Dimensionality Reduction

- The **Curse of Dimensionality** is the problem with ml involve thousands or even millions of features for each training instance, These features make training extremely slow and also make it much harder to find a good solution.
- IRL problems, it is often possible to reduce the number of features considerably, turning an intractable problem into a tractable one. This can be done by dropping unimportant pixels from an image or merging two correlated pixels that are often highly correlated into a single pixel, you won't lose much information.
- **Reducing Dimensionality** does cause some information loss, it will speed up your training, but it may make your system perform slightly worse. It also makes your pipeline a bit more complex, thus harder to maintain.
- In some cases, reducing the **dimensionality** of the training data may filter out some noise and unnecessary details and thus result in higher performance.
- **Dimensionality Reduction** is extremely useful for **data visualization**. Reducing the no. 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 gain some important insights by visually detecting patterns, such as **clusters**

The Curse of Dimensionality

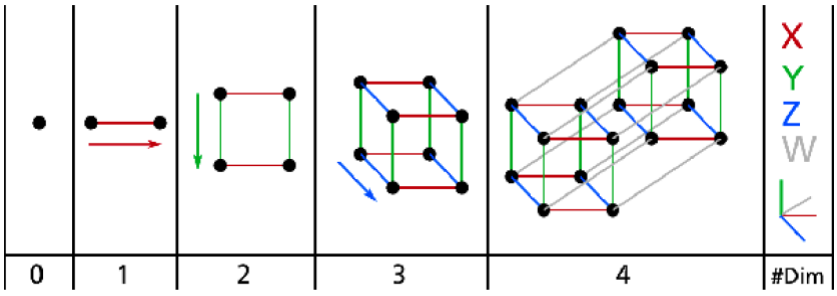


Figure 8-1. Point, segment, square, cube, and tesseract (0D to 4D hypercubes)[2]

- how can two points be so far apart when they both lie within the same unit hypercube?
  - 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. A new instance will likely be far away from any training instance, making predictions much less reliable than in lower dimensions.
  - The more dimensions the training set has, the greater the risk of overfitting it.

Main Approaches for Dimensionality Reduction

- The two approaches for Dimensionality Reduction are **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 a result, all training instances lie within (or close to) a much lower-dimensional **subspace** of the high-dimensional space.
- in Fig 8-2, all training instances lie close to a plane(this is a lower dimensional (2D) subspace of the higher-dimensional (3D) space)
- If we project every training instance perpendicularly onto this subspace, we get the new 2D dataset.
- We reduced the dataset's dimensionality from 3D to 2D.
- In Fig 8-3, the axes correspond to new features z_1 and z_2 : they are the coordinates of the projections on the plane.

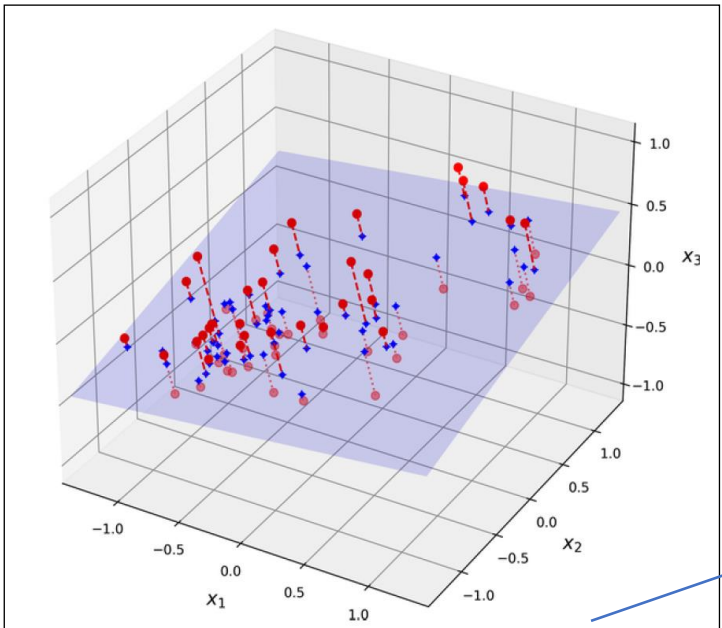


Figure 8-2. A 3D dataset lying close to a 2D subspace

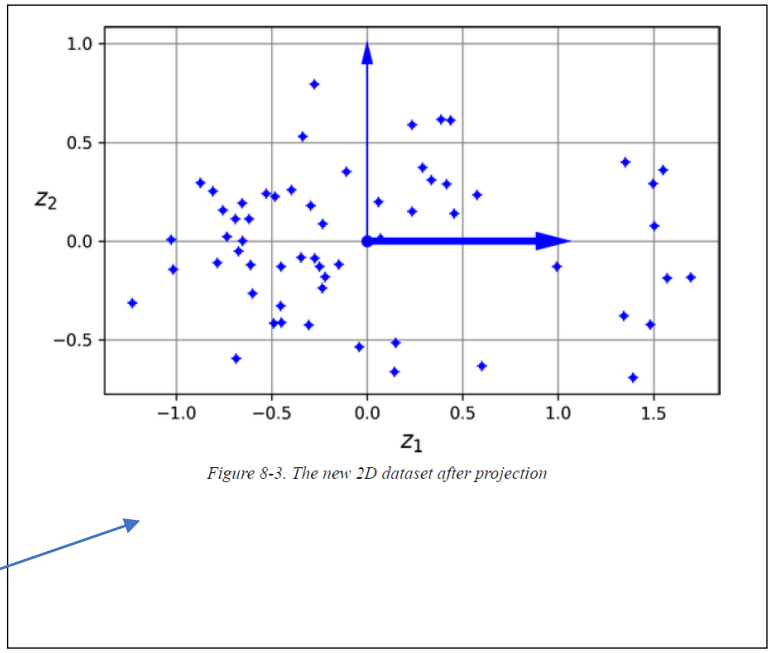


Figure 8-3. The new 2D dataset after projection

Manifold Learning

- 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.
- Simply projecting onto a plane (e.g., by dropping x) would squash different layers of the Swiss roll together.
- you probably want instead is to unroll the Swiss roll to obtain the 2D dataset

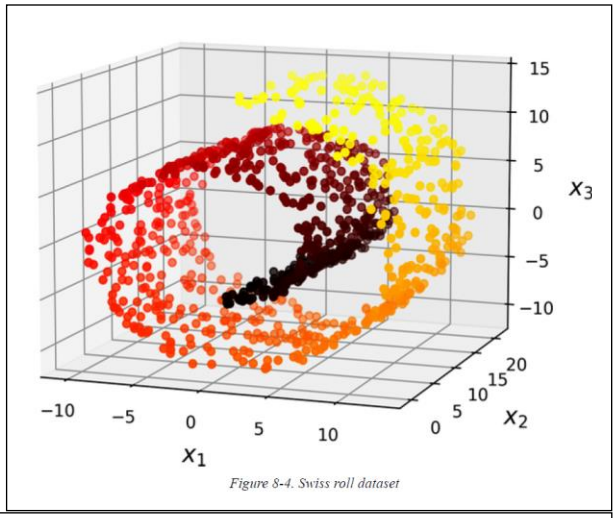


Figure 8-4. Swiss roll dataset

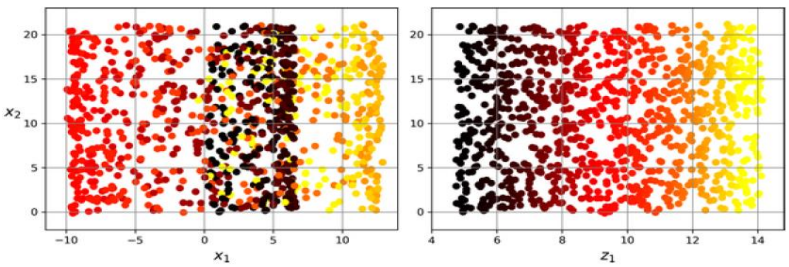


Figure 8-5. Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

- The **Swiss roll** is an example of a 2D **manifold.**
- 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.
- **manifold learning** relies on the **manifold assumption(**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.

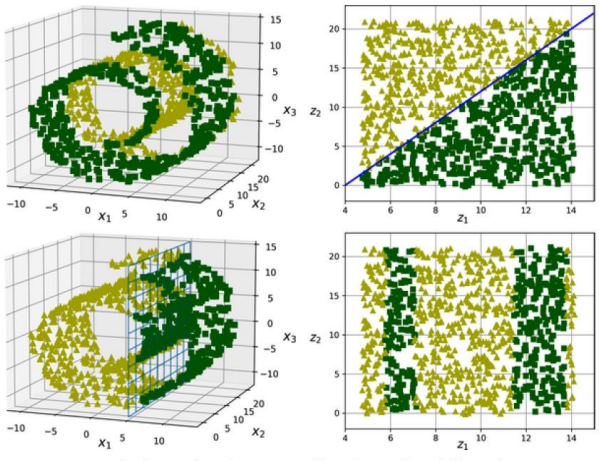


Figure 8-6. The decision boundary may not always be simpler with lower dimensions

- the DOFs available to you if you try to create a digit image are dramatically lower than the DOFs you have if you are allowed to generate any image you want. These constraints tend to squeeze the dataset into a lower-dimensional manifold.

- The manifold assumption is often accompanied by another implicit assumption, that the task at hand will be simpler if expressed in the lower-dimensional space of the manifold.
- However, this implicit assumption does not always hold.
- 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.

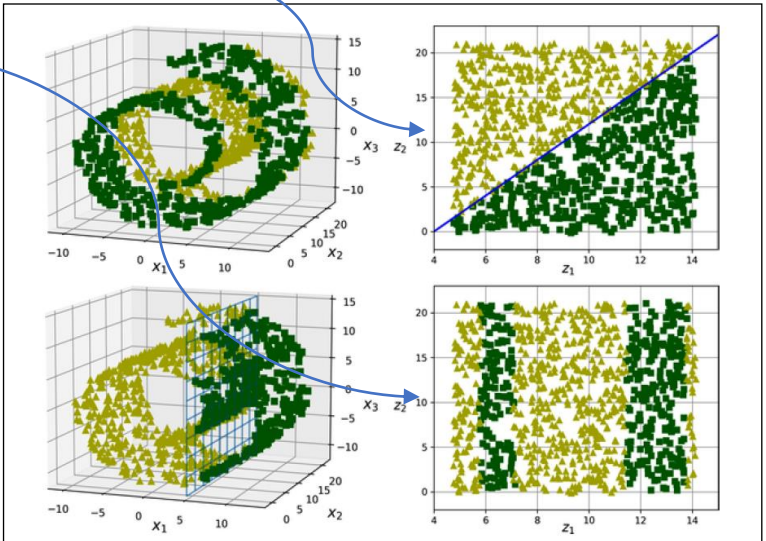


Figure 8-6. The decision boundary may not always be simpler with lower dimensions

PCA

- **Principal component analysis (PCA)** is by far the most popular dimensionality reduction algorithm.
- First it identifies the hyperplane that lies closest to the data, and then it projects the data onto it.

Preserving the Variance

- Before you can project the training set onto a lower-dimensional hyperplane, you first need to choose the right hyperplane.
- You need to select the axis that preserves the maximum amount of variance, as it will most likely lose less information than the other projections.
- it is the axis that minimizes the mean squared distance between the original dataset and its projection onto that axis.

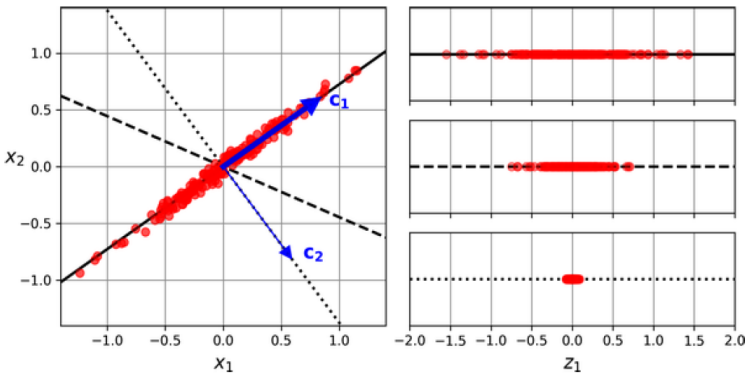


Figure 8-7. Selecting the subspace on which to project

Principal components

- The solid line in Fig 8-7, is the axis that PCA identifies which accounts for the largest amount of variance in the training set.
- It also finds a second axis, orthogonal to the first one, that accounts for the largest amount of the remaining variance.
- The i^{th} axis is called the i^{th} principal component (PC) of the data.
- 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.

- how can you find the principal components of a training set?
    - 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 the unit vectors that define all the principal components.
    - Equation 8-1. Principal components matrix

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

    - Using NumPy's **svd()** function to obtain all the principal components of the 3D training set represented in Figure 8-2, then it extracts the two-unit vectors that define the first two PCs

```
import numpy as np

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

- PCA assumes that the dataset is centered around the origin. As you will see, Scikit- Learn's PCA classes take care of centering the data for you.

Projecting Down to d Dimensions

- 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.
- 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 **X** by the matrix $\mathbf{W}_d$(the matrix containing the first **d** columns of **V)**

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

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

- Python code to project the training set onto the plane defined by the first two principal components.

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

Using Scikit-Learn

- Scikit-Learn's **PCA** class uses **SVD** to implement PCA

```
from sklearn.decomposition import PCA

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

- **components_** attribute holds the transpose of $\mathbf{W}_d$
- **pca.components_.T[: , 0]**, Defines the first principal component

Explained Variance Ratio

- **explained_variance_ratio_** attribute. contains the proportion of the dataset's variance that lies along each principal component.

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

Choosing the Right Number of Dimensions

- you want to choose the number of dimensions that add up to large portion of the variance – say 95%
- if you are reducing dimensionality for data visualization, in this case you will want to reduce the dimensionality down to 2 or 3
- preserving 95% of the training set's variance on MNIST's dataset

```
from sklearn.datasets import fetch_openml

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

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

n_components= d and run PCA again

- 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)
```

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

- you can plot the explained variance as a function of the number of dimensions (plot **cumsum)**

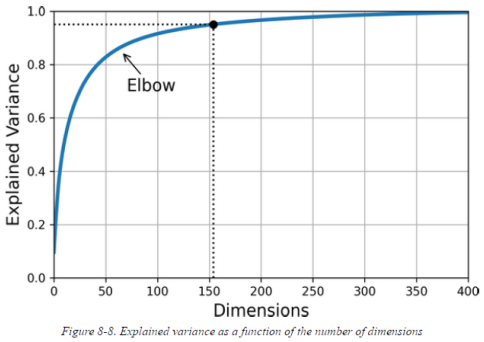


Figure 8-8. Explained variance as a function of the number of dimensions

PCA for Compression

- after applying PCA to the MNIST dataset while preserving 95% of its variance, we are left with 154 features, instead of the original 784 features(20% of its original size), and we only lost 5 % variance.
- It is also possible to decompress the reduced dataset back to 784 dimensions by applying the **inverse transformation** of the PCA projection
- This won't give you back the original data, since the projection lost a bit of information (within the 5% variance that was dropped), but it will likely be close to the original data.
- **The reconstruction error** is the mean squared distance between the original data and the reconstructed data (compressed and then decompressed)
- The **inverse_transform()** method decompress the reduced MNIST dataset back to 784 dimensions:

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

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

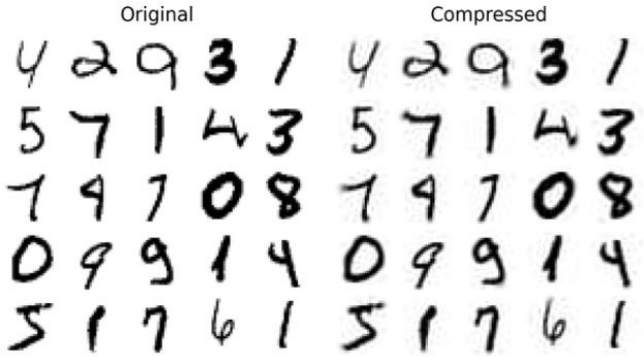


Figure 8-9. MNIST compression that preserves 95% of the variance

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

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

Randomized PCA

- If you set the **svd_solver** hyperparameter to "**randomized**", Scikit-Learn uses a stochastic algorithm called **randomized PCA** that quickly finds an approximation of the first d principal components.
- Its computational complexity is **O(m × 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", random_state=42)
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 **max(m, n) > 500** and **n_components** is an integer smaller than **80%** of **min(m, n)**, or else it uses the full SVD approach.
- you can set the **svd_solver** hyperparameter to "**full**", If you want to use full SVD for a slightly more precise result.

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.
- **incremental PCA (IPCA)** algorithms have been developed that allow you to split the training set into mini-batches and feed these in one mini-batch at a time.
- This is useful for large training sets and for applying PCA online (i.e., on the fly, as new instances arrive).
- You can split MNIST training set into 100 mini-batches (using NumPy's **array_split()** function) and feeds them to Scikit-Learn's **IncrementalPCA** class to reduce the dimensionality of the MNIST dataset down to 154 dimensions, just like before.
- 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)
```

- you can also 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.
  - We first create a memory-mapped (memmap) file and copy the MNIST training set to it, then call flush() to ensure that any data still in the cache gets saved to disk.
  - In real life, the training set would not fit in memory, so you would load it chunk by chunk and save each chunk to the right part of the memmap array

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

  - Next, we load the **memmap** file and use it like a regular NumPy array and use the **IncrementalPCA** class to reduce its dimensionality.
  - Since this algorithm uses only a small part of the array at any given time, memory usage remains under control.
  - This makes it possible to call the usual **fit()** method instead of **partial_fit()**

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

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

Random Projection

- the random projection algorithm projects the data to a lower-dimensional space using a random linear projection.
- it turns out that such a random projection is actually very likely to preserve distances fairly well
- The **sklearn.random_projection** module implements a simple and computationally efficient way to reduce the dimensionality of the data.
- This module implements two types of unstructured random matrix: **Gaussian random matrix** and **sparse random matrix.**

The Johnson-Lindenstrauss lemma

- Demonstrate that two similar instances will remain similar after the projection, and two very different instances will remain very different.
- They came up with an equation that determines the minimum number of dimensions to preserve in order to ensure—with high probability—that distances won't change by more than a given tolerance ε
- the equation does not use n, it only relies on **m**(no. of training instances) and **ε**. This equation is implemented by the **johnson_lindenstrauss_min_dim()** function

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

Gaussian random projection

- The **GaussianRandomProjection** reduces the dimensionality by projecting the original input space on a randomly generated matrix where components are drawn from the following distribution

$N(0, \frac{1}{n_{components}})$.

```
>>> import numpy as np
>>> from sklearn import random_projection
>>> X = np.random.rand(100, 10000)
>>> transformer = random_projection.GaussianRandomProjection()
>>> X_new = transformer.fit_transform(X)
>>> X_new.shape
(100, 3947)
```

Sparse random projection

- The **SparseRandomProjection** reduces the dimensionality by projecting the original input space using a sparse random matrix.
- **Sparse random matrices** are an alternative to **dense Gaussian random projection matrix** that guarantees similar embedding quality while being much more memory efficient and allowing faster computation of the projected data.

If we define `s = 1 / density`, the elements of the random matrix are drawn from

$$\begin{cases} -\sqrt{\frac{s}{n_{\text{components}}}} & 1/2s \\ 0 & \text{with probability} \quad 1 - 1/s \\ +\sqrt{\frac{s}{n_{\text{components}}}} & 1/2s \end{cases}$$

```
>>> import numpy as np
>>> from sklearn import random_projection
>>> X = np.random.rand(100, 10000)
>>> transformer = random_projection.SparseRandomProjection()
>>> X_new = transformer.fit_transform(X)
>>> X_new.shape
(100, 3947)
```

Inverse Transform

- The random projection transformers have **compute_inverse_components** parameter. When set to **True**, after creating the random **components_** matrix during fitting, the transformer computes the pseudo-inverse of this matrix and stores it as **inverse_components_.**
- The **inverse_components_** matrix has shape $n_{features} \times n_{components}$, and it is always a dense matrix, regardless of whether the components matrix is sparse or dense. So, depending on the number of features and components, it may use a lot of memory.

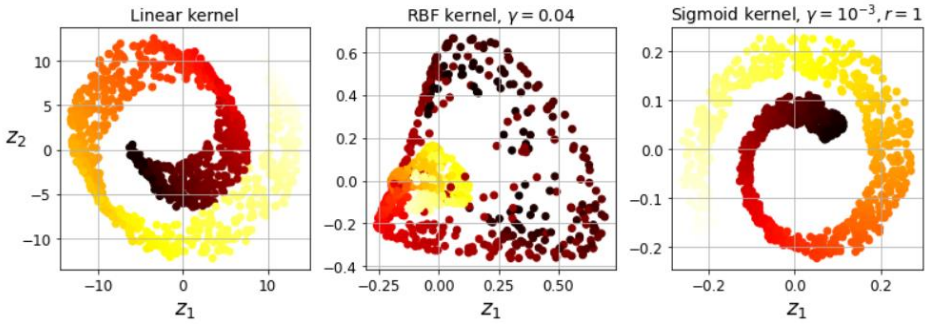
```
>>> import numpy as np
>>> from sklearn.random_projection import SparseRandomProjection
>>> X = np.random.rand(100, 10000)
>>> transformer = SparseRandomProjection(
...   compute_inverse_components=True
... )
...
>>> X_new = transformer.fit_transform(X)
>>> X_new.shape
(100, 3947)
>>> X_new_inversed = transformer.inverse_transform(X_new)
>>> X_new_inversed.shape
(100, 10000)
```

Kernel PCA

- **Kernel Trick,** a mathematical technique that implicitly maps instances into a very high-dimensional space(**feature space**), enabling non-linear classification and regression with **SVM**
- A linear decision boundary in high-dimensional feature space corresponds to a complex nonlinear decision boundary in the original space
- we can apply the same trick to **PCA,** making it possible to perform complex nonlinear projections for dimensionality reduction, this is called **Kernel PCA(KPCA)**
- it is good at preserving clusters of instances after projections, or sometimes unrolling datasets that lie close to a twisted manifold.
- Using scikit-Learn's **KernelPCA** class to perform **KPCA** with an **RBF Kernel**

```
from sklearn.decomposition import KernelPCA

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



Selecting a Kernel and Tuning Hyperparameters

- using dimensionality reduction as a preprocessing step for a supervised learning task

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

clf = make_pipeline(PCA(random_state=42),
                    RandomForestClassifier(random_state=42))
param_distrib = {
    "pca__n_components": np.arange(10, 80),
    "randomforestclassifier__n_estimators": np.arange(50, 500)


}
rnd_search = RandomizedSearchCV(clf, param_distrib, n_iter=10, cv=3,
                                random_state=42)
rnd_search.fit(X_train[:1000], y_train[:1000])
```

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

- a **KPCA** is an unsupervised learning algorithm, there is no obvious performance to help you select the best kernel and hyperparameters values.

```
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(solver="lbfgs"))
    ])

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)
```

```
print(grid_search.best_params_)
```

```
{'kpca__gamma': 0.043333333333333335, 'kpca__kernel': 'rbf'}
```

- another approach, which is entirely unsupervised, is to select the kernel and hyperparameters that yield the lowest reconstruction error, the reconstruction is hard to get with nonlinear projections(manifold)

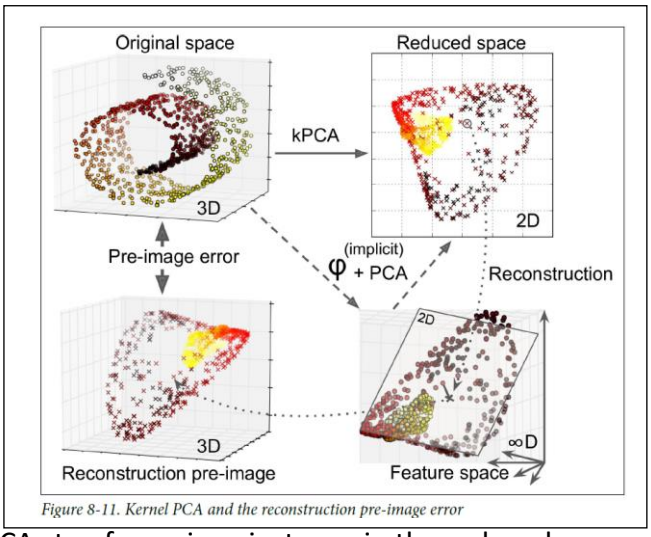


Figure 8-11. Kernel PCA and the reconstruction pre-image error

- If we invert the linear PCA step for a given instance in the reduced space, the reconstructed point will lie in feature space, not in the original.
- 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 possible 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 kernel and hyperparameters that minimize this reconstruction pre-image error.
- How to perform this reconstruction pre-image?
  - One solution is to train a supervised regression model, with projected instances as the training set and the original instances as targets.
  - Scikit-learn will do this automatically if you set **fit_inverse_transform = True**

```
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.**
- How to compute the Reconstruction pre-image error?

```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(X, X_preimage)
32.786308795766132
```

LLE

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

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

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

- The variable **t** is a 1D NumPy array containing the position of each instance along the rolled axis of the Swiss roll. We don't use it in this example, but it can be used as a target for a nonlinear regression task.
- in fig 8-10, the Swiss roll is completely unrolled, and the distances between instances are locally well preserved.
- However, distances are not preserved on a larger scale, the unrolled Swiss roll should be a rectangle, not this kind of stretched and twisted band.
- Nevertheless, LLE did a pretty good job of modeling the manifold.
- How does Locally Linear Embedding work?
  1. Use a KNN approach to find the k nearest neighbors of every data point.(**n-neighbors** in the model)
  2. Construct a weight matrix where every point has its weights determined by minimizing the error of the cost function
  3. Note that every point is a linear combination of its neighbors, which means that weights for non-neighbors are 0.

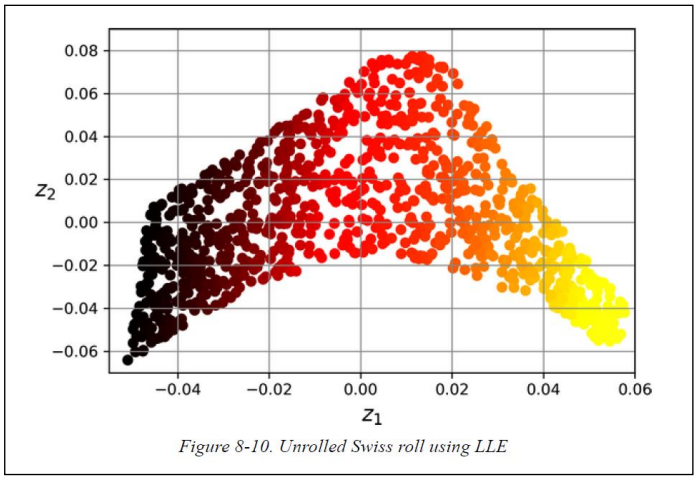
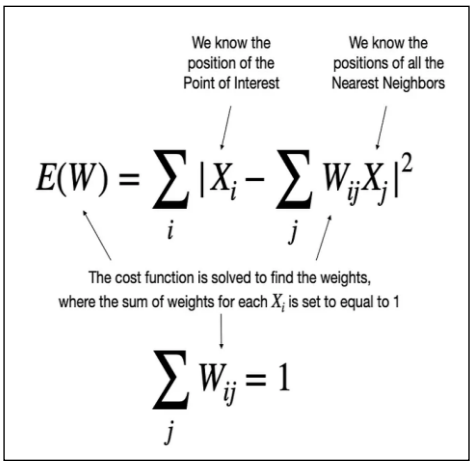
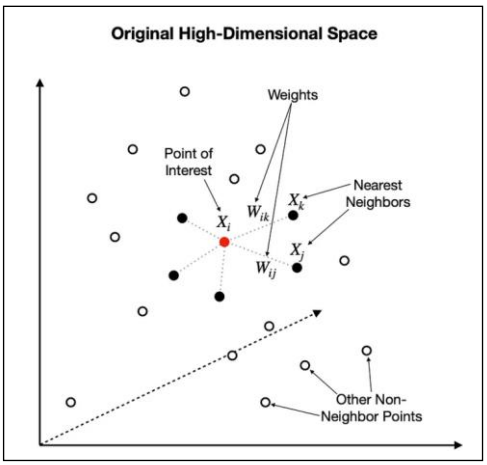


Figure 8-10. Unrolled Swiss roll using LLE



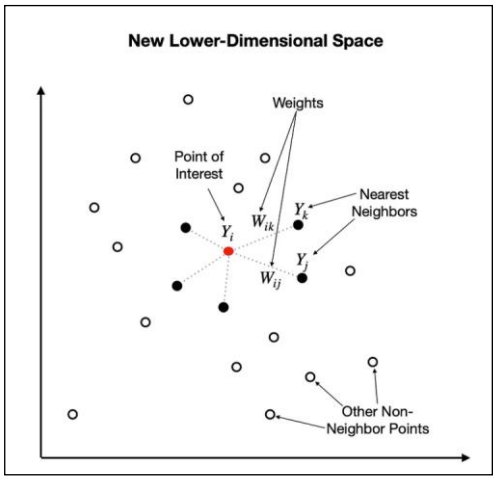


4. Find the positions of all the points in the new lower-dimensional embedding by minimizing the cost function, we use weights (W) from step two and solve for Y.

We know the weights from the previous step

$$C(Y) = \sum_i | Y_i - \sum_j W_{ij} Y_j |^2$$

The cost function is solved to find the positions of Y_i and its neighbors in the new lower-dimensional space using weights from the previous step.



- Scikit-Learn's **LLE** implementation has the following computational complexity: **O(m log(m)n log(k))** for finding the **k-nearest neighbors**, **O(mnk^3)** for optimizing the weights, and **O(dm^2)** for constructing the lowdimensional representations. Unfortunately, the **m^2** in the last term makes this algorithm scale poorly to very large datasets.
- **Modified LLE (MLLE)** — One well-known issue with LLE is the regularization problem. A way to address it is to use multiple weight vectors in each neighborhood. This is the essence of MLLE.

Other Dimensionality Reduction Techniques

- **Multidimensional scaling (MDS)** reduces dimensionality while trying to preserve the distances between the instances. **Random projection** does that for high-dimensional data, but it doesn't work well on lowdimensional data. **sklearn.manifold.MDS**
- **Isomap** creates a graph by connecting each instance to its nearest neighbors, then reduces dimensionality while trying to preserve the geodesic distances between the instances. The geodesic distance between two nodes in a graph is the number of nodes on the shortest path between these nodes. **sklearn.manifold.Isomap**
- **t-distributed stochastic neighbor embedding (t-SNE)** reduces dimensionality while trying to keep similar instances close and dissimilar instances apart. It is mostly used for visualization, in particular to visualize clusters of instances in high-dimensional space. **sklearn.manifold.TSNE**
- **Linear discriminant analysis (LDA)** is a linear classification algorithm that, during training, learns the most discriminative axes between the classes. These axes can then be used to define a hyperplane onto which to project the data. The benefit of this approach is that the projection will keep classes as far apart as possible, so LDA is a good technique to reduce dimensionality before running another classification algorithm (unless LDA alone is sufficient). **sklearn.discriminant_analysis.LinearDiscriminantAnalysis**

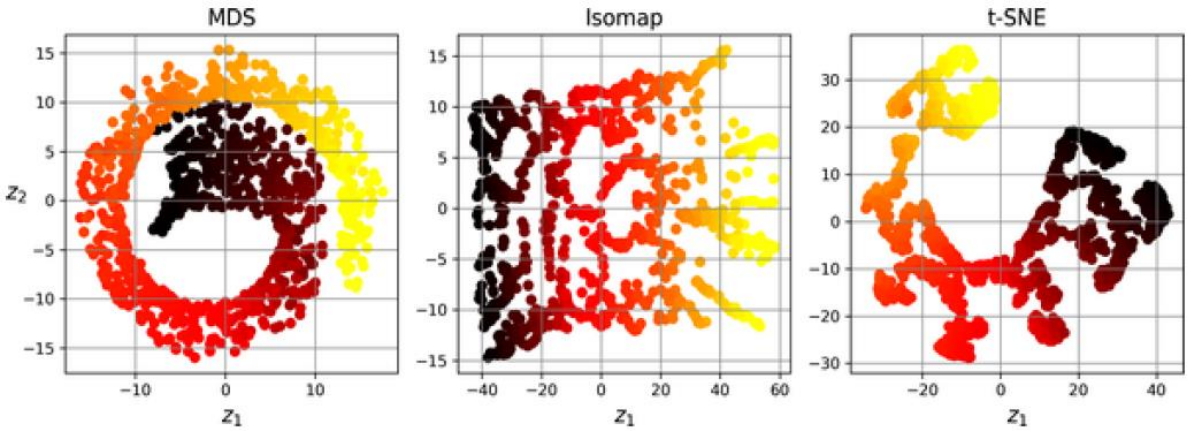


Figure 8-11. Using various techniques to reduce the Swiss roll to 2D

Questions

- **What are the main motivations for reducing a dataset's dimensionality? What are the main drawbacks?**
  - the main motivations for dimensionality reduction are: To speed up a training algorithm, to visualize data and gain insights and to save space (compression)
  - the main drawbacks are some information is lost possibly degrading the performance of the training algorithm, can be computationally expensive, adds some complexity to your machine learning pipelines and transformed features are hard to interpret
- **What is the curse of dimensionality?**
  - it is the fact that many problems that don't arise in low dimensional space arise in high dimensional space. In machine learning, one common manifestation is the fact that randomly sampled high-dimensional vectors are generally very sparse, increasing the risk of overfitting and making it very difficult to identify patterns in the data without having plenty of training data
- **Can PCA be used to reduce the dimensionality of a highly nonlinear dataset?**
  - **PCA** can be used to reduce the dimensionality of most datasets, even if they are highly nonlinear, because it can get rid of useless dimensions
  - However, if there is no useless dimensions__ as in Swiss roll dataset__ then reducing dimensionality with PCA will lose too much information. You want to unroll the Swiss roll , not to squash it.
- **Can PCA be used to reduce the dimensionality of a highly nonlinear dataset? 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?**
  - Depends on the dataset, if the dataset is composed of points that are almost perfectly aligned, in this case PCA can reduce the dataset to just one dimension while still preserving the 95% of the variance
  - If the dataset is composed of perfectly random points, scattered all around the 1000 dimensions, in this case roughly 950 dimensions are required to preserve the 95% variance.
  - So, the answer depends on the dataset, you can plot the explained variance as a function
- **How can you evaluate the performance of a dimensionality reduction algorithm on your dataset?**
  - A dimensionality reduction algorithm performs well if it eliminates a lot of dimensions from the dataset without losing too much information
  - One way to measure this is to apply the reverse transformation and measure the reconstruction error
  - If the dimensionality reductions doesn't have reverse transformation, and you are using the dimensionality reduction as a preprocessing step, you can measure the performance of the second algorithm; if the dimensionality reduction didn't lose too much information, then the algorithm should perform just as well as when using the original dataset
- **Does it make any sense to chain two different dimensionality reduction algorithms?**
  - Yes, it makes sense. One example is using **PCA** to quickly get rid of a large no. of useless dimensions, then applying another much slower dimensionality reduction algorithm, such as **LLE**. This two-step approach will likely yield the same performance as using the LLE only, but in a fraction of time.