SYDE 572 A3:

Kritik Kaushal, 20854655, k3kausha@uwaterloo.ca

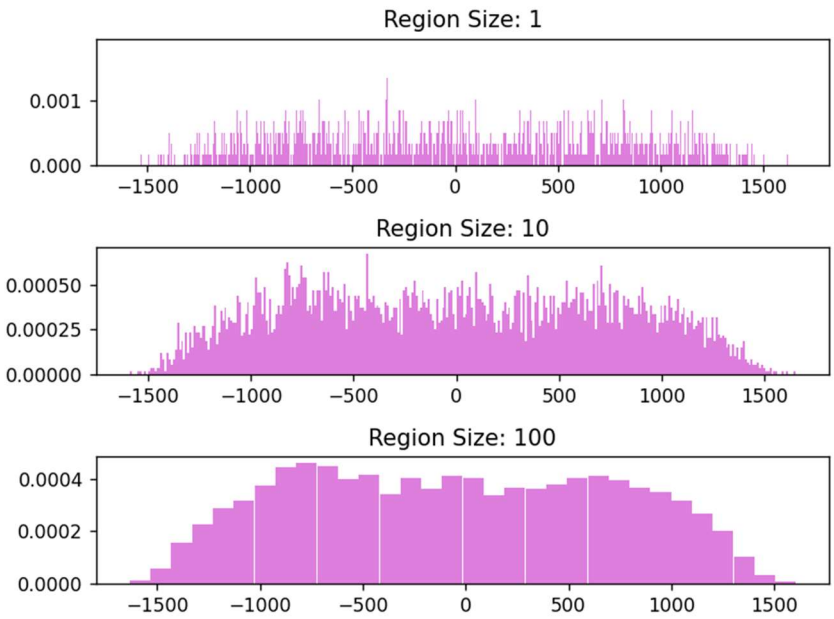
The source code for each question can be found in the zip file of the submission.

Exercise 1:

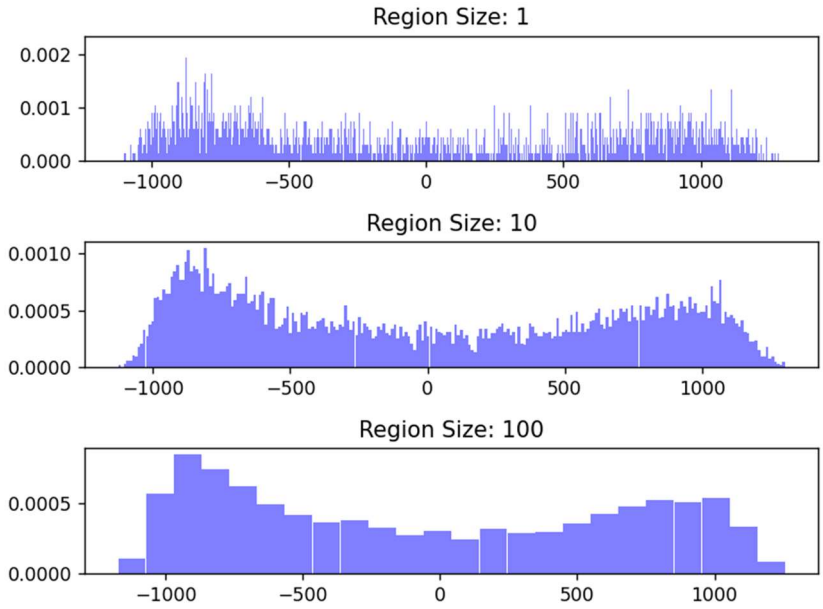
1. First, the training data images from the MNIST dataset were imported and flattened into 784x1 vectors. Then, using PCA, the data was made 1-dimensional. Then, using the following code, the probability distribution estimates were plotted.

```
# use PCA to convert 784x1 vectors into 1x1 vectors (announcement made on LEARN about this)
pca = PCA(n_components=1)
train_images_pca = pca.fit_transform(train_images_flattened).astype(np.float32)

# store all region sizes
region_sizes = [1, 10, 100]

# find probability distribution for each region size
for region_size in region_sizes:
  # number of bins will be at least 1
  number_of_bins = max(int((max(train_images_pca) - min(train_images_pca)) / region_size), 1)
  histogram, bin_edges = np.histogram(train_images_pca, bins=number_of_bins, density=True)

  # plot the probability distribution for each region size
  plt.subplot(len(region_sizes), 1, region_sizes.index(region_size) + 1)
  plt.bar(bin_edges[:-1], histogram, width=region_size, align='center', alpha=0.5, color='m')
  plt.title(f'Region Size: {region_size}')

plt.tight_layout()
plt.show()
```

   The probability distribution for class 0 (images with label 0) is shown below.



The probability distribution for class 1 (images with label 1) is shown below.



From the figures above, it appears that a region size of 10 gives the best estimation of the probability distribution. The region size of 1 gives more detailed results, but it is also noisier. The largest region size of 100 appears smoother but is less detailed.
The region size of 10 is right in the middle with a smooth estimation while preserving detail.

The probability distributions I will use for each class and region size are summarized in the table below. Note that for the region size of 1, there was too much noise to conclusively estimate a probability distribution so I will estimate uniform for both.

Class 0		Class 1	
Region Size	Probability Distribution Estimate	Region Size	Probability Distribution Estimate
1	Uniform	1	Uniform
10	U-quadratic	10	U-quadratic
100	U-quadratic	100	U-quadratic

2. Using the probability distribution estimates from the histograms of various sizes, the labels of the test data were predicted. ML classifiers were implemented using a uniform probability distribution and a u-quadratic probability distribution.

   For the uniform distribution, the class boundaries were chosen based on the histogram graphs produced above. Specifically, the class boundaries are:

```
# defining the boundaries for uniform distribution based ML classifier
class_0_boundaries = [-1500, -1000, 1000, 1500]
class_1_boundaries = [-999, 999]
```

   From there, the following accuracy scores resulted:

```
Class 0 Accuracy assuming quadratic distribution:  0.4227587371264562

Class 1 Accuracy assuming quadratic distribution:  0.5459804212399881

Class 0 Accuracy assuming uniform distribution:  0.21124851770286296

Class 1 Accuracy assuming uniform distribution:  0.8838901262063845
PS C:\Users\kriti\Desktop\4A\syde 572\a3>
```

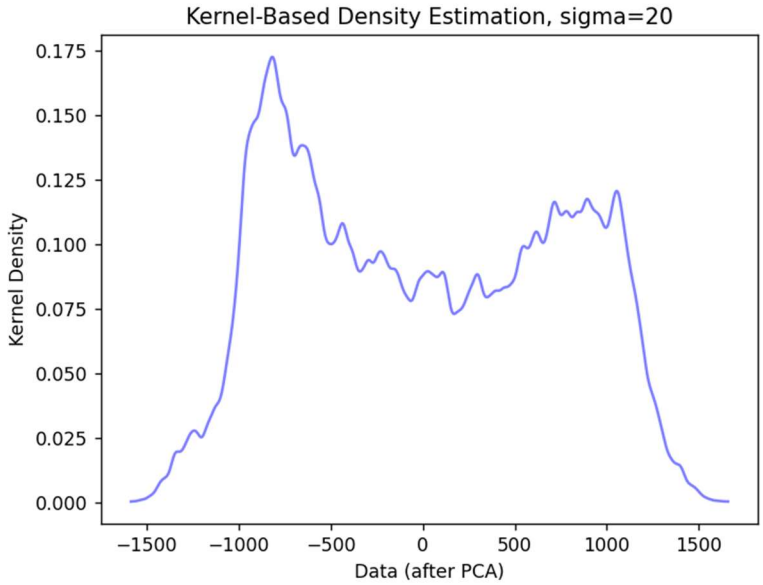
   In summary:

   Accuracy using quadratic probability distribution (estimated by region sizes of 10 and 100): **49%**

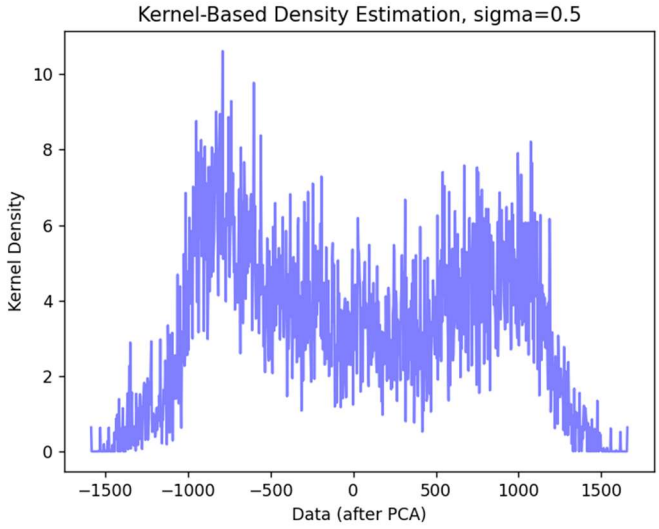
   Accuracy using uniform probability distribution (estimated by region size of 1): **55%**

   Although the region size of 1 has better accuracy, I think the region sizes of 10 and 100 are still better because they produce more similar results for both classes. The uniform distribution ML classifier has a strong bias towards one class because the PCA gives values for both classes in a similar range (-1500 – 1500). So, the region size of 10 and 100 is better in terms of test error.

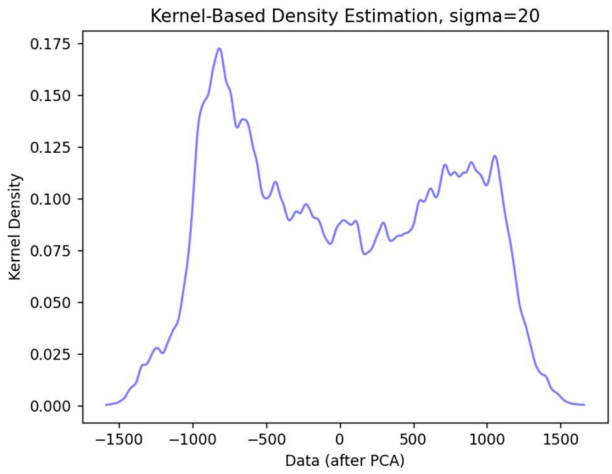
3. Using kernel-based estimation with a Gaussian kernel size of 20, the following probability distribution graph was produced:



Note that to test my implementation, I changed the size of the Gaussian kernel to 2 which produced the following graph, which shows the kernel-based estimation is behaving as expected.



Referring back to the graph with kernel size of 20, we observe that again, the probability distribution appears to be u-quadratic shaped.



From this, we can conclude that the accuracy will be the same as the histogram-based estimation using region sizes of 10 and 100, since those graphs also estimate a u-quadratic distribution.

Thus, accuracy using quadratic probability distribution (estimated by kernel-based estimation using kernel size of 20): **49%**

4. In this case, the kernel-based estimation and histogram-based estimation with region size 10 and 100 produced the same results. However, broadly speaking kernel-based estimation provides a more accurate representation of the distribution for smooth and continuous data. This is because kernel-based estimation is immune to the flaw of histogram-based estimation which assumes that the probability is constant in each region.

5. Parametric estimation assumes a form of data distribution. In this case, we did not know the distribution of the data (PCA from two classes of MNIST dataset). Thus, we would have had to make some assumptions about the data distribution which could potentially result in poor accuracy.

   For density estimation, if some information about the data distribution is known before-hand and it is valid, then parametric estimation will give better results. In cases where no information about the data distribution is available, non-parametric estimation methods may be better.

Exercise 2

1. For this exercise, I first loaded the MNIST dataset and flattened the data into 784x1 vectors. I know I did this successfully because I printed the shape and length of the flattened training data, as follows:

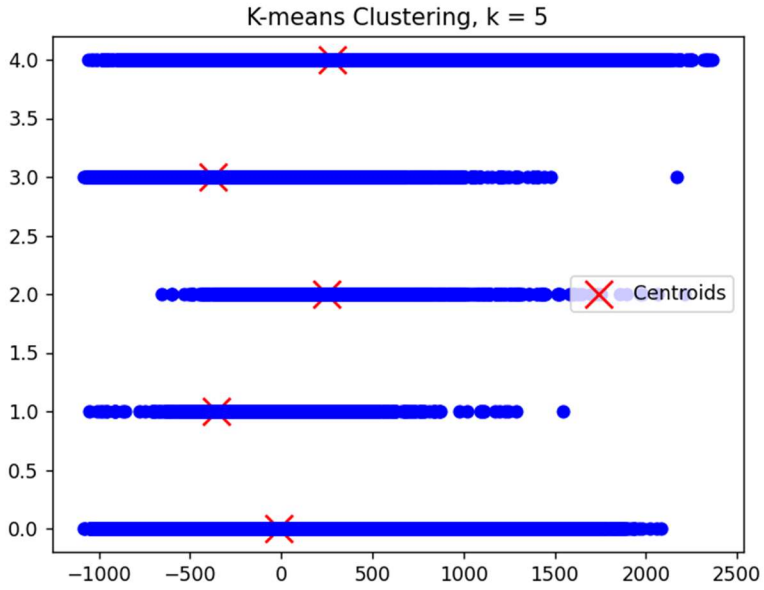
```
(784,)
60000
PS C:\Users\kriti\Desktop\4A\syde 572\a3>
```

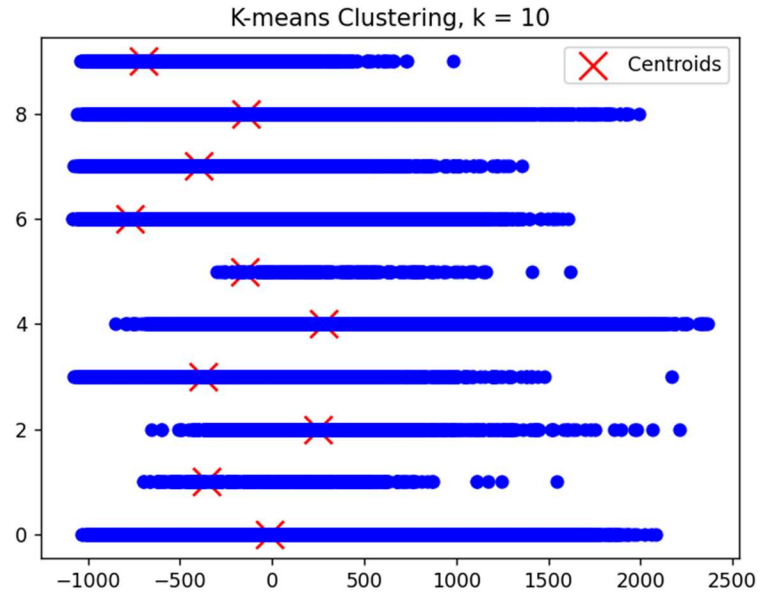
The k-means clustering algorithm was implemented as follows. I used Euclidean distance as the distance metric, and specifically I used the np.linalg.norm function for this.

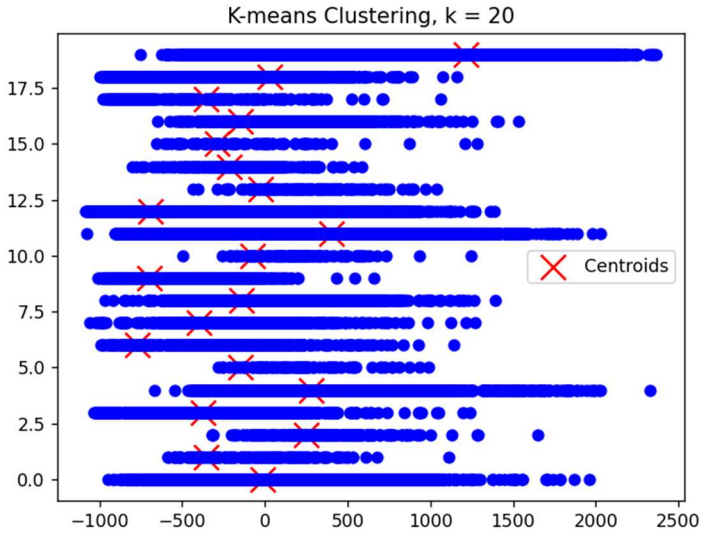
```python
# pick centroids randomly since we are not told explicitly where to place them
np.random.seed(1)
centroids = training_data_flattened[np.random.choice(training_data_flattened.shape[0], K[x], replace=False)]

# convergence tolerance
tolerance = 1e-4

# k-means clustering, runs only for the predefined number of iterations
for j in range(max_iterations):
  # initialize array for distances
  distances = np.zeros((training_data_flattened.shape[0], K[x]))
  for i in range(K[x]):
    # update the distances from data points to cluster centroids
    distances[:, i] = np.linalg.norm(training_data_flattened - centroids[i], axis=1)

  # update labels to closest centroid
  labels = np.argmin(distances, axis=1)
  # update centroids
  new_centroids = np.array([training_data_flattened[labels == k].mean(axis=0) for k in range(K[x])])
  # used to check for convergence
  delta_centroids = new_centroids - centroids

  # check if centroids are changing (comparing with a tolerance value that is very small)
  if delta_centroids.all() < tolerance:
    break

  # update centroids
  centroids = new_centroids
```

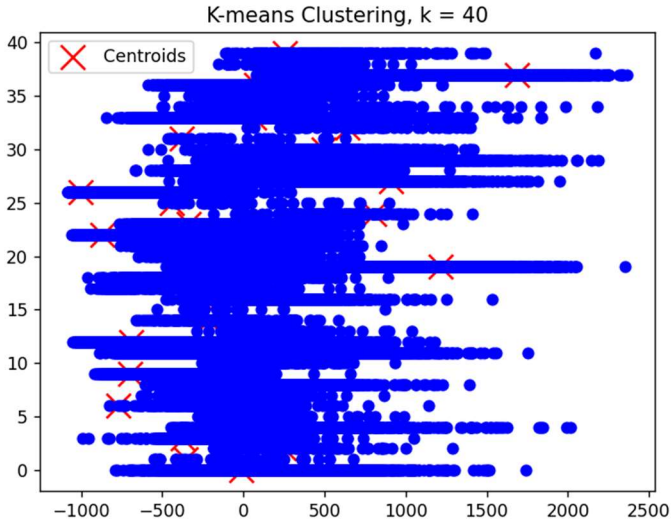
2. Then, the above k-means clustering algorithm was applied to the MNIST dataset. The ground truth class labels were not used during the clustering process. For visualization purposes, and for purposes of this writeup, I used a 1-dimensional PCA to visualize the clusters and the centroids.

   The results are shown in the figures below, for each value of k. The PCA value is on the x-axis and the cluster number is on the y-axis.


K-means Clustering, k = 5
Centroids
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−1000
−500
0
500
1000
1500
2000
2500


K-means Clustering, k = 10
Centroids
8
6
4
2
0
−1000
−500
0
500
1000
1500
2000
2500





3. The overall clustering consistency for all 4 values of k is reported below.

```
Overall Cluster Consistency, k = 5:  35.5%
Overall Cluster Consistency, k = 10:  37.3%
Overall Cluster Consistency, k = 20:  51.3%
Overall Cluster Consistency, k = 40:  61.6%
PS C:\Users\kriti\Desktop\4A\syde 572\a3>
```

4. Based on the above, a k-value of 40 produces the best result. However, this is not intuitive and does not make sense. Intuitively, a k-value of 10 should produce the best results on the MNIST dataset since we know there are 10 distinct classes.

Results from cluster consistency can be misleading. This is because of a lack of ground truth. Cluster consistency does not have any sense of correctness, only the agreement between clustering and class labels. Also, in the case of k = 20 or k = 40, the data becomes over-segmented. This results in non-meaningful patterns. In our case, k = 10 should produce the best results, and results in well-segmented data.

Exercise 3

1. First, we start with the derivation of the EM algorithm. We modify the EM algorithm presented in class which assumes identity covariance matrices.

   Initialization:

   Initialize the means, $\widehat{\mu_k}$, mixing coefficients, $\widehat{\pi_k}$, and diagonal covariance matrices, $\widehat{\Sigma_k}$. Note that in the case of identity covariance matrices, we did not need to initialize them since the determinant of an identity matrix is simply 1.

   E-step:

   For each data point in the set, calculate r_k^i, the posterior probabilities. We use the diagonal covariance matrices in the Gaussian PDF. The posterior probabilities are calculated as follows:

$$r_k^i = \frac{\widehat{\pi_k}\ \mathrm{N}(x_i|\ \widehat{\mu_k}, \widehat{\Sigma_k})}{\sum_{j=1}^{K} \widehat{\pi_j}\ \mathrm{N}(x_i|\ \widehat{\mu_j}, \widehat{\Sigma_j})}$$

   M-step:

   Re-estimate the parameters and update the diagonal covariance matrices as follows:

$$\widehat{\mu_k} = \frac{\sum_{i=1}^{N} r_k^i * x_i}{\sum_{i=1}^{N} r_k^i}$$

$$\widehat{\pi_k} = \frac{1}{N}\sum_{i=1}^{N} r_k^i$$

$$\widehat{\Sigma_k} = \mathrm{diag}\left(\frac{\sum_{i=1}^{N} r_k^i * (x_i - \widehat{\mu_k})^2}{\sum_{i=1}^{N} r_k^i}\right) = \mathrm{diag}\left(\frac{\sum_{i=1}^{N} r_k^i * {x_i}^2}{\sum_{i=1}^{N} r_k^i} - \widehat{\mu_k}^2\right)$$

   Then, we check the negative-log likelihood for convergence. The negative-log likelihood should be decreasing from step to step, and we know we've converged when the negative-log likelihood is smaller than some predefined threshold. In the case of diagonal covariance matrices, the negative-log likelihood looks like the following:

$$l = -\sum_{i=1}^{n} \log \left[\sum_{k=1}^{K} \widehat{\pi_k} (2\pi)^{-\frac{d}{2}} \left|\widehat{\Sigma_k}\right|^{-\frac{1}{2}} \exp \left[-\frac{1}{2} (x_i - \widehat{\mu_k})^T \widehat{\Sigma_k}^{-1} (x_i - \widehat{\mu_k}) \right] \right]$$

Using the information presented in the question, we can simplify the above as follows:

$$l = -\sum_{i=1}^{n} \log \left[\sum_{k=1}^{K} \widehat{\pi_k} (2\pi)^{-\frac{d}{2}} \prod_{k=1}^{K} \widehat{\Sigma_k}^{-\frac{1}{2}} \exp \left[-\frac{1}{2} (x_i - \widehat{\mu_k})^T \widehat{\Sigma_k}^{-1} (x_i - \widehat{\mu_k}) \right] \right]$$

Then, based on the derivation above, and the pseudocode presented in the assignment, the EM algorithm for dGMM was implemented as follows:

```python
# Exercise 3

# set threshold for the negative-log likelihood value
tol = 1e-5
neg_log_likelihood_vals = []

# helper function to calculate mv gaussian
def multivariate_gaussian(x, mean, covariance):
    d = len(x)
    inv_covariance = 1 / covariance
    diff = x - mean
    exponent = -0.5 * np.sum(diff * (diff * inv_covariance))
    # using properties of diagonal covariance matrices to make computation a bit easier
    likelihood = (1 / np.sqrt((2 * np.pi) ** d * np.prod(covariance))) * np.exp(exponent)
    return likelihood
```

```python
# EM algorithm for diagonal Gaussian Mixture Model

def EM_dGMM(X, K, max_iter=500):
    n, d = X.shape
    # random initialization of means, diagonal covariance matrices, and cluster coefficients
    means = X[np.random.choice(n, K, replace=False)]
    covariances = [np.diag(np.random.rand(d)) for _ in range(K)]
    cluster_coefficients = np.random.rand(K)

    likelihood = np.zeros((n, K))

    for iter in range(max_iter):
        # expectation step
        for k in range(K):
            # calculate responsibilities
            for i in range(n):
                likelihood[i, k] = multivariate_gaussian(X[i], means[k], covariances[k])
        responsibilities = cluster_coefficients * likelihood

        # maximization step
        N_k = responsibilities.sum(axis=0)
        # re-estimate the parameters and update the diagonal covariance matrices
        cluster_coefficients = N_k / n
        means = np.dot(responsibilities.T, X) / N_k[:, np.newaxis]
        covariances = [np.dot(responsibilities[:, k] * (X - means[k]).T, (X - means[k])) / N_k[k] for k in range(K)]
```

```
    # compute negative-log likelihood, I used log1p to prevent divide-by-zero errors
    neg_log_likelihood = -1*np.log1p(np.dot(likelihood, cluster_coefficients)).sum()
    print(f"Iteration {iter + 1}: Negative-Log Likelihood = {neg_log_likelihood}")
    neg_log_likelihood_vals.append(neg_log_likelihood)

    # check negative-log likelihood for convergence
    if (iter > 1) & (abs((neg_log_likelihood_vals[iter])-(neg_log_likelihood_vals[iter-1]))<= tol*abs(neg_log_likelihood_vals[iter])):
      break

  return cluster_coefficients, means, covariances

# import the training and test data sets seperately
mnist_training_data = MNIST(root='./data', train=True, download=True, transform=None)
mnist_test_data = MNIST(root='./data', train=False, download=True, transform=None)
```

```
# store training data and labels
training_data = mnist_training_data.data.numpy()
training_labels = mnist_training_data.targets.numpy()

# reshape training data into 784x1 vector
training_data_flattened = training_data.reshape(-1, 28*28)

K = 5

# applying PCA on flattened data because I was getting overflow errors
pca = PCA(n_components=1)
training_data_pca = pca.fit_transform(training_data_flattened).astype(np.float32)

# fit the model
cluster_coefficients, means, covariances = EM_dGMM(training_data_pca, K)
```

2. The adapted EM algorithm was applied to the MNIST dataset. I applied PCA to reduce dimensions because it took a long time to run the script. Due to time constraint, I only fit the GMM for K = 5.
   I printed out the values of the negative-log likelihood to test if my code was working, and the results are interesting. My values of negative-log likelihood are decreasing from step to step, which is a good sign. Strangely though, after about 53 iterations, I started getting “NaN” as the negative-log likelihood. This is likely due to limitations with floating point arithmetic in Python.

```
Iteration 1: Negative-Log Likelihood = -0.08360522270862167
Iteration 2: Negative-Log Likelihood = -6.257623800074809e-05
Iteration 3: Negative-Log Likelihood = -4.803176146595921e-08
Iteration 4: Negative-Log Likelihood = -3.75842289034481e-11
Iteration 5: Negative-Log Likelihood = -2.9867394462583704e-14
Iteration 6: Negative-Log Likelihood = -2.403034505322554e-17
Iteration 7: Negative-Log Likelihood = -1.9515929140440017e-20
Iteration 8: Negative-Log Likelihood = -1.5961110016054267e-23
```

```
Iteration 54: Negative-Log Likelihood = nan
Iteration 55: Negative-Log Likelihood = nan
Iteration 56: Negative-Log Likelihood = nan
Iteration 57: Negative-Log Likelihood = nan
Iteration 58: Negative-Log Likelihood = nan
```

Still, it was a good sign that my negative-log likelihood was decreasing from step to step. As such, I set the maximum number of iterations to 50 only, to prevent the not-a-number error. The runtime was in the order of minutes, which was also another good indicator.

As mentioned previously, time was a concern, so I only used K = 5. I implemented the Bayes classifier as follows and ran it on every image in the test dataset.

```
# bayes classifier for test image label prediction
def bayes_classifier(X):
  predicted_class = []
  for cluster in range(0, 9, 1):
    # calculating using the density of the cth cluster in the training set
    predicted_class.append((np.bincount(training_labels)[cluster]/np.bincount(training_labels).sum()) * X)
  # return the most likely class
  return np.argmax(predicted_class)

# applying the bayes classifier to each image in the test set, storing results in array
results = []
for image in test_data_pca:
  results.append(bayes_classifier(image))

# calculate and print the error
print(f"\nTest error, K = 5: {(1-(sum(results)/sum(test_labels))): .2%}" )
```

Then, this gave me the following result for test error:

```
Test error, K = 5:  28.56%
PS C:\Users\kriti\Desktop\4A\syde 572\a3>
```

I used PCA with n = 1 to reduce the dimension of the problem and increase accuracy slightly. Overall, I think this value of test error is a bit higher than expected. All code can be found in the zip file of this submission.