UNIVERSITY of York
Europe Campus
CITY College

Unsupervised Learning: Clustering

K-means, Density-based

Dr Ioanna Stamatopoulou

All material in these lecture notes is based on our textbook:
Aurélien Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, 3rd ed., O'Reilly, 2022

Clustering

- Clustering is one of the most common tasks of unsupervised learning, where data is unlabelled
- Most available data is unlabelled! The process of labelling the data is an option (e.g. by using human experts) but it is:
  - Time-consuming
  - Costly
  - Error-prone

  and needs to be repeated every time the dataset changes
- Like in classification, instances need to be assigned to clusters but these are not known a priori

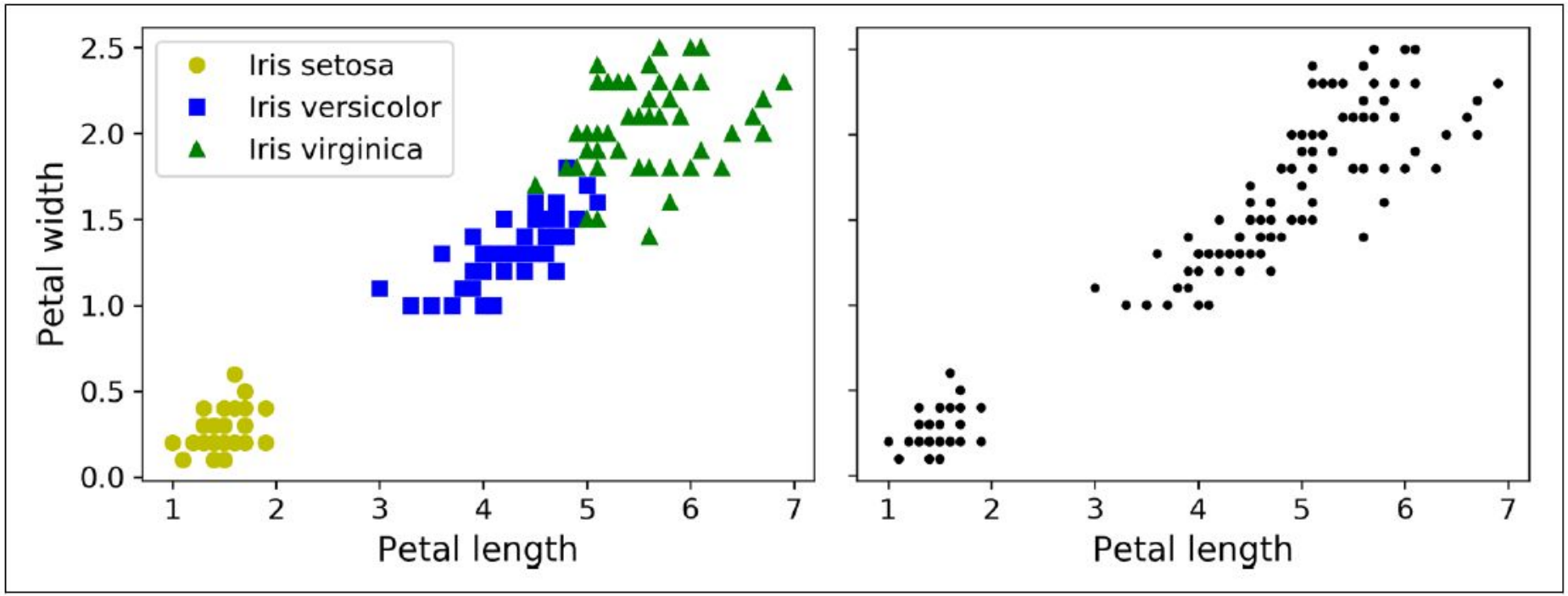


Figure 9-1. Classification (left) versus clustering (right)

Using only the two features, petal width and length, it is not clear that there are three clusters

Classification vs Clustering on the iris dataset

k-means

k-means

on a random 5-blob dataset

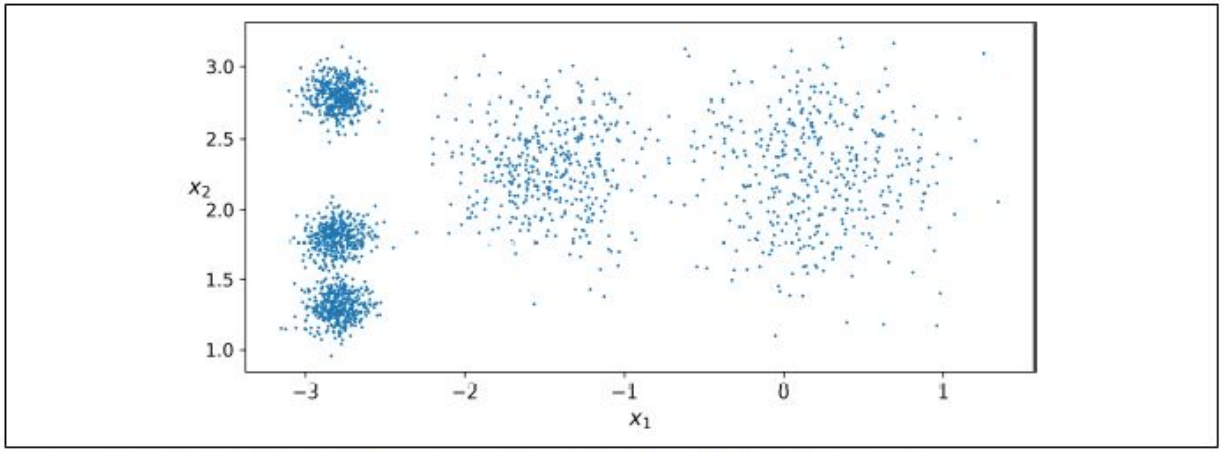


Figure 9-2. An unlabeled dataset composed of five blobs of instances

```
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

# extra code – the exact arguments of make_blobs() are not important
blob_centers = np.array([[ 0.2,  2.3], [-1.5 ,  2.3], [-2.8,  1.8],
                         [-2.8,  2.8], [-2.8,  1.3]])
blob_std = np.array([0.4, 0.3, 0.1, 0.1, 0.1])
X, y = make_blobs(n_samples=2000, centers=blob_centers, cluster_std=blob_std,
                  random_state=7)

k = 5
kmeans = KMeans(n_clusters=k, n_init=10, random_state=42)
y_pred = kmeans.fit_predict(X)
```

k-means

```
kmeans.cluster_centers_
```

```
array([[-2.80389616,  1.80117999],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80037642,  1.30082566]])
```

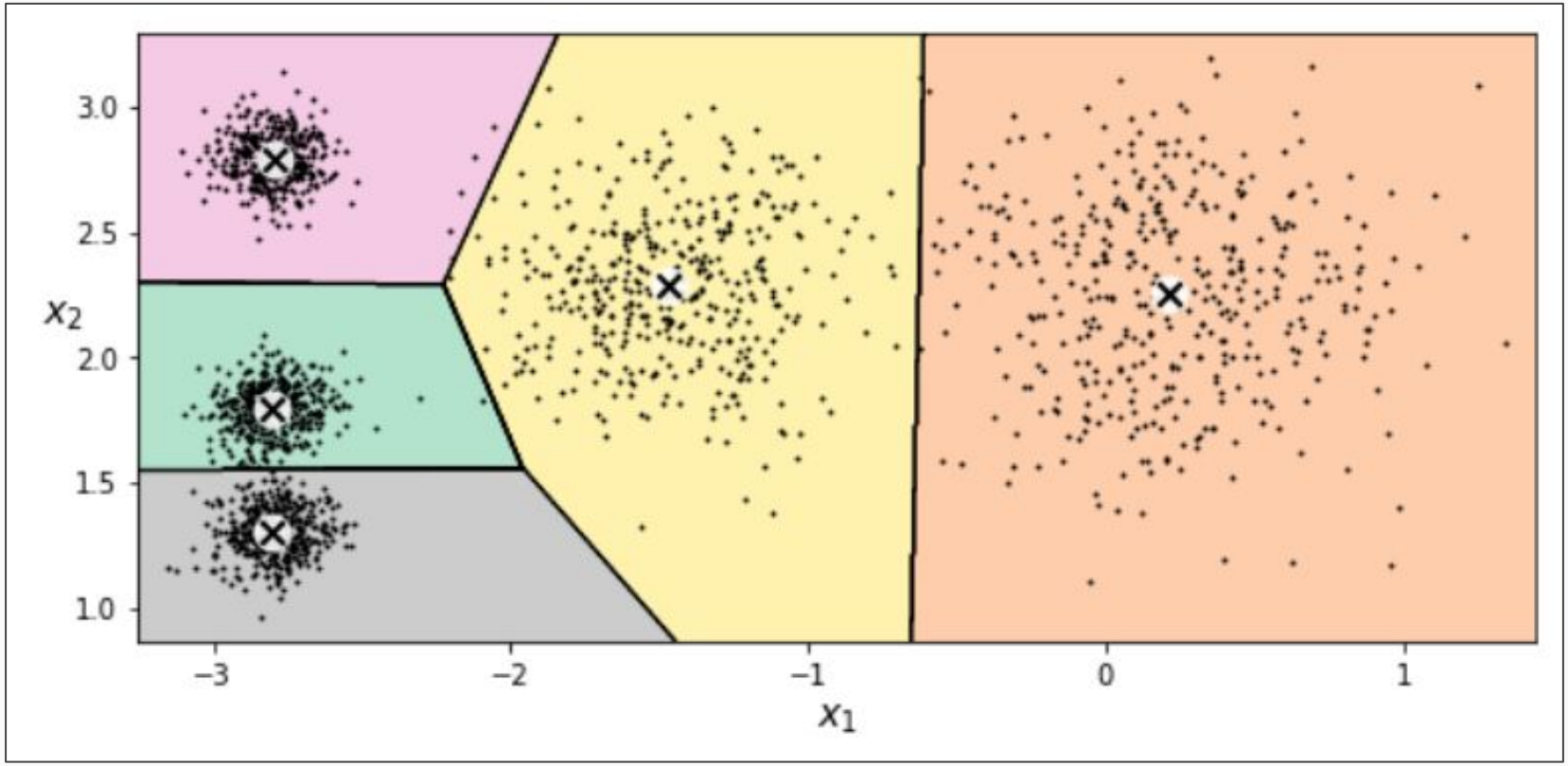
- Specify the number of clusters ***k***
  - Not always easy to identify ***k***…
- The goal of ***k***-means is to find the centroids of the k clusters
- After the centroids have been identified, any new instance is predicted to belong to the cluster whose centroid is closest to the instance
  - The predicted label of the algorithm is the index of the cluster that it is assigned to

```
y_pred
```

```
array([4, 0, 1, ..., 2, 1, 0], dtype=int32)
```

k-means Decision Boundaries

Voronoi tessellation (diagram)



K-means

How are the centroids identified?

```
init="random",
```

k-means is one of the simplest and fastest clustering algorithms

- First initialise ***k*** centroids randomly: e.g., ***k*** distinct instances are chosen randomly from the dataset and the centroids are placed at their locations
- Repeat until convergence (i.e., until the centroids stop moving):
  - Assign each instance to the closest centroid
  - Update the centroids to be the mean of the instances that are assigned to them

The algorithm is guaranteed to converge in a finite number of steps

k-means

Limitation of Random Centroid initialisation

- Depending on the initial (random) centroid initialisation, ***k***-means may converge in a sub-optimal solution (local optimum)
- Two popular solutions:
  - **n** random initialisations
  - ***k***-means++

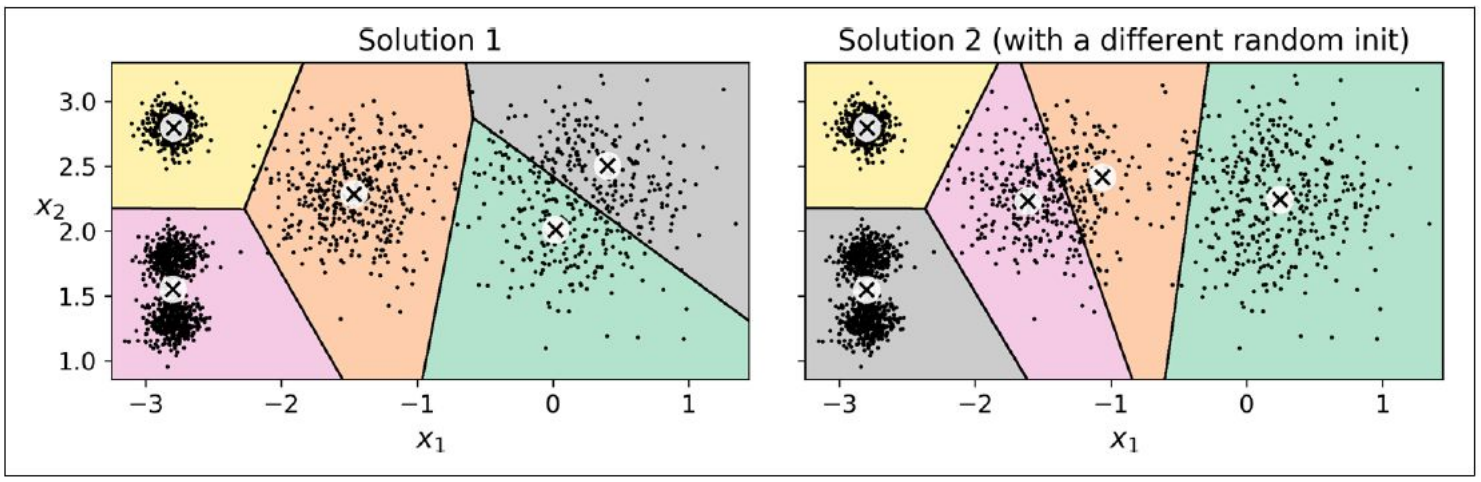


Figure 9-5. Suboptimal solutions due to unlucky centroid initializations

***k*-means centroid initialisation**

n random initialisations

- Run the algorithm **n** times with n different random initialisations and keep the best one
- Controlled by the **`n_init`** hyperparameter

```
kmeans_rnd_10_inits = KMeans(n_clusters=5, init="random", n_init=10,
                             random_state=2)
kmeans_rnd_10_inits.fit(X)
```

k-means centroid initialisation

Inertia

```
kmeans_rnd_10_inits.inertia_
```

```
211.5985372581684
```

- To know which of the **n** is best, you need a **performance measure**: **inertia**
  - the mean squared distance between each instance and its closest centroid
- The best is the one with the lowest inertia

k-means centroid initialisation

k-means++

- Smarter random initialization
- Selects centroids that are distant from one another
- Scikit's `KMeans` class uses this as the **default initialisation method** —just do not define the `init` parameter

```
kmeans_rnd_10_inits = KMeans(n_clusters=5, n_init=10,
                             random_state=2)
kmeans_rnd_10_inits.fit(X)
```

k-means

Finding the optimal number of clusters

- Finding ***k***, in most cases, is not easy
- Inertia is not a good measure when trying to choose ***k***
  - It becomes smaller as ***k*** increases
  - Because the distance of the instances from the closest centroids will get smaller as clusters increase
- A simple rule-of-thumb to adapt ***k***:
  - If single clusters contain different entities, increase ***k***
  - If entities spread across clusters, decrease ***k***
- But there are techniques that can be used to help you

k-means
Finding the optimal number of clusters

Inertia as a function of *k* — the elbow

- Computationally cheap but not necessarily accurate
- Not enough on its own

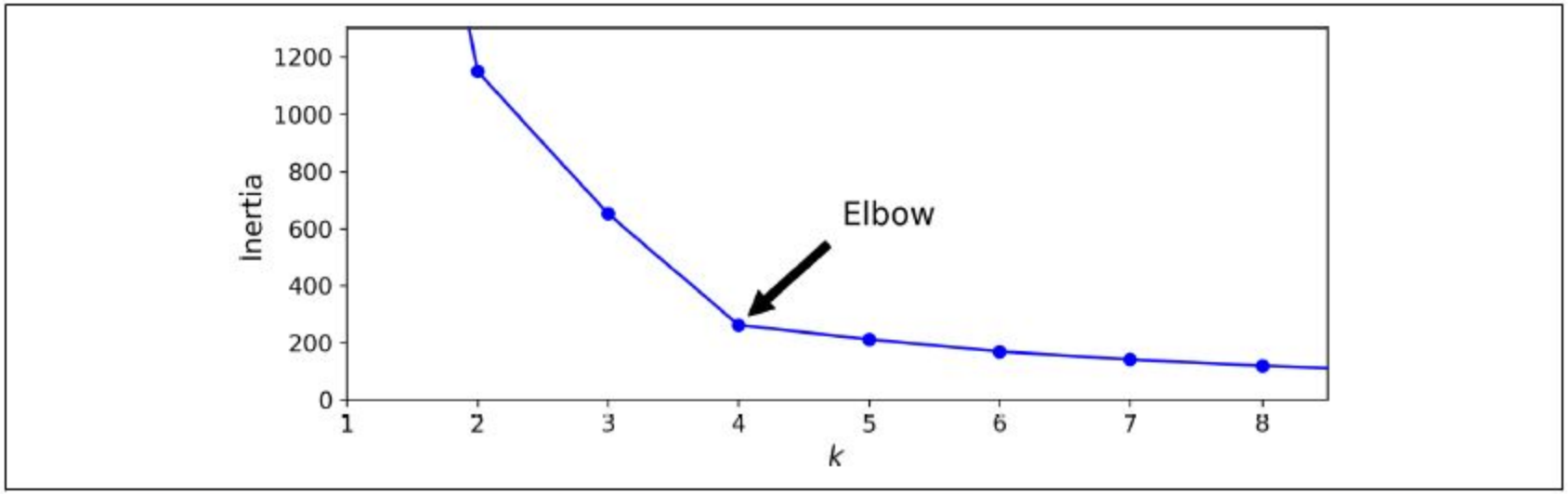


Figure 9-8. When plotting the inertia as a function of the number of clusters k, the curve often contains an inflexion point called the "elbow"

k-means
Finding the optimal number of clusters

The silhouette score

- More precise, but
- Computationally more expensive
- The **silhouette score** is the mean silhouette coefficient over all the instances
- An instance's **silhouette coefficient** is in the range [-1, 1]
  - **~1**: instance is well inside its cluster
  - **~0**: instance is close to cluster's boundary
  - **~-1**: instance may have been assigned to a wrong cluster

Comparison of silhouette scores for various ***k***

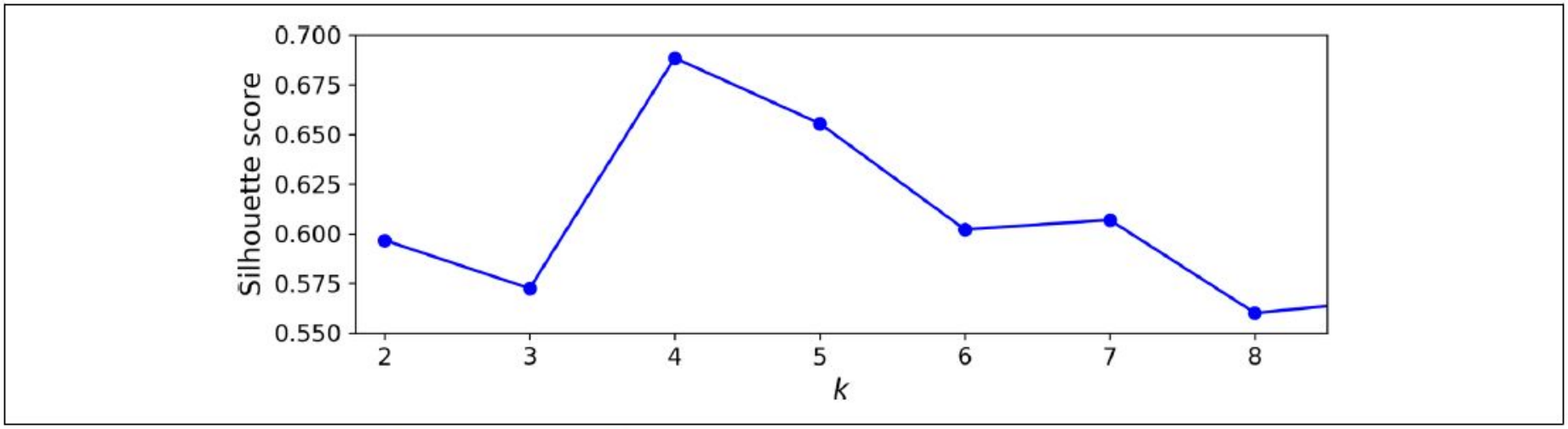


Figure 9-9. Selecting the number of clusters k using the silhouette score

- ***k***=4 still appears to be the most promising
- But now ***k***=5 looks more promising than demonstrated by the inertia
- We can further explore by plotting a silhouette diagram

Silhouette diagram for various ***k***

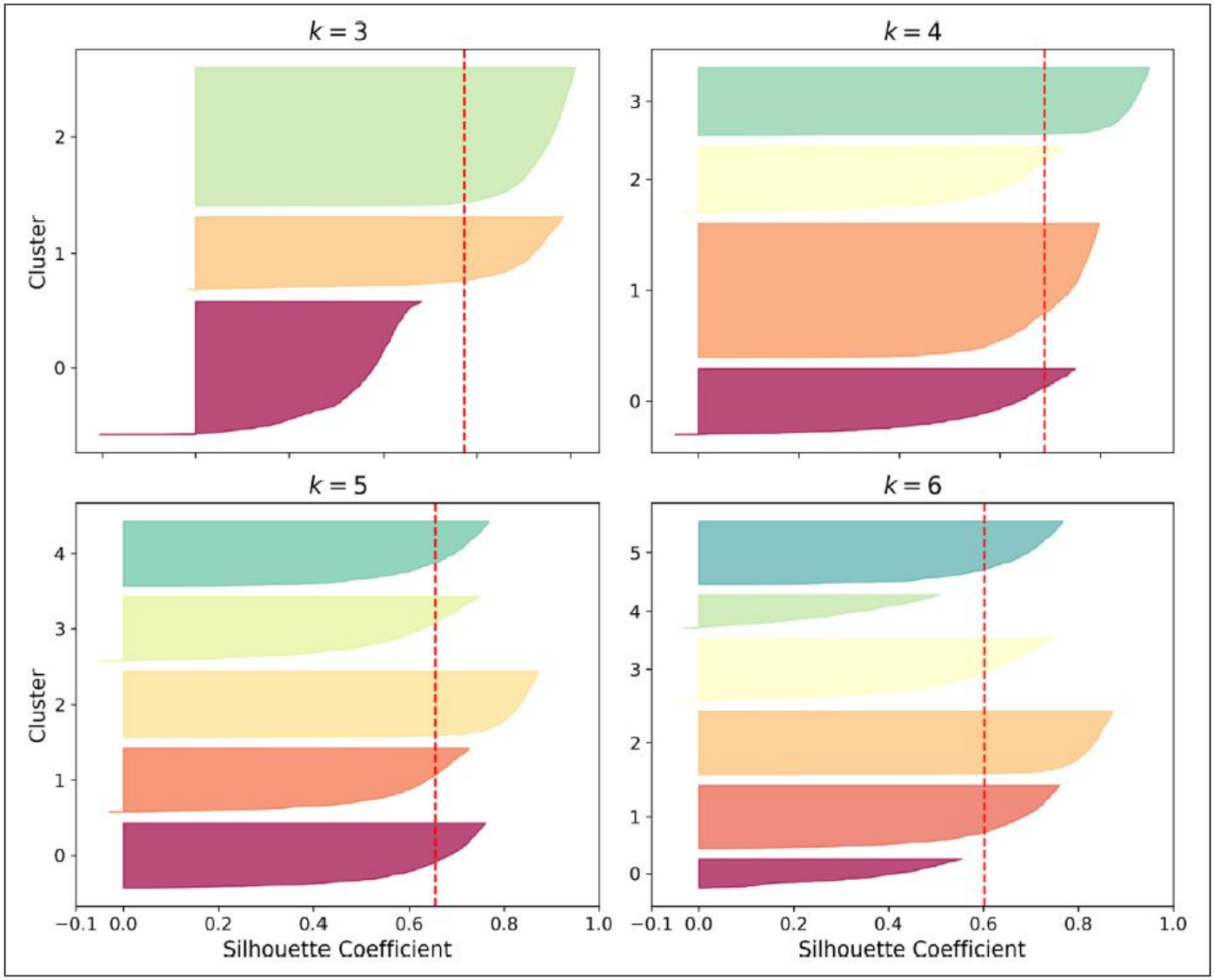


Figure 9-10. Analyzing the silhouette diagrams for various values of k

The vertical dashed lines represent the silhouette score for each number of clusters

k=3 and ***k***=6 are clearly bad because clusters fall short of the silhouette score

Analysing the diagram we can identify ***k***=5 as better than ***k***=4 because clusters have more similar sizes

k-means

Limitations

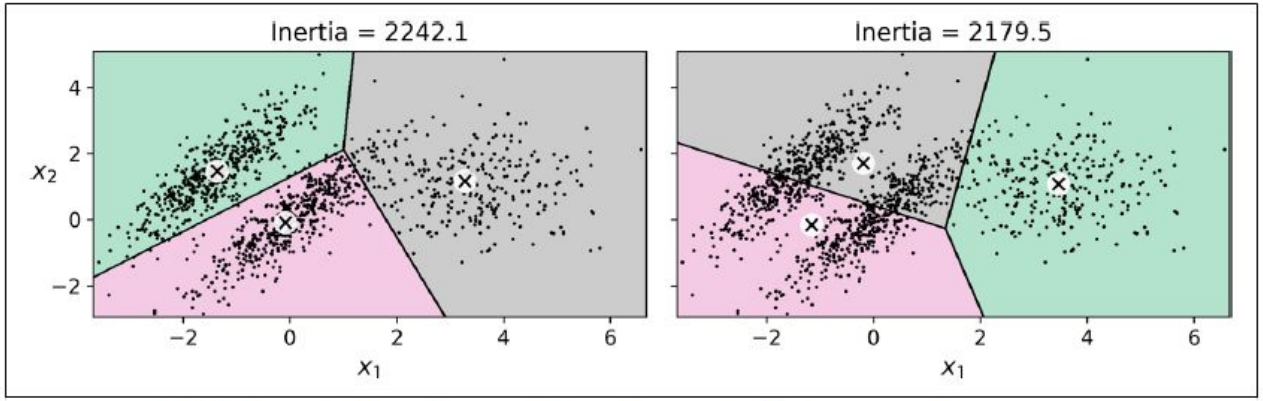


Figure 9-11. K-Means fails to cluster these ellipsoidal blobs properly

- Even after you go through the hassle of identifying the best ***k*** and running several times to avoid suboptimal solutions, ***k***-means will not do a good job when the clusters are:
  - of **varying sizes**
  - of **varying densities**
  - **non-spherical** in shape
- For **non-spherical clusters**:
  - Try scaling the features (still does not guarantee a spherical form)
  - Or prefer **Gaussian mixture models**

(outside the scope of this module)

DBSCAN

DBSCAN

ε and **`min_samples`** are the only two hyperparameters of DBSCAN

Density-Based Spatial Clustering of Applications with Noise

- DBSCAN defines clusters as continuous regions of high density
- For each instance, count how many instances are located within a small distance **ε** (epsilon) from it
  ⇒ **instance's ε-neighborhood**
- If an instance has at least **`min_samples`** instances in its ε-neighborhood (including itself)
  ⇒ **core instance** (i.e., located in dense regions)

DBSCAN

Density-Based Spatial Clustering of Applications with Noise (cont'd)

- All instances in the neighborhood of a core instance belong to the same cluster
- A core instance's neighborhood may include other core instances

  ⟹ a sequence of neighboring core instances forms a single cluster
- Any instance that is not a core instance and does not have one in its neighborhood is considered an anomaly

DBSCAN

on the moons dataset

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.05, random_state=42)
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
```

To get the identified labels of the (first 10) instances:
(-1 denotes the instance is identified as an anomaly)

```
dbscan.labels_[:10]
```

```
array([0, 0, 0, 0, 1, 0, 0, 0, 0, 1])
```

To get the core samples and their labels:

```
dbscan.components_
```

```
array([[-0.02137124,  0.40618608],
       [-0.84192557,  0.53058695],
       [ 0.58930337, -0.32137599],
       ...,
       [ 1.66258462, -0.3079193 ],
       [-0.94355873,  0.3278936 ],
       [ 0.79419406,  0.60777171]])
```

```
dbscan.core_sample_indices_[:10]
```

```
array([ 0,  4,  5,  6,  7,  8, 10, 11, 12, 13])
```

Regularisation of DBSCAN using ε

With ε=0.05 DBSCAN identifies 7 clusters (identified by the different colours).

The crossed instances are identified as anomalies.

With ε=0.2 DBSCAN identifies 2 clusters that make sense.

Not bad!!

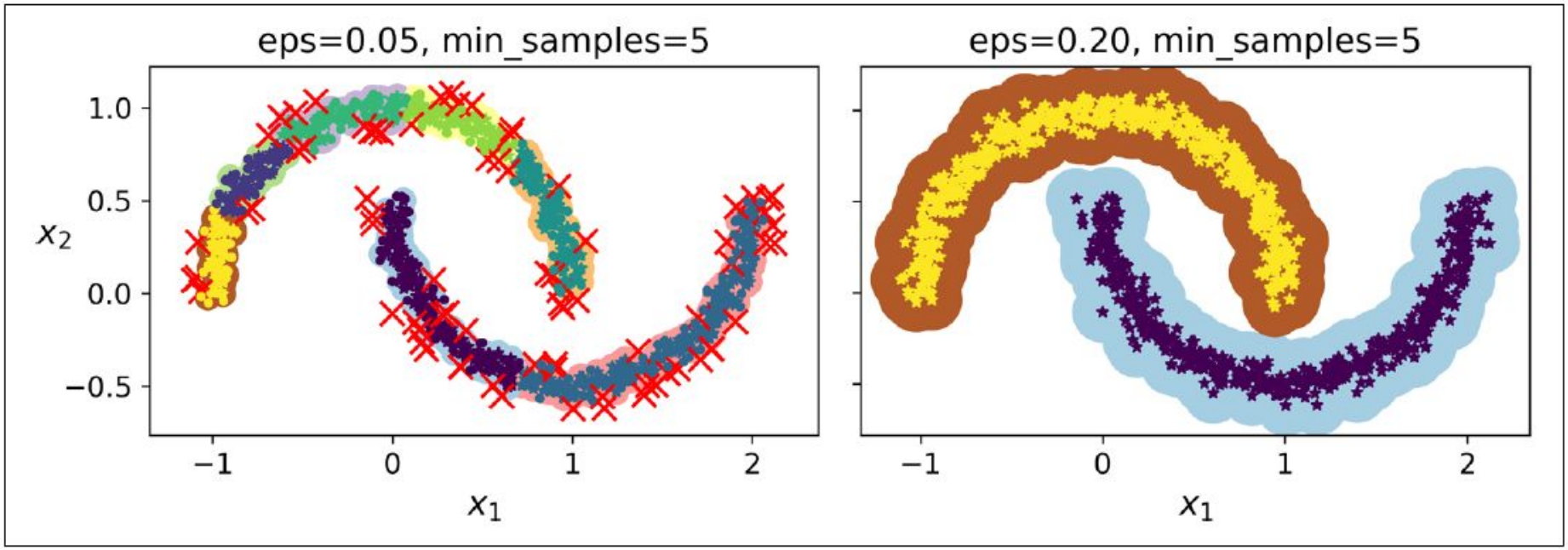


Figure 9-14. DBSCAN clustering using two different neighborhood radiuses

DBSCAN performs Clustering, not Classification!

- DBSCAN is a clustering, not a classification algorithm, which means:
- it does not predict the class of new instances!
  - Does not have a `predict()` method
- The rationale is that, after the clusters have been identified by DBSCAN, different classification algorithms can be used, depending on the task

Combining DBSCAN with KNN for Classification

- You may use a ***k*-nearest neighbor (KNN)** classifier to predict the class
- You can train the KNN classifier:
  - Only on the core instances that DBSCAN identifies
  - OR on the entire training set
  - OR on the entire training set excluding the anomalies
- KNN is supervised learning classifier which uses proximity to make classifications
- ***k*** in KNN defines the **number of neighbours** to be identified for any new instance

k-Nearest Neighbors

for Classification and Regression

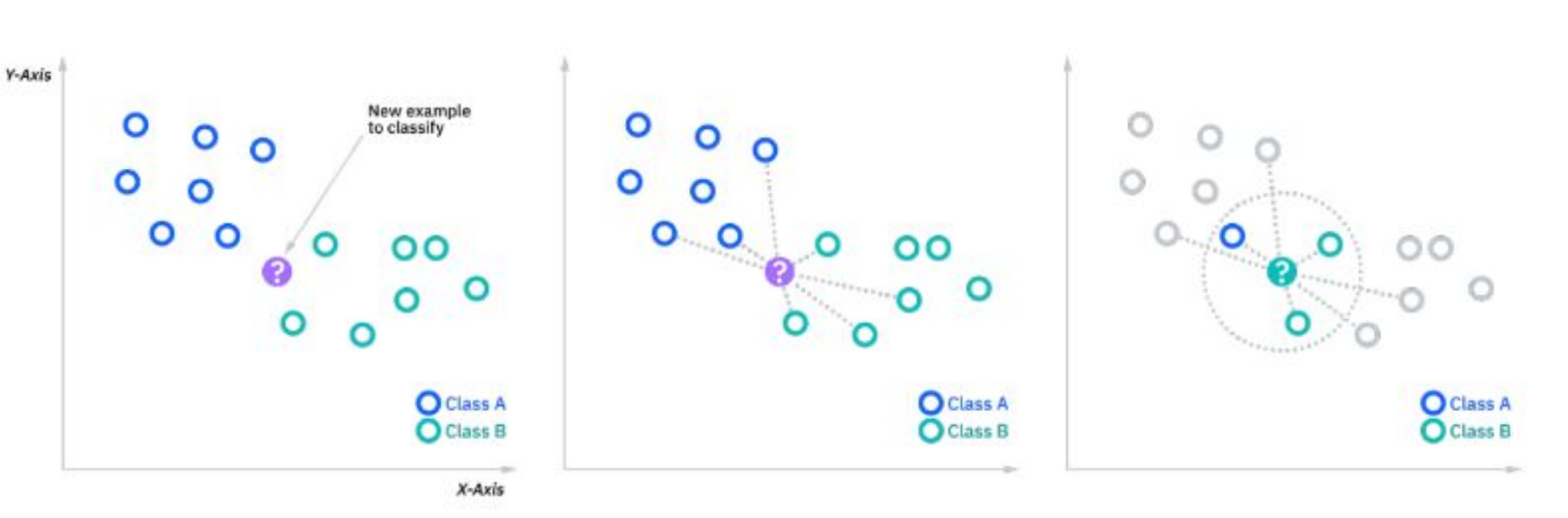
Classification

- It predicts the class of the new instance by a **majority vote**
  - If there are overall two classes, the one with >50% wins
  - If there are overall four classes, the one with >25% wins, etc.
- A **plurality vote** may also be used
  - select the class with the more votes

Regression

- The concept is the same but the target value is determined as the average of the values of the ***k***-nearest neighbors

k-Nearest Neighbors



- Various **Distance Metrics** can be used:
  - Euclidean distance, Manhattan distance, etc.
- In case of a **tie**, reduce ***k*** until tie is broken

DBSCAN combined with KNN

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbscan.components_, dbscan.labels_[dbscan.core_sample_indices_])
```

for Classification

on the moons dataset

```
X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
knn.predict(X_new)
```

```
array([1, 0, 1, 0])
```

```
knn.predict_proba(X_new)
```

```
array([[0.18, 0.82],
       [1.  , 0.  ],
       [0.12, 0.88],
       [1.  , 0.  ]])
```

Result of combining BDSCAN with KNN

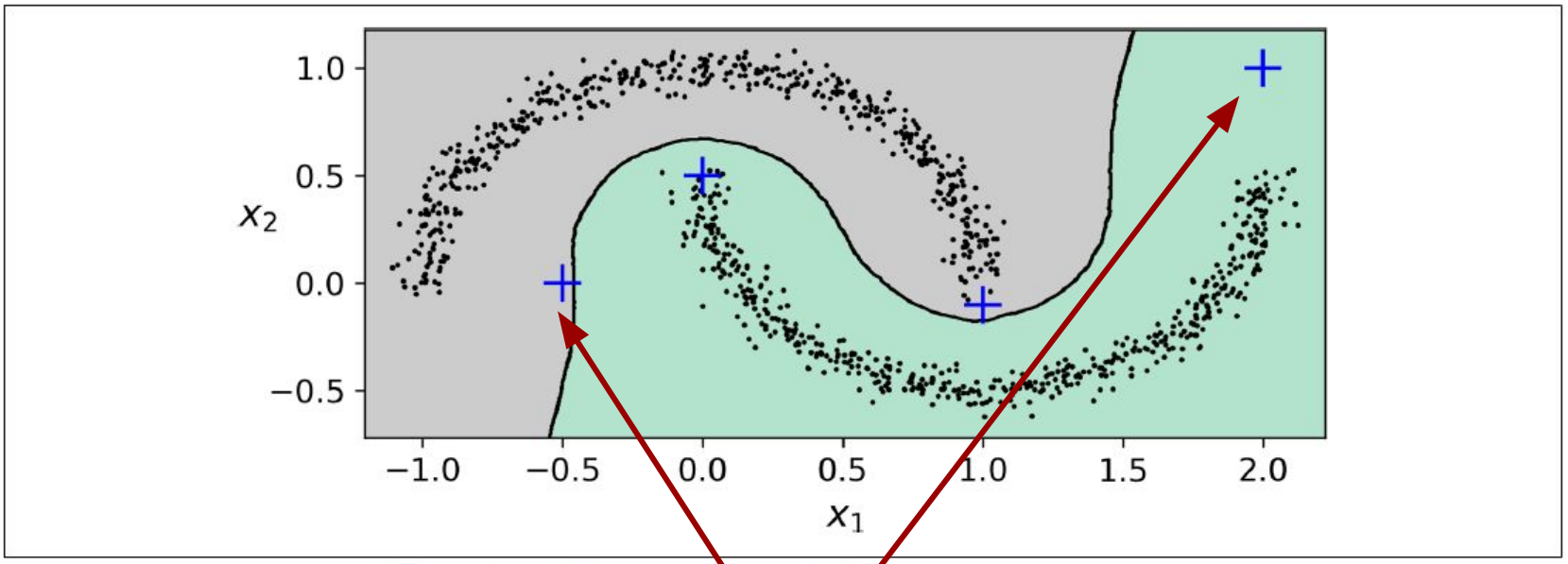


Figure 9-15. Decision boundary between two clusters

- These two new instances will be assigned to the closest class
  - Unless you define a maximum distance for instances to belong to the class
- If the distance is exceeded, the instance will be classified as an anomaly

DBSCAN

Limitations

- DBSCAN will underperform, when:
  - Density varies significantly across clusters
  - There is no sufficiently low-density region around the clusters

  ⇒ in such cases, use **hierarchical DBSCAN**
- Computationally complex so it **does not scale well to large datasets**

That’s all, Folks!

Thank you!