Laboratory for
Reliable
Computing

# Working with Unlabeled Data - Clustering Analysis

Hsi-Pin Ma 馬席彬

http://lms.nthu.edu.tw/course/40724

Department of Electrical Engineering

National Tsing Hua University

## Outline

- Grouping Objects by Similarity Using k-Means
- Organizing Clusters as a Hierarchical Tree
- Locating Regions of High Density via DBSCAN

## Clustering/Cluster Analysis

- A category of unsupervised learning techniques
  - Discover hidden structures in data
- Goal is to find a natural grouping in data so items in the same cluster are more similar to each other than to those from different clusters
- In this chapter
  - Finding centers of similarity using the popular k-means
  - Taking a bottom-up approach to build hierarchical clustering trees
  - Identifying arbitrary shapes of objects using a density-based clustering approach

## Grouping Objects by Similarity Using *k*-Means

## Clustering

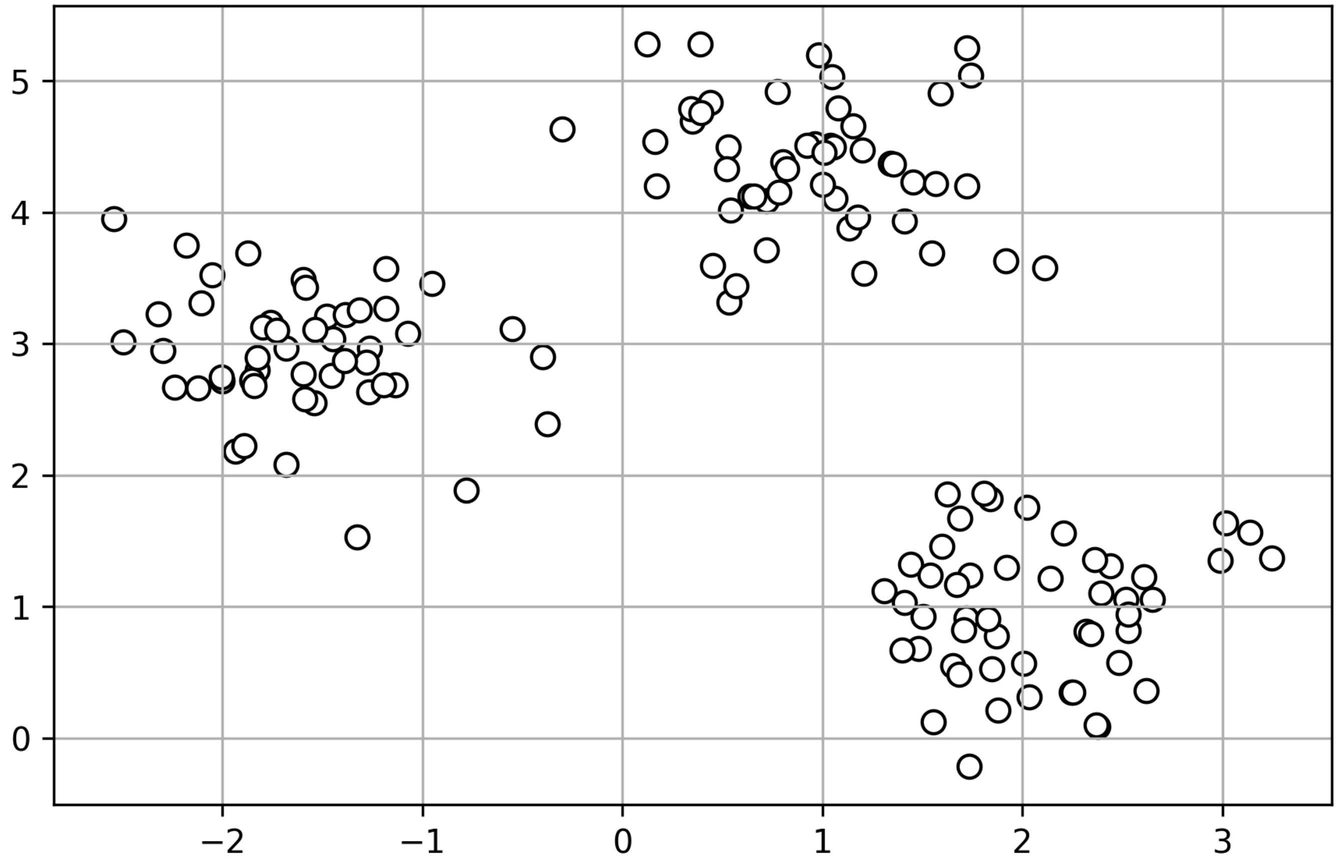
- Categories of clustering algorithm
  - Prototype-based clustering (*k*-means belongs to this)
  - Hierarchical clustering
  - Density-based clustering
- Prototype-based clustering
  - Each cluster is represented by a prototype
  - A prototype can either be the **centroid** (*average*) of similar points with continuous features or the **medoid** (the most *representative* or most frequently occurring point) in the case of categorical features
  - Usually formulated as a cost minimization clustering problem

## *k*-means Algorithm

- ***Randomly*** pick *k* centroids from the sample points as initial cluster centers
- Assign each instance to the nearest centroid $\mu^{(j)},\ j \in \{1,\ldots,k\}$
- Move each centroid to the center of the sample points that were assigned to it
- Repeat 2 and 3 until the cluster assignments do not change or a user-defined tolerance or maximum of iterations is reached

# Generating Clusters for Visualization

```
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=150,
                  n_features=2,
                  centers=3,
                  cluster_std=0.5,
                  shuffle=True,
                  random_state=0)
```



```
import matplotlib.pyplot as plt

plt.scatter(X[:, 0], X[:, 1],
            c='white', marker='o', edgecolor='black', s=50)
plt.grid()
plt.tight_layout()
#plt.savefig('images/11_01.png', dpi=300)
plt.show()
```

## Empirical k-means Cost Function

- Similarity measurement between objects
  - **Squared Euclidean distance** between two points *x* and *y* in m-dimensional space $d(\boldsymbol{x},\boldsymbol{y})^2 = \sum_{j=1}^{m}(x_j - y_j)^2 = \|\boldsymbol{x}-\boldsymbol{y}\|_2^2$
  - *i*: sample index, *j*: cluster index
- For continuous feature values, the empirical k-means cost function is usually taken as the within cluster **sum of squared errors** (SSE), sometimes called **cluster inertia**

centroid for cluster j

$$SSE = \sum_{i=1}^{n}\sum_{j=1}^{k} w^{(i,j)} \left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2$$

is 1 if $x^{(i)}$ is in cluster j, otherwise 0

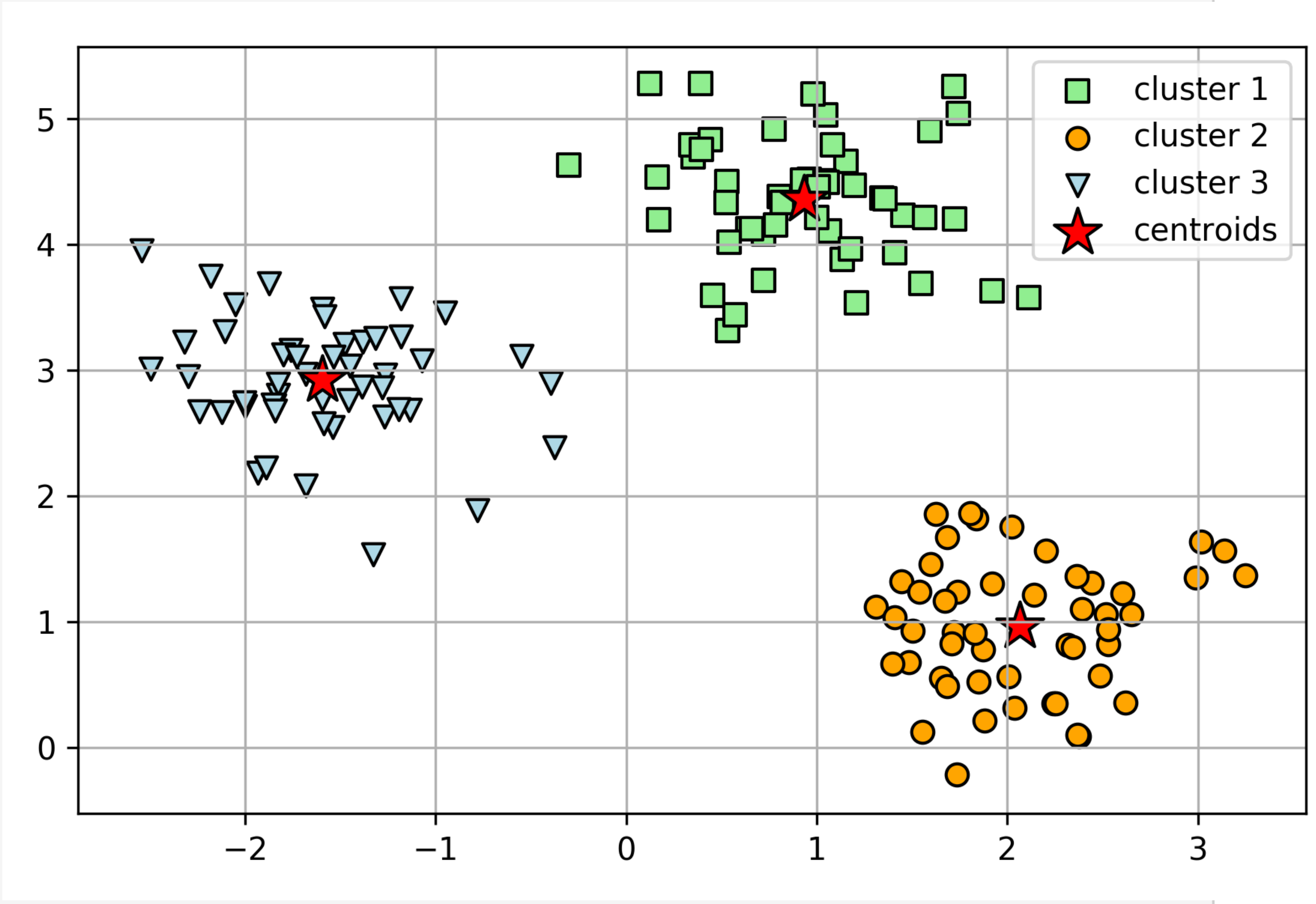
# **KMeans** classs form **cluster** Module

- Specifying *k* a priori is one of the limitations of *k*-means

```
from sklearn.cluster import KMeans

km = KMeans(n_clusters=3,
            init='random',
            n_init=10,
            max_iter=300,
            tol=1e-04,
            random_state=0)

y_km = km.fit_predict(X)
```

# Clustering Visualization

```
plt.scatter(X[y_km == 0, 0],
            X[y_km == 0, 1],
            s=50, c='lightgreen',
            marker='s', edgecolor='black',
            label='cluster 1')
plt.scatter(X[y_km == 1, 0],
            X[y_km == 1, 1],
            s=50, c='orange',
            marker='o', edgecolor='black',
            label='cluster 2')
plt.scatter(X[y_km == 2, 0],
            X[y_km == 2, 1],
            s=50, c='lightblue',
            marker='v', edgecolor='black',
            label='cluster 3')
plt.scatter(km.cluster_centers_[:, 0],
            km.cluster_centers_[:, 1],
            s=250, marker='*',
            c='red', edgecolor='black',
            label='centroids')
plt.legend(scatterpoints=1)
plt.grid()
plt.tight_layout()
#plt.savefig('images/11_02.png', dpi=300)
plt.show()
```



## Comments

- It is possible that one or more clusters resulted from *k*-means algorithm can be empty
- In the current *k*-means implementation of scikit-learn, if a cluster is empty, the algorithm will search for the instance that is farthest away from the centroid of the empty cluster and reassign the centroid to be this farthest point
- Using a random seed to place the initial centroid may result in *bad clustering* or *slow convergence* if the centroids are chosen poorly

## *k*-means++

- A smarter way to place the initial centroids
  - Initialize an empty set **M** to store the *k* centroids being selected
  - Randomly choose the first centroid $\mu^{(j)}$ from the input samples and assign it to **M**
  - For each sample $\mathbf{x}^{(i)}$ that is not in M, find the minimum squared distance $d(\mathbf{x}^{(i)},\mathbf{M})^2$ to any of the centroids in **M**
  - To randomly select the next centroid $\mu^{(p)}$, use a weighted probability distribution equal to $\dfrac{d\left(\boldsymbol{\mu}^{(p)},\mathbf{M}\right)^2}{\sum_i d\left(\boldsymbol{x}^{(i)},\mathbf{M}\right)^2}$
    - Select the largest
  - Repeat step 2 and 3 until *k* centroids are chosen
  - Proceed with the classic *k*-means algorithm

## *k*-means++

- To use *k*-means++ with scikit-learn's **KMeans** object, we just need to set the **init** parameter to '*k*-means++'
- In fact, '*k*-means++' is the default argument to the **init** parameter
  - classic *k*-means via **init='random'**
  - *k*-means++ via **init='k-means++'**

# Fuzzy C-Means (FCM) Algorithm

- Hard clustering
  - Each sample in a dataset is assigned to exactly one cluster

$$\begin{bmatrix} \boldsymbol{\mu}^{(1)} \rightarrow 0 \\ \boldsymbol{\mu}^{(2)} \rightarrow 1 \\ \boldsymbol{\mu}^{(3)} \rightarrow 0 \end{bmatrix}$$

- Soft clustering (fuzzy clustering)
  - Assign a sample to one or more clusters
  - A popular example: fuzzy C-means/soft k-means/fuzzy k-means
  - Replace hard cluster assignment with probability for each point belonging to each cluster

$$\begin{bmatrix} \boldsymbol{\mu}^{(1)} \rightarrow 0.10 \\ \boldsymbol{\mu}^{(2)} \rightarrow 0.85 \\ \boldsymbol{\mu}^{(3)} \rightarrow 0.05 \end{bmatrix}$$

## FCM Algorithm

- Specify the number of *k* centroids and randomly assign the cluster memberships for each point
- Compute the cluster centroids $\mu^{(j)},\ j \in \{1,\ldots,k\}$
- Update the cluster memberships for each point
- Repeat steps 2 and 3 until the membership coefficients do not change, or a user-defined tolerance or maximum number of iterations is reached

## Objective Function of FCM

- Similar to the within cluster sum-squared-error that we minimize in k-means

$$J_m = \sum_{i=1}^{n}\sum_{j=1}^{k} w^{m\,(i,j)} \left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2 \qquad w^{(i,j)} \in [0,1]$$

  - But *w* is a real value denoting the cluster membership probability, not a binary value
  - *m*: fuzzy coefficient or fuzzier that controls the degree of fuzziness. The larger the value of *m*, the smaller the cluster membership *w* becomes

- Cluster membership probability

$$w^{(i,j)} = \left[ \sum_{p=1}^{k} \left( \frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(p)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1} \qquad \boldsymbol{\mu}^{(j)} = \frac{\sum_{i=1}^{n} w^{m(i,j)} \boldsymbol{x}^{(i)}}{\sum_{i=1}^{n} w^{m(i,j)}}$$

## Quality of Clustering

- One of the main challenges in unsupervised learning is that we do not know the definitive answer
- To quantify the quality of an unsupervised learning task such as clustering, we have to use intrinsic metrics such as the within-class SSE (a distortion metric) we discussed previously
- In scikit-learn, the within-class SSE can be accessed via **inertia_** attribute after fitting the Means model

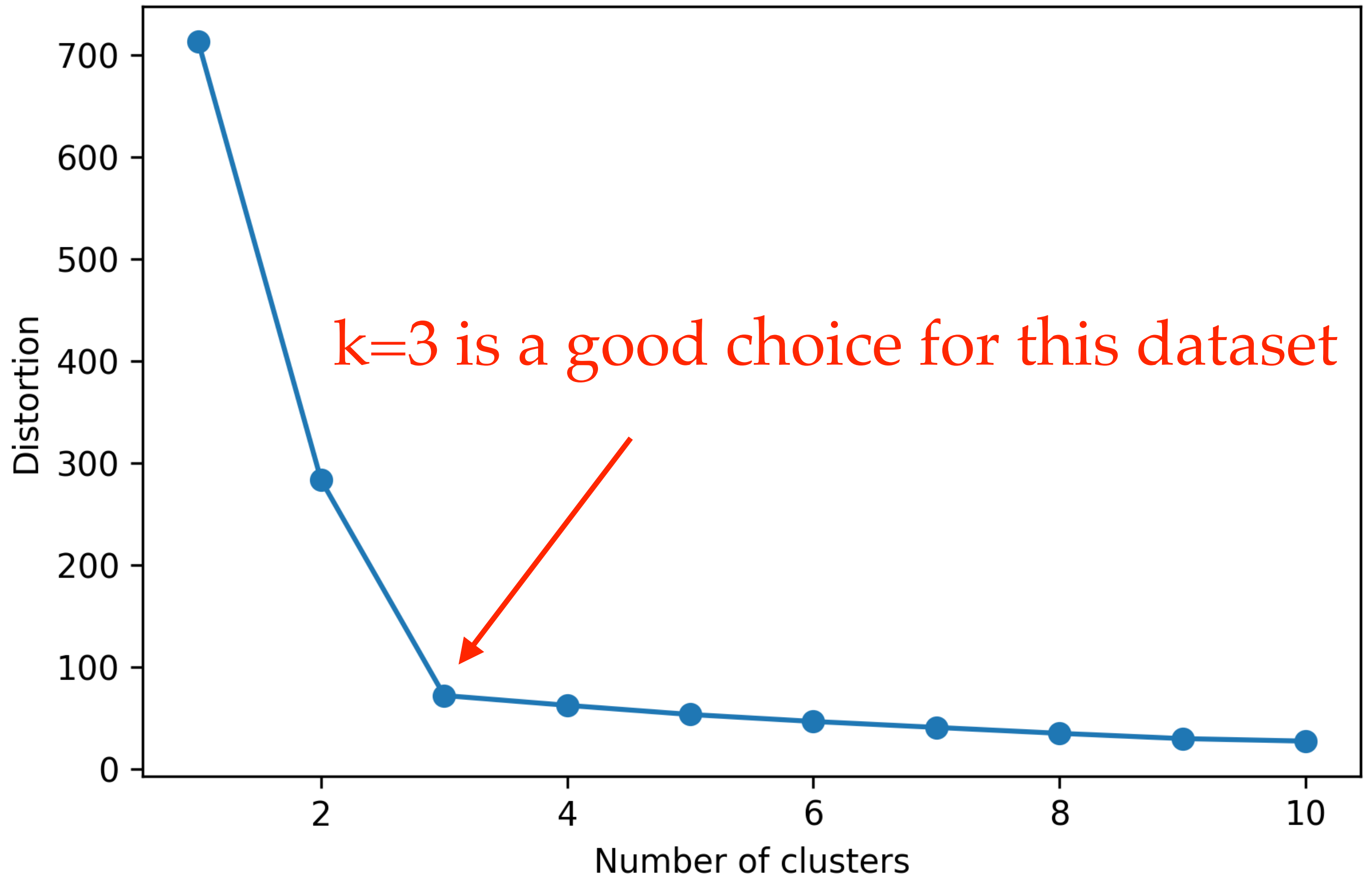
```
print('Distortion: %.2f' % km.inertia_)
```

```
Distortion: 72.48
```

## Using the Elbow Method to Find the Optimal Number of Clusters

- Intuitively, if *k* increases, the distortion will decrease
  - Samples will be closer to the centroids they are assigned to
- The idea of elbow method is to identify the value of *k* where the distortion begins to increase most rapidly

# An Elbow Method Example



```
distortions = []
for i in range(1, 11):
    km = KMeans(n_clusters=i,
                init='k-means++',
                n_init=10,
                max_iter=300,
                random_state=0)
    km.fit(X)
    distortions.append(km.inertia_)
plt.plot(range(1, 11), distortions, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Distortion')
plt.tight_layout()
#plt.savefig('images/11_03.png', dpi=300)
plt.show()
```

## Quantifying the Quality of Clustering via Silhouette Plots

- Silhouette analysis can be used as a graphic tool to plot a measure of how tightly grouped the samples in the clusters are
- Steps to calculate the silhouette coefficient
  - Calculate the cluster cohesion $\boldsymbol{a}^{(i)}$ as the average distance between a sample $\boldsymbol{x}^{(i)}$ and all other points in the same cluster
  - Calculate the cluster separation $\boldsymbol{b}^{(i)}$ from the next closet cluster as the average distance between the sample $\boldsymbol{x}^{(i)}$ and all samples in the nearest cluster
  - Calculate the silhouette $\boldsymbol{s}^{(i)}$ as the difference between cluster cohesion and separation divided by the greater of the two, as

$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{\max\left\{b^{(i)}, a^{(i)}\right\}}$$

Laboratory for Reliable Computing

## Quantifying the Quality of Clustering via Silhouette Plots

- The silhouette coefficient is available as **silhouette_samples** from scikit-learn's **metric** module
- The **silhouette_scores** function calculates the average silhouette coefficient across all samples, which is equivalent to numpy.mean(silhouette_samples(…))

Laboratory for Reliable Computing

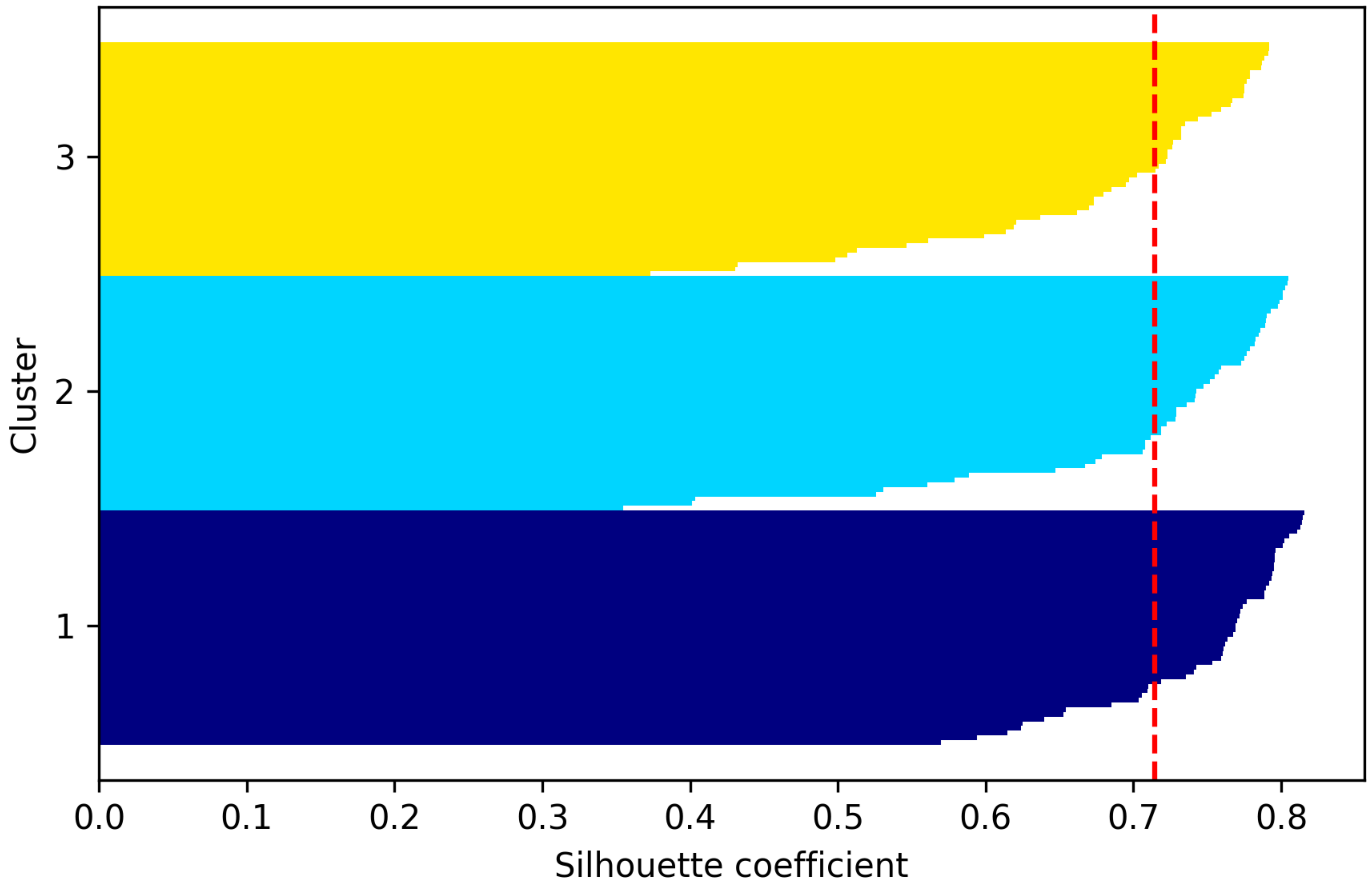
# A Silhouette Plot Example (1/3)

```
import numpy as np
from matplotlib import cm
from sklearn.metrics import silhouette_samples

km = KMeans(n_clusters=3,
            init='k-means++',
            n_init=10,
            max_iter=300,
            tol=1e-04,
            random_state=0)
y_km = km.fit_predict(X)

cluster_labels = np.unique(y_km)
n_clusters = cluster_labels.shape[0]
silhouette_vals = silhouette_samples(X, y_km, metric='euclidean')
y_ax_lower, y_ax_upper = 0, 0
yticks = []
```

# A Silhouette Plot Example (2/3)

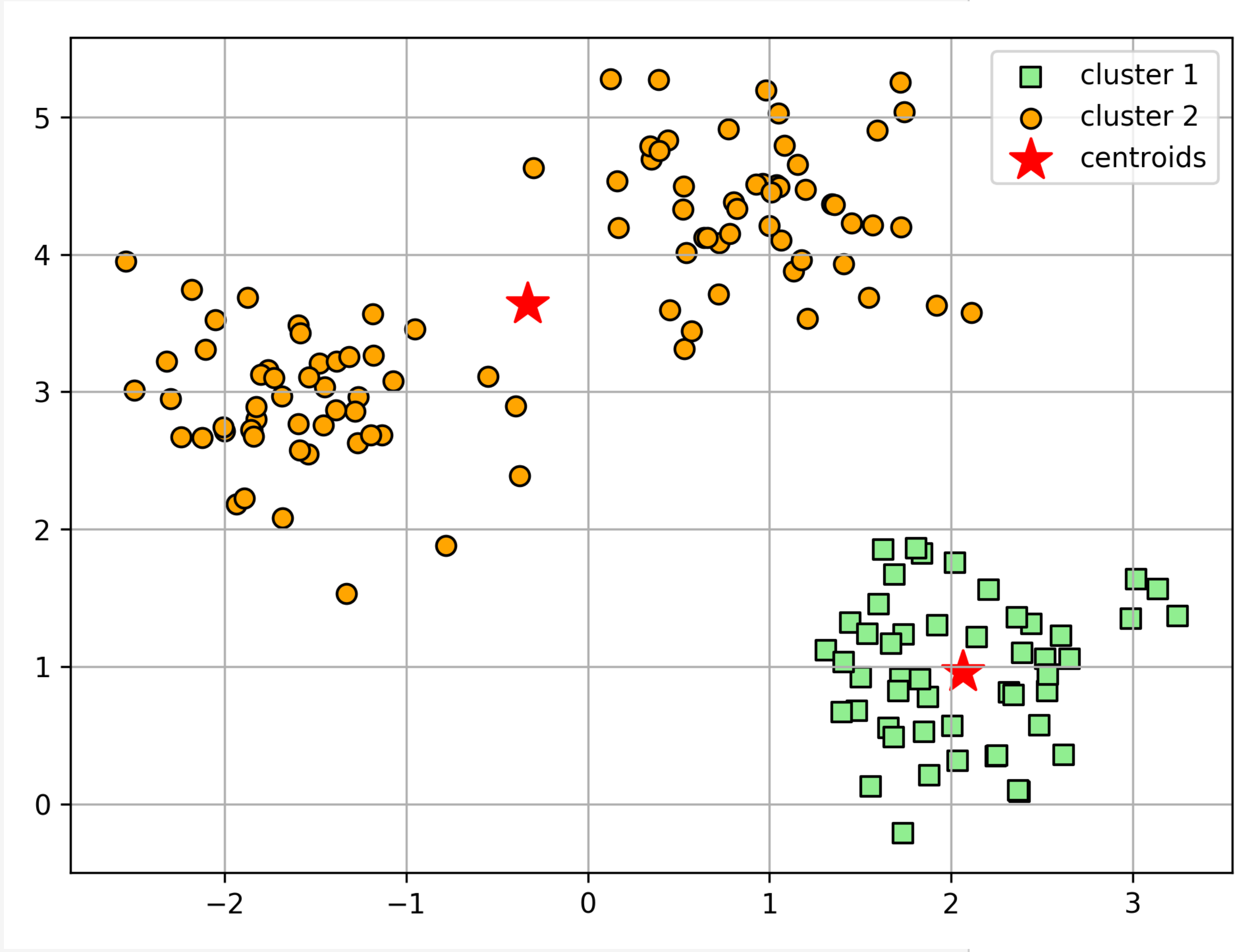
```
for i, c in enumerate(cluster_labels):
    c_silhouette_vals = silhouette_vals[y_km == c]
    c_silhouette_vals.sort()
    y_ax_upper += len(c_silhouette_vals)
    color = cm.jet(float(i) / n_clusters)
    plt.barh(range(y_ax_lower, y_ax_upper), c_silhouette_vals, height=1.0,
             edgecolor='none', color=color)

    yticks.append((y_ax_lower + y_ax_upper) / 2.)
    y_ax_lower += len(c_silhouette_vals)

silhouette_avg = np.mean(silhouette_vals)
plt.axvline(silhouette_avg, color="red", linestyle="--")

plt.yticks(yticks, cluster_labels + 1)
plt.ylabel('Cluster')
plt.xlabel('Silhouette coefficient')

plt.tight_layout()
#plt.savefig('images/11_04.png', dpi=300)
plt.show()
```

## A Silhouette Plot Example (3/3)



## Silhouette Plot for “Bad” Clustering (1/3)

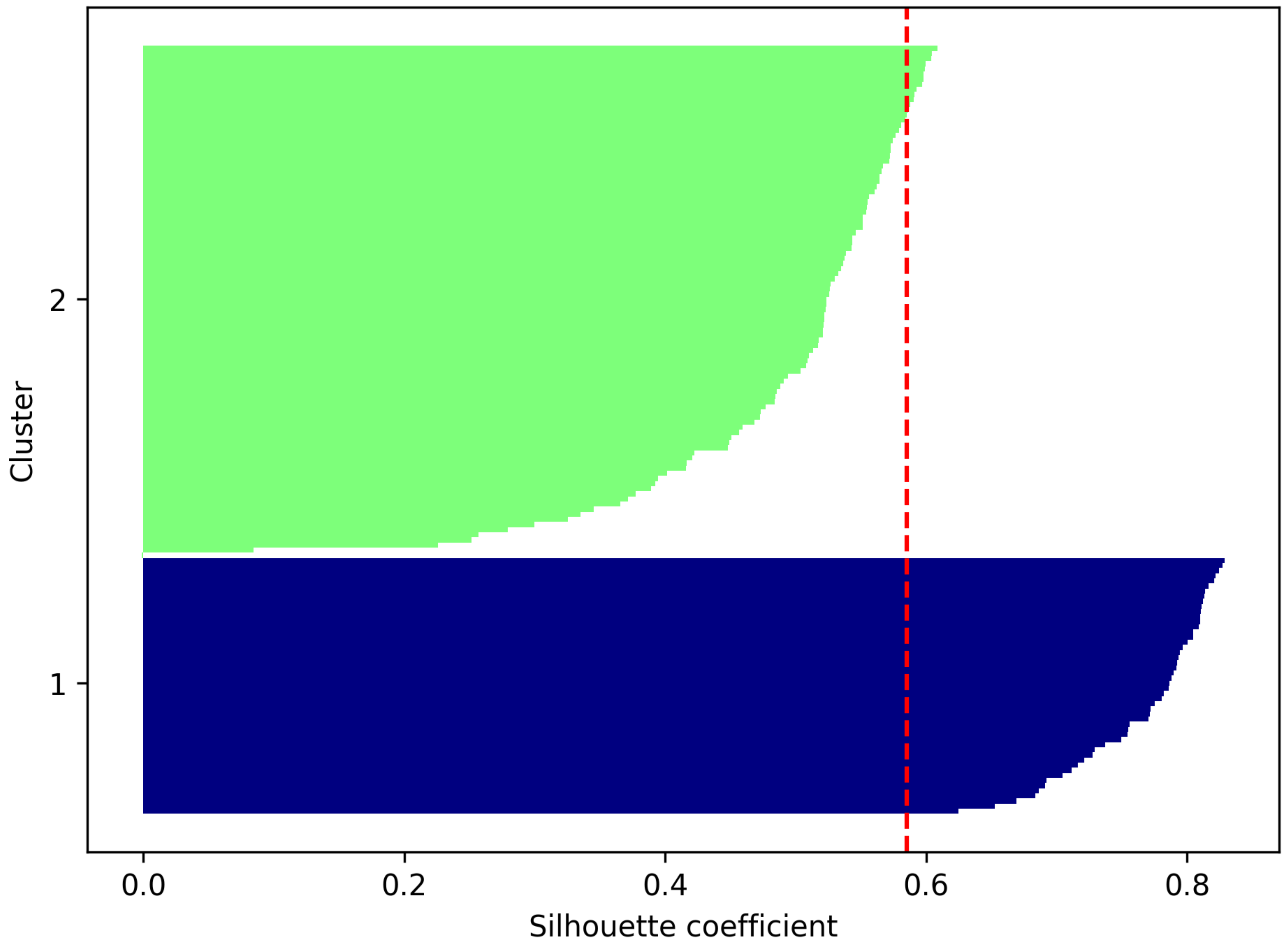
```
km = KMeans(n_clusters=2,
            init='k-means++',
            n_init=10,
            max_iter=300,
            tol=1e-04,
            random_state=0)
y_km = km.fit_predict(X)

plt.scatter(X[y_km == 0, 0],
            X[y_km == 0, 1],
            s=50,
            c='lightgreen',
            edgecolor='black',
            marker='s',
            label='cluster 1')
plt.scatter(X[y_km == 1, 0],
            X[y_km == 1, 1],
            s=50,
            c='orange',
            edgecolor='black',
            marker='o',
            label='cluster 2')
plt.scatter(km.cluster_centers_[:, 0], km.cluster_centers_[:, 1],
            s=250, marker='*', c='red', label='centroids')
plt.legend()
plt.grid()
plt.tight_layout()
#plt.savefig('images/11_05.png', dpi=300)
plt.show()
```



# Silhouette Plot for “Bad” Clustering (2/3)

```
cluster_labels = np.unique(y_km)
n_clusters = cluster_labels.shape[0]
silhouette_vals = silhouette_samples(X, y_km, metric='euclidean')
y_ax_lower, y_ax_upper = 0, 0
yticks = []
for i, c in enumerate(cluster_labels):
    c_silhouette_vals = silhouette_vals[y_km == c]
    c_silhouette_vals.sort()
    y_ax_upper += len(c_silhouette_vals)
    color = cm.jet(float(i) / n_clusters)
    plt.barh(range(y_ax_lower, y_ax_upper), c_silhouette_vals, height=1.0,
             edgecolor='none', color=color)

    yticks.append((y_ax_lower + y_ax_upper) / 2.)
    y_ax_lower += len(c_silhouette_vals)

silhouette_avg = np.mean(silhouette_vals)
plt.axvline(silhouette_avg, color="red", linestyle="--")
plt.yticks(yticks, cluster_labels + 1)
plt.ylabel('Cluster')
plt.xlabel('Silhouette coefficient')

plt.tight_layout()
#plt.savefig('images/11_06.png', dpi=300)
plt.show()
```

## Silhouette Plot for “Bad” Clustering (3/3)



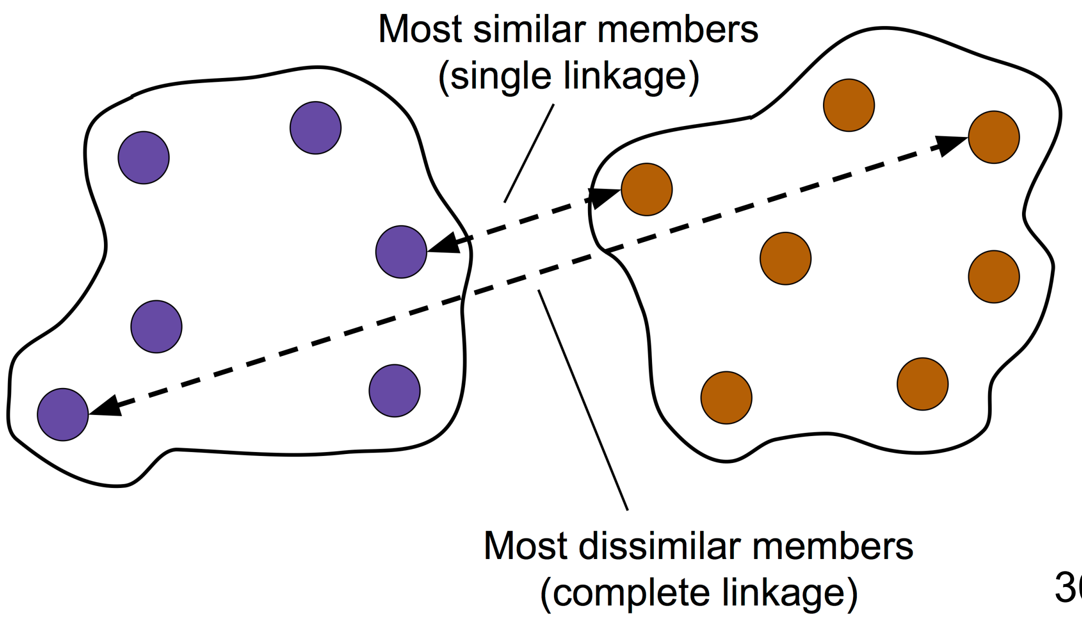
## Hierarchical Clustering

## Hierarchical Clustering

- Advantages
  - Plot dendrograms (visualization of a binary hierarchical clustering) for interpretation of the results by creating meaningful taxonomies
  - Do not need to specify the number of clusters beforehand
- Two approaches
  - Divisive hierarchical clustering
    - A top-down approach which starts with one cluster that encompasses all points in the dataset, and iteratively split the cluster into smaller clusters until each cluster only contains one instance
  - **Agglomerative hierarchical clustering**
    - A bottom-up approach which starts with each instance forming an individual cluster and merges the closest pairs of clusters until only one cluster remains

## Linkage-Based Clustering

- Agglomerative iteration
  - Merge two clusters to form a new cluster if the distance is the smallest among all pairs of clusters until a stopping criteria in reached
    - Single linkage (Min linkage)
      - Compute the distances between most similar members for each pair of clusters
      - Merge the two cluster for each other which the distance between the most similar members is the smallest
    - Complete linkage (Max linkage)
      - Similar to single linkage but compare distance between the most dissimilar members to perform the merge
  - Stopping criteria
    - A fixed number *k* of clusters is reached
    - An upper bound *r* of cluster distance is broken



## Complete Linkage Clustering

- Compute the distance matrix of all samples
- Represent each data point as a singleton cluster
- Merge the two closest clusters based on the distance between the most dissimilar (distant) members
- Update the similarity matrix
- Repeat steps 2-4 until only one single cluster remains

## Sample Data Generation for Demo

```python
import pandas as pd
import numpy as np

np.random.seed(123)

variables = ['X', 'Y', 'Z']
labels = ['ID_0', 'ID_1', 'ID_2', 'ID_3', 'ID_4']

X = np.random.random_sample([5, 3])*10
df = pd.DataFrame(X, columns=variables, index=labels)
df
```

| | X | Y | Z |
|---|---|---|---|
| **ID_0** | 6.964692 | 2.861393 | 2.268515 |
| **ID_1** | 5.513148 | 7.194690 | 4.231065 |
| **ID_2** | 9.807642 | 6.848297 | 4.809319 |
| **ID_3** | 3.921175 | 3.431780 | 7.290497 |
| **ID_4** | 4.385722 | 0.596779 | 3.980443 |

## Performing Hierarchical Clustering on a Distance Matrix

```python
from scipy.spatial.distance import pdist, squareform

row_dist = pd.DataFrame(squareform(pdist(df, metric='euclidean')),
                        columns=labels,
                        index=labels)
row_dist
```

| | ID_0 | ID_1 | ID_2 | ID_3 | ID_4 |
|---|---|---|---|---|---|
| **ID_0** | 0.000000 | 4.973534 | 5.516653 | 5.899885 | 3.835396 |
| **ID_1** | 4.973534 | 0.000000 | 4.347073 | 5.104311 | 6.698233 |
| **ID_2** | 5.516653 | 4.347073 | 0.000000 | 7.244262 | 8.316594 |
| **ID_3** | 5.899885 | 5.104311 | 7.244262 | 0.000000 | 4.382864 |
| **ID_4** | 3.835396 | 6.698233 | 8.316594 | 4.382864 | 0.000000 |

## Approaches for Hierarchical Clustering (1/3)

```python
# 1. incorrect approach: Squareform distance matrix

from scipy.cluster.hierarchy import linkage

row_clusters = linkage(row_dist, method='complete', metric='euclidean')
pd.DataFrame(row_clusters,
             columns=['row label 1', 'row label 2',
                      'distance', 'no. of items in clust.'],
             index=['cluster %d' % (i + 1)
                    for i in range(row_clusters.shape[0])])
```

| | row label 1 | row label 2 | distance | no. of items in clust. |
|---|---|---|---|---|
| **cluster 1** | 0.0 | 4.0 | 6.521973 | 2.0 |
| **cluster 2** | 1.0 | 2.0 | 6.729603 | 2.0 |
| **cluster 3** | 3.0 | 5.0 | 8.539247 | 3.0 |
| **cluster 4** | 6.0 | 7.0 | 12.444824 | 5.0 |

## Approaches for Hierarchical Clustering (2/3)

```
# 2. correct approach: Condensed distance matrix

row_clusters = linkage(pdist(df, metric='euclidean'), method='complete')
pd.DataFrame(row_clusters,
             columns=['row label 1', 'row label 2',
                      'distance', 'no. of items in clust.'],
             index=['cluster %d' % (i + 1)
                    for i in range(row_clusters.shape[0])])
```

| | row label 1 | row label 2 | distance | no. of items in clust. |
|---|---|---|---|---|
| **cluster 1** | 0.0 | 4.0 | 3.835396 | 2.0 |
| **cluster 2** | 1.0 | 2.0 | 4.347073 | 2.0 |
| **cluster 3** | 3.0 | 5.0 | 5.899885 | 3.0 |
| **cluster 4** | 6.0 | 7.0 | 8.316594 | 5.0 |

# Approaches for Hierarchical Clustering (3/3)

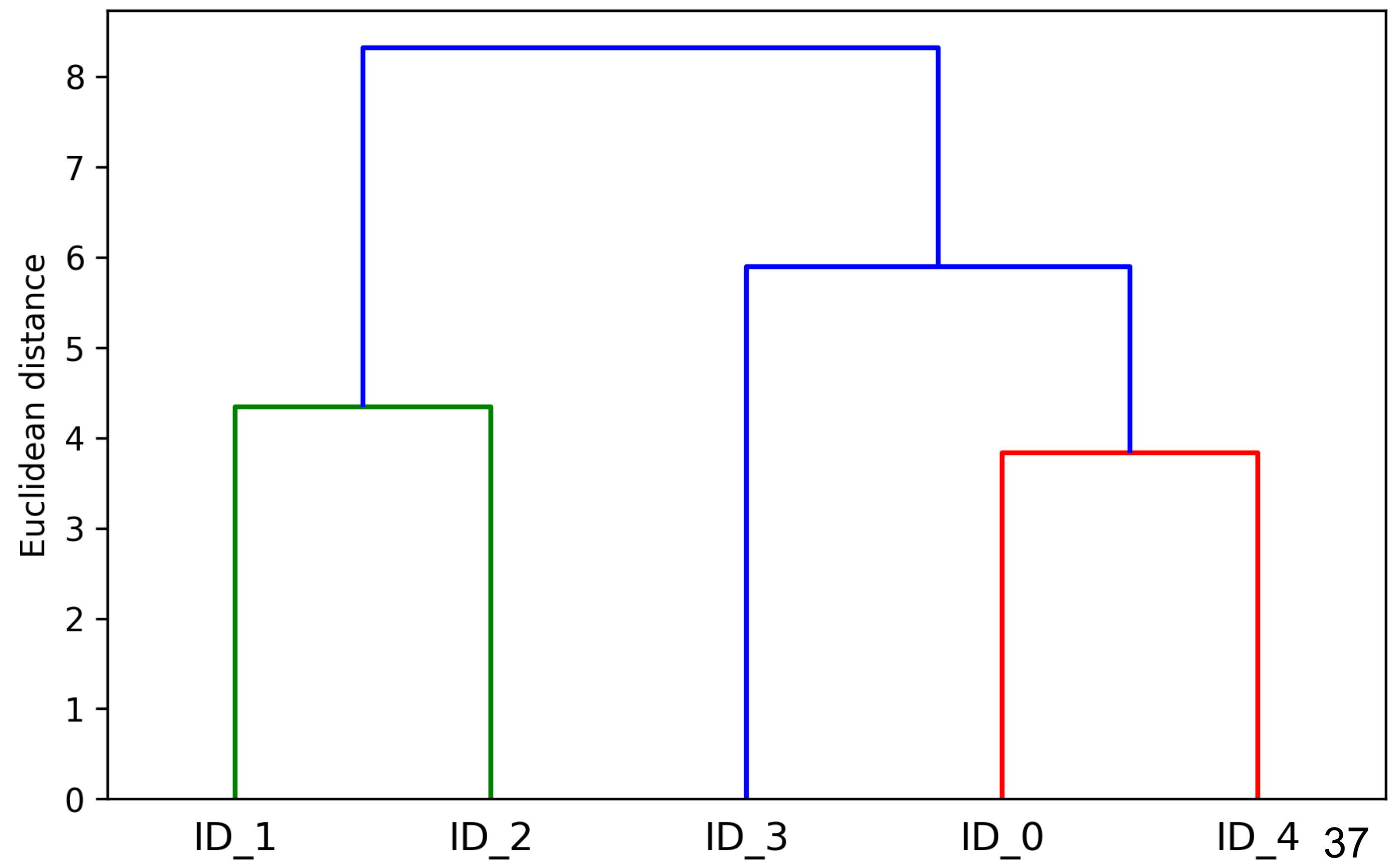
```python
# 3. correct approach: Input sample matrix

row_clusters = linkage(df.values, method='complete', metric='euclidean')
pd.DataFrame(row_clusters,
             columns=['row label 1', 'row label 2',
                      'distance', 'no. of items in clust.'],
             index=['cluster %d' % (i + 1)
                    for i in range(row_clusters.shape[0])])
```

| | row label 1 | row label 2 | distance | no. of items in clust. |
|---|---|---|---|---|
| **cluster 1** | 0.0 | 4.0 | 3.835396 | 2.0 |
| **cluster 2** | 1.0 | 2.0 | 4.347073 | 2.0 |
| **cluster 3** | 3.0 | 5.0 | 5.899885 | 3.0 |
| **cluster 4** | 6.0 | 7.0 | 8.316594 | 5.0 |

# Dendrogram for Visualization

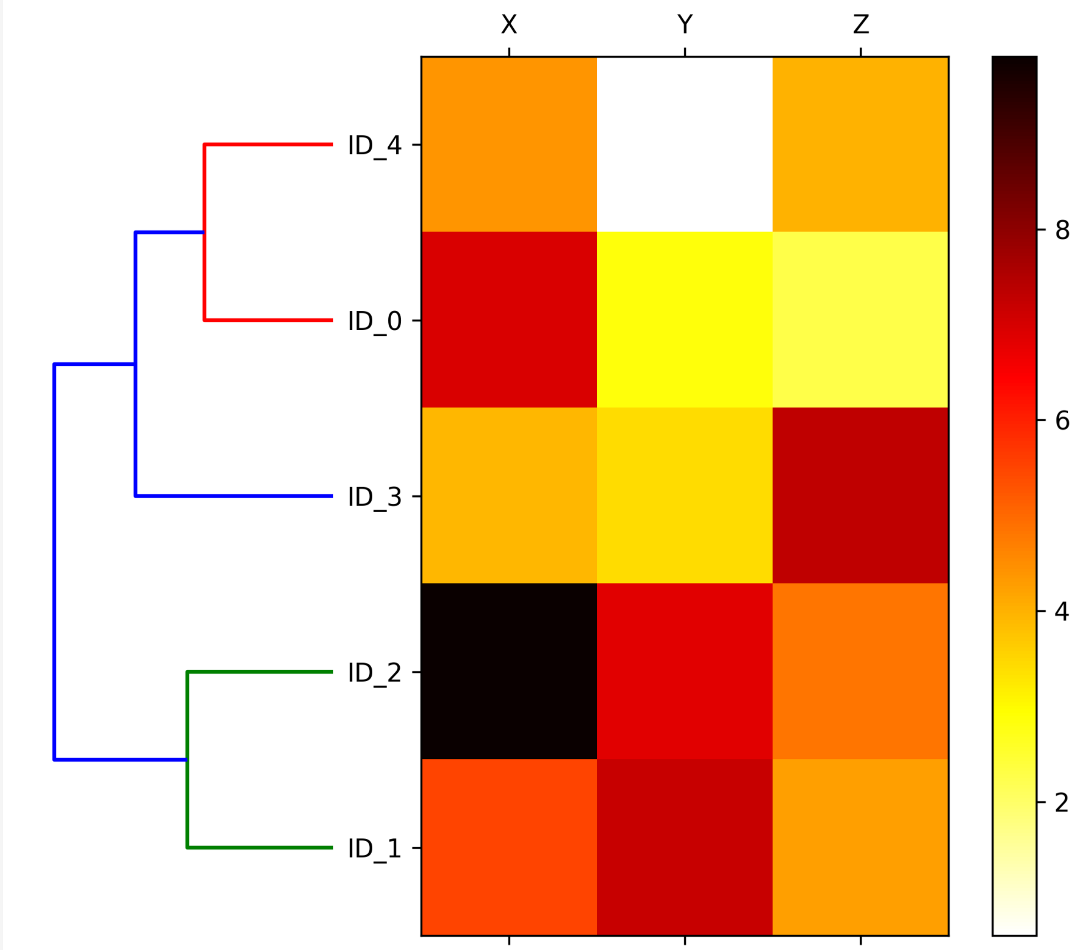
```
from scipy.cluster.hierarchy import dendrogram

# make dendrogram black (part 1/2)
# from scipy.cluster.hierarchy import set_link_color_palette
# set_link_color_palette(['black'])

row_dendr = dendrogram(row_clusters,
                       labels=labels,
                       # make dendrogram black (part 2/2)
                       # color_threshold=np.inf
                       )
plt.tight_layout()
plt.ylabel('Euclidean distance')
#plt.savefig('images/11_11.png', dpi=300,
#            bbox_inches='tight')
plt.show()
```

dissimilarity



# Attaching Dendrograms to a Heat Map

```
# plot row dendrogram
fig = plt.figure(figsize=(8, 8), facecolor='white')
axd = fig.add_axes([0.09, 0.1, 0.2, 0.6])

# note: for matplotlib < v1.5.1, please use orientation='right'
row_dendr = dendrogram(row_clusters, orientation='left')

# reorder data with respect to clustering
df_rowclust = df.iloc[row_dendr['leaves'][::-1]]

axd.set_xticks([])
axd.set_yticks([])

# remove axes spines from dendrogram
for i in axd.spines.values():
    i.set_visible(False)

# plot heatmap
axm = fig.add_axes([0.23, 0.1, 0.6, 0.6])  # x-pos, y-pos, width, height
cax = axm.matshow(df_rowclust, interpolation='nearest', cmap='hot_r')
fig.colorbar(cax)
axm.set_xticklabels([''] + list(df_rowclust.columns))
axm.set_yticklabels([''] + list(df_rowclust.index))

#plt.savefig('images/11_12.png', dpi=300)
plt.show()
```



## Applying Agglomerative Clustering via Scikit-learn

- Use **AgglomerativeClustering** and set **n_clusters**

```
from sklearn.cluster import AgglomerativeClustering

ac = AgglomerativeClustering(n_clusters=3,
                             affinity='euclidean',
                             linkage='complete')
labels = ac.fit_predict(X)
print('Cluster labels: %s' % labels)
```

```
Cluster labels: [1 0 0 2 1]
```

```
ac = AgglomerativeClustering(n_clusters=2,
                             affinity='euclidean',
                             linkage='complete')
labels = ac.fit_predict(X)
print('Cluster labels: %s' % labels)
```

```
Cluster labels: [0 1 1 0 0]
```

## Locating Regions of High Density via DBSCAN

## DBSCAN

- Density-based Spatial Clustering of Applications with Noise
  - DBSCAN does not make assumptions about spherical clusters like *k*-means, nor it partition the dataset into a hierarchical tree that requires a manual cut-off points
  - Assign cluster labels based on dense region of points
  - Density is defined as the number of points within a specific radius $\varepsilon$
  - Given a set of points in some space, it groups together points that are closely packed together, marking as outliers points that lie alone in low-density regions

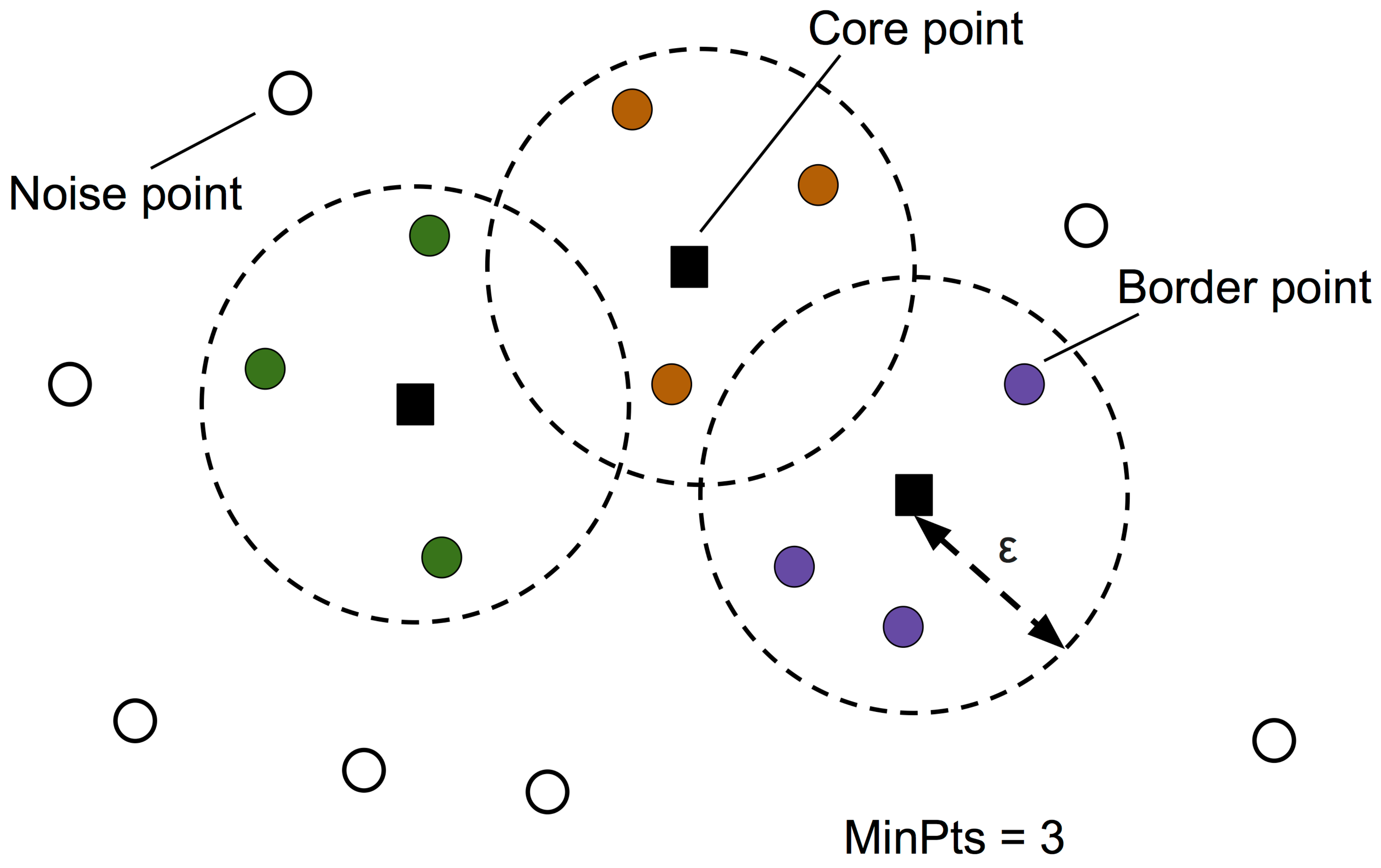
## DBSCAN Algorithm (1/3)

- Step 1: A special label is assigned to each data point using the following criteria
  - A **core** point: a point which has at least a specific number (MinPts) of neighboring points falling within the specified radius $\varepsilon$
  - A **border** point: a point which has fewer neighbors than MinPts within the specified radius $\varepsilon$ but lies within the $\varepsilon$ radius of a core point
  - A **noise** point: a point which is neither a core nor a border point

## DBSCAN Algorithm (2/3)

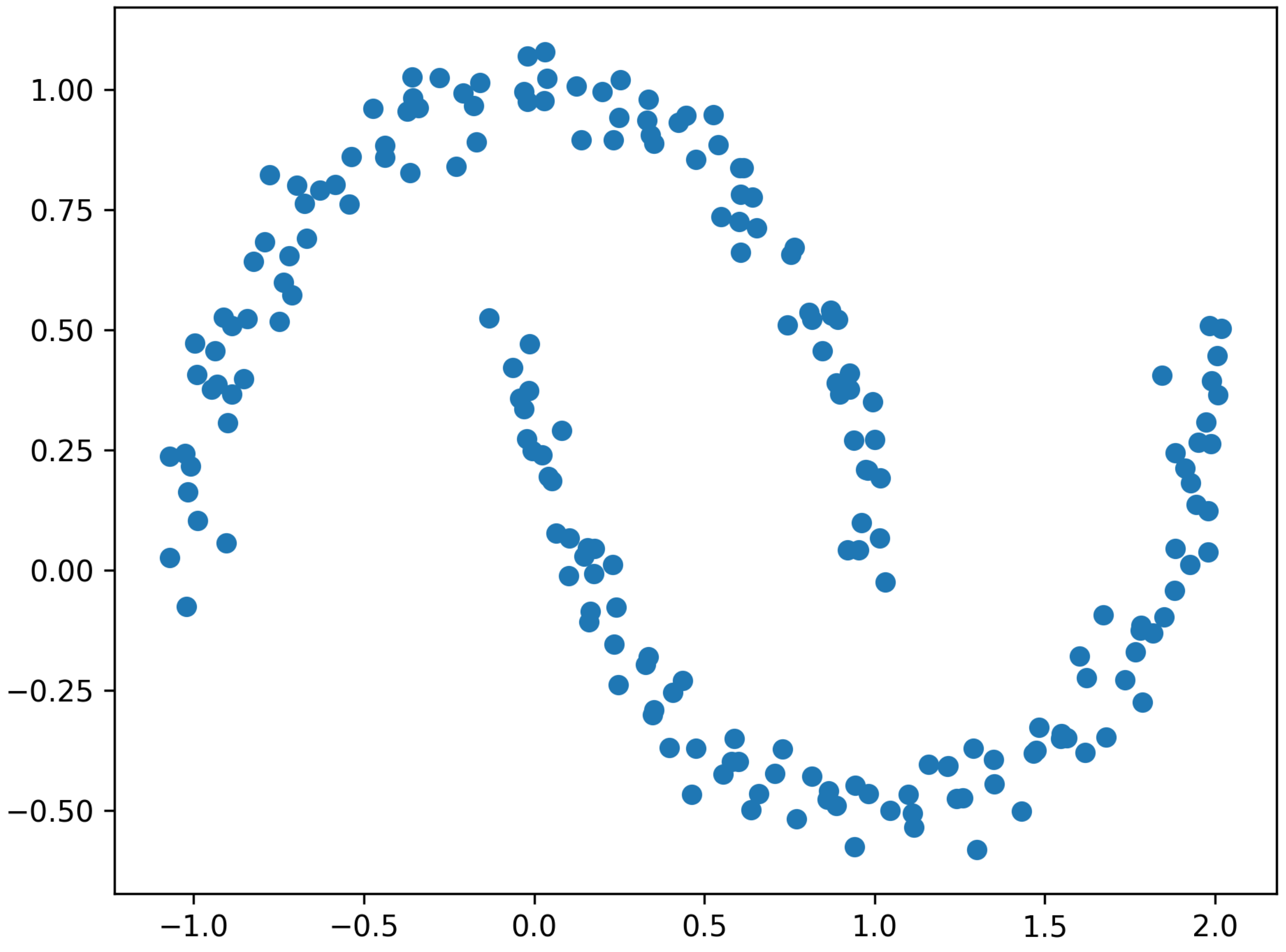
- Step 2: Form a separate cluster for each disconnected core point or for a connected group of core points
  - Two core points are connected (by an edge) if they are no farther away than $\varepsilon$. This establish a graph of core points
  - A connected group of core points is a (path-)connected component of the graph of core points
  - A disconnected core point is a core point which forms a (path-)connected component by itself in the graph of core points
- Step 3: Assign each border point to the cluster of its corresponding core points

## DBSCAN Algorithm (3/3)



# Half-moon-shaped Dataset

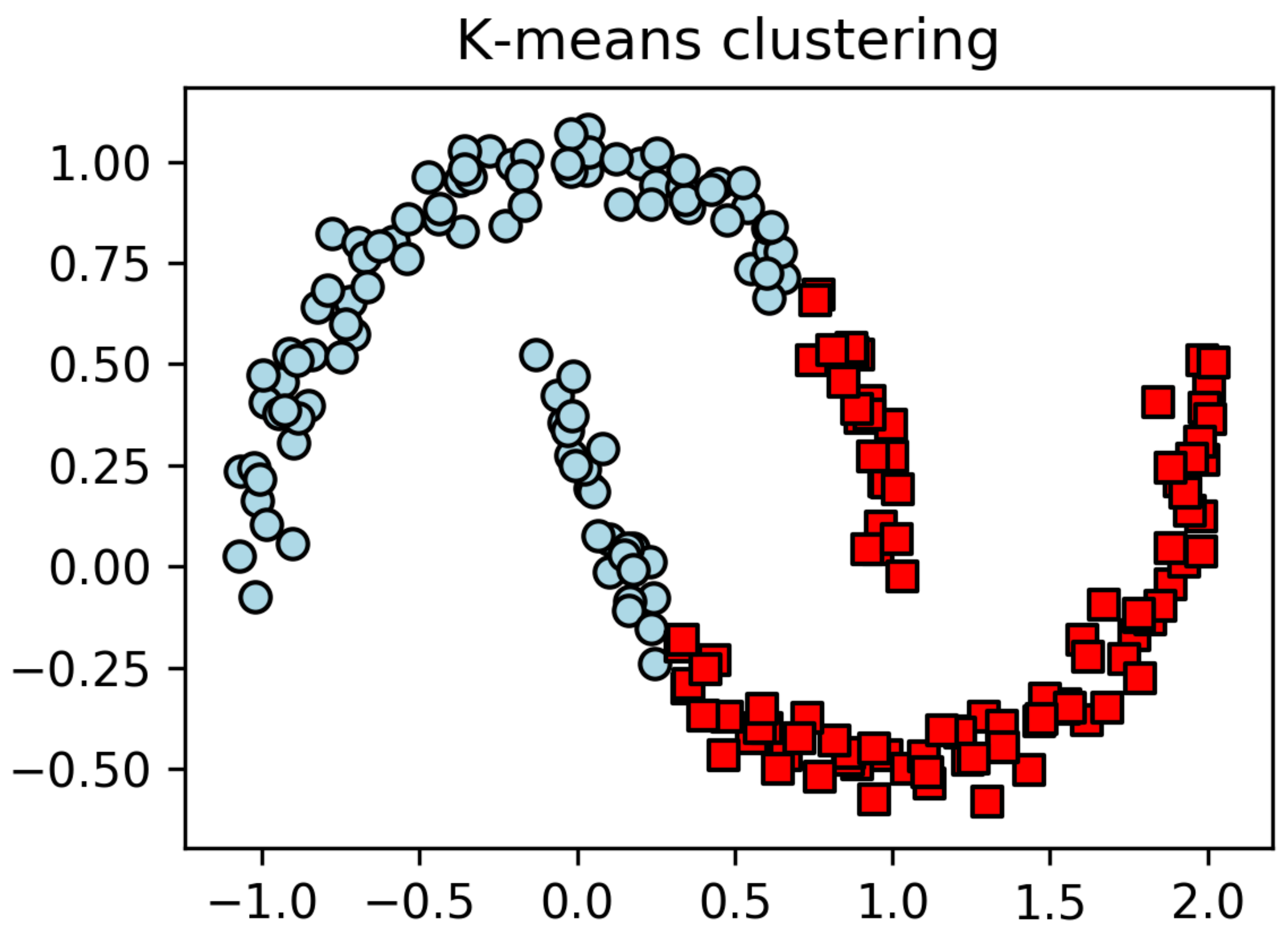
```
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=200, noise=0.05, random_state=0)
plt.scatter(X[:, 0], X[:, 1])
plt.tight_layout()
#plt.savefig('images/11_14.png', dpi=300)
plt.show()
```

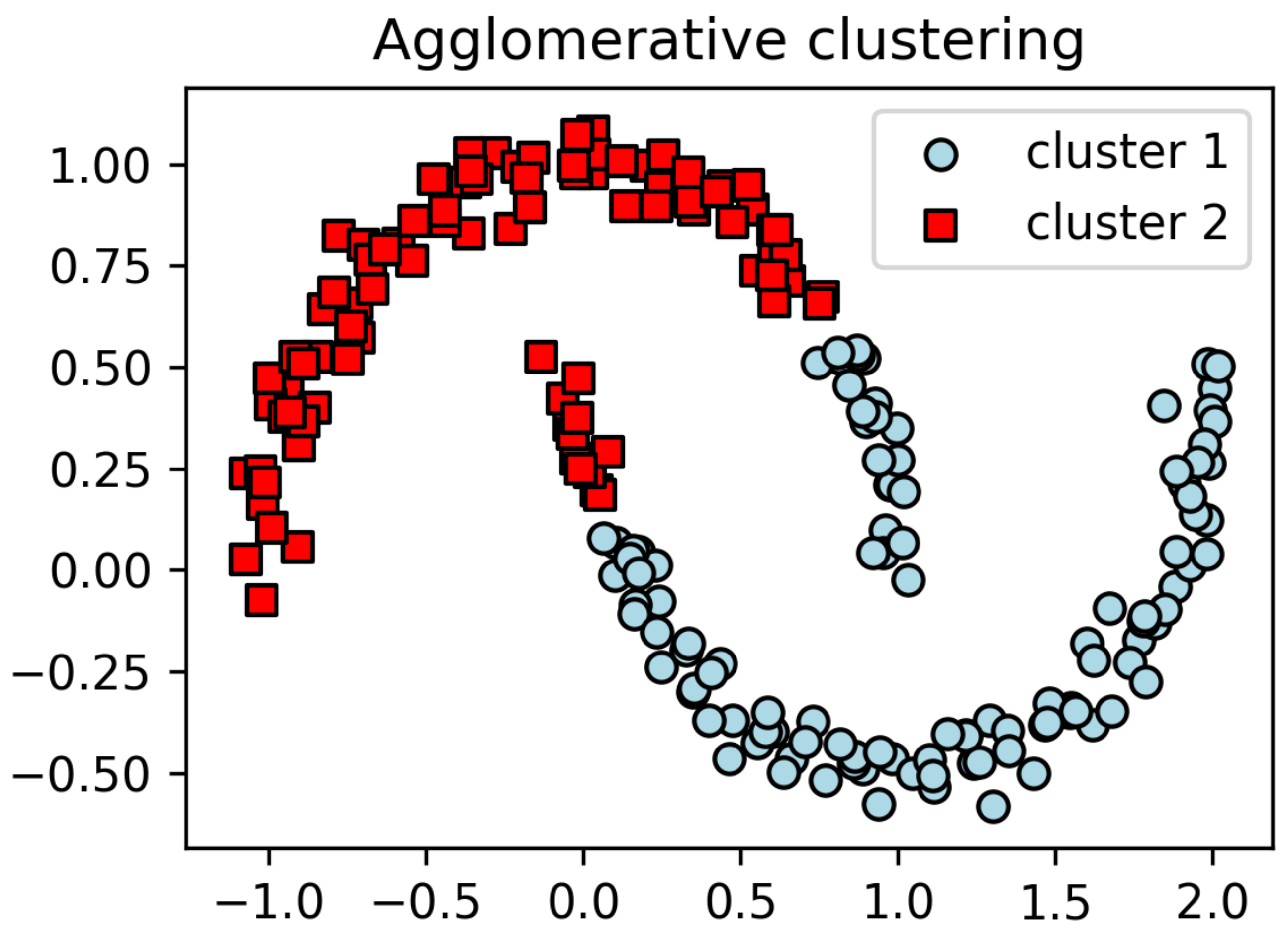


## Use k-means and Hierarchical Clustering

```
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3))

km = KMeans(n_clusters=2, random_state=0)
y_km = km.fit_predict(X)
ax1.scatter(X[y_km == 0, 0], X[y_km == 0, 1],
            edgecolor='black',
            c='lightblue', marker='o', s=40, label='cluster 1')
ax1.scatter(X[y_km == 1, 0], X[y_km == 1, 1],
            edgecolor='black',
            c='red', marker='s', s=40, label='cluster 2')
ax1.set_title('K-means clustering')

ac = AgglomerativeClustering(n_clusters=2,
                             affinity='euclidean',
                             linkage='complete')
y_ac = ac.fit_predict(X)
ax2.scatter(X[y_ac == 0, 0], X[y_ac == 0, 1], c='lightblue',
            edgecolor='black',
            marker='o', s=40, label='cluster 1')
ax2.scatter(X[y_ac == 1, 0], X[y_ac == 1, 1], c='red',
            edgecolor='black',
            marker='s', s=40, label='cluster 2')
ax2.set_title('Agglomerative clustering')

plt.legend()
plt.tight_layout()
# plt.savefig('images/11_15.png', dpi=300)
plt.show()
```

# k-means and Hierarchical Clustering

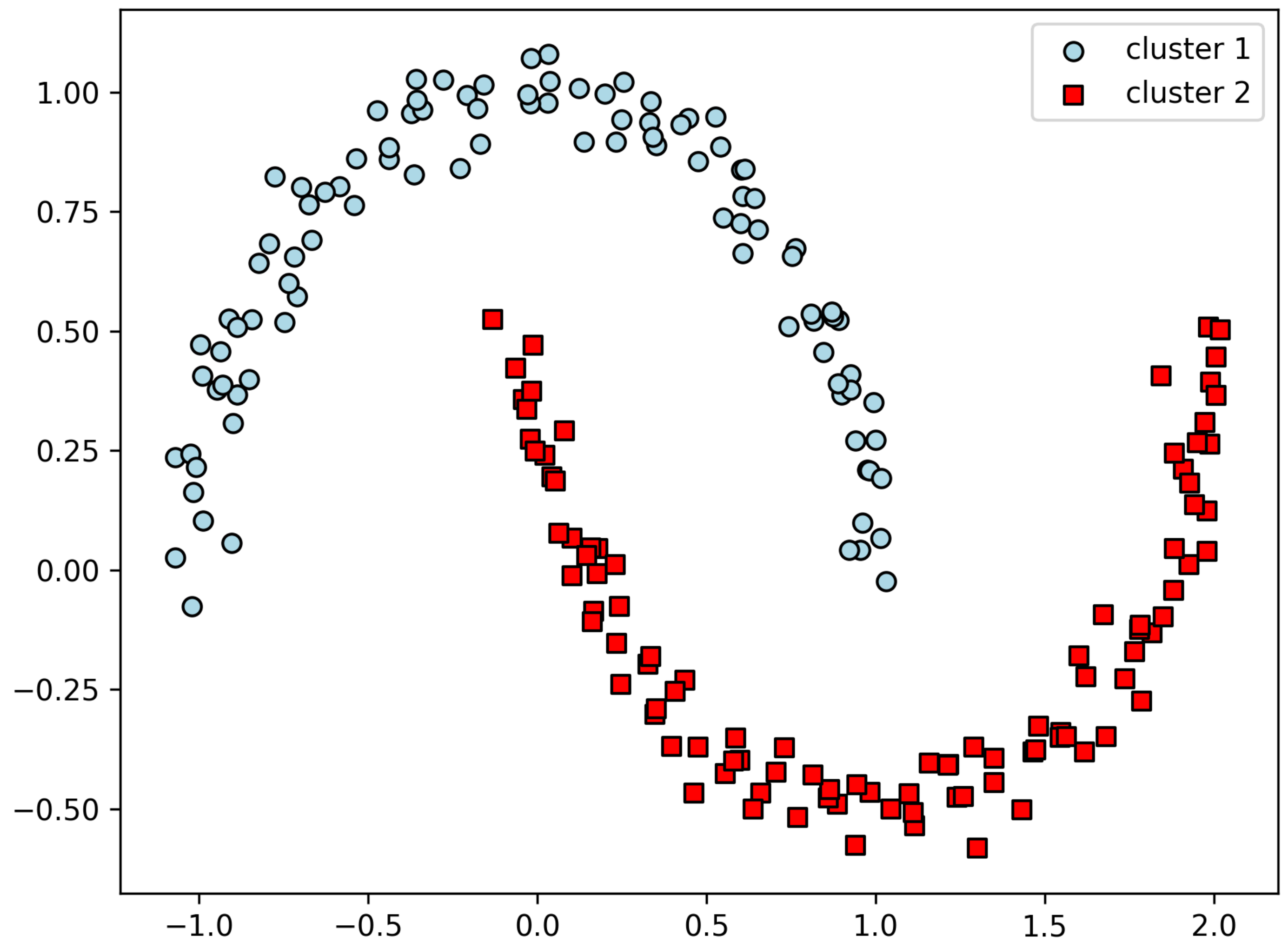




# DBSCAN

```python
from sklearn.cluster import DBSCAN

db = DBSCAN(eps=0.2, min_samples=5, metric='euclidean')
y_db = db.fit_predict(X)
plt.scatter(X[y_db == 0, 0], X[y_db == 0, 1],
            c='lightblue', marker='o', s=40,
            edgecolor='black',
            label='cluster 1')
plt.scatter(X[y_db == 1, 0], X[y_db == 1, 1],
            c='red', marker='s', s=40,
            edgecolor='black',
            label='cluster 2')
plt.legend()
plt.tight_layout()
#plt.savefig('images/11_16.png', dpi=300)
plt.show()
```

Clustering data of any shapes



## Issues of DBSCAN

- Two hyperparameters, i.e., MinPts and $\varepsilon$ to be optimized
- Finding a good combination of MinPts and $\varepsilon$ can be problematic if the density differences in the dataset are relatively large
- curse of dimensionality increases as increasing number of features and fixed number of training samples, especially when using Euclidean distance metric

## Common Practices for Clustering Algorithms

- To reduce the curse of dimensionality, apply unsupervised dimensionality reduction techniques prior to performing clustering, such as PCA or RBF-kernel PCA
- To visualize the clusters, compress datasets down to 2D subspace. This is particularly helpful for evaluating results
- A successful clustering not only depends on the algorithm and its hyperparameters, but also on the ***choice of an appropriate distance metric*** and the use of ***domain knowledge***