AI 1013: Programming for AI

Assignment 4

Due on: 24 April 2025, 11:59 PM IST.

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Read each question carefully, and answer all parts. Write well-documented codes with contextual variable names. Put the codes (.py or .ipynb) and plots (.png, .jpg, .jpeg, .pdf) of all the questions into one single folder, compress this folder, and upload the .zip file. You should name the .zip file as `AI1013_assignment_4_<your roll number>.zip`.

Contents

1 K-Means Clustering

For this task, we will generate a synthetic dataset, and implement a popular clustering technique called *K-means clustering* to cluster the points in the dataset.

1.1 Dataset Generation

1. We will generate two semi-circles of radius 1 each. One of the semi-circles faces downwards, centered at $(0, 0)$, and the other faces upward, centered at $(1, 0)$.
2. Recall that the parameterized coordinates of any point on the circumference of a circle with center (a, b) and radius r are of the form $(a + r\cos\theta, b + r\sin\theta)$ for some $\theta \in [0, 2\pi]$.
3. Using the functions `np.linspace` or `np.arange`, generate an array `theta` which comprises 500 equidistant points between 0 and π.
4. Add the following line of code: `np.random.seed(123)`. A seed helps to reproduce results, especially if there is same randomness present in the code.
5. The coordinates of any point on the circumference of the first semicircle are of the form $(\cos\theta, \sin\theta)$. Store the x-coordinates in an array called `semicircle1_x`. Generate an array of noise using `np.random.normal` with mean 0 and standard deviation 0.1 and add it to `semicircle1_x`. Similarly, store the y-coordinates in an array called `semicircle1_y`. Generate an array of noise using `np.random.normal` with mean 0 and standard deviation 0.1 and add it to `semicircle1_y`.
6. The coordinates of any point on the circumference of the second semicircle are of the form $(1+\cos\theta, -\sin\theta)$. Repeat the previous step to store the points for the second semicircle and store the x- and y-coordinates in `semicircle2_x` and `semicircle2_y` respectively.
7. Plot the points for the first and the second semicircle on the same graph using the same color. We use the same color because these points comprise the original dataset, and given the original dataset, we do not yet know the clusters. Our aim now is to use a clustering algorithm to identify clusters within the dataset.
8. Store all the data points in one single array as tuples for convenience. Tuples are a hashable data type, whereas lists are not. This becomes important when we use dictionaries for clustering. Create an array `data_points`, and you can use the following code-sketch for adding all the data points as tuples to it:

```
d1 = [(x,y) for x,y in zip(s1_x,s1_y)]
d2 = [(x,y) for x,y in zip(s2_x,s2_y)]
data_points = d1 + d2
```

1.2 K-Means Clustering

K-Means clustering is an iterative algorithm comprising three parts: initialization, iteration, and termination. The idea behind the algorithm is that each cluster has a representative centroid. The centroids are initialised randomly, and then updated sequentially with each iteration over the data. This process continues until a certain stopping criterion is met. Several stopping criteria such as checking for the number of points updated during an iteration, checking for the average distance of points from centroids, etc., are employed in practice.

1. Initialisation:
   Note that all of our data lies in the region $[-2, 3] \times [-2, 2]$. Pick two points uniformly at random from this region, and set these as the initial values for the centroids. Let the first centroid be given by `C_1` while the second centroid be given by `C_2`.
2. Iteration:
   (a) Initialize a dictionary called `mappings`to keep track of each point and the cluster associated with it. Your dictionary `mappings`should be as follows (only a template):

$$\{(x_1, y_1) : 1, (x_2, y_2) : 2, \ldots (x_{1000}, y_{1000}) : 2\}.$$

(b) Initialize a variable called `num_points_updated` to 0. This will keep track of the number of points whose cluster was updated during an iteration. Also, initialize `max_iters` to $10,000$ and `current_iters` to 0. Your algorithm should stop if no point gets updated during an iteration, or if the maximum number of iterations is reached.

(c) Your algorithm should execute the following steps until stoppage:

   i. Assign each point in the dictionary to one of the two clusters. For this, check the point's Euclidean distance from each of the two centroids, and assign the cluster corresponding to the nearest centroid. Keep track of the number of points whose clusters were updated.

   ii. Update the centroid values. For this, compute the average of the coordinates of all the data points mapped to a particular cluster. For instance, if A is the set of all points that have been mapped to cluster 1, then

$$\texttt{C\_1} = \frac{1}{|A|}\sum_{\vec{a}\in A}\vec{a},$$

   where $\vec{a}$ denotes the coordinates of a generic point in A, and $|A|$ denotes the number of points in A.

3. Plot the clusters you have obtained with different colors on the same plot.

Note: You may observe that K-means clustering may not give the expected clustering of data points. Thus, we have other methods such as DBSCAN which also takes into account the densities of the points, i.e., number of points in the neighbourhood of a point to assign clusters.

2 K-Means++ Clustering on the `Covertype` Dataset

The `Covertype` dataset[1] contains $581{,}012$ forest-cover observations described by 54 features (10 quantitative, 4 binary "Wilderness Area" flags, and 40 binary "Soil Type" flags). Each observation is annotated with one of seven cover-type labels:

`{1: Spruce/Fir, 2: Lodgepole-Pine, 3: Ponderosa-Pine, 4: Cottonwood/Willow, 5: Aspen, 6: Douglas-fir, 7: Krummholz}`

Our goal is to cluster a manageable subsample of this high-dimensional dataset using the *K-means++* initialization strategy and to compare it with random initialization.

2.1 Data Preparation

1. **Download or load** the dataset. In `scikit-learn` you may call `sklearn.datasets.fetch_covtype`.
2. **Subsample** exactly 20000 observations chosen uniformly at random (to keep runtime reasonable). Use a random seed of 42 (`np.random.default_rng(seed=42)`) for reproducibility.
3. **Standardize** the quantitative features to zero mean and unit variance. Keep the binary indicators unchanged. Denote the resulting data matrix by $X \in \mathbb{R}^{20000\times 54}$.

2.2 Implementing K-Means++

Recall that K-means++ selects the first centroid uniformly at random from the data and, for each subsequent centroid, samples a point with probability proportional to the squared distance to the nearest already-chosen centroid.

1. Write your own function `kmeans_plus_plus_init(X, k)` that returns k initial centroids using the above procedure.
2. Set $k = 7$ (the number of cover types) and initialize the centroids with your implementation. Then run vanilla K-means iterations until either:
   - the cluster assignments stop changing, or
   - `max_iters` $= 500$ iterations are reached.

   Use Euclidean distance in the full 54-dimensional space.
3. Record the final centroids $\{C_1, \ldots, C_7\}$ and the cluster assignment for every point.

2.3 Evaluation and Visualisation

1. Report the within-cluster sum of squared distances (WC-SSD) achieved by **(i)** your K-means++ run and **(ii)** a baseline run that initializes the k centroids *completely at random*. Comment (2–3 sentences) on the difference.
2. Run your K-means++ implementation for values of K from 1 to 20, with 10 runs for each value of K (remove the random seed of 42 for this step). For each K from 1 to 20, compute the mean value of within-cluster sum of squared distances for the 10 runs. Plot these values on the Y-axis with the value of K on the X-axis. Read about the Elbow method and comment on it in this context. In particular, do you see any sharp drop in WC-SSD for any particular value of K?
3. Run the above step (10 runs each for K from 1 to 20) by initializing completely at random. Now plot these values along with the values from the step above (for K-Means++) on Y-axis with value of K on X-axis. Comment on the differences.

Notes

- Because `Covertype` is highly imbalanced (some cover types are rare), your clusters are *not* expected to align perfectly with the labels; the goal is to observe whether K-means^{++} initialization yields more coherent (lower WC-SSD) partitions than naive random initialization.
- **Do not** call `sklearn.cluster.KMeans`—implement the algorithm yourself.

[1]Blackard, Jock *Covertype Data Set*. UCI Machine-Learning Repository, 1998. Available at `https://archive.ics.uci.edu/dataset/31/covertype`.