n this module, you will learn how unsupervised learning techniques uncover hidden patterns in data without using labeled responses. You'll describe clustering concepts and apply K-Means to real-world customer segmentation tasks. You'll also compare DBSCAN and HDBSCAN models to identify dense clusters in spatial data. Moving beyond clustering, you'll explore dimensionality reduction as a tool for simplifying high-dimensional datasets. You'll apply PCA to uncover key components and use advanced techniques like t-SNE and UMAP to visualize data structure. To support your learning, you'll receive a Cheat Sheet: Building Unsupervised Learning Models, highlighting core methods, practical use cases, and comparison guidelines.

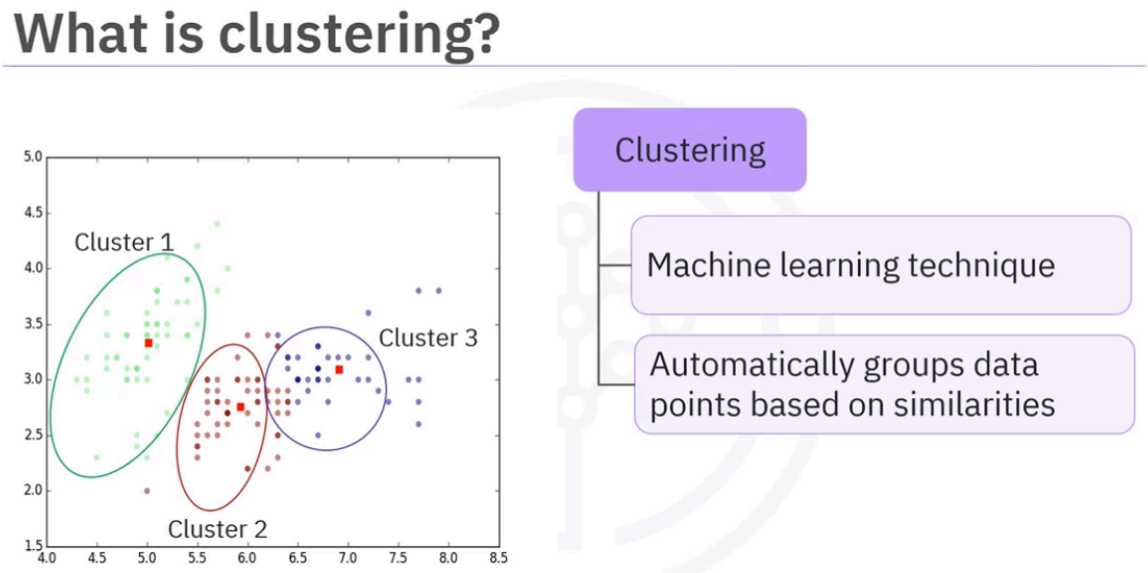
Learning Objectives

- Describe how unsupervised learning identifies patterns in unlabeled data
- Explain the concept of clustering and its application in grouping similar data points
- Apply K-Means clustering using scikit-learn to segment customers based on behavioral data
- Compare the characteristics and results of DBSCAN and HDBSCAN clustering methods
- Explain the purpose of dimensionality reduction in simplifying complex datasets.
- Apply Principal Component Analysis (PCA) to reduce dimensions and interpret data patterns
- Use advanced dimensionality reduction algorithms like t-SNE and UMAP to visualize high-dimensional data

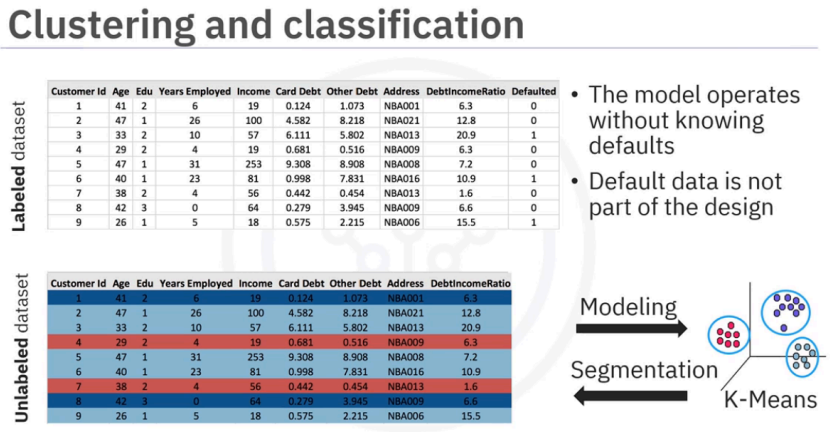
Here are **well-structured notes** from your clustering module:

📌 Clustering Strategies in Real-World Applications

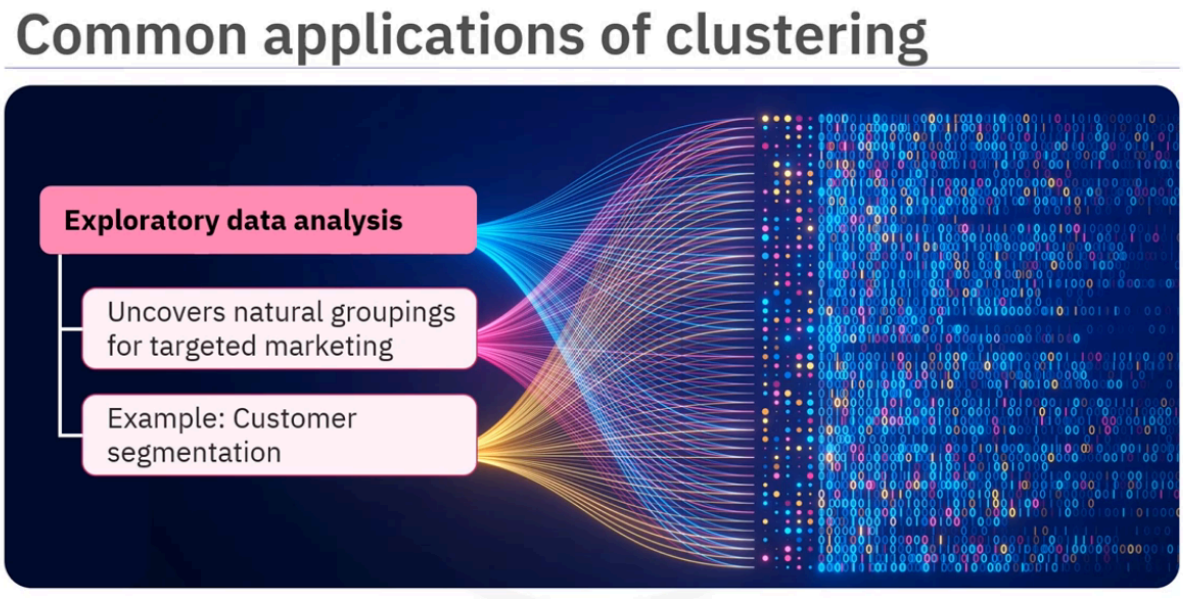
- **What is Clustering?**



- **Definition**: Clustering is an **unsupervised machine learning technique** that groups data points into clusters based on similarities.
- Works with **unlabeled data** (no target labels).
- Finds **natural groupings/patterns** within datasets.



🔹 Applications of Clustering

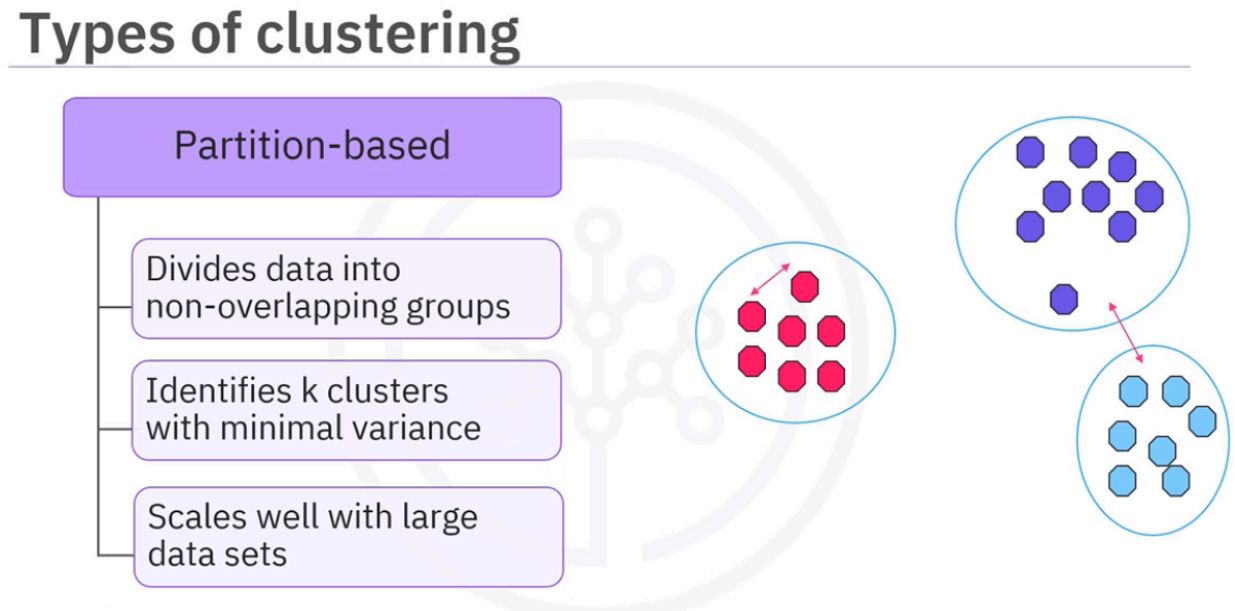


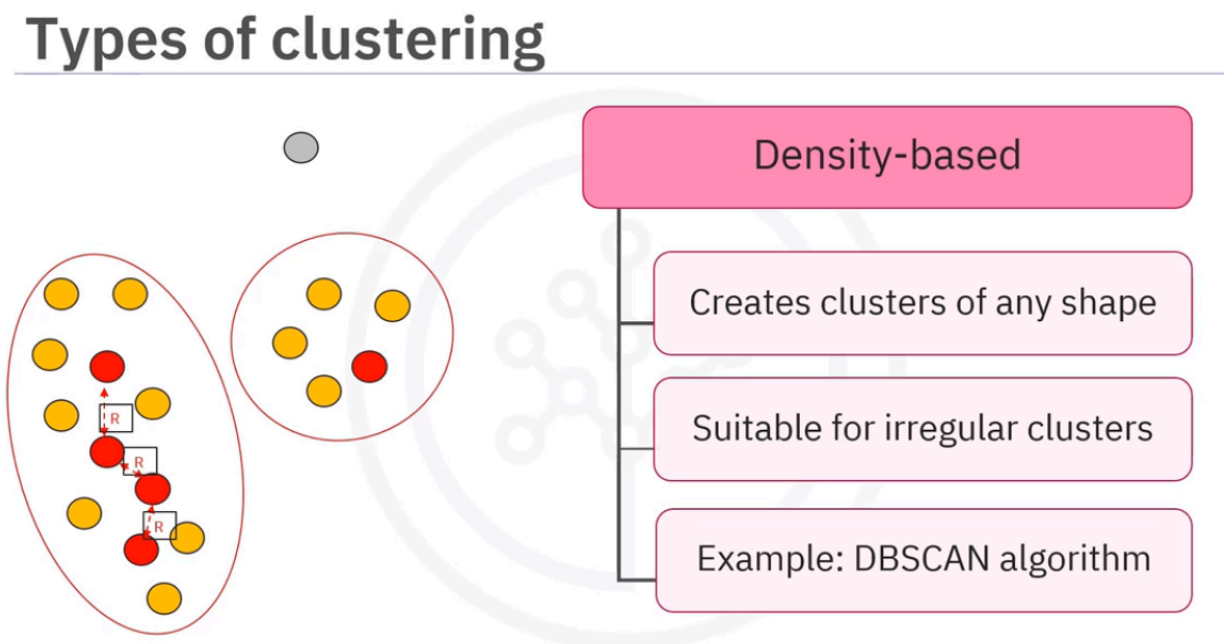
1. **Customer Segmentation** – group customers for targeted marketing.
2. **Pattern Recognition** – e.g., music genre identification, market segmentation.
3. **Image Segmentation** – detecting medical abnormalities.
4. **Anomaly Detection** – spotting fraud, equipment malfunction, or outliers.
5. **Feature Engineering** – create new features or reduce dimensionality.
6. **Data Summarization** – simplify data by representing groups with cluster centers.
7. **Data Compression** – reduce image/data size by replacing points with cluster centroids.

🔹 Clustering vs Classification

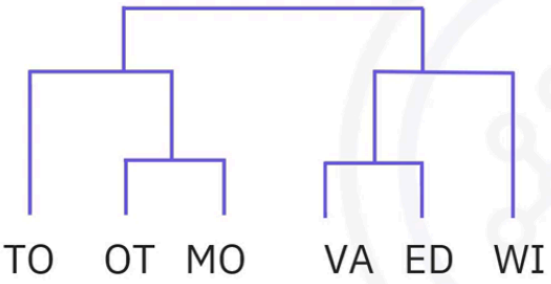
Clustering (Unsupervised)	Classification (Supervised)
No labels available	Requires labeled data
Groups data into clusters	Predicts predefined categories
Example: Group customers by spending	Example: Predict loan default status

- **Types of Clustering Methods**





Types of clustering



Hierarchical clustering

- Organizes data into nested clusters
- Contains smaller sub-clusters
- Generates a dendrogram
- Reveals relationships between clusters

1. **Partition-based Clustering**
    - Divides data into **non-overlapping groups**.
    - Example: **K-Means Algorithm**.
    - Efficient, scalable, works well with large datasets.
    - Limitation: struggles with **irregular shapes**.
2. **Density-based Clustering**
    - Groups points based on **density of data points**.
    - Can detect **arbitrary shaped clusters**.
    - Handles **noise and outliers** well.
    - Example: **DBSCAN**.
3. **Hierarchical Clustering**
    - Builds a **tree of nested clusters** (dendrogram).
    - Suitable for small–mid-sized datasets.
    - Two main approaches:

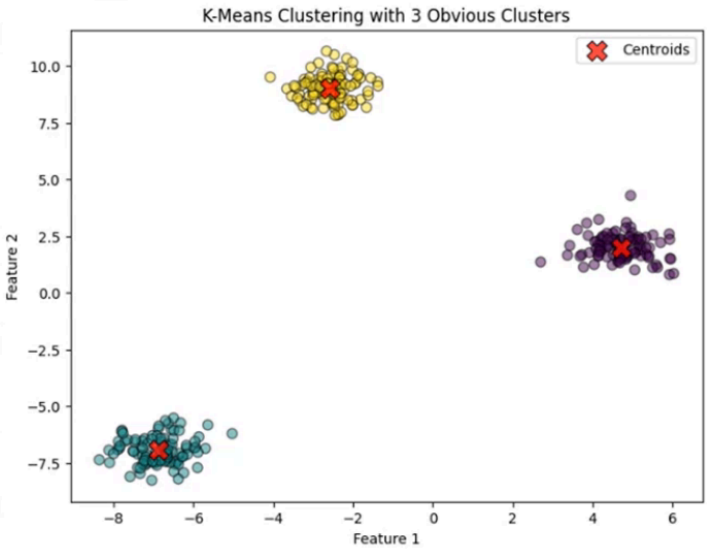
- **Agglomerative (Bottom-Up)**: Start with individual points → merge closest clusters.
- **Divisive (Top-Down)**: Start with one big cluster → split into smaller clusters.

Partition-based clustering

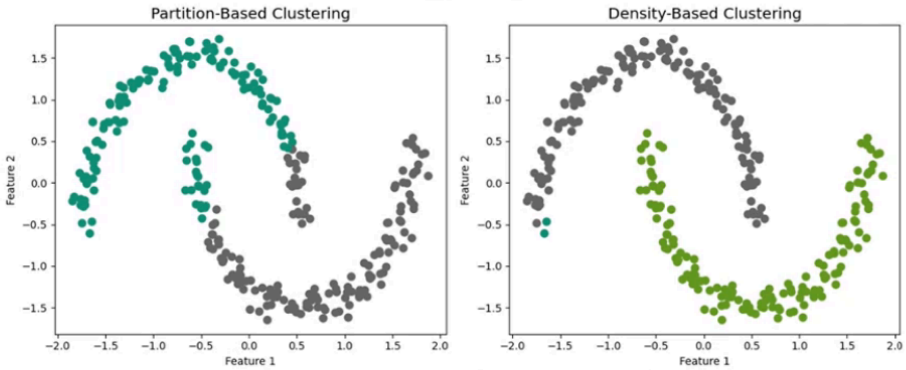
Uses the "make blobs" function

Generates three color-coded clusters

Displays clusters in a scatterplot



Partition and density-based clustering



Uses the "make moons" function

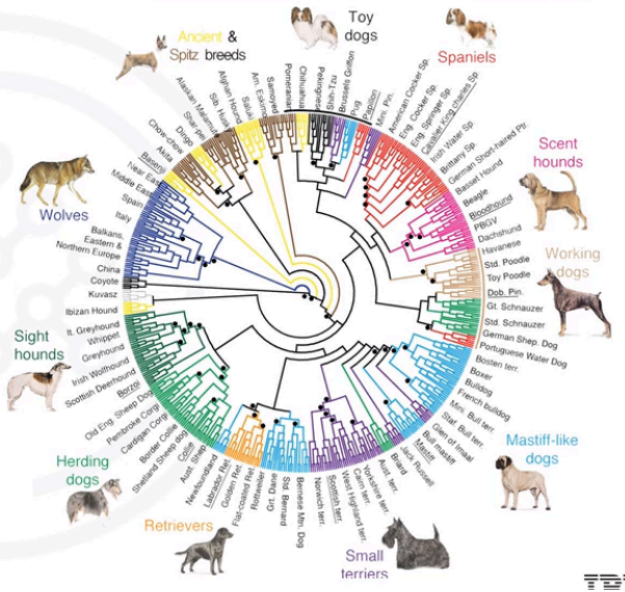
Generates interlocking half-circles

Distinguishes clusters using color

3:26

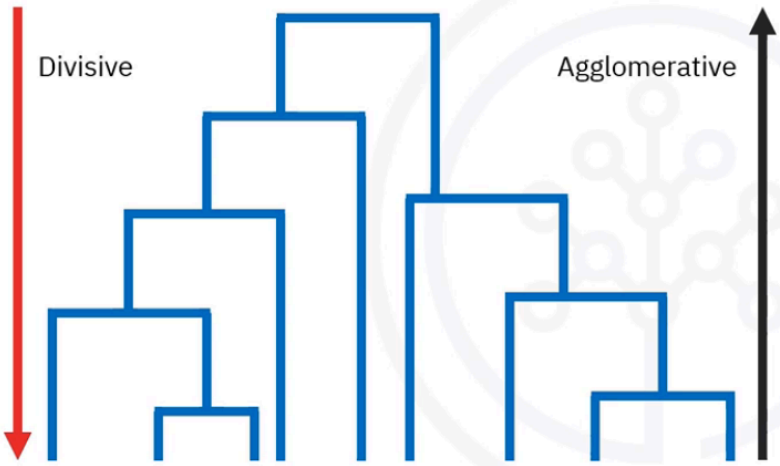
Hierarchical clustering

Presents genetic data from over 900 dogs
Analyzed 48,000 genetic markers
Illustrates hierarchical clustering of animal groups



- **Hierarchical Clustering Process**

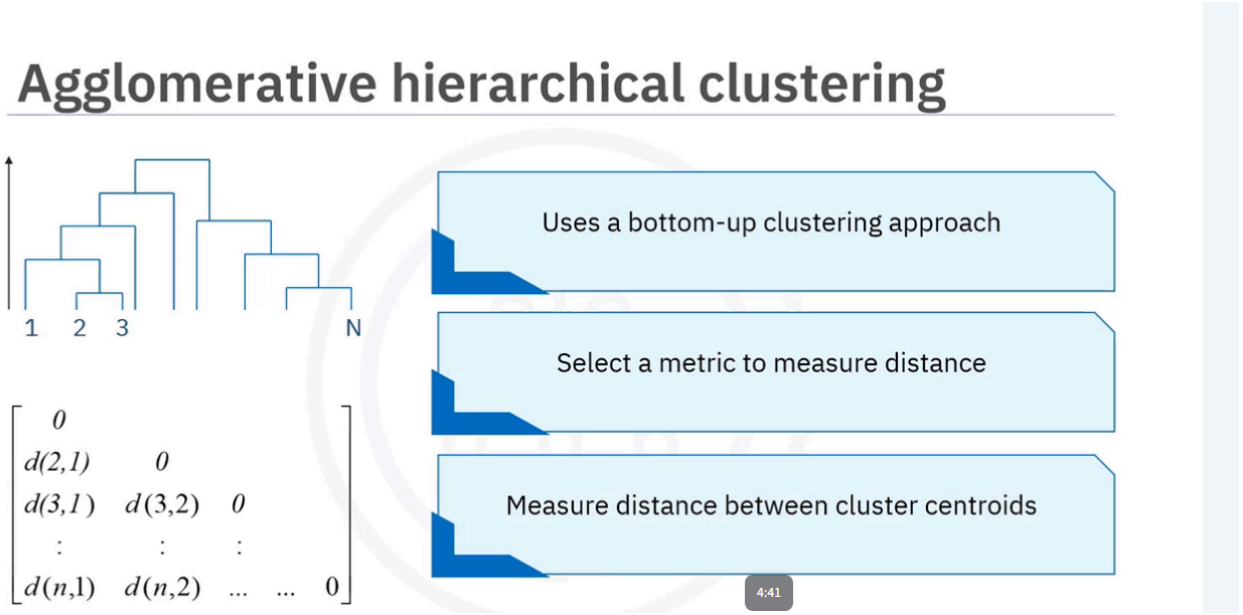
Hierarchical clustering



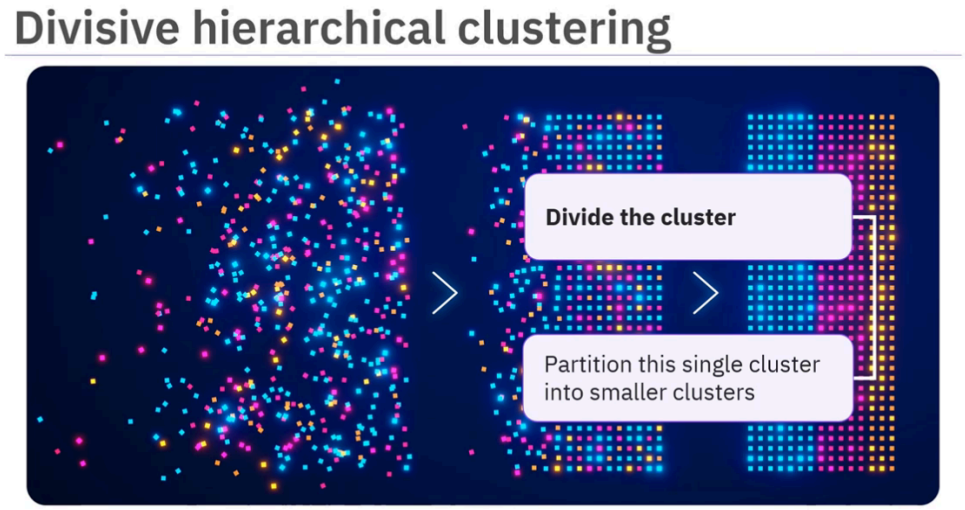
Divisive clustering

- Uses a top-down approach
- Starts with a single root cluster
- Iteratively splits into smaller child clusters

Agglomerative (Bottom-Up):



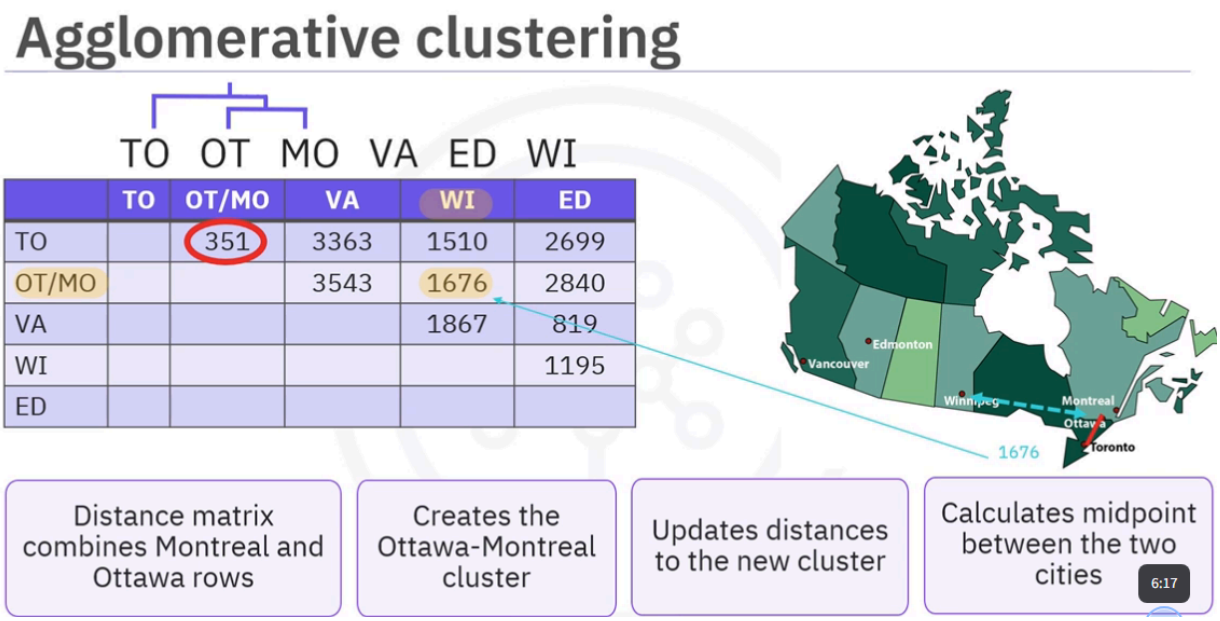
1. Start with **N clusters** (each data point = 1 cluster).
2. Compute a **distance matrix** (pairwise distances).
3. Merge the two closest clusters.
4. Update the matrix with new distances.
5. Repeat until the desired number of clusters is reached.
   👉 Output: **Dendrogram** showing cluster hierarchy.



Divisive (Top-Down):

1. Start with **all data points in one cluster**.
2. Split into smaller clusters based on similarities/dissimilarities.
3. Continue splitting until a **stopping condition** (e.g., min cluster size) is reached.

🔹 Examples in Practice

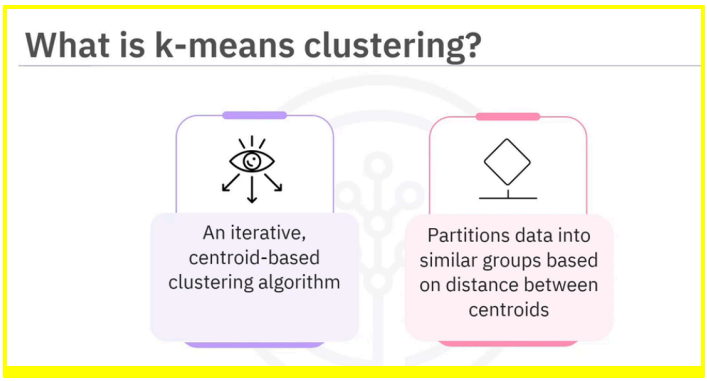


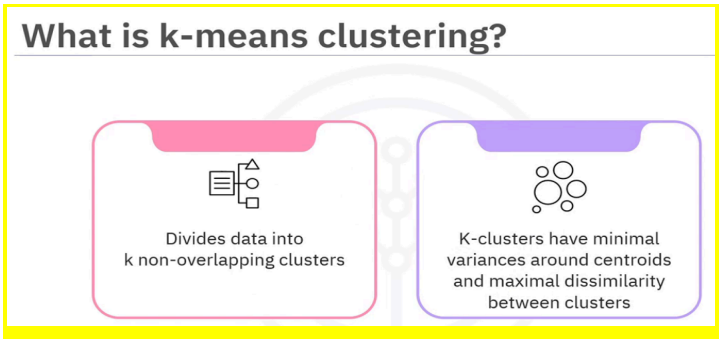
- **K-Means (Partition-Based):** Clear, non-overlapping clusters.
- **DBSCAN (Density-Based):** Works well with irregular shapes but may create unnecessary clusters.
- **Hierarchical Clustering (Agglomerative/Divisive):** Visualized with dendrograms (e.g., grouping dog breeds by genetic data).

✅ Key Takeaways

- Clustering = unsupervised method for grouping unlabeled data.
- Main algorithms: **K-Means, DBSCAN, Hierarchical (Agglomerative & Divisive)**.
- Useful for **customer segmentation, anomaly detection, image recognition, and feature engineering**.
- Choice of method depends on **data size, shape, and noise**.

📌 K-Means Clustering

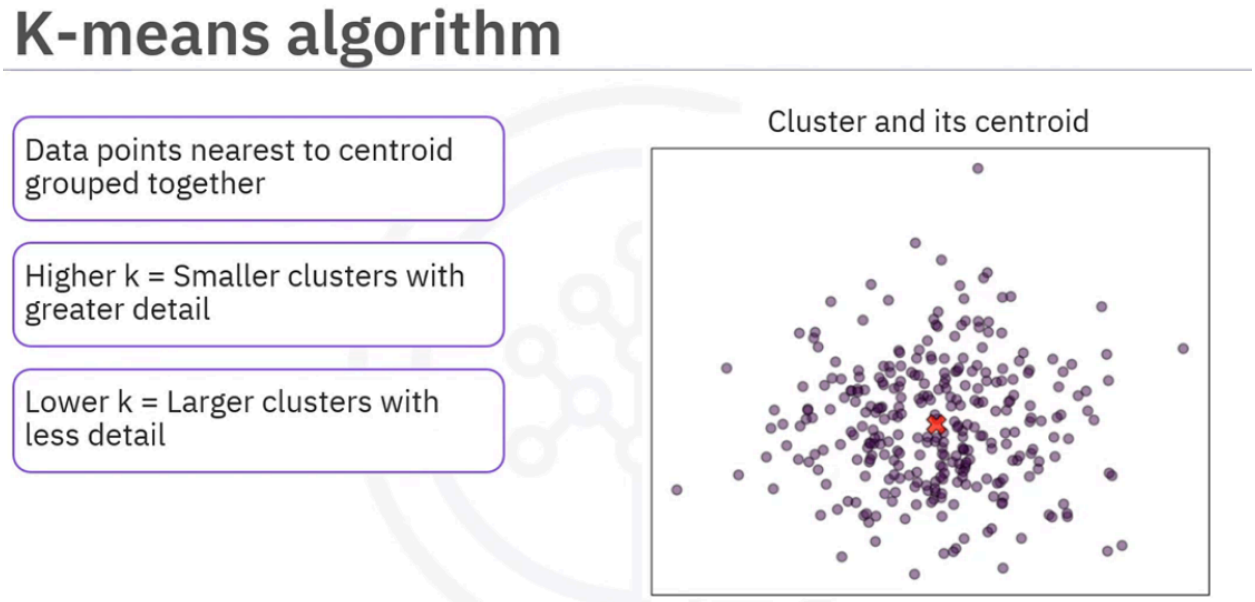




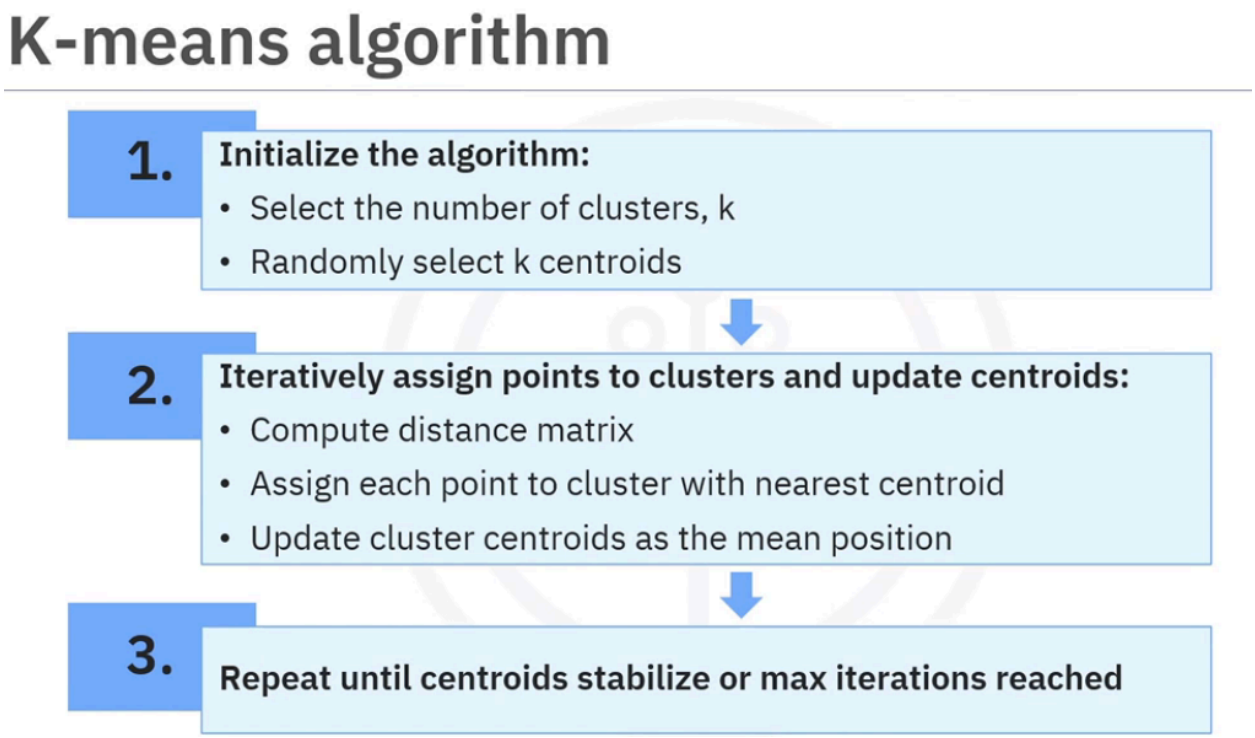
🔹 What is K-Means?

- **Definition**: An **iterative, centroid-based clustering algorithm**.
- **Goal**: Partition the dataset into *k* non-overlapping clusters with:
  - **Low variance** within clusters.
  - **High dissimilarity** between clusters.

- **Centroid (μ)**: Average position of all points in a cluster (marked as X).
- **Parameter (k)**: Number of clusters (chosen before running the algorithm).



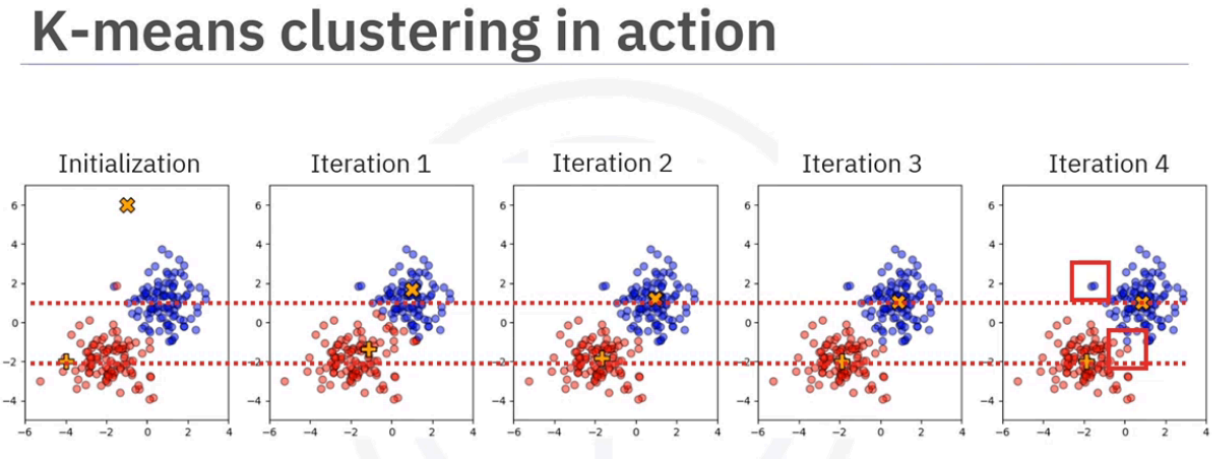
🔹 **How K-Means Works (Algorithm Steps)**

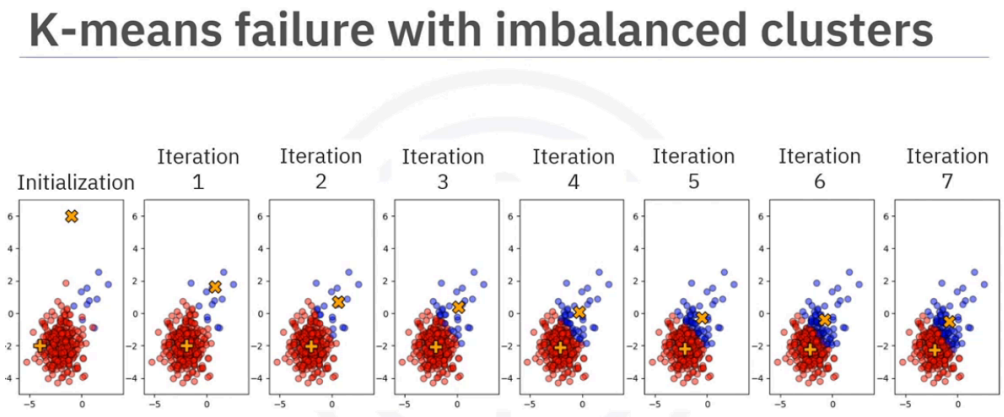


1. **Initialize**: Choose *k* clusters and select random centroids.
2. **Assign points**: Each data point → nearest centroid (distance measure).
3. **Update centroids**: Compute mean of all points in a cluster.
4. **Repeat**: Reassign points and update centroids until convergence.
5. **Convergence**: When centroids stop moving (or max iterations reached).

🔹 Characteristics of K

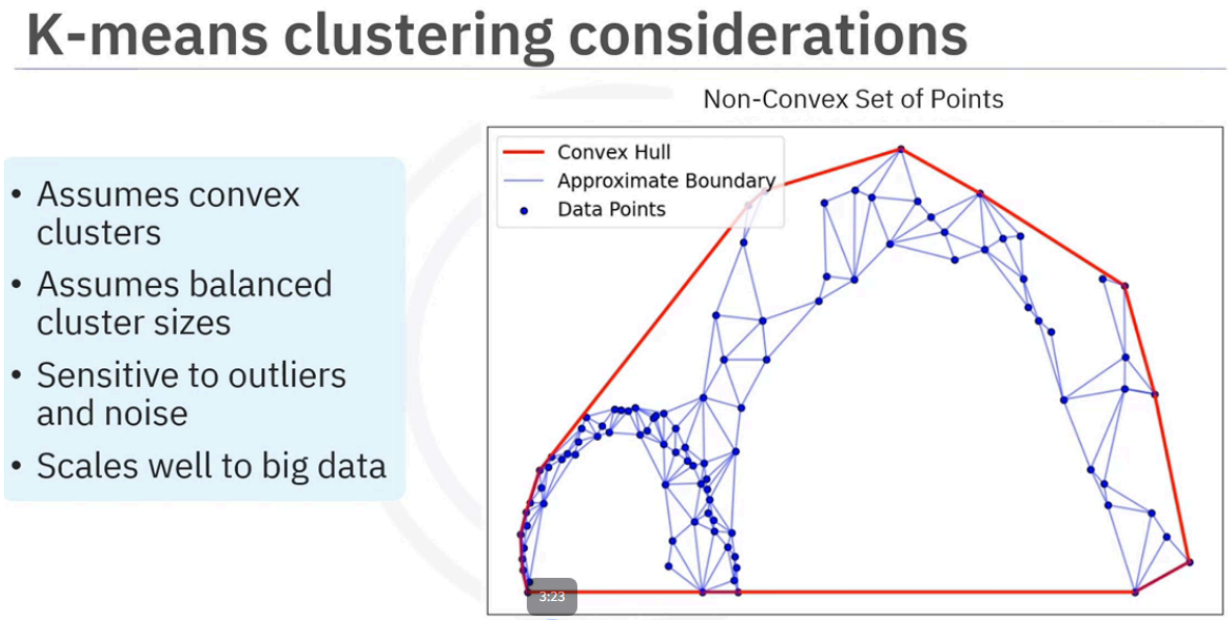
- Higher **k** → smaller, more detailed clusters.
- Lower **k** → larger, less detailed clusters.
- Choosing **right k** is critical (not always obvious).





🔹 Strengths of K-Means

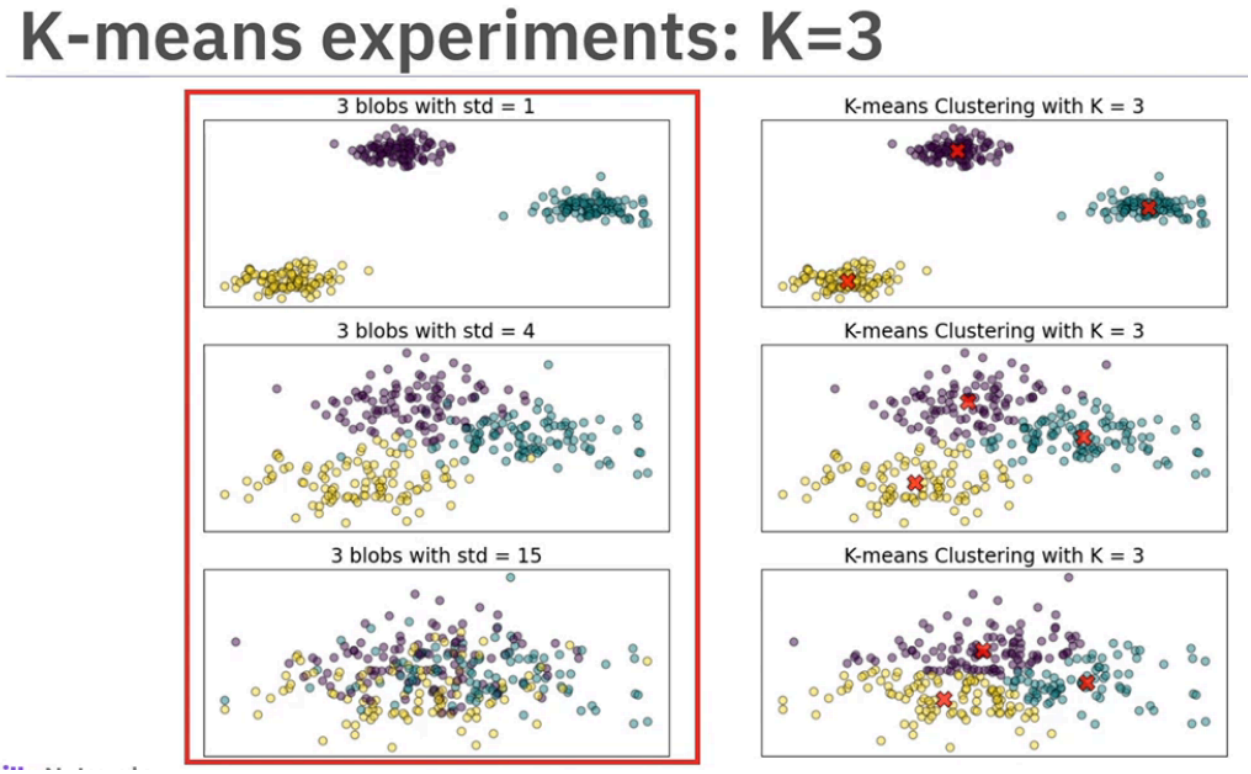
- Efficient and scales well for **big data**.
- Simple, easy to implement.
- Works well on **convex-shaped clusters** with similar sizes.



🔹 Limitations of K-Means

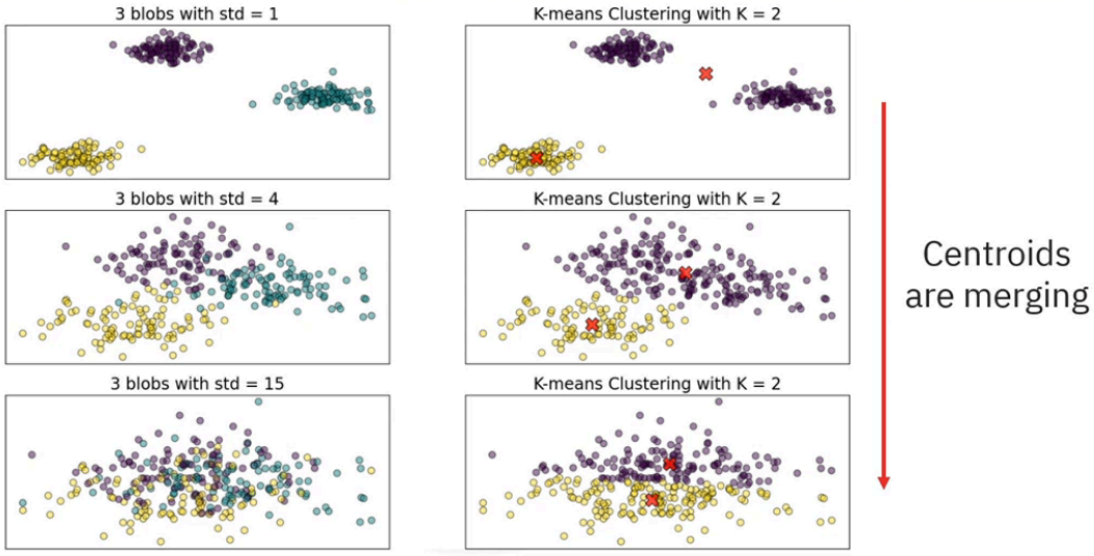
- Assumes clusters are **convex & spherical**.
- Struggles with **imbalanced clusters** (e.g., one large, one small).
- Sensitive to **noise & outliers**.
- Poor performance when clusters overlap significantly.

🔹 **Performance Examples**



- **Low variance (SD = 1–4)** → K-Means separates blobs well.
- **High variance (SD = 15)** → blobs overlap, clustering fails.
- **K ≠ true number of clusters**:
    - If *k* too small → merges clusters incorrectly.
    - If *k* too large → creates meaningless extra clusters.

K-means experiments: K=2



-

- **Objective Function**

K-means optimization

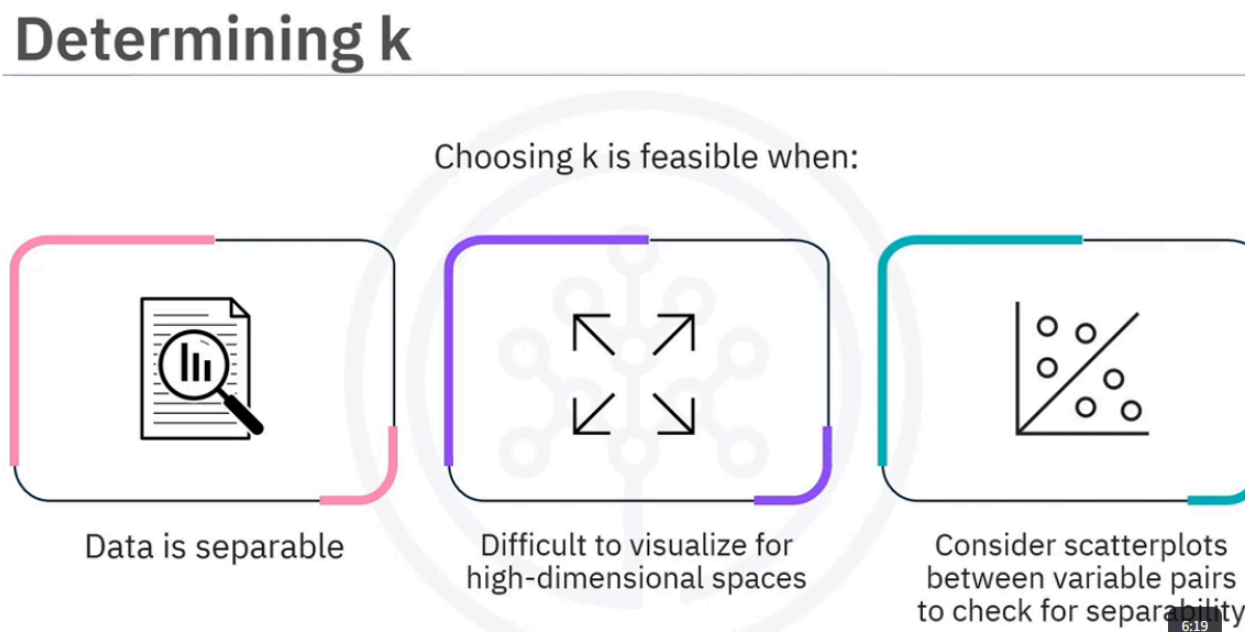
Goal: Minimize within-cluster sum of squares:

$$\sum_{i=1}^{K} \sum_{x \in C_i} \|x - \mu_i\|^2$$

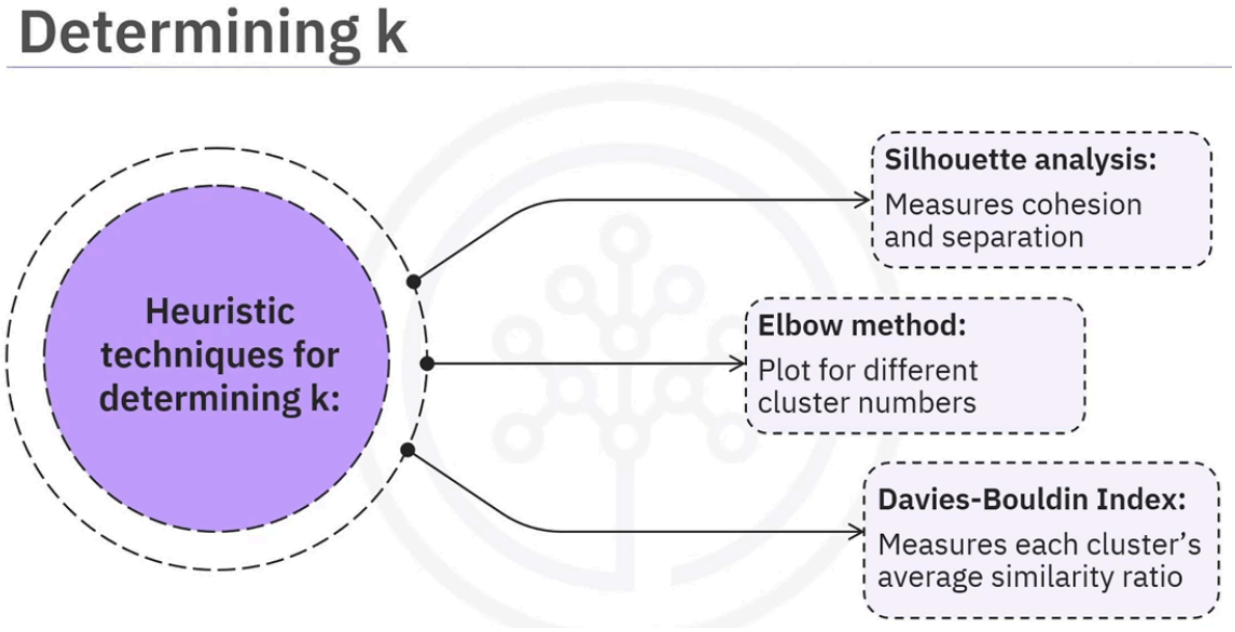
- K = Number of clusters
- C_i = i th cluster
- x = Data point
- μ_i = Centroid of cluster C_i
- $\|x - \mu_i\|^2$ = Squared distance between x and its cluster centroid

- Minimize **within-cluster sum of squared distances (WCSS)**:
  ∑i=1k∑x∈Ci||x−μi||2\sum_{i=1}^k \sum_{x \in C_i} ||x - \mu_i||^2

🔹 **Choosing the Best K**



- **Elbow Method**: Plot WCSS vs. k, choose point where curve bends.
- **Silhouette Analysis**: Measures cohesion vs. separation of clusters.
- **Davies-Bouldin Index**: Lower values = better clustering.



✅ **Key Takeaways**

- K-Means = centroid-based, efficient clustering.
- Works best for **balanced, convex clusters**.
- Sensitive to noise, outliers, and choice of *k*.
- Use heuristic methods (Elbow, Silhouette, DB Index) to find optimal *k*.

📘 K-Means Customer Segmentation Notes

1. Introduction

- **K-means** is one of the most popular clustering algorithms in machine learning.
- It is used when labels are not available (**unsupervised learning**).
- **Applications**:
    - Customer segmentation
    - Website visitor behavior analysis
    - Pattern recognition
    - Feature engineering
    - Data compression

---

2. Install & Import Libraries

!pip install numpy pandas matplotlib scikit-learn plotly

import numpy as np

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler
import plotly.express as px

%matplotlib inline
import warnings
warnings.filterwarnings('ignore')
```

---

3. K-Means on Synthetic Data

Generate Data

```
np.random.seed(0)

X, y = make_blobs(
    n_samples=5000,
    centers=[[4,4], [-2,-1], [2,-3],[1,1]],
    cluster_std=0.9
)

plt.scatter(X[:, 0], X[:, 1], marker='.', alpha=0.3, edgecolors='k', s=80)
plt.show()
```

Train K-Means

```
k_means = KMeans(init="k-means++", n_clusters=4, n_init=12)

k_means.fit(X)


labels = k_means.labels_

centers = k_means.cluster_centers_
```

Visualize Clusters

```
fig = plt.figure(figsize=(6, 4))

ax = fig.add_subplot(1, 1, 1)

colors = plt.cm.tab10(np.linspace(0, 1, len(set(labels))))


for k, col in zip(range(len(centers)), colors):

    members = (labels == k)

    cluster_center = centers[k]

    ax.plot(X[members, 0], X[members, 1], 'w', markerfacecolor=col, marker='.', ms=10)

    ax.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,

            markeredgecolor='k', markersize=6)


ax.set_title("K-Means Clustering")

plt.show()
```

---

4. Exercises (Synthetic Data)

- **Change number of clusters (k=3, k=5) and visualize results.**

```
k_means3 = KMeans(init="k-means++", n_clusters=3, n_init=12).fit(X)

k_means5 = KMeans(init="k-means++", n_clusters=5, n_init=12).fit(X)
```

- **Observation**:
    - k=3 → higher inertia (underfitting).
    - k=5 → lower inertia (overfitting).

---

5. Customer Segmentation Example

Load Dataset

```
cust_df = pd.read_csv(
    "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/"
"IBMDeveloperSkillsNetwork-ML0101EN-SkillsNetwork/labs/Module%204/data/Cust_Segmentation.csv"
)

cust_df = cust_df.drop('Address', axis=1)  # Remove categorical feature
cust_df = cust_df.dropna()  # Remove missing values
cust_df.info()
```

Normalize Data

```
X = cust_df.values[:,1:]   # Exclude Customer ID
```

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

Apply K-Means

```
clusterNum = 3

k_means = KMeans(init="k-means++", n_clusters=clusterNum, n_init=12)

k_means.fit(X_scaled)

labels = k_means.labels_


cust_df["Clus_km"] = labels
```

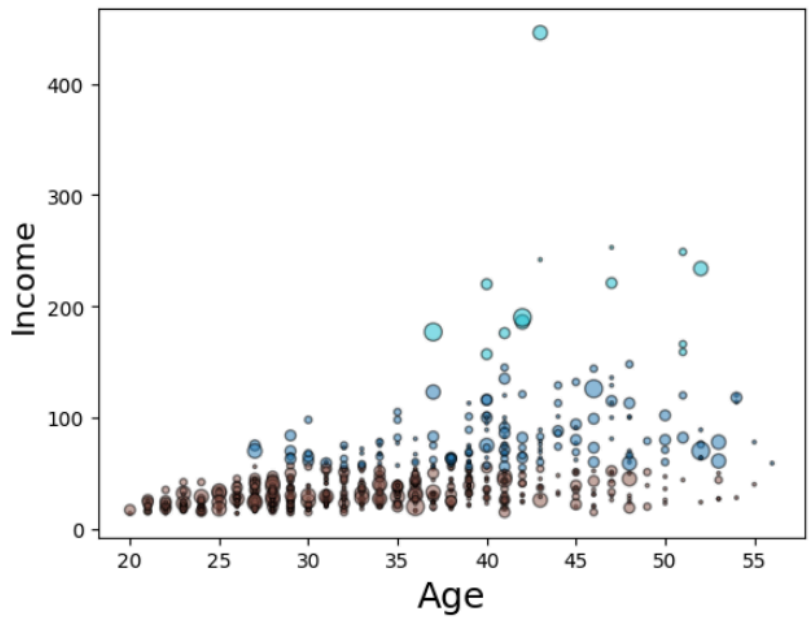
Cluster Centroids

```
cust_df.groupby("Clus_km").mean()
```

---

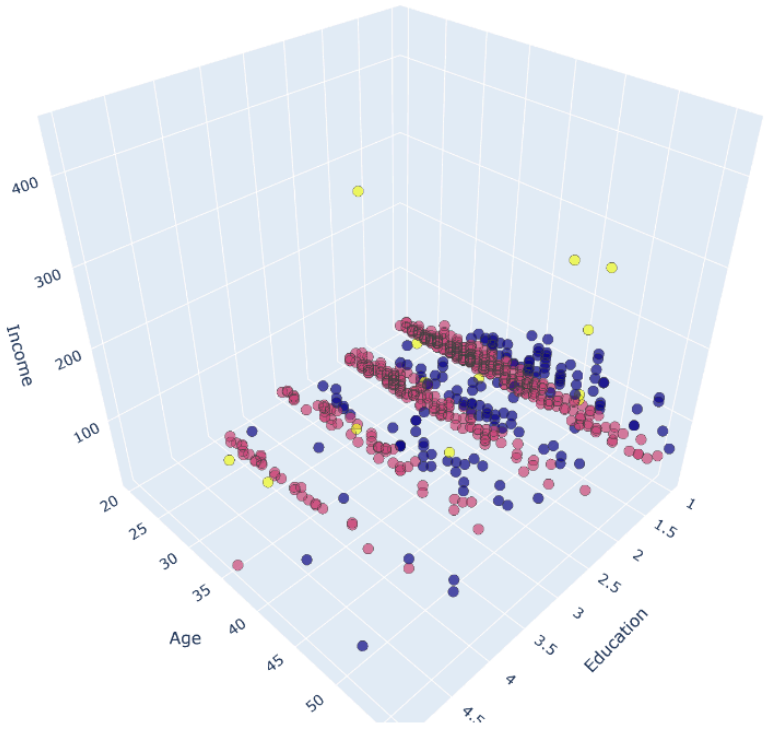
6. Visualizations

2D Scatter Plot



```
area = np.pi * (X[:, 1])**2
plt.scatter(X[:, 0], X[:, 3], s=area, c=labels.astype(float), cmap='tab10',
            edgecolors='k', alpha=0.5)
plt.xlabel("Age", fontsize=18)
plt.ylabel("Income", fontsize=16)
plt.show()
```

3D Scatter Plot




```
fig = px.scatter_3d(X, x=1, y=0, z=3, opacity=0.7, color=labels.astype(float))
fig.update_traces(marker=dict(size=5, line=dict(width=.25)), showlegend=False)
fig.update_layout(
   coloraxis_showscale=False, width=1000, height=800,
   scene=dict(
```

```
        xaxis=dict(title='Education'),
        yaxis=dict(title='Age'),
        zaxis=dict(title='Income')
    )
)
fig.show()
```

---

7. Insights (Cluster Profiles)

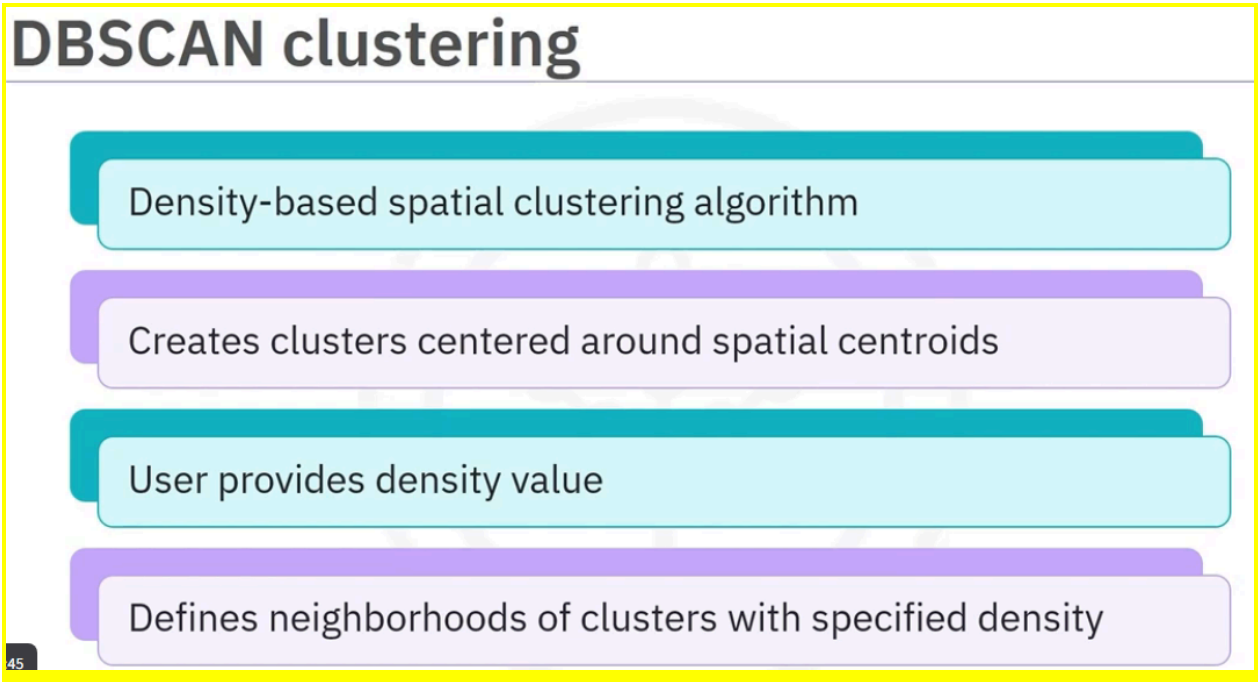
- **Cluster 0**: Late career, affluent, and educated.
- **Cluster 1**: Mid-career, middle income.
- **Cluster 2**: Early career, low income.

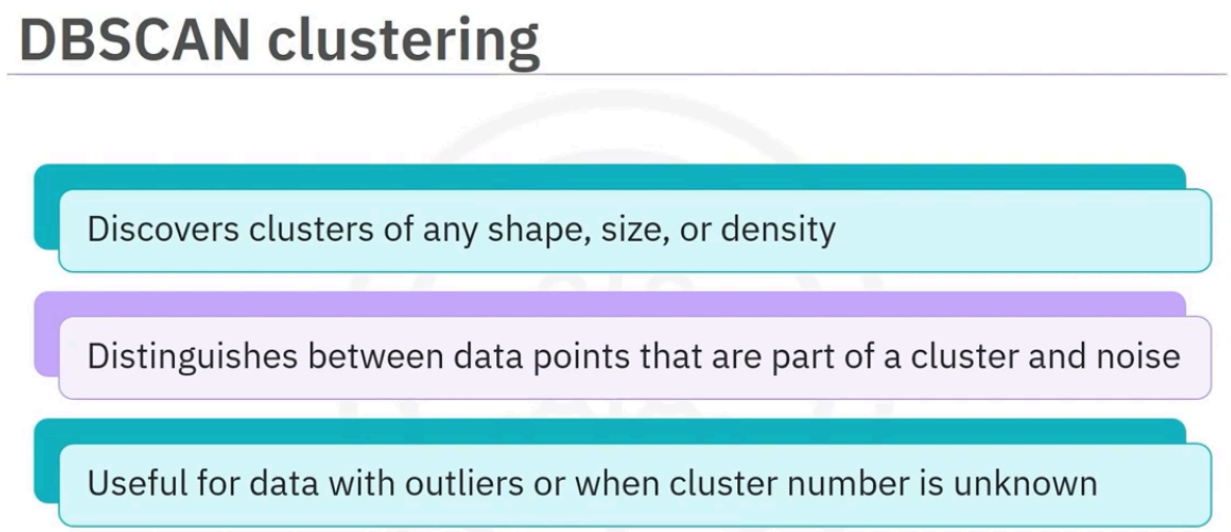
✅ **Takeaways**:

- K-means groups similar customers together → allows businesses to personalize strategies.
- Choosing k is critical (under/overfitting).
- Feature scaling (StandardScaler) is essential.
- Visualization (2D/3D) helps interpret clusters.

Here's a **clean, structured summary** of your DBSCAN and HDBSCAN clustering lesson:

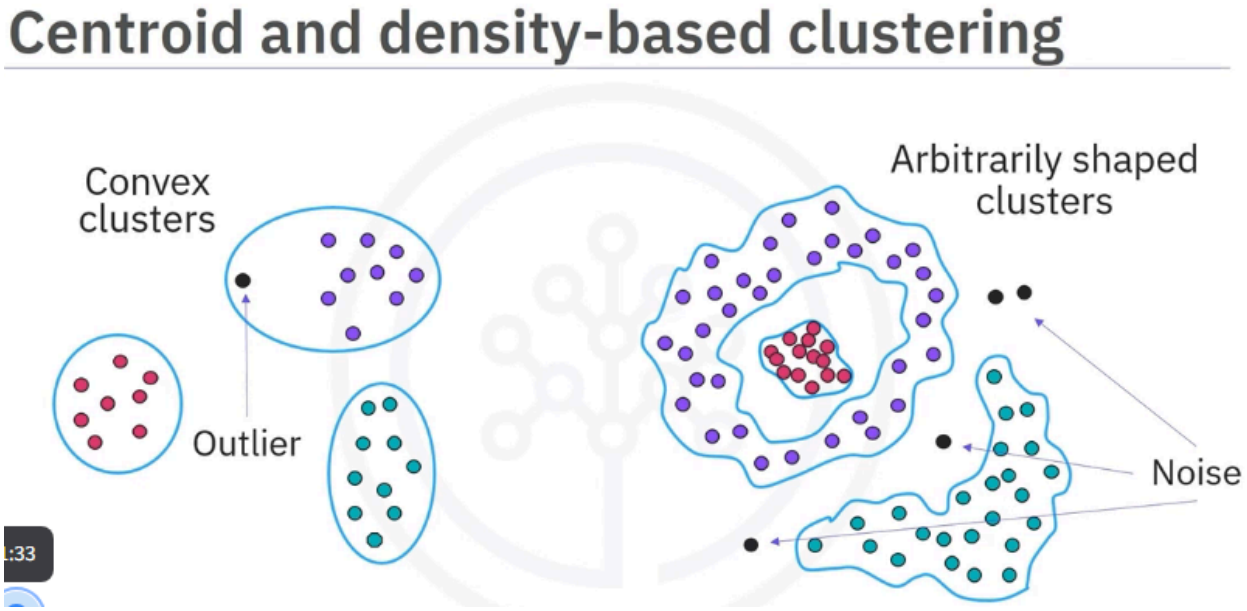
📌 DBSCAN & HDBSCAN Clustering





- **DBSCAN (Density-Based Spatial Clustering of Applications with Noise)**
  - **Type**: Density-based clustering algorithm.
  - **Key Idea**: Groups together points in **dense regions**; marks sparse points as **noise/outliers**.
  - **Advantages**:

- Finds clusters of **any shape/size**.
- Handles **noise and outliers**.
- Useful when the number of clusters is **unknown**.



🔹 Parameters in DBSCAN

1. **Epsilon (ε)** → radius of neighborhood around a point.
2. **MinPts (n)** → minimum number of points required in an ε-neighborhood.

🔹 Point Types in DBSCAN

- **Core Point** → has ≥ MinPts within ε.
- **Border Point** → lies within a core point's neighborhood but has < MinPts.
- **Noise Point** → neither a core nor border point.

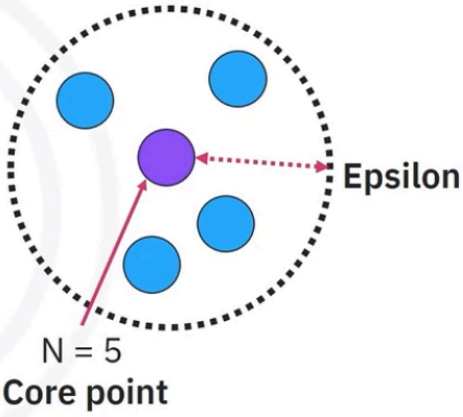
🔹 DBSCAN Algorithm Steps

DBSCAN algorithm

Parameters: N, epsilon

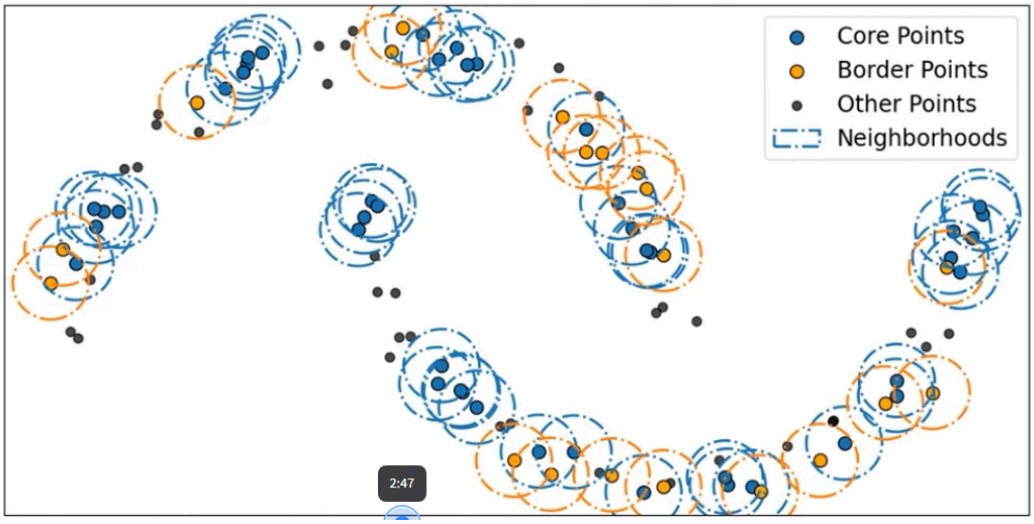
Every data point is labelled as one of the following:

- **Core point:** Has at least N points within the epsilon neighborhood
- **Border point:** Non-core points within a core-point neighborhood
- **Noise point:** Isolated from all core-point neighborhoods
- **Clusters:** Core points and border points
- **Noise:** Non-clustered points



1. For each point, check neighbors within ε.
2. Classify as **core, border, or noise**.
3. Grow clusters from core points → include neighbors recursively.
4. Border points join clusters; noise points remain unassigned.
5. Clustering is done **in one pass** (not iterative).

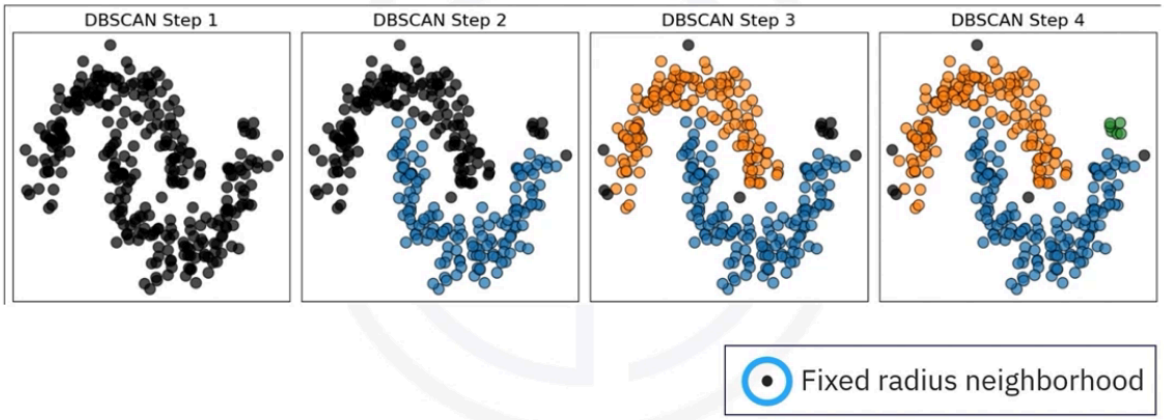
Core and border points

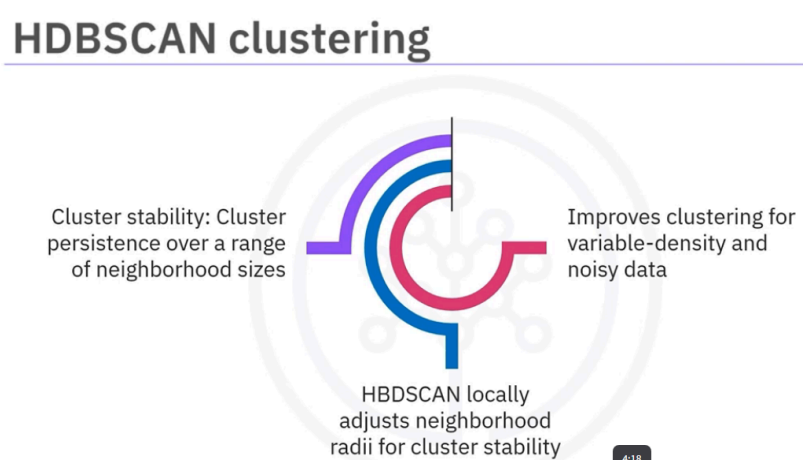


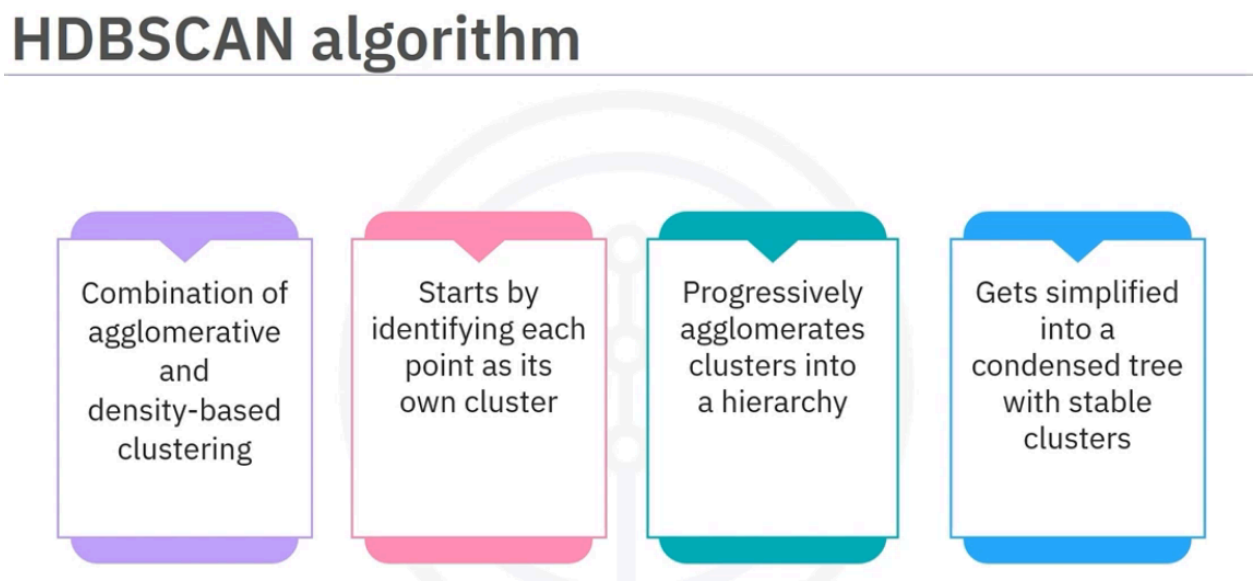
DBSCAN experiment

6.

DBSCAN experiment







🔹 HDBSCAN (Hierarchical DBSCAN)

- **Improvement over DBSCAN**.
- **Key Features**:
  - Doesn’t require manual setting of ε.
  - Less sensitive to **noise/outliers**.
  - Uses **cluster stability** → measures persistence of clusters across density levels.
- **How it works**:

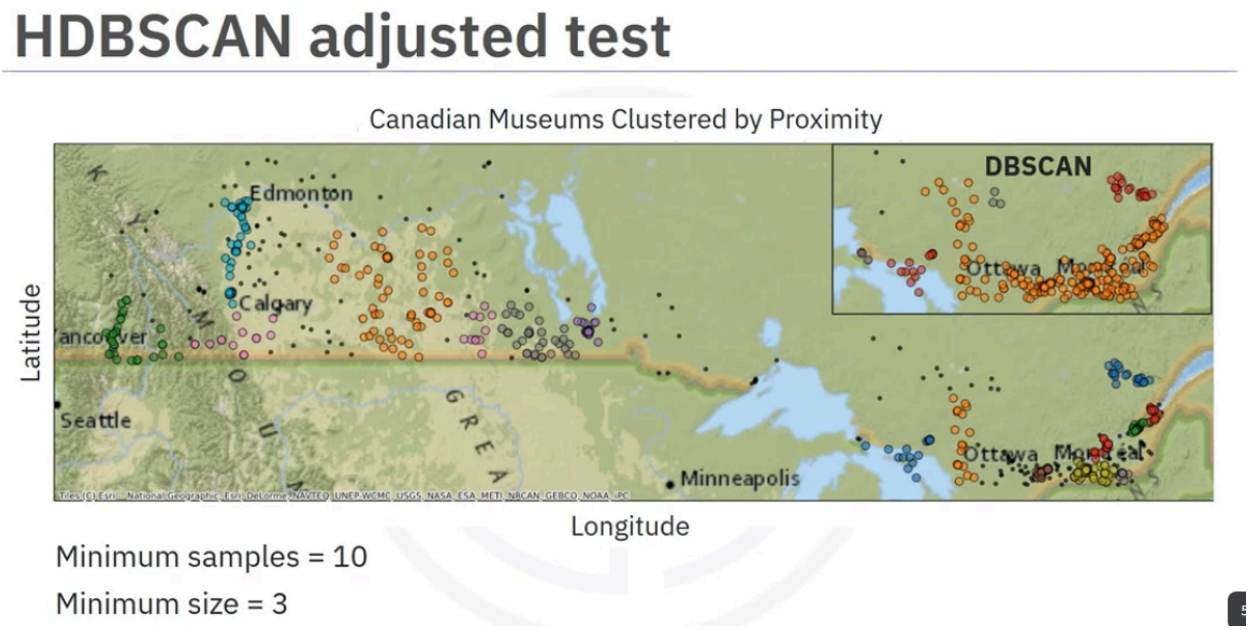
- Starts with each point as its own cluster (like agglomerative clustering).
- Gradually merges clusters by lowering density thresholds.
- Builds a **hierarchical tree**, then condenses it → keeping only the most stable clusters.

🔹 DBSCAN vs HDBSCAN

Feature	DBSCAN	HDBSCAN
Parameters	Requires ε + MinPts	Requires only MinPts + min cluster size
Shape Handling	Any shape, but fixed density	Any shape, adaptive density
Noise Sensitivity	More sensitive	More robust
Output	Flat clusters	Hierarchical clusters (tree, condensed to stable ones)
Use Case	Works well if density is uniform	Better for varying densities

---

🔹 **Real-World Example**



- **DBSCAN** on Canadian museum dataset → ~10 clusters, but dense eastern regions lumped together.
- **HDBSCAN** → more refined clusters, distinguished curved patterns, less noisy, adaptive to local density.

✅ **Key Takeaways**

- **DBSCAN**: density-based, requires ε + MinPts, good for arbitrary shapes, struggles with varying densities.
- **HDBSCAN**: hierarchical, adaptive, more robust, requires fewer parameters, produces more meaningful clusters.

Here are clean, structured **study notes with working code snippets** for the lab *Comparing DBSCAN and HDBSCAN clustering*. I've organized them step by step for quick revision and practice.

📘 Comparing DBSCAN and HDBSCAN Clustering

🎯 Objectives

By the end of this lab, you will be able to:

- Implement **DBSCAN** and **HDBSCAN** clustering models using `scikit-learn` and `hdbscan`.
- Apply clustering on real-world geospatial data (Canadian museums).
- Visualize clusters on a **Canada basemap**.
- Compare model performances.

📖 Introduction

- **Dataset:** The Open Database of Cultural and Art Facilities (ODCAF)
  Source: StatCan ODCAF
  Data includes **names, types, and locations** of cultural/art facilities across Canada.
- We will focus on **museums** only.
- Two clustering models:
  - **DBSCAN (Density-Based Spatial Clustering of Applications with Noise)**
  - **HDBSCAN (Hierarchical DBSCAN)**

🛠️ Step 1. Import Libraries

```
import numpy as np
```

```
import pandas as pd

import matplotlib.pyplot as plt


from sklearn.cluster import DBSCAN

import hdbscan

from sklearn.preprocessing import StandardScaler


# Geospatial tools

import geopandas as gpd

import contextily as ctx

from shapely.geometry import Point


import warnings

warnings.filterwarnings('ignore')
```

🗺️ Step 2. Canada Map Download

```
import requests, zipfile, io, os


zip_file_url = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/YcUk-ytgrPkmvZAh5bf7zA/Canada.zip'

output_dir = './'

os.makedirs(output_dir, exist_ok=True)
```

```
response = requests.get(zip_file_url)
with zipfile.ZipFile(io.BytesIO(response.content)) as zip_ref:
    for file_name in zip_ref.namelist():
        if file_name.endswith('.tif'):
            zip_ref.extract(file_name, output_dir)
            print(f"Downloaded and extracted: {file_name}")
```

🎨 Step 3. Helper Function (Plot Clusters)

```
def plot_clustered_locations(df, title='Museums Clustered by Proximity'):
    gdf = gpd.GeoDataFrame(
        df, geometry=gpd.points_from_xy(df['Longitude'], df['Latitude']), crs="EPSG:4326"
    ).to_crs(epsg=3857)

    fig, ax = plt.subplots(figsize=(15, 10))
    non_noise = gdf[gdf['Cluster'] != -1]
    noise = gdf[gdf['Cluster'] == -1]

    noise.plot(ax=ax, color='k', markersize=30, ec='r', label='Noise')
    non_noise.plot(ax=ax, column='Cluster', cmap='tab10', markersize=30, ec='k', alpha=0.6)

    ctx.add_basemap(ax, source='./Canada.tif', zoom=4)
    plt.title(title)
    plt.xlabel('Longitude'); plt.ylabel('Latitude')
```

```
    ax.set_xticks([]); ax.set_yticks([])

    plt.show()
```

📊 Step 4. Load and Explore Dataset

```
url = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/r-maSj5Yegvw2sJraT15FA/ODCAF-v1-0.csv'

df = pd.read_csv(url, encoding="ISO-8859-1")

df.head()
```

Data Cleaning

- Missing values: Represented by `'..'` or `NaN`.
- Facility types:

```
df.ODCAF_Facility_Type.value_counts()
```

Keep only museums

```
df = df[df.ODCAF_Facility_Type == 'museum']
```

Select coordinates

```
df = df[['Latitude', 'Longitude']]


# Convert to float & drop invalids

df['Latitude'] = pd.to_numeric(df['Latitude'], errors='coerce')
```

```
df['Longitude'] = pd.to_numeric(df['Longitude'], errors='coerce')

df = df.dropna(subset=['Latitude', 'Longitude'])

df[['Latitude','Longitude']] = df[['Latitude','Longitude']].astype(float)
```

🤖 Step 5. DBSCAN Model

Scaling

```
coords = df[['Latitude', 'Longitude']]

coords_scaled = StandardScaler().fit_transform(coords)
```

Apply DBSCAN

```
dbscan = DBSCAN(eps=1.0, min_samples=3, metric='euclidean')

df['Cluster'] = dbscan.fit_predict(coords_scaled)


print(df['Cluster'].value_counts())

plot_clustered_locations(df, title="DBSCAN Clustering of Museums")
```

🌳 Step 6. HDBSCAN Model

```
hdb = hdbscan.HDBSCAN(min_cluster_size=15, metric='euclidean')

df['Cluster'] = hdb.fit_predict(coords_scaled)


print(df['Cluster'].value_counts())
```

```
plot_clustered_locations(df, title="HDBSCAN Clustering of Museums")
```

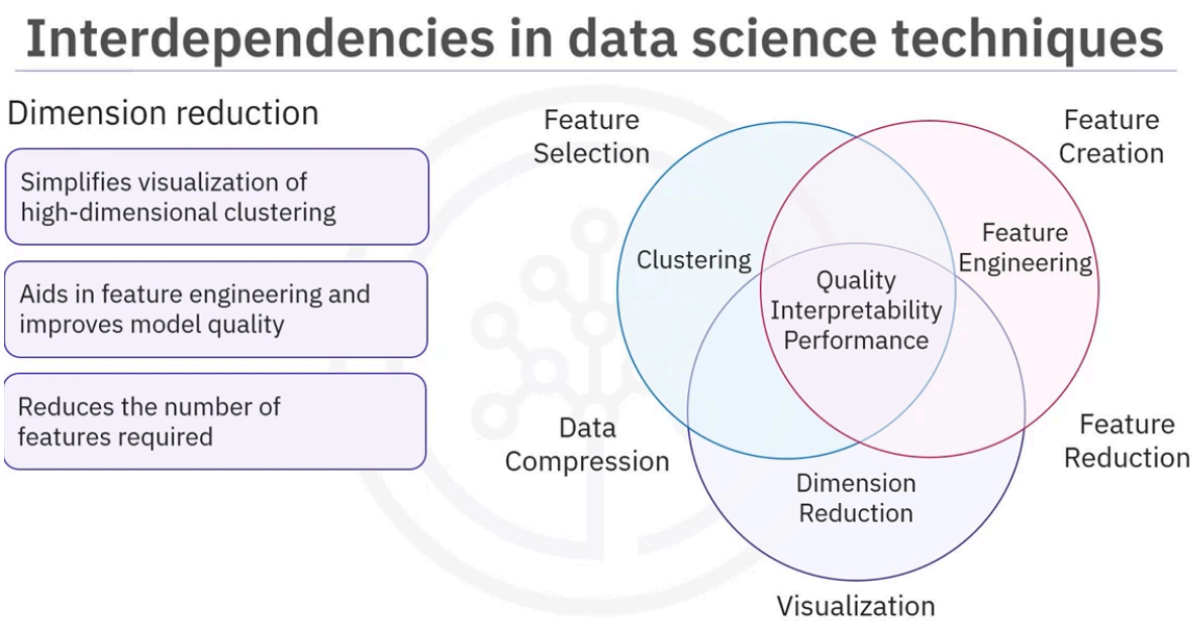
📈 Step 7. Comparison DBSCAN vs HDBSCAN

- **DBSCAN**
  - Requires tuning `eps` and `min_samples`.
  - May misclassify noise if density varies.
- **HDBSCAN**
  - Automatically adapts to **variable densities**.
  - Produces more natural clusters, fewer parameters.

✅ Key Takeaways

- Both DBSCAN and HDBSCAN are **density-based clustering methods**.
- DBSCAN works well when density is uniform; HDBSCAN is better for real-world noisy data.
- Visualizations on the **Canada basemap** make cluster patterns easier to interpret.

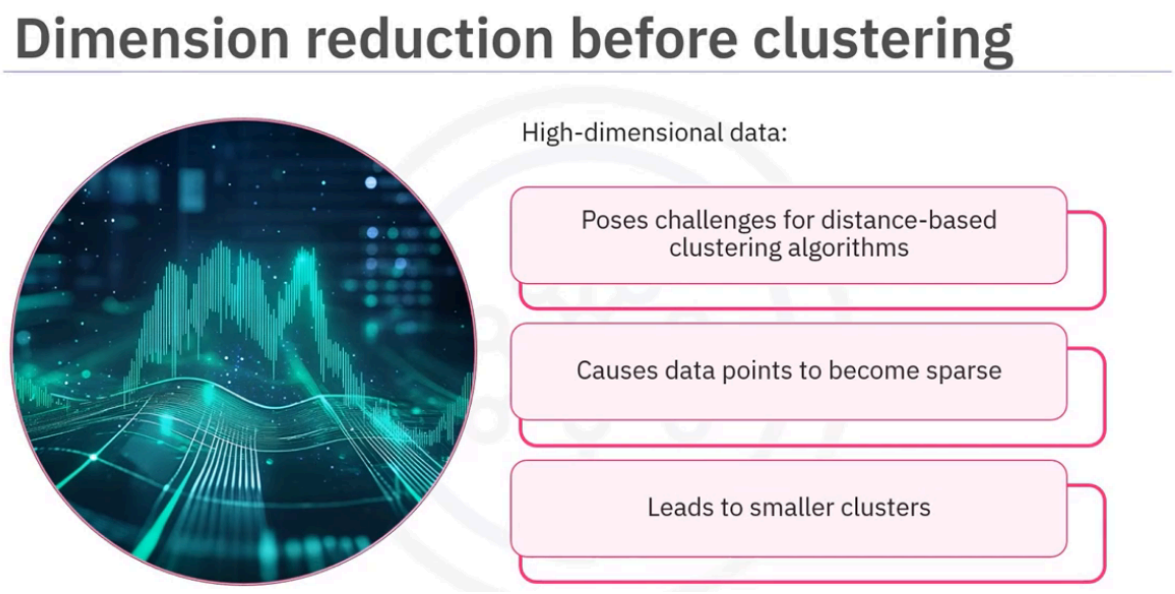
📌 Clustering, Dimension Reduction & Feature Engineering



🔹 Why They Work Together

- **Clustering** → helps in **feature selection/creation**, supports dimension reduction, improves scalability.
- **Dimension Reduction** → simplifies **visualization & computation**, enhances clustering, improves model quality.
- **Feature Engineering** → uses insights from clustering & reduction to build better predictive features.

1️⃣ Clustering



- Groups similar data points or features.
- **Applications**:
    - Feature selection (choose one representative from redundant features).
    - Feature engineering (create transformations or interactions if clusters reveal subgroups).

Example:

- K-means applied on features (not data) → features with the same mean/variance fall in one cluster → only one representative feature is kept.

2️⃣ Dimension Reduction

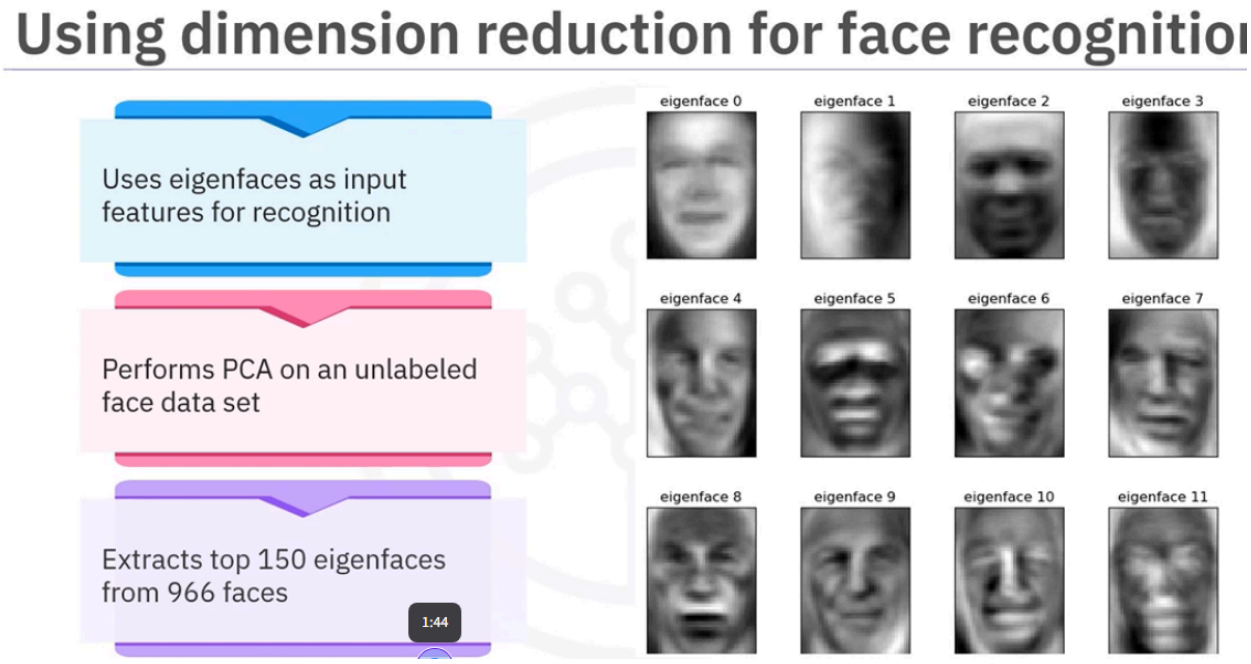
- Reduces the number of features while keeping **key information**.
- Needed because:

- High-dimensional data → sparse, less similar → distance-based clustering (K-means, DBSCAN) fails.
- Improves visualisation (2D/3D scatter plots of high-D clusters).

🔑 Techniques

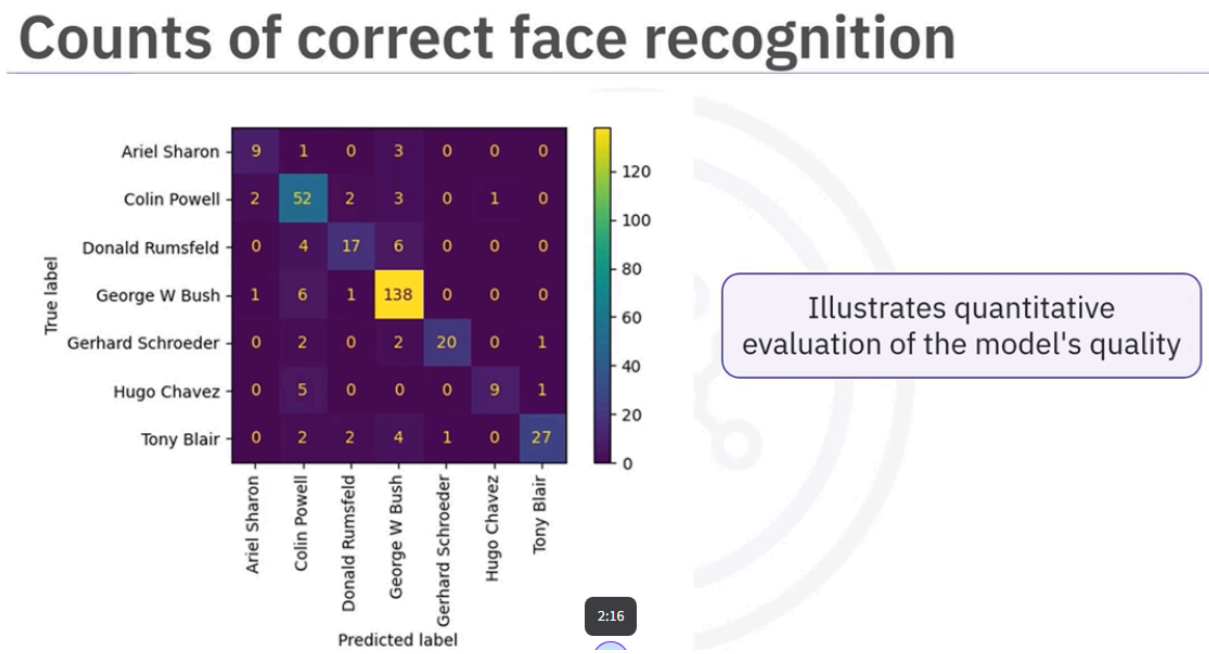
- **PCA** (Principal Component Analysis)
- **t-SNE**
- **UMAP**

Example: Face Recognition



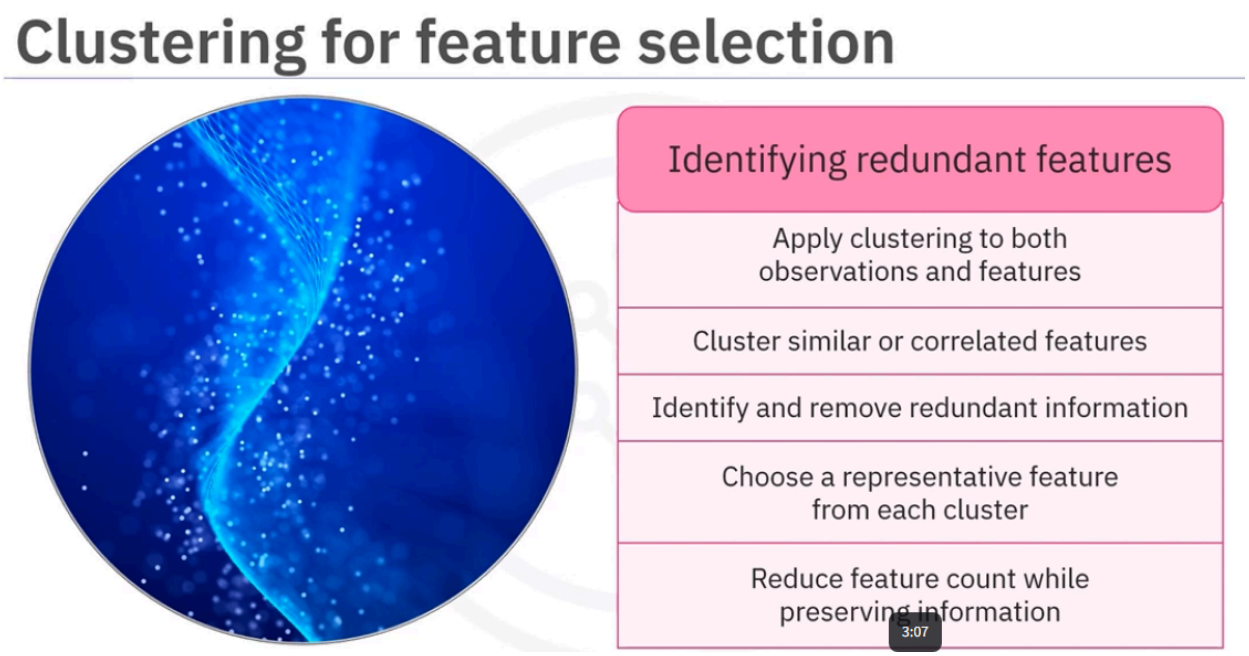
- Dataset: 966 unlabeled faces.
- PCA → extract **150 eigenfaces** (orthonormal basis).
- Project input data onto eigenface basis.
- Train SVM → predicts 12 faces accurately.

- Benefit → preserves essential features, reduces computation.



---

3 Feature Engineering

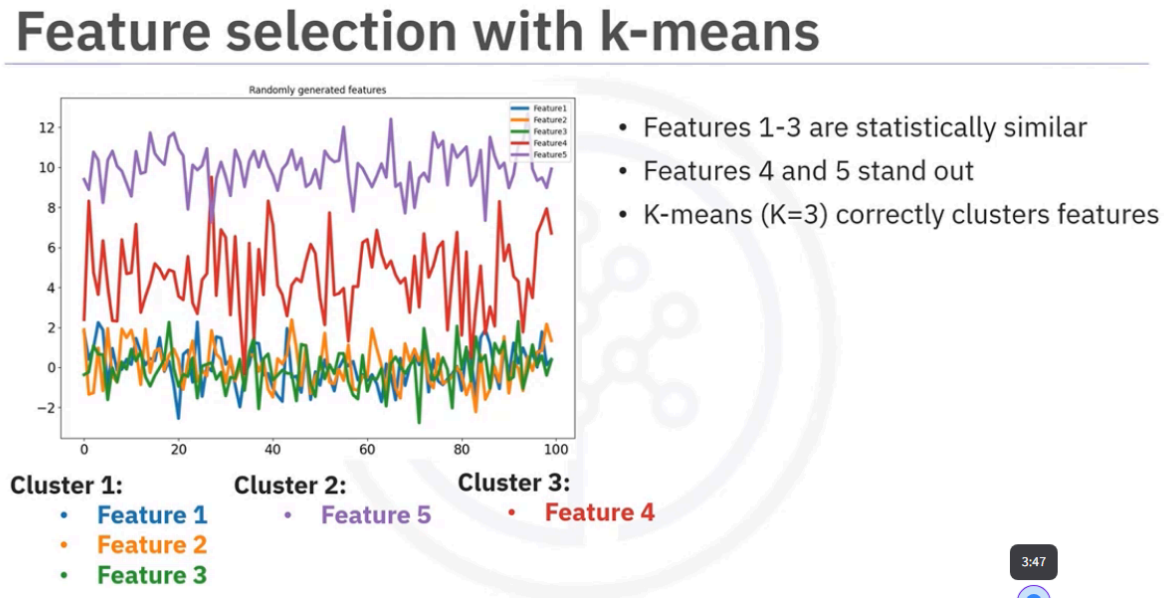


- Uses clustering & dimension reduction to:

- Remove redundant features.
- Create new, meaningful features.
- Improve interpretability & model performance.

Example:

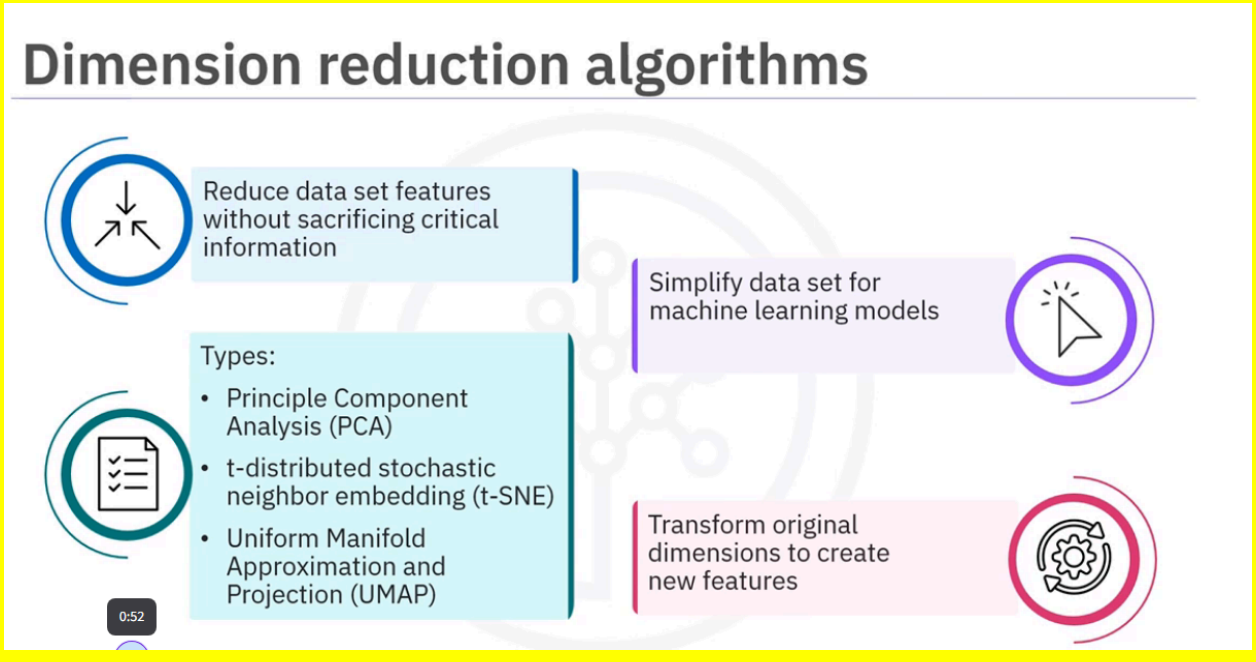
- K-means on features → identifies redundancy.
- Keep 1 from a cluster of duplicates → reduces dimensions while keeping information.



🔑 Key Takeaways

- **Clustering** = grouping data/features → useful for feature selection.
- **Dimension Reduction** = simplifies data → improves clustering, visualisation, computation.
- **Feature Engineering** = builds better features using insights from clustering & reduction.
- Together → improve **performance, scalability, and interpretability** of ML models.

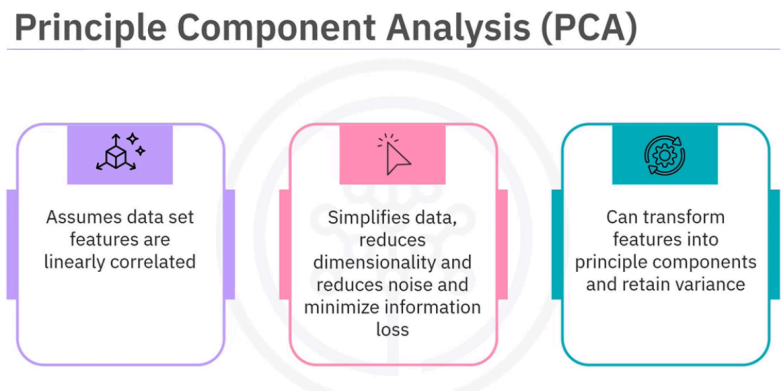
📌 Dimension Reduction Algorithms



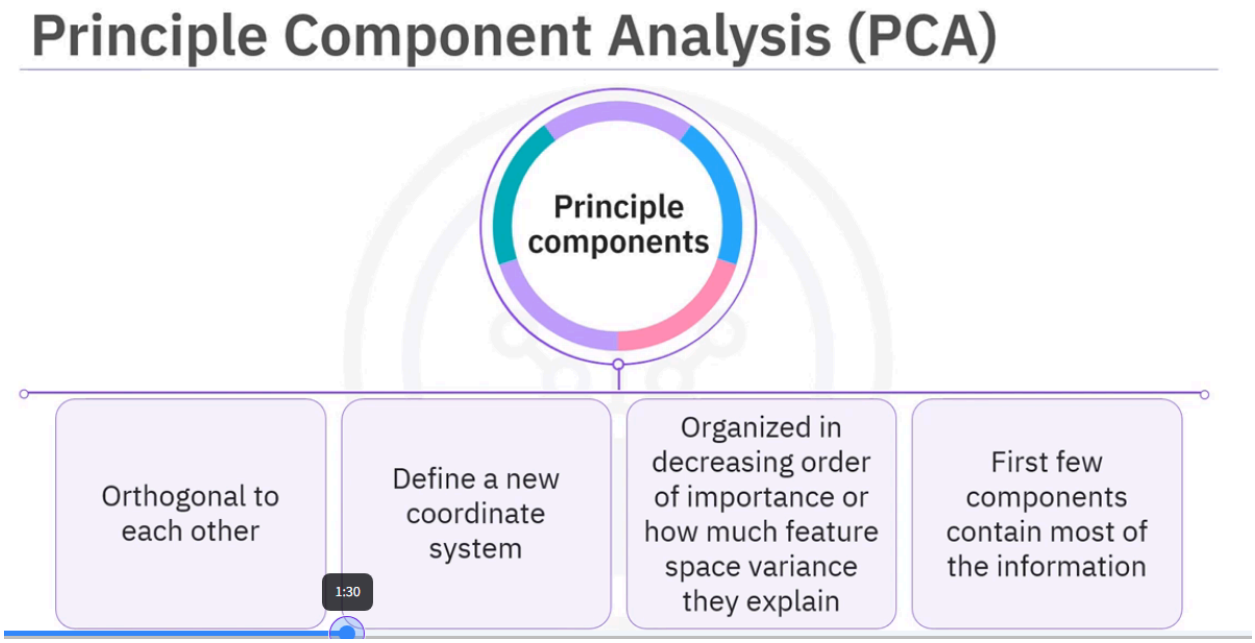
- **Purpose**
  - Reduce the number of dataset features **without losing critical information**.
  - Help in analyzing, visualizing, and simplifying high-dimensional data.
  - Improve the efficiency and performance of ML models.

---

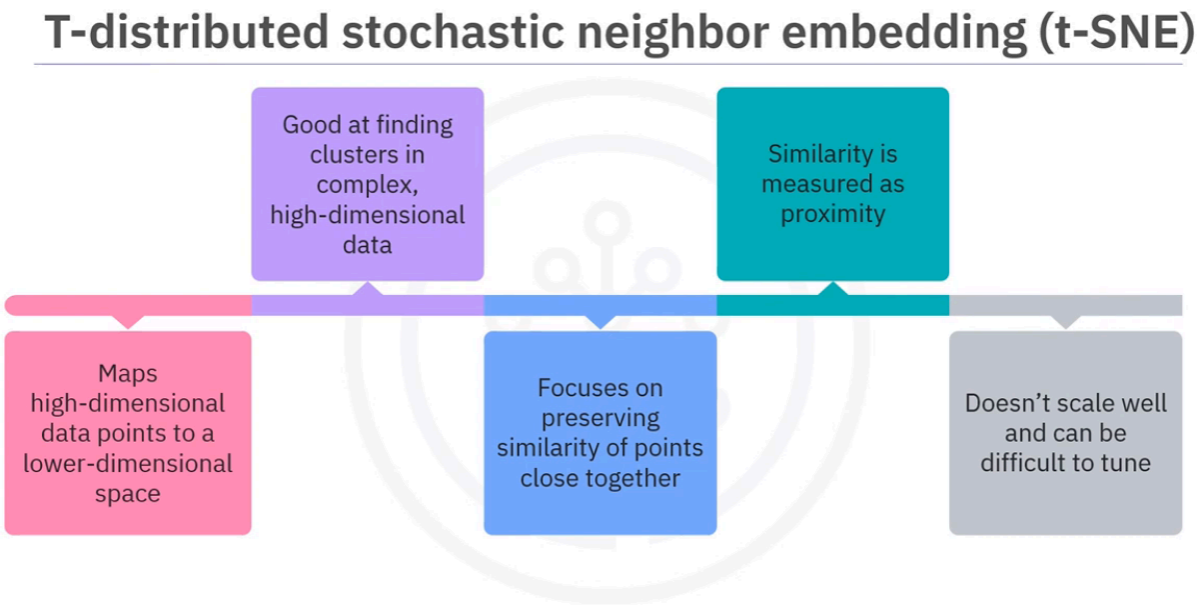
1 Principal Component Analysis (PCA)



- **Assumption:** Features are **linearly correlated**.
- **Process:**
    - Transforms features into new uncorrelated variables → **Principal Components (PCs)**.
    - PCs are **orthogonal** and ordered by explained variance.
    - First few PCs retain most of the information; later ones capture mostly noise.
- **Benefits:** Reduces dimensionality, denoises data, minimizes information loss.
- **Best for:** Linearly correlated datasets.

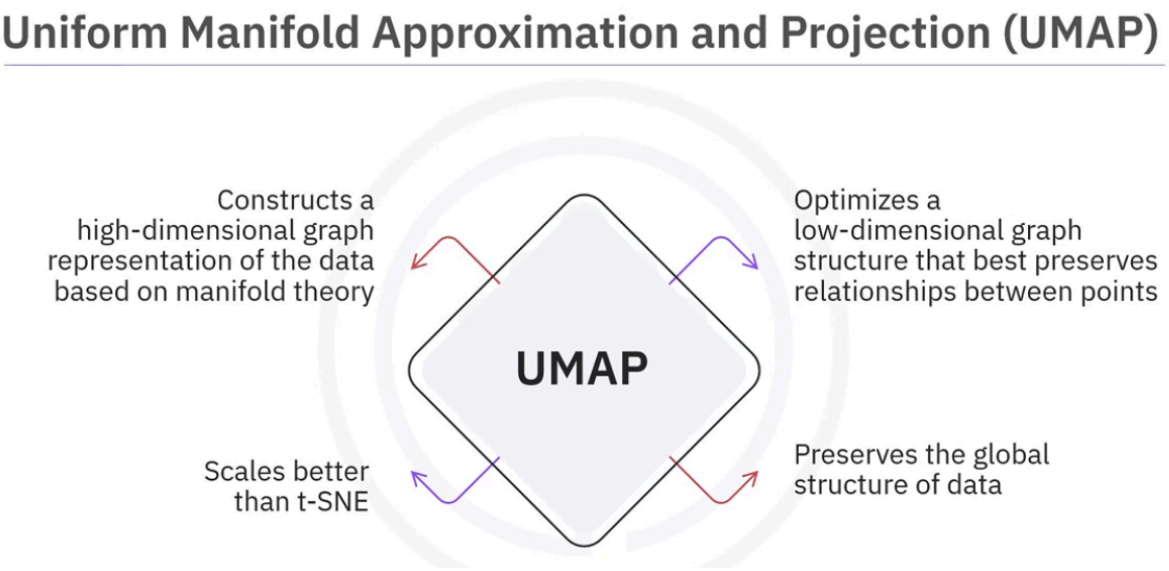


2️⃣ T-Distributed Stochastic Neighbour Embedding (t-SNE)



- **Type:** Nonlinear dimensionality reduction.
- **Focus:** Preserves **local similarity** (nearby points remain close).
- **Applications:** High-dimensional complex data like images, text.
- **Pros:** Finds meaningful clusters in 2D/3D visualization.
- **Cons:**
    - Doesn't preserve global structure well.
    - Poor scalability → computationally expensive.
    - Sensitive to hyperparameters (hard to tune).

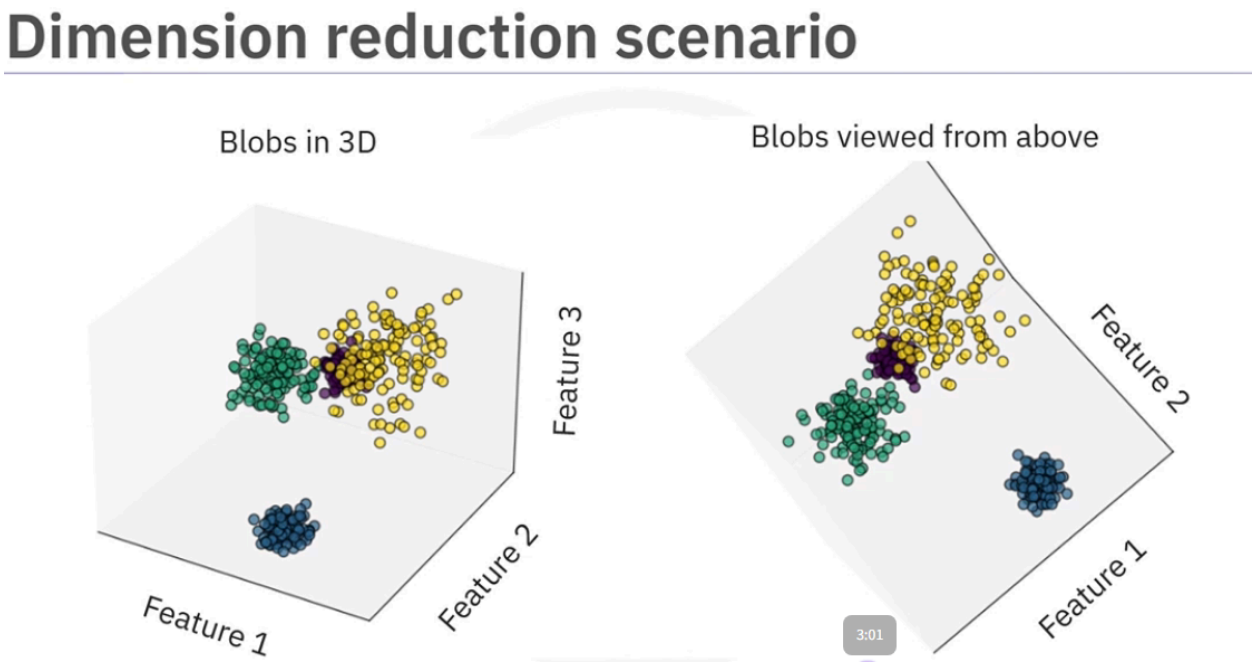
3️⃣ Uniform Manifold Approximation and Projection



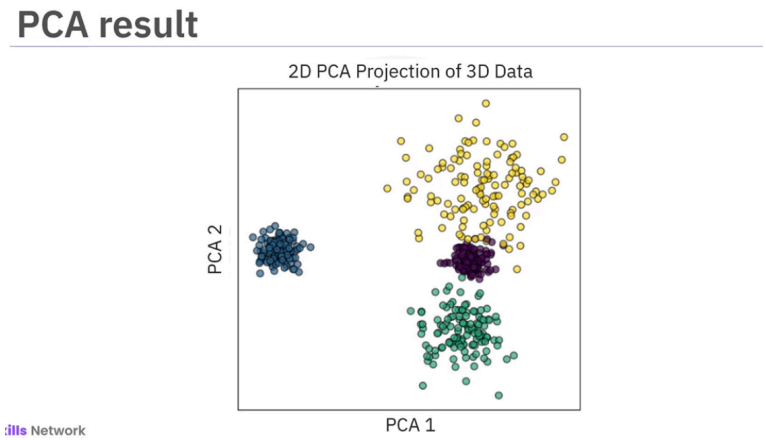
(UMAP)

- **Type:** Nonlinear dimensionality reduction, alternative to t-SNE.
- **Approach:**
    - Assumes data lies on a **low-dimensional manifold** embedded in higher dimensions.
    - Builds a **high-dimensional graph**, then optimizes a **low-dimensional graph** that preserves relationships.
- **Advantages over t-SNE:**
    - Scales better (faster, more efficient).
    - Preserves both **local** and **global** structures.
    - Often achieves better clustering performance.

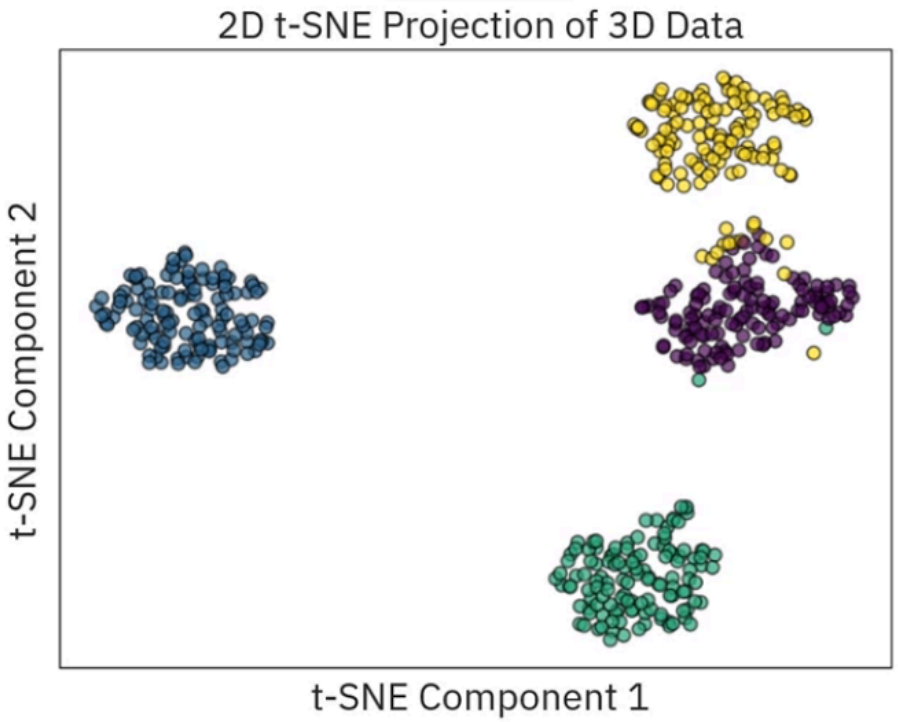
🔬 Example: Comparing PCA, t-SNE, and UMAP on Simulated Data



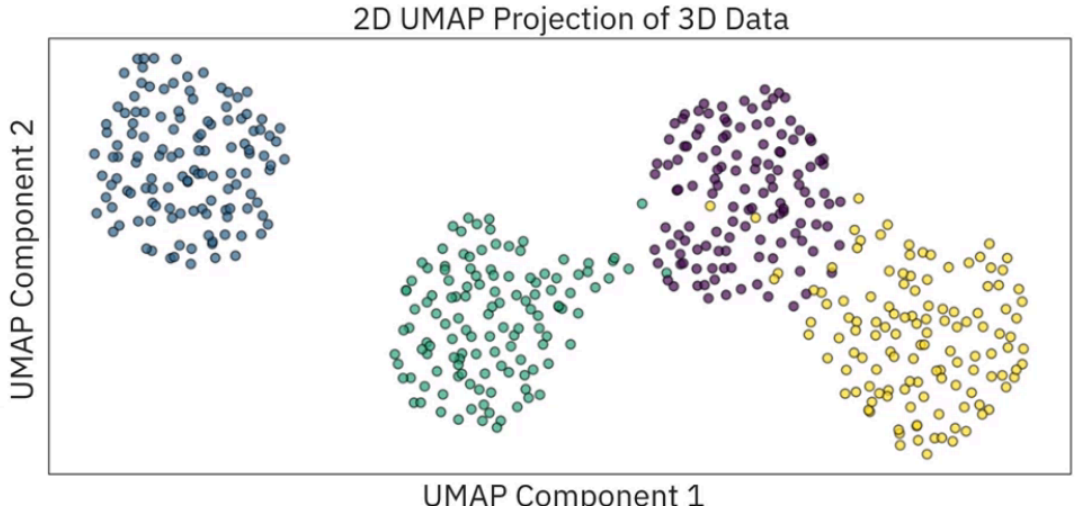
- **PCA:** Separated blobs effectively (since data was linearly correlated).
- **t-SNE:** Found 4 clusters, mostly separated, but with some overlap/mislabeled points.
- **UMAP:** Similar results, preserved global structure better than t-SNE, slight overlaps remained (as expected due to original data).



t-SNE result



UMAP result



✅ Key Takeaways

- **PCA** → Linear, efficient, noise reduction, variance-based.
- **t-SNE** → Nonlinear, local similarity, great visualization, but slow & hard to tune.
- **UMAP** → Nonlinear, preserves local + global structure, scalable, better clustering.
- All three → Reduce features while retaining essential information.

Perfect 👍 You've shared the **full PCA Lab (Applications of PCA)** — I'll now convert it into **clear, organized study notes** with explanations + working code (ready to run in Jupyter/Colab).

Applications of Principal Component Analysis (PCA)

Date: October 10, 2025
Estimated time: ~30 minutes
Dataset used: Custom 2D Gaussian + Iris Dataset

🎯 Objectives

After completing this lab, you'll be able to:

- Use PCA to project 2D data onto its principal axes.
- Apply PCA for **dimensionality reduction**.
- Interpret **explained variance** to identify important features and noise.

🧠 Introduction

Principal Component Analysis (PCA) is a statistical technique used to simplify complex data.

It works by finding **orthogonal directions (principal components)** that:

- Capture **maximum variance** in the data.
- Are **uncorrelated** with each other.
- Can be used to **reduce dimensions** while preserving most information.

We'll explore two applications:

1. Projecting **2D data** onto its principal axes.
2. Reducing **4D Iris data** to **2D** for visualization and efficiency.

⚙️ Step 1: Install Required Libraries

!pip install numpy==2.2.0 scikit-learn==1.6.0 matplotlib==3.9.3

🧩 Step 2: Import Libraries

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.decomposition import PCA

from sklearn import datasets

from sklearn.preprocessing import StandardScaler
```

🧭 Part I — PCA on 2D Data: Projecting onto Principal Axes

Step 3: Create a 2D Dataset

We'll generate a **bivariate normal distribution** with correlated features.

Covariance matrix:
$$
\begin{bmatrix}
3 & 2 \\
2 & 2
\end{bmatrix}
$$

- Variances: 3, 2
- Covariance: 2 (shows positive correlation between X1 and X2)

```
np.random.seed(42)

mean = [0, 0]

cov = [[3, 2], [2, 2]]

X = np.random.multivariate_normal(mean=mean, cov=cov, size=200)
```

Step 4: Visualize the Relationship Between X_1 and X_2

```
plt.figure()

plt.scatter(X[:, 0], X[:, 1], edgecolor='k', alpha=0.7)

plt.title("Scatter Plot of Bivariate Normal Distribution")

plt.xlabel("X1")
```

```
plt.ylabel("X2")

plt.axis('equal')

plt.grid(True)

plt.show()
```

✅ Observation:
You'll notice data points aligned along a diagonal — that's the **direction of the first principal component**.

Step 5: Perform PCA

```
pca = PCA(n_components=2)

X_pca = pca.fit_transform(X)
```

Step 6: View Principal Components

```
components = pca.components_

print(components)
```

Example Output:

```
[[ 0.7821  0.6230]

 [-0.6230  0.7821]]
```

These are **unit vectors** showing directions of maximum variance.

Step 7: Check Explained Variance

```
pca.explained_variance_ratio_
```

Example Output:

```
array([0.9111946, 0.0888054])
```

✅ Interpretation:

- PC1 explains **~91%** of total variance.
- PC2 explains **~9%** (perpendicular, minor direction).

Step 8: Project Data onto Principal Axes

```
projection_pc1 = np.dot(X, components[0])
projection_pc2 = np.dot(X, components[1])

x_pc1 = projection_pc1 * components[0][0]
y_pc1 = projection_pc1 * components[0][1]
x_pc2 = projection_pc2 * components[1][0]
y_pc2 = projection_pc2 * components[1][1]
```

Step 9: Visualize Projections

```
plt.figure()
plt.scatter(X[:, 0], X[:, 1], label='Original Data', ec='k', s=50, alpha=0.6)
plt.scatter(x_pc1, y_pc1, c='r', ec='k', marker='X', s=70, alpha=0.5, label='Projection onto PC1')
```

```
plt.scatter(x_pc2, y_pc2, c='b', ec='k', marker='X', s=70, alpha=0.5, label='Projection onto PC2')
plt.title('Linearly Correlated Data Projected onto Principal Components')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.grid(True)
plt.axis('equal')
plt.show()
```

✅ Insights:

- **Red line (PC1):** Major direction with highest variance.
- **Blue line (PC2):** Orthogonal minor direction capturing less variance.

🌸 Part II — PCA for Dimensionality Reduction (Iris Dataset)

Step 1: Load and Standardize Iris Dataset

```
iris = datasets.load_iris()
X = iris.data
y = iris.target
target_names = iris.target_names
```

```
scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)


print(target_names)
```

Output:

```
['setosa' 'versicolor' 'virginica']
```

Step 2: Apply PCA (Reduce from 4D → 2D)

```
pca = PCA(n_components=2)

X_pca = pca.fit_transform(X_scaled)
```

Step 3: Visualize Reduced Data

```
plt.figure(figsize=(8,6))

colors = ['navy', 'turquoise', 'darkorange']


for color, i, target_name in zip(colors, [0, 1, 2], target_names):

    plt.scatter(X_pca[y == i, 0], X_pca[y == i, 1],

                color=color, s=50, ec='k', alpha=0.7, label=target_name)


plt.title('PCA 2D Projection of Iris Dataset')

plt.xlabel("PC1")
```

```
plt.ylabel("PC2")

plt.legend()

plt.show()
```

✅ Observation:

- **Setosa** is clearly separable.
- **Versicolor** and **Virginica** have slight overlap but are distinguishable.

Step 4: Check Variance Explained by 2 Components

```
100 * pca.explained_variance_ratio_.sum()
```

Output:

95.81

✅ Interpretation:
The first two principal components capture **~95.8%** of the total variance — so 2D is a good compressed representation of 4D.

📊 Part III — Exploring All Components' Variance

Step 1: Apply PCA Without Reducing Dimensions

```
scaler = StandardScaler()
```

```
X_scaled = scaler.fit_transform(X)


pca = PCA()
X_pca = pca.fit_transform(X_scaled)
```

Step 2: Plot Explained and Cumulative Variance

```
explained_variance_ratio = pca.explained_variance_ratio_
cumulative_variance = np.cumsum(explained_variance_ratio)


plt.figure(figsize=(10,6))
plt.bar(range(1, len(explained_variance_ratio)+1),
        explained_variance_ratio, label='Explained Variance Ratio')
plt.step(range(1, 5), cumulative_variance, where='mid',
         linestyle='--', lw=3, color='red', label='Cumulative Variance')


plt.xlabel('Principal Components')
plt.ylabel('Explained Variance Ratio')
plt.title('Explained Variance by Principal Components')
plt.xticks(range(1, 5))
plt.legend()
plt.grid(True)
plt.show()
```

✅ Insights:

- The **red dashed line** (cumulative variance) shows how much variance is retained as we add more components.
- After PC2, most variance is already captured → **good for noise reduction**.

💡 Key Takeaways

Concept	Description
Principal Component	Direction of maximum variance in data.
Explained Variance	How much information each component captures.
Dimensionality Reduction	Compressing features while preserving important variance.
Noise Reduction	Lower PCs often represent noise or less informative variation.

🏁 Summary

- PCA reorients data along uncorrelated, variance-maximizing axes.
- In 2D data, you can visualize principal directions easily.
- In multi-dimensional data, PCA simplifies feature space efficiently.
- Iris PCA showed that **2 components ≈ 96% information retained**.