Chapter 8. Dimensionality Reduction

Many machine learning problems involve thousands or even millions of features for each training instance. Not only do all these features make training extremely slow, but they can also make it much harder to find a good solution, as you will see. This problem is often referred to as the *curse of dimensionality*.

Fortunately, in real-world problems, it is often possible to reduce the number of features considerably, turning an intractable problem into a tractable one. For example, consider the MNIST images (introduced in Chapter 3): the pixels on the image borders are almost always white, so you could completely drop these pixels from the training set without losing much information. As we saw in the previous chapter, (Figure 7-6) confirms that these pixels are utterly unimportant for the classification task. Additionally, two neighboring pixels are often highly correlated: if you merge them into a single pixel (e.g., by taking the mean of the two pixel intensities), you will not lose much information.

WARNING

Reducing dimensionality does cause some information loss, just like compressing an image to JPEG can degrade its quality, so even though it will speed up training, it may make your system perform slightly worse. It also makes your pipelines a bit more complex and thus harder to maintain. Therefore, I recommend you first try to train your system with the original data before considering using dimensionality reduction. In some cases, reducing the dimensionality of the training data may filter out some noise and unnecessary details and thus result in higher performance, but in general it won't; it will just speed up training.

Apart from speeding up training, dimensionality reduction is also extremely useful for data visualization. Reducing the number of dimensions down to two (or three) makes it possible to plot a condensed view of a high-dimensional training set on a graph and often gain some important insights by visually detecting patterns, such as clusters. Moreover, data visualization is essential to communicate your conclusions to people who are not data scientists—in particular, decision makers who will use your results.

In this chapter we will first discuss the curse of dimensionality and get a sense of what goes on in high-dimensional space. Then we will consider the two main approaches to dimensionality reduction (projection and manifold learning), and we will go through three of the most popular dimensionality reduction techniques: PCA, random projection, and locally linear embedding (LLE).

The Curse of Dimensionality

We are so used to living in three dimensions[1] that our intuition fails us when we try to imagine a high-dimensional space. Even a basic 4D hypercube is incredibly hard to picture in our minds (see Figure 8-1), let alone a 200-dimensional ellipsoid bent in a 1,000-dimensional space.
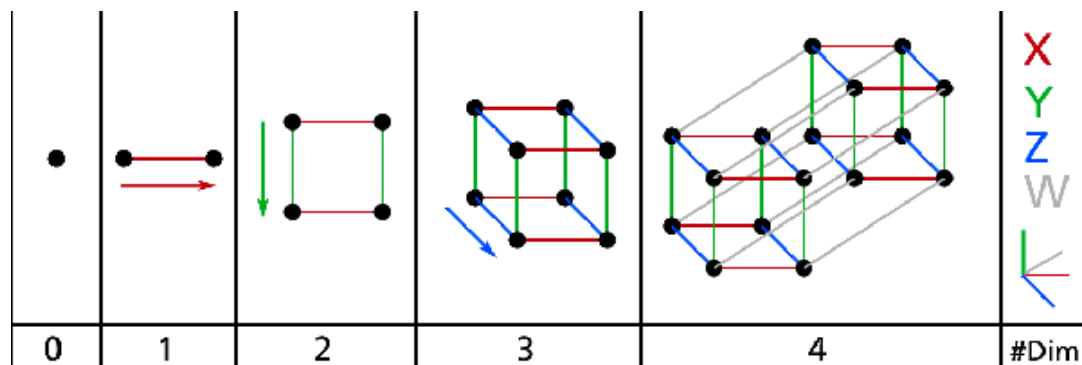

Figure 8-1. Point, segment, square, cube, and tesseract (0D to 4D hypercubes)[2]

It turns out that many things behave very differently in high-dimensional space. For example, if you pick a random point in a unit square (a 1 × 1 square), it will have only about a 0.4% chance of being located less than 0.001 from a border (in other words, it is very unlikely that a random point will be "extreme" along any dimension). But in a 10,000-dimensional unit hypercube, this probability is greater than 99.999999%. Most points in a high-dimensional hypercube are very close to the border.[3]

Here is a more troublesome difference: if you pick two points randomly in a unit square, the distance between these two points will be, on average, roughly 0.52. If you pick two random points in a 3D unit cube, the average distance will be roughly 0.66. But what about two points picked randomly in a 1,000,000-dimensional unit hypercube? The average distance, believe it or not, will be about 408.25 (roughly $\sqrt{1,000,000/6}$)! This is counterintuitive: how can two points be so far apart when they both lie within the same unit hypercube? Well, there's just plenty of space in high dimensions. As a result, high-dimensional datasets are at risk of being very sparse: most training instances are likely to be far away from each

other. This also means that a new instance will likely be far away from any training instance, making predictions much less reliable than in lower dimensions, since they will be based on much larger extrapolations. In short, the more dimensions the training set has, the greater the risk of overfitting it.

In theory, one solution to the curse of dimensionality could be to increase the size of the training set to reach a sufficient density of training instances. Unfortunately, in practice, the number of training instances required to reach a given density grows exponentially with the number of dimensions. With just 100 features—significantly fewer than in the MNIST problem—all ranging from 0 to 1, you would need more training instances than atoms in the observable universe in order for training instances to be within 0.1 of each other on average, assuming they were spread out uniformly across all dimensions.

Main Approaches for Dimensionality Reduction

Before we dive into specific dimensionality reduction algorithms, let's take a look at the two main approaches to reducing dimensionality: projection and manifold learning.

Projection

In most real-world problems, training instances are *not* spread out uniformly across all dimensions. Many features are almost constant, while others are highly correlated (as discussed earlier for MNIST). As a result, all training instances lie within (or close to) a much lower-dimensional *subspace* of the high-dimensional space. This sounds very abstract, so let's look at an example. In Figure 8-2 you can see a 3D dataset represented by small spheres.
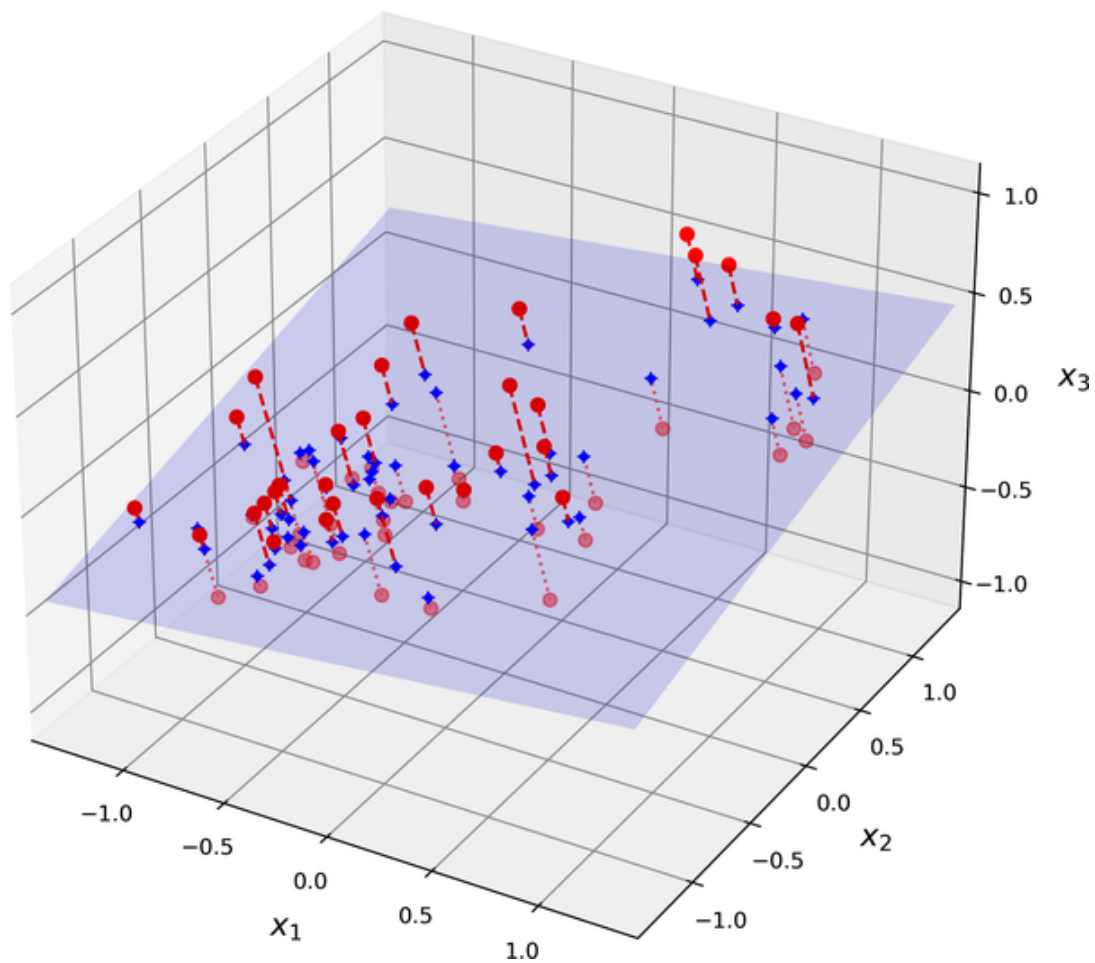
Figure 8-2. A 3D dataset lying close to a 2D subspace

Notice that all training instances lie close to a plane: this is a lower-dimensional (2D) subspace of the higher-dimensional (3D) space. If we project every training instance perpendicularly onto this subspace (as represented by the short dashed lines connecting the instances to the plane), we get the new 2D dataset shown in Figure 8-3. Ta-da! We have just reduced the dataset's dimensionality from 3D to 2D. Note that the axes correspond to new features z_1 and z_2: they are the coordinates of the projections on the plane.
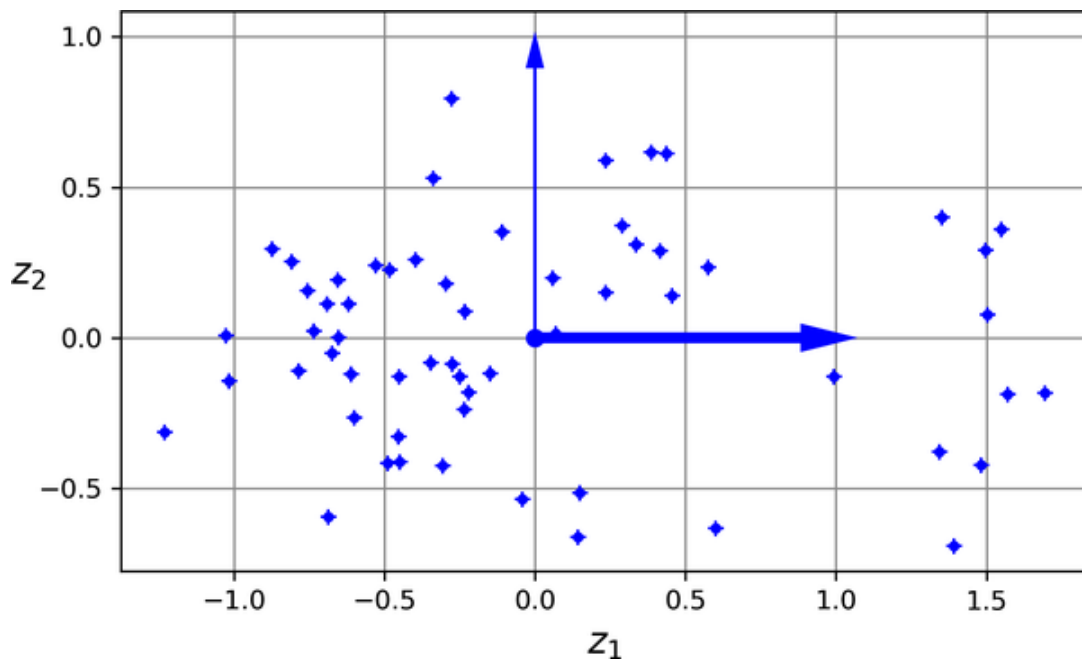
Figure 8-3. The new 2D dataset after projection

Manifold Learning

However, projection is not always the best approach to dimensionality re-
duction. In many cases the subspace may twist and turn, such as in the fa-
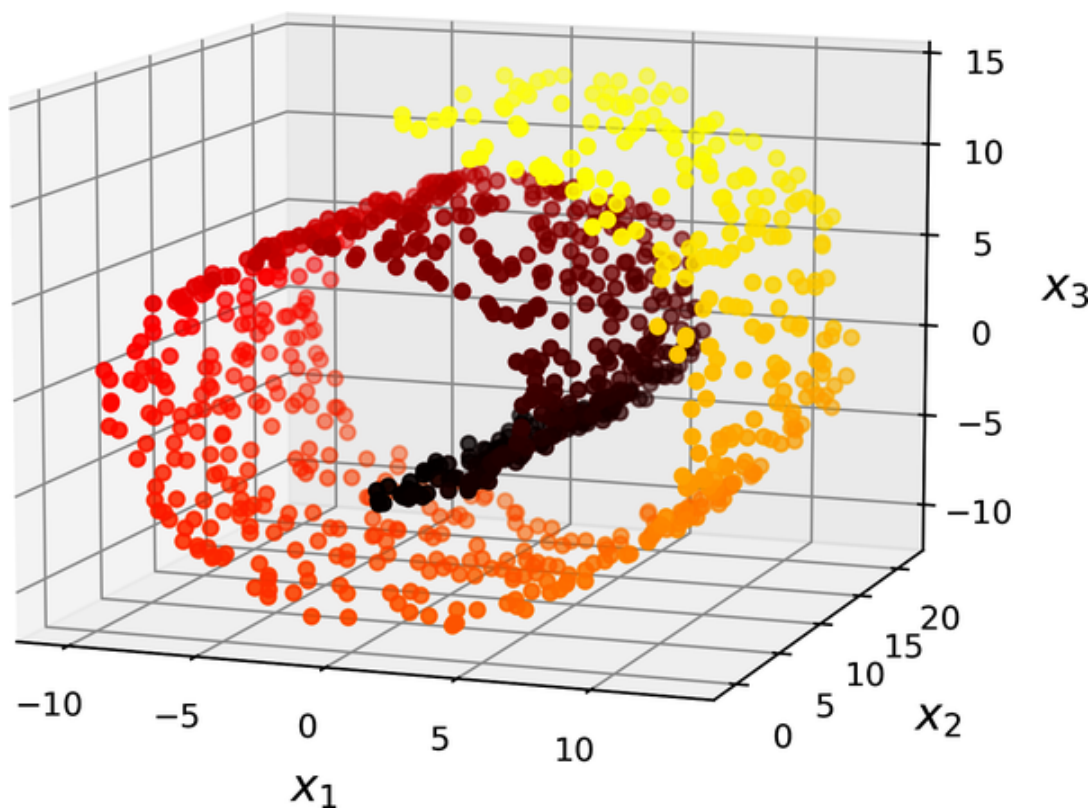mous Swiss roll toy dataset represented in Figure 8-4.

Figure 8-4. Swiss roll dataset

Simply projecting onto a plane (e.g., by dropping x_3) would squash differ-
ent layers of the Swiss roll together, as shown on the left side of Figure 8-
5. What you probably want instead is to unroll the Swiss roll to obtain the
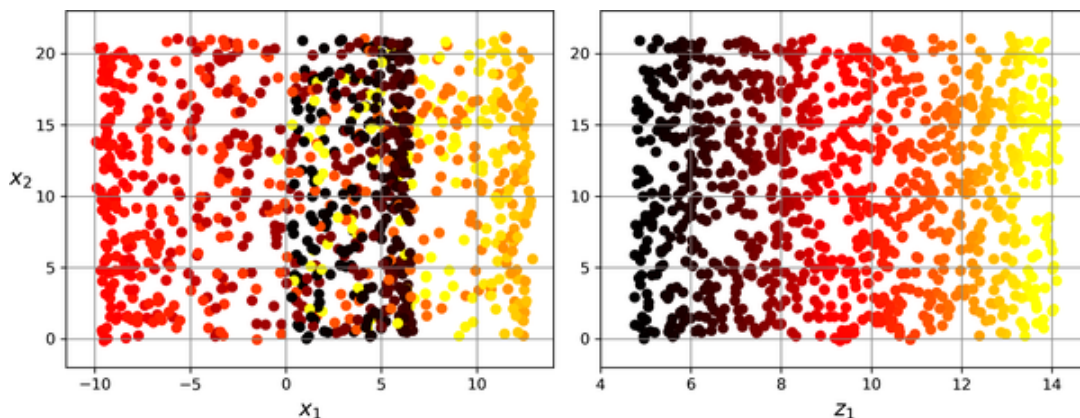2D dataset on the right side of Figure 8-5.

Figure 8-5. Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

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

Many dimensionality reduction algorithms work by modeling the manifold on which the training instances lie; this is called *manifold learning*. It relies on the *manifold assumption*, also called the *manifold hypothesis*, which holds that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold. This assumption is very often empirically observed.

Once again, think about the MNIST dataset: all handwritten digit images have some similarities. They are made of connected lines, the borders are white, and they are more or less centered. If you randomly generated images, only a ridiculously tiny fraction of them would look like handwritten digits. In other words, the degrees of freedom available to you if you try to create a digit image are dramatically lower than the degrees of freedom you have if you are allowed to generate any image you want. These constraints tend to squeeze the dataset into a lower-dimensional manifold.

The manifold assumption is often accompanied by another implicit assumption: that the task at hand (e.g., classification or regression) will be simpler if expressed in the lower-dimensional space of the manifold. For example, in the top row of Figure 8-6 the Swiss roll is split into two classes: in the 3D space (on the left) the decision boundary would be fairly complex, but in the 2D unrolled manifold space (on the right) the decision boundary is a straight line.

However, this implicit assumption does not always hold. For example, in the bottom row of [Figure 8-6](#), the decision boundary is located at $x_1 = 5$. This decision boundary looks very simple in the original 3D space (a vertical plane), but it looks more complex in the unrolled manifold (a collection of four independent line segments).

In short, reducing the dimensionality of your training set before training a model will usually speed up training, but it may not always lead to a better or simpler solution; it all depends on the dataset.

Hopefully you now have a good sense of what the curse of dimensionality is and how dimensionality reduction algorithms can fight it, especially when the manifold assumption holds. The rest of this chapter will go through some of the most popular algorithms for dimensionality reduction.
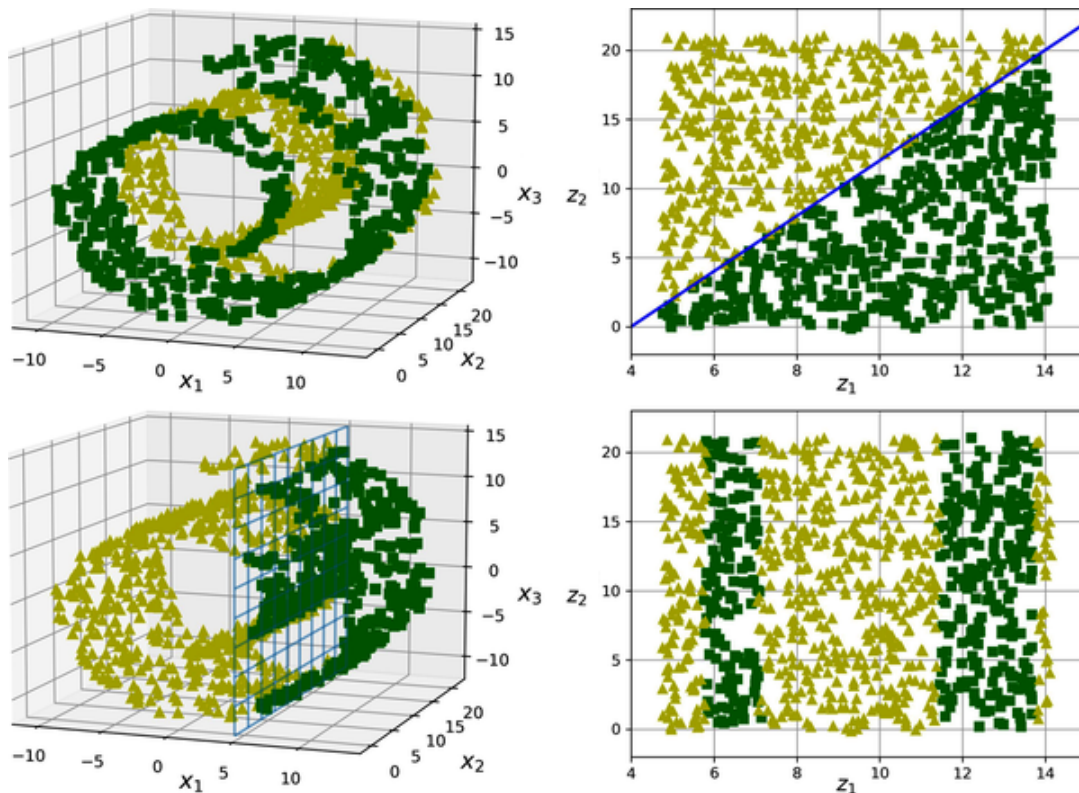

Figure 8-6. The decision boundary may not always be simpler with lower dimensions

PCA

Principal component analysis (PCA) is by far the most popular dimensionality reduction algorithm. First it identifies the hyperplane that lies closest to the data, and then it projects the data onto it, just like in [Figure 8-2](#).

Preserving the Variance

Before you can project the training set onto a lower-dimensional hyper-plane, you first need to choose the right hyperplane. For example, a simple 2D dataset is represented on the left in Figure 8-7, along with three different axes (i.e., 1D hyperplanes). On the right is the result of the projection of the dataset onto each of these axes. As you can see, the projection onto the solid line preserves the maximum variance (top), while the projection onto the dotted line preserves very little variance (bottom) and the projection onto the dashed line preserves an intermediate amount of variance (middle).
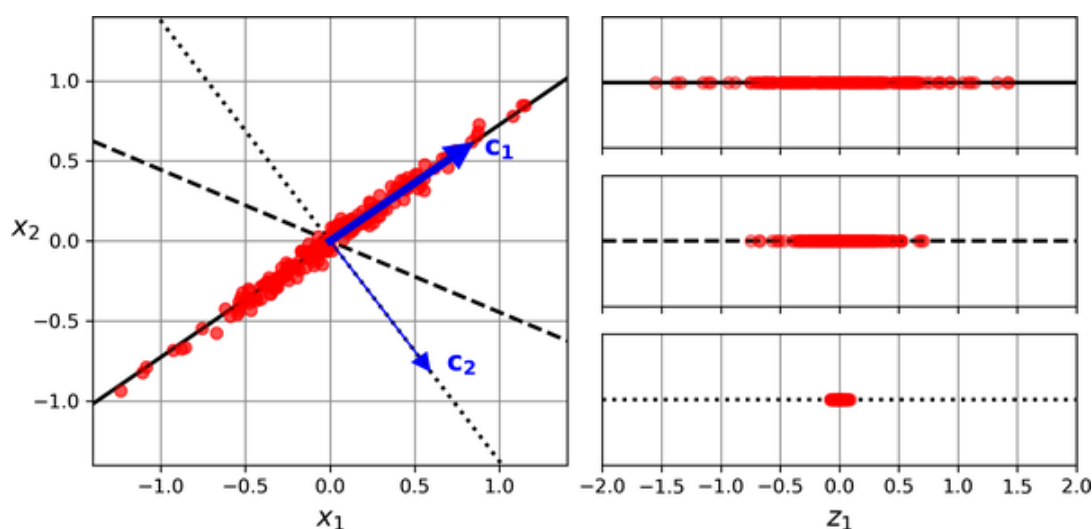

Figure 8-7. Selecting the subspace on which to project

It seems reasonable to select the axis that preserves the maximum amount of variance, as it will most likely lose less information than the other projections. Another way to justify this choice is that it is the axis that minimizes the mean squared distance between the original dataset and its projection onto that axis. This is the rather simple idea behind PCA.[4]

Principal Components

PCA identifies the axis that accounts for the largest amount of variance in the training set. In Figure 8-7, it is the solid line. It also finds a second axis, orthogonal to the first one, that accounts for the largest amount of the remaining variance. In this 2D example there is no choice: it is the dotted line. If it were a higher-dimensional dataset, PCA would also find a third axis, orthogonal to both previous axes, and a fourth, a fifth, and so on—as many axes as the number of dimensions in the dataset.

The ith axis is called the ith *principal component* (PC) of the data. In Figure 8-7, the first PC is the axis on which vector c_1 lies, and the second

PC is the axis on which vector c_2 lies. In [Figure 8-2](#) the first two PCs are on the projection plane, and the third PC is the axis orthogonal to that plane. After the projection, in [Figure 8-3](#), the first PC corresponds to the z_1 axis, and the second PC corresponds to the z_2 axis.

NOTE

For each principal component, PCA finds a zero-centered unit vector pointing in the direction of the PC. Since two opposing unit vectors lie on the same axis, the direction of the unit vectors returned by PCA is not stable: if you perturb the training set slightly and run PCA again, the unit vectors may point in the opposite direction as the original vectors. However, they will generally still lie on the same axes. In some cases, a pair of unit vectors may even rotate or swap (if the variances along these two axes are very close), but the plane they define will generally remain the same.

So how can you find the principal components of a training set? Luckily, there is a standard matrix factorization technique called *singular value decomposition* (SVD) that can decompose the training set matrix $\mathbf{X}$ into the matrix multiplication of three matrices $\mathbf{U}\ \Sigma\ \mathbf{V}^{\mathsf{T}}$, where $\mathbf{V}$ contains the unit vectors that define all the principal components that you are looking for, as shown in [Equation 8-1](#).

Equation 8-1. Principal components matrix

$$
\mathbf{V} = \begin{pmatrix} | & | & & | \\ \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_n \\ | & | & & | \end{pmatrix}
$$

The following Python code uses NumPy's `svd()` function to obtain all the principal components of the 3D training set represented in [Figure 8-2](#), then it extracts the two unit vectors that define the first two PCs:

```python
import numpy as np

X = [...]  # create a small 3D dataset
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt[0]
c2 = Vt[1]
```

Projecting Down to d Dimensions

Once you have identified all the principal components, you can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components. Selecting this hyperplane ensures that the projection will preserve as much variance as possible. For example, in [Figure 8-2](#) the 3D dataset is projected down to the 2D plane defined by the first two principal components, preserving a large part of the dataset's variance. As a result, the 2D projection looks very much like the original 3D dataset.

To project the training set onto the hyperplane and obtain a reduced dataset $\mathbf{X}_{d\text{-proj}}$ of dimensionality d, compute the matrix multiplication of the training set matrix $\mathbf{X}$ by the matrix $\mathbf{W}_d$, defined as the matrix containing the first d columns of $\mathbf{V}$, as shown in [Equation 8-2](#).

Equation 8-2. Projecting the training set down to d dimensions

$$\mathbf{X}_{d\text{-proj}} = \mathbf{X}\mathbf{W}_d$$

The following Python code projects the training set onto the plane defined by the first two principal components:

```
W2 = Vt[:2].T
X2D = X_centered @ W2
```

There you have it! You now know how to reduce the dimensionality of any dataset by projecting it down to any number of dimensions, while preserving as much variance as possible.

Using Scikit-Learn

Scikit-Learn's `PCA` class uses SVD to implement PCA, just like we did earlier in this chapter. The following code applies PCA to reduce the dimen-

sionality of the dataset down to two dimensions (note that it automatically takes care of centering the data):

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X2D = pca.fit_transform(X)
```

After fitting the `PCA` transformer to the dataset, its `components_` attribute holds the transpose of $\mathbf{W}_d$: it contains one row for each of the first d principal components.

Explained Variance Ratio

Another useful piece of information is the *explained variance ratio* of each principal component, available via the `explained_variance_ratio_` variable. The ratio indicates the proportion of the dataset's variance that lies along each principal component. For example, let's look at the explained variance ratios of the first two components of the 3D dataset represented in [Figure 8-2](#):

```
>>> pca.explained_variance_ratio_
array([0.7578477 , 0.15186921])
```

This output tells us that about 76% of the dataset's variance lies along the first PC, and about 15% lies along the second PC. This leaves about 9% for the third PC, so it is reasonable to assume that the third PC probably carries little information.

Choosing the Right Number of Dimensions

Instead of arbitrarily choosing the number of dimensions to reduce down to, it is simpler to choose the number of dimensions that add up to a sufficiently large portion of the variance—say, 95% (An exception to this rule, of course, is if you are reducing dimensionality for data visualization, in which case you will want to reduce the dimensionality down to 2 or 3).

The following code loads and splits the MNIST dataset (introduced in [Chapter 3](#)) and performs PCA without reducing dimensionality, then com-

putes the minimum number of dimensions required to preserve 95% of the training set's variance:

```python
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', as_frame=False)
X_train, y_train = mnist.data[:60_000], mnist.target[:60_000]
X_test, y_test = mnist.data[60_000:], mnist.target[60_000:]

pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1  # d equals 154
```

You could then set `n_components=d` and run PCA again, but there's a better option. Instead of specifying the number of principal components you want to preserve, you can set `n_components` to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve:

```python
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

The actual number of components is determined during training, and it is stored in the `n_components_` attribute:

```python
>>> pca.n_components_
154
```

Yet another option is to plot the explained variance as a function of the number of dimensions (simply plot `cumsum`; see Figure 8-8). There will usually be an elbow in the curve, where the explained variance stops growing fast. In this case, you can see that reducing the dimensionality down to about 100 dimensions wouldn't lose too much explained variance.
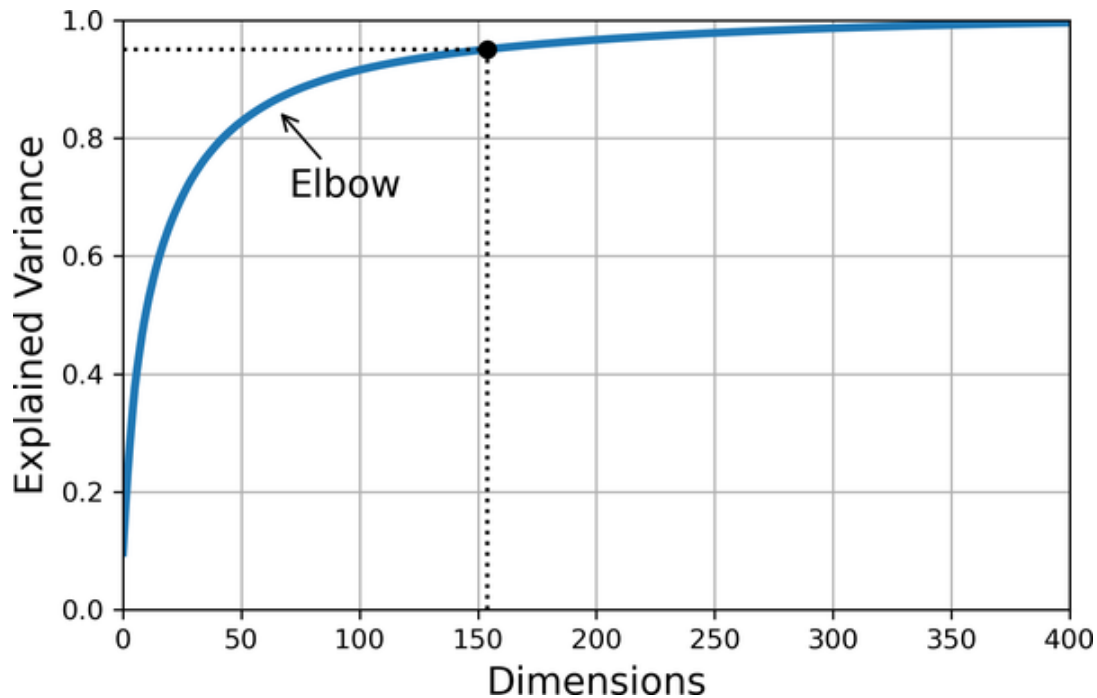
Figure 8-8. Explained variance as a function of the number of dimensions

Lastly, if you are using dimensionality reduction as a preprocessing step for a supervised learning task (e.g., classification), then you can tune the number of dimensions as you would any other hyperparameter (see Chapter 2). For example, the following code example creates a two-step pipeline, first reducing dimensionality using PCA, then classifying using a random forest. Next, it uses `RandomizedSearchCV` to find a good combination of hyperparameters for both PCA and the random forest classifier. This example does a quick search, tuning only 2 hyperparameters, training on just 1,000 instances, and running for just 10 iterations, but feel free to do a more thorough search if you have the time:

```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.pipeline import make_pipeline

clf = make_pipeline(PCA(random_state=42),
                    RandomForestClassifier(random_state=42))
param_distrib = {
    "pca__n_components": np.arange(10, 80),
    "randomforestclassifier__n_estimators": np.arange(50, 500)
}
rnd_search = RandomizedSearchCV(clf, param_distrib, n_iter=10, cv=3,
                                random_state=42)
rnd_search.fit(X_train[:1000], y_train[:1000])
```

Let's look at the best hyperparameters found:

```
>>> print(rnd_search.best_params_)
{'randomforestclassifier__n_estimators': 465, 'pca__n_components': 23}
```

It's interesting to note how low the optimal number of components is: we reduced a 784-dimensional dataset to just 23 dimensions! This is tied to the fact that we used a random forest, which is a pretty powerful model. If we used a linear model instead, such as an `SGDClassifier`, the search would find that we need to preserve more dimensions (about 70).

PCA for Compression

After dimensionality reduction, the training set takes up much less space. For example, after applying PCA to the MNIST dataset while preserving 95% of its variance, we are left with 154 features, instead of the original 784 features. So the dataset is now less than 20% of its original size, and we only lost 5% of its variance! This is a reasonable compression ratio, and it's easy to see how such a size reduction would speed up a classification algorithm tremendously.

It is also possible to decompress the reduced dataset back to 784 dimensions by applying the inverse transformation of the PCA projection. This won't give you back the original data, since the projection lost a bit of information (within the 5% variance that was dropped), but it will likely be close to the original data. The mean squared distance between the original data and the reconstructed data (compressed and then decompressed) is called the *reconstruction error*.

The `inverse_transform()` method lets us decompress the reduced MNIST dataset back to 784 dimensions:

```
X_recovered = pca.inverse_transform(X_reduced)
```

Figure 8-9 shows a few digits from the original training set (on the left), and the corresponding digits after compression and decompression. You can see that there is a slight image quality loss, but the digits are still mostly intact.
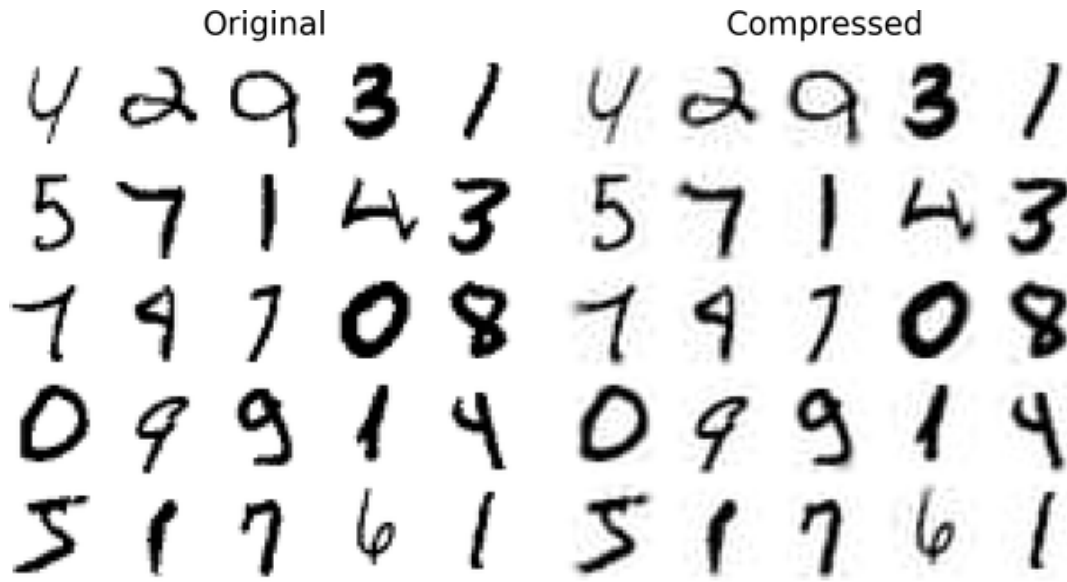
Figure 8-9. MNIST compression that preserves 95% of the variance

The equation for the inverse transformation is shown in [Equation 8-3](#).

Equation 8-3. PCA inverse transformation, back to the original number of dimensions

$$\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d\text{-proj}}\mathbf{W}_d{}^{\intercal}$$

Randomized PCA

If you set the `svd_solver` hyperparameter to `"randomized"`, Scikit-Learn uses a stochastic algorithm called *randomized PCA* that quickly finds an approximation of the first d principal components. Its computational complexity is $O(m \times d^2) + O(d^3)$, instead of $O(m \times n^2) + O(n^3)$ for the full SVD approach, so it is dramatically faster than full SVD when d is much smaller than n:

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

TIP

By default, `svd_solver` is actually set to `"auto"`: Scikit-Learn automatically uses the randomized PCA algorithm if max(m, n) > 500 and `n_components` is an integer smaller than 80% of min(m, n), or else it uses the full SVD approach. So the preceding code would use the randomized PCA algorithm even if you removed the `svd_solver="randomized"` argument, since 154 < 0.8 × 784. If you want to force Scikit-Learn to use full SVD for a slightly more precise result, you can set the `svd_solver` hyperparameter to `"full"`.

Incremental PCA

One problem with the preceding implementations of PCA is that they require the whole training set to fit in memory in order for the algorithm to run. Fortunately, *incremental PCA* (IPCA) algorithms have been developed that allow you to split the training set into mini-batches and feed these in one mini-batch at a time. This is useful for large training sets and for applying PCA online (i.e., on the fly, as new instances arrive).

The following code splits the MNIST training set into 100 mini-batches (using NumPy's `array_split()` function) and feeds them to Scikit-Learn's `IncrementalPCA` class[5] to reduce the dimensionality of the MNIST dataset down to 154 dimensions, just like before. Note that you must call the `partial_fit()` method with each mini-batch, rather than the `fit()` method with the whole training set:

```python
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

Alternatively, you can use NumPy's `memmap` class, which allows you to manipulate a large array stored in a binary file on disk as if it were entirely in memory; the class loads only the data it needs in memory, when it needs it. To demonstrate this, let's first create a memory-mapped (memmap) file and copy the MNIST training set to it, then call `flush()` to ensure that any data still in the cache gets saved to disk. In real life, `X_train` would typically not fit in memory, so you would load it chunk by chunk and save each chunk to the right part of the memmap array:

```python
filename = "my_mnist.mmap"
X_mmap = np.memmap(filename, dtype='float32', mode='write', shape=X_train.shap
X_mmap[:] = X_train  # could be a loop instead, saving the data chunk by chunk
X_mmap.flush()
```

Next, we can load the memmap file and use it like a regular NumPy array. Let's use the `IncrementalPCA` class to reduce its dimensionality. Since this algorithm uses only a small part of the array at any given time, mem-

ory usage remains under control. This makes it possible to call the usual `fit()` method instead of `partial_fit()`, which is quite convenient:

```
X_mmap = np.memmap(filename, dtype="float32", mode="readonly").reshape(-1, 784
batch_size = X_mmap.shape[0] // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mmap)
```

WARNING

Only the raw binary data is saved to disk, so you need to specify the data type and shape of the array when you load it. If you omit the shape, `np.memmap()` returns a 1D array.

For very high-dimensional datasets, PCA can be too slow. As you saw earlier, even if you use randomized PCA its computational complexity is still $O(m \times d^2) + O(d^3)$, so the target number of dimensions d must not be too large. If you are dealing with a dataset with tens of thousands of features or more (e.g., images), then training may become much too slow: in this case, you should consider using random projection instead.

Random Projection

As its name suggests, the random projection algorithm projects the data to a lower-dimensional space using a random linear projection. This may sound crazy, but it turns out that such a random projection is actually very likely to preserve distances fairly well, as was demonstrated mathematically by William B. Johnson and Joram Lindenstrauss in a famous lemma. So, two similar instances will remain similar after the projection, and two very different instances will remain very different.

Obviously, the more dimensions you drop, the more information is lost, and the more distances get distorted. So how can you choose the optimal number of dimensions? Well, Johnson and Lindenstrauss came up with an equation that determines the minimum number of dimensions to preserve in order to ensure—with high probability—that distances won't change by more than a given tolerance. For example, if you have a dataset containing m = 5,000 instances with n = 20,000 features each, and you don't want the squared distance between any two instances to

change by more than ε = 10%,[6] then you should project the data down to d dimensions, with $d \geq 4 \log(m) / (\frac{1}{2} \varepsilon^2 - \frac{1}{3} \varepsilon^3)$, which is 7,300 dimensions. That's quite a significant dimensionality reduction! Notice that the equation does not use n, it only relies on m and ε. This equation is implemented by the `johnson_lindenstrauss_min_dim()` function:

```
>>> from sklearn.random_projection import johnson_lindenstrauss_min_dim
>>> m, ε = 5_000, 0.1
>>> d = johnson_lindenstrauss_min_dim(m, eps=ε)
>>> d
7300
```

Now we can just generate a random matrix **P** of shape [d, n], where each item is sampled randomly from a Gaussian distribution with mean 0 and variance $1 / d$, and use it to project a dataset from n dimensions down to d:

```
n = 20_000
np.random.seed(42)
P = np.random.randn(d, n) / np.sqrt(d)  # std dev = square root of variance

X = np.random.randn(m, n)  # generate a fake dataset
X_reduced = X @ P.T
```

That's all there is to it! It's simple and efficient, and no training is required: the only thing the algorithm needs to create the random matrix is the dataset's shape. The data itself is not used at all.

Scikit-Learn offers a `GaussianRandomProjection` class to do exactly what we just did: when you call its `fit()` method, it uses `johnson_lindenstrauss_min_dim()` to determine the output dimensionality, then it generates a random matrix, which it stores in the `components_` attribute. Then when you call `transform()`, it uses this matrix to perform the projection. When creating the transformer, you can set `eps` if you want to tweak ε (it defaults to 0.1), and `n_components` if you want to force a specific target dimensionality d. The following code example gives the same result as the preceding code (you can also verify that `gaussian_rnd_proj.components_` is equal to `P`):

```
from sklearn.random_projection import GaussianRandomProjection
```

```
    gaussian_rnd_proj = GaussianRandomProjection(eps=ε, random_state=42)
    X_reduced = gaussian_rnd_proj.fit_transform(X)  # same result as above
```

Scikit-Learn also provides a second random projection transformer, known as `SparseRandomProjection`. It determines the target dimensionality in the same way, generates a random matrix of the same shape, and performs the projection identically. The main difference is that the random matrix is sparse. This means it uses much less memory: about 25 MB instead of almost 1.2 GB in the preceding example! And it's also much faster, both to generate the random matrix and to reduce dimensionality: about 50% faster in this case. Moreover, if the input is sparse, the transformation keeps it sparse (unless you set `dense_output=True`). Lastly, it enjoys the same distance-preserving property as the previous approach, and the quality of the dimensionality reduction is comparable. In short, it's usually preferable to use this transformer instead of the first one, especially for large or sparse datasets.

The ratio r of nonzero items in the sparse random matrix is called its *density*. By default, it is equal to $1/\sqrt{n}$. With 20,000 features, this means that only 1 in ~141 cells in the random matrix is nonzero: that's quite sparse! You can set the `density` hyperparameter to another value if you prefer. Each cell in the sparse random matrix has a probability r of being nonzero, and each nonzero value is either $-v$ or $+v$ (both equally likely), where $v = 1/\sqrt{dr}$.

If you want to perform the inverse transform, you first need to compute the pseudo-inverse of the components matrix using SciPy's `pinv()` function, then multiply the reduced data by the transpose of the pseudo-inverse:

```
    components_pinv = np.linalg.pinv(gaussian_rnd_proj.components_)
    X_recovered = X_reduced @ components_pinv.T
```

WARNING

Computing the pseudo-inverse may take a very long time if the components matrix is large, as the computational complexity of `pinv()` is $O(dn^2)$ if $d < n$, or $O(nd^2)$ otherwise.

In summary, random projection is a simple, fast, memory-efficient, and surprisingly powerful dimensionality reduction algorithm that you should keep in mind, especially when you deal with high-dimensional datasets.

LLE

Locally linear embedding (LLE)[8] is a *nonlinear dimensionality reduction* (NLDR) technique. It is a manifold learning technique that does not rely on projections, unlike PCA and random projection. In a nutshell, LLE works by first measuring how each training instance linearly relates to its nearest neighbors, and then looking for a low-dimensional representation of the training set where these local relationships are best preserved (more details shortly). This approach makes it particularly good at unrolling twisted manifolds, especially when there is not too much noise.

The following code makes a Swiss roll, then uses Scikit-Learn's `LocallyLinearEmbedding` class to unroll it:

```python
from sklearn.datasets import make_swiss_roll
from sklearn.manifold import LocallyLinearEmbedding

X_swiss, t = make_swiss_roll(n_samples=1000, noise=0.2, random_state=42)
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10, random_state=42)
X_unrolled = lle.fit_transform(X_swiss)
```

The variable `t` is a 1D NumPy array containing the position of each instance along the rolled axis of the Swiss roll. We don't use it in this example, but it can be used as a target for a nonlinear regression task.

The resulting 2D dataset is shown in Figure 8-10. As you can see, the Swiss roll is completely unrolled, and the distances between instances are locally well preserved. However, distances are not preserved on a larger scale: the unrolled Swiss roll should be a rectangle, not this kind of stretched and twisted band. Nevertheless, LLE did a pretty good job of modeling the manifold.
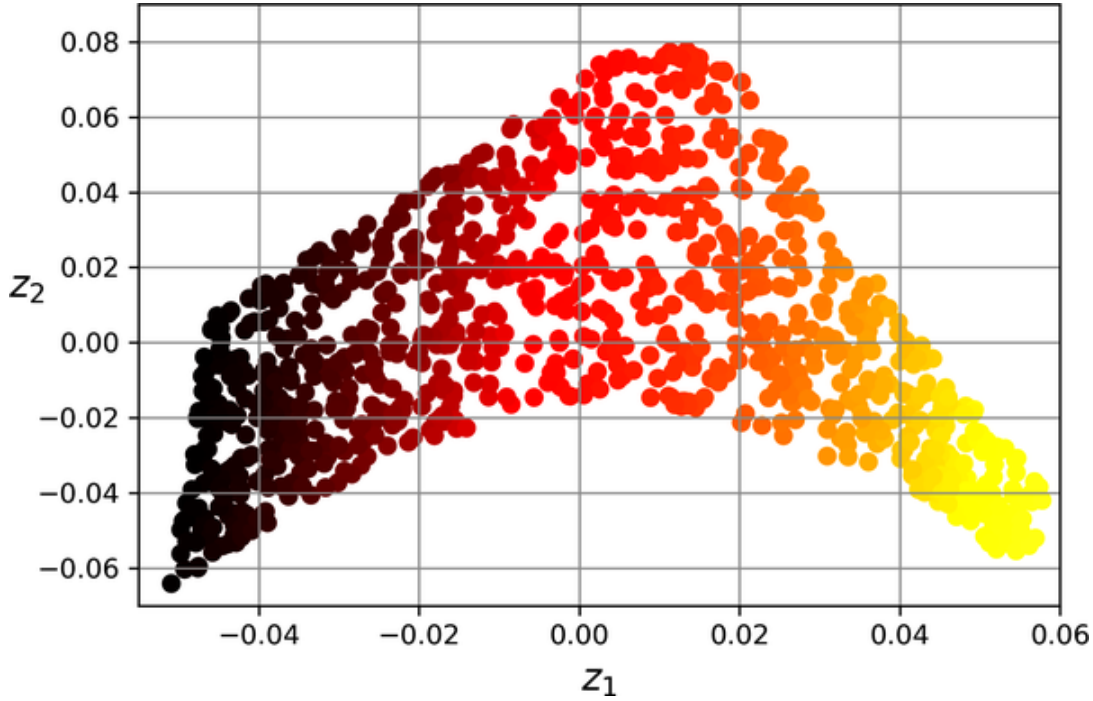

Figure 8-10. Unrolled Swiss roll using LLE

Here's how LLE works: for each training instance $\mathbf{x}^{(i)}$, the algorithm identifies its k-nearest neighbors (in the preceding code $k = 10$), then tries to reconstruct $\mathbf{x}^{(i)}$ as a linear function of these neighbors. More specifically, it tries to find the weights $w_{i,j}$ such that the squared distance between $\mathbf{x}^{(i)}$ and $\sum_{j=1}^{m} w_{i,j}\mathbf{x}^{(j)}$ is as small as possible, assuming $w_{i,j} = 0$ if $\mathbf{x}^{(j)}$ is not one of the k-nearest neighbors of $\mathbf{x}^{(i)}$. Thus the first step of LLE is the constrained optimization problem described in Equation 8-4, where $\mathbf{W}$ is the weight matrix containing all the weights $w_{i,j}$. The second constraint simply normalizes the weights for each training instance $\mathbf{x}^{(i)}$.

Equation 8-4. LLE step 1: linearly modeling local relationships

$$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^{m} \left(\mathbf{x}^{(i)} - \sum_{j=1}^{m} w_{i,j}\mathbf{x}^{(j)} \right)^2$$

$$\text{subject to} \begin{cases} w_{i,j} = 0 & \text{if } \mathbf{x}^{(j)} \text{ is not one of the } k \text{ n.n. of } \mathbf{x}^{(i)} \\ \sum_{j=1}^{m} w_{i,j} = 1 & \text{for } i = 1, 2, \cdots, m \end{cases}$$

After this step, the weight matrix $\widehat{\mathbf{W}}$ (containing the weights $\hat{w}_{i,j}$) encodes the local linear relationships between the training instances. The second step is to map the training instances into a d-dimensional space

(where $d < n$) while preserving these local relationships as much as possible. If $\mathbf{z}^{(i)}$ is the image of $\mathbf{x}^{(i)}$ in this d-dimensional space, then we want the squared distance between $\mathbf{z}^{(i)}$ and $\sum_{j=1}^{m} \widehat{w}_{i,j} \mathbf{z}^{(j)}$ to be as small as possible. This idea leads to the unconstrained optimization problem described in Equation 8-5. It looks very similar to the first step, but instead of keeping the instances fixed and finding the optimal weights, we are doing the reverse: keeping the weights fixed and finding the optimal position of the instances' images in the low-dimensional space. Note that $\mathbf{Z}$ is the matrix containing all $\mathbf{z}^{(i)}$.

Equation 8-5. LLE step 2: reducing dimensionality while preserving relationships

$$\widehat{\mathbf{Z}} = \underset{\mathbf{Z}}{\mathrm{argmin}} \sum_{i=1}^{m} \left(\mathbf{z}^{(i)} - \sum_{j=1}^{m} \widehat{w}_{i,j} \mathbf{z}^{(j)} \right)^2$$

Scikit-Learn's LLE implementation has the following computational complexity: $O(m \log(m) n \log(k))$ for finding the k-nearest neighbors, $O(mnk^3)$ for optimizing the weights, and $O(dm^2)$ for constructing the low-dimensional representations. Unfortunately, the m^2 in the last term makes this algorithm scale poorly to very large datasets.

As you can see, LLE is quite different from the projection techniques, and it's significantly more complex, but it can also construct much better low-dimensional representations, especially if the data is nonlinear.

Other Dimensionality Reduction Techniques

Before we conclude this chapter, let's take a quick look at a few other popular dimensionality reduction techniques available in Scikit-Learn:

`sklearn.manifold.MDS`

Multidimensional scaling (MDS) reduces dimensionality while trying to preserve the distances between the instances. Random projection does that for high-dimensional data, but it doesn't work well on low-dimensional data.

`sklearn.manifold.Isomap`

Isomap creates a graph by connecting each instance to its nearest neighbors, then reduces dimensionality while trying to preserve the *geodesic distances* between the instances. The geodesic distance between two nodes in a graph is the number of nodes on the shortest path between these nodes.

`sklearn.manifold.TSNE`

t-distributed stochastic neighbor embedding (t-SNE) reduces dimensionality while trying to keep similar instances close and dissimilar instances apart. It is mostly used for visualization, in particular to visualize clusters of instances in high-dimensional space. For example, in the exercises at the end of this chapter you will use t-SNE to visualize a 2D map of the MNIST images.

`sklearn.discriminant_analysis.LinearDiscriminantAnalysis`

Linear discriminant analysis (LDA) is a linear classification algorithm that, during training, learns the most discriminative axes between the classes. These axes can then be used to define a hyperplane onto which to project the data. The benefit of this approach is that the projection will keep classes as far apart as possible, so LDA is a good technique to reduce dimensionality before running another classification algorithm (unless LDA alone is sufficient).

Figure 8-11 shows the results of MDS, Isomap, and t-SNE on the Swiss roll. MDS manages to flatten the Swiss roll without losing its global curvature, while Isomap drops it entirely. Depending on the downstream task, preserving the large-scale structure may be good or bad. t-SNE does a reasonable job of flattening the Swiss roll, preserving a bit of curvature, and it also amplifies clusters, tearing the roll apart. Again, this might be good or bad, depending on the downstream task.
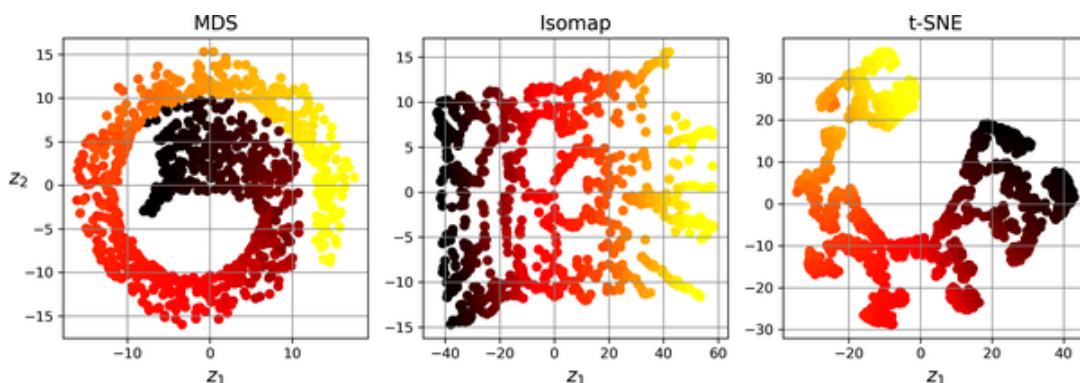

Figure 8-11. Using various techniques to reduce the Swiss roll to 2D

Exercises

1. What are the main motivations for reducing a dataset's dimensionality? What are the main drawbacks?
2. What is the curse of dimensionality?
3. Once a dataset's dimensionality has been reduced, is it possible to reverse the operation? If so, how? If not, why?
4. Can PCA be used to reduce the dimensionality of a highly nonlinear dataset?
5. Suppose you perform PCA on a 1,000-dimensional dataset, setting the explained variance ratio to 95%. How many dimensions will the resulting dataset have?
6. In what cases would you use regular PCA, incremental PCA, randomized PCA, or random projection?
7. How can you evaluate the performance of a dimensionality reduction algorithm on your dataset?
8. Does it make any sense to chain two different dimensionality reduction algorithms?
9. Load the MNIST dataset (introduced in Chapter 3) and split it into a training set and a test set (take the first 60,000 instances for training, and the remaining 10,000 for testing). Train a random forest classifier on the dataset and time how long it takes, then evaluate the resulting model on the test set. Next, use PCA to reduce the dataset's dimensionality, with an explained variance ratio of 95%. Train a new random forest classifier on the reduced dataset and see how long it takes. Was training much faster? Next, evaluate the classifier on the test set. How does it compare to the previous classifier? Try again with an `SGDClassifier`. How much does PCA help now?
10. Use t-SNE to reduce the first 5,000 images of the MNIST dataset down to 2 dimensions and plot the result using Matplotlib. You can use a scatterplot using 10 different colors to represent each image's target class. Alternatively, you can replace each dot in the scatterplot with the corresponding instance's class (a digit from 0 to 9), or even plot scaled-down versions of the digit images themselves (if you plot all digits the visualization will be too cluttered, so you should either draw a random sample or plot an instance only if no other instance has already been plotted at a close distance). You should get a nice visualization with well-separated clusters of digits. Try using other dimensionality reduction algorithms, such as PCA, LLE, or MDS, and compare the resulting visualizations.

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab3*.

1 Well, four dimensions if you count time, and a few more if you are a string theorist.

2 Watch a rotating tesseract projected into 3D space at *https://homl.info/30*. Image by Wikipedia user NerdBoy1392 (Creative Commons BY-SA 3.0). Reproduced from *https://en.wikipedia.org/wiki/Tesseract*.

3 Fun fact: anyone you know is probably an extremist in at least one dimension (e.g., how much sugar they put in their coffee), if you consider enough dimensions.

4 Karl Pearson, "On Lines and Planes of Closest Fit to Systems of Points in Space", *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2, no. 11 (1901): 559–572.

5 Scikit-Learn uses the algorithm described in David A. Ross et al., "Incremental Learning for Robust Visual Tracking", *International Journal of Computer Vision* 77, no. 1–3 (2008): 125–141.

6 ε is the Greek letter epsilon, often used for tiny values.

7 Sanjoy Dasgupta et al., "A neural algorithm for a fundamental computing problem", *Science* 358, no. 6364 (2017): 793–796.

8 Sam T. Roweis and Lawrence K. Saul, "Nonlinear Dimensionality Reduction by Locally Linear Embedding", *Science* 290, no. 5500 (2000): 2323–2326.