School of Computer Science Engineering and Technology
Assignment-07

Course- B.Tech	**Type-** Core
Course Code-	**Course Name-** Statistical Machine Learning
Year- 2023-2024	**Semester-** odd
Date– 11-09-2023	**Batch-** AIML-B, D

1 Implement K-Means Clustering using Synthetic Data

Part 1 – Import the required Python, Pandas, Matplotlib, Seaborn packages

Problem: you have a multidimensional set of data (such as a set of hidden unit activations) and you want to see which points are closest to others. The k-means algorithm searches for a pre-determined number of clusters within an unlabeled multidimensional dataset. It accomplishes this using a simple conception of what the optimal clustering looks like:

- The "cluster center" is the arithmetic mean of all the points belonging to the cluster.
- Each point is closer to its own cluster center than to other cluster centers.

Step 1 –Create synthetic dataset of unlabeled blobs

The dataset would be synthesized using `sklearn.datasets.samples_generator` from the `sklearn` package. You will import *binary large objects- blobs* to form clusters from the synthetic dataset.

Step 2 –Import KMeans class from Scikit-learn and fit the data

Verify the syntentic dataset and fit the data to the K-Means model.

Step -3: Visualize the fitted data by coloring the blobs by assigned label numbers

We will use the c argument in the `plt.scatter()` function. We will also try to make the cluster centers prominent.

How k-means is a special case of Expectation-maximization (EM) algorithm

Expectation–maximization (EM) is a powerful algorithm that comes up in a variety of contexts within data science. k-means is a particularly simple and special case of this more general algorithm. The basic algorithmic flow of k-means is to,

- Guess some cluster center (initialization)
- Repeat following steps untill converged,

- E-step: assign points to the nearest cluster center
- M-Step: set the cluster centers to the mean

Here the "E-step" or "Expectation step" involves updating our expectation of which cluster each point belongs to.

The "M-step" or "Maximization step" involves maximizing some fitness function that defines the location of the cluster centers. In the case of k-means, that maximization is accomplished by taking a simple mean of the data in each cluster.

Implementing k-means from scratch

By taking `from sklearn.metrics import pairwise_distances_argmin` we can implement the K-Means from scratch in 3 steps as mentioned earlier.

- Randomly choose the clusters
- Assign Labels based on closest cluster center
- Find new centers by computing the means of points close to the cluster center.
- Check for convergence point.
- Plot the centers and labels using scatter plot.

Code to Implement K-Means from Scratch

```
from sklearn.metrics import pairwise_distances_argmin
def find_clusters(X, n_clusters, rseed=2):
# 1. Randomly choose clusters
rng = np.random.RandomState(rseed)
i = rng.permutation(X.shape[0])[:n_clusters]
centers = X[i]

while True:
# 2a. Assign labels based on closest center
labels = pairwise_distances_argmin(X, centers)

# 2b. Find new centers from means of points
new_centers = np.array([X[labels == i].mean(0)
for i in range(n_clusters)])

# 2c. Check for convergence
if np.all(centers == new_centers):
break
centers = new_centers

return centers, labels
```

```
centers, labels = find_clusters(X, 4)
plt.scatter(X[:, 0], X[:, 1], c=labels,
s=50, cmap='viridis');
```

Although the E–M procedure is guaranteed to improve the result in each step, there is no assurance that it will lead to the global best solution. The initialization is important and particularly bad initialization can sometimes lead to clearly sub-optimal clustering.

How many Number of clusters?

A common challenge with k-means is that you must tell it how many clusters you expect. It cannot learn the number of clusters from the data.

If we force the k-means to look for 6 clusters instead of 4, it will come back with 6 but they may not be what we are looking for!

Some methods like **elbow and silhouette analysis** can be used to gauge a good number of clusters.

Limitation of K-Means Clustering

k-means algorithm will often be ineffective if the clusters have complicated geometries. In particular, the boundaries between k-means clusters will always be linear, which means that it will fail for more complicated boundaries. Verify the limitation using the code:

```
from sklearn.datasets import make_moons
X, y = make_moons(200, noise=.05, random_state=0)
labels = KMeans(2, random_state=0).fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=50, cmap='viridis');
```

Kernel Transformation Trick

The situation above is reminiscent of the Support Vector Machines, where we use a kernel transformation to project the data into a higher dimension where a linear separation is possible. We might imagine using the same trick to allow k-means to discover non-linear boundaries.

One version of this kernelized k-means is implemented in `Scikit-Learn` within the `SpectralClustering` estimator. It uses the graph of nearest neighbors to compute a higher-dimensional representation of the data, and then assigns labels using a k-means algorithm using the following code:

```
from sklearn.cluster import SpectralClustering
model = SpectralClustering(n_clusters=2, affinity='nearest_neighbors',
assign_labels='kmeans')
labels = model.fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels,s=50, cmap='viridis');
```

2. Implement Gaussian Mixture Model using Synthetic Dataset

Challenges in K-Means can be overcomed using:

- You could measure uncertainty in cluster assignment by comparing the distances of each point to all cluster centers, rather than focusing on just the closest.
- You might also imagine allowing the cluster boundaries to be ellipses rather than circles, so as to account for non-circular clusters.

A **Gaussian mixture model (GMM)** attempts to find a mixture of multi-dimensional Gaussian probability distributions that best model any input dataset. In the simplest case, GMMs can be used for finding clusters in the same manner as k-means.

However, because GMM contains a probabilistic model under the hood, it is also possible to find probabilistic cluster assignments. in Scikit-Learn this is done using the`predict_proba` method. This returns a matrix of size `[n_samples, n_clusters]` which measures the probability that any point belongs to the given cluster.

Step -1: Generate Synthetic Data using unlabeled blobs

```
from sklearn.datasets.samples_generator import make_blobs
X, y_true = make_blobs(n_samples=400, centers=4,
cluster_std=0.7, random_state=0)
X = X[:, ::-1] # flip axes for better plotting
```

Step-2: Visualize the uncertinity by making data point size proportional to probability

You have to use `predict_proba` and `probs_max` to predict the max probability for the Gaussian Mixture Model.

```
from matplotlib.patches import Ellipse

def draw_ellipse(position, covariance, ax=None, **kwargs):
"""Draw an ellipse with a given position and covariance"""
ax = ax or plt.gca()

# Convert covariance to principal axes
if covariance.shape == (2, 2):
U, s, Vt = np.linalg.svd(covariance)
angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
width, height = 2 * np.sqrt(s)
else:
angle = 0
width, height = 2 * np.sqrt(covariance)

# Draw the Ellipse
```

```
for nsig in range(1, 4):
ax.add_patch(Ellipse(position, nsig * width, nsig * height,
angle, **kwargs))

def plot_gmm(gmm, X, label=True, ax=None):
ax = ax or plt.gca()
labels = gmm.fit(X).predict(X)
if label:
ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis', zorder=2,edgecolor='k')
else:
ax.scatter(X[:, 0], X[:, 1], s=40, zorder=2,cmap='viridis',edgecolor='k')
ax.axis('equal')

w_factor = 0.2 / gmm.weights_.max()
for pos, covar, w in zip(gmm.means_, gmm.covariances_, gmm.weights_):
draw_ellipse(pos, covar, alpha=w * w_factor)

gmm = GaussianMixture(n_components=4, covariance_type='full', random_state=42)
plot_gmm(gmm, X_stretched)
```

Step-3: GMM as Density Estimation and Generative Model

Using `sklearn.datasets import make_moons` we can plot the data using scatter plot.