CS 335: Lab Assignment 4

(TAs in charge: Abhinav Goyal, Rishabh Shah)

The focus of this lab is **clustering** algorithms. We will look at the vanilla **k-means clustering** algorithm, assess its performance, and develop some variants. Finally, we will use clustering for image compression.

Code and data sets

Some starter code and data sets have been provided for you here: clustering.tar.gz. Uncompress this directory to find the following files. You will need to implement some functions in `cluster.py` and `image.py`

Note: You are not allowed to change the code anywhere else other than these functions. Also, you are not allowed to use any external libraries: **don't use any import statements other than the ones already present in the code**.

`cluster.py`

This file implements the clustering algorithms. You need to implement the following functions in the file, under different tasks (described below).

1. `distance_euclidean(p1, p2)`
2. `distance_manhattan(p1, p2)`
3. `initialization_forgy(data, k)`
4. `initialization_kmeansplusplus(data, distance, k)`
5. `kmeans_iteration_one(data, centroids, distance)`
6. `kmedians_iteration_one(data, centroids, distance)`
7. `hasconverged(old_centroids, new_centroids, epsilon=1e-1)`
8. `iteration_many(data, centroids, distance, maxiter, algorithm, epsilon=1e-1)`
9. `performance_SSE(data, centroids, distance)`
10. `performance_L1(data, centroids, distance)`

Run `python cluster.py -h` to get help on how to run the program and command line options.

`image.py`

This file implements the image processing algorithms. You need to implement the following functions in the file, under different tasks (described below).

1. `read_image(filename)`
2. `preprocess_image(img)`
3. `label_image(img, cluster_centroids)`
4. `write_image(filename, img)`
5. `compress_image(cluster_labels, cluster_centroids)`

Run `python image.py -h` to get help on how to run the program and command line options.

Tip: In both files, each function is labeled with the task number. If you are working on Task 1, search the corresponding file for `task1`.

`autograder.py`

Execute this script as `python autograder.py [task-number]`. Running `python autograder.py all` will give you an estimate of your final grade (except for the manually graded tasks). We may use our own test cases and judgement before finalising the grade.

`datasets/*`

This directory contains the data sets you will be using during this lab. Each line in the files is a comma separated list of numbers, representing a point in $\mathbb{R}^d$.

`images/*`

This directory contains the images you will be needing for section 4 (Image Compression).

Submission details are at the bottom.

Section 1: Understanding k-means clustering

k-means clustering is an unsupervised learning algorithm, which aims to cluster the given data set into k clusters.

Formally, the algorithm tries to solve the following problem:

Given a set of observations $(x^1, x^2, \ldots, x^n), x^i \in \mathbb{R}^d$, partition them into $k~(\leq n)$ sets $S = \{S_1, S_2, \ldots, S_k\}$ so as to minimise the within-cluster sum of squares (WCSS) (sum of squared distance of each point in the cluster to its cluster centre). In other words, the objective is to find:
$${\displaystyle{\underset{\mathbf{S}}{\operatorname{argmin}}}
\sum_{i=1}^{k}\sum_{\mathbf{x}\in S_{i}}\left\|x-\mu_i\right\|_{2}^{2}}$$
where μ_i is the mean of points in S_i.

Finding the optimal solution to this problem is NP-Hard for general n, d, and k. k-means clustering presents an iterative solution to the above clustering problem, that provably converges to a **local optimum**.

k-means Algorithm[1]

1. Define the initial clusters' centroids. This step can be done using different strategies. A very common one is to assign random values for the centroids of all groups. Another approach, known as ***Forgy initialisation***, is to use the positions of k different points in the data set as the centroids.
2. Assign each entity to the cluster that has the closest centroid. In order to find the cluster with the closest centroid, the algorithm must calculate the distance between all the points and each centroid.
3. Recalculate the centroids. Each component of the centroid is updated, and set to be the average of the corresponding components of the points that belong to the cluster.
4. Repeat steps 2 and 3 until points no longer change cluster.

Note:

- In step 2, if more than one centroid is closest to a data point, you can break the ties arbitrarily.
- In step 3, if any cluster has no points in it, just pass on the value of the centroid from the previous step.

Cluster Initialisation: Forgy

Forgy initialisation is one of the simplest ways to initialise the clustering algorithm. The Forgy method randomly chooses k points from the data set and uses these as the initial centroids. This ensures that the initial clusters are uniformly spread out across the data.

Measuring Performance: Sum Squared Error

(SSE)

k-means clustering tries to locally minimise the Sum Squared Error, where the error associated with each data point is taken as its Euclidean distance from the cluster centre.

Task 1: Implementing k-means clustering (3 marks)

Implement all of the following 6 functions in `cluster.py`.

1. `distance_euclidean(p1, p2)`
2. `initialization_forgy(data, k)`
3. `kmeans_iteration_one(data, centroids, distance)`
4. `hasconverged(old_centroids, new_centroids, epsilon)`
5. `iteration_many(data, centroids, distance, maxiter, algorithm, epsilon)`
6. `performance_SSE(data, centroids, distance)`

Test your code by running this command.

```
python cluster.py -s $RANDOM datasets/flower.csv
```

Try different values of k with different random seeds using command line options (see `python cluster.py -h`). You can also try changing epsilon and max. number of iterations.

Evaluation: Each correctly implemented function will fetch you 0.5 marks.

Test with `python autograder.py 1`. This shows your final grade for this task.

Task 2: Testing and Performance (2 marks)

Test your code on the following data sets.
`datasets/100.csv`: Use , numexperiments
`datasets/1000.csv` : Use , numexperiments
`datasets/10000.csv` : Use , numexperiments

Use epsilon. Here is an example.

```
python cluster.py -e 1e-2 -k 2 -n 100 datasets/100.csv
```

Answer the following 2 questions in the file named `solutions.txt`.

1. Run your code on `datasets/garden.csv`, with different values of k. Look at the performance plots and answer whether the SSE of the k-means clustering algorithm ever increases as the iterations are performed. Why or why not?**[1 mark]**
2. Look at the files `3lines.png` and `mouse.png`. Manually draw cluster boundaries around the 3 clusters visible in each file (no need to submit the hand drawn clustering). Test the k-means algorithm with different random seeds on the data sets `datasets/3lines.csv` and `datasets/mouse.csv`. How does the algorithm's clustering compare with the clustering you did by hand? Why do you think this happens? **[1 mark]**

Evaluation: The text questions carry marks as specified. Make sure to write clean, succinct answers.

It is worth noting that k-means can sometimes perform poorly! Test your algorithm on the `datasets/rectangle.csv` data set several times, using $k=2$. Depending on the initialisation, k-means can converge to poor clusterings.

Section 2: k-means++[2]

The clustering performance of the k-means algorithm is sensitive to the initial

cluster centroids. A poor initialisation can lead to arbitrarily poor clustering. The k-means++ initialisation algorithm addresses this problem; standard k-means clustering is performed after the initialisation step. With the k-means++ initialisation, the algorithm is guaranteed to find a solution that is $\mathcal{O}(\log k)$ competitive to the optimal k-means solution in expectation.

The intuition behind this approach is that spreading out the k initial cluster centres is a good idea: the first cluster centre is chosen uniformly at random from the data points, after which each subsequent cluster centre is chosen from the remaining data points with a probability proportional to the point's squared distance to that point's closest existing cluster centre.

The algorithm is as follows.

1. Choose one centre uniformly at random from among the data points.
2. For each data point x^i, compute $D(x^i)$, the distance between x^i and the nearest centre that has already been chosen.
3. Choose one new data point at random as a new cluster centre, using a probability distribution in which point is chosen with probability proportional to . In other words, the probability that is made a cluster centre is .
4. Repeat steps 2 and 3 until centres have been chosen.
5. Now that the initial centres have been chosen, proceed using standard k-means clustering.

This seeding method yields considerable improvement in both the speed of convergence, and the final error of k-means.

Task 3: Implementing k-means++ (3 marks)

Implement the following function in `cluster.py`.

1. `initialization_kmeansplusplus(data, distance, k)`

Note: You are expected to **provide elaborate comments along with your code** (for the function). Your marks depend on whether the TAs are able to understand your code and establish its correctness.

Test with `python autograder.py 3`.

Use your code by running the following command (this is for you to check that the code is working, not used for evaluation).

```
python cluster.py -i kmeans++ -k 8 -e 1e-2 datasets/flower.csv
```

After implementing your code, test it on these data sets.

`datasets/100.csv`: Use , numexperiments
`datasets/1000.csv` : Use , numexperiments
`datasets/10000.csv` : Use , numexperiments

Use epsilon.

Answer the following question in the file `solutions.txt`.

1. For each data set and initialisation algorithm (Forgy and k-means++), report "average SSE" and "average iterations". Explain the results.

Evaluation: Correct implementation of the kmeans++ function will fetch you mark. The text question is worth marks.

Notice how:

- kmeans++ initialisation leads to considerably less cluster movement compared to Forgy initialisation;
- Despite using kmeans++, the algorithm will sometimes converge to poor solutions.

Section 3: k-medians clustering[3]

Though k-means clustering gives good results most of the time, as you would have seen by now, it is very sensitive to any outliers in the data set. In this section we will explore k-medians clustering which is more robust to outliers as compared to k-means.

The algorithm for k-medians clustering is same as that of k-means clustering except for the recalculation of centroids in step 3. Here we **update each component of the centroid to be the median of the corresponding components of the points that belong to the cluster**: that is, where are the points in cluster and is the corresponding centroid.

Measuring Performance: Sum of L1-norm Error

As you might have already guessed, k-medians algorithm tries to locally minimise the sum of L1-norm distance of each point from it's cluster centroid. Formally, k-medians algorithm locally solves the optimisation problem: find where is the median of points in .

Task 4: Implementing and comparing k-medians clustering (3 marks)

Implement the following 3 functions in `cluster.py`.

1. `distance_manhattan(p1, p2)`
2. `kmedians_iteration_one(data, centroids, distance)`
3. `performance_L1(data, centroids, distance)`

```
python cluster.py -a kmedians -k 8 -e 1e-2 datasets/flower.csv
```

Test with `python autograder.py 4`.

Visually compare the clustering obtained from both k-means and k-medians on the following data sets.
`datasets/outliers3.csv` : Use
`datasets/outliers4.csv` : Use
`datasets/outliers5.csv` : Use

Use epsilon and forgy initialisation for all experiments. Use **euclidean distance metric for k-means and manhattan distance metric for k-medians**. Ideally you should run the experiments multiple times with different random seeds to visualise the difference between both the algorithms as random initialisation can cause both of them to converge to bad clustering sometimes.

You can use `--outliers` flag to better visualise the clustering without plotting the outliers.

Answer the following question in the file named `solutions.txt`.

1. Can you observe from the visualisation that k-medians algorithm is more robust to outliers as compared to k-means? Why do you think this happens?

Evaluation: Each correctly implemented function in this task will fetch you marks. The text question is worth marks.

Section 4: Image Compression

Having understood how k-means clustering and its variants work, we will use the algorithm to compress an RGB image. A naive approach to store such an image is to use three 2-D matrices with R, G, and B values. If the image dimensions are and we are using 8-bits to store each value (0-255) in the matrices, we will have to store (possibly) different numbers to represent all the colours. The basic idea

behind image compression using clustering is to use less number of colours to represent the image. A typical procedure to achieve this will proceed as follows:

1. **Preprocessing** - Read the RGB image and convert it into a data format suitable for clustering.
2. **Clustering** - Use k-means clustering to find and store cluster centroids representing the data.
3. **Compressing** - For each pixel in the image, replace its pixel value with its representative cluster number and store the matrix thus optained as a greyscale image.

To get the image back, we read the stored grayscale image containing cluster numbers and for each pixel, we replace the pixel value with the value of corresponding cluster centroid.

Task 5: Image Preprocessing (1 mark)

This task requires you to implement functions for reading an image and processing it to get a data format suitable for clustering. For reading the image, you'll need to understand how RGB and greyscale images are stored in PPM and PGM file formats (see PPM and PGM binary formats). For more details on how to implement any function, please read the comments written in the respective function.

Implement the following functions in `image.py`.

1. `read_image(filename)`
2. `preprocess_image(image)`

Run the following command to process the image data and save it as a csv file.

```
python image.py preprocess images/tiger.ppm
```

Test with `python autograder.py 5`.

Evaluation: Each correctly implemented function in this task will fetch you mark.

Task 6: Clustering (Ungraded)

Now run the following command to cluster the data saved in task 5 using k-means algorithm with forgy initialisation and 64 clusters.

```
python cluster.py -k 64 -i forgy -e 1 -o image_centroids.csv
image_data.csv
```

Task 7: Image Compression (1 mark)

In this task you will use the saved cluster centroids to actually compress the image: step 3 of the above-stated procedure. For more details on how to implement any function, please read the comments written in the respective function.

Implement the following functions in `image.py`.

1. `label_image(img, cluster_centroids)`
2. `write_image(filename, img)`

Run the following command to compress the image and save it as a PGM file.

```
python image.py compress -o compressed.pgm images/tiger.ppm image_centroids.csv
```

Test with `python autograder.py 7`.

Evaluation: Each correctly implemented function in this task will fetch you marks.

Task 8: Image Decompression and analysis (2 marks)

Now that you have compressed the image, let's decompress it and observe its differences from the original image if any. You will need to use the saved cluster labels image and the cluster centroids to decompress the image. For more details on how to implement the function, please read the comments written in the function.

Implement the following function in `image.py`.

1. `decompress_image(cluster_labels, cluster_centroids)`

Run the following command to decompress the image and save it as a PPM file.

```
python image.py decompress -o tiger_decompressed.ppm compressed.pgm image_centroids.csv
```

Observe the decompressed image and compare it with the original one. Can you notice any difference? Try compression followed by decompression for different values of and answer the following question in the file named `solutions.txt`. Also, submit `tiger_decompressed.ppm` thus created (using 64 clusters).

1. What do you observe as we reduce the number of clusters ()? Answer in reference to the quality of decompressed image. **[0.5 marks]**
2. You can observe that for the small number of clusters, the degree of compression (original size/compressed size) is about the same as that of when we use larger number of clusters even though we need to store lesser number of colours. Can you tell why? How can we increase this ratio in case of smaller number of clusters? **[1 mark]**

Test with `python autograder.py 8`.

Evaluation: Correctly implemented function in this task will fetch you 0.5 marks. The text questions carry marks as indicated above.

Submission

You are expected to work on this assignment by yourself. You may not consult with your classmates or anybody else about their solutions. You are also not to look at solutions to this assignment or related ones on the Internet. You are allowed to use resources on the Internet for programming (say to understand a particular command or a data structure), and also to understand concepts (so a Wikipedia page or someone's lecture notes or a textbook can certainly be consulted). However, you **must** list every resource you have consulted or used in a file named `references.txt`, explaining exactly how the resource was used. Failure to list all your sources will be considered an academic violation.

Be sure to write all the observations/explanations in the **`solutions.txt`** file. We have mentioned wherever there is such a requirement. Find the keyword **`'solutions.txt'`** in the page.

Your submission directory must be named as `la4-[rollnumber]`. It must contain the following 5 files.

1. `cluster.py`
2. `image.py`
3. `solutions.txt`
4. `tiger_decompressed.ppm`
5. `references.txt`

Do not include any other files.

Compress your directory into `la4-[rollnumber].tar.gz`, and upload it to Moodle under Lab Assignment 4.
