- In manifold learning the computational expense of manifold methods scales as $O[N^2]$ or $O[N^3]$. For PCA, there exist randomized approaches that are generally much faster (though see the megaman package for some more scalable implementations of manifold learning).

With all that on the table, the only clear advantage of manifold learning methods over PCA is their ability to preserve nonlinear relationships in the data; for that reason I tend to explore data with manifold methods only after first exploring them with PCA.

Scikit-Learn implements several common variants of manifold learning beyond Isomap and LLE: the Scikit-Learn documentation has a nice discussion and comparison of them. Based on my own experience, I would give the following recommendations:

- For toy problems such as the S-curve we saw before, locally linear embedding (LLE) and its variants (especially *modified LLE*), perform very well. This is implemented in `sklearn.manifold.LocallyLinearEmbedding`.
- For high-dimensional data from real-world sources, LLE often produces poor results, and isometric mapping (Isomap) seems to generally lead to more meaningful embeddings. This is implemented in `sklearn.manifold.Isomap`.
- For data that is highly clustered, *t-distributed stochastic neighbor embedding* (t-SNE) seems to work very well, though can be very slow compared to other methods. This is implemented in `sklearn.manifold.TSNE`.

If you're interested in getting a feel for how these work, I'd suggest running each of the methods on the data in this section.

Example: Isomap on Faces

One place manifold learning is often used is in understanding the relationship between high-dimensional data points. A common case of high-dimensional data is images; for example, a set of images with 1,000 pixels each can be thought of as collection of points in 1,000 dimensions—the brightness of each pixel in each image defines the coordinate in that dimension.

Here let's apply Isomap on some faces data. We will use the Labeled Faces in the Wild dataset, which we previously saw in "In-Depth: Support Vector Machines" on page 405 and "In Depth: Principal Component Analysis" on page 433. Running this command will download the data and cache it in your home directory for later use:

```
In[16]: from sklearn.datasets import fetch_lfw_people
        faces = fetch_lfw_people(min_faces_per_person=30)
        faces.data.shape

Out[16]: (2370, 2914)
```

We have 2,370 images, each with 2,914 pixels. In other words, the images can be thought of as data points in a 2,914-dimensional space!

Let's quickly visualize several of these images to see what we're working with (Figure 5-104):

```
In[17]: fig, ax = plt.subplots(4, 8, subplot_kw=dict(xticks=[], yticks=[]))
        for i, axi in enumerate(ax.flat):
            axi.imshow(faces.images[i], cmap='gray')
```

Figure 5-104. Examples of the input faces

We would like to plot a low-dimensional embedding of the 2,914-dimensional data to learn the fundamental relationships between the images. One useful way to start is to compute a PCA, and examine the explained variance ratio, which will give us an idea of how many linear features are required to describe the data (Figure 5-105):

```
In[18]: from sklearn.decomposition import RandomizedPCA
        model = RandomizedPCA(100).fit(faces.data)
        plt.plot(np.cumsum(model.explained_variance_ratio_))
        plt.xlabel('n components')
        plt.ylabel('cumulative variance');
```

We see that for this data, nearly 100 components are required to preserve 90% of the variance. This tells us that the data is intrinsically very high dimensional—it can't be described linearly with just a few components.

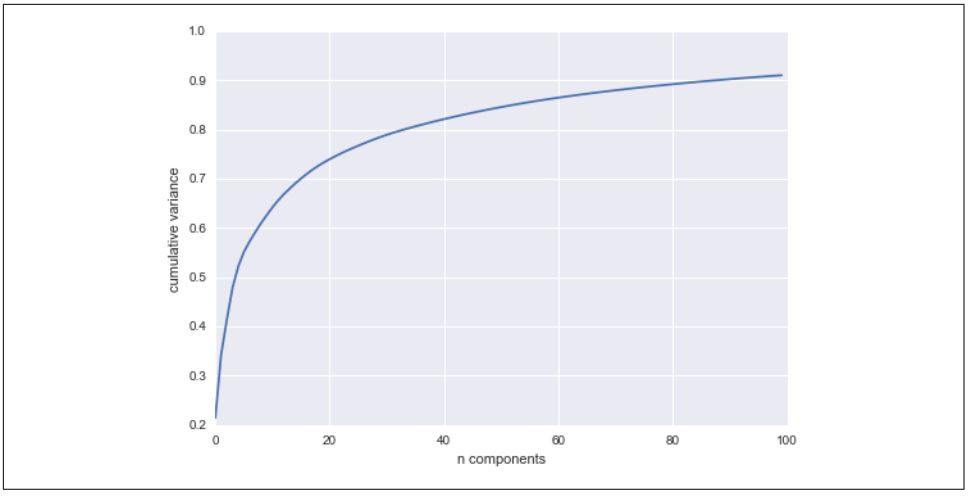

Figure 5-105. Cumulative variance from the PCA projection

When this is the case, nonlinear manifold embeddings like LLE and Isomap can be helpful. We can compute an Isomap embedding on these faces using the same pattern shown before:

```
In[19]: from sklearn.manifold import Isomap
        model = Isomap(n_components=2)
        proj = model.fit_transform(faces.data)
        proj.shape
```

```
Out[19]: (2370, 2)
```

The output is a two-dimensional projection of all the input images. To get a better idea of what the projection tells us, let's define a function that will output image thumbnails at the locations of the projections:

```
In[20]: from matplotlib import offsetbox

        def plot_components(data, model, images=None, ax=None,
                            thumb_frac=0.05, cmap='gray'):
            ax = ax or plt.gca()

            proj = model.fit_transform(data)
            ax.plot(proj[:, 0], proj[:, 1], '.k')

            if images is not None:
                min_dist_2 = (thumb_frac * max(proj.max(0) - proj.min(0))) ** 2
                shown_images = np.array([2 * proj.max(0)])
                for i in range(data.shape[0]):
                    dist = np.sum((proj[i] - shown_images) ** 2, 1)
                    if np.min(dist) < min_dist_2:
                        # don't show points that are too close
                        continue
```

```
                shown_images = np.vstack([shown_images, proj[i]])
                imagebox = offsetbox.AnnotationBbox(
                    offsetbox.OffsetImage(images[i], cmap=cmap),
                                          proj[i])
                ax.add_artist(imagebox)
```

Calling this function now, we see the result (Figure 5-106):

```
In[21]: fig, ax = plt.subplots(figsize=(10, 10))
        plot_components(faces.data,
                        model=Isomap(n_components=2),
                        images=faces.images[:, ::2, ::2])
```

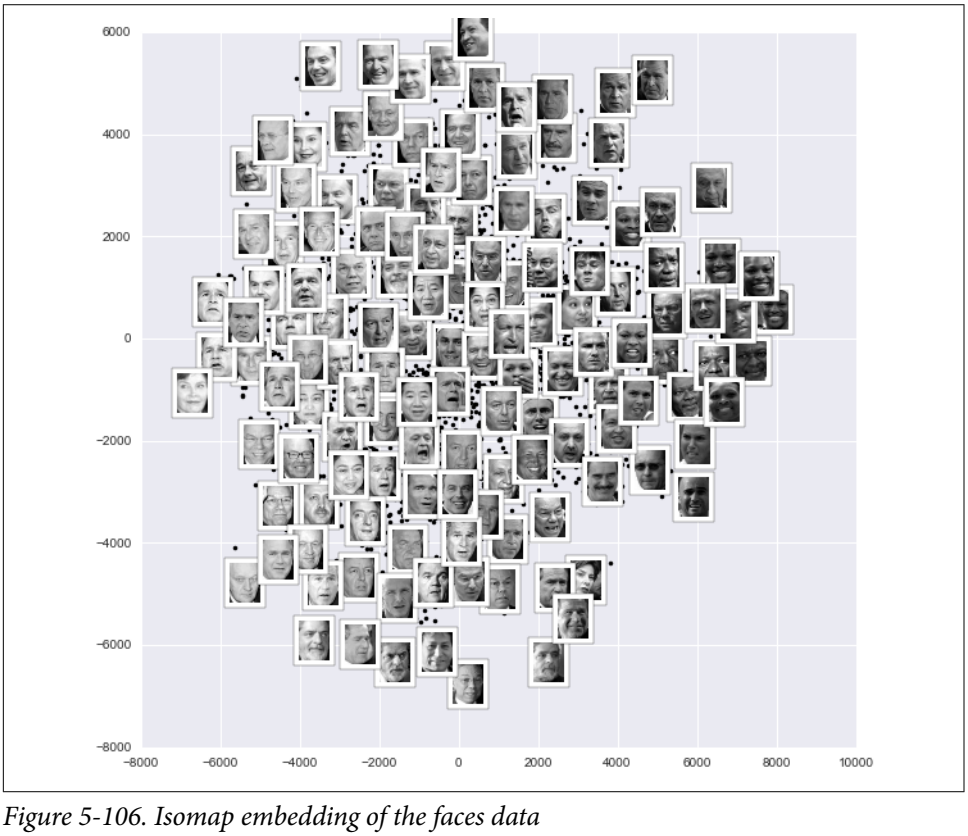

Figure 5-106. Isomap embedding of the faces data

The result is interesting: the first two Isomap dimensions seem to describe global image features: the overall darkness or lightness of the image from left to right, and the general orientation of the face from bottom to top. This gives us a nice visual indication of some of the fundamental features in our data.

We could then go on to classify this data, perhaps using manifold features as inputs to the classification algorithm as we did in "In-Depth: Support Vector Machines" on page 405.

Example: Visualizing Structure in Digits

As another example of using manifold learning for visualization, let's take a look at the MNIST handwritten digits set. This data is similar to the digits we saw in "In-Depth: Decision Trees and Random Forests" on page 421, but with many more pixels per image. It can be downloaded from *http://mldata.org/* with the Scikit-Learn utility:

```
In[22]: from sklearn.datasets import fetch_mldata
        mnist = fetch_mldata('MNIST original')
        mnist.data.shape

Out[22]: (70000, 784)
```

This consists of 70,000 images, each with 784 pixels (i.e., the images are 28×28). As before, we can take a look at the first few images (Figure 5-107):

```
In[23]: fig, ax = plt.subplots(6, 8, subplot_kw=dict(xticks=[], yticks=[]))
        for i, axi in enumerate(ax.flat):
            axi.imshow(mnist.data[1250 * i].reshape(28, 28), cmap='gray_r')
```

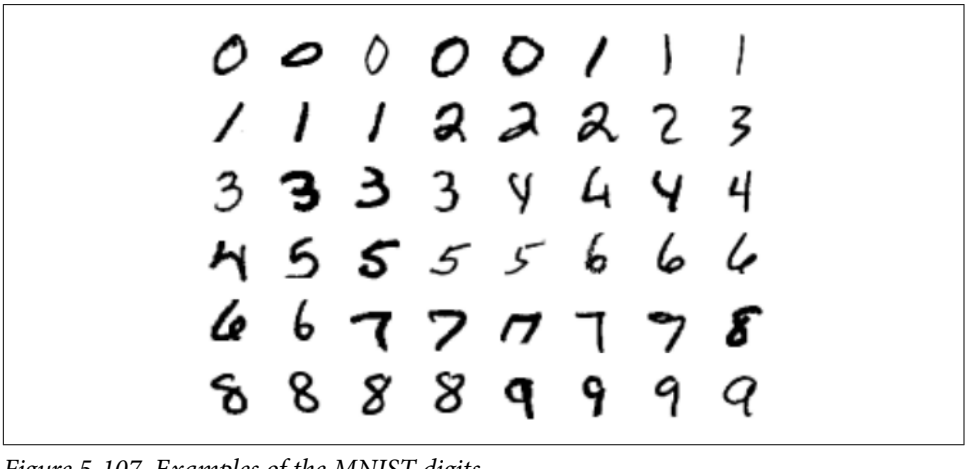

Figure 5-107. Examples of the MNIST digits

This gives us an idea of the variety of handwriting styles in the dataset.

Let's compute a manifold learning projection across the data, illustrated in Figure 5-108. For speed here, we'll only use 1/30 of the data, which is about ~2,000 points (because of the relatively poor scaling of manifold learning, I find that a few thousand samples is a good number to start with for relatively quick exploration before moving to a full calculation):