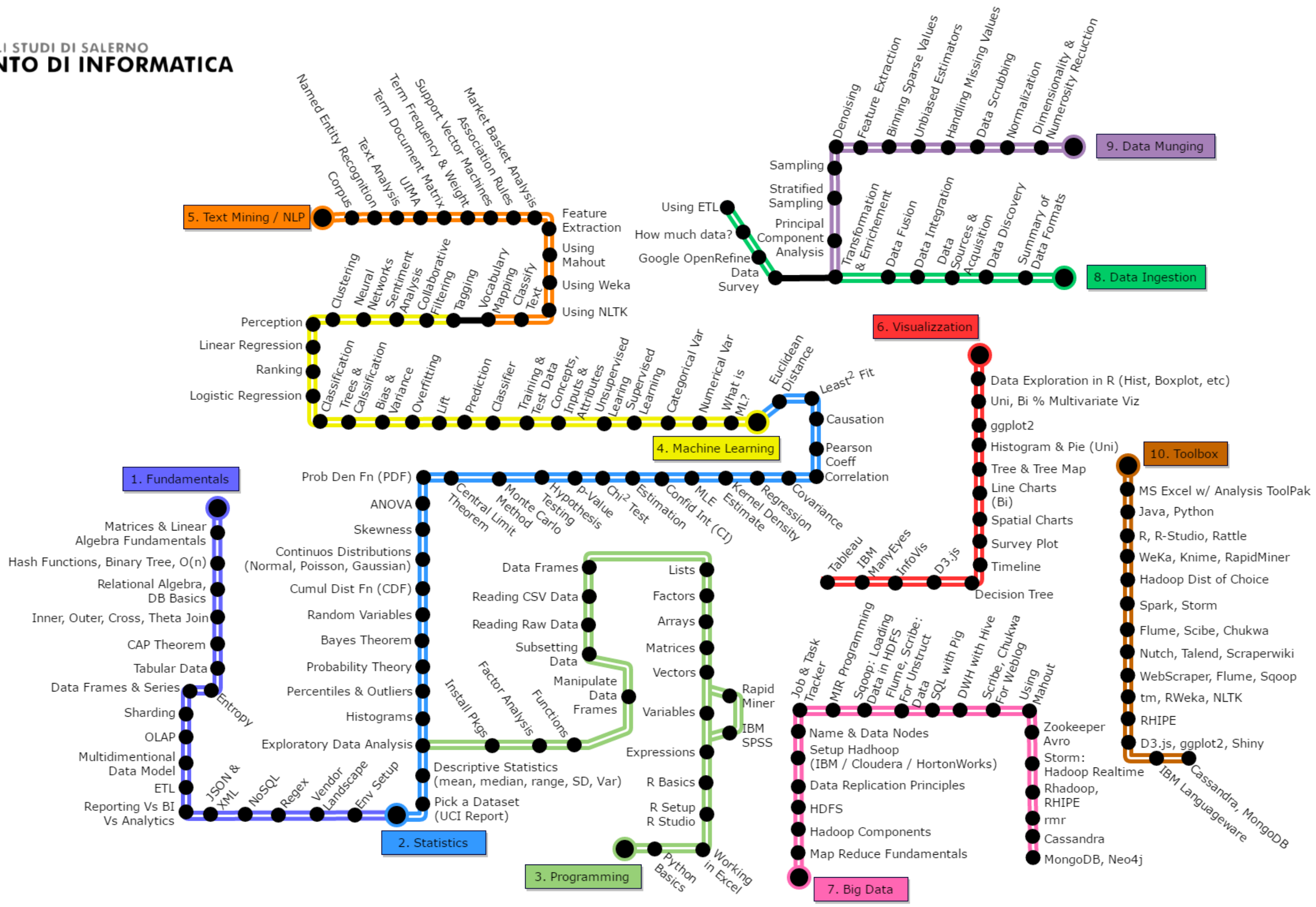
UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA



Fondamenti di Data Science e Machine Learning

Dimensionality Reduction

Aurelien Geron: «Hands on Machine Learning with Scikit Learn and TensorFlow, O'Reilly ed.

Prof. Giuseppe Polese, aa 2024-25

Outline

Introduction

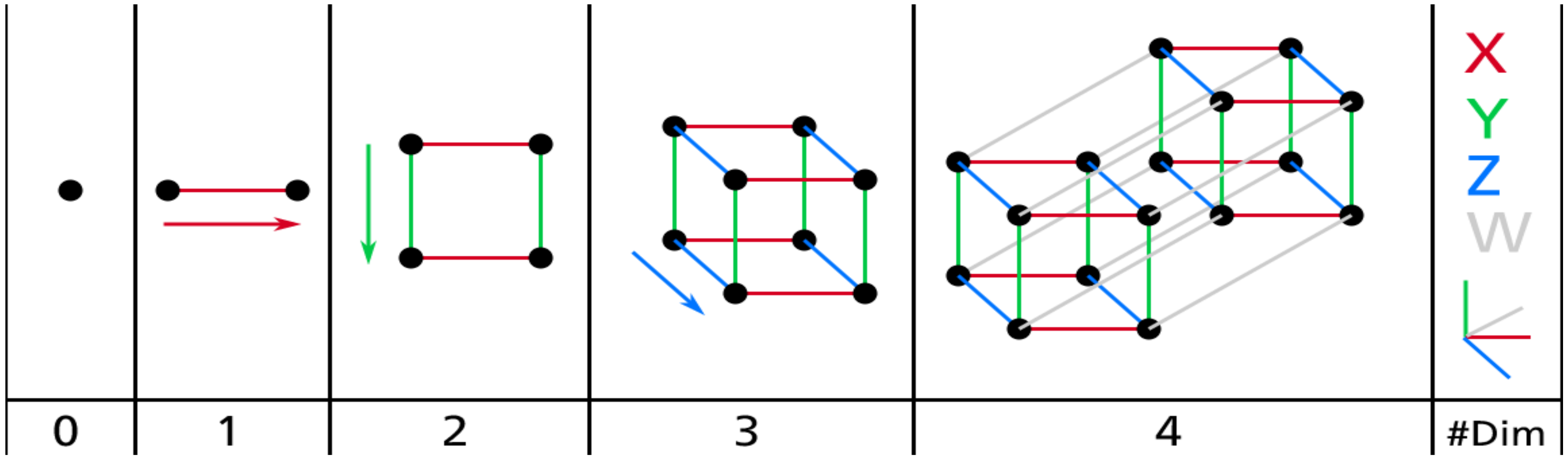
- Many Machine Learning problems involve thousands or even millions of features for each training instance
  - this make training extremely slow
  - it can also make it much harder to find a good solution
- This problem is often referred as curse of dimensionality
- In real-world problems
  - it is often possible to reduce the number of features considerably
    - Turning an intractable problem into a tractable one

Reducing Dimensionality

- Reducing dimensionality does lose some information
  - just like compressing an image to JPEG can degrade its quality
- Even if it will speed up training, it may also make our system perform slightly worse
- It also makes our pipelines a bit more complex and thus harder to maintain
  - we should first try to train our system on the original data before considering dimensionality reduction if training is slow
- Reducing the dimensionality of training data may filter out some noise and thus result in higher performances
  - but in general it won't; it will just speed up training

The Curve of Dimensionality (1)

- In general, our intuition fails us when we try to imagine a dimensional space higher than 3 dimensions
  - 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



Point, segment, square, cube, and tesseract (0D to 4D hypercubes)

The Curve of Dimensionality (2)

- It turns out that many things behave very differently in high-dimensional space
  - 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
    - it is very unlikely that a random point will be “extreme” along any dimension
- But 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%
  - Most points in a high-dimensional hypercube are very close to the border

The Curve of Dimensionality

- A more troublesome difference: if we randomly pick two points in a unit square, the distance between them will be, on average, roughly 0.52
- The average distance between two random points randomly picked in a unit 3D cube will be roughly 0.66.
- What about 2 points randomly picked in a 1,000,000-dimensional hypercube? Their average distance will be about 408.25 (rough $\sqrt{1,000,000/6}$)!)
- This is counterintuitive: how can 2 points be so far apart when both lie within the same unit hypercube?

The Curve of Dimensionality (3)

- High-dimensional datasets risk of being very sparse
  - most training instances are likely to be far away from each other
- This also means that a new instance will likely be far away from any training instance, 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

The Curve of Dimensionality (4)

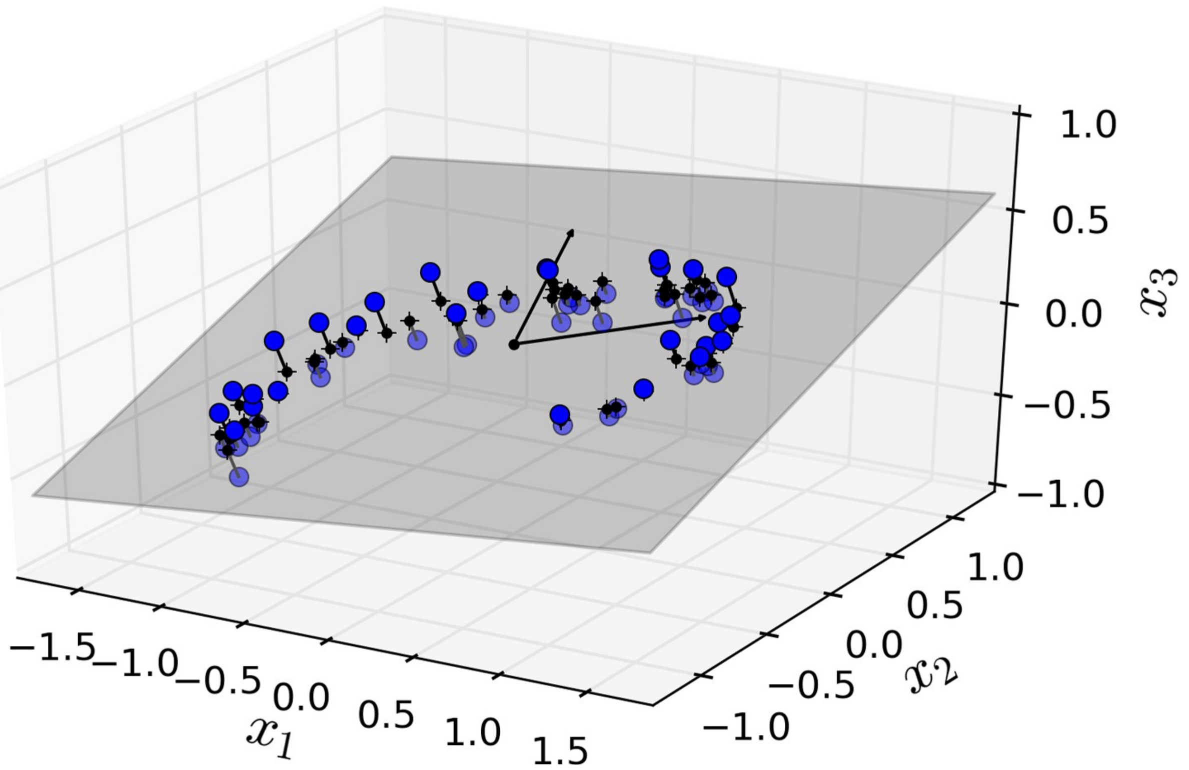
- In theory, one solution to the curse of dimensionality could be to increase the size of the training set to reach a sufficient density of training instances
- In practice, the number of training instances required to reach a given density grows exponentially with the number of dimensions
- Two main approaches to reducing dimensionality are
  - Projection
  - ManifoldLearning

Projection (1)

- 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
- Let’s look at an example
  - A 3D dataset represented by the circles (next slide)

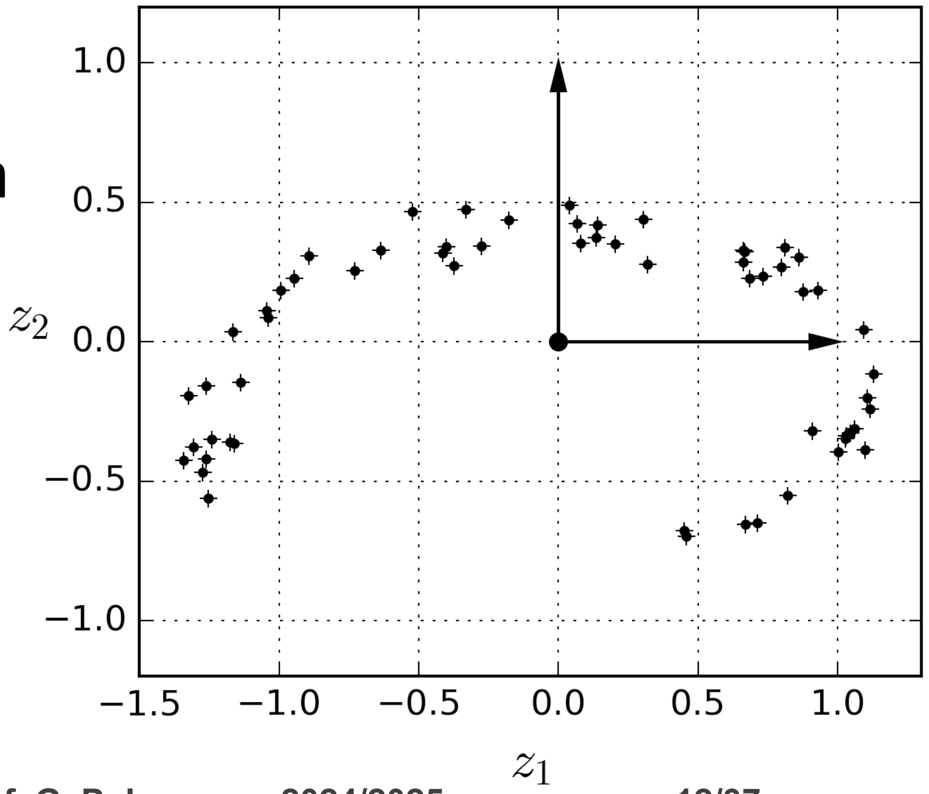
Projection (2)

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



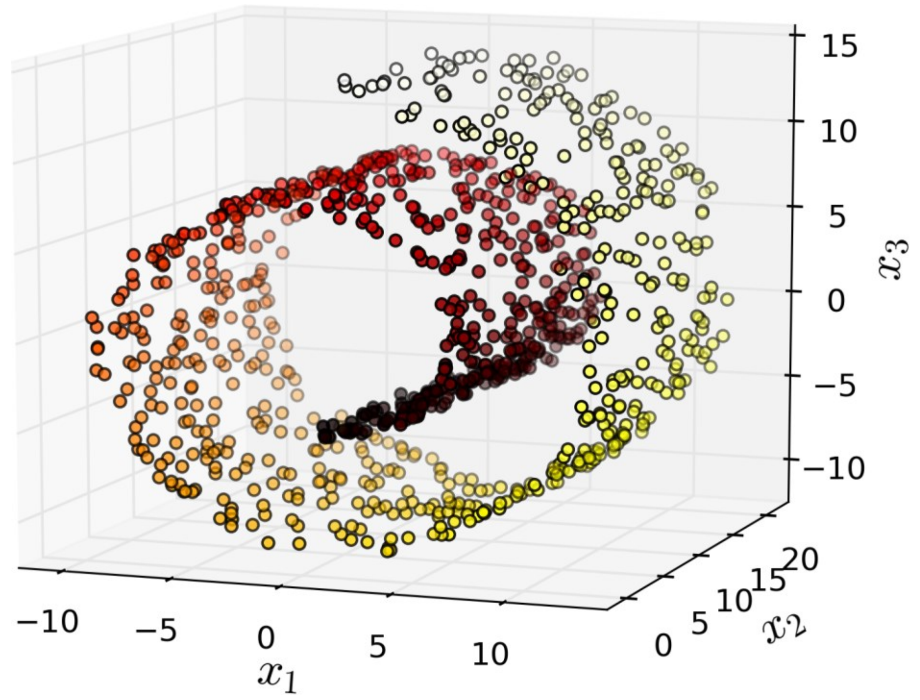
Projection (3)

- If we project every training instance perpendicularly onto this subspace
  - represented by the short lines connecting the instances to the plane
- we get the new 2D data
  - Note that the axes correspond to new features z_1 and z_2 (the coordinates of the projections on the plane)



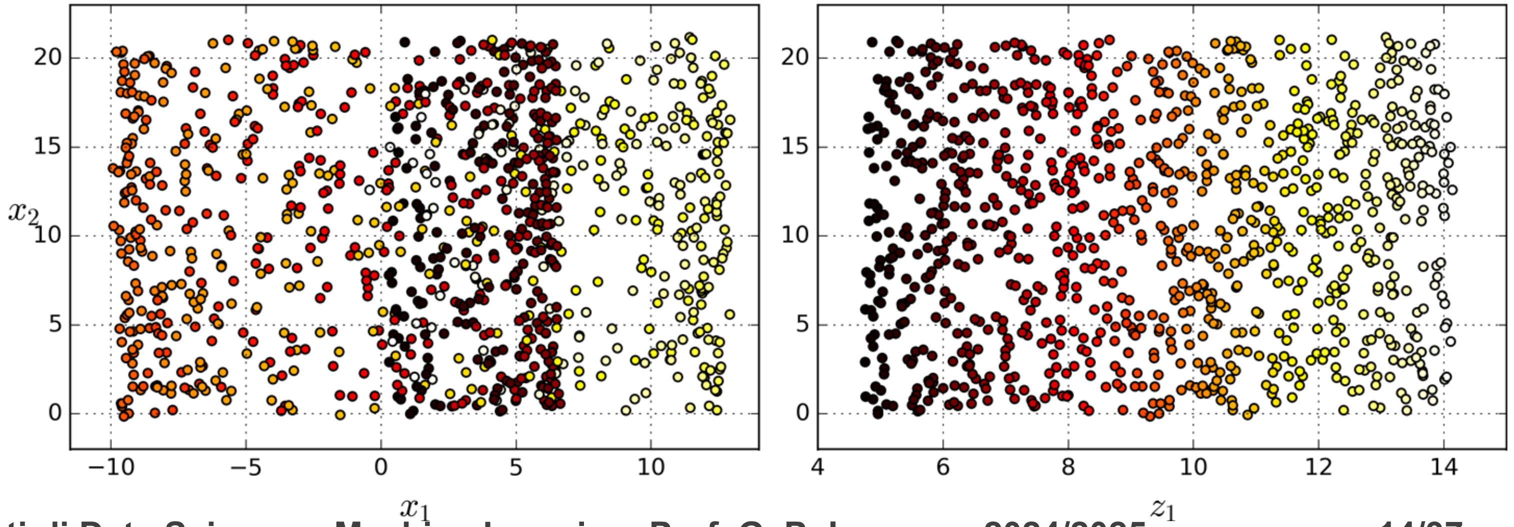
Projection: Another Case (1)

- 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



Projection: Another Case (2)

- Simply projecting onto a plane (e.g., by dropping *x*3) would squash different layers of the Swiss roll together,
  - As shown on the left of Figure
- However, what you really want is to unroll the Swiss roll to obtain the 2D dataset
  - As shown on the right of Figure



Manifold Learning (1)

- 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

Manifold Learning (2)

- 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 (3)

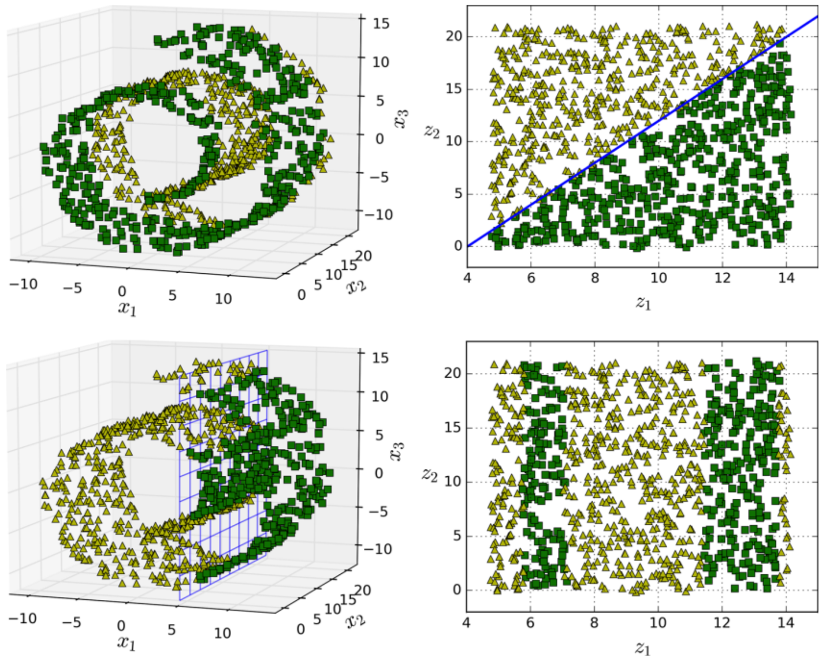
- Think about the MNIST dataset
  - All handwritten digit images have some similarities
- They are made of connected lines, the borders are white, they are more or less centered, and so on
- If we randomly generated images, only a tiny fraction of them would look like handwritten digits
  - The degrees of freedom available to us if we try to create a digit image are dramatically lower than the degrees of freedom we would have if we were allowed to generate any image we wanted
- These constraints tend to squeeze the dataset into a lower-dimensional manifold

Manifold Learning (4)

- Another implicit assumption
  - The task at hand (e.g., classification or regression) will be simpler if expressed in the lower-dimensional space of the manifold
- In the top row of Figure (next slide) the Swiss roll is split into two classes
  - In the 3D space (on the left), the decision boundary would be fairly complex,
  - In the 2D unrolled manifold space (on the right), the decision boundary is a simple straight line

Manifold Learning (5)

- The previous assumption does not always hold
  - In the bottom row of Figure the decision boundary is located at $x_1 = 5$
- This decision boundary looks very simple in the original 3D space (a vertical plane), but it looks more complex in the unrolled manifold (a collection of four independent line segments)

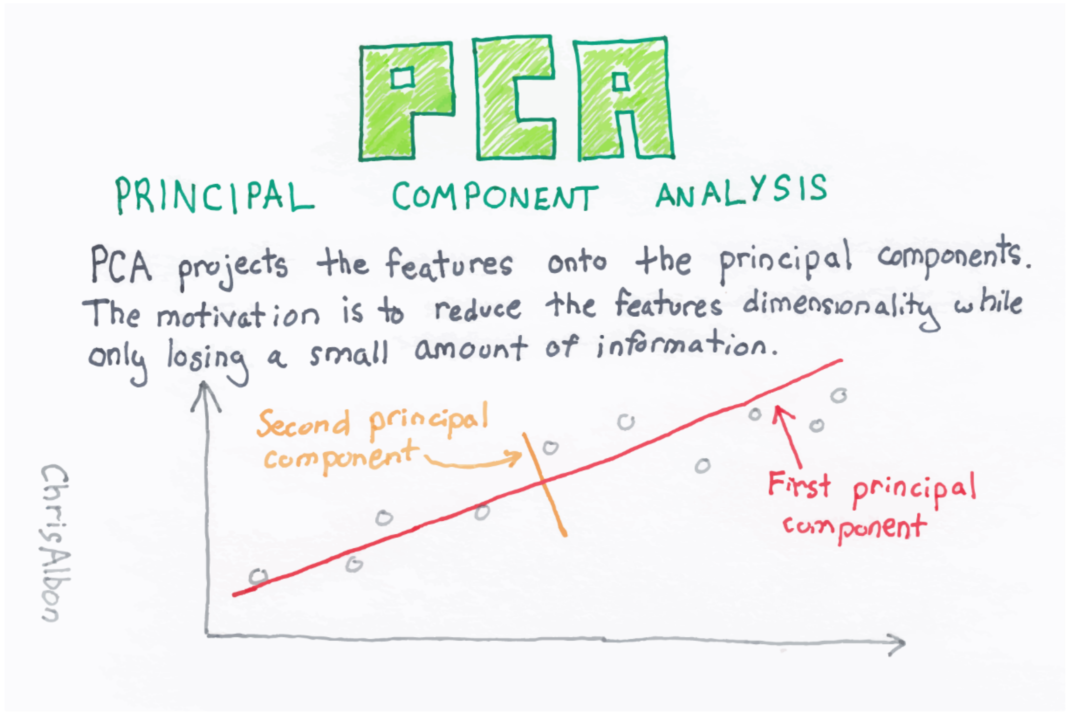


Summarizing Manifold Learning

- If we reduce the dimensionality of our training set before training a model
  - It will definitely speed up training
  - It may not always lead to a better or simpler solution
  - It all depends on the dataset
- We have a good sense of what the curse of dimensionality is and how dimensionality reduction algorithms can fight it, especially when the manifold assumption holds

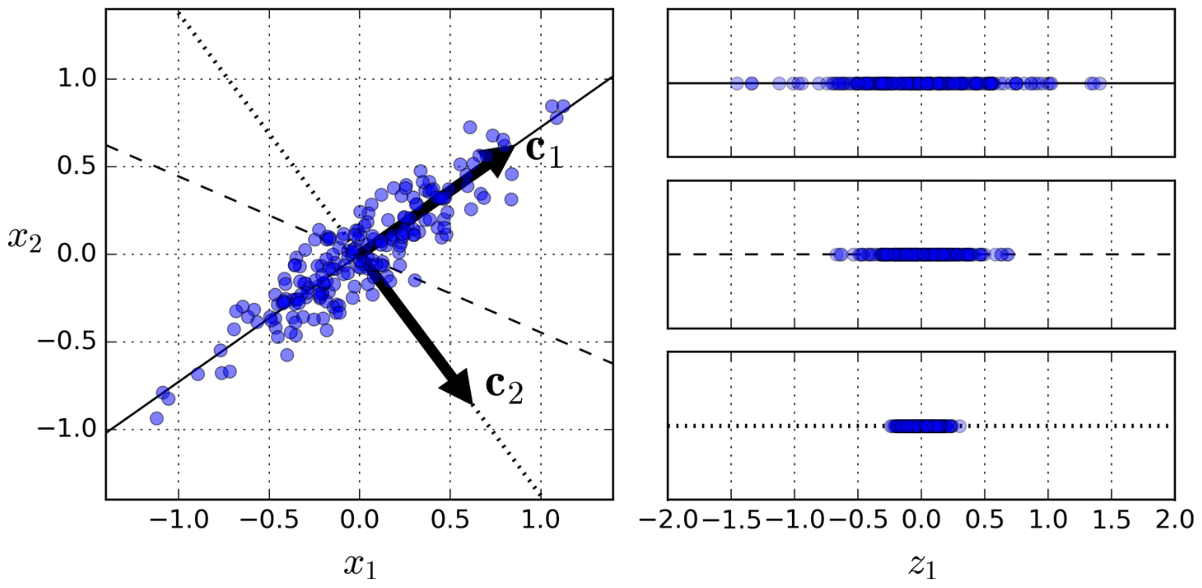
PCA

- Principal Component Analysis (PCA) is by far the most popular dimensionality reduction algorithm
  - First it identifies the hyperplane that lies closest to the data
  - Then it projects the data onto it

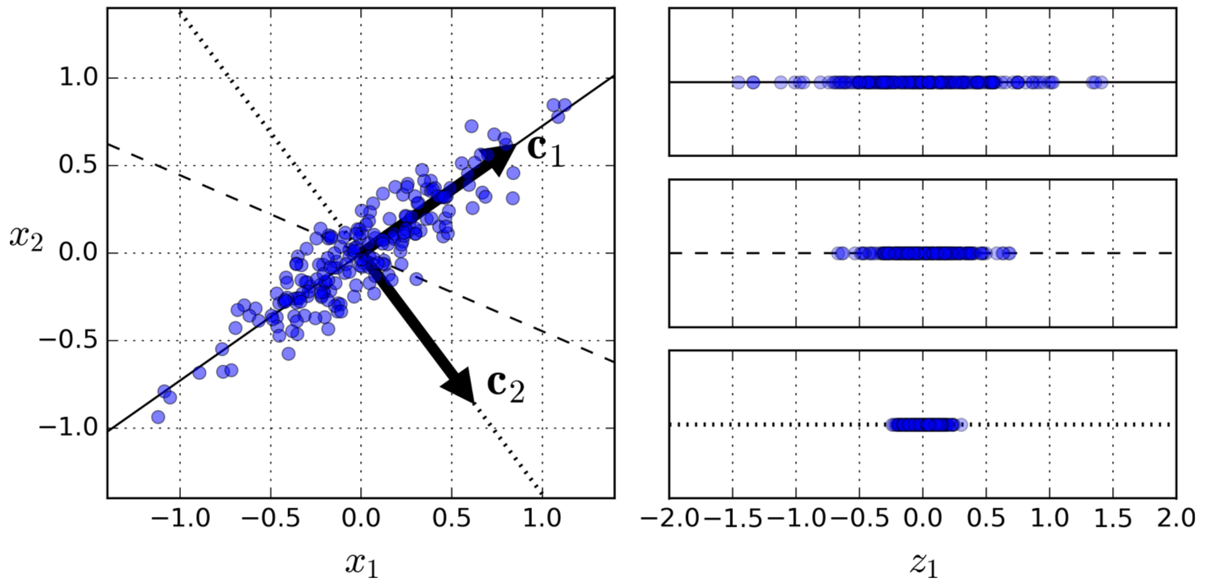

PCA
PRINCIPAL COMPONENT ANALYSIS
PCA projects the features onto the principal components. The motivation is to reduce the features dimensionality while only losing a small amount of information.
Second principal component
First principal component
ChrisAlbon

Preserving the Variance (1)

- Before projecting 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 of Figure, along with three different axes (i.e., one-dimensional hyperplanes)
  - The result of the projection onto each of these axes (on the right)

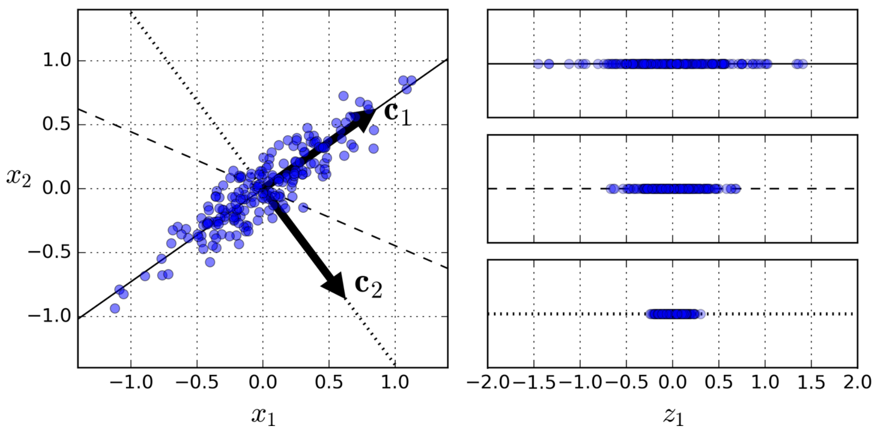


Preserving the Variance (2)



- As you can see from the figure on the right
  - the projection onto the solid line preserves the maximum variance
  - the projection onto the dotted line preserves very little variance,
  - the projection onto the dashed line preserves an intermediate amount of variance

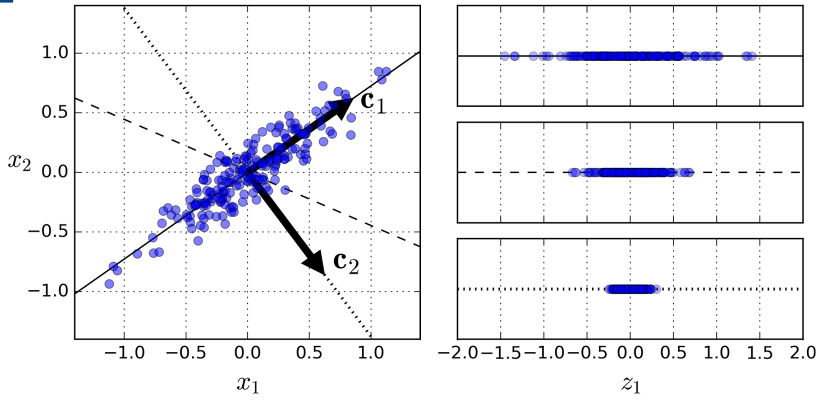
Preserving the Variance (3)



- We will first project on the axis preserving the max variance
- This is the one minimizing the mean square distance between the original dataset and its projection on the selected axis
- Then, we project on the axis preserving the max residual variance, which is not the dashed line, since some of its variance was in $\mathbf{c}_1$
- **This is the rather simple idea behind PCA**

Principal Components (1)

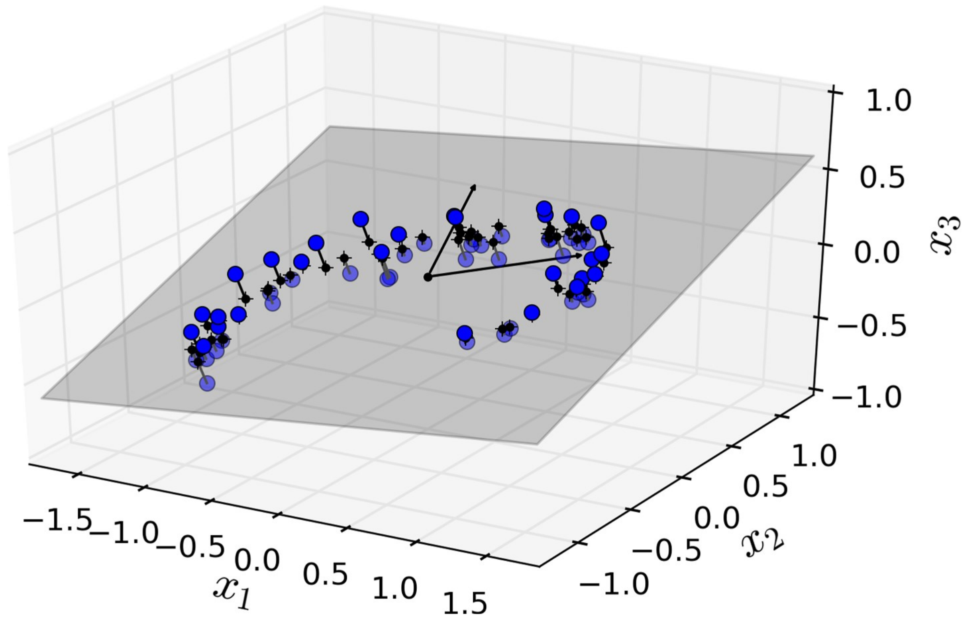
- PCA identifies the axis that accounts for the largest amount of variance in the training set (the solid line $\mathbf{c_1}$)
- It also finds a second axis, orthogonal to the first one, that accounts for the largest amount of remaining variance (the dotted line $\mathbf{c_2}$)



- 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

Principal Components (2)

- The unit vector that defines the ith axis is called the ith principal component (PC)
  - In the previous Figure, the 1st PC is $\mathbf{c_1}$ and the 2nd PC is $\mathbf{c_2}$
  - In the following Figure 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)



Principal Components (3)

- The direction of the principal components is not stable
  - if you perturb the training set slightly and run PCA again, some of the new PCs may point in the opposite direction of the original PCs
- However, they will generally still lie on the same axes
- In some cases, a pair of PCs may even rotate or swap, but the plane they define will generally remain the same

Principal Component Matrix

How can you find the principal components of a training set?

- There is a standard matrix factorization technique called Singular Value Decomposition (SVD)
  - It can decompose the training set matrix **X** into the dot product of three matrices **U** · Σ · **V**^{T}, where **V** contains all the principal components that we are looking for

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

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
  - You can simply compute the dot product of the training set matrix **X** by the matrix $\mathbf{W}_d$
    - defined as the matrix containing the first *d* principal components

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

Using Scikit-Learn for PCA (1)

- 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

```python
from sklearn.decomposition import PCA
import numpy as np
#Build 3D dataset:
np.random.seed(4)
m = 60
w1, w2 = 0.1, 0.3
noise = 0.1
angles = np.random.rand(m) * 3 * np.pi / 2 - 0.5
X = np.empty((m, 3))
X[:, 0] = np.cos(angles) + np.sin(angles)/2 + noise * np.random.randn(m) / 2
X[:, 1] = np.sin(angles) * 0.7 + noise * np.random.randn(m) / 2
X[:, 2] = X[:, 0] * w1 + X[:, 1] * w2 + noise * np.random.randn(m)
#PCA using Scikit-Learn
pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
print("X2D[:5]:\n{}".format(X2D[:5]))
```

```
X2D[:5]:
[[ 1.26203346  0.42067648]
 [-0.08001485 -0.35272239]
 [ 1.17545763  0.36085729]
 [ 0.89305601 -0.30862856]
 [ 0.73016287 -0.25404049]]

Process finished with exit code 0
```

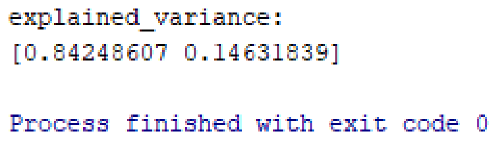
Using Scikit-Learn for PCA (2)

- 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
  - for example, 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
- For example

```
print("explained_variance:\n{}".format
    (pca.explained_variance_ratio_))
```



```
explained_variance:
[0.84248607 0.14631839]

Process finished with exit code 0
```

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

The Right Number of Dimensions (1)

- 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 you are reducing dimensionality for data visualization
    - In that case you will generally want to reduce the dimensionality down to 2 or 3
- 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

The Right Number of Dimensions (2)

```python
from sklearn.decomposition import PCA
import numpy as np
from six.moves import urllib
#Use of MNIST dataset
try:
    from sklearn.datasets import fetch_openml
    mnist = fetch_openml('mnist_784', version=1)
    mnist.target = mnist.target.astype(np.int64)
except ImportError:
    from sklearn.datasets import fetch_mldata
    mnist = fetch_mldata('MNIST original')
from sklearn.model_selection import train_test_split
X = mnist["data"]
y = mnist["target"]
X_train, X_test, y_train, y_test = train_test_split(X, y)
#PCA using Scikit-Learn
pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
print("d:{}".format(d))
```

```
d:154

Process finished with exit code 0
```

The Right Number of Dimensions (3)

- You could then set `n_components=d` and run PCA again
  - Instead of specifying the number of principal, you can set `n_components` to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve

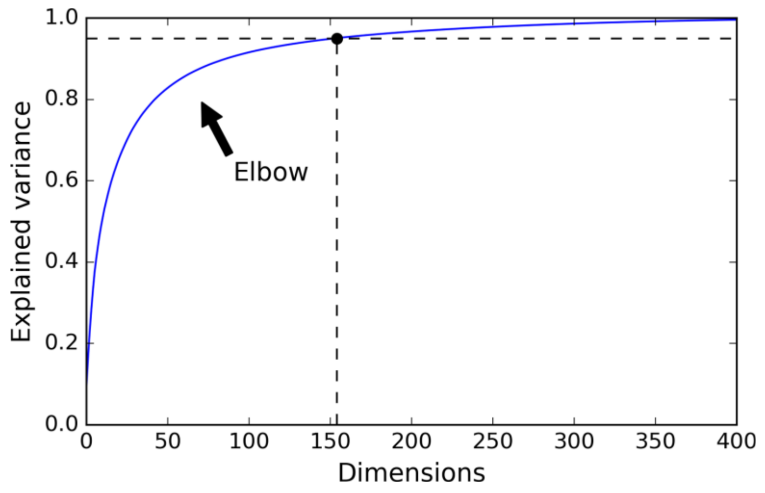
```
from sklearn.decomposition import PCA
import numpy as np
from six.moves import urllib
#Use of MNIST dataset
try:
    from sklearn.datasets import fetch_openml
    mnist = fetch_openml('mnist_784', version=1)
    mnist.target = mnist.target.astype(np.int64)
except ImportError:
    from sklearn.datasets import fetch_mldata
    mnist = fetch_mldata('MNIST original')
from sklearn.model_selection import train_test_split
X = mnist["data"]
y = mnist["target"]
X_train, X_test, y_train, y_test = train_test_split(X, y)
#PCA using Scikit-Learn
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
print("pca.n_components_:{}".format(pca.n_components_))
```

```
pca.n_components_:154

Process finished with exit code 0
```

The Right Number of Dimensions (4)

- Yet another option is to plot the explained variance as a function of the number of dimensions
  - simply plot `cumsum`
  - There will usually be an elbow in the curve, where the explained variance stops growing fast



  - You can think of this as the intrinsic dimensionality of the dataset
    - In this case, you can see that reducing the dimensionality down to about 100 dimensions wouldn’t lose too much explained variance