Dimensionality_Reduction

August 31, 2020

Dimensionality Reduction

Some ML problems involve thausands and millions of features. Estimation of such large number can slow, redundant, and sometimes impossible. We will study how can we reduce the number of dimensions without losing a great deal of information. In our study of digit recognition the pixels in the corenrs did not add much information.

if you pick a random point in a unit square (a 1 x 1 square), it will have only about a 0.4% chance of being located less than 0.001 from a border (in other words, it is very unlikely that a random point will be "extreme" along any dimension). But in a 10,000-dimensional unit hypercube (a $1 \times 1 \times \ \times 1$ cube, with ten thousand 1s), this probability is greater than 99.999999%. Most points in a high-dimensional hypercube are very close to the border. Each dimension adds a new border.

if you pick two points randomly in a unit square, the distance between these two points will be, on average, roughly 0.52. If you pick two random points in a unit 3D cube, the average distance will be roughly 0.66. But what about two points picked randomly in a 1,000,000-dimensional hypercube 408.25.

High-dimensional datasets are at risk of being very sparse: most training instances are likely to be far away from each other. A new instance will likely be far away from any training instance, making predictions much less reliable than in lower dimensions, since they will be based on much larger extrapolations. More dimensions -> more risk of overfitting. We can eighter increase the amount of data (often impractical as data needs to grow exponentially with each dimension) or reduce dimensionality. With just 100 features (much less than in the MNIST problem), you would need more training instances than atoms in the observable universe in order for training instances to be within 0.1 of each other on average, assuming they were spread out uniformly across all dimensions.

In most real-world problems, training instances are not spread out uniformly across all dimensions. Many features are almost constant, while others are highly correlated. As a result, all training instances actually lie within (or close to) a much lower-dimensional subspace of the high-dimensional space. (picture projection of 3D on 2D)

1 Manifold

The Swiss roll is an example of a 2D manifold. Put simply, a 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space. More generally, a d-dimensional manifold is a part of an ndimensional space (where $d < n$) that locally resembles a d-dimensional hyperplane. In the case of the Swiss roll, $d = 2$ and $n = 3$: it locally resembles a 2D plane, but it is rolled in the third dimension.

If you randomly generated images, only a ridiculously tiny fraction of them would look like handwritten digits. The degrees of freedom available to you if you try to create a digit image are dramatically lower than the degrees of freedom you would have if you were allowed to generate any image you wanted. If you assume you data is a manifold then the task will be simpler if expressed in the lower-dimensional space of the manifold.

If you reduce the dimensionality of your training set before training a model, it will definitely speed up training, but it may not always lead to a better or simpler solution; it all depends on the dataset.

2 PCA

Principal Component Analysis (PCA) is by far the most popular dimensionality reduction algorithm. You need to choose the right dimensions to select the dimensions that preserves the maximum amount of variance, as it will most likely lose less information than the other projections. Another way to justify this choice is that it is the axis that minimizes the mean squared distance between the original dataset and its projection onto that axis.

3 Setup

First, let's make sure this notebook works well in both python 2 and 3, import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures:

[2]:
```
# Python 3.5 is required
import sys
assert sys.version_info >= (3, 5)

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

# Common imports
import numpy as np
import os

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

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

# Where to save the figures
PROJECT_ROOT_DIR = "."
```

```
CHAPTER_ID = "dim_reduction"
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)

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

4 Projection methods

Build 3D dataset:

[3]:
```
# Create 3D dataset
np.random.seed(4)
m = 60
w1, w2 = 0.1, 0.3
noise = 0.1

angles = np.random.rand(m) * 3 * np.pi / 2 - 0.5
X = np.empty((m, 3))
X[:, 0] = np.cos(angles) + np.sin(angles)/2 + noise * np.random.randn(m) / 2
X[:, 1] = np.sin(angles) * 0.7 + noise * np.random.randn(m) / 2
X[:, 2] = X[:, 0] * w1 + X[:, 1] * w2 + noise * np.random.randn(m)
```

[4]:
```
X[0:10,:]
```

[4]:
```
array([[-1.01570027, -0.55091331, -0.26132626],
       [-0.00771675,  0.59958572,  0.03507755],
       [-0.95317135, -0.46453691, -0.24920288],
       [-0.92012304,  0.21009593,  0.02182381],
       [-0.76309739,  0.158261  ,  0.19152496],
       [ 1.11816122,  0.32508721,  0.31710572],
       [-1.02258878, -0.64384064, -0.13368695],
       [ 0.67351984, -0.27342519, -0.00787835],
       [ 1.01619558,  0.51546608,  0.46783297],
       [ 0.54957723,  0.67728016,  0.2340159 ]])
```

4.1 PCA using SVD decomposition

Note: the `svd()` function returns `U`, `s` and `Vt`, where `Vt` is equal to $\mathbf{V}^T$, the transpose of the matrix $\mathbf{V}$.

$$\mathbf{V} = \begin{pmatrix} | & | & & | \\ \mathbf{c_1} & \mathbf{c_2} & \cdots & \mathbf{c_n} \\ | & | & & | \end{pmatrix}$$

PCA identifies the axis that accounts for the largest amount of variance in the training set. The unit vector that defines the ith axis is called the ith principal component (PC). The 1st. PC is c1 and the 2nd PC is c2. The unit vector that defines the ith axis is called the ith principal component (PC). So how can you find the principal components of a training set? Luckily, there is a standard matrix factorization technique called Singular Value Decomposition (SVD) that can decompose the training set matrix X into the dot product of three matrices

SVD: Singular value decomposition is the generalization of the eigendecomposition of a positive semidefinite normal matrix (for example, a symmetric matrix with positive eigenvalues) to any $m \times n$ matrix via an extension of the polar decomposition. Suppose M is a $m \times n$ matrix:

$$M = U \times \Sigma \times V^T$$

where: * U is $m \times m$ unitary or orthogonal matrix: $U^T U = I$ and $U^T = U^{-1}$ * Σ is a diagnomal $m \times n$ matrix with non-negative real number on the diagonal, which are the singular values (absolute eigenvalues). * V is the $n \times n$ unitary matrix, are the eigenvectors of M.

[5]:
```
# Center the data around zero
X_centered = X - X.mean(axis=0)
# SVD decomposition
U, s, Vt = np.linalg.svd(X_centered)
# Extract first two principal components
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
```

[6]:
```
# extract matrix dimensions
m, n = X.shape
# Create n x m matrix of zeros
S = np.zeros(X_centered.shape)
# substitute the diagonal elements with singular values
S[:n, :n] = np.diag(s)
```

[7]:
```
S[0:3,0:3]
```

[7]:
```
array([[6.77645005, 0.        , 0.        ],
       [0.        , 2.82403671, 0.        ],
       [0.        , 0.        , 0.78116597]])
```

[8]:
```
# Test if the two matrices are roughly equal.
np.allclose(X_centered, U.dot(S).dot(Vt))
```

[8]:
```
True
```

5 Projecting down to d Dimensions

Once you have identified all the principal components, you can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components. Selecting this hyperplane ensures that the projection will preserve as much variance as possible. To project the training set onto the hyperplane, you can simply compute the dot product of the training set matrix X by the matrix W_d, defined as the matrix containing the first d principal components (i.e., the matrix composed of the first d columns of V^T), as shown:

$$X_{d-proj} = X \cdot W_d$$

[9]:
```
# Get the projection matrix for the first two components:
W2 = Vt.T[:, :2]
# Create a projection
X2D = X_centered.dot(W2)
```

[10]:
```
X2D_using_svd = X2D
X2D[0:3,0:3]
```

[10]:
```
array([[-1.26203346, -0.42067648],
       [ 0.08001485,  0.35272239],
       [-1.17545763, -0.36085729]])
```

[11]:
```
X_centered[0:3,0:3]
```

[11]:
```
array([[-1.03976771, -0.76023846, -0.33288048],
       [-0.03178419,  0.39026057, -0.03647667],
       [-0.9772388 , -0.67386206, -0.3207571 ]])
```

5.1 PCA using Scikit-Learn

With Scikit-Learn, PCA is really trivial. It even takes care of mean centering for you:

[12]:
```
from sklearn.decomposition import PCA
# save the model
pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
```

[13]:
```
X2D[:5]
```

[13]:
```
array([[ 1.26203346,  0.42067648],
       [-0.08001485, -0.35272239],
       [ 1.17545763,  0.36085729],
       [ 0.89305601, -0.30862856],
       [ 0.73016287, -0.25404049]])
```

[14]:
```
X2D_using_svd[:5]
```

[14]:
```
array([[-1.26203346, -0.42067648],
       [ 0.08001485,  0.35272239],
       [-1.17545763, -0.36085729],
       [-0.89305601,  0.30862856],
       [-0.73016287,  0.25404049]])
```

[ ]:
```
```

Notice that running PCA multiple times on slightly different datasets may result in different results. In general the only difference is that some axes may be flipped. In this example, PCA using Scikit-Learn gives the same projection as the one given by the SVD approach, except both axes are flipped:

[15]:
```
np.allclose(X2D, -X2D_using_svd)
```

[15]:
```
True
```

Recover the 3D points projected on the plane (PCA 2D subspace).

[16]:
```
X3D_inv = pca.inverse_transform(X2D)
X3D_inv[:5]
```

[16]:
```
array([[-1.01450604, -0.54656333, -0.27441525],
       [-0.02103231,  0.55108376,  0.18101894],
       [-0.95379477, -0.4668077 , -0.24237013],
       [-0.91717404,  0.22083765, -0.01049779],
       [-0.74607229,  0.22027492,  0.00492637]])
```

[17]:
```
X[:5]
```

[17]:
```
array([[-1.01570027, -0.55091331, -0.26132626],
       [-0.00771675,  0.59958572,  0.03507755],
       [-0.95317135, -0.46453691, -0.24920288],
       [-0.92012304,  0.21009593,  0.02182381],
       [-0.76309739,  0.158261  ,  0.19152496]])
```

Of course, there was some loss of information during the projection step, so the recovered 3D points are not exactly equal to the original 3D points:

[18]:
```
np.allclose(X3D_inv, X)
```

[18]:
```
False
```

We can compute the reconstruction error: $\Sigma_{i=1}^{n}(X^i - X_{rec}^i)^2$

[19]:
```
np.mean(np.sum(np.square(X3D_inv - X), axis=1))
```

[19]:
```
0.01017033779284855
```

The inverse transform in the SVD approach looks like this:

[20]:
```
X3D_inv_using_svd = X2D_using_svd.dot(Vt[:2, :])
X3D_inv_using_svd[:5]
```

[20]:
```
array([[-1.03857349, -0.75588848, -0.34596947],
       [-0.04509976,  0.34175861,  0.10946472],
       [-0.97786221, -0.67613285, -0.31392435],
       [-0.94124149,  0.01151249, -0.08205201],
       [-0.77013974,  0.01094977, -0.06662785]])
```

The reconstructions from both methods are not identical because Scikit-Learn's `PCA` class automatically takes care of reversing the mean centering, but if we subtract the mean, we get the same reconstruction:

[21]:
```
np.allclose(X3D_inv_using_svd, X3D_inv - pca.mean_)
```

[21]:
```
True
```

The `PCA` object gives access to the principal components that it computed:

[22]:
```
pca.components_
```

[22]:
```
array([[-0.93636116, -0.29854881, -0.18465208],
       [ 0.34027485, -0.90119108, -0.2684542 ]])
```

Compare to the first two principal components computed using the SVD method:

[23]:
```
Vt[:2]
```

[23]:
```
array([[ 0.93636116,  0.29854881,  0.18465208],
       [-0.34027485,  0.90119108,  0.2684542 ]])
```

Notice how the axes are flipped.

Now let's look at the explained variance ratio. It indicates the proportion of the dataset's variance that lies along the axis of each principal component.

[24]:
```
pca.explained_variance_ratio_
```

[24]:
```
array([0.84248607, 0.14631839])
```

The first dimension explains 84.2% of the variance, while the second explains 14.6%.

By projecting down to 2D, we lost about 1.1% of the variance, so it is reasonable to assume that 3rd axis probably carries little information.

[25]:
```
1 - pca.explained_variance_ratio_.sum()
```

[25]:
```
0.011195535570688975
```

Here is how to compute the explained variance ratio using the SVD approach (recall that `s` is the diagonal of the matrix `S`):

[26]:
```
np.square(s) / np.square(s).sum()
```

[26]:
```
array([0.84248607, 0.14631839, 0.01119554])
```

[27]:
```
# You can specify a target variance to be extracted by principle components.
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X)
```

Next, let's generate some nice figures! :)

Utility class to draw 3D arrows (copied from http://stackoverflow.com/questions/11140163)

[28]:
```
from matplotlib.patches import FancyArrowPatch
from mpl_toolkits.mplot3d import proj3d

class Arrow3D(FancyArrowPatch):
    def __init__(self, xs, ys, zs, *args, **kwargs):
        FancyArrowPatch.__init__(self, (0,0), (0,0), *args, **kwargs)
        self._verts3d = xs, ys, zs

    def draw(self, renderer):
        xs3d, ys3d, zs3d = self._verts3d
        xs, ys, zs = proj3d.proj_transform(xs3d, ys3d, zs3d, renderer.M)
        self.set_positions((xs[0],ys[0]),(xs[1],ys[1]))
        FancyArrowPatch.draw(self, renderer)
```

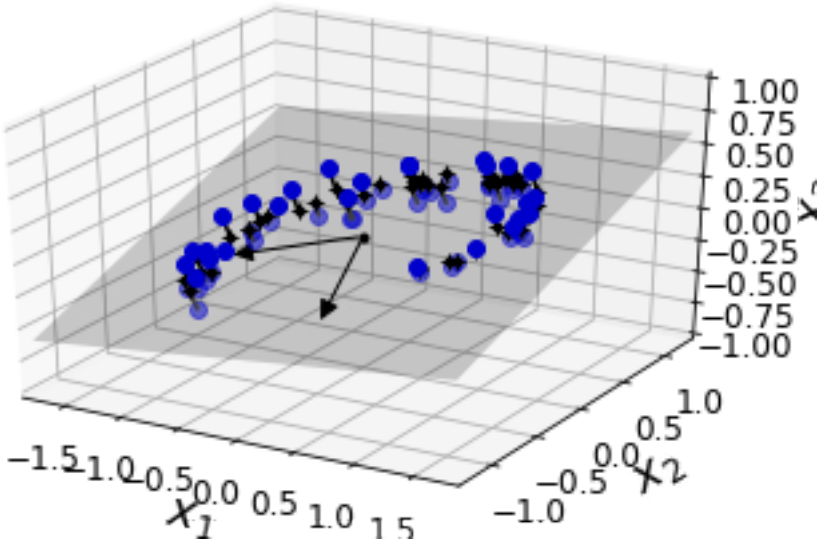
Express the plane as a function of x and y.

[29]:
```
axes = [-1.8, 1.8, -1.3, 1.3, -1.0, 1.0]

x1s = np.linspace(axes[0], axes[1], 10)
x2s = np.linspace(axes[2], axes[3], 10)
x1, x2 = np.meshgrid(x1s, x2s)

C = pca.components_
R = C.T.dot(C)
z = (R[0, 2] * x1 + R[1, 2] * x2) / (1 - R[2, 2])
```

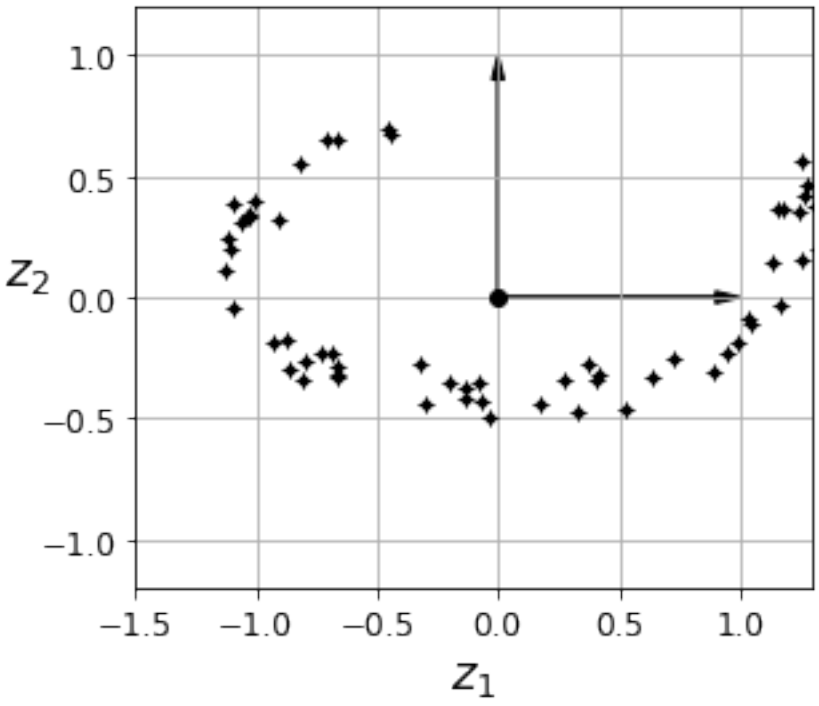
Plot the 3D dataset, the plane and the projections on that plane.

[30]:
```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(6, 3.8))
ax = fig.add_subplot(111, projection='3d')

X3D_above = X[X[:, 2] > X3D_inv[:, 2]]
```

```
X3D_below = X[X[:, 2] <= X3D_inv[:, 2]]

ax.plot(X3D_below[:, 0], X3D_below[:, 1], X3D_below[:, 2], "bo", alpha=0.5)

ax.plot_surface(x1, x2, z, alpha=0.2, color="k")
np.linalg.norm(C, axis=0)
ax.add_artist(Arrow3D([0, C[0, 0]],[0, C[0, 1]],[0, C[0, 2]],␣
 ↪mutation_scale=15, lw=1, arrowstyle="-|>", color="k"))
ax.add_artist(Arrow3D([0, C[1, 0]],[0, C[1, 1]],[0, C[1, 2]],␣
 ↪mutation_scale=15, lw=1, arrowstyle="-|>", color="k"))
ax.plot([0], [0], [0], "k.")

for i in range(m):
    if X[i, 2] > X3D_inv[i, 2]:
        ax.plot([X[i][0], X3D_inv[i][0]], [X[i][1], X3D_inv[i][1]], [X[i][2],␣
 ↪X3D_inv[i][2]], "k-")
    else:
        ax.plot([X[i][0], X3D_inv[i][0]], [X[i][1], X3D_inv[i][1]], [X[i][2],␣
 ↪X3D_inv[i][2]], "k-", color="#505050")

ax.plot(X3D_inv[:, 0], X3D_inv[:, 1], X3D_inv[:, 2], "k+")
ax.plot(X3D_inv[:, 0], X3D_inv[:, 1], X3D_inv[:, 2], "k.")
ax.plot(X3D_above[:, 0], X3D_above[:, 1], X3D_above[:, 2], "bo")
ax.set_xlabel("$x_1$", fontsize=18)
ax.set_ylabel("$x_2$", fontsize=18)
ax.set_zlabel("$x_3$", fontsize=18)
ax.set_xlim(axes[0:2])
ax.set_ylim(axes[2:4])
ax.set_zlim(axes[4:6])
plt.show()
```



[31]:
```
fig = plt.figure()
ax = fig.add_subplot(111, aspect='equal')

ax.plot(X2D[:, 0], X2D[:, 1], "k+")
ax.plot(X2D[:, 0], X2D[:, 1], "k.")
ax.plot([0], [0], "ko")
ax.arrow(0, 0, 0, 1, head_width=0.05, length_includes_head=True, head_length=0.
 ↪1, fc='k', ec='k')
ax.arrow(0, 0, 1, 0, head_width=0.05, length_includes_head=True, head_length=0.
 ↪1, fc='k', ec='k')
ax.set_xlabel("$z_1$", fontsize=18)
ax.set_ylabel("$z_2$", fontsize=18, rotation=0)
ax.axis([-1.5, 1.3, -1.2, 1.2])
ax.grid(True)
```



6 Manifold learning

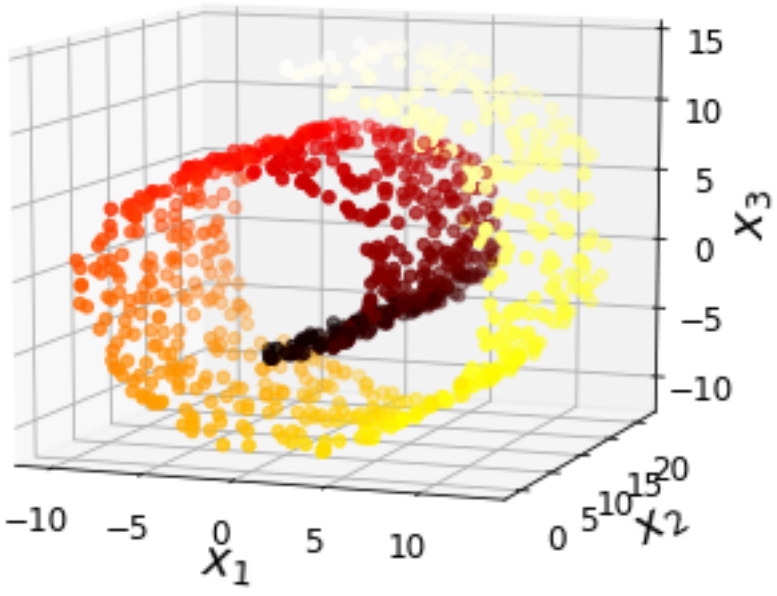
Swiss roll:

[32]:
```

```

```
# Let's make a swiss roll.
# t is the The univariate position of the sample according to the main␣
 ↪dimension of the points in the manifold.
from sklearn.datasets import make_swiss_roll
X, t = make_swiss_roll(n_samples=1000, noise=0.2, random_state=42)
```

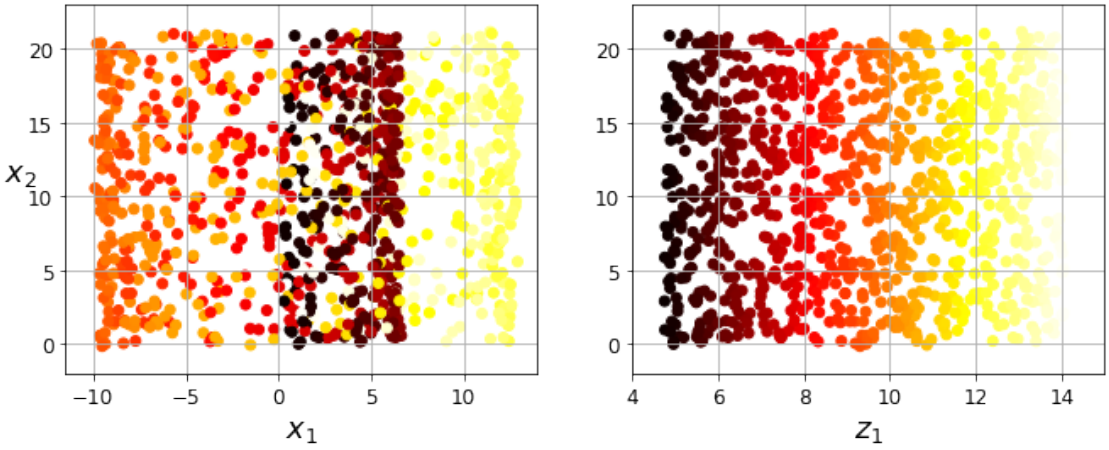
[ ]:

[33]:
```
axes = [-11.5, 14, -2, 23, -12, 15]

fig = plt.figure(figsize=(6, 5))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=t, cmap=plt.cm.hot)
ax.view_init(10, -70)
ax.set_xlabel("$x_1$", fontsize=18)
ax.set_ylabel("$x_2$", fontsize=18)
ax.set_zlabel("$x_3$", fontsize=18)
ax.set_xlim(axes[0:2])
ax.set_ylim(axes[2:4])
ax.set_zlim(axes[4:6])

plt.show()
```



Left figure collapses swiss roll to two dimensions. The right figure unrolls the manifold.

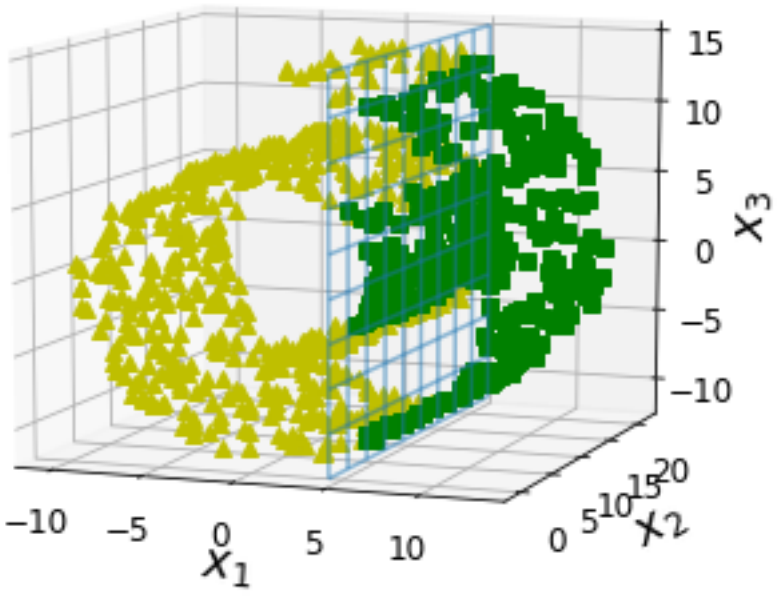
[34]:
```
plt.figure(figsize=(11, 4))

plt.subplot(121)
plt.scatter(X[:, 0], X[:, 1], c=t, cmap=plt.cm.hot)
plt.axis(axes[:4])
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$x_2$", fontsize=18, rotation=0)
plt.grid(True)

plt.subplot(122)
plt.scatter(t, X[:, 1], c=t, cmap=plt.cm.hot)
plt.axis([4, 15, axes[2], axes[3]])
plt.xlabel("$z_1$", fontsize=18)
plt.grid(True)

plt.show()
```

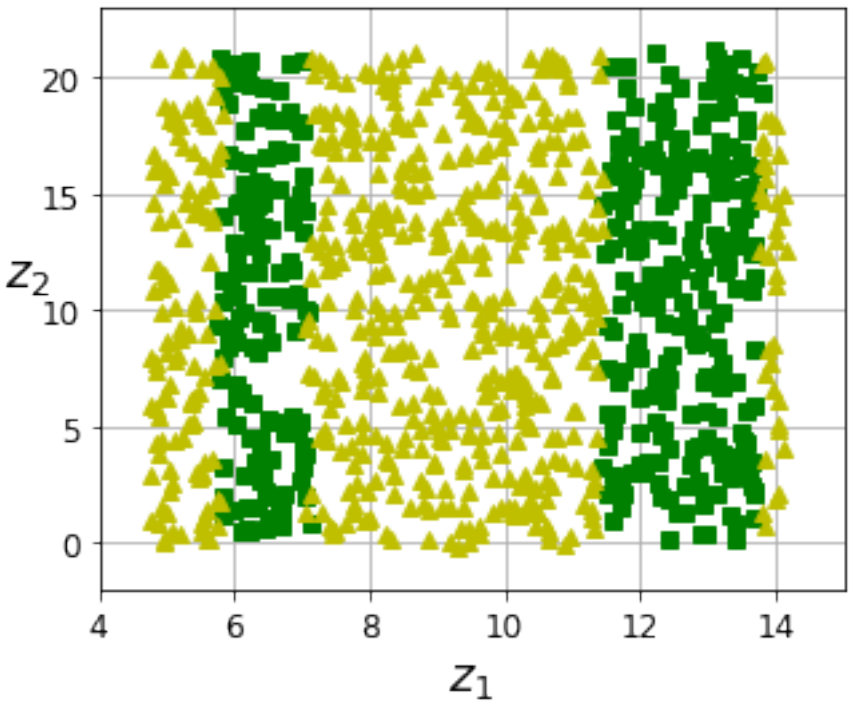


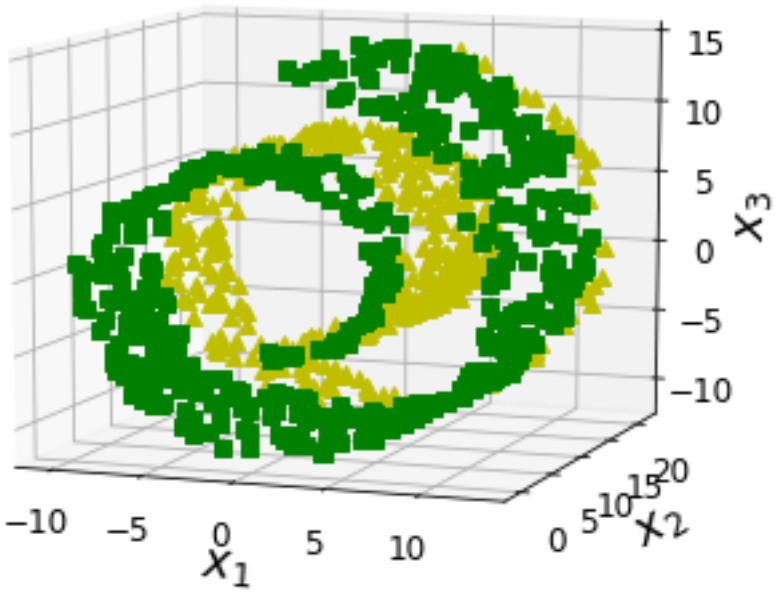
[35]:
```
from matplotlib import gridspec

axes = [-11.5, 14, -2, 23, -12, 15]

x2s = np.linspace(axes[2], axes[3], 10)
x3s = np.linspace(axes[4], axes[5], 10)
x2, x3 = np.meshgrid(x2s, x3s)

fig = plt.figure(figsize=(6, 5))
ax = plt.subplot(111, projection='3d')

positive_class = X[:, 0] > 5
X_pos = X[positive_class]
```

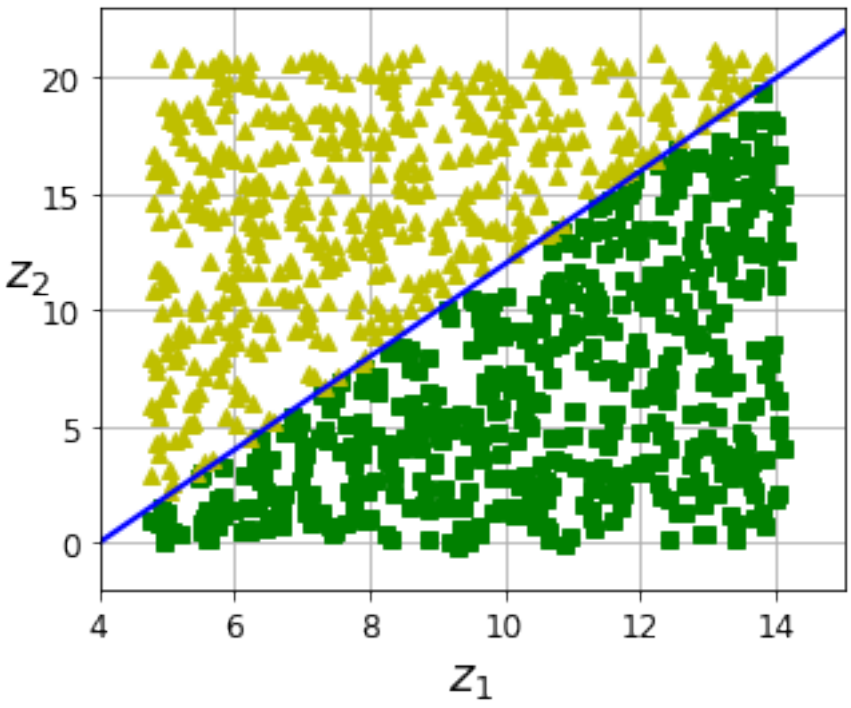
```
X_neg = X[~positive_class]
ax.view_init(10, -70)
ax.plot(X_neg[:, 0], X_neg[:, 1], X_neg[:, 2], "y^")
ax.plot_wireframe(5, x2, x3, alpha=0.5)
ax.plot(X_pos[:, 0], X_pos[:, 1], X_pos[:, 2], "gs")
ax.set_xlabel("$x_1$", fontsize=18)
ax.set_ylabel("$x_2$", fontsize=18)
ax.set_zlabel("$x_3$", fontsize=18)
ax.set_xlim(axes[0:2])
ax.set_ylim(axes[2:4])
ax.set_zlim(axes[4:6])


plt.show()

fig = plt.figure(figsize=(5, 4))
ax = plt.subplot(111)

plt.plot(t[positive_class], X[positive_class, 1], "gs")
plt.plot(t[~positive_class], X[~positive_class, 1], "y^")
plt.axis([4, 15, axes[2], axes[3]])
plt.xlabel("$z_1$", fontsize=18)
plt.ylabel("$z_2$", fontsize=18, rotation=0)
plt.grid(True)

plt.show()

fig = plt.figure(figsize=(6, 5))
ax = plt.subplot(111, projection='3d')

positive_class = 2 * (t[:] - 4) > X[:, 1]
X_pos = X[positive_class]
X_neg = X[~positive_class]
ax.view_init(10, -70)
ax.plot(X_neg[:, 0], X_neg[:, 1], X_neg[:, 2], "y^")
ax.plot(X_pos[:, 0], X_pos[:, 1], X_pos[:, 2], "gs")
ax.set_xlabel("$x_1$", fontsize=18)
ax.set_ylabel("$x_2$", fontsize=18)
ax.set_zlabel("$x_3$", fontsize=18)
ax.set_xlim(axes[0:2])
ax.set_ylim(axes[2:4])
ax.set_zlim(axes[4:6])

plt.show()

fig = plt.figure(figsize=(5, 4))
ax = plt.subplot(111)
```

```
plt.plot(t[positive_class], X[positive_class, 1], "gs")
plt.plot(t[~positive_class], X[~positive_class, 1], "y^")
plt.plot([4, 15], [0, 22], "b-", linewidth=2)
plt.axis([4, 15, axes[2], axes[3]])
plt.xlabel("$z_1$", fontsize=18)
plt.ylabel("$z_2$", fontsize=18, rotation=0)
plt.grid(True)


plt.show()
```




20
15
10
5
0
z_2
4
6
8
10
12
14
z_1

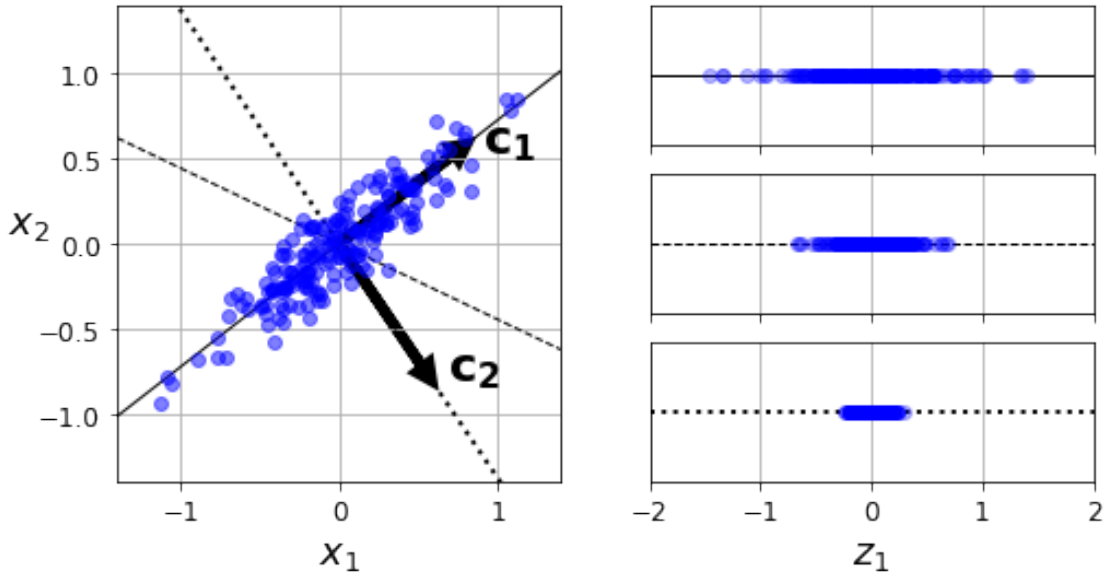

15
10
5
0
−5
−10
x_3
−10
−5
0
5
10
x_1
0
5
10
15
20
x_2



7 PCA

[36]:
```
angle = np.pi / 5
stretch = 5
m = 200

np.random.seed(3)
X = np.random.randn(m, 2) / 10
X = X.dot(np.array([[stretch, 0],[0, 1]])) # stretch
X = X.dot([[np.cos(angle), np.sin(angle)], [-np.sin(angle), np.cos(angle)]]) #␣
 ↪rotate

u1 = np.array([np.cos(angle), np.sin(angle)])
u2 = np.array([np.cos(angle - 2 * np.pi/6), np.sin(angle - 2 * np.pi/6)])
u3 = np.array([np.cos(angle - np.pi/2), np.sin(angle - np.pi/2)])

X_proj1 = X.dot(u1.reshape(-1, 1))
X_proj2 = X.dot(u2.reshape(-1, 1))
X_proj3 = X.dot(u3.reshape(-1, 1))

plt.figure(figsize=(8,4))
plt.subplot2grid((3,2), (0, 0), rowspan=3)
plt.plot([-1.4, 1.4], [-1.4*u1[1]/u1[0], 1.4*u1[1]/u1[0]], "k-", linewidth=1)
```

```
plt.plot([-1.4, 1.4], [-1.4*u2[1]/u2[0], 1.4*u2[1]/u2[0]], "k--", linewidth=1)
plt.plot([-1.4, 1.4], [-1.4*u3[1]/u3[0], 1.4*u3[1]/u3[0]], "k:", linewidth=2)
plt.plot(X[:, 0], X[:, 1], "bo", alpha=0.5)
plt.axis([-1.4, 1.4, -1.4, 1.4])
plt.arrow(0, 0, u1[0], u1[1], head_width=0.1, linewidth=5,␣
 ↪length_includes_head=True, head_length=0.1, fc='k', ec='k')
plt.arrow(0, 0, u3[0], u3[1], head_width=0.1, linewidth=5,␣
 ↪length_includes_head=True, head_length=0.1, fc='k', ec='k')
plt.text(u1[0] + 0.1, u1[1] - 0.05, r"$\mathbf{c_1}$", fontsize=22)
plt.text(u3[0] + 0.1, u3[1], r"$\mathbf{c_2}$", fontsize=22)
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$x_2$", fontsize=18, rotation=0)
plt.grid(True)

plt.subplot2grid((3,2), (0, 1))
plt.plot([-2, 2], [0, 0], "k-", linewidth=1)
plt.plot(X_proj1[:, 0], np.zeros(m), "bo", alpha=0.3)
plt.gca().get_yaxis().set_ticks([])
plt.gca().get_xaxis().set_ticklabels([])
plt.axis([-2, 2, -1, 1])
plt.grid(True)

plt.subplot2grid((3,2), (1, 1))
plt.plot([-2, 2], [0, 0], "k--", linewidth=1)
plt.plot(X_proj2[:, 0], np.zeros(m), "bo", alpha=0.3)
plt.gca().get_yaxis().set_ticks([])
plt.gca().get_xaxis().set_ticklabels([])
plt.axis([-2, 2, -1, 1])
plt.grid(True)

plt.subplot2grid((3,2), (2, 1))
plt.plot([-2, 2], [0, 0], "k:", linewidth=2)
plt.plot(X_proj3[:, 0], np.zeros(m), "bo", alpha=0.3)
plt.gca().get_yaxis().set_ticks([])
plt.axis([-2, 2, -1, 1])
plt.xlabel("$z_1$", fontsize=18)
plt.grid(True)


plt.show()
```



Before you can project the training set onto a lower-dimensional hyperplane, you first need to choose the right hyperplane. For example, a simple 2D dataset is represented on the left of Figure 8-7, along with three different axes (i.e., one-dimensional hyperplanes). On the right is the result of the projection of the dataset onto each of these axes. As you can see, the projection onto the solid line preserves the maximum variance, while the projection onto the dotted line preserves very little variance, and the projection onto the dashed line preserves an intermediate amount of variance.

8 MNIST compression

Obviously after dimensionality reduction, the training set takes up much less space. For example, try applying PCA to the MNIST dataset while preserving 95% of its variance. Each instance will have just over 150 features, instead of the original 784 features. So while most of the variance is preserved, the dataset is now less than 20% of its original size! This is a reasonable compression ratio, and you can see how this can speed up a classification algorithm (such as an SVM classifier) tremendously.

[37]:
```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
mnist.target = mnist.target.astype(np.uint8)
```

[38]:
```
from sklearn.model_selection import train_test_split
X = mnist["data"]
y = mnist["target"]
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

[39]:
```
pca = PCA()
pca.fit(X_train)
```

```
cumsum = np.cumsum(pca.explained_variance_ratio_)
# find the number of components that explain 95% of the variance.
d = np.argmax(cumsum >= 0.95) + 1
```

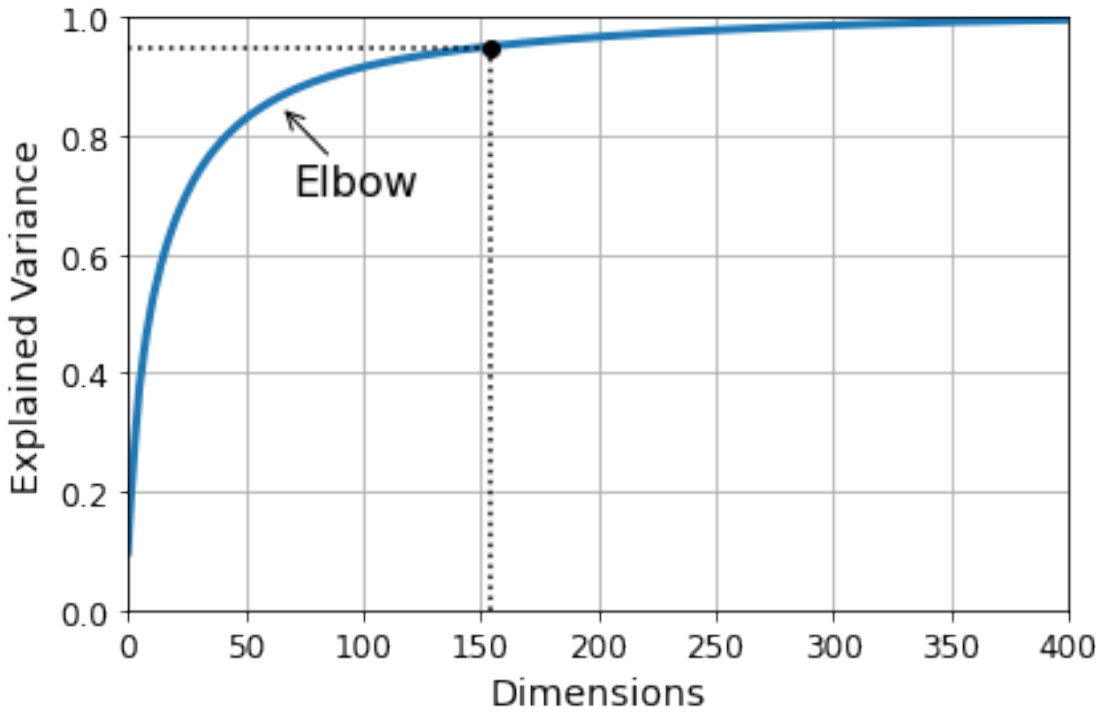
[40]:
```
d
```

[40]:
```
154
```

Let's graph the percent of explained variance and number of features:

[41]:
```
plt.figure(figsize=(6,4))
plt.plot(cumsum, linewidth=3)
plt.axis([0, 400, 0, 1])
plt.xlabel("Dimensions")
plt.ylabel("Explained Variance")
plt.plot([d, d], [0, 0.95], "k:")
plt.plot([0, d], [0.95, 0.95], "k:")
plt.plot(d, 0.95, "ko")
plt.annotate("Elbow", xy=(65, 0.85), xytext=(70, 0.7),
             arrowprops=dict(arrowstyle="->"), fontsize=16)
plt.grid(True)
save_fig("explained_variance_plot")
plt.show()
```

```
Saving figure explained_variance_plot
```



100 features would be enough to explain most variance.

[42]:
```
# Create Collapsed space for X.
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

[43]:
```
# We just have 154 dimensions
pca.n_components_
```

[43]: 154

[44]:
```
# Explain 95.04% of variance
np.sum(pca.explained_variance_ratio_)
```

[44]: 0.9504334914295707

[45]:
```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
# Recover the data after transformation.
X_recovered = pca.inverse_transform(X_reduced)
```

Let's try to transform the collapse data back to the picture and plot it.

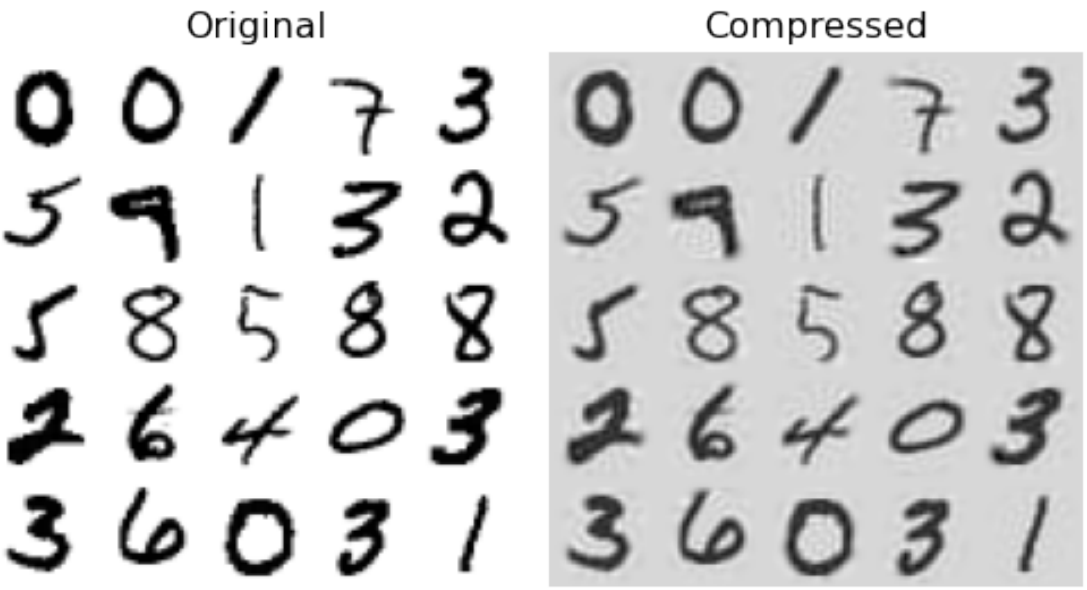
$$X_{recovered} = X_{d-proj} \cdot W_d^T$$

[46]:
```
def plot_digits(instances, images_per_row=5, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    images = [instance.reshape(size,size) for instance in instances]
    n_rows = (len(instances) - 1) // images_per_row + 1
    row_images = []
    n_empty = n_rows * images_per_row - len(instances)
    images.append(np.zeros((size, size * n_empty)))
    for row in range(n_rows):
        rimages = images[row * images_per_row : (row + 1) * images_per_row]
        row_images.append(np.concatenate(rimages, axis=1))
    image = np.concatenate(row_images, axis=0)
    plt.imshow(image, cmap = plt.cm.binary, **options)
    plt.axis("off")
```

[47]:
```
plt.figure(figsize=(7, 4))
plt.subplot(121)
plot_digits(X_train[::2100])
plt.title("Original", fontsize=16)
plt.subplot(122)
plot_digits(X_recovered[::2100])
```

```
plt.title("Compressed", fontsize=16)

save_fig("mnist_compression_plot")
```

```
Saving figure mnist_compression_plot
```



[48]:
```
X_reduced_pca = X_reduced
```

8.1 Incremental PCA

One problem with the preceding implementation of PCA is that it requires the whole training set to fit in memory in order for the SVD algorithm to run. Fortunately, Incremental PCA (IPCA) algorithms have been developed: you can split the training set into mini-batches and feed an IPCA algorithm one minibatch at a time. This is useful for large training sets, and also to apply PCA online (i.e., on the fly, as new instances arrive). The following code splits the MNIST dataset into 100 mini-batches (using NumPy's array_split() function) and feeds them to Scikit-Learn's IncrementalPCA class5 to reduce the dimensionality of the MNIST dataset down to 154 dimensions. Note that you must call the partial_fit() method with each mini-batch rather than the fit() method with the whole training set:

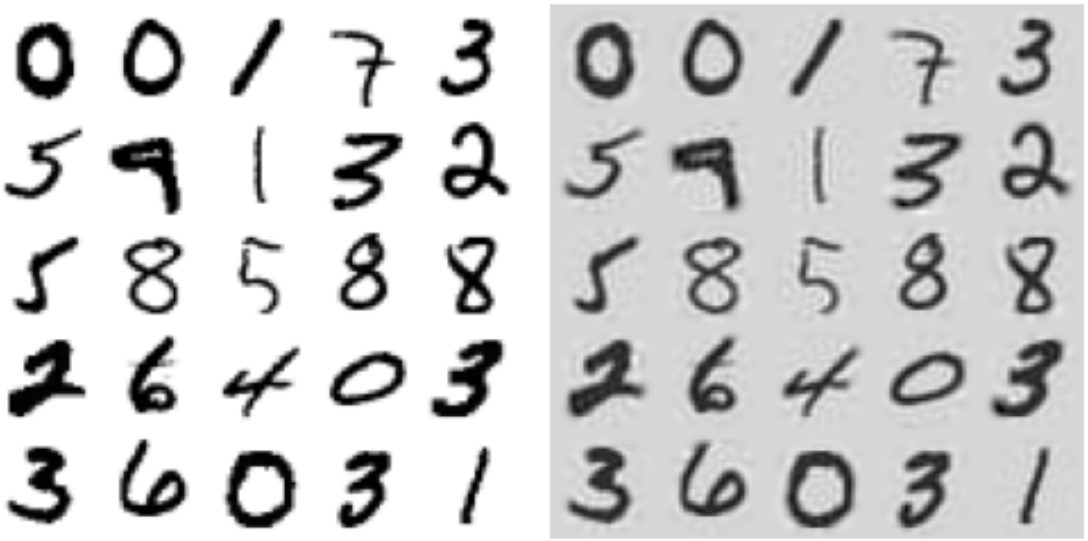
[49]:
```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
# setup incremental PCA
inc_pca = IncrementalPCA(n_components=154)
# loop over 100 batches
for X_batch in np.array_split(X_train, n_batches):
```

```
    print(".", end="") # not shown in the book
    # partial fit append data to the matrix
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

…
…

[50]:
```
X_recovered_inc_pca = inc_pca.inverse_transform(X_reduced)
```

[51]:
```
plt.figure(figsize=(7, 4))
plt.subplot(121)
plot_digits(X_train[::2100])
plt.subplot(122)
plot_digits(X_recovered_inc_pca[::2100])
plt.tight_layout()
```



[52]:
```
X_reduced_inc_pca = X_reduced
```

Let's compare the results of transforming MNIST using regular PCA and incremental PCA. First, the means are equal:

[53]:
```
np.allclose(pca.mean_, inc_pca.mean_)
```

[53]:
```
True
```

But the results are not exactly identical. Incremental PCA gives a very good approximate solution, but it's not perfect because the principle components in 100 batches are sligthtly different.

[54]:
```
np.allclose(X_reduced_pca, X_reduced_inc_pca)
```

[54]:
```
False
```

8.1.1 Using `memmap()`

Alternative of running large files is to use memmap. Memory-mapped files are used for accessing small segments of large files on disk, without reading the entire file into memory. Let's create the `memmap()` structure and copy the MNIST data into it. This would typically be done by a first program:

[55]:
```
filename = "my_mnist.data"
m, n = X_train.shape
X_mm = np.memmap(filename, dtype='float32', mode='write', shape=(m, n))
X_mm[:] = X_train
```

Now deleting the `memmap()` object will trigger its Python finalizer, which ensures that the data is saved to disk.

[56]:
```
del X_mm
```

Next, another program would load the data and use it for training:

[57]:
```
X_mm = np.memmap(filename, dtype="float32", mode="readonly", shape=(m, n))

batch_size = m // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mm)
```

[57]:
```
IncrementalPCA(batch_size=525, n_components=154)
```

Scikit-Learn offers yet another option to perform PCA, called Randomized PCA. This is a stochastic algorithm that quickly finds an approximation of the first d principal components. Its computational complexity is $O(m \times d^2) + O(d^3)$, instead of $O(m \times n^2) + O(n^3)$, so it is dramatically faster than the previous algorithms when d is much smaller than n.

[58]:
```
rnd_pca = PCA(n_components=154, svd_solver="randomized", random_state=42)
X_reduced = rnd_pca.fit_transform(X_train)
```

8.2 Time complexity

Let's time regular PCA against Incremental PCA and Randomized PCA, for various number of principal components:

[59]:
```
import time

for n_components in (2, 10, 154):
    print("n_components =", n_components)
    regular_pca = PCA(n_components=n_components)
```

```
    inc_pca = IncrementalPCA(n_components=n_components, batch_size=500)
    rnd_pca = PCA(n_components=n_components, random_state=42,␣
↪svd_solver="randomized")

    for pca in (regular_pca, inc_pca, rnd_pca):
        t1 = time.time()
        pca.fit(X_train)
        t2 = time.time()
        print("    {}: {:.1f} seconds".format(pca.__class__.__name__, t2 - t1))
```

```
n_components = 2
    PCA: 1.3 seconds
    IncrementalPCA: 9.0 seconds
    PCA: 0.7 seconds
n_components = 10
    PCA: 0.8 seconds
    IncrementalPCA: 8.9 seconds
    PCA: 0.8 seconds
n_components = 154
    PCA: 2.6 seconds
    IncrementalPCA: 11.8 seconds
    PCA: 2.5 seconds
```

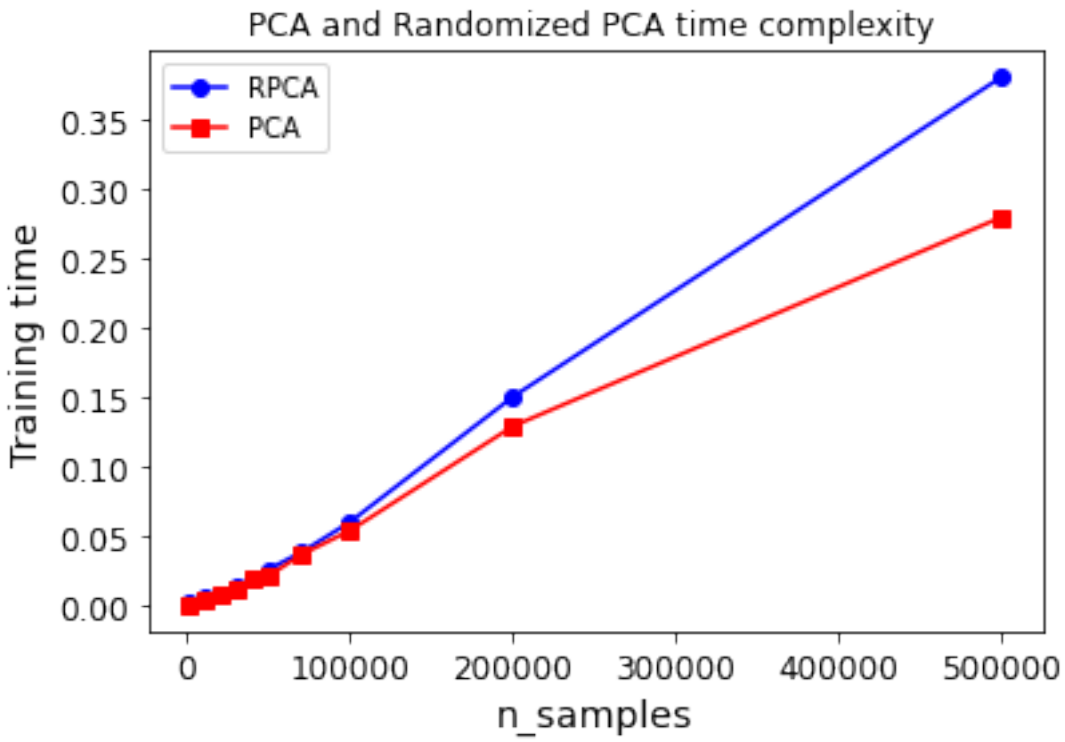
Incremental PCA is slower because it relies on a slower hard-drive instead of fast memory, it needs to be used only if you cannot fit the data into RAM.

Now let's compare PCA and Randomized PCA for datasets of different sizes (number of instances):

[60]:
```
times_rpca = []
times_pca = []
sizes = [1000, 10000, 20000, 30000, 40000, 50000, 70000, 100000, 200000, 500000]
for n_samples in sizes:
    X = np.random.randn(n_samples, 5)
    pca = PCA(n_components = 2, svd_solver="randomized", random_state=42)
    t1 = time.time()
    pca.fit(X)
    t2 = time.time()
    times_rpca.append(t2 - t1)
    pca = PCA(n_components = 2)
    t1 = time.time()
    pca.fit(X)
    t2 = time.time()
    times_pca.append(t2 - t1)

plt.plot(sizes, times_rpca, "b-o", label="RPCA")
plt.plot(sizes, times_pca, "r-s", label="PCA")
plt.xlabel("n_samples")
plt.ylabel("Training time")
```

```
plt.legend(loc="upper left")
plt.title("PCA and Randomized PCA time complexity ")
```

[60]:
```
Text(0.5, 1.0, 'PCA and Randomized PCA time complexity ')
```

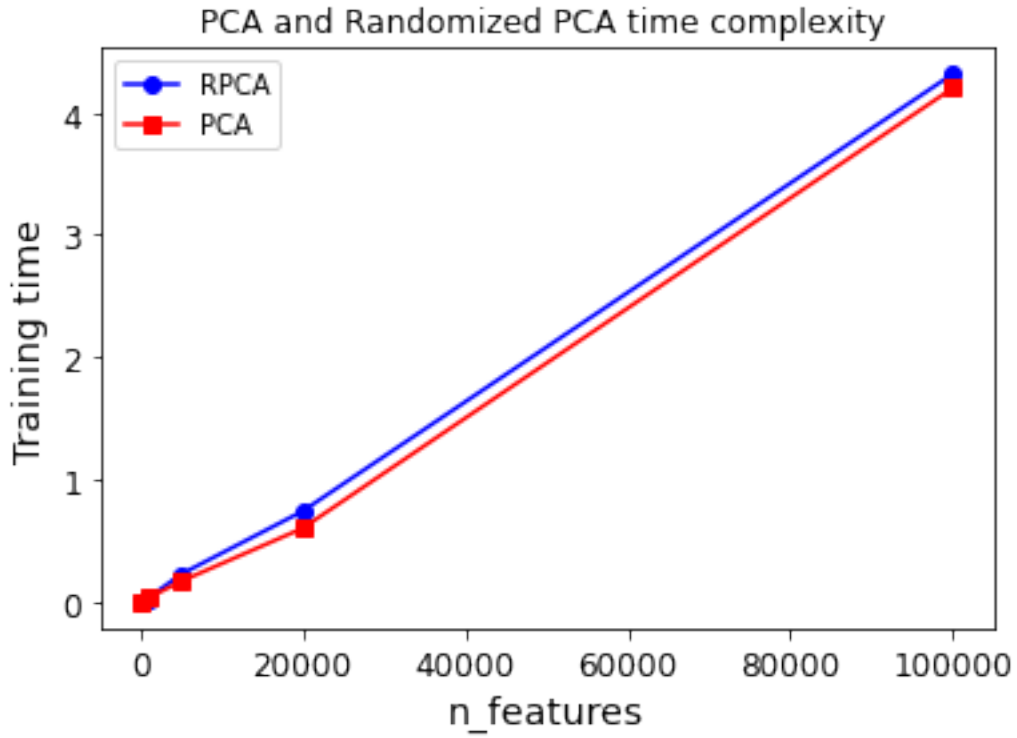


And now let's compare their performance on datasets of 2,000 instances with various numbers of features:

[61]:
```
times_rpca = []
times_pca = []
sizes = [100, 1000, 5000, 20000, 100000]
for n_features in sizes:
    X = np.random.randn(2000, n_features)
    pca = PCA(n_components = 2, random_state=42, svd_solver="randomized")
    t1 = time.time()
    pca.fit(X)
    t2 = time.time()
    times_rpca.append(t2 - t1)
    pca = PCA(n_components = 2)
    t1 = time.time()
    pca.fit(X)
    t2 = time.time()
    times_pca.append(t2 - t1)
```

```
plt.plot(sizes, times_rpca, "b-o", label="RPCA")
plt.plot(sizes, times_pca, "r-s", label="PCA")
plt.xlabel("n_features")
plt.ylabel("Training time")
plt.legend(loc="upper left")
plt.title("PCA and Randomized PCA time complexity ")
```

[61]: `Text(0.5, 1.0, 'PCA and Randomized PCA time complexity ')`



9 Kernel PCA

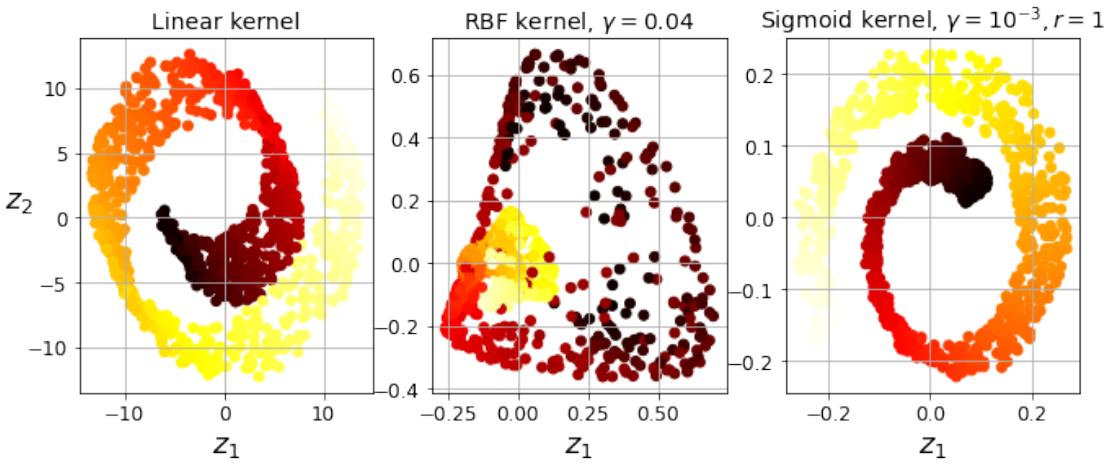
In Chapter 5 we discussed the kernel trick, a mathematical technique that implicitly maps instances into a very high-dimensional space (called the feature space), enabling nonlinear classification and regression with Support Vector Machines. Recall that a linear decision boundary in the high-dimensional feature space corresponds to a complex nonlinear decision boundary in the original space.

The same trick can be applied to PCA, making it possible to perform complex nonlinear projections for dimensionality reduction. This is called Kernel PCA (kPCA). It is often good at preserving clusters of instances after projection, or sometimes even unrolling datasets that lie close to a twisted manifold.

[62]:
```
# Get a swiss roll
X, t = make_swiss_roll(n_samples=1000, noise=0.2, random_state=42)
```

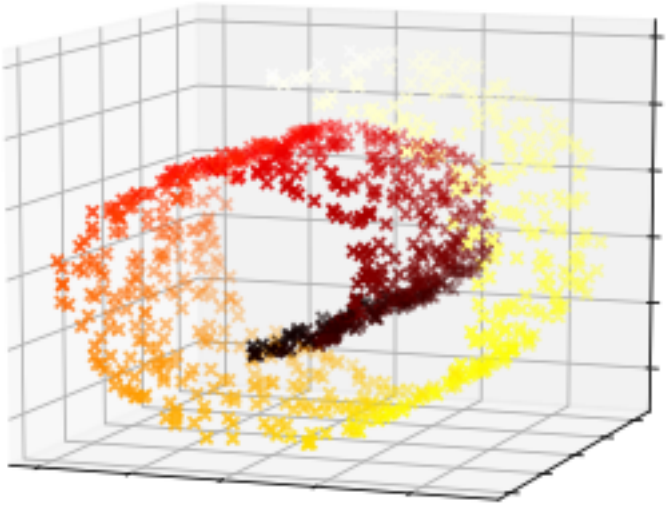
[63]:
```
from sklearn.decomposition import KernelPCA
# extract 2 components
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

[64]:
```
# Plot different kernels
from sklearn.decomposition import KernelPCA

lin_pca = KernelPCA(n_components = 2, kernel="linear",␣
 ↪fit_inverse_transform=True)
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.0433,␣
 ↪fit_inverse_transform=True)
sig_pca = KernelPCA(n_components = 2, kernel="sigmoid", gamma=0.001, coef0=1,␣
 ↪fit_inverse_transform=True)

y = t > 6.9

plt.figure(figsize=(11, 4))
for subplot, pca, title in ((131, lin_pca, "Linear kernel"), (132, rbf_pca,␣
 ↪"RBF kernel, $\gamma=0.04$"), (133, sig_pca, "Sigmoid kernel,␣
 ↪$\gamma=10^{-3}, r=1$")):
    X_reduced = pca.fit_transform(X)
    if subplot == 132:
        X_reduced_rbf = X_reduced

    plt.subplot(subplot)
    #plt.plot(X_reduced[y, 0], X_reduced[y, 1], "gs")
    #plt.plot(X_reduced[~y, 0], X_reduced[~y, 1], "y^")
    plt.title(title, fontsize=14)
    plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=plt.cm.hot)
    plt.xlabel("$z_1$", fontsize=18)
    if subplot == 131:
        plt.ylabel("$z_2$", fontsize=18, rotation=0)
    plt.grid(True)


plt.show()
```

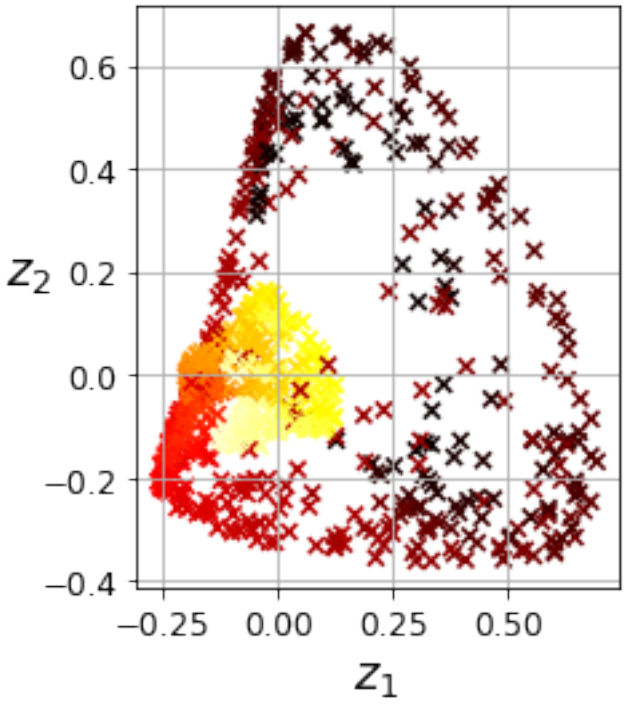


shows the Swiss roll, reduced to two dimensions using a linear kernel (equivalent to simply using the PCA class), an RBF kernel, and a sigmoid kernel (Logistic). Next we try to recover initial swiss-roll from the collapsed data. We start with RBF kernel.

[65]:
```
plt.figure(figsize=(6, 5))

X_inverse = rbf_pca.inverse_transform(X_reduced_rbf)

ax = plt.subplot(111, projection='3d')
ax.view_init(10, -70)
ax.scatter(X_inverse[:, 0], X_inverse[:, 1], X_inverse[:, 2], c=t, cmap=plt.cm.
 ↪hot, marker="x")
ax.set_xlabel("")
ax.set_ylabel("")
ax.set_zlabel("")
ax.set_xticklabels([])
ax.set_yticklabels([])
ax.set_zticklabels([])


plt.show()
```



[66]:
```
X_reduced = rbf_pca.fit_transform(X)

plt.figure(figsize=(11, 4))
plt.subplot(132)
plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=plt.cm.hot, marker="x")
plt.xlabel("$z_1$", fontsize=18)
plt.ylabel("$z_2$", fontsize=18, rotation=0)
plt.grid(True)
```



As kPCA is an unsupervised learning algorithm, there is no obvious performance measure to help you select the best kernel and hyperparameter values. However, dimensionality reduction is often a preparation step for a supervised learning task (e.g., classification), so you can simply use grid search to select the kernel and hyperparameters that lead to the best performance on that task. For example, the following code creates a two-step pipeline, first reducing dimensionality to two dimensions using kPCA, then applying Logistic Regression for classification. Then it uses GridSearchCV to find the best kernel and gamma value for kPCA in order to get the best classification accuracy at the end of the pipeline:

[67]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

clf = Pipeline([
        ("kpca", KernelPCA(n_components=2)),
        ("log_reg", LogisticRegression())
    ])
# try different kernels
param_grid = [{
        "kpca__gamma": np.linspace(0.03, 0.05, 10),
        "kpca__kernel": ["rbf", "sigmoid"]
    }]
# Do the grid search
grid_search = GridSearchCV(clf, param_grid, cv=3)
```

```
grid_search.fit(X, y)
```

[67]:
```
GridSearchCV(cv=3,
             estimator=Pipeline(steps=[('kpca', KernelPCA(n_components=2)),
                                       ('log_reg', LogisticRegression())]),
             param_grid=[{'kpca__gamma': array([0.03      , 0.03222222,
0.03444444, 0.03666667, 0.03888889,
       0.04111111, 0.04333333, 0.04555556, 0.04777778, 0.05      ]),
                          'kpca__kernel': ['rbf', 'sigmoid']}])
```

[68]:
```
# Best kernell
print(grid_search.best_params_)
```

```
{'kpca__gamma': 0.043333333333333335, 'kpca__kernel': 'rbf'}
```

Another approach, this time entirely unsupervised, is to select the kernel and hyperparameters that yield the lowest reconstruction error.

[69]:
```
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.0433,
                    fit_inverse_transform=True)
X_reduced = rbf_pca.fit_transform(X)
X_preimage = rbf_pca.inverse_transform(X_reduced)
# Recover initial image from the collpese data
```

[70]:
```
# Calculate MSE for recovered X-data.
from sklearn.metrics import mean_squared_error
mean_squared_error(X, X_preimage)
```

[70]:
```
1.1346172111704741e-26
```

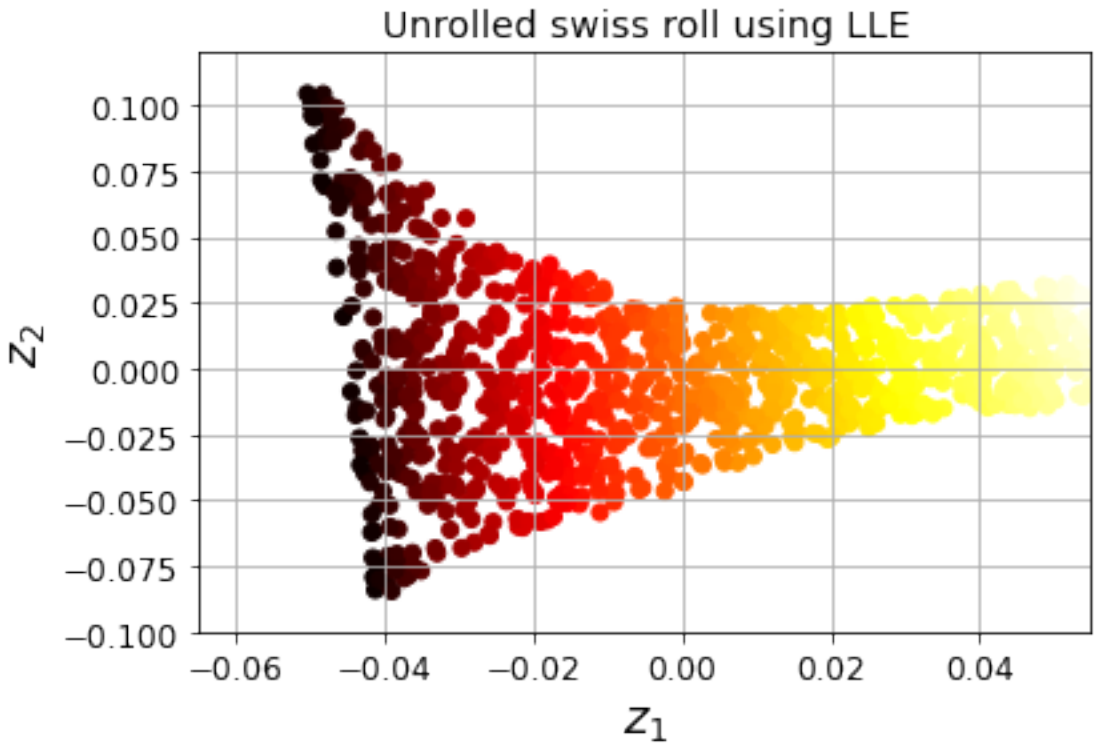
10 LLE

Locally Linear Embedding (LLE) is another very powerful nonlinear dimensionality reduction (NLDR) technique. It is a Manifold Learning technique that does not rely on projections like the previous algorithms. LLE works by first measuring how each training instance linearly relates to its closest neighbors (c.n.), and then looking for a low-dimensional representation of the training set where these local relationships are best preserved. This makes it particularly good at unrolling twisted manifolds, especially when there is not too much noise. For example, the following code uses Scikit-Learn's LocallyLinearEmbedding class to unroll the Swiss roll. As you can see, the Swiss roll is completely unrolled and the distances between instances are locally well preserved. However, distances are not preserved on a larger scale: the left part of the unrolled Swiss roll is squeezed, while the right part is stretched. Nevertheless, LLE did a pretty good job at modeling the manifold.

[71]:
```
X, t = make_swiss_roll(n_samples=1000, noise=0.2, random_state=41)
```

[72]:
```
from sklearn.manifold import LocallyLinearEmbedding

lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10, random_state=42)
X_reduced = lle.fit_transform(X)
```

[73]:
```
plt.title("Unrolled swiss roll using LLE", fontsize=14)
plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=plt.cm.hot)
plt.xlabel("$z_1$", fontsize=18)
plt.ylabel("$z_2$", fontsize=18)
plt.axis([-0.065, 0.055, -0.1, 0.12])
plt.grid(True)


plt.show()
```



11 MDS, Isomap and t-SNE

There are many other dimensionality reduction techniques, several of which are available in Scikit-Learn. Here are some of the most popular: * Multidimensional Scaling (MDS) reduces dimensionality while trying to preserve the distances between the instances. * Isomap creates a graph by connecting each instance to its nearest neighbors, then reduces dimensionality while trying to preserve the geodesic distances9 between the instances. * t-Distributed Stochastic Neighbor Em-

bedding (t-SNE) reduces dimensionality while trying to keep similar instances close and dissimilar instances apart. It is mostly used for visualization, in particular to visualize clusters of instances in high-dimensional space (e.g., to visualize the MNIST images in 2D). * Linear Discriminant Analysis (LDA) is actually a classification algorithm, but during training it learns the most discriminative axes between the classes, and these axes can then be used to define a hyperplane onto which to project the data. The benefit is that the projection will keep classes as far apart as possible, so LDA is a good technique to reduce dimensionality before running another classification algorithm such as an SVM classifier.

[74]:
```
from sklearn.manifold import MDS

mds = MDS(n_components=2, random_state=42)
X_reduced_mds = mds.fit_transform(X)
```

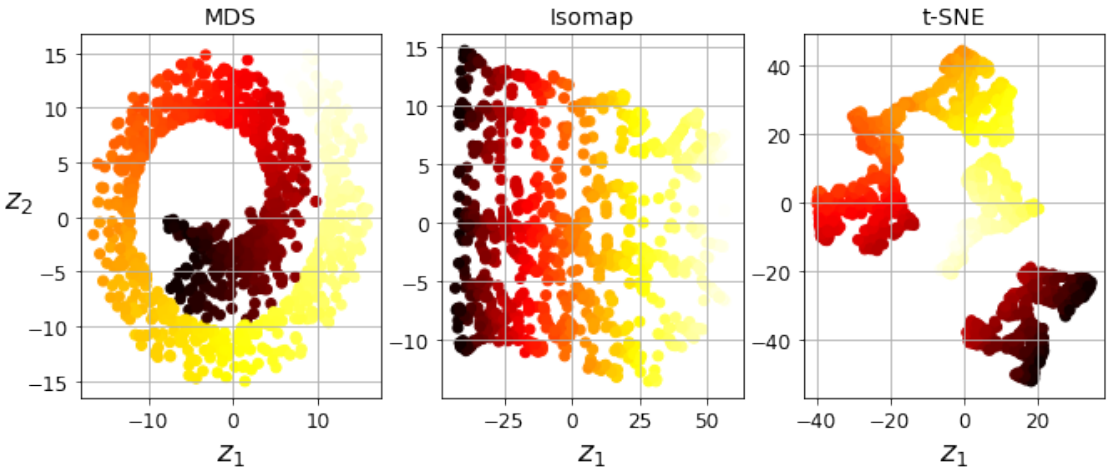
[75]:
```
from sklearn.manifold import Isomap

isomap = Isomap(n_components=2)
X_reduced_isomap = isomap.fit_transform(X)
```

[76]:
```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, random_state=42)
X_reduced_tsne = tsne.fit_transform(X)
```

[77]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

lda = LinearDiscriminantAnalysis(n_components=2)
X_mnist = mnist["data"]
y_mnist = mnist["target"]
lda.fit(X_mnist, y_mnist)
X_reduced_lda = lda.transform(X_mnist)
```

[78]:
```
titles = ["MDS", "Isomap", "t-SNE"]

plt.figure(figsize=(11,4))

for subplot, title, X_reduced in zip((131, 132, 133), titles,
                                     (X_reduced_mds, X_reduced_isomap,␣
 ↪X_reduced_tsne)):
    plt.subplot(subplot)
    plt.title(title, fontsize=14)
    plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=plt.cm.hot)
    plt.xlabel("$z_1$", fontsize=18)
    if subplot == 131:
        plt.ylabel("$z_2$", fontsize=18, rotation=0)
    plt.grid(True)
```

```
plt.show()
```



[79]:
```
def learned_parameters(model):
    return [m for m in dir(model)
            if m.endswith("_") and not m.startswith("_")]
```

12 Unsupervised Learning

Sometimes we don't know what we are looking for in a sea of unstructured an unlabeled data.

If you want to create a classifier on 'defective'/'good' data a classifier would do. But if you don't know which data is defective, you need to predict the labels first.

We will look into:

Clustering - grouping of similar instances together across multiple dimensions. Applications: customer segmentation, recommender systems, search engines, dimensionality reduction.

Anomaly Detection - learn what is a 'normal' data looks like to predict good and bad 'anomalies' for further investigation. Applications: find defective items and new opportunities.

Density Estimation - estimation of the probability density function. Instances with a very low probability of occuring are likely anomalies.

12.1 Introduction – Classification *vs* Clustering

If you stumble or new flowers hiking in the mountains. The flowers are not exactly the same, but they are sufficiently similar to be a group clearly different from other species of flowers, so thus the new speicie is discovered, clustered and later classified.

Clustering is not supervised, the algorithm identify a group of similar objects, but will not tell you what these are. Labeling cluster is a separate supervised or unsupervised task.

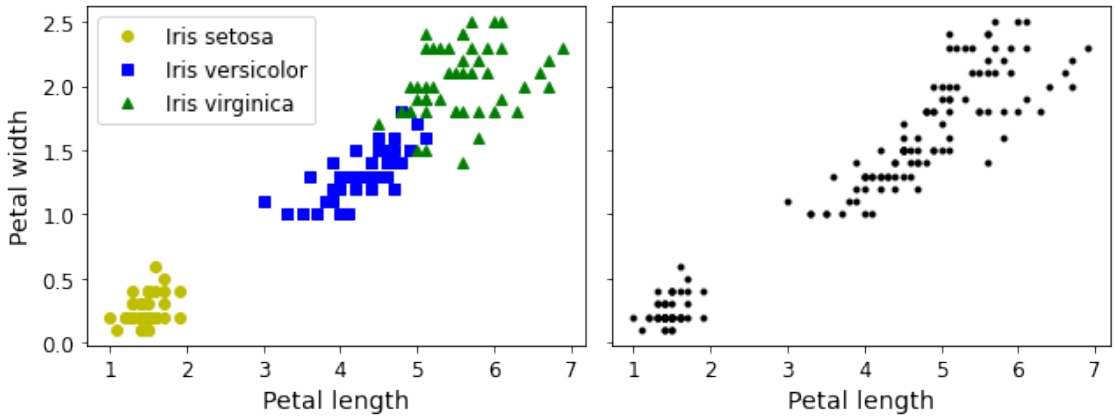
[80]:
```
from sklearn.datasets import load_iris
```

[81]:
```
data = load_iris()
X = data.data
y = data.target
data.target_names
```

[81]:
```
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

[82]:
```
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
```



Algorithm can look at this data and 1, 2, 3 or more clusters.

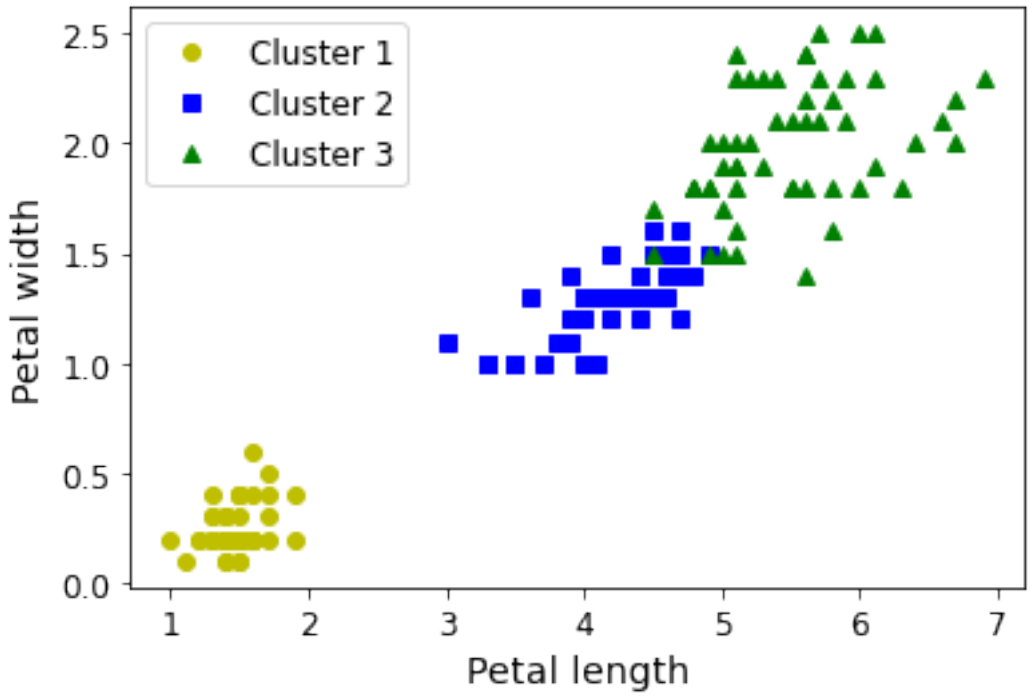
A Gaussian mixture model (explained below) can actually separate these clusters pretty well.

[83]:
```
from sklearn.mixture import GaussianMixture
```

[84]:
```
from copy import deepcopy
y_pred = GaussianMixture(n_components=3, random_state=42).fit(X).predict(X)
# replace prediction with the most common value
dic_val = {}
index = {}
for i in range(3):
  newval = np.bincount(y[np.where(y_pred == i)[0]]).argmax()
  index[i] = deepcopy(np.where(y_pred == i)[0])
  dic_val[i] = deepcopy(newval)

for i in range(3):
  y_pred[index[i]] = dic_val[i]
```

[ ]:
```
```

[85]:
```
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()
```



[86]:
```
np.sum(y_pred==y)
```

[86]:
```
145
```

[87]:
```
np.sum(y_pred==y) / len(y_pred)
```

[87]:
```
0.9666666666666667
```

There is no common definition of a cluster. Some clusters are based on a centroid, others look for contigious regions. Next we will look at the popular clustering algorihtms

13 K-Means

13.1 Basic Idea

k-means clustering is a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining. k-means clustering aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells.

They both use cluster centers to model the data; however, k-means clustering tends to find clusters of comparable spatial extent, while the expectation-maximization mechanism allows clusters to have different shapes.

One of the most straightforward tasks we can perform on a data set without labels is to find groups of data in our dataset which are similar to one another – what we call clusters.

K-Means is one of the most popular "clustering" algorithms. K-means stores k centroids that it uses to define clusters. A point is considered to be in a particular cluster if it is closer to that cluster's centroid than any other centroid.

K-Means finds the best centroids by alternating between (1) assigning data points to clusters based on the current centroids (2) choosing centroids (points which are the center of a cluster) based on the current assignment of data points to clusters.

13.2 Algorithm

In the clustering problem, we are given a training set $x^{(1)}, ..., x^{(m)}$, and want to group the data into a few cohesive "clusters." Here, we are given feature vectors for each data point $x^{(i)} \in \mathbb{R}^n$ as usual; but no labels $y^{(i)}$ (making this an unsupervised learning problem). Our goal is to predict k centroids and a label $c^{(i)}$ for each datapoint. The k-means clustering algorithm is as follows:

1. Initialize cluster centroids $\mu_1, \mu_2, ..., \mu_k \in \mathbb{R}^n$
2. Repeat until convergence: { For every observation i set cluster c which has the closest

$$c^{(i)} := arg\min_j \|x^{(i)} - \mu_j\|^2$$

For every cluster j update cluster location by calculating a new mean based on the observations assigned to a cluster. KMeans command uses Lloyd's algorithm:

$$\mu_j := \frac{\sum_{i=1}^{m} 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^{m} 1\{c^{(i)} = j\}}$$

In some specifications the new location of the centroid can be weighted by the inverse distance to old centroid.

Let's start by generating some blobs:

[88]:
```
from sklearn.datasets import make_blobs
```

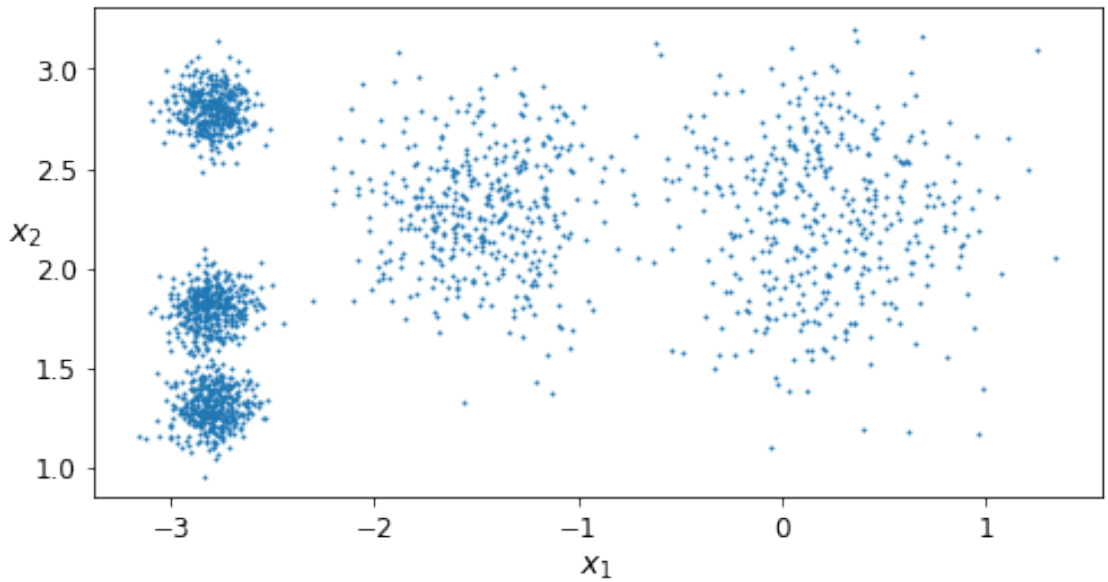
[89]:
```
blob_centers = np.array(
    [[ 0.2,  2.3],
     [-1.5 ,  2.3],
     [-2.8,  1.8],
     [-2.8,  2.8],
     [-2.8,  1.3]])
blob_std = np.array([0.4, 0.3, 0.1, 0.1, 0.1])
```

[90]:
```
X, y = make_blobs(n_samples=2000, centers=blob_centers,
                  cluster_std=blob_std, random_state=7)
```

Now let's plot them:

[91]:
```
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)
```

[92]:
```
plt.figure(figsize=(8, 4))
plot_clusters(X)

plt.show()
```



13.2.1 Fit and Predict

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

[93]:
```
from sklearn.cluster import KMeans
```

[94]:
```
k = 5
kmeans = KMeans(n_clusters=k, random_state=42)
y_pred = kmeans.fit_predict(X)
```

Each instance was assigned to one of the 5 clusters:

[95]:
```
y_pred
```

[95]:
```
array([0, 4, 1, …, 2, 1, 4], dtype=int32)
```

[96]:
```
y_pred is kmeans.labels_
```

[96]:
```
True
```

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

[97]:
```
kmeans.cluster_centers_
```

[97]:
```
array([[-2.80037642,  1.30082566],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80389616,  1.80117999]])
```

Note that the `KMeans` instance preserves the labels of the instances it was trained on. Somewhat confusingly, in this context, the *label* of an instance is the index of the cluster that instance gets assigned to:

[98]:
```
kmeans.labels_
```

[98]:
```
array([0, 4, 1, …, 2, 1, 4], dtype=int32)
```

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

[99]:
```
X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
kmeans.predict(X_new)
```

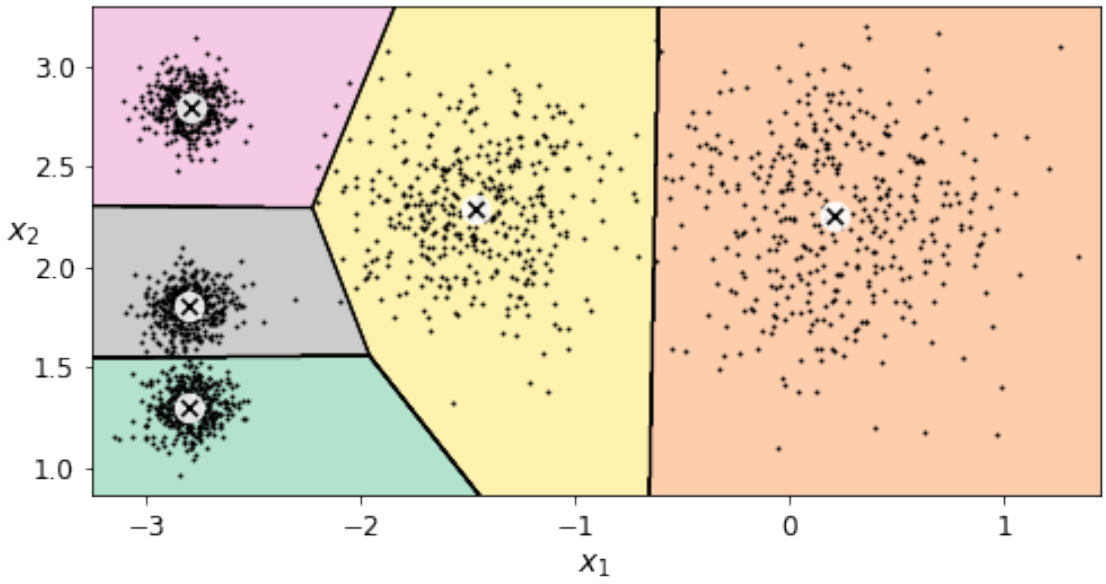
[99]:
```
array([1, 1, 2, 2], dtype=int32)
```

13.2.2 Decision Boundaries

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

[100]:
```
def plot_data(X):
    plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)

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

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

[101]:
```
plt.figure(figsize=(8, 4))
plot_decision_boundaries(kmeans, X)

plt.show()
```



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

The K-means does not work very well if the blobs have different diameter, because it cares only about the distances from the centroid.

13.2.3 Hard Clustering *vs* Soft Clustering

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

[102]:
```
kmeans.transform(X_new)
```

[102]:
```
array([[2.88633901, 0.32995317, 2.9042344 , 1.49439034, 2.81093633],
       [5.84236351, 2.80290755, 5.84739223, 4.4759332 , 5.80730058],
       [1.71086031, 3.29399768, 0.29040966, 1.69136631, 1.21475352],
       [1.21567622, 3.21806371, 0.36159148, 1.54808703, 0.72581411]])
```

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

[103]:
```
np.linalg.norm(np.tile(X_new, (1, k)).reshape(-1, k, 2) - kmeans.
↪cluster_centers_, axis=2)
```

[103]:
```
array([[2.88633901, 0.32995317, 2.9042344 , 1.49439034, 2.81093633],
       [5.84236351, 2.80290755, 5.84739223, 4.4759332 , 5.80730058],
       [1.71086031, 3.29399768, 0.29040966, 1.69136631, 1.21475352],
       [1.21567622, 3.21806371, 0.36159148, 1.54808703, 0.72581411]])
```

13.2.4 K-Means Algorithm

The K-Means algorithm is one of the fastest clustering algorithms, but also one of the simplest: * First initialize k centroids randomly: k distinct instances are chosen randomly from the dataset and the centroids are placed at their locations. * Repeat until convergence (i.e., until the centroids stop moving): * Assign each instance to the closest centroid. * Update the centroids to be the mean of the instances that are assigned to them.

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

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

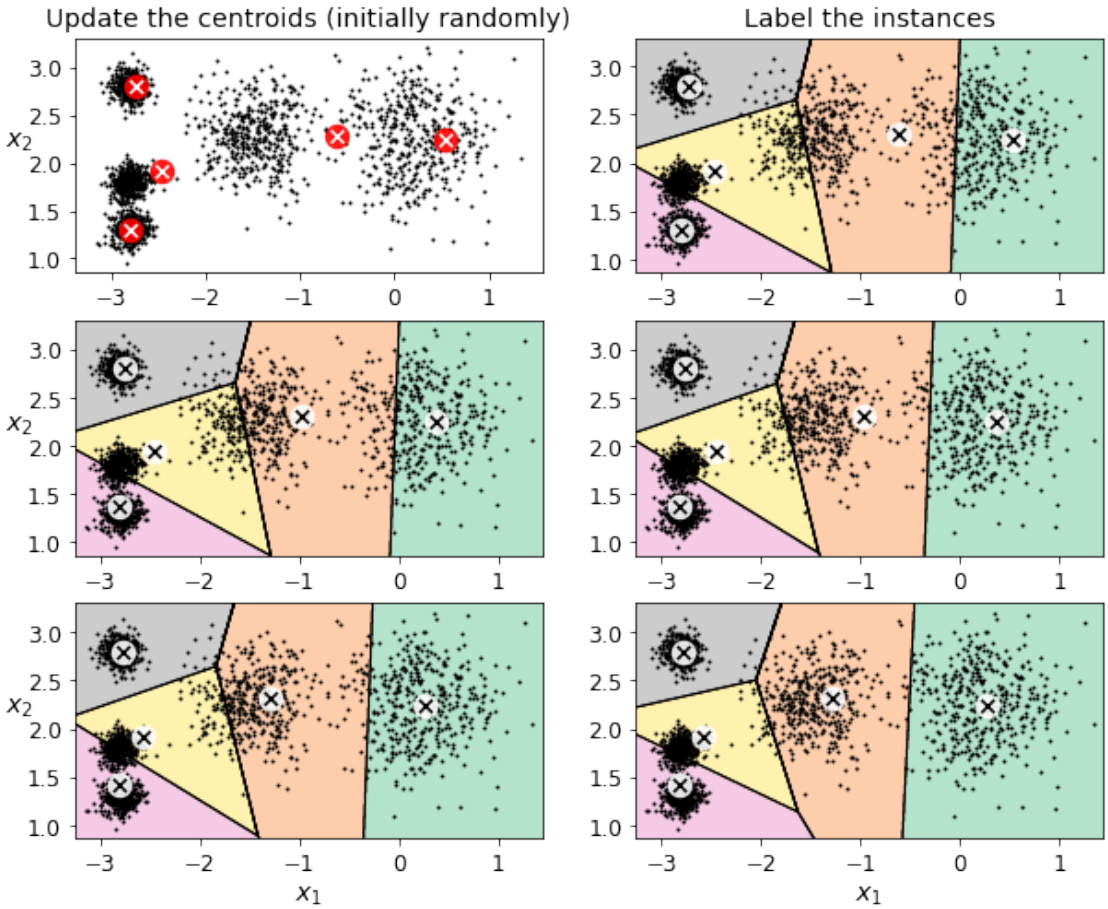
[104]:
```
kmeans_iter1 = KMeans(n_clusters=5, init="random", n_init=1,
                      algorithm="full", max_iter=1, random_state=1)
kmeans_iter2 = KMeans(n_clusters=5, init="random", n_init=1,
                      algorithm="full", max_iter=2, random_state=1)
kmeans_iter3 = KMeans(n_clusters=5, init="random", n_init=1,
                      algorithm="full", max_iter=3, random_state=1)
kmeans_iter1.fit(X)
kmeans_iter2.fit(X)
kmeans_iter3.fit(X)
```

[104]:
```
KMeans(algorithm='full', init='random', max_iter=3, n_clusters=5, n_init=1,
       random_state=1)
```

And let's plot this:

[105]:
```
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='off')
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)


plt.show()
```



13.2.5 K-Means Variability ST

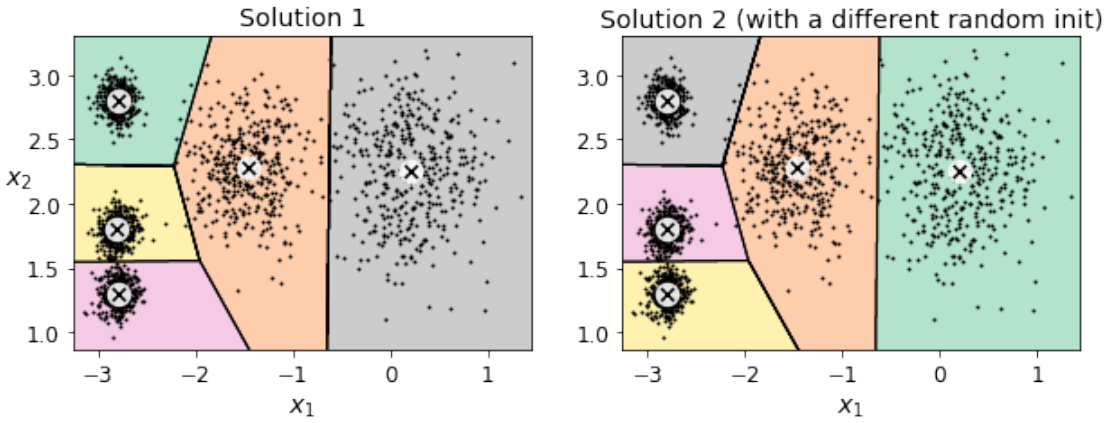
In the original K-Means algorithm, the centroids are just initialized randomly, and the algorithm simply runs a single iteration to gradually improve the centroids, as we saw above.

However, one major problem with this approach is that if you run K-Means multiple times (or with

different random seeds), it can converge to very different solutions, as you can see below:

[106]:
```
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)
```

[107]:
```
kmeans_rnd_init1 = KMeans(n_clusters=5, init="random", n_init=1,
                          algorithm="full", random_state=11)
kmeans_rnd_init2 = KMeans(n_clusters=5, init="random", n_init=1,
                          algorithm="full", random_state=19)

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

plt.show()
```



13.2.6 Inertia

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

[108]:
```
kmeans.inertia_
```

[108]:
```
211.5985372581683
```

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

$$Inertia = \sum_{j=0}^{C}(\sum_{i=0}^{N_c}(x^{(i)} - \mu^{(j)})^2)$$

[109]:
```
X_dist = kmeans.transform(X)
np.sum(X_dist[np.arange(len(X_dist)), kmeans.labels_]**2)
```

[109]:
```
211.5985372581684
```

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

[110]:
```
kmeans.score(X)
```

[110]:
```
-211.5985372581683
```

13.2.7 Multiple Initializations

So one approach to solve the variability issue is to simply run the K-Means algorithm multiple times with different random initializations, and select the solution that minimizes the inertia. For example, here are the inertias of the two "bad" models shown in the previous figure:

[111]:
```
kmeans_rnd_init1.inertia_
```

[111]:
```
211.60832621558365
```

[112]:
```
kmeans_rnd_init2.inertia_
```

[112]:
```
211.62301821329766
```

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

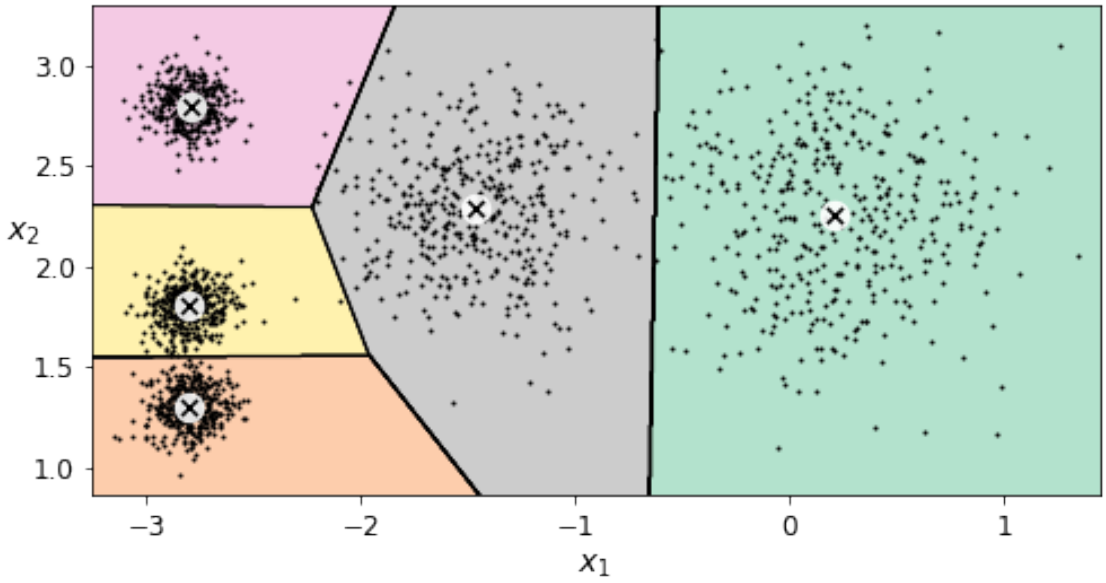
When you set the `n_init` hyperparameter, Scikit-Learn runs the original algorithm `n_init` times, and selects the solution that minimizes the inertia. By default, Scikit-Learn sets `n_init=10`.

[113]:
```
kmeans_rnd_10_inits = KMeans(n_clusters=5, init="random", n_init=10,
                             algorithm="full", random_state=11)
kmeans_rnd_10_inits.fit(X)
```

[113]:
```
KMeans(algorithm='full', init='random', n_clusters=5, random_state=11)
```

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

[114]:
```
plt.figure(figsize=(8, 4))
plot_decision_boundaries(kmeans_rnd_10_inits, X)
plt.show()
```



[115]:
```
kmeans_rnd_10_inits.inertia_
```

[115]:
```
211.5985372581683
```

13.2.8 K-Means++

Instead of initializing the centroids entirely randomly, it is preferable to initialize them using the following algorithm, proposed in a 2006 paper by David Arthur and Sergei Vassilvitskii: * Take one centroid c_1, chosen uniformly at random from the dataset. * Take a new center c_i, choosing an instance $\mathbf{x}_i$ with probability: $D(\mathbf{x}_i)^2 \,/\, \sum_{j=1}^{m} D(\mathbf{x}_j)^2$ where $D(\mathbf{x}_i)$ is the distance between the instance $\mathbf{x}_i$ and the closest centroid that was already chosen. This probability distribution ensures that instances that are further away from already chosen centroids are much more likely be selected as centroids. * Repeat the previous step until all k centroids have been chosen.

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

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

[116]:
```
KMeans()
```

[116]:
```
KMeans()
```

[117]:
```
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_
```

[117]:
```
211.62337889822365
```

13.2.9 Accelerated K-Means

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

To use Elkan's variant of K-Means, just set `algorithm="elkan"`. Note that it does not support sparse data, so by default, Scikit-Learn uses `"elkan"` for dense data, and `"full"` (the regular K-Means algorithm) for sparse data.

[118]:
```
%timeit -n 50 KMeans(algorithm="elkan").fit(X)
```

```
87.4 ms ± 3.13 ms per loop (mean ± std. dev. of 7 runs, 50 loops each)
```

[119]:
```
%timeit -n 50 KMeans(algorithm="full").fit(X)
```

```
82.7 ms ± 877 µs per loop (mean ± std. dev. of 7 runs, 50 loops each)
```

13.2.10 Mini-Batch K-Means

Scikit-Learn also implements a variant of the K-Means algorithm that supports mini-batches (see this paper):

[120]:
```
from sklearn.cluster import MiniBatchKMeans
```

[121]:
```
minibatch_kmeans = MiniBatchKMeans(n_clusters=5, random_state=42)
minibatch_kmeans.fit(X)
```

[121]:
```
MiniBatchKMeans(n_clusters=5, random_state=42)
```

[122]:
```
minibatch_kmeans.inertia_
```

[122]:
```
211.93186531476786
```

If the dataset does not fit in memory, the simplest option is to use the `memmap` class, just like we did for incremental PCA:

[123]:
```
filename = "my_mnist.data"
m, n = 50000, 28*28
X_mm = np.memmap(filename, dtype="float32", mode="readonly", shape=(m, n))
```

[124]:
```
minibatch_kmeans = MiniBatchKMeans(n_clusters=10, batch_size=10,␣
↪random_state=42)
minibatch_kmeans.fit(X_mm)
```

[124]:
```
MiniBatchKMeans(batch_size=10, n_clusters=10, random_state=42)
```

If your data is so large that you cannot use `memmap`, things get more complicated. Let's start by writing a function to load the next batch (in real life, you would load the data from disk):

[125]:
```
def load_next_batch(batch_size):
    return X[np.random.choice(len(X), batch_size, replace=False)]
```

Now we can train the model by feeding it one batch at a time. We also need to implement multiple initializations and keep the model with the lowest inertia:

[126]:
```
np.random.seed(42)
```

[127]:
```
k = 5
n_init = 10
n_iterations = 100
batch_size = 100
init_size = 500  # more data for K-Means++ initialization
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
```

[128]:
```
best_kmeans.score(X)
```

[128]:
```
-211.70999744411446
```

Mini-batch K-Means is much faster than regular K-Means:

[129]:
```
%timeit KMeans(n_clusters=5).fit(X)
```

```
49.6 ms ± 1.16 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[130]:
```
%timeit MiniBatchKMeans(n_clusters=5).fit(X)
```

```
15 ms ± 700 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

That's *much* faster! However, its performance is often lower (higher inertia), and it keeps degrading as k increases. Let's plot the inertia ratio and the training time ratio between Mini-batch K-Means and regular K-Means:

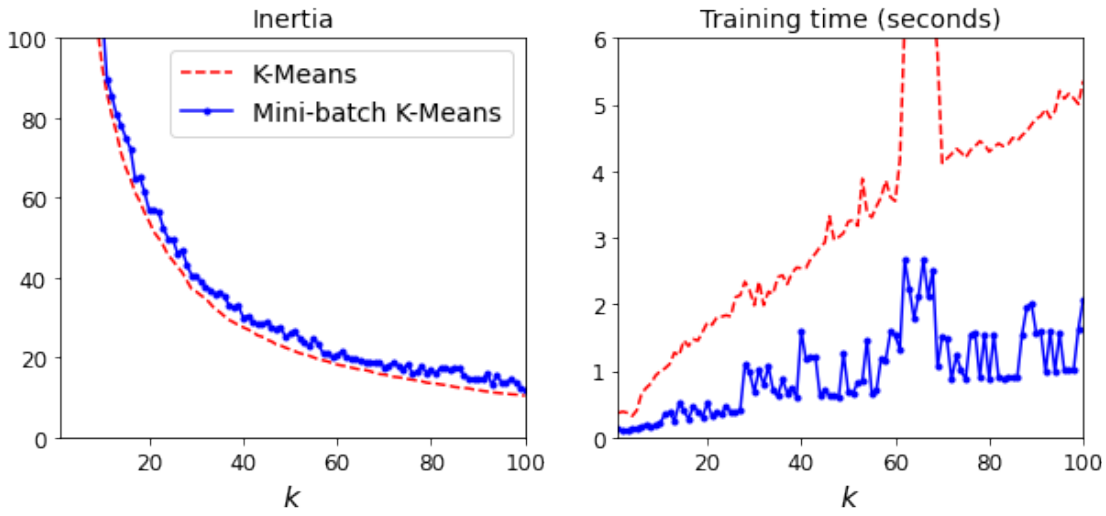
[131]:
```
from timeit import timeit
```

[132]:
```
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
```

[133]:
```
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.ylabel("Inertia", fontsize=14)
plt.title("Inertia", fontsize=14)
plt.legend(fontsize=14)
```

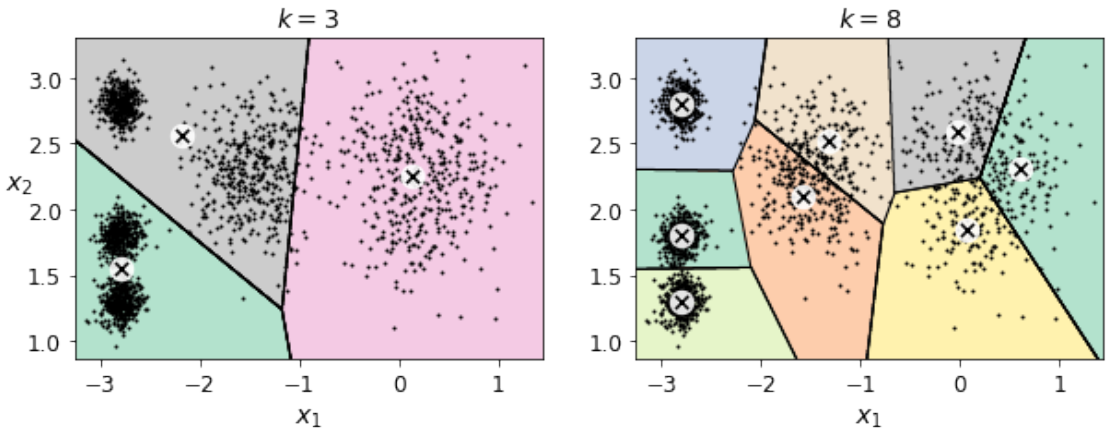
```
plt.axis([1, 100, 0, 100])

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.ylabel("Training time (seconds)", fontsize=14)
plt.title("Training time (seconds)", fontsize=14)
plt.axis([1, 100, 0, 6])
#plt.legend(fontsize=14)


plt.show()
```



13.2.11 Finding the optimal number of clusters

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

[134]:
```
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, "$k=3$", "$k=8$")

plt.show()
```



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

[135]:
```
kmeans_k3.inertia_
```

[135]:
```
653.2223267580945
```

[136]:
```
kmeans_k8.inertia_
```

[136]:
```
118.44108623570082
```

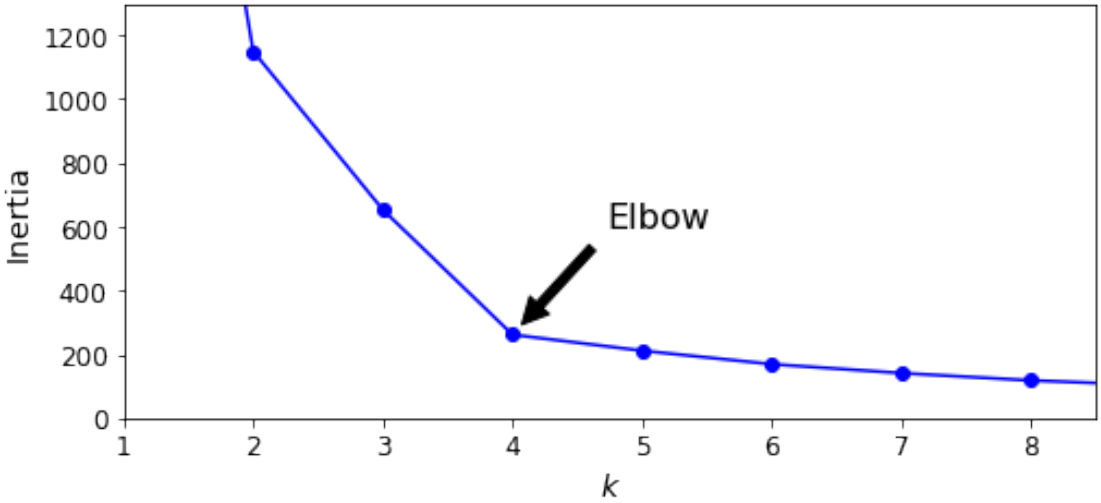
No, we cannot simply take the value of k that minimizes the inertia, since it keeps getting lower as we increase k. Inertia is the sum of distances between point and cluster centroids.

Indeed, the more clusters there are, the closer each instance will be to its closest centroid, and therefore the lower the inertia will be. However, we can plot the inertia as a function of k and analyze the resulting curve:

[137]:
```
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]
```

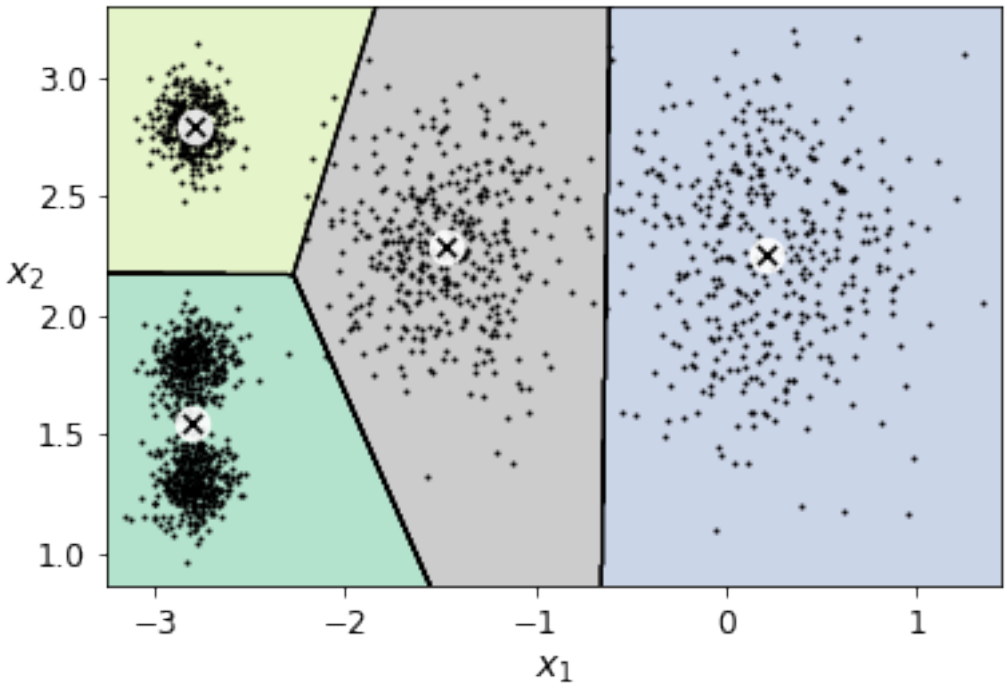
[138]:
```
plt.figure(figsize=(8, 3.5))
plt.plot(range(1, 10), inertias, "bo-")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Inertia", fontsize=14)
plt.annotate('Elbow',
             xy=(4, inertias[3]),
             xytext=(0.55, 0.55),
             textcoords='figure fraction',
             fontsize=16,
             arrowprops=dict(facecolor='black', shrink=0.1)
            )
plt.axis([1, 8.5, 0, 1300])
```

```
plt.show()
```



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

[139]:
```
plot_decision_boundaries(kmeans_per_k[4-1], X)
plt.show()
```



Another approach is to look at the *silhouette score*, which is the mean *silhouette coefficient* over all the instances. An instance's silhouette coefficient is equal to $(a - b)/\max(a, b)$ where a is the mean distance to the other instances in the same cluster (it is the *mean intra-cluster distance*), and b is the *mean nearest-cluster distance*, that is the mean distance to the instances of the next closest cluster (defined as the one that minimizes b, excluding the instance's own cluster).

The silhouette coefficient can vary between -1 and +1: a coefficient close to +1 means that the instance is well inside its own cluster and far from other clusters, while a coefficient close to 0 means that it is close to a cluster boundary, and finally a coefficient close to -1 means that the instance may have been assigned to the wrong cluster.

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

[140]:
```
from sklearn.metrics import silhouette_score
```

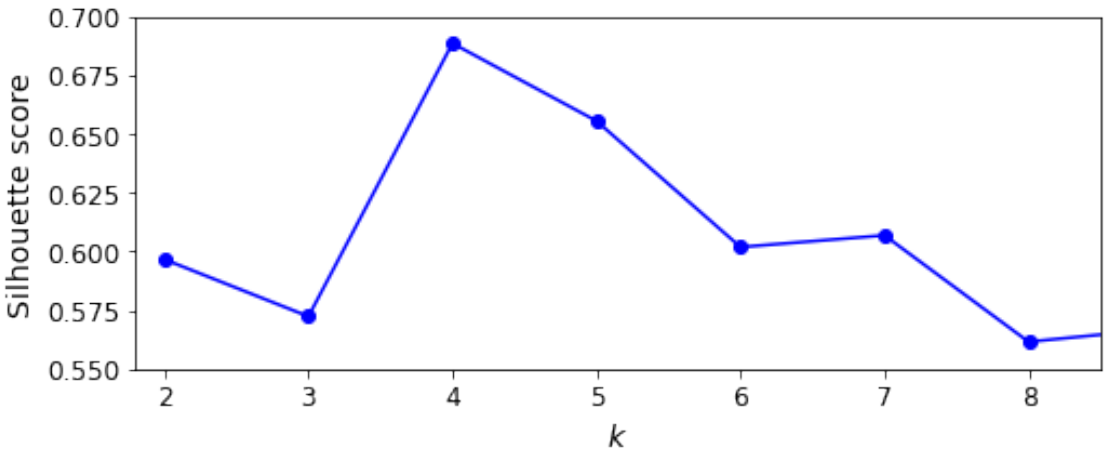
[141]:
```
silhouette_score(X, kmeans.labels_)
```

[141]:
```
0.3431853302333277
```

[142]:
```
silhouette_scores = [silhouette_score(X, model.labels_)
                     for model in kmeans_per_k[1:]]
```

[143]:
```
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])
plt.show()
```

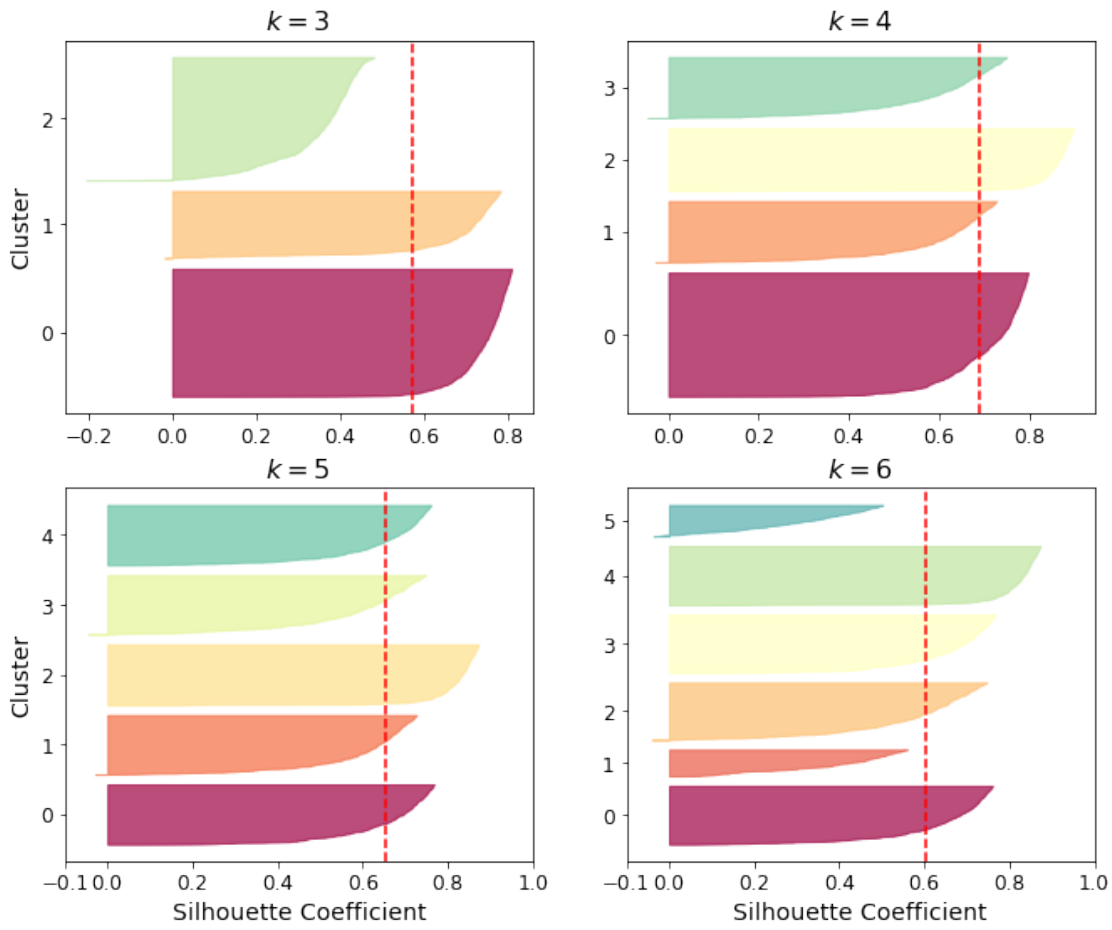


As you can see, this visualization is much richer than the previous one: in particular, although it confirms that $k = 4$ is a very good choice, but it also underlines the fact that $k = 5$ is quite good as well.

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

[144]:
```
from sklearn.metrics import silhouette_samples
from matplotlib.ticker import FixedLocator, FixedFormatter
import matplotlib.cm as cm
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()
        cmap = cm.get_cmap("Spectral")
        color = cmap(i / k)
```

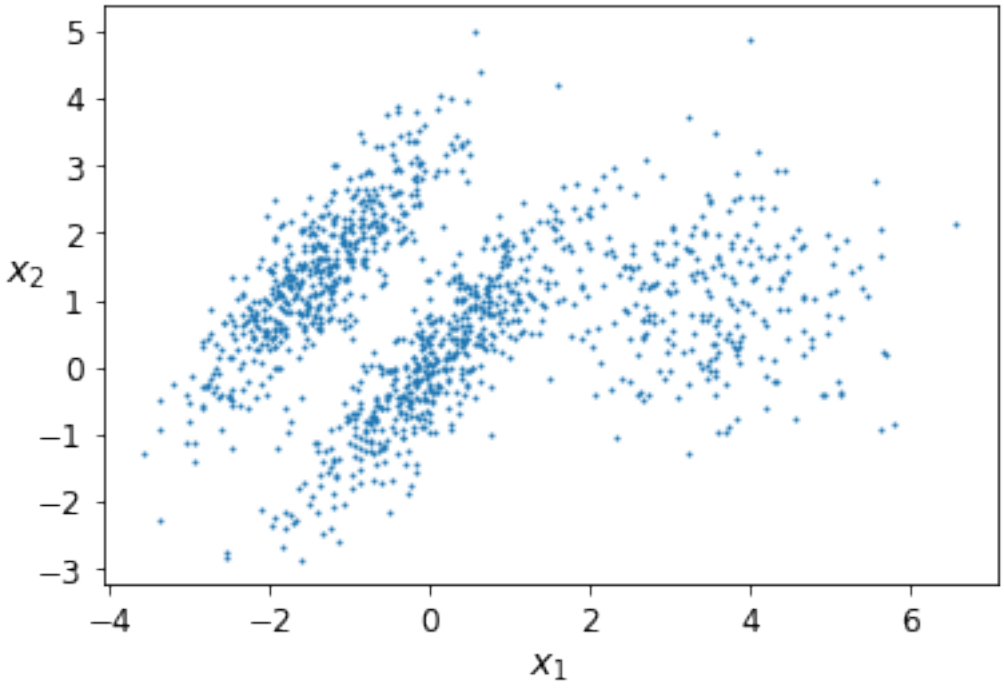
```
        plt.fill_betweenx(np.arange(pos, pos + len(coeffs)), 0, coeffs,
                          facecolor=color, edgecolor=color, alpha=0.7)
        ticks.append(pos + len(coeffs) // 2)
        pos += len(coeffs) + padding

    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='off')

    plt.axvline(x=silhouette_scores[k - 2], color="red", linestyle="--")
    plt.title("$k={}$".format(k), fontsize=16)

plt.show()
```



13.2.12 Limits of K-Means

[145]:
```
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]
```

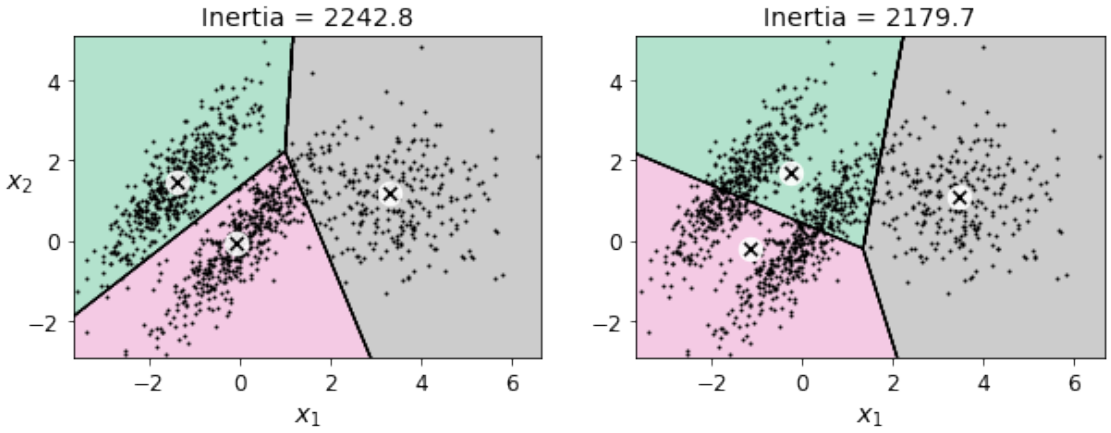
[146]:
```
plot_clusters(X)
```



[147]:
```
# Set centroids for initial clusters.
# n_init is the number of times the algorithm will be run with the inital␣
 ↪values, akin to random draws.
kmeans_good = KMeans(n_clusters=3, init=np.array([[-1.5, 2.5], [0.5, 0], [4,␣
 ↪0]]), n_init=1, random_state=42)
# No initial values
kmeans_bad = KMeans(n_clusters=3, random_state=42)
kmeans_good.fit(X)
kmeans_bad.fit(X)
```

[147]:
```
KMeans(n_clusters=3, random_state=42)
```

[148]:
```
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)

# Good has high inertia and bad has low inertia

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

plt.show()
```



13.2.13 Using clustering for image segmentation

[151]:
```
# Download the ladybug image
import urllib
images_path = os.path.join(PROJECT_ROOT_DIR, "images", "unsupervised_learning")
os.makedirs(images_path, exist_ok=True)
DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"
filename = "ladybug.png"
print("Downloading", filename)
url = DOWNLOAD_ROOT + "images/unsupervised_learning/" + filename
urllib.request.urlretrieve(url, os.path.join(images_path, filename))

from matplotlib.image import imread
image = imread(os.path.join("images","unsupervised_learning","ladybug.png"))
image.shape
```

```
Downloading ladybug.png
```

[151]:
```
(533, 800, 3)
```

[152]:
```
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)
```

[153]:
```
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))
```

[154]:
```
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')

plt.show()
```

13.2.14 Using Clustering for Preprocessing

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

[155]:
```
from sklearn.datasets import load_digits
```

[156]:
```
X_digits, y_digits = load_digits(return_X_y=True)
```

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

[157]:
```
from sklearn.model_selection import train_test_split
```

[158]:
```
X_train, X_test, y_train, y_test = train_test_split(X_digits, y_digits,␣
↪random_state=42)
```

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

[159]:
```
from sklearn.linear_model import LogisticRegression
```

[160]:
```
log_reg = LogisticRegression(multi_class="ovr", solver="lbfgs", max_iter=5000,␣
↪random_state=42)
log_reg.fit(X_train, y_train)
```

[160]:
```
LogisticRegression(max_iter=5000, multi_class='ovr', random_state=42)
```

[161]:
```
log_reg.score(X_test, y_test)
```

[161]:
```
0.9688888888888889
```

Okay, that's our baseline: 96.7% accuracy. Let's see if we can do better by using K-Means as a preprocessing step. We will create a pipeline that will first cluster the training set into 50 clusters and replace the images with their distances to the 50 clusters, then apply a logistic regression model:

[162]:
```
from sklearn.pipeline import Pipeline
```

[163]:
```
pipeline = Pipeline([
    ("kmeans", KMeans(n_clusters=50, random_state=42)),
    ("log_reg",  LogisticRegression(multi_class="ovr", solver="lbfgs",␣
↪max_iter=5000, random_state=42))
])
pipeline.fit(X_train, y_train)
```

[163]:
```
Pipeline(steps=[('kmeans', KMeans(n_clusters=50, random_state=42)),
                ('log_reg',
```

```
                    LogisticRegression(max_iter=5000, multi_class='ovr',
                                       random_state=42))])
```

[164]:
```
pipeline.score(X_test, y_test)
```

[164]:
```
0.98
```

[165]:
```
1 - (1 - 0.98) / (1 - 0.9666666)
```

[165]:
```
0.40000119999759876
```

How about that? We almost divided the error rate by a factor of 2! But we chose the number of clusters k completely arbitrarily, we can surely do better. Since K-Means is just a preprocessing step in a classification pipeline, finding a good value for k is much simpler than earlier: there's no need to perform silhouette analysis or minimize the inertia, the best value of k is simply the one that results in the best classification performance.

[166]:
```
from sklearn.model_selection import GridSearchCV
```

[167]:
```
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=   0.1s
[CV] kmeans__n_clusters=3 …
[CV] … kmeans__n_clusters=3, total=   0.1s
[CV] kmeans__n_clusters=4 …
[CV] … kmeans__n_clusters=4, total=   0.1s
[CV] kmeans__n_clusters=4 …
[CV] … kmeans__n_clusters=4, total=   0.1s
[CV] kmeans__n_clusters=4 …
[CV] … kmeans__n_clusters=4, total=   0.1s
[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=   0.2s
[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=   0.3s
[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, total=   0.5s
[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=   0.5s
[CV] kmeans__n_clusters=12 …
[CV] … kmeans__n_clusters=12, 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=   0.7s
[CV] kmeans__n_clusters=13 …
[CV] … kmeans__n_clusters=13, total=   0.8s
[CV] kmeans__n_clusters=13 …
```

```
[CV] … kmeans__n_clusters=13, total=   0.8s
[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=   1.1s
[CV] kmeans__n_clusters=15 …
[CV] … kmeans__n_clusters=15, total=   1.0s
[CV] kmeans__n_clusters=15 …
[CV] … kmeans__n_clusters=15, total=   1.2s
[CV] kmeans__n_clusters=16 …
[CV] … kmeans__n_clusters=16, total=   1.4s
[CV] kmeans__n_clusters=16 …
[CV] … kmeans__n_clusters=16, total=   1.4s
[CV] kmeans__n_clusters=16 …
[CV] … kmeans__n_clusters=16, total=   1.5s
[CV] kmeans__n_clusters=17 …
[CV] … kmeans__n_clusters=17, total=   1.7s
[CV] kmeans__n_clusters=17 …
[CV] … kmeans__n_clusters=17, total=   1.3s
[CV] kmeans__n_clusters=17 …
[CV] … kmeans__n_clusters=17, total=   1.2s
[CV] kmeans__n_clusters=18 …
[CV] … kmeans__n_clusters=18, total=   1.6s
[CV] kmeans__n_clusters=18 …
[CV] … kmeans__n_clusters=18, total=   1.4s
[CV] kmeans__n_clusters=18 …
[CV] … kmeans__n_clusters=18, total=   1.4s
[CV] kmeans__n_clusters=19 …
[CV] … kmeans__n_clusters=19, total=   1.6s
[CV] kmeans__n_clusters=19 …
[CV] … kmeans__n_clusters=19, total=   1.4s
[CV] kmeans__n_clusters=19 …
[CV] … kmeans__n_clusters=19, total=   1.3s
[CV] kmeans__n_clusters=20 …
[CV] … kmeans__n_clusters=20, total=   1.6s
[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=   1.6s
[CV] kmeans__n_clusters=21 …
[CV] … kmeans__n_clusters=21, total=   1.8s
[CV] kmeans__n_clusters=22 …
[CV] … kmeans__n_clusters=22, total=   1.8s
[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=   2.0s
[CV] kmeans__n_clusters=23 …
[CV] … kmeans__n_clusters=23, total=   2.0s
[CV] kmeans__n_clusters=24 …
[CV] … kmeans__n_clusters=24, total=   2.2s
[CV] kmeans__n_clusters=24 …
[CV] … kmeans__n_clusters=24, total=   2.0s
[CV] kmeans__n_clusters=24 …
[CV] … kmeans__n_clusters=24, total=   2.0s
[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=25 …
[CV] … kmeans__n_clusters=25, total=   1.9s
[CV] kmeans__n_clusters=26 …
[CV] … kmeans__n_clusters=26, total=   2.1s
[CV] kmeans__n_clusters=26 …
[CV] … kmeans__n_clusters=26, total=   2.1s
[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=   2.0s
[CV] kmeans__n_clusters=27 …
[CV] … kmeans__n_clusters=27, total=   2.0s
[CV] kmeans__n_clusters=28 …
[CV] … kmeans__n_clusters=28, total=   2.1s
[CV] kmeans__n_clusters=28 …
[CV] … kmeans__n_clusters=28, total=   2.1s
[CV] kmeans__n_clusters=28 …
[CV] … kmeans__n_clusters=28, total=   2.2s
[CV] kmeans__n_clusters=29 …
[CV] … kmeans__n_clusters=29, total=   2.3s
[CV] kmeans__n_clusters=29 …
```

```
[CV] … kmeans__n_clusters=29, total=   2.2s
[CV] kmeans__n_clusters=29 …
[CV] … kmeans__n_clusters=29, total=   2.0s
[CV] kmeans__n_clusters=30 …
[CV] … kmeans__n_clusters=30, total=   2.3s
[CV] kmeans__n_clusters=30 …
[CV] … kmeans__n_clusters=30, total=   2.2s
[CV] kmeans__n_clusters=30 …
[CV] … kmeans__n_clusters=30, total=   1.9s
[CV] kmeans__n_clusters=31 …
[CV] … kmeans__n_clusters=31, total=   2.1s
[CV] kmeans__n_clusters=31 …
[CV] … kmeans__n_clusters=31, total=   2.2s
[CV] kmeans__n_clusters=31 …
[CV] … kmeans__n_clusters=31, total=   2.1s
[CV] kmeans__n_clusters=32 …
[CV] … kmeans__n_clusters=32, total=   2.3s
[CV] kmeans__n_clusters=32 …
[CV] … kmeans__n_clusters=32, total=   2.2s
[CV] kmeans__n_clusters=32 …
[CV] … kmeans__n_clusters=32, total=   2.1s
[CV] kmeans__n_clusters=33 …
[CV] … kmeans__n_clusters=33, total=   2.2s
[CV] kmeans__n_clusters=33 …
[CV] … kmeans__n_clusters=33, total=   2.5s
[CV] kmeans__n_clusters=33 …
[CV] … kmeans__n_clusters=33, total=   2.1s
[CV] kmeans__n_clusters=34 …
[CV] … kmeans__n_clusters=34, total=   2.8s
[CV] kmeans__n_clusters=34 …
[CV] … kmeans__n_clusters=34, total=   2.7s
[CV] kmeans__n_clusters=34 …
[CV] … kmeans__n_clusters=34, total=   2.9s
[CV] kmeans__n_clusters=35 …
[CV] … kmeans__n_clusters=35, total=   2.5s
[CV] kmeans__n_clusters=35 …
[CV] … kmeans__n_clusters=35, total=   2.6s
[CV] kmeans__n_clusters=35 …
[CV] … kmeans__n_clusters=35, total=   2.5s
[CV] kmeans__n_clusters=36 …
[CV] … kmeans__n_clusters=36, total=   2.4s
[CV] kmeans__n_clusters=36 …
[CV] … kmeans__n_clusters=36, total=   2.5s
[CV] kmeans__n_clusters=36 …
[CV] … kmeans__n_clusters=36, total=   2.5s
[CV] kmeans__n_clusters=37 …
[CV] … kmeans__n_clusters=37, total=   2.4s
[CV] kmeans__n_clusters=37 …
```

```
[CV] … kmeans__n_clusters=37, total=   2.4s
[CV] kmeans__n_clusters=37 …
[CV] … kmeans__n_clusters=37, total=   2.6s
[CV] kmeans__n_clusters=38 …
[CV] … kmeans__n_clusters=38, total=   2.9s
[CV] kmeans__n_clusters=38 …
[CV] … kmeans__n_clusters=38, total=   2.9s
[CV] kmeans__n_clusters=38 …
[CV] … kmeans__n_clusters=38, total=   3.3s
[CV] kmeans__n_clusters=39 …
[CV] … kmeans__n_clusters=39, total=   2.4s
[CV] kmeans__n_clusters=39 …
[CV] … kmeans__n_clusters=39, total=   3.2s
[CV] kmeans__n_clusters=39 …
[CV] … kmeans__n_clusters=39, total=   2.5s
[CV] kmeans__n_clusters=40 …
[CV] … kmeans__n_clusters=40, total=   2.7s
[CV] kmeans__n_clusters=40 …
[CV] … kmeans__n_clusters=40, total=   2.9s
[CV] kmeans__n_clusters=40 …
[CV] … kmeans__n_clusters=40, total=   2.7s
[CV] kmeans__n_clusters=41 …
[CV] … kmeans__n_clusters=41, total=   2.6s
[CV] kmeans__n_clusters=41 …
[CV] … kmeans__n_clusters=41, total=   2.8s
[CV] kmeans__n_clusters=41 …
[CV] … kmeans__n_clusters=41, total=   2.7s
[CV] kmeans__n_clusters=42 …
[CV] … kmeans__n_clusters=42, total=   2.8s
[CV] kmeans__n_clusters=42 …
[CV] … kmeans__n_clusters=42, total=   2.7s
[CV] kmeans__n_clusters=42 …
[CV] … kmeans__n_clusters=42, total=   2.5s
[CV] kmeans__n_clusters=43 …
[CV] … kmeans__n_clusters=43, total=   3.0s
[CV] kmeans__n_clusters=43 …
[CV] … kmeans__n_clusters=43, total=   3.4s
[CV] kmeans__n_clusters=43 …
[CV] … kmeans__n_clusters=43, total=   3.0s
[CV] kmeans__n_clusters=44 …
[CV] … kmeans__n_clusters=44, total=   3.5s
[CV] kmeans__n_clusters=44 …
[CV] … kmeans__n_clusters=44, total=   3.5s
[CV] kmeans__n_clusters=44 …
[CV] … kmeans__n_clusters=44, total=   3.5s
[CV] kmeans__n_clusters=45 …
[CV] … kmeans__n_clusters=45, total=   2.7s
[CV] kmeans__n_clusters=45 …
```

```
[CV] … kmeans__n_clusters=45, total=   2.5s
[CV] kmeans__n_clusters=45 …
[CV] … kmeans__n_clusters=45, total=   2.7s
[CV] kmeans__n_clusters=46 …
[CV] … kmeans__n_clusters=46, total=   2.6s
[CV] kmeans__n_clusters=46 …
[CV] … kmeans__n_clusters=46, total=   2.7s
[CV] kmeans__n_clusters=46 …
[CV] … kmeans__n_clusters=46, total=   2.8s
[CV] kmeans__n_clusters=47 …
[CV] … kmeans__n_clusters=47, total=   2.6s
[CV] kmeans__n_clusters=47 …
[CV] … kmeans__n_clusters=47, total=   2.6s
[CV] kmeans__n_clusters=47 …
[CV] … kmeans__n_clusters=47, total=   2.4s
[CV] kmeans__n_clusters=48 …
[CV] … kmeans__n_clusters=48, total=   2.7s
[CV] kmeans__n_clusters=48 …
[CV] … kmeans__n_clusters=48, total=   2.8s
[CV] kmeans__n_clusters=48 …
[CV] … kmeans__n_clusters=48, total=   2.8s
[CV] kmeans__n_clusters=49 …
[CV] … kmeans__n_clusters=49, total=   2.6s
[CV] kmeans__n_clusters=49 …
[CV] … kmeans__n_clusters=49, total=   2.7s
[CV] kmeans__n_clusters=49 …
[CV] … kmeans__n_clusters=49, total=   2.8s
[CV] kmeans__n_clusters=50 …
[CV] … kmeans__n_clusters=50, total=   2.8s
[CV] kmeans__n_clusters=50 …
[CV] … kmeans__n_clusters=50, total=   2.9s
[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=   2.9s
[CV] kmeans__n_clusters=51 …
[CV] … kmeans__n_clusters=51, total=   3.4s
[CV] kmeans__n_clusters=52 …
[CV] … kmeans__n_clusters=52, total=   3.5s
[CV] kmeans__n_clusters=52 …
[CV] … kmeans__n_clusters=52, total=   3.2s
[CV] kmeans__n_clusters=52 …
[CV] … kmeans__n_clusters=52, total=   2.8s
[CV] kmeans__n_clusters=53 …
[CV] … kmeans__n_clusters=53, total=   2.8s
[CV] kmeans__n_clusters=53 …
```

```
[CV] … kmeans__n_clusters=53, total=   2.9s
[CV] kmeans__n_clusters=53 …
[CV] … kmeans__n_clusters=53, total=   2.9s
[CV] kmeans__n_clusters=54 …
[CV] … kmeans__n_clusters=54, total=   3.3s
[CV] kmeans__n_clusters=54 …
[CV] … kmeans__n_clusters=54, total=   2.9s
[CV] kmeans__n_clusters=54 …
[CV] … kmeans__n_clusters=54, total=   3.1s
[CV] kmeans__n_clusters=55 …
[CV] … kmeans__n_clusters=55, total=   2.6s
[CV] kmeans__n_clusters=55 …
[CV] … kmeans__n_clusters=55, total=   3.1s
[CV] kmeans__n_clusters=55 …
[CV] … kmeans__n_clusters=55, total=   2.9s
[CV] kmeans__n_clusters=56 …
[CV] … kmeans__n_clusters=56, total=   2.9s
[CV] kmeans__n_clusters=56 …
[CV] … kmeans__n_clusters=56, total=   3.0s
[CV] kmeans__n_clusters=56 …
[CV] … kmeans__n_clusters=56, total=   2.6s
[CV] kmeans__n_clusters=57 …
[CV] … kmeans__n_clusters=57, total=   2.7s
[CV] kmeans__n_clusters=57 …
[CV] … kmeans__n_clusters=57, total=   3.2s
[CV] kmeans__n_clusters=57 …
[CV] … kmeans__n_clusters=57, total=   2.9s
[CV] kmeans__n_clusters=58 …
[CV] … kmeans__n_clusters=58, total=   3.0s
[CV] kmeans__n_clusters=58 …
[CV] … kmeans__n_clusters=58, total=   2.8s
[CV] kmeans__n_clusters=58 …
[CV] … kmeans__n_clusters=58, total=   2.9s
[CV] kmeans__n_clusters=59 …
[CV] … kmeans__n_clusters=59, total=   3.1s
[CV] kmeans__n_clusters=59 …
[CV] … kmeans__n_clusters=59, total=   3.2s
[CV] kmeans__n_clusters=59 …
[CV] … kmeans__n_clusters=59, total=   3.1s
[CV] kmeans__n_clusters=60 …
[CV] … kmeans__n_clusters=60, total=   3.3s
[CV] kmeans__n_clusters=60 …
[CV] … kmeans__n_clusters=60, total=   3.4s
[CV] kmeans__n_clusters=60 …
[CV] … kmeans__n_clusters=60, total=   3.0s
[CV] kmeans__n_clusters=61 …
[CV] … kmeans__n_clusters=61, total=   3.1s
[CV] kmeans__n_clusters=61 …
```

```
[CV] … kmeans__n_clusters=61, total=   3.4s
[CV] kmeans__n_clusters=61 …
[CV] … kmeans__n_clusters=61, total=   2.6s
[CV] kmeans__n_clusters=62 …
[CV] … kmeans__n_clusters=62, total=   3.2s
[CV] kmeans__n_clusters=62 …
[CV] … kmeans__n_clusters=62, total=   3.4s
[CV] kmeans__n_clusters=62 …
[CV] … kmeans__n_clusters=62, total=   3.2s
[CV] kmeans__n_clusters=63 …
[CV] … kmeans__n_clusters=63, total=   3.1s
[CV] kmeans__n_clusters=63 …
[CV] … kmeans__n_clusters=63, total=   3.2s
[CV] kmeans__n_clusters=63 …
[CV] … kmeans__n_clusters=63, total=   3.2s
[CV] kmeans__n_clusters=64 …
[CV] … kmeans__n_clusters=64, total=   3.3s
[CV] kmeans__n_clusters=64 …
[CV] … kmeans__n_clusters=64, total=   3.4s
[CV] kmeans__n_clusters=64 …
[CV] … kmeans__n_clusters=64, total=   3.2s
[CV] kmeans__n_clusters=65 …
[CV] … kmeans__n_clusters=65, total=   2.8s
[CV] kmeans__n_clusters=65 …
[CV] … kmeans__n_clusters=65, total=   3.3s
[CV] kmeans__n_clusters=65 …
[CV] … kmeans__n_clusters=65, total=   3.1s
[CV] kmeans__n_clusters=66 …
[CV] … kmeans__n_clusters=66, total=   3.0s
[CV] kmeans__n_clusters=66 …
[CV] … kmeans__n_clusters=66, total=   3.5s
[CV] kmeans__n_clusters=66 …
[CV] … kmeans__n_clusters=66, total=   2.9s
[CV] kmeans__n_clusters=67 …
[CV] … kmeans__n_clusters=67, total=   3.2s
[CV] kmeans__n_clusters=67 …
[CV] … kmeans__n_clusters=67, total=   3.2s
[CV] kmeans__n_clusters=67 …
[CV] … kmeans__n_clusters=67, total=   3.2s
[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=68 …
[CV] … kmeans__n_clusters=68, total=   3.0s
[CV] kmeans__n_clusters=69 …
[CV] … kmeans__n_clusters=69, total=   3.1s
[CV] kmeans__n_clusters=69 …
```

```
[CV] … kmeans__n_clusters=69, total=   3.2s
[CV] kmeans__n_clusters=69 …
[CV] … kmeans__n_clusters=69, total=   3.3s
[CV] kmeans__n_clusters=70 …
[CV] … kmeans__n_clusters=70, total=   3.1s
[CV] kmeans__n_clusters=70 …
[CV] … kmeans__n_clusters=70, total=   3.1s
[CV] kmeans__n_clusters=70 …
[CV] … kmeans__n_clusters=70, total=   3.2s
[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=   3.5s
[CV] kmeans__n_clusters=72 …
[CV] … kmeans__n_clusters=72, total=   2.9s
[CV] kmeans__n_clusters=72 …
[CV] … kmeans__n_clusters=72, total=   3.6s
[CV] kmeans__n_clusters=72 …
[CV] … kmeans__n_clusters=72, total=   3.1s
[CV] kmeans__n_clusters=73 …
[CV] … kmeans__n_clusters=73, total=   3.1s
[CV] kmeans__n_clusters=73 …
[CV] … kmeans__n_clusters=73, total=   3.7s
[CV] kmeans__n_clusters=73 …
[CV] … kmeans__n_clusters=73, total=   3.1s
[CV] kmeans__n_clusters=74 …
[CV] … kmeans__n_clusters=74, total=   2.9s
[CV] kmeans__n_clusters=74 …
[CV] … kmeans__n_clusters=74, total=   3.2s
[CV] kmeans__n_clusters=74 …
[CV] … kmeans__n_clusters=74, total=   3.3s
[CV] kmeans__n_clusters=75 …
[CV] … kmeans__n_clusters=75, total=   2.9s
[CV] kmeans__n_clusters=75 …
[CV] … kmeans__n_clusters=75, total=   3.4s
[CV] kmeans__n_clusters=75 …
[CV] … kmeans__n_clusters=75, total=   3.1s
[CV] kmeans__n_clusters=76 …
[CV] … kmeans__n_clusters=76, total=   2.8s
[CV] kmeans__n_clusters=76 …
[CV] … kmeans__n_clusters=76, total=   3.3s
[CV] kmeans__n_clusters=76 …
[CV] … kmeans__n_clusters=76, total=   3.4s
[CV] kmeans__n_clusters=77 …
[CV] … kmeans__n_clusters=77, total=   2.9s
[CV] kmeans__n_clusters=77 …
```

```
[CV] … kmeans__n_clusters=77, total=   2.7s
[CV] kmeans__n_clusters=77 …
[CV] … kmeans__n_clusters=77, total=   3.3s
[CV] kmeans__n_clusters=78 …
[CV] … kmeans__n_clusters=78, total=   2.9s
[CV] kmeans__n_clusters=78 …
[CV] … kmeans__n_clusters=78, total=   3.1s
[CV] kmeans__n_clusters=78 …
[CV] … kmeans__n_clusters=78, total=   3.1s
[CV] kmeans__n_clusters=79 …
[CV] … kmeans__n_clusters=79, total=   3.2s
[CV] kmeans__n_clusters=79 …
[CV] … kmeans__n_clusters=79, total=   3.3s
[CV] kmeans__n_clusters=79 …
[CV] … kmeans__n_clusters=79, total=   3.2s
[CV] kmeans__n_clusters=80 …
[CV] … kmeans__n_clusters=80, total=   2.9s
[CV] kmeans__n_clusters=80 …
[CV] … kmeans__n_clusters=80, total=   3.5s
[CV] kmeans__n_clusters=80 …
[CV] … kmeans__n_clusters=80, total=   3.0s
[CV] kmeans__n_clusters=81 …
[CV] … kmeans__n_clusters=81, total=   3.0s
[CV] kmeans__n_clusters=81 …
[CV] … kmeans__n_clusters=81, total=   3.5s
[CV] kmeans__n_clusters=81 …
[CV] … kmeans__n_clusters=81, total=   3.2s
[CV] kmeans__n_clusters=82 …
[CV] … kmeans__n_clusters=82, total=   3.0s
[CV] kmeans__n_clusters=82 …
[CV] … kmeans__n_clusters=82, total=   3.1s
[CV] kmeans__n_clusters=82 …
[CV] … kmeans__n_clusters=82, total=   3.5s
[CV] kmeans__n_clusters=83 …
[CV] … kmeans__n_clusters=83, total=   3.1s
[CV] kmeans__n_clusters=83 …
[CV] … kmeans__n_clusters=83, total=   3.1s
[CV] kmeans__n_clusters=83 …
[CV] … kmeans__n_clusters=83, total=   3.1s
[CV] kmeans__n_clusters=84 …
[CV] … kmeans__n_clusters=84, total=   3.2s
[CV] kmeans__n_clusters=84 …
[CV] … kmeans__n_clusters=84, total=   3.4s
[CV] kmeans__n_clusters=84 …
[CV] … kmeans__n_clusters=84, total=   3.2s
[CV] kmeans__n_clusters=85 …
[CV] … kmeans__n_clusters=85, total=   2.8s
[CV] kmeans__n_clusters=85 …
```

```
[CV] … kmeans__n_clusters=85, total=   3.4s
[CV] kmeans__n_clusters=85 …
[CV] … kmeans__n_clusters=85, total=   3.0s
[CV] kmeans__n_clusters=86 …
[CV] … kmeans__n_clusters=86, total=   3.1s
[CV] kmeans__n_clusters=86 …
[CV] … kmeans__n_clusters=86, total=   3.5s
[CV] kmeans__n_clusters=86 …
[CV] … kmeans__n_clusters=86, total=   3.0s
[CV] kmeans__n_clusters=87 …
[CV] … kmeans__n_clusters=87, total=   3.0s
[CV] kmeans__n_clusters=87 …
[CV] … kmeans__n_clusters=87, total=   3.1s
[CV] kmeans__n_clusters=87 …
[CV] … kmeans__n_clusters=87, total=   3.4s
[CV] kmeans__n_clusters=88 …
[CV] … kmeans__n_clusters=88, total=   2.9s
[CV] kmeans__n_clusters=88 …
[CV] … kmeans__n_clusters=88, total=   3.0s
[CV] kmeans__n_clusters=88 …
[CV] … kmeans__n_clusters=88, total=   3.2s
[CV] kmeans__n_clusters=89 …
[CV] … kmeans__n_clusters=89, 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=   3.2s
[CV] kmeans__n_clusters=90 …
[CV] … kmeans__n_clusters=90, total=   3.1s
[CV] kmeans__n_clusters=90 …
[CV] … kmeans__n_clusters=90, total=   3.0s
[CV] kmeans__n_clusters=90 …
[CV] … kmeans__n_clusters=90, total=   3.1s
[CV] kmeans__n_clusters=91 …
[CV] … kmeans__n_clusters=91, total=   3.4s
[CV] kmeans__n_clusters=91 …
[CV] … kmeans__n_clusters=91, total=   3.3s
[CV] kmeans__n_clusters=91 …
[CV] … kmeans__n_clusters=91, total=   3.2s
[CV] kmeans__n_clusters=92 …
[CV] … kmeans__n_clusters=92, total=   3.1s
[CV] kmeans__n_clusters=92 …
[CV] … kmeans__n_clusters=92, total=   3.2s
[CV] kmeans__n_clusters=92 …
[CV] … kmeans__n_clusters=92, total=   3.0s
[CV] kmeans__n_clusters=93 …
[CV] … kmeans__n_clusters=93, total=   3.4s
[CV] kmeans__n_clusters=93 …
```

```
[CV] … kmeans__n_clusters=93, total=   3.2s
[CV] kmeans__n_clusters=93 …
[CV] … kmeans__n_clusters=93, total=   2.8s
[CV] kmeans__n_clusters=94 …
[CV] … kmeans__n_clusters=94, total=   3.1s
[CV] kmeans__n_clusters=94 …
[CV] … kmeans__n_clusters=94, total=   3.1s
[CV] kmeans__n_clusters=94 …
[CV] … kmeans__n_clusters=94, total=   2.9s
[CV] kmeans__n_clusters=95 …
[CV] … kmeans__n_clusters=95, total=   2.8s
[CV] kmeans__n_clusters=95 …
[CV] … kmeans__n_clusters=95, total=   3.2s
[CV] kmeans__n_clusters=95 …
[CV] … kmeans__n_clusters=95, total=   3.0s
[CV] kmeans__n_clusters=96 …
[CV] … kmeans__n_clusters=96, total=   3.1s
[CV] kmeans__n_clusters=96 …
[CV] … kmeans__n_clusters=96, total=   3.2s
[CV] kmeans__n_clusters=96 …
[CV] … kmeans__n_clusters=96, total=   3.0s
[CV] kmeans__n_clusters=97 …
[CV] … kmeans__n_clusters=97, total=   3.1s
[CV] kmeans__n_clusters=97 …
[CV] … kmeans__n_clusters=97, total=   3.4s
[CV] kmeans__n_clusters=97 …
[CV] … kmeans__n_clusters=97, total=   3.0s
[CV] kmeans__n_clusters=98 …
[CV] … kmeans__n_clusters=98, total=   3.3s
[CV] kmeans__n_clusters=98 …
[CV] … kmeans__n_clusters=98, total=   3.6s
[CV] kmeans__n_clusters=98 …
[CV] … kmeans__n_clusters=98, total=   3.0s
[CV] kmeans__n_clusters=99 …
[CV] … kmeans__n_clusters=99, total=   2.9s
[CV] kmeans__n_clusters=99 …
[CV] … kmeans__n_clusters=99, total=   3.6s
[CV] kmeans__n_clusters=99 …
[CV] … kmeans__n_clusters=99, total=   2.9s

[Parallel(n_jobs=1)]: Done 294 out of 294 | elapsed: 12.0min finished
```

```
[167]: GridSearchCV(cv=3,
                    estimator=Pipeline(steps=[('kmeans',
                                               KMeans(n_clusters=50, random_state=42)),
                                              ('log_reg',
                                               LogisticRegression(max_iter=5000,
                                                                  multi_class='ovr',
```

```
                                                        random_state=42))]),
             param_grid={'kmeans__n_clusters': range(2, 100)}, verbose=2)
```

[168]:
```
grid_clf.best_params_
```

[168]:
```
{'kmeans__n_clusters': 57}
```

[169]:
```
grid_clf.score(X_test, y_test)
```

[169]:
```
0.98
```

The performance is slightly improved when $k = 57$, so 57 it is.

13.2.15 Clustering for Semi-supervised Learning

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

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

[170]:
```
# Imagine we start with just 50 labeled instances.
n_labeled = 50
```

[171]:
```
log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
log_reg.score(X_test, y_test)
```

```
/Users/ir177/opt/anaconda3/lib/python3.8/site-
packages/sklearn/linear_model/_logistic.py:762: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
```

[171]:
```
0.8266666666666667
```

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

[172]:
```
k = 50
```

[173]:
```
# Create 50 clusters
kmeans = KMeans(n_clusters=k, random_state=42)
X_digits_dist = kmeans.fit_transform(X_train)
# Find representative digit in the middle of the cluster
representative_digit_idx = np.argmin(X_digits_dist, axis=0)
# Assign representative digit to the cluster as a whole
X_representative_digits = X_train[representative_digit_idx]
```

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

[174]:
```
plt.figure(figsize=(8, 2))
for index, X_representative_digit in enumerate(X_representative_digits):
    plt.subplot(k // 10, 10, index + 1)
    plt.imshow(X_representative_digit.reshape(8, 8), cmap="binary",␣
 ↪interpolation="bilinear")
    plt.axis('off')


plt.show()
```

[175]:
```
y_representative_digits = np.array([
    4, 8, 0, 6, 8, 3, 7, 7, 9, 2,
    5, 5, 8, 5, 2, 1, 2, 9, 6, 1,
    1, 6, 9, 0, 8, 3, 0, 7, 4, 1,
    6, 5, 2, 4, 1, 8, 6, 3, 9, 2,
    4, 2, 9, 4, 7, 6, 2, 3, 1, 1])
```

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

[177]:
```
log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_representative_digits, y_representative_digits)
log_reg.score(X_test, y_test)
```

```
/Users/ir177/opt/anaconda3/lib/python3.8/site-
```

```
packages/sklearn/linear_model/_logistic.py:762: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
```

[177]:
```
0.14222222222222222
```

Wow! We jumped from 82.7% accuracy to 92.4%, although we are still only training the model on 50 instances. Since it's often costly and painful to label instances, especially when it has to be done manually by experts, it's a good idea to make them label representative instances rather than just random instances.

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

[178]:
```
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]
```

[179]:
```
log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_train, y_train_propagated)
```

```
/Users/ir177/opt/anaconda3/lib/python3.8/site-
packages/sklearn/linear_model/_logistic.py:762: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
```

[179]:
```
LogisticRegression(random_state=42)
```

[180]:
```
log_reg.score(X_test, y_test)
```

[180]:
```
0.18
```

We got a tiny little accuracy boost. Better than nothing, but we should probably have propagated the labels only to the instances closest to the centroid, because by propagating to the full cluster,

we have certainly included some outliers. Let's only propagate the labels to the 20th percentile closest to the centroid:

[181]:
```
percentile_closest = 20

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
```

[182]:
```
partially_propagated = (X_cluster_dist != -1)
X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]
```

[183]:
```
log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
```

```
/Users/ir177/opt/anaconda3/lib/python3.8/site-
packages/sklearn/linear_model/_logistic.py:762: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
```

[183]:
```
LogisticRegression(random_state=42)
```

[184]:
```
log_reg.score(X_test, y_test)
```

[184]:
```
0.17333333333333334
```

Nice! With just 50 labeled instances (just 5 examples per class on average!), we got 94.2% performance, which is pretty close to the performance of logistic regression on the fully labeled *digits* dataset (which was 96.7%).

This is because the propagated labels are actually pretty good: their accuracy is very close to 99%:

[185]:
```
np.mean(y_train_partially_propagated == y_train[partially_propagated])
```

[185]:
```
0.20069204152249134
```

You could now do a few iterations of *active learning*: 1. Manually label the instances that the

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

13.3 DBSCAN

[186]:
```
from sklearn.datasets import make_moons
```

[187]:
```
X, y = make_moons(n_samples=1000, noise=0.05, random_state=42)
```

[188]:
```
from sklearn.cluster import DBSCAN
```

[189]:
```
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
```

[189]:
```
DBSCAN(eps=0.05)
```

[190]:
```
dbscan.labels_[:10]
```

[190]:
```
array([ 0,  2, -1, -1,  1,  0,  0,  0,  2,  5])
```

[191]:
```
len(dbscan.core_sample_indices_)
```

[191]:
```
808
```

[192]:
```
dbscan.core_sample_indices_[:10]
```

[192]:
```
array([ 0,  4,  5,  6,  7,  8, 10, 11, 12, 13])
```

[193]:
```
dbscan.components_[:3]
```

[193]:
```
array([[-0.02137124,  0.40618608],
       [-0.84192557,  0.53058695],
       [ 0.58930337, -0.32137599]])
```

[194]:
```
np.unique(dbscan.labels_)
```

[194]:
```
array([-1,  0,  1,  2,  3,  4,  5,  6])
```

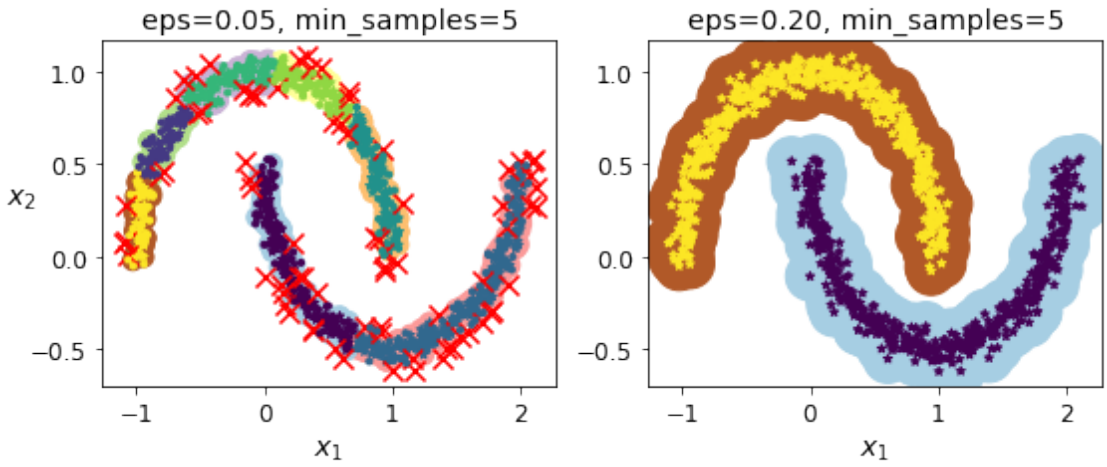
[195]:
```
dbscan2 = DBSCAN(eps=0.2)
dbscan2.fit(X)
```

[195]:
```
DBSCAN(eps=0.2)
```

[196]:
```
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], marker=".")
    if show_xlabels:
        plt.xlabel("$x_1$", fontsize=14)
    else:
        plt.tick_params(labelbottom='off')
    if show_ylabels:
        plt.ylabel("$x_2$", fontsize=14, rotation=0)
    else:
        plt.tick_params(labelleft='off')
    plt.title("eps={:.2f}, min_samples={}".format(dbscan.eps, dbscan.
↪min_samples), fontsize=14)
```

[197]:
```
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)


plt.show()
```



[198]:
```
dbscan = dbscan2
```

[199]:
```
from sklearn.neighbors import KNeighborsClassifier
```

[200]:
```
knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbscan.components_, dbscan.labels_[dbscan.core_sample_indices_])
```

[200]:
```
KNeighborsClassifier(n_neighbors=50)
```

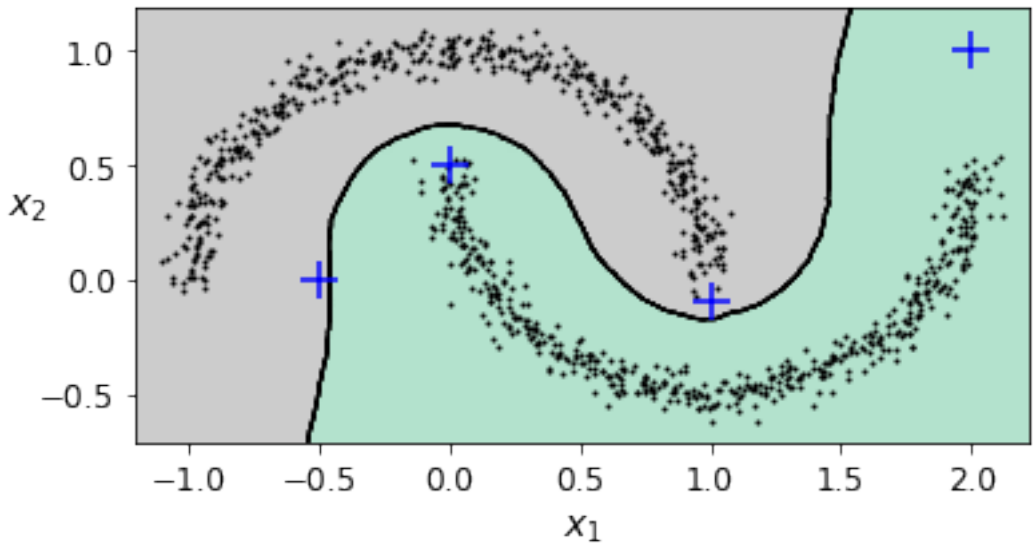
[201]:
```
X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
knn.predict(X_new)
```

[201]:
```
array([1, 0, 1, 0])
```

[202]:
```
knn.predict_proba(X_new)
```

[202]:
```
array([[0.18, 0.82],
       [1.  , 0.  ],
       [0.12, 0.88],
       [1.  , 0.  ]])
```

[203]:
```
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)

plt.show()
```



[204]:
```
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()
```

[204]:
```
array([-1,  0,  1, -1])
```

13.4 Other Clustering Algorithms

13.4.1 Spectral Clustering

[205]:
```
from sklearn.cluster import SpectralClustering
```

[206]:
```
sc1 = SpectralClustering(n_clusters=2, gamma=100, random_state=42)
sc1.fit(X)
```

[206]:
```
SpectralClustering(gamma=100, n_clusters=2, random_state=42)
```

[207]:
```
sc2 = SpectralClustering(n_clusters=2, gamma=1, random_state=42)
sc2.fit(X)
```

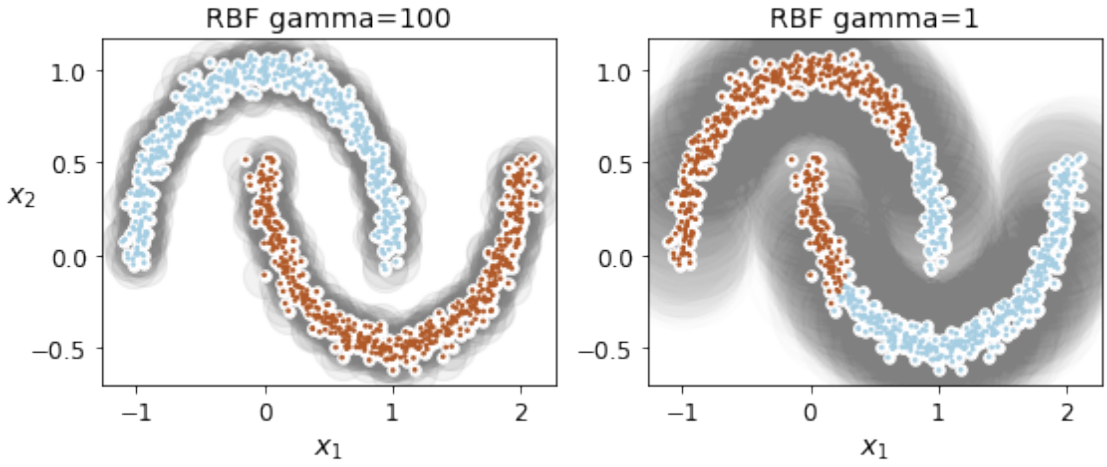
[207]:
```
SpectralClustering(gamma=1, n_clusters=2, random_state=42)
```

[208]:
```
np.percentile(sc1.affinity_matrix_, 95)
```

[208]:
```
0.04251990648936265
```

[209]:
```
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=alpha)
    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='off')
    if show_ylabels:
        plt.ylabel("$x_2$", fontsize=14, rotation=0)
    else:
        plt.tick_params(labelleft='off')
    plt.title("RBF gamma={}".format(sc.gamma), fontsize=14)
```

[210]:
```
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()
```



13.4.2 Agglomerative Clustering

[211]:
```
from sklearn.cluster import AgglomerativeClustering
```

[212]:
```
X = np.array([0, 2, 5, 8.5]).reshape(-1, 1)
agg = AgglomerativeClustering(linkage="complete").fit(X)
```

[213]:
```
learned_parameters(agg)
```

[213]:
```
['children_',
 'labels_',
 'n_clusters_',
 'n_connected_components_',
 'n_features_in_',
 'n_leaves_']
```

[214]:
```
agg.children_
```

[214]:
```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

14 Gaussian Mixtures

A Gaussian mixture model is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. One can think of mixture models as generalizing k-means clustering to incorporate information about the covariance structure of the data as well as the centers of the latent Gaussians.

1. high speed to learn mixture models.
2. As this algorithm maximizes only the likelihood, it will not bias the means towards zero, or bias the cluster sizes to have specific structures that might or might not apply.
3. When one has insufficiently many points per mixture, estimating the covariance matrices becomes difficult, and the algorithm is known to diverge and find solutions with infinite likelihood unless one regularizes the covariances artificially.
4. This algorithm will always use all the components it has access to, needing held-out data or information theoretical criteria to decide how many components to use in the absence of external cues.

[215]:
```
X1, y1 = make_blobs(n_samples=1000, centers=((4, -4), (0, 0)), random_state=42)
X1 = X1.dot(np.array([[0.374, 0.95], [0.732, 0.598]]))
X2, y2 = make_blobs(n_samples=250, centers=1, random_state=42)
X2 = X2 + [6, -8]
X = np.r_[X1, X2]
y = np.r_[y1, y2]
```

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

[216]:
```
from sklearn.mixture import GaussianMixture
```

[217]:
```
gm = GaussianMixture(n_components=3, n_init=10, random_state=42)
gm.fit(X)
```

[217]:
```
GaussianMixture(n_components=3, n_init=10, random_state=42)
```

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

[218]:
```
gm.weights_
```

[218]:
```
array([0.39054348, 0.2093669 , 0.40008962])
```

[219]:
```
gm.means_
```

[219]:
```
array([[ 0.05224874,  0.07631976],
       [ 3.40196611,  1.05838748],
       [-1.40754214,  1.42716873]])
```

[220]:
```
gm.covariances_
```

[220]:
```
array([[[ 0.6890309 ,  0.79717058],
        [ 0.79717058,  1.21367348]],

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

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

Did the algorithm actually converge?

[221]:
```
gm.converged_
```

[221]:
```
True
```

Yes, good. How many iterations did it take?

[222]:
```
gm.n_iter_
```

[222]:
```
4
```

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

[223]:
```
gm.predict(X)
```

[223]:
```
array([0, 0, 2, …, 1, 1, 1])
```

[224]:
```
gm.predict_proba(X)
```

[224]:
```
array([[9.77227791e-01, 2.27715290e-02, 6.79898914e-07],
       [9.83288385e-01, 1.60345103e-02, 6.77104389e-04],
       [7.51824662e-05, 1.90251273e-06, 9.99922915e-01],
       …,
       [4.35053542e-07, 9.99999565e-01, 2.17938894e-26],
       [5.27837047e-16, 1.00000000e+00, 1.50679490e-41],
       [2.32355608e-15, 1.00000000e+00, 8.21915701e-41]])
```

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

[225]:
```
X_new, y_new = gm.sample(6)
X_new
```

[225]:
```
array([[-0.8690223 , -0.32680051],
       [ 0.29945755,  0.2841852 ],
       [ 1.85027284,  2.06556913],
       [ 3.98260019,  1.50041446],
       [ 3.82006355,  0.53143606],
       [-1.04015332,  0.7864941 ]])
```

[226]:
```
y_new
```

[226]:
```
array([0, 0, 1, 1, 1, 2])
```

Notice that they are sampled sequentially from each cluster.

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

[227]:
```
gm.score_samples(X)
```

[227]:
```
array([-2.60674489, -3.57074133, -3.33007348, …, -3.51379355,
       -4.39643283, -3.8055665 ])
```

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

[228]:
```
resolution = 100
grid = np.arange(-10, 10, 1 / resolution)
xx, yy = np.meshgrid(grid, grid)
X_full = np.vstack([xx.ravel(), yy.ravel()]).T
```

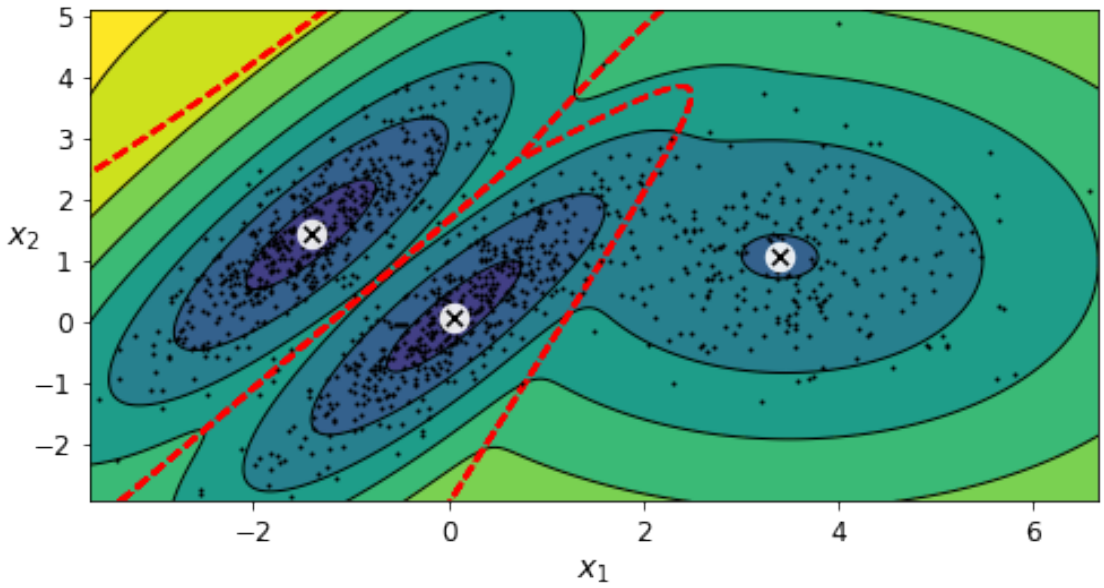
```
pdf = np.exp(gm.score_samples(X_full))
pdf_probas = pdf * (1 / resolution) ** 2
pdf_probas.sum()
```

[228]:
```
0.9999999999271592
```

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

[229]:
```
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='off')
```

[230]:
```
plt.figure(figsize=(8, 4))

plot_gaussian_mixture(gm, X)


plt.show()
```



You can impose constraints on the covariance matrices that the algorithm looks for by setting the `covariance_type` hyperparameter: * `"full"` (default): no constraint, all clusters can take on any ellipsoidal shape of any size. * `"tied"`: all clusters must have the same shape, which can be any ellipsoid (i.e., they all share the same covariance matrix). * `"spherical"`: all clusters must be spherical, but they can have different diameters (i.e., different variances). * `"diag"`: clusters can take on any ellipsoidal shape of any size, but the ellipsoid's axes must be parallel to the axes (i.e., the covariance matrices must be diagonal).

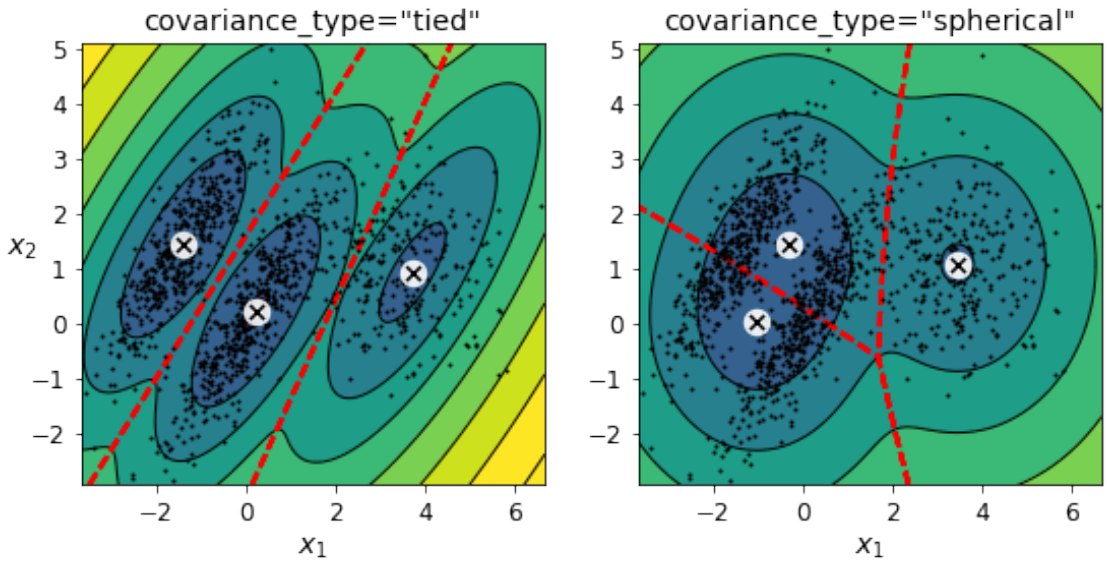
[231]:
```
gm_full = GaussianMixture(n_components=3, n_init=10, covariance_type="full",␣
↪random_state=42)
gm_tied = GaussianMixture(n_components=3, n_init=10, covariance_type="tied",␣
↪random_state=42)
gm_spherical = GaussianMixture(n_components=3, n_init=10,␣
↪covariance_type="spherical", random_state=42)
gm_diag = GaussianMixture(n_components=3, n_init=10, covariance_type="diag",␣
↪random_state=42)
gm_full.fit(X)
gm_tied.fit(X)
gm_spherical.fit(X)
gm_diag.fit(X)
```

[231]:
```
GaussianMixture(covariance_type='diag', n_components=3, n_init=10,
                random_state=42)
```

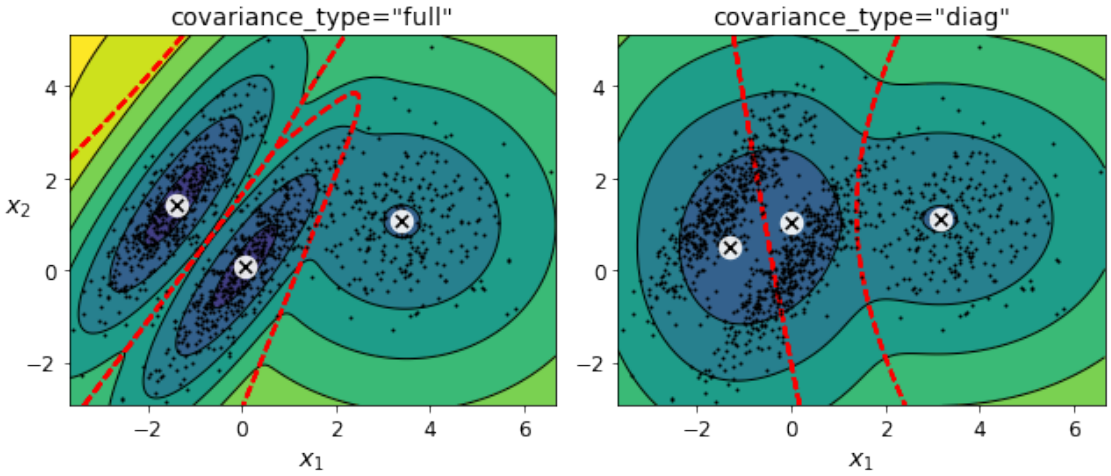
[232]:
```
def compare_gaussian_mixtures(gm1, gm2, X):
    plt.figure(figsize=(9, 4))
```

```
    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)
```

[233]:
```
compare_gaussian_mixtures(gm_tied, gm_spherical, X)


plt.show()
```



[234]:
```
compare_gaussian_mixtures(gm_full, gm_diag, X)
plt.tight_layout()
plt.show()
```

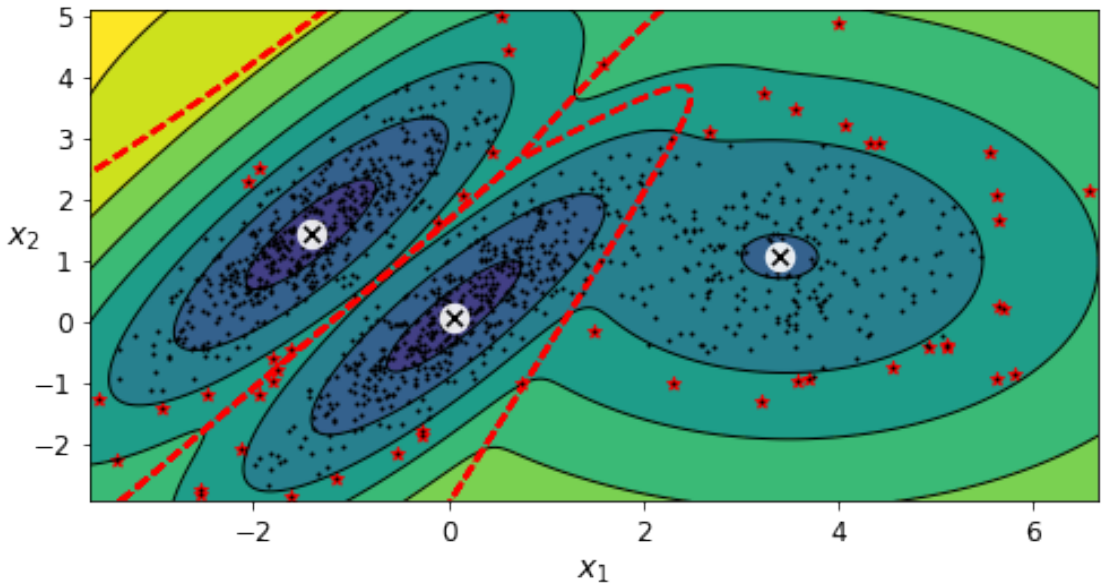


14.1 Anomaly Detection using Gaussian Mixtures

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

[235]:
```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 4)
anomalies = X[densities < density_threshold]
```

[236]:
```
plt.figure(figsize=(8, 4))

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



14.2 Model selection

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

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

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

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

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

[237]:
```
gm.bic(X)
```

[237]:
```
8189.662685850679
```

[238]:
```
gm.aic(X)
```

[238]:
```
8102.437405735641
```

We could compute the BIC manually like this:

[239]:
```
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) # log(L^)
bic = np.log(len(X)) * n_params - 2 * max_log_likelihood
aic = 2 * n_params - 2 * max_log_likelihood
```

[240]:
```
bic, aic
```

[240]:
```
(8189.662685850679, 8102.437405735641)
```

[241]:
```
n_params
```

[241]:
```
17
```

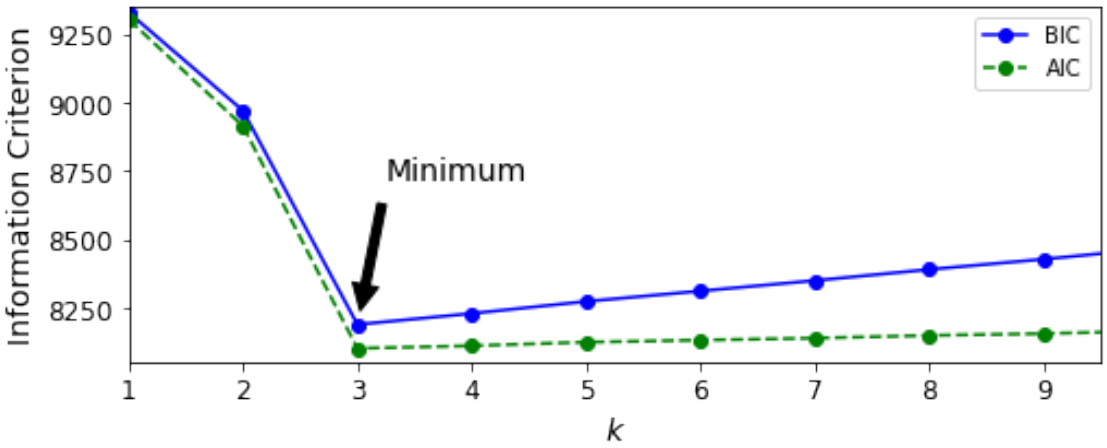
There's one weight per cluster, but the sum must be equal to 1, so we have one degree of freedom less, hence the -1. Similarly, the degrees of freedom for an $n \times n$ covariance matrix is not n^2, but $1 + 2 + \cdots + n = \dfrac{n(n+1)}{2}$.

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

[242]:
```
gms_per_k = [GaussianMixture(n_components=k, n_init=10, random_state=42).fit(X)
             for k in range(1, 11)]
```

[243]:
```
bics = [model.bic(X) for model in gms_per_k]
aics = [model.aic(X) for model in gms_per_k]
```

[244]:
```
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()
plt.show()
```



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

[245]:
```
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
```

[246]:
```
best_k
```

[246]:
```
3
```

[247]:
```
best_covariance_type
```

[247]:
```
'full'
```

14.3 Variational Bayesian Gaussian Mixtures

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

[248]:
```
from sklearn.mixture import BayesianGaussianMixture
```

[249]:
```
bgm = BayesianGaussianMixture(n_components=10, n_init=10, random_state=42)
bgm.fit(X)
```

```
/Users/ir177/opt/anaconda3/lib/python3.8/site-
packages/sklearn/mixture/_base.py:265: ConvergenceWarning: Initialization 10 did
not converge. Try different init parameters, or increase max_iter, tol or check
for degenerate data.
  warnings.warn('Initialization %d did not converge. '
```

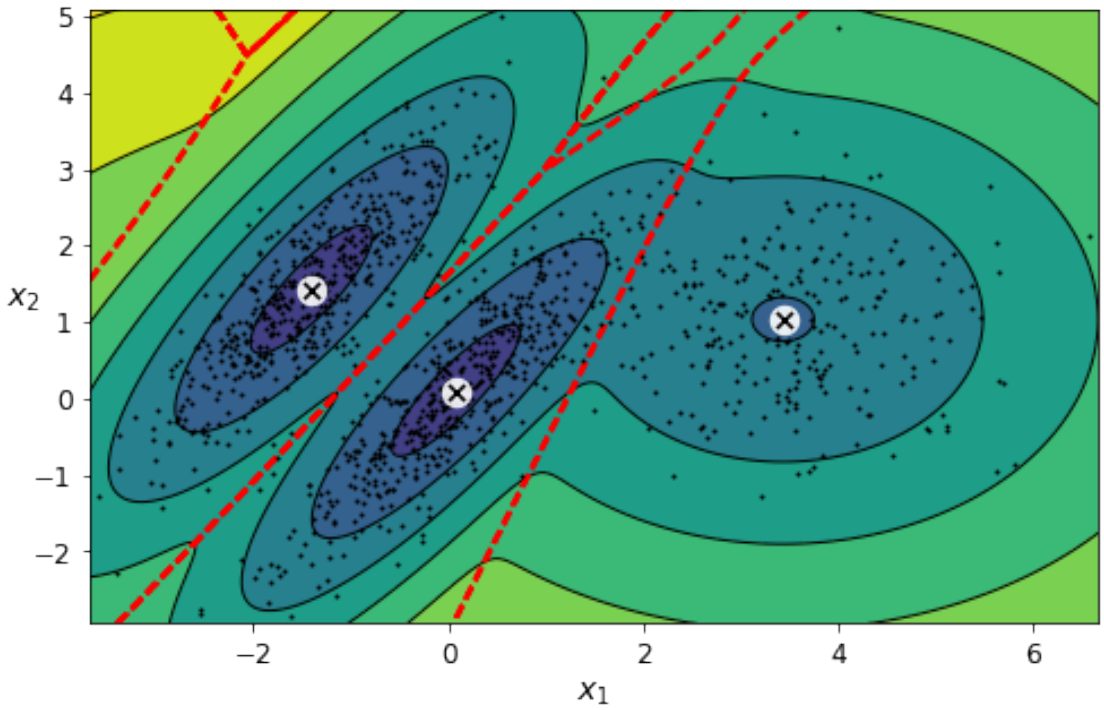
[249]:
```
BayesianGaussianMixture(n_components=10, n_init=10, random_state=42)
```

The algorithm automatically detected that only 3 components are needed:

[250]:
```
np.round(bgm.weights_, 2)
```

[250]:
```
array([0.4 , 0.  , 0.  , 0.  , 0.39, 0.2 , 0.  , 0.  , 0.  , 0.  ])
```

[251]:
```
plt.figure(figsize=(8, 5))
plot_gaussian_mixture(bgm, X)
plt.show()
```



[252]:
```
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])
```

[252]:
```
BayesianGaussianMixture(max_iter=1000, n_components=10, random_state=42,
                        weight_concentration_prior=10000)
```

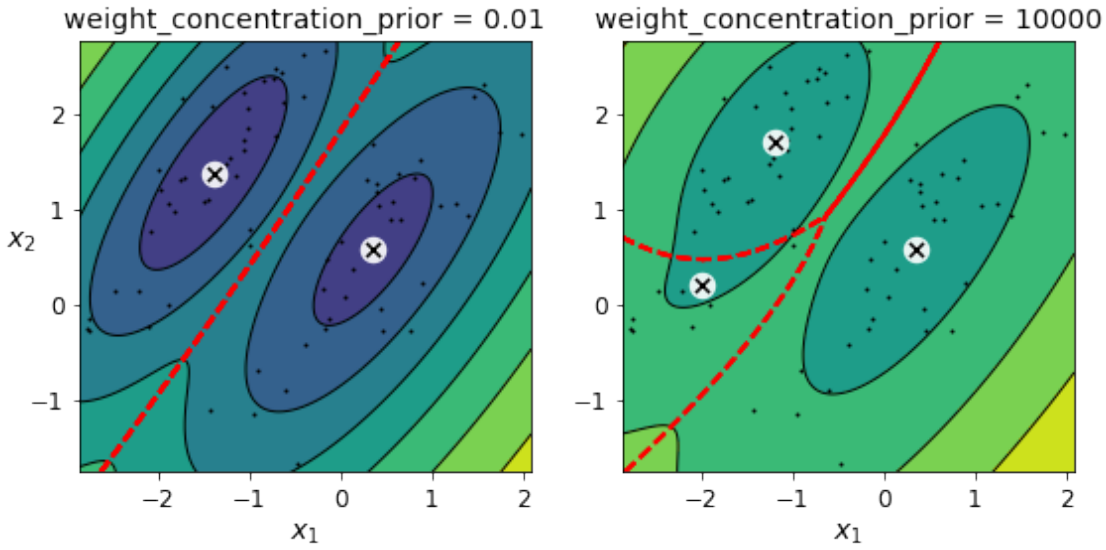
[253]:
```
np.round(bgm_low.weights_, 2)
```

[253]:
```
array([0.49, 0.51, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ])
```

[254]:
```
np.round(bgm_high.weights_, 2)
```

[254]:
```
array([0.43, 0.01, 0.01, 0.11, 0.01, 0.01, 0.01, 0.37, 0.01, 0.01])
```

[255]:
```
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)

plt.show()
```



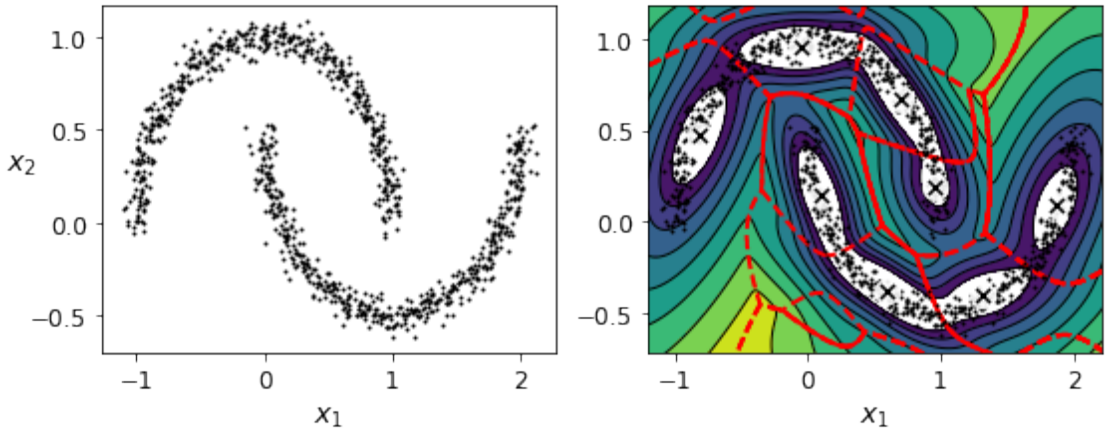
Note: the fact that you see only 3 regions in the right plot although there are 4 centroids is not a bug. The weight of the top-right cluster is much larger than the weight of the lower-right cluster, so the probability that any given point in this region belongs to the top right cluster is greater than the probability that it belongs to the lower-right cluster.

[256]:
```
X_moons, y_moons = make_moons(n_samples=1000, noise=0.05, random_state=42)
```

[257]:
```
bgm = BayesianGaussianMixture(n_components=10, n_init=10, random_state=42)
bgm.fit(X_moons)
```

[257]:
```
BayesianGaussianMixture(n_components=10, n_init=10, random_state=42)
```

[258]:
```
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)

plt.show()
```



Oops, not great… instead of detecting 2 moon-shaped clusters, the algorithm detected 8 ellipsoidal clusters. However, the density plot does not look too bad, so it might be usable for anomaly detection.

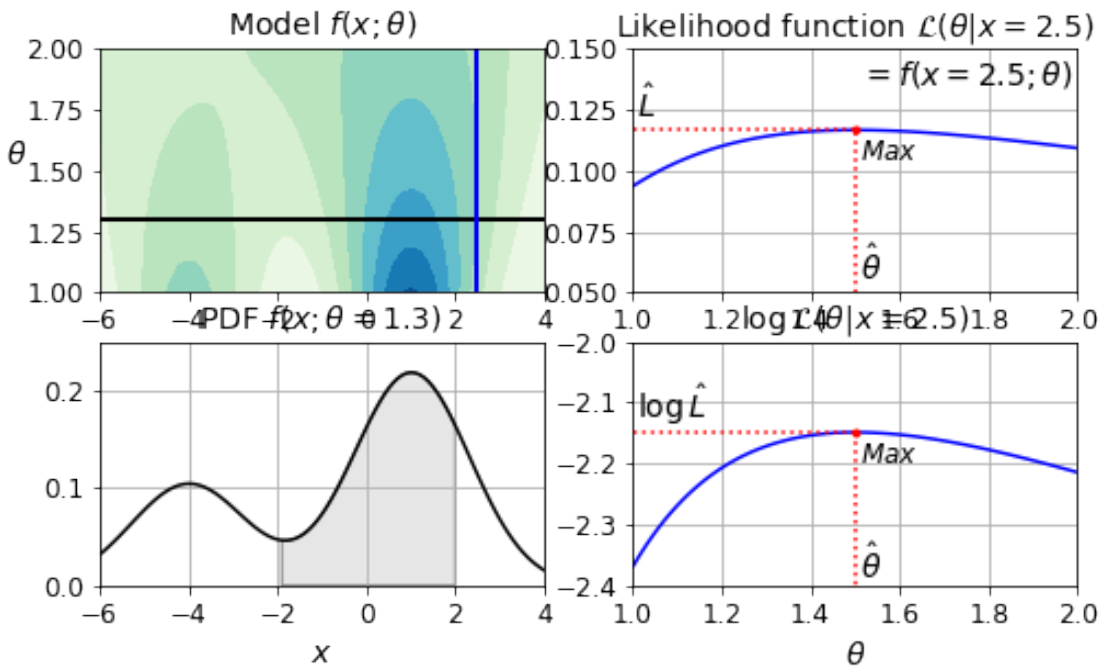
14.4 Likelihood Function

[259]:
```
from scipy.stats import norm
```

[260]:
```
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) / (xx[1] - xx[0])
```

[261]:
```
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)

plt.show()
```


Model $f(x;\theta)$
Likelihood function $\mathcal{L}(\theta|x=2.5)$
$= f(x=2.5;\theta)$
$\hat{L}$
Max
$\hat{\theta}$
PDF $f(x;\theta=1.3)$
$\log \mathcal{L}(\theta|x=2.5)$
$\log \hat{L}$
Max
$\hat{\theta}$
θ
x
θ

[ ]:

[ ]:

[ ]:

[ ]: