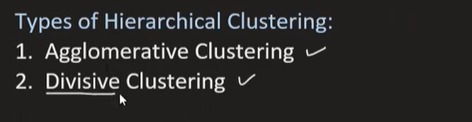
**Agglomerative Hierarchical Clustering**

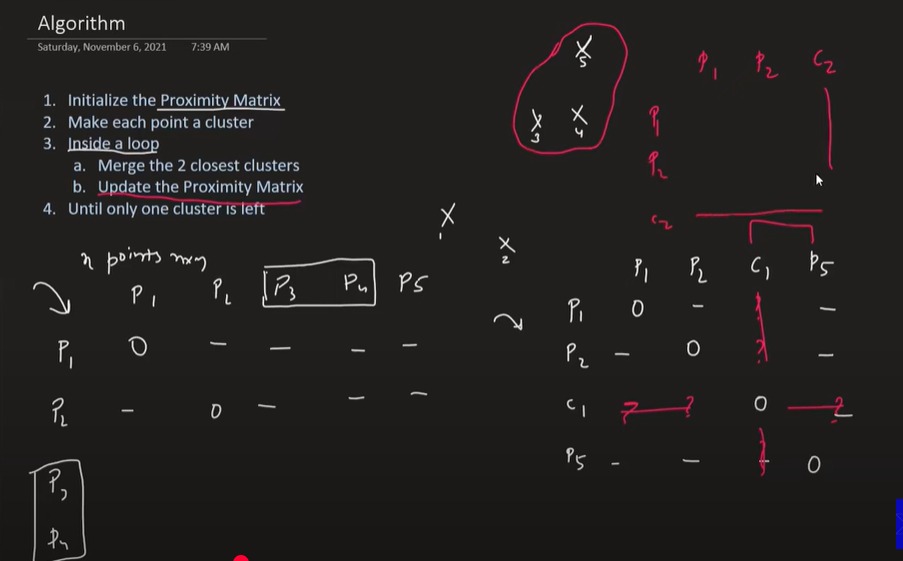
<https://www.analyticsvidhya.com/blog/2022/11/hierarchical-clustering-in-machine-learning/>

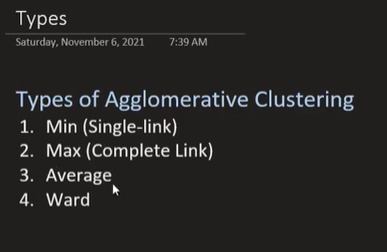
<https://builtin.com/machine-learning/agglomerative-clustering>

<https://github.com/campusx-official/agglomerative-hierarchical-clustering-demo>









**Introduction**

Hierarchical clustering is one of the most famous clustering techniques used in unsupervised machine learning. K-means and hierarchical clustering are the two most popular and effective clustering algorithms. The working mechanism they apply in the backend allows them to provide such a high level of performance.



In this article, we will discuss hierarchical clustering and its types, its working mechanisms, its core intuition, the pros and cons of using this clustering strategy and conclude with some fundamentals to remember for this practice. Knowledge about these concepts would help one to understand the working mechanism and help answer interview questions related to hierarchical clustering in a better and more efficient way.

**Overview**

Hierarchical clustering is an unsupervised machine-learning clustering strategy. Unlike [K-means clustering](https://www.analyticsvidhya.com/blog/2021/11/understanding-k-means-clustering-in-machine-learningwith-examples/), tree-like morphologies are used to bunch the dataset, and dendrograms are used to create the hierarchy of the clusters.

Here, dendrograms are the tree-like morphologies of the dataset, in which the X axis of the dendrogram represents the features or columns of the dataset, and the Y axis of the dendrogram represents the Euclidian distance between data observations.

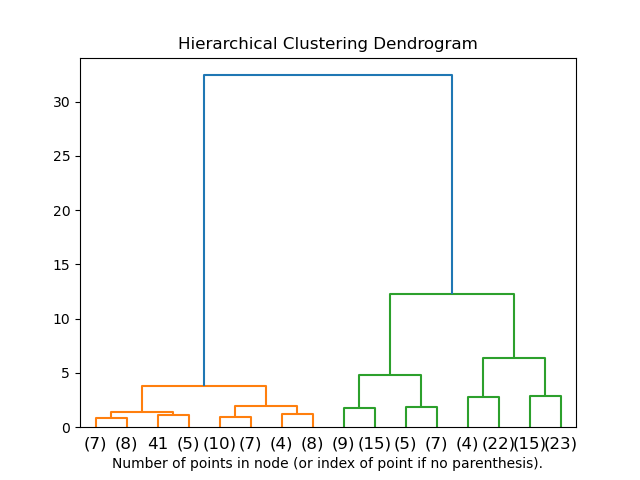
import scipy.clusters.heirarchy

plt.figure(figsize=(11,7))

plt.title("Dendrogram")

dendrogram = schs.dendrogram(shc.linkage(data,method='ward'))Copy Code

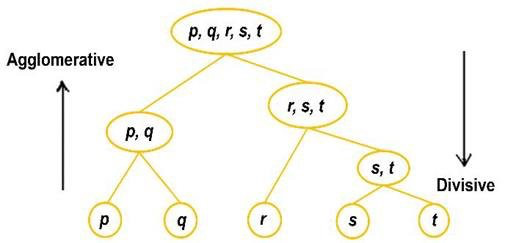
**Typical dendrograms look like this**:



**Types of Hierarchal Clustering**

There are two types of hierarchal clustering:

1. Agglomerative clustering
2. Divisive Clustering



**Agglomerative Clustering**

Each dataset is one particular data observation and a set in agglomeration clustering. Based on the distance between groups, similar collections are merged based on the loss of the algorithm after one iteration. Again the loss value is calculated in the next iteration, where similar clusters are combined again. The process continues until we reach the minimum value of the loss.

**Code**

import pandas as pd

from sklearn.cluster import AgglomerativeClustering

cluster = AgglomerativeClustering(n\_clusters=5,affinity = 'l1',linkage='single')

data=pd.read\_csv('toy\_dataset.csv')

data.drop(columns=['Illness','City','Gender'],inplace=True)

cluster.fit\_predict(data)Copy Code

**Divisive Clustering**

Divisive clustering is the opposite of agglomeration clustering. The whole dataset is considered a single set, and the loss is calculated. According to the Euclidian distance and similarity between data observations in the next iteration, the whole single set is divided into multiple clusters, hence the name “divisive.” This same process continues until we achieve the minimum loss value.

There is no method of implementing divisive clustering in Sklearn, although we can do it manually using the code below:

**Importing Required Libraries**

import numpy

import pandas

import copy

import matplotlib.pyplot

from ditsance\_matrix import distanceMatricCopy Code

**Creating The Diana Class**

Class DianakClustering:

def \_\_init\_\_(self,datak):

self.data = datak

self.n\_samples, self.n\_features = datak.shape

def fit(self,no\_clusters):

self.n\_samples, self.n\_features = data.shape

similarity\_matrix = DistanceMatrix(self.datak)

clusters = [list(range(self.n\_samples))]

while True:

csd= [np.max(similarity\_matri[clusters][:, clusters]) for clusters in clusters]

mcd = np.argmax(cd)

max\_difference\_index = np.argmax(np.mean(similarity\_matrix[clusters[mcd]][:, clusters[mcd]], axis=1))

spin = [clusters[mcd][mdi]]

lc = clusters[mcd]

del last\_clusters[mdi]

while True:

split = False

for j in ranges(len(lc))[::-1]:

spin = similarity\_matrix[lc[j], splinters]

ld = similarity\_matrix[lc[j], np.delete(lc, j, axis=0)]

if np.mean(sd) <= np.mean(lc):

spin.append(lc[j])

del lc[j]

split = True

break

if split == False:

break

del clusters[mcd]

clusters.append(splinters)

clusters.append(lc)

if len(clusters) == n\_clusters:

break

cluster\_labels = np.zeros(self.n\_samples)

for i in ranges(len(clusters)):

cl[clusters[i]] = i

return clCopy Code

Run the below code with your data:

if \_\_name\_\_ == '\_\_main\_\_':

data = pd.read\_csv('thedata.csv')

data = data.drop(columns="Name")

data = data.drop(columns="Class")

dianak = DianaClustering(data)

clusters = dianak.fit(3)

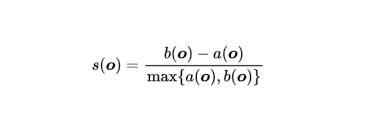
print(clusters)Copy Code

**Loss Function in Clustering**

In most clustering techniques, the silhouette score can be used to calculate the loss of the particular clustering algorithm. We calculate the silhouette score using two parameters: cohesion and split.

Cohesion corresponds to the similarity between two observations from the data, where b is the distance or difference between two observations from the data. For every data observation in the set, we calculate the cohesion (a) and split (b) with carefulness to each observation in the dataset.

The formula for the **Silhouette Score** is:



**Hierarchical Clustering vs KMeans**

The difference between Kmeans and hierarchical clustering is that in Kmeans clustering, the number of clusters is pre-defined and is denoted by “K”, but in hierarchical clustering, the number of sets is either one or similar to the number of data observations.

Another difference between these two clustering techniques is that K-means clustering is more effective on much larger datasets than hierarchical clustering. But hierarchical clustering spheroidal shape small datasets.

K-means clustering is effective on dataset spheroidal shape of clusters compared to hierarchical clustering.

Advantages

**1. Performance:**

It is effective in data observation from the data shape and returns accurate results. Unlike KMeans clustering, here, better performance is not limited to the spheroidal shape of the data; data having any values is acceptable for hierarchical clustering.

**2. Easy:**

It is easy to use and provides better user guidance with good community support. So much content and good documentation are available for a better user experience.

**3. More Approaches:**

Two approaches are there using which datasets can be trained and tested, agglomerative and divisive. So if the dataset provided is complex and very hard to train on, we can use another approach.

**4. Performance on Small Datasets:**

The hierarchical clustering algorithms are effective on small datasets and return accurate and reliable results with lower training and testing time.

Disadvantages

**1. Time Complexity:**

As many iterations and calculations are associated, the time complexity of hierarchical clustering is high. In some cases, it is one of the main reasons for preferring KMeans clustering.

**2. Space Complexity:**

As many calculations of errors with losses are associated with every epoch, the space complexity of the algorithm is very high. Due to this, while implementing the hierarchical clustering, the space of the model is considered. In such cases, we prefer KMeans clustering.

**3. Poor performance on Large Datasets:**

When training a hierarchical clustering algorithm for large datasets, the training process takes so much time with space which results in poor performance of the algorithms.

**Applications of Hierarchical Clustering**

Hierarchical clustering is widely used across various fields:

* **Biology**: Used to classify species based on genetic information, hierarchical clustering helps visualize evolutionary relationships.
* **Marketing**: Businesses use hierarchical clustering for **customer segmentation**, enabling them to target specific groups with tailored marketing strategies.
* **Social Network Analysis**: It helps uncover hidden community structures in social networks by clustering individuals based on interactions and relationships.
* **Image Processing**: Hierarchical clustering is useful in **image segmentation**, helping identify regions of interest in an image.

**Real-World Example:**

In the healthcare industry, hierarchical clustering is used to analyze patient data, helping identify subgroups of patients with similar characteristics for personalized treatments.

Hierarchical clustering’s flexibility makes it a valuable tool for a range of domains that require a deeper understanding of relationships within the data.

**Challenges and Limitations**

While hierarchical clustering offers unique benefits, it also faces several challenges:

* **Memory and Computational Complexity**: Hierarchical clustering is computationally expensive, especially for large datasets, as it requires calculating and storing the distances between all pairs of data points.
* **Difficulty with Large Datasets**: As the dataset size increases, the performance of hierarchical clustering can degrade, making it less suitable for big data applications.
* **Irreversible Merging**: Once clusters are merged or split, the algorithm does not revisit earlier decisions, which could result in suboptimal clusters.

**Solutions:**

To overcome these limitations, consider using **Ward’s method**, which minimizes the variance between clusters and reduces computational costs, or opt for sampling techniques to handle large datasets efficiently

**Hierarchical Clustering vs. K-Means Clustering**

**Hierarchical clustering** and **K-means clustering** are both popular clustering techniques, but they differ in several key ways:

**Key Differences:**

1. **Cluster Numbers**: K-means requires specifying the number of clusters in advance, while hierarchical clustering does not.
2. **Algorithm Structure**: Hierarchical clustering builds a hierarchy of clusters, while K-means aims to partition the data into K clusters directly.
3. **Dendrogram**: Hierarchical clustering produces a dendrogram, allowing for better visualization of the data’s structure, whereas K-means does not.

**Benefits of Hierarchical Clustering:**

* No need to predefine the number of clusters.
* Better for small datasets and hierarchical data.

**Benefits of K-Means Clustering:**

* Faster and more efficient for large datasets.
* Works well when the number of clusters is known.

**When to Use:**

Hierarchical clustering is ideal when you want to visualize the data hierarchy and don’t know the number of clusters upfront. K-means is preferred for larger datasets where speed and efficiency are more important.

**How Does Hierarchical Clustering Work?**

The hierarchical clustering process involves finding the two data points closest to each other and combining the two most similar ones. After repeating this process until all data points are grouped into clusters, the end result is a hierarchical tree of related groups known as a dendrogram.

Hierarchical clustering is based on the core idea that similar objects lie nearby to each other in a data space while others lie far away. It uses distance functions to find nearby data points and group the data points together as clusters.

There are different types of clustering algorithms, including centroid-based clustering algorithms, connectivity-based clustering algorithms (hierarchical clustering), distribution-based clustering algorithms and density-based clustering algorithms. The two main types of hierarchical clustering include agglomerative clustering and divisive clustering.

**Agglomerative Clustering – Full Detailed Theory**

Agglomerative Clustering is a **bottom-up** clustering algorithm where:

* Each **data point starts as its own cluster**.
* At each step, the **closest two clusters are merged**.
* This process continues until:
  + All points belong to a single cluster (for dendrogram),
  + Or a stopping condition (like number of clusters k) is met.

**🔁 Step-by-Step Working**

Let’s say we have n data points. Here are the steps:

**✅ Step 1: Start with Each Point as a Cluster**

* Initially, you have n clusters: each cluster contains exactly one point.
* Example:

ini

CopyEdit

Cluster1 = {x1}, Cluster2 = {x2}, ..., Clustern = {xn}

**✅ Step 2: Compute Distance Matrix**

* Calculate **pairwise distances** between all clusters.
* If each point is a cluster, you compute distances between points.
* The result is an **n × n symmetric matrix**.

**✅ Step 3: Find Closest Pair of Clusters**

* Find the two clusters with the **minimum distance** between them.
* Merge these two clusters into a single new cluster.

**✅ Step 4: Update the Distance Matrix**

* After merging, update the distance matrix:
  + Remove rows/columns of merged clusters.
  + Add a new row/column for the new merged cluster.
* **How you compute distance between new and existing clusters** depends on the **linkage criteria** (see below).

**✅ Step 5: Repeat Until Termination Condition is Met**

* Keep repeating Steps 3 and 4:
  + Merge two closest clusters.
  + Update distances.
* Terminate when:
  + Only one cluster remains (for dendrogram),
  + Or desired number of clusters is reached (e.g., k=3).

**🔗 Linkage Criteria (How Distances Between Clusters Are Computed)**

This is **key to how Agglomerative Clustering behaves**:

| **Linkage Type** | **Formula** | **Meaning** |
| --- | --- | --- |
| **Single Linkage** | min distance between any two points in the two clusters | Produces long, chain-like clusters |
| **Complete Linkage** | max distance between any two points in the clusters | Produces compact, spherical clusters |
| **Average Linkage** | average distance between all point pairs across clusters | Balanced method |
| **Ward's Linkage** | minimizes the **variance** within clusters | Works best for compact, spherical clusters |

**📊 Example – Small Dataset**

Suppose you have points A, B, C, D:

1. Start with: {A}, {B}, {C}, {D}
2. Compute distances:

css

CopyEdit

A B C D

A - 2 6 10

B 2 - 5 9

C 6 5 - 4

D 10 9 4 -

1. Closest pair: A-B (distance = 2) → merge → {AB}, {C}, {D}
2. Recalculate distances between new {AB} and other clusters depending on linkage.
3. Repeat...

**🌲 Dendrogram**

A **dendrogram** is a tree-like structure that visually represents the merging process.

* X-axis: individual data points
* Y-axis: distance at which clusters merged
* You can **cut the dendrogram at a certain height** to get desired number of clusters.

**What is Linkage?**

In hierarchical clustering, when we merge two clusters, we must **calculate the distance between clusters**, not just points.

**Linkage** defines **how to measure the distance between two clusters** based on the distances between their individual points.

**📌 Linkage Types (with Full Details)**

**1️⃣ Single Linkage (Minimum Linkage)**

**🔍 Definition:**

Distance between two clusters = **minimum distance** between any point in one cluster to any point in the other.

**📐 Formula:**

D(C1, C2) = min { dist(x, y) | x ∈ C1, y ∈ C2 }

**📊 Behavior:**

* Forms **"chains" of points** — tends to produce **long, elongated** clusters.
* Very sensitive to **noise/outliers** (can create false connections).

**📈 Visual:**

A----B----C D----E

| |

chain gap

**2️⃣ Complete Linkage (Maximum Linkage)**

**🔍 Definition:**

Distance between two clusters = **maximum distance** between any point in one cluster to any point in the other.

**📐 Formula:**

D(C1, C2) = max { dist(x, y) | x ∈ C1, y ∈ C2 }

**📊 Behavior:**

* Produces **compact**, **tight** clusters.
* Sensitive to **outliers** but less than single-link.
* Clusters are usually **spherical or compact in shape**.

**📈 Visual:**

C1 [ o o o ] C2 [ o o o ]

tight gap enforced before merge

**3️⃣ Average Linkage (UPGMA)**

UPGMA = Unweighted Pair Group Method with Arithmetic Mean

**🔍 Definition:**

Distance between two clusters = **average distance** between all point pairs across the two clusters.

**📐 Formula:**

D(C1, C2) = (1 / (|C1| \* |C2|)) \* ∑ dist(x, y), for all x ∈ C1, y ∈ C2

**📊 Behavior:**

* Balances between single and complete linkage.
* Produces more **natural and balanced clusters**.
* Less sensitive to outliers.
* Good default if you’re unsure which to use.

**📈 Visual:**

Takes all point-pair distances and averages.

Results in moderate-sized, evenly shaped clusters.

**4️⃣ Ward's Linkage (Variance Minimization)**

**🔍 Definition:**

Merge the two clusters that result in the **smallest increase in total within-cluster variance (inertia)**.

**📐 Formula (conceptually):**

Minimize the increase in:

SSE = Sum of Squared Errors = ∑ || x - μ(cluster) ||²

Merges clusters that keep SSE low.

**📊 Behavior:**

* Encourages **compact, spherical clusters** (like K-Means).
* Usually performs very well in **Euclidean spaces**.
* Less sensitive to noise.
* **Works only with Euclidean distance**.

**📈 Visual:**

plaintext

CopyEdit

Clusters are merged only if variance stays low.

Prefers globular, balanced partitions.

**🧮 Summary Table of Linkage Types**

| **Linkage Type** | **Distance Used** | **Behavior** | **Sensitive to Outliers?** | **Cluster Shape** |
| --- | --- | --- | --- | --- |
| **Single** | Minimum distance between pairs | Chained, elongated | ✅ Very | Irregular |
| **Complete** | Maximum distance between pairs | Compact and tight | ✅ Moderate | Compact |
| **Average** | Average distance of all pairs | Balanced and natural | ❌ Less | Balanced |
| **Ward** | Increase in within-cluster variance | Globular, compact (like K-Means) | ❌ Less | Spherical |

**When to Use Each Linkage Type in Agglomerative Clustering**

**🔹 1. Single Linkage (Minimum Distance)**

**✅ Use When:**

* Your data forms **irregular or elongated shapes** (e.g., clusters along curves or chains).
* You want to **preserve connectivity** between closely linked points.
* You care more about **distance chaining** than compactness.

**❌ Avoid When:**

* There's **noise or outliers** – it will wrongly connect them (producing “loose” clusters).
* You want compact, dense clusters.
* You care about **tightness or variance** within clusters.

**🔹 2. Complete Linkage (Maximum Distance)**

**✅ Use When:**

* You want **compact**, **spherical clusters**.
* You want to avoid long chains or stretched clusters.
* You want to **reduce outlier influence** compared to single-link.

**❌ Avoid When:**

* Data naturally forms **irregular or chain-like clusters**.
* You have many outliers — max distances can still be skewed.

**🔹 3. Average Linkage (Mean Distance)**

**✅ Use When:**

* You want a **balance** between compactness and flexibility.
* You expect **reasonably well-separated clusters**, not necessarily spherical.
* You want robustness to **moderate noise or outliers**.

**❌ Avoid When:**

* You need extremely tight clusters (complete linkage is better).
* Your data is highly imbalanced in density.

**🔹 4. Ward’s Linkage (Minimize Variance)**

**✅ Use When:**

* You want **K-means-like clustering behavior**, but with a hierarchical view.
* You have **Euclidean data** with **compact, globular clusters**.
* You care about **minimizing variance** within clusters.
* You want excellent performance on numeric data with real-world distributions.

**❌ Avoid When:**

* Your data has **non-Euclidean** distance (e.g., cosine, Jaccard).
* Your clusters are **non-spherical or vary in size**.
* You’re dealing with **categorical or sparse data** (Ward doesn’t handle it well).

**🧠 Rule of Thumb Summary**

| **If Your Data Has...** | **Best Linkage** |
| --- | --- |
| Compact, spherical clusters | **Ward** |
| Irregular or chain-shaped clusters | **Single** |
| Balanced, general-purpose clustering | **Average** |
| Want to minimize max within-cluster distance | **Complete** |

**⚠️ Pro Tips:**

1. **Always normalize your data** before clustering if features are on different scales.
2. **Use dendrograms** to visually inspect how linkage type affects merging behavior.
3. Try multiple linkage types and **compare with domain knowledge or silhouette scores**.

**Benefits of Agglomerative Clustering**

| **Benefit** | **Explanation** |
| --- | --- |
| 🔹 **No need to specify number of clusters upfront** | Unlike K-Means, you can build the hierarchy and decide the number of clusters later (via dendrogram). |
| 🔹 **Hierarchical output (Dendrogram)** | You get a full tree showing all possible merges — great for understanding structure at different levels. |
| 🔹 **Flexible distance metrics and linkage types** | You can customize clustering behavior using different linkage criteria and distance measures. |
| 🔹 **Works with non-globular clusters (sometimes)** | Depending on the linkage, it can discover non-spherical clusters. |
| 🔹 **Interpretable** | Merging decisions are explicit and traceable in a tree-like format. |

**❌ Limitations of Agglomerative Clustering**

| **Limitation** | **Details** |
| --- | --- |
| ⚠️ **Computationally expensive** | Time complexity is **O(n² log n)** or worse — not practical for large datasets (e.g., n > 10,000). |
| ⚠️ **No backtracking** | Once two clusters are merged, the decision is **permanent** (greedy strategy). Mistakes early on can’t be fixed. |
| ⚠️ **Sensitive to noise and outliers** | Especially with Single or Complete linkage, noise can distort clusters. |
| ⚠️ **Doesn't scale well** | Hierarchical trees (dendrograms) grow in size quickly with more data points. |
| ⚠️ **Choice of linkage and distance can drastically affect results** | And there's **no universal rule** to choose them. |
| ⚠️ **Only for numeric/categorical data with custom distances** | Doesn’t naturally handle mixed data types (e.g., text + numeric). |

**🚀 How to Overcome These Limitations — Alternatives**

| **Problem in Agglomerative** | **Better Algorithm** | **Why It Helps** |
| --- | --- | --- |
| 🔁 High computational cost | **K-Means**, **MiniBatch K-Means** | Much faster (O(nkdi)), good for large datasets with globular clusters |
| 🌪 Sensitive to noise/outliers | **DBSCAN**, **HDBSCAN** | Robust to noise, doesn’t require specifying k, finds arbitrary shapes |
| 🚫 No backtracking in merges | **Spectral Clustering** | Uses graph Laplacians and eigenvectors to better capture cluster structure |
| 🔄 Doesn’t scale to large datasets | **Birch (Balanced Iterative Reducing and Clustering using Hierarchies)** | Scalable hierarchical clustering; approximates the tree-building |
| ❌ Needs careful linkage/distance choice | **GMM (Gaussian Mixture Models)** | Probabilistic model, can model elliptical clusters; not based on distance heuristics |
| ⚖ Mixed data types (e.g., numbers + categories) | **K-Prototypes**, **Gower Distance + DBSCAN** | Designed for categorical + numeric data clustering |

**🧠 Rule of Thumb – Which to Choose?**

| **Data Scenario** | **Suggested Algorithm** |
| --- | --- |
| Small, interpretable clustering needed | **Agglomerative** |
| Large dataset, globular clusters | **K-Means** |
| Noise/outliers, arbitrary shape | **DBSCAN / HDBSCAN** |
| Probabilistic modeling required | **Gaussian Mixture Model** |
| Mixed data types | **K-Prototypes / Gower + DBSCAN** |
| Scalable hierarchical clustering | **Birch** |

**Print all shaps**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make\_moons, make\_blobs, make\_circles

from sklearn.preprocessing import StandardScaler

# Create datasets with different shapes

X1, \_ = make\_blobs(n\_samples=300, centers=3, random\_state=42)

X2, \_ = make\_moons(n\_samples=300, noise=0.05, random\_state=42)

X3, \_ = make\_circles(n\_samples=300, factor=0.5, noise=0.05, random\_state=42)

# Standardize features

scaler = StandardScaler()

X1\_scaled = scaler.fit\_transform(X1)

X2\_scaled = scaler.fit\_transform(X2)

X3\_scaled = scaler.fit\_transform(X3)

# Plotting

fig, axes = plt.subplots(1, 3, figsize=(18, 5))

axes[0].scatter(X1\_scaled[:, 0], X1\_scaled[:, 1], c='blue', s=30)

axes[0].set\_title("Blob-shaped (Spherical) Data")

axes[1].scatter(X2\_scaled[:, 0], X2\_scaled[:, 1], c='green', s=30)

axes[1].set\_title("Moon-shaped (Non-linear) Data")

axes[2].scatter(X3\_scaled[:, 0], X3\_scaled[:, 1], c='purple', s=30)

axes[2].set\_title("Circular-shaped Data")

for ax in axes:

ax.grid(True)

plt.tight\_layout()

plt.show()

import numpy as np

import matplotlib.pyplot as plt

from scipy.spatial.distance import euclidean

import itertools

# Step 1: Define sample data

X = np.array([

[1, 2],

[2, 1],

[4, 3],

[5, 4]

])

# Step 2: Helper function to compute distance between clusters

def cluster\_distance(c1, c2, linkage='single'):

"""Calculate distance between clusters c1 and c2"""

if linkage == 'single':

return min(euclidean(p1, p2) for p1 in c1 for p2 in c2)

elif linkage == 'complete':

return max(euclidean(p1, p2) for p1 in c1 for p2 in c2)

elif linkage == 'average':

distances = [euclidean(p1, p2) for p1 in c1 for p2 in c2]

return sum(distances) / len(distances)

else:

raise ValueError("Unsupported linkage")

# Step 3: Initial clusters (each point is its own cluster)

clusters = [[x] for x in X.tolist()]

# For visualization

history = []

# Step 4: Manual Agglomerative Clustering

while len(clusters) > 1:

# Compute all pairwise distances

pair\_distances = {

(i, j): cluster\_distance(clusters[i], clusters[j], linkage='single')

for i, j in itertools.combinations(range(len(clusters)), 2)

}

# Find the pair with the minimum distance

(i\_min, j\_min), min\_dist = min(pair\_distances.items(), key=lambda item: item[1])

# Merge the two clusters

new\_cluster = clusters[i\_min] + clusters[j\_min]

# Save history for visualization

history.append((clusters[i\_min], clusters[j\_min], min\_dist))

# Remove and update clusters

clusters = [clusters[k] for k in range(len(clusters)) if k not in (i\_min, j\_min)]

clusters.append(new\_cluster)

# Step 5: Display merge history

for step, (c1, c2, dist) in enumerate(history):

print(f"Step {step + 1}: Merged {c1} and {c2} at distance {dist:.2f}")

# Step 6: Plot original points

plt.scatter(X[:, 0], X[:, 1], color='blue')

plt.title("Original Data Points")

plt.grid(True)

plt.show()

**Key Hyperparameters in Agglomerative Clustering**

| **Hyperparameter** | **Description** |
| --- | --- |

**1. n\_clusters (int)**

* **What it is**: The number of clusters you want in the final output.
* **Default**: Usually required (e.g., in sklearn.AgglomerativeClustering).
* **Impact**: Determines how far the algorithm goes in the merging process (i.e., when to stop).

✅ If you don’t know this ahead of time, use **dendrograms** or **distance thresholds** to estimate it.

**2. linkage (str)**

* **What it is**: The strategy used to calculate the distance between clusters.
* **Options**:
  + 'single': Minimum distance between points.
  + 'complete': Maximum distance between points.
  + 'average': Average pairwise distance between clusters.
  + 'ward': Minimizes the variance within clusters.

⚠ Choosing the right linkage method **changes the shape** and **tightness** of the resulting clusters.

**3. affinity (or metric) (str)**

* **What it is**: The distance metric used to compute distances between data points or clusters.
* **Examples**: 'euclidean', 'manhattan', 'cosine', 'l1', 'l2'.

⚠ 'ward' linkage **requires** 'euclidean' distance — other distances are invalid with it.

**4. distance\_threshold (float)**

* **What it is**: If set, the algorithm will stop merging when clusters are farther apart than this threshold.
* **Mutually exclusive with** n\_clusters: you set **either one**, not both.

Useful if you want to **control clustering based on real-world distances**, not number of clusters.

**5. compute\_full\_tree (bool)**

* **What it is**: Whether to compute the full linkage tree (even if n\_clusters is set).
* **Use case**: Needed for **dendrogram plots** or cluster tree inspection.

**🧠 Summary Table**

| **Param** | **Type** | **Required** | **Notes** |
| --- | --- | --- | --- |
| n\_clusters | int | ✅ | Or use distance\_threshold |
| linkage | str | ✅ | 'ward', 'single', etc. |
| affinity/metric | str | ✅ | 'euclidean', etc. |
| distance\_threshold | float | ❌ | Alternative to n\_clusters |
| compute\_full\_tree | bool | ❌ | Useful for dendrograms |