

Working with Unlabeled Data

Clustering Analysis

Aprendizaje Automático

Ingeniería de Robótica Software

Universidad Rey Juan Carlos

Content

- Introduction
- Grouping objects by similarity using k-means
- Organizing clusters as a hierarchical tree
- Locating regions of high density via DBSCAN

Introduction

- **Supervised learning** techniques build machine learning models using data where the answer was already known.
 - For example, in classification, the class labels are available in training data.
- **Unsupervised learning** techniques build machine learning models that allows us to discover hidden structures in data where we do not know the right answer upfront.
 - For example, in **clustering**, the model attempts to **find a natural grouping in data** so that items in the same cluster are more similar to each other than to those from different clusters

Grouping objects by similarity using k-means

- The **k-means** algorithm is one of the most popular **clustering algorithms**, which is widely used in academia as well as in industry.
- Try to find **groups of similar objects** that are more related to each other than to objects in other groups.
- Examples of business-oriented applications of clustering include the **grouping of documents, music, and movies by different topics**, or **finding customers** that share similar interests based on common purchase behaviors as a basis for recommendation engines

Grouping objects by similarity using k-means

- The **k-means** algorithm is extremely **easy to implement**, and it is also **computationally very efficient** compared to other clustering algorithms.
- The k-means algorithm belongs to the category of **prototype-based** clustering.
 - There are other categories of clustering, such as **hierarchical** and **density-based** clustering.

Grouping objects by similarity using k-means

- **Prototype-based clustering** means that **each cluster is represented by a prototype**, which is usually either the **centroid** (average) of similar points with continuous features, or the **medoid** (the most representative or the point that minimizes the distance to all other points that belong to a particular cluster) in the case of categorical features.

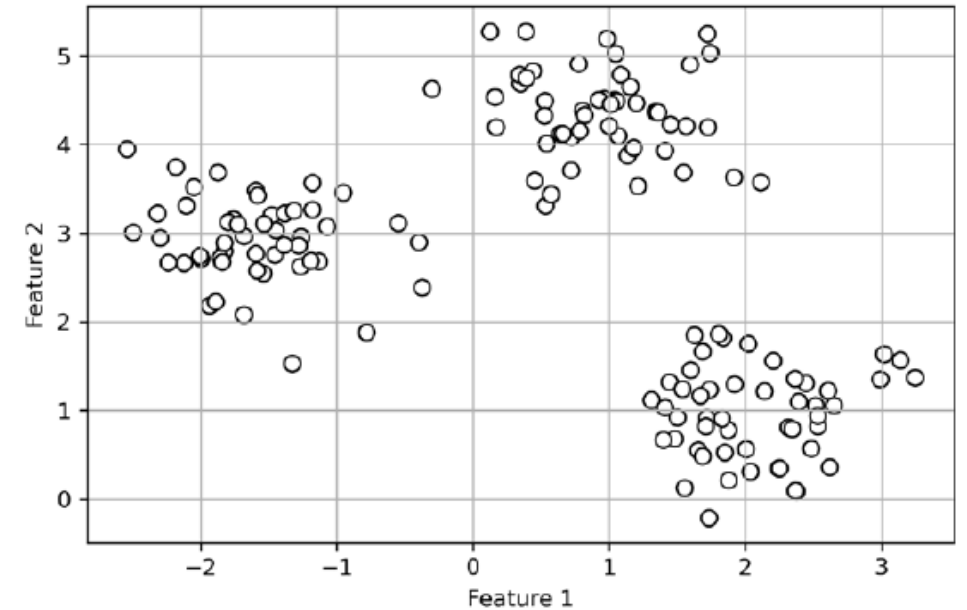
Grouping objects by similarity using k-means

- **K-means** is very good at identifying clusters with a **spherical shape**.
 - This is because the algorithm minimizes the sum of the squared distances between the data points and the cluster centroid. In a Euclidean space, this minimization tends to form clusters that are spherical around their centroids.
- One of the **drawbacks** of this clustering algorithm is that we have to **specify the number of clusters, k , a priori**. An inappropriate choice for k can result in poor clustering performance.
- The **elbow method** and **silhouette plots** are useful techniques to evaluate the quality of a clustering to **help us determine the optimal number of clusters, k** .

Grouping objects by similarity using k-means

Although k-means clustering can be applied to data in higher dimensions, we will walk through the following examples using a simple two-dimensional dataset for the purpose of 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.xlabel('Feature 1')
>>> plt.ylabel('Feature 2')
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```



A scatterplot of our unlabeled dataset

Grouping objects by similarity using k-means

- The goal is to **group the examples based on their feature similarities** using the k-means algorithm, as summarized by the following four steps:
 1. **Randomly pick k centroids** from the examples **as initial cluster centers**.
 2. **Assign each example to the nearest (more similar) centroid**, $\mu^{(j)}$, $j \in \{1, \dots, k\}$.
 3. **Move the centroids to the center of the examples** that were assigned to it.
 4. **Repeat steps 2 and 3** until the cluster assignments do not change or a user-defined tolerance of change or maximum number of iterations is reached.

Grouping objects by similarity using k-means

- How do we measure **similarity between objects**?
 - We can define **similarity** as **the opposite of distance**, and a commonly used distance for clustering examples with continuous features is the squared **Euclidean distance** between two points, \mathbf{x} and \mathbf{y} , in m -dimensional space:

$$d(\mathbf{x}, \mathbf{y})^2 = \sum_{j=1}^m (x_j - y_j)^2 = \|\mathbf{x} - \mathbf{y}\|_2^2$$

Note: In this equation, the index j refers to the j th dimension (feature column) of the example inputs, \mathbf{x} and \mathbf{y} .

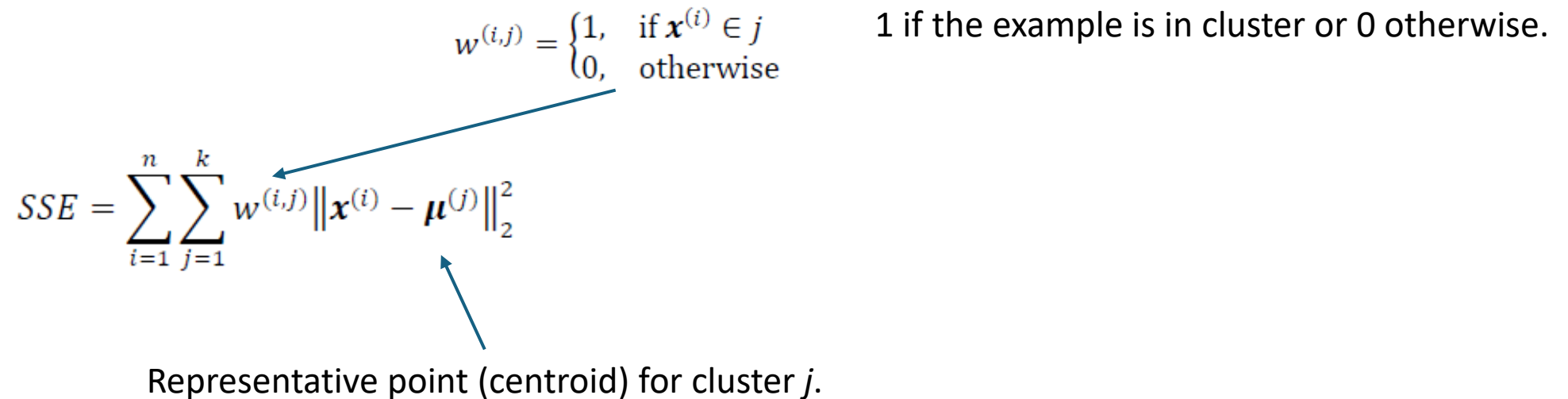
Grouping objects by similarity using k-means

- Based on this Euclidean distance metric, we can describe the k-means algorithm as a simple optimization problem.
 - An iterative approach for minimizing the **within-cluster sum of squared errors (SSE)** or also called **cluster inertia**:

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

$w^{(i,j)} = \begin{cases} 1, & \text{if } \mathbf{x}^{(i)} \in j \\ 0, & \text{otherwise} \end{cases}$ 1 if the example is in cluster or 0 otherwise.

Representative point (centroid) for cluster j .



Grouping objects by similarity using 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)
```

We set the number of desired clusters to 3 (having to specify the number of clusters *a priori* is one of the limitations of k-means).

We set `n_init=10` to run the k-means clustering algorithms 10 times independently, with different random centroids to choose the final model as the one with the lowest SSE.

Via the `max_iter` parameter, we specify the maximum number of iterations for each single run (here, 300). Note that the k-means implementation in scikit-learn stops early if it converges before the maximum number of iterations is reached.

However, it is possible that k-means does not reach convergence for a particular run, which can be problematic (computationally expensive) if we choose relatively large values for `max_iter`. One way to deal with convergence problems is to choose larger values for `tol`, which is a parameter that controls the tolerance with regard to the changes in the within-cluster SSE to declare convergence. In the preceding code, we chose a tolerance of `1e-04` (=0.0001).

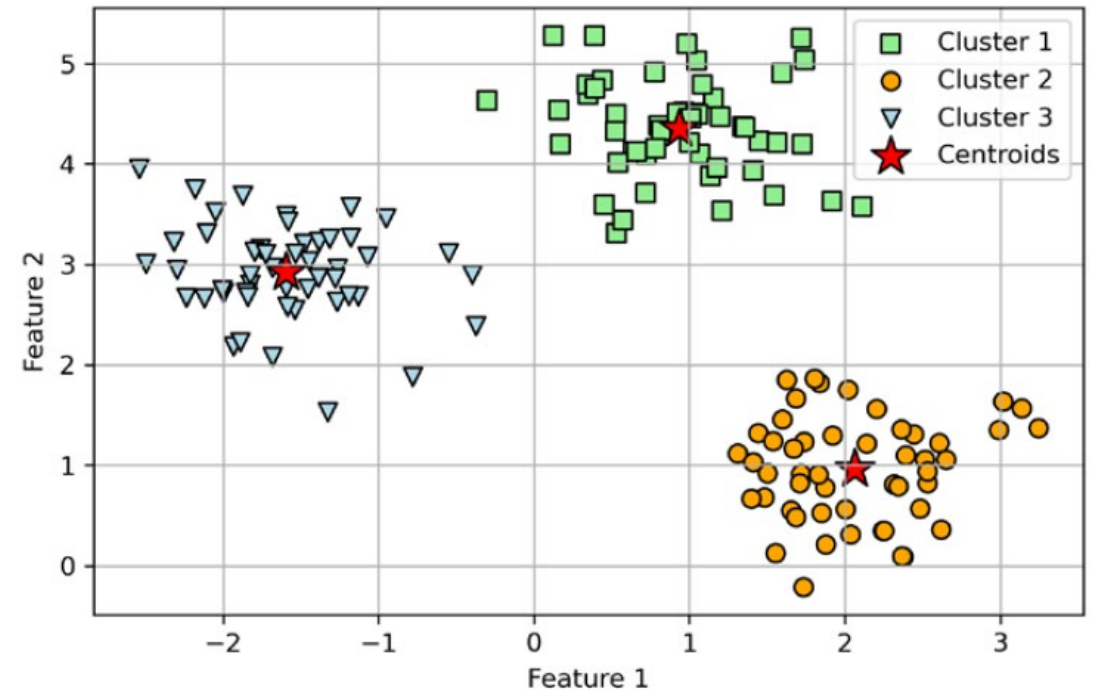
Grouping objects by similarity using k-means

- **Feature scaling:**

- When we are applying k-means to real-world data using a Euclidean distance metric, we want to make sure that the **features are measured on the same scale** and apply **z-score standardization or min-max scaling** if necessary.

Grouping objects by similarity using k-means

```
>>> 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.xlabel('Feature 1')  
>>> plt.ylabel('Feature 2')  
>>> plt.legend(scatterpoints=1)  
>>> plt.grid()  
>>> plt.tight_layout()  
>>> plt.show()
```



The k-means clusters and their centroids

Grouping objects by similarity using k-means

- We have the drawback of having to specify the number of clusters, k , a priori. The **number of clusters to choose** may **not** always be so **obvious in real-world applications**, especially if we are working with a **higher-dimensional dataset** that cannot be visualized.
- Also, k-means algorithm uses a **random seed to place the initial centroids**, which **can** sometimes result in **bad clusters or slow convergence** if the initial centroids are chosen poorly.
 - One way to address this issue is to **run the k-means algorithm multiple times** on a dataset and **choose the best-performing model** in terms of the SSE.
 - Another strategy is to place the initial centroids far away from each other via the **k-means++** algorithm, which leads to better and more consistent results than the classic k-means.

Grouping objects by similarity using k-means

- The **initialization in k-means++** can be summarized as follows:

1. Initialize an empty set, M , to store the k centroids being selected.
2. Randomly choose the first centroid, $\mu^{(j)}$, from the input examples and assign it to M .
3. For each example, $x^{(i)}$, that is not in M , find the minimum squared distance, $d(x^{(i)}, M)^2$, to any of the centroids in M .
4. To randomly select the next centroid, $\mu^{(p)}$, use a weighted probability distribution equal to $\frac{d(\mu^{(p)}, M)^2}{\sum_i d(x^{(i)}, M)^2}$. For instance, we collect all points in an array and choose a weighted random sampling, such that the larger the squared distance, the more likely a point gets chosen as the centroid.
5. Repeat *steps 3 and 4* until k centroids are chosen.
6. Proceed with the classic k-means algorithm.

Grouping objects by similarity using k-means

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



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

Grouping objects by similarity using k-means

- **Hard vs. soft clustering**
 - **Hard clustering** describes a family of algorithms where **each example in a dataset is assigned to exactly one cluster**, as in the k-means and k-means++ algorithms that we discussed earlier in this chapter.
 - In contrast, algorithms for **soft clustering** (sometimes also called fuzzy clustering) **assign an example to one or more clusters**. A popular example of soft clustering is the fuzzy C-means (FCM) algorithm (also called soft k-means or fuzzy k-means).

Grouping objects by similarity using k-means

- One of the **main challenges** in **unsupervised learning** is that we **do not know the definitive answer**.
 - We don't have the ground-truth class labels in our dataset that allow us to evaluate the performance of the model.
- Thus, to quantify the quality of clustering, we **need to use intrinsic metrics**, such as the **within-cluster SSE (inertia)**, to **compare the performance** of different k-means clustering models.
 - In scikit-learn is already accessible via the `inertia_` attribute after fitting a KMeans model.

```
>>> print(f'Distortion: {km.inertia_:.2f}')  
Distortion: 72.48
```

Grouping objects by similarity using k-means

- **Based on the within-cluster SSE**, we can use a graphical tool, the so-called **elbow method**, to **estimate the optimal number of clusters**, k , for a given task.
- We can say that **if k increases**, the **inertia will decrease**. This is because the **examples will be closer to the centroids they are assigned to**.
- The idea behind the elbow method is to **identify the value of k where the distortion begins to increase** most rapidly.

Grouping objects by similarity using k-means

```
>>> 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.show()
```

Grouping objects by similarity using k-means

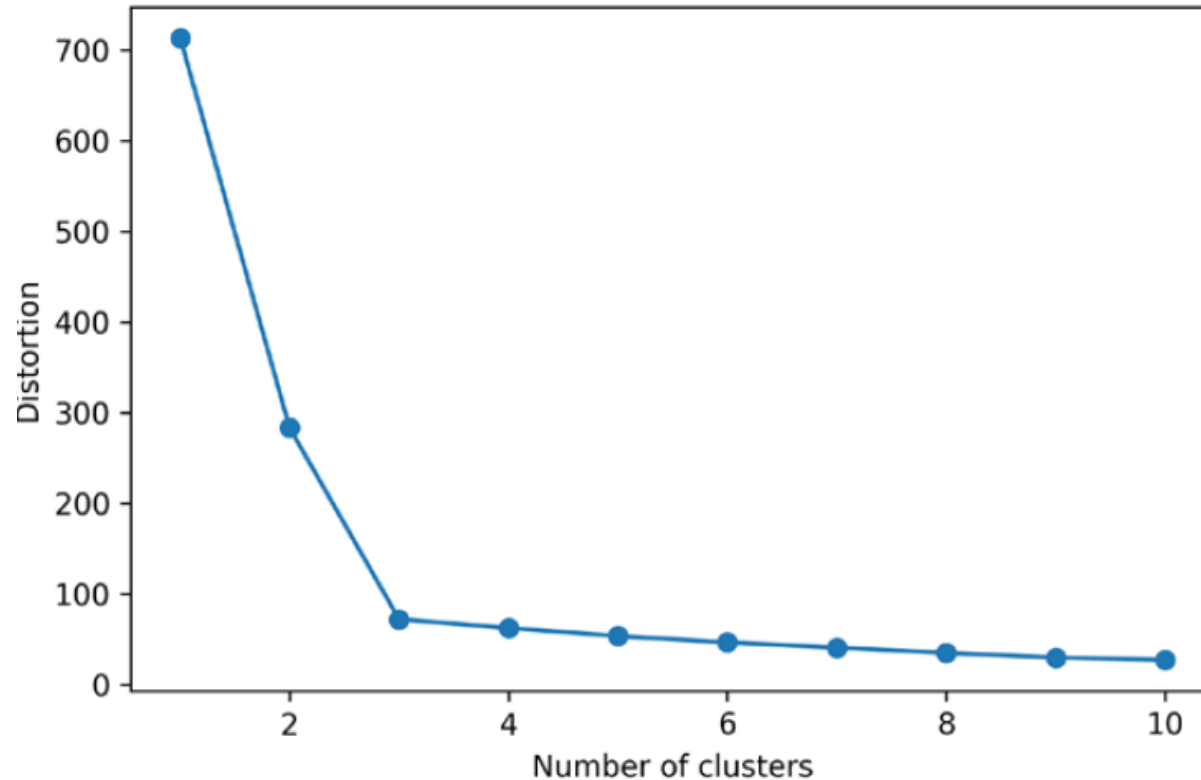


Figure 10.3: Finding the optimal number of clusters using the elbow method

Grouping objects by similarity using k-means

- Another intrinsic **metric to evaluate the quality of a clustering** is **silhouette analysis**, which can also be applied to clustering algorithms other than k-means that we will discuss later in this chapter.
 - Silhouette analysis can be used as a graphical tool to plot a measure of how tightly grouped the examples in the clusters are (**measure how similar an object is to its own cluster compared to other clusters**).

Grouping objects by similarity using k-means

- To calculate the **silhouette coefficient of a single example** in our dataset, we can apply the following three steps:
 1. Calculate the **cluster cohesion**, $a^{(i)}$, as the average distance between an example, $x^{(i)}$, and all other points in the same cluster.
 2. Calculate the **cluster separation**, $b^{(i)}$, from the next closest cluster as the average distance between the example, $x^{(i)}$, and all examples in the nearest cluster.
 3. Calculate the silhouette, $s^{(i)}$, as the difference between cluster cohesion and separation divided by the greater of the two, as shown here:

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

Grouping objects by similarity using k-means

- Score close to 1:
 - A value close to 1 indicates that the data point is well grouped within its own cluster and is clearly separated from other clusters.
 - This suggests that the data point is similar to other points in its cluster and different from points in neighboring clusters. It is a sign that the cluster is compact and well defined.
- Score of 0:
 - A value of 0 indicates that the data point is on the boundary between two clusters.
 - This means that the data point is equidistant between its own cluster and the nearest neighboring cluster. In this case, the point could belong to either cluster, suggesting that the clusters are not well separated.
- Score close to -1:
 - A value close to -1 indicates that the data point is probably misassigned to its current cluster.
 - This suggests that the data point is more similar to a neighboring cluster than to the cluster to which it has been assigned. It is a sign that the cluster is fuzzy or that the data point could be in the wrong cluster.

Grouping objects by similarity using k-means

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

Grouping objects by similarity using k-means

```
>>> 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)

>>> import numpy as np
>>> from matplotlib import cm
>>> from sklearn.metrics import silhouette_samples
>>> 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.show()
```

Grouping objects by similarity using k-means

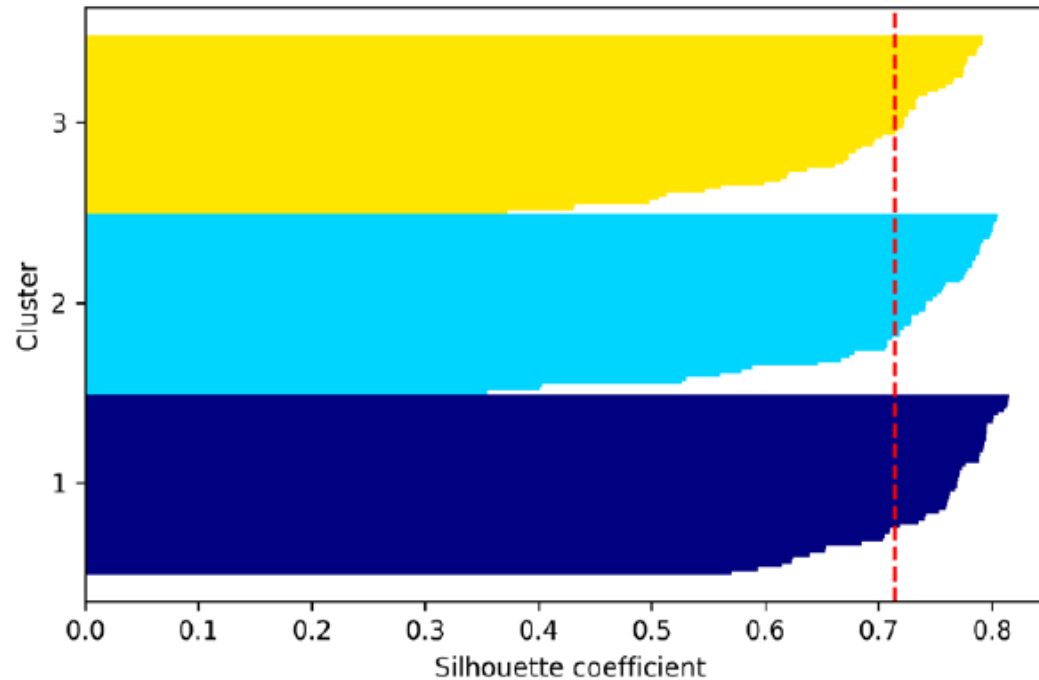


Figure 10.4: A silhouette plot for a good example of clustering

The silhouette coefficients are not close to 0 and are approximately equally far away from the average silhouette score, which is, in this case, an indicator of *good* clustering. Furthermore, to summarize the goodness of our clustering, we added the average silhouette coefficient to the plot (dotted line).

Grouping objects by similarity using k-means

Example for a *bad* clustering.

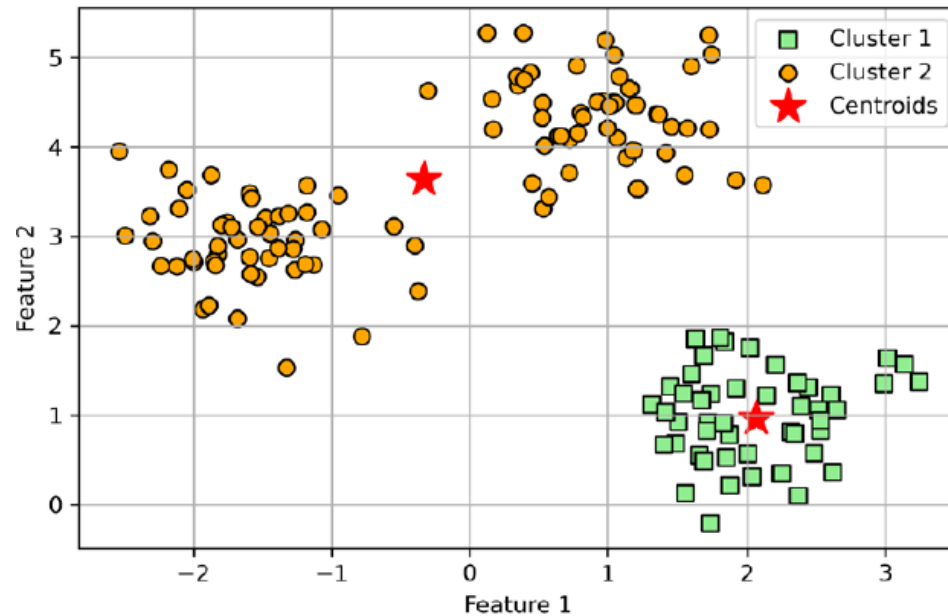


Figure 10.5: A suboptimal example of clustering

```
>>> 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.xlabel('Feature 1')  
>>> plt.ylabel('Feature 2')  
>>> plt.legend()  
>>> plt.grid()  
>>> plt.tight_layout()  
>>> plt.show()
```

Grouping objects by similarity using k-means

Example for a *bad* clustering.

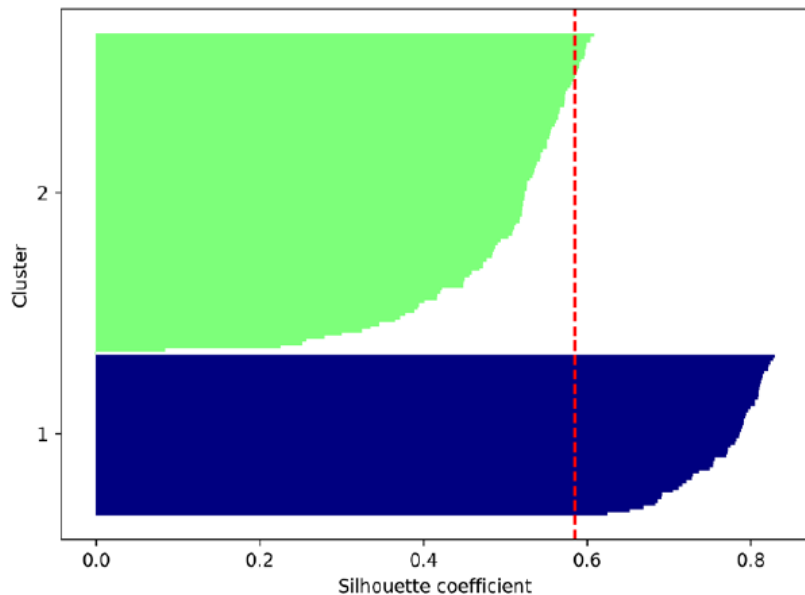


Figure 10.6: A silhouette plot for a suboptimal example of clustering

The silhouettes have visibly different lengths and widths, which is evidence of a relatively *bad* or at least *suboptimal* clustering.

```
>>> 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.show()
```

Grouping objects by similarity using k-means

Ejercicio

En este ejercicio, realizarás un análisis no supervisado utilizando el método k-means sobre el conjunto de datos proporcionado en el archivo dataset1.csv.

Para ello, emplearás la librería scikit-learn con las siguientes instrucciones:

- Normalización de datos: Antes de aplicar k-means, asegúrate de normalizar los datos para mejorar la precisión del análisis.
- Estimación del número óptimo de clusters: Utiliza el método del codo (elbow method) para determinar el número óptimo de clusters.
- Cálculo de clusters con k-means: Una vez determinado el número óptimo de clusters, aplica el algoritmo k-means para calcular los clusters.
- Representación gráfica con PCA: Realiza un PCA para reducir la dimensionalidad de los datos y representar los clusters en un gráfico bidimensional.
- Cálculo de coeficientes de silueta: Calcula los coeficientes de silueta para cada grupo, así como el coeficiente de silueta global del conjunto de datos. Representa gráficamente estos coeficientes para evaluar la calidad de los clusters formados.

Organizing clusters as a hierarchical tree

- We will look at an alternative approach to prototype-based clustering: **hierarchical clustering**.
 - It organizes the data in a **hierarchical structure**, similar to a tree, where each level of the hierarchy represents a different grouping of the data.
 - This structure allows to observe how the data are grouped at different levels of similarity.
- One advantage of the hierarchical clustering algorithm is that it allows us to plot **dendrograms** (**visualizations** of a binary hierarchical clustering), which can help with the interpretation of the results.
- Another advantage of this hierarchical approach is that **we do not need to specify the number of clusters** upfront.

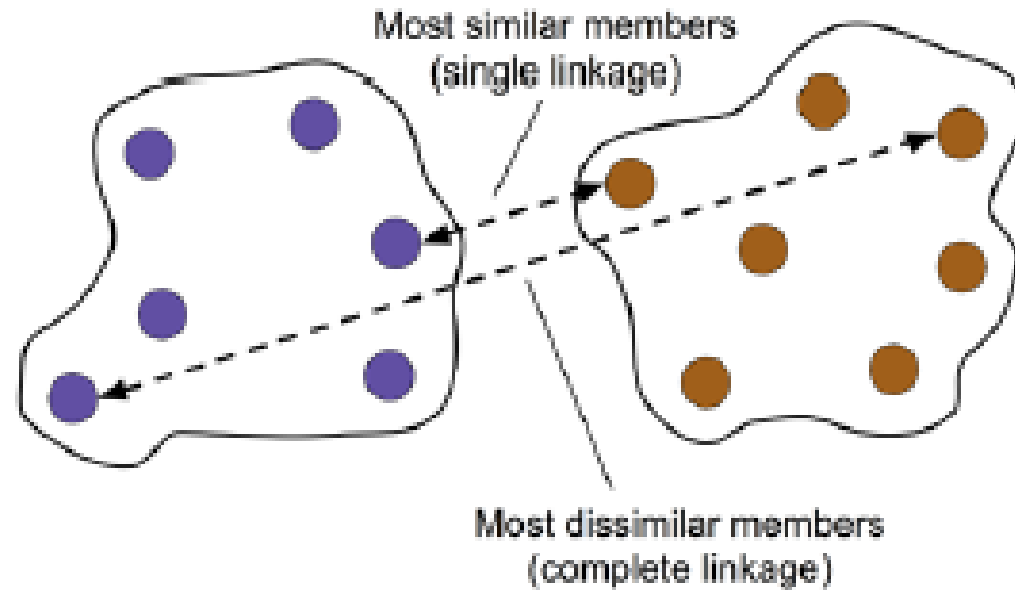
Organizing clusters as a hierarchical tree

- The **two main approaches** to hierarchical clustering are **agglomerative** and **divisive** hierarchical clustering.
- In **divisive hierarchical clustering**, we **start with one cluster** that encompasses the complete dataset, and **we iteratively split the cluster into smaller clusters** until each cluster only contains one example.
- In **agglomerative hierarchical clustering**, we take the opposite approach. We **start with each example as an individual cluster** and **merge the closest pairs of clusters** until only one cluster remains.

Organizing clusters as a hierarchical tree

- The two standard algorithms for **agglomerative hierarchical clustering** are **single linkage** and **complete linkage**.
- Using **single linkage**, we compute the **distances between the more similar (closest) members for each pair of clusters** and **merge the two clusters** for which **the distance** between the closest members is the **smallest**.
- The **complete linkage** approach is similar to single linkage but, instead of comparing the closest members in each pair of clusters, we compare the **most dissimilar (farthest) members to perform the merge**.

Organizing clusters as a hierarchical tree



Organizing clusters as a hierarchical tree

- We will focus on **agglomerative clustering** using the **complete linkage approach**.
 - It is an **iterative procedure** that can be summarized by the following steps:
 1. Compute a pair-wise distance matrix of all examples.
 2. Represent each data point as a singleton cluster.
 3. Merge the two closest clusters based on the distance between the most dissimilar (distant) members.
 4. Update the cluster linkage matrix.
 5. Repeat steps 2-4 until one single cluster remains.

Organizing clusters as a hierarchical tree

```
>>> 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

Organizing clusters as a hierarchical tree

Calculate the distance matrix of all examples -> Euclidean distance between each pair of input examples in our dataset.

```
>>> 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

Organizing clusters as a hierarchical tree

```
>>> from scipy.cluster.hierarchy import linkage
```

```
>>> 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=[f'cluster {(i + 1)}' for i in  
...                     range(row_clusters.shape[0])])
```

Organizing clusters as a hierarchical tree

The **linkage matrix** consists of several rows where each row represents one merge. The first and second columns denote the most dissimilar members in each cluster, and the third column reports the distance between those members. The last column returns the count of the members in each cluster.

	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

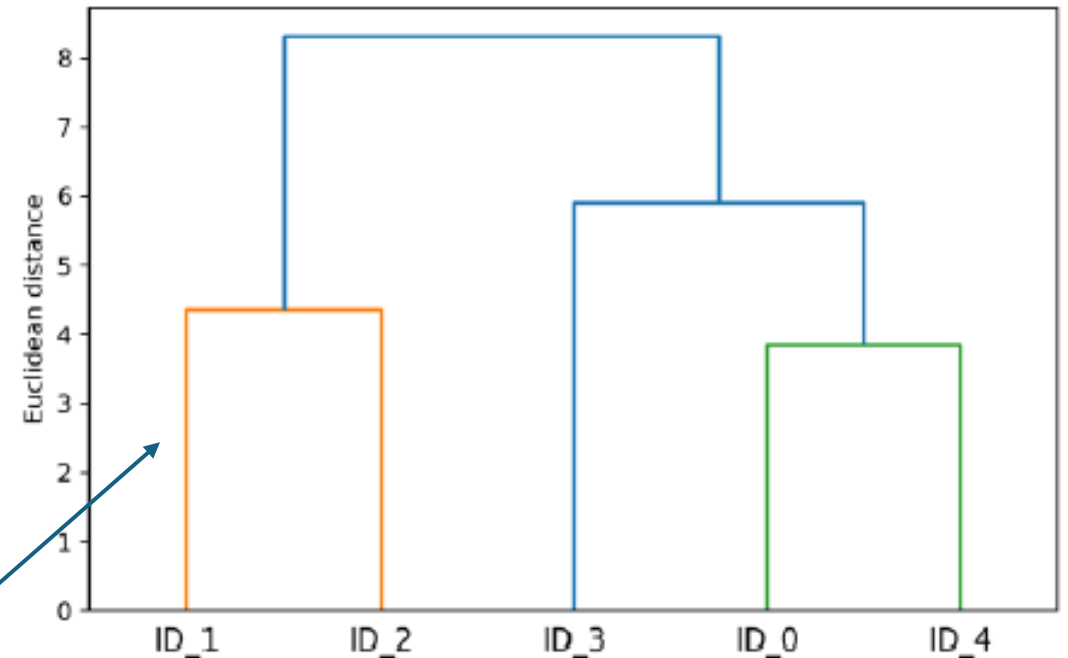
Proceso de Clustering

1. **Inicio:** Comienzas con 5 clusters, uno por cada punto (ID_0 a ID_4).
2. **Paso 1:**
 - **Cluster 1:** Se combinan los puntos ID_0 e ID_4, formando un nuevo cluster con una distancia de 3.835396.
3. **Paso 2:**
 - **Cluster 2:** Se combinan los puntos ID_1 e ID_2, formando otro cluster con una distancia de 4.347073.
4. **Paso 3:**
 - **Cluster 3:** El punto ID_3 se combina con el cluster formado por ID_0 e ID_4, con una distancia de 5.899885.
5. **Paso 4:**
 - **Cluster 4:** Finalmente, el cluster formado por ID_1 e ID_2 se combina con el cluster que incluye ID_0, ID_3, e ID_4, con una distancia de 8.316594.

Organizing clusters as a hierarchical tree

Now that we have computed the linkage matrix, we can **visualize the results** in the form of a **dendrogram**.

```
>>> 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.show()
```



La altura de las uniones indica la distancia o disimilitud entre los clústeres fusionados.

Organizing clusters as a hierarchical tree

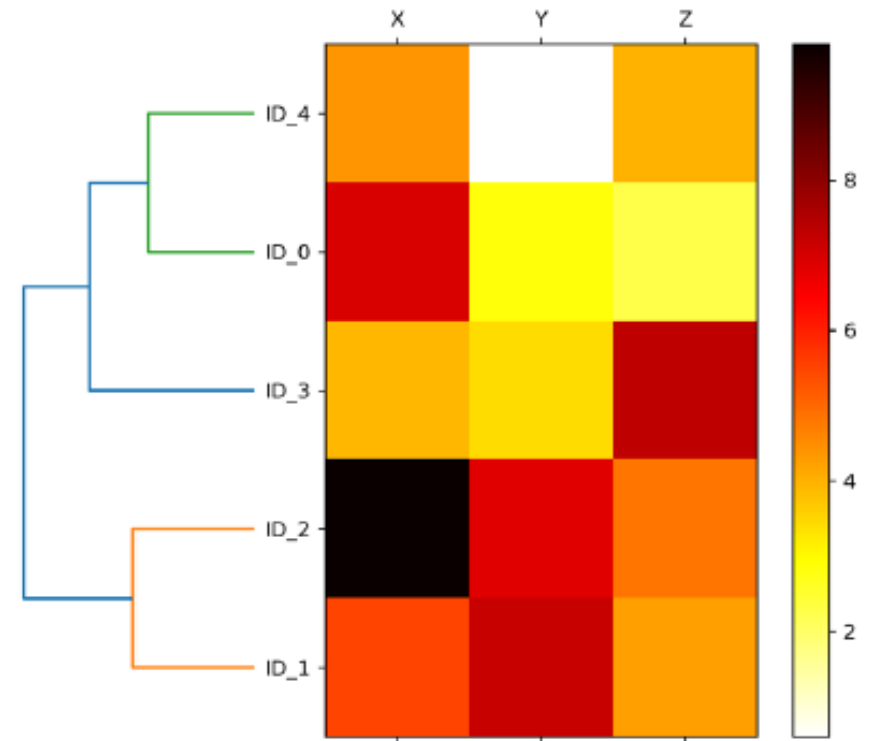
- In practical applications, hierarchical clustering **dendrograms are often used in combination with a heat map**, which allows us to represent the individual values in the data array or matrix containing our training examples with a color code.

Organizing clusters as a hierarchical tree

```
>>> fig = plt.figure(figsize=(8, 8), facecolor='white')
>>> axd = fig.add_axes([0.09, 0.1, 0.2, 0.6])
>>> row_dendr = dendrogram(row_clusters,
...                         orientation='left')
>>> # note: for matplotlib < v1.5.1, please use
>>> # orientation='right'
```

```
>>> df_rowclust = df.iloc[row_dendr['leaves'][:, -1]]
```

```
>>> axd.set_xticks([])
>>> axd.set_yticks([])
>>> for i in axd.spines.values():
...     i.set_visible(False)
>>> fig.colorbar(cax)
>>> axm.set_xticklabels([''] + list(df_rowclust.columns))
>>> axm.set_yticklabels([''] + list(df_rowclust.index))
>>> plt.show()
```



Organizing clusters as a hierarchical tree

Applying agglomerative clustering via scikit-learn

```
>>> from sklearn.cluster import AgglomerativeClustering
>>> ac = AgglomerativeClustering(n_clusters=3,
...                               affinity='euclidean',
...                               linkage='complete')
>>> labels = ac.fit_predict(X)
>>> print(f'Cluster labels: {labels}')
Cluster labels: [1 0 0 2 1]
```

```
>>> ac = AgglomerativeClustering(n_clusters=2,
...                               affinity='euclidean',
...                               linkage='complete')
>>> labels = ac.fit_predict(X)
>>> print(f'Cluster labels: {labels}')
Cluster labels: [0 1 1 0 0]
```

Organizing clusters as a hierarchical tree

Ejercicio

En este ejercicio, realizarás un análisis no supervisado utilizando agglomerative clustering con complete linkage sobre el conjunto de datos proporcionado en el archivo dataset1.csv.

Para ello, emplearás las siguientes instrucciones:

- Normalización de datos: Antes de aplicar el algoritmo, asegúrate de normalizar los datos para mejorar la precisión del análisis.
- Cálculo de clusters con scipy o con scikit-learn.
- Representación gráfica de dendrograma y mapa de calor.
- Representación gráfica con PCA: Realiza un PCA para reducir la dimensionalidad de los datos y representar los clusters en un gráfico bidimensional.

Locating regions of high density via DBSCAN

- We are going to include one more approach to clustering: density-based spatial clustering of applications with noise (**DBSCAN**).
- As its name implies, density-based clustering **assigns cluster labels based on dense regions** of points.
- In DBSCAN, the **notion of density** is defined as **the number of points within a specified radius, ϵ** .

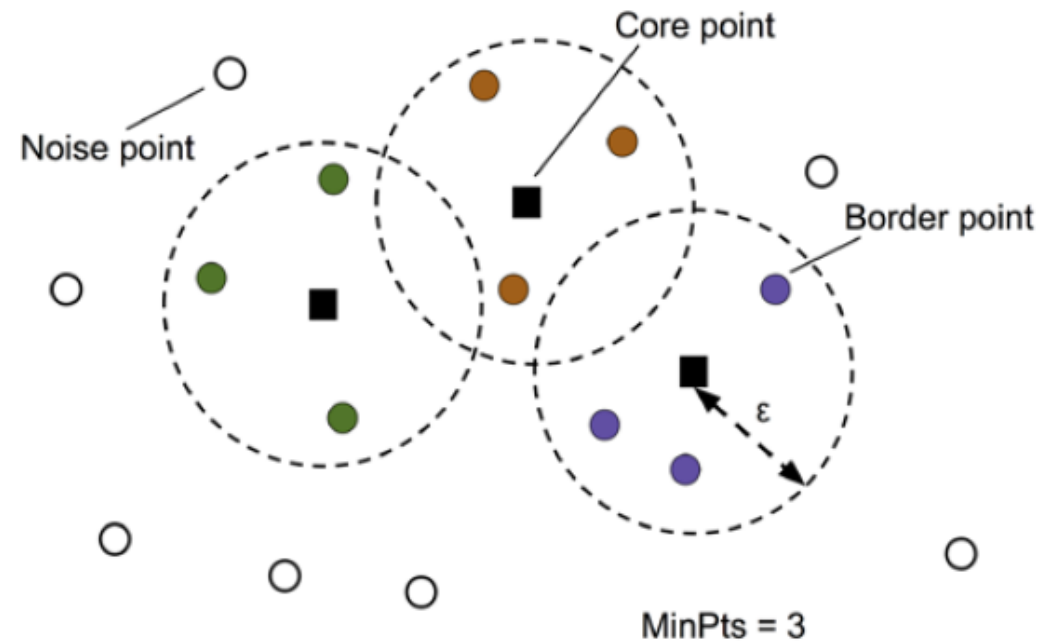
Locating regions of high density via DBSCAN

- According to the DBSCAN algorithm, a **special label** is assigned to each example (data point) using the following criteria:
 - A point is considered a **core point** if **at least a specified number (MinPts) of neighboring points** fall within the specified radius, ϵ .
 - A **border point** is a point that **has fewer neighbors than MinPts within ϵ** but lies within the ϵ radius of a core point.
 - All other points that are neither core nor border points are considered **noise points**.

Locating regions of high density via DBSCAN

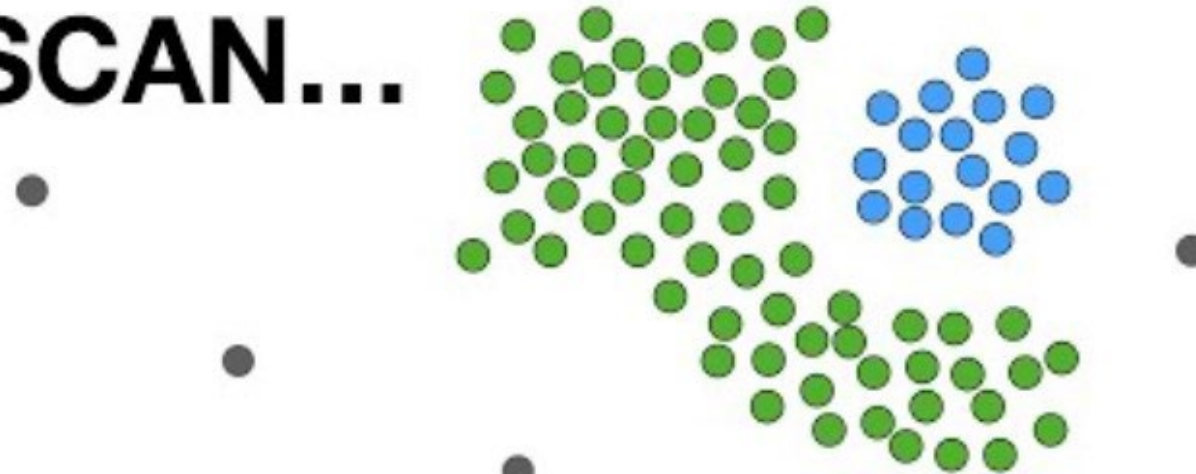
- After labeling the points as core, border, or noise, the DBSCAN algorithm can be summarized in two simple steps:
 1. Form a separate cluster for each core point or connected group of core points (core points are connected if they are no farther away than ϵ .)
 2. Assign each border point to the cluster of its corresponding core point.

Locating regions of high density via DBSCAN



Locating regions of high density via DBSCAN

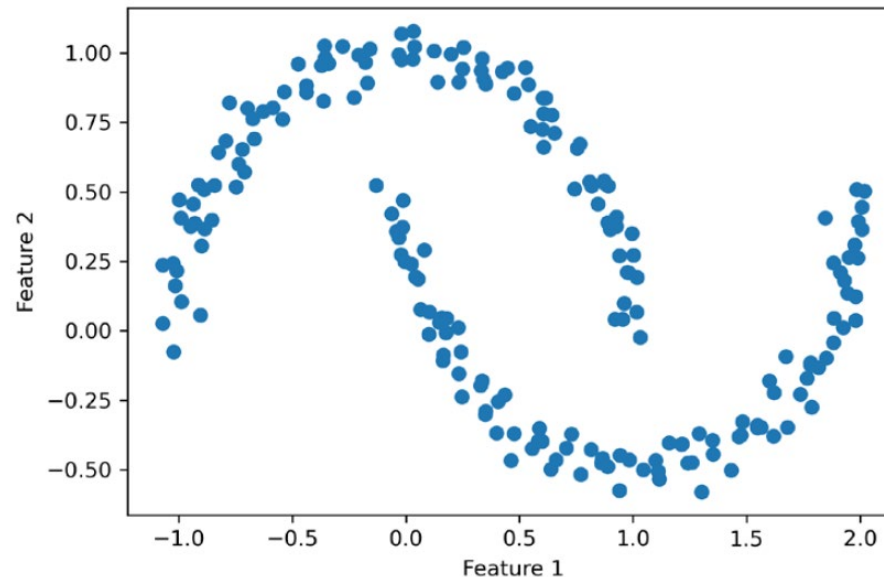
**Clustering with
DBSCAN...**



...Clearly Explained!!!

Locating regions of high density via DBSCAN

```
>>> 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.xlabel('Feature 1')
>>> plt.ylabel('Feature 2')
>>> plt.tight_layout()
>>> plt.show()
```



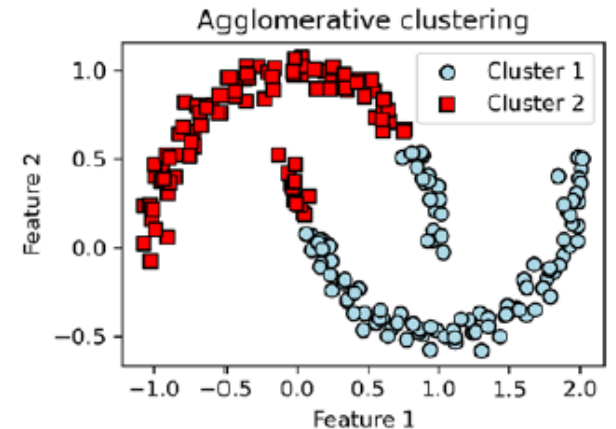
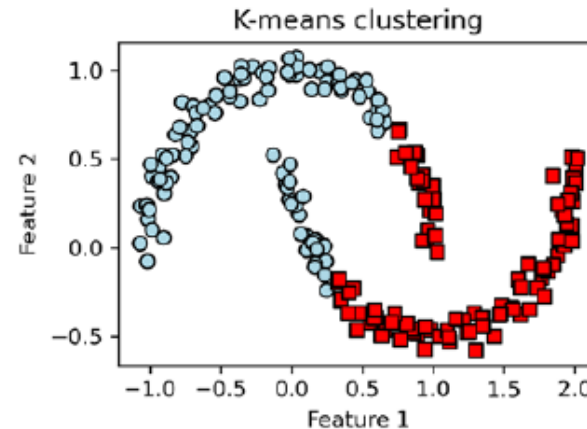
Locating regions of high density via DBSCAN

```
>>> 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],
...             c='lightblue',
...             edgecolor='black',
...             marker='o',
...             s=40,
...             label='cluster 1')
```

Locating regions of high density via DBSCAN

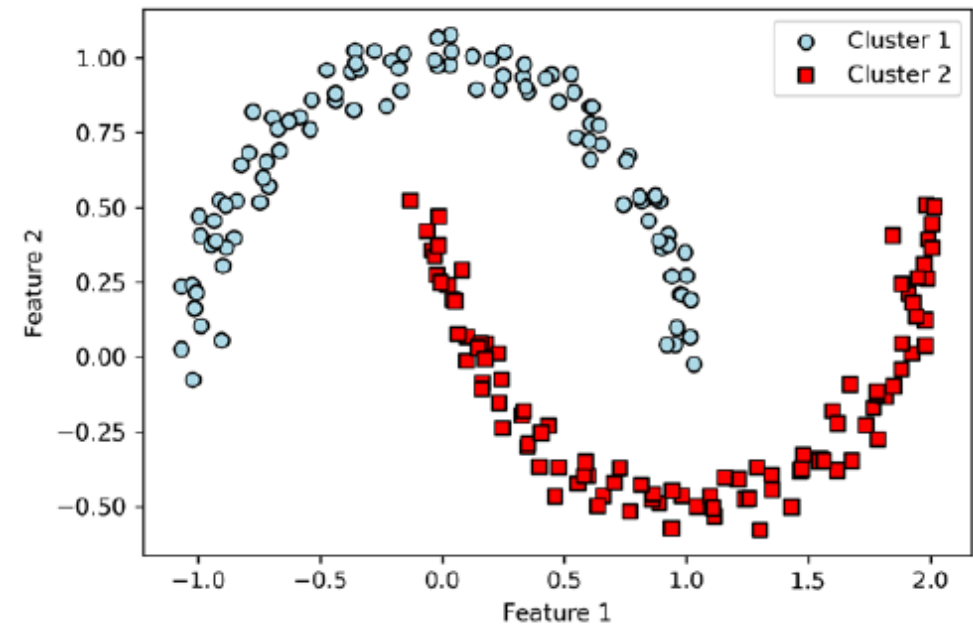
```
>>> ax1.scatter(X[y_km == 1, 0],
...             X[y_km == 1, 1],
...             c='red',
...             edgecolor='black',
...             marker='s',
...             s=40,
...             label='cluster 2')
>>> ax1.set_title('K-means clustering')
>>> ax1.set_xlabel('Feature 1')
>>> ax1.set_ylabel('Feature 2')

>>> 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')
>>> ax2.set_xlabel('Feature 1')
>>> ax2.set_ylabel('Feature 2')
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```



Locating regions of high density via DBSCAN

```
>>> 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',
...             edgecolor='black',
...             marker='o',
...             s=40,
...             label='Cluster 1')
>>> plt.scatter(X[y_db == 1, 0],
...             X[y_db == 1, 1],
...             c='red',
...             edgecolor='black',
...             marker='s',
...             s=40,
...             label='Cluster 2')
>>> plt.xlabel('Feature 1')
>>> plt.ylabel('Feature 2')
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```



Locating regions of high density via DBSCAN

Ejercicio

En este ejercicio, realizarás un análisis no supervisado utilizando DBSCAN sobre el conjunto de datos proporcionado en el archivo dataset1.csv.

Para ello, emplearás las siguientes instrucciones:

- Normalización de datos: Antes de aplicar el algoritmo, asegúrate de normalizar los datos para mejorar la precisión del análisis.
- Cálculo de clusters con scikit-learn.
- Representación gráfica con PCA: Realiza un PCA para reducir la dimensionalidad de los datos y representar los clusters en un gráfico bidimensional.