In Depth: Principal Component Analysis

Up until now, we have been looking in depth at supervised learning estimators: those estimators that predict labels based on labeled training data. Here we begin looking at several unsupervised estimators, which can highlight interesting aspects of the data without reference to any known labels.

In this chapter we will explore what is perhaps one of the most broadly used unsupervised algorithms, principal component analysis (PCA). PCA is fundamentally a dimensionality reduction algorithm, but it can also be useful as a tool for visualization, noise filtering, feature extraction and engineering, and much more. After a brief conceptual discussion of the PCA algorithm, we will explore a couple examples of these further applications.

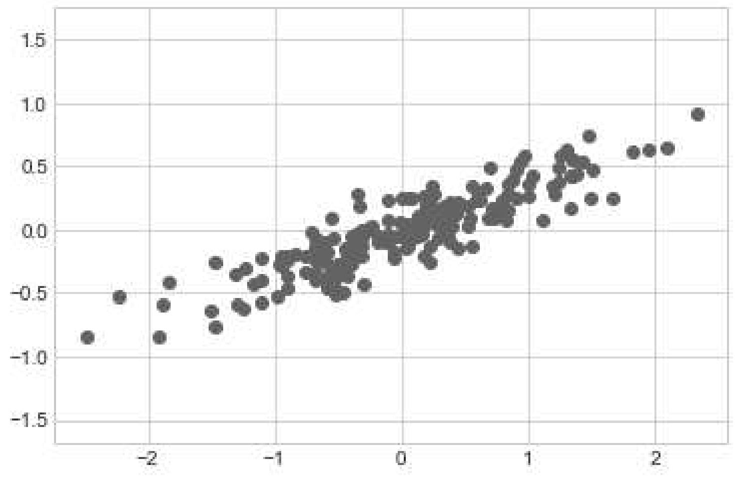
We begin with the standard imports:

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
```

Introducing Principal Component Analysis

Principal component analysis is a fast and flexible unsupervised method for dimensionality reduction in data, which we saw briefly in Introducing Scikit-Learn (05.02-Introducing-Scikit-Learn.ipynb). Its behavior is easiest to visualize by looking at a two-dimensional dataset. Consider these 200 points (see the following figure):

```
In [2]: rng = np.random.RandomState(1)
        X = np.dot(rng.rand(2, 2), rng.randn(2, 200)).T
        plt.scatter(X[:, 0], X[:, 1])
        plt.axis('equal');
```



By eye, it is clear that there is a nearly linear relationship between the *x* and *y* variables. This is reminiscent of the linear regression data we explored in In Depth: Linear Regression (05.06-Linear-Regression.ipynb), but the problem setting here is slightly different: rather than attempting to *predict* the *y* values from the *x* values, the unsupervised learning problem attempts to learn about the *relationship* between the *x* and *y* values.

In principal component analysis, this relationship is quantified by finding a list of the *principal axes* in the data, and using those axes to describe the dataset. Using Scikit-Learn's `PCA` estimator, we can compute this as follows:

```
In [3]: from sklearn.decomposition import PCA
        pca = PCA(n_components=2)
        pca.fit(X)

Out[3]: PCA(n_components=2)
```

The fit learns some quantities from the data, most importantly the components and explained variance:

```
In [4]: print(pca.components_)

        [[-0.94446029 -0.32862557]
         [-0.32862557  0.94446029]]
```

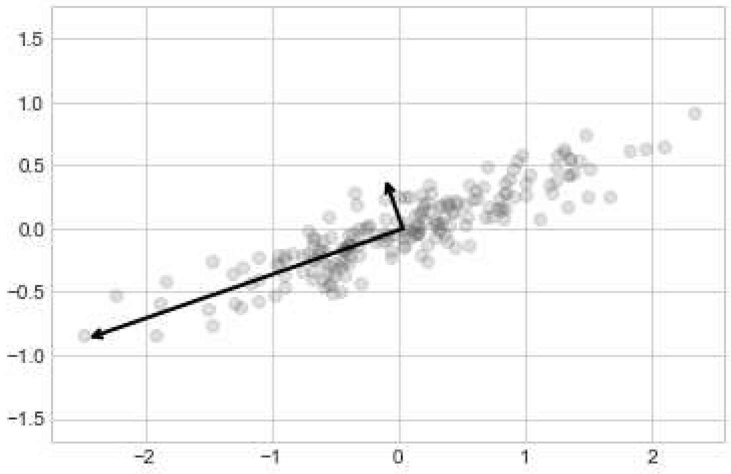
```
In [5]: print(pca.explained_variance_)

        [0.7625315 0.0184779]
```

To see what these numbers mean, let's visualize them as vectors over the input data, using the components to define the direction of the vector and the explained variance to define the squared length of the vector (see the following figure):

```
In [6]: def draw_vector(v0, v1, ax=None):
            ax = ax or plt.gca()
            arrowprops=dict(arrowstyle='->', linewidth=2,
                            shrinkA=0, shrinkB=0)
            ax.annotate('', v1, v0, arrowprops=arrowprops)

        # plot data
        plt.scatter(X[:, 0], X[:, 1], alpha=0.2)
        for length, vector in zip(pca.explained_variance_, pca.components_):
            v = vector * 3 * np.sqrt(length)
            draw_vector(pca.mean_, pca.mean_ + v)
        plt.axis('equal');
```



These vectors represent the principal axes of the data, and the length of each vector is an indication of how "important" that axis is in describing the distribution of the data—more precisely, it is a measure of the variance of the data when projected onto that axis. The projection of each data point onto the principal axes are the principal components of the data.

If we plot these principal components beside the original data, we see the plots shown in the following figure:

figure source in Appendix (https://github.com/jakevdp/PythonDataScienceHandbook/blob/master/notebooks/06.00-Figure-Code.ipynb#Principal-Components-Rotation)

This transformation from data axes to principal axes is an *affine transformation*, which means it is composed of a translation, rotation, and uniform scaling.

While this algorithm to find principal components may seem like just a mathematical curiosity, it turns out to have very far-reaching applications in the world of machine learning and data exploration.

PCA as Dimensionality Reduction

Using PCA for dimensionality reduction involves zeroing out one or more of the smallest principal components, resulting in a lower-dimensional projection of the data that preserves the maximal data variance.

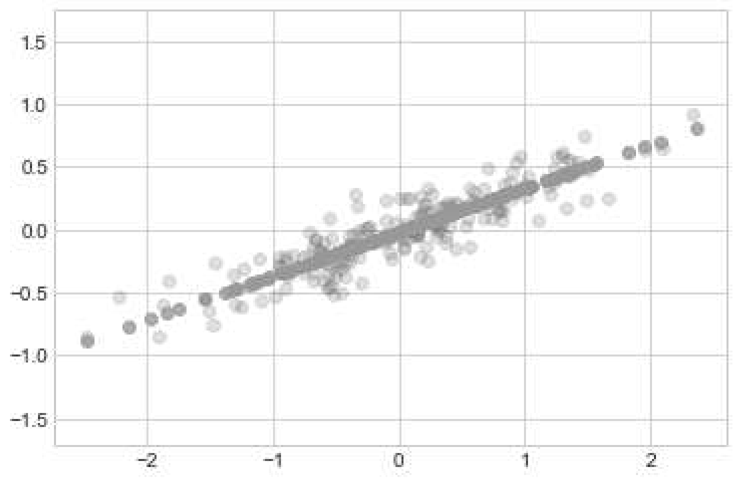
Here is an example of using PCA as a dimensionality reduction transform:

In [7]:
```
pca = PCA(n_components=1)
pca.fit(X)
X_pca = pca.transform(X)
print("original shape:   ", X.shape)
print("transformed shape:", X_pca.shape)
```

```
original shape:    (200, 2)
transformed shape: (200, 1)
```

The transformed data has been reduced to a single dimension. To understand the effect of this dimensionality reduction, we can perform the inverse transform of this reduced data and plot it along with the original data (see the following figure):

In [8]:
```
X_new = pca.inverse_transform(X_pca)
plt.scatter(X[:, 0], X[:, 1], alpha=0.2)
plt.scatter(X_new[:, 0], X_new[:, 1], alpha=0.8)
plt.axis('equal');
```



The light points are the original data, while the dark points are the projected version. This makes clear what a PCA dimensionality reduction means: the information along the least important principal axis or axes is removed, leaving only the component(s) of the data with the highest variance. The fraction of variance that is cut out (proportional to the spread of points about the line formed in the preceding figure) is roughly a measure of how much "information" is discarded in this reduction of dimensionality.

PCA for Visualization: Handwritten Digits

The usefulness of dimensionality reduction may not be entirely apparent in only two dimensions, but it becomes clear when looking at high-dimensional data. To see this, let's take a quick look at the application of PCA to the digits dataset we worked with in In-Depth: Decision Trees and Random Forests (05.08-Random-Forests.ipynb).

We'll start by loading the data:

In [9]:
```
from sklearn.datasets import load_digits
digits = load_digits()
digits.data.shape
```

Out[9]:
```
(1797, 64)
```

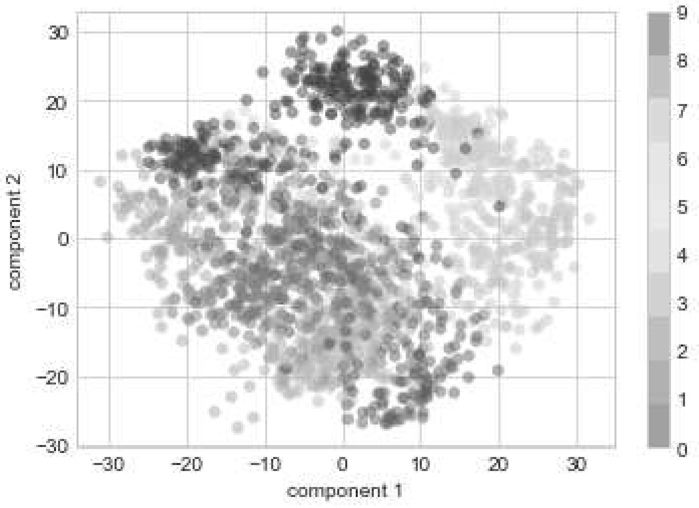
Recall that the digits dataset consists of 8 × 8–pixel images, meaning that they are 64-dimensional. To gain some intuition into the relationships between these points, we can use PCA to project them into a more manageable number of dimensions, say two:

In [10]:
```
pca = PCA(2)  # project from 64 to 2 dimensions
projected = pca.fit_transform(digits.data)
print(digits.data.shape)
print(projected.shape)
```

```
(1797, 64)
(1797, 2)
```

We can now plot the first two principal components of each point to learn about the data, as seen in the following figure:

```
In [11]: plt.scatter(projected[:, 0], projected[:, 1],
                     c=digits.target, edgecolor='none', alpha=0.5,
                     cmap=plt.cm.get_cmap('rainbow', 10))
         plt.xlabel('component 1')
         plt.ylabel('component 2')
         plt.colorbar();
```



Recall what these components mean: the full data is a 64-dimensional point cloud, and these points are the projection of each data point along the directions with the largest variance. Essentially, we have found the optimal stretch and rotation in 64-dimensional space that allows us to see the layout of the data in two dimensions, and we have done this in an unsupervised manner—that is, without reference to the labels.

What Do the Components Mean?

We can go a bit further here, and begin to ask what the reduced dimensions *mean*. This meaning can be understood in terms of combinations of basis vectors. For example, each image in the training set is defined by a collection of 64 pixel values, which we will call the vector x:

$$x = [x_1, x_2, x_3 \cdots x_{64}]$$

One way we can think about this is in terms of a pixel basis. That is, to construct the image, we multiply each element of the vector by the pixel it describes, and then add the results together to build the image:

$$\text{image}(x) = x_1 \cdot (\text{pixel 1}) + x_2 \cdot (\text{pixel 2}) + x_3 \cdot (\text{pixel 3}) \cdots x_{64} \cdot (\text{pixel 64})$$

One way we might imagine reducing the dimensionality of this data is to zero out all but a few of these basis vectors. For example, if we use only the first eight pixels, we get an eight-dimensional projection of the data (the following figure). However, it is not very reflective of the whole image: we've thrown out nearly 90% of the pixels!

figure source in Appendix

The upper row of panels shows the individual pixels, and the lower row shows the cumulative contribution of these pixels to the construction of the image. Using only eight of the pixel-basis components, we can only construct a small portion of the 64-pixel image. Were we to continue this sequence and use all 64 pixels, we would recover the original image.

But the pixel-wise representation is not the only choice of basis. We can also use other basis functions, which each contain some predefined contribution from each pixel, and write something like:

$$image(x) = \text{mean} + x_1 \cdot (\text{basis 1}) + x_2 \cdot (\text{basis 2}) + x_3 \cdot (\text{basis 3}) \cdots$$

PCA can be thought of as a process of choosing optimal basis functions, such that adding together just the first few of them is enough to suitably reconstruct the bulk of the elements in the dataset. The principal components, which act as the low-dimensional representation of our data, are simply the coefficients that multiply each of the elements in this series. the following figure shows a similar depiction of reconstructing the same digit using the mean plus the first eight PCA basis functions.

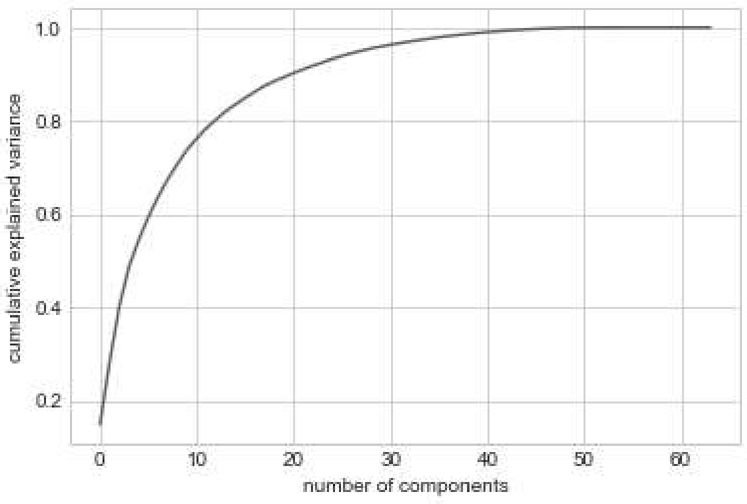
figure source in Appendix (https://github.com/jakevdp/PythonDataScienceHandbook/blob/master/notebooks/06.00-Figure-Code.ipynb#Digits-PCA-Components)

Unlike the pixel basis, the PCA basis allows us to recover the salient features of the input image with just a mean, plus eight components! The amount of each pixel in each component is the corollary of the orientation of the vector in our two-dimensional example. This is the sense in which PCA provides a low-dimensional representation of the data: it discovers a set of basis functions that are more efficient than the native pixel basis of the input data.

Choosing the Number of Components

A vital part of using PCA in practice is the ability to estimate how many components are needed to describe the data. This can be determined by looking at the cumulative *explained variance ratio* as a function of the number of components (see the following figure):

In [12]:
```
pca = PCA().fit(digits.data)
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
```



This curve quantifies how much of the total, 64-dimensional variance is contained within the first N components. For example, we see that with the digits data the first 10 components contain approximately 75% of the variance, while you need around 50 components to describe close to 100% of the variance.

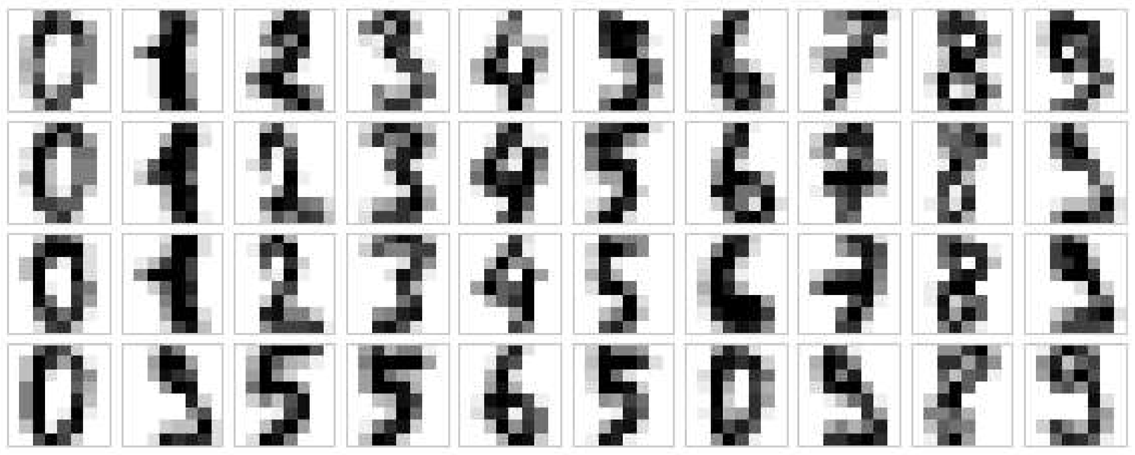
This tells us that our 2-dimensional projection loses a lot of information (as measured by the explained variance) and that we'd need about 20 components to retain 90% of the variance. Looking at this plot for a high-dimensional dataset can help you understand the level of redundancy present in its features.

PCA as Noise Filtering

PCA can also be used as a filtering approach for noisy data. The idea is this: any components with variance much larger than the effect of the noise should be relatively unaffected by the noise. So, if you reconstruct the data using just the largest subset of principal components, you should be preferentially keeping the signal and throwing out the noise.

Let's see how this looks with the digits data. First we will plot several of the input noise-free input samples (the following figure):

In [13]:
```
def plot_digits(data):
    fig, axes = plt.subplots(4, 10, figsize=(10, 4),
                             subplot_kw={'xticks':[], 'yticks':[]},
                             gridspec_kw=dict(hspace=0.1, wspace=0.1))
    for i, ax in enumerate(axes.flat):
        ax.imshow(data[i].reshape(8, 8),
                  cmap='binary', interpolation='nearest',
                  clim=(0, 16))
plot_digits(digits.data)
```

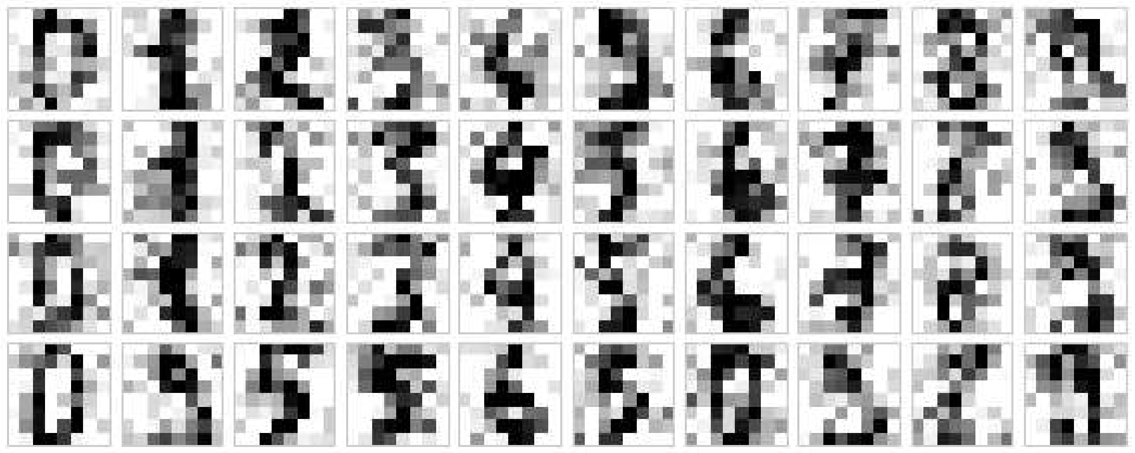


Now let's add some random noise to create a noisy dataset, and replot it (the following figure):

In [14]:
```
rng = np.random.default_rng(42)
rng.normal(10, 2)
```

Out[14]:
```
10.609434159508863
```

In [15]:
```
rng = np.random.default_rng(42)
noisy = rng.normal(digits.data, 4)
plot_digits(noisy)
```



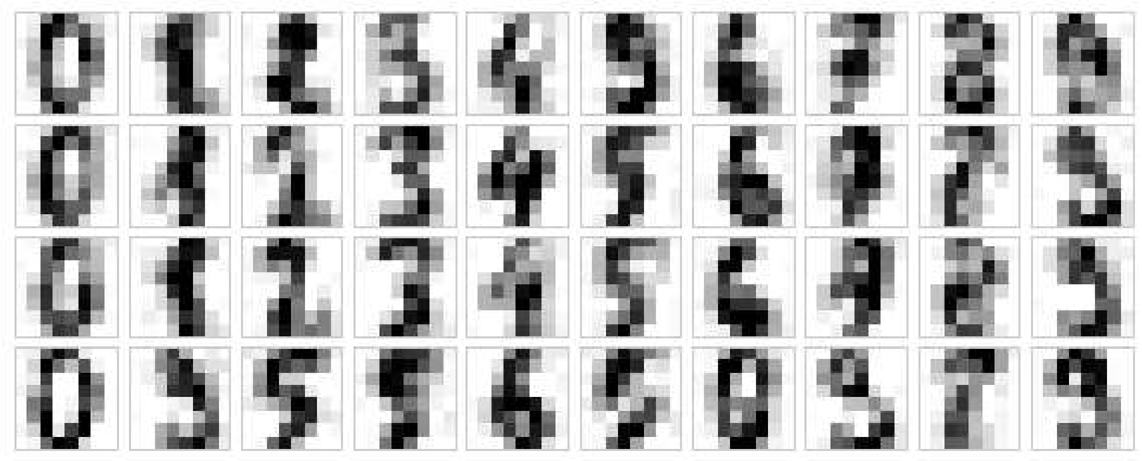
The visualization makes the presence of this random noise clear. Let's train a PCA model on the noisy data, requesting that the projection preserve 50% of the variance:

```
In [16]: pca = PCA(0.50).fit(noisy)
         pca.n_components_
```

```
Out[16]: 12
```

Here 50% of the variance amounts to 12 principal components, out of the 64 original features. Now we compute these components, and then use the inverse of the transform to reconstruct the filtered digits; the following figure shows the result:

```
In [17]: components = pca.transform(noisy)
         filtered = pca.inverse_transform(components)
         plot_digits(filtered)
```



This signal preserving/noise filtering property makes PCA a very useful feature selection routine —for example, rather than training a classifier on very high-dimensional data, you might instead train the classifier on the lower-dimensional principal component representation, which will automatically serve to filter out random noise in the inputs.

Example: Eigenfaces

Earlier we explored an example of using a PCA projection as a feature selector for facial recognition with a support vector machine (see In-Depth: Support Vector Machines (05.07-Support-Vector-Machines.ipynb)). Here we will take a look back and explore a bit more of what went into that. Recall that we were using the Labeled Faces in the Wild (LFW) dataset made available through Scikit-Learn:

```
In [18]: from sklearn.datasets import fetch_lfw_people
         faces = fetch_lfw_people(min_faces_per_person=60)
         print(faces.target_names)
         print(faces.images.shape)
```

```
['Ariel Sharon' 'Colin Powell' 'Donald Rumsfeld' 'George W Bush'
 'Gerhard Schroeder' 'Hugo Chavez' 'Junichiro Koizumi' 'Tony Blair']
(1348, 62, 47)
```

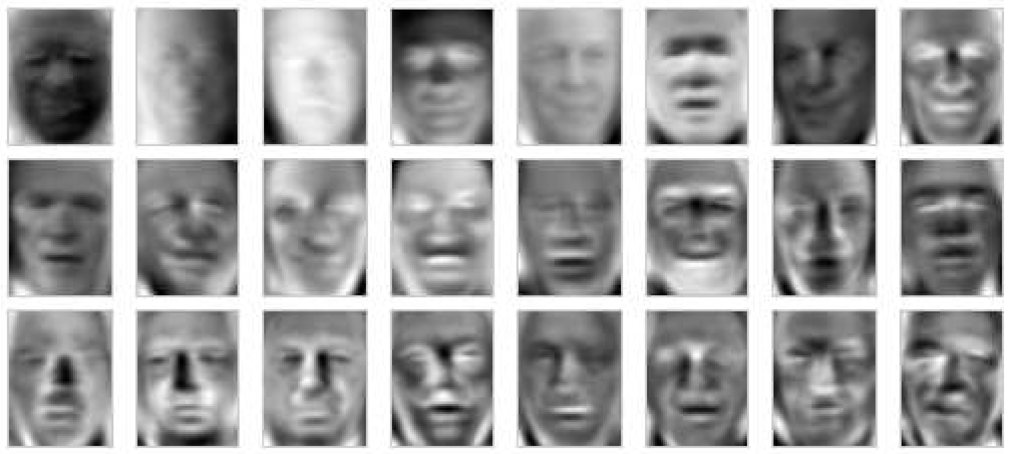
Let's take a look at the principal axes that span this dataset. Because this is a large dataset, we will use the `"random"` eigensolver in the `PCA` estimator: it uses a randomized method to approximate the first N principal components more quickly than the standard approach, at the expense of some accuracy. This trade-off can be useful for high-dimensional data (here, a dimensionality of nearly 3,000). We will take a look at the first 150 components:

In [19]:
```
pca = PCA(150, svd_solver='randomized', random_state=42)
pca.fit(faces.data)
```

Out[19]:
```
PCA(n_components=150, random_state=42, svd_solver='randomized')
```

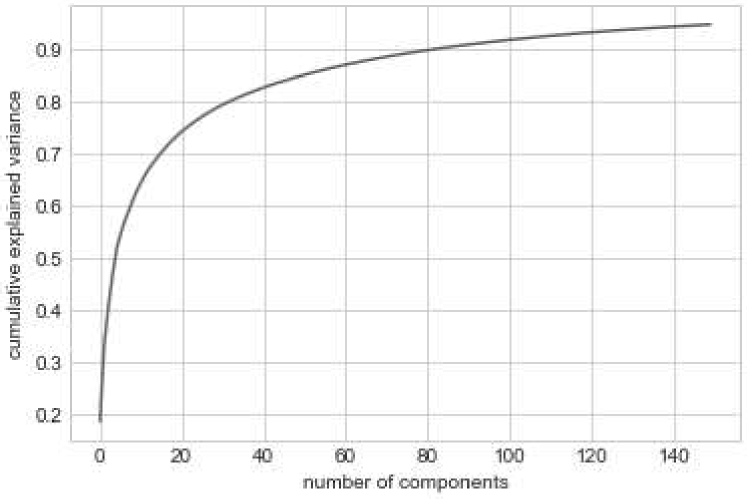
In this case, it can be interesting to visualize the images associated with the first several principal components (these components are technically known as *eigenvectors*, so these types of images are often called *eigenfaces*; as you can see in the following figure, they are as creepy as they sound):

In [20]:
```
fig, axes = plt.subplots(3, 8, figsize=(9, 4),
                         subplot_kw={'xticks':[], 'yticks':[]},
                         gridspec_kw=dict(hspace=0.1, wspace=0.1))
for i, ax in enumerate(axes.flat):
    ax.imshow(pca.components_[i].reshape(62, 47), cmap='bone')
```



The results are very interesting, and give us insight into how the images vary: for example, the first few eigenfaces (from the top left) seem to be associated with the angle of lighting on the face, and later principal vectors seem to be picking out certain features, such as eyes, noses, and lips. Let's take a look at the cumulative variance of these components to see how much of the data information the projection is preserving (see the following figure):

In [21]:
```python
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
```

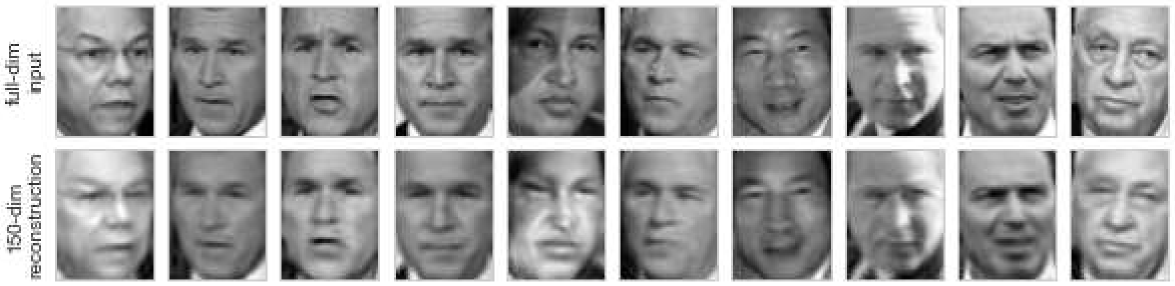


The 150 components we have chosen account for just over 90% of the variance. That would lead us to believe that using these 150 components, we would recover most of the essential characteristics of the data. To make this more concrete, we can compare the input images with the images reconstructed from these 150 components (see the following figure):

In [22]:
```python
# Compute the components and projected faces
pca = pca.fit(faces.data)
components = pca.transform(faces.data)
projected = pca.inverse_transform(components)
```

In [23]:
```python
# Plot the results
fig, ax = plt.subplots(2, 10, figsize=(10, 2.5),
                       subplot_kw={'xticks':[], 'yticks':[]},
                       gridspec_kw=dict(hspace=0.1, wspace=0.1))
for i in range(10):
    ax[0, i].imshow(faces.data[i].reshape(62, 47), cmap='binary_r')
    ax[1, i].imshow(projected[i].reshape(62, 47), cmap='binary_r')

ax[0, 0].set_ylabel('full-dim\ninput')
ax[1, 0].set_ylabel('150-dim\nreconstruction');
```



The top row here shows the input images, while the bottom row shows the reconstruction of the images from just 150 of the ~3,000 initial features. This visualization makes clear why the PCA feature selection used in In-Depth: Support Vector Machines (05.07-Support-Vector-Machines.ipynb) was so successful: although it reduces the dimensionality of the data by nearly a factor of 20, the projected images contain enough information that we might, by eye, recognize the individuals in each image. This means our classification algorithm only needs to be trained on 150-dimensional data rather than 3,000-dimensional data, which, depending on the particular algorithm we choose, can lead to much more efficient classification.

Summary

In this chapter we explored the use of principal component analysis for dimensionality reduction, visualization of high-dimensional data, noise filtering, and feature selection within high-dimensional data. Because of its versatility and interpretability, PCA has been shown to be effective in a wide variety of contexts and disciplines. Given any high-dimensional dataset, I tend to start with PCA in order to visualize the relationships between points (as we did with the digits data), to understand the main variance in the data (as we did with the eigenfaces), and to understand the intrinsic dimensionality (by plotting the explained variance ratio). Certainly PCA is not useful for every high-dimensional dataset, but it offers a straightforward and efficient path to gaining insight into high-dimensional data.

PCA's main weakness is that it tends to be highly affected by outliers in the data. For this reason, several robust variants of PCA have been developed, many of which act to iteratively discard data points that are poorly described by the initial components. Scikit-Learn includes a number of interesting variants on PCA in the `sklearn.decomposition` submodule; one example is `SparsePCA`, which introduces a regularization term (see In Depth: Linear Regression (05.06-Linear-Regression.ipynb)) that serves to enforce sparsity of the components.

In the following chapters, we will look at other unsupervised learning methods that build on some of the ideas of PCA.