# 💻Week10_예습과제_이재린

| | |
|---|---|
| 태그 | 예습과제 |
| 주차 | 9주차 |

# Ch06. 차원 축소

## ▼ 01. 차원 축소 개요

### 차원 축소란?

> 💡 매우 많은 피처로 구성된 다차원 데이터 세트의 차원을 축소해 새로운 차원의 데이터 세트를 생성하는 것
>
> 데이터 압축을 의미하는 것이 아닌 데이터를 잘 설명할 수 있는 잠재적 요소를 추출

### 그래서 왜 함?

- 차원이 증가 > 데이터 포인트 간의 거리가 기하급수적으로 멀어짐(희소한 구조) > 적은 차원에서 학습된 모델보다 예측 신뢰도 ⬇️
- 피처가 많을수록 피처간 상관관계 ⬆ > 선형 회귀의 경우 다중 공선성 문제 > 모델의 예측 성능 ⬇️
- 다차원의 피처를 차원 축소해 피처 수를 줄이면 직관적으로 데이터 해석 가능 & 학습 데이터의 크기가 줄어들어 학습에 필요한 처리 능력도 ⬇️

### 피처 선택과 피처 추출

- 피처 선택 : 기존 피처를 저차원의 중요 피처로 압축해서 추출 >> 특정 피처에 종속성이 강한 불필요한 피처는 아예 제거, 데이터 특징을 잘 나타내는 주요 피처만 선택 >> 완전히 다른 값으로
- 피처 추출 : not just for **compression,** 함축적인 요약 특성으로 또 다른 공간으로 매핑해 추출

### 차원 축소 알고리즘 : 잠재적 요소를 찾는

1. 차원 축소 알고리즘의 종류

- PCA
- SVD
- NMF

2. 차원 축소 알고리즘은 어디에 사용되나

- 이미지 변환 및 압축 : 원본 이미지보다 적은 차원  >> 분류 수행 시 과적합 영향력이 작아짐 >> 예측 성능 ⬆
- 텍스트 문서의 숨겨진 의미를 추출 : 문서 내 단어들의 구성에서 숨겨져 있는 시맨틱 의미나 토픽을 잠재 요소로 간주하고 찾아냄 (SVD, NMF)

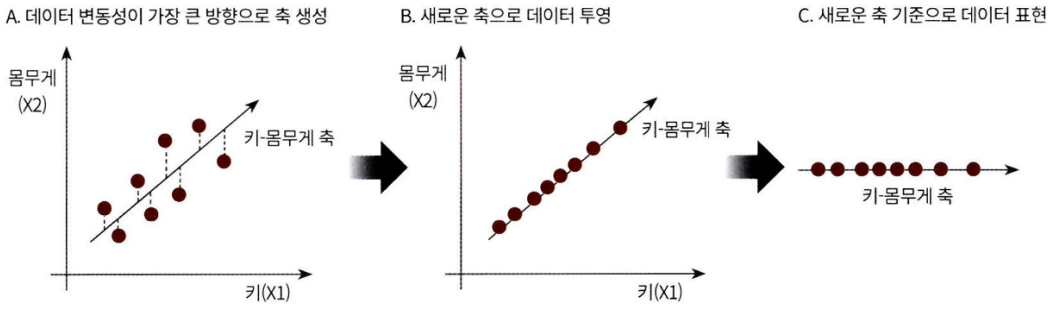
## ▼ 02. PCA

### PCA 개요

> 💡 가장 대표적인 차원 축소 기법
>
> 여러 변수 간에 존재하는 상관관계를 이용해 주성분을 추출 > 차원 축소

- PCA는 가장 높은 분산을 갖는 데이터의 축을 찾아 이 축으로 차원을 축소함 > PCA의 주성분으로 데이터의 특성을 가장 잘 나타냄

ex) 2개의 피처를 한 개의 주성분을 가진 데이터 세트로 차원 축소하는 과정

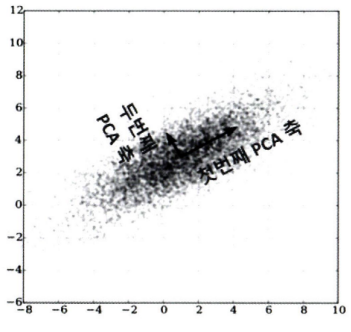



## 주성분 분석

- 주성분 분석이란? 원본 데이터의 피처 개수에 비해 매우 작은 주성분으로 원본 데이터의 총 변동성을 대부분 설명할 수 있는 분석법
- how?

1. 가장 큰 데이터 변동성을 기반으로 첫 번째 벡터 축을 생성
2. 두 번째 벡터 축은 첫 번째에 직교 벡터로
3. 세 번째는 두 번째 축과 직교하는 벡터

>> 생성된 벡터 축에 원본 데이터를 투영하면 벡터 축의 개수만큼 차원으로 원본 데이터가 차원 축소됨




## 선형대수 관점에서 해석한 PCA

- 입력 데이터의 공분산 행렬을 고유값 분해하고, 구한 고유 벡터에 입력 데이터를 선형 변환하는 것
- 고유벡터 : PCA의 주성분 벡터 > 입력 데이터의 분산이 큰 방향을 나타냄
- 고윳값 :  고유벡터의 크기 & 입력 데이터의 분산

## 선형 변환, 공분산 행렬, 고유벡터

1. 선형 변환 : 특정 벡터에 행렬 A를 곱해 새로운 벡터로 변환하는 것 > 행렬을 바로 공간으로 가정
2. 공분산 행렬 : 공분산은 두 변수 간의 변동을 의미 > 공분산 행렬이란 여러 변수와 고나련도니 공분산을 포함하는 정방형 행렬

- 공분산 행렬은 정방행렬이며 대칭행렬
- 행렬의 대각선 원소는 각 변수 X,Y,Z의 분산을 의미, 나머지는 각 변수 쌍 간의 공분산

| | X | Y | Z |
|---|---|---|---|
| X | 3.0 | −0.71 | −0.24 |
| Y | −0.71 | 4.5 | 0.28 |
| Z | −0.24 | 0.28 | 0.91 |

3. 고유벡터 : 행렬 A를 곱하더라도 방향이 변하지 않고 그 키기만 변하는 벡터 Ax=ax (x는 고유벡터, a는 스칼라값) > 정방행렬은 차원의 수만큼 고유벡터를 갖는다.

- 대칭행렬은 항상 고유벡터를 직교행렬로, 고유값을 정방 행렬로 대각화할 수 있다.

공분산 행렬 C의 분해 : P는 n*n 직교 행렬, 시그마는 정방행렬

$$C = P \sum P^T$$

>> 공분산 C는 직교행렬 * 고유값 정방 행렬 * 고유벡터 직교 행렬의 전치 행렬로 분해

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

>> 입력 데이터의 공분산 행렬이 고유 벡터와 고유값으로 분해되며, 분해된 고유벡터를 이용해 입력 데이터를 선형 변환하는 방식이 PCA 임

## PCA의 수행

1. 입력 데이터 세트의 공분산 행렬을 생성
2. 공분산 행렬의 교유벡터와 고유값을 계산
3. 고유값이 가장 큰 순으로 K개(PCA 변환 차수만큼)만큼 고유벡터를 추출
4. 고유값이 가장 큰 순으로 추출된 고유벡터를 이용해 새롭게 입력 데이터를 변환

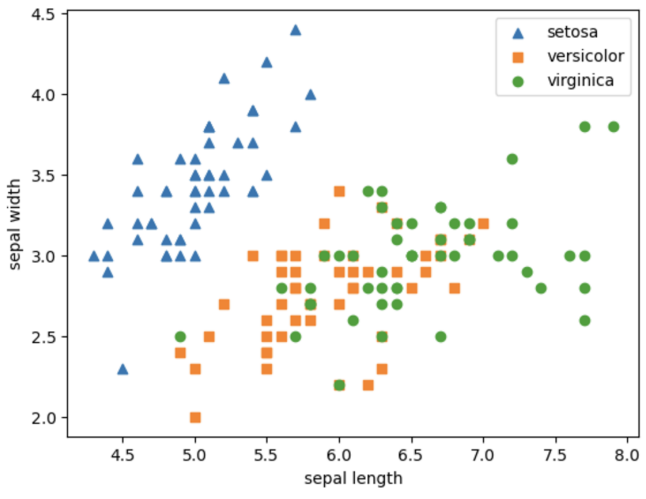
## 붓꽃데이터에 적용

1. 데이터 로드

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

iris = load_iris()

# 넘파이 데이터 셋을 pd df로 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 |

2. 각 품종에 따른 데이터 시각화




- s : width가 3.0보다 크고 length가 6.0이하인 곳에 일정하게 분포 / 나머지는 분류가 어려운 복잡한 조건

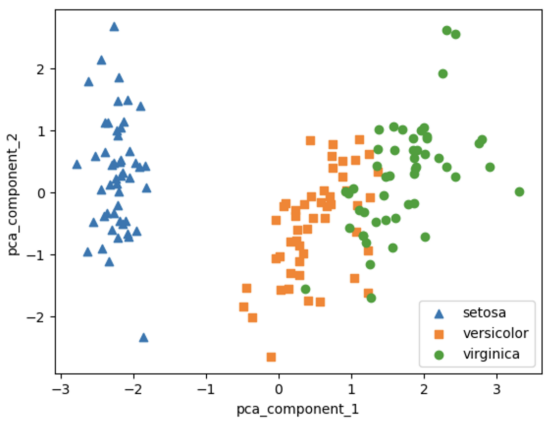
3. 개별 속성을 동일한 스케일로 변환 standardscaeler > 스케일링 적용 후 PCA 적용해 4차원 > 2차원으로 변환

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

# 주성분 분석 전에 표준화 (평균0, 편차 1의 표준정규분포로 각 칼럼데이터 표준화시키기)
iris_scaled = StandardScaler().fit_transform(irisDF.iloc[:,:-1])

# 주성분 분석, 피처 n_components : 몇 차원으로 줄일건지
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)

# PCA 환된 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)
```

```
(150, 2)
```

| | 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 |

4. PCA로 변환된 데이터 세트를 2차원상에서 시각화 : X축 pca_component_1, Y축 pca_component_2




```
# 각 컴포넌트별 차지하는 변동성 비율 출력하기
print(pca.explained_variance_ratio_)
```

```
[0.72962445 0.22850762]
```

>> 1이 특히 변동성 잘 반영 / 두개만 있어도 변동성의 약 95%를 설명\

5. 원본 vs PCA 적용 데이터 세트 정확도 결과 비교

- RandomForestClassifier / cross_val_score() 로 3개의 교차 검증 세트로 반환

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

rcf=RandomForestClassifier(random_state=156)
# 원본
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))

# PCA로 2차원으로 변환된 데이터 세트
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('원본 데이터 교차 검증 개별 정확도:', scores_pca)
print('원본 데이터 평균 정확도:',np.mean(scores_pca))
```

```
원본 데이터 교차 검증 개별 정확도: [0.98 0.94 0.96]
원본 데이터 평균 정확도: 0.96
원본 데이터 교차 검증 개별 정확도: [0.88 0.88 0.88]
원본 데이터 평균 정확도: 0.88
```

>> 속성 개수가 반타작난 것 치고, 정확도는 10%밖에 하락하지 않음

## 더 많은 피처를 가진 데이터 세트에 적용 : 신용카드 데이터 세트

1. 데이터 로드 후 피처 이름 변경 및 타깃 분리

```
# hearder로 의미 없는 첫 행 제거, iloc으로 기존 id 제거
import pandas as pd

df = pd.read_excel('/Users/bluecloud/Documents/대학/유런/데이터셋/credit_card.xls',sheet_name='Data',
print(df.shape)
df.head(3)
```

```
(30000, 24)
```

| T4 | BILL_AMT5 | BILL_AMT6 | PAY_AMT1 | PAY_AMT2 | PAY_AMT3 | PAY_AMT4 | PAY_AMT5 | PAY_AMT6 | default payment next month |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 689 | 0 | 0 | 0 | 0 | 1 |
| 72 | 3455 | 3261 | 0 | 1000 | 1000 | 1000 | 0 | 2000 | 1 |
| 31 | 14948 | 15549 | 1518 | 1500 | 1000 | 1000 | 1000 | 5000 | 0 |

- target 값으로 default~ 즉 다음달 연체 여부 속성을 target으로, 이름 길어서 default로
- PAY_0 칼럼을 PAY_1으로 변환

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

2. 각 속성 간의 상관도를 구해 시각화

- BILL_AMT1~6까지의 상관도가 매우 높다 / PAY1~6까지의 속성도 상관도 높은편 >> 높은 상관도의 속성들로 변동성 수용 가능 >> BILL_AMT1~6까지 6개 속성을 2개의 컴포넌트로 PCA 변환 후 개별 컴포넌트의 변동성을 알아보자

3. BILL_AMT1~6까지 6개 속성을 2개의 컴포넌트로 PCA 변환 후 개별 컴포넌트의 변동성 알아보기

```
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 ]
```

>> 2개의 PCA 컴포넌트만으로 6개 속성의 변동성의 약 95%를 설명할 수 있음

>> 첫번째 PCA cnrdmfh 90%나 수용가능

4. 원본 vs PCA 적용 데이터 세트 정확도 결과 비교

- RandomForestClassifier / cross_val_score() 로 3개의 교차 검증 세트로 반환

```python
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import cross_val_score

rcf = RandomForestClassifier(n_estimators=300, random_state=156)

#원본
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)))

#PCA 변환한 데이터 세트에서 예측
# 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 세트 평균 정확도:{0:.4f}'.format(np.mean(scores_pca)))
```

```
CV=3 인 경우의 개별 Fold세트별 정확도: [0.808  0.8212 0.8241]
평균 정확도:0.8178
CV=3 인 경우의 PCA 변환된 개별 Fold세트별 정확도: [0.7899 0.797  0.8025]
PCA 변환 데이터 세트 평균 정확도:0.7965
```

>> 전체 속성의 거의 1/4 수준인 6개의 PCA 컴포넌트만으로 1~2%만의 예측 성능 저하만 발생

> PCA는 차원 축소를 통해 데이터를 쉽게 인지하는데 활용할 수 있지만, Computer Vision 분야에서 더 활발하게 적용됨  ex) 얼굴 인식

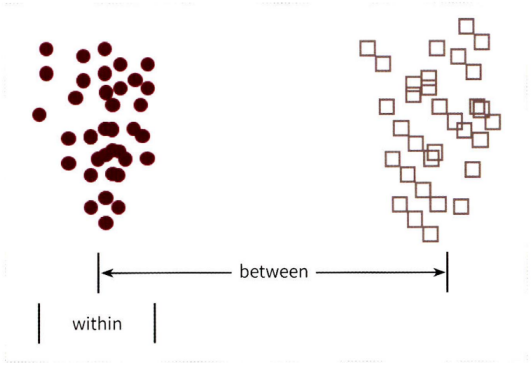
## ▼ 03. LDA

### LDA 개요

💡 선형 판별 분석법 (Linear Discriminant Analysis)

PCA와 유사하게 입력 데이터 세트를 저차원 공간에 투영해 차원을 축소하는 기법이지만, LDA는 지도학습의 분류에서 사용하기 쉽도록 입력 데이터의 결정 값 클래스를 최대한 분리할 수 있는 축을 찾는다

- 클래스 분리 최대화하는 축 찾기 : 클래스 간의 분산은 최대한 크게, 클래스 내부의 분산은 최대한 작게




- PCA vs. LDA : 공분산 행렬이 아닌 클래스 간 분산과 클래스 내부분산 행렬을 생성 후 고유벡터를 구해 입력 데이터를 투영

### LDA를 구하는 스텝

1. 클래스 내부와 클래스 간 분산 행렬을 구합니다. 이 두 개의 행렬은 입력 데이터의 결정 값 클래스별로 개별 피처의 평균 벡터(mean vector)를 기반으로 구합니다.
2. 클래스 내부 분산 행렬을 $S_W$, 클래스 간 분산 행렬을 $S_B$라고 하면 다음 식으로 두 행렬을 고유벡터로 분해할 수 있습니다.

$$S_W^T S_B = \begin{bmatrix} e_1 & \cdots & e_n \end{bmatrix} \begin{bmatrix} \lambda_1 & \cdots & 0 \\ \cdots & \cdots & \cdots \\ 0 & \cdots & \lambda_n \end{bmatrix} \begin{bmatrix} e_1^T \\ \cdots \\ e_n^T \end{bmatrix}$$

3. 고유값이 가장 큰 순으로 K개(LDA변환 차수만큼) 추출합니다.
4. 고유값이 가장 큰 순으로 추출된 고유벡터를 이용해 새롭게 입력 데이터를 변환합니다.

## 붓꽃 데이터 세트에 LDA 적용하기

- 사이킷런의 LinearDiscriminantAnalysis 클래스로 제공

1. 데이터 로드 후 표준 정규 분포로 스케일링

```
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)
```

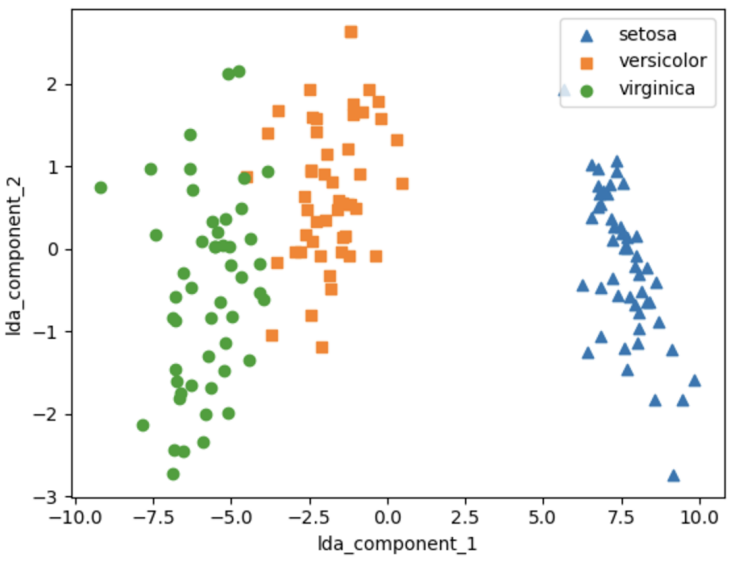
2. 2개의 컴포넌트로 붓꽃 데이터를 LDA 변환

- PCA와 다르게 지도학습 > 클래스의 결정값이 변환 시에 필요

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

```
(150, 2)
```

3. LDA 변환된 데이터 값을 품종별로 표현하기




# ▼ 04. SVD

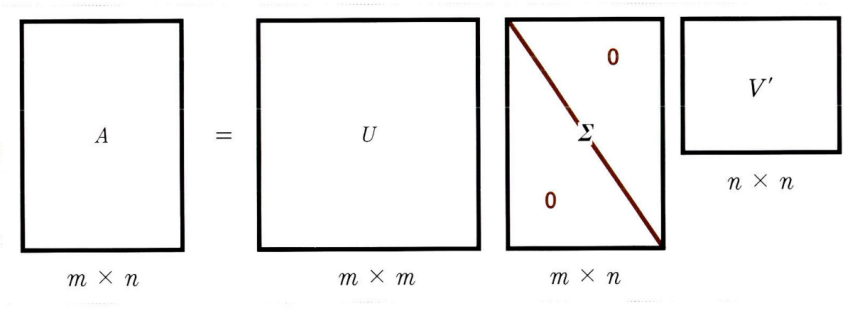
## SVD 개요

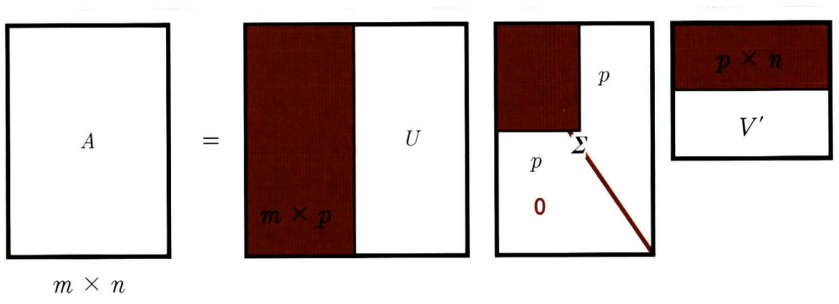
💡 정방행렬뿐만 아니라 행과 열의 크기가 다른 행렬에도 적용 가능

>> m*n 크기의 행렬 A를 다음과 같이 분해하는 것을 의미

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

- SVD는 특이값 분해
- 행렬 U와 V에 속한 벡터는 특이벡터 > 서로 직교한다 / Σ는 대각행렬




- 일반적으로 Σ의 비대각인 부분과 대각원소 중에 특이값이 0인 부분을 모두 제거하고 이에 대응되는 U와 V 원소도 함께 제거해 차원을 줄인 형태로 SVD 적용

## 넘파이에서 SVD 적용하기

1. 넘파이의 SVD 모듈 로딩 후 랜덤 4*4 행렬 생성 > 개별 로우끼리의 의존성 없애기

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

# 넘파이 랜덤 행렬 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]]
```

2. a 행렬에 SVD를 적용

- 이때 Σ 행렬의 경우 행렬의 대각에 위치한 값만(0이 아닌) 1차원 행렬로 표현

```
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]]
```

- U 행렬이 4*4, Vt행렬이 4*4, Sigma의 경우 1차원 행렬인 (4, )로 반환

check point 1) 원본으로 복원되는가? Yes

```
# Sigma를 다시 0을 포함한 대칭행렬로 변환 >> 원본으로 복원되는가? Yes!
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]]
```

check point 2) 로우 간 의존성이 있을 때 Sigma 값의 변화 >> 일부러 First+Second로 Third Row를 부여, Fourth is same as First one.

```
# 로우 간 의존성이 있을 때 Sigma 값의 변화
# First+Second로 Third Row를 부여, Fourth is same as First one.
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 ]]
```

>> 로우 간 관계가 매우 높아짐

3. 다시 SVD로 분해 후 전체가 아닌 0에 대응되는 데이터를 제외하고 복원하기

```
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.   ]
```

>> Σ 에서 0인 것이 두개 = 선형 독립인 로우 벡터의 개수가 2개

>> Σ 의 0에 대응되는 U, Sigma, Vt의 데이터 제외하고 복원 >> 선행 두 개의 행만 추출

```python
# Sigma에서 0인 것이 두개 = 선형 독립인 로우 벡터의 개수가 2개

# 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 ]]
```

## Truncated SVD 를 이용한 행렬 분해

- 사이파이에서만 지원 : Truncated SVD는 희소행렬로만 지원되기에 scipy.sparse.linalg를 이용
- 6*6 행렬을 Normal SVD로 분해해 차원과 Sigma 행렬 내의 특이값을 확인후 Truncated SVD로 분해해 차원, Sigma 행렬 내의 특이값, 복원된 데이터와 원본 데이터 비교해보기

```
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]]
```

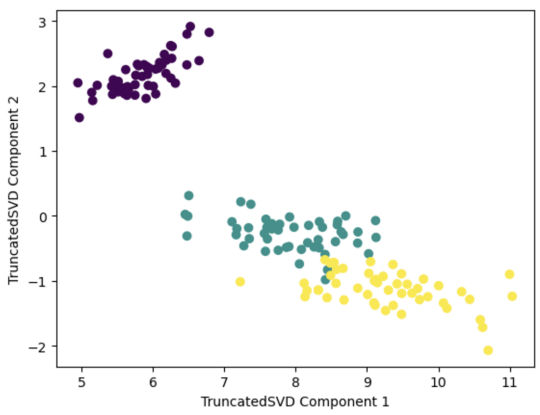
>> Truncated SVD로 복원하면 완벽하진 않지만 근사적으로 복원됨

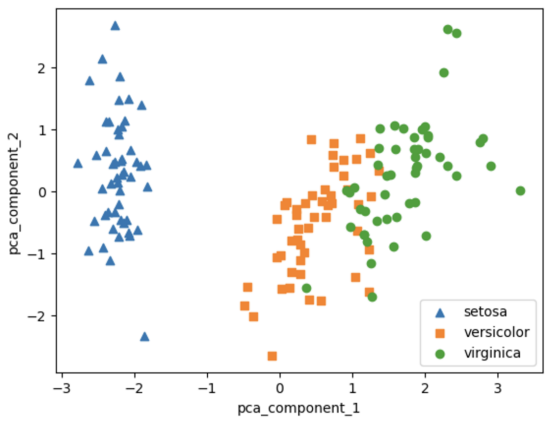
## 사이킷런 TruncatedSVD 클래스를 이용한 변환

- 사이킷런의 **TruncatedSVD** 클래스는 사이파이와 다르게 원본 행렬을 분해한 U, Sigma, Vt 행렬을 반환하진 않는다
- PCA와 유사하게 fit() transform()을 호출해 원본 데이터를 주요 컴포넌트로 차원을 축소해 변환
- 원본 데이터를 Truncated SVD 방식으로 분해된 U*Sigma 행렬에 선형 변환해 생성

1. PCA와 비교해보기

- Truncated SVD는 PCA와 유사하게 품종별로 클러스터링이 가능할 정도로 각 변환 속성으로 고유성이 뛰어남






2. 스케일링 변환 뒤 PCA와 비교




>> 거의 유사

- 두 개의 변환 행렬 값과 원복 속성별 컴포넌트 비율값을 비교해보면 ? 거의 같음

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

```
2.3124731296508826e-15
9.410874857174178e-17
```

3. 결론

- PCA가 SVD 알고리즘으로 구현되었음을 알 수 있다
- PCA는 밀집 행렬에 대한 변환만 가능 but SVD는 희소 행렬에 대한 변환도 가능

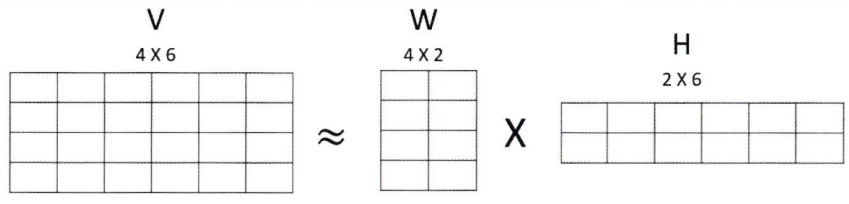
# ▼ 05. NMF

## NMF 개요

💡 Truncated SVD와 같이 낮은 랭크를 통한 행렬 근사 방식의 변형

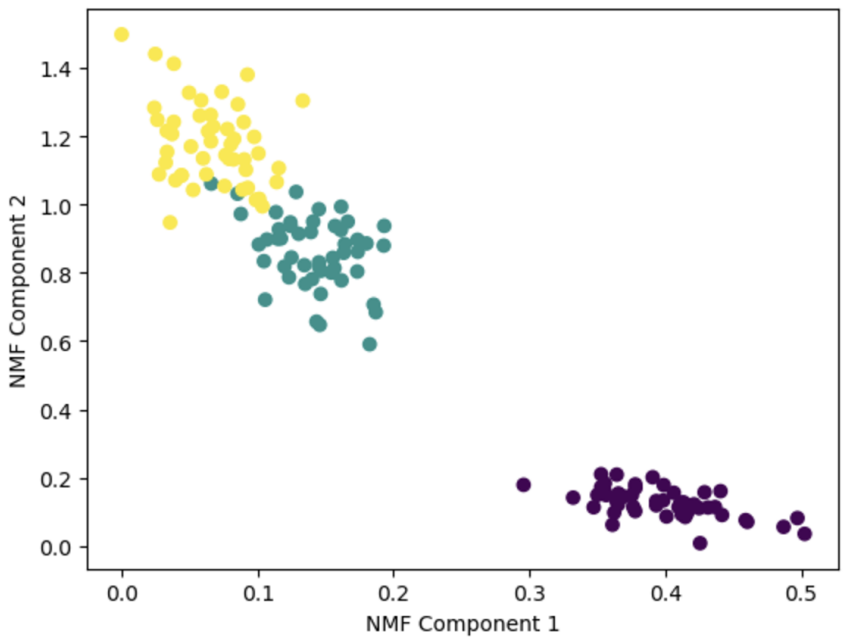
if 원본 행렬 내의 모든 원소 값이 양수

then 두 개 기반 양수 행렬로 분해될 수 있는 기법




>> SVD와 같은 행렬 분해 기법을 통칭

## 사이킷런의 NMF 클래스 이용




## 결론

- SVD와 유사하게 이미지 압축을 통한 패턴 인식, 텍스트의 토픽 모델링 기법, 문서 유사도 및 클러스터링에 잘 사용됨
- 추천 영역에서 적용 : 사용자-평가순위 데이터 세트를 행렬 분해 기법을 통해 분해