

차원 축소



다차원의 피처를 차원 축소해 피처 수를 줄이면 더 직관적으로 데이터를 해석할 수 있다.

1. 피처 선택(feature selection): 데이터의 특징을 잘 나타내는 주요 피처만 선택
 - 특정 피처 종속성 강한 불필요한 피처 삭제
 - 기존 피처 → 저차원의 중요 피처로 압축 추출 → 기존의 피처와 완전히 다른 값

2. 피처 추출(feature extraction)

- 단순 압축이 아닌, 피처를 함축적으로 더 잘 설명할 수 있는 또 다른 공간으로 매핑해 추출
- 기존 피처가 인지하기 어려웠던 잠재적인 요소(Latent Factor)를 추출
- PCA, SVD, NMF

(1) 차원 축소

- 차원 축소는 매우 많은 피처로 구성된 다차원 데이터 세트의 차원을 축소해 새로운 차원의 데이터 세트를 생성하는 것
- 차원이 증가할수록 데이터 포인트 간의 거리가 멀어지게 되고, 희소(sparse)한 구조를 가짐
- 차원 증가 → 예측 신뢰도 하락, 개별 피처 간 상관 관계가 높음(다중 공선성 문제)

(2) 언제

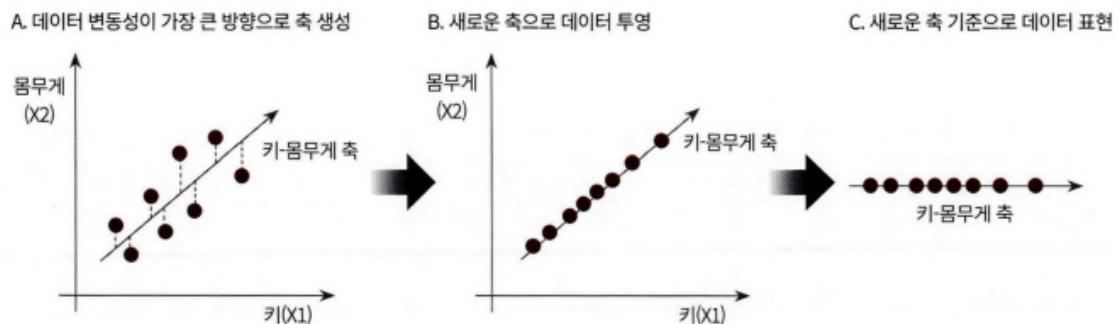
- 이미지 데이터에서 잠재된 특성 → 피처로 도출 → 함축적 이미지 변환&압축
- 텍스트 문서의 숨겨진 의미
 - SVD와 NMF는 이러한 시맨틱 토픽 (Semantic Topic) 모델링을 위한 기반 알고리즘으로 사용

1. PCA(Principal Component Analysis)

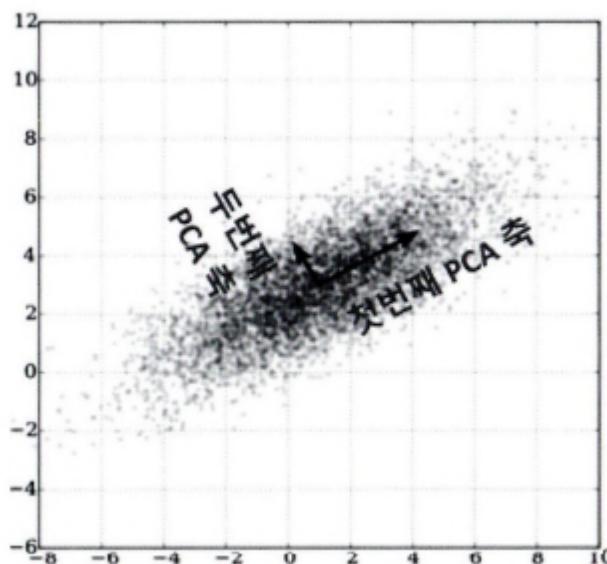


여러 변수 간에 존재하는 상관관계를 이용해 이를 대표하는 주성분(Principal Component)을 추출해 차원을 축소 하는 기법

- 기존 데이터의 정보 유실이 최소화
- 데이터 주성분: 분산(PCA는 가장 높은 분산을 가지는 데이터의 축으로 차원 축소)



- 첫 번째 축: 큰 데이터 변동성(Variance)을 기반
 - 두 번째 축: 이 벡터 축에 직각이 되는 벡터(직교 벡터)
 - 세 번째 축: 다시 두 번째 축과 직각
- 벡터 축 개수 만큼의 차원으로 축소



- 원본 피처 개수보다 매우 작은 주성분 → 원본 데이터의 총 변동성 설명

- 고유값 분해 (공분산 행렬) → 입력 데이터 선형 변환 → 고유벡터의 크기 = 고윳값 = 분산

선형 변환, 공분산 행렬, 고유 벡터

- 특정 벡터에 x 행렬 A → 새로운 벡터
- 공분산: 두 변수 간의 변동
- 공분산 행렬: 여러 변수와 관련된 공분산을 포함하는 정방형 행렬
 - 정방행렬(Square Matrix): 열과 행이 같음
 - 대칭행렬(Symmetric Matrix): 항상 고유벡터를 직교행렬(orthogonal matrix)로, 고유값을 정방 행렬로 대각화할 수 있음

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

- 대각선 = 각 변수(X, Y, Z)의 분산
- 대각선 이외 = 변수 쌍 간 공분산
- 고유벡터: 행렬 A를 곱하더라도 방향이 변하지 않고 그 크기만 변하는 벡터
 - $Ax = ax$ (A 는 행렬, x 는 고유벡터, a 는 스칼라값)

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

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

- C = 입력 데이터의 공분산 행렬
- n X n의 직교행렬 * n X n 정방행렬 * 행렬 P의 전치 행렬

- i번째 고유벡터/i번째 고유 벡터의 크기
- e1은 가장 분산이 큰 방향을 가진 고유 벡터, e2는 e1에 수직이며 다음으로 가장 분산이 큰 방향을 가진 고유벡터

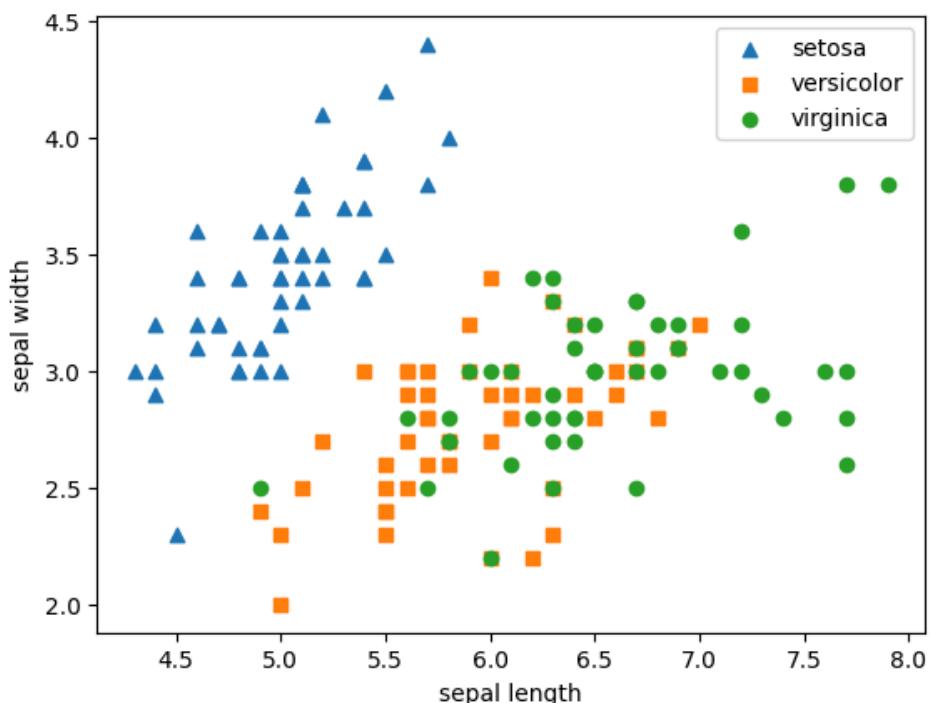


PCA?

多 속성의 원본 데이터 → 핵심 데이터로 압축

- 입력 데이터의 공분산 행렬은 고유벡터와 고유값으로 분해
- 분해된 고유벡터 → 입력 데이터를 선형 변환
 1. 입력 데이터 세트의 공분산 행렬을 생성
 2. 공분산 행렬의 고유벡터와 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

붓꽃 예제



- setosa만 분류 가능: $x < 6$, $y > 3$ 일정하게 분류됨

▼ 코드

```
from sklearn.datasets import load_iris
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
iris = load_iris()
columns = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
df = pd.DataFrame(iris.data, columns=columns)
df['target'] = iris.target
df.head(3)
#setosa는 세모, versicolor는 네모, virginica는 동그라미

markers = ['^', 's', 'o']

#setos target 값은 0, versicolor1 virginica 2
for i, marker in enumerate(markers):
    x_axis_data = df[df['target'] == i]['sepal_length']
    y_axis_data = df[df['target'] == i]['sepal_width']

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

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

1. 개별 속성 스케일링 - 사이킷런의 StandardScaler

- 여러 속성의 값을 연산해야 하므로 속성의 스케일에 영향을 받음
- 각 속성값을 동일한 스케일로 변환하는 것이 필요

▼ 코드

```
from sklearn.preprocessing import StandardScaler

iris_scaled = StandardScaler().fit_transform(df.iloc[:, :-1])
```

2. PCA 적용

- 4차원(4개 속성)의 붓꽃 데이터를 2차원(2개의 PCA 속성) PCA 데이터로 변환
- `PCA(n_components =)`
 - `fit(), transform()`

```
from sklearn.decomposition import PCA

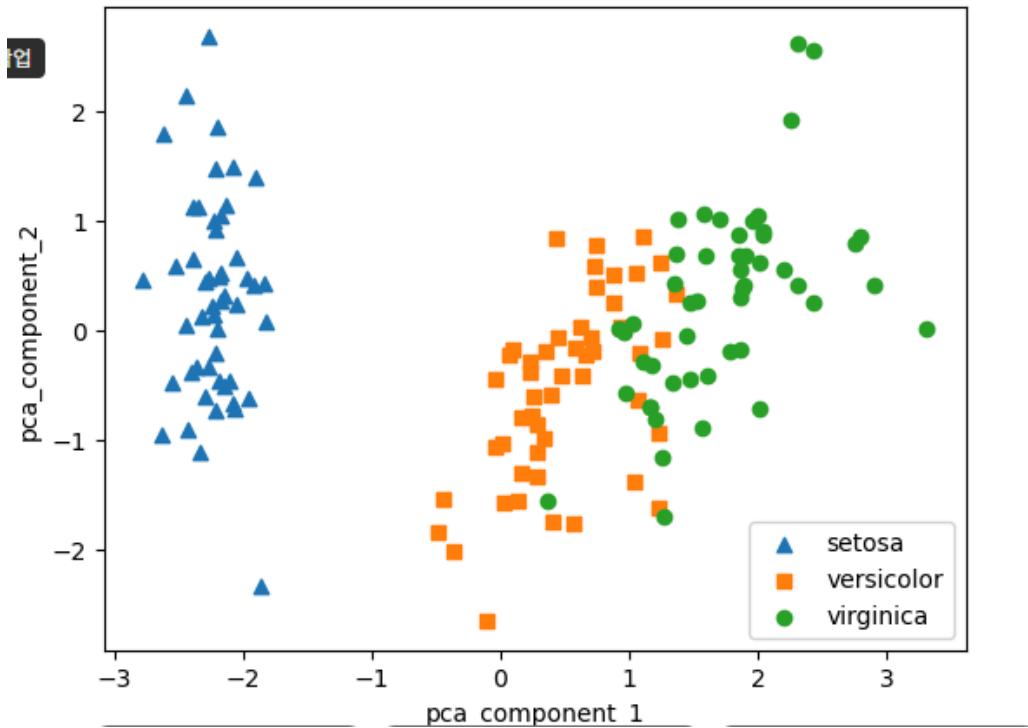
pca = PCA(n_components = 2)

pca.fit(iris_scaled)

iris_pca = pca.transform(iris_scaled)
print(iris_pca.shape)
```

```
1 pca_columns = ['pca_component_1', 'pca_component_2']
2 df_pca = pd.DataFrame(iris_pca, columns=pca_columns)
3 df_pca['target'] = iris.target
4 df_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



- setosa 명확히 구분/ 나머지도 조금 겹치지만 잘 구분됨
- 첫 번째 새로운 축 pca_component_1 원본 데이터의 변동성을 잘 반영

3. PCA Component별로 원본 데이터의 변동성을 얼마나 반영?

- PCA 객체의 `explained_variance_ratio_` 속성

```
1 print(pca.explained_variance_ratio_)
...
[0.72962445 0.22850762]
```

- PCA를 2개 요소로만 변환해도 원본 데이터의 변동성을 95% 설명

4. 분류를 적용한 후 결과를 비교

- Estimator `RandomForestClassifier`
- `cross_val_score()=3`

```

1 from sklearn.ensemble import RandomForestClassifier
2 from sklearn.model_selection import cross_val_score
3 import numpy as np
4
5 rcf = RandomForestClassifier(random_state=156)
6 scores = cross_val_score(rcf, iris.data, iris.target, scoring='accuracy', cv=3)
7 print('원본 데이터 교차 검증 개별 정확도:', scores)
8 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

```

```

1 pca_X = df_pca[['pca_component_1', 'pca_component_2']]
2 scores_pca = cross_val_score(rcf, pca_X, iris.target, scoring='accuracy', cv=3 )
3 print('PCA 변환 데이터 교차 검증 개별 정확도:', scores_pca)
4 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

```

- 하락
- 8%의 정확도 하락은 비교적 큰 성능 수치의 감소
- 하지만 속성 개수가 50% 감소 = PCA 변환 후에도 원본 데이터의 특성을 상당 부분 유지

신용카드 고객 데이터 세트(Credit Card Clients Data Set)

0. 데이터 명 변경, 상관관계 확인

▼ 코드

```

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

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)

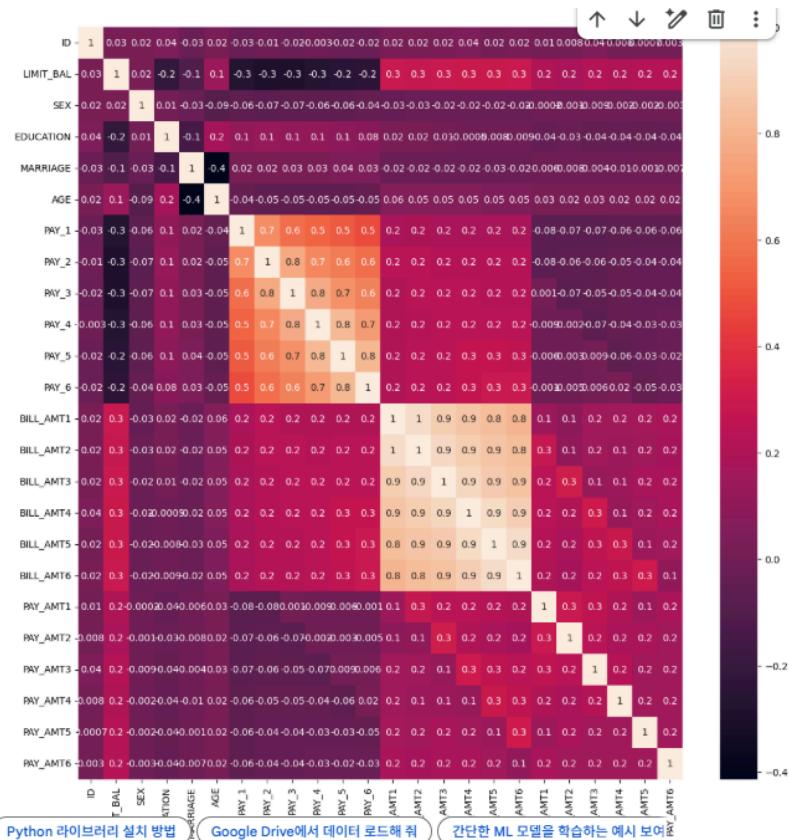
corr = X_features.corr()
plt.figure(figsize=(14,14))

```

```

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

```



- BILL_AMT1 ~ BILL_AMT6 6개 속성끼리의 상관도가 대부분 0.9 이상으로 매우 높음
- PAY_1 ~ PAY_6까지의 속성 역시 상관도가 높습니다.



소수의 PCA만으로 자연스럽게 속성들의 변동성을 수용

1. PCA 변환

```

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

cols_bill = ['BILL_AMT'+str(i) for i in range(1,7)]

```

```
scaler = StandardScaler()
df_cols_scale= scaler.fit_transform(df[cols_bill])
pca = PCA(n_components=2)
pca.fit(df_cols_scale)
print('PCA component별 변동성: ', pca.explained_variance_ratio_)
```

PCA component별 변동성: [0.90555253 0.0509867]

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

2. 상호 비교

```
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('원본 데이터 교차 검증 개별 정확도:', scores)
print('원본 데이터 평균 정확도:{0:.4f}'.format(np.mean(scores)))
```

원본 데이터 평균 정확도:0.6206

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

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_features)

pca = PCA(n_components=6)
X_pca = pca.fit_transform(X_scaled)
```

```
scores_pca = cross_val_score(rcf, X_pca, y_target, scoring='accuracy', cv=3)

print('CV=3인 경우의 PCA 변환된 개별 Fold 세트별 정확도:', scores_pca)
print('PCA 변환 데이터 세트 평균 정확도: {:.4f}'.format(np.mean(scores_pc
a)))
```

PCA 변환 데이터 세트 평균 정확도: 0.7934

- 약 1~2% 정도의 예측 성능 저하
- 컴퓨터 비전 영역: 얼굴 인식 원본 얼굴 이미지 변환하여 사용

LDA(Linear Discriminant Analysis)

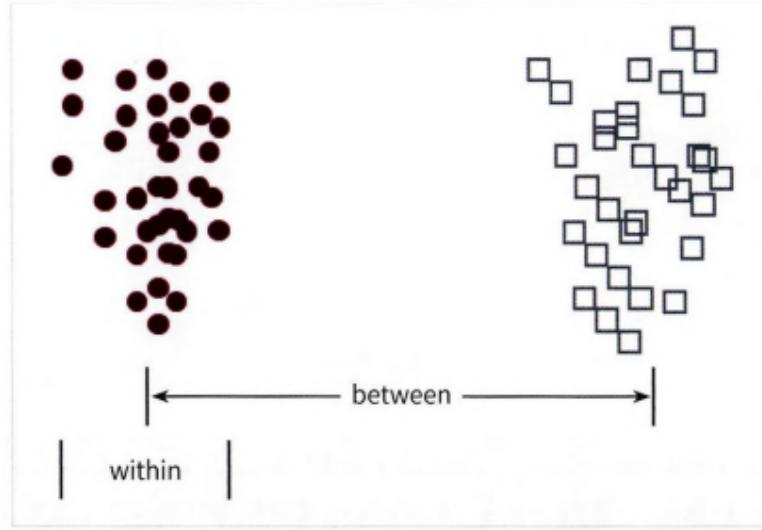
- LDA(Linear Discriminant Analysis)는 선형 판별 분석법으로 불리며, PCA와 매우 유사
 - 입력 데이터 세트를 저차원 공간에 투영해 차원을 축소
- LDA는 지도학습의 분류(Classification)에서 사용하기 쉽도록 개별 클래스를 분별할 수 있는 기준을 최대한 유지하면서 차원을 축소



PCA vs LDA

- PCA 입력 데이터의 변동성의 가장 큰 축
- LDA는 입력 데이터의 결정 값 클래스를 최대한으로 분리할 수 있는 축

- 클래스 간 분산(between-class scatter) & 클래스 내부 분산(within-class scatter) 비율 최대화
- 클래스 간 분산은 최대한 크게 가져가고, 클래스 내부의 분산은 최대한 작게 가져가는 방식



1. 클래스 내부와 클래스 간 분산 행렬

- a. 입력 데이터의 결정 값 클래스별로 개별 피처의 평균 벡터(mean vector)를 기반으로 구함

2. 수식으로 고유벡터로 분해

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

- 고유값 큰 순으로 K개 추출(LDA 변환 차수)
- 3. 고유값이 가장 큰 순으로 추출된 고유벡터를 이용하여 새롭게 입력 데이터 변환

붓꽃 데이터

- LDA는 단순히 입력 특성(X)만 보고 축을 찾지 않고 데이터가 어떤 클래스(라벨 y)에 속하는지도 이용해서 클래스 간의 구분이 잘 되도록 새로운 축을 학습

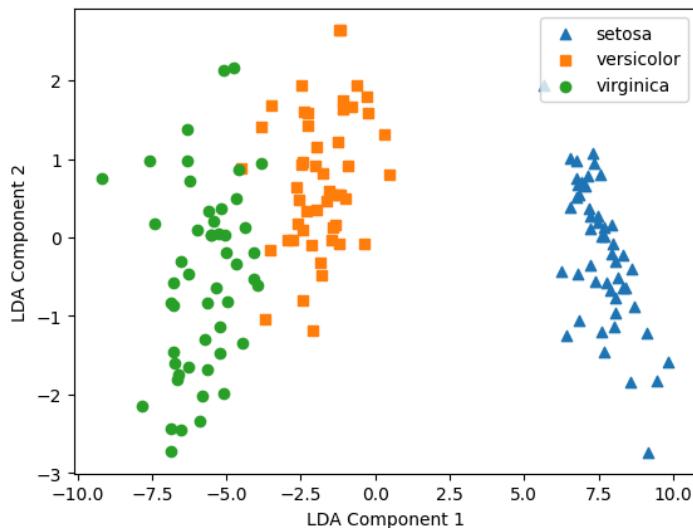
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris
```

```

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

lدا = LinearDiscriminantAnalysis(n_components=2)
lدا.fit(iris_scaled, iris.target)
iris_lدا = lدا.transform(iris_scaled)
print(iris_lدا.shape)

```



SVD(Singular Value Decomposition)

- 정방행렬뿐 + 행과 열의 크기가 다른 행렬에도 적용
- 특이값 분해

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

- 행렬 1/V에 속한 벡터는 특이벡터(singular vector) → 서로 직교
- 대각행렬, 대각 위치만 0이 아니고 나머지는 모두 0
- 0 아닌 값이 행렬 A의 특이값
- $m \times n \rightarrow m \times m, m \times n, n \times n$ 으로 분해

$$A = U \Sigma V'$$

$$A = U \Sigma V'$$

- 보통은 비대각인 부분과 대각원소 중에 특이값이 0인 부분도 모두 제거
- Truncated SVD
 - 대각 원소 중 상위 몇 개만 추출
- 보통 넘파이나 사이파이 라이브러리 이용
 - 넘파이의 SVD 모듈인 numpy.linalg.svd

예제

1. 행렬 생성, SVD 적용하여 U, Sigma, Vt 도출

```

import numpy as np
from numpy.linalg import svd

np.random.seed(121)
a = np.random.randn(4,4)

```

```

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))

```

- Sigma 행렬: 대각 위치한 값만 0이 아니고, 나머지는 0이니까 1차원 행렬로 표현

(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]]
```

- 다시 원본 행렬로 복원되는지 → 내적
 - sigma는 다시 0 포함한 대칭 행렬로 변환해야 됨

```
Sigma_mat = np.diag(Sigma)
a_ = np.dot(np.dot(U, Sigma_mat), Vt)
```

- 로우 간 의존성 있을 때
- a 행렬 3번째 로우 = 첫번째 로우 + 두번째 로우
- 4번째 로우 = 첫 번째 로우

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

```
U, Sigma, Vt = svd(a)
print(U.shape, Sigma.shape, Vt.shape)
```

```
print('Sigma Value:\n', np.round(Sigma, 3))
```

- 다시 분해하면 Sigma 값 중 2개가 0으로 변함

(4, 4) (4,) (4, 4)

Sigma Value:

[2.663 0.807 0. 0.]

- 다시 원본으로 복원
- Sigma의 0에 대응되는 U, Sigma, Vt 데이터 제외하고 복원

```
U_ = U[:, :2]
Sigma_ = np.diag(Sigma[:2])
Vt_ = Vt[:2]
print(U_.shape, Sigma_.shape, Vt_.shape)
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]]

2. Truncated SVD를 이용해 행렬을 분해

- 원본 행렬을 정확하게 다시 원복할 수는 없지만
- 데이터 정보가 압축되어 분해됨에도 불구하고 상당한 수준으로 원본 행렬을 근사
- 사이파이에서만 지원, `scipy.sparse.linalg.svds`

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

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('\sigma 값 행렬:', Sigma)

num_components = 4
U_tr, Sigma_tr, Vt_tr = svds(matrix, k=num_components)
print('\nTruncated SVD 차원:', U_tr.shape, Sigma_tr.shape, Vt_tr.shape)
print('\nTruncated SVD Sigma 값 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, 3)

```

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)

σ 값 행렬: [3.2535007 0.88116505 0.83865238 0.55463089 0.35834824 0.0349925]

Truncated SVD 차원: (6, 4) (4,) (4, 6)

Truncated SVD Sigma 값 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]] 3
<:10: SyntaxWarning: invalid escape sequence '\s'
<:10: SyntaxWarning: invalid escape sequence '\s'
/tmpp/ipython-input-806651789.py:10: SyntaxWarning: invalid escape sequence '\s'
    print('\sigma 값 행렬:', Sigma)

```

사이킷런 TruncatedSVD 클래스를 이용한 변환

- 사이파이의 svds와 같이 Truncated SVD 연산을 수행해 원본 행렬을 분해한 U, Sigma, Vt 행렬을 반환하지는 않음
- PCA 클래스와 유사하게 fit()와 transform()을 호출해 원본 데이터를 몇 개의 주요 컴포넌트(즉, Truncated SVD의 K 컴포넌트 수)로 차원을 축소해 변환

-

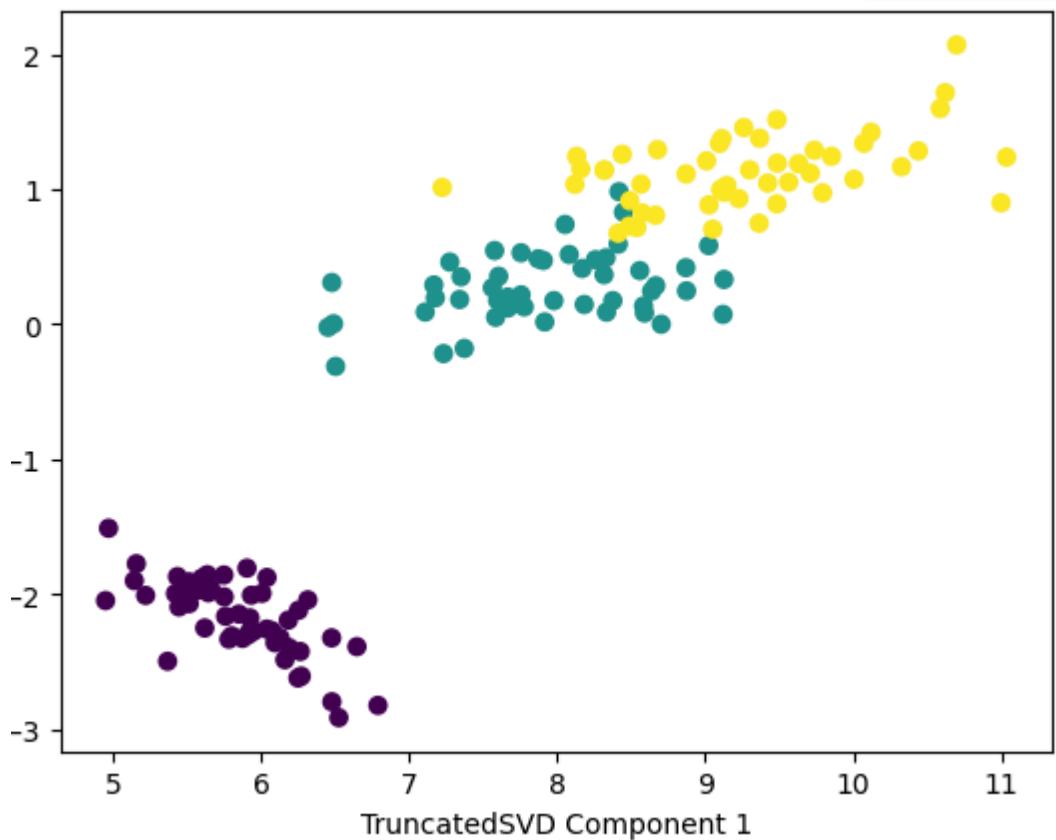
```
from sklearn.decomposition import TruncatedSVD, PCA
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
%matplotlib inline

# Iris 데이터 로드
iris = load_iris()
iris_ftrs = iris.data

tsvd = TruncatedSVD(n_components=2)
iris_tsvd = tsvd.fit_transform(iris_ftrs)

# 산점도로 시각화 (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')

plt.show()
```



- 품종별로 어느 정도 클러스터링이 가능할 정도로 각 변환 속성으로 뛰어난 고유성
- 두 개 클래스 모두 SVD 를 이용해 행렬을 분해
- 붓꽃 데이터를 스케일링으로 변환한 뒤에 TruncatedSVD와 PCA 클래 스 변환을 해보면 두 개가 거의 동일함

```

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

iris = load_iris()
iris_ftrs = iris.data
y = iris.target

scaler = StandardScaler()
iris_scaled = scaler.fit_transform(iris_ftrs)

```

```

tsvd = TruncatedSVD(n_components=2)
iris_tsvd = tsvd.fit_transform(iris_scaled)

pca = PCA(n_components=2)
iris_pca = pca.fit_transform(iris_scaled)

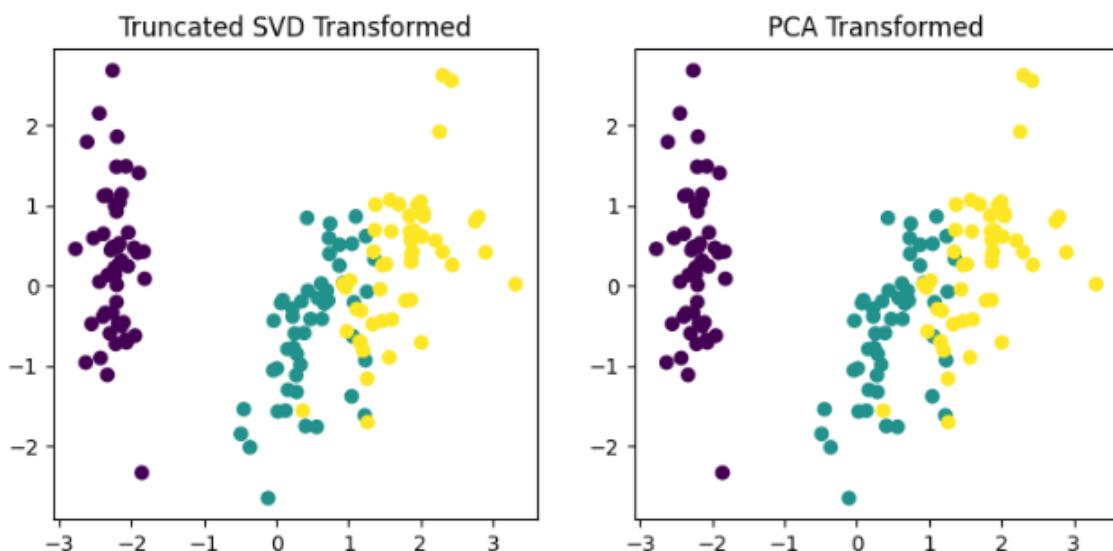
fig, (ax1, ax2) = plt.subplots(figsize=(9, 4), ncols=2)

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')

plt.show()

```



```

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

```

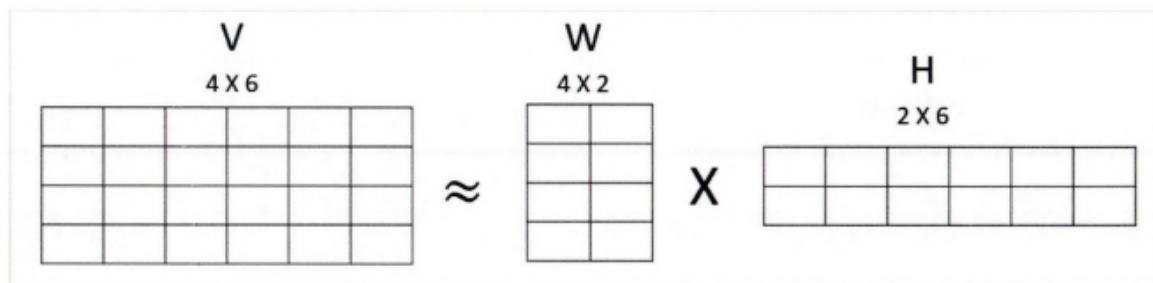
2.3376092728177866e-15
-6.938893903907228e-18

→ 거의 동일

- 하지만 PCA는 밀집 행렬(Dense Matrix)에 대한 변환만 가능하며 SVD는 희소 행렬(Sparse Matrix)에 대한 변환도 가능
- 이미지 분석 & 텍스트의 토픽 모델링 기법인 LSA(Latent Semantic Analysis)의 기반 알고리즘

NMF(Non-Negative Matrix Factorization)

- 낮은 랭크를 통한 행렬 근사 (Low—Rank Approximation) 방식의 변형
- 원본 행렬 내의 모든 원소 값이 모두 양수(0 이상)라는 게 보장되면 간단히 두개로 분해



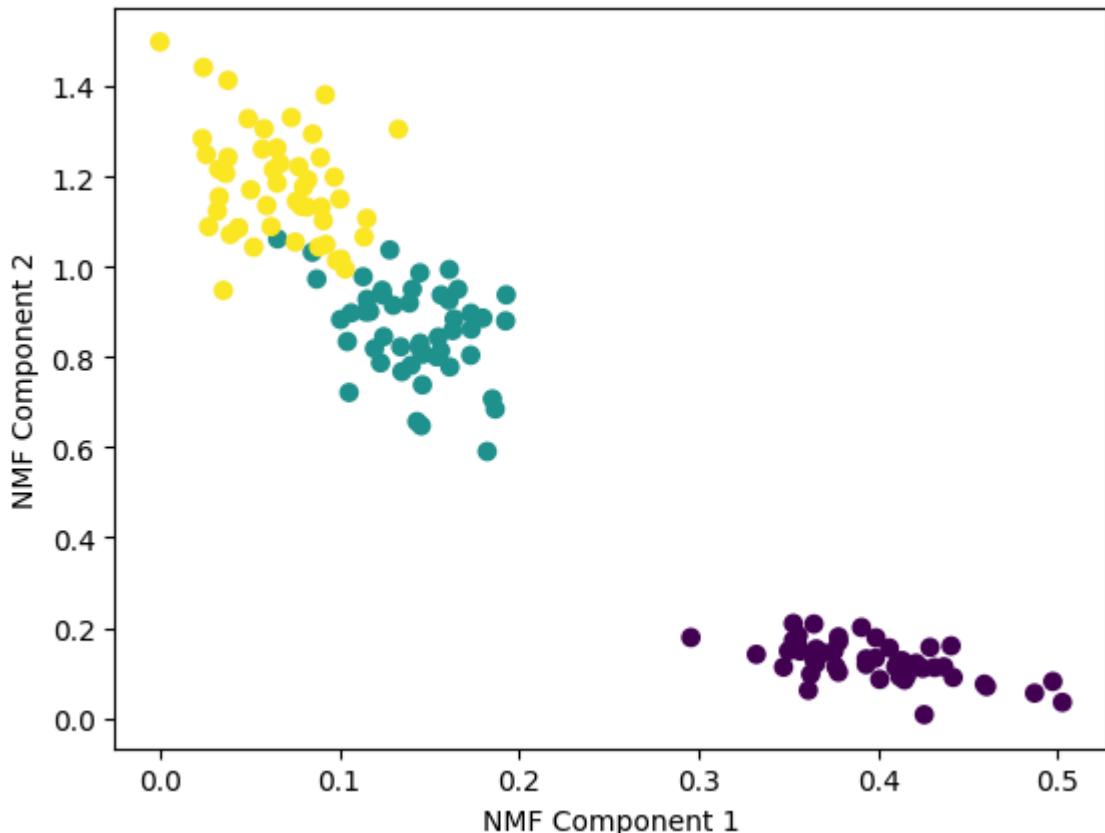
- 행렬 분해: 일반적으로 길고 가는 행렬 W & 작고 넓은 행렬 H
- 잠재 요소를 특성으로 가지게 됨
 - W : 원본 행에서 잠재 요소의 값
 - H : 원본 열로 잠재 요소가 어떻게 구성?
- 사이킷런에서 NMF 클래스로 지원

```
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)
iris_nmf = nmf.fit_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')  
  
plt.show()
```



- 이미지 압축을 통한 패턴 인식, 텍스트의 토픽 모델링 기법, 문서 유사도 및 클러스터링, 영화 추천과 같은 추천(Recommendations) 영역