빅데이터분석 실습

NMF, 매니폴드 학습, k-Means

데이터 사이언스 전공

담당교수: 곽철완

강의 내용

- 비음수행렬분해 non-negative matrix factorization, NMF
- t-SNE를 이용한 매니폴드 학습
- k-평균 군집

1. 비음수 행렬 분해 NMF

▪ 특징

- 유용한 피처를 추출하기 위한 비지도 학습 알고리즘의 일종이다
- PCA와 비슷하고 차원 축소에도 사용할 수 있다
- PCA와 차이점
  - PCA : 데이터의 분산이 가장 크고 수직인 성분을 찾음
  - NMF: 음수가 아닌 성분과 계수 값을 찾음
- 주성분과 계수는 모두 0보다 크거나 같아야 한다 → 음수가 아닌 피처를 가진 데이터에만 적용할 수 있다
- 적용 분야
  - 여러 사람의 목소리가 담긴 오디오 트랙이나 여러 악기로 이루어진 음악

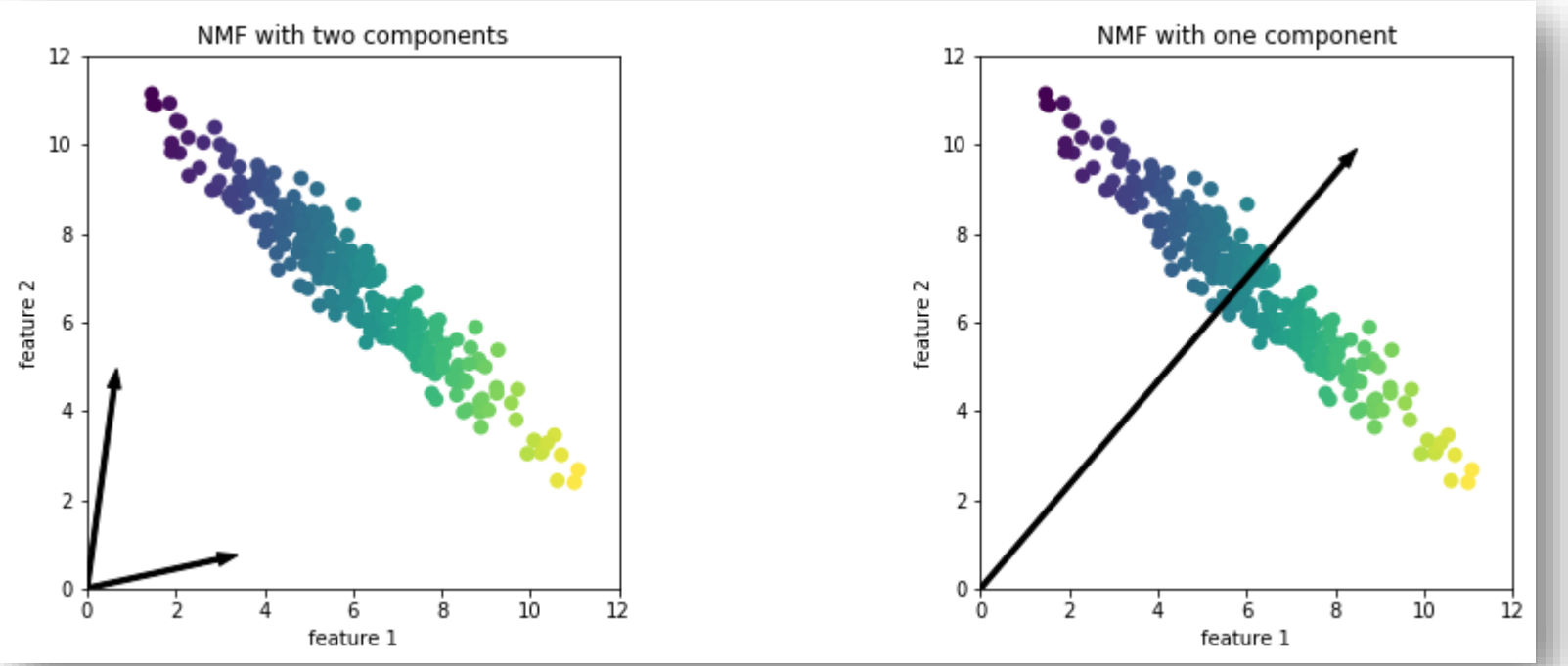
▪ 인위적 데이터에 NMF 적용

- 주어진 데이터가 먼저 양수인지 확인해야 한다
  - 원점(0, 0)에서 데이터가 상대적으로 어디에 위치해 있는지 파악
  - 그러므로, 원점(0, 0)에서 데이터로 가는 방향을 추출하여 데이터에 음수가 포함하지 않은 것을 파악

- NMF 적용한 결과 예

mglearn.plots.plot_nmf_illustration( )

```
mglearn.plots.plot_nmf_illustration( )
```



- 왼쪽 그림
  - 성분이 2개인 NMF로 데이터 세트의 모든 포인트를 양수로 이루어진 2개의 성분으로 표현할 수 있다
  - 만약 성분이 피처 수 만큼 아주 많다면, 알고리즘은(화살표) 각 성분 끝에 위치한 포인트를 가리키는 방향을 선택할 것이다
- 오른쪽 그림
  - 하나의 성분을 사용한다면, 데이터를 가장 잘 표현할 수 있는 평균으로 향하는 성분을 만든다
  - PCA와 반대로 성분 개수를 줄이면 특정 방향이 제거될 뿐만 아니라 전체 성분도 완전히 바뀐다 → 모든 성분을 동등하게 취급

▪ 얼굴 이미지에 NMF 적용

- LFW 데이터 세트에 NMF를 적용한다
- NMF의 핵심 매개변수는 추출할 성분의 개수이다
  - 보통 이 값은 피처 수보다 작다(만약 크다면 픽셀 하나가 2개의 성분으로 나뉘어 표현될 수 있다)
- NMF를 사용해 데이터를 재구성하는데 성분의 개수 주는 영향을 파악한다

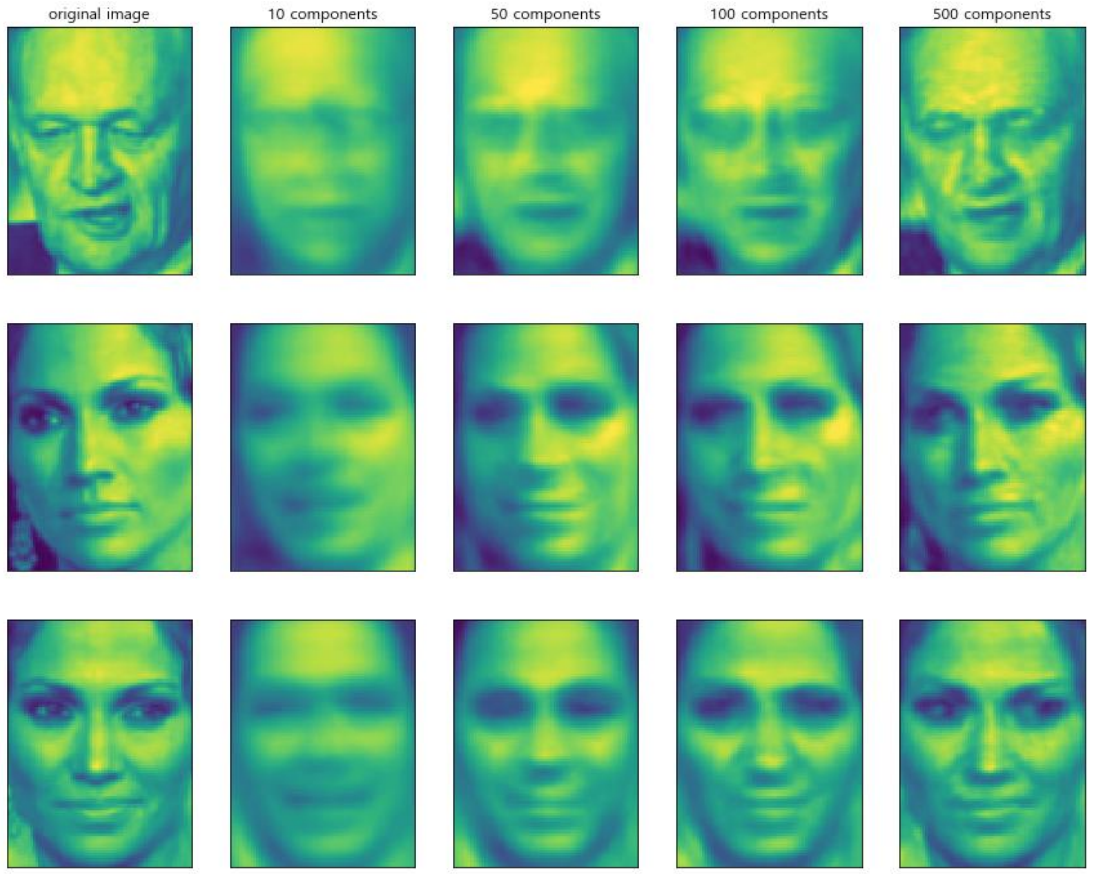
```
mglearn.plots.plot_nmf_faces(X_train, X_test, image_shape)
```

```
from sklearn.datasets import fetch_lfw_people
people = fetch_lfw_people(min_faces_per_person=20, resize=0.7)
image_shape = people.images[0].shape

fig, axes = plt.subplots(2, 5, figsize=(15, 8), subplot_kw={'xticks' : ( ), 'yticks' : ( )})
for target, image, ax in zip(people.target, people.images, axes.ravel( )):
    ax.imshow(image)
    ax.set_title(people.target_names[target])

mask = np.zeros(people.target.shape, dtype=np.bool)
for target in np.unique(people.target):
    mask[np.where(people.target == target)[0][:50]] = 1

X_people = people.data[mask]
y_people = people.target[mask]
X_people = X_people / 225.
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_people, y_people, stratify=y_people, random_state=0)

mglearn.plots.plot_nmf_faces(X_train, X_test, image_shape)
```


original image
10 components
50 components
100 components
500 components

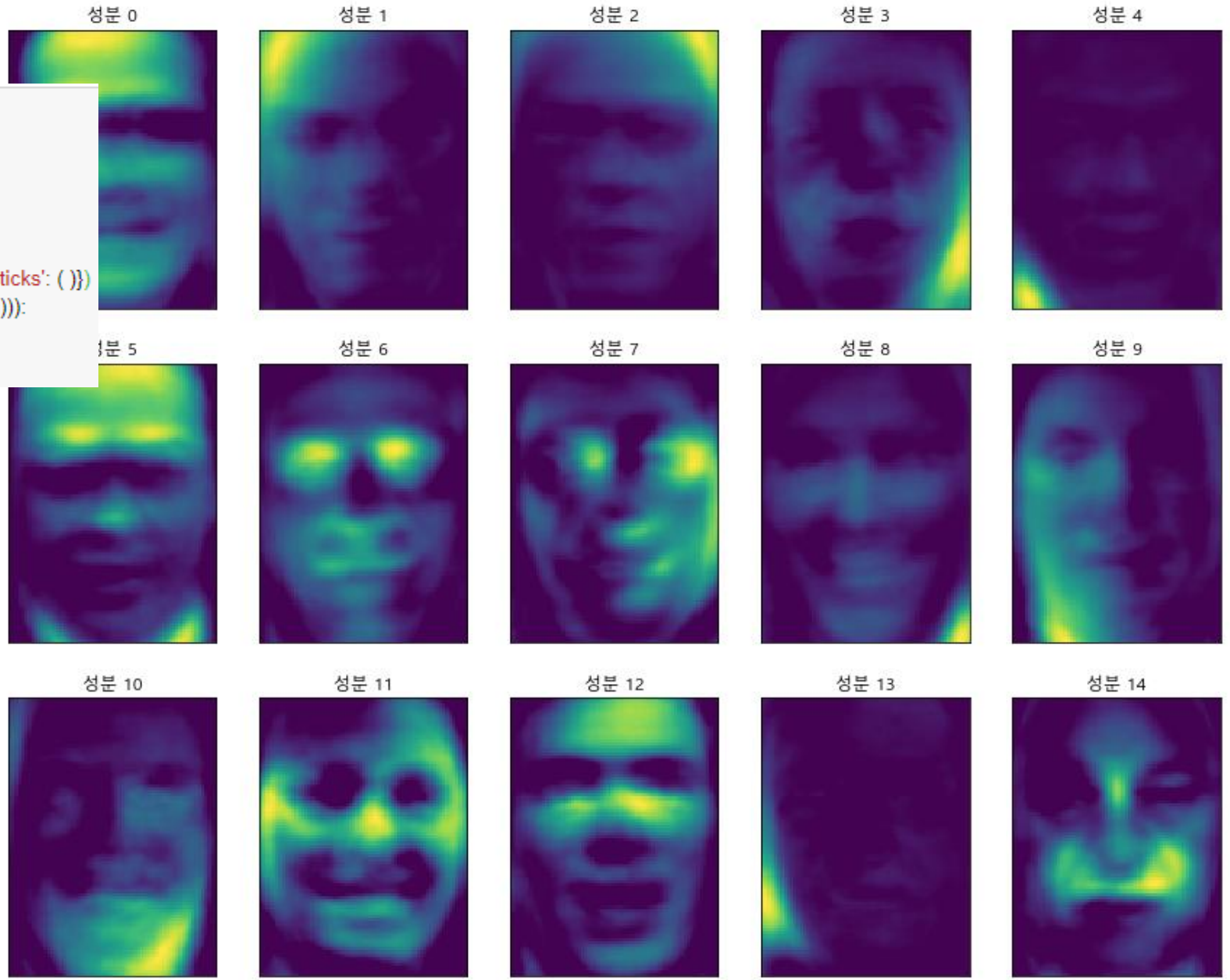
- NMF 성분 개수에 따른 얼굴 이미지의 재구성
  - PCA를 사용했을 때와 비슷하지만, 품질이 약간 떨어진다
    - PCA가 재구성 측면에서 최선의 방향을 찾기 때문임
  - NMF는 데이터를 인코딩하거나 재구성하는 용도보다는 데이터에 있는 유용한 패턴을 찾는데 활용한다



◦ 성분 15개를 추출한 결과 확인

```
from sklearn.decomposition import NMF
nmf = NMF(n_components=15, random_state=0)
nmf.fit(X_train)
X_train_nmf = nmf.transform(X_train)
X_test_nmf = nmf.transform(X_test)

fig, axes = plt.subplots(3, 5, figsize=(15, 12), subplot_kw={'xticks': ( ), 'yticks': ( )})
for i, (component, ax) in enumerate(zip(nmf.components_, axes.ravel( ))):
        ax.imshow(component.reshape(image_shape))
        ax.set_title("성분 {}".format(i))
```

```
from sklearn.decomposition import NMF
nmf = NMF(n_components=15, random_state=0)
nmf.fit(X_train)
X_train_nmf = nmf.transform(X_train)
X_test_nmf = nmf.transform(X_test)

fig, axes = plt.subplots(3, 5, figsize=(15, 12), subplot_kw={'xticks': ( ), 'yticks': ( )})
for i, (component, ax) in enumerate(zip(nmf.components_, axes.ravel( ))):
    ax.imshow(component.reshape(image_shape))
    ax.set_title("성분 {}".format(i))
```

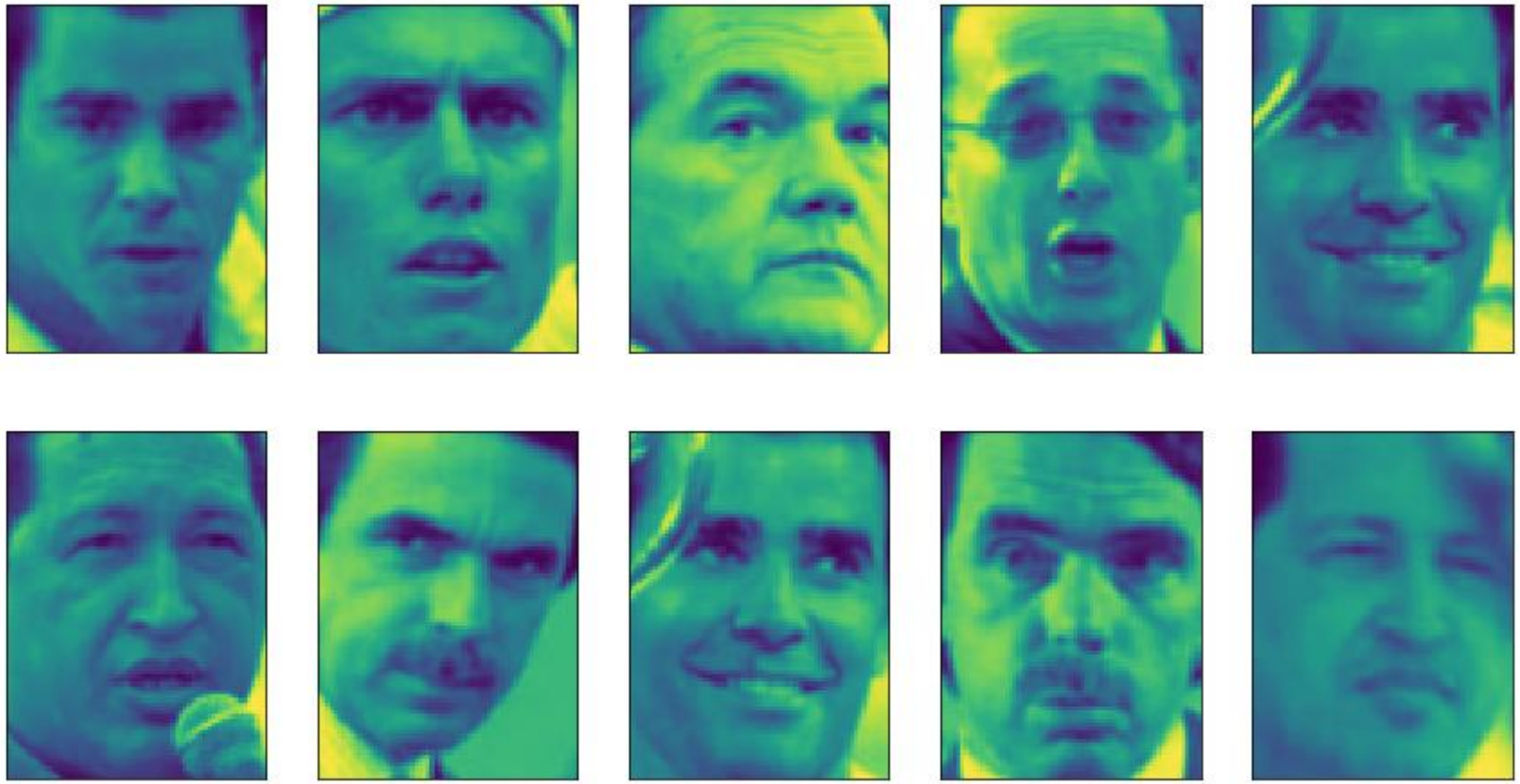


- 성분3은 얼굴이 약간 오른쪽으로 향하고 있는 모습이다
- 성분3을 기준으로 정렬하여 처음 이미지 10개를 출력한다
  - plt.subplots(2, 5): 2행 5열 = 10개 이미지

```
compn = 3
inds = np.argsort(X_train_nmf[:, compn])[::-1]
fig, axes = plt.subplots(2, 5, figsize=(15, 8), subplot_kw={‘xticks’: ( ), ‘yticks’: ( )})
for i, (ind, ax) in enumerate(zip(inds, axes.ravel( ))):
        ax.imshow(X_train[ind].reshape(image_shape))
```


```
compn = 3
inds = np.argsort(X_train_nmf[:, compn])[::-1]
fig, axes = plt.subplots(2, 5, figsize=(15, 8), subplot_kw={'xticks': ( ), 'yticks': ( )})
for i, (ind, ax) in enumerate(zip(inds, axes.ravel( ))):
    ax.imshow(X_train[ind].reshape(image_shape))
```

◦ 성분3을 기준으로 출력한 결과 대부분의 얼굴이 오른쪽으로 향하고 있다



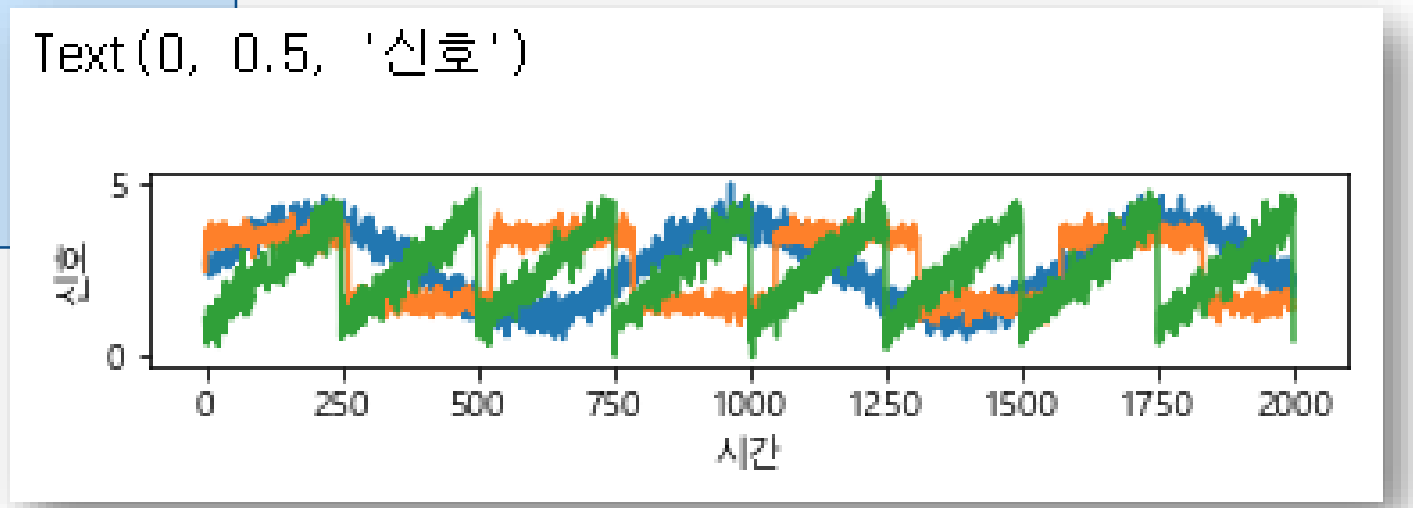
```
compn = 7
inds = np.argsort(X_train_nmf[:, compn])[::-1]
fig, axes = plt.subplots(2, 5, figsize=(15, 8), subplot_kw={'xticks': ( ), 'yticks': ( )})
for i, (ind, ax) in enumerate(zip(inds, axes.ravel( ))):
    ax.imshow(X_train[ind].reshape(image_shape))
```

- 성분7을 기준으로 출력한 결과 대부분의 얼굴이 왼쪽으로 향하고 있다

▪ NMF 적용 사례

- NMF는 특정 패턴을 추출하는데 효과적이다
  - 그러므로, 소리, 유전자 표현, 텍스트 데이터 같은 분야에 적합
- 다음은 3개의 서로 다른 입력으로부터 합성된 신호이다

```
S = mglearn.datasets.make_signals()
plt.figure(figsize = (6, 1))
plt.plot(S, '-')
plt.xlabel("시간")
plt.ylabel("신호")
```



- 원본 신호로 복원
  - 원본 데이터를 이용해 100개의 측정 데이터를 만든다

```
A = np.random.RandomState(0).uniform(size=(100, 3))
X = np.dot(S, A.T)
print("측정 데이터 형태: ", X.shape)
```

```
A = np.random.RandomState(0).uniform(size=(100, 3))
X = np.dot(S, A.T)
print("측정 데이터 형태: ", X.shape)
```

```
측정 데이터 형태:  (2000, 100)
```

◦ NMF를 이용해 3개의 신호를 복원한다

```
nmf = NMF(n_components = 3, random_state=42)
S_ = nmf.fit_transform(X)
print("복원한 신호 데이터 형태: ", S_.shape)
```

```
nmf = NMF(n_components = 3, random_state=42)
S_ = nmf.fit_transform(X)
print("복원한 신호 데이터 형태: ", S_.shape)
```

```
복원한 신호 데이터 형태:  (2000, 3)
```

◦ PCA와 비교를 위해 PCA를 적용한다

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

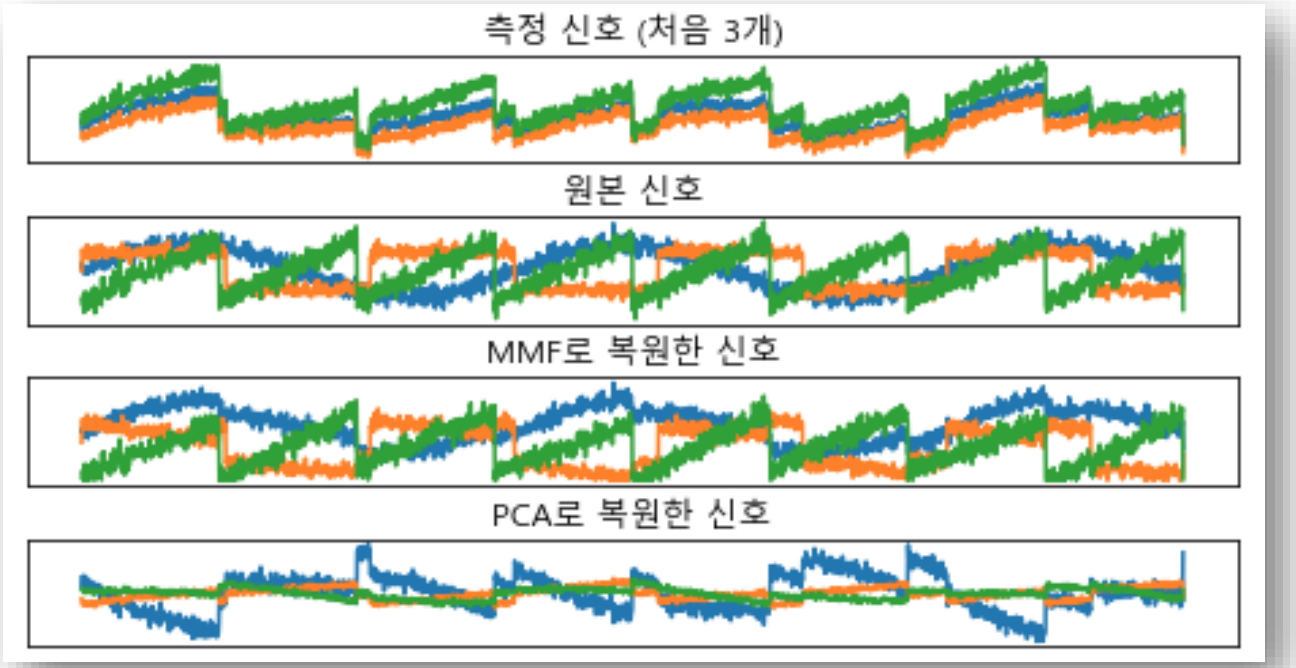
```
from sklearn.decomposition import PCA
pca = PCA(n_components=3)
H = pca.fit_transform(X)
```

◦ NMF와 PCA로 찾은 신호를 비교한다

```
models = [X, S, S_, H]
names = [‘측정 신호 (처음 3개)’, ‘원본 신호’, ‘NMF로 복원한 신호’, ‘PCA로 복원한 신호’]
fig, axes = plt.subplots(4, figsize = (8, 4), gridspec_kw={‘hspace’ : .5},
                         subplot_kw={‘xticks’: ( ), ‘yticks’: ( )})
for model, name, ax in zip(models, names, axes):
        ax.set_title(name)
        ax.plot(model[:, :3], ‘-’)
```

```
models = [X, S, S_, H]
names = ['측정 신호 (처음 3개)', '원본 신호', 'MMF로 복원한 신호', 'PCA로 복원한 신호']
fig, axes = plt.subplots(4, figsize = (8, 4), gridspec_kw={'hspace' : .5},
                         subplot_kw={'xticks': ( ), 'yticks': ( )})
for model, name, ax in zip(models, names, axes):
        ax.set_title(name)
        ax.plot(model[:, :3], '-')
```

- 오른쪽 그림에서 보듯이 NMF는 신호를 잘 복원했지만, PCA는 실패했다
- PCA는 대부분을 파랑색 신호의 성분을 사용해 복원했다
- NMF는 성분의 순서가 없다 – 여기에서 원본 신호와 우연히 같았을 뿐이다



2. t-SNE를 이용한 매니폴드 학습 manifold learning

- 특징
  - 시각화 알고리즘으로 시각화가 목적이기 때문에 3개 이하의 피처를 추출한다
  - 학습용 데이터를 새로운 표현으로 변환시키지만, 새로운 데이터에 적용하지 못한다
    - 평가용 데이터 세트에 적용할 수 없다
  - 그러므로, 탐색적 데이터 분석에 유용한다

아이디어

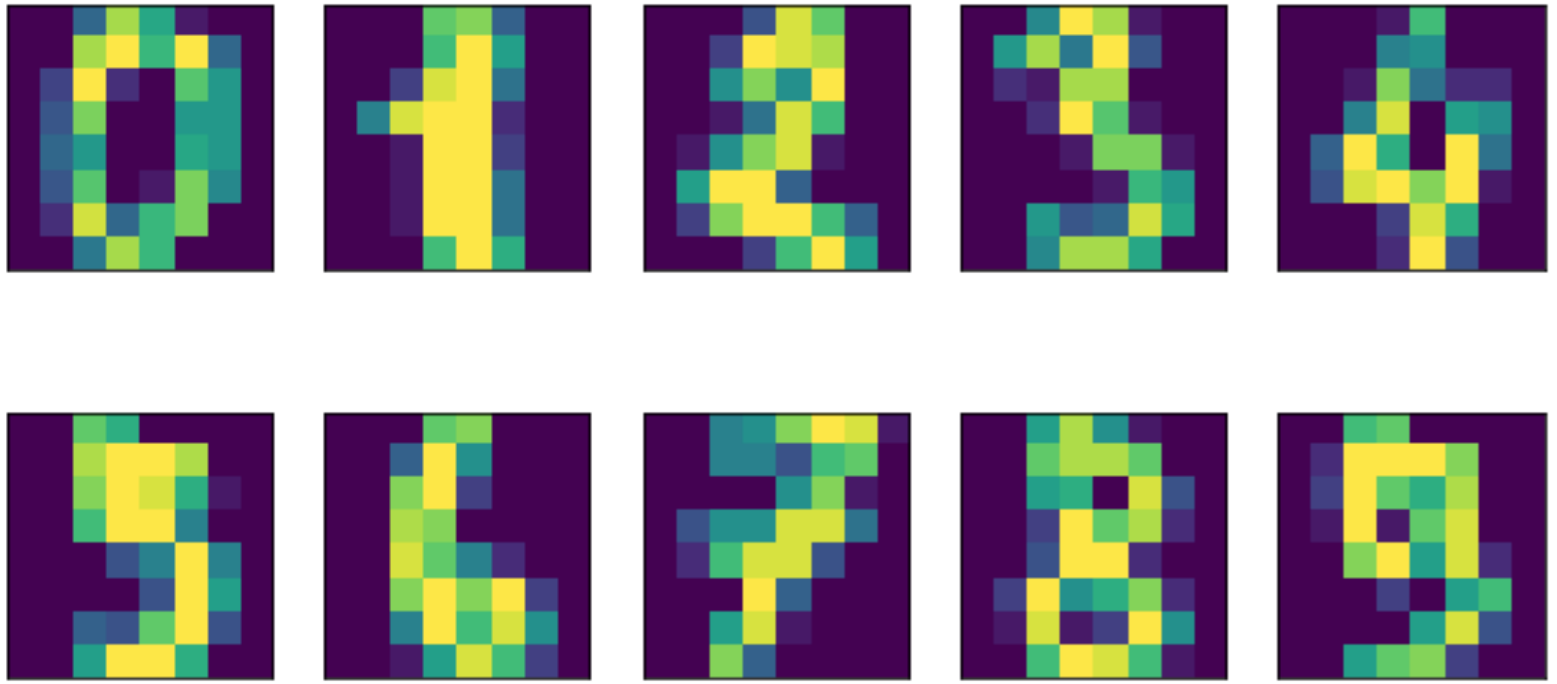
- 기본 아이디어
  - 데이터 포인트 사이의 거리를 가장 잘 보존하는 2차원 표현을 찾는 것이다
- 각 데이터 포인트를 2차원으로 무작위로 표현한 후, 원본 특성 공간에서 가까운 포인트는 가깝게, 멀리 떨어진 포인트는 멀어지게 만든다
- 가까이 있는 포인트에 더 비중을 둔다: 이웃 데이터 포인트에 대한 정보를 보존하려고 노력한다

▪ 적용

- 손글씨 숫자 데이터 세트(scikit-klearn 에 포함되어 있음)를 이용하여 매니폴드 학습을 적용한다
  - 이 데이터 세트의 각 포인트는 0에서 9 사이의 손글씨 숫자를 표현한 8x8 크기의 흑백 이미지이다

```
from sklearn.datasets import load_digits
digits = load_digits( )

fig, axes = plt.subplots(2, 5, figsize = (10, 5), subplot_kw={'xticks': ( ),
'yticks': ( )})
for ax, img in zip(axes.ravel( ), digits.images):
        ax.imshow(img)
```

```
from sklearn.datasets import load_digits
digits = load_digits( )

fig, axes = plt.subplots(2, 5, figsize = (10, 5), subplot_kw={'xticks': ( ), 'yticks': ( )})
for ax, img in zip(axes.ravel( ), digits.images):
        ax.imshow(img)
```

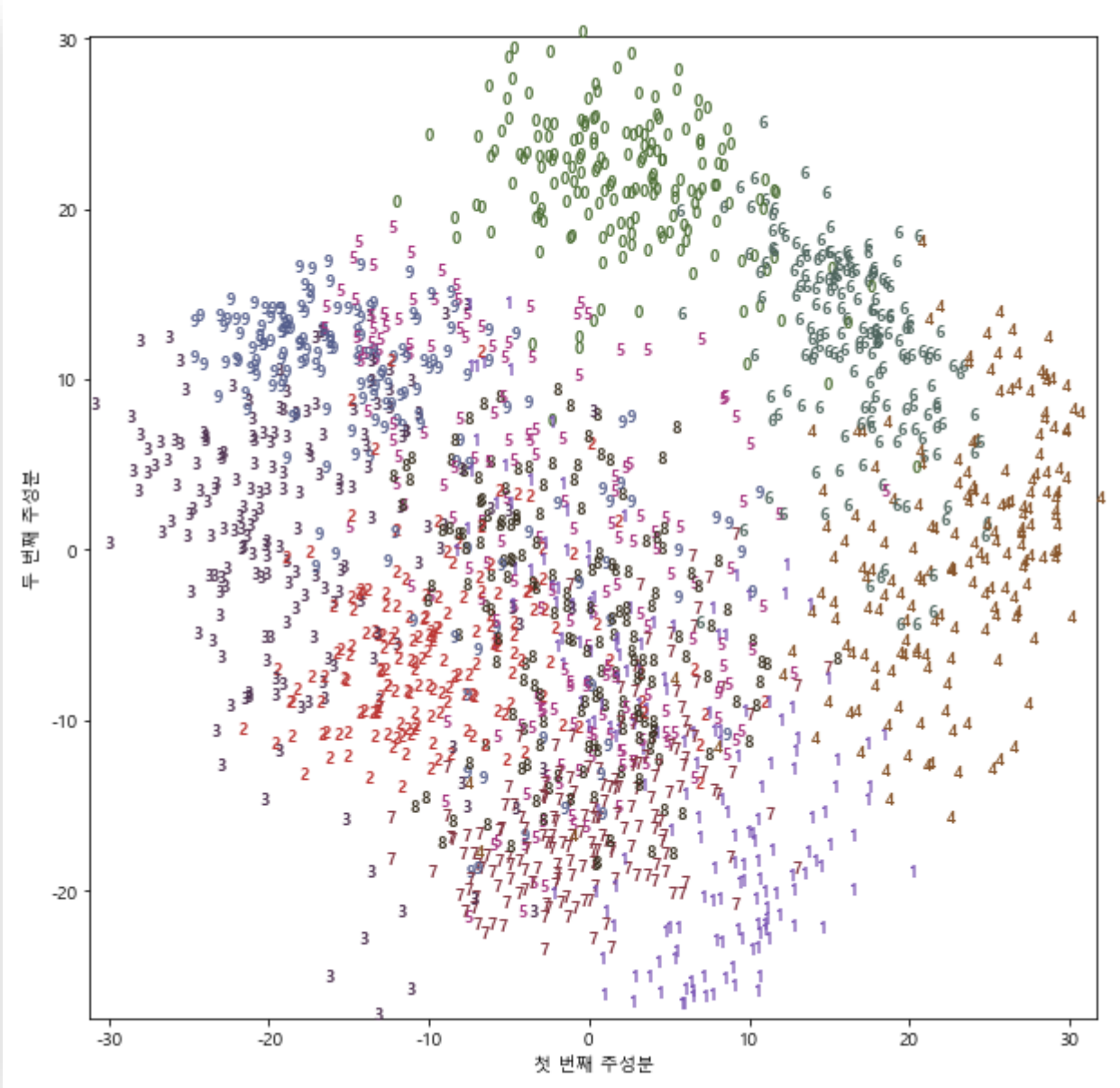


- PCA를 이용해 데이터를 2차원으로 축소해서 시각화한다
  - 처음 2개의 주성분을 이용해 그래프를 그리고 각 샘플에 해당하는 클래스를 숫자로 표시하였다

```
pca =PCA(n_components = 2)
pca.fit(digits.data)

digits_pca = pca.transform(digits.data)
colors = ["#476A2A", "#7851B8", "#BD3430", "#4A2D4E", "#875525",
        "#A83683", "#4E655E", "#853541", "#3A3120", "#535D8E"]
plt.figure(figsize=(10, 10))
plt.xlim(digits_pca[:, 0].min( ), digits_pca[:, 0].max( ))
plt.ylim(digits_pca[:, 1].min( ), digits_pca[:, 1].max( ))
for i in range(len(digits.data)):
    plt.text(digits_pca[i, 0], digits_pca[i, 1], str(digits.target[i]),
             color = colors[digits.target[i]],
             fontdict = {'weight': 'bold', 'size' : 9})
plt.xlabel("첫 번째 주성분")
plt.ylabel("두 번째 주성분")
```

```
pca =PCA(n_components = 2)
pca.fit(digits.data)

digits_pca = pca.transform(digits.data)
colors = ["#476A2A", "#7851B8", "#BD3430", "#4A2D4E", "#875525",
          "#A83683", "#4E655E", "#853541", "#3A3120", "#535D8E"]
plt.figure(figsize=(10, 10))
plt.xlim(digits_pca[:, 0].min( ), digits_pca[:, 0].max( ))
plt.ylim(digits_pca[:, 1].min( ), digits_pca[:, 1].max( ))
for i in range(len(digits.data)):
    plt.text(digits_pca[i, 0], digits_pca[i, 1], str(digits.target[i]),
             color = colors[digits.target[i]],
             fontdict = {'weight': 'bold', 'size' : 9})
plt.xlabel("첫 번째 주성분")
plt.ylabel("두 번째 주성분")
```

- 각 클래스(0 ~ 9)가 어디에 위치해 있는지 보기 위해 실제 숫자를 사용하여 산점도를 그렸다
- 0, 4, 6은 2개의 주성분 만으로도 잘 분리되어 보인다
- 다른 숫자들은 많이 중첩되어 있다



- t-SNE 매니폴드 학습을 적용하여 비교
  - TSNE는 transform 메서드가 없으므로, fit_transform을 사용한다

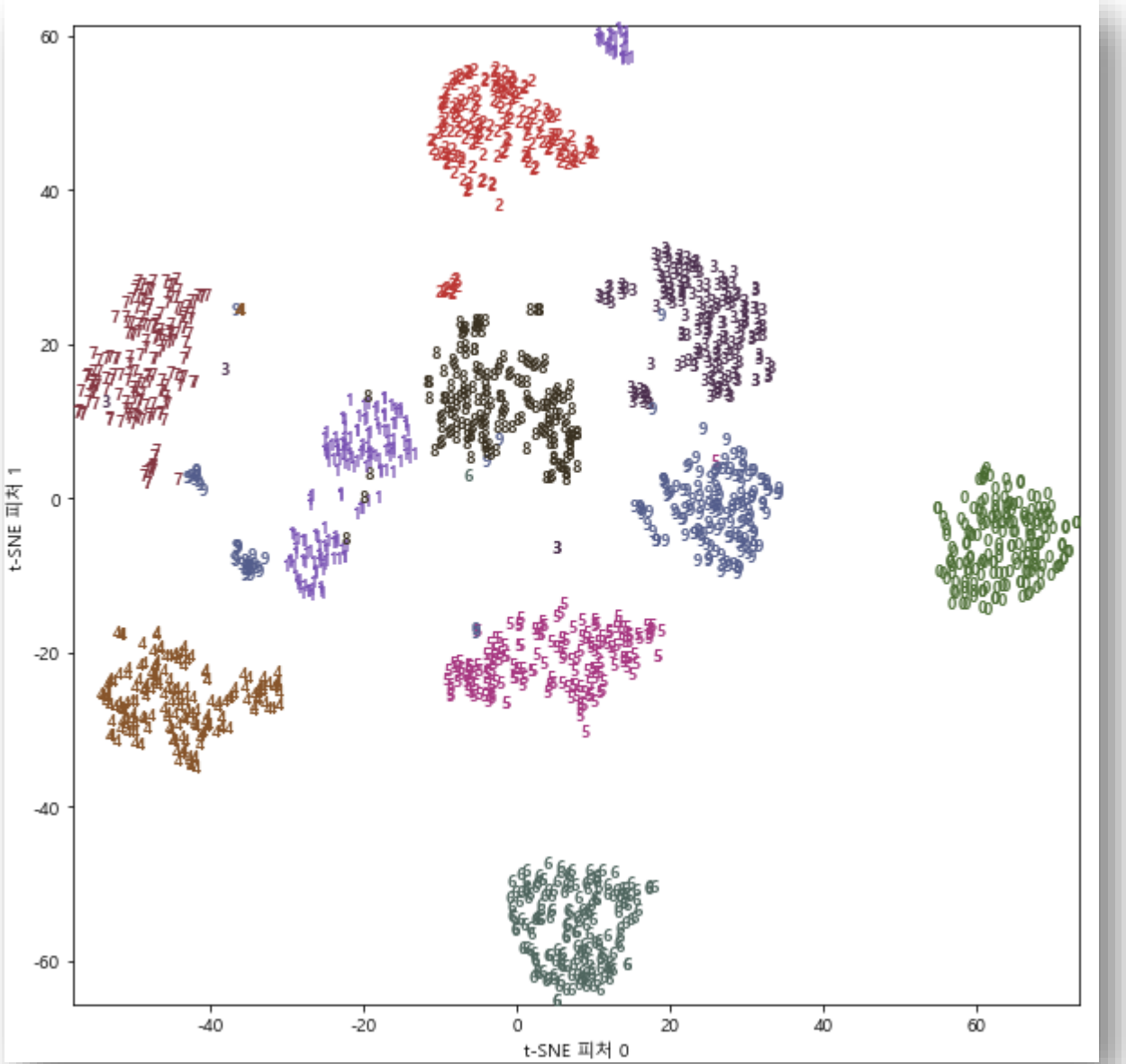
```
from sklearn.manifold import TSNE
tsne = TSNE(random_state = 42)
digits_tsne = tsne.fit_transform(digits.data)
```

```
from sklearn.manifold import TSNE
tsne = TSNE(random_state = 42)
digits_tsne = tsne.fit_transform(digits.data)
```

```
plt.figure(figsize=(10, 10))
plt.xlim(digits_tsne[:, 0].min( ), digits_tsne[:, 0].max( ) + 1)
plt.ylim(digits_tsne[:, 1].min( ), digits_tsne[:, 1].max( ) + 1)
for i in range(len(digits.data)):
      plt.text(digits_tsne[i, 0], digits_tsne[i, 1], str(digits.target[i]),
                color = colors[digits.target[i]],
                fontdict = {‘weight’: ‘bold’, ‘size’ : 9})
plt.xlabel(“t-SNE 피처 0“)
plt.ylabel(“t-SNE 피처 1“)
```


```
plt.figure(figsize=(10, 10))
plt.xlim(digits_tsne[:, 0].min( ), digits_tsne[:, 0].max( ) + 1)
plt.ylim(digits_tsne[:, 1].min( ), digits_tsne[:, 1].max( ) + 1)
for i in range(len(digits.data)):
    plt.text(digits_tsne[i, 0], digits_tsne[i, 1], str(digits.target[i]),
             color = colors[digits.target[i]],
             fontdict = {"weight": 'bold', 'size' : 9})
plt.xlabel("t-SNE 피처 0")
plt.ylabel("t-SNE 피처 1")
```

- 모든 결과가 확실하게 구분 되었다



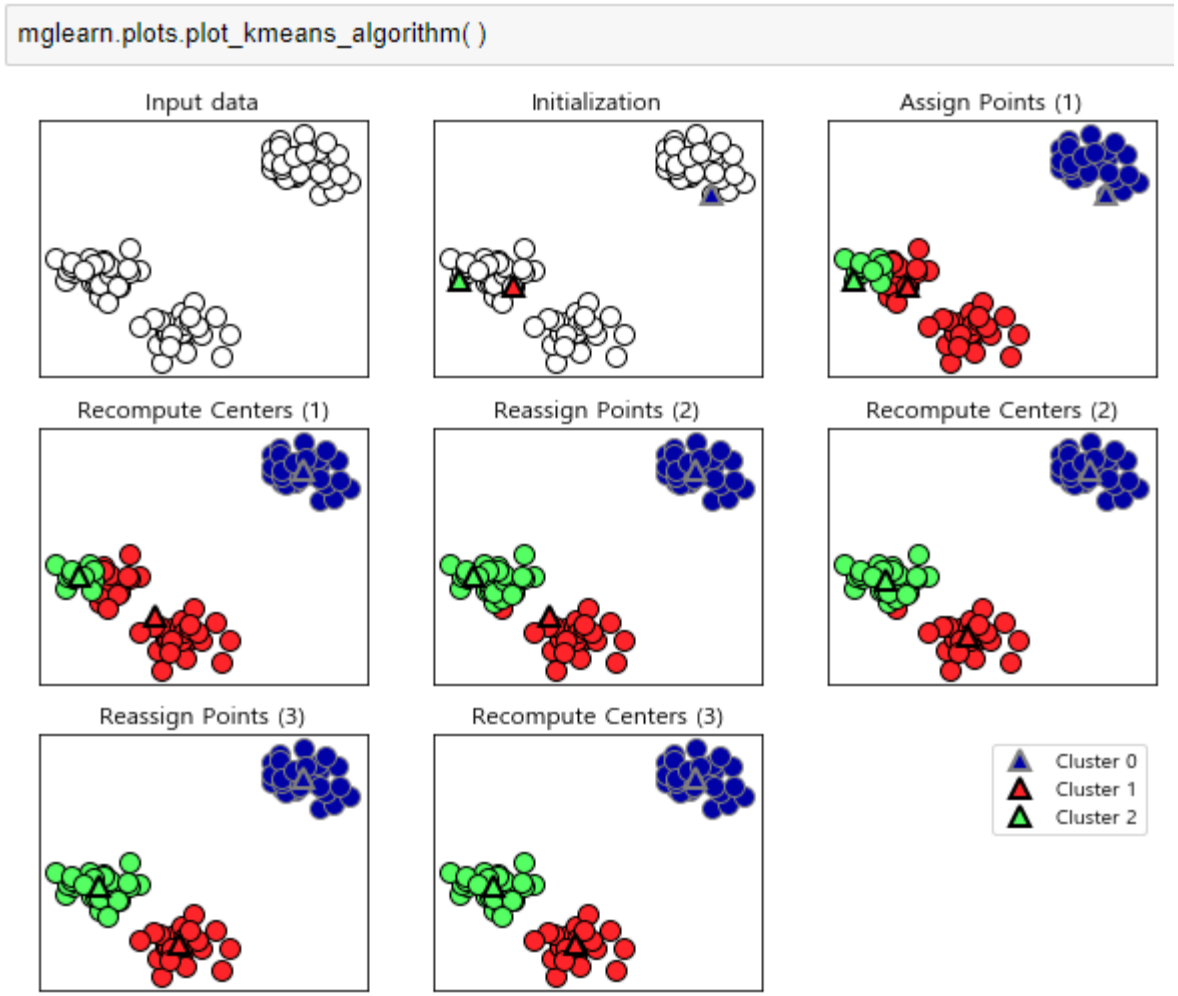
3. k-평균 군집

▪ 기본 개념

- 가장 간단하고 널리 사용되는 군집 알고리즘이다
- 개념
  - 데이터의 어떤 영역을 대표하는 클러스터 중심 cluster center를 찾는다
  - 데이터 포인트를 가장 가까운 클러스터 중심에 할당한다
  - 그런 다음 클러스터에 할당된 데이터 포인트의 평균으로 클러스터 중심을 다시 지정한다
  - 클러스터에 할당되는 데이터 포인트에 변화가 없을 때 알고리즘이 종료된다

mglearn.plots.plot_kmeans_algorithm( )

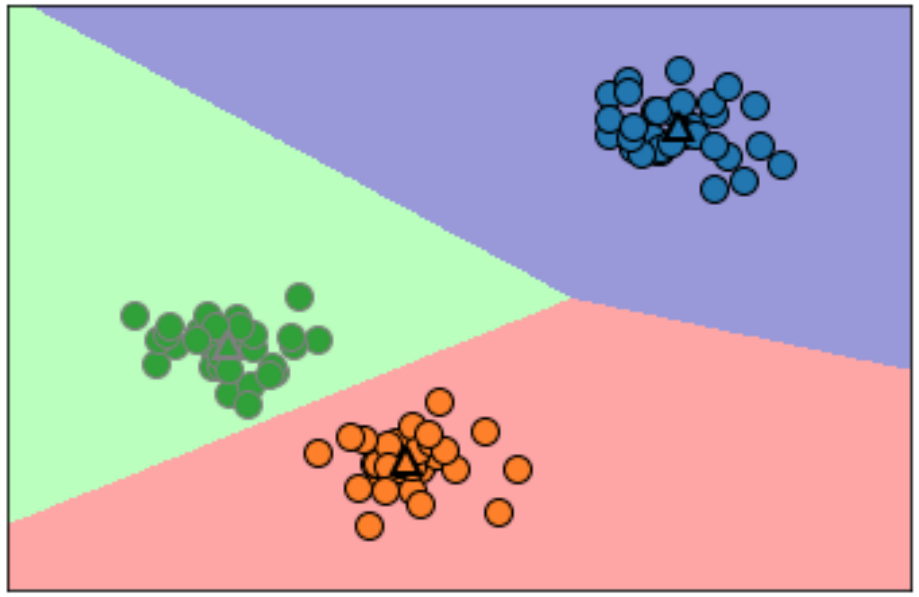
- 왼쪽 그림에서 삼각형이 클러스터 중심이다
- 각 데이터 포인트를 가장 가까운 클러스터 중심에 할당한다
- 할당된 포인트의 평균값으로 클러스터 중심을 갱신한다
- 위의 과정을 반복한다
- 중심에 할당되는 포인트의 변화가 없으므로 알고리즘은 멈춘다



- 새로운 데이터 포인트가 주어지면 학습한 k-평균 알고리즘은 가장 가까운 클러스터 중심에 할당한다

```
mglearn.plots.plot_kmeans_boundaries( )
```

```
mglearn.plots.plot_kmeans_boundaries( )
```



▪ 알고리즘 적용

- 사용 데이터 세트 : 인위적 데이터 세트
- 객체를 생성하고 찾고자 하는 클러스터의 수를 지정한다
- 다음 fit 메서드를 호출한다

```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
X, y = make_blobs(random_state=1)
kmeans = KMeans(n_clusters = 3)
kmeans.fit(X)
```


```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
X, y = make_blobs(random_state=1)
kmeans = KMeans(n_clusters = 3)
kmeans.fit(X)
```

- 알고리즘을 적용하면 X에 담긴 각 학습용 데이터 포인트에 클러스터 레이블이 할당된다
- kmeans.labels_ 속성에서 이 레이블을 확인할 수 있다

```
print("클러스터 레이블: \n{ }".format(kmeans.labels_))
```

```
print("클러스터 레이블: \n{}".format(kmeans.labels_))
```

```
클러스터 레이블:
[1 0 0 0 2 2 2 0 1 1 0 0 2 1 2 2 2 1 0 0 2 0 2 1 0 2 2 1 1 2 1 1 2 1 0 2 0
 0 0 2 2 0 1 0 0 2 1 1 1 1 0 2 2 2 1 2 0 0 1 1 0 2 2 0 0 2 1 2 1 0 0 0 2 1
 1 0 2 2 1 0 1 0 0 2 1 1 1 1 0 1 2 1 1 0 0 2 2 1 2 1]
```

- 3개 레이블을 지정했으므로 각 클러스터에 0~2까지의 번호가 붙는다

- predict 메서드를 이용해 새로운 데이터의 클러스터 레이블을 예측할 수 있다

```
print(kmeans.predict(X))
```

```
print(kmeans.predict(X))
```

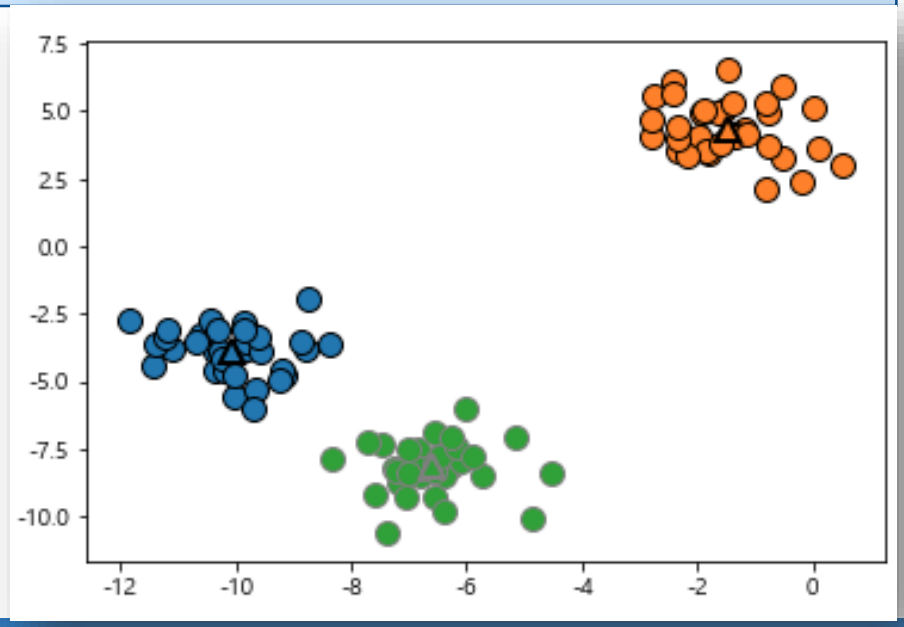
```
[1 0 0 0 2 2 2 0 1 1 0 0 2 1 2 2 2 1 0 0 2 0 2 1 0 2 2 1 1 2 1 1 2 1 0 2 0
 0 0 2 2 0 1 0 0 2 1 1 1 1 0 2 2 2 1 2 0 0 1 1 0 2 2 0 0 2 1 2 1 0 0 0 2 1
 1 0 2 2 1 0 1 0 0 2 1 1 1 1 0 1 2 1 1 0 0 2 2 1 2 1]
```

- 군집은 분류처럼 보이지만, 정답을 모르고 있기 때문에 레이블의 의미가 없다
- 군집 알고리즘을 적용할 때 그룹의 레이블 지정은 중요하지 않다. 초기화를 무작위로 하기 때문에 알고리즘을 다시 실행하면 클러스터 번호가 다르게 부여될 수 있다

◦ 그림 비교하기

```
mglearn.discrete_scatter(X[:, 0], X[:, 1], kmeans.labels_, markers='o')
mglearn.discrete_scatter(
    kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], [0, 1, 2],
    markers='^', markeredgewidth=2)
```

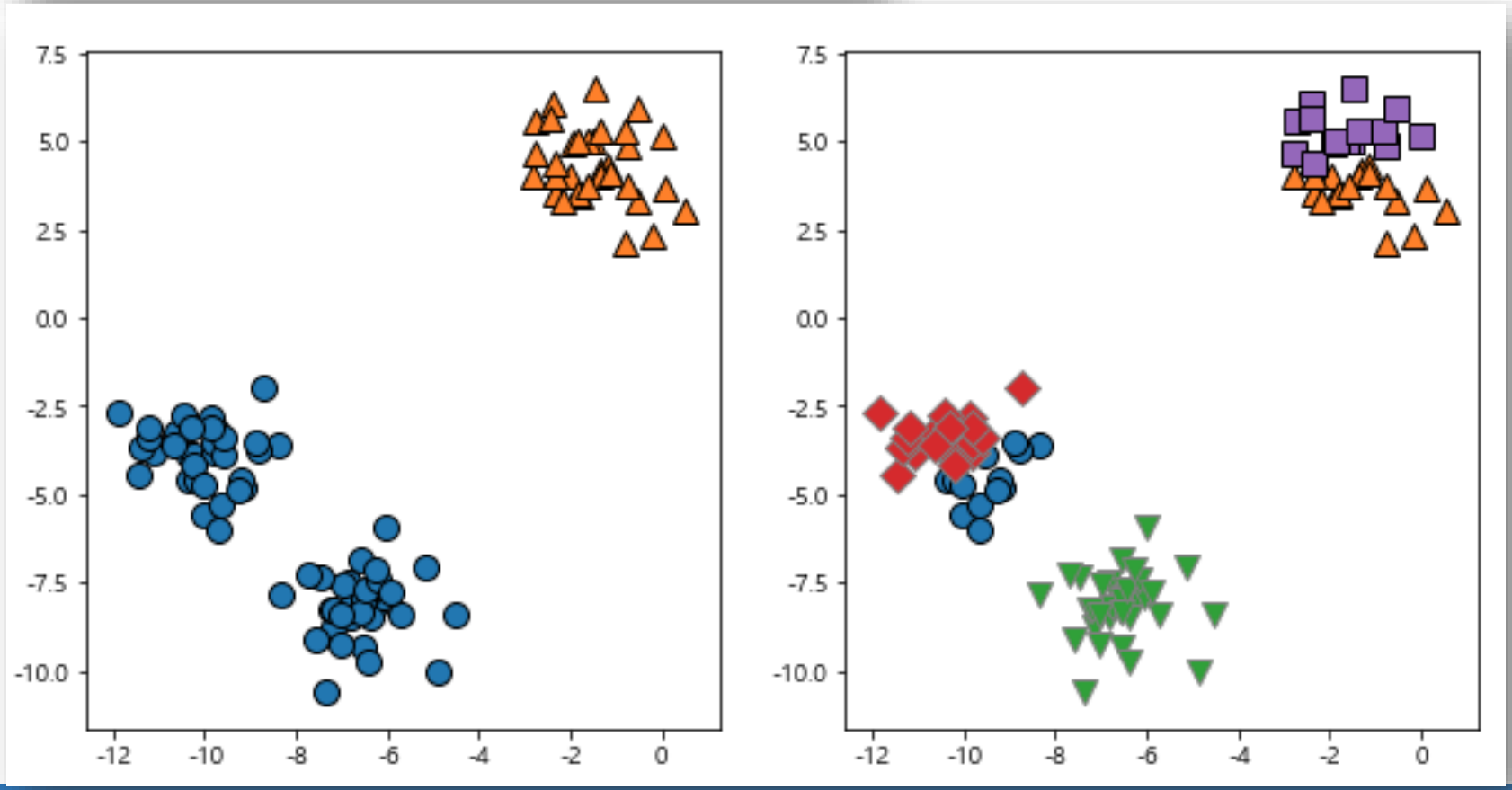
◦ cluster_centers_ 속성에 저장된 클러스터 중심을 삼각형으로 표시했다



◦ 클러스터 수 변경하기
  ◦ 2개의 클러스터 중심과 5개 클러스터 중심 사용 비교

```
fig, axes = plt.subplots(1, 2, figsize=(10, 5)
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
assignments = kmeans.labels_
mglearn.discrete_scatter(X[:, 0], X[:, 1], assignments, ax=axes[0])
kmeans = KMeans(n_clusters=5)
kmeans.fit(X)
assignments = kmeans.labels_
mglearn.discrete_scatter(X[:, 0], X[:, 1], assignments, ax=axes[1])
```

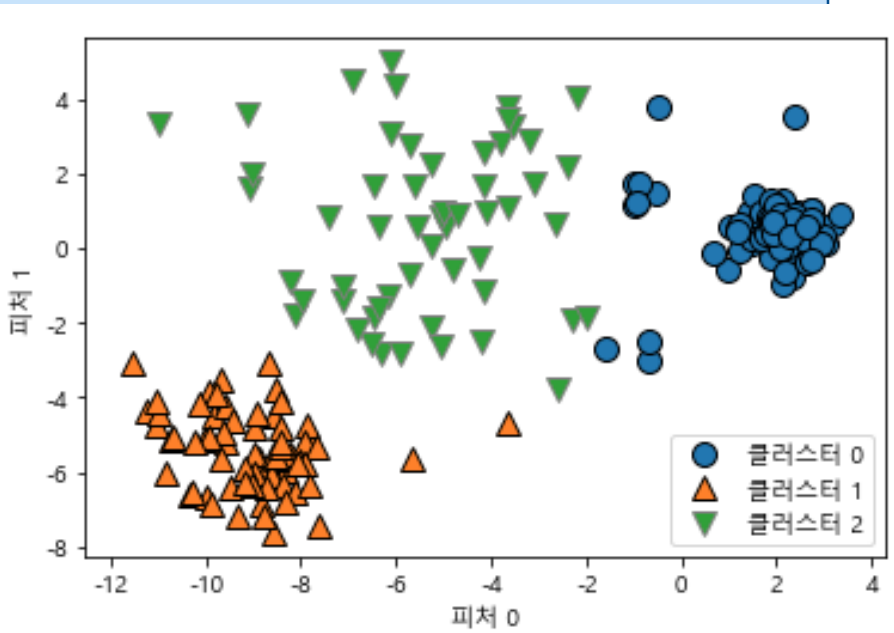
```
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
assignments = kmeans.labels_
mglearn.discrete_scatter(X[:, 0], X[:, 1], assignments, ax=axes[0])
kmeans = KMeans(n_clusters=5)
kmeans.fit(X)
assignments = kmeans.labels_
mglearn.discrete_scatter(X[:, 0], X[:, 1], assignments, ax=axes[1])
```



k-Means 알고리즘이 실패하는 경우

- k-Means 알고리즘은 데이터 세트의 클러스터 개수를 정확하게 구분하지 못한다
- 각 클러스터를 정의하는 것이 중심 하나뿐이므로 클러스터는 둥근 형태로 나타난다 → 비교적 간단한 형태로 구분이 가능하다
- k-Means는 모든 클러스터의 반경이 똑 같다고 가정한다 → 클러스터 중심 사이의 경계를 정확하게 그린다
- 이는 가끔 예상치 않은 결과를 만든다

```
X_varied, y_varied = make_blobs(n_samples=200,
        cluster_std=[1.0, 2.5, 0.5], random_state=170)
y_pred = KMeans(n_clusters = 3, random_state=0).fit_predict(X_varied)
mglearn.discrete_scatter(X_varied[:, 0], X_varied[:, 1], y_pred)
plt.legend(["클러스터 0", "클러스터 1", "클러스터 2"], loc='best')
plt.xlabel("피처 0")
plt.ylabel("피처 1")
```

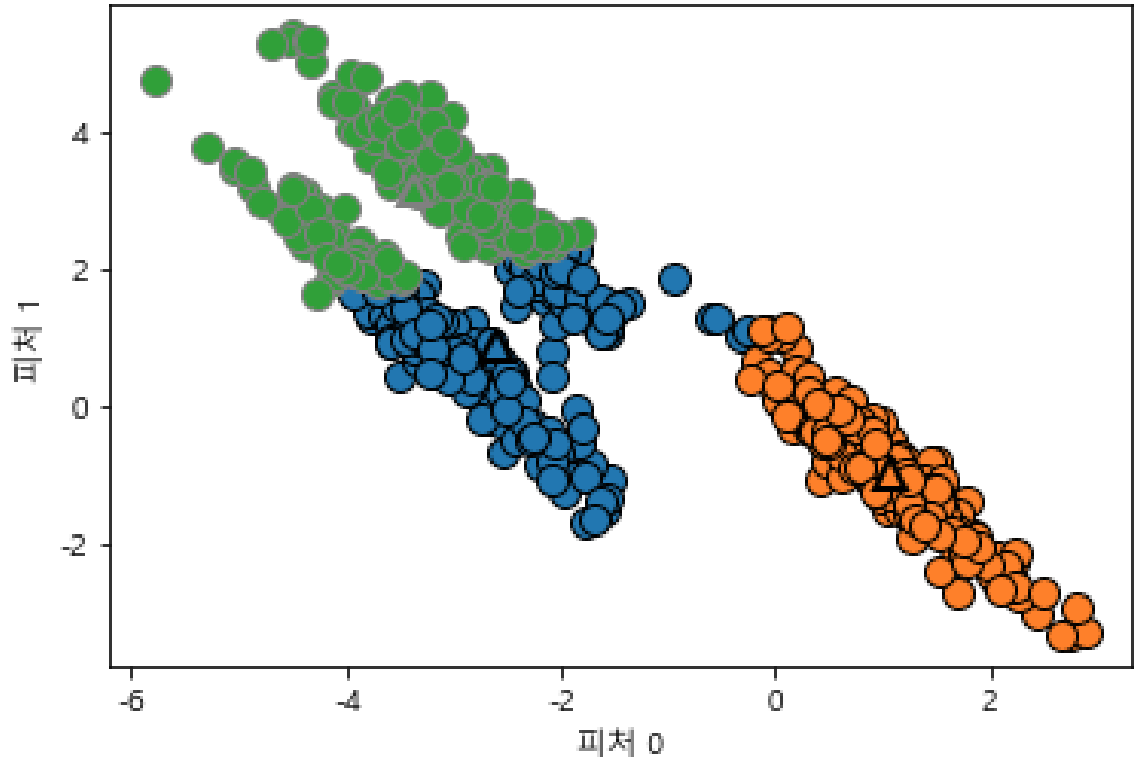


- 클러스터 0, 클러스터1은 잘 모여 있지만, 일부 중심에서 멀리 떨어져 있다
- 클러스터 2는 분산된 모습니다

◦ 또 다른 문제

```
X, y = make_blobs(random_state=170, n_samples=600)
rng = np.random.RandomState(74)
transformation = rng.normal(size=(2,2))
X = np.dot(X, transformation)
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
y_pred = kmeans.predict(X)
mglearn.discrete_scatter(X[:, 0], X[:, 1], kmeans.labels_, markers = 'o')
mglearn.discrete_scatter(
        kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], [0, 1, 2],
        markers='^', markeredgewidth=2)
plt.xlabel("피처 0")
plt.ylabel("피처 1")
```

```
X, y = make_blobs(random_state=170, n_samples=600)
rng = np.random.RandomState(74)
transformation = rng.normal(size=(2,2))
X = np.dot(X, transformation)
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
y_pred = kmeans.predict(X)
mglearn.discrete_scatter(X[:, 0], X[:, 1], kmeans.labels_, markers = 'o')
mglearn.discrete_scatter(
        kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], [0, 1, 2],
        markers='^', markeredgewidth=2)
plt.xlabel("피처 0")
plt.ylabel("피처 1")
```



- k-means는 클러스터에서 모든 방향이 똑같이 중요하다고 가정한다
- 위의 데이터 세트는 대각선으로 늘어져 있다
- k-means는 가장 가까운 클러스터 중심까지의 거리만 고려하기 때문에 이런 데이터는 잘 처리하지 못한다

▪ 벡터 양자화 or 분해 메서드로서의 k-Means

- PCA → 분산이 가장 큰 방향을 찾는다 ---------- 성분의 합
- NMF → 음수가 아닌 성분과 계수 값을 찾는다 ----- 성분의 합
- k-Means → 클러스터 중심으로 각 데이터 포인트를 표현한다
  - 각 포인트가 하나의 성분으로 분해되는 관점으로 보는 것을 벡터 양자화라고 한다

- k-Means와 NMF를 이용한 성분 추출과 재구성

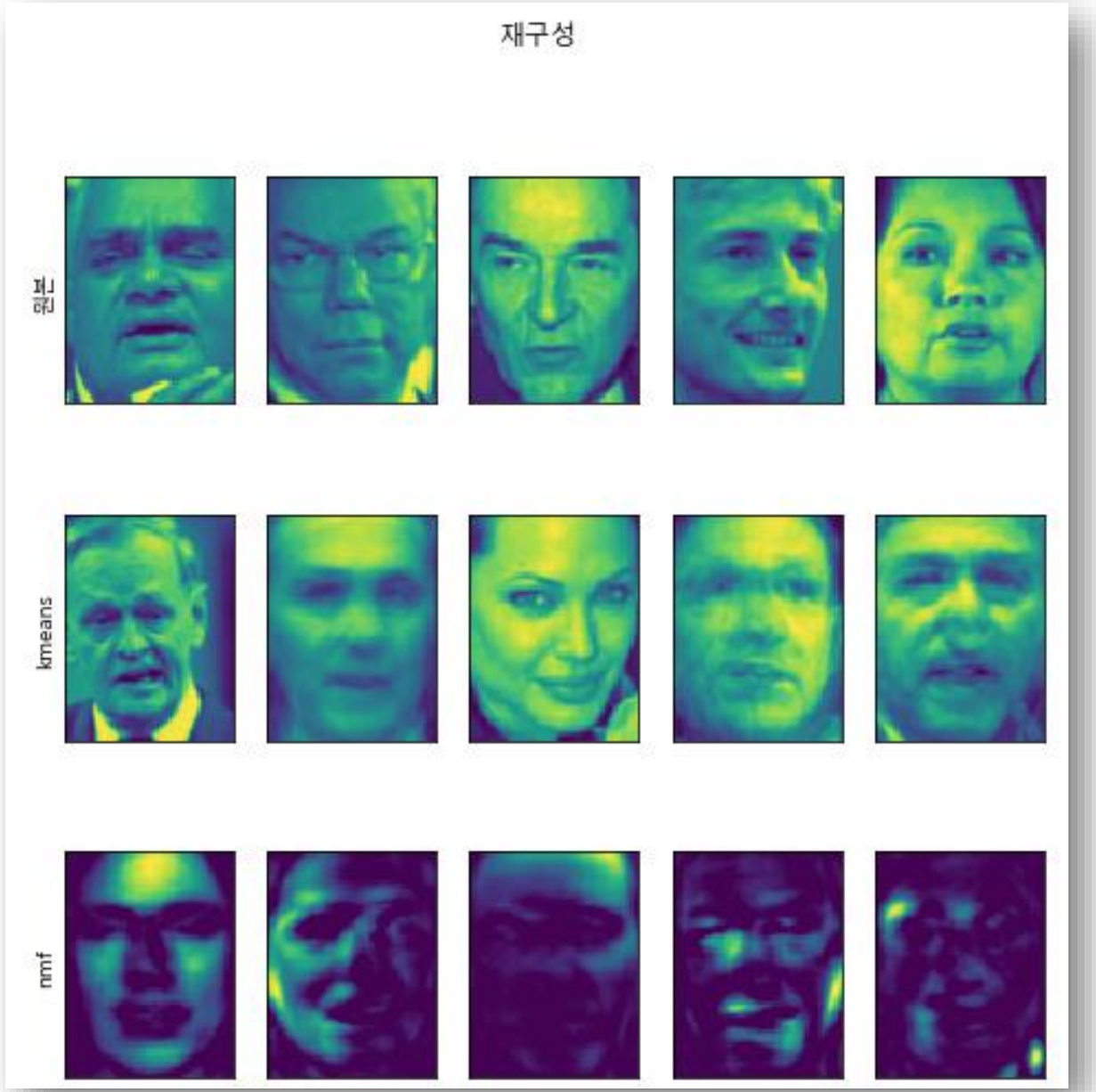
```
X_train, X_test, y_train, y_test = train_test_split(X_people, y_people,
stratify=y_people, random_state=42)
nmf = NMF(n_components=100, random_state=0)
nmf.fit(X_train)
kmeans = KMeans(n_clusters=100, random_state=0)
kmeans.fit(X_train)
X_reconstructed_kmeans = kmeans.cluster_centers_[kmeans.predict(X_test)]
X_reconstructed_nmf = np.dot(nmf.transform(X_test), nmf.components_)
```


```
X_train, X_test, y_train, y_test = train_test_split(X_people, y_people, stratify=y_people, random_state=42)
nmf = NMF(n_components=100, random_state=0)
nmf.fit(X_train)
kmeans = KMeans(n_clusters=100, random_state=0)
kmeans.fit(X_train)
X_reconstructed_kmeans = kmeans.cluster_centers_[kmeans.predict(X_test)]
X_reconstructed_nmf = np.dot(nmf.transform(X_test), nmf.components_)
```

```
fig, axes= plt.subplots(2, 5, figsize=(8, 8), subplot_kw={'xticks': (), 'yticks': ()})
fig.suptitle("추출한 성분")
for ax, comp_kmeans, comp_nmf in zip(
                    axes.T, kmeans.cluster_centers_, nmf.components_):
    ax[0].imshow(comp_kmeans.reshape(image_shape))
    ax[1].imshow(comp_nmf.reshape(image_shape))
axes[0,0].set_ylabel("kmeans")
axes[1,0].set_ylabel("nmf")

fig, axes= plt.subplots(3, 5, figsize=(8, 8), subplot_kw={'xticks': (), 'yticks': ()})
fig.suptitle("재구성")
for ax, orig, comp_kmeans, comp_nmf in zip(
                      axes.T, X_test, kmeans.cluster_centers_, nmf.components_):
    ax[0].imshow(orig.reshape(image_shape))
    ax[1].imshow(comp_kmeans.reshape(image_shape))
    ax[2].imshow(comp_nmf.reshape(image_shape))
axes[0,0].set_ylabel("원본")
axes[1,0].set_ylabel("kmeans")
axes[2,0].set_ylabel("nmf")
```

```
fig, axes= plt.subplots(2, 5, figsize=(8, 6), subplot_kw={'xticks': (), 'yticks': ()})
fig.suptitle("추출한 성분")
for ax, comp_kmeans, comp_nmf in zip(
                axes.T, kmeans.cluster_centers_, nmf.components_):
    ax[0].imshow(comp_kmeans.reshape(image_shape))
    ax[1].imshow(comp_nmf.reshape(image_shape))
axes[0,0].set_ylabel("kmeans")
axes[1,0].set_ylabel("nmf")

fig, axes= plt.subplots(3, 5, figsize=(8, 8), subplot_kw={'xticks': (), 'yticks': ()})
fig.suptitle("재구성")
for ax, orig, comp_kmeans, comp_nmf in zip(
                 axes.T, X_test, kmeans.cluster_centers_, nmf.components_):
    ax[0].imshow(orig.reshape(image_shape))
    ax[1].imshow(comp_kmeans.reshape(image_shape))
    ax[2].imshow(comp_nmf.reshape(image_shape))
axes[0,0].set_ylabel("원본")
axes[1,0].set_ylabel("kmeans")
axes[2,0].set_ylabel("nmf")
```

추출한 성분


재구성
pca
kmeans
nmf

k-Means 요약

- k-means는 비교적 이해하기 쉽고 구현도 쉽고, 빠르게 작동하기 때문에 가장 인기있는 군집 알고리즘이다
- 단점은 무작위 초기화를 사용하여 알고리즘의 출력이 무작위 수 초기값에 따라 달라진다
- 또 다른 단점은 클러스터의 모양을 가정하고 있어 활용 범위가 비교적 제한적이며, 찾으려하는 클러스터의 개수를 지정해야 한다는 점에 있다

요약

- 비음수행렬분해 non-negative matrix factorization, NMF
- t-SNE를 이용한 매니폴드 학습
- k-평균 군집