Kalinga University

Faculty of Computer Science and Information Technology

Program - BTCSAIML **Sem- 3rd**

Subject- Machine Learning **Subject Code: BTCS304**

Unit 3

K means Clustering – Introduction

K-Means Clustering groups similar data points into clusters without needing labeled data. It is used to uncover hidden patterns when the goal is to organize data based on similarity.

- Helps identify natural groupings in unlabeled datasets
- Works by grouping points based on distance to cluster centers
- Commonly used in customer segmentation, image compression and pattern discovery
- Useful when you need structure from raw, unorganized data

Working of K-Means Clustering

Suppose we are given a data set of items with certain features and values for these features like a vector. The task is to categorize those items into groups. To achieve this we will use the K-means algorithm. "*k*" represents the number of groups or clusters we want to classify our items into.

The algorithm will categorize the items into "*k*" groups or clusters of similarity. To calculate that similarity we will use the Euclidean distance as a measurement. The algorithm works as follows:

1. **Initialization:** We begin by randomly selecting k cluster centroids.
2. **Assignment Step:** Each data point is assigned to the nearest centroid, forming clusters.
3. **Update Step:** After the assignment, we recalculate the centroid of each cluster by averaging the points within it.
4. **Repeat:** This process repeats until the centroids no longer change or the maximum number of iterations is reached.

The goal is to partition the dataset into *k* clusters such that data points within each cluster are more similar to each other than to those in other clusters.

> *Selecting the right number of clusters is important for meaningful segmentation to do this we use Elbow Method for optimal value of k in KMeans which is a graphical tool used to determine the optimal number of clusters (k) in K-means.*

Mathematical Formulation

1. Euclidean Distance

The distance between a data point xxx and a centroid ccc is calculated as:

$$d(x, c) = \sqrt{\sum_{i=1}^{n} (x_i - c_i)^2}$$

where:

- xi*xi* is the ith*ith*feature of the data point.
- ci*ci* is the ith*ith*feature of the centroid.
- n is the number of features.

2. Centroid Update

After assigning all data points to clusters, each centroid is updated as the mean of all points belonging to that cluster:

$$c_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i$$

where:

- Cj is the set of points assigned to thejth*jth* cluster.
- ∣Cj∣ is the number of points in that cluster.

Uses of K-Means Clustering

K-Means is popular in a wide variety of applications due to its simplicity, efficiency and effectiveness. Here's why it is widely used:

1. **Data Segmentation:** One of the most common uses of K-Means is segmenting data into distinct groups. For example, businesses use K-Means to group customers based on behavior, such as purchasing patterns or website interaction.
2. **Image Compression**: K-Means can be used to reduce the complexity of images by grouping similar pixels into clusters, effectively compressing the image. This is useful for image storage and processing.
3. **Anomaly Detection:** K-Means can be applied to detect anomalies or outliers by identifying data points that do not belong to any of the clusters.
4. **Document Clustering:** In natural language processing (NLP), K-Means is used to group similar documents or articles together. It's often used in applications like recommendation systems or news categorization.
5. **Organizing Large Datasets:** When dealing with large datasets, K-Means can help in organizing the data into smaller, more manageable chunks based on similarities, improving the efficiency of data analysis.

Implementation of K-Means Clustering

We will be using blobs datasets and show how clusters are made using Python programming language.

Step 1: Importing the necessary libraries

We will be importing the following libraries.

- Numpy**:** for numerical operations (e.g., distance calculation).
- Matplotlib: for plotting data and results.
- Scikit learn**:** to create a synthetic dataset using **make_blobs**

```
import numpy as npimport matplotlib.pyplot as pltfrom sklearn.datasets import make_blobs
```

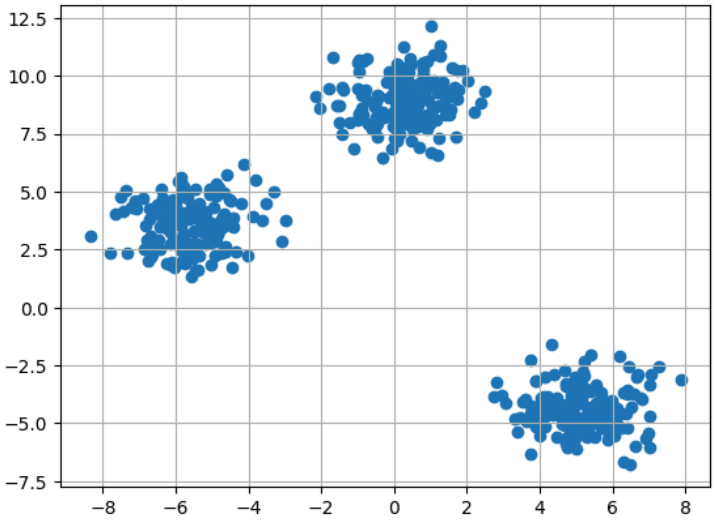
Step 2: Creating Custom Dataset

We will generate a synthetic dataset with make_blobs.

- **make_blobs(n_samples=500, n_features=2, centers=3):** Generates 500 data points in a 2D space, grouped into 3 clusters.
- **plt.scatter(X[:, 0], X[:, 1]):** Plots the dataset in 2D, showing all the points.
- **plt.show():** Displays the plot

```
X,y = make_blobs(n_samples = 500,n_features = 2,centers = 3,random_state = 23)
fig = plt.figure(0)plt.grid(True)plt.scatter(X[:,0],X[:,1])plt.show()
```

Output:



Clustering dataset

Step 3 :Feature Scaling using StandardScaler

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X = scaler.fit_transform(X)
```

Step 4: Initializing Random Centroids

We will randomly initialize the centroids for K-Means clustering

- **np.random.seed(23):** Ensures reproducibility by fixing the random seed.
- The for loop initializes k random centroids, with values between -2 and 2, for a 2D dataset.

```
k = 3
clusters = {}np.random.seed(23)
for idx in range(k):
    center = 2*(2*np.random.random((X.shape[1],))-1)
    points = []
    cluster = {
        'center' : center,
        'points' : []
    }

    clusters[idx] = cluster
    clusters
```

Output:

```
{0: {'center': array([0.06919154, 1.78785042]), 'points': []},
 1: {'center': array([ 1.06183904, -0.87041662]), 'points': []},
 2: {'center': array([-1.11581855,  0.74488834]), 'points': []}}
```

Random Centroids

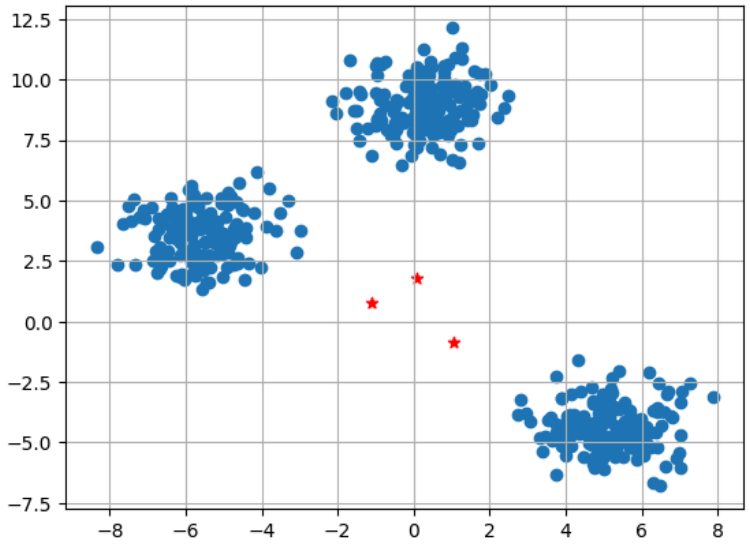
Step 5: Plotting Random Initialized Center with Data Points

We will now plot the data points and the initial centroids.

- **plt.grid()**: Plots a grid.
- **plt.scatter(center[0], center[1], marker='*', c='red'):** Plots the cluster center as a red star (* marker).

```
plt.scatter(X[:,0],X[:,1])plt.grid(True)for i in clusters:
    center = clusters[i]['center']
    plt.scatter(center[0],center[1],marker = '*',c = 'red')plt.show()
```

Output:



Data points with random center

Step 6: Defining Euclidean Distance

To assign data points to the nearest centroid, we define a distance function:

- **np.sqrt():** Computes the square root of a number or array element-wise.
- **np.sum():** Sums all elements in an array or along a specified axis

```
def distance(p1,p2):
    return np.sqrt(np.sum((p1-p2)**2))
```

Step 7: Creating Assign and Update Functions

Next, we define functions to assign points to the nearest centroid and update the centroids based on the average of the points assigned to each cluster.

- **dist.append(dis):** Appends the calculated distance to the list dist.
- **curr_cluster = np.argmin(dist):** Finds the index of the closest cluster by selecting the minimum distance.
- **new_center = points.mean(axis=0):** Calculates the new centroid by taking the mean of the points in the cluster.

```
def assign_clusters(X, clusters):
    for idx in range(X.shape[0]):
        dist = []

        curr_x = X[idx]

        for i in range(k):
            dis = distance(curr_x,clusters[i]['center'])
            dist.append(dis)
        curr_cluster = np.argmin(dist)
        clusters[curr_cluster]['points'].append(curr_x)
    return clusters
```

```
def update_clusters(X, clusters):
    for i in range(k):
        points = np.array(clusters[i]['points'])
        if points.shape[0] > 0:
            new_center = points.mean(axis =0)
            clusters[i]['center'] = new_center

            clusters[i]['points'] = []
    return clusters
```

Step 8: Predicting the Cluster for the Data Points

We create a function to predict the cluster for each data point based on the final centroids.

- **pred.append(np.argmin(dist)):** Appends the index of the closest cluster (the one with the minimum distance) to pred.

```
def pred_cluster(X, clusters):
    pred = []
    for i in range(X.shape[0]):
        dist = []
        for j in range(k):
            dist.append(distance(X[i],clusters[j]['center']))
        pred.append(np.argmin(dist))
    return pred
```

Step 9: Assigning, Updating and Predicting the Cluster Centers

The assign and update steps are repeated multiple times until the cluster centers stabilize or a maximum number of iterations is reached.

- **old_centers:** Stores the current centroids before updating them.
- **assign_clusters(X, clusters):** Assigns each data point to its nearest centroid.
- **update_clusters(X, clusters):** Recalculates each centroid as the mean of the points assigned to its cluster.
- **np.allclose(old_centers, new_centers):** Checks whether the centroids have stabilized. If they have, the algorithm stops early.
- **pred_cluster(X, clusters):** Predicts the final cluster label for each data point using the converged centroids.

```
# Repeat until convergence or maximum iterationsmax_iters = 100
for _ in range(max_iters):
    old_centers = [clusters[i]['center'].copy() for i in range(k)]

    clusters = assign_clusters(X, clusters)
    clusters = update_clusters(X, clusters)

    new_centers = [clusters[i]['center'] for i in range(k)]

    if np.allclose(old_centers, new_centers):
```

```
        break
pred = pred_cluster(X, clusters)
```

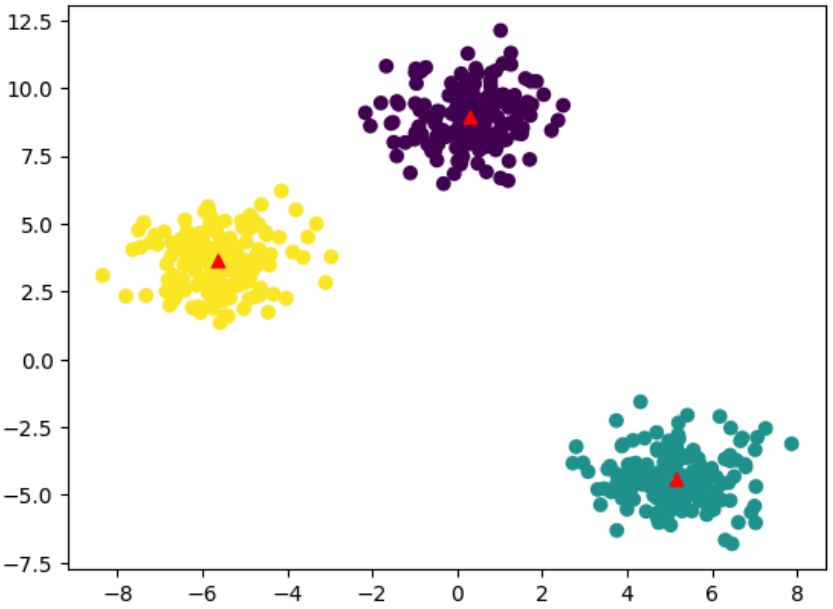
Step 10: Plotting Data Points with Predicted Cluster Centers

Finally, we plot the data points, colored by their predicted clusters, along with the updated centroids.

- **center = clusters[i]['center']:** Retrieves the center (centroid) of the current cluster.
- **plt.scatter(center[0], center[1], marker='^', c='red'):** Plots the cluster center as a red triangle (^ marker).

```
plt.scatter(X[:,0],X[:,1],c = pred)for i in clusters:
    center = clusters[i]['center']
    plt.scatter(center[0],center[1],marker = '^',c = 'red')plt.show()
```

Output:



K-means Clustering

You can download the source code from here.

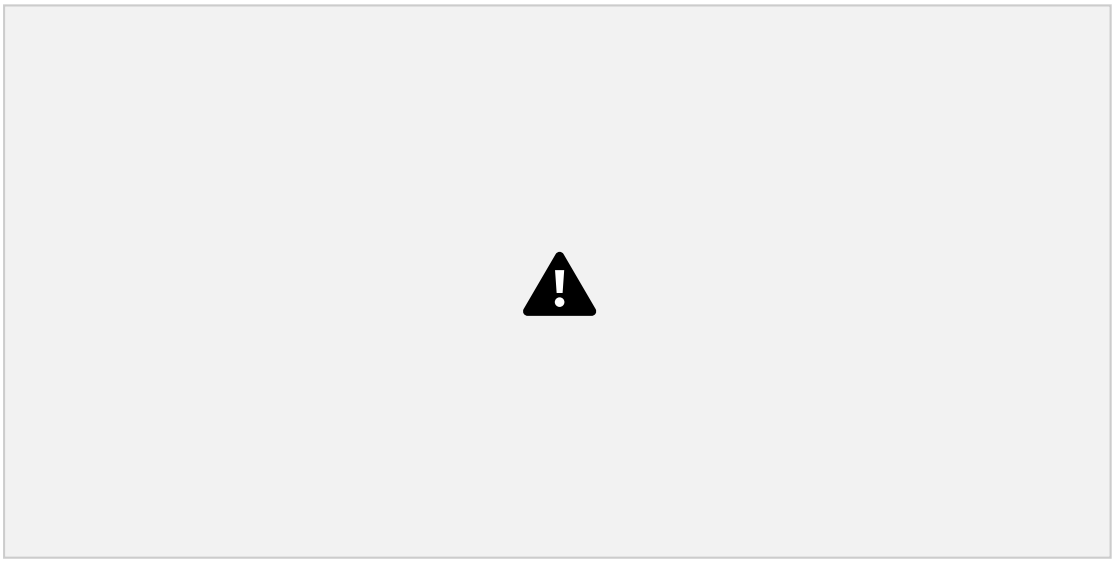
Challenges

- **Choosing the Right Number of Clusters** (k*k*): One of the biggest challenges is deciding how many clusters to use.
- **Sensitive to Initial Centroids:** The final clusters can vary depending on the initial random placement of centroids.
- **Non-Spherical Clusters:** K-Means assumes that the clusters are spherical and equally sized. This can be a problem when the actual clusters in the data are of different shapes or densities.
- **Outliers**: K-Means is sensitive to outliers, which can distort the centroid and, ultimately, the clusters.

Elbow Method for optimal value of k in KMeans

The Elbow Method is used to find the optimal number of clusters (k) in K-Means by analyzing how the clustering performance changes with different k values.

- Plots WCSS (Within-Cluster Sum of Squares) against different k values
- WCSS decreases as k increases, showing better fit
- The “elbow point” indicates the optimal k where improvement slows down



Elbow Method

Working of Elbow Point

The Elbow Method works in the following steps:

1. We begin by selecting a range of k values (for example, 1 to 10).

2. For each k, we run K-Means and calculate WCSS (Within-Cluster Sum of Squares), which shows how close the data points are to their cluster centroids:

$$\text{WCSS} = \sum_{i=1}^{k} \sum_{j=1}^{n_i} \text{distance}(x_j^{(i)}, c_i)^2$$

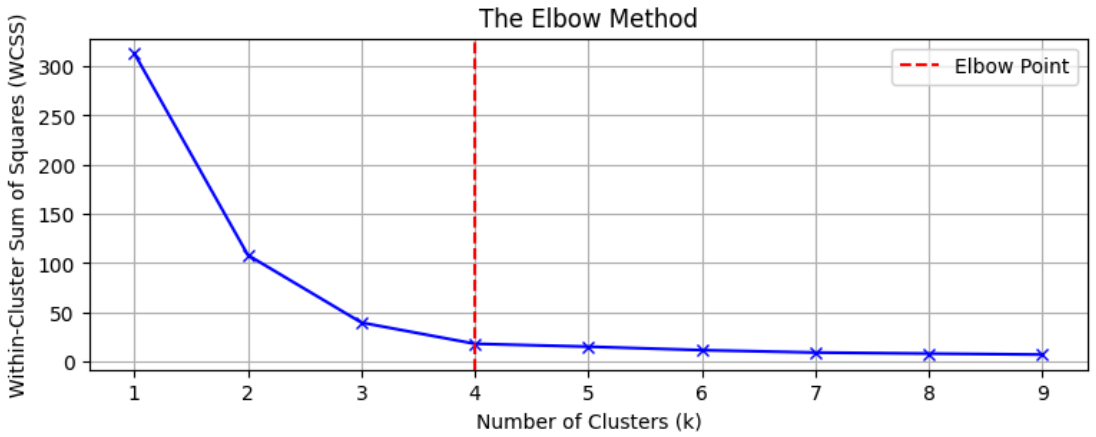
Where $\text{distance}(x_j^{(i)}, c_i)$ represents the distance between the j^{th} data point $x_j^{(i)}$ in cluster i and the centroid c_i of that cluster.

3. After computing WCSS for all k values, we plot k vs WCSS.

4. WCSS always decreases as k increases because more clusters reduce the internal spread.

5. However, after a certain point, the improvement becomes very small. This bend or “elbow” in the curve indicates the point where adding more clusters no longer gives meaningful improvement.

- **Before the elbow:** WCSS drops quickly -> clusters become much better.
- **After the elbow:** WCSS drops slowly -> extra clusters add little value and may lead to overfitting.



Elbow Point

The goal is to identify the point where the rate of decrease in WCSS sharply changes, indicating that adding more clusters (beyond this point) yields diminishing returns. This "elbow" point suggests the optimal number of clusters.

Understanding Distortion and Inertia in K-Means Clustering

Two metrics commonly used in the Elbow Method are Distortion and Inertia.

1. Distortion

Distortion measures the average squared distance between each data point and its assigned cluster center. It's a measure of how well the clusters represent the data. A lower distortion value indicates better clustering.

$$\text{Distortion} = \frac{1}{n}\sum_{i=1}^{n} \min_{c \in \text{clusters}} \|x_i - c\|^2$$

where,

- Xi is the i^{th} data point
- c*c* is a cluster center from the set of all cluster centroids
- // xi−c // 2 // *xi−c* // 2 is the squared Euclidean distance between the data point and the cluster center
- n*n* is the total number of data points

2. Inertia

Inertia is the sum of squared distances of each data point to its closest cluster center. It's essentially the total squared error of the clustering. Like distortion, a lower inertia value suggests better clustering.

$$\text{Inertia} = \sum_{i=1}^{n} \text{distance}(x_i, c_j^*)^2$$

In the Elbow Method, we compute distortion or inertia for different k values and plot them. The point where the decrease begins to slow the “elbow” usually indicates the optimal number of clusters.

Implementation of Elbow Method

Let's implement the Elbow method,

Step 1: Importing the required libraries

We will import numpy, matplotlib, scikit learn and scipy for this.

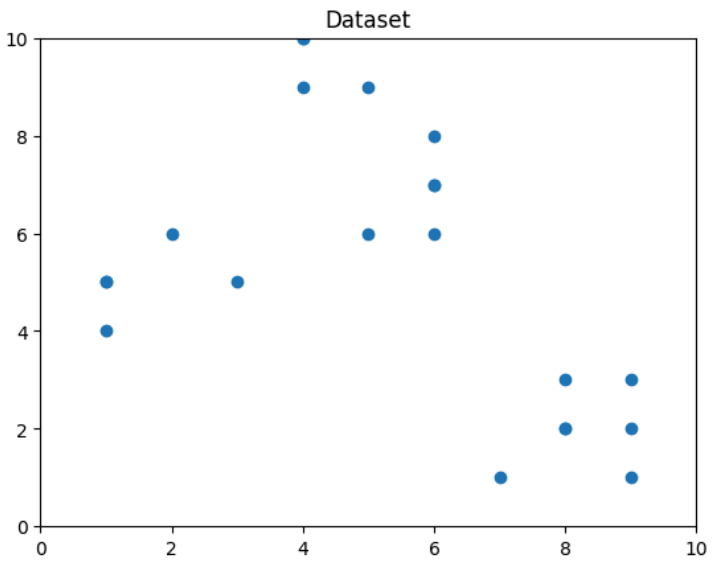
```
from sklearn.cluster import KMeansfrom sklearn import metricsfrom scipy.spatial.distance import cdistimport numpy as npimport matplotlib.pyplot as plt
```

Step 2: Creating and Visualizing the data

We will create a random array and visualize its distribution

```
x1 = np.array([3, 1, 1, 2, 1, 6, 6, 6, 5, 6,
               7, 8, 9, 8, 9, 9, 8, 4, 4, 5, 4])x2 = np.array([5, 4, 5, 6, 5, 8, 6, 7, 6, 7,
               1, 2, 1, 2, 3, 2, 3, 9, 10, 9, 10])X = np.array(list(zip(x1,
x2))).reshape(len(x1), 2)
plt.scatter(x1, x2, marker='o')plt.xlim([0, 10])plt.ylim([0, 10])plt.title('Dataset Visualization')plt.xlabel('Feature 1')plt.ylabel('Feature 2')plt.show()
```

Output:



From the above visualization, we can see that the optimal number of clusters should be around 3. But visualizing the data alone cannot always give the right answer. Hence we demonstrate the following steps.

Step 3: Building the Clustering Model and Calculating Distortion and Inertia

In this step, we will fit the K-means model for different values of k (number of clusters) and calculate both the distortion and inertia for each value.

```
distortions = []inertias = []mapping1 = {}mapping2 = {}K = range(1, 10)
for k in K:
    kmeanModel = KMeans(n_clusters=k, random_state=42).fit(X)

    distortions.append(sum(np.min(cdist(X, kmeanModel.cluster_centers_, 'euclidean'),
axis=1)**2) / X.shape[0])

    inertias.append(kmeanModel.inertia_)

    mapping1[k] = distortions[-1]
    mapping2[k] = inertias[-1]
```

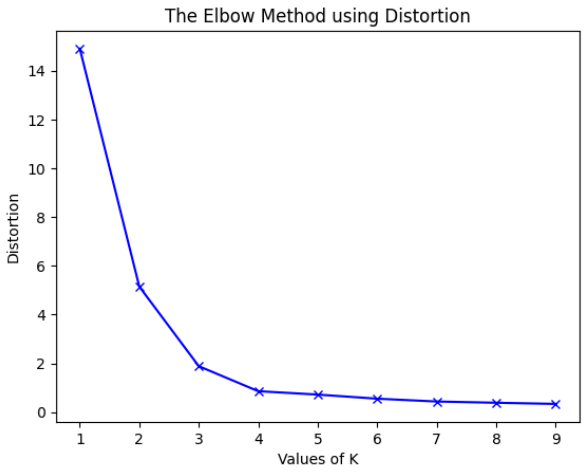
Step 4: Tabulating and Visualizing the Results

a) Displaying Distortion Values

```
print("Distortion values:")for key, val in mapping1.items():
    print(f'{key} : {val}')
plt.plot(K, distortions, 'bx-')plt.xlabel('Number of Clusters
(k)')plt.ylabel('Distortion')plt.title('The Elbow Method using Distortion')plt.show()
```

Output:

Distortion values:
1 : 14.90249433106576
2 : 5.146258503401359
3 : 1.8817838246409675
4 : 0.856122448979592
5 : 0.7166666666666667
6 : 0.5484126984126984
7 : 0.4325396825396825
8 : 0.3817460317460318
9 : 0.3341269841269841

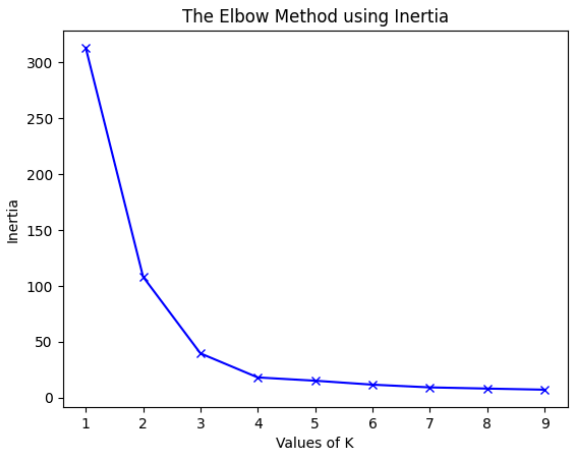


Plotting Distortion Values

b) Displaying Inertia Values:

```
print("Inertia values:")for key, val in mapping2.items():
    print(f'{key} : {val}')
plt.plot(K, inertias, 'bx-')plt.xlabel('Number of Clusters
(k)')plt.ylabel('Inertia')plt.title('The Elbow Method using Inertia')plt.show()
```

Output:

Inertia values: *1 : 312.95238095238096* *2 : 108.07142857142854* *3 : 39.51746031746032* *4 : 17.978571428571428* *5 : 15.049999999999997* *6 : 11.516666666666666* *7 : 9.083333333333334* *8 : 8.016666666666667* *9 : 7.0166666666666675*	

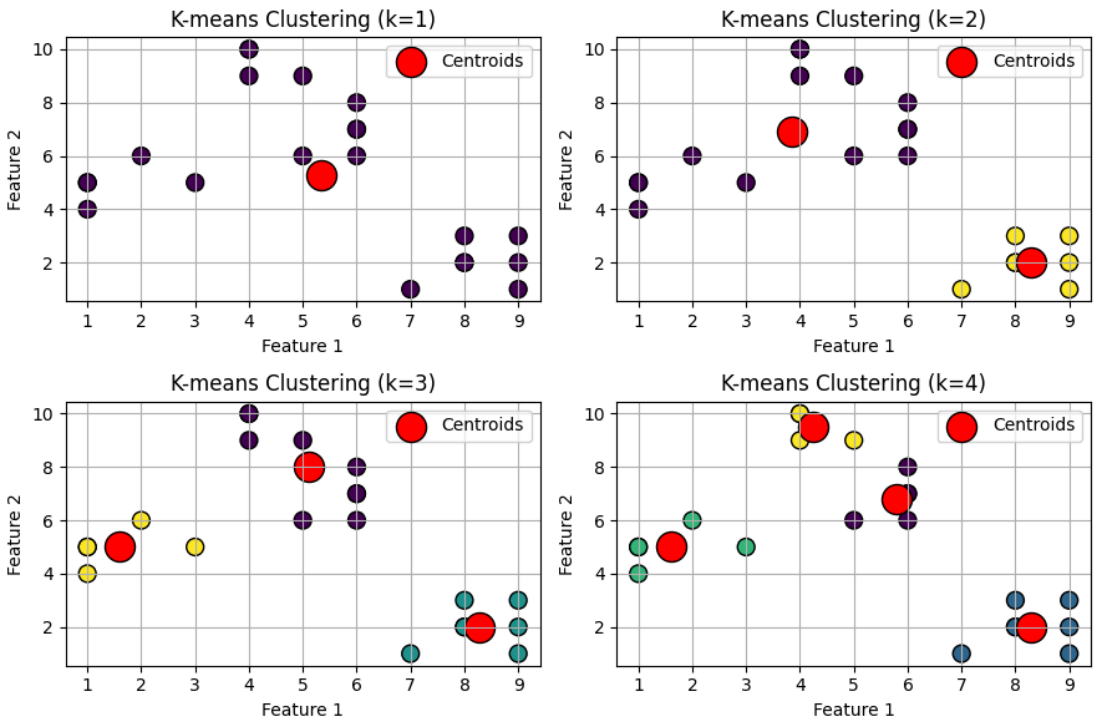
Step 5: Clustered Data Points For Different k Values

We will plot images of data points clustered for different values of k. For this, we will apply the k-means algorithm on the dataset by iterating on a range of k values.

```
k_range = range(1, 5)
for k in k_range:
    kmeans = KMeans(n_clusters=k, init='k-means++', random_state=42)
    y_kmeans = kmeans.fit_predict(X)

    plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, cmap='viridis', marker='o', edgecolor='k',
s=100)
    plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
                s=300, c='red', label='Centroids', edgecolor='k')
    plt.title(f'K-means Clustering (k={k})')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.legend()
    plt.grid()
    plt.show()
```

Output:

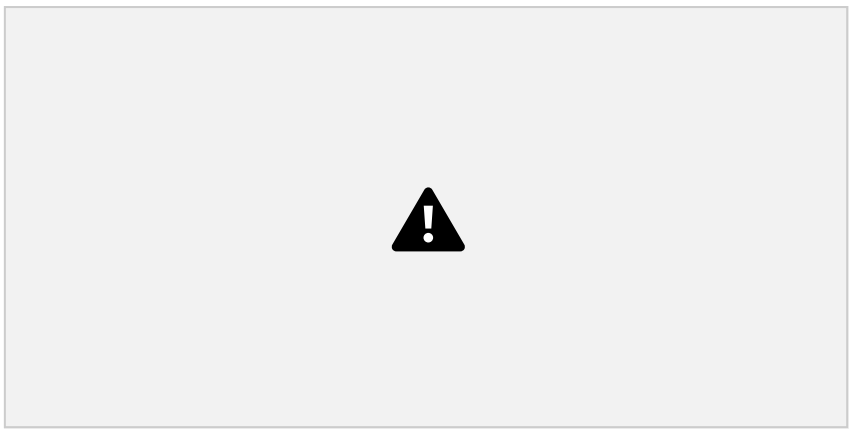


Visualizing Clustered Data Points

Hierarchical Clustering in Machine Learning

Hierarchical Clustering is an unsupervised learning technique that groups data into a hierarchy of clusters based on similarity. It builds a tree-like structure (dendrogram) that helps visualize relationships and decide the optimal number of clusters.

- Does not require pre-selecting the number of clusters.
- Uses agglomerative or divisive approaches.
- Commonly applied in data exploration and pattern discovery.
- It is commonly used in pattern recognition, customer segmentation and image grouping.



Imagine we have four fruits with different weights: an apple (100g), a banana (120g), a cherry (50g) and a grape (30g). Hierarchical clustering starts by treating each fruit as its own group.

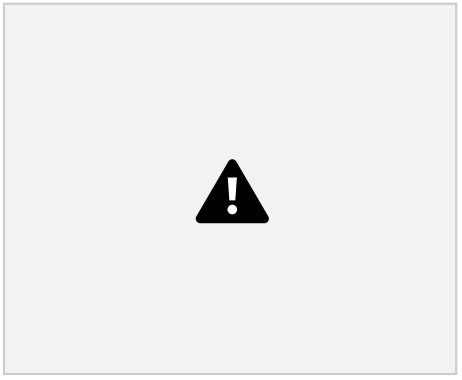
- Start with each fruit as its own cluster.
- Merge the closest items: grape (30g) and cherry (50g) are grouped first.
- Next, apple (100g) and banana (120g) are grouped.
- Finally, these two clusters merge into one.

Finally all the fruits are merged into one large group, showing how hierarchical clustering progressively combines the most similar data points.

Dendrogram

A dendrogram is like a family tree for clusters. It shows how individual data points or groups of data merge together. The bottom shows each data point as its own group and as we move up, similar groups are combined.

The lower the merge point, the more similar the groups are. It helps us see how things are grouped step by step.



Dendrogram

- At the bottom of the dendrogram the points P, Q, R, S and T are all separate.
- As we move up, the closest points are merged into a single group.
- The lines connecting the points show how they are progressively merged based on similarity.

- The height at which they are connected shows how similar the points are to each other; the shorter the line the more similar they are

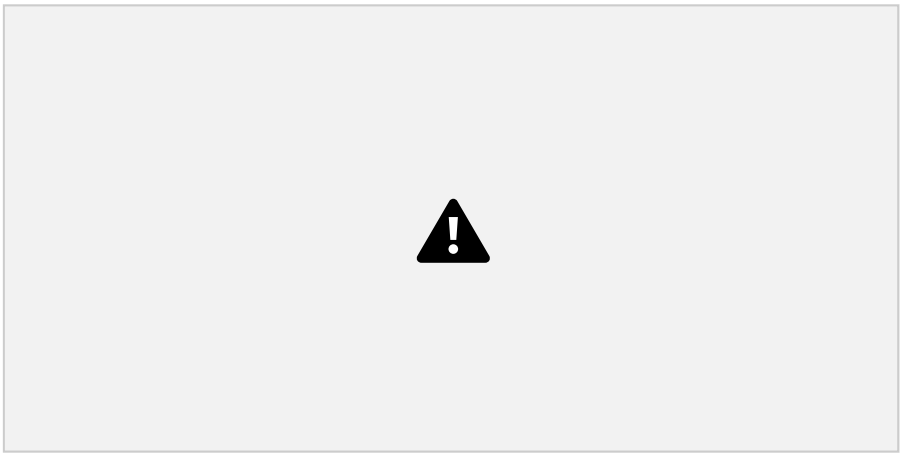
Types of Hierarchical Clustering

Now we understand the basics of hierarchical clustering. There are two main types of hierarchical clustering.

1. Agglomerative Clustering
2. Divisive clustering

1. Hierarchical Agglomerative Clustering

It is also known as the bottom-up approach or hierarchical agglomerative clustering (HAC). Bottom-up algorithms treat each data as a singleton cluster at the outset and then successively agglomerate pairs of clusters until all clusters have been merged into a single cluster that contains all data.



Workflow for Hierarchical Agglomerative clustering

1. **Start with individual points:** Each data point is its own cluster. For example if we have 5 data points we start with 5 clusters each containing just one data point.
2. **Calculate distances between clusters**: Calculate the distance between every pair of clusters. Initially since each cluster has one point this is the distance between the two data points.
3. **Merge the closest clusters**: Identify the two clusters with the smallest distance and merge them into a single cluster.
4. **Update distance matrix**: After merging we now have one less cluster. Recalculate the distances between the new cluster and the remaining clusters.
5. **Repeat steps 3 and 4**: Keep merging the closest clusters and updating the distance matrix until we have only one cluster left.
6. **Create a dendrogram**: As the process continues we can visualize the merging of clusters using a tree-like diagram called a dendrogram. It shows the hierarchy of how clusters are merged.

Implementation

Let's see the implementation of Agglomerative Clustering,

- Start with each data point as its own cluster.
- Compute distances between all clusters.
- Merge the two closest clusters based on a linkage method.
- Update the distances to reflect the new cluster.
- Repeat merging until the desired number of clusters or one cluster remains.
- The dendrogram visualizes these merges as a tree, showing cluster relationships and distances.

```
import numpy as npimport matplotlib.pyplot as pltfrom sklearn.cluster import AgglomerativeClusteringfrom scipy.cluster.hierarchy import dendrogramfrom sklearn.datasets import make_blobs
X, _ = make_blobs(n_samples=30, centers=3, cluster_std=10, random_state=42)
clustering = AgglomerativeClustering(n_clusters=3)labels = clustering.fit_predict(X)
agg = AgglomerativeClustering(
    distance_threshold=0,
    n_clusters=None,
    compute_distances=True)agg.fit(X)

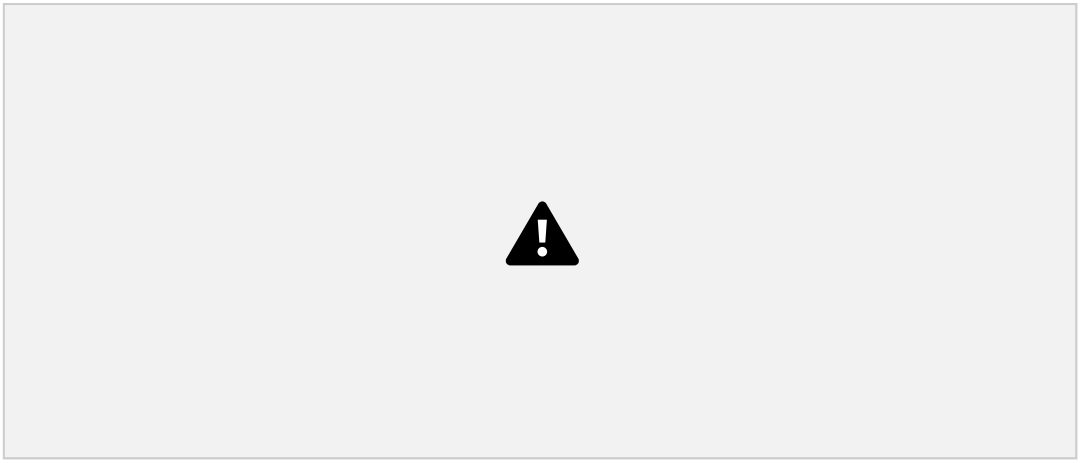
def plot_dendrogram(model, **kwargs):
    counts = np.zeros(model.children_.shape[0])
    n_samples = len(model.labels_)

    for i, merge in enumerate(model.children_):
        current_count = 0
        for child_idx in merge:
            if child_idx < n_samples:
                current_count += 1
            else:
                current_count += counts[child_idx - n_samples]
        counts[i] = current_count

    linkage_matrix = np.column_stack(
        [model.children_, model.distances_, counts]).astype(float)
    dendrogram(linkage_matrix, **kwargs)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))
ax1.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis',
s=70)ax1.set_title("Agglomerative Clustering")ax1.set_xlabel("Feature 1")ax1.set_ylabel("Feature 2")
plt.sca(ax2)plot_dendrogram(agg, truncate_mode='level', p=5)plt.title("Hierarchical Clustering Dendrogram")plt.xlabel("Sample index")plt.ylabel("Distance")
plt.tight_layout()plt.show()
```

Output :



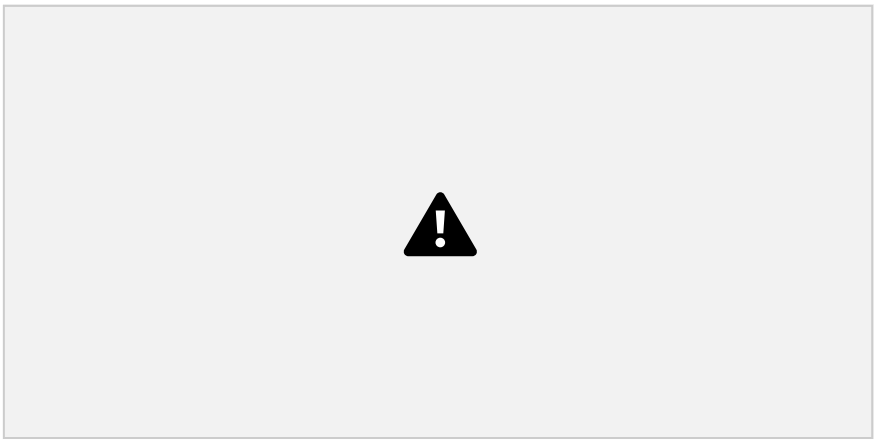
Agglomerative Clustering

2. Hierarchical Divisive clustering

Divisive clustering is also known as a top-down approach. Top-down clustering requires a method for splitting a cluster that contains the whole data and proceeds by splitting clusters recursively until individual data have been split into singleton clusters.

Workflow for Hierarchical Divisive clustering

1. **Start with all data points in one cluster:** Treat the entire dataset as a single large cluster.
2. **Split the cluster**: Divide the cluster into two smaller clusters. The division is typically done by finding the two most dissimilar points in the cluster and using them to separate the data into two parts.
3. **Repeat the process**: For each of the new clusters, repeat the splitting process: Choose the cluster with the most dissimilar points and split it again into two smaller clusters.
4. **Stop when each data point is in its own cluster**: Continue this process until every data point is its own cluster or the stopping condition (such as a predefined number of clusters) is met.



Implementation

Let's see the implementation of Divisive Clustering,

- Starts with all data points as one big cluster.
- Finds the largest cluster and splits it into two using KMeans.
- Repeats splitting the largest cluster until reaching the desired number of clusters.
- Assigns cluster labels to each data point based on the splits.
- Returns history of clusters at each step and final labels.
- Visualizes data points colored by their final cluster.

```
import numpy as npimport matplotlib.pyplot as pltfrom sklearn.cluster import
KMeansfrom sklearn.datasets import make_blobsfrom scipy.cluster.hierarchy
import dendrogram, linkage
X, _ = make_blobs(n_samples=30, centers=5, cluster_std=10, random_state=42)

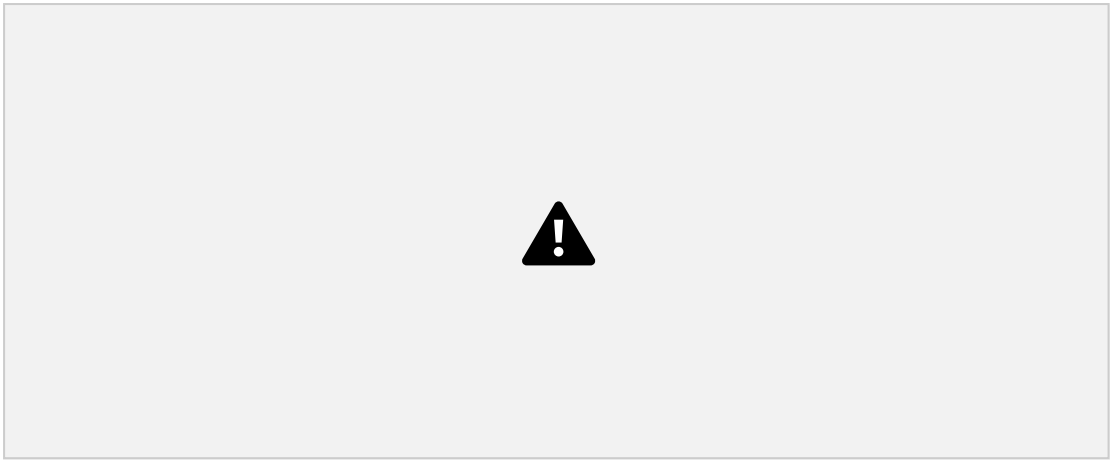
def divisive_clustering(data, max_clusters=3):
    clusters = [data]
    while len(clusters) < max_clusters:
        cluster_to_split = max(clusters, key=lambda x: len(x))
        clusters.remove(cluster_to_split)

        kmeans = KMeans(n_clusters=2, random_state=42).fit(cluster_to_split)
        cluster1 = cluster_to_split[kmeans.labels_ == 0]
        cluster2 = cluster_to_split[kmeans.labels_ == 1]

        clusters.extend([cluster1, cluster2])
    return clusters

clusters = divisive_clustering(X, max_clusters=3)
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)colors = ['r', 'g', 'b', 'c', 'm', 'y']for i, cluster in enumerate(clusters):
    plt.scatter(cluster[:, 0], cluster[:, 1], s=50,
                c=colors[i], label=f'Cluster {i+1}')plt.title('Divisive Clustering
Result')plt.legend()
linked = linkage(X, method='ward')
plt.subplot(1, 2, 2)dendrogram(linked, orientation='top',
           distance_sort='descending', show_leaf_counts=True)plt.title('Hierarchical
Clustering Dendrogram')
plt.tight_layout()plt.show()
```

Output:



Divisive Clustering

Computing Distance Matrix

While merging two clusters we check the distance between two every pair of clusters and merge the pair with the least distance/most similarity. But the question is how is that distance determined. There are different ways of defining Inter Cluster distance/similarity. Some of them are:

- **Min Distance**: Find the minimum distance between any two points of the cluster.
- **Max Distance:** Find the maximum distance between any two points of the cluster.
- **Group Average**: Find the average distance between every two points of the clusters.
- **Ward's Method**: The similarity of two clusters is based on the increase in squared error when two clusters are merged.



Distance Matrix

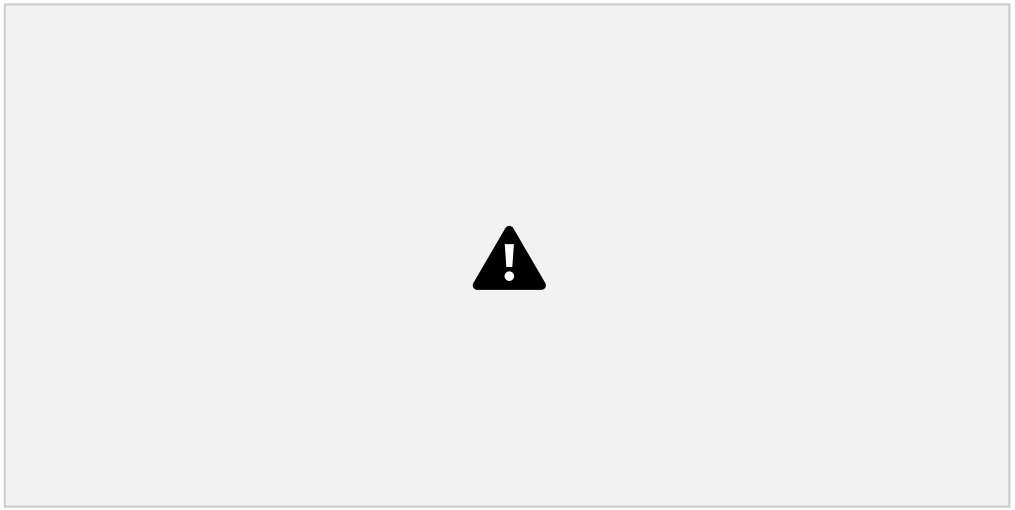
The image compares cluster distance methods:

- Min uses the shortest distance between clusters
- Max uses the longest
- Group Average computes the mean of all pairwise distances
- Ward’s method minimizes the increase in within-cluster variance during merging

DBSCAN Clustering in ML - Density based clustering

DBSCAN is a density-based clustering algorithm that groups data points that are closely packed together and marks outliers as noise based on their density in the feature space. It identifies clusters as dense regions in the data space separated by areas of lower density. Unlike K-Means or hierarchical clustering which assumes clusters are compact and spherical, DBSCAN perform well in handling real-world data irregularities such as:

- **Arbitrary-Shaped Clusters**: Clusters can take any shape not just circular or convex.
- **Noise and Outliers**: It effectively identifies and handles noise points without assigning them to any cluster.



Data Sets with clustering algorithms

The figure above shows a data set with clustering algorithms: K-Means and Hierarchical handling compact, spherical clusters with varying noise tolerance while DBSCAN manages arbitrary-shaped clusters and noise handling.

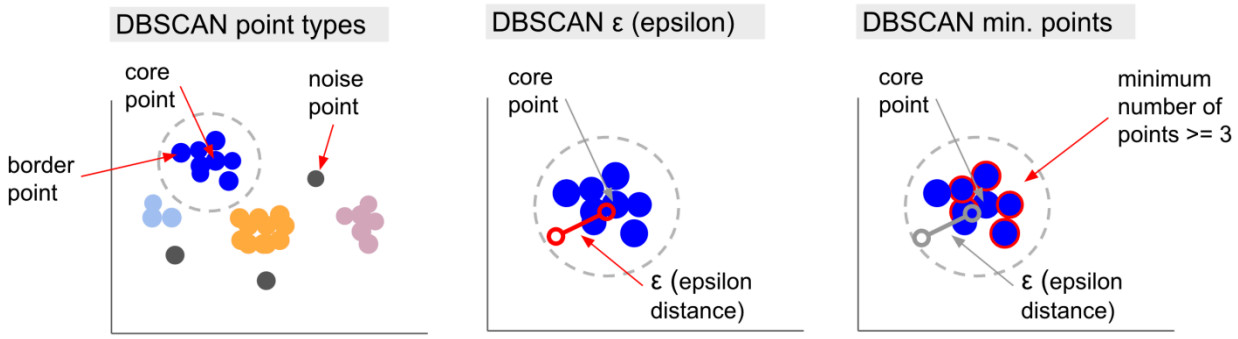
Key Parameters in DBSCAN

1. eps: This defines the radius of the neighborhood around a data point. If the distance between two points is less than or equal to eps they are considered neighbors. A common method to determine eps is by analyzing the k-distance graph. Choosing the right eps is important:

- If eps is too small most points will be classified as noise.
- If eps is too large clusters may merge and the algorithm may fail to distinguish between them.

2. MinPts: This is the minimum number of points required within the **eps** radius to form a dense region. A general rule of thumb is to set MinPts >= D+1 where **D** is the number of dimensions in the dataset.

*For most cases a minimum value of **MinPts = 3** is recommended.*

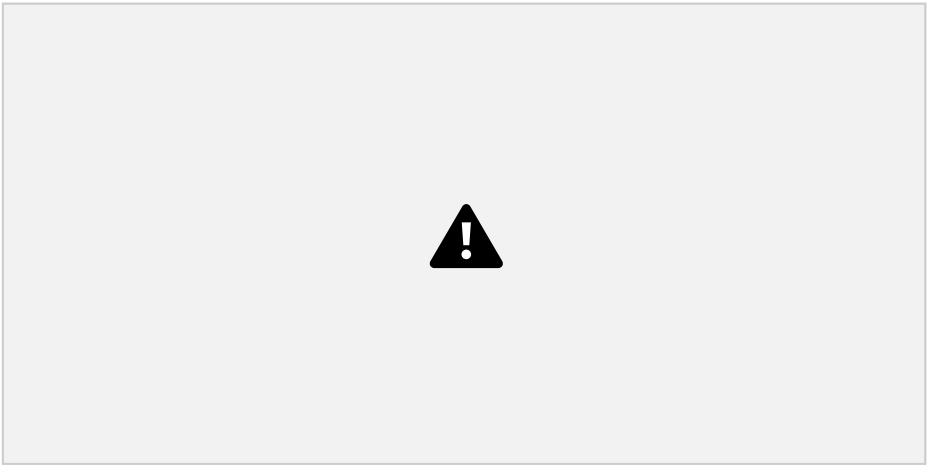


How Does DBSCAN Work

DBSCAN works by categorizing data points into three types:

1. Core points which have a sufficient number of neighbors within a specified radius (eplison)
2. Border points which are near core points but lack enough neighbors to be core points themselves
3. Noise points which do not belong to any cluster.

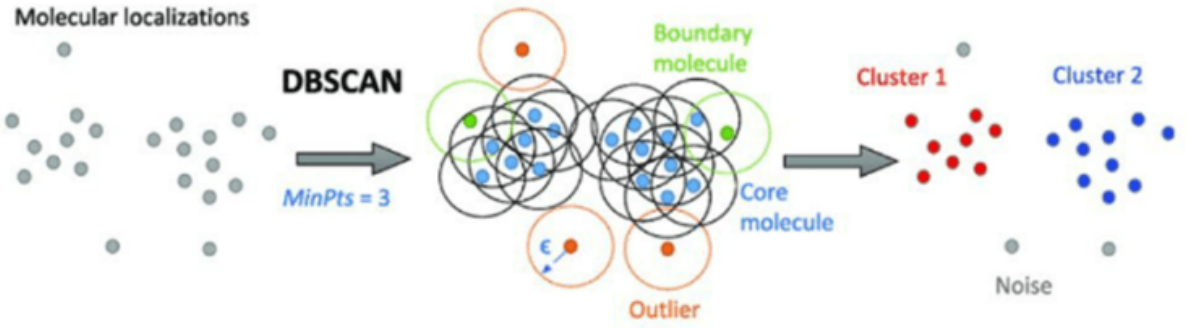
By iteratively expanding clusters from core points and connecting density-reachable points, DBSCAN forms clusters without relying on rigid assumptions about their shape or size.



DBSCAN ALgorithm

Steps in the DBSCAN Algorithm

1. **Identify Core Points**: For each point in the dataset count the number of points within its eps neighborhood. If the count meets or exceeds MinPts mark the point as a core point.
2. **Form Clusters**: For each core point that is not already assigned to a cluster create a new cluster. Recursively find all density-connected points i.e points within the eps radius of the core point and add them to the cluster.
3. **Density Connectivity**: Two points a and b are density-connected if there exists a chain of points where each point is within the eps radius of the next and at least one point in the chain is a core point. This chaining process ensures that all points in a cluster are connected through a series of dense regions.
4. **Label Noise Points**: After processing all points any point that does not belong to a cluster is labeled as noise.



Implementation of DBSCAN Algorithm In Python

Here we'll use the Python library sklearn to compute DBSCAN and matplotlib.pyplot library for visualizing clusters.

Step 1: Importing Libraries

We import all the necessary library like numpy , matplotlib and scikit-learn.

```
import matplotlib.pyplot as pltimport numpy as npfrom sklearn.cluster import DBSCANfrom sklearn import metricsfrom sklearn.datasets import make_blobsfrom sklearn.preprocessing import StandardScalerfrom sklearn import datasetsfrom sklearn.metrics import adjusted_rand_score
```

Step 2: Preparing Dataset

We will create a dataset of 4 clusters using make_blob. The dataset have 300 points that are grouped into 4 visible clusters.

```
X, y_true = make_blobs(n_samples=300, centers=4,
                       cluster_std=0.50, random_state=0)
```

Step 3: Applying DBSCAN Clustering

Now we apply DBSCAN clustering on our data, count it and visualize it using the matplotlib library.

- **eps=0.3:** The radius to look for neighboring points.
- **min_samples:** Minimum number of points required to form a dense region a cluster.
- **labels:** Cluster numbers for each point. -1 means the point is considered noise.

```
db = DBSCAN(eps=0.3, min_samples=10).fit(X)core_samples_mask =
np.zeros_like(db.labels_, dtype=bool)core_samples_mask[db.core_sample_indices_]
= Truelabels = db.labels_
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
unique_labels = set(labels)colors = ['y', 'b', 'g', 'r']print(colors)for k, col in
zip(unique_labels, colors):
    if k == -1:

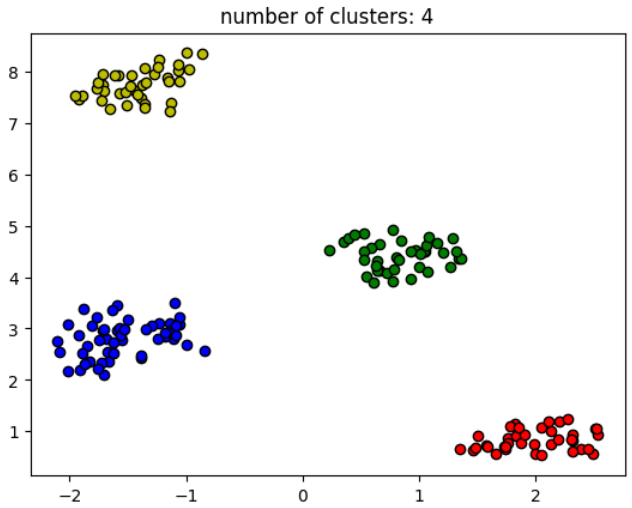
        col = 'k'

    class_member_mask = (labels == k)

    xy = X[class_member_mask & core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=col,
             markeredgecolor='k',
             markersize=6)

    xy = X[class_member_mask & ~core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'x', markerfacecolor=col,
             markeredgecolor='k',
             markersize=6)
plt.title('number of clusters: %d' % n_clusters_)plt.show()
```

Output:



Cluster of dataset

As shown in above output image cluster are shown in different colours like yellow, blue, green and red.

Step 4: Evaluation Metrics

We will use the Silhouette score and Adjusted rand score for evaluating clustering algorithms.

- Silhouette's score is in the range of -1 to 1. A score near 1 denotes the best meaning that the data point i is very compact within the cluster to which it belongs and far away from the other clusters. The worst value is -1. Values near 0 denote overlapping clusters.
- Adjusted Rand Score is in the range of 0 to 1. More than 0.9 denotes excellent cluster recovery and above 0.8 is a good recovery. Less than 0.5 is considered to be poor recovery.

```
from sklearn import metricssc = metrics.silhouette_score(X, labels)print("Silhouette Coefficient:%0.2f" % sc)ari = metrics.adjusted_rand_score(y_true, labels)print("Adjusted Rand Index: %0.2f" % ari)
```

Output:

Coefficient:0.13
Adjusted Rand Index: 0.31

Black points represent outliers. By changing the eps and the MinPts we can change the cluster configuration.

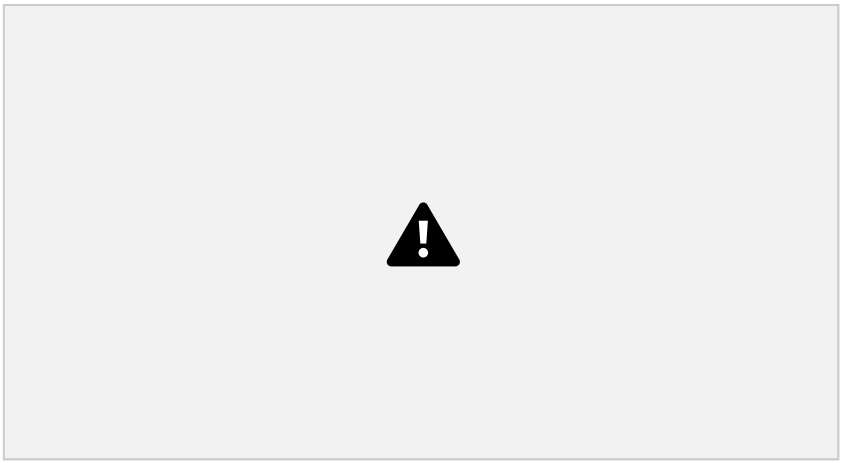
You can download the source code from here.

When Should We Use DBSCAN Over K-Means Clustering

DBSCAN and K-Means are both clustering algorithms that group together data that have the same characteristic. However they work on different principles and are suitable for different types of data. We prefer to use DBSCAN when the data is not spherical in shape or the number of classes is not known beforehand.

DBSCAN	K-Mean
In DBSCAN we need not specify the number of clusters.	It is very sensitive to the number of clusters (k), which must be specified in advance.
Clusters formed in DBSCAN can be of any arbitrary shape.	Clusters formed are spherical or convex in shape
It can work well with datasets having noise and outliers	It does not work well with outliers data. Outliers can skew the

DBSCAN	K-Mean
	clusters in K-Means to a very large extent.
In DBSCAN two parameters are required for training the Model	In K-Means only one parameter is required is for training the model



DBSCAN Vs K-Means

As it can identify clusters of arbitrary shapes and effectively handle noise. K-Means on the other hand is better suited for data with well-defined, spherical clusters and is less effective with noise or complex cluster structures.

Dimensionality Reduction Techniques

Dimensionality reduction is the process of reducing the number of input variables in a dataset while retaining the most important information. It helps to improve model performance, reduces noise and makes complex data easier to visualize and interpret. In this article we will see various Dimensionality Reduction Techniques each designed to uncover hidden structures in high dimensional data.



Dimensionality reduction

Various Dimensionality Reduction Techniques

1. Principal Component Analysis (PCA)

- Principal Component Analysis (PCA) is a classic linear dimensionality reduction method that identifies the directions called principal components in which the data varies the most.
- It works by calculating the covariance matrix of the features and finding its eigenvectors and eigenvalues. The top k eigenvectors with the largest eigenvalues define the new axes that capture the most variance in the data.
- By projecting the original data onto these axes, PCA reduces the number of features while retaining as much information as possible. It's fast, robust and especially effective when the dataset has linearly correlated features. However, it cannot capture complex non linear relationships.

2. Linear Discriminant Analysis (LDA)

- Linear Discriminant Analysis (LDA) is a supervised linear technique designed to find a feature subspace that best separates multiple classes. It does this by maximizing the ratio of between class variance to within class variance in the data, which leads to better class discrimination in the lower-dimensional space.
- It works under the assumption that each class is normally distributed with identical covariance matrices as it focuses on class separability.
- It is very popular in classification tasks like face recognition or handwriting recognition but it is not suitable for purely unsupervised settings.

3. t-Distributed Stochastic Neighbor Embedding (t-SNE)

- t-SNE is a non linear non linear manifold learning technique mainly used for visualization. It converts high dimensional pairwise distances into conditional probabilities that represent similarities between points.
- In the lower-dimensional space usually 2D or 3D it arranges points so that similar points stay close together while dissimilar ones stay apart.

- By minimizing the divergence between these probability distributions t-SNE reveals clusters and local structures that other techniques might miss. It works well for exploring patterns in embeddings but is computationally expensive and does not preserve global distances well.

4. Independent Component Analysis (ICA)

- Independent Component Analysis (ICA) is a linear technique that focuses on separating a multivariate signal into additive, statistically independent non Gaussian components.
- Unlike PCA which decorrelates data by finding orthogonal axes ICA goes further by maximizing statistical independence often using measures like kurtosis or negentropy.
- This makes it particularly useful for blind source separation problems like separating overlapping audio signals. ICA assumes the source signals are non gaussian and independent so it's less effective when this isn't true.

5. Non negative Matrix Factorization (NMF)

- NMF factorizes a non negative data matrix into two non negative lower rank matrices. This decomposition enforces a parts based representation because the components must add up not subtract from each other.
- This is useful when the data naturally has non negative features like pixel intensities or word counts.
- It has gained popularity in image processing, topic modeling and bioinformatics because it produces more interpretable results compared to methods like PCA which may mix positive and negative components.

6. FastICA

- FastICA is an optimized and faster version of ICA that uses a fixed point iteration scheme to efficiently find independent components. It's widely used when standard ICA would be too slow specially for large datasets or realtime applications.
- It preserves the core idea of maximizing non Gaussianity to extract independent sources but improves convergence speed and stability. It's commonly used in signal processing tasks like EEG analysis and audio source separation where quick and reliable separation is needed.

7. Isomap (Isometric Mapping)

- Isomap is a non linear algorithm that extends classical Multidimensional Scaling (MDS). Instead of preserving straight line distances, Isomap computes geodesic distances the shortest paths along the manifold surface using a neighborhood graph.
- Then, MDS is applied to these geodesic distances to generate a low dimensional embedding that maintains the intrinsic geometry of the data.

- Isomap is effective when the data lies on a smooth, non linear manifold but it can be sensitive to outliers and requires careful tuning of the neighborhood size.

8. Locally Linear Embedding (LLE)

- Locally Linear Embedding (LLE) is another non linear manifold learning method that preserves local relationships within the data. It assumes that each data point can be reconstructed as a linear combination of its nearest neighbors.
- The algorithm first finds these local reconstruction weights in the original space and then maps the data to a lower dimensional space while preserving the same weights.
- This approach unfolds complex manifolds revealing the underlying structure. LLE is sensitive to the neighborhood size parameter and doesn't preserve global distances well, but it's very good at capturing local geometry.

9. Manifold Learning

- Manifold Learning is a broader concept that includes methods like Isomap, LLE, Hessian LLE, Laplacian Eigenmaps, t-SNE and UMAP.
- The idea is that high dimensional data often lies on a smooth, lower dimensional manifold embedded in a higher dimensional space. These techniques aim to uncover and flatten this manifold, preserving either local or global geometric relationships.
- Manifold learning is particularly valuable for visualizing and exploring complex data structures that cannot be captured by linear methods like PCA.

10. Autoencoders

- Autoencoders are a type of unsupervised neural network architecture designed to learn an efficient, compressed representation called encodings of input data.
- They consist of two main parts: an encoder which maps the input data to a lower dimensional latent space and a decoder which reconstructs the original data from this compressed representation.
- The network is trained to minimize the difference between the input and the reconstruction forcing it to learn the most important features. Unlike linear methods like PCA, autoencoders can capture complex non linear relationships as they use multiple hidden layers and non linear activation functions.

11. Multidimensional Scaling (MDS)

- Multidimensional Scaling (MDS) is a dimensionality reduction technique that helps visualize the similarities or dissimilarities between data points.
- It works by taking a matrix of pairwise distances and finding a lower dimensional space like 2D or 3D where these distances are preserved as much as possible.

- Unlike PCA, which focuses on maximizing variance and assumes linearity, MDS is more flexible because it directly uses distances, making it useful for uncovering hidden structures, clusters or patterns in complex datasets.

With these techniques we can easily do dimensionality reduction to our dataset.

Advantages of Dimensionality Reduction Techniques

- Improves the performance of machine learning models.
- Reduces training time and computation cost.
- Helps in removing irrelevant or duplicate features.
- Makes data easier to visualize in 2D or 3D.
- Makes models less likely to overfit.
- Helps in storing and processing large datasets efficiently.
- Reveals hidden patterns or structure in complex data.