SAIF

Shanghai Advanced Institute of Finance
上海高级金融学院

量化俱乐部-机器学习-DimensionalityReduction

2019-08-31

Menu

1. The Curse of Dimensionality
2. Main Approaches for Dimension Reduction
3. PCA
4. Kernal PCA
5. LLE
6. Clustering-KNN

自我介绍

张骁喆，2010年大连理工大学软件学院本科毕业，高金FMBA 2017PTE。9年计算机工作经验，熟悉开发，测试，运维，项目管理，产品设计各环节。
曾就职eBay，唯品会，2016加入SAP，现担任SAP Cloud部门开发团队主管。
2017开始接触量化投资和python，目前毕业论文研究方向使用机器学习在A股进行量化投资。
量化投资/机器学习咨询培训/项目管理产品管理。

Intro

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.

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 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

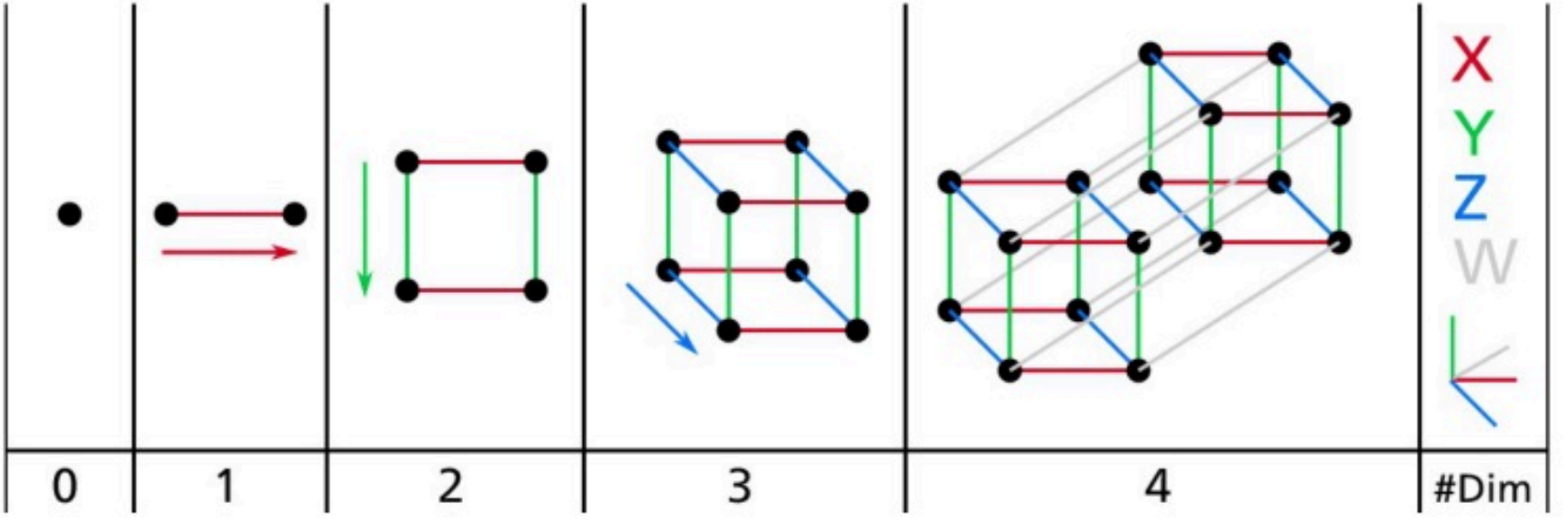


Figure 8-1. Point, segment, square, cube, and tesseract (0D to 4D hypercubes)[2]

The Curse of Dimensionality

It turns out that many things behave very differently in high-dimensional Space:

For example, 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% chance of being located less than 0.001 from a border. Most points in a high-dimensional hypercube are very close to the border. 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.

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. 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

The Curse of Dimensionality

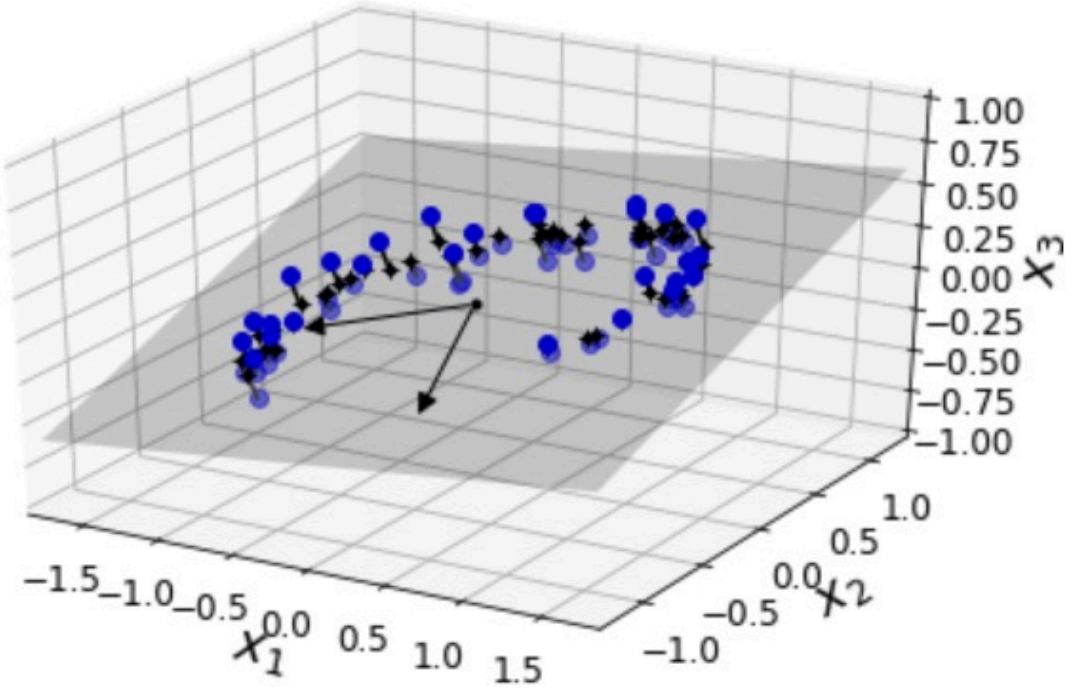
In theory, one solution to the curse of dimensionality could be to increase the size of the training set to reach a sufficient density of training instances. Unfortunately, in practice, the number of training instances required to reach a given density grows exponentially with the number of dimensions. With just 100 features (much less than 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**.

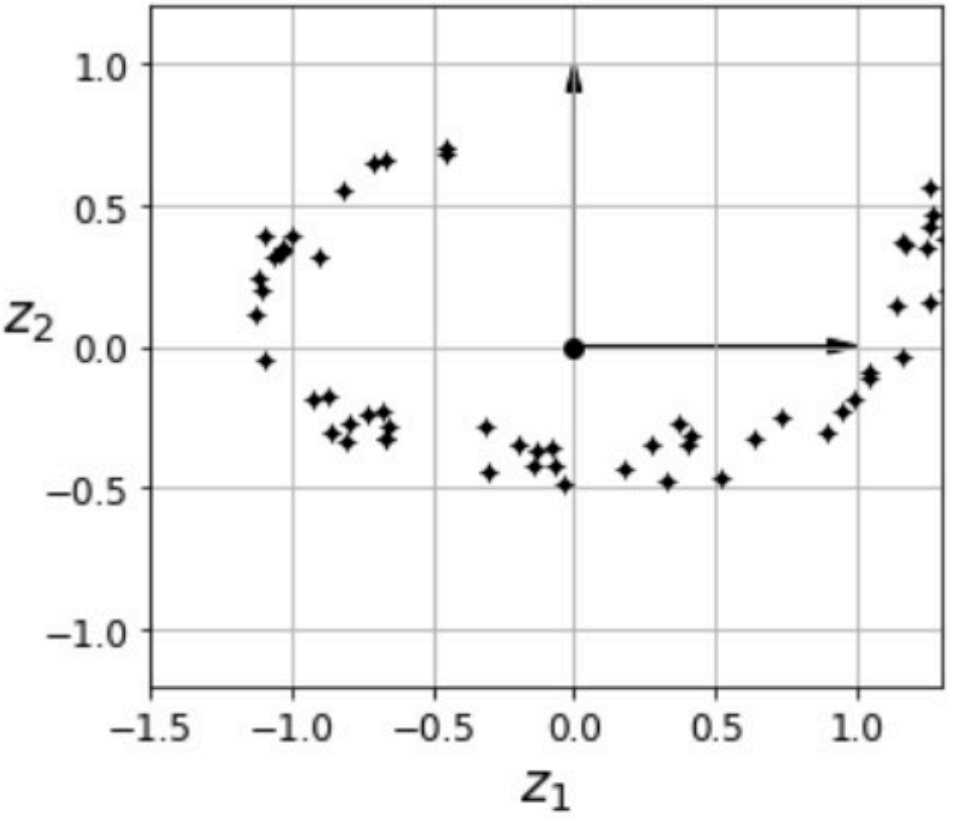
Main Approaches - Projection

you can see a 3D dataset represented by the circles. Notice that all training instances lie close to a plane: this is a lowerdimensional (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



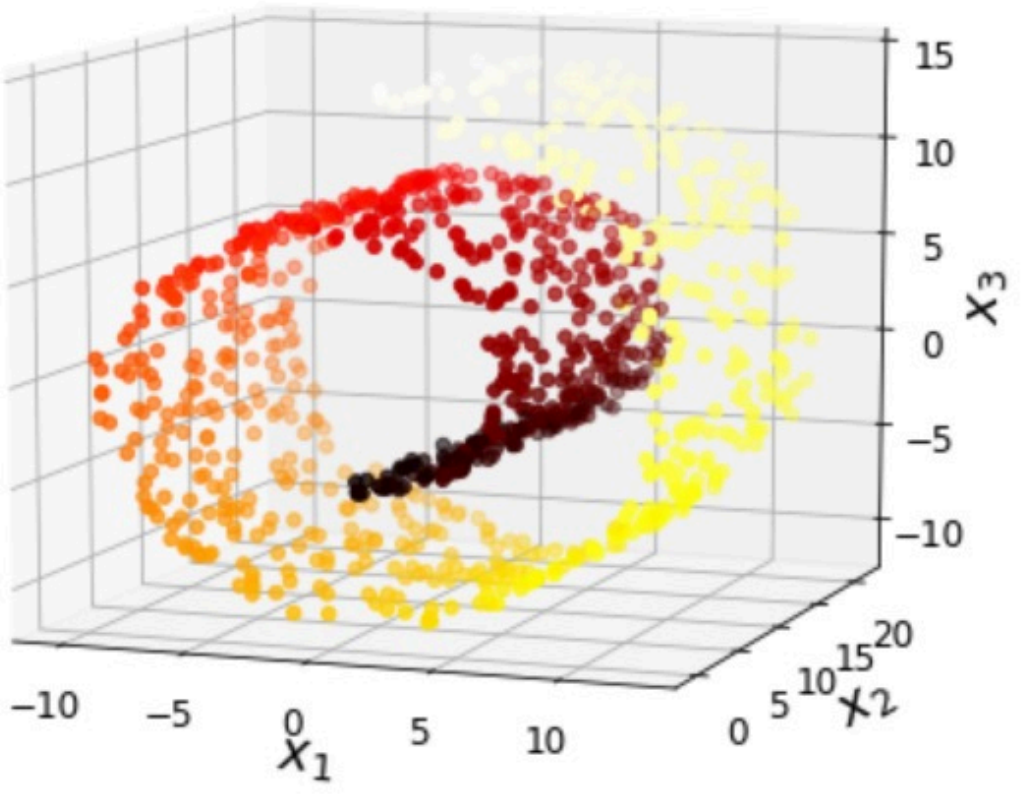
Main Approaches - Projection

We have just reduced the dataset' s dimensionality from 3D to 2D. Note that the axes correspond to new features z1 and z2 (the coordinates of the projections on the plane)



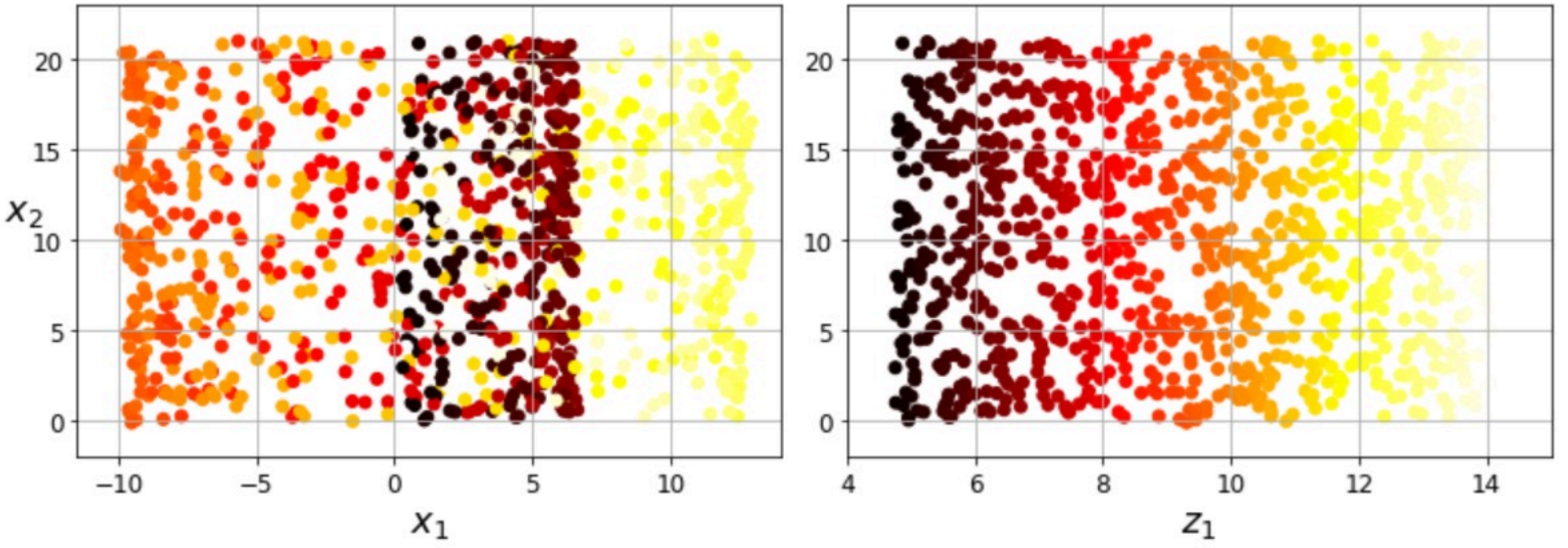
Main Approaches - Projection

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



Main Approaches - Projection

Simply projecting onto a plane (e.g., by dropping x3) 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.



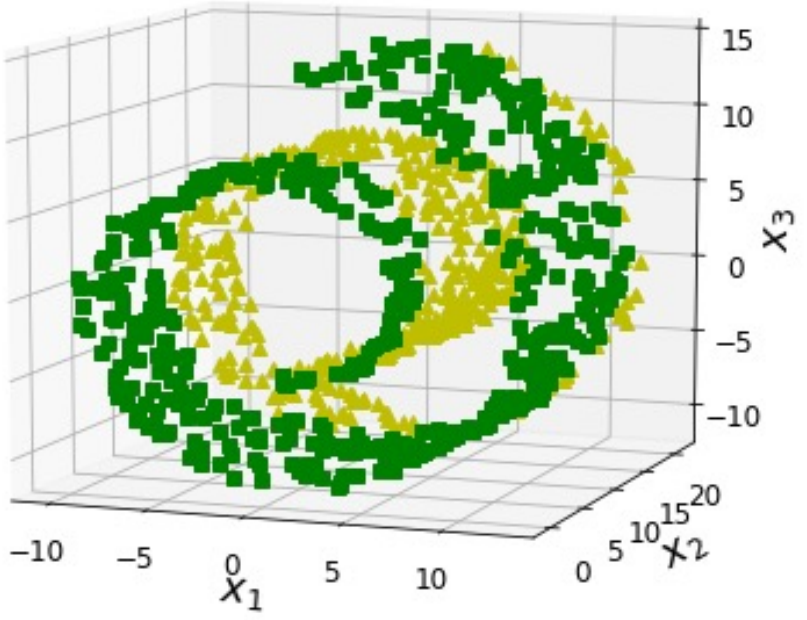
Main Approaches – Manifold Learning

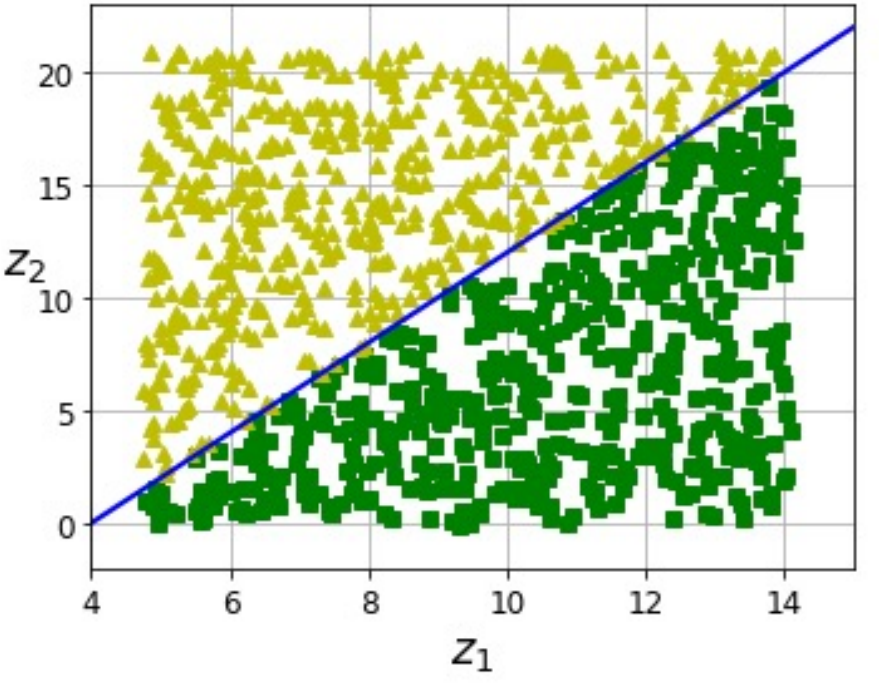
The Swiss roll is an example of a 2D manifold. Put simply, a 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space. More generally, a d-dimensional manifold is a part of an n-dimensional space (where d < n) that locally resembles a d-dimensional hyperplane. In the case 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.

Main Approaches – Manifold Learning

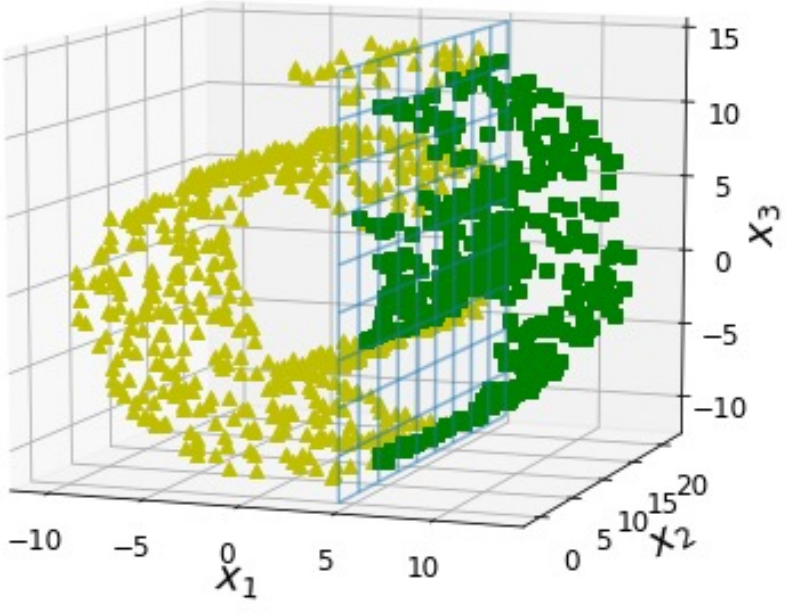
The manifold assumption is often accompanied by another implicit assumption: that the task at hand (e.g., classification or regression) will be simpler if expressed in the lower-dimensional space of the manifold. For example, in the top row of Figure the Swiss roll is split into two classes: in the 3D 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.

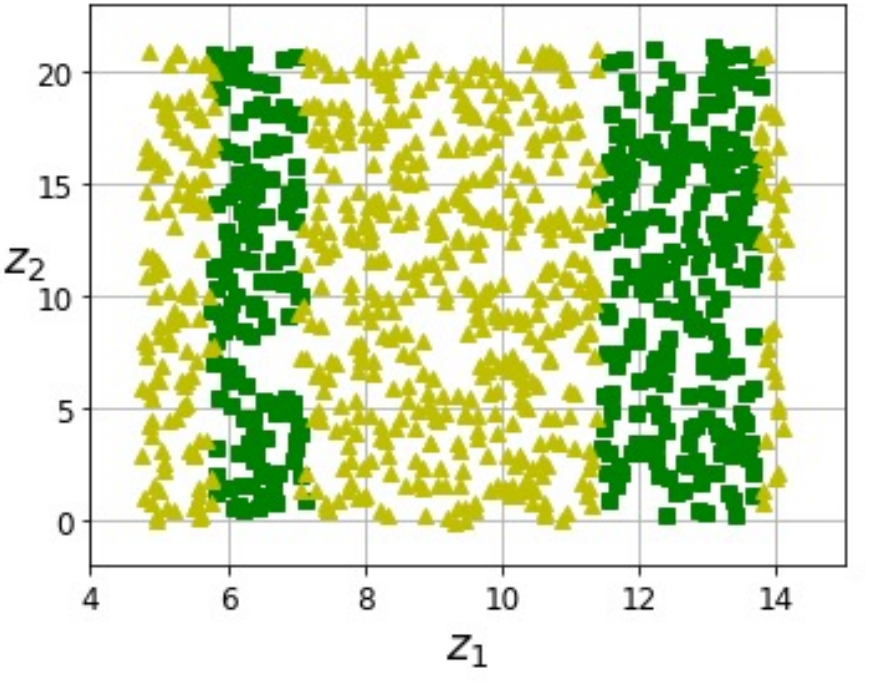




Main Approaches – Manifold Learning

However, this assumption does not always hold. For example, in the bottom row, the decision boundary is located at x1 = 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)





Main Approaches – Manifold Learning

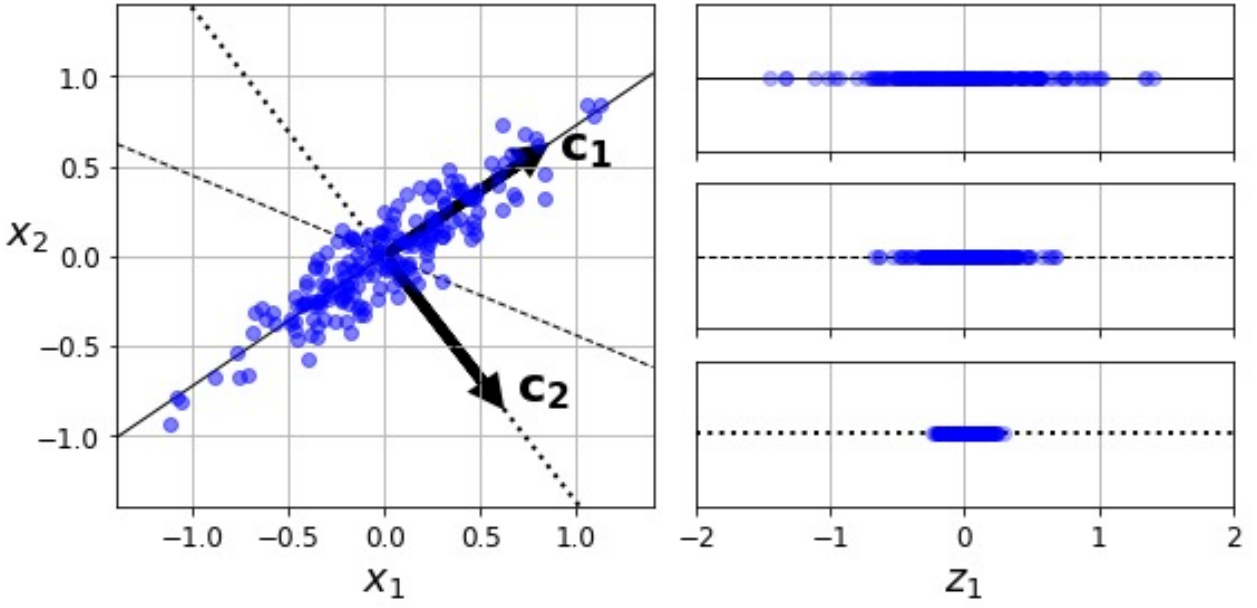
In short, if you reduce the dimensionality of your training set before training a model, it will definitely speed up training, but it may not always lead to a better or simpler solution; it all depends on the dataset

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.

PCA - 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, along with three different axes. 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.



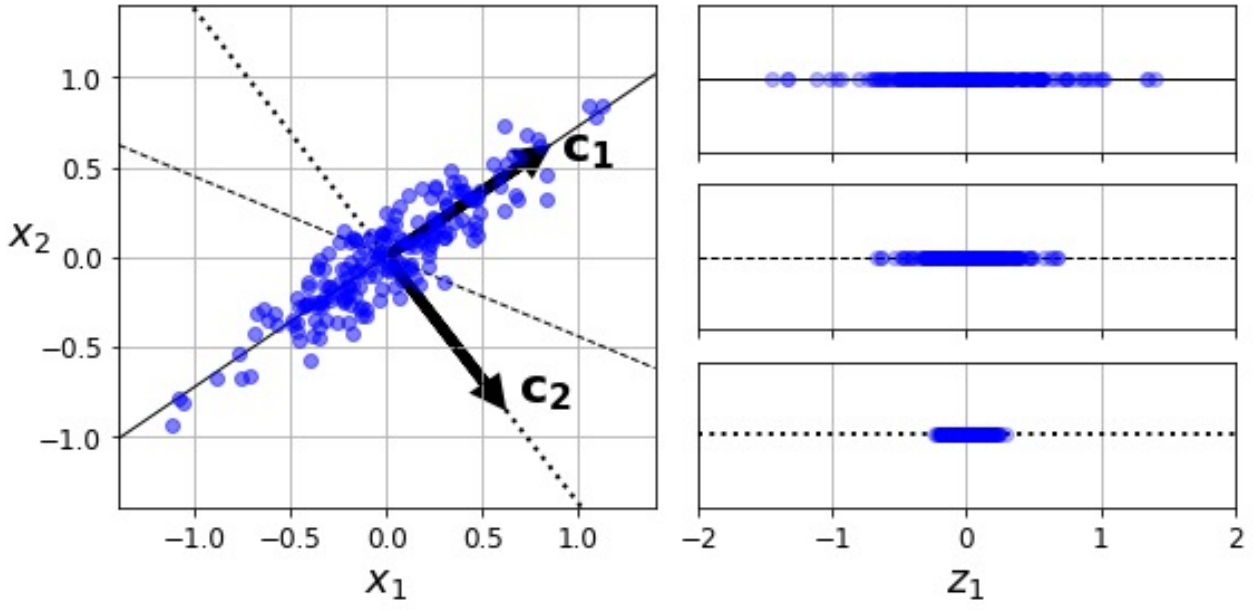
PCA - Preserving the Variance

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.

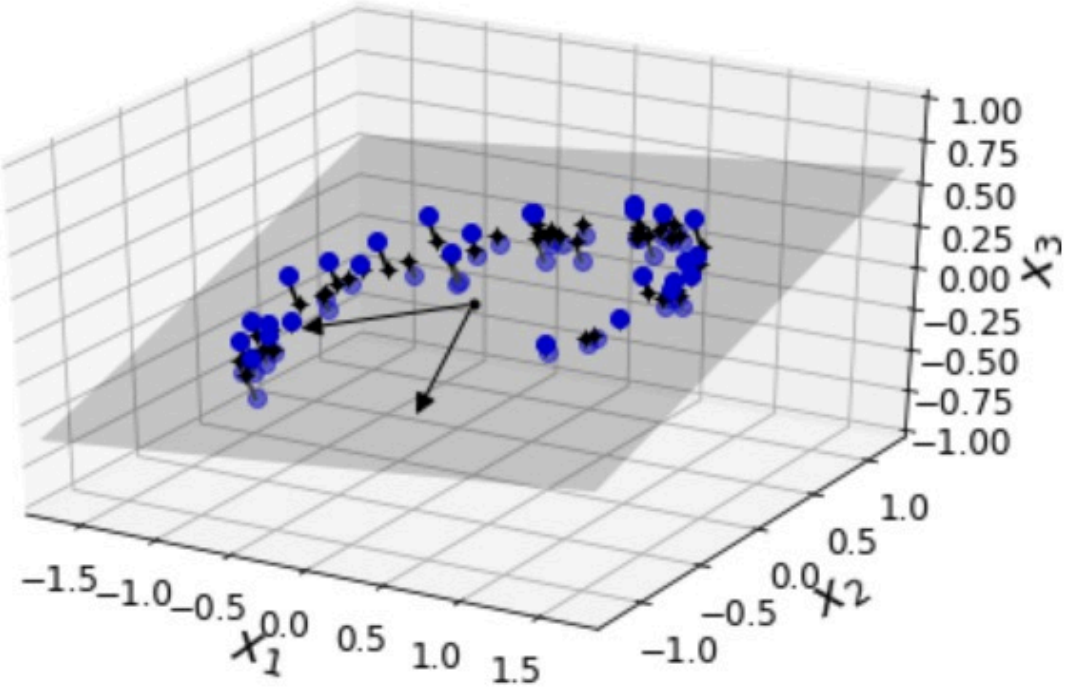
PCA - Principal Components

PCA identifies the axis that accounts for the largest amount of variance in the training set. It is the solid line below. 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.



PCA - Principal Components

The unit vector that defines the i-th axis is called the i-th principal component (PC). In Figure P-20, the 1st PC is c1 and the 2nd PC is c2. In Figure P-9 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).



PCA - Principal Components

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 dot product of three matrices $U \cdot \Sigma \cdot VT$, where VT contains all the principal components that we are looking for, as shown below:

$$\mathbf{V} = \begin{pmatrix} | & | & & | \\ \mathbf{c_1} & \mathbf{c_2} & \cdots & \mathbf{c_n} \\ | & | & & | \end{pmatrix}$$

PCA - Principal Components

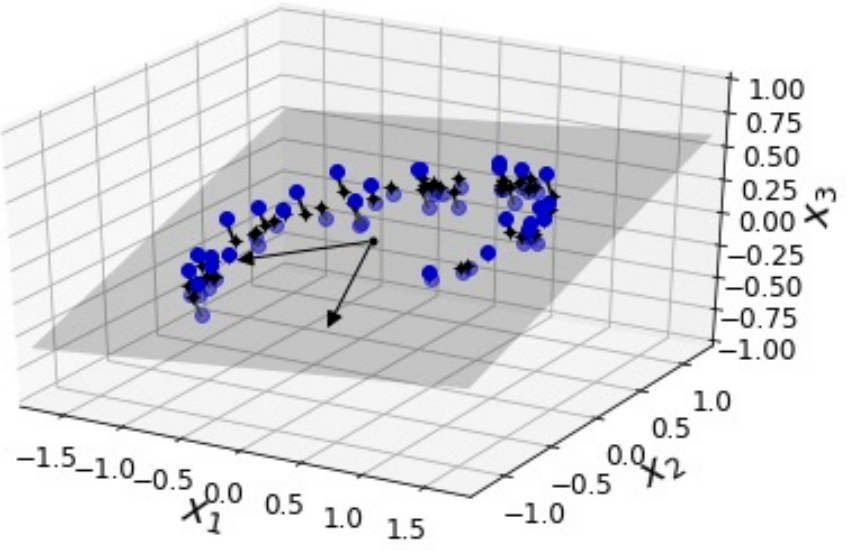
The following Python code uses NumPy’ s svd() function to obtain all the principal components of the training set, then extracts the first two PCs

```
In [45]: X_centered = X - X.mean(axis=0)
         U, s, Vt = np.linalg.svd(X_centered)
         c1 = Vt.T[:, 0]
         c2 = Vt.T[:, 1]
         Vt.T.shape

Out[45]: (3, 3)
```

PCA - 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, below 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



PCA - Projecting Down to d-Dimensions

To project the training set onto the hyperplane, you can simply compute the dot product of the training set matrix X by the matrix Wd
, defined as the matrix containing the first d principal components (i.e., the matrix composed of the first d columns of VT), as shown below:

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

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

PCA - Projecting Down to d-Dimensions

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

```
In [6]: W2 = Vt.T[:, :2]
        X2D = X_centered.dot(W2)

In [7]: X2D_using_svd = X2D
```

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!

PCA - 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

```
In [9]: 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 vectors, so, for example, the first principal component is equal to pca.components_.T[:, 0])

```
In [47]: from sklearn.decomposition import PCA

         pca = PCA(n_components = 2)
         X2D = pca.fit_transform(X)
         pca.components_.T[:, 0]

Out[47]: array([-0.93636116, -0.29854881, -0.18465208])
```

PCA - 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 P-24

```
In [20]: # Now let's look at the explained variance ratio:
         pca.explained_variance_ratio_
         # The first dimension explains 84.2% of the variance, while the second explains 14.6%.

Out[20]: array([0.84248607, 0.14631839])

In [21]: # By projecting down to 2D, we lost about 1.1% of the variance:

         1 - pca.explained_variance_ratio_.sum()

Out[21]: 0.011195535570688975
```

PCA - Choosing 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

PCA - Choosing Right Number of Dimensions

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

In [51]:
```
pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1

d
```

Out[51]: 154

PCA - Choosing Right Number of Dimensions

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:

```
In [54]: pca = PCA(n_components=0.95)
         X_reduced = pca.fit_transform(X_train)

In [55]: pca.n_components_
Out[55]: 153

In [56]: np.sum(pca.explained_variance_ratio_)
Out[56]: 0.9500608823439146
```

PCA for Compression

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.

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

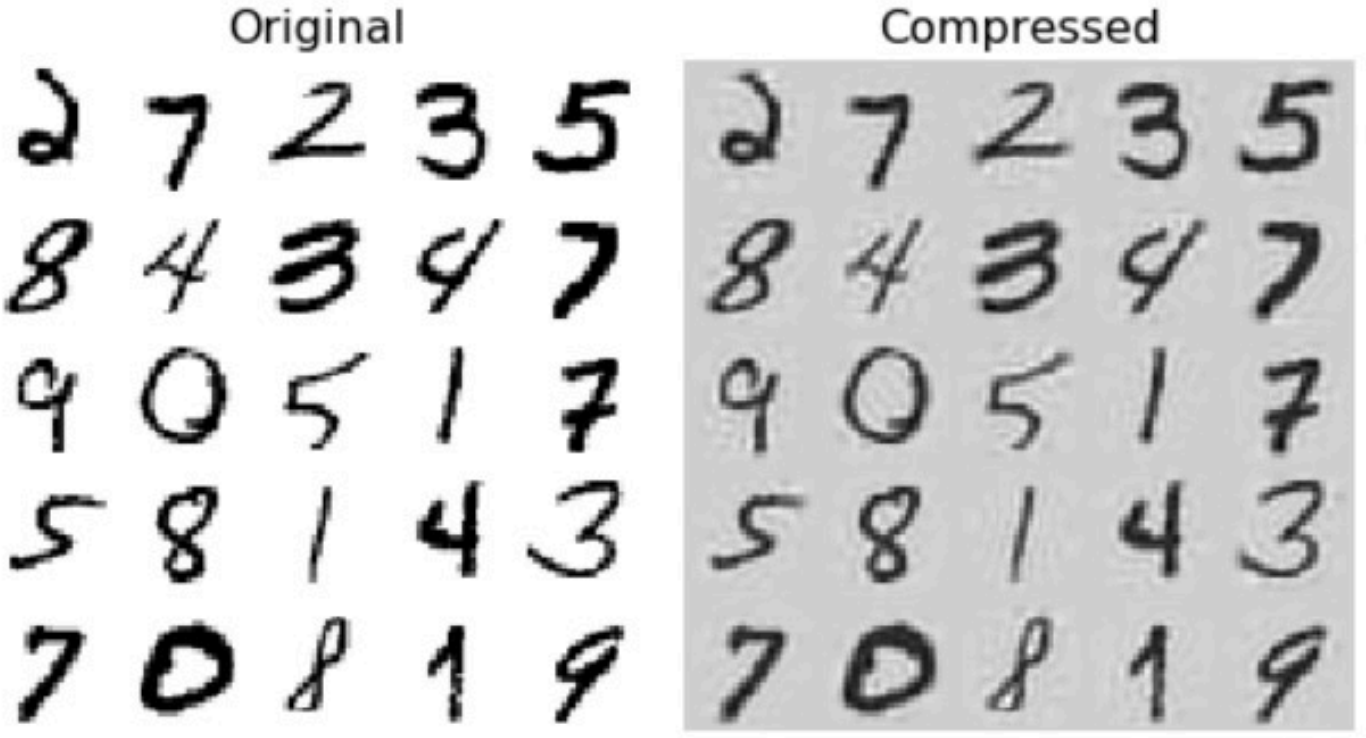
PCA for Compression

The following code compresses the MNIST dataset down to 154 dimensions, then uses the inverse_transform() method to decompress it back to 784 dimensions

```
In [38]: pca = PCA(n_components = 154)
         X_reduced = pca.fit_transform(X_train)
         X_recovered = pca.inverse_transform(X_reduced)
```

PCA for Compression

Below 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 for Compression

The equation of the inverse transformation is shown:

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

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

PCA - Incremental PCA

PCA is that it requires the whole training set to fit in memory in order for the SVD 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.

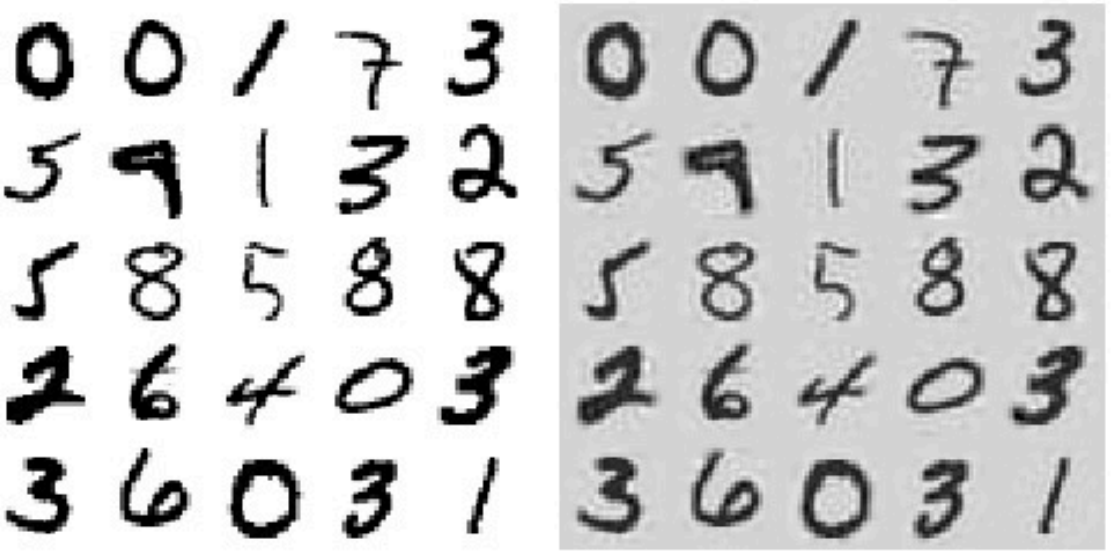
The following code splits the MNIST dataset into 100 mini-batches (using NumPy's array_split() function) and feeds them to Scikit-Learn's IncrementalPCA class.

```
In [43]: 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):
             print(".", end="") # not shown in the book
             inc_pca.partial_fit(X_batch)

         X_reduced = inc_pca.transform(X_train)
```

PCA - Incremental PCA

Below shows a few digits from the original training set (on the left), and the corresponding digits after compression and decompression with Inc-PCA

```
In [45]: plt.figure(figsize=(7, 4))
         plt.subplot(121)
         plot_digits(X_train[::2100])
         plt.subplot(122)
         plot_digits(X_recovered_inc_pca[::2100])
         plt.tight_layout()
```



PCA - Incremental PCA

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

In [53]:
```
# Next, another program would load the data and use it for training:
X_mm = np.memmap("./datasets/" + 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)
```

Out[53]:
```
IncrementalPCA(batch_size=525, copy=True, n_components=154, whiten=False)
```

PCA - Randomized PCA

Scikit-Learn offers yet another option to perform PCA, called Randomized PCA. This is a stochastic algorithm 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), so it is dramatically faster than theprevious algorithms when d is much smaller than n.

```
In [55]: rnd_pca = PCA(n_components=154, svd_solver="randomized", random_state=42)
         X_reduced = rnd_pca.fit_transform(X_train)
```

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.

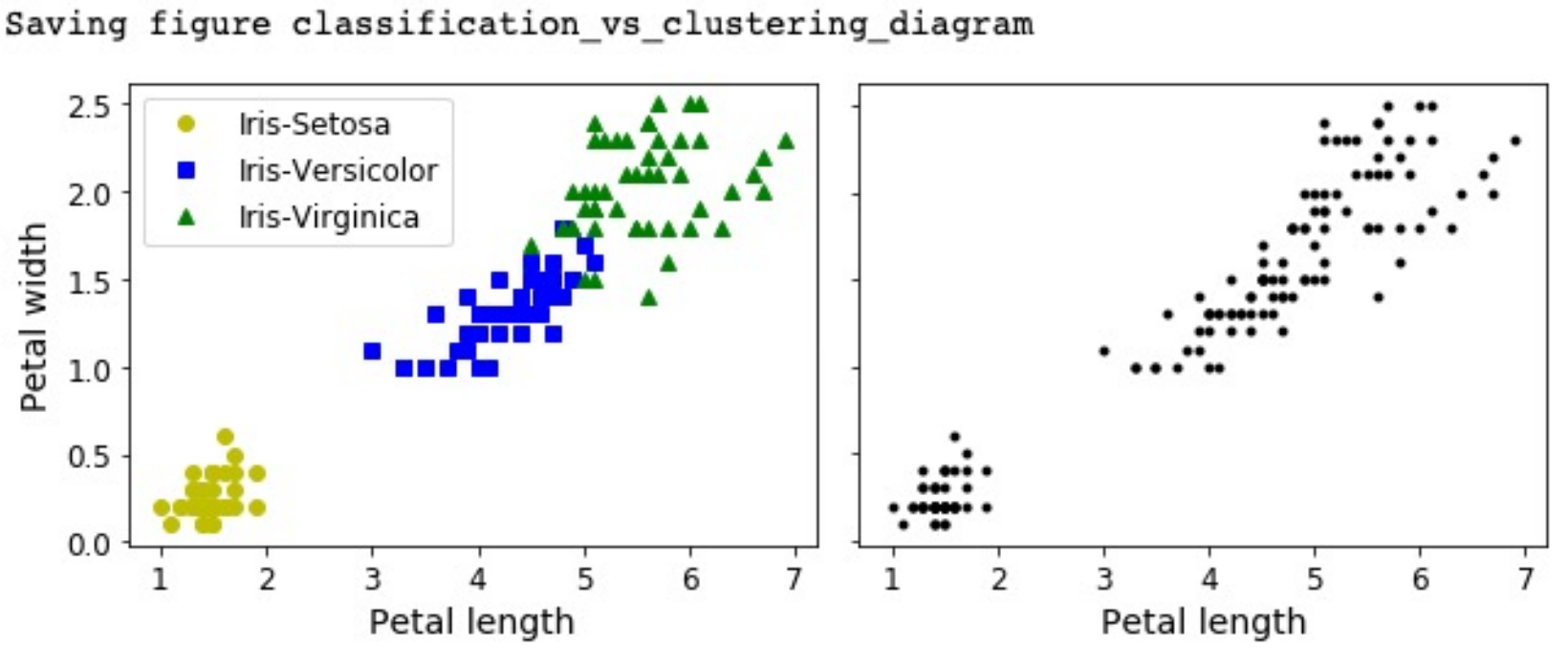
LLE

Locally Linear Embedding (LLE) is another very powerful nonlinear dimensionality reduction (NLDR) technique. It is a Manifold Learning technique that does not rely on projections like the previous algorithms. In a nutshell, LLE works by first measuring 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.

Clustering - KNN

归类：聚类(clustering)属于非监督学习(unsupervised learning), 无类别标记(class label)



Clustering - KNN

K-means 算法：Clustering 中的经典算法，算法接受参数k，然后将事先输入的n个数据划分为k个聚类以便使得所获得的聚类满足：同一聚类中的对象相似度较高；而不同聚类中的对象相似度较小。

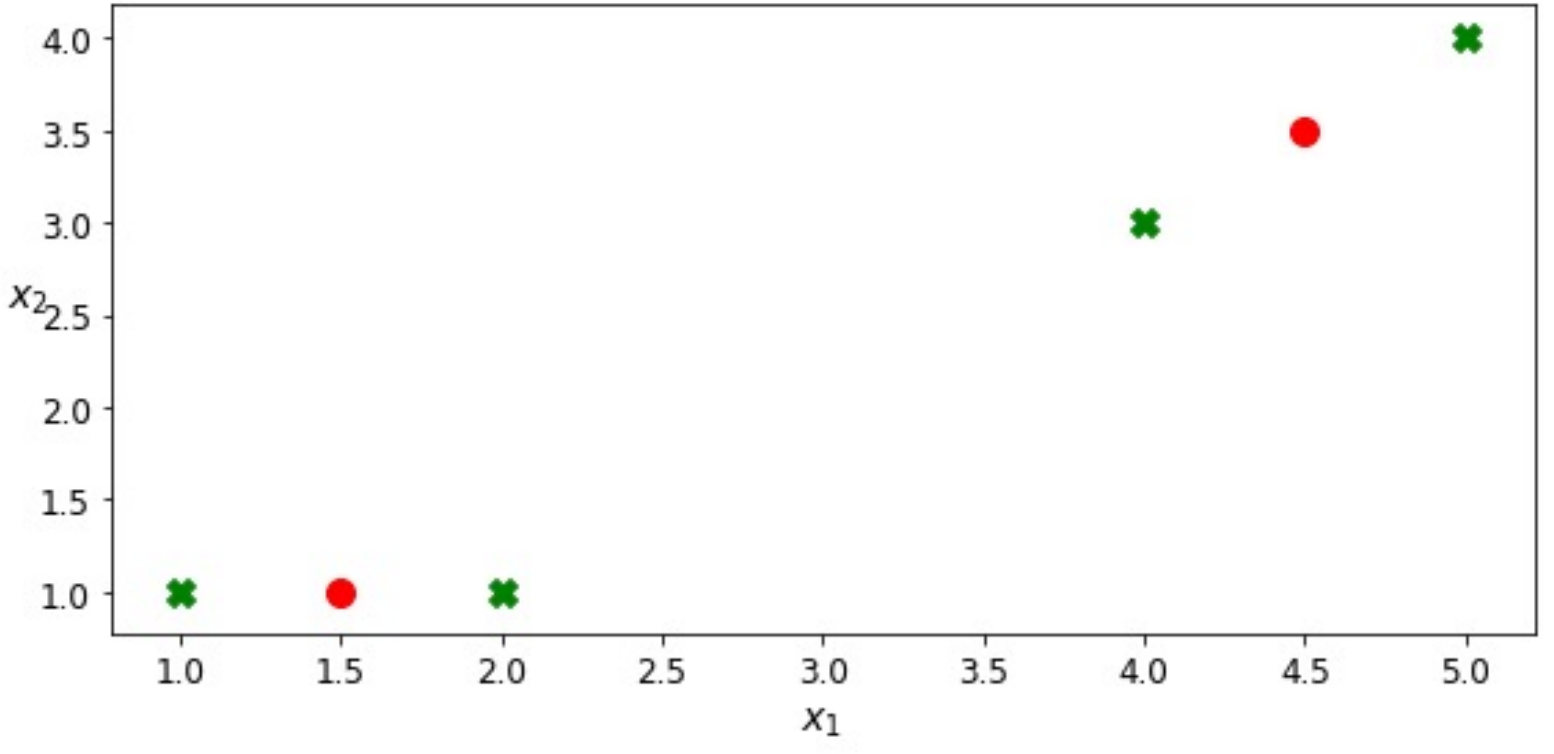
算法思想：以空间中K个点为中心进行聚类，对最靠近他们的对象归类，通过迭代的方法，逐次更新各聚类中心的值，直到得到更好的聚类结果。

算法描述：

1. 适当选择K个类的初始中心；
2. 在第K次迭代中，对任意一个样本，求其到K各中心的距离，将该样本归到距离最近的中心所在的类；
3. 利用均值等方法更新该类的中心值；
4. 对于所有K个聚类中心，如果利用s2, s3的迭代法更新后，值保持不变，则迭代结束，否则继续迭代。

Clustering - KNN

实例演示



Clustering - KNN

Below is how to train with sk-learn

```
In [113]: # Fit and Predict with sk-learn
          from sklearn.cluster import KMeans

          k = 2
          kmeans = KMeans(n_clusters=k, random_state=42)
          y_pred = kmeans.fit_predict(nodes)
          y_pred

Out[113]: array([1, 1, 0, 0], dtype=int32)

In [115]: kmeans.labels_

Out[115]: array([1, 1, 0, 0], dtype=int32)

In [116]: kmeans.cluster_centers_

Out[116]: array([[4.5, 3.5],
                 [1.5, 1. ]])
```

Clustering - KNN

优点： 速度快，简单
缺点： 最终结果与初始点选择相关，容易陷入局部最优，需知道K值。

SAIF

Shanghai Advanced
Institute of Finance
上海高级金融学院