

# Machine Learning with Python

Unsupervised Learning

# Contacts

Haesun Park

Email : [haesunrpark@gmail.com](mailto:haesunrpark@gmail.com)

Meetup: <https://www.meetup.com/Hongdae-Machine-Learning-Study/>

Facebook : <https://facebook.com/haesunrpark>

Blog : <https://tensorflow.blog>

# Book

파이썬 라이브러리를 활용한 머신러닝, 박해선.

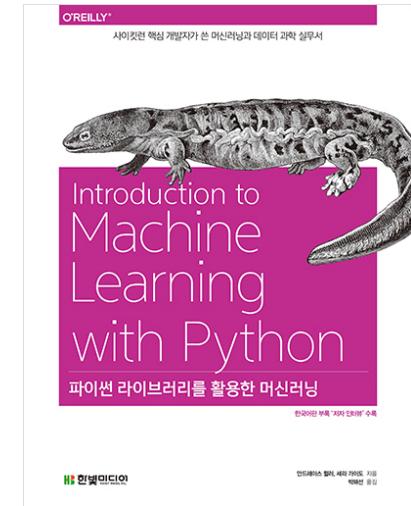
(Introduction to Machine Learning with Python, Andreas Muller & Sarah Guido의 번역서입니다.)

번역서의 1장과 2장은 블로그에서 무료로 읽을 수 있습니다.

원서에 대한 프리뷰를 온라인에서 볼 수 있습니다.

Github:

[https://github.com/rickiepark/introduction\\_to\\_ml\\_with\\_python/](https://github.com/rickiepark/introduction_to_ml_with_python/)



# 비지도 학습

# 종류

비지도 변환 unsupervised transformation

사람이나 머신러닝 알고리즘을 위해 새로운 데이터 표현을 만듦

차원축소: PCA, NMF, t-SNE, 오토인코더 *autoencoder*

토픽 모델링: LDA (e.g. 소셜 미디어 토론 그룹핑)

군집 clustering

비슷한 데이터 샘플을 모으는 것 (e.g. 사진 어플리케이션)

k-평균, 병합, DBSCAN

데이터 스케일링 scaling

이상치 탐지 *anomaly detection*, GAN *generative adversarial networks*

# 비지도 학습의 어려움

레이블이 없기 때문에 올바르게 학습되었는지 알고리즘을 평가하기가 어려움

사진 애플리케이션에서 옆모습, 앞모습으로 군집되었다 해도 직접 눈으로 확인하기 전에는 결과를 평가하기가 어려움

비지도 학습은 데이터를 잘 이해하기 위해서(탐색적 분석) 사용되거나,

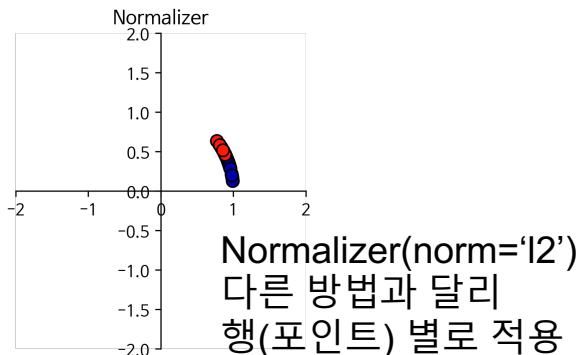
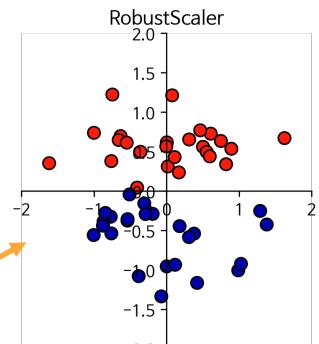
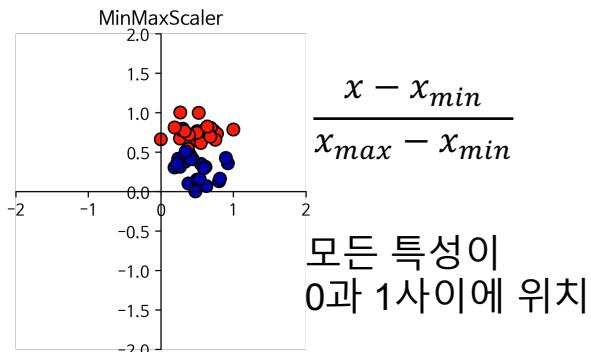
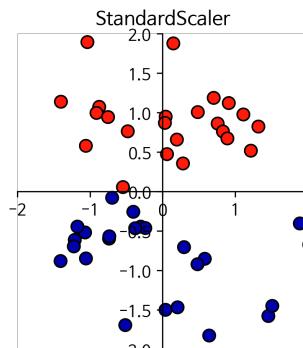
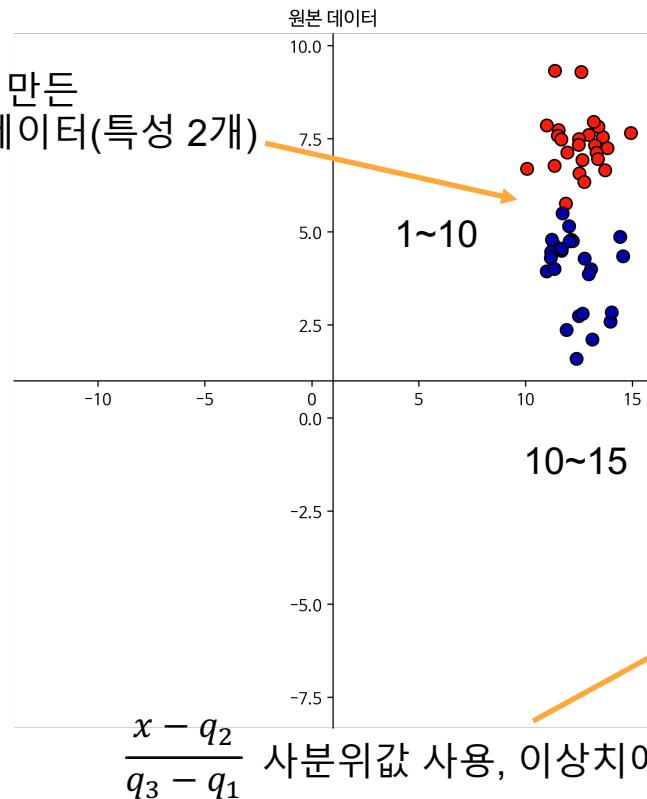
지도학습 알고리즘의 정확도 향상과 메모리/시간 절약을 위해 전처리 단계로 활용

# 데이터 스케일 조정

# 네 가지 스케일링

$$\frac{x - \bar{x}}{\sigma} \text{ 표준점수, z-점수: 평균 } 0, \text{ 분산 } 1$$

인위적으로 만든  
이진 분류 데이터(특성 2개)



# cancer + MinMaxScaler

```
In [4]: from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    random_state=1)
print(X_train.shape)
print(X_test.shape)

(426, 30)
(143, 30)
```

```
In [5]: from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
```

y\_train은 사용하지 않음

```
In [6]: scaler.fit(X_train)
```

```
Out[6]: MinMaxScaler(copy=True, feature_range=(0, 1))
```

# MinMaxScaler.transform

fit → transform

In [7]:

```
# 데이터 변환
X_train_scaled = scaler.transform(X_train)
# 스케일이 조정된 후 데이터셋의 속성을 출력합니다
print("변환된 후 크기: {}".format(X_train_scaled.shape))
print("스케일 조정 전 특성별 최소값:\n {}".format(X_train.min(axis=0)))
print("스케일 조정 전 특성별 최대값:\n {}".format(X_train.max(axis=0)))
print("스케일 조정 후 특성별 최소값:\n {}".format(X_train_scaled.min(axis=0)))
print("스케일 조정 후 특성별 최대값:\n {}".format(X_train_scaled.max(axis=0)))
```

변환된 후 크기: (426, 30)

스케일 조정 전 특성별 최소값:

```
[ 6.981    9.71   43.79   143.5     0.053    0.019     0.      0.
  0.106    0.05   0.115    0.36     0.757    6.802     0.002    0.002
  0.        0.      0.01    0.001    7.93     12.02    50.41    185.2
  0.071    0.027   0.       0.      0.157    0.055]
```

스케일 조정 전 특성별 최대값:

```
[ 28.11    39.28   188.5   2501.     0.163    0.287    0.427
  0.201    0.304    0.096    2.873    4.885    21.98    542.2
  0.031    0.135    0.396    0.053    0.061    0.03     36.04
  49.54    251.2   4254.    0.223    0.938    1.17     0.291
  0.577    0.1491]
```

스케일 조정 후 특성별 최소값:

```
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

스케일 조정 후 특성별 최대값:

```
[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

numpy 배열의  
min, max 메서드

최소와 최댓값이  
0, 1로 바뀜

# transform(X\_test)

$$\frac{x_{test} - x_{train\_min}}{x_{train\_max} - x_{train\_min}}$$

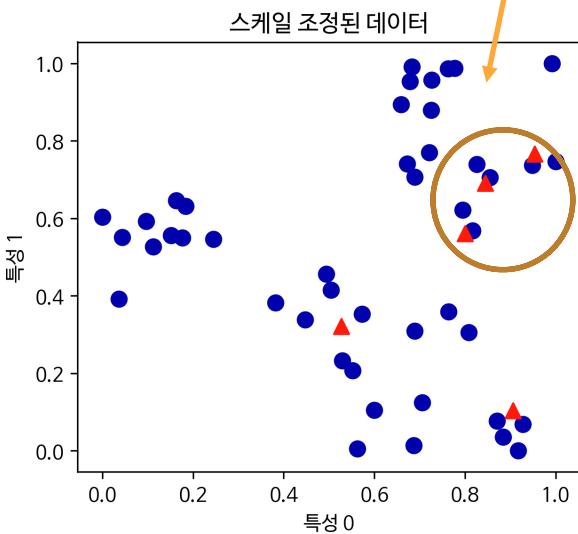
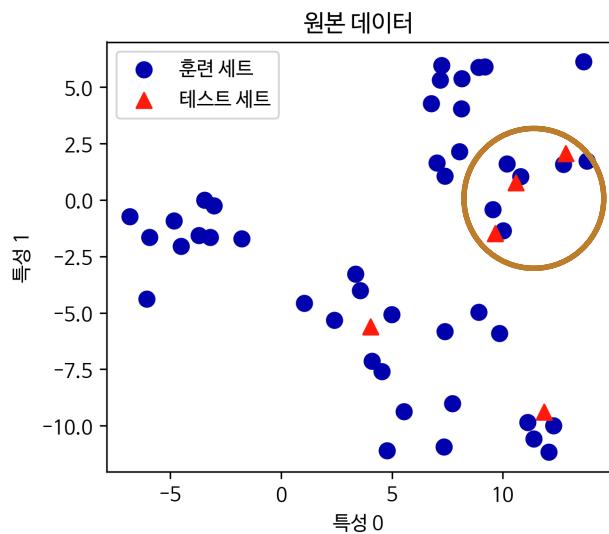
훈련 세트로 학습시킨 scaler를 사용해 X\_test를 변환

```
In [8]: # 테스트 데이터 변환  
X_test_scaled = scaler.transform(X_test)  
# 스케일이 조정된 후 테스트 데이터의 속성을 출력합니다  
print("스케일 조정 후 특성별 최소값:\n{}".format(X_test_scaled.min(axis=0)))  
print("스케일 조정 후 특성별 최대값:\n{}".format(X_test_scaled.max(axis=0)))
```

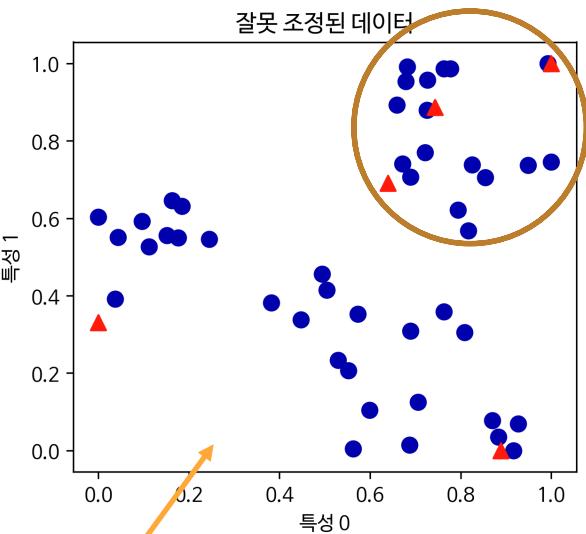
스케일 조정 후 특성별 최소값:  
[ 0.034 0.023 0.031 0.011 0.141 0.044 0. 0. 0.154 -0.006  
 -0.001 0.006 0.004 0.001 0.039 0.011 0. 0. -0.032 0.007  
 0.027 0.058 0.02 0.009 0.109 0.026 0. 0. -0. -0.002 ]  
스케일 조정 후 특성별 최대값:  
[ 0.958 0.815 0.956 0.894 0.811 1.22 0.88 0.933 0.932 1.037  
 0.427 0.498 0.441 0.284 0.487 0.739 0.767 0.629 1.337 0.391  
 0.896 0.793 0.849 0.745 0.915 1.132 1.07 0.924 1.205 1.631 ]

테스트 세트의 최소, 최댓값이 0~1 사이를 벗어남

# train과 test의 스케일 조정



```
scaler = MinMaxScaler()  
scaler.fit(X_train)  
X_train_scaled = scaler.transform(X_train)  
X_test_scaled = scaler.transform(X_test)
```



```
test_scaler = MinMaxScaler()  
test_scaler.fit(X_test)  
X_test_scaled_badly = test_scaler.transform(X_test)
```

# <단축 메서드>

```
In [10]: from sklearn.preprocessing import StandardScaler  
scaler = StandardScaler()  
# 메소드 체이닝(chaining)을 사용하여 fit과 transform을 연달아 호출합니다  
X_scaled = scaler.fit(X_train).transform(X_train)  
# 위와 동일하지만 더 효율적입니다  
X_scaled_d = scaler.fit_transform(X_train)
```

transform 메서드를 가진 변환기 클래스는 모두 fit\_transform 메서드를 가지고 있음

TransformerMixin 클래스를 상속하는 대부분의 경우 fit\_transform은 단순히  
scaler.fit().transform()처럼 연이어 호출

일부의 경우 fit\_transform 메서드가 효율적인 경우가 있음(PCA)

# scaler + SVC

```
In [11]: from sklearn.svm import SVC

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    random_state=0)

svm = SVC(C=100)
svm.fit(X_train, y_train)
print("테스트 세트 정확도: {:.2f}".format(svm.score(X_test, y_test)))
```

테스트 세트 정확도: 0.63

```
In [12]: # 0~1 사이로 스케일 조정
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 조정된 데이터로 SVM 학습
svm.fit(X_train_scaled, y_train)

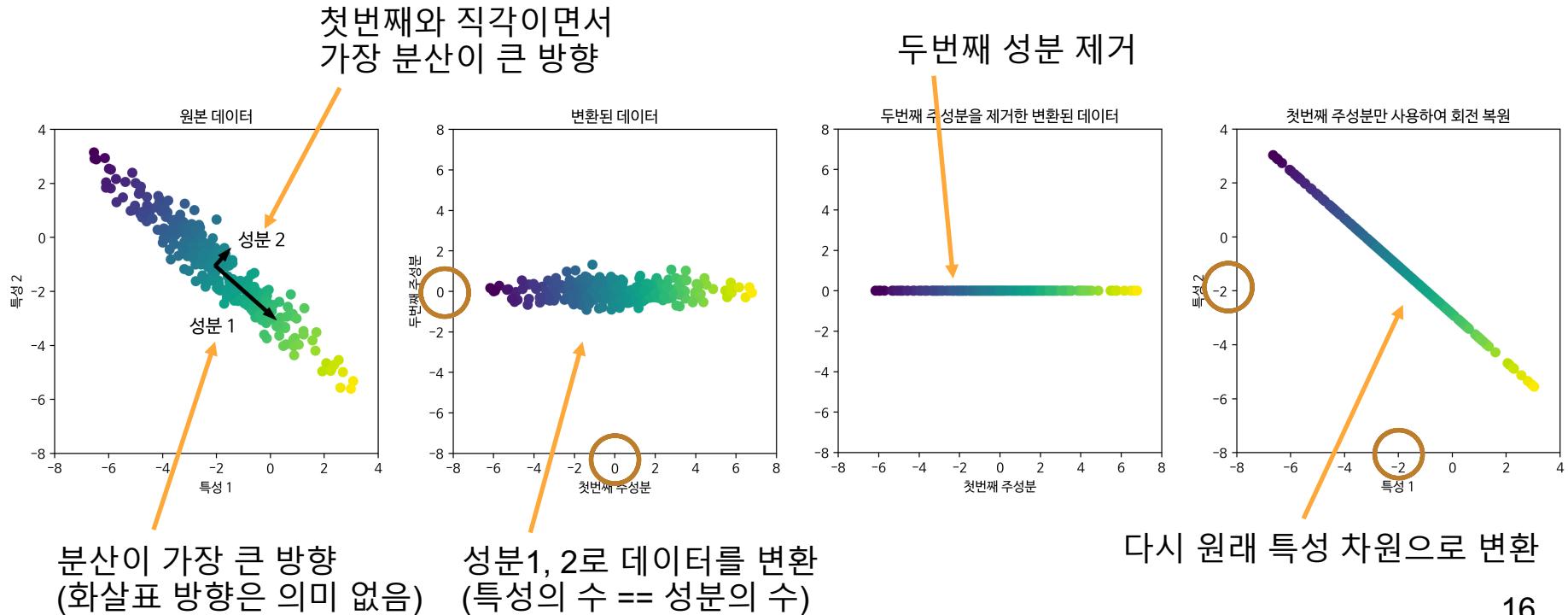
# 스케일 조정된 테스트 세트의 정확도
print("스케일 조정된 테스트 세트의 정확도: {:.2f}".format(svm.score(X_test_scaled, y_test)))
```

스케일 조정된 테스트 세트의 정확도: 0.97

# PCA

# 주성분 분석 Principal component analysis

상관관계가 없는 특성으로 데이터셋을 변환(회전)하고 일부 특성을 선택하는 기술



# PCA와 특이값분해

행렬  $X$ 의 공분산

$$\text{Cov}(X_i, X_j) = \frac{1}{n-1} [(X_i - \bar{X}_i)(X_j - \bar{X}_j)]$$

평균을 뺀 행렬로 표현하면

$$\text{Cov}(X, X) = \frac{1}{n-1} X^T X$$

특이값 분해(SVD)  $X = USV^T$ 로 부터

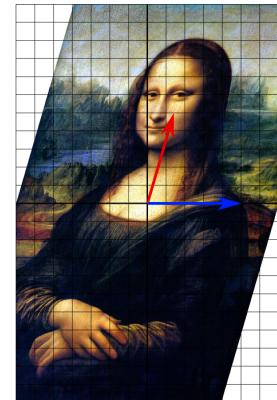
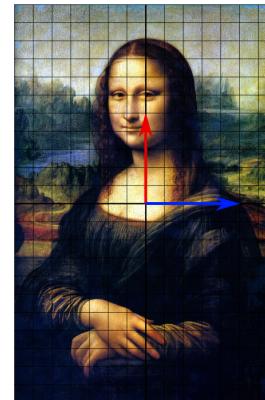
$$\text{Cov}(X, X) = \frac{1}{n-1} X^T X = \frac{1}{n-1} (VSU^T)USV^T = V \frac{S^2}{n-1} V^T$$

대칭행렬의 고유벡터인  $V$ 가 주성분의 방향

$$X_{pca} = XV = USV^T V = US$$

$$\begin{bmatrix} V_1 & C_{1,2} & C_{1,3} \\ C_{1,2} & V_2 & C_{2,3} \\ C_{1,3} & C_{2,3} & V_3 \end{bmatrix}$$

행렬의 고유벡터 →  
크기만 바뀌고 방향은 바뀌지 않음  
 $y = Ax = \lambda x$



공분산 행렬의 고유벡터 →  
분산의 주 방향

# PCA + cancer

특성의 스케일을 맞추기 위해  
분산이 1이 되도록 변경



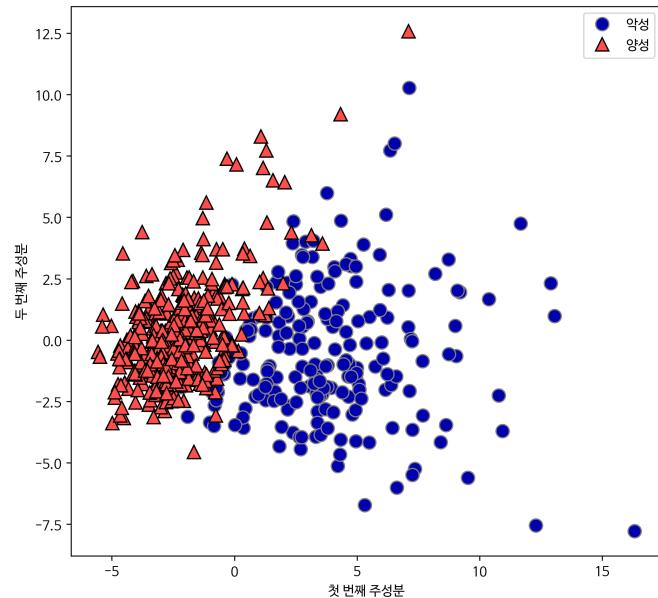
```
scaler = StandardScaler()  
scaler.fit(cancer.data)  
X_scaled = scaler.transform(cancer.data)
```

In [17]:

```
from sklearn.decomposition import PCA  
# 데이터의 처음 두 개 주성분만 유지시킵니다  
pca = PCA(n_components=2)  
# 유방암 데이터로 PCA 모델을 만듭니다  
pca.fit(X_scaled)  
  
# 처음 두 개의 주성분을 사용해 데이터를 변환합니다  
X_pca = pca.transform(X_scaled)  
print("원본 데이터 형태: {}".format(str(X_scaled.shape)))  
print("축소된 데이터 형태: {}".format(str(X_pca.shape)))
```

원본 데이터 형태: (569, 30)

축소된 데이터 형태: (569, 2)



# pca.components\_

components\_ 속성에 주성분 방향 principal axis  $V^T$ 가 저장되어 있음

components\_ 크기 (n\_components, n\_features)

```
In [17]: from sklearn.decomposition import PCA  
# 데이터의 처음 두 개 주성분만 유지시킵니다  
pca = PCA(n_components=2)  
# 유방암 데이터로 PCA 모델을 만듭니다  
pca.fit(X_scaled)
```

$$\begin{aligned} X &= USV^T \\ X_{pca} &= XV = US \\ &[n\_samples, n\_features] \cdot [n\_features, n\_components] \\ &= [n\_samples, n\_components] \end{aligned}$$

```
In [19]: print("PCA 주성분 형태: {}".format(pca.components_.shape))
```

PCA 주성분 형태: (2, 30)

```
In [20]: print("PCA 주성분: {}".format(pca.components_))
```

```
PCA 주성분: [[ 0.219  0.104  0.228  0.221  0.143  0.239  0.258  0.261  0.138  0.064  
              0.206  0.017  0.211  0.203  0.015  0.17   0.154  0.183  0.042  0.103  
              0.228  0.104  0.237  0.225  0.128  0.21   0.229  0.251  0.123  0.132]  
             [-0.234 -0.06  -0.215 -0.231  0.186  0.152  0.06  -0.035  0.19   0.367  
              -0.106  0.09  -0.089 -0.152  0.204  0.233  0.197  0.13   0.184  0.28  
              -0.22  -0.045 -0.2   -0.219  0.172  0.144  0.098 -0.008  0.142  0.275]]
```

주성분 방향을 직관적으로  
설명하기 어려움

# LFW dataset

메사추세츠 애머스트 주립대의 비전랩에서 만든 LFW Labeled Faces in the Wild 데이터셋

인터넷에서 모은 2000년대 초반 유명 인사의 얼굴 이미지

```
In [22]: from sklearn.datasets import fetch_lfw_people  
people = fetch_lfw_people(min_faces_per_person=20, resize=0.7)
```



총 62명의 얼굴  
3,023개 → 2,063개 이미지  
각 이미지는 87x65 사이즈  
→ 5,655 개의 특성

# k-NN + Ifw

(지도 학습) 1-최근접 이웃을 적용

87x65 픽셀 값의 거리를 계산하므로 위치에 매우 민감해 짐

```
In [27]: from sklearn.neighbors import KNeighborsClassifier
# 데이터를 훈련 세트와 테스트 세트로 나눕니다
X_train, X_test, y_train, y_test = train_test_split(
    X_people, y_people, stratify=y_people, random_state=0)
# 이웃 개수를 한 개로 하여 KNeighborsClassifier 모델을 만듭니다
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
print("1-최근접 이웃의 테스트 세트 점수: {:.2f}".format(knn.score(X_test, y_test)))
```

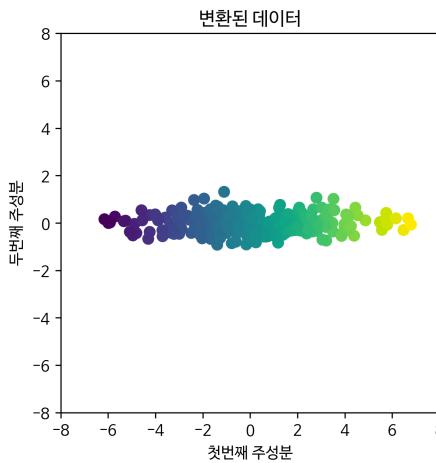
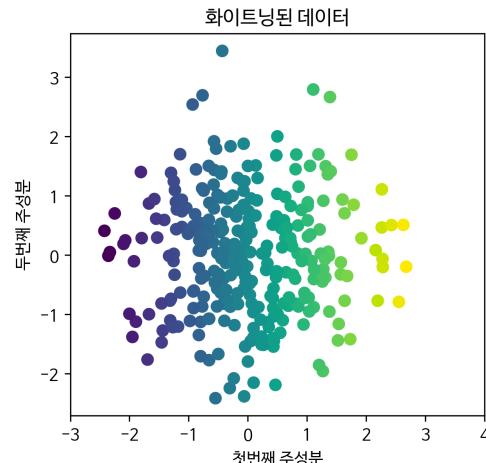
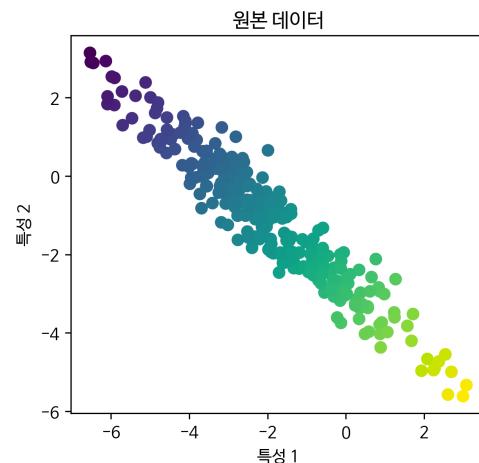
1-최근접 이웃의 테스트 세트 점수: 0.23

무작위로 분류할 경우  $1/62=1.6\%$

# 화이트닝 whitening

백색소음에서 이름이 유래됨, 특성 간의 (대각행렬을 제외한) 공분산이 모두 0이 되고(PCA) 특성의 분산이 1로 되는 변환

```
In [29]: pca = PCA(n_components=100, whiten=True, random_state=0).fit(X_train)
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)
```



# k-NN + whitening

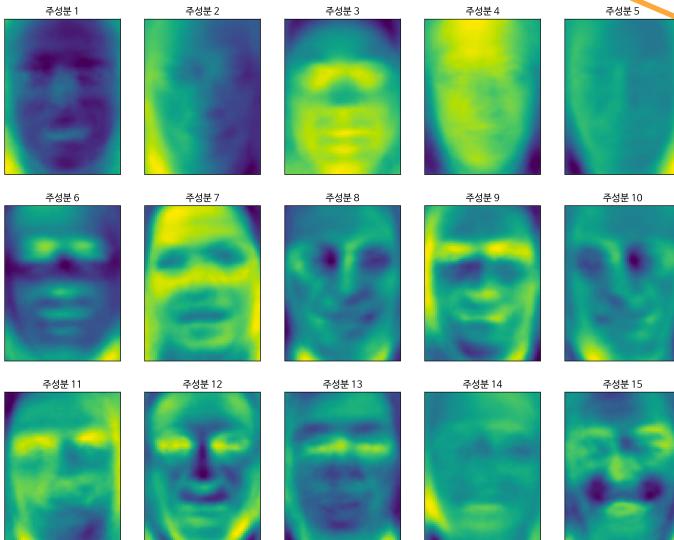
```
In [30]: knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train_pca, y_train)
print("테스트 세트 정확도: {:.2f}".format(knn.score(X_test_pca, y_test)))
```

테스트 세트 정확도: 0.31

주성분의 갯수

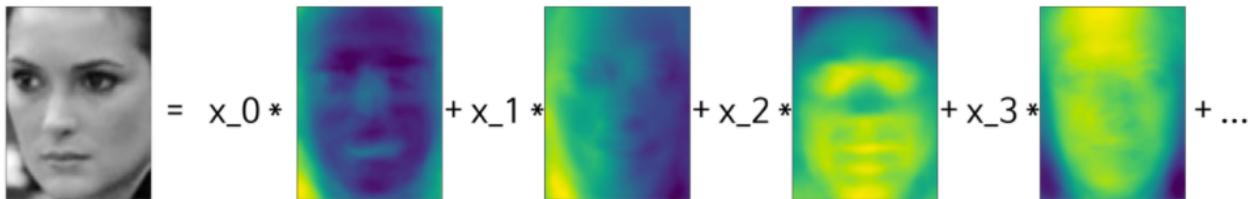
```
In [31]: print("pca.components_.shape: {}".format(pca.components_.shape))
```

pca.components\_.shape: (100, 5655)



픽셀수==특성의수==주성분의 벡터 방향

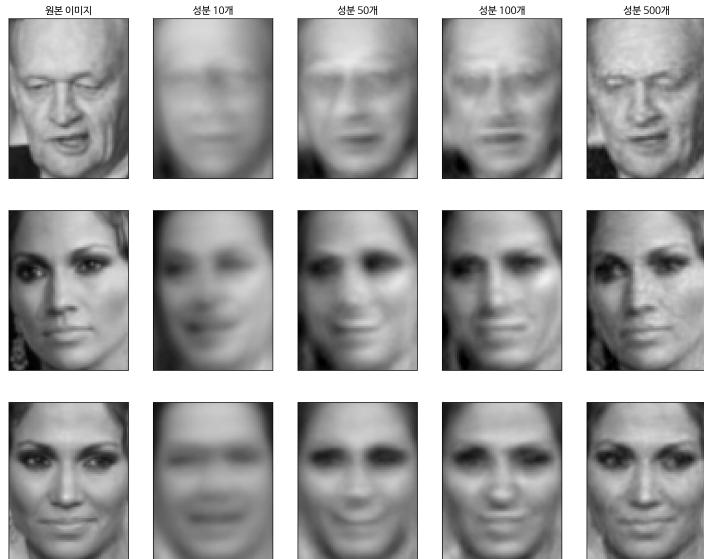
# PCA as weighted sum



주성분 방향으로 변환한 데이터( $x$ )에 주성분 방향을 곱하여 원본 샘플 복원

$$[1 \times 100] \cdot [100 \times 5655] = [1 \times 5655]$$

원본 샘플을 주성분의 가중치 합으로 표현



# 비음수 행렬 분해(NMF)

# NMF

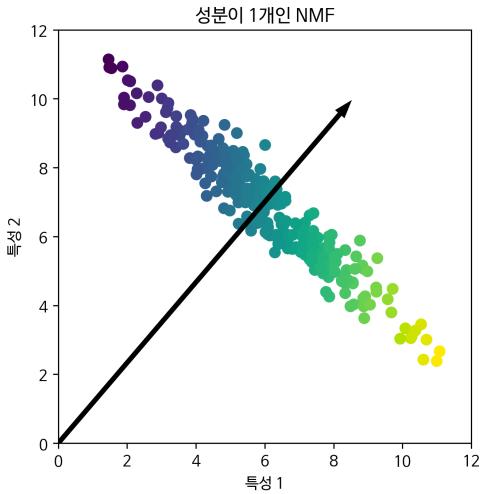
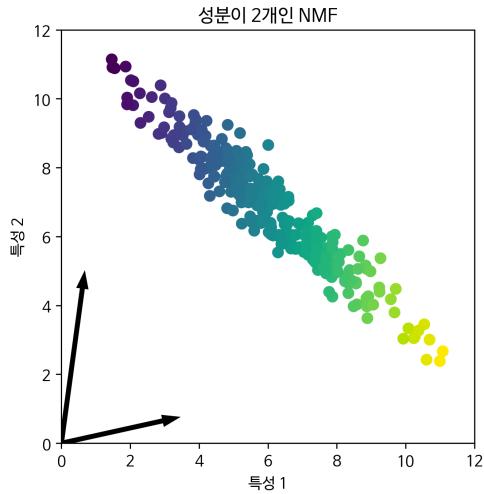
PCA와 비슷하고 차원 축소에 사용할 수 있음

원본 = 뽑아낸 성분의 가중치의 합

음수가 아닌 성분과 계수를 만드므로 양수로 이루어진 데이터에 적용 가능

예) 여러 악기나 목소리가 섞인 오디오 트랙

# NMF + 인공 데이터셋

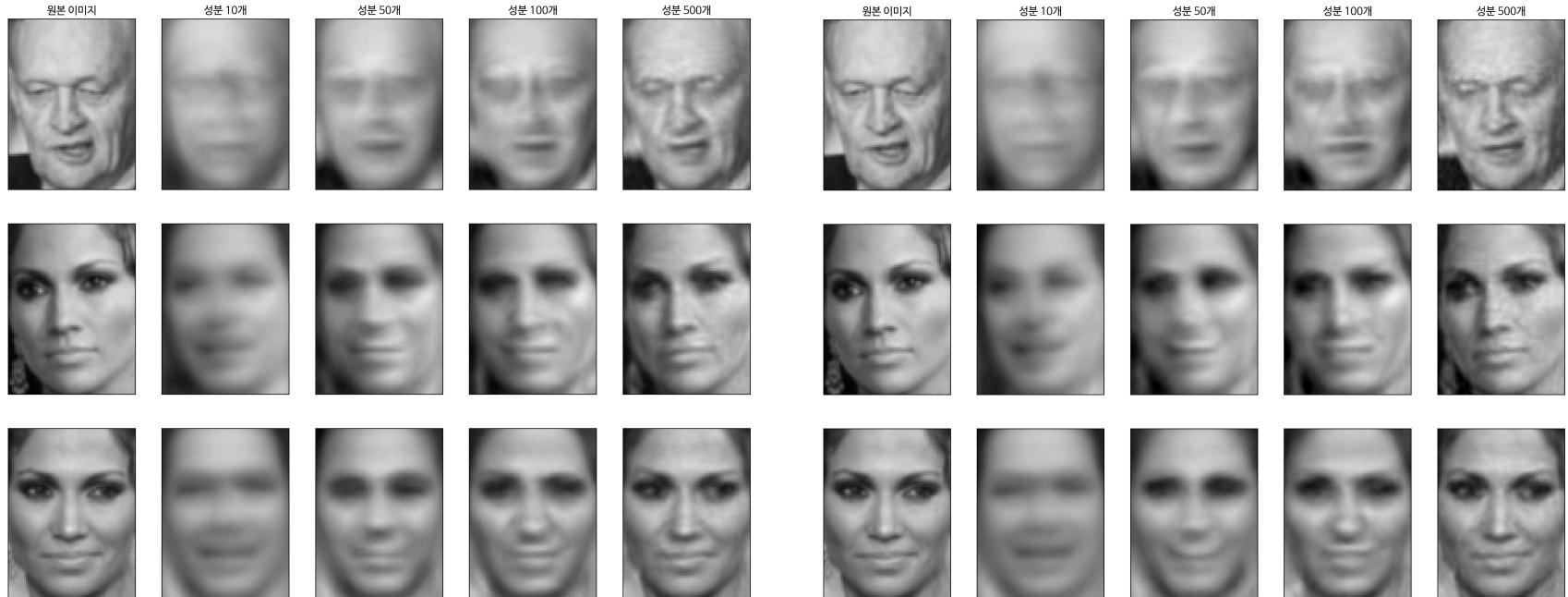


성분이 특성 개수 만큼 많다면 데이터의 끝 부분을 가리킴

하나일 경우에는 평균 방향을 가리킴

성분의 개수를 변경하면 전체 성분이 모두 바뀌고 성분의 순서가 없음

# NMF + Ifw



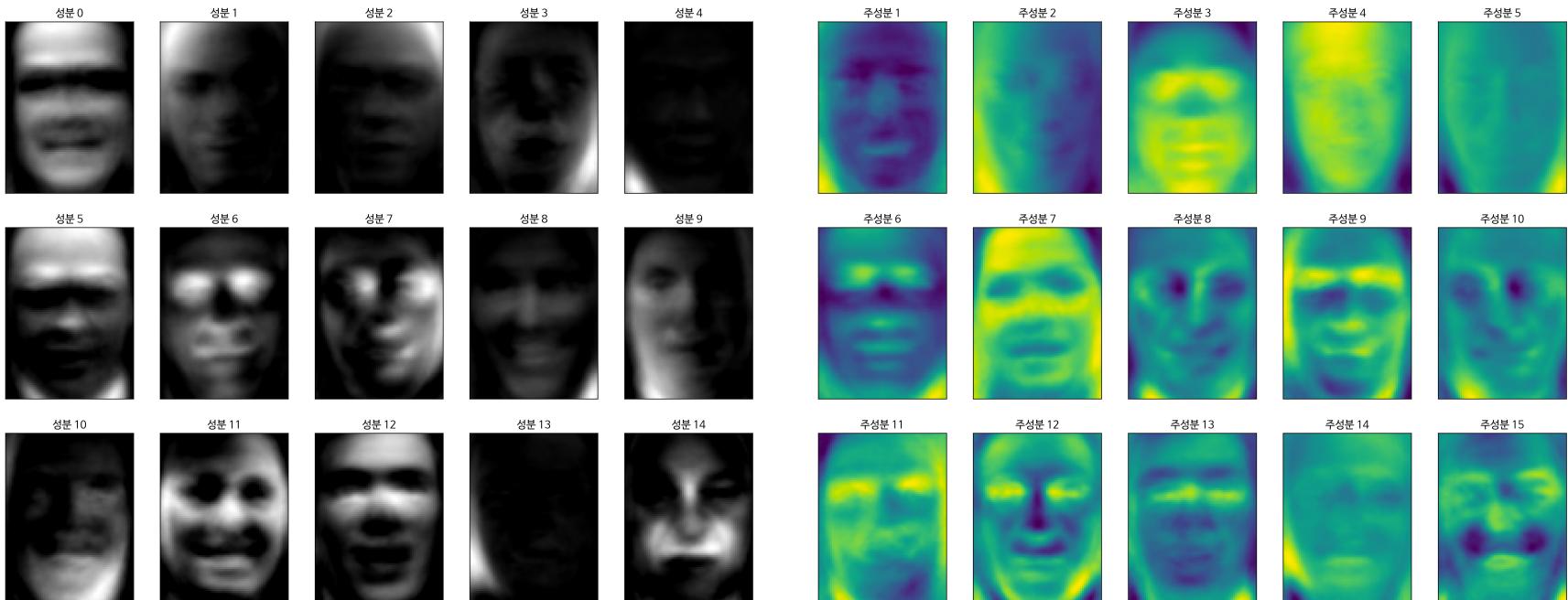
NMF

<

PCA

NMF는 주로 데이터에 있는 패턴을 분할하는데 주로 사용함

# NMF - Ifw의 성분

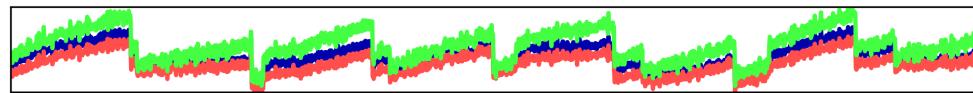
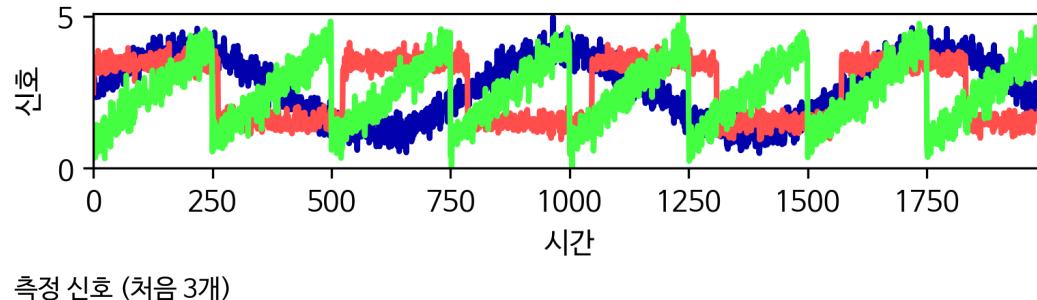


NMF

PCA

성분이 모두 양수이므로 얼굴 이미지와 가까운 형태를 띤

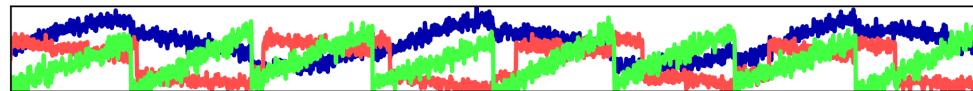
# NMF + signal data



In [42]:

```
nmf = NMF(n_components=3, random_state=42)
S_ = nmf.fit_transform(X)
```

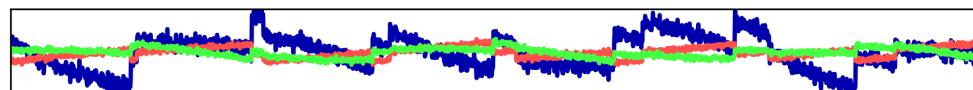
NMF로 복원한 신호



In [43]:

```
pca = PCA(n_components=3)
H = pca.fit_transform(X)
```

PCA로 복원한 신호



t-SNE

# t-SNE t-Distributed Stochastic Neighbor Embedding

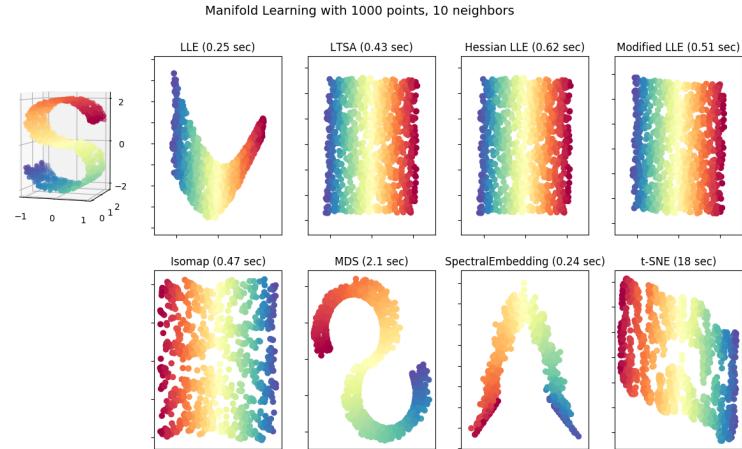
매니폴드 학습 manifold learning 은 비선형 차원 축소의 기법

PCA는 저차원의 평면에 투영하는 효과를 낸다고 말할 수 있음

매니폴드 학습은 시각화에 뛰어난 성능을 발휘함

탐색적 데이터 분석에 사용(no transform 메서드)

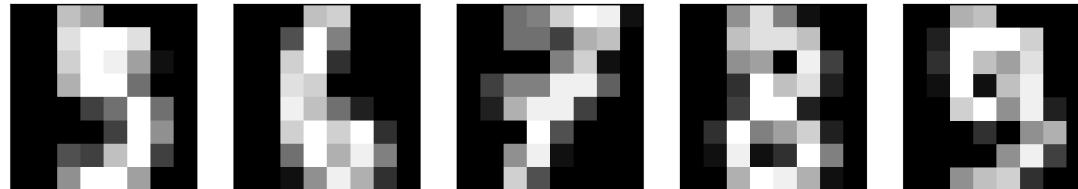
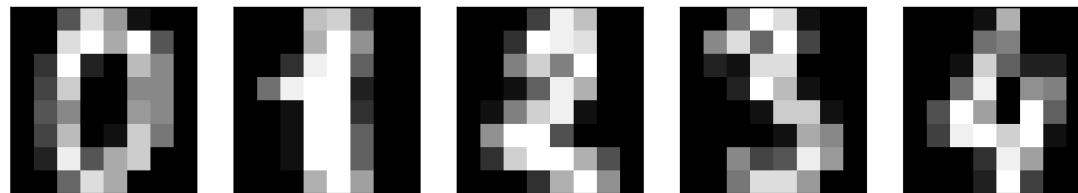
2차원 평면에 데이터를 퍼뜨린 후 원본 특성에서  
의 거리를 유지하도록 학습



# load\_digits

8x8 흑백 이미지의 손글씨 숫자 데이터셋(MNIST 데이터셋 아님)

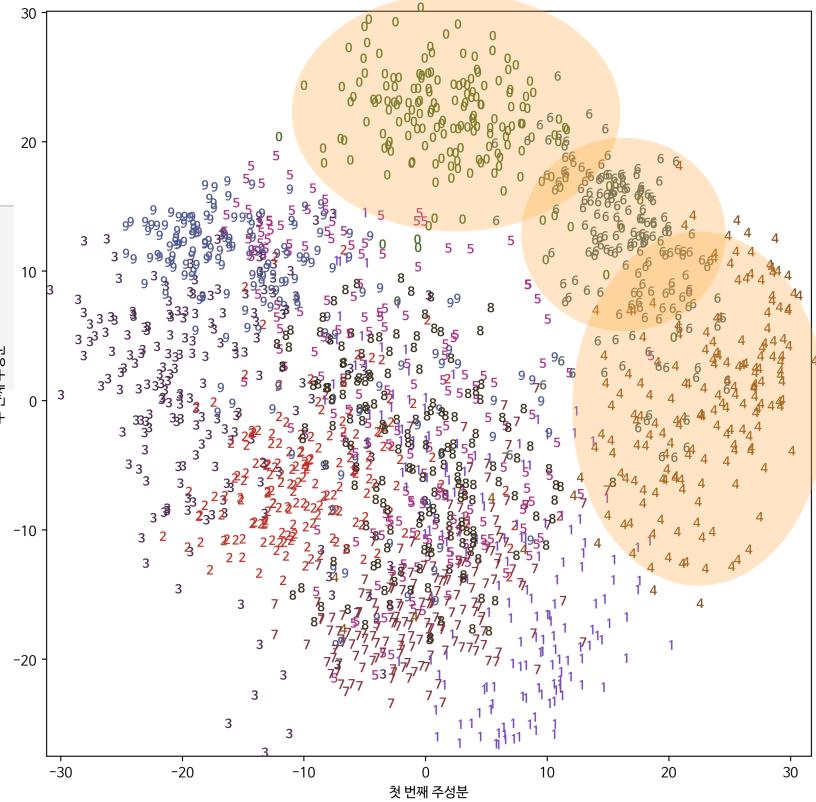
```
In [45]: from sklearn.datasets import load_digits  
digits = load_digits()
```



# PCA + load\_digits

2개의 주성분으로 데이터 변환

```
In [46]: # PCA 모델을 생성합니다  
pca = PCA(n_components=2)  
pca.fit(digits.data)  
# 처음 두 개의 주성분으로 숫자 데이터를 변환합니다  
digits_pca = pca.transform(digits.data)
```



# TSNE + load\_digits

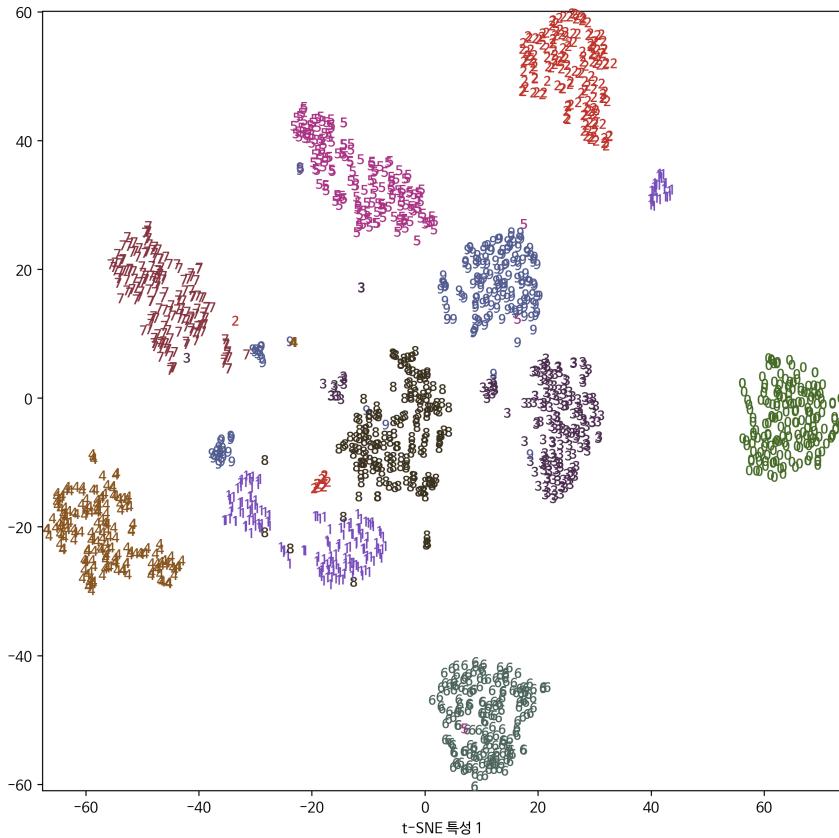
v0.19에서 버그 수정됨

뛰어난 클래스 시각화 능력

perplexity(5~50, default 30)가 크면 이웃을  
많이 포함시킴(큰 데이터셋)

early\_exaggeration(default 1)에서 초기 과장  
단계의 정도를 지정함

```
In [47]: from sklearn.manifold import TSNE
tsne = TSNE(random_state=42)
# TSNE에는 transform 메소드가 없으므로 대신 fit_transform
digits_tsne = tsne.fit_transform(digits.data)
```



k-평균 군집

# 군집

데이터셋을 클러스터<sup>cluster</sup>라는 그룹으로 나누는 작업

한 클러스터 안의 데이터는 매우 비슷하고 다른 클러스터와는 구분되도록 만듬

분류 알고리즘처럼 테스트 데이터에 대해서 어느 클러스터에 속할지 예측을 만들 수 있음

- k-평균 군집
- 병합 군집
- DBSCAN

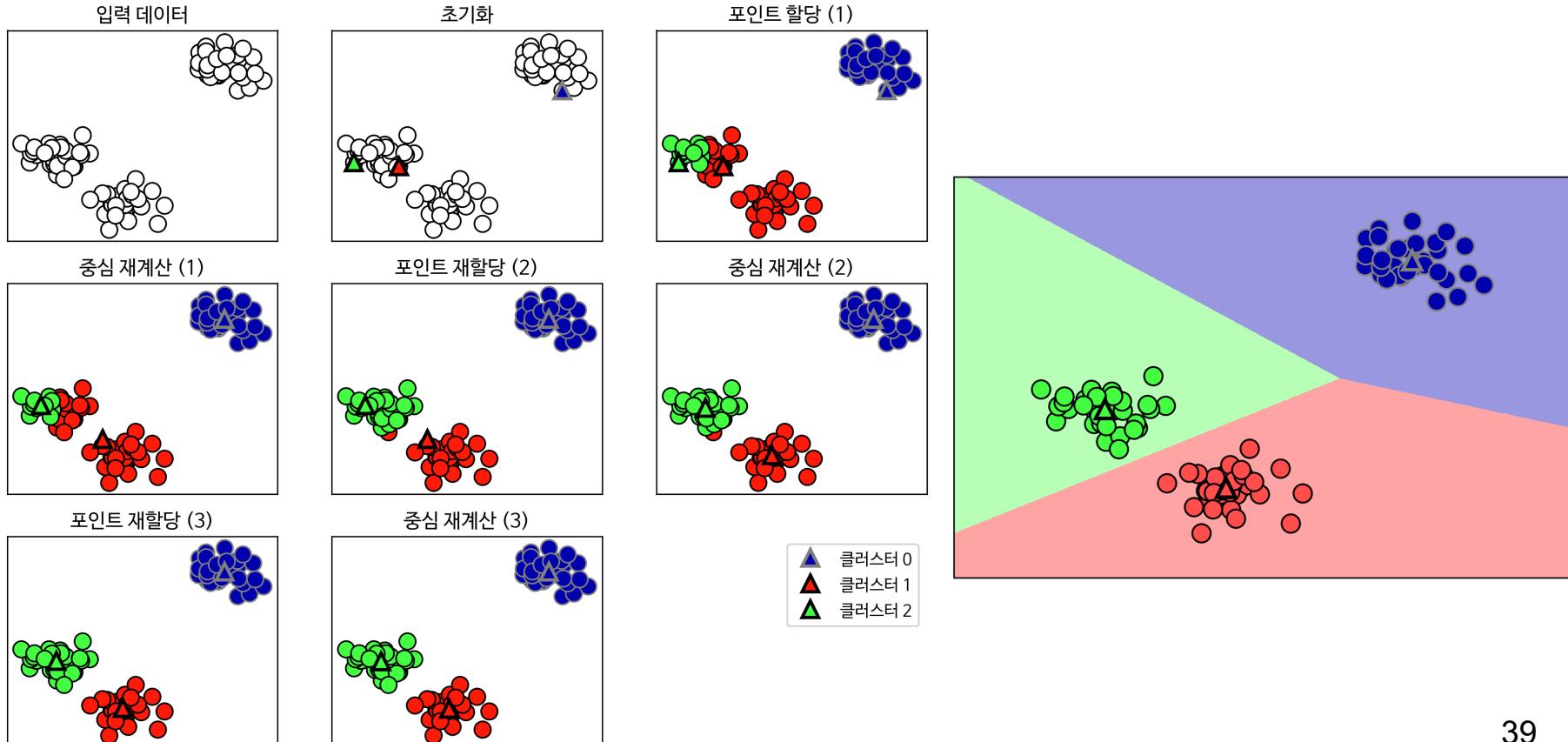
## k-평균means 군집

가장 간단하고 널리 사용됨

임의의 클러스터 중심에 데이터 포인트를 할당하고 평균을 내어 다시 클러스터 중심을 계산함

클러스터에 할당되는 데이터 포인트의 변화가 없으면 알고리즘 종료됨

# k-평균 example



# KMeans

```
In [51]: from sklearn.datasets import make_blobs  
from sklearn.cluster import KMeans
```

```
# 인위적으로 2차원 데이터를 생성합니다  
X, y = make_blobs(random_state=1)
```

```
# 군집 모델을 만듭니다
```

```
kmeans = KMeans(n_clusters=3)  
kmeans.fit(X)
```

레이블에 어떤 의미가 없으며  
순서가 무작위임(분류와 다른점)

```
In [52]: print(kmeans.labels_)
```

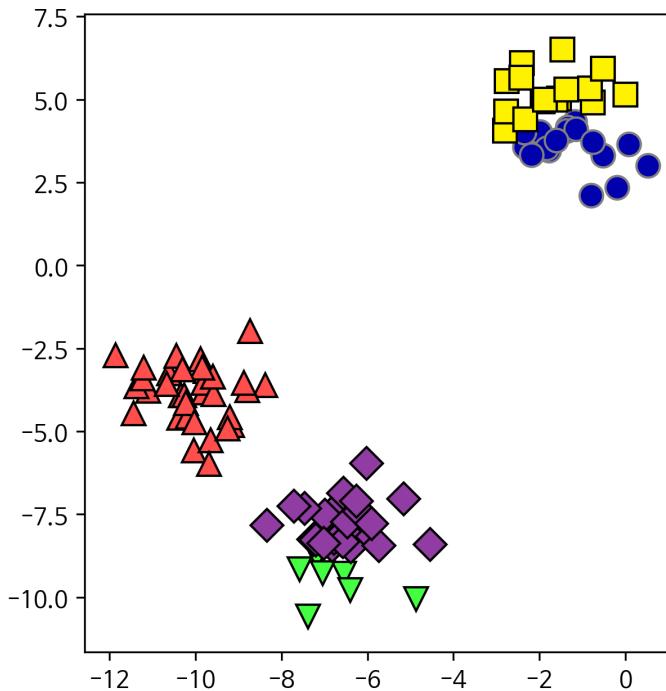
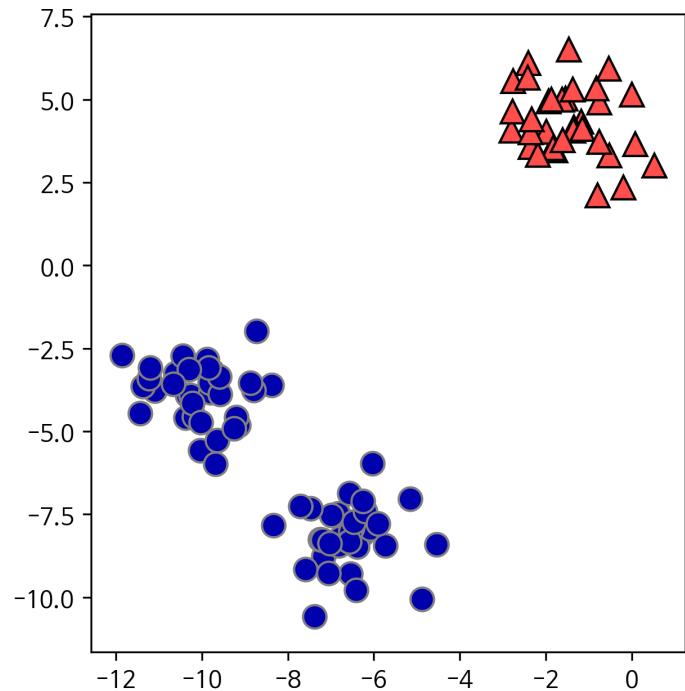
```
[1 2 2 2 0 0 0 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 2 0 0 1 1 0 1 1 0 1 2 0 2  
2 2 0 0 2 1 2 2 0 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 0 0 1 0 1 2 2 2 0 1  
1 2 0 0 1 2 1 2 2 0 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1 0 1 1 0 1 2 0 2 0 0 1 0 1 1 0 1 2 0 2]
```

```
In [53]: print(kmeans.predict(X))
```

```
[1 2 2 2 0 0 0 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 2 0 0 1 1 0 1 1 0 1 1 0 1 2 0 2  
2 2 0 0 2 1 2 2 0 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 0 0 1 0 1 2 2 2 0 1  
1 2 0 0 1 2 1 2 2 0 1 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1 0 1 1 0 1 2 2 2 0 0 1 0 1 2 2 2 0 1]
```

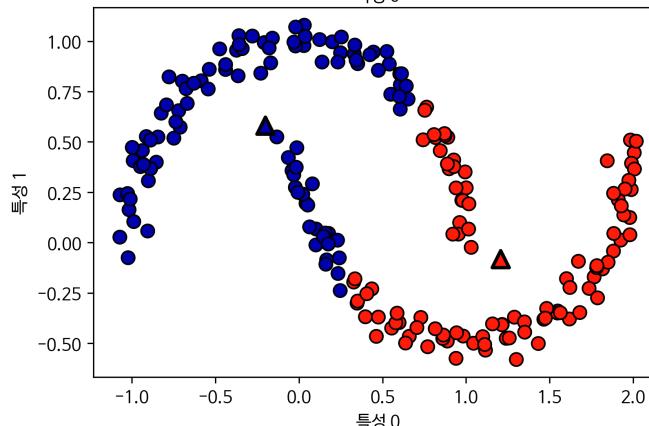
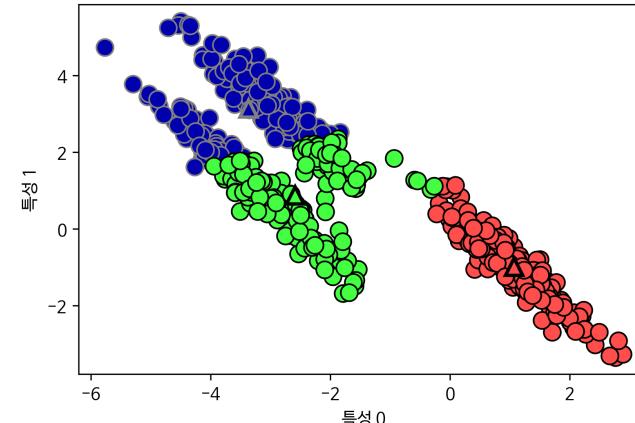
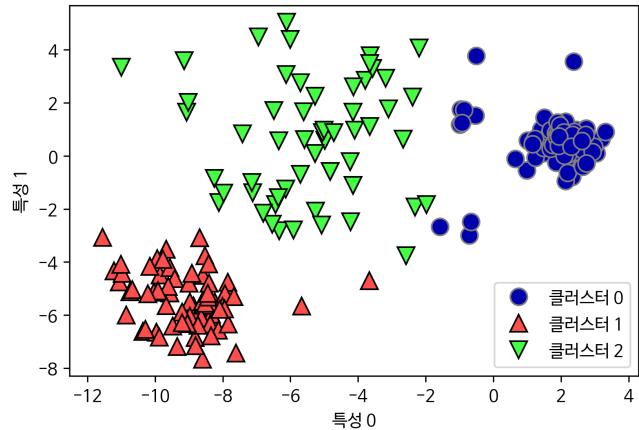
# n\_clusters=2 or 5

n\_clusters 매개변수에 따라 군집이 크게 좌우됨



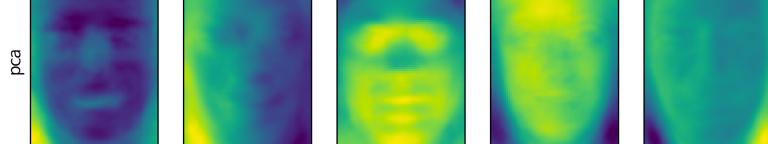
# k-평균의 단점

클러스터가 원형이고  
반경이 동일하다고 간주함



# 벡터 양자화

PCA, NMF가 데이터 포인트를 성분의 합으로 표현할 수 있는 것처럼 k-평균은 데이터 포인트를 하나의 성분으로 나타내는 것으로 볼 수 있음(벡터 양자화)

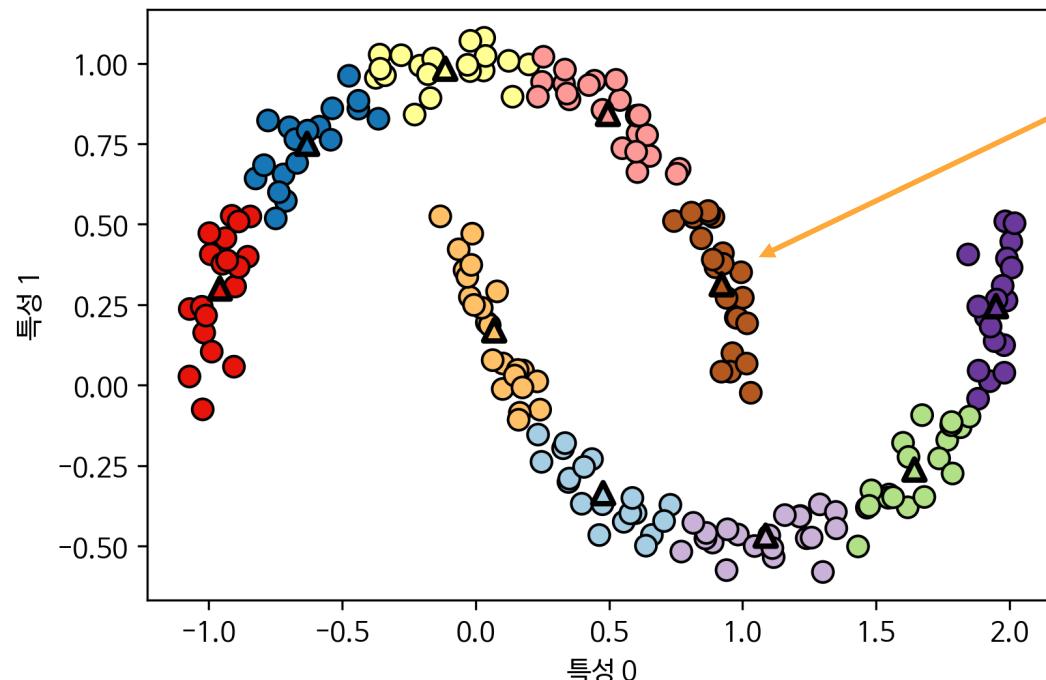


클러스터 평균

# 차원 확대

```
In [61]: X, y = make_moons(n_samples=200, noise=0.05, random_state=0)
```

```
kmeans = KMeans(n_clusters=10, random_state=0)  
kmeans.fit(X)
```



두개의 특성에서  
10개의 특성으로 늘어남  
e.g. [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]

# KMeans.transform()

각 데이터 포인트에서 모든 클러스터까지의 거리를 반환

10개의 새로운 특성으로 활용 가능

```
In [62]: distance_features = kmeans.transform(X)
print("클러스터 거리 데이터의 형태: {}".format(distance_features.shape))
print("클러스터 거리:\n{}".format(distance_features))
```

클러스터 거리 데이터의 형태: (200, 10)

클러스터 거리:

```
[[ 0.922  1.466  1.14 ...,  1.166  1.039  0.233]
 [ 1.142  2.517  0.12 ...,  0.707  2.204  0.983]
 [ 0.788  0.774  1.749 ...,  1.971  0.716  0.944]
 ...,
 [ 0.446  1.106  1.49 ...,  1.791  1.032  0.812]
 [ 1.39   0.798  1.981 ...,  1.978  0.239  1.058]
 [ 1.149  2.454  0.045 ...,  0.572  2.113  0.882]]
```

# 장단점

## 장점

이해와 구현이 쉬움

비교적 빠르고 인기가 높음

대규모 데이터셋에 적용가능(MiniBatchKMeans)

## 단점

무작위로 초기화하므로 난수에 따라 결과가 달라짐

`n_init(default 10)` 매개변수만큼 반복하여 클러스터 분산이 작은 결과를 선택함

클러스터 모양에 제한적임

클러스터 개수를 해야 함(실제로는 알 수 없음)

# 병합 군집

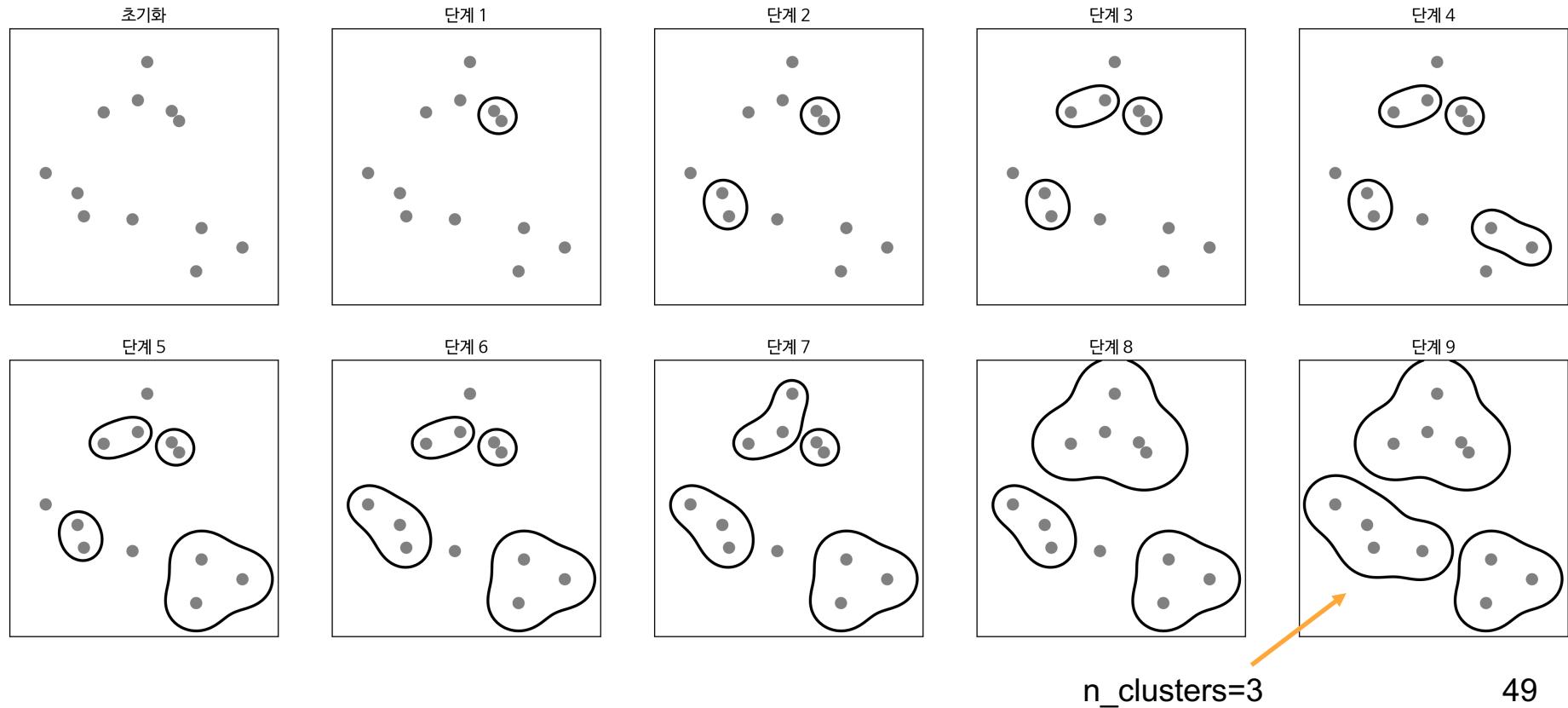
# 병합 군집 agglomerative clustering

데이터 포인트를 하나의 클러스터로 지정해 지정된 클러스터 개수가 될 때까지 두 클러스터를 합쳐 나감

비슷한 클러스터를 선택하는 방법을 linkage 매개변수에 지정함

- ward: 기본값, 클러스터 내의 분산을 가장 작게 만드는 두 클러스터를 병합(클러스터의 크기가 비슷해짐)
- average: 클러스터 포인트 사이의 평균 거리가 가장 작은 두 클러스터를 병합
- complete: 클러스터 포인트 사이의 최대 거리가 가장 짧은 두 클러스터를 병합

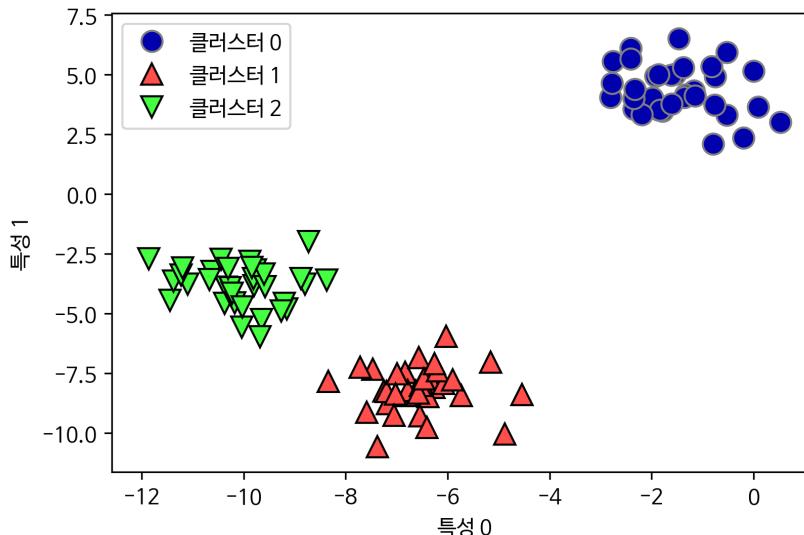
# 병합 군집 example



# AgglomerativeClustering

새로운 데이터에 적용할 수 없음(fit\_predict 만 있음)

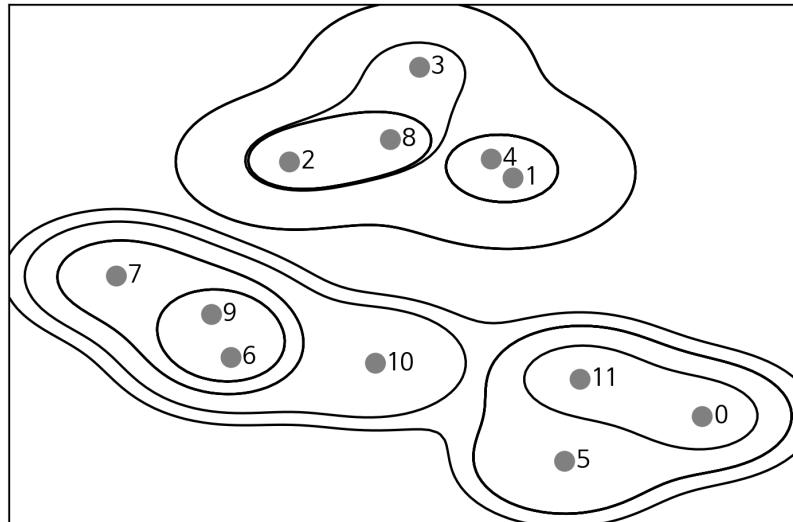
```
In [64]: from sklearn.cluster import AgglomerativeClustering  
X, y = make_blobs(random_state=1)  
  
agg = AgglomerativeClustering(n_clusters=3)  
assignment = agg.fit_predict(X)
```



# 계층적 군집

병합 군집을 사용하려면 클러스터의 개수를 지정해야 함

병합 군집은 계층적 군집이므로 클러스터가 병합되는 과정을 살피면 클러스터 개수를 선택하는 데 도움이 됨



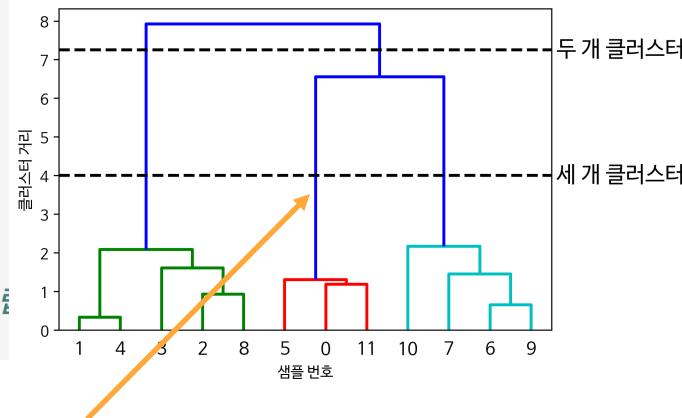
# 덴드로그램 dendrogram

2차원 이상의 계층적 군집을 표현하기 위한 도구

scipy의 연결 linkage 함수와 덴드로그램 함수를 사용

```
In [66]: # SciPy에서 ward 구집 함수와 덴드로그램 함수를 임포트합니다
from scipy.cluster.hierarchy import dendrogram, ward

X, y = make_blobs(random_state=0, n_samples=12)
# 데이터 배열 X에 ward 함수를 적용합니다
# SciPy의 ward 함수는 병합 군집을 수행할 때 생성된
# 거리 정보가 담긴 배열을 리턴합니다
linkage_array = ward(X)
# 클러스터 간의 거리 정보가 담긴 linkage_array를 사용해 덴드로그램을 그림
dendrogram(linkage_array)
```



# DBSCAN

# DBSCAN 특징

클러스터 개수를 미리 지정할 필요 없음

복잡한 형상에 적용 가능, 노이즈 포인트 구분

k-평균이나 병합 군집보다는 느림

데이터가 많은 밀집 지역을 찾아 다른 지역과 구분하는 클러스터를 구성함

밀집 지역의 포인트를 핵심 샘플이라고 부름

한 데이터 포인트에서  $\text{eps}$  안의 거리에  $\text{min\_samples}$  개수만큼 데이터가 있으면 이 포인트를 핵심 샘플로 분류함. → 클러스터가 됨

# DBSCAN 알고리즘

처음 무작위로 포인트를 선택하고  $\text{eps}$  거리안의 포인트를 찾음

$\text{eps}$  거리안의 포인트 <  $\text{min\_samples}$  이면 **잡음 포인트(-1)**로 분류

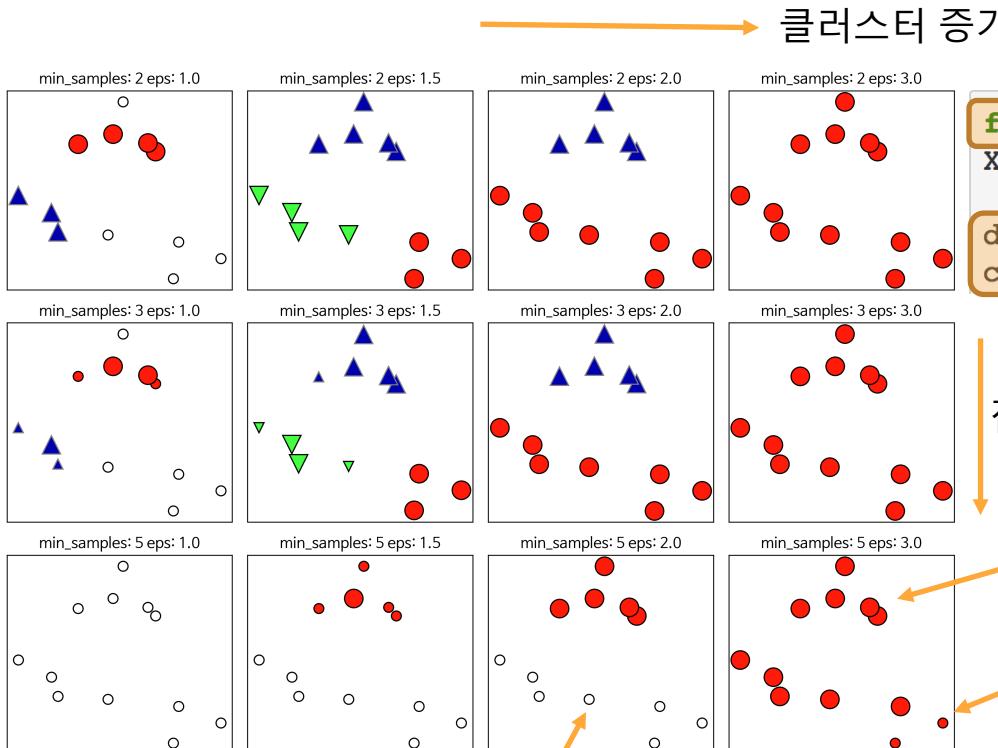
$\text{eps}$  거리안의 포인트 >  $\text{min\_samples}$  이면 **핵심 포인트**로 분류, 새 클러스터 할당

핵심 포인트안의  $\text{eps}$  안의 포인트를 확인하여 클러스터 할당이 없으면 핵심 샘플의 클러스터를 할당함(**경계 포인트**)

$\text{eps}$  안의 포인트가 이미 핵심 샘플이면 그 포인트의 이웃을 또 차례로 방문하여 클러스터가 자라남

아직 방문하지 못한 포인트를 다시 선택해 동일한 과정을 반복함

# DBSCAN + 인공 데이터셋



# eps, min\_samples

eps가 가까운 포인트의 범위를 정하기 때문에 더 중요함

eps가 아주 작으면 핵심 포인트가 생성되지 않고 모두 잡음으로 분류됨

eps를 아주 크게하면 모든 포인트가 하나의 클러스터가 됨

eps로 클러스터의 개수를 간접적으로 조정할 수 있음

StandardScaler나 MinMaxScaler로 특성의 스케일을 조정할 필요 있음

min\_samples는 덜 조밀한 지역이 잡음 포인트가 될지를 결정함

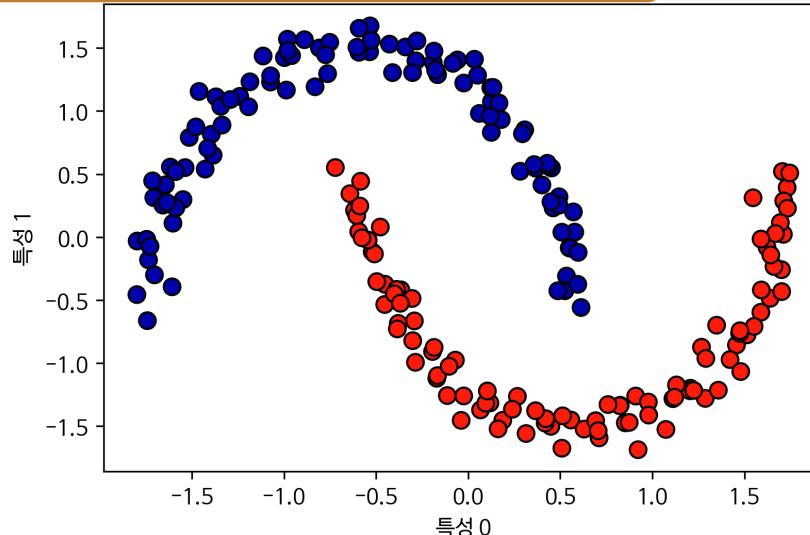
min\_samples 보다 작은 개수가 모여 있는 지역은 잡음 포인트가 됨

# DBSCAN + Two Moons

```
In [69]: X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

# 평균이 0, 분산이 1이 되도록 데이터의 스케일을 조정합니다
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)

dbSCAN = DBSCAN()
clusters = dbSCAN.fit_predict(X_scaled)
```



# 군집 알고리즘 비교, 평가

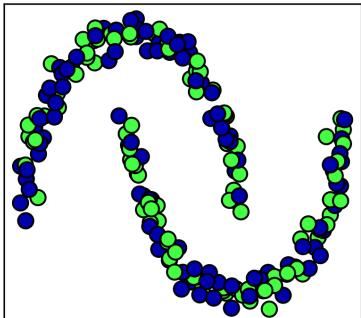
# ARI, NMI

normalized\_mutual\_info\_score

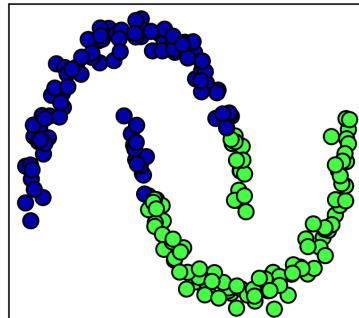
0(무작위)~1(최적) 사이의 값, ARI의 최저값은 -0.5 혹은 -1 임

```
In [70]: from sklearn.metrics.cluster import adjusted_rand_score  
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)  
  
# 평균이 0, 분산이 1이 되도록 데이터의 스케일을 조정합니다  
scaler = StandardScaler()  
scaler.fit(X)  
X_scaled = scaler.transform(X)  
clusters = algorithm.fit_predict(X_scaled)  
adjusted rand score(y, clusters)))
```

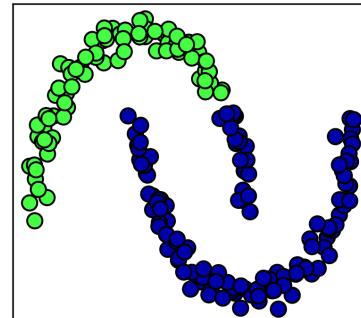
무작위 할당 - ARI: 0.00



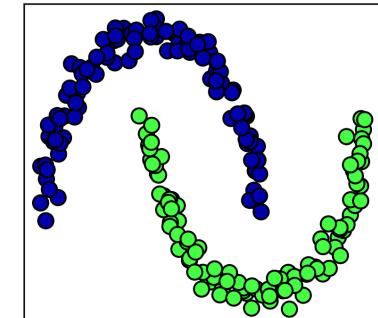
KMeans - ARI: 0.50



AgglomerativeClustering - ARI: 0.61



DBSCAN - ARI: 1.00



# accuracy?

군집의 클래스 레이블은 의미가 없음에도 정확도는 클래스 레이블이 같은지를 확인함

```
In [71]: from sklearn.metrics import accuracy_score  
  
# 포인트가 클러스터로 나뉜 두 가지 경우  
clusters1 = [0, 0, 1, 1, 0]  
clusters2 = [1, 1, 0, 0, 1]  
# 모든 레이블이 달라졌으므로 정확도는 0입니다  
print("정확도: {:.2f}".format(accuracy_score(clusters1, clusters2)))  
# 같은 포인트가 클러스터에 모였으므로 ARI는 1입니다  
print("ARI: {:.2f}".format(adjusted_rand_score(clusters1, clusters2)))
```

정확도: 0.00

ARI: 1.00

# 실루엣 silhouette 계수

ARI, NMI는 타깃값이 있어야 가능, 애플리케이션 보다 알고리즘을 개발할 때 도움이 됨

타깃값 필요없이 클러스터의 밀집 정도를 평가

-1: 잘못된 군집, 0: 중첩된 군집, 1: 가장 좋은 군집

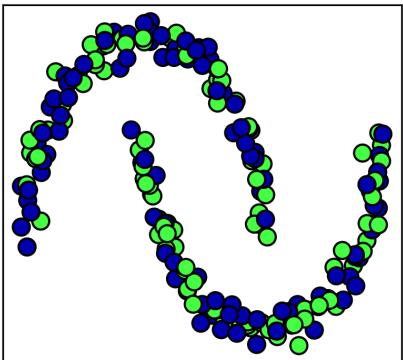
모양이 복잡할 때는 밀집 정도를 평가하는 것이 잘 맞지 않음

원형 클러스터의 실루엣 점수가 높게 나옴

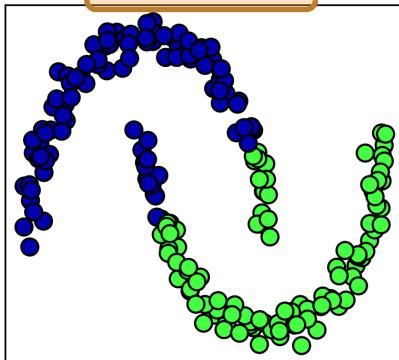
# silhouette\_score

```
In [72]: from sklearn.metrics.cluster import silhouette_score  
silhouette_score(X_scaled, clusters)
```

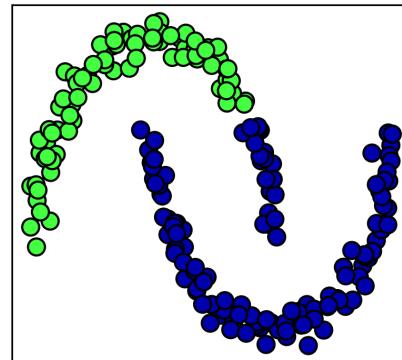
무작위 할당: -0.00



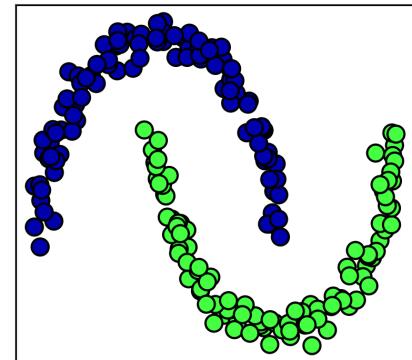
KMeans : 0.49



AgglomerativeClustering : 0.46



DBSCAN : 0.38



# 군집 평가의 어려움

실루엣 점수가 높다하더라도 찾아낸 군집이 흥미로운 것인지는 알 수 없음

사진 애플리케이션이 두 개의 클러스터를 만들었다면

앞모습 vs 옆모습

밤사진 vs 낮사진

아이폰 vs 안드로이드

클러스터가 기대한 대로인지는 직접 확인해야만 알 수 있습니다.

군집 + LFW

# DBSCAN

2063x100

In [73]:

```
# LFW 데이터에서 고유얼굴을 찾은 다음 데이터를 변환합니다
from sklearn.decomposition import PCA
pca = PCA(n_components=100, whiten=True, random_state=0)
pca.fit_transform(X_people)
X_pca = pca.transform(X_people)
```

In [74]:

```
# 기본 매개변수로 DBSCAN을 적용합니다
dbscan = DBSCAN()
labels = dbscan.fit_predict(X_pca)
print("고유한 레이블: {}".format(np.unique(labels)))
```

min\_samples=5

고유한 레이블: [-1] 잡음 포인트

In [75]:

```
dbscan = DBSCAN(min_samples=3)
labels = dbscan.fit_predict(X_pca)
print("고유한 레이블: {}".format(np.unique(labels)))
```

고유한 레이블: [-1]

In [76]:

```
dbscan = DBSCAN(min_samples=3, eps=15)
labels = dbscan.fit_predict(X_pca)
print("고유한 레이블: {}".format(np.unique(labels)))
```

고유한 레이블: [-1 0]

클러스터 범위 증가

# DBSCAN 잡음 포인트

```
In [76]: dbSCAN = DBSCAN(min_samples=3, eps=15)
labels = dbSCAN.fit_predict(X_pca)
print("고유한 레이블: {}".format(np.unique(labels)))
```

고유한 레이블: [-1 0]



# LWF의 클러스터

eps=1  
클러스터 수: 1  
클러스터 크기: [2063]

eps=3  
클러스터 수: 1  
클러스터 크기: [2063]

eps=5  
클러스터 수: 1  
클러스터 크기: [2063]

eps=7  
클러스터 수: 14  
클러스터 크기: [2004 3 14 7 4 3 3 4 4 3 3 5 3 3 ]

eps=9  
클러스터 수: 4  
클러스터 크기: [1307 750 3 3 ]

eps=11  
클러스터 수: 2  
클러스터 크기: [ 413 1650 ]

eps=13  
클러스터 수: 2  
클러스터 크기: [ 120 1943 ]

대다수의 얼굴이미지는 비슷하거나 비슷하지 않음

# k-평균 + LWF

In [81]:

n\_clusters = 10

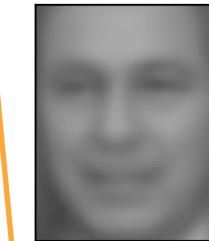
# k-평균으로 클러스터를 추출합니다

km = KMeans(n\_clusters=n\_clusters, random\_state=0)

labels\_km = km.fit\_predict(X\_pca)

print("k-평균의 클러스터 크기: {}".format(np.bincount(labels\_km)))

100x5655



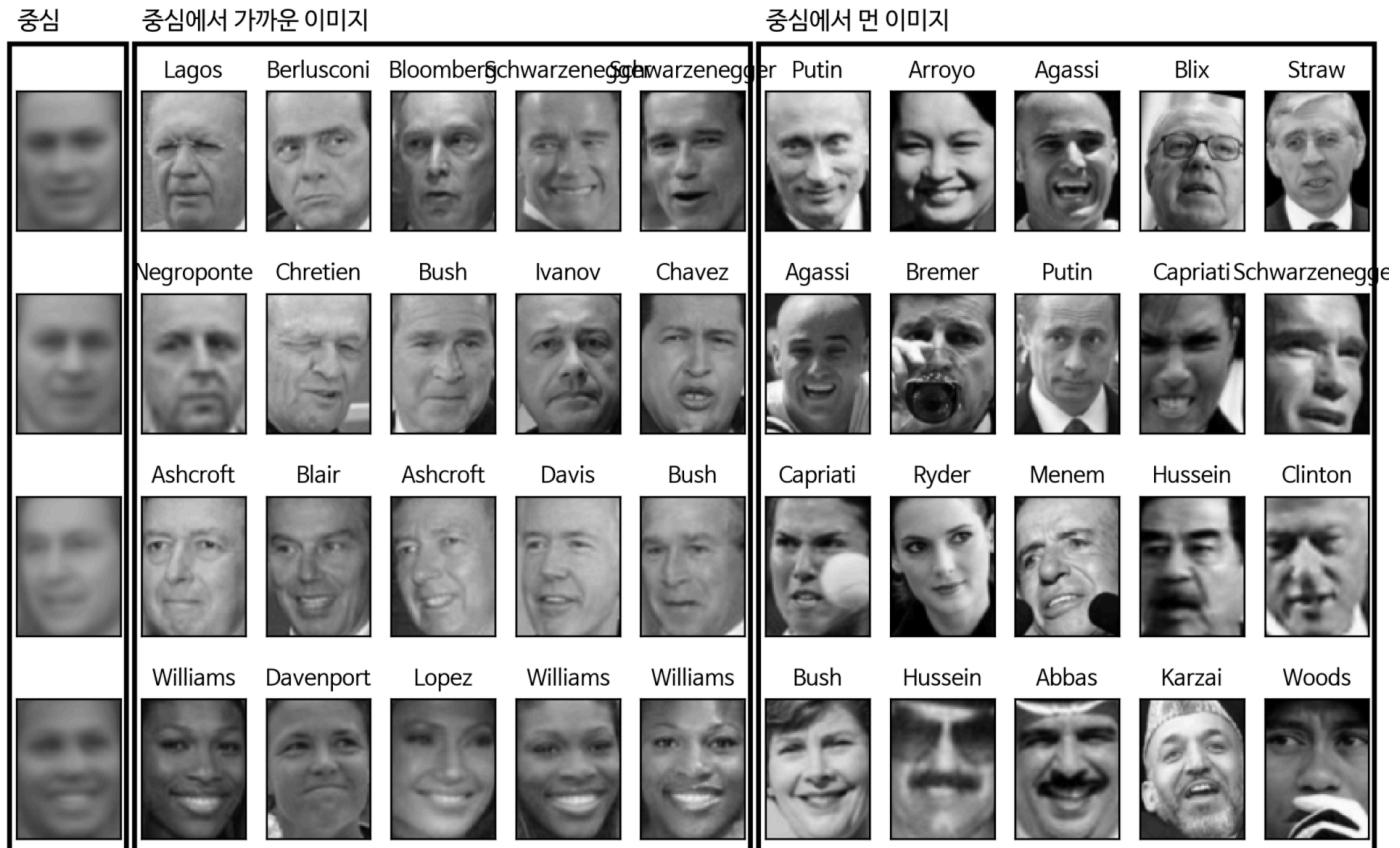
2063x100

10x100

k-평균의 클러스터 크기: [155 175 238 75 358 257 91 219 323 172]

pca.inverse\_transform(km.cluster\_centers\_)

# k-평균의 중심에서 가깝고 먼 이미지



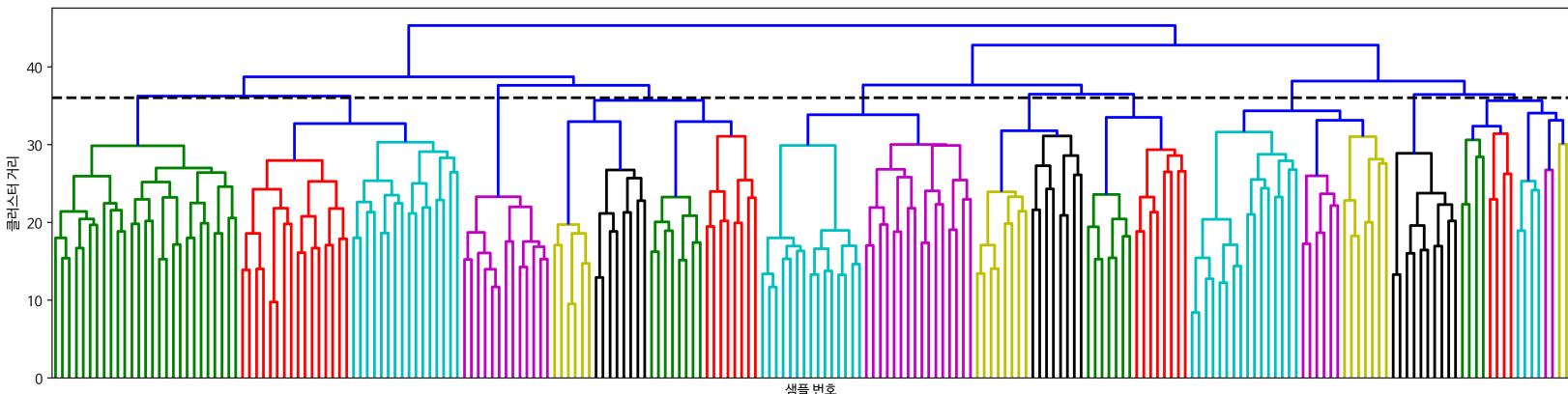
# 병합 군집 + LWF

```
In [84]: # 병합 군집으로 클러스터를 추출합니다  
agglomerative = AgglomerativeClustering(n_clusters=10)  
labels_agg = agglomerative.fit_predict(X_pca)  
print("병합 군집의 클러스터 크기: {}".format(  
    np.bincount(labels_agg)))
```

병합 군집의 클러스터 크기: [169 660 144 329 217 85 18 261 31 149]

```
In [85]: print("ARI: {:.2f}".format(adjusted_rand_score(labels_agg, labels_km)))
```

ARI: 0.09



# 장단점

군집 알고리즘은 정성적 분석 과정이나 탐색적 분석 단계에 유용

k-평균, 병합군집: 클러스터 개수 지정

k-평균: 클러스터의 중심을 데이터 포인트의 분해 방법으로 볼 수 있음

DBSCAN: eps 매개변수로 간접적으로 클러스터 크기를 조정, 잡음 포인트 인식, 클러스터 개수 자동 인식, 복잡한 형상 파악 가능

병합 군집: 계층적 군집으로 덴드로그램을 사용해 표현할 수 있음