CHAPTER 8
Dimensionality Reduction

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles. The following will be Chapter 8 in the final release of the book.

Many Machine Learning problems involve thousands or even millions of features for each training instance. Not only does this make training extremely slow, it can also make it much harder to find a good solution, as we 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. Figure 7-6 confirms that these pixels are utterly unimportant for the classification task. Moreover, 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.

Reducing dimensionality does lose some information (just like compressing an image to JPEG can degrade its quality), so even though it will speed up training, it may also make your system perform slightly worse. It also makes your pipelines a bit more complex and thus harder to maintain. So you should first try to train your system with the original data before considering using dimensionality reduction if training is too slow. In some cases, however, 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 (or *DataViz*). 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, DataViz 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 discuss the curse of dimensionality and get a sense of what goes on in high-dimensional space. Then, we will present 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, Kernel PCA, and 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 mind (see Figure 8-1), let alone a 200-dimensional ellipsoid bent in a 1,000-dimensional space.

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

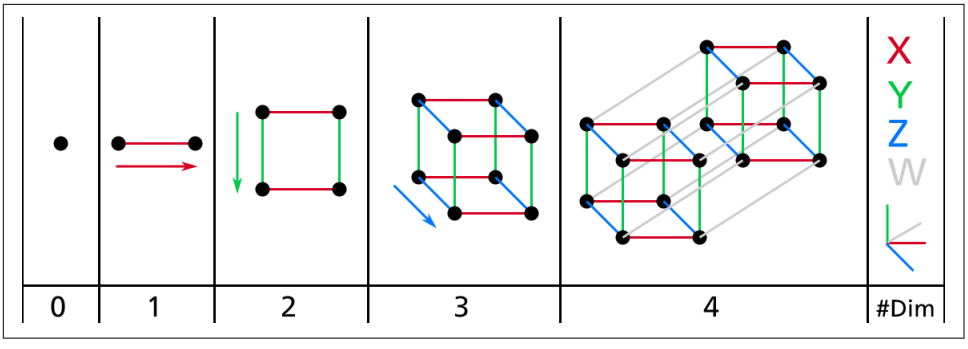

Figure 8-1. Point, segment, square, cube, and tesseract (0D to 4D hypercubes)[2]

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 (a $1 \times 1 \times \cdots \times 1$ cube, with ten thousand 1s), this probability is greater than 99.999999%. Most points in a high-dimensional hypercube are very close to the border.[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 unit 3D cube, the average distance will be roughly 0.66. But what about two points picked randomly in a 1,000,000-dimensional hypercube? Well, the average distance, believe it or not, will be about 408.25 (roughly $\sqrt{1,000,000/6}$)! This is quite counterintuitive: how can two points be so far apart when they both lie within the same unit hypercube? This fact implies that high-dimensional datasets are at risk of being very sparse: most training instances are likely to be far away from each other. Of course, 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 (much less than

2 Watch a rotating tesseract projected into 3D space at *https://homl.info/30*. Image by Wikipedia user Nerd-Boy1392 (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.

in the MNIST problem), you would need more training instances than atoms in the observable universe in order for training instances to be within 0.1 of each other on average, assuming they were spread out uniformly across all dimensions.

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 actually 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 the circles.

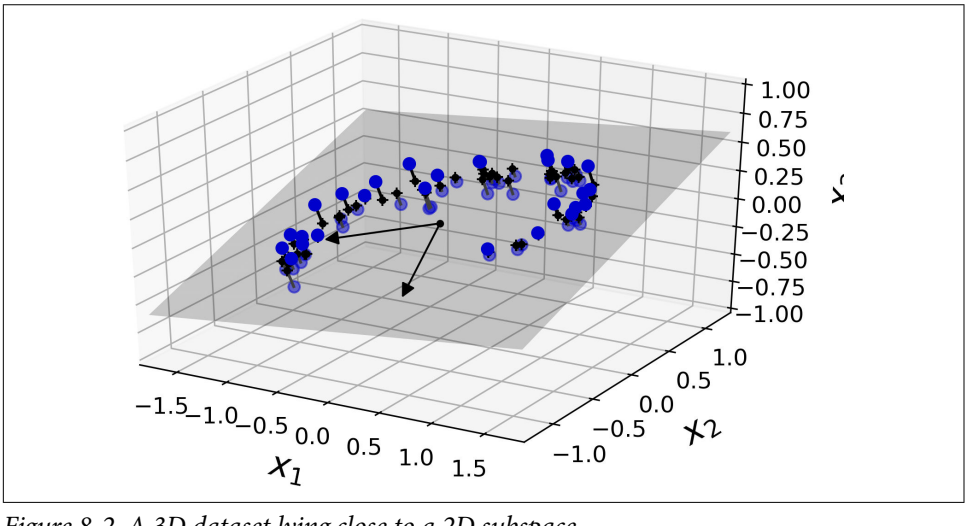

Figure 8-2. A 3D dataset lying close to a 2D subspace

Notice that all training instances lie close to a plane: this is a lower-dimensional (2D) subspace of the high-dimensional (3D) space. Now if we project every training instance perpendicularly onto this subspace (as represented by the short 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 (the coordinates of the projections on the plane).

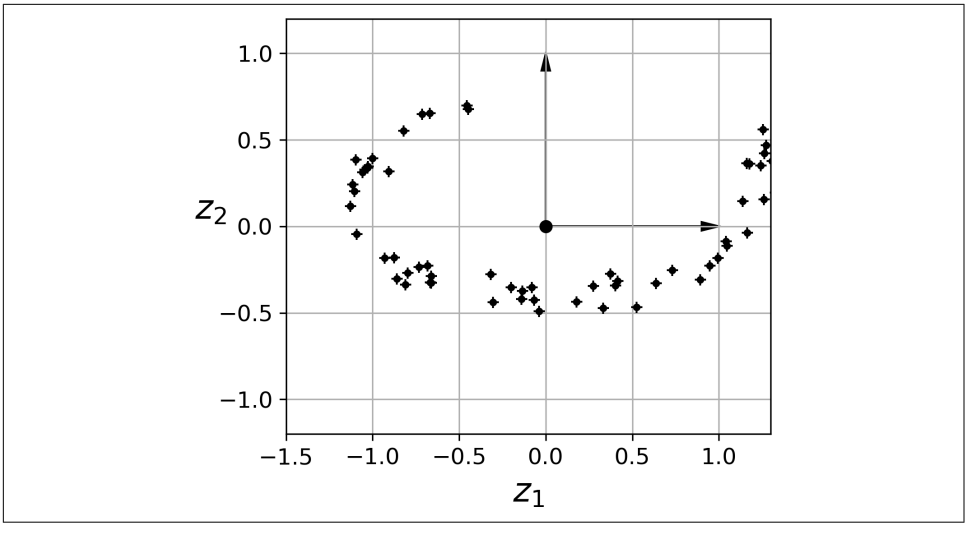

Figure 8-3. The new 2D dataset after projection

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

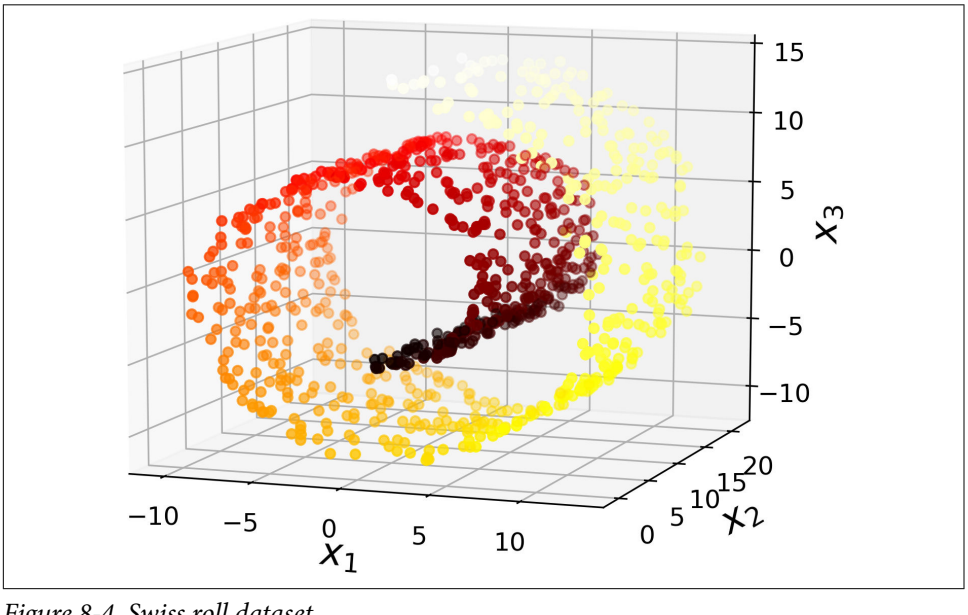

Figure 8-4. Swiss roll dataset

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

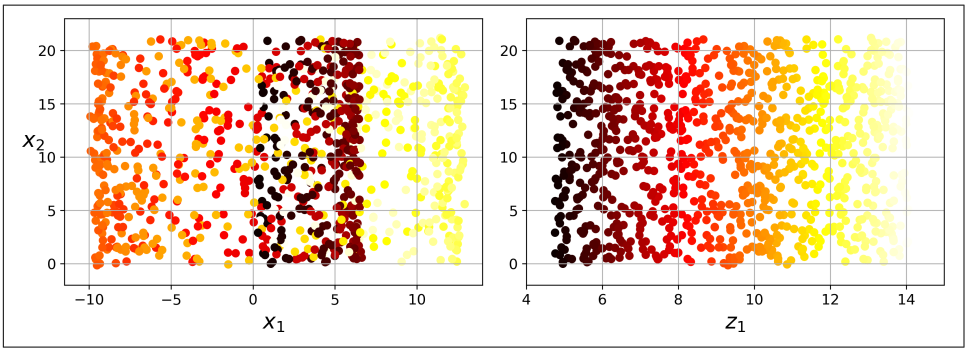

Figure 8-5. Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

Manifold Learning

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, they are more or less centered, and so on. 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 would have if you were allowed to generate any image you wanted. 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 simple straight line.

However, this 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, if you reduce the dimensionality of your training set before training a model, it 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.

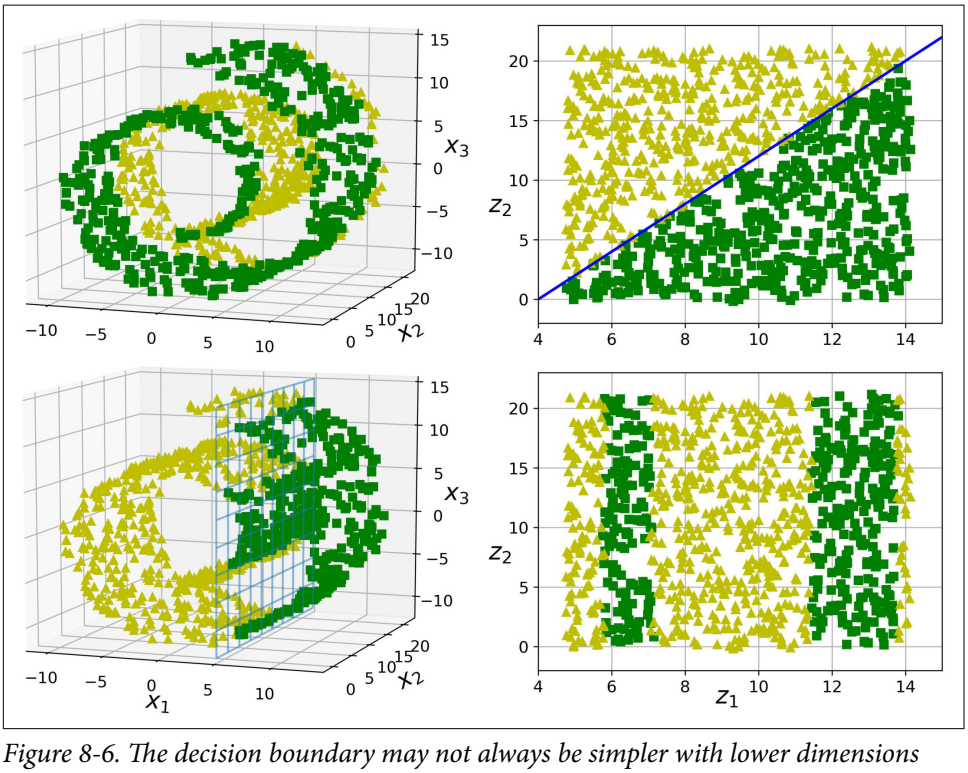

Figure 8-6. The decision boundary may not always be simpler with lower dimensions

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

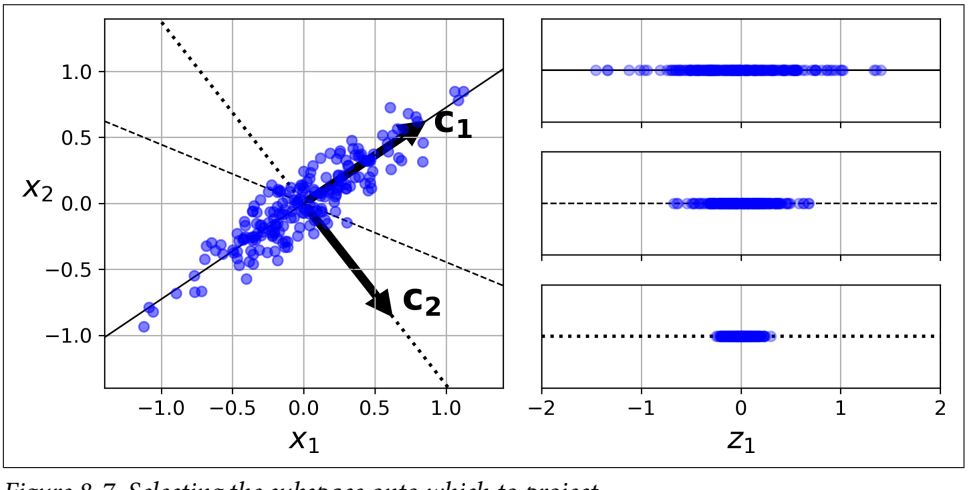

Figure 8-7. Selecting the subspace onto which to project

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]

4 "On Lines and Planes of Closest Fit to Systems of Points in Space," K. Pearson (1901).

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 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 unit vector that defines the i^{th} axis is called the i^{th} *principal component* (PC). In Figure 8-7, the 1^{st} PC is $\mathbf{c}_1$ and the 2^{nd} PC is $\mathbf{c}_2$. In Figure 8-2 the first two PCs are represented by the orthogonal arrows in the plane, and the third PC would be orthogonal to the plane (pointing up or down).

The direction of the principal components is not stable: if you perturb the training set slightly and run PCA again, some of the new PCs may point in the opposite direction of the original PCs. However, they will generally still lie on the same axes. In some cases, a pair of PCs may even rotate or swap, 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 $\mathbf{U}\ \Sigma\ \mathbf{V}^T$, where $\mathbf{V}$ contains all the principal components that we 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 training set, then extracts the first two PCs:

```
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
```

PCA assumes that the dataset is centered around the origin. As we will see, Scikit-Learn's PCA classes take care of centering the data for you. However, if you implement PCA yourself (as in the preceding example), or if you use other libraries, don't forget to center the data first.

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, you can simply 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 principal components (i.e., the matrix composed of 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.T[:, :2]
X2D = X_centered.dot(W2)
```

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

Using Scikit-Learn

Scikit-Learn's `PCA` class implements PCA using SVD decomposition just like we did before. The following code applies PCA to reduce the dimensionality 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, you can access the principal components using the `components_` variable (note that it contains the PCs as horizontal vec-

tors, so, for example, the first principal component is equal to `pca.components_.T[:, 0]`).

Explained Variance Ratio

Another very useful piece of information is the *explained variance ratio* of each principal component, available via the `explained_variance_ratio_` variable. It indicates the proportion of the dataset's variance that lies along the axis of 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([0.84248607, 0.14631839])
```

This tells you that 84.2% of the dataset's variance lies along the first axis, and 14.6% lies along the second axis. This leaves less than 1.2% for the third axis, so it is reasonable to assume that it probably carries little information.

Choosing the Right Number of Dimensions

Instead of arbitrarily choosing the number of dimensions to reduce down to, it is generally preferable to choose the number of dimensions that add up to a sufficiently large portion of the variance (e.g., 95%). Unless, of course, you are reducing dimensionality for data visualization—in that case you will generally want to reduce the dimensionality down to 2 or 3.

The following code computes PCA without reducing dimensionality, then computes the minimum number of dimensions required to preserve 95% of the training set's variance:

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

You could then set `n_components=d` and run PCA again. However, there is a much 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:

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

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. You can think of this as the intrinsic dimensionality of the dataset. In this case, you can see that reducing the

dimensionality down to about 100 dimensions wouldn't lose too much explained variance.

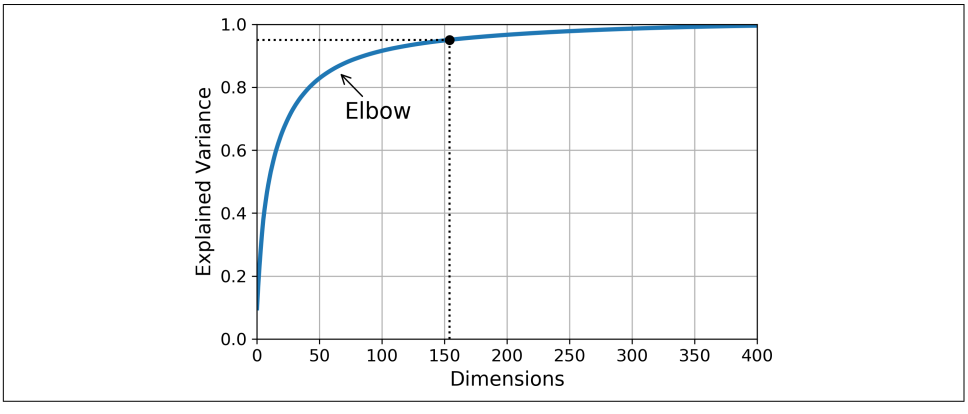

Figure 8-8. Explained variance as a function of the number of dimensions

PCA for Compression

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

It is also possible to decompress the reduced dataset back to 784 dimensions by applying the inverse transformation of the PCA projection. Of course 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 quite 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*. For example, the following code compresses the MNIST dataset down to 154 dimensions, then uses the `inverse_transform()` method to decompress it back to 784 dimensions. 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.

```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

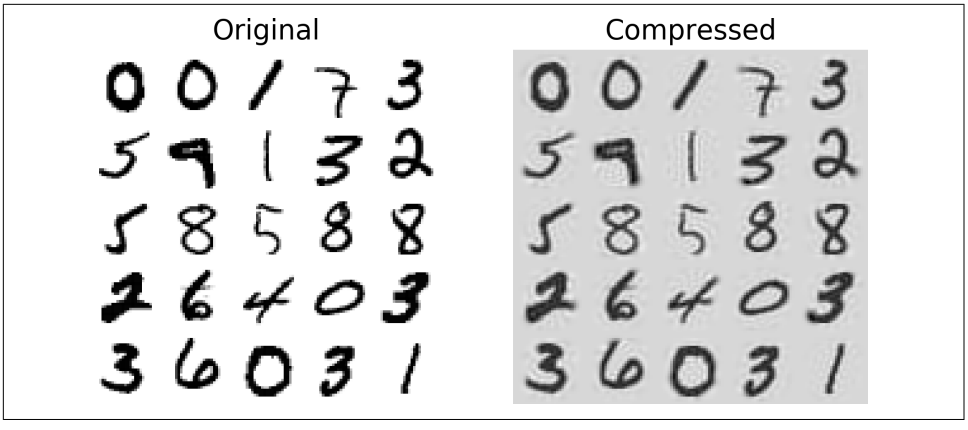

Figure 8-9. MNIST compression preserving 95% of the variance

The equation of the inverse transformation is shown in Equation 8-3.

> *Equation 8-3. PCA inverse transformation, back to the original number of dimensions*
>
> $$\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d\text{-proj}}\mathbf{W}_d^T$$

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")
X_reduced = rnd_pca.fit_transform(X_train)
```

By default, `svd_solver` is actually set to `"auto"`: Scikit-Learn automatically uses the randomized PCA algorithm if m or n is greater than 500 and d is less than 80% of m or n, or else it uses the full SVD approach. If you want to force Scikit-Learn to use full SVD, 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: you can split the training set into mini-batches and feed an IPCA algorithm one mini-batch at a time. This is

useful for large training sets, and also to apply PCA online (i.e., on the fly, as new instances arrive).

The following code splits the MNIST dataset into 100 mini-batches (using NumPy's `array_split()` function) and feeds them to Scikit-Learn's `IncrementalPCA` 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:

```
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. Since the `IncrementalPCA` class uses only a small part of the array at any given time, the memory usage remains under control. This makes it possible to call the usual `fit()` method, as you can see in the following code:

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

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

Kernel PCA

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

It turns out that the same trick can be applied to PCA, making it possible to perform complex nonlinear projections for dimensionality reduction. This is called *Kernel*

5 Scikit-Learn uses the algorithm described in "Incremental Learning for Robust Visual Tracking," D. Ross et al. (2007).

13 t-Distributed Stochastic Neighbor Embedding (t-SNE)

13.1 EXPLANATION AND WORKING

We have discussed various non-linear dimensionality reduction techniques that preserve the local structure of the data in a low dimensional space. While these algorithms very well preserve the local geometry of the data, they do not retain both the local as well as the global structure of the data in a single mapping onto the low dimensional space. In this chapter, we discuss a relatively new dimensionality reduction technique called t-SNE which captures the local structure of the original data and at the same time, reveals the global structure at different scales [1]. t-SNE is a variant of Stochastic Neighbor Embedding (SNE) [2] which addresses some of the pitfalls of SNE. Firstly, let us discuss the SNE algorithm which forms the basis for t-SNE.

13.1.1 Stochastic Neighbor Embedding (SNE)

SNE converts distances between data points to conditional probabilities. For any two data points x_i and x_j in high dimensional space d, the conditional probability $p_{j|i}$ is a measure of how likely it is for x_i to choose x_j as its neighbor if the neighbors were to be chosen based on their probability density under a Gaussian centered at x_i. This is as if, for any data point x_i, we put a Gaussian centered at x_i, and based on the density of this Gaussian we decide if other points are its neighbors or not. If two data points x_i and x_j are close in the original space, then $P_{j|i}$ will be relatively high, whereas if they are far apart, $p_{j|i}$ will be low. This conditional probability is given by

$$p_{j|i} = \frac{\exp\left(\frac{-|x_i - x_j|^2}{2\sigma_i^2}\right)}{\sum_{k \neq i} \exp\left(\frac{-|x_i - x_k|^2}{2\sigma_i^2}\right)} \tag{13.1}$$

where σ_i is the variance of the Gaussian centered at x_i. This variance depends on the individual data point x_i, and it is not a unique variance defined over the whole space. Also, $P_{i|i}$ is set to zero as we are concerned only about pairwise similarities between the data points and hence avoid any point choosing itself as its neighbor.

Similarly, let y_i and y_j be the mapping of the original data points x_i and x_j respectively in the low dimensional space p. It is possible to compute a similar probability $q_{j|i}$ in the low dimensional space given by:

$$q_{j|i} = \frac{\exp\left(-\left|y_i - y_j\right|^2\right)}{\sum_{k \neq i} \exp\left(-\left|y_i - y_k\right|^2\right)} \tag{13.2}$$

Here the variance $\sigma = \frac{1}{\sqrt{2}}$ is constant throughout the low dimensional space. Also, $q_{i|i}$ is set to zero. If the similarity between the data points in the original space is retained in the low dimensional mapping as well, then the conditional probabilities in both the spaces, $p_{j|i}$ and $q_{j|i}$ will be equal. Hence, we need to choose a mapping $y_1, \ldots, y_n \in \mathbb{R}^p$ in the low dimensional space p, such that it minimizes the difference between $p_{j|i}$ and $q_{j|i}$. We define a cost function C, called a Kullback-Leibler divergence (KL divergence), as a measure of the distance between the two conditional probabilities. The objective of SNE is to find the low dimensional representation by minimizing this cost function, that is, by minimizing the sum of KL divergences over all the points. The cost function is defined as:

$$C = \sum_i KL(P_i \,||\, Q_i) = \sum_{ij} p_{j|i} log \frac{p_{j|i}}{q_{j|i}} \tag{13.3}$$

where, for a given point x_i and its mapping y_i, P_i and Q_i are the conditional probability distributions over all the other points and map points respectively. Note that $KL(P_i \,||\, Q_i)$ is not the same as $KL(Q_i \,||\, P_i)$, that is, KL divergence is not symmetric. Hence, the different types of mismatch in the distances between points when mapped onto a low dimensional space have different effects on the value of the cost function. If a high $p_{j|i}$ (x_i and x_j are neighbors) is modelled by a small $q_{j|i}$ (y_i and y_j are far apart), it incurs a high cost (since $p_{j|i} > q_{j|i}$ and log of a large value is large), whereas, if a low $p_{j|i}$ (x_i and x_j are far apart) is modelled by a high $q_{j|i}$ (y_i and y_j are neighbors), a low cost is incurred.

The optimization of the cost function becomes difficult, and moreover, SNE suffers from the "crowding problem." This crowding problem arises due to the fact that high dimensions have more space to accommodate data points than low dimensions. When we embed a data point from a high dimensional space to a lower dimensional space, the mapped points become overcrowded. Suppose, if $d+1$ points can be accommodated equidistant from each other on a d dimensional space, then when we embed these points in a lower dimensional space p (where $p < d$), these points have to be squished or moved closer to each other as the area available to accommodate the points is reduced, leading to distortion of the distances between them. This is termed the crowding problem.

Thus, a new technique called t-distributed SNE (t-SNE) was introduced as a variant of SNE that aims at alleviating these drawbacks by making the cost function

of SNE symmetric and using t-distribution instead of Gaussian distribution. This algorithm is different from SNE in two ways:

1. t-SNE uses symmetric joint probabilities where $p_{ij} = p_{ji}$ (in high dimension) and $q_{ij} = q_{ji}$ (in low dimension), unlike the asymmetric conditional probabilities, $p_{j|i}$ and $q_{j|i}$ used in SNE. Here, the pairwise similarity in high dimension in terms of probability, p_{ij} is given by:

$$p_{ij} = \frac{\exp\left(\frac{-|x_i - x_j|^2}{2\sigma^2}\right)}{\sum_{k \neq i} \exp\left(\frac{-|x_i - x_k|^2}{2\sigma^2}\right)} \tag{13.4}$$

2. In the high dimensional space, Gaussian distribution is used to convert distances into probabilities, whereas in the low dimensional mapping, rather than using Gaussian, t-SNE uses student t-distribution which has heavier tails than Gaussian. As heavy-tailed distribution allows more space for points in the low dimensional space, using t-distribution overcomes the crowding problem. The joint probability q_{ij} in low dimension using t-distribution is given by:

$$q_{ij} = \frac{\frac{1}{1+|y_i - y_j|^2}}{\sum_{k \neq i}\left(\frac{1}{1+|y_i - y_k|^2}\right)}$$

$$q_{ij} = \frac{\left(1+|y_i - y_j|^2\right)^{-1}}{\sum_{k \neq i}\left(1+|y_i - y_k|^2\right)^{-1}} \tag{13.5}$$

Thus, t-SNE finds the low dimensional mapping $y_1, \ldots, y_n \in \mathbb{R}^p$ by minimizing the symmetric cost function using the gradient descent method, that is, by minimizing the KL divergence between the symmetric joint probability distribution P in high dimension and the t-distribution based joint probability distribution Q in the low dimension, given by:

$$C = KL(P||Q) = \sum_{ij} p_{ij} log \frac{p_{ij}}{q_{ij}} \tag{13.6}$$

13.2 ADVANTAGES AND LIMITATIONS

- t-SNE outperforms other techniques in handling non-linear data efficiently as it very well captures the complex polynomial relationships between the features making it a highly efficient nonlinear dimensionality reduction technique.

- Also, as discussed earlier in Section 13.1, t-SNE very well preserves the locality of the data while at the same time reveals some of the important global structure of the data in the low dimensional mapping. It has the tendency to identify the structure of the data at many scales and also reveals high dimensional data that lie on multiple manifolds and clusters.
- Although t-SNE outperforms various other non-linear dimensionality reduction techniques by providing very good mappings in low dimensional space, it has a few potential drawbacks.
- While its performance is efficient on small datasets, its quadratic time and space complexity makes it computationally complex when applied on reasonably large datasets [1]. This makes it computationally very slow on datasets with more than a few thousand data points. To overcome this, the landmark approach to t-SNE was proposed that made it computationally efficient to visualize large real-world datasets.
- Additionally, the performance of t-SNE on general dimensionality reduction where the dimensionality of the data is reduced to a dimension greater than three ($p>3$) is not very clear. The performance of t-SNE when reducing to a two- or three-dimensional space cannot be generalized to dimensions more than three because of the heavy tails of the student t-distribution which, in high dimensional spaces, lead to mappings that do not preserve the local structure of the data as the heavy tails encompass a large proportion of the probability mass under the t-distribution in high dimensional spaces.
- Another major weakness of this algorithm is that it fails when the data has intrinsically high dimensional structure as t-SNE performs dimensionality reduction mainly based on the local properties of the data. t-SNE's assumption on the local linearity of the manifold is debased in the case of data having high intrinsic dimensionality and highly varying manifold, making it less successful on such data.
- Furthermore, another limitation is the non-convexity of its cost function. Since the cost function of t-SNE is not convex it is required to choose many optimization parameters and this affects the constructed solutions as they depend on the choice of the optimization parameters. Sometimes the same choice of parameters can give different results in different runs.
- However, experiments on various real-world datasets show that t-SNE outperforms various state-of-the-art dimensionality reduction and data visualization techniques in many ways.

13.3 USE CASES

t-SNE is a powerful data exploration and data visualization technique with various applications in image processing, bioinformatics, signal processing, speech processing, NLP, and many more. It is used to visualize high dimensional data by identifying patterns in data and mapping them into low dimensional space for effective visualization with diverse applications such as computational biology, computer security, music analysis, and cancer biology [3]. This method is extensively used in

image processing applications like facial expression recognition and medical imaging and is also used for natural language processing problems like text comparison using word vector representations and finding word embeddings in low dimensional space. It is also applied on genomic data which are generally of very high dimensions for applications like gene sequence analysis [4].

13.4. EXAMPLES

Example 1

Let us now visualize the results of dimensionality reduction by t-SNE on the 64-dimensional digits dataset and the MNIST digits dataset. Firstly, let us perform dimensionality reduction on the digits dataset with 1083 samples lying in a 64-dimensional space to map it to a two-dimensional feature space.

For this example, we use the *sklearn.manifold* module from the scikit-learn library which implements various manifold learning algorithms among which t-SNE is one.

First, import the necessary libraries.

```
import sklearn
from sklearn import datasets
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
```

Next, load the dataset from sklearn. There are 6 classes of digits and 1083 data points in this loaded dataset. Each data point is of size 8×8 pixels. Hence, the dimensionality is 64.

```
digits = datasets.load_digits(n_class=6)
X = digits.data
y = digits.target
n_samples, n_features = X.shape
print(n_features)
print(n_samples)
```

Output:

```
64
1083
```

Use the t-SNE model from sklearn where the parameter *n_components* denotes the dimensionality of the target projection space, which in this case is 2. Then fit and transform the data using t-SNE.

```
tsne = TSNE(n_components=2, random_state=0)
X_tsne = tsne.fit_transform(X)
```

Now, let us visualize the transformed data using matplotlib. The results are visualized in Figure 13.1.

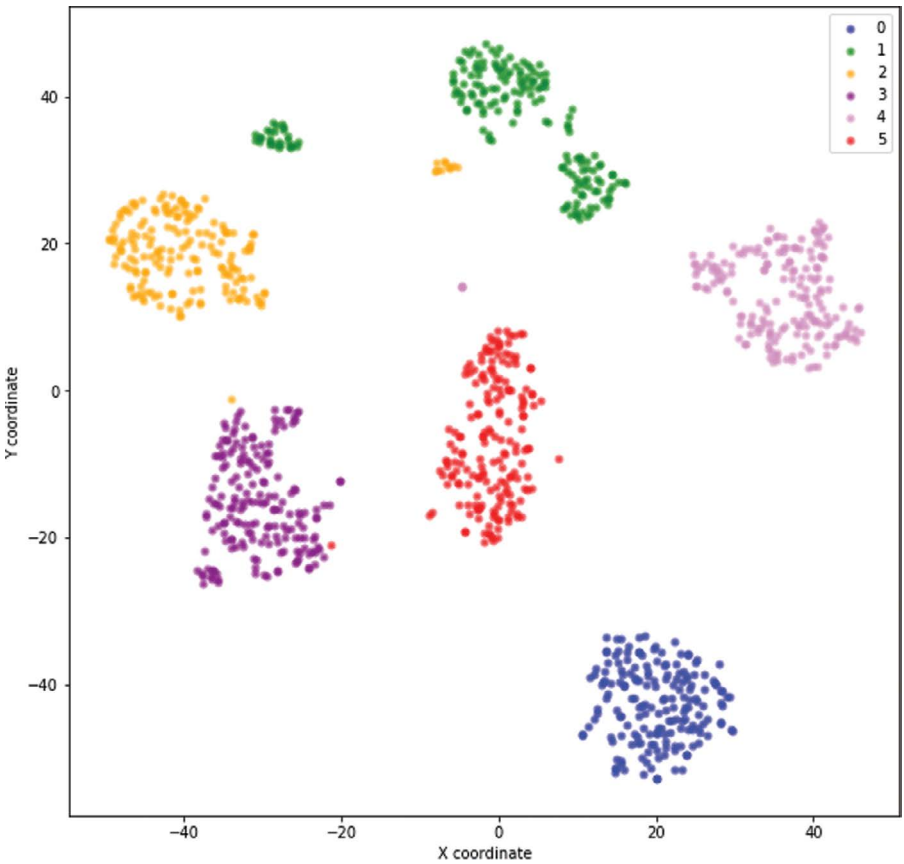

FIGURE 13.1 t-SNE on digits dataset.

```
plt.figure(figsize=(10,10))
plt.scatter(X_tsne[y==0, 0], X_tsne [y==0, 1], color='blue',
alpha=0.5,label='0', s=9, lw=1)
plt.scatter(X_tsne [y==1, 0], X_tsne [y==1, 1], color='green',
alpha=0.5,label='1',s=9, lw=1)
plt.scatter(X_tsne [y==2, 0], X_tsne [y==2, 1],
color='orange', alpha=0.5,label='2',s=9, lw=1)
plt.scatter(X_tsne [y==3, 0], X_tsne [y==3, 1],
color='purple', alpha=0.5,label='3',s=9, lw=1)
plt.scatter(X_tsne [y==4, 0], X_tsne [y==4, 1],
color='violet', alpha=0.5,label='4',s=9, lw=1)
plt.scatter(X_tsne [y==5, 0], X_tsne [y==5, 1], color='red',
alpha=0.5,label='5',s=9, lw=1)
plt.ylabel('Y coordinate')
plt.xlabel('X coordinate')
plt.legend()
plt.show()
```

Example 2

In this example, dimensionality reduction is performed by t-SNE on the MNIST dataset by taking a sample of 10000 data points to reduce the computational complexity to produce a two-dimensional mapping of the data points. This dataset is loaded from the tensorflow library and similar to the previous example, we use the implementation of the t-SNE algorithm from the sklearn library.

Import all the necessary libraries.

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import sklearn
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
```

Load the dataset. This dataset has 55000 data points with a dimensionality of 784. However, we are taking a sample of 10000 data points for this example.

```
mnist = input_data.read_data_sets("MNIST_data/")
X_train = mnist.train.images
y_train = mnist.train.labels
X_train = X_train [0:10000]
y_train = y_train [0:10000]
n_samples, n_features = X_train.shape
print(n_features)
print(n_samples)
```

Output:

```
784
10000
```

Use the t-SNE model from the *sklearn.manifold* module to reduce the data from 784 to 2 dimensions.

```
tsne = TSNE(n_components=2, random_state=0)
X_tsne = tsne.fit_transform(X_train)
```

Finally, plot the transformed data with each data point denoted by a color corresponding to its class label.

```
plt.figure(figsize=(10,10))
plt.scatter(X_tsne[y_train==0, 0], X_tsne[y_train==0, 1],
color='blue', alpha=0.5,label='0', s=9, lw=2)
plt.scatter(X_tsne[y_train==1, 0], X_tsne[y_train==1, 1],
color='purple', alpha=0.5,label='1',s=9, lw=2)
plt.scatter(X_tsne[y_train==2, 0], X_tsne[y_train==2, 1],
color='yellow', alpha=0.5,label='2',s=9, lw=2)
plt.scatter(X_tsne[y_train==3, 0], X_tsne[y_train==3, 1],
color='black', alpha=0.5,label='3',s=9, lw=2)
plt.scatter(X_tsne[y_train==4, 0], X_tsne[y_train==4, 1],
color='gray', alpha=0.5,label='4',s=9, lw=2)
```

```
plt.scatter(X_tsne[y_train==5, 0], X_tsne[y_train==5, 1],
color='lightgreen', alpha=0.5,label='5',s=9, lw=2)
plt.scatter(X_tsne[y_train==6, 0], X_tsne[y_train==6, 1],
color='red', alpha=0.5,label='6',s=9, lw=2)
plt.scatter(X_tsne[y_train==7, 0], X_tsne[y_train==7, 1],
color='green', alpha=0.5,label='7',s=9, lw=2)
plt.scatter(X_tsne[y_train==8, 0], X_tsne[y_train==8, 1],
color='lightblue', alpha=0.5,label='8',s=9, lw=2)
plt.scatter(X_tsne[y_train==9, 0], X_tsne[y_train==9, 1],
color='orange', alpha=0.5,label='9',s=9, lw=2)
plt.ylabel('Y coordinate')
plt.xlabel('X coordinate')
plt.legend()
plt.show()
```

The resulting low dimensional embeddings are visualized in Figure 13.2. Note that t-SNE does a good job as the classes are clearly separated, thus revealing the natural classes of digits in the data.

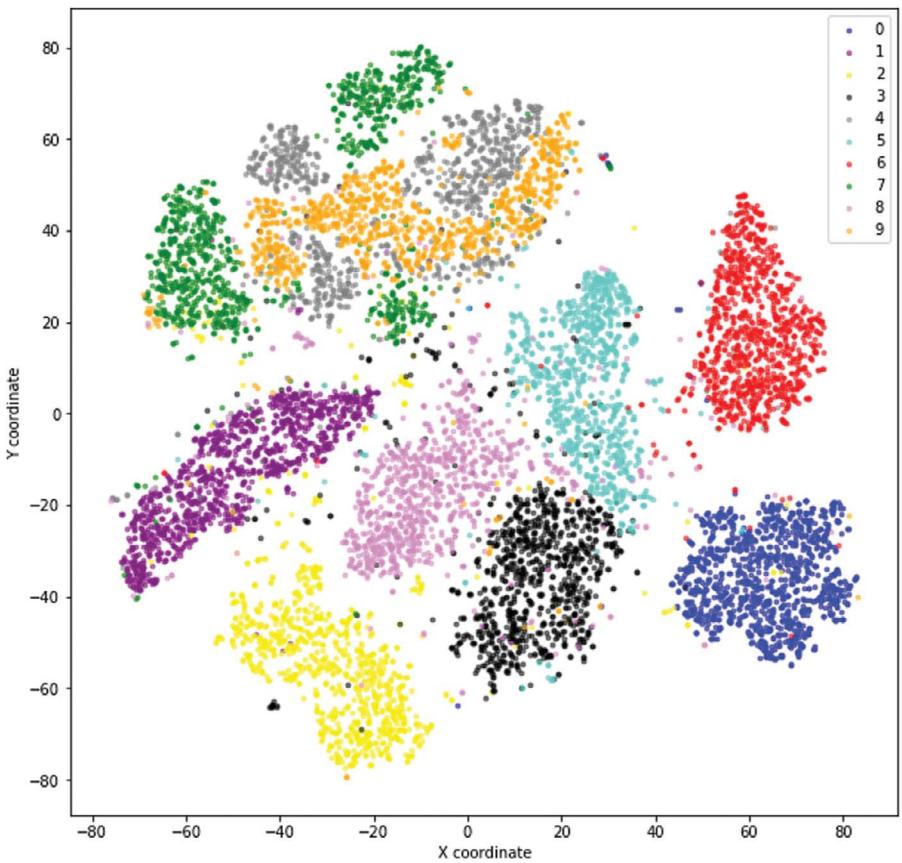

FIGURE 13.2 t-SNE on MNIST dataset.

Try this demo of t-SNE: https://distill.pub/2016/misread-tsne/

How to Use t-SNE Effectively

Although extremely useful for visualizing high-dimensional data, t-SNE plots can sometimes be mysterious or misleading. By exploring how it behaves in simple cases, we can learn to use it more effectively.