Machine Learning

Lecture 6: Unsupervised Learning and Clustering

Instructor: Dr. Farhad Pourkamali Anaraki

Introduction

- The vast majority of available data is unlabeled, i.e., we have the input features $\mathbf{X}$ but not the labels $\mathbf{y}$
  - Diagnose skin cancer from lesion images
  - Take thousands of pictures every day
  - Need to label each picture as "cancerous" or "non-cancerous" to train binary classifiers
  - Long, costly, and tedious task

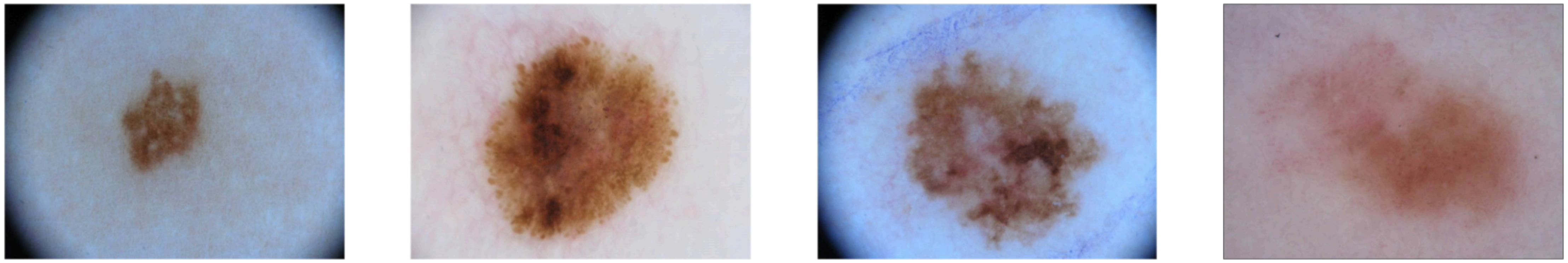
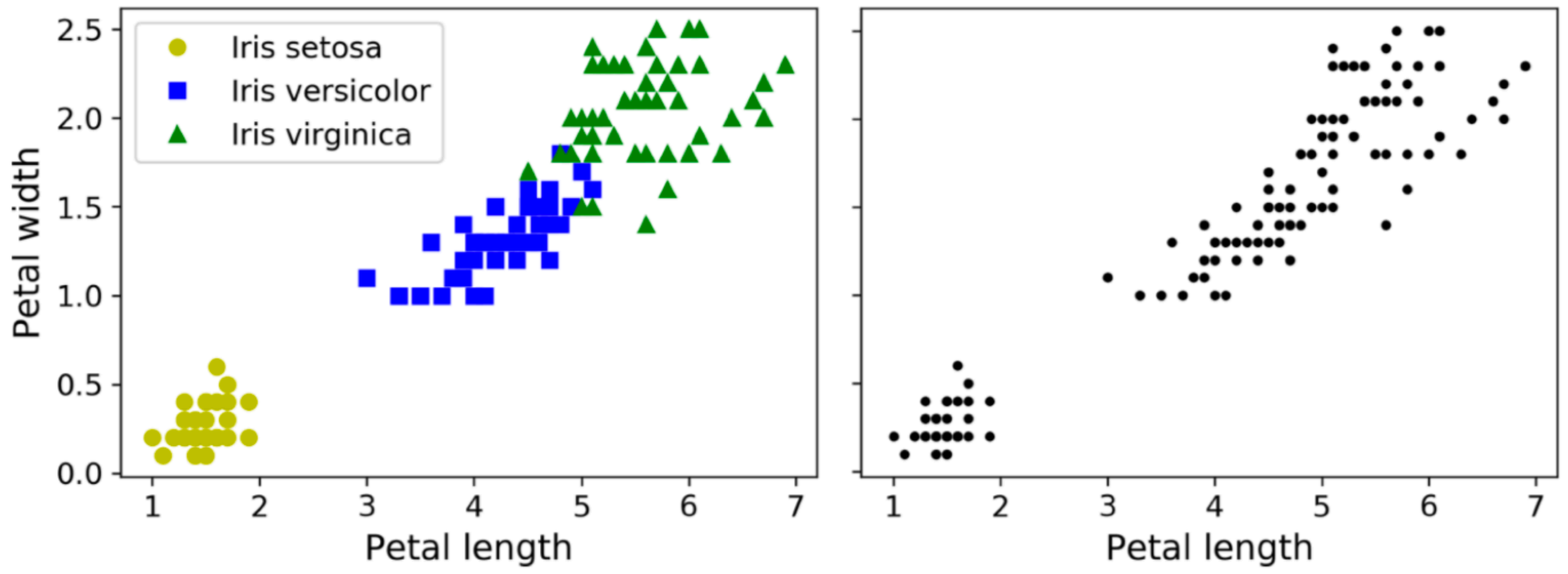


Figure 4. Skin cancer (melanoma) example lesions from the ISIC 2016 melanoma diagnosis dataset. The two lesions on the left are benign (non-cancerous), while the two lesions on the right are malignant (cancerous).

Types of unsupervised learning

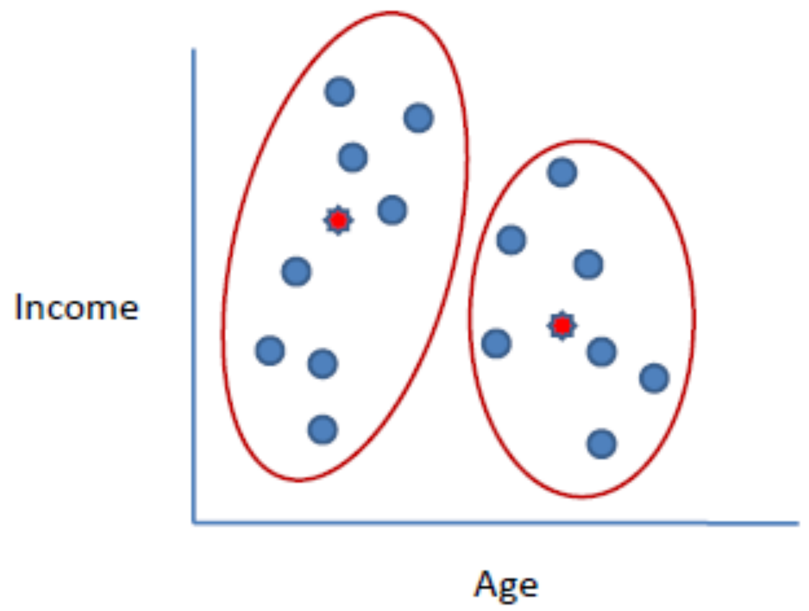
- Clustering
  - Group similar instances together into clusters



- Anomaly detection
  - Learn “what” normal data look like and then use to detect abnormal instances
- Density estimation
  - Estimating the probability density function of the random process that generated the data set

Different clustering algorithms

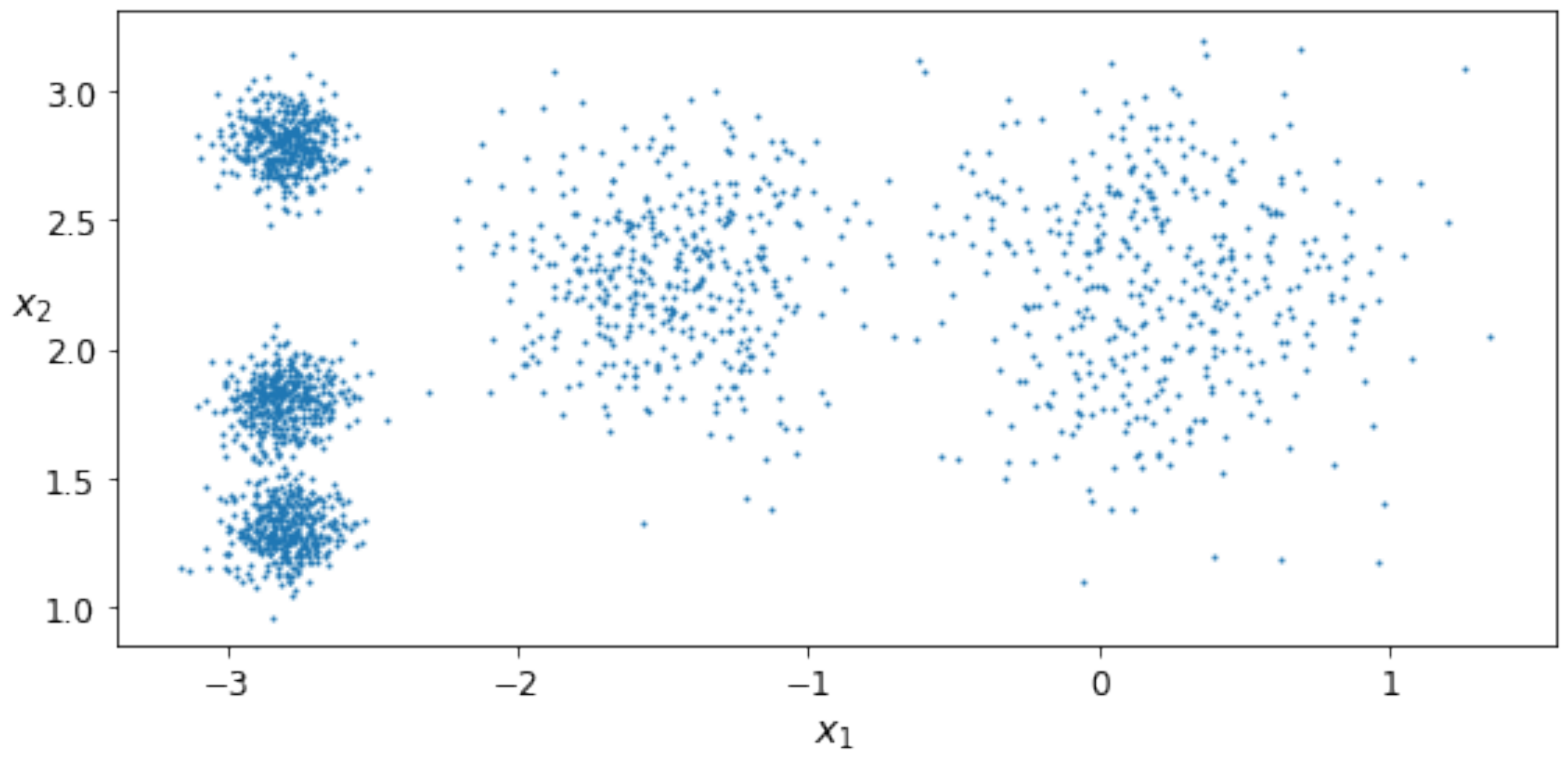
- There is no universal definition of what a cluster is
  - It depends on the context and different algorithms will capture different kinds of clusters
  - Most common approach is to look for instances centered around a particular point, called centroid



- In this lecture note, we will discuss two popular clustering algorithms
  - K-means
  - DBSCAN

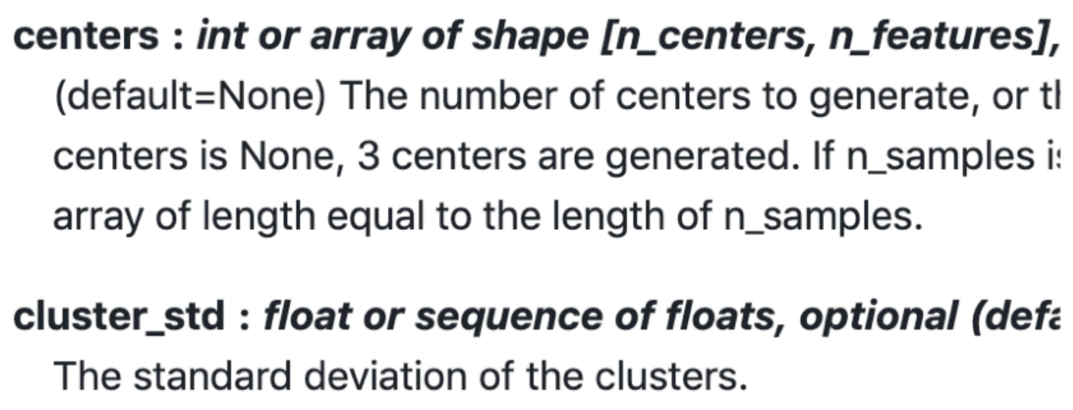
K-means

- Consider the unlabeled data set with 5 blobs of instances
- The K-means clustering algorithm is a simple technique for finding these clusters



- Generated using built-in functions

```
X, y = make_blobs(n_samples=2000, centers=blob_centers,
                  cluster_std=blob_std, random_state=7)
```


centers : *int or array of shape [n_centers, n_features],*
(default=None) The number of centers to generate, or t
centers is None, 3 centers are generated. If n_samples i
array of length equal to the length of n_samples.

cluster_std : *float or sequence of floats, optional (def*
The standard deviation of the clusters.

Training K-means clustering

- Find each blob's center or assign each instance to the closest blob

```
from sklearn.cluster import KMeans
k = 5
kmeans = KMeans(n_clusters=k, random_state=42).fit(X)
kmeans.labels_
```

Index of the cluster each instance gets assigned to

```
array([4, 1, 0, ..., 3, 0, 1], dtype=int32)
```

- Estimated cluster centers

```
kmeans.cluster_centers_
```

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

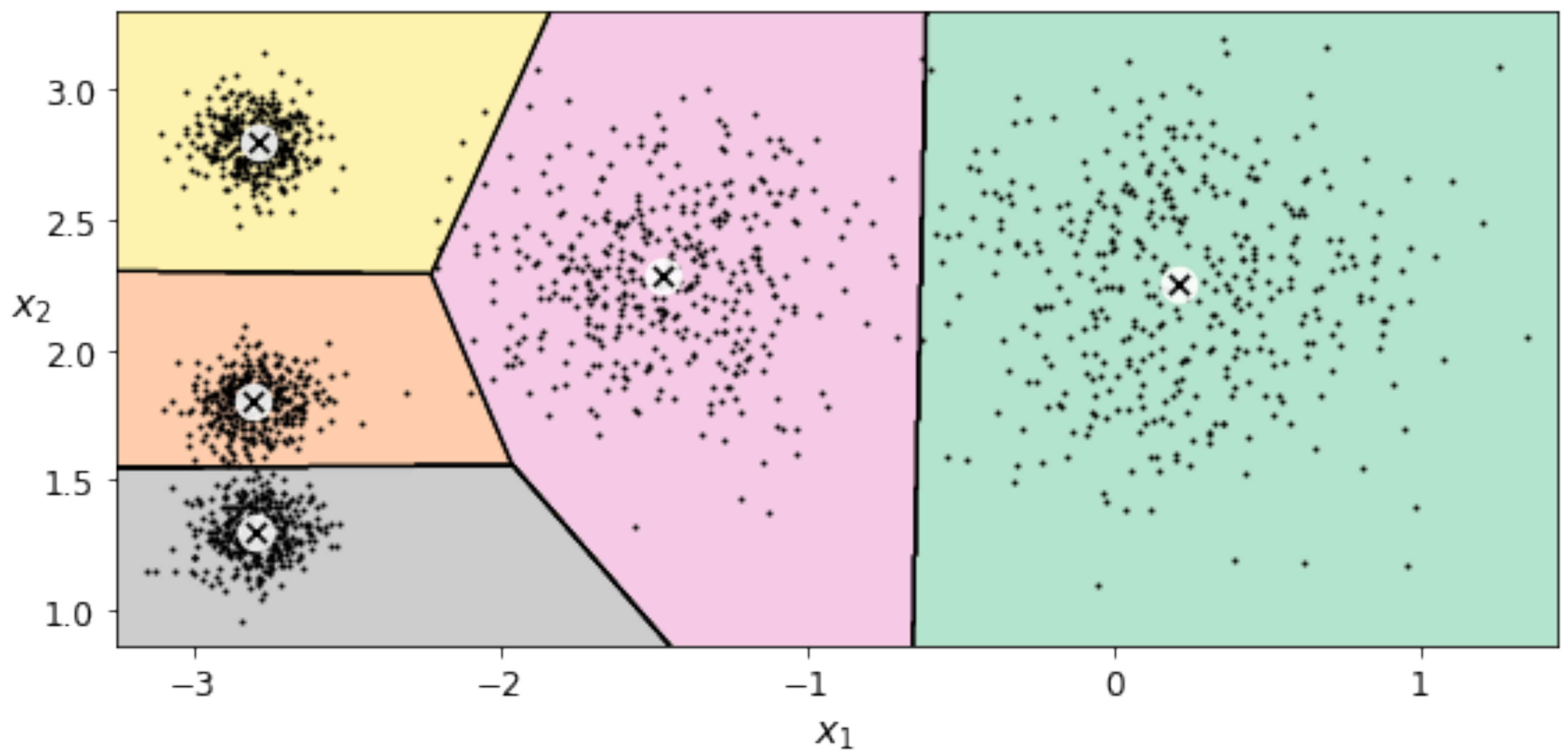
- Making predictions

```
X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
kmeans.predict(X_new)
```

```
array([0, 0, 3, 3], dtype=int32)
```

Decision boundaries

- Let's plot the model's decision boundaries



- A few instances are mislabeled (especially near the boundary between the top-left cluster and the central cluster)

- K-means performance degrades when the blobs have very different diameters

Hard clustering vs soft clustering

- Hard clustering
  - Assigning each instance to a single cluster
- Soft clustering
  - Assigning each instance a score per cluster (like distance between the instance and the centroid or a similarity score)

```
kmeans.transform(X_new)
```

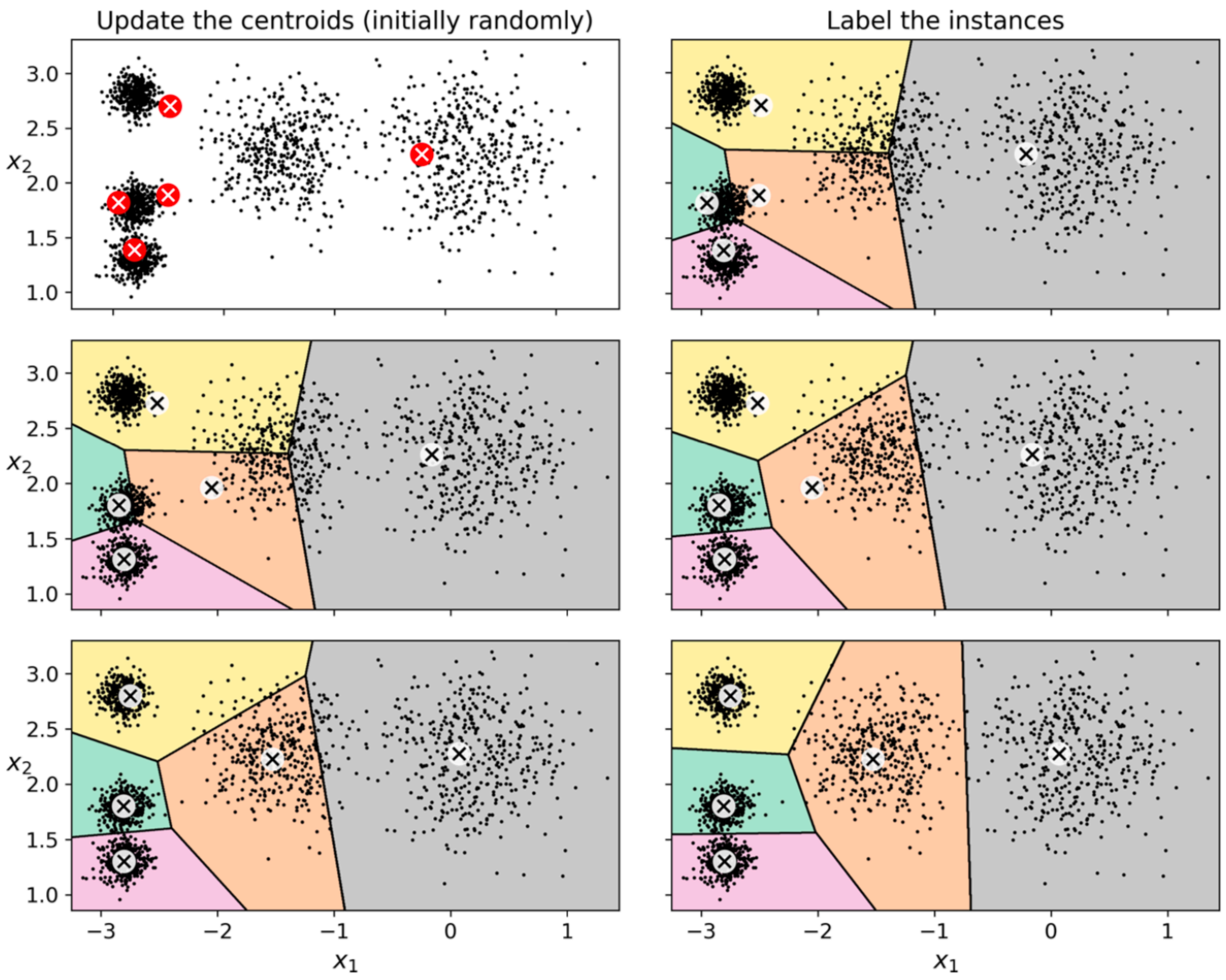
```
array([[0.32995317, 2.81093633, 1.49439034, 2.9042344 , 2.88633901],
       [2.80290755, 5.80730058, 4.4759332 , 5.84739223, 5.84236351],
       [3.29399768, 1.21475352, 1.69136631, 0.29040966, 1.71086031],
       [3.21806371, 0.72581411, 1.54808703, 0.36159148, 1.21567622]])
```

How does K-means work?

- Start by placing the centroids randomly (pick k instances at random)
- Iterate over the following two steps
  - Assign each instance to the cluster whose centroid is closest
  - Update the centroids by computing the mean of instances for each cluster

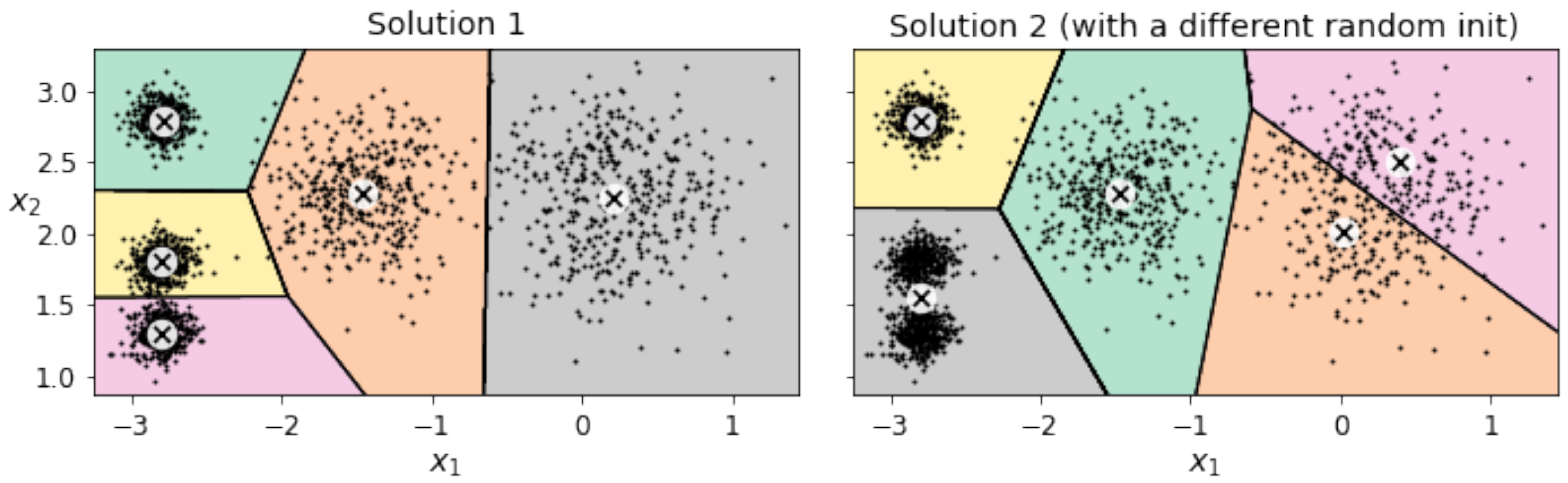
- The algorithm is guaranteed to converge in a finite number of iterations
- The computational complexity is linear concerning the number of samples, number of features, and number of clusters
  - K-means is generally one of the fastest clustering algorithms

Illustration of K-means



Centroid initialization methods

- Although the algorithm is guaranteed to converge, it may not converge to the right solution
    - Highly sensitive to the centroid initialization



```
kmeans_rnd_init1 = KMeans(n_clusters=5, init="random", n_init=1,
                          algorithm="full", random_state=11)
kmeans_rnd_init2 = KMeans(n_clusters=5, init="random", n_init=1,
                          algorithm="full", random_state=25)
```

Reducing the impact of initialization

- Run the algorithm multiple times with different random initializations and keep the best solution
  - How to keep the best solution?
    - Compute the sum of squared distances of instances to their closest centroid (inertia)

```
print(kmeans_rnd_init1.inertia_, kmeans_rnd_init2.inertia_)
```

```
211.59853725816822 223.29108572819035
```

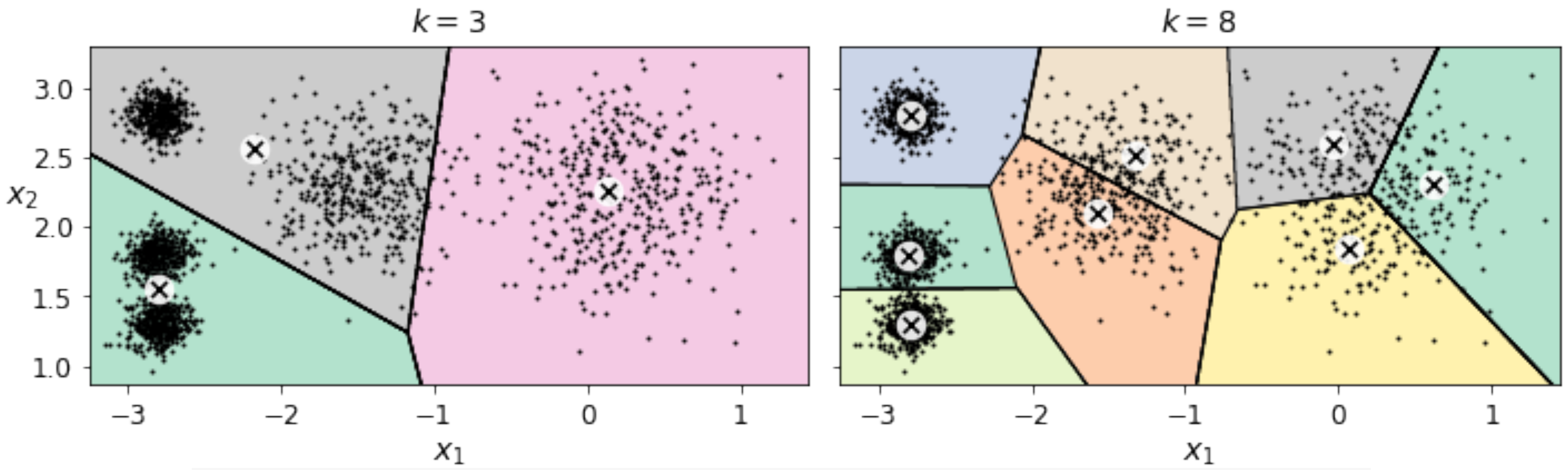
    - The score() method return the negative inertia so that "greater is better"

```
print(kmeans_rnd_init1.score(X),kmeans_rnd_init2.score(X))
```

```
-211.59853725816856 -223.2910857281904
```

Finding the optimal number of clusters

- So far, we have set the number of clusters k to 5
    - Setting k to 3 or 8 results in fairly bad results



```
kmeans_k3 = KMeans(n_clusters=3, random_state=42)
kmeans_k8 = KMeans(n_clusters=8, random_state=42)
```

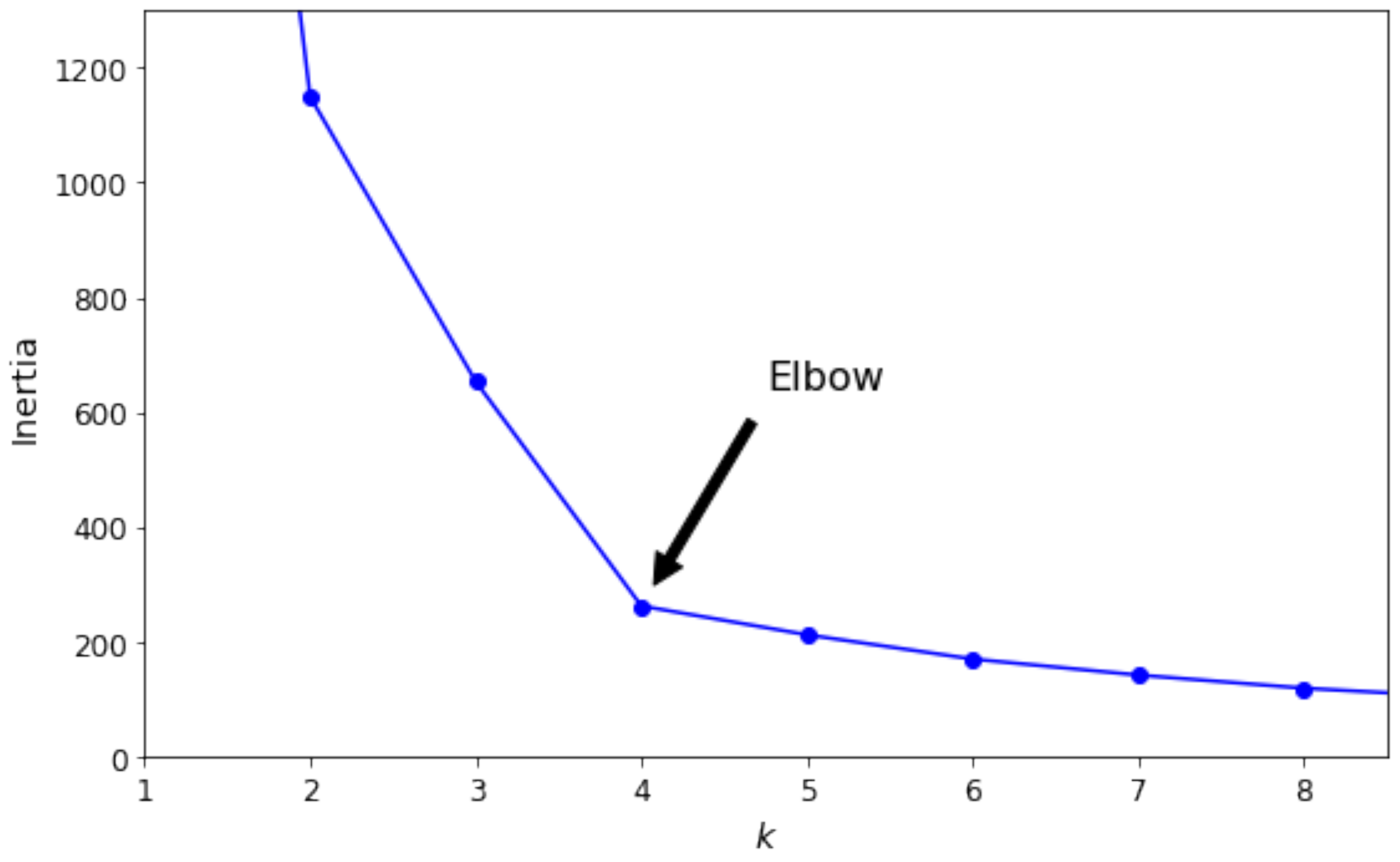
- Can we just pick the model with the lowest inertia? It's not that simple!

```
print(kmeans_k3.inertia_, kmeans_k8.inertia_)
```

```
653.2167190021553 118.41983763508077
```

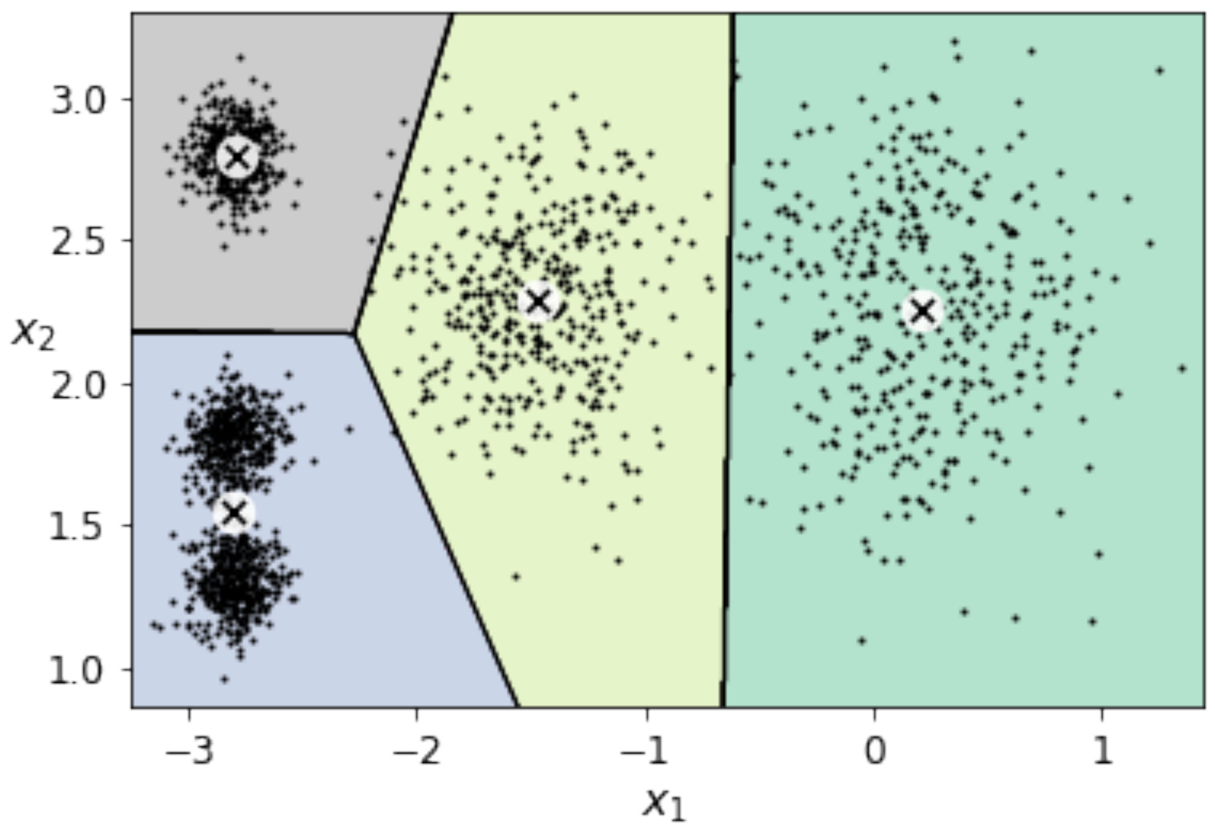
Finding the optimal number of clusters

- The values of inertia keep getting smaller as we increase k
    - Having more clusters implies that, on average, each instance should be closer to its centroid
- Plot the inertia as a function of k



Finding the optimal number of clusters

- When we set the number of clusters $k = 4$, we get the following partitioning



- While this approach works, there are more principled techniques
    - Silhouette coefficient: measure of clustering quality

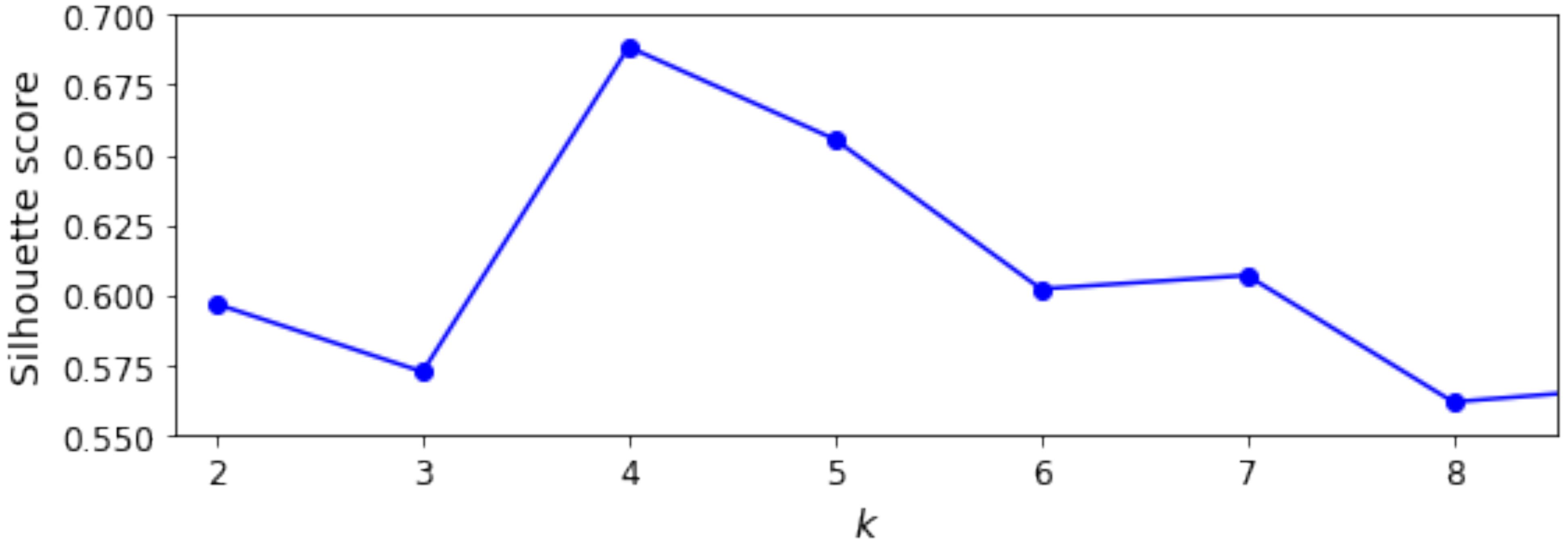
Silhouette coefficient

- Silhouette coefficient for each instance can be computed as $(b - a)/\max(a, b)$
    - a: mean distance to the other instances in the same cluster (mean intra-cluster distance)
    - b: mean nearest-cluster distance
        - +1: instance is well inside its own cluster
        - 0: close to a cluster boundary
        - -1: assigned to the wrong cluster
- Scikit-learn implementation

```
from sklearn.metrics import silhouette_score
silhouette_scores = [silhouette_score(X, model.labels_)
                     for model in kmeans_per_k[1:]]
```

Finding the optimal number of clusters

- Using the silhouette score

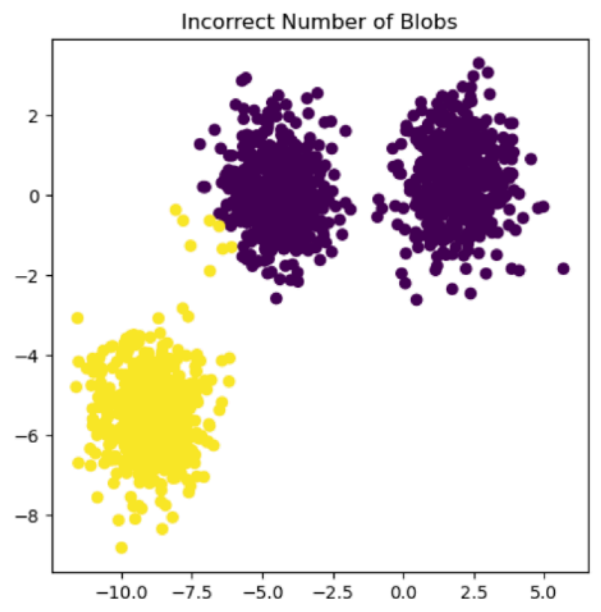


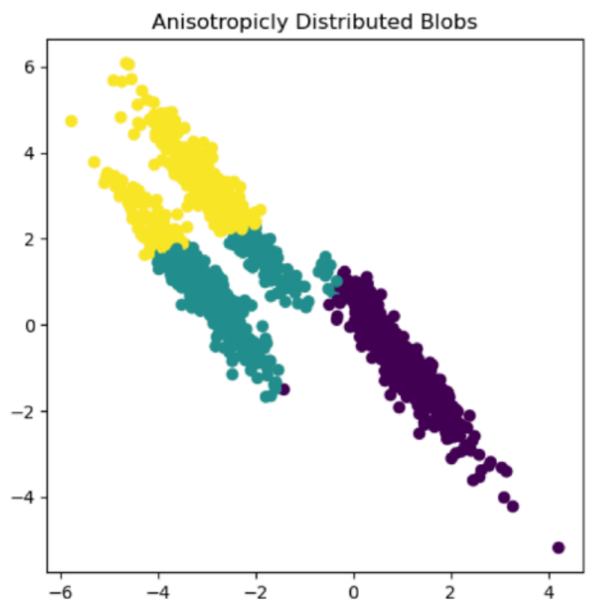
- Both $k = 4$ and $k = 5$ are good choices.

Limits of K-means

- While K-means is fast, simple, and scalable, there are several limitations
  - Run the algorithm several times and choose the number of clusters
  - Clusters that have varying sizes, different densities, and non spherical shapes

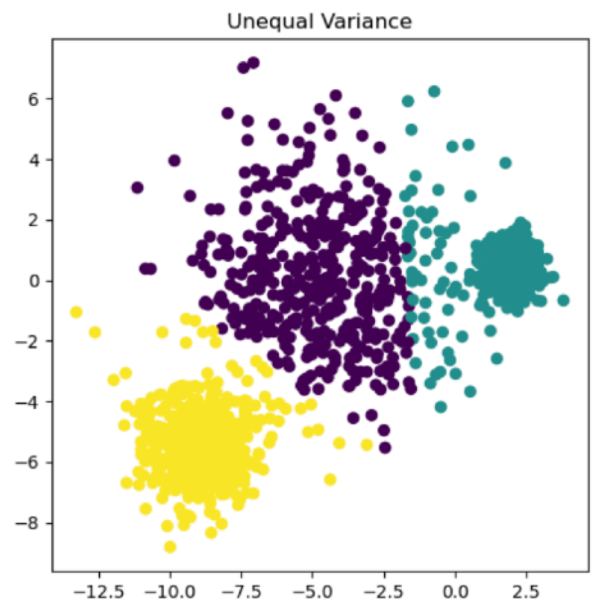
Incorrect number of blobs

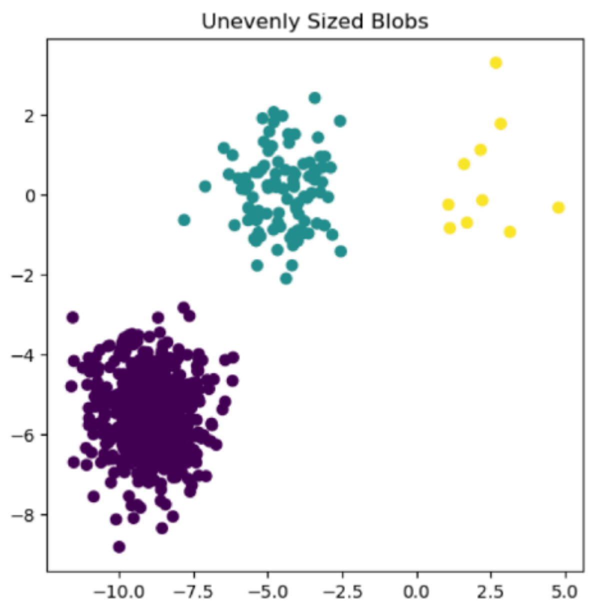




Non spherical shapes

Unequal variances





Unequally sized blobs

Using clustering for image segmentation

- Image segmentation is the task of partitioning an image into multiple segments
    - All pixels that are part of the same object type get assigned to the same segment based on some features (like color or more complicated features obtained by convolutional neural networks)

```
# Download the ladybug image
from six.moves import urllib
images_path = os.path.join(PROJECT_ROOT_DIR, "images", "unsupervised_learning")
os.makedirs(images_path, exist_ok=True)
DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"
filename = "ladybug.png"
print("Downloading", filename)
url = DOWNLOAD_ROOT + "images/unsupervised_learning/" + filename
urllib.request.urlretrieve(url, os.path.join(images_path, filename))
```

```
Downloading ladybug.png
('./images/unsupervised_learning/ladybug.png',
 <http.client.HTTPMessage at 0x7f13825abba8>)
```

```
from matplotlib.image import imread
image = imread(os.path.join(images_path, filename))
image.shape
```

```
(533, 800, 3)
```

Data set description

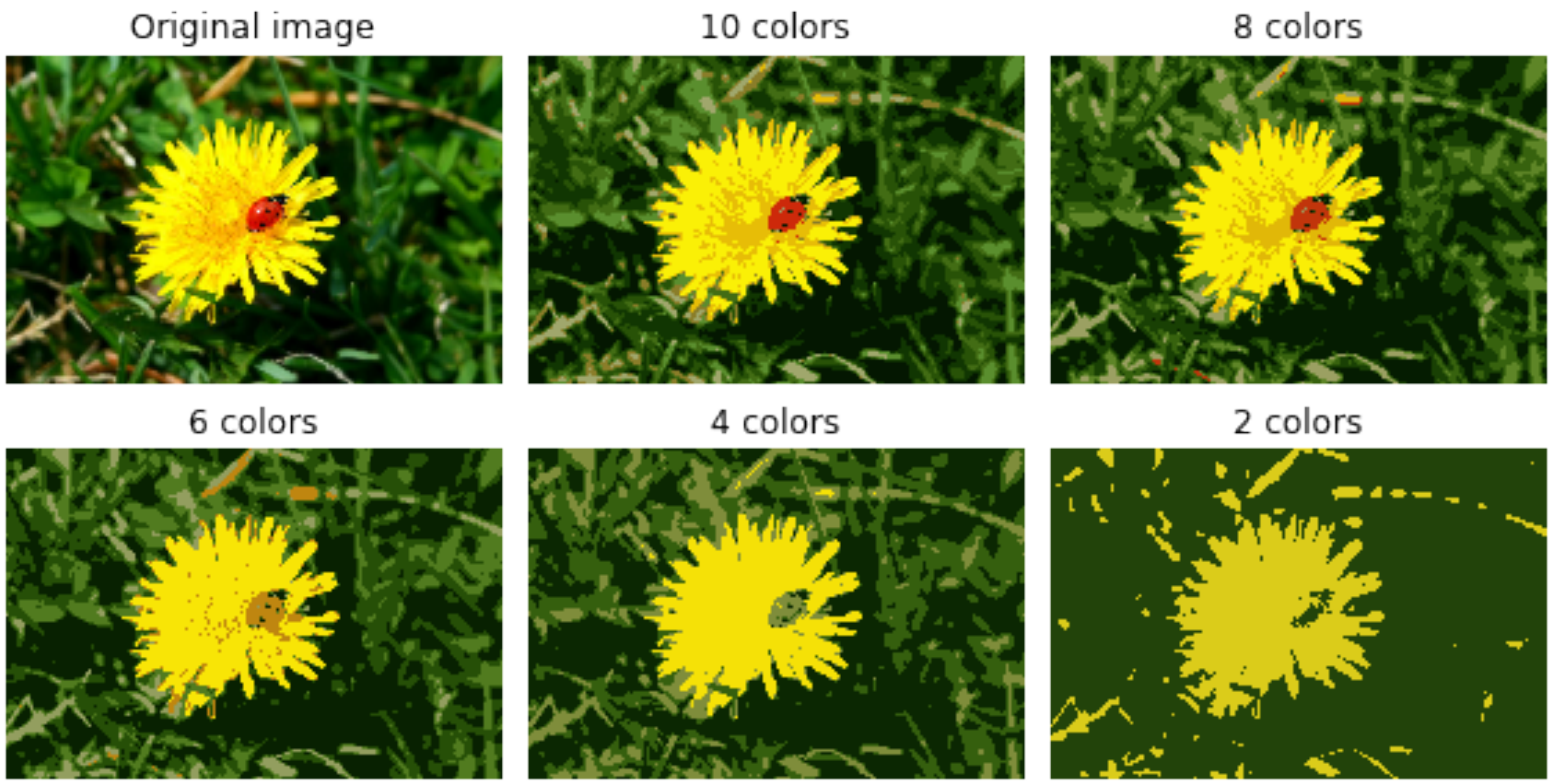
- Image represented as a 3D array
  - First dimension: height
  - Second dimension: width
  - Third dimension: number of color channels (RGB)
  - Each pixel between 0 and 1 (or 0 and 255)

```
X = image.reshape(-1, 3)
kmeans = KMeans(n_clusters=8, random_state=42).fit(X)
segmented_img = kmeans.cluster_centers_[kmeans.labels_]
segmented_img = segmented_img.reshape(image.shape)
```

```
print(X.shape, segmented_img.shape)
```

```
(426400, 3) (533, 800, 3)
```

Results



Clustering for preprocessing

- Clustering can be used to reduce the number of features before applying a supervised learning algorithm

- Let's work with the MNIST data set and use logistic regression for classification

```
from sklearn.datasets import load_digits
X_digits, y_digits = load_digits(return_X_y=True)
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X_digits, y_digits, random_state=42)
```

```
from sklearn.linear_model import LogisticRegression
```

```
log_reg = LogisticRegression(multi_class="ovr",
        solver="lbfgs", max_iter=5000, random_state=42)
log_reg.fit(X_train, y_train)
```

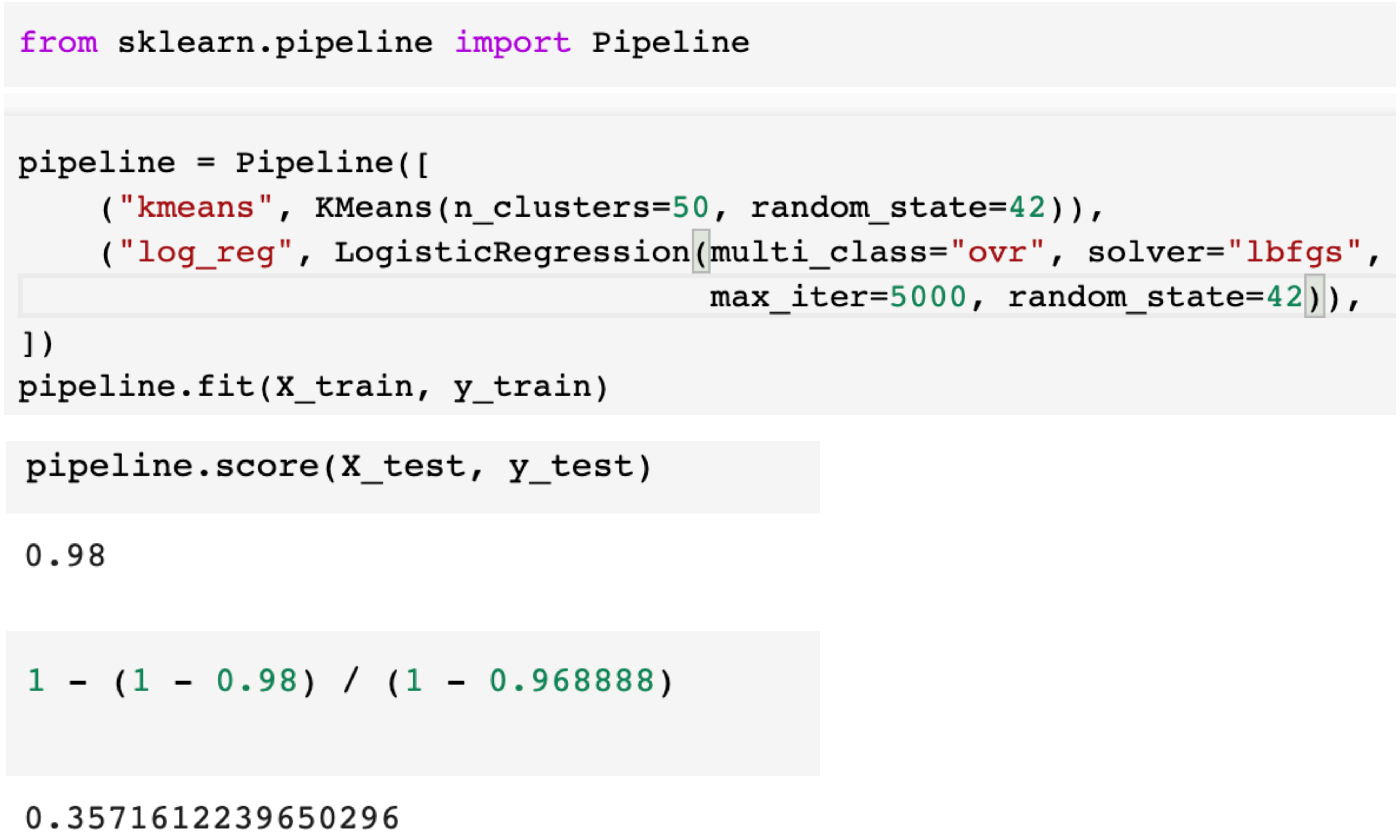
```
LogisticRegression(C=1.0, class_weight=None, dual=False,
                   intercept_scaling=1, l1_ratio=None, ma
                   multi_class='ovr', n_jobs=None, penalt
```

```
log_reg.score(X_test, y_test)
```

```
0.9688888888888889
```

K-means clustering in the Pipeline

- Create a pipeline that will first cluster the training set into 50 groups and replace the images with corresponding labels, then apply a logistic regression model



```
from sklearn.pipeline import Pipeline
```

```
pipeline = Pipeline([
    ("kmeans", KMeans(n_clusters=50, random_state=42)),
    ("log_reg", LogisticRegression(multi_class="ovr", solver="lbfgs",
                                   max_iter=5000, random_state=42)),
])
pipeline.fit(X_train, y_train)
```

```
pipeline.score(X_test, y_test)
```

```
0.98
```

```
1 - (1 - 0.98) / (1 - 0.968888)
```

```
0.3571612239650296
```

- We reduced the error rate by almost 35%

Preprocessing — finding the number of clusters

- We can use an exhaustive search over specified parameter values

```
from sklearn.model_selection import GridSearchCV
param_grid = dict(kmeans__n_clusters=range(2, 100))
grid_clf = GridSearchCV(pipeline, param_grid, cv=3, verbose=2)
grid_clf.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 98 candidates, totalling 294 fits
[CV] kmeans__n_clusters=2 ............................................
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[CV] ............................. kmeans__n_clusters=2, total=   0.2s
[CV] kmeans__n_clusters=2 ............................................
[CV] ............................. kmeans__n_clusters=2, total=   0.2s
[CV] kmeans__n_clusters=2 ............................................
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.2s remaining:    0.0s
[CV] ............................. kmeans__n_clusters=2, total=   0.2s
[CV] kmeans__n_clusters=3 ............................................
[CV] ............................. kmeans__n_clusters=3, total=   0.3s
[CV] kmeans__n_clusters=3 ............................................
[CV] ............................. kmeans__n_clusters=3, total=   0.3s
[CV] kmeans__n_clusters=3 ............................................
```

```
grid_clf.best_params_
```

```
{'kmeans__n_clusters': 57}
```

DBSCAN

- This algorithm defines clusters as continuous regions of high density
  - For each instance, it counts how many samples are located within a small distance ε from it (ε-neighborhood)
  - If an instance has at least *min_samples* samples in its ε-neighborhood, then it is considered a "core instance"
  - All instances in the neighborhood of a core instance belong to the same cluster
  - Any instance that is not a "core instance" and does not have one in its neighborhood is considered an anomaly

Implementation

- The DBSCAN class in Scikit-Learn has two main input arguments

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

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

- Note that some instances have a cluster index equal to -1 (anomalies)

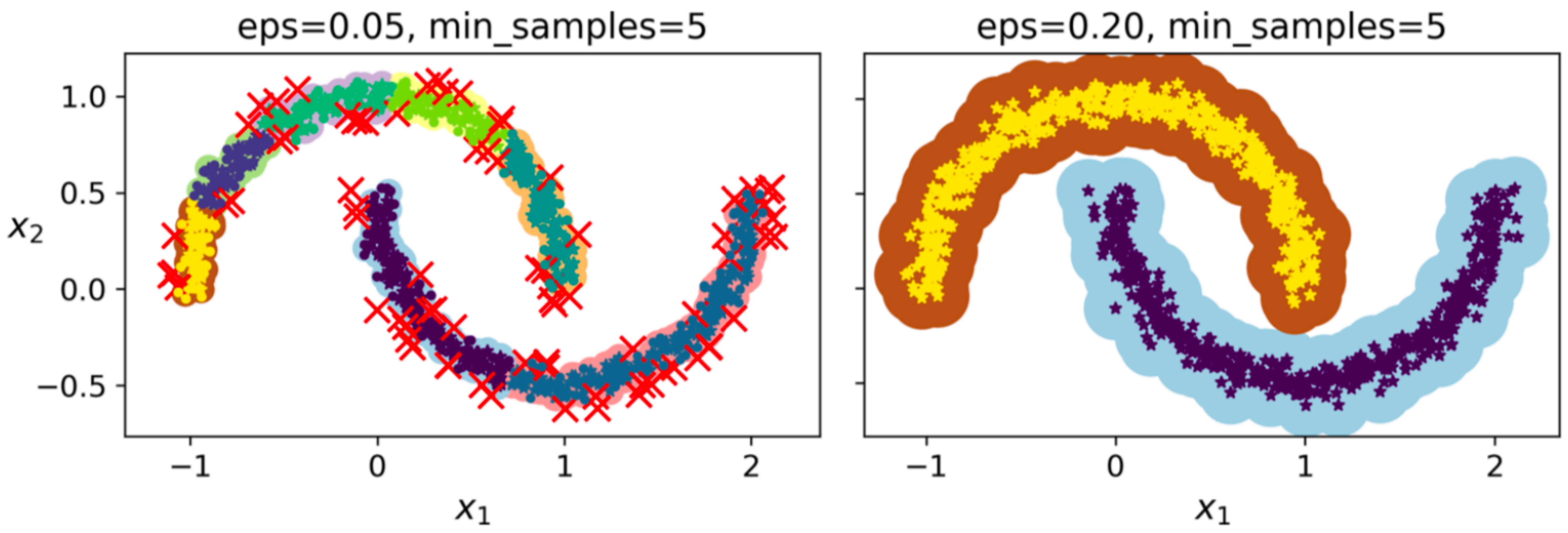
```
>>> dbscan.labels_
array([ 0,  2, -1, -1,  1,  0,  0,  0, ...,  3,  2,  3,  3,  4,  2,  6,  3])
```

- We can find the indices of the core instances

```
>>> len(dbscan.core_sample_indices_)
808
>>> dbscan.core_sample_indices_
array([ 0,   4,   5,   6,   7,   8,  10,  11, ..., 992, 993, 995, 997, 998, 999])
```

DBSCAN results

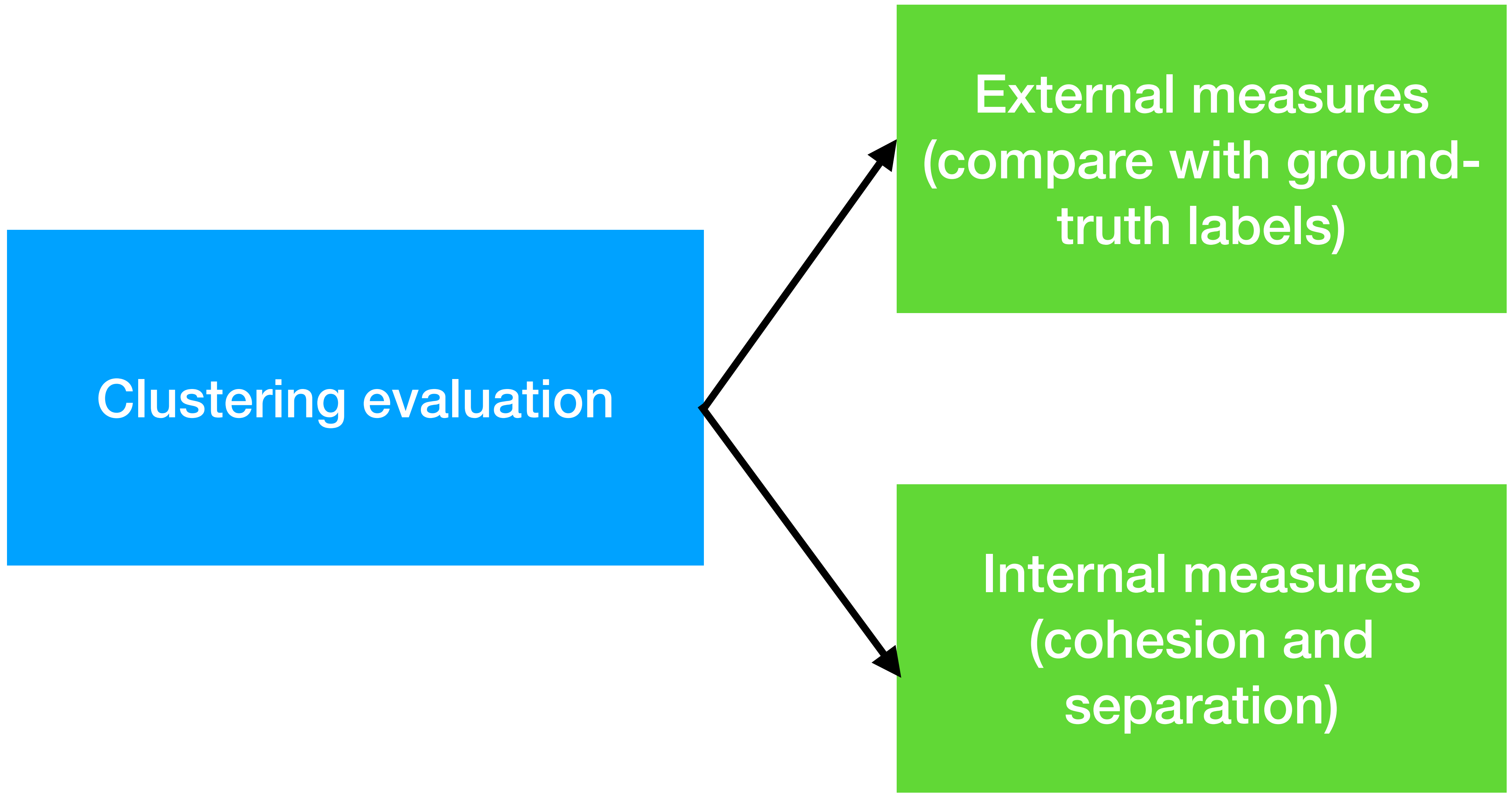
- DBSCAN clustering using two different neighborhood radiuses



- DBSCAN is capable of identifying any number of clusters of any shape
- Robust to outliers
- If the density varies significantly, it may be impossible to capture all clusters

Reading Assignment: Chapter 9 of textbook
"Unsupervised Learning Techniques"
Pages 235—258

Evaluating clustering performance


Clustering evaluation
External measures (compare with ground-truth labels)
Internal measures (cohesion and separation)

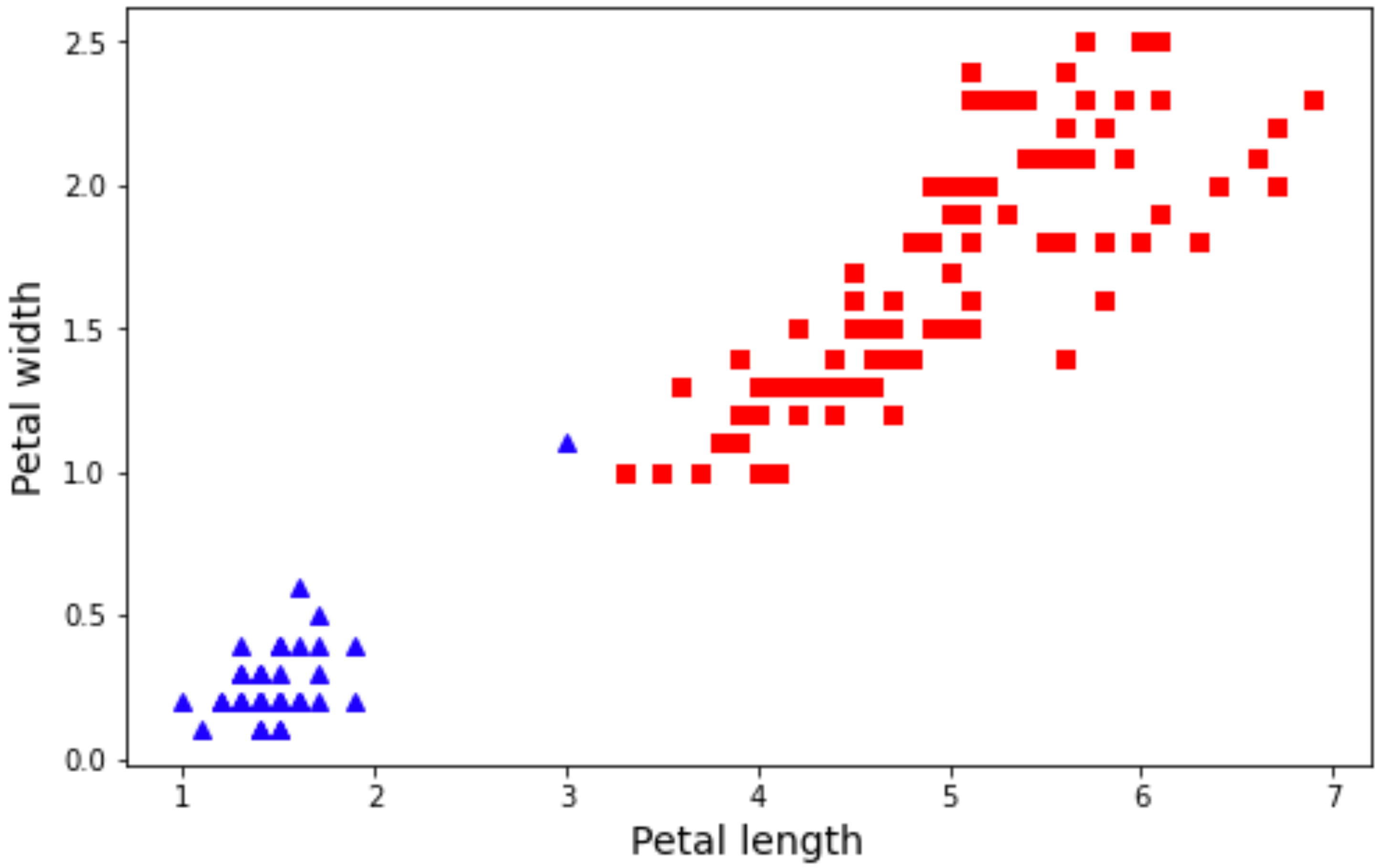
Example: K-means on Iris flower data set

```
from sklearn.cluster import KMeans

y_pred1 = KMeans(n_clusters=2,
                 random_state = 1).fit_predict(X)
plt.figure(figsize=(8, 5))
plt.plot(X[y_pred1==0, 0], X[y_pred1==0, 1], "rs")
plt.plot(X[y_pred1==1, 0], X[y_pred1==1, 1], "b^")
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.show()
```

```
acc1 = np.sum(y_pred1==y)/len(y)
print(acc1)
```

```
0.32666666666666666
```

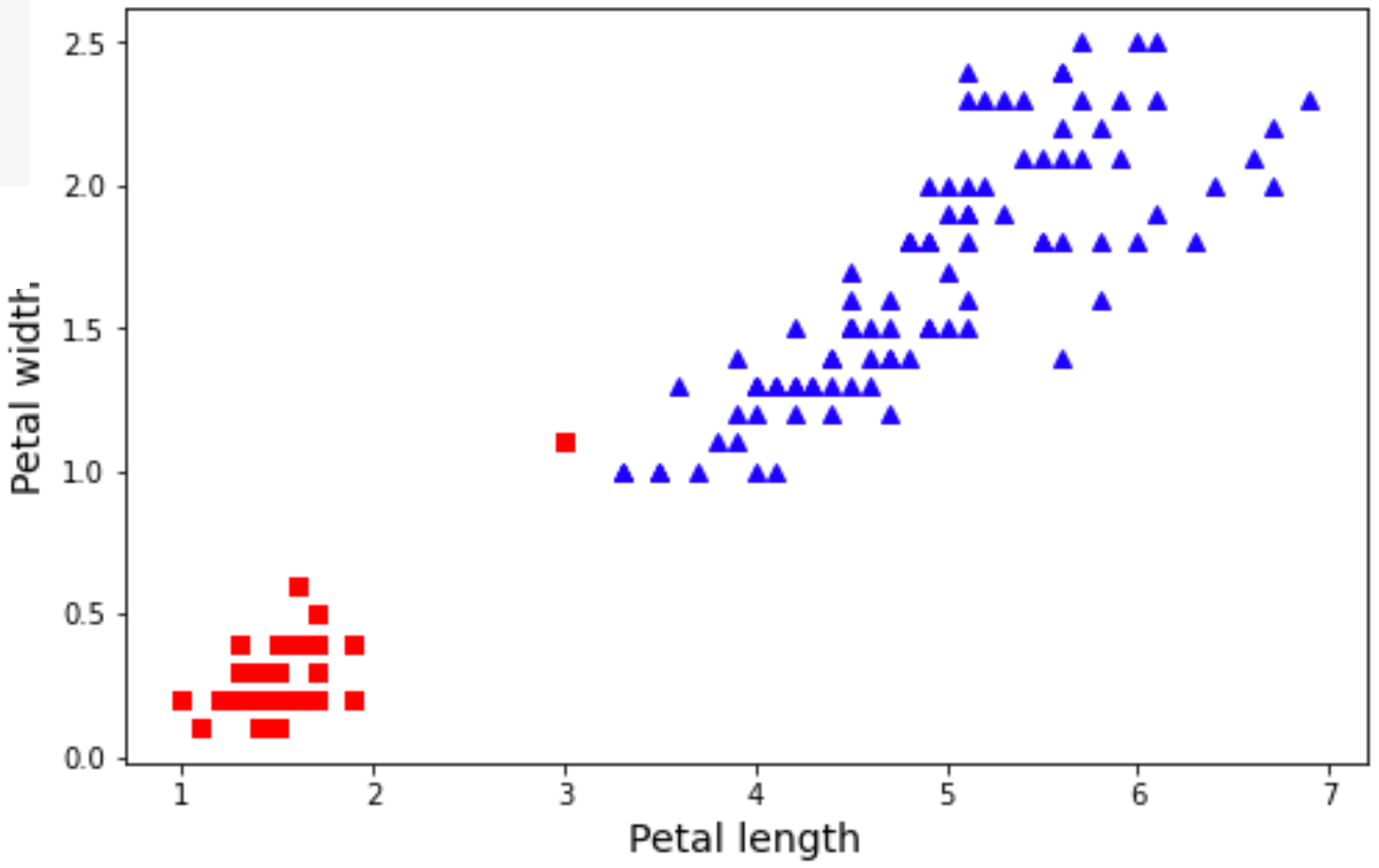


Another run

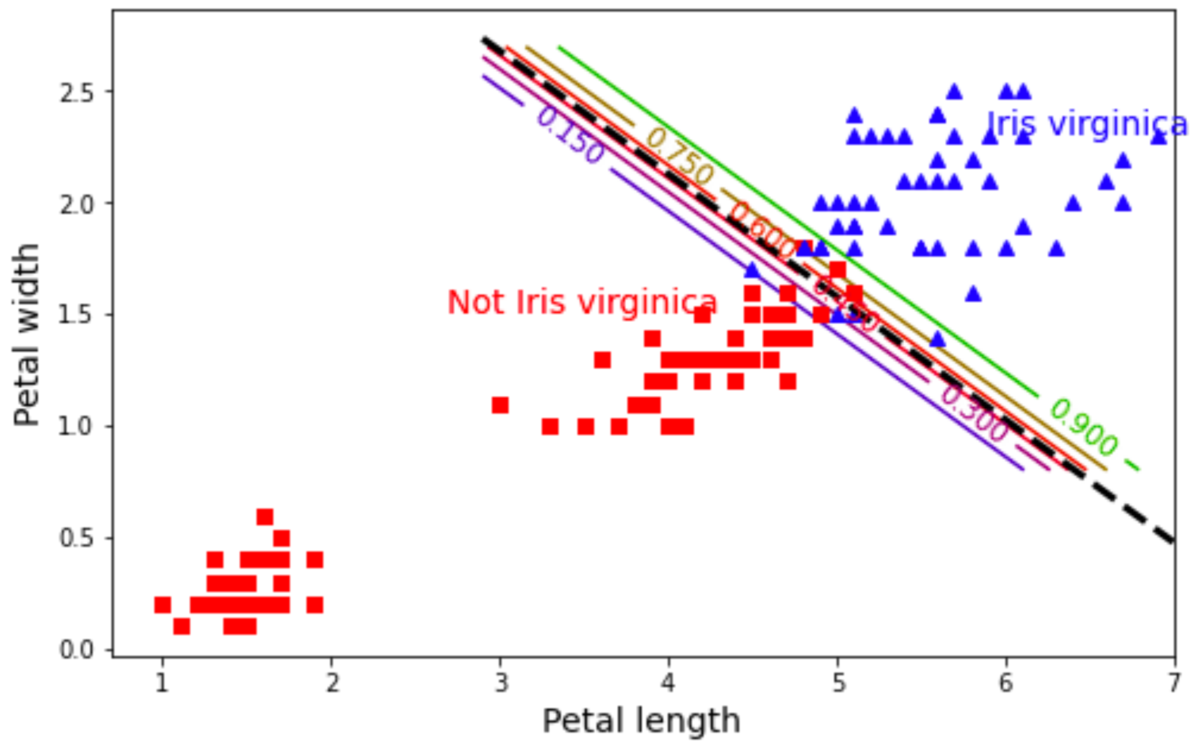
```
y_pred2 = KMeans(n_clusters=2,
                 random_state = 42).fit_predict(X)
plt.figure(figsize=(8, 5))
plt.plot(X[y_pred2==0, 0], X[y_pred2==0, 1], "rs")
plt.plot(X[y_pred2==1, 0], X[y_pred2==1, 1], "b^")
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.show()
```

```
acc2 = np.sum(y_pred2==y)/len(y)
print(acc2)
```

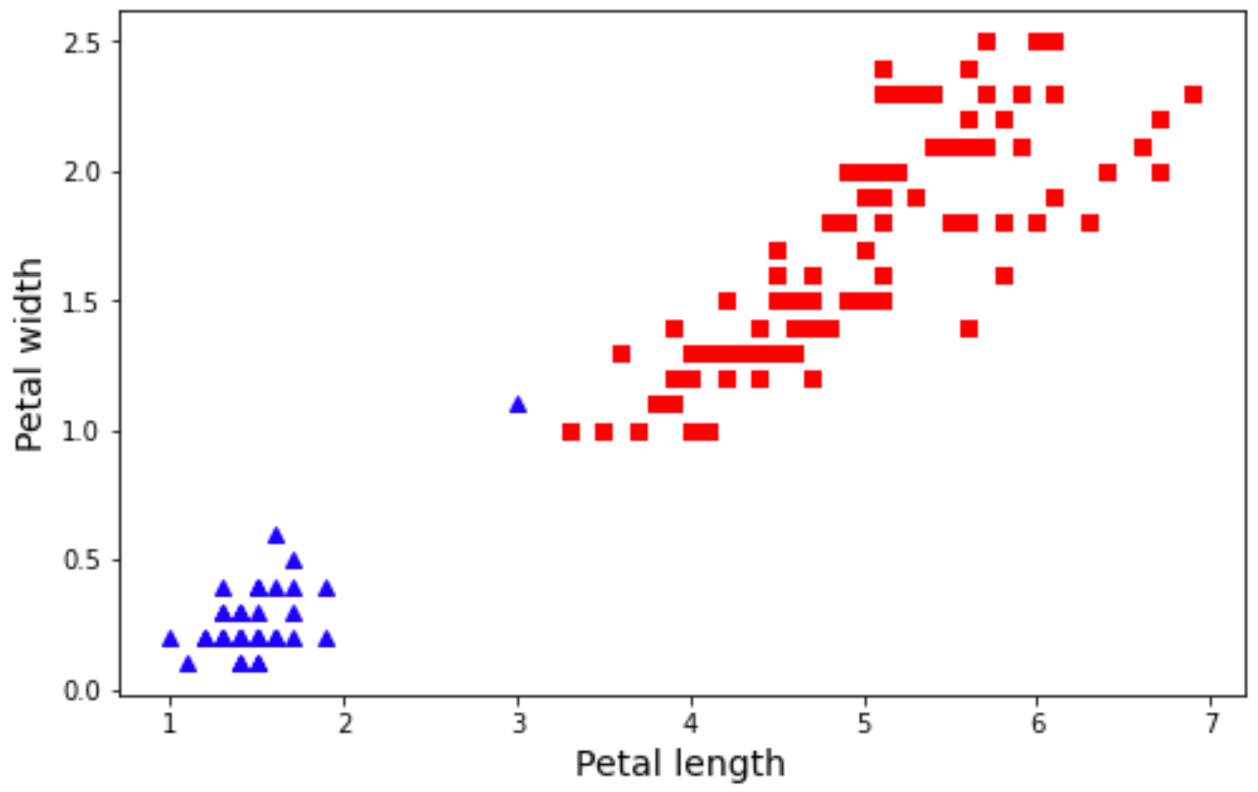
```
0.6733333333333333
```



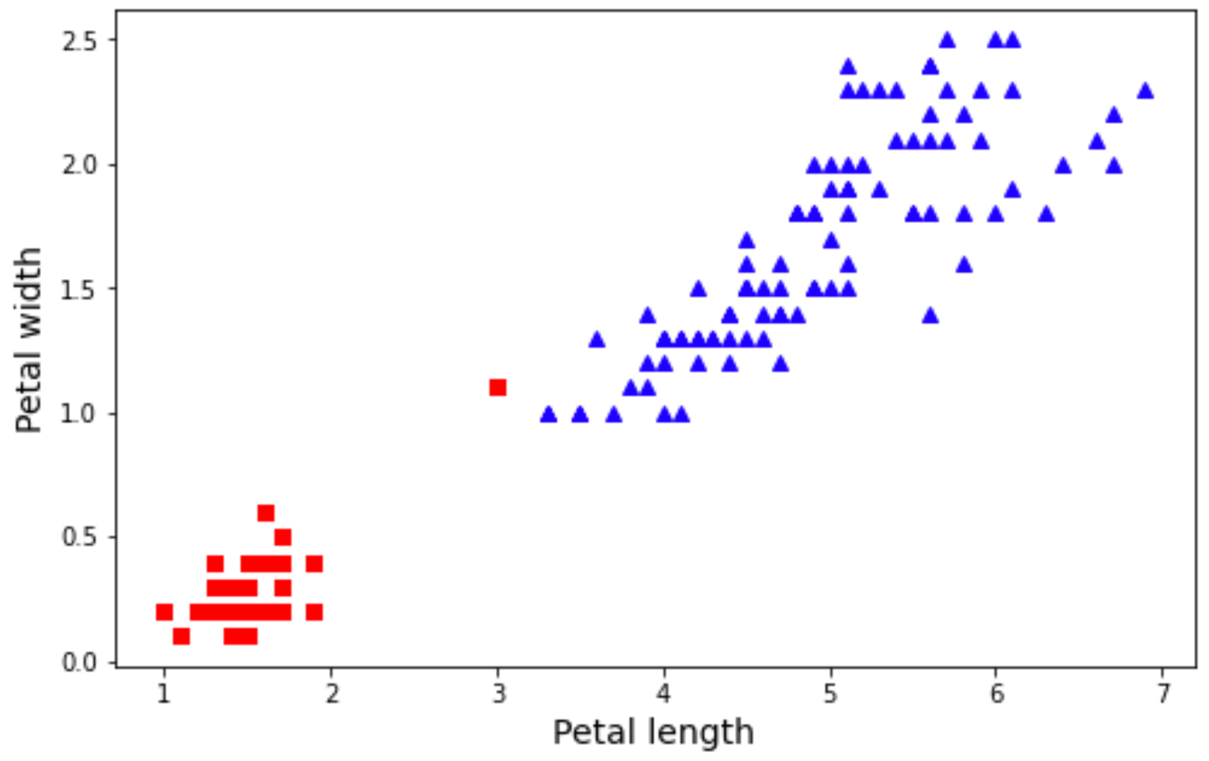
Why this happened?



returned labels 1



returned labels 2



What did we learn?

- Classification

$$acc(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n-1} \mathbf{1}\left(\hat{y}_i = y_i\right)$$

- Clustering

$$acc(y, \hat{y}) = \max_{perm} \frac{1}{n} \sum_{i=1}^{n-1} \mathbf{1}\left(prem(\hat{y}_i) = y_i\right)$$