Assignment 4. K-Means Clustering

Gary Geunbae Lee
CSED342-01 Artificial Intelligence

Contact: TA Seonjeong Hwang (seonjeongh@postech.ac.kr)

General Instructions

This (and every) assignment has a written part and a programming part.

✎This icon means a written answer is expected. Some of these problems are multiple choice questions that impose negative scores if the answers are incorrect. So, don't write answers unless you are confident.

⌨This icon means you should write code. you can add other helper functions outside the answer block if you want.

Introduction

In the assignment 4, we will implement a commonly-used unsupervised learning algorithm – K-means clustering. K-means 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. K-means clustering has been applied to many problems in science and still remains popular today for its simplicity and effectiveness.

Files and Folders

Replace the **studentid** with your student ID.

- **kmeans.py** : Your code for K-means clustering.
- **soft_kmeans.py** : Your code for Soft K-means clustering.
- **kmeans_MNIST.py** : The file for trying K-means algorithm on MNIST.
- **kmeans_tests.py** : Testing code for kmeans.py.
- **soft_kmeans_tests.py** : Testing code for soft_kmeans.py.
- **analysis_tests.py** : Testing code for analysis.py.
- **utils.py** : Helper code.
- **results/(2D, 2D_soft, MNIST)** : Where your result figures will be saved.

In this assignment, you have to submit a report and two .py files:

`name_studentid.pdf`, `kmeans.py`, `soft_kmeans.py`

There is no format for the report, but **the attached images must be clearly visible and the report must be a pdf file.** Penalties will be imposed if these conditions are not met. **Please write your answer in Word, then convert it to a PDF file.**

Warmup: Data, Distance, and Centroids

Data

We will assume we have a dataset of m data points where each data point is n-dimensional. For example, we could have 100 points on a two dimensional plane, then m=100 and n=2.

Distance

Given two n-dimensional points a and b where:

$$a = [a_1, a_2, ..., a_n], b = [b_1, b_2, ..., b_n]$$

We define the **distance** between points a and b as:

$$D(a, b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + ... + (a_n - b_n)^2}$$

This is commonly known as the Euclidean distance. For example, the Euclidean distance between the three-dimensional points x_1 and x_2 (displayed as Python lists) is:

```
x1 = [-2, -1, -4], x2 = [10, -5, 5]
```

$$D(x_1, x_2) = \sqrt{(-2 - 10)^2 + (-1 + 5)^2 + (-4 - 5)^2)} = \sqrt{241}$$

Here is another example of two four-dimensional points:

```
x1 = [0, 0, 1, 1], x2 = [1, 1, 0, 0]
```

$$D(x_1, x_2) = \sqrt{(0 - 1)^2 + (0 - 1)^2 + (1 - 0)^2 + (1 - 0)^2} = \sqrt{4} = 2$$

Centroids

A centroid can be seen as the "center of mass" of a group of points. **In K-means, we assume that the centroid of a cluster is the mean of all the points in that cluster.** The mean data point consists of **the mean of the components on each dimension.** Say, for three n-dimensional points a, b, and c, we have the mean as:

$$Mean = \left[\frac{(a_1 + b_1 + c_a)}{3}, \frac{(a_2 + b_2 + c_2)}{3}, ..., \frac{(a_n + b_n + c_n)}{3}\right]$$

For example, if a cluster contains three 2D points (displayed as Python lists) as below:

```
x1 = [1, 2], x2 = [3, 4], x3 = [5, 6]
```

The centroid for this cluster is defined as:

```
C = [(1 + 3 + 5)/3, (2 + 4 + 6)/3] = [3, 4]
```

Problems

Problem 1: K-Means Clustering

Our ultimate goal is to partition the data points into K clusters. How do we know which data point belongs to which cluster? It is actually extremely hard to find the "optimal" answer to this, but K-means takes a simple, heuristic approach - we simply assign a data point to the centroid whose distance to the data point is the smallest (the "closest centroid" to this data point).

Let's assume we already have K centroids, initialized randomly. Now, if our randomly initialized centroids were already in good locations, we would have been done! However, it is most likely that the initial centroid locations do not reflect any information about the data (since they were just random). To update the centroid locations to something that makes more sense we use an iterative approach:

1. First, we assign all data points to their corresponding centroids by finding the centroid that is closest.
2. Then, we adjust the centroid location to be the average of all points in that cluster.

We iteratively repeat these two steps to achieve better and better cluster centroids and meaningful assignments of data points.

Note that the data points do not change! Only the location of the centroids change with each iteration (and thus the points that are closest to each centroid changes).

One last question you may have is, when do we stop repeating these two steps? I.e., what does it means for the algorithm to have converged? We say the algorithm has converged if **the locations of all centroids did not change much between two iterations**.

Take this example of two centroids of 2-dimensional points. If at the previous iteration, we have the two centroid locations as:

```
c1 = [0.45132, -0.99134], c2 = [-0.34135, -2.3525]
```

And at the next iteration the centroids are updated to:

```
c1 = [1.43332, -1.9334], c2 = [-1.78782, -2.3523]
```

Then we say the algorithm has **not converged**, since the location of the two centroids are very different between iterations. However, if the new locations of the centroids are instead:

```
c1 = [0.45132, -0.99134], c2 = [-0.34135, -2.3524]
```

Then we say the algorithm has **converged**, since the locations of the two centroids are very similar between iterations (Typically, "very similar" means for every dimension of two points a and b, their difference ($|a - b|$) is smaller than some constant (1e-5 is a common choice)).

You will be given the following data structures for K-means clustering:

- `data`: A list of data points, where each data point is a list of floats. Example:
  `data = [[0.34, -0.2, 1.13], [5.1, -12.6, -7.0], [-15.7, 0.06, -7.1]]`
- `centroids`: A dictionary of centroids, where the keys are strings (centroid names) and the values are lists (centroid locations). Example:
  `centroids = {"centroid1": [1.1, 0.2, -3.1], "centroid2": [9.3, 6.1, -4.7]}`
  Here, centroids contain two key-value pairs, where key is the centroid name, and value is the location of the centroid. You should **NOT** change the names of the centroids when you later update their locations.

It is up to you to decide what other data structures to use to complete the algorithm. In the following problems, you will complete functions in `kmeans.py` needed for `main()` to implement K-means clustering. You should use the file: `kmeans_tests.py` to test your functions as you complete them. Running `kmeans.py` before you have implemented all steps is likely to result in errors.

Problem 1a: Calculating Euclidean distances [2 points] ⌨

Implement `euclidean_distance()` function in `kmeans.py`. In the code, the body of the function is empty except for a single statement: `pass`. This is a Python statement that does nothing, but can be used in situations where you need an instruction to be there but you don't want it to do anything. Please REMOVE the `pass` statement when you have finished implementing the function.

We have provided testing code for you to verify your implementation. Uncomment this line:

```
test_euclidean_distance(data1, data2)
```

at the bottom of `kmeans_tests.py` and run `python kmeans_tests.py` in a terminal window to verify your implementation. If you passed our tests, you should see:

```
test_euclidean_distance passed.
```

Problem 1b: Assigning data points to closest centroids [3 points] ⌨

Implement `assign_data(data_point, centroids)` function in `kmeans.py`. We have provided testing code for you to verify your implementation. Uncomment this line:

```
test_assign_data()
```

at the bottom of `kmeans_tests.py` and run `python kmeans_tests.py` in a terminal window to verify your implementation.

Problem 1c: Update assignment of points to centroids [3 points] ⌨

Implement `update_assignment(data, centroids)` function in `kmeans.py`. We have provided testing code for you to verify your implementation. Uncomment this line:

```
test_update_assignment()
```

at the bottom of `kmeans_tests.py` and run `python kmeans_tests.py` in a terminal window to verify your implementation.

Problem 1d: Update centroids [3 points] ⌨

Implement `mean_of_points(data)` function in `kmeans.py`. We have provided testing code for you to verify your implementation. Uncomment this line:

```
test_mean_of_points()
```

at the bottom of `kmeans_tests.py` and run `python kmeans_tests.py` in a terminal window to verify your implementation.

Now Implement `update_centroids(assignment_dict)` function in `kmeans.py`. Notice that you should not hard-code this dimensionality of data - your code needs to run regardless of the dimension of data! We have provided testing code for you to verify your implementation. Uncomment this line:

```
test_update_centroids()
```

at the bottom of `kmeans_tests.py` and run `python kmeans_tests.py` in a terminal window to verify your implementation.

You have now completed all parts needed to run `main()`. As a sanity check, try running all the tests together one more time by typing:

```
python kmeans_tests.py
```

in the terminal window. You should see:

```
All tests passed.
```

Problem 1e: Clustering 2D points [1 points] ✎

You have now completed all parts needed to run `main()`. Now, let's verify your implementation by running the algorithm on 2D points. At the bottom of `kmeans.py`, there are codes for loading the 2D data points and the initial centroids we provided in from .csv files, and calling the main function. Run the program by typing this:

```
python kmeans.py
```

in the terminal window. If you are succesful, you should see:

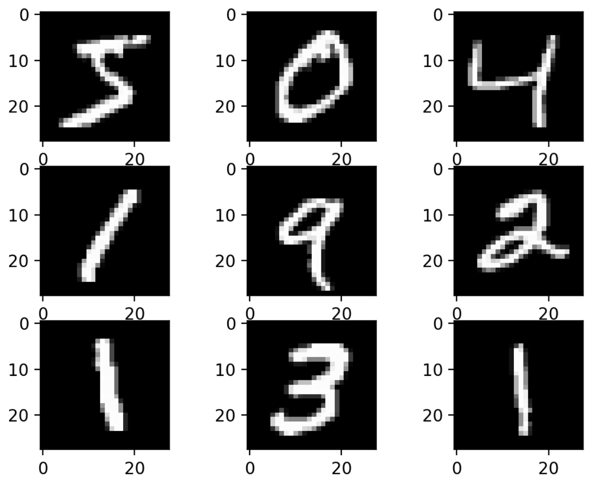
```
K-means converged after 7 steps.
```

as the output of the program. In addition, there should be 7 plots generated, in the results/2D/ folder. **Attach the 7 plot images to** `name_studentid.pdf`

Problem 1f: Trying the Algorithm on MNIST [3 points] ✎

Now that you have successfully run K-means clustering on 2-dimensional data points with 2 centroids, we will use your code to do clustering of 784-dimensional data points with 10 centroids. You won't need to change any of the code that you wrote.

We will apply the algorithm to a real world data set known as the Modified national Institute of Standards and Technology database (MNIST). It is a dataset that consists of images of hand-written digits. Here are nine examples images from the MNIST data set (we are using a very small subset of all MNIST images):



Each of these images is 28 pixels by 28 pixels, for a total of 784 pixels (pixels have values between 0 and 255 (0=black, 255=white). Each image can be thought of as a 784-dimensional data point, represented as a list containing 784 numbers. For the images shown above, many of the numbers would be zero since the image is mostly black. so the nine sample images above are actually nine sample data points!

We will see how applying K-means to these digit images helps us discover clusters of images which contain mostly one kind of digit.

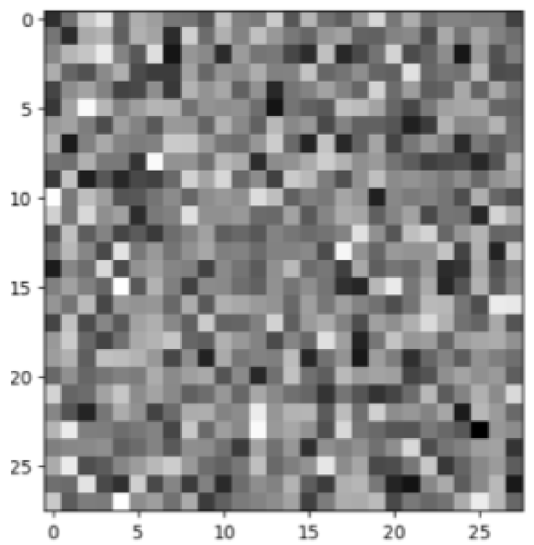
Run this command (it may take a minute to run on this data set):

```
python kmeans_MNIST.py
```

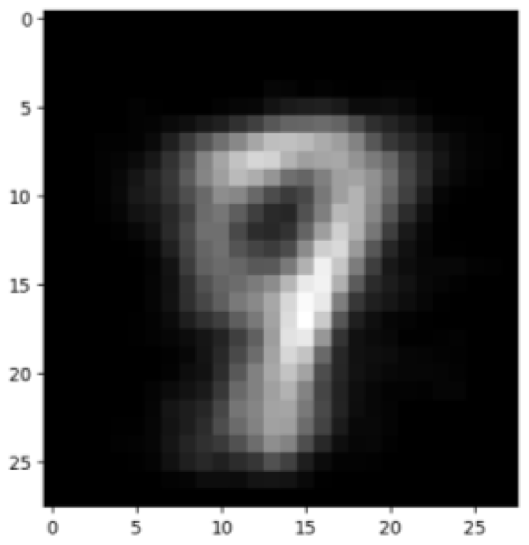
You should see the following output:

```
K-means converged after 18 steps.
```

Meanwhile, in the results/MNIST folder, the `init` and `final` folders should be filled with the centroids plotted as images. In the `init` folder, the centroids should look like Gaussian noise (left below), because that's how we randomly initialized them. In the `final` folder, they should look like actual digits (right below).



Centroid 0: Initialization



Centroid 0: After Convergence

Why the centroid of each cluster looks like an actual digit? This quesion will be ungraded. But if you do not write any answers, you will get 0 points. We want you to at least think about what happened to the centroids. **Write your answers in** `name_studentid.pdf`.

Problem 2: Soft K-Means Clustering

In Problem 1, we use the hard K-means algorithm which assign all data points to one of the k clusters. However, some data points may be ambiguous to assign to one cluster. In this case, we can use soft K-means clustering which yields soft assignments of data points to clusters.

As with the previous hard K-means clustering, let's assume we already have K centroids, initialized randomly. To update the centroid locations, we use the following iterative approach:

1. First, we compute all data point's responsibilities for each cluster. The data point x_i has a responsibility r_{ki} to each cluster k when the distance between x_i and the centroid of cluster k is $dist_{ki}$:

$$r_{ki} = \frac{exp(-\beta\, dist_{ki})}{\sum_{l \in K} exp(-\beta\, dist_{li})}$$

2. Then, update the centroid locations for each cluster. The new centroid is the weighted sum of all data points using the responsibilities of the corresponding cluster:

$$centroid_k = \frac{\sum_{i \in N} r_{ki} x_i}{\sum_{i \in N} r_{ki}}$$

Now, we implement the functions in the `soft_kmeans.py` and see how the 2D points dataset we used in Problem 1 is clustered. You can check the data structures of the output of each function in the `soft_kmeans_tests.py`.

Problem 2a: Calculate the responsibility of each cluster [3 points] ⌨

Implement `get_responsibility(data_point, centroids, beta)` function in `soft_kmeans.py`. We have provided testing code for you to verify your implementation. Uncomment this line:

```
test_get_responsibility()
```

at the bottom of `soft_kmeans_tests.py` and run `python soft_kmeans_tests.py` in a terminal window to verify your implementation.

Problem 2b: Update soft assignment of points to centroids [2 points] ⌨

Implement `update_soft_assignment(data, centroids, beta)` function in `soft_kmeans.py`. We have provided testing code for you to verify your implementation. Uncomment this line:

```
test_update_soft_assignment()
```

at the bottom of `soft_kmeans_tests.py` and run `python soft_kmeans_tests.py` in a terminal window to verify your implementation.

Problem 2c: Update centroids [3 points]

Implement `update_centroids(soft_assignment_dict)` function in `soft_kmeans.py`. We have provided testing code for you to verify your implementation. Uncomment this line:

```
test_update_centroids()
```

at the bottom of `soft_kmeans_tests.py` and run `python soft_kmeans_tests.py` in a terminal window to verify your implementation.

You have now completed all parts needed to run `main()`. As a sanity check, try running all the tests together one more time by typing:

```
python soft_kmeans_tests.py
```

in the terminal window. You should see:

```
All tests passed.
```

Problem 2d: Clustering 2D points [2 points]

You have now completed all parts needed to run the soft K-means algorithm. Let's verify your implementation by running the algorithm on 2D points. Run the program by typing this:

```
python soft_kmeans.py
```

in the terminal window. If you are successful, you should see:

```
7 iterations were completed.
```

as the output of the program.

The 7 plots should be generated in the `results/soft_2D` folder. In the plot, the greater the responsibility of centroid 0 for a data point, the darker the red color. Conversely, the greater the responsibility of centroid 1 for a data point, the darker the blue color. Let's compare the newly generated plots with the plots generated in Problem 1e. Also, **attach the new 7 plots** to `name_studentid.pdf`.

As you may have already noticed, the hyper-parameter β is used when calculating responsibility. Change the value of β to 50 in `main()` function of `soft_kmeans.py` file. What changes have been made to the plots? Think about how β affects soft K-means clustering and **write your answer with a step6 plot when setting β to 50** in `name_studentid.pdf`.