



# AI in Biomedical Data

Dr. M.B. Khodabakhshi

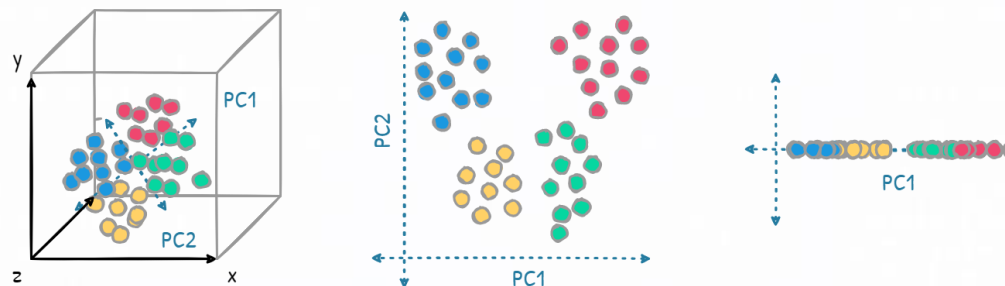
Amir Hossein Fouladi

Alireza Javadi



[github.com/mbkhodabakhshi/AI\\_in\\_BiomedicalData](https://github.com/mbkhodabakhshi/AI_in_BiomedicalData)

## Dimensionality Reduction



# یادگیری ماشین در زیست پزشکی

## Chapter 8. Dimensionality Reduction

دکتر محمدباقر خدابخشی

mb.khodabakhshi@gmail.com

# Curse of dimensionality

- Many Machine Learning problems involve **a lot 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**.
- Consider the MNIST images: 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.
- 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.

Apart from speeding up training, dimensionality reduction is also extremely useful for data visualization (or DataViz).

- ✓ 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.
- ✓ 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

# Curse of dimensionality

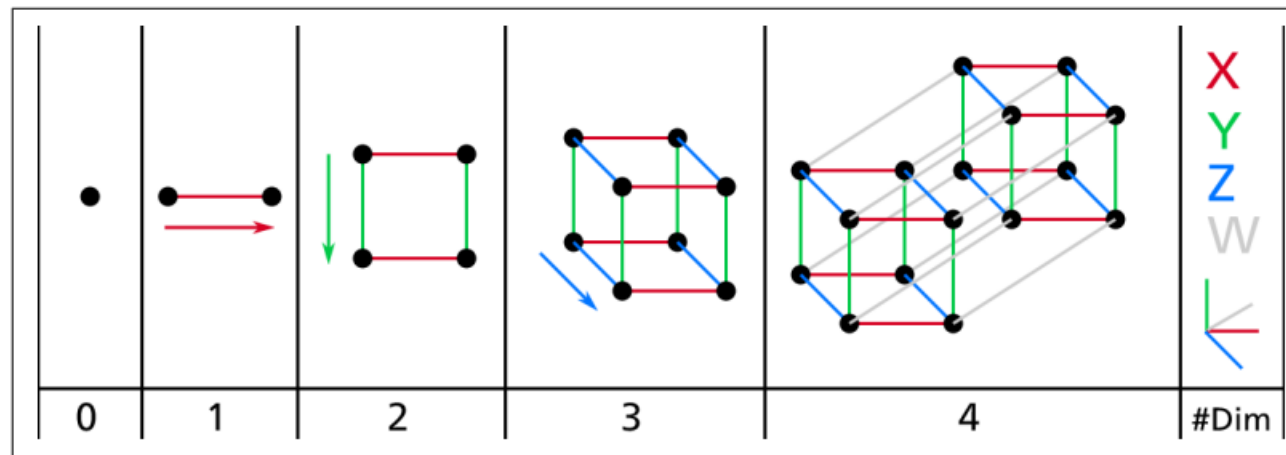


Figure 8-1. Point, segment, square, cube, and tesseract (0D to 4D hypercubes)<sup>2</sup>

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? The average distance, believe it or not, will be about 408.25 (roughly 1, 000, 000/6)!

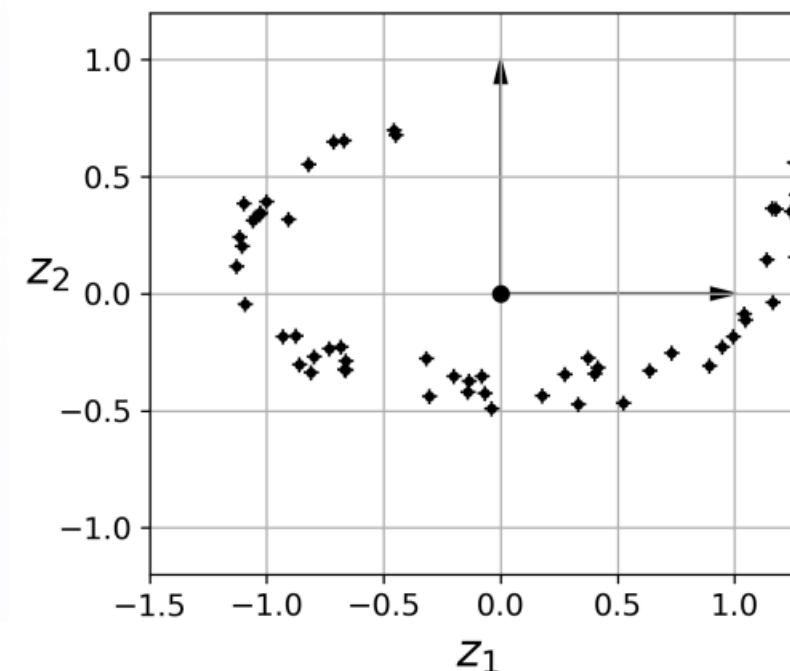
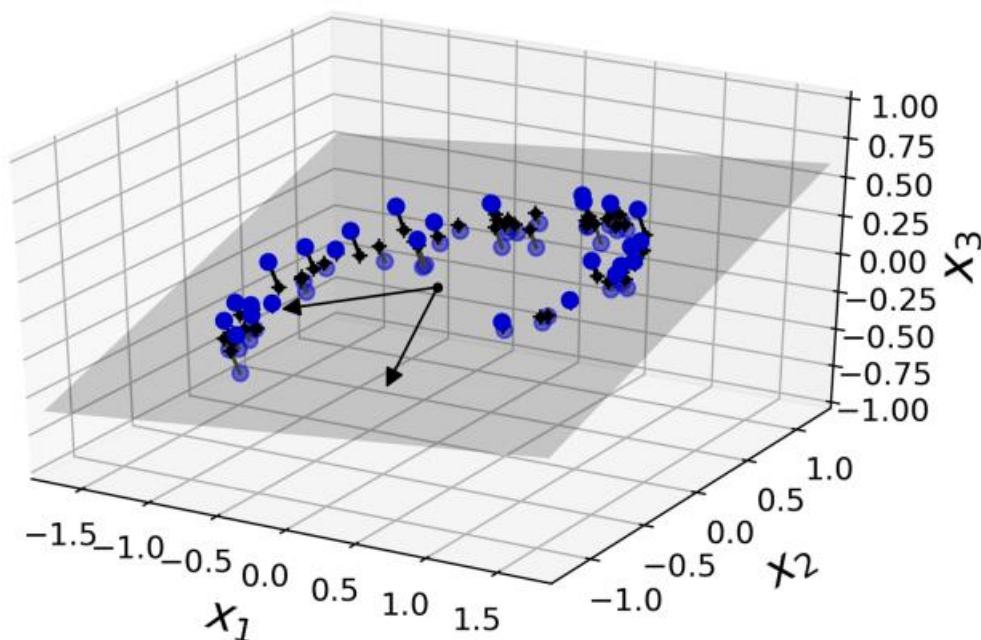
Well, there's just plenty of space in high dimensions. As a result, high-dimensional datasets are at risk of being very sparse: most training instances are likely to be far away from each other.

# 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

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.





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

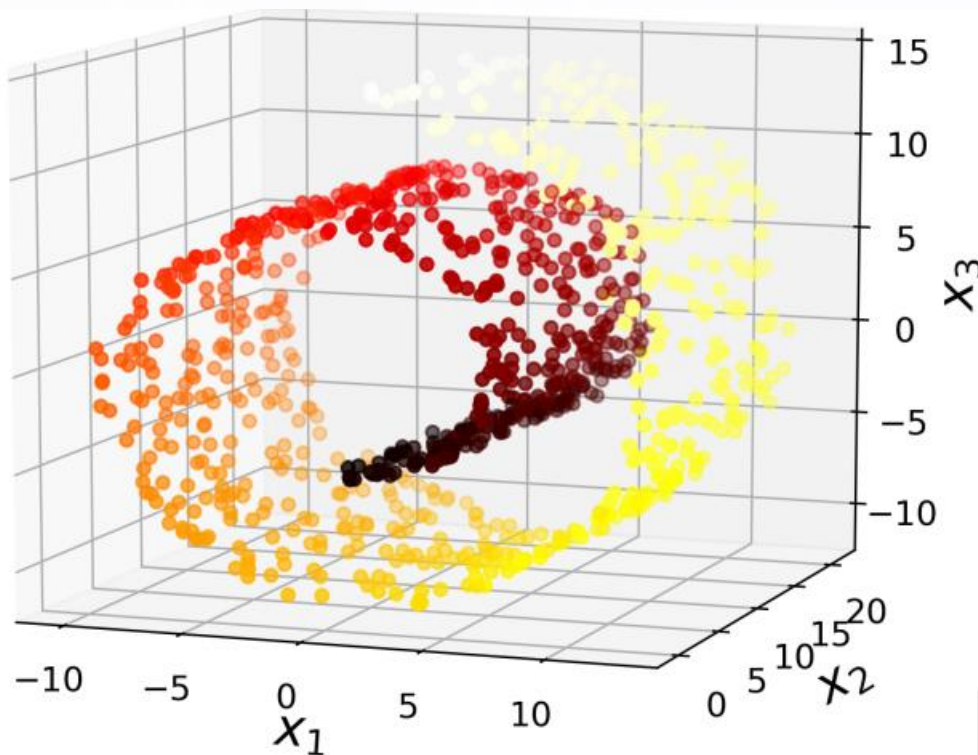


Figure 8-4. Swiss roll dataset

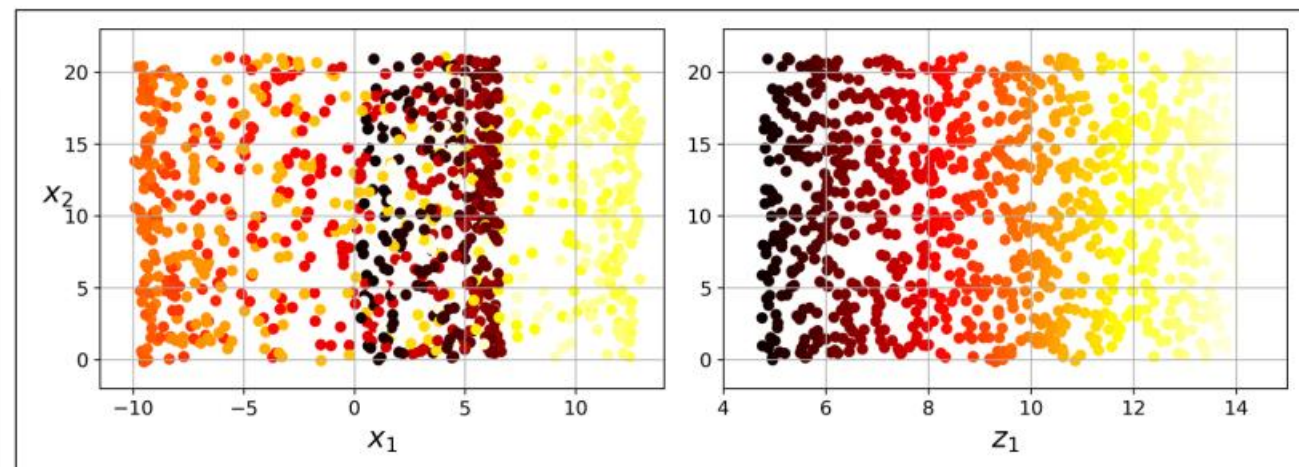
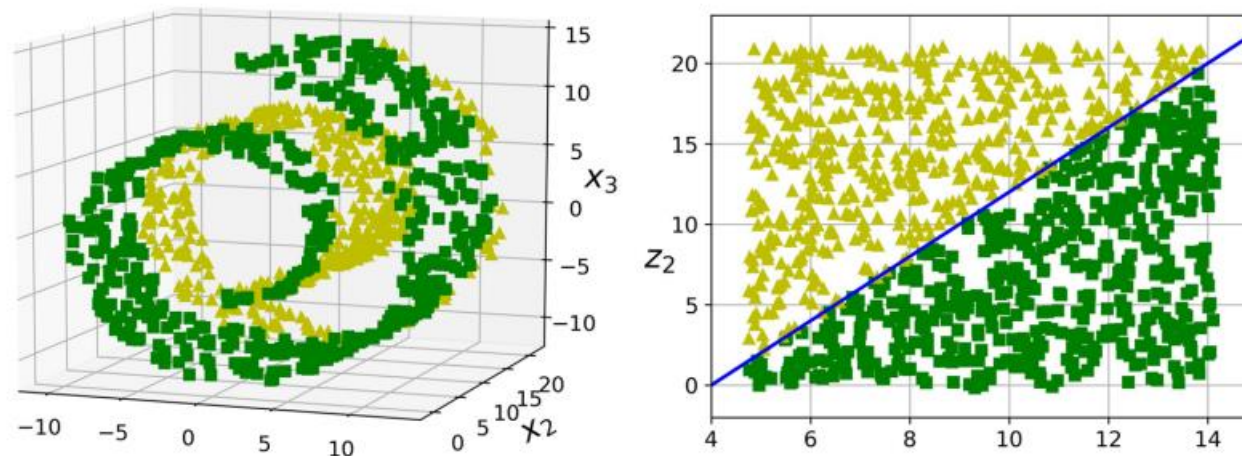


Figure 8-5. Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

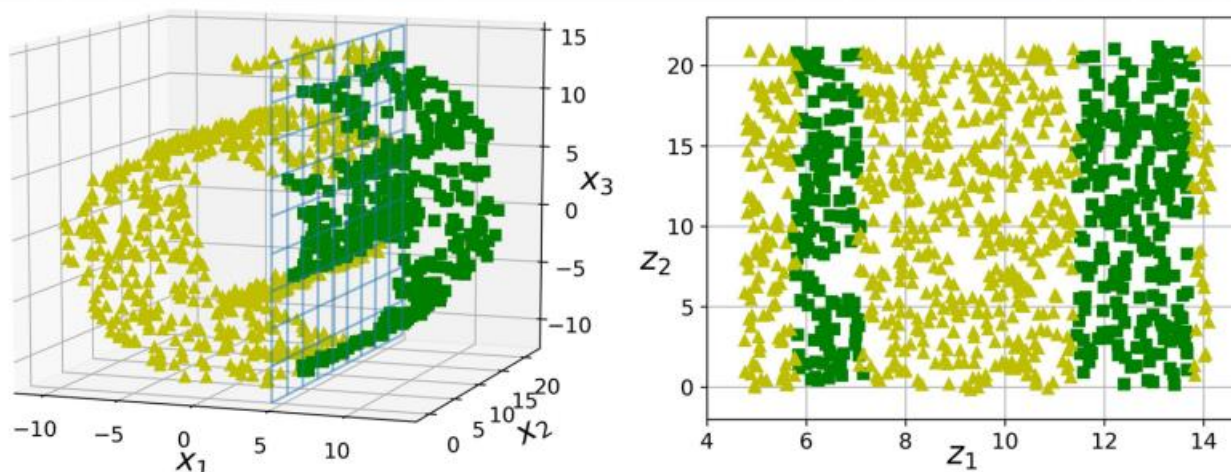
## Manifold Learning

- 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.
- Manifold Learning is based on the assumption that real-world data often lies on a lower-dimensional manifold embedded within a higher-dimensional space. This means that, despite appearing complex and high-dimensional, the data might be intrinsically simpler and lower-dimensional.
- 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. In the following figure, 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.

**Manifold learning** is a general concept that refers to a set of techniques used to uncover the underlying structure of high-dimensional data: PCA, LLE, t-SNE, and Isomap.





# Principal Component Analysis (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.

## Preserving the Variance

Before you can project the training set onto a lower-dimensional hyperplane, you first need to choose the right hyperplane.

You can see the result of the projection of the dataset onto each of three 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.

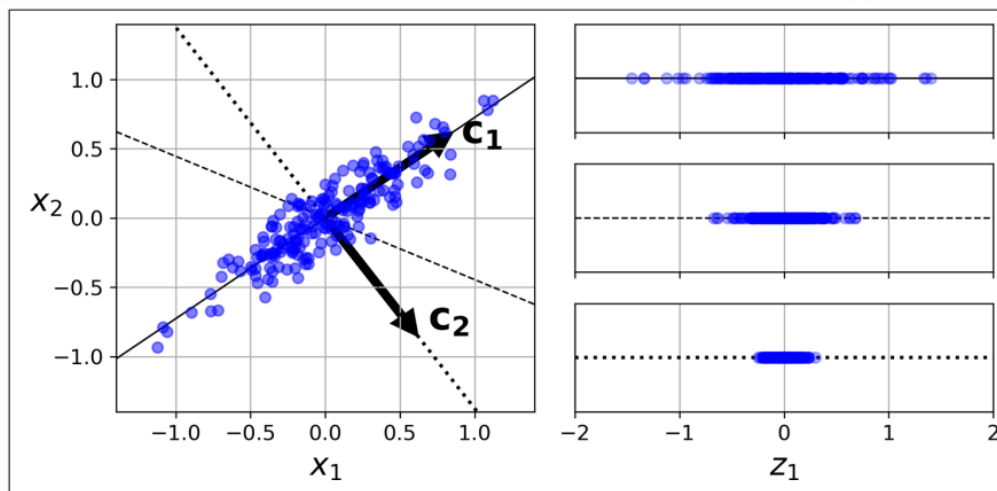


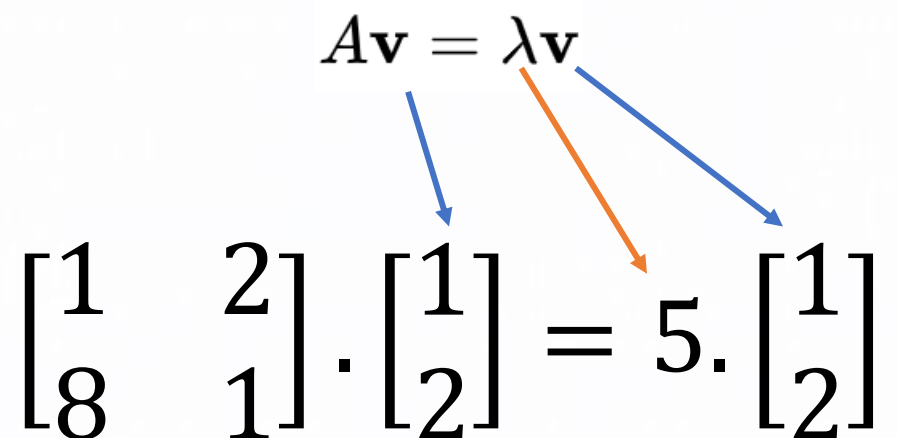
Figure 8-7. Selecting the subspace to project on

- 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

- ❑ 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.
- ❑ The  $i^{\text{th}}$  axis is called the  $i^{\text{th}}$  principal component (PC) of the data.

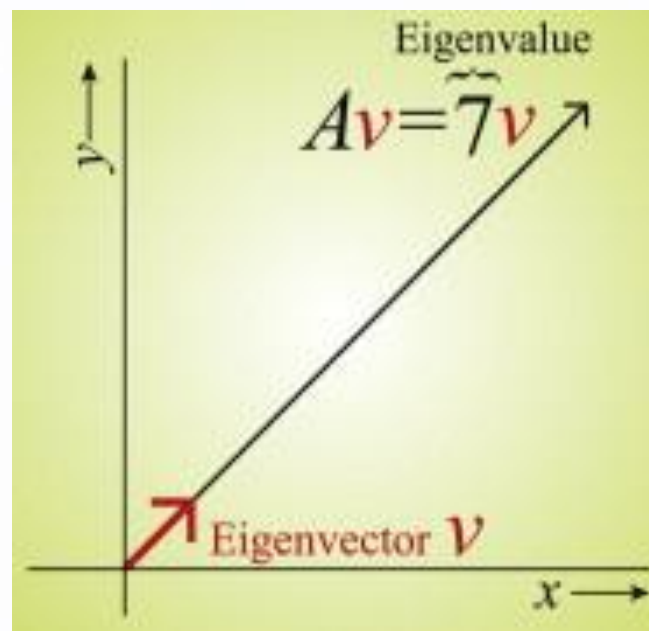
# Eigen Vectors and Eigen Values

Consider an  $n \times n$  matrix  $A$  and a nonzero vector  $\mathbf{v}$  of length  $n$ . If multiplying  $A$  with  $\mathbf{v}$  (denoted by  $A\mathbf{v}$ ) simply scales  $\mathbf{v}$  by a factor of  $\lambda$ , where  $\lambda$  is a **scalar**, then  $\mathbf{v}$  is called an eigenvector of  $A$ , and  $\lambda$  is the corresponding eigenvalue. This relationship can be expressed as:

$$A\mathbf{v} = \lambda\mathbf{v}$$

$$\begin{bmatrix} 1 & 2 \\ 8 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \end{bmatrix} = 5 \cdot \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

The eigenvector specified above is a vector that when multiplied by the matrix, the result is equal to the multiplication of the same eigenvector by a number such as the number 5 shown in the image above.

- ❑ Eigenvectors can determine the distribution direction of the data.
- ❑ Each eigenvector defines a new axis in the data space and the corresponding eigenvalue shows the importance of that axis.
- ❑ Eigenvalues and eigenvectors are very useful when:
  - ❑ Your goal is just to find the main directions of the scatter and you don't need other information like variance or dimensionality reduction.
  - ❑ When you are dealing with low-dimensional data and visualizing the results in the original space is not a problem.





$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

where, for each row,

$$w_i = A_{i1}v_1 + A_{i2}v_2 + \cdots + A_{in}v_n = \sum_{j=1}^n A_{ij}v_j.$$

If it occurs that  $\mathbf{v}$  and  $\mathbf{w}$  are scalar multiples, that is if

$$A\mathbf{v} = \mathbf{w} = \lambda\mathbf{v}, \quad (1)$$

then  $\mathbf{v}$  is an **eigenvector** of the linear transformation  $A$  and the scale factor  $\lambda$  is the **eigenvalue** corresponding to that eigenvector. Equation (1) is the **eigenvalue equation** for the matrix  $A$ .

Equation (1) can be stated equivalently as

$$(A - \lambda I)\mathbf{v} = \mathbf{0}, \quad (2)$$

where  $I$  is the  $n$  by  $n$  **identity matrix** and  $\mathbf{0}$  is the zero vector.

Equation (2) has a nonzero solution  $\mathbf{v}$  if and only if the determinant of the matrix  $(A - \lambda I)$  is zero. Therefore, the eigenvalues of  $A$  are values of  $\lambda$  that satisfy the equation

$$\det(A - \lambda I) = 0 \quad (3)$$

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}.$$

Taking the determinant of  $(A - \lambda I)$ , the characteristic polynomial of  $A$  is

$$\det(A - \lambda I) = \begin{vmatrix} 2 - \lambda & 1 \\ 1 & 2 - \lambda \end{vmatrix} = 3 - 4\lambda + \lambda^2.$$

Setting the characteristic polynomial equal to zero, it has roots at  $\lambda=1$  and  $\lambda=3$ , which are the two eigenvalues of  $A$ . The eigenvectors corresponding to each eigenvalue can be found by solving for the components of  $\mathbf{v}$  in the equation  $(A - \lambda I) \mathbf{v} = \mathbf{0}$ . In this example, the eigenvectors are any nonzero scalar multiples of

$$\mathbf{v}_{\lambda=1} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \quad \mathbf{v}_{\lambda=3} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

# Singular Value Decomposition (SVD)

The SVD decomposition of data is mathematically based on linear algebra, where the original matrix is decomposed into three matrices  $U$ ,  $\Sigma$ , and  $V^T$ . This decomposition follows these steps:

## 1. Calculating Singular Values and Eigenvectors

- First, we consider the matrix  $A$  (which often represents the data). If  $A$  is an  $m \times n$  matrix, the goal is to find the matrices  $U$  (an  $m \times m$  matrix),  $\Sigma$  (a diagonal  $m \times n$  matrix), and  $V$  (an  $n \times n$  matrix) such that:

$$A = U\Sigma V^T$$

## 2. Finding Right Singular Vectors $V$

- The matrix  $V$  consists of the right singular vectors of  $A$ , which can be obtained by calculating the eigenvalues and eigenvectors from the matrix  $A^T A$ .
- First, we compute the matrix  $A^T A$ . Then, by decomposing the eigenvalues and eigenvectors of  $A^T A$ , we obtain the primary directions of variance in the data as the columns of  $V$ . These eigenvalues also correspond to scales (magnitudes) that will later be placed in  $\Sigma$ .

### 3. Finding Left Singular Vectors $U$

- To compute the left singular vectors, we use the matrix  $AA^T$ . The eigenvectors of  $AA^T$  represent directions that  $A$  maps outputs towards. These eigenvectors are placed as columns in the matrix  $U$ .
- Similarly, the eigenvalues related to  $AA^T$  also contribute to the diagonal matrix  $\Sigma$ .

### 4. Constructing the Diagonal Matrix $\Sigma$

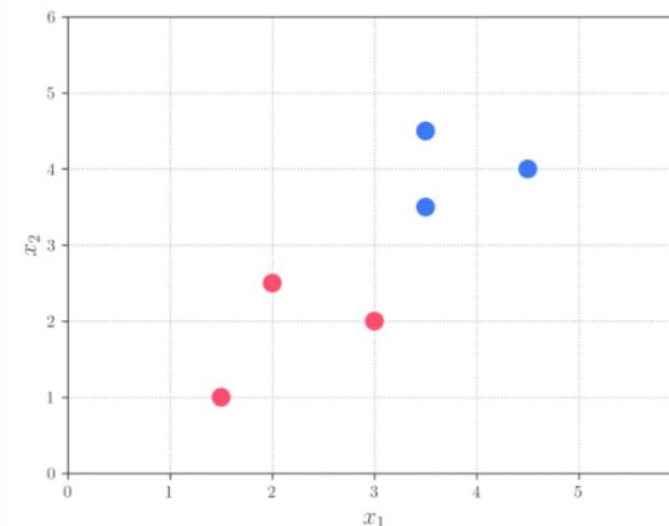
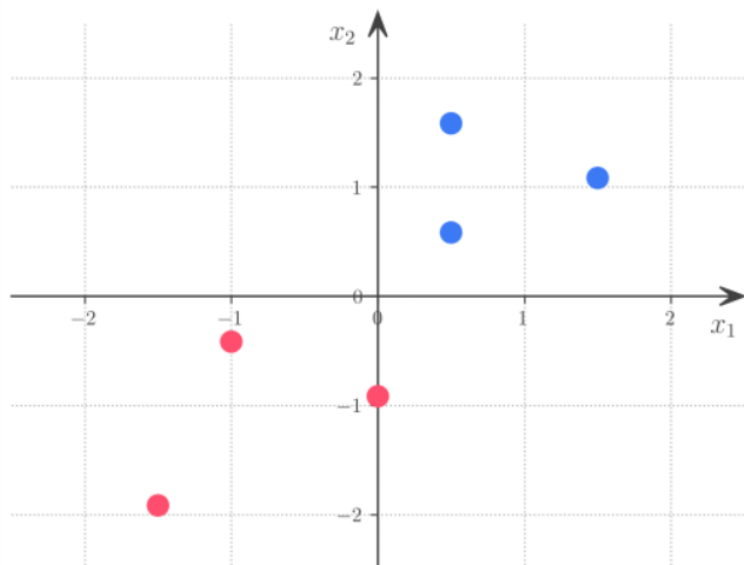
- The diagonal matrix  $\Sigma$  is composed of singular values, which are the square roots of the eigenvalues of  $A^T A$  or  $AA^T$ .
- These singular values indicate the importance of each principal direction and are usually arranged in  $\Sigma$  in descending order. If some singular values are very small, they can be discarded to enable data dimensionality reduction.



# Step-by-Step: How SVD is Used in PCA

## 1. Data Centering:

- First, in PCA, the data matrix  $X$  is centered. This means subtracting the mean of each feature from the respective feature values across all samples. After centering, the data matrix  $X$  has a mean of zero for each feature.



## 2. Calculating SVD on the Centered Data:

- Once centered, we apply SVD directly to  $X$ . If  $X$  has dimensions  $m \times n$  (where  $m$  is the number of samples, and  $n$  is the number of features), then we decompose  $X$  as:

$$X = U\Sigma V^T$$

- Here:
  - $U$  is an  $m \times m$  matrix whose columns are the left singular vectors of  $X$ .
  - $\Sigma$  is an  $m \times n$  diagonal matrix of singular values, which represent the magnitude of each principal component (direction of maximum variance).
  - $V$  is an  $n \times n$  matrix whose columns are the right singular vectors of  $X$ , corresponding to the principal directions in feature space.

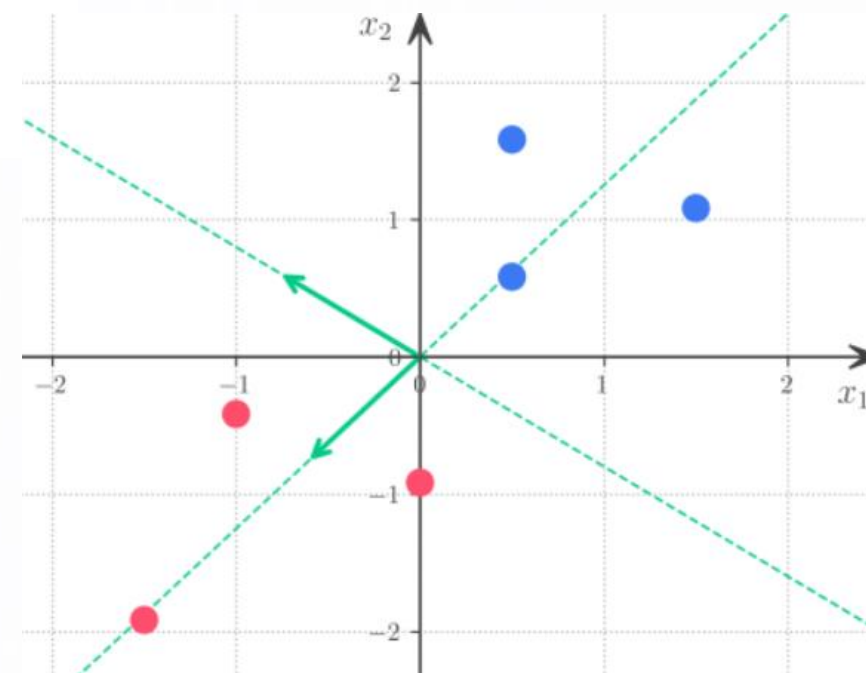
### 3. Principal Components from $V$ :

- The columns of  $V$  contain the principal components of  $X$  (i.e., the directions of maximum variance). These are the directions along which the data varies most, ordered by the magnitude of singular values in  $\Sigma$ .

### 4. Selecting Principal Components:

- To reduce the dimensionality of the data while preserving as much variance as possible, we select the first  $k$  columns of  $V$  (where  $k$  is the number of principal components we want to retain).
- The corresponding singular values in  $\Sigma$  indicate the "importance" or "weight" of each component, allowing us to decide how many components are necessary to capture most of the variance.

$$\lambda = [0.24 \quad 2.7], V = \begin{bmatrix} -0.78 & -0.62 \\ 0.62 & -0.78 \end{bmatrix}$$



## 5. Projecting the Data onto the Principal Components:

- By multiplying the centered data matrix  $X$  by the first  $k$  columns of  $V$ , we project  $X$  onto a lower-dimensional space defined by these principal components:

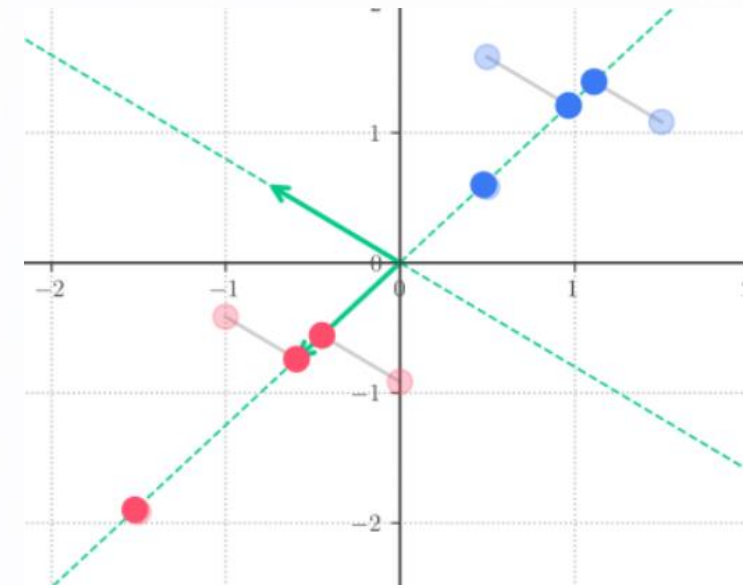
$$X_{PCA} = X \cdot V_k$$

where  $V_k$  contains the first  $k$  principal components.

- This results in a transformed dataset  $X_{PCA}$  with reduced dimensionality  $k$  but retaining most of the variance in the data.

$$V = \begin{bmatrix} -0.78 & -0.62 \\ 0.62 & -0.78 \end{bmatrix} \quad X = \begin{bmatrix} -1.5 & -1.9 \\ -1.0 & -0.4 \\ 0.0 & -0.9 \\ 0.5 & 0.6 \\ 0.5 & 1.6 \\ 1.5 & 1.1 \end{bmatrix}, \quad \mathbf{v} = \begin{bmatrix} -0.62 \\ -0.78 \end{bmatrix} \rightarrow X_p = X\mathbf{v}$$

$$X_p = [2.43 \quad 0.95 \quad 0.72 \quad -0.77 \quad -1.55 \quad -1.78]$$





# Projecting Down to $d$ Dimensions

The following Python code uses NumPy's `svd()` function to obtain all the principal components of the training set, then extracts the two unit vectors that define 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]
```

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.T[:, :2]
X2D = X_centered.dot(W2)
```

## Using Scikit-Learn

Scikit-Learn's PCA class uses SVD decomposition to implement PCA, just like we did earlier in this chapter. 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, its `components_` attribute holds the transpose of  $\mathbf{W}_d$  (e.g., the unit vector that defines the first principal component is equal to `pca.components_.T[:, 0]`).

## 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([0.84248607, 0.14631839])
```

This output tells you that 84.2% of the dataset's variance lies along the first PC, and 14.6% lies along the second PC. This leaves less than 1.2% 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 (e.g., 95%). Unless, of course, you are reducing dimensionality for data visualization—in that case you will want to reduce the dimensionality down to 2 or 3.

The following code performs 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. But 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. In this case, you can see that reducing the dimensionality down to about 100 dimensions wouldn't lose too much explained variance.

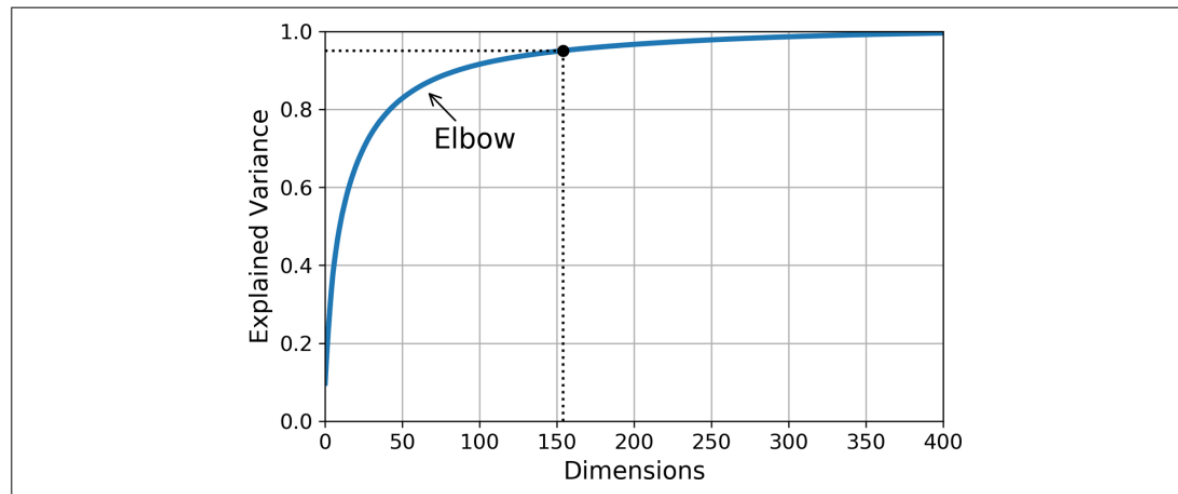


Figure 8-8. Explained variance as a function of the number of dimensions

# PCA for Compression

- ❖ After dimensionality reduction, the training set takes up much less space. As an 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, this is a reasonable compression ratio, and you can see how this size reduction 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.

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

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

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

The mean squared distance between the original data and the reconstructed data (compressed and then decompressed) is called the *reconstruction error*.

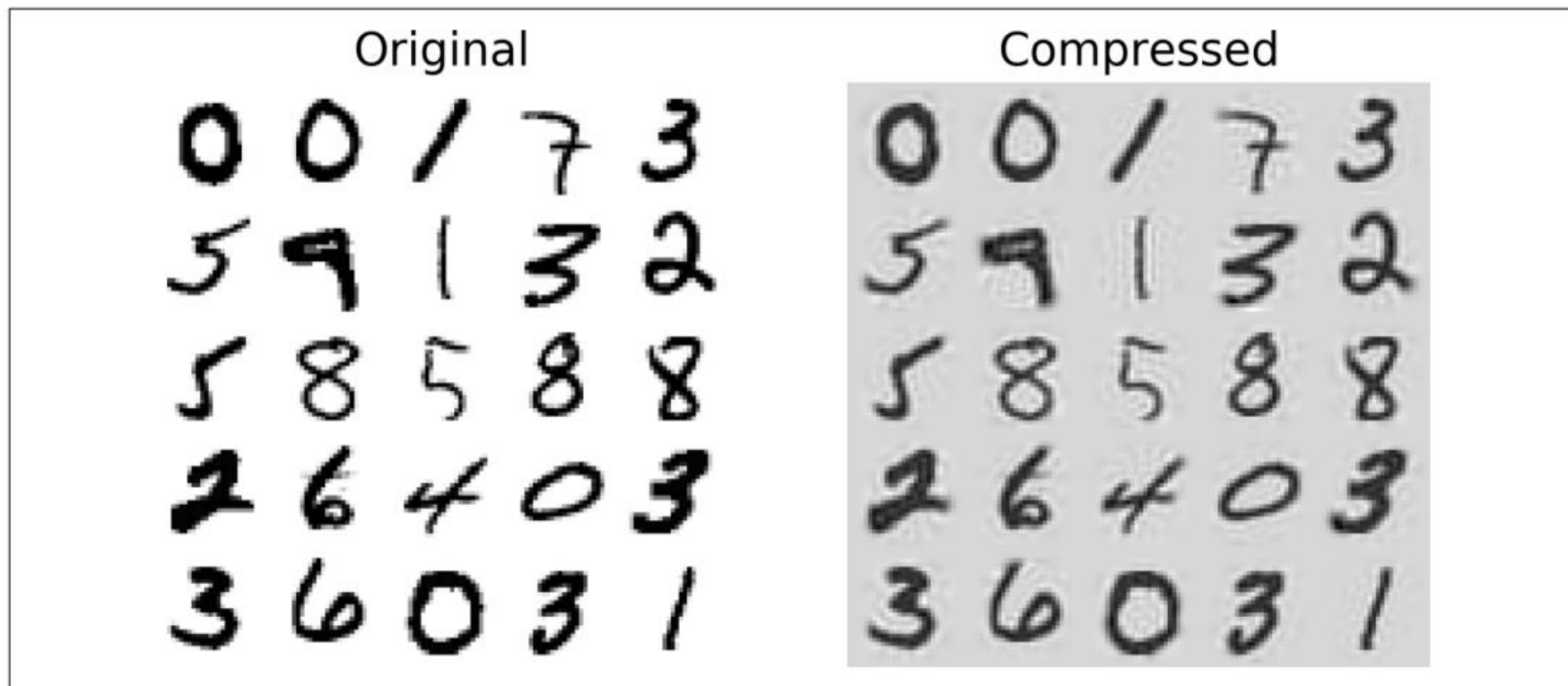


Figure 8-9. MNIST compression that preserves 95% of the variance

# 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 PCA* (kPCA).
- It is often good at preserving clusters of instances after projection, or sometimes even unrolling datasets that lie close to a twisted manifold.

```
from sklearn.decomposition import KernelPCA

rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

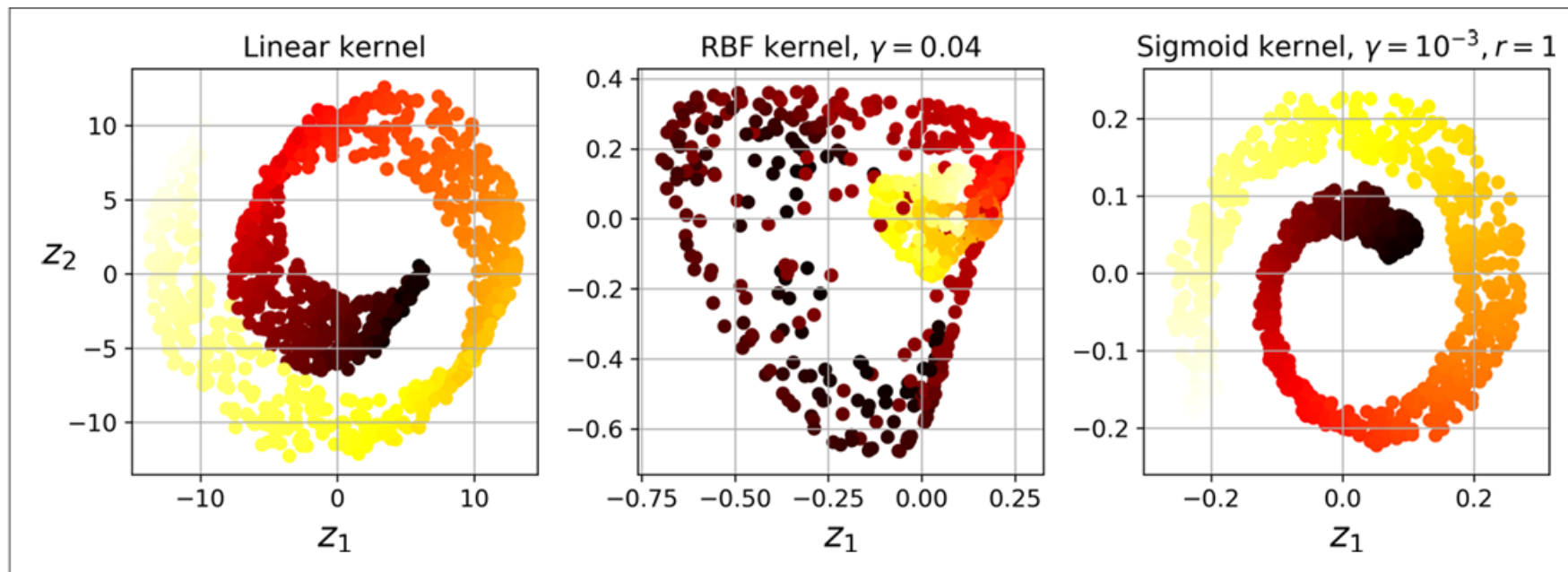


Figure 8-10. Swiss roll reduced to 2D using kPCA with various kernels



- In Kernel PCA, data is first **mapped into a higher-dimensional feature space via a nonlinear transformation**.
- **Dimensionality reduction** is then performed in this feature space (also called the kernel space). The goal of this nonlinear mapping is to transform complex, nonlinear relationships into simpler, nearly linear relationships in the kernel space.
- This nonlinear mapping is achieved through a **kernel function**, allowing us to compute the inner products between samples in the high-dimensional space without directly calculating the coordinates in that space. This approach is known as the **kernel trick**.

## Steps of Kernel PCA

### 1. Selecting a Kernel Function and Calculating the Kernel Matrix:

- A suitable kernel function (such as Gaussian, polynomial, or linear kernel) is chosen. This kernel function  $k(x_i, x_j)$  computes the similarity between each pair of samples  $x_i$  and  $x_j$  in the kernel space.
- Next, the kernel matrix  $K$  is calculated, where each element  $K_{ij} = k(x_i, x_j)$  represents the similarity between samples  $x_i$  and  $x_j$ .

## 2. Centering the Kernel Matrix:

- To perform PCA in the kernel space, the kernel matrix must be centered so that the mean of the data in this space is zero. This step involves centering  $K$ .

## 3. Eigenvalue Decomposition of the Kernel Matrix:

- An eigenvalue decomposition is then applied to the centered kernel matrix to find the principal directions in the kernel space. The eigenvectors of the kernel matrix represent the principal directions (or nonlinear principal components), and the eigenvalues indicate the significance of each direction.

## 4. Selecting Principal Components and Reducing Dimensions:

- By selecting a set number of principal components (e.g., the top  $k$  eigenvectors with the largest eigenvalues), the data can be reduced to a new, lower-dimensional space. This new space preserves the most important information while maintaining the nonlinear structure of the data.

# 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.
- ❖ 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 GridSearchCV 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

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"]
}]

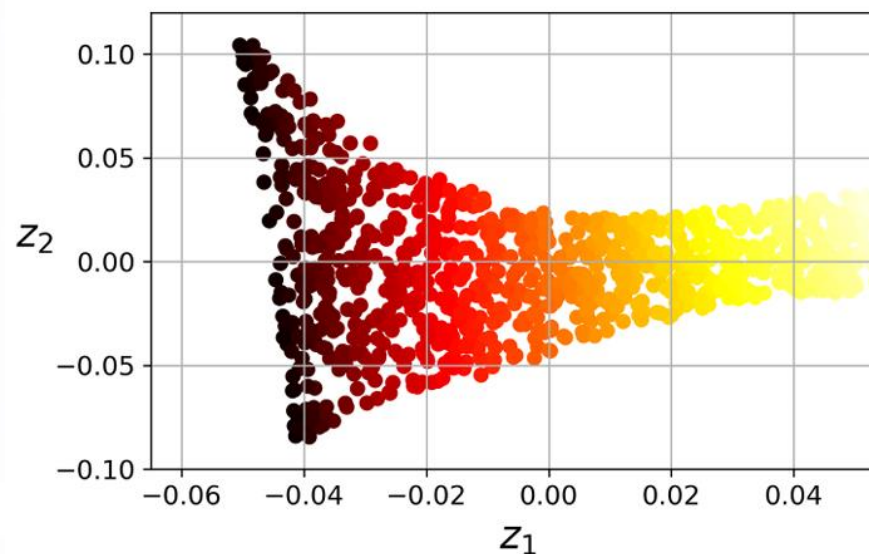
grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
```

# Locally Linear Embedding (LLE)

- It is another powerful *nonlinear dimensionality reduction* (NLDR) technique.
- It is a **Manifold Learning** technique that does not rely on **projections**, like the previous algorithms do.
- LLE works by first **measuring** how each training instance **linearly relates** to its **closest neighbors**, and then **looking for a low-dimensional representation** of the training set where these local relationships are best preserved.

```
from sklearn.manifold import LocallyLinearEmbedding
```

```
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)  
X_reduced = lle.fit_transform(X)
```



Here's how LLE works: for each training instance  $\mathbf{x}^{(i)}$ , the algorithm identifies its  $k$  closest neighbors (in the preceding code  $k = 10$ ), then tries to reconstruct  $\mathbf{x}^{(i)}$  as a linear function of these neighbors. More specifically, it finds 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  $k$  closest 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 one: linearly modeling local relationships*

$$\begin{aligned} \widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \quad & \sum_{i=1}^m \left( \mathbf{x}^{(i)} - \sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)} \right)^2 \\ \text{subject to} \quad & \begin{cases} w_{i,j} = 0 & \text{if } \mathbf{x}^{(j)} \text{ is not one of the } k \text{ c.n. of } \mathbf{x}^{(i)} \\ \sum_{j=1}^m w_{i,j} = 1 & \text{for } i = 1, 2, \dots, m \end{cases} \end{aligned}$$



After this step, the weight matrix  $\widehat{\mathbf{W}}$  (containing the weights  $\widehat{w}_{i,j}$ ) 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 two: reducing dimensionality while preserving relationships*

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