.

     It uses a similar API to tf.reduce_sum.

  Args:
    input_tensor: The tensor to reduce. Should have numeric type.
    reduction_indices: The dimensions to reduce.
    keep_dims: If true, retains reduced dimensions with length 1.
  Returns:
    The reduced tensor.
  """
  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):
  """Computes normal softmax nonlinearity in log domain.

     It can be used to normalize log probability.
     The softmax is always computed along the second dimension of the input Tensor.

  Args:
    input_tensor: Unnormalized log probability.
  Returns:
    normalized log probability.
  """
  return input_tensor - reduce_logsumexp(input_tensor, reduction_indices=1, keep_dims=T
rue)
```

In [ ]:

```
# K is used to define the number of clusters we want
def build_graph(K, learning_rate):
    tf.set_random_seed(421)
    X = tf.placeholder(tf.float32, shape=[None, dim], name="X")

    # Defining variables to learn
    init = tf.initializers.random_normal(mean=0, stddev=1)
    MU = tf.get_variable('MU',
        shape=[K, dim],
        initializer=init)

    sigma_unconstrained = tf.get_variable('sigma_unconstrained',
        shape=[1, K],
        initializer=tf.initializers.random_normal(mean=0, stddev=1))
    sigma = tf.exp(sigma_unconstrained, name='sigma')

    w_unconstrained = tf.get_variable('weight_unconstrained',
        shape=[1, K],
        initializer=tf.initializers.random_normal(mean=0, stddev=1))
    ln_w = logsoftmax(w_unconstrained)
    w = tf.exp(ln_w, name='weight')

    # Loss function
    loss = calculate_loss(X, MU, sigma, w)
    # Determining clusters assignment
    s = cluster_assignments(X, MU, sigma, w)
    # Initialization of GD optimizer (using Adam)
    optimizer = tf.train.AdamOptimizer(
        learning_rate=learning_rate,
        beta1=0.9,
        beta2=0.99,
        epsilon=1e-5
    ).minimize(loss)

    return optimizer, X, MU, sigma, w, s, loss
```

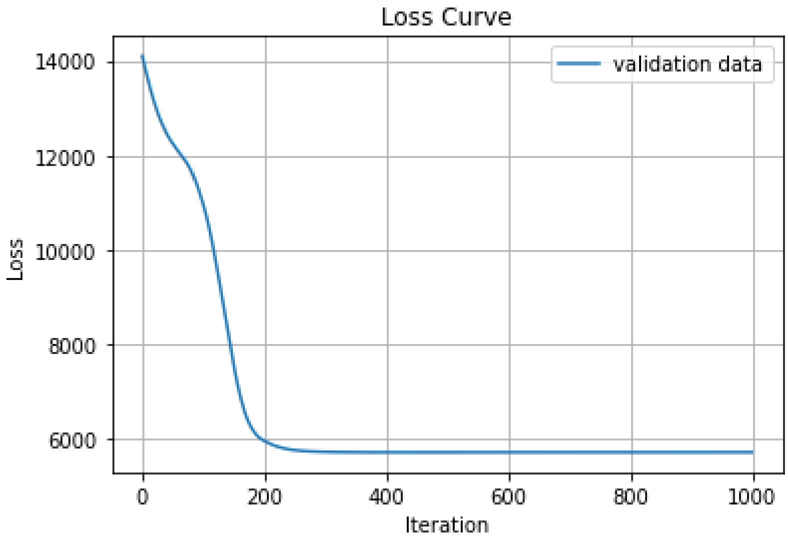
In [ ]:

```
def train(K, learning_rate, n_epochs):
    optimizer, X, MU, sigma, w, s, loss = build_graph(K, learning_rate)
    global_init = tf.global_variables_initializer()

    with tf.Session() as sess:
        sess.run(global_init)
        loss_curves = {'train': [], 'valid': []}
        cluster_assignments = {}

        for iter in range(n_epochs):
            # Gradient descent step on dataset
            feed_dict_batch = {X: data}
            [_opt, _loss] = sess.run([optimizer, loss], feed_dict=feed_dict_batch)
            loss_curves['train'].append(_loss)
            #If dealing with validation dataset, get validation loss
            if validation:
                feed_dict_batch = {X: val_data}
                [_loss] = sess.run([loss], feed_dict=feed_dict_batch)
                loss_curves['valid'].append(_loss)

        # Getting assignment of clisters for training dataset
        feed_dict_batch = {X: data}
        [cluster_assignments['train']] = sess.run([s], feed_dict=feed_dict_batch)
        # Getting assignment of clisters for validation dataset
        if validation: # (only for validation == True)
            feed_dict_batch = {X: val_data}
            [cluster_assignments['valid']] = sess.run([s], feed_dict=feed_dict_batch)

        # Getting learned K-Means clusters
        [MU, sigma, w] = sess.run([MU, sigma, w], feed_dict={})

    return MU, sigma, w, cluster_assignments, loss_curves
```

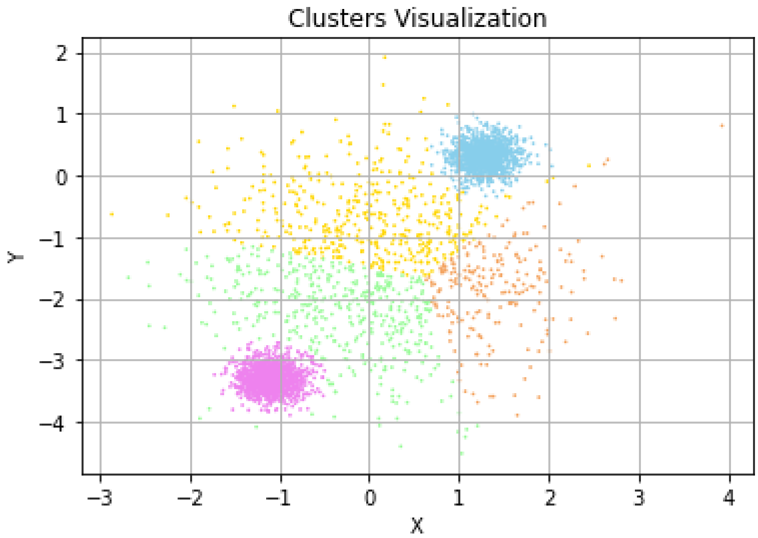
In [4]:

```
def main():
    K = 5
    MU, sigma, w, cluster_assignments, loss = train(K=K, learning_rate=0.01, n_epochs=1
000)

    # Print out the final training & validation Loss
    print("Training Loss:", loss['train'][-1])
    type = 'train'
    if validation:
        print("Validation Loss:", loss['valid'][-1])
        type = 'valid'

    # Printing out final cluster distribution
    for cluster in range(K):
        print("Cluster {}:\n\t# Points: {}\n\tPercent of Points: {}".format(
            cluster,
            np.sum(cluster_assignments[type]==cluster),
            np.mean(cluster_assignments[type]==cluster))
        )
    print("MU:\n", MU)
    print("sigma:\n", sigma[0])
    print("weights:\n", w[0])

    # Plotting loss curve
    plt.plot(loss[type], label='training data' if not validation else 'validation data'
)
    plt.legend()
    plt.title('Loss Curve')
    plt.ylabel('Loss')
    plt.xlabel('Iteration')
    plt.grid()
    plt.show()

    # Plotting cluster assignment
    colors = ['gold','palegreen','skyblue','violet', 'sandybrown']
    plt.scatter(
        data[:,0] if not validation else val_data[:,0],
        data[:,1] if not validation else val_data[:,1],
        s=0.5,
        c=cluster_assignments[type],
        cmap=matplotlib.colors.ListedColormap(colors)
    )
    plt.title('Clusters Visualization')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.grid()
    plt.show()

if __name__ == '__main__':
    main()
```

```
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/
python/util/dispatch.py:201: calling reduce_max_v1 (from tensorflow.pytho
n.ops.math_ops) with keep_dims is deprecated and will be removed in a futu
re version.
Instructions for updating:
keep_dims is deprecated, use keepdims instead
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/
python/util/dispatch.py:201: calling reduce_sum_v1 (from tensorflow.pytho
n.ops.math_ops) with keep_dims is deprecated and will be removed in a futu
re version.
Instructions for updating:
keep_dims is deprecated, use keepdims instead
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/
python/util/dispatch.py:201: calling reduce_logsumexp_v1 (from tensorflow.
python.ops.math_ops) with keep_dims is deprecated and will be removed in a
future version.
Instructions for updating:
keep_dims is deprecated, use keepdims instead
Training Loss: [11409.884]
Validation Loss: [5724.0635]
Cluster 0:
        # Points: 425
        Percent of Points: 0.1275127512751275
Cluster 1:
        # Points: 430
        Percent of Points: 0.12901290129012902
Cluster 2:
        # Points: 1117
        Percent of Points: 0.3351335133513351
Cluster 3:
        # Points: 1141
        Percent of Points: 0.34233423342334235
Cluster 4:
        # Points: 220
        Percent of Points: 0.066006600660066
MU:
 [[-0.01967611 -1.0733134 ]
 [-0.24217467 -1.86899   ]
 [ 1.3033597   0.3078387 ]
 [-1.1028985  -3.302389  ]
 [ 0.8397459  -1.7412274 ]]
sigma:
 [0.91591066 0.91723967 0.19811487 0.19851376 0.8651596 ]
weights:
 [0.12877943 0.12992884 0.3371587  0.32991755 0.07421546]
```



Loss Curve
validation data
14000
12000
10000
8000
6000
Loss
0
200
400
600
800
1000
Iteration



Clusters Visualization
2
1
0
−1
−2
−3
−4
Y
−3
−2
−1
0
1
2
3
4
X