Due date: Nov 29th. Please submit an .ipynb file.

Instructions:

1) The .ipynb file shall include not only the **source code**, but also necessary **plots/figures** and **discussions** which include your *observations*, *thoughts* and *insights*.
2) Please avoid using a single big block of code for everything then plotting all figures altogether. Instead, use a small bock of code for each sub-task which is followed by its plots and discussions. This will make your homework more readable.
3) Please follow common software engineering practices, e.g., by including sufficient **comments** to functions, important statements, etc.

Programming Question (60 points):

In week 10, we learned K-means clustering which is a popular machine learning and data mining algorithm that discovers potential clusters within a dataset. Finding these clusters in a dataset can often reveal interesting and meaningful structures underlying the distribution of data.

In this programming problem, you will get familiar with how to implement K-means algorithms and use K-means to cluster a 2D dataset. Then you will further try K-means on the MNIST dataset which we met in HW4. (For the introduction of MNIST dataset, please refer to HW4).

To help you get started, we summarize the K-means algorithm in pseudo code as follows.

```
data = Load Data
centroids = Randomly Create K Centroids
while centroids not converged:
for each data point:
assign data point to the closest centroid
for each cluster:
set new centroid location to be the mean of
all points in this cluster
```

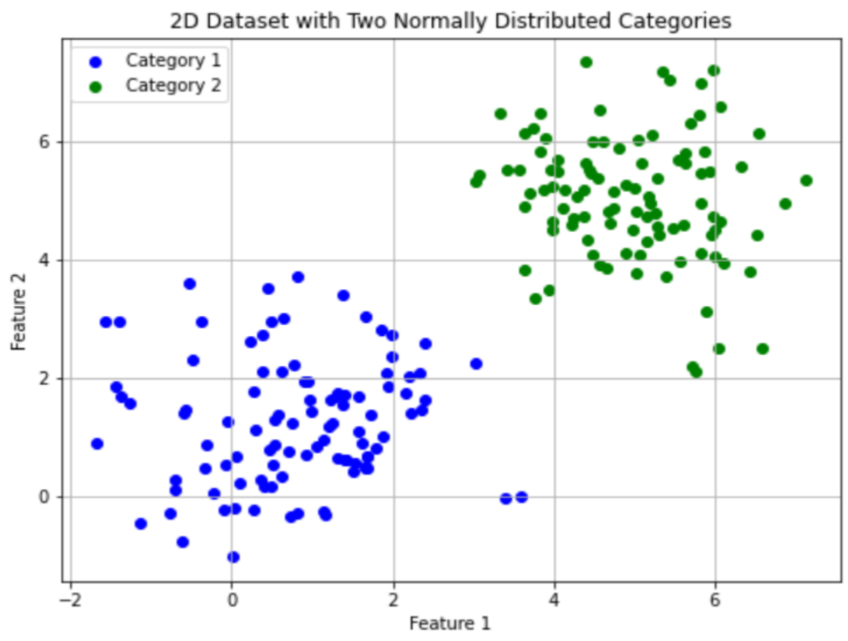
In this homework, you will follow the step-by-step instructions to accomplish the tasks.

Step 1 2D Data Generalization and Visualization (10 pts):

In this step, you need to:

a) Generate a 2D dataset that includes two separate categories of data, each of which contains 100 data points that normally distributed. The two categories have a mean of (1, 1) and (5, 5), respectively. Both categories have a standard deviation of 1.
   Tips 1: To generate the data, you can explore the function **np.random.normal** which draws random samples from a normal distribution.
   Tips 2: After you generate two categories of data, please remember to combine the data into one dataset. You have multiple methods to do so. But remember to combine the data in a **row-wise** manner.

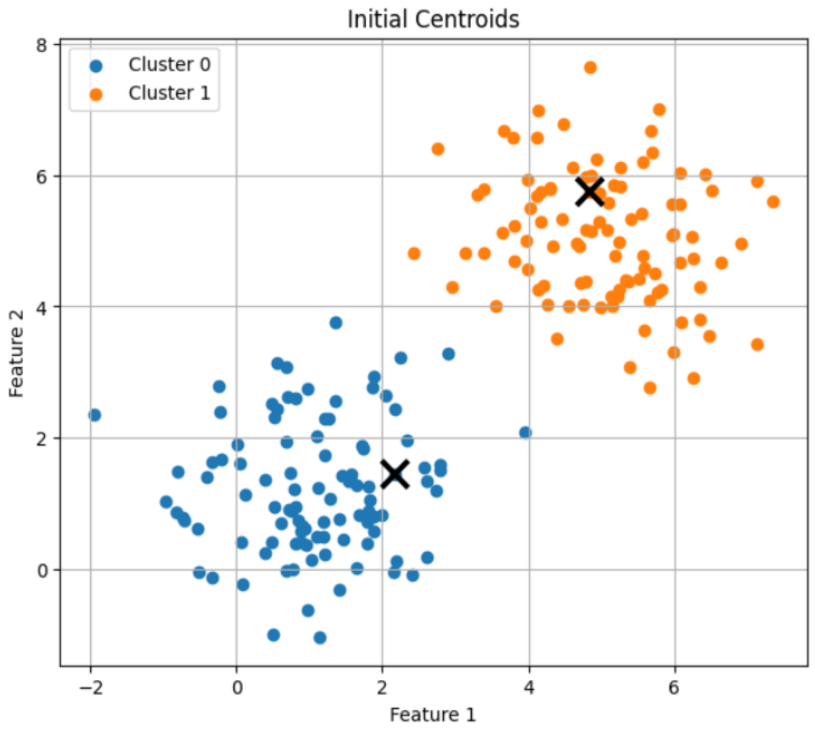
b) Visualize the data using scatter plot. You will see something like this:



Step 2 Helper Function and Initialization (10 pts)

In this step, you need to:

a) Define a helper function, i.e., the Euclidean distance function.
b) Implement a **initialize_centroids** function that randomly select K (In this example, K=2) centroids as the initial centroids. Plot the results and use marker in pyplot to highlight the centroids. You should have something like this:



Step 3 K-means Implementation (25 pts)

As shown in the pseudo code, each iteration of K-means algorithm includes two actions: 1) Assign each point to the nearest centroids 2) Update the centroid. To implement K-means, we define two functions **assign_cluster** and **update_centroids** corresponding to these two actions.

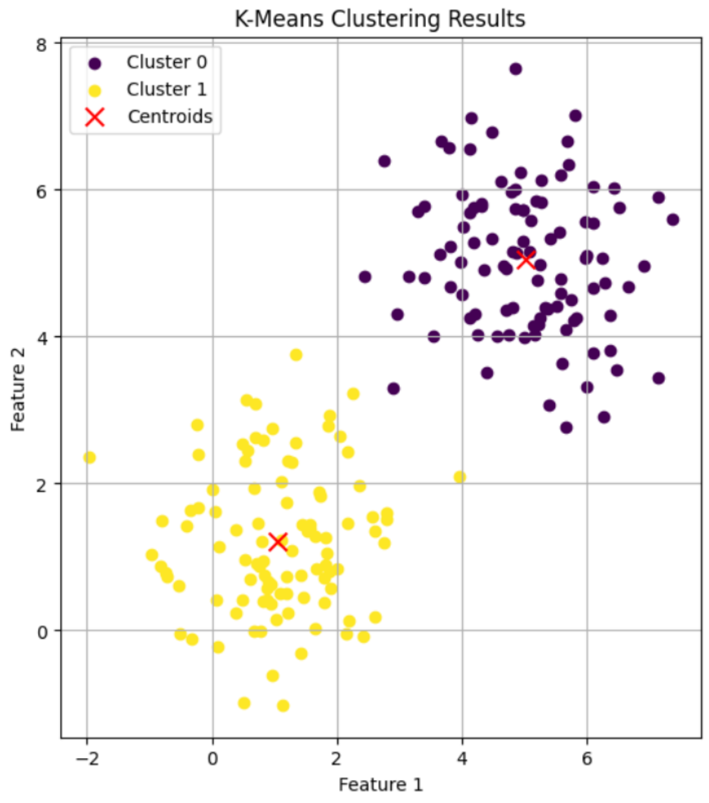
In this step, you need to:

a) Implement **assign_cluster** function. In this function, you need to calculate the distance between the assigned point and the centroids. Then you need to assign the point to the nearest centroid.

b) Implement **update_centroids** function. You need to calculate the mean of all the points assigned to each centroid to determine the new centroids.

c) Implement **K-means** function. The algorithm converges when all centroids remain unchanged, i.e., all the new centroids = old centroids. You need to use this as the stop criteria.

d) Apply the K-means algorithm on the generated data and visualize the result. You will get something like this:



Now, you have finished implementing K-means algorithm to achieve clustering with 2D data with 2 centroids. Next, you will use your code to do clustering on 784-dimensional data. The dataset you will use is the MNIST dataset which we used in HW4. You have read the data and flattened the data in HW4. You can check by printing out the dimensions of

train_images.shape and see if it is **(60000,784)**.

We will start from here.

Step 4 [Optional]Train K-means model on MNIST dataset (10 pts)

In this step, you need to:

a) Train a K-means model using the training images of MNIST dataset using K = 10. For this step, you can record the time needed for training.
Tips: You can use the functions in packages "datetime" to compute the time difference.

Step 5 Mini-Batch K-means (20 pts)

You will find that the training takes a long time. This is because the classic implementation of the clustering method consumes the whole set of input data at each iteration.

Thus, typically with the increasing size of the dataset, we are more likely to use Mini-Batch K-means. The idea is to use small random batches of data of a fixed size. Each iteration a new mini batch from the dataset is obtained and used to update the clusters until convergence.

In this homework, we do not require you to implement mini-batch K-means. Instead, you can use the SKLearn package. You can start with:

```
import sklearn
from sklearn.cluster import MiniBatchKMeans
```

You can learn more about Mini-Batch K-means here: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.MiniBatchKMeans.html

In this step, you need to:

a) Use the train_images to train a mini-batch K-means model. You need to train the model under different cluster numbers: K= [10, 16, 64, 256].

b) To evaluate the model, you need to compare the **inertia** value of each model to decide which K is better.

   Tips: You can use the metrics provided by sklearn to do so. Check the attributes of **sklearn.cluster.MiniBatchKMeans**.

c) Apply the best model with the test dataset and print out the inertia velue.

[Additional Material]

You might want to use a more straightforward measurement to evaluate the performance of your model. You can also use the 'accuracy' of evaluate K-means.

So we give two practical tools in the unsupervised-learning evaluation process and they are typically used together. First, you can use **assign_labels_to_clusters** to assign labels to each cluster based on the most common true label in that cluster. Then, you can use **manual_accuracy_score** to calculate how accurately these cluster-based labels match the true labels (you can also use sklearn.metrics.accuracy_score function to calculate the accuracy.)

```python
def assign_labels_to_clusters(clusters, true_labels, k):
    labels = np.zeros_like(clusters)
    for i in range(k):
        mask = (clusters == i)
        # Assign the most common label to the centroid
        labels[mask] = np.bincount(true_labels[mask]).argmax()
    return labels

def manual_accuracy_score(true_labels, predicted_labels):
    correct_predictions = sum(p == t for p, t in zip(predicted_labels,
true_labels))
    accuracy = correct_predictions / len(true_labels)
    return accuracy
```