차원축소

비타민 7기 3조

고혜영 이준석 이휘정 장효정

INDEX

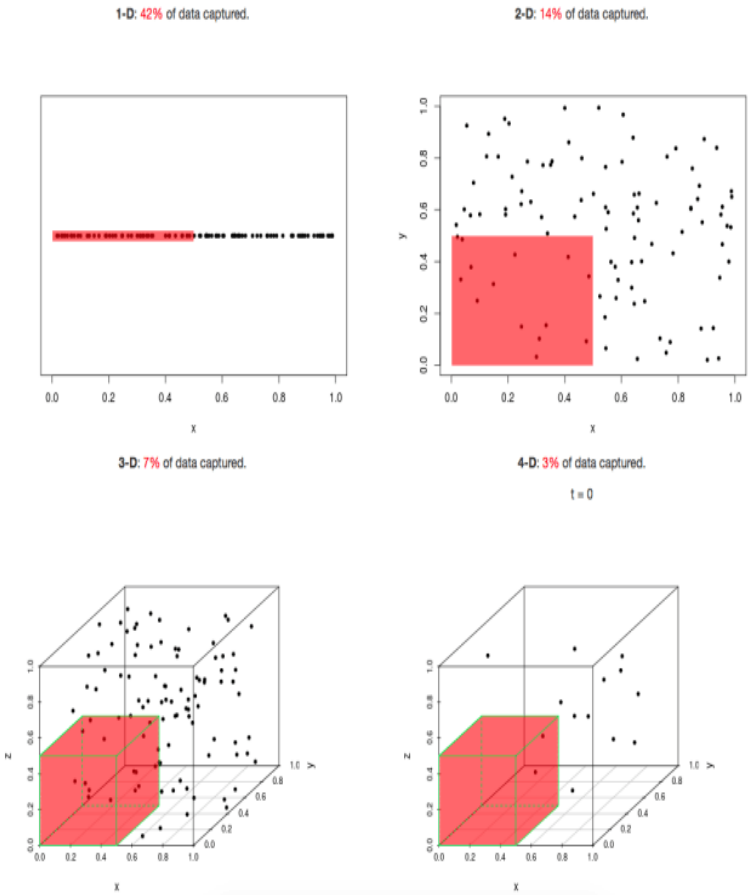
1. 차원 축소의 필요성

차원 축소의 필요성

▪ 차원의 저주

차원이 증가할수록 정보량을 표현하기 위해 필요한 데이터의 수는 지수적으로 증가한다는 의미

- 데이터보다 변수가 많을 때 생기는 현상
- 학습을 느리게 하고 과적합(overfitting)이 발생할 가능성이 높아짐.
- 고차원으로 갈수록 전체 공간에서 데이터가 차지하는 영역이 매우 작아짐
- 입력 변수의 수가 너무 많으면 잡음(noise)이 발생하여 분류 모형의 정확도 감소함.
- 입력 변수 간에 상관관계가 있는 경우 다중 공선성이 발생해 모형이 불안정해짐.



차원 축소의 필요성

차원축소의 효과

- 시각화 : 3차원 이내로 시각화하여 데이터 패턴 인지 용이
- 노이즈 제거 -> 정확도 향상
- 메모리 절약 -> 학습 속도 향상

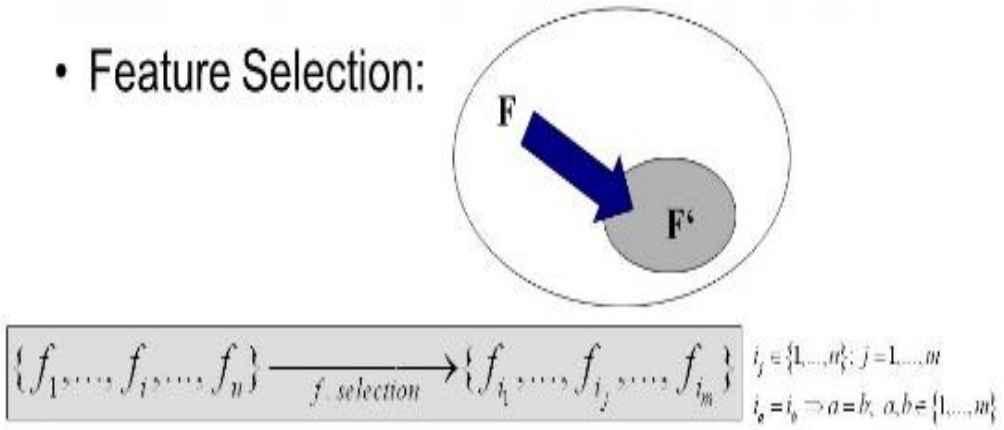
차원축소의 목적

- 모델 성능을 최대로 해주는 변수의 일부 셋을 찾는 것

Feature selection / extraction

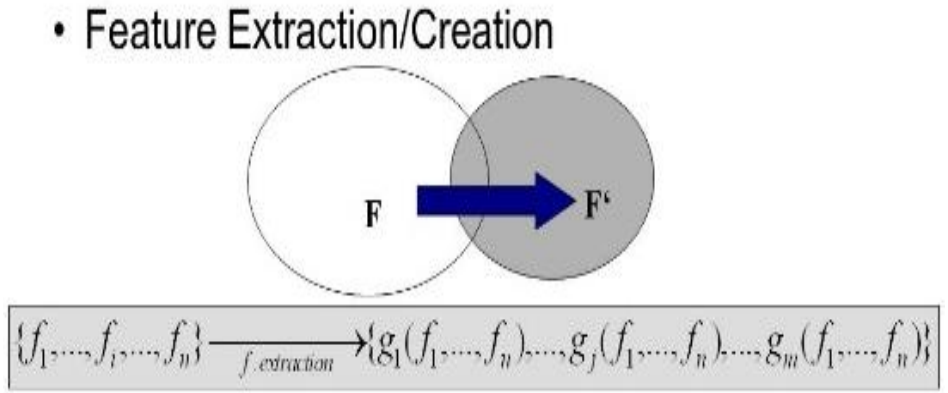
▪ Feature selection

- 중요한 변수를 찾는 과정으로 데이터 속에 존재하는 중복되고 상관없는 변수를 제거하여 데이터를 잘 나타내고 출력 변수와 관련 깊은 입력 변수를 선택하는 과정
- LASSO, Ridge, mRMR, SVM-REF
- 예: $x_1, x_2, \dots, x_{100} \rightarrow x_1, x_5$



▪ Feature extraction

- 기존 변수들의 조합으로 새로운 특징을 생성하는 과정으로 서로 중복되지 않으며 출력 변수에 유의한 특징을 추출함.
- PCA, LDA, SVD, NMF
- 예: $z = f(x_1, x_2, \dots, x_{100})$

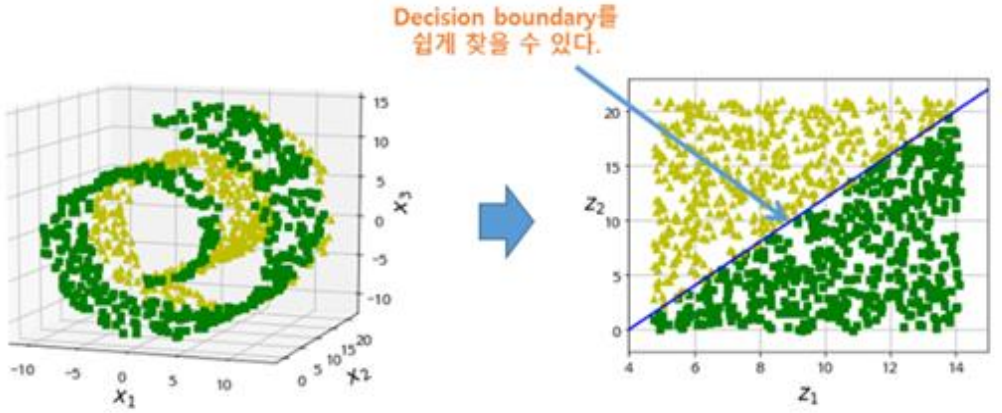


2. PCA

PCA 개요

PCA

- n개의 관측치와 p개의 변수로 구성된 데이터를 상관관계가 없는 k개의 변수로 구성된 데이터(n개의 관측치)로 요약하는 방식
- 요약된 변수는 **기존 변수의 선형조합**으로 생성된다.
- 일반적으로 PCA는 전체 분석 과정 중 초기에 사용

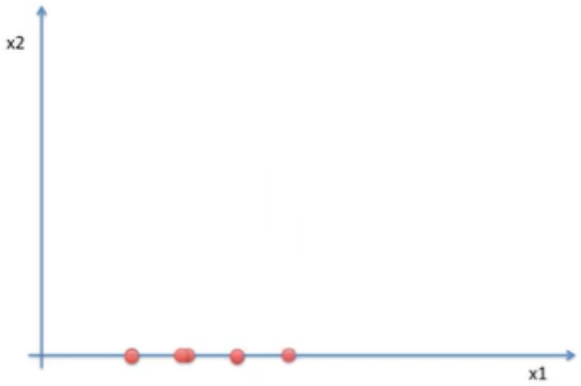


PCA 개요

- PCA 원리

다음과 같은 2차원 데이터를 1차원으로 축소하기 위한 축을 어떻게 그리면 좋을까?

➢ x1축으로 내리기



➢ x2축으로 내리기

x1과 x2축에 모든 데이터를 몰아넣어보면
겹치는 부분이 생긴다.

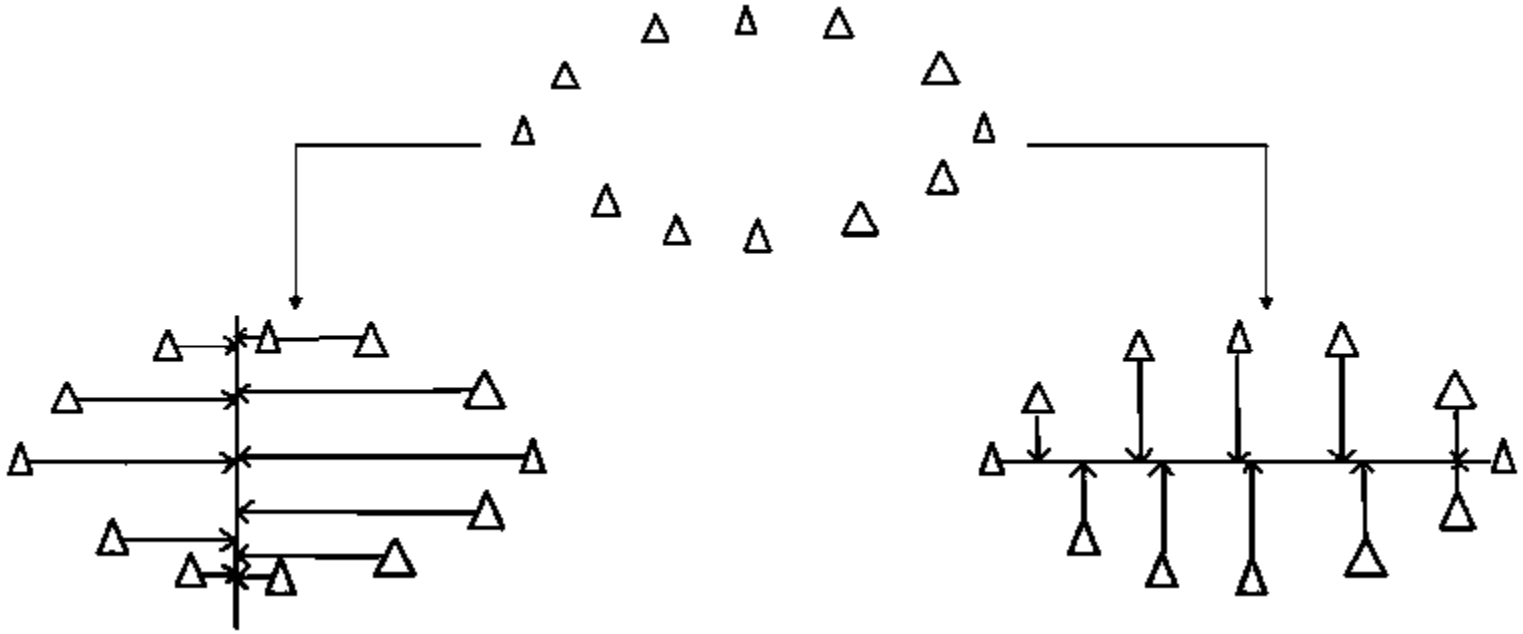
“정보유실”

PCA 개요

PCA 원리

- 다음과 같은 2차원 데이터를 좌측/ 우측 두 개의 축에 사영시킬 경우, 우측 기저(basis)가 좌측 기저에 비해 손실되는 정보의 양이 적음.

  우측 기저가 상대적으로 선호되는 기저



PCA 수리적 배경

- **주성분 분석 : 선형 결합**

데이터(X) 사영 변환 후 (Z)에도 분산이 보존하는 기저 (α)를 찾는 것

$$Z_1 = \alpha_1^T X = \alpha_{11}X_1 + \alpha_{12}X_2 + \cdots + \alpha_{1p}X_p$$
$$Z_2 = \alpha_2^T X = \alpha_{21}X_1 + \alpha_{22}X_2 + \cdots + \alpha_{2p}X_p$$
$$\vdots$$
$$Z_p = \alpha_p^T X = \alpha_{p1}X_1 + \alpha_{p2}X_2 + \cdots + \alpha_{pp}X_p$$

- $X_1, X_2, \cdots, X_P$: 원래 변수 (original variable)
- $a_i = [a_{i1}, a_{i2}, \ldots., a_{ip}]$: i번째 기저(basis) 또는 계수
- $Z_1, Z_2, \cdots, Z_p$: 각 기저로 사영 변환 후 변수(주성분)

PCA 수리적 배경

변수 / 관측치	X_1	…	X_i	…	X_p
N_1	x_{11}	…	x_{1i}	…	x_{1p}
…	…	…	…	…	…
N_i	x_{i1}	…	x_{ii}	…	x_{ip}
…	…	…	…	…	…
N_n	x_{n1}	…	x_{ni}	…	x_{np}

Mean vector

$$\bar{X} = \begin{bmatrix} \bar{x}_1 \\ \bar{x}_2 \\ \cdots \\ \bar{x}_p \end{bmatrix}$$

Covariance Matrix

$$C_n = \begin{bmatrix} s_{11} & \cdots & s_{1p} \\ \vdots & \ddots & \vdots \\ s_{p1} & \cdots & s_{pp} \end{bmatrix}$$

Correlation Matirx

$$R = \begin{bmatrix} 1 & r_{12} & \cdots & r_{1p} \\ r_{21} & 1 & \cdots & r_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ r_{p1} & r_{p2} & \cdots & 1 \end{bmatrix}$$

PCA 수리적 배경

▪ 공분산(Covariance)의 성질

- X를 p개의 변수와 n개의 개체로 구성된 n x p 행렬로 정의할 때

  X의 공분산 행렬: $Cov(X) = \frac{1}{n}(X - \bar{X})(X - \bar{X})^T$

- 공분산 행렬의 대각요소는 각 변수의 분산과 같으며,
- 비대각요소는 대응하는 두 변수의 공분산과 같다 (변수 개수: p)

$$C_x = Var[x] = \begin{bmatrix} Var[x_1] & Cov[x_1, x_2] & \dots & Cov[x_1, x_p] \\ Cov[x_2, x_1] & Var[x_2] & \dots & Cov[x_1, x_p] \\ \vdots & \vdots & \ddots & \vdots \\ Cov[x_p, x_1] & Cov[x_p, x_2] & \dots & Var[x_p] \end{bmatrix}$$

$$= \begin{bmatrix} \sigma_{11} & \sigma_{12} & \dots & \sigma_{1p} \\ \sigma_{21} & \sigma_{22} & \dots & \sigma_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{p1} & \sigma_{p2} & \dots & \sigma_{pp} \end{bmatrix} = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \dots & \sigma_{1p} \\ \sigma_{21} & \sigma_2^2 & \dots & \sigma_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{p1} & \sigma_{p2} & \dots & \sigma_p^2 \end{bmatrix}$$

- 데이터의 총분산은 공분산행렬의 대각성분들의 합

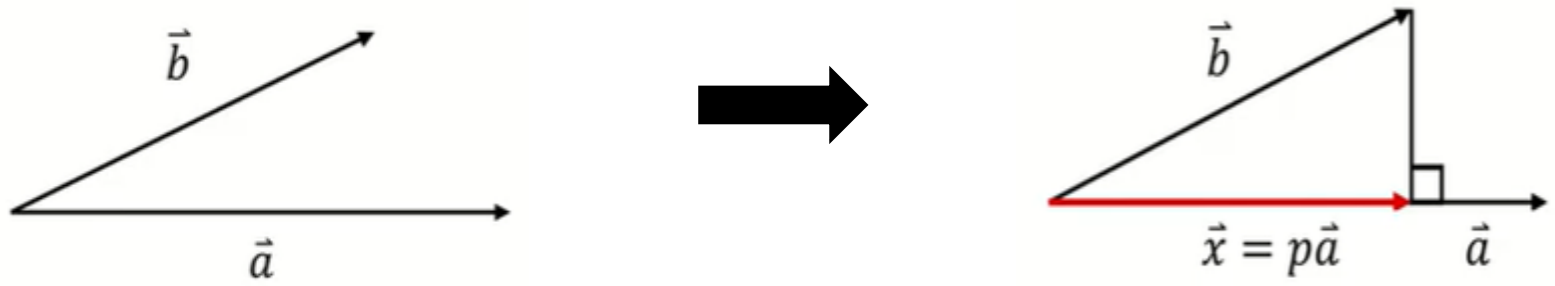
$$tr[Cov(\mathbf{X})] = Cov(\mathbf{X})_{11} + Cov(\mathbf{X})_{22} + Cov(\mathbf{X})_{33} + \dots + Cov(\mathbf{X})_{pp}$$

PCA 수리적 배경

▪ 사영(Projection)

- 한 벡터 $\vec{b}$를 다른 벡터 $\vec{a}$에 사영시킨다.

  즉, 벡터 $\vec{b}$ 로부터 벡터 $\vec{a}$ 에 수직인 점까지의 길이를 가지며 벡터 $\vec{a}$ 와 같은 방향을 갖는 벡터를 찾는다.



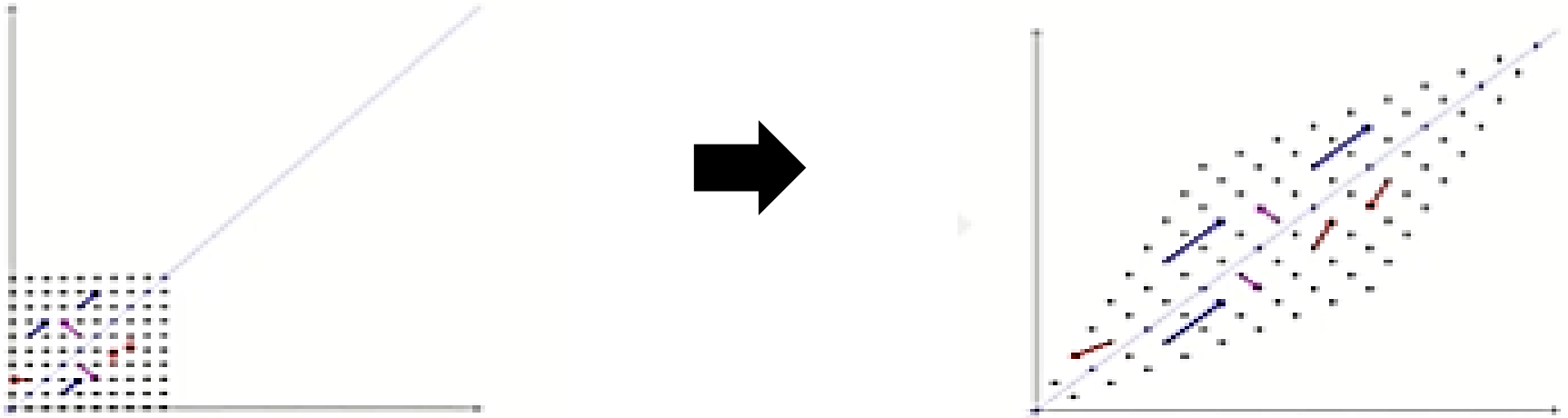
PCA 수리적 배경

▪ 고윳값 및 고유벡터

- 어떤 행렬 A에 대해 상수 λ와 벡터 x 가 아래의 식을 만족할 때,
  λ와 x 는 각각 행렬 의 고윳값(eigenvalue) 및 고유벡터(eigenvector)이다.

$$\mathbf{Ax} = \lambda\mathbf{x} \quad \rightarrow \quad (\mathbf{A} - \lambda\mathbf{I})\mathbf{x} = 0$$

- 벡터에 행렬을 곱한다는 것은 해당 벡터를 선형변환한다는 의미.
- 고유벡터: 선형변환에 의해 방향이 변하지 않는 벡터



PCA

▪ PCA알고리즘-주성분 추출

- X값들의 mean vector들이 있다고 가정한다. ($\bar{X}_i = 0, i = 1, \ldots, p$)
- X는 p-dimensional random vector 이다. 이로부터 구한 공분산행렬을 **∑** (pxp 행렬) 이라한다.
- α는 p-dimensional vector이며 길이는 1이다. ($\alpha^T\alpha$=1)
- Z는 원래변수의 선형결합으로 이루어져 있다. ($Z = \alpha^T X$)

이제 $\boldsymbol{\alpha}$ 를 구해야한다. 즉, Z ($Z = \alpha^T X$) **의 분산을 최대화하는** $\boldsymbol{\alpha}$를 구해야함.

$$\text{Max Var}(\mathbf{Z}) = \text{Var}(\alpha^T \mathbf{X}) = \alpha^T \text{Var}(\mathbf{X})\alpha = \alpha^T \Sigma \alpha$$

$$\text{s.t. } \|\alpha\| = \alpha^T\alpha = 1$$

PCA

- PCA알고리즘–주성분 추출

$$\text{Max } \alpha^T \Sigma \alpha = \alpha^T E \Lambda E^T \alpha$$
$$\text{s.t. } ||\alpha|| = 1$$

$\beta = E^T \alpha$ 로 치환

$$\text{Max } \beta^T \Lambda \beta \text{ where } \beta = E^T \alpha$$
$$\text{s.t. } ||\beta|| = 1$$

$$Max \quad \lambda_1 \beta_1^2 + \lambda_2 \beta_2^2 + \cdots + \lambda_m \beta_m^2$$
$$s.t \quad \beta_1^2 + \beta_2^2 + \cdots + \beta_m^2 = 1$$
$$\lambda_1 > \lambda_2 > \cdots > \lambda_m$$

eigenvalues and eigenvector of Σ

$$[E \wedge V] = svd(\Sigma)$$
$$\lambda_1 \geq \cdots \geq \lambda_m \geq 0$$
$$e_1, \ldots, e_m$$
$$\Lambda = diag(\lambda_1, \ldots, \lambda_m)$$

- 이 목적식이 최대가 되려면,
  β_1=1, β_2 =0, ⋯, β_m =0 이다.
  그리고, λ_1이 제일 크므로 λ_1=1 이다.
- 이를 $\beta = E^T \alpha$ 에 대입하면, α는 첫번째 eigenvector, 즉, e_1이 된다.
- 즉, 가장 큰 eigenvalue에 해당하는 eigenvector가 α가 된다.

∴ optimal value는 $\boldsymbol{\lambda_1}$이고, $\boldsymbol{\alpha = e_1}$ 이다.

PCA

▪ PCA 예제

- 다음과 같은 데이터 $\boldsymbol{X}$가 주어졌다고 하자. 변수는 3개, 관측치는 5개로 구성된다.

구분	n_1	n_2	n_3	n_4	n_5
p_1	0.2	0.45	0.33	0.54	0.77
p_2	5.6	5.89	6.37	7.9	7.87
p_3	3.56	2.4	1.95	1.32	0.98

우선, 변수(행)별로 평균을 0으로 centering한 행렬 $\boldsymbol{X}'$을 만든다.

구분	n_1	n_2	n_3	n_4	n_5
p_1	-1.1930	-0.0370	-0.5919	0.3792	1.4427
p_2	-1.0300	-0.7647	-0.3257	1.0739	1.0464
p_3	1.5012	0.3540	-0.0910	-0.7140	-1.0502

PCA

- PCA 예제

$$\sum = Cov(X') = \frac{1}{5-1} X' X'^{T}$$

$$= \begin{bmatrix} 0.0468 & 0.1990 & -0.1993 \\ 0.1990 & 1.1951 & -1.0096 \\ -0.1993 & -1.0096 & 1.0225 \end{bmatrix}$$

$$Correlation(X') = \begin{bmatrix} 1 & 0.8417 & -0.8840 \\ 0.8417 & 1 & -0.9133 \\ -0.8840 & -0.9133 & 1 \end{bmatrix}$$

우리는 correlation matrix (3x3)의 eigenvalue-eigenvector 를 구해야한다. $[E \wedge V] = svd(\Sigma)$

$$\lambda_1 = 0.0786, \quad e_1^T = [0.2590 \quad 0.5502 \quad 0.7938]$$
$$\lambda_2 = 0.1618, \quad e_2^T = [0.7798 \quad -0.6041 \quad 0.1643]$$
$$\lambda_3 = 2.7596, \quad e_3^T = [0.5699 \quad 0.5765 \quad -0.5855]$$

PCA

- PCA 예제

$$\lambda_1 = 0.0786, \quad e_1^T = [0.2590 \quad 0.5502 \quad 0.7938]$$
$$\lambda_2 = 0.1618, \quad e_2^T = [0.7798 \quad -0.6041 \quad 0.1643]$$
$$\lambda_3 = 2.7596, \quad e_3^T = [0.5699 \quad 0.5765 \quad -0.5855]$$

$X' =$

X_1	X_2	X_3
-1.1930	-1.0300	1.5012
-0.0370	-0.7647	0.3540
-0.5919	-0.3257	-0.0910
0.3792	1.0739	-0.7140
1.4427	1.0464	-1.0502

$$\lambda_3 > \lambda_2 > \lambda_1$$

$$Z_1 = e_1{}^T X' = 0.5699X_1 + 0.5765X_2 - 0.5855X_3 = 0.5699 \cdot \begin{bmatrix} -1.1930 \\ -0.0370 \\ -0.5919 \\ 0.3792 \\ 1.4427 \end{bmatrix} + 0.5765 \cdot \begin{bmatrix} -1.0300 \\ -0.7647 \\ -0.3257 \\ 1.0739 \\ 1.0464 \end{bmatrix} - 0.5855 \cdot \begin{bmatrix} 1.5012 \\ 0.3540 \\ -0.0910 \\ -0.7140 \\ -1.0502 \end{bmatrix} = \begin{bmatrix} -2.1527 \\ -0.6692 \\ -0.4718 \\ 1.2533 \\ 2.0404 \end{bmatrix}$$

$$Z_2 = e_2{}^T X' = \begin{bmatrix} -0.0615 \\ 0.4912 \\ -0.2798 \\ -0.4703 \\ 0.3204 \end{bmatrix}$$

$$Z_3 = e_3{}^T X' = \begin{bmatrix} 0.3160 \\ -0.1493 \\ -0.4047 \\ 0.1223 \\ 0.1157 \end{bmatrix}$$

$$\therefore Z = \begin{bmatrix} -2.1527 & -0.0615 & 0.3160 \\ -0.6692 & 0.4912 & -0.1493 \\ -0.4718 & -0.2798 & -0.4047 \\ 1.2533 & -0.4703 & 0.1223 \\ 2.0404 & 0.3204 & 0.1157 \end{bmatrix}$$

PCA

- PCA 예제

$$\therefore Z = \begin{bmatrix} -2.1527 & -0.0615 & 0.3160 \\ -0.6692 & 0.4912 & -0.1493 \\ -0.4718 & -0.2798 & -0.4047 \\ 1.2533 & -0.4703 & 0.1223 \\ 2.0404 & 0.3204 & 0.1157 \end{bmatrix} \rightarrow \text{Cov}(Z) = \begin{bmatrix} 2.7596 & 0 & 0 \\ 0 & 0.1618 & 0 \\ 0 & 0 & 0.0786 \end{bmatrix}$$

주성분(Z)들은 서로 독립이다.

결론적으로,

X' =

X_1	X_2	X_3
-1.1930	-1.0300	1.5012
-0.0370	-0.7647	0.3540
-0.5919	-0.3257	-0.0910
0.3792	1.0739	-0.7140
1.4427	1.0464	-1.0502

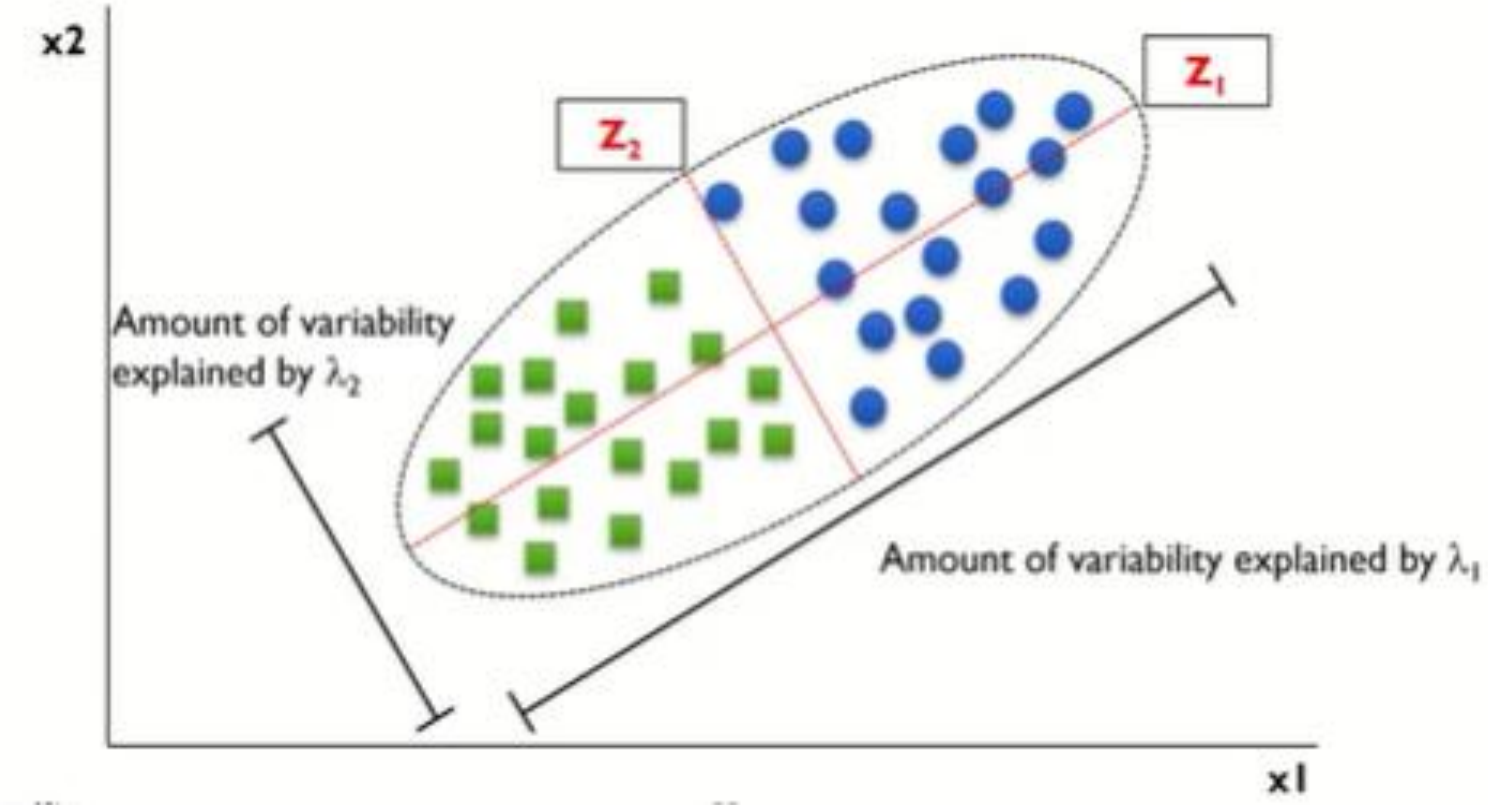
Z =

Z_1	Z_2	Z_3
-2.1527	-0.0615	0.3160
-0.6692	0.4912	-0.1493
-0.4718	-0.2798	-0.4047
1.2533	-0.4703	0.1223
2.0404	0.3204	0.1157

- 아직까지 차원을 줄이지는 않았다. 그러면, 몇 개의 주성분을 사용해야할까?

PCA

- PCA 예제
  - 공분산행렬의 고윳값 $(\lambda_1, \lambda_2, \lambda_3)$ =각 주성분의 분산



PCA

- PCA 예제
  - 공분산행렬의 고윳값$(\lambda_1, \lambda_2, \lambda_3)$ = 각 주성분의 분산

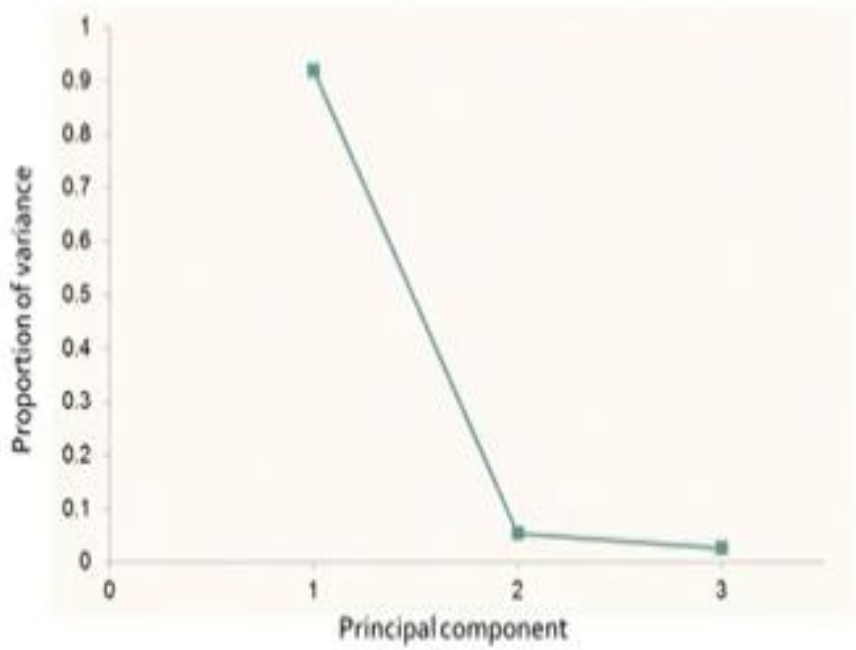
$$\mathrm{Cov}(Z) = \begin{bmatrix} 2.7596 & 0 & 0 \\ 0 & 0.1618 & 0 \\ 0 & 0 & 0.0786 \end{bmatrix}$$

$$\boldsymbol{Var(Z_1) = 2.7596 = \lambda_3} \text{ (가장 큰 고윳값)}$$
$$\boldsymbol{Var(Z_2) = 0.1618 = \lambda_2}$$
$$\boldsymbol{Var(Z_3) = 0.0786 = \lambda_1}$$

- 가장 큰 분산을 가진 축인 Z_1에 의해 설명되는 분산의 비율 $= \frac{\lambda_3}{\lambda_1+\lambda_2+\lambda_3} = \frac{2.7596}{0.0786+0.1618+2.7596} = 0.920$ (92%)

- 즉, 새 변수로 z_1 만 남기고 나머지를 생략하면,
  원데이터 X분산의 92%를 보존하면서도 원데이터를 3차원에서 1차원으로 줄일 수 있다.

PCA-주성분 개수 선택



- **방식1**: 고윳값 감소율이 유의미하게 낮아지는 elbow point 에 해당하는 주성분 수를 선택

- **방식2**: 일정 수준 이상의 분산비(보통 70%이상) 를 보존하는 최소의 주성분을 선택

PCA 요약

- 1단계: 데이터 정규화 (mean centering)
- 2단계: 기존 변수의 공분산 행렬 계산
- 3단계: 공분산 행렬로부터 고윳값 및 이에 해당하는 고유벡터를 계산
- 4단계: 고윳값 및 해당되는 고유벡터를 순서대로 나열

$$\lambda(1) > \lambda(2) > \lambda(3) > \lambda(4) > \lambda(5)$$

$$e(1) > e(2) > e(3) > e(4) > e(5),\ e(i),\ i=1,\ldots,5 \text{ is a vector}$$

- 5단계: 정렬된 고유벡터를 토대로 기존 변수를 변환

$$\begin{aligned} Z_1 &= e(1)\mathbf{X} = e_{11}\cdot X_1 + e_{12}\cdot X_2 + \ldots + e_{15}\cdot X_5 \\ Z_2 &= e(2)\mathbf{X} = e_{21}\cdot X_1 + e_{22}\cdot X_2 + \ldots + e_{25}\cdot X_5 \\ \ldots &\quad = \quad \ldots \\ Z_5 &= e(5)\mathbf{X} = e_{51}\cdot X_1 + e_{52}\cdot X_2 + \ldots + e_{55}\cdot X_5 \end{aligned}$$

iris 데이터에 PCA 적용하기

- **PCA실습1**
  - 사이킷런의 붓꽃 데이터를 load_iris() API를 이용해 로딩
  - 시각화를 편하게 하기 위해 DataFrame으로 변환

```
from sklearn.datasets import load_iris
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

# 사이킷런 내장 데이터 셋 API 호출
iris = load_iris()

# 넘파이 데이터 셋을 Pandas DataFrame으로 변환
columns = ['sepal_length','sepal_width','petal_length','petal_width']
irisDF = pd.DataFrame(iris.data , columns=columns)
irisDF['target']=iris.target
irisDF.head(3)
```

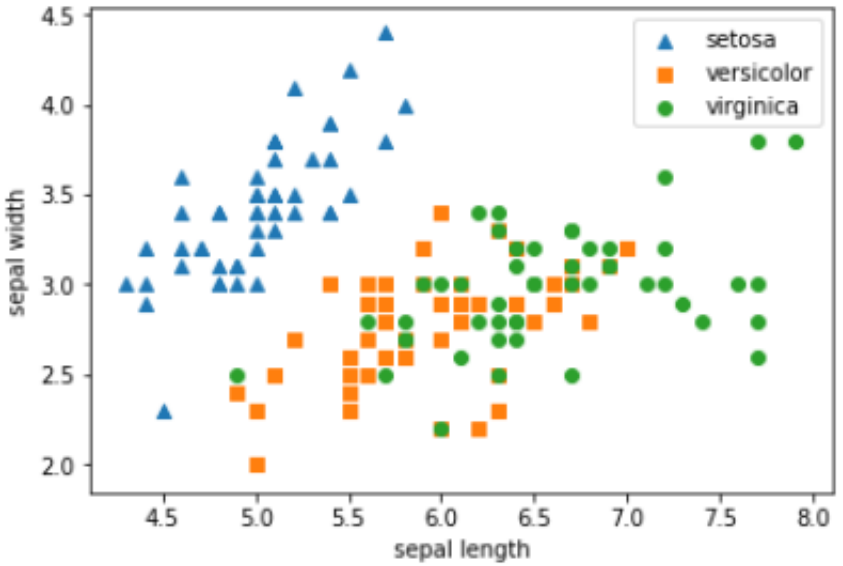
	sepal_length	sepal_width	petal_length	petal_width	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0

iris 데이터에 PCA 적용하기

- PCA실습1

  - sepal_length, sepal_width 두 개의 속성으로 데이터 산포 시각화

```
#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()
```

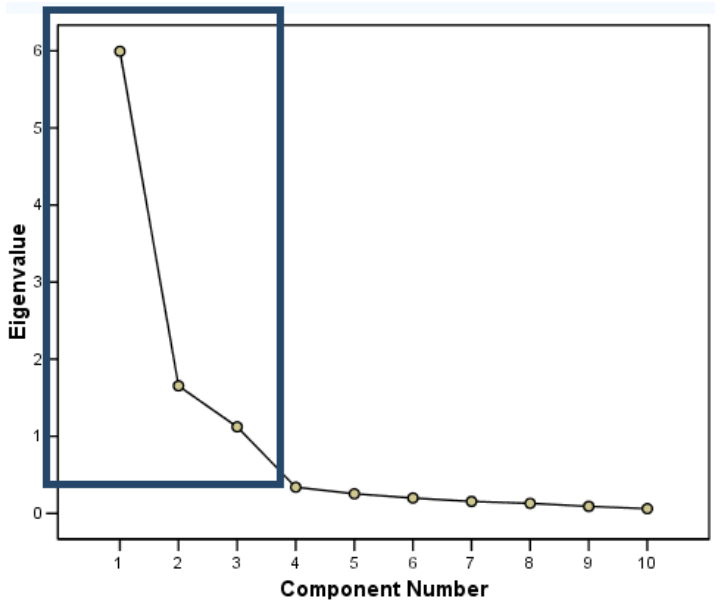


iris 데이터에 PCA 적용하기

- PCA실습1

  - PC의 개수 구하는 방법

    1) scree plot



scree plot은 x축을 주성분 개수, y축을 고윳값(설명가능한 분산값)으로 하는 linegraph를 의미한다.
line이 급작스럽게 완만해지는 지점인 4가 바로 적절한 주성분 개수라고 볼 수 있다.

iris 데이터에 PCA 적용하기

- PCA실습1

  - PC의 개수 구하는 방법

    1) scree plot

```
from sklearn.preprocessing import StandardScaler
iris_scaled = StandardScaler().fit_transform(irisDF.iloc[:,:-1])
iris_scaled.shape
```

```
(150, 4)
```

  - n_components=4로 수행

```
from sklearn.decomposition import PCA

pca = PCA(n_components=4)

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

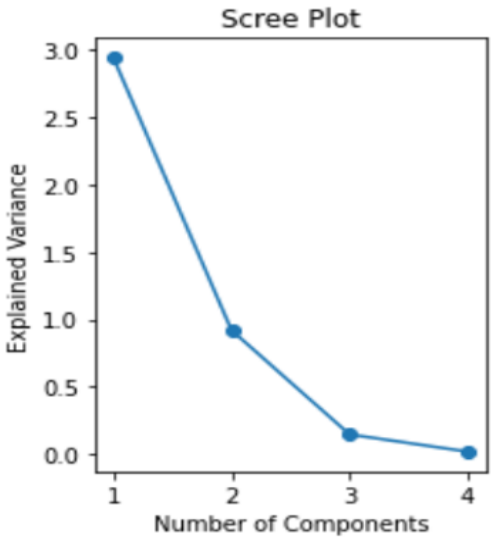
```
(150, 4)
```

iris 데이터에 PCA 적용하기

- PCA실습1

```
pc_values=np.arange(pca.n_components_)+1

plt.figure(figsize=(3,4))
plt.xlabel('Number of Components')
plt.ylabel('Explained Variance')
plt.plot(pc_values,pca.explained_variance_,'o-')
plt.title('Scree plot')
plt.show()
```

- iris 데이터로 확인한 결과



But! 데이터에 따라서 완만해지는 지점이 명확하지 않을 수 있기 때문에 적절한 주성분 개수를 관찰하기 위해 Scree plot을 사용하는 것이 애매모호할 수 있다.
=〉 따라서, 보통은 Scree plot 보다는 '변동성 비율' 과 '누적 변동성 비율' 을 주로 이용한다.

iris 데이터에 PCA 적용하기

- PCA실습1
  - 변동성 비율 & 누적 변동성 비율

변동성 비율=특정 주성분의 분산에 대한 비율=특정 고윳값의 비율 = $\frac{\text{특정 주성분 분산(=특정 고윳값)}}{\text{모든 주성분 분산의 합(=모든 고윳값의 합)}}$

변동성 비율을 계산하는 공식은 위와 같다.
하지만, scikit-learn의 PCA 메소드는 공식을 직접 구현할 필요없이 'explained_variance_ratio_'를 반환하면 쉽게 얻을 수 있다.

```
cumsum=np.cumsum(pca.explained_variance_ratio_)
result=pd.DataFrame({'고윳값':pca.explained_variance_,
                     '변동성비율':pca.explained_variance_ratio_},
                     index=np.array([f'pca{num+1}' for num in range(iris_scaled.shape[1])]))
result['누적변동성비율']=result['변동성비율'].cumsum()
result
```

	고윳값	변동성비율	누적변동성비율
pca1	2.938085	0.729624	0.729624
pca2	0.920165	0.228508	0.958132
pca3	0.147742	0.036689	0.994821
pca4	0.020854	0.005179	1.000000

iris 데이터에 PCA 적용하기

- PCA실습1

	고윳값	변동성비율	누적변동성비율
pca1	2.938085	0.729624	0.729624
pca2	0.920165	0.228508	0.958132
pca3	0.147742	0.036689	0.994821
pca4	0.020854	0.005179	1.000000

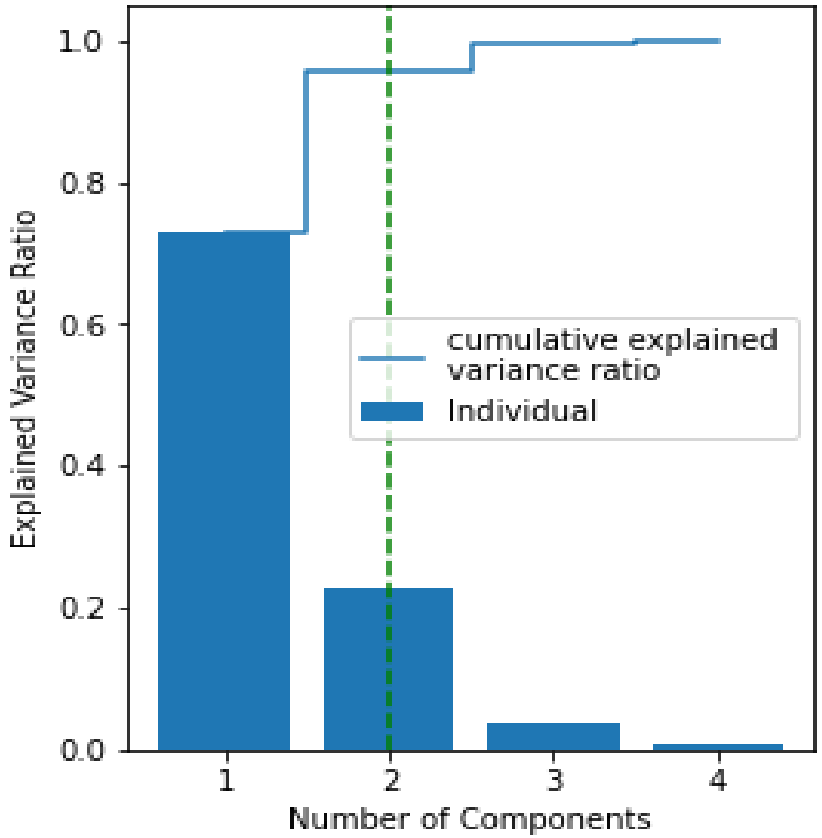
- 각 주성분의 고윳값과 주성분 마다 기여율을 누적한 누적 기여율을 계산한 DataFrame이 완성되었다.

- 여기서, 개별 고윳값, 즉, 각 주성분마다 고윳값이 0.7이상인 주성분들, 누적기여율이 80~90% 이상이 넘어가는 지점까지의 주성분들을 기준으로 적절한 주성분 개수를 설정한다.

iris 데이터에 PCA 적용하기

- PCA실습1

```
num_d=np.argmax(cumsum>=0.80)+1

var_ratio=pd.DataFrame({'Variance':pca.explained_variance_ratio_,'n_components':range(1,5)})

plt.figure(figsize=(4,5))
plt.bar('n_components','Variance',data=var_ratio,label='Individual')
plt.step(range(1,5),cumsum,where='mid',label='cumulative explained #n variance ratio')
plt.axvline(num_d,color='g',linestyle='--')
plt.xlabel('Number of Components')
plt.ylabel('Explained Variance Ratio')
plt.legend()
plt.show()
```



- 막대그래프는 변동성 비율, 계단 그래프는 누적 변동성 비율을 나타낸다.
- 초록색 점선은 누적 변동성 비율이 80% 이상인 기준선을 의미한다.
- 따라서 pc의 개수를 2개로 설정할 수 있으며, n_components = 2로 pca를 진행한다.

iris 데이터에 PCA 적용하기

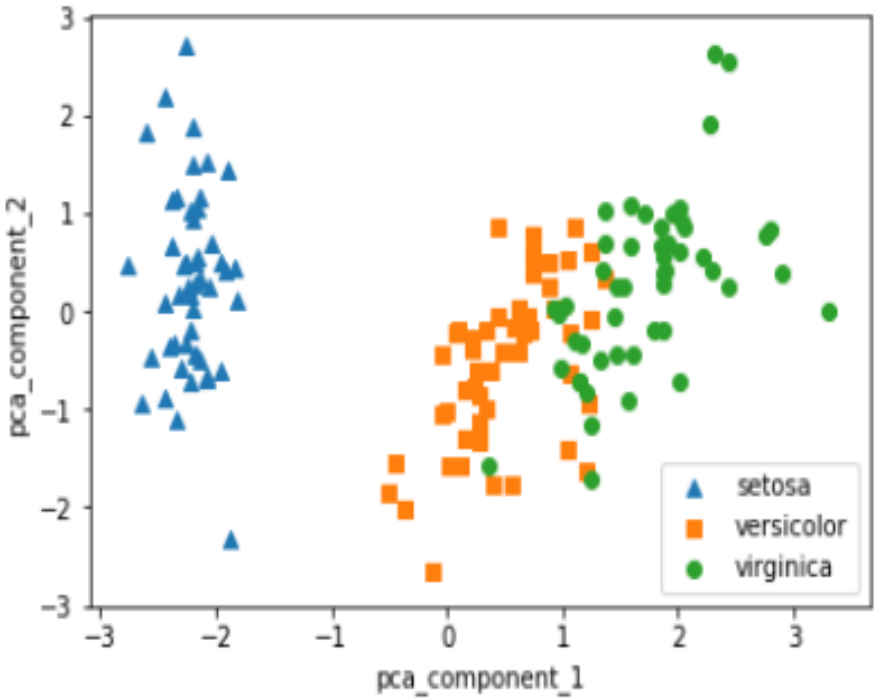
- PCA실습1
  - PCA **(n_components = 2)** 로 차원 축소된 피처들로 데이터 산포도 시각화

```
pca=PCA(n_components=2)

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

```
# PCA 변환된 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	-2.264703	0.480027	0
1	-2.080961	-0.674134	0
2	-2.364229	-0.341908	0

```
#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()
```



iris 데이터에 PCA 적용하기

- PCA실습1
  - 각 PCA Component별 변동성 비율

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

```
[0.72962445 0.22850762]
```

  - 원본 데이터와 PCA 변환된 데이터 기반에서 예측 성능 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
```

```
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
```

신용카드 데이터 세트 PCA 변환

- PCA실습2

- 데이터 로드 및 컬럼명 변환

```
import pandas as pd

df = pd.read_excel('default of credit card clients.xls', sheet_name='Data', header=1)
print(df.shape)
df.head(3)
```

```
(30000, 25)
```

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	...	BILL_AMT4	BILL_AMT5	BILL_AMT6	PAY_AMT1	PAY_AMT2
0	1	20000	2	2	1	24	2	2	-1	-1	...	0	0	0	0	689
1	2	120000	2	2	2	26	-1	2	0	0	...	3272	3455	3261	0	1000
2	3	90000	2	2	2	34	0	0	0	0	...	14331	14948	15549	1518	1500

3 rows × 25 columns

```
df.rename(columns={'PAY_0':'PAY_1','default payment next month':'default'}, inplace=True)
y_target = df['default']
# ID, default 컬럼 Drop
X_features = df.drop(['ID','default'], axis=1)
```

```
y_target.value_counts()
```

```
0    23364
1     6636
Name: default, dtype: int64
```

출처: https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients

신용카드 데이터 세트 PCA 변환

- PCA실습2
  - 데이터에 대한 정보 보기

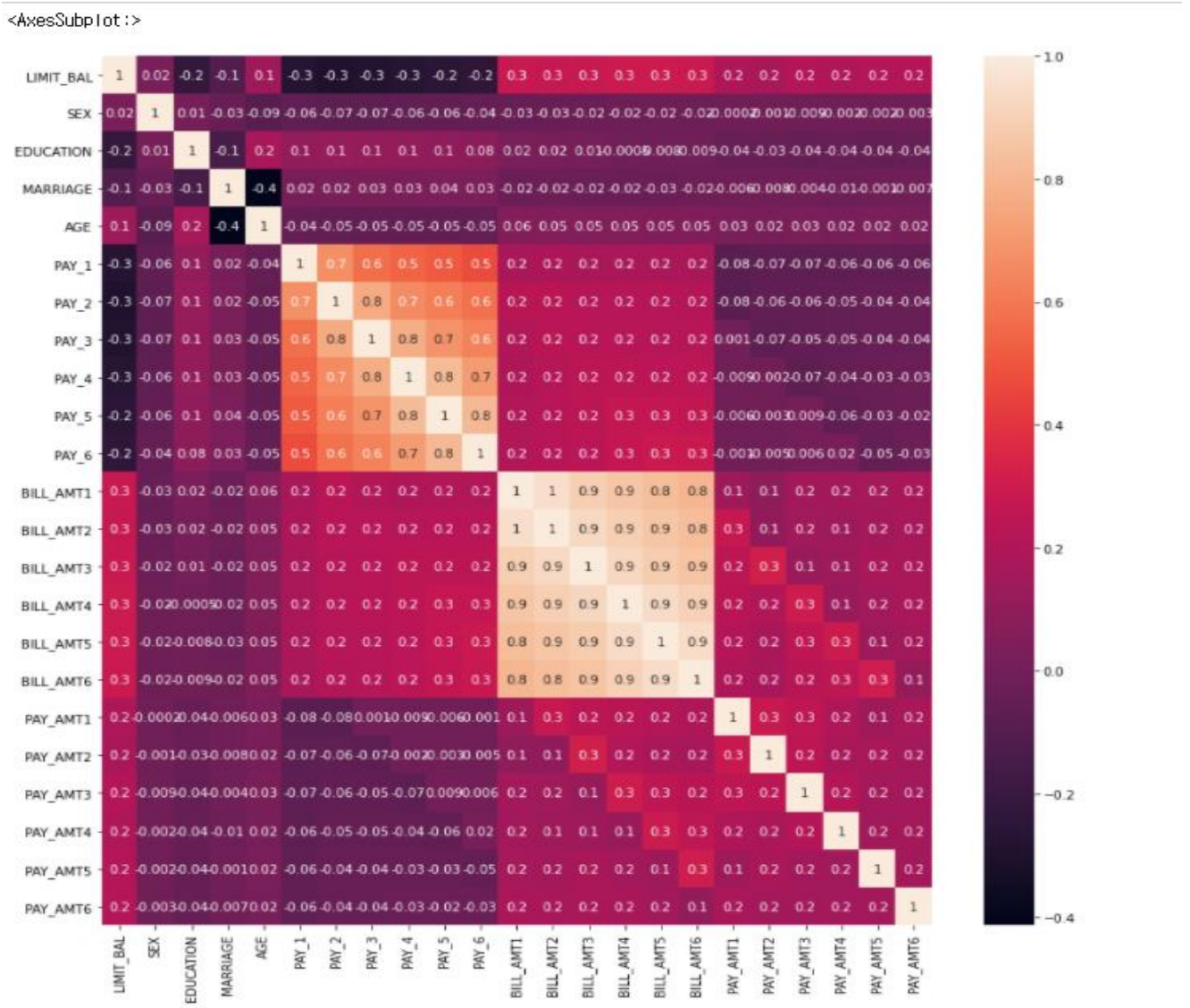
```
X_features.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 23 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   LIMIT_BAL  30000 non-null  int64
 1   SEX        30000 non-null  int64
 2   EDUCATION  30000 non-null  int64
 3   MARRIAGE   30000 non-null  int64
 4   AGE        30000 non-null  int64
 5   PAY_1      30000 non-null  int64
 6   PAY_2      30000 non-null  int64
 7   PAY_3      30000 non-null  int64
 8   PAY_4      30000 non-null  int64
 9   PAY_5      30000 non-null  int64
 10  PAY_6      30000 non-null  int64
 11  BILL_AMT1  30000 non-null  int64
 12  BILL_AMT2  30000 non-null  int64
 13  BILL_AMT3  30000 non-null  int64
 14  BILL_AMT4  30000 non-null  int64
 15  BILL_AMT5  30000 non-null  int64
 16  BILL_AMT6  30000 non-null  int64
 17  PAY_AMT1   30000 non-null  int64
 18  PAY_AMT2   30000 non-null  int64
 19  PAY_AMT3   30000 non-null  int64
 20  PAY_AMT4   30000 non-null  int64
 21  PAY_AMT5   30000 non-null  int64
 22  PAY_AMT6   30000 non-null  int64
dtypes: int64(23)
memory usage: 5.3 MB
```

신용카드 데이터 세트 PCA 변환

- PCA실습2
  - 피처 간 상관도 시각화

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

corr = X_features.corr()
plt.figure(figsize=(14,14))
sns.heatmap(corr, annot=True, fmt='.1g')
```



신용카드 데이터 세트 PCA 변환

- PCA실습2
  - 상관도가 높은 피처들의 PCA 변환 후 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 ]
```

신용카드 데이터 세트 PCA 변환

- PCA실습2

  - 원본 데이터 세트와 6개 컴포넌트로 PCA 변환된 데이터 세트로 분류 예측 성능 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('CV=3 인 경우의 개별 Fold세트별 정확도:',scores)
print('평균 정확도:{0:.4f}'.format(np.mean(scores)))
```

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

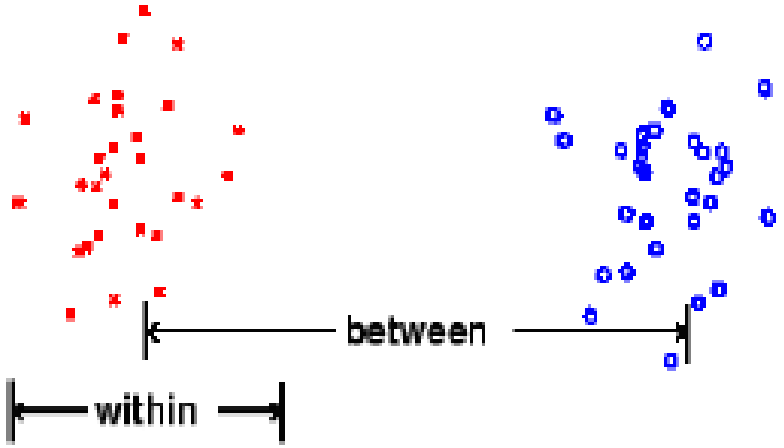
```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# 원본 데이터셋에 먼저 StandardScaler적용
scaler = StandardScaler()
df_scaled = scaler.fit_transform(X_features)

# 6개의 Component를 가진 PCA 변환을 수행하고 cross_val_score( )로 분류 예측 수행.
pca = PCA(n_components=6,random_state=0)
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 Fold세트별 정확도:',scores_pca)
print('PCA 변환 데이터 셋 평균 정확도:{0:.4f}'.format(np.mean(scores_pca)))
```

```
CV=3 인 경우의 PCA 변환된 개별 Fold세트별 정확도: [0.7902 0.7964 0.8025]
PCA 변환 데이터 셋 평균 정확도:0.7964
```

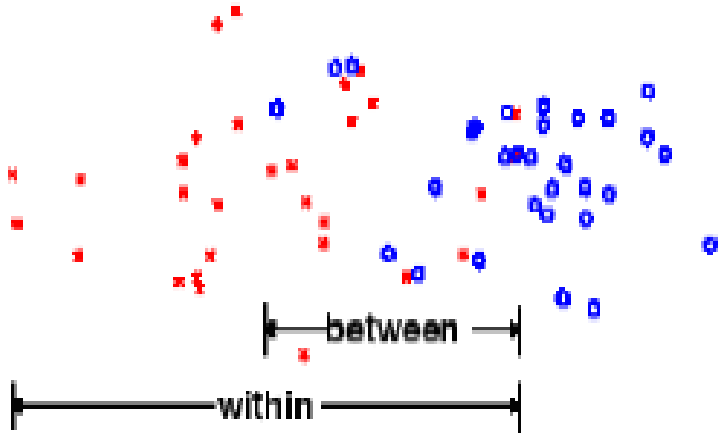
3. LDA

LDA

- LDA (Linear Discriminant Analysis) 선형 판별 분석
- 클래스 간 분산(between-class scatter)은 최대화 클래스 내 분산(within-class scatter)은 최소화하는 방식
  - -> 데이터에 대한 특징 벡터의 차원을 축소



[판별 용이한 데이터 분포]



[판별 어려운 데이터 분포]

->LDA는 가능한 클래스 간의 분별 정보를 최대한 유지시키기 위해,
특징 공간에서 클래스 분리를 최대로 하는 주축을 기준으로 사상시켜 차원을 축소 !

LDA VS PCA

- LDA
  - 데이터의 최적 분류(best discrimination between classes)의 관점에서 차원을 축소한다.
  - 데이터의 클래스의 차이가 분산보다 평균의 차이에 있을 때, LDA는 PCA보다 뛰어난 성능을 보여준다.
  - 3D plot으로 데이터를 표현할 때, LDA는 PCA보다 뛰어난 성능을 보여준다.
- PCA
  - 데이터의 최적 표현(best description of the data in its entirety)의 관점에서 데이터를 축소한다.
  - 데이터의 클래스의 차이가 평균보다 분산의 차이에 있을 때, PCA는 LDA보다 뛰어난 성능을 보여준다.

LDA 구하기

Step1 : 클래스 내부와 클래스 간 분산 행렬을 구한다.

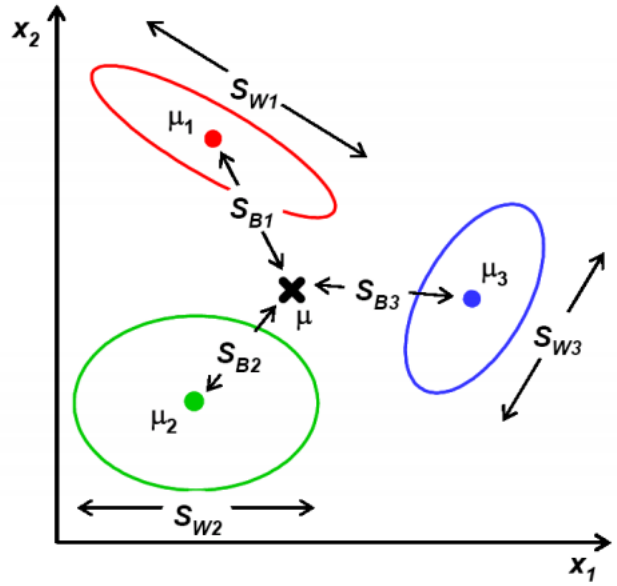
- 각 클래스내의 데이터에 대한 클래스내 분산 행렬

$$\mathbf{S_W} = \sum_{i=1}^{c} \mathbf{S_i} \quad , \quad \text{여기서} \quad \mathbf{S_i} = \sum_{\mathbf{x}\in\boldsymbol{\omega}_\mathbf{i}} (\mathbf{x}-\boldsymbol{\mu}_\mathbf{i})(\mathbf{x}-\boldsymbol{\mu}_\mathbf{i})^{\mathbf{T}} \quad , \quad \boldsymbol{\mu}_i = \frac{1}{N_i}\sum_{\mathbf{x}\in\omega_i} \mathbf{x}$$

- 각 클래스간 분산 행렬

$$\mathbf{S_B} = \sum_{i=1}^{c} N_i(\boldsymbol{\mu}_\mathbf{i}-\boldsymbol{\mu})(\boldsymbol{\mu}_\mathbf{i}-\boldsymbol{\mu})^{\mathrm{T}} \quad , \text{ 여기서 } \quad \boldsymbol{\mu} = \frac{1}{N}\sum_{\forall \mathbf{x}}^{c} \mathbf{x} = \frac{1}{N}\sum_{\mathbf{x}\in\omega_i} N_i\boldsymbol{\mu}_\mathbf{i}$$

- 총 분산행렬(total scatter matrix) ➡ $S_T = S_B + S_W$



3개의 클래스에 대한 LDA

LDA 구하기

Step2 : 클래스 내부 분산 행렬을 Sw, 클래스 간 분산 행렬을 Sb라고 하면 다음 식으로 두 행렬을 고유 벡터로 분해한다.

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

LDA 구하기

Step3 : 고윳값이 가장 큰 순으로 K개 (LDA변환 차수만큼)의 고유 벡터를 추출한다.

Step4 : 추출된 고유 벡터를 열로 하는 변환 행렬 W를 이용해 새롭게 입력 데이터를 변환한다.

$$\mathbf{y} = \mathbf{W}_i^T \mathbf{x}$$

더 자세한 Step

- **선형 판별 분석의 내부 동작 방식**

1. d차원 데이터셋을 표준화 전처리(d = 특성 갯수)
2. 각 클래스에 대해 d차원 평균 벡터를 계산
3. 클래스 간의 산포 행렬(scatter matrix) S(B)와 클래스 내부의 산포행렬 S(W)를 구성
4. S(W)의 역행렬과 S(B)의 곱행렬의 고유벡터와 고윳값을 계산
5. 고윳값을 내림차순으로 정렬해 순서를 매긴다.
6. 고윳값이 가장 큰 k개의 고유벡터를 선택해 d * k 차원의 변환행렬 W를 구한다.
7. 변환 행렬 W를 사용해 새로운 특성 부분 공간으로 투영

- LDA와 PCA는 행렬을 고윳값과 고유벡터로 분해하여 새로운 저차원 공간을 구성한다는 점에서 매우 비슷하다. 하지만 LDA는 2단계에서 평균 벡터를 만들 때 클래스 별로 데이터를 나누어 평균을 구한다는 차이점이 존재한다.

iris 데이터에 LDA 적용하기

- LDA 실습

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

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

  - 2개의 컴포넌트로 LDA 변환
  - LDA는 PCA(비지도학습)와 다르게 지도학습이기 때문에 클래스의 결정값이 변환 시에 필요
  - LDA 객체의 fit()메서드를 호출할 때 결정값(iris.target)이 입력됐음에 유의!

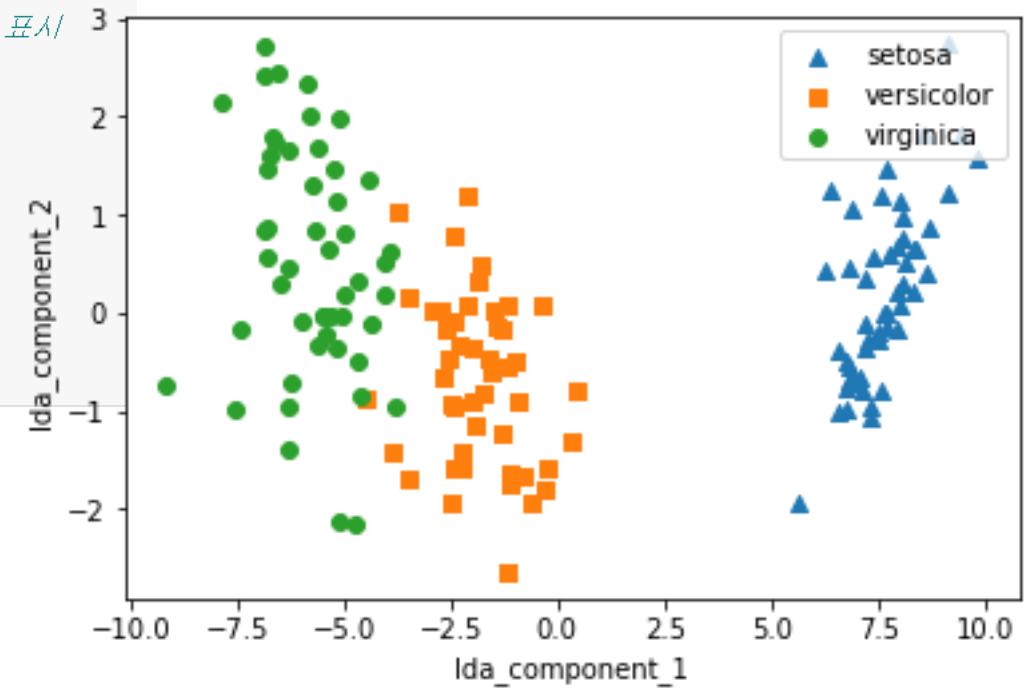
```
lda = LinearDiscriminantAnalysis(n_components = 2)
lda.fit(iris_scaled, iris.target)
iris_lda = lda.transform(iris_scaled)
print(iris_lda.shape)
```

```
(150, 2)
```

iris 데이터에 LDA 적용하기

- LDA 실습

```
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 target별 다른 모양으로 산점도 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()
```



4. SVD

SVD 개요

SVD (Singular Value Decomposition, 특이값 분해)

- PCA와 유사한 행렬 분해 기법
  - PCA -> 정방행렬만을 고유 벡터로 분해 가능
  - SVD -> 행과 열의 크기가 다른 행렬에도 적용 가능
- 고윳값 분해(eigen decomposition)처럼 행렬을 대각화하는 방법 중 하나

SVD의 정의

행렬 A를 다음과 같이 분해 가능하다

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

A : m x n matrix

U : m x m orthogonal matrix

Σ : m x n diagonal matrix

V : n x n orthogonal matrix

orthogonal matrix (직교행렬)

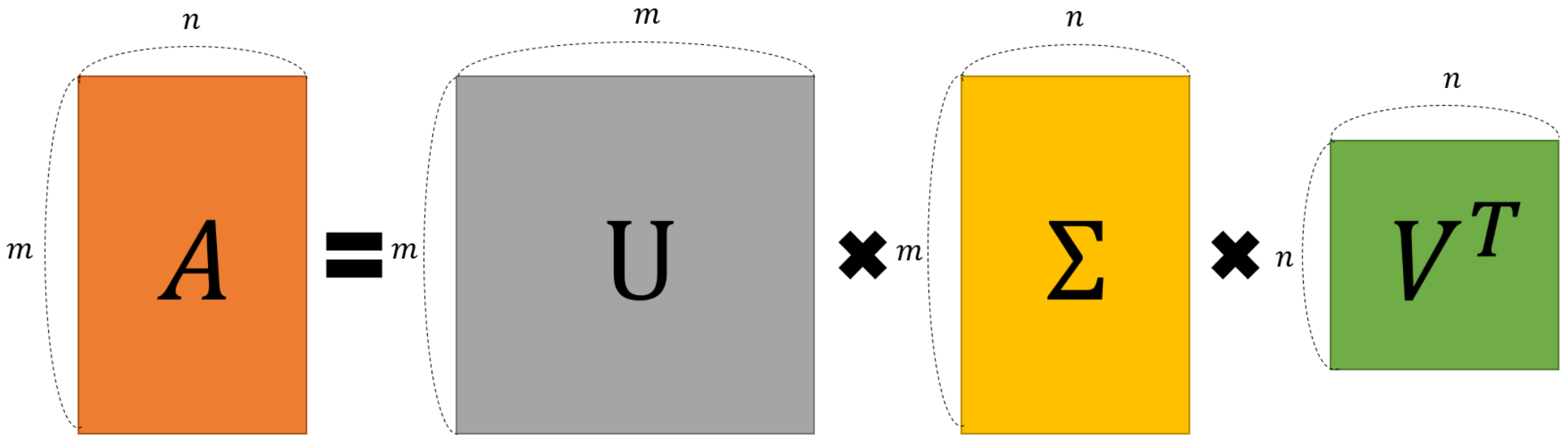
$$UU^T = U^T U = I, U^{-1} = U^T$$

$$VV^T = V^T V = I, V^{-1} = V^T$$

diagonal matrix (대각 행렬)

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix}$$

SVD의 정의



SVD의 기하학적 의미

EX. $A \begin{bmatrix} v_1 & v_2 \end{bmatrix} = \begin{bmatrix} u_1 & u_2 \end{bmatrix} \times \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix}$

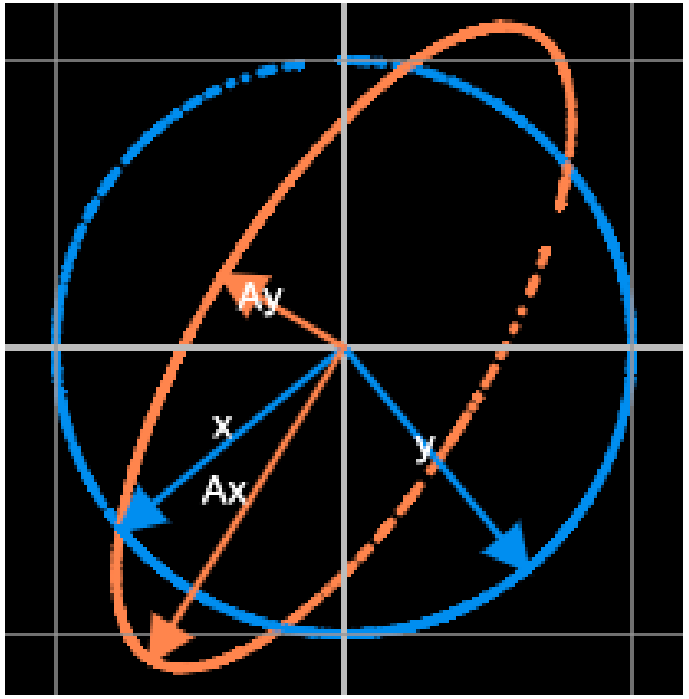
직교하는 벡터 집합에 대하여,

선형 변환 후에 그 크기는 변하지만,

여전히 직교할 수 있게 만드는

그 직교 벡터 집합은 무엇이고, 변형 후의 결과는 무엇인가?

SVD의 기하학적 의미



직교하는 벡터 x, y에 대하여

선형 변환 후에 크기는 변하지만,

여전히 직교하는 두 벡터 Ax, Ay

SVD의 기하학적 의미

EX. $A \begin{bmatrix} v_1 & v_2 \end{bmatrix} = \begin{bmatrix} u_1 & u_2 \end{bmatrix} \times \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix}$

$$AV = U\Sigma$$

$$AVV^T = U\Sigma V^T$$

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

U, V 계산

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

$$A^T = \left(U\Sigma V^T\right)^T = V\Sigma^T U^T$$

$$\Sigma\Sigma^T = \begin{bmatrix} \sigma_1^2 & & 0 \\ & \ddots & \\ 0 & & \sigma_n^2 \end{bmatrix}$$

1. $\longrightarrow$ $AA^T = U\Sigma V^T V\Sigma^T V^T$

$$= U\Sigma\Sigma^T U^T$$

$$= U\Sigma^2 U^T$$

고윳값 분해를 통해 계산

2. $\longrightarrow$ $A^T A = V\Sigma^2 V^T$

numpy를 이용한 SVD

- a 행렬에 numpy.linalg.svd 적용해 U, Sigma, Vt 도출
- Sigma는 0이 아닌 값의 경우만 1차원 행렬로 표현

```
# 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]]
```

```
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]]
```

numpy를 이용한 SVD

- 원본 행렬로 정확히 복원되는지 확인!

```
# 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]]
```

numpy를 이용한 SVD

- 데이터 세트가 row 간 의존성이 있는 경우 Sigma 값 중 2개가 0으로 변함.
- 즉, 선형 독립인 row 벡터의 개수가 2개

```
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 ]]
```

```
# 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))
```

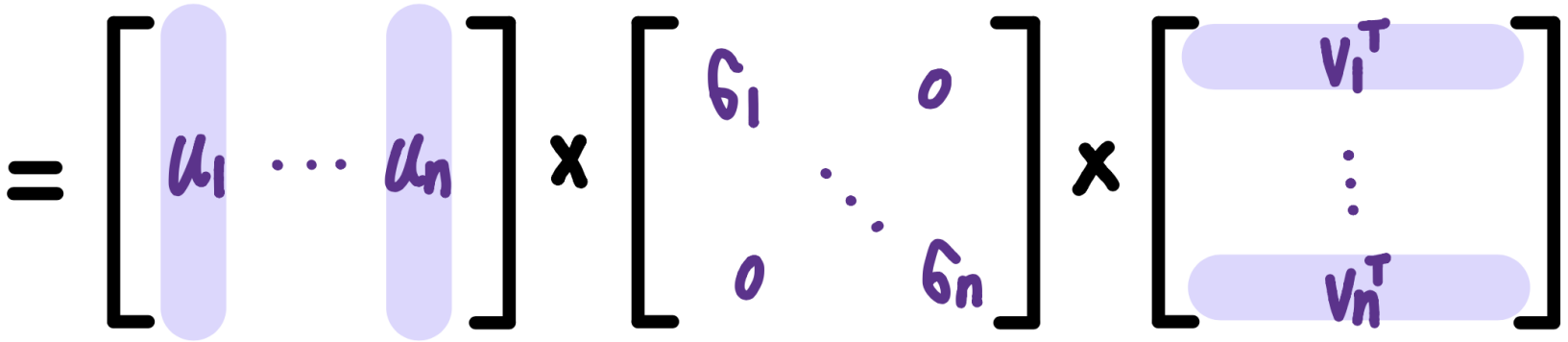
```
(4, 4) (4,) (4, 4)
Sigma Value:
 [2.663 0.807 0.    0.   ]
```

```
# 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 ]]
```

SVD의 목적과 활용

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

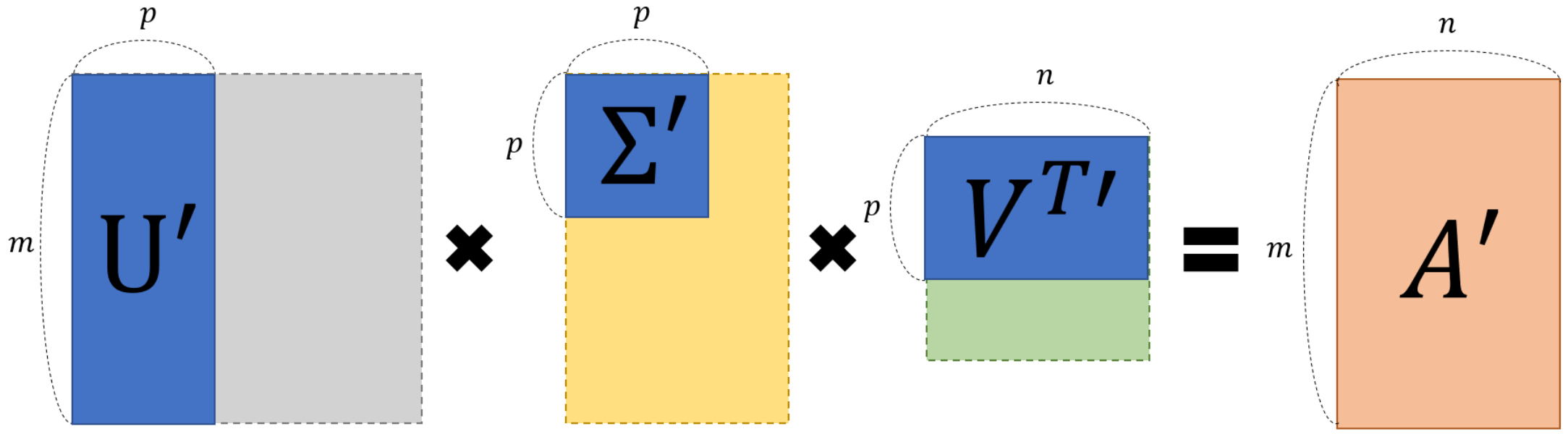


$$= \sigma_1 u_1 v_1^T + \cdots + \sigma_n u_n v_n^T$$

정보량의 크기 matrix 정보량의 크기 matrix

-> 정보량의 크기에 따라 A를 여러 layer로 쪼개어 준다.

SVD의 목적과 활용



- 기존의 A행렬을 특이값(Singular Value) p개만을 이용해 A'라는 행렬로 부분 복원
- 특이값의 크기에 따라 A의 정보량이 결정 -> 값이 큰 몇 개의 특이값들을 가지고도 충분히 유용한 정보를 유지할 수 있다.

=>Truncated SVD

SVD의 목적과 활용

- Truncated SVD로 근사한 행렬 A'은 데이터압축, 노이즈제거 등에 활용될 수 있다.
- 데이터 압축의 한 예로 아래와 같은 600 × 367 이미지를 SVD로 압축해 보자.

SVD의 목적과 활용

- p = singular value 개수

20개의 singular value로 근사 (p = 20)

50개의 singular value로 근사 (p = 50)

100개의 singular value로 근사 (p = 100)

SVD의 목적과 활용

〈원본 이미지〉

〈100개의 singular value로 근사 (p = 100)〉

압축률 43.9%이지만 데이터의 핵심을 잘 잡아내고 있음

Truncated SVD

- Truncated SVD는 Sigma행렬에 있는 대각 원소, 즉 특이값 중 상위 일부 데이터만 추출해 분해하는 방식.
- scipy모듈의 svds를 이용.

```
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)

# 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 TruncatedSVD

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

Truncated SVD

- 완벽하게 복원되지 않고 근사적으로 복원됨을 알 수 있다.

```
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]]
```

사이킷런 Truncated SVD 클래스 이용

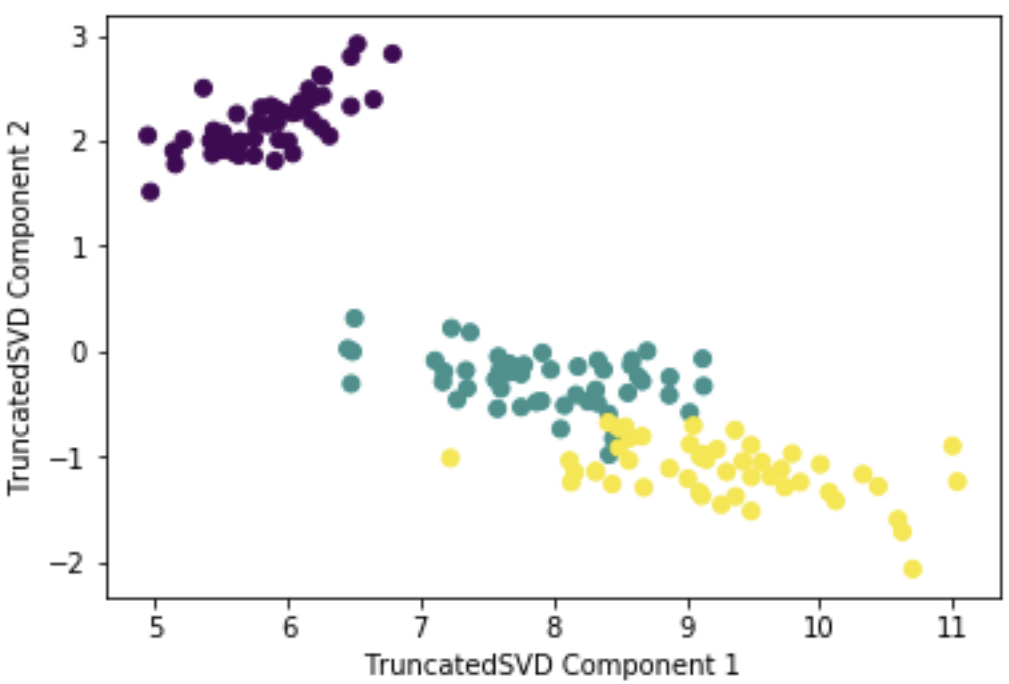
- 사이킷런의 Truncated SVD 클래스는 PCA 클래스와 유사하게 fit(), transform()을 호출해 원본 데이터를 몇 개의 주요 컴포넌트로 차원을 축소해 변환.

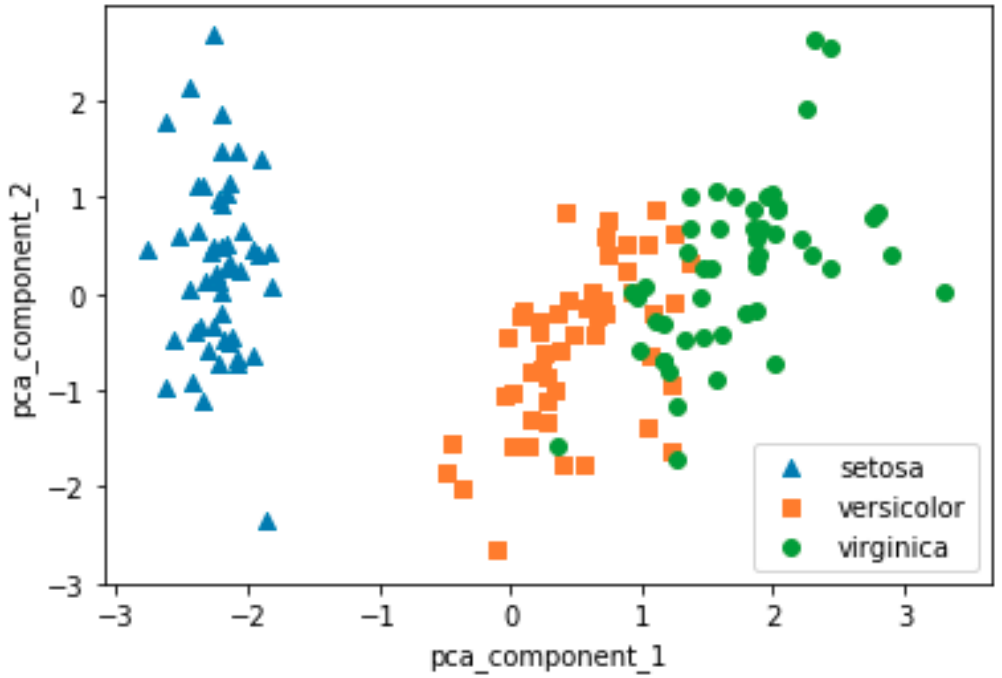
```
from sklearn.decomposition import TruncatedSVD, PCA
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
%matplotlib inline

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')
```

사이킷런 Truncated SVD 클래스 이용

- PCA와 유사하게 변환 후에 품종별로 어느 정도 클러스터링이 가능할 정도로 각 변환 속성으로 뛰어난 고유성을 가지고 있음을 알 수 있다.

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





사이킷런 Truncated SVD 와 PCA

- 사이킷런의 Truncated SVD, PCA 클래스 구현을 살펴보면 모두 SVD를 이용해 행렬을 분해한다.

```
from sklearn.preprocessing import StandardScaler

# iris 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')
```

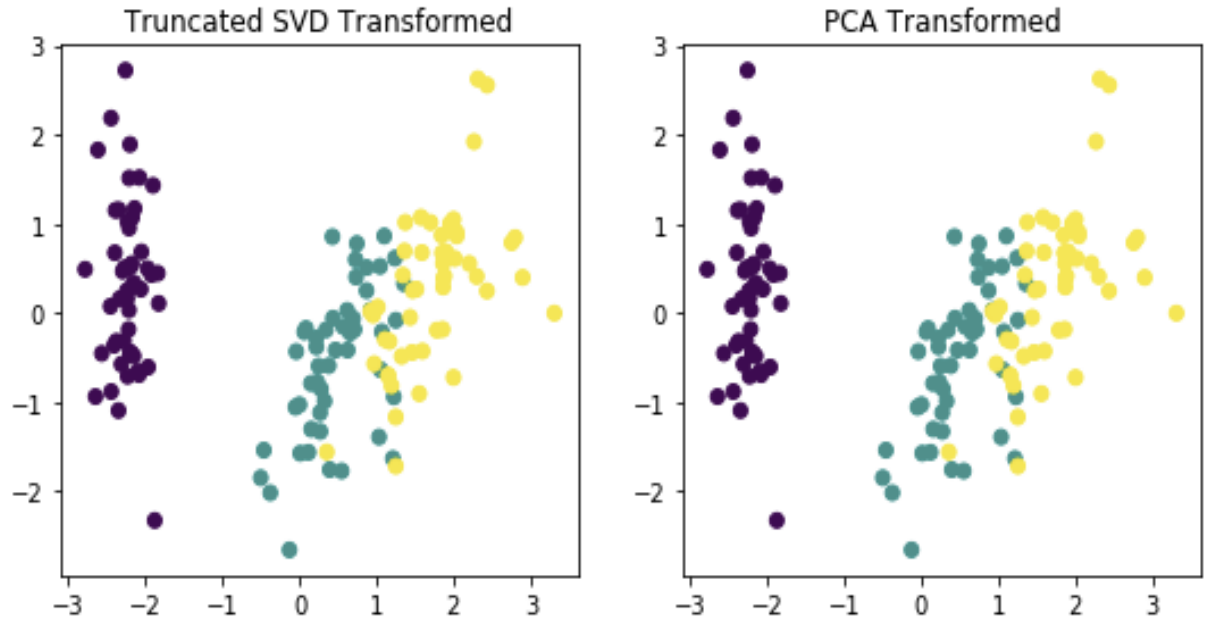
사이킷런 Truncated SVD 와 PCA

- 두 개의 변환 행렬 값과 원복 속성별 컴포넌트 비율값을 비교해 보면 거의 같음을 알 수 있다.

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

```
2.339760329927998e-15
4.85722573273506e-17
```

```
Text(0.5,1,'PCA Transformed')
```



사이킷런 Truncated SVD 와 PCA

- 모두 0에 가까운 값 -> 2개의 변환이 서로 동일하다
- 데이터 세트가 스케일링으로 데이터 중심이 동일해지면 사이킷런의 SVD와 PCA는 동일한 변환을 수행. 이는 PCA가 SVD 알고리즘으로 구현됐음을 의미

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

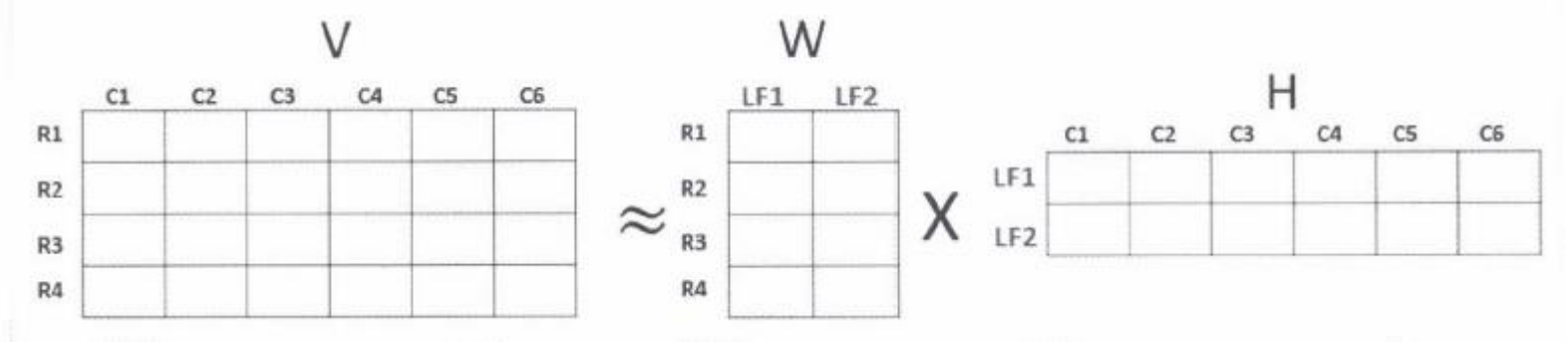
```
2.339760329927998e-15
4.8572573273506e-17
```

5.NMF

NMF

원본 행렬 내의 모든 원소값이 모두 양수라는 게 보장되면, 간단하게 두 개의 기반 행렬로 분해될 수 있는 기법

- 비음수 행렬 분해
- 근사 해 분해
- 전체 원소가 양수인 행렬 V를 음수를 포함하지 않는 행렬 W와 H의 곱으로 분해하는 알고리즘
- 차원 축소 보다는 요인 추출! -> 잠재 요소 도출

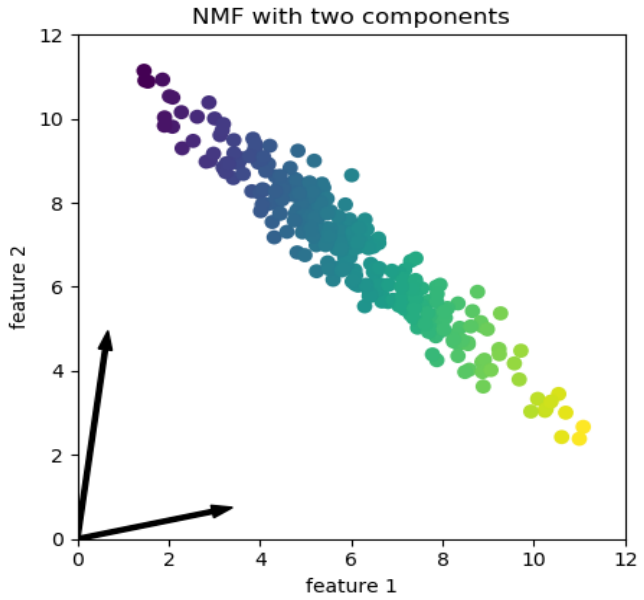


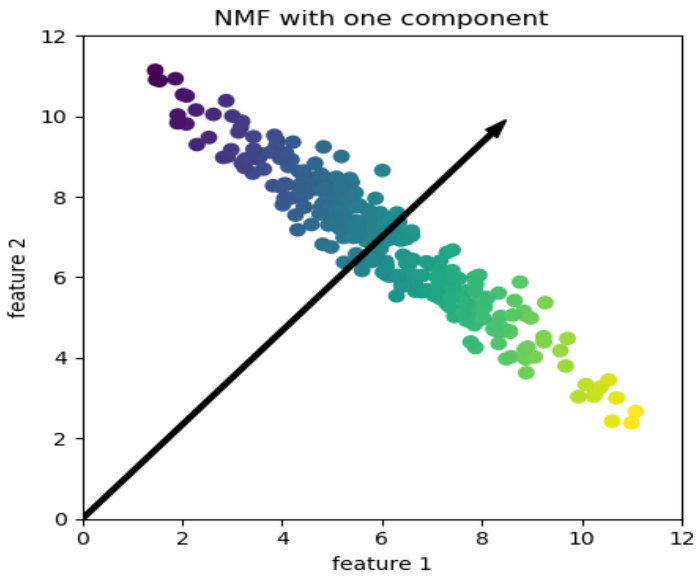
- 분해 행렬 W는 원본 행에 대해서 이 잠재 요소의 값이 얼마나 되는지에 대응
- 분해 행렬 H는 이 잠재요소가 원본 열(즉, 원본 속성)로 어떻게 구성됐는지를 나타내는 행렬

PCA vs NMF

- PCA
  - 최대 분산의 방향=주성분->성분 간의 우열이 있다.
  - Feature들 간의 직교성이 보장된다.

- NMF
  - 성분의 개수를 줄이면 특정 방향이 제거될 뿐만 아니라 전체 성분이 완전히 바뀐다.
  - 특성이 양수이기만 하면 성분의 우열 없이 특징을 나눌 수 있다.
  - PCA나 SVD에 비해 데이터 구조를 좀 더 잘 반영
    (PCA나 SVD처럼 feature벡터들이 서로 직교하면 실제 데이터구조를 잘 반영 못할 수 있음.)





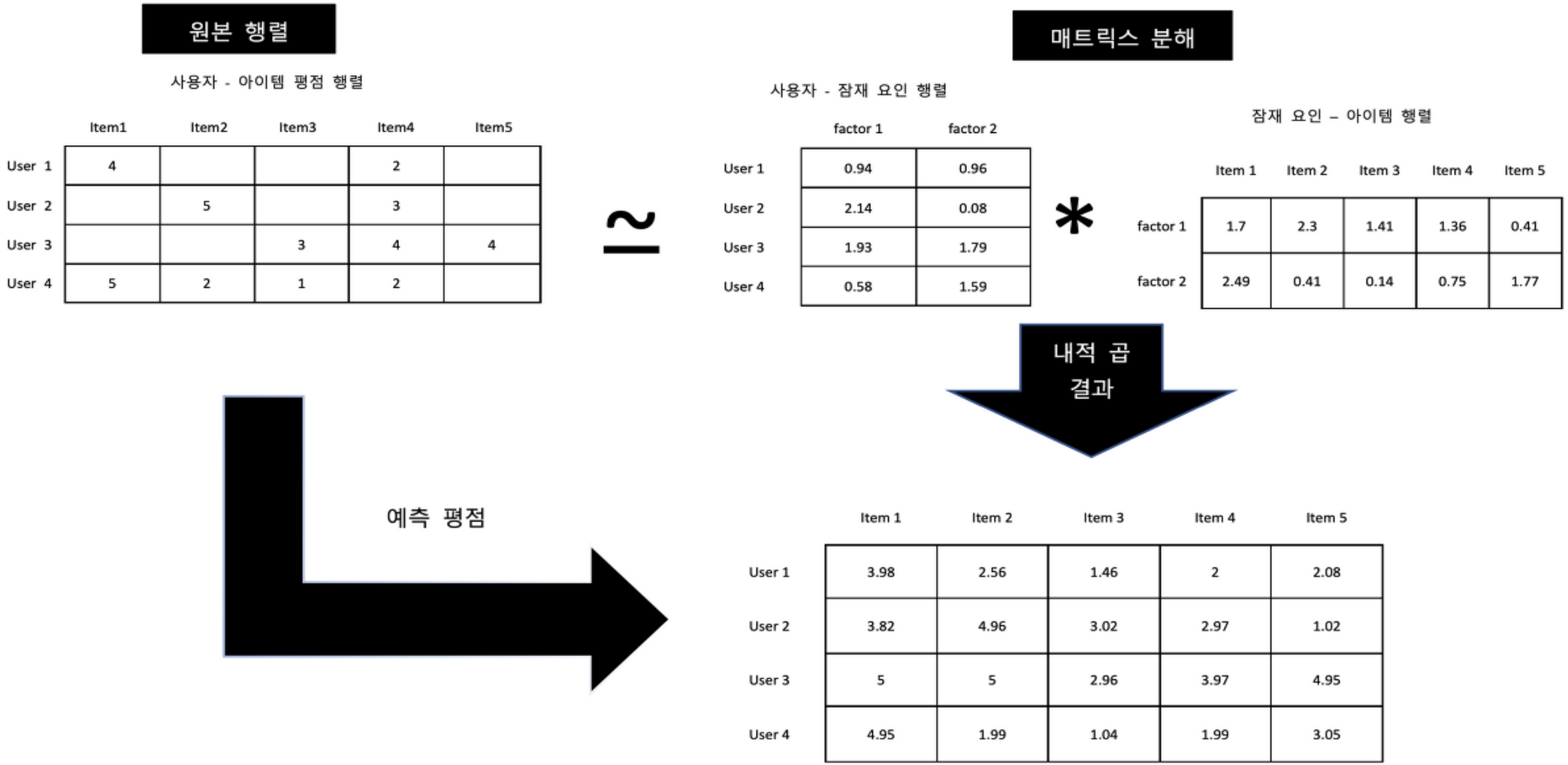
하나의 성분만을 사용한다면
NMF는 데이터를 가장 잘 표현할 수 있는
평균으로 향하는 성분을 만든다.

NMF의 활용

- 이미지 압축을 통한 패턴 인식
- 텍스트의 토픽모델링 기법
- 문서 유사도 및 클러스터링
- 추천 영역
  - 잠재요소(Latent Factoring) 기반의 추천방식
  - 예: 사용자의 상품 평가 데이터 세트인 사용자-평가 순위(user-Rating) 데이터 세트를 행렬 분해 기법을 통해 분해
    ->사용자가 평가하지 않은 상품에 대한 잠재적인 요소 추출
    ->평가순위(Rating) 예측 & 높은 순위로 예측된 상품 추천

NMF의 활용

- 추천 영역


원본 행렬
사용자 - 아이템 평점 행렬
Item1 Item2 Item3 Item4 Item5
User 1 4 2
User 2 5 3
User 3 3 4 4
User 4 5 2 1 2
≃
매트릭스 분해
사용자 - 잠재 요인 행렬
factor 1 factor 2
User 1 0.94 0.96
User 2 2.14 0.08
User 3 1.93 1.79
User 4 0.58 1.59
*
잠재 요인 – 아이템 행렬
Item 1 Item 2 Item 3 Item 4 Item 5
factor 1 1.7 2.3 1.41 1.36 0.41
factor 2 2.49 0.41 0.14 0.75 1.77
내적 곱 결과
예측 평점
Item 1 Item 2 Item 3 Item 4 Item 5
User 1 3.98 2.56 1.46 2 2.08
User 2 3.82 4.96 3.02 2.97 1.02
User 3 5 5 2.96 3.97 4.95
User 4 4.95 1.99 1.04 1.99 3.05

iris 데이터에 NMF 적용하기

- **NMF 실습**

```
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')
```