파머완 06 차원 축소

01 차원 축소 개요

차원 축소

: 다차원 데이터 세트의 차원을 축소해 새로운 차원의 데이터 세트 생성

- 차원이 증가할수록

  → 데이터 포인트 간의 거리가 기하급수적으로 멀어짐 & 희소한 구조를 가짐

피처가 많을 경우

- 예측 신뢰도 감소
- 피처 간 상관관계가 높을 가능성이 있음 → 다중 공선성 문제로 예측 성능 저하
- 데이터 특성 시각화 불가능

차원 축소 목표

- 데이터를 압축하여 시각화 가능
- 학습에 필요한 처리 능력 줄일 수 있음

1. 피처 선택

: 특정 피처에 종속성이 강한 불필요한 피처 제거

& 특징을 잘 나타내는 주요 피처만 선택

2. 피처 추출

: 기존 피처를 저차원의 중요 피처로 압축하여 추출

→ 기존의 피처와 완전히 다른 값이 됨

- 기존 피처의 단순 압축 X
- 피처를 함축적으로 더 잘 설명하는 또 다른 공간으로 매핑하여 추출하는 것

함축적인 특성 추출

: 기존 피처가 인지하기 어려웠던 잠재적인 요소를 추출하는 것

잠재적인 요소를 찾는 차원 축소 알고리즘: PCA, SVD, NMF

---

1. **잠재된 특성을 피처로 도출 → 함축적 형태의 이미지 변환과 압축을 수행함**

- 변환된 이미지: 분류 수행 시 과적합 영향력 감소 → 원본 데이터보다 예측 성능이 증가함
- 원본 이미지를 사용한다면?

  → 비슷한 이미지라도 적은 픽셀의 차이가 잘못된 예측으로 이어질 수 있음

2. **텍스트 문서의 숨겨진 의미 추출**

- 문서 내 단어들의 구성에서 숨겨진 semantic 의미나 topic을 잠재요소로 간주하고 찾아냄
- SVD, NMF: semantic topic 모델링을 위한 기반 알고리즘으로 사용

---

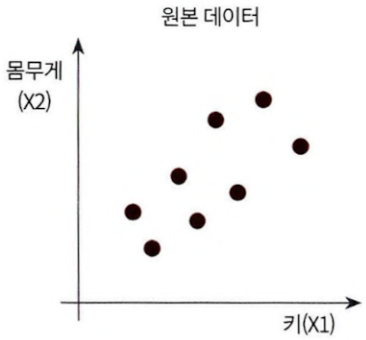
02 PCA(Principal Component Analysis)

PCA 개요

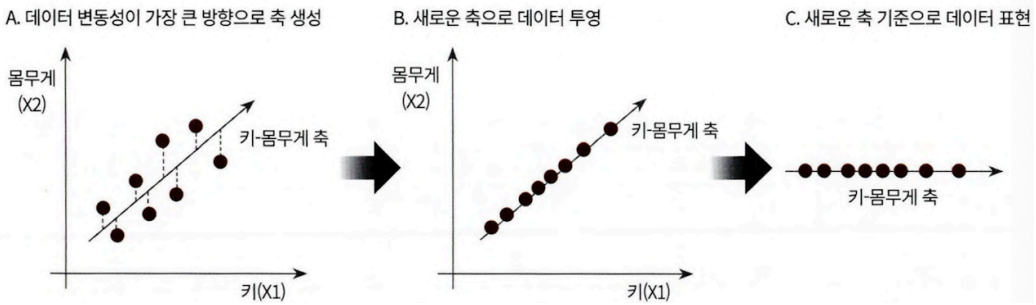
PCA

: 변수 간의 상관관계를 이용해 **주성분을 추출**하여 차원을 축소하는 기법

- 기존 데이터 정보 유실 최소화
- 가장 높은 분산을 갖는 **데이터**의 축을 찾아 차원 축소 → PCA의 주성분



- 데이터 변동성이 가장 큰 방향으로 축 생성
- 생성된 축으로 데이터 투영



PCA 과정

1. 가장 큰 데이터 변동성(variance)를 기반으로 첫번째 벡터 축 생성
2. 첫번째 벡터 축에 직각이 되는 벡터를 축으로 두번째 축 생성
3. 두번째 축과 직각이 되는 벡터를 축으로 세번째 축 생성
4. 이렇게 생성된 벡터 축에 원본 데이터를 투영

   → 벡터 축의 수만큼의 차원으로 차원 축소

✅ PCA란?

원본 데이터 피처 개수에 비해 매우 작은 주성분으로

원본 데이터의 총 변동성을 대부분 설명할 수 있는 분석법!

PCA - 선형대수 관점

: 입력 데이터의 공분산 행렬을 고유값 분해로 구한 **고유벡터**에 입력 데이터 선형 변환

- 고유벡터: PCA 주성분 벡터 - 입력 데이터의 분산이 큰 방향

- 고윳값: 고유벡터의 크기 - 입력 데이터의 분산

고유벡터

: 행렬을 곱해도 방향이 변하지 않고 크기만 변하는 벡터

- Ax = ax (A: matrix, x: eigen vector, a: scalar)
- 정방 행렬: 차원 수만큼의 고유벡터를 가질 수 있음
- 행렬이 작용하는 힘의 방향과 관계가 있음 → 행렬 분해에 사용

선형변환

: 특정 벡터 x matrix A ⇒ vector B 변환

- 특정 벡터를 다른 공간으로 투영하는 개념과 동일
  → 이 행렬을 공간으로 가정하는 것

공분산

: 두 변수 간의 변동

- Cov(X,Y) > 0 : X가 증가할 때 Y도 증가

공분산 행렬

: 여러 변수와 관련된 공분산을 포함하는 정방형 행렬

	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,Z의 분산
- 이 외의 원소: 모든 변수 쌍의 공분산
  - Cov(X,Y): -0.71
  - Cov(Y,Z): 0.28
  - Cov(Z,X): 0.24

- 개별 분산값을 대각 원소로 하는 대칭행렬

- 대칭행렬: 항상 고유벡터를 직교행렬로, 고유값을 정방 행렬로 대각화 가능

공분산 C 분해

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

- C = 고유벡터 직교 행렬 * 고유값 정방 행렬 * 고유벡터 직교 행렬의 전치 행렬
- ei: 분산이 i번째로 큰 방향을 가진 고유벡터
- λi: 고유벡터 ei의 크기

✅ **선형 변환과 PCA?**

입력 데이터의 공분산 행렬 → 고유벡터, 고유값으로 분해

분해된 고유벡터를 이용하여 입력 데이터를 선형 변환하는 방식이 **PCA**

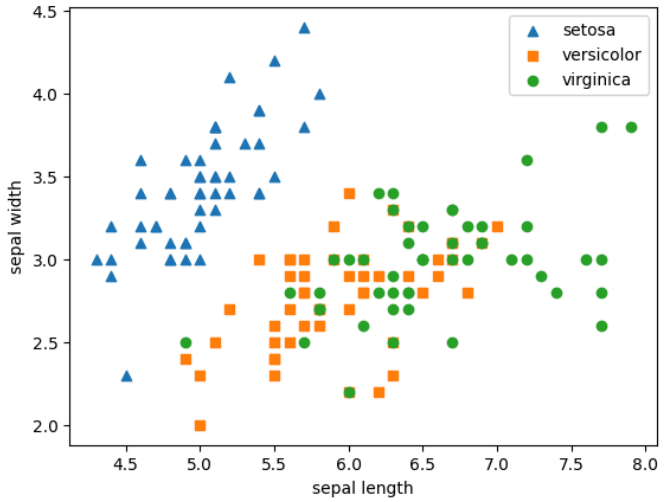
PCA 과정

1. 입력 데이터 세트의 공분산 행렬 생성
2. 공분산 행렬의 고유벡터와 고유값 계산
3. 고유값이 큰 순으로 K개(PCA 변환 차수)만큼 고유벡터 추출
4. 고유값이 큰 순으로 추출된 고유벡터를 이용해 새롭게 입력 데이터 변환

---

iris_data 실습

원본 데이터 세트 시각화



- Setosa: sepal_length < 6.0 , sepal_width > 3.0 일정하게 분포
- Versicolor, Virginica: 분류가 어려움

PCA 변환

개별 속성 스케일링

PCA 압축 전, 각 속성값을 동일한 스케일로 변환해야함

```
from sklearn.preprocessing import StandardScaler

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

2차원 PCA 데이터 변환

PCA class: n_components를 생성 파라미터로 입력받음

- n_components: 변환할 차원의 수

```
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)
```

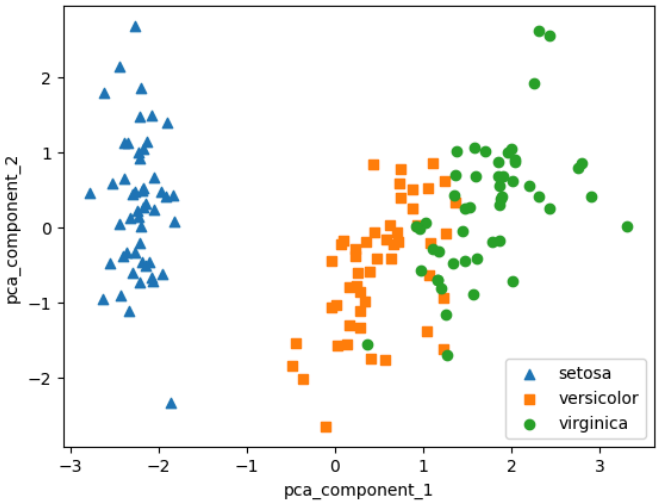
PCA 변환 후 DataFrame 변환

```
# PCA 변환된 데이터의 칼럼명을 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)
```

output)

	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

PCA 변환 후 시각화



- Setosa: pca_component_1 축으로 명확하게 구분 가능

- Versicolor, Virginica: 겹치는 부분이 존재하지만, 잘 구분됨

⇒ PCA의 첫번째 새로운 축이 원본 데이터의 변동성을 잘 반영했기 때문

PCA Component 별 데이터 변동성 비율

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

output)

```
[0.72962445 0.22850762
```

- 2개의 요소가 원본 데이터의 변동성을 95% 설명

원본 데이터 vs PCA 데이터 비교

원본 데이터 성능

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

PCA 데이터 성능

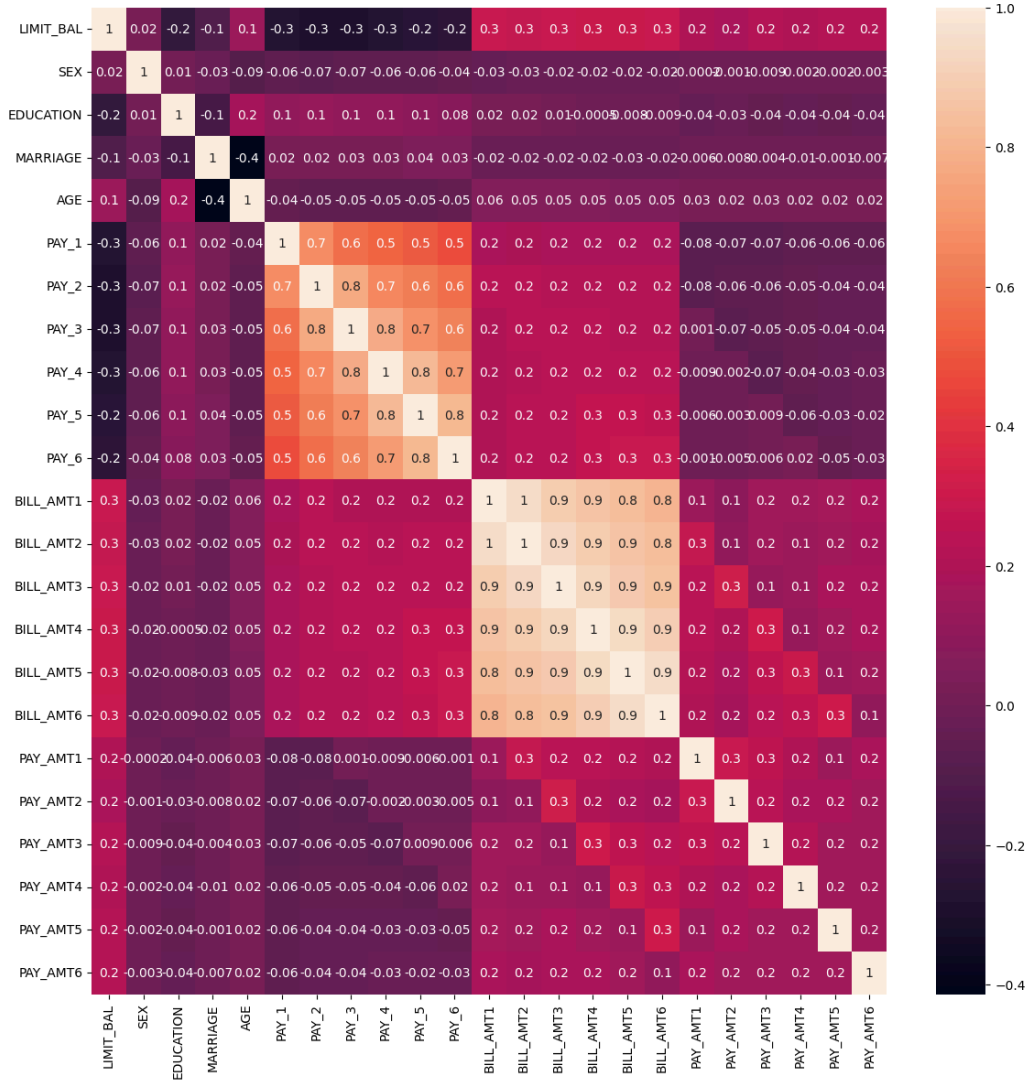
```
PCA 변환 데이터 교차 검증 개별 정확도: [0.88 0.88 0.88]
PCA 변환 데이터 평균 정확도: 0.88
```

- 속성이 50% 감소 → 예측 성능 감소
- PCA 변환 후에도 원본 데이터의 특성을 상당 부분 유지하고 있음

---

credit card clients data set 실습

feature 상관관계 시각화



- BILL_AMT1 ~ BILL_AMT6: 상관도 대부분이 0.9 이상
- PAY_1 ~ PAY_6: 상관도 높음

✅ 이렇게 높은 상관들도를 가진 속성들은 소수의 PCA만으로 자연스럽게 이 속성들의 변동성을 수용할 수 있음

PCA 변환

- BILL_AMT1 ~ BILL_AMT6 6개 속성 → 2개 component PCA 변환
- 개별 component 변동성 확인

```
#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
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_)
```

output)

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

⇒ 2개의 component로 6개 속성의 변동성을 95%이상 설명 가능!

원본 데이터 vs PCA 데이터 비교

원본 데이터 성능

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

PCA 데이터 성능

```
CV=3인 경우의 PCA 변환된 개별 Fold세트별 정확도: [0.7912 0.7974 0.802 ]
PCA 변환 데이터 세트 평균 정확도:0.7969
```

- 약 1~2% 성능 저하
- PCA의 뛰어난 압축 능력을 보여줌

PCA 활용 분야 - 컴퓨터 비전

얼굴 인식: PCA 변환으로 원본 얼굴 이미지를 변환해 사용함

---

03 LDA(Linear Discriminant Analysis)

LDA 개요

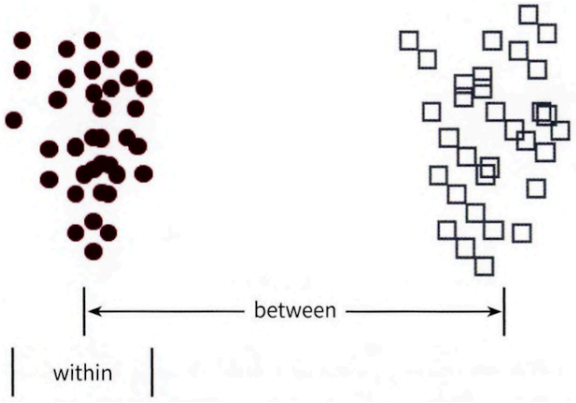
LDA

: 개별 클래스를 분별할 수 있는 기준을 최대한 유지하며 차원 축소

LDA 차원 축소 방법

: 클래스 간 분산과 클래스 내부 분산의 비율을 최대화하는 방식으로 차원 축소

- 클래스 간 분산 maximize & 클래스 내부 분산 minimize



1. 클래스 내부와 클래스 간 분산 행렬 구함 (mean vector 기반)
2. Sw: 클래스 내부 분산 행렬, Sb: 클래스 간 분산 행렬

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

3. 고유값이 가장 큰 순으로 K개 추출

4. 고유값이 가장 큰 순으로 추출된 고유벡터를 이용해 새롭게 입력 데이터 변환

PCA와 LDA의 차이점

PCA	LDA
입력 데이터의 변동성의 가장 큰 축을 찾음	입력 데이터의 결정 값 클래스를 최대한으로 분리할 수 있는 축을 찾음
클래스 간 분산 & 클래스 내부 분산 행렬 생성	공분산 행렬 생성

붓꽃 데이터 세트에 LDA 적용

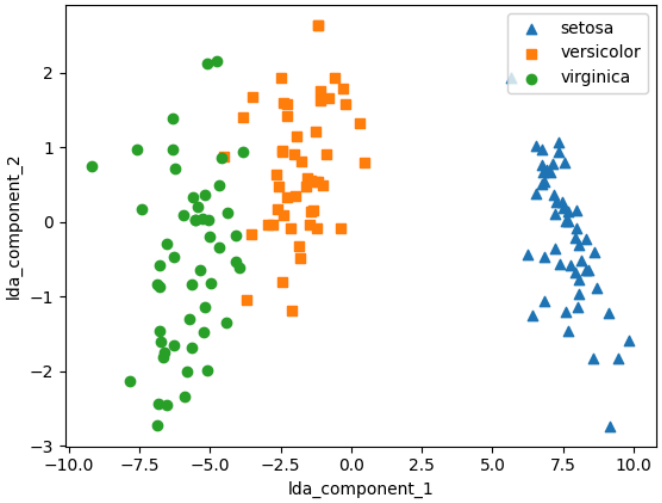
LDA 변환

LDA: 지도학습 (PCA: 비지도학습)

→ 클래스의 결정 값이 필요함

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

LDA 변환 데이터 시각화



- PCA 변환 데이터와 좌우 대칭 형태

---

04 SVD(Singular Value Decomposition)

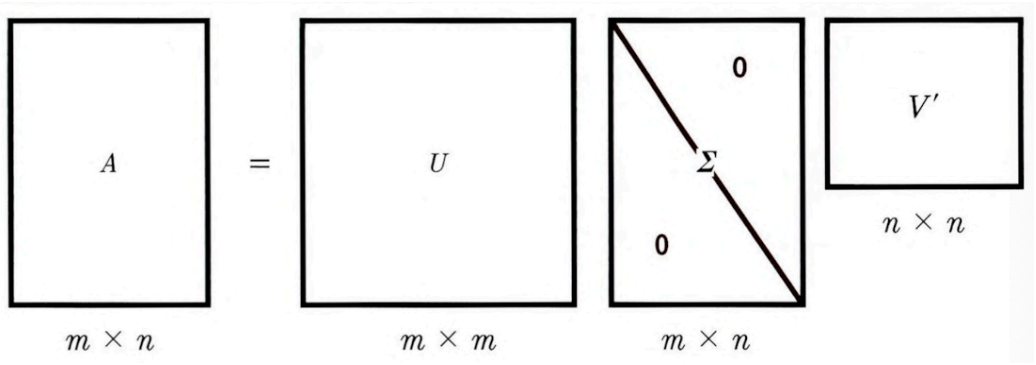
SVD 개요

SVD

: 행과 열의 크기가 다른 행렬도 분해 가능

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

A: m*n matrix ⇒ U: m*m , Σ: m*n, V′: n*n 으로 분해

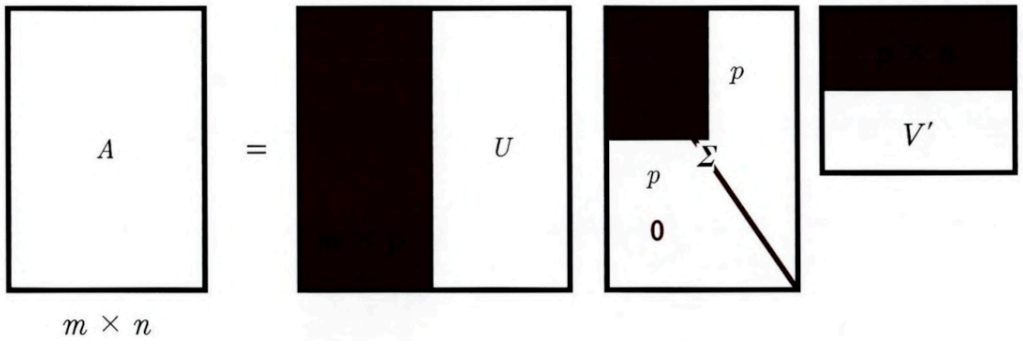


- 행렬 U,V에 속한 벡터: singular vector
  - 모든 singular vector는 서로 직교함
- Σ: 대각행렬
  - 행렬의 대각에 위치한 값을 제외한 값이 모두 0

    ⇒ Σ이 위치한 0이 아닌 값이 A의 singular value

Truncated SVD

A: m*n matrix ⇒ U: m*p , Σ: p*p, V′: p*n 으로 분해



- Σ의 비대각인 부분 + 대각 원소 중 0인 부분 모두 제거
- 제거된 Σ에 대응되는 U,V 원소도 함께 제거하여 차원 축소

넘파이 SVD 실습

랜덤 행렬 생성

: 행렬의 개별 로우끼리 의존성을 없애기 위해

```
# 넘파이 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))
```

SVD 분해

svd 파라미터에 원본 행렬 입력 → U, Sigma, V 행렬 반환

- Sigma: 0이 아닌 값만 1차원 행렬로 표현

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

output)

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

원본 행렬 복원

U,Sigma, Vt 내적

- Sigma를 0을 포함한 대칭행렬로 변환한 뒤에 내적 수행

```
# Sigma를 다시 0을 포함한 대칭 행렬로 변환
Sigma_mat=np.diag(Sigma)
a_=np.dot(np.dot(U,Sigma_mat),Vt)
```

의존성을 부여한 행렬 생성

```
a[2]=a[0]+a[1]
a[3]=a[0]
```

SVD 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))
```

output)

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

- Sigma 값 중 2개가 0으로 변함
    - 선형 독립인 로우 벡터의 개수가 2개 (행렬의 랭크=2)

원본 행렬 복원

Sigma의 0에 대응되는 U, Sigma, Vt 데이터를 제외하고 복원

- U 행렬: 선행 두개의 열만 추출
- Vt 행렬: 선행 두개의 행만 추출

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

사이파이 Truncated SVD 실습

Truncated SVD 분해

: 더 작은 차원으로 분해하므로 원본 행렬로 다시 복원할 수 X

- 원래 차원의 차수와 가깝게 잘라낼수록 원본 행렬에 더 가깝게 복원 가능

```
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('\분해 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 TruncatedS\
```

output)

6×6 행렬 SVD: U: (6,6), Sigma: (6,) , Vt: (6,6) 분해

Truncated SVD n_components=4 설정: U:(6,4), Sigma:(4,), Vt:(4,6) 분해

- Truncated SVD로 분해된 행렬을 복원할 경우, 근사적으로 복원됨

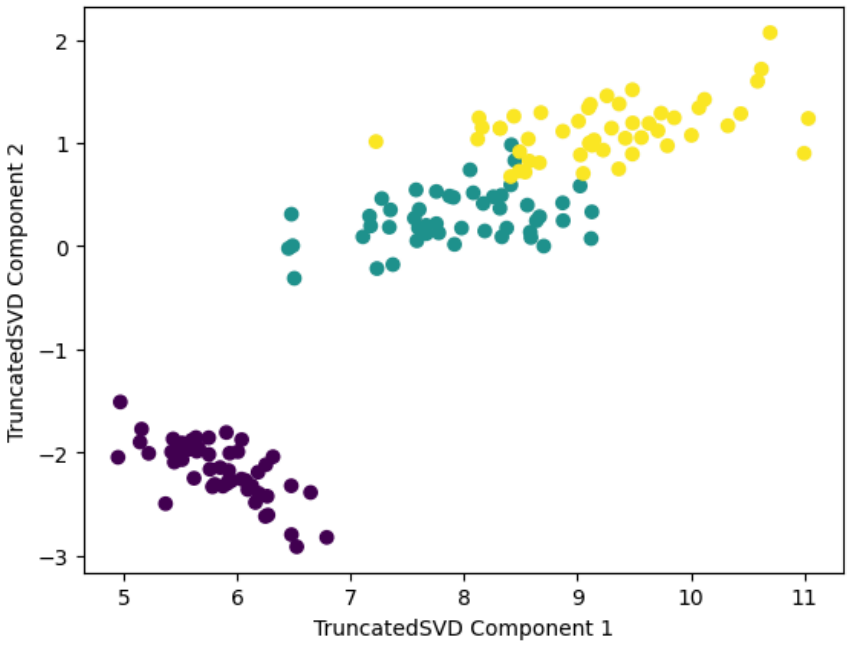
사이킷런 TruncatedSVD 실습

: PCA와 유사하게 fit(), transform() 호출

→ 원본 데이터의 주요 컴포넌트로 차원 축소

붓꽃 데이터 세트 - TruncatedSVD 변환

```
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)
```



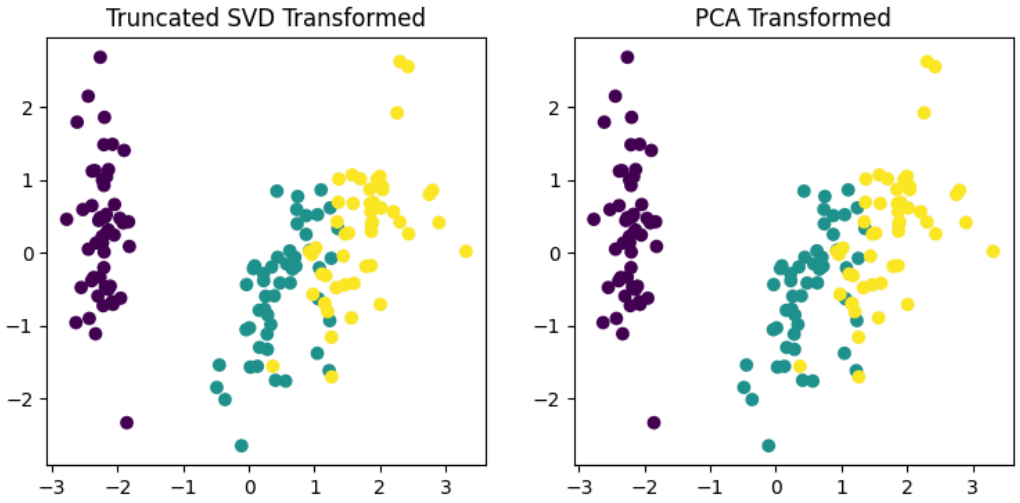
PCA vs TruncatedSVD 비교

```
# 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)
```



- 스케일링 변환 이후 두 변환 실행 ⇒ 결과가 거의 동일함

두 변환 행렬 값과 원본 속성별 컴포넌트 비율값 비교 → 거의 동일!

✅ 데이터 세트가 스케일링으로 중심이 동일해진다면?

사이킷런의 SVD,PCA 동일한 변환 수행 ⇒ PCA가 SVD 알고리즘으로 구현됐다는 뜻!

- PCA: 밀집 행렬 변환만 가능
- SVD: 희소 행렬 변환도 가능

---

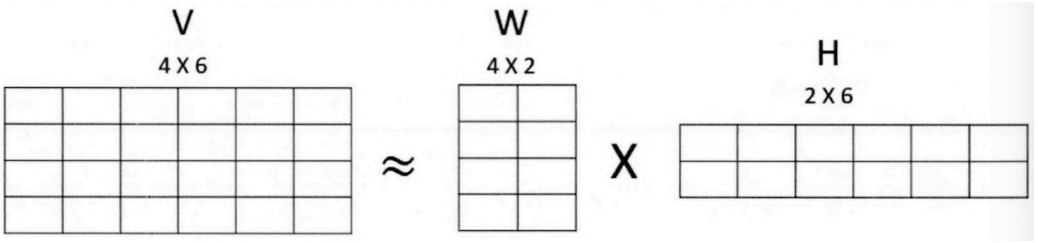
05 NMF(Non-Negative Matrix Factorization)

NMF 개요

NMF

: 낮은 랭크를 통한 행렬 근사 방식의 변형

- 원본 행렬 내 모든 원소 값이 모두 양수라는 게 보장 될 경우 ⇒ ⇒ 두개의 기반 양수 행렬로 분해



길고 가는 행렬 W와 작고 넓은 행렬 H로 분해됨 ⇒ **잠재 요소를 가짐**

- W: 원본 행에 대해 잠재 요소의 값이 얼마나 되는가
- H: 이 잠재 요소가 원본 열로 어떻게 구성됐는가