Assignment 2: Clustering Digits

Toolkit

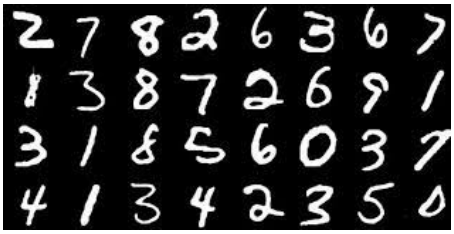
What you'll need for this assignment:

- Python3
- NumPy
- SciPy
- SciKit-Learn
- Matplotlib

If you want to use the `show_image_mnist()` function in `utils.py`, you'll need to install Pillow with: `pip install Pillow`

Data

We're using a subset of the MNIST dataset (see below) consisting of handwritten digits from 0 to 9. Each image is 784 (28x28) pixels, in black-and-white (this means we have only 1 channel). The data have been for you into numpy float arrays of 784 length, with each value ranging from 0 (black) to 1 (white).



You are provided the following files in the data folder:

- `data.npy`
- `labels.npy`

As most clustering methods can't predict for new data (k-Means however being a notable exception), we typically **won't do a train/validation/test split**.

Code

The code is split into 3 files:

1. `main.py` - This is where you'll do most of the coding, apart from the k-Means implementation. This is also the only file you should be running, i.e. with `python main.py`
2. `utils.py` - This has some utility functions which will help you with various parts of the assignment. You shouldn't need to modify anything here.
3. `kmeans.py` - This is where you'll implement k-Means.

Tasks

Before you start the tasks, you might want to get a better sense of the data you're working with. Currently, `main.py` is set up to load in the data, in a 2D array of shape [1000, 784], where 1000 is the number of images, and 784 is the number of pixels per image. The labels are also loaded, in an array of shape [1000], with values from 0 to 9 corresponding to the digit in the image. You can view an image with the function `show_image_mnist` from `utils.py`, which converts the provided array of [784] back to a 28x28 pixel image.

1. k-Means

(This will likely be the hardest task.)

Implement k-Means clustering in `kmeans.py`. Your implementation needs to contain these functions:

1. Initialize the clusters in `initialize_clusters`
    - Use the Forgy method (seen in class – W3): Set the centroids equal to random points in your data (making sure no 2 centroids are given the same point)
2. Assign the data to clusters in `assign`
    - Use the `euclidean_distance` function to calculate the distance of every point in your data to each centroid
    - Assign each point to the centroid whose distance is lowest
3. Update the cluster centroids in `update`
    - The value for each centroid is the mean of all of the points assigned to it
4. Put it all together in `fit_predict`
    - You first need to call `initialize_clusters`

- Next, you iteratively call `assign` and `update`
- Stop after a max of 100 iterations
- Return the assignments for each datapoint (i.e. the predictions)

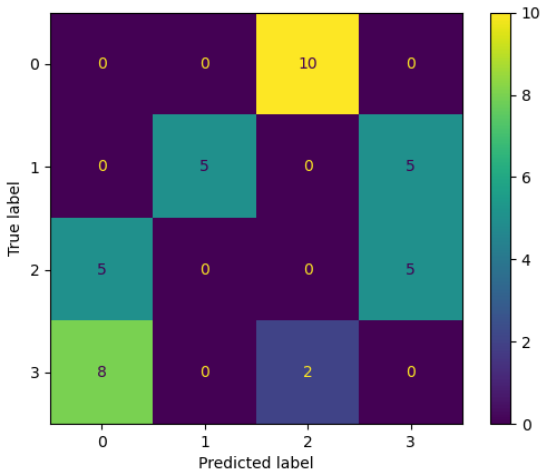
The `__init__` function is given to you. It stores the number of clusters, number of features, and the centroids. When you've completed your `KMeansClusterer` class, all you should need to do is instantiate the class, and then call `fit_predict`.

To make sure your k-Means is working, you can use the function (which is already imported): `adjusted_rand_score(labels, predictions)`
This will give you the Adjusted Rand Index (ARI). You should expect scores within the range of 0.25 to 0.40.

In the report: Over 5 runs of your k-Means algorithm, what is the average ARI?

2. Confusion Matrix

Confusion matrices can be used for clustering as well, however they are a bit more "confusing" to parse because the clusters' indices won't necessarily match the label's indices. Let's look at an example:



Here we have 4 classes, each with 10 samples per class. Each row corresponds to all the samples in a true class. The columns correspond to which cluster is predicted by the clustering algorithm. The columns can be swapped without affecting the results, since each cluster is not assigned to a specific class.
In this example, Class 0 appears well-clustered since all examples are in a single cluster, and there are only 2 examples from another class in that cluster. For Class 1, the examples are split into 2 clusters, one of them containing half of the examples from Class 2.

Use `plot_confusion_matrix(labels, predictions)` from `utils.py` to analyze your clusters for one run of k-Means.

In the report: Show the confusion matrices for k-Means (for 1 run). Which digits do seem to cluster well? Are there digits that the models easily confuse? Are there classes where one clearly performs differently?

Submission

- Before the submission of the code, make sure that when we call "`python main.py`" it runs k-Means once, prints the ARI, and nothing else (no plotting / showing images). Don't remove any code to achieve this, just comment it out. Make sure to clean up the code so that it is clear what we would need to uncomment to run certain things.
    - As a reminder: if you changed the loading filepath to something besides "data/data.npy" and "data/labels.npy", be sure to change it back.
- Zip your files into a .zip, and name the zip file s123456_clustering.zip, where s123456 corresponds to your student number.
- Within the zipped folder, include all of the files given, plus a report called s123456_clustering.pdf, again with s123456 corresponding to your student number

Grading criteria (max 12 points)

Task 1: **8 points**

2 points for correct `initialize_clusters`
2 points for correct `assign`
2 points for correct `update`
2 points for correct `fit_predict`

Task 4: **2 points**

2 points for correct analysis of a confusion matrix

Report: **2 points**

explanations, structure, and language