Clustering Algorithms in Machine Learning

Subramani Suresh

Mentored By:
Ramisha Rani K
Ramya Dinesh

What is Clustering in Machine Learning?

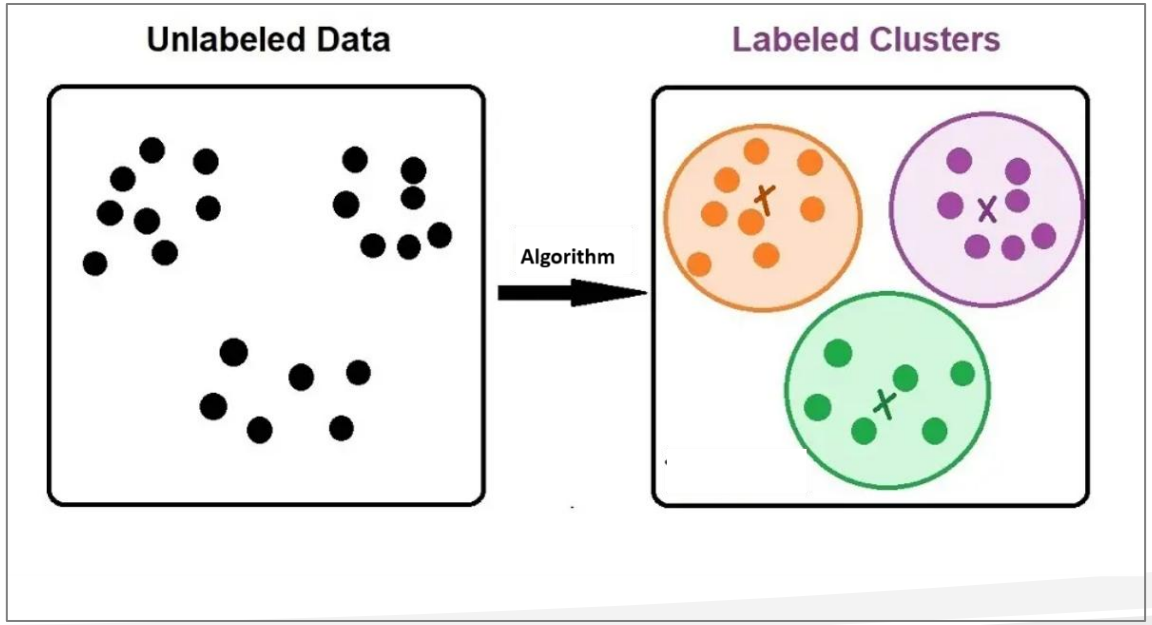
- Clustering is an **unsupervised machine learning technique** used to group similar data points together based on their features, without using predefined labels.
- Data points in the **same cluster** are more similar to each other.
- Data points in **different clusters** are more dissimilar

Why Use Clustering?

- Customer segmentation (marketing).
- Image compression & pattern recognition.
- Document/topic grouping (NLP).

Clustering Algorithms?

- Clustering algorithms are methods that automatically **discover natural groupings** (clusters) in a dataset by measuring **similarity or distance** between data points.



Key Concepts in Clustering

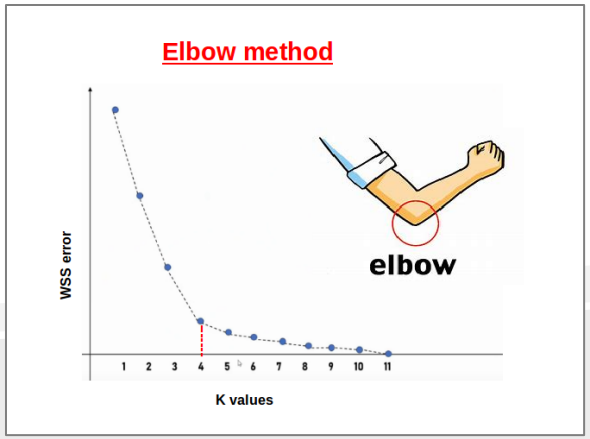
- **Similarity measure** → Often Euclidean distance, cosine similarity, Manhattan distance.
- **Number of clusters (k)** → Some algorithms (like K-Means) require it, others (like DBSCAN) don't.
- **Cluster shape** → Clusters can be spherical, arbitrary, or overlapping.
- **Hard clustering** → Each point belongs to only one cluster.
- **Soft clustering** → A point can belong to multiple clusters with probabilities

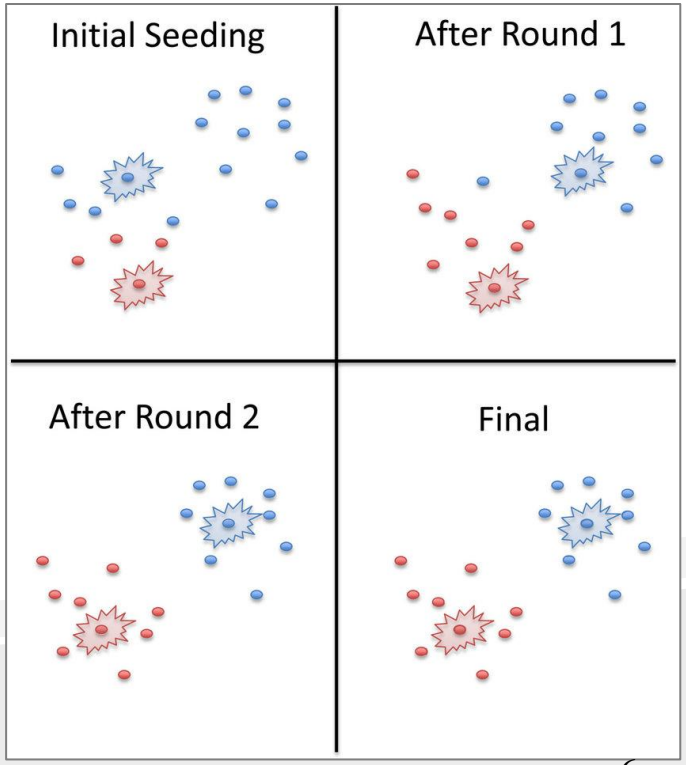
Types of Clustering Algorithms

- **Partition-based** → e.g., K-Means, K-Medoids.
- **Hierarchical** → e.g., Agglomerative, Divisive.
- **Density-based** → e.g., DBSCAN, OPTICS.
- **Model-based** → e.g., Gaussian Mixture Models (GMMs).
- **Others** → Spectral Clustering, Mean-Shift, Affinity Propagation, BIRCH, Fuzzy C-Means.

1. K-Means Clustering

- **Definition:** Unsupervised ML algorithm that partitions data into K clusters.
- **Goal:** Minimize distance between data points & their cluster center (**centroid**).
- **Objective:** Minimize Within-Cluster Sum of Squares (WCSS).
- **Steps**:
  1. The Elbow Method is a technique to find the optimal no. of clusters (K)
  2. Choose K (no. of clusters).
  3. Randomly initialize centroids.
  4. Assign points → nearest centroid.
  5. Update centroids → mean of cluster.
  6. Repeat until stable.

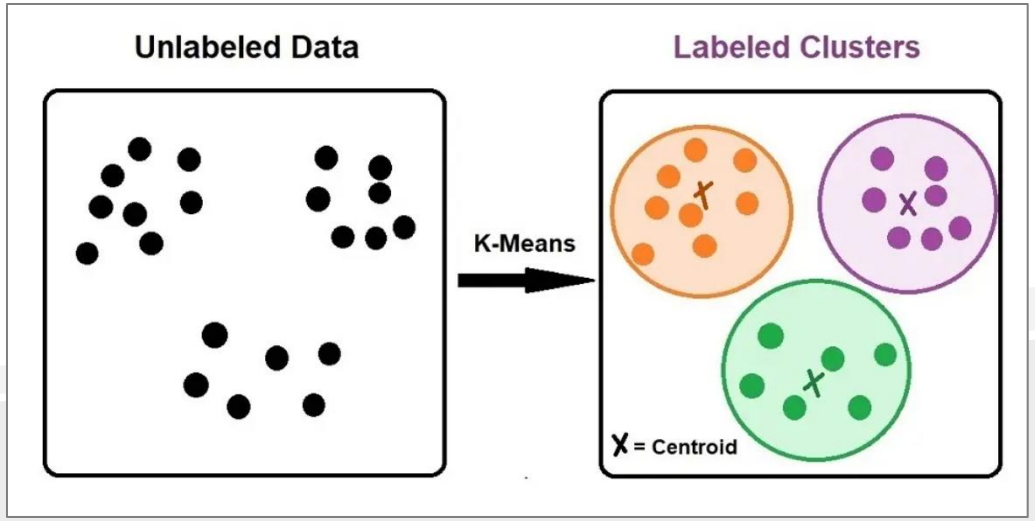




Advantages & Limitation

- **Advantages**:
  - ✓ Simple & fast.
  - ✓ Works well for large datasets.
  - ✓ Easy to implement
- **Limitations**:
  - Must predefine K
  - Sensitive to outliers & noise.
  - Works best for spherical clusters.
  - Results depend on initial centroids

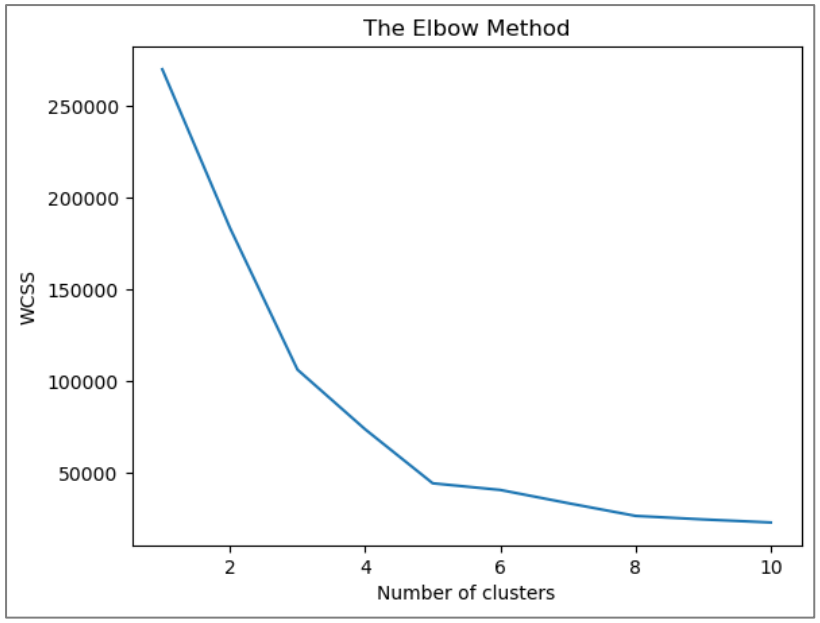
- **Applications**:
  - ✓ Customer segmentation
  - ✓ Image compression
  - ✓ Market basket analysis
  - ✓ Document clustering



Python:

- **Elbow Method in K-Means:** The bend (elbow) shows the optimal K

```python
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
list1 = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 42)
    kmeans.fit(X)
    list1.append(kmeans.inertia_)
plt.plot(range(1, 11), list1)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```



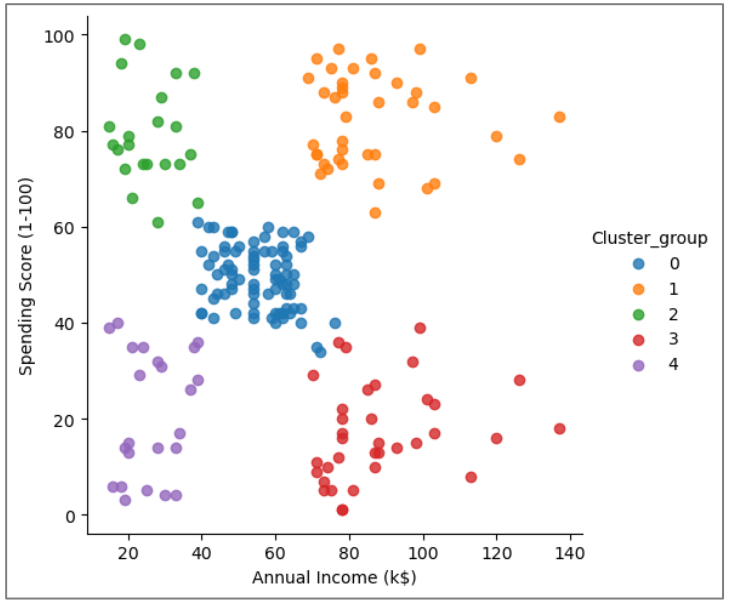
Python:

- **Model Creation**

```
#model creation
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters = 5, init = 'k-means++', random_state = 42)
y_kmeans = kmeans.fit_predict(X)
```

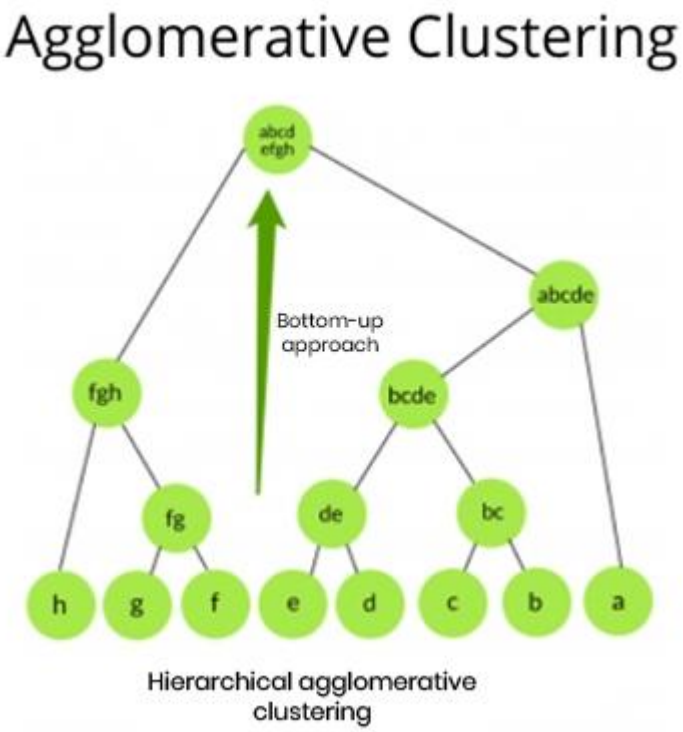
K-Means clustering visualization

```
import seaborn as sns
facet = sns.lmplot(data=supervised, x=supervised.columns[3],
                   y=supervised.columns[4], hue=supervised.columns[5],
                   fit_reg=False, legend=True, legend_out=True)
```



2. Agglomerative Clustering

- Type of **Hierarchical Clustering** (bottom-up).
- Starts with **each data** point as its **own cluster**
- Iteratively merges the **closest clusters** until all points form one big cluster (**dendrogram**).
- **Output:** A hierarchy of clusters (tree structure).
- **Steps**:
  1. Start with N clusters (each data point = 1 cluster).
  2. Compute pairwise distance (similarity) between clusters.
  3. Merge the two closest clusters.
  4. Repeat until desired number of clusters (K) is reached.

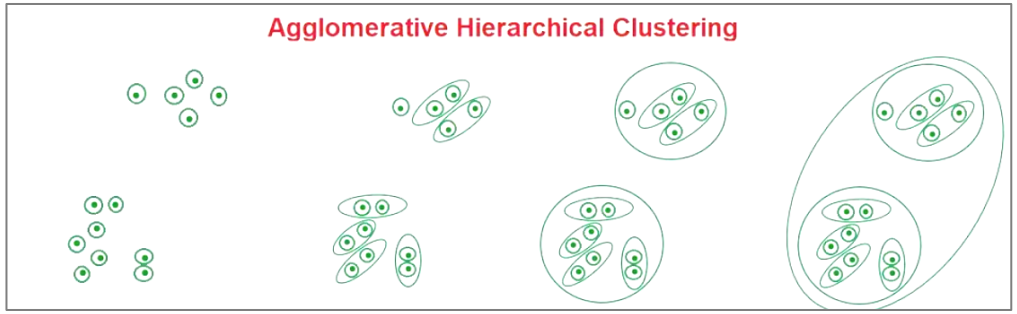


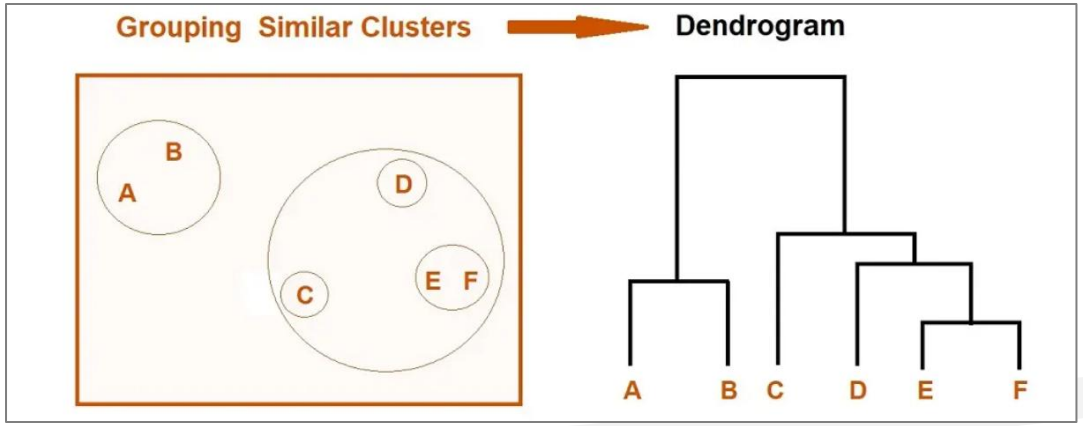
Linkage Criteria (decides cluster merging):

- Single linkage → min distance.
- Complete linkage → max distance.
- Average linkage → mean distance.
- Ward's method → minimizes variance

Advantages & Limitation

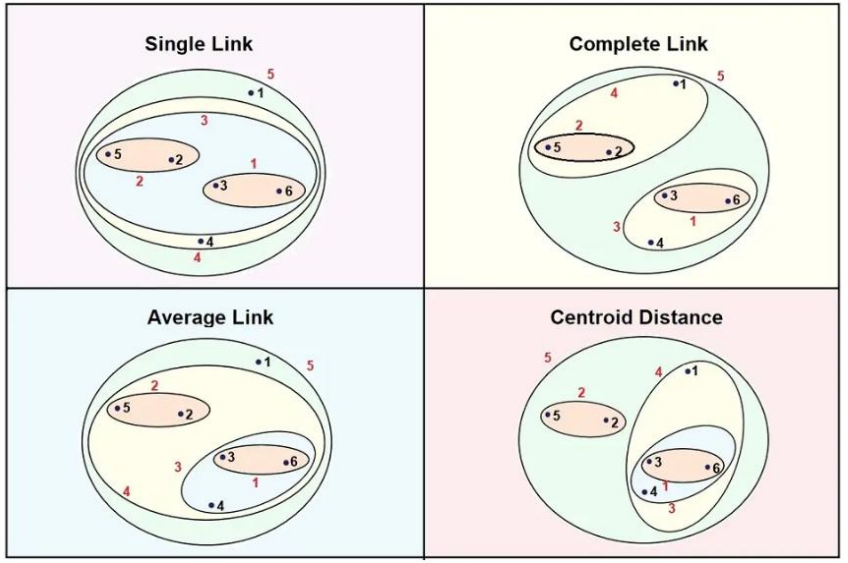
- **Advantages**:
  - ✓ No need to specify K (can cut dendrogram later).
  - ✓ Produces a full **hierarchy of clusters**.
  - ✓ Works with different distance metrics
- **Limitations**:
  - Computationally expensive for large datasets.
  - Sensitive to noise & outliers.
  - Once merged, clusters cannot be split.



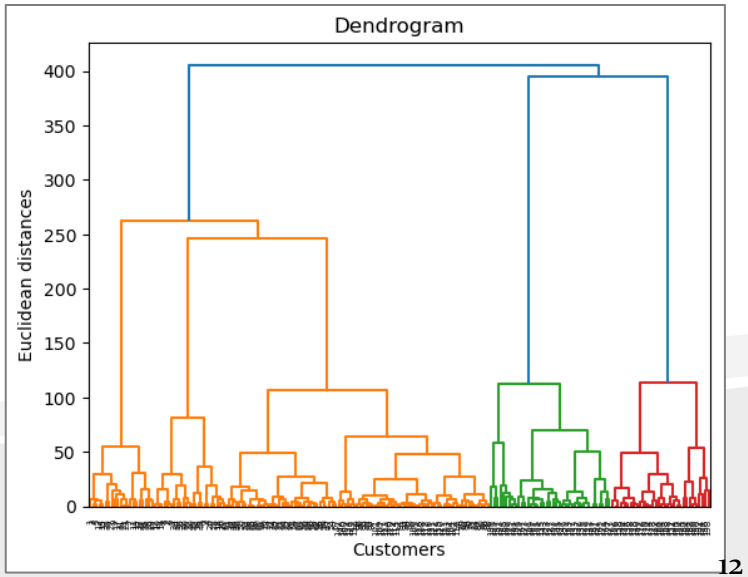


Python:

- **Choose K in Agglomerative Clustering**
- Dendrogram "Cut" Method (Most Common)
- Run hierarchical clustering and build a **dendrogram** (tree diagram).
- Look for a large vertical gap (big jump in linkage distance).
- Cut the dendrogram horizontally at that gap → number of clusters = number of vertical lines cut.




```
import matplotlib.pyplot as plt
import scipy.cluster.hierarchy as sch
dendrogram = sch.dendrogram(sch.linkage(X, method = 'ward'))
plt.title('Dendrogram')
plt.xlabel('Customers')
plt.ylabel('Euclidean distances')
plt.show()
```



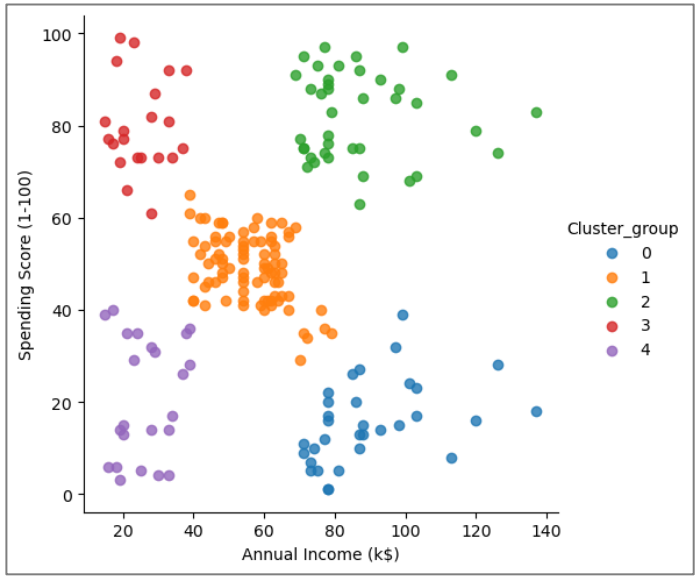
Python:

- **Model Creation: Agglomerative Clustering**

```
#model Creation
from sklearn.cluster import AgglomerativeClustering
clusmodel = AgglomerativeClustering(n_clusters = 5)
label = clusmodel.fit_predict(X)
```

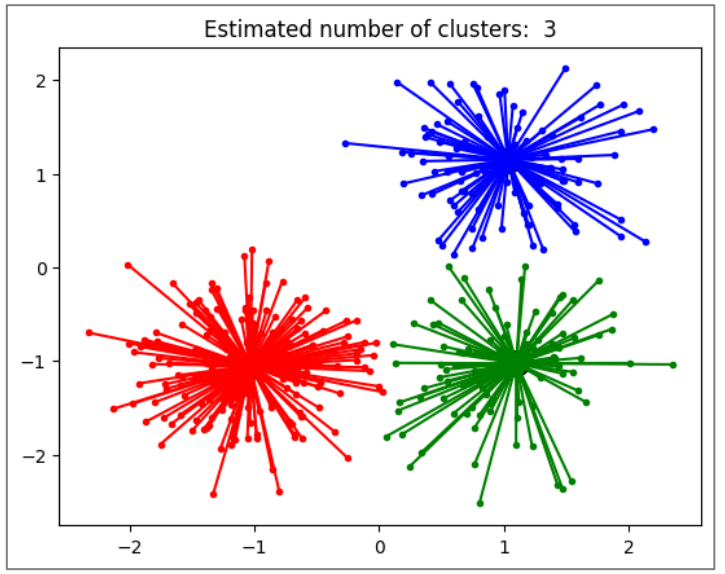
clustering visualization

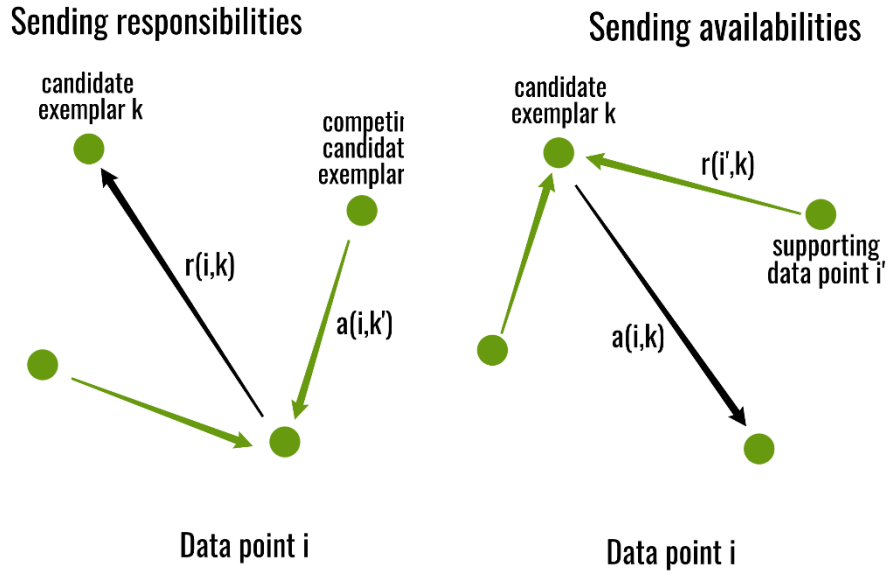
```
import seaborn as sns
facet = sns.lmplot(data=supervised, x=supervised.columns[3],
                   y=supervised.columns[4], hue=supervised.columns[5],
                   fit_reg=False, legend=True, legend_out=True)
```



3. Affinity Propagation

- **Affinity** Propagation (AP) is an **unsupervised clustering algorithm**.
- Unlike K-Means, **you don't need to predefine K** (number of clusters).
- It works by passing **messages between data points** until clusters (exemplars) emerge.
- **Exemplars** = representative points that act as cluster centers.
- **Steps**:
  1. Compute **similarity matrix** (measure how similar points are).
  2. Initialize all points as potential exemplars.
  3. Exchange two types of messages:
     - **Responsibility (r)**: How well a point would serve as exemplar.
     - **Availability (a):** How appropriate it is for a point to choose another as exemplar.
  4. Iteratively update messages until **convergence**.
  5. Points are assigned to their **exemplar cluster**





Advantages & Limitation

- **Advantages**:
  - ✓ No need to choose **K**.
  - ✓ Finds clusters of **different shapes & sizes**.
  - ✓ Can identify **outliers** (points not chosen as exemplars)
- **Limitations**:
  - Computationally expensive for large datasets.
  - Sensitive to similarity measure & preference parameter.
  - Can produce many small clusters if not tuned properly.

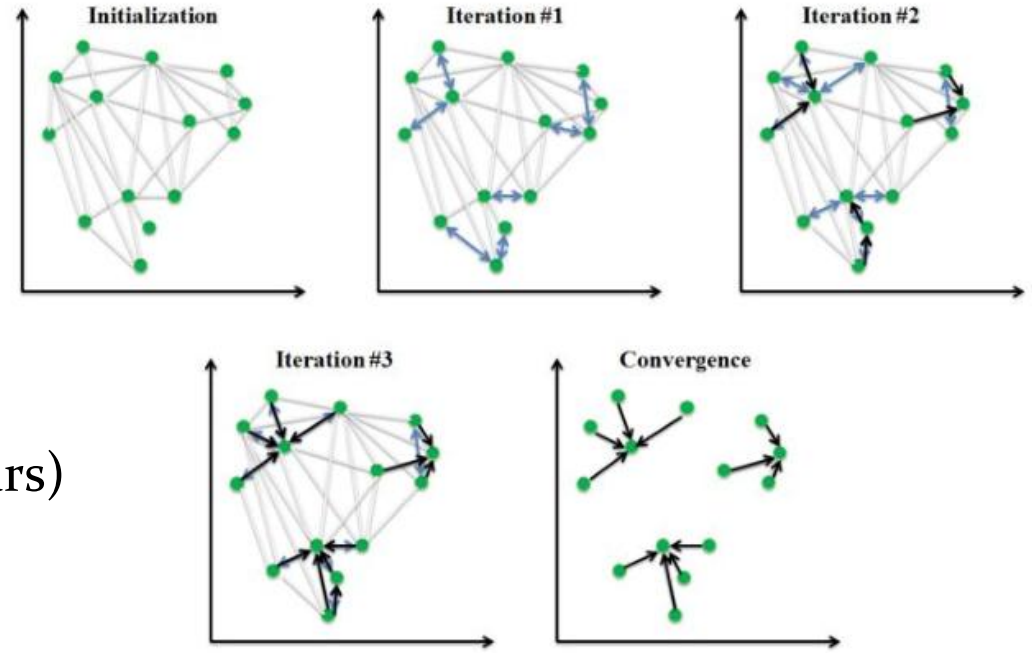
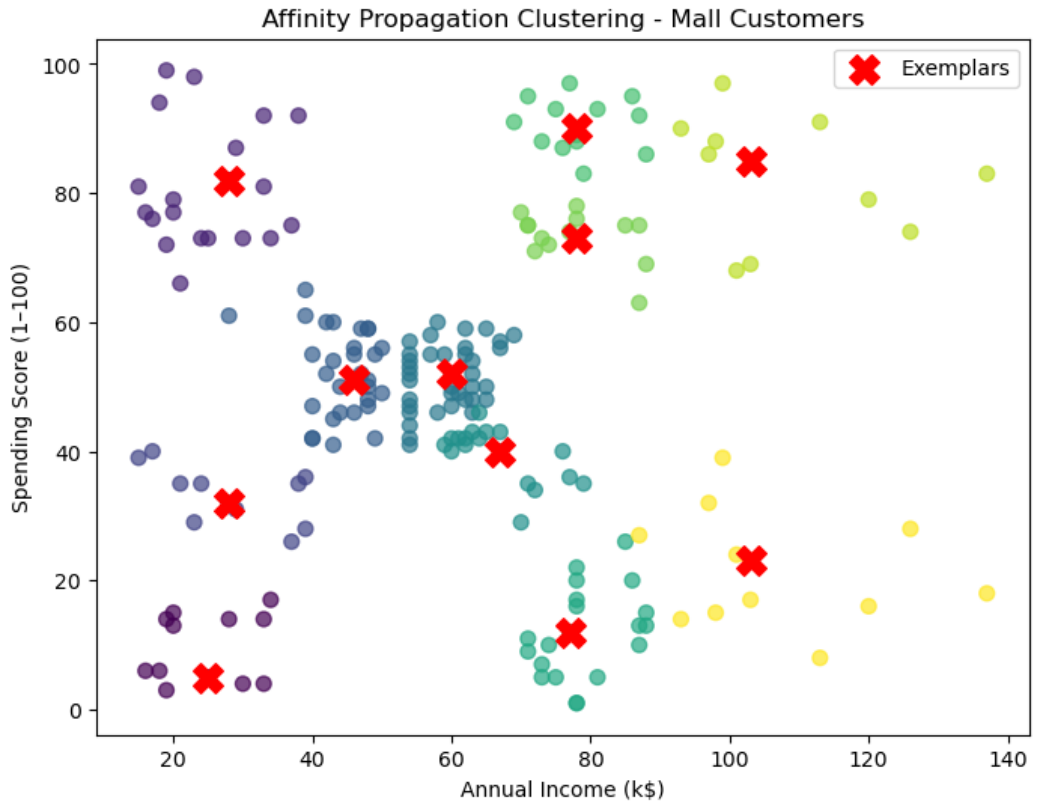


Figure 2. The progressing of the affinity Propagation.

Python:

Explanation of Output
•**Points** = Customers (based on Annual Income vs Spending Score).
•**Colors** = Different clusters formed by Affinity Propagation.
•**Red** ✖ = **Exemplars** (chosen representative customers for each cluster).

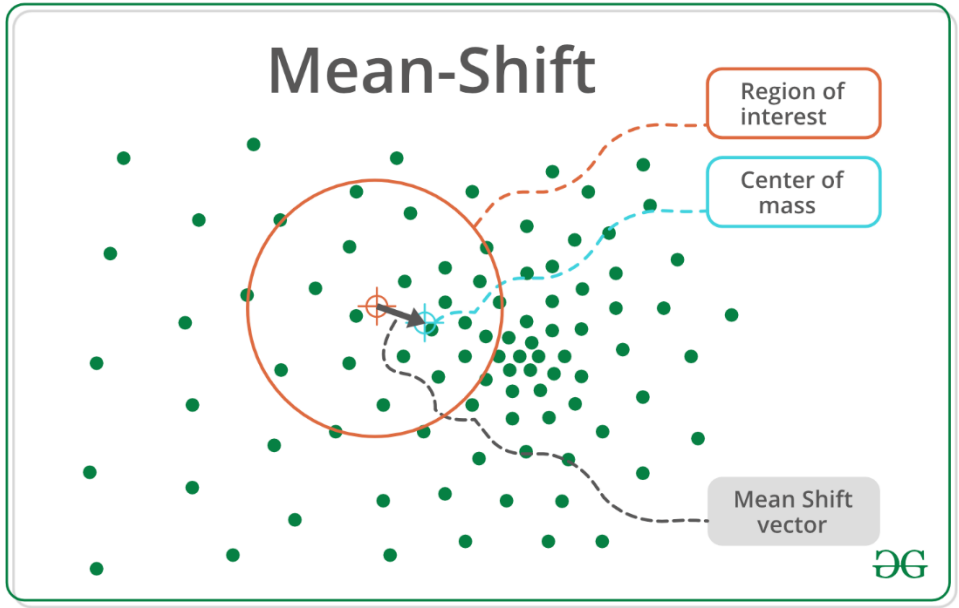



```python
import numpy as np
import pandas as pd
#-----data collection------
dataset = pd.read_csv('Mall_Customers.csv')
X = dataset.iloc[:, [3, 4]].values
#-----model Creation-------
from sklearn.cluster import AffinityPropagation
ap = AffinityPropagation(random_state=42)
labels = ap.fit_predict(X)
cluster_centers = ap.cluster_centers_
#-----Visualization-------
import matplotlib.pyplot as plt
plt.figure(figsize=(8,6))
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=50, alpha=0.7)
plt.scatter(cluster_centers[:, 0], cluster_centers[:, 1],
            c='red', marker='X', s=200, label='Exemplars')
plt.xlabel("Annual Income (k$)")
plt.ylabel("Spending Score (1-100)")
plt.title("Affinity Propagation Clustering - Mall Customers")
plt.legend()
plt.show()
```

4. Mean Shift Clustering

- **Mean Shift** is an **unsupervised clustering algorithm**.
- Works by **sliding a window (kernel)** towards the region of **highest data density.**
- Unlike K-Means. **No need to predefine K** (number of clusters), Finds **arbitrarily shaped clusters.**
- **Steps**:

1. Choose a **window radius (bandwidth).**
2. Place a window around each data point.
3. Compute **mean of points inside the window**.
4. Shift the window center to this mean.
5. Repeat until convergence → nearby windows merge into clusters



Advantages & Limitation

- **Advantages**:
  - ✓ No need to specify number of clusters.
  - ✓ Can detect **arbitrary shaped clusters.**
  - ✓ Good at finding **high-density regions.**
- **Limitations**:
  - Computationally expensive (especially for large datasets).
  - Performance depends on **bandwidth selection.**
  - Can produce too many clusters if bandwidth is smal

Python: Apply Mean Shift clustering

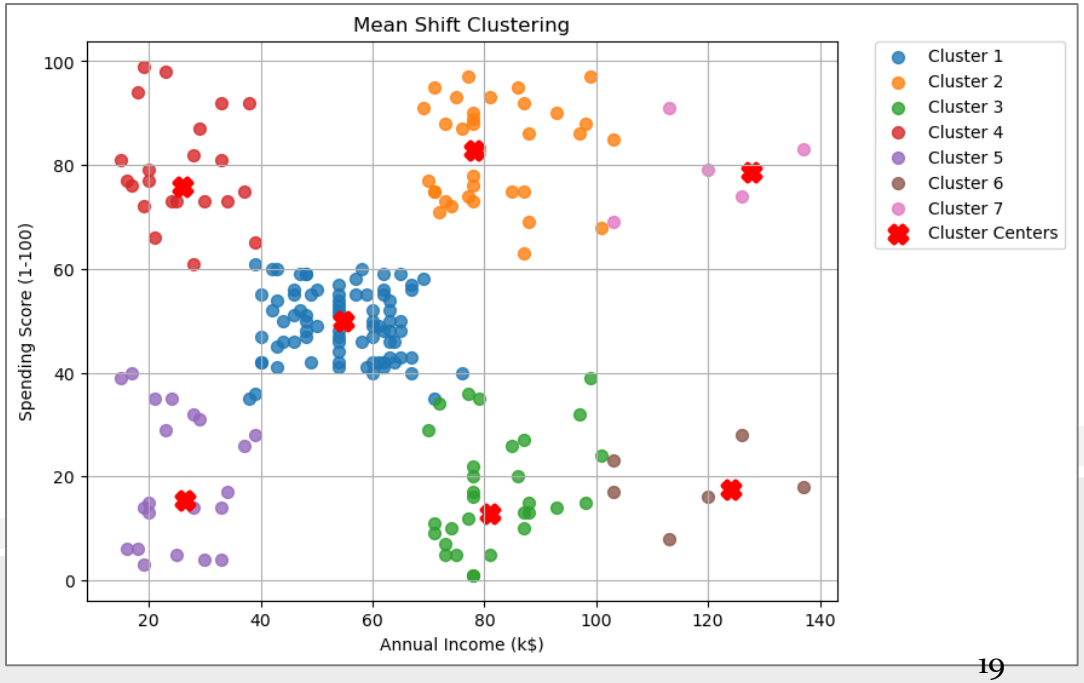
```python
# Apply Mean Shift clustering
from sklearn.cluster import MeanShift, estimate_bandwidth
bandwidth = estimate_bandwidth(X, quantile=0.1)
clustering = MeanShift(bandwidth=bandwidth,bin_seeding=True).fit(X)

# Extract results
labels = clustering.labels_
cluster_centers = clustering.cluster_centers_
n_clusters_ = len(np.unique(labels))
print(f"Number of estimated clusters: {n_clusters_}")
unique_labels = np.unique(labels)
```

- In Mean Shift clustering, the **bandwidth** is the radius (**window size**)
- used to search for nearby points when computing the mean.
- quantile (like 0.1): → **smaller window** → more clusters.
- quantile (like 0.3 or 0.4): → **bigger window** → fewer clusters.

Cluster visualization

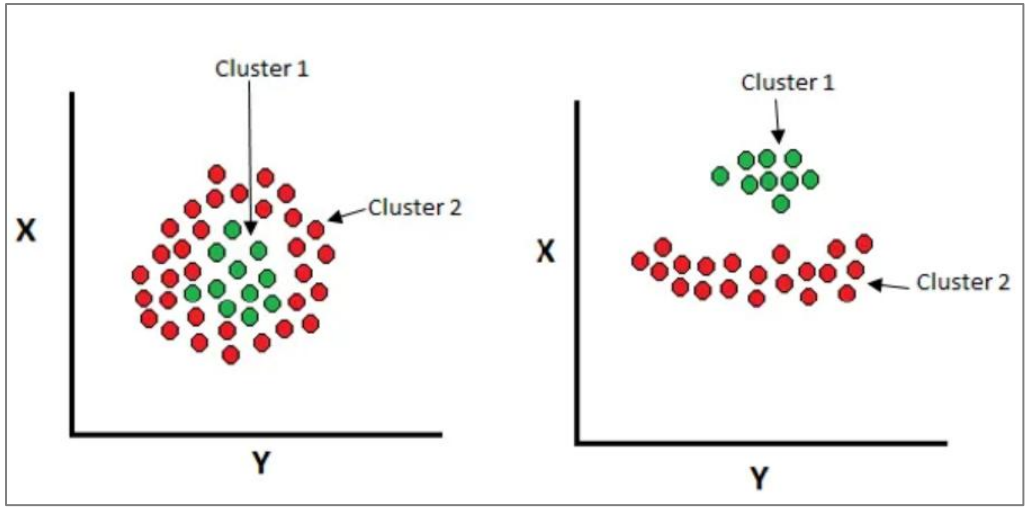
```python
#Visualize the results
plt.figure(figsize=(8, 6))
# Plot each cluster separately with a label
for cluster in unique_labels:
    plt.scatter(X[labels == cluster, 0], X[labels == cluster, 1],
                marker='o', s=50, alpha=0.8,
                label=f"Cluster {cluster+1}")
plt.scatter(cluster_centers[:, 0], cluster_centers[:, 1], c='red',
            marker='X', s=100, linewidths=3, label='Cluster Centers')
plt.title('Mean Shift Clustering')
plt.xlabel(dataset.columns[3])
plt.ylabel(dataset.columns[4])
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.grid(True)
plt.show()
```

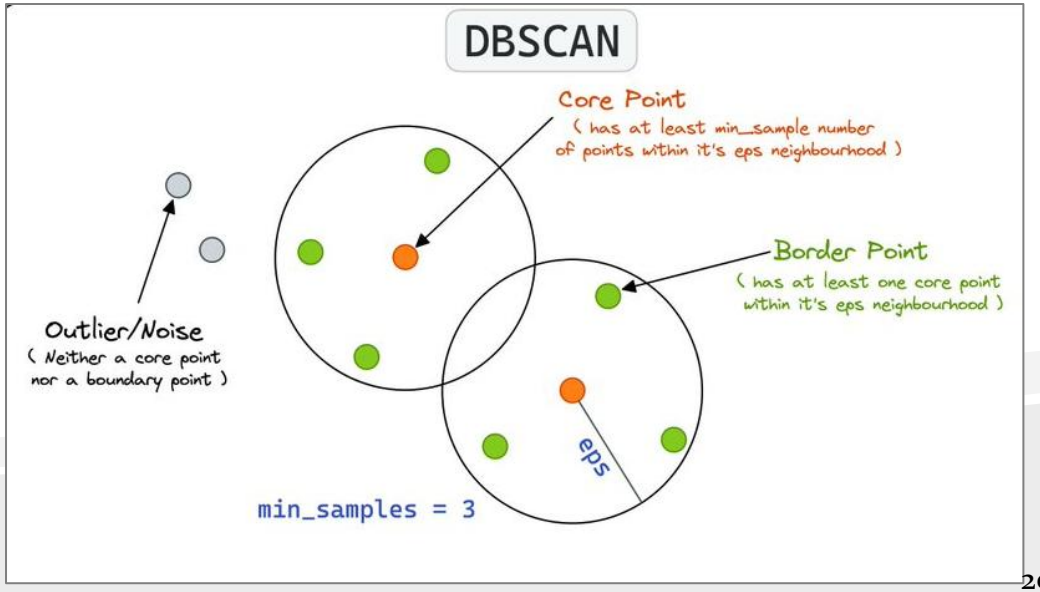


5. DBSCAN

- **DBSCAN** (**D**ensity-**B**ased **S**patial **C**lustering of **Applications** with **N**oise)
- Finds clusters of **arbitrary shape**
- Identifies outliers (**noise**)
- Groups together points that are **closely packed**
- **Steps**:

1. Choose parameters: **eps** and **min_samples**
2. For each point:.
   - **If ≥ min_samples** points within **eps** → **Core Point**
   - Points within neighborhood → **Cluster Members**
   - Points not belonging to any cluster → **Noise (-1)**
3. Repeat until all points are **visited**





Advantages & Limitation

- **Advantages**:
  - ✓ Finds clusters of **arbitrary shapes**
  - ✓ Detects outliers (noise)
  - ✓ No need to predefine number of clusters (unlike K-Means).
- **Limitations**:
  - Sensitive to parameter choice
  - Struggles with varying densities
  - Struggles with varying densities as it requires a single epsilon value for all points.

Python: Apply DBSCAN clustering

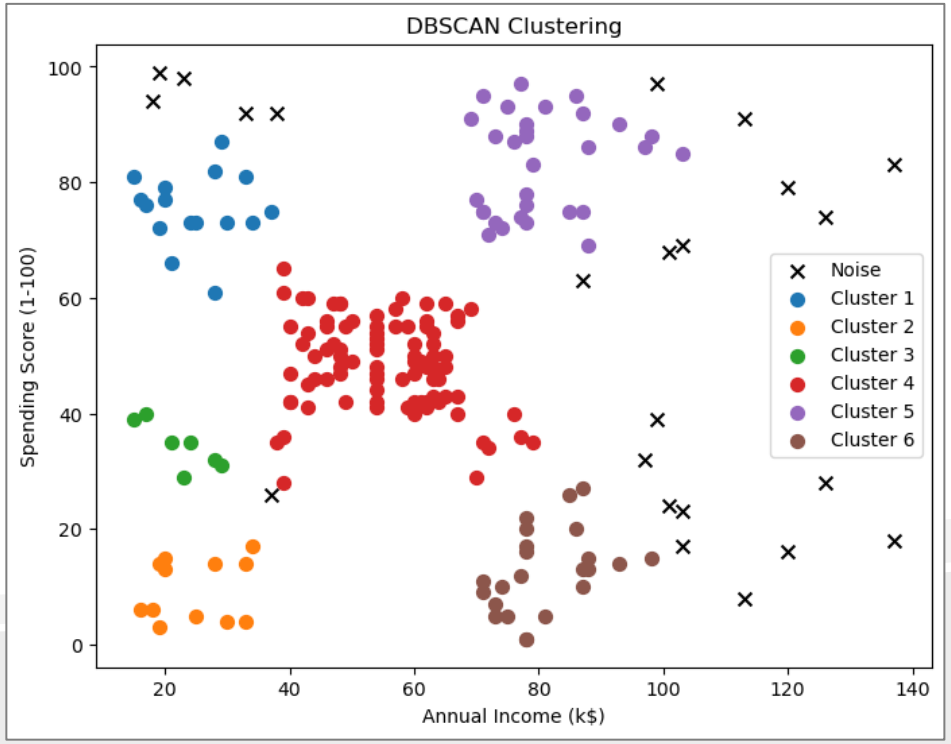
```
# Apply DBSCAN
from sklearn.cluster import DBSCAN
# tune eps for your dataset
db = DBSCAN(eps=9, min_samples=5).fit(X)

#Get cluster labels
labels = db.labels_
print("Cluster labels:", np.unique(labels))
```

- **Tune in DBSCAN:**
- eps=5 is just a starting value. Since your features are in the same scale (Income in 15–137, Spending Score in 1–99), you can skip scaling.
- If clusters don't look good, adjust eps (try 3 → 10).
- -1 = noise (customers not in any cluster)

Cluster visualization

```
#Plot clusters with legend
plt.figure(figsize=(8,6))
unique_labels = np.unique(labels)
for cluster in unique_labels:
    if cluster == -1:
        # Noise points
        plt.scatter(X[labels == cluster, 0], X[labels == cluster, 1],
                    c='black', marker='x', s=50, label='Noise')
    else:
        # Cluster points
        plt.scatter(X[labels == cluster, 0], X[labels == cluster, 1],
                    s=50, label=f'Cluster {cluster+1}')
plt.title("DBSCAN Clustering")
plt.xlabel(dataset.columns[3])
plt.ylabel(dataset.columns[4])
plt.legend()
plt.show()
```



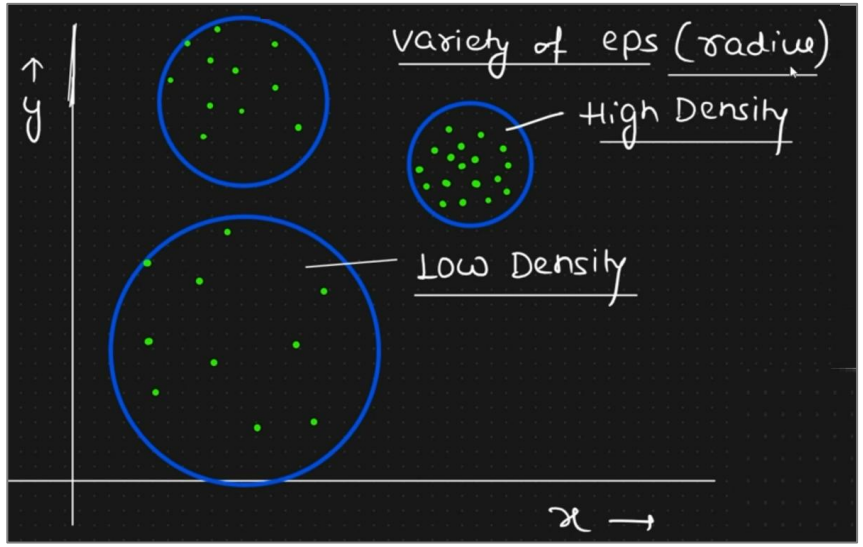
6. OPTICS

- **OPTICS** (**O**rdering **P**oints **T**o **I**dentify the **C**lustering **S**tructure)
- **Density-based clustering** (like DBSCAN)
- Handles clusters of **varying density**
- Produces an **ordering of points** (reachability plot) instead of just labels

- **Steps**:

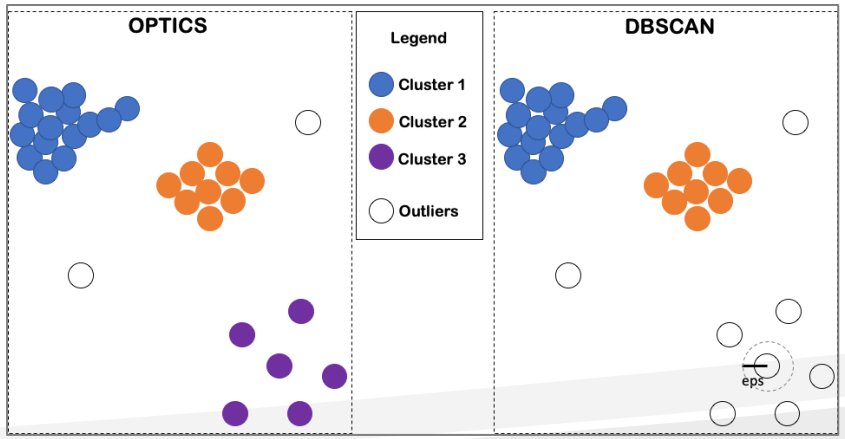
1. **Pick parameters**: min_samples, max_eps (optional), xi, min_cluster_size
2. **Compute reachability distances** for all points
3. **Order points** based on density connectivity
4. **Generate reachability plot** to visualize cluster structure
5. **Extract clusters** at different density levels
6. **Label points**: Cluster IDs → 0,1,2,… | Noise points → -1

Reachability distances



Advantages & Limitation

- **Advantages**:
  - ✓ Detects **clusters of varying density** (better than DBSCAN).
  - ✓ No need to predefine **eps**.
  - ✓ Produces a **reachability plot** → helps visualize clustering structure.
  - ✓ Handles **noise & outliers** naturally..
- **Limitations**:
  - **More complex** and **slower than** DBSCAN.
  - Interpretation of the reachability plot can be tricky.
  - **Requires parameter tuning** (xi, min_cluster_size).
  - May produce **many small clusters** if not tuned well



Python: Apply OPTICS Clustering

```python
# Apply OPTICS
# tune min_samples & xi/clust_method for better results
from sklearn.cluster import OPTICS
optics = OPTICS(min_samples=5, xi=0.05)
optics.fit(X)

# Get cluster labels
labels = optics.labels_
print("Cluster labels:", np.unique(labels))
```

Cluster visualization

```python
# Plot clusters with legend
plt.figure(figsize=(8, 6))
unique_labels = np.unique(labels)
for cluster in unique_labels:
    if cluster == -1:
        # Noise points
        plt.scatter(X[labels == cluster, 0], X[labels == cluster, 1],
                    c='black', marker='x', s=50, label='Noise')
    else:
        # Cluster points
        plt.scatter(X[labels == cluster, 0], X[labels == cluster, 1],
                    s=50, label=f'Cluster {cluster+1}')

plt.title("OPTICS Clustering")
plt.xlabel(dataset.columns[3])
plt.ylabel(dataset.columns[4])
plt.legend()
plt.show()
```

Key Parameters to Tune in OPTICS

1.min_samples

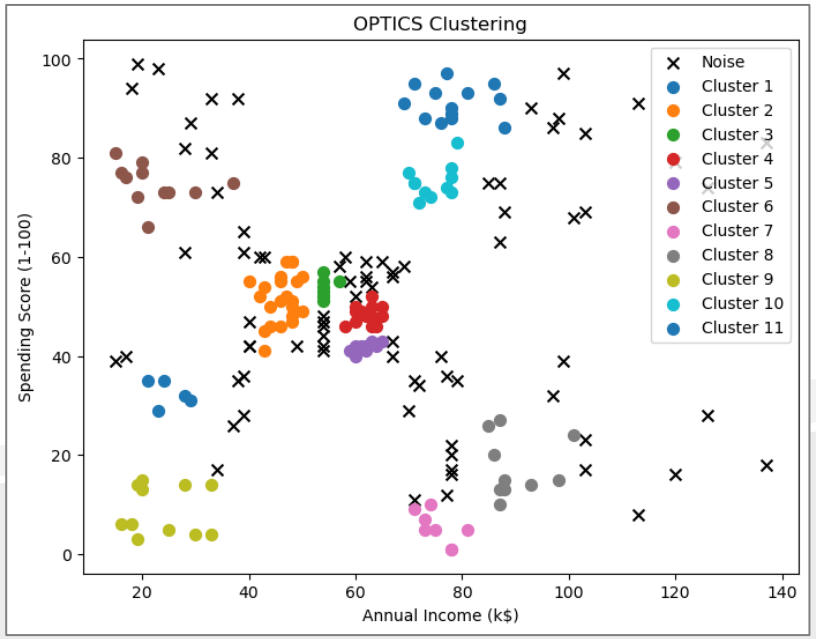
- Controls density requirement.
- Small = more clusters, including tiny ones.
- Large = fewer, denser clusters.
- Common range: **3** → **10**.

2.xi (cluster steepness)

- Lower = more fine-grained clusters.
- Higher = merges clusters.
- Typical range: **0.03** → **0.1**.

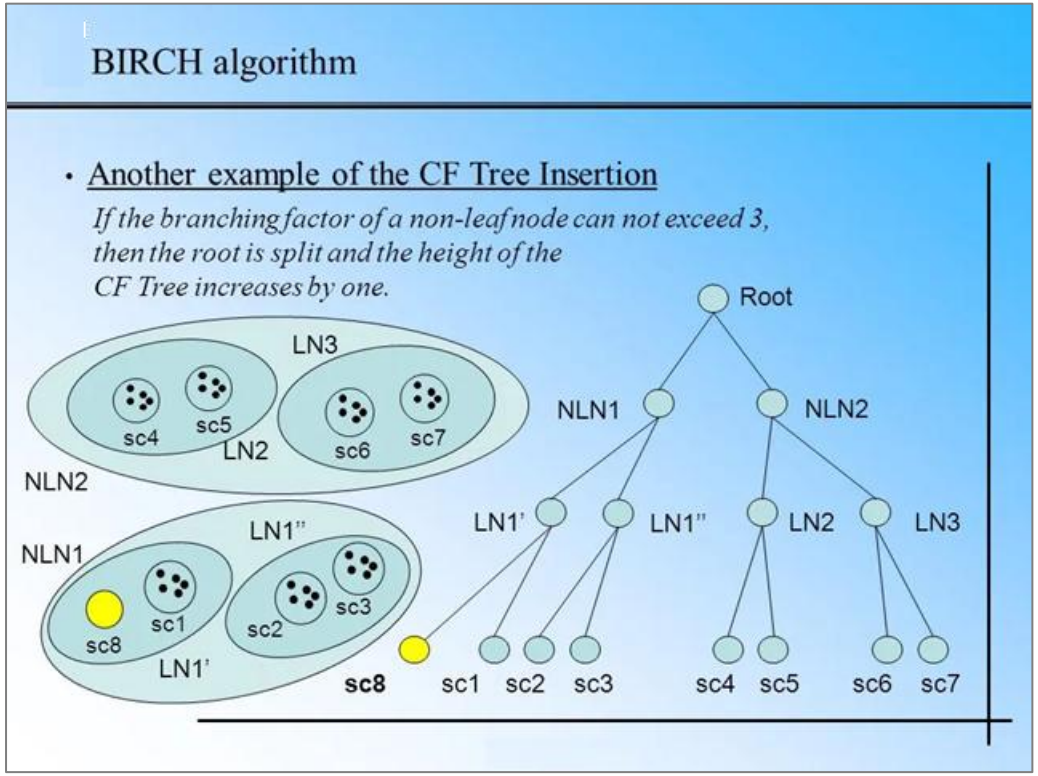
3.min_cluster_size

- Minimum fraction of total points in a cluster.
- Small (e.g., 0.02) → more, smaller clusters.
- Larger (e.g., 0.1) → fewer, bigger clusters.



7. BIRCH

- **BIRCH Clustering** (**B**alanced **I**terative **R**educing and **C**lustering using **H**ierarchies)
- **Hierarchical** + **centroid-based** clustering method.
- Designed for **large datasets** — scalable and memory efficient.
- Uses a **CF (Clustering Feature) Tree** to incrementally build cluster.



- **Steps**:

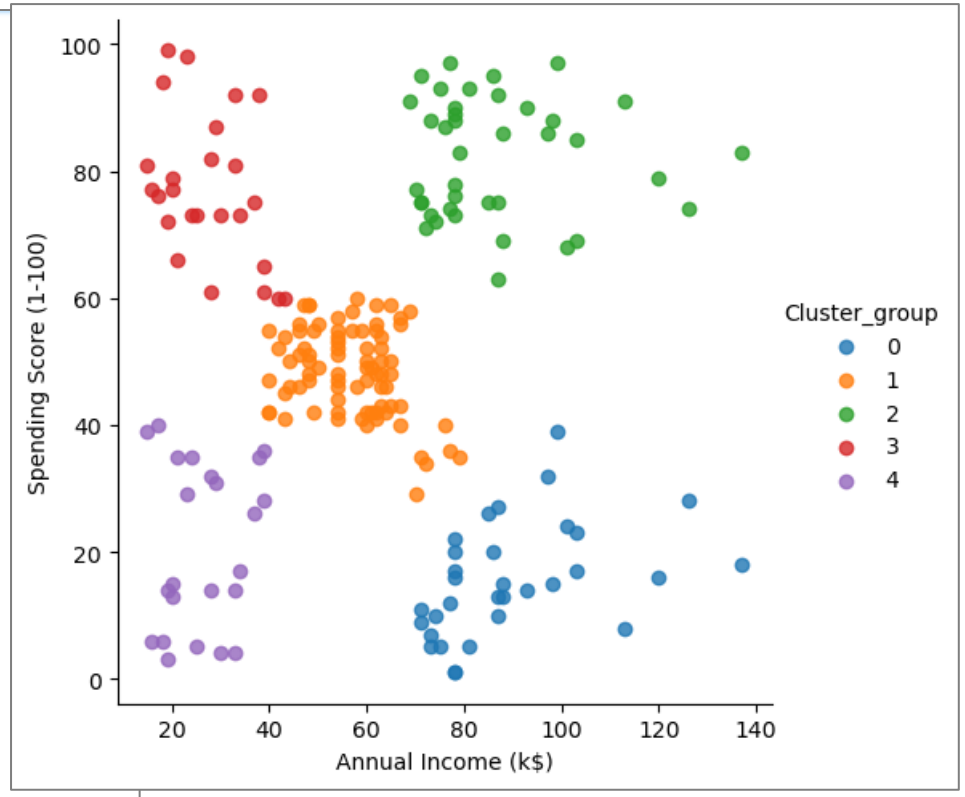
1. **Input Data** → Feed dataset into algorithm
2. **Build CF Tree** → Summarize data into compact Clustering Features (CF)
3. **Insert Points Incrementally** → Place each new point into closest leaf node
4. **Split if Needed** → If node exceeds threshold, it splits
5. **Condense Tree** → Merge or prune clusters for compactness
6. **Final Clustering** → Apply global method (e.g., K-Means / Agglomerative) on leaf nodes
7. **Assign Labels** → Each point inherits cluster ID from its leaf

Advantages & Limitation

- **Advantages**:
  - ✓ Handles **large datasets efficiently** (better than K-Means, DBSCAN for huge data).
  - ✓ Incremental — processes data in a **single scan**.
  - ✓ Good for **outlier detection**.
  - ✓ Works well as a **pre-clustering step**.
- **Limitations**:
  - Works best with **spherical-shaped clusters**.
  - Sensitive to parameter choices (**branching_factor**, **threshold**).
  - Not ideal for datasets with **non-convex shapes** (where DBSCAN/OPTICS work better).

Python: Apply BIRCH Clustering

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
#Load your dataset
dataset = pd.read_csv("Mall_Customers.csv")
#Select features for clustering
# Annual Income & Spending Score
X = dataset.iloc[:, [3, 4]].values
#Apply BIRCH clustering with 5 clusters
from sklearn.cluster import Birch
birch_model = Birch(n_clusters=5)
labels = birch_model.fit_predict(X)
#Plot clusters with legend
supervised=pd.DataFrame(dataset)
supervised['Cluster_group']=labels
import seaborn as sns
facet = sns.lmplot(data=supervised, x=supervised.columns[3],
                   y=supervised.columns[4], hue=supervised.columns[5],
                   fit_reg=False, legend=True, legend_out=True)
```



Thank You