



# Aichemist Session

CHAP 06 차원축소

# CONTENTS

## 차원축소

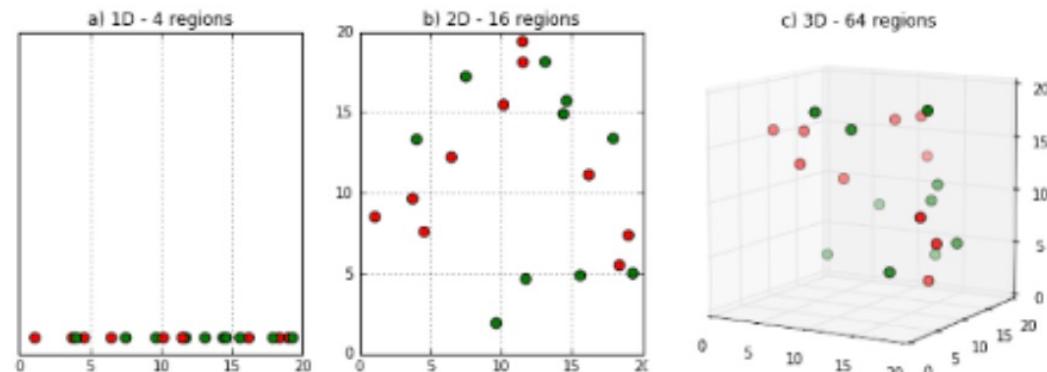
1. 차원 축소( Dimension Reduction )
2. PCA( Principal Component Analysis )
3. LDA( Linear Discriminant Analysis )
4. SVD( Singular Value Decomposition )
5. NMF( Non-Negative Matrix Factorization )

# 01. 차원축소

# 차원 축소

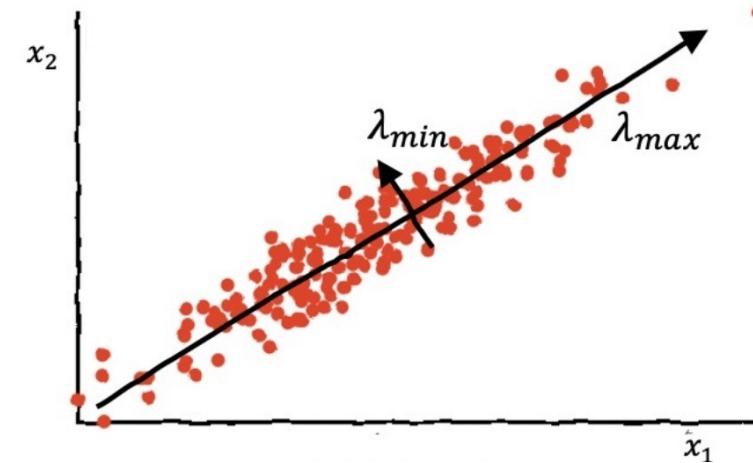
차원 축소 : 매우 많은 피처로 구성된 **다차원 데이터의 차원을 축소**해 새로운 차원의 데이터 세트를 생성하는 것

차원 축소가 필요한 이유? 차원 축소를 통해 데이터를 직관적으로 해석: 함축적 표현과 학습 처리량 감소



차원이 증가할수록 데이터간의 공간이 늘어남

- 희소한 구조를 가지게 됨
- 예측 성능 저하



차원이 증가할수록 개별 피처간의 상관 관계가 높아질 가능성이 커짐

- 예측 신뢰도 저하
- 예측 성능 저하
- 다중공선성 문제로 예측성능 저하

# 차원의 저주

고차원의 데이터에서 점들 사이의 공간이 늘어나 **데이터의 희소성이 증가**하여 예측 성능이 저하되는 현상

데이터 희소성 증가

데이터 포인트 사이의 거리 증가

→ **데이터 패턴** 파악 어려움

ex )

군집화나 분류 작업에서

데이터간의 유사성 평가가 어려워지면  
부정확한 결과값이 나타남

계산 복잡성 증가

데이터 차원의 개수가  $d$ 이고

각 차원이  $q$ 개의 구간을 가질 때

$q^d$ 에 비례하는 메모리가 필요

-> 고차원일수록 **데이터 처리 비용이**

기하급수적으로 증가함

과적합 위험 증가

차원이 높다는 것은

피처가 많다는 것을 의미하고

이는 피처간 상관관계 증가,

무의미한 피처가 결과에

영향을 미치는 문제로 인해

**과적합**으로 이어질 가능성을 높임

## 다중공선성 문제

피처들간에(=독립변수들간에) 강한 상관관계가(=선형관계가) 예측 성능에 부정적인 영향을 주는 문제

다중공선성 문제는 주로 회귀에서 중요하게 고려해줘야 하는 사항이기 때문에 회귀 분석 과정을 예로 들겠다.  
회귀 함수식을 구하는 과정을 요약하자면, 특정 피처를 제외한 나머지 피처들을 고정시킨 상태에서  
특정 피처의 결과값에 대한 영향력을 계산하여 회귀 계수를 부여하는 것이다.  
즉, 정확한 회귀 계수를 구하기 위해서는 특정 피처가 독립되어 있어야 한다는 것이다.  
이때 몇몇 피처들이 서로에게 큰 영향을 주고 있다면, 피처를 완전히 독립시키기가 어려워지고  
이에 따라 불안정한 회귀 계수를 도출해 예측 성능에 부정적인 영향을 주게 되는 것이다.



# 다중공선성 판단 기준

## 1. VIF(Variance Inflation Factor) - variance\_inflation\_factor(X, i)

1) 피처간 상관관계↑ → 회귀 계수의 분산↑

$$Var(b) = s^2 \frac{1}{\det(X^T X)} adj(X^T X)_{ij}$$

Var(b) : 회귀 계수 b에 대한 분산 / X : 회귀 계수 행렬

회귀 계수를 행렬로 나타냈을 때

두 피처가 선형 종속 관계에 가까울수록 (= 두 피처간의 상관관계가 클수록)

$\det$ 값이 0에 가까워짐

→ 회귀 계수의 분산 커짐

2) 회귀 계수의 분산↑ → 회귀 계수의 불안정성↑ → 예측 성능 저하

$$t = \frac{\hat{\beta}}{SE(\hat{\beta})}$$

위 식은 t-검정통계량 공식으로, 회귀 계수의 유의성 판별에 사용됨

t : t-검정통계량 /  $\beta$  : 회귀 계수 /  $SE(\beta)$  : 회귀 계수의 표준 오차

회귀 계수의 분산이 커질수록 t값이 작아짐 (표준오차  $\propto$  분산)

→ t값이 작아질수록 회귀 계수의 유의성이 작아짐

→ 예측 성능 저하

## 3) VIF 공식

$$Var(b_j) = \frac{s^2}{Var^2(X_j)} \frac{1}{1 - R_j^2}$$

분산 공식을 다른 방식으로 쓰면 왼쪽과 같이 된다

이때 분산과  $\frac{1}{1 - R_j^2}$  가 비례함을 알 수 있다

VIF는  $\frac{1}{1 - R_j^2}$  값을 이용해 분산을 파악하고

다중 공선성을 판단하는 데에 사용된다

VIF 값이 10 이상이면 다중 공선성을 의심해봐야 한다

피처간 상관관계, 분산, 예측 성능은 밀접한 관계가 있다



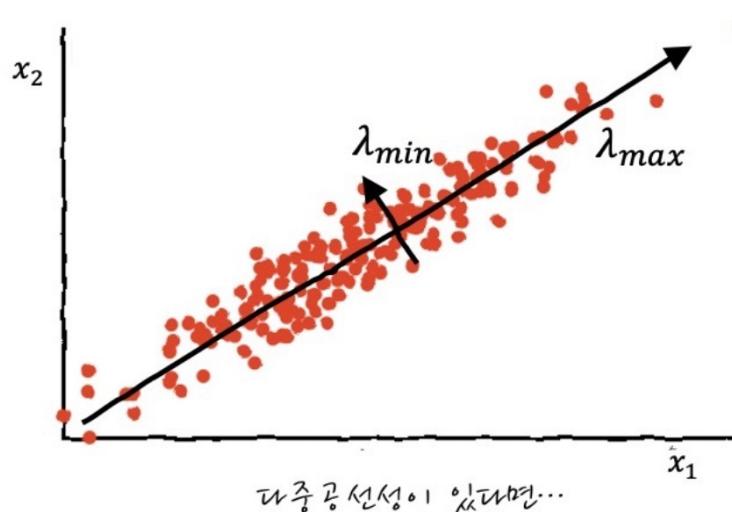
# 다중공선성 판단 기준

## 2. Condition Number - np.linalg.cond(x, p)

앞선 VIF 값이 특정 피처의 상관관계를 보여준다면 Condition Number는 모델 자체에 다중 공선성이 있는지 판별해준다

$$\text{Condition Number} = \frac{\lambda_{\max}}{\lambda_{\min}}$$

Condition Number는 회귀 분석에서 **공분산 행렬의 최대 고유값 / 최소고유값의 비율**을 의미한다



Condition Number가 크면 다중 공선성이 있다는 의미고  
데이터 분포가 왼쪽 그래프와 같이 **얇은 형태**로 나타난다  
왼쪽 그래프를 보면  $x_1$  과  $x_2$  가 **강한 선형 종속 관계**에 있음을 알 수 있다

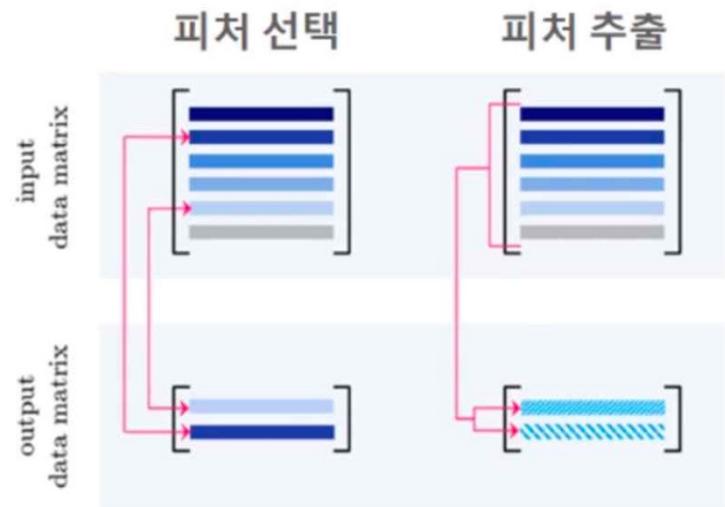
따라서 다르게 말하자면  $x_1$  과  $x_2$  가 높은 상관관계를 가질 때  
왼쪽 그래프와 같은 얇은 형태를 띠고  
Condition Number가 크게 나온다는 것이다

Condition Number가 30이상이면 다중 공선성을 의심해봐야 한다



# 차원 축소 방법

- **피처 선택** : 특정 피처에 종속성이 강한 불필요한 피처는 아예 제거하고, 데이터의 특징을 잘 나타내는 주요 피처만 선택하는 방법
- **피처 추출** : 기존 피처를 단순 압축이 아닌 피처를 함축적으로 더 잘 설명할 수 있는 또 다른 공간으로 매핑해 추출하는 방법  
기존 피처들이 나타내지 못한 잠재적인 요소를 추출함



차원 축소의 중요한 의미는 단순히 데이터의 압축이 아니라  
데이터를 잘 설명하는 잠재적 요소를 추출하는 데에 있다!



## 02. PCA

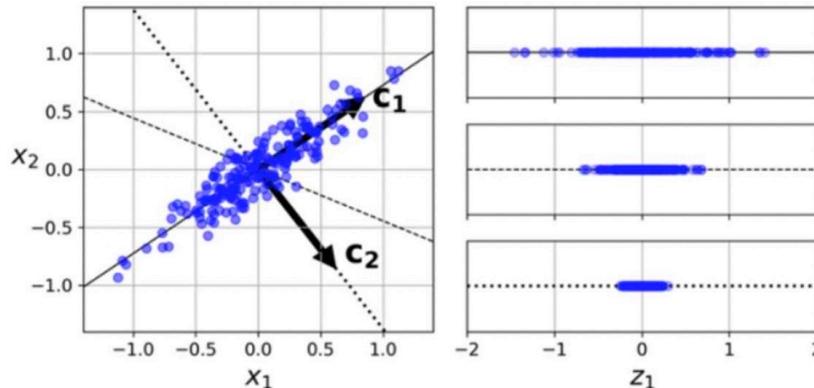
# PCA

principal component analysis

- 가장 대표적인 차원 축소 기법
- 여러 변수 간에 존재하는 상관 관계를 이용해 이를 대표하는 주성분을 추출해서 차원을 축소하는 기법
- 주성분 : 데이터의 변동성이 가장 큰 축 / 가장 높은 분산을 가지는 데이터의 축



정보의 유실을 최소화하기 위해 가장 중요한 것은 데이터 분포를 유지하는 것  
즉, 분산을 보존하는 것이다



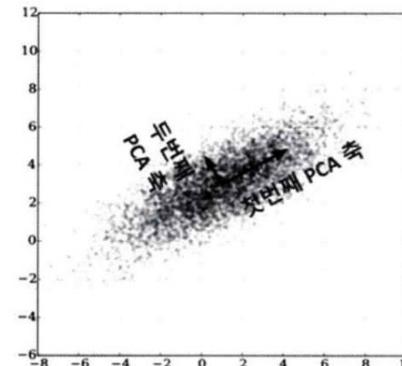
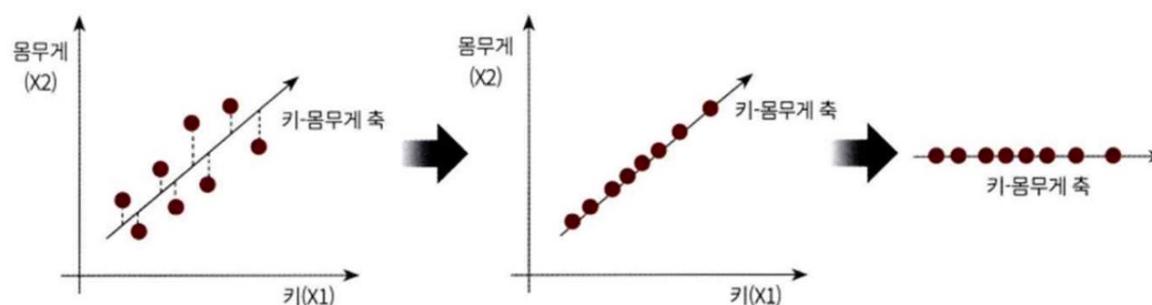
→ 분산을 최대로 보존하는 것은 첫번째 실선

→ 첫번째 실선을 첫번째 PCA축으로 설정

# PCA

## 주성분

PCA 작업이 이뤄지는 방법 : n차원  $\rightarrow$  d차원 가정



1. 훈련 세트에서 **분산**이 **최대인 축** 찾기  $\rightarrow$  첫번째 PCA 축
- 데이터  
투영 2. 1번 축에 **직교**하면서 남은 **분산**을 최대한 보존하는 두번째 축 찾기  $\rightarrow$  두번째 PCA 축
3. 2번 축에 직교하면서 남은 **분산**을 최대한 보존하는 세번째 축 찾기  $\rightarrow$  세번째 PCA 축
4. 위의 단계를 반복하면서 데이터셋에 있는 차원의 수만큼 d번째 축 찾기  $\rightarrow$  d번째 PCA 축
5. 주성분을 모두 추출한 후 d개의 PCA축들에 **원본 데이터를 투영**하면 d차원으로 차원이 축소된다

이 때 i번째 축을 이 데이터의 i번째 **주성분(PC)**이라고 부른다

# 선형 대수 관점에서 PCA

앞서 말했듯이 주성분은 데이터의 분산이 가장 큰 축으로 설정한다

그렇다면 데이터의 분산이 가장 큰 축은 어떻게 구하는 것인지 선형대수 관점에서 설명하겠다

“PCA는 입력 데이터의 공분산 행렬을 고유값 분해하고 분해된 고유벡터에 대해 입력데이터를 선형 변환하는 방식”

	선형대수에서 개념	PCA에서 의미하는 바
선형 변환	$Av = b$ 와 같이 선형 결합을 보존하는, 두 벡터 공간 사이의 함수	$N$ 차원을 $D$ 차원으로 투영시키는 것
고유 벡터	$Av = \lambda v$ 를 만족하는 0이 아닌 벡터 $v$ 변환을 해도 방향이 바뀌지 않음	주성분 벡터 (PCA 축) 입력 데이터의 분산이 큰 방향
고유 값	$Av = \lambda v$ 를 만족하는 0이 아닌 상수 $\lambda$	주성분 벡터의 입력 데이터의 분산
고유값 분해	고유값과 고유벡터를 찾는 작업	

# 선형 대수 관점에서 PCA

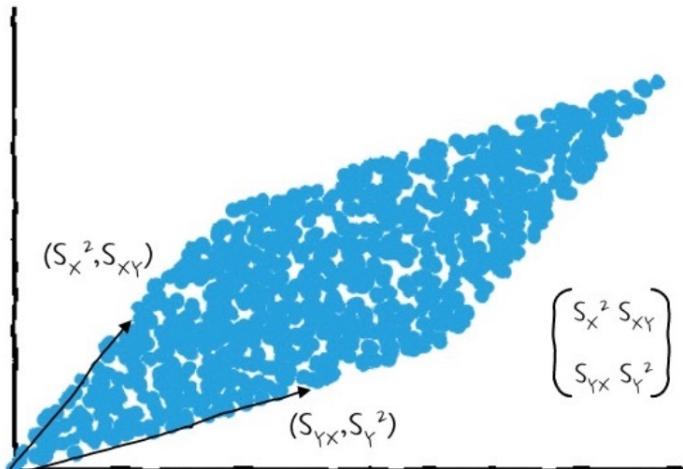
1) 입력 데이터를 공분산 행렬로 표현

공분산 행렬

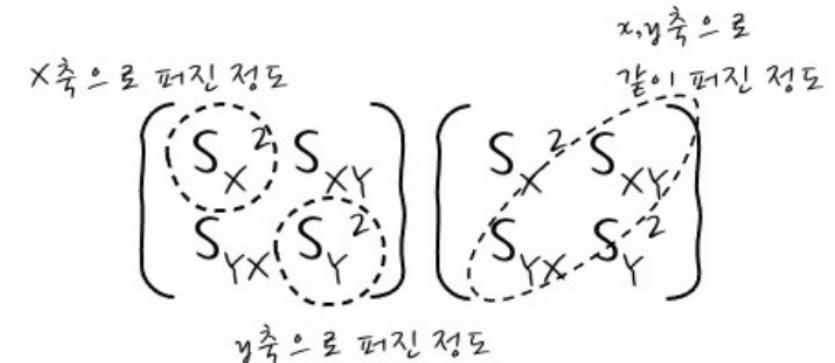
: 데이터의 좌표 성분들 사이의 **공분산** 값을 원소로 하는 행렬

**정방행렬**이며,  $C = C^T$  인 **대칭 행렬**임

공분산: 두 변수 간의 **변동**, 하나의 변수가 증감할 때 다른 하나의 변수의 증감의 경향



모든 데이터는 공분산 행렬의 기저 -  $(S_x^2, S_{xy}) / (S_y^2, S_{yx})$  에 인한  
선형 조합의 합으로 표현 가능하다



# 선형 대수 관점에서 PCA

## 2 ) 공분산 행렬 고유값 분해

$C = N$  차원의 대칭행렬

$V = [e_1 \dots e_n]$  고유 벡터 행렬

$\Lambda = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_N \end{bmatrix}$  고유값 행렬

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

공분산 행렬은

고유벡터와 고유값으로  
로 분해됨

분해된 고유벡터를 이용해 입력 데이터를 선형 변환하는 방식이 PCA

- 공분산 행렬의 고유 벡터는 행렬이 작용하는 주축의 방향을 나타낸다. 즉, 데이터가 가장 많이 분산되어 있는 방향을 나타낸다 → PCA 축
- 공분산 행렬의 고유 벡터에 해당하는 고유값은 그 방향으로의 크기를 나타낸다 → 분산
- 고유값이 가장 큰 고유벡터가 분산이 가장 큰 데이터 축, 첫번째 PCA 축이 된다
- $N$  차원의 공분산 행렬은  $N$  개의 고유벡터와 고유값을 갖는다  
→  $D$  차원으로 축소하고 싶다면 크기순으로  $D$  개의 고유값을 도출하고 각각의 고유벡터를 PCA 축으로 삼아 원본 데이터를 투영하면 된다
- 공분산 행렬은 대칭 행렬이자 정방 행렬이기 때문에  $n$  번째 PCA 축과  $n+1$  번째 PCA 축이 서로 직교 관계다

# PCA 구현 순서

1. 입력 데이터 세트의 **공분산 행렬**을 생성
2. 공분산 행렬의 **고유벡터** 와 **고유값** 계산
3. 고유값이 큰 순서대로 d개만큼의 고유벡터 추출
4. 고유벡터를 이용해서 새롭게 입력 데이터 변환

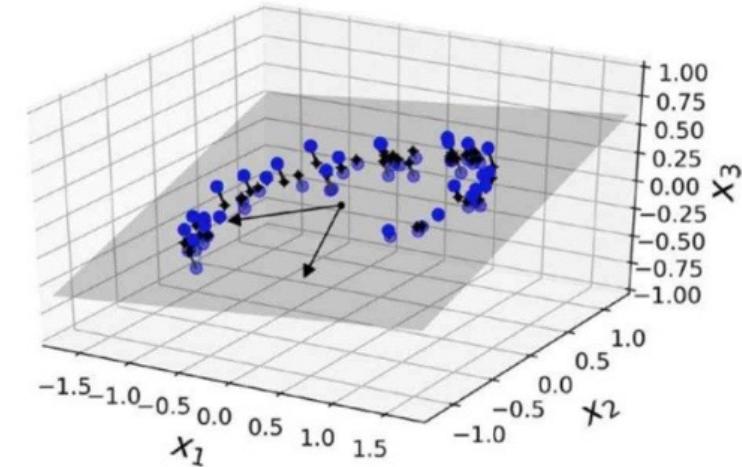
$$\mathbf{X}_{d\text{-proj}} = \mathbf{X} \mathbf{W}_d$$

d차원으로 축소된 데이터셋은

기존 데이터 행렬  $\mathbf{X}$ 와 주성분 단위 벡터의 행렬 곱으로 얻을 수 있음

```
from sklearn.decomposition import PCA  
  
pca = PCA(n_components = 2)  
X2D = pca.fit_transform(X)
```

sklearn에서 PCA를 적용하여 데이터셋을 2차원으로 줄이는 코드



# 붓꽃 품종 예측 - PCA 구현

붓꽃 데이터 불러온 뒤, 데이터가 어떻게 분포되었는지 2차원으로 시각화

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

# 사이킷런 내장 데이터셋 API 호출
iris = load_iris()

# 네파이 데이터셋을 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

평균이 0, 분산이 1인 정규 분포로 속성값 변환

```
from sklearn.preprocessing import StandardScaler

# Target 값을 제외한 모든 속성 값을 StandardScaler를 이용하여 표준 정규 분포를 가지는 값들로 변환
iris_scaled = StandardScaler().fit_transform(irisDF.iloc[:, :-1])

iris_scaled.shape
```

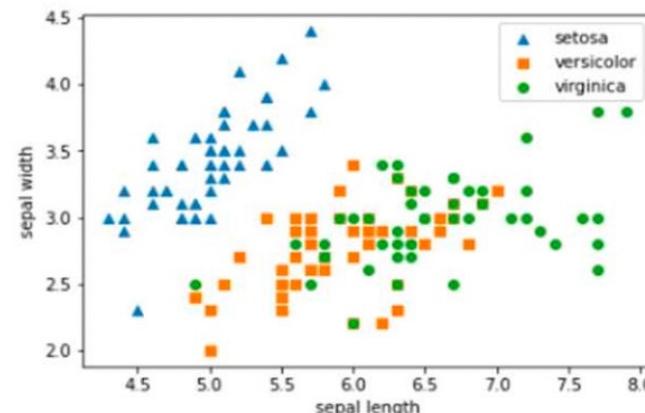
(150, 4)

데이터가 4차원임을 확인

```
# 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()
```



→ setosa는 명확히 구분되지만  
versicolor와 virginica는  
서로 겹치는 부분이 꽤 큼

# 붓꽃 품종 예측 - PCA 구현

4차원의 데이터를 2차원 PCA 데이터로 변환하고 dataframe으로 데이터 값 확인

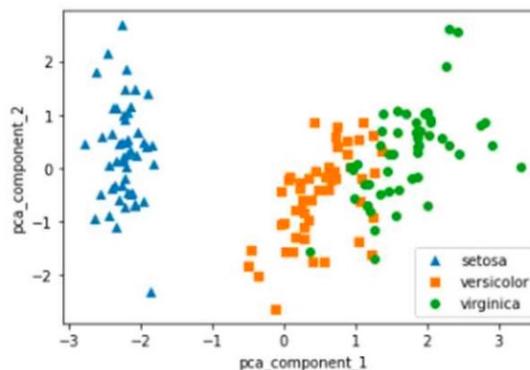
```
from sklearn.decomposition import PCA  
pca = PCA(n_components=2)  
  
#fit()과 transform()을 호출하여 PCA 변환 데이터 받는  
pca.fit(iris_scaled)  
iris_pca = pca.transform(iris_scaled)  
print(iris_pca.shape)  
  
(150, 2) → 데이터가 2차원으로 변환됨
```

```
# PCA 변환 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

2차원으로 변환된 데이터 시각화

```
#setosa 세모, versicolor 네모, virginica 등그라미로 표시  
markers=['^', 's', 'o']  
  
#pca_component_1은 x축, pca_component_2는 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()
```



→ 차원 축소하기 전보다  
pca\_component\_1 축을 기반으로  
setosa는 완벽하게 분류되었고  
versicolor와 virginica는  
서로 겹치는 부분이 많이 줄어듦

# 붓꽃 품종 예측 - PCA 구현

원본 붓꽃 데이터에 랜덤 포레스트를 적용한 결과

```
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 Component별 원본 데이터의 변동성 반영 비율

```
print(pca.explained_variance_ratio_)

[ 0.72962445, 0.22850762 ]
```

pca\_component\_1 : 약 72.9% , pca\_component\_2 : 약 22.8%  
→ 2개의 PCA 축으로도 약 95% 의 원본 데이터의 변동성을 설명할 수 있음



4개의 속성이 2개의 변환 속성으로 감소한 것을 고려하면

PCA 변환 후에도 원본 데이터의 특성을 상당 부분 유지하고 있음을 알 수 있음

# 신용카드 데이터 분석 - PCA 구현

## 데이터 로드 및 타겟값 / 피처값 설정

```
# header로 의미없는 첫행 제거, iloc로 기존 id 제거
import pandas as pd
pd.set_option('display.max_columns', 30)

df = pd.read_excel('pca_credit_card.xls', header=1, sheet_name='Data').iloc[:,1:]
print(df.shape)
df.head(3)
```

(30000, 24)

LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	PAY_5	PAY_6	BILL_AMT1	BILL_AMT2	BILL_AMT3	BILL_AMT4	BILL_AMT5	BILL_AMT6	PAY_AMT1
20000	2	2	1	24	2	2	-1	-1	-2	-2	3913	3102	689	0	0	0	0
120000	2	2	2	26	-1	2	0	0	0	2	2682	1725	2682	3272	3455	3261	0
90000	2	2	2	34	0	0	0	0	0	0	29239	14027	13559	14331	14948	15549	1518

```
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)
```

# 신용카드 데이터 분석 - PCA 구현

In [15]: 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

-> 총 23개의 features

고차원 데이터이므로

차원 축소 고려

# 신용카드 데이터 분석 - PCA 구현

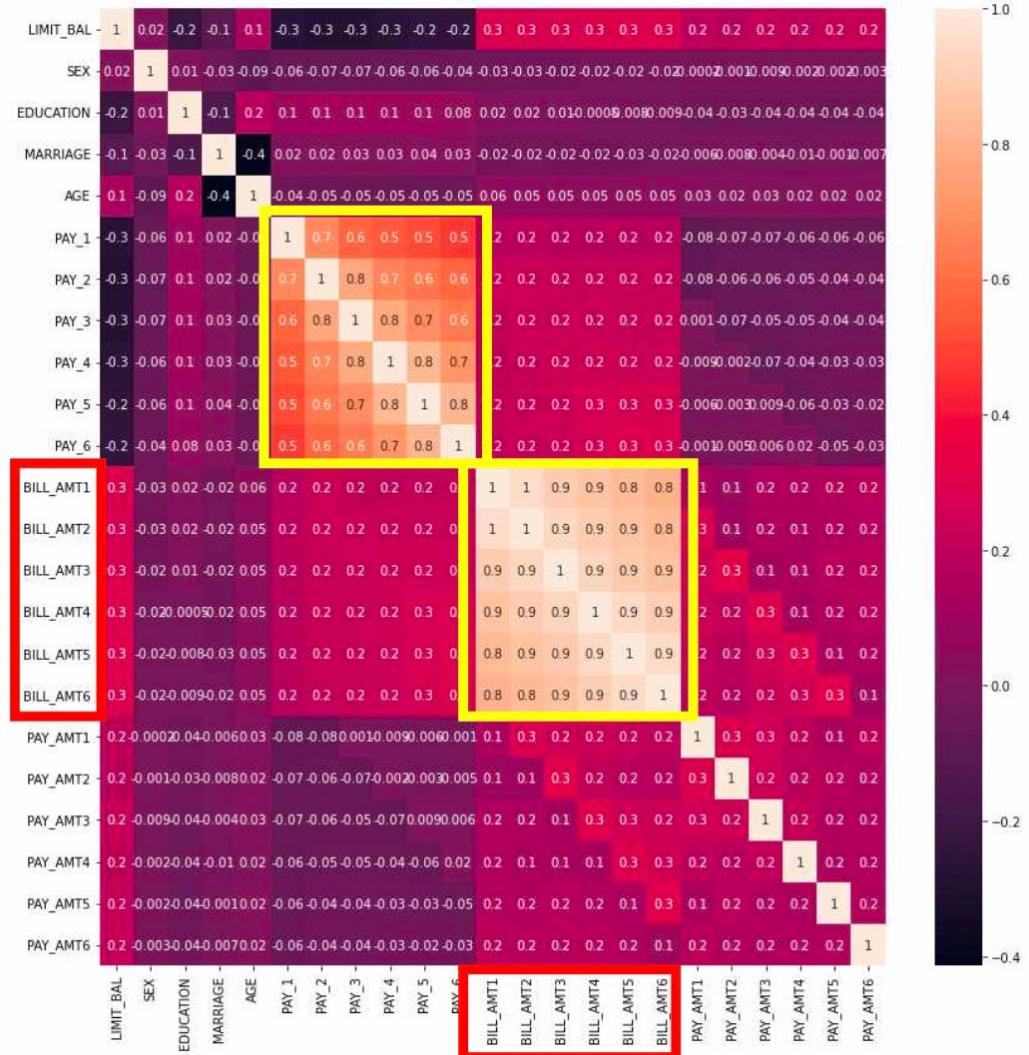
## Feature 간 상관도 시각화

```
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')
plt.show()
```

상관도를 보았을 때 PAY끼리,  
BILL\_AMT끼리, 특히 BILL끼리  
상관도가 높음을 볼 수 있다



# 신용카드 데이터 분석 - PCA 구현

BILL\_AMT1 ~ BILL\_AMT6를 두 개의 피처로 차원 축소

In [20]:

```
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) → 데이터를 2차원으로 변환
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 ]

단 2개의 PCA 컴포넌트만으로도 6개 속성의 변동성을 약 95% 이상 설명할 수 있으며

특히 첫 번째 PCA축으로 90%의 변동성을 수용할 수 있을 정도로 이 6개 속성의 상관도가 매우 높음

# 신용카드 데이터 분석 - PCA 구현

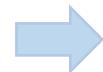
23차원  $\rightarrow$  6차원으로 차원 축소 후 분류 성능 평가

```
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: {:.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 StandardScaler 적용
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)
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: {:.4f}'.format(np.mean(scores_pca)))
```

CV=3 인 경우의 PCA 변환된 개별 Fold세트별 정확도: [0.793 0.7958 0.8026]  
PCA 변환 데이터 셋 평균 정확도: 0.7971

전체 23개 속성이 약 1/4 수준인 6개의 PCA 컴포넌트만으로  
분류 예측의 1~2% 정도의 예측 성능 저하만 발생

# 03. LDA



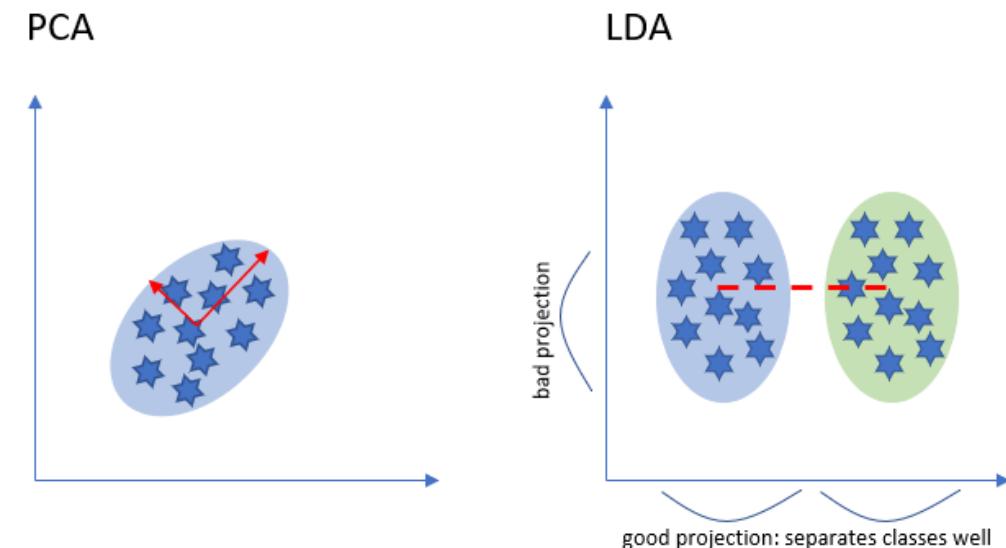
# LDA

## LDA

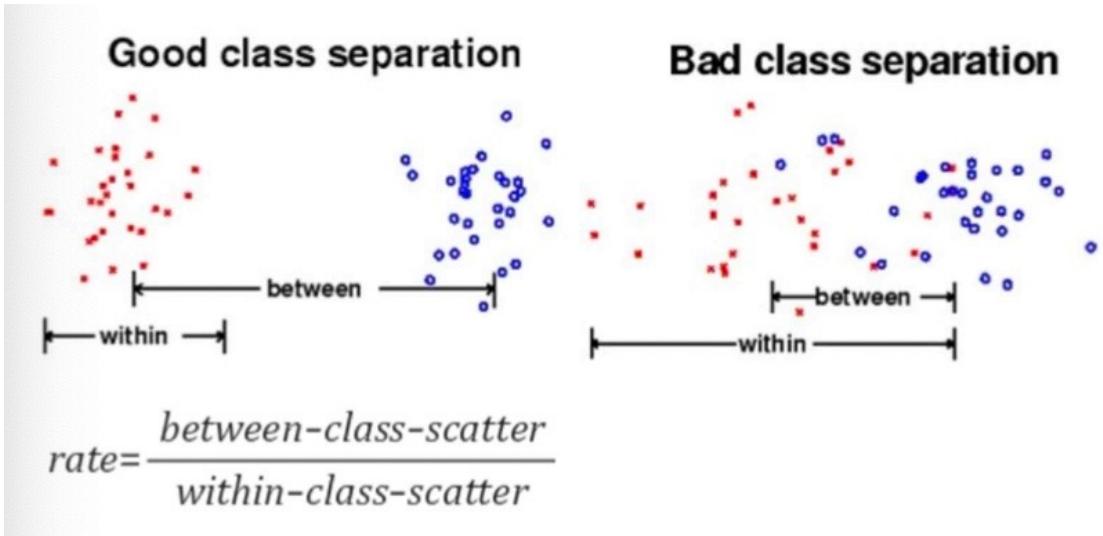
- 선형 판별 분석으로 불림
- PCA와 유사하게 저차원 공간에 데이터세트를 투영해 차원을 축소함

## LDA vs PCA

- LDA  
입력 데이터의 결정값 클래스를 최대한으로 분리할 수 있는 축을 찾음  
-> 분류에서 사용하기 쉽도록
- PCA  
입력 데이터의 변동성이 가장 큰 축을 찾음



# LDA의 축 결정 방식



특정 공간상에서 클래스 분리를 최대화하는 축을 찾기 위해  
클래스 간 분산과 클래스 내부 분산의 비율을  
최대화하는 방식으로 차원 축소  
클래스 간 분산은 최대한 (크게 / 작게),  
클래스 내부 분산은 최대한 (크게 / 작게) 만들수록  
좋은 클래스 분리임

## 선형 대수 관점에서 LDA

1) 클래스 내부와 클래스간 분산 행렬 구하기

2) 두 행렬을 고유값 분해

$$S_W^T S_B = [e_1 \dots e_n] \begin{bmatrix} \lambda_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_n \end{bmatrix} \begin{bmatrix} e_1^T \\ \vdots \\ e_n^T \end{bmatrix}$$

$S_W$  : 클래스 내부 분산 행렬  
 $S_b$  : 클래스 간 분산 행렬

3) n차원으로 줄이고 싶다면 크기순으로 n개의 고유값과 고유벡터를 이용해 LDA 축을 생성

4) LDA 축에 원본 데이터 투영

# 붓꽃 품종 예측 - LDA

데이터세트, LDA 로드 및 정규화

```
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)
```

LDA는 LinearDiscriminantAnalysis 클래스로 제공됨

LDA를 이용해 2차원으로 차원 축소

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

(150, 2) → 데이터 2차원으로 변환됨

분류에 사용되고 있으므로 결정값 필요  
fit()에서 결정값인 iris.target 입력해야 함

# 붓꽃 품종 예측 - LDA

## LDA로 차원 축소 후 분류 수행 및 시각화

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

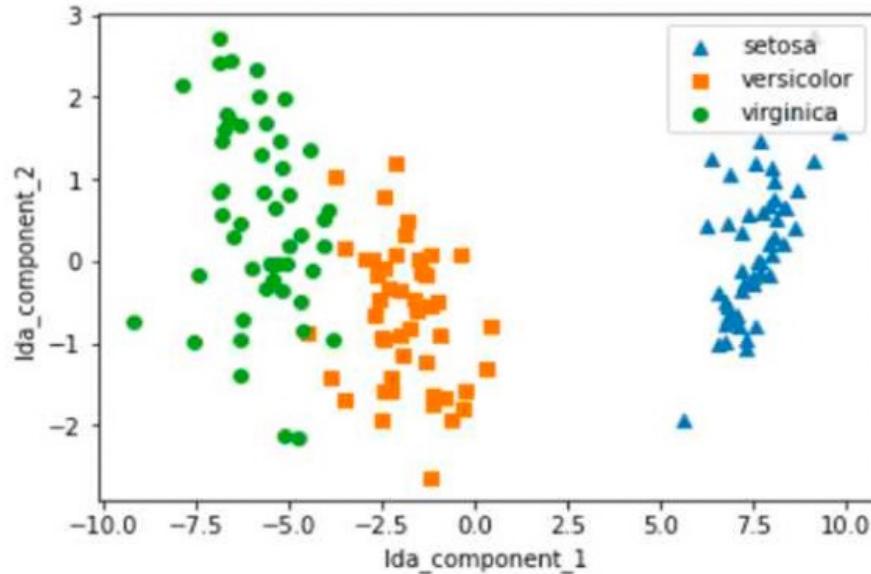
lda_columns=['lda_component_1','lda_component_2']
irisDF_lda = pd.DataFrame(iris_lda, columns=lda_columns)
irisDF_lda['target']=iris.target

# setosa는 세모, versicolor는 네모, virginica는 동그라미로 표현
markers=['^', 's', 'o']

# setosa의 target 값은 0, versicolor는 1, virginica는 2, 각 target 별로 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()
```



setosa는 lda\_component\_1을 기준으로 명확히 구분되었고  
versicolor와 virginica도 서로 겹치는 부분이 있으나 잘 구분되었음

PCA와 좌우 대칭 형태로 닮아있음

# 04. SVD

# SVD

$$A = \overset{\bullet}{U} \Sigma V^T$$

SVD : 특이값 분해, 고유값 분해와 달리 정방행렬뿐만 아니라 행과 열의 크기가 다른 행렬 분해가 가능한 기법

- U: mxm orthogonal matrix(singular)
- V: nxn orthogonal matrix(singular)
- Sigma: mxn diagonal matrix
- A: mxn rectangular matrix

orthogonal matrix 특징  
 $\overset{\bullet}{U} U^T = U^T U = I$

diagonal matrix 특징  
대각 원소 제외하고 모두 0

A의 차원이 mxn일 때 U, sigma, V로 분해

$$\boxed{A} = \boxed{U} \begin{pmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_s \\ & & 0 \end{pmatrix} \boxed{V^T}$$

# SVD

앞선 PCA에서 고유값 분해를 할 때 크기순으로 고유값과 고유벡터를 선별해 차원 축소를 진행했다  
이는 원본 데이터의 변동성이 95% 정도 유지되는 정도라면 축을 일부 걸러내도 된다는 뜻이다

$$A = U \begin{pmatrix} S \\ VT \end{pmatrix}$$

Matrix sizes:  
A:  $m \times n$   
U:  $m \times p$   
 $S$ :  $p \times p$   
VT:  $p \times n$

Sigma에서 특이값은 모두 제거

이에 대응하는 U와 VT의 원소도 제거  
-> 다중공선성이 높은 피처

$$A = U \begin{pmatrix} S \\ VT \end{pmatrix}$$

Matrix sizes:  
A:  $m \times n$   
U:  $m \times r$   
 $S$ :  $r \times r$   
VT:  $r \times n$

Truncated SVD

: sigma의 대각원소 중 상위 몇 개만 추출하여  
이에 대응하는 U, V의 원소도 함께 제거해 차원을 줄인 형태로 분해  
이때 r이 축소하고 싶은 차원 수를 의미한다

# SVD

## SVD로 차원 축소 구현

```
# 넘파이의 svd 모듈 임포트
import numpy as np
from numpy.linalg import svd
```

```
# 4x4 랜덤 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:', np.round(U,3))
print('Sigma Value:', np.round(Sigma,3))
print('V transpose matrix:', 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]]
```

numpy.linalg.svd에 파라미터로 원본 행렬을 입력하면 U 행렬, Sigma 행렬, V 전치 행렬 반환  
Sigma 행렬은 행렬의 대각에 위치한 값만 0이 아닌 값이므로 그 값만 1차원 행렬로 표현

# SVD

## 차원 축소했던 행렬 복원

```
# Sigma를 다시 0을 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]]
```

```
a[2]=a[0]+a[1]          → 로우 간 의존성 부여  
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 수행해 Sigma 값 확인  
U, Sigma, Vt = svd(a)  
print(U.shape, Sigma.shape, Vt.shape)  
print('Sigma Value:', 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]]
```

## 로우 간 의존성 X

원본 행렬로의 복원은 U, Sigma,  $V^T$ 의 **내적**을 통해 수행한다  
Sigma는 0을 포함한 대칭행렬로 변환 뒤 내적을 수행해야 한다.

→  $a_$ 는 원본 행렬  $a$ 와 동일하게 복원됨.

## 로우 간 의존성 O

→ 다시 SVD로 분해한 결과 이전과 차원은 같지만  
sigma 값 중 2개가 0으로 바뀜

## Sigma 값이 0이 나온 데이터를 제외하고 복원 진행

→  $a_$ 는 원본 행렬  $a$ 와 동일하게 복원됨

## 원본 a

```
[[ -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]]
```

# SVD

## TruncatedSVD로 차원 축소

```
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 :', 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:', matrix_tr)
```

- `scipy.sparse.linalg.svds` 이용
- 임의의 원본 행렬 6x6을 normal SVD로 분해해 특이값 확인 후 특이값을 4개로 하여 Truncated SVD로 분해 후 복원된 데이터와 원본 데이터 비교
- Truncated SVD로 분해된 행렬의 sigma 형태가 (6,)가 아닌 (4,)임을 확인

```
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]]
```

- 원본 행렬을 정확하게 다시 복원할 수는 없으나 상당한 수준으로 원본 행렬 근사 가능
- 원래 차원의 차수에 가깝게 잘라낼수록 원본 행렬에 더 가깝게 복원 가능

# SVD

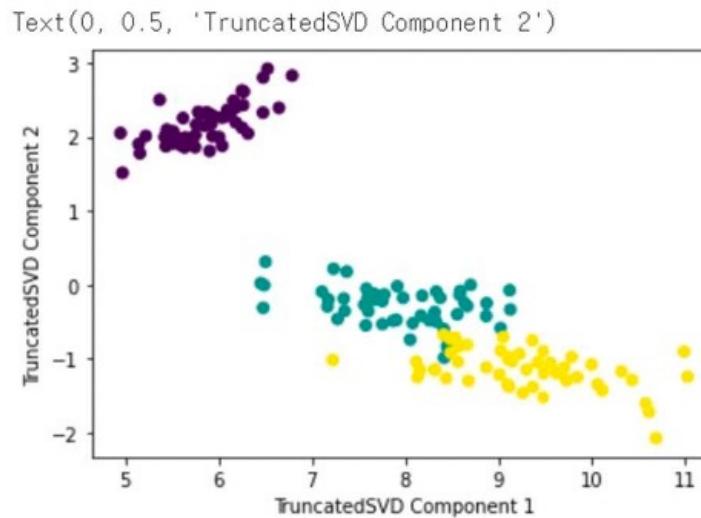
## 사이킷런 TruncatedSVD 클래스를 이용한 변환

```
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 컴포넌트로 TruncatedSVD 변환
tsvd = TruncatedSVD(n_components=2)
tsvd.fit(iris_ftrs)
iris_tsvd = tsvd.transform(iris_ftrs)

# 산점도 2차원으로 TruncatedSVD 변환된 데이터 표현. 품종을 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')
```

- 사이킷런의 TruncatedSVD 클래스는 원본 행렬을 분해한 뒤 U, Sigma, Vt 행렬 반환하지 않음
- PCA 클래스와 유사하게 fit(), transform()을 호출해 원본 데이터를 몇 개의 주요 컴포넌트로 차원 축소해 변환
- 원본 데이터를 TruncatedSVD 방식으로 분해된  $U^*Sigma$  행렬에 선형 변환해 생성

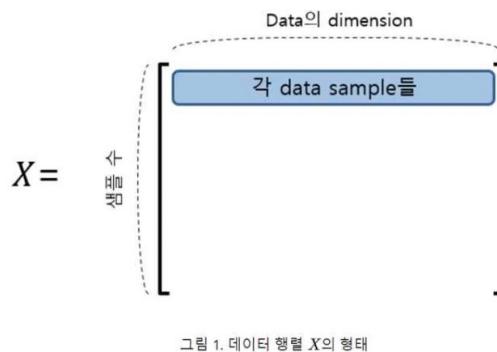


- 붓꽃 데이터 세트를 TruncatedSVD를 이용해 변환
- PCA와 유사하게 변환 후 품종별로 어느 정도 클러스터링이 가능할 정도로 각 변환 속성으로 뛰어난 고유성을 가지고 있음

# 05. NMF



# NMF

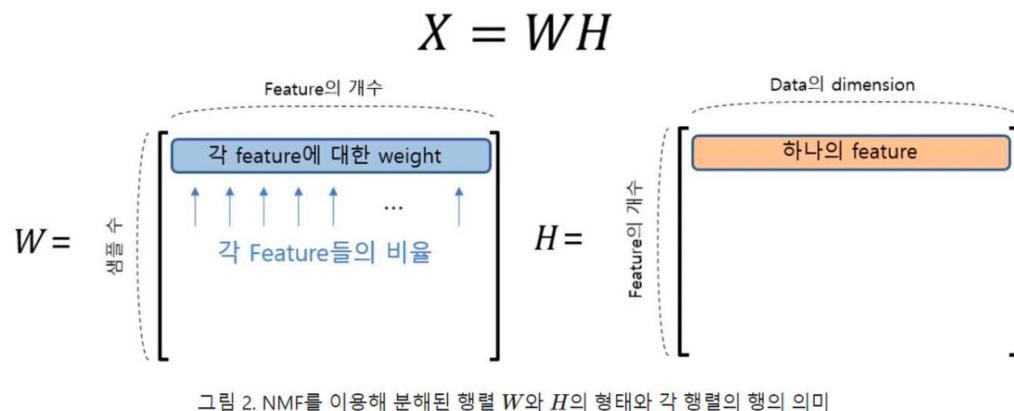


NMF: 음수 미포함 행렬 분해로 음수를 포함하지 않는 행렬  $X(V)$ 를 행렬  $W$ 와  $H$ 의 곱으로 분해하는 기법

행렬 분해(Matrix Factorization):  
일반적으로 SVD와 같은 행렬 분해 기법 통칭

일반적으로 길고 가는 행렬  $W$ , 작고 넓은 행렬  $H$ 로 분해됨.  
원본 행렬이  $m \times n$ 이고  $W$ 가  $m \times p$ ,  $H$ 가  $p \times n$ 이면  $p$ 차원으로 차원 축소

분해된 행렬은 잠재 요소를 특성으로 가짐.  
 $W$ 는 원본 행에 대해 이 잠재 요소의 값이 얼마나 되는지에 대응  
 $H$ 는 잠재 요소가 원본 열로 어떻게 구성됐는지 나타내는 행렬



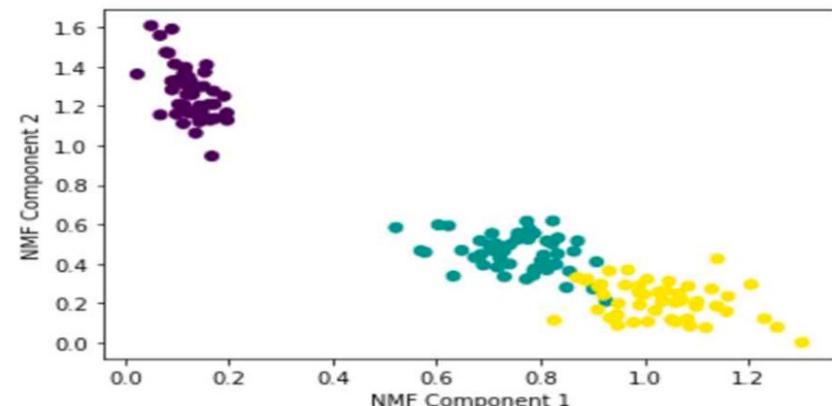
# NMF

사이킷런에서 NMF는 NMF 클래스를 통해 지원됨  
붓꽃 데이터를 NMF를 이용해 2개의 컴포넌트로 변환하고 시각화한 결과

```
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')

/usr/local/lib/python3.7/dist-packages/sklearn/decomposition/_  
FutureWarning,  
/usr/local/lib/python3.7/dist-packages/sklearn/decomposition/_  
ConvergenceWarning,  
Text(0, 0.5, 'NMF Component 2')
```



수고하셨습니다