Notebook 9 – Aprendizaje no supervisado

Este cuaderno contiene ejemplos muy detallados sobre cómo funciona el aprendizaje no supervisado

Configuración

Primero, importemos algunos módulos comunes, asegurémonos de que MatplotLib trace figuras en línea y prepare una función para guardar las figuras. También verificamos que Python 3.5 o posterior esté instalado (aunque Python 2.x puede funcionar, está obsoleto, por lo que le recomendamos que use Python 3 en su lugar), así como Scikit-Learn ≥0.20.

In [1]:

```
# Se requiere python ≥3.5
import sys
assert sys.version_info >= (3, 5)

# Se requiere scikit-learn ≥0.20
import sklearn
assert sklearn.__version__ >= "0.20"

# Importaciones comunes
import numpy as np
import os

# Para hacer que la salida de este portátil sea estable a lo largo de las ejecuciones
np.random.seed(42)

# Para trazar figuras bonitas
%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)

# Dónde guardar las figuras
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=300):
    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)
```

Agrupación

Introducción: clasificación *frente* a agrupamiento

In [2]:

```
from sklearn.datasets import load_iris
```

In [3]:

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

```
data.target_names
```

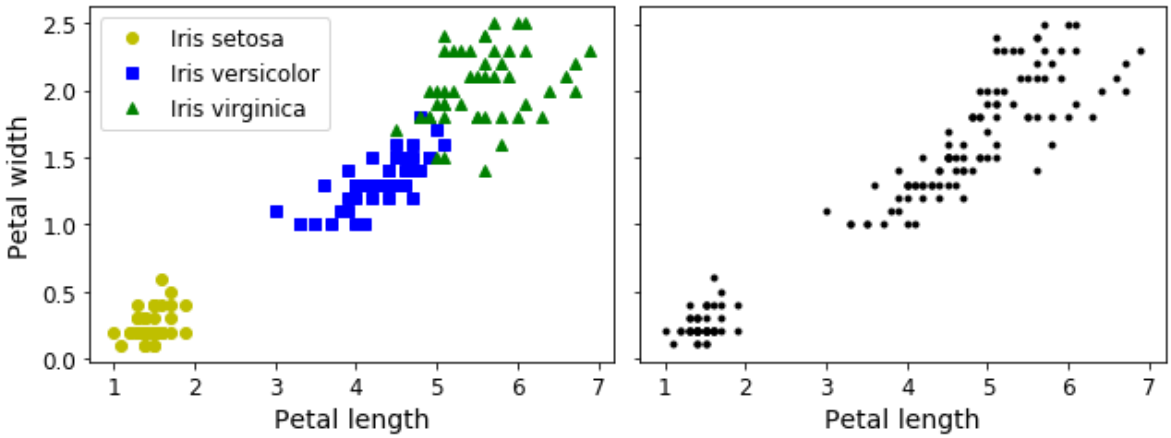
Out[3]:

```
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
```



Un modelo de mezcla gaussiana (explicado a continuación) en realidad puede separar estos grupos bastante bien (usando las 4 características: largo y ancho de los pétalos, y largo y ancho de los sépalos).

In [5]:

```
from sklearn.mixture import GaussianMixture
```

In [6]:

```
y_pred = GaussianMixture(n_components=3, random_state=42).fit(X).predict(X)
```

Mapeemos cada grupo a una clase. En lugar de codificar el mapeo (como se hace en el libro, para simplificar), elegiremos la clase más común para cada grupo (usando la función `scipy.stats.mode()`):

In [7]:

```
from scipy import stats

mapping = {}
for class_id in np.unique(y):
    mode, _ = stats.mode(y_pred[y==class_id])
    mapping[mode[0]] = class_id

mapping
```

Out[7]:

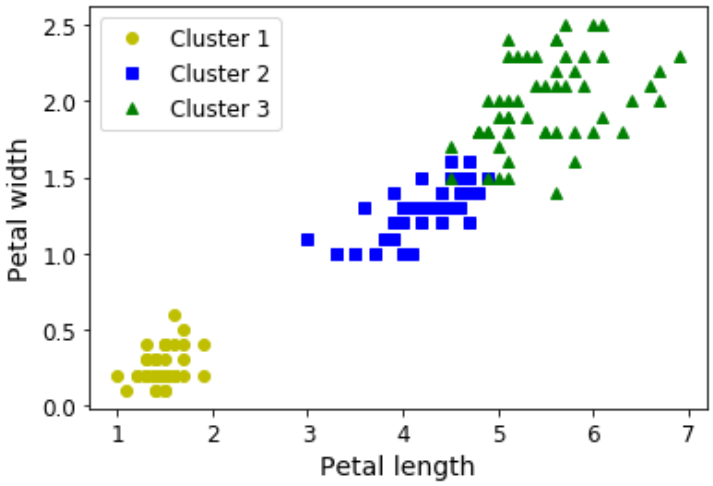
{2: 0, 0: 1, 1: 2}

In [8]:

```
y_pred = np.array([mapping[cluster_id] for cluster_id in y_pred])
```

In [9]:

```
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()
```



In [10]:

```
np.sum(y_pred==y)
```

Out[10]:

```
145
```

In [11]:

```
np.sum(y_pred==y) / len(y_pred)
```

Out[11]:

```
0.9666666666666667
```

Nota : los resultados de este cuaderno pueden diferir ligeramente de los del libro. Esto se debe a que los algoritmos a veces se pueden modificar un poco entre las versiones de Scikit-Learn.

K-medias

Comencemos generando algunos blobs:

In [12]:

```
from sklearn.datasets import make_blobs
```

In [13]:

```
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 [14]:

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

Ahora vamos a graficarlos:

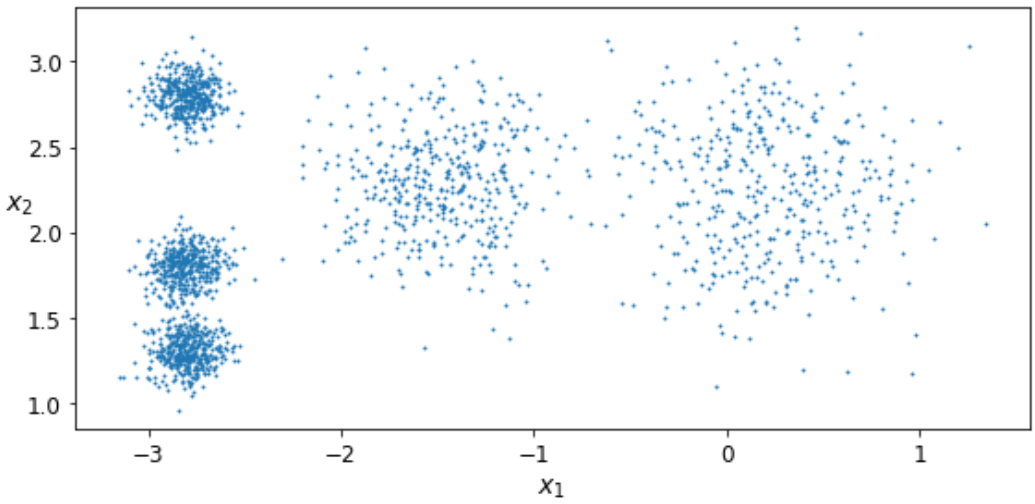
In [15]:

```
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 [16]:

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

```
Saving figure blobs_plot
```



Ajustar y predecir

Entrenemos un agrupador de K-Means en este conjunto de datos. Intentará encontrar el centro de cada blob y asignar cada instancia al blob más cercano:

In [17]:

```
from sklearn.cluster import KMeans
```

In [18]:

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

Cada instancia se asignó a uno de los 5 clústeres:

In [19]:

```
y_pred
```

Out[19]:

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

In [20]:

```
y_pred is kmeans.labels_
```

Out[20]:

```
True
```

Y se estimaron los siguientes 5 *centroides* (es decir, centros de conglomerados):

In [21]:

```
kmeans.cluster_centers_
```

Out[21]:

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

Tenga en cuenta que la instancia de `KMeans` conserva las etiquetas de las instancias en las que se entrenó. Algo confuso, en este contexto, la *etiqueta* de una instancia es el índice del clúster al que se asigna esa instancia:

In [22]:

```
kmeans.labels_
```

Out[22]:

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

Por supuesto, podemos predecir las etiquetas de nuevas instancias:

In [23]:

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

Out[23]:

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

Límites de decisión

Tracemos los límites de decisión del modelo. Esto nos da un *diagrama de Voronoi*:

In [24]:

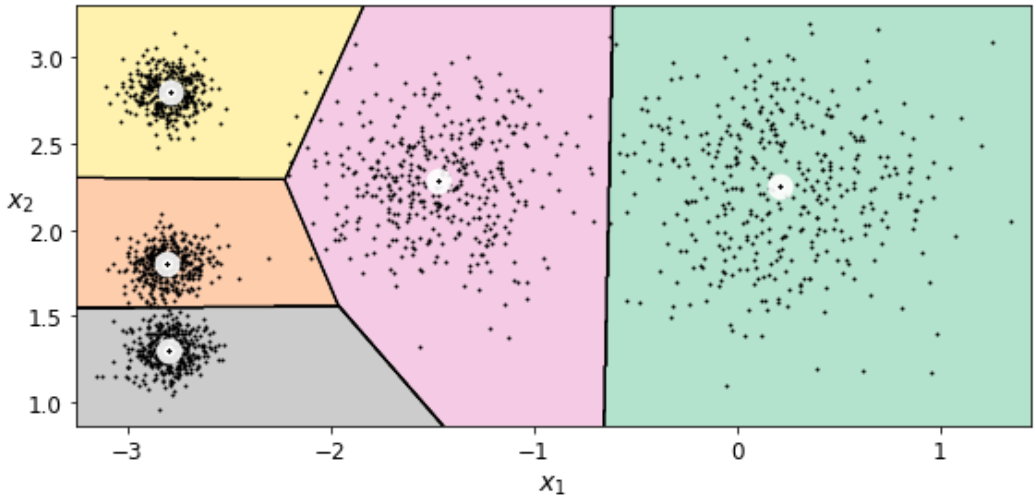
```
def plot_data(X):
    plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)

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=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_centroids=True,
                             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 [25]:

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

```
Saving figure voronoi_plot
```



¡Nada mal! Algunas de las instancias cercanas a los bordes probablemente se asignaron al clúster incorrecto, pero en general se ve bastante bien.

Agrupamiento duro *frente* a agrupamiento suave

En lugar de elegir arbitrariamente el clúster más cercano para cada instancia, lo que se denomina *agrupación dura*, podría ser mejor medir la distancia de cada instancia a los 5 centroides. Esto es lo que hace el método `transform()`:

In [26]:

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

Out[26]:

```
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]])
```

Puede verificar que esta es de hecho la distancia euclidiana entre cada instancia y cada centroide:

In [27]:

```
np.linalg.norm(np.tile(X_new, (1, k)).reshape(-1, k, 2) - kmeans.cluster_centers_, axis=
2)
```

Out[27]:

```
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]])
```

El algoritmo de K-Means

El algoritmo K-Means es uno de los algoritmos de agrupamiento más rápidos y también uno de los más simples:

- **Primero inicialice los centroides de k al azar: las instancias distintas de k se eligen al azar del conjunto de datos y los centroides se colocan en sus ubicaciones.**
- **Repita hasta la convergencia (es decir, hasta que los centroides dejen de moverse):**
  - **Asigne cada instancia al centroide más cercano.**
  - **Actualice los centroides para que sean la media de las instancias que se les asignan.**

La clase `KMeans` aplica un algoritmo optimizado por defecto. Para obtener el algoritmo original de K-Means (solo con fines educativos), debe configurar `init="random"` , `n_init=1` y `algorithm="full"` . Estos hiperparámetros se explicarán a continuación.

Ejecutemos el algoritmo K-Means para 1, 2 y 3 iteraciones, para ver cómo se mueven los centroides:

In [28]:

```
kmeans_iter1 = KMeans(n_clusters=5, init="random", n_init=1,
                      algorithm="full", max_iter=1, random_state=0)
kmeans_iter2 = KMeans(n_clusters=5, init="random", n_init=1,
                      algorithm="full", max_iter=2, random_state=0)
kmeans_iter3 = KMeans(n_clusters=5, init="random", n_init=1,
                      algorithm="full", max_iter=3, random_state=0)
kmeans_iter1.fit(X)
kmeans_iter2.fit(X)
kmeans_iter3.fit(X)
```

Out[28]:

```
KMeans(algorithm='full', copy_x=True, init='random', max_iter=3, n_clusters=5,
       n_init=1, n_jobs=None, precompute_distances='auto', random_state=0,
       tol=0.0001, verbose=0)
```

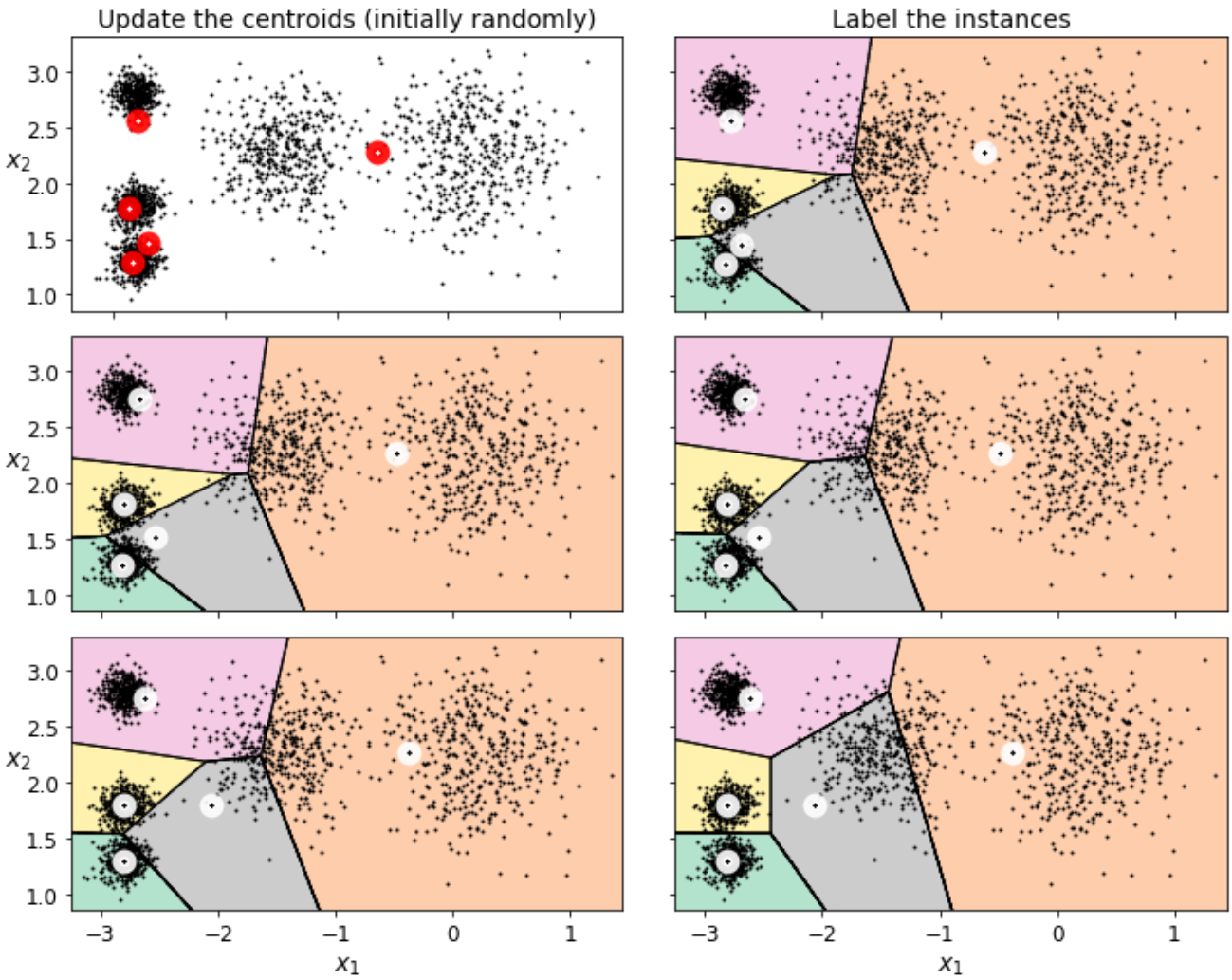
Y vamos a graficar esto:

In [29]:

```
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=False)
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=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()
```

```
Saving figure kmeans_algorithm_plot
```



Variabilidad de las medias K

En el algoritmo original de K-Means, los centroides simplemente se inicializan aleatoriamente y el algoritmo simplemente ejecuta una sola iteración para mejorar gradualmente los centroides, como vimos anteriormente.

Sin embargo, un problema importante con este enfoque es que si ejecuta K-Means varias veces (o con semillas aleatorias diferentes), puede converger en soluciones muy diferentes, como puede ver a continuación:

In [30]:

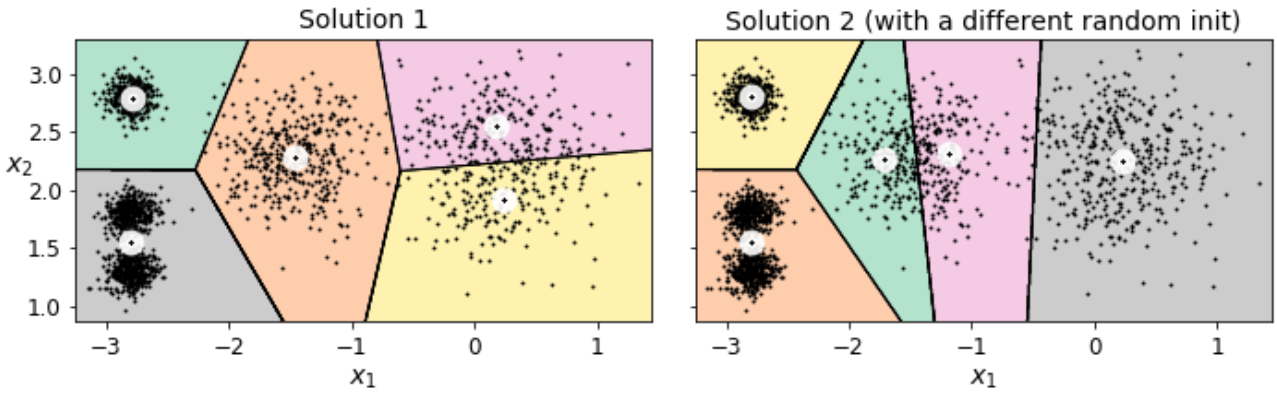
```
def plot_clusterer_comparison(clusterer1, clusterer2, X, title1=None, title2=None):
    clusterer1.fit(X)
    clusterer2.fit(X)

    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 [31]:

```
kmeans_rnd_init1 = KMeans(n_clusters=5, init="random", n_init=1,
                          algorithm="full", random_state=2)
kmeans_rnd_init2 = KMeans(n_clusters=5, init="random", n_init=1,
                          algorithm="full", random_state=5)

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
```



Inercia

Para seleccionar el mejor modelo, necesitaremos una forma de evaluar el rendimiento del modelo K-Mean. Lamentablemente, la agrupación en clústeres es una tarea no supervisada, por lo que no tenemos los objetivos. Pero al menos podemos medir la distancia entre cada instancia y su centroide. Esta es la idea detrás de la métrica de *inercia* :

In [32]:

```
kmeans.inertia_
```

Out[32]:

```
211.5985372581684
```

Como puedes verificar fácilmente, la inercia es la suma de las distancias al cuadrado entre cada instancia de entrenamiento y su centroide más cercano:

In [33]:

```
X_dist = kmeans.transform(X)
np.sum(X_dist[np.arange(len(X_dist)), kmeans.labels_]**2)
```

Out[33]:

```
211.59853725816856
```

El método `score()` devuelve la inercia negativa. ¿Por qué negativo? Bueno, es porque el método `score()` de un predictor siempre debe respetar la regla "cuanto *mayor es mejor* ".

In [34]:

```
kmeans.score(X)
```

Out[34]:

```
-211.59853725816856
```

Múltiples inicializaciones

Entonces, un enfoque para resolver el problema de la variabilidad es simplemente ejecutar el algoritmo K-Means varias veces con diferentes inicializaciones aleatorias y seleccionar la solución que minimiza la inercia. Por ejemplo, aquí están las inercias de los dos modelos "malos" que se muestran en la figura anterior:

In [35]:

```
kmeans_rnd_init1.inertia_
```

Out[35]:

```
219.8385799007183
```

In [36]:

```
kmeans_rnd_init2.inertia_
```

Out[36]:

```
236.94908363907354
```

Como puede ver, tienen una inercia mayor que el primer modelo "bueno" que entrenamos, lo que significa que probablemente sean peores.

Cuando establece el hiperparámetro `n_init` , Scikit-Learn ejecuta el algoritmo original `n_init` veces y selecciona la solución que minimiza la inercia. De forma predeterminada, Scikit-Learn establece `n_init=10` .

In [37]:

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

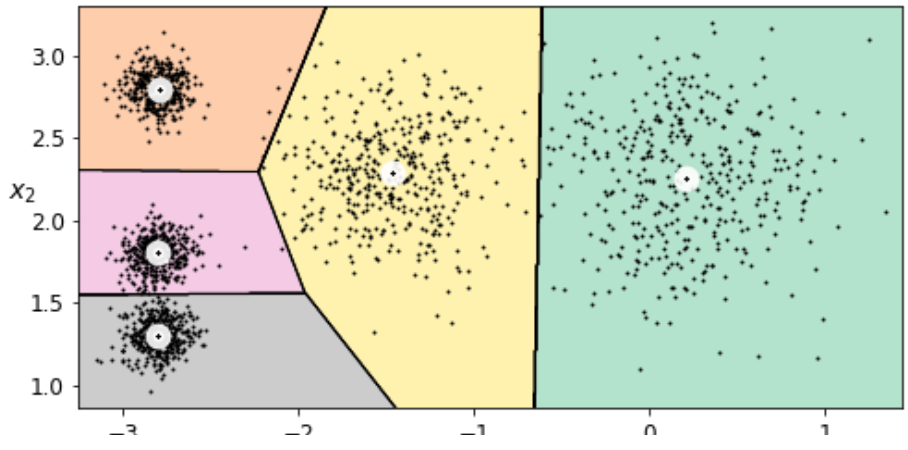
Out[37]:

```
KMeans(algorithm='full', copy_x=True, init='random', max_iter=300, n_clusters=5,
       n_init=10, n_jobs=None, precompute_distances='auto', random_state=2,
       tol=0.0001, verbose=0)
```

Como puede ver, terminamos con el modelo inicial, que sin duda es la solución óptima de K-Means (al menos en términos de inercia y suponiendo $k = 5$).

In [38]:

```
plt.figure(figsize=(8, 4))
plot_decision_boundaries(kmeans_rnd_10_inits, X)
plt.show()
```



x_1

Métodos de inicialización del centroide

En lugar de inicializar los centroides completamente al azar, es preferible inicializarlos usando el siguiente algoritmo, propuesto en un artículo de 2006 por David Arthur y Sergei Vassilvitskii:

- **Tome un centroide c_1, elegido uniformemente al azar del conjunto de datos.**
- **Tome un nuevo centro c_i, eligiendo una instancia $\mathbf{x} * i$ con probabilidad: $D(\mathbf{x}_i)^2$ / $\sum * j = 1^m D(\mathbf{x}_j)^2$ donde $D(\mathbf{x}_i)$ es la distancia entre la instancia $\mathbf{x}_i$ y el centroide más cercano que ya se eligió. Esta distribución de probabilidad asegura que las instancias que están más alejadas de los centroides ya elegidos tienen muchas más probabilidades de ser seleccionadas como centroides.**
- **Repita el paso anterior hasta que se hayan elegido todos los centroides de k .**

El resto del algoritmo K-Means++ es simplemente K-Means normal. Con esta inicialización, es mucho menos probable que el algoritmo K-Means converja a una solución subóptima, por lo que es posible reducir `n_init` considerablemente. La mayoría de las veces, esto compensa en gran medida la complejidad adicional del proceso de inicialización.

Para establecer la inicialización en K-Means++, simplemente configure `init="k-means++"` (este es el valor predeterminado):

In [39]:

```
KMeans()
```

Out[39]:

```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
       n_clusters=8, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=None, tol=0.0001, verbose=0)
```

In [40]:

```
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[40]:

```
211.5985372581684
```

K-medias aceleradas

El algoritmo K-Means se puede acelerar significativamente evitando muchos cálculos de distancia innecesarios: esto se logra explotando la desigualdad del triángulo (dados tres puntos A, B y C, la distancia AC siempre es tal que AC ≤ AB + BC) y manteniendo seguimiento de los límites inferior y superior para las distancias entre instancias y centroides (consulte este artículo de 2003 de Charles Elkan para obtener más detalles).

Para usar la variante de K-Means de Elkan, simplemente establezca `algorithm="elkan"` . Tenga en cuenta que no admite datos dispersos, por lo que, de manera predeterminada, Scikit-Learn usa `"elkan"` para datos densos y `"full"` (el algoritmo K-Means regular) para datos dispersos.

In [41]:

```
%timeit -n 50 KMeans(algorithm="elkan", random_state=42).fit(X)
```

```
56.6 ms ± 3.74 ms per loop (mean ± std. dev. of 7 runs, 50 loops each)
```

In [42]:

```
%timeit -n 50 KMeans(algorithm="full", random_state=42).fit(X)
```

```
76.5 ms ± 2.66 ms per loop (mean ± std. dev. of 7 runs, 50 loops each)
```

No hay gran diferencia en este caso, ya que el conjunto de datos es bastante pequeño.

Medias K de minilotes

Scikit-Learn también implementa una variante del algoritmo K-Means que admite minilotes (consulte [este artículo](#)):

In [43]:

```
from sklearn.cluster import MiniBatchKMeans
```

In [44]:

```
minibatch_kmeans = MiniBatchKMeans(n_clusters=5, random_state=42)
minibatch_kmeans.fit(X)
```

Out[44]:

```
MiniBatchKMeans(batch_size=100, compute_labels=True, init='k-means++',
                init_size=None, max_iter=100, max_no_improvement=10,
                n_clusters=5, n_init=3, random_state=42,
                reassignment_ratio=0.01, tol=0.0, verbose=0)
```

In [45]:

```
minibatch_kmeans.inertia_
```

Out[45]:

```
211.93186531476775
```

Si el conjunto de datos no cabe en la memoria, la opción más simple es usar la clase `memmap` , tal como lo hicimos para PCA incremental en el capítulo anterior. Primero vamos a cargar MNIST:

Advertencia: desde Scikit-Learn 0.24, `fetch_openml()` devuelve un Pandas `DataFrame` de forma predeterminada. Para evitar esto y mantener el mismo código que en el libro, usamos `as_frame=False` .

In [46]:

```
import urllib.request
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', version=1, as_frame=False)
mnist.target = mnist.target.astype(np.int64)
```

In [47]:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    mnist["data"], mnist["target"], random_state=42)
```

A continuación, vamos a escribirlo en un `memmap` :

In [48]:

```
filename = "my_mnist.data"
X_mm = np.memmap(filename, dtype='float32', mode='write', shape=X_train.shape)
X_mm[:] = X_train
```

In [49]:

```
minibatch_kmeans = MiniBatchKMeans(n_clusters=10, batch_size=10, random_state=42)
minibatch_kmeans.fit(X_mm)
```

Out[49]:

```
MiniBatchKMeans(batch_size=10, compute_labels=True, init='k-means++',
                init_size=None, max_iter=100, max_no_improvement=10,
                n_clusters=10, n_init=3, random_state=42,
                reassignment_ratio=0.01, tol=0.0, verbose=0)
```

Si sus datos son tan grandes que no puede usar `memmap` , las cosas se complican más. Comencemos escribiendo una función para cargar el siguiente lote (en la vida real, cargaría los datos desde el disco):

In [50]:

```
def load_next_batch(batch_size):
    return X[np.random.choice(len(X), batch_size, replace=False)]
```

Ahora podemos entrenar el modelo alimentándolo un lote a la vez. También necesitamos implementar múltiples inicializaciones y mantener el modelo con la inercia más baja:

In [51]:

```
np.random.seed(42)
```

In [52]:

```
k = 5
n_init = 10
n_iterations = 100
batch_size = 100
init_size = 500  # Más datos para la inicialización de k-means++
evaluate_on_last_n_iters = 10

best_kmeans = None

for init in range(n_init):
    minibatch_kmeans = MiniBatchKMeans(n_clusters=k, init_size=init_size)
    X_init = load_next_batch(init_size)
    minibatch_kmeans.partial_fit(X_init)

    minibatch_kmeans.sum_inertia_ = 0
    for iteration in range(n_iterations):
        X_batch = load_next_batch(batch_size)
        minibatch_kmeans.partial_fit(X_batch)
        if iteration >= n_iterations - evaluate_on_last_n_iters:
            minibatch_kmeans.sum_inertia_ += minibatch_kmeans.inertia_

    if (best_kmeans is None or
        minibatch_kmeans.sum_inertia_ < best_kmeans.sum_inertia_):
        best_kmeans = minibatch_kmeans
```

In [53]:

```
best_kmeans.score(X)
```

Out[53]:

```
-211.70999744411483
```

Los K-Means de minilotes son mucho más rápidos que los K-Means normales:

In [54]:

```
%timeit KMeans(n_clusters=5, random_state=42).fit(X)
```

```
32.6 ms ± 2.94 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [55]:

```
%timeit MiniBatchKMeans(n_clusters=5, random_state=42).fit(X)
```

```
18 ms ± 1.75 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

¡Eso es *mucho* más rápido! Sin embargo, su rendimiento suele ser menor (mayor inercia) y se sigue degradando a medida que aumenta *k*. Grafiquemos la relación de inercia y la relación del tiempo de entrenamiento entre K-Means de minilotes y K-Means regulares:

In [56]:

```
from timeit import timeit
```

In [57]:

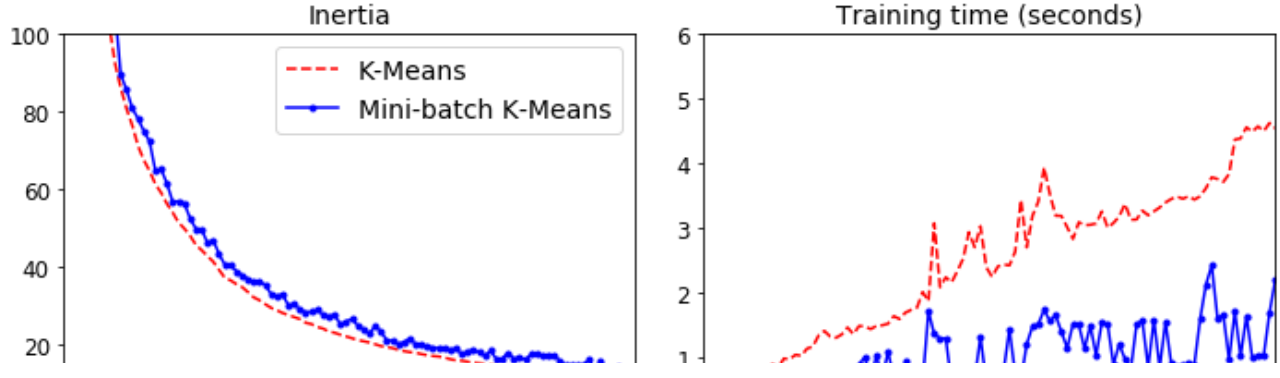
```
times = np.empty((100, 2))
inertias = np.empty((100, 2))
for k in range(1, 101):
    kmeans_ = KMeans(n_clusters=k, random_state=42)
    minibatch_kmeans = MiniBatchKMeans(n_clusters=k, random_state=42)
    print("\r{}/{}".format(k, 100), end="")
    times[k-1, 0] = timeit("kmeans_.fit(X)", number=10, globals=globals())
    times[k-1, 1]  = timeit("minibatch_kmeans.fit(X)", number=10, globals=globals())
    inertias[k-1, 0] = kmeans_.inertia_
    inertias[k-1, 1] = minibatch_kmeans.inertia_
```

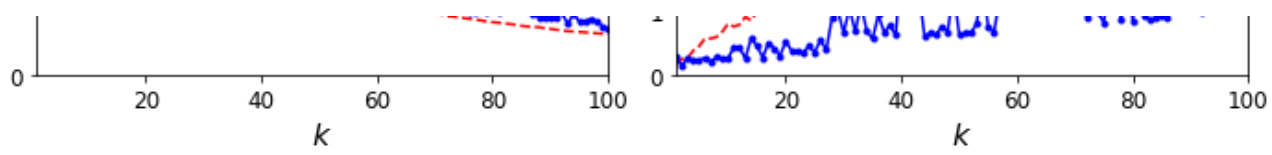
```
100/100
```

In [58]:

```
plt.figure(figsize=(10,4))

plt.subplot(121)
plt.plot(range(1, 101), inertias[:, 0], "r--", label="K-Means")
plt.plot(range(1, 101), inertias[:, 1], "b.-", label="Mini-batch K-Means")
plt.xlabel("$k$", fontsize=16)
plt.title("Inertia", fontsize=14)
plt.legend(fontsize=14)
plt.axis([1, 100, 0, 100])

plt.subplot(122)
plt.plot(range(1, 101), times[:, 0], "r--", label="K-Means")
plt.plot(range(1, 101), times[:, 1], "b.-", label="Mini-batch K-Means")
plt.xlabel("$k$", fontsize=16)
plt.title("Training time (seconds)", fontsize=14)
plt.axis([1, 100, 0, 6])

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

```
Saving figure minibatch_kmeans_vs_kmeans
```





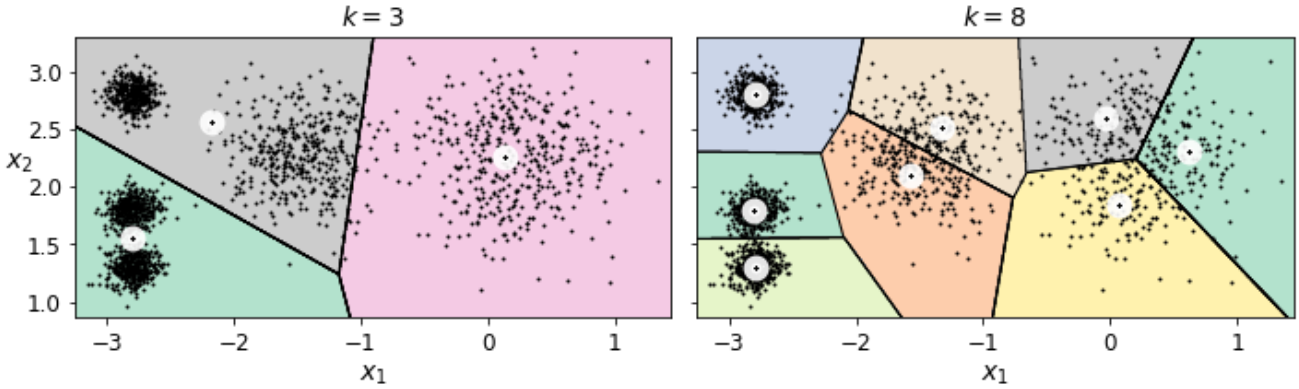
Encontrar el número óptimo de clústeres

¿Qué sucede si el número de clústeres se estableció en un valor menor o mayor que 5?

In [59]:

```
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, estos dos modelos no se ven muy bien. ¿Qué pasa con sus inercias?

In [60]:

```
kmeans_k3.inertia_
```

Out[60]:

```
653.2167190021553
```

In [61]:

```
kmeans_k8.inertia_
```

Out[61]:

```
118.41983763508077
```

No, no podemos simplemente tomar el valor de k que minimiza la inercia, ya que sigue disminuyendo a medida que aumentamos k. De hecho, cuantos más clústeres haya, más cerca estará cada instancia de su centroide más cercano y, por lo tanto, menor será la inercia. Sin embargo, podemos graficar la inercia en función de k y analizar la curva resultante:

In [62]:

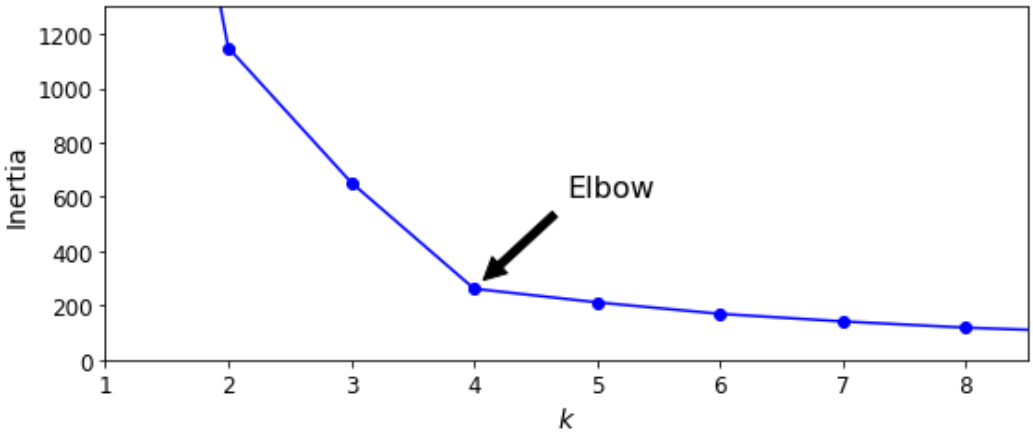
```
kmeans_per_k = [KMeans(n_clusters=k, random_state=42).fit(X)
                for k in range(1, 10)]
inertias = [model.inertia_ for model in kmeans_per_k]
```

In [63]:

```
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()
```

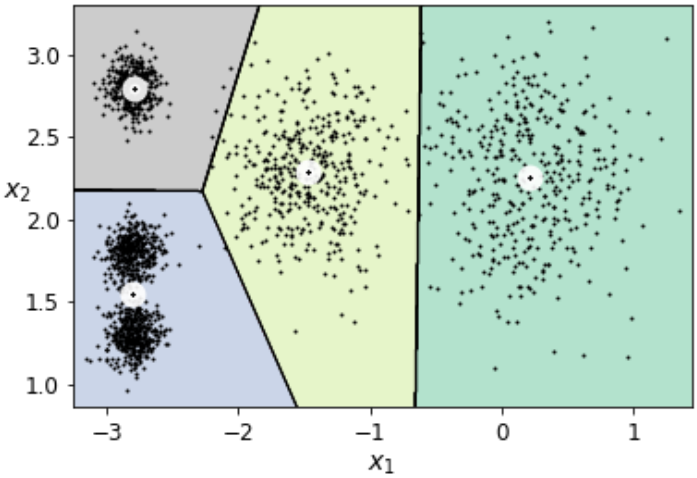
```
Saving figure inertia_vs_k_plot
```



**Como puede ver, hay un codo en $k = 4$
, lo que significa que menos grupos que eso sería malo, y más grupos no ayudarían mucho y podrían reducir los grupos a la mitad. Entonces $k = 4$
es una muy buena opción. Por supuesto, en este ejemplo no es perfecto, ya que significa que los dos blobs en la parte inferior izquierda se considerarán como un solo grupo, pero de todos modos es un agrupamiento bastante bueno.**

In [64]:

```
plot_decision_boundaries(kmeans_per_k[4-1], X)
plt.show()
```



**Otro enfoque es observar la *puntuación de silueta* , que es el *coeficiente de silueta* medio de todas las instancias. El coeficiente de silueta de una instancia es igual a $(b - a)/ \max(a, b)$
donde a
es la distancia media a las otras instancias en el mismo clúster (es la *distancia media dentro del clúster*), y b
es la *distancia media del clúster más cercano* , es decir, la distancia media a las instancias del siguiente clúster más cercano (definido como el que minimiza b
, excluyendo el propio clúster de la instancia). El coeficiente de silueta puede variar entre -1 y +1: un coeficiente cercano a +1 significa que la instancia está bien dentro de su propio clúster y lejos de otros clústeres, mientras**

que un coeficiente cercano a 0 significa que está cerca de un límite de clúster, y finalmente un coeficiente cercano a -1 significa que la instancia puede haber sido asignada al clúster equivocado.

Grafiquemos la puntuación de la silueta en función de k
:

In [65]:

```
from sklearn.metrics import silhouette_score
```

In [66]:

```
silhouette_score(X, kmeans.labels_)
```

Out[66]:

```
0.655517642572828
```

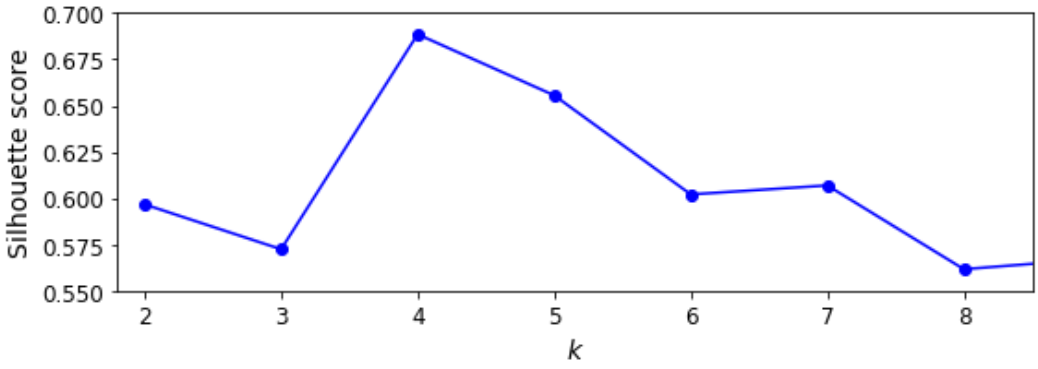
In [67]:

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

In [68]:

```
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
```



Como puede ver, esta visualización es mucho más rica que la anterior: en particular, aunque confirma que $k = 4$ **es una muy buena opción, también subraya el hecho de que** $k = 5$
también es bastante bueno. .

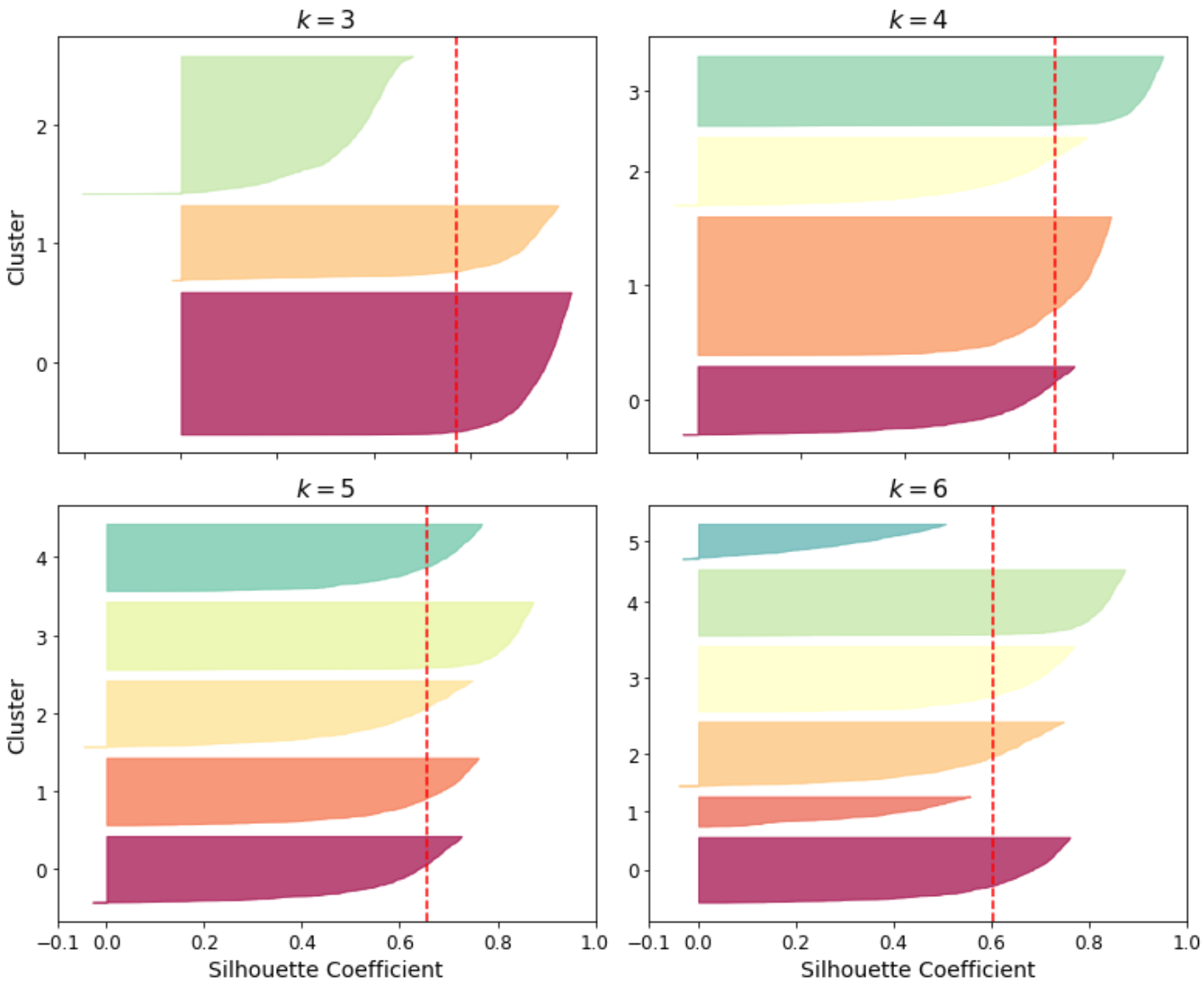
Se proporciona una visualización aún más informativa cuando traza el coeficiente de silueta de cada instancia, ordenada por el grupo al que están asignadas y por el valor del coeficiente. Esto se llama un *diagrama de silueta*
:

In [69]:

```
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
```



Como puede ver, $k = 5$
parece la mejor opción aquí, ya que todos los grupos tienen aproximadamente el mismo tamaño y todos cruzan la línea discontinua, que representa la puntuación media de la silueta.

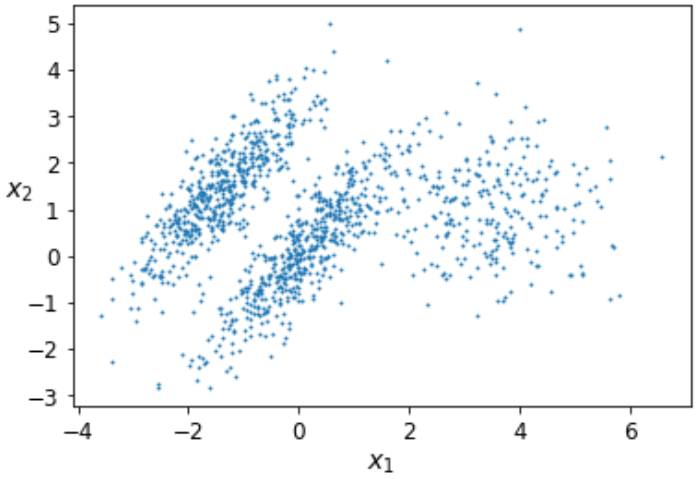
Límites de K-Means

In [70]:

```
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 [71]:

```
plot_clusters(X)
```



In [72]:

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

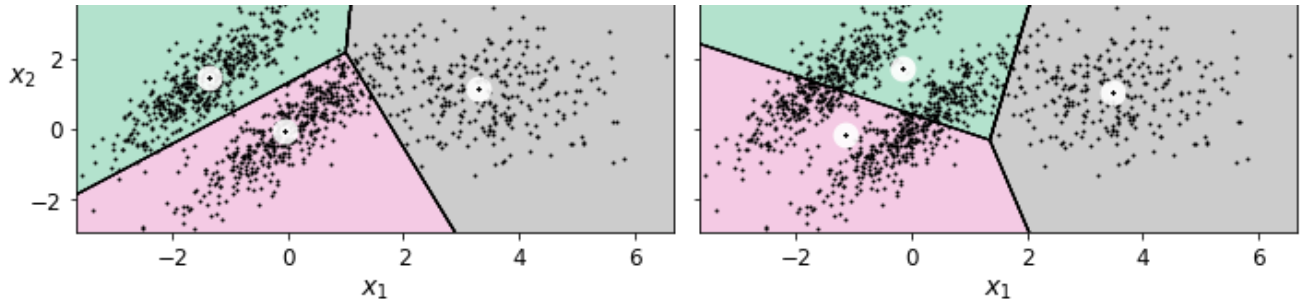
Out[72]:

```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
       n_clusters=3, n_init=10, n_jobs=None, precompute_distances='auto',
       random_state=42, tol=0.0001, verbose=0)
```

In [73]:

```
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
```



Uso de agrupamiento para la segmentación de imágenes

In [74]:

```
# Descarga la imagen de la mariquita
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
```

Out[74]:

```
('./images/unsupervised_learning/ladybug.png',
 <http.client.HTTPMessage at 0x7fa69fb0c0d0>)
```

In [75]:

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

Out[75]:

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

In [76]:

```
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 [77]:

```
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))
```

In [78]:

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

plt.subplot(231)
plt.imshow(image)
plt.title("Original image")
plt.axis('off')

for idx, n_clusters in enumerate(n_colors):
    plt.subplot(232 + idx)
    plt.imshow(segmented_imgs[idx])
```

```
    plt.title("{} colors".format(n_clusters))
    plt.axis('off')

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

```
Saving figure image_segmentation_diagram
```

Uso de la agrupación en clústeres para el preprocesamiento

Abordemos el conjunto de *datos de dígitos* , que es un conjunto de datos simple similar a MNIST que contiene 1797 imágenes de 8 × 8 en escala de grises que representan los dígitos 0 a 9.

In [79]:

```
from sklearn.datasets import load_digits
```

In [80]:

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

Dividámoslo en un conjunto de entrenamiento y un conjunto de prueba:

In [81]:

```
from sklearn.model_selection import train_test_split
```

In [82]:

```
X_train, X_test, y_train, y_test = train_test_split(X_digits, y_digits, random_state=42)
```

Ahora ajustemos un modelo de regresión logística y evaluémoslo en el conjunto de prueba:

In [83]:

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

In [84]:

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

Out[84]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=5000,
                   multi_class='ovr', n_jobs=None, penalty='l2',
                   random_state=42, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

In [85]:

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

Out[85]:

```
0.9688888888888889
```

De acuerdo, esa es nuestra línea de base: 96,89 % de precisión. Veamos si podemos hacerlo mejor usando K-Means como un paso de preprocesamiento. Crearemos una tubería que primero agrupará el conjunto de entrenamiento en 50 grupos y reemplazará las imágenes con sus distancias a los 50 grupos, luego aplicará un modelo de regresión logística:

In [86]:

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

In [87]:

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

Out[87]:

```
Pipeline(memory=None,
         steps=[('kmeans',
                 KMeans(algorithm='auto', copy_x=True, init='k-means++',
                        max_iter=300, n_clusters=50, n_init=10, n_jobs=None,
                        precompute_distances='auto', random_state=42,
                        tol=0.0001, verbose=0)),
                ('log_reg',
                 LogisticRegression(C=1.0, class_weight=None, dual=False,
                                    fit_intercept=True, intercept_scaling=1,
                                    l1_ratio=None, max_iter=5000,
                                    multi_class='ovr', n_jobs=None,
                                    penalty='l2', random_state=42,
                                    solver='lbfgs', tol=0.0001, verbose=0,
                                    warm_start=False))],
         verbose=False)
```

In [88]:

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

Out[88]:

```
0.98
```

¿Cuánto bajó la tasa de error?

In [89]:

```
1 - (1 - pipeline_score) / (1 - log_reg_score)
```

Out[89]:

```
0.3571428571428561
```

¿Qué hay sobre eso? ¡Redujimos la tasa de error en más del 35%! Pero elegimos el número de grupos k de manera completamente arbitraria, seguramente podemos hacerlo mejor. Dado que K-Means es solo un paso de preprocesamiento en una tubería de clasificación, encontrar un buen valor para k es mucho más simple que antes: no es necesario realizar un análisis de silueta o minimizar la inercia, el mejor valor de k

valor de k
es simplemente el uno que resulte en el mejor rendimiento de clasificación.

In [90]:

```
from sklearn.model_selection import GridSearchCV
```

Advertencia : la siguiente celda puede tardar cerca de 20 minutos en ejecutarse, o más, dependiendo de su hardware.

In [91]:

```
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 ............................................
[CV] ............................ kmeans__n_clusters=2, total=   0.1s
[CV] kmeans__n_clusters=2 ............................................
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.1s remaining:    0.0s
[CV] ............................ kmeans__n_clusters=2, total=   0.1s
[CV] kmeans__n_clusters=2 ............................................
[CV] ............................ kmeans__n_clusters=2, total=   0.1s
[CV] kmeans__n_clusters=3 ............................................
[CV] ............................ kmeans__n_clusters=3, total=   0.1s
[CV] kmeans__n_clusters=3 ............................................
[CV] ............................ kmeans__n_clusters=3, total=   0.2s
[CV] kmeans__n_clusters=3 ............................................
[CV] ............................ kmeans__n_clusters=3, total=   0.1s
[CV] kmeans__n_clusters=4 ............................................
[CV] ............................ kmeans__n_clusters=4, total=   0.2s
[CV] kmeans__n_clusters=4 ............................................
[CV] ............................ kmeans__n_clusters=4, total=   0.2s
[CV] kmeans__n_clusters=4 ............................................
[CV] ............................ kmeans__n_clusters=4, total=   0.2s
[CV] kmeans__n_clusters=5 ............................................
[CV] ............................ kmeans__n_clusters=5, total=   0.2s
[CV] kmeans__n_clusters=5 ............................................
[CV] ............................ kmeans__n_clusters=5, total=   0.2s
[CV] kmeans__n_clusters=5 ............................................
[CV] ............................ kmeans__n_clusters=5, total=   0.2s
[CV] kmeans__n_clusters=6 ............................................
[CV] ............................ kmeans__n_clusters=6, total=   0.2s
[CV] kmeans__n_clusters=6 ............................................
[CV] ............................ kmeans__n_clusters=6, total=   0.2s
[CV] kmeans__n_clusters=6 ............................................
[CV] ............................ kmeans__n_clusters=6, total=   0.2s
[CV] kmeans__n_clusters=7 ............................................
[CV] ............................ kmeans__n_clusters=7, total=   0.3s
[CV] kmeans__n_clusters=7 ............................................
[CV] ............................ kmeans__n_clusters=7, total=   0.3s
[CV] kmeans__n_clusters=7 ............................................
[CV] ............................ kmeans__n_clusters=7, total=   0.3s
[CV] kmeans__n_clusters=8 ............................................
[CV] ............................ kmeans__n_clusters=8, total=   0.3s
[CV] kmeans__n_clusters=8 ............................................
[CV] ............................ kmeans__n_clusters=8, total=   0.3s
[CV] kmeans__n_clusters=8 ............................................
[CV] ............................ kmeans__n_clusters=8, total=   0.4s
[CV] kmeans__n_clusters=9 ............................................
[CV] ............................ kmeans__n_clusters=9, total=   0.4s
[CV] kmeans__n_clusters=9 ............................................
[CV] ............................ kmeans__n_clusters=9, total=   0.4s
[CV] kmeans__n_clusters=9 ............................................
[CV] ............................ kmeans__n_clusters=9, total=   0.4s
[CV] kmeans__n_clusters=10 ...........................................
[CV] ........................... kmeans__n_clusters=10, total=   0.5s
[CV] kmeans__n_clusters=10
```

```
[CV] kmeans__n_clusters=10 ...........................................
[CV] ............................ kmeans__n_clusters=10, total=   0.6s
[CV] kmeans__n_clusters=10 ...........................................
[CV] ............................ kmeans__n_clusters=10, total=   0.5s
[CV] kmeans__n_clusters=11 ...........................................
[CV] ............................ kmeans__n_clusters=11, total=   0.6s
[CV] kmeans__n_clusters=11 ...........................................
[CV] ............................ kmeans__n_clusters=11, total=   0.6s
[CV] kmeans__n_clusters=11 ...........................................
[CV] ............................ kmeans__n_clusters=11, total=   0.6s
[CV] kmeans__n_clusters=12 ...........................................
[CV] ............................ kmeans__n_clusters=12, total=   0.7s
[CV] kmeans__n_clusters=12 ...........................................
[CV] ............................ kmeans__n_clusters=12, total=   0.8s
[CV] kmeans__n_clusters=12 ...........................................
[CV] ............................ kmeans__n_clusters=12, total=   0.8s
[CV] kmeans__n_clusters=13 ...........................................
[CV] ............................ kmeans__n_clusters=13, total=   0.9s
[CV] kmeans__n_clusters=13 ...........................................
[CV] ............................ kmeans__n_clusters=13, total=   0.8s
[CV] kmeans__n_clusters=13 ...........................................
[CV] ............................ kmeans__n_clusters=13, total=   0.9s
[CV] kmeans__n_clusters=14 ...........................................
[CV] ............................ kmeans__n_clusters=14, total=   1.0s
[CV] kmeans__n_clusters=14 ...........................................
[CV] ............................ kmeans__n_clusters=14, total=   0.9s
[CV] kmeans__n_clusters=14 ...........................................
[CV] ............................ kmeans__n_clusters=14, total=   1.0s
[CV] kmeans__n_clusters=15 ...........................................
[CV] ............................ kmeans__n_clusters=15, total=   1.2s
[CV] kmeans__n_clusters=15 ...........................................
[CV] ............................ kmeans__n_clusters=15, total=   1.0s
[CV] kmeans__n_clusters=15 ...........................................
[CV] ............................ kmeans__n_clusters=15, total=   1.1s
[CV] kmeans__n_clusters=16 ...........................................
[CV] ............................ kmeans__n_clusters=16, total=   1.3s
[CV] kmeans__n_clusters=16 ...........................................
[CV] ............................ kmeans__n_clusters=16, total=   1.3s
[CV] kmeans__n_clusters=16 ...........................................
[CV] ............................ kmeans__n_clusters=16, total=   1.4s
[CV] kmeans__n_clusters=17 ...........................................
[CV] ............................ kmeans__n_clusters=17, total=   1.4s
[CV] kmeans__n_clusters=17 ...........................................
[CV] ............................ kmeans__n_clusters=17, total=   1.4s
[CV] kmeans__n_clusters=17 ...........................................
[CV] ............................ kmeans__n_clusters=17, total=   2.2s
[CV] kmeans__n_clusters=18 ...........................................
[CV] ............................ kmeans__n_clusters=18, total=   1.8s
[CV] kmeans__n_clusters=18 ...........................................
[CV] ............................ kmeans__n_clusters=18, total=   1.7s
[CV] kmeans__n_clusters=18 ...........................................
[CV] ............................ kmeans__n_clusters=18, total=   1.6s
[CV] kmeans__n_clusters=19 ...........................................
[CV] ............................ kmeans__n_clusters=19, total=   1.8s
[CV] kmeans__n_clusters=19 ...........................................
[CV] ............................ kmeans__n_clusters=19, total=   1.7s
[CV] kmeans__n_clusters=19 ...........................................
[CV] ............................ kmeans__n_clusters=19, total=   1.5s
[CV] kmeans__n_clusters=20 ...........................................
[CV] ............................ kmeans__n_clusters=20, total=   1.9s
[CV] kmeans__n_clusters=20 ...........................................
[CV] ............................ kmeans__n_clusters=20, total=   1.8s
[CV] kmeans__n_clusters=20 ...........................................
[CV] ............................ kmeans__n_clusters=20, total=   1.7s
[CV] kmeans__n_clusters=21 ...........................................
[CV] ............................ kmeans__n_clusters=21, total=   1.8s
[CV] kmeans__n_clusters=21 ...........................................
[CV] ............................ kmeans__n_clusters=21, total=   1.9s
[CV] kmeans__n_clusters=21 ...........................................
[CV] ............................ kmeans__n_clusters=21, total=   1.7s
[CV] kmeans__n_clusters=22 ...........................................
[CV] ............................ kmeans__n_clusters=22, total=   1.8s
[CV] kmeans__n_clusters=22
```

```
[CV] kmeans__n_clusters=22 .....................................
[CV] ......................... kmeans__n_clusters=22, total=   1.9s
[CV] kmeans__n_clusters=22 .....................................
[CV] ......................... kmeans__n_clusters=22, total=   1.9s
[CV] kmeans__n_clusters=23 .....................................
[CV] ......................... kmeans__n_clusters=23, total=   1.9s
[CV] kmeans__n_clusters=23 .....................................
[CV] ......................... kmeans__n_clusters=23, total=   1.8s
[CV] kmeans__n_clusters=23 .....................................
[CV] ......................... kmeans__n_clusters=23, total=   1.8s
[CV] kmeans__n_clusters=24 .....................................
[CV] ......................... kmeans__n_clusters=24, total=   2.1s
[CV] kmeans__n_clusters=24 .....................................
[CV] ......................... kmeans__n_clusters=24, total=   2.0s
[CV] kmeans__n_clusters=24 .....................................
[CV] ......................... kmeans__n_clusters=24, total=   2.1s
[CV] kmeans__n_clusters=25 .....................................
[CV] ......................... kmeans__n_clusters=25, total=   2.0s
[CV] kmeans__n_clusters=25 .....................................
[CV] ......................... kmeans__n_clusters=25, total=   2.1s
[CV] kmeans__n_clusters=25 .....................................
[CV] ......................... kmeans__n_clusters=25, total=   2.0s
[CV] kmeans__n_clusters=26 .....................................
[CV] ......................... kmeans__n_clusters=26, total=   2.0s
[CV] kmeans__n_clusters=26 .....................................
[CV] ......................... kmeans__n_clusters=26, total=   2.2s
[CV] kmeans__n_clusters=26 .....................................
[CV] ......................... kmeans__n_clusters=26, total=   2.0s
[CV] kmeans__n_clusters=27 .....................................
[CV] ......................... kmeans__n_clusters=27, total=   2.2s
[CV] kmeans__n_clusters=27 .....................................
[CV] ......................... kmeans__n_clusters=27, total=   2.2s
[CV] kmeans__n_clusters=27 .....................................
[CV] ......................... kmeans__n_clusters=27, total=   2.1s
[CV] kmeans__n_clusters=28 .....................................
[CV] ......................... kmeans__n_clusters=28, total=   2.6s
[CV] kmeans__n_clusters=28 .....................................
[CV] ......................... kmeans__n_clusters=28, total=   2.4s
[CV] kmeans__n_clusters=28 .....................................
[CV] ......................... kmeans__n_clusters=28, total=   2.3s
[CV] kmeans__n_clusters=29 .....................................
[CV] ......................... kmeans__n_clusters=29, total=   2.4s
[CV] kmeans__n_clusters=29 .....................................
[CV] ......................... kmeans__n_clusters=29, total=   2.5s
[CV] kmeans__n_clusters=29 .....................................
[CV] ......................... kmeans__n_clusters=29, total=   2.2s
[CV] kmeans__n_clusters=30 .....................................
[CV] ......................... kmeans__n_clusters=30, total=   2.4s
[CV] kmeans__n_clusters=30 .....................................
[CV] ......................... kmeans__n_clusters=30, total=   2.6s
[CV] kmeans__n_clusters=30 .....................................
[CV] ......................... kmeans__n_clusters=30, total=   2.1s
[CV] kmeans__n_clusters=31 .....................................
[CV] ......................... kmeans__n_clusters=31, total=   2.4s
[CV] kmeans__n_clusters=31 .....................................
[CV] ......................... kmeans__n_clusters=31, total=   2.4s
[CV] kmeans__n_clusters=31 .....................................
[CV] ......................... kmeans__n_clusters=31, total=   2.1s
[CV] kmeans__n_clusters=32 .....................................
[CV] ......................... kmeans__n_clusters=32, total=   2.7s
[CV] kmeans__n_clusters=32 .....................................
[CV] ......................... kmeans__n_clusters=32, total=   2.7s
[CV] kmeans__n_clusters=32 .....................................
[CV] ......................... kmeans__n_clusters=32, total=   2.4s
[CV] kmeans__n_clusters=33 .....................................
[CV] ......................... kmeans__n_clusters=33, total=   2.8s
[CV] kmeans__n_clusters=33 .....................................
[CV] ......................... kmeans__n_clusters=33, total=   2.9s
[CV] kmeans__n_clusters=33 .....................................
[CV] ......................... kmeans__n_clusters=33, total=   2.5s
[CV] kmeans__n_clusters=34 .....................................
[CV] ......................... kmeans__n_clusters=34, total=   2.7s
[CV] kmeans__n_clusters=34
```

```
[CV] kmeans__n_clusters=34 ....................................
[CV] ........................ kmeans__n_clusters=34, total=   2.5s
[CV] kmeans__n_clusters=34 ....................................
[CV] ........................ kmeans__n_clusters=34, total=   2.7s
[CV] kmeans__n_clusters=35 ....................................
[CV] ........................ kmeans__n_clusters=35, total=   2.6s
[CV] kmeans__n_clusters=35 ....................................
[CV] ........................ kmeans__n_clusters=35, total=   2.8s
[CV] kmeans__n_clusters=35 ....................................
[CV] ........................ kmeans__n_clusters=35, total=   2.6s
[CV] kmeans__n_clusters=36 ....................................
[CV] ........................ kmeans__n_clusters=36, total=   2.5s
[CV] kmeans__n_clusters=36 ....................................
[CV] ........................ kmeans__n_clusters=36, total=   2.7s
[CV] kmeans__n_clusters=36 ....................................
[CV] ........................ kmeans__n_clusters=36, total=   3.0s
[CV] kmeans__n_clusters=37 ....................................
[CV] ........................ kmeans__n_clusters=37, total=   2.6s
[CV] kmeans__n_clusters=37 ....................................
[CV] ........................ kmeans__n_clusters=37, total=   2.4s
[CV] kmeans__n_clusters=37 ....................................
[CV] ........................ kmeans__n_clusters=37, total=   2.8s
[CV] kmeans__n_clusters=38 ....................................
[CV] ........................ kmeans__n_clusters=38, total=   2.7s
[CV] kmeans__n_clusters=38 ....................................
[CV] ........................ kmeans__n_clusters=38, total=   2.5s
[CV] kmeans__n_clusters=38 ....................................
[CV] ........................ kmeans__n_clusters=38, total=   2.7s
[CV] kmeans__n_clusters=39 ....................................
[CV] ........................ kmeans__n_clusters=39, total=   2.9s
[CV] kmeans__n_clusters=39 ....................................
[CV] ........................ kmeans__n_clusters=39, total=   3.1s
[CV] kmeans__n_clusters=39 ....................................
[CV] ........................ kmeans__n_clusters=39, total=   2.9s
[CV] kmeans__n_clusters=40 ....................................
[CV] ........................ kmeans__n_clusters=40, total=   2.9s
[CV] kmeans__n_clusters=40 ....................................
[CV] ........................ kmeans__n_clusters=40, total=   2.8s
[CV] kmeans__n_clusters=40 ....................................
[CV] ........................ kmeans__n_clusters=40, total=   2.6s
[CV] kmeans__n_clusters=41 ....................................
[CV] ........................ kmeans__n_clusters=41, total=   2.8s
[CV] kmeans__n_clusters=41 ....................................
[CV] ........................ kmeans__n_clusters=41, total=   2.6s
[CV] kmeans__n_clusters=41 ....................................
[CV] ........................ kmeans__n_clusters=41, total=   2.7s
[CV] kmeans__n_clusters=42 ....................................
[CV] ........................ kmeans__n_clusters=42, total=   2.7s
[CV] kmeans__n_clusters=42 ....................................
[CV] ........................ kmeans__n_clusters=42, total=   2.9s
[CV] kmeans__n_clusters=42 ....................................
[CV] ........................ kmeans__n_clusters=42, total=   2.8s
[CV] kmeans__n_clusters=43 ....................................
[CV] ........................ kmeans__n_clusters=43, total=   2.9s
[CV] kmeans__n_clusters=43 ....................................
[CV] ........................ kmeans__n_clusters=43, total=   2.9s
[CV] kmeans__n_clusters=43 ....................................
[CV] ........................ kmeans__n_clusters=43, total=   2.7s
[CV] kmeans__n_clusters=44 ....................................
[CV] ........................ kmeans__n_clusters=44, total=   2.8s
[CV] kmeans__n_clusters=44 ....................................
[CV] ........................ kmeans__n_clusters=44, total=   2.9s
[CV] kmeans__n_clusters=44 ....................................
[CV] ........................ kmeans__n_clusters=44, total=   3.1s
[CV] kmeans__n_clusters=45 ....................................
[CV] ........................ kmeans__n_clusters=45, total=   3.0s
[CV] kmeans__n_clusters=45 ....................................
[CV] ........................ kmeans__n_clusters=45, total=   3.0s
[CV] kmeans__n_clusters=45 ....................................
[CV] ........................ kmeans__n_clusters=45, total=   2.9s
[CV] kmeans__n_clusters=46 ....................................
[CV] ........................ kmeans__n_clusters=46, total=   2.8s
[CV] kmeans__n_clusters=46
```

```
[CV] kmeans__n_clusters=46 ..........................................
[CV] ........................... kmeans__n_clusters=46, total=   3.0s
[CV] kmeans__n_clusters=46 ..........................................
[CV] ........................... kmeans__n_clusters=46, total=   3.2s
[CV] kmeans__n_clusters=47 ..........................................
[CV] ........................... kmeans__n_clusters=47, total=   2.6s
[CV] kmeans__n_clusters=47 ..........................................
[CV] ........................... kmeans__n_clusters=47, total=   2.8s
[CV] kmeans__n_clusters=47 ..........................................
[CV] ........................... kmeans__n_clusters=47, total=   2.9s
[CV] kmeans__n_clusters=48 ..........................................
[CV] ........................... kmeans__n_clusters=48, total=   2.9s
[CV] kmeans__n_clusters=48 ..........................................
[CV] ........................... kmeans__n_clusters=48, total=   2.7s
[CV] kmeans__n_clusters=48 ..........................................
[CV] ........................... kmeans__n_clusters=48, total=   2.9s
[CV] kmeans__n_clusters=49 ..........................................
[CV] ........................... kmeans__n_clusters=49, total=   2.9s
[CV] kmeans__n_clusters=49 ..........................................
[CV] ........................... kmeans__n_clusters=49, total=   3.0s
[CV] kmeans__n_clusters=49 ..........................................
[CV] ........................... kmeans__n_clusters=49, total=   2.9s
[CV] kmeans__n_clusters=50 ..........................................
[CV] ........................... kmeans__n_clusters=50, total=   2.7s
[CV] kmeans__n_clusters=50 ..........................................
[CV] ........................... kmeans__n_clusters=50, total=   3.1s
[CV] kmeans__n_clusters=50 ..........................................
[CV] ........................... kmeans__n_clusters=50, total=   3.1s
[CV] kmeans__n_clusters=51 ..........................................
[CV] ........................... kmeans__n_clusters=51, total=   2.9s
[CV] kmeans__n_clusters=51 ..........................................
[CV] ........................... kmeans__n_clusters=51, total=   3.1s
[CV] kmeans__n_clusters=51 ..........................................
[CV] ........................... kmeans__n_clusters=51, total=   3.0s
[CV] kmeans__n_clusters=52 ..........................................
[CV] ........................... kmeans__n_clusters=52, total=   3.0s
[CV] kmeans__n_clusters=52 ..........................................
[CV] ........................... kmeans__n_clusters=52, total=   2.9s
[CV] kmeans__n_clusters=52 ..........................................
[CV] ........................... kmeans__n_clusters=52, total=   3.0s
[CV] kmeans__n_clusters=53 ..........................................
[CV] ........................... kmeans__n_clusters=53, total=   2.7s
[CV] kmeans__n_clusters=53 ..........................................
[CV] ........................... kmeans__n_clusters=53, total=   3.9s
[CV] kmeans__n_clusters=53 ..........................................
[CV] ........................... kmeans__n_clusters=53, total=   3.6s
[CV] kmeans__n_clusters=54 ..........................................
[CV] ........................... kmeans__n_clusters=54, total=   2.8s
[CV] kmeans__n_clusters=54 ..........................................
[CV] ........................... kmeans__n_clusters=54, total=   3.7s
[CV] kmeans__n_clusters=54 ..........................................
[CV] ........................... kmeans__n_clusters=54, total=   3.2s
[CV] kmeans__n_clusters=55 ..........................................
[CV] ........................... kmeans__n_clusters=55, total=   3.3s
[CV] kmeans__n_clusters=55 ..........................................
[CV] ........................... kmeans__n_clusters=55, total=   3.3s
[CV] kmeans__n_clusters=55 ..........................................
[CV] ........................... kmeans__n_clusters=55, total=   3.0s
[CV] kmeans__n_clusters=56 ..........................................
[CV] ........................... kmeans__n_clusters=56, total=   3.0s
[CV] kmeans__n_clusters=56 ..........................................
[CV] ........................... kmeans__n_clusters=56, total=   3.6s
[CV] kmeans__n_clusters=56 ..........................................
[CV] ........................... kmeans__n_clusters=56, total=   3.5s
[CV] kmeans__n_clusters=57 ..........................................
[CV] ........................... kmeans__n_clusters=57, total=   3.6s
[CV] kmeans__n_clusters=57 ..........................................
[CV] ........................... kmeans__n_clusters=57, total=   3.4s
[CV] kmeans__n_clusters=57 ..........................................
[CV] ........................... kmeans__n_clusters=57, total=   3.1s
[CV] kmeans__n_clusters=58 ..........................................
[CV] ........................... kmeans__n_clusters=58, total=   3.1s
[CV] kmeans__n_clusters=58
```

```
[CV] kmeans__n_clusters=58 ..........................................
[CV] ........................... kmeans__n_clusters=58, total=   3.1s
[CV] kmeans__n_clusters=58 ..........................................
[CV] ........................... kmeans__n_clusters=58, total=   2.9s
[CV] kmeans__n_clusters=59 ..........................................
[CV] ........................... kmeans__n_clusters=59, total=   3.3s
[CV] kmeans__n_clusters=59 ..........................................
[CV] ........................... kmeans__n_clusters=59, total=   3.0s
[CV] kmeans__n_clusters=59 ..........................................
[CV] ........................... kmeans__n_clusters=59, total=   2.8s
[CV] kmeans__n_clusters=60 ..........................................
[CV] ........................... kmeans__n_clusters=60, total=   3.1s
[CV] kmeans__n_clusters=60 ..........................................
[CV] ........................... kmeans__n_clusters=60, total=   3.0s
[CV] kmeans__n_clusters=60 ..........................................
[CV] ........................... kmeans__n_clusters=60, total=   3.0s
[CV] kmeans__n_clusters=61 ..........................................
[CV] ........................... kmeans__n_clusters=61, total=   3.2s
[CV] kmeans__n_clusters=61 ..........................................
[CV] ........................... kmeans__n_clusters=61, total=   3.3s
[CV] kmeans__n_clusters=61 ..........................................
[CV] ........................... kmeans__n_clusters=61, total=   2.7s
[CV] kmeans__n_clusters=62 ..........................................
[CV] ........................... kmeans__n_clusters=62, total=   3.2s
[CV] kmeans__n_clusters=62 ..........................................
[CV] ........................... kmeans__n_clusters=62, total=   3.3s
[CV] kmeans__n_clusters=62 ..........................................
[CV] ........................... kmeans__n_clusters=62, total=   3.1s
[CV] kmeans__n_clusters=63 ..........................................
[CV] ........................... kmeans__n_clusters=63, total=   3.1s
[CV] kmeans__n_clusters=63 ..........................................
[CV] ........................... kmeans__n_clusters=63, total=   3.3s
[CV] kmeans__n_clusters=63 ..........................................
[CV] ........................... kmeans__n_clusters=63, total=   3.2s
[CV] kmeans__n_clusters=64 ..........................................
[CV] ........................... kmeans__n_clusters=64, total=   3.0s
[CV] kmeans__n_clusters=64 ..........................................
[CV] ........................... kmeans__n_clusters=64, total=   3.5s
[CV] kmeans__n_clusters=64 ..........................................
[CV] ........................... kmeans__n_clusters=64, total=   3.7s
[CV] kmeans__n_clusters=65 ..........................................
[CV] ........................... kmeans__n_clusters=65, total=   3.1s
[CV] kmeans__n_clusters=65 ..........................................
[CV] ........................... kmeans__n_clusters=65, total=   3.5s
[CV] kmeans__n_clusters=65 ..........................................
[CV] ........................... kmeans__n_clusters=65, total=   3.6s
[CV] kmeans__n_clusters=66 ..........................................
[CV] ........................... kmeans__n_clusters=66, total=   4.4s
[CV] kmeans__n_clusters=66 ..........................................
[CV] ........................... kmeans__n_clusters=66, total=   4.1s
[CV] kmeans__n_clusters=66 ..........................................
[CV] ........................... kmeans__n_clusters=66, total=   3.0s
[CV] kmeans__n_clusters=67 ..........................................
[CV] ........................... kmeans__n_clusters=67, total=   3.3s
[CV] kmeans__n_clusters=67 ..........................................
[CV] ........................... kmeans__n_clusters=67, total=   3.2s
[CV] kmeans__n_clusters=67 ..........................................
[CV] ........................... kmeans__n_clusters=67, total=   3.4s
[CV] kmeans__n_clusters=68 ..........................................
[CV] ........................... kmeans__n_clusters=68, total=   3.6s
[CV] kmeans__n_clusters=68 ..........................................
[CV] ........................... kmeans__n_clusters=68, total=   3.4s
[CV] kmeans__n_clusters=68 ..........................................
[CV] ........................... kmeans__n_clusters=68, total=   3.3s
[CV] kmeans__n_clusters=69 ..........................................
[CV] ........................... kmeans__n_clusters=69, total=   3.4s
[CV] kmeans__n_clusters=69 ..........................................
[CV] ........................... kmeans__n_clusters=69, total=   3.1s
[CV] kmeans__n_clusters=69 ..........................................
[CV] ........................... kmeans__n_clusters=69, total=   3.4s
[CV] kmeans__n_clusters=70 ..........................................
[CV] ........................... kmeans__n_clusters=70, total=   3.0s
```

```
[CV] kmeans__n_clusters=70 .....................................
[CV] ......................... kmeans__n_clusters=70, total=   3.3s
[CV] kmeans__n_clusters=70 .....................................
[CV] ......................... kmeans__n_clusters=70, total=   3.7s
[CV] kmeans__n_clusters=71 .....................................
[CV] ......................... kmeans__n_clusters=71, total=   3.1s
[CV] kmeans__n_clusters=71 .....................................
[CV] ......................... kmeans__n_clusters=71, total=   3.2s
[CV] kmeans__n_clusters=71 .....................................
[CV] ......................... kmeans__n_clusters=71, total=   3.1s
[CV] kmeans__n_clusters=72 .....................................
[CV] ......................... kmeans__n_clusters=72, total=   3.0s
[CV] kmeans__n_clusters=72 .....................................
[CV] ......................... kmeans__n_clusters=72, total=   3.0s
[CV] kmeans__n_clusters=72 .....................................
[CV] ......................... kmeans__n_clusters=72, total=   2.8s
[CV] kmeans__n_clusters=73 .....................................
[CV] ......................... kmeans__n_clusters=73, total=   3.2s
[CV] kmeans__n_clusters=73 .....................................
[CV] ......................... kmeans__n_clusters=73, total=   3.3s
[CV] kmeans__n_clusters=73 .....................................
[CV] ......................... kmeans__n_clusters=73, total=   3.3s
[CV] kmeans__n_clusters=74 .....................................
[CV] ......................... kmeans__n_clusters=74, total=   3.2s
[CV] kmeans__n_clusters=74 .....................................
[CV] ......................... kmeans__n_clusters=74, total=   3.4s
[CV] kmeans__n_clusters=74 .....................................
[CV] ......................... kmeans__n_clusters=74, total=   3.3s
[CV] kmeans__n_clusters=75 .....................................
[CV] ......................... kmeans__n_clusters=75, total=   3.0s
[CV] kmeans__n_clusters=75 .....................................
[CV] ......................... kmeans__n_clusters=75, total=   3.3s
[CV] kmeans__n_clusters=75 .....................................
[CV] ......................... kmeans__n_clusters=75, total=   3.6s
[CV] kmeans__n_clusters=76 .....................................
[CV] ......................... kmeans__n_clusters=76, total=   3.2s
[CV] kmeans__n_clusters=76 .....................................
[CV] ......................... kmeans__n_clusters=76, total=   3.0s
[CV] kmeans__n_clusters=76 .....................................
[CV] ......................... kmeans__n_clusters=76, total=   2.9s
[CV] kmeans__n_clusters=77 .....................................
[CV] ......................... kmeans__n_clusters=77, total=   3.0s
[CV] kmeans__n_clusters=77 .....................................
[CV] ......................... kmeans__n_clusters=77, total=   3.1s
[CV] kmeans__n_clusters=77 .....................................
[CV] ......................... kmeans__n_clusters=77, total=   3.7s
[CV] kmeans__n_clusters=78 .....................................
[CV] ......................... kmeans__n_clusters=78, total=   3.4s
[CV] kmeans__n_clusters=78 .....................................
[CV] ......................... kmeans__n_clusters=78, total=   3.4s
[CV] kmeans__n_clusters=78 .....................................
[CV] ......................... kmeans__n_clusters=78, total=   3.1s
[CV] kmeans__n_clusters=79 .....................................
[CV] ......................... kmeans__n_clusters=79, total=   3.0s
[CV] kmeans__n_clusters=79 .....................................
[CV] ......................... kmeans__n_clusters=79, total=   3.5s
[CV] kmeans__n_clusters=79 .....................................
[CV] ......................... kmeans__n_clusters=79, total=   3.8s
[CV] kmeans__n_clusters=80 .....................................
[CV] ......................... kmeans__n_clusters=80, total=   3.5s
[CV] kmeans__n_clusters=80 .....................................
[CV] ......................... kmeans__n_clusters=80, total=   3.5s
[CV] kmeans__n_clusters=80 .....................................
[CV] ......................... kmeans__n_clusters=80, total=   3.3s
[CV] kmeans__n_clusters=81 .....................................
[CV] ......................... kmeans__n_clusters=81, total=   3.1s
[CV] kmeans__n_clusters=81 .....................................
[CV] ......................... kmeans__n_clusters=81, total=   3.2s
[CV] kmeans__n_clusters=81 .....................................
[CV] ......................... kmeans__n_clusters=81, total=   3.2s
[CV] kmeans__n_clusters=82 .....................................
[CV] ......................... kmeans__n_clusters=82, total=   3.4s
[CV] kmeans__n_clusters=82
```

```
[CV] kmeans__n_clusters=82 .....................................
[CV] ......................... kmeans__n_clusters=82, total=   3.0s
[CV] kmeans__n_clusters=82 .....................................
[CV] ......................... kmeans__n_clusters=82, total=   3.5s
[CV] kmeans__n_clusters=83 .....................................
[CV] ......................... kmeans__n_clusters=83, total=   3.9s
[CV] kmeans__n_clusters=83 .....................................
[CV] ......................... kmeans__n_clusters=83, total=   3.2s
[CV] kmeans__n_clusters=83 .....................................
[CV] ......................... kmeans__n_clusters=83, total=   3.6s
[CV] kmeans__n_clusters=84 .....................................
[CV] ......................... kmeans__n_clusters=84, total=   3.7s
[CV] kmeans__n_clusters=84 .....................................
[CV] ......................... kmeans__n_clusters=84, total=   3.8s
[CV] kmeans__n_clusters=84 .....................................
[CV] ......................... kmeans__n_clusters=84, total=   2.9s
[CV] kmeans__n_clusters=85 .....................................
[CV] ......................... kmeans__n_clusters=85, total=   3.5s
[CV] kmeans__n_clusters=85 .....................................
[CV] ......................... kmeans__n_clusters=85, total=   3.6s
[CV] kmeans__n_clusters=85 .....................................
[CV] ......................... kmeans__n_clusters=85, total=   3.2s
[CV] kmeans__n_clusters=86 .....................................
[CV] ......................... kmeans__n_clusters=86, total=   3.3s
[CV] kmeans__n_clusters=86 .....................................
[CV] ......................... kmeans__n_clusters=86, total=   3.3s
[CV] kmeans__n_clusters=86 .....................................
[CV] ......................... kmeans__n_clusters=86, total=   3.6s
[CV] kmeans__n_clusters=87 .....................................
[CV] ......................... kmeans__n_clusters=87, total=   3.1s
[CV] kmeans__n_clusters=87 .....................................
[CV] ......................... kmeans__n_clusters=87, total=   3.5s
[CV] kmeans__n_clusters=87 .....................................
[CV] ......................... kmeans__n_clusters=87, total=   3.3s
[CV] kmeans__n_clusters=88 .....................................
[CV] ......................... kmeans__n_clusters=88, total=   3.6s
[CV] kmeans__n_clusters=88 .....................................
[CV] ......................... kmeans__n_clusters=88, total=   3.2s
[CV] kmeans__n_clusters=88 .....................................
[CV] ......................... kmeans__n_clusters=88, total=   3.5s
[CV] kmeans__n_clusters=89 .....................................
[CV] ......................... kmeans__n_clusters=89, total=   3.2s
[CV] kmeans__n_clusters=89 .....................................
[CV] ......................... kmeans__n_clusters=89, total=   3.1s
[CV] kmeans__n_clusters=89 .....................................
[CV] ......................... kmeans__n_clusters=89, total=   3.1s
[CV] kmeans__n_clusters=90 .....................................
[CV] ......................... kmeans__n_clusters=90, total=   3.1s
[CV] kmeans__n_clusters=90 .....................................
[CV] ......................... kmeans__n_clusters=90, total=   3.5s
[CV] kmeans__n_clusters=90 .....................................
[CV] ......................... kmeans__n_clusters=90, total=   3.3s
[CV] kmeans__n_clusters=91 .....................................
[CV] ......................... kmeans__n_clusters=91, total=   3.3s
[CV] kmeans__n_clusters=91 .....................................
[CV] ......................... kmeans__n_clusters=91, total=   3.0s
[CV] kmeans__n_clusters=91 .....................................
[CV] ......................... kmeans__n_clusters=91, total=   3.1s
[CV] kmeans__n_clusters=92 .....................................
[CV] ......................... kmeans__n_clusters=92, total=   3.1s
[CV] kmeans__n_clusters=92 .....................................
[CV] ......................... kmeans__n_clusters=92, total=   3.1s
[CV] kmeans__n_clusters=92 .....................................
[CV] ......................... kmeans__n_clusters=92, total=   3.0s
[CV] kmeans__n_clusters=93 .....................................
[CV] ......................... kmeans__n_clusters=93, total=   3.0s
[CV] kmeans__n_clusters=93 .....................................
[CV] ......................... kmeans__n_clusters=93, total=   3.3s
[CV] kmeans__n_clusters=93 .....................................
[CV] ......................... kmeans__n_clusters=93, total=   3.1s
[CV] kmeans__n_clusters=94 .....................................
[CV] ......................... kmeans__n_clusters=94, total=   3.6s
[CV] kmeans__n_clusters=94
```

```
[CV] kmeans__n_clusters=94 ...........................................
[CV] ............................ kmeans__n_clusters=94, total=   3.2s
[CV] kmeans__n_clusters=94 ...........................................
[CV] ............................ kmeans__n_clusters=94, total=   2.9s
[CV] kmeans__n_clusters=95 ...........................................
[CV] ............................ kmeans__n_clusters=95, total=   3.1s
[CV] kmeans__n_clusters=95 ...........................................
[CV] ............................ kmeans__n_clusters=95, total=   3.0s
[CV] kmeans__n_clusters=95 ...........................................
[CV] ............................ kmeans__n_clusters=95, total=   3.1s
[CV] kmeans__n_clusters=96 ...........................................
[CV] ............................ kmeans__n_clusters=96, total=   3.1s
[CV] kmeans__n_clusters=96 ...........................................
[CV] ............................ kmeans__n_clusters=96, total=   3.1s
[CV] kmeans__n_clusters=96 ...........................................
[CV] ............................ kmeans__n_clusters=96, total=   2.9s
[CV] kmeans__n_clusters=97 ...........................................
[CV] ............................ kmeans__n_clusters=97, total=   3.1s
[CV] kmeans__n_clusters=97 ...........................................
[CV] ............................ kmeans__n_clusters=97, total=   3.2s
[CV] kmeans__n_clusters=97 ...........................................
[CV] ............................ kmeans__n_clusters=97, total=   3.0s
[CV] kmeans__n_clusters=98 ...........................................
[CV] ............................ kmeans__n_clusters=98, total=   3.1s
[CV] kmeans__n_clusters=98 ...........................................
[CV] ............................ kmeans__n_clusters=98, total=   3.5s
[CV] kmeans__n_clusters=98 ...........................................
[CV] ............................ kmeans__n_clusters=98, total=   2.9s
[CV] kmeans__n_clusters=99 ...........................................
[CV] ............................ kmeans__n_clusters=99, total=   3.1s
[CV] kmeans__n_clusters=99 ...........................................
[CV] ............................ kmeans__n_clusters=99, total=   3.3s
[CV] kmeans__n_clusters=99 ...........................................
[CV] ............................ kmeans__n_clusters=99, total=   3.4s

[Parallel(n_jobs=1)]: Done 294 out of 294 | elapsed: 12.5min finished
```

Out[91]:

```
GridSearchCV(cv=3, error_score=nan,
             estimator=Pipeline(memory=None,
                                steps=[('kmeans',
                                        KMeans(algorithm='auto', copy_x=True,
                                               init='k-means++', max_iter=300,
                                               n_clusters=50, n_init=10,
                                               n_jobs=None,
                                               precompute_distances='auto',
                                               random_state=42, tol=0.0001,
                                               verbose=0)),
                                       ('log_reg',
                                        LogisticRegression(C=1.0,
                                                           class_weight=None,
                                                           dual=False,
                                                           fit_intercept=True,
                                                           intercept_scaling=1,
                                                           l1_ratio=None,
                                                           max_iter=5000,
                                                           multi_class='ovr',
                                                           n_jobs=None,
                                                           penalty='l2',
                                                           random_state=42,
                                                           solver='lbfgs',
                                                           tol=0.0001,
                                                           verbose=0,
                                                           warm_start=False))],
                                verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid={'kmeans__n_clusters': range(2, 100)},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=2)
```

Veamos cuál es el mejor número de clústeres:

In [92]:

```
grid_clf.best_params_
```

Out[92]:

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

In [93]:

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

Out[93]:

```
0.98
```

Uso de la agrupación en clústeres para el aprendizaje semisupervisado

Otro caso de uso para el agrupamiento es en el aprendizaje semisupervisado, cuando tenemos muchas instancias sin etiquetar y muy pocas instancias etiquetadas.

Veamos el rendimiento de un modelo de regresión logística cuando solo tenemos 50 instancias etiquetadas:

In [94]:

```
n_labeled = 50
```

In [95]:

```
log_reg = LogisticRegression(multi_class="ovr", solver="lbfgs", random_state=42)
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
log_reg.score(X_test, y_test)
```

Out[95]:

```
0.8333333333333334
```

Es mucho menos que antes, por supuesto. Veamos cómo podemos hacerlo mejor. Primero, agrupemos el conjunto de entrenamiento en 50 grupos, luego, para cada grupo, busquemos la imagen más cercana al centroide. Llamaremos a estas imágenes las imágenes representativas:

In [96]:

```
k = 50
```

In [97]:

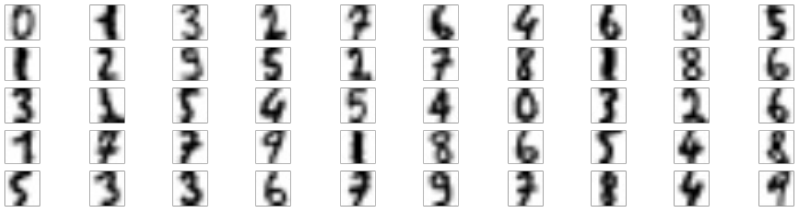
```
kmeans = KMeans(n_clusters=k, random_state=42)
X_digits_dist = kmeans.fit_transform(X_train)
representative_digit_idx = np.argmin(X_digits_dist, axis=0)
X_representative_digits = X_train[representative_digit_idx]
```

Ahora tracemos estas imágenes representativas y etiquetémoslas manualmente:

In [98]:

```
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="bilin
ear")
    plt.axis('off')

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

```
Saving figure representative_images_diagram
```



In [99]:

```
y_train[representative_digit_idx]
```

Out[99]:

```
array([0, 1, 3, 2, 7, 6, 4, 6, 9, 5, 1, 2, 9, 5, 2, 7, 8, 1, 8, 6, 3, 1,
       5, 4, 5, 4, 0, 3, 2, 6, 1, 7, 7, 9, 1, 8, 6, 5, 4, 8, 5, 3, 3, 6,
       7, 9, 7, 8, 4, 9])
```

In [100]:

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

Ahora tenemos un conjunto de datos con solo 50 instancias etiquetadas, pero en lugar de ser instancias completamente aleatorias, cada una de ellas es una imagen representativa de su grupo. A ver si el rendimiento es mejor:

In [101]:

```
log_reg = LogisticRegression(multi_class="ovr", solver="lbfgs", max_iter=5000, random_st
ate=42)
log_reg.fit(X_representative_digits, y_representative_digits)
log_reg.score(X_test, y_test)
```

Out[101]:

```
0.9133333333333333
```

¡Guau! Saltamos del 83,3 % de precisión al 91,3 %, aunque todavía estamos entrenando el modelo solo en 50 instancias. Dado que a menudo es costoso y doloroso etiquetar instancias, especialmente cuando los expertos deben hacerlo manualmente, es una buena idea hacer que etiqueten instancias representativas en lugar de solo instancias aleatorias.

Pero tal vez podamos ir un paso más allá: ¿qué pasa si propagamos las etiquetas a todas las demás instancias en el mismo clúster?

In [102]:

```
y_train_propagated = np.empty(len(X_train), dtype=np.int32)
for i in range(k):
    y_train_propagated[kmeans.labels_==i] = y_representative_digits[i]
```

In [103]:

```
log_reg = LogisticRegression(multi_class="ovr", solver="lbfgs", max_iter=5000, random_st
ate=42)
log_reg.fit(X_train, y_train_propagated)
```

Out[103]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=5000,
                   multi_class='ovr', n_jobs=None, penalty='l2',
```

```
                   random_state=42, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

In [104]:

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

Out[104]:

```
0.9244444444444444
```

Tenemos un pequeño aumento de precisión. Mejor que nada, pero probablemente deberíamos haber propagado las etiquetas solo a las instancias más cercanas al centroide, porque al propagar al clúster completo, ciertamente hemos incluido algunos valores atípicos. Solo propaguemos las etiquetas al percentil 75 más cercano al centroide:

In [105]:

```
percentile_closest = 75

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
```

In [106]:

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

In [107]:

```
log_reg = LogisticRegression(multi_class="ovr", solver="lbfgs", max_iter=5000, random_st
ate=42)
log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
```

Out[107]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=5000,
                   multi_class='ovr', n_jobs=None, penalty='l2',
                   random_state=42, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

In [108]:

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

Out[108]:

```
0.9266666666666666
```

Un poco mejor. Con solo 50 instancias etiquetadas (¡solo 5 ejemplos por clase en promedio!), obtuvimos un rendimiento del 92,7 %, que se acerca al rendimiento de la regresión logística en el conjunto de datos de *dígitos* completamente etiquetados (que fue del 96,9 %).

Esto se debe a que las etiquetas propagadas son bastante buenas: su precisión es cercana al 96 %:

In [109]:

```
np.mean(y_train_partially_propagated == y_train[partially_propagated])
```

Out[109]:

```
0.9592039800995025
```

Ahora podría hacer algunas iteraciones de *aprendizaje activo* :

1. **Etiquete manualmente las instancias de las que el clasificador está menos seguro, si es posible seleccionándolas en grupos distintos.**
2. **Entrena un nuevo modelo con estas etiquetas adicionales.**

DBSCAN

In [110]:

```
from sklearn.datasets import make_moons
```

In [111]:

```
X, y = make_moons(n_samples=1000, noise=0.05, random_state=42)
```

In [112]:

```
from sklearn.cluster import DBSCAN
```

In [113]:

```
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
```

Out[113]:

```
DBSCAN(algorithm='auto', eps=0.05, leaf_size=30, metric='euclidean',
       metric_params=None, min_samples=5, n_jobs=None, p=None)
```

In [114]:

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

Out[114]:

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

In [115]:

```
len(dbscan.core_sample_indices_)
```

Out[115]:

```
808
```

In [116]:

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

Out[116]:

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

In [117]:

```
dbscan.components_[:3]
```

Out[117]:

```
array([[-0.02137124,  0.40618608],
       [-0.84192557,  0.53058695],
       [ 0.58930337, -0.32137599]])
```

In [118]:

```
np.unique(dbscan.labels_)
```

Out[118]:

```
array([-1,  0,  1,  2,  3,  4,  5,  6])
```

In [119]:

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

Out[119]:

```
DBSCAN(algorithm='auto', eps=0.2, leaf_size=30, metric='euclidean',
       metric_params=None, min_samples=5, n_jobs=None, p=None)
```

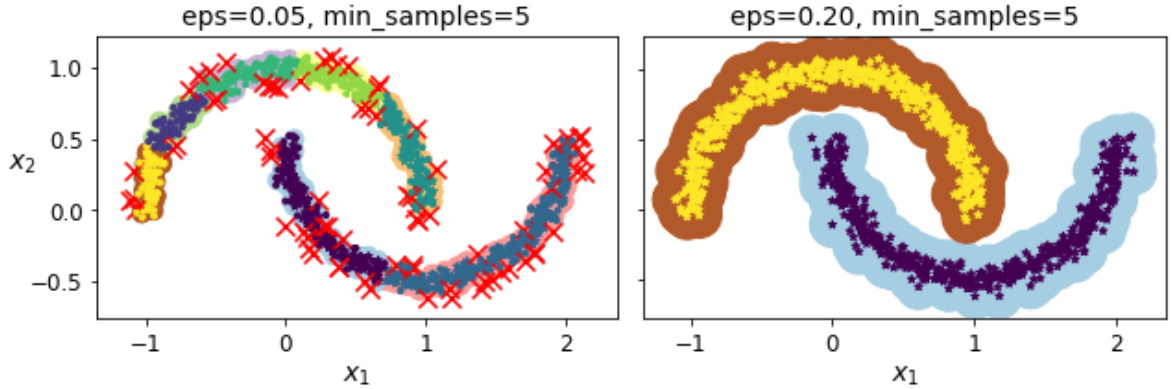
In [120]:

```
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="Paired")
    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], marke
r=".")
    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)
    plt.title("eps={:.2f}, min_samples={}".format(dbscan.eps, dbscan.min_samples), fonts
ize=14)
```

In [121]:

```
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()
```

```
Saving figure dbscan_plot
```



In [122]:

```
dbscan = dbscan2
```

In [123]:

```
from sklearn.neighbors import KNeighborsClassifier
```

In [124]:

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

Out[124]:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=50, p=2,
                     weights='uniform')
```

In [125]:

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

Out[125]:

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

In [126]:

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

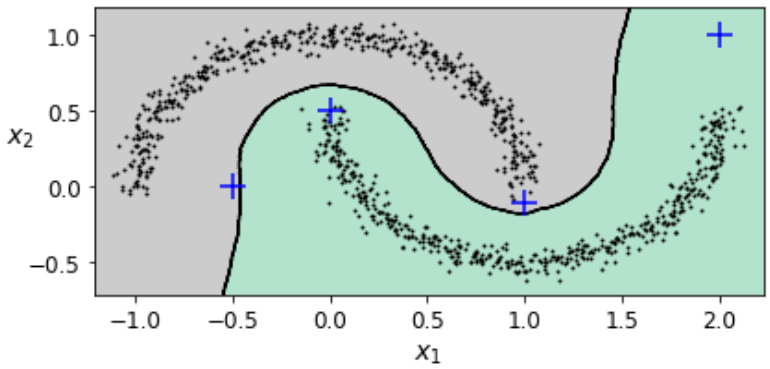
Out[126]:

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

In [127]:

```
plt.figure(figsize=(6, 3))
plot_decision_boundaries(knn, X, show_centroids=False)
plt.scatter(X_new[:, 0], X_new[:, 1], c="b", marker="+", s=200, zorder=10)
save_fig("cluster_classification_plot")
plt.show()
```

```
Saving figure cluster_classification_plot
```



In [128]:

```
y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)
y_pred = dbscan.labels_[dbscan.core_sample_indices_][y_pred_idx]
y_pred[y_dist > 0.2] = -1
y_pred.ravel()
```

Out[128]:

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

Otros algoritmos de agrupamiento

Agrupación espectral

In [129]:

```
from sklearn.cluster import SpectralClustering
```

In [130]:

```
sc1 = SpectralClustering(n_clusters=2, gamma=100, random_state=42)
sc1.fit(X)
```

Out[130]:

```
SpectralClustering(affinity='rbf', assign_labels='kmeans', coef0=1, degree=3,
                   eigen_solver=None, eigen_tol=0.0, gamma=100,
                   kernel_params=None, n_clusters=2, n_components=None,
                   n_init=10, n_jobs=None, n_neighbors=10, random_state=42)
```

In [131]:

```
sc2 = SpectralClustering(n_clusters=2, gamma=1, random_state=42)
sc2.fit(X)
```

Out[131]:

```
SpectralClustering(affinity='rbf', assign_labels='kmeans', coef0=1, degree=3,
                   eigen_solver=None, eigen_tol=0.0, gamma=1,
                   kernel_params=None, n_clusters=2, n_components=None,
                   n_init=10, n_jobs=None, n_neighbors=10, random_state=42)
```

In [132]:

```
np.percentile(sc1.affinity_matrix_, 95)
```

Out[132]:

```
0.04251990648936265
```

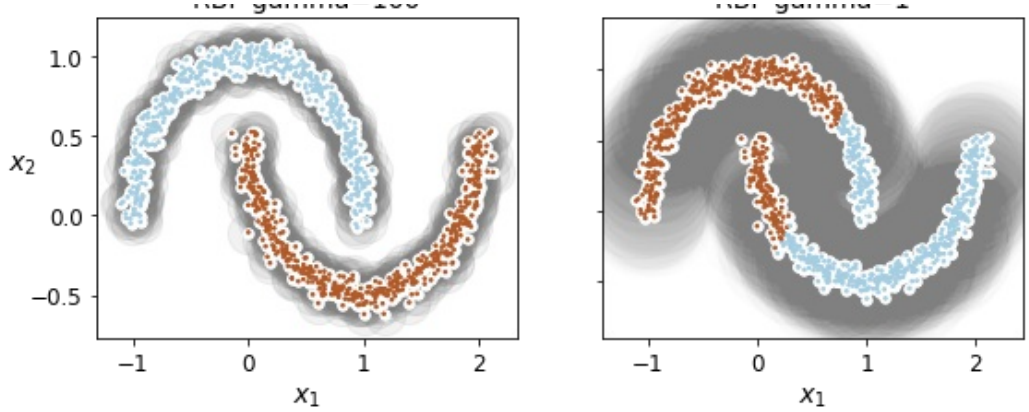
In [133]:

```
def plot_spectral_clustering(sc, X, size, alpha, show_xlabels=True, show_ylabels=True):
    plt.scatter(X[:, 0], X[:, 1], marker='o', s=size, c='gray', cmap="Paired", alpha=alp
ha)
    plt.scatter(X[:, 0], X[:, 1], marker='o', s=30, c='w')
    plt.scatter(X[:, 0], X[:, 1], marker='.', s=10, c=sc.labels_, cmap="Paired")

    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)
    plt.title("RBF gamma={}".format(sc.gamma), fontsize=14)
```

In [134]:

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

plt.subplot(121)
plot_spectral_clustering(sc1, X, size=500, alpha=0.1)

plt.subplot(122)
plot_spectral_clustering(sc2, X, size=4000, alpha=0.01, show_ylabels=False)

plt.show()
```

RBF gamma=100 RBF gamma=1



Agrupación aglomerativa

In [135]:

```
from sklearn.cluster import AgglomerativeClustering
```

In [136]:

```
X = np.array([0, 2, 5, 8.5]).reshape(-1, 1)
agg = AgglomerativeClustering(linkage="complete").fit(X)
```

In [137]:

```
def learned_parameters(estimator):
    return [attrib for attrib in dir(estimator)
            if attrib.endswith("_") and not attrib.startswith("_")]
```

In [138]:

```
learned_parameters(agg)
```

Out[138]:

```
['children_',
 'labels_',
 'n_clusters_',
 'n_components_',
 'n_connected_components_',
 'n_leaves_']
```

In [139]:

```
agg.children_
```

Out[139]:

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

Mezclas Gaussianas

In [140]:

```
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]
```

Entrenemos un modelo de mezcla gaussiana en el conjunto de datos anterior:

In [141]:

```
from sklearn.mixture import GaussianMixture
```

In [142]:

```
gm = GaussianMixture(n_components=3, n_init=10, random_state=42)
gm.fit(X)
```

Out[142]:

```
GaussianMixture(covariance_type='full', init_params='kmeans', max_iter=100,
                means_init=None, n_components=3, n_init=10,
                precisions_init=None, random_state=42, reg_covar=1e-06,
                tol=0.001, verbose=0, verbose_interval=10, warm_start=False,
                weights_init=None)
```

Veamos los parámetros que estimó el algoritmo EM:

In [143]:

```
gm.weights_
```

Out[143]:

```
array([0.39032584, 0.20961444, 0.40005972])
```

In [144]:

```
gm.means_
```

Out[144]:

```
array([[ 0.05145113,  0.07534576],
       [ 3.39947665,  1.05931088],
       [-1.40764129,  1.42712848]])
```

In [145]:

```
gm.covariances_
```

Out[145]:

```
array([[[ 0.68825143,  0.79617956],
        [ 0.79617956,  1.21242183]],

       [[ 1.14740131, -0.03271106],
        [-0.03271106,  0.95498333]],

       [[ 0.63478217,  0.72970097],
        [ 0.72970097,  1.16094925]]])
```

¿El algoritmo realmente convergió?

In [146]:

```
gm.converged_
```

Out[146]:

```
True
```

Si bien. ¿Cuántas iteraciones tomó?

In [147]:

```
gm.n_iter_
```

Out[147]:

```
4
```

Ahora puede usar el modelo para predecir a qué clúster pertenece cada instancia (agrupación dura) o las probabilidades de que provenga de cada clúster. Para esto, solo usa el método `predict()` o el método `predict_proba()` :

In [148]:

```
gm.predict(X)
```

Out[148]:

```
array([0, 0, 2, ..., 1, 1, 1])
```

In [149]:

```
gm.predict_proba(X)
```

Out[149]:

```
array([[9.76815996e-01, 2.31833274e-02, 6.76282339e-07],
       [9.82914418e-01, 1.64110061e-02, 6.74575575e-04],
       [7.52377580e-05, 1.99781831e-06, 9.99922764e-01],
       ...,
       [4.31902443e-07, 9.99999568e-01, 2.12540639e-26],
       [5.20915318e-16, 1.00000000e+00, 1.45002917e-41],
       [2.30971331e-15, 1.00000000e+00, 7.93266114e-41]])
```

Este es un modelo generativo, por lo que puede probar nuevas instancias de él (y obtener sus etiquetas):

In [150]:

```
X_new, y_new = gm.sample(6)
X_new
```

Out[150]:

```
array([[-0.86951041, -0.32742378],
       [ 0.29854504,  0.28307991],
       [ 1.84860618,  2.07374016],
       [ 3.98304484,  1.49869936],
       [ 3.8163406 ,  0.53038367],
       [-1.04030781,  0.78655831]])
```

In [151]:

```
y_new
```

Out[151]:

```
array([0, 0, 1, 1, 1, 2])
```

Observe que se muestrean secuencialmente de cada grupo.

También puedes estimar el logaritmo de la *función de densidad de probabilidad* (PDF) en cualquier lugar usando el método `score_samples()` :

In [152]:

```
gm.score_samples(X)
```

Out[152]:

```
array([-2.60786904, -3.57094519, -3.3302143 , ..., -3.51359636,
       -4.39793229, -3.80725953])
```

Comprobemos que el PDF se integra a 1 en todo el espacio. Simplemente tomamos un cuadrado grande alrededor de los grupos y lo cortamos en una cuadrícula de cuadrados pequeños, luego calculamos la probabilidad aproximada de que las instancias se generen en cada cuadrado pequeño (al multiplicar el PDF en una esquina del cuadrado pequeño por el área del cuadrado), y finalmente sumando todas estas probabilidades). El resultado es muy cercano a 1:

In [153]:

```
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[153]:

```
0.9999999999225095
```

Ahora, tracemos los límites de decisión resultantes (líneas discontinuas) y los contornos de densidad:

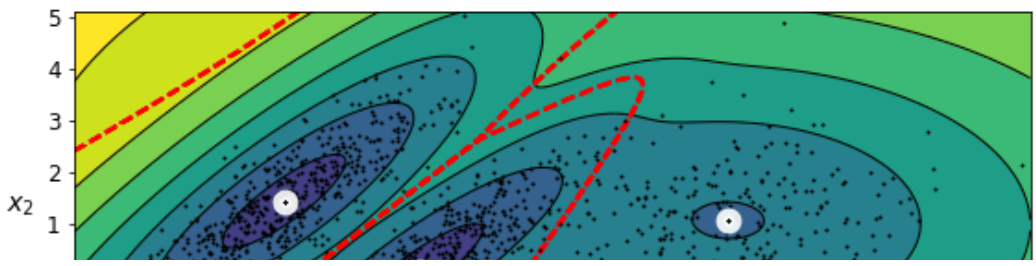
In [154]:

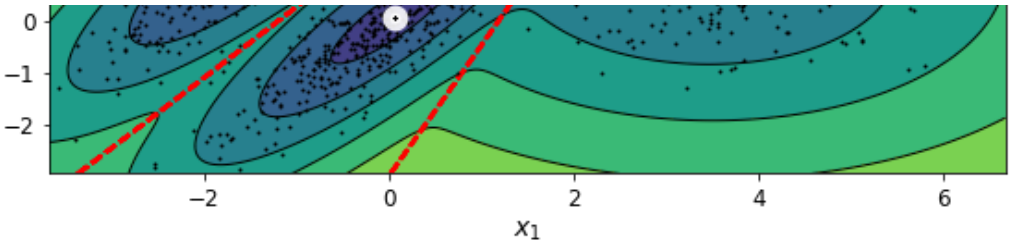
```
from matplotlib.colors import LogNorm

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 [155]:

```
plt.figure(figsize=(8, 4))

plot_gaussian_mixture(gm, X)

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

```
Saving figure gaussian_mixtures_plot
```





Puede imponer restricciones a las matrices de covarianza que busca el algoritmo configurando el hiperparámetro `covariance_type` :

- **`"full"` (predeterminado): sin restricciones, todos los clústeres pueden adoptar cualquier forma elipsoidal de cualquier tamaño.**
- **`"tied"` : todos los grupos deben tener la misma forma, que puede ser cualquier elipsoide (es decir, todos comparten la misma matriz de covarianza).**
- **`"spherical"` : todos los grupos deben ser esféricos, pero pueden tener diferentes diámetros (es decir, diferentes varianzas).**
- **`"diag"` : los clústeres pueden adoptar cualquier forma elipsoidal de cualquier tamaño, pero los ejes del elipsoide deben ser paralelos a los ejes (es decir, las matrices de covarianza deben ser diagonales).**

In [156]:

```
gm_full = GaussianMixture(n_components=3, n_init=10, covariance_type="full", random_stat
e=42)
gm_tied = GaussianMixture(n_components=3, n_init=10, covariance_type="tied", random_stat
e=42)
gm_spherical = GaussianMixture(n_components=3, n_init=10, covariance_type="spherical", r
andom_state=42)
gm_diag = GaussianMixture(n_components=3, n_init=10, covariance_type="diag", random_stat
e=42)
gm_full.fit(X)
gm_tied.fit(X)
gm_spherical.fit(X)
gm_diag.fit(X)
```

Out[156]:

```
GaussianMixture(covariance_type='diag', init_params='kmeans', max_iter=100,
                means_init=None, n_components=3, n_init=10,
                precisions_init=None, random_state=42, reg_covar=1e-06,
                tol=0.001, verbose=0, verbose_interval=10, warm_start=False,
                weights_init=None)
```

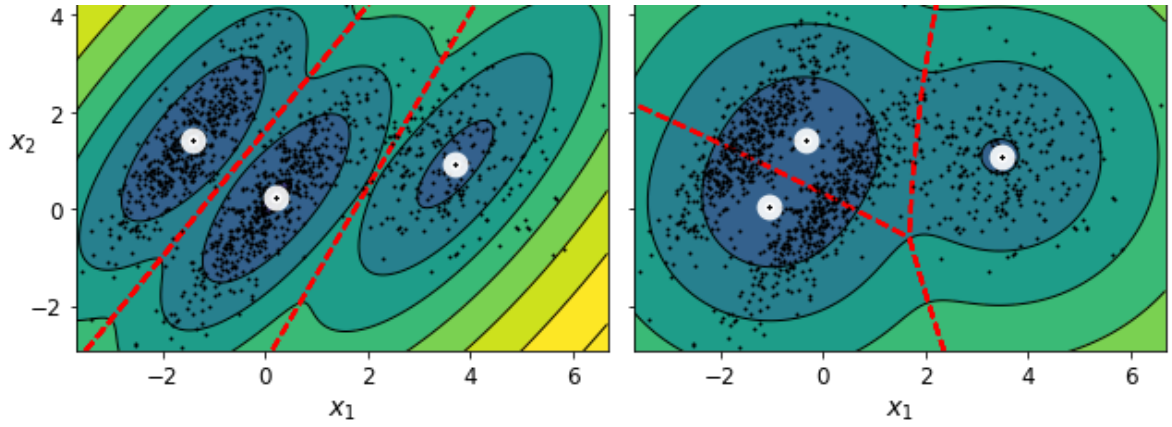
In [157]:

```
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=14)

    plt.subplot(122)
    plot_gaussian_mixture(gm2, X, show_ylabels=False)
    plt.title('covariance_type="{}"'.format(gm2.covariance_type), fontsize=14)
```

In [158]:

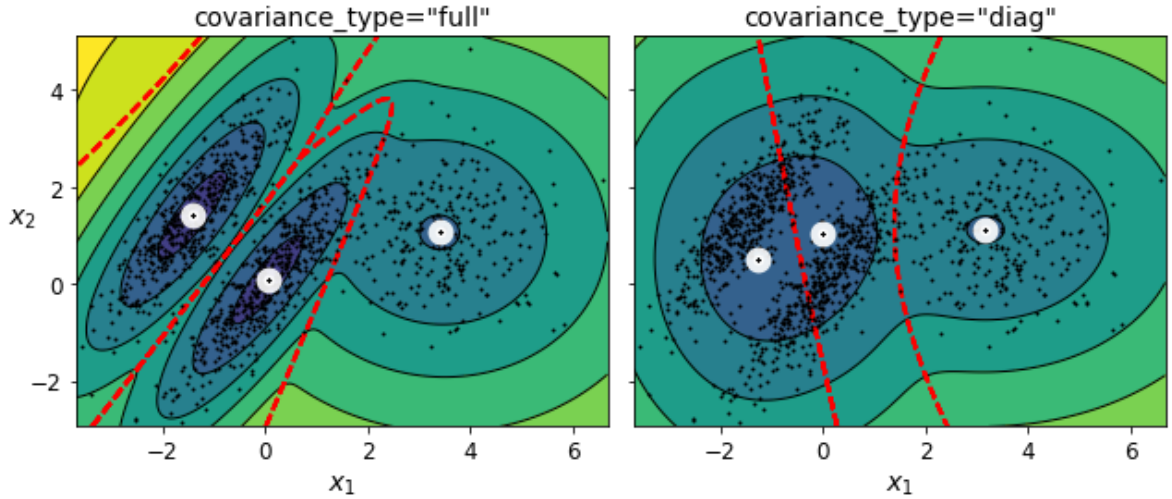
```
compare_gaussian_mixtures(gm_tied, gm_spherical, X)

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

```
Saving figure covariance_type_plot
```



In [159]:

```
compare_gaussian_mixtures(gm_full, gm_diag, X)
plt.tight_layout()
plt.show()
```



Detección de anomalías mediante mezclas gaussianas

Las mezclas gaussianas se pueden utilizar para *la detección de anomalías* : las instancias ubicadas en regiones de baja densidad se pueden considerar anomalías. Debe definir qué umbral de densidad desea utilizar. Por ejemplo, en una empresa de fabricación que trata de detectar productos defectuosos, la proporción de productos defectuosos suele ser bien conocida. Digamos que es igual al 4%, luego puede establecer el umbral de densidad para que sea el valor que resulte en tener el 4% de las instancias ubicadas en áreas por debajo de ese umbral de densidad:

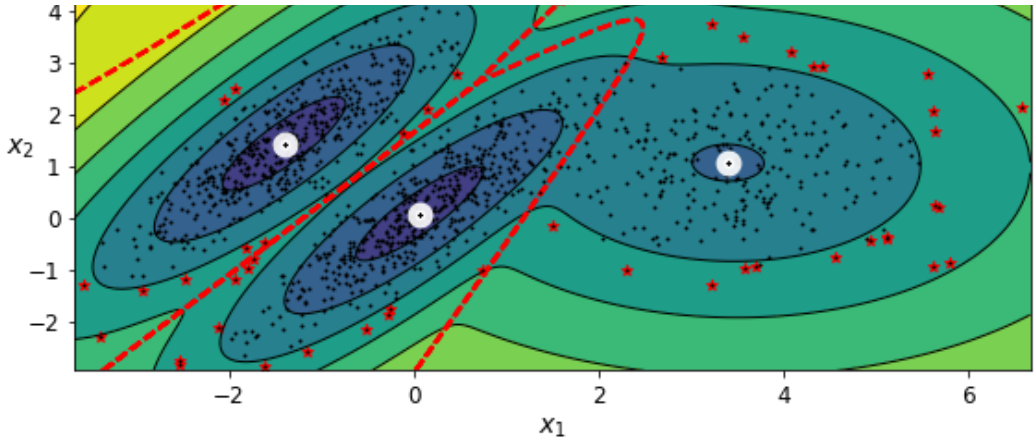
In [160]:

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

In [161]:

```
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
```



Selección del número de clústeres

No podemos usar la inercia o la puntuación de la silueta porque ambas asumen que los cúmulos son esféricos. En su lugar, podemos intentar encontrar el modelo que minimice un criterio de información teórico como el Criterio de Información Bayesiano (BIC) o el Criterio de Información de Akaike (AIC):

$BIC = \log(m)p - 2\log(\sombrero L)$

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

- m
  es el número de instancias.
- p
  es el número de parámetros aprendidos por el modelo.
- $\hat{L}$
  es el valor maximizado de la función de verosimilitud del modelo. Esta es la probabilidad condicional de los datos observados X
  , dado el modelo y sus parámetros optimizados.

  Tanto BIC como AIC penalizan los modelos que tienen más parámetros para aprender (p. ej., más conglomerados) y recompensan los modelos que se ajustan bien a los datos (es decir, modelos que otorgan una alta probabilidad a los datos observados).

In [162]:

```
gm.bic(X)
```

Out[162]:

```
8189.733705221635
```

In [163]:

```
gm.aic(X)
```

Out[163]:

```
8102.508425106597
```

Podríamos calcular el BIC manualmente así:

In [164]:

```
n_clusters = 3
n_dims = 2
n_params_for_weights = n_clusters - 1
n_params_for_means = n_clusters * n_dims
n_params_for_covariance = n_clusters * n_dims * (n_dims + 1) // 2
n_params = n_params_for_weights + n_params_for_means + n_params_for_covariance
max_log_likelihood = gm.score(X) * len(X) # Registro (l^)
```

```
bic = np.log(len(X)) * n_params - 2 * max_log_likelihood
aic = 2 * n_params - 2 * max_log_likelihood
```

In [165]:

```
bic, aic
```

Out[165]:

```
(8189.733705221635, 8102.508425106597)
```

In [166]:

```
n_params
```

Out[166]:

```
17
```

Hay un peso por grupo, pero la suma debe ser igual a 1, por lo que tenemos un grado de libertad menos, de ahí el -1. De manera similar, los grados de libertad para una matriz de covarianza $n \times n$ no son n^2, sino $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$.

Entrenemos modelos de mezcla gaussiana con varios valores de k y midamos su BIC:

In [167]:

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

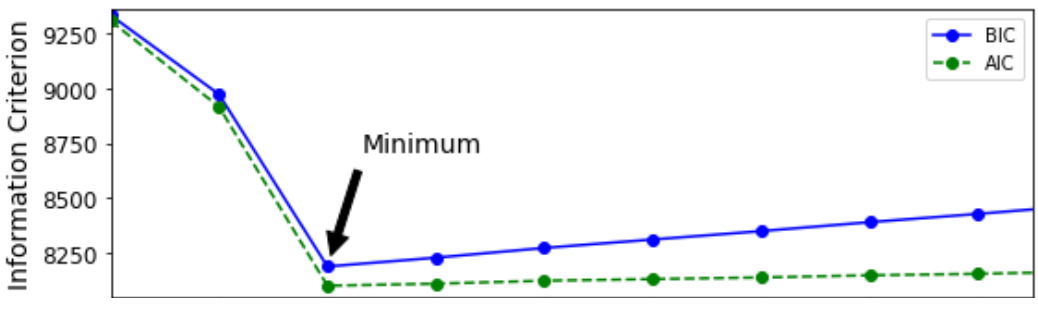
In [168]:

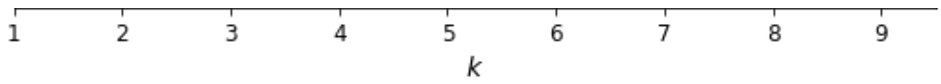
```
bics = [model.bic(X) for model in gms_per_k]
aics = [model.aic(X) for model in gms_per_k]
```

In [169]:

```
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
```





Busquemos la mejor combinación de valores tanto para el número de clústeres como para el hiperparámetro `covariance_type` :

In [170]:

```
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 [171]:

```
best_k
```

Out[171]:

```
3
```

In [172]:

```
best_covariance_type
```

Out[172]:

```
'full'
```

Modelos de mezcla bayesiana gaussiana

En lugar de buscar manualmente el número óptimo de clústeres, es posible utilizar la clase `BayesianGaussianMixture` , que es capaz de otorgar pesos iguales (o cercanos) a cero a los clústeres innecesarios. Simplemente establezca la cantidad de componentes en un valor que crea que es mayor que la cantidad óptima de clústeres y el algoritmo eliminará automáticamente los clústeres innecesarios.

In [173]:

```
from sklearn.mixture import BayesianGaussianMixture
```

In [174]:

```
bgm = BayesianGaussianMixture(n_components=10, n_init=10, random_state=42)
bgm.fit(X)
```

```
/Users/juanba1984/anaconda3/lib/python3.7/site-packages/sklearn/mixture/_base.py:267: Con
vergenceWarning: Initialization 10 did not converge. Try different init parameters, or in
crease max_iter, tol or check for degenerate data.
  % (init + 1), ConvergenceWarning)
```

Out[174]:

```
BayesianGaussianMixture(covariance_prior=None, covariance_type='full',
                        degrees_of_freedom_prior=None, init_params='kmeans',
                        max_iter=100, mean_precision_prior=None,
                        mean_prior=None, n_components=10, n_init=10,
                        random_state=42, reg_covar=1e-06, tol=0.001, verbose=0,
                        verbose_interval=10, warm_start=False,
                        weight_concentration_prior=None,
                        weight_concentration_prior_type='dirichlet_process')
```

El algoritmo detectó automáticamente que solo se necesitan 3 componentes:

In [175]:

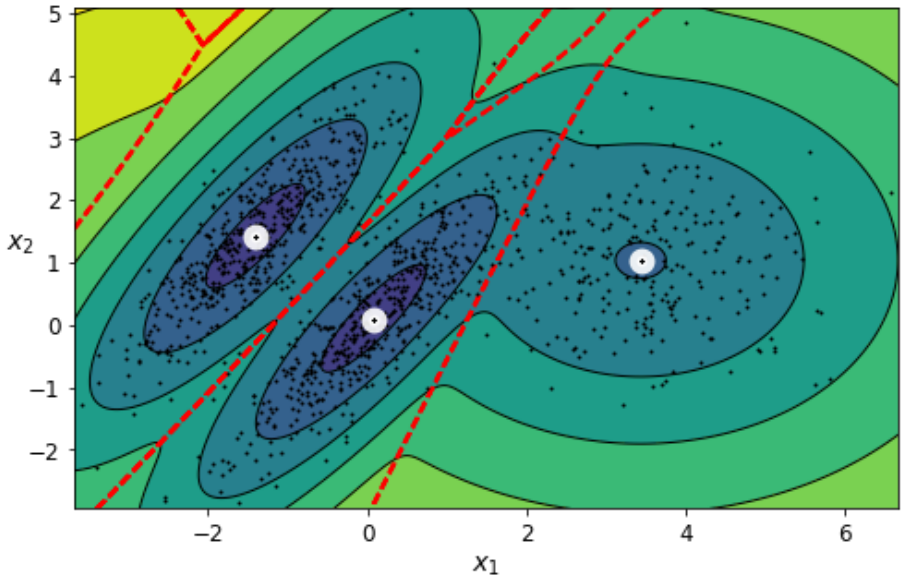
```
np.round(bgm.weights_, 2)
```

Out[175]:

```
array([0.4 , 0.  , 0.  , 0.  , 0.39, 0.2 , 0.  , 0.  , 0.  , 0.  ])
```

In [176]:

```
plt.figure(figsize=(8, 5))
plot_gaussian_mixture(bgm, X)
plt.show()
```



In [177]:

```
bgm_low = BayesianGaussianMixture(n_components=10, max_iter=1000, n_init=1,
                                  weight_concentration_prior=0.01, random_state=42)
bgm_high = BayesianGaussianMixture(n_components=10, max_iter=1000, n_init=1,
                                   weight_concentration_prior=10000, random_state=42)
nn = 73
bgm_low.fit(X[:nn])
bgm_high.fit(X[:nn])
```

Out[177]:

```
BayesianGaussianMixture(covariance_prior=None, covariance_type='full',
                        degrees_of_freedom_prior=None, init_params='kmeans',
                        max_iter=1000, mean_precision_prior=None,
                        mean_prior=None, n_components=10, n_init=1,
                        random_state=42, reg_covar=1e-06, tol=0.001, verbose=0,
                        verbose_interval=10, warm_start=False,
                        weight_concentration_prior=10000,
                        weight_concentration_prior_type='dirichlet_process')
```

In [178]:

```
np.round(bgm_low.weights_, 2)
```

Out[178]:

```
array([0.49, 0.51, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ])
```

In [179]:

```
np.round(bgm_high.weights_, 2)
```

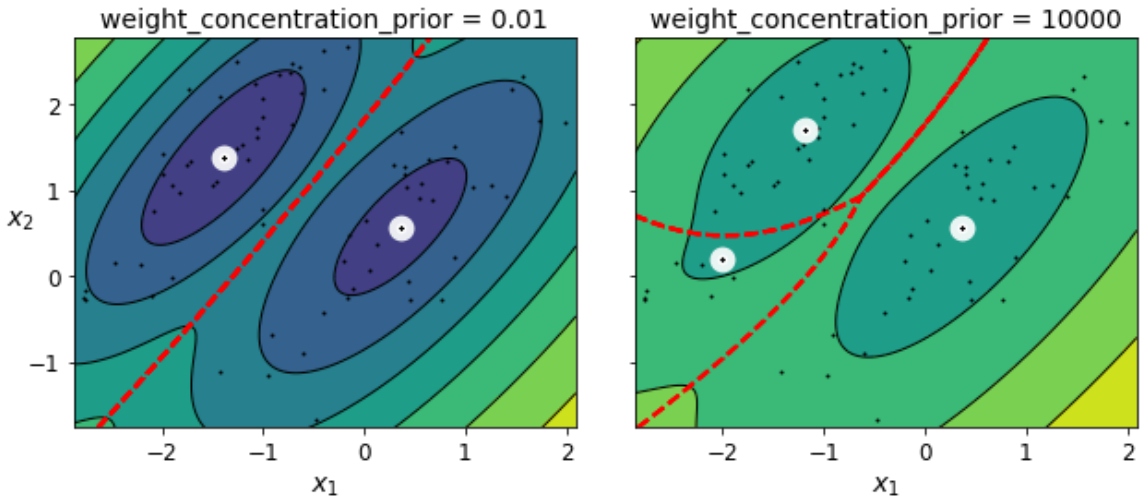
Out[179]:

```
array([0.43, 0.01, 0.01, 0.11, 0.01, 0.01, 0.01, 0.37, 0.01, 0.01])
```

In [180]:

```
plt.figure(figsize=(9, 4))

plt.subplot(121)
plot_gaussian_mixture(bgm_low, X[:nn])
plt.title("weight_concentration_prior = 0.01", fontsize=14)

plt.subplot(122)
plot_gaussian_mixture(bgm_high, X[:nn], show_ylabels=False)
plt.title("weight_concentration_prior = 10000", fontsize=14)

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

```
Saving figure mixture_concentration_prior_plot
```



Nota: el hecho de que solo vea 3 regiones en el diagrama correcto aunque haya 4 centroides no es un error. El peso del conglomerado de arriba a la derecha es mucho mayor que el peso del conglomerado de abajo a la derecha, por lo que la probabilidad de que cualquier punto dado en esta región pertenezca al conglomerado de arriba a la derecha es mayor que la probabilidad de que pertenezca al de abajo a la derecha. grupo.

In [181]:

```
X_moons, y_moons = make_moons(n_samples=1000, noise=0.05, random_state=42)
```

In [182]:

```
bgm = BayesianGaussianMixture(n_components=10, n_init=10, random_state=42)
bgm.fit(X_moons)
```

Out[182]:

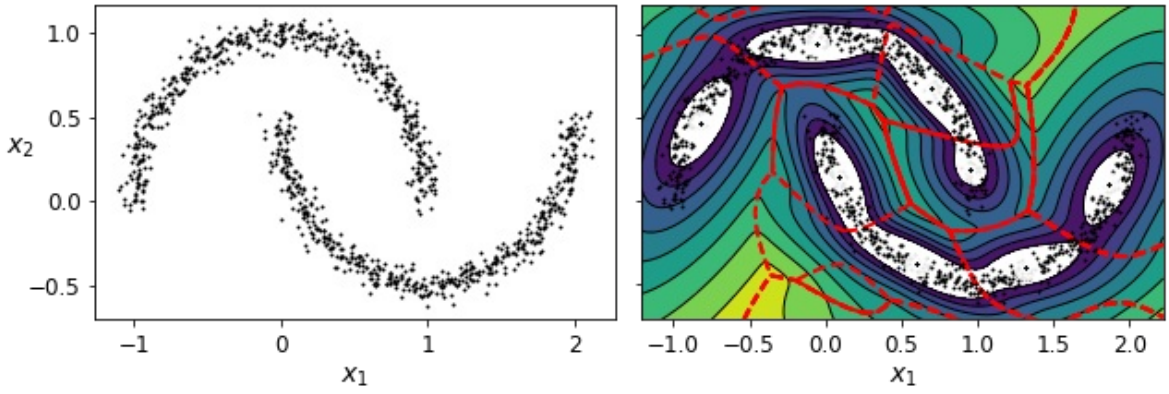
```
BayesianGaussianMixture(covariance_prior=None, covariance_type='full',
                        degrees_of_freedom_prior=None, init_params='kmeans',
                        max_iter=100, mean_precision_prior=None,
                        mean_prior=None, n_components=10, n_init=10,
                        random_state=42, reg_covar=1e-06, tol=0.001, verbose=0,
                        verbose_interval=10, warm_start=False,
                        weight_concentration_prior=None,
                        weight_concentration_prior_type='dirichlet_process')
```

In [183]:

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

plt.subplot(121)
plot_data(X_moons)
plt.xlabel("$x_1$", fontsize=14)
plt.ylabel("$x_2$", fontsize=14, rotation=0)

plt.subplot(122)
```

```
plot_gaussian_mixture(bgm, X_moons, show_ylabels=False)

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

```
Saving figure moons_vs_bgm_plot
```



Vaya, no muy bien... en lugar de detectar 2 cúmulos en forma de luna, el algoritmo detectó 8 cúmulos elipsoidales. Sin embargo, el gráfico de densidad no se ve tan mal, por lo que podría usarse para la detección de anomalías.

Función de probabilidad

In [184]:

```
from scipy.stats import norm
```

In [185]:

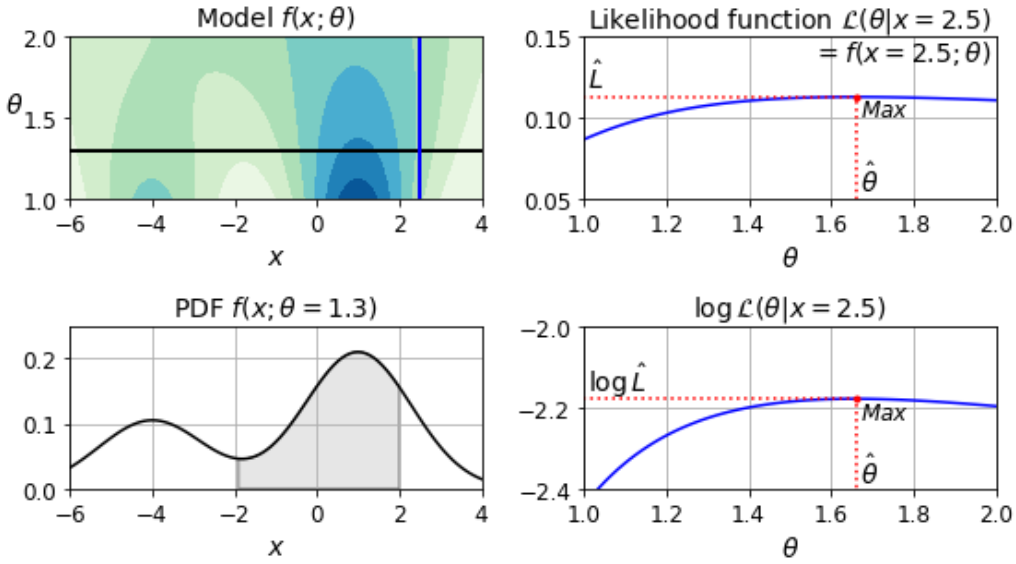
```
xx = np.linspace(-6, 4, 101)
ss = np.linspace(1, 2, 101)
XX, SS = np.meshgrid(xx, ss)
ZZ = 2 * norm.pdf(XX - 1.0, 0, SS) + norm.pdf(XX + 4.0, 0, SS)
ZZ = ZZ / ZZ.sum(axis=1)[:,np.newaxis] / (xx[1] - xx[0])
```

In [186]:

```
from matplotlib.patches import Polygon

plt.figure(figsize=(8, 4.5))

x_idx = 85
s_idx = 30

plt.subplot(221)
plt.contourf(XX, SS, ZZ, cmap="GnBu")
plt.plot([-6, 4], [ss[s_idx], ss[s_idx]], "k-", linewidth=2)
plt.plot([xx[x_idx], xx[x_idx]], [1, 2], "b-", linewidth=2)
plt.xlabel(r"$x$")
plt.ylabel(r"$\theta$", fontsize=14, rotation=0)
plt.title(r"Model $f(x; \theta)$", fontsize=14)

plt.subplot(222)
plt.plot(ss, ZZ[:, x_idx], "b-")
max_idx = np.argmax(ZZ[:, x_idx])
max_val = np.max(ZZ[:, x_idx])
plt.plot(ss[max_idx], max_val, "r.")
plt.plot([ss[max_idx], ss[max_idx]], [0, max_val], "r:")
plt.plot([0, ss[max_idx]], [max_val, max_val], "r:")
plt.text(1.01, max_val + 0.005, r"$\hat{L}$", fontsize=14)
plt.text(ss[max_idx]+ 0.01, 0.055, r"$\hat{\theta}$", fontsize=14)
plt.text(ss[max_idx]+ 0.01, max_val - 0.012, r"$Max$", fontsize=12)
plt.axis([1, 2, 0.05, 0.15])
plt.xlabel(r"$\theta$", fontsize=14)
plt.grid(True)
```

```
plt.text(1.99, 0.135, r"$=f(x=2.5; \theta)$", fontsize=14, ha="right")
plt.title(r"Likelihood function $\mathcal{L}(\theta|x=2.5)$", fontsize=14)

plt.subplot(223)
plt.plot(xx, ZZ[s_idx], "k-")
plt.axis([-6, 4, 0, 0.25])
plt.xlabel(r"$x$", fontsize=14)
plt.grid(True)
plt.title(r"PDF $f(x; \theta=1.3)$", fontsize=14)
verts = [(xx[41], 0)] + list(zip(xx[41:81], ZZ[s_idx, 41:81])) + [(xx[80], 0)]
poly = Polygon(verts, facecolor='0.9', edgecolor='0.5')
plt.gca().add_patch(poly)

plt.subplot(224)
plt.plot(ss, np.log(ZZ[:, x_idx]), "b-")
max_idx = np.argmax(np.log(ZZ[:, x_idx]))
max_val = np.max(np.log(ZZ[:, x_idx]))
plt.plot(ss[max_idx], max_val, "r.")
plt.plot([ss[max_idx], ss[max_idx]], [-5, max_val], "r:")
plt.plot([0, ss[max_idx]], [max_val, max_val], "r:")
plt.axis([1, 2, -2.4, -2])
plt.xlabel(r"$\theta$", fontsize=14)
plt.text(ss[max_idx]+ 0.01, max_val - 0.05, r"$Max$", fontsize=12)
plt.text(ss[max_idx]+ 0.01, -2.39, r"$\hat{\theta}$", fontsize=14)
plt.text(1.01, max_val + 0.02, r"$\log \, \hat{L}$", fontsize=14)
plt.grid(True)
plt.title(r"$\log \, \mathcal{L}(\theta|x=2.5)$", fontsize=14)

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

```
Saving figure likelihood_function_plot
```



In [ ]: