Dimensionality Reduction

Introduction: the curse of dimensionality

Outline

- **Introduction: the curse of dimensionality**
- PCA
- Kernel PCA and LLE

Feature importance

- Output each feature's importance

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
...     print(name, score)
...
sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```

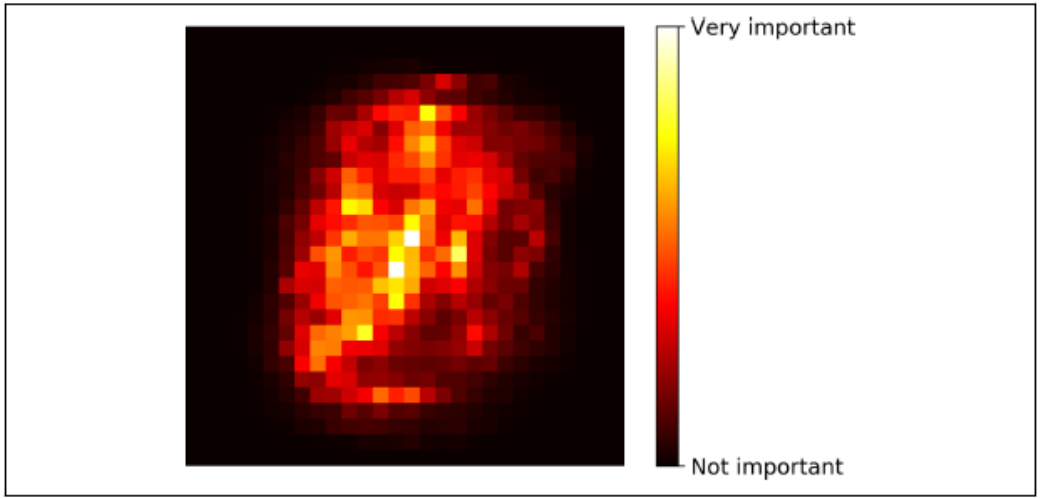

Figure 7-6. MNIST pixel importance (according to a Random Forest classifier)

The curse of dimensionality

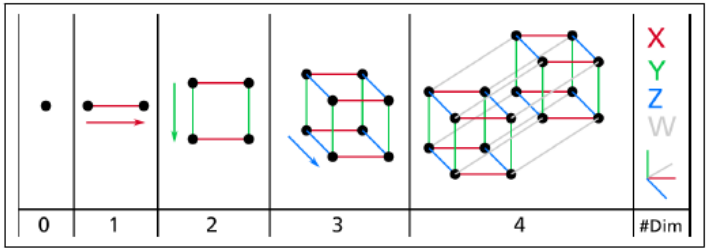

Figure 8-1. Point, segment, square, cube, and tesseract (0D to 4D hypercubes)[2]

- Pick a random point in a unit square (a 1 × 1 square), it will have only about a **0.4% chance of being located less than 0.001 from a border** (in other words, it is very unlikely that a random point will be “extreme” along any dimension).
- But in a 10,000-dimensional unit hypercube, **this probability is greater than 99.999999%. Most points in a high-dimensional hypercube are very close to the border.**

The curse of dimensionality

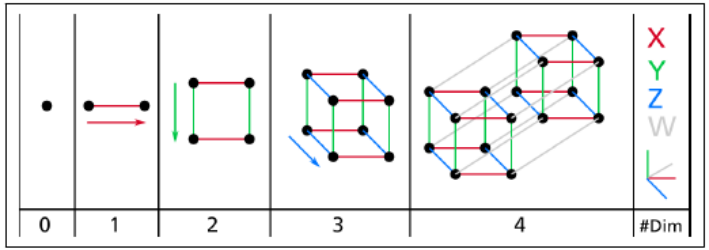

Figure 8-1. Point, segment, square, cube, and tesseract (0D to 4D hypercubes)[2]

- Pick two points randomly in a unit square, the distance between these two points will be, on average, **roughly 0.52**.
- If you pick two random points in a unit 3D cube, the average distance will **be roughly 0.66**.
- What about two points picked randomly in a 1,000,000-dimensional hypercube? The **average distance, believe it or not, will be about 408.25**
- As a result, high-dimensional datasets are at risk of being very sparse: most training instances are likely **to be far away from each other.**
- The more dimensions the training set has, the greater the risk of overfitting it.

The curse of dimensionality

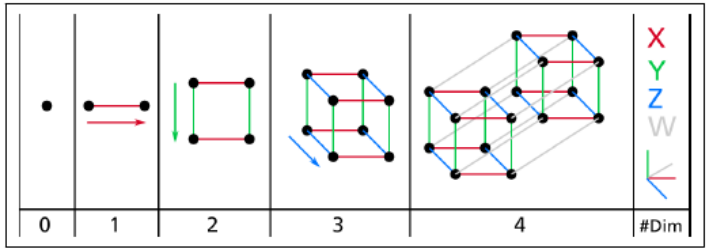

Figure 8-1. Point, segment, square, cube, and tesseract (0D to 4D hypercubes)[2]

- As a result, high-dimensional datasets are at risk of being very sparse: most training instances are likely **to be far away from each other.**
- The more dimensions the training set has, the greater the risk of overfitting it.
- One potential solution is to increase the size of training instances. However, it is difficult to do as the number of training instances required to reach a given density grows exponentially with the number of dimensions.
- Dimensionality reduction methods are useful.

Main approach for dimensionality reduction: projection

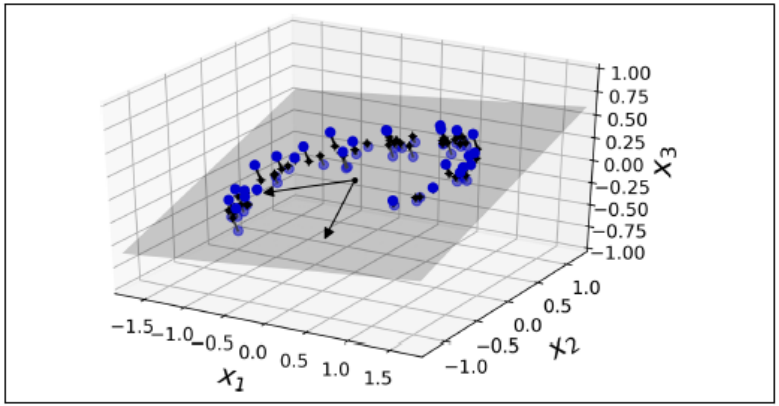

Figure 8-2. A 3D dataset lying close to a 2D subspace

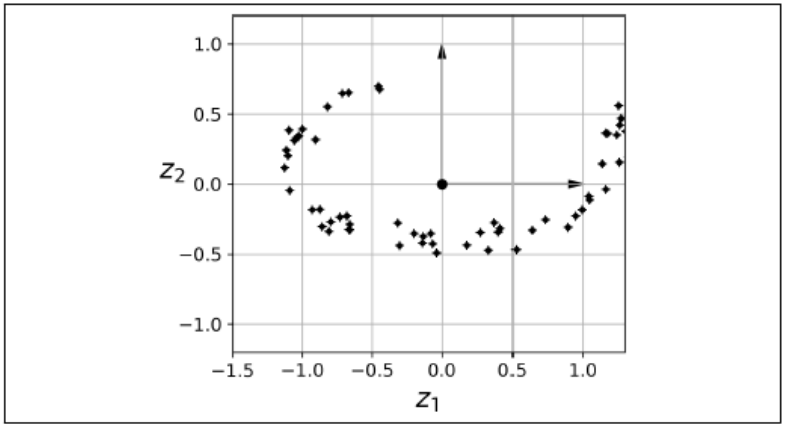

Figure 8-3. The new 2D dataset after projection

Main approach for dimensionality reduction: projection

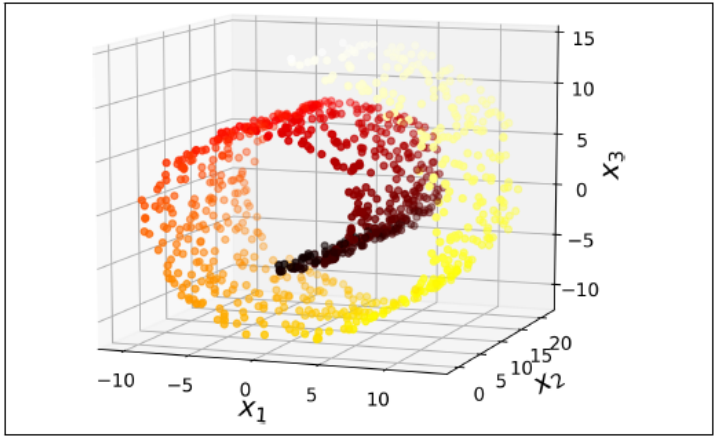

Figure 8-4. Swiss roll dataset

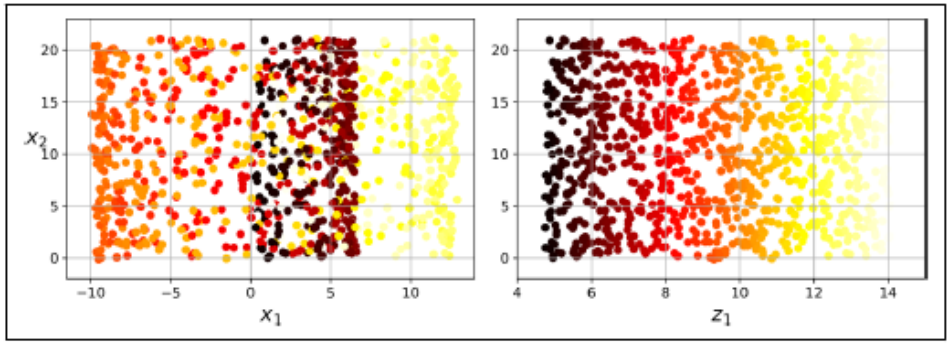

Figure 8-5. Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

Main approach for dimensionality reduction: manifold learning

- The Swiss roll is an example of a 2D *manifold*. Put simply, a 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space.
- Many dimensionality reduction algorithms work by modeling the manifold on which the training instances lie; this is called ***Manifold Learning***.
- It relies on the ***manifold assumption***, also called the *manifold hypothesis*, which holds that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold.
- This assumption is very often **empirically observed**.
- 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**.

Main approach for dimensionality reduction: manifold learning

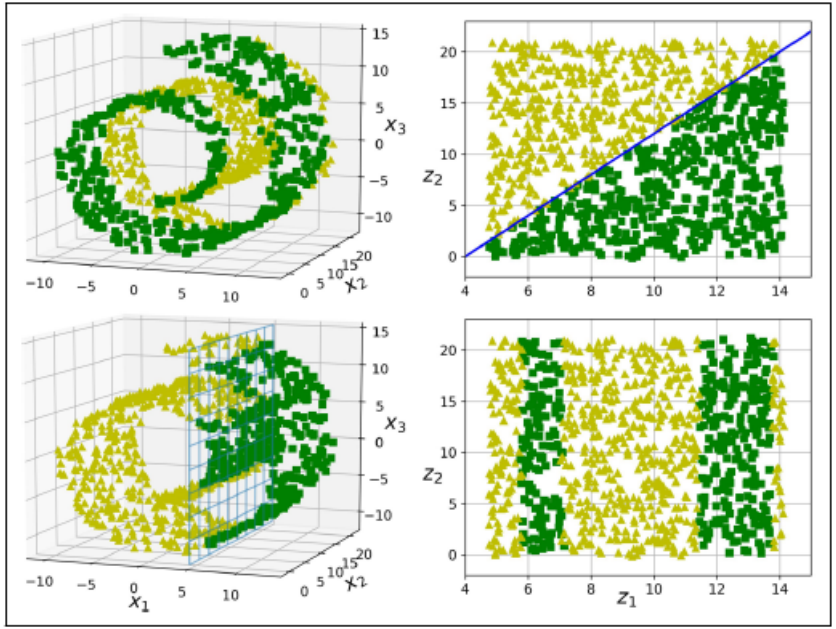

Figure 8-6. The decision boundary may not always be simpler with lower dimensions

Dimensionality Reduction

PCA

Outline

- Introduction: the curse of dimensionality
- **PCA**
- Kernel PCA and LLE

PCA

- Principal Component Analysis (PCA) (most popular)
- Select the axis that **preserves the maximum amount of variance**, as it will most likely lose less information than the other projections.
- Or choose the axis that **minimizes the mean squared distance between the original dataset and its projection onto that axis**.

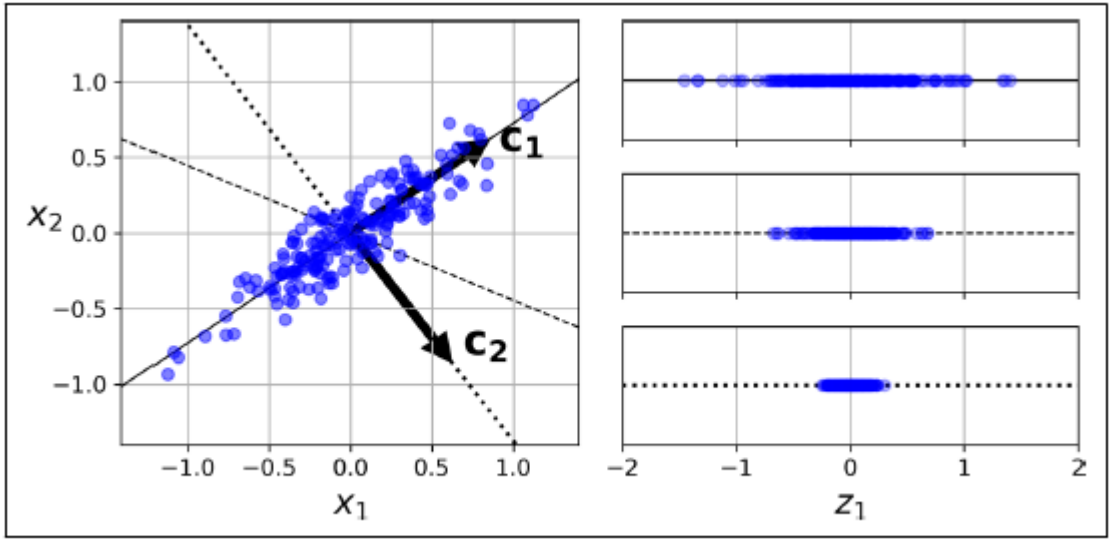

Figure 8-7. Selecting the subspace to project on

SVD

- Singular Value Decomposition (SVD)
- Decompose the training set matrix **X** into matrix multiplication of three matrices $\mathbf{U}\,\Sigma\,\mathbf{V}^T$
- **V** contains the unit vector that define all the principal components

Equation 8-1. Principal components matrix

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

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

Projecting down to d dimensions

- Once the principal components (**d components**) are identified, 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**

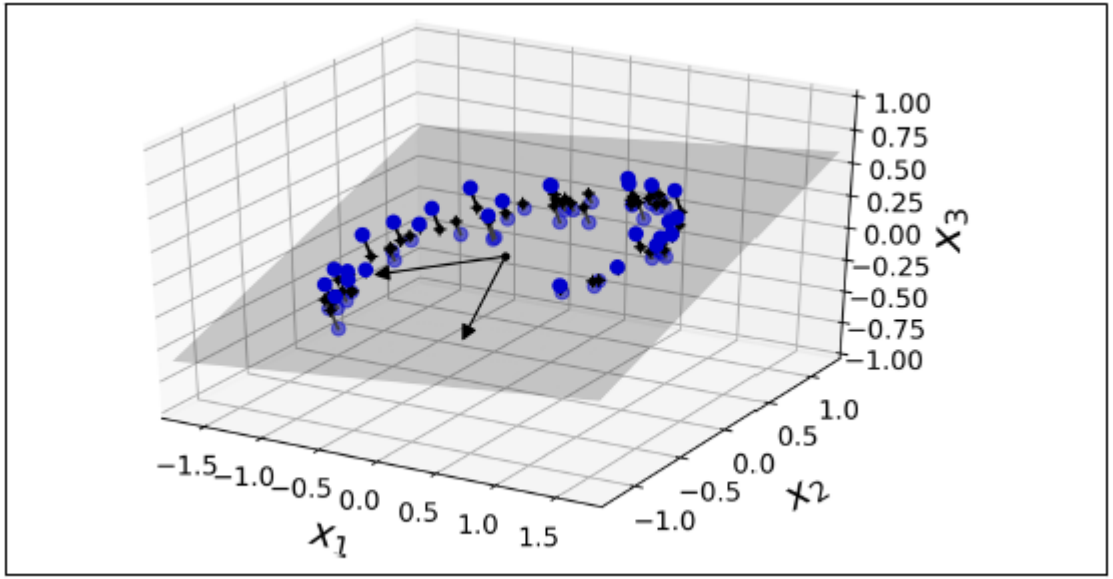

Figure 8-2. A 3D dataset lying close to a 2D subspace

Projecting down to d dimensions

- **Wd** is the first d columns of **V**

Equation 8-2. Projecting the training set down to d dimensions

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

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

Implementation in Scikit-Learn

- PCA in Scikit-Learn
- Explained variance ratio
    - The ratio indicates the proportion of the dataset's variance that lies along each principal component

```
from sklearn.decomposition import PCA

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

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

- Choosing the right number of dimensions

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

Alternatively,

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

Explained variance

- Elbow

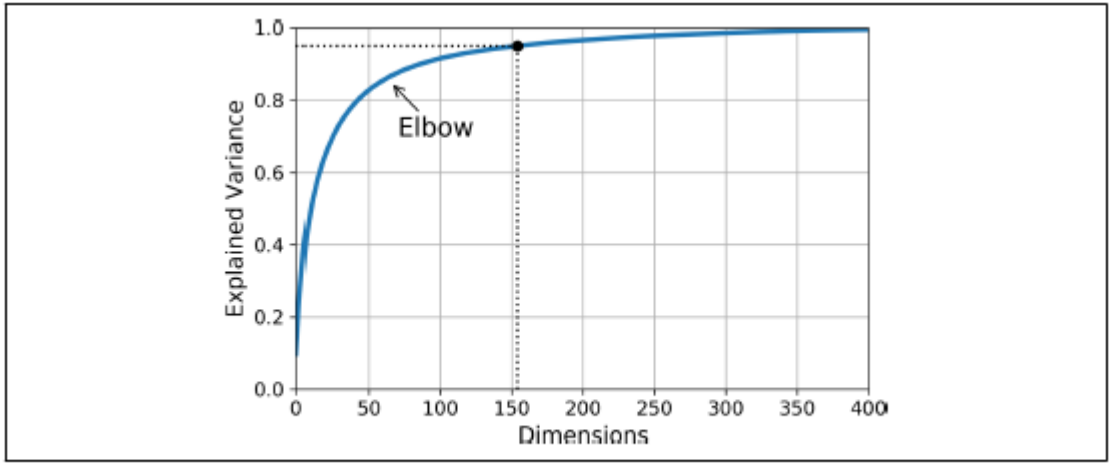

Figure 8-8. Explained variance as a function of the number of dimensions

PCA for compression

- MNIST dataset (preserving 95% of its variance)
- Reduce from 784 features to 150 features
- Dataset is less than 20% of its original size
- You can also decompress from 150 features back to 784. However, there is a chance of projection loss. The squared distance between the original data and the reconstructed data (compressed and then decompressed) is called the reconstruction error.

```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

MNIST dataset compression

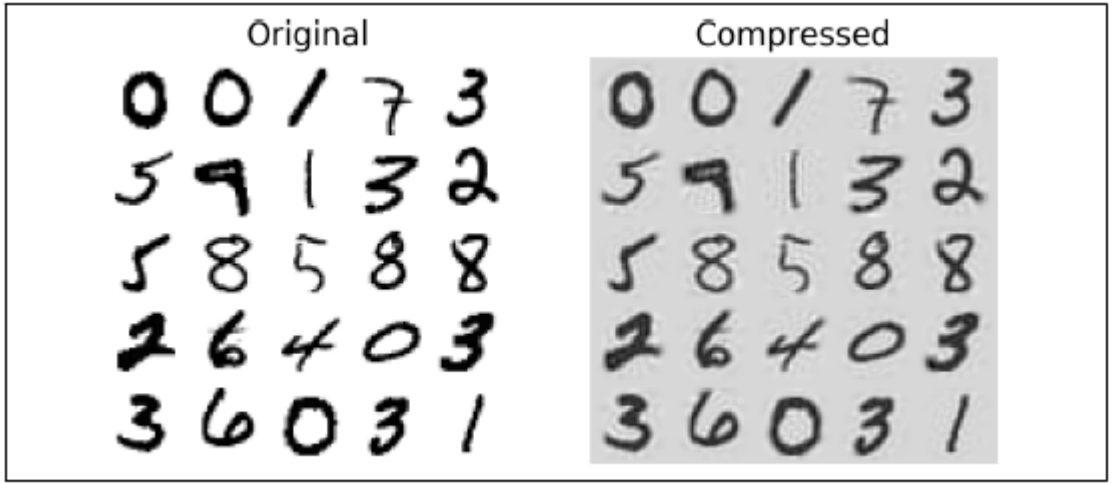

Figure 8-9. MNIST compression that preserves 95% of the variance

Randomized PCA

- Scikit-Learn uses a stochastic algorithm called Ranomized PCA that quickly find an approximation of the first d principal components. (“full”)

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

Incremental PCA

- Incremental PCA (IPCA) allows you to split the training set (X) into min-batches and feed an IPCA algorithm one mini-batch at a time. (apply PCA online, i.e., on the fly)

```
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):
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

Dimensionality Reduction

Kernel PCA and LLE

Outline

- Introduction: the curse of dimensionality
- PCA
- **Kernel PCA and LLE**

Kernel PCA

- Kernel trick, a mathematical technique that implicitly maps instances into a very high-dimensional space (called the feature space), enabling nonlinear classification and regression with SVM.
- A linear decision boundary in high-dimensional space corresponds to a complex nonlinear decision boundary in the original space.

```
from sklearn.decomposition import KernelPCA

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

Kernel PCA

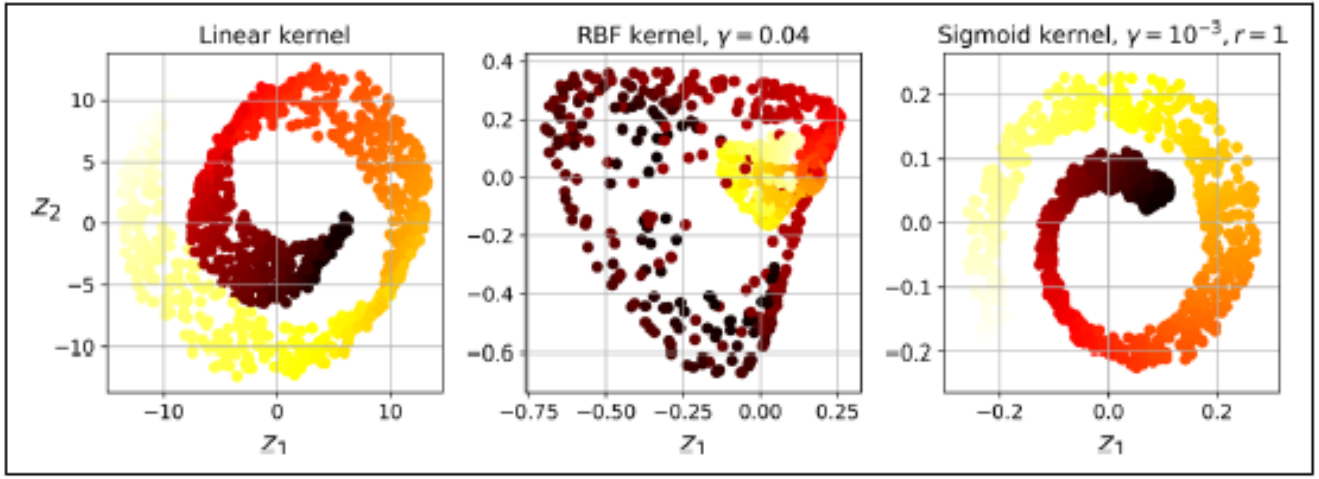

Figure 8-10. Swiss roll reduced to 2D using kPCA with various kernels

Selecting a Kernel

- Two step pipeline
  - Reducing dimensionality using kPCA
  - Logistic regression for classification
  - Use GridSearchCV to find the best kernel and gamma value for kPCA

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

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

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

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

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

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

Selecting a Kernel

- Yield the lowest reconstruction error

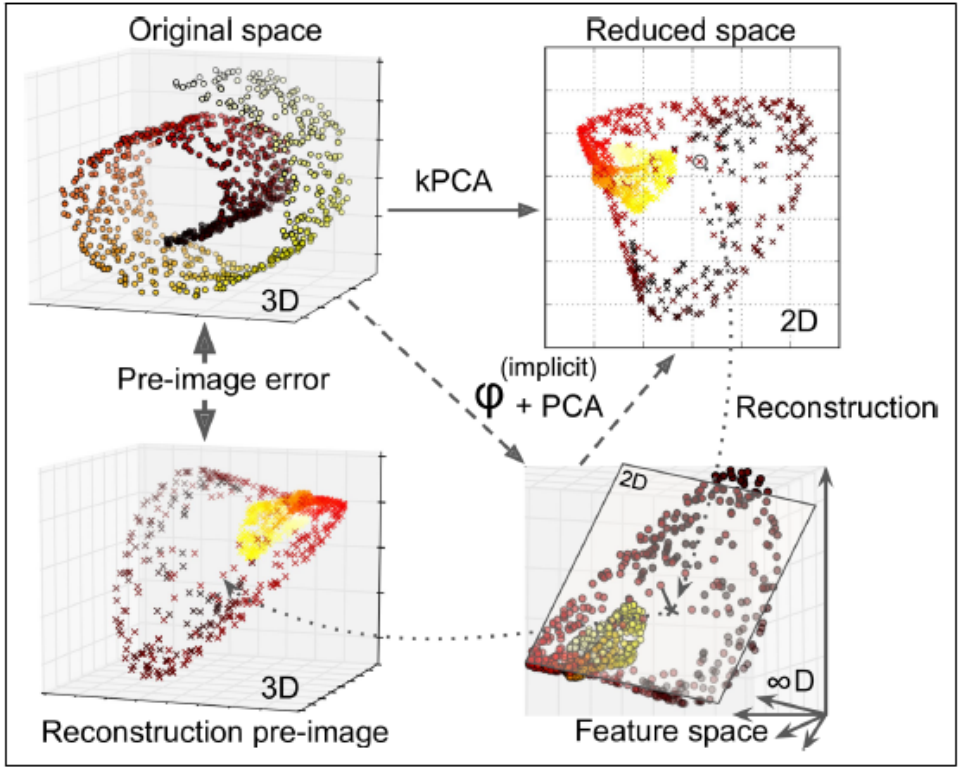

Figure 8-11. Kernel PCA and the reconstruction pre-image error

LLE

- Locally Linear Embedding (LLE)
- It is a Manifold Learning technique that does not rely on projections, like the previous algorithms do. In a nutshell, LLE works by first measuring how each training instance linearly relates to its closest neighbors (c.n.), and then looking for a low-dimensional representation of the training set where these local relationships are best preserved.

```
from sklearn.manifold import LocallyLinearEmbedding

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

Other dimensionality reduction techniques

- Random Projection (sklear.random_projection)
- Multidimensional Scaling (MDS)
- Isomap
- t-Distributed Stochastic Neighbor Embedding (t-SNE)
- Linear Discriminant Analysis (LDA)

.

Figure 8-13 shows the results of a few of these techniques.

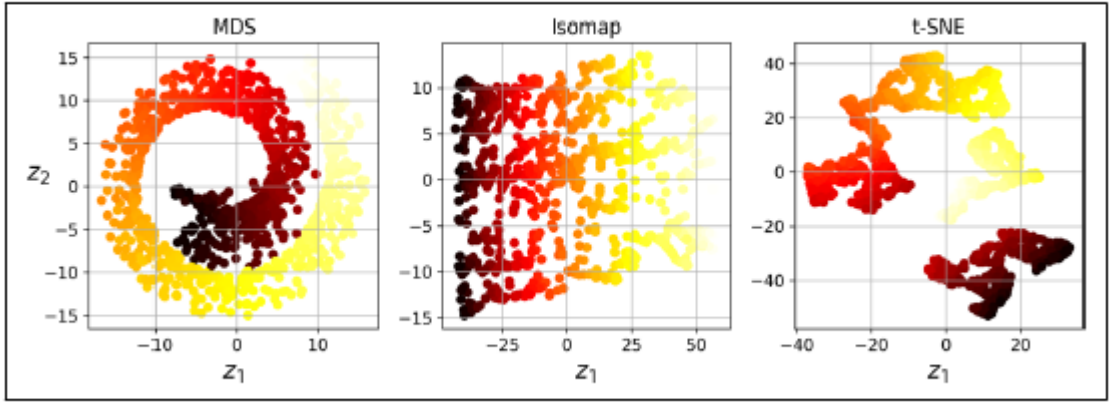

Figure 8-13. Using various techniques to reduce the Swill roll to 2D