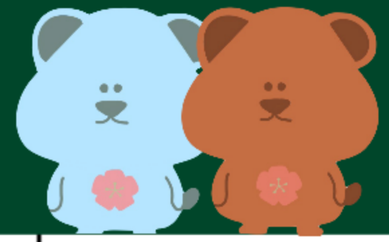


6장: 차원 축소

4팀 김두현, 노현선, 진규빈

EWHA EURON

목차

#01 차원 축소 개요

#02 PCA

#03 LDA

#04 SVD

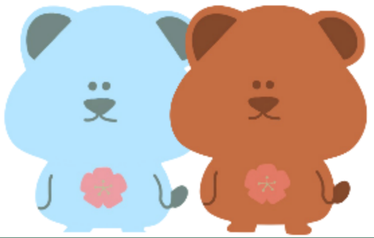
#05 NMF

#06 추가: 다른 차원축소 기법들

#07 추가 캐글 노트북: Dimensionality Reduction for Beginners

EWHA EURON

6.1 차원 축소 개요



EWHA EURON

#6.1 차원 축소 개요

#1 차원 축소
매우 많은 피처로 구성된 다차원 데이터 세트의 차원을 축소해 새로운 차원의 데이터 세트를 생성하는 것

일반적으로 *차원이 증가*할 때 문제점
(1) 데이터 포인트 간의 거리가 기하급수적으로 멀어지게 되고, 희소(sparse)한 구조를 가짐
(2) 적은 차원에서 학습된 모델보다 예측 신뢰도가 떨어짐
(3) 피처가 많을 경우 개별 피처 간에 상관관계가 높을 가능성이 큼. 선형 모델에서는 다중 공선성 문제로 모델의 예측 성능이 저하됨

→ 매우 많은 다차원의 피처를 차원 축소해 피처 수를 줄이면 더 직관적으로 데이터를 해석할 수 있음

#2 차원 축소 – 피처 선택(feature selection)
: 특정 피처에 종속성이 강한 불필요한 피처는 아예 제거하 고, 데이터의 특징을 잘 나타내는 주요 피처만 선택하는 것

#3 차원 축소 – 피처 추출(feature extraction)
: 기존 피처를 단순 압축이 아닌, 피처를 함축적으로 더 잘 설명할 수 있는 또 다른 공간으로 매핑해 저차원의 중요 피처로 압출해서 추출
새롭게 추출된 중요 특성은 기존의 피처가 압축된 것이므로 기존의 피처와는 완전히 다른 값이 됨

Ex) 학생을 평가하는 다양한 요소: 모의고사 성적, 종합 내신성적, 수능성적, 봉사활동, 대외활동, 학교 내외 수상경력 등
→ 학업 성취도, 커뮤니케이션 능력, 문세 해결력 등 더 함축적인 요약 특성으로 추출

#6.1 차원 축소 개요

#4 차원 축소 알고리즘

PCA, SVD, NMF는 차원 축소를 통해 좀 더 데이터를 잘 설명할 수 있는 잠재적인 요소를 추출하는 대표적인 차원 축소 알고리즘

- 이미지 데이터에서의 활용

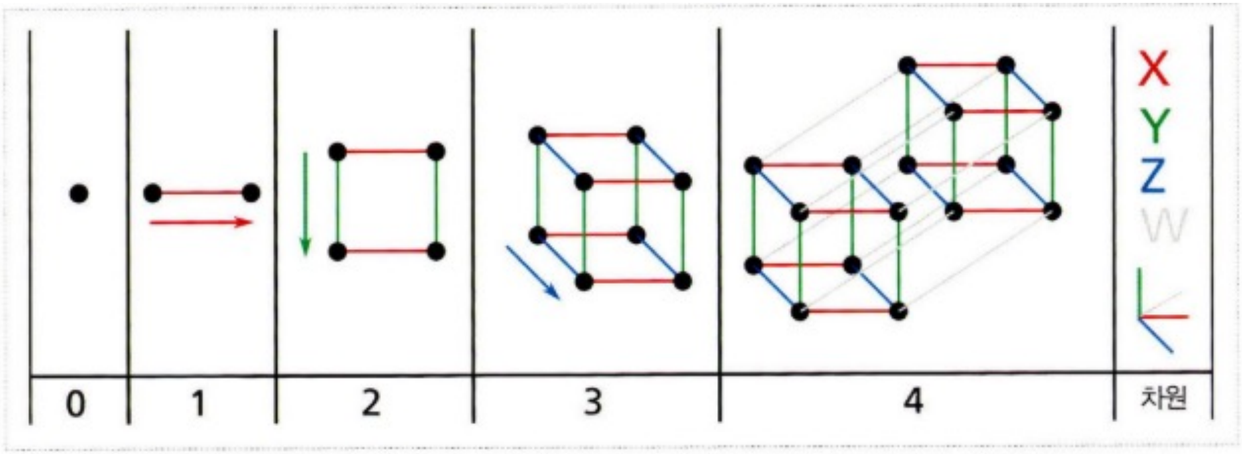
매우 많은 픽셀로 이뤄진 이미지 데이터에서 잠재된 특성을 피처로 도출해 함 축적 형태의 이미지 변환과 압축을 수행
과적합(overfitting) 영향력이 작아져서 원본 데이터로 예측하는 것보다 예측 성능을 더 끌어 올릴 수 있음

- 텍스트 문서에서의 활용

문서 내 단어들의 구성에서 숨겨져 있는 시맨틱(Semantic) 의미나 토픽(Topic)을 잠재 요 소로 간주하고 이를 찾아낼 수 있음
SVD와 NMF는 이러한 시맨틱 토픽 (Semantic Topic) 모델 링을 위한 기반 알고리즘으로 사용됨

#추가자료: 핸즈온 머신러닝 ch8_차원 축소

 # 차원의 저주
데이터의 차원(즉, 변수 수)이 늘어나면, 데이터 분석이나 학습이 점점 어려워지는 현상



점, 선, 정사각형, 정육면체, 테서랙트(0차원에서 4차원까지의 초입방체)

ㄴ 차원 직교좌표계에서 각각의 축에 평행하거나 직교하며 길이가 같은 모서리로만 이루어진 닫혀 있는 볼록한 도형

3차원 큐브에서 임의의 두 점을 선택하면 평균 거리는 0.66
1,000,000차원의 초입방체에서 두 점을 무작위로 선택하면 평균 거리는 428.35. 두 점이 너무 멀리 떨어져 있음

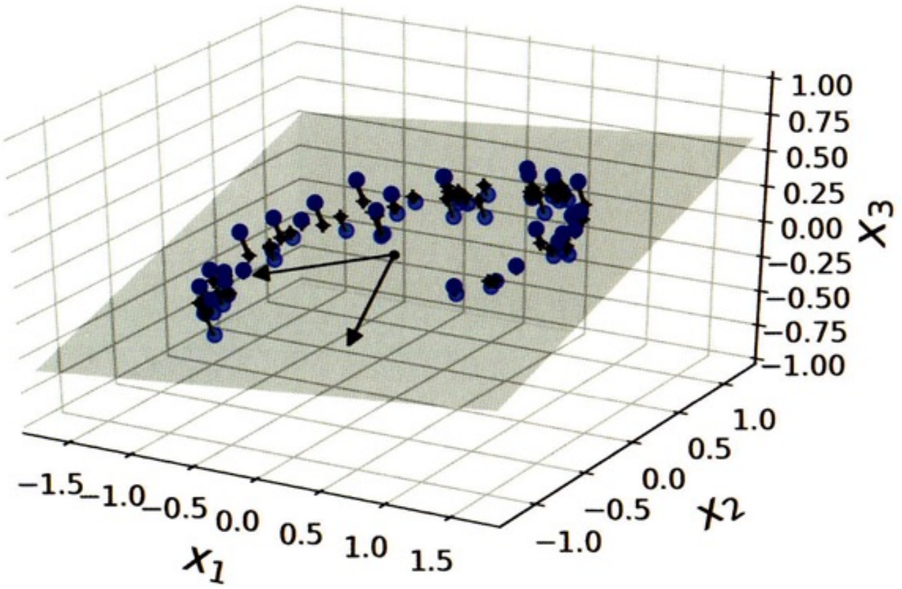
→ 고차원 데이터셋은 대부분의 훈련 데이터가 서로 멀리 떨어져 있음.
→ 새로운 샘플도 훈련 샘플과 멀리 떨어져 있을 가능성 높음

EWHA EURON

#추가자료: 핸즈온 머신러닝 ch8_차원 축소

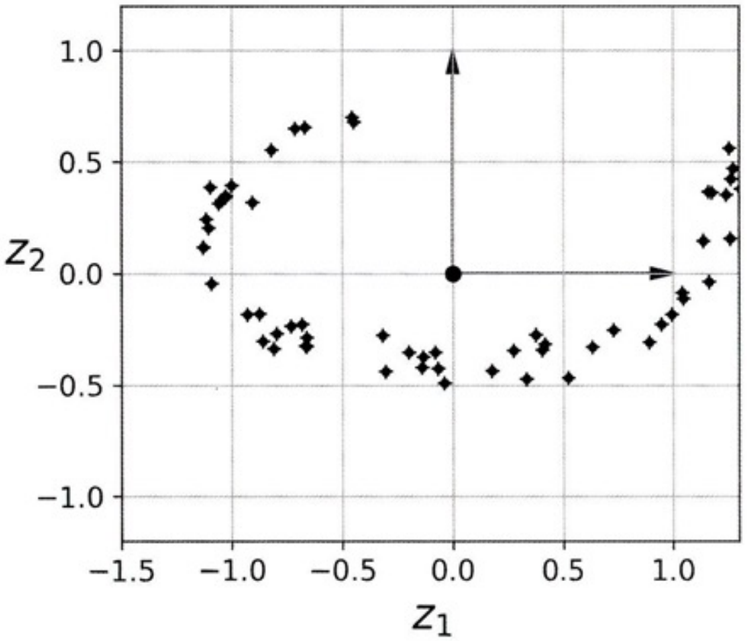
차원 축소를 위한 접근 방법

1. 투영



모든 훈련 샘플이 거의 평면 형태로 놓여 있음
→ 고차원(3D) 공간에 있는 저차원(2D)부분 공간

모든 훈련 샘플을 부분 공간에
수직으로 투영



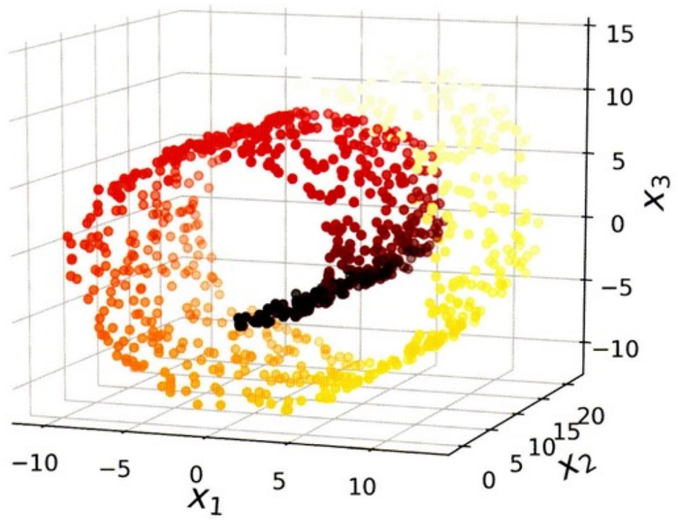
데이터셋의 차원을 3D에서 2D로 줄였음
각 축은 새로운 특성 z1, z2에 대응

#추가자료: 핸즈온 머신러닝 ch8_차원 축소

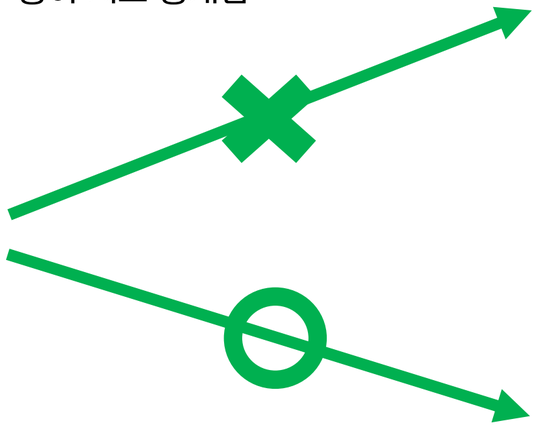
 # 차원 축소를 위한 접근 방법

1. 투영

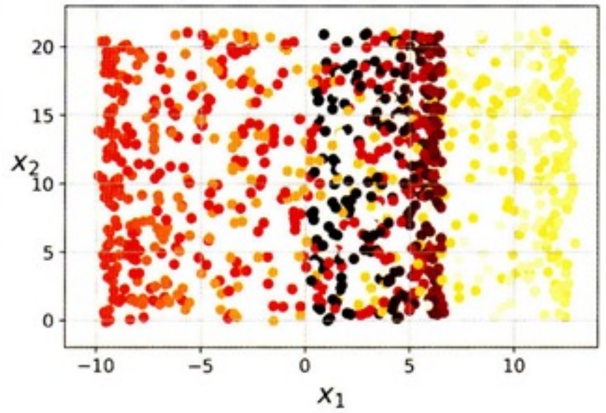
스위스롤 데이터셋의 경우
(부분 공간이 스위스 롤 모양으로 뒤틀려있는 데이터셋)

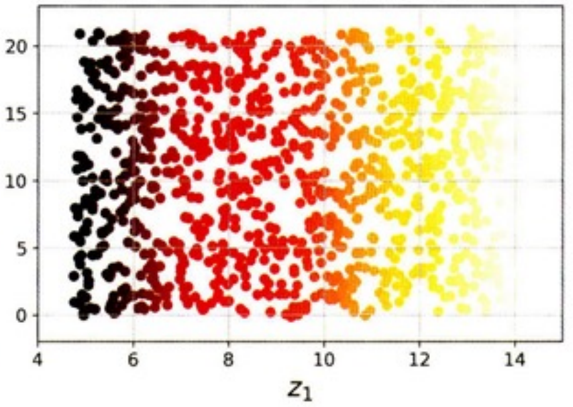


그냥 평면에 투영시키면 스위스 롤의 층이 서로 뭉개짐



스위스 롤을 펼쳐서 2D 데이터셋 얻어야 함





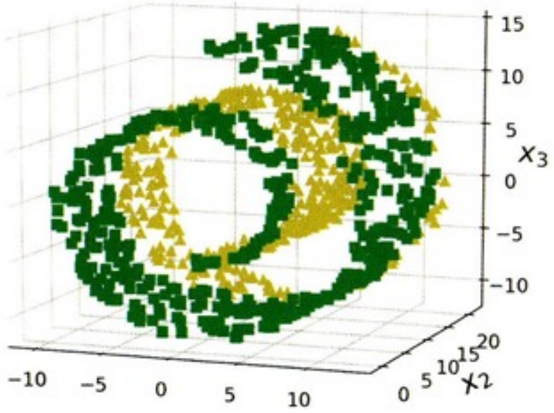
#추가자료: 핸즈온 머신러닝 ch8_차원 축소

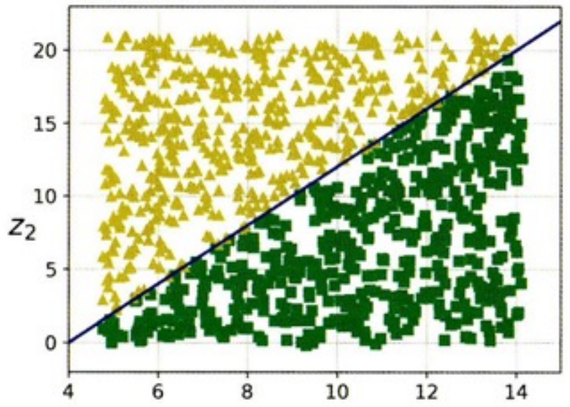
차원 축소를 위한 접근 방법

2. 매니폴드 학습

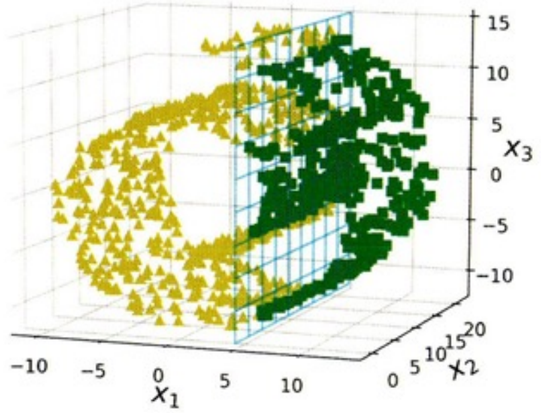
d차원 매니폴드: 국부적으로 d차원 초평면으로 보일 수 있는 n차원의 공간의 일부 (d<n)
Ex) 스위스 롤: d=2, n=3

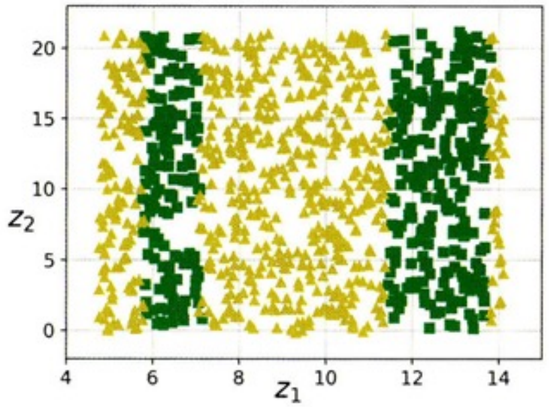
매니폴드 학습: 훈련 샘플이 놓여있는 매니폴드를 모델링
1. 대부분 실제 고차원 데이터셋이 더 낮은 저차원 매니폴드에 가깝게 놓여있다는 **매니폴드 가정**에 근거
2. 암묵적으로 분류나 회귀 등이 저차원의 매니폴드 공간에 표현되면 더 간단해질 것이란 가정에 근거 <- 항상 유효하진 않음





두번째 분류나 회귀 등이 저차원의 매니폴드 공간에 표현되면
더 간단해질 것이란 가정이 성립





펼쳐진 매니폴드에서 결정경계가 더 복잡해짐. 두번째 가정 성립x

EWHA EURON

6.2 PCA(Principal Component Analysis)

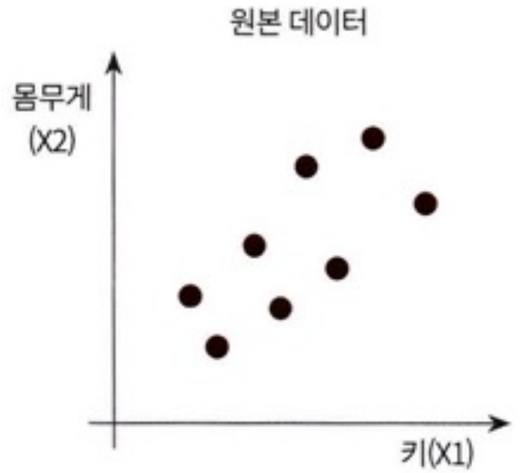
EWHA EURON

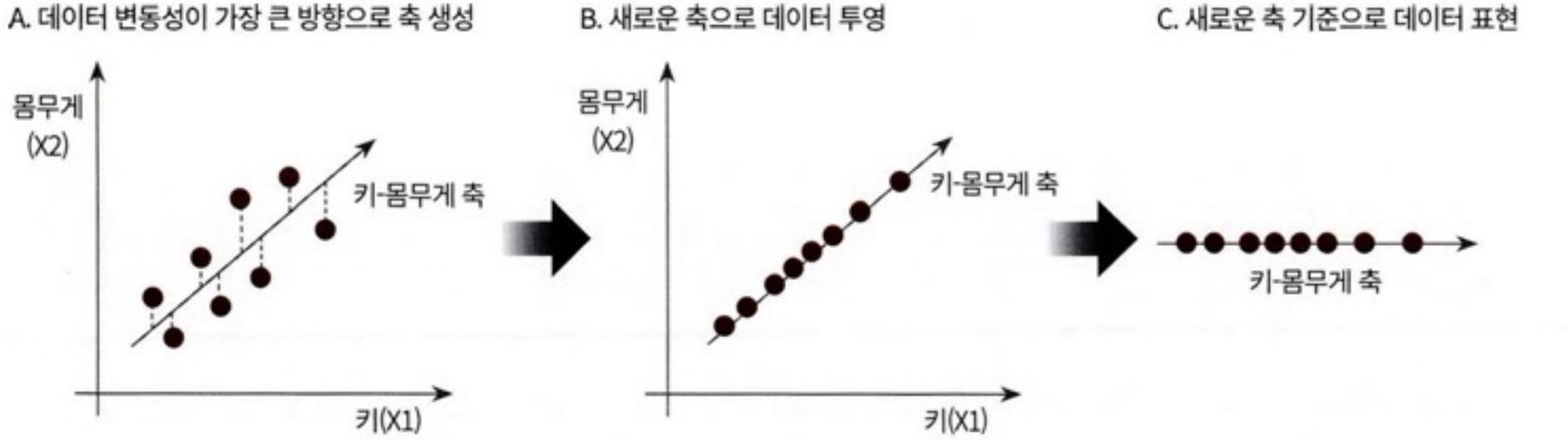
#6.2 PCA(Principal Component Analysis)

#1 PCA 개요

PCA: 여러 변수 간 에 존재하는 상관관계를 이용해 이를 대표하는 주성분(Principal Component)을 추출해 차원을 축소하는 기법

가장 높은 분산을 가지는 데이터의 축을 찾아 이 축으로 차원을 축소하는데, 이것이 PCA의 주성분이 됨



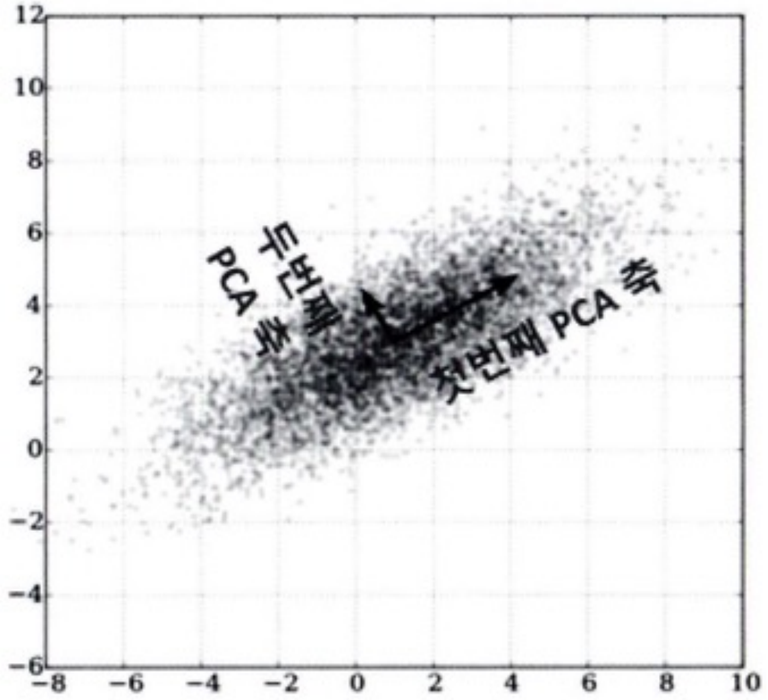


#6.2 PCA(Principal Component Analysis)

PCA 차원 축소 방법

1. 가장 큰 데이터 변동성(Variance)을 기반으로 첫 번째 벡터 축을 생성
2. 두 번째 축은 이 벡터 축에 직각이 되는 벡터(직교 벡터)를 축으로 함.
3. 세 번째 축은 다시 두 번째 축과 직각 이 되는 벡터를 설정하는 방식으로 축을 생성.
4. 이렇게 생성된 벡터 축에 원본 데이터를 투영하면 벡터 축의 개수만큼의 차원으로 원본 데이터가 차원 축소

→ 원본 데이터의 피처 개수에 비해 매우 작은 주성분으로 원본 데이터의 총 변동성을 대부분 설명할 수 있는 분석법



#6.2 PCA(Principal Component Analysis)

PCA 차원 축소 방법(선형대수 관점)

입력 데이터의 공분산 행렬(Covariance Matrix)을 고유값 분해하고, 이렇게 구한 고유벡터에 입력 데이터를 선형 변환하는 것

고유벡터: PCA의 주성분 벡터로서 입력 데이터의 분산이 큰 방향을 나타냄
고윳값(eigenvalue): 바로 이 고유벡터의 크기를 나타내며, 동시에 입력 데이터의 분산을 나타냄

<선형변환>
: 특정 벡터에 행렬 A를 곱해 새로운 벡터로 변환하는 것
특정 벡터를 하나의 공간에서 다른 공간으로 투영하는 개념

<고유벡터>
: 고유벡터는 행렬 A를 곱하더라도 방향이 변하지 않고 그 크기만 변하는 벡터를 지칭

Ax = ax(A는 행렬, x는 고유벡터, a는 스칼라값)
고유벡터는 여러 개가 존재하며, 정방 행렬은 최대 그 차원 수만큼의 고유벡터를 가질 수 있음

#6.2 PCA(Principal Component Analysis)

공분산 행렬

: 개별 분산값을 대각 원소로 하는 대칭행렬.

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

모든 변수 쌍 간의 공분산

각 변수의 분산

+ 정방행렬 : 열과 행이 같은 행렬

+ 대칭행렬 : 정방행렬 중에서 대각 원소를 중심으로 원소 값이 대칭되는 행렬
대칭행렬은 항상 고유벡터를 직교행렬(orthogonal matrix)로, 고유값을 정방 행렬로 대각화할 수 있음

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

입력 데이터의 공분산 행렬 = C 일 때,
P는 n X n의 직교행렬(모든 열 벡터가 서로 직교하고, 크기가 1인 단위 벡터로 구성된 정방행렬)
Σ는 n X n 정방행렬
P^T는 행렬 P의 전치 행렬

#6.2 PCA(Principal Component Analysis)

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

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

공분산 C = 고유벡터 직교 행렬 * 고유값 정방 행렬 * 고유벡터 직교 행렬의 전치 행렬

e_i = i번째 고유벡터
Λ_i = i번째 고유벡터의 크기
e1 = 가장 분산이 큰 방향을 가진 고유벡터
e2 = e1에 수직이면서 다음으로 가장 분산이 큰 방향을 가진 고유벡터

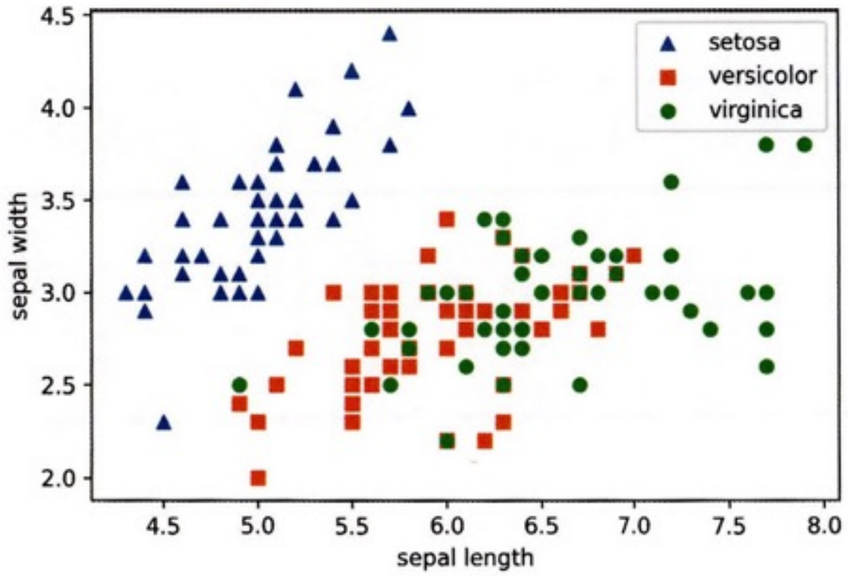
입력 데이터의 공분산 행렬이 고유벡터와 고유값으로 분해될 수 있으며,
이렇게 분해된 고유벡터를 이용해 입력 데이터를 선형 변환하는 방식이 PCA다

PCA의 스텝

1. 입력 데이터 세트의 공분산 행렬을 생성합니다.
2. 공분산 행렬의 고유벡터와 고유값을 계산합니다.
3. 고유값이 가장 큰 순으로 K개(PCA 변환 차수만큼)만큼 고유벡터를 추출합니다.
4. 고유값이 가장 큰 순으로 추출된 고유벡터를 이용해 새롭게 입력 데이터를 변환합니다.

#6.2 PCA(Principal Component Analysis)

붓꽃(iris) 데이터 세트
sepal length, sepal width, petal length, petal width의 4개의 속성을 2개의 PCA 차원으로 압축하기



Setosa : sepal width가 3.0보다 크고, sepal length가 6.0 이하인 곳에 일정하게 분포

Wrsicolor와 virginica: sepal width와 sepal length 조건만으로는 분류가 어려운 복잡한 조건임

#6.2 PCA(Principal Component Analysis)

1. 개별 속성을 함께 스케일링

PCA는 여러 속성의 값을 연산해야 하므로 여러 속성을 PCA로 압축 하기 전에 각 속성값을 동일한 스케일로 변환하는 것이 필요

```
from sklearn.preprocessing import StandardScaler

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

사이킷런의 StandardScaler를 이용해 평균이 0, 분산이 1 인 표준 정규 분포로 iris 데이터 세트의 속성값들을 변환

2. 스케일링이 적용된 데이터 세트에 PCA를 적용해 4차원(의 붓꽃 데이터를 2차원 PCA 데이터로 변환

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)

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

사이킷런은 PCA 변환을 위해 PCA 클래스를 제공
PCA 클래스는 생성 파라미터로 n_components(PCA로 변환 할 차원의 수)를 입력받음

EWHA EURON

#6.2 PCA(Principal Component Analysis)

1. 개별 속성을 함께 스케일링

PCA는 여러 속성의 값을 연산해야 하므로 여러 속성을 PCA로 압축 하기 전에 각 속성값을 동일한 스케일로 변환하는 것이 필요

```
from sklearn.preprocessing import StandardScaler

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

사이킷런의 StandardScaler를 이용해 평균이 0, 분산이 1 인 표준 정규 분포로 iris 데이터 세트의 속성값들을 변환

2. 스케일링이 적용된 데이터 세트에 PCA를 적용해 4차원(의 붓꽃 데이터를 2차원 PCA 데이터로 변환

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)

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

사이킷런은 PCA 변환을 위해 PCA 클래스를 제공
PCA 클래스는 생성 파라미터로 n_components(PCA로 변환 할 차원의 수)를 입력받음

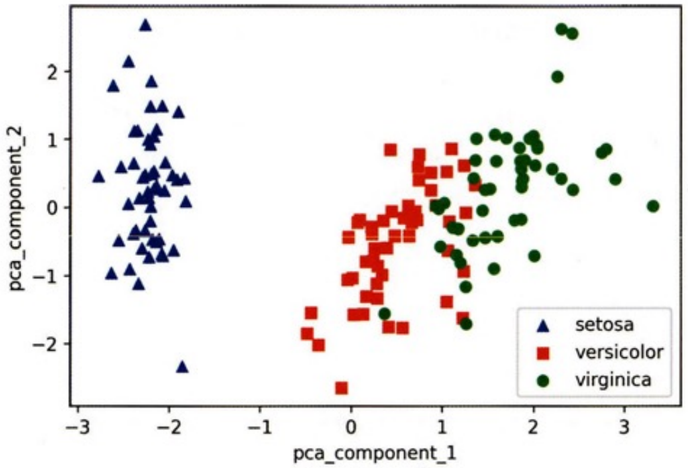
PCA 객체의 transform() 메서드를 호출해 원본 데이터 세트를 (150, 2)의 데이터 세트로 iris_pca 객체 변수로 반환

	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

	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

#6.2 PCA(Principal Component Analysis)

3. 2개의 속성으로 PCA 변환된 데이터 세트를 2차원상에서 시각화



- PCA로 변환한 후에도 pca_component_1축을 기반으로 Setosa 품종은 명확하게 구분이 가능.
- Versicolor와 Virginica는 pca_component_1 축을 기반으로 서로 겹치는 부분이 일부 존재하지만, 비교적 잘 구분됨.
- PCA의 첫 번째 새로운 축인 pca_component_1이 원본 데이터의 변동성을 잘 반영

4. PCA Component별로 원본 데이터의 변동성을 얼마나 반영하고 있는지 확인

PCA 변환을 수행한 PCA 객체의 explained_variance_ratio_ 속성은 전체 변동성에서 개별 PCA 컴포넌트별로 차지하는 변동성 비율을 제공

```
print(pca.explained_variance_ratio_)
```

```
[0.72962445 0.22850762]
```

첫 번째 PCA 변환 요소인 pca_component_1이 전체 변동성의 약 72.9%를 차지
두 번째 PCA 변환 요소인 pca_component_2가 약 22.8%를 차지.

→ PCA를 2개 요소로만 변환해도 원본 데이터 의 변동성을 95% 설명할 수 있음

#6.2 PCA(Principal Component Analysis)

5. 원본 붓꽃 데이터 세트와 PCA로 변환된 데이터 세트에 각각 분류를 적용한 후 결과를 비교

```
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 포레스트(Random Forest)를 적용>

```
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
```

<기존 4차원 데이터를 2차원으로 PCA 변환한 데이터 세트에 랜덤 포레스트를 적용>

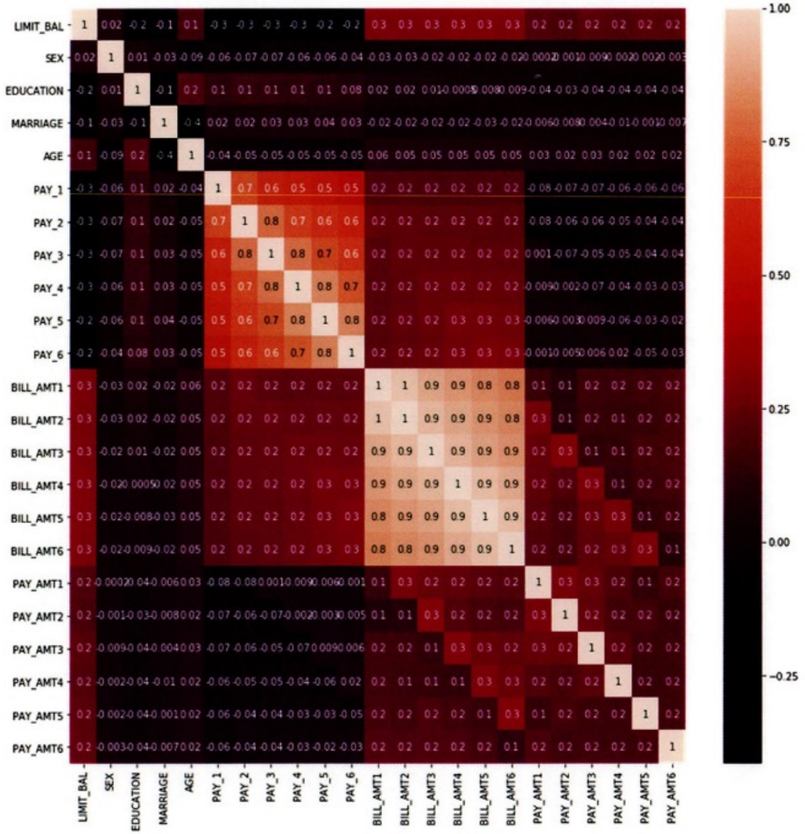
4개의 속성이 2개의 변환 속성으로 감소하면서 예측 성능의 정확도가 원본 데이터 대비 약 8% 하락
원본 데이터 세트 대비 예측 정확도는 PCA 변환 차원 개수에 따라 예측 성능이 떨어질 수밖에 없음

4개의 속성이 2개로, 속성 개수가 50% 감소한 것을 고려한다면 PCA 변환 후에도 원본 데이터의 특성을 상당 부분 유지하고 있음

#6.2 PCA(Principal Component Analysis)

UCI Machine Learning Repository에 있는 신용카드 고객 데이터 세트(Credit Card Clients Data Set)
: 신용카드 데이터 세트는 30,000개의 레코드와 24개의 속성을 가지고 있음
'default payment next month' 속성이 Target 값으로 '다음달 연체 여부' 를 의미하며 '연체' 일 경우 1, '정상납부' 가 0

1. DataFrame의 corr()를 이용해 각 속성 간의 상관도를 구한 뒤 이를 시본(Seaborn) 의 heatmap으로 시각화



BILL_AMT1 ~ BILL_AMT6 6개 속성끼리의 상관도가 대부분 0.9 이상으로 매우 높음을 알 수 있음

높은 상관도를 가진 속성들은 소수의 PCA만으로 이 속성들의 변동성을 수용할 수 있음

#6.2 PCA(Principal Component Analysis)

2. 6개 속성을 2개의 컴포넌트로 PCA 변환한 뒤 개별 컴포넌트의 변동성 확인

```
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 있음
특히 첫 번째 PCA 축으로 90%의 변동성을 수용할 정도로 이 6개 속성의 상관도가 매우 높음

#6.2 PCA(Principal Component Analysis)

3. 원본 데이터 세트와 6개의 컴포넌트로 PCA 변환한 데이터 세트의 분류 예측 결과를 상호 비교

<원본 데이터 세트의 분류 예측>

```
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: [0.8081 0.8197 0.8232]
평균 정확도:0.8170
```

<6개의 컴포넌트로 PCA 변환한 데이터 세트의 분류 예측>

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

# 원본 데이터 세트에 먼저 StandardScaler 적용
scaler = StandardScaler()
df_scaled = scaler.fit_transform(X_features)

# 6개의 컴포넌트를 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 Fold 세트별 정확도:', scores_pca)
print('PCA 변환 데이터 세트 평균 정확도:{0:.4f}'.format(np.mean(scores_pca)))
```

```
CV=3 인 경우의 PCA 변환된 개별 Fold세트별 정확도: [0.7917 0.7968 0.8035]
PCA 변환 데이터 셋 평균 정확도:0.7973
```

전체 23개 속성의 약 1/4 수준인 6개의 PCA 컴포넌트만으로도 원본 데이터를 기반으로 한 분류 예측 결과보다 약 1~2% 정도의 예측 성능 저하만 발생 → PCA의 뛰어난 압축 능력

#추가자료 핸즈온 머신러닝 ch8_차원 축소

 # 적절한 차원 수 선택하기

충분한 분산(예: 95%)이 될 때까지 더해야 할 차원수를 선택. 축소할 차원 수를 임의로 정하지 x

방법1. 충분한 분산을 유지하는 데 필요한 차원수 계산

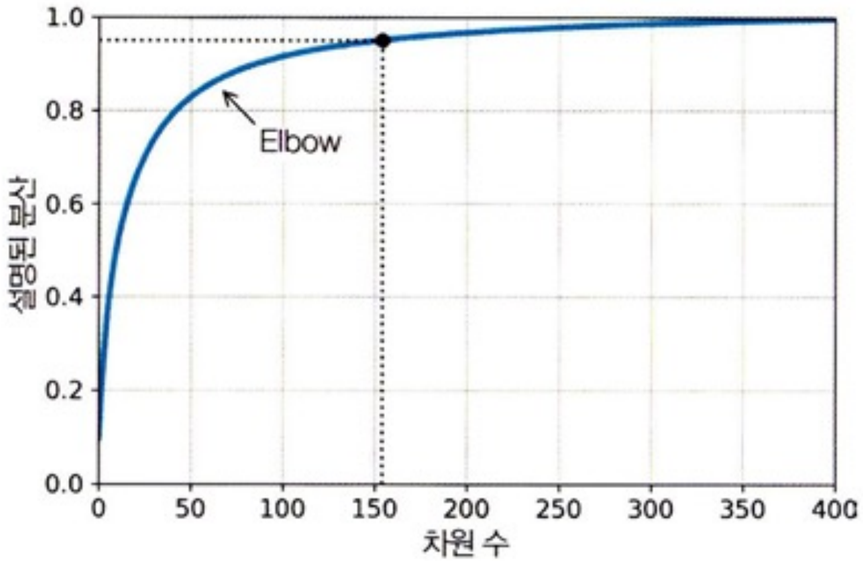
```
pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
```

차원을 축소하지 않고 PCA계산한 뒤
훈련 세트의 분산을 95 %로 유지하는 데
필요한 최소한의 차원 수 계산

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

보존하려는 분산의 비율을 n_components에
0.0에서 1.0 사이로 설정하여 PCA 다시 실행

방법2. 설명된 분산을 차원 수에 대한 함수로 그리기



설명된 분산의 빠른 성장이 멈추는 변곡점이 있음
그 지점에서 차원을 축소

6.3 LDA(Linear Discriminant Analysis)

EWHA EURON

#6.3 LDA(Linear Discriminant Analysis)

#1 LDA 개요

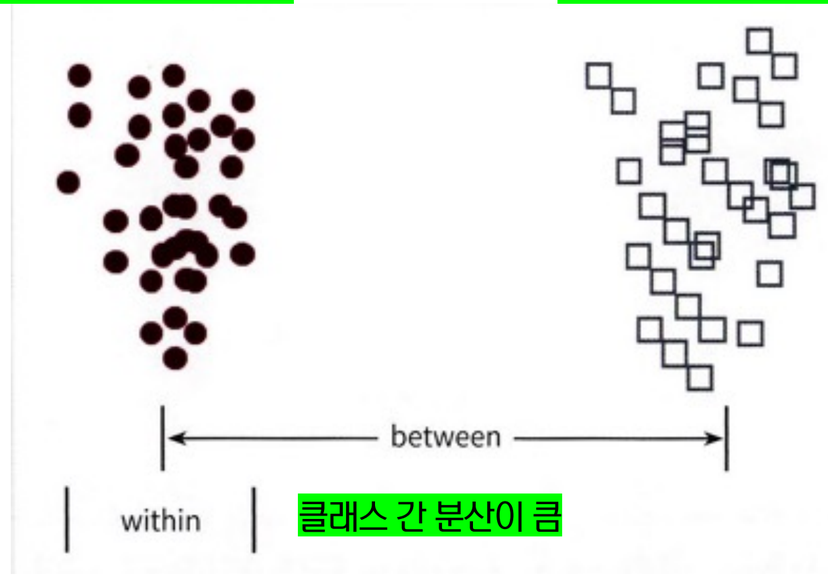
LDA : 선형 판별 분석법

PCA와 유사점: 입력 데이터 세트를 저차원 공간에 투영해 차원을 축소하는 기법임

PCA와 차이점: LDA는 지도학습의 분류(Classification)에서 사용하기 쉽도록 개별 클래스를 분별할 수 있는 기준을 최대한 유지하면서 차원을 축소.
PCA는 입력 데이터의 변동성의 가장 큰 축을 찾았지만, LDA는 입력 데이터의 결정 값 클래스를 최대한으로 분리할 수 있는 축을 찾음

클래스 내부 분산이 작음

클래스 내부 분산이 작음



클래스 간 분산이 큼

<좋은 클래스 분리>

LDA는 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 차원을 축소

#6.3 LDA(Linear Discriminant Analysis)

<LDA 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_B라고 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.

#6.3 LDA(Linear Discriminant Analysis)

#2 붓꽃 데이터 세트에 LDA 적용하기

사이킷런은 LDA를 LinearDiscriminantAnalysis 클래스로 제공

1. 붓꽃 데이터 세트를 로드하고 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

LDA는 PCA와 다르게 비지도학습이 아닌 *지도학습*임.
즉, 클래스의 결정 값이 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)
```

EWHA EURON

#6.3 LDA(Linear Discriminant Analysis)

#2 붓꽃 데이터 세트에 LDA 적용하기

사이킷런은 LDA를 LinearDiscriminantAnalysis 클래스로 제공

1. 붓꽃 데이터 세트를 로드하고 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

LDA는 PCA와 다르게 비지도학습이 아닌 *지도학습*임.
즉, 클래스의 결정 값이 변환 시에 필요

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

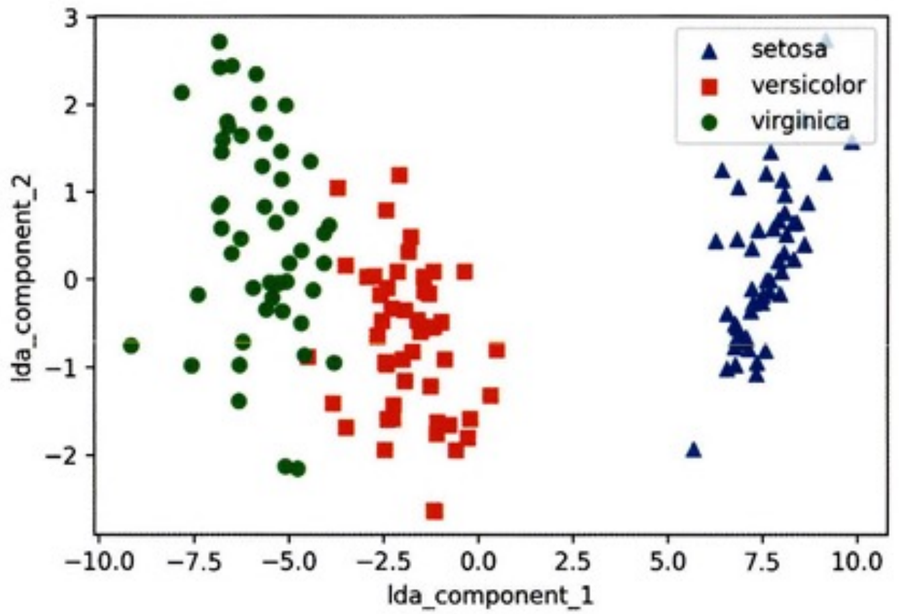
```
(150, 2)
```

EWHA EURON

#4.10 분류 실습 – 캐글 신용카드 사기 검출

3. LDA 변환된 입력 데이터 값을 2차원 평면에 품종별로 시각화

```
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 target별로 다른 모양으로 산점도로 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()
```



6.4 SVD

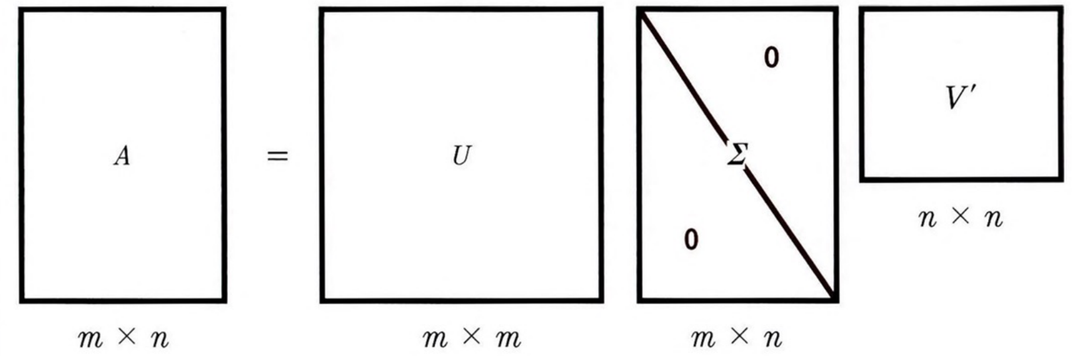
EWHA EURON

SVD 개요

SVD(Singular Value Decomposition)
- PCA와 유사한 행렬 분해 기법 이용
- 정방행렬만 고유벡터로 분해할 수 있는 PCA와 달리 SVD는 정방행렬뿐만 아니라 m*n 크기의 행렬 특이값 분해

특이벡터 (singular vector) => 서로 직교하는 성질 가짐

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



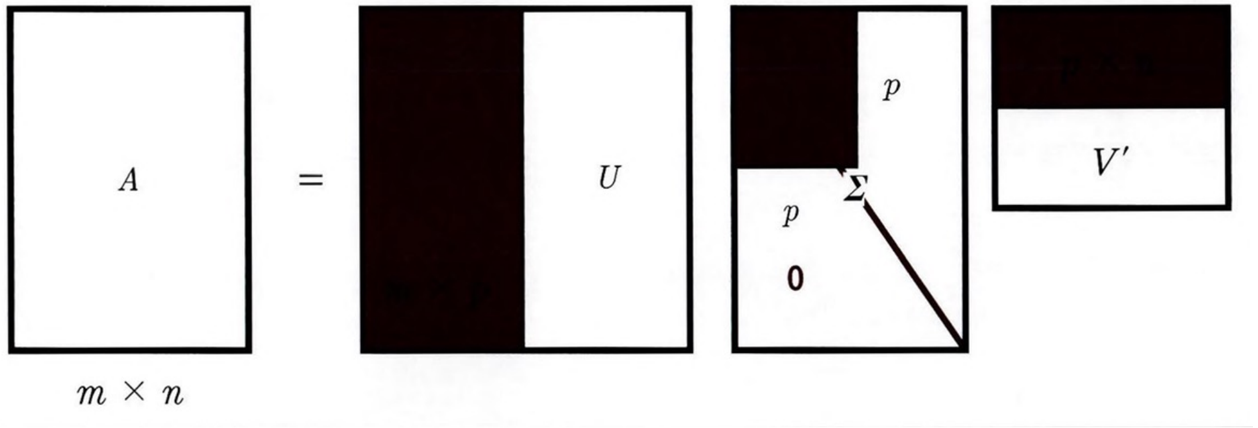
대각행렬 => 행렬의 대각에 위치한 값만 0 아니고 나머지 위치 값 모두 0

특이값

A의 차원이 m*n일 때
U의 차원이 m*m, Σ의 차원이 m*n,
V^T의 차원이 n*n으로 분해

SVD 개요

일반적으로 Σ의 비대각인 부분 & 대각원소 중 특이값 0인 부분 모두 제거하고
제거된 Σ에 대응되는 U와 V 원소도 함께 제거해 차원 줄인 형태로 SVD 적용



A의 차원이 m*n일 때
U의 차원이 m*p,
Σ의 차원이 p*p,
V^T의 차원이 p*n으로 분해

Truncated SVD
Σ의 대각원소 중 상위 몇 개만 추출해 대응하는 U와 V의 원소도 함께 제거해 더욱 차원 줄인 형태로 분해하는 것

SVD 예제

넘파이 SVD 모듈 numpy.linalg.svd 로딩
랜덤한 4*4 행렬 생성 (행렬의 개별 로우끼리의 의존성 없애기 위해 랜덤 생성)

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

# 4*4 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]]
```

생성된 a 행렬에 SVD 적용해
U, Sigma, Vt 도출

```
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]]
```

Sigma 행렬
- $A=U\Sigma V^T$에서 Σ 행렬 나타냄
- 0이 아닌 값의 경우만 1차원 행렬로 표현

EWHA EURON

SVD 예제

분해된 U, Sigma, Vt 내적하여 다시 원본 행렬로 복원

```
# 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]]
```

Sigma 행렬은 다시 0을 포함한 대칭행렬로 변환한 뒤 내적 수행해야 함 주의!

복원된 행렬 a_와 원본 행렬 a 동일

SVD 예제

데이터 세트 로우 간 의존성 있을 경우 Sigma 값 변화와 차원 축소 확인

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

a 행렬의 3번째 로우를 '첫 번째 로우 + 두 번째 로우'로 업데이트
4번째 로우는 첫 번째 로우와 같다고 업데이트

```
[[-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 ]]
```

a 행렬은 이전과 달리 로우 간 관계 매우 높아짐

데이터 SVD로 다시 분해

```
# 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.   ]
```

이전과 차원은 같지만 Sigma 값 중 2개가 0으로 변함
= 선형 독립인 로우 벡터의 개수가 2개
= 행렬의 Rank가 2

SVD 예제

분해된 U, Sigma, Vt 이용해 다시 원본 행렬로 복원

```
# U 행렬의 경우는 Sigma와 내적을 수행하므로 Sigma의 앞 2행에 대응되는 앞 2열만
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 ]]
```

Sigma의 0에 대응되는 U, Sigma, Vt의 데이터 제외하고 복원
- Sigma: 앞의 2개 요소만 0 아니므로 U 행렬 중 선형 두 개의 열만 추출
- Vt: 선행 두 개의 행만 추출

SVD 예제

Truncated SVD 이용해 행렬 분해

Σ 행렬에 있는 대각원소(특이값) 중 상위 일부 데이터만 추출해 분해하는 방식

- 인위적으로 더 작은 차원의 U. Σ, V^T로 분해하기 때문에 원본 행렬 정확하게 다시 원복 X
- BUT 데이터 정보가 압축되어 분해됨에도 불구하고 상당한 수준으로 원복 행렬 근사 가능
(원래 차원의 차수에 가깝게 잘라낼수록 원본 행렬에 더 가깝게 복원 가능)

- 넘파이 아닌 사이파이에서만 지원
- 희소행렬로만 지원돼서 scipy.linalg.svd 말고 scipy.sparse.linalg.svds 이용해야 함

SVD 예제

```
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.sha
print('₩nTruncated SVD Sigma값 행렬:', Sigma_tr)
matrix_tr = np.dot(np.dot(U_tr, np.diag(Sigma_tr)), Vt_tr)

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

1) 임의의 원본 행렬 6*6을 Normal SVD로 분해
2) 분해된 행렬의 차원과 Sigma 행렬 내의 특이값 확인
3) 다시 Truncated SVD로 분해
4) 분해된 행렬의 차원, Sigma 행렬 내의 특이값 확인
5) Truncated SVD로 분해된 행렬의 내적 계산해 다시 복원된 데이터와 원본 데이터 비교

```
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]]
```

완벽 X 근사적으로 복원됨

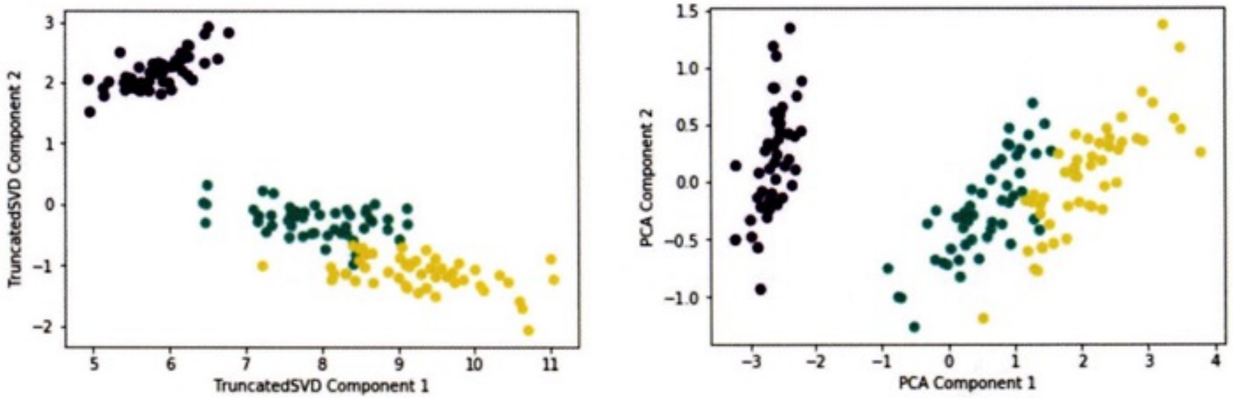
사이킷런 TruncatedSVD 클래스 이용한 변환

사이킷런 TruncatedSVD 클래스

⇒ PCA 클래스와 유사하게 fit()와 transform() 호출해 원본 데이터를 몇 개의 주요 컴포넌트로 차원 축소해 변환

⇒ 원본 데이터를 Trucated SVD 방식으로 분해된 U*Sigma 행렬에 선형 변환해 생성

```
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. 품종은 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')
```



(왼) TruncatedSVD로 변환된 붓꽃 데이터셋

(오) PCA로 변환된 붓꽃 데이터셋

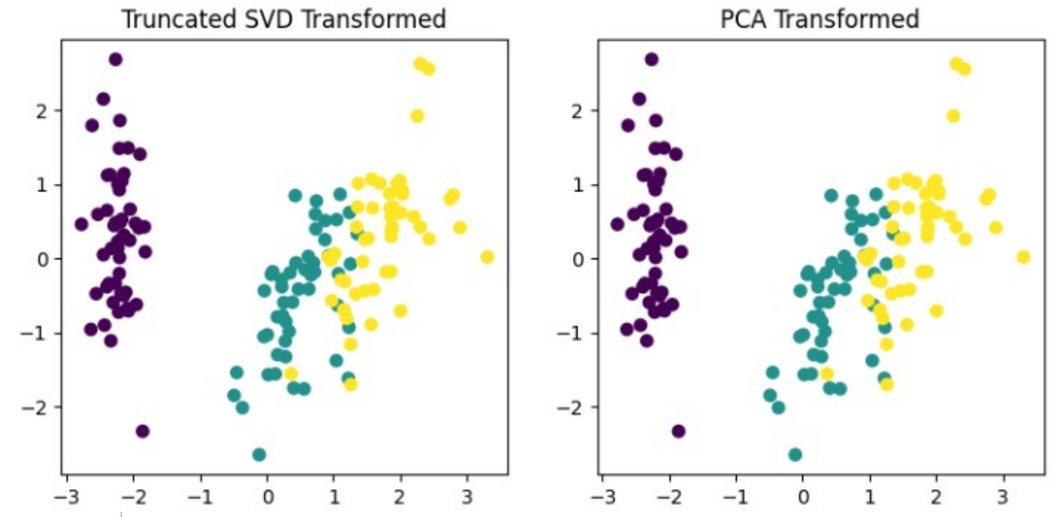
=> TruncatedSVD 변환 역시 PCA와 유사하게 변환 후에 품종별로 클러스터링 가능할 정도로 각 변환 속성으로 뛰어난 고유성 가지고 있음

사이킷런 TruncatedSVD 클래스 이용한 변환

사이킷런 TruncatedSVD와 PCA 클래스 모두 SVD 이용해 행렬 분해

```
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, PCA변환 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')
```

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



붓꽃 데이터 스케일링으로 변환 후
TruncatedSVD & PCA 클래스 변환
⇒ 두 개 거의 동일

사이킷런 TruncatedSVD 클래스 이용한 변환

두 개의 변환 행렬 값과 원본 속성별 컴포넌트 비율값 비교

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

```
2.3521693851928187e-15
-1.249000902703301e-16
```

모두 0에 가까운 값 => 두 개 변환 서로 동일

데이터셋 스케일링으로 데이터 중심 동일해지면 사이킷런의 SVD와 PCA는 동일한 변환 수행
⇒ PCA가 SVD 알고리즘으로 구현됐음 의미

- PCA는 밀집 행렬(Dense Matrix)에 대한 변환만 가능
- SVD는 희소 행렬(Sparse Matrix)에 대한 변환도 가능

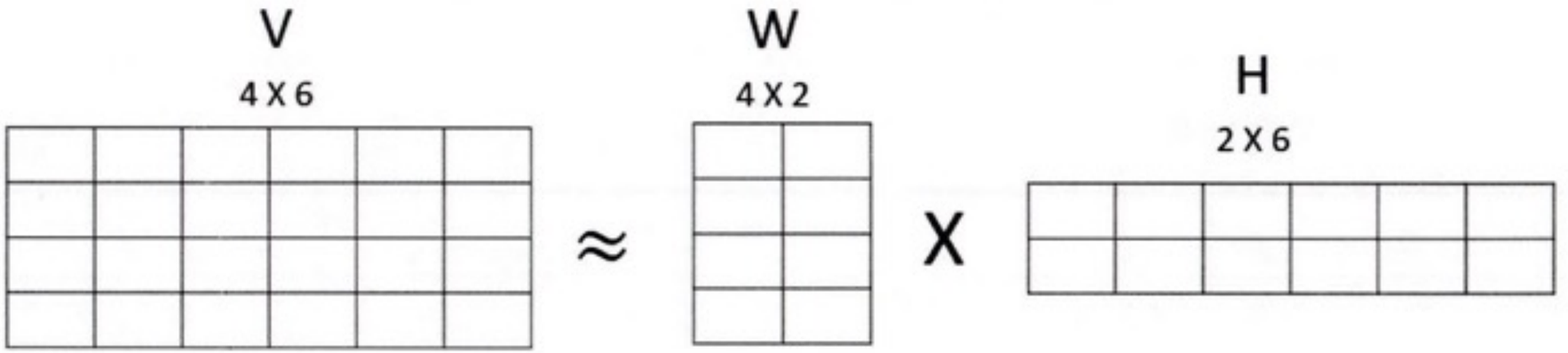
SVD는 PCA와 유사하게 컴퓨터 비전 영역에서
이미지 압축 통한 패턴 인식 & 신호 처리 분야에 사용되며,
텍스트의 토픽 모델링 기법인
LSA(Latent Semantic Analysis)의 기반 알고리즘임

6.5 NMF

EWHA EURON

NMF 개요

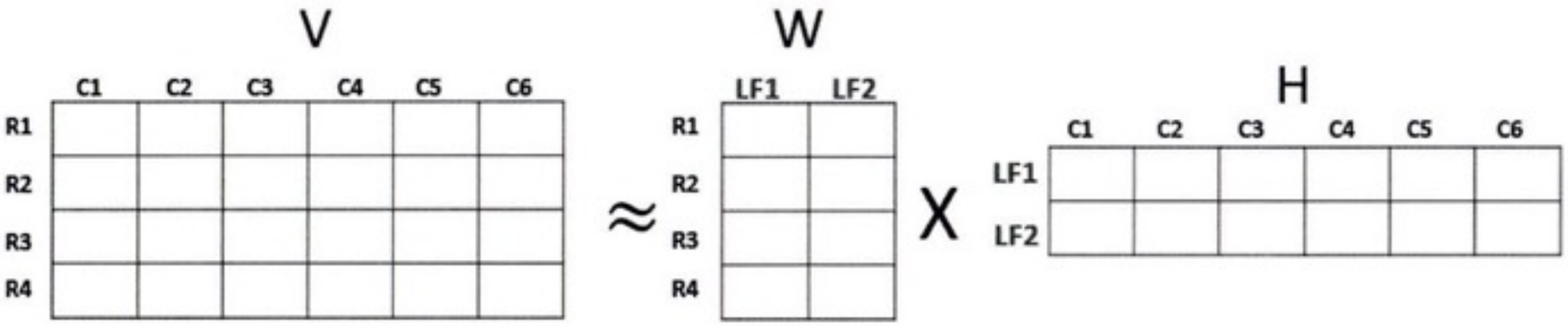
NMF (Non-Negative Matrix Factorization)
- Truncated SVD와 같이 낮은 랭크 통한 행렬 근사(Low-Rank Approximation) 방식의 변형
- 원본 행렬 내의 모든 원소 값이 모두 양수라는 게 보장되면 더 간단하게 두 개의 기반 양수 행렬로 분해될 수 있는 기법



4*6 원본 행렬 V는 4*2 행렬 W와 2*6 행렬 H로 근사해 분해될 수 있음

NMF 개요

행렬 분해 (Matrix Factorization)
- 일반적으로 SVD와 같은 행렬 분해 기법 통칭



길고 가는 행렬 W
(원본 행렬의 행 크기와
같고 열 크기보다 작은 행렬)

작고 넓은 행렬 H
(원본 행렬의 행 크기보다 작고
열 크기와 같은 행렬)

분해된 행렬은 잠재 요소를 특성으로 가짐

원본 행에 대해 이 잠재
요소의 값이 얼마나 되는지

이 잠재 요소가 원본
열(속성)로 어떻게 구성됐는지

NMF 개요 (추가 자료)

"하나의 객체정보를 음수를 포함하지 않은 두 개의 부분 정보로 인수분해하는 방법"
목적 - 공통 특성만을 갖고 정보 줄이는 것

<수식과 뉴스 기사에서 단어 기반으로 특성 추출하는 예시>

W 행렬: 행렬 R과 같은 인덱스의 행에 특성이 얼마나 적합한지 판단하는 가중치 나타내는 Latent representation 행렬

H 행렬: 행렬 R과 같은 열이 특성에 얼마나 중요한지 나타내는 Latent features 행렬

R 행렬: 데이터셋 / 행: 샘플 / 열: feature
p: 잠재적인 특성을 얼마나 찾을 것인지 직접 설정해줘야 하는 값 (토픽 모델링->토픽 개수, 차원 축소->축소할 차원 개수)

NMF 개요 (추가 자료)

행렬 W, H에 값이 채워지는 방식
- 행렬 W, H에 초기값 무작위로 설정하고 행렬 R와 W*H 간의 거리 최소화하는 방향으로 값 경신
- 거리 최소화하기 위해 사용하는 거리함수나 목적함수에 따라 값 달라짐

=> **증배 갱신 규칙 (Multiplicative Update Rules)**

1) 네 개의 갱신 행렬 생성
2) 행렬 W 갱신하기 위해 행렬 W의 모든 값을 식 (a) 내의 대응하는 값과 곱하고 식 (b) 내의 대응하는 값으로 나눔
3) 행렬 H를 갱신하기 위해 행렬 H 내의 모든 값을 식 (c) 내의 대응하는 값과 곱하고 식 (d) 내의 대응하는 값으로 나눔
4) 위 과정을 행렬 R과 행렬 W*H의 차이가 0이 될 때까지 반복

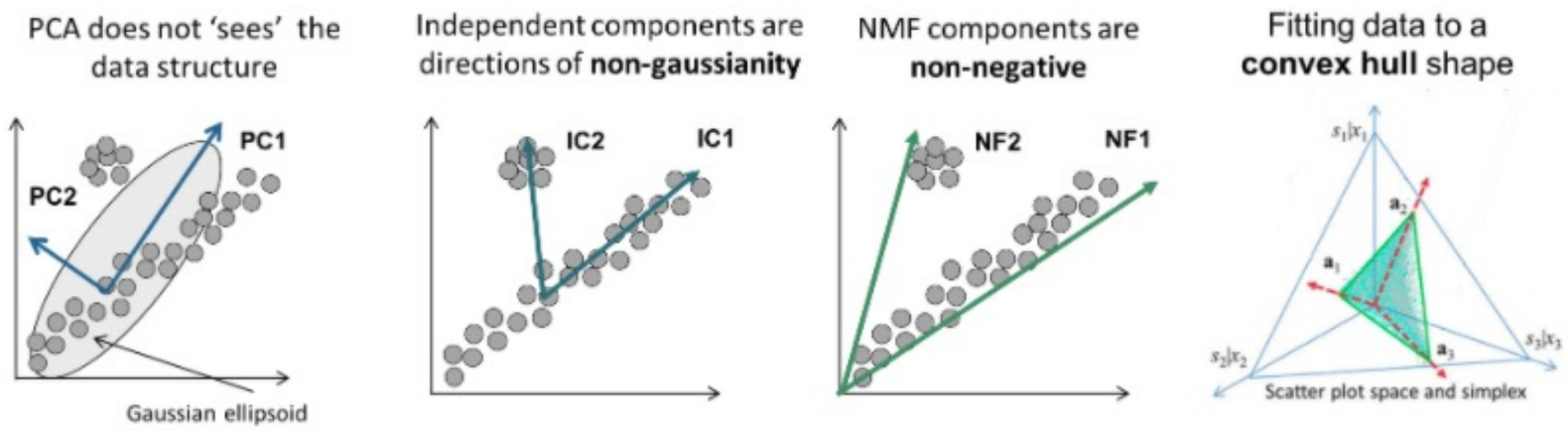
(a) $WN = R \times W^T$
(b) $WD = H \times W \times W^T$
(c) $HN = H^T \times R$
(d) $HD = H^T \times H \times W$

NMF 개요 (추가 자료)

PCA
- 고차원 데이터로부터 데이터 구조 밝히거나 데이터 차원 낮추는데 많이 이용되는 다변량 통계 분석 방법
- 공분산 행렬의 고유 벡터 이용해 분해하는 방법
- 항상 고유 벡터들 직교하기 때문에 데이터셋의 실제 데이터 구조 잘 반영하지 못함

NMF
- 대량의 정보 의미 특징과 의미 변수로 나누어 효율적으로 표현할 수 있는 방법

<PCA와 NMF의 기하학적 해석>

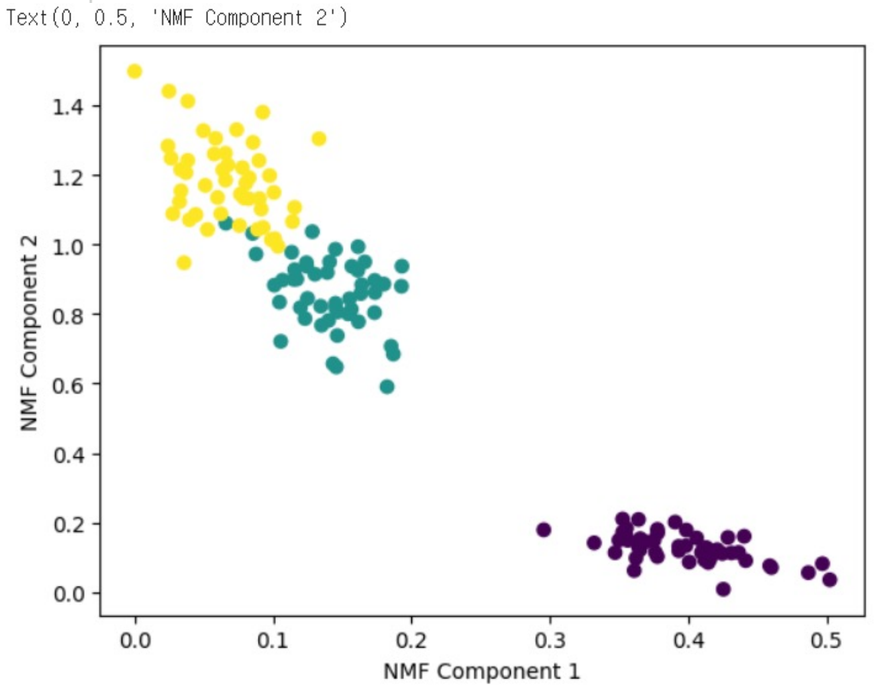


사이킷런 NMF 클래스

사이킷런에서 NMF는 NMF 클래스를 이용해 지원됨

```
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')
```

붓꽃 데이터 NMF 이용해 2개의 컴포넌트로 변환하고 시각화



사이킷런 NMF 클래스

NMF도 SVD와 유사하게

- 이미지 압축 통한 패턴 인식
- 텍스트의 토픽 모델링 기법
- 문사 유사도 및 클러스터링
- 추천 영역 → 잠재 요소(Latent Factoring) 기반의 추천 방식

등에 잘 사용됨

사용자의 상품(ex. 영화) 평가 데이터셋인
사용자-평가 순위(user-Rating) 데이터셋을 행렬 분해 기법 통해 분해
사용자가 평가하지 않은 상품에 대한 잠재적인 요소를 추출
이를 통해 평가 순위(Rating) 예측하고 높은 순위로 예측된 상품 추천

추가 자료: 다른 축소 차원 기법

EWHA EURON

추가 자료: 다른 축소 차원 기법

사이킷런이 제공하는 다양한 축소 차원 기법 중 활용도가 높은 기법:

1. 랜덤 투영 (Random Projection)
2. 다차원 스케일링 (MDS, Multidimensional Scaling)
3. IsoMap
4. t-SNE
5. 선형 판별 분석 (LDA, Linear Discriminant Analysis)

랜덤 투영

개념: 랜덤한 선형 투영을 사용해 고차원 데이터를 저차원으로 변환
핵심 아이디어: 무작위 행렬로 변환해도 거리 정보가 잘 보존됨 (존슨-린덴스트라우스 정리)
특징:
- 계산 빠름, 단순함
- 데이터 수나 차원 수에 의존도 낮음

활용 패키지: sklearn.random_projection

추가 자료: 다른 축소 차원 기법

다차원 스케일링 (MDS, Multidimensional Scaling)

개념: 샘플 간의 거리(distance) 를 최대한 보존하면서 차원

특징:

- 거리 기반 시각화에 적합
- 계산 복잡도 높음: 대규모 데이터에는 비효율적

Isomap

개념: 각 샘플을 가까운 이웃과 연결해 그래프를 만들고, 그 그래프상의 지오데식 거리(geodesic distance) 를 유지하면서 차원 축소

특징:

- 데이터의 비선형 구조(매니폴드) 를 보존
- 전형적인 MDS보다 곡면형 데이터(예: 스위스 롤)에 강함

추가 자료: 다른 축소 차원 기법

t-SNE (t-distributed Stochastic Neighbor Embedding)

개념: 비슷한 샘플은 가깝게, 다른 샘플은 멀리 배치되도록 확률적 방식으로 임베딩
특징:

- 비선형 구조 시각화에 매우 강력
- 고차원 데이터의 군집 구조 시각화에 자주 사용 (예: MNIST)
- 계산량 많고, 하이퍼파라미터에 민감

선형 판별 분석 (LDA, Linear Discriminant Analysis)

개념: 클래스 간 분산을 최대화하고 클래스 내 분산을 최소화하는 축을 학습
특징:

- 실제는 분류 알고리즘이지만 차원 축소에도 사용
- 클래스 간 분리도를 높이는 투영면 생성
- 다른 분류기(SVM 등) 적용 전 전처리용으로 적합

[Kaggle] Dimensionality Reduction for Beginners

EWHA EURON

캐글 노트북: 차원 축소

Dimensionality Reduction for Beginners: 유방암 환자들의 검사 결과에 관한 데이터를 활용해 차원 축소 기법에 대해 다룸
- https://www.kaggle.com/code/lazrus/dimensionality-reduction-demystified-for-beginners

*활용되는 차원 축소 기법
1. PCA (주성분 분석)
2. MDS (다차원 척도법)
3. t-SNE

[1단계] 준비: 라이브러리 및 데이터 로드
- pandas (pd): 데이터 표(DataFrame) 처리
- matplotlib (plt), seaborn (sns): 데이터 시각화
- sklearn (Scikit-learn): 머신러닝 모델 및 전처리 도구 (PCA, MDS, t-SNE 등)

- pd.read_csv(...): 유방암 데이터셋(CSV 파일)을 df 변수에 로드

캐글 노트북: 차원 축소

[2단계] 데이터 탐색 및 전처리

1. 데이터 분리 (피처/타겟)

y = df['diagnosis']: 타겟(Target) 변수 y에 'diagnosis'(악성/양성) 컬럼 저장

X = df.drop(...): 특징(Feature) 변수 X에 y를 제외한 30+개의 검사 수치 저장

2. 라벨 인코딩 (타겟 변환)

from sklearn.preprocessing import LabelEncoder

y = le.fit_transform(y)

목적: 컴퓨터가 이해하도록 문자(M/B)를 숫자(1/0)로 변환 – M(악성)은 1, B(양성)은 0

3. 스탠다드 스케일링 (피처 변환)

from sklearn.preprocessing import StandardScaler

X = scaler.fit_transform(X)

목적: 각 특징(차원)의 스케일(단위)을 통일 (e.g., '면적' vs '질감')

결과: 모든 특징이 평균 0, 표준편차 1을 갖게 되어 공평한 분석 가능

차원 축소: PCA

[3단계] 차원 축소 적용 및 시각화

기법 1: PCA (주성분 분석)

분석 목적: 유방암 데이터의 30개 Feature를 2개의 '주성분'으로 축소함.

이유: 30차원은 시각화가 불가능하므로, 데이터의 분포를 파악하기 쉬운 2차원으로 압축.

핵심 전처리: StandardScaler (표준화)

PCA 적용 전, StandardScaler를 사용해 모든 특징의 스케일을 통일해야 함.

이유: PCA는 분산(데이터가 퍼진 정도)을 기반으로 작동한다. '면적'처럼 단위가 큰 특징이 '질감'처럼 단위가 작은 특징보다 분석에 과도하게 큰 영향을 미치는 것을 방지하기 위함.

```
# Before applying PCA, each feature should be centered (zero mean)
# and with unit variance
# This can be done by using StandardScaler of sklearn Library

# 1. 스케일러로 데이터 표준화
canc_norm = StandardScaler().fit(X_canc).transform(X_canc)
```

차원 축소: PCA

[3단계] 차원 축소 적용 및 시각화

기법 1: PCA (주성분 분석)

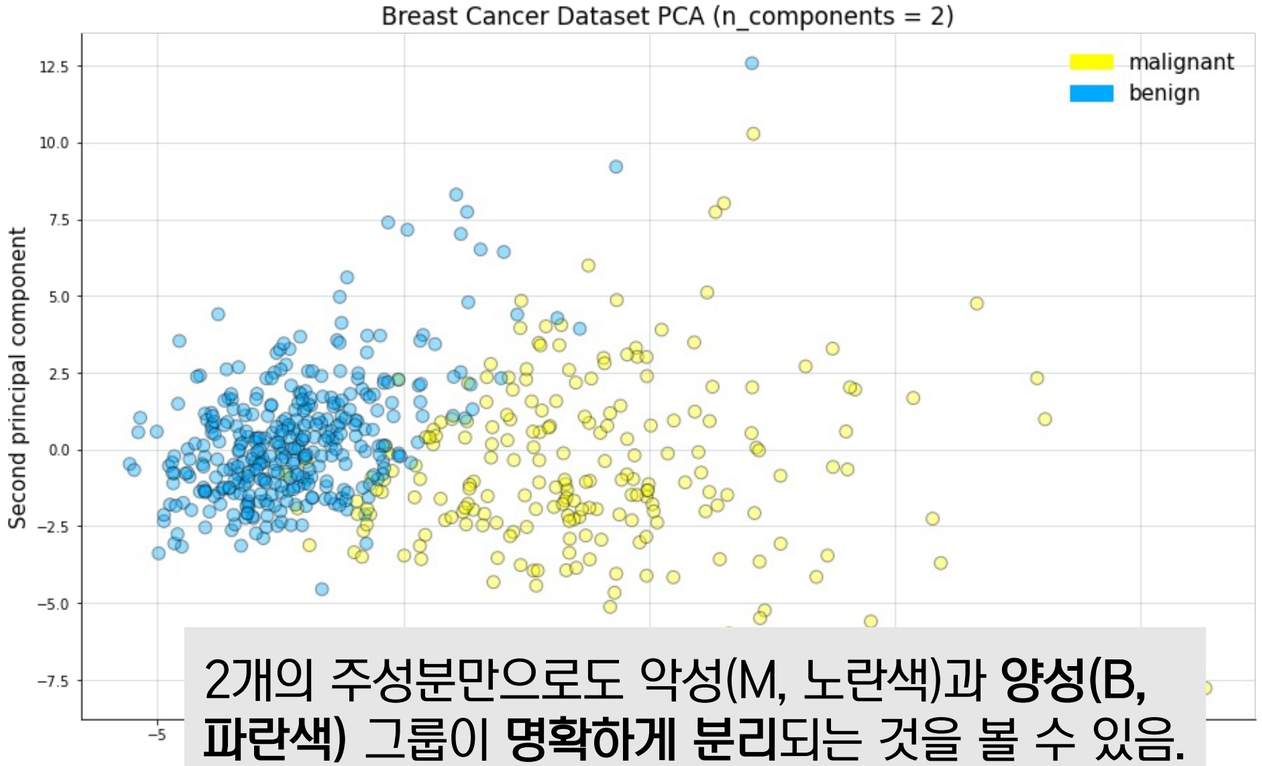
[실행 및 시각화]

```
# 1. 2개의 주성분을 찾도록 PCA 설정 및 실행
pca = PCA(n_components = 2)

# 2. 스케일링된 데이터로 PCA 학습(fit) 및 변환(transform) 동시 실행
canc_pca = pca.fit_transform(canc_norm)
```

[2D 시각화 결과]

X축: First principal component (첫 번째 주성분)

Y축: Second principal component (두 번째 주성분)



2개의 주성분만으로도 악성(M, 노란색)과 **양성(B, 파란색)** 그룹이 **명확하게 분리**되는 것을 볼 수 있음.

차원 축소: PCA

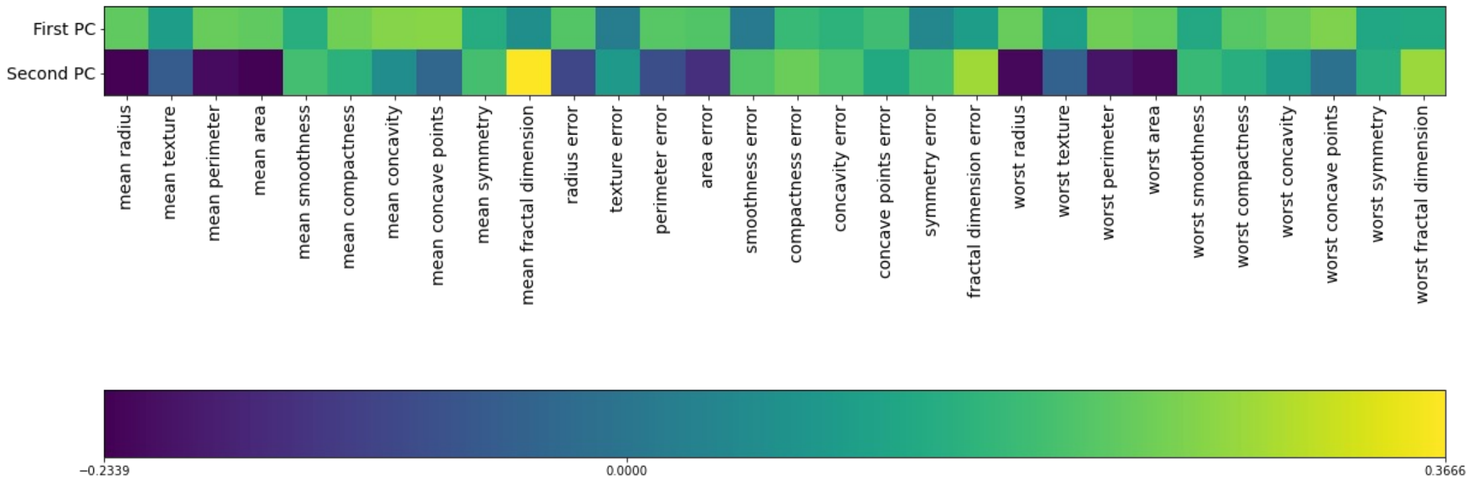
[3단계] 차원 축소 적용 및 시각화

PCA 심층 분석 Heatmap

목적: PCA가 찾아낸 첫 번째 주성분과 두 번째 주성분이 의미하는 바를 찾음.

PCA의 '.components_' 속성을 이용: 2개의 주성분이 30개의 원본 특징과 각각 어떤 관계를 갖는지 확인하기 위함.

Y축: PC1, PC2
X축: 30개의 원본 특징 (mean radius, mean texture...)
색상: 노란색(강한 양의 관계), 보라색(강한 음의 관계)



PC 1.
- 대부분의 특징이 양수(녹색/노란색) 값: 30개 특징 대부분이 PC1과 양의 상관관계

PC 2.
- 양수와 음수 섞임: 특징들 간 상반된 관계

차원 축소: PCA

[3단계] 차원 축소 적용 및 시각화

PCA의 결론 및 한계:

장점

- 데이터 탐색을 위한 **좋은 초기 도구.**
- 작동이 빠르고 대부분의 데이터셋에서 잘 작동함.
- 30차원 -> 2차원 축소를 통해 시각화 및 해석이 쉬워짐.

한계

- 더 복잡한 데이터셋의 미묘한 그룹(groupings)이나 비선형 구조를 찾는 데는 한계가 있을 수 있음.

차원 축소: MDS

[3단계] 차원 축소 적용 및 시각화

기법 2: MDS (Multi-Dimensional Scaling)

분석 목적: 유방암 데이터의 30개 Feature를 2개의 'MDS 차원'으로 축소.

이유: 고차원 데이터 내에 숨어있는 저차원 구조(Manifold)를 찾아 2차원으로 시각화.

핵심 원리: 거리 보존

PCA가 **분산**을 보존하는 것과 달리, MDS는 원본 30차원에서의 샘플 간 '거리'를 2차원에서도 최대한 보존하려 함.

```python
from sklearn.manifold import MDS

# 1. 2개의 차원으로 줄이도록 MDS 설정
# (random_state는 결과를 고정하기 위함)
mds = MDS(n_components = 2, random_state = 2)

# 2. 스케일링된 데이터로 MDS 변환 실행
canc_mds = mds.fit_transform(canc_norm)

# 3. 결과 확인
# Number of Features in Breat Cancer DataSet Before MDS : 30
# Number of Features in Breat Cancer DataSet After MDS : 2
```

EWHA EURON

차원 축소: MDS

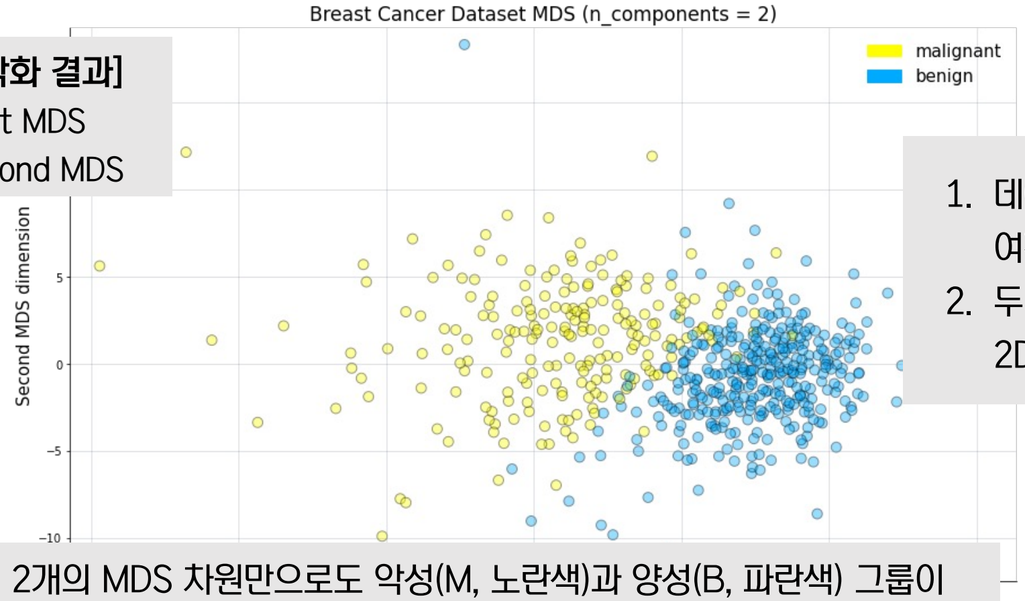
[3단계] 차원 축소 적용 및 시각화

MDS 2D 시각화 Scatter Plot 분석

30개의 특징(차원)으로 이루어진 데이터를 MDS 알고리즘을 사용해 2D 평면에 그린 그림

PCA와 달리, 이 차트의 목표는 '**샘플 간의 거리**'를 보존하는 것.

[2D 시각화 결과]
X축: First MDS
Y축: Second MDS



2개의 MDS 차원만으로도 악성(M, 노란색)과 양성(B, 파란색) 그룹이 잘 분리되는 것을 볼 수 있음.

1. 데이터의 분산(PCA)이 아닌 '거리' 라는 다른 기준으로 2D 지도를 그려도, 여전히 악성 그룹과 양성 그룹은 잘 분리됨.
2. 두 그룹이 30차원 공간에서 실제로 멀리 떨어져 있으며, MDS가 그 관계를 2D 평면에 성공적으로 시각화했음을 의미함.

차원 축소: t-SNE

[3단계] 차원 축소 적용 및 시각화

기법 3: t-SNE

분석 목적: 유방암 데이터의 30개 Feature를 2개의 't-SNE 차원'으로 축소.

이유: 군집(클러스터)을 찾는 데 최적화된 기법.

핵심 원리: 이웃 보존

고차원(30D)에서 '가까운 이웃'이었던 샘플들을 2차원(2D)에서도 '가까운 이웃'이 되도록 뭉치게 만듦.

MDS(전체적 '거리' 보존)와 달리, t-SNE는 '지역적 구조(Local structure)' 보존에 집중.

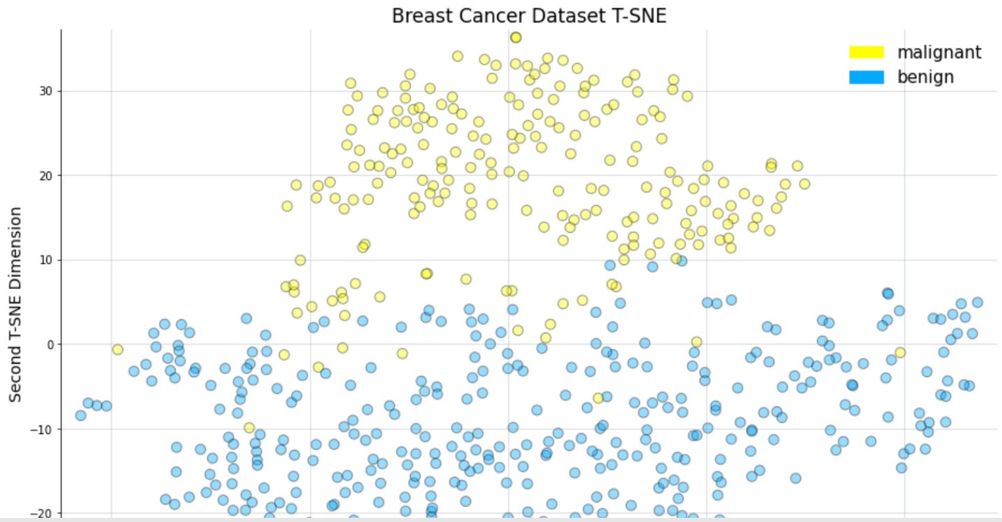
```python
from sklearn.manifold import TSNE

# 1. t-SNE 도구 설정
# (n_components=2가 기본값이므로 생략됨)
tsne = TSNE(random_state = 42)

# 2. 스케일링된 데이터로 t-SNE 변환 실행
canc_tsne = tsne.fit_transform(canc_norm)

# 3. 결과 확인
# Number of Features in Breat Cancer DataSet Before T-SNE : 30
# Number of Features in Breat Cancer DataSet After T-SNE : 2
```

차원 축소: t-SNE

[3단계] 차원 축소 적용 및 시각화

기법 3: t-SNE

2D 시각화(Scatter Plot) 분석



두 그룹을 **가장 명확하게 두 개의 덩어리(군집)로 분리**
시각적으로 그룹 간의 경계가 뚜렷하게 나타나며, 이는 t-SNE가 데이터의 '지역적 이웃 구조'를 잘 찾아냈음을 의미

결론

- t-SNE는 유방암 데이터의 구조를 찾는 데 훌륭한 성능을 보여줌.
- 데이터 시각화 시 **여러 기법을 시도**하여 특정 데이터셋에 가장 적합한 것을 찾는 것이 중요함.
- t-SNE는 특히 '지역적 이웃 패턴'이 잘 정의된(명확한) 데이터셋에서 더 잘 작동하는 경향이 있음.

한계

- PCA, MDS에 비해 계산 속도가 느릴 수 있음.
- 군집 간의 '거리'나 '크기'는 실제 의미를 갖지 않을 수 있음 (오직 '누가 누구와 뭉쳐 있는가'가 중요).

정리

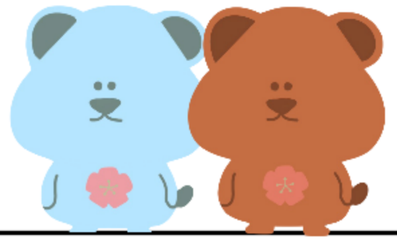
기법별 요약 및 비교

기법	핵심 원리 (2D 변환 기준)	유방암 데이터 시각화 결과	특징 및 장단점
PCA	분산(Variance) 보존	그룹이 분리되나, 일부 겹침. 전반적인 분포를 보여줌.	장점: 빠르다, 축(PC1, PC2)이 해석 가능(히트맵). 한계: 비선형/복잡한 구조에 약함.
MDS	거리(Distance) 보존	그룹이 잘 분리됨. PCA와 다른 형태의 분포.	장점: 샘플 간의 '상대적 거리'를 보존, 비선형 구조 가능. 한계: PCA보다 느릴 수 있음, 축 해석이 어려움.
t-SNE	이웃(Neighbor) 보존	가장 명확한 군집으로 분리. 경계가 뚜렷함.	장점: 군집 시각화에 매우 강력, 복잡한 비선형 구조에 특화. 한계: 계산량이 많음, 축/군집 간 거리 해석 주의.

30차원의 유방암 데이터는 세 가지 기법 모두 2차원 시각화를 통해 성공적으로 그룹 분리가 가능했는데, 목적에 따라 가장 적합한 기법이 다르다.

- PCA: 가장 빠르고, 데이터의 전반적인 구조와 주요 성분을 확인할 때
- MDS: 샘플 간의 상대적인 거리 관계가 중요할 때
- t-SNE: 데이터의 숨겨진 군집을 시각적으로 명확하게 확인하고 싶을 때

THANK YOU



EWHA EURON