CHAPTER 45

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 Chapter 38. Its behavior is easiest to visualize by looking at a two-dimensional dataset. Consider these 200 points (see Figure 45-1).

```
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');
```

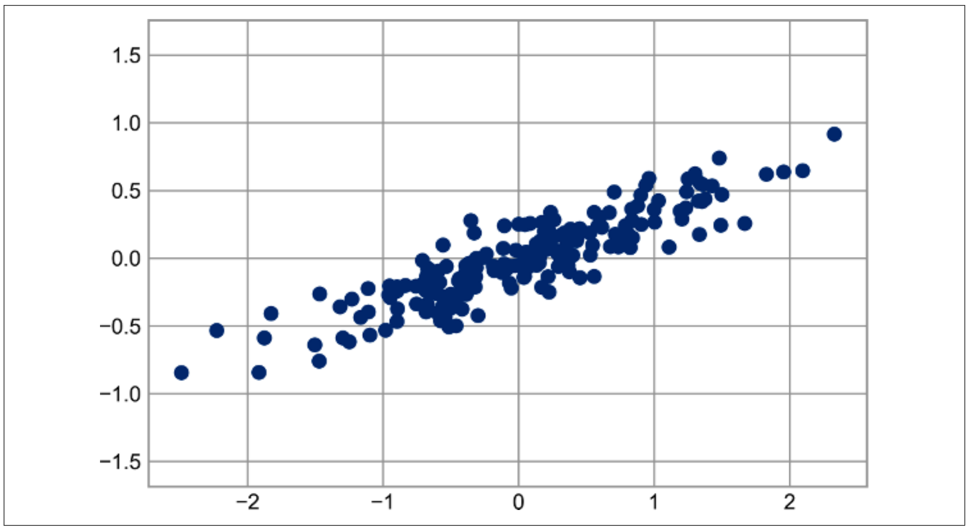

Figure 45-1. Data for demonstration of PCA

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 Chapter 42, 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_)
Out[4]: [[-0.94446029 -0.32862557]
         [-0.32862557  0.94446029]]
```

```
In [5]: print(pca.explained_variance_)
Out[5]: [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 Figure 45-2).

```
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');
```

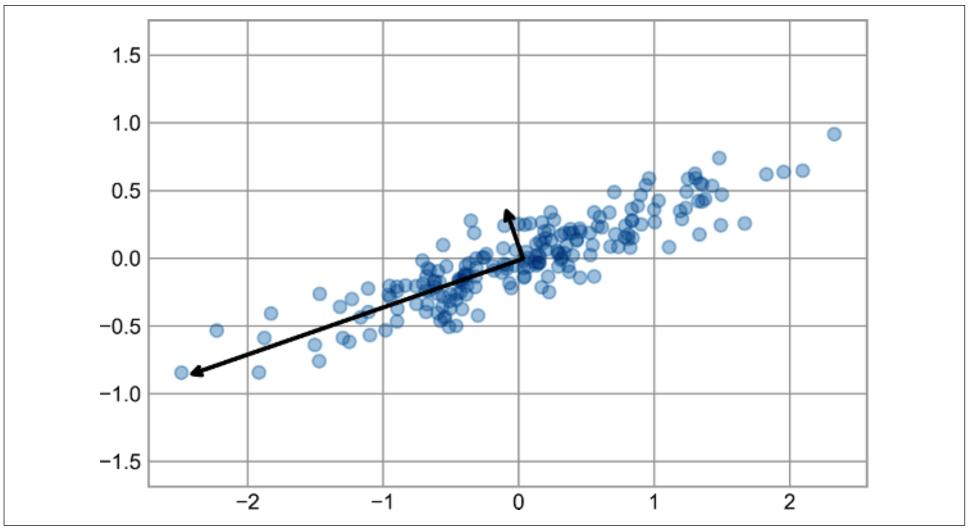

Figure 45-2. Visualization of the principal axes in the data

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 Figure 45-3.

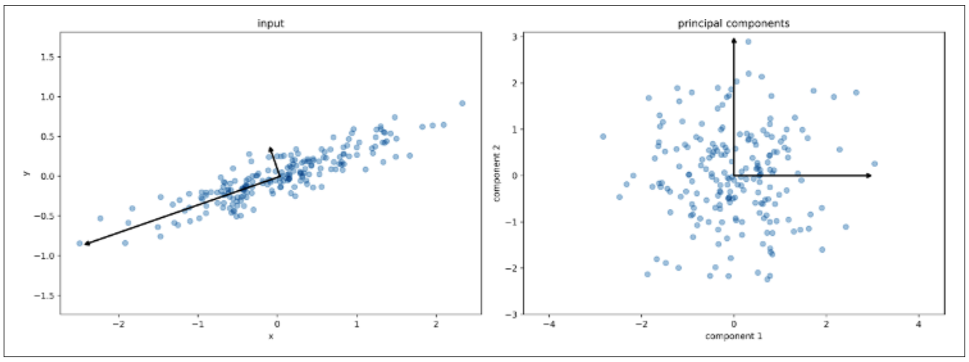

Figure 45-3. Transformed principal axes in the data[1]

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)
Out[7]: 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 Figure 45-4).

```
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');
```

1 Code to produce this figure can be found in the online appendix.

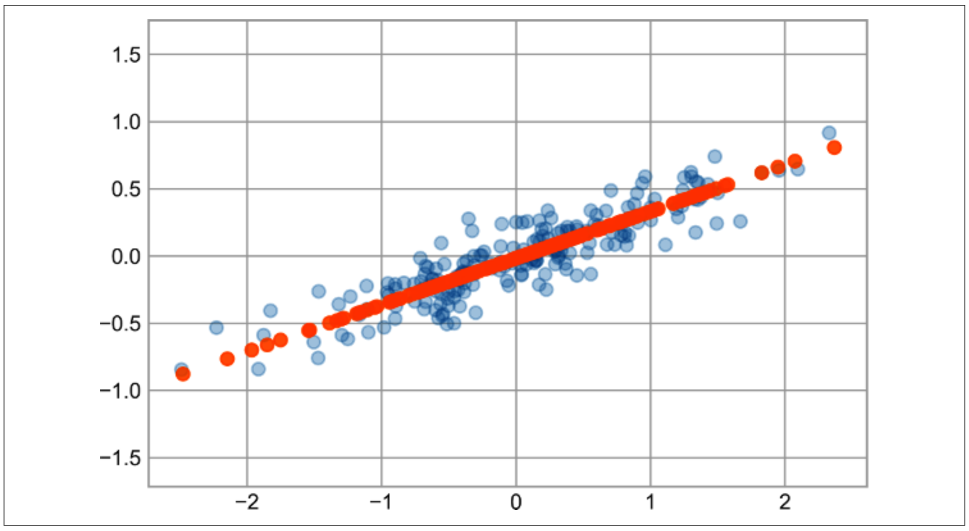

Figure 45-4. Visualization of PCA as dimensionality reduction

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.

This reduced-dimension dataset is in some senses "good enough" to encode the most important relationships between the points: despite reducing the number of data features by 50%, the overall relationships between the data points are mostly preserved.

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 Chapter 44.

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)
Out[10]: (1797, 64)
         (1797, 2)
```

We can now plot the first two principal components of each point to learn about the data, as seen in Figure 45-5.

```
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();
```

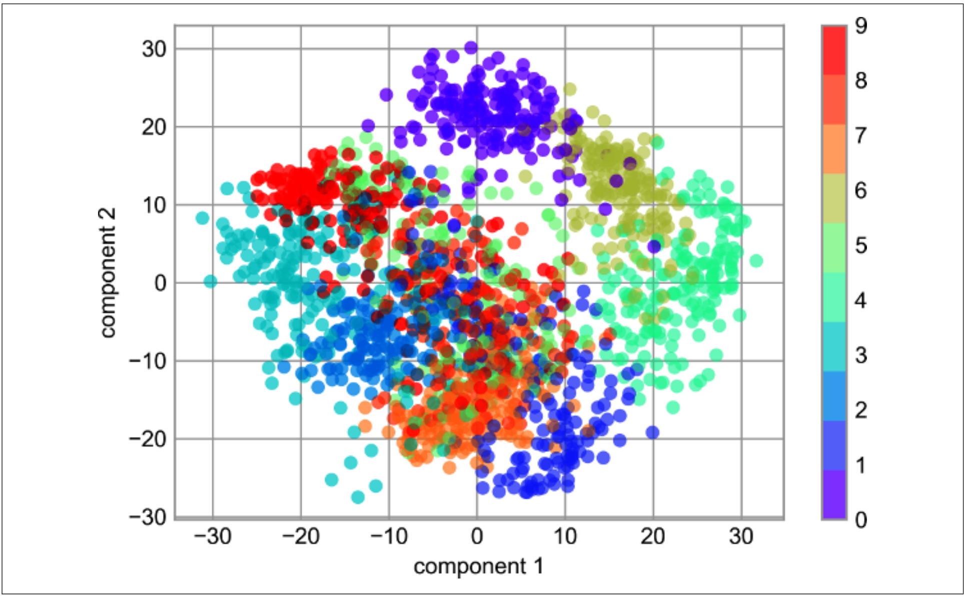

Figure 45-5. PCA applied to the handwritten digits data

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 (Figure 45-6). However, it is not very reflective of the whole image: we've thrown out nearly 90% of the pixels!

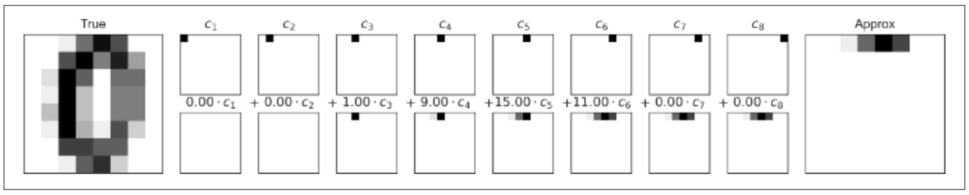

Figure 45-6. A naive dimensionality reduction achieved by discarding pixels[2]

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:

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

2 Code to produce this figure can be found in the online appendix.

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. Figure 45-7 shows a similar depiction of reconstructing the same digit using the mean plus the first eight PCA basis functions.

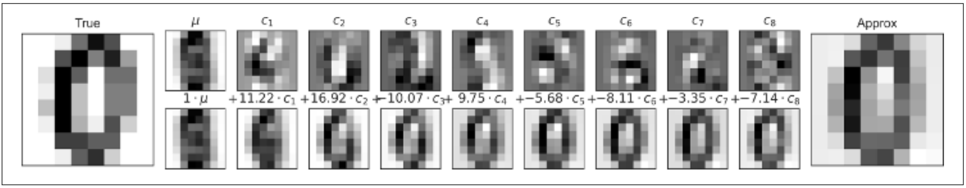

Figure 45-7. A more sophisticated dimensionality reduction achieved by discarding the least important principal components (compare to Figure 45-6)[3]

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 Figure 45-8).

```
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.

3 Code to produce this figure can be found in the online appendix.

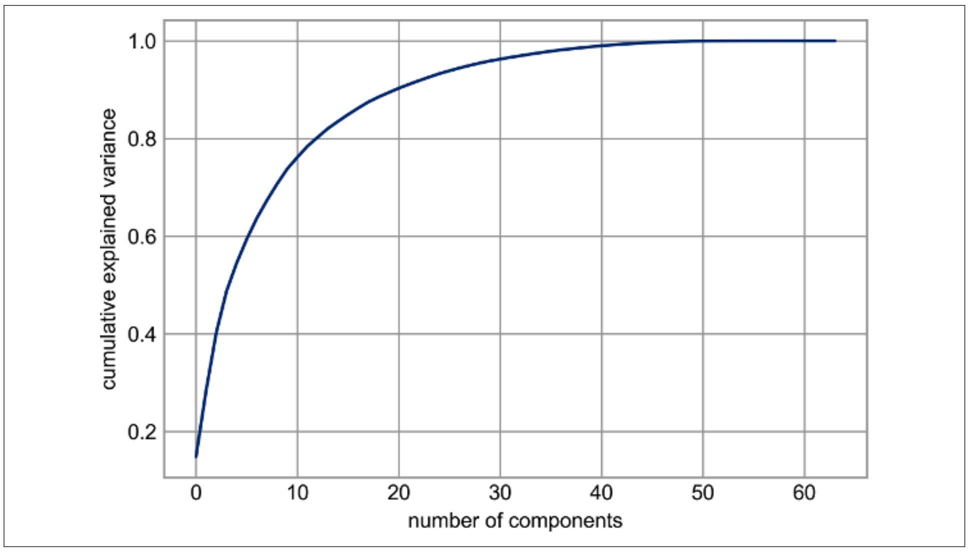

Figure 45-8. The cumulative explained variance, which measures how well PCA preserves the content of the data

This tells us that our two-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 (Figure 45-9).

```
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)
```

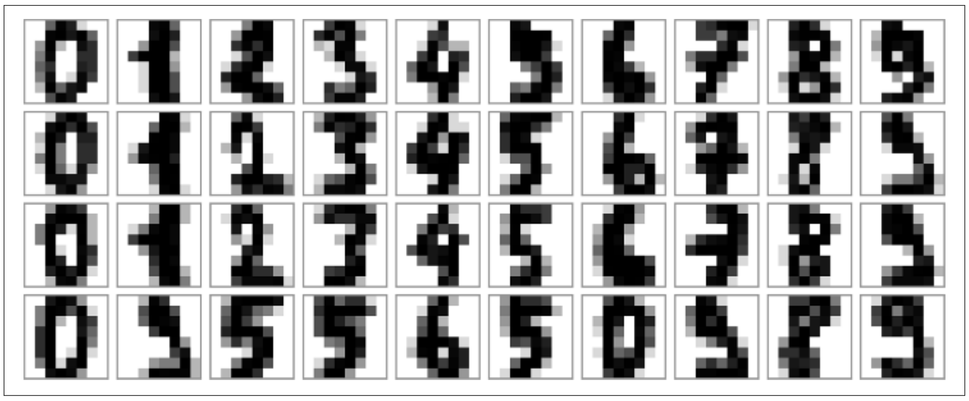
Figure 45-9. Digits without noise

Now let's add some random noise to create a noisy dataset, and replot it (Figure 45-10).

```
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)
```

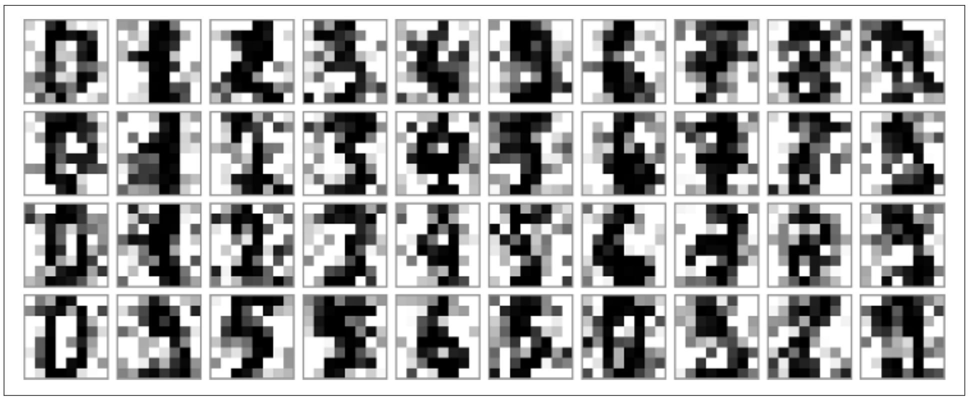
Figure 45-10. Digits with Gaussian random noise added

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; Figure 45-11 shows the result.

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

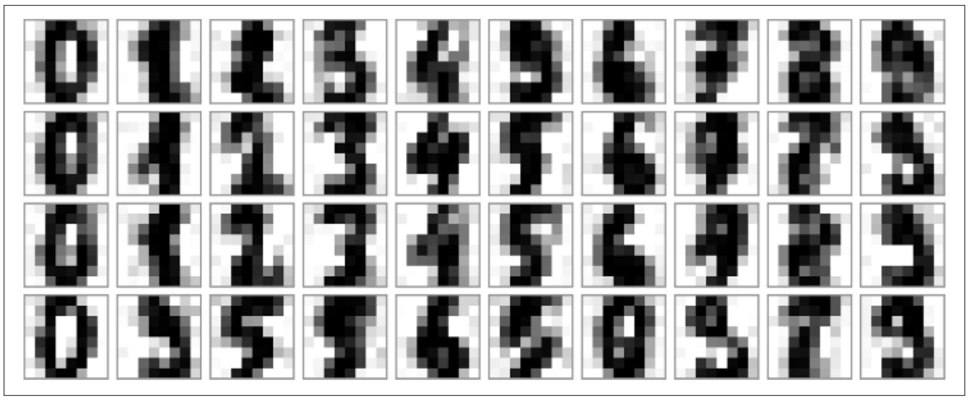
Figure 45-11. Digits "denoised" using PCA

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 Chapter 43). 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)
Out[18]: ['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 Figure 45-12, 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')
```

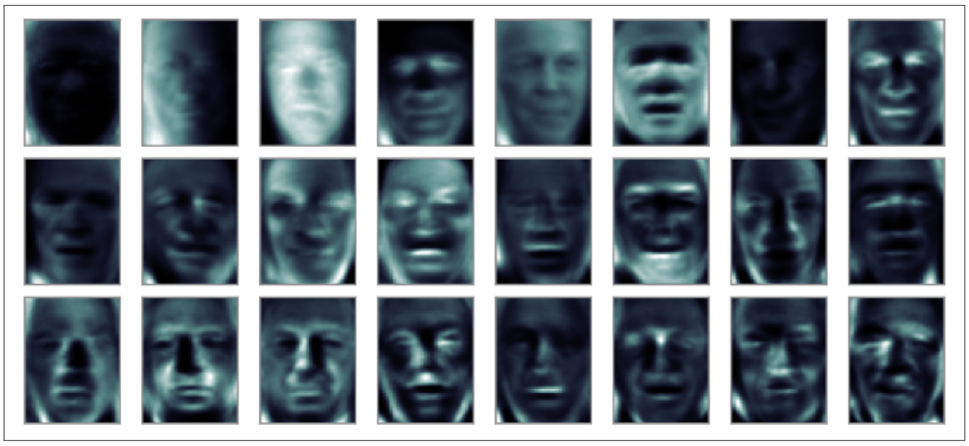

Figure 45-12. A visualization of eigenfaces learned from the LFW dataset

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 Figure 45-13).

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

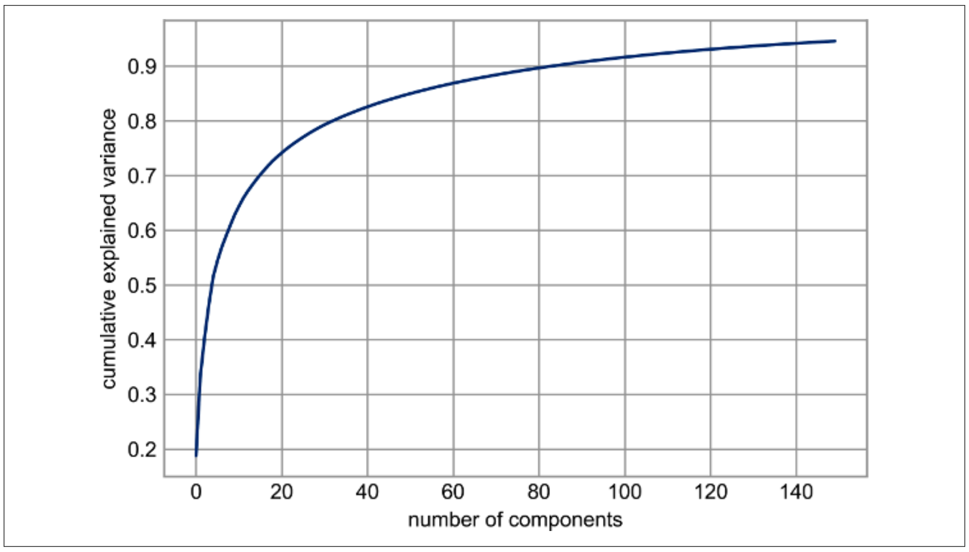

Figure 45-13. Cumulative explained variance for the LFW data

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 Figure 45-14).

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

```
In [23]: # 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');
```

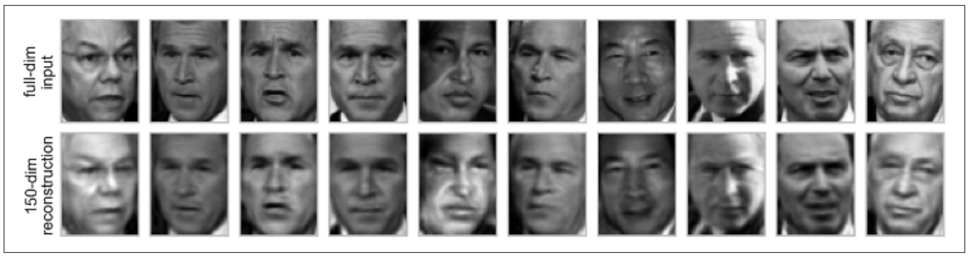

Figure 45-14. 150-dimensional PCA reconstruction of the LFW data

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 Chapter 43 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 Chapter 42) 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.

CHAPTER 46

In Depth: Manifold Learning

In the previous chapter we saw how PCA can be used for dimensionality reduction, reducing the number of features of a dataset while maintaining the essential relationships between the points. While PCA is flexible, fast, and easily interpretable, it does not perform so well when there are *nonlinear* relationships within the data, some examples of which we will see shortly.

To address this deficiency, we can turn to *manifold learning algorithms*—a class of unsupervised estimators that seek to describe datasets as low-dimensional manifolds embedded in high-dimensional spaces. When you think of a manifold, I'd suggest imagining a sheet of paper: this is a two-dimensional object that lives in our familiar three-dimensional world.

In the parlance of manifold learning, you can think of this sheet as a two-dimensional manifold embedded in three-dimensional space. Rotating, reorienting, or stretching the piece of paper in three-dimensional space doesn't change its flat geometry: such operations are akin to linear embeddings. If you bend, curl, or crumple the paper, it is still a two-dimensional manifold, but the embedding into the three-dimensional space is no longer linear. Manifold learning algorithms seek to learn about the fundamental two-dimensional nature of the paper, even as it is contorted to fill the three-dimensional space.

Here we will examine a number of manifold methods, going most deeply into a subset of these techniques: multidimensional scaling (MDS), locally linear embedding (LLE), and isometric mapping (Isomap).

We begin with the standard imports:

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

Manifold Learning: “HELLO”

To make these concepts more clear, let’s start by generating some two-dimensional data that we can use to define a manifold. Here is a function that will create data in the shape of the word “HELLO”:

```
In [2]: def make_hello(N=1000, rseed=42):
            # Make a plot with "HELLO" text; save as PNG
            fig, ax = plt.subplots(figsize=(4, 1))
            fig.subplots_adjust(left=0, right=1, bottom=0, top=1)
            ax.axis('off')
            ax.text(0.5, 0.4, 'HELLO', va='center', ha='center',
                    weight='bold', size=85)
            fig.savefig('hello.png')
            plt.close(fig)

            # Open this PNG and draw random points from it
            from matplotlib.image import imread
            data = imread('hello.png')[::-1, :, 0].T
            rng = np.random.RandomState(rseed)
            X = rng.rand(4 * N, 2)
            i, j = (X * data.shape).astype(int).T
            mask = (data[i, j] < 1)
            X = X[mask]
            X[:, 0] *= (data.shape[0] / data.shape[1])
            X = X[:N]
            return X[np.argsort(X[:, 0])]
```

Let’s call the function and visualize the resulting data (Figure 46-1).

```
In [3]: X = make_hello(1000)
        colorize = dict(c=X[:, 0], cmap=plt.cm.get_cmap('rainbow', 5))
        plt.scatter(X[:, 0], X[:, 1], **colorize)
        plt.axis('equal');
```

The output is two dimensional, and consists of points drawn in the shape of the word “HELLO”. This data form will help us to see visually what these algorithms are doing.

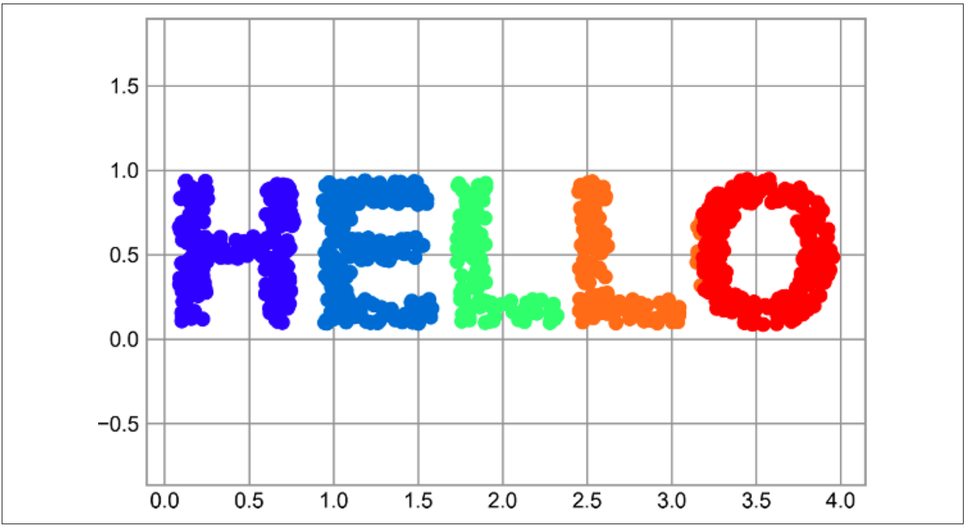

Figure 46-1. Data for use with manifold learning

Multidimensional Scaling

Looking at data like this, we can see that the particular choices of *x* and *y* values of the dataset are not the most fundamental description of the data: we can scale, shrink, or rotate the data, and the "HELLO" will still be apparent. For example, if we use a rotation matrix to rotate the data, the *x* and *y* values change, but the data is still fundamentally the same (see Figure 46-2).

```
In [4]: def rotate(X, angle):
            theta = np.deg2rad(angle)
            R = [[np.cos(theta), np.sin(theta)],
                 [-np.sin(theta), np.cos(theta)]]
            return np.dot(X, R)

        X2 = rotate(X, 20) + 5
        plt.scatter(X2[:, 0], X2[:, 1], **colorize)
        plt.axis('equal');
```

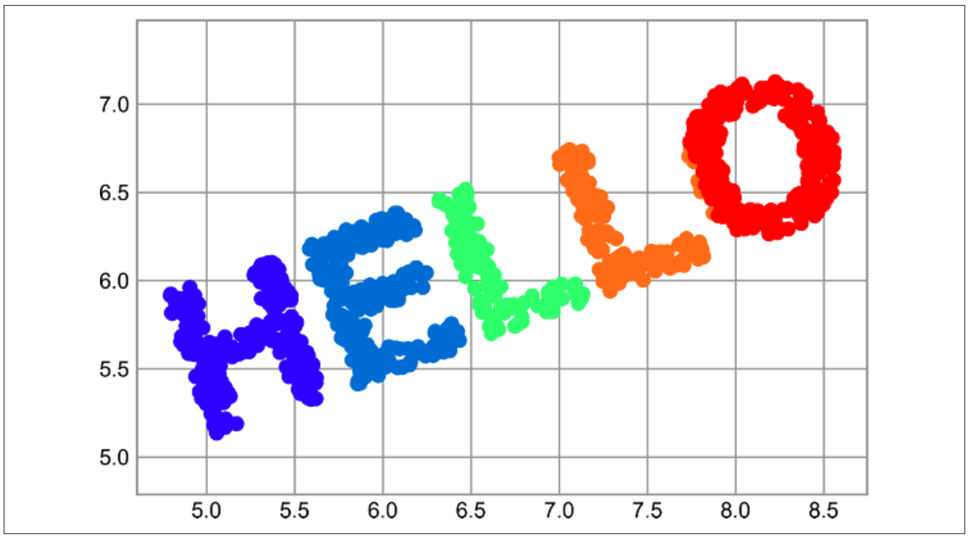

Figure 46-2. Rotated dataset

This confirms that the x and y values are not necessarily fundamental to the relationships in the data. What *is* fundamental, in this case, is the *distance* between each point within the dataset. A common way to represent this is to use a distance matrix: for N points, we construct an $N \times N$ array such that entry (i, j) contains the distance between point i and point j. Let's use Scikit-Learn's efficient `pairwise_distances` function to do this for our original data:

```
In [5]: from sklearn.metrics import pairwise_distances
        D = pairwise_distances(X)
        D.shape
Out[5]: (1000, 1000)
```

As promised, for our N=1,000 points, we obtain a 1,000 × 1,000 matrix, which can be visualized as shown here (see Figure 46-3).

```
In [6]: plt.imshow(D, zorder=2, cmap='viridis', interpolation='nearest')
        plt.colorbar();
```

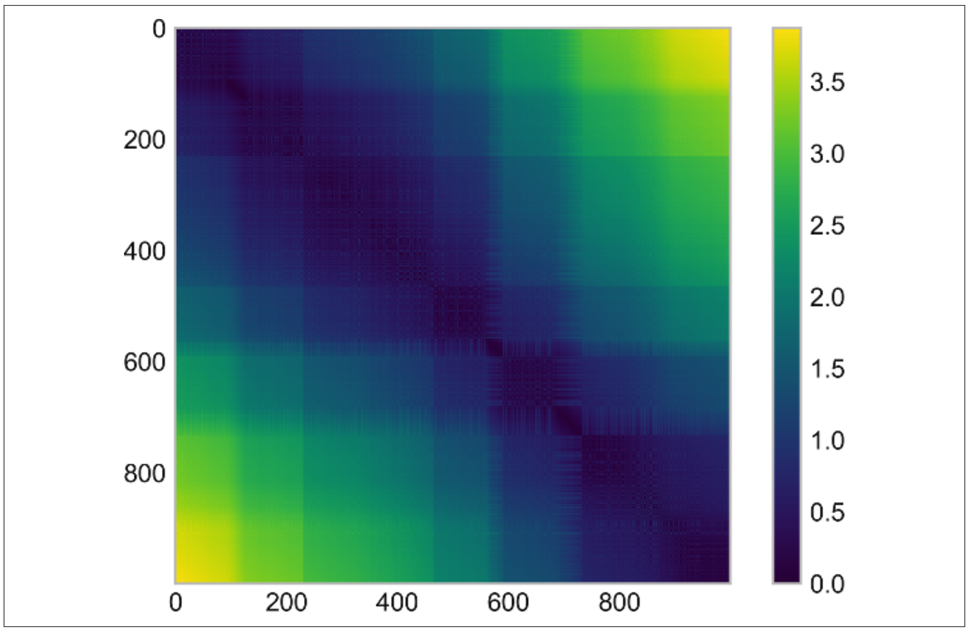

Figure 46-3. Visualization of the pairwise distances between points

If we similarly construct a distance matrix for our rotated and translated data, we see that it is the same:

```
In [7]: D2 = pairwise_distances(X2)
        np.allclose(D, D2)
Out[7]: True
```

This distance matrix gives us a representation of our data that is invariant to rotations and translations, but the visualization of the matrix in Figure 46-3 is not entirely intuitive. In the representation shown there, we have lost any visible sign of the interesting structure in the data: the "HELLO" that we saw before.

Further, while computing this distance matrix from the (x, y) coordinates is straightforward, transforming the distances back into x and y coordinates is rather difficult. This is exactly what the multidimensional scaling algorithm aims to do: given a distance matrix between points, it recovers a D-dimensional coordinate representation of the data. Let's see how it works for our distance matrix, using the `precomputed` dissimilarity to specify that we are passing a distance matrix (Figure 46-4).

```
In [8]: from sklearn.manifold import MDS
        model = MDS(n_components=2, dissimilarity='precomputed', random_state=1701)
        out = model.fit_transform(D)
        plt.scatter(out[:, 0], out[:, 1], **colorize)
        plt.axis('equal');
```

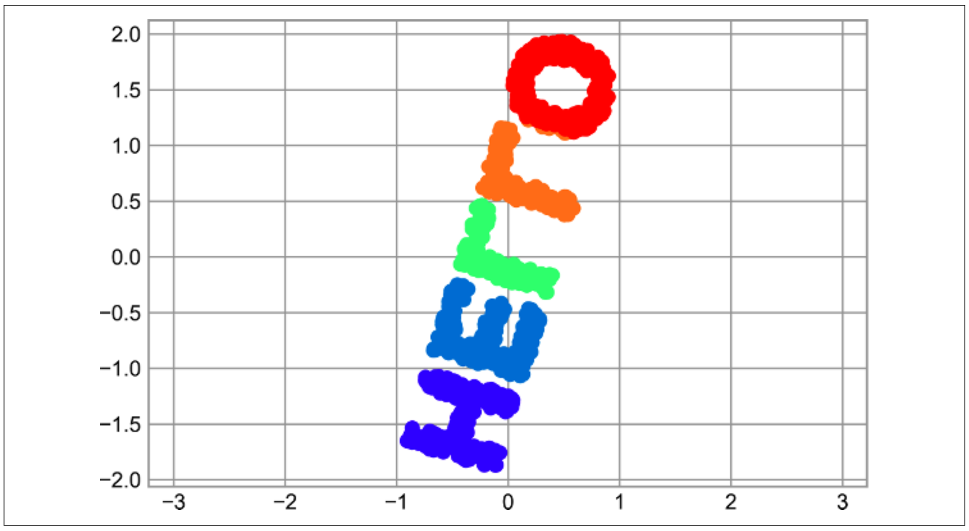

Figure 46-4. An MDS embedding computed from the pairwise distances

The MDS algorithm recovers one of the possible two-dimensional coordinate representations of our data, using *only* the $N \times N$ distance matrix describing the relationship between the data points.

MDS as Manifold Learning

The usefulness of this becomes more apparent when we consider the fact that distance matrices can be computed from data in *any* dimension. So, for example, instead of simply rotating the data in the two-dimensional plane, we can project it into three dimensions using the following function (essentially a three-dimensional generalization of the rotation matrix used earlier):

```
In [9]: def random_projection(X, dimension=3, rseed=42):
            assert dimension >= X.shape[1]
            rng = np.random.RandomState(rseed)
            C = rng.randn(dimension, dimension)
            e, V = np.linalg.eigh(np.dot(C, C.T))
            return np.dot(X, V[:X.shape[1]])

        X3 = random_projection(X, 3)
        X3.shape
Out[9]: (1000, 3)
```

Let's visualize these points to see what we're working with (Figure 46-5).

```
In [10]: from mpl_toolkits import mplot3d
         ax = plt.axes(projection='3d')
         ax.scatter3D(X3[:, 0], X3[:, 1], X3[:, 2],
                      **colorize);
```

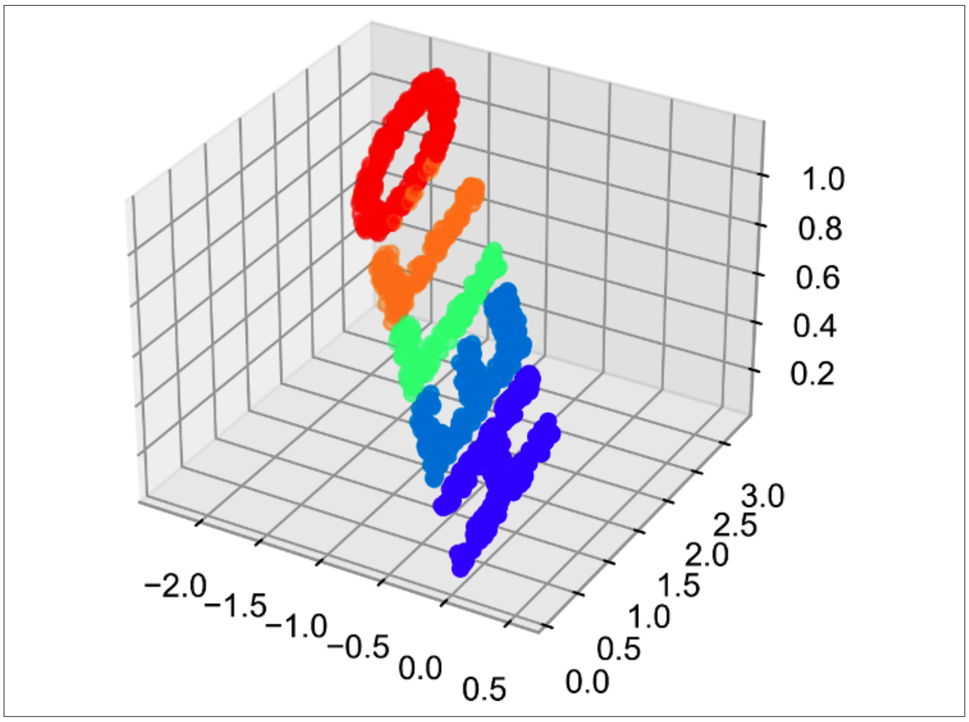

Figure 46-5. Data embedded linearly into three dimensions

We can now ask the `MDS` estimator to input this three-dimensional data, compute the distance matrix, and then determine the optimal two-dimensional embedding for this distance matrix. The result recovers a representation of the original data, as shown in Figure 46-6.

```
In [11]: model = MDS(n_components=2, random_state=1701)
         out3 = model.fit_transform(X3)
         plt.scatter(out3[:, 0], out3[:, 1], **colorize)
         plt.axis('equal');
```

This is essentially the goal of a manifold learning estimator: given high-dimensional embedded data, it seeks a low-dimensional representation of the data that preserves certain relationships within the data. In the case of MDS, the quantity preserved is the distance between every pair of points.

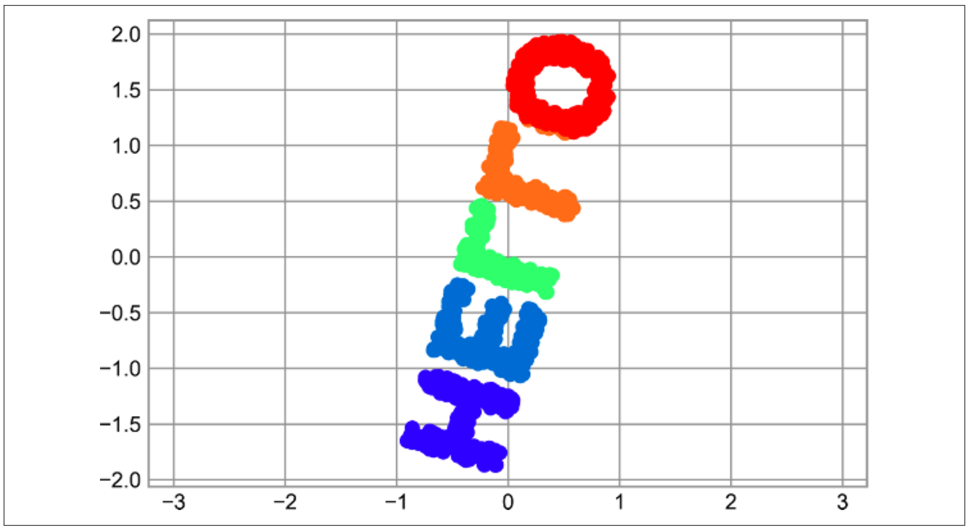

Figure 46-6. The MDS embedding of the three-dimensional data recovers the input up to a rotation and reflection

Nonlinear Embeddings: Where MDS Fails

Our discussion thus far has considered *linear* embeddings, which essentially consist of rotations, translations, and scalings of data into higher-dimensional spaces. Where MDS breaks down is when the embedding is nonlinear—that is, when it goes beyond this simple set of operations. Consider the following embedding, which takes the input and contorts it into an "S" shape in three dimensions:

```
In [12]: def make_hello_s_curve(X):
             t = (X[:, 0] - 2) * 0.75 * np.pi
             x = np.sin(t)
             y = X[:, 1]
             z = np.sign(t) * (np.cos(t) - 1)
             return np.vstack((x, y, z)).T

         XS = make_hello_s_curve(X)
```

This is again three-dimensional data, but as we can see in Figure 46-7 the embedding is much more complicated.

```
In [13]: from mpl_toolkits import mplot3d
         ax = plt.axes(projection='3d')
         ax.scatter3D(XS[:, 0], XS[:, 1], XS[:, 2],
                      **colorize);
```

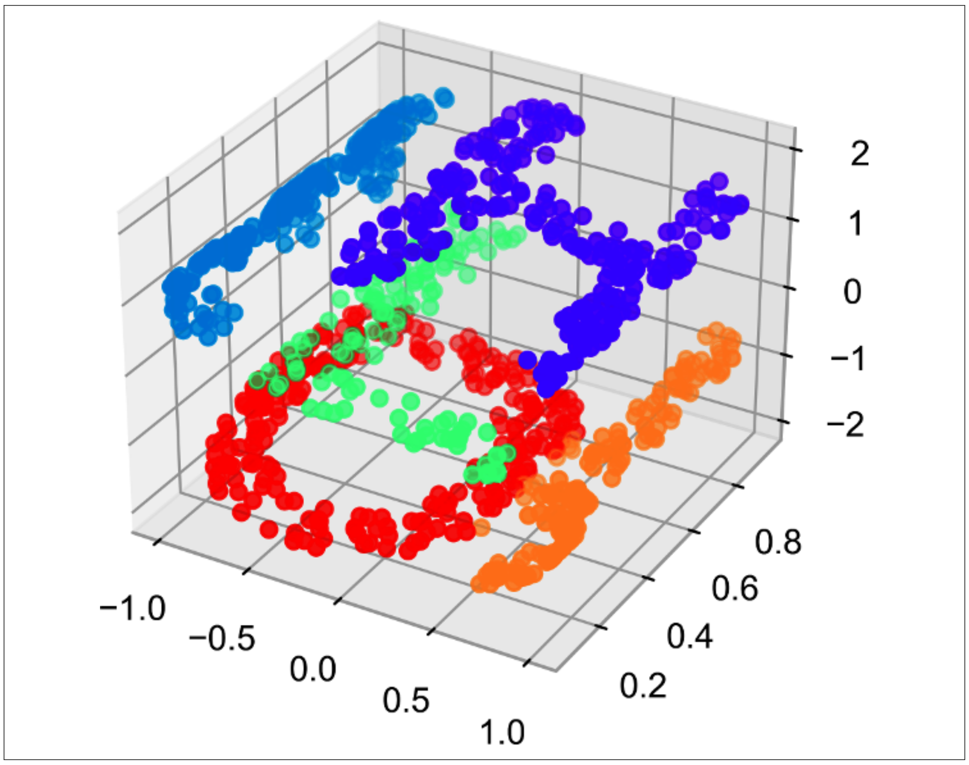

Figure 46-7. Data embedded nonlinearly into three dimensions

The fundamental relationships between the data points are still there, but this time the data has been transformed in a nonlinear way: it has been wrapped up into the shape of an "S."

If we try a simple MDS algorithm on this data, it is not able to "unwrap" this nonlinear embedding, and we lose track of the fundamental relationships in the embedded manifold (see Figure 46-8).

```
In [14]: from sklearn.manifold import MDS
         model = MDS(n_components=2, random_state=2)
         outS = model.fit_transform(XS)
         plt.scatter(outS[:, 0], outS[:, 1], **colorize)
         plt.axis('equal');
```

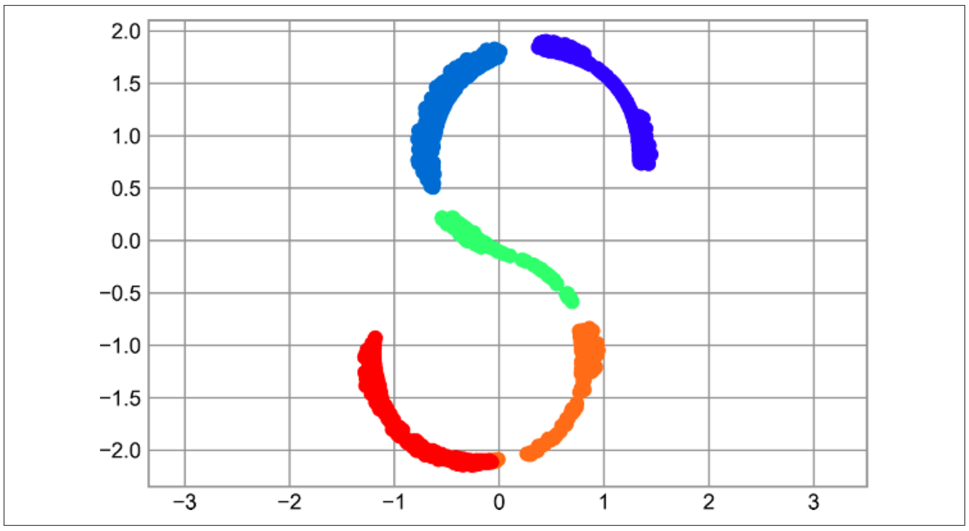

Figure 46-8. The MDS algorithm applied to the nonlinear data; it fails to recover the underlying structure

The best two-dimensional *linear* embedding does not unwrap the S-curve, but instead discards the original y-axis.

Nonlinear Manifolds: Locally Linear Embedding

How can we move forward here? Stepping back, we can see that the source of the problem is that MDS tries to preserve distances between faraway points when constructing the embedding. But what if we instead modified the algorithm such that it only preserves distances between nearby points? The resulting embedding would be closer to what we want.

Visually, we can think of it as illustrated Figure 46-9.

Here each faint line represents a distance that should be preserved in the embedding. On the left is a representation of the model used by MDS: it tries to preserve the distances between each pair of points in the dataset. On the right is a representation of the model used by a manifold learning algorithm called *locally linear embedding*: rather than preserving *all* distances, it instead tries to preserve only the distances between *neighboring points* (in this case, the nearest 100 neighbors of each point).

Thinking about the left panel, we can see why MDS fails: there is no way to unroll this data while adequately preserving the length of every line drawn between the two points. For the right panel, on the other hand, things look a bit more optimistic. We could imagine unrolling the data in a way that keeps the lengths of the lines approximately the same. This is precisely what LLE does, through a global optimization of a cost function reflecting this logic.

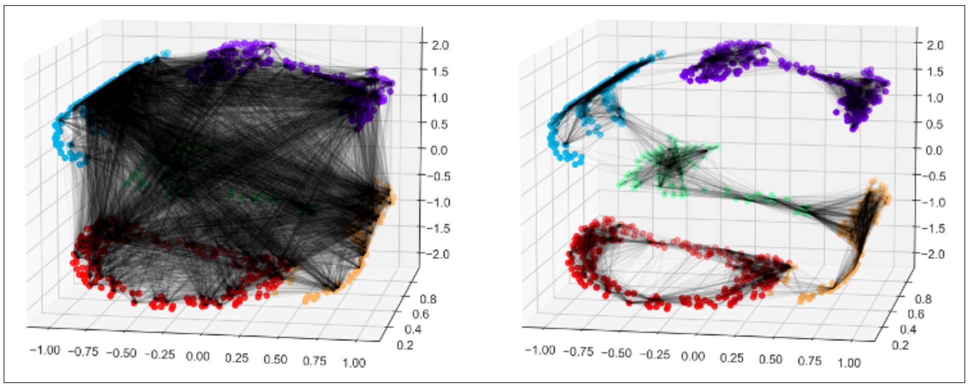

Figure 46-9. Representation of linkages between points within MDS and LLE[1]

LLE comes in a number of flavors; here we will use the *modified LLE* algorithm to recover the embedded two-dimensional manifold. In general, modified LLE does better than other flavors of the algorithm at recovering well-defined manifolds with very little distortion (see Figure 46-10).

```
In [15]: from sklearn.manifold import LocallyLinearEmbedding
         model = LocallyLinearEmbedding(
             n_neighbors=100, n_components=2,
             method='modified', eigen_solver='dense')
         out = model.fit_transform(XS)

         fig, ax = plt.subplots()
         ax.scatter(out[:, 0], out[:, 1], **colorize)
         ax.set_ylim(0.15, -0.15);
```

The result remains somewhat distorted compared to our original manifold, but captures the essential relationships in the data!

1 Code to produce this figure can be found in the online appendix.

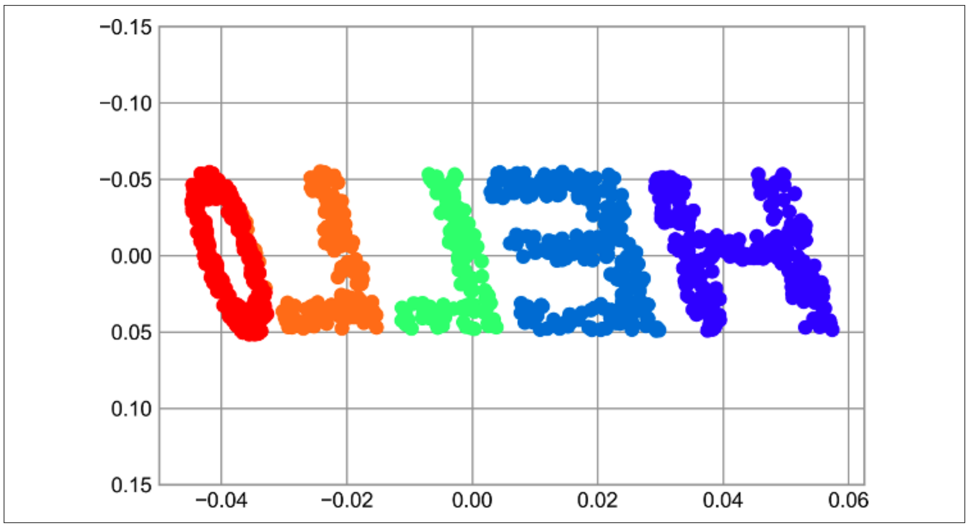

Figure 46-10. Locally linear embedding can recover the underlying data from a nonlinearly embedded input

Some Thoughts on Manifold Methods

Compelling as these examples may be, in practice manifold learning techniques tend to be finicky enough that they are rarely used for anything more than simple qualitative visualization of high-dimensional data.

The following are some of the particular challenges of manifold learning, which all contrast poorly with PCA:

- In manifold learning, there is no good framework for handling missing data. In contrast, there are straightforward iterative approaches for dealing with missing data in PCA.
- In manifold learning, the presence of noise in the data can "short-circuit" the manifold and drastically change the embedding. In contrast, PCA naturally filters noise from the most important components.
- The manifold embedding result is generally highly dependent on the number of neighbors chosen, and there is generally no solid quantitative way to choose an optimal number of neighbors. In contrast, PCA does not involve such a choice.
- In manifold learning, the globally optimal number of output dimensions is difficult to determine. In contrast, PCA lets you find the number of output dimensions based on the explained variance.
- In manifold learning, the meaning of the embedded dimensions is not always clear. In PCA, the principal components have a very clear meaning.

- 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 it with PCA.

Scikit-Learn implements several common variants of manifold learning beyond LLE and Isomap (which we've used in a few of the previous chapters and will look at in the next section): 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, 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 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 it 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 a collection of points in 1,000 dimensions, with the brightness of each pixel in each image defining the coordinate in that dimension.

To illustrate, let's apply Isomap on some data from the Labeled Faces in the Wild dataset, which we previously saw in Chapters 43 and 45. Running this command will download the dataset 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 display several of these images to remind us what we're working with (see Figure 46-11).

```
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')
```

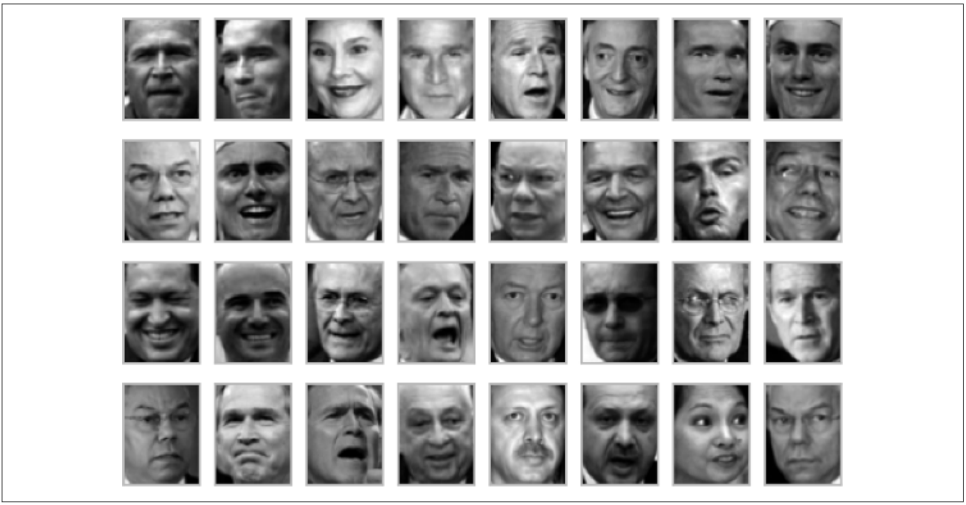

Figure 46-11. Examples of the input faces

When we encountered this data in Chapter 45, our goal was essentially compression: to use the components to reconstruct the inputs from the lower-dimensional representation.

PCA is versatile enough that we can also use it in this context, where we would like to plot a low-dimensional embedding of the 2,914-dimensional data to learn the fundamental relationships between the images. Let's again look at the explained variance ratio, which will give us an idea of how many linear features are required to describe the data (see Figure 46-12).

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

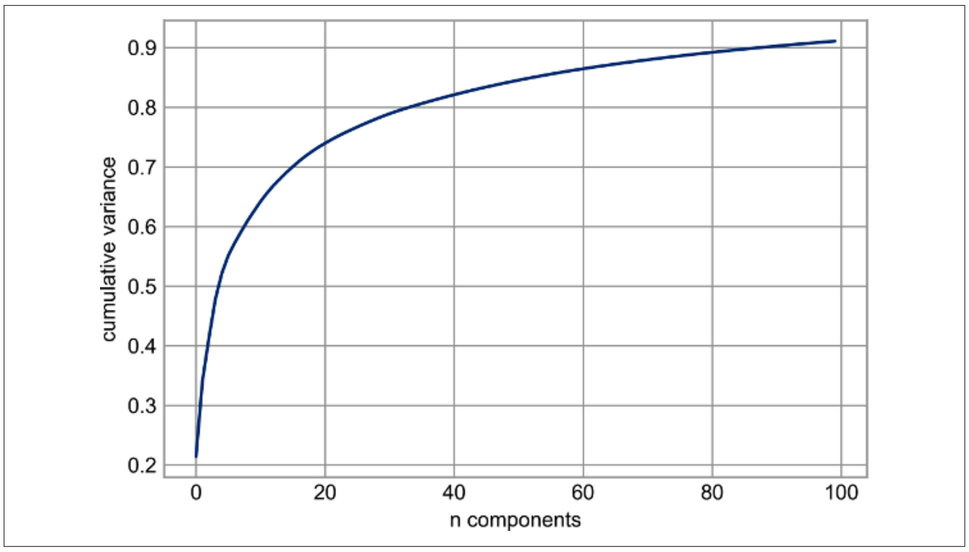

Figure 46-12. Cumulative variance from the PCA projection

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.

When this is the case, nonlinear manifold embeddings like LLE and Isomap may 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 in Figure 46-13.

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

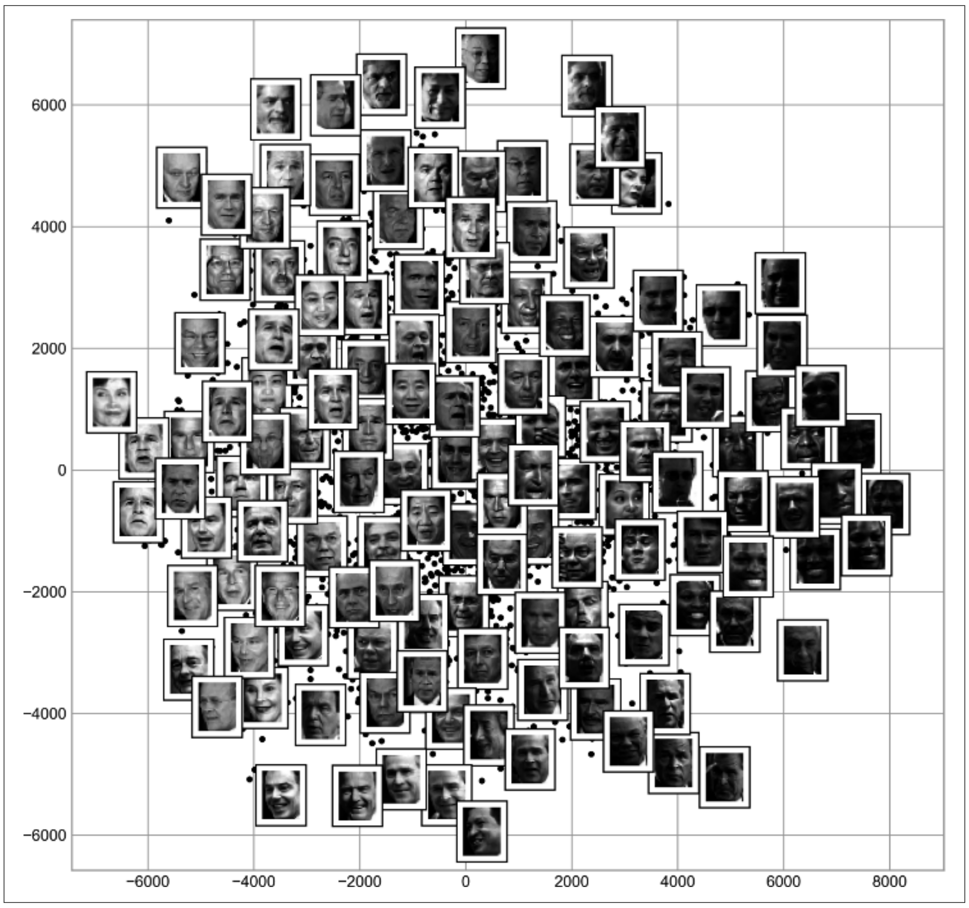

Figure 46-13. Isomap embedding of the LFW data

The result is interesting. The first two Isomap dimensions seem to describe global image features: the overall brightness 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.

From here, we could then go on to classify this data (perhaps using manifold features as inputs to the classification algorithm) as we did in Chapter 43.

Example: Visualizing Structure in Digits

As another example of using manifold learning for visualization, let's take a look at the MNIST handwritten digits dataset. This is similar to the digits dataset we saw in Chapter 44, but with many more pixels per image. It can be downloaded from *http://openml.org* with the Scikit-Learn utility:

```
In [22]: from sklearn.datasets import fetch_openml
         mnist = fetch_openml('mnist_784')
         mnist.data.shape
Out[22]: (70000, 784)
```

The dataset 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 (see Figure 46-14).

```
In [23]: mnist_data = np.asarray(mnist.data)
         mnist_target = np.asarray(mnist.target, dtype=int)

         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')
```

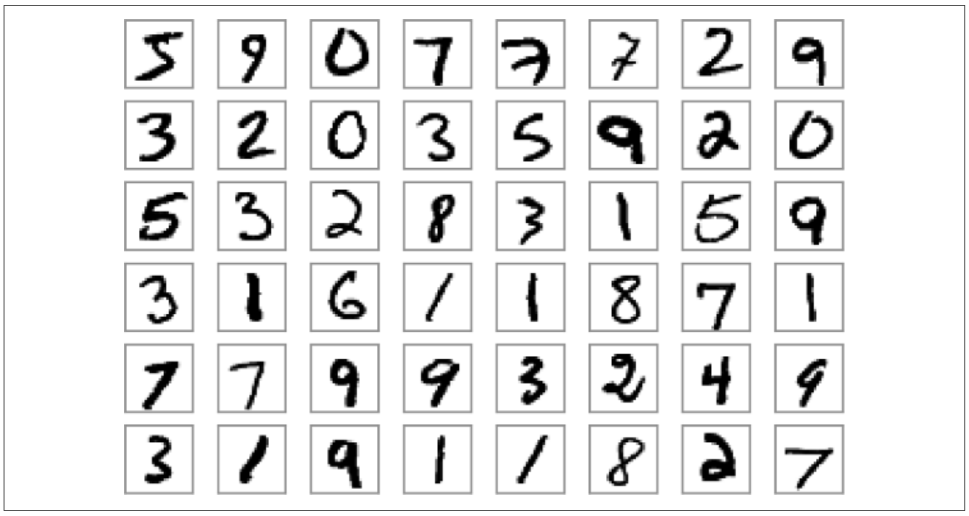

Figure 46-14. Examples of MNIST digits

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

Let's compute a manifold learning projection across the data. 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). Figure 46-15 shows the result.

```
In [24]: # Use only 1/30 of the data: full dataset takes a long time!
         data = mnist_data[::30]
         target = mnist_target[::30]

         model = Isomap(n_components=2)
         proj = model.fit_transform(data)

         plt.scatter(proj[:, 0], proj[:, 1], c=target,
                                 cmap=plt.cm.get_cmap('jet', 10))
         plt.colorbar(ticks=range(10))
         plt.clim(-0.5, 9.5);
```

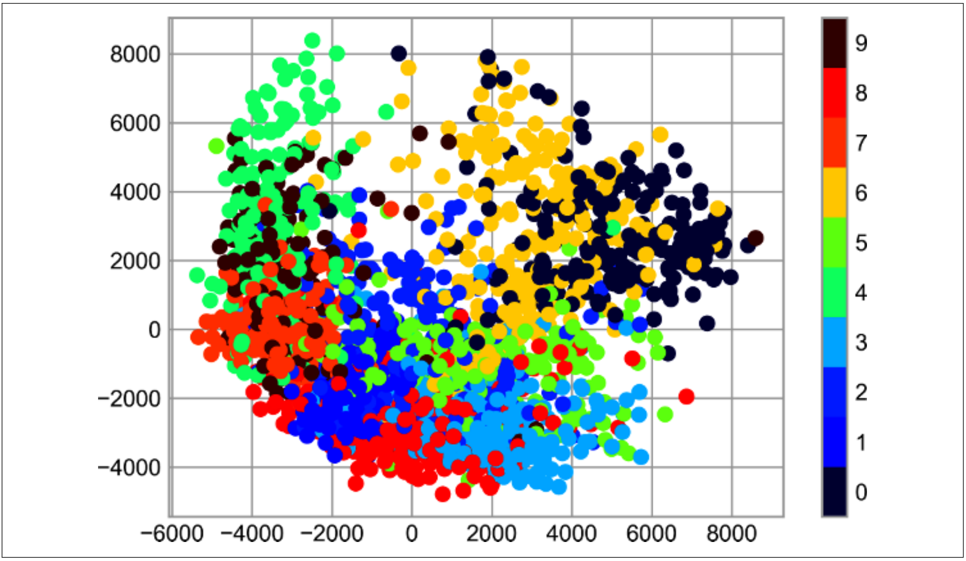

Figure 46-15. Isomap embedding of the MNIST digit data

The resulting scatter plot shows some of the relationships between the data points, but is a bit crowded. We can gain more insight by looking at just a single number at a time (see Figure 46-16).

```
In [25]: # Choose 1/4 of the "1" digits to project
         data = mnist_data[mnist_target == 1][::4]

         fig, ax = plt.subplots(figsize=(10, 10))
         model = Isomap(n_neighbors=5, n_components=2, eigen_solver='dense')
```

```
plot_components(data, model, images=data.reshape((-1, 28, 28)),
                ax=ax, thumb_frac=0.05, cmap='gray_r')
```

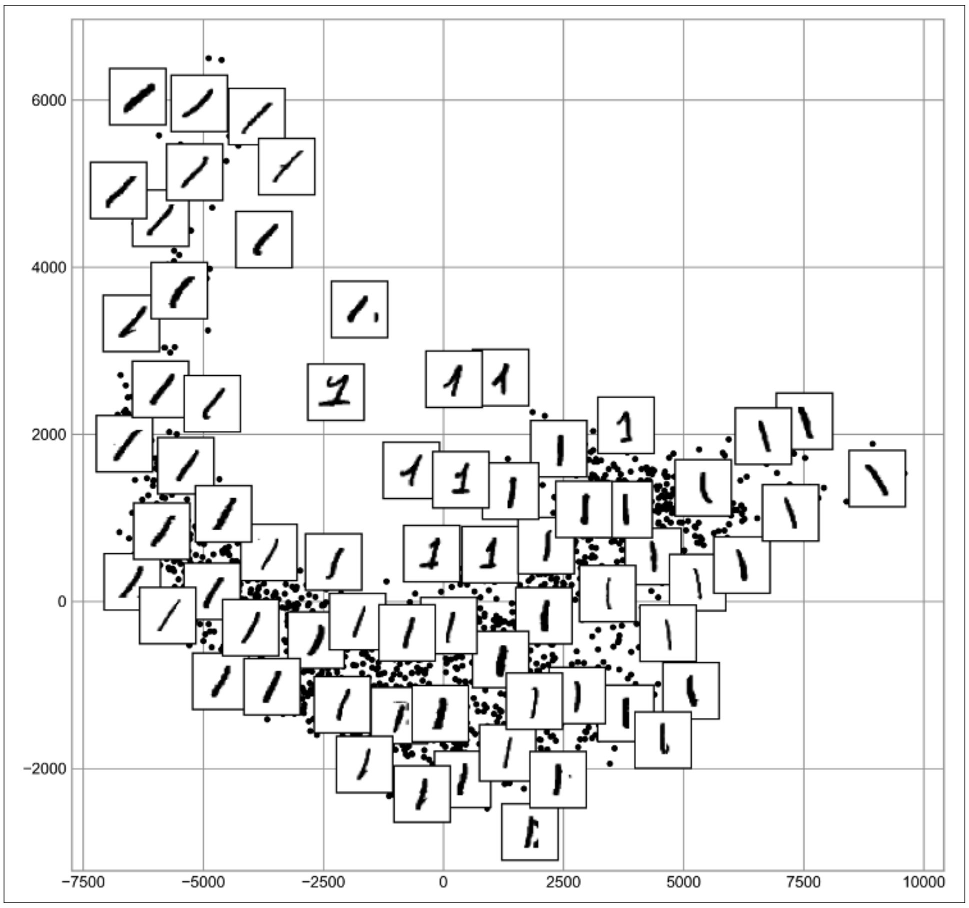

Figure 46-16. Isomap embedding of only the 1s within the MNIST dataset

The result gives you an idea of the variety of forms that the number 1 can take within the dataset. The data lies along a broad curve in the projected space, which appears to trace the orientation of the digit. As you move up the plot, you find 1s that have hats and/or bases, though these are very sparse within the dataset. The projection lets us identify outliers that have data issues: for example, pieces of the neighboring digits that snuck into the extracted images.

Now, this in itself may not be useful for the task of classifying digits, but it does help us get an understanding of the data, and may give us ideas about how to move forward—such as how we might want to preprocess the data before building a classification pipeline.

CHAPTER 49

In Depth: Kernel Density Estimation

In Chapter 48 we covered Gaussian mixture models, which are a kind of hybrid between a clustering estimator and a density estimator. Recall that a density estimator is an algorithm that takes a *D*-dimensional dataset and produces an estimate of the *D*-dimensional probability distribution that data is drawn from. The GMM algorithm accomplishes this by representing the density as a weighted sum of Gaussian distributions. *Kernel density estimation* (KDE) is in some senses an algorithm that takes the mixture-of-Gaussians idea to its logical extreme: it uses a mixture consisting of one Gaussian component *per point*, resulting in an essentially nonparametric estimator of density. In this chapter, we will explore the motivation and uses of KDE.

We begin with the standard imports:

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

Motivating Kernel Density Estimation: Histograms

As mentioned previously, a density estimator is an algorithm that seeks to model the probability distribution that generated a dataset. For one-dimensional data, you are probably already familiar with one simple density estimator: the histogram. A histogram divides the data into discrete bins, counts the number of points that fall in each bin, and then visualizes the results in an intuitive manner.

For example, let's create some data that is drawn from two normal distributions:

```
In [2]: def make_data(N, f=0.3, rseed=1):
            rand = np.random.RandomState(rseed)
            x = rand.randn(N)
            x[int(f * N):] += 5
            return x

        x = make_data(1000)
```

We have previously seen that the standard count-based histogram can be created with the `plt.hist` function. By specifying the `density` parameter of the histogram, we end up with a normalized histogram where the height of the bins does not reflect counts, but instead reflects probability density (see Figure 49-1).

```
In [3]: hist = plt.hist(x, bins=30, density=True)
```

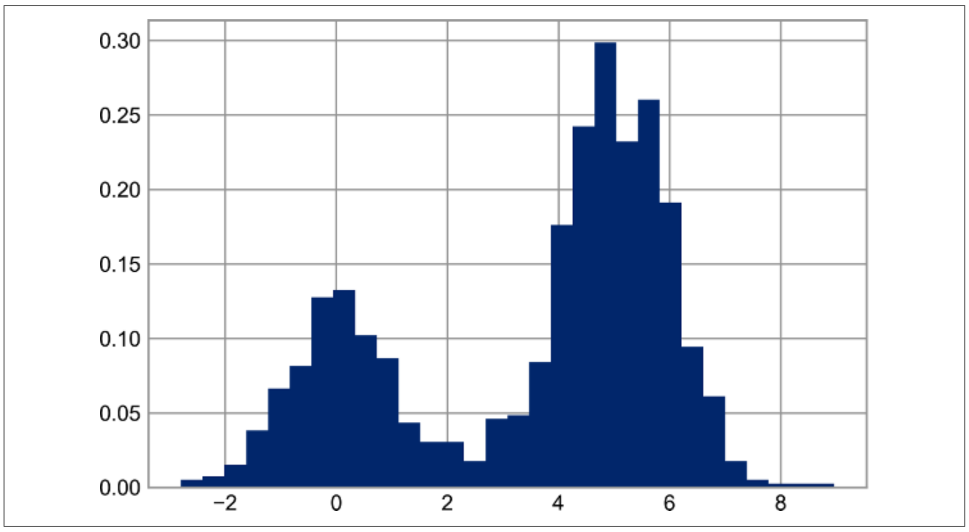

Figure 49-1. Data drawn from a combination of normal distributions

Notice that for equal binning, this normalization simply changes the scale on the y-axis, leaving the relative heights essentially the same as in a histogram built from counts. This normalization is chosen so that the total area under the histogram is equal to 1, as we can confirm by looking at the output of the histogram function:

```
In [4]: density, bins, patches = hist
        widths = bins[1:] - bins[:-1]
        (density * widths).sum()
Out[4]: 1.0
```

One of the issues with using a histogram as a density estimator is that the choice of bin size and location can lead to representations that have qualitatively different features. For example, if we look at a version of this data with only 20 points, the choice of how to draw the bins can lead to an entirely different interpretation of the data! Consider this example, visualized in Figure 49-2.

```
In [5]: x = make_data(20)
        bins = np.linspace(-5, 10, 10)
```

```
In [6]: fig, ax = plt.subplots(1, 2, figsize=(12, 4),
                               sharex=True, sharey=True,
                               subplot_kw={'xlim':(-4, 9),
                                           'ylim':(-0.02, 0.3)})
        fig.subplots_adjust(wspace=0.05)
        for i, offset in enumerate([0.0, 0.6]):
            ax[i].hist(x, bins=bins + offset, density=True)
            ax[i].plot(x, np.full_like(x, -0.01), '|k',
                       markeredgewidth=1)
```

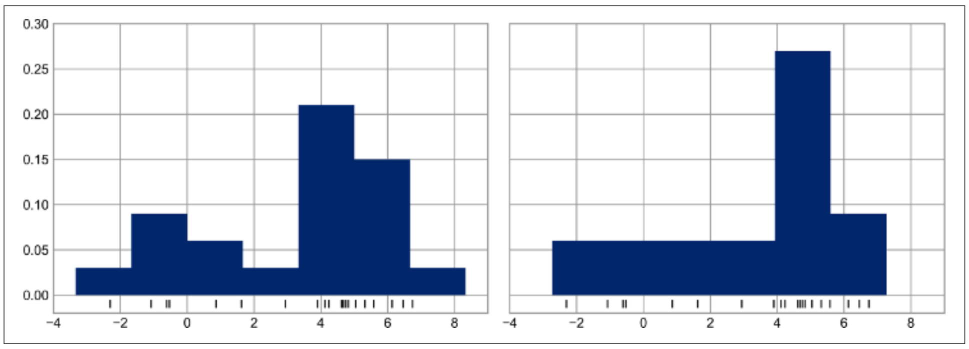

Figure 49-2. The problem with histograms: the bin locations can affect interpretation

On the left, the histogram makes clear that this is a bimodal distribution. On the right, we see a unimodal distribution with a long tail. Without seeing the preceding code, you would probably not guess that these two histograms were built from the same data. With that in mind, how can you trust the intuition that histograms confer? And how might we improve on this?

Stepping back, we can think of a histogram as a stack of blocks, where we stack one block within each bin on top of each point in the dataset. Let's view this directly (see Figure 49-3).

```
In [7]: fig, ax = plt.subplots()
        bins = np.arange(-3, 8)
        ax.plot(x, np.full_like(x, -0.1), '|k',
                markeredgewidth=1)
        for count, edge in zip(*np.histogram(x, bins)):
            for i in range(count):
                ax.add_patch(plt.Rectangle(
                  (edge, i), 1, 1, ec='black', alpha=0.5))
        ax.set_xlim(-4, 8)
        ax.set_ylim(-0.2, 8)
Out[7]: (-0.2, 8.0)
```

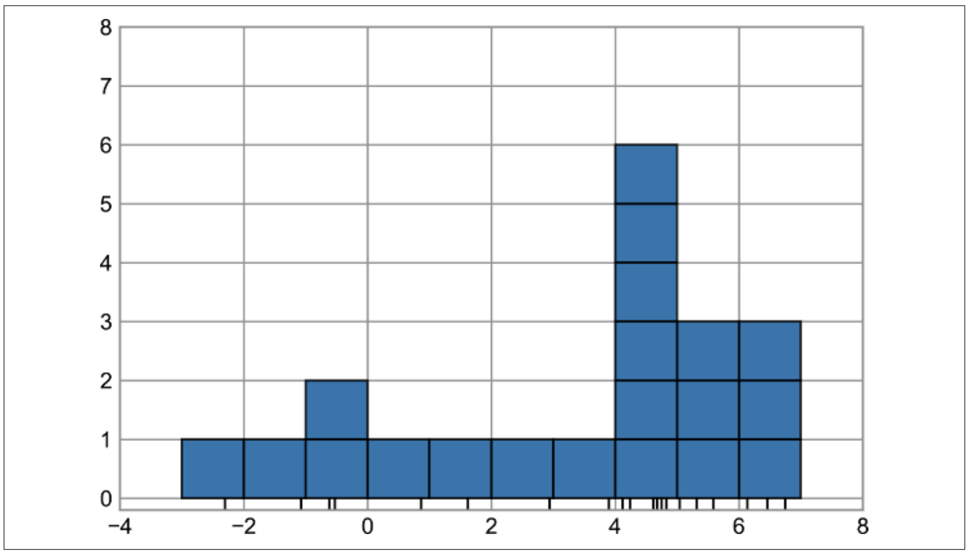

Figure 49-3. Histogram as stack of blocks

The problem with our two binnings stems from the fact that the height of the block stack often reflects not the actual density of points nearby, but coincidences of how the bins align with the data points. This misalignment between points and their blocks is a potential cause of the poor histogram results seen here. But what if, instead of stacking the blocks aligned with the *bins*, we were to stack the blocks aligned with the *points they represent*? If we do this, the blocks won't be aligned, but we can add their contributions at each location along the x-axis to find the result. Let's try this (see Figure 49-4).

```
In [8]: x_d = np.linspace(-4, 8, 2000)
        density = sum((abs(xi - x_d) < 0.5) for xi in x)

        plt.fill_between(x_d, density, alpha=0.5)
        plt.plot(x, np.full_like(x, -0.1), '|k', markeredgewidth=1)

        plt.axis([-4, 8, -0.2, 8]);
```

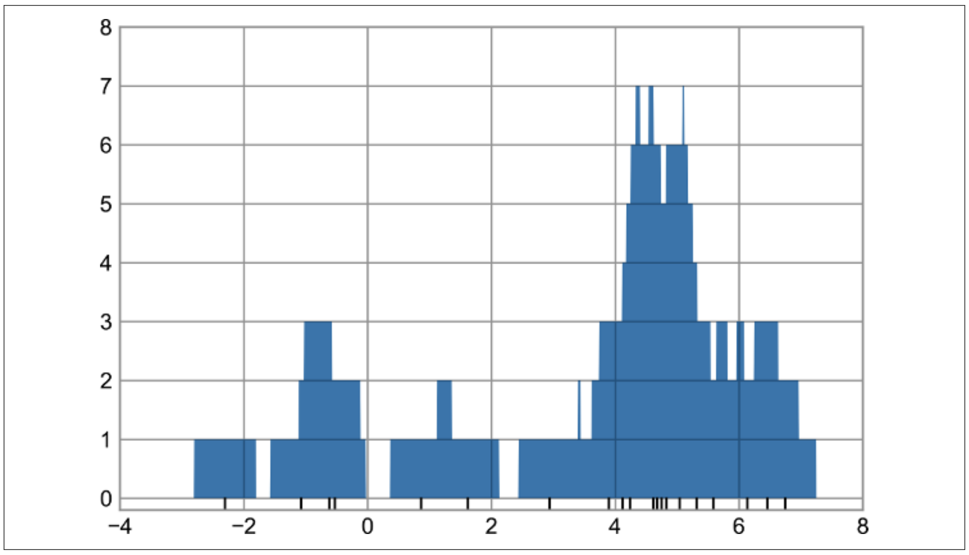

Figure 49-4. A "histogram" where blocks center on each individual point; this is an example of a kernel density estimate

The result looks a bit messy, but it's a much more robust reflection of the actual data characteristics than is the standard histogram. Still, the rough edges are not aesthetically pleasing, nor are they reflective of any true properties of the data. In order to smooth them out, we might decide to replace the blocks at each location with a smooth function, like a Gaussian. Let's use a standard normal curve at each point instead of a block (see Figure 49-5).

```
In [9]: from scipy.stats import norm
        x_d = np.linspace(-4, 8, 1000)
        density = sum(norm(xi).pdf(x_d) for xi in x)

        plt.fill_between(x_d, density, alpha=0.5)
        plt.plot(x, np.full_like(x, -0.1), '|k', markeredgewidth=1)

        plt.axis([-4, 8, -0.2, 5]);
```

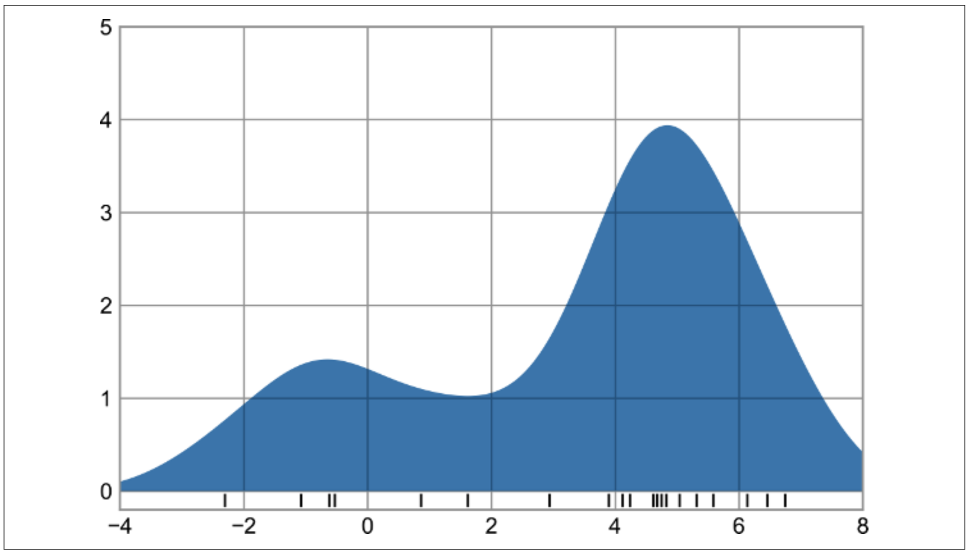

Figure 49-5. A kernel density estimate with a Gaussian kernel

This smoothed-out plot, with a Gaussian distribution contributed at the location of each input point, gives a much more accurate idea of the shape of the data distribution, and one that has much less variance (i.e., changes much less in response to differences in sampling).

What we've landed on in the last two plots is what's called kernel density estimation in one dimension: we have placed a "kernel"—a square or top hat–shaped kernel in the former, a Gaussian kernel in the latter—at the location of each point, and used their sum as an estimate of density. With this intuition in mind, we'll now explore kernel density estimation in more detail.

Kernel Density Estimation in Practice

The free parameters of kernel density estimation are the *kernel*, which specifies the shape of the distribution placed at each point, and the *kernel bandwidth*, which controls the size of the kernel at each point. In practice, there are many kernels you might use for kernel density estimation: in particular, the Scikit-Learn KDE implementation supports six kernels, which you can read about in the "Density Estimation" section of the documentation.

While there are several versions of KDE implemented in Python (notably in the SciPy and `statsmodels` packages), I prefer to use Scikit-Learn's version because of its efficiency and flexibility. It is implemented in the `sklearn.neighbors.KernelDensity` estimator, which handles KDE in multiple dimensions with one of six kernels and one of a couple dozen distance metrics. Because KDE can be fairly computationally intensive, the Scikit-Learn estimator uses a tree-based algorithm under the hood and can trade off computation time for accuracy using the `atol` (absolute tolerance) and `rtol` (relative tolerance) parameters. The kernel bandwidth can be determined using Scikit-Learn's standard cross-validation tools, as we will soon see.

Let's first show a simple example of replicating the previous plot using the Scikit-Learn `KernelDensity` estimator (see Figure 49-6).

```
In [10]: from sklearn.neighbors import KernelDensity

         # instantiate and fit the KDE model
         kde = KernelDensity(bandwidth=1.0, kernel='gaussian')
         kde.fit(x[:, None])

         # score_samples returns the log of the probability density
         logprob = kde.score_samples(x_d[:, None])

         plt.fill_between(x_d, np.exp(logprob), alpha=0.5)
         plt.plot(x, np.full_like(x, -0.01), '|k', markeredgewidth=1)
         plt.ylim(-0.02, 0.22);
```

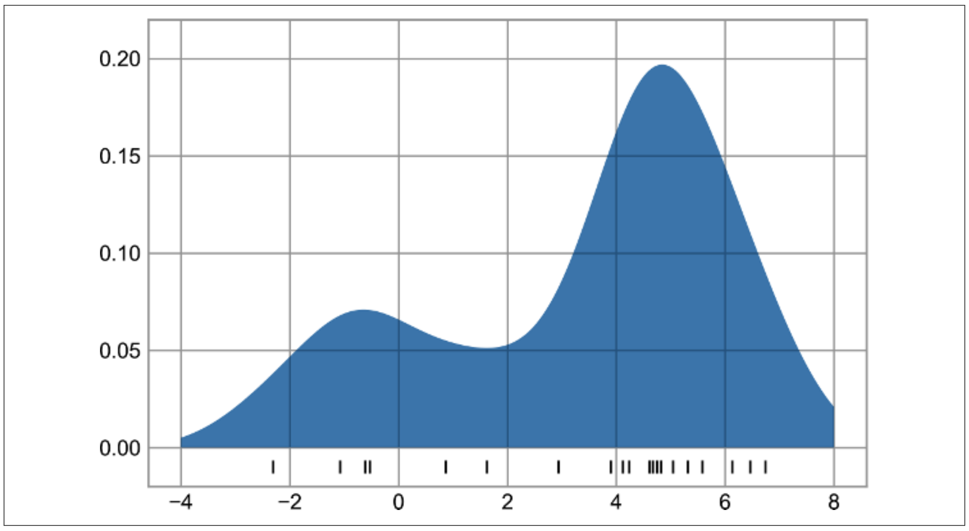

Figure 49-6. A kernel density estimate computed with Scikit-Learn

The result here is normalized such that the area under the curve is equal to 1.

Selecting the Bandwidth via Cross-Validation

The final estimate produced by a KDE procedure can be quite sensitive to the choice of bandwidth, which is the knob that controls the bias–variance trade-off in the estimate of density. Too narrow a bandwidth leads to a high-variance estimate (i.e., overfitting), where the presence or absence of a single point makes a large difference. Too wide a bandwidth leads to a high-bias estimate (i.e., underfitting), where the structure in the data is washed out by the wide kernel.

There is a long history in statistics of methods to quickly estimate the best bandwidth based on rather stringent assumptions about the data: if you look up the KDE implementations in the SciPy and `statsmodels` packages, for example, you will see implementations based on some of these rules.

In machine learning contexts, we've seen that such hyperparameter tuning often is done empirically via a cross-validation approach. With this in mind, Scikit-Learn's `KernelDensity` estimator is designed such that it can be used directly within the package's standard grid search tools. Here we will use `GridSearchCV` to optimize the bandwidth for the preceding dataset. Because we are looking at such a small dataset, we will use leave-one-out cross-validation, which minimizes the reduction in training set size for each cross-validation trial:

```
In [11]: from sklearn.model_selection import GridSearchCV
         from sklearn.model_selection import LeaveOneOut

         bandwidths = 10 ** np.linspace(-1, 1, 100)
         grid = GridSearchCV(KernelDensity(kernel='gaussian'),
                             {'bandwidth': bandwidths},
                             cv=LeaveOneOut())
         grid.fit(x[:, None]);
```

Now we can find the choice of bandwidth that maximizes the score (which in this case defaults to the log-likelihood):

```
In [12]: grid.best_params_
Out[12]: {'bandwidth': 1.1233240329780276}
```

The optimal bandwidth happens to be very close to what we used in the example plot earlier, where the bandwidth was 1.0 (i.e., the default width of `scipy.stats.norm`).

Example: Not-so-Naive Bayes

This example looks at Bayesian generative classification with KDE, and demonstrates how to use the Scikit-Learn architecture to create a custom estimator.

In Chapter 41 we explored naive Bayesian classification, in which we create a simple generative model for each class, and use these models to build a fast classifier. For Gaussian naive Bayes, the generative model is a simple axis-aligned Gaussian. With a density estimation algorithm like KDE, we can remove the "naive" element and perform the same classification with a more sophisticated generative model for each class. It's still Bayesian classification, but it's no longer naive.

The general approach for generative classification is this:

1. Split the training data by label.
2. For each set, fit a KDE to obtain a generative model of the data. This allows you, for any observation x and label y, to compute a likelihood $P(x \mid y)$.
3. From the number of examples of each class in the training set, compute the *class prior*, $P(y)$.
4. For an unknown point x, the posterior probability for each class is $P(y \mid x) \propto P(x \mid y)P(y)$. The class that maximizes this posterior is the label assigned to the point.

The algorithm is straightforward and intuitive to understand; the more difficult piece is couching it within the Scikit-Learn framework in order to make use of the grid search and cross-validation architecture.

This is the code that implements the algorithm within the Scikit-Learn framework; we will step through it following the code block:

```
In [13]: from sklearn.base import BaseEstimator, ClassifierMixin


         class KDEClassifier(BaseEstimator, ClassifierMixin):
             """Bayesian generative classification based on KDE

             Parameters
             ----------
             bandwidth : float
                 the kernel bandwidth within each class
             kernel : str
                 the kernel name, passed to KernelDensity
             """
             def __init__(self, bandwidth=1.0, kernel='gaussian'):
                 self.bandwidth = bandwidth
                 self.kernel = kernel

             def fit(self, X, y):
                 self.classes_ = np.sort(np.unique(y))
                 training_sets = [X[y == yi] for yi in self.classes_]
                 self.models_ = [KernelDensity(bandwidth=self.bandwidth,
                                               kernel=self.kernel).fit(Xi)
                                 for Xi in training_sets]
                 self.logpriors_ = [np.log(Xi.shape[0] / X.shape[0])
                                    for Xi in training_sets]
                 return self
```

```
    def predict_proba(self, X):
        logprobs = np.array([model.score_samples(X)
                             for model in self.models_]).T
        result = np.exp(logprobs + self.logpriors_)
        return result / result.sum(axis=1, keepdims=True)

    def predict(self, X):
        return self.classes_[np.argmax(self.predict_proba(X), 1)]
```

Anatomy of a Custom Estimator

Let's step through this code and discuss the essential features:

```
from sklearn.base import BaseEstimator, ClassifierMixin

class KDEClassifier(BaseEstimator, ClassifierMixin):
    """Bayesian generative classification based on KDE

    Parameters
    ----------
    bandwidth : float
        the kernel bandwidth within each class
    kernel : str
        the kernel name, passed to KernelDensity
    """
```

Each estimator in Scikit-Learn is a class, and it is most convenient for this class to inherit from the `BaseEstimator` class as well as the appropriate mixin, which provides standard functionality. For example, here the `BaseEstimator` contains (among other things) the logic necessary to clone/copy an estimator for use in a cross-validation procedure, and `ClassifierMixin` defines a default `score` method used by such routines. We also provide a docstring, which will be captured by IPython's help functionality (see Chapter 1).

Next comes the class initialization method:

```
    def __init__(self, bandwidth=1.0, kernel='gaussian'):
        self.bandwidth = bandwidth
        self.kernel = kernel
```

This is the actual code that is executed when the object is instantiated with `KDEClassifier`. In Scikit-Learn, it is important that *initialization contains no operations* other than assigning the passed values by name to `self`. This is due to the logic contained in `BaseEstimator` required for cloning and modifying estimators for cross-validation, grid search, and other functions. Similarly, all arguments to `__init__` should be explicit: i.e., `*args` or `**kwargs` should be avoided, as they will not be correctly handled within cross-validation routines.

Next comes the fit method, where we handle training data:

```
    def fit(self, X, y):
        self.classes_ = np.sort(np.unique(y))
        training_sets = [X[y == yi] for yi in self.classes_]
        self.models_ = [KernelDensity(bandwidth=self.bandwidth,
                                      kernel=self.kernel).fit(Xi)
                        for Xi in training_sets]
        self.logpriors_ = [np.log(Xi.shape[0] / X.shape[0])
                           for Xi in training_sets]
        return self
```

Here we find the unique classes in the training data, train a KernelDensity model for each class, and compute the class priors based on the number of input samples. Finally, fit should always return self so that we can chain commands. For example:

```
label = model.fit(X, y).predict(X)
```

Notice that each persistent result of the fit is stored with a trailing underscore (e.g., self.logpriors_). This is a convention used in Scikit-Learn so that you can quickly scan the members of an estimator (using IPython's tab completion) and see exactly which members are fit to training data.

Finally, we have the logic for predicting labels on new data:

```
    def predict_proba(self, X):
        logprobs = np.vstack([model.score_samples(X)
                              for model in self.models_]).T
        result = np.exp(logprobs + self.logpriors_)
        return result / result.sum(axis=1, keepdims=True)

    def predict(self, X):
        return self.classes_[np.argmax(self.predict_proba(X), 1)]
```

Because this is a probabilistic classifier, we first implement predict_proba, which returns an array of class probabilities of shape [n_samples, n_classes]. Entry [i, j] of this array is the posterior probability that sample i is a member of class j, computed by multiplying the likelihood by the class prior and normalizing.

The predict method uses these probabilities and simply returns the class with the largest probability.

Using Our Custom Estimator

Let's try this custom estimator on a problem we have seen before: the classification of handwritten digits. Here we will load the digits and compute the cross-validation score for a range of candidate bandwidths using the `GridSearchCV` meta-estimator (refer back to Chapter 39):

```
In [14]: from sklearn.datasets import load_digits
         from sklearn.model_selection import GridSearchCV

         digits = load_digits()

         grid = GridSearchCV(KDEClassifier(),
                             {'bandwidth': np.logspace(0, 2, 100)})
         grid.fit(digits.data, digits.target);
```

Next we can plot the cross-validation score as a function of bandwidth (see Figure 49-7).

```
In [15]: fig, ax = plt.subplots()
         ax.semilogx(np.array(grid.cv_results_['param_bandwidth']),
                     grid.cv_results_['mean_test_score'])
         ax.set(title='KDE Model Performance', ylim=(0, 1),
                xlabel='bandwidth', ylabel='accuracy')
         print(f'best param: {grid.best_params_}')
         print(f'accuracy = {grid.best_score_}')
Out[15]: best param: {'bandwidth': 6.135907273413174}
         accuracy = 0.9677298050139276
```

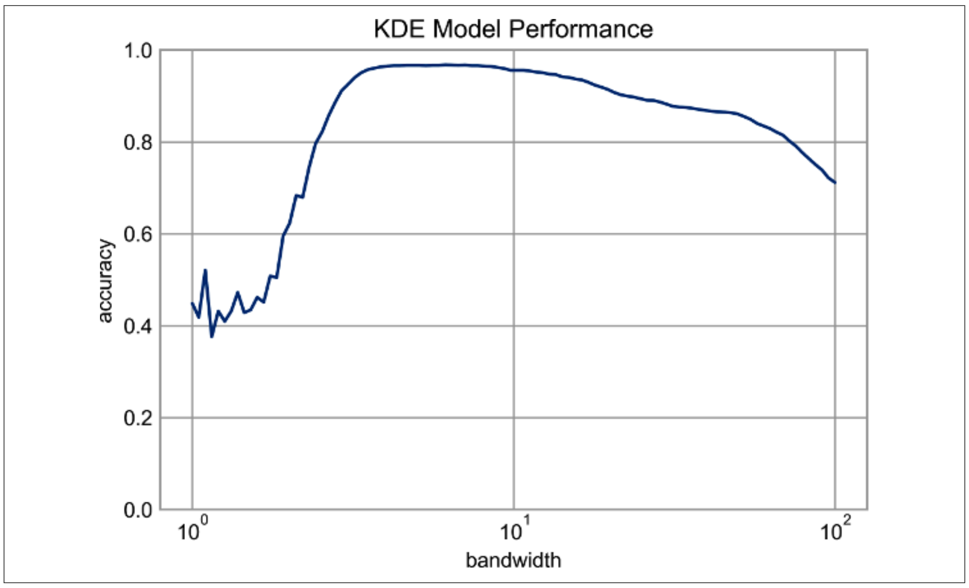

Figure 49-7. Validation curve for the KDE-based Bayesian classifier

This indicates that our KDE classifier reaches a cross-validation accuracy of over 96%, compared to around 80% for the naive Bayes classifier:

```
In [16]: from sklearn.naive_bayes import GaussianNB
         from sklearn.model_selection import cross_val_score
         cross_val_score(GaussianNB(), digits.data, digits.target).mean()
Out[16]: 0.8069281956050759
```

One benefit of such a generative classifier is interpretability of results: for each unknown sample, we not only get a probabilistic classification, but a *full model* of the distribution of points we are comparing it to! If desired, this offers an intuitive window into the reasons for a particular classification that algorithms like SVMs and random forests tend to obscure.

If you would like to take this further, here are some ideas for improvements that could be made to our KDE classifier model:

- You could allow the bandwidth in each class to vary independently.
- You could optimize these bandwidths not based on their prediction score, but on the likelihood of the training data under the generative model within each class (i.e. use the scores from `KernelDensity` itself rather than the global prediction accuracy).

Finally, if you want some practice building your own estimator, you might tackle building a similar Bayesian classifier using Gaussian mixture models instead of KDE.

CHAPTER 22

Network Analysis

Your connections to all the things around you literally define who you are.

—Aaron O'Connell

Many interesting data problems can be fruitfully thought of in terms of *networks*, consisting of *nodes* of some type and the *edges* that join them.

For instance, your Facebook friends form the nodes of a network whose edges are friendship relations. A less obvious example is the World Wide Web itself, with each web page a node and each hyperlink from one page to another an edge.

Facebook friendship is mutual—if I am Facebook friends with you, then necessarily you are friends with me. In this case, we say that the edges are *undirected*. Hyperlinks are not—my website links to *whitehouse.gov*, but (for reasons inexplicable to me) *whitehouse.gov* refuses to link to my website. We call these types of edges *directed*. We'll look at both kinds of networks.

Betweenness Centrality

In Chapter 1, we computed the key connectors in the DataSciencester network by counting the number of friends each user had. Now we have enough machinery to take a look at other approaches. We will use the same network, but now we'll use `NamedTuples` for the data.

Recall that the network (Figure 22-1) comprised users:

```
from typing import NamedTuple

class User(NamedTuple):
    id: int
    name: str
```

```
users = [User(0, "Hero"), User(1, "Dunn"), User(2, "Sue"), User(3, "Chi"),
         User(4, "Thor"), User(5, "Clive"), User(6, "Hicks"),
         User(7, "Devin"), User(8, "Kate"), User(9, "Klein")]
```

and friendships:

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

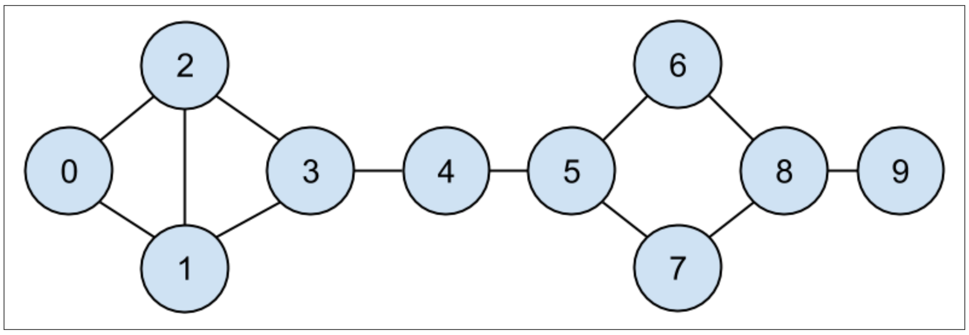

Figure 22-1. The DataSciencester network

The friendships will be easier to work with as a `dict`:

```
from typing import Dict, List

# type alias for keeping track of Friendships
Friendships = Dict[int, List[int]]

friendships: Friendships = {user.id: [] for user in users}

for i, j in friend_pairs:
    friendships[i].append(j)
    friendships[j].append(i)

assert friendships[4] == [3, 5]
assert friendships[8] == [6, 7, 9]
```

When we left off we were dissatisfied with our notion of *degree centrality*, which didn't really agree with our intuition about who the key connectors of the network were.

An alternative metric is *betweenness centrality*, which identifies people who frequently are on the shortest paths between pairs of other people. In particular, the betweenness centrality of node *i* is computed by adding up, for every other pair of nodes *j* and *k*, the proportion of shortest paths between node *j* and node *k* that pass through *i*.

That is, to figure out Thor's betweenness centrality, we'll need to compute all the shortest paths between all pairs of people who aren't Thor. And then we'll need to count how many of those shortest paths pass through Thor. For instance, the only shortest path between Chi (`id` 3) and Clive (`id` 5) passes through Thor, while neither of the two shortest paths between Hero (`id` 0) and Chi (`id` 3) does.

So, as a first step, we'll need to figure out the shortest paths between all pairs of people. There are some pretty sophisticated algorithms for doing so efficiently, but (as is almost always the case) we will use a less efficient, easier-to-understand algorithm.

This algorithm (an implementation of breadth-first search) is one of the more complicated ones in the book, so let's talk through it carefully:

1. Our goal is a function that takes a `from_user` and finds *all* shortest paths to every other user.
2. We'll represent a path as a `list` of user IDs. Since every path starts at `from_user`, we won't include her ID in the list. This means that the length of the list representing the path will be the length of the path itself.
3. We'll maintain a dictionary called `shortest_paths_to` where the keys are user IDs and the values are lists of paths that end at the user with the specified ID. If there is a unique shortest path, the list will just contain that one path. If there are multiple shortest paths, the list will contain all of them.
4. We'll also maintain a queue called `frontier` that contains the users we want to explore in the order we want to explore them. We'll store them as pairs `(prev_user, user)` so that we know how we got to each one. We initialize the queue with all the neighbors of `from_user`. (We haven't talked about queues, which are data structures optimized for "add to the end" and "remove from the front" operations. In Python, they are implemented as `collections.deque`, which is actually a double-ended queue.)
5. As we explore the graph, whenever we find new neighbors that we don't already know the shortest paths to, we add them to the end of the queue to explore later, with the current user as `prev_user`.
6. When we take a user off the queue, and we've never encountered that user before, we've definitely found one or more shortest paths to him—each of the shortest paths to `prev_user` with one extra step added.
7. When we take a user off the queue and we *have* encountered that user before, then either we've found another shortest path (in which case we should add it) or we've found a longer path (in which case we shouldn't).
8. When no more users are left on the queue, we've explored the whole graph (or, at least, the parts of it that are reachable from the starting user) and we're done.

We can put this all together into a (large) function:

```python
from collections import deque
```

```
Path = List[int]

def shortest_paths_from(from_user_id: int,
                        friendships: Friendships) -> Dict[int, List[Path]]:
    # A dictionary from user_id to *all* shortest paths to that user.
    shortest_paths_to: Dict[int, List[Path]] = {from_user_id: [[]]}

    # A queue of (previous user, next user) that we need to check.
    # Starts out with all pairs (from_user, friend_of_from_user).
    frontier = deque((from_user_id, friend_id)
                     for friend_id in friendships[from_user_id])

    # Keep going until we empty the queue.
    while frontier:
        # Remove the pair that's next in the queue.
        prev_user_id, user_id = frontier.popleft()

        # Because of the way we're adding to the queue,
        # necessarily we already know some shortest paths to prev_user.
        paths_to_prev_user = shortest_paths_to[prev_user_id]
        new_paths_to_user = [path + [user_id] for path in paths_to_prev_user]

        # It's possible we already know a shortest path to user_id.
        old_paths_to_user = shortest_paths_to.get(user_id, [])

        # What's the shortest path to here that we've seen so far?
        if old_paths_to_user:
            min_path_length = len(old_paths_to_user[0])
        else:
            min_path_length = float('inf')

        # Only keep paths that aren't too long and are actually new.
        new_paths_to_user = [path
                             for path in new_paths_to_user
                             if len(path) <= min_path_length
                             and path not in old_paths_to_user]

        shortest_paths_to[user_id] = old_paths_to_user + new_paths_to_user

        # Add never-seen neighbors to the frontier.
        frontier.extend((user_id, friend_id)
                        for friend_id in friendships[user_id]
                        if friend_id not in shortest_paths_to)

    return shortest_paths_to
```

Now let's compute all the shortest paths:

```
# For each from_user, for each to_user, a list of shortest paths.
shortest_paths = {user.id: shortest_paths_from(user.id, friendships)
                  for user in users}
```

And we're finally ready to compute betweenness centrality. For every pair of nodes *i* and *j*, we know the *n* shortest paths from *i* to *j*. Then, for each of those paths, we just add 1/n to the centrality of each node on that path:

```
betweenness_centrality = {user.id: 0.0 for user in users}

for source in users:
    for target_id, paths in shortest_paths[source.id].items():
        if source.id < target_id:      # don't double count
            num_paths = len(paths)     # how many shortest paths?
            contrib = 1 / num_paths    # contribution to centrality
            for path in paths:
                for between_id in path:
                    if between_id not in [source.id, target_id]:
                        betweenness_centrality[between_id] += contrib
```

As shown in Figure 22-2, users 0 and 9 have centrality 0 (as neither is on any shortest path between other users), whereas 3, 4, and 5 all have high centralities (as all three lie on many shortest paths).

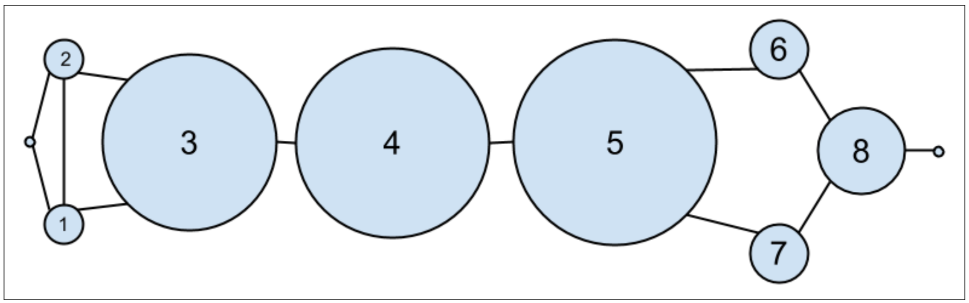

Figure 22-2. The DataSciencester network sized by betweenness centrality

Generally the centrality numbers aren't that meaningful themselves. What we care about is how the numbers for each node compare to the numbers for other nodes.

Another measure we can look at is *closeness centrality*. First, for each user we compute her *farness*, which is the sum of the lengths of her shortest paths to each other user. Since we've already computed the shortest paths between each pair of nodes, it's easy to add their lengths. (If there are multiple shortest paths, they all have the same length, so we can just look at the first one.)

```
def farness(user_id: int) -> float:
    """the sum of the lengths of the shortest paths to each other user"""
    return sum(len(paths[0])
               for paths in shortest_paths[user_id].values())
```

after which it's very little work to compute closeness centrality (Figure 22-3):

```
closeness_centrality = {user.id: 1 / farness(user.id) for user in users}
```

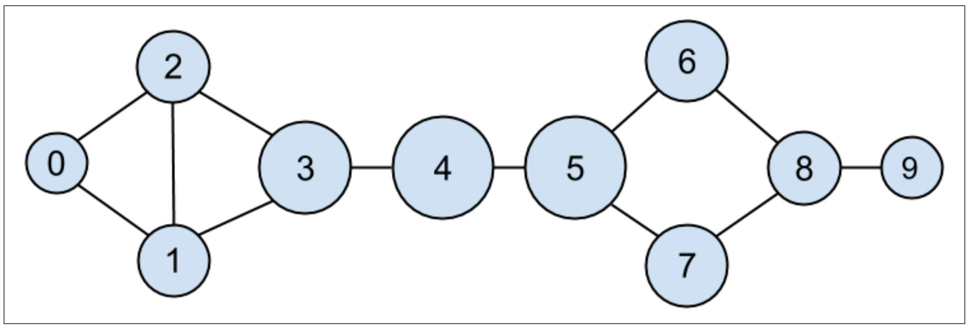

Figure 22-3. The DataSciencester network sized by closeness centrality

There is much less variation here—even the very central nodes are still pretty far from the nodes out on the periphery.

As we saw, computing shortest paths is kind of a pain. For this reason, betweenness and closeness centrality aren't often used on large networks. The less intuitive (but generally easier to compute) *eigenvector centrality* is more frequently used.

Eigenvector Centrality

In order to talk about eigenvector centrality, we have to talk about eigenvectors, and in order to talk about eigenvectors, we have to talk about matrix multiplication.

Matrix Multiplication

If A is an $n \times m$ matrix and B is an $m \times k$ matrix (notice that the second dimension of A is same as the first dimension of B), then their product AB is the $n \times k$ matrix whose (i,j)th entry is:

$$A_{i1}B_{1j} + A_{i2}B_{2j} + \cdots + A_{im}B_{mj}$$

which is just the dot product of the ith row of A (thought of as a vector) with the jth column of B (also thought of as a vector).

We can implement this using the `make_matrix` function from Chapter 4:

```
from scratch.linear_algebra import Matrix, make_matrix, shape

def matrix_times_matrix(m1: Matrix, m2: Matrix) -> Matrix:
    nr1, nc1 = shape(m1)
    nr2, nc2 = shape(m2)
    assert nc1 == nr2, "must have (# of columns in m1) == (# of rows in m2)"

    def entry_fn(i: int, j: int) -> float:
        """dot product of i-th row of m1 with j-th column of m2"""
        return sum(m1[i][k] * m2[k][j] for k in range(nc1))

    return make_matrix(nr1, nc2, entry_fn)
```

If we think of an *m*-dimensional vector as an `(m, 1)` matrix, we can multiply it by an `(n, m)` matrix to get an `(n, 1)` matrix, which we can then think of as an *n*-dimensional vector.

This means another way to think about an `(n, m)` matrix is as a linear mapping that transforms *m*-dimensional vectors into *n*-dimensional vectors:

```
from scratch.linear_algebra import Vector, dot

def matrix_times_vector(m: Matrix, v: Vector) -> Vector:
    nr, nc = shape(m)
    n = len(v)
    assert nc == n, "must have (# of cols in m) == (# of elements in v)"

    return [dot(row, v) for row in m]  # output has length nr
```

When *A* is a *square* matrix, this operation maps *n*-dimensional vectors to other *n*-dimensional vectors. It's possible that, for some matrix *A* and vector *v*, when *A* operates on *v* we get back a scalar multiple of *v*—that is, that the result is a vector that points in the same direction as *v*. When this happens (and when, in addition, *v* is not a vector of all zeros), we call *v* an *eigenvector* of *A*. And we call the multiplier an *eigenvalue*.

One possible way to find an eigenvector of *A* is by picking a starting vector *v*, applying `matrix_times_vector`, rescaling the result to have magnitude 1, and repeating until the process converges:

```
from typing import Tuple
import random
from scratch.linear_algebra import magnitude, distance

def find_eigenvector(m: Matrix,
                     tolerance: float = 0.00001) -> Tuple[Vector, float]:
    guess = [random.random() for _ in m]

    while True:
        result = matrix_times_vector(m, guess)    # transform guess
        norm = magnitude(result)                  # compute norm
        next_guess = [x / norm for x in result]   # rescale

        if distance(guess, next_guess) < tolerance:
            # convergence so return (eigenvector, eigenvalue)
            return next_guess, norm
            guess = next_guess
```

By construction, the returned `guess` is a vector such that, when you apply `matrix_times_vector` to it and rescale it to have length 1, you get back a vector very close to itself—which means it's an eigenvector.

Not all matrices of real numbers have eigenvectors and eigenvalues. For example, the matrix:

```
rotate = [[ 0, 1],
          [-1, 0]]
```

rotates vectors 90 degrees clockwise, which means that the only vector it maps to a scalar multiple of itself is a vector of zeros. If you tried `find_eigenvector(rotate)` it would run forever. Even matrices that have eigenvectors can sometimes get stuck in cycles. Consider the matrix:

```
flip = [[0, 1],
        [1, 0]]
```

This matrix maps any vector `[x, y]` to `[y, x]`. This means that, for example, `[1, 1]` is an eigenvector with eigenvalue 1. However, if you start with a random vector with unequal coordinates, `find_eigenvector` will just repeatedly swap the coordinates forever. (Not-from-scratch libraries like NumPy use different methods that would work in this case.) Nonetheless, when `find_eigenvector` does return a result, that result is indeed an eigenvector.

Centrality

How does this help us understand the DataSciencester network? To start, we'll need to represent the connections in our network as an `adjacency_matrix`, whose (*i*,*j*)th entry is either 1 (if user *i* and user *j* are friends) or 0 (if they're not):

```
def entry_fn(i: int, j: int):
    return 1 if (i, j) in friend_pairs or (j, i) in friend_pairs else 0

n = len(users)
adjacency_matrix = make_matrix(n, n, entry_fn)
```

The eigenvector centrality for each user is then the entry corresponding to that user in the eigenvector returned by `find_eigenvector` (Figure 22-4).

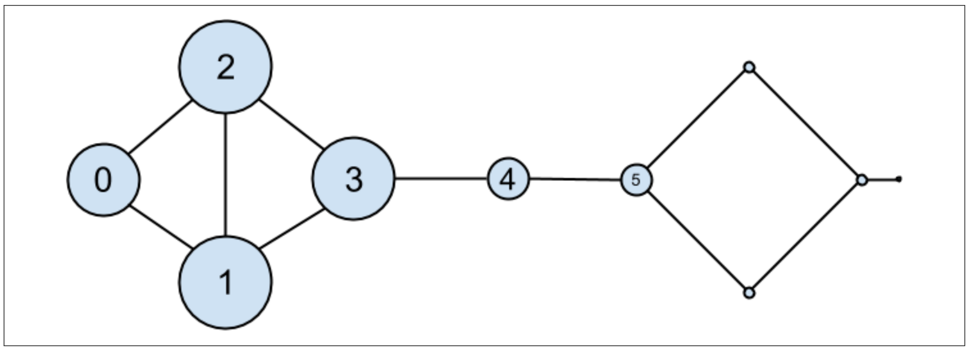

Figure 22-4. The DataSciencester network sized by eigenvector centrality

For technical reasons that are way beyond the scope of this book, any nonzero adjacency matrix necessarily has an eigenvector, all of whose values are nonnegative. And fortunately for us, for this `adjacency_matrix` our `find_eigenvector` function finds it.

```
eigenvector_centralities, _ = find_eigenvector(adjacency_matrix)
```

Users with high eigenvector centrality should be those who have a lot of connections, and connections to people who themselves have high centrality.

Here users 1 and 2 are the most central, as they both have three connections to people who are themselves highly central. As we move away from them, people's centralities steadily drop off.

On a network this small, eigenvector centrality behaves somewhat erratically. If you try adding or subtracting links, you'll find that small changes in the network can dramatically change the centrality numbers. In a much larger network, this would not particularly be the case.

We still haven't motivated why an eigenvector might lead to a reasonable notion of centrality. Being an eigenvector means that if you compute:

```
matrix_times_vector(adjacency_matrix, eigenvector_centralities)
```

the result is a scalar multiple of `eigenvector_centralities`.

If you look at how matrix multiplication works, `matrix_times_vector` produces a vector whose *i*th element is:

```
dot(adjacency_matrix[i], eigenvector_centralities)
```

which is precisely the sum of the eigenvector centralities of the users connected to user *i*.

In other words, eigenvector centralities are numbers, one per user, such that each user's value is a constant multiple of the sum of his neighbors' values. In this case centrality means being connected to people who themselves are central. The more centrality you are directly connected to, the more central you are. This is of course a circular definition—eigenvectors are the way of breaking out of the circularity.

Another way of understanding this is by thinking about what `find_eigenvector` is doing here. It starts by assigning each node a random centrality. It then repeats the following two steps until the process converges:

1. Give each node a new centrality score that equals the sum of its neighbors' (old) centrality scores.
2. Rescale the vector of centralities to have magnitude 1.

Although the mathematics behind it may seem somewhat opaque at first, the calculation itself is relatively straightforward (unlike, say, betweenness centrality) and is pretty easy to perform on even very large graphs. (At least, if you use a real linear algebra library it's easy to perform on large graphs. If you used our matrices-as-lists implementation you'd struggle.)

Directed Graphs and PageRank

DataSciencester isn't getting much traction, so the VP of Revenue considers pivoting from a friendship model to an endorsement model. It turns out that no one particularly cares which data scientists are *friends* with one another, but tech recruiters care very much which data scientists are *respected* by other data scientists.

In this new model, we'll track endorsements `(source, target)` that no longer represent a reciprocal relationship, but rather that `source` endorses `target` as an awesome data scientist (Figure 22-5).

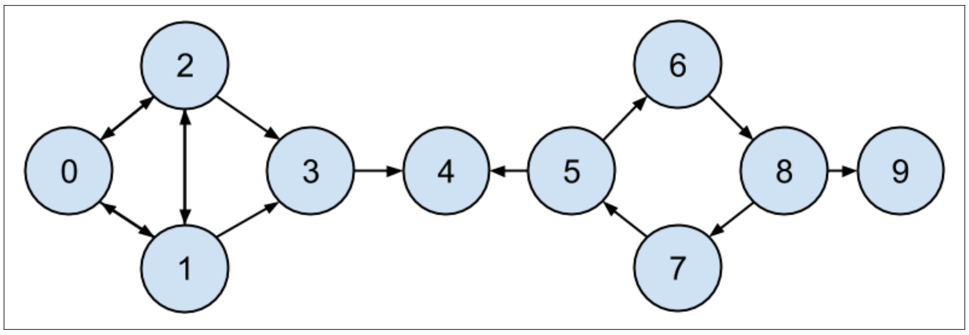

Figure 22-5. The DataSciencester network of endorsements

We'll need to account for this asymmetry:

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

after which we can easily find the `most_endorsed` data scientists and sell that information to recruiters:

```
from collections import Counter

endorsement_counts = Counter(target for source, target in endorsements)
```

However, "number of endorsements" is an easy metric to game. All you need to do is create phony accounts and have them endorse you. Or arrange with your friends to endorse each other. (As users 0, 1, and 2 seem to have done.)

A better metric would take into account *who* endorses you. Endorsements from people who have a lot of endorsements should somehow count more than endorsements from people with few endorsements. This is the essence of the PageRank algorithm, used by Google to rank websites based on which other websites link to them, which other websites link to those, and so on.

(If this sort of reminds you of the idea behind eigenvector centrality, it should.)

A simplified version looks like this:

1. There is a total of 1.0 (or 100%) PageRank in the network.
2. Initially this PageRank is equally distributed among nodes.
3. At each step, a large fraction of each node's PageRank is distributed evenly among its outgoing links.
4. At each step, the remainder of each node's PageRank is distributed evenly among all nodes.

```
import tqdm

def page_rank(users: List[User],
              endorsements: List[Tuple[int, int]],
              damping: float = 0.85,
              num_iters: int = 100) -> Dict[int, float]:
    # Compute how many people each person endorses
    outgoing_counts = Counter(target for source, target in endorsements)

    # Initially distribute PageRank evenly
    num_users = len(users)
    pr = {user.id : 1 / num_users for user in users}

    # Small fraction of PageRank that each node gets each iteration
    base_pr = (1 - damping) / num_users

    for iter in tqdm.trange(num_iters):
        next_pr = {user.id : base_pr for user in users}  # start with base_pr

        for source, target in endorsements:
            # Add damped fraction of source pr to target
            next_pr[target] += damping * pr[source] / outgoing_counts[source]

        pr = next_pr

    return pr
```

If we compute page ranks:

```
pr = page_rank(users, endorsements)

# Thor (user_id 4) has higher page rank than anyone else
assert pr[4] > max(page_rank
                   for user_id, page_rank in pr.items()
                   if user_id != 4)
```

PageRank (Figure 22-6) identifies user 4 (Thor) as the highest-ranked data scientist.

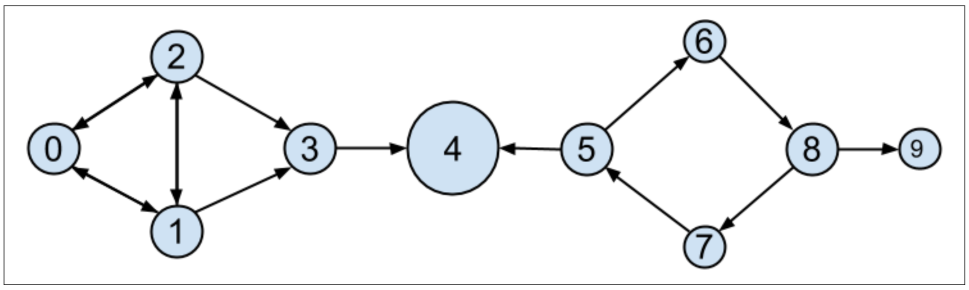

Figure 22-6. The DataSciencester network sized by PageRank

Even though Thor has fewer endorsements (two) than users 0, 1, and 2, his endorsements carry with them rank from their endorsements. Additionally, both of his endorsers endorsed only him, which means that he doesn't have to divide their rank with anyone else.

For Further Exploration

- There are many other notions of centrality besides the ones we used (although the ones we used are pretty much the most popular ones).
- NetworkX is a Python library for network analysis. It has functions for computing centralities and for visualizing graphs.
- Gephi is a love-it/hate-it GUI-based network visualization tool.