Dimensionality Reduction

Lecture 10

Last time: Learning Objectives

1. Discuss Unsupervised Learning methods using the Simpsons!
2. Represent “group” with distance of similarity
3. Learn some clustering algorithms: partitional and hierarchical
4. See how K-means algorithm works

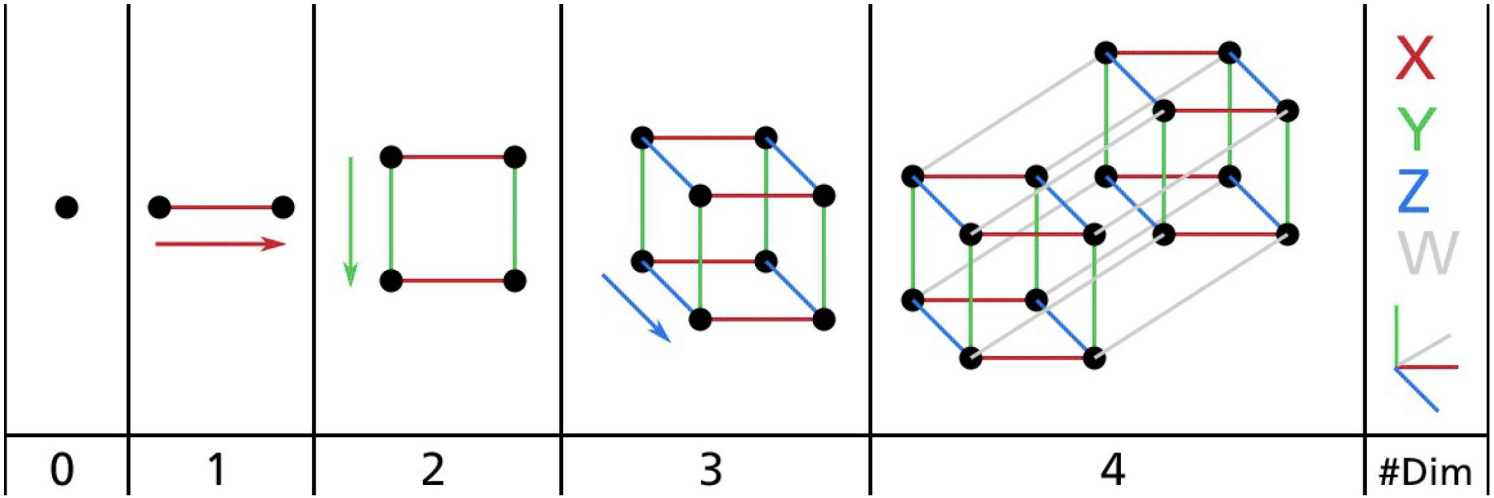
Today: Learning Objectives

1. Discuss Dimensionality Reduction methods including projection!
2. Know Principal Component Analysis (PCA) Algorithm
3. Use PCA and its variants
4. Understand Manifold Learning using Swiss Rolls
5. Learn Locally Linear Embedding (LLE) algorithm

Feature Dimensionality

Feature Dimensionality

- ML problem can involve hundreds or thousands of features for each sample.
- Slow training, hard to find good solution → the curse of dimensionality
  - Hard to visualize beyond 3D (try to image a 4D hypercube?)
  - Things behave very differently in high-dimensional space (ie. they are **sparse**)



Dimensionality Reduction

Warning: It speeds up training, but you will also lose some information

Useful for data visualization (2D, 3D graph)

We will discuss:

- 2 main approaches in reduction: **Projection** and **Manifold Learning**
- 3 popular reduction techniques: **PCA**, **Kernel PCA**, and **LLE**.

Projection: An Approach to Dim Reduce

Training samples are not distributed uniformly across all dimensions

- some features are almost constant
- while other are highly correlated

Project them into a lower-dim subspace

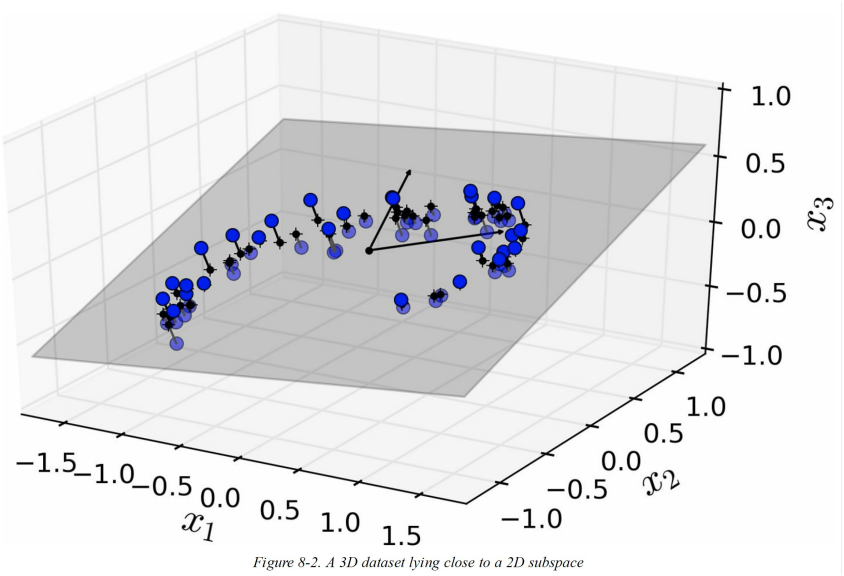


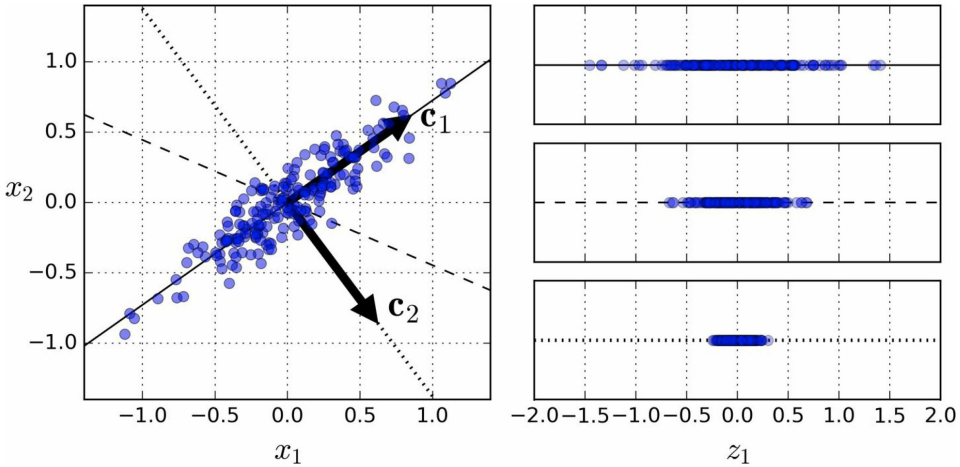
Figure 8-2. A 3D dataset lying close to a 2D subspace

Principal Component Analysis

Dimensionality Reduction Algorithm: PCA

Principal Component Analysis (**PCA**): works by **identifies** the hyperplane that lies **closest** to the data, and then **projects** the data onto it.

PCA selects the axis that preserves the **maximum amount of variance**, as it will most likely lose less information than other projections.



Applications of PCA

Uses:

- Data Visualization
- Data Reduction
- Data Classification
- Trend Analysis
- Factor Analysis
- Noise Reduction

Examples:

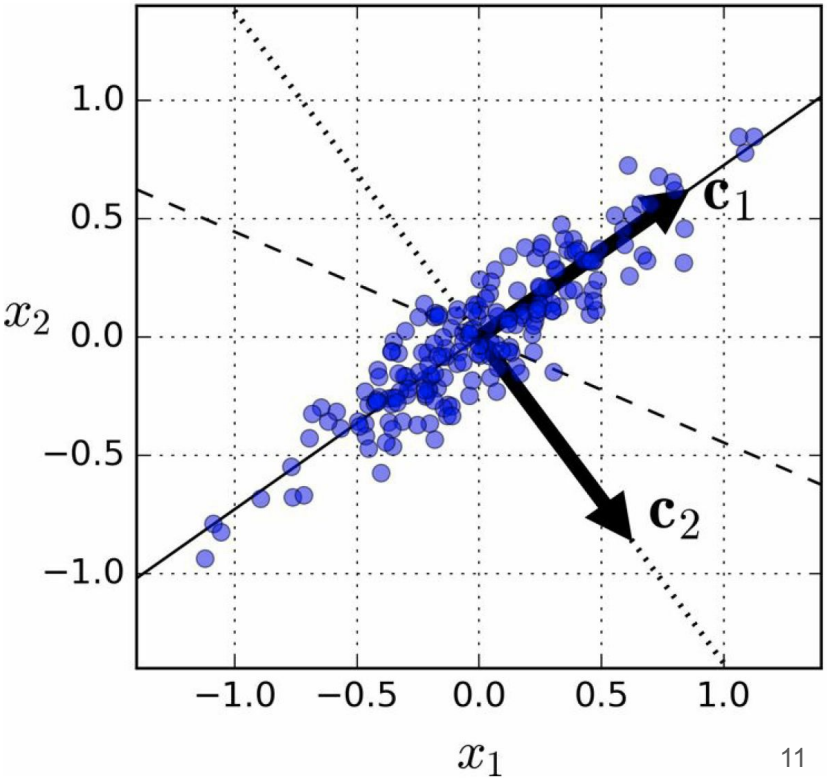
- How many unique “subsets” are in the data?
- How are they similar / different?
- What are the underlying factors that influence the data?
- How to best present what is“interesting”?
- Which “subset” does this new example rightfully belong?

Principal Components

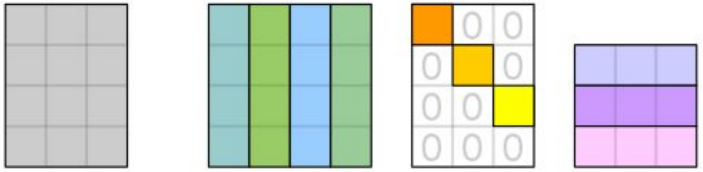
PCA identifies the axis ($\mathbf{c}_1$) that accounts for the **largest** amount of **variance**

It also finds a second axis ($\mathbf{c}_2$), orthogonal to $\mathbf{c}_1$, that amounts to the remaining variance

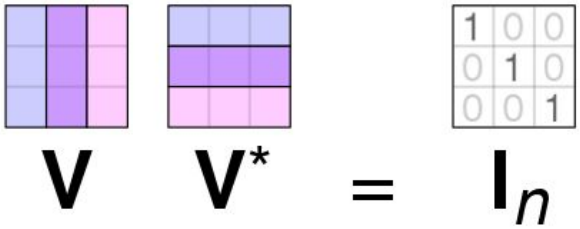
$\mathbf{c}_1$ is also called 1st principal component (1st PC), $\mathbf{c}_2$ is called 2nd PC.



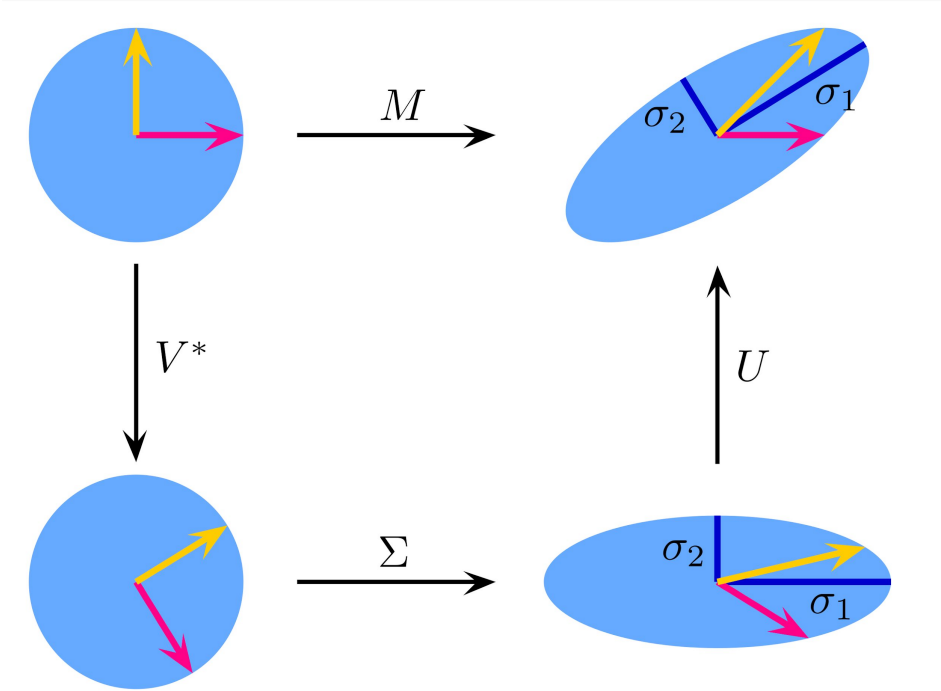
Review: Singular Value Decomposition (SVD)



V* is the conjugate transpose of V



Review: SVD on coordinate axes



$$M = U \cdot \Sigma \cdot V^*$$

How to find the PCs in a training set

Use a standard matrix factorization technique called Singular Value Decomposition (SVD) that decompose $\mathbf{X}$ into dot products

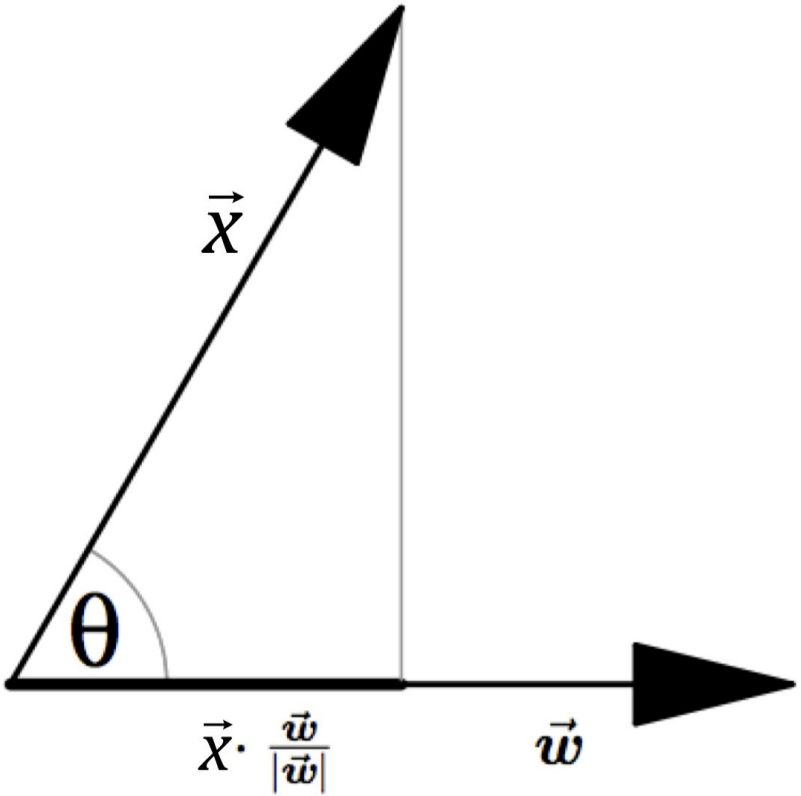
$$\mathbf{X} = \mathbf{U} \cdot \mathbf{\Sigma} \cdot \mathbf{V}^*$$

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

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

Must center the data since PCA assumes the dataset is centered around the origin

Review: Dot Product as Projection



Projecting Down to d Dimension

After identifying all principal components (PCs), we can project the dataset down to the hyperplane defined by the first d PCs using:

$$X_{proj} = X \cdot V_d^T$$

Containing the first d-PCs.

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

You can reduce the dimensionality of any dataset to any number of dimensions, while preserving as much variance as possible...but with what cost?

Using Scikit Learn

Similar to what we implemented before using SVD Decomposition

```
from sklearn.decomposition import PCA

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

pca.components_.T[:, 0])
```

First Principle Component

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

1st PC explained 84% variance

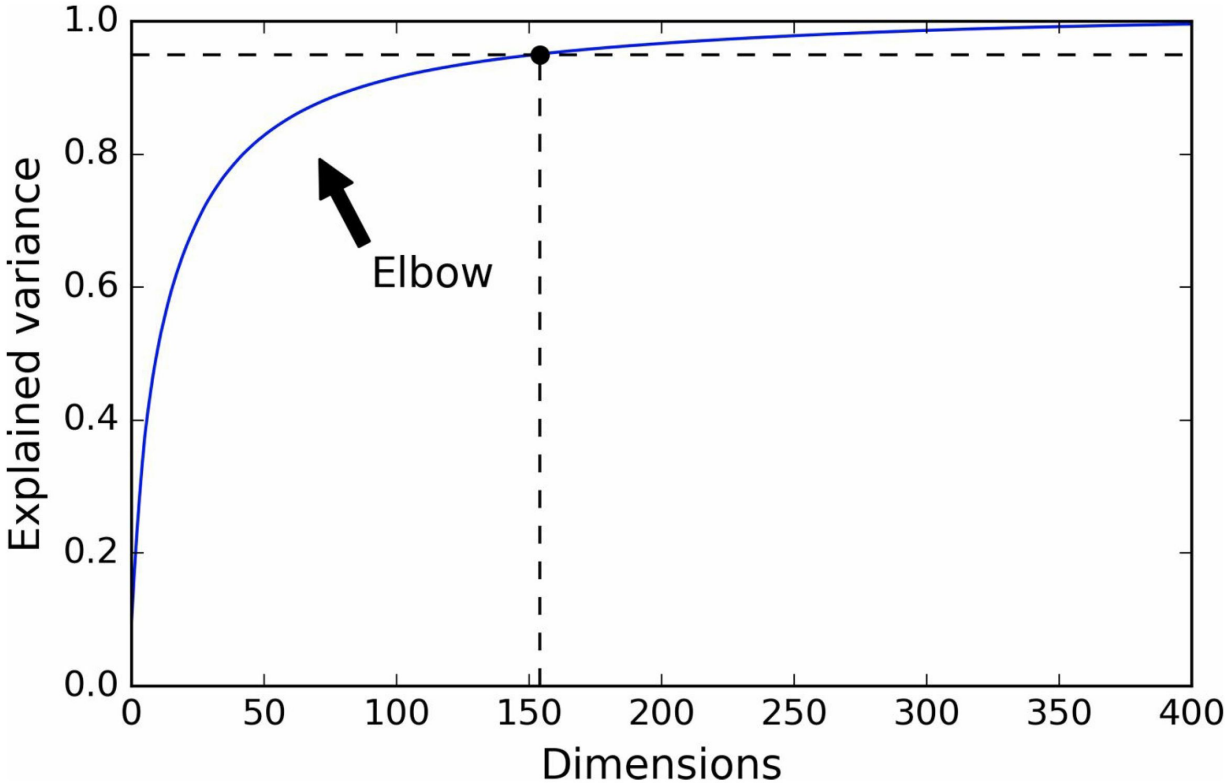
Choosing the right number of dimensions

Prefer to choose the number of dim. to add up a sufficiently large portion of variance (95%)

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

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

Explained Variance as a function of dimensions

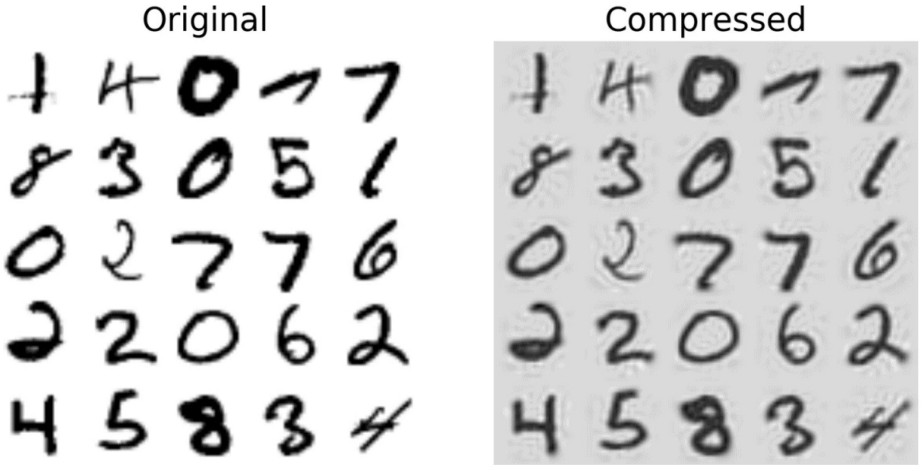


PCA Variants

PCA for Compression

MNIST dataset: reserved 95% variance: 784 features → 154 features

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



Incremental PCA

A problem with PCA: needs the whole training set to fit in memory to train

Incremental PCA: split training set into mini-batches to feed in at a time

→ works with large training sets, also apply to PCA online

```
from sklearn.decomposition import IncrementalPCA

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

X_mnist_reduced = inc_pca.transform(X_mnist)
```

Kernel PCA (kPCA)

Recall the **kernel trick** of SVM that implicitly maps samples into high-D space?

Same trick can be applied with PCA → Kernel PCA: perform complex non-linear projection for dimensionality reduction.

Good at **preserving clusters** of samples after projection.

```
from sklearn.decomposition import KernelPCA

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

Projection: 3D → 2D

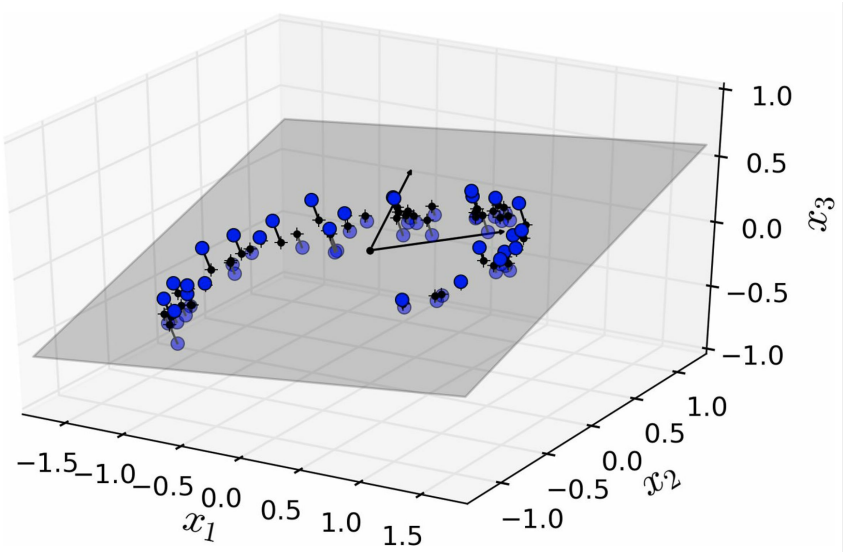
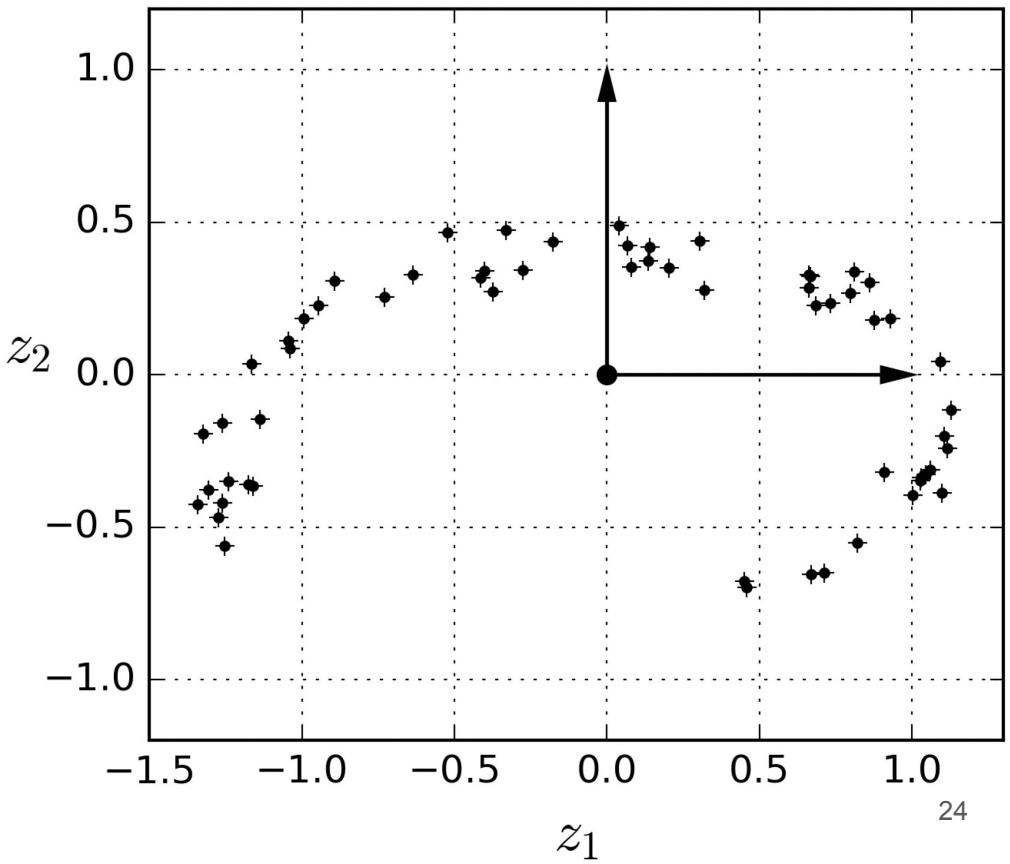


Figure 8-2. A 3D dataset lying close to a 2D subspace

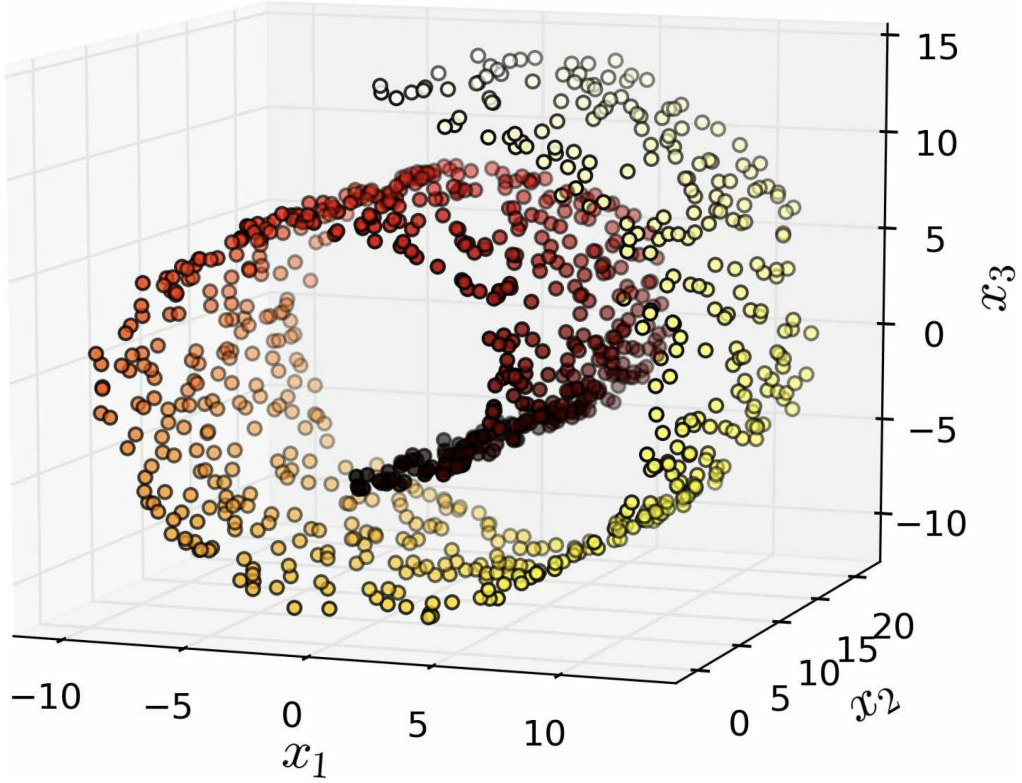


Not always work out this nice...

Swiss Roll Dataset

`sklearn.datasets.` **`make_swiss_roll`** (*n_samples=100, noise=0.0, random_state=None*) [source]

Let's try Projection on this!



Projection on Swiss Roll

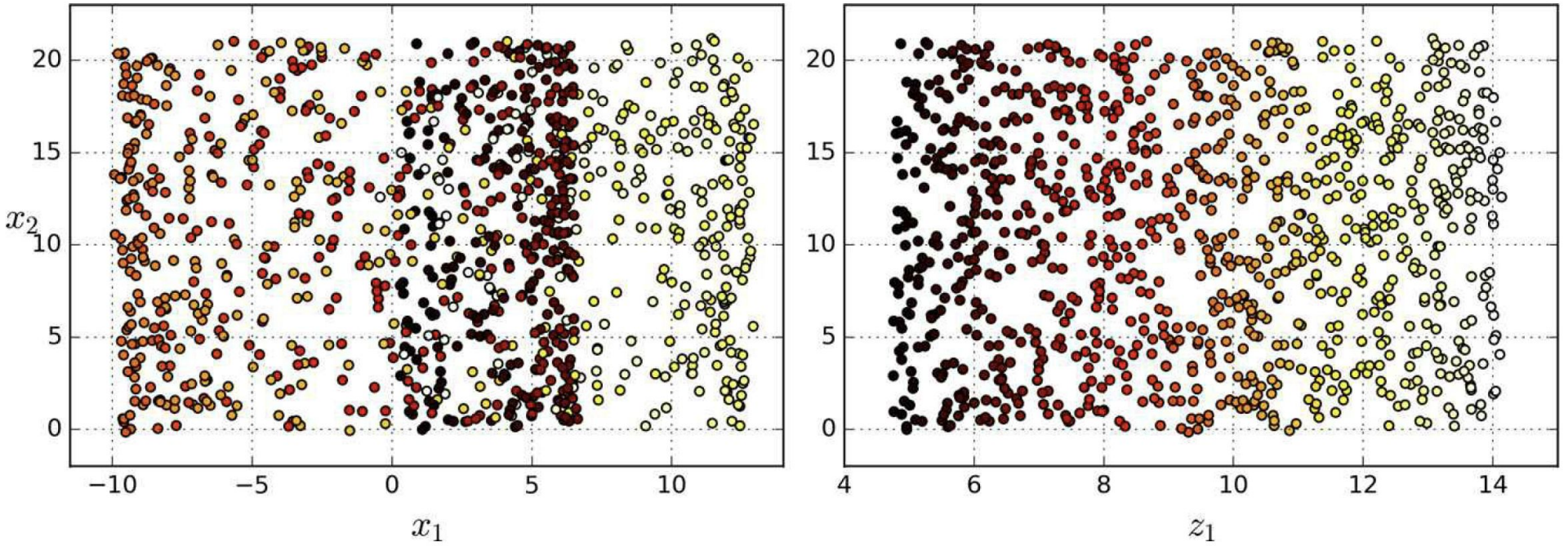


Figure 8-5. Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

Which approach works better? (obviously)

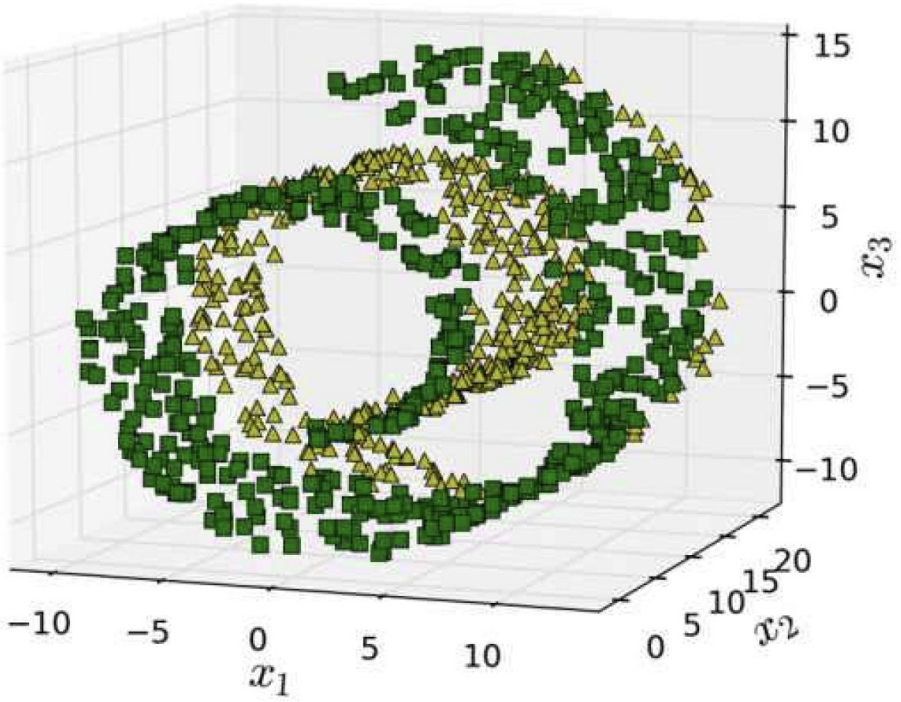
Manifold Learning

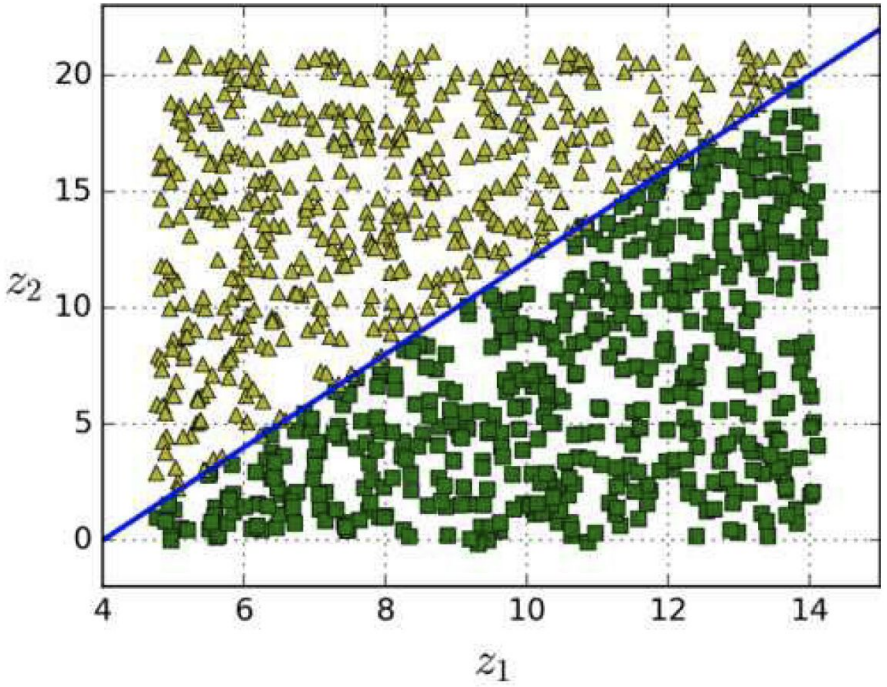
Manifold Learning

- A d-dimensional manifold is a part of an n-dimensional space (where `d < n`) that locally resembles a d-dimensional hyperplane.
    - The Swiss roll is an example of a 2D manifold. `d= 2` and `n=3`: it locally resembles a 2D plane, but it is rolled in the 3rd dimension.
- Manifold Learning is a non-linear technique to reduce the dimensionality by modeling a manifold on which the training samples lie.
    - It relies on the manifold assumption that most real-world high-dimensional dataset lie close to a much lower-dimensional manifold (empirically observed).

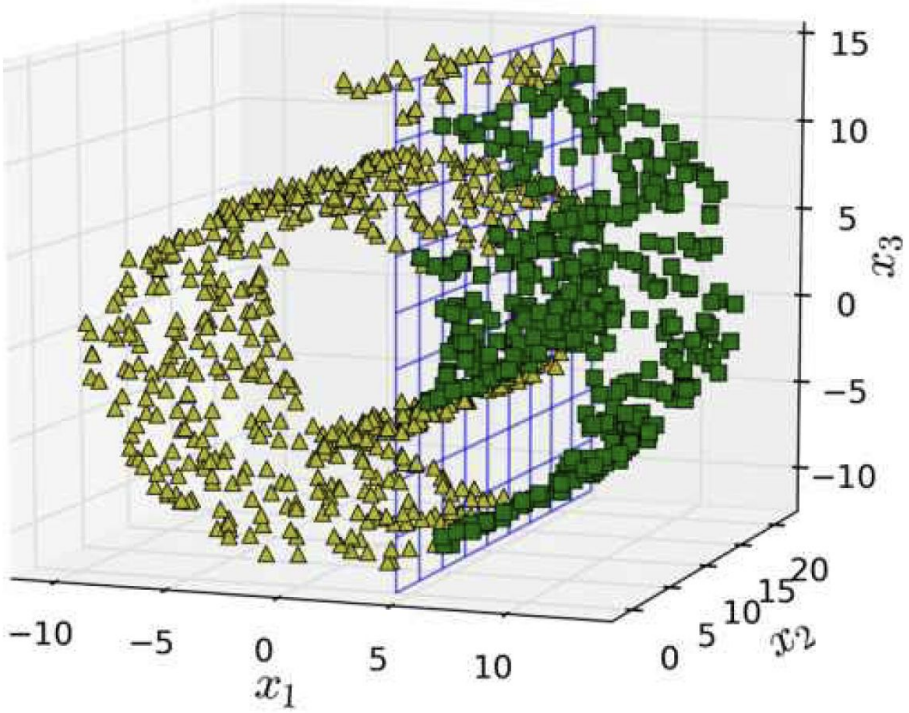
Real world examples of a manifold

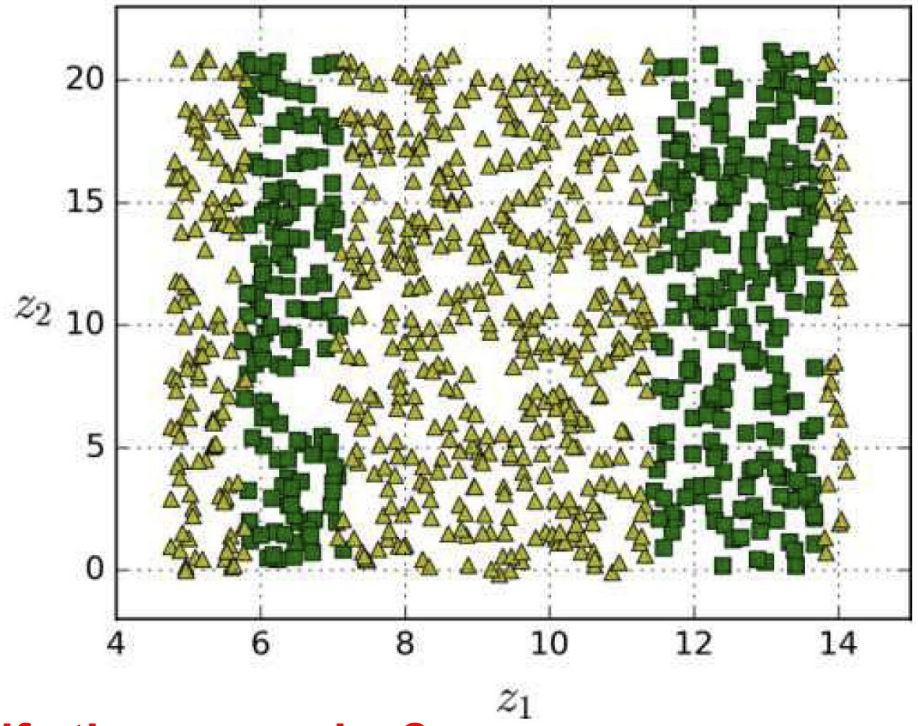
Decision Boundary





DB may not always be simpler with lower dim





Can you still classify these samples?

Locally Linear Embedding (LLE)

Locally Linear Embedding (LLE)

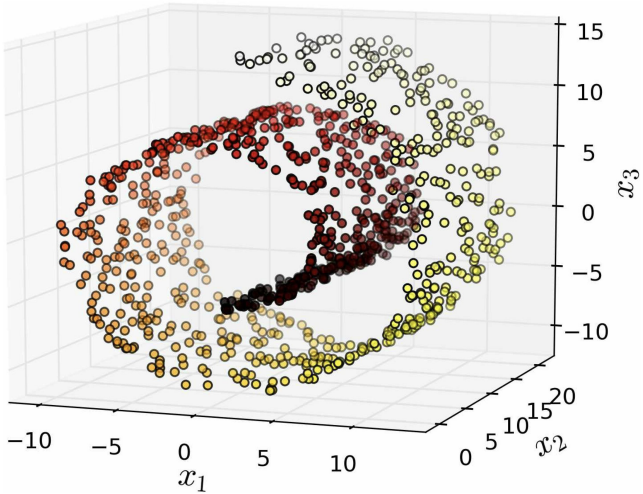
A powerful non-linear dimensionality reduction technique

Manifold Learning method that does **not** rely on projections

1. Measures how each training sample linearly relates to its closest neighbors
2. Look for a low-d representation of training set when these local relationships are best preserved

Good at unrolling twisted manifold or unravelling

the internal structure of the data



How it works

- For each training samples $x^{(i)}$, identifies k (ie. 10) **closest** neighbors
- Reconstruct $x^{(i)}$ as a linear function of the neighbors using weights $w(i,j)$
- **Optimize weight matrix W**:

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

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

- Map the samples $x^{(i)}$ into a d-dimensional space $z^{(i)}$ using reverse step
- **Optimize the squared distance** keeping the weights fixed:

$$\mathbf{Z} = \arg\min_{\mathbf{Z}} \sum_{i-1}^{m} ||\mathbf{z}^{(i)} - \sum_{j=1}^{m} w_{i,j}\mathbf{z}^{(j)}||^2$$

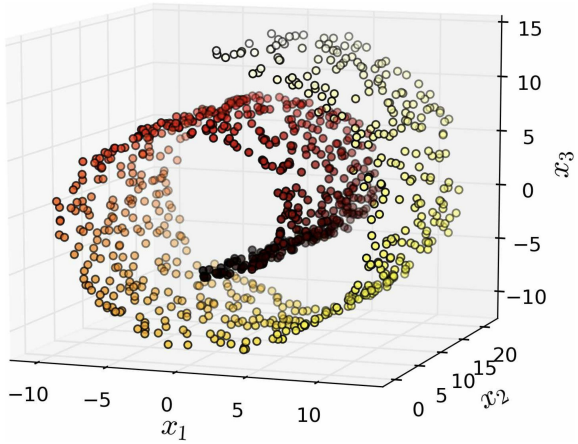
Unrolled the Swiss roll using LLE

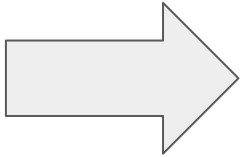
Distances between samples are locally well preserved.

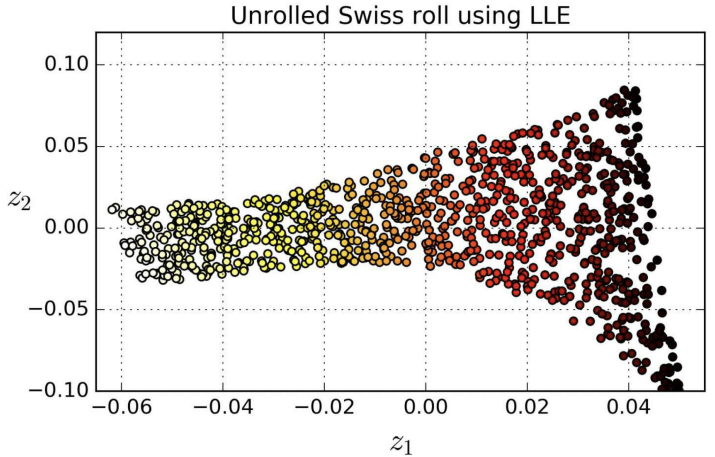
The distances are not preserved on a larger scale.

```
from sklearn.manifold import LocallyLinearEmbedding

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



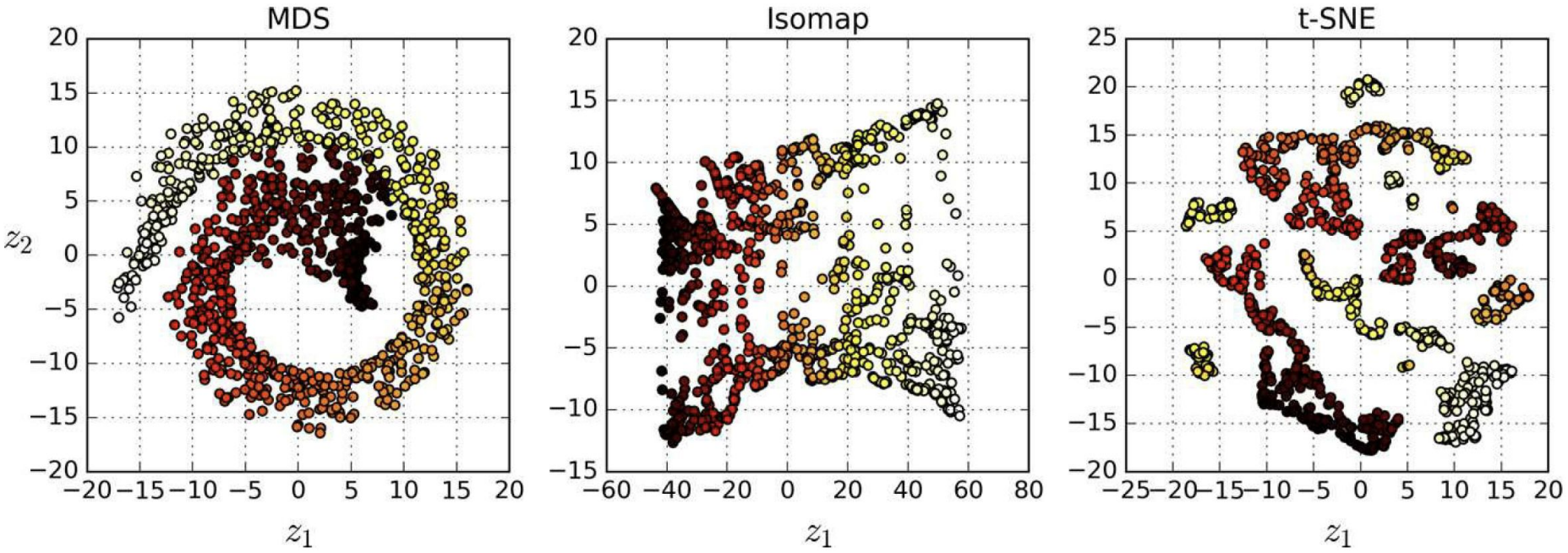




Other Dimensionality Reduction Techniques

1. **Multidimensional Scaling (MDS):** reduce dimensionality while trying to preserve the distances between the instances
2. **Isomap:** creates a graph by connecting each instance to its nearest neighbors and tries to preserve the *geodesic distances* between the samples
3. **T-distributed Stochastic Neighbor Embedding (t-SNE):** tries to keep similar samples close and dissimilar samples apart (visualization purposes)

Reducing the Swiss roll to 2D



PCA on Swiss Roll

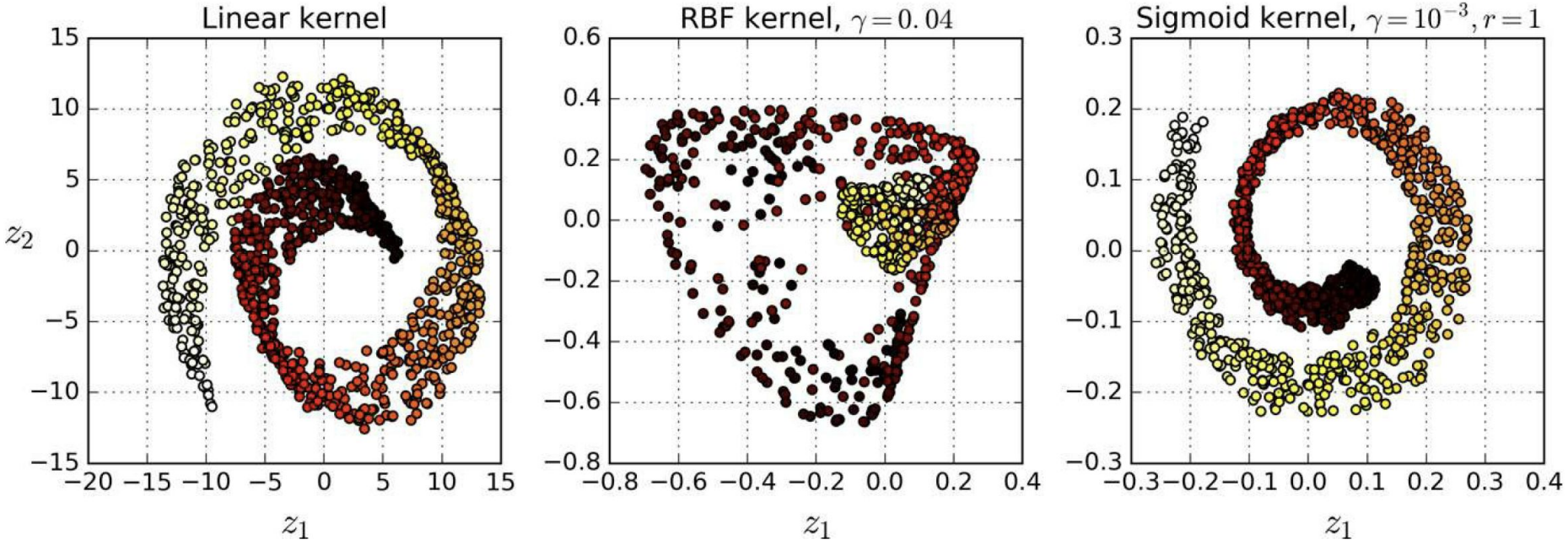


Figure 8-10. Swiss roll reduced to 2D using kPCA with various kernels

Recap: Learning Objectives

1. Discuss Dimensionality Reduction methods including projection!
2. Know Principal Component Analysis algorithm
3. Use PCA and its variants
4. Understand Manifold Learning using Swiss Rolls
5. Learn Locally Linear Embedding (LLE) algorithm

Bonus Slides

Randomize PCA

Alternative way to perform PCA stochastically: finds an approx of the first `d` PCs.

Reduce computation complexity from **$O(m \times n^2) + O(n^3) \rightarrow O(m \times d^2) + O(d^3)$**

Much faster when `d` is much smaller than `n`

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

Selecting a Kernel

- A preparation step for a supervised task (ie. classification)
- Simply use a grid search to select kernel and some hyperparameters

```
clf = Pipeline([
        ("kpca", KernelPCA(n_components=2)),
        ("log_reg", LogisticRegression())
    ])

param_grid = [{
        "kpca__gamma": np.linspace(0.03, 0.05, 10),
        "kpca__kernel": ["rbf", "sigmoid"]
    }]

grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
```

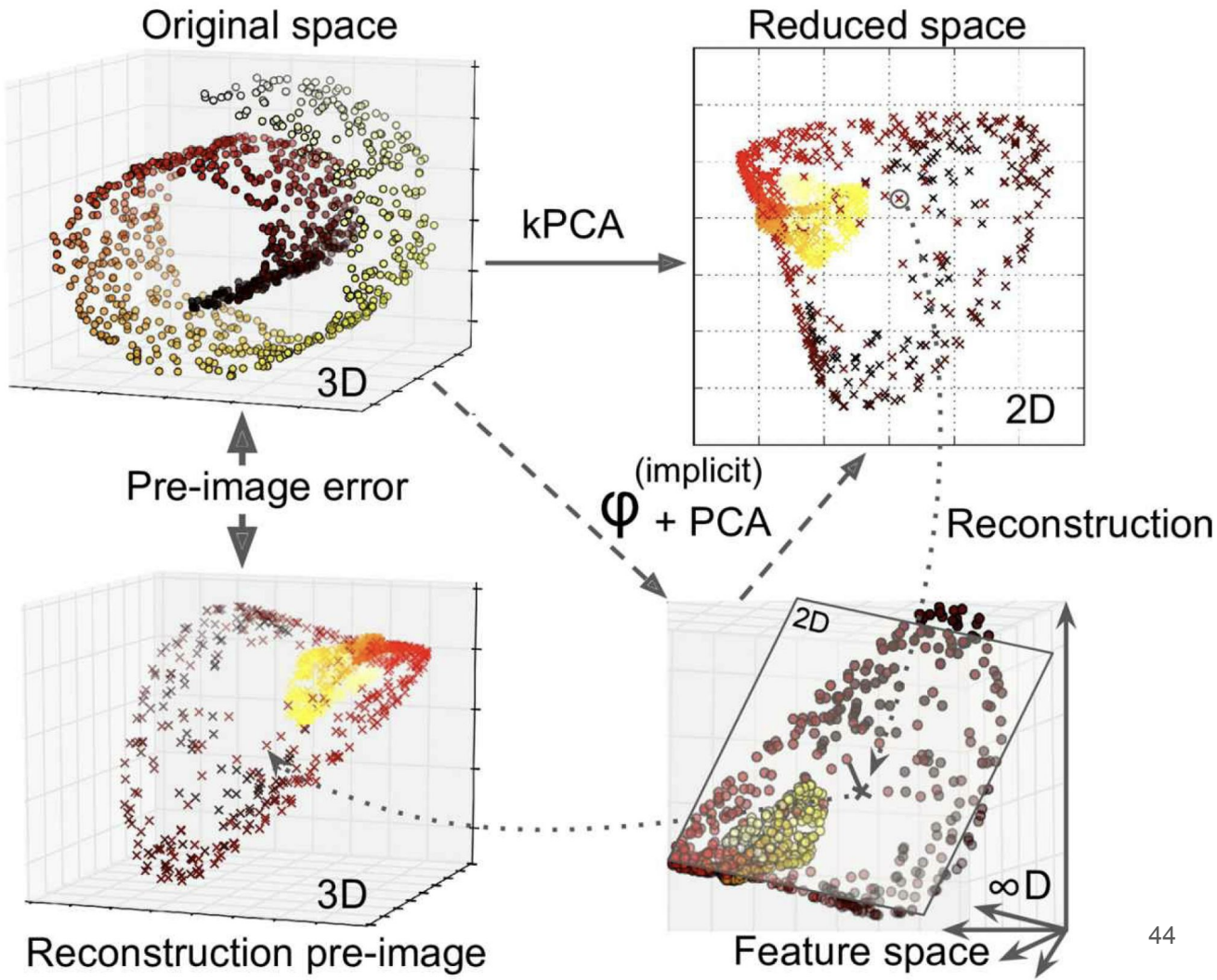
```
>>> print(grid_search.best_params_)
{'kpca__gamma': 0.043333333333333335, 'kpca__kernel': 'rbf'}
```

Computational Complexity of LLE

- Finding nearest neighbors: $O(m \log(m)\ n \log(k))$
- Optimizing weights W: $O(mnk^3)$
- Constructing low-d representation: $O(dm^2)$

What makes LLE scale poorly to very large dataset?

BONUS: Tuning by Reconstructior and Classification



Review: Eigenvector / Eigenvalue

The eigenvector of a linear transformation is a nonzero vector that changes by only a scalar factor when that linear transformation is applied to it:

$$\mathbf{T}(\mathbf{x}) = A\mathbf{x} = \lambda\mathbf{x} \qquad \Rightarrow (A - \lambda\mathbf{I})\mathbf{x} = 0 \quad (1)$$

Eigenvector Eigenvalue

Equation (1) has a nonzero solution x if and only if the det of the matrix is zero:

$$\Rightarrow \det(A - \lambda\mathbf{I}) = 0$$

Eigenvectors are columns of the matrix U such that:

$$A = UDU^T$$

$$\mathrm{D}= \begin{pmatrix} \lambda_1 0 \ldots\ldots 0 \\ 0 \quad \lambda_2 \ldots\ldots 0 \\ 0 \\ 0 \ldots\ldots\ldots\ldots \lambda_p \end{pmatrix}$$

Review: Mean and Variance

Mean: average (expected) value of a random variable

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

Variance: measures how far a set of variables are spread out from their mean.

$$\text{Var}\ (X) = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu)^2$$

Covariance: measures the joint variability of two random variables.

$$\text{Cov}\ (X, Y) = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu_x)(y^{(i)} - \mu_y)$$