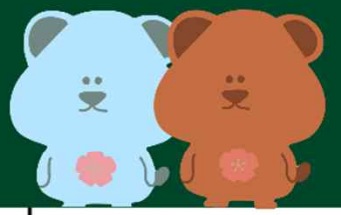
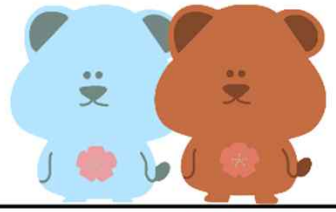


7주차 발표

여채윤, 이가영 조혜빈

EWHA EURON

목차



EWHA EURON

1. 차원 축소

EWHA EURON

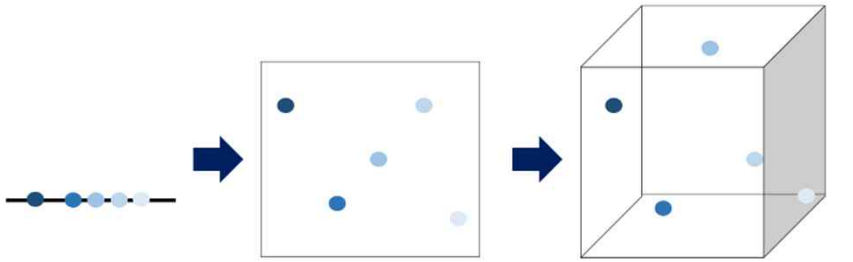
1.1 차원 축소

- 차원 축소란?

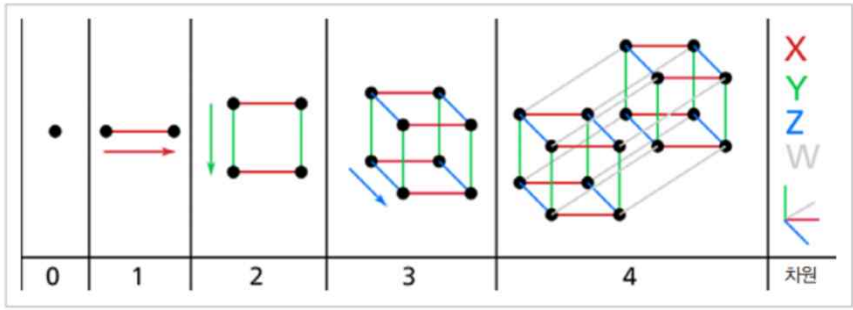
매우 많은 피처로 구성된 다차원 데이터 세트의 차원을 축소해

새로운 차원의 데이터 세트를 생성하는 것

- 차원 축소를 해야 하는 이유는?



차원이 증가할 수록 점들 사이의 공간이 늘어남
-> 희소(sparse)한 구조를 가지게 됨
-> 예측 성능 저하



차원이 증가할 수록 어떤 점을 선택하든 경계선과 가까워짐
-> 개별 피처간의 상관 관계가 높아질 가능성이 커짐
-> 다중 공선성 문제로 예측 성능 저하

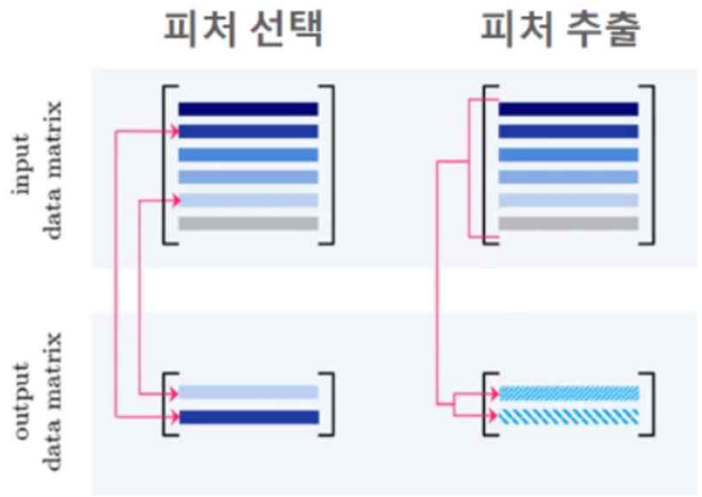
차원의 저주가 발생함!

EWHA EURON

1.1 차원 축소

- 피처 선택: 특정 피처에 종속성이 강한 불필요한 피처는 아예 제거하고,
  데이터의 특징을 잘 나타내는 주요 피처만 선택하는 방법
- 피처 추출: 기존 피처를 단순 압축이 아닌 피처를 함축적으로
  더 잘 설명할 수 있는 또 다른 공간으로 매핑해 추출하는 방법
  기존 피처들이 나타내지 못한 잠재적인 요소를 추출함

차원 축소의 중요한 의미는 단순히 데이터의 압축이 아니라
데이터를 잘 설명하는 잠재적 요소를 추출하는 데에 있다

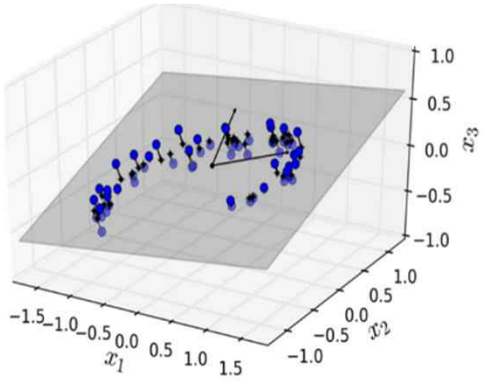


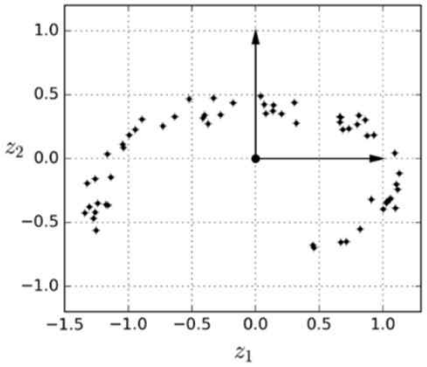
EWHA EURON

1.2 차원 축소 방법

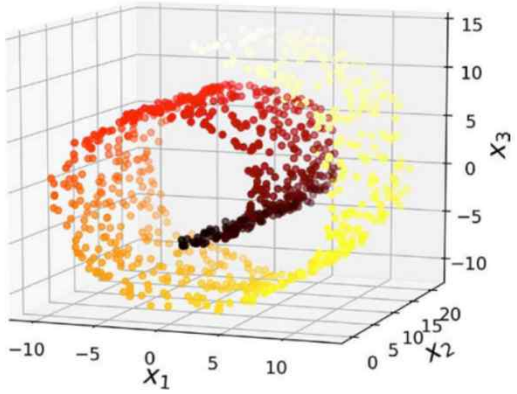
1. 투영

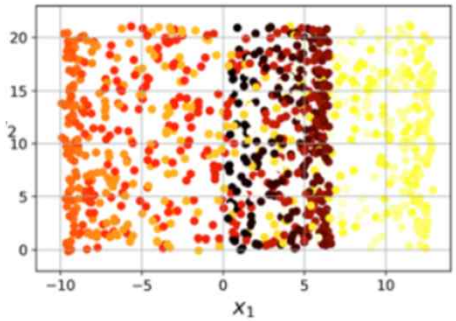
고차원 공간에 있는 훈련 샘플을 저차원 공간에 그대로 수직으로 옮기는 것





모든 data들이 거의 평면 형태로 놓여있기 때문에
3차원 데이터를 2차원에 투영해도 데이터 특성이
크게 뭉개지지 않고 거의 그대로 보존됨





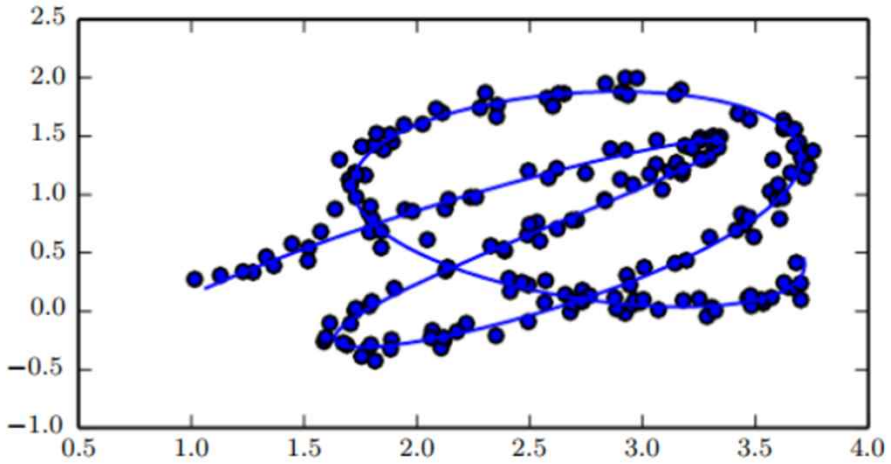
스위스 롤처럼 데이터가 말려있는 경우
2차원으로 투영시켰을 때 데이터가 뭉개지기 때문에
좋은 방법이 아님

1.2 차원 축소 방법

2. 매니폴드

고차원 데이터가 있을 때, 고차원 데이터를 데이터 공간에 뿌리면

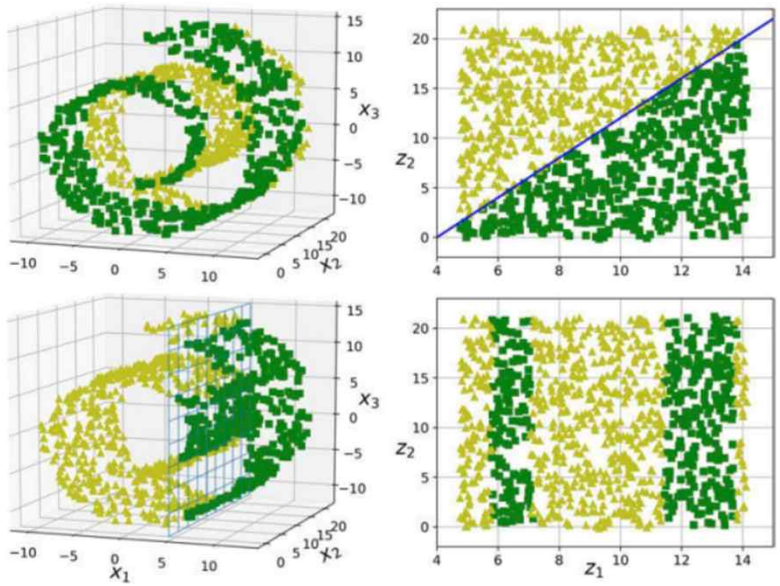
샘플들을 잘 아우르는 subspace가 있을 것이라고 가정하고 학습하는 방법



풀어 헤친 형태를 모델링하는 식으로 작동

고차원 데이터를 저차원에서도 잘 표현하는 공간인

manifold를 찾아 차원을 축소시킴



저차원의 매니폴드 공간에 표현된다고 해서

항상 들어맞는 것은 아님

2. PCA

EWHA EURON

2.1 PCA (Principal Component Analysis) 개요

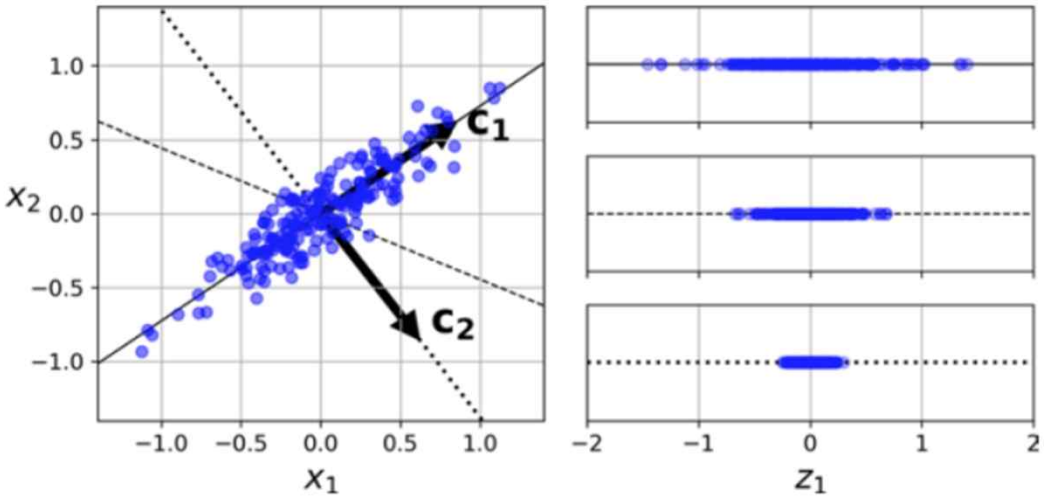
- PCA란?

가장 대표적인 차원 축소 기법 중 하나로

여러 변수 간에 존재하는 상관 관계를 이용해 이를 대표하는 주성분을 추출해서 차원을 축소하는 기법

정보의 유실을 최소화하기 위해 가장 중요한 것은 데이터 분포를 유지하는 것

즉, 분산을 보존하는 것이다



-> 분산이 최대로 보존하는 것은 첫번째 실선임

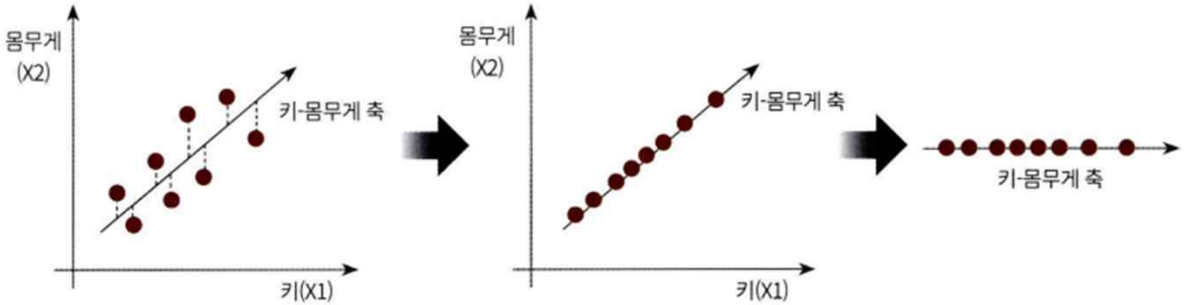
-> 투영되기 전 데이터와 투영된 데이터 간 평균 제곱 거리를 최소화하는 축을 선택하는 것이 PCA 기법

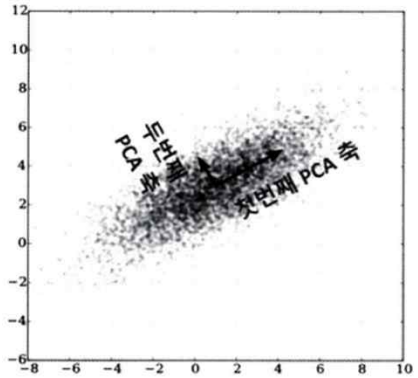
EWHA EURON

2.2 주성분

https://box-world.tistory.com/33

- PCA 작업이 이뤄지는 방법 (n차원 -> d차원)





1. 훈련 세트에서 분산이 최대인 축 찾기
2. 1번 축에 직교하면서 남은 분산을 최대한 보존하는 두번째 축 찾기
3. 2번 축에 직교하면서 남은 분산을 최대한 보존하는 세번째 축 찾기
4. 위의 단계를 반복하면서 데이터셋에 있는 차원의 수만큼 n번째 축 찾기
5. 주성분을 모두 추출한 후 처음 d개의 주성분으로 정의한 초평면에 투영해 데이터셋을 d차원으로 축소

이 때 i번째 축을 이 데이터의 i번째 주성분(PC)라고 부른다.

2.3 선형대수 관점에서 PCA

PCA는 선형대수 관점에서

입력 데이터의 **공분산 행렬**을 **고유값 분해**하고, **고유벡터**에 입력 데이터를 **선형 변환**하는 것

이라고 볼 수 있다.

	선형대수에서 개념	PCA에서 의미하는 바
선형 변환	Av=b와 같이 선형 결합을 보존하는, 두 벡터 공간 사이의 함수	N차원을 d차원으로 투영시키는 것
고유 벡터	Av = λv를 만족하는 0이 아닌 벡터 v 변환을 해도 방향이 바뀌지 않음	주성분 벡터 입력 데이터의 분산이 큰 방향
고유 값	Av = λv를 만족하는 0이 아닌 상수 λ	주성분 벡터의 길이 입력 데이터의 분산
고유값 분해	고유값과 고유벡터를 찾는 작업	

2.3 선형대수 관점에서 PCA - 공분산 행렬

공분산 행렬

: 데이터의 좌표 성분들 사이의 공분산 값을 원소로 하는 행렬로써

데이터의 i번째 좌표 성분과 j번째 좌표 성분의 공분산 값을 행렬의 i행 j열 원소값으로 함

$$C = \begin{bmatrix} cov(x,x) & cov(x,y) \\ cov(y,x) & cov(y,y) \end{bmatrix} = \frac{1}{N} \begin{bmatrix} \Sigma(x_i - m_x)^2 & \Sigma(x_i - m_x)(y_i - m_y) \\ \Sigma(x_i - m_x)(y_i - m_y) & \Sigma(y_i - m_y)^2 \end{bmatrix}$$

C 는 nxn 크기의 정방행렬이며, $C = C^T$ 인 대칭 행렬임 (cov(i,j) = cov(j,i))

2.3 선형대수 관점에서 PCA - 공분산 행렬

대칭행렬은 고유벡터를 직교행렬로, 고유값을 정방행렬로 대각화 할 수 있다.

C = N차원의 대칭행렬

V = $[e_1 \cdots e_n]$ 고유 벡터 행렬

$\Lambda = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_N \end{bmatrix}$ 고유값 행렬

$$C = V \Lambda V^T \qquad C = [e_1 \cdots e_n] \begin{bmatrix} \lambda_1 & \cdots & 0 \\ \cdots & \cdots & \cdots \\ 0 & \cdots & \lambda_n \end{bmatrix} \begin{bmatrix} e_1^t \\ \cdots \\ e_n^t \end{bmatrix}$$

즉, 공분산 행렬 C는 대칭행렬이기 때문에

고유벡터 행렬 * 고유값 정방 행렬 * 고유벡터 행렬의 전치행렬로 분해됨

e_1은 가장 분산이 큰 방향을 가진 고유 벡터 (주성분 벡터)고 e_2는 e_1에 수직이면서 두번째로 큰 방향을 가진 고유벡터

2.3 선형대수 관점에서 PCA - 공분산 행렬

https://ratsgo.github.io/machine%20learning/2017/04/24/PCA/

1. 데이터 셋 선형 변환

$$\vec{z_1} = \alpha_{11}\vec{x_1} + \alpha_{12}\vec{x_2} + \ldots + \alpha_{1p}\vec{x_p} = \vec{\alpha_1}^T X$$
$$\vec{z_2} = \alpha_{21}\vec{x_1} + \alpha_{22}\vec{x_2} + \ldots + \alpha_{2p}\vec{x_p} = \vec{\alpha_2}^T X$$
$$\ldots$$
$$\vec{z_p} = \alpha_{p1}\vec{x_1} + \alpha_{p2}\vec{x_2} + \ldots + \alpha_{pp}\vec{x_p} = \vec{\alpha_p}^T X$$

$$Z = \begin{bmatrix} \vec{z_1} \\ \vec{z_2} \\ \ldots \\ \vec{z_p} \end{bmatrix} = \begin{bmatrix} \vec{\alpha_1}^T X \\ \vec{\alpha_2}^T X \\ \ldots \\ \vec{\alpha_p}^T X \end{bmatrix} = \begin{bmatrix} \vec{\alpha_1}^T \\ \vec{\alpha_2}^T \\ \ldots \\ \vec{\alpha_p}^T \end{bmatrix} X = A^T X$$

2. PCA의 목적인 최대 분산 구하기

$$\begin{aligned} \max_{\alpha} \{Var(Z)\} &= \max_{\alpha} \{Var(\vec{\alpha}^T X)\} \\ &= \max_{\alpha} \{\vec{\alpha}^T Var(X) \vec{\alpha}\} \\ &= \max_{\alpha} \{\vec{\alpha}^T \Sigma \vec{\alpha}\} \end{aligned}$$

3. 라그랑지안 문제로 변형

$$L = \vec{\alpha}^T \Sigma \vec{\alpha} - \lambda(\vec{\alpha}^T \vec{\alpha} - 1)$$

4. 최댓값을 구하기 위해 미분

α=데이터의 공분산행렬 Σ의 고유벡터, Λ=Σ의 고유값

$$\frac{\partial L}{\partial \vec{\alpha}} = \Sigma\vec{\alpha} - \lambda\vec{\alpha} = 0$$
$$(\Sigma - \lambda)\vec{\alpha} = 0$$

$$\begin{aligned} \Sigma^T &= \left(A^{-1}\right)^T \Lambda A^T \\ &= A\Lambda A^{-1} = \Sigma \end{aligned}$$

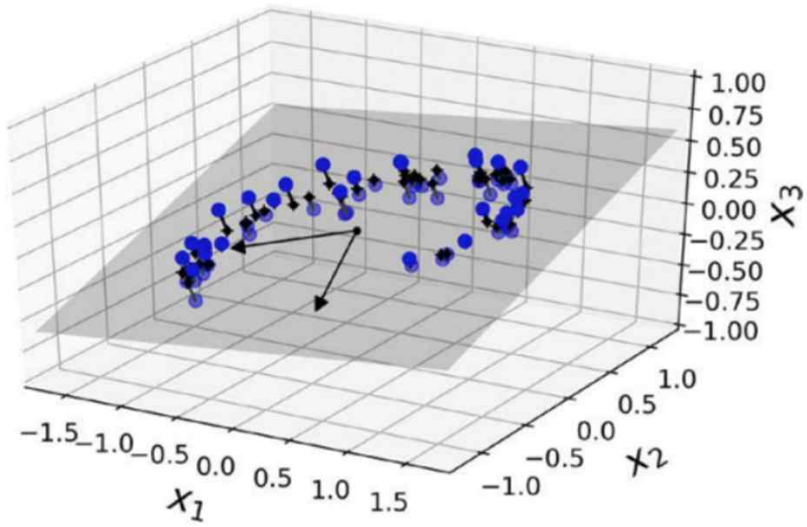
$$\therefore A^{-1} = A^T$$
$$A^T A = I$$

#Σ는 대칭 행렬이므로 다음과 같이 정리 가능
즉, 고유벡터끼리는 서로 직교이며
이는 PCA 변환에 의해 좌표축이 바뀐 데이터들은
서로 상관이 없음을 의미함

EWHA EURON

2.4 최종적인 PCA step

1. 입력 데이터 세트의 공분산 행렬을 생성
2. 공분산 행렬의 고유벡터와 고유값 계산
3. 고유값이 큰 순서대로 k개만큼의 고유벡터 추출
4. 고유벡터를 이용해서 새롭게 입력 데이터 변환



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

d차원으로 축소된 데이터 셋은
기존 데이터 행렬 X와 주성분 단위 벡터의 행렬 곱으로 얻을 수 있음

```
from sklearn.decomposition import PCA

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

sklearn에서 PCA를 적용하여 데이터 셋을 2차원으로 줄이는 코드

EWHA EURON

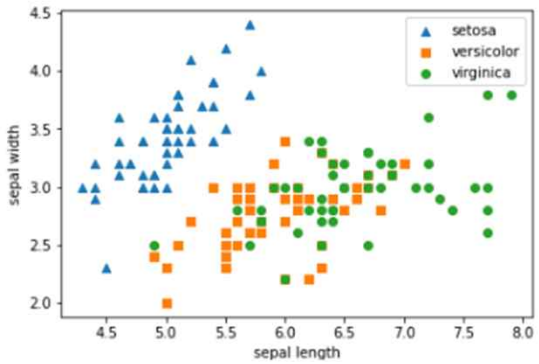
2.5 PCA 예제 - 붓꽃 데이터 세트

붓꽃 데이터 불러온 뒤, 데이터가 어떻게 분포됐는지 2차원으로 시각화

```
from sklearn.datasets import load_iris
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

# 사이킷런 내장 데이터 셋 API 호출
iris = load_iris()

# 넘파이 데이터 셋을 Pandas DataFrame으로 변환
columns = ['sepal_length','sepal_width','petal_length','petal_width']
irisDF = pd.DataFrame(iris.data , columns=columns)
irisDF['target']=iris.target
irisDF.head(3)
```

	sepal_length	sepal_width	petal_length	petal_width	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0

```
#setosa는 세모, versicolor는 네모, virginica는 동그라미로 표현
markers=['^', 's', 'o']

#setosa의 target 값은 0, versicolor는 1, virginica는 2. 각 target 별로 다른 shape으로 scatter plot
for i, marker in enumerate(markers):
    x_axis_data = irisDF[irisDF['target']==i]['sepal_length']
    y_axis_data = irisDF[irisDF['target']==i]['sepal_width']
    plt.scatter(x_axis_data, y_axis_data, marker=marker,label=iris.target_names[i])

plt.legend()
plt.xlabel('sepal length')
plt.ylabel('sepal width')
plt.show()
```




```
from sklearn.preprocessing import StandardScaler

# Target 값을 제외한 모든 속성 값을 StandardScaler를 이용하여 표준 정규 분포를 가지는 값들로 변환
iris_scaled = StandardScaler().fit_transform(irisDF.iloc[:, :-1])
```

```
iris_scaled.shape
```

```
(150, 4)
```

평균이 0, 분산이 1인 정규 분포로 속성값 변환

EWHA EURON

2.5 PCA 예제 - 붓꽃 데이터 세트

4차원의 데이터를 2차원 PCA 데이터로 변환하고 dataframe으로 데이터 값 확인

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)

#fit( )과 transform( ) 을 호출하여 PCA 변환 데이터 반환
pca.fit(iris_scaled)
iris_pca = pca.transform(iris_scaled)
print(iris_pca.shape)
```

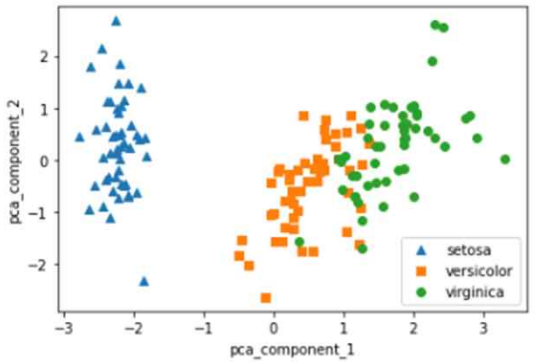
(150, 2)

```
# PCA 변환된 데이터의 컬럼명을 각각 pca_component_1, pca_component_2로 명명
pca_columns=['pca_component_1','pca_component_2']
irisDF_pca = pd.DataFrame(iris_pca, columns=pca_columns)
irisDF_pca['target']=iris.target
irisDF_pca.head(3)
```

	pca_component_1	pca_component_2	target
0	-2.264703	0.480027	0
1	-2.080961	-0.674134	0
2	-2.364229	-0.341908	0

4차원의 데이터를 2차원 PCA 데이터로 변환

```
#setosa를 세모, versicolor를 네모, virginica를 동그라미로 표시
markers=['^', 's', 'o']

#pca_component_1 을 x축, pc_component_2를 y축으로 scatter plot 수행.
for i, marker in enumerate(markers):
    x_axis_data = irisDF_pca[irisDF_pca['target']==i]['pca_component_1']
    y_axis_data = irisDF_pca[irisDF_pca['target']==i]['pca_component_2']
    plt.scatter(x_axis_data, y_axis_data, marker=marker,label=iris.target_names[i])

plt.legend()
plt.xlabel('pca_component_1')
plt.ylabel('pca_component_2')
plt.show()
```



2.5 PCA 예제 - 붓꽃 데이터 세트

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
import numpy as np

rcf = RandomForestClassifier(random_state=156)
scores = cross_val_score(rcf, iris.data, iris.target,scoring='accuracy',cv=3)
print('원본 데이터 교차 검증 개별 정확도:',scores)
print('원본 데이터 평균 정확도:', np.mean(scores))
```

```
원본 데이터 교차 검증 개별 정확도: [0.98 0.94 0.96]
원본 데이터 평균 정확도: 0.96
```

원본 붓꽃 데이터에 랜덤 포레스트를 적용한 결과

```
pca_X = irisDF_pca[['pca_component_1', 'pca_component_2']]
scores_pca = cross_val_score(rcf, pca_X, iris.target, scoring='accuracy', cv=3 )
print('PCA 변환 데이터 교차 검증 개별 정확도:',scores_pca)
print('PCA 변환 데이터 평균 정확도:', np.mean(scores_pca))
```

```
PCA 변환 데이터 교차 검증 개별 정확도: [0.88 0.88 0.88]
PCA 변환 데이터 평균 정확도: 0.88
```

2차원으로 PCA 변환한 데이터에
랜덤 포레스트를 적용한 결과

4개의 속성이 2개의 변환 속성으로 감소한 것을 고려하면
PCA 변환 후에도 원본 데이터의 특성을 상당 부분 유지하고 있음을 알 수 있음

2.6 expained_variance_ratio_ 변수

expained_variance_ratio_ 변수에는 원본 데이터셋에 대해 PC가 보존하는 분산의 비율이 들어있습니다. 다음은 가장 높게 보존하는 순으로 두가지 PC의 expained_variance_ratio_ 를 살펴보는 코드입니다.

```
pca.explained_variance_ratio_

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

= > 데이터셋 분산의 84.2%가 첫번째 PC에 놓이고, 14.6%의 데이터가 두 번째 PC를 따라 놓임을 의미

2.6 적절한 차원 수 선택하기

축소할 차원 수는 임의로 정하기 보다는 각 PC별로 표현하는 데이터 분산의 합이 충분할 때까지(ex. 95% 이상) 필요한 PC의 개수로 차원 수를 선택하는 것이 좋습니다. 물론 데이터 시각화를 위해 차원을 축소하는 경우는 보통 2, 3차원을 씁니다.

방법1) 원본 데이터셋의 분산을 95%로 유지하는데 필요한 최소한의 PC 개수, 즉 차원 수 d 를 계산

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

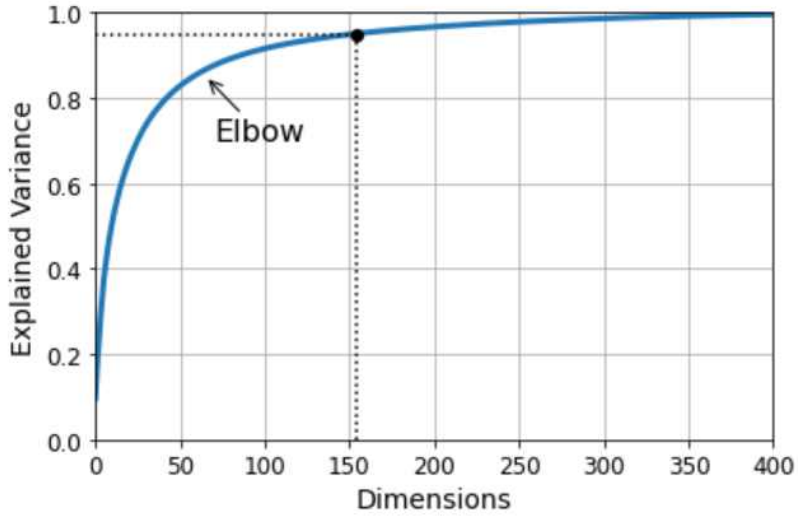
n_components를 보존하려는 분산의 비율을 지정(0~1)하여 PCA실행

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

2.6 적절한 차원 수 선택하기

축소할 차원 수는 임의로 정하기 보다는 각 PC별로 표현하는 데이터 분산의 합이 충분할 때까지(ex. 95% 이상) 필요한 PC의 개수로 차원 수를 선택하는 것이 좋습니다. 물론 데이터 시각화를 위해 차원을 축소하는 경우는 보통 2, 3차원을 씁니다.

방법2) 보존되는 분산의 비율을 차원 수에 대한 함수로 그린다. 이 그래프에서는 보존되는 분산의 비율이 빠르게 성장하다 멈추는 변곡점이 있는데, 이걸로 축소할 차원 수를 결정



2.7 압축을 위한 PCA

차원 축소는 dataset의 크기를 줄임. 이러한 압축은 SVM과 같은 Classification 알고리즘의 속도를 크게 높임.

반대로 압축된 데이터셋에 PCA 투영의 변환을 반대로 적용하여 다시 원래의 차원으로 되돌릴 수 있음. 다만 축소에서 일부 정보를 잃어버렸기 때문에 완벽한 원본 데이터셋을 얻을 순 없지만 매우 비슷

재구성 오차 (reconstruction error) : 원본 데이터와 축소 후 다시 복원된 데이터 사이의 평균 제곱 거리

PCA 역변환 공식

$$X_{recovered} = X_{d-proj} W_d^T$$

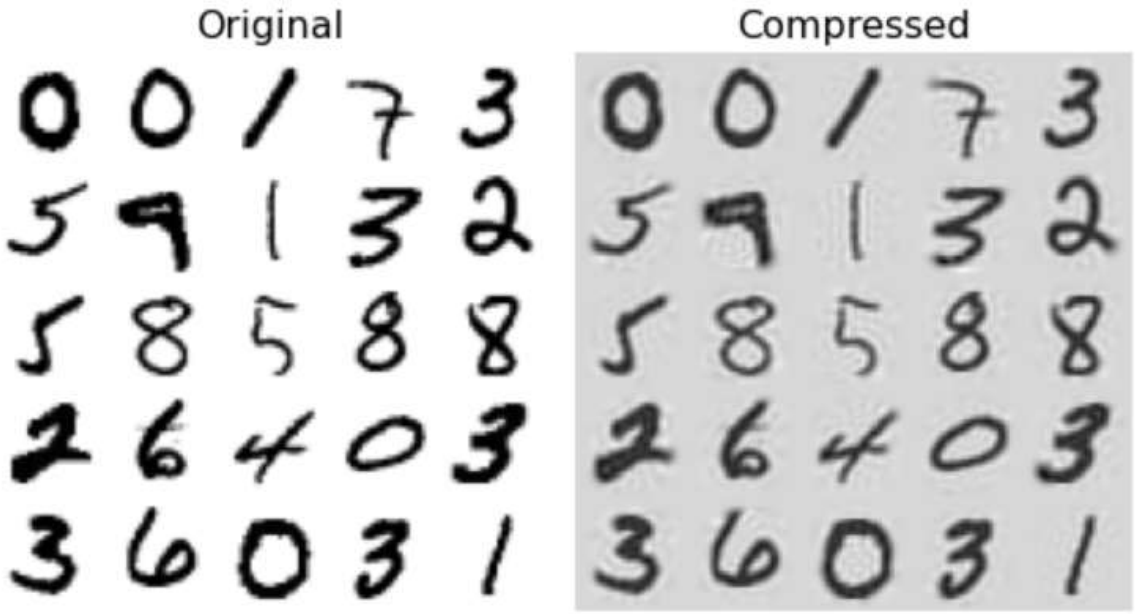
PCA 역변환 코드

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

EWHA EURON

2.7 압축을 위한 PCA

MNIST 데이터셋에 대하여 원본 데이터셋과 압축 후 복원된 결과를 비교한 그림입니다.
이미지의 품질이 손상되긴 했지만 숫자의 모양은 온전한 것을 확인할 수 있습니다.



2.8 PCA 예제 - 신용카드 데이터 분석

데이터 로드 및 컬럼명 변환

신용카드 고객 데이터 세트 :
http://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients

```
# header로 의미없는 첫행 제거, iloc로 기존 id 제거
import pandas as pd
pd.set_option('display.max_columns', 30)

df = pd.read_excel('pca_credit_card.xls', header=1, sheet_name='Data').iloc[:,1:]
print(df.shape)
df.head(3)
```

```
(30000, 24)
```

LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	PAY_5	PAY_6	BILL_AMT1	BILL_AMT2	BILL_AMT3	BILL_AMT4	BILL_AMT5	BILL_AMT6	PAY_AMT1
20000	2	2	1	24	2	2	-1	-1	-2	-2	3913	3102	689	0	0	0	0
120000	2	2	2	26	-1	2	0	0	0	2	2682	1725	2682	3272	3455	3261	0
90000	2	2	2	34	0	0	0	0	0	0	29239	14027	13559	14331	14948	15549	1518

```
df.rename(columns={'PAY_0':'PAY_1','default payment next month':'default'}, inplace=True)
y_target = df['default']
X_features = df.drop('default', axis=1)
```

2.8 PCA 예제 - 신용카드 데이터 분석

In [15]:
```
X_features.info()
```

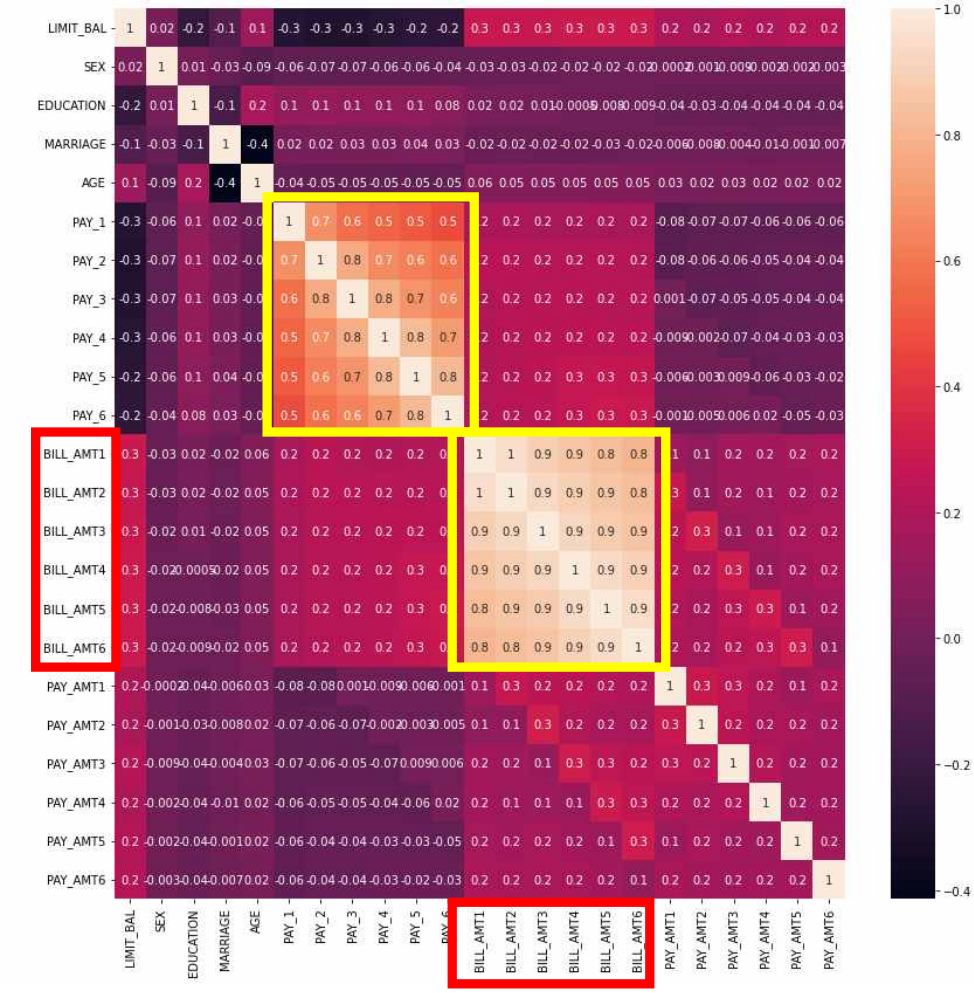
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 23 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   LIMIT_BAL  30000 non-null  int64
 1   SEX        30000 non-null  int64
 2   EDUCATION  30000 non-null  int64
 3   MARRIAGE   30000 non-null  int64
 4   AGE        30000 non-null  int64
 5   PAY_1      30000 non-null  int64
 6   PAY_2      30000 non-null  int64
 7   PAY_3      30000 non-null  int64
 8   PAY_4      30000 non-null  int64
 9   PAY_5      30000 non-null  int64
 10  PAY_6      30000 non-null  int64
 11  BILL_AMT1  30000 non-null  int64
 12  BILL_AMT2  30000 non-null  int64
 13  BILL_AMT3  30000 non-null  int64
 14  BILL_AMT4  30000 non-null  int64
 15  BILL_AMT5  30000 non-null  int64
 16  BILL_AMT6  30000 non-null  int64
 17  PAY_AMT1   30000 non-null  int64
 18  PAY_AMT2   30000 non-null  int64
 19  PAY_AMT3   30000 non-null  int64
 20  PAY_AMT4   30000 non-null  int64
 21  PAY_AMT5   30000 non-null  int64
 22  PAY_AMT6   30000 non-null  int64
dtypes: int64(23)
memory usage: 5.3 MB
```

신용카드 데이터
-> 총 23개의 features

2.8 PCA 예제 - 신용카드 데이터 분석

Feature 간 상관도 시각화

```
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

corr = X_features.corr()
plt.figure(figsize=(14,14))

sns.heatmap(corr, annot=True, fmt='.1g')
plt.show()
```



상관도를 보았을 때 PAY끼리,
BILL_AMT끼리, 특히 BILL끼리
상관도가 높음을 볼 수 있다

EWHA EURON

2.8 PCA 예제 - 신용카드 데이터 분석

In [20]:

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

#BILL_AMT1 ~ BILL_AMT6까지 6개의 속성명 생성
cols_bill = ['BILL_AMT'+str(i) for i in range(1, 7)]
print('대상 속성명:', cols_bill)

# 2개의 PCA 속성을 가진 PCA 객체 생성하고, explained_variance_ratio_ 계산을 위해 fit( ) 호출
scaler = StandardScaler()
df_cols_scaled = scaler.fit_transform(X_features[cols_bill])

pca = PCA(n_components=2)
pca.fit(df_cols_scaled)
print('PCA Component별 변동성:', pca.explained_variance_ratio_)
```

```
대상 속성명: ['BILL_AMT1', 'BILL_AMT2', 'BILL_AMT3', 'BILL_AMT4', 'BILL_AMT5', 'BILL_AMT6']
PCA Component별 변동성: [0.90555253 0.0509867 ]
```

단 2개의 PCA 컴포넌트만으로도 6개 속성의 변동성을 약 95% 이상 설명할 수 있으며 특히 첫 번째 PCA축으로 90%의 변동성을 수용할 수 있을 정도로 이 6개 속성의 상관도가 매우 높음

EWHA EURON

2.8 PCA 예제 - 신용카드 데이터 분석

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score

rcf = RandomForestClassifier(n_estimators=300, random_state=156)
scores = cross_val_score(rcf, X_features, y_target, scoring='accuracy', cv=3 )

print('CV=3 인 경우의 개별 Fold세트별 정확도:',scores)
print('평균 정확도:{0:.4f}'.format(np.mean(scores)))
```

```
CV=3 인 경우의 개별 Fold세트별 정확도: [0.8083 0.8196 0.8232]
평균 정확도:0.8170
```

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# 원본 데이터셋에 먼저 StandardScaler적용
scaler = StandardScaler()
df_scaled = scaler.fit_transform(X_features)

# 6개의 Component를 가진 PCA 변환을 수행하고 cross_val_score( )로 분류 예측 수행.
pca = PCA(n_components=6)
df_pca = pca.fit_transform(df_scaled)
scores_pca = cross_val_score(rcf, df_pca, y_target, scoring='accuracy', cv=3)

print('CV=3 인 경우의 PCA 변환된 개별 Fold세트별 정확도:',scores_pca)
print('PCA 변환 데이터 셋 평균 정확도:{0:.4f}'.format(np.mean(scores_pca)))
```

```
CV=3 인 경우의 PCA 변환된 개별 Fold세트별 정확도: [0.793  0.7958 0.8026]
PCA 변환 데이터 셋 평균 정확도:0.7971
```

전체 23개 속성이 약 1/4 수준인 6개의
PCA 컴포넌트만으로 분류 예측의
1~2% 정도의 예측 성능 저하만 발생

EWHA EURON

3. 랜덤 PCA vs 점진적 PCA vs 커널 PCA

EWHA EURON

3.1 랜덤 PCA

확률적 알고리즘을 이용하여 축소할 d차원에 대한 d개의 PC를 '근삿값'으로 빠르게 찾는다

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

- sample 개수가 많을 때는 PCA, feature 개수가 많을 때는 Randomized PCA가 유리하다

- svd_solver = randomized로 설정

- svd_solver의 기본값은 "auto"인데, 원본 데이터의 크기나 차원 수가 500보다 크고,
축소할 차원이 이것들의 80%보다 작으면 sklearn은 자동으로 랜덤 PCA 알고리즘을 사용합니다.
만약 이것을 방지하고 싶다면 "full"을 사용

3.2 점진적 PCA

데이터셋을 mini-batch로 나눈 뒤 하나 씩 주입하여 적용.

```
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)
```

- 기존의 PCA 구현의 문제(SVD 알고리즘 실행을 위해 전체 데이터셋을 메모리에 올려야 함.)를 미니 배치 단위로 나누면서 보완함

- 훈련 세트 크기가 클 때 유용하고 온라인으로 PCA적용 가능

- 메모리 효율, 실시간 추가 분석이 용이하다는 장점

EWHA EURON

3.3 커널 PCA

커널 트릭을 PCA에 적용해 차원 축소를 위한 복잡한 비선형 투영을 수행

커널 트릭이란?

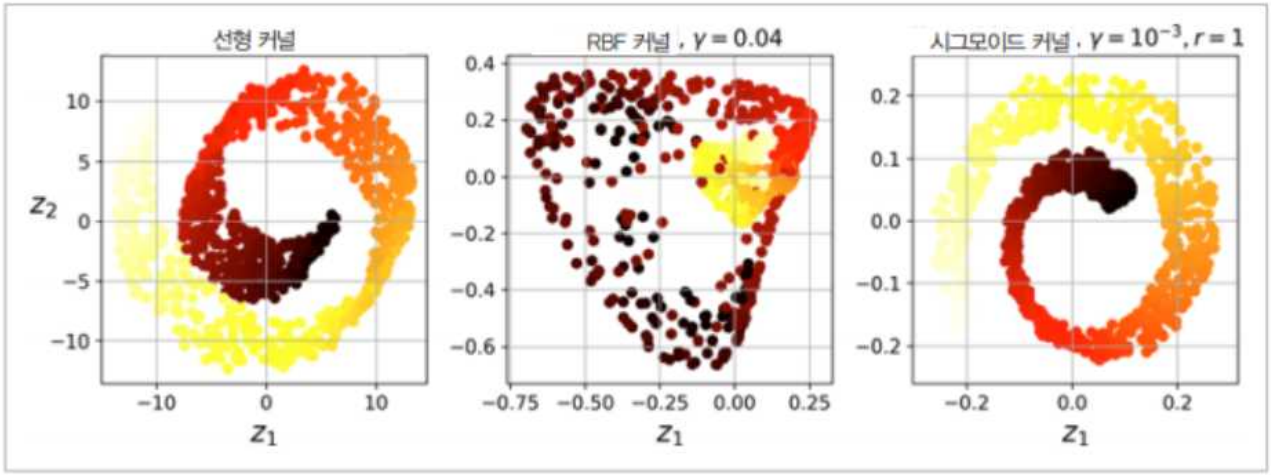
SVM은 많이 사용하는 분류 알고리즘입니다. 이 알고리즘은 비선형방식에서는 다른 방법으로 분류를 하는데 이때 사용되는 방법이 커널트릭이다. 곧 선형 분류가 불가능한 데이터에 대한 처리를 하기 위해 데이터의 차원을 증가시켜 하나의 초평면으로 분류할 수 있도록 도와주는 커널 함수를 의미합니다. 커널 함수는 마진을 최대로 하는 초평면을 구하는 알고리즘을 사용합니다. 선형분류가 힘든 데이터를 분류하기 위한 방법이다 보니 연산량이 증가하고 그에 따른 시간도 증가하지만 정확도는 높일 수 있습니다.

3.4 커널 PCA

커널 트릭을 PCA에 적용해 차원 축소를 위한 복잡한 비선형 투영을 수행

사이킷런 KernelPCA사용

```
from sklearn.decomposition import KernelPCA

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



투영 후 샘플의 군집을 유지하거나 꼬인 매니폴드에 가까운 데이터셋을 펼칠 때에도 유용하다.

3.5 커널 선택과 하이퍼파라미터 튜닝

kPCA는 Unsupervised Learning(비지도학습)이므로 어떤 커널과 하이퍼파라미터를 선택해야 좋은 성능을 내는지 명확하게 알 수 있는 기준이 없음
그러나 그리드 탐색을 이용하면 이를 보완할 수 있음

```
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'}
```

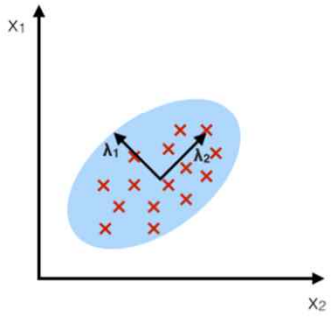
EWHA EURON

4. LDA

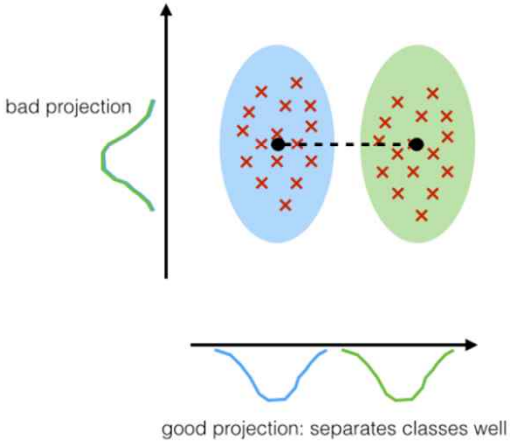
EWHA EURON

4.1 LDA(Linear Discriminant Analysis) 개요

PCA:
component axes that maximize the variance



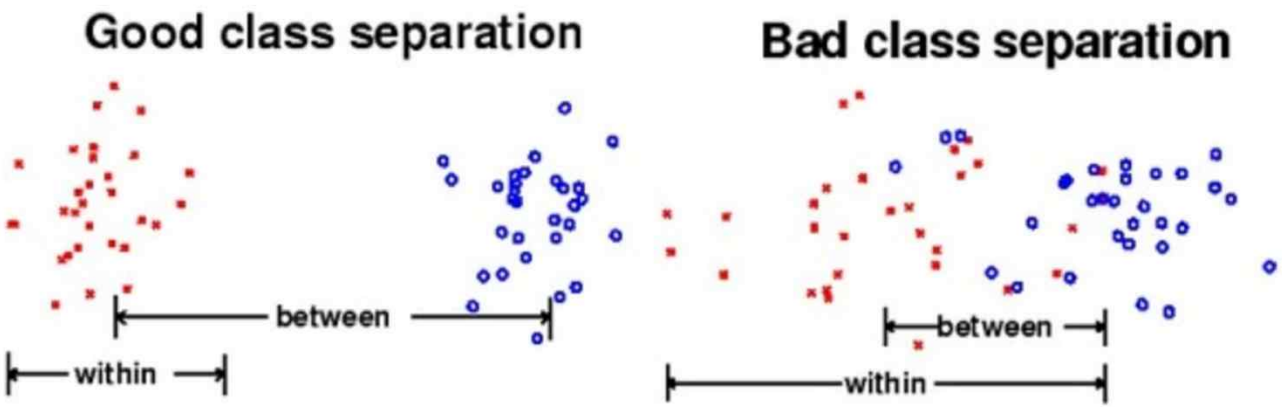
LDA:
maximizing the component axes for class-separation



- ✓ LDA(Linear Discriminant Analysis): 선형 판별 분석으로 불리며 PCA와 매우 유사.
- ✓ LDA VS PCA
  - ✓ LDA: 지도학습의 분류에서 사용하기 쉽도록 개별 클래스를 분별할 수 있는 기준을 최대한 유지하며 차원 축소. 입력 데이터의 결정 값 클래스를 최대한으로 분리할 수 있는 축을 찾음.
  - ✓ PCA: 입력 데이터 변동성의 가장 큰 축을 찾음.

EWHA EURON

4.1 LDA(Linear Discriminant Analysis) 개요



$$rate = \frac{between\text{-}class\text{-}scatter}{within\text{-}class\text{-}scatter}$$

✓ 특정 공간상에서 클래스 분리를 최대화하는 축을 찾기 위해 클래스 간 분산(between-class-scatter)과 클래스 내부 분산(within-class-scatter)의 비율을 최대화하는 방식으로 차원 축소. 클래스 간 분산은 최대한 크게, 클래스 내부 분산은 최대한 작게 만들수록 좋은 클래스 분리임.

4.1 LDA(Linear Discriminant Analysis) 개요

$$\tilde{\boldsymbol{\mu}}_i = \frac{1}{N_i}\sum_{y\in\omega_i}\mathbf{y} \qquad \tilde{\boldsymbol{\mu}} = \frac{1}{N}\sum_{\forall y}\mathbf{y}$$

$$\tilde{\mathbf{S}}_W = \sum_{i=1}^{C}\sum_{y\in\omega_i}(\mathbf{y}-\tilde{\boldsymbol{\mu}}_i)(\mathbf{y}-\tilde{\boldsymbol{\mu}}_i)^T \qquad \tilde{\mathbf{S}}_B = \sum_{i=1}^{C} N_i(\tilde{\boldsymbol{\mu}}_i-\tilde{\boldsymbol{\mu}})(\tilde{\boldsymbol{\mu}}_i-\tilde{\boldsymbol{\mu}})^T$$

$$\tilde{\mathbf{S}}_W = \mathbf{W}^T\mathbf{S}_W\mathbf{W} \qquad \tilde{\mathbf{S}}_B = \mathbf{W}^T\mathbf{S}_B\mathbf{W}$$

사영이 이제는 더 이상 스칼라가 아니며 (C-1 차원이다), 스칼라형태의 목적함수를 얻해서 분산행렬의 행렬식을 사용한다. 따라서

$$J(\mathbf{W}) = \frac{\left|\tilde{\mathbf{S}}_B\right|}{\left|\tilde{\mathbf{S}}_W\right|} = \frac{\left|\mathbf{W}^T\mathbf{S}_B\mathbf{W}\right|}{\left|\mathbf{W}^T\mathbf{S}_W\mathbf{W}\right|}$$

를 최대화하는 최적의 사영행렬

$$W^* = \left[w_1^* \mid w_2^* \mid \cdots \mid w_{C-1}^*\right] = \text{argmax}\left\{\frac{\left|W^T S_B W\right|}{\left|W^T S_W W\right|}\right\} \Rightarrow (S_B - \lambda_i S_W)w_i^* = 0$$

- ✓ 입력 데이터의 결정 값 클래스 별로 개별 피처의 평균 벡터를 기반으로 클래스 내부와 클래스 간 분산 행렬을 구함.
- ✓ 클래스 내부 분산 행렬을 Sw, 클래스 간 분산 행렬을 Sb라 하고 두 행렬 분해
- ✓ 고유값이 가장 큰 순으로 K개 추출
- ✓ 고유값이 가장 큰 순으로 추출된 고유벡터를 이용해 새롭게 입력 데이터 변환

EWHA EURON

4.2 붓꽃 데이터 세트에 LDA 적용하기

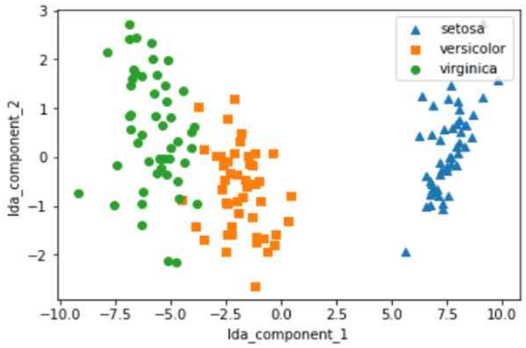
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris

iris=load_iris()
iris_scaled = StandardScaler().fit_transform(iris.data)
```

```
lda=LinearDiscriminantAnalysis(n_components=2)
lda.fit(iris_scaled, iris.target)
iris_lda = lda.transform(iris_scaled)
print(iris_lda.shape)
```

```
(150, 2)
```

- ✓ LDA는 LinearDiscriminantAnalysis 클래스로 제공됨.
- ✓ LDA는 PCA와 달리 지도학습이므로 클래스의 결정 값이 변환 시에 필요. Lda 객체의 fit() 메소드를 호출할 때 결정값이 입력됐음에 유의

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

lda_columns=['lda_component_1','lda_component_2']
irisDF_lda = pd.DataFrame(iris_lda, columns=lda_columns)
irisDF_lda['target']=iris.target

# setosa는 세모, versicolor는 네모, virginica는 동그라미로 표현
markers=['^','s','o']

# setosa의 target 값은 0, versicolor는 1, virginica는 2, 각 target 별로 다른 모양으로 산점도 표시
for i, marker in enumerate(markers):
  x_axis_data = irisDF_lda[irisDF_lda['target']==i]['lda_component_1']
  y_axis_data = irisDF_lda[irisDF_lda['target']==i]['lda_component_2']

  plt.scatter(x_axis_data, y_axis_data, marker=marker, label=iris.target_names[i])

plt.legend(loc='upper right')
plt.xlabel('lda_component_1')
plt.ylabel('lda_component_2')
plt.show()
```



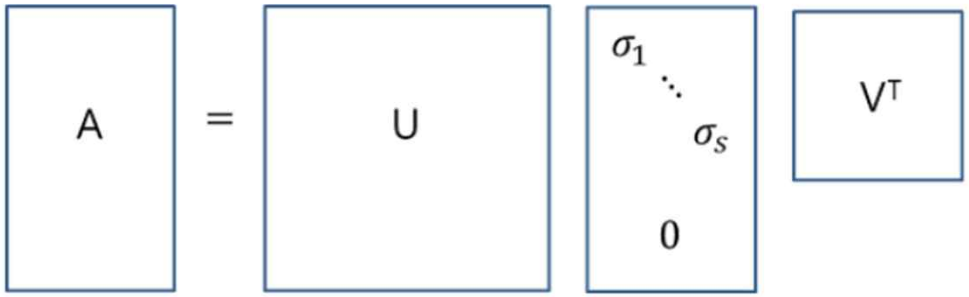
5. SVD

EWHA EURON

5.1 SVD(Singular Value Decomposition) 개요

$$A = U\Sigma V^T \qquad UU^T = U^TU = I$$

- ✓ SVD: 특이값 분해. 정방행렬뿐만 아니라 행과 열의 크기가 다른 행렬에도 적용 가능.
  - ✓ U: mxm orthogonal matrix(singular)
  - ✓ V: nxn orthogonal matrix(singular)
  - ✓ A: mxn rectangular matrix
  - ✓ Sigma: mxn diagonal matrix
- ✓ A의 차원이 mxn일때 U, sigma, V로 분해



<그림 2> full SVD

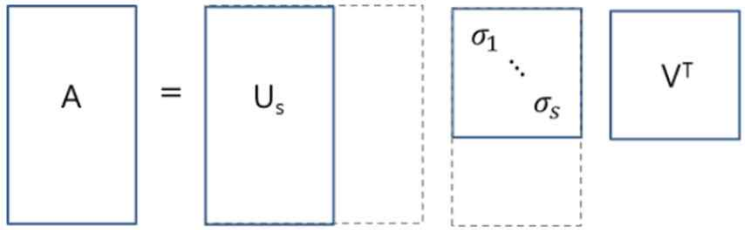
5.1 SVD(Singular Value Decomposition) 개요

$$A = U\Sigma V^T$$

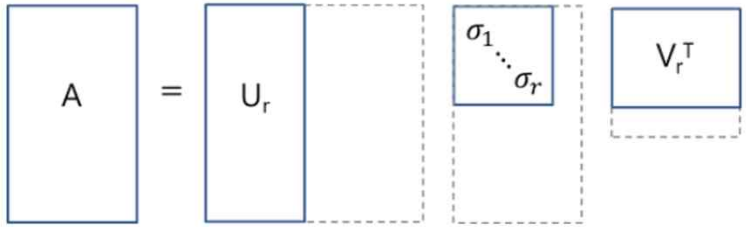
- ✓ SVD: 특이값 분해. 정방행렬뿐만 아니라 행과 열의 크기가 다른 행렬에도 적용 가능.
  - ✓ U: mxm orthogonal matrix(singular)
  - ✓ V: nxn orthogonal matrix(singular)
  - ✓ A: mxn rectangular matrix
  - ✓ Sigma: mxn diagonal matrix

- ✓ 컴팩트한 형태로 SVD를 적용하면 A의 차원이 mxn일 때, U 차원이 mxp,
- ✓ Sigma 차원이 pxp, V의 차원을 nxp로 분해.

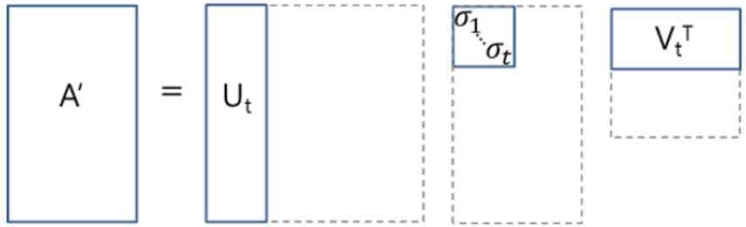
- ✓ Truncated SVD: sigma의 대각원소 중 상위 몇 개만 추출하여 U, V의 원소도
- ✓ 함께 제거해 더욱 차원을 줄인 형태로 분해.



<그림 3> thin SVD



<그림 4> Compact SVD



<그림 5> Truncated SVD

EWHA EURON

5.2 SVD 실습

```
# 넘파이의 svd 모듈 임포트
import numpy as np
from numpy.linalg import svd

# 4x4 랜덤 행렬 a 생성
np.random.seed(121)
a = np.random.randn(4,4)
print(np.round(a,3))
```

```
[[-0.212 -0.285 -0.574 -0.44 ]
 [-0.33   1.184  1.615  0.367]
 [-0.014  0.63   1.71  -1.327]
 [ 0.402 -0.191  1.404 -1.969]]
```

```
U, Sigma, Vt = svd(a)
print(U.shape, Sigma.shape, Vt.shape)
print('U matrix:₩n', np.round(U,3))
print('Sigma Value:₩n', np.round(Sigma,3))
print('V transpose matrix:₩n', np.round(Vt,3))
```

```
(4, 4) (4,) (4, 4)
U matrix:
 [[-0.079 -0.318  0.867  0.376]
 [ 0.383  0.787  0.12   0.469]
 [ 0.656  0.022  0.357 -0.664]
 [ 0.645 -0.529 -0.328  0.444]]
Sigma Value:
 [3.423 2.023 0.463 0.079]
V transpose matrix:
 [[ 0.041  0.224  0.786 -0.574]
 [-0.2    0.562  0.37   0.712]
 [-0.778  0.395 -0.333 -0.357]
 [-0.593 -0.692  0.366  0.189]]
```

✓Numpy.Linalg.svd에 파라미터로 원본 행렬을 입력하면 U 행렬, Sigma 행렬, V 전치 행렬 반환

✓Sigma 행렬은 행렬의 대각에 위치한 값만 0이 아닌 값이므로 그 값만 1차원 행렬로 표현.

5.2 SVD 실습

```
# Sigma를 다시 0을 포함한 대칭행렬로 변환
Sigma_mat = np.diag(Sigma)
a_ = np.dot(np.dot(U, Sigma_mat), Vt)
print(np.round(a_,3))
```

```
[[-0.212 -0.285 -0.574 -0.44 ]
 [-0.33   1.184  1.615  0.367]
 [-0.014  0.63   1.71  -1.327]
 [ 0.402 -0.191  1.404 -1.969]]
```

```
a[2]=a[0]+a[1]
a[3]=a[0]
print(np.round(a,3))
```

```
[[-0.212 -0.285 -0.574 -0.44 ]
 [-0.33   1.184  1.615  0.367]
 [-0.542  0.899  1.041 -0.073]
 [-0.212 -0.285 -0.574 -0.44 ]]
```

✓ 다시 복원할 때 Sigma는 원래 2차원 4x4 배열이었던 것을 1차원 배열로 만든 것이므로 다시 0을 포함한 대칭행렬로 변환 뒤 내적을 수행해야 한다.

✓ A_는 원본 행렬 a와 동일하게 복원됨.

✓ 데이터 세트가 로우 간 의존성이 있을 경우 어떻게 Sigma 값이 변하고 차원 축소가 진행되는지 알아보기 위해 행렬 업데이트

```
# 다시 SVD를 수행해 Sigma 값 확인
U, Sigma, Vt = svd(a)
print(U.shape, Sigma.shape, Vt.shape)
print('Sigma Value:₩n', np.round(Sigma,3))
```

```
(4, 4) (4,) (4, 4)
Sigma Value:
 [2.663 0.807 0.    0.   ]
```

```
# U 행렬의 경우는 Sigma와 내적을 수행하므로 Sigma의 앞 2행에 대응되는 앞 2열만 추출
U_ = U[:,:2]
Sigma_ = np.diag(Sigma[:2])
# V 전치 행렬의 경우는 앞 2행만 추출
Vt_ = Vt[:2]
print(U_.shape, Sigma_.shape, Vt_.shape)
# U, Sigma, Vt의 내적을 수행하며, 다시 원본 행렬 복원
a_ = np.dot(np.dot(U_, Sigma_), Vt_)
print(np.round(a_,3))
```

```
(4, 2) (2, 2) (2, 4)
[[-0.212 -0.285 -0.574 -0.44 ]
 [-0.33   1.184  1.615  0.367]
 [-0.542  0.899  1.041 -0.073]
 [-0.212 -0.285 -0.574 -0.44 ]]
```

✓ 다시 SVD로 분해한 결과 이전과 차원은 같지만 sigma 값 중 2개가 0으로 바뀜.

✓ Sigma 값이 00이 나온 데이터를 제외하고 복원 진행

EWHA EURON

5.2 SVD 실습

```
import numpy as np
from scipy.sparse.linalg import svds
from scipy.linalg import svd

# 원본 행렬을 출력하고 SVD를 적용할 경우 U, Sigma, Vt의 차원 확인
np.random.seed(121)
matrix = np.random.random((6,6))
print('원본 행렬:₩n', matrix)
U, Sigma, Vt = svd(matrix, full_matrices=False)
print('₩n분해 행렬 차원:', U.shape, Sigma.shape, Vt.shape)
print('₩nSigma값 행렬:', Sigma)

# Truncated SVD로 Sigma 행렬의 특이값을 4개로 하여 Truncated SVD 수행
num_components=4
U_tr, Sigma_tr, Vt_tr = svds(matrix, k=num_components)
print('₩nTruncated SVD 분해 행렬 차원:', U_tr.shape, Sigma_tr.shape, Vt_tr.shape)
print('₩nTruncated SVD Sigma값 행렬:', Sigma_tr)
matrix_tr = np.dot(np.dot(U_tr, np.diag(Sigma_tr)), Vt_tr) # output of TruncatedSVD

print('₩nTruncated SVD로 분해 후 복원 행렬:₩n', matrix_tr)
```

```
원본 행렬:
 [[0.11133083 0.21076757 0.23296249 0.15194456 0.83017814 0.40791941]
 [0.5557906  0.74552394 0.24849976 0.9686594  0.95268418 0.48984885]
 [0.01829731 0.85760612 0.40493829 0.62247394 0.29537149 0.92958852]
 [0.4056155  0.56730065 0.24575605 0.22573721 0.03827786 0.58098021]
 [0.82925331 0.77326256 0.94693849 0.73632338 0.67328275 0.74517176]
 [0.51161442 0.46920965 0.6439515  0.82081228 0.14548493 0.01806415]]

분해 행렬 차원: (6, 6) (6,) (6, 6)

Sigma값 행렬: [3.2535007  0.88116505 0.83865238 0.55463089 0.35834824 0.0349925 ]

Truncated SVD 분해 행렬 차원: (6, 4) (4,) (4, 6)

Truncated SVD Sigma값 행렬: [0.55463089 0.83865238 0.88116505 3.2535007 ]

Truncated SVD로 분해 후 복원 행렬:
 [[0.19222941 0.21792946 0.15951023 0.14084013 0.81641405 0.42533093]
 [0.44874275 0.72204422 0.34594106 0.99148577 0.96866325 0.4754868 ]
 [0.12656662 0.88860729 0.30625735 0.59517439 0.28036734 0.93961948]
 [0.23989012 0.51026588 0.39697353 0.27308905 0.05971563 0.57156395]
 [0.83806144 0.78847467 0.93868685 0.72673231 0.6740867  0.73812389]
 [0.59726589 0.47953891 0.56613544 0.80746028 0.13135039 0.03479656]]
```

- ✓ Truncated SVD를 이용해 행렬 분해
- ✓ Sigma 행렬에 있는 대각원소 즉 특이값 중 상위 일부 데이터만 추출해 분해하는 방식

- ✓ 원본 행렬을 정확하게 다시 복원할 수는 없으나 상당한 수준으로 원본 행렬 근사 가능
- ✓ 원래 차원의 차수에 가깝게 잘라낼수록 원본 행렬에 더 가깝게 복원 가능

- ✓ Truncated SVD는 scipy.sparse.linalg.svds를 이용

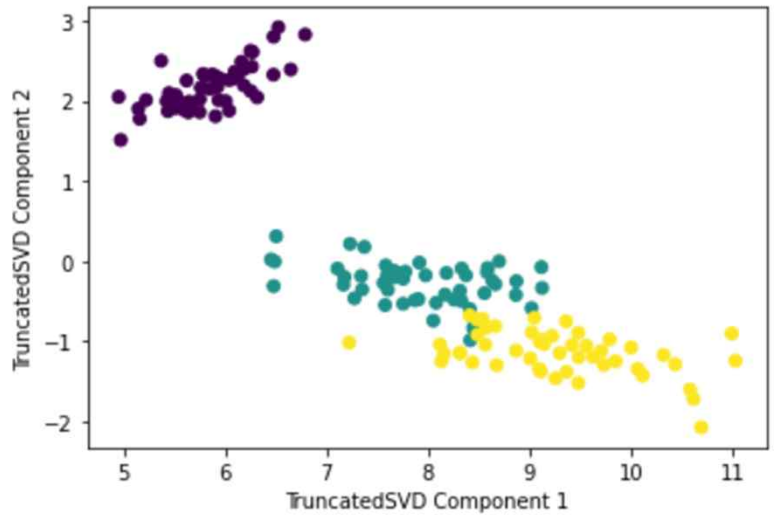
- ✓ 임의의 원본 행렬 6x6을 normal SVD로 분해해 특이값 확인 후 다시 Truncated SVD로 분해 후 복원된 데이터와 원본 데이터 비교.

- ✓ Truncated SVD로 분해된 행렬의 sigma 형태가 (6,)가 아닌 (4,)로 근사적으로 복원됨을 확인.

5.3 사이킷런 TruncatedSVD 클래스를 이용한 변환

```
from sklearn.decomposition import TruncatedSVD, PCA
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
%matplotlib inline

iris=load_iris()
iris_ftrs = iris.data
# 2개의 주요 컴포넌트로 TruncatedSVD 변환
tsvd = TruncatedSVD(n_components=2)
tsvd.fit(iris_ftrs)
iris_tsvd = tsvd.transform(iris_ftrs)

# 산점도 2차원으로 TruncatedSVD 변환된 데이터 표현. 품종을 색깔로 구분
plt.scatter(x=iris_tsvd[:,0], y=iris_tsvd[:,1], c=iris.target)
plt.xlabel('TruncatedSVD Component 1')
plt.ylabel('TruncatedSVD Component 2')
```

```
Text(0, 0.5, 'TruncatedSVD Component 2')
```



- ✓ 사이킷런의 TruncatedSVD 클래스는 원본 행렬을 분해한 U, Sigma, Vt 행렬 반환 하지 않음.
- ✓ PCA 클래스와 유사하게 fit(), transform()을 호출해 원본 데이터를 몇 개의 주요 컴포넌트로 차원 축소해 변환.
- ✓ 원본 데이터를 Truncated SVD 방식으로 분해된 U*Sigma 행렬에 선형 변환해 생성

- ✓ 붓꽃 데이터 세트를 TruncatedSVD를 이용해 변환
- ✓ PCA와 유사하게 변환 후 품종별로 어느 정도 클러스터링이 가능할 정도로 각 변환 속성으로 뛰어난 고유성을 가지고 있음.

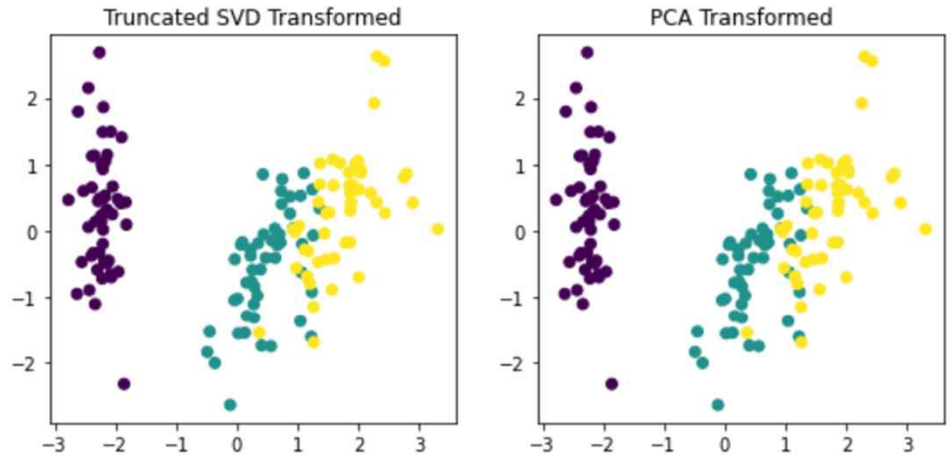
EWHA EURON

5.3 사이킷런 TruncatedSVD 클래스를 이용한 변환

```
from sklearn.preprocessing import StandardScaler

# 붓꽃 데이터를 StandardScaler로 변환
scaler = StandardScaler()
iris_scaled = scaler.fit_transform(iris_ftrs)

# 스케일링된 데이터를 기반으로 TruncatedSVD 변환 수행
tsvd = TruncatedSVD(n_components=2)
tsvd.fit(iris_scaled)
iris_tsvd = tsvd.transform(iris_scaled)

# 스케일링된 데이터를 기반으로 PCA 변환 수행
pca = PCA(n_components=2)
pca.fit(iris_scaled)
iris_pca = pca.transform(iris_scaled)

# TruncatedSVD 변환 데이터를 왼쪽에, PCA 변환 데이터를 오른쪽에 표현
fig, (ax1, ax2) = plt.subplots(figsize=(9,4), ncols=2)
ax1.scatter(x=iris_tsvd[:,0], y=iris_tsvd[:,1], c=iris.target)
ax2.scatter(x= iris_pca[:,0], y=iris_pca[:,1], c=iris.target)
ax1.set_title('Truncated SVD Transformed')
ax2.set_title('PCA Transformed')
```

- ✓ 붓꽃 데이터를 스케일링으로 변환 후 TrunscatedSVD와 PCA 클래스 변환
- ✓ 2개의 변환 행렬 값과 원본 속성별 컴포넌트 비율값을 실제로 서로 비교해 보면 거의 같음.

```
Text(0.5, 1.0, 'PCA Transformed')
```




```
print((iris_pca - iris_tsvd).mean())
print((pca.components_ - tsvd.components_).mean())
```

```
2.3224709192840956e-15
2.7755575615628914e-17
```

- ✓ PCA 값과 Truncated의 값의 차가 거의 0에 가까운 값이므로 2개의 변환이 서로 동일함.
- ✓ 데이터 세트가 스케일링으로 데이터 중심이 동일해지면 SVD와 PCA는 동일한 변환 수행.

EWHA EURON

6. NMF

EWHA EURON

6.1 NMF(Non-Negative Matrix Factorization) 개요

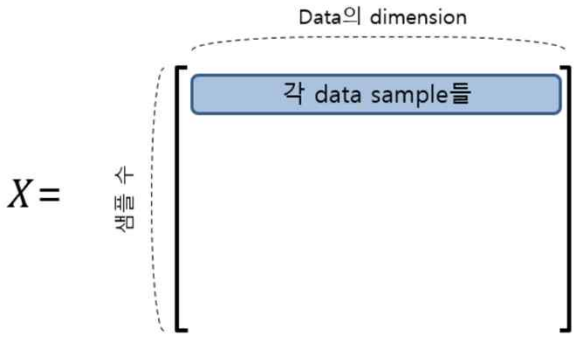


그림 1. 데이터 행렬 X의 형태

$$X = WH$$

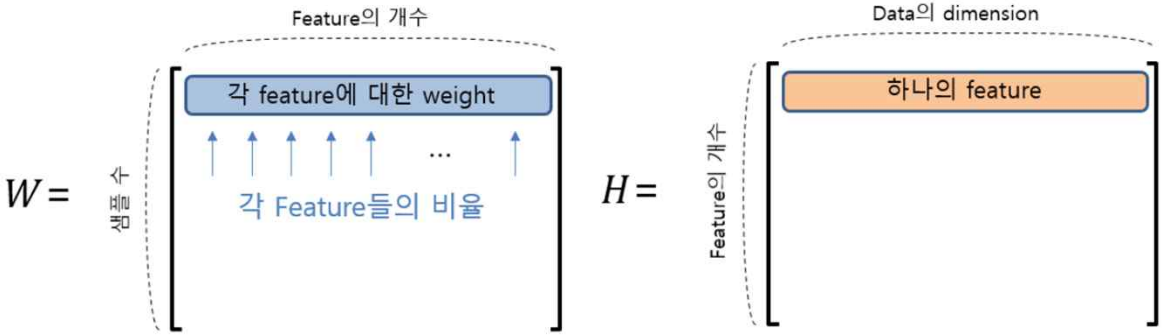


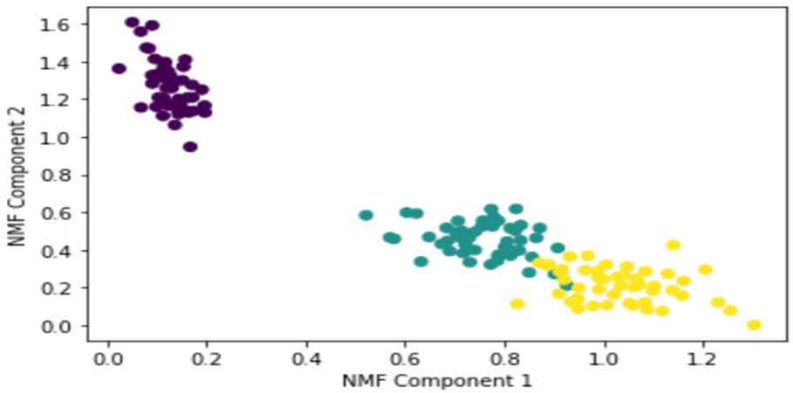
그림 2. NMF를 이용해 분해된 행렬 W와 H의 형태와 각 행렬의 행의 의미

- ✓ NMF: 음수 미포함 행렬 분해로 음수를 포함하지 않는 행렬 X(V)를 음수를 포함하지 않는 행렬 W와 H의 곱으로 분해하는 알고리즘
- ✓ Truncated SVD와 같이 낮은 랭크를 통한 행렬 근사 방식의 변형
- ✓ 행렬 분해(Matrix Factorization): 일반적으로 SVD와 같은 행렬 분해 기법 통칭
- ✓ 일반적으로 길고 가는 행렬 W, 작고 넓은 행렬 H로 분해됨. 분해된 행렬은 잠재 요소를 특성으로 가짐.
- ✓ W는 원본 행에 대해 이 잠재 요소의 값이 얼마나 되는지에 대응. H는 잠재 요소가 원본 열로 어떻게 구성됐는지 나타내는 행렬

6.2 NMF 실습

```
from sklearn.decomposition import NMF
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
%matplotlib inline

iris = load_iris()
iris_ftrs = iris.data
nmf = NMF(n_components=2)
nmf.fit(iris_ftrs)
iris_nmf = nmf.transform(iris_ftrs)
plt.scatter(x=iris_nmf[:,0],y=iris_nmf[:,1], c=iris.target)
plt.xlabel('NMF Component 1')
plt.ylabel('NMF Component 2')
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/decomposition/_
  FutureWarning,
/usr/local/lib/python3.7/dist-packages/sklearn/decomposition/_
  ConvergenceWarning,
Text(0, 0.5, 'NMF Component 2')
```



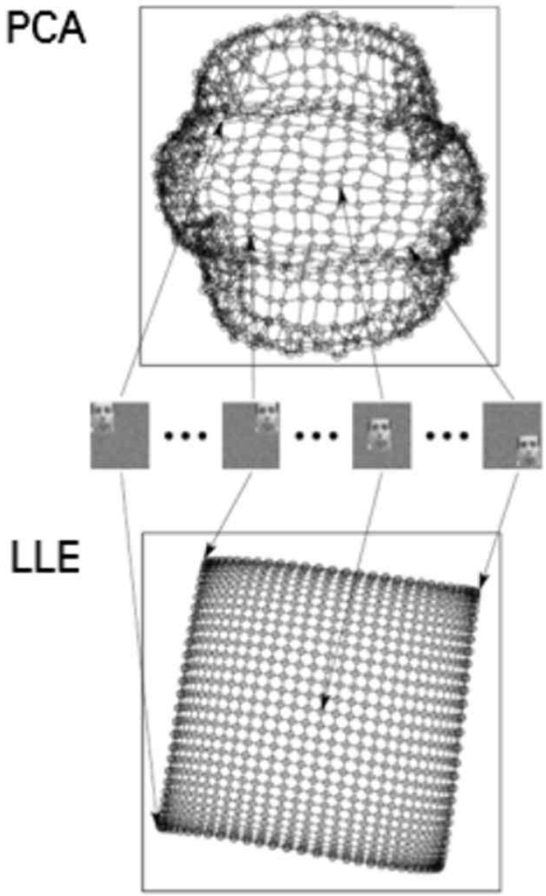
- ✓ NMF는 SVD와 유사하게 차원 축소를 통한 잠재 요소 도출로 이미지 변환 및 압축, 텍스트의 토픽 도출 등의 영역에서 사용됨
- ✓ 이미지 압축을 통한 패턴 인식, 텍스트의 토픽 모델링 기법, 문서 유사도 및 클러스터링에 잘 사용됨.
- ✓ 사이킷런에서 NMF는 NMF 클래스를 통해 지원됨. 붓꽃 데이터를 NMF를 이용해 2개의 컴포넌트로 변환하고 시각화한 결과.
- ✓ Ex) 영화 추천(잠재 요소 기반의 추천 방식)
  - ✓ 사용자의 상품 평가 데이터 세트인 사용자-평가 순위 데이터 세트를 분해
  - ✓ 사용자가 평가하지 않은 상품에 대한 잠재적 요소 추출해 평가 순위 예측
  - ✓ 높은 순위로 예측된 상품 추천

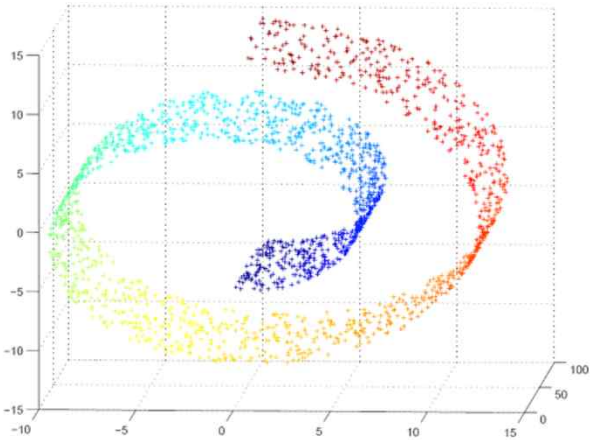
EWHA EURON

7. LLE + a

EWHA EURON

7.1 LLE(Locally Linear Embedding)





https://t-lab.tistory.com/26

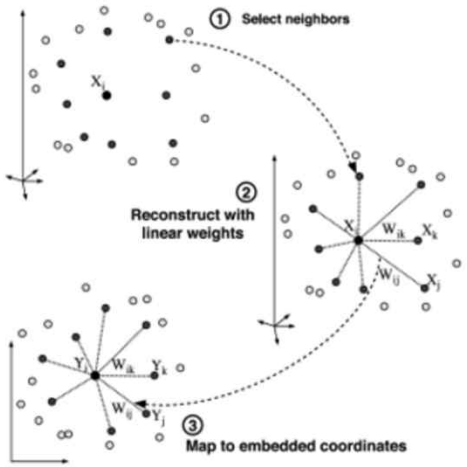
✓ 데이터의 형태가 이렇게 매니폴드의 형태라면 PCA 차원 축소는 정확하지 않을 수 있음.

✓ 이렇게 Nonlinear한 데이터의 형태일 경우 차원을 축소할 때 T-SNE, LLE, ISOMAP등의 방법을 사용

✓ LLE: 고차원의 공간에서 인접한 데이터들 사이의 선형적 구조를 보존하며 저차원으로 임베딩 하는 것. 이미지 축소에 많이 사용됨.

7.1 LLE(Locally Linear Embedding)

LLE Algorithm Step



LLE Algorithm Step

Step 1: Compute the neighbors of each data point $\overrightarrow{X_i}$

Step 2: Compute the wieght W_{ij} that best reconstruct each data point from its neighbors, minimizing the cost function by constrained linear fits

$$\varepsilon(W) = \sum_i \left| \overrightarrow{X_i} - \sum_j W_{ij} \overrightarrow{X_j} \right|^2$$

$s.t\ W_{ij}=0$ *if* $\overrightarrow{X_j}$ *does not belong to the neighbors of* $\overrightarrow{X_i}$

$\sum_j W_{ij} = 1, for\ all\ i$

Step 3: Compute the vectors best reconstructed by the weights W_{ij}, minimizing the quadratic form by its bottom nonzero eigenvectors

$$\Phi(Y) = \sum_i \left| \overrightarrow{Y_i} - \sum_j W_{ij} \overrightarrow{Y_j} \right|^2$$

✓ 각 데이터 포인터 점에서 K개의 이웃을 찾는다.

✓현재의 데이터를 나머지 k개의 데이터의 가중치의 합으로 뺄 때 최소가 되는 가중치 W matrix를 찾는다

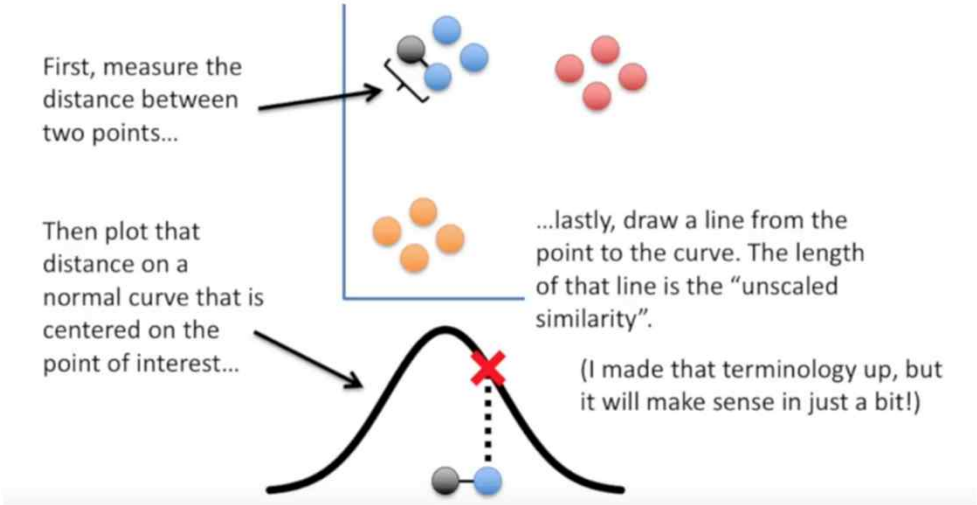
✓가중치를 최대한 가져가며 차원을 줄이는데 이 때 차원 축소 전의 점이 X라면 차원 축소된 후의 점은 Y로 표현

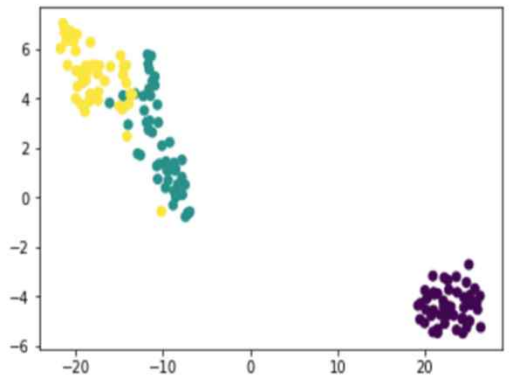
✓차원 축소된 Yi와의 값 차이를 최소화하는 Y 찾음.

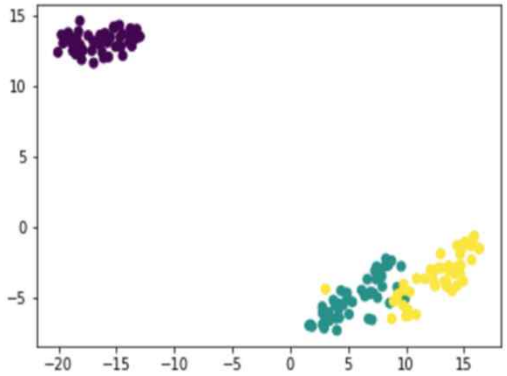
EWHA EURON

7.2 T-SNE

https://bcho.tistory.com/1210







✓ PCA 기반 차원 감소의 문제점으로 차원이 감소되며 군집화 된 데이터들이 뭉게져 제대로 구별할 수 없는 문제점이 있음,

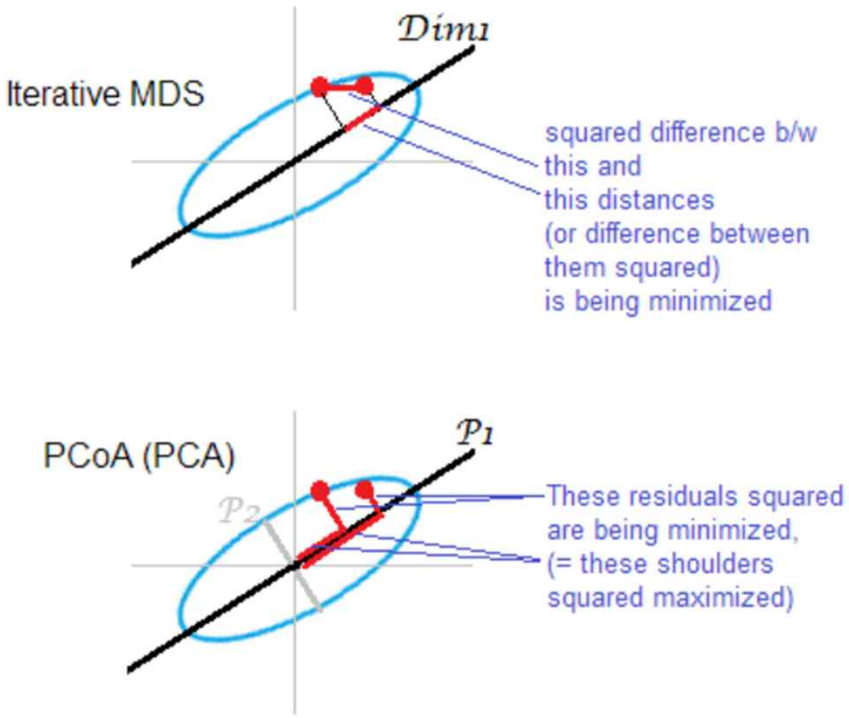
✓ 이러한 문제를 해결하기 위한 차원 감소 방법으로 T-SNE가 있다.

✓ 점을 하나 선택하고 점에서부터 다른 점까지의 거리를 측정
✓ T 분포 그래프를 이용해 기준점을 T 분포 상의 가운데에 위치시키고, 기준점으로부터 상대점까지 거리에 있는 T 분포의 값을 선택해 이 값을 친밀도로 하고 친밀도가 가까운 값끼리 묶음.
✓ 군집이 중복되지 않음. 매번 계산할 때마다 축의 위치가 바뀌어 다른 모양으로 나타남. 데이터의 군집성 같은 특성들은 유지되어 시각화를 통한 데이터 분석에 유용.

✓ 군집에 대한 특성은 그대로 유지되지만 값 자체는 변화됨.

EWHA EURON

7.3 MDS(Multi Dimensional Scaling)



https://velog.io/@swan9405/MDS-Multidimensional-Scaling

- ✓ MDS(Multi Dimensional Scaling): 다차원 스케일링. 출력이 없는 입력 상태에서의 스케일 문제를 해결하는 것으로 비지도 학습 범주에 해당된다.
- ✓ 원 공간에서 모든 점들 간에 정의된 거리가 주어졌을 때 임베딩 공간에서의 거리와 거리 행렬의 차이가 최소가 되는 임베딩을 학습.
- ✓ MDS는 모든 점들 간의 거리 정보를 보존한다. 낮은 차원에서 자료들의 거리가 멀리 떨어져 위치한다는 것은 비유사성이 높다는 뜻이고, 자료가 가까울수록 비유사성이 낮다.
- ✓ MDS를 이용해 데이터를 시각화할 때 데이터들의 유사도를 확인할 수 있다는 점이 최대 장점이다.
- ✓ PCA는 원 데이터의 분산을 보존하고 MDS는 인스턴스의 거리 정보를 보존한다.

7.4 IsoMap

https://woosikyang.github.io/first-post.html

https://wordbe.tistory.com/entry/Manifold-Learning-IsoMap-LLE-t-SNE-%EC%84%A4%EB%AA%85

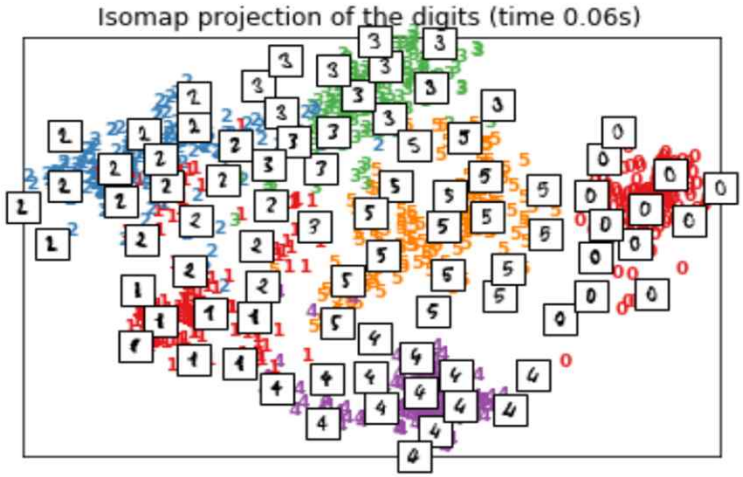
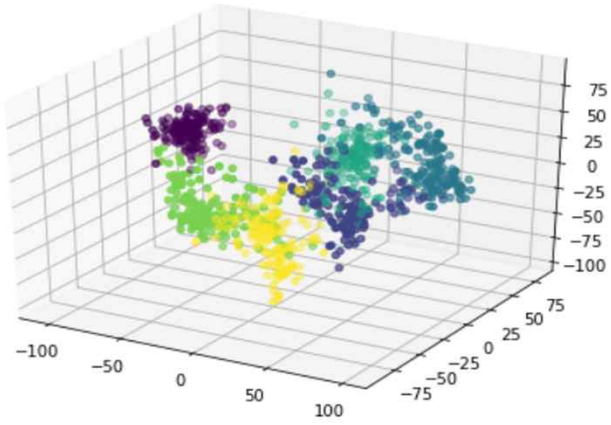


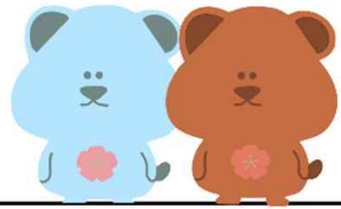
그림5



- ✓ IsoMap: MDS(다차원 스케일링)와 PCA(주성분 분석)의 확장이자 두 방법론을 결합한 방법론. 두 특징을 결합하여 모든 점 사이의 측지선 거리를 유지하는 더 낮은 차원의 임베딩 추구.
- ✓ 측지 거리: 두 측점 사이의 타원체면을 따라 이루어진 거리
- ✓ 각 점에 대한 K-nearest neighbor을 찾고, 유클리디언 거리를 계산해 거리 행렬 M에 기록.
- ✓ 두 점 사이 최단 경로를 Floyd 알고리즘을 이용해 M의 0인 값들 계산. M은 nxn 행렬로 데이터 분포의 비선형 구조를 반영
- ✓ Floyd 알고리즘은 시간이 매우 오래 걸린다는 단점이 있다.
- ✓ IsoMap이 사용하는 distance matrix M은 밀집행렬로 고유 벡터를 구하는 데 어려움이 있다.

EWHA EURON

THANK YOU



EWHA EURON