BTS | Barcelona Technology School

Classical Data Analysis

Master in Big Data Solutions 2020-2021

Filipa Peleja

Víctor Pajuelo

filipa.peleja@bts.tech

victor.pajuelo@bts.tech

Today's class

Contents

- Dimensionality reduction
  - PCA
  - PCA for compression
  - PCA for dim reduction
  - PCA for feature selection
  - Limits of PCA

Today's objective

- Learn how and when to use PCA
- Understand the needs of Dimensionality Reduction and the curse of dimensionality
- Learn the goods and limits of projection as dimensionality reduction

Let's git things done!

Let's see it again

Pull Session 11 notebooks

```
$ git clone https://github.com/vfp1/bts-cda-2020.git

# If you have done that already
$ git pull origin master
```

BTS | Barcelona Technology School

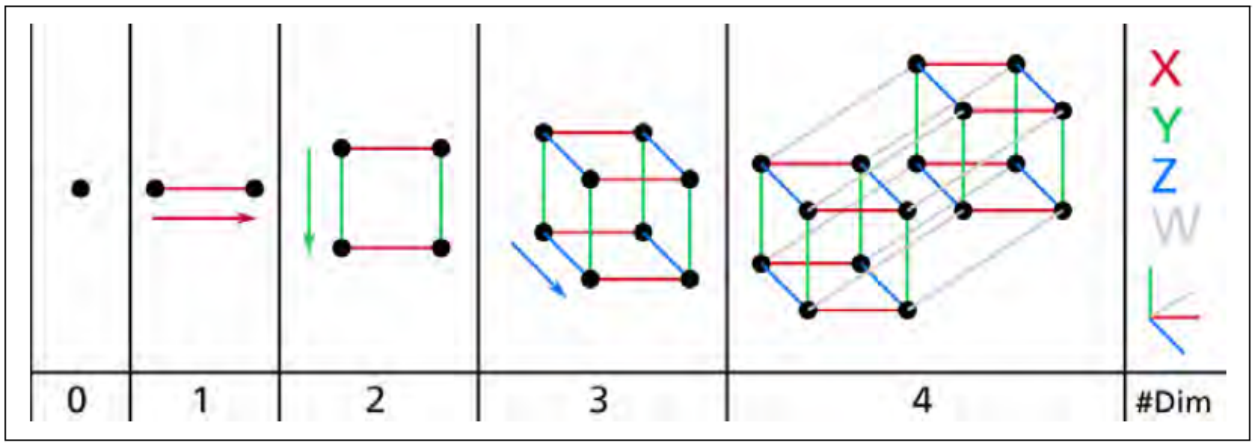
MBDS 2020-2021

Dimensionality Reduction

Dimensionality Reduction

Big Data dimensions

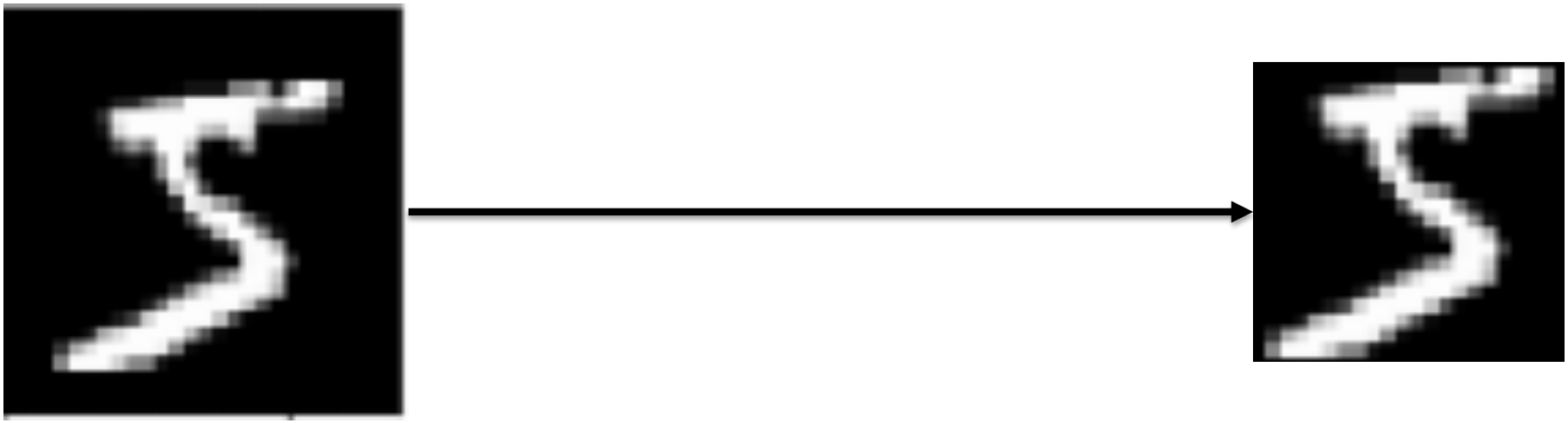
- Big Data is hungry for training samples.
- Those datasets involve thousands or even millions of features for each training instance.
- While Big Data can disclose relations in high dimensional data, it also has two main problems:
  - SNR: Signal to Noise ratio
  - Computation: bigger datasets need more power to compute
- This is also known as the ***curse of dimensionality***



Dimensionality Reduction

Making intractable problems tractable

- Sometimes noise beats signal and there is a lot of noise in the dataset.
- We can drop data (dimensions) from our dataset without losing a lot of information:
  - The surrounding pixels
  - The neighboring pixels that can be merged into a single pixel



Dimensionality Reduction

Why we want to loose information?

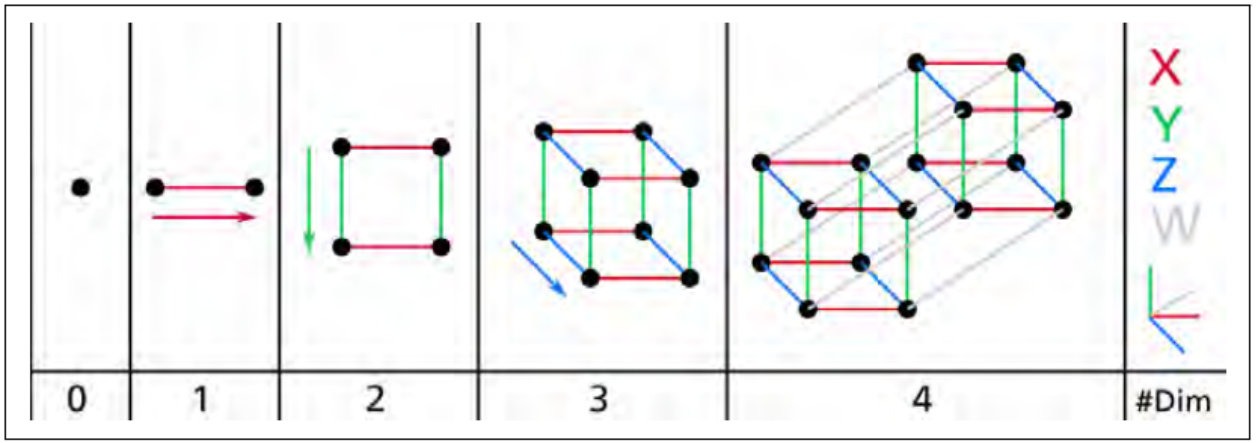
- Reducing dimensionality involves also **losing certain information** (think into a JPEG compression, that reduces size and degrades the quality)
- It will certainly **speed up training** but can make the model **perform a bit worse**
- Reducing dimensionality is very useful for data visualization (going from 10d to 2d for instance)

- Before performing dimensionality reduction try to train your problem with the original data. Dimensionality reduction usually speeds up training, sometimes it can filter unnecessary noise, but normally it just **speeds up training**

Dimensionality Reduction

High dimensional spaces

- Due to its nature, high dimensional datasets have the **risk of being very sparse**
  - Training instances are likely to be far away from each other
- New instances are likely to be **very far away** from any training instance, making **predictions less reliable** since they are based on **very large extrapolations** over the dataset
- Generally, the **more dimensions, the greater risk for overfitting**



Dimensionality Reduction

Approaches to Dimensionality Reduction

- In order to address those issues, one could increase the size of the training set in order to reach a sufficient density of training instances
- However, the number of training instances required to reach a given density **grows exponentially with the number of dimensions**
- There is a better way….
  - Projecting the dataset in the hyperplane where the variance is highly maintained

BTS | Barcelona Technology School

MBDS 2020-2021

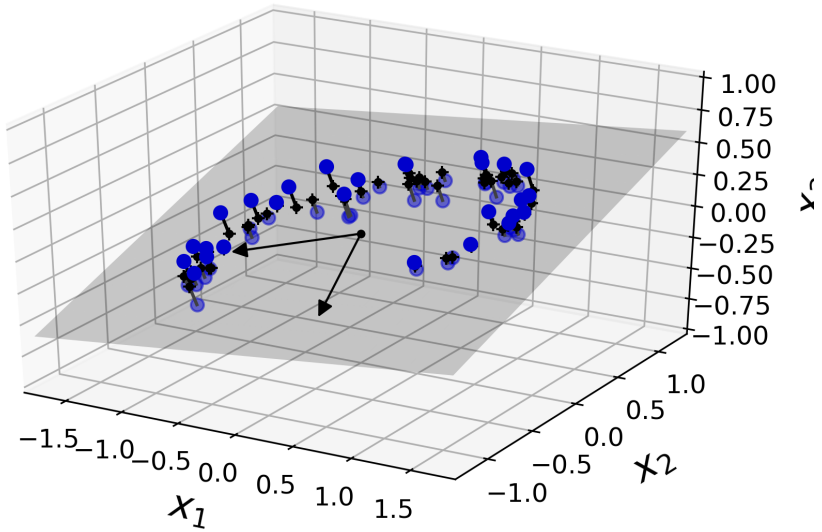
Dimensionality Reduction

Projection

Dimensionality Reduction

Projection

- In most of 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.

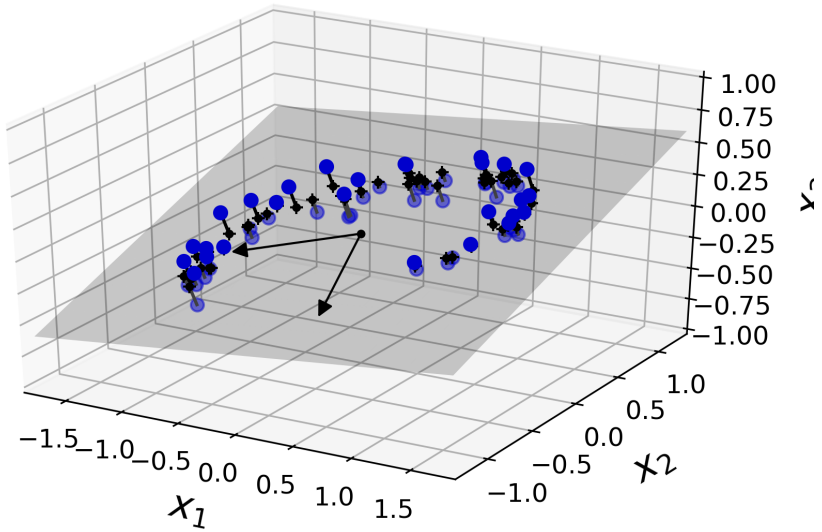


*Code to reproduce the graphs can be seen at **UUID - #S8C1***

Dimensionality Reduction

Projection

- All the training instances below lie close to a plane
- This plane is the lower-dimensional (2D) subspace of the high-dimensional (3D) space.
- In order to reduce dimensionality, we could **project** the instances perpendicularly onto the plane, getting a 2D dataset.

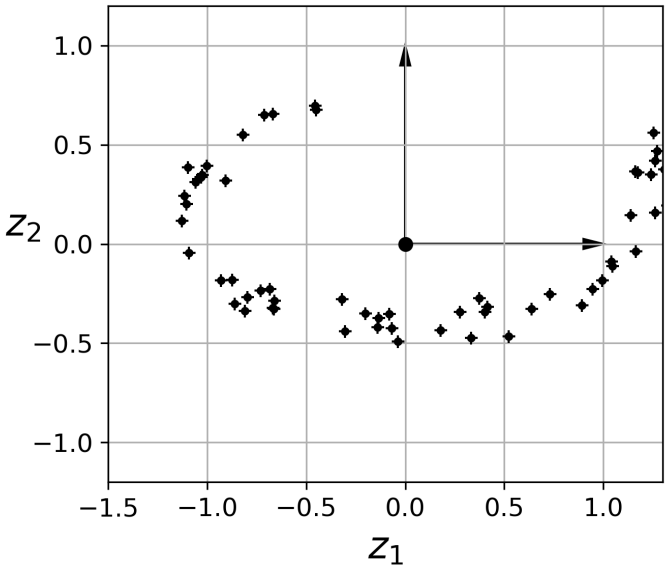


*Code to reproduce the graphs can be seen at **UUID - #S8C1***

Dimensionality Reduction

Projection

- Through the perpendicular projection, we have reduced the dataset's dimensionality from 3D to 2D
- We have two new features *z1* and *z2,* which are the two coordinates of the projected plane
- Projection is an easy approach to dimensionality reduction (in which PCA is based on)



*Code to reproduce the graphs can be seen at **UUID - #S8C1***

BTS | Barcelona Technology School

MBDS 2020-2021

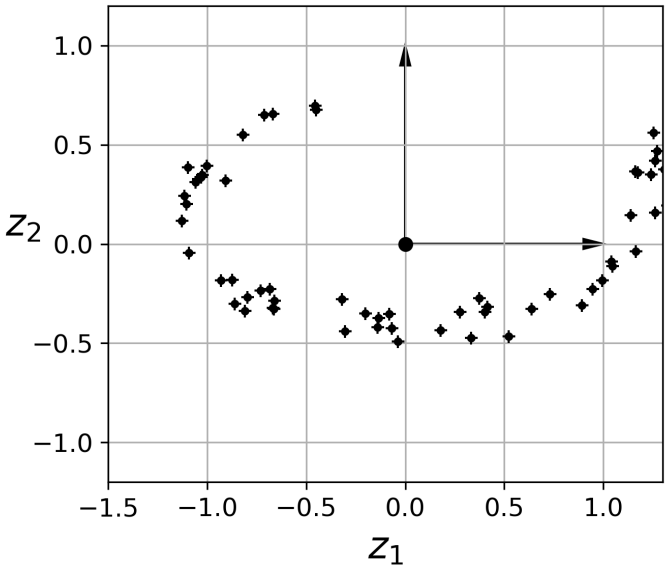
Principal Component Analysis

PCA

Principal Component Analysis

Introduction

- PCA is **one of the most popular** dimensionality reduction algorithms
- The task of PCA is fairly simple
  - First identifies the hyperplane that lies closest to the data
  - Then projects the data onto it
- You are then left with the components that “explain” the variance of the dataset, i.e., its signal

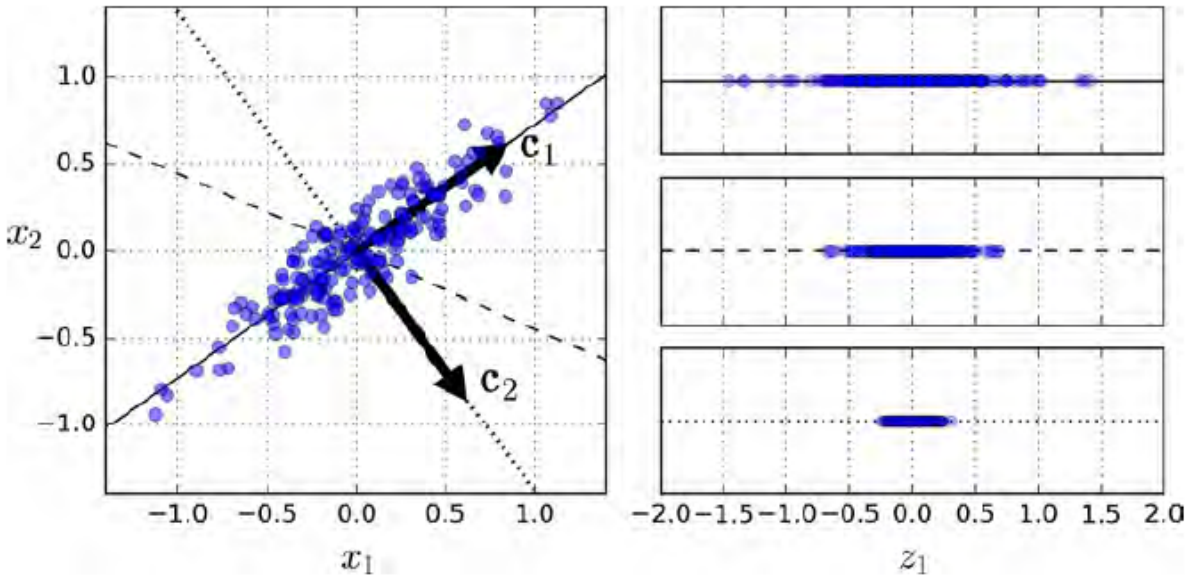


*Code to reproduce the graphs can be seen at **UUID - #S8C1***

Principal Component Analysis

Keeping the Variance

- The right hyperplanes need to be chosen before we project the training set onto a lower-dimensional hyperplane
- Below you have a 2D dataset represented along three different 1D hyperplanes
- The projection over the solid line preserves the maximum variance

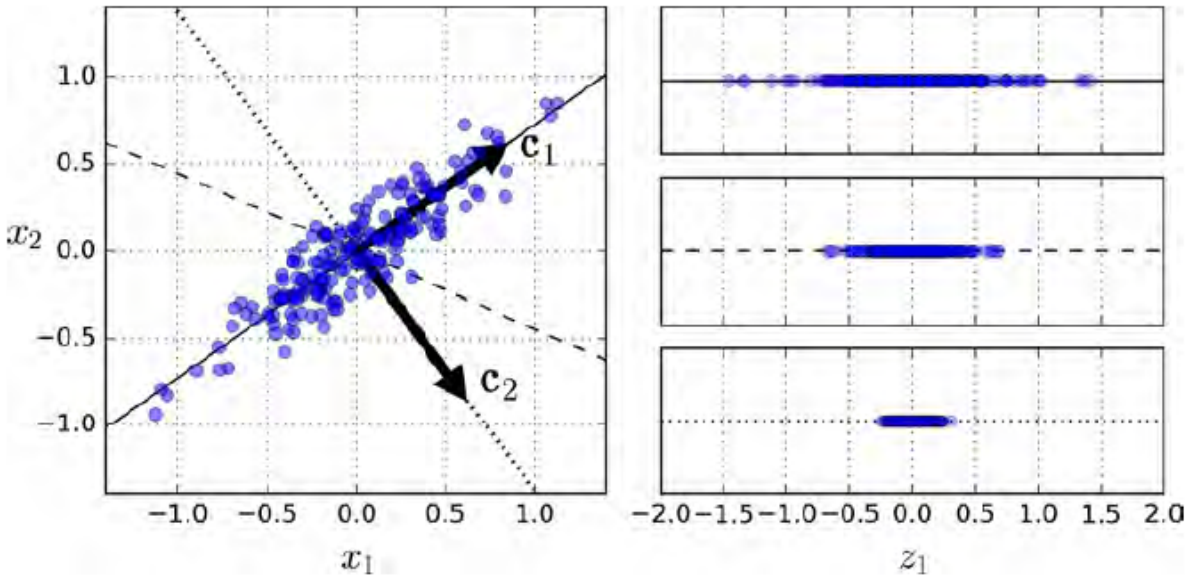


*Code to reproduce the graphs can be seen at **UUID - #S8C2***

Principal Component Analysis

Keeping the Variance

- We will need to choose the hyperplane **that preserves the maximum amount of variance**, in order to lose the less amount of information
- In a way, PCA chooses the hyperplane that **minimizes the mean squared distance** between the original dataset and its projection onto that hyperplane



*Code to reproduce the graphs can be seen at **UUID - #S8C2***

Principal Component Analysis

Principal Components

- PCA identifies the **first axis** that 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 remaining variance.
  - In our previous example, there is no choice: it is the dotted line.
  - If it were a higher-dimensional data set, 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 COMPONENT**: the unit vector that defines the i^{th} hyperplane
  - In the case above, we have two (c1 and c2), which are orthogonal arrows over the selected hyperplane
  - The third one will be orthogonal to the plane

Principal Component Analysis

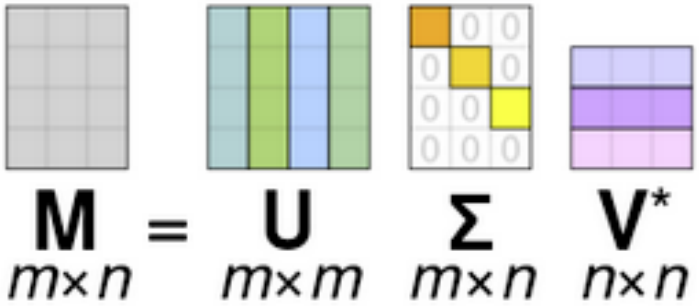
Principal Components

DO NOT GET SCARED! The direction of principal components is not stable, they usually switch between PCA runs, but the hyperplane tends to remain stable

Principal Component Analysis

Singular Value Decomposition (SVD)

- The **principal components** of a training set can be calculated using SVD
  - SVD is a matrix factorization technique
  - It decomposes the training set matrix **X** into the dot product of three matrices $\mathbf{U} \cdot \Sigma \cdot \mathbf{V}^T$, where $\mathbf{V}^T$ contains all the **principal components that we are looking for**



https://en.wikipedia.org/wiki/Singular_value_decomposition

Principal Component Analysis

Singular Value Decomposition (SVD)

- The **principal components** of a training set can be calculated using SVD
  - SVD is a matrix factorization technique
  - It decomposes the training set matrix **X** into the dot product of three matrices $\mathbf{U} \cdot \Sigma \cdot \mathbf{V}^T$, where $\mathbf{V}^T$ contains all the **principal components that we are looking for**

Numpy implementation for SVD [numpy.linalg.svd]

```
1 import numpy as np
2
3 X_centered = X - X.mean(axis=0)
4 U, s, Vt = np.linalg.svd(X_centered)
5 c1 = Vt.T[:, 0]
6 c2 = Vt.T[:, 1]
```

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

*Code to reproduce the graphs can be seen at **UUID - #S8C3***

Principal Component Analysis

Singular Value Decomposition (SVD)

PCA needs the dataset to be centered around the origin.

Sklearn does it automatically for us,
but if you use Numpy **remember to center the dataset first!!**
(By subtracting the mean for instance as done in the previous slide)

Principal Component Analysis

Projecting 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 the hyperplane defined by the principal components allows to keep the maximum variance

Principal Component Analysis

Projecting to D dimensions

- To project the training set onto the hyperplane, you can simply compute the dot product of the training set matrix **X** by the matrix $\mathbf{W}_d$, defined as the matrix containing the *d* principal components (i.e., the matrix composed of the first *d* columns of $\mathbf{V}^T$)

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

```
1 import numpy as np
2
3 X_centered = X - X.mean(axis=0)
4 U, s, Vt = np.linalg.svd(X_centered)
5 c1 = Vt.T[:, 0]
6 c2 = Vt.T[:, 1]
```

```
1 W2 = Vt.T[:, :2]
2 print(W2)
3 X2D = X_centered.dot(W2)
```

```
[[-0.79644131 -0.60471583]
 [-0.60471583  0.79644131]]
```

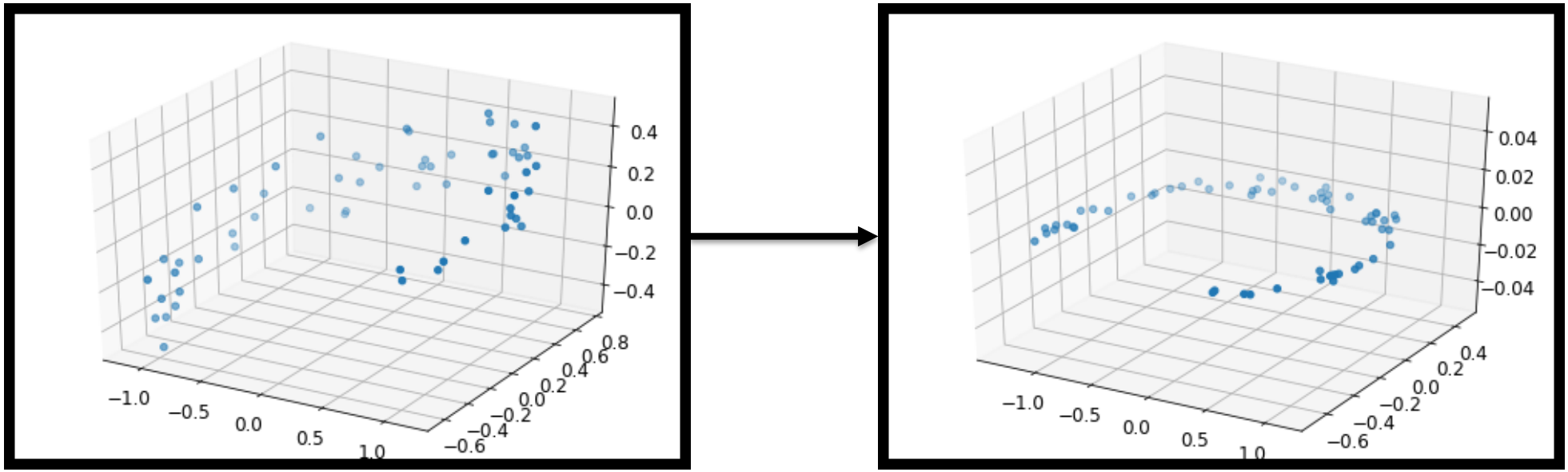
*Code to reproduce the graphs can be seen at **UUID - #S8C3***

Principal Component Analysis

Projecting to D dimensions

```
1 W2 = Vt.T[:, :2]
2 print(W2)
3 X2D = X_centered.dot(W2)
```

```
[[-0.79644131 -0.60471583]
 [-0.60471583  0.79644131]]
```



*Code to reproduce the graphs can be seen at **UUID - #S8C3***

BTS | Barcelona Technology School

MBDS 2020-2021

PCA

Sklearn implementation

PCA – Sklearn implementation

Everything is taken care of

- The PCA implemented in sklearn uses the SVD decomposition used above, but it basically takes care of centering the dataset. The code below does basically the same that we did above

```
1 from sklearn.decomposition import PCA
2
3 pca = PCA(n_components = 2)
4 X2D = pca.fit_transform(X)
```

*Code to reproduce the graphs can be seen at **UUID - #S8C3***

PCA – Sklearn implementation

Accessing the principal components

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

```
1 pca.components_
```

```
array([[-0.93636116, -0.29854881, -0.18465208],
       [ 0.34027485, -0.90119108, -0.2684542 ]])
```

```
[86]  1 pca.components_.T[:, 0]
```

```
array([-0.93636116, -0.29854881, -0.18465208])
```

*Code to reproduce the graphs can be seen at **UUID - #S8C3***

PCA – Sklearn implementation

Explaining the Variance Ratio

- We can access the *explained variance ratio* of each PC through the `explained_variance_ratio`
  - It shows the proportion of the dataset's variance that lies along the axis of each principal component
  - In the case below, both PC's explain up to 98% of the variance, which is very good and we are not loosing a lot of information through the reduction

```
1 pca.explained_variance_ratio_
```

```
array([0.84248607, 0.14631839])
```

*Code to reproduce the graphs can be seen at **UUID - #S8C3***

PCA – Sklearn implementation

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 (e.g., 95%).
- However, if you are reducing dimensionality for data visualization—in that case you will generally want to reduce the dimensionality down to 2 or 3.
- But how do we choose the number of dimensions?
  - Using the `n_components` parameter

PCA – Sklearn implementation

Right number of dimensions: Method 1

- First, we compute the PCA without reducing dimensionality
- Then we compute the minimum number of dimensions needed to preserve 95% of the training set variance
- Once we know “D” we could run PCA again

```
1 pca = PCA()
2 pca.fit(X_train)
3 cumsum = np.cumsum(pca.explained_variance_ratio_)
4 d = np.argmax(cumsum >= 0.95) + 1
```

```
1 d
```

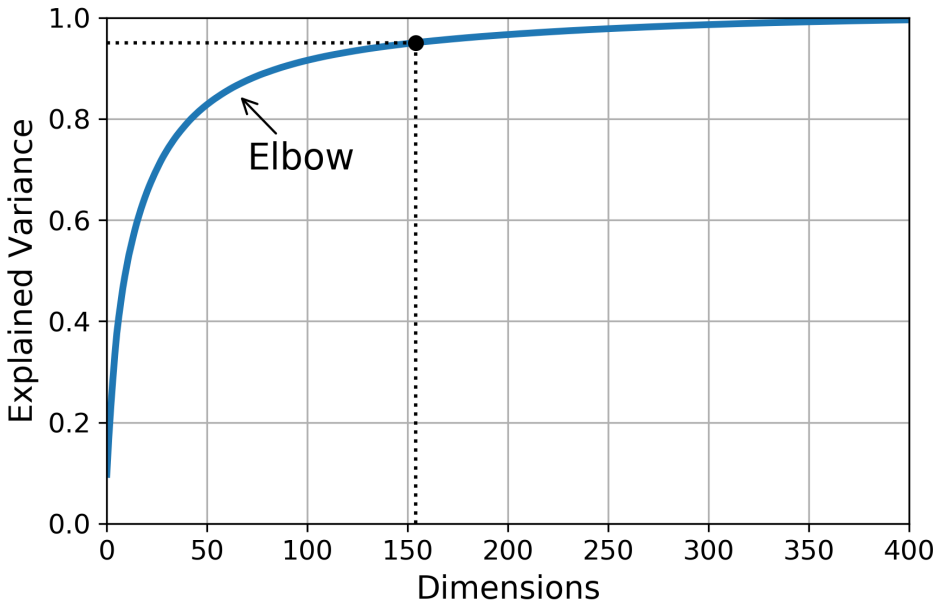
```
154
```

*Code to reproduce the graphs can be seen at **UUID - #S8C4***

PCA – Sklearn implementation

Right number of dimensions: Method 2

- Plot the explained variance against the number of dimensions (plotting cumsum)
- Then we can choose when the explained variance stops growing fast (looking for the elbow)



*Code to reproduce the graphs can be seen at **UUID - #S8C4***

PCA – Sklearn implementation

Right number of dimensions: Method 3

- The easiest and best method is to directly set the expected ratio of variance we want to preserve in `n_components` parameter
- If we use a float between 0 to 1, we can pass to the PCA the desired ratio

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

```
1 pca.n_components_
```

```
154
```

```
1 np.sum(pca.explained_variance_ratio_)
```

```
0.9503684424557437
```

*Code to reproduce the graphs can be seen at **UUID - #S8C4***

PCA for compression

PCA for compression

Throwing out the clutter

- After doing our dimensionality reduction, our datasets are much lighter
- In the lines of code above we applied PCA to MNIST, preserving 95% of its variance
  - Now each instance has 154 features instead of the original 784!!!
  - While most of the variance is preserved, the dataset is now less than 20% of its original size.
- This compression is highly used in high-dimensional datasets such as multispectral imagery and hyperspectral imagery
- This compression can save TONS OF MONEY if you do your processing on the cloud

PCA for compression

Decompression

- While it is always important to keep a backup of the original data, you can also decompress the dataset back to its original size
  - However, it will not be perfect!! We lost some information on the way…
  - But it will look similar to the original data
- This is done by applying the **inverse transformation of the PCA projection**

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

PCA for compression

Decompression

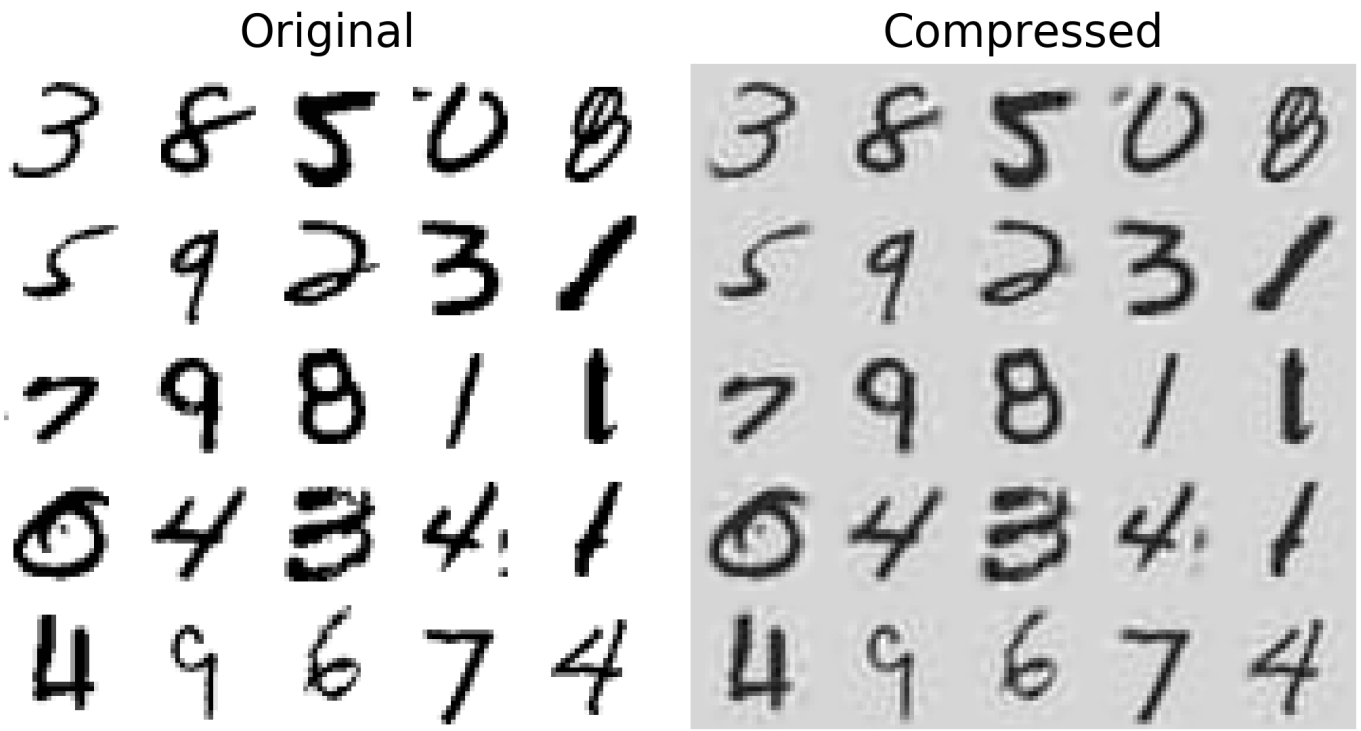
- While it is always important to keep a backup of the original data, you can also decompress the dataset back to its original size
  - However, it will not be perfect!! We lost some information on the way…
  - But it will look similar to the original data
- This is done by applying the **inverse transformation of the PCA projection**

```
# Placing the number of components discovered before
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

*Code to reproduce the graphs can be seen at **UUID - #S8C5***

PCA for compression

Decompression



*Code to reproduce the graphs can be seen at **UUID - #S8C5***

PCA for feature selection

PCA for feature selection

Filtering what's important

- PCA can be used as a feature selector for datasets which have a large amount of features, possibly correlated. Images is an example where this happens often.
- Within image feature selection we will end up with many principal components or eigenvectors (or eigenfaces)

*Code to reproduce the graphs can be seen at **UUID - #S8C7***

Randomized PCA

Randomized PCA

Boosting speed

- If we set up the `svd_solver` to `randomized`, sklearn uses a stochastic algorithm called Randomized PCA that quickly finds approximations 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
- The randomized approach is then dramatically faster than full SVD when *d* is much smaller than *n*

```
t_start = time.time()
pca = PCA(200, svd_solver='randomized')
pca.fit(faces.data)
print(f'Finished with randmomized svd method i
pca.components_[:, 0]
```

*Code to reproduce the graphs can be seen at **UUID - #S8C7***

Randomized PCA

Boosting speed

- Usually the `svd_solver` is set to `auto`
- Sklearn automatically uses the randomized PCA when
  - Samples (m) or features (n) are greater than 500
  - Dimensions or components (d) are less than 80% of m or n
  - Otherwise, it swill use the SVD approach
- We can force the usage of full SVD by using `svd_solver=full`

```
t_start = time.time()
pca = PCA(200, svd_solver='randomized')
pca.fit(faces.data)
print(f'Finished with randmomized svd method i
pca.components_[:, 0]
```

*Code to reproduce the graphs can be seen at **UUID - #S8C7***

PCA Sklearn

svd_solver{'auto', 'full', 'arpack', 'randomized'}

- **full**
  - run exact SVD calling the standard LAPACK solver via scipy.linalg.svd and select the components by postprocessing
- **arpack**
  - run SVD truncated to n_components calling ARPACK solver via scipy.sparse.linalg.svds. It requires strictly *0 < n_components < X.shape[1]*
- **randomized**
  - Randomized truncated SVD by the method of Halko et al. 2009
  - Depends on the shape of the input data and the number of components to extract

http://www.suyongeum.com/ML/tutorials/tutorial4-yang.pdf

Incremental PCA

Incremental PCA

PCA meets real Big Data

- The problem with the PCA implementations above is that they **require the whole dataset to fit in memory** in order for the SVD algorithm to run
- Real world datasets will rarely fit in memory, so we will need to load them in batches
- **Incremental PCA (IPCA)** algorithms are implemented in sklearn so that we can deal with this
  - We can split the training into mini-batches and feed IPCA one batch at a time

*Code to reproduce the graphs can be seen at **UUID - #S8C8***

Incremental PCA

PCA meets real Big Data

```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    print(".", end="")
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

*Code to reproduce the graphs can be seen at **UUID - #S8C8***

BTS | Barcelona Technology School

MBDS 2020-2021

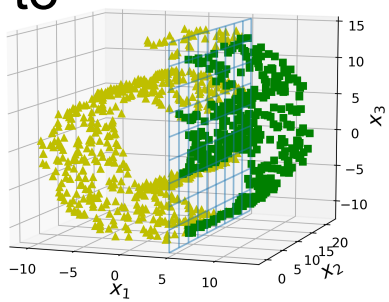
Dimensionality reduction

PCA Recap

PCA

Recap

- PCA is one of the most widely used dimensionality reduction algorithms
- It can be used to speed up training but always with care, as performance can drop
- It can be used to compress data
- It can also be used for feature selection
- In some cases, **it can make training go slower!**
  - Some datasets are not projectable into a hyperplane and another type of dimensionality reduction needs to be used (MANIFOLD LEARNING)



BTS | Barcelona Technology School

MBDS 2020-2021

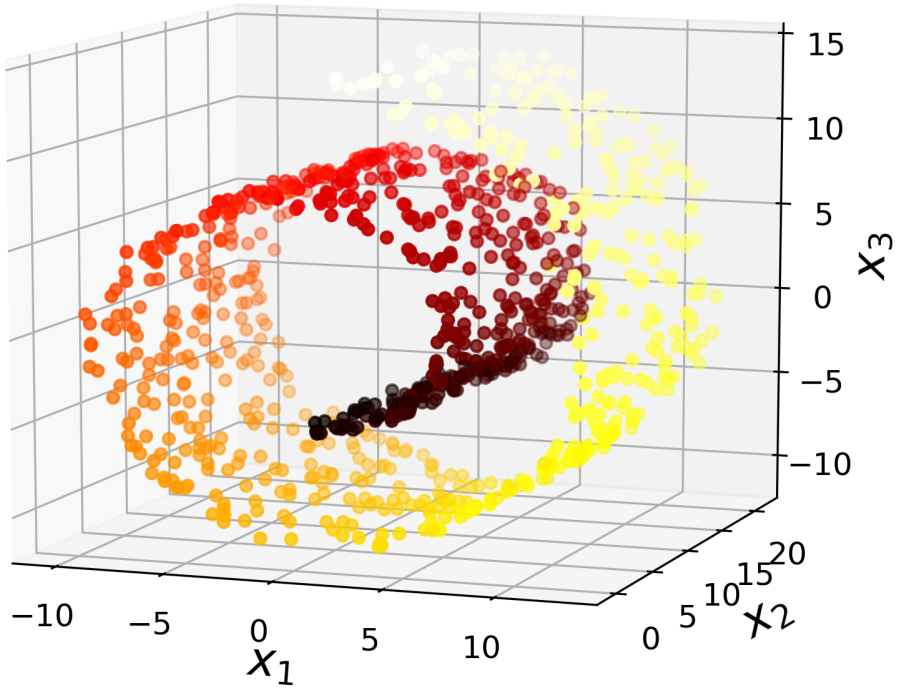
Dimensionality reduction

Manifold Learning

Manifold learning

Projection does not solve everything

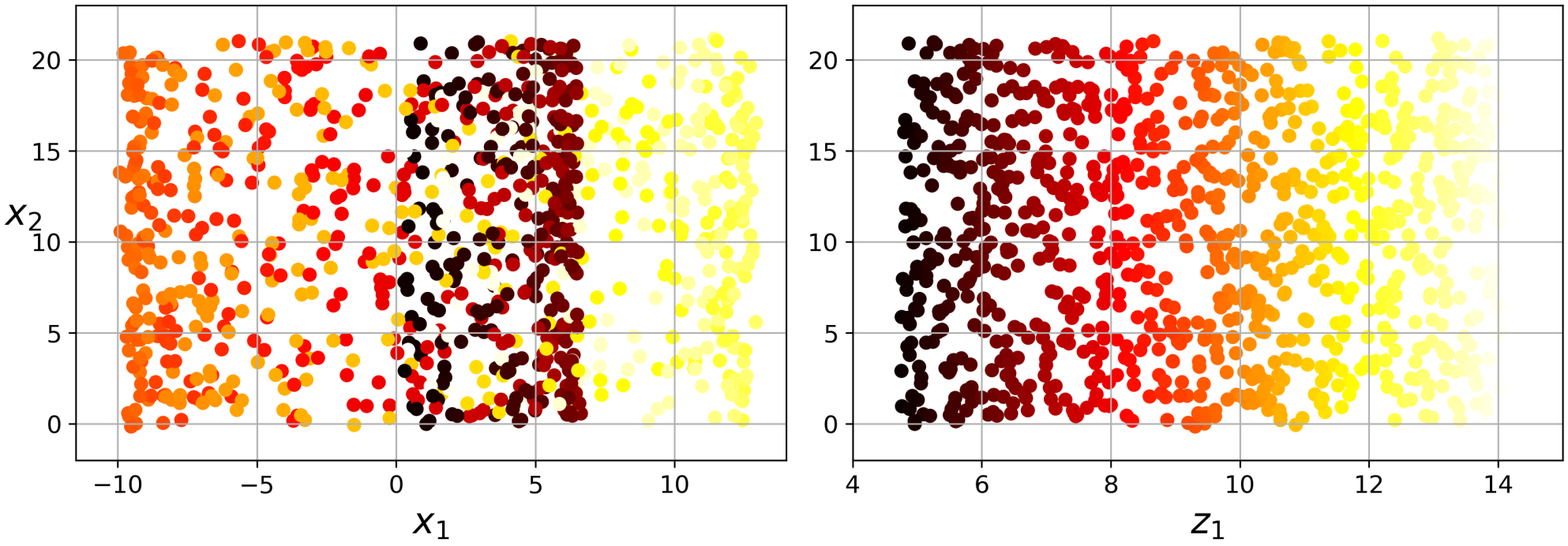
- Projection is not always the best approach for dimensionality reduction
- Sometimes subspaces may twist and turn, consequently projection messes up the dataset



Manifold learning

Projection does not solve everything

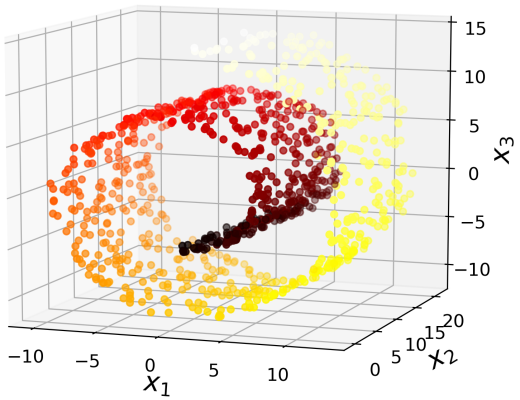
- If we drop the x3 variable, we will be squashing the different data together, creating more complex dataset by reducing dimensions
- What we really want is to unroll the dataset in an organized manner



Manifold learning

The manifold hypothesis

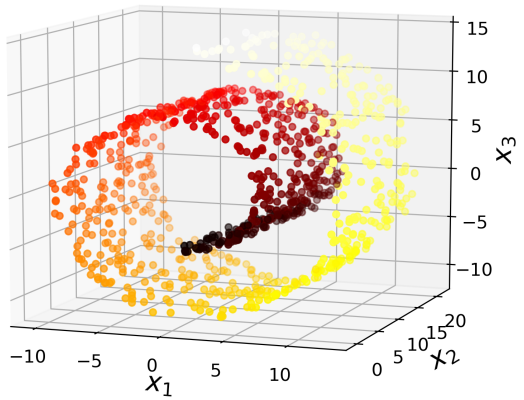
- The “cinnamon roll” is an example of a **2D *manifold***
  - A 2D shape that can be bent and twisted in a higher dimensional space (like a cannelloni)
- A *d*-dimensional manifold is a part of an *n*-dimensional space (where $d < n$) that locally resembles a *d*-dimensional hyperplane.
- In the case of the “cinnamon roll” , $d = 2$ and $n = 3$:
  - It locally resembles a 2D plane, but it is rolled in the third dimension.



Manifold learning

The manifold hypothesis

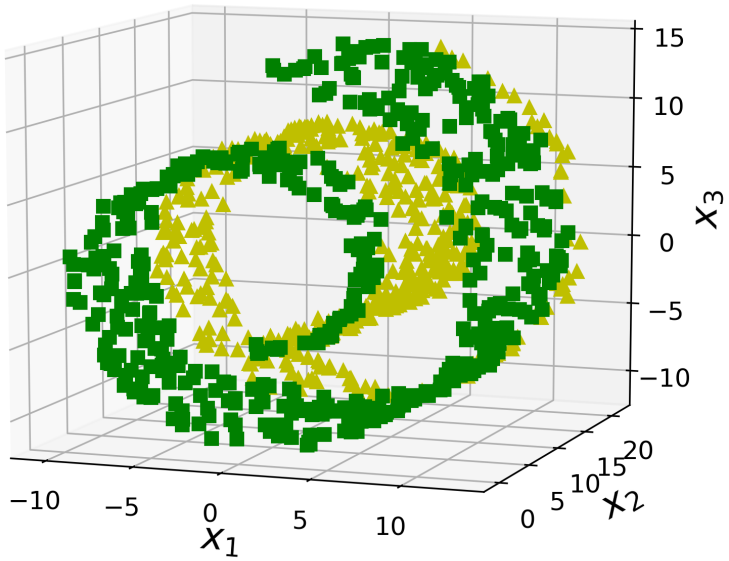
- Manifold Learning is a type of dimensionality reduction in which the *manifold* in which the training instances lie is modelled (unrolled, massaged, etc.)
- Manifolds are unrolled and modelled based on the **manifold hypothesis** which states that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold
  - This is similar to what projection stood for, but there are no spatial twists on projected datasets

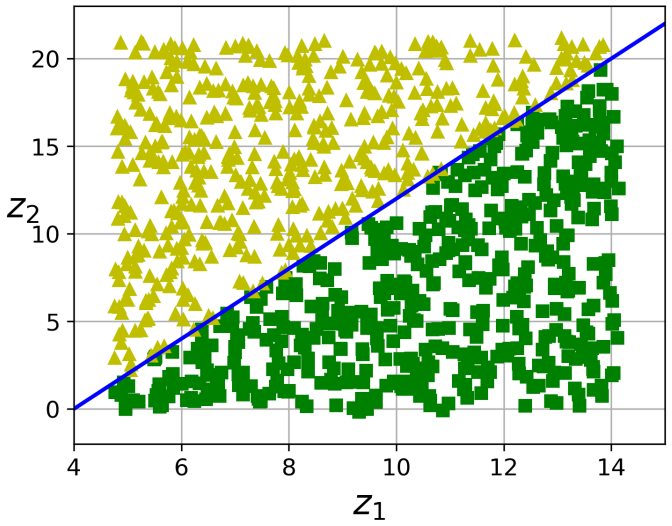


Manifold learning

The manifold hypothesis

- The manifold assumption is often accompanied by another implicit assumption:
  - That the task at hand (e.g., classification or regression) will be simpler if expressed in the lower-dimensional space of the manifold.

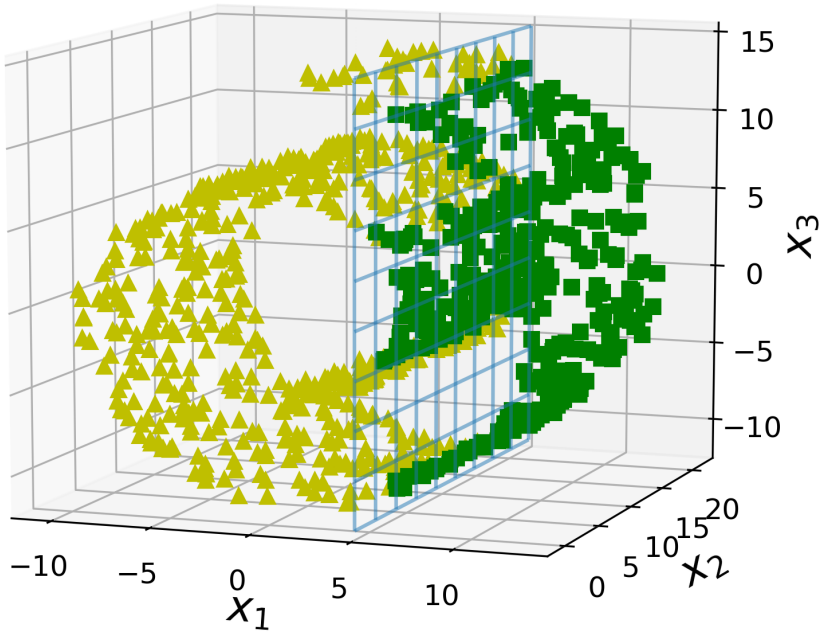


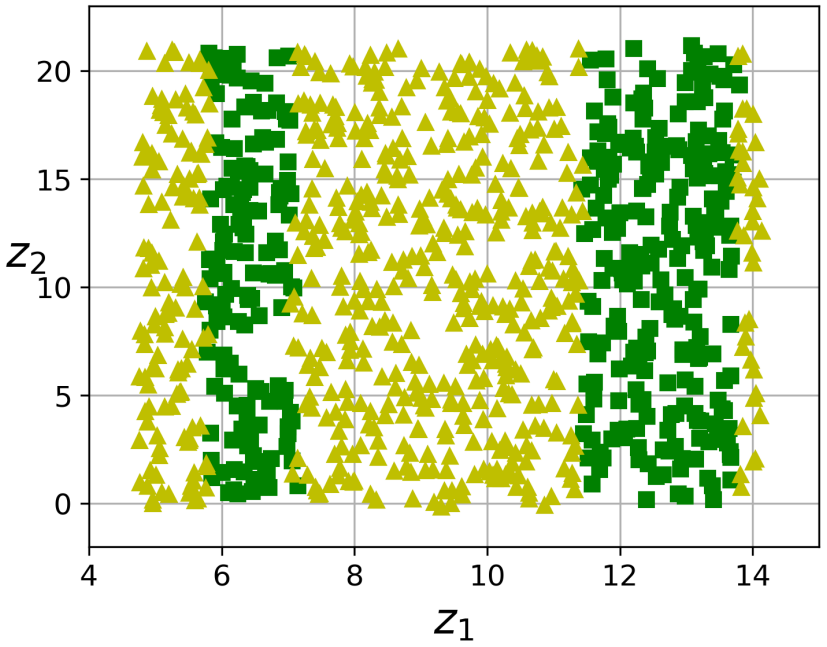


Manifold learning

The manifold hypothesis

- However, this assumption does not always hold.
- 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).





Manifold learning

The manifold hypothesis

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

BTS | Barcelona Technology School

MBDS 2020-2021

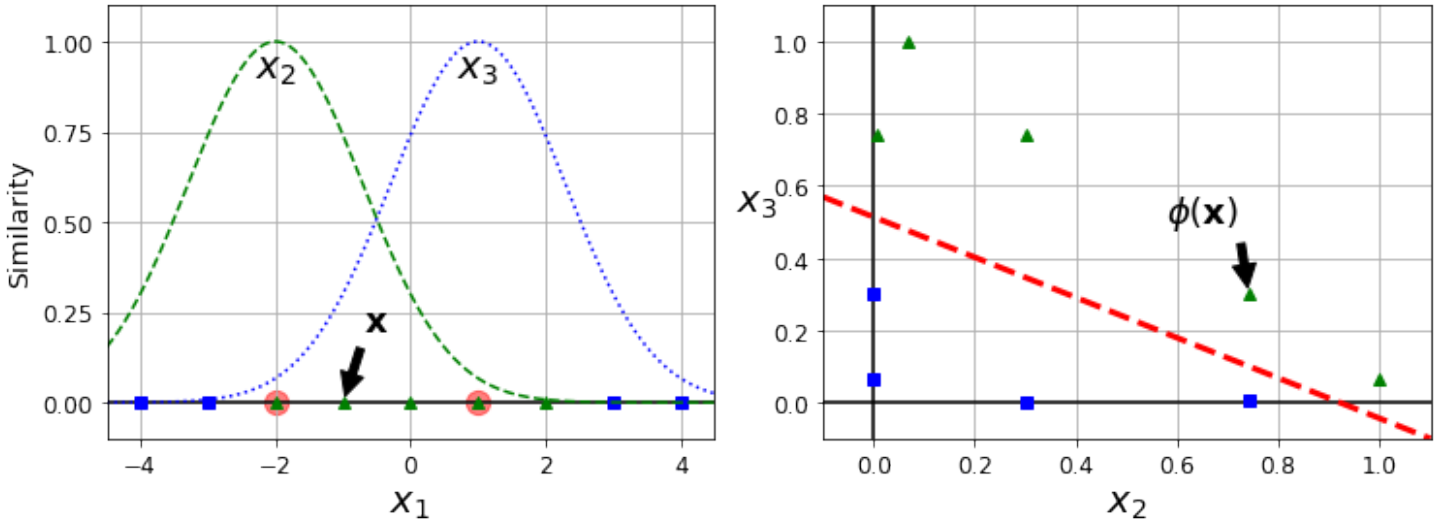
Manifold Learning

Kernel PCA

Kernel PCA

The kernel trick

- The **kernel trick** is a common mathematical technique used in SVMs to tackle intractable nonlinear data
- The kernel trick adds features computed using a *similarity function* that measures how much each instance resembles to a particular *landmark*

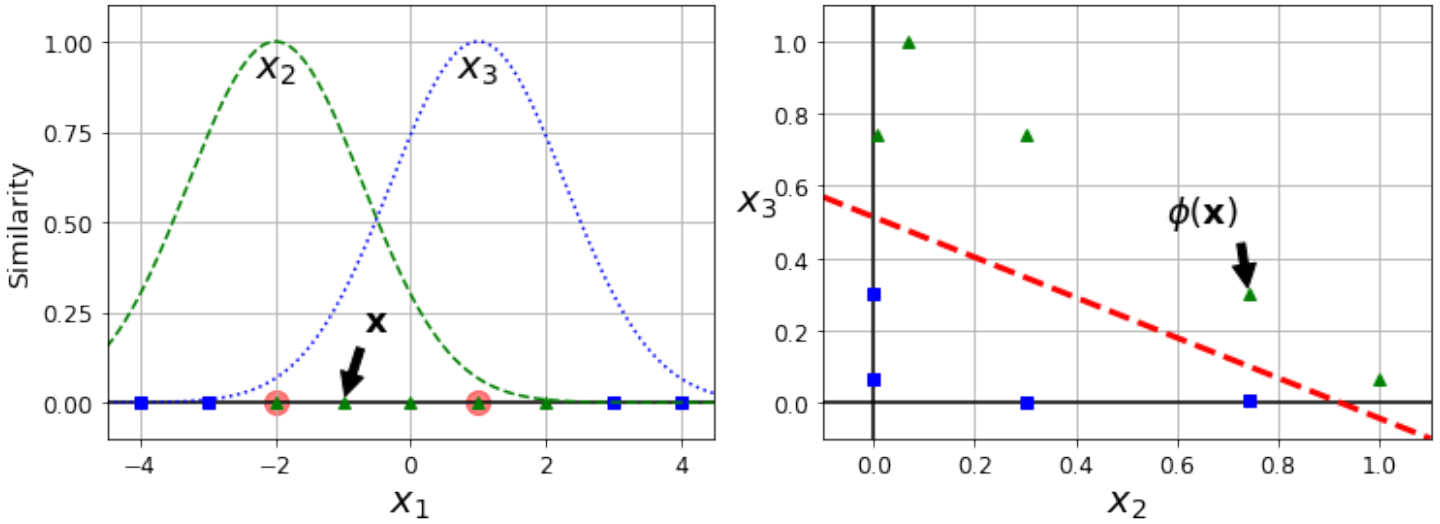


Kernel PCA

The kernel trick

- We can take a one-dimensional dataset which is nonlinearly separable
- Place two landmarks at x=-2 and x=1
- Using the Gaussian Radial Basis Function (RBF) as a similarity function with gamma = 3

$$\phi\gamma(\mathbf{x}, \ell) = \exp\left(-\gamma \| \mathbf{x} - \ell \|^2\right)$$



Kernel PCA

The kernel trick

- The kernel trick then maps instances into a very high-dimensional space (the feature space), enabling nonlinear classification with SVMs
- Those linear decision boundaries in the high-dimensional feature space correspond to a complex nonlinear decision boundary in the *original space*
- You guessed it, the very same trick can be applied to PCA, allowing complex nonlinear projections for dimensionality reduction
  - Kernel PCA is **extremely good** at preserving clusters of instances after projection (keeping close to original shape)
  - And sometimes can **unroll** datasets that are close to a twisted manifold

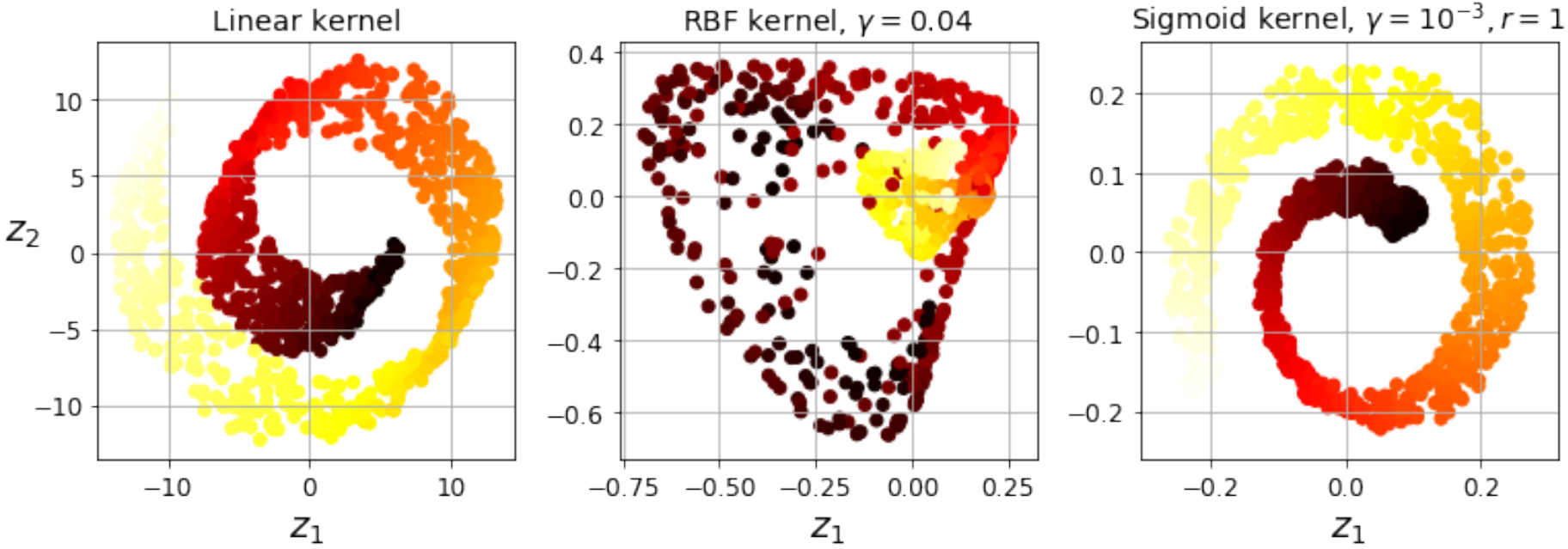
Kernel PCA

The kernel trick

- We can use the same kernels that we used in SVM

```
from sklearn.decomposition import KernelPCA

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



*Code to reproduce the graphs can be seen at **UUID - #S8C9***

Kernel PCA

Selecting kernels and hyperparameters

- KPCA is an unsupervised learning algorithm, so there is not a lot of performance gain by tuning hyperparameters
- However, dimensionality reduction is often used as a preparation step for a supervised learning task (like classification) so...
- We can build a pipeline that incorporates:
  - First: reduction of dimensionality to two dimensions using KPCA
  - Second: application of an algorithm for dataset separation and classification
  - We can use `GridSearchCV` in order to find the best kernel and gamma values for KPCA in order to get the best of the classification

Kernel PCA

Selecting kernels and 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(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)
```

*Code to reproduce the graphs can be seen at **UUID - #S8C9***

Kernel PCA

Selecting kernels and hyperparameters

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

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

```
1 rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.0433,
2                     fit_inverse_transform=True)
3 X_reduced = rbf_pca.fit_transform(X)
4 X_preimage = rbf_pca.inverse_transform(X_reduced)
```

*Code to reproduce the graphs can be seen at **UUID - #S8C9***

Kernel PCA

Selecting kernels and hyperparameters

The main reason we cannot play with KernelPCA to get the ideal number of components is because its “impossible” to calculate meaningful variances, because the kernel transformation of the data lives in a different feature space. That’s also why we cannot access `explained_variance_ratio`.
So, we should not interpret KPCA as bare PCA.

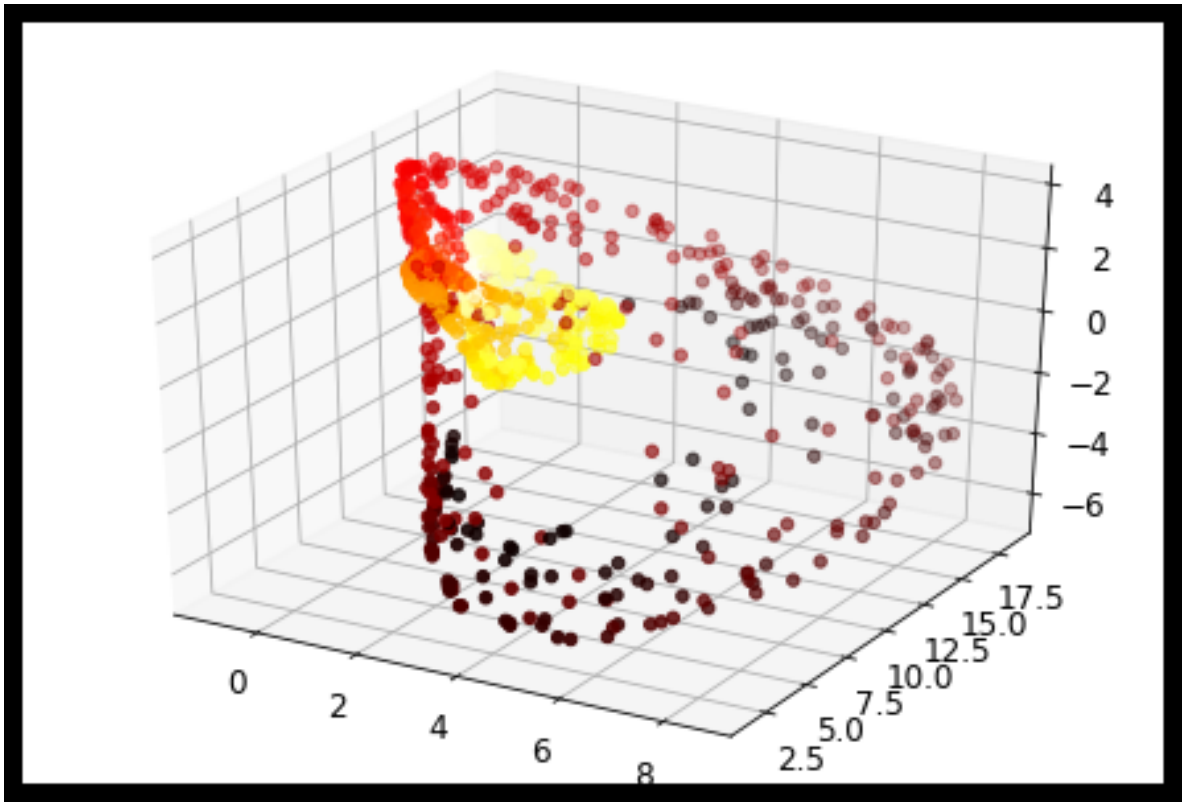
Kernel PCA

Reconstruction

- Precisely because the kernel trick maps the training set to an infinite-dimensional feature space
- If we could invert the linear PCA step for a given instance in the reduced space, the reconstructed point would lie in feature space, not in the original space
- 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***

Kernel PCA

Reconstruction



*Code to reproduce the graphs can be seen at **UUID - #S8C9***

Kernel PCA

Reconstruction

In order to access the `inverse_transform` method that allows us to access the *preimage* we need to set the parameter `fit_inverse_transform` to `True`

Exercises

Dimensionality Reduction

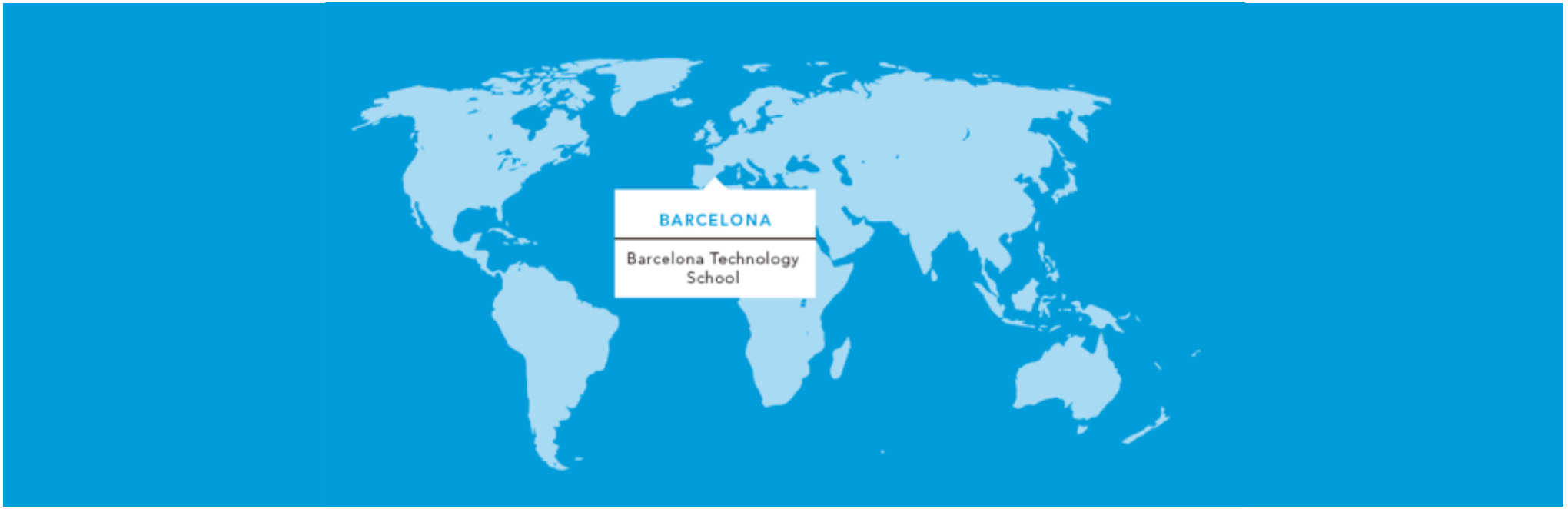
Exercises

- UUID - #S11E1 [Optional]
- UUID - #S11E2
- UUID - #S11E3

Resources

Important resources

- Lex Friedman Series (MIT)
- Sklearn docs
- “Kernel Principal Component Analysis,” B. Scholkopf, A. Smola, K. Muller (1999).
- “Nonlinear Dimensionality Reduction by Locally Linear Embedding,” S. Roweis, L. Saul (2000).
- Aurelien Geron, “Hands-on machine learning with scikit-learn, Keras & Tensorflow”


BARCELONA
Barcelona Technology School