06
차원 축소

차원 축소(Demension Reduction)

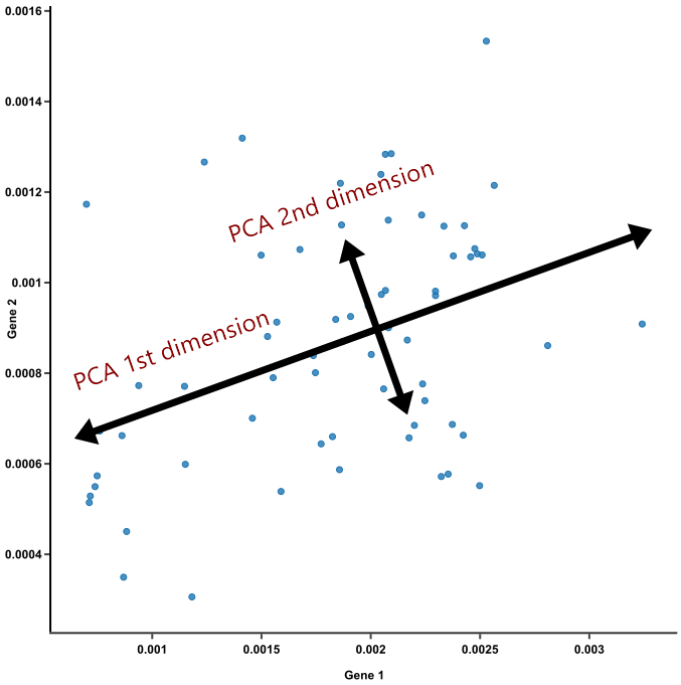
차원 축소는 매우 많은 feature 로 구성된 다차원 데이터셋의 차원을 축소해 새로운 차원의 데이터셋을 생성하는 것입니다. 일반적으로 차원이 증가할수록 데이터 포인트 간의 거리가 기하급수적으로 멀어지게 되고, 희소(sparse)한 구조를 가지게 됩니다. 따라서 수백 개 이상의 feature 로 구성된 데이터셋의 경우 상대적으로 예측 신뢰도가 떨어지며 개별 feature 간의 상관관계가 높을 가능성이 큽니다. 선형회귀와 같은 선형 모델에서는 입력 변수 간의 상관관계가 높을 경우 이로 인한 다중 공선성의 문제로 모델의 예측 성능이 저하됩니다.

일반적으로 차원 축소는 feature 선택과 feature 추출로 나눌 수 있습니다. feature 선택은 데이터의 특징을 잘 나타내는 주요 feature 만 선택하는 것을, feature 추출은 기존 feature 를 저차원의 중요 feature 로 압축해서 추출하는 것입니다. 이렇게 새롭게 추출된 주요 feature 는 기존의 feature 가 압축된 것이므로 기존과 완전히 다른 값이 됩니다.

차원 축소 알고리즘은 매우 많은 픽셀로 이뤄진 이미지 데이터에서 잠재된 특성을 feature 로 도출해 변환된 이미지는 원본보다 훨씬 적은 차원이기 때문에 이미지 분류 등의 분류 수행 시 overfitting 영향력이 작아져 예측 성능을 끌어올릴 수 있습니다. 또한, 텍스트문서의 숨겨진 의미를 추출하는 데도 차원 축소 알고리즘이 자주 사용됩니다. 문서 내 단어들의 구성에서 숨겨져 있는 semantic 의미나 topic 을 잠재 요소로 간주하고 이를 찾아낼 수 있으며 SVD 와 NMF 는 이러한 semantic topic 모델링을 위한 기반 알고리즘으로 사용됩니다.

PCA

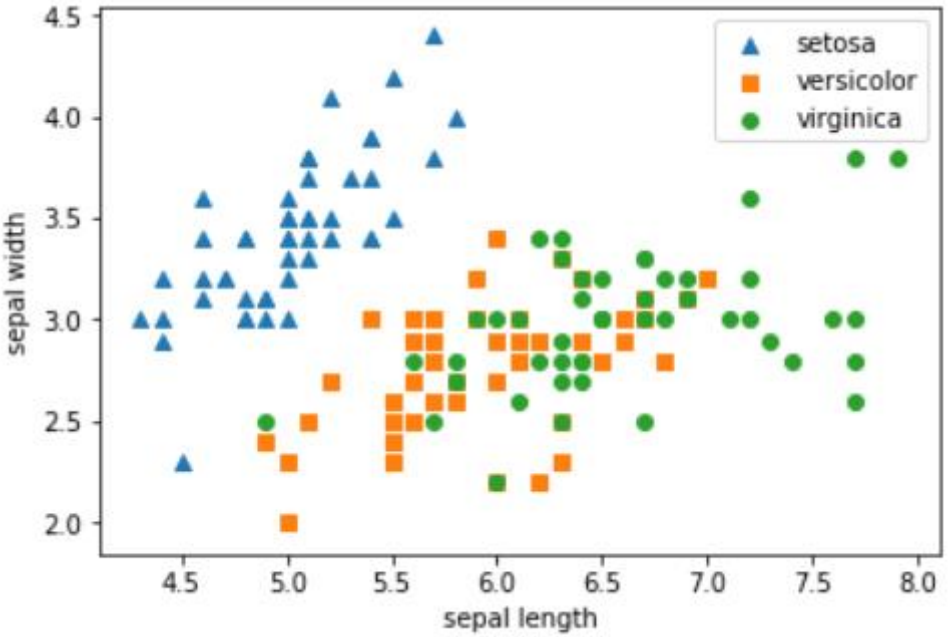
PCA 는 여러 변수 간에 존재하는 상관관계를 이용해 이를 대표하는 주성분(Principal Component)을 추출해 차원을 축소하는 기법입니다. PCA 는 가장 높은 분산을 가지는 데이터 축을 찾아 이 축으로 차원을 축소하는데, 이것이 PCA 의 주성분이 됩니다. PCA 는 제일 먼저 가장 큰 데이터 변동성(variance)을 기반으로 첫 번째 벡터 축을 생성하고, 두 번째 벡터 축은 이 벡터 축에 직각이 되는 벡터를 축으로 합니다. 세 번째 축은 다시 두 번재 축과 직각이 되는 벡터를 생성하는 방식으로 축을 생성합니다. 이렇게 생성된 벡터 축에 원본 데이터를 투영하면 벡터 축의 개수만큼의 차원으로 원본 데이터가 차원 축소됩니다.



PCA 를 선형대수 관점에서 보면 입력 데이터의 공분산 행렬을 고유값 분해하고, 이렇게 구한 고유 벡터에 입력 데이터를 선형 변환하는 것입니다. 이 고유 벡터가 PCA 의 주성분 벡터로서 입력 데이터의 분산이 큰 방향을 나타냅니다. 고유값은 고유 벡터의 크기를 나타내며, 동시에 입력 데이터의 분산을 나타냅니다. 따라서 보통 PCA 는 다음과 같은 과정으로 수행됩니다.

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

이제부터 붓꽃데이터셋을 PCA 로 차원축소를 진행해보도록 하겠습니다. 붓꽃데이터셋은 4 개의 속성으로 되어있는데, 이를 2 개의 PCA 차원으로 압축해 어떻게 달라지는지를 확인해보겠습니다. 4 개 중 2 개의 속성인 sepal length 와 sepal width 를 x,y 축으로 하여 품종 데이터 분포를 나타내면 아래와 같습니다.



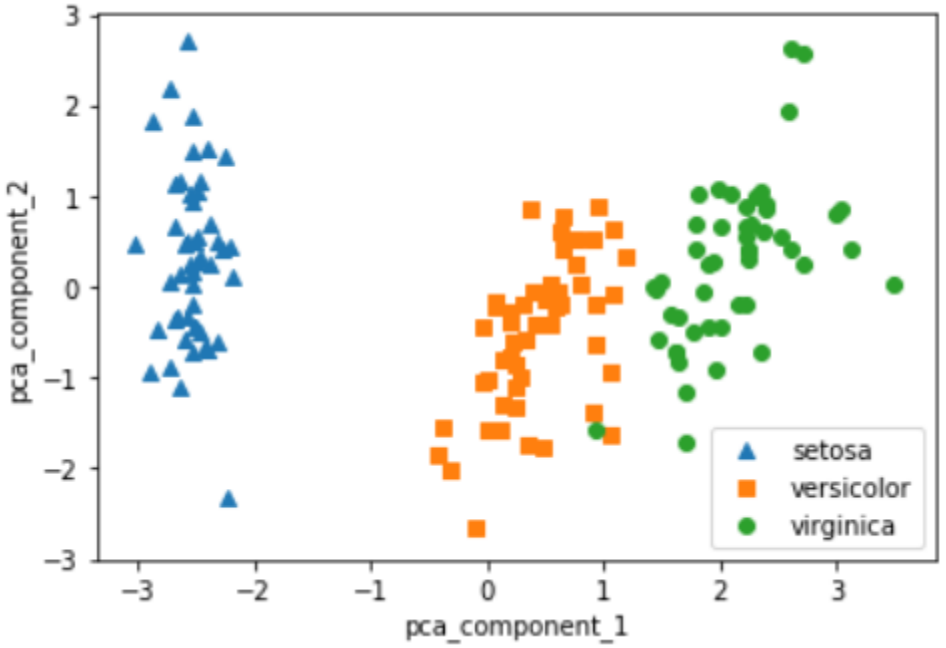
이제 PCA 로 4 개의 속성을 2 개로 압축한 뒤 앞의 예제와 비슷하게 2 개의 PCA 속성으로 붓꽃데이터셋의 분포를 2 차원 시각화해보겠습니다. 먼저 바로 PCA 를 적용하기 전에 각 속성값을 동일한 스케일로 변환하는 것이 필요합니다. scikit-learn 의 StandardScaler 를 이용해 표준 정규 분포로 모든 속성값을 변환합니다.

```
from sklearn.preprocessing import StandardScaler

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

스케일링이 적용된 데이터셋에 PCA 를 적용해보겠습니다. scikit-learn 은 PCA 변환을 위해 PCA 클래스를 제공합니다. PCA 클래스는 생성 parameter 로 n_components 를 입력받습니다. n_components는 PCA로 변환할 차원의 수를 의미하므로 여기서는 2로 설정하겠습니다. 이후에 fit 과 transform 을 호출해 PCA 로 변환을 수행합니다.

```
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)
```

이렇게 PCA 변환된 데이터셋을 pca_component_1 을 x 축으로, pca_component_2 를 y 축으로 해서 데이터셋 분포를 확인해보도록 하겠습니다.



PCA 변환을 수행한 객체의 explained_variance_ration_속성은 전체에서 개별 PCA 컴포넌트별로 차지하는 변동성의 비율을 제공하고 있습니다.

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

```
[0.76590853 0.18427757]
```

첫번째 PCA 변환 요소인 pca_component_1 이 전체 변동성의 약 76.5%를 차지하며, 두번째인 pca_component_2 가 약 18.4%를 차지합니다. 따라서 PCA 를 2 개 요소로만 변환해도 원본 데이터 변동성의 95%를 설명할 수 있습니다. 이번에는 원본 데이터셋과 PCA 로 변환된 데이터셋에 각각 분류를 적용한 뒤 결과를 비교해보겠습니다. Estimator 로

RandomForestClassifier 를 이용하고, cross_val_score( )로 3 개의 교차 검증 세트로 정확도를 비교하겠습니다.

```
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]
```

아래는 PCA 변환 데이터셋입니다.

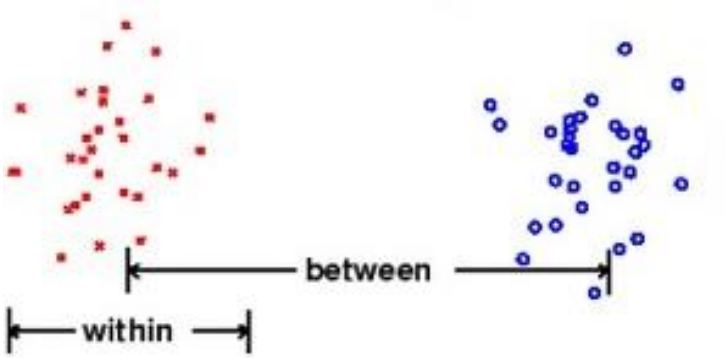
```
pca_X = irisDF_pca[['pca_component_1', 'pca_component_2']]
scores_pca = cross_val_score(rcf, pca_X, iris.target, scoring='accuracy', c
v=3 )
print(scores_pca)
```

```
[0.98039216 0.98039216 0.97916667]
```

결과적으로 원본 데이터와 동일하거나 일부는 더 나은 정확도를 나타내고 있습니다. 이렇게 PCA 변환 데이터셋이 원본보다 더 나은 예측 정확도를 보이는 경우는 흔치 않습니다. 대부분 PCA 변환 차원의 개수에 따라 예측 성능이 떨어질 수 밖에 없기 때문입니다.

LDA

LDA 는 선형 판별 분석법으로 불리며, PCA 와 매우 유사합니다. 다만 LDA 는 지도학습의 classification 에서 사용되기 쉽도록 개별 클래스를 분별할 수 있는 기준을 최대한 유지하면서 차원을 축소한다는 점이 다른 점입니다. PCA 는 입력 데이터의 변동성이 가장 큰 축을 찾았지만, LDA 는 입력 데이터의 결정 값 클래스를 최대한으로 분리할 수 있는 축을 찾습니다. 즉, LDA 는 클래스 간 분산(between-class scatter)과 클래스 내부 분산(within-class scatter)의 비율을 최대화하는 방식으로 차원을 축소합니다. 클래스 간 분산을 최대한 크게 가져가고, 클래스 내부 분산은 최대한 작게 가져가는 방식입니다.



일반적으로 LDA 를 구하는 스텝은 아래와 같습니다.

1) 클래스 내부와 클래스 간 분산 행렬을 구합니다. 이 두개의 행렬은 입력 데이터의 결정값 클래스 별로 개별 feature 의 평균 벡터를 기반으로 구합니다.

2) 다음 식으로 두 행렬을 고유벡터로 분해합니다.

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

3) 고유값이 가장 큰 순으로 LDA 변환 차수만큼 k 개 추출합니다.

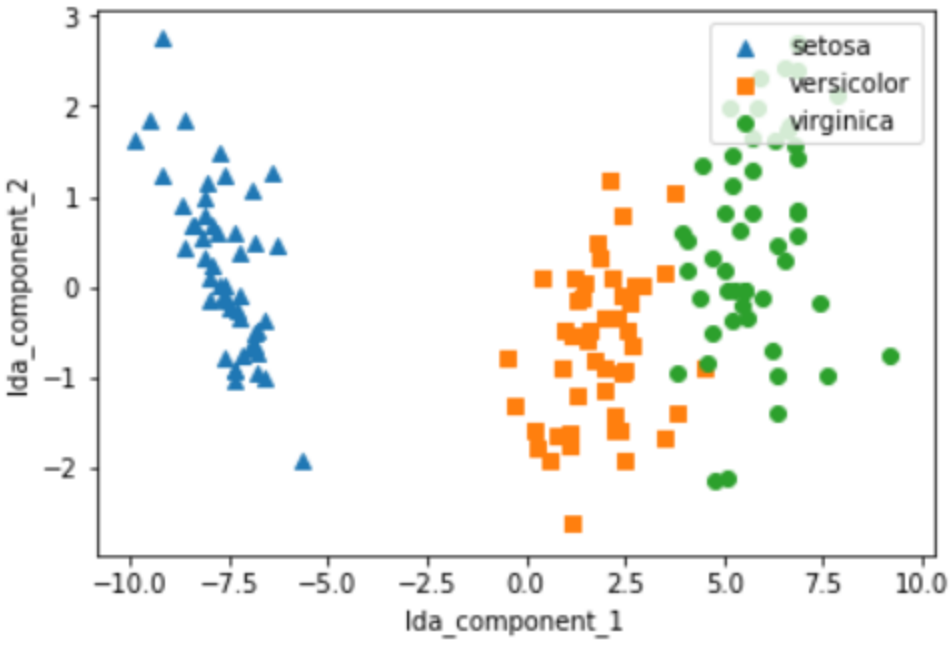
4) 고유값이 가장 큰 순으로 추출된 고유벡터를 이용해 새롭게 입력 데이터를 변환합니다.

붓꽃데이터셋을 scikit-learn 의 LDA 를 이용해 변환하고 품종별로 시각화해보겠습니다. scikit-learn 은 LDA 를 LinearDiscriminantAnalysis 클래스로 제공합니다. 유의해야할 점은 LDA 는 PCA 와 다르게 지도학습이라는 것입니다. 즉, 클래스의 결정값이 변환 시 필요합니다.

```
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)
```

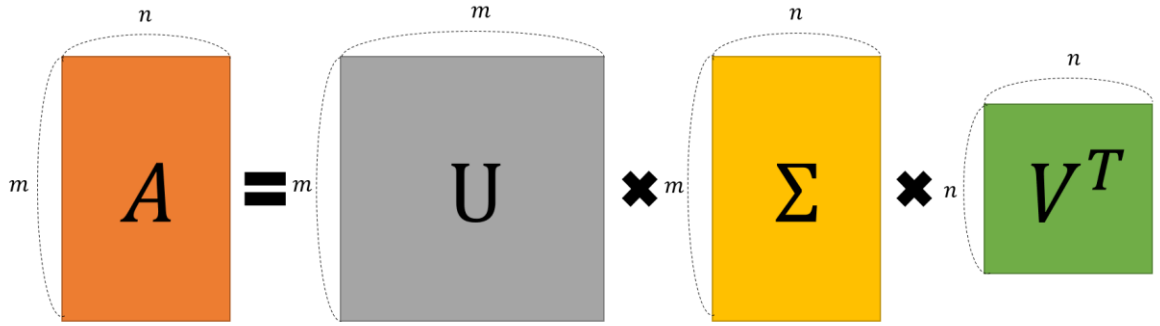
```
lda = LinearDiscriminantAnalysis(n_components=2)
lda.fit(iris_scaled, iris.target)
iris_lda = lda.transform(iris_scaled)
```

아래는 LDA 변환된 데이터를 시각화한 자료입니다.



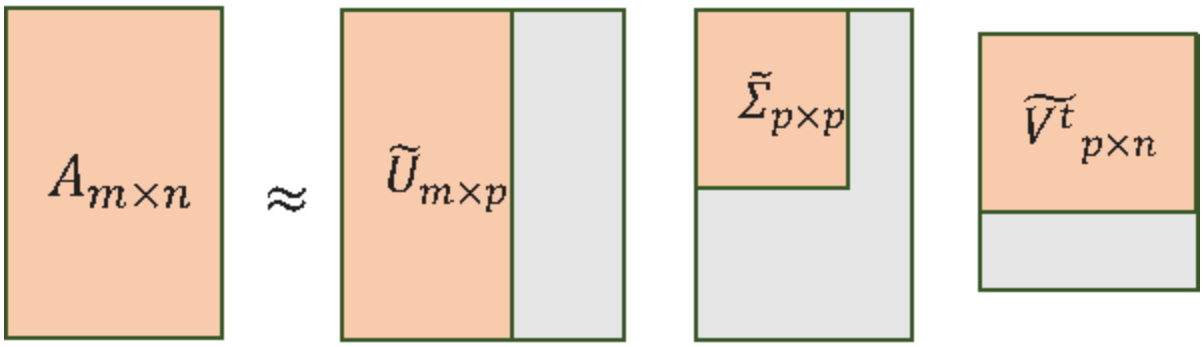
SVD

SVD 역시 PCA 와 유사한 행렬 분해 기법을 이용하나, PCA 의 경우 정방행렬 만을 고유벡터로 분해할 수 있지만, SVD 는 정방행렬 뿐만 아니라 행과 열의 크기가 다른 행렬에도 적용할 수 있습니다.



SVD 는 특이값 분해로 불리며 행렬 U 와 V 에 속한 벡터는 특이벡터(singular vector)이며, 모든 특이벡터는 서로 직교하는 성질을 가집니다. Σ는 대각행렬이며, 행렬의 대각에 위치한 값만 0 이 아니고 나머지 값은 모두 0 입니다. Σ에 위치한 0 이 아닌 값이 바로 행렬 A 의 특이값입니다. SVD 는 A 의 차원이 mxn 일 때, U 의 차원이 mxm, Σ의 차원이 mxn, V_T 의 차원이 nxn 으로 분해합니다.

하지만 일반적으로 다음과 같이 Σ의 비대각인 부분과 대각원소 중에서도 특이값이 0 인 부분도 모두 제거하고, 제거된 Σ에 대응되는 U, V 원소도 함께 제거해 차원을 줄인 형태로 SVD 를 적용합니다. 여기서 Σ의 대각원소 중 상위 몇 개만 추출하여 차원을 더욱 축소한 컴팩트한 SVD 를 Truncated SVD 라고 부릅니다.



SVD 를 구현하기 위해서 numpy.linalg.svd 혹은 scipy.linalg.svd 를 사용할 수 있습니다. 해당 모듈을 로딩 후, 랜덤한 4*4 numpy array 를 생성합니다. 여기서 랜덤행렬을 생성하는 이유는 row 간의 의존성을 없애기 위함입니다.

```
# 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]]
```

이렇게 생성된 a 행렬에 SVD 를 적용해 U, Sigma, Vt 를 도출하겠습니다. numpy.linalg.svd 에 parameter 로 원본 행렬을 입력하면 U, Sigma, Vt 를 반환합니다. Sigma 의 경우 행렬의 대각에 위치한 값만 0이 아니고, 나머지는 모두 0이므로 아닌 값만 1차원의 행렬로 표현할 수 있습니다.

```
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]]
```

U는 4*4, Vt는 4*4, Signma의 경우 1차원 행렬인 (4,)로 반환되었습니다. 분해된 성분들이 다시 원본 행렬로 정확히 복원되는지 확인해보겠습니다. 한 가지 유의할 것은 Sigma 의 경우 0 이

아닌 값만 1 차원으로 추출했기 때문에 0 을 포함한 대칭행렬로 변환한 뒤에 내적을 수행해야한다는 점입니다.

```
# 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]]
```

이를 통해, 복원된 a_는 원본 행렬 a 와 동일함을 확인할 수 있습니다.

이번에는 데이터셋 row 간 의존성이 있을 경우를 살펴보겠습니다. 일부러 의존성을 부여하기 위해 a 의 세 번째 row 를 '첫 번째 row+두 번째 row'로 업데이트하고, 네 번째 row 는 첫 번째 row 와 같다고 두겠습니다.

```
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 ]]
```

이제 이 데이터를 SVD 로 분해해보겠습니다.

```
# 다시 SVD를 수행하여 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 2 개가 0 으로 변했습니다. 즉, 선형 독립인 row vector 개수가 2 개라는 의미입니다.(즉, 행렬의 rank 가 2 입니다.)

이번에는 U, Sigma, Vt 의 전체 데이터를 이용하지 않고 Sigma 의 0 에 대응되는 U, V 원소를 제외하고 복원해보겠습니다. 즉 Sigma 의 경우 앞의 2 개 요소만 0 이 아니므로 U 중 선행 두 개의 열만 추출하고 Vt 는 선행 두 개의 행만 추출해 복원하는 것입니다.

```
# 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 ]]
```

이번에는 Truncated SVD 를 이용해 행렬을 분해해보겠습니다. Truncated SVD 는 Σ행렬의 특이값 중 상위 일부 데이터만 추출해 분해하는 방식입니다. 이렇게 분해하면 인위적으로 더 작은 차원의 U, Σ, V_T 로 분해하기 때문에 원본 행렬을 정확하게 다시 복원할 수는 없습니다.

Truncated SVD 는 scipy 의 scipy.sparse.linalg.svds 또는 scikit-learn 의 TruncatedSVD 클래스를 통해 구현할 수 있습니다. 먼저 scipy 를 통해 임의의 원본 행렬 6*6 을 normal SVD 로 분해한 것과 Truncated SVD 로 분해된 것을 비교하고, Truncated SVD 로 분해된 값을 통해 원본 행렬로 복원하는 작업까지 진행해보도록 하겠습니다.

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

# 원본 행렬을 출력하고, SVD를 적용할 경우 U, Sigma, Vt 의 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)
```

```
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 0.034
9925 ]
```

```
# 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.s
hape)
print('₩nTruncated SVD Sigma값 행렬:', Sigma_tr)
matrix_tr = np.dot(np.dot(U_tr,np.diag(Sigma_tr)), Vt_tr)  # output of Trun
catedSVD

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

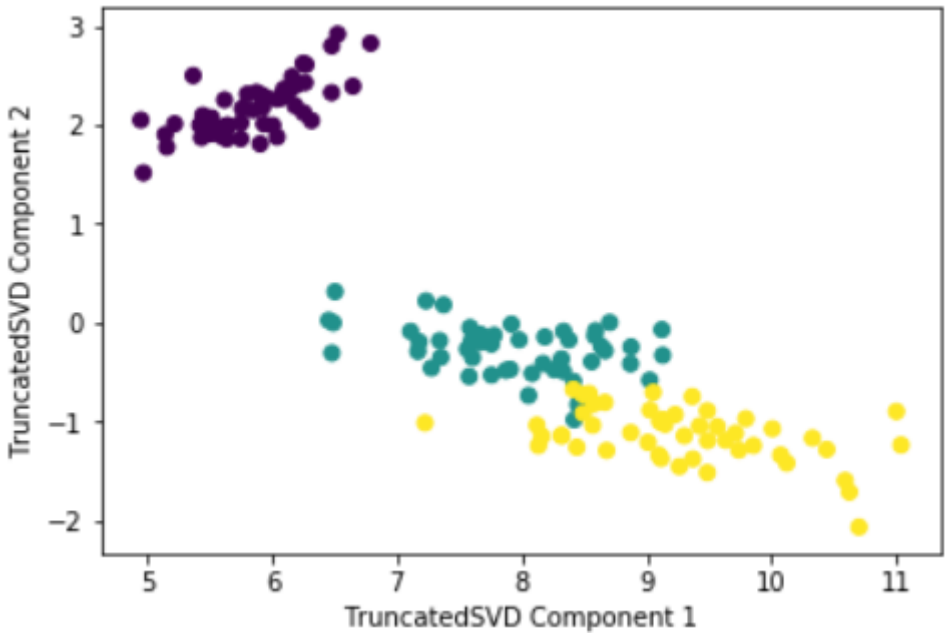
```
Truncated SVD 분해 행렬 차원: (6, 4) (4,) (4, 6)

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]]
```

normal SVD 로 분해하면 U, Sigma, Vt 가 각각 (6,6) (6,) (6,6) 차원이지만, Truncated SVD 의 n_components = 4 로 하면 각각 (6,4) (4,) (4,6)으로 분해했습니다. 원본 행렬로 다시 복원하는 경우 완벽하지는 않고 근사적으로 복원되었습니다.

이번에는 Scikit-lean 의 TruncatedSVD 클래스로 붓꽃 데이터 분해를 진행해보도록 하겠습니다. scikit-lean 의 TruncatedSVD 클래스는 scipy 처럼 원본행렬을 분해한 U, Sigma, Vt 행렬을

반환하지는 않습니다. 대신 fit( )와 transoform( )을 호출해 원본 데이터를 몇 개의 주요 컴포넌트로 차원을 축소해 변환합니다.

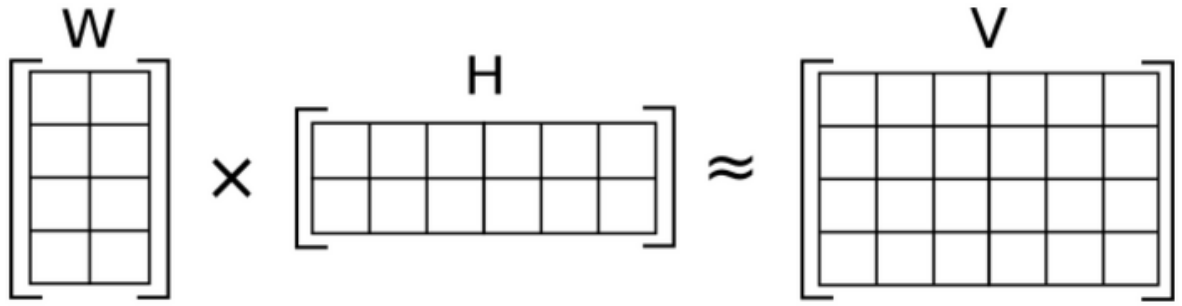
```
from sklearn.decomposition import TruncatedSVD, PCA
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
%matplotlib inline

iris = load_iris()
iris_ftrs = iris.data
# 2개의 주요 component로 TruncatedSVD 변환
tsvd = TruncatedSVD(n_components=2)
tsvd.fit(iris_ftrs)
iris_tsvd = tsvd.transform(iris_ftrs)

# Scatter plot 2차원으로 TruncatedSVD 변환 된 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')
```



SVD 는 PCA 와 유사하게 computer vision 영역에서 이미지 압축을 통한 패턴 인식과 신호 처리 분야에 주로 사용됩니다.

NMF

NMF 는 Truncated SVD 와 같이 낮은 랭크를 통한 행렬 근사(Low-Rank Approximation) 방식의 변형입니다. NMF 는 원본 행렬 내의 모든 원소 값이 양수라는 게 보장되면 다음과 같이 좀 더 간단하게 두 개의 기반 행렬로 분해될 수 있는 기법을 지칭합니다.

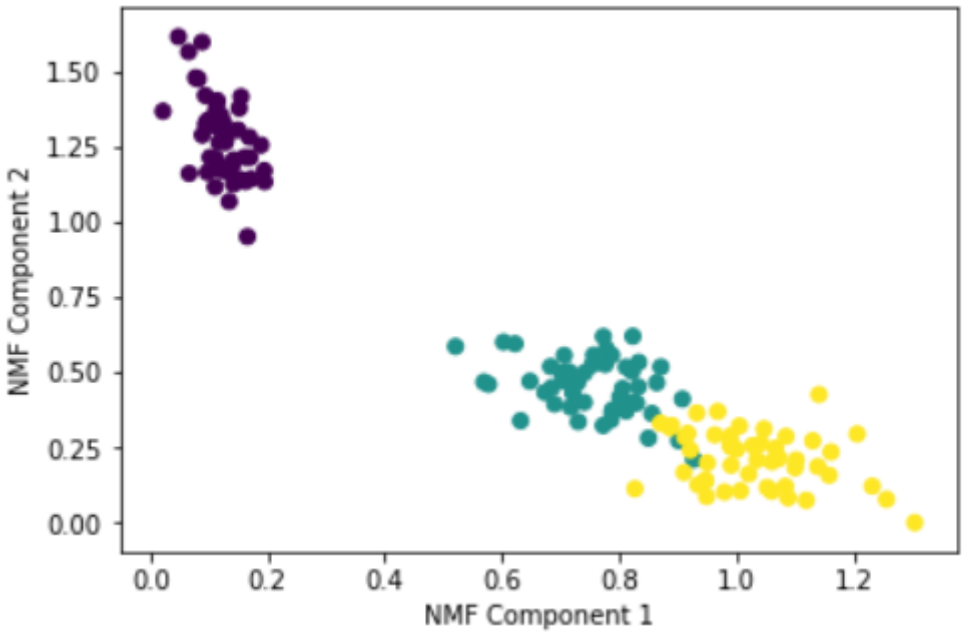


이처럼 행렬 분해를 하게 되면 W 행렬과 H 행렬은 일반적으로 길고 가는 행렬 W(즉, 원본 행렬의 행 크기와 같고 열 크기가 작은 행렬)와 작고 넓은 H(원본 행렬의 행 크기 보다 작고 열 크기가 같은 행렬)로 분해됩니다. 이렇게 분해된 행렬은 잠재 요소를 특성으로 가지게 됩니다. 분해 행렬 W 는 원본 행에 대해서 이 잠재 요소의 값이 얼마나 되는지에 대응하며, 분해 행렬 H 는 이 잠재 요소가 원본 속성으로 어떻게 구성됐는지를 나타내는 행렬입니다.

NMF 는 scikit-lean 의 NMF 클래스를 통해 지원됩니다. 붓꽃 데이터를 NMF 를 이용해 2 개의 컴포넌트로 변환하고 이를 시각화해보겠습니다.

```
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')
```

```
Text(0,0.5,'NMF Component 2')
```



NMF 는 이미지 압축을 통한 패턴인식, 문서 유사도 및 클러스터링에 잘 사용됩니다. 또한 추천(Recommendations) 영역에 활발히 적용됩니다. 사용자의 상품 평가 데이터셋인 사용자-평가 순위(user-Rating) 데이터셋을 행렬 분해 기법을 통해 분해하면서 사용자가 평가하지 않은 상품에 대한 잠재적인 요소를 추출해 추천해주는 방식입니다(이를 잠재요소(Latent Factoring) 기반의 추천 방식이라고 합니다).