Chapter 9 – Unsupervised Learning

This notebook contains all the sample code and solutions to the exercises in chapter 9.

Open in Colab Open in Kaggle

Setup

This project requires Python 3.7 or above:

```
In [1]: import sys

        assert sys.version_info >= (3, 7)
```

It also requires Scikit-Learn ≥ 1.0.1:

```
In [2]: from packaging import version
        import sklearn
        import numpy as np

        assert version.parse(sklearn.__version__) >= version.parse("1.0.1")
```

As we did in previous chapters, let's define the default font sizes to make the figures prettier:

```
In [3]: import matplotlib.pyplot as plt

        plt.rc('font', size=14)
        plt.rc('axes', labelsize=14, titlesize=14)
        plt.rc('legend', fontsize=14)
        plt.rc('xtick', labelsize=10)
        plt.rc('ytick', labelsize=10)
```

And let's create the `images/unsupervised_learning` folder (if it doesn't already exist), and define the `save_fig()` function which is used through this notebook to save the figures in high-res for the book:

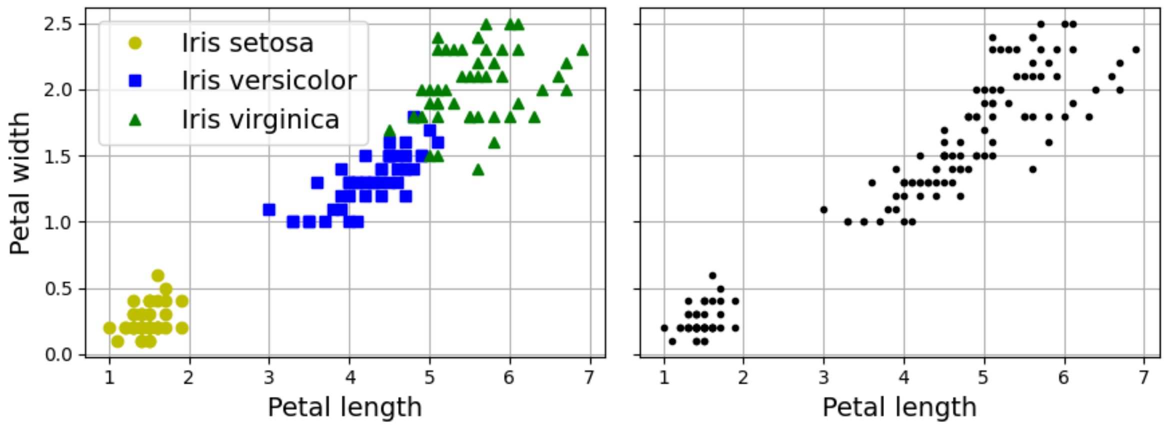
```
In [4]: from pathlib import Path

        IMAGES_PATH = Path() / "images" / "unsupervised_learning"
        IMAGES_PATH.mkdir(parents=True, exist_ok=True)

        def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=3
            path = IMAGES_PATH / f"{fig_id}.{fig_extension}"
            if tight_layout:
                plt.tight_layout()
            plt.savefig(path, format=fig_extension, dpi=resolution)
```

First, let's import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures.

Clustering

Introduction – Classification *vs* Clustering

In [5]:
```
# extra code – this cell generates and saves Figure 9–1

import matplotlib.pyplot as plt
from sklearn.datasets import load_iris

data = load_iris()
X = data.data
y = data.target
data.target_names

plt.figure(figsize=(9, 3.5))

plt.subplot(121)
plt.plot(X[y==0, 2], X[y==0, 3], "yo", label="Iris setosa")
plt.plot(X[y==1, 2], X[y==1, 3], "bs", label="Iris versicolor")
plt.plot(X[y==2, 2], X[y==2, 3], "g^", label="Iris virginica")
plt.xlabel("Petal length")
plt.ylabel("Petal width")
plt.grid()
plt.legend()

plt.subplot(122)
plt.scatter(X[:, 2], X[:, 3], c="k", marker=".")
plt.xlabel("Petal length")
plt.tick_params(labelleft=False)
plt.gca().set_axisbelow(True)
plt.grid()

save_fig("classification_vs_clustering_plot")
plt.show()
```



Note: the next cell shows how a Gaussian mixture model (explained later in this chapter) can actually separate these clusters pretty well using all 4 features: petal length & width, and sepal length & width. This code maps each cluster to a class. Instead of hard coding the mapping, the code picks the most common class for each cluster using the `scipy.stats.mode()` function:

K-Means

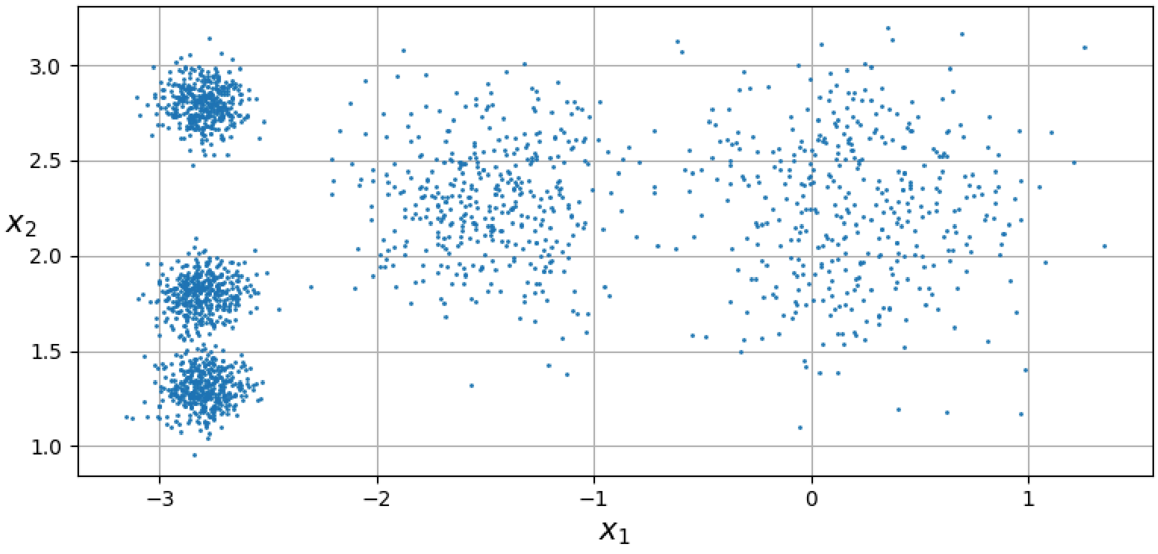
Fit and predict

Let's train a K-Means clusterer on a dataset if blobs. It will try to find each blob's center and assign each instance to the closest blob:

In [6]:
```
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_
                  random_state=7)

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

Now let's plot them:

In [7]:
```
# extra code – this cell generates and saves Figure 9–2

def plot_clusters(X, y=None):
    plt.scatter(X[:, 0], X[:, 1], c=y, s=1)
    plt.xlabel("$x_1$")
    plt.ylabel("$x_2$", rotation=0)

plt.figure(figsize=(8, 4))
plot_clusters(X)
plt.gca().set_axisbelow(True)
plt.grid()
save_fig("blobs_plot")
plt.show()
```



Each instance was assigned to one of the 5 clusters:

In [8]:
```
y_pred
```
Out[8]:
```
array([2, 2, 4, ..., 1, 4, 2], shape=(2000,), dtype=int32)
```
In [9]:
```
y_pred is kmeans.labels_
```
Out[9]:
```
True
```
And the following 5 *centroids* (i.e., cluster centers) were estimated:

In [10]:
```
kmeans.cluster_centers_
```
Out[10]:
```
array([[-0.066884  ,  2.10378803],
       [-2.79290307,  2.79641063],
       [-2.80214068,  1.55162671],
       [-1.47468607,  2.28399066],
       [ 0.47042841,  2.41380533]])
```
Note that the `KMeans` instance preserves the labels of the instances it was trained on. Somewhat confusingly, in this context, the *label* of an instance is the index of the cluster that instance gets assigned to (they are not targets, they are predictions):

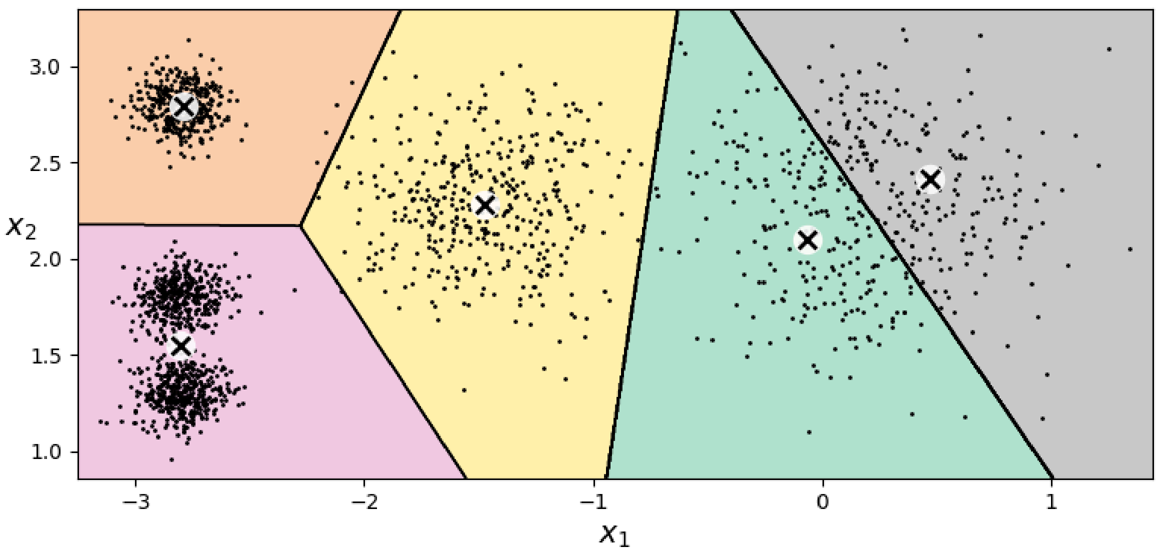
In [11]:
```
kmeans.labels_
```
Out[11]:
```
array([2, 2, 4, ..., 1, 4, 2], shape=(2000,), dtype=int32)
```
Of course, we can predict the labels of new instances:

In [12]:
```
import numpy as np

X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
kmeans.predict(X_new)
```
Out[12]:
```
array([0, 4, 1, 1], dtype=int32)
```
Decision Boundaries

Let's plot the model's decision boundaries. This gives us a *Voronoi diagram*:

In [13]:
```
# extra code – this cell generates and saves Figure 9–3

def plot_data(X):
    plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)

def plot_centroids(centroids, weights=None, circle_color='w', cross_color
    if weights is not None:
        centroids = centroids[weights > weights.max() / 10]
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='o', s=35, linewidths=8,
                color=circle_color, zorder=10, alpha=0.9)
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='x', s=2, linewidths=12,
                color=cross_color, zorder=11, alpha=1)

def plot_decision_boundaries(clusterer, X, resolution=1000, show_centroid
                             show_xlabels=True, show_ylabels=True):
```

```
    mins = X.min(axis=0) - 0.1
    maxs = X.max(axis=0) + 0.1
    xx, yy = np.meshgrid(np.linspace(mins[0], maxs[0], resolution),
                         np.linspace(mins[1], maxs[1], resolution))
    Z = clusterer.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(Z, extent=(mins[0], maxs[0], mins[1], maxs[1]),
                cmap="Pastel2")
    plt.contour(Z, extent=(mins[0], maxs[0], mins[1], maxs[1]),
                linewidths=1, colors='k')
    plot_data(X)
    if show_centroids:
        plot_centroids(clusterer.cluster_centers_)

    if show_xlabels:
        plt.xlabel("$x_1$")
    else:
        plt.tick_params(labelbottom=False)
    if show_ylabels:
        plt.ylabel("$x_2$", rotation=0)
    else:
        plt.tick_params(labelleft=False)

plt.figure(figsize=(8, 4))
plot_decision_boundaries(kmeans, X)
save_fig("voronoi_plot")
plt.show()
```



Not bad! Some of the instances near the edges were probably assigned to the wrong cluster, but overall it looks pretty good.

Hard Clustering *vs* Soft Clustering

Rather than arbitrarily choosing the closest cluster for each instance, which is called *hard clustering*, it might be better to measure the distance of each instance to all 5 centroids. This is what the `transform()` method does:

In [14]:
```
kmeans.transform(X_new).round(2)
```

```
Out[14]: array([[0.12, 2.9 , 2.84, 1.5 , 0.63],
       [3.07, 5.85, 5.82, 4.48, 2.56],
       [3.07, 0.29, 1.46, 1.69, 3.52],
       [2.96, 0.36, 0.97, 1.54, 3.47]])
```

You can verify that this is indeed the Euclidian distance between each instance and each centroid:

```
In [15]: # extra code
np.linalg.norm(np.tile(X_new, (1, k)).reshape(-1, k, 2)
               - kmeans.cluster_centers_, axis=2).round(2)
```

```
Out[15]: array([[0.12, 2.9 , 2.84, 1.5 , 0.63],
       [3.07, 5.85, 5.82, 4.48, 2.56],
       [3.07, 0.29, 1.46, 1.69, 3.52],
       [2.96, 0.36, 0.97, 1.54, 3.47]])
```

The K-Means Algorithm

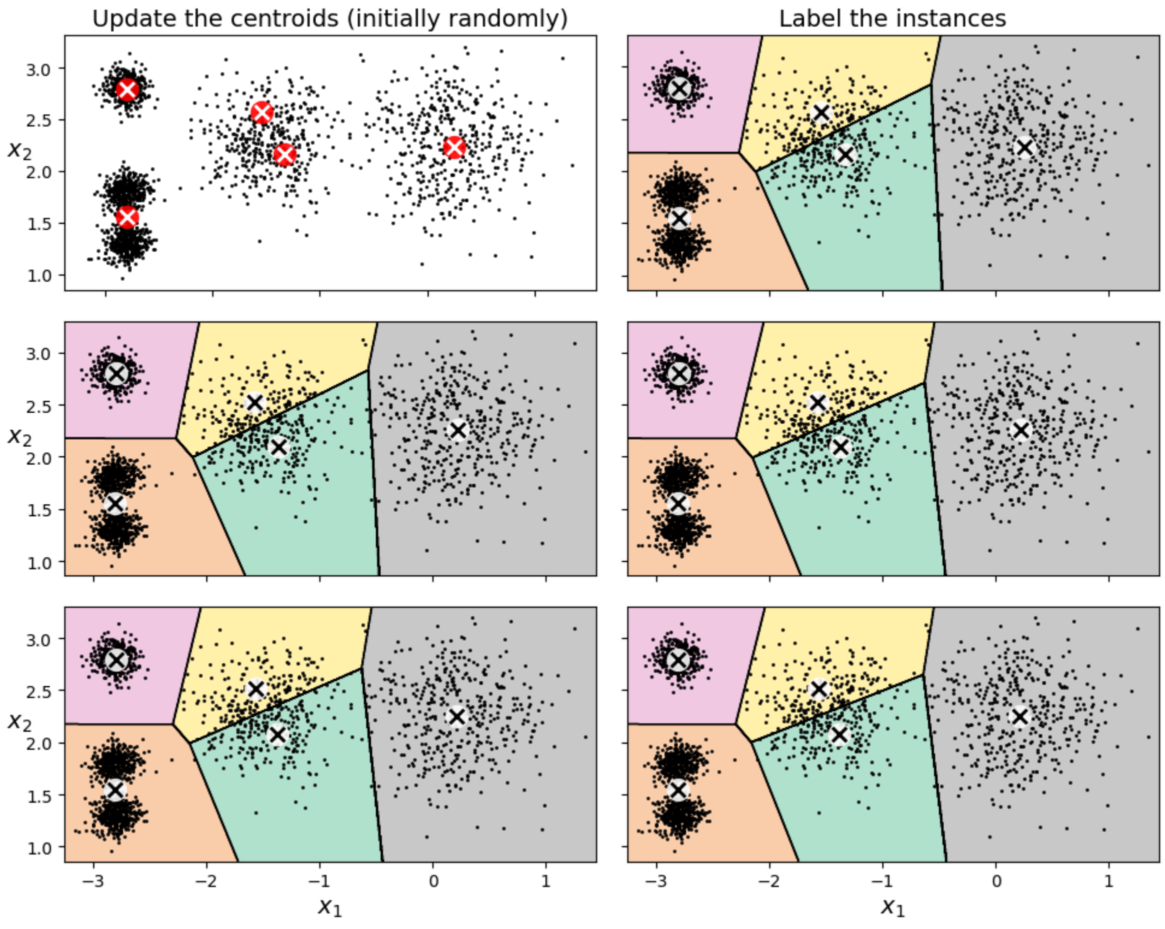
The K-Means algorithm is one of the fastest clustering algorithms, and also one of the simplest:

- First initialize 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 `KMeans` class uses an optimized initialization technique by default. To get the original K-Means algorithm (for educational purposes only), you must set `init="random"` and `n_init=1`. More on this later in this chapter.

Let's run the K-Means algorithm for 1, 2 and 3 iterations, to see how the centroids move around:

```
In [16]: # extra code – this cell generates and saves Figure 9–4

kmeans_iter1 = KMeans(n_clusters=5, init="random", n_init=1, max_iter=1,
                      random_state=5)
kmeans_iter2 = KMeans(n_clusters=5, init="random", n_init=1, max_iter=2,
                      random_state=5)
kmeans_iter3 = KMeans(n_clusters=5, init="random", n_init=1, max_iter=3,
                      random_state=5)
kmeans_iter1.fit(X)
kmeans_iter2.fit(X)
kmeans_iter3.fit(X)

plt.figure(figsize=(10, 8))

plt.subplot(321)
plot_data(X)
plot_centroids(kmeans_iter1.cluster_centers_, circle_color='r', cross_col
plt.ylabel("$x_2$", rotation=0)
plt.tick_params(labelbottom=False)
plt.title("Update the centroids (initially randomly)")

plt.subplot(322)
plot_decision_boundaries(kmeans_iter1, X, show_xlabels=False,
                         show_ylabels=False)
plt.title("Label the instances")

plt.subplot(323)
plot_decision_boundaries(kmeans_iter1, X, show_centroids=False,
                         show_xlabels=False)
plot_centroids(kmeans_iter2.cluster_centers_)

plt.subplot(324)
plot_decision_boundaries(kmeans_iter2, X, show_xlabels=False,
                         show_ylabels=False)

plt.subplot(325)
plot_decision_boundaries(kmeans_iter2, X, show_centroids=False)
plot_centroids(kmeans_iter3.cluster_centers_)

plt.subplot(326)
plot_decision_boundaries(kmeans_iter3, X, show_ylabels=False)

save_fig("kmeans_algorithm_plot")
plt.show()
```

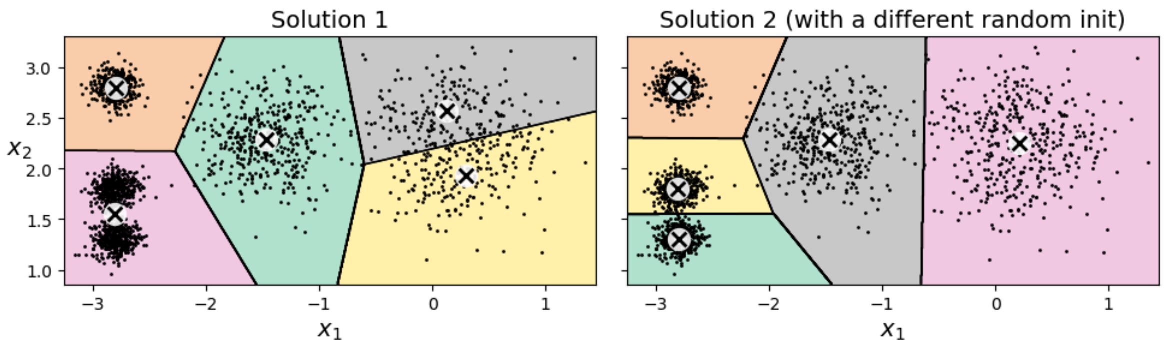


K-Means Variability

In the original K-Means algorithm, the centroids are just initialized randomly, and the algorithm simply runs a single iteration to gradually improve the centroids, as we saw above.

However, one major problem with this approach is that if you run K-Means multiple times (or with different random seeds), it can converge to very different solutions, as you can see below:

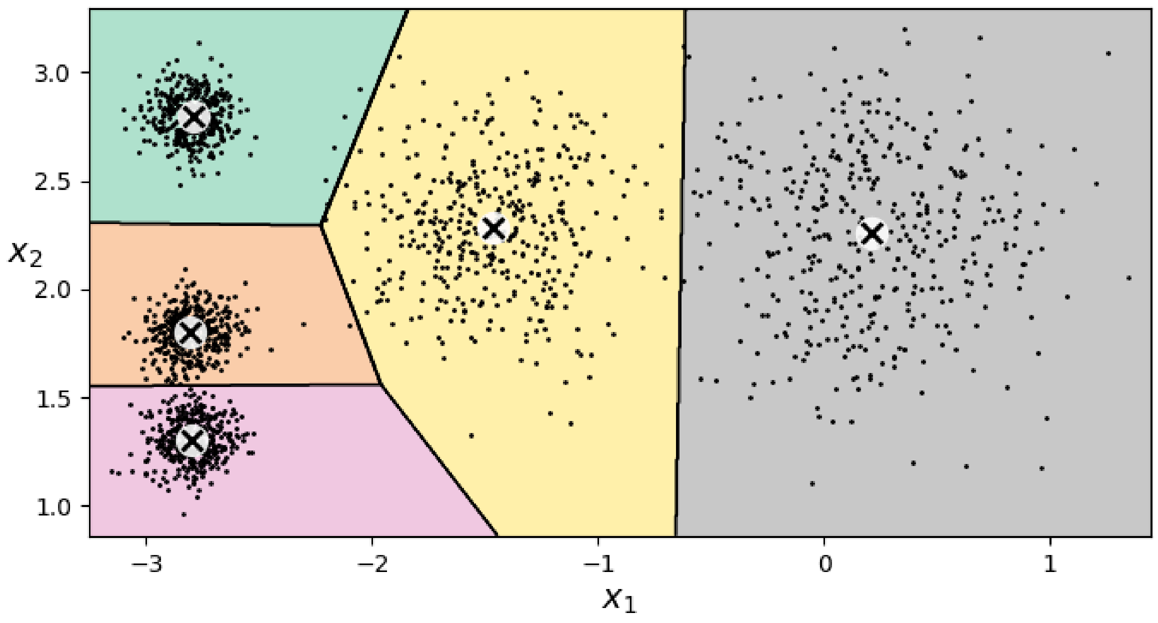
In [17]:
```
# extra code – this cell generates and saves Figure 9–5

def plot_clusterer_comparison(clusterer1, clusterer2, X, title1=None,
                              title2=None):
    clusterer1.fit(X)
    clusterer2.fit(X)

    plt.figure(figsize=(10, 3.2))

    plt.subplot(121)
    plot_decision_boundaries(clusterer1, X)
    if title1:
        plt.title(title1)

    plt.subplot(122)
    plot_decision_boundaries(clusterer2, X, show_ylabels=False)
    if title2:
        plt.title(title2)

kmeans_rnd_init1 = KMeans(n_clusters=5, init="random", n_init=1, random_s
kmeans_rnd_init2 = KMeans(n_clusters=5, init="random", n_init=1, random_s

plot_clusterer_comparison(kmeans_rnd_init1, kmeans_rnd_init2, X,
                          "Solution 1",
                          "Solution 2 (with a different random init)")

save_fig("kmeans_variability_plot")
plt.show()
```



In [18]:
```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1, random_state=42)
kmeans.fit(X)
```

Out[18]:
```
▾ KMeans ⓘ ?
▶ Parameters
```

In [19]:
```
# extra code
plt.figure(figsize=(8, 4))
plot_decision_boundaries(kmeans, X)
```



Inertia

To select the best model, we will need a way to evaluate a K-Mean model's performance. Unfortunately, clustering is an unsupervised task, so we do not have the targets. But at least we can measure the distance between each instance and its centroid. This is the idea behind the *inertia* metric:

```
In [20]: kmeans.inertia_
```

```
Out[20]: 211.5985372581684
```

```
In [21]: kmeans_rnd_init1.inertia_  # extra code
```

```
Out[21]: 219.58201503602288
```

```
In [22]: kmeans_rnd_init2.inertia_  # extra code
```

```
Out[22]: 211.59853725816836
```

As you can easily verify, inertia is the sum of the squared distances between each training instance and its closest centroid:

```
In [23]: # extra code
         X_dist = kmeans.transform(X)
         (X_dist[np.arange(len(X_dist)), kmeans.labels_] ** 2).sum()
```

```
Out[23]: np.float64(211.59853725816862)
```

The `score()` method returns the negative inertia. Why negative? Well, it is because a predictor's `score()` method must always respect the "*greater is better*" rule.

```
In [24]: kmeans.score(X)
```

```
Out[24]: -211.59853725816836
```

Multiple Initializations

So one approach to solve the variability issue is to simply run the K-Means algorithm multiple times with different random initializations, and select the solution that minimizes the inertia.

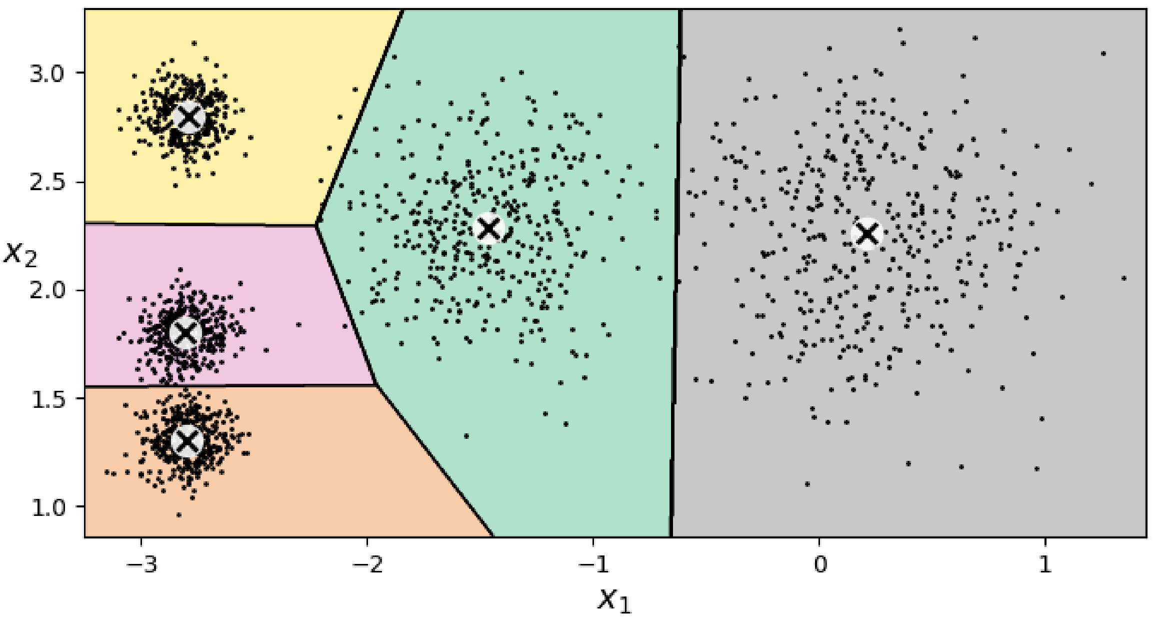
When you set the `n_init` hyperparameter, Scikit-Learn runs the original algorithm `n_init` times, and selects the solution that minimizes the inertia. By default, Scikit-Learn sets `n_init=10`.

In [25]:
```
# extra code
kmeans_rnd_10_inits = KMeans(n_clusters=5, init="random", n_init=10,
                             random_state=2)
kmeans_rnd_10_inits.fit(X)
```

Out[25]:
```
▾ KMeans  ⓘ ?

▶ Parameters
```

As you can see, we end up with the initial model, which is certainly the optimal K-Means solution (at least in terms of inertia, and assuming $k = 5$).

In [26]:
```
# extra code
plt.figure(figsize=(8, 4))
plot_decision_boundaries(kmeans_rnd_10_inits, X)
plt.show()
```



In [27]:
```
kmeans_rnd_10_inits.inertia_
```

Out[27]:
```
211.59853725816836
```

Centroid initialization methods

Instead of initializing the centroids entirely randomly, it is preferable to initialize them using the following algorithm, proposed in a 2006 paper by David Arthur and Sergei Vassilvitskii:

- Take one centroid c_1, chosen uniformly at random from the dataset.
- Take a new center c_i, choosing an instance $\mathbf{x}_i$ with probability: $D(\mathbf{x}_i)^2$ / $\sum\limits_{j=1}^{m} D(\mathbf{x}_j)^2$ where $D(\mathbf{x}_i)$ is the distance between the instance $\mathbf{x}_i$ and the closest centroid that was already chosen. This probability distribution ensures that instances that are further away from already chosen centroids are much more likely be selected as centroids.
- Repeat the previous step until all k centroids have been chosen.

The rest of the K-Means++ algorithm is just regular K-Means. With this initialization, the K-Means algorithm is much less likely to converge to a suboptimal solution, so it is possible to reduce `n_init` considerably. Most of the time, this largely compensates for the additional complexity of the initialization process.

To set the initialization to K-Means++, simply set `init="k-means++"` (this is actually the default):

Accelerated K-Means

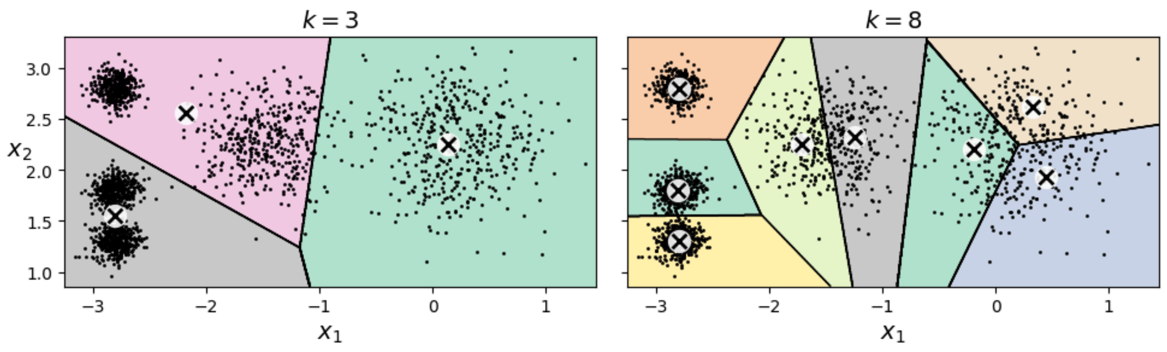
The K-Means algorithm can sometimes be accelerated by avoiding many unnecessary distance calculations: this is achieved by exploiting the triangle inequality (given three points A, B and C, the distance AC is always such that AC ≤ AB + BC) and by keeping track of lower and upper bounds for distances between instances and centroids (see this 2003 paper by Charles Elkan for more details).

For Elkan's variant of K-Means, use `algorithm="elkan"`. For regular KMeans, use `algorithm="full"`. The default is `"auto"`, which uses the full algorithm since Scikit-Learn 1.1 (it used Elkan's algorithm before that).

Finding the optimal number of clusters

What if the number of clusters was set to a lower or greater value than 5?

In [28]:
```
# extra code – this cell generates and saves Figure 9–7

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

plot_clusterer_comparison(kmeans_k3, kmeans_k8, X, "$k=3$", "$k=8$")
save_fig("bad_n_clusters_plot")
plt.show()
```



Ouch, these two models don't look great. What about their inertias?

In [29]:
```python
kmeans_k3.inertia_
```

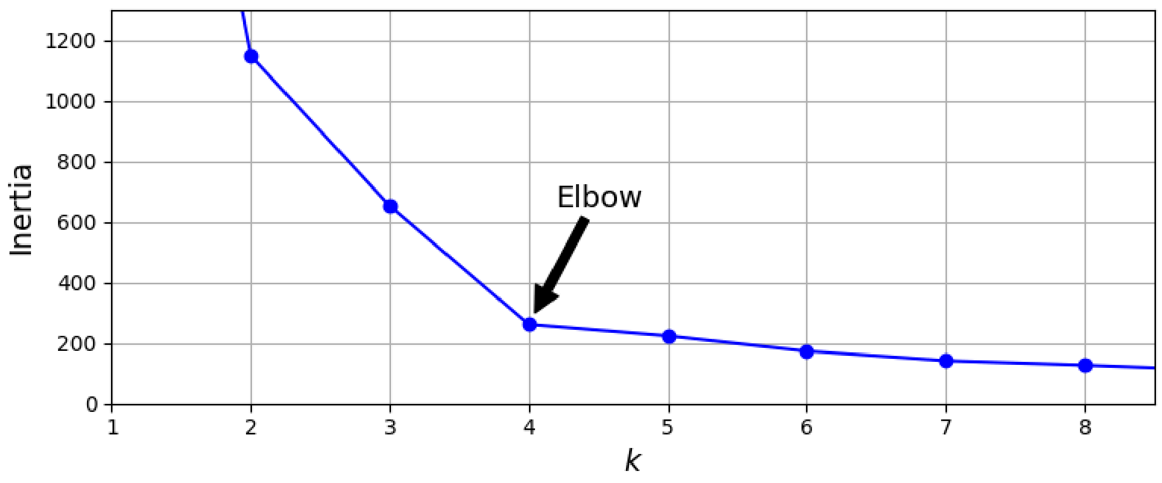
Out[29]:
```
653.2167190021551
```

In [30]:
```python
kmeans_k8.inertia_
```

Out[30]:
```
127.1314188046183
```

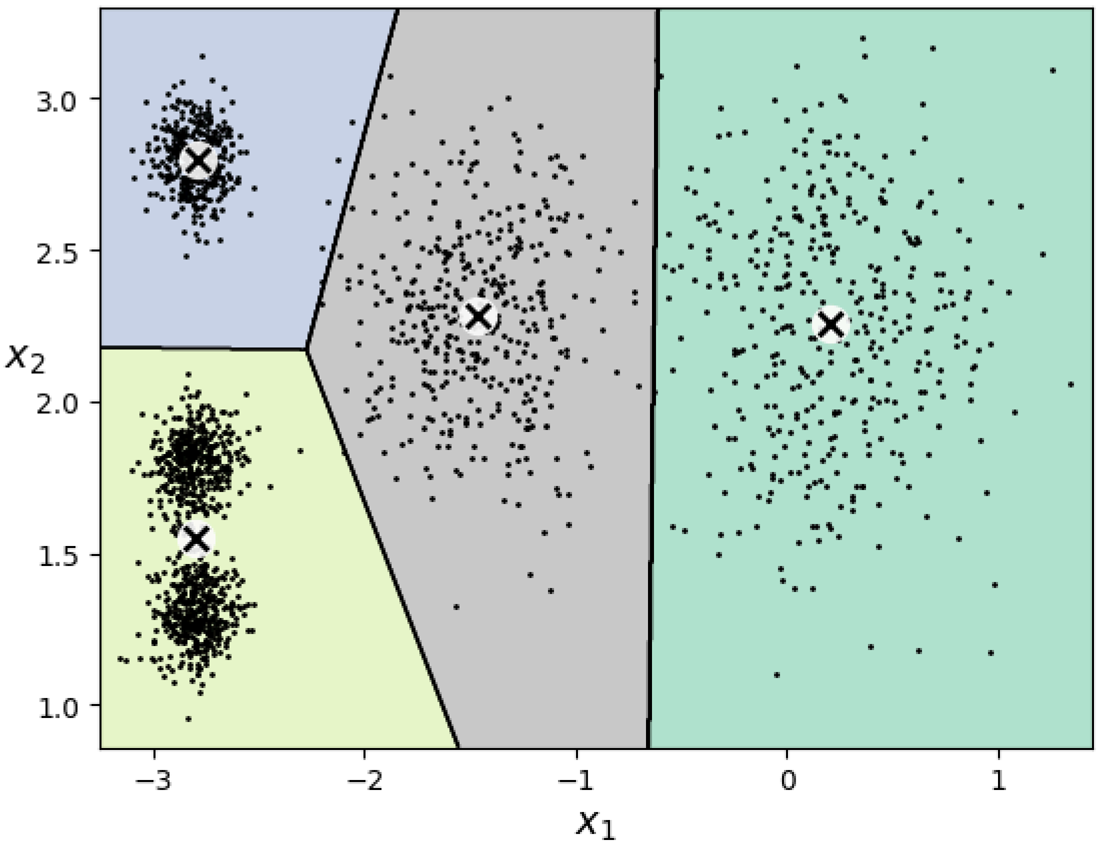
No, we cannot simply take the value of $k$ that minimizes the inertia, since it keeps getting lower as we increase $k$. Indeed, the more clusters there are, the closer each instance will be to its closest centroid, and therefore the lower the inertia will be. However, we can plot the inertia as a function of $k$ and analyze the resulting curve:

In [31]:
```python
# extra code – this cell generates and saves Figure 9–8

kmeans_per_k = [KMeans(n_clusters=k, random_state=42).fit(X)
                for k in range(1, 10)]
inertias = [model.inertia_ for model in kmeans_per_k]

plt.figure(figsize=(8, 3.5))
plt.plot(range(1, 10), inertias, "bo-")
plt.xlabel("$k$")
plt.ylabel("Inertia")
plt.annotate("", xy=(4, inertias[3]), xytext=(4.45, 650),
             arrowprops=dict(facecolor='black', shrink=0.1))
plt.text(4.5, 650, "Elbow", horizontalalignment="center")
plt.axis([1, 8.5, 0, 1300])
plt.grid()
save_fig("inertia_vs_k_plot")
plt.show()
```



As you can see, there is an elbow at $k = 4$, which means that less clusters than that would be bad, and more clusters would not help much and might cut clusters in half. So $k = 4$ is a pretty good choice. Of course in this example it is not perfect since it means that the two blobs in the lower left will be considered as just a single cluster, but it's a pretty good clustering nonetheless.

In [32]:
```
# extra code
plot_decision_boundaries(kmeans_per_k[4 - 1], X)
plt.show()
```



Another approach is to look at the *silhouette score*, which is the mean *silhouette coefficient* over all the instances. An instance's silhouette coefficient is equal to (*b* - *a*) / max(*a*, *b*) where *a* is the mean distance to the other instances in the same cluster (it is the *mean intra-cluster distance*), and *b* is the *mean nearest-cluster distance*, that is the mean distance to the instances of the next closest cluster (defined as the one that minimizes *b*, excluding the instance's own cluster). The silhouette coefficient can vary between -1 and +1: a coefficient close to +1 means that the instance is well inside its own cluster and far from other clusters, while a coefficient close to 0 means that it is close to a cluster boundary, and finally a coefficient close to -1 means that the instance may have been assigned to the wrong cluster.

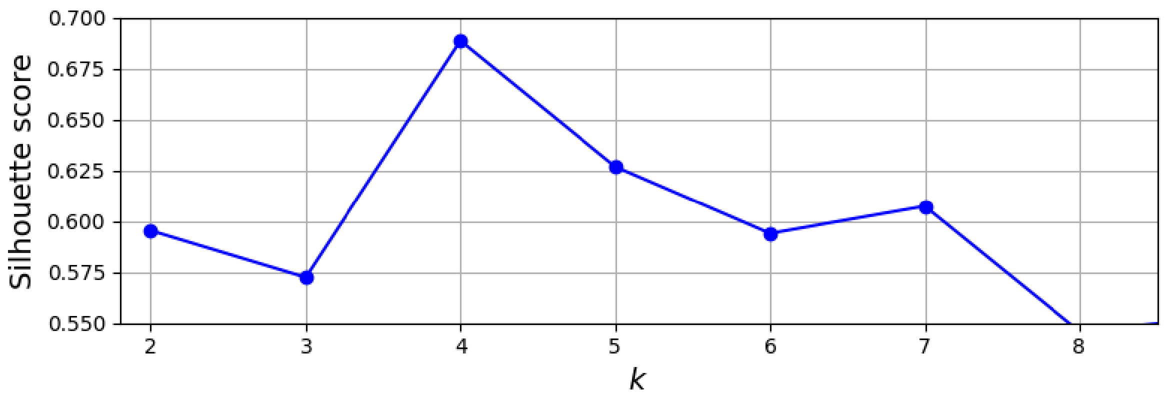
Let's plot the silhouette score as a function of $k$:

In [33]:
```
from sklearn.metrics import silhouette_score
```

In [34]:
```
silhouette_score(X, kmeans.labels_)
```

Out[34]:
```
0.655517642572828
```

In [35]:
```
# extra code – this cell generates and saves Figure 9–9

silhouette_scores = [silhouette_score(X, model.labels_)
                     for model in kmeans_per_k[1:]]

plt.figure(figsize=(8, 3))
plt.plot(range(2, 10), silhouette_scores, "bo-")
plt.xlabel("$k$")
plt.ylabel("Silhouette score")
plt.axis([1.8, 8.5, 0.55, 0.7])
plt.grid()
save_fig("silhouette_score_vs_k_plot")
plt.show()
```

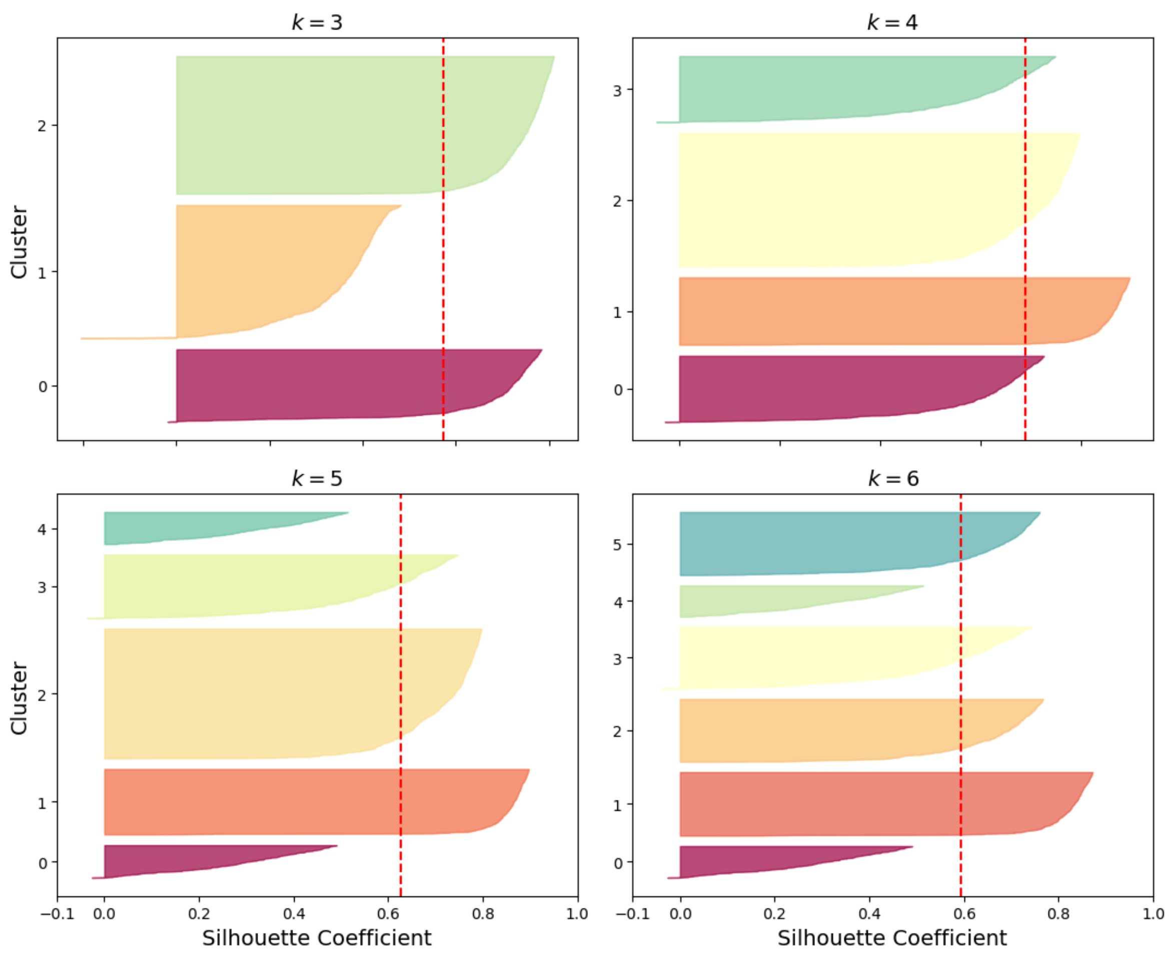


As you can see, this visualization is much richer than the previous one: in particular, although it confirms that $k = 4$ is a very good choice, but it also underlines the fact that $k = 5$ is quite good as well.

An even more informative visualization is given when you plot every instance's silhouette coefficient, sorted by the cluster they are assigned to and by the value of the coefficient. This is called a *silhouette diagram*:

In [36]:
```
# extra code – this cell generates and saves Figure 9–10

from sklearn.metrics import silhouette_samples
from matplotlib.ticker import FixedLocator, FixedFormatter

plt.figure(figsize=(11, 9))
```

```python
for k in (3, 4, 5, 6):
    plt.subplot(2, 2, k - 2)

    y_pred = kmeans_per_k[k - 1].labels_
    silhouette_coefficients = silhouette_samples(X, y_pred)

    padding = len(X) // 30
    pos = padding
    ticks = []
    for i in range(k):
        coeffs = silhouette_coefficients[y_pred == i]
        coeffs.sort()

        color = plt.cm.Spectral(i / k)
        plt.fill_betweenx(np.arange(pos, pos + len(coeffs)), 0, coeffs,
                          facecolor=color, edgecolor=color, alpha=0.7)
        ticks.append(pos + len(coeffs) // 2)
        pos += len(coeffs) + padding

    plt.gca().yaxis.set_major_locator(FixedLocator(ticks))
    plt.gca().yaxis.set_major_formatter(FixedFormatter(range(k)))
    if k in (3, 5):
        plt.ylabel("Cluster")

    if k in (5, 6):
        plt.gca().set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])
        plt.xlabel("Silhouette Coefficient")
    else:
        plt.tick_params(labelbottom=False)

    plt.axvline(x=silhouette_scores[k - 2], color="red", linestyle="--")
    plt.title(f"$k={k}$")

save_fig("silhouette_analysis_plot")
plt.show()
```

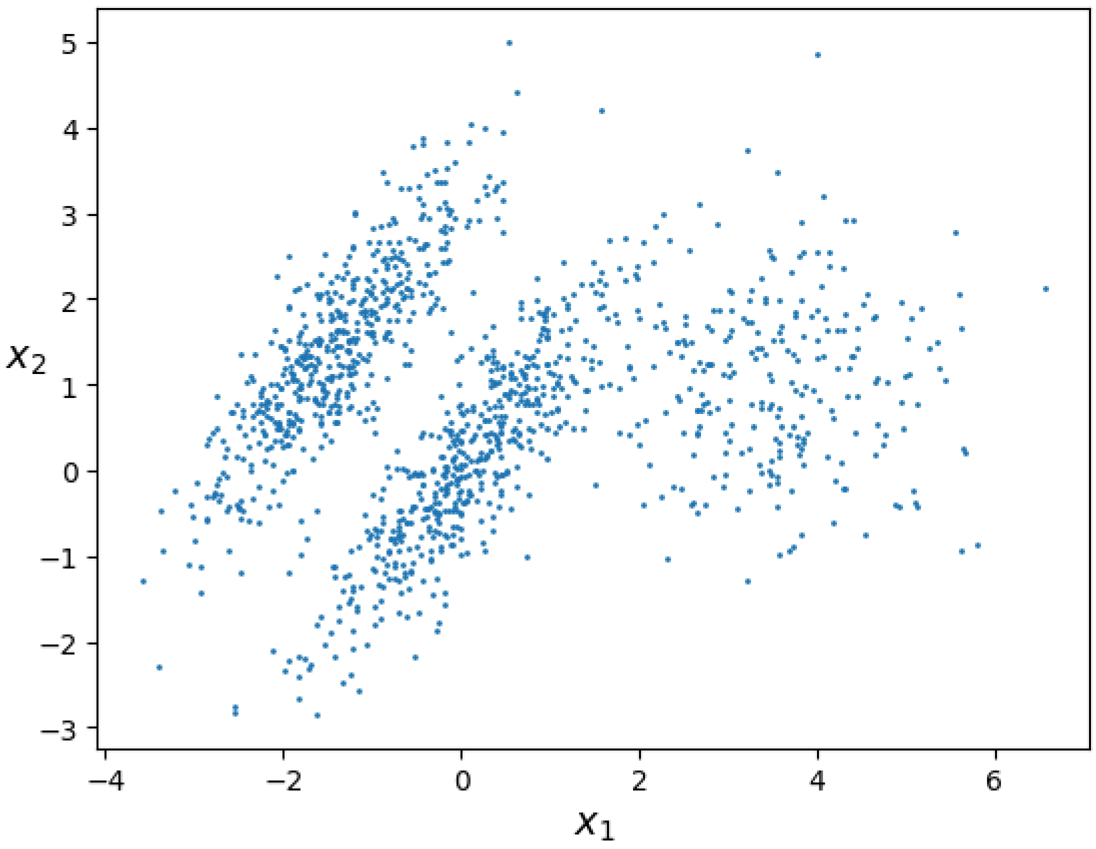


As you can see, $k = 5$ looks like the best option here, as all clusters are roughly the same size, and they all cross the dashed line, which represents the mean silhouette score.

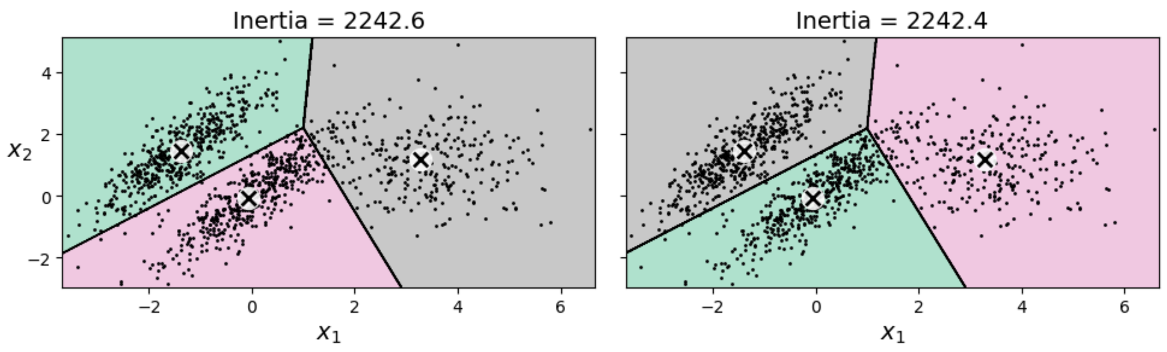
Limits of K-Means

Let's generate a more difficult dataset, with elongated blobs and varying densities, and show that K-Means struggles to cluster it correctly:

In [37]:
```
# extra code – this cell generates and saves Figure 9–11

X1, y1 = make_blobs(n_samples=1000, centers=((4, -4), (0, 0)), random_sta
X1 = X1.dot(np.array([[0.374, 0.95], [0.732, 0.598]]))
X2, y2 = make_blobs(n_samples=250, centers=1, random_state=42)
X2 = X2 + [6, -8]
X = np.r_[X1, X2]
y = np.r_[y1, y2]

plot_clusters(X)
```



```
In [38]: kmeans_good = KMeans(n_clusters=3,
                              init=np.array([[-1.5, 2.5], [0.5, 0], [4, 0]]),
                              n_init=1, random_state=42)
         kmeans_bad = KMeans(n_clusters=3, random_state=42)
         kmeans_good.fit(X)
         kmeans_bad.fit(X)

         plt.figure(figsize=(10, 3.2))

         plt.subplot(121)
         plot_decision_boundaries(kmeans_good, X)
         plt.title(f"Inertia = {kmeans_good.inertia_:.1f}")

         plt.subplot(122)
         plot_decision_boundaries(kmeans_bad, X, show_ylabels=False)
         plt.title(f"Inertia = {kmeans_bad.inertia_:.1f}")

         save_fig("bad_kmeans_plot")
         plt.show()
```



Using Clustering for Image Segmentation

Download the ladybug image:

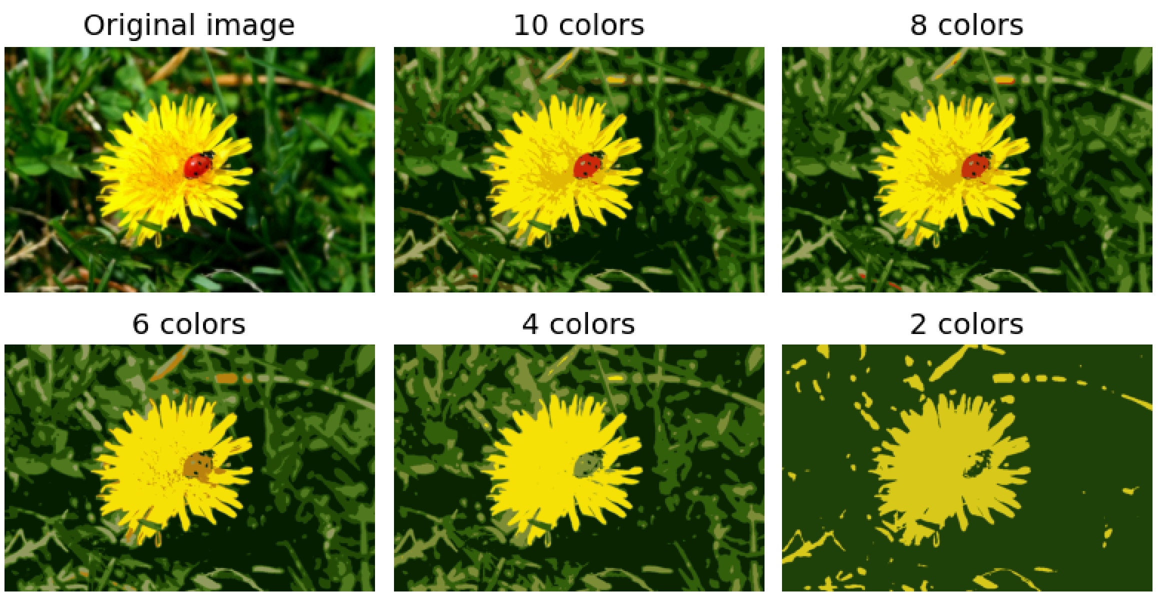
```
In [39]: # extra code – downloads the ladybug image

         import urllib.request

         homl3_root = "https://github.com/ageron/handson-ml3/raw/main/"
         filename = "ladybug.png"
         filepath = IMAGES_PATH / filename
         if not filepath.is_file():
             print("Downloading", filename)
             url = f"{homl3_root}/images/unsupervised_learning/{filename}"
             urllib.request.urlretrieve(url, filepath)
```

```
In [40]: import PIL

         image = np.asarray(PIL.Image.open(filepath))
         image.shape
```

```
Out[40]: (533, 800, 3)
```

```
In [41]: 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)
```

```
In [42]: # extra code – this cell generates and saves Figure 9–12

         segmented_imgs = []
         n_colors = (10, 8, 6, 4, 2)
         for n_clusters in n_colors:
             kmeans = KMeans(n_clusters=n_clusters, random_state=42).fit(X)
             segmented_img = kmeans.cluster_centers_[kmeans.labels_]
             segmented_imgs.append(segmented_img.reshape(image.shape))

         plt.figure(figsize=(10, 5))
         plt.subplots_adjust(wspace=0.05, hspace=0.1)

         plt.subplot(2, 3, 1)
         plt.imshow(image)
         plt.title("Original image")
         plt.axis('off')

         for idx, n_clusters in enumerate(n_colors):
             plt.subplot(2, 3, 2 + idx)
             plt.imshow(segmented_imgs[idx] / 255)
             plt.title(f"{n_clusters} colors")
             plt.axis('off')

         save_fig('image_segmentation_plot', tight_layout=False)
         plt.show()
```



Using Clustering for Preprocessing

Let's tackle the *digits dataset* which is a simple MNIST-like dataset containing 1,797 grayscale 8×8 images representing digits 0 to 9.

```
In [43]: from sklearn.datasets import load_digits
```

```
In [44]: X_digits, y_digits = load_digits(return_X_y=True)
```

Let's split it into a training set and a test set:

```
In [45]: from sklearn.model_selection import train_test_split
```

```
In [46]: X_train, X_test, y_train, y_test = train_test_split(X_digits, y_digits, r
```

Now let's fit a Logistic Regression model and evaluate it on the test set:

```
In [47]: from sklearn.linear_model import LogisticRegression
```

```
In [48]: log_reg = LogisticRegression(max_iter=10000, random_state=42)
         log_reg.fit(X_train, y_train)
```

```
Out[48]: ▾ LogisticRegression ⓘ ?
         ▶ Parameters
```

```
In [49]: log_reg.score(X_test, y_test)
```

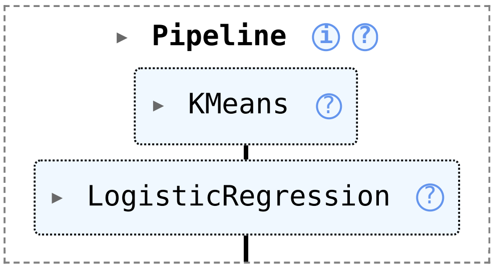
```
Out[49]: 0.9733333333333334
```

Okay, that's our baseline: 97.3% accuracy. Let's see if we can do better by using K-Means as a preprocessing step. We will create a pipeline that will first cluster the training set into 50 clusters and replace the images with their distances to the 50 clusters, then apply a logistic regression model:

```
In [50]: from sklearn.pipeline import Pipeline
```

```
In [51]: pipeline = Pipeline([
             ("kmeans", KMeans(n_clusters=50, random_state=42)),
             ("log_reg", LogisticRegression(max_iter=10000, random_state=42)),
         ])
         pipeline.fit(X_train, y_train)
```

Out[51]:



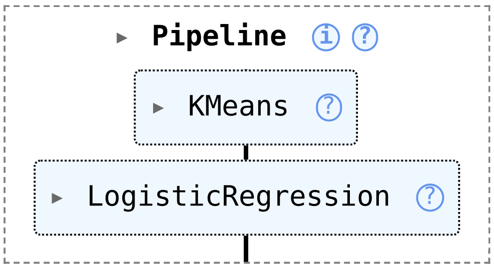
```
In [52]: pipeline.score(X_test, y_test)
```

```
Out[52]: 0.9711111111111111
```

- With 50 clusters we are not yet as good as the original logistic regression
- If we increase the number of clusters, say 88, we can get better results than logistic regression

```
In [53]: pipeline = Pipeline([
             ("kmeans", KMeans(n_clusters=88, random_state=42)),
             ("log_reg", LogisticRegression(max_iter=10000, random_state=42)),
         ])
         pipeline.fit(X_train, y_train)
```

Out[53]:



```
In [54]: pipeline.score(X_test, y_test)
```

```
Out[54]: 0.9822222222222222
```

Using Clustering for Semi-Supervised Learning

Another use case for clustering is semi-supervised learning, when we have plenty of unlabeled instances and very few labeled instances.

Let's tackle the *digits dataset* which is a simple MNIST-like dataset containing 1,797 grayscale 8×8 images representing digits 0 to 9.

In [55]:
```python
from sklearn.datasets import load_digits

X_digits, y_digits = load_digits(return_X_y=True)
```

In [56]:
```python
X_train, X_test, y_train, y_test = train_test_split(X_digits, y_digits, r
```

Let's look at the performance of a logistic regression model when we only have 50 labeled instances:

In [57]:
```python
from sklearn.linear_model import LogisticRegression

log_reg_full = LogisticRegression(max_iter=10_000, random_state=42)
log_reg_full.fit(X_train, y_train)
log_reg_full.score(X_test, y_test)
```

Out[57]:
```
0.9733333333333334
```

In [58]:
```python
n_labeled = 50
log_reg = LogisticRegression(max_iter=10_000, random_state=42)
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

Out[58]:
```
▾ LogisticRegression ⓘ ?

▶ Parameters
```

In [59]:
```python
log_reg.score(X_test, y_test)
```

Out[59]:
```
0.8266666666666667
```

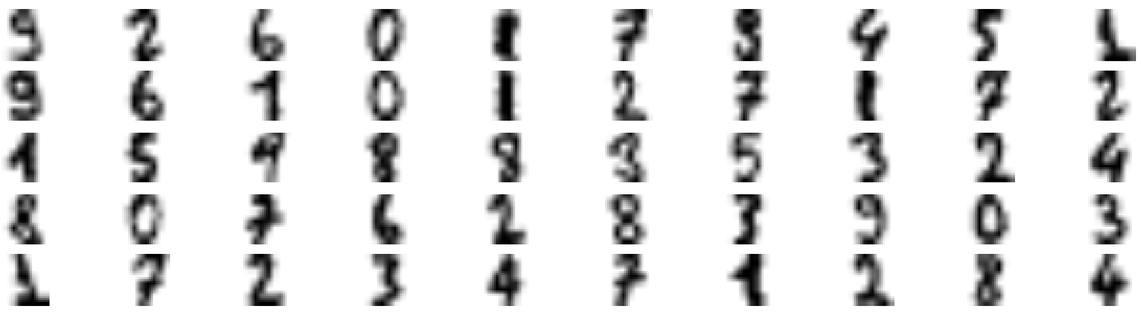
It's much less than earlier of course. Let's see how we can do better. First, let's cluster the training set into 50 clusters, then for each cluster let's find the image closest to the centroid. We will call these images the representative images:

In [60]:
```python
k = 50
kmeans = KMeans(n_clusters=k, random_state=42)
X_digits_dist = kmeans.fit_transform(X_train)
representative_digit_idx = X_digits_dist.argmin(axis=0)
X_representative_digits = X_train[representative_digit_idx]
```

Now let's plot these representative images and label them manually:

In [61]:
```python
# extra code – this cell generates and saves Figure 9–13

plt.figure(figsize=(8, 2))
for index, X_representative_digit in enumerate(X_representative_digits):
    plt.subplot(k // 10, 10, index + 1)
    plt.imshow(X_representative_digit.reshape(8, 8), cmap="binary",
               interpolation="bilinear")
    plt.axis('off')
```

```
save_fig("representative_images_plot", tight_layout=False)
plt.show()
```



In [62]:
```
y_representative_digits = np.array([
    9, 2, 6, 0, 1, 7, 3, 4, 5, 1,
    9, 6, 1, 0, 1, 2, 7, 1, 7, 2,
    1, 5, 9, 8, 8, 3, 5, 3, 2, 4,
    8, 0, 7, 6, 2, 8, 3, 9, 0, 3,
    1, 7, 2, 3, 4, 7, 1, 2, 8, 4
])
```

Now we have a dataset with just 50 labeled instances, but instead of being completely random instances, each of them is a representative image of its cluster. Let's see if the performance is any better:

In [63]:
```
log_reg = LogisticRegression(max_iter=10_000, random_state=42)
log_reg.fit(X_representative_digits, y_representative_digits)
log_reg.score(X_test, y_test)
```

Out[63]:
```
0.9288888888888889
```

Wow! We jumped from 82.7% accuracy to 92.9%, although we are still only training the model on 50 instances. Since it's often costly and painful to label instances, especially when it has to be done manually by experts, it's a good idea to make them label representative instances rather than just random instances.

But perhaps we can go one step further: what if we propagated the labels to all the other instances in the same cluster?

In [64]:
```
y_train_propagated = np.empty(len(X_train), dtype=np.int64)
for i in range(k):
    y_train_propagated[kmeans.labels_ == i] = y_representative_digits[i]
```

In [65]:
```
log_reg = LogisticRegression(max_iter=10_000, random_state=42)
log_reg.fit(X_train, y_train_propagated)
```

Out[65]:
```
▾ LogisticRegression
▶ Parameters
```

In [66]:
```
log_reg.score(X_test, y_test)
```

Out[66]:
```
0.9377777777777778
```

We got another significant accuracy boost! Let's see if we can do even better by ignoring the 75% instances that are farthest from their cluster center: this should eliminate some outliers:

In [67]:
```python
percentile_closest = 25

X_cluster_dist = X_digits_dist[np.arange(len(X_train)), kmeans.labels_]
for i in range(k):
    in_cluster = (kmeans.labels_ == i)
    cluster_dist = X_cluster_dist[in_cluster]
    cutoff_distance = np.percentile(cluster_dist, percentile_closest)
    above_cutoff = (X_cluster_dist > cutoff_distance)
    X_cluster_dist[in_cluster & above_cutoff] = -1

partially_propagated = (X_cluster_dist != -1)
X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]
```

In [68]:
```python
log_reg = LogisticRegression(max_iter=10_000, random_state=42)
log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
log_reg.score(X_test, y_test)
```

Out[68]:
```
0.9422222222222222
```

Our propagated labels are actually pretty good: their accuracy is about 99.4%:

In [69]:
```python
(y_train_partially_propagated == y_train[partially_propagated]).mean()
```

Out[69]:
```
np.float64(0.9943820224719101)
```

You could now do a few iterations of *active learning*:

1. Manually label the instances that the classifier is least sure about, if possible by picking them in distinct clusters.
2. Train a new model with these additional labels.

DBSCAN

In [70]:
```python
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)
```

Out[70]:
```
▾ DBSCAN ⓘ ?

▶ Parameters
```

In [71]:
```python
dbscan.labels_[:10]
```

```
Out[71]:  array([ 0,  2, -1, -1,  1,  0,  0,  0,  2,  5])
```

```
In [72]:  dbscan.core_sample_indices_[:10]
```

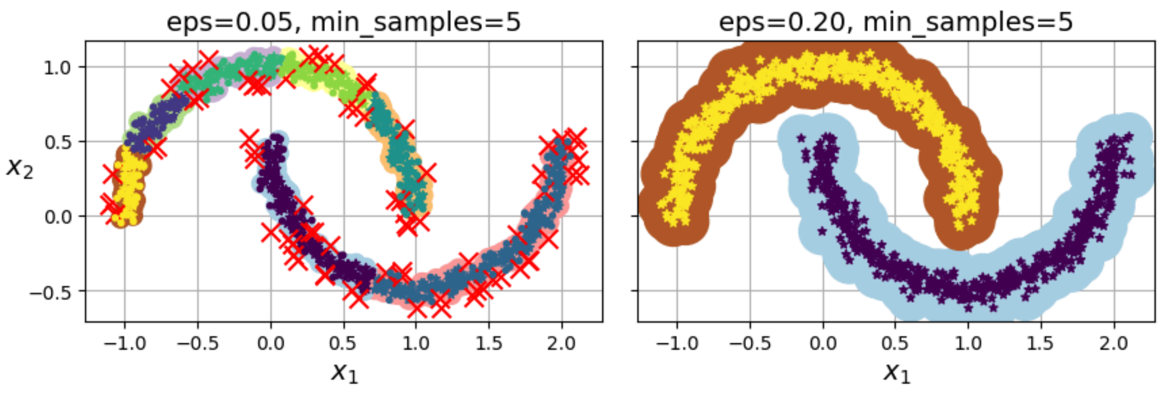
```
Out[72]:  array([ 0,  4,  5,  6,  7,  8, 10, 11, 12, 13])
```

```
In [73]:  dbscan.components_
```

```
Out[73]:  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]], shape=(808, 2))
```

```
In [74]:  # extra code – this cell generates and saves Figure 9–14

def plot_dbscan(dbscan, X, size, show_xlabels=True, show_ylabels=True):
    core_mask = np.zeros_like(dbscan.labels_, dtype=bool)
    core_mask[dbscan.core_sample_indices_] = True
    anomalies_mask = dbscan.labels_ == -1
    non_core_mask = ~(core_mask | anomalies_mask)

    cores = dbscan.components_
    anomalies = X[anomalies_mask]
    non_cores = X[non_core_mask]

    plt.scatter(cores[:, 0], cores[:, 1],
                c=dbscan.labels_[core_mask], marker='o', s=size, cmap="Pa
    plt.scatter(cores[:, 0], cores[:, 1], marker='*', s=20,
                c=dbscan.labels_[core_mask])
    plt.scatter(anomalies[:, 0], anomalies[:, 1],
                c="r", marker="x", s=100)
    plt.scatter(non_cores[:, 0], non_cores[:, 1],
                c=dbscan.labels_[non_core_mask], marker=".")
    if show_xlabels:
        plt.xlabel("$x_1$")
    else:
        plt.tick_params(labelbottom=False)
    if show_ylabels:
        plt.ylabel("$x_2$", rotation=0)
    else:
        plt.tick_params(labelleft=False)
    plt.title(f"eps={dbscan.eps:.2f}, min_samples={dbscan.min_samples}")
    plt.grid()
    plt.gca().set_axisbelow(True)

dbscan2 = DBSCAN(eps=0.2)
dbscan2.fit(X)

plt.figure(figsize=(9, 3.2))

plt.subplot(121)
plot_dbscan(dbscan, X, size=100)

plt.subplot(122)
plot_dbscan(dbscan2, X, size=600, show_ylabels=False)
```

```
save_fig("dbscan_plot")
plt.show()
```


eps=0.05, min_samples=5
eps=0.20, min_samples=5
x_1
x_2
x_1