**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 fea‐ tures considerably, turning an intractable problem into a tractable one. For example, consider the MNIST images (introduced in Chapter 3): the pixels on the image bor‐ ders 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.



**215**

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 per‐ form slightly worse. It also makes your pipelines a bit more com‐ plex and thus harder to maintain. So you should first try to train your system with the original data before considering using dimen‐ sionality 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 per‐ formance (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**

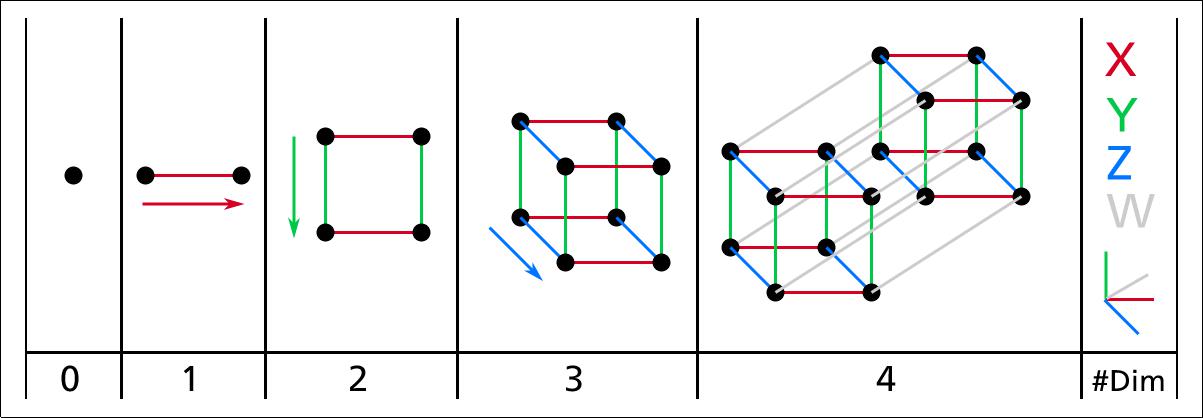
We are so used to living in three dimensions[1](#page2) that our intuition fails us when we try to imagine a high-dimensional space. Even a basic 4D hypercube is incredibly hard to picture in our mind (see [Figure 8-1](#page3)), let alone a 200-dimensional ellipsoid bent in a 1,000-dimensional space.



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



**216** **|** **Chapter 8: Dimensionality Reduction**



*Figure 8-1. Point, segment, square, cube, and tesseract (0D to 4D hypercubes)*[*2*](#page3)

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 × 1 × ⋯ × 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](#page3)

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

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 relia‐ ble 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



1. Watch a rotating tesseract projected into 3D space at [*https://homl.info/30*](https://homl.info/30). Image by Wikipedia user Nerd‐ Boy1392 ([Creative Commons BY-SA 3.0](https://creativecommons.org/licenses/by-sa/3.0/)). Reproduced from [*https://en.wikipedia.org/wiki/Tesseract*](https://en.wikipedia.org/wiki/Tesseract).
2. 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.
3. 

**The Curse of Dimensionality** **|** **217**

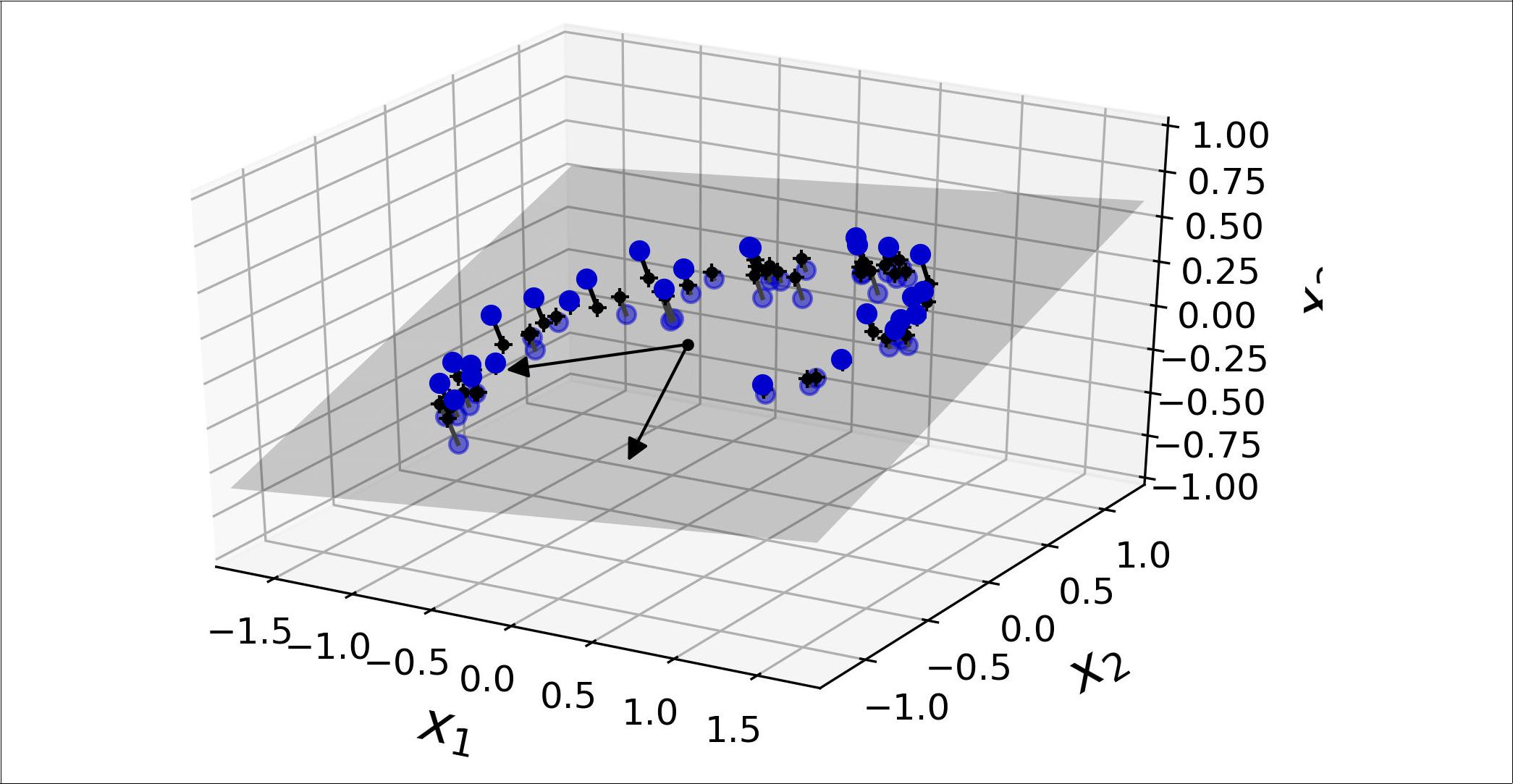
in the MNIST problem), you would need more training instances than atoms in the observable universe in order for training instances to be within 0.1 of each other on average, assuming they were spread out uniformly across all dimensions.

**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](#page4) you can see a 3D data‐ set represented by the circles.

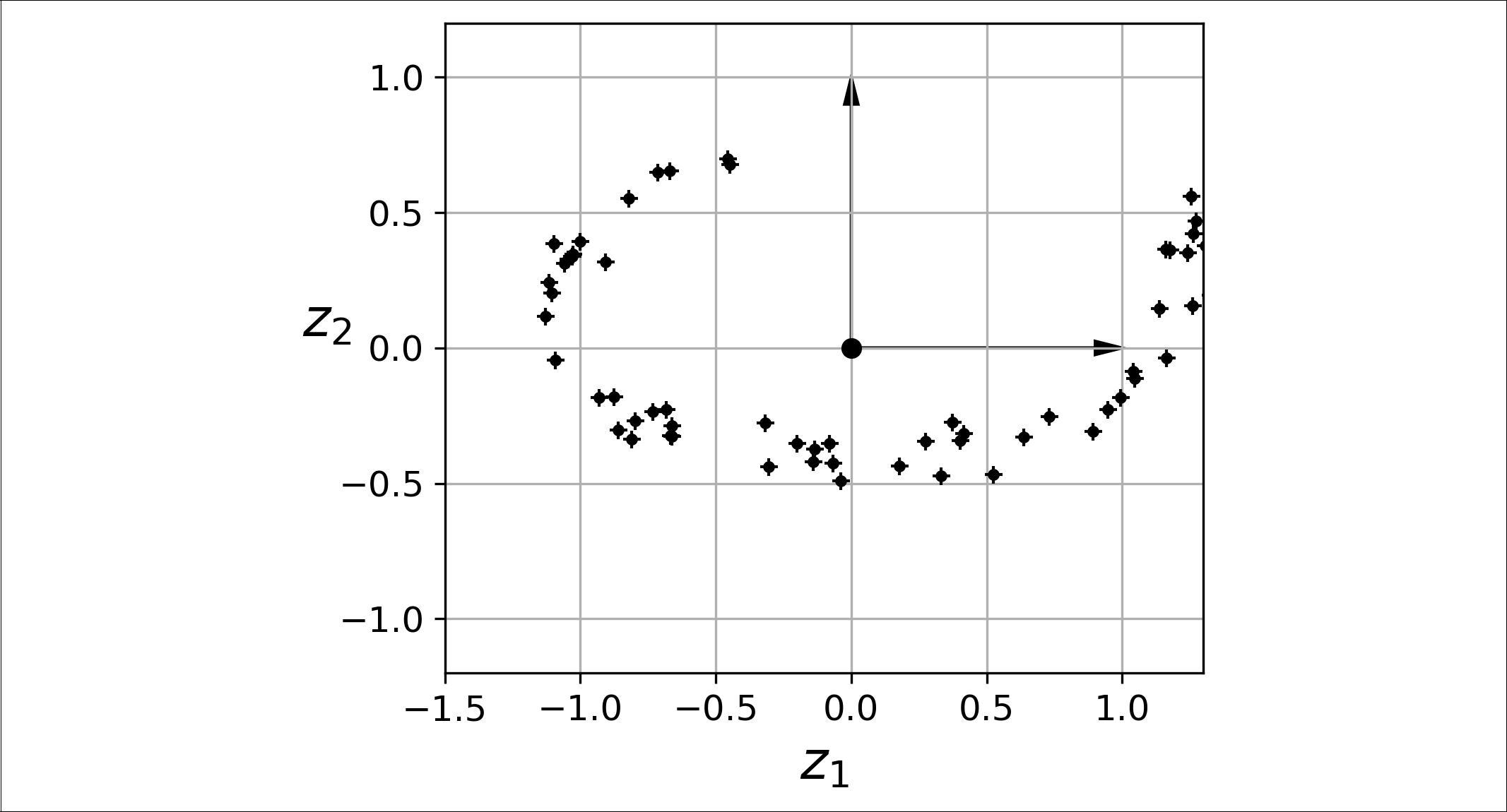


*Figure 8-2. A 3D dataset lying close to a 2D subspace*

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 con‐ necting the instances to the plane), we get the new 2D dataset shown in [Figure 8-3](#page5). 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).

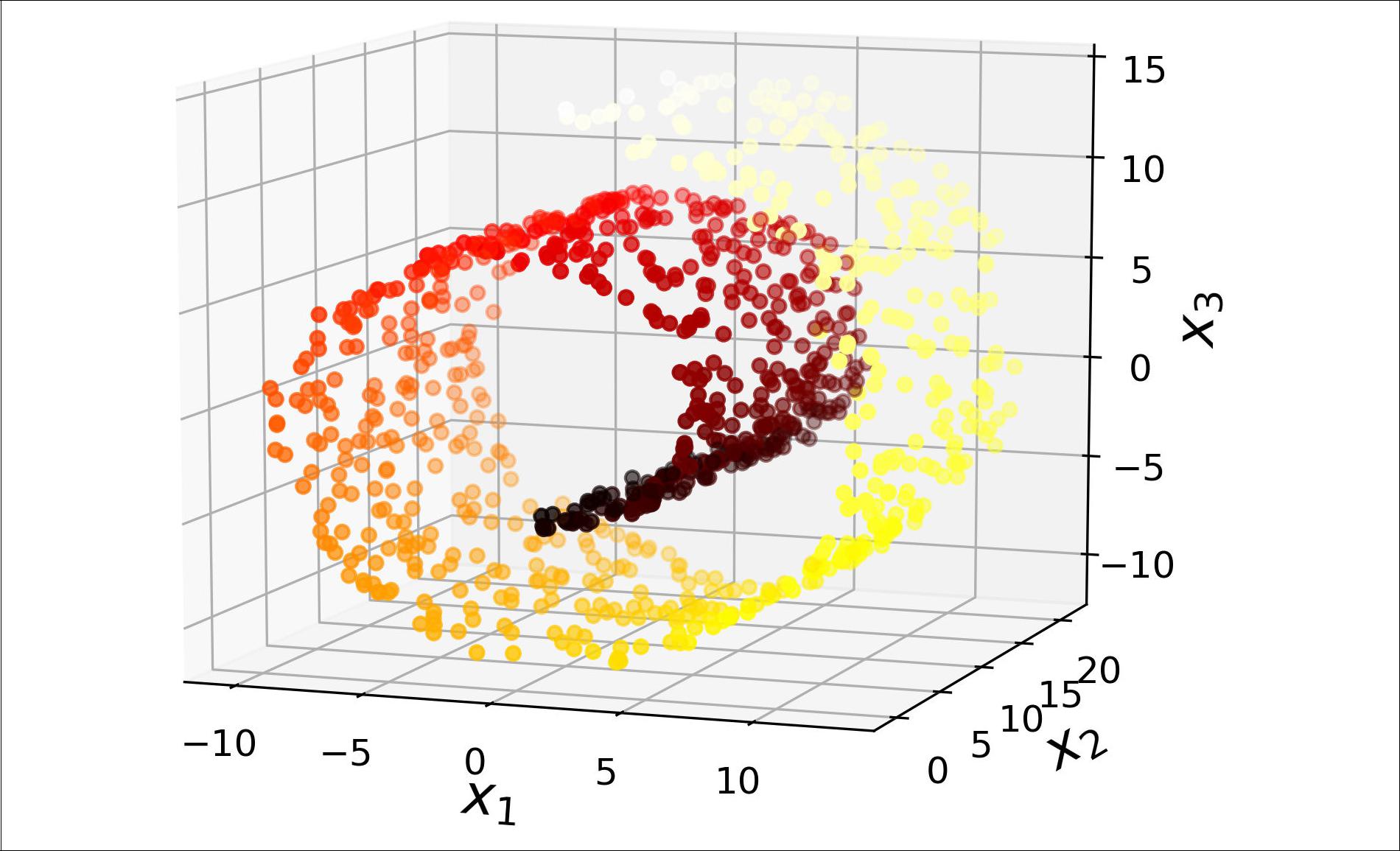


**218** **|** **Chapter 8: Dimensionality Reduction**



*Figure 8-3. The new 2D dataset a•er 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 data‐ set represented in [Figure 8-4](#page5).

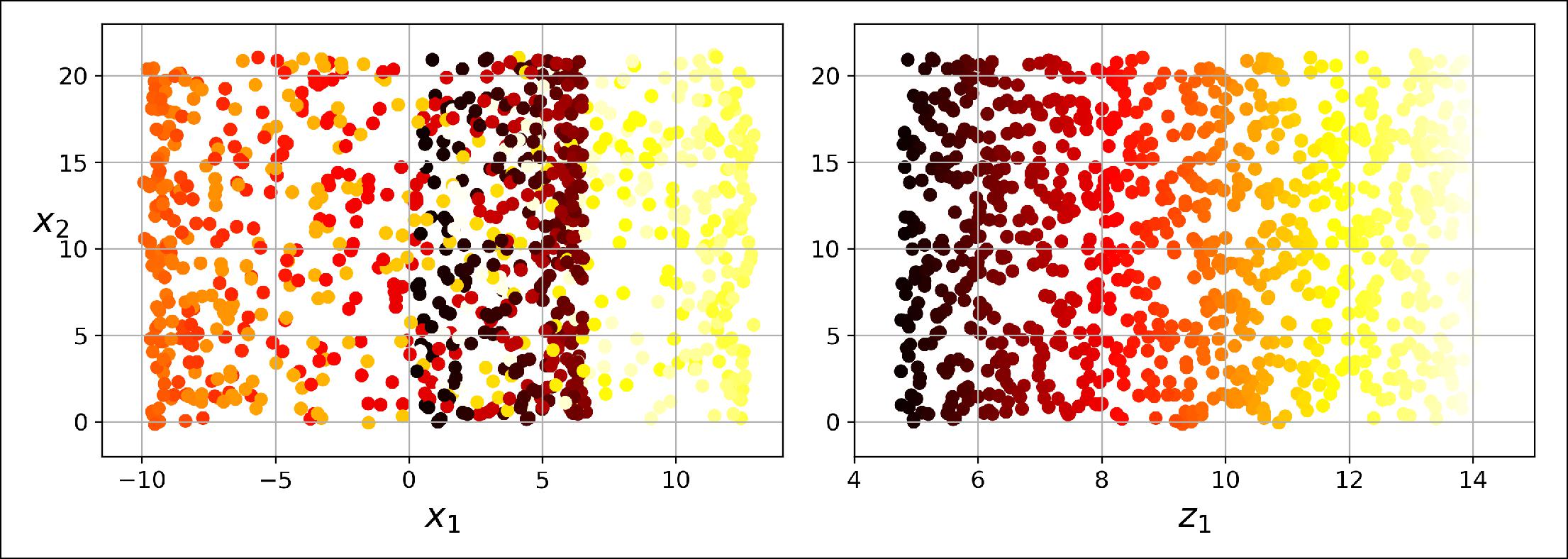


*Figure 8-4. Swiss roll dataset*



**Main Approaches for Dimensionality Reduction** **|** **219**

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](#page6). However, what you really want is to unroll the Swiss roll to obtain the 2D dataset on the right of [Figure 8-5](#page6).



*Figure 8-5. Squashing by projecting onto a plane (le•) 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 locallyresembles 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-worldhigh-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](#page7) the Swiss roll is split into two classes: in the 3D space (on the left), the decision



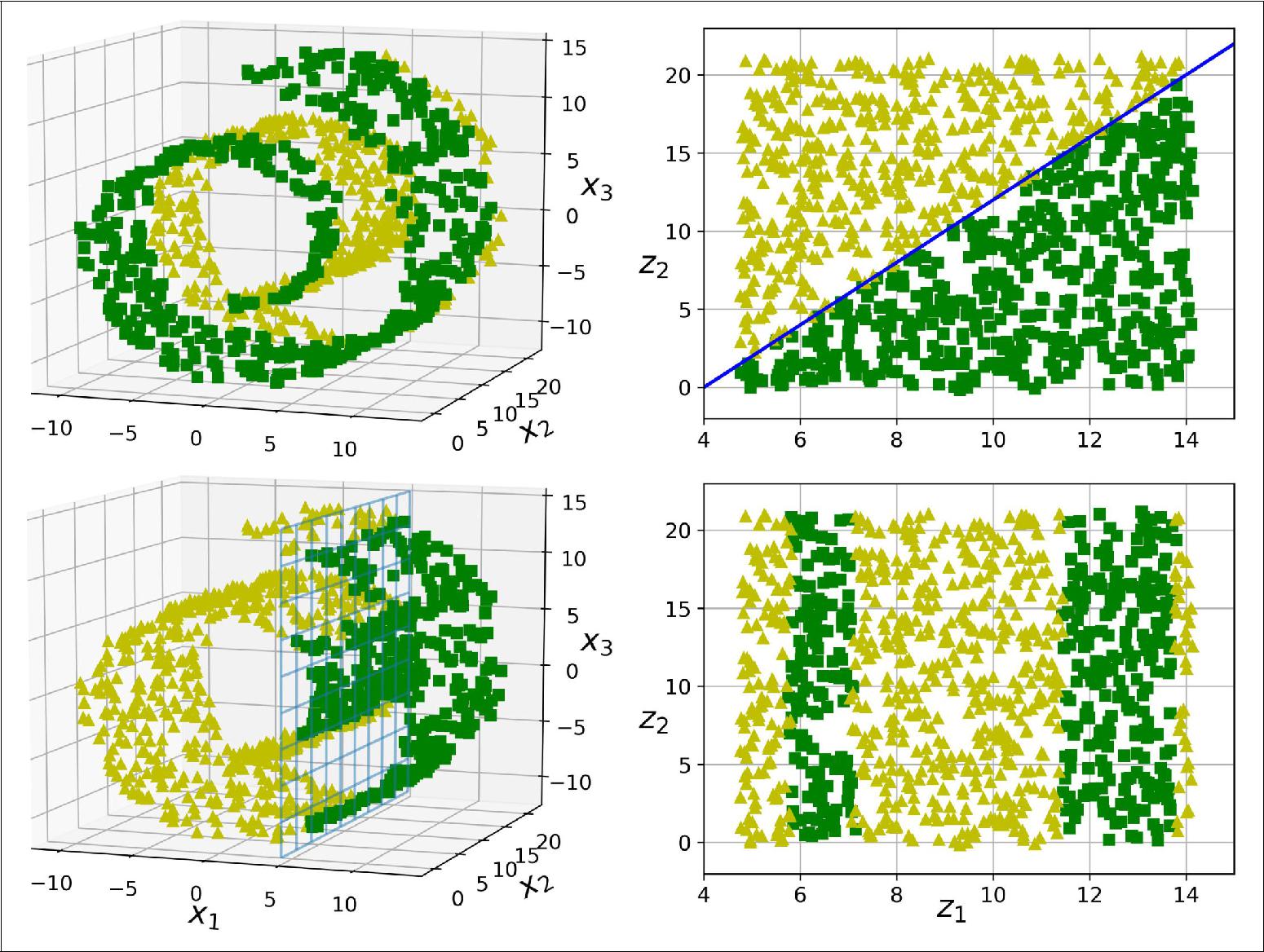
**220** **|** **Chapter 8: Dimensionality Reduction**

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](#page7), 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 sim‐ pler 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*



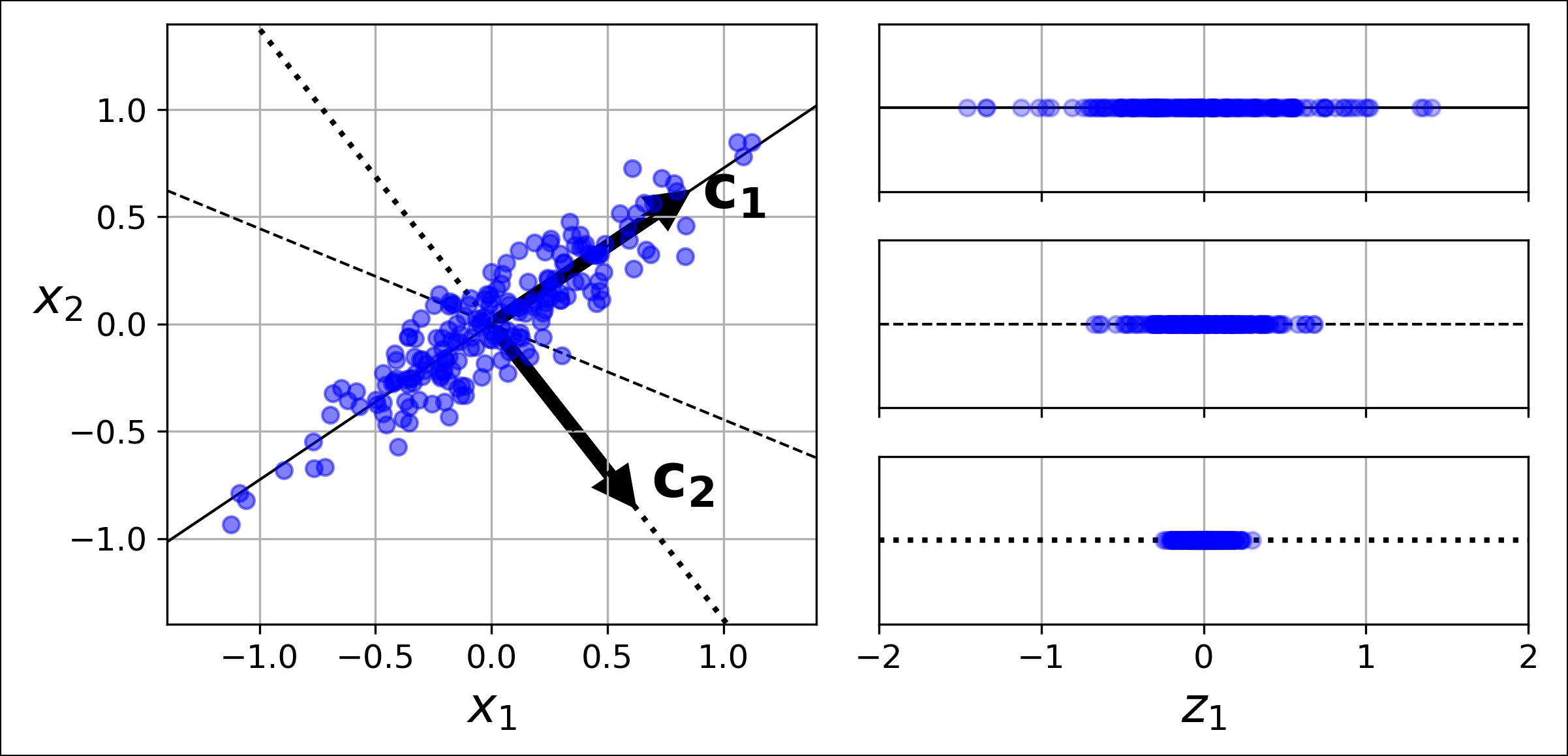
**Main Approaches for Dimensionality Reduction** **|** **221**

**PCA**

*Principal Component Analysis* (PCA) is by far the most popular dimensionality reduc‐tion 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](#page4).

**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 repre‐ sented on the left of [Figure 8-7](#page8), 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 var‐ iance, 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 dis‐ tance between the original dataset and its projection onto that axis. This is the rather simple idea behind [PCA](https://homl.info/31).[4](#page8)



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



**222** **|** **Chapter 8: Dimensionality Reduction**

**Principal Components**

PCA identifies the axis that accounts for the largest amount of variance in the train‐ ing set. In [Figure 8-7](#page8), 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 data‐ set, 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 ith axis is called the ith *principal component* (PC). In [Figure 8-7](#page8), the 1st PC is **c1** and the 2nd PC is **c2**. In [Figure 8-2](#page4) 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 per‐ turb the training set slightly and run PCA again, some of the new PCs may point in the opposite direction of the original PCs. How‐ ever, 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 **X** into the matrix multiplication of three matrices **U** Σ **V***T*, where **V** contains all the principal components that we are looking for, as shown in [Equation 8-1](#page9).

*Equation 8-1. Principal components matrix*

**V** = **c1 c2** ⋯ **cn**

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** **|** **223**

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 pre‐ ceding 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 dimen‐ sionality 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](#page4) the 3D dataset is projected down to the 2D plane defined by the first two principal com‐ ponents, preserving a large part of the dataset’s variance. As a result, the 2D projec‐ tion 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 **X** by the matrix **W***d*, defined as the matrix containing the first *d* principal components (i.e., the matrix composed of the first *d* columns of **V**), as shown in [Equation 8-2](#page10).

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

**X***d*‐proj=**XW***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 compo‐ nents using the components\_ variable (note that it contains the PCs as horizontal vec‐



**224** **|** **Chapter 8: Dimensionality Reduction**

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 prin‐ cipal 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 com‐ ponent. For example, let’s look at the explained variance ratios of the first two compo‐ nents of the 3D dataset represented in [Figure 8-2](#page4):

**>>>** 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 reason‐ able 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 dimen‐ sionality 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:

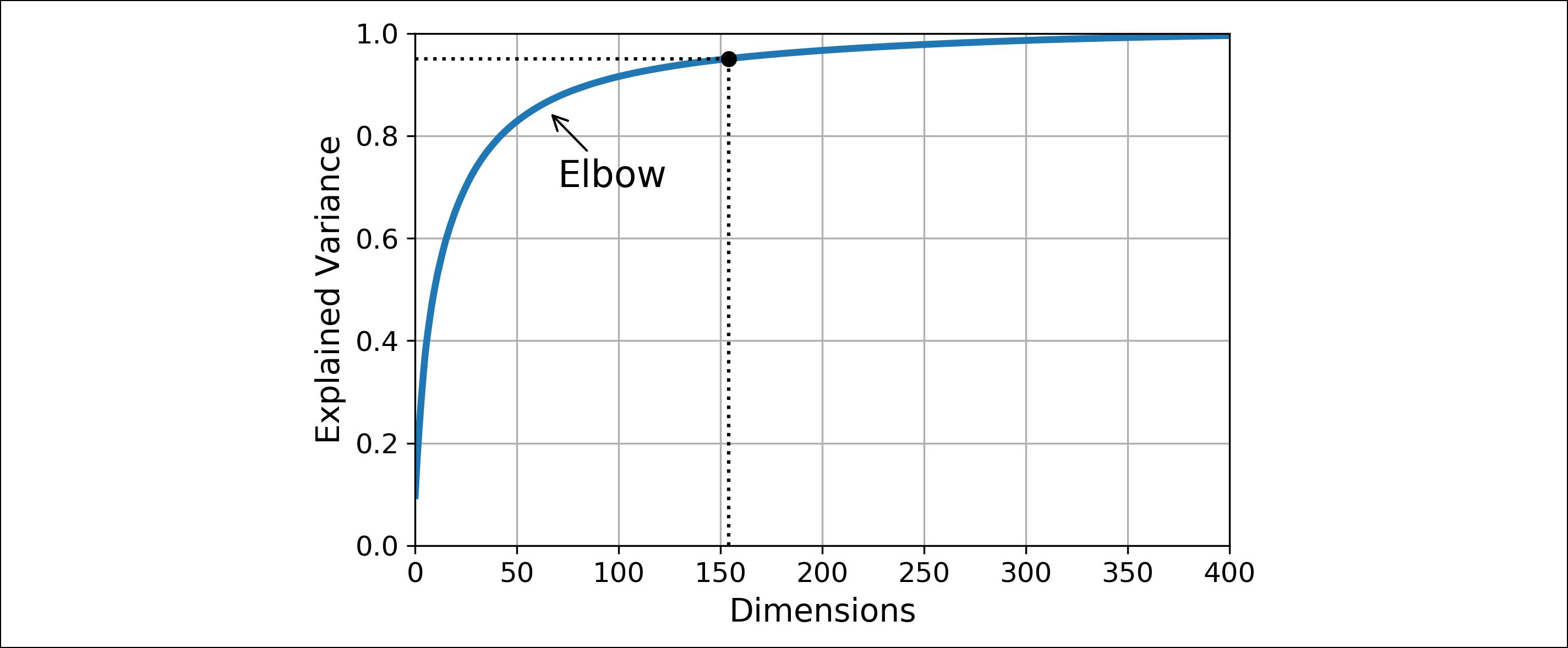
pca = PCA(n\_components=0.95) X\_reduced = pca.fit\_transform(X\_train)

Yet another option is to plot the explained variance as a function of the number of dimensions (simply plot cumsum; see [Figure 8-8](#page12)). 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



**PCA** **|** **225**

dimensionality down to about 100 dimensions wouldn’t lose too much explained var‐ iance.



*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 var‐ iance. 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.

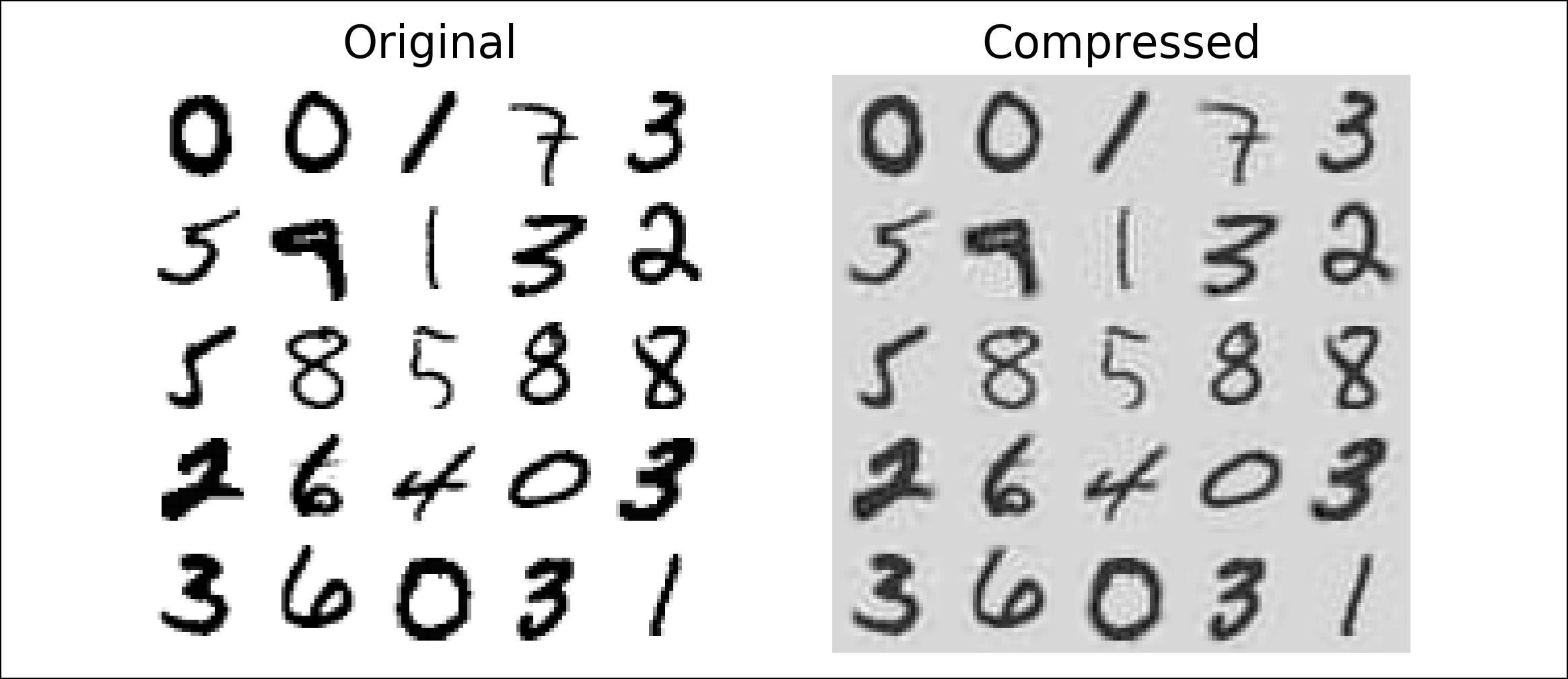
It is also possible to decompress the reduced dataset back to 784 dimensions by applying the inverse transformation of the PCA projection. Of course this won’t give you back the original data, since the projection lost a bit of information (within the 5% variance that was dropped), but it will likely 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](#page13) shows a few digits from the original training set (on the left), and the cor‐ responding 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)



**226** **|** **Chapter 8: Dimensionality Reduction**



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

The equation of the inverse transformation is shown in [Equation 8-3](#page13).

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

**X**recovered = **X***d*‐proj**W***dT*

**Randomized PCA**

If you set the svd\_solver hyperparameter to "randomized", Scikit-Learn uses a sto‐ chastic algorithm called *Randomized PCA* that quickly finds an approximation of the first *d* principal components. Its computational complexity is *O*(*m* × *d*2) + *O*(*d*3), instead of *O*(*m* × *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 trainingset into mini-batches and feed an IPCA algorithm one mini-batch at a time. This is



**PCA** **|** **227**

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](https://homl.info/32)[5](#page14) 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*](https://homl.info/33)



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



**228** **|** **Chapter 8: Dimensionality Reduction**

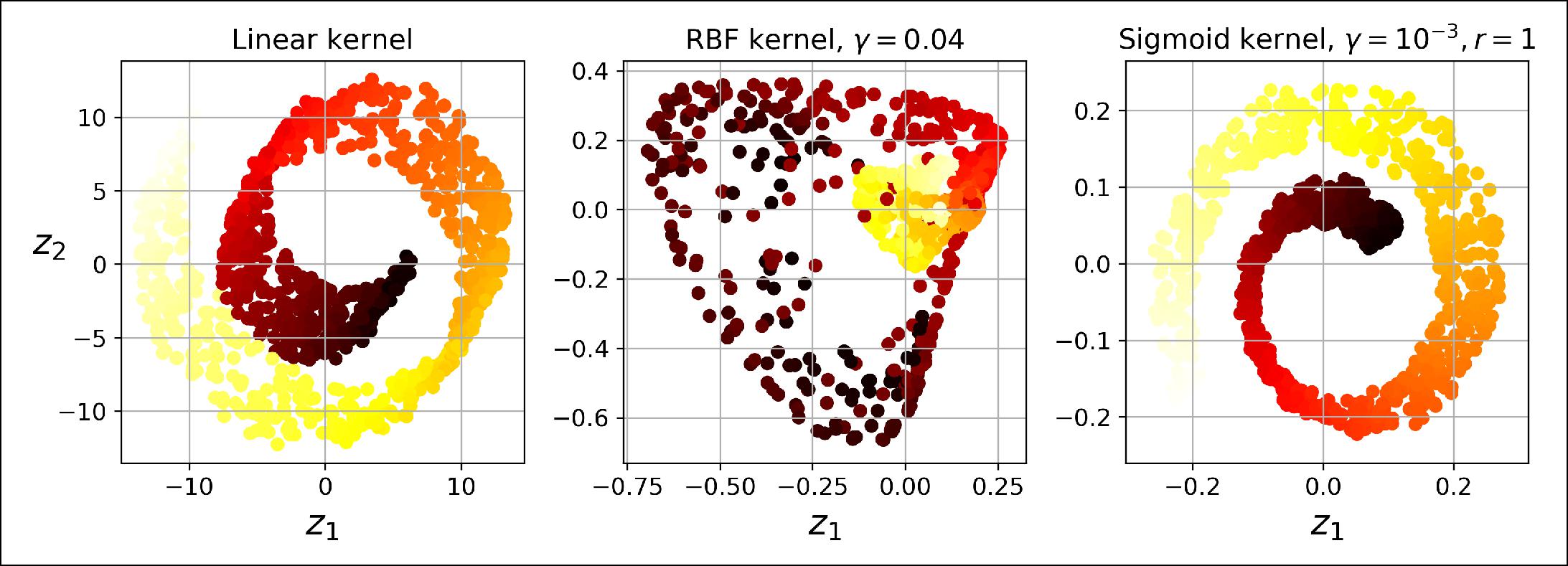
[*PCA* (kPCA)](https://homl.info/33).[6](#page15)Itis often good at preserving clusters of instances after projection, orsometimes even unrolling datasets that lie close to a twisted manifold.

For example, the following code uses Scikit-Learn’s KernelPCA class to perform kPCA with an RBF kernel (see Chapter 5 for more details about the RBF kernel and the other kernels):

**from sklearn.decomposition import** KernelPCA

rbf\_pca = KernelPCA(n\_components = 2, kernel="rbf", gamma=0.04) X\_reduced = rbf\_pca.fit\_transform(X)

[Figure 8-10](#page15) shows the Swiss roll, reduced to two dimensions using a linear kernel (equivalent to simply using the PCA class), an RBF kernel, and a sigmoid kernel (Logistic).



*Figure 8-10. Swiss roll reduced to 2D using kPCA with various kernels*

**Selecting a Kernel and Tuning Hyperparameters**

As kPCA is an unsupervised learning algorithm, there is no obvious performance measure to help you select the best kernel and hyperparameter values. However, dimensionality reduction is often a preparation step for a supervised learning task (e.g., classification), so you can simply use grid search to select the kernel and hyper‐ parameters that lead to the best performance on that task. For example, the following code creates a two-step pipeline, first reducing dimensionality to two dimensions using kPCA, then applying Logistic Regression for classification. Then it uses Grid SearchCV to find the best kernel and gamma value for kPCA in order to get the best classification accuracy at the end of the pipeline:

**from sklearn.model\_selection import** GridSearchCV **from sklearn.linear\_model import** LogisticRegression **from sklearn.pipeline import** Pipeline



6 “Kernel Principal Component Analysis,” B. Schölkopf, A. Smola, K. Müller (1999).



**Kernel PCA** **|** **229**

clf = Pipeline([

("kpca", KernelPCA(n\_components=2)), ("log\_reg", LogisticRegression())

])

param\_grid = [{

"kpca\_\_gamma": np.linspace(0.03, 0.05, 10), "kpca\_\_kernel": ["rbf", "sigmoid"]

}]

grid\_search = GridSearchCV(clf, param\_grid, cv=3) grid\_search.fit(X, y)

The best kernel and hyperparameters are then available through the best\_params\_ variable:

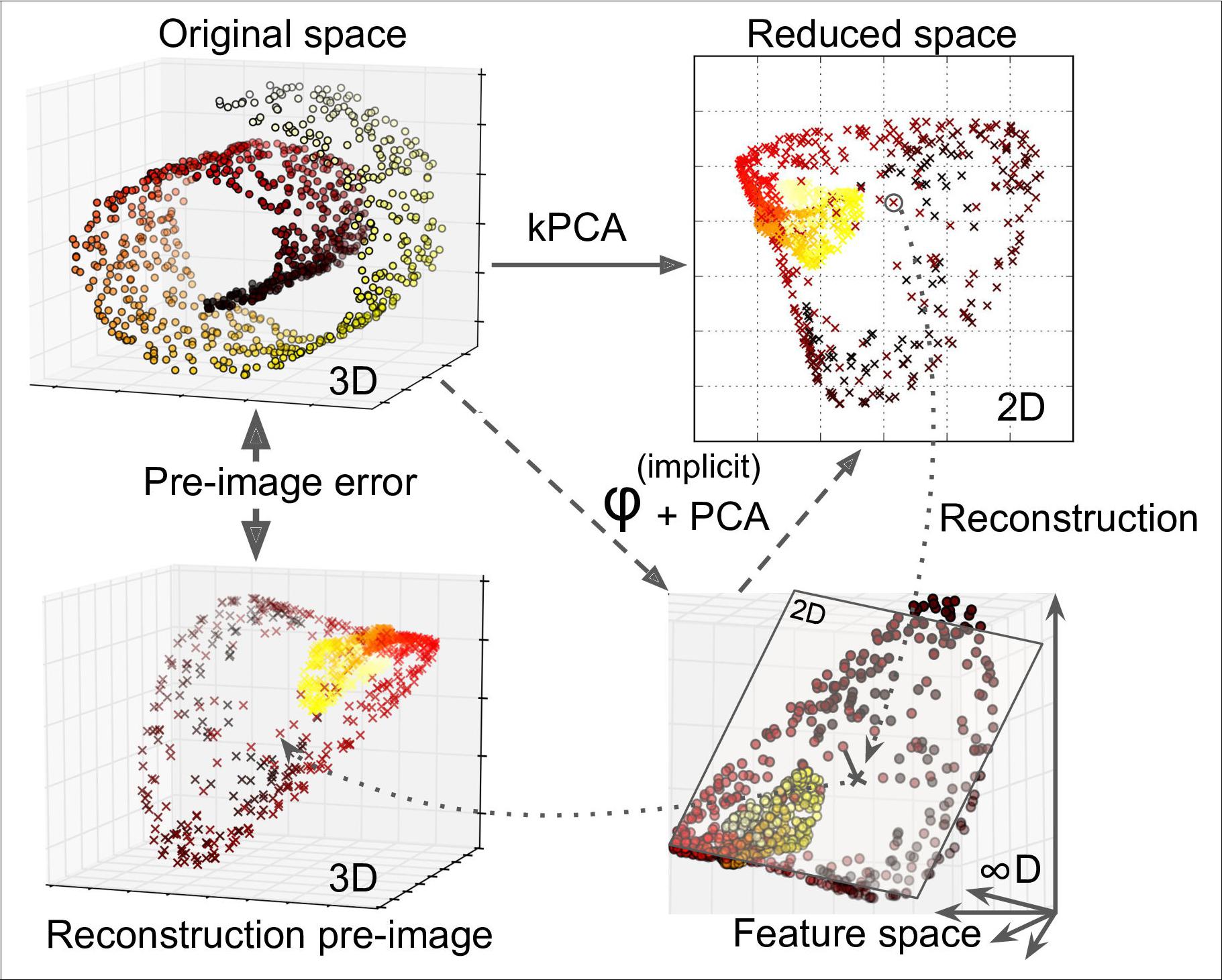
**>>> print**(grid\_search.best\_params\_)

{'kpca\_\_gamma': 0.043333333333333335, 'kpca\_\_kernel': 'rbf'}

Another approach, this time entirely unsupervised, is to select the kernel and hyper‐ parameters that yield the lowest reconstruction error. However, reconstruction is not as easy as with linear PCA. Here’s why. [Figure 8-11](#page17) shows the original Swiss roll 3D dataset (top left), and the resulting 2D dataset after kPCA is applied using an RBF kernel (top right). Thanks to the kernel trick, this is mathematically equivalent to mapping the training set to an infinite-dimensional feature space (bottom right) using the *feature map* φ, then projecting the transformed training set down to 2D using linear PCA. Notice that if we could invert the linear PCA step for a given instance in the reduced space, the reconstructed point would lie in feature space, not in the original space (e.g., like the one represented by an x in the diagram). Since the feature space is infinite-dimensional, we cannot compute the reconstructed point, and therefore we cannot compute the true reconstruction error. Fortunately, it is pos‐ sible to find a point in the original space that would map close to the reconstructed point. This is called the reconstruction *pre-image*. Once you have this pre-image, you can measure its squared distance to the original instance. You can then select the ker‐ nel and hyperparameters that minimize this reconstruction pre-image error.



**230** **|** **Chapter 8: Dimensionality Reduction**



*Figure 8-11. Kernel PCA and the reconstruction pre-image error*

You may be wondering how to perform this reconstruction. One solution is to train a supervised regression model, with the projected instances as the training set and the original instances as the targets. Scikit-Learn will do this automatically if you set fit\_inverse\_transform=True, as shown in the following code:[7](#page17)

rbf\_pca = KernelPCA(n\_components = 2, kernel="rbf", gamma=0.0433, fit\_inverse\_transform=True)

X\_reduced = rbf\_pca.fit\_transform(X)

X\_preimage = rbf\_pca.inverse\_transform(X\_reduced)

By default, fit\_inverse\_transform=False and KernelPCA has no inverse\_transform() method. This method only gets created

when you set fit\_inverse\_transform=True.



1. Scikit-Learn uses the algorithm based on Kernel Ridge Regression described in Gokhan H. Bakır, Jason Weston, and Bernhard Scholkopf, [“Learning to Find Pre-images”](https://homl.info/34) (Tubingen, Germany: Max Planck Institute for Biological Cybernetics, 2004).



**Kernel PCA** **|** **231**

You can then compute the reconstruction pre-image error:

* **from sklearn.metrics import** mean\_squared\_error
* mean\_squared\_error(X, X\_preimage) 32.786308795766132

Now you can use grid search with cross-validation to find the kernel and hyperpara‐ meters that minimize this pre-image reconstruction error.

**LLE**

[*Locally Linear Embedding*](https://homl.info/35) (LLE)[8](#page18)isanother very powerful *nonlinear dimensionality reduction* (NLDR) technique. It is a Manifold Learning technique that does not relyon projections like the previous algorithms. In a nutshell, LLE works by first measur‐ ing how each training instance linearly relates to its closest neighbors (c.n.), and then looking for a low-dimensional representation of the training set where these local relationships are best preserved (more details shortly). This makes it particularly good at unrolling twisted manifolds, especially when there is not too much noise.

For example, the following code uses Scikit-Learn’s LocallyLinearEmbedding class to unroll the Swiss roll. The resulting 2D dataset is shown in [Figure 8-12](#page19). 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 left part of the unrolled Swiss roll is stretched, while the right part is squeezed. Neverthe‐ less, LLE did a pretty good job at modeling the manifold.

**from sklearn.manifold import** LocallyLinearEmbedding

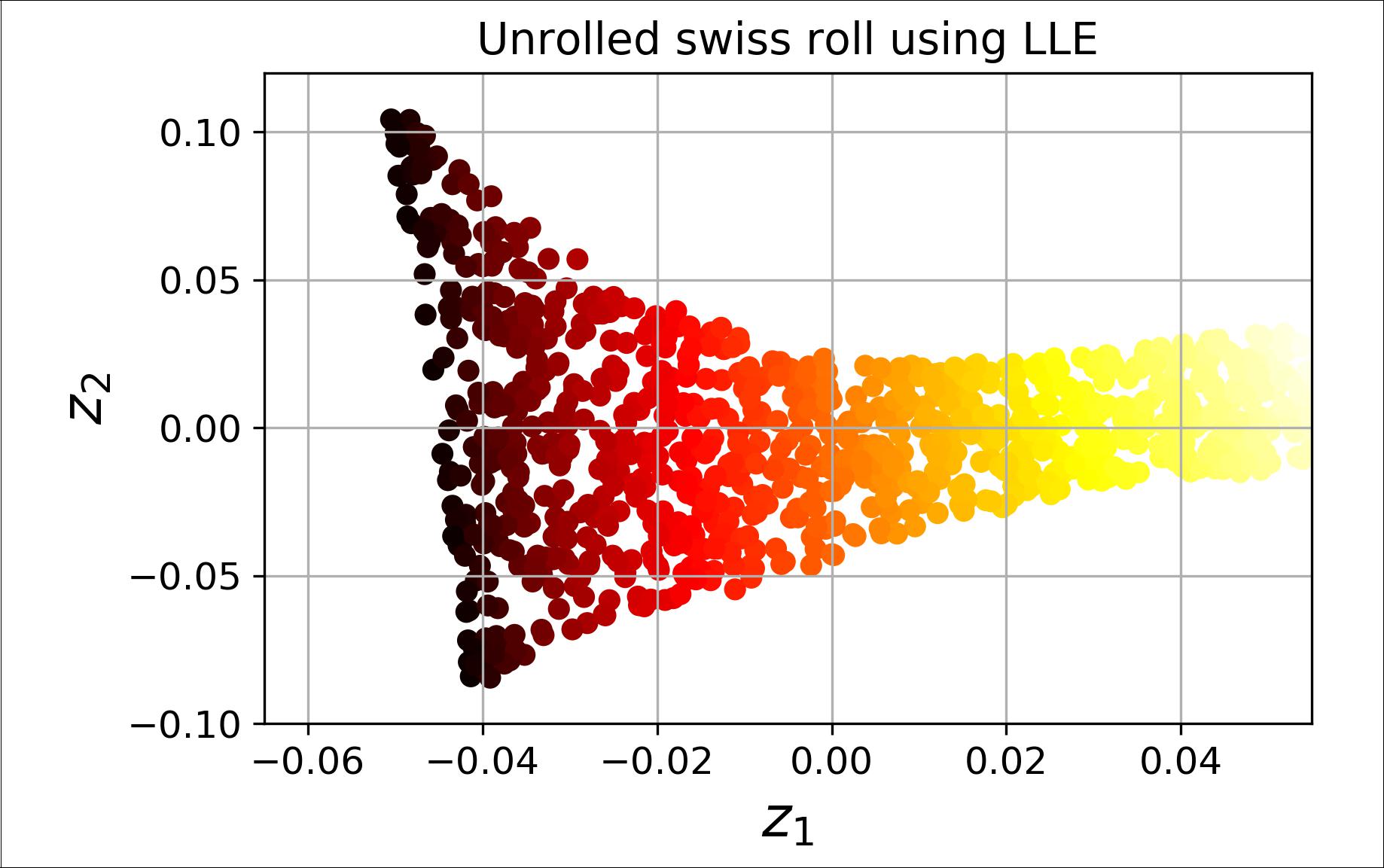
lle = LocallyLinearEmbedding(n\_components=2, n\_neighbors=10) X\_reduced = lle.fit\_transform(X)



8 “Nonlinear Dimensionality Reduction by Locally Linear Embedding,” S. Roweis, L. Saul (2000).



**232** **|** **Chapter 8: Dimensionality Reduction**



*Figure 8-12. Unrolled Swiss roll using LLE*

Here’s how LLE works: first, for each training instance **x**(*i*), the algorithm identifies its *k* closest neighbors (in the preceding code *k* = 10), then tries to reconstruct**x**(*i*)as alinear function of these neighbors. More specifically, it finds the weights *wi,j* such that

the squared distance between **x**(*i*) and ∑*mj*= 1 *wi*, *j***x***j* is as small as possible, assuming *wi,j* = 0 if **x**(*j*) is not one of the *k* closest neighbors of **x**(*i*). Thus the first step of LLE is the constrained optimization problem described in [Equation 8-4](#page20), where **W** is the weight matrix containing all the weights *wi,j*. The second constraint simply normalizes the weights for each training instance **x**(*i*).



**LLE** **|** **233**

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | *m* | | *m* | 2 |  |
| **W** = argmin | | *i* | ∑ **x** *i* − | | ∑ *wi*, *j***x** *j* | |  |
| **W** |  | = 1 | | *j* = 1 |  |  |
|  | *w* | *i*, | *j* | = 0 | if **x** *j* | is not one of the *k* c.n. of **x** *i* |  |
| subject to |  |  |  |  |  |
| *m* | |  |  |  |  |  |
|  | ∑ *wi*, *j* = 1 for *i* = 1, 2, ⋯, *m* | | | | | |  |
|  | *j* = 1 | | |  |  |  |  |



After this step, the weight matrix **W** (containing the weights *wi*, *j*) encodes the local

linear relationships between the training instances. Now 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 **z**(*i*) is the image of **x**(*i*) in this *d*-dimensional

space, then we want the squared distance between **z**(*i*) and ∑*mj*= 1 *wi*, *j***z***j* to be as small

as possible. This idea leads to the unconstrained optimization problem described in [Equation 8-5](#page20). It looks very similar to the first step, but instead of keeping the instan‐ ces 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 **Z** is the matrix containing all **z**(*i*).

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

|  |  |  |  |
| --- | --- | --- | --- |
|  | *m* | *m* | 2 |
| **Z** = argmin | ∑ | **z** *i*−∑*wi*,*j***z** *j* |  |
| **Z** | *i* = 1 | *j* = 1 |  |



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 theweights, and *O*(*dm*2) for constructing the low-dimensional representations. Unfortu‐ nately, the *m*2 in the last term makes this algorithm scale poorly to very large datasets.

**Other Dimensionality Reduction Techniques**

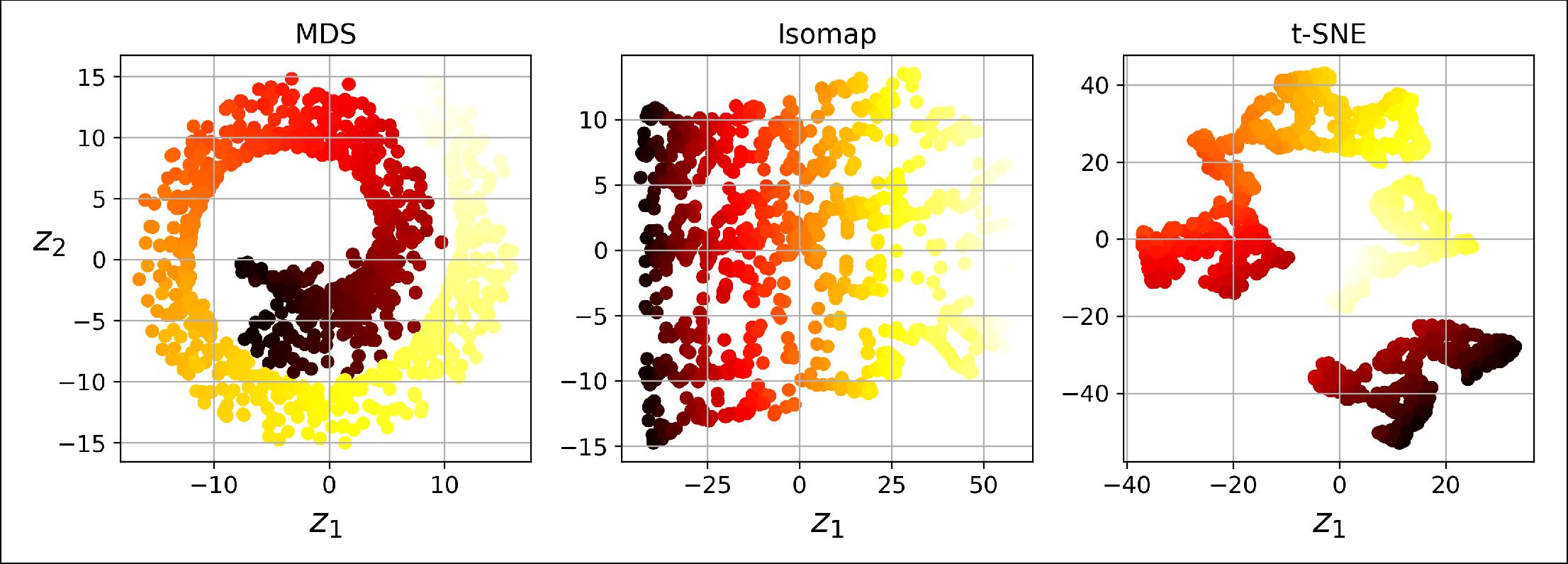
There are many other dimensionality reduction techniques, several of which are available in Scikit-Learn. Here are some of the most popular:

* *Multidimensional Scaling* (MDS) reduces dimensionality while trying to preservethe distances between the instances (see [Figure 8-13](#page21)).



**234** **|** **Chapter 8: Dimensionality Reduction**

* *Isomap* creates a graph by connecting each instance to its nearest neighbors, thenreduces dimensionality while trying to preserve the *geodesic distances*[9](#page21) between the instances.
* *t-Distributed Stochastic Neighbor Embedding* (t-SNE) reduces dimensionalitywhile 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 (e.g., to visualize the MNIST images in 2D).
* *Linear Discriminant Analysis* (LDA) is actually a classification algorithm, but dur‐ing training it learns the most discriminative axes between the classes, and these axes can then be used to define a hyperplane onto which to project the data. The benefit 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 such as an SVM classifier.



*Figure 8-13. Reducing the Swiss roll to 2D using various techniques*

**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?



1. The geodesic distance between two nodes in a graph is the number of nodes on the shortest path between these nodes.



**Exercises** **|** **235**

1. In what cases would you use vanilla PCA, Incremental PCA, Randomized PCA, or Kernel PCA?
2. How can you evaluate the performance of a dimensionality reduction algorithm on your dataset?
3. Does it make any sense to chain two different dimensionality reduction algo‐ rithms?
4. 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?
5. Use t-SNE to reduce the MNIST dataset down to two dimensions and plot the result using Matplotlib. You can use a scatterplot using 10 different colors to rep‐ resent each image’s target class. Alternatively, you can write colored digits at the location of each instance, 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 in ???.



**236** **|** **Chapter 8: Dimensionality Reduction**