Chapter 9 – Unsupervised Learning

This notebook contains all the sample code in chapter 9.

Run in Google Colab (https://colab.research.google.com/github/ageron/handson-ml2/blob/master/09_unsupervised_learning.ipynb)

Setup

First, let's import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures. We also check that Python 3.5 or later is installed (although Python 2.x may work, it is deprecated so we strongly recommend you use Python 3 instead), as well as Scikit-Learn ≥0.20.

In [1]:
```
# Python ≥3.5 is required
import sys
assert sys.version_info >= (3, 5)

# Scikit-Learn ≥0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"

# Common imports
import numpy as np
import os

# to make this notebook's output stable across runs
np.random.seed(42)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "unsupervised_learning"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=30
0):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)

# Ignore useless warnings (see SciPy issue #5998)
import warnings
warnings.filterwarnings(action="ignore", message="^internal gelsd")
```

Clustering

Introduction – Classification *vs* Clustering

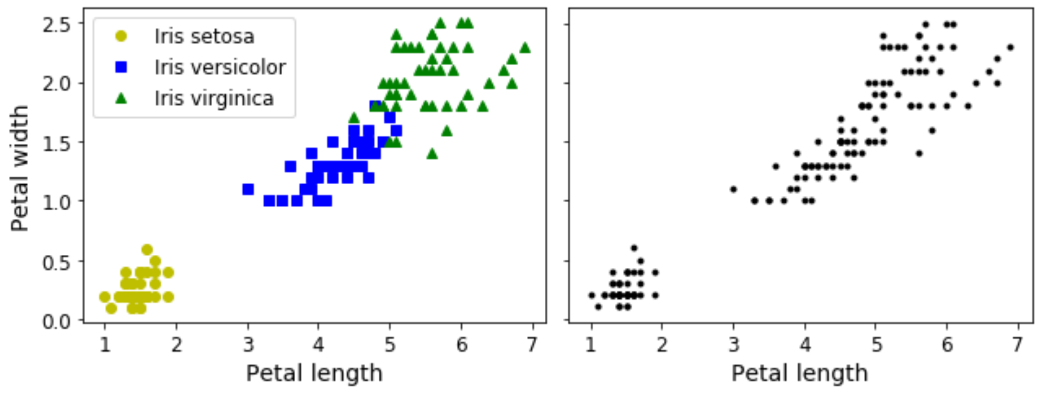
In [2]:
```
from sklearn.datasets import load_iris
```

In [3]:
```
data = load_iris()
X = data.data
y = data.target
data.target_names
```

Out[3]:
```
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

In [4]:
```
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", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(fontsize=12)

plt.subplot(122)
plt.scatter(X[:, 2], X[:, 3], c="k", marker=".")
plt.xlabel("Petal length", fontsize=14)
plt.tick_params(labelleft=False)

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

```
Saving figure classification_vs_clustering_plot
```

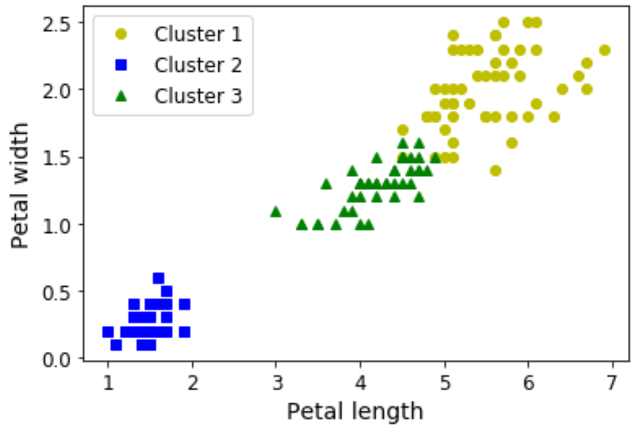


A Gaussian mixture model (explained below) can actually separate these clusters pretty well (using all 4 features: petal length & width, and sepal length & width).

In [5]:
```
from sklearn.mixture import GaussianMixture
```

In [6]:
```
y_pred = GaussianMixture(n_components=3, random_state=42).fit(X).predict(X)
mapping = np.array([2, 0, 1])
y_pred = np.array([mapping[cluster_id] for cluster_id in y_pred])
```

In [7]:
```
plt.plot(X[y_pred==0, 2], X[y_pred==0, 3], "yo", label="Cluster 1")
plt.plot(X[y_pred==1, 2], X[y_pred==1, 3], "bs", label="Cluster 2")
plt.plot(X[y_pred==2, 2], X[y_pred==2, 3], "g^", label="Cluster 3")
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="upper left", fontsize=12)
plt.show()
```



K-Means

Let's start by generating some blobs:

In [8]:
```
from sklearn.datasets import make_blobs
```

In [9]:
```
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])
```

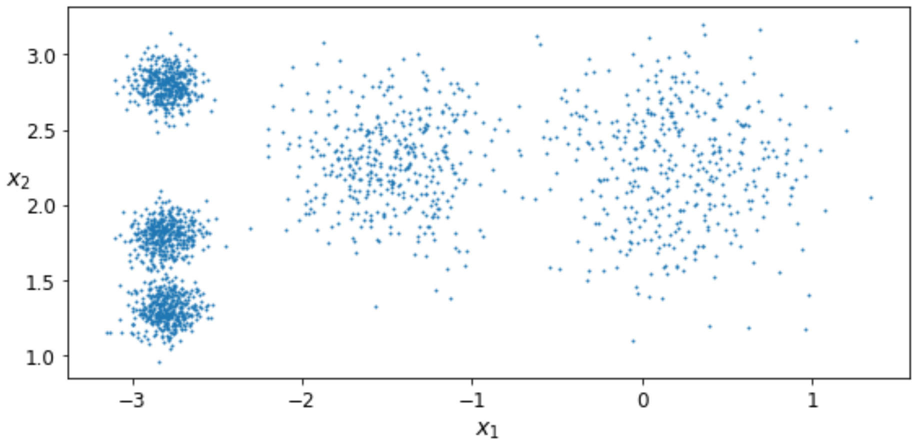
In [10]:
```
X, y = make_blobs(n_samples=2000, centers=blob_centers,
                  cluster_std=blob_std, random_state=7)
```

Now let's plot them:

In [11]:
```
def plot_clusters(X, y=None):
    plt.scatter(X[:, 0], X[:, 1], c=y, s=1)
    plt.xlabel("$x_1$", fontsize=14)
    plt.ylabel("$x_2$", fontsize=14, rotation=0)
```

In [12]:
```
plt.figure(figsize=(8, 4))
plot_clusters(X)
save_fig("blobs_plot")
plt.show()
```

```
Saving figure blobs_plot
```



Fit and Predict

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

In [13]:
```
from sklearn.cluster import KMeans
```

In [14]:
```
k = 5
kmeans = KMeans(n_clusters=k, random_state=42)
y_pred = kmeans.fit_predict(X)
```

Each instance was assigned to one of the 5 clusters:

In [15]:
```
y_pred
```

Out[15]:
```
array([0, 4, 1, ..., 2, 1, 4])
```

And the following 5 *centroids* (i.e., cluster centers) were estimated:

In [16]:
```
kmeans.cluster_centers_
```

Out[16]:
```
array([[-2.80037642,  1.30082566],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80389616,  1.80117999]])
```

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:

In [17]:
```
kmeans.labels_
```

Out[17]:
```
array([0, 4, 1, ..., 2, 1, 4])
```

Of course, we can predict the labels of new instances:

In [18]:
```
X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
kmeans.predict(X_new)
```

Out[18]:
```
array([1, 1, 2, 2])
```

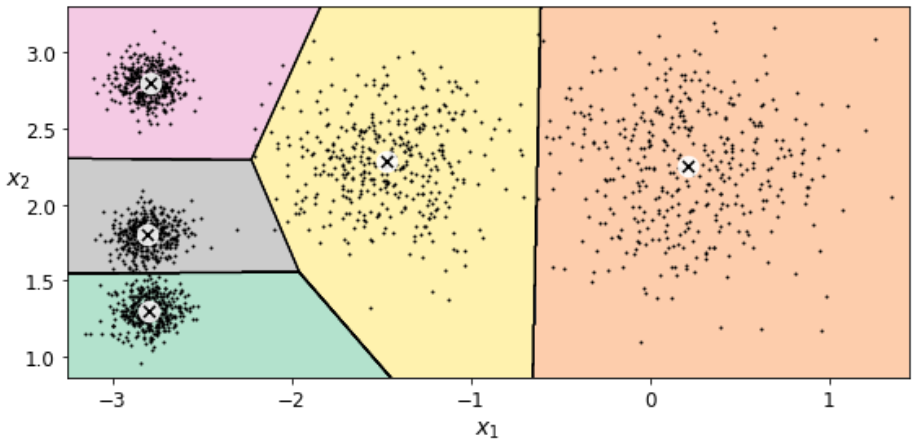
Decision Boundaries

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

In [19]:
```
def plot_data(X):
    plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)

def plot_centroids(centroids, weights=None, circle_color='w', cross_color='k
'):
    if weights is not None:
        centroids = centroids[weights > weights.max() / 10]
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='o', s=30, linewidths=8,
                color=circle_color, zorder=10, alpha=0.9)
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='x', s=50, linewidths=50,
                color=cross_color, zorder=11, alpha=1)

def plot_decision_boundaries(clusterer, X, resolution=1000, show_centroids=T
rue,
                             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("$x_1$", fontsize=14)
    else:
        plt.tick_params(labelbottom=False)
    if show_ylabels:
        plt.ylabel("$x_2$", fontsize=14, rotation=0)
    else:
        plt.tick_params(labelleft=False)
```

In [20]:
```
plt.figure(figsize=(8, 4))
plot_decision_boundaries(kmeans, X)
save_fig("voronoi_plot")
plt.show()
```

```
Saving figure voronoi_plot
```



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 measure the distance of each instance to all 5 centroids. This is what the `transform()` method does:

In [21]:
```
kmeans.transform(X_new)
```

Out[21]:
```
array([[2.88633901, 0.32995317, 2.9042344 , 1.49439034, 2.81093633],
       [5.84236351, 2.80290755, 5.84739223, 4.4759332 , 5.80730058],
       [1.71086031, 3.29399768, 0.29040966, 1.69136631, 1.21475352],
       [1.21567622, 3.21806371, 0.36159148, 1.54808703, 0.72581411]])
```

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

In [22]:
```
np.linalg.norm(np.tile(X_new, (1, k)).reshape(-1, k, 2) - kmeans.cluster_cen
ters_, axis=2)
```

Out[22]:
```
array([[2.88633901, 0.32995317, 2.9042344 , 1.49439034, 2.81093633],
       [5.84236351, 2.80290755, 5.84739223, 4.4759332 , 5.80730058],
       [1.71086031, 3.29399768, 0.29040966, 1.69136631, 1.21475352],
       [1.21567622, 3.21806371, 0.36159148, 1.54808703, 0.72581411]])
```

K-Means Algorithm

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

- First initialize k centroids randomly: 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 applies an optimized algorithm by default. To get the original K-Means algorithm (for educational purposes only), you must set `init="random"`, `n_init=1` and `algorithm="full"`. These hyperparameters will be explained below.

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

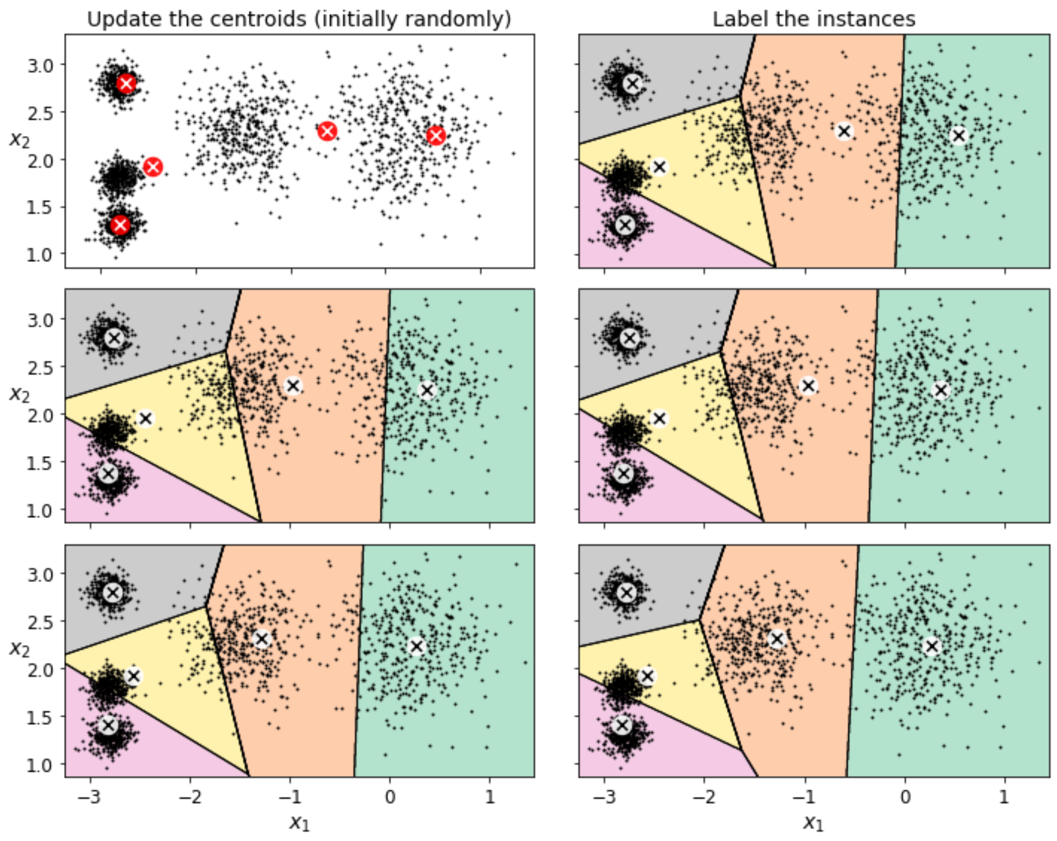
In [23]:
```
kmeans_iter1 = KMeans(n_clusters=5, init="random", n_init=1,
                     algorithm="full", max_iter=1, random_state=1)
kmeans_iter2 = KMeans(n_clusters=5, init="random", n_init=1,
                     algorithm="full", max_iter=2, random_state=1)
kmeans_iter3 = KMeans(n_clusters=5, init="random", n_init=1,
                     algorithm="full", max_iter=3, random_state=1)
kmeans_iter1.fit(X)
kmeans_iter2.fit(X)
kmeans_iter3.fit(X)
```

Out[23]:
```
KMeans(algorithm='full', init='random', max_iter=3, n_clusters=5, n_init=1,
       random_state=1)
```

And let's plot this:

In [24]:
```
plt.figure(figsize=(10, 8))

plt.subplot(321)
plot_data(X)
plot_centroids(kmeans_iter1.cluster_centers_, circle_color='r', cross_color=
'w')
plt.ylabel("$x_2$", fontsize=14, rotation=0)
plt.tick_params(labelbottom=False)
plt.title("Update the centroids (initially randomly)", fontsize=14)

plt.subplot(322)
plot_decision_boundaries(kmeans_iter1, X, show_xlabels=False, show_ylabels=F
alse)
plt.title("Label the instances", fontsize=14)

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=F
alse)

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()
```

```
Saving figure kmeans_algorithm_plot
```



K-Means Variability

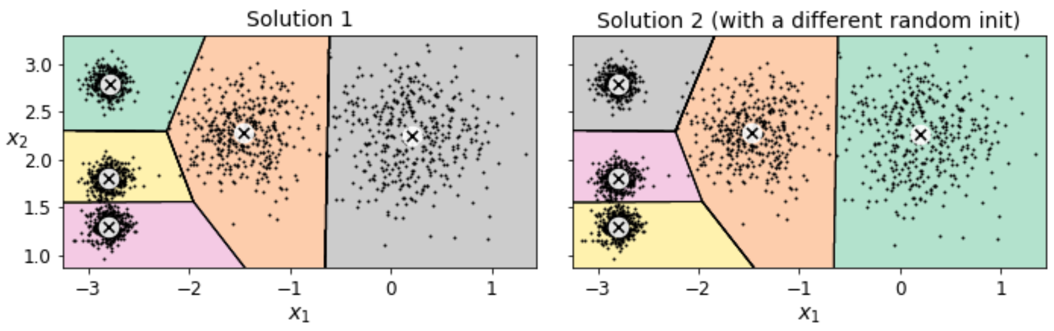
In the original K-Means algorithm, the centroids are just initialized randomly, and the algorithm simply runs a single iteration to gradually improve the centroids, as we saw above.

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 [25]:
```
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, fontsize=14)

    plt.subplot(122)
    plot_decision_boundaries(clusterer2, X, show_ylabels=False)
    if title2:
        plt.title(title2, fontsize=14)
```

In [26]:
```
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=19)

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()
```

```
Saving figure kmeans_variability_plot
```



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 [27]: kmeans.inertia_
Out[27]: 211.5985372581683
```

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

```
In [28]: X_dist = kmeans.transform(X)
         np.sum(X_dist[np.arange(len(X_dist)), kmeans.labels_]**2)
Out[28]: 211.5985372581687
```

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

```
In [29]: kmeans.score(X)
Out[29]: -211.59853725816828
```

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. For example, here are the inertias of the two "bad" models shown in the previous figure:

```
In [30]: kmeans_rnd_init1.inertia_
Out[30]: 211.60832621558367
```

```
In [31]: kmeans_rnd_init2.inertia_
Out[31]: 211.62301821329766
```

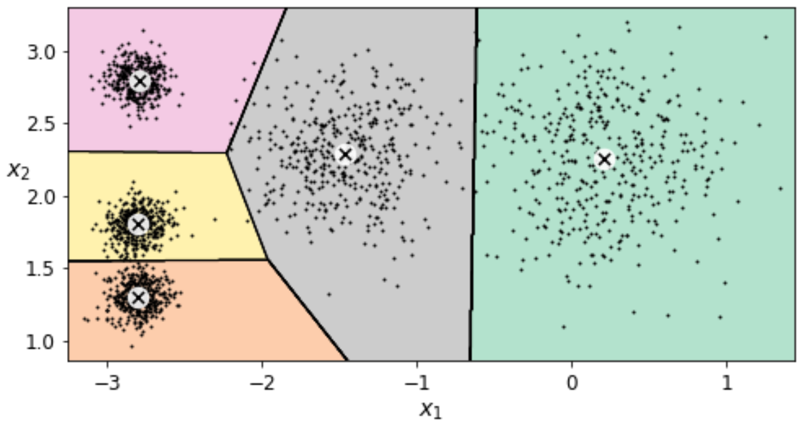
As you can see, they have a higher inertia than the first "good" model we trained, which means they are probably worse.

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 [32]: kmeans_rnd_10_inits = KMeans(n_clusters=5, init="random", n_init=10,
                                      algorithm="full", random_state=11)
         kmeans_rnd_10_inits.fit(X)
Out[32]: KMeans(algorithm='full', init='random', n_clusters=5, random_state=11)
```

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 [33]: plt.figure(figsize=(8, 4))
         plot_decision_boundaries(kmeans_rnd_10_inits, X)
         plt.show()
```



K-Means++

Instead of initializing the centroids entirely randomly, it is preferable to initialize them using the following algorithm, proposed in a 2006 paper (https://goo.gl/eNUPw6) 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):

```
In [34]: KMeans()
Out[34]: KMeans()
```

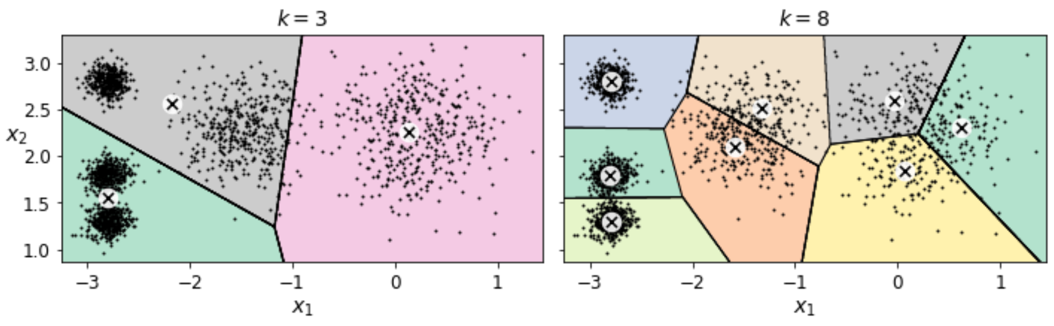
```
In [35]: good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
         kmeans = KMeans(n_clusters=5, init=good_init, n_init=1, random_state=42)
         kmeans.fit(X)
         kmeans.inertia_
Out[35]: 211.62337889822362
```

Finding the optimal number of clusters

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

In [36]:
```
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()
```

```
Saving figure bad_n_clusters_plot
```



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

In [37]:
```
kmeans_k3.inertia_
```

Out[37]:
```
653.2223267580945
```

In [38]:
```
kmeans_k8.inertia_
```

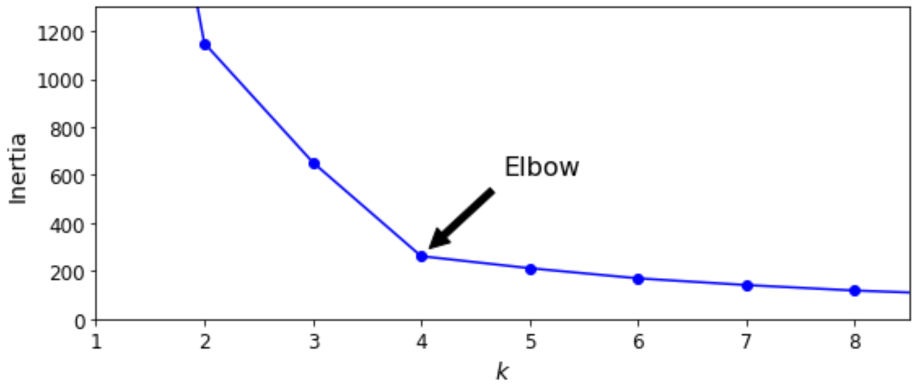
Out[38]:
```
118.44108623570082
```

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 [39]:
```
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]
```

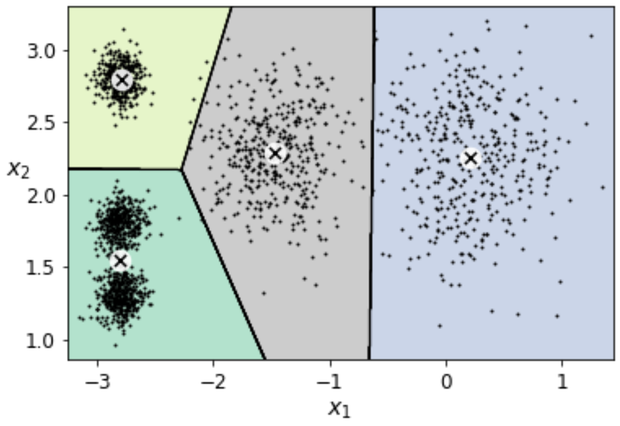
In [40]:
```
plt.figure(figsize=(8, 3.5))
plt.plot(range(1, 10), inertias, "bo-")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Inertia", fontsize=14)
plt.annotate('Elbow',
             xy=(4, inertias[3]),
             xytext=(0.55, 0.55),
             textcoords='figure fraction',
             fontsize=16,
             arrowprops=dict(facecolor='black', shrink=0.1)
            )
plt.axis([1, 8.5, 0, 1300])
save_fig("inertia_vs_k_plot")
plt.show()
```

```
Saving figure inertia_vs_k_plot
```



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 [41]:
```
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 [42]: from sklearn.metrics import silhouette_score
```

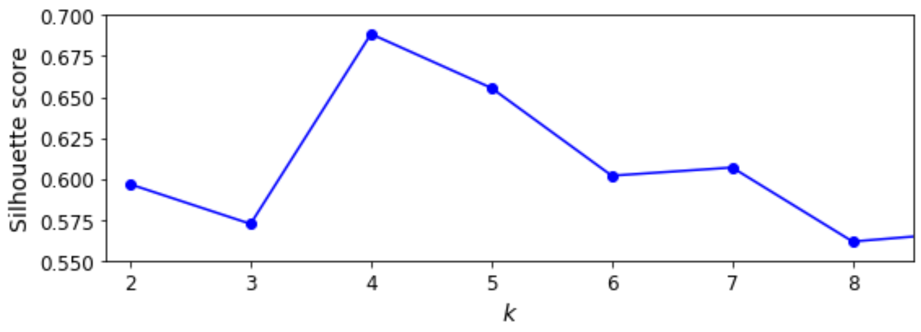
```
In [43]: silhouette_score(X, kmeans.labels_)
```

```
Out[43]: 0.655517642572828
```

```
In [44]: silhouette_scores = [silhouette_score(X, model.labels_)
                              for model in kmeans_per_k[1:]]
```

```
In [45]: plt.figure(figsize=(8, 3))
         plt.plot(range(2, 10), silhouette_scores, "bo-")
         plt.xlabel("$k$", fontsize=14)
         plt.ylabel("Silhouette score", fontsize=14)
         plt.axis([1.8, 8.5, 0.55, 0.7])
         save_fig("silhouette_score_vs_k_plot")
         plt.show()
```

```
Saving figure silhouette_score_vs_k_plot
```

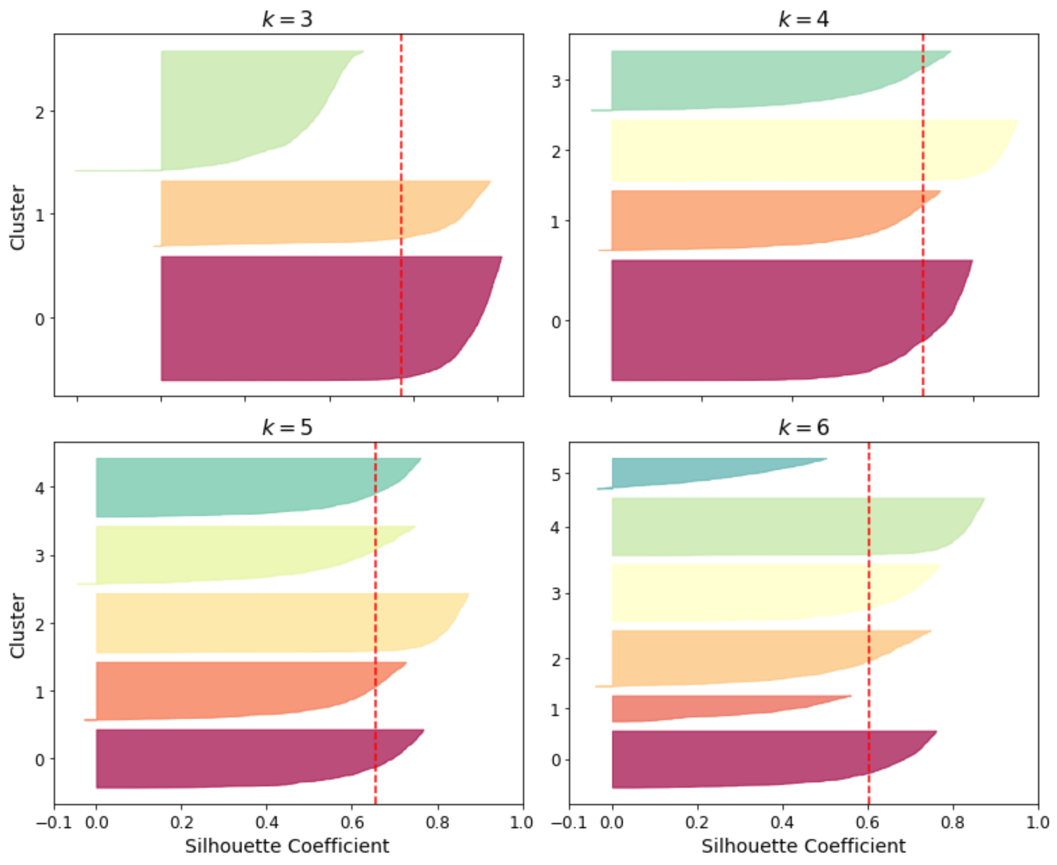


As you can see, this visualization is much richer than the previous one: in particular, although it confirms that $k = 4$ is a very good choice, but it also underlines the fact that $k = 5$ is quite good as well.

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 [46]:
```
from sklearn.metrics import silhouette_samples
from matplotlib.ticker import FixedLocator, FixedFormatter

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

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 = mpl.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("$k={}$".format(k), fontsize=16)

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

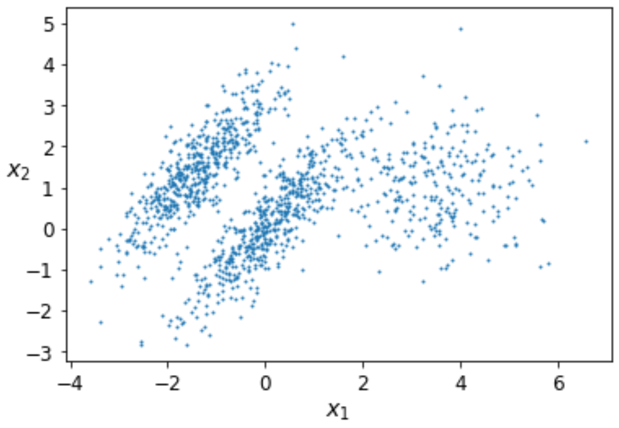
```
Saving figure silhouette_analysis_plot
```



Limits of K-Means

In [47]:
```
X1, y1 = make_blobs(n_samples=1000, centers=((4, -4), (0, 0)), random_state=
42)
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]
```

In [48]:
```
plot_clusters(X)
```

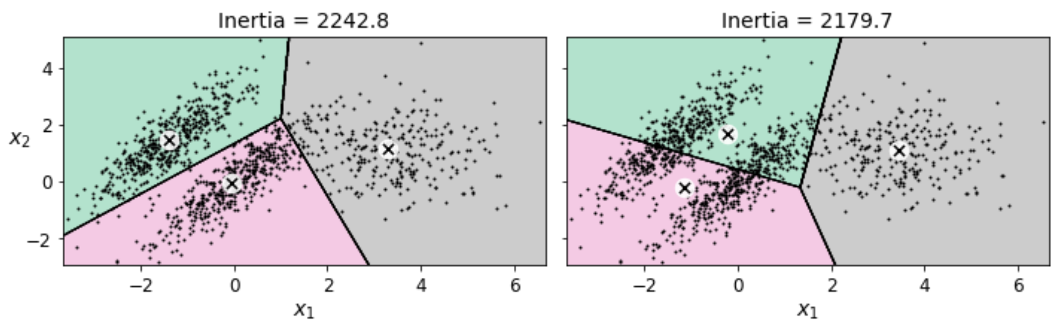


In [49]:
```
kmeans_good = KMeans(n_clusters=3, init=np.array([[-1.5, 2.5], [0.5, 0], [4,
0]]), n_init=1, random_state=42)
kmeans_bad = KMeans(n_clusters=3, random_state=42)
kmeans_good.fit(X)
kmeans_bad.fit(X)
```

Out[49]:
```
KMeans(n_clusters=3, random_state=42)
```

In [50]:
```
plt.figure(figsize=(10, 3.2))

plt.subplot(121)
plot_decision_boundaries(kmeans_good, X)
plt.title("Inertia = {:.1f}".format(kmeans_good.inertia_), fontsize=14)

plt.subplot(122)
plot_decision_boundaries(kmeans_bad, X, show_ylabels=False)
plt.title("Inertia = {:.1f}".format(kmeans_bad.inertia_), fontsize=14)

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

```
Saving figure bad_kmeans_plot
```



Gaussian Mixtures

In [51]:
```
X1, y1 = make_blobs(n_samples=1000, centers=((4, -4), (0, 0)), random_state=
42)
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]
```

Let's train a Gaussian mixture model on the previous dataset:

In [52]:
```
from sklearn.mixture import GaussianMixture
```

In [53]:
```
gm = GaussianMixture(n_components=3, n_init=10, random_state=42)
gm.fit(X)
```

Out[53]:
```
GaussianMixture(n_components=3, n_init=10, random_state=42)
```

Let's look at the parameters that the EM algorithm estimated:

```
In [54]: gm.weights_
Out[54]: array([0.39054348, 0.2093669 , 0.40008962])
```

```
In [55]: gm.means_
Out[55]: array([[ 0.05224874,  0.07631976],
               [ 3.40196611,  1.05838748],
               [-1.40754214,  1.42716873]])
```

```
In [56]: gm.covariances_
Out[56]: array([[[ 0.6890309 ,  0.79717058],
                [ 0.79717058,  1.21367348]],

               [[ 1.14296668, -0.03114176],
                [-0.03114176,  0.9545003 ]],

               [[ 0.63496849,  0.7298512 ],
                [ 0.7298512 ,  1.16112807]]])
```

Did the algorithm actually converge?

```
In [57]: gm.converged_
Out[57]: True
```

Yes, good. How many iterations did it take?

```
In [58]: gm.n_iter_
Out[58]: 4
```

You can now use the model to predict which cluster each instance belongs to (hard clustering) or the probabilities that it came from each cluster. For this, just use `predict()` method or the `predict_proba()` method:

```
In [59]: gm.predict(X)
Out[59]: array([0, 0, 2, ..., 1, 1, 1], dtype=int64)
```

```
In [60]: gm.predict_proba(X)
Out[60]: array([[9.77227791e-01, 2.27715290e-02, 6.79898914e-07],
               [9.83288385e-01, 1.60345103e-02, 6.77104389e-04],
               [7.51824662e-05, 1.90251273e-06, 9.99922915e-01],
               ...,
               [4.35053542e-07, 9.99999565e-01, 2.17938894e-26],
               [5.27837047e-16, 1.00000000e+00, 1.50679490e-41],
               [2.32355608e-15, 1.00000000e+00, 8.21915701e-41]])
```

This is a generative model, so you can sample new instances from it (and get their labels):

```
In [61]: X_new, y_new = gm.sample(6)
         X_new
```

```
Out[61]: array([[-0.8690223 , -0.32680051],
                [ 0.29945755,  0.2841852 ],
                [ 1.85027284,  2.06556913],
                [ 3.98260019,  1.50041446],
                [ 3.82006355,  0.53143606],
                [-1.04015332,  0.7864941 ]])
```

```
In [62]: y_new
```

```
Out[62]: array([0, 0, 1, 1, 1, 2])
```

Notice that they are sampled sequentially from each cluster.

You can also estimate the log of the *probability density function* (PDF) at any location using the `score_samples()` method:

```
In [63]: gm.score_samples(X)
```

```
Out[63]: array([-2.60674489, -3.57074133, -3.33007348, ..., -3.51379355,
                -4.39643283, -3.8055665 ])
```

Let's check that the PDF integrates to 1 over the whole space. We just take a large square around the clusters, and chop it into a grid of tiny squares, then we compute the approximate probability that the instances will be generated in each tiny square (by multiplying the PDF at one corner of the tiny square by the area of the square), and finally summing all these probabilities). The result is very close to 1:

```
In [64]: resolution = 100
         grid = np.arange(-10, 10, 1 / resolution)
         xx, yy = np.meshgrid(grid, grid)
         X_full = np.vstack([xx.ravel(), yy.ravel()]).T

         pdf = np.exp(gm.score_samples(X_full))
         pdf_probas = pdf * (1 / resolution) ** 2
         pdf_probas.sum()
```

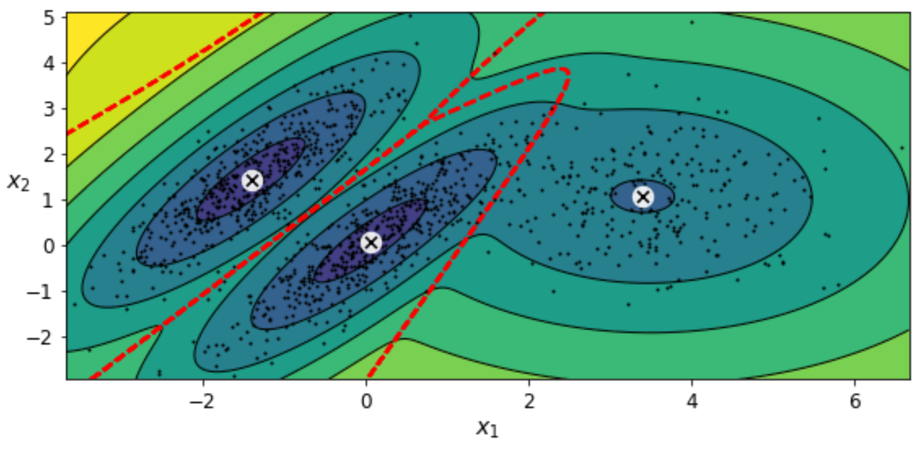
```
Out[64]: 0.9999999999271592
```

Now let's plot the resulting decision boundaries (dashed lines) and density contours:

In [65]:
```
from matplotlib.colors import LogNorm

def plot_gaussian_mixture(clusterer, X, resolution=1000, 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.score_samples(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z,
                 norm=LogNorm(vmin=1.0, vmax=30.0),
                 levels=np.logspace(0, 2, 12))
    plt.contour(xx, yy, Z,
                norm=LogNorm(vmin=1.0, vmax=30.0),
                levels=np.logspace(0, 2, 12),
                linewidths=1, colors='k')

    Z = clusterer.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.contour(xx, yy, Z,
                linewidths=2, colors='r', linestyles='dashed')

    plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)
    plot_centroids(clusterer.means_, clusterer.weights_)

    plt.xlabel("$x_1$", fontsize=14)
    if show_ylabels:
        plt.ylabel("$x_2$", fontsize=14, rotation=0)
    else:
        plt.tick_params(labelleft=False)
```

In [66]:
```
plt.figure(figsize=(8, 4))

plot_gaussian_mixture(gm, X)

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

```
Saving figure gaussian_mixtures_plot
```



You can impose constraints on the covariance matrices that the algorithm looks for by setting the `covariance_type` hyperparameter:

- `"full"` (default): no constraint, all clusters can take on any ellipsoidal shape of any size.
- `"tied"` : all clusters must have the same shape, which can be any ellipsoid (i.e., they all share the same covariance matrix).
- `"spherical"` : all clusters must be spherical, but they can have different diameters (i.e., different variances).
- `"diag"` : clusters can take on any ellipsoidal shape of any size, but the ellipsoid's axes must be parallel to the axes (i.e., the covariance matrices must be diagonal).

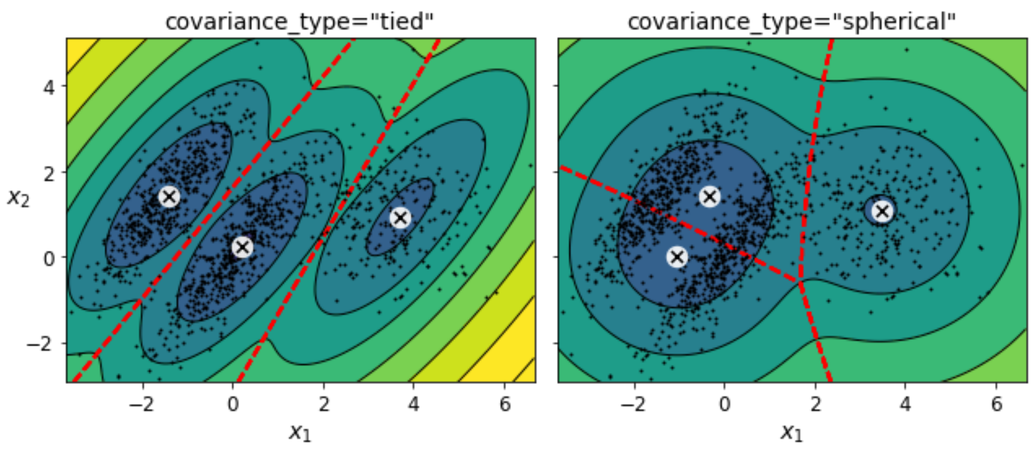
In [67]:
```
gm_full = GaussianMixture(n_components=3, n_init=10, covariance_type="full",
random_state=42)
gm_tied = GaussianMixture(n_components=3, n_init=10, covariance_type="tied",
random_state=42)
gm_spherical = GaussianMixture(n_components=3, n_init=10, covariance_type="s
pherical", random_state=42)
gm_diag = GaussianMixture(n_components=3, n_init=10, covariance_type="diag",
random_state=42)
gm_full.fit(X)
gm_tied.fit(X)
gm_spherical.fit(X)
gm_diag.fit(X)
```

Out[67]:
```
GaussianMixture(covariance_type='diag', n_components=3, n_init=10,
                random_state=42)
```

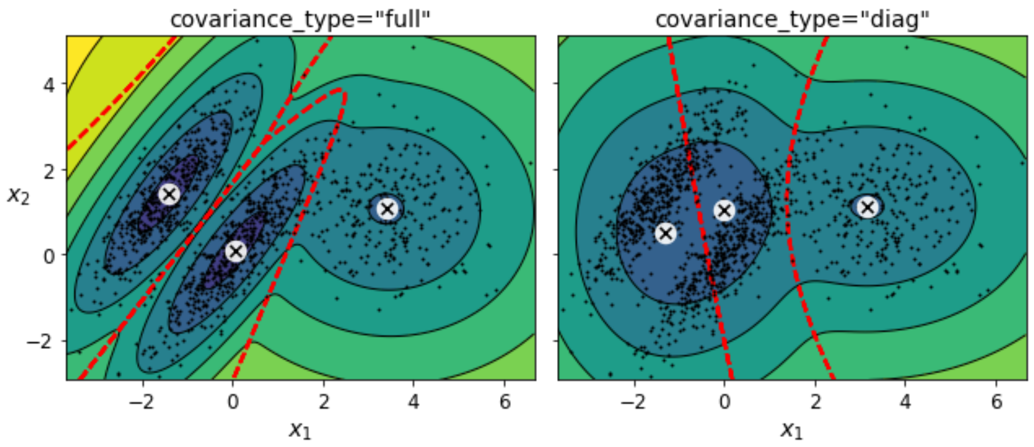
In [68]:
```
def compare_gaussian_mixtures(gm1, gm2, X):
    plt.figure(figsize=(9, 4))

    plt.subplot(121)
    plot_gaussian_mixture(gm1, X)
    plt.title('covariance_type="{}"'.format(gm1.covariance_type), fontsize=1
4)

    plt.subplot(122)
    plot_gaussian_mixture(gm2, X, show_ylabels=False)
    plt.title('covariance_type="{}"'.format(gm2.covariance_type), fontsize=1
4)
```

In [69]:
```
compare_gaussian_mixtures(gm_tied, gm_spherical, X)

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

```
Saving figure covariance_type_plot
```



In [70]:
```
compare_gaussian_mixtures(gm_full, gm_diag, X)
plt.tight_layout()
plt.show()
```



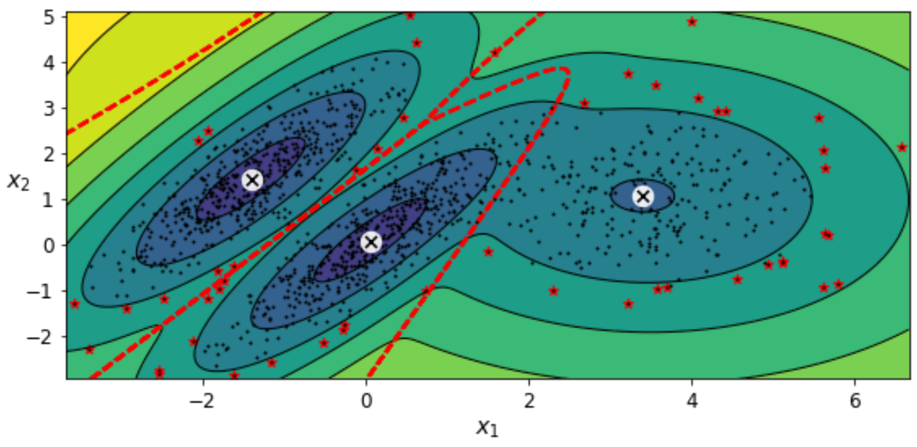
Anomaly Detection using Gaussian Mixtures

Gaussian Mixtures can be used for *anomaly detection*: instances located in low-density regions can be considered anomalies. You must define what density threshold you want to use. For example, in a manufacturing company that tries to detect defective products, the ratio of defective products is usually well-known. Say it is equal to 4%, then you can set the density threshold to be the value that results in having 4% of the instances located in areas below that threshold density:

In [71]:
```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 4)
anomalies = X[densities < density_threshold]
```

In [72]:
```
plt.figure(figsize=(8, 4))

plot_gaussian_mixture(gm, X)
plt.scatter(anomalies[:, 0], anomalies[:, 1], color='r', marker='*')
plt.ylim(top=5.1)

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

```
Saving figure mixture_anomaly_detection_plot
```



Model selection

We cannot use the inertia or the silhouette score because they both assume that the clusters are spherical. Instead, we can try to find the model that minimizes a theoretical information criterion such as the Bayesian Information Criterion (BIC) or the Akaike Information Criterion (AIC):

$$BIC = \log(m)p - 2\log(\hat{L})$$

$$AIC = 2p - 2\log(\hat{L})$$

- m is the number of instances.
- p is the number of parameters learned by the model.
- $\hat{L}$ is the maximized value of the likelihood function of the model. This is the conditional probability of the observed data $\mathbf{X}$, given the model and its optimized parameters.

Both BIC and AIC penalize models that have more parameters to learn (e.g., more clusters), and reward models that fit the data well (i.e., models that give a high likelihood to the observed data).

```
In [73]: gm.bic(X)
Out[73]: 8189.662685850679
```

```
In [74]: gm.aic(X)
Out[74]: 8102.437405735641
```

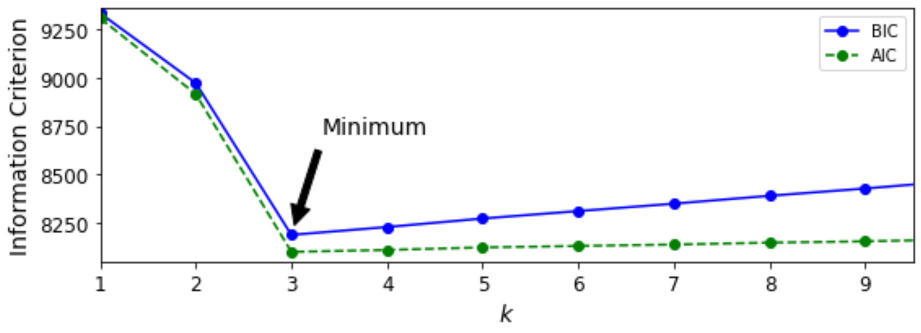
Let's train Gaussian Mixture models with various values of k and measure their BIC:

```
In [75]: gms_per_k = [GaussianMixture(n_components=k, n_init=10, random_state=42).fit
         (X)
                      for k in range(1, 11)]
```

```
In [76]: bics = [model.bic(X) for model in gms_per_k]
         aics = [model.aic(X) for model in gms_per_k]
```

In [77]:
```python
plt.figure(figsize=(8, 3))
plt.plot(range(1, 11), bics, "bo-", label="BIC")
plt.plot(range(1, 11), aics, "go--", label="AIC")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Information Criterion", fontsize=14)
plt.axis([1, 9.5, np.min(aics) - 50, np.max(aics) + 50])
plt.annotate('Minimum',
             xy=(3, bics[2]),
             xytext=(0.35, 0.6),
             textcoords='figure fraction',
             fontsize=14,
             arrowprops=dict(facecolor='black', shrink=0.1)
            )
plt.legend()
save_fig("aic_bic_vs_k_plot")
plt.show()
```

```
Saving figure aic_bic_vs_k_plot
```



Let's search for best combination of values for both the number of clusters and the `covariance_type` hyperparameter:

In [78]:
```python
min_bic = np.infty

for k in range(1, 11):
    for covariance_type in ("full", "tied", "spherical", "diag"):
        bic = GaussianMixture(n_components=k, n_init=10,
                              covariance_type=covariance_type,
                              random_state=42).fit(X).bic(X)
        if bic < min_bic:
            min_bic = bic
            best_k = k
            best_covariance_type = covariance_type
```

In [79]:
```python
best_k
```

Out[79]:
```
3
```

In [80]:
```python
best_covariance_type
```

Out[80]:
```
'full'
```

Note : Rather than manually searching for the optimal number of clusters, it is possible to use instead the `BayesianGaussianMixture` class which is capable of giving weights equal (or close) to zero to unnecessary clusters. Just set the number of components to a value that you believe is greater than the optimal number of clusters, and the algorithm will eliminate the unnecessary clusters automatically.

In [ ]: