Department of
Computer Science and Engineering

Title: Clustering: Implementation of K-Means Clustering Algorithm Using Python

Data Mining LAB

CSE 424

Green University of Bangladesh

1 Objective(s)

- To learn the fundamentals and mathematics behind the popular k-means clustering algorithm.
- To implement k-means clustering algorithm using Python.

2 Clustering

Clustering (or cluster analysis) is a technique that allows us to find groups of similar objects, objects that are more related to each other than to objects in other groups. Examples of business-oriented applications of clustering include the grouping of documents, music, and movies by different topics, or finding customers that share similar interests based on common purchase behaviors as a basis for recommendation engines.
In this tutorial, we will learn about one of the most popular clustering algorithms, k-means, which is widely used in academia as well as in industry.

2.1 Fundamentals of K-Means Clustering

As we will see, the k-means algorithm is extremely easy to implement and is also computationally very efficient compared to other clustering algorithms, which might explain its popularity. The k-means algorithm belongs to the category of prototype-based clustering.

Prototype-based clustering means that each cluster is represented by a prototype, which can either be the centroid (average) of similar points with continuous features, or the medoid (the most representative or most frequently occurring point) in the case of categorical features.

While k-means is very good at identifying clusters with a spherical shape, one of the drawbacks of this clustering algorithm is that we have to specify the number of clusters, k, a priori. An inappropriate choice for k can result in poor clustering performance — we will discuss later in this tutorial how to choose k.

Although k-means clustering can be applied to data in higher dimensions, we will walk through the following examples using a simple two-dimensional dataset for the purpose of visualization. It can be seen in figure 1.

```python
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

# create dataset
X, y = make_blobs(
   n_samples=150, n_features=2,
   centers=3, cluster_std=0.5,
   shuffle=True, random_state=0
)

# plot
plt.scatter(
   X[:, 0], X[:, 1],
   c='white', marker='o',
   edgecolor='black', s=50
)
plt.show()
```

The dataset that we just created consists of 150 randomly generated points that are roughly grouped into three regions with higher density, which is visualized via a two-dimensional scatterplot.

In real-world applications of clustering, we do not have any ground truth category information (information provided as empirical evidence as opposed to inference) about those samples; otherwise, it would fall into the category of supervised learning. Thus, our goal is to group the samples based on their feature similarities, which can be achieved using the k-means algorithm that can be summarized by the following four steps:

1. Randomly pick k centroids from the sample points as initial cluster centers.
2. Assign each sample to the nearest centroid.
3. Move the centroids to the center of the samples that were assigned to it.

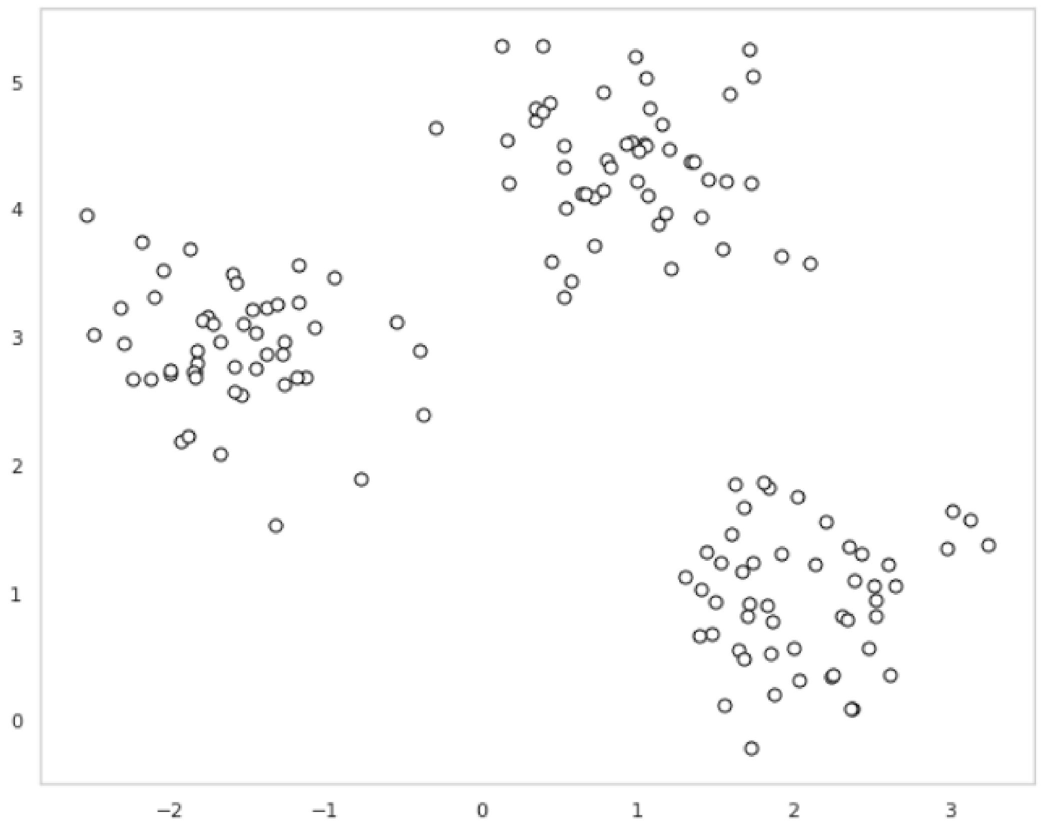

Figure 1: Output

4. Repeat steps 2 and 3 until the cluster assignments do not change or a user-defined tolerance or maximum number of iterations is reached.

Now, the next question is how do we measure similarity between objects? We can define similarity as the opposite of distance, and a commonly used distance for clustering samples with continuous features is the squared Euclidean distance between two points x and y in m-dimensional space.

Based on this Euclidean distance metric, we can describe the k-means algorithm as a simple optimization problem, an iterative approach for minimizing the within-cluster Sum of Squared Errors(SSE), which is sometimes also called cluster inertia.

Note that when we are applying k-means to real-world data using a Euclidean distance metric, we want to make sure that the features are measured on the same scale and apply z-score standardization or min-max scaling if necessary.

2.2 K-means Clustering Using scikit-learn

Now that we have learned how the k-means algorithm works, let's apply it to our sample dataset using the KMeans class from scikit-learn's cluster module:

```python
from sklearn.cluster import KMeans

km = KMeans(
    n_clusters=3, init='random',
    n_init=10, max_iter=300,
    tol=1e-04, random_state=0
)
y_km = km.fit_predict(X)
```

Using the preceding code, we set the number of desired clusters to 3. We set n_init=10 to run the k-means clustering algorithms 10 times independently with different random centroids to choose the final model as the one with the lowest SSE. Via the max_iter parameter, we specify the maximum number of iterations for each single run (here, 300).

Note that the k-means implementation in scikit-learn stops early if it converges before the maximum number of iterations is reached. However, it is possible that k-means does not reach convergence for a particular run, which can be problematic (computationally expensive) if we choose relatively large values for max$_iter$.

One way to deal with convergence problems is to choose larger values for tol, which is a parameter that controls the tolerance with regard to the changes in the within-cluster sum-squared-error to declare convergence. In the preceding code, we chose a tolerance of 1e-04 (= 0.0001).

A problem with k-means is that one or more clusters can be empty. However, this problem is accounted for in the current k-means implementation in scikit-learn. If a cluster is empty, the algorithm will search for the sample that is farthest away from the centroid of the empty cluster. Then it will reassign the centroid to be this farthest point.

Now that we have predicted the cluster labels y_km, let's visualize the clusters that k-means identified in the dataset together with the cluster centroids. These are stored under the cluster_centers_ attribute of the fitted KMeans object:

```
# plot the 3 clusters
plt.scatter(
    X[y_km == 0, 0], X[y_km == 0, 1],
    s=50, c='lightgreen',
    marker='s', edgecolor='black',
    label='cluster 1'
)

plt.scatter(
    X[y_km == 1, 0], X[y_km == 1, 1],
    s=50, c='orange',
    marker='o', edgecolor='black',
    label='cluster 2'
)

plt.scatter(
    X[y_km == 2, 0], X[y_km == 2, 1],
    s=50, c='lightblue',
    marker='v', edgecolor='black',
    label='cluster 3'
)

# plot the centroids
plt.scatter(
    km.cluster_centers_[:, 0], km.cluster_centers_[:, 1],
    s=250, marker='*',
    c='red', edgecolor='black',
    label='centroids'
)
plt.legend(scatterpoints=1)
plt.grid()
plt.show()
```

In the resulting scatterplot in figure 2, we can see that k-means placed the three centroids at the center of each sphere, which looks like a reasonable grouping given this dataset.

2.3 The Elbow Method

Although k-means worked well on this toy dataset, it is important to reiterate that a drawback of k-means is that we have to specify the number of clusters, k, before we know what the optimal k is. The number of clusters to choose may not always be so obvious in real-world applications, especially if we are working with a higher dimensional dataset that cannot be visualized.

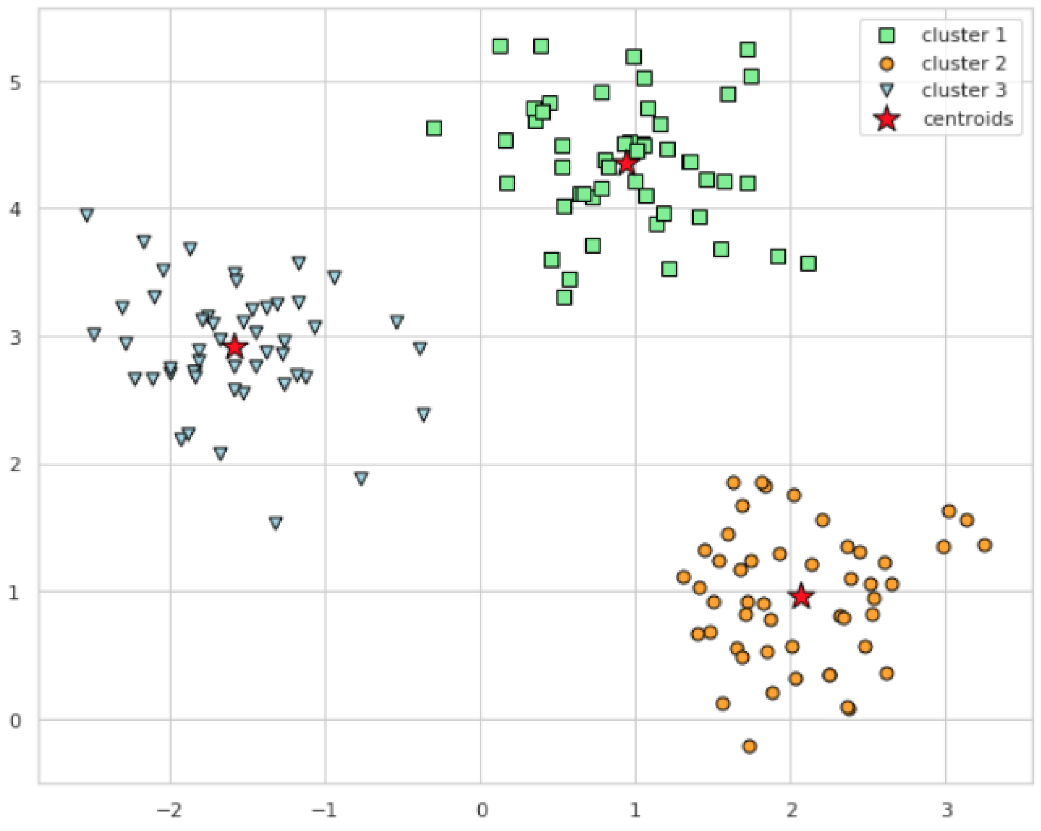

Figure 2: Output

The elbow method is a useful graphical tool to estimate the optimal number of clusters k for a given task. Intuitively, we can say that, if k increases, the within-cluster SSE (“distortion”) will decrease. This is because the samples will be closer to the centroids they are assigned to.

The idea behind the elbow method is to identify the value of k where the distortion begins to decrease most rapidly, which will become clearer if we plot the distortion for different values of k:

```python
# calculate distortion for a range of number of cluster
distortions = []
for i in range(1, 11):
    km = KMeans(
        n_clusters=i, init='random',
        n_init=10, max_iter=300,
        tol=1e-04, random_state=0
    )
    km.fit(X)
    distortions.append(km.inertia_)

# plot
plt.plot(range(1, 11), distortions, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Distortion')
plt.show()
```

As we can see in figure 3 the resulting plot, the elbow is located at k = 3, which is evidence that k = 3 is indeed a good choice for this dataset.

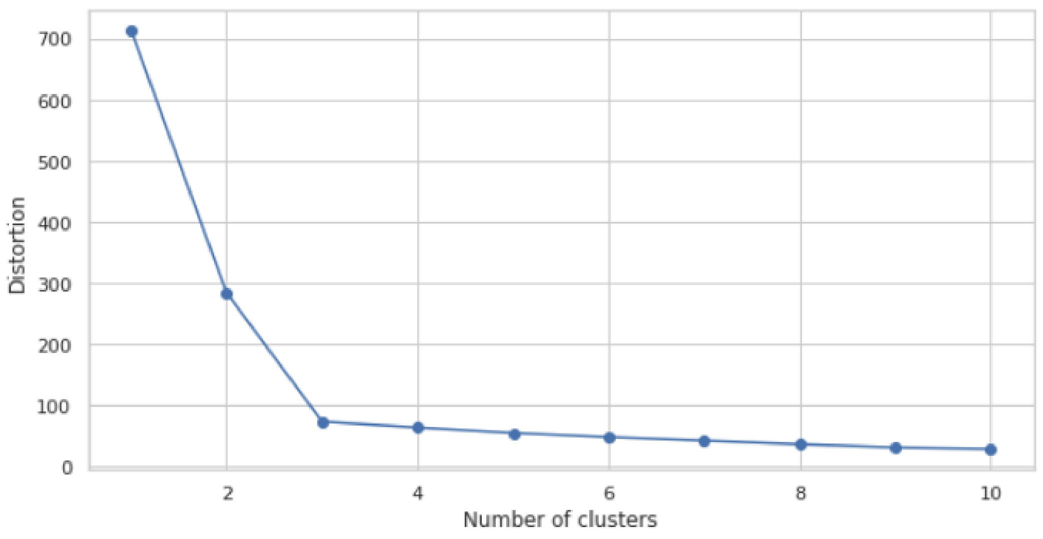

Figure 3: Output

3 Discussion & Conclusion

Based on the focused objective(s) to understand about clustering and k-means algorithm, the additional lab exercise made me more confident towards the fulfilment of the objectives(s).

4 Lab Exercise (Submit as a report)

Implement other categories of clustering algorithms, such as hierarchical and density-based clustering, that do not require us to specify the number of clusters upfront or assume spherical structures in our dataset.

5 Policy

Copying from internet, classmate, seniors, or from any other source is strongly prohibited. 100% marks will be *deducted* if any such copying is detected.