


A background image of a kitchen. At the top, a wooden shelf holds several teapots and cups in various colors like blue, red, and white. Below the shelf, a light-colored wall features a small, square-framed picture. To the left, a wooden cutting board with a dark, irregular shape is visible. A wooden handle, possibly for a knife or spoon, hangs on the right side of the wall.

**Machine Learning**

# **Dimensionality Reduction**

김선녕(sykim.lecture@gmail.com)



## 차원의 저주

주성분 분석(PCA)

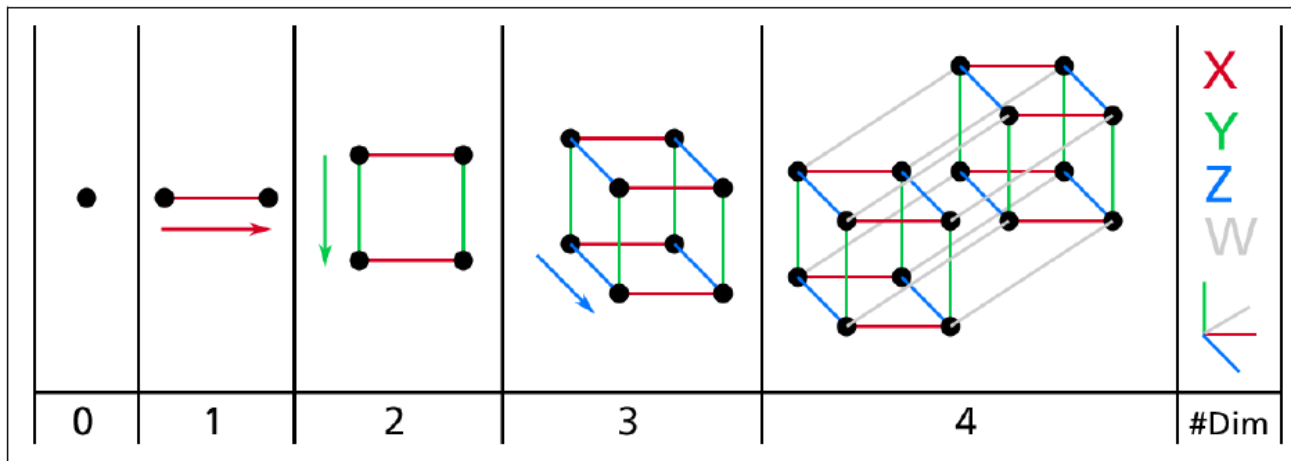
선형판별 분석(LDA)

커널 PCA(KPCA)

기타 차원축소 기법

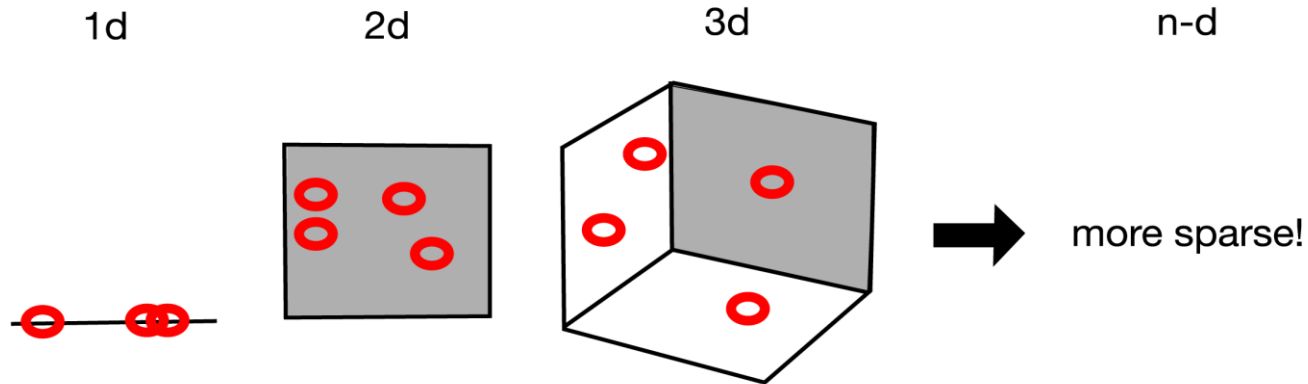
# 차원의 저주(Curse of Dimensionality)

- 특성선택(*feature selection*)
  - 가지고 있는 특성 중에서 훈련에 가장 유용한 특성을 선택
- 특성추출(*feature extraction*)
  - 특성을 결합하여 더 유용한 특성을 만든다.
  - 저장공간을 줄이거나 학습 알고리즘의 계산 효율성 향상
  - 차원의 저주 문제를 감소시켜 예측성능을 향상하기도 한다
- 훈련샘플 각각 수천, 수백만 개등 수 많은 특성은 훈련을 느리게 할 뿐만 아니라, 좋은 해결을 찾기 어렵다.
- 훈련 세트의 차원이 클수록 과대적합 위험이 크다
  - 점,선,정사각형,정육면체,테서랙트(0차원에서 4차원까지의 초입방체)



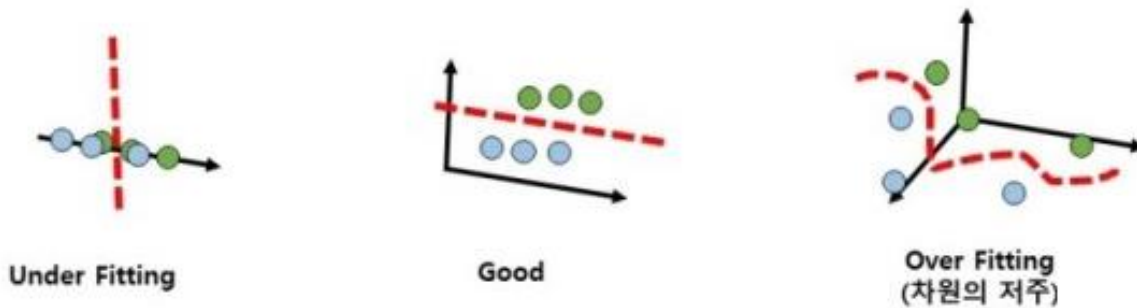
# 두 점 사이의 거리

4



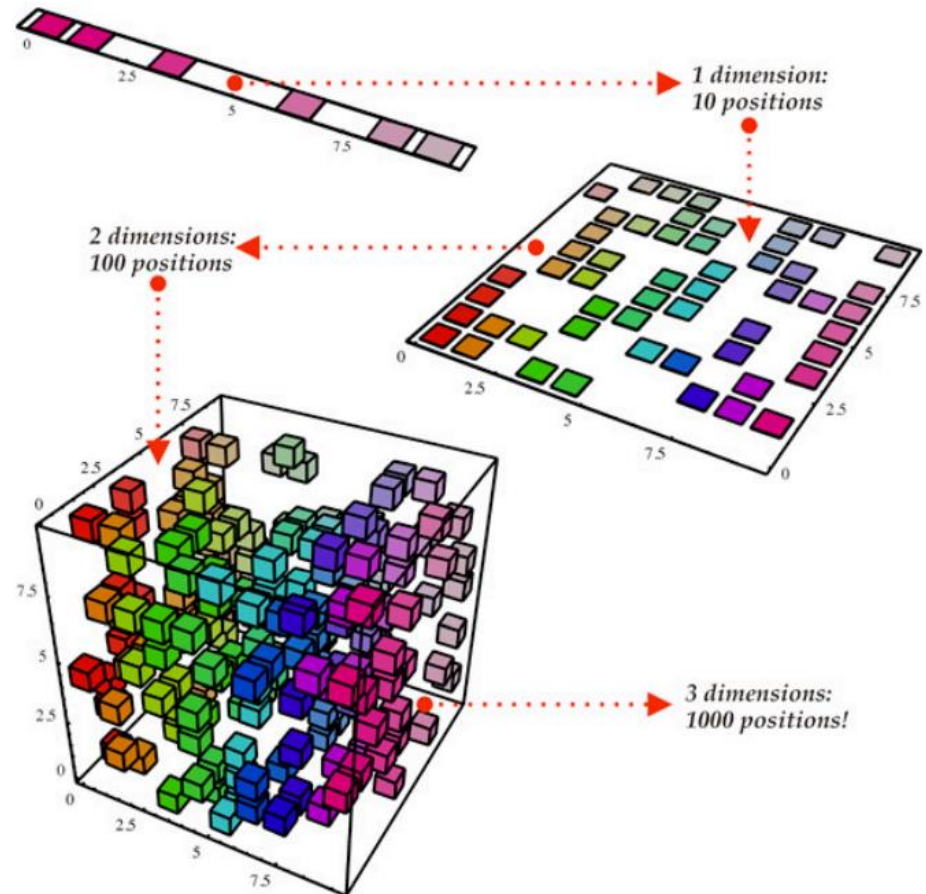
Ref: <https://journals.plos.org/plosone/article/figure?id=10.1371/journal.pone.0179180.g002>

- 단위 면적에서 두 점 사이의 평균거리 : 약 0.52
- 3차원 큐브에서 임의의 두 점의 평균거리 : 약 0.66
- 1,000,000차원의 초 입방체의 두 점의 평균거리 : 약 408.25



- 고차원은 많은 공간과 데이터셋은 매우 희박. 때문에 예측도 불안정 : 차원이 클수록 과대적합

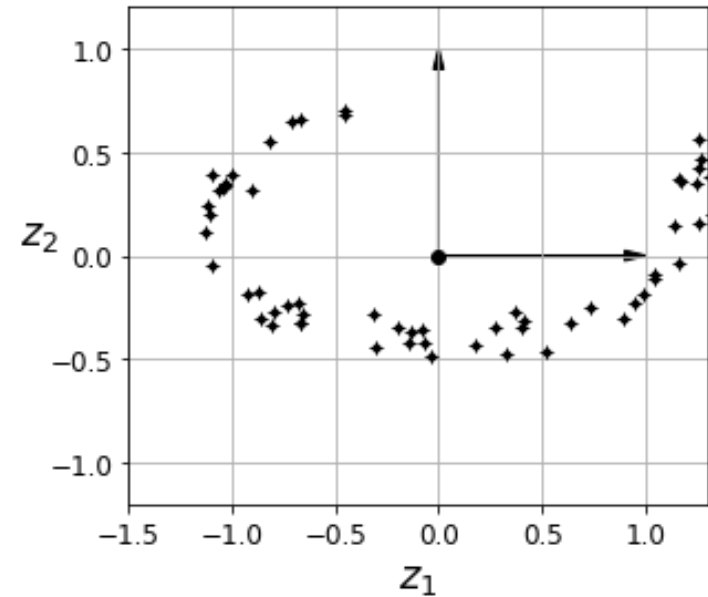
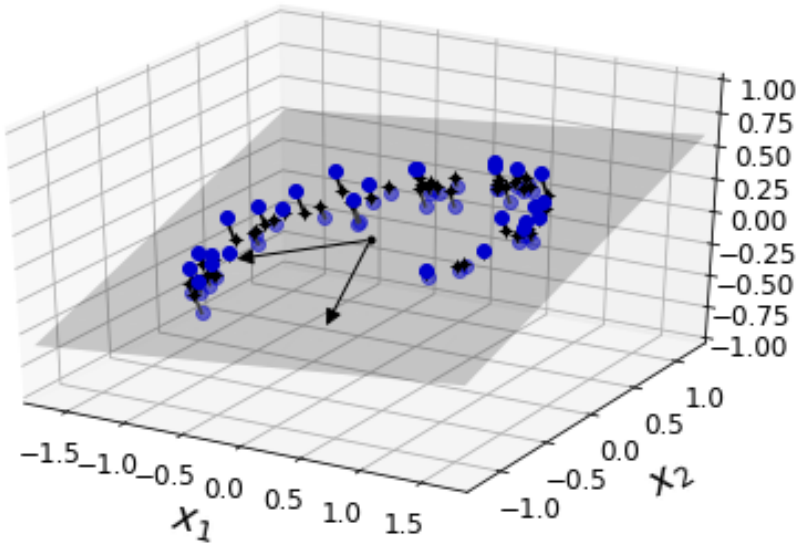
- 차원의 저주를 해결하는 해결책 하나는 훈련 샘플의 밀도가 충분히 높아질 때까지 훈련 세트의 크기를 늘리는 것
  - 불행하게도 일정 밀도에 도달하기 위해 필요한 훈련 샘플 수는 차원 수가 커짐에 따라 기하급수적으로 늘어난다
- 훈련샘플 수 많은 특성은 훈련을 느리게 할 뿐만 아니라, 좋은 해결을 찾기 어렵다.



# 투영(Projection)

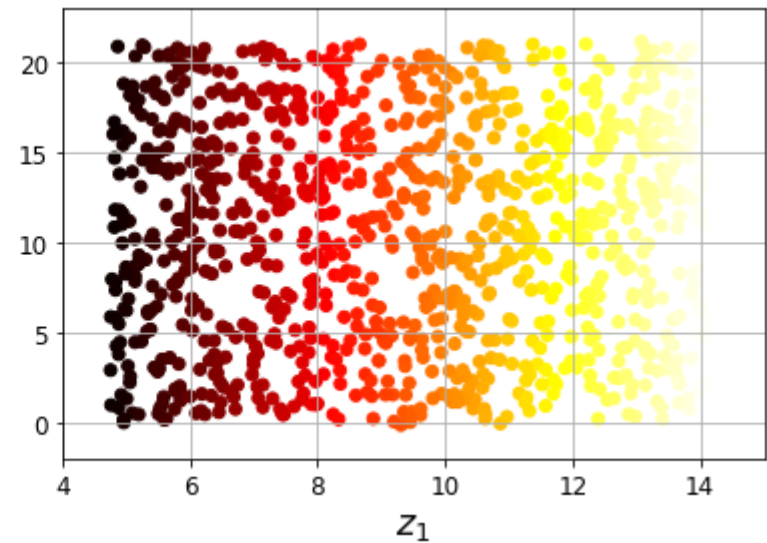
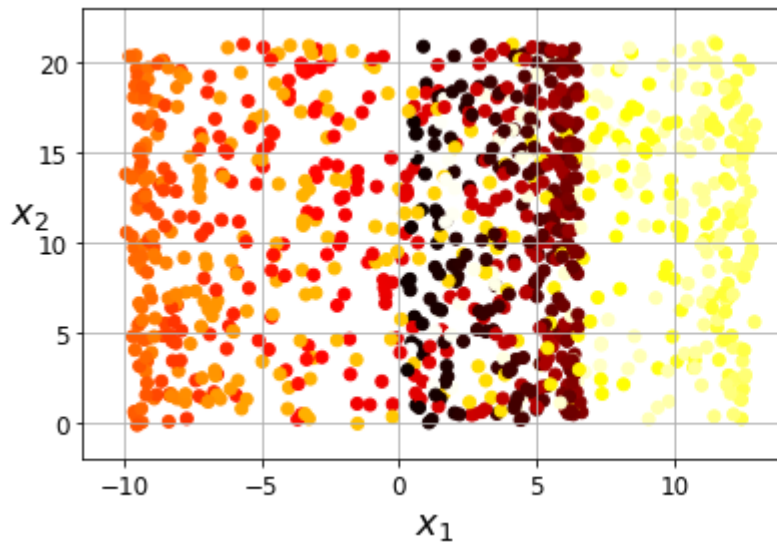
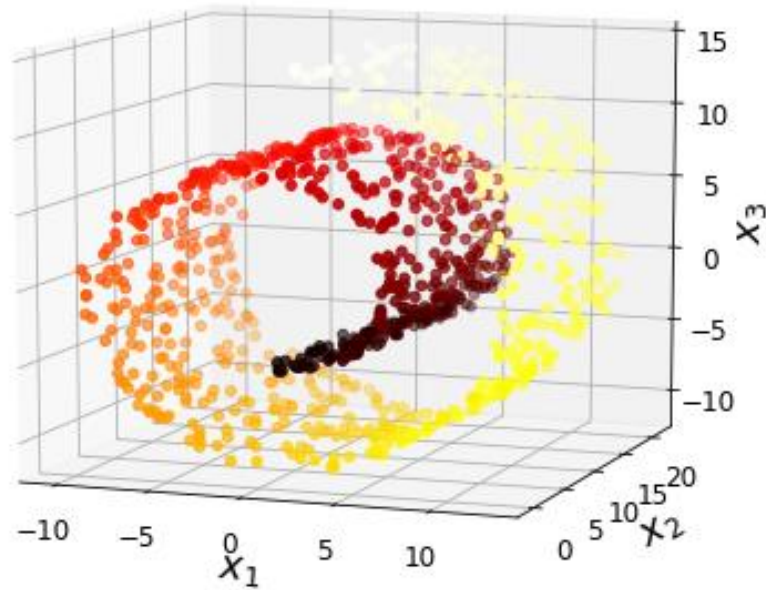
6

- 차원을 감소시키는 두 가지 주요한 접근법 : 투영과 매니폴드 학습
- $3D \rightarrow 2D$



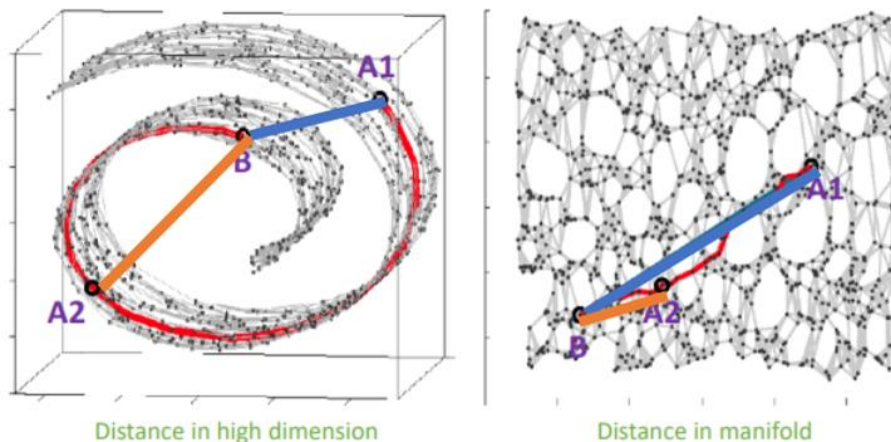
# 투영(Projection) - Swiss roll dataset

7





- 매니폴드 : 고차원 공간에 내재한 저차원 공간
- 고차원 공간 데이터간 관계를 저차원공간에서 효율적으로 유지하는 것
  - 자동차 위치 데이터가  $x$  이면  $x = (\text{위도}, \text{경도}, \text{고도})^T$  의 3차원을  
 $x = (\text{기준점에서의 거리})^T$  의 1차원으로 표현
    - 자동차 데이터는 무작위로 분포하지 않고, 도로라는 1차원 비선형 공간에 분포
- $d$ 차원 매니폴드는 국부적으로  $d$ 차원 초평면으로 보일 수 있는  $n$ 차원 공간의 일부( $d < n$ )
  - swiss roll : 2D 매니폴드의 한 예로서 고차원 공간에서 휘어지거나 뒤틀린 2D모양  
 $d = 2, n = 3$



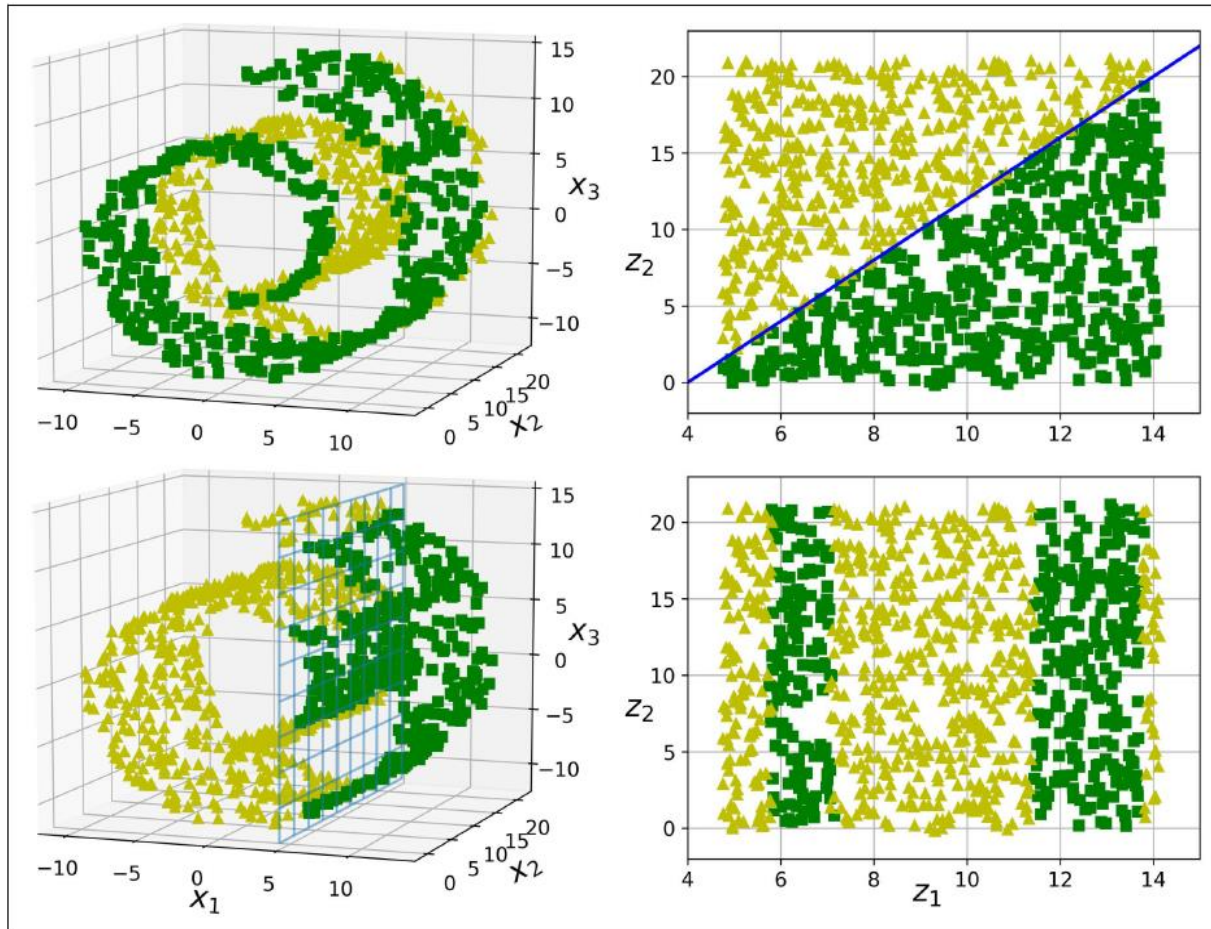
Ref : 기계학습 - 한빛아카데미






# 매니폴드 학습(Manifold Learning)

- 훈련 세트의 차원을 감소시키면 훈련속도는 빨라지지만 항상 더 낮거나 간단한 솔루션이 되는 것은 아니다.
  - 전적으로 데이터셋에 달려있다.



결정경계  
복잡 : 단순

결정경계  
단순 : 복잡

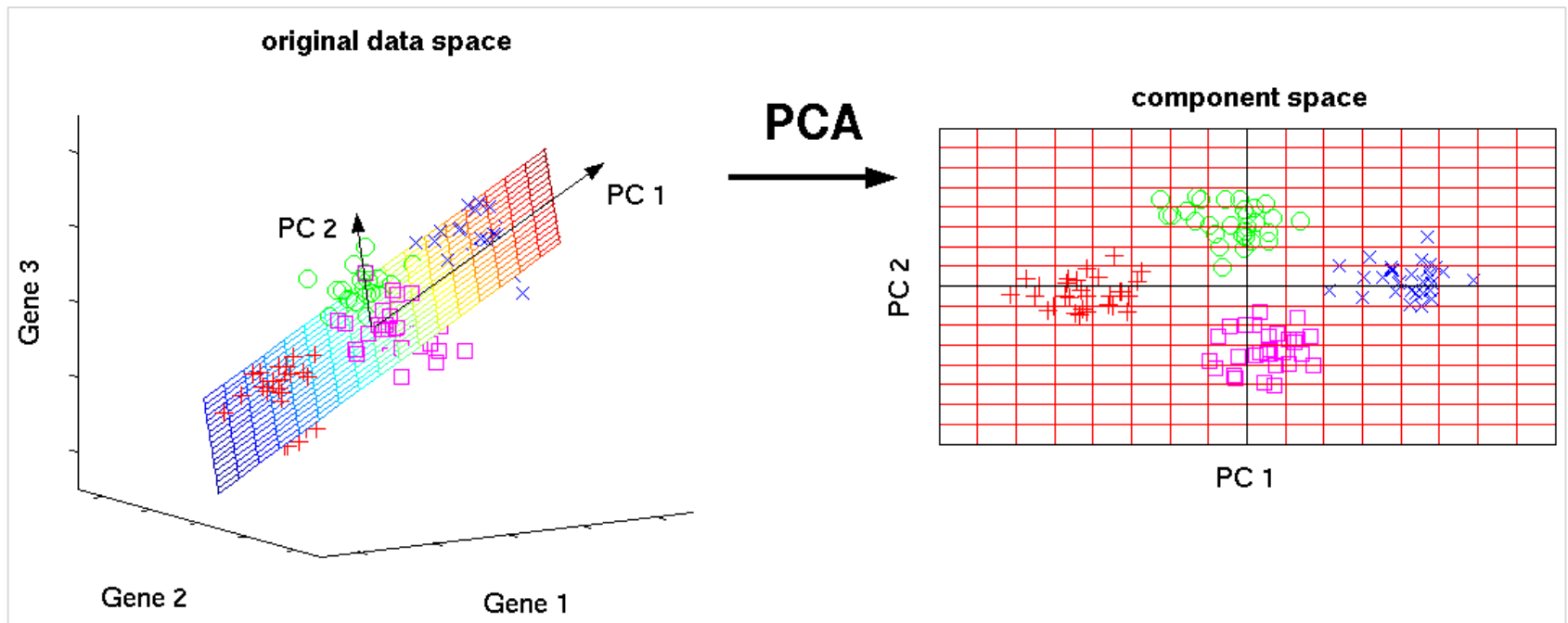


차원의 저주  
주성분 분석(PCA)  
선형판별 분석(LDA)  
커널 PCA(KPCA)  
기타 차원축소 기법

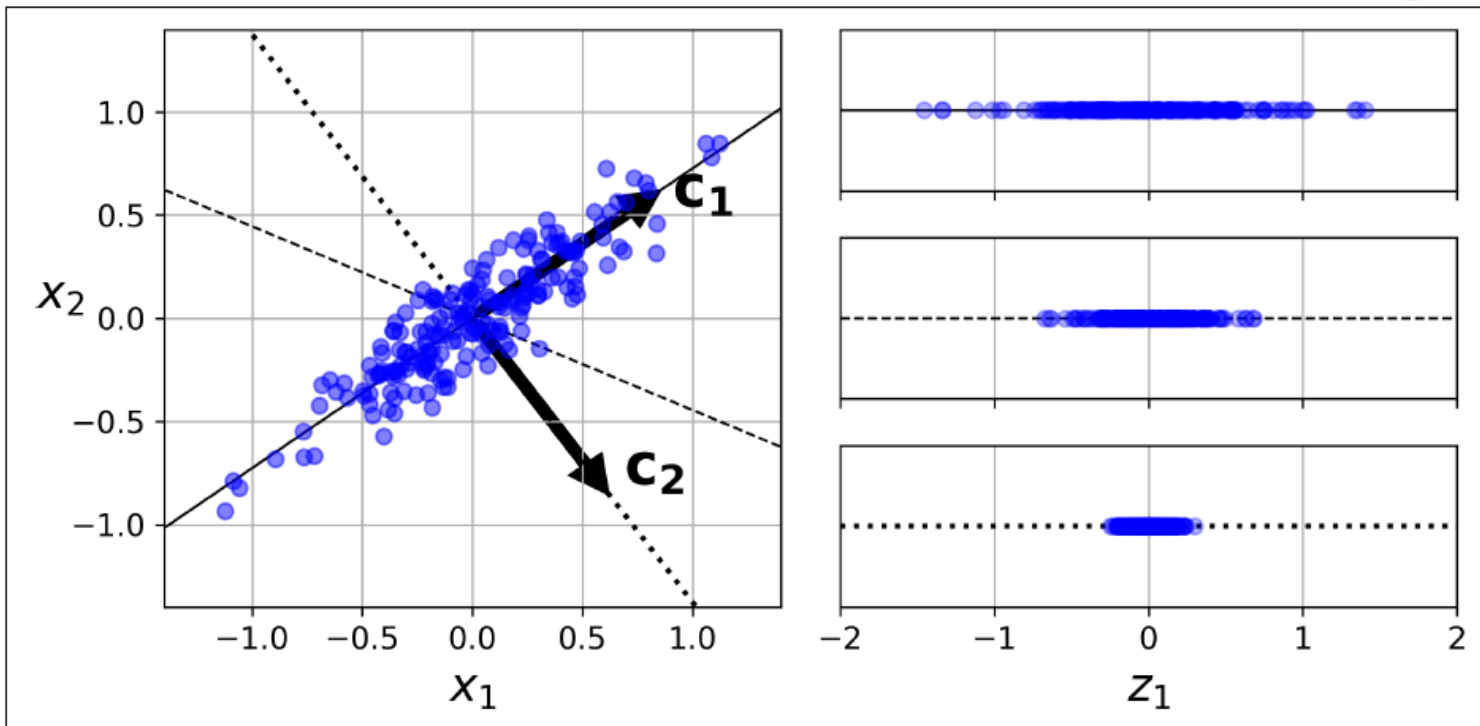
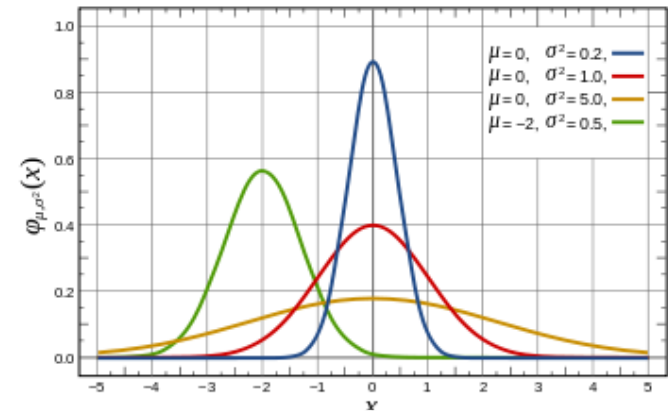
# 주성분 분석(PCA)이란?

11

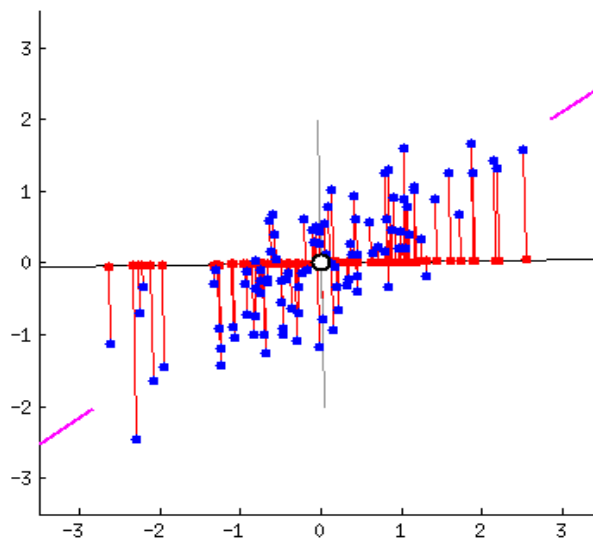
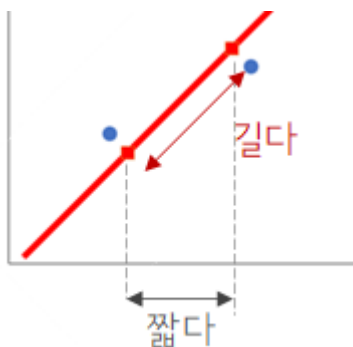
- 고차원적 데이터(*high dimensional data*)를 줄여주는 가장 인기 있는 차원 축소 알고리즘
- 비지도 선형 변환 기법
- 데이터들의 퍼짐의 주성분(*Principal Component, PC*)을 찾는 방법



- 분산이 최대로 보존되는 축 선택
  - 원본 데이터셋과 투영된 것 사이의 평균 제곱 거리 최소화
- 실선에 투영( $C_1$ ) : 분산을 최대로 보존



- 데이터의 **분산(variance)**을 최대한 보존하면서 서로 직교하는 새 기저(축)를 찾아, 고차원 공간의 표본들을 선형 연관성이 없는 저차원 공간으로 변환



- 공분산(*covariance*) : 두 변수 사이의 상관관계. 각 특징(feature)의 유사성
- 공분산 행렬(*covariance matrix*)
  - $d \times d$  차원의 대칭 행렬
  - $d$  : 데이터셋에 있는 차원 개수
  - $\mu_j, \mu_k$  : 특성  $j$ 와  $k$ 의 평균샘플
  - $\sigma$  : 공분산
  - $\Sigma$  : 공분산 행렬

$$\sigma_{jk} = \frac{1}{n} \sum_{i=1}^n (x_j^{(i)} - \mu_j)(x_k^{(i)} - \mu_k)$$

$a_1$ 의 분산

$$\Sigma = \begin{bmatrix} \sigma_{11}^2 & \sigma_{12}^2 & \sigma_{13}^2 \\ \sigma_{21}^2 & \sigma_{22}^2 & \sigma_{23}^2 \\ \sigma_{31}^2 & \sigma_{32}^2 & \sigma_{33}^2 \end{bmatrix}$$

(세개의 특성인 경우)

$a_1$  과  $a_3$ 의 상관관계



- 정방행렬을 사용하여 선형변환할 때 크기는 변하더라도 방향이 변하지 않는 벡터를 고유벡터라 하고, 이때 크기 변화의 비율을 고윳값이라 한다.

$A$  :  $n$ 차 정방행렬

$v$  : 고유벡터(*eigenvector*)

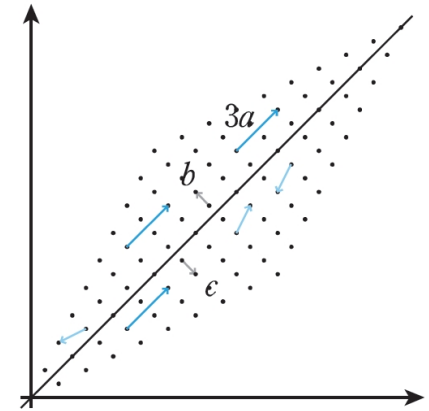
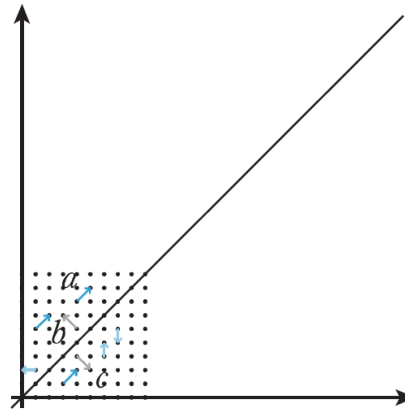
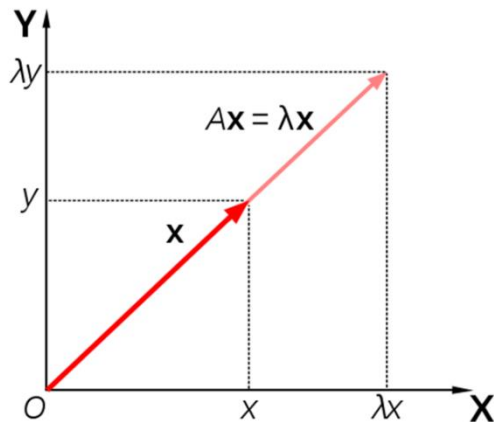
$\lambda$  : 고윳값(*eigenvalue*)

$$Av = \lambda v \quad (v \neq 0)$$

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = \lambda \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$$

$\uparrow$   
 $PC1, PC2, \dots, PCn$

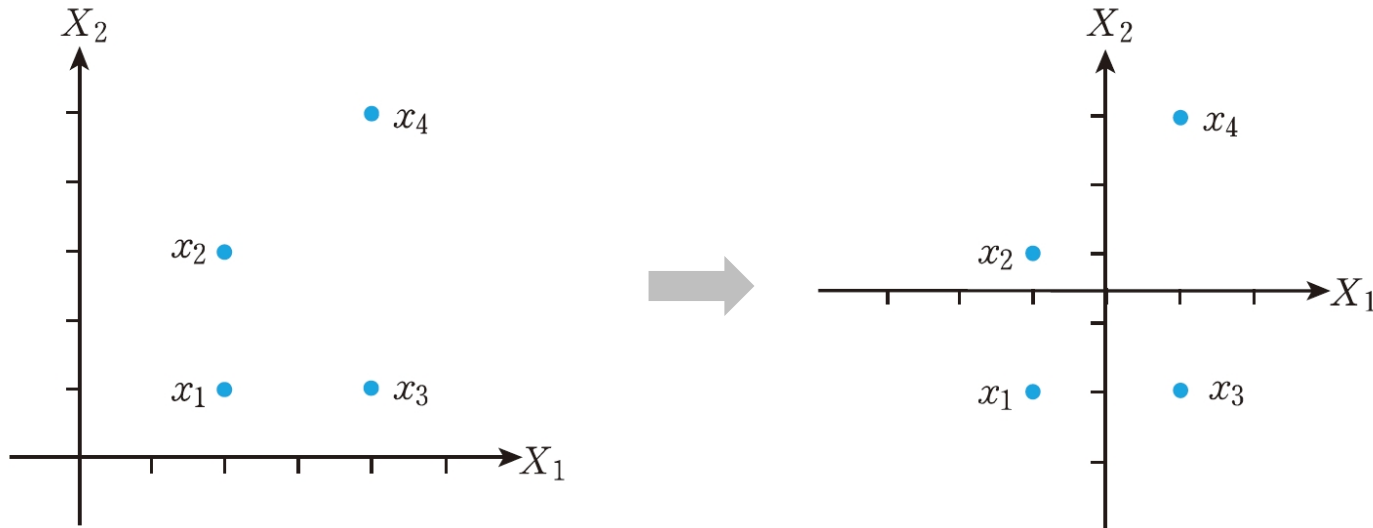
$\downarrow$   
*Magnitude of spread*



## 데이터셋을 표준화 전처리

16

- 각 데이터셋의 평균을 0으로 만들어 줌( $mean\ vector = 0$ )



① 데이터셋을 표준화 전처리

- 각 데이터셋의 평균을 0으로 만들어 줌( $mean\ vector = 0$ )

② 공분산 행렬( $covariance\ matrix$ )을 구성

$$\begin{bmatrix} cov(x_1, x_1) & cov(x_1, x_2) & \cdots & cov(x_1, x_k) \\ cov(x_2, x_1) & cov(x_2, x_2) & \cdots & cov(x_2, x_k) \\ \vdots & \vdots & \ddots & \vdots \\ cov(x_k, x_1) & cov(x_k, x_2) & \cdots & cov(x_k, x_k) \end{bmatrix}$$

③ 공분산 행렬의 고윳값( $eigenvalue$ )과 고유벡터( $eigenvector, PC\ axis$ )를 구한다

$$\begin{bmatrix} cov(x_1, x_1) & cov(x_1, x_2) & \cdots & cov(x_1, x_k) \\ cov(x_2, x_1) & cov(x_2, x_2) & \cdots & cov(x_2, x_k) \\ \vdots & \vdots & \ddots & \vdots \\ cov(x_k, x_1) & cov(x_k, x_2) & \cdots & cov(x_k, x_k) \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_k \end{bmatrix} = \lambda \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_k \end{bmatrix}$$

④ 고윳값을 내림차순으로 정렬하여 고유벡터의 순위를 매긴다

178개의 와인 샘플과 세 가지 와인 유형과 화학 성분을 나타내는 13개의 특성으로 구성

```
import pandas as pd

df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/'
                      'machine-learning-
databases/wine/wine.data', header=None)

df_wine.columns = ['Class label', 'Alcohol', 'Malic acid', 'Ash',
                   'Alcalinity of ash', 'Magnesium', 'Total phenols',
                   'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins',
                   'Color intensity', 'Hue',
                   'OD280/OD315 of diluted wines', 'Proline']

df_wine.head()
```

	Class label	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins	Color intensity	Hue	OD280/OD315 of diluted wines	Proline
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

```

# Splitting the data into 70% training and 30% test subsets
from sklearn.model_selection import train_test_split
X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    stratify=y, random_state=0)

# 표준화 전처리
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)
# 고윳값
import numpy as np
cov_mat = np.cov(X_train_std.T) # 표준화 전처리된 훈련 데이터셋의 공분산 행렬 계산
eigen_vals, eigen_vecs = np.linalg.eig(cov_mat) # 고윳값 분해

print('\nEigenvalues \n%s' % eigen_vals) # 13개
print('\nEigenvectors \n%s' % eigen_vecs)

```

Eigenvalues

```

[4.84274532  2.41602459  1.54845825  0.96120438  0.84166161  0.6620634
 0.51828472  0.34650377  0.3131368   0.10754642  0.21357215  0.15362835
 0.1808613]

```

Eigenvectors

**[[-1.37242175e-01 5.03034778e-01 -1.37748734e-01 -3.29610003e-03  
-2.90625226e-01 2.99096847e-01 7.90529293e-02 -3.68176414e-01  
-3.98377017e-01 -9.44869777e-02 3.74638877e-01 -1.27834515e-01  
2.62834263e-01]**

**[ 2.47243265e-01 1.64871190e-01 9.61503863e-02 5.62646692e-01  
8.95378697e-02 6.27036396e-01 -2.74002014e-01 -1.25775752e-02  
1.10458230e-01 2.63652406e-02 -1.37405597e-01 8.06401578e-02  
-2.66769211e-01]**

**[-2.54515927e-02 2.44564761e-01 6.77775667e-01 -1.08977111e-01  
-1.60834991e-01 3.89128239e-04 1.32328045e-01 1.77578177e-01  
3.82496856e-01 1.42747511e-01 4.61583035e-01 1.67924873e-02  
-1.15542548e-01]**

⋮        ⋮        ⋮

**[-2.96696514e-01 3.80229423e-01 -7.06502178e-02 -1.67682173e-01  
-1.28029045e-01 1.38018388e-01 8.11335043e-04 5.60817288e-03  
5.17278463e-01 1.21112574e-02 -5.42532072e-01 3.87395209e-02  
3.67763359e-01]]**

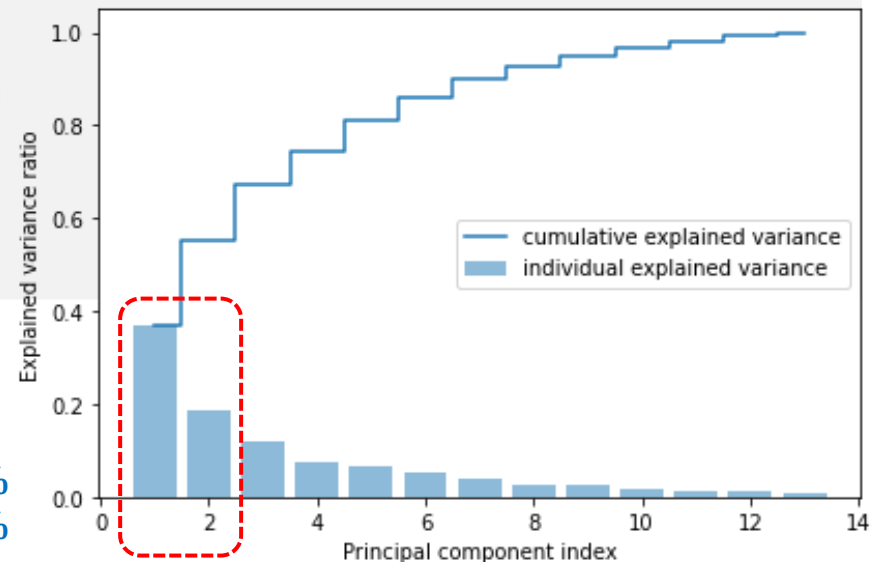


```
#  $\frac{\lambda_j}{\sum_{j=1}^d \lambda_j}$ 
tot = sum(eigen_vals)
var_exp = [(i / tot) for i in sorted(eigen_vals, reverse=True)]
cum_var_exp = np.cumsum(var_exp)

import matplotlib.pyplot as plt

plt.bar(range(1, 14), var_exp, alpha=0.5, align='center',
        label='individual explained variance')
plt.step(range(1, 14), cum_var_exp, where='mid', label='cumulative explained
variance')
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal component index')
plt.legend(loc='best')
plt.tight_layout()
plt.show()
```

첫번째 주성분 : 40%  
 처음 두 개의 주성분 : 60%



```
# Make a list of (eigenvalue, eigenvector) tuples
eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i])
                for i in range(len(eigen_vals))]

# 정렬 : Sort the (eigenvalue, eigenvector) tuples from high to low
eigen_pairs.sort(key=lambda k: k[0], reverse=True)

# 상위 두 개의 고유 벡터 13x2 차원의 투영행렬 W를 생성
w = np.hstack((eigen_pairs[0][1][:, np.newaxis],
                eigen_pairs[1][1][:, np.newaxis]))
print('Projection Matrix W:\n', w)
```

Projection Matrix W:

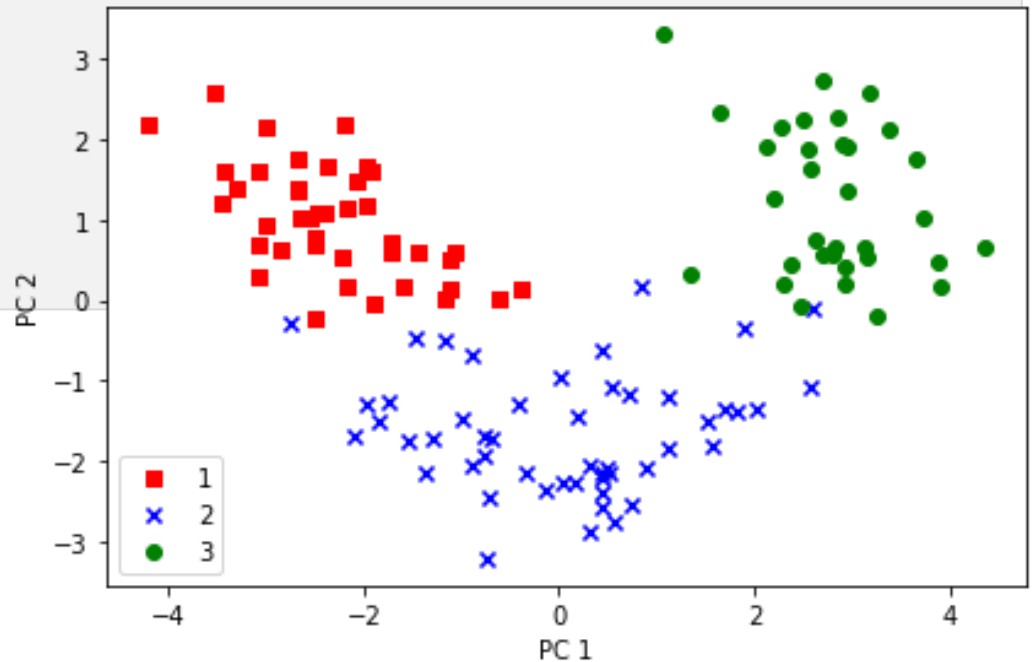
```
[[[-0.13724218  0.50303478]
 [ 0.24724326  0.16487119]
 [-0.02545159  0.24456476]
 [ 0.20694508 -0.11352904]
 [-0.15436582  0.28974518]
 [-0.39376952  0.05080104]
 [-0.41735106 -0.02287338]
 [ 0.30572896  0.09048885]
 [-0.30668347  0.00835233]
 [ 0.07554066  0.54977581]
 [-0.32613263 -0.20716433]
 [-0.36861022 -0.24902536]
 [-0.29669651  0.38022942]]]
```

- 고윳값이 가장 큰  $k$ 개의 고유벡터 선택
  - (예) 최상위 두 개의 고유벡터 선택
- 최상위  $k$ 개의 고유 벡터로 투영 행렬  $W$ 를 만든다
  - (예)  $13 \times 2$ 차원의 투영행렬(*Projection Matrix*)  $W$ 생성

```
# 투영 행렬  $W$  를 사용해서  $d$  차원 입력 데이터셋  $X$  를 새로운  $k$  차원의 특성 부분 공간으로 변환:  $X' = XW$ 
# 124x2 차원의 행렬로 변환된 wine 훈련 세트를 2차원 산점도로 시각화 (70% : 124개)
X_train_pca = X_train_std.dot(w)
colors = ['r', 'b', 'g']
markers = ['s', 'x', 'o']

for l, c, m in zip(np.unique(y_train), colors, markers):
    plt.scatter(X_train_pca[y_train == l, 0],
                X_train_pca[y_train == l, 1],
                c=c, label=l, marker=m)

plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.tight_layout()
plt.show()
```

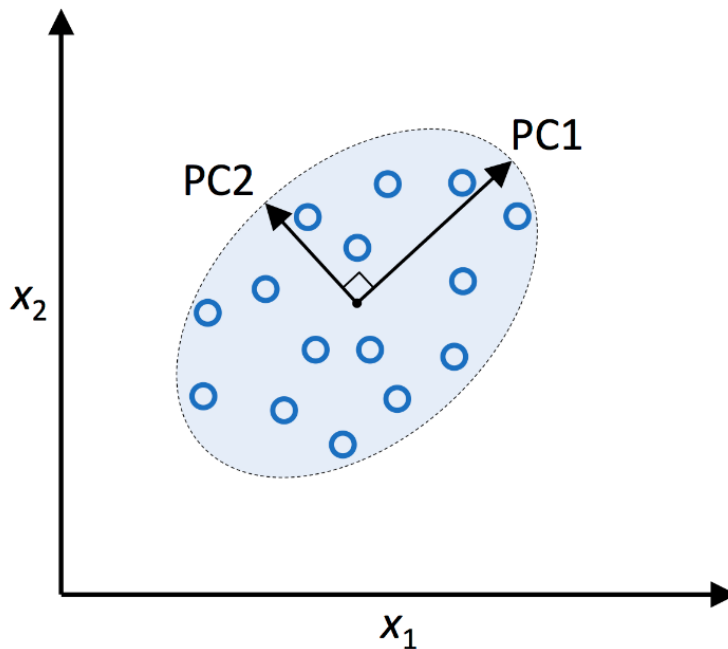



```
# 사이킷런의 PCA모델은 자동으로 데이터를 중앙으로 맞춰준다 (표준화 전처리)
from sklearn.decomposition import PCA

# 차원 축소를 수행하는 대신 분산의 크기 순서대로 모든 주성분이 반환
pca = PCA(n_components=None)
X_train_pca = pca.fit_transform(X_train_std)
# 모든 주성분의 설명된 분산의 비율 (데이터셋의 분산비율 확인)
pca.explained_variance_ratio_

# 2차원으로 축소
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_std)
```

- 주로 특성 추출과 차원 축소
- 탐색적 데이터 분석
- 주식 거래 시장의 잡음 제거
- 생물정보학 분야에서 게놈(genome) 데이터나 유전자 발현 분석(gene expression) 등

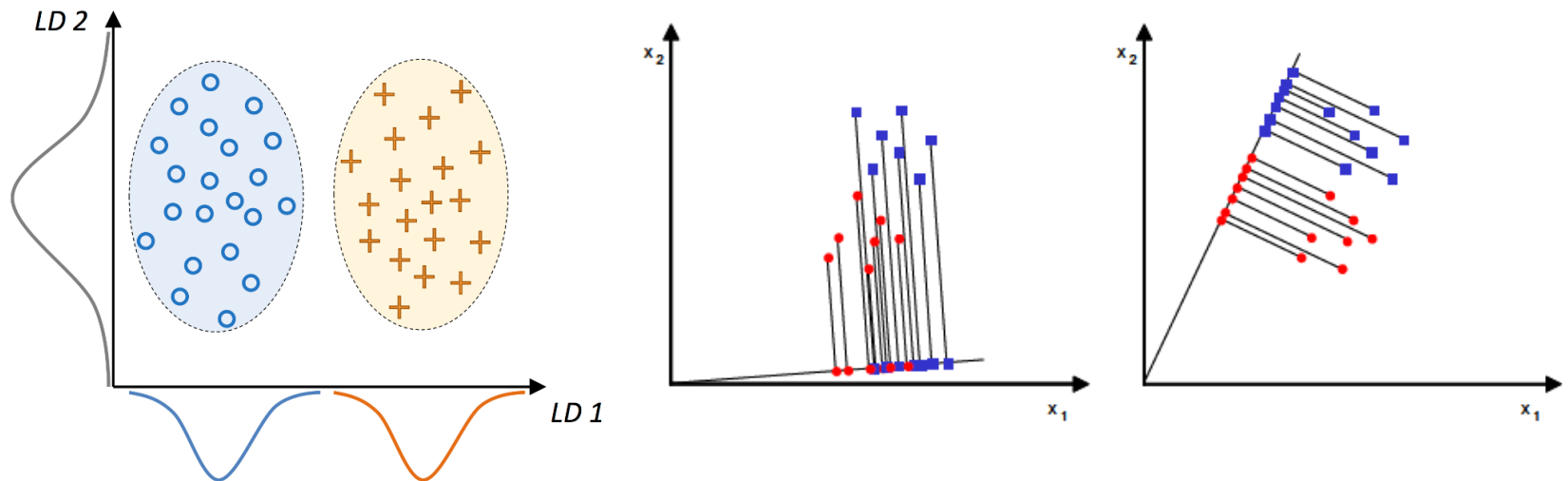




차원의 저주  
주성분 분석(PCA)  
**선형판별 분석(LDA)**  
커널 PCA(KPCA)  
기타 차원축소 기법

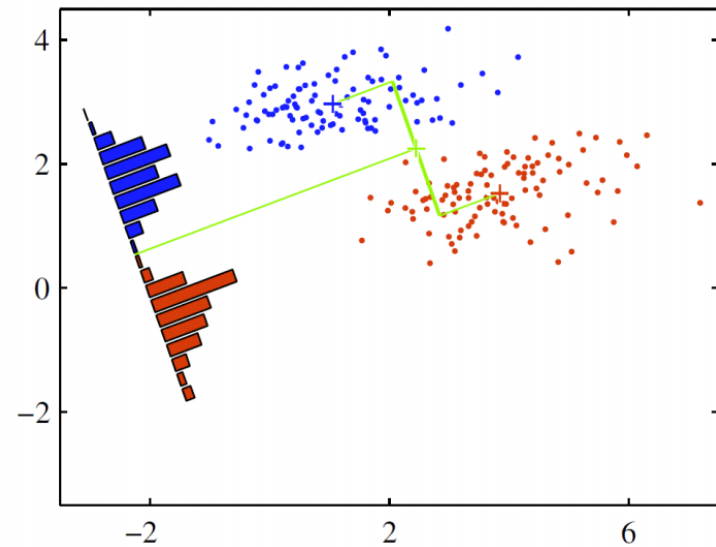
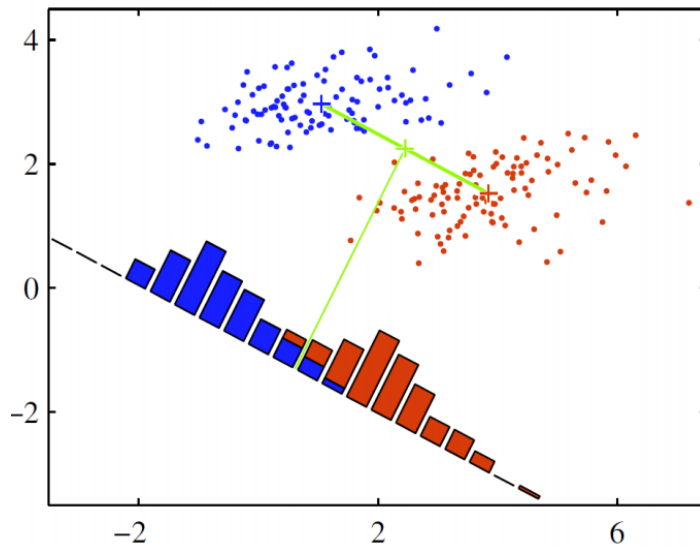
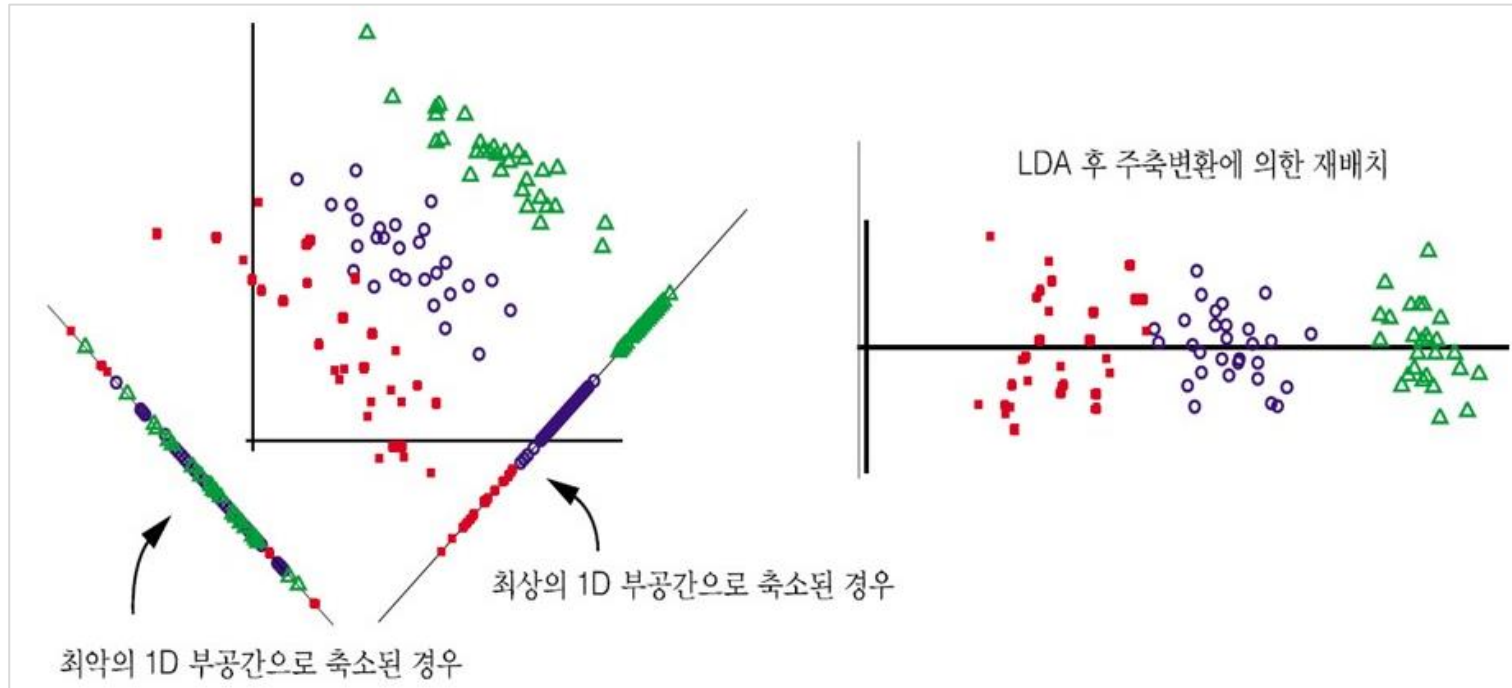


- *PCA(Principal Component Analysis)*
  - 분산이 최대인 직교 성분 축을 찾는 것
  - 비지도 학습 알고리즘
- *LDA(Linear Discrimination Analysis)*
  - 클래스를 최적으로 구분할 수 있는 특성 부분 공간을 찾는 것
    - 투영(projection)후에 클래스를 잘 구분할 수 있는 직선을 찾는 것이 목표
  - 지도학습 알고리즘
  - (예)이진 분류 문제를 위한 LDA
    - x축(LD1): 두 개의 정규 분포 클래스를 잘 구분
    - y축(LD2): 데이터셋에 있는 분산을 많이 잡아내지만 클래스 판별 정보가 없어서 좋지 않다.



# 선형판별 분석이란?

28



- 산포란 결과가 중심점(목표점)을 벗어난 정도. 산포가 클수록 변동이 크다.
  - 산포가 어느 정도 이루어 지고 있는지 통계적으로 나타내는 것이 **표준편차**
1.  $d$ 차원의 데이터셋을 표준화 전처리 한다( $d$ :특성개수)
  2. 각 클래스에 대해  $d$ 차원의 평균 벡터계산
  3. 클래스 간의 산포행렬(*scatter matrix*)  $S_B$ 와 클래스 내 산포행렬  $S_W$ 를 구성
  4.  $S_W^{-1}S_B$  행렬의 고유 벡터와 고윳값 계산
  5. 고윳값을 내림차순으로 정렬하여 고유 벡터의 순서를 매긴다
  6. 고윳값이 가장 큰  $k$ 개의 고유벡터를 선택하여  $d \times k$ 차원의 변환  $W$ 를 구성. 이 행렬의 열이 고유벡터
  7. 변환 행렬  $W$ 를 사용하여 샘플을 새로운 특성 부분 공간으로 투영

# 산포 행렬(scatter matrix) 계산

30

- $m_i$  : 평균벡터. 클래스  $i$  샘플 특성의 평균값  $\mu_m$  을 저장

$$m_i = \frac{1}{n_i} \sum_{x \in D_i}^c x_m$$

$$m_i = \begin{bmatrix} \mu_{i,alcohol} \\ \mu_{i,malic\ acid} \\ \vdots \\ \mu_{i,proline} \end{bmatrix} \quad i \in \{1,2,3\} : \text{세개의 평균벡터}$$

- $S_i$  : 개별 클래스  $i$ 의 산포 행렬

$$S_i = \sum_{x \in D_i}^c (x - m_i)(x - m_i)^T$$

- $S_W$  : 클래스 내 산포 행렬

$$S_W = \sum_{i=1}^c S_i$$

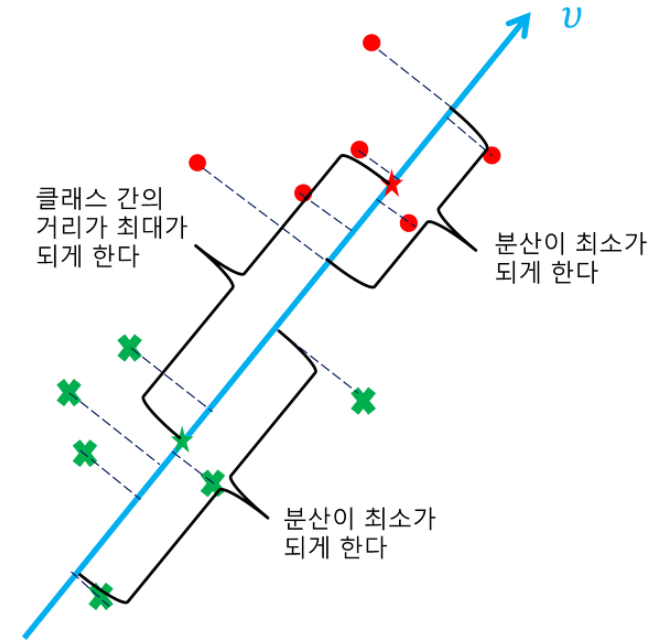
- 공분산 행렬 : 클래스 샘플 개수  $n$ 로 나눈다(스케일 조정, 산포행렬의 정규화)

$$\Sigma_i = \frac{1}{n_i} S_i = \frac{1}{n_i} \sum_{x \in D_i}^c (x - m_i)(x - m_i)^T$$

- $S_B$  : 클래스 간의 산포 행렬

$$S_B = \sum_{i=1}^c n_i (m_i - m)(m_i - m)^T$$

( $m$ : 전체 클래스 샘플의 평균)



```
# 세개의 평균벡터 생성
import numpy as np

# 부동 소수점, 배열, 그리고 다른 NumPy 객체가 표시되는 방식을 설정
np.set_printoptions(precision=4)

mean_vecs = []
for label in range(1, 4):
    mean_vecs.append(np.mean(X_train_std[y_train == label], axis=0))
    print('Mean Vector %s: %s\n' % (label, mean_vecs[label - 1]))
```

**Mean Vector 1:** [ 0.9066 -0.3497 0.3201 -0.7189 0.5056 0.8807 0.9589 -0.5516 0.5416  
0.2338 0.5897 0.6563 1.2075]

**Mean Vector 2:** [-0.8749 -0.2848 -0.3735 0.3157 -0.3848 -0.0433 0.0635 -0.0946 0.0703  
-0.8286 0.3144 0.3608 -0.7253]

**Mean Vector 3:** [ 0.1992 0.866 0.1682 0.4148 -0.0451 -1.0286 -1.2876 0.8287 -0.7795  
0.9649 -1.209 -1.3622 -0.4013]

## 평균 벡터를 사용하여 클래스 내 산포행렬 $S_W$ 를 계산

32

```
# 클래스 내의 산포 행렬
'''

$$S_i = \sum_{x \in D_i}^c (x - m_i)(x - m_i)^T$$


$$S_W = \sum_{i=1}^c S_i$$

'''
d = 13 # number of features
S_W = np.zeros((d, d))
for label, mv in zip(range(1, 4), mean_vecs):
    class_scatter = np.zeros((d, d)) # scatter matrix for each class
    for row in X_train_std[y_train == label]:
        row, mv = row.reshape(d, 1), mv.reshape(d, 1) # make column vectors
        class_scatter += (row - mv).dot((row - mv).T)
    S_W += class_scatter # sum class scatter matrices

print('Within-class scatter matrix: %sx%s' % (S_W.shape[0], S_W.shape[1]))
# 클래스 내 레이블 분포
print('Class label distribution: %s' % np.bincount(y_train)[1:])
```

Within-class scatter matrix: 13x13

Class label distribution: [41 50 33]



```
'''  

$$\Sigma_i = \frac{1}{n_i} S_i = \frac{1}{n_i} \sum_{x \in D_i}^c (x - m_i)(x - m_i)^T$$
  
'''  
d = 13 # number of features  
S_W = np.zeros((d, d))  
for label, mv in zip(range(1, 4), mean_vecs):  
    class_scatter = np.cov(X_train_std[y_train == label].T)  
    S_W += class_scatter  
# 스케일 조정된 클래스 내의 산포행렬  
print('Scaled within-class scatter matrix: %sx%s' % (S_W.shape[0],  
                                                       S_W.shape[1]))
```

Scaled within-class scatter matrix: 13x13

```
'''

$$S_B = \sum_{i=1}^c n_i (m_i - m)(m_i - m)^T$$

'''
mean_overall = np.mean(X_train_std, axis=0)
d = 13 # number of features
S_B = np.zeros((d, d))
for i, mean_vec in enumerate(mean_vecs):
    n = X_train[y_train == i + 1, :].shape[0]
    mean_vec = mean_vec.reshape(d, 1) # make column vector
    mean_overall = mean_overall.reshape(d, 1) # make column vector
    S_B += n * (mean_vec - mean_overall).dot((mean_vec - mean_overall).T)

# 클래스간의 산포행렬
print('Between-class scatter matrix: %sx%s' % (S_B.shape[0], S_B.shape[1]))
```

Between-class scatter matrix: 13x13

```
# PCA의 공분산 행렬에 대한 고윳값 분해를 수행하는 대신 행렬  $S_W^{-1}S_B$ 의 고윳값 계산
eigen_vals, eigen_vecs = np.linalg.eig(np.linalg.inv(S_W).dot(S_B))

# Make a list of (eigenvalue, eigenvector) tuples
eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i])
                for i in range(len(eigen_vals))]

# Sort the (eigenvalue, eigenvector) tuples from high to low
eigen_pairs = sorted(eigen_pairs, key=lambda k: k[0], reverse=True)

# Visually confirm that the list is correctly sorted by decreasing eigenvalues
# 내림차순의 고유값
print('Eigenvalues in descending order:\n')
for eigen_val in eigen_pairs:
    print(eigen_val[0])
```

Eigenvalues in descending order:

```
349.61780890599397
172.7615221897938
3.389259780547781e-14
2.842170943040401e-14
1.9284611807586422e-14
1.9284611807586422e-14
1.8639179987230033e-14
1.8639179987230033e-14
7.057897559458914e-15
7.057897559458914e-15
6.596592553773414e-15
3.81059209269662e-15
3.3908455462202616e-15
```

- LDA에서 선형 판별 벡터는 최대  $c - 1$ 개( $c$  : 클래스 레이블 개수)
- 클래스 내 산포행렬  $S_B$ 가 랭크 1 또는 그 이하인  $c$ 개의 행렬을 합한 것
- 앞 예제에서 0이 아닌 고윳값이 두 개만 있는 것을 볼 수 있다.
- 모든 샘플이 동일 선상에 위치한 경우 공분산 행렬의 랭크는 1이다.
  - 0이 아닌 고윳값을 가진 고유 벡터가 하나만 만들어진다

```

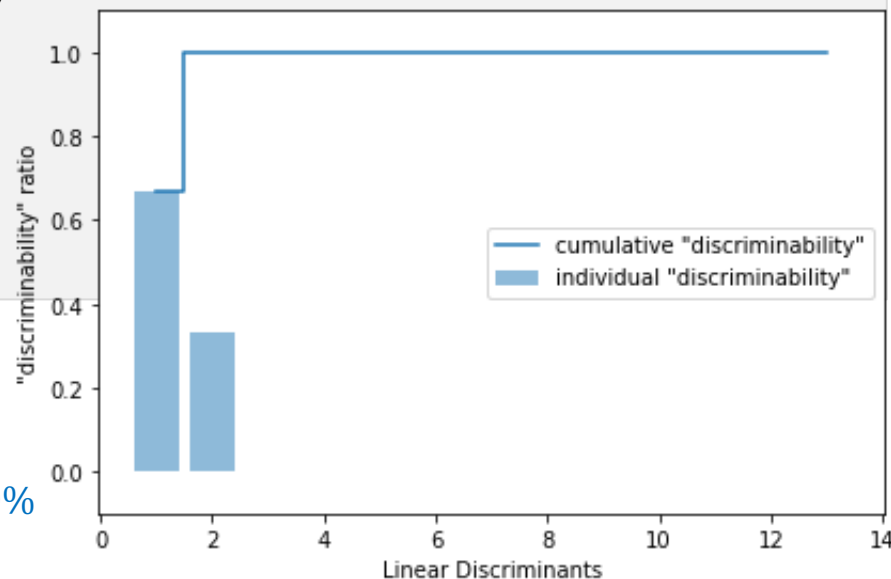
tot = sum(eigen_vals.real)
discr = [(i / tot) for i in sorted(eigen_vals.real, reverse=True)]
cum_discr = np.cumsum(discr)

import matplotlib.pyplot as plt

plt.bar(range(1, 14), discr, alpha=0.5, align='center',
        label='individual "discriminability"')
plt.step(range(1, 14), cum_discr, where='mid',
        label='cumulative "discriminability"')
plt.ylabel('"discriminability" ratio')
plt.xlabel('Linear Discriminants')
plt.ylim([-0.1, 1.1])
plt.legend(loc='best')
plt.tight_layout()
plt.show()

```

처음 두 개의 정보 : 100%



```
w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,  
               eigen_pairs[1][1][:, np.newaxis].real))  
print('Matrix W:\n', w)
```

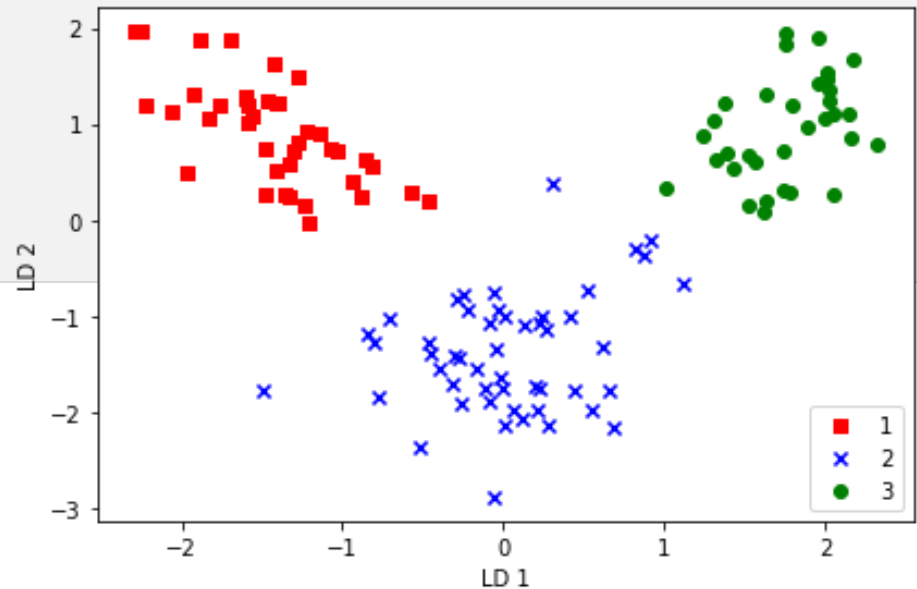
Matrix W:

```
[[-0.1481 -0.4092]  
 [ 0.0908 -0.1577]  
 [-0.0168 -0.3537]  
 [ 0.1484  0.3223]  
 [-0.0163 -0.0817]  
 [ 0.1913  0.0842]  
 [-0.7338  0.2823]  
 [-0.075  -0.0102]  
 [ 0.0018  0.0907]  
 [ 0.294  -0.2152]  
 [-0.0328  0.2747]  
 [-0.3547 -0.0124]  
 [-0.3915 -0.5958]]
```

```
#  $X' = XW$ 
X_train_lda = X_train_std.dot(w)
colors = ['r', 'b', 'g']
markers = ['s', 'x', 'o']

for l, c, m in zip(np.unique(y_train), colors, markers):
    plt.scatter(X_train_lda[y_train == l, 0],
                X_train_lda[y_train == l, 1] * (-1),
                c=c, label=l, marker=m)


plt.xlabel('LD 1')
plt.ylabel('LD 2')
plt.legend(loc='lower right')
plt.tight_layout()
plt.show()
```



```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

lda = LDA(n_components=2)
X_train_lda = lda.fit_transform(X_train_std, y_train)
```





차원의 저주  
주성분 분석(PCA)  
선형판별 분석(LDA)  
**커널 PCA(KPCA)**  
기타 차원축소 기법

- 커널 PCA
  - 비선형 매핑
  - 꼬인 매니폴드에 가까운 데이터셋을 펼칠 때도 유용
- 어떤 특징공간에 정의된 두 특징벡터  $x^{(i)}, x^{(j)}$ 에 대해  $K(x^{(i)}, x^{(j)}) = \phi(x^{(i)}) \cdot \phi(x^{(j)})$ 인 변환함수  $\phi$ 가 존재하면  $K(x^{(i)}, x^{(j)})$ 를 커널함수라 부른다
- 선형 커널
$$K(x^{(i)}, x^{(j)}) = (x^{(i)})^T \cdot x^{(j)}$$
- 다항식 커널(polynomial kernel)
$$K(x^{(i)}, x^{(j)}) = (\gamma(x^{(i)})^T \cdot x^{(j)} + r)^d, \quad d: \text{integer}(+)$$
- 방사기저 함수(radial basis function, RBF), 가우시안 커널(Gaussian kernel)
$$K(x^{(i)}, x^{(j)}) = \exp\left(-\frac{\|x^{(i)} - x^{(j)}\|^2}{2\sigma^2}\right)$$
$$K(x^{(i)}, x^{(j)}) = \exp\left(-\gamma\|x^{(i)} - x^{(j)}\|^2\right), \quad \gamma = \frac{1}{2\sigma^2}$$
- 쌍곡 탄젠트 커널(hyperbolic tangent), 시그모이드(sigmoid)
$$K(x^{(i)}, x^{(j)}) = \tanh(\gamma(x^{(i)})^T \cdot x^{(j)} + r)$$

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import make_swiss_roll
# 스위스 롤 데이터셋
X, t = make_swiss_roll(n_samples=1000, noise=0.2, random_state=42)

# KPCA
from sklearn.decomposition import KernelPCA

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

lin_pca = KernelPCA(n_components = 2, kernel="linear", fit_inverse_transform=True)

rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.0433, fit_inverse_transform=True)

sig_pca = KernelPCA(n_components = 2, kernel="sigmoid", gamma=0.001, coef0=1, fit_inverse_transform=True)

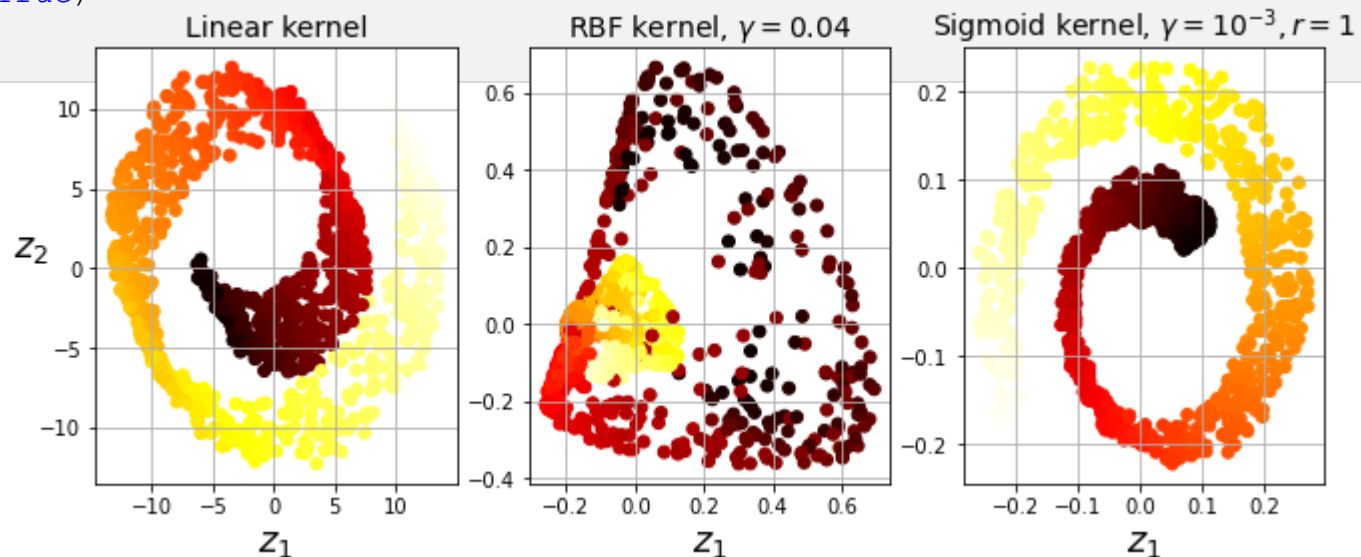
y = t > 6.9
```

```

plt.figure(figsize=(11, 4))
for subplot, pca, title in ((131, lin_pca, "Linear kernel"), (132, rbf_pca, "RBF kernel,
    $\gamma=0.04$"), (133, sig_pca, "Sigmoid kernel, $\gamma=10^{-3}$, $r=1$")):
    X_reduced = pca.fit_transform(X)
    if subplot == 132:
        X_reduced_rbf = X_reduced

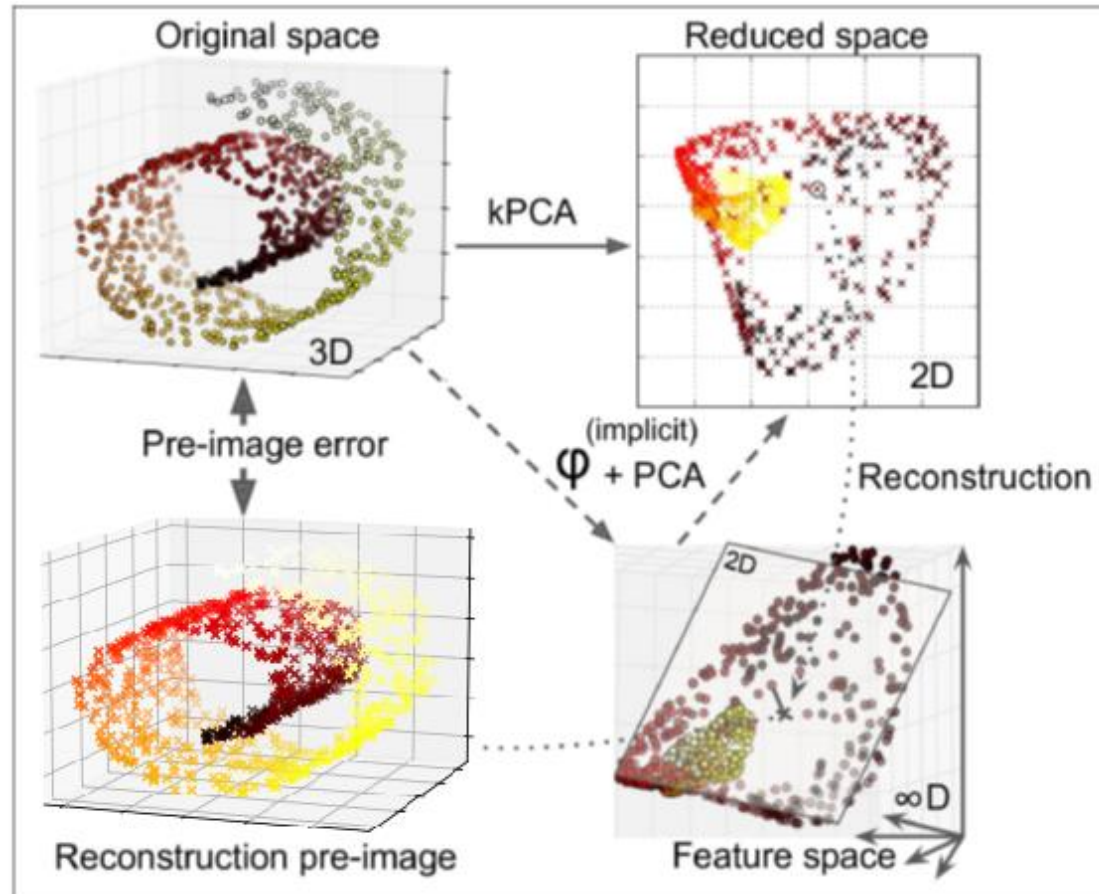
    plt.subplot(subplot)
    #plt.plot(X_reduced[y, 0], X_reduced[y, 1], "gs")
    #plt.plot(X_reduced[~y, 0], X_reduced[~y, 1], "y^")
    plt.title(title, fontsize=14)
    plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=plt.cm.hot)
    plt.xlabel("$z_1$", fontsize=18)
    if subplot == 131:
        plt.ylabel("$z_2$", fontsize=18, rotation=0)
    plt.grid(True)
plt.show()

```



재구성 원상의 오차를 최소화하는 커널과 하이퍼파라미터를 선택

**RBF커널**의 KPCA를 적용한 2D 데이터셋



재구성원상 : 재구성된 포인트에 가깝게 매핑된 원본 공간의 포인트를 찾는 것

특성맵을  $\phi$ 을 사용하여 훈련 세트를 무한 차원의 특성 공간에 매핑한 다음 변환된 데이터 셋을 선형 PCA를 사용해 2D로 투영한 것과 수학적으로 동일

```
from sklearn.decomposition import PCA

scikit_pca = PCA(n_components=2)
X_spca = scikit_pca.fit_transform(X)

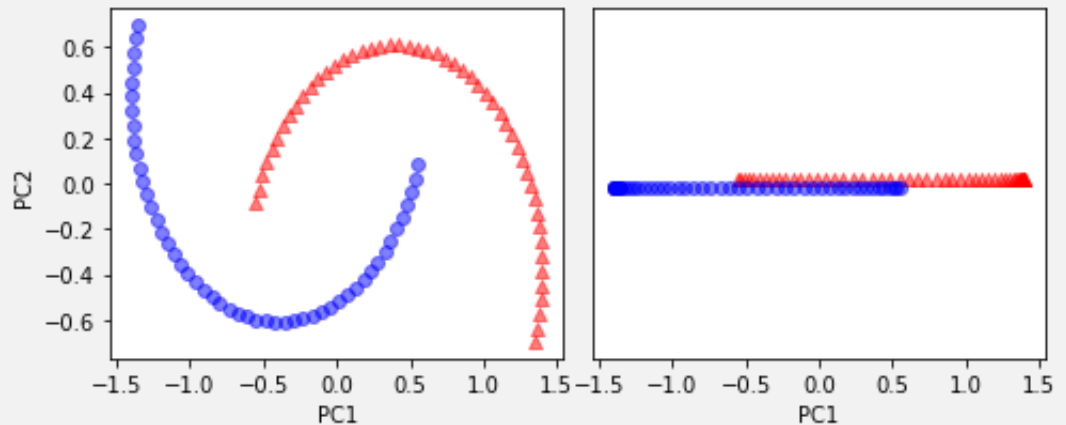
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7, 3))

ax[0].scatter(X_spca[y == 0, 0], X_spca[y == 0, 1],
              color='red', marker='^', alpha=0.5)
ax[0].scatter(X_spca[y == 1, 0], X_spca[y == 1, 1],
              color='blue', marker='o', alpha=0.5)

ax[1].scatter(X_spca[y == 0, 0], np.zeros((50, 1)) + 0.02,
              color='red', marker='^', alpha=0.5)
ax[1].scatter(X_spca[y == 1, 0], np.zeros((50, 1)) - 0.02,
              color='blue', marker='o', alpha=0.5)

ax[0].set_xlabel('PC1')
ax[0].set_ylabel('PC2')
ax[1].set_ylim([-1, 1])
ax[1].set_yticks([])
ax[1].set_xlabel('PC1')

plt.tight_layout()
plt.show()
```



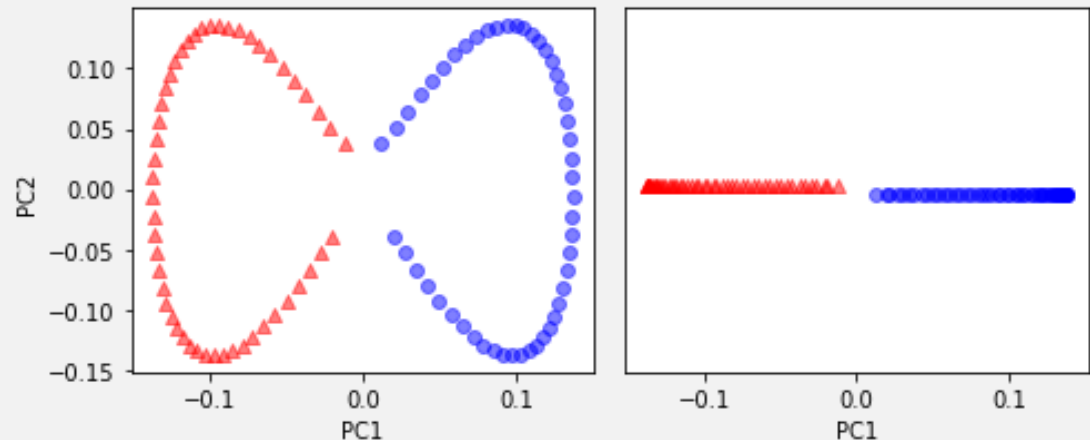
```
X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)

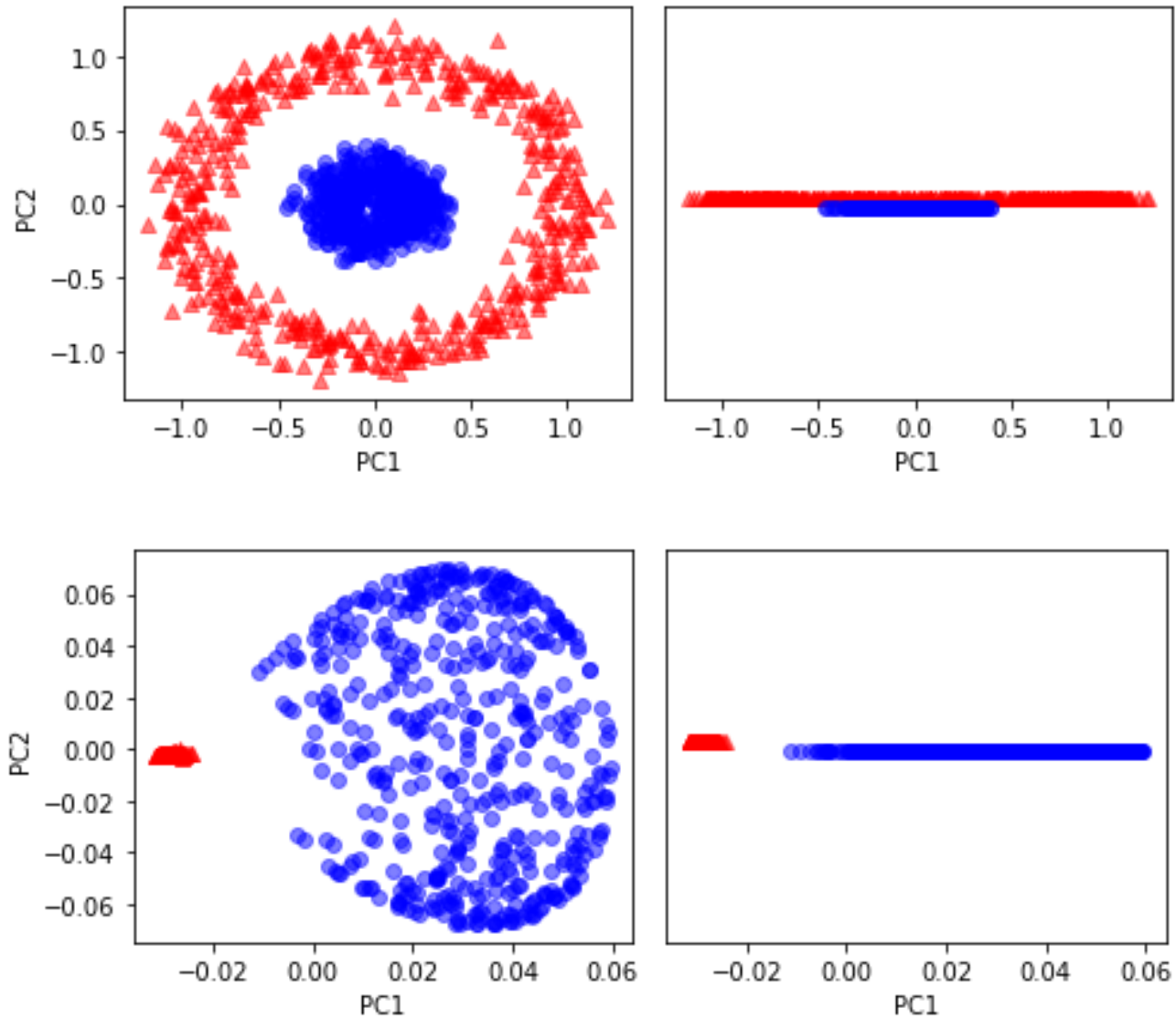
fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
              color='red', marker='^', alpha=0.5)
ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
              color='blue', marker='o', alpha=0.5)

ax[1].scatter(X_kpca[y==0, 0], np.zeros((50,1))+0.02,
              color='red', marker='^', alpha=0.5)
ax[1].scatter(X_kpca[y==1, 0], np.zeros((50,1))-0.02,
              color='blue', marker='o', alpha=0.5)

ax[0].set_xlabel('PC1')
ax[0].set_ylabel('PC2')
ax[1].set_ylim([-1, 1])
ax[1].set_yticks([])
ax[1].set_xlabel('PC1')

plt.tight_layout()
plt.show()
```





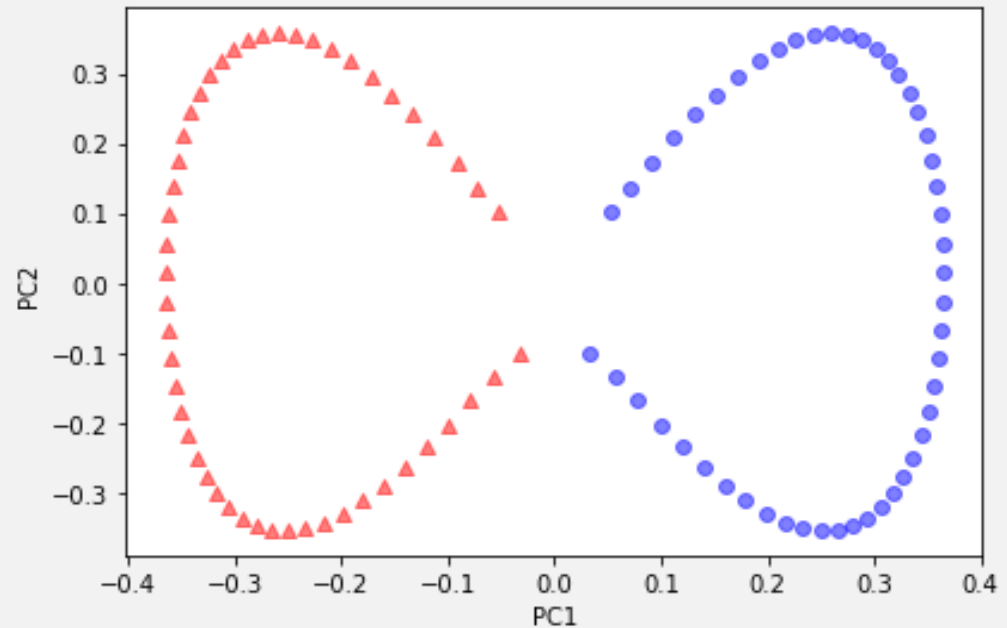



```
from sklearn.decomposition import KernelPCA

X, y = make_moons(n_samples=100, random_state=123)
scikit_kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)
X_skernpca = scikit_kpca.fit_transform(X)

plt.scatter(X_skernpca[y == 0, 0], X_skernpca[y == 0, 1],
            color='red', marker='^', alpha=0.5)
plt.scatter(X_skernpca[y == 1, 0], X_skernpca[y == 1, 1],
            color='blue', marker='o', alpha=0.5)

plt.xlabel('PC1')
plt.ylabel('PC2')
plt.tight_layout()
plt.show()
```



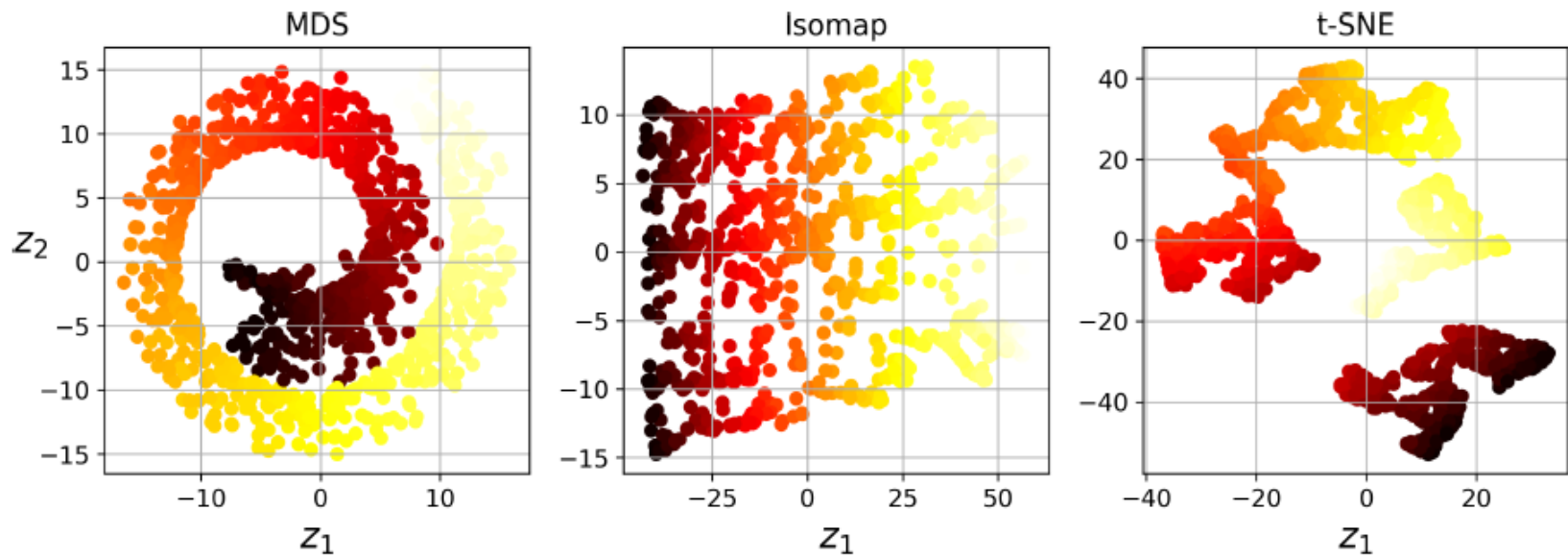


차원의 저주  
주성분 분석(PCA)  
선형판별 분석(LDA)  
커널 PCA(KPCA)  
기타 차원축소 기법

- 랜덤 투영(random projection)
  - 랜덤한 선형 투영을 사용해 데이터를 저차원 공간으로 투영
  - `sklearn.random_projection` 패키지 참고
- 다차원 스케일링(multidimensional scaling, MDS)
  - 샘플간의 거리를 보존하면서 차원 축소
- Isomap
  - 각 샘플을 가장 가까운 이웃과 연결하는 식으로 그래프 생성
  - 샘플 간의 지오데식 거리(geodesic distance)를 유지하면서 차원 축소
- t-SNE(t-distributed stochastic neighbor embedding)
  - 비슷한 샘플은 가까이, 비슷하지 않은 샘플은 멀리 떨어지도록 하면서 차원 축소
  - 주로 시각화에 많이 사용 - 특히 고차원 공간에 있는 샘플의 군집을 시각화할 때 사용

# 스위스 롤을 2D로 축소하기

52

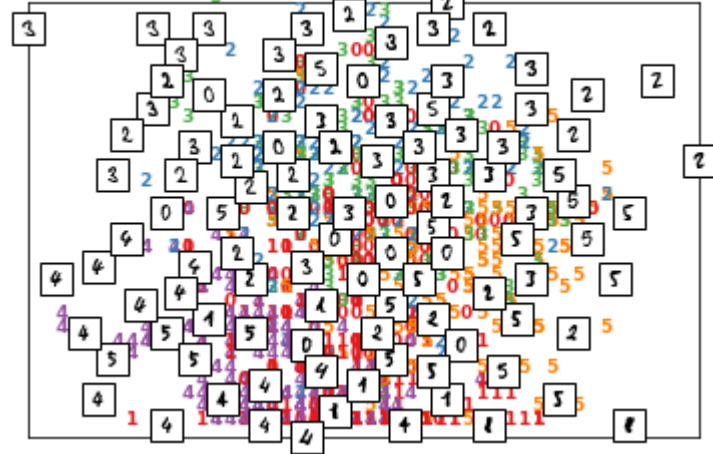


# Manifold learning on handwritten digits

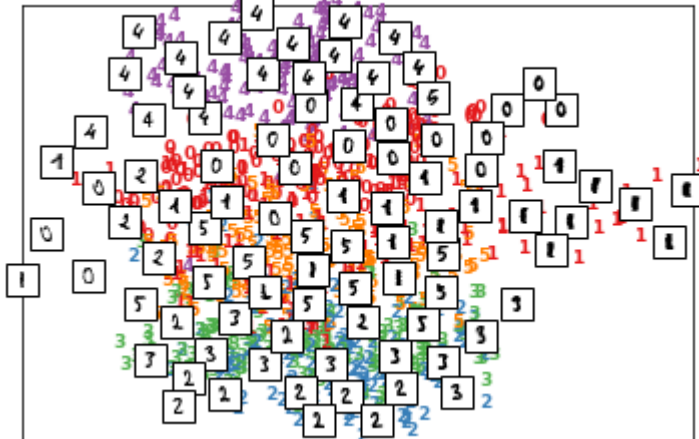
A selection from the 64-dimensional digits dataset



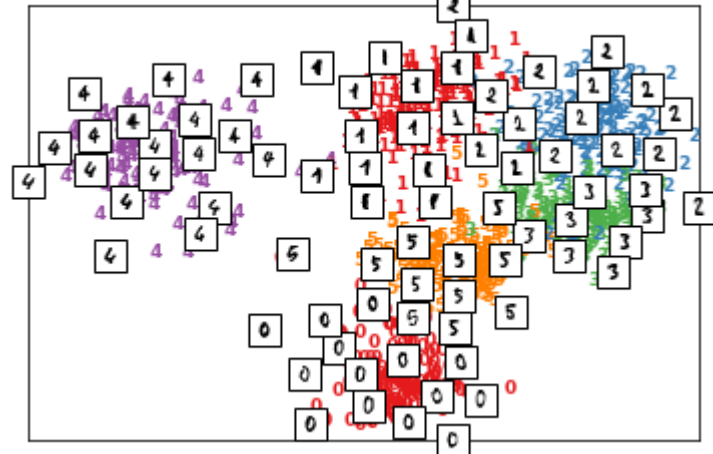
Random Projection of the digits



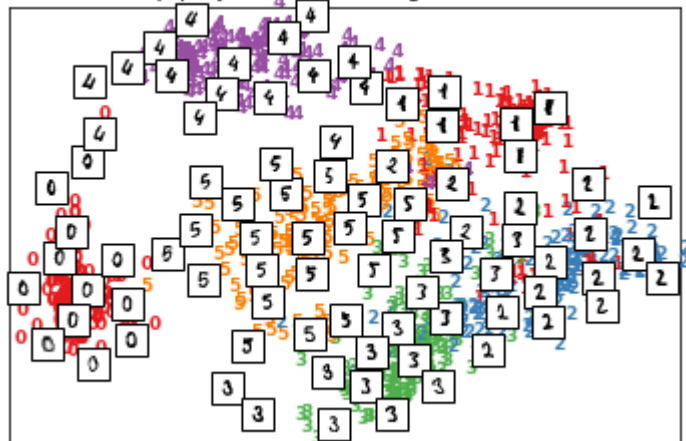
Principal Components projection of the digits (time 0.03s)



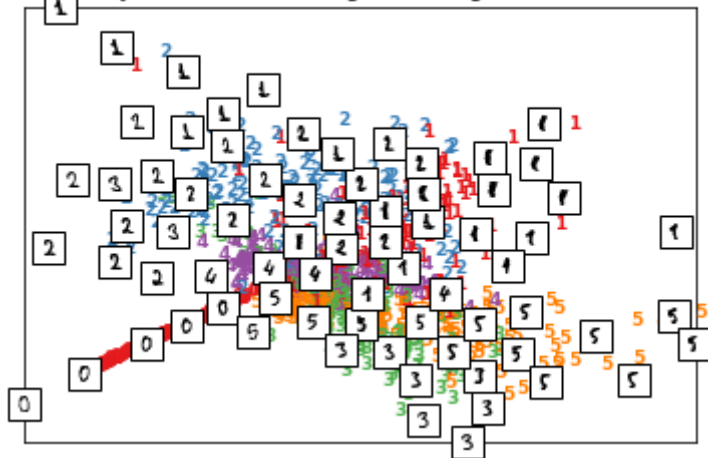
Linear Discriminant projection of the digits (time 0.02s)



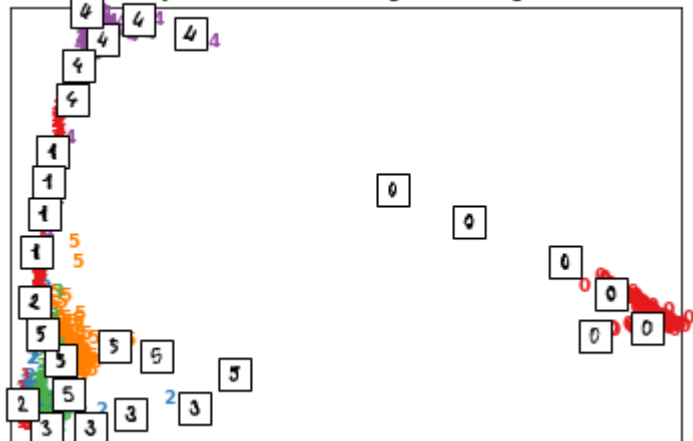
Isomap projection of the digits (time 1.27s)



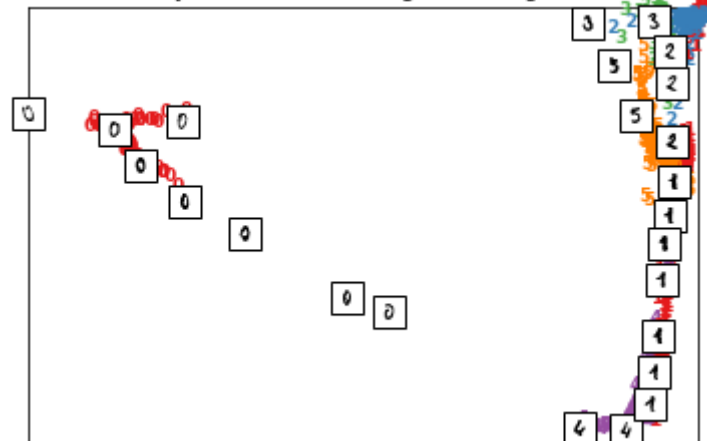
Locally Linear Embedding of the digits (time 0.43s)



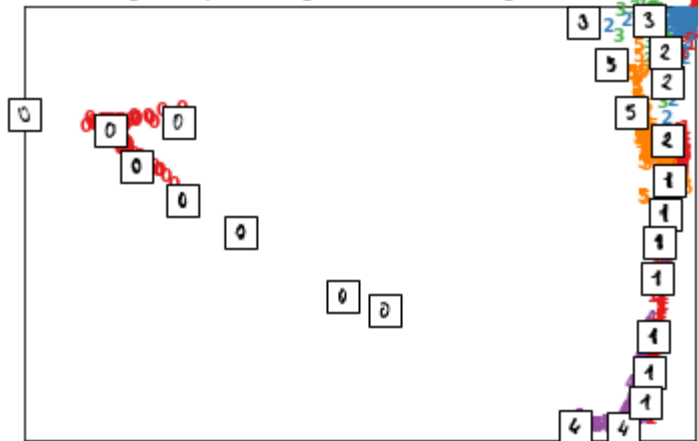
Modified Locally Linear Embedding of the digits (time 0.98s)



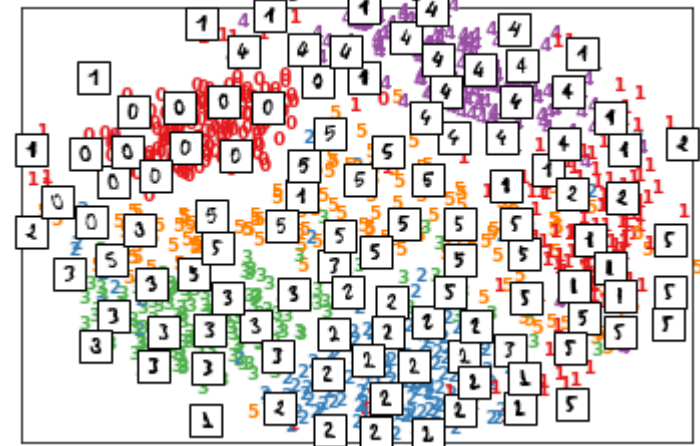
Hessian Locally Linear Embedding of the digits (time 1.23s)



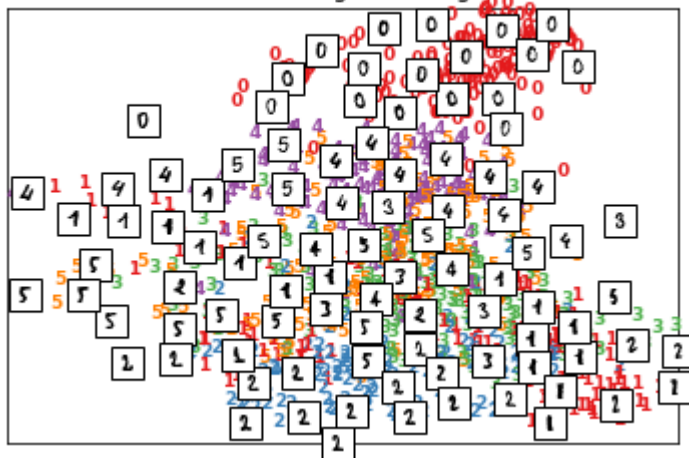
Local Tangent Space Alignment of the digits (time 0.93s)



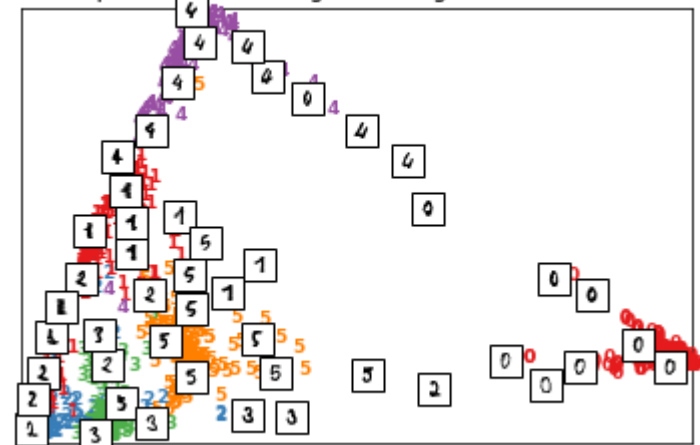
MDS embedding of the digits (time 2.98s)



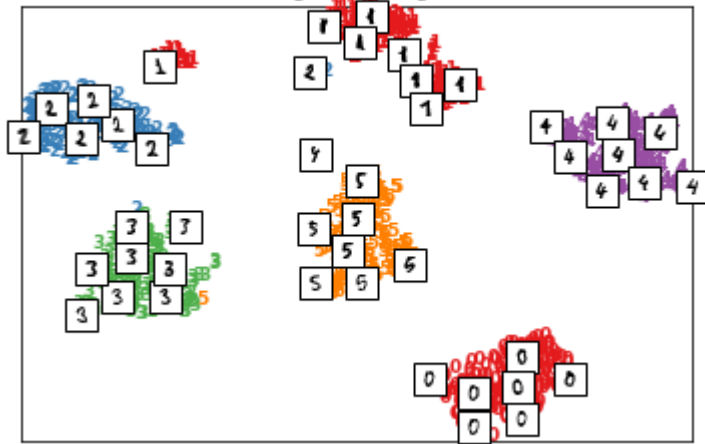
Random forest embedding of the digits (time 0.30s)



Spectral embedding of the digits (time 0.46s)



t-SNE embedding of the digits (time 7.31s)



NCA embedding of the digits (time 4.91s)

