



# Week 10 발표

차수빈, 황선경, 소예림

# 목차

---

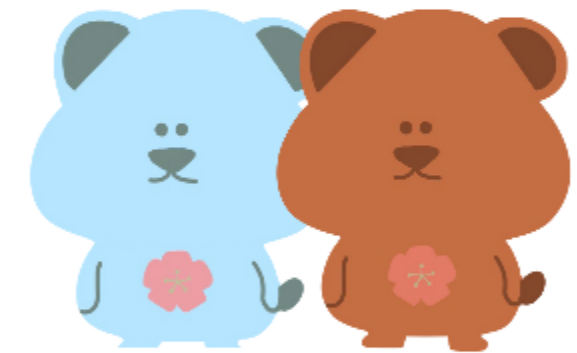
#01 와인 품질 예측 노트북(classification)

#02 이미지 데이터 차원축소 mnist

#03 차원축소 기법들



## 와인 품질 예측 노트북 (classification)



# #1.1 대회 소개

## #1 와인 품질 예측 시스템이란?

- 현재 와인 품질 예측 시스템
  - 제품 홍보를 위해 제품 품질 인증을 사용
    - 시간, 비용이 많이 소모됨
    - 와인이 책정되는 가격은 시음자의 추상적인 평가에 달려있음

➔ 인증 및 품질 평가와 보증 과정이 보다 통제될 수 있도록 와인의 여러 화학적 특성을 바탕으로 와인의 품질을 예측하고 분류 모델을 만드는 것이 목표

# #1.2 Data Description

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1599 entries, 0 to 1598
Data columns (total 12 columns):
#   Column                Non-Null Count  Dtype
---  -
0   fixed acidity         1599 non-null   float64
1   volatile acidity      1599 non-null   float64
2   citric acid           1599 non-null   float64
3   residual sugar        1599 non-null   float64
4   chlorides             1599 non-null   float64
5   free sulfur dioxide   1599 non-null   float64
6   total sulfur dioxide  1599 non-null   float64
7   density              1599 non-null   float64
8   pH                   1599 non-null   float64
9   sulphates            1599 non-null   float64
10  alcohol              1599 non-null   float64
11  quality              1599 non-null   int64
dtypes: float64(11), int64(1)
memory usage: 150.0 KB
```

- 1. fixed acidity (float)    고정 산도
- 2. volatile acidity (float)    휘발성 산도
- 3. citric acid (float)    시트르산
- 4. residual sugar (float)    잔류 설탕
- 5. chlorides (float)    염화물
- 6. free sulfur dioxide (float)    독립 이산화황
- 7. total sulfur dioxide (float)    총 이산화황
- 8. density (float)    밀도
- 9. pH (float)    수소이온농도
- 10. sulphates (float)    황산염
- 11. alcohol (float)    도수
- 12. quality (int)    품질 (0에서 10 사이의 점수)

➔ 12개 변수와 1599개의 관측치 존재

# #1.2 Data Description

## 1. Duplicated values 삭제

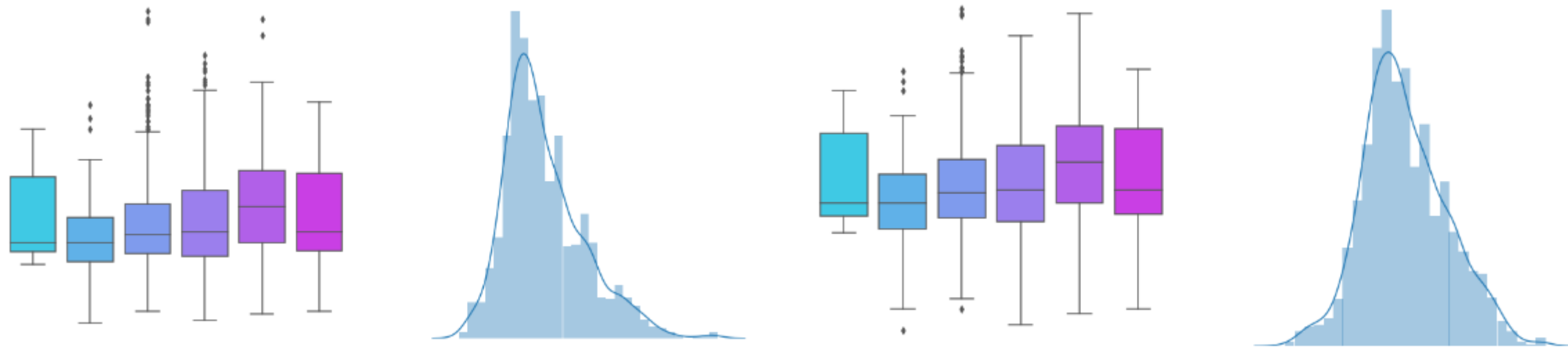
```
#Finding the duplicates from dataset...  
df.duplicated().sum()  
#Removing all the duplicated records  
df.drop_duplicates(inplace=True)
```

```
<class 'pandas.core.frame.DataFrame'>  
Int64Index: 1359 entries, 0 to 1598  
Data columns (total 12 columns):  
#   Column                Non-Null Count  Dtype  
---  -  
0   fixed acidity          1359 non-null   float64  
1   volatile acidity       1359 non-null   float64  
2   citric acid            1359 non-null   float64  
3   residual sugar         1359 non-null   float64  
4   chlorides              1359 non-null   float64  
5   free sulfur dioxide    1359 non-null   float64  
6   total sulfur dioxide   1359 non-null   float64  
7   density                1359 non-null   float64  
8   pH                    1359 non-null   float64  
9   sulphates              1359 non-null   float64  
10  alcohol                1359 non-null   float64  
11  quality                1359 non-null   int64  
dtypes: float64(11), int64(1)  
memory usage: 138.0 KB
```

240개의 관측치 삭제

# #1.2 Feature Analysis & Finding Outliers

#1 Distribution of the fixed Acidity before & after treating outlier



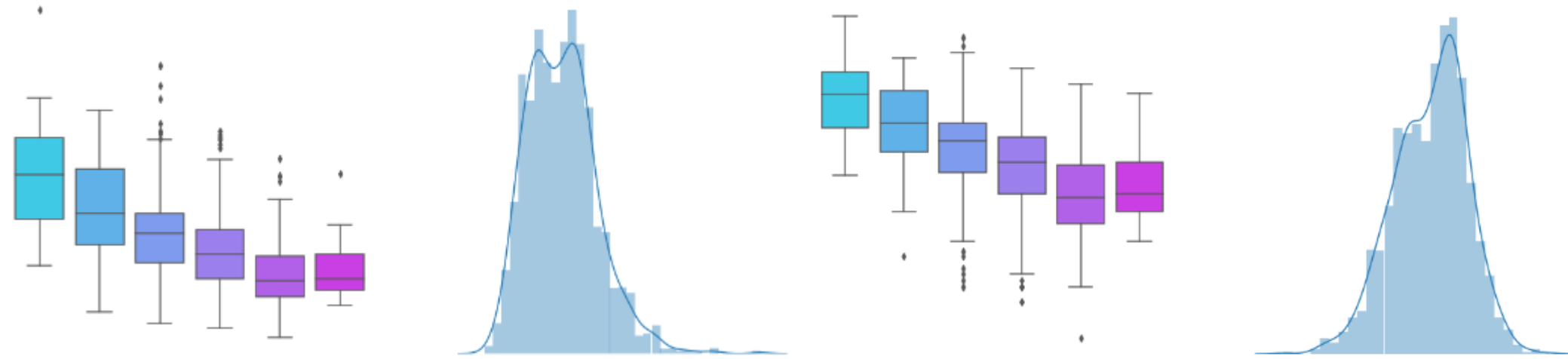
```
df["Log_fixed acidity"] = np.log(df["fixed acidity"])
```

outlier로 분포가 편중됨 → 변수에 대해 log transformation 실행

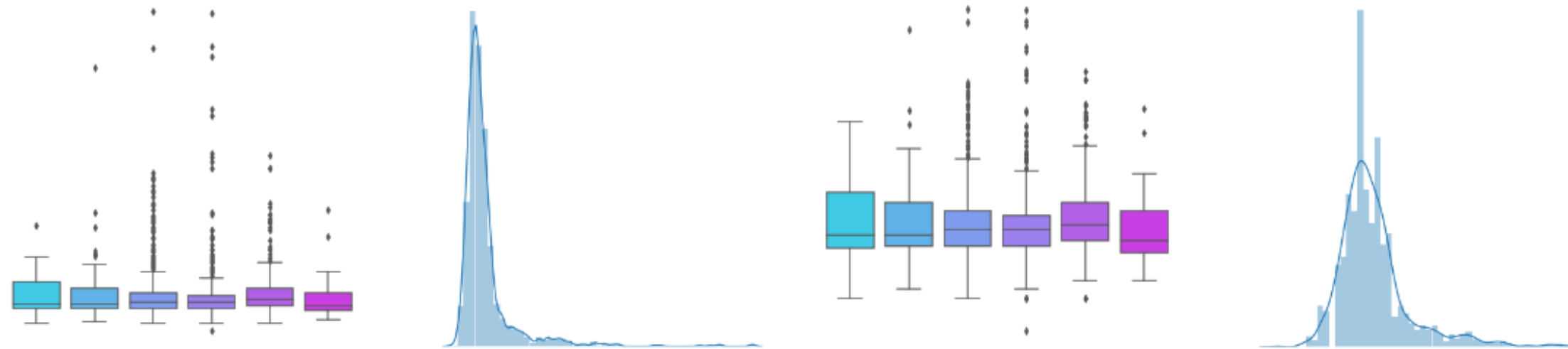
→ outlier와 분포의 편중이 완화된 것을 확인

# #1.2 Feature Analysis & Finding Outliers

#2 Distribution of the Volatile Acidity before & after treating outlier



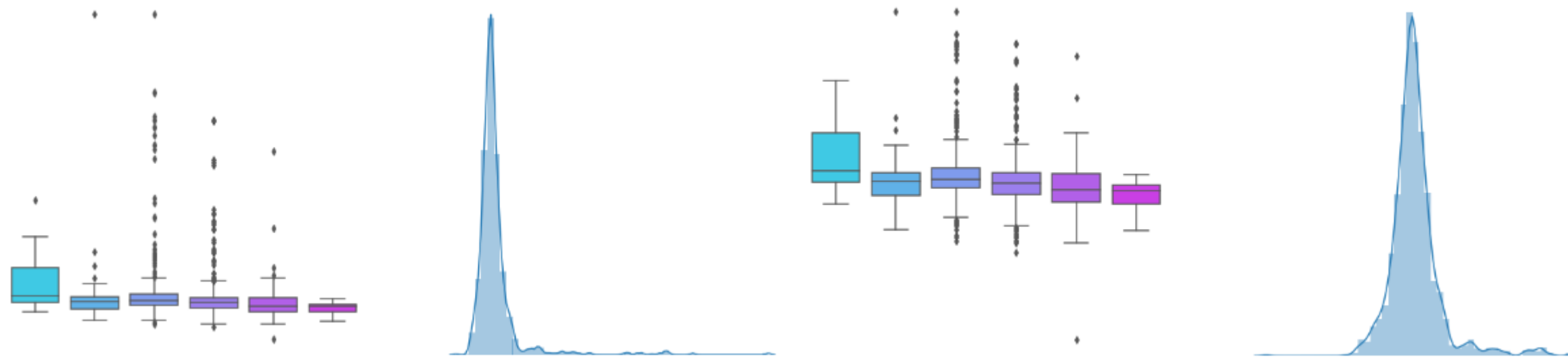
#3 Distribution of the Residual Sugar before & after treating outlier



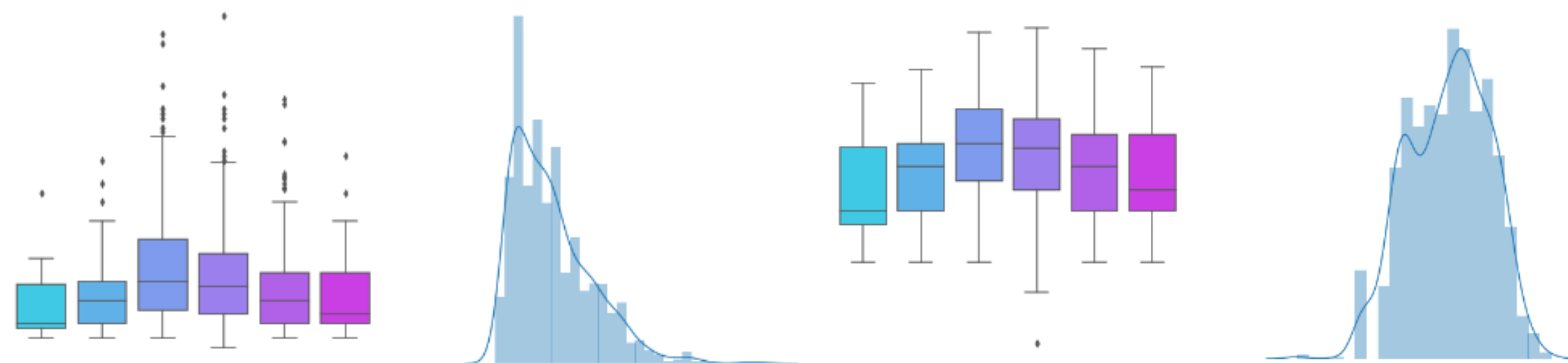


# #1.2 Feature Analysis & Finding Outliers

#4 Distribution of the Chlorides before & after treating outlier

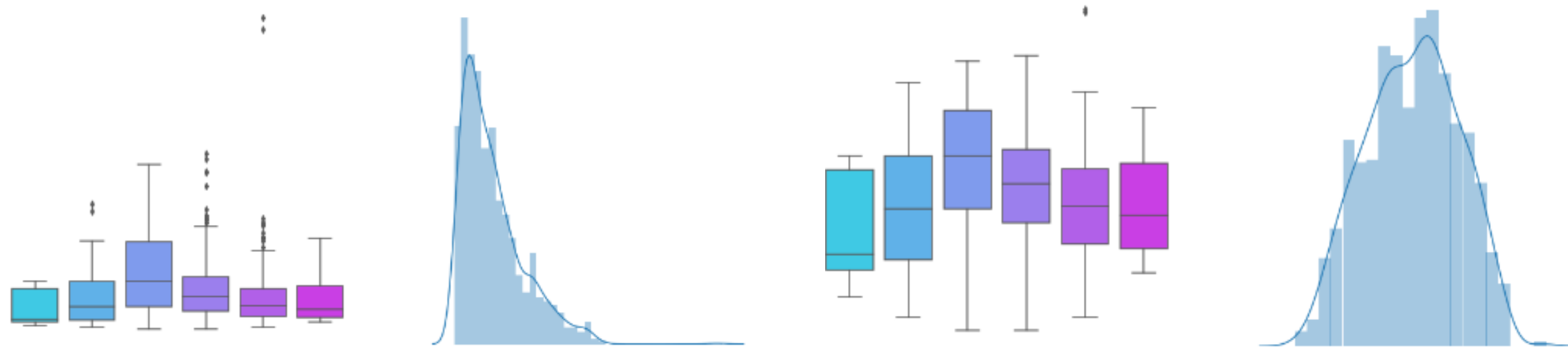


#5 Distribution of the free sulfur dioxide before & after treating outlier

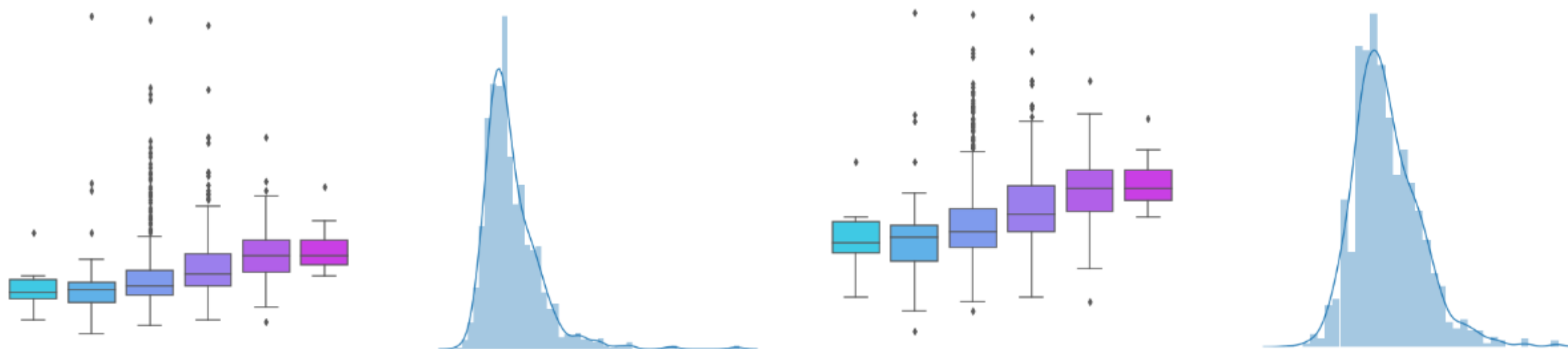


# #1.2 Feature Analysis & Finding Outliers

#6 Distribution of the total sulfur dioxide before & after treating outlier

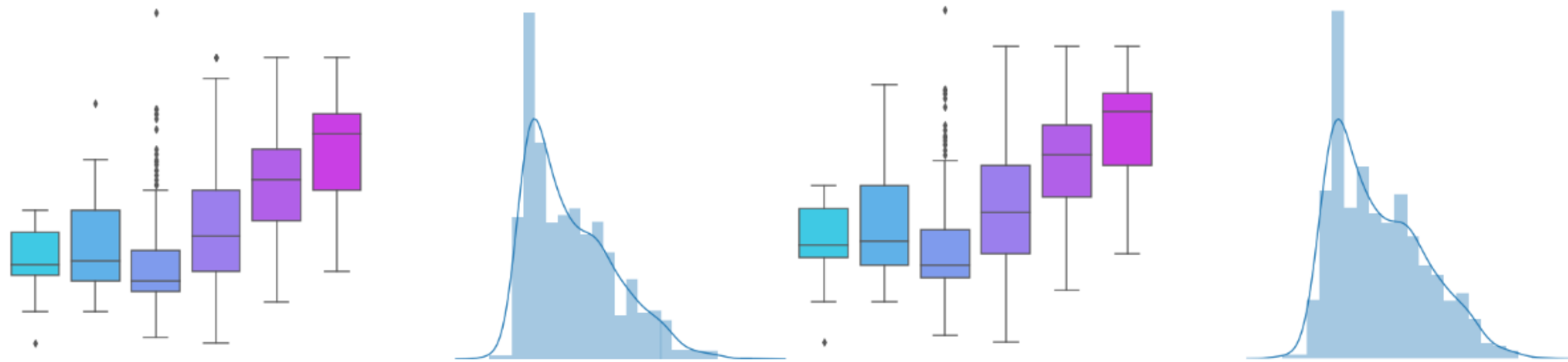


#7 Distribution of the Sulphates before & after treating outlier

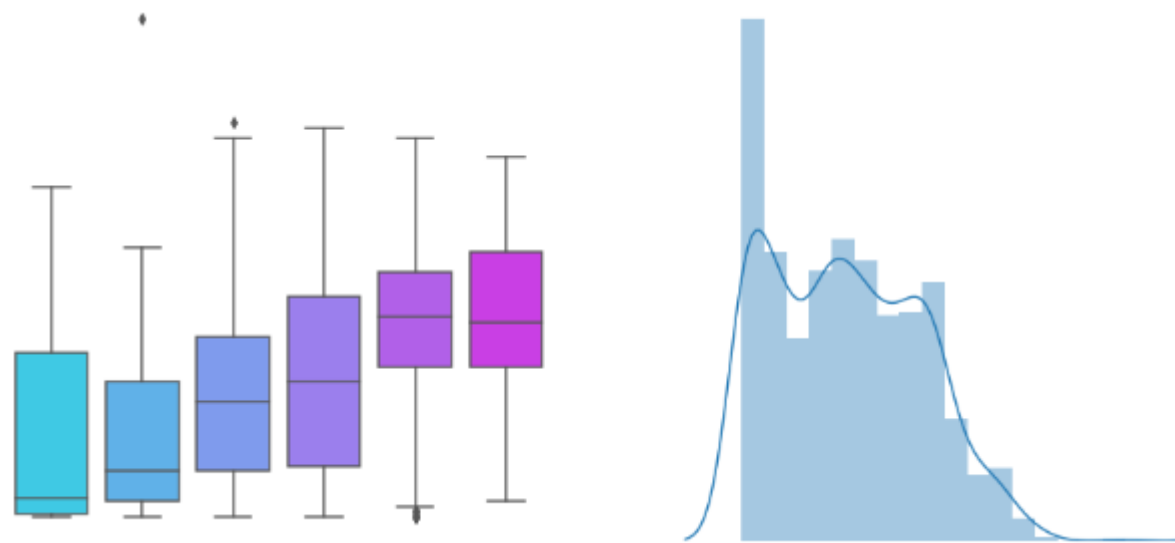


# #1.2 Feature Analysis & Finding Outliers

#8 Distribution of the Alcohol before & after treating outlier

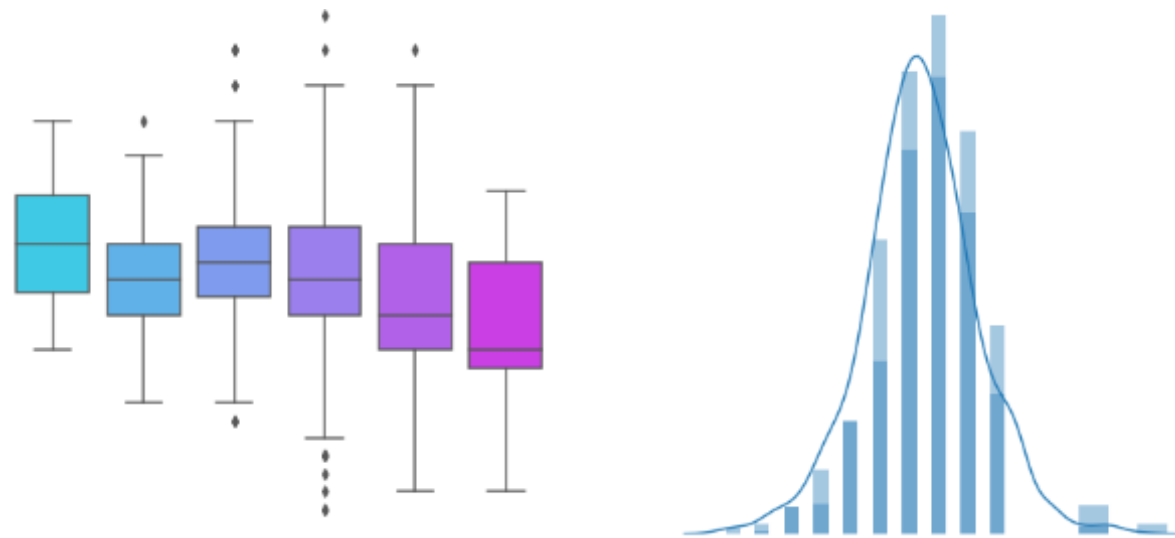


#9 Distribution of the citric acidity

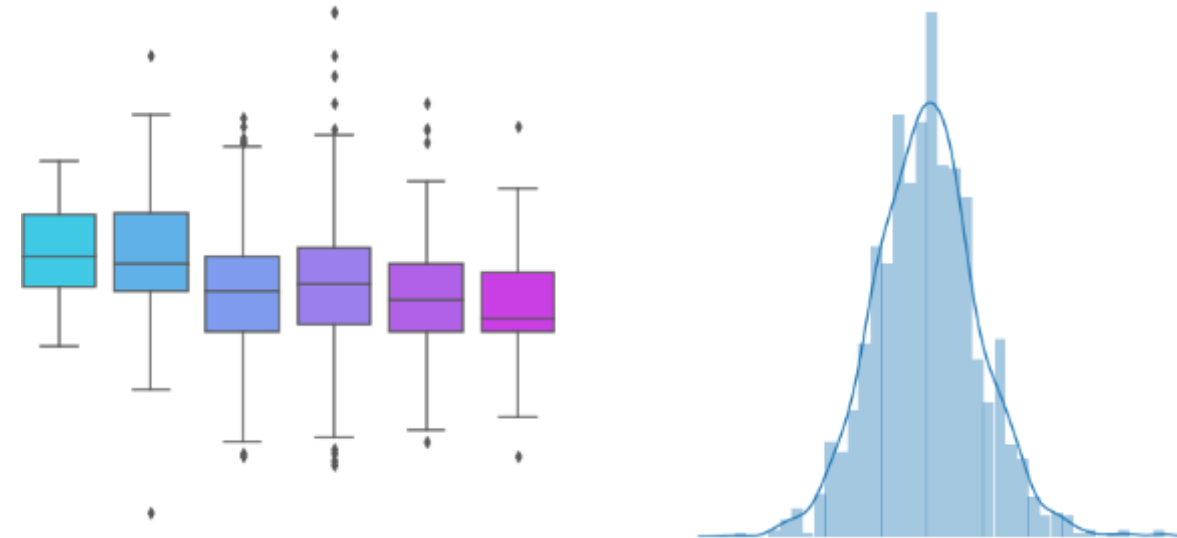


# #1.2 Feature Analysis & Finding Outliers

#10 Distribution of the Density



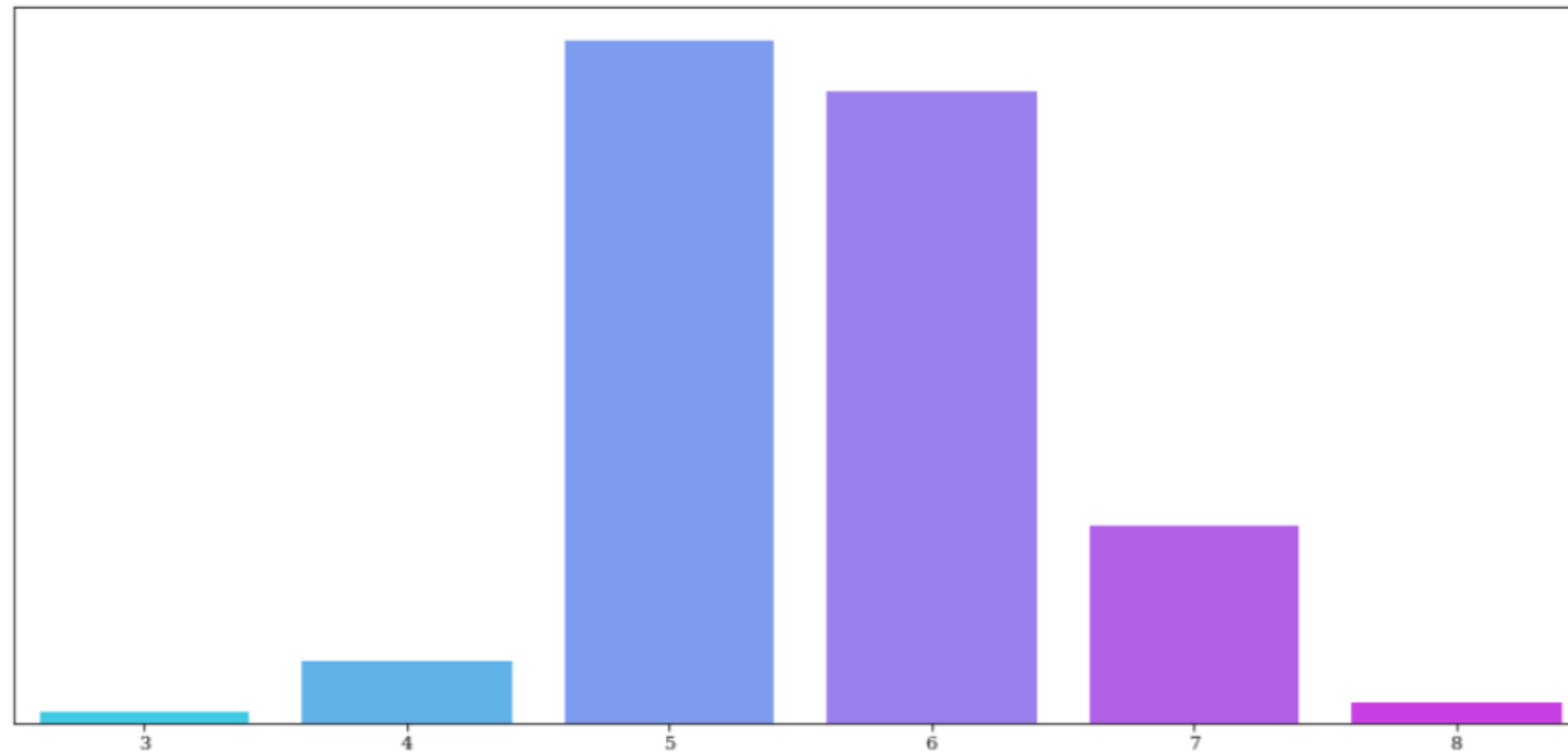
#11 Distribution of the pH



# #1.3 Feature Engineering & Transformation

# 타겟 변수 Quality의 분포도

**Before-Distribution Of The Quality**

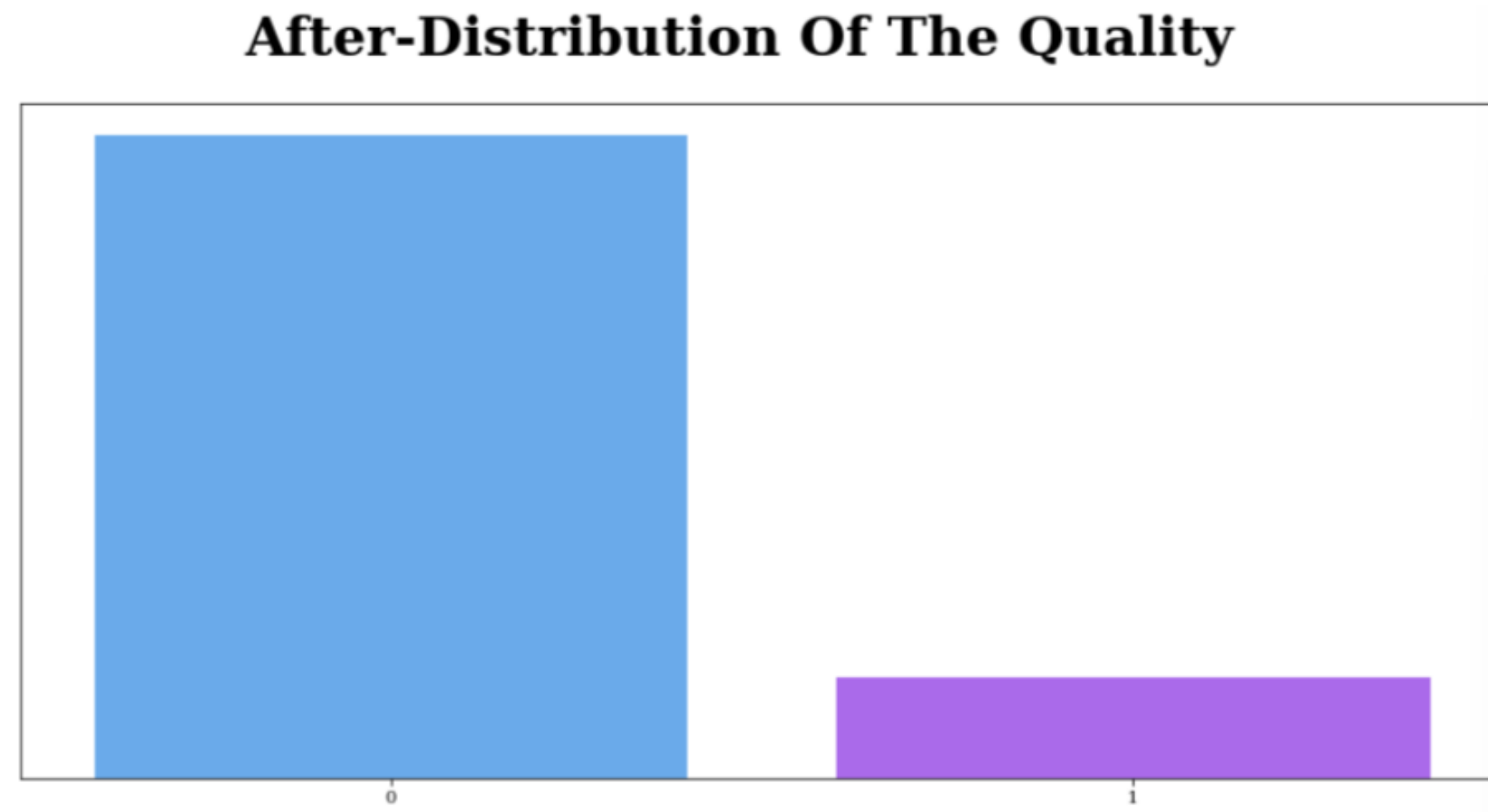


5, 6에 Quality 값이 편중되어 있는 것을 확인할 수 있음

→ target dataset을 binary classification으로 변환!

# #1.3 Feature Engineering & Transformation

# 타겟 변수 Quality의 분포도



```
#Feature Engineering...
bins = (2, 6.5, 8)
group_names = ['bad', 'good']
df['quality'] = pd.cut(df['quality'], bins = bins, labels = group_names)

#Feature Transformation...
df['quality'].replace({'bad':0, 'good':1}, inplace=True)
```

bad(0): Quality가 2~6.5 값일 때  
good(1): Quality가 6.5~8 값일 때

# #1.4 Dimensionality Reduction Using PCA

## # PCA

기존 데이터를 거의 유지하면서 large 변수 세트를 smaller 변수 세트로 변환시켜 large data sets의 차원을 줄이는 기법

How it works?

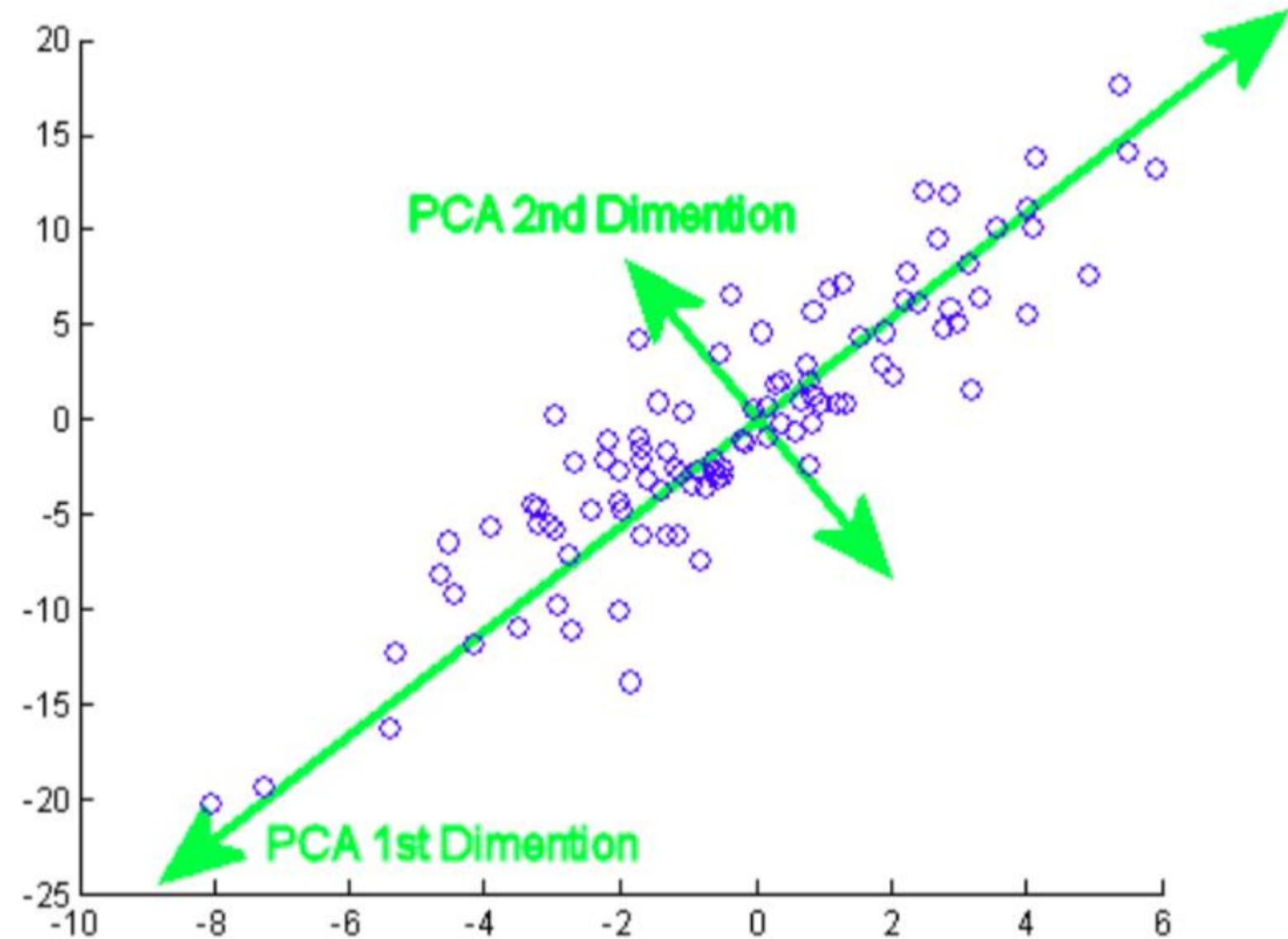
Step 1: Standardization

Step 2: Covariance Matrix computation

Step 3: Compute the eigenvectors and eigenvalues of the covariance matrix to identify the principal components

Step 4: Feature vector

Step 5: Recast the data along the principal components axes



# #1.5 PCA Implementation

# PCA 후 explained variance 확인 (n\_component=None)

```
#Feature Variables
x = df.drop('quality',axis=1)
#Target Variable
Y = df['quality']

x_train,x_test,Y_train,Y_test = train_test_split(x,Y,test_size = 0.25,random_state=44)

# Applying PCA FOR DIMENSIONALITY REDUCTION.....
pca = PCA(n_components = None)
x_train = pca.fit_transform(x_train)
x_test = pca.transform(x_test)
explained_variance = pca.explained_variance_ratio_

print("Sorted List returned :")
print(sorted(explained_variance,reverse = True))
```

Sorted List returned :

```
[0.5856416532823628, 0.10864226287909255, 0.09851545429655259, 0.07647690474821395, 0.0604299870497445, 0.0288552593279704
85, 0.025009548971263548, 0.007609445922171323, 0.004804967010158156, 0.004014204251241388, 3.1226122865626945e-07]
```



# #1.5 PCA Implementation

# 5개의 component를 가진 PCA 변환

```
# Applying PCA
from sklearn.decomposition import PCA
pca = PCA(n_components = 5)
x_train = pca.fit_transform(x_train)
x_test = pca.transform(x_test)
explained_variance = pca.explained_variance_ratio_
print(explained_variance)
```

```
[0.58564165 0.10864226 0.09851545 0.0764769 0.06042999]
```

# #1.6 Model Creation

```
kfold = StratifiedKFold(n_splits=8,shuffle=True, random_state=42)
```

```
rs = 15  
clrs = []
```

```
clrs.append(AdaBoostClassifier(random_state=rs))  
clrs.append(GradientBoostingClassifier(random_state=rs))  
clrs.append(RandomForestClassifier(random_state=rs))  
clrs.append(DecisionTreeClassifier(random_state = rs))
```

```
cv_results = []  
for clr in clrs :  
    cv_results.append(cross_val_score(clr, x_train, Y_train , scoring = 'accuracy', cv = kfold, n_jobs=-1))  
cv_means = []  
cv_std = []  
for cv_result in cv_results:  
    cv_means.append(cv_result.mean())  
    cv_std.append(cv_result.std())
```

```
cv_df = pd.DataFrame({"CrossVal_Score_Means":cv_means, "CrossValerrors": cv_std, "Algo":["RandomForestClassifier", "AdaBoostClassifier", "Gradient Boosting", 'DecisionTreeClassifier']})  
g = sns.barplot("CrossVal_Score_Means", "Algo", data = cv_df, orient = "h", **{'xerr':cv_std}, palette='cool', edgecolor="black", linewidth=3)  
g.set_xlabel("Mean Accuracy", fontsize = 18)  
g = g.set_title("Cross validation scores", fontsize = 24)  
plt.figure(figsize = (15,7))  
print(cv_df)
```

#Cross validation on different set of algorithm!!!

AdaBoost, GradientBoosting, RandomForest,  
DecisionTree를 이용

# #1.6 Model Creation

# Ada boosting is the winner!

	CrossVal_Score_Means	CrossValerrors	Algo
0	0.848846	0.020138	RandomForestClassifier
1	0.854820	0.015563	AdaBoostClassifier
2	0.862602	0.014184	Gradient Boosting
3	0.803685	0.011787	DecisionTreeClassifier

→ cross\_val\_score가 높고 cross\_val\_error가 낮은 AdaBoost가 성능이 가장 좋아 보임

# Ada boosting의 confusion matrix

```
from sklearn.tree import DecisionTreeClassifier
dtc = DecisionTreeClassifier()
dtc.fit(x_train, Y_train)
Y_pred = dtc.predict(x_test)
confusion_matrix(Y_test, Y_pred)
```

```
array([[254,  39],
       [ 31,  16]])
```

# #1.7 Hyper Parameter Tuning

## #1 DecisionTreeClassifier의 Hyper Parameter Tuning

```
from sklearn.model_selection import GridSearchCV
grid_params = {
    'criterion' : ['gini', 'entropy'],
    'max_depth' : [3, 5, 7, 10],
    'min_samples_split' : range(2, 10, 1),
    'min_samples_leaf' : range(2, 10, 1)
}

grid_search = GridSearchCV(dtc, grid_params, cv = 5, n_jobs = -1, verbose = 1)
grid_search.fit(x_train, Y_train)

dtc = grid_search.best_estimator_
Y_pred = dtc.predict(x_test)
print(accuracy_score(Y_test, Y_pred))
```

0.8647058823529412

## #2 AdaBoost의 Hyper Parameter Tuning

```
ada = AdaBoostClassifier(base_estimator = dtc)

parameters = {
    'n_estimators' : [50, 70, 90, 120, 180, 200],
    'learning_rate' : [0.001, 0.01, 0.1, 1, 10],
    'algorithm' : ['SAMME', 'SAMME.R']
}

grid_search = GridSearchCV(ada, parameters, n_jobs = -1, cv = 10, verbose = 1)
grid_search.fit(x_train, Y_train)

print(grid_search.best_params_)
print(grid_search.best_score_)
```

```
{'algorithm': 'SAMME', 'learning_rate': 0.1, 'n_estimators': 90}
0.8684985279685966
```

# #1.8 Final Model

```
ada = AdaBoostClassifier(base_estimator = dtc, algorithm= "SAMME", learning
_rate= 0.1, n_estimators= 90)
ada.fit(x_train, Y_train)

print(confusion_matrix(Y_test, Y_pred))

print(classification_report(Y_test, Y_pred))
```

```
[[291  2]
 [ 44  3]]
```

	precision	recall	f1-score	support
0	0.87	0.99	0.93	293
1	0.60	0.06	0.12	47
micro avg	0.86	0.86	0.86	340
macro avg	0.73	0.53	0.52	340
weighted avg	0.83	0.86	0.81	340

## 2. 이미지 데이터 차원 축소(MNIST)





# # 2-0. 지난 세션 복습

## • 차원 축소

- 매우 많은 특징들로 구성된 다차원 데이터 세트의 차원을 감소시킴으로써 새로운 차원의 데이터 세트를 생성하는 것
- 일반적으로 차원이 증가함에 따라 데이터 점들 사이의 거리는 기하급수적으로 멀어지고 **희박한(sparse)** 구조를 가짐
  - 수백 개 이상의 기능으로 구성된 데이터 세트의 경우 예측 신뢰도는 상대적으로 적은 차원에 대해 훈련된 모델보다 낮음
- 또한 특징이 많으면 개별 feature 간의 상관관계가 높을 가능성이 커짐
  - 선형 회귀와 같은 선형 모형에서는 입력 변수 간의 상관관계가 높을 때 **다중 공선성** 문제로 인해 모형의 예측 성능이 저하됨

## • 차원의 저주

- 주로 수치해석, 샘플링, 조합론, 머신러닝, 데이터 마이닝 및 데이터베이스와 같은 영역에서 발생
- 이러한 문제의 공통 주제는 차원이 증가할수록 점들 사이의 공간이 너무 빠르게 증가하여 사용 가능한 데이터가 희소해진다는(sparse) 것
  - 예측 성능 저하로 이어질 수 있음
- 신뢰할 수 있는 결과를 얻기 위해 필요한 데이터의 양은 종종 차원에 따라 기하급수적으로 증가
- 또한 데이터를 구성하고 검색하는 것은 종종 객체가 유사한 속성을 가진 그룹을 형성하는 영역을 탐지하는 데 의존
- 그러나 고차원 데이터에서는 모든 객체가 희박하고(sparse) 여러 면에서 서로 다른 것으로 나타나기 때문에 일반적인 데이터 구성 전략이 효율적이지 못함

# # 2-1. 대회 소개

- 다차원 feature의 차원 축소를 통해 feature의 개수를 줄임으로써 데이터를 보다 직관적으로 해석할 수 있는 **여러 차원 축소 방법들을 요약한 노트북**
- 활용 데이터셋: MNIST 데이터
- 차원 축소를 통해 이 데이터 세트를 살펴볼 때 데이터 간 규칙성이 있는지, 각 차원 축소 방법에 따라 label이 어떻게 clustering되는지에 중점을 둘 예정

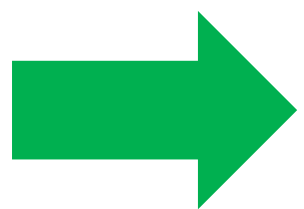
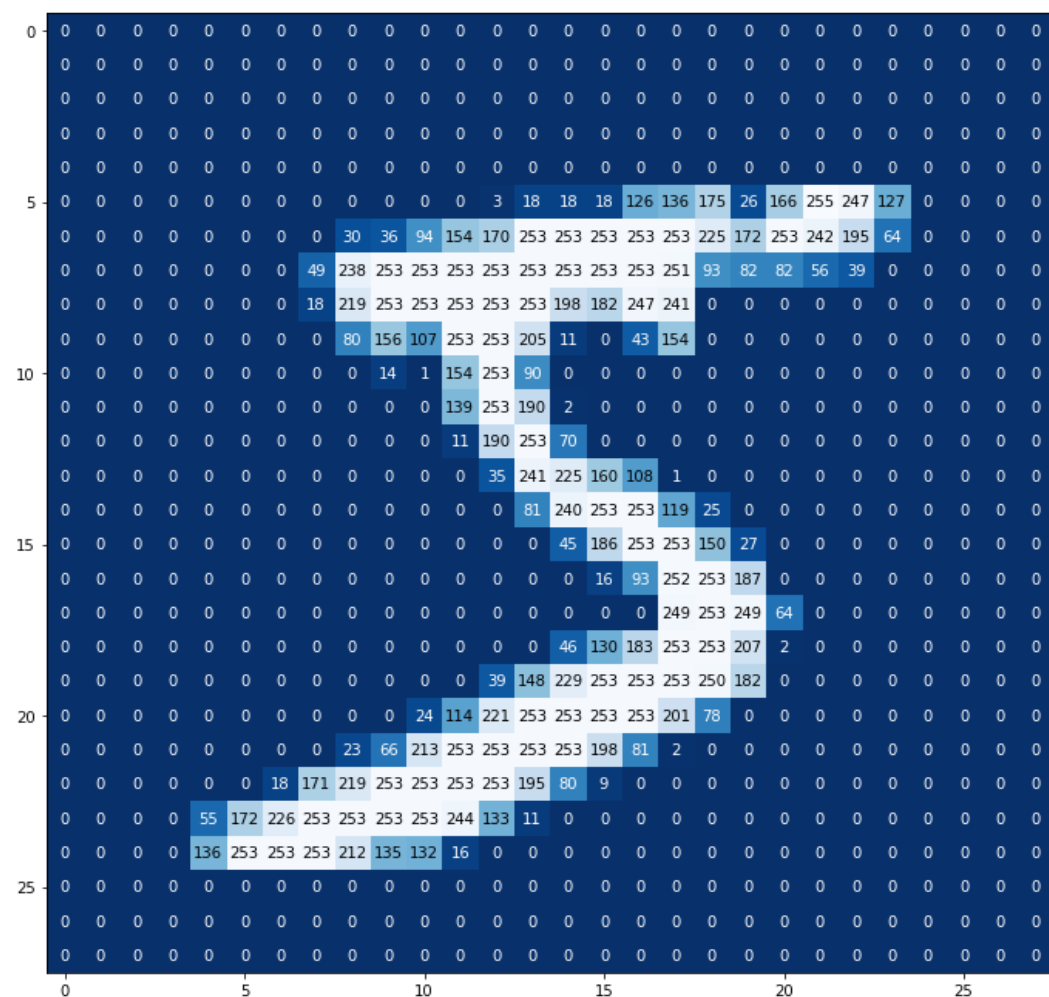




# # 2-2. Data Description

## 🔑 MNIST 데이터 셋?

- 손으로 쓴 0 ~ 9까지의 숫자 이미지로 구성된 데이터 셋
- 28 \* 28 사진 파일 -> 784개의 pixel로 구성됨
- 이미지를 구성하는 각각의 pixel은 0~ 255 사이의 값을 가짐 => 0에 가까울수록 **어두운 색**, 255에 가까울수록 **밝은 색**
- 각 이미지의 784개의 pixel값은 평탄화 과정을 거쳐 784개의 열을 가진 1차원 numpy 배열로 변환



	0	1	2	3	4	5	6	7	8	9	...	774	775	776	777	778	779	780	781	782	783
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
4995	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4996	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4997	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4998	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4999	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

5000 rows × 784 columns

# # 2-2. Data Description

## 📌 라이브러리 & 데이터셋 준비하기

```
# library

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
# Dataset Download

from keras.datasets import mnist
(train_x, train_y), (test_x, test_y) = mnist.load_data()

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [=====] - 0s 0us/step
```

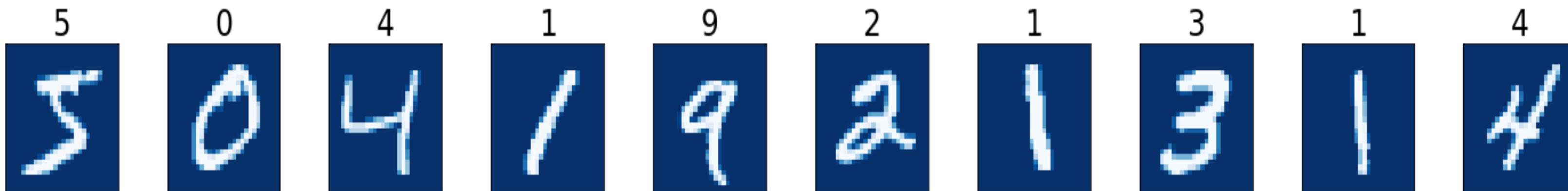
- train\_x: 손글씨 숫자 이미지
- train\_y: 이미지가 의미하는 숫자
- test\_x: 손글씨 숫자 이미지
- test\_y: 이미지가 의미하는 숫자

# # 2-2. Data Description

## 📌 훈련 데이터 셋 확인하기

# 시각화

```
fig = plt.figure(figsize = (25, 4))  
for idx in np.arange(10):  
    ax = fig.add_subplot(2, 10, idx + 1, xticks = [], yticks = [])  
    ax.imshow(train_x[idx], cmap = 'Blues_r')  
    ax.set_title(str(train_y[idx]), fontsize = 25)
```



# # 2-2. Data Description

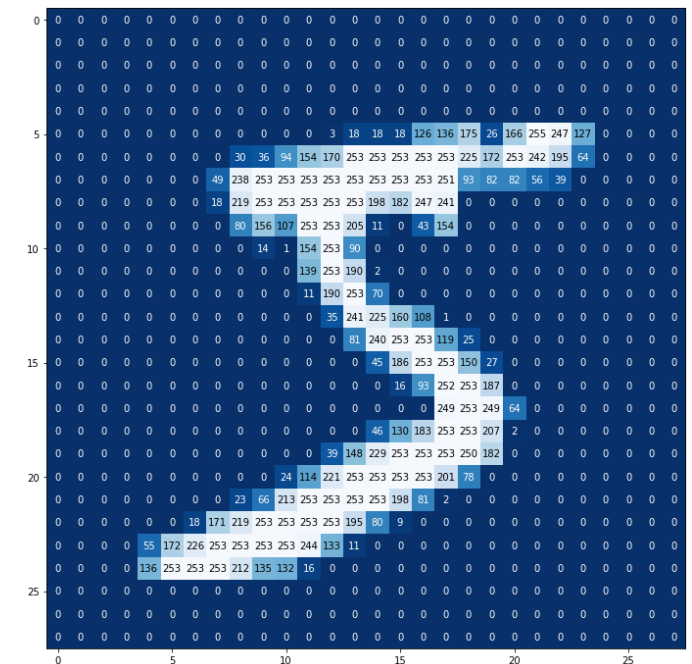
## 📌 훈련 데이터 셋 확인하기

# 0번 이미지를 임의로 선택하여 pixel값으로 숫자 시각화하기

```
img = train_x[0]

fig = plt.figure(figsize = (12,12))
ax = fig.add_subplot(111)
ax.imshow(img, cmap = 'Blues_r')
width, height = img.shape
thresh = img.max()/2.5 # 경계 설정 -> img.max()가 255이니 대략 102 이상

for x in range(width):
    for y in range(height):
        val = round(img[x][y],2) if img[x][y] != 0 else 0 # pixel값 얻어오기
        ax.annotate(str(val), xy = (y,x), # pixel값 표시
                    horizontalalignment = 'center', # 글자 가운데 정렬
                    verticalalignment = 'center', # 글자 가운데 정렬
                    color = 'white' if img[x][y]<thresh else 'black') # 해당 경계를 벗어나는 pixel값 -> 특이점으로 인식, 검은색으로 표시
```



# # 2-2. Data Description

## 📌 훈련 데이터 셋 확인하기

```
# 2차원 데이터로 reshape
train_x = train_x.reshape(train_x.shape[0], -1)
print(train_x.shape)

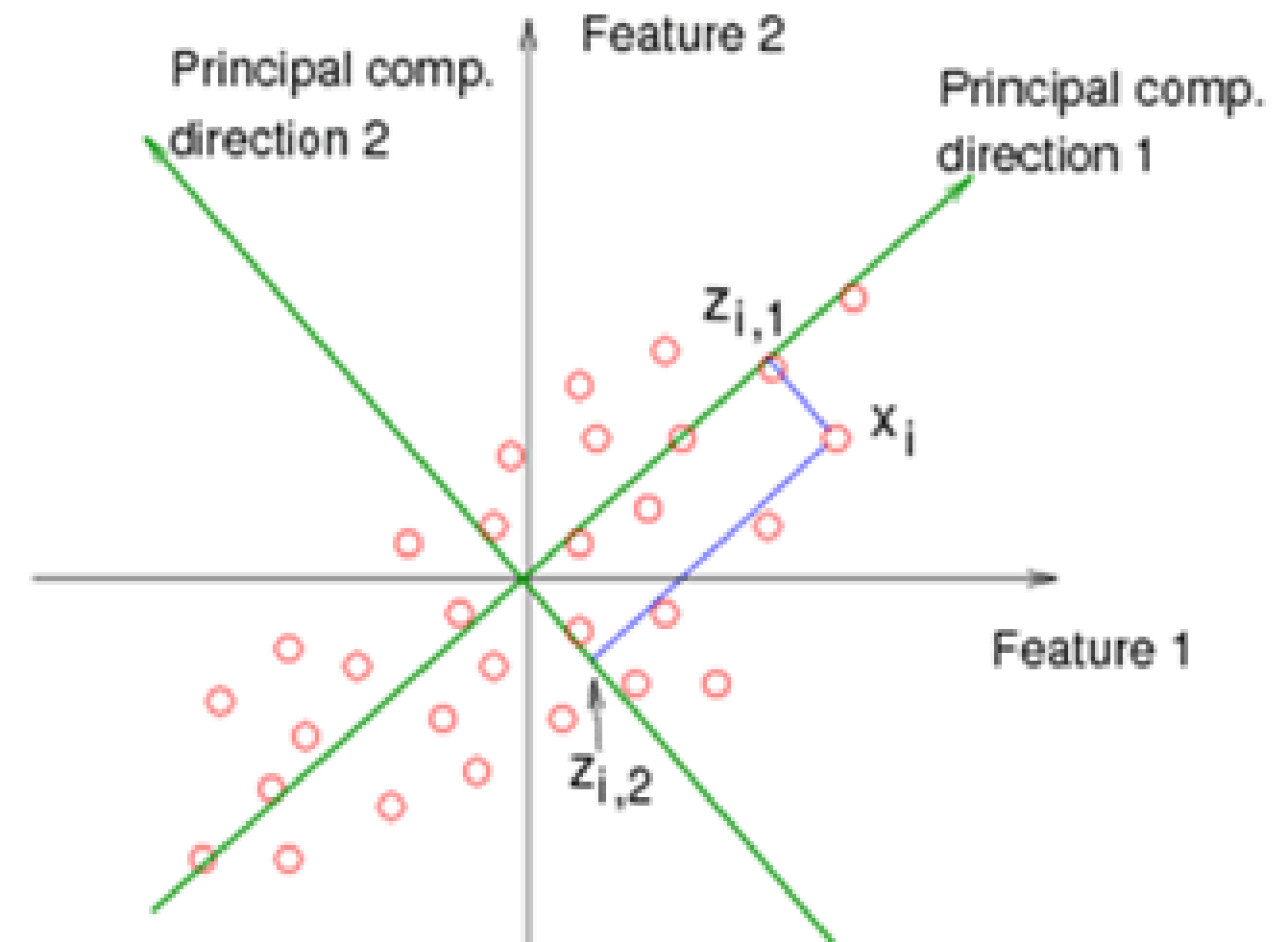
# 학습 데이터로 상위 1000개의 데이터만 선택하여 사용
sample_size = 5000
train_x = pd.DataFrame(train_x[:sample_size, :])
train_y = train_y[:sample_size]
```

```
(60000, 784)
```

# # 3. 여러 가지 차원 축소 기법 적용하기

## 1) PCA(Principal Component Analysis)

- 차원 축소의 가장 대표적인 방법으로, 다차원 데이터에서 분산이 최대한 보존되는 방향 (= 데이터의 분포를 가장 잘 표현하는 방향)으로 축을 계속해서 재설정하는 방법
    - $i$ 번째 축:  $i$ 번째 주성분(PC/ Principal Component)
    - $(i+1)$ 번째 주성분:  $i$ 번째 주성분에 수직이고 분산이 가장 큰 방향
  - 주성분 자체는 원본 차원과 동일하고, 주성분을 통해 변경된 데이터는 차원이 감소함
  - 일반적으로 주성분은 원본 특성의 개수만큼 찾을 수 있음
  - 변수 간의 의존성이 클수록 주성분은 원래 데이터를 나타낼 수 있음  
but 각 feature는 정규 분포를 따른다고 가정
- 왜곡된 분포를 갖는 변수를 PCA에 적용하는 것은 적절하지 않음



# # 3. 여러 가지 차원 축소 기법 적용하기

## 1) PCA(Principal Component Analysis)

```
### 라이브러리 import  
  
from sklearn.decomposition import PCA
```

```
### 차원 축소  
  
pca = PCA()  
x_pca = pca.fit_transform(train_x)
```

```
# 데이터 형태 확인  
  
x_pca.shape  
  
(5000, 784)
```

# # 3. 여러 가지 차원 축소 기법 적용하기

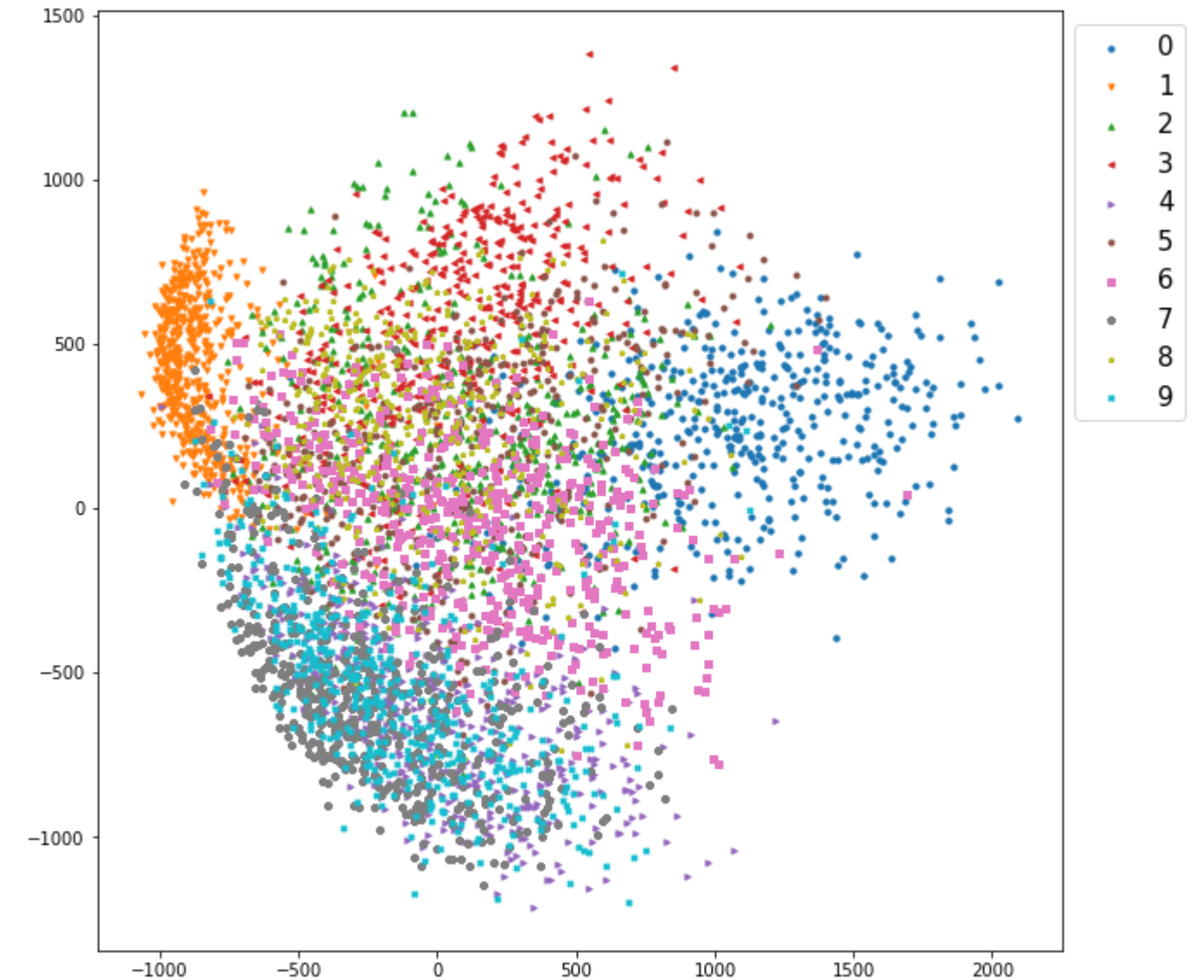
## 1) PCA(Principal Component Analysis)

```
### 시각화

markers = ['o', 'v', '^', '<', '>', '8', 's', 'P', '*', 'X']

# 각 데이터가 속하는 class를 plotting

plt.figure(figsize = (10,10))
for i,marker in enumerate(markers):
    mask = train_y == i
    plt.scatter(x_pca[mask, 0], x_pca[mask, 1], label = i, s = 10, alpha = 1,marker = marker)
plt.legend(bbox_to_anchor = (1.00, 1), loc = 'upper left',fontSize=15)
```



- 숫자 0,1,7은 어느 정도 다른 숫자와 잘 구분되지만(클러스터의 경계가 뚜렷하지만) 다른 숫자들은 거의 구분되고 있지 않음을 확인할 수 있다. (클러스터의 경계가 뚜렷하지 않다.)



# # 3. 여러 가지 차원 축소 기법 적용하기

## 2) Truncated SVD

### 📌 SVD

- 특이값 분해 기법 => 임의의  $m \times n$  차원의 행렬  $A$ 에 대하여 다음과 같이 행렬을 분해할 수 있다는 행렬 분해(decomposition) 방법 중 하나

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

- $A$ : 임의의  $m \times n$  행렬(rectangular matrix)
- $U$ :  $m \times m$  직교행렬(orthogonal matrix)
- $\Sigma$ :  $m \times n$  대각행렬(diagonal matrix)
  - 행렬의 대각에 위치한 값만 0이 아니고 나머지 위치의 값은 모두 0
  - 대각 원소 값이 바로 행렬  $A$ 의 특이값(Singular Value)
- $V$ :  $n \times n$  직교행렬(orthogonal matrix)

- 특이 벡터(Singular Vector): 행렬  $U$ 와  $V$ 에 속한 벡터  
→ 모든 특이 벡터는 서로 직교하는 성질을 가짐
- 정방행렬 뿐만 아니라 행과 열의 크기가 다른 행렬에도 적용 가능

$$A_{m \times n} = U_{m \times m} \times \begin{matrix} \text{[diagonal matrix with } m \text{ rows]} \end{matrix} \Sigma_{m \times n} \times V_{n \times n}^T$$

$(m < n)$

$$A_{m \times n} = U_{m \times m} \times \begin{matrix} \text{[diagonal matrix with } n \text{ columns]} \end{matrix} \Sigma_{m \times n} \times V_{n \times n}^T$$

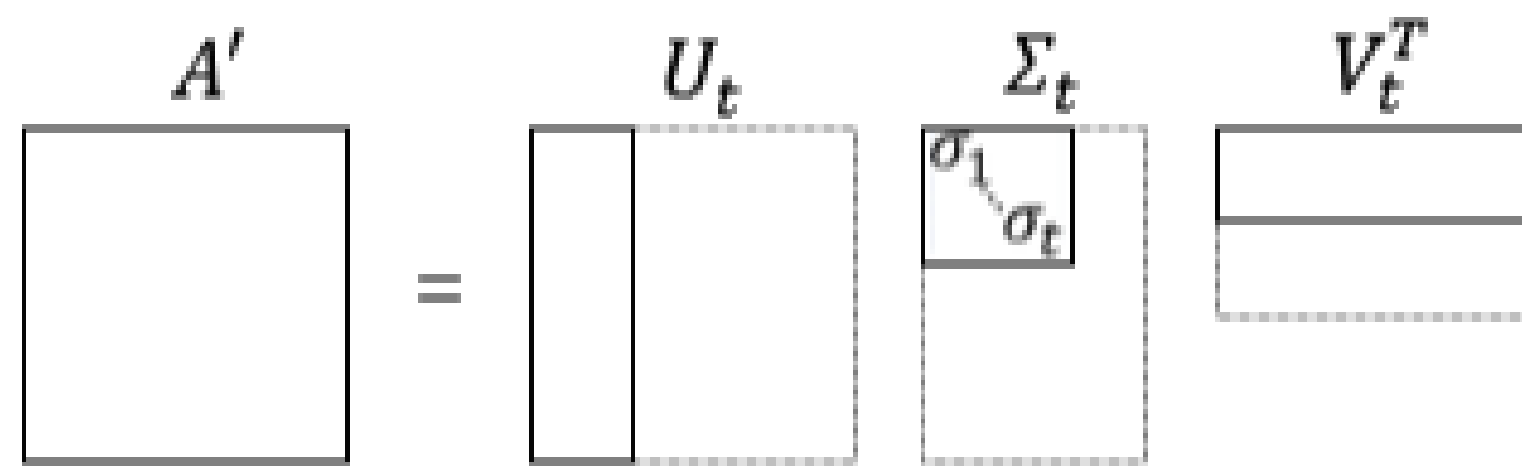
$(m > n)$

# # 3. 여러 가지 차원 축소 기법 적용하기

## 2) Truncated SVD

- $\Sigma$  행렬에서 대각(diagonal) 원소의 윗부분, 즉 특이값(singular value)의 상위 몇 개 만을 추출하여 분해하는 방법
- $\Sigma$  행렬의 비대각 부분과 대각 원소 중 특이값이 0인 부분을 모두 제거하고, 제거된  $\Sigma$  행렬에 대응되는  $U$  행렬과  $V$  행렬의 원소도 함께 제거해 차원을 줄인 형태로 분해
- 이러한 분해로 인해  $A=U\Sigma V^T$ 는 더 작은 차원으로 인위적으로 분해됨
  - 원소의 손실로 인해 원본 행렬을 정확하게 복원할 수 없음
  - 그러나, 데이터 정보가 압축되고 분해되었음에도 원래의 행렬을 상당한 정도로 근사하는 것은 가능

**Truncated SVD**

$$A' = U_t \Sigma_t V_t^T$$


# # 3. 여러 가지 차원 축소 기법 적용하기

## 2) Truncated SVD

```
### 라이브러리 import  
  
from sklearn.decomposition import TruncatedSVD
```

```
### 차원 축소  
  
tsvd = TruncatedSVD()  
x_tsvd = tsvd.fit_transform(train_x)
```

```
### 데이터 형태 확인  
  
x_tsvd.shape  
  
(5000, 2)
```

# # 3. 여러 가지 차원 축소 기법 적용하기

## 2) Truncated SVD

```
### 시각화

markers=['o','v','^','<','>','8','s','P','*','X']

# 각 데이터가 속하는 class를 plotting
plt.figure(figsize = (10,10))
for i,marker in enumerate(markers):
    mask = train_y == i
    plt.scatter(x_tsvd[mask, 0], x_tsvd[mask, 1], label = i, s = 10, alpha = 1,marker = marker)
plt.legend(bbox_to_anchor = (1.00, 1), loc = 'upper left',fontsize = 15)
```



- 숫자 0,1은 어느 정도 다른 숫자와 잘 구분되지만(클러스터의 경계가 뚜렷하지만) 다른 숫자들은 거의 구분되고 있지 않음을 확인할 수 있다.  
(클러스터의 경계가 뚜렷하지 않다.)
- PCA와 유사하게 변환 후 숫자별로 어느 정도 클러스터링이 가능함을 확인할 수 있음

# # 3. 여러 가지 차원 축소 기법 적용하기

Reference: <https://bcho.tistory.com/1216>

## 3) NMF(Non-Negative Matrix Factorization)

- 음수 미포함 행렬 분해 기법 → 원본 행렬(V)에 있는 모든 요소의 값이 양수라는 것이 보장되어야 함
- SVD와 같은 low-rank 근사법의 변형
- 음수를 포함하지 않는 행렬 V를 음수를 포함하지 않는 W 행렬과 H 행렬의 곱으로 분해
  - W 행렬(Weight matrix, 가중치 행렬)
    - > 주로 길고 가는 행렬
    - > 잠재 요소의 값이 원본 행렬에 얼마나 잘 대응되는지를 나타냄
  - H 행렬(Feature matrix, 특성 행렬)
    - > 주로 작고 넓은 행렬
    - > 잠재 요소가 원본 행렬의 특징들로 어떻게 구성되어 있는지를 나타냄
- 새로운 feature 데이터 셋이 어떻게 원본 데이터들과 관계를 가지는지 확인 가능  
→ PCA나 t-SNE와 달리 NMF에 의해 추출된 특징의 경우에는 해석이 가능하다는 장점이 있음

$$\begin{matrix} W \\ \left[ \begin{array}{|c|c|} \hline & \\ \hline & \\ \hline & \\ \hline & \\ \hline \end{array} \right] \end{matrix} \times \begin{matrix} H \\ \left[ \begin{array}{|c|c|c|c|c|c|} \hline & & & & & \\ \hline & & & & & \\ \hline & & & & & \\ \hline \end{array} \right] \end{matrix} \approx \begin{matrix} V \\ \left[ \begin{array}{|c|c|c|c|c|c|} \hline & & & & & \\ \hline & & & & & \\ \hline & & & & & \\ \hline & & & & & \\ \hline \end{array} \right] \end{matrix}$$

# # 3. 여러 가지 차원 축소 기법 적용하기

## 3) NMF(Non-Negative Matrix Factorization)

```
### 라이브러리 import  
  
from sklearn.decomposition import NMF
```

```
### 차원 축소  
  
nmf = NMF(n_components = 2, init = 'random', random_state = 0)  
x_nmf = nmf.fit_transform(train_x)
```

```
### 데이터 형태 확인  
  
x_nmf.shape  
  
(5000, 2)
```

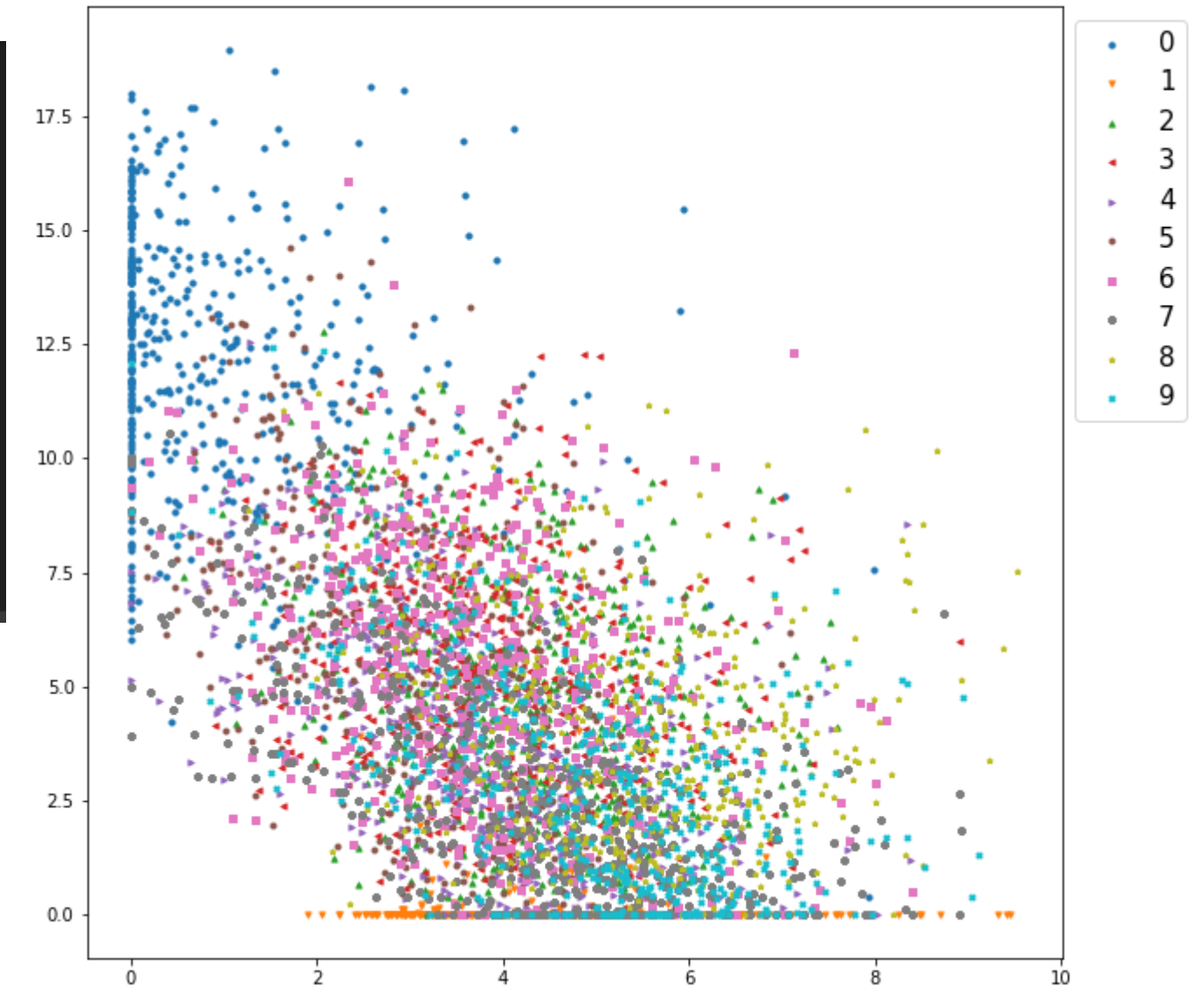
# # 3. 여러 가지 차원 축소 기법 적용하기

## 3) NMF(Non-Negative Matrix Factorization)

```
### 시각화

markers=['o','v','^','<','>','8','s','P','*','X']

# 각 데이터가 속하는 class를 plotting
plt.figure(figsize = (10,10))
for i,marker in enumerate(markers):
    mask = train_y == i
    plt.scatter(x_nmf[mask, 0], x_nmf[mask, 1], label = i, s = 10, alpha = 1,marker = marker)
plt.legend(bbox_to_anchor = (1.00, 1), loc = 'upper left',fontsize = 15)
```



- 숫자 0,1은 다른 숫자와 매우 잘 구분된다.(클러스터의 경계가 뚜렷하지만)
- 숫자 9도 다른 숫자들에 비해 꽤 잘 구분된다.
- 다른 숫자들은 거의 구분되고 있지 않음을 확인할 수 있다.  
(클러스터의 경계가 뚜렷하지 않다.)



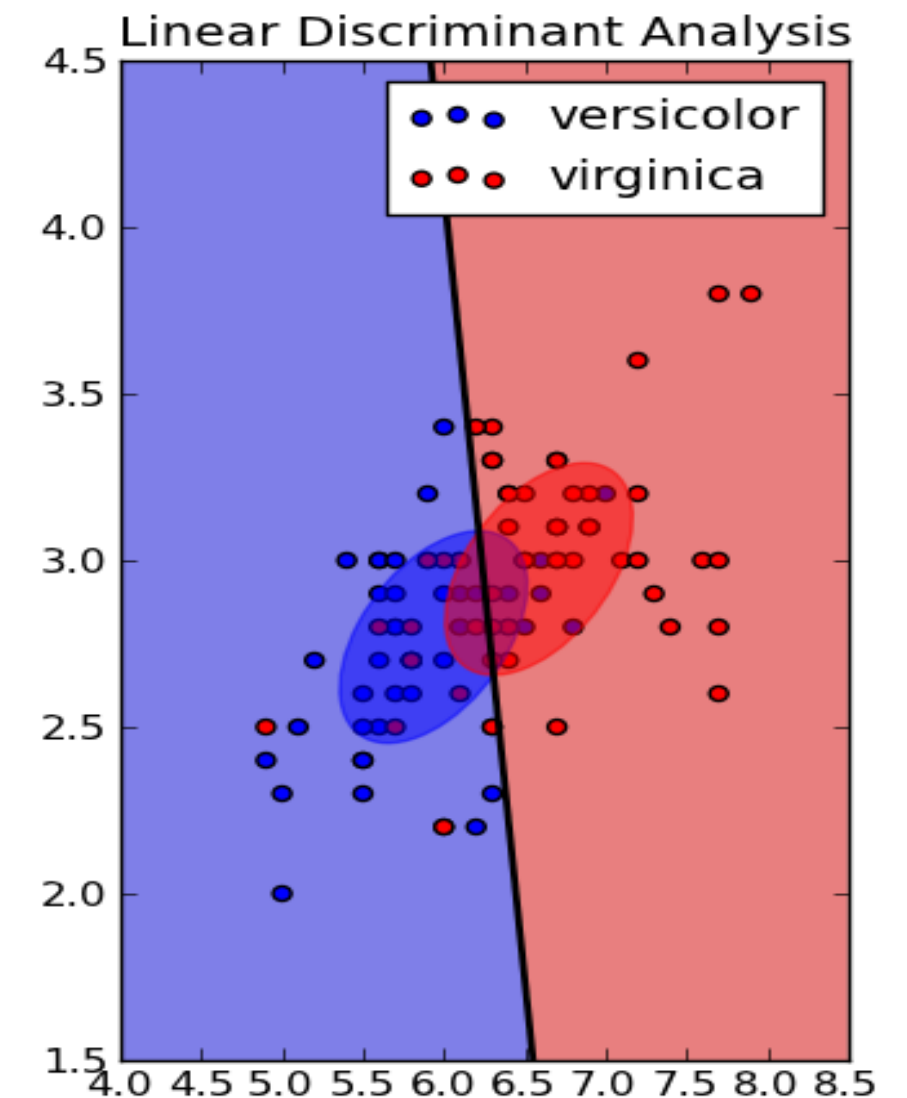
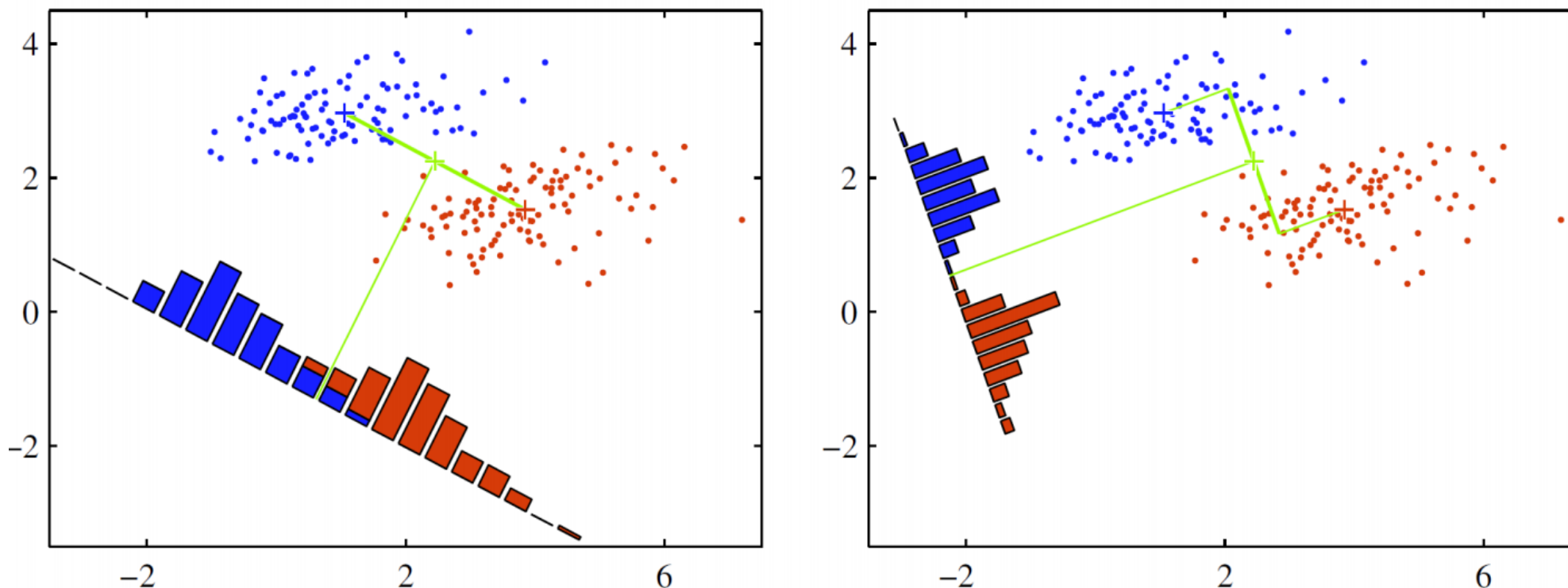
# # 3. 여러 가지 차원 축소 기법 적용하기

## 4) LDA(Linear Discriminant Analysis)

Reference:

<https://ratsgo.github.io/machine%20learning/2017/03/21/LDA/>

- 지도 학습의 분류 문제에서 차원을 줄이는 방법  
→ 개별 클래스를 분별할 수 있는 기준을 최대한 유지하며 차원을 축소
- 학습 데이터의 결정 값 클래스를 최대한으로 분리할 수 있는 저차원 feature 공간을 찾고, 해당 공간에 원래 feature를 투영해 차원 축소  
→ 이후 feature들을 잘 구분할 수 있는 직선을 찾는 걸 목표로 함
- 클래스 간 분산은 최대한 크게, 클래스 내부 분산은 최대한 작게 되도록 클래스를 분리함





# # 3. 여러 가지 차원 축소 기법 적용하기

## 4) LDA(Linear Discriminant Analysis)

```
### 라이브러리 import

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
```

```
### 차원 축소

lda = LDA(n_components=2)
x_lda = lda.fit_transform(train_x, train_y) # LDA는 지도학습 -> 차원 축소 시(transform 시) 클래스 결정값이 필요
```

```
### 데이터 형태 확인
```

```
x_lda.shape
```

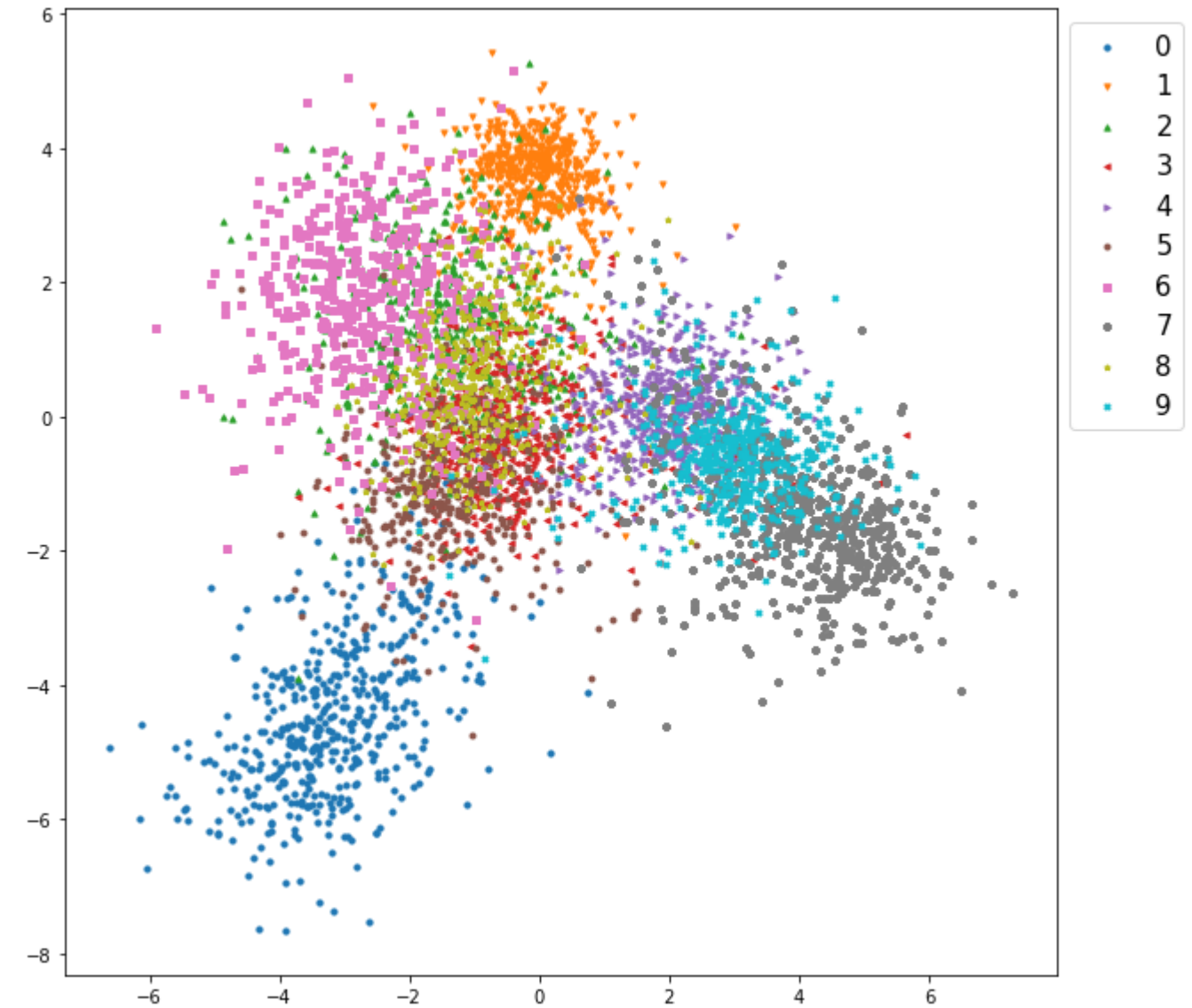
```
(5000, 2)
```

# # 3. 여러 가지 차원 축소 기법 적용하기

## 4) LDA(Linear Discriminant Analysis)

### 시각화

```
plt.figure(figsize = (10,10))
for i,marker in enumerate(markers):
    mask = train_y == i
    plt.scatter(x_lda[mask, 0], x_lda[mask, 1], label = i, s = 10, alpha = 1,marker = marker)
plt.legend(bbox_to_anchor = (1.00, 1), loc = 'upper left',fontSize = 15)
```



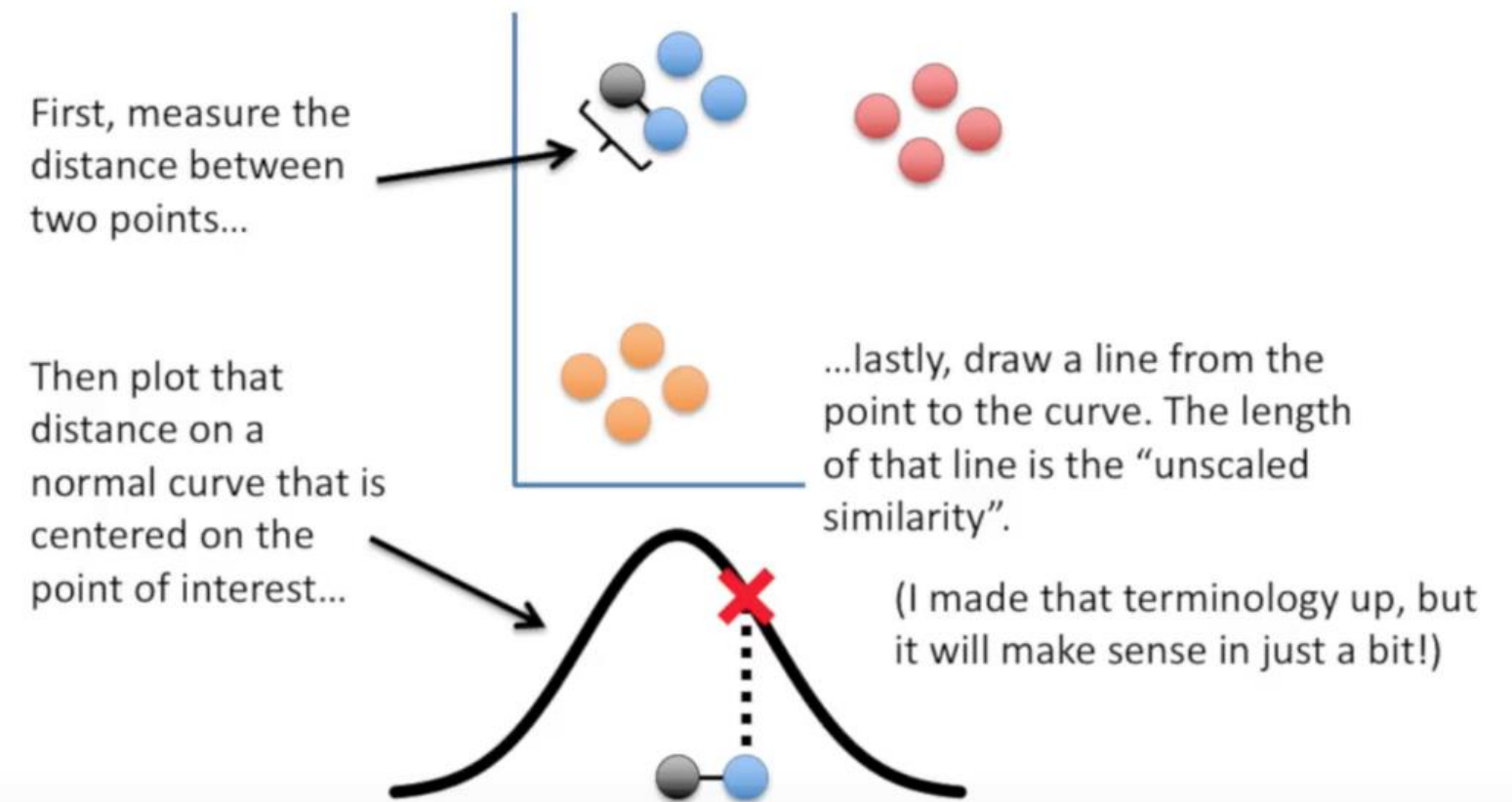
- 이전의 차원 축소 기법들에 비해 각 숫자들의 클러스터 경계가 뚜렷해짐
- 숫자 0,1,6,7,9는 어느 정도 다른 숫자와 잘 구분되지만(클러스터의 경계가 뚜렷하지만) 숫자 2,3,5,8의 경우 클러스터 경계가 거의 겹쳐 있음을 확인할 수 있음

# # 3. 여러 가지 차원 축소 기법 적용하기

Reference: <https://bcho.tistory.com/1210>

## 5) t-SNE(t-distributed Stochastic Neighbor Embedding)

- 고차원의 데이터를 압축하여 2차원(저차원) 평면에 시각화하기 위해 사용
- 높은 차원 공간에서 비슷한 데이터 구조는 낮은 차원 공간에서 가깝게 대응하며 비슷하지 않은 데이터 구조는 멀리 떨어져 대응
  - 거리 측정 시 t 분포 활용
  - 압축 후 원래의 feature 공간에 가까운 점도 2차원 평면으로 표현됨
- 비선형 관계를 식별할 수 있음
  - t-SNE에 의해 표현된 압축 결과를 원래 특징에 추가하여 모델 성능을 향상시킬 수 있음



# # 3. 여러 가지 차원 축소 기법 적용하기

## 5) t-SNE(t-distributed Stochastic Neighbor Embedding)

```
### 라이브러리 import  
  
from sklearn.manifold import TSNE
```

```
### 차원 축소  
  
tsne = TSNE(n_components=2)  
x_tsne = tsne.fit_transform(train_x)
```

```
### 데이터 형태 확인
```

```
x_tsne.shape
```

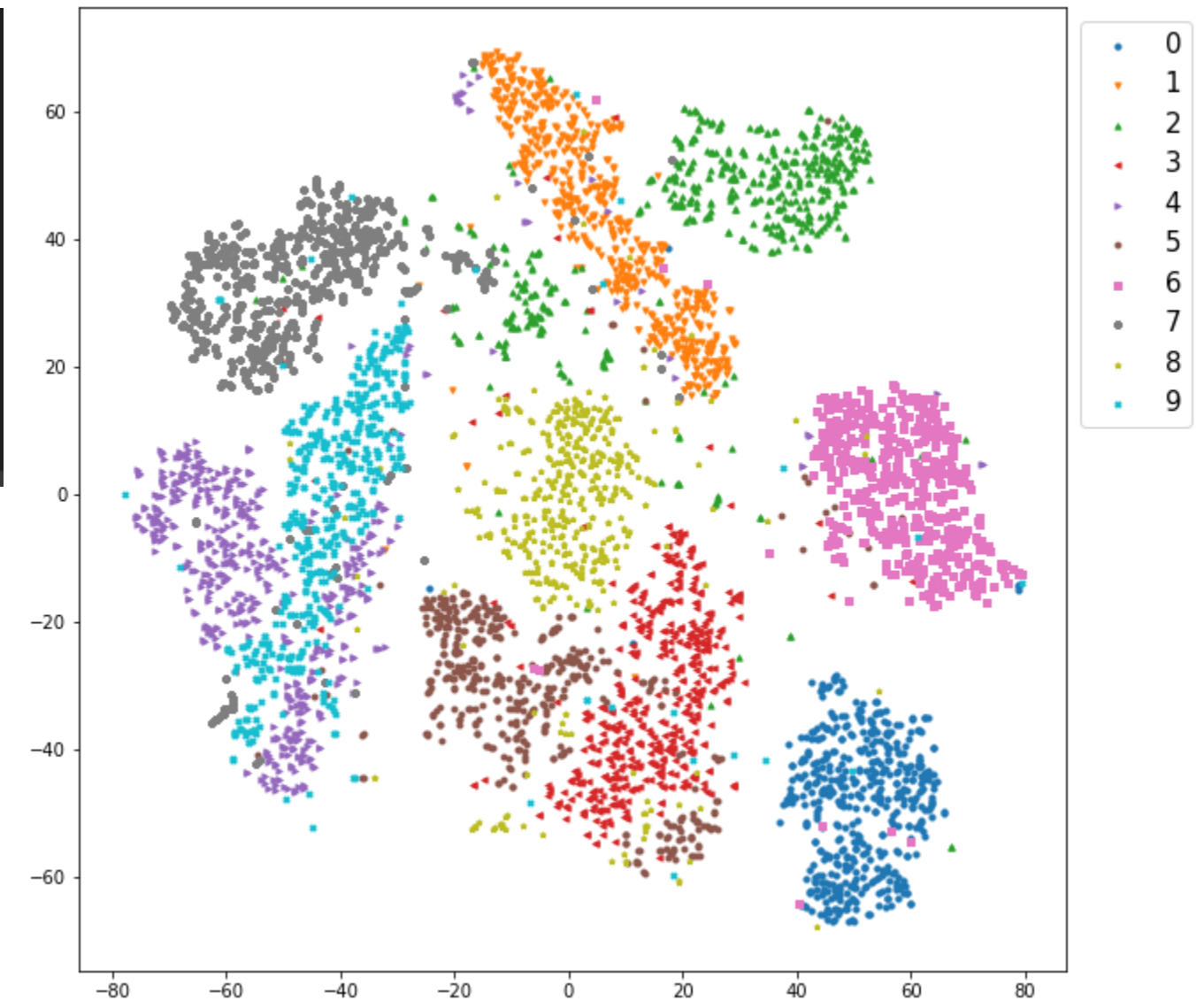
```
(5000, 2)
```

# # 3. 여러 가지 차원 축소 기법 적용하기

## 5) t-SNE(t-distributed Stochastic Neighbor Embedding)

### 시각화

```
plt.figure(figsize = (10,10))
for i,marker in enumerate(markers):
    mask = train_y == i
    plt.scatter(x_tsne[mask, 0], x_tsne[mask, 1], label = i, s = 10, alpha = 1,marker = marker)
plt.legend(bbox_to_anchor=(1.00, 1), loc = 'upper left',fontSize = 15)
```



- 각 숫자들의 클러스터링 경계가 대체로 뚜렷함을 확인할 수 있음
- 숫자들이 대체로 잘 구분되지만 숫자 3과 5, 숫자 4와 9의 경우 클러스터링 경계가 겹쳐 잘 구분되지 않음을 확인할 수 있음

# # 3. 여러 가지 차원 축소 기법 적용하기

## 6) UMAP(Uniform Manifold Approximation and Projection)

- 비선형 차원 축소를 위해 제안됨 → t-SNE보다 빠르고 데이터 공간을 잘 분리함
- 고차원에서의 데이터를 graph로 만들고, 저차원에서의 graph projection을 수행
- 단계

1) data point 에서 simplex 복합체로 만들어서 graph 구성

2) 각 node에서의 길이 k의 radius 를 그림(knn 등 이용, 정해진 nearest neighbor n개 만큼을 포함하게 되는 radius k를 그림)

이때 k가 작으면 local structure, 크면 global structure 를 가져올 수 있음

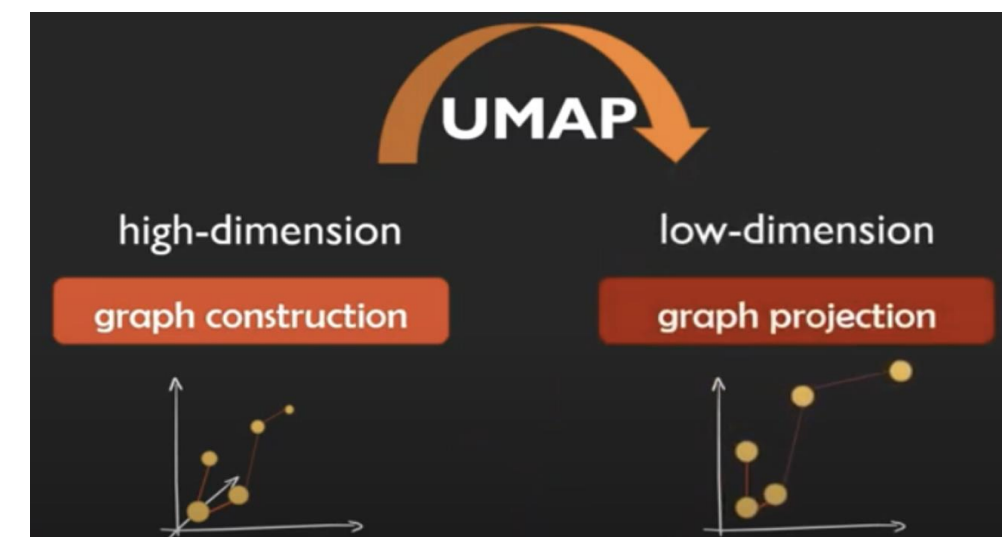
→ 겹치는 정도로 연결의 가중치(weight)를 결정

3) 이 strength를 그대로 저차원으로 이동 → UMAP 완성

- 매우 큰 데이터 세트를 빠르게 처리할 수 있으며, 희소 행렬 데이터 처리에 적합
- embedding 차원 크기에 대한 제한이 X + 다른 머신러닝 모델에서 새로운 데이터가 들어오면 즉시 embedding이 가능 → 일반적인 차원 축소 알고리즘으로 적용 가능
- global structure 를 더 잘 보존한다는 장점이 있음

Reference:

[https://velog.io/@stella\\_y/%EC%B0%A8%EC%9B%90-%EC%B6%95%EC%86%8C-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98%EC%9D%84-%EB%B9%84%EA%B5%90%ED%95%B4%EB%B3%B4%EC%9E%90-PCA-T-sne-UMAP](https://velog.io/@stella_y/%EC%B0%A8%EC%9B%90-%EC%B6%95%EC%86%8C-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98%EC%9D%84-%EB%B9%84%EA%B5%90%ED%95%B4%EB%B3%B4%EC%9E%90-PCA-T-sne-UMAP)



# # 3. 여러 가지 차원 축소 기법 적용하기

## 6) UMAP(Uniform Manifold Approximation and Projection)

```
# 라이브러리 설치
```

```
!pip install umap-learn
```

```
### 라이브러리 import
```

```
import umap.umap_ as umap
```

```
### 차원 축소
```

```
um = umap.UMAP()
```

```
x_umap = um.fit_transform(train_x)
```

```
### 데이터 형태 확인
```

```
x_umap.shape
```

```
(5000, 2)
```

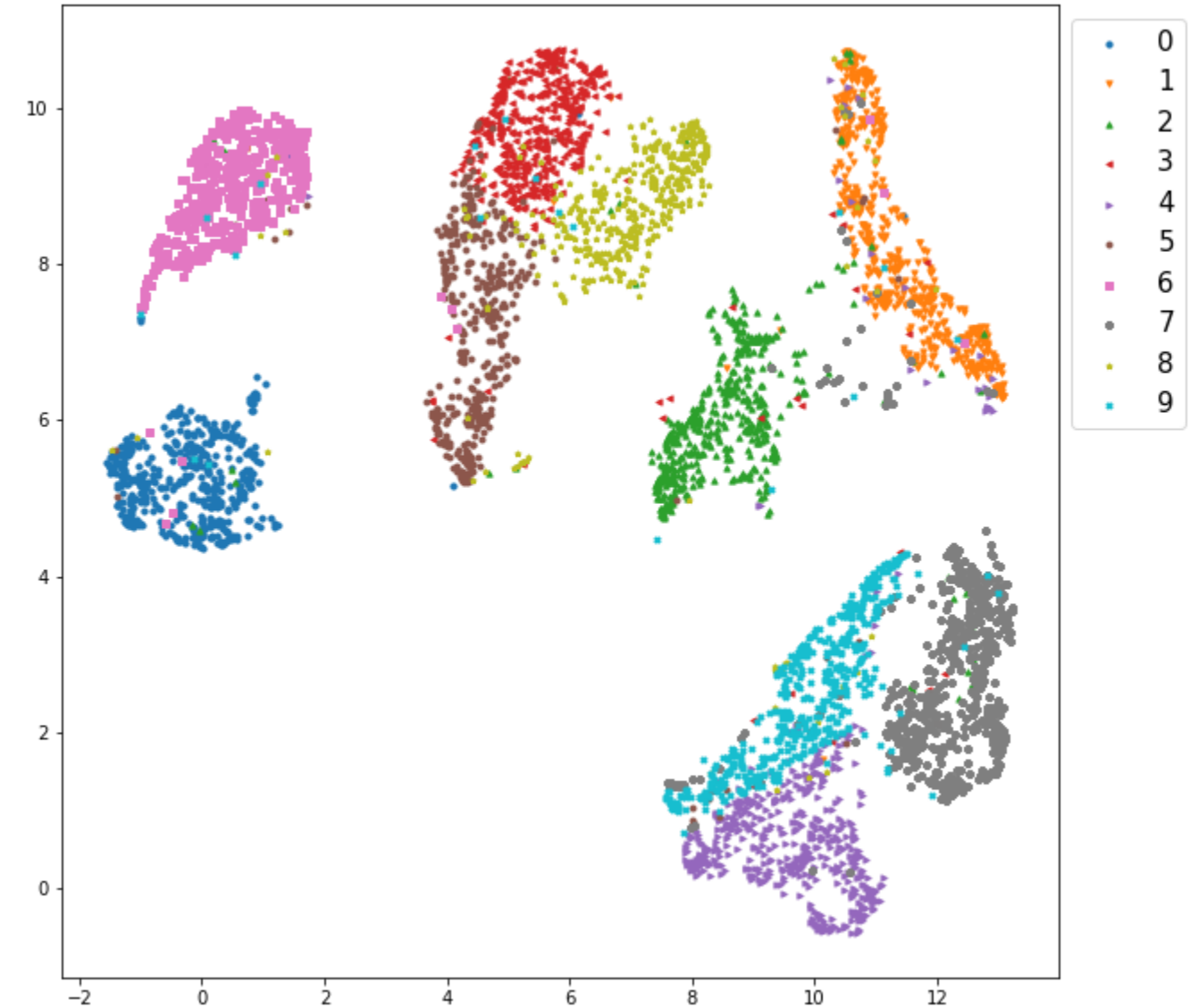


# # 3. 여러 가지 차원 축소 기법 적용하기

## 6) UMAP(Uniform Manifold Approximation and Projection)

### 시각화

```
plt.figure(figsize = (10,10))
for i,marker in enumerate(markers):
    mask = train_y == i
    plt.scatter(x_umap[mask, 0], x_umap[mask, 1], label = i, s = 10, alpha = 1,marker = marker)
plt.legend(bbox_to_anchor = (1.00, 1), loc = 'upper left',fontsize = 15)
```



- t-SNE에 비해 각 숫자들의 클러스터링 경계가 더 뚜렷하게 나타남을 확인할 수 있음
- 숫자들이 대체로 잘 구분되지만 숫자 3과 5와 8, 숫자 4와 7과 9의 경우 클러스터링 경계가 일부 겹쳐 해당 클러스터의 샘플들 중 몇 개가 혼동을 일으킬 가능성이 존재함을 확인할 수 있음



# # 3. 여러 가지 차원 축소 기법 적용하기

## 6) UMAP(Uniform Manifold Approximation and Projection)

### 📌 UMAP connectivity plot

- 데이터가 샘플링 되었을 수 있는 대략적인 매니폴드의 중간 위상 표현을 구성함으로써 작동
- 해당 구조를 가중 그래프로 단순화할 수 있음
- 때때로 결과 embedding과 관련하여 그래프(다양체에서 연결성을 나타냄)가 어떻게 보이는지 보는 것이 유익할 수 있음
- embedding을 더 잘 이해하고 진단 목적으로 사용할 수 있음

# # 3. 여러 가지 차원 축소 기법 적용하기

## 6-1) UMAP Connectivity Plot

```
### 라이브러리 설치
```

```
!pip install umap-learn[plot]
```

```
### 라이브러리 import
```

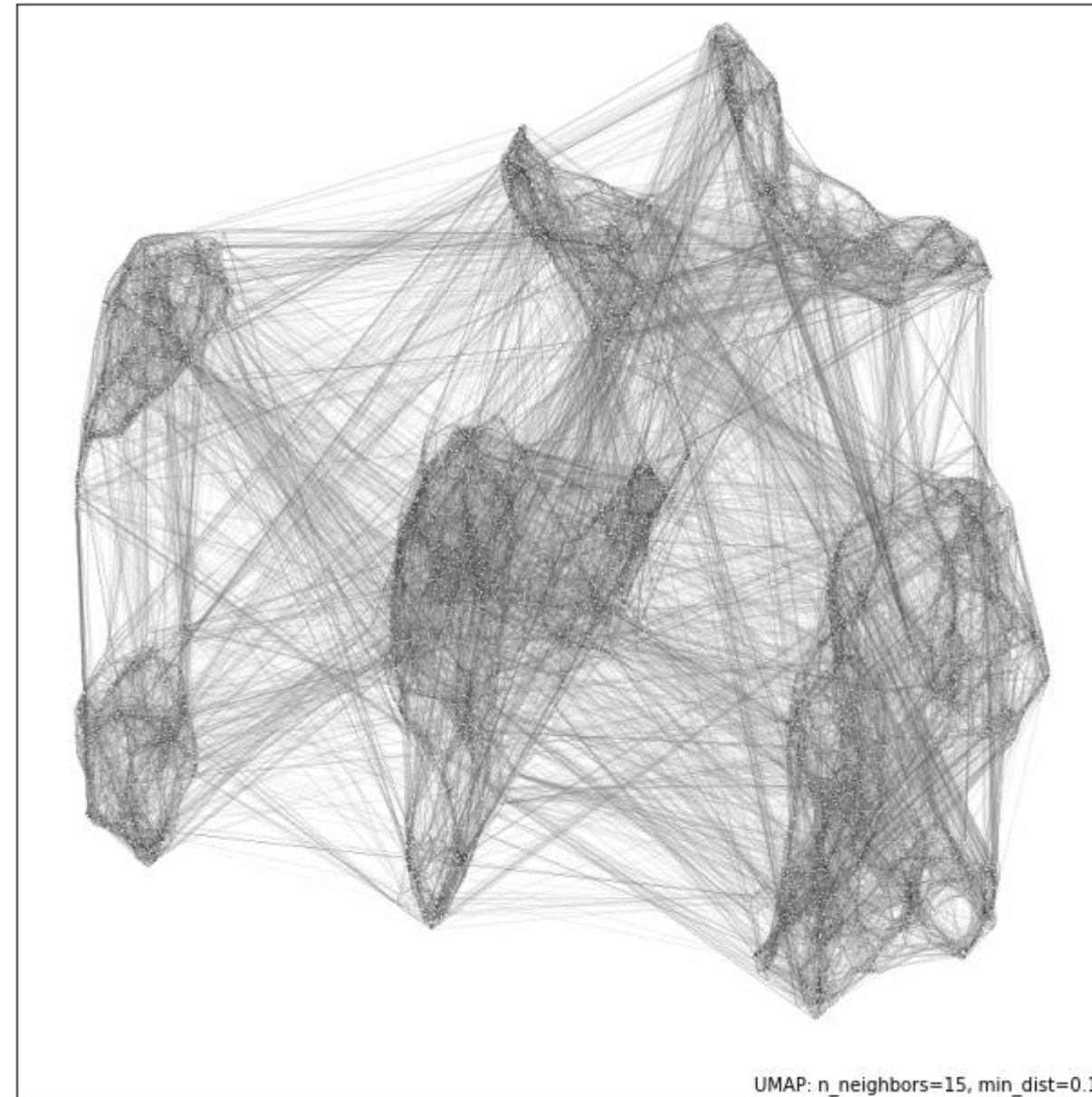
```
import umap.plot
```

```
### 차원 축소
```

```
mapper = umap.UMAP().fit(train_x)
```

```
### Plotting
```

```
umap.plot.connectivity(mapper, show_points = True)
```

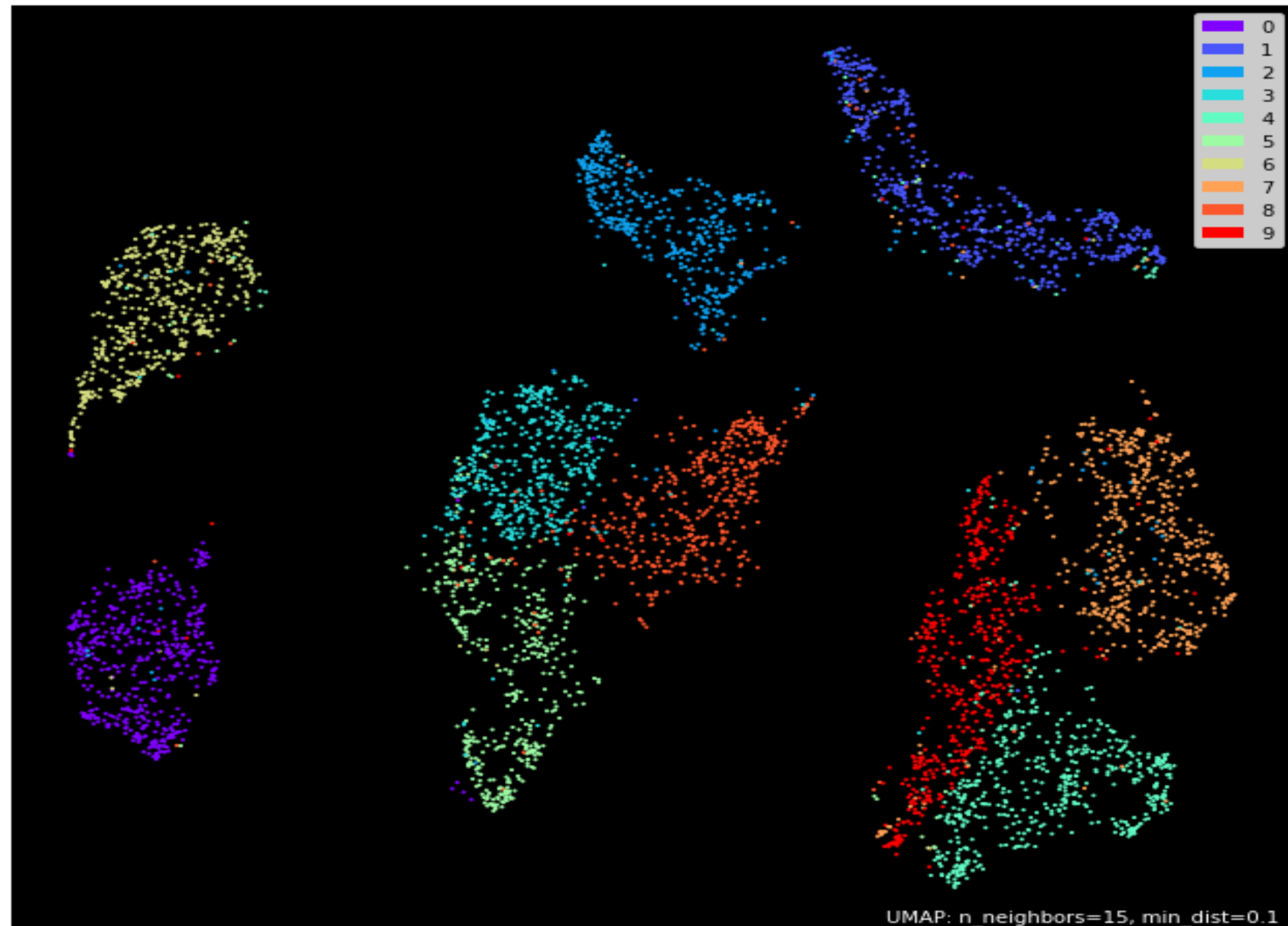


# # 3. 여러 가지 차원 축소 기법 적용하기

## 6) UMAP(Uniform Manifold Approximation and Projection)

📌 Another UMAP Plot

```
umap.plot.points(mapper, labels = train_y, theme = 'fire')
```



# # 3. 여러 가지 차원 축소 기법 적용하기

## 6) UMAP(Uniform Manifold Approximation and Projection)

### 📌 UMAP 3D Plot

```
### 라이브러리 import
```

```
import plotly
import plotly.express as px
from umap import UMAP
```

```
### 차원 축소
```

```
umap_3d = UMAP(n_components = 3, init = 'random', random_state = 0)
x_umap = umap_3d.fit_transform(train_x)
```

```
### 시각화를 위해 필요한 DataFrame 생성하기
```

```
umap_df = pd.DataFrame([x_umap])
train_y_sr = pd.Series(train_y, name = 'label')
print(type(x_umap))
```

```
<class 'numpy.ndarray'>
```

```
new_df = pd.concat([umap_df, train_y_sr], axis = 1)
new_df
```

	0	1	2	label
0	2.883474	1.497582	8.677372	5
1	2.395933	10.447811	1.046121	0
2	2.977624	4.968257	4.293287	4
3	9.695361	3.686780	7.597613	1
4	4.760168	4.342690	5.267987	9
...	...	...	...	...
4995	6.973429	3.377479	4.079491	7
4996	3.830196	1.181673	10.180694	3
4997	7.902909	0.685053	7.101771	2
4998	7.486504	3.487115	9.325881	1
4999	7.715492	1.235513	7.302202	2

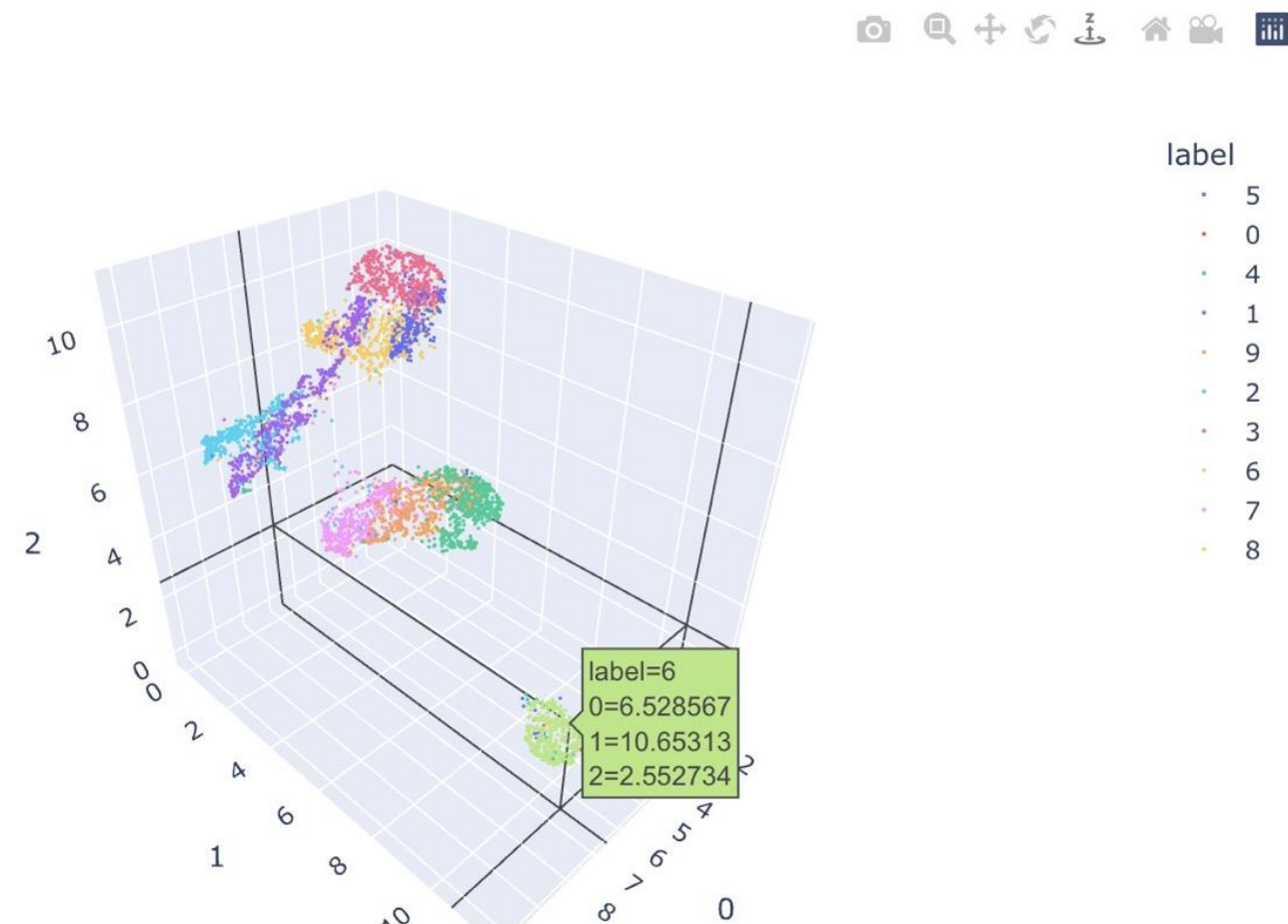
```
5000 rows x 4 columns
```

# # 3. 여러 가지 차원 축소 기법 적용하기

## 6-3) UMAP 3D Plot

### 시각화

```
fig = px.scatter_3d(new_df, x = 0, y = 1, z = 2, color = 'label', labels = {'color': 'number'})  
fig.update_traces(marker_size = 1)  
fig.show()
```



- 해당 그래프는 UMAP 차원축소 기법을 이용하여 고차원 데이터를 3차원으로 축소한 것을 나타낸 그래프임
- 이전의 2차원 plot과 비교 시 시각적으로 더 복잡하고 점들이 공간 상에서 더 희소(sparse)한 것을 확인할 수 있음
- 차원이 증가함에 따라 결정 경계의 수가 증가하고, 데이터의 복잡도가 증가할 것임  
→ 차원의 저주에 빠질 수 있음

# # 4. AutoEncoder

- 신경망(NN)을 이용한 차원 축소 방식
- 입력 데이터의 차원보다 작은 중간 레이어를 사용하여 입력과 동일한 값을 출력하는 신경망을 학습함  
→ 원본 데이터를 재현할 수 있는 저차원 표현을 학습
- 항상 Encoder와 Decoder의 두 파트로 구현됨
  - 1) Encoder(인식 네트워크): 입력 -> 내부 표현
  - 2) Decoder(생성 네트워크): 내부 표현 -> 출력

### 3. 차원축소 기법들





# 3.1 커널 소개

## Dataset Decomposition Techniques

### Problem Statement: Santander Value Prediction

Santander Group wants to identify the value of transactions for each potential customer. This is a first step that Santander needs to nail in order to personalize their services at scale. The dataset can be downloaded from this [link](#). In this kernel I have explained different approaches for dataset decomposition.

### Introduction

The purpose of this kernel is to walkthrough different dataset decomposition techniques and their implementations. Decomposition of dataset into lower dimensions often becomes an important task while dealing with datasets having larger number of features. Dimensionality Reduction refers to the process of converting a dataset having vast dimensions into a dataset with lesser number of dimensions. This process is done by ensuring that the information conveyed by the original dataset is not lost.



“이 커널의 목적은 다양한 데이터셋 분해 기술들과 그 구현을 살펴보는 것이다.”

→ PCA, ICA, Factor Analysis, t-SNE



# 3.2 Data Description

```
train = pd.read_csv('train.csv')

target = train['target']
train = train.drop(["target", "ID"], axis=1)

print ("Rows: " + str(train.shape[0]) + ", Columns: " + str(train.shape[1]))
train.head()
```

# 데이터 로드

Rows: 4459, Columns: 4991

	48df886f9	0deb4b6a8	34b15f335	a8cb14b00	2f0771a37	30347e683	d08d1fbe3	6ee66e115	20aa07010	dc5a8f1d8	...	3ecc09859	9281abeea	8675bec
0	0.0	0	0.0	0	0	0	0	0	0.0	0.0	...	0.0	0.0	(
1	0.0	0	0.0	0	0	0	0	0	2200000.0	0.0	...	0.0	0.0	(
2	0.0	0	0.0	0	0	0	0	0	0.0	0.0	...	0.0	0.0	(
3	0.0	0	0.0	0	0	0	0	0	0.0	0.0	...	0.0	0.0	(
4	0.0	0	0.0	0	0	0	0	0	2000000.0	0.0	...	0.0	0.0	(

-> 행 4459개, 열 4991개

# 3.2 Data Description

```
from sklearn.preprocessing import StandardScaler

standardized_train = StandardScaler().fit_transform(train.values)
```

# 정규화

```
feature_df = train.describe().T
feature_df = feature_df.reset_index().rename(columns = {'index' : 'columns'})
feature_df['distinct_vals'] = feature_df['columns'].apply(lambda x : len(train[x].value_counts()))
feature_df['column_var'] = feature_df['columns'].apply(lambda x : np.var(train[x]))
feature_df['column_std'] = feature_df['columns'].apply(lambda x : np.std(train[x]))
feature_df['column_mean'] = feature_df['columns'].apply(lambda x : np.mean(train[x]))
feature_df['target_corr'] = feature_df['columns'].apply(lambda x : np.corrcoef(target, train[x])[0][1])
feature_df.head()
```

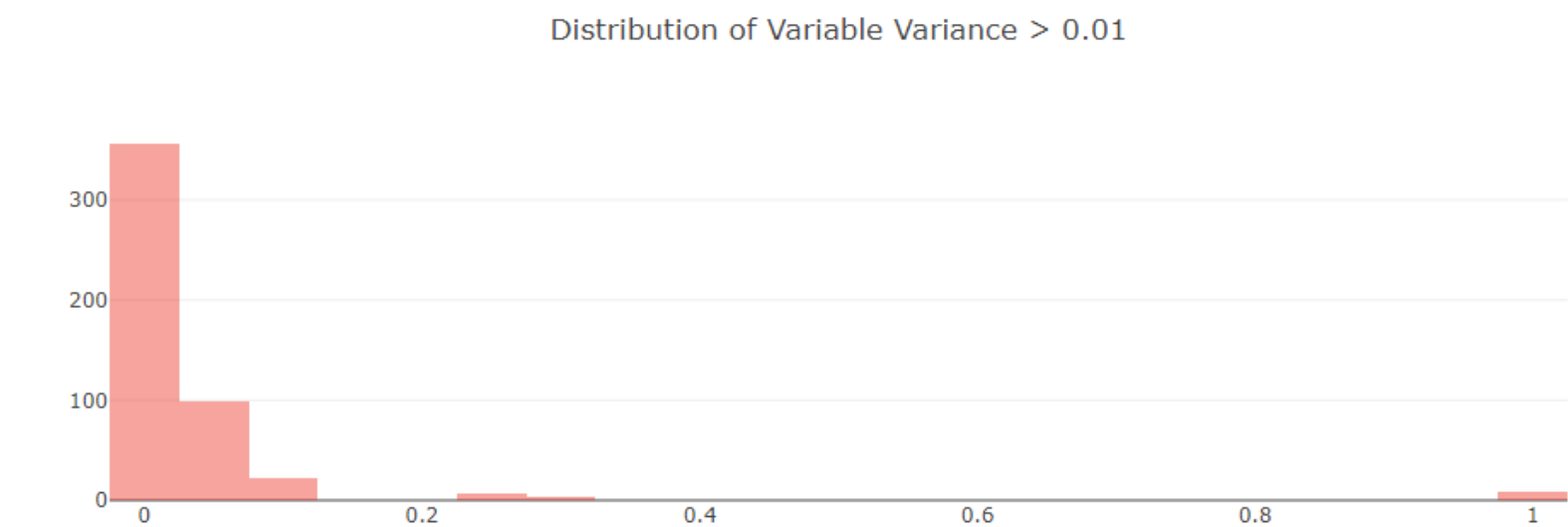
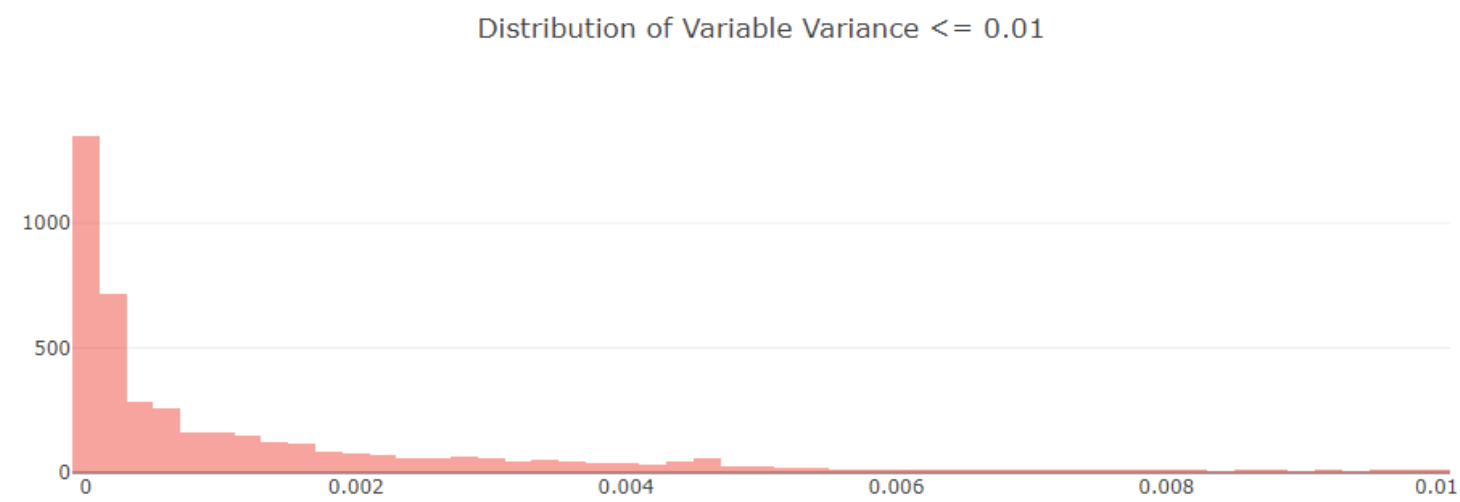
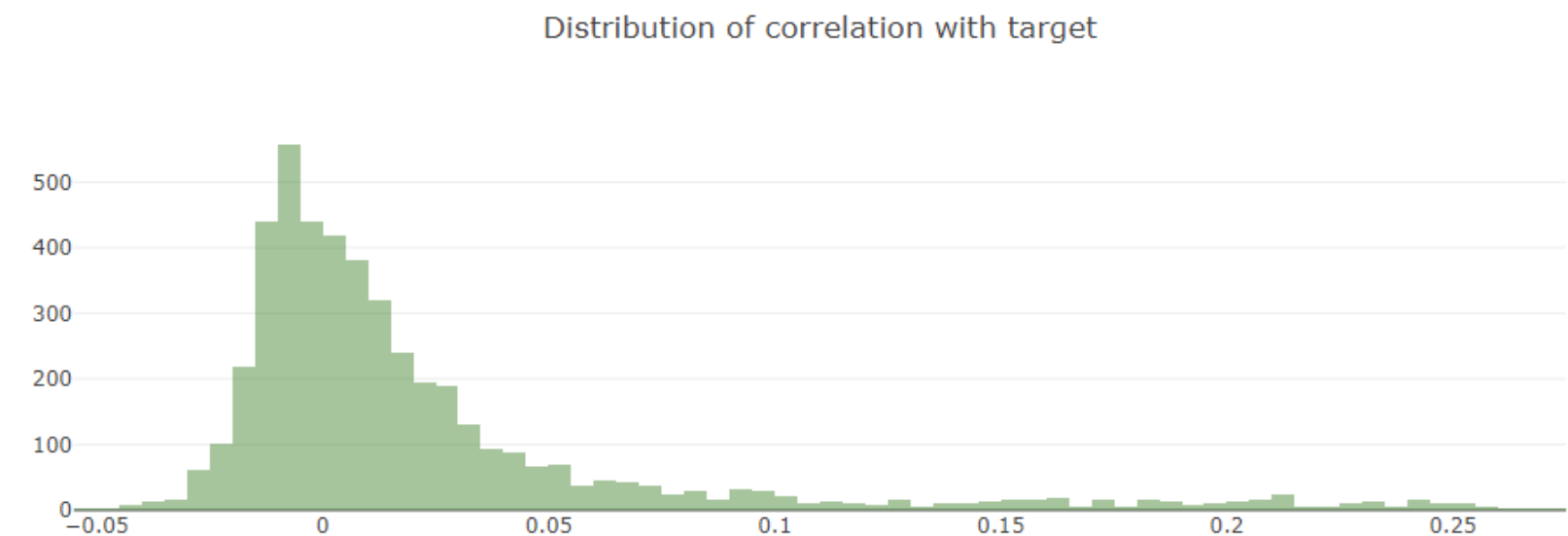
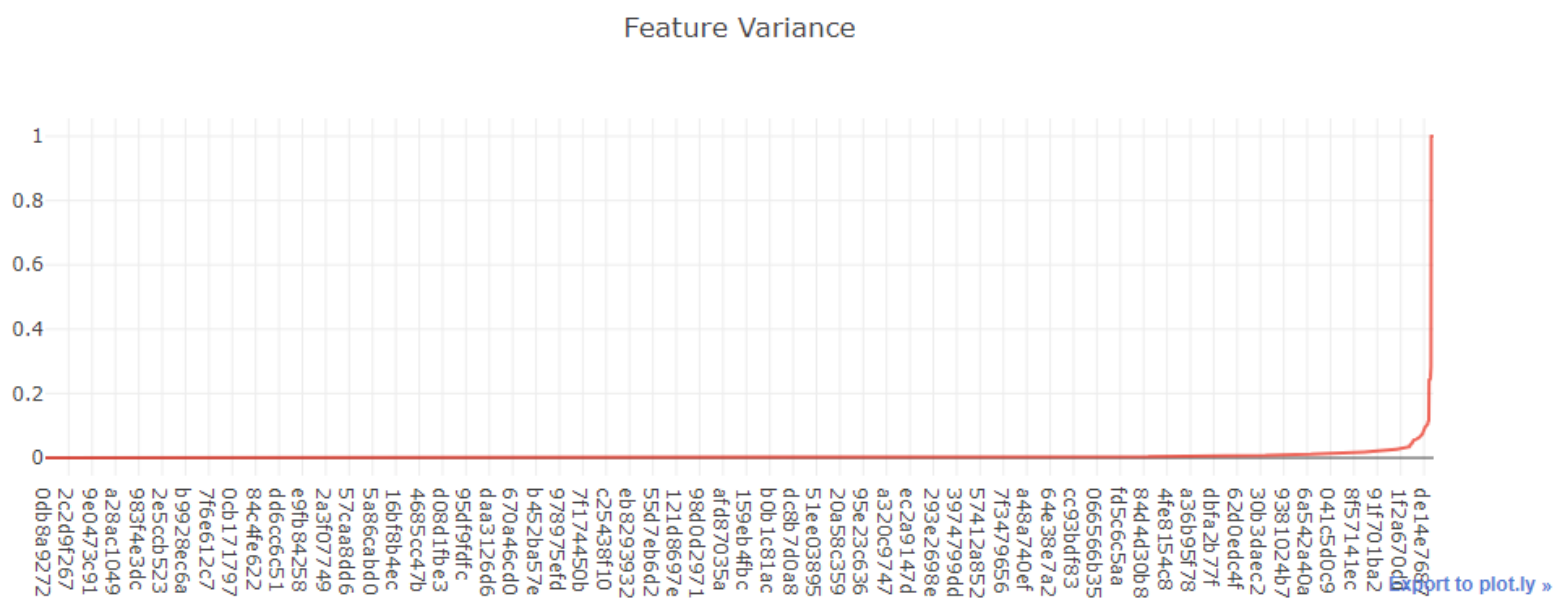
# 각 칼럼의 통계량에 대한 표 만들기

	columns	count	mean	std	min	25%	50%	75%	max	distinct_vals	column_var	column_std	column_mean	target_c
0	48df886f9	4459.0	14654.930101	3.893298e+05	0.0	0.0	0.0	0.0	20000000.0	32	1.515437e+11	3.892862e+05	14654.930101	0.0101
1	0deb4b6a8	4459.0	1390.894819	6.428302e+04	0.0	0.0	0.0	0.0	4000000.0	5	4.131381e+09	6.427582e+04	1390.894819	0.0138
2	34b15f335	4459.0	26722.450922	5.699652e+05	0.0	0.0	0.0	0.0	20000000.0	29	3.247875e+11	5.699013e+05	26722.450922	0.0146
3	a8cb14b00	4459.0	4530.163714	2.359124e+05	0.0	0.0	0.0	0.0	14800000.0	3	5.564218e+10	2.358860e+05	4530.163714	-0.0029
4	2f0771a37	4459.0	26409.957390	1.514730e+06	0.0	0.0	0.0	0.0	100000000.0	6	2.293893e+12	1.514560e+06	26409.957390	0.0166

```
len(feature_df[feature_df['column_var'].astype(float) == 0.0])
```

# 분산이 0인 칼럼의 개수

# 3.2 Data Description



# 3.3 $TV=\lambda V$ 고유벡터와 고유값 분해

```
# Calculating Eigenvectors and eigenvalues of Cov matrix
mean_vec = np.mean(standardized_train, axis=0)
cov_matrix = np.cov(standardized_train.T)
eig_vals, eig_vecs = np.linalg.eig(cov_matrix)

# Create a list of (eigenvalue, eigenvector) tuples
eig_pairs = [ (np.abs(eig_vals[i]),eig_vecs[:,i]) for i in range(len(eig_vals))]

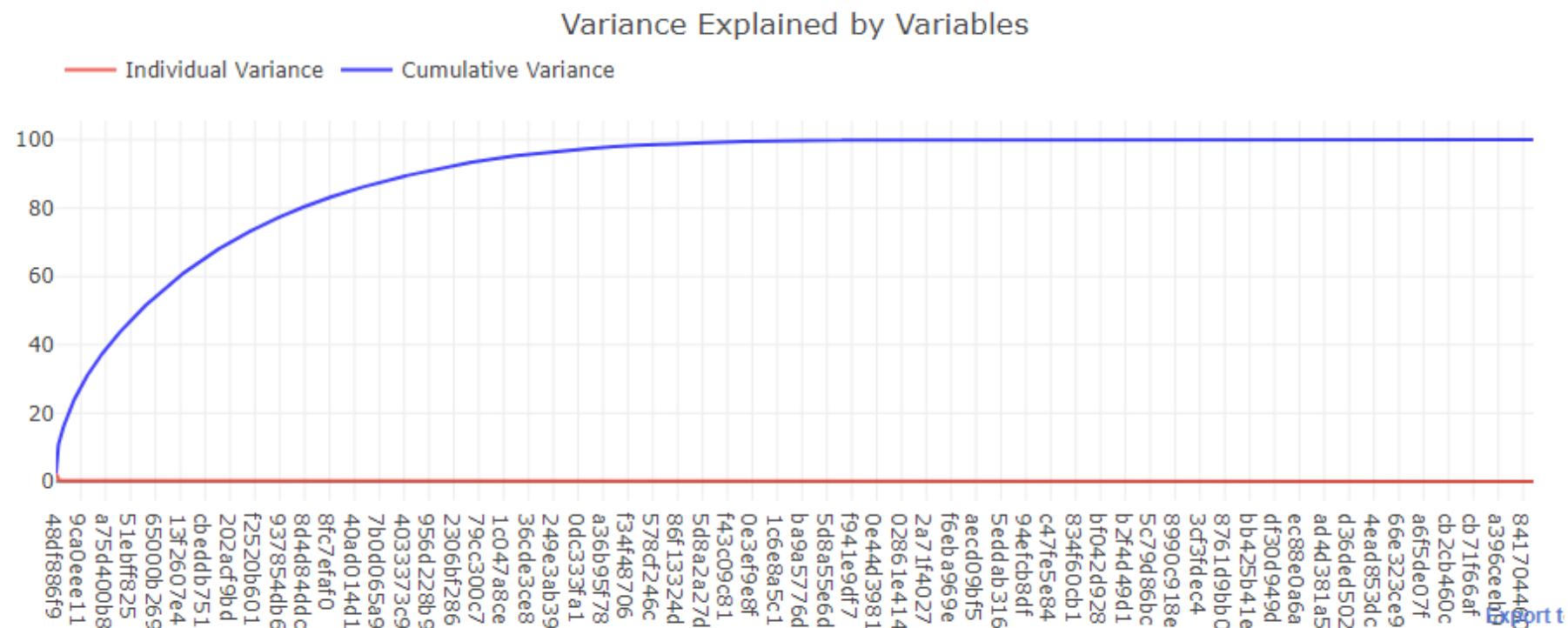
# Sort the eigenvalue, eigenvector pair from high to low
eig_pairs.sort(key = lambda x: x[0], reverse=True)

# Calculation of Explained Variance from the eigenvalues
tot = sum(eig_vals)

# Individual explained variance
var_exp = [(i/tot)*100 for i in sorted(eig_vals, reverse=True)]
var_exp_real = [v.real for v in var_exp]

# Cumulative explained variance
cum_var_exp = np.cumsum(var_exp)
cum_exp_real = [v.real for v in cum_var_exp]

## plot the variance and cumulative variance
trace1 = go.Scatter(x=train.columns, y=var_exp_real, name="Individual Variance", opacity=0.75, marker=dict(color="red"))
trace2 = go.Scatter(x=train.columns, y=cum_exp_real, name="Cumulative Variance", opacity=0.75, marker=dict(color="blue"))
layout = dict(height=400, title='Variance Explained by Variables', legend=dict(orientation="h", x=0, y=1.2));
fig = go.Figure(data=[trace1, trace2], layout=layout);
iplot(fig);
```



# 3.4.1 PCA

## # PCA (Principal Component Analysis)

- 가장 대표적인 차원축소 기법 중 하나
- 여러 변수 간에 존재하는 상관 관계를 이용해 이를 대표하는 주성분을 추출해서 차원을 축소하는 기법
- 분산이 데이터의 특성을 가장 잘 나타내는 것으로 간주하여 가장 높은 분산을 가지는 데이터의 축을 찾아 이 축으로 차원을 축소한다.
- 입력 데이터의 공분산 행렬은 고유벡터와 고유값으로 분해될 수 있으며, 이렇게 분해된 고유벡터를 이용해 입력 데이터를 선형 변환하는 방식

1. 입력 데이터 세트의 공분산 행렬을 생성한다.
2. 공분산 행렬의 고유벡터와 고유값을 계산한다.
3. 고유값이 가장 큰 순으로 k개만큼 고유벡터를 추출한다.
4. 고유값이 가장 큰 순으로 추출된 고유벡터를 이용해 새롭게 입력 데이터를 변환한다.

# 3.4.1 PCA

```
def _get_number_components(model, threshold):
    component_variance = model.explained_variance_ratio_
    explained_variance = 0.0
    components = 0

    for var in component_variance:
        explained_variance += var
        components += 1
        if(explained_variance >= threshold):
            break
    return components

### Get the optimal number of components
pca = PCA()
train_pca = pca.fit_transform(standardized_train)
components = _get_number_components(pca, threshold=0.85)
components
```

# 적절한 k개 찾기

993

-> Threshold 0.85에서 993개의 피처를 추출할 수 있으며, 이 피처들은 데이터셋의 분산의 85%를 설명할 수 있다.

# 3.4.1 PCA

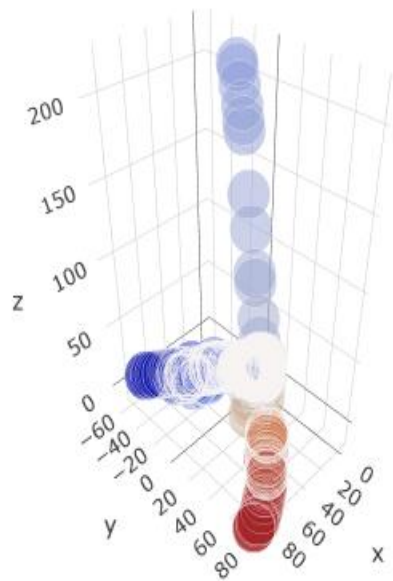
```
### Implement PCA
obj_pca = model = PCA(n_components = components)
X_pca = obj_pca.fit_transform(standardized_train)
```

# PCA 변환 수행

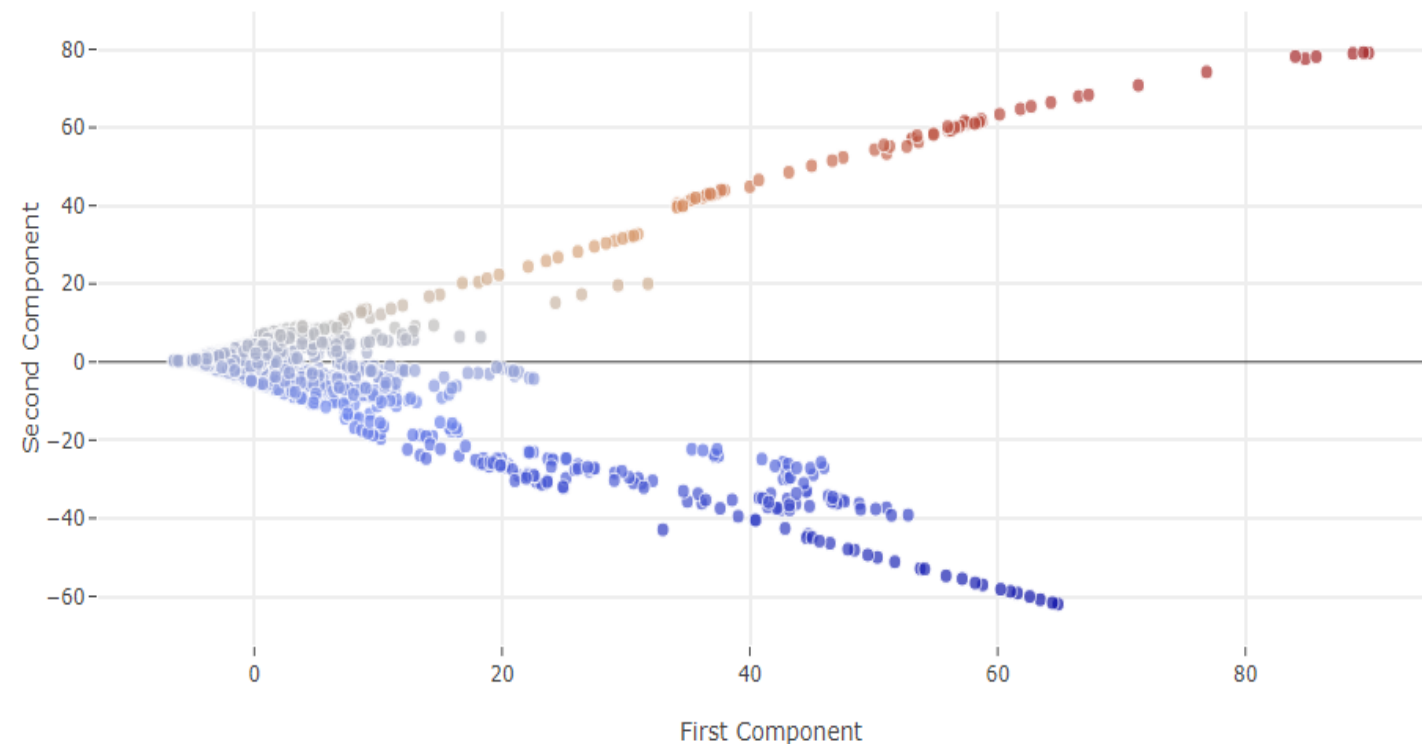
n\_components는 PCA로 변환할 차원의 수이며, fit\_transform으로 PCA 변환 데이터 반환

```
## Visualize the Components
plot_3_components(X_pca, 'PCA - First Three Component')
plot_2_components(X_pca, 'PCA - First Two Components')
```

PCA - First Three Component



PCA - First Two Components



구성 요소의 주요 속성이 서로 직교하고 있음  
-> 서로 상관관계가 없다.

# 3.4.2 Kernel PCA

## # Kernel PCA

- 커널 트릭을 PCA에 적용해 차원축소를 위한 복잡한 비선형 투영을 수행
- 커널 트릭: 선형 분류가 불가능한 데이터에 대한 처리를 위해 데이터의 차원을 증가시켜 하나의 초평면으로 분류할 수 있도록 도와주는 커널 함수
- 커널 PCA로 비선형 매핑을 수행해서 고차원 공간으로 변환하고 표준 PCA로 샘플이 선형 분류기로 구분될 수 있는 저차원으로 데이터를 투영시킨다. 이때 계산 비용이 매우 비싸다는 단점이 있는데, 이때 커널 트릭을 사용하여 원본 특성 공간에서 두 고차원 특성 벡터의 유사도를 계산할 수 있다.



# 3.4.2 Kernel PCA

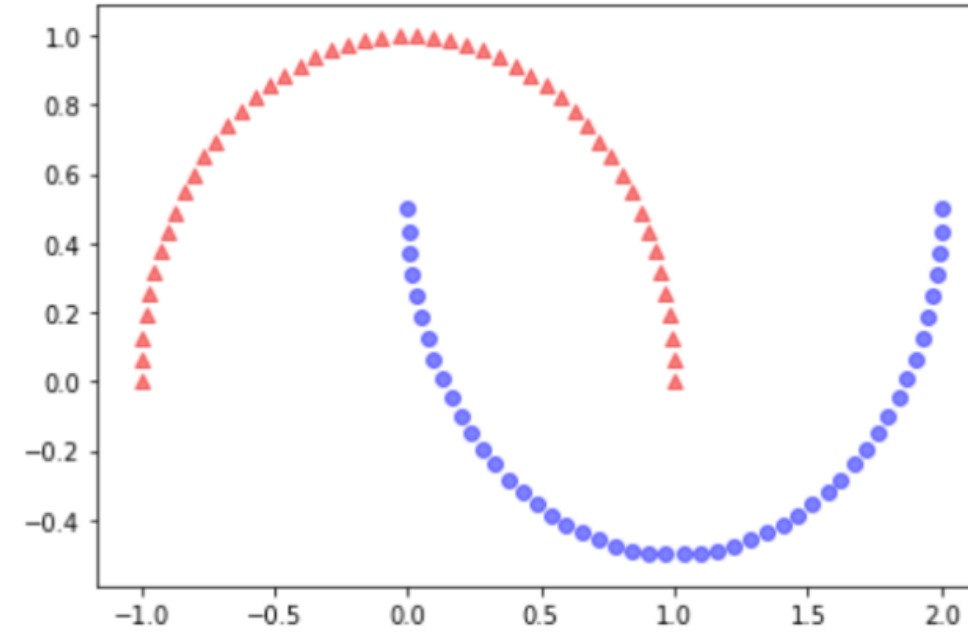
```
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100, random_state=123)

plt.scatter(X[y == 0, 0], X[y == 0, 1], color='red', marker='^', alpha=0.5)
plt.scatter(X[y == 1, 0], X[y == 1, 1], color='blue', marker='o', alpha=0.5)

plt.tight_layout()
plt.show()
```

# 반달모양 데이터셋 로드

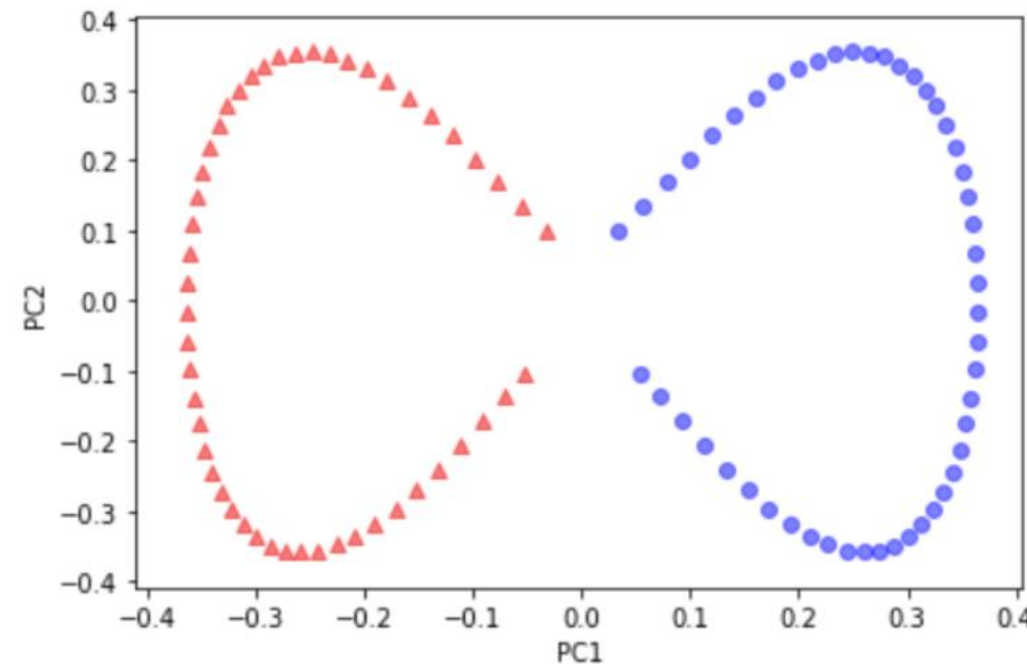


```
from sklearn.decomposition import KernelPCA

X, y = make_moons(n_samples=100, random_state=123)
scikit_kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)
X_skernpca = scikit_kpca.fit_transform(X)

plt.scatter(X_skernpca[y == 0, 0], X_skernpca[y == 0, 1],
            color='red', marker='^', alpha=0.5)
plt.scatter(X_skernpca[y == 1, 0], X_skernpca[y == 1, 1],
            color='blue', marker='o', alpha=0.5)

plt.xlabel('PC1')
plt.ylabel('PC2')
plt.tight_layout()
plt.show()
```



# 사이킷런으로 커널 PCA 수행

# 3.4.3 Incremental PCA

## # Incremental PCA

- SVD를 수행하기 위해서는 전체 학습 데이터셋을 메모리에 올려야 한다는 단점을 보완하기 위해 개발된 PCA 알고리즘
- 학습 데이터셋을 미니배치로 나눈 뒤 IPCA 알고리즘에 하나의 미니배치를 입력으로 넣어준다.
- 학습 데이터셋이 클 때 유용

```
from sklearn.decomposition import IncrementalPCA

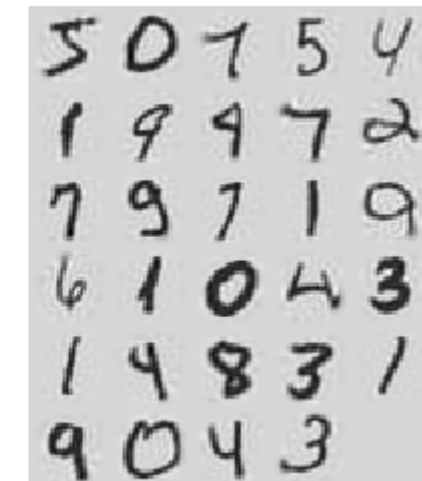
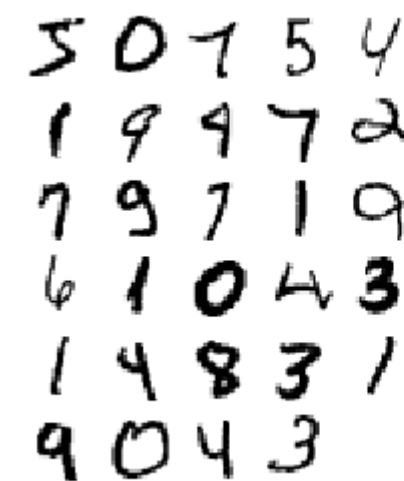
n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for batch_x in np.array_split(train_x, n_batches):
    print(".", end=" ") # not shown in the book
    inc_pca.partial_fit(batch_x)

X_reduced = inc_pca.transform(train_x)
```

```
X_recovered_inc_pca = inc_pca.inverse_transform(X_reduced)

plt.figure(figsize=(7, 4))
plt.subplot(121)
plot_digits(train_x[:2100])
plt.subplot(122)
plot_digits(X_recovered_inc_pca[:2100])
plt.tight_layout()
```

# MNIST 데이터를 100개의 미니배치로 나눠 Incremental PCA 수행



# 3.4.4 Sparse PCA

## # Sparse PCA

- 데이터를 최적으로 재구성할 수 있는 희소 구성 요소 집합을 찾는 방법
- 희소성의 정도는 조정 가능한 파라미터이다.

```
from sklearn.decomposition import SparsePCA
from sklearn.datasets import load_digits
digits = load_digits()
print(digits.data.shape)
```

# 데이터 로드

```
(1797, 64)
```

```
sparse_pca = SparsePCA(n_components=60, alpha=0.1)
sparse_pca.fit_transform(digits.data / 255)
print(sparse_pca.components_.shape)
```

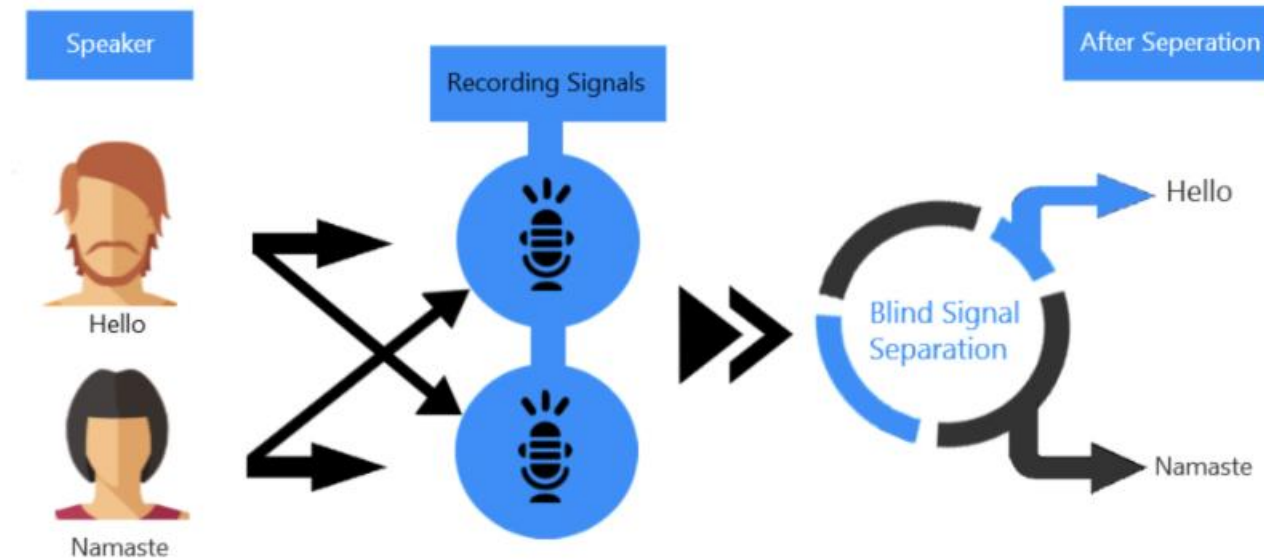
# SparsePCA 적용

```
(60, 64)
```

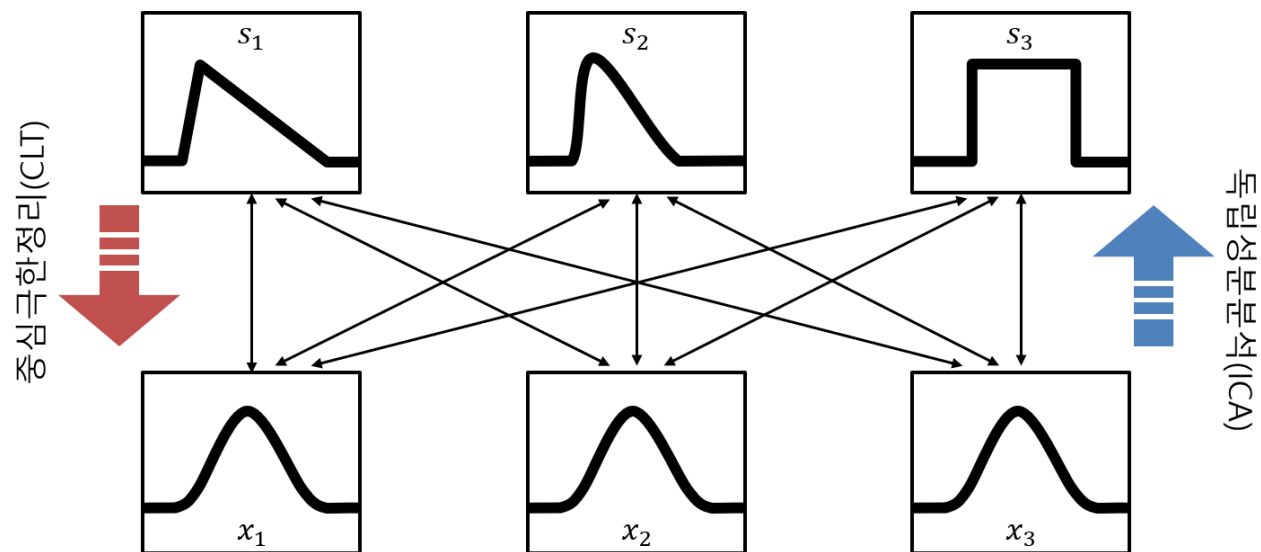
# 3.5 ICA

## # ICA (Independent Component Analysis)

- 다변량의 신호를 통계적으로 독립적인 하부 성분으로 분리하는 계산 방법
- 가정: source들이 서로 독립적이다
- 차원 축소보다는 개별 요소를 분리할 때 사용된다.



## 중심극한정리(CLT)와 독립성분분석(ICA) 관계 개요도



→ 서로 독립적인 랜덤 변수들의 분포의 선형조합은 가우스 분포를 따른다.

→ (s에 비해 더 가우스 분포를 따르는) x들을 어떻게 조합하면 s를 얻을까?

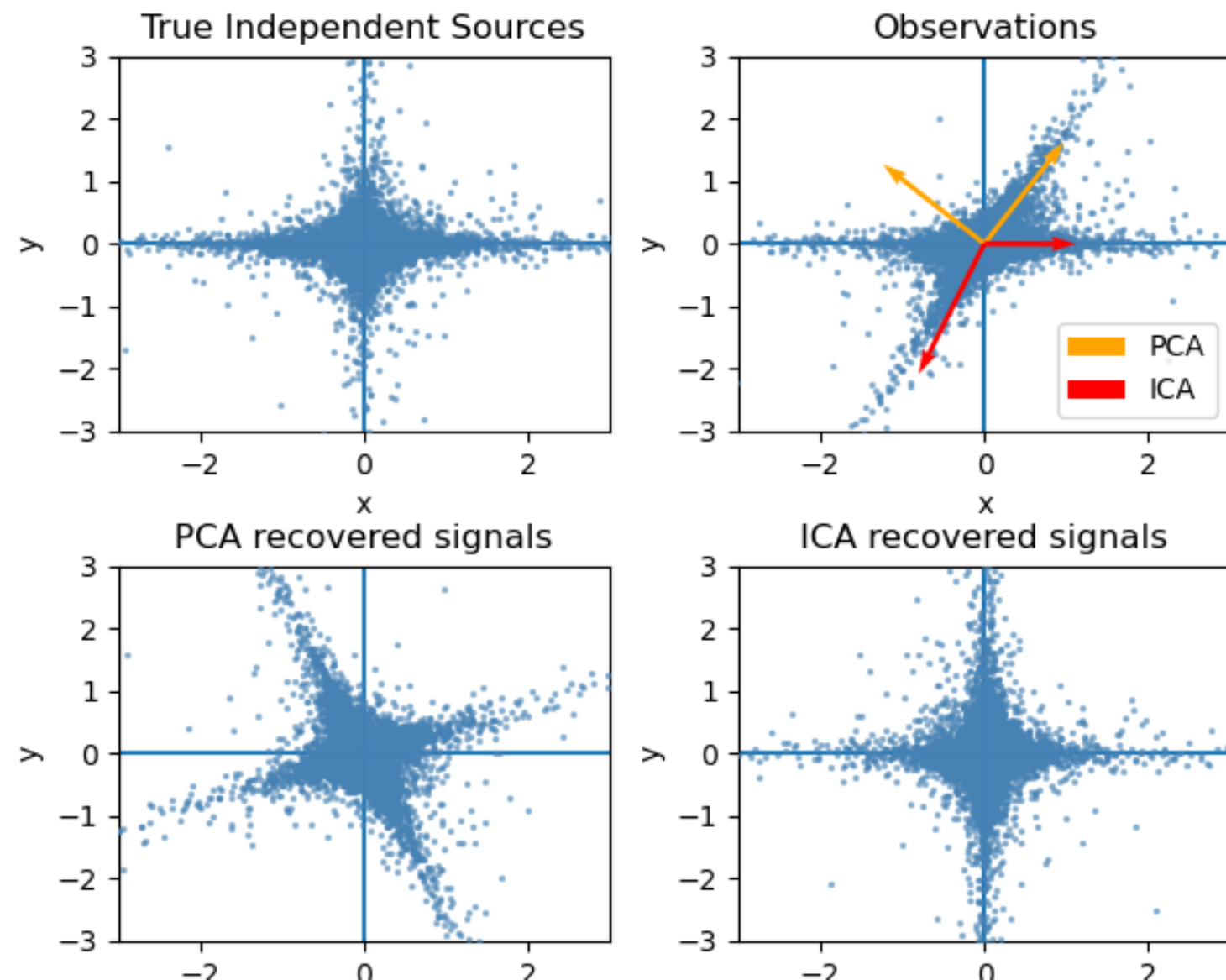
ICA를 중심극한정리의 반대 과정으로 생각해본다면, ICA는 독립 랜덤 변수들의 조합으로 얻어진  $x$ 에 적절한 행렬을 곱해 원래의 독립 랜덤 변수들인 source들  $s$ 를 찾는 과정이다.

→ source들이 서로 독립적이라는 가정을 최대한 만족할 수 있도록 하는 행렬을 찾는 것이 목적

# 3.5 ICA

## # PCA와 ICA의 비교

- 공통점: 주어진 데이터를 대표하는 기저벡터를 찾아준다.
- 차이점:
  - PCA는 속성 공간에서 직교하는 기저벡터 집합을 찾아준다.  
데이터를 정사영했을 때 최대 분산을 얻을 수 있는 벡터를 차례대로 기저벡터로 삼는다.
  - ICA를 통해 찾은 기저벡터들은 서로 직교하지 않을 수도 있다.  
ICA를 통해 얻은 기저벡터들의 데이터를 정사영했을 때 그 결과들이 최대한 독립적일 수 있도록 하는 벡터들을 기저벡터로 삼는다.

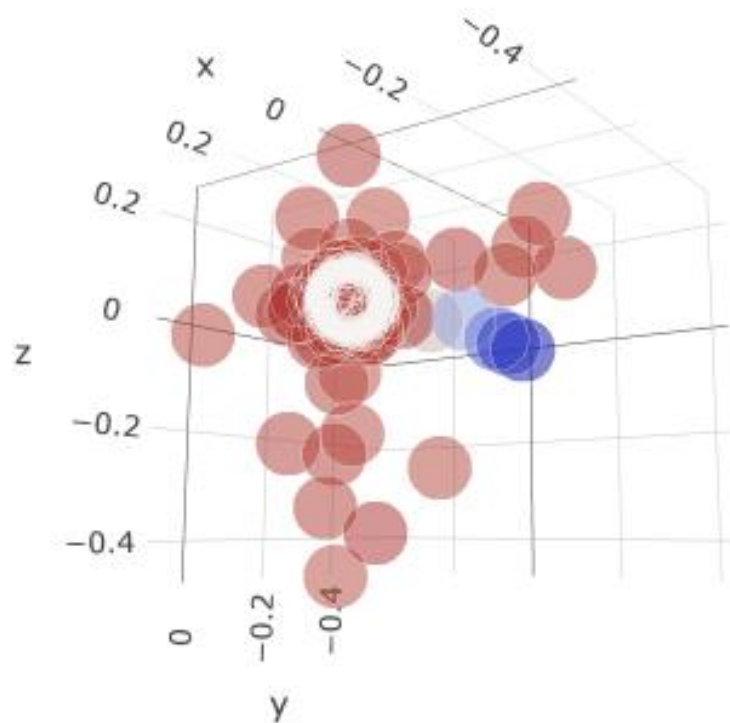


# 3.5 ICA

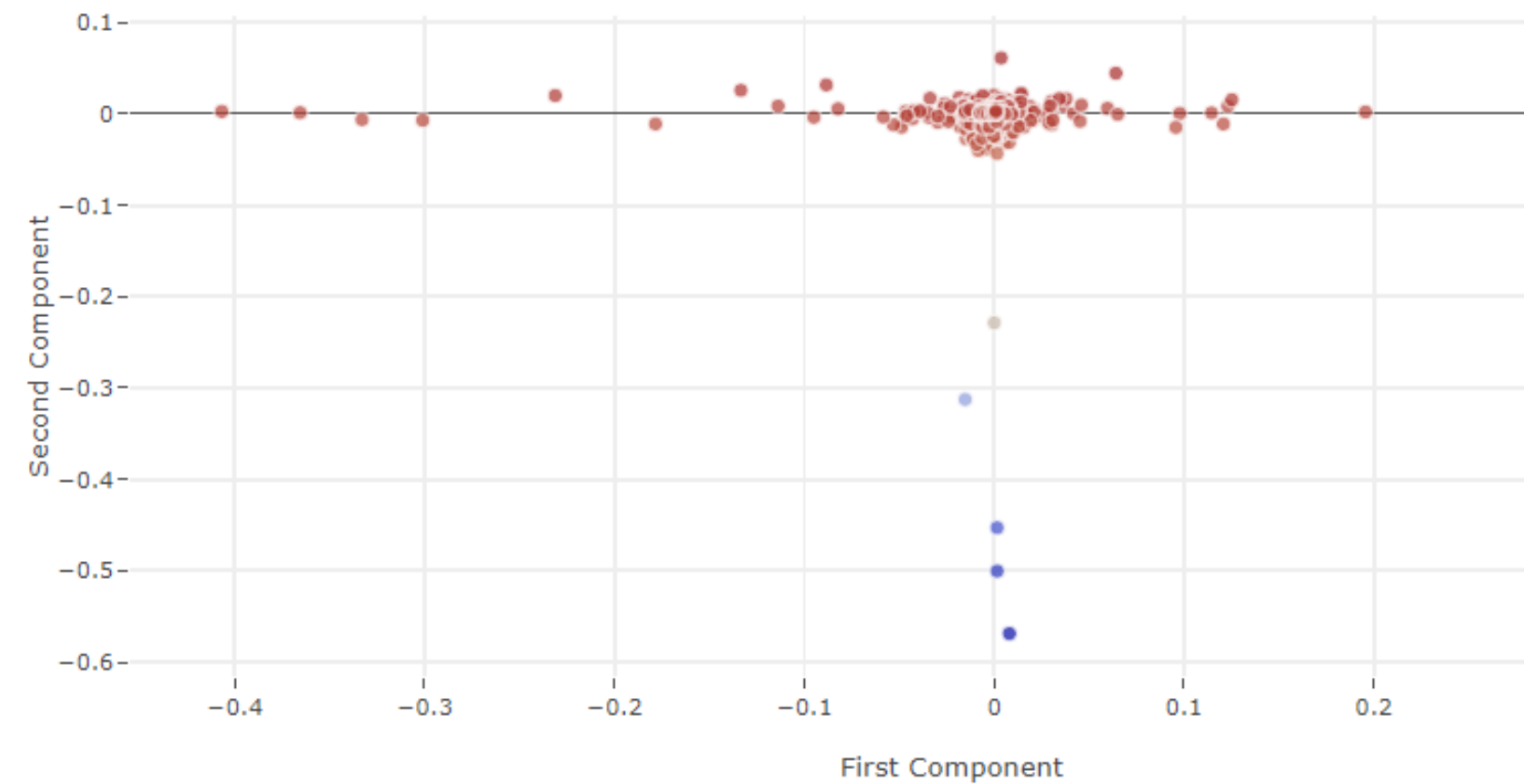
```
### Implement ICA
obj_ica = FastICA(n_components = 30)
X_ica = obj_ica.fit_transform(standardized_train)

## Visualize the Components
plot_3_components(X_ica, 'ICA - First three components')
plot_2_components(X_ica, 'ICA - First two components')
```

ICA - First three components



ICA - First two components



-> 정사영했을 때 수직이 된다. 최대독립을 만드는 basis는 직교하지 않을 수 있다.

# 3.6 Factor Analysis

## # Factor Analysis

- 수많은 변수들 중에서 잠재된 몇 개의 변수(요인)를 찾아내는 것
  - 예시) 학생들의 시험 성적 데이터
  - 이 데이터가 수학, 과학, 영어, 중국어, 독어, 작곡, 연주의 점수(0~100점)으로 구성되어 있다고 하면,
  - 수학, 과학은 상관관계가 있을 것이고 (수리계산능력)
  - 영어, 중국어, 독어가 상관관계가 있을 것이고 (외국어능력)
  - 작곡, 연주가 상관관계가 있을 것이다. (음악적 능력)
- > 원래 7개의 변수로 구성되어 있지만 **내부적으로는 3개의 잠재변수**로 구성된 것을 파악할 수 있다.

가정:

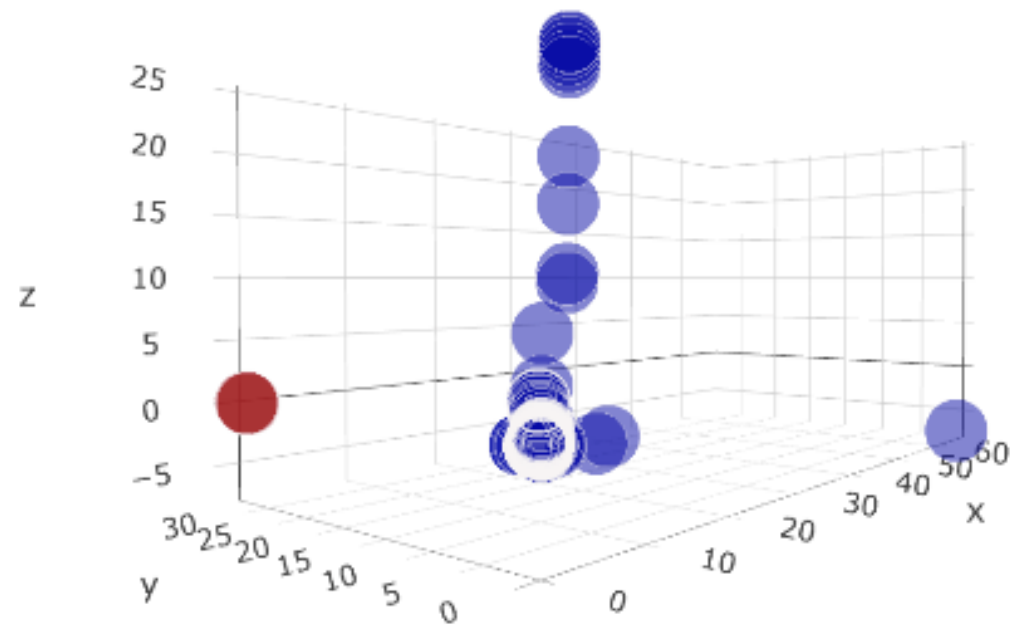
1. 데이터에는 특이치가 없다.
2. 표본 크기는 요인보다 커야 한다.
3. 완벽한 다중공선성은 없어야 한다. (변수들 간 상관관계가 있지만, 다른 변수의 선형결합으로 표현되면 안 됨)
4. 변수들 사이에 동질성이 있어서는 안 된다.

# 3.6 Factor Analysis

```
### Implement Factor Analysis
obj_fa = FactorAnalysis(n_components = 30)
X_fa = obj_fa.fit_transform(standardized_train)

## Visualize the Components
plot_3_components(X_fa, 'Factor Analysis - First three components')
```

Factor Analysis - First three components



-> 잠재 공간의 차원 변환 후 얻은 X의 구성 요소 수가 30개이고, 그 중 3개의 주요 잠재요소에 대한 그래프이다.

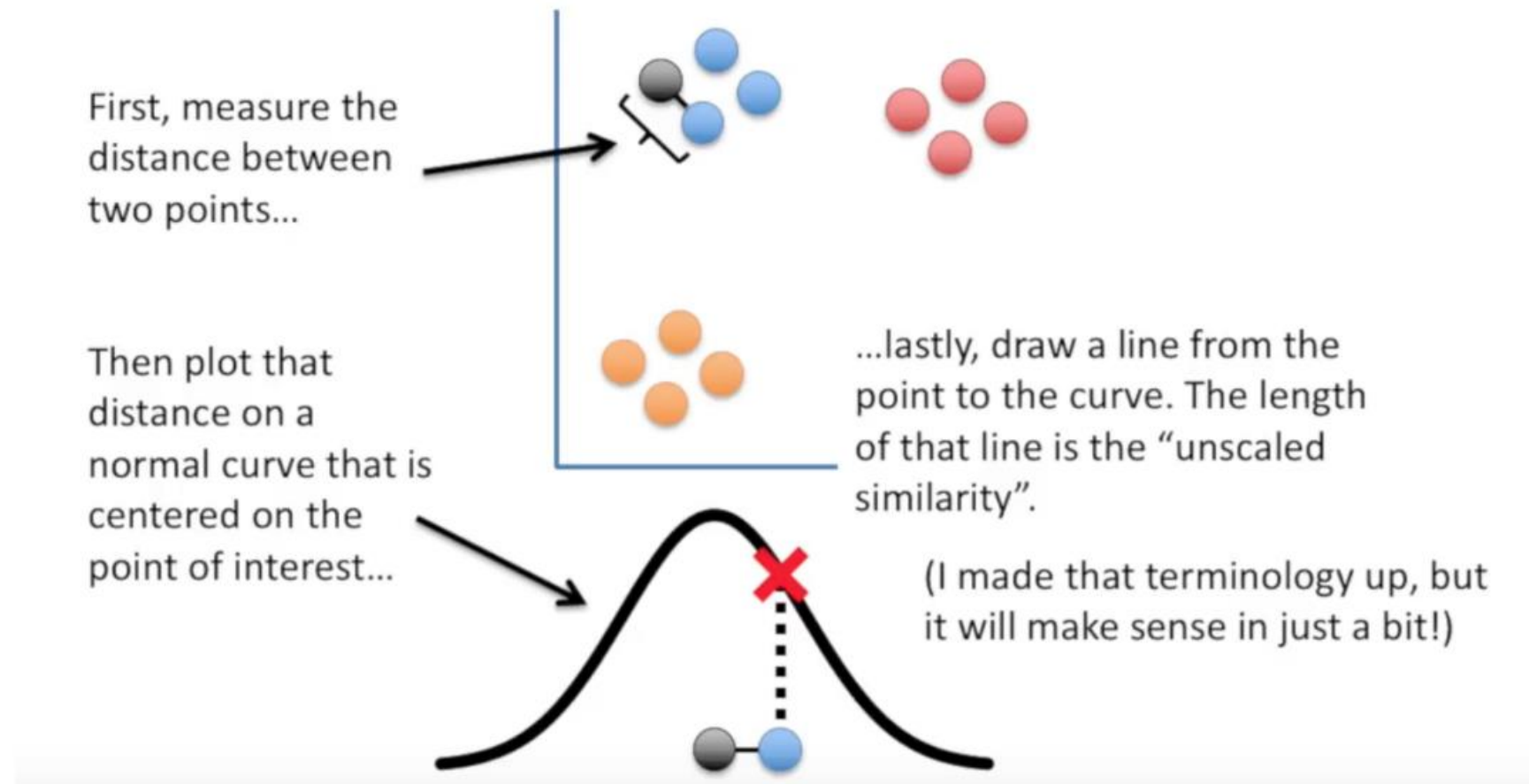


# 3.7 t-SNE

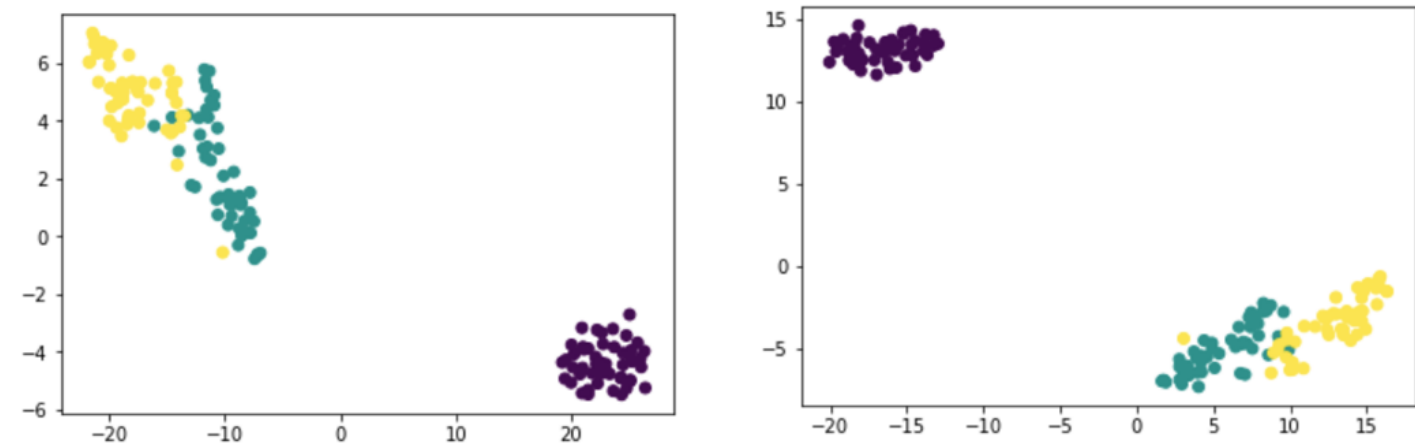
## # t-SNE

- 비선형 관계를 이용한 데이터셋 분해 방법
- 차원이 감소되어 군집화된 데이터들이 뭉개져 제대로 구별할 수 없는 PCA의 문제점 해결
- 고차원 공간에서 점 세트를 가져와 저차원 공간에서 해당 점의 표현을 찾는 것이 목적
- 정규분포 대신 t분포를 이용한다.
- 탁월한 성능을 가졌지만 많은 시간을 소요한다는 단점 존재

# 3.7 t-SNE



1. 점을 하나 선택하여 이 점에서 다른 점까지의 거리를 측정한다.
2. T 분포 그래프를 이용하여 기준점이 T 분포 상 가운데 위치한다면 이로부터 상대점까지의 거리에 있는 T 분포 값을 선택하여 이 값을 친밀도로 한다.
3. 이 친밀도가 가까운 값끼리 묶는다.



-> 군집이 중복되지 않고 매번 계산할 때마다 축의 위치가 바뀌어 다른 모양으로 나타난다.  
데이터의 군집성은 그대로 유지되어 시각화를 통한 데이터 분석에 유용하다.  
하지만 값 자체는 계속 변화되어 머신러닝 모델의 학습 피처로 사용하기는 어렵다.

# 3.7 t-SNE

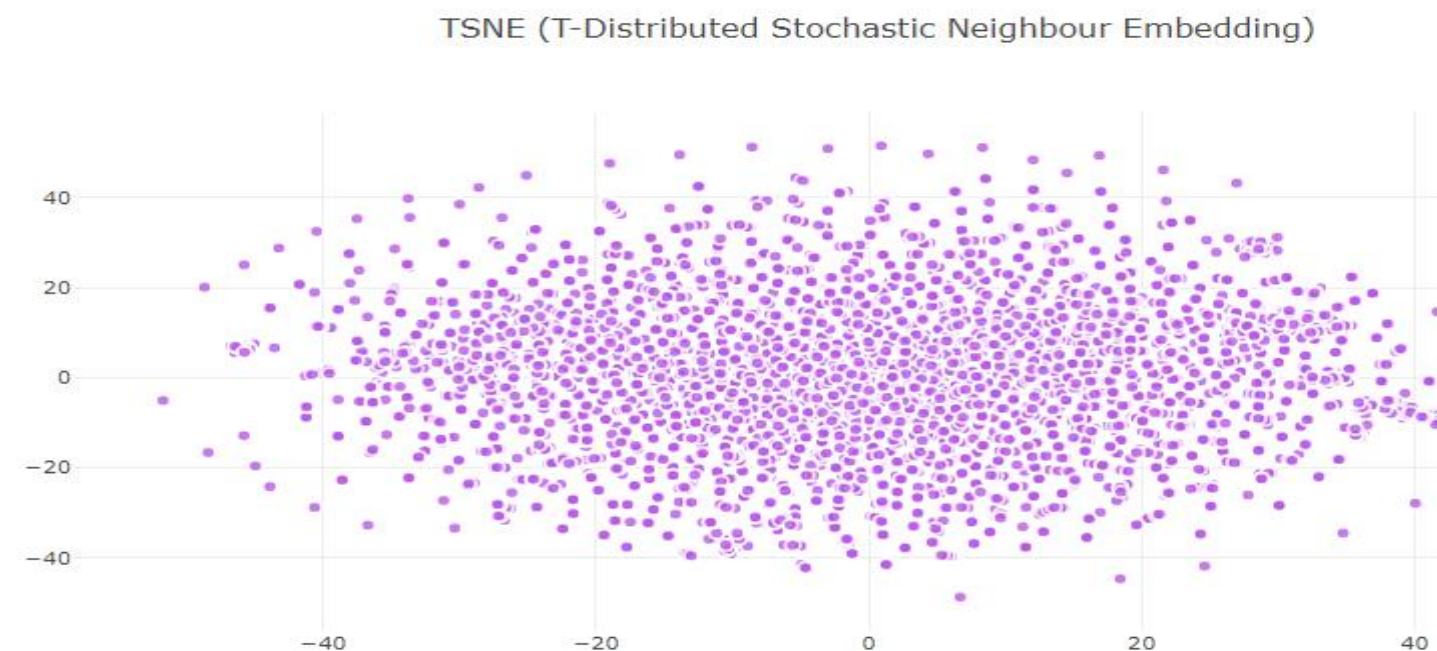
```
tsne_model = TSNE(n_components=2, verbose=1, random_state=42, n_iter=500)
tsne_results = tsne_model.fit_transform(X_svd)

traceTSNE = go.Scatter(
    x = tsne_results[:,0],
    y = tsne_results[:,1],
    name = target,
    hoveron = target,
    mode = 'markers',
    text = target,
    showlegend = True,
    marker = dict(
        size = 8,
        color = '#c94ff2',
        showscale = False,
        line = dict(
            width = 2,
            color = 'rgb(255, 255, 255)'
        ),
        opacity = 0.8
    )
)
data = [traceTSNE]

layout = dict(title = 'TSNE (T-Distributed Stochastic Neighbour Embedding)',
    hovermode = 'closest',
    yaxis = dict(zeroline = False),
    xaxis = dict(zeroline = False),
    showlegend = False,
)

fig = dict(data=data, layout=layout)
iplot(fig)
```

```
[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 4459 samples in 0.343s...
[t-SNE] Computed neighbors for 4459 samples in 55.276s...
[t-SNE] Computed conditional probabilities for sample 1000 / 4459
[t-SNE] Computed conditional probabilities for sample 2000 / 4459
[t-SNE] Computed conditional probabilities for sample 3000 / 4459
[t-SNE] Computed conditional probabilities for sample 4000 / 4459
[t-SNE] Computed conditional probabilities for sample 4459 / 4459
[t-SNE] Mean sigma: 0.192247
[t-SNE] KL divergence after 250 iterations with early exaggeration: 85.749001
[t-SNE] KL divergence after 500 iterations: 2.605067
```



-> 거리를 보존했으므로 원본에서 데이터 간 거리를 추측할 수 있다.

# THANK YOU

