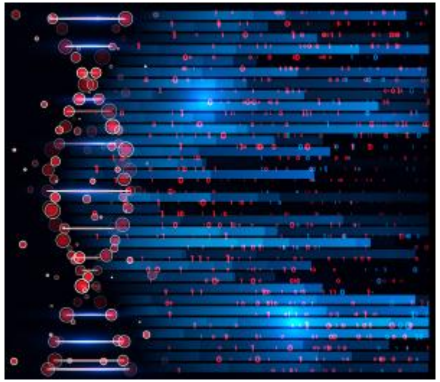
Hands-on Machine Learning

Dimensionality Reduction

The Curse of Dimensionality

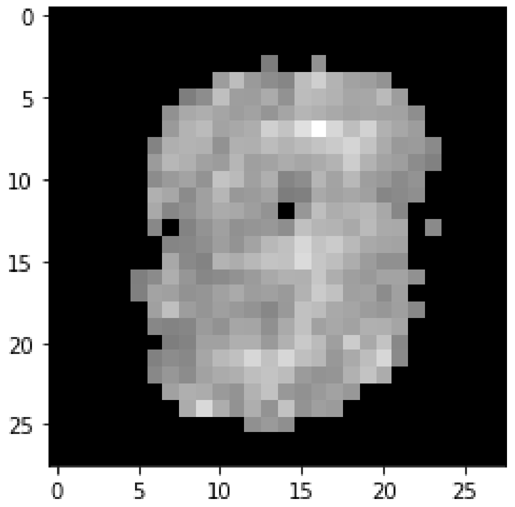
- Many machine learning problems involve thousands or even millions of features for each training instance.

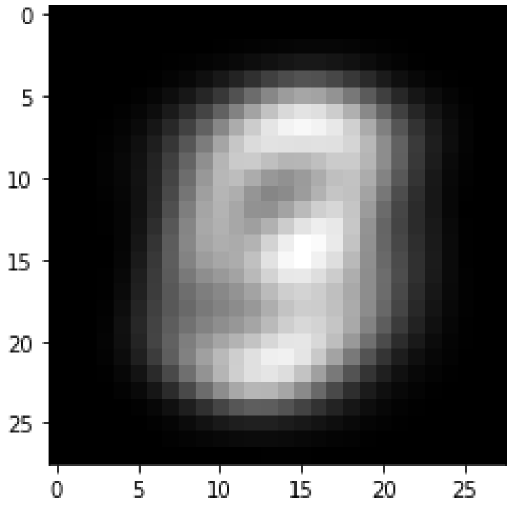


- Not only do all these features make training extremely slow, but they can also make it much harder to find a good solution.
  - This problem is often referred to as the *curse of dimensionality*.

Dimensionality Reduction

➢ It is possible to reduce the number of features considerably, turning an intractable problem into a tractable one.





Pros and Cons of Dimensionality Reduction

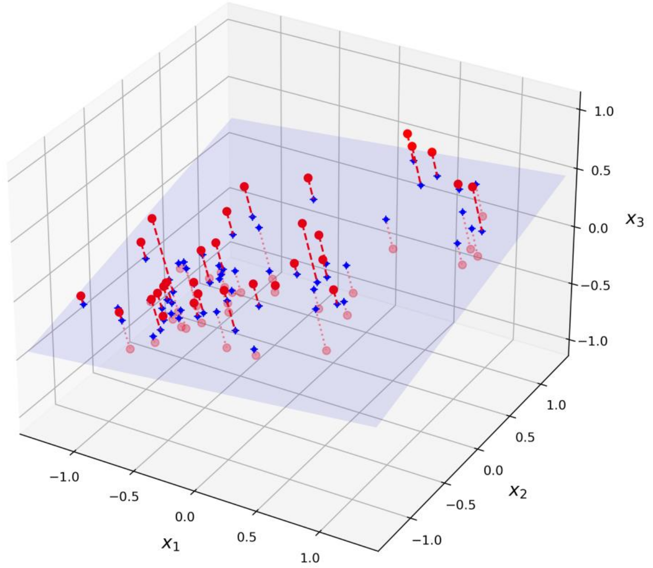
- Advantages:
  - Speed up training
  - May filter out some noise and unnecessary details and result in higher performance in some cases
  - Extremely useful for data visualization
  - Reduced overfitting
- Disadvantages:
  - Information loss and slightly worse performance
  - More complex pipelines
  - Interpretability challenges

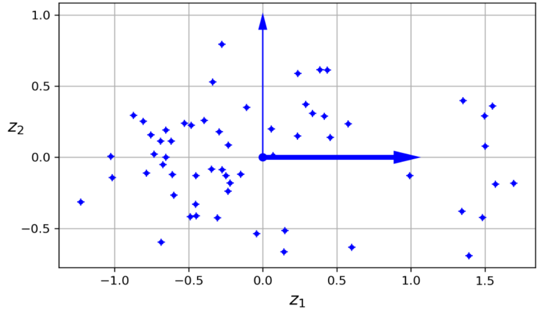
1.
Main Approaches for Dimensionality Reduction

Projection

- In most real-world problems, training instances are *not* spread out uniformly across all dimensions.
  - Many features are almost constant.
  - Others are highly correlated.

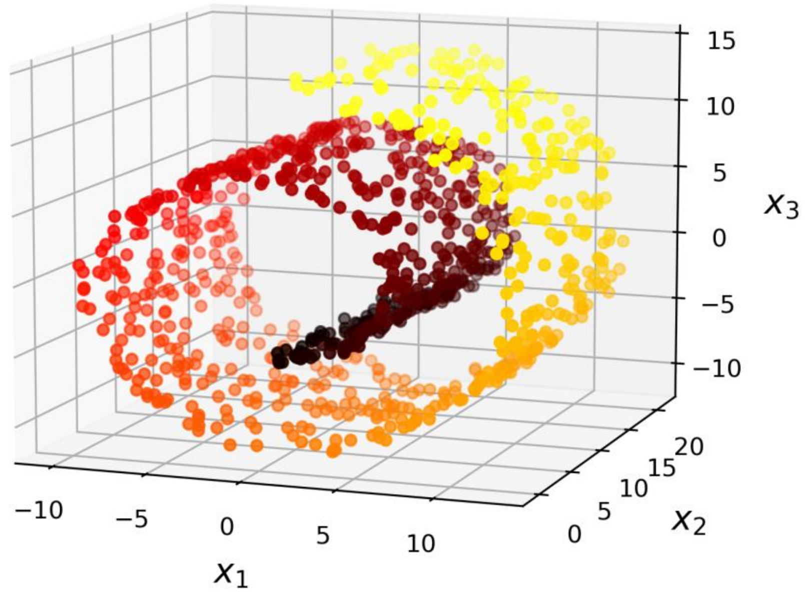
- All training instances lie within (or close to) a much lower-dimensional *subspace* of the high-dimensional space.



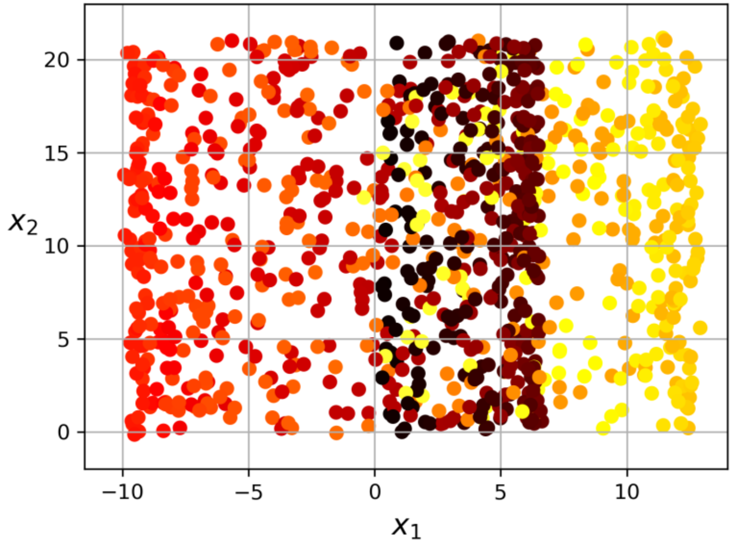


Manifold Learning

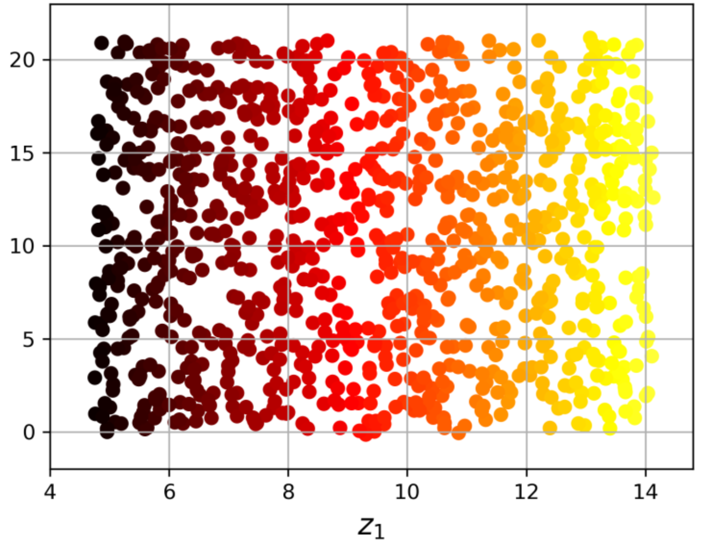
- In many cases the subspace may twist and turn.
- Simply projecting onto a plane (e.g., by dropping x_3) would squash different layers of the Swiss roll together.
- We want to unroll the Swiss roll to obtain the 2D dataset.



Manifold Learning



projecting onto a plane



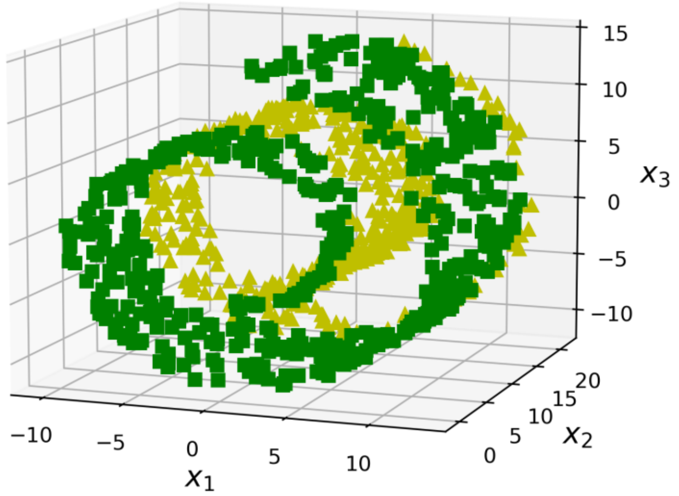
unrolling the Swiss roll

Manifold Learning

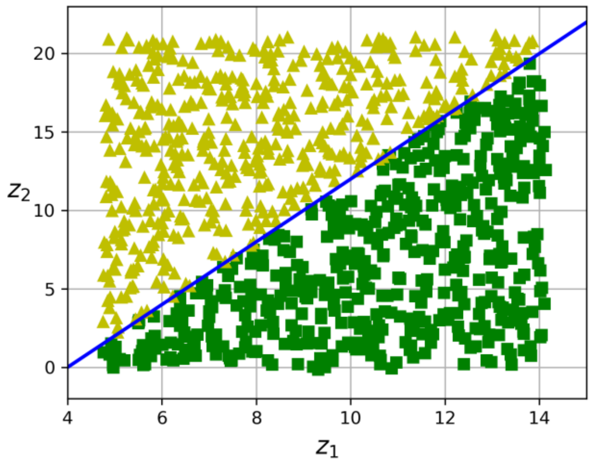
- In simple terms, a 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space.
  - The Swiss roll is an example of a 2D *manifold*.
- A d-dimensional manifold is a part of an n-dimensional space ($d < n$) that locally resembles a d-dimensional hyperplane.
- Many dimensionality reduction algorithms work by modeling the manifold on which the training instances lie;
  - This is called *manifold learning*.
- *Manifold assumption*: most real-world high-dimensional datasets lie close to a much lower-dimensional manifold.

Example

- Consider the MNIST dataset: all handwritten digit images have some similarities.
  - They are made of connected lines, the borders are white, and they are more or less centered.
- If you randomly generated images, only a tiny fraction of them would look like handwritten digits.
  - The degrees of freedom available to you if you try to create a digit image are dramatically lower than the degrees of freedom you have if you are allowed to generate any image you want.
- These constraints tend to squeeze the dataset into a lower-dimensional manifold.



x_1
x_2
x_3
−10
−5
0
5
10
15
20



z_1
z_2
4
6
8
10
12
14
0
5
10
15
20

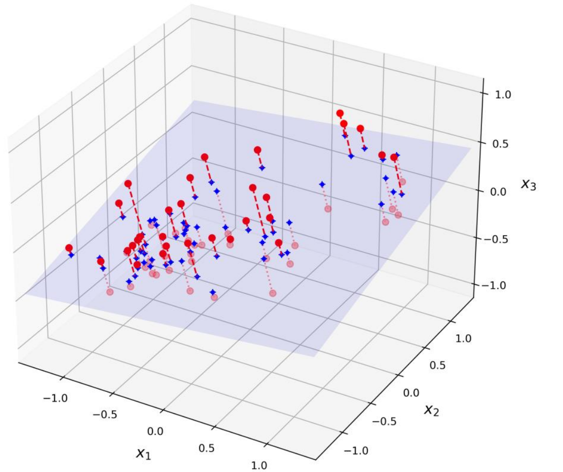
2.
Principal Component Analysis

Principal Component Analysis

- *Principal component analysis* (PCA) is the most popular dimensionality reduction algorithm.

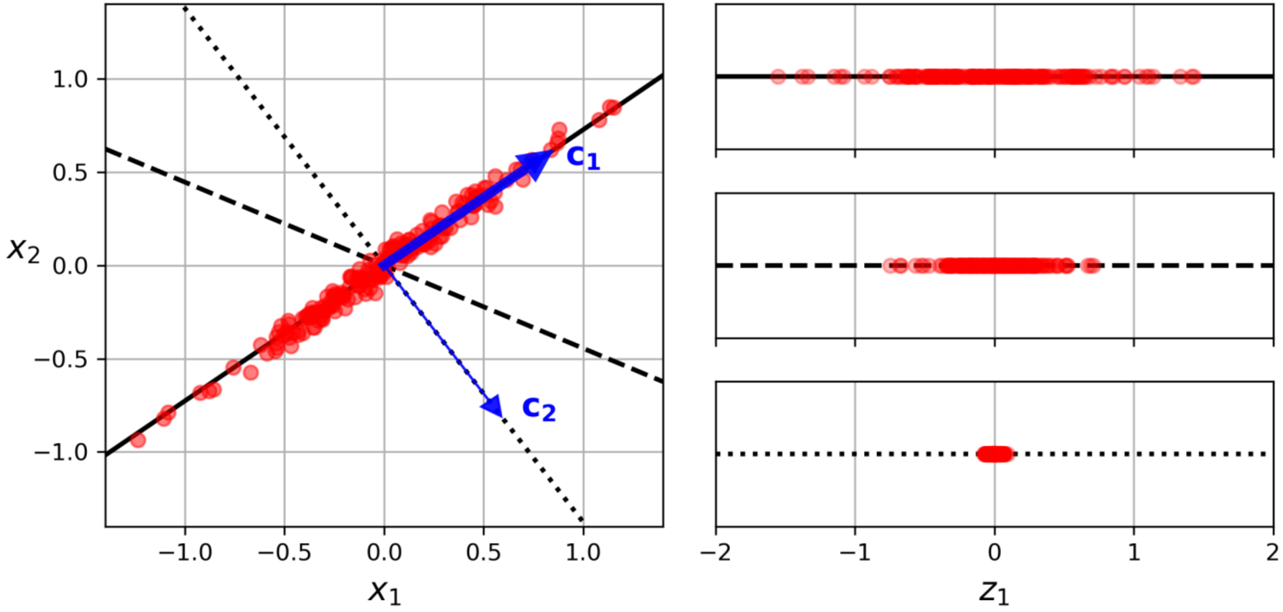
- First it identifies the hyperplane that lies closest to the data, and then it projects the data onto it.

- How to choose the right hyperplane?
  - Select the axis that preserves the maximum amount of variance



Preserving the Variance

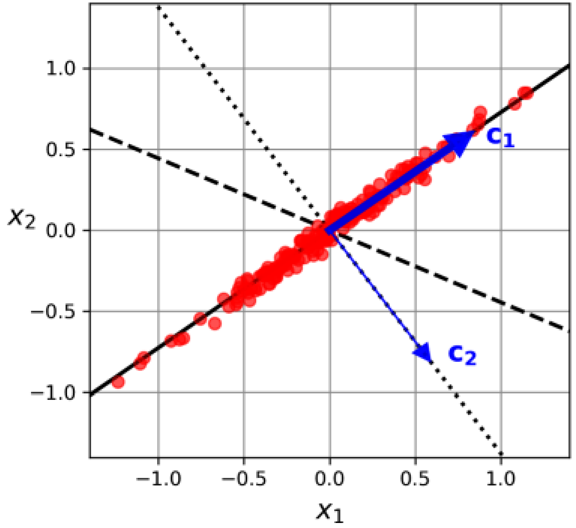
- Select the axis that preserves the maximum amount of variance, as it will lose less information than the other projections



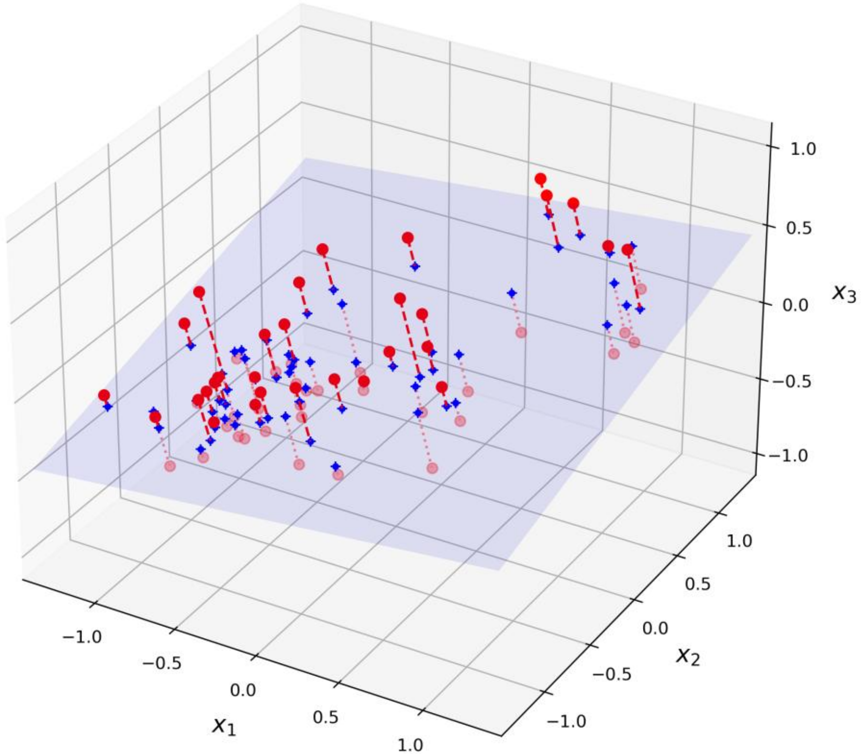
Principal Components

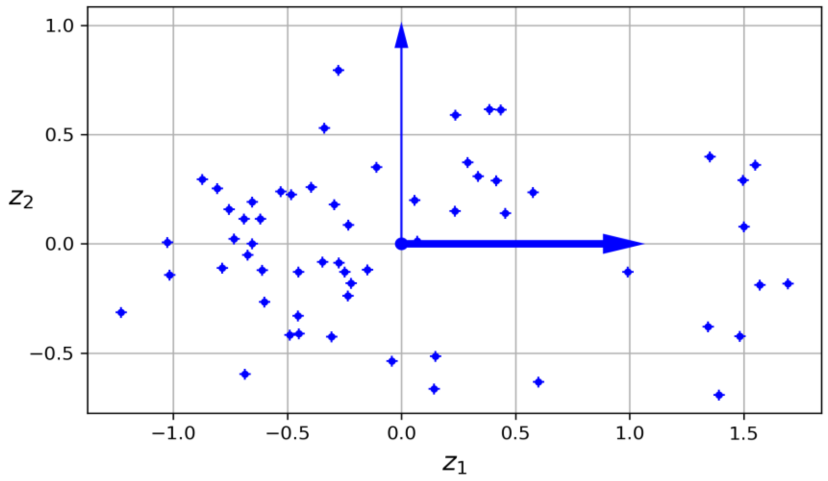
- PCA identifies the axis that accounts for the largest amount of variance in the training set (c_1), then a second axis, orthogonal to the first one (c_2), that accounts for the largest amount of the remaining variance.

- The i-th axis is called the *i-th principal component* (PC) of the data.



Principal Components





Finding Principal Components

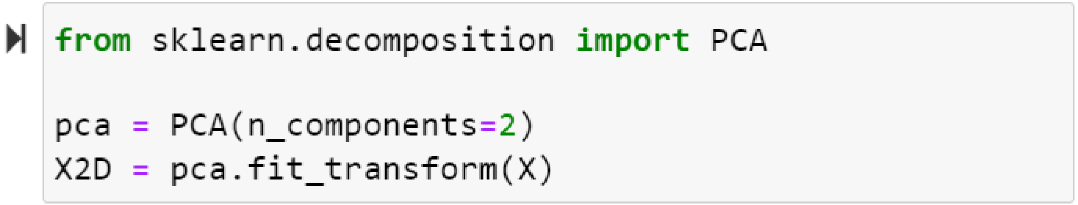
- We can use *singular value decomposition* (SVD) that decompose the training set matrix $\boldsymbol{X}$ into the matrix multiplication of three matrices $\boldsymbol{U}\,\boldsymbol{\Sigma}\,\boldsymbol{V}^T$, where $\boldsymbol{V}$ contains the unit vectors that define all the principal components:

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

- 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: $\boldsymbol{X}_{d-\text{proj}} = \boldsymbol{X}\,\boldsymbol{W}_{\boldsymbol{d}}$

PCA in Scikit-Learn

- Scikit-Learn's `PCA` class uses SVD to implement PCA.
- The following code applies PCA to reduce the dimensionality of the dataset down to two dimensions:

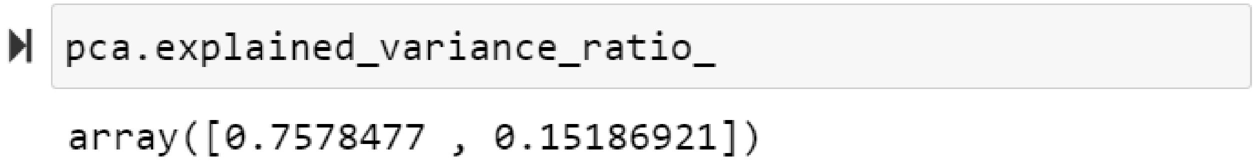


```
from sklearn.decomposition import PCA

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

- After fitting the PCA transformer, its `components_` attribute holds the transpose of $\boldsymbol{W}_d$: it contains one row for each of the first d principal components.

Explained Variance Ratio

- The *explained variance ratio* of each principal component, available via the `explained_variance_ratio_` variable, indicates the proportion of the dataset's variance that lies along each principal component.

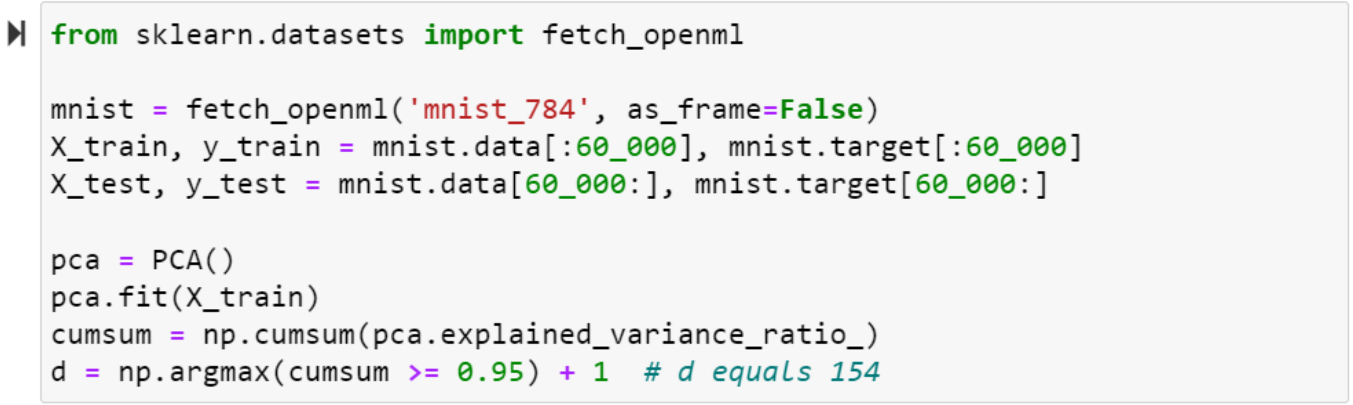


```
pca.explained_variance_ratio_
```

```
array([0.7578477 , 0.15186921])
```

Choosing the Right Number of Dimensions

- Choose the number of dimensions that add up to a sufficiently large portion of the variance—say, 95%.

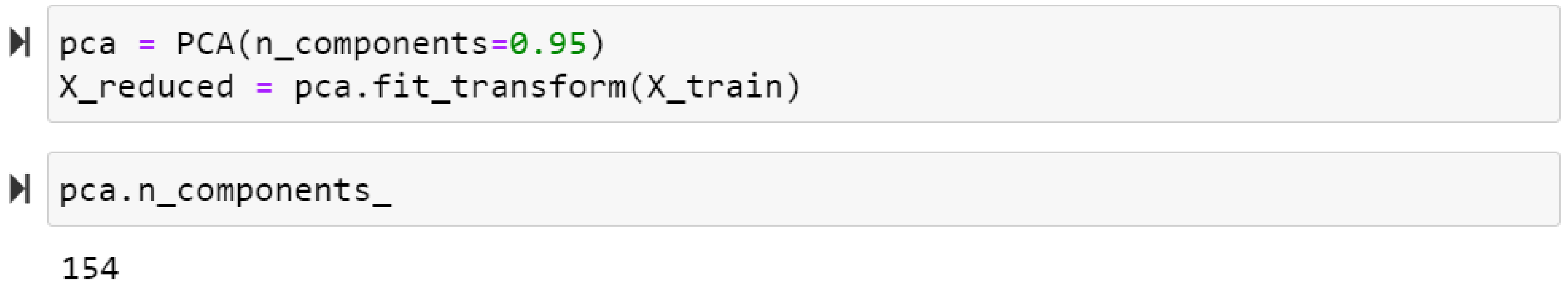


```
from sklearn.datasets import fetch_openml

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
```

- You could then set `n_components=d` and run PCA again.

Choosing the Right Number of Dimensions

- 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:



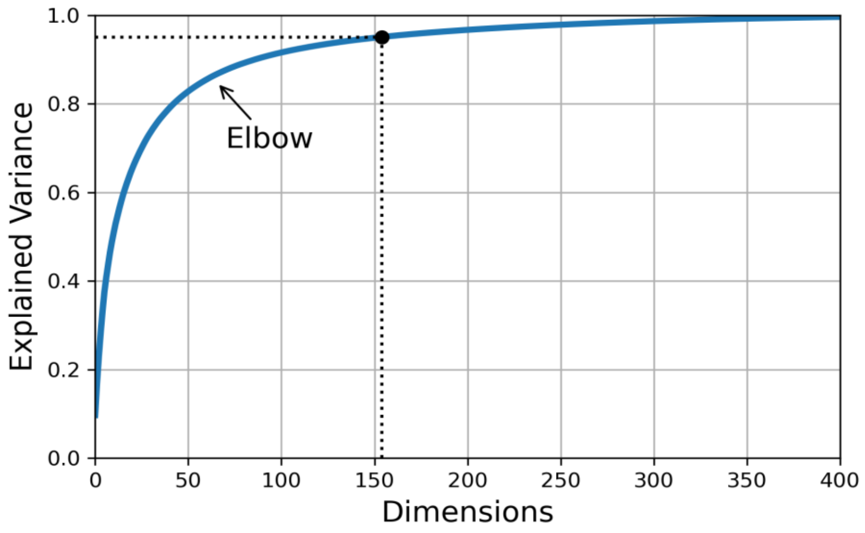
```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

```
pca.n_components_
```

```
154
```

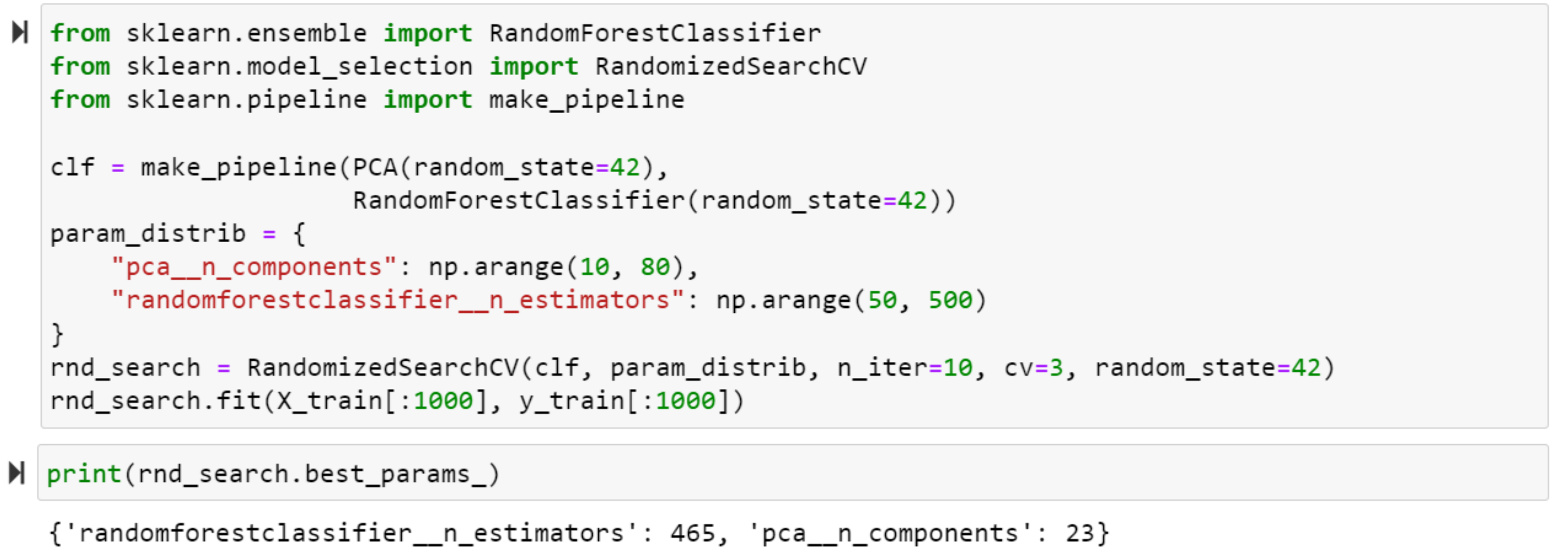
Choosing the Right Number of Dimensions

- Plot the explained variance as a function of the number of dimensions, there will usually be an elbow in the curve, where the explained variance stops growing fast:



Choosing the Right Number of Dimensions

➢ If you are using dimensionality reduction as a preprocessing step for a supervised learning task, then you can tune the number of dimensions as you would any other hyperparameter.



```python
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])
```

```python
print(rnd_search.best_params_)
```

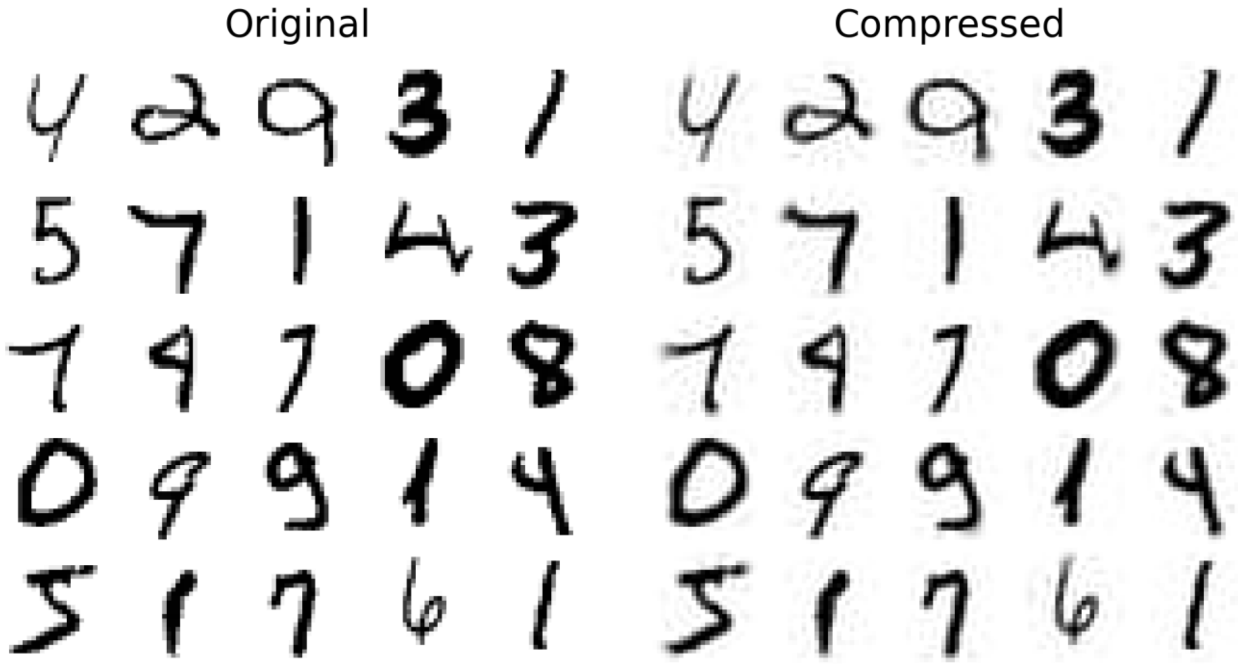
```
{'randomforestclassifier__n_estimators': 465, 'pca__n_components': 23}
```

PCA for Compression

- After dimensionality reduction, the training set takes up less space.
  - E.g. MNIST dataset after PCA is less than 20% of its original size, and we only lost 5% of its variance!
- It is possible to decompress the reduced dataset 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*.
- Use the `inverse_transform()` method to decompress:

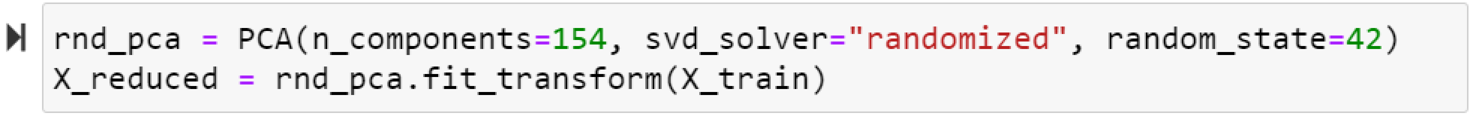
```
X_recovered = pca.inverse_transform(X_reduced)
```

PCA for Compression



Randomized PCA

- If you set the `svd_solver` hyperparameter to "randomized", Scikit-Learn uses a stochastic algorithm called *randomized PCA* that quickly finds an *approximation* of the first d principal components.
- Its computational complexity is $O(m \times d^2) + O(d^3)$, instead of $O(m \times n^2) + O(n^3)$ for the full SVD approach.
  - Much faster than full SVD when d is much smaller than n.

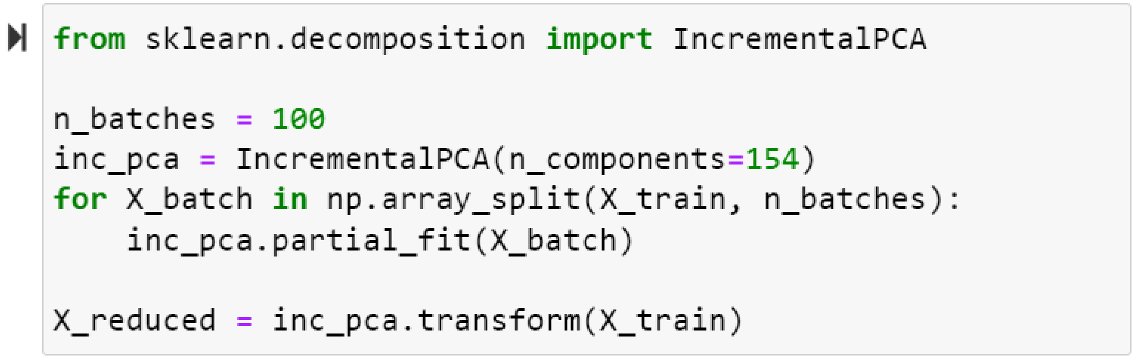


```
rnd_pca = PCA(n_components=154, svd_solver="randomized", random_state=42)
X_reduced = rnd_pca.fit_transform(X_train)
```

- By default, `svd_solver` is actually set to "`auto`": Scikit-Learn uses the randomized PCA algorithm automatically if $\max(m, n) > 500$ and `n_components` is an integer smaller than 80% of $\min(m, n)$.

Incremental PCA

- PCA requires the whole training set to fit in memory.
- *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.



```
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)
```

3.
Random Projection

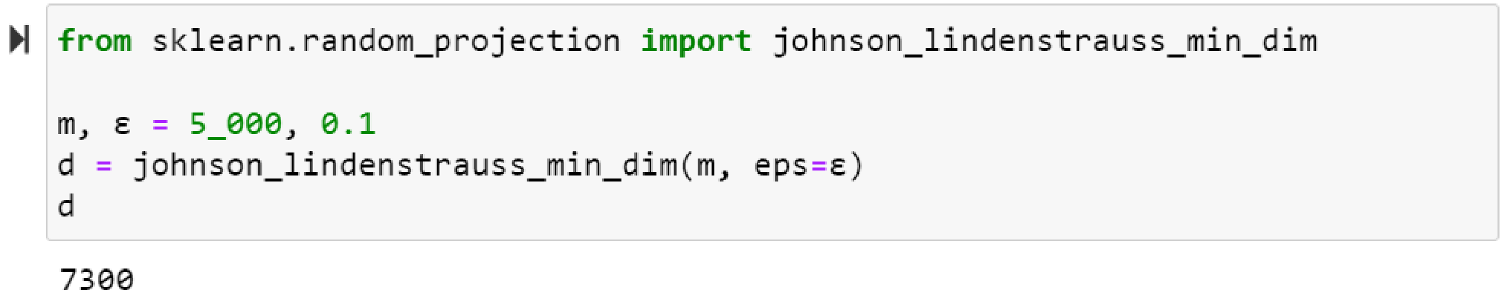
Random Projection

- For very high-dimensional datasets, PCA can be too slow.
  - Even if you use randomized PCA, its computational complexity is still $O(m \times d^2) + O(d^3)$, so the target number of dimensions d must not be too large.
- If you are dealing with a dataset with tens of thousands of features or more (e.g., images), you should consider using *random projection* instead.
- The random projection algorithm projects the data to a lower dimensional space using a random linear projection.
  - Very likely to preserve the distance according to a lemma by Johnson and Lindenstrauss.

Random Projection

- Johnson and Lindenstrauss showed 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 is:

$$d \geq 4\log(m) / (½\,\varepsilon^2 - ⅓\,\varepsilon^3)$$



```
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
```

Random Projection

- Generate a random matrix $\boldsymbol{P}$ of shape $[d, n]$, where each item is sampled randomly from a Gaussian distribution with mean 0 and variance $1/d$, and use it to project a dataset from n dimensions down to d:

```python
n = 20_000
np.random.seed(42)
P = np.random.randn(d, n) / np.sqrt(d)  # std dev = square root of variance

X = np.random.randn(m, n)  # generate a fake dataset
X_reduced = X @ P.T
```

- Scikit-Learn offers a `GaussianRandomProjection` class to do this:

```python
from sklearn.random_projection import GaussianRandomProjection

gaussian_rnd_proj = GaussianRandomProjection(eps=ε, random_state=42)
X_reduced = gaussian_rnd_proj.fit_transform(X)  # same result as above
```

Sparse Random Projection

- Scikit-Learn also provides `SparseRandomProjection`.
  - The main difference is that the random matrix is sparse.
- This means it uses much less memory: about 25 MB instead of almost 1.2 GB in the previous example!
  - And much faster, both to generate the random matrix and to reduce dimensionality: about 50% faster in this case.
- If the input is sparse, the transformation keeps it sparse.
- It enjoys the same distance-preserving property, and the quality of the dimensionality reduction is comparable.
- In short, it's usually preferable to use this transformer instead of the first one, especially for large or sparse datasets.

4.
Locally Linear Embedding

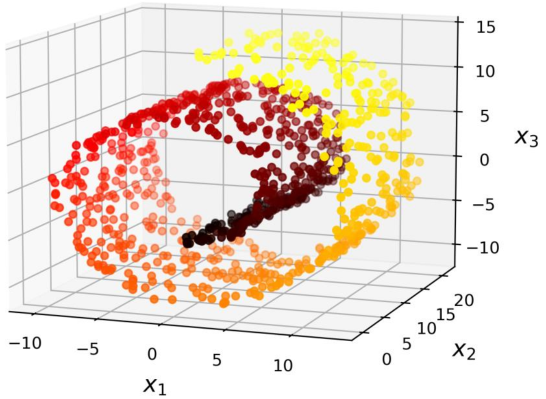
Locally Linear Embedding

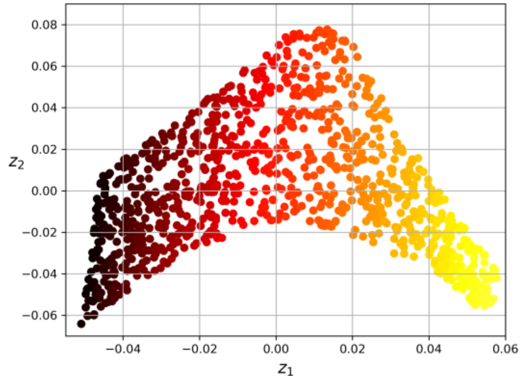
- *Locally linear embedding* (LLE) is a *nonlinear dimensionality reduction* (NLDR) technique that uses manifold learning.
- LLE:
  1. measures how each training instance linearly relates to its nearest neighbors
  2. looks for a low-dimensional representation of the training set where these local relationships are best preserved.
- This approach makes it particularly good at unrolling twisted manifolds, especially when there is not too much noise.

LLE in Scikit-Learn

➢ The following code makes a Swiss roll, then uses Scikit-Learn's `LocallyLinearEmbedding` class to unroll it:

```
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)
```





Complexity of LLE

- $O(m\log(m)n\log(k))$ for finding the k-nearest neighbors.
- $O(mnk^3)$ for optimizing the weights
- $O(dm^2)$ for constructing the low-dimensional representations.
  - The m^2 in the last term makes this algorithm scale poorly to very large datasets.
- LLE is significantly more complex than projection-based techniques.
  - It can construct better low-dimensional representations, especially if the data is nonlinear.

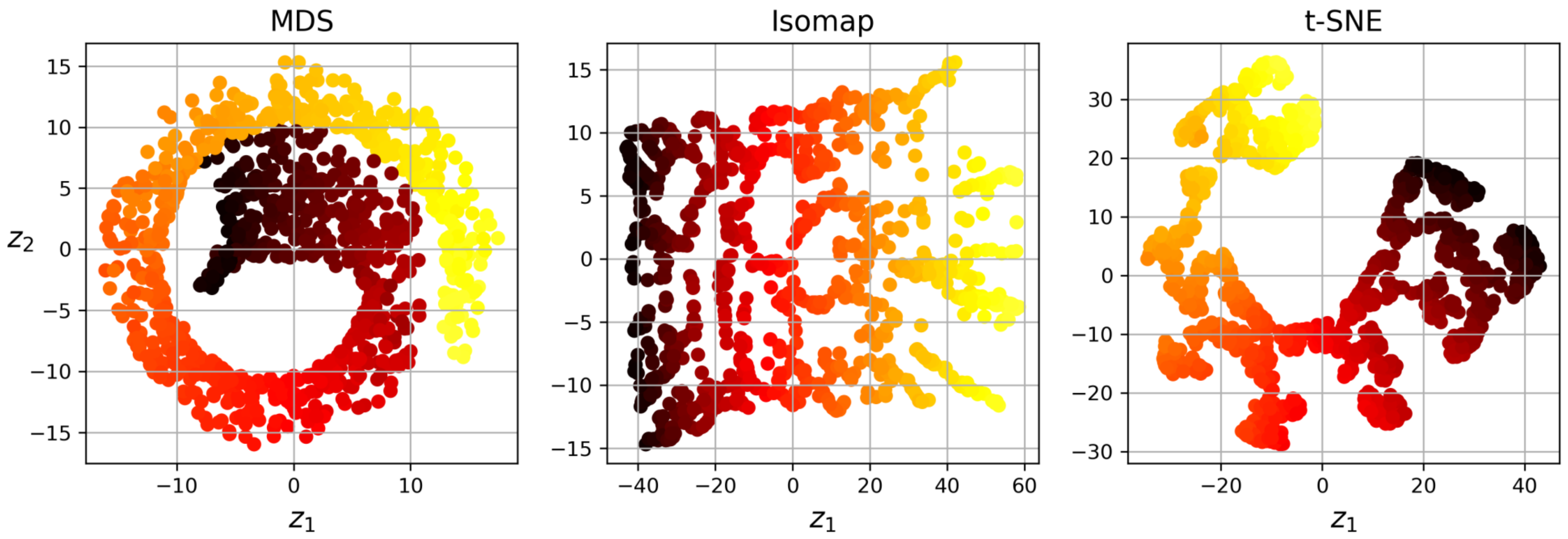
Other Dimensionality Reduction Techniques

- `sklearn.manifold.MDS`: *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 low-dimensional data.
- `sklearn.manifold.Isomap`: *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.

Other Dimensionality Reduction Techniques

- `sklearn.manifold.TSNE`: *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.discriminant_analysis.LinearDiscriminantAnalysis`: *Linear discriminant analysis* (LDA) is a classification algorithm that, during training, learns the most discriminative axes between classes.
  - These axes can then be used to define a hyperplane onto which to project the data.
  - The projection will keep classes as far apart as possible.

Other Dimensionality Reduction Techniques


MDS
Isomap
t-SNE
z_2
z_1
z_1
z_1