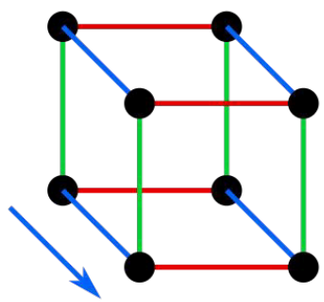


Dimensionality Reduction

Introduction - Curse of Dimensionality

Some problem sets may have

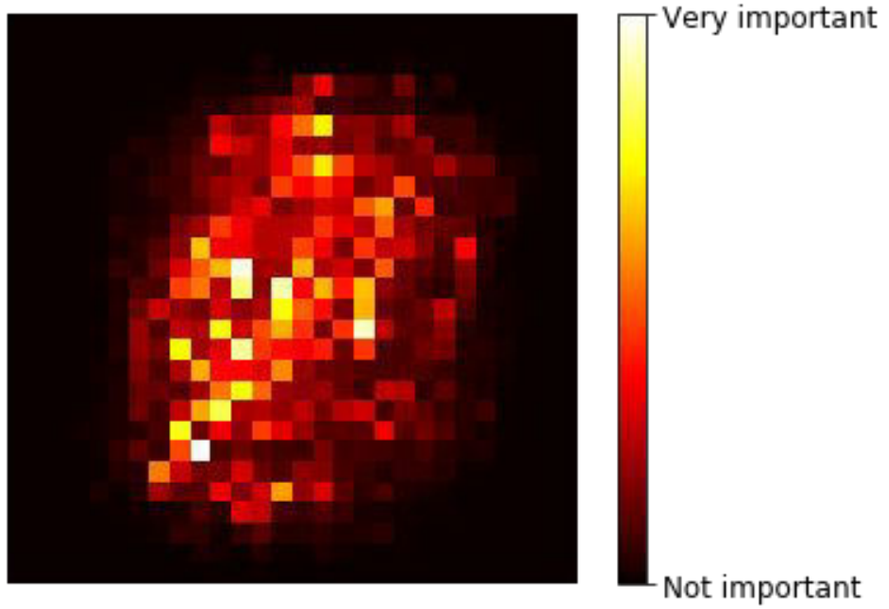
- Large number of feature set
- Making the model extremely slow
- Even making it difficult to find a solution
- This is referred to as ‘Curse of Dimensionality’

Introduction - Curse of Dimensionality

Example

- MNIST Dataset
  a. Each pixel was a feature
  b. (28*28) number of features for each image
  c. Border feature had no importance and could be ignored

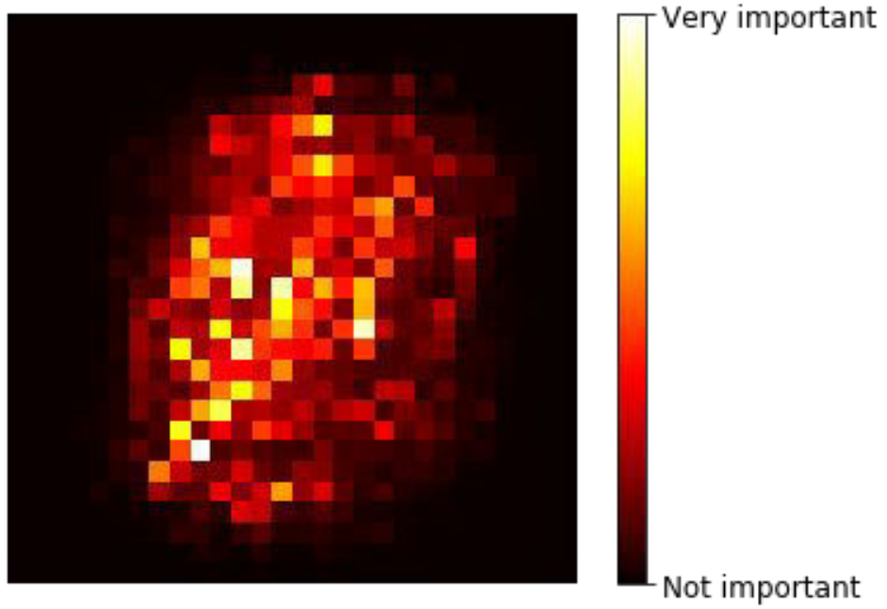
Border data (features) can be ignored for all datasets



Introduction - Curse of Dimensionality

Example

- MNIST Dataset
  - Also, neighbouring pixels are highly correlated
  - Neighbouring pixels can be merged into one without losing much of information
  - Hence, further reducing the dimensions or features



Introduction - Curse of Dimensionality

Some benefits of dimension reduction

- **Faster** and more efficient model
- Better **visualization** to gain important insights by detecting patterns

Drawbacks:

- Lossy - we lose some information - we should try with the original dataset before going for dimension reduction

Introduction - Curse of Dimensionality

Some important facts

- Q. Probability that a random point chosen in a unit metre square is 0.001 m from the border?
- Ans. ?

Introduction - Curse of Dimensionality

Some important facts

- Q. Probability that a random point chosen in a unit metre square is 0.001 m from the border?
- Ans. 0.004 = 1 - (0.998)**2

Introduction - Curse of Dimensionality

Some important facts

- Q. Probability that a random point chosen in a unit metre square is 0.001 m from the border?
- Ans. 0.004, meaning chances are very low that the point will extreme along any dimension

- Q. Probability that a random point chosen on a 10,000 dimensional unit metre hypercube is 1 mm from the border?
- Ans. >99.999999 %

Introduction - Curse of Dimensionality

Some more important facts

If we pick 2 points randomly on a unit square

- The distance between these 2 points shall be roughly 0.52

If we pick 2 points randomly in a 1,000,000 dimension hypercube

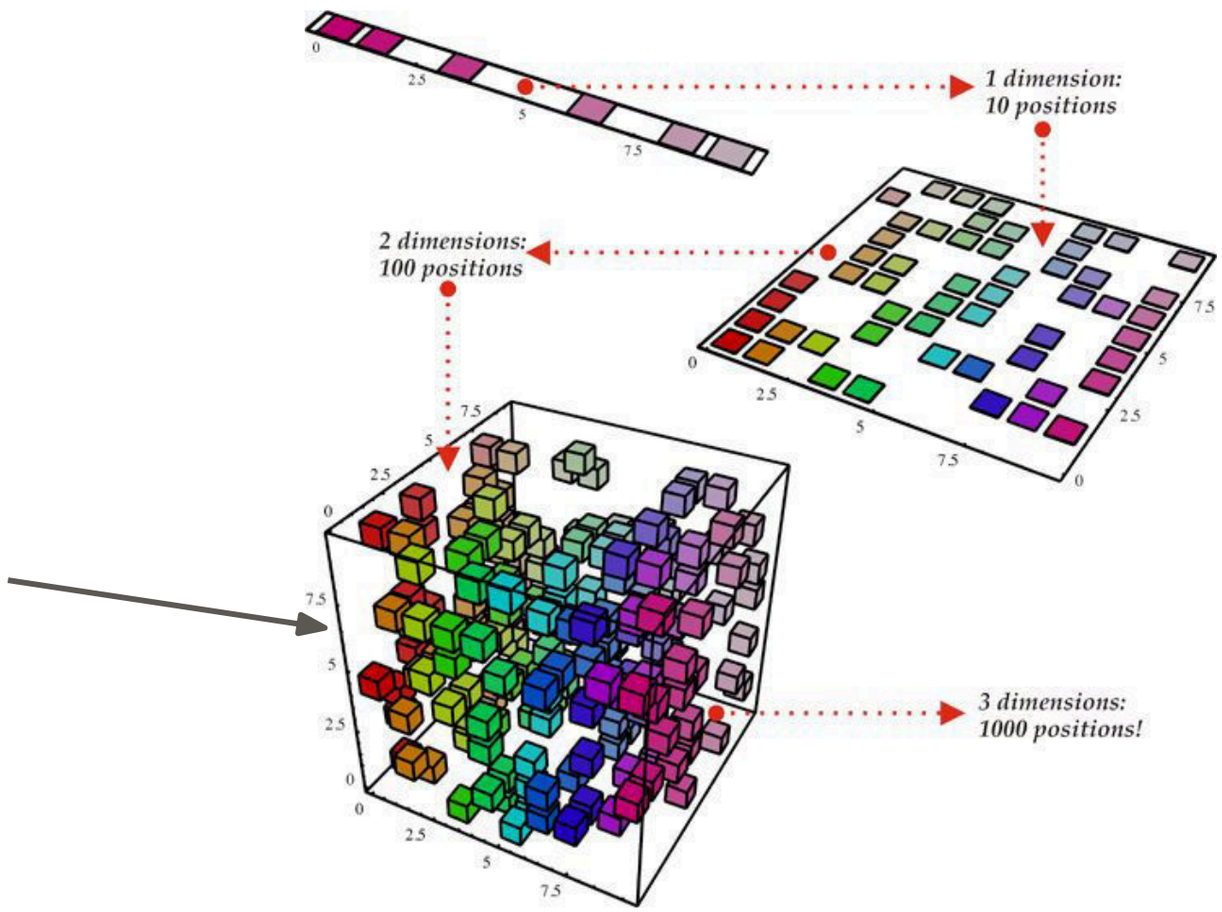
- The distance between these 2 points shall be roughly sqrt(1000000/6)

Introduction - Curse of Dimensionality

Some important observations about large dimension datasets

- Higher dimensional datasets are at risk of being very sparse
- Most training sets are likely to be far away from each other

Instances much more scattered in higher dimensions, hence sparse



Introduction - Curse of Dimensionality

New dataset (test) dataset will also likely be far away from any training instance

- making predictions much less reliable

Hence,

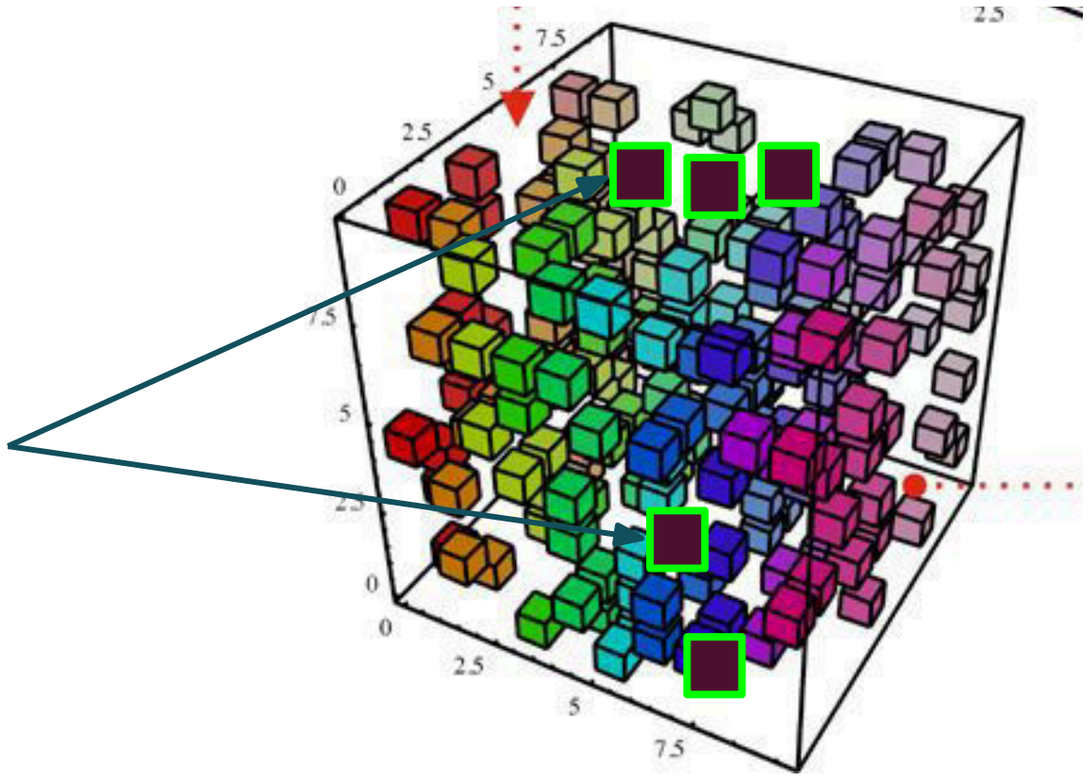
- **more dimensional the training set is,**
- **the greater the risk of overfitting.**

Introduction - Curse of Dimensionality

How to reduce the curse of dimensionality?

- Increase the size of training set (number of datasets) to reach a sufficient density of training instances
  - However, number of instances required to reach a given density grows exponentially with the number of dimensions (features)

Adding more instances will increase the density



Introduction - Curse of Dimensionality

How to reduce the curse of dimensionality?

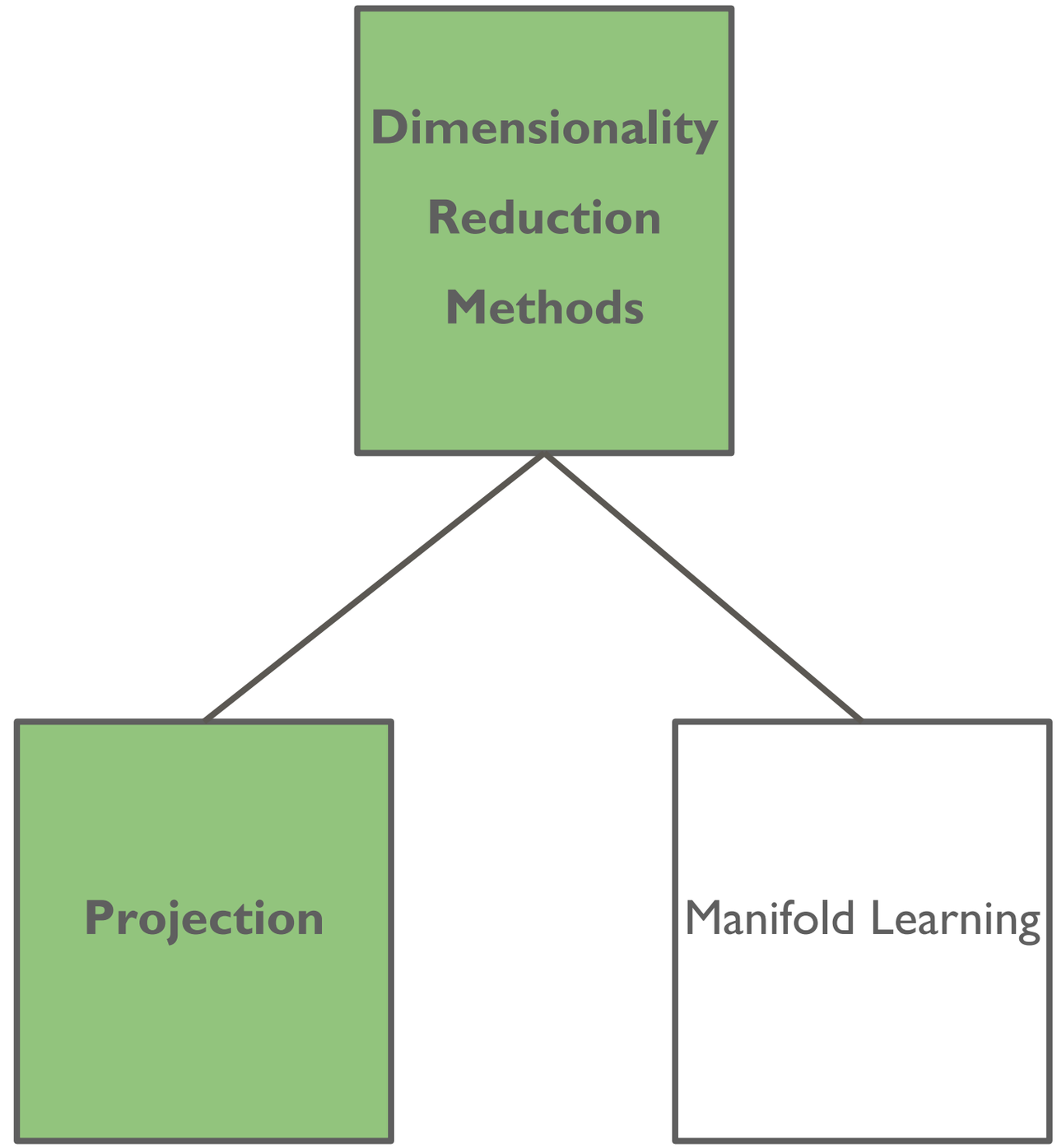
- Example:
  - For a dataset with 100 features
  - Will need more training datasets than atoms in observable universe
  - To have the instances on an average 0.1 distance from each other (assuming they are spread out equally)
- Hence, we reduce the dimensions

Dimensionality Reduction

Main approaches for dimensionality reduction

- Projection
- Manifold Learning

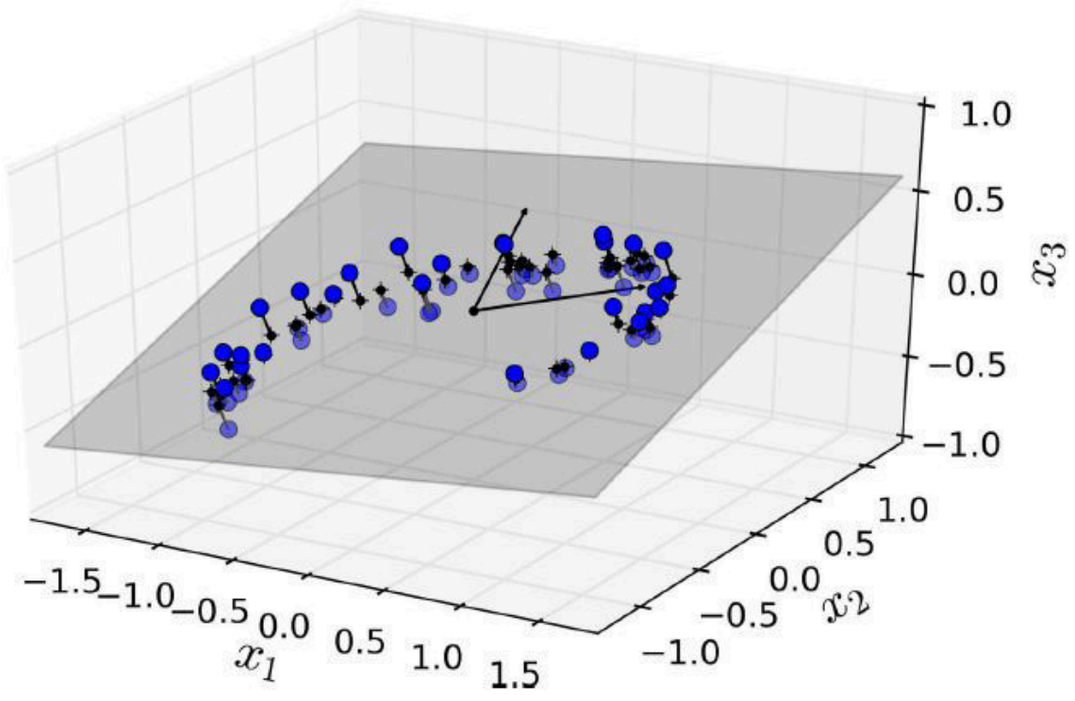
Dimensionality Reduction



Dimensionality Reduction - Projection

Most real-world problems do not have training instances spread out across all dimensions

- Many features are almost constant -
- While others are correlated

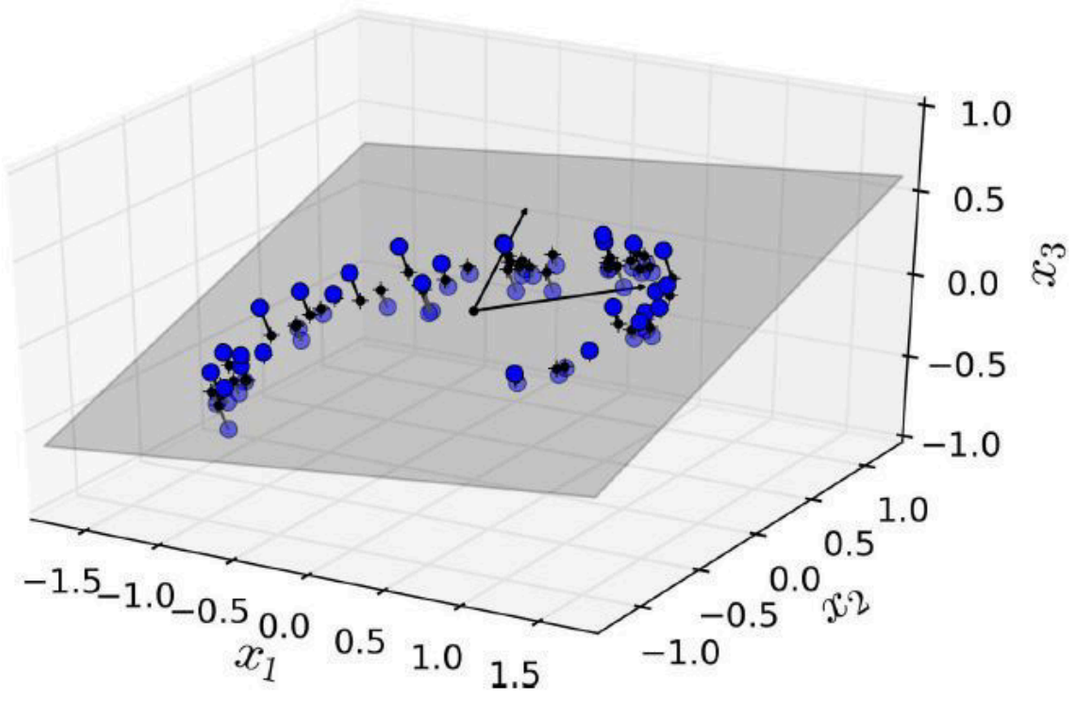


Q. How many features are there in the above graph?

Dimensionality Reduction - Projection

Most real-world problems do not have training instances spread out across all dimensions

- Many features are almost constant -
- While others are correlated

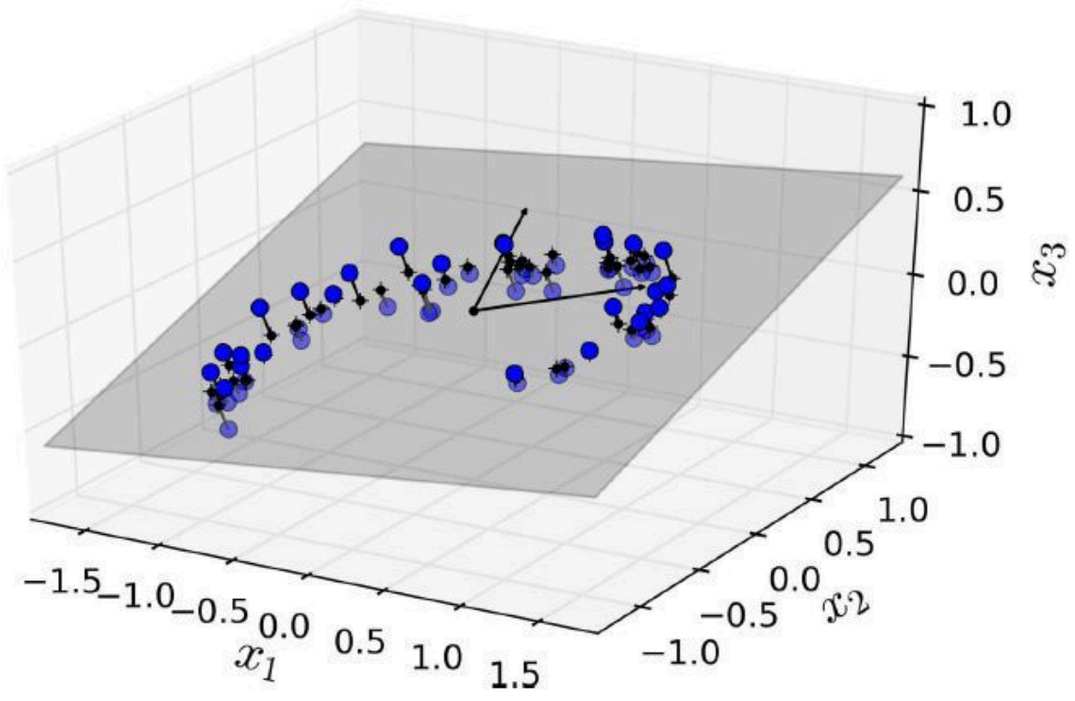


Q. How many features are there in the above graph? **3**

Dimensionality Reduction - Projection

Most real-world problems do not have training instances spread out across all dimensions

- Many features are almost constant -
- While others are correlated

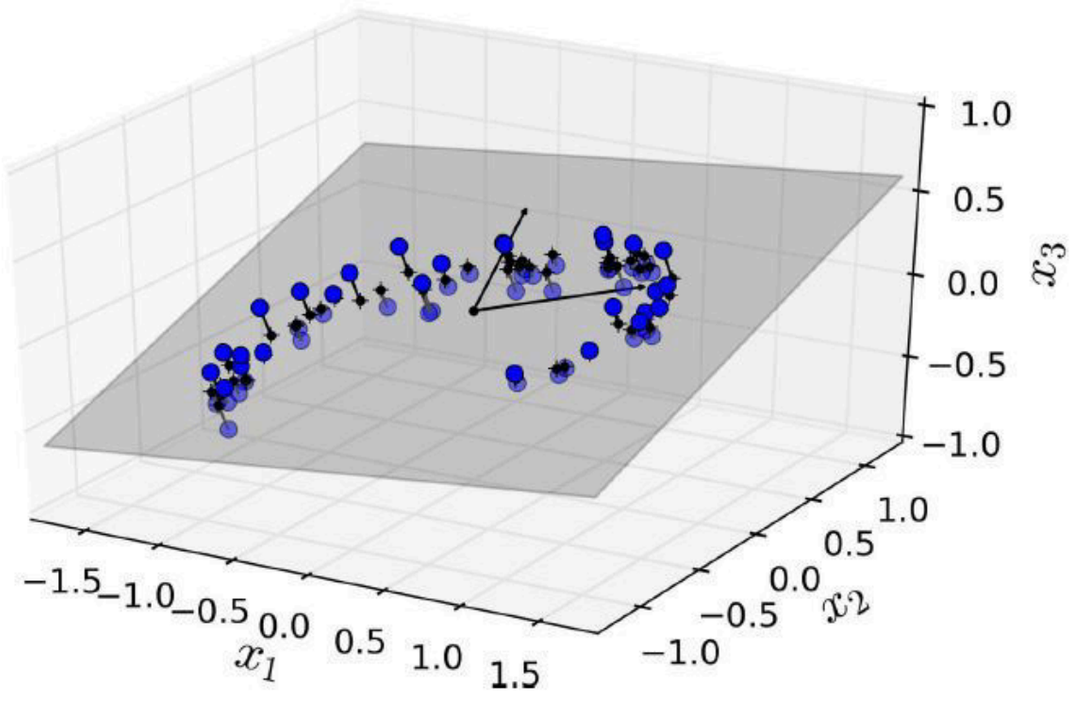


Q. Which of the feature is almost constant for almost all instances? x1, x2 or x3?

Dimensionality Reduction - Projection

Most real-world problems do not have training instances spread out across all dimensions

- Many features are almost constant -
- While others are correlated

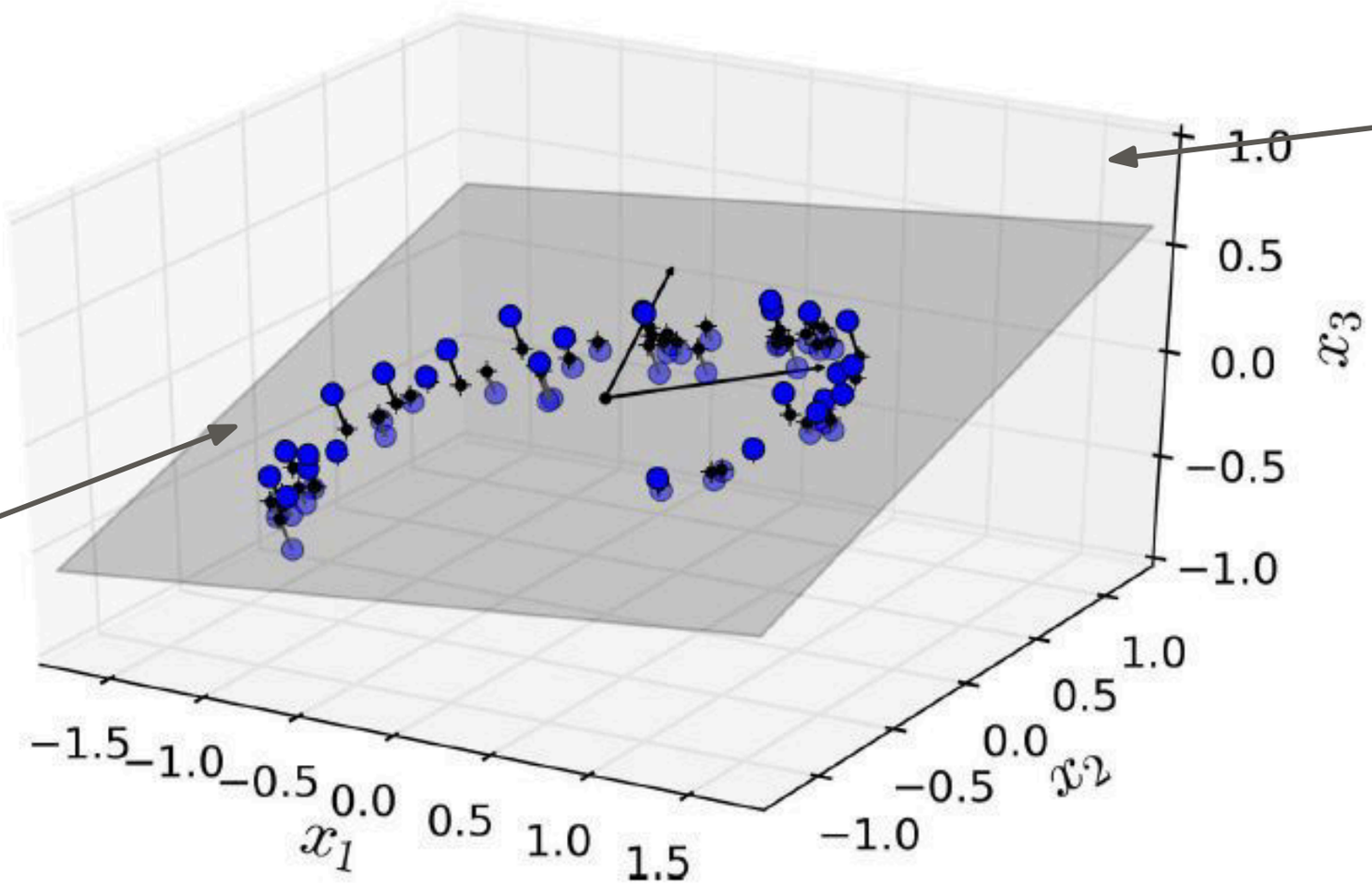


Q. Which of the feature is almost constant for almost all instances? **Ans: x3**

Dimensionality Reduction - Projection

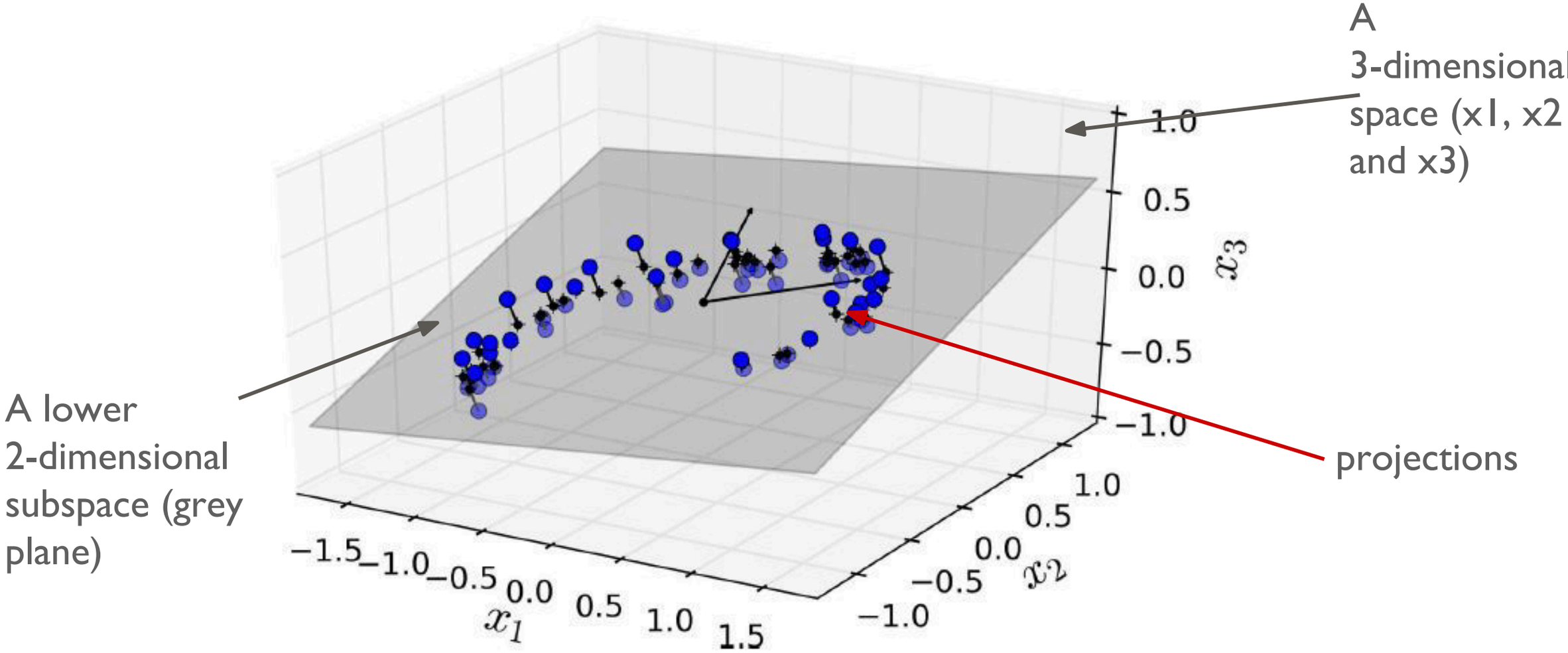
Most of the training instances actually lie within (or close to) a much **lower-dimensional subspace**.

- Refer the diagram below



Dimensionality Reduction - Projection

- Not all instances are ON the 2-dimensional subspace
- If we project all the instances perpendicularly on the subspace
  - We get the new 2d dataset with features z1 and z2



Remember projection from Linear Algebra?

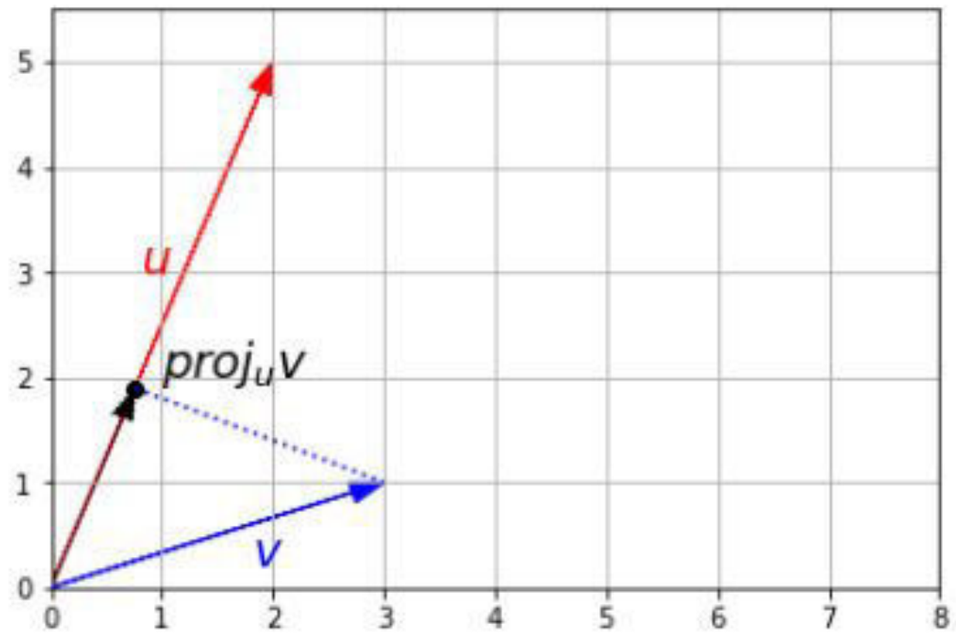
As we have seen in linear algebra session,

- A vector v can be projected onto
- another vector u
- By doing a dot product of v and u.

$$\mathbf{proj_u v} = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|^2} \times \mathbf{u}$$

Which is equivalent to:

$$\mathbf{proj_u v} = (\mathbf{v} \cdot \hat{\mathbf{u}}) \times \hat{\mathbf{u}}$$



Remember projection from Linear Algebra?

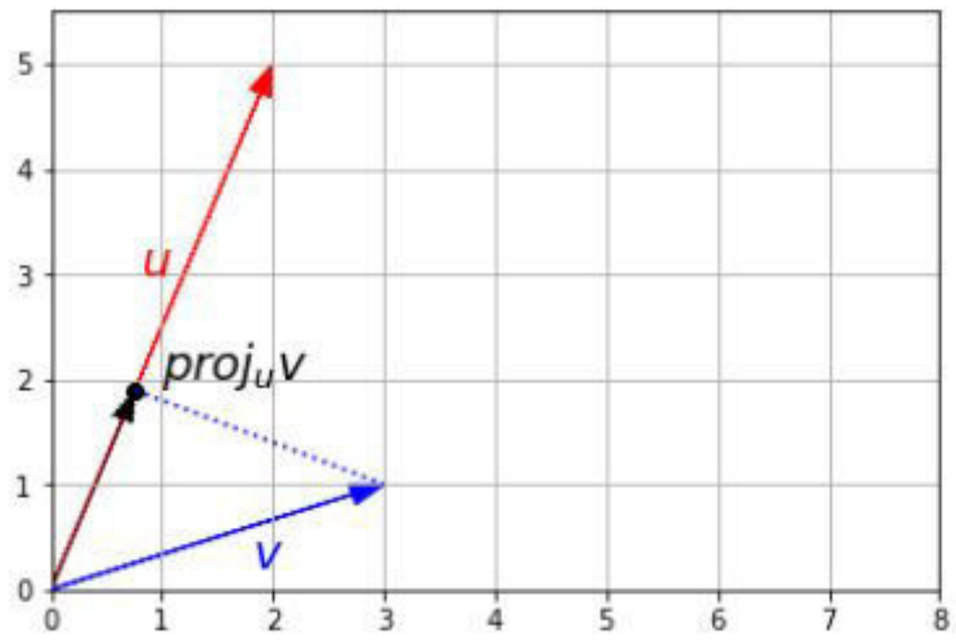
Q. For the graph below, which of these is true?

a. Vector v is orthogonal to u
b. Vector v is projected into vector u
c. Vector u is projected into vector v

$$\mathbf{proj_u v} = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|^2} \times \mathbf{u}$$

Which is equivalent to:

$$\mathbf{proj_u v} = (\mathbf{v} \cdot \hat{\mathbf{u}}) \times \hat{\mathbf{u}}$$



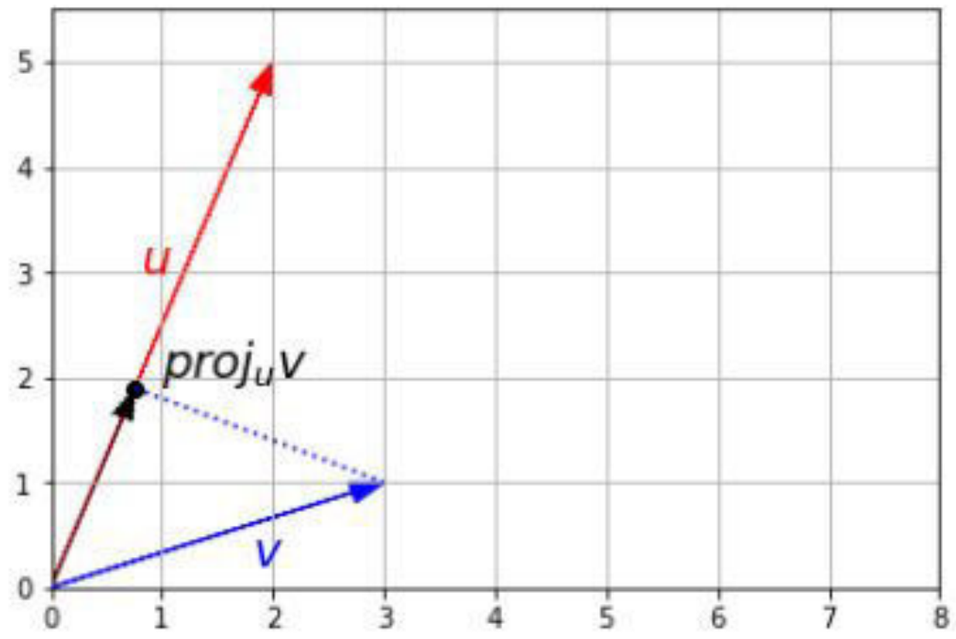
Remember projection from Linear Algebra?

A. For the graph below, which of these is true?
   a. Vector v is orthogonal to u
   b. ✓ Vector v is projected into vector u
   c. Vector u is projected into vector v

$$\mathbf{proj_u v} = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|^2} \times \mathbf{u}$$

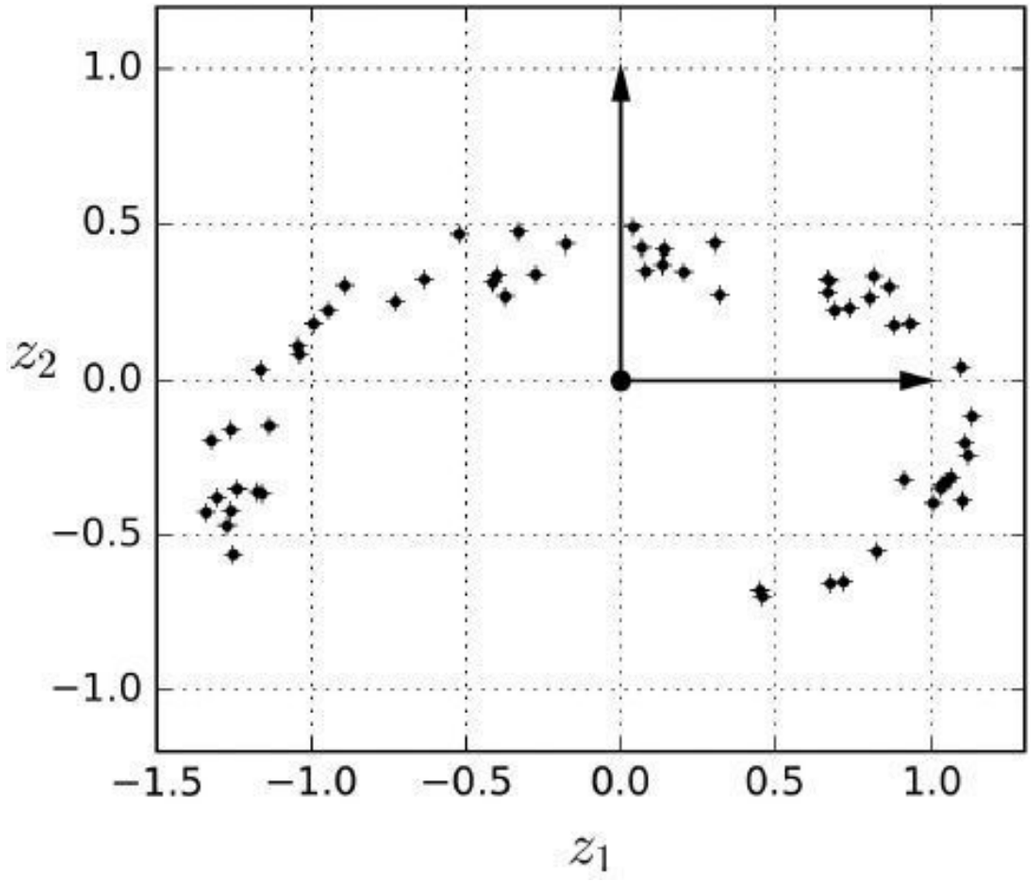
Which is equivalent to:

$$\mathbf{proj_u v} = (\mathbf{v} \cdot \hat{\mathbf{u}}) \times \hat{\mathbf{u}}$$



Dimensionality Reduction - Projection

- Like we project a vector onto another, we can project a vector onto a plane by a dot product.
- If we project all the instances perpendicularly on the subspace
  - We get the new 2d dataset with features z1 and z2



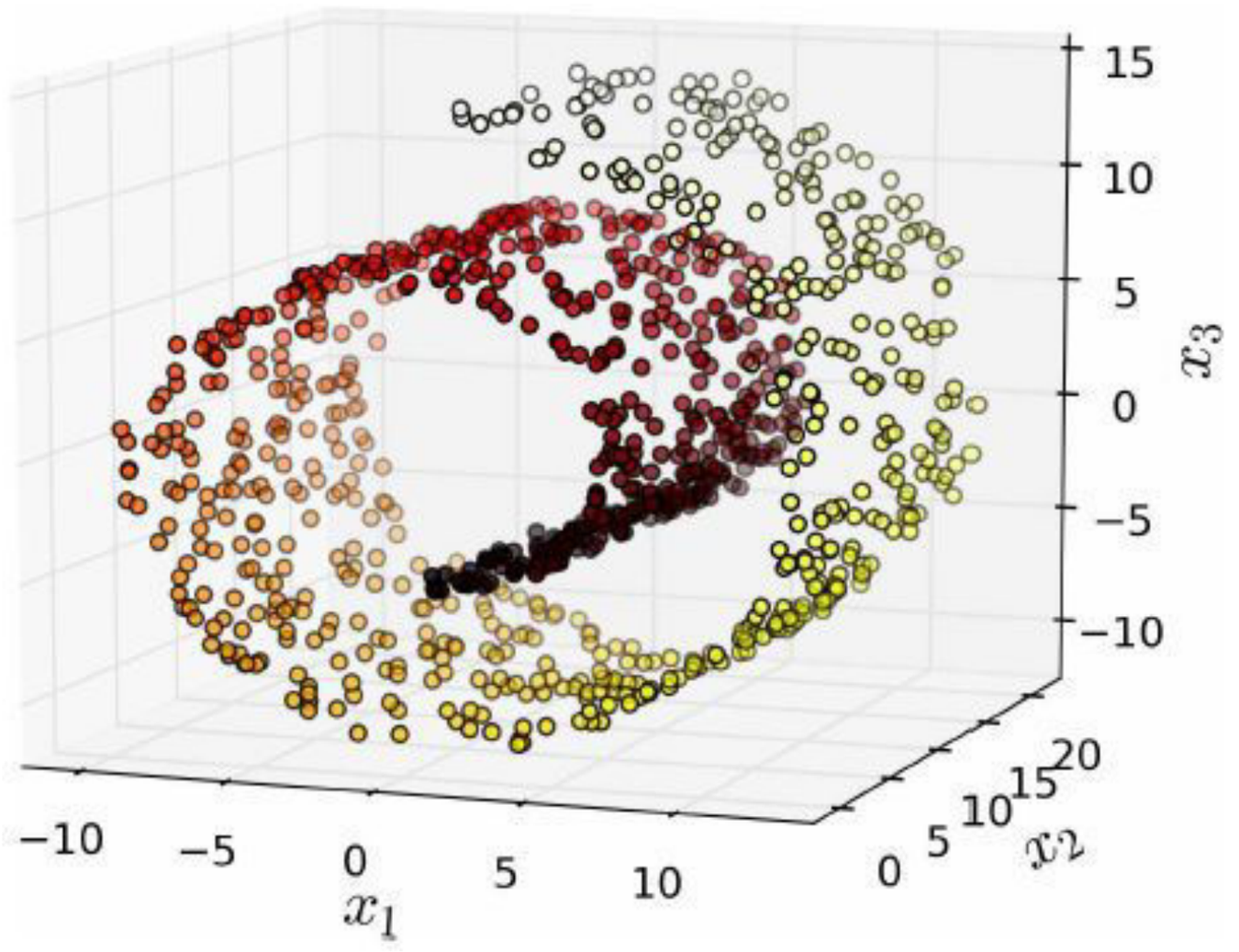
Dimensionality Reduction - Projection

- The above example is demonstrated on notebook
  - Download the 3d dataset
  - Reduce it to 2 dimensions using PCA - a dimensionality reduction technique based on projection
  - Define a utility to plot the projection arrows
  - Plot the 3d dataset, the plane and the projection arrows
  - Draw the 2d equivalent

Switch to Notebook

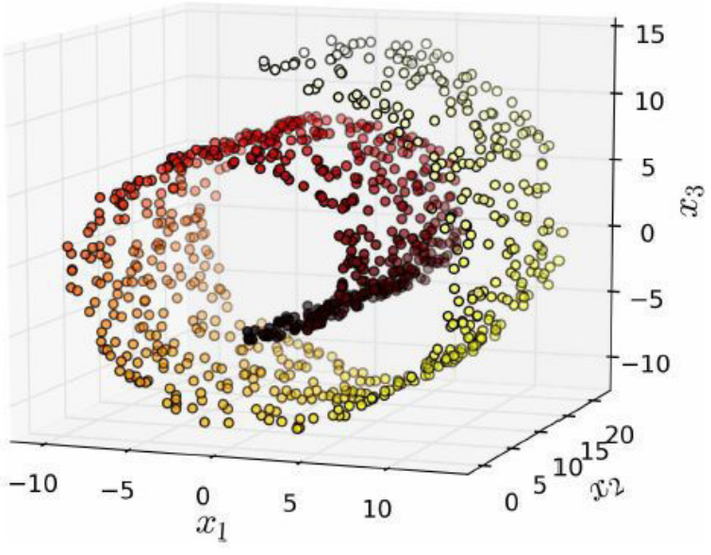
Dimensionality Reduction - Projection

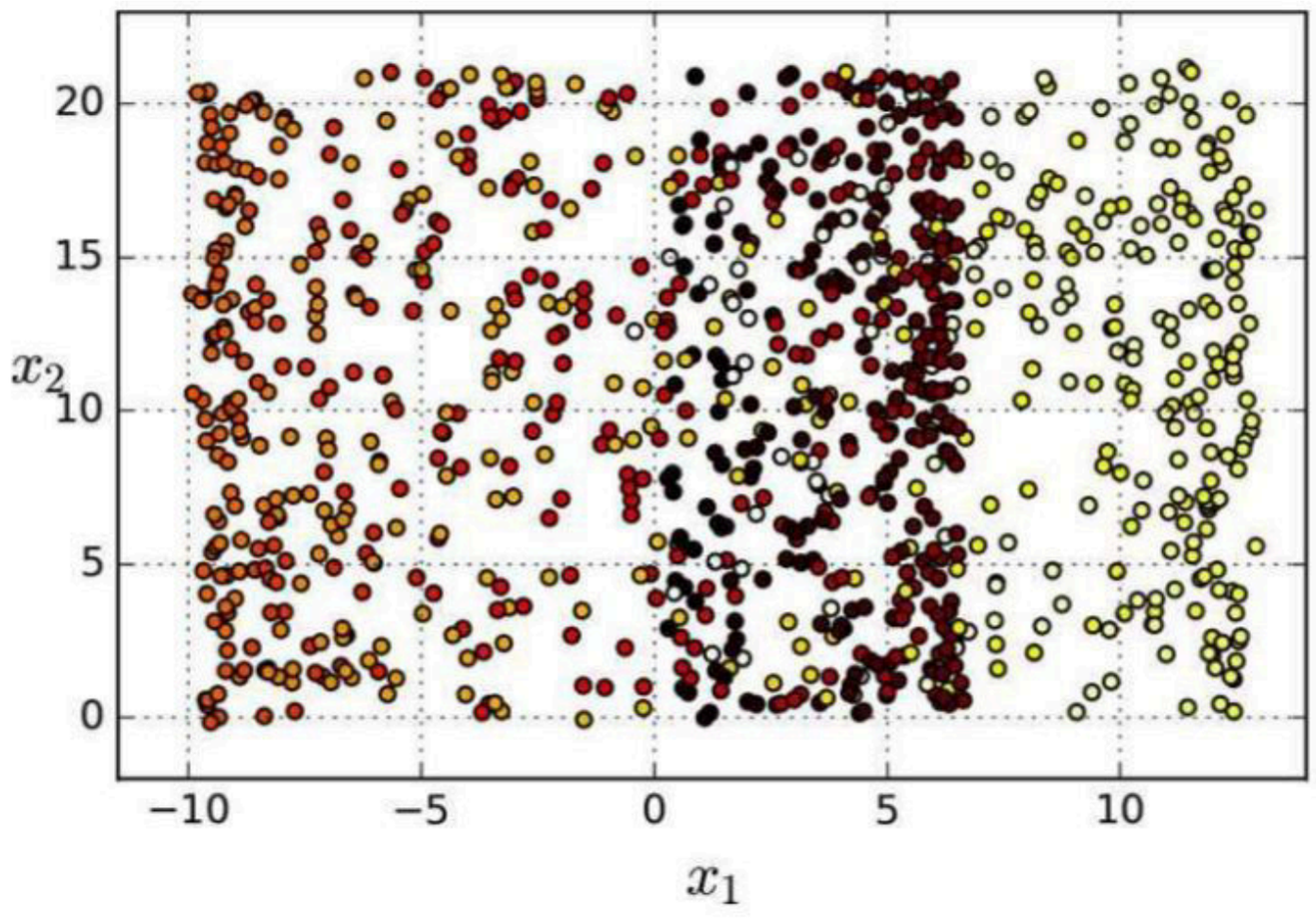
- Is projection always good?
  - Not really! Example: Swiss roll toy dataset



Dimensionality Reduction - Projection

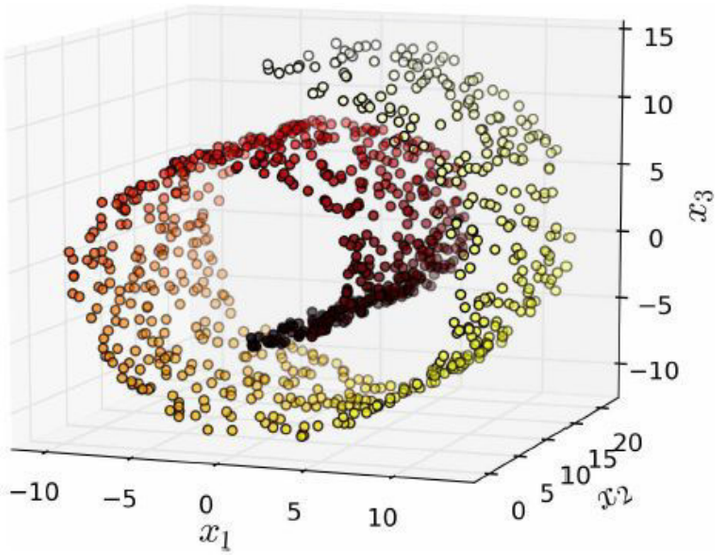
- Is projection always good?
  - Not really! Example: Swiss roll toy dataset
  - What if we project the training dataset onto x1 and x2.
  - The projection squashes the the different layers and hence classification is difficult

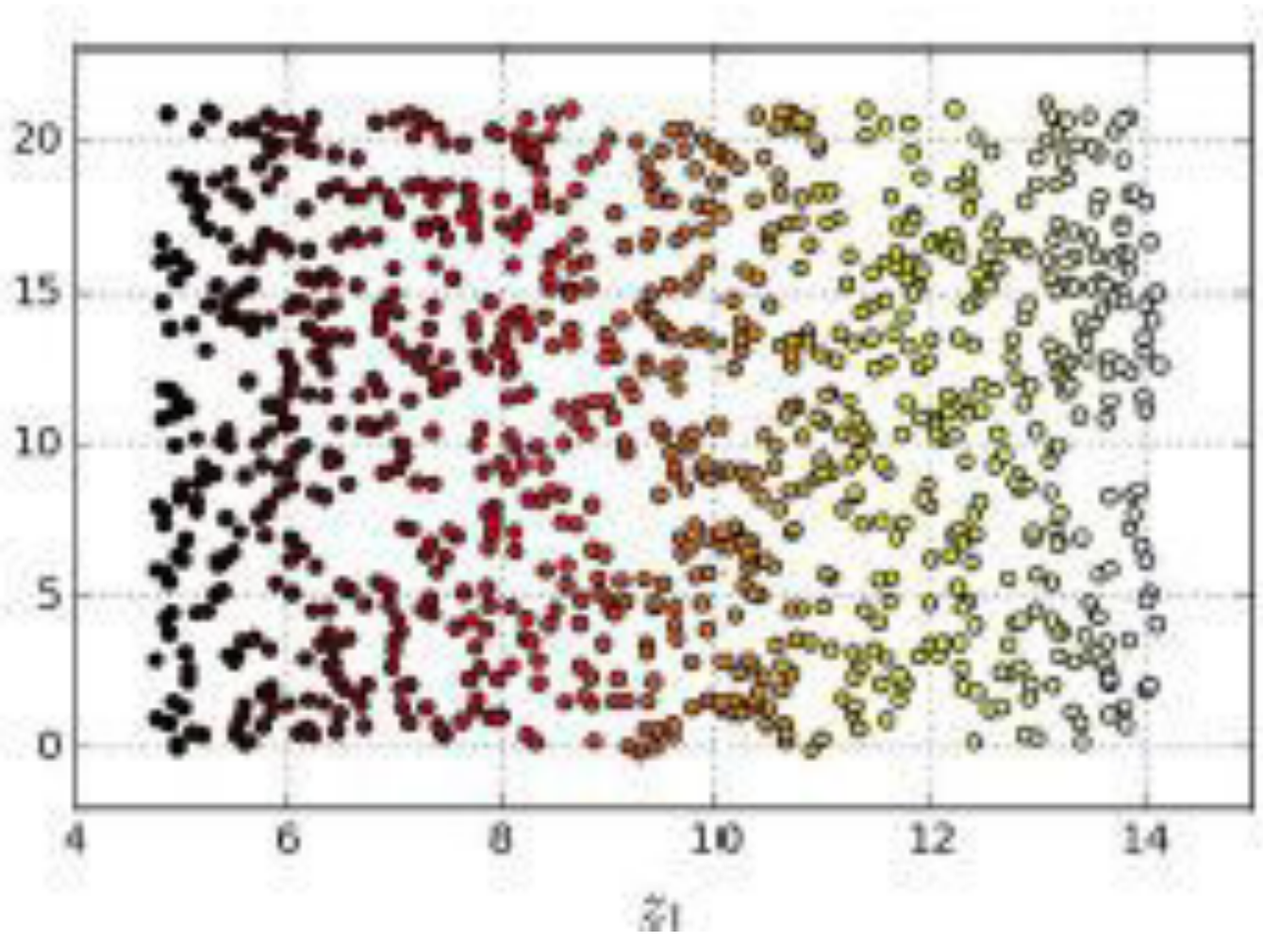




Dimensionality Reduction - Projection

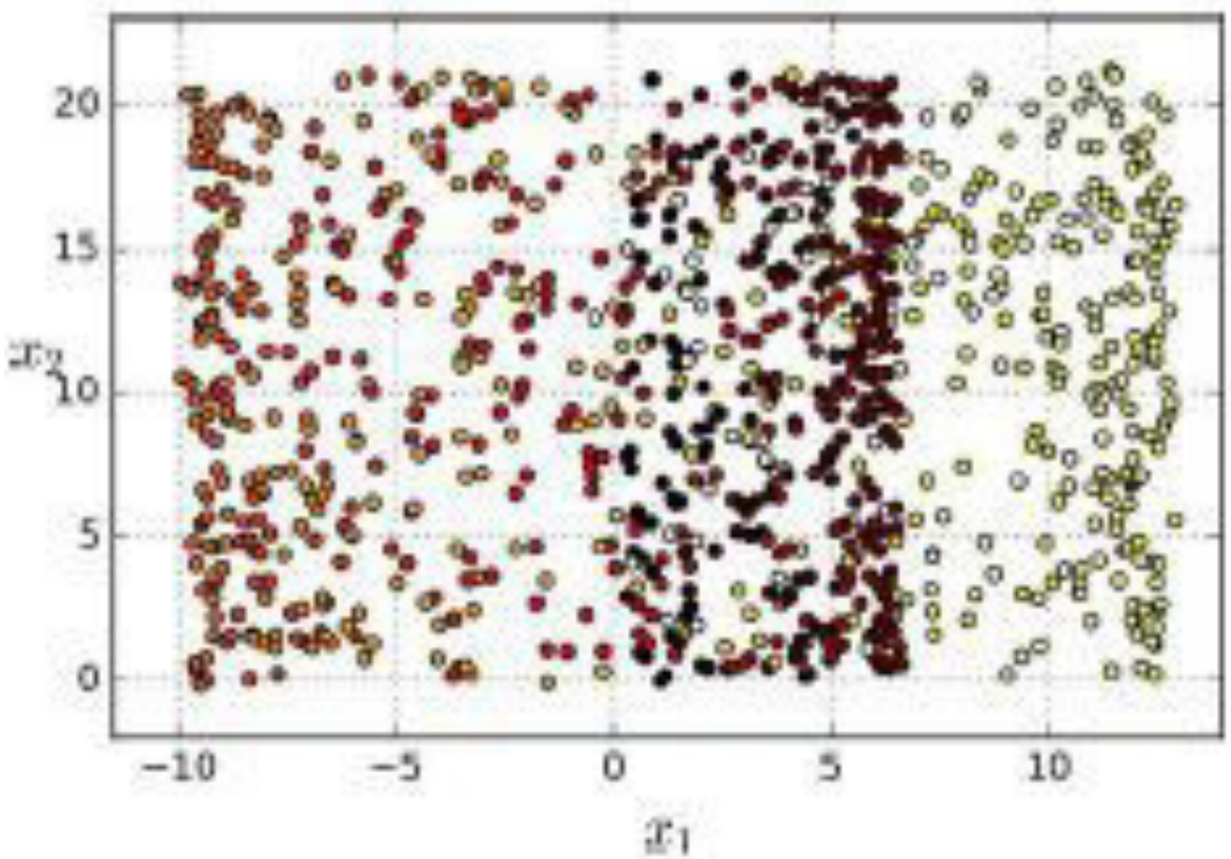
- What if we instead open the swiss roll?
  - Opening the swiss roll does not squash the different layers
  - The layers are classifiable.

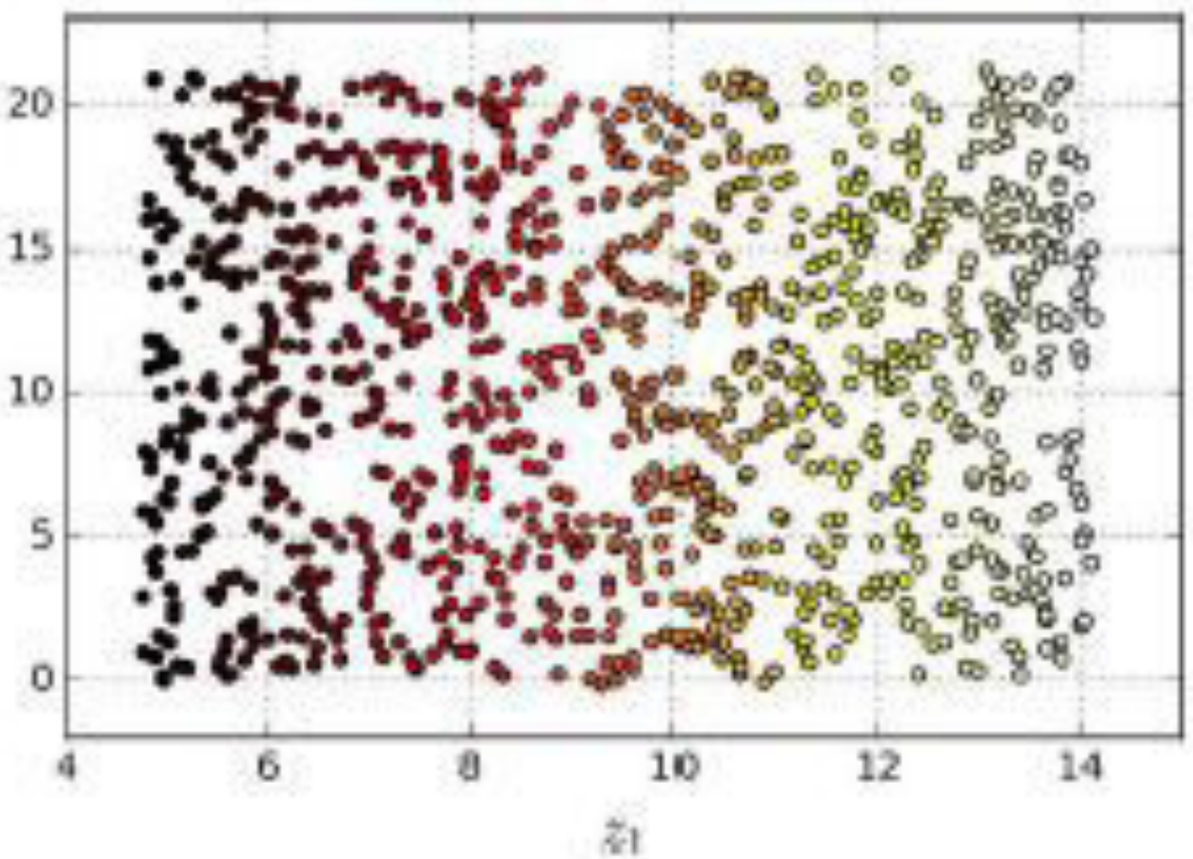




Dimensionality Reduction - Projection

- Projection does not seem to work in the case of swiss roll or similar datasets



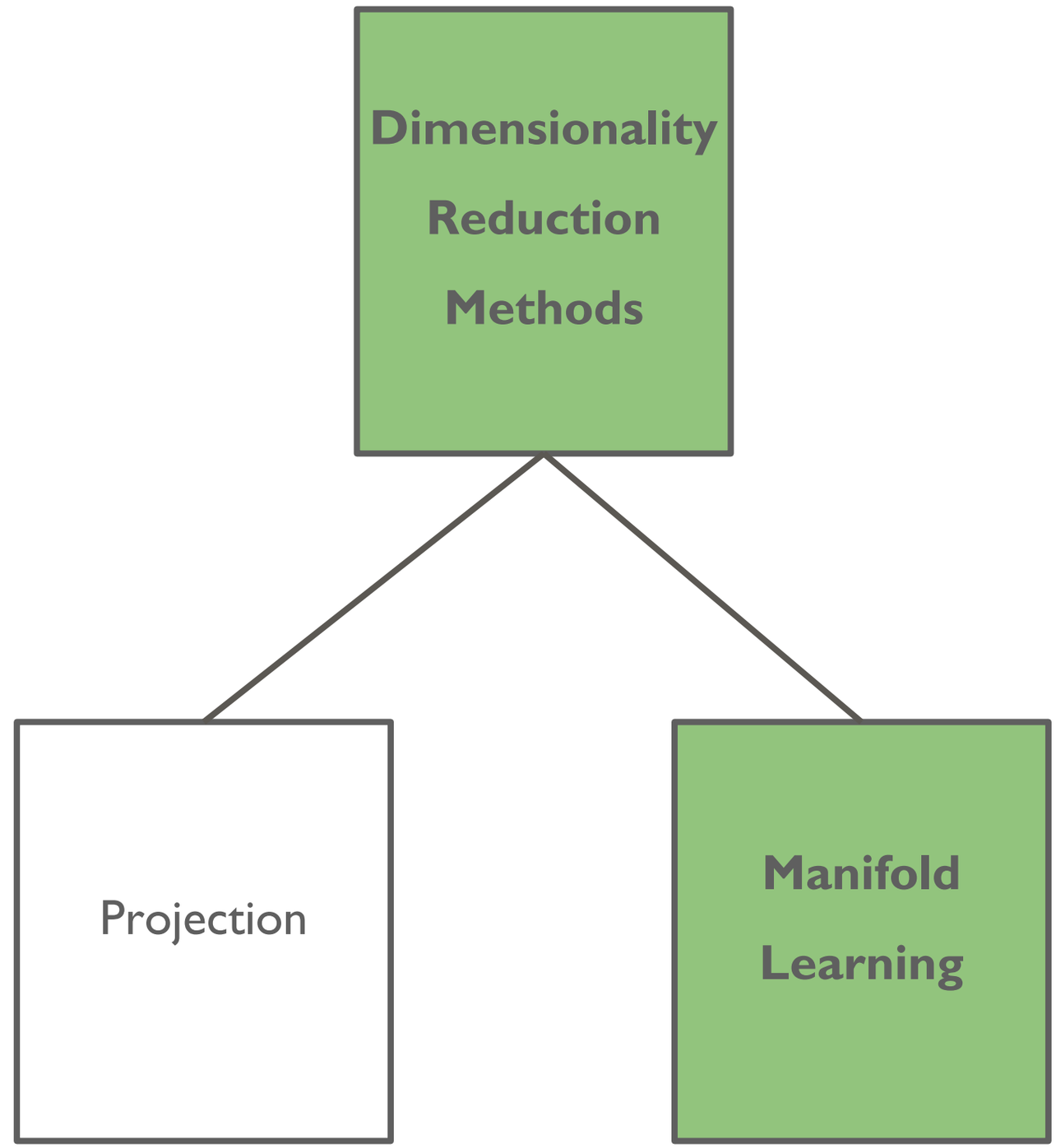


Dimensionality Reduction - Projection

- The above limitation of Projection can be demoed in the following steps:
  - Visualizing the swiss roll on a 3d plot
  - Projecting the swiss roll on the x1 and x2
    - Visualizing the squashed projection
  - Visualizing the rolled out plot

Switch to Notebook

Dimensionality Reduction

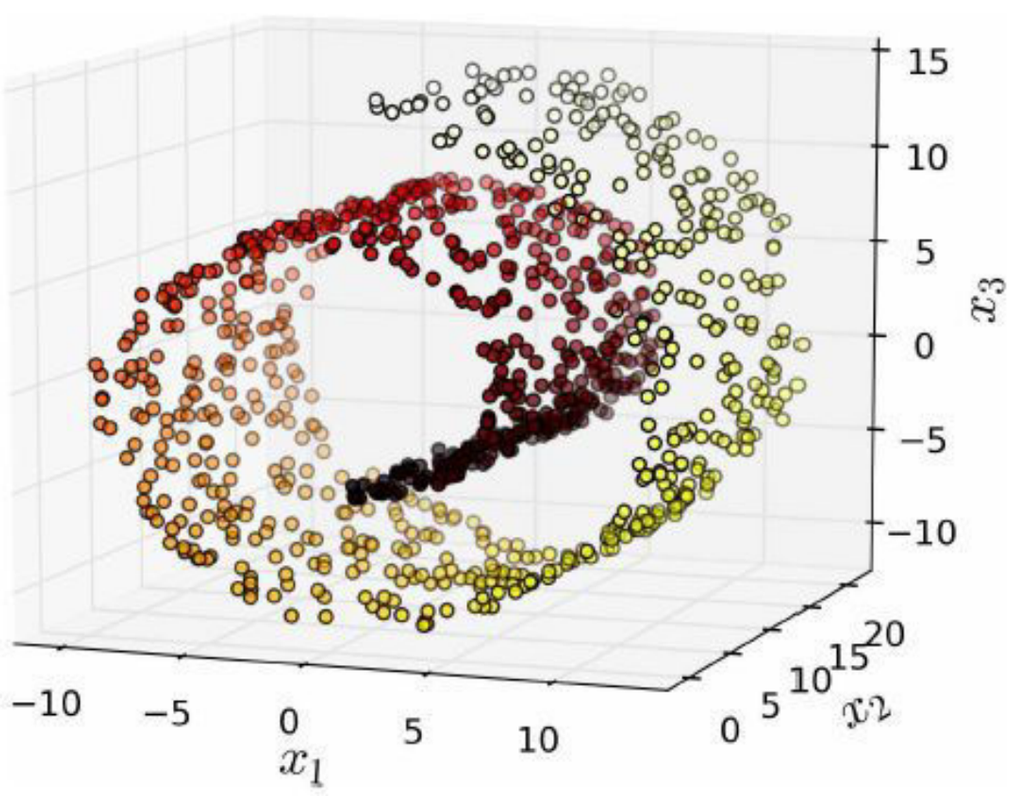


Dimensionality Reduction - Manifold Learning

Swiss roll is an example 2d manifold

- 2d manifold is a 2d shape that can be bent and twisted in a higher-dimensional space
- A d-dimensional space is a part of n-dimensional space (d<n)

Q. For swiss roll, d =? , n =?

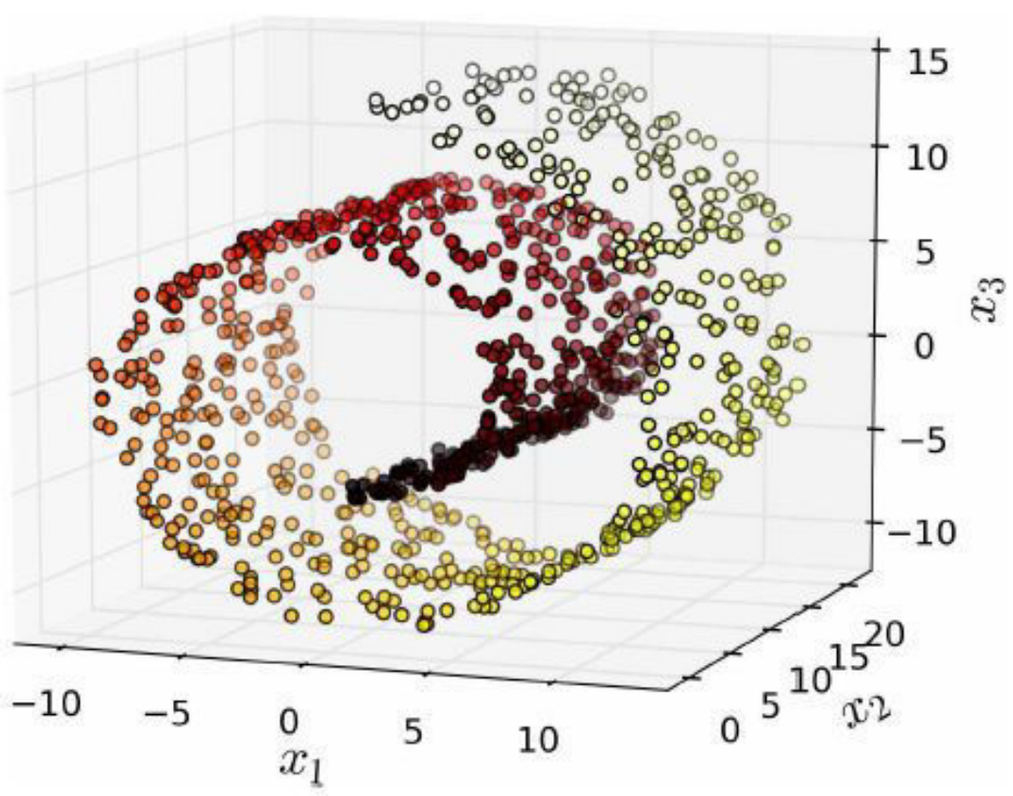


Dimensionality Reduction - Manifold Learning

Swiss roll is an example 2d manifold

- 2d manifold is a 2d shape that can be bent and twisted in a higher-dimensional space
- A d-dimensional space is a part of n-dimensional space (d<n)

Q. For swiss roll, **d =2 , n =3**



Dimensionality Reduction - Manifold Learning

- Many dimensionality reduction algorithms work by
  - modeling the manifold on which the training instances lie
- This is called manifold learning

So, for the swiss roll

- We can model the 2d plane
- Which is rolled in a swiss roll fashion
- Hence occupying a 3d space (like rolling of a paper)

Dimensionality Reduction - Manifold Learning

Manifold Learning

- Relies on manifold assumption, i.e.,
  - Most real-world high-dimensional datasets lie close to a much lower-dimensional manifold
- This is observed often empirically

Dimensionality Reduction - Manifold Learning

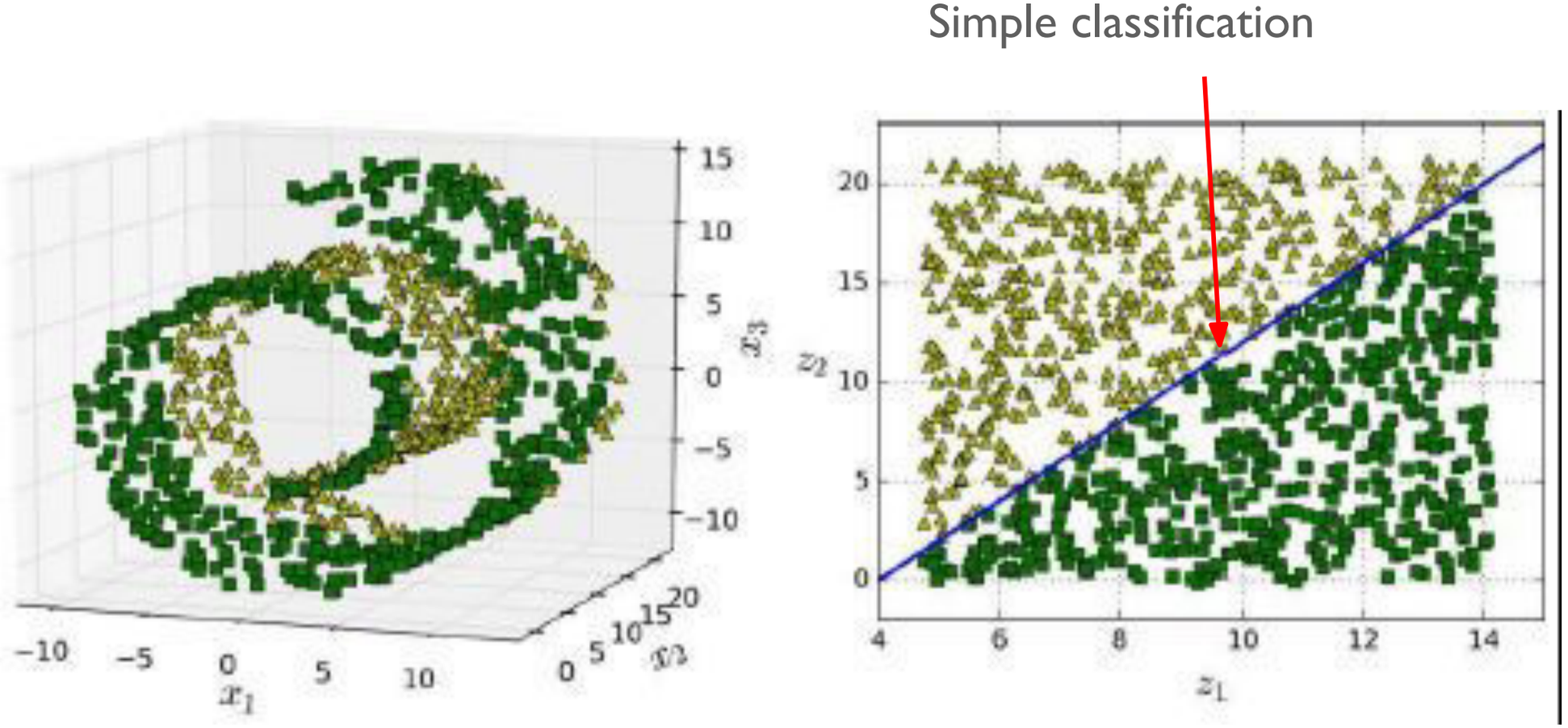
Manifold assumption is observed empirically in case of

- MNIST dataset where images of the digits have similarities:
  - Made of connected lines
  - Borders are white
  - More or less centered
- A randomly generated image would have much larger degree of freedom as compared to the images of digits
- Hence, the constraints in the MNIST images tend to squeeze the dataset into a lower-dimensional manifold.

Dimensionality Reduction - Manifold Learning

Manifold learning is accompanied by another assumption

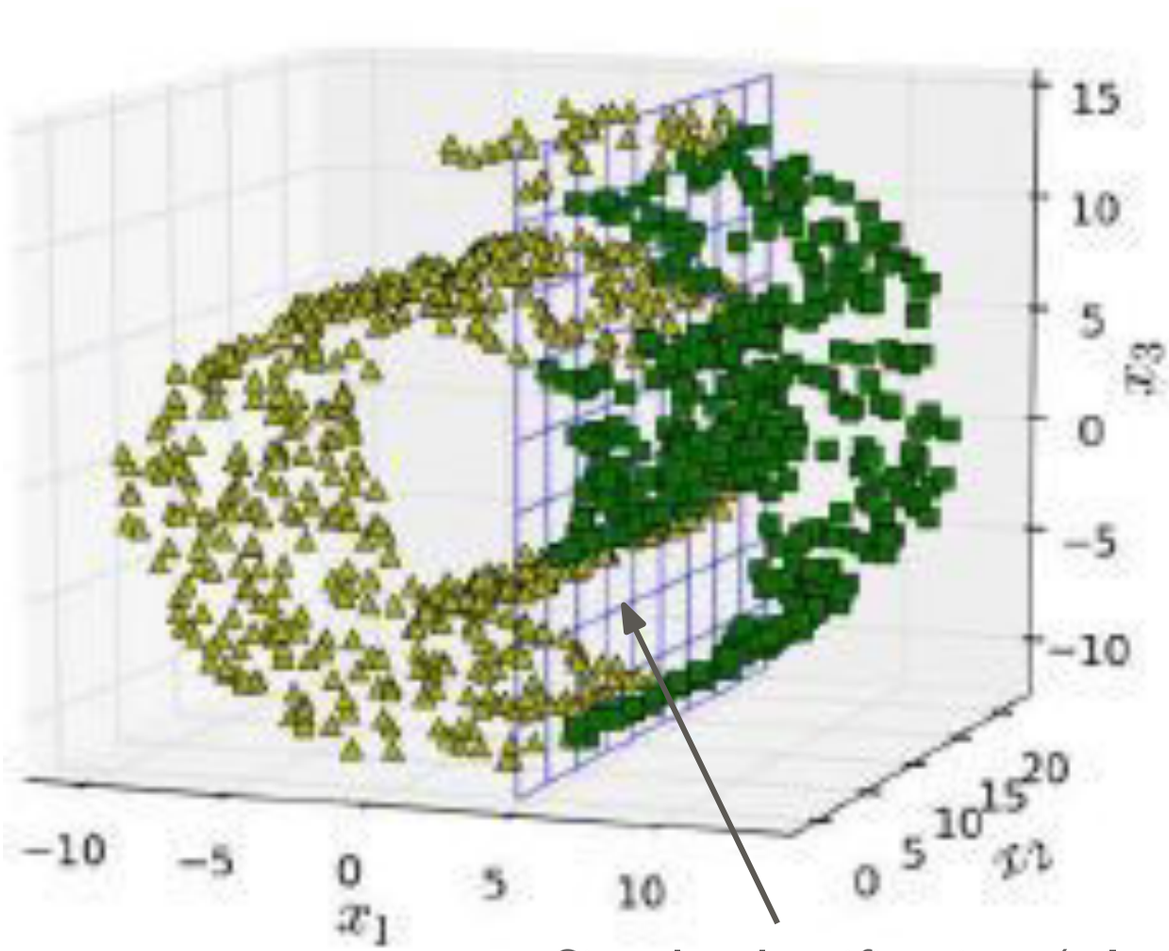
- Going to a lower-dimensional space shall make the task-at-hand simpler (holds true in below case)



Dimensionality Reduction - Manifold Learning

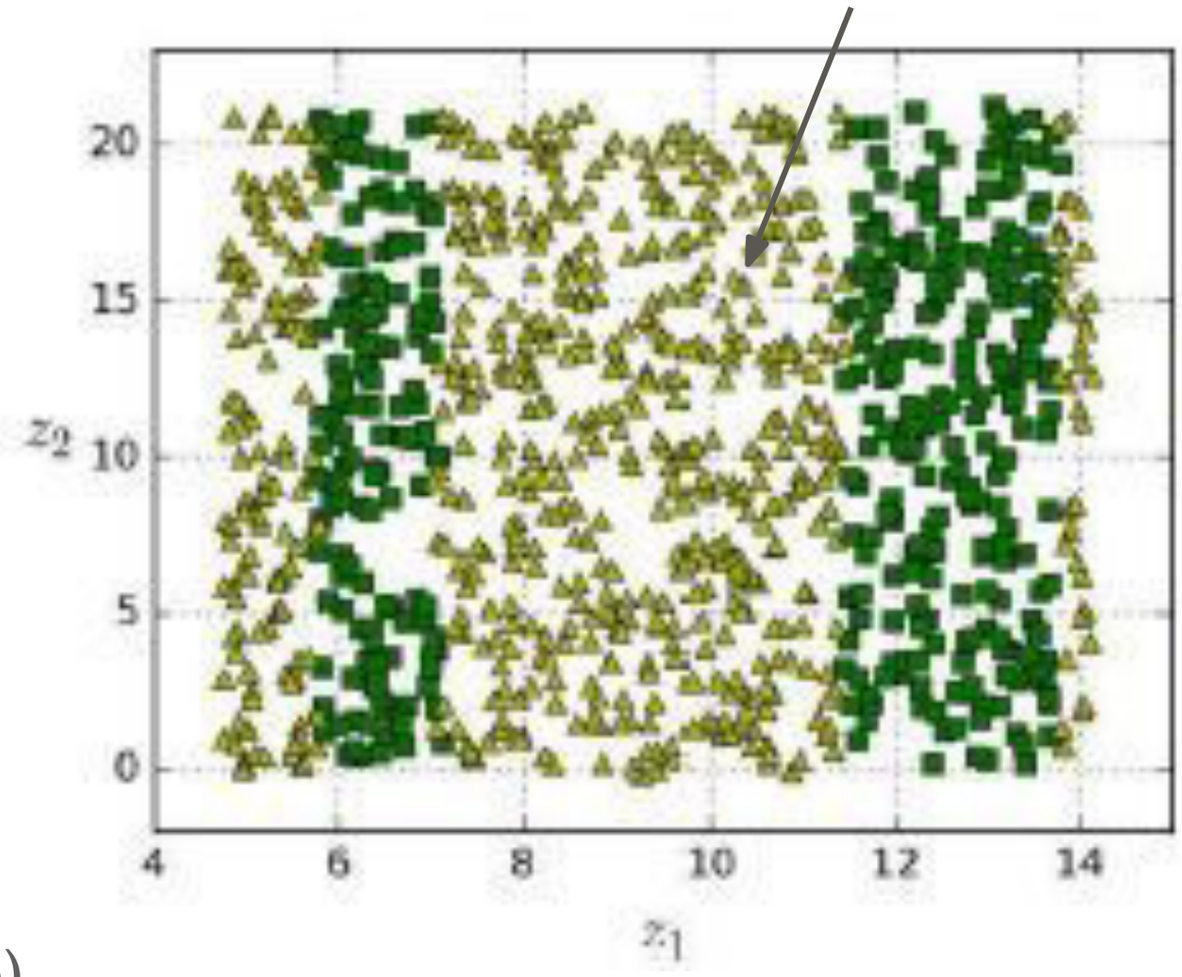
Manifold assumption accompanied by another assumption

- Going to a lower-dimensional space shall make the task-at-hand simpler (Not always the case)



Simple classification (x1=5)

Fairly complex classification



Dimensionality Reduction - Manifold Learning

The previous 2 cases can be demonstrated in these steps:

- Using the 3d swiss roll dataset
- Plotting the case where the classification gets easier with manifold
- Plotting the case where the classification gets difficult with manifold
- Plotting the decision boundary in each case

Switch to Notebook

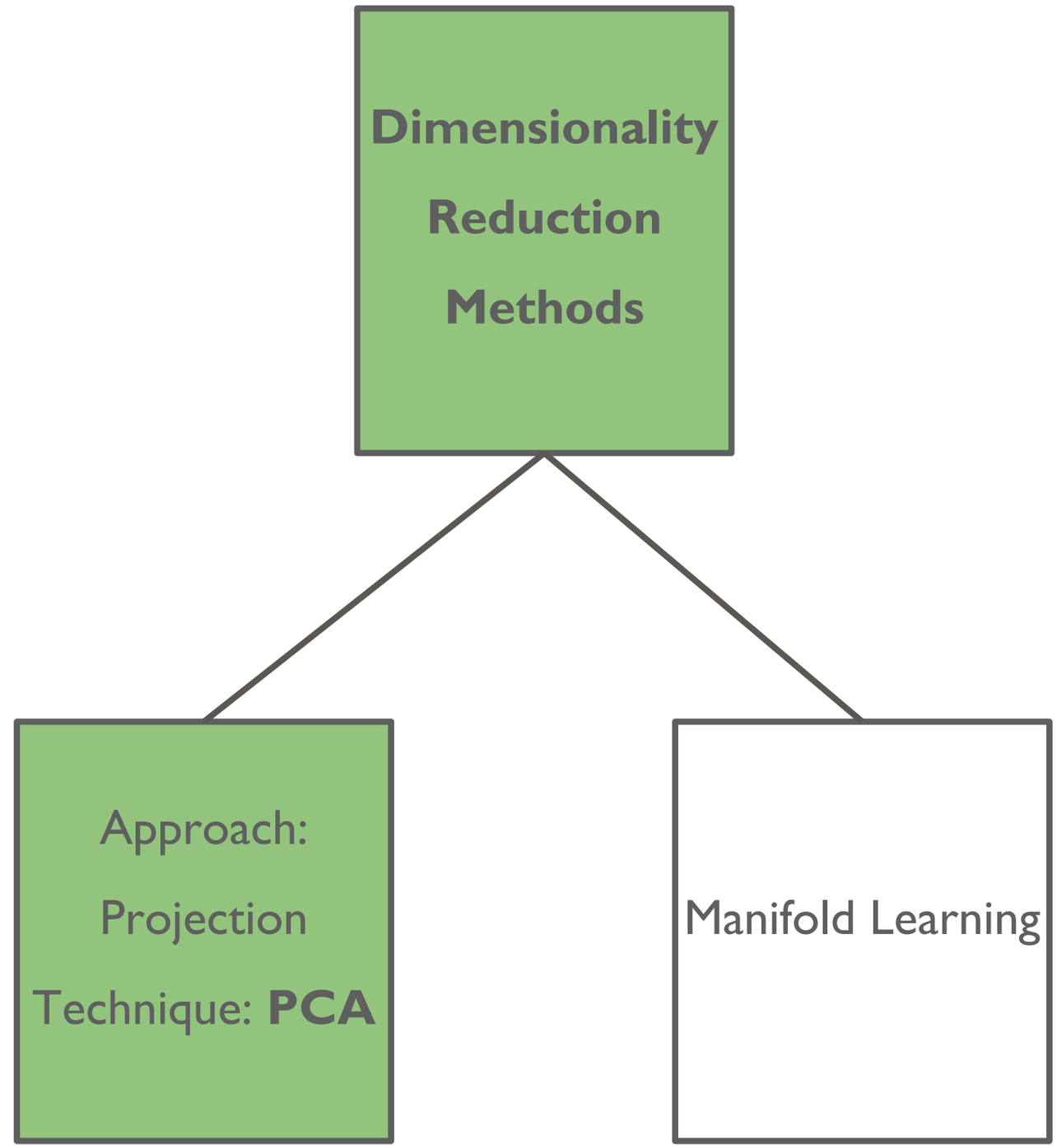
Dimensionality Reduction

Summary - Dimensionality Reduction

- 2 approaches: Projection and Manifold Learning
  - Depends on the dataset, which should be used

- Leads to better **visualization**
- **Faster** training
- May not always lead to a better or simpler or better solution
  - Valid both for projection or manifold learning
  - Depends on the dataset
- Lossy
  - Should always try with the original dataset before going for dimensionality reduction

Dimensionality Reduction



Principal Component Analysis (PCA)

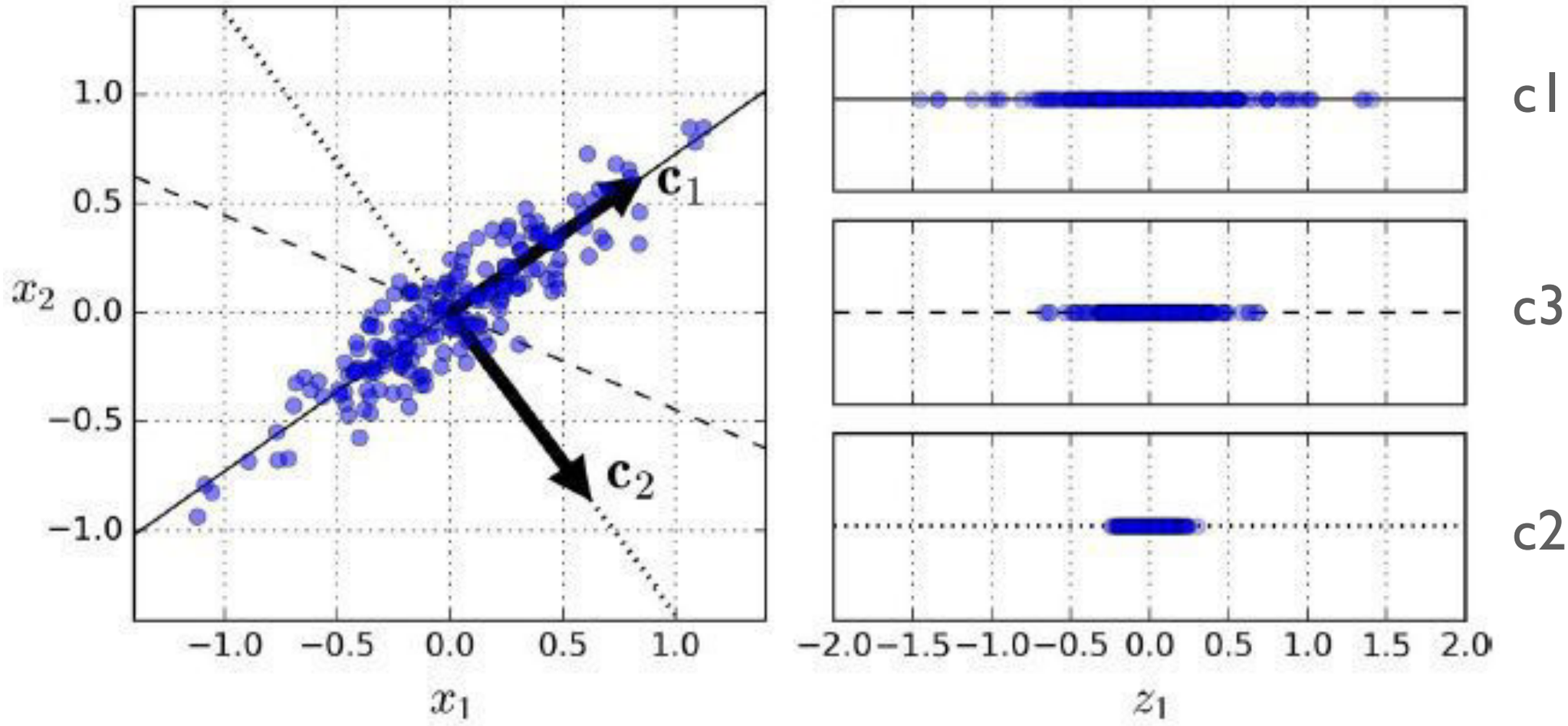
- The most popular dimensionality reduction algorithm
- Identify the hyperplane that lies closest to the data
- Projects the data onto the hyperplane

PCA- Preserving the variance

How do we select the best hyperplane to project the datasets into?

- Select the axis that preserves the maximum amount of variance
- Lose less information than other projections

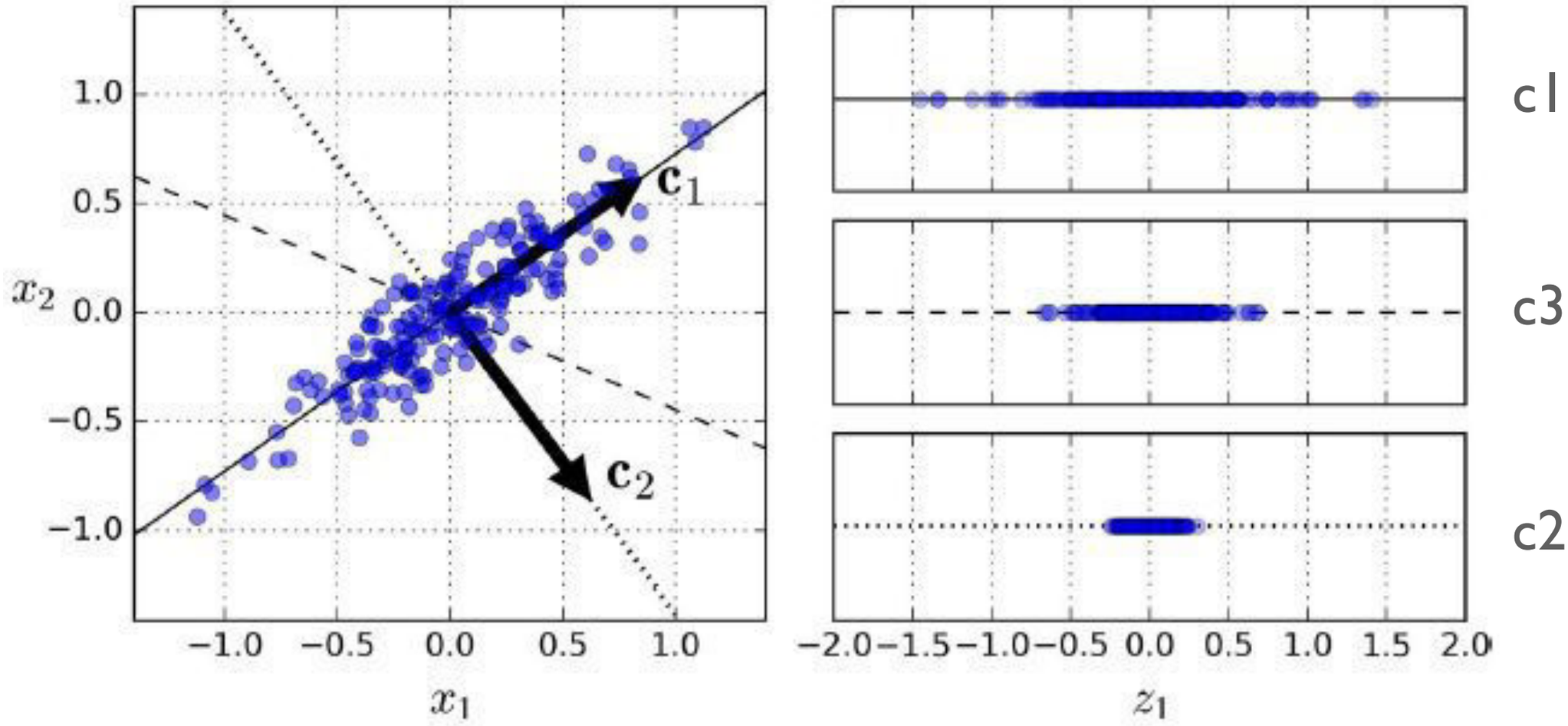
PCA- Preserving the variance



Q. Which of these is the best axes to select (preserves maximum variance)?

c1 or c2 or c3?

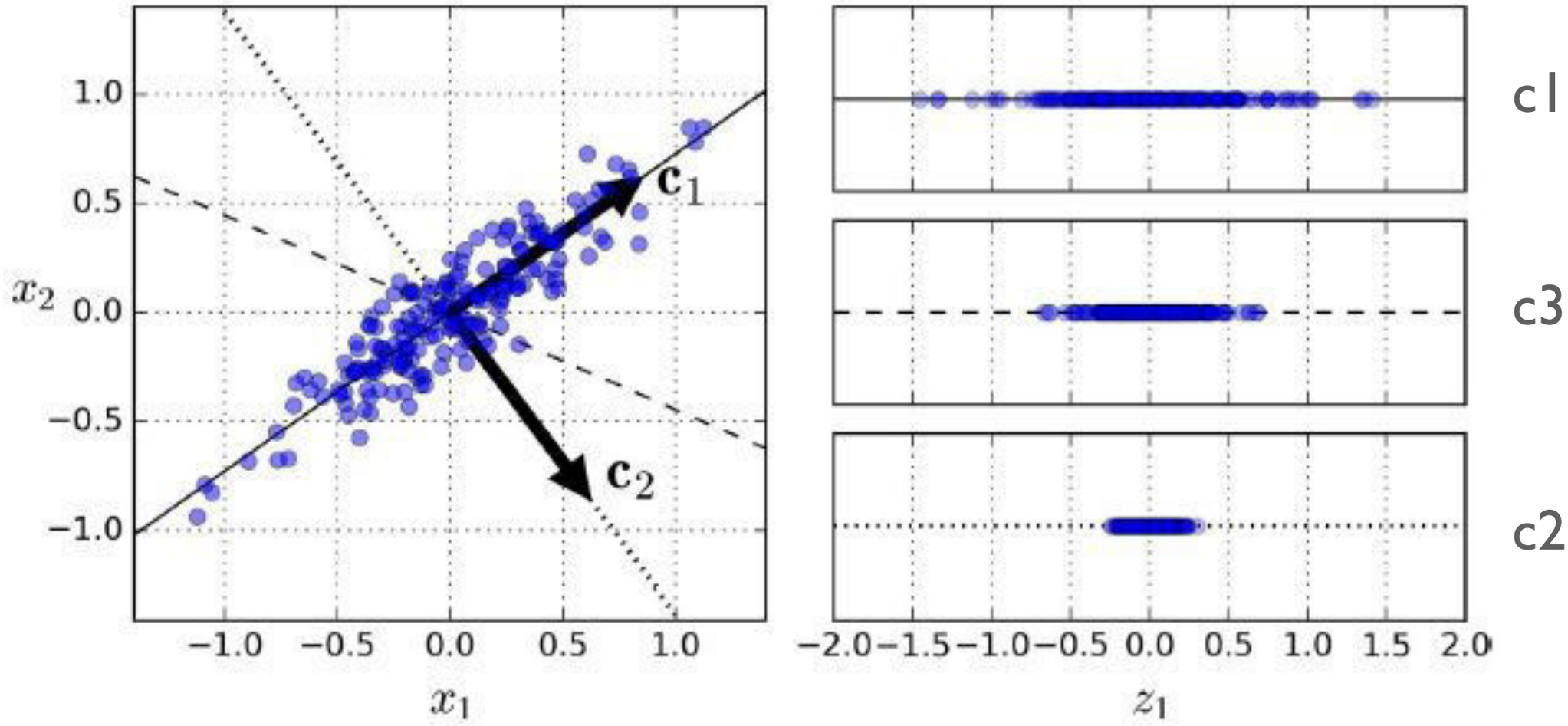
PCA- Preserving the variance



Q. Which of these is the best axes to select? **Ans: c1.**

- Preserves maximum variance as compared to other axes.

PCA- Preserving the variance



Another way to say the axis that minimizes the mean squared distance between the original dataset and its projection onto that axis.

Dimensionality Reduction - Manifold Learning

The previous case can be demonstrated in these steps:

- Generate a random 2d dataset
- Stretch it along a particular direction
- Project it along certain 3 axis
- Plot the stretched random numbers, the projections along the axes

Switch to Notebook

PCA- Principal Components

How do we select the best hyperplane to project the datasets into?

Ans: PCA

- identifies the axis that accounts for the largest amount of variance in the training set - 1st principal component
- Provides a second axis orthogonal to the first one that accounts for second largest
- And so on.. Third axis, fourth axis..

PCA- Principal Components

The unit vector that defines that ‘i’th axis is called the ‘i’th principal component (PC)

- 1st PC = c1
- 2nd PC = c2
- 3rd PC = c3

C1 is orthogonal to c2, c3 would be orthogonal to the plane formed by c1 and c2,
And hence orthogonal to both c1 and c2.

Image in 3d space for a minute!

PCA- Principal Components

Next Ques: How do we find the principal components?

- Standard factorization technique called Singular Value Decomposition (SVD) - based on eigen value calculation!
- It divides the training dataset into the dot product of 3 matrices
  - U
  - ∑
  - transpose(V)

$$U \cdot \Sigma \cdot V^T$$

- Transpose(V) contains the principal components (PC) - unit vectors

PCA- Principal Components

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

Transpose(V) contains the principal components (PC) - unit vectors

- 1st PC = c1
- 2nd PC = c2
- 3rd PC = c3
- ...

PCA- Principal Components - SVD

SVD can implemented in scikit-learn using the code below

- SVD assumes that the data is centered around the origin

```
# Data needs to centralized before performing SVD
>>> X_centered = X - X.mean(axis=0)
# Performing SVD
>>> U,s,V = np.linalg.svd(X_centered)

# Printing the principal components
>>> c1, c2 = V.T[:,0], V.T[:,1]
print(c1,c2)
```

Q. How many principal components are we printing in the above code?

PCA- Principal Components - SVD

SVD can implemented in scikit-learn using the code below

- SVD assumes that the data is centered around the origin

```
# Data needs to centralized before performing SVD
>>> X_centered = X - X.mean(axis=0)
# Performing SVD
>>> U,s,V = np.linalg.svd(X_centered)

# Printing the principal components
>>> c1, c2 = V.T[:,0], V.T[:,1]
print(c1,c2)
```

Q. How many principal components are we printing in the above code?

Ans: 2

PCA- Projecting down to d dimensions

Once, the PCs have been found, original dataset has to be projected on the PCs

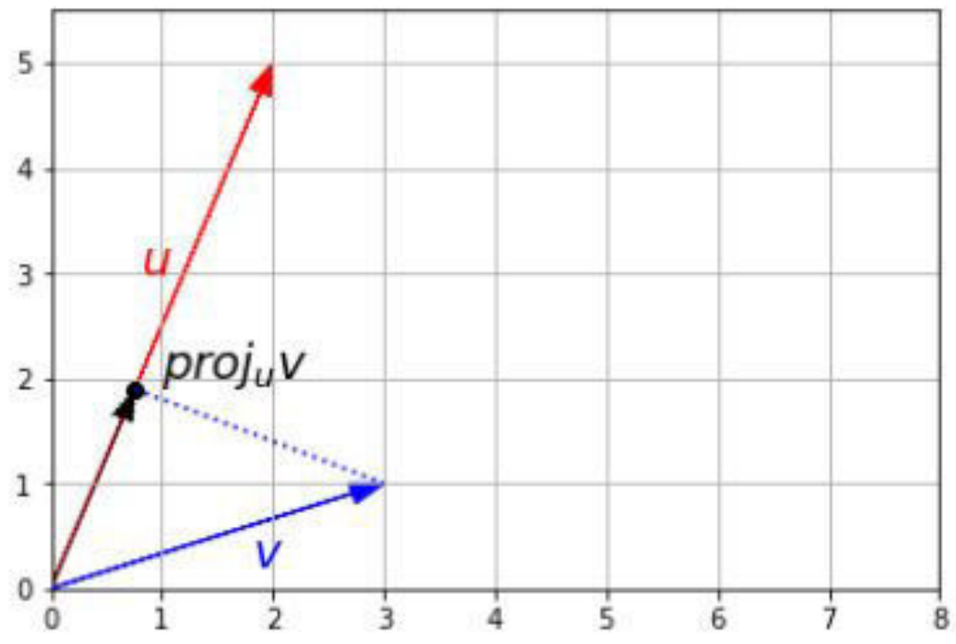
As we have seen in linear algebra session,

- A vector v can be projected onto
- another vector u
- By doing a dot product of v and u.

$$\mathbf{proj_u v} = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|^2} \times \mathbf{u}$$

Which is equivalent to:

$$\mathbf{proj_u v} = (\mathbf{v} \cdot \hat{\mathbf{u}}) \times \hat{\mathbf{u}}$$



PCA- Projecting down to d dimensions

Similarly, the

- original training dataset X can be projected onto
- the first ‘d’ principal components Wd
  - Composed of first ‘d’ columns of transpose(V) obtained in SVD
- Reducing the dataset dimensions to ‘d’

$$X_{d\text{-}proj} = X.W_d$$

Wd = first d columns of transpose(V) containing the first d principal components

PCA- Projecting down to d dimensions

Similarly, the

- original training dataset X can be projected onto
- the first ‘d’ principal components Wd
  - Composed of first ‘d’ columns of transpose(V) obtained in SVD
- Reducing the dataset dimensions to ‘d’

First ‘d’ columns of the transpose(V)

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

PCA- SVD and PCA

So, PCA involves two steps

- SVD and
- Projection of the training dataset onto the orthogonal principal components

Scikit-Learn provides functions for both

- SVD, projection and
- Combined PCA

We will be comparing the codes for these

PCA- SVD and PCA

PCA using SVD in Sci-kit Learn	PCA using Sci-kit Learn PCA function

`# Centering the data and doing SVD`
`X_centered = X - X.mean(axis=0)`
 	
`U,s,V = np.linalg.svd(X_centered)`
 	
`# Extracting the components and projecting the`
`# original dataset`
 	
`W2 = V.T[:, :2]`
`X2D = X_centered.dot(W2)`
	`from sklearn.decomposition import PCA`
	
	`# Directly doing PCA and transforming the`
	`original dataset`
	`# Takes care of centering`
	
	`pca = PCA(n_components = 2)`
	`X2D = pca.fit_transform(X)`

Switch to Notebook

PCA- Explained Variance Ratio

Variances explained by each of the components is important

- We would like to cover as much variance as in the original dataset
- available via the explained_variance_ratio_ variable

```
>>> print(pca.explained_variance_ratio_)
[ 0.95369864  0.04630136]
```

1st component covers 95.3 % of the variance

2nd component covers 4.6 % of the variance

PCA- Number of PCs

How to select the number of principal components

- The principal components should explain 95% of the variance in original dataset
- For visualization, it has to be reduced to 2 or 3

```
Calculating the variance explained in Scikit-Learn
>>> pca = PCA()
>>> pca.fit(X)
>>> cumsum = np.cumsum(pca.explained_variance_ratio_)

# Calculating the number of dimensions which explain 95% of variance
>>> d = np.argmax(cumsum >= 0.95) + 1
2
```

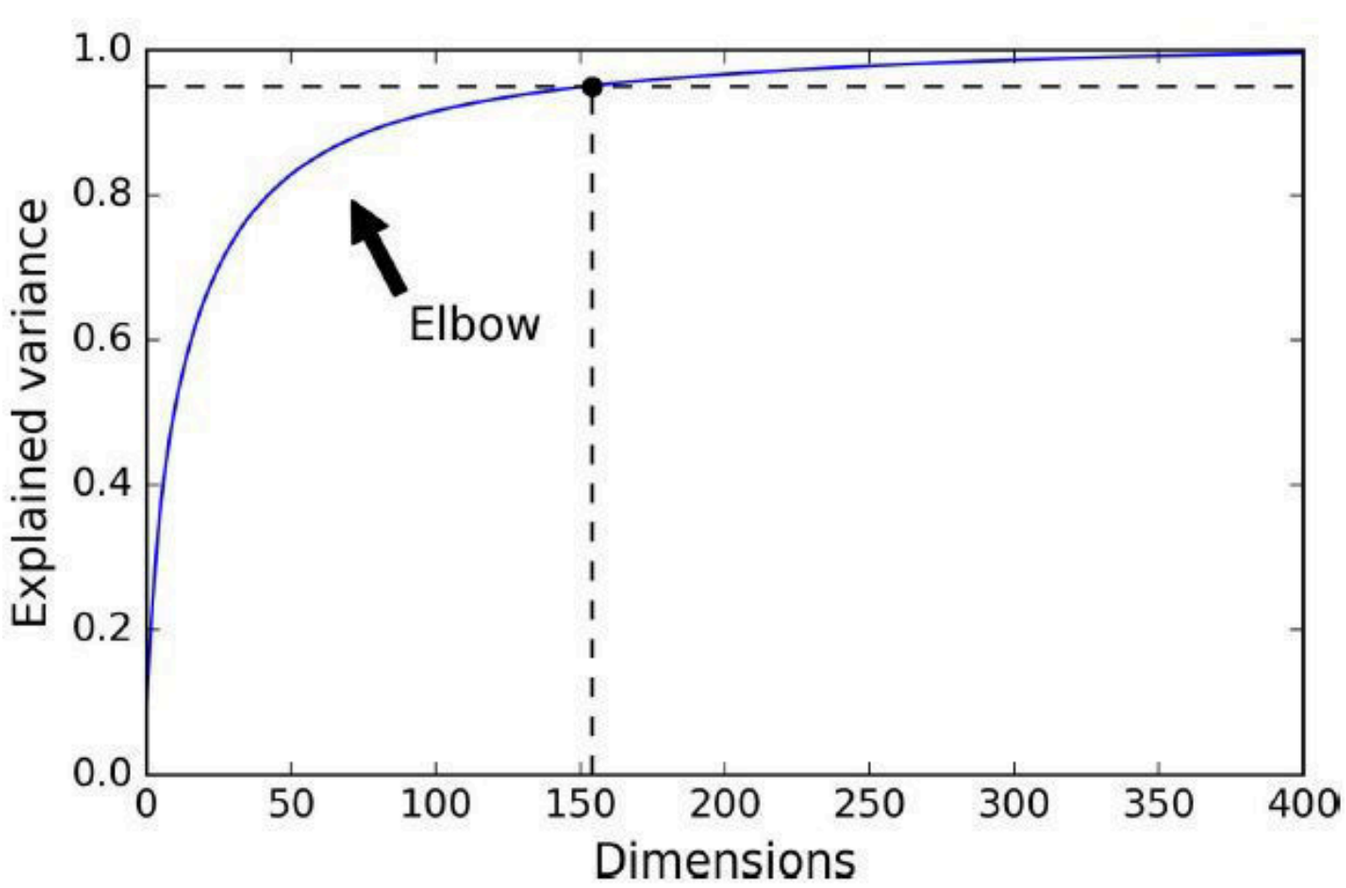
PCA- Number of PCs

```
#  Calculating the PCs directly specifying the variance to be
explained

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

PCA- Number of PCs

Another option is to plot the explained variance

- As a function of the number of dimensions
- Elbow curve: explained variance stops growing fast after certain number of dimensions



Dimensionality Reduction - Projection

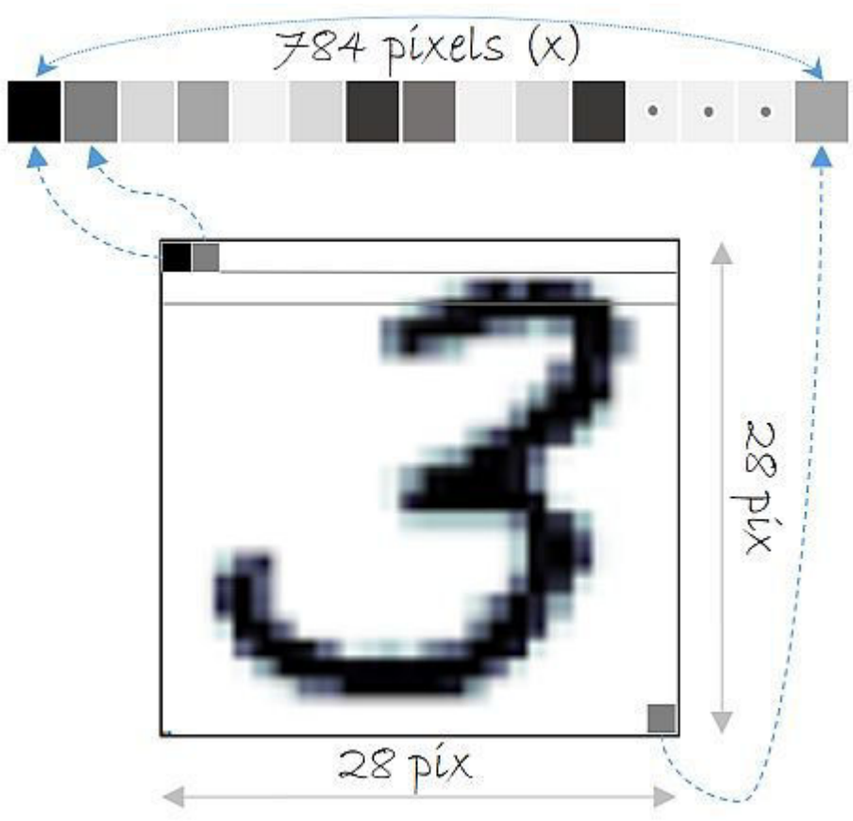
- For the above 2d dataset, we shall demonstrate
  - Calculating the estimated variance ratio
  - Calculating the number of principal components

Switch to Notebook

PCA- Compression of dataset

Another aspect of dimensionality reduction,

- the training set takes up much less space.
- For example, applying PCA to MNIST dataset



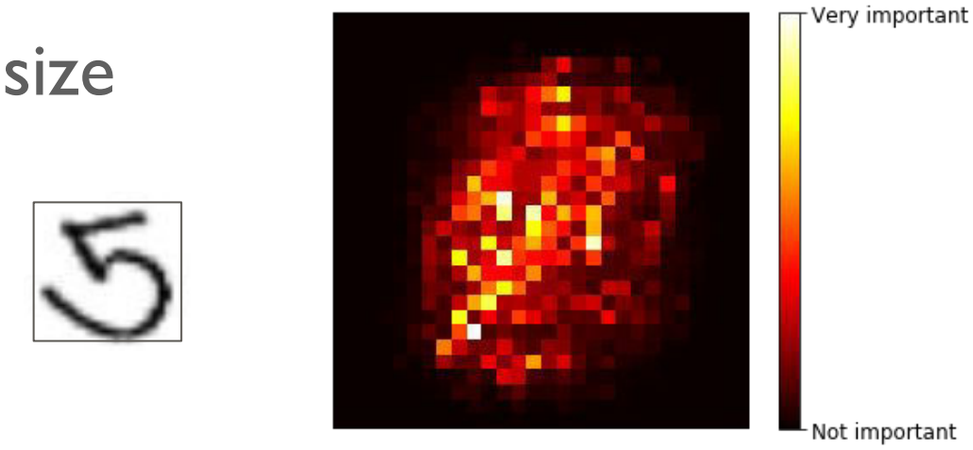
- **ORIGINAL:** Each image
  - 28 X 28 pixels
  - 784 features
  - Each pixel is either on or off 0 or 1

PCA- Compression of dataset

After applying PCA to the MNIST data

- Number of dimensions reduces to 154 features from 784 features
- Keeping 95% of its variance

Hence, the training set is 20% of its original size




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

Number of features required to explain 95% variance

PCA- Compression of dataset - Demo

Loading the MNIST Dataset

```
#MNIST compression:
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.datasets import fetch_mldata

>>> mnist = fetch_mldata('MNIST original')

>>> X, y = mnist["data"], mnist["target"]
>>> X_train, X_test, y_train, y_test = train_test_split(X, y)
>>> X = X_train
```

PCA- Compression of dataset

Applying PCA to the MNIST dataset

```
# Applying PCA to the MNIST Dataset
>>> pca = PCA()
>>> pca.fit(X)
>>> d = np.argmax(np.cumsum(pca.explained_variance_ratio_) >= 0.95) + 1
154
```

```
# Projecting onto the principal components
>>> pca = PCA(n_components=0.95)
>>> X_reduced = pca.fit_transform(X)
>>> pca.n_components_
154

# Checking for the variance explained
# did we hit the 95% minimum?
>>> np.sum(pca.explained_variance_ratio_)
0.9503623084769206
```

Switch to Notebook

PCA- Decompression

The compressed dataset can be decompressed to the original size

- For MNIST dataset, the reduced dataset (154 features)
- Back to 784 features
- Using inverse transformation of the PCA projection

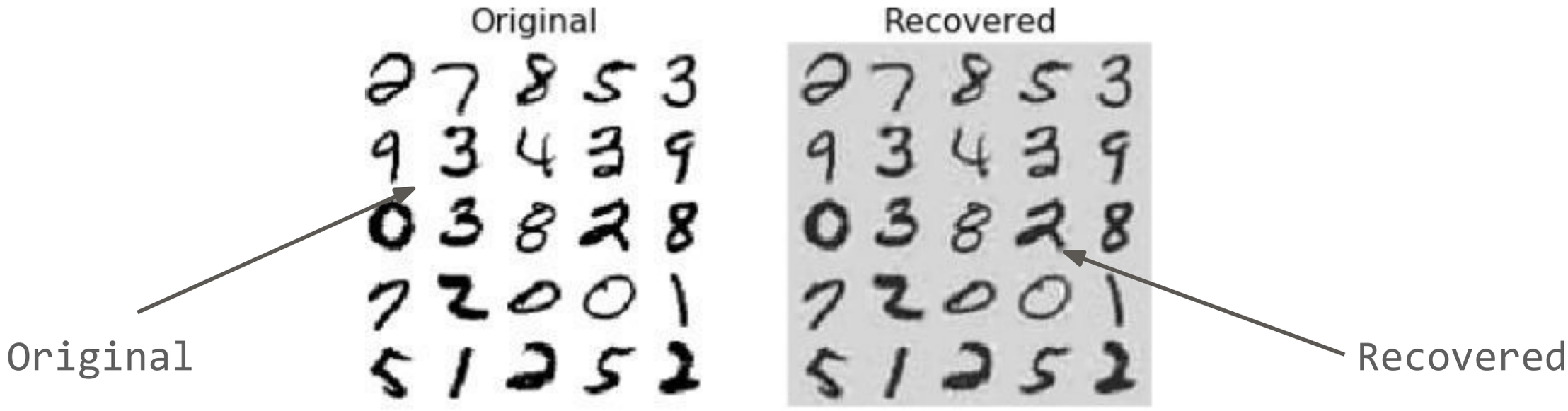
$$\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d\text{-proj}} \cdot \mathbf{W}_d^T$$

```
# use inverse_transform to decompress back to 784 dimensions
>>> X_mnist = X_train

>>> pca = PCA(n_components = 154)
>>> X_mnist_reduced = pca.fit_transform(X_mnist)
>>> X_mnist_recovered = pca.inverse_transform(X_mnist_reduced)
```

PCA- Decompression

Plotting the recovered digits

- Recovered digits has lost some information
- Dimensionality reduction captured only 95% of variance
- It is called reconstruction error



Switch to Notebook

PCA- Incremental PCA

Problem with PCA (Batch-PCA)

- Requires the entire training dataset in-the-memory to run SVD

Incremental PCA (IPCA)

- Splits the training set into mini-batches
- Feeds one mini-batch at a time to the IPCA algorithm
- Useful for large datasets and online learning

PCA- Incremental PCA

Incremental PCA using Scikit Learn's IncrementalPCA class

- And associated partial_fit() function instead of fit() and fit_transform()

```
# split MNIST into 100 mini-batches using Numpy array_split()
# reduce MNIST down to 154 dimensions as before.
# note use of partial_fit() for each batch.
```

```
>>> 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):
       print(".", end="")
        inc_pca.partial_fit(X_batch)

>>> X_mnist_reduced_inc = inc_pca.transform(X_mnist)
```

PCA- Incremental PCA

Another way is to use Numpy memap class

- Uses binary array on the disk as if it was in-memory

```
# alternative: Numpy memmap class (use binary array on disk as if it was in memory)

>>> filename = "my_mnist.data"
>>> X_mm = np.memmap(
    filename, dtype='float32', mode='write', shape=X_mnist.shape)

>>> X_mm[:] = X_mnist
>>> del X_mm

>>> X_mm = np.memmap(filename, dtype='float32', mode='readonly', shape=X_mnist.shape)
>>> batch_size = len(X_mnist) // n_batches
>>> inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
>>> inc_pca.fit(X_mm)
```

Switch to Notebook

PCA- Randomized PCA

Using a stochastic algorithm

- To approximate the first d principal components
- O(m × d^2) + O(d^3), instead of O(m × n^2) + O(n^3)
- Dramatically faster than (Batch) PCA and Incremental PCA
  - When d << n

```
>>> rnd_pca = PCA(n_components=154, svd_solver="randomized")

>>> t1 = time.time()
>>> X_reduced = rnd_pca.fit_transform(X_mnist)
>>> t2 = time.time()
>>> print(t2-t1, "seconds")
4.414088487625122 seconds
```

Switch to Notebook

Kernel PCA

Using Kernel PCA

- Kernel trick can also be applied to PCA
- Makes nonlinear projections possible for dimensionality reduction
- This is called Kernel PCA (kPCA)

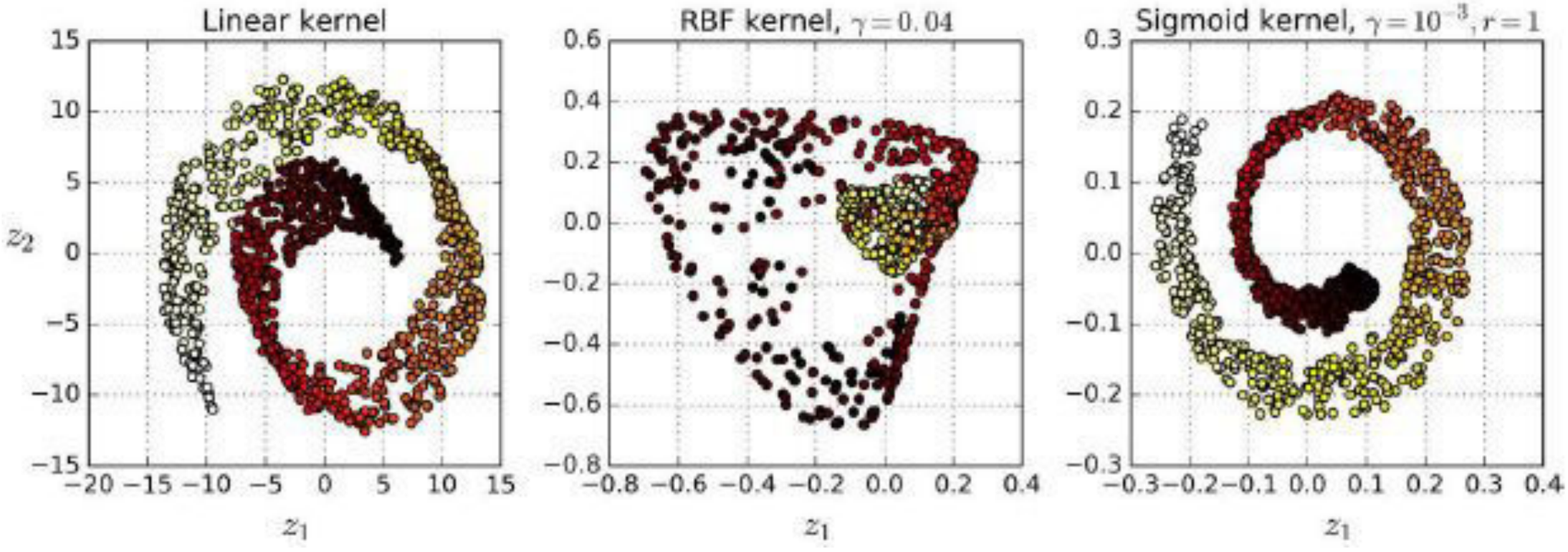
Important point about Kernel PCA we should remember is:

- Good at preserving clusters
- Useful when unrolling datasets that lies close to a twisted manifold

Kernel PCA

Kernel PCA in Scikit-Learn using KernelPCA class

- Linear Kernel
- RBF Kernel
- Sigmoid Kernel



Kernel PCA

Kernel PCA in Scikit-Learn using KernelPCA class

```
>>> from sklearn.decomposition import KernelPCA
>>> rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
>>> X_reduced = rbf_pca.fit_transform(X)
```

Switch to Notebook

Kernel PCA - Selecting hyperparameters

Selecting hyper parameters

- Kernel PCA is an unsupervised learning algorithm
- No obvious performance measure to help select the best kernel and hyperparameters

Instead, we can follow these steps:

- Create a pipeline with KernelPCA and Classification model
- Do a grid search using GridSearchCV to find the best kernel and gamma value for kPCA

Kernel PCA - Selecting hyperparameters

Selecting hyper parameters

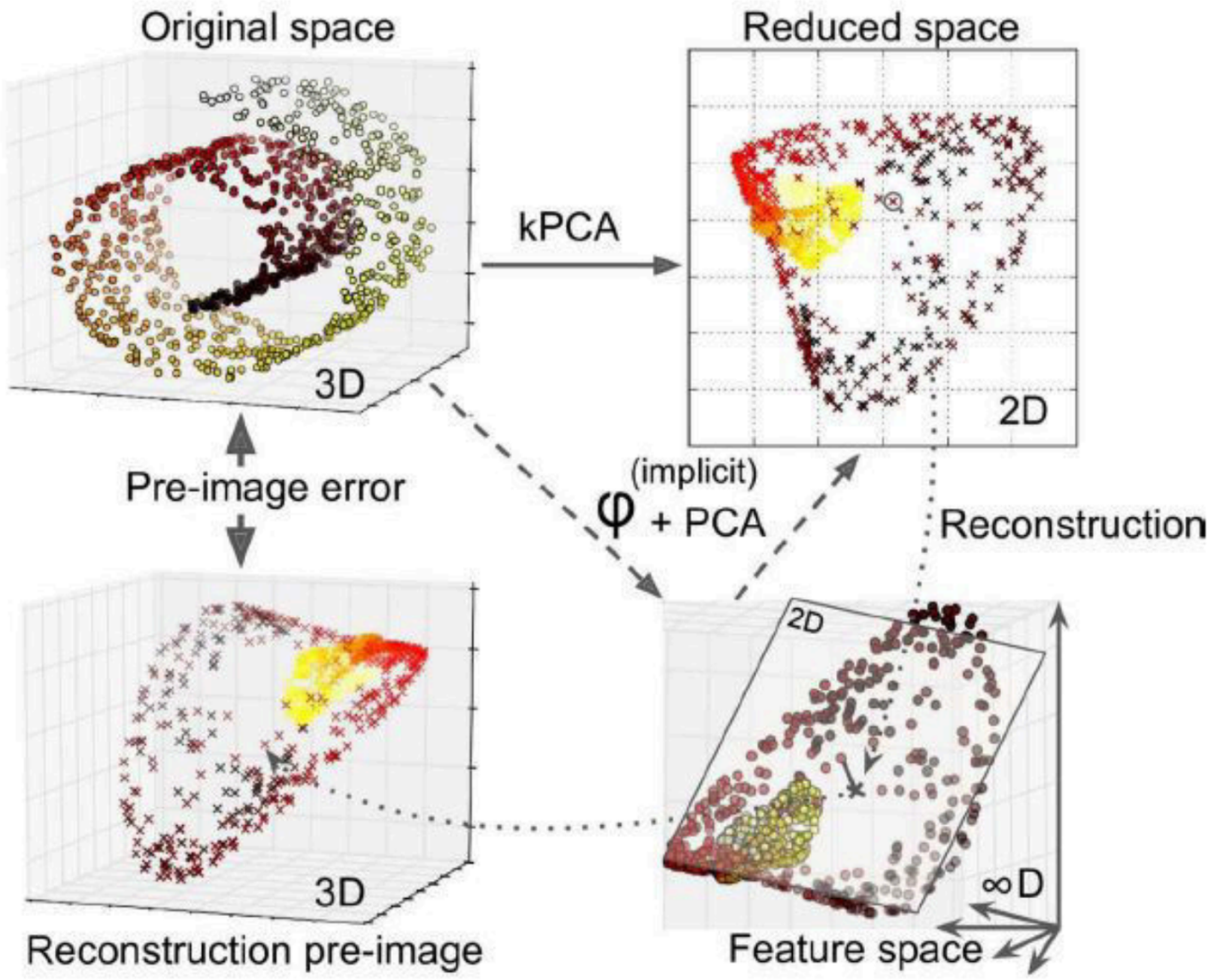
- Create a pipeline with KernelPCA and Classification model
- Doing a grid search using GridSearchCV to find the best kernel and gamma value for kPCA

```
>>> 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)
```

Switch to Notebook

Kernel PCA - Reconstruction

Reconstruction in Kernel PCA



Kernel PCA - Reconstruction

Reconstruction in Kernel PCA

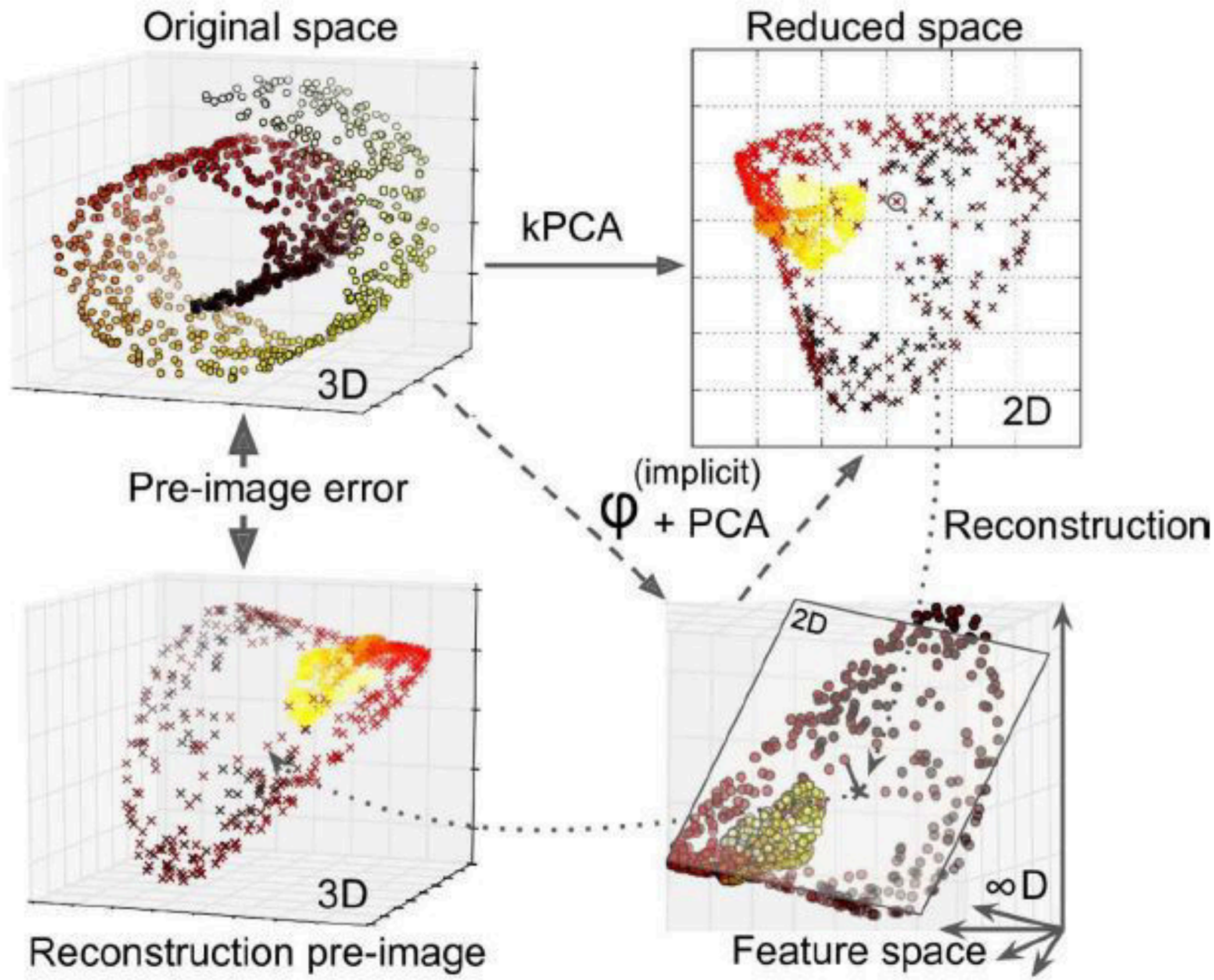
- 2 steps followed in Kernel PCA
  - Mapping to a higher infinite-dimensional feature space
  - Then projecting the transformed training set into 2d using linear PCA
- Inverse of linear PCA step would lie in the feature space, not in the original space
  - Since infinite-dimensional, we cannot compute the reconstruction point
  - Therefore, cannot compute the true reconstruction error

Kernel PCA - Reconstruction

Reconstruction in Kernel PCA

- For reconstruction, we instead use a pre-image
  - By finding a point in the original space that would map close to the reconstructed point
  - Can find the squared distance with the original space
  - Then select the kernel and hyperparameters that minimize the reconstruction pre-image error

Kernel PCA - Reconstruction



Kernel PCA - Reconstruction Error

Calculating reconstruction error when using kernel PCA

- Inverse_transform in scikit-learn creates the pre-image
- Which can be used to calculate the mean squared error

```
## Performing Kernel PCA and enabling inverse transform
## to enable pre-image computation
>>> rbf_pca = KernelPCA(
        n_components = 2,
        kernel="rbf",
        gamma=0.0433,
        fit_inverse_transform=True) # perform reconstruction
```

...contd

Kernel PCA - Reconstruction Error

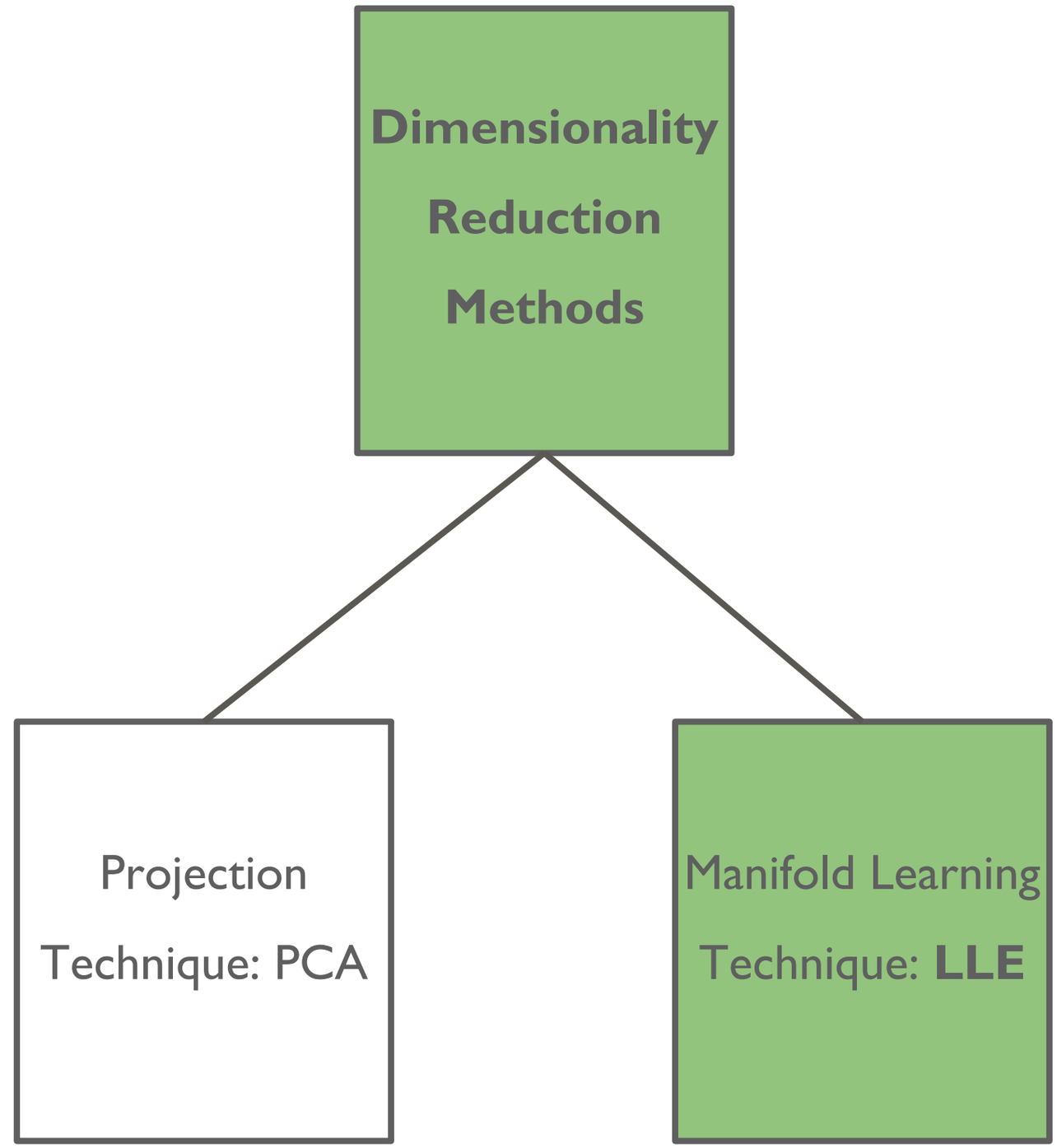
Calculating reconstruction error when using kernel PCA

- Inverse_transform in scikit-learn creates the pre-image
- Which can be used to calculate the mean squared error

```
## Calculating the reduced space using kernel PCA and pre-image
>>> X_reduced  = rbf_pca.fit_transform(X)
>>> X_preimage = rbf_pca.inverse_transform(X_reduced)

# return reconstruction pre-image error
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(X, X_preimage)
```

Switch to Notebook

Dimensionality Reduction



Incremental PCA,
Randomized PCA,
Kernel PCA

LLE

Local Linear Embedding (LLE)

- Another powerful nonlinear dimensionality reduction (NLDR) technique
- Manifold technique that does not rely on projections
- Works by
  - Measuring how each training instance linearly relates to its closest neighbours
  - Then looking for low-dimensional representation where these local relationships are best preserved
- Good at unrolling twisted manifolds, especially when there is not much noise

LLE

Local Linear Embedding (LLE) in scikit-learn

- LocallyLinearEmbedding class in sklearn.manifold
- Run on the swiss roll example

- **Step 1:** Make the swiss roll

```
>>> from sklearn.datasets import make_swiss_roll

>>> X, t = make_swiss_roll(
        n_samples=1000,
        noise=0.2,
        random_state=41)
```

...contd

LLE

Local Linear Embedding (LLE) in scikit-learn

- LocallyLinearEmbedding class in sklearn.manifold
- Run on the swiss roll example

- **Step 2:** Instantiate LLE class in sklearn and fit the swiss roll training features using the LLE model

```
>>> from sklearn.manifold import LocallyLinearEmbedding

>>> lle = LocallyLinearEmbedding(
        n_neighbors=10,
        n_components=2,
        random_state=42)

>>> X_reduced = lle.fit_transform(X)
```

...contd

LLE

Local Linear Embedding (LLE) in scikit-learn

- LocallyLinearEmbedding class in sklearn.manifold
- Run on the swiss roll example

- **Step 3:** Plot the reduced dimension data

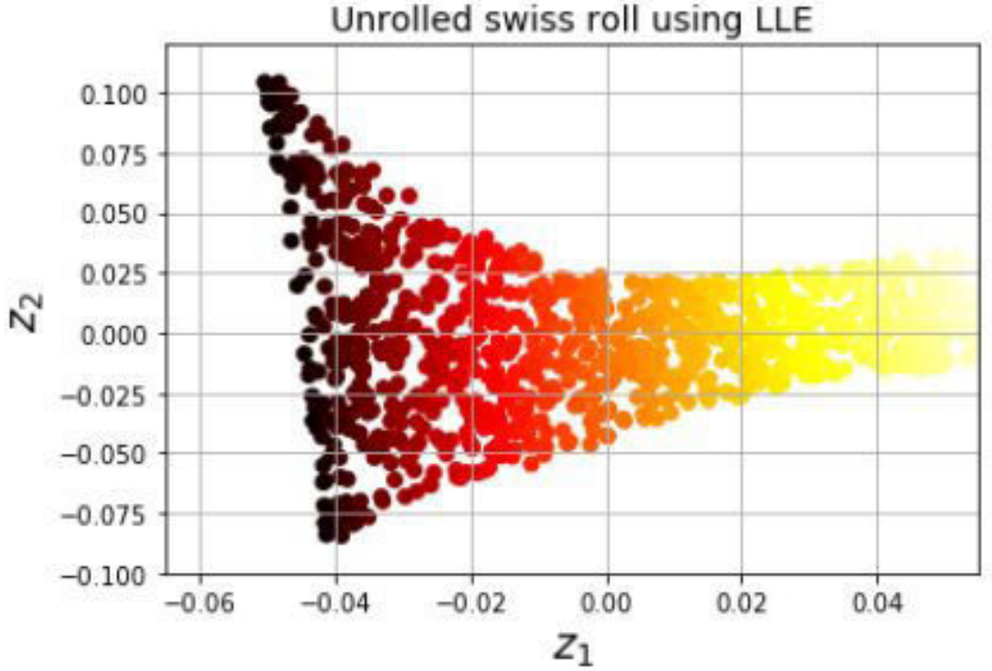
```
>>> plt.title("Unrolled swiss roll using LLE", fontsize=14)
>>> plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=plt.cm.hot)
>>> plt.xlabel("$z_1$", fontsize=18)
>>> plt.ylabel("$z_2$", fontsize=18)
>>> plt.axis([-0.065, 0.055, -0.1, 0.12])
>>> plt.grid(True)

>>> plt.show()
```

...contd

LLE

Local Linear Embedding (LLE) in scikit-learn

- LocallyLinearEmbedding class in sklearn.manifold
- Run on the swiss roll example

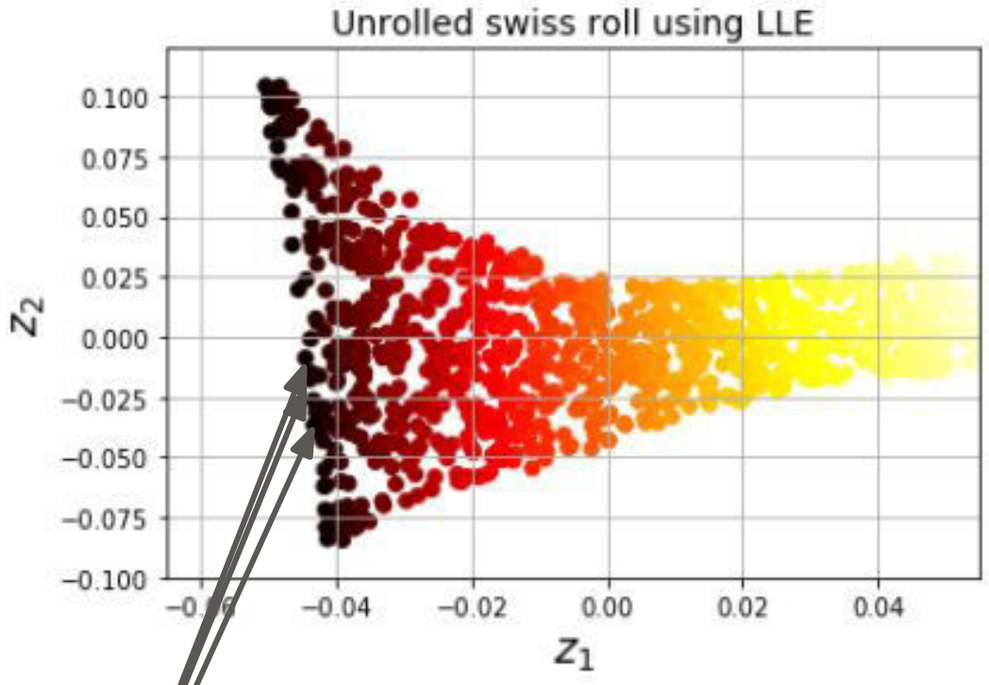


Switch to Notebook

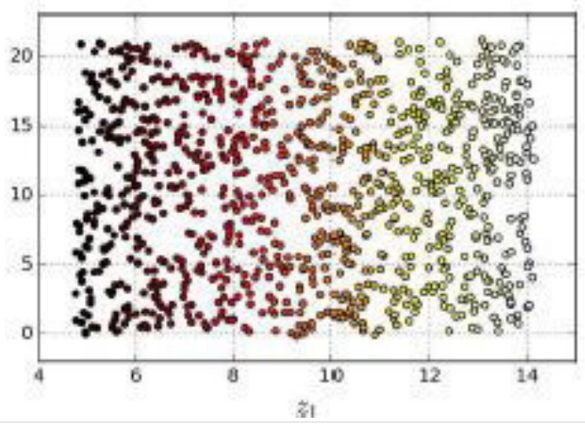
LLE

Observations

- Swiss roll is completely unrolled
- Distances between the instances are locally preserved
- Not preserved on a larger scale
  - Left most part is squeezed
  - Right part is stretched



Distance locally preserved



LLE - How it Works? Maths!

How LLE works?

Step 1: For each training instance, the algorithm identifies the k closest neighbours

Step 2: reconstructs the instance as a linear function of these closest neighbours

- More specifically, finds the weight w vector such that distance between the closest neighbours and the instance is as small as possible.

$$\hat{\boldsymbol{W}} = \arg\min_{\boldsymbol{W}} \sum_{i=1}^{m} \| \boldsymbol{x}^{(i)} - \sum_{j=1}^{m} w_{i,j} \boldsymbol{x}^{(j)} \|^2$$

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

LLE - How it Works? Maths!

How LLE works?

Step 3: Map the training instances into a d-dimensional space while preserving the local relationship as much as possible

- Basically, keeping the same weight as calculated in the previous step, the new instance should have minimum distances with the previous closest neighbours (same weights and relationship)

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

LLE - Time Complexity

How LLE works?

Step 1: finding K nearest neighbors: O(m x log(m) x n x log(k))

Step 2: weight optimization: O(m x n x k^3)

Step 3: constructing low-d representations: O(d x m^2)

Where m = number of training datasets,

n = number of original dimensions

k = nearest neighbours

d = reduced dimensions

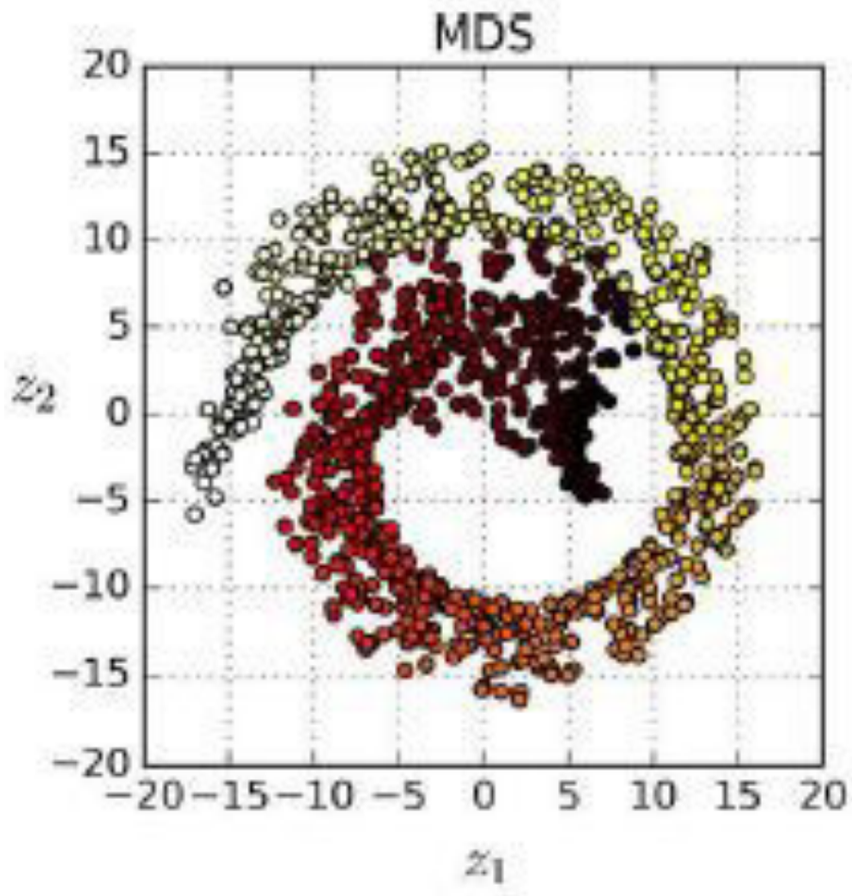
Step 3 makes the model very slow for large number of training datasets

Other dimensionality techniques

Multidimensional Scaling (MDS)

- Reduces dimensionality
- trying to preserve the instances

```
>>> from sklearn.manifold import MDS
>>> mds = MDS(n_components=2,
random_state=42)
>>> X_reduced_mds = mds.fit_transform(X)
```

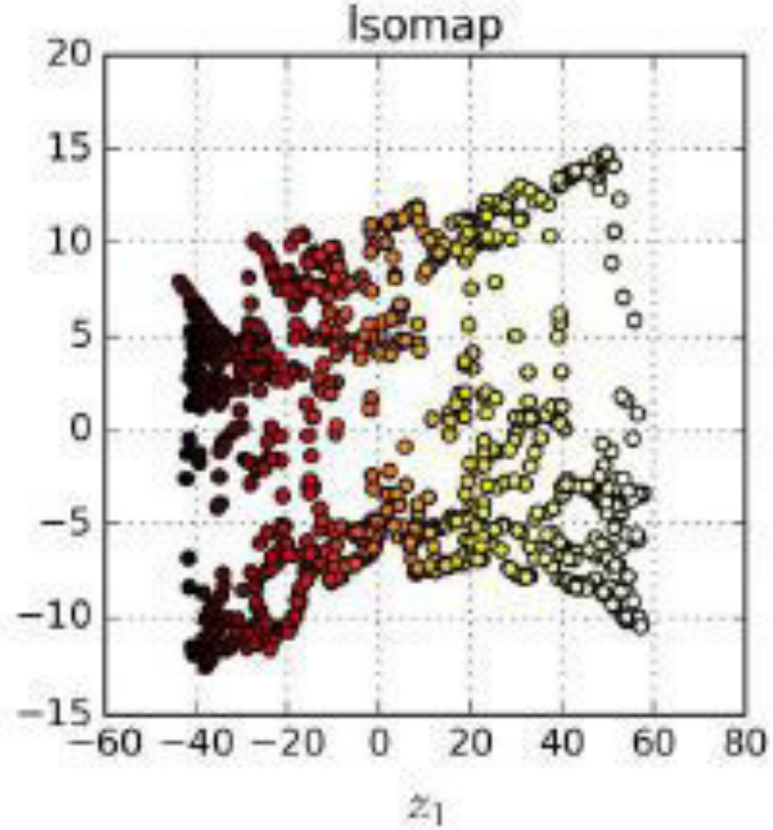


Other dimensionality techniques

Isomap

- Creates a graph connecting each instance to its nearest neighbours
- Then, reduces dimensionality
- Trying to preserve geodesic distances between instances

```
>>> from sklearn.manifold import Isomap
>>> isomap = Isomap(n_components=2)
>>> X_reduced_isomap =
isomap.fit_transform(X)
```

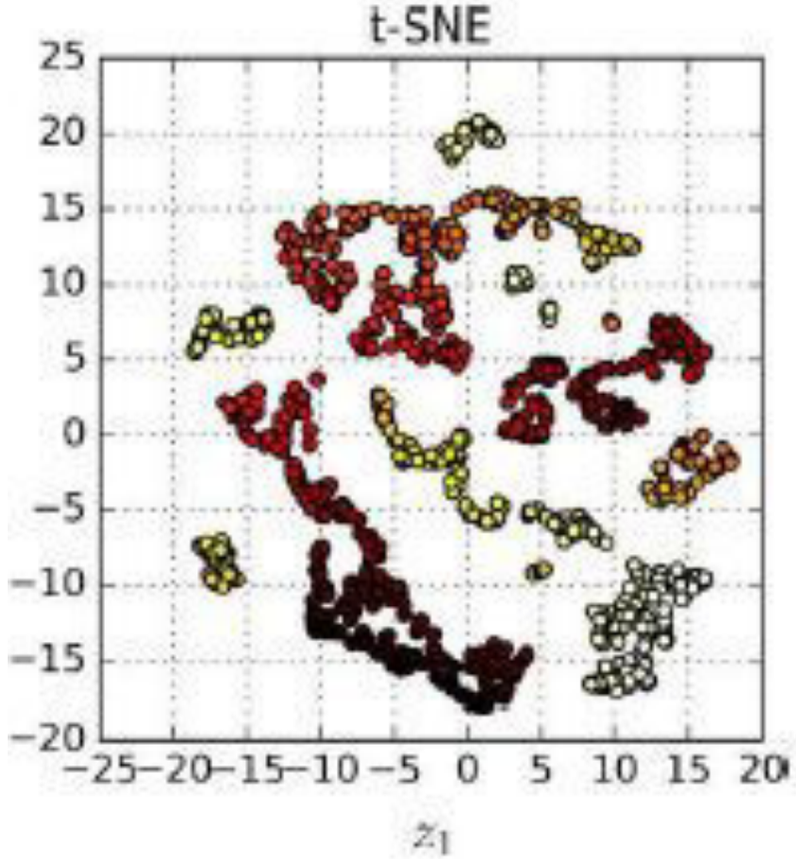


Other dimensionality techniques

T-distributed Stochastic Neighbour Embedding

- Reduces dimensionality
- Keeping similar instances close and dissimilar apart
- Mostly used for visualize clusters in high-dimensional space

```
>>> from sklearn.manifold import TSNE
>>> tsne = TSNE(n_components=2)
>>> X_reduced_tsne = tsne.fit_transform(X)
```



Other dimensionality techniques

Linear Discriminant Analysis (LDA)

- A classification algorithm
- During training learns the most discriminative axes between the classes
- Axes can be used to define the hyper plant to project the data
- Projection will keep the classes as far apart as possible
- A good technique to reduce dimensionality before running classification algorithms such as SVM Classifier

Other dimensionality techniques

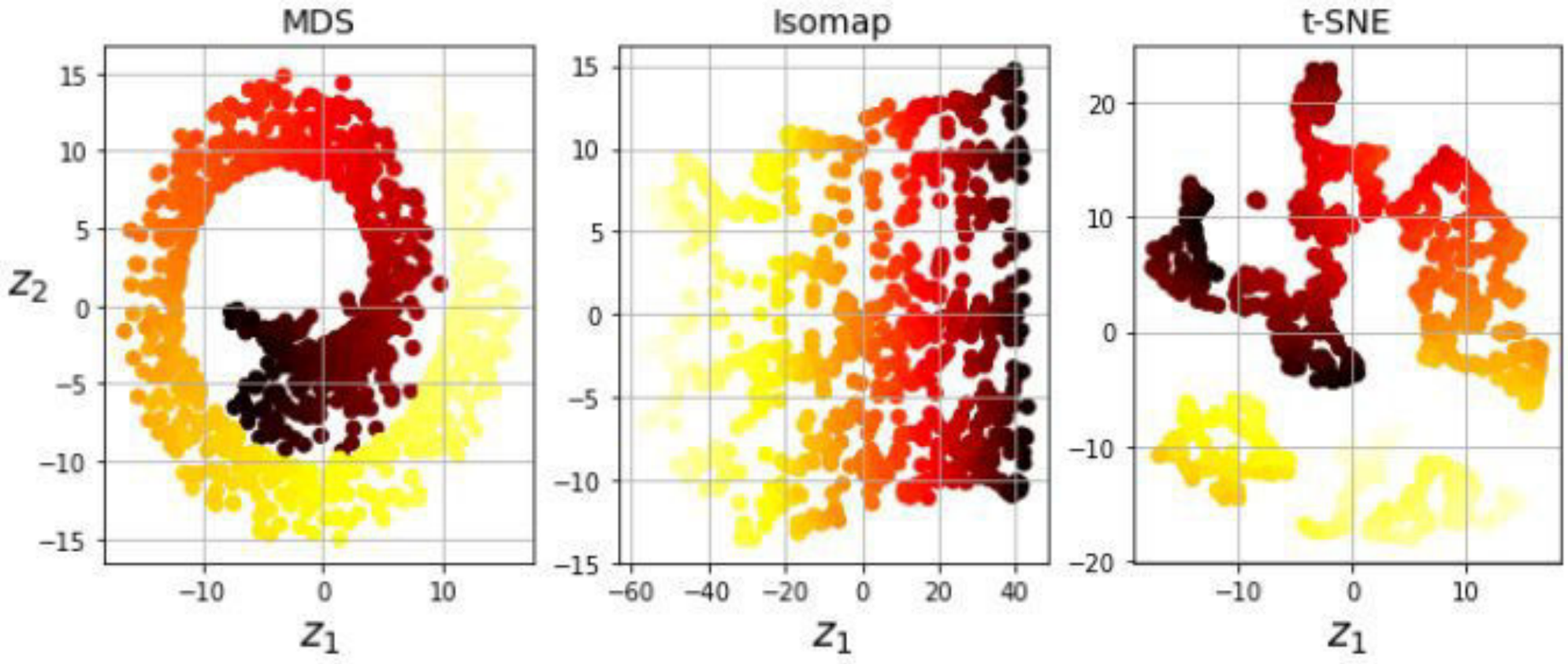
Plotting the results for each of the techniques on the notebook

```
>>> titles = ["MDS", "Isomap", "t-SNE"]

>>> plt.figure(figsize=(11,4))
for subplot, title, X_reduced in zip((131, 132, 133), titles,
                                     (X_reduced_mds, X_reduced_isomap, X_reduced_tsne)):
    plt.subplot(subplot)
    plt.title(title, fontsize=14)
    plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=plt.cm.hot)
    plt.xlabel("$z_1$", fontsize=18)
    if subplot == 131:
        plt.ylabel("$z_2$", fontsize=18, rotation=0)
    plt.grid(True)

>>> plt.show()
```

Switch to Notebook

Other dimensionality techniques

Plotting the results for each of the techniques on the notebook



Archives

PCA- Projecting down to d dimensions

Similarly, the

- original training dataset X can be projected onto
- the first ‘d’ principal components Wd
  - Composed of first ‘d’ columns of transpose(V) obtained in SVD
- Reducing the dataset dimensions to ‘d’

$$X_{d\text{-}proj} = X.W_d$$

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

Wd = first d columns of transpose(V) containing the first d principal components