Dimensionality Reduction

01 Curse of Dimensionality

Many characteristics slow down training and make it difficult to find a good solution

=> Dimension reduction speeds up training and is useful for data visualization

- **High-dimensional Dataset**

  1) A lot of space = new samples are also likely to stay away from training samples

  2) Predictions are more unstable than at low dimensions

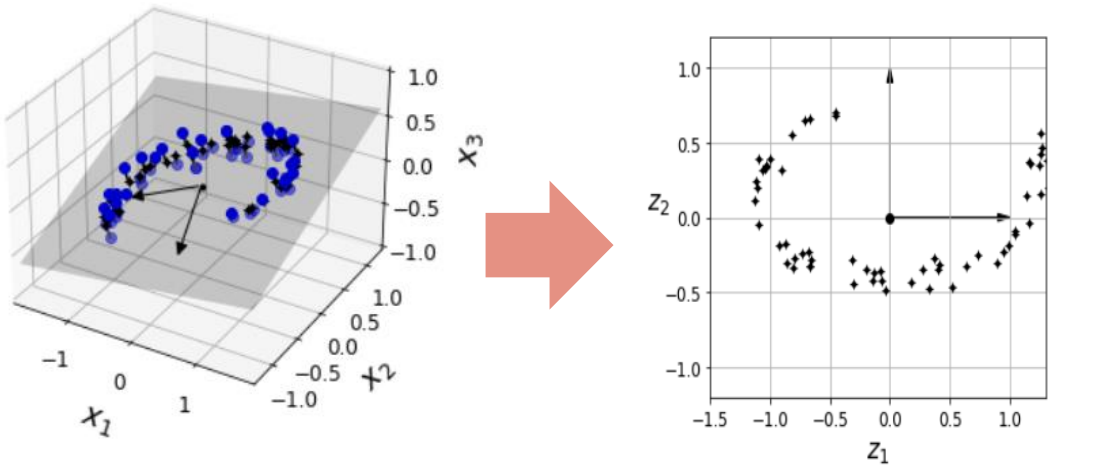
  => The larger dimension, the greater risk of overfitting

02 How to approach dimensionality reduction

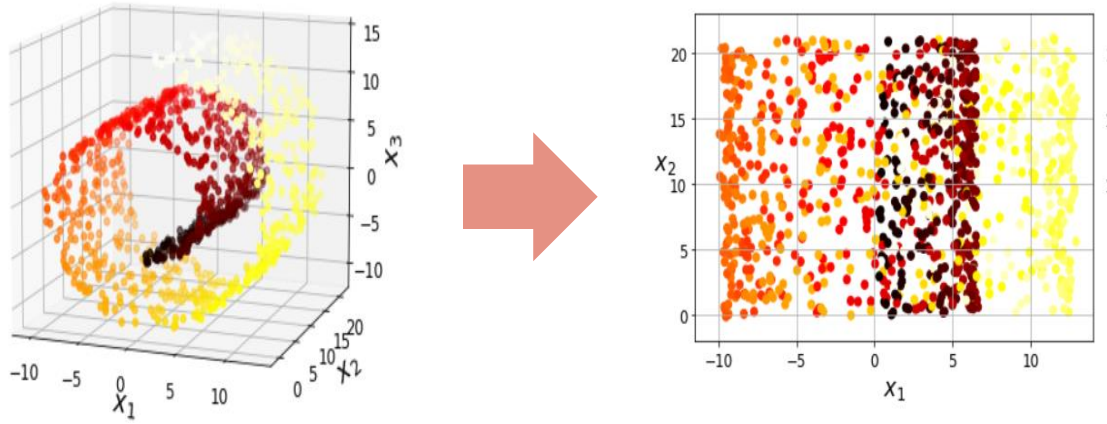
1) Projection

: Training samples are not uniformly spread across all dimensions
= Exists in low-dimensional subspace in high-dimensional space

- Project training samples vertically



- Swiss Roll

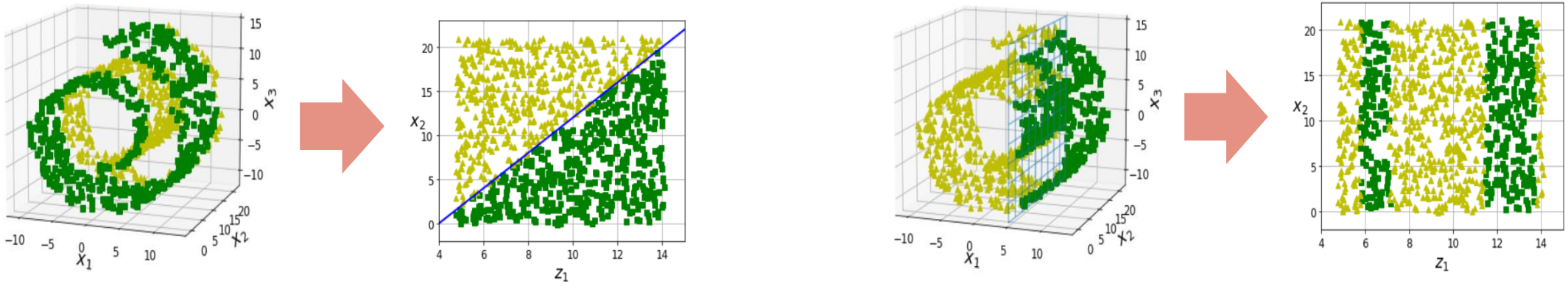


02 How to approach dimensionality reduction

2) Manifold : 2D Manifolds are curved or twisted in high-dimensional space

- Manifold Learning : Many dimensionalities reduction algorithms work by modeling manifolds where training samples are placed

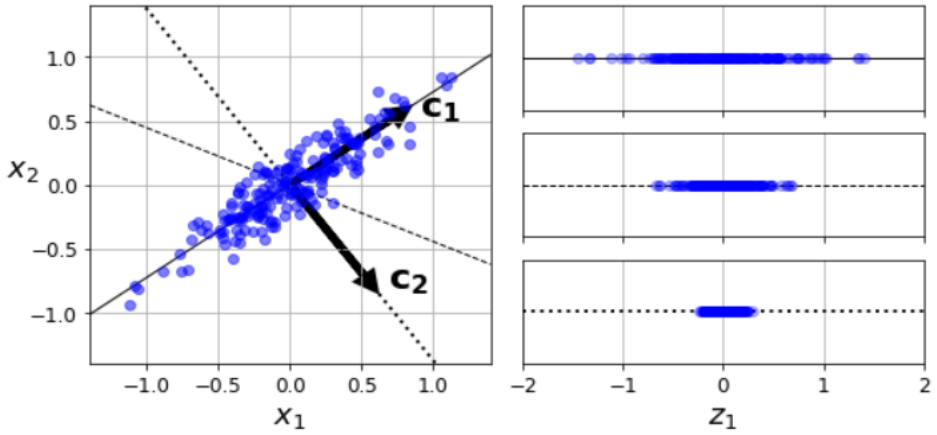
- Manifold Assumption : The real high-dimensional dataset is closer to a lower-dimensional method



⇒ Reducing dimension of training set speeds up training
⇒ but does not always make it a better or simpler solution : depends on dataset

03 PCA : Principal Component Analysis

- **Distributed Preservation**



An axis that minimizes the mean square distance between the original dataset and the projected one should be selected

- **Principal Component**

$X = U \sum V^T$ 에서 V가 주성분

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

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

svd() is used to extract the principal components of the training set

03 PCA

- **Projecting in d-dimensions**

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

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

Reduce the dimension of dataset to d dimensions by projecting it onto hyperplane defined by d principal components

- **Proportion of variance**

```
pca.explained_variance_ratio_
```

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

Variance ratio of the dataset along the axis of each principal component

03 PCA

- **Select the appropriate number of dimensions**

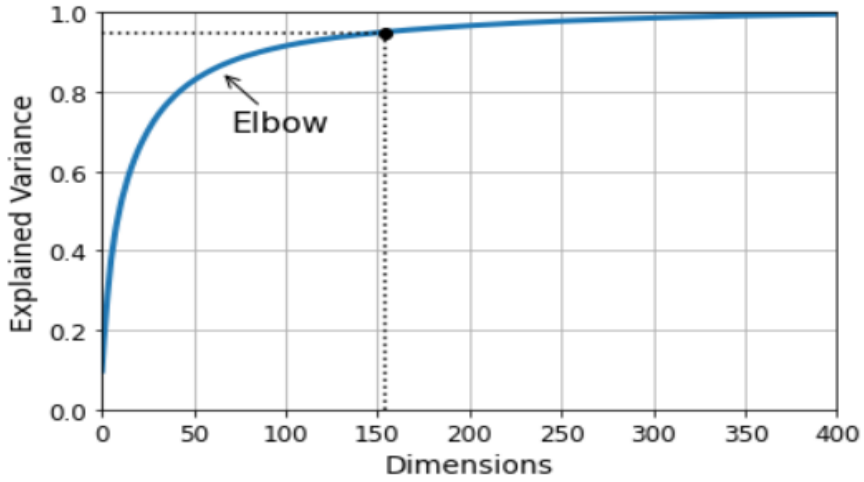
(1) Select the number of dimensions to add until sufficient variance is available

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

```
d
```

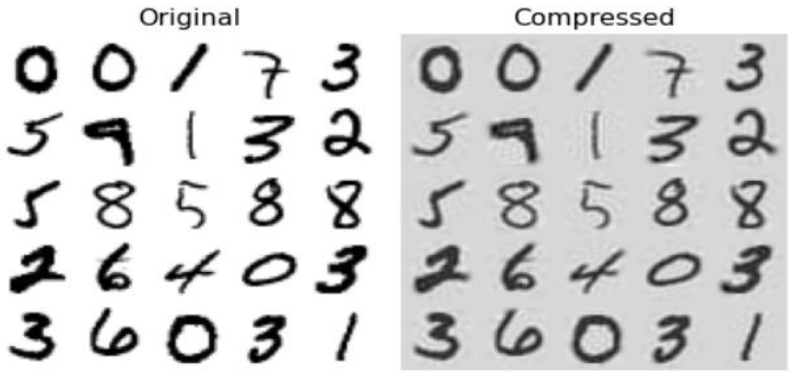
```
154
```

(2) Draw the described variance as a function of the number of dimensions



03 PCA

- PCA for compresssion



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

Lost a certain amount of information in projection,
so it can be recovered to a similar dataset,
not the same

- Random PCA

```
rnd_pca = PCA(n_components=154, svd_solver="randomized", random_state=42)
X_reduced = rnd_pca.fit_transform(X_train)
```

Computational Complexity : $O(m * d^2) + O(d^3)$

Random PCA uses stochastic algorithms
to quickly find approximations
for d principal components

03 PCA

- **Incremental PCA**

Problems with implementation : SVD algorithms require the entire training set to run

=> Develop an incremental PCA

(1) Injected into the incrementalPCA and called partial_fit() per mini-batch

```
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="") # 책에는 없음
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

(2) Use memmap to load data into memory when needed

```
X_mm = np.memmap(filename, dtype="float32", mode="readonly", shape=(m, n))

batch_size = m // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mm)
```

```
IncrementalPCA(batch_size=525, n_components=154)
```

04 Kernel PCA

Kernel PCA enables complex nonlinear transformation for dimensionality reduction

```
from sklearn.decomposition import KernelPCA

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

- Kernel Selection and Hyper parameter Tuning

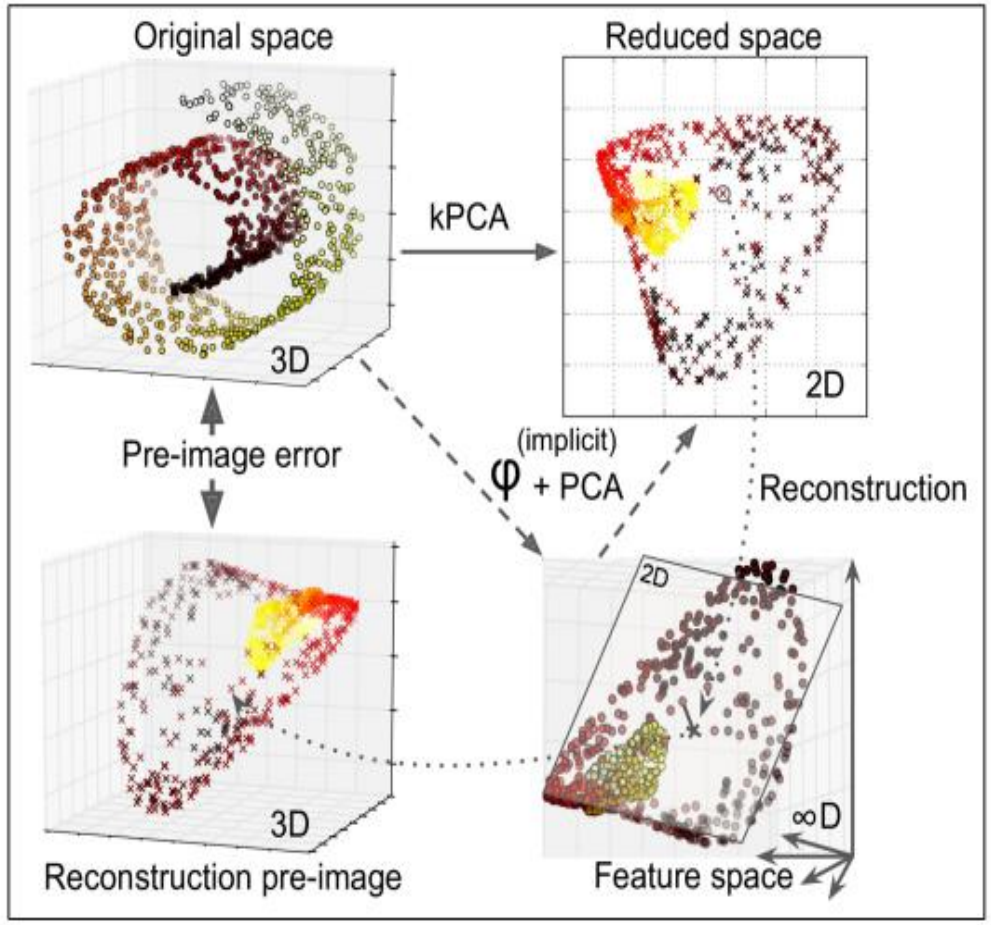
```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

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)
```

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

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

To obtain the highest classification accuracy,
Use GridSearch to find the best kernel
And gamma parameters of kPCA

04 Kernel PCA




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

```
from sklearn.metrics import mean_squared_error

mean_squared_error(X, X_preimage)
```

```
9.183245059705947e-27
```

Select the kernel and hyper parameters that create the lowest reconstruction error through the kernel trick

05 LLE : Locally linear embedding

Non-projection-dependent manifold learning as a technique for nonlinear dimensionality reduction

- **How LLE works**

1) Find the nearest neighbor k for training sample x

2) Reconfigure x as a linear function for neighbors

$$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^{m} \left(\mathbf{x}^{(i)} - \sum_{j=1}^{m} w_{i,j} \mathbf{x}^{(j)} \right)^2$$

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

3) Map the training samples to d-dimensional to preserve the local linear relationship between training samples

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

- **Computational Complexity**

1) Find the nearest neighbor k
: O(mlog(m) nlog(k))

2) Weight optimization : $O(mk^3)$

3) Create a low-dimensional Representation
: $O(dm^2)$

THANK YOU