06 차원 축소

01 차원 축소 개요

차원 축소: 매우 많은 피처로 구성된 다차원 데이터 세트의 차원을 축소해 새로운 차원의 데이터를 생성하는 것

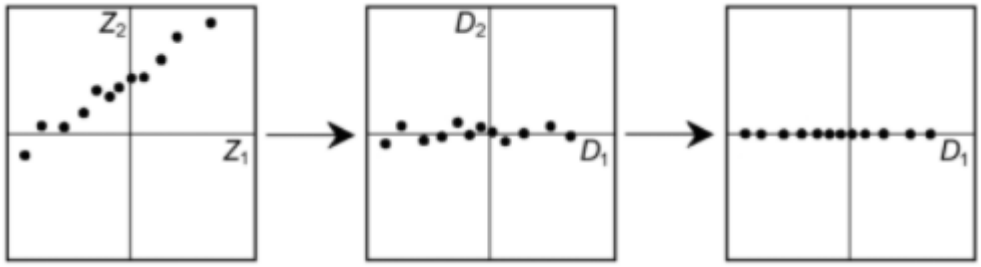
- 피처 선택 : 특정 피처에 종속성이 강한 불필요한 피처는 아예 제거하고, 데이터의 특징을 잘 나타내는 주요 피처만 선택하는 것
- 피처 추출 : 기존 피처를 저차원의 중요 피처로 압축해서 추출하는 것

  → 이렇게 새롭게 추출된 중요 특성은 기존의 피처가 압축된 것이므로 기존의 피처와는 완전히 다른 값이 된다.

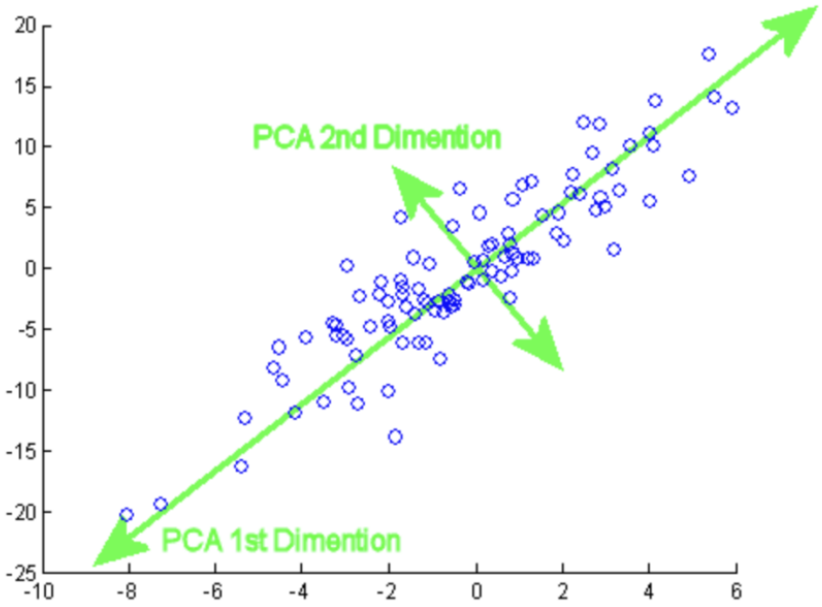
차원 축소 알고리즘

1. PCA(Principal Component Analysis)

- 분산이 데이터의 특성을 가장 잘 나타내는 것으로 간주하여, 데이터 변동성이 가장 큰 방향으로 축을 생성하고, 새롭게 생성된 축으로 데이터를 투영하는 방식.



Principal Component Analysis



첫 번째 벡터 축 : 가장 큰 데이터 변동성을 기반으로 생성

두 번째 벡터 축 : 첫 번째 벡터 축에 직각이 되는 벡터(직교벡터)

세 번째 벡터 축 : 두 번째 벡터 축과 직각이 되는 벡터를 설정하는 방식으로 축 생성

선형 대수 관점에서 바라본 PCA

입력 데이터의 공분산 행렬을 고유값(eigenvalue)으로 분해하고, 이렇게 구한 고유벡터에 입력 데이터를 선형 변환하는 것.

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

C=공분산행렬, P=n*n 직교행렬, ∑=n*n 정방행렬

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

e=고유벡터(Ax=ax,행렬A를 곱하더라도 방향이 변하지 않고 그 크기만 변하는 벡터), ≻=고유벡터의 크기

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

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

[PCA step]

1.입력 데이터 세트의 공분산 행렬을 생성합니다.

2.공분산 행렬의 고유벡터와 고유값을 계산합니다.

3.고유값이 가장 큰 순으로 k개(PCA 변환 차수)만큼 고유벡터를 추출합니다,

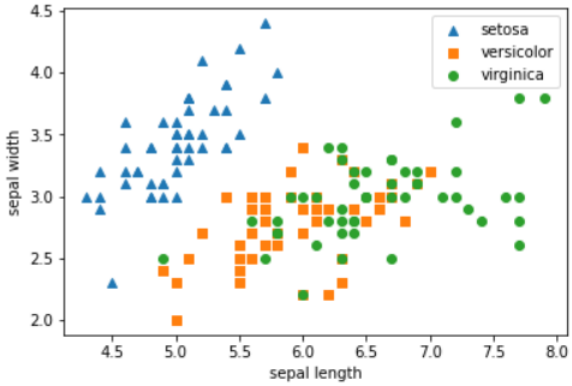
4.고유값이 가장 큰 순으로 추출된 고유벡터를 이용해 새롭게 입력 데이터를 반환합니다.

- 예제

```
#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

iris_scaled = StandardScaler().fit_transform(irisDF)
```

```
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)
```

- PCA는 속성의 스케일에 영향을 받기 때문에 PCA로 압축하기 전에 개별 속성을 함께 스케일링해야한다.

- 사이킷런의 `StandardScaler`를 이용해 평균이 0, 분산이 1인 표준 정규 분포로 모든 속성 값을 변환한다.
- PCA 클래스는 생성 파라미터로 `n_components`(PCA로 변환할 차원의 수)를 입력받는다.

```
# 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)
```

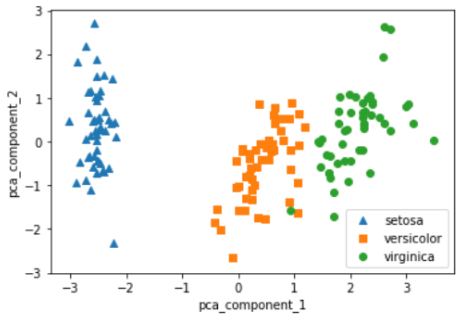
	pca_component_1	pca_component_2	target
0	-2.576198	0.498650	0
1	-2.419715	-0.660518	0
2	-2.662302	-0.326611	0

- >넘파이 행렬을 dataframe으로 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()
```



- >PCA 변환된 데이터 세트를 2차원상에서 시각화

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

```
[0.76590853 0.18427757]
```

- >PCA 객체의 explained_variance_ratio_ 속성은 전체 변동성에서 개별 PCA 컴포넌트별로 차지하는 변동성 비율을 제공, PCA를 2개 요소로만 변환해도 원본 데이터의 변동성을 95% 설명할 수 있다.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score

rcf = RandomForestClassifier(random_state=156)
scores = cross_val_score(rcf, iris.data, iris.target,scoring='accuracy',cv=3)
print(scores)
```

```
[0.98039216 0.92156863 0.97916667]
```

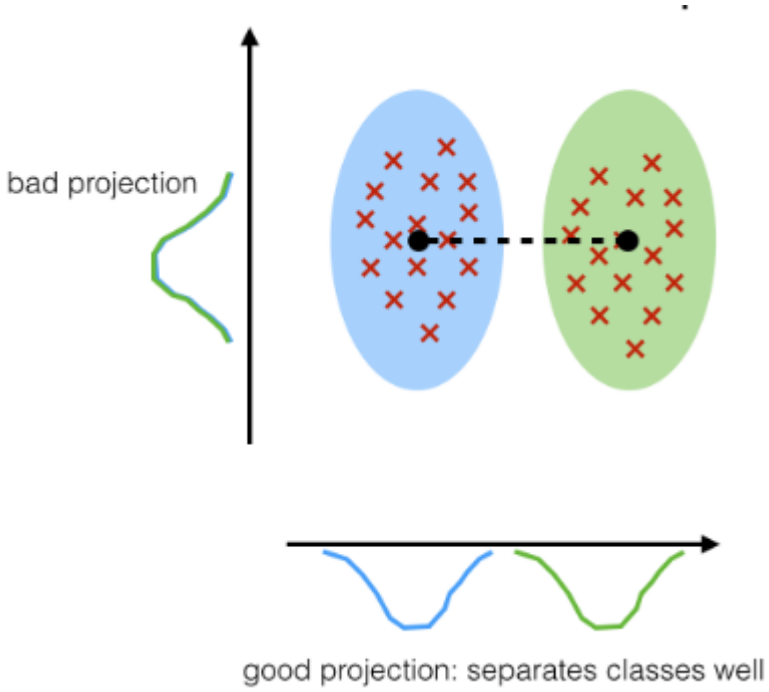
```
pca_X = irisDF_pca[['pca_component_1', 'pca_component_2']]
scores_pca = cross_val_score(rcf, pca_X, iris.target, scoring='accuracy', cv=3 )
print(scores_pca)
```

```
[0.98039216 0.98039216 0.97916667]
```

- >Estimator는 RandomForestClassifier를 이용하고 cross_val_score()로 3개의 교차 검증 세트로 정확도 결과 비교
- >4개의 속성이 2개의 변환 속성이 돼도 예측 성능에 전혀 영향을 받지 않을 정도로 PCA 변환이 잘 적용됐음을 의미
- >대부분은 PCA 변환 차원 개수에 따라 예측 성능이 떨어질 수 밖에 없다.

2. LDA(Linear Discriminant Analysis)

- 선형 판별 분석법. PCA와 매우 유사한 방식
- **지도학습의 분류**에서 **클래스 간 분산**과 **클래스 내부 분산**의 비율을 최대화하는 방식으로 차원을 축소한다.



[LDA step]

1.클래스 내부와 클래스 간 분산 행렬을 구한다. 이 두 개의 행렬은 입력 데이터의 결정 값 클래스별로 개별 피처의 평균 벡터(mean vector)를 기반으로 구한다.

2.클래스 내부 분산 행렬을 Sw, 클래스 간 분산 행렬을 Sb라고 하면 두 행렬을 고유벡터로 분해할 수 있다.

3.고유값이 가장 큰 순으로 k개(LDA변환 차수만큼) 추출한다.

4.고유값이 가장 큰 순으로 추출된 고유벡터를 이용해 새롭게 입력 데이터를 반환한다.

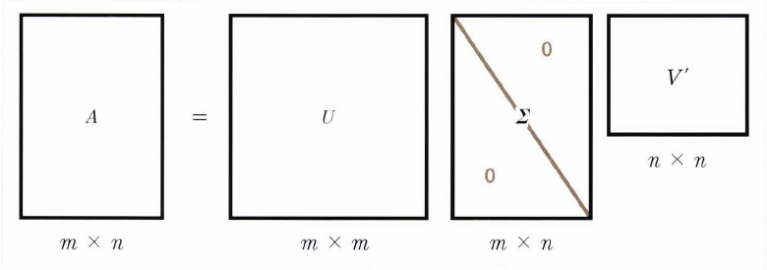
$$S_W^T S_B = \begin{bmatrix} e_1 & \cdots & e_n \end{bmatrix} \begin{bmatrix} \lambda_1 & \cdots & 0 \\ \cdots & \cdots & \cdots \\ 0 & \cdots & \lambda_n \end{bmatrix} \begin{bmatrix} e_1^T \\ \cdots \\ e_n^T \end{bmatrix}$$

3. SVD(Singular Value Decomposition, 특이값 분해)

- 정방행렬만을 고유벡터로 분해할 수 있는 PCA와 달리, SVD는 행과 열의 크기가 다른 행렬에도 적용할 수 있다.
- 사이파이의 SVD는 scipy.linalg.svd 이용

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

U,V=특이 벡터, 서로 직교하는 성질



- 예제

```
# numpy의 svd 모듈 import
import numpy as np
from numpy.linalg import svd

# 4X4 Random 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]]
```

- >랜덤 행렬을 생성하는 이유는 행렬의 개별 로우끼리의 의존성을 없애기 위함

```
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]]
```

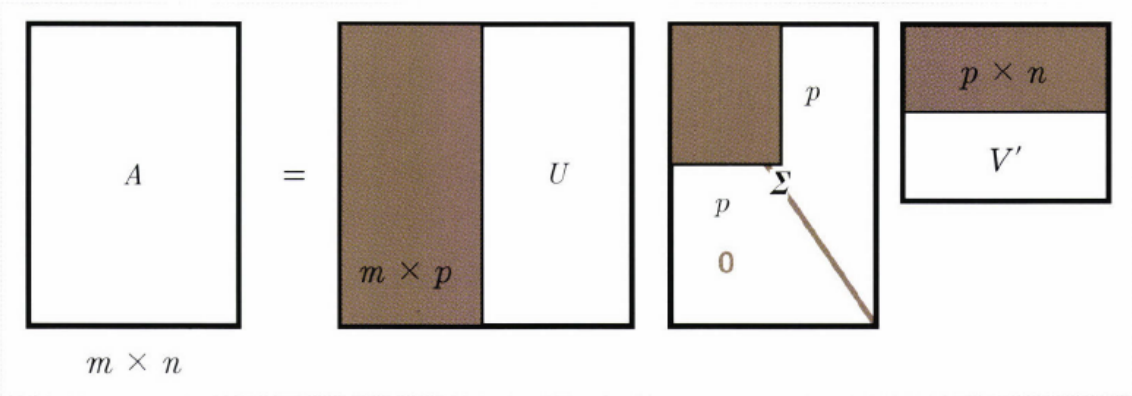
```
# Sima를 다시 0 을 포함한 대칭행렬로 변환
Sigma_mat = np.diag(Sigma)
a_ = np.dot(np.dot(U, Sigma_mat), Vt)
print(np.round(a_, 3))
```

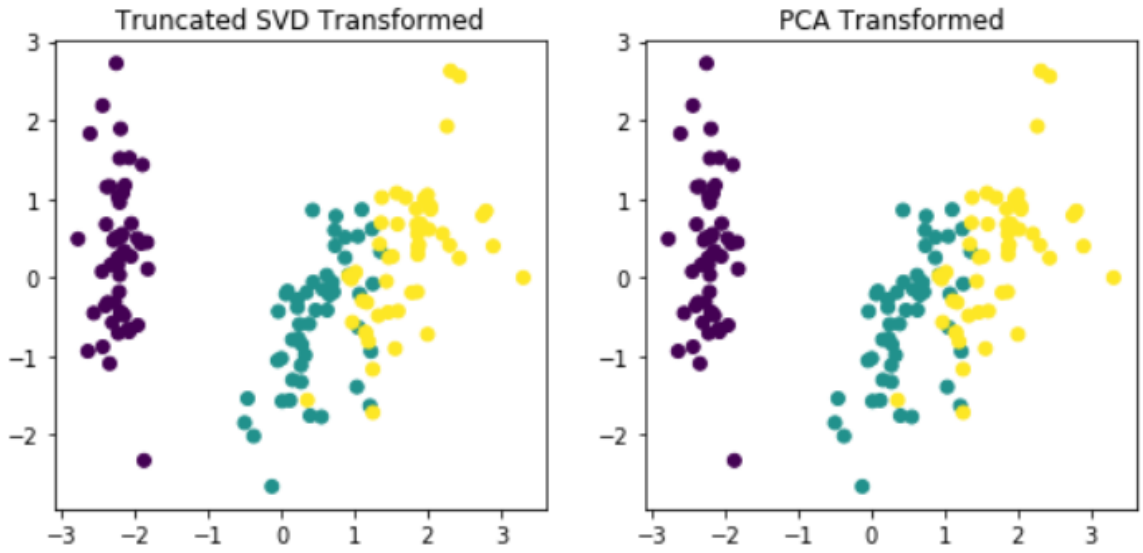
```
[[-0.212 -0.285 -0.574 -0.44 ]
 [-0.33   1.184  1.615  0.367]
 [-0.014  0.63   1.71  -1.327]
 [ 0.402 -0.191  1.404 -1.969]]
```

- > 생성된 a행렬에 U, Sigma, Vt를 도출. Sigma 행렬의 경우 행렬의 대각에 위치한 값만 0이 아니고, 그렇지 않은 경우는 모두 0이므로 0이 아닌 값의 경우만 1차원 행렬로 표현한다. 분해된 U, Sigma, Vt를 이용해 다시 원본 행렬로 복원. Sigma는 다시 0을 포함한 대칭행렬로 먼저 변환한 뒤에 내적을 수행해야 한다.
- Truncated SVD

: ∑의 대각원소 중에 상위 몇 개만 추출해서 여기에 대응하는 U와 V의 원소도 함께 제거해 더욱 차원을 줄인 형태로 분해하는 것

: 넘파이가 아닌 사이파이에서만 지원. 사이파이의 truncated svd는 희소행렬로만 지원돼서 scipy.sparse.linalg.svds를 이용





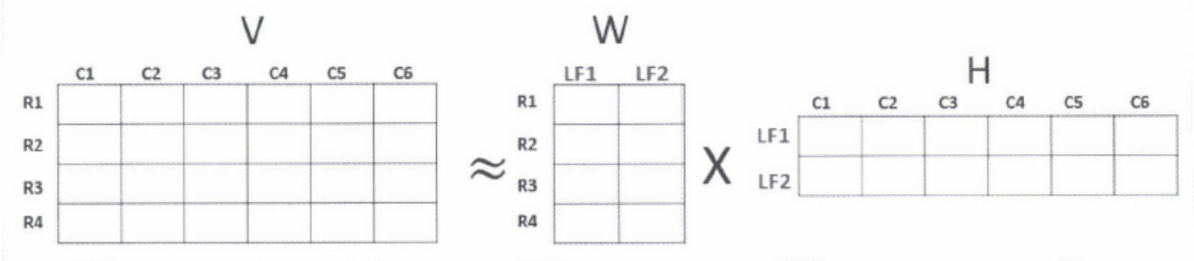
- >붓꽃 데이터를 스케일링으로 변환한 뒤에 사이킷런 TruncatedSVD클래스, PCA클래스 변환. 거의 동일한 결과
- 데이터 세트가 스케일링으로 데이터 중심이 동일해지면 사이킷런의 SVD와 PCA는 동일한 변환을 수행

SVD와 PCA는 동일한 변환을 수행

PCA는 밀집행렬(Dense Matrix)에 대한 변환만 가능하며 SVD는 희소 행렬(Sparse Matrix)에 대한 변환도 가능

4. NMF(Non-Negative Matrix Factorization)

- Truncated SVD와 같이 낮은 랭크를 통한 행렬 근사(Low-Rank Approximation)방식의 변형
- 원본 행렬 내의 모든 원소 값이 모두 양수(0이상)라는 게 보장되면, 좀 더 간단하게 두 개의 기반 양수 행렬로 분해될 수 있는 기법을 지칭한다.



- 분해 행렬 W는 원본 행에 대해서 이 잠재 요소의 값이 얼마나 되는지에 대응하며, 분해 행렬 H는 이 잠재 요소가 원본 열(즉, 원본 속성)로 어떻게 구성됐는지를 나타내는 행렬
- 사이킷런에서 NMF는 NMF 클래스를 이용해 지원된다.