AIN2002

Introduction to Data Science

Dimensionality Reduction

ML Overview

Dr. Fatih KAHRAMAN

fatih.kahraman@bau.edu.tr

Introduction

- 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*.

Introduction

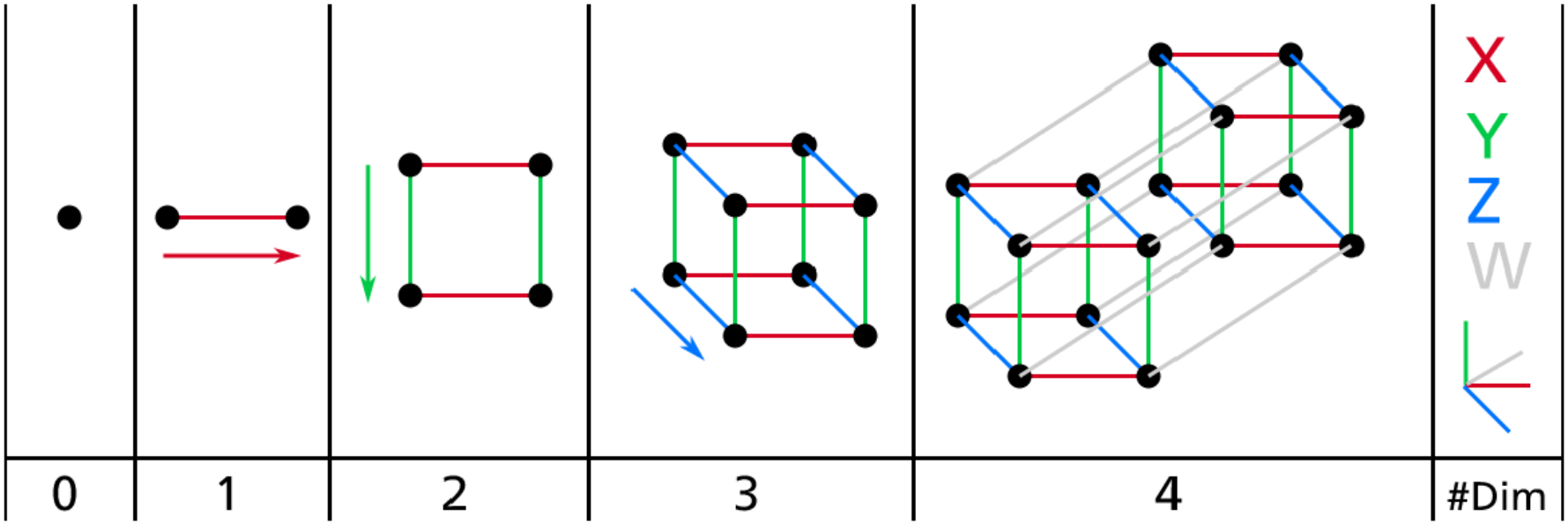
- Fortunately, in real-world problems, it is often possible to reduce the number of features considerably, turning an intractable problem into a tractable one.
- For example, consider the MNIST images
  - the pixels on the image borders are almost always white, so you could completely drop these pixels from the training set without losing much information
  - Moreover, two neighboring pixels are often highly correlated: if you merge them into a single pixel (e.g., by taking the mean of the two pixel intensities), you will not lose much information

Content

- In this module, 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
  - 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, let alone a 200-dimensional ellipsoid bent in a 1,000-dimensional space.

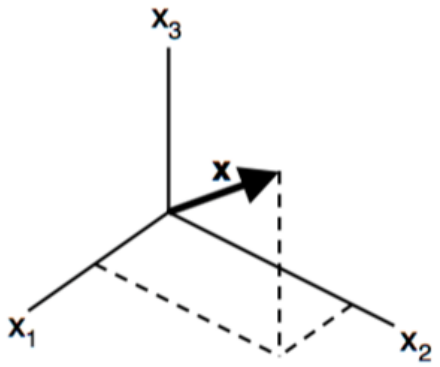


The Curse of Dimensionality

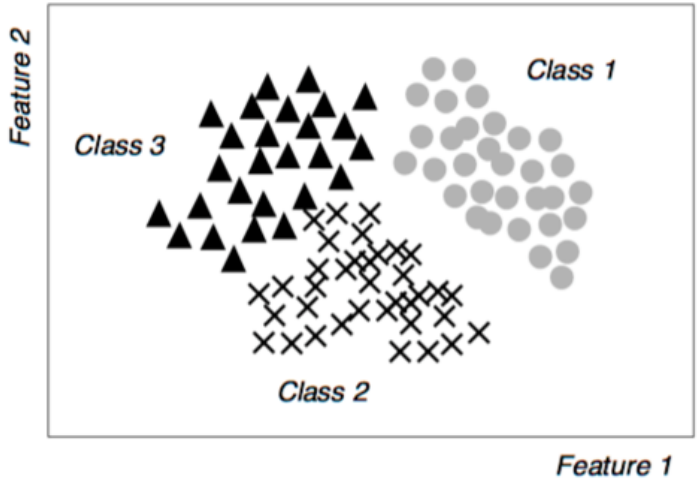
> X is d-dimensional feature vector. If d=3, this vector create 3D feature space.

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \\ x_d \end{bmatrix}$$

Feature vector



Feature space (3D)



Scatter plot (2D)

The Curse of Dimensionality

Fun fact:

- if you pick a random point in a unit square (a 1 × 1 square), it will have only about a 0.4% chance of being located less than 0.001 from a border
- In a 10,000-dimensional unit hypercube, this probability is greater than 99.999999%.
- Most points in a high-dimensional hypercube are very close to the border

The Curse of Dimensionality

Fun fact:

> if you pick two points randomly in a unit square, the distance between these two points will be, on average, roughly 0.52

The Curse of Dimensionality

Fun fact:

- 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

The Curse of Dimensionality

Fun fact:

- 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 Curse of Dimensionality

Fun fact:

- 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 is about 408.25

The Curse of Dimensionality

Fun fact:

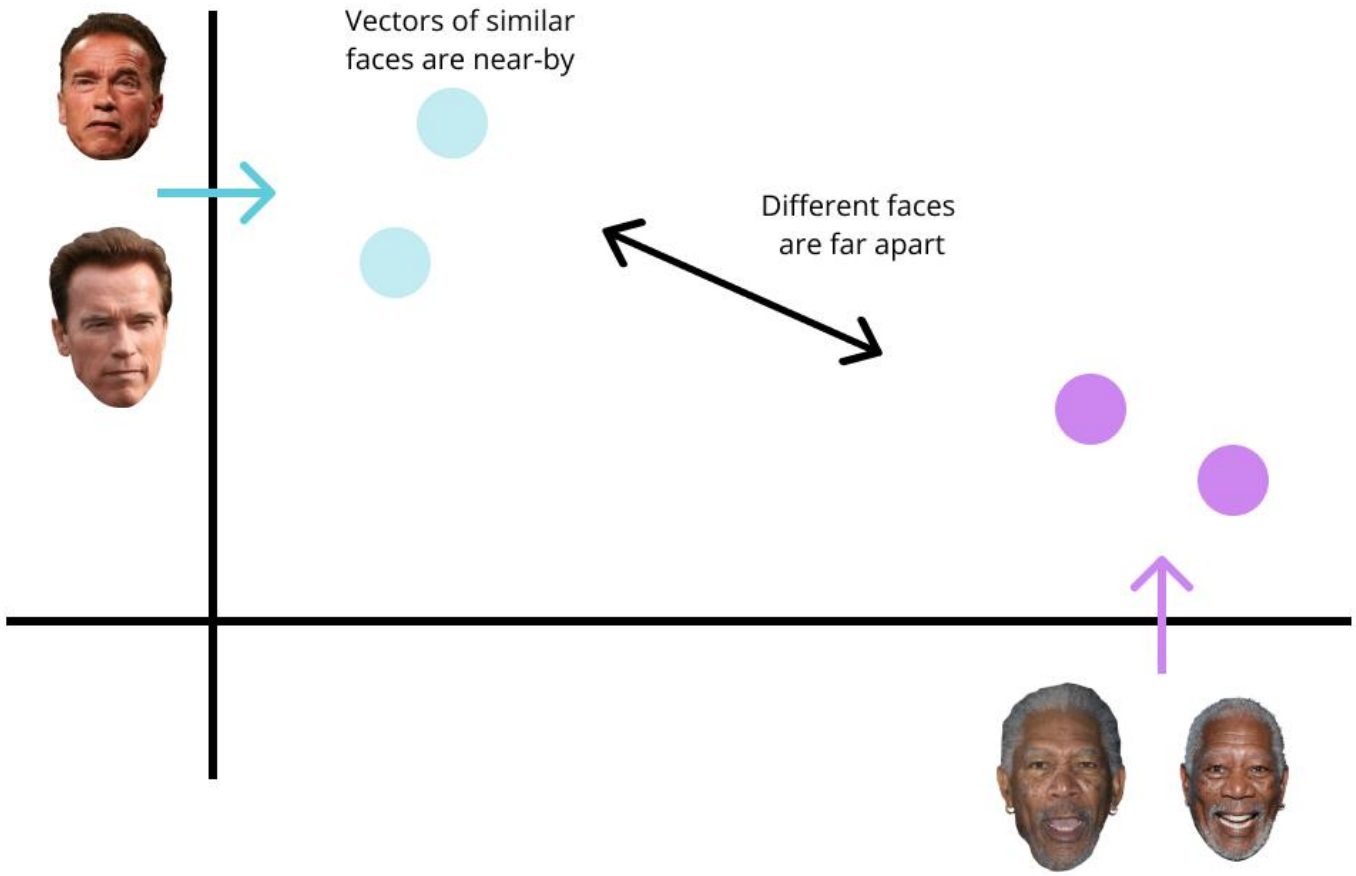
- if you pick two points randomly in a unit square, the distance between these two points will be, on average, roughly 0.52
- If you pick two random points in a unit 3D cube, the average distance will be roughly 0.66
- But what about two points picked randomly in a 1,000,000-dimensional hypercube?
  - The average distance is 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, 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

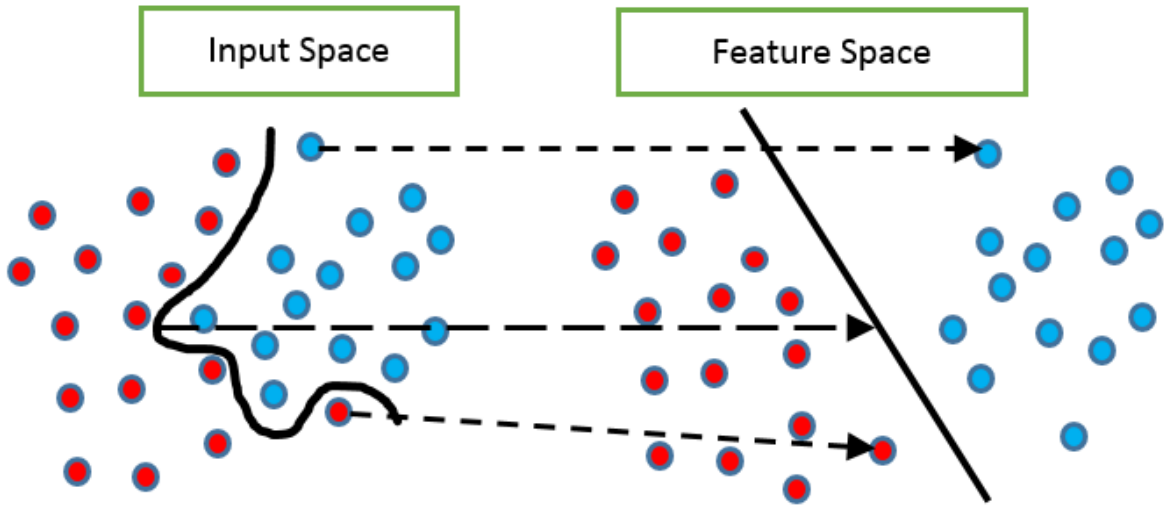
Feature Space

Feature Space



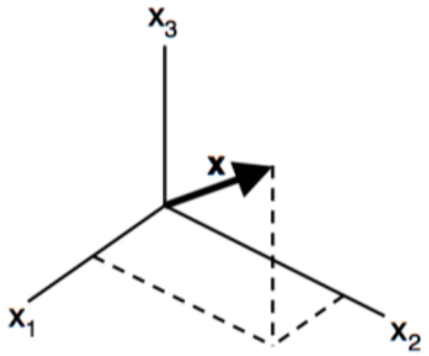
2-Dimension Face Recognition Feature Space

Feature Space

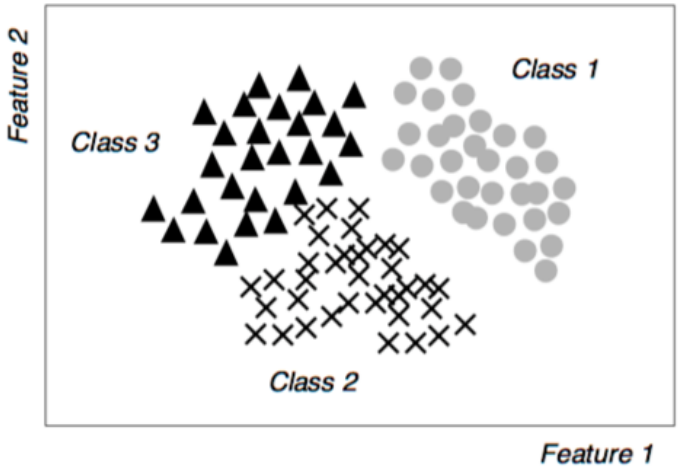


$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \\ x_d \end{bmatrix}$$

Feature vector

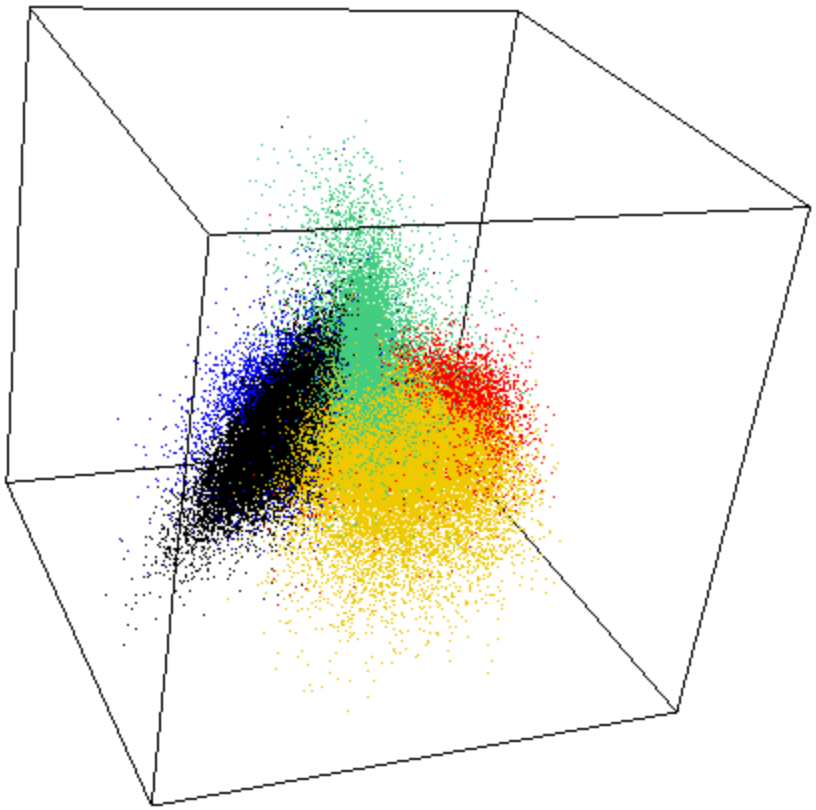


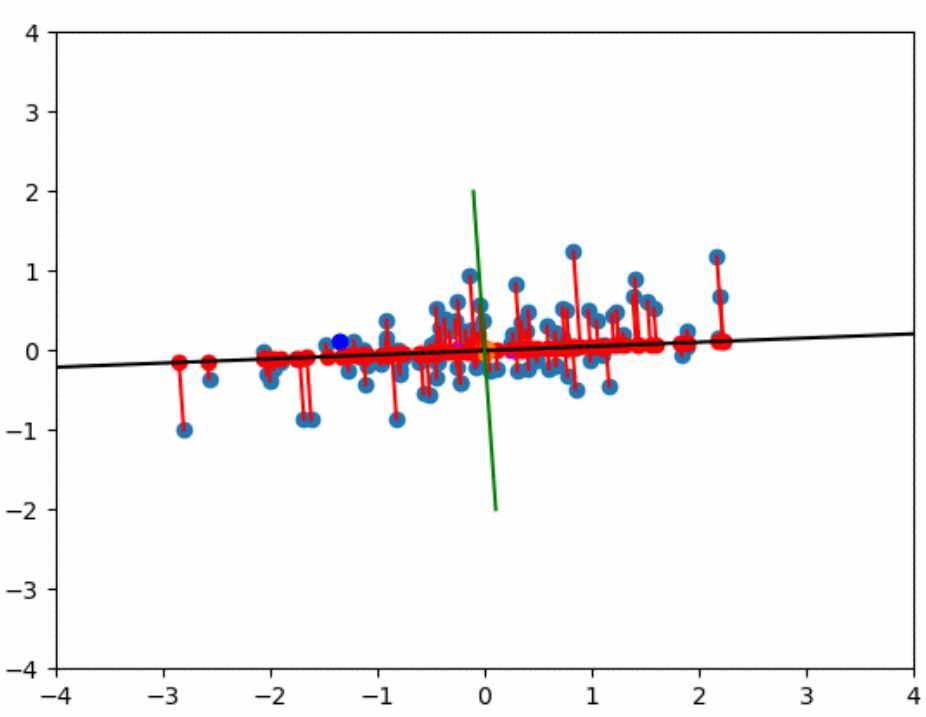
Feature space (3D)



Scatter plot (2D)

Feature Space

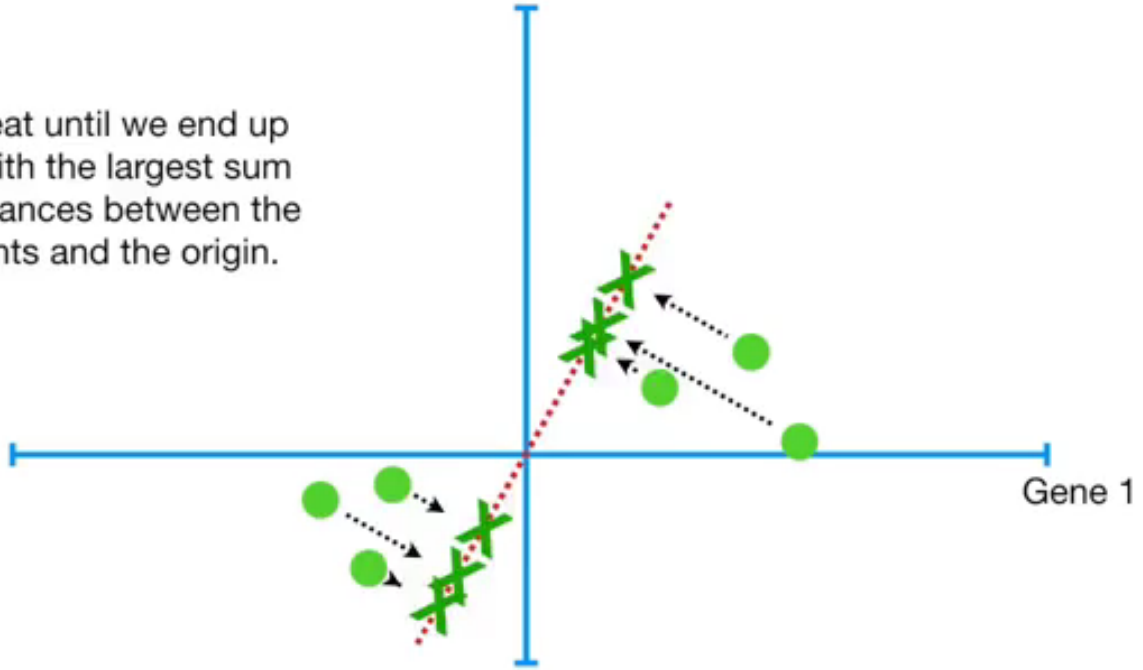




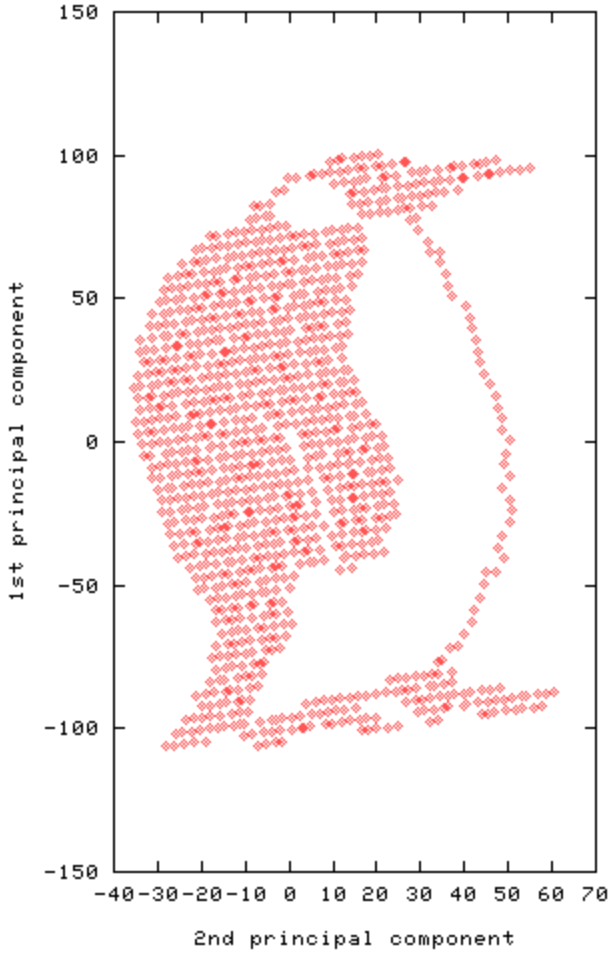
Feature Space

$d_1^2 + d_2^2 + d_3^2 + d_4^2 + d_5^2 + d_6^2$ = sum of squared distances = SS(distances)

…and we repeat until we end up with the line with the largest sum of squared distances between the projected points and the origin.

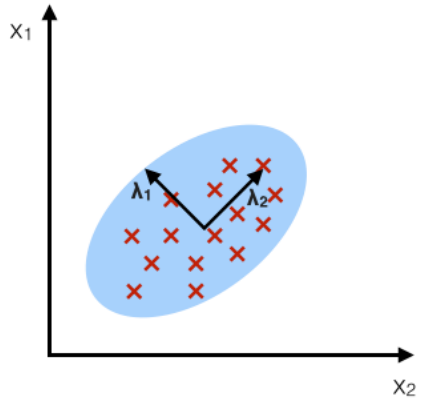


Feature Space

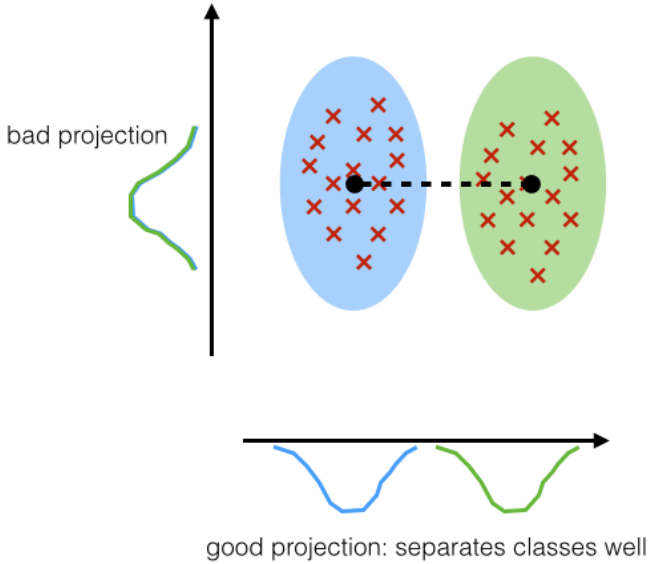


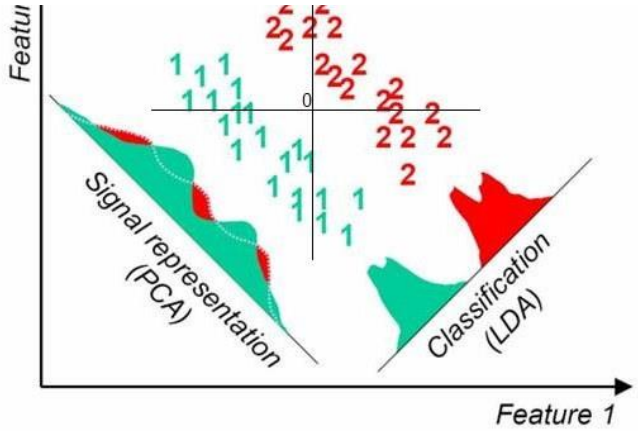
PCA vs. LDA

PCA:
component axes that maximize the variance

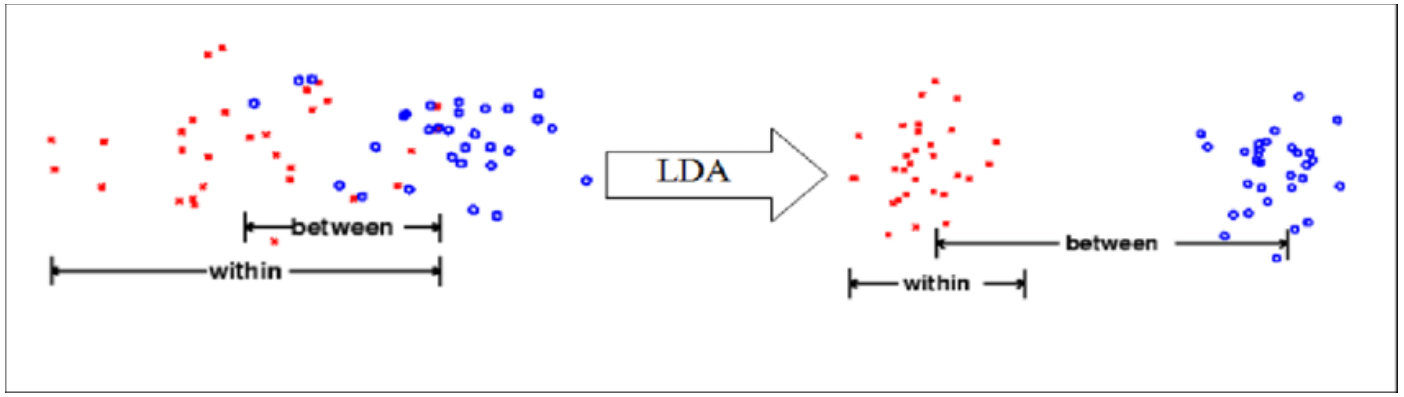


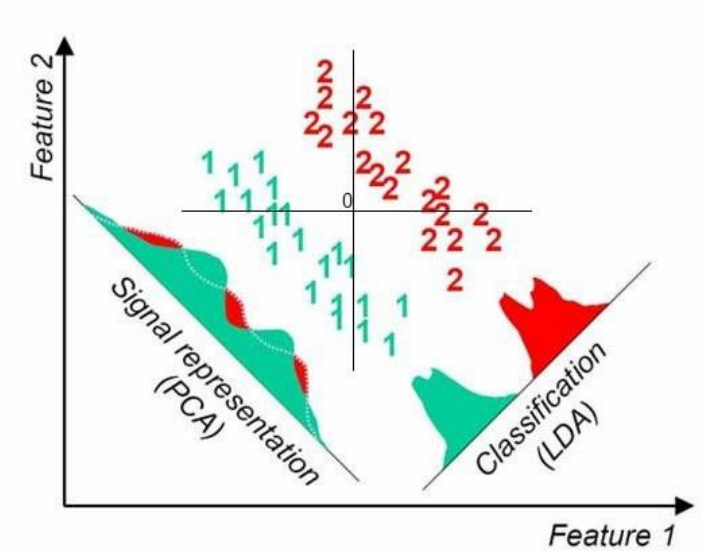
LDA:
maximizing the component axes for class-separation

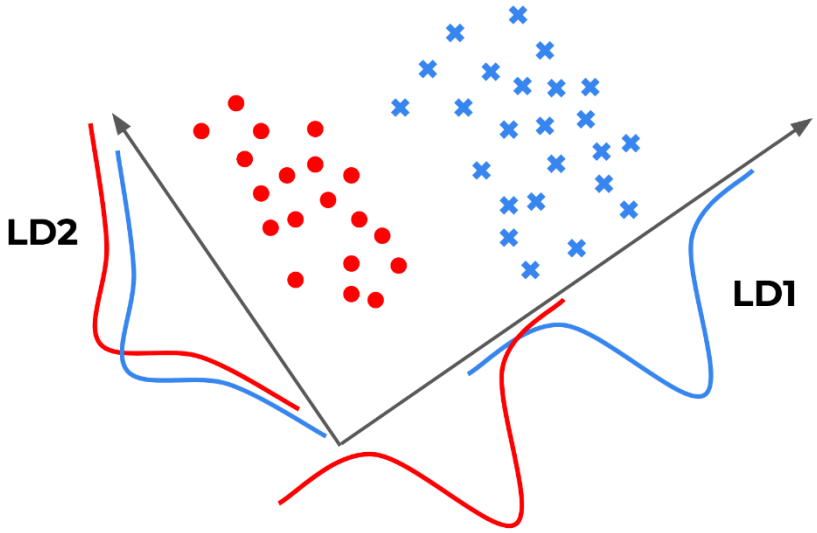




Between / Within Class Variance





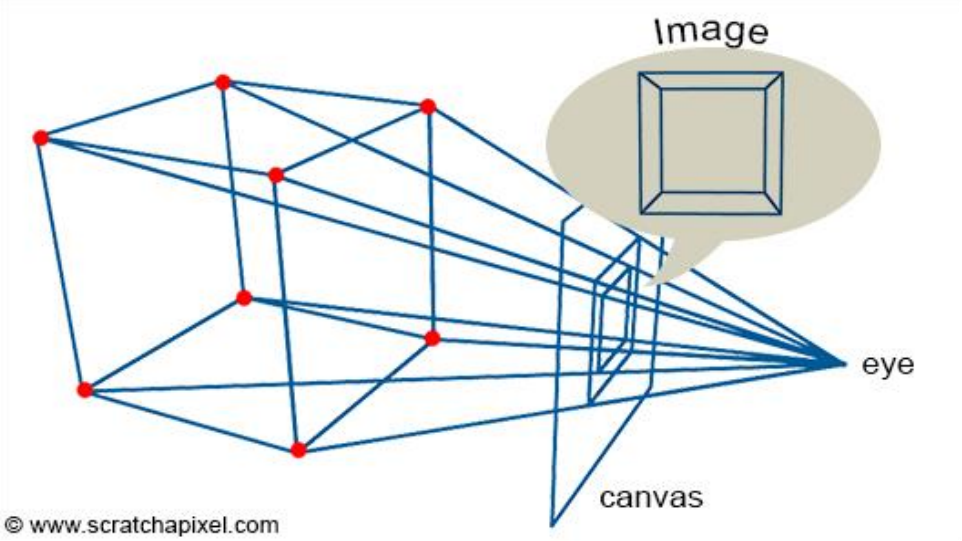


Main Approaches for Dimensionality Reduction

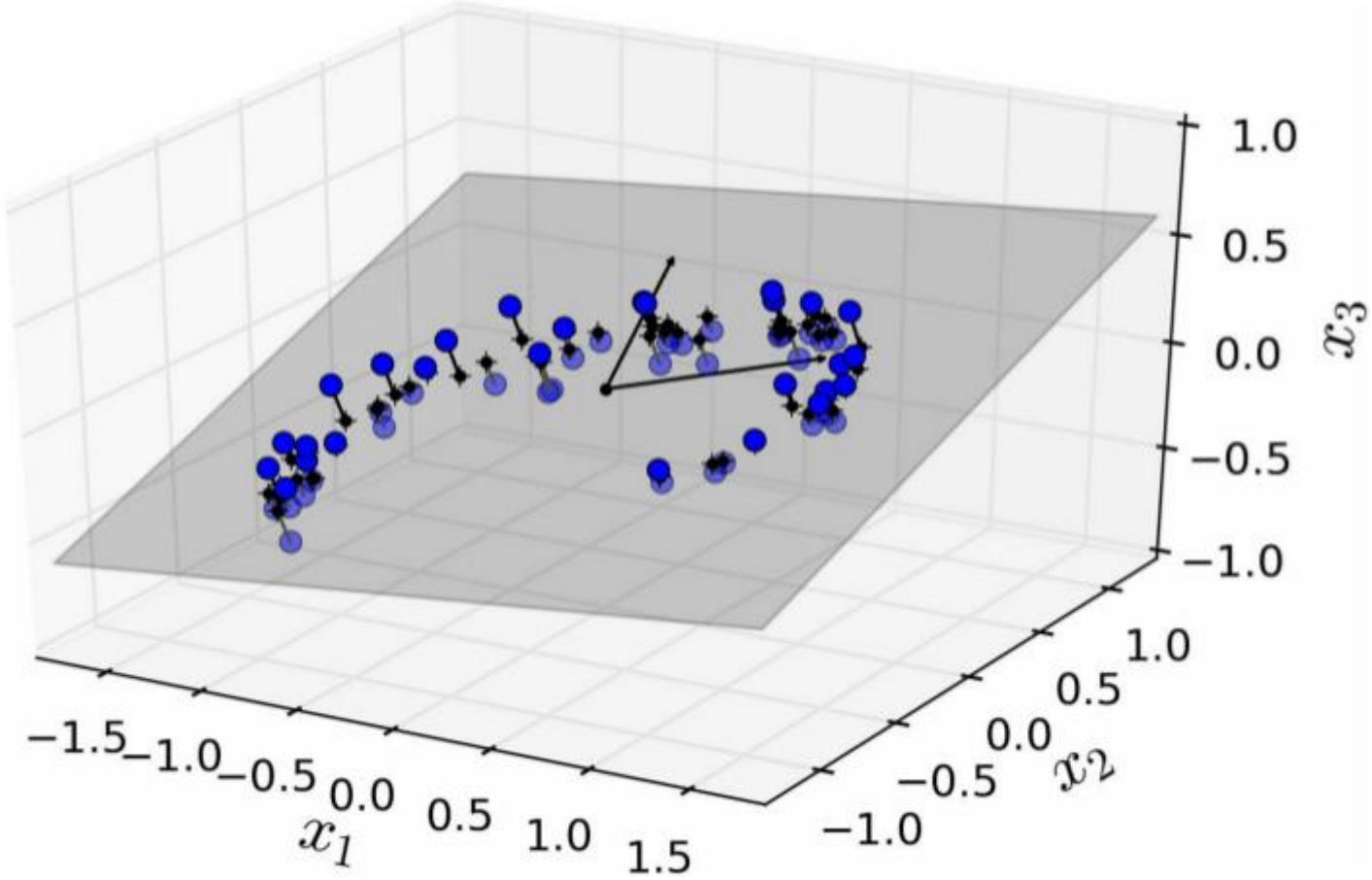
- Before we dive into specific dimensionality reduction algorithms, let's take a look at the two main approaches to reducing dimensionality
  - Projection
  - Manifold Learning

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 actually lie within (or close to) a much lower-dimensional *subspace* of the high-dimensional space



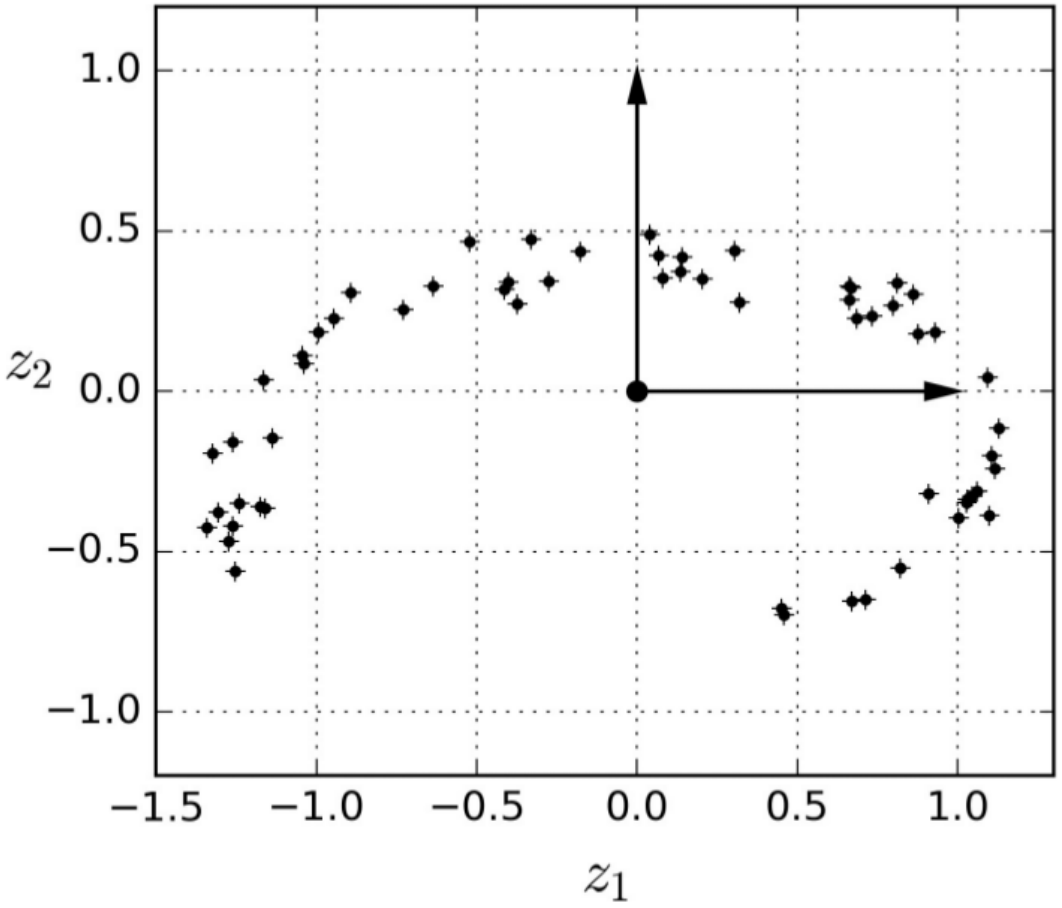
Projection



> Notice that all training instances lie close to a plane: this is a lower-dimensional (2D) subspace of the high-dimensional (3D) space

Projection

> 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

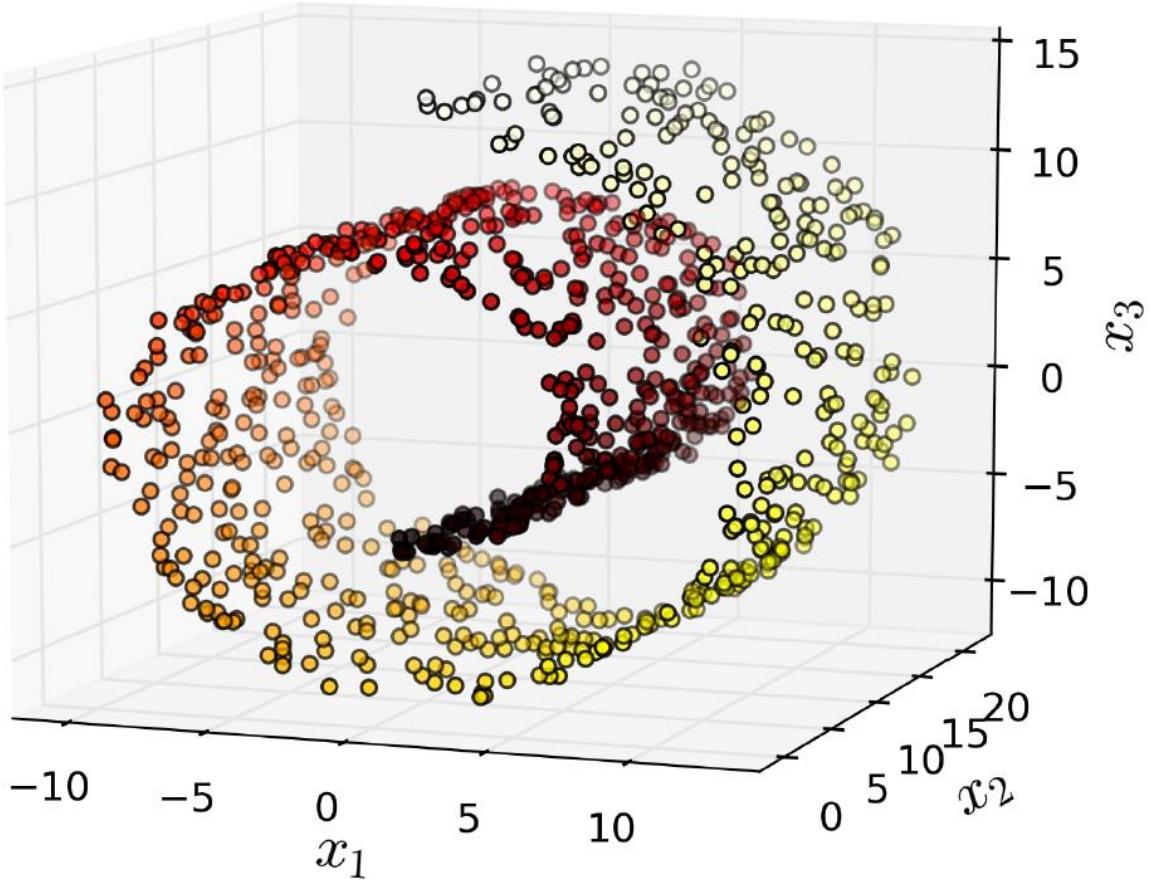


Projection

- However, projection is not always the best approach to dimensionality reduction.
- In many cases the subspace may twist and turn
  - Swiss Roll

Projection

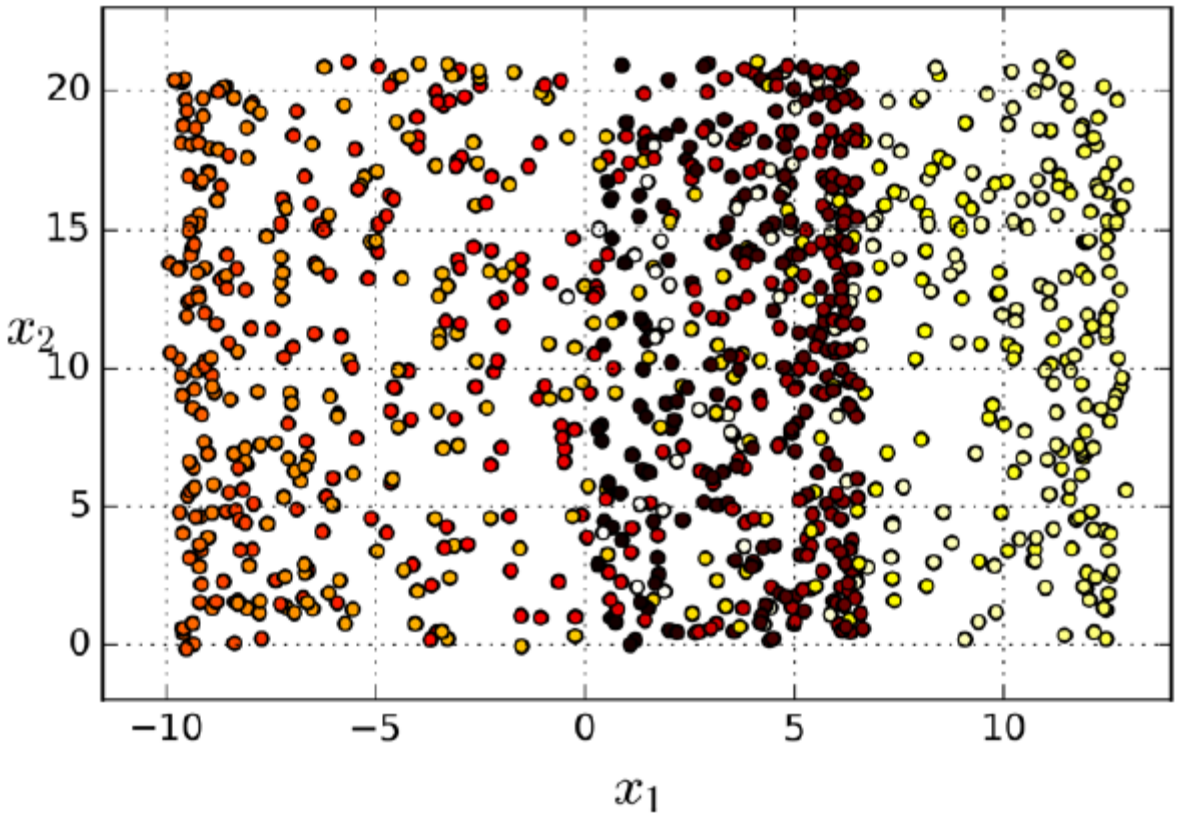
- However, projection is not always the best approach to dimensionality reduction.
- In many cases the subspace may twist and turn



the famous *Swiss roll* toy dataset

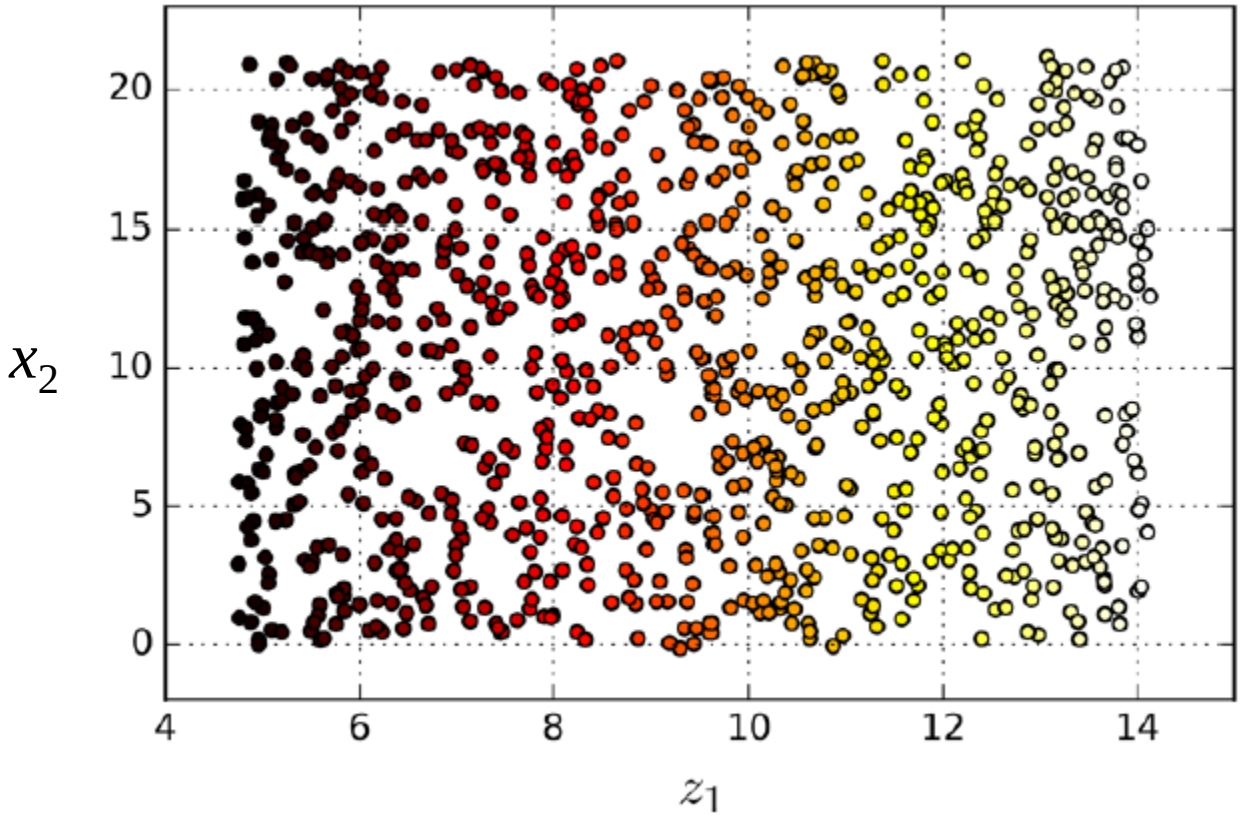
Projection

> Simply projecting onto a plane (e.g., by dropping *x*3) would squash different layers of the Swiss roll together,



Projection

> However, what you really want is to unroll the Swiss roll to obtain the 2D dataset given as the following figure



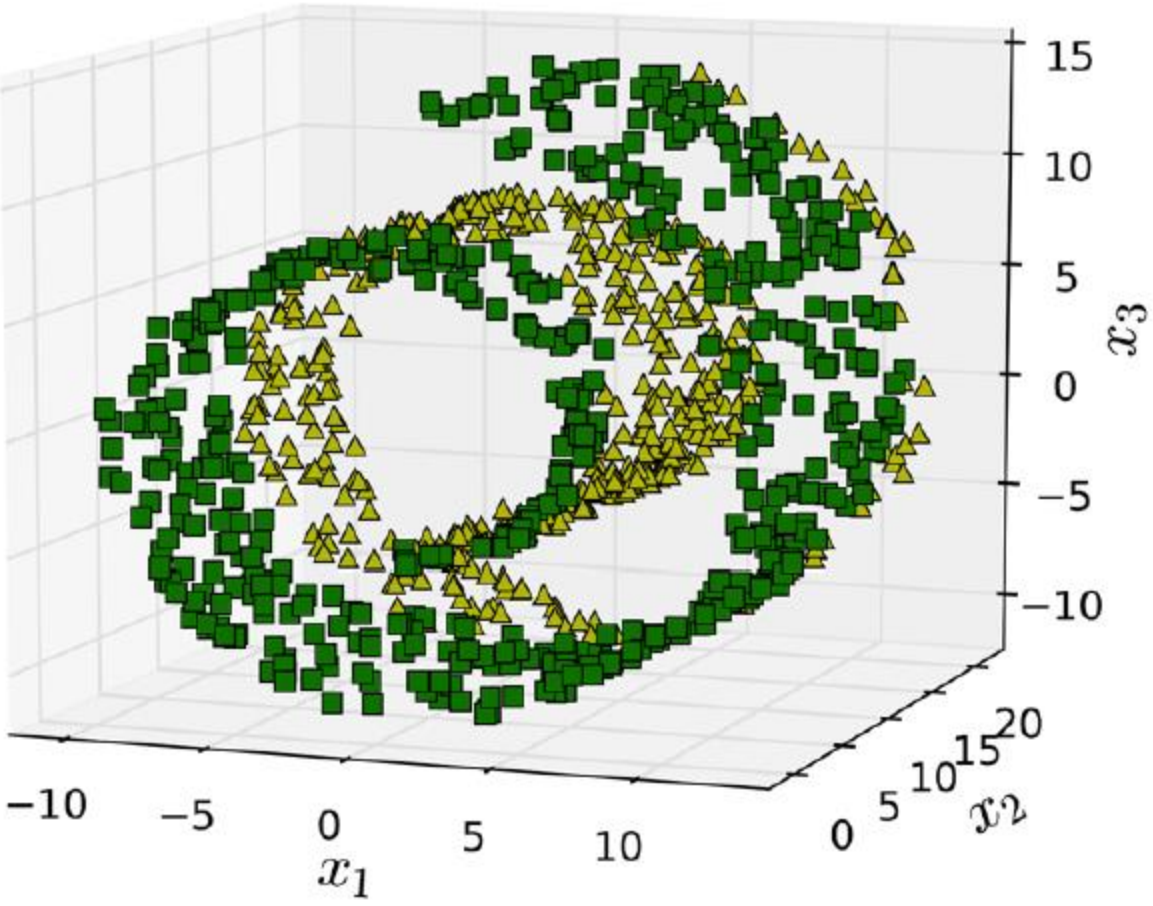
Manifold Learning

- 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
- A d-dimensional manifold is a part of an n-dimensional space (where $d < n$) that locally resembles a d-dimensional hyper-plane.
- 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.

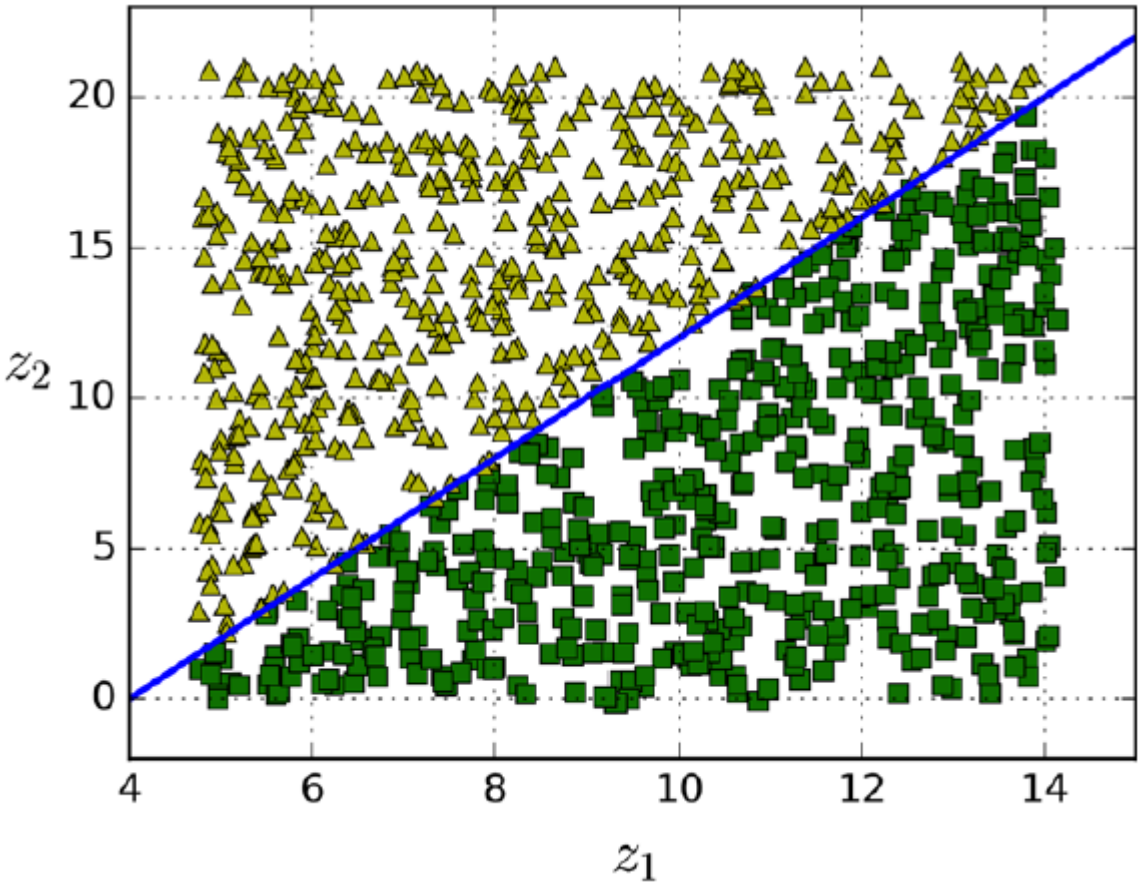
Manifold Learning

- 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*
  - Most real-world high-dimensional datasets lie close to a much lower-dimensional manifold.
  - This assumption is very often empirically observed

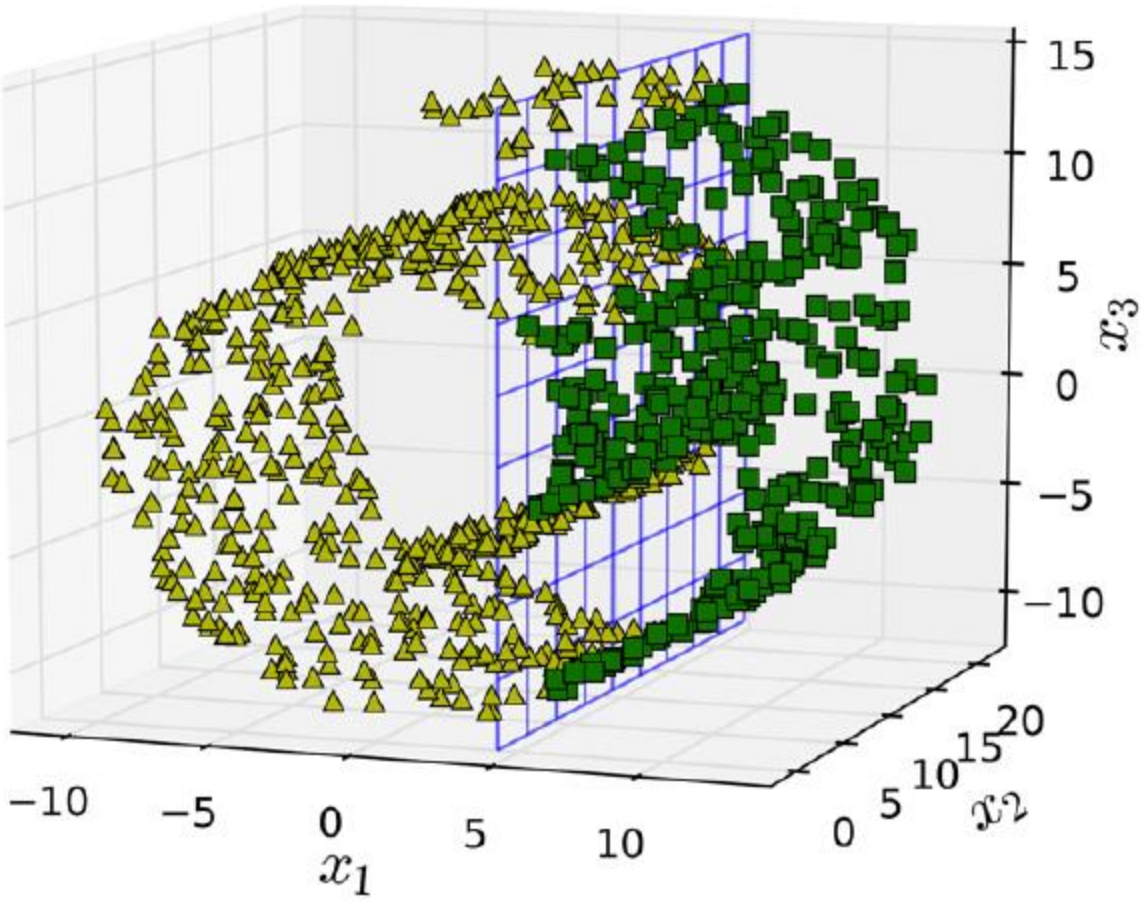
Manifold Learning



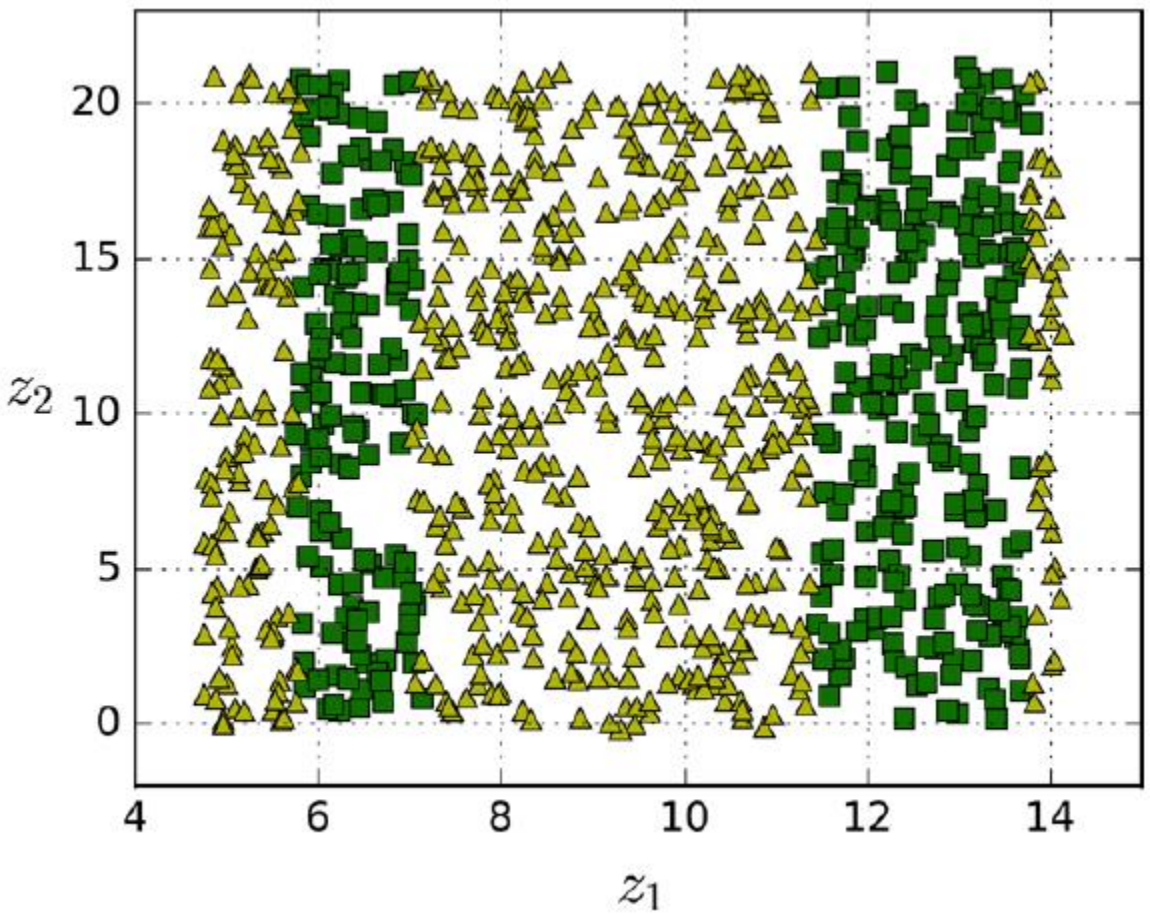
Manifold Learning



Manifold Learning



Manifold Learning

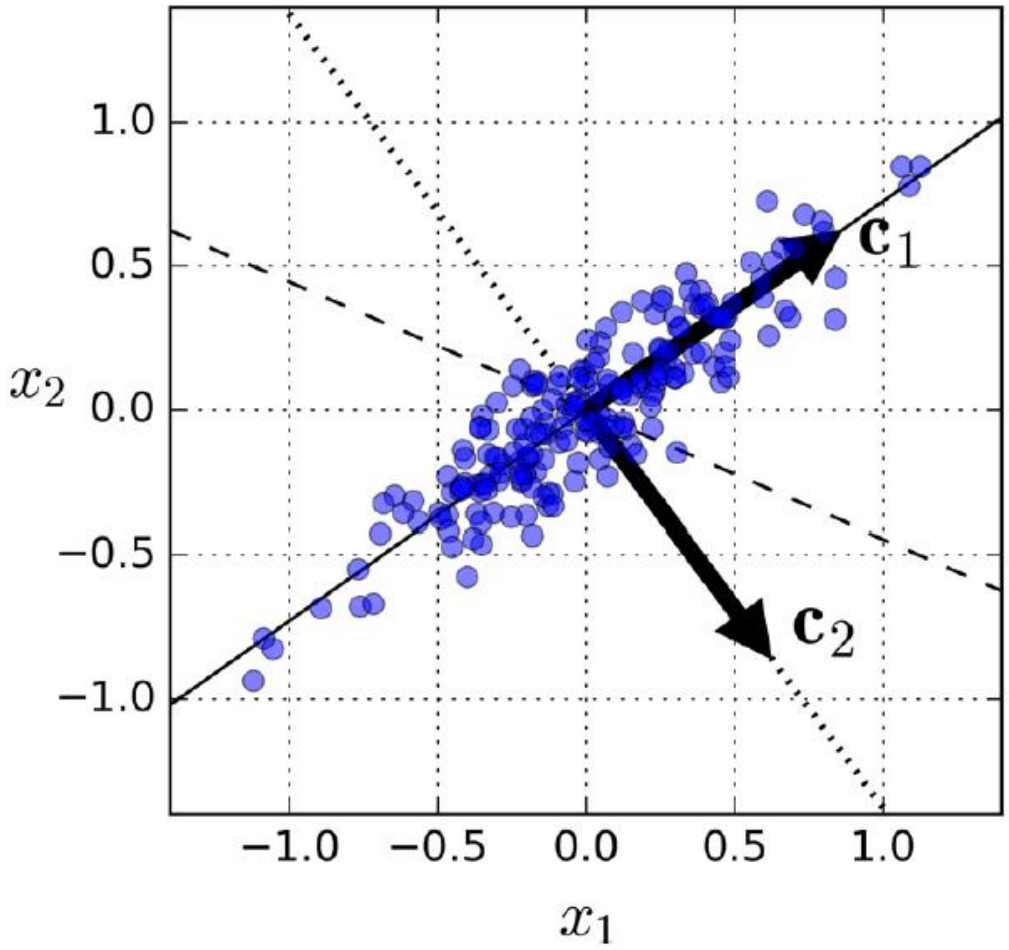


PCA

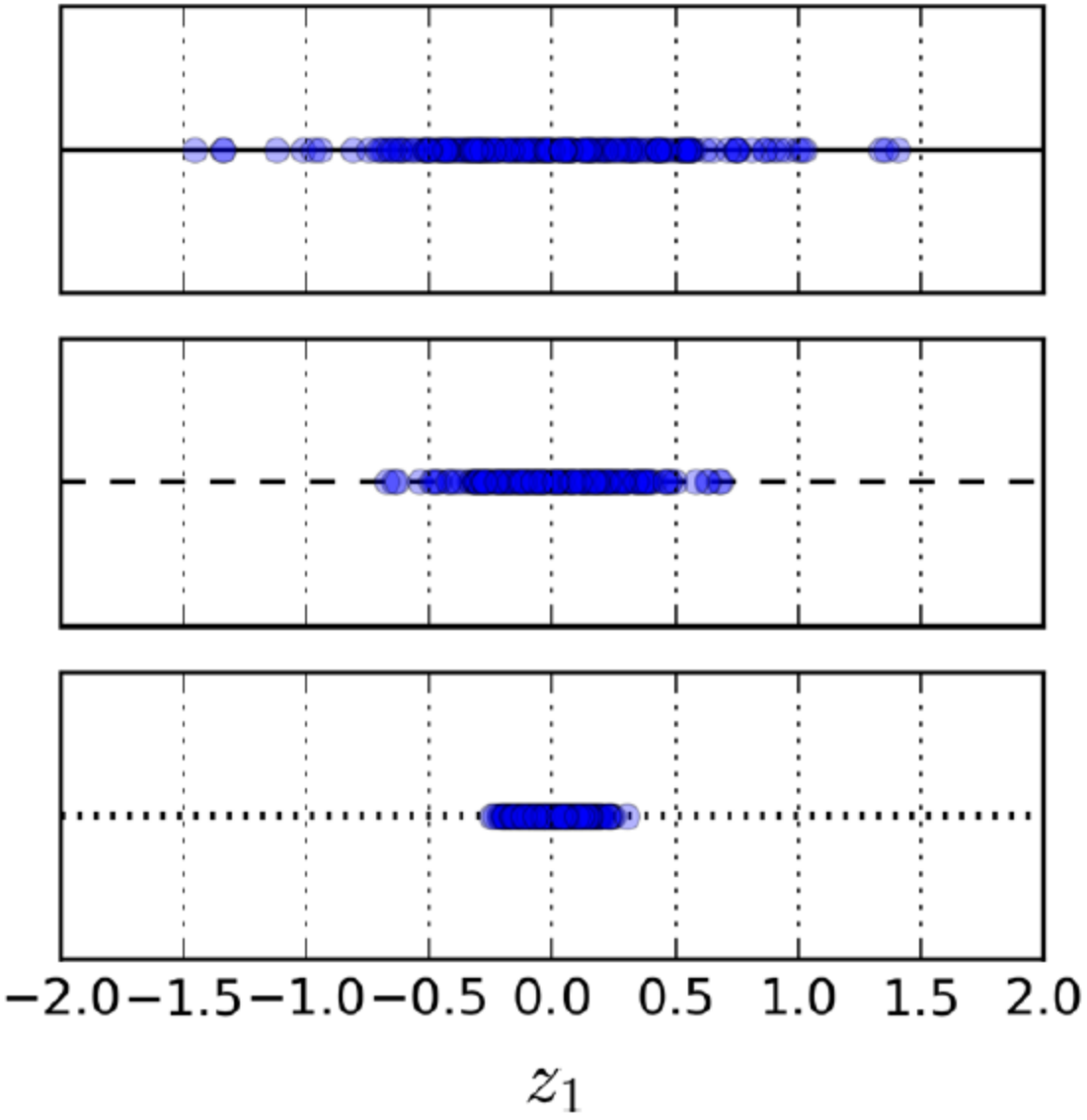
- *Principal Component Analysis* (PCA) is by far the most popular dimensionality reduction algorithm.
- First it identifies the hyper-plane that lies closest to the data, and then it projects the data onto it.

Preserving the Variance

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



Preserving the Variance



Principal Components

- PCA identifies the axis that accounts for the largest amount of variance in the training set.
- *Singular Value Decomposition* (SVD)

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

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

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

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

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

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

Using Scikit-Learn

- Scikit-Learn's PCA class implements PCA using SVD decomposition

```
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
  - The first principal component is equal to **pca.components_.T[:,0]**

Explained Variance Ratio

- Another very useful piece of information is the *explained variance ratio* of each 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

Explained Variance Ratio

```
>>> print(pca.explained_variance_ratio_)
array([ 0.84248607,  0.14631839])
```

- This tells you that
  - **84.2%** of the dataset's variance lies along the first axis,
  - **14.6%** lies along the second axis
  - This leaves less than **1.2%** for the third axis
    - It is reasonable to assume that it probably carries little information.

Choosing the Right Number of Dimensions

- Instead of arbitrarily choosing the number of dimensions to reduce down to, it is generally preferable to choose the number of dimensions that add up to a sufficiently large portion of the variance
- Compute the minimum number of dimensions required to preserve 95% of the training set's variance:

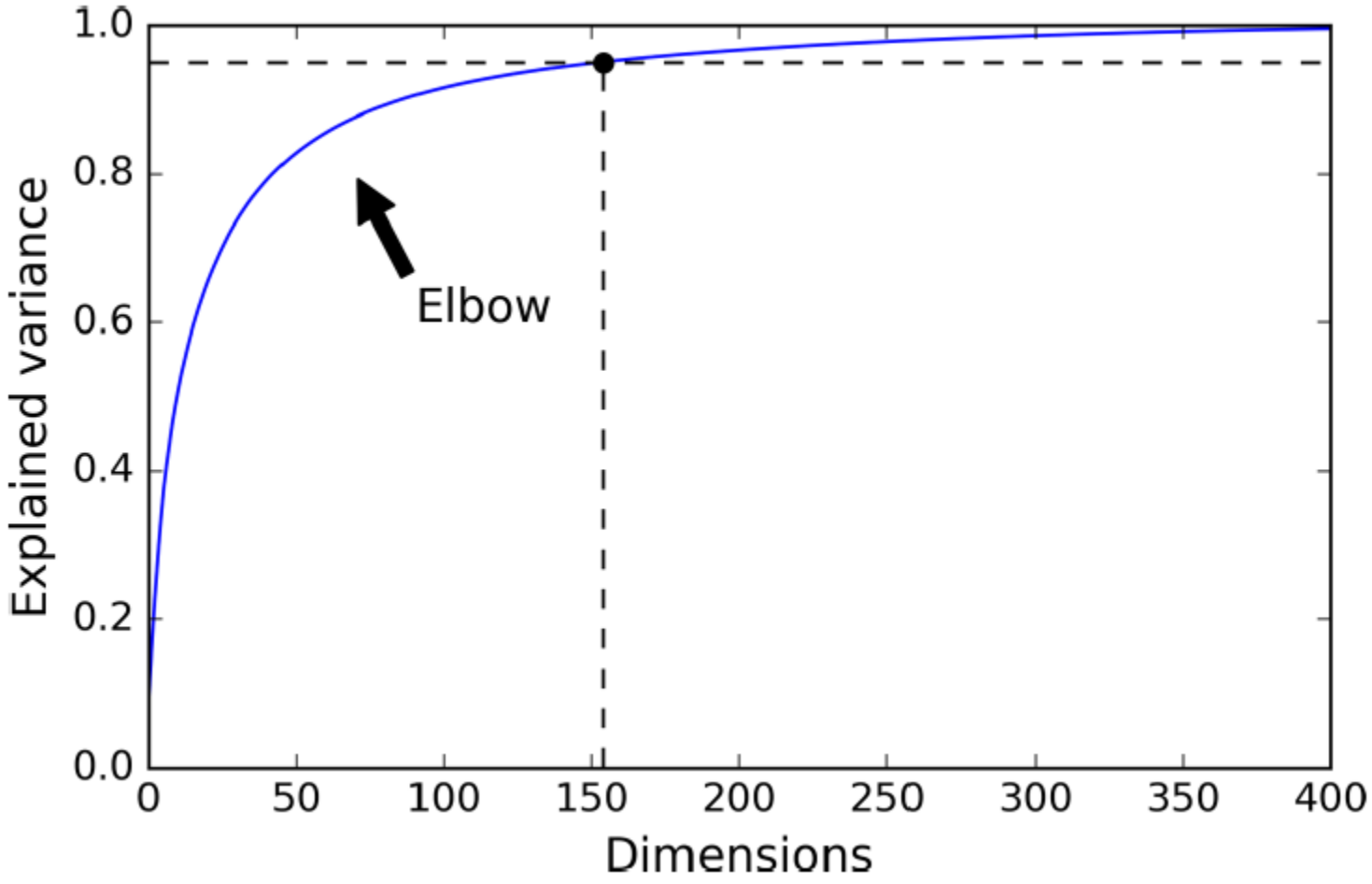
```
pca = PCA()
pca.fit(X)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
```

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

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

Choosing the Right Number of Dimensions

> another option is to plot the explained variance as a function of the number of dimensions



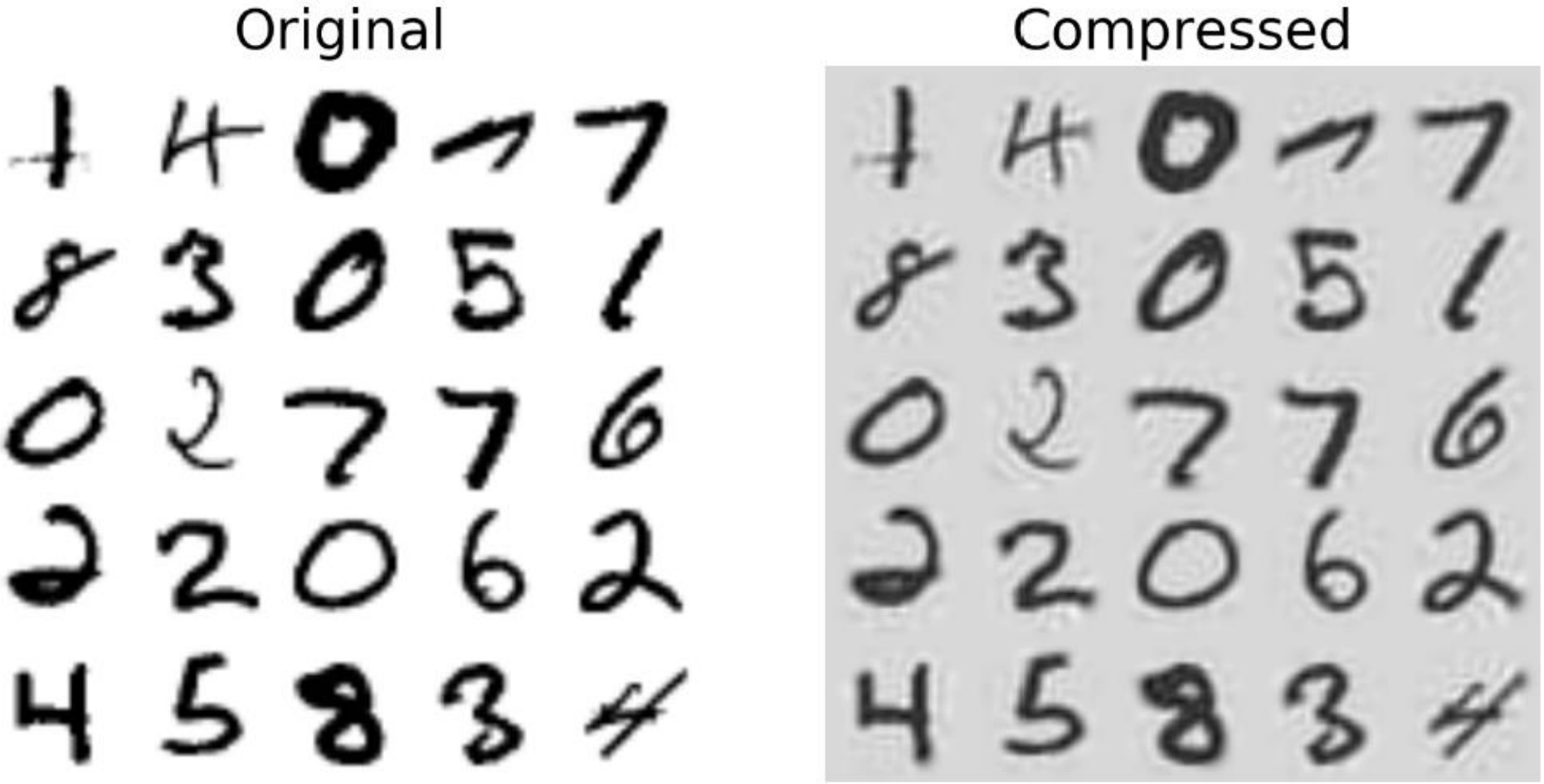
PCA for Compression

- Obviously after dimensionality reduction, the training set takes up much less space
- *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 = PCA(n_components = 154)
X_mnist_reduced = pca.fit_transform(X_mnist)
X_mnist_recovered = pca.inverse_transform(X_mnist_reduced)
```

PCA for Compression



Incremental PCA

- One problem with the preceding implementation of PCA is that it requires the whole training set to fit in memory in order for the SVD algorithm to run
- *Incremental PCA* (IPCA) algorithm splits the training set into mini-batches and feed an IPCA algorithm one mini-batch at a time.
- This is useful for large training sets, and also to apply PCA online

Incremental PCA

```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_mnist, n_batches):
    inc_pca.partial_fit(X_batch)

X_mnist_reduced = inc_pca.transform(X_mnist)
```

Incremental PCA

> Alternatively, you can use NumPy's memmap class

```
X_mm = np.memmap(filename, dtype="float32", mode="readonly", shape=(m, n))

batch_size = m // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mm)
```

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 $O(m \times d^2) + O(d^3)$
  - It is dramatically faster than the previous algorithms when d is much smaller than n

    $O(m \times n^2) + O(n^3)$

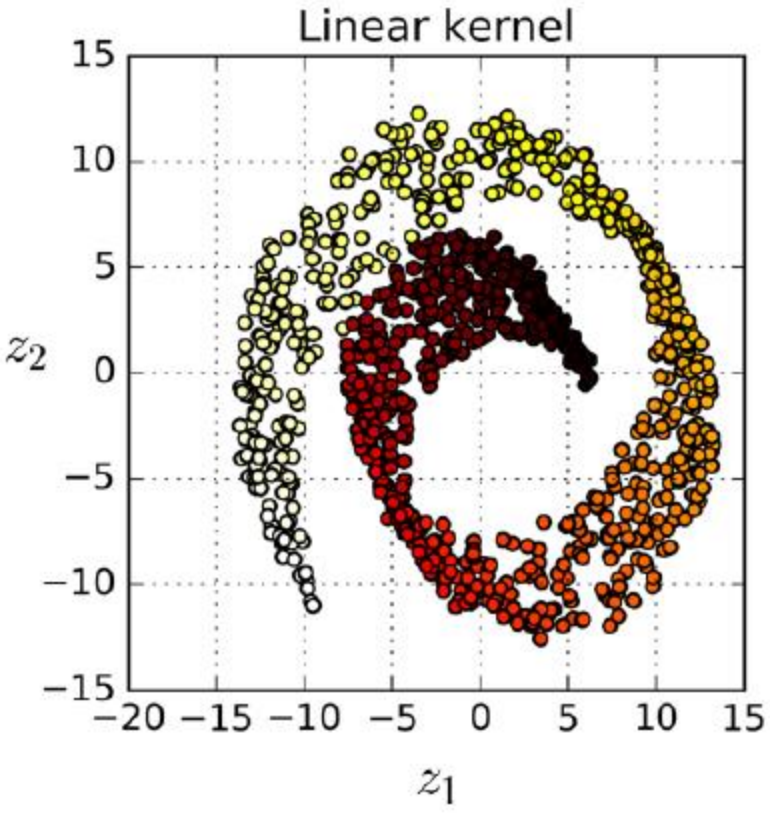
```
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_mnist)
```

Kernel PCA

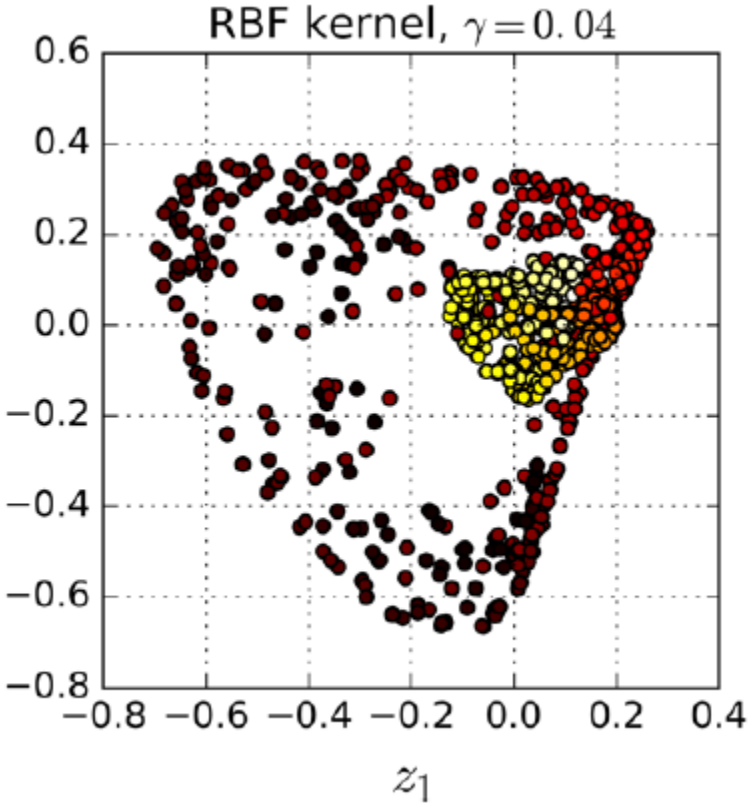
> Scikit-Learn's KernelPCA class to perform kPCA with an RBF kernel

```
from sklearn.decomposition import KernelPCA

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

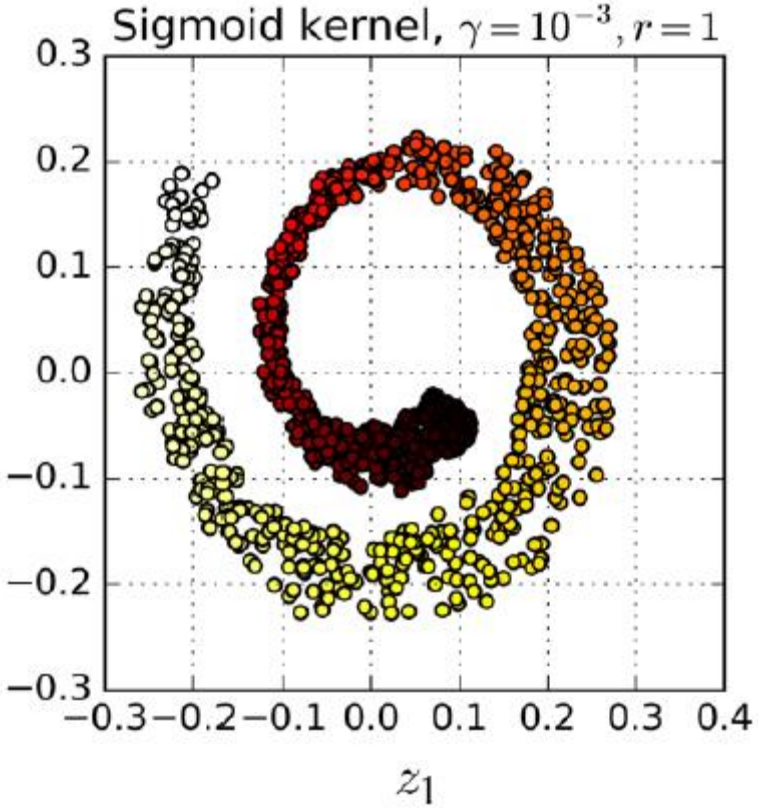
Kernel PCA



Kernel PCA



Kernel PCA


Sigmoid kernel, $\gamma = 10^{-3}, r = 1$
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.3 −0.2 −0.1 0.0 0.1 0.2 0.3 0.4
z_1

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
- Dimensionality reduction is often a preparation step for a supervised learning task (e.g., classification), so you can simply use grid search to select the kernel and hyperparameters that lead to the best performance on that task

Selecting a Kernel and Tuning Hyperparameters

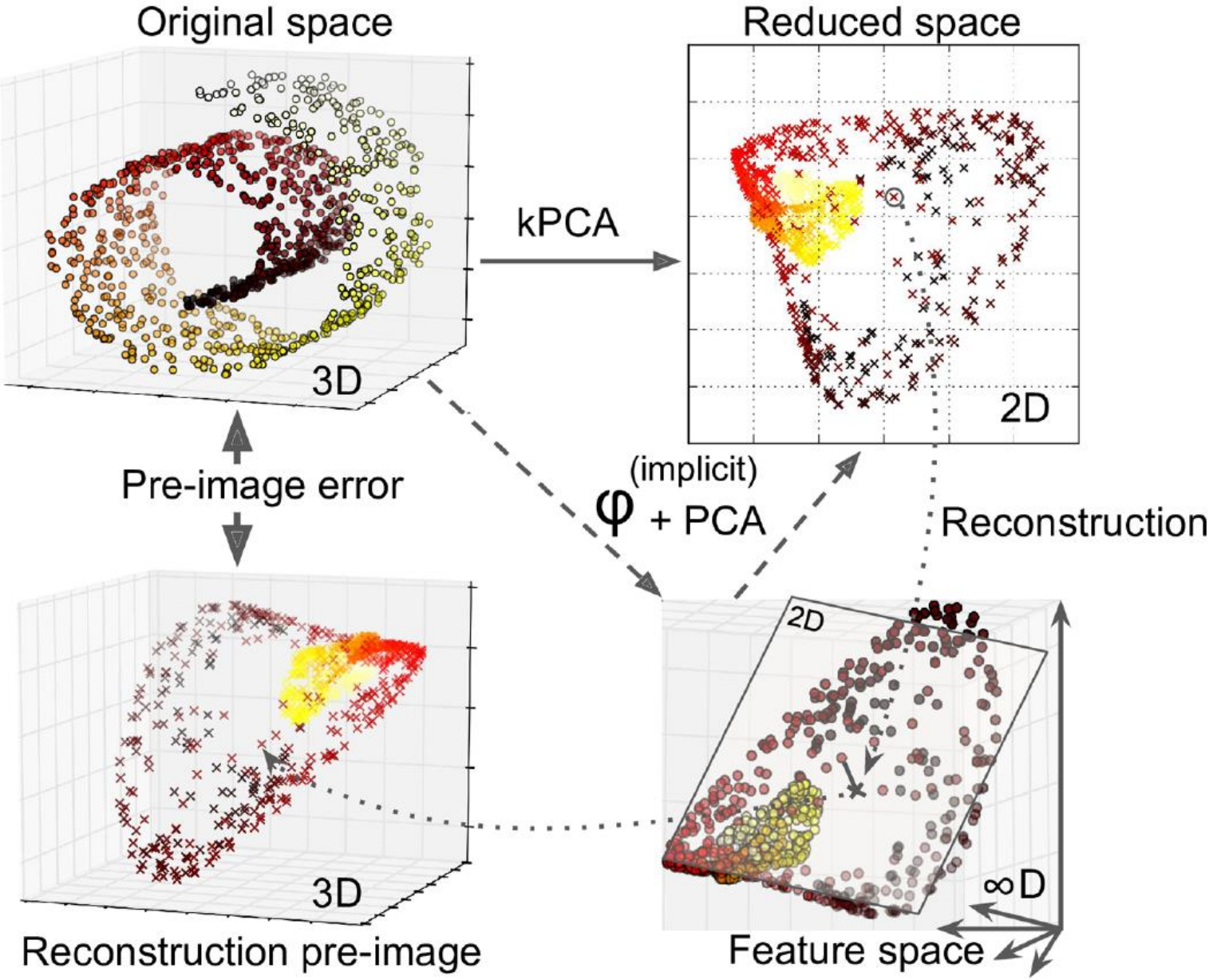
```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

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)
```

Selecting a Kernel and Tuning Hyperparameters

> The best kernel and hyperparameters are then available through the **`best_params_`** variable:

```
>>> print(grid_search.best_params_)
{'kpca__gamma': 0.04333333333333335, 'kpca__kernel': 'rbf'}
```

Selecting a Kernel and Tuning Hyperparameters



Selecting a Kernel and Tuning Hyperparameters

> 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)
```

> You can then compute the reconstruction pre-image error:

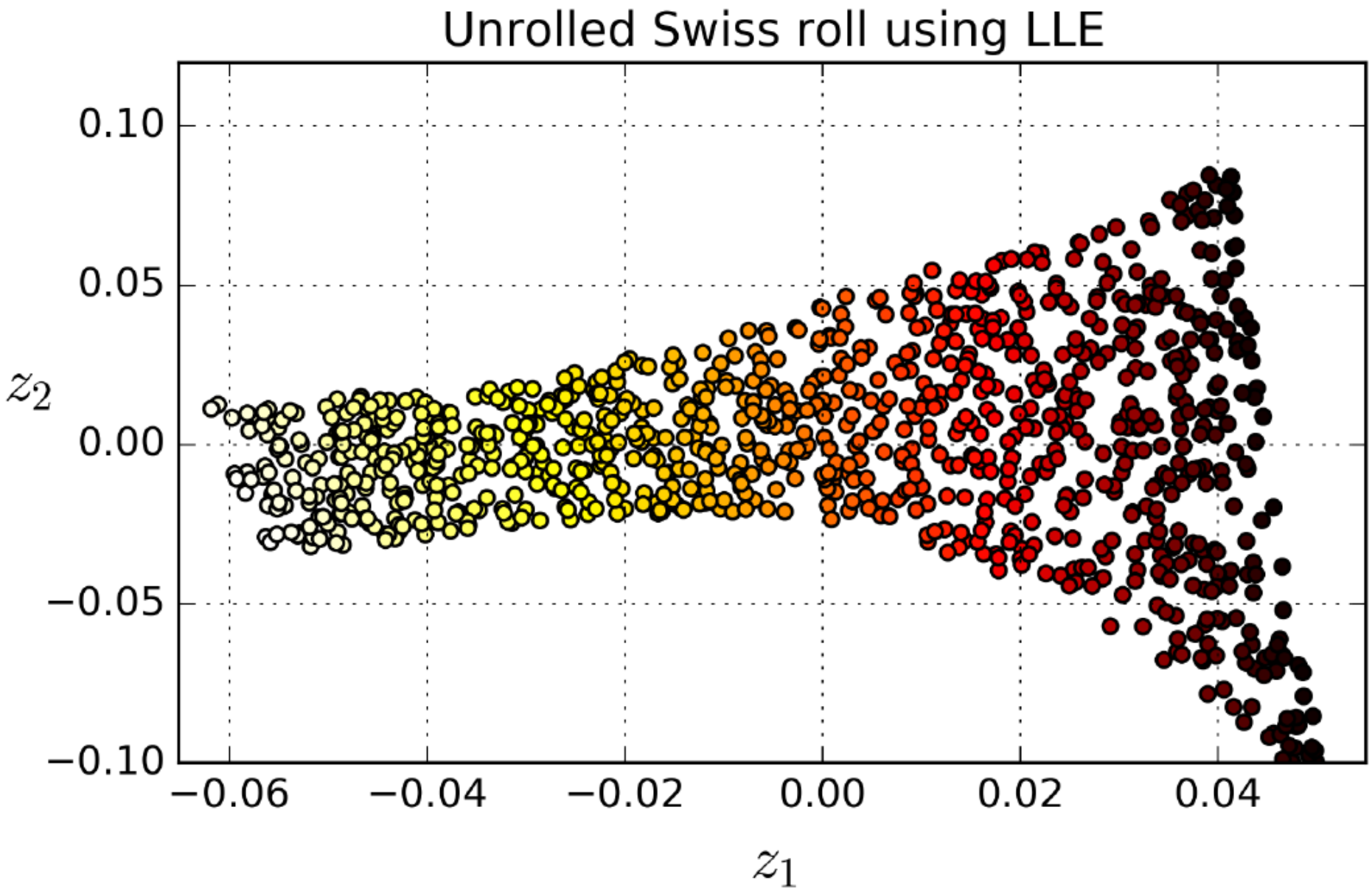
```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(X, X_preimage)
32.786308795766132
```

LLE (Locally Linear Embedding)

- LLE first measures how each training instance linearly relates to its closest neighbors (c.n.)
- Then looking for a low-dimensional representation of the training set where these local relationships are best preserved
- This makes it particularly good at unrolling twisted manifolds, especially when there is not too much noise

```
from sklearn.manifold import LocallyLinearEmbedding

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

LLE (Locally Linear Embedding)



LLE (Locally Linear Embedding)

> Linearly modeling local relationships

$$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^{m} \| \mathbf{x}^{(i)} - \sum_{j=1}^{m} w_{i,j} \mathbf{x}^{(j)} \|^2$$

$$\text{subject to} \begin{cases} w_{i,j} = 0 & \text{if } \mathbf{x}^{(j)} \text{ is not one of the } k \text{ c.n. of } \mathbf{x}^{(i)} \\ \sum_{j=1}^{m} w_{i,j} = 1 & \text{for } i = 1, 2, \cdots, m \end{cases}$$

> Reducing dimensionality while preserving relationships

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

LLE (Locally Linear Embedding)

- 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 low-dimensional representations.
- Unfortunately, the m^2 in the last term makes this algorithm scale poorly to very large datasets

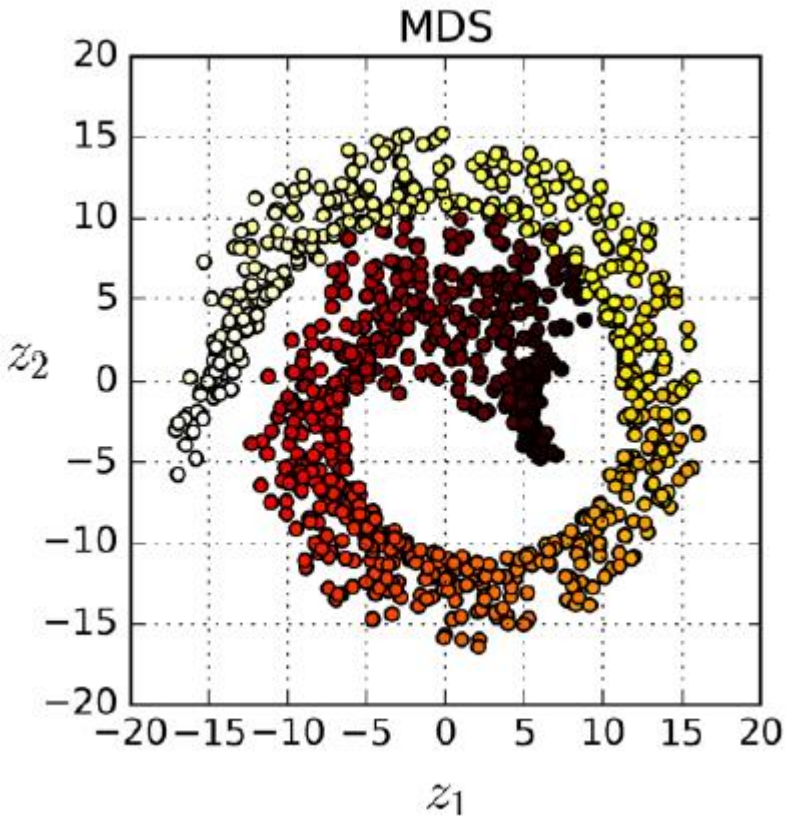
Other Dimensionality Reduction Techniques

- *Multidimensional Scaling* (MDS) reduces dimensionality while trying to preserve the distances between the instances
- *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.
- *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

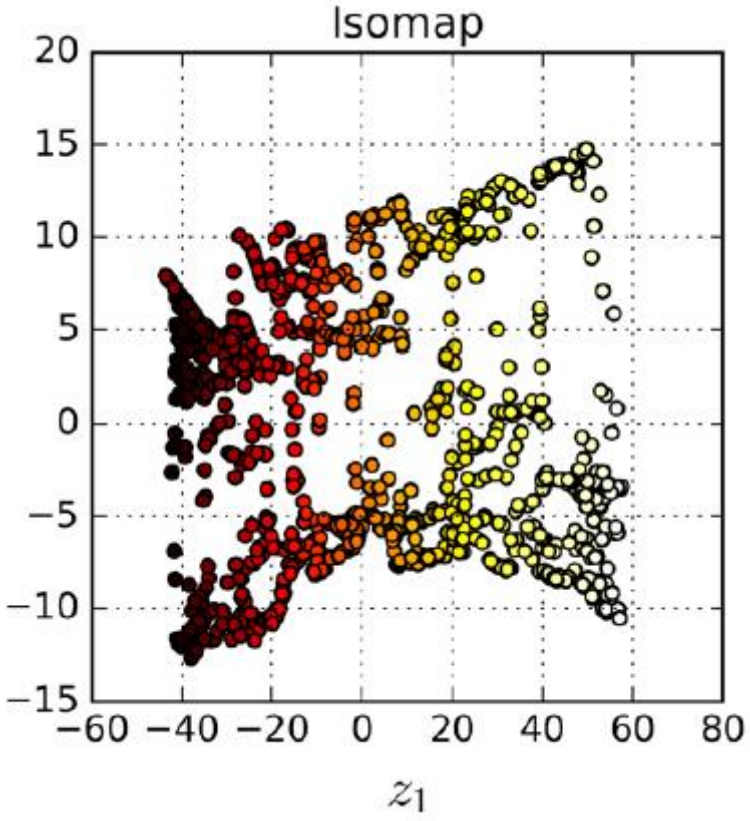
Other Dimensionality Reduction Techniques

- *Linear Discriminant Analysis* (LDA) is actually a classification algorithm
  - During training it learns the most discriminative axes between the classes, and these axes can then be used to define a hyperplane onto which to project the data.
  - The benefit is that the projection will keep classes as far apart as possible
  - LDA is a good technique to reduce dimensionality before running another classification algorithm such as an SVM classifier.

Other Dimensionality Reduction Techniques



Other Dimensionality Reduction Techniques



Other Dimensionality Reduction Techniques