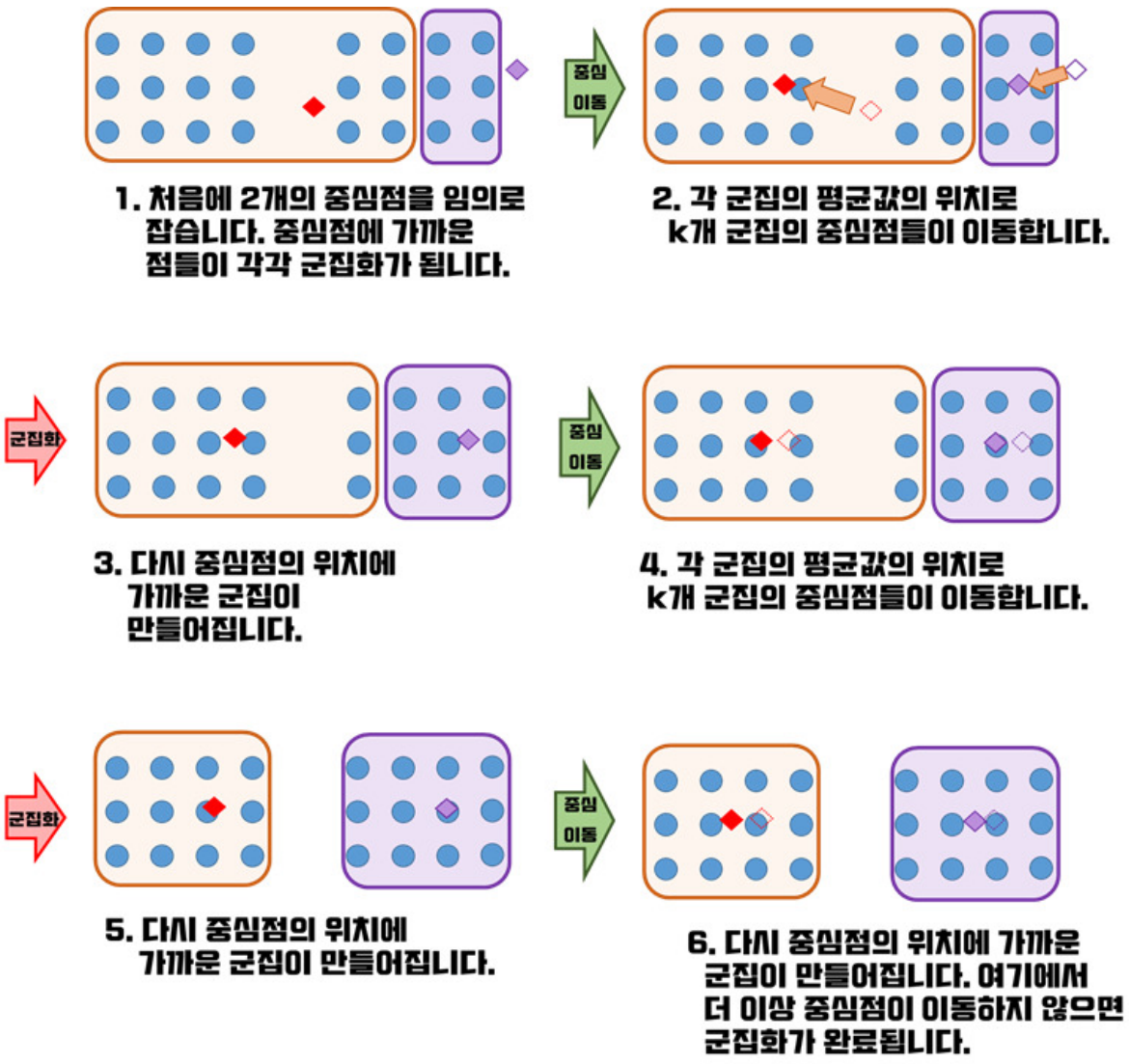
Chap7. 군집화

1. K-평균 알고리즘 이해

→ 군집 중심점(centroid)이라는 특정한 임의의 지점을 선택해 해당 중심에 가장 가까운 포인트를 선택하는 군집화 기법



centroid는 선택된 포인트의 평균 지점으로 이동하고 이동된 중심점에서 다시 가까운 포인트를 선택 및 다시 중심점을 평균 지점으로 이동하는 프로세스를 **반복적으로 수행** → 모든 데이터 포인트

에서 더이상 **중심점의 이동이 없을 경우에 반복을 멈추고 해당 중심점에 속하는 데이터 포인트들을 군집화**한다.

- K-평균의 장점

1. 일반적인 군집화에서 가장 많이 활용되는 알고리즘
2. 알고리즘이 쉽고 간결하다.

- K-평균의 단점

1. 거리 기반 알고리즘으로, 속성의 개수가 매우 많을 경우 군집화 정확도가 떨어짐(이를 위해 PCA로 차원 감소를 적용해야 할 수도 있음)
2. 반복을 수행하는데, 반복 횟수가 많을 경우 수행 시간이 매우 느려짐
3. 몇 개의 cluster(군집)을 선택해야 할 지 가이드하기 어려움

사이킷런 KMeans 클래스

```
from sklearn.cluster import KMeans
```

중요 파라미터

→ n_clusters(가장 중요한 파라미터) : 군집화할 개수, 즉 군집 중심점 centroid의 개수를 의미

→ init : 초기에 군집 중심점의 좌표를 설정할 방식, 보통 임의로 중심을 설정하지 않고 일반적으로 k-means++ 방식으로 최초 설정

→ max_iter : 최대 반복 횟수, 이 횟수 이전에 모든 데이터의 중심점 이동이 없으면 종료

다른 사이킷런 비지도학습 클래스와 마찬가지로 fit_transform(데이터세트) 메서드를 이용해 수행

→ 군집화 수행 완료, 군집화와 관련된 주요 속성 알 수 있음

→ labels_ : 각 데이터 포인트가 속한 군집 중심점 레이블

→ cluster_centers_ : 각 군집 중심점 좌표, 이를 이용해 군집 중심점 좌표가 어디인지 시각화 가능

K-평균을 이용한 붓꽃 데이터 세트 시각화

→ 붓꽃의 꽃받침(sepal)과 꽃잎(petal) 길이와 너비에 따른 품종 분류

⇒ 꽃받침, 꽃잎 길이에 따라 각 데이터의 군집화가 어떻게 결정되는지 확인해보고, 이를 분류 값과 비교해보자

```
from sklearn.preprocessing import scale
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

iris = load_iris()
#데이터프레임으로 변환
irisDF = pd.DataFrame(data=iris.data, columns=iris.feature_names)
irisDF.head(3)
```

→ 이를 3개의 cluster로 군집화해보자(n_cluster=3, init은 디폴트값인 k-means++, max_iter=300)

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=3, init='k-means++', max_iter=300, random_state=0)
kmeans.fit(irisDF)
print(kmeans.labels_)
```

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

→ fit()이후 labels_메서드를 이용해 irisDF의 각 데이터가 어떤 중심에 속하는지 알 수 있음

- 실제 붓꽃 품종 분류 값과 얼마나 차이가 나는지 확인(군집화의 효과성 검증)

```
irisDF['target'] = iris.target
irisDF['cluster'] = kmeans.labels_
iris_result = irisDF.groupby(['target', 'cluster'])['sepal length (cm)'].count()
print(iris_result)
```

```
target  cluster
0       1          50
1       0           2
        2          48
2       0          36
        2          14
Name: sepal length (cm), dtype: int64
```

→ 분류 타겟이 0인 데이터는 모두 1번 cluster로 잘 grouping됨

→ 분류 타겟이 1인 데이터는 2개만 0번 군집, 나머지 48개는 모두 2번 군집으로 grouping

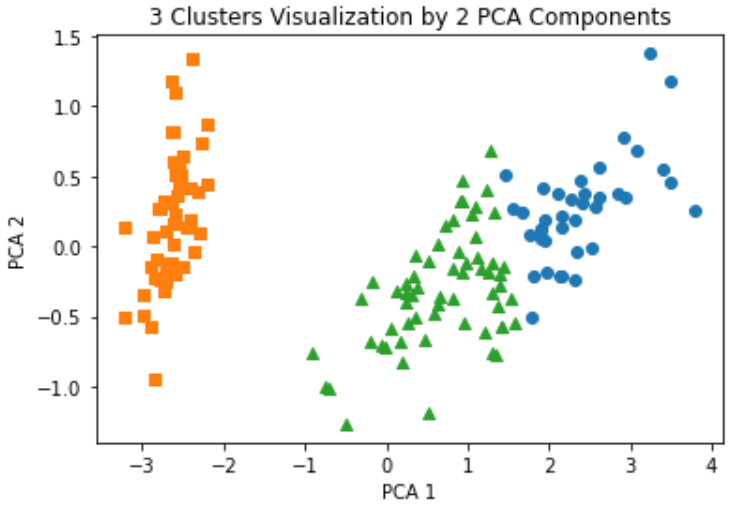
→ BUT 분류 타겟이 2인 데이터는 0번 군집에 36개, 2번 군집에 14개로 분산됨

- 붓꽃 데이터 세트의 군집화 시각화(PCA를 이용해 4개의 속성을 2개로 차원 축소)

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
pca_transformed = pca.fit_transform(iris.data)
irisDF['pca_x'] = pca_transformed[:,0]
irisDF['pca_y'] = pca_transformed[:,1]
irisDF.head(3)
```

→ 각 군집별로 cluster 0은 마커 'o', cluster 1은 마커 's', cluster 2는 마커'^'로 표현

```
#군집 값이 0,1,2인 경우마다 별도의 인덱스로 추출
marker0_ind = irisDF[irisDF['cluster']==0].index
marker1_ind = irisDF[irisDF['cluster']==1].index
marker2_ind = irisDF[irisDF['cluster']==2].index

#군집값 0,1,2에 해당하는 인덱스로 각 군집 레벨의 pca_x, pca_y 값 추출, o,s,^로 마커 표시
plt.scatter(x=irisDF.loc[marker0_ind, 'pca_x'], y=irisDF.loc[marker0_ind, 'pca_y'], marker='o')
plt.scatter(x=irisDF.loc[marker1_ind, 'pca_x'], y=irisDF.loc[marker1_ind, 'pca_y'], marker='s')
plt.scatter(x=irisDF.loc[marker2_ind, 'pca_x'], y=irisDF.loc[marker2_ind, 'pca_y'], marker='^')

plt.xlabel('PCA 1')
plt.ylabel('PCA 2')
plt.title('3 Clusters Visualization by 2 PCA Components')
plt.show()
```



→ cluster 1을 나타내는 네모(s)는 명확히 다른 군집과 잘 분리되어 있음

→ cluster 0을 나타내는 동그라미(o)와 cluster 2를 나타내는 세모(^)는 상당 수준 분리되어 있지만 네모만큼 명확히 분리되어 있진 않음

군집화 알고리즘 테스트를 위한 데이터 생성

✅**make_blobs(), make_classification()**

→ 다양한 유형의 군집화 알고리즘을 테스트해보기 위한 간단한 데이터 생성기

→ 여러 개의 클래스에 해당하는 데이터 세트 만들 수 있음, 하나의 클래스에 여러 개의 군집이 분포 될 수 있게 데이터 생성 가능

(둘이 비슷하긴 하지만 make_blobs()는 개별 군집의 중심점과 표준 편차 제어 기능 有, make_classification()은 노이즈를 포함한 데이터를 만드는데 유용하게 사용)

(make_circle(), make_moon()은 중심 기반 군집화로 해결하기 어려운 데이터 세트 만듦)

- make_blobs()의 호출 파라미터

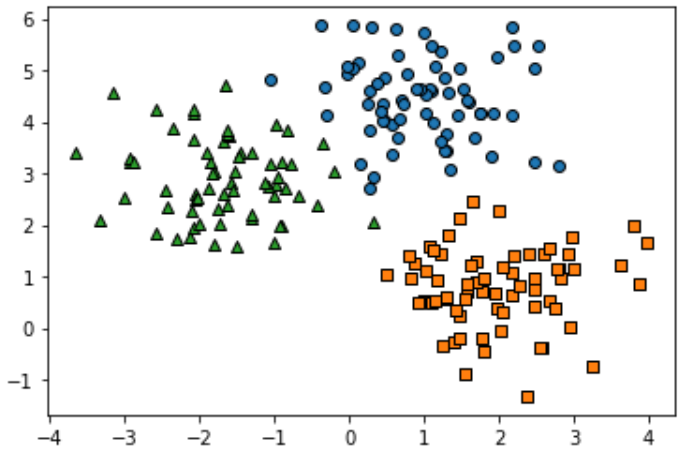
1. n_samples : 생성할 총 데이터의 개수, default=100
2. n_features : 데이터의 피처 개수, 시각화가 목표이면 2개로 설정해 보통 첫번째 피처는 x좌표, 두번째 피처는 y좌표상에 표현
3. centers : int 값, 3으로 설정하면 군집의 개수를 나타냄, ndarray 형태로 표현할 경우 개별 군집 중심점의 좌표를 의미
4. cluster_std : 생성될 군집 데이터의 표준 편자, float값(0.8) 형태로 지정하면 군집 내에서 데이터가 표준편차 0.8을 가진 값으로 만들어짐, [0.8, 1.2, 0.6]과 같은 형태로 지정하면 군집별로 서로 다른 표준 편차를 가진 데이터 세트가 만들어짐 (cluster_std가 작을수록 군집 중심에 데이터가 모여 있고, 클수록 데이터가 퍼져 있음)

X, y = make_blobs(n_samples=200, n_features=2, center=3, random_state=0)을 호출하면 총 200개의 레코드와 2개의 피처가 3개의 군집화 기반 분포도를 가진 피처 데이터 세트 X와 동시에 3개의 군집화 값을 가진 타겟 데이터 세트 y가 반환됨

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=200, n_features=2, centers=3, cluster_std=0.8, random_state=0)
print(X.shape, y.shape)
#y 타겟값의 분포 확인
unique, counts = np.unique(y, return_counts=True)
print(unique, counts)
#DataFrame으로 변경
clusterDF = pd.DataFrame(data=X, columns=['ftr1', 'ftr2'])
```

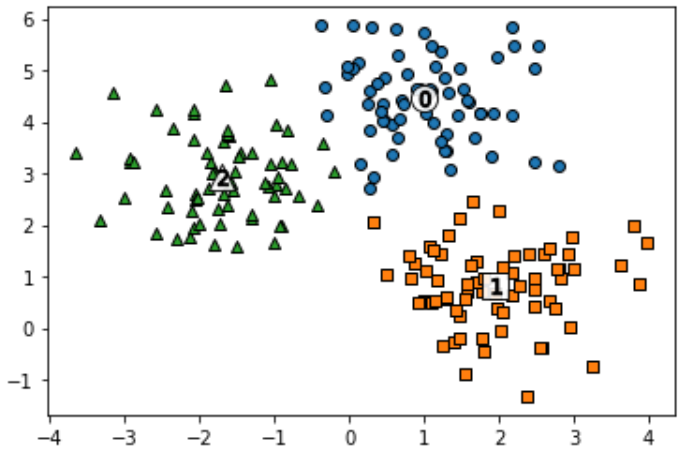
```
clusterDF['target'] = y
clusterDF.head(3)
```

```
(200, 2) (200,)
[0 1 2] [67 67 66]

ftr1  ftr2  target
0 -1.692427 3.622025  2
1 0.697940  4.428867  0
2 1.100228  4.606317  0
```



- KMeans 군집화 수행 후 군집별 시각화

```
#kmeans 객체를 이용해 X 데이터 클러스터링 수행
kmeans = KMeans(n_clusters=3, init='k-means++', max_iter=200, random_state=0)
cluster_labels = kmeans.fit_predict(X)
clusterDF['kmeans_label'] = cluster_labels

#cluster_centers_는 개별 클러스터의 중심 위치 좌표 시각화를 위해 추출
centers = kmeans.cluster_centers_
unique_labels = np.unique(cluster_labels)
markers=['o', 's', '^', 'P','D','H','x']

# 군집된 label 유형별로 iteration 하면서 marker 별로 scatter plot 수행.
for label in unique_labels:
    label_cluster = clusterDF[clusterDF['kmeans_label']==label]
    center_x_y = centers[label]
    plt.scatter(x=label_cluster['ftr1'], y=label_cluster['ftr2'], edgecolor='k',
                marker=markers[label] )

    # 군집별 중심 위치 좌표 시각화
    plt.scatter(x=center_x_y[0], y=center_x_y[1], s=200, color='white',
                alpha=0.9, edgecolor='k', marker=markers[label])
    plt.scatter(x=center_x_y[0], y=center_x_y[1], s=70, color='k', edgecolor='k',
                marker='$%d$' % label)

plt.show()
```




```
print(clusterDF.groupby('target')['kmeans_label'].value_counts())
```

```
target  kmeans_label
0       0               66
        2                1
1       1               67
2       2               65
        1                1
Name: kmeans_label, dtype: int64
```

⇒ 대부분 잘 grouping 되어짐

2. 군집 평가(Cluster Evaluation)

대부분의 군집화 데이터 세트는 비교할만한 타깃 레이블 X

분류(Classification)과 유사해보이나 성격이 다르다 → 데이터 내에 숨어있는 별도의 그룹을 찾아서 의미를 부여하거나 동일한 분류 값에 속하더라도 그 안에서 더 세분화된 군집화를 추구하거나 서로 다른 분류 값의 데이터도 더 넓은 군집화 레벨화 등의 영역을 가지고 있음

비지도 학습의 특성상 어떠한 지표라도 정확하게 성능을 평가하기 어려움

⇒ 실루엣 분석을 이용하자!

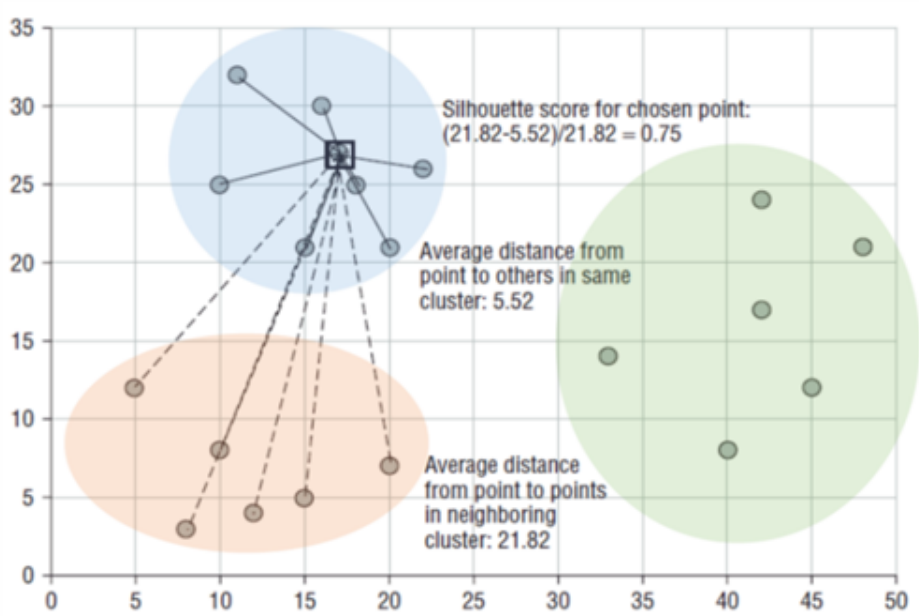
실루엣 분석의 개요

→ 각 군집 간의 거리가 얼마나 효율적으로 분리돼 있는지를 나타냄

→ 효율적인 분리 = 다른 군집과의 거리는 떨어져있고 동일 군집끼리의 데이터는 서로 가깝게 뭉쳐 있음, 군집화가 잘 될수록 개별 군집은 비슷한 정도의 여유공간을 가지고 떨어져 있을 것

- 실루엣 계수(silhouette coefficient) : 개별 데이터가 가지는 군집화 지표

⇒ 해당 데이터가 같은 군집 내의 데이터와 얼마나 가깝게 군집화되어 있고, 다른 군집에 있는 데이터와는 얼마나 멀리 분리되어 있는지를 나타내는 지표



→ 특정 데이터 포인트의 실루엣 계수 값은 해당 데이터 포인트와 같은 군집 내에 있는 다른 데이터 포인트와의 거리를 평균한 값 a(i), 해당 데이터 포인트가 속하지 않은 군집 중 가장 가까운 군집과의 평균 거리 b(i)를 기반으로 계산 됨

$$s(i) = \frac{(b(i) - a(i))}{(\max(a(i), b(i))}$$

→ b(i) - a(i) : 두 군집 간의 거리가 얼마나 떨어져 있는가를 나타냄

→ 위 값을 정규화 하기 위해 max(a(i), b(i))로 나눠줌

→ 실루엣 계수 s(i)는 -1~1 사이의 값을 가짐, 1로 가까워질수록 근처의 군집과 더 멀리 떨어져 있다는 것이고, 0에 가까우루록 근처의 군집과 가까워진다는 뜻, 음수 값은 아예 다른 군집에 데이터 포인트가 할당됐음을 의미

1. sklearn.metrics.silhouette_samples(X, labels, metric='euclidean', **kwds)

→ 인자로 X feature 데이터 세트와 각 피처 데이터 세트가 속한 군집 레이블 값인 labels 데이터 입력

⇒ 각 데이터 포인트의 실루엣 계수를 계산해 반환해줌

2. sklearn.metrics.silhouette_score(X, labels, metric='euclidean', sample_size=None, **kwds)

→ 인자로 X feature 데이터 세트와 각 피처 데이터 세트가 속한 군집 레이블 값인 labels 데이터 입력

⇒ 전체 데이터의 실루엣 계수 값을 평균해 반환 (= np.mean(silhouette_samples())

⇒ 일반적으로 이 값으로 높을수록 군집화가 어느정도 잘 되었다고 판단 가능 BUT 무조건 이 값이 높다고 해서 군집화가 잘 되었다고 판단하면 안됨

- 좋은 군집화의 조건

1. 전체 실루엣 계수의 평균값, 즉 사이킷런의 silhouette_score() 값은 0~1사이의 값을 가지며, 1에 가까울수록 좋음
2. 하지만 전체 실루엣 계수의 평균값과 더불어 개별 군집의 평균값의 편차가 크지 않아야 함

→ 즉, 개별 군집의 실루엣 계수 평균값이 전체 실루엣 계수의 평균값에서 크게 벗어나지 않는 것이 중요!

(만약 전체 실루엣 계수의 평균값은 높지만, 특정 군집의 실루엣 계수 평균값만 유난히 높고 다른 군집들의 실루엣 계수 평균값은 낮으면 좋은 군집화 X)

붓꽃 데이터 세트를 이용한 군집 평가

```
from sklearn.preprocessing import scale
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans

#실루엣 분석 평가 지표 값을 구하기 위한 API 추가
from sklearn.metrics import silhouette_samples, silhouette_score
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

iris = load_iris()
feature_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
irisDF = pd.DataFrame(data=iris.data, columns=feature_names)
kmeans = KMeans(n_clusters=3, init='k-means++', max_iter=300, random_state=0).fit(irisDF)
irisDF['cluster'] = kmeans.labels_

#iris의 모든 개별 데이터의 실루엣 계수 값 구하기
score_samples = silhouette_samples(iris.data, irisDF['cluster'])
print('silhouette_samples() return 값의 shape', score_samples.shape)
```

```
#irisDF에 실루엣 계수 칼럼 추가
irisDF['silhouette_coeff'] = score_samples

#모든 데이터의 평균 실루엣 계수 값 구하기
average_score = silhouette_score(iris.data, irisDF['cluster'])
print('붓꽃 데이터 세트 Silhouette Analysis Score:', average_score)
irisDF.head(3)
```

```
silhouette_samples() return 값의 shape (150,)
붓꽃 데이터 세트 Silhouette Analysis Score: 0.5528190123564091
```

	sepal_length	sepal_width	petal_length	petal_width	cluster	silhouette_coeff
0	5.1	3.5	1.4	0.2	1	0.852955
1	4.9	3.0	1.4	0.2	1	0.815495
2	4.7	3.2	1.3	0.2	1	0.829315

➕군집별 평균 실루엣 계수 구하기

```
irisDF.groupby('cluster')['silhouette_coeff'].mean()
```

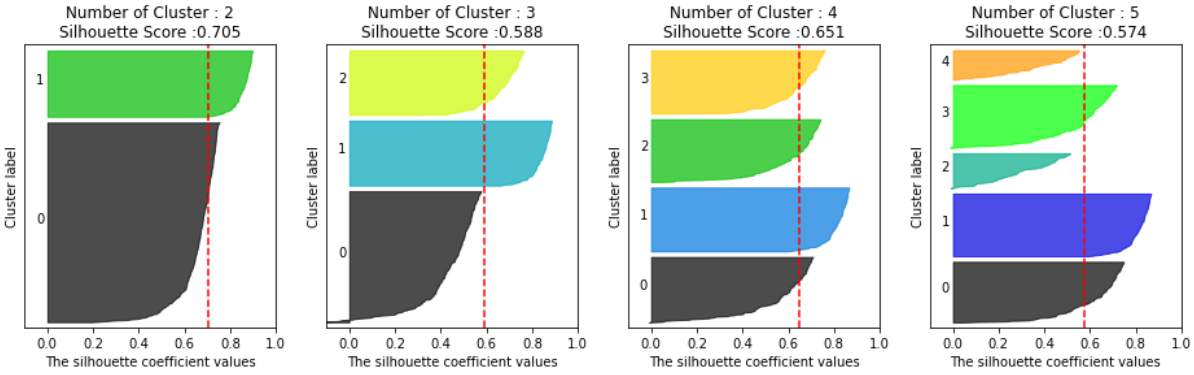
```
cluster
0    0.451105
1    0.798140
2    0.417320
Name: silhouette_coeff, dtype: float64
```

⇒ 0번, 2번 군집의 실루엣 계수 값이 1번 군집에 비해 낮음

군집별 평균 실루엣 계수의 시각화를 통한 군집 개수 최적화 방법

```
### 여러개의 클러스터링 갯수를 List로 입력 받아 각각의 실루엣 계수를 면적으로 시각화한 함수 작성
def visualize_silhouette(cluster_lists, X_features):

    from sklearn.datasets import make_blobs
    from sklearn.cluster import KMeans
    from sklearn.metrics import silhouette_samples, silhouette_score

    import matplotlib.pyplot as plt
    import matplotlib.cm as cm
    import math

    # 입력값으로 클러스터링 갯수들을 리스트로 받아서, 각 갯수별로 클러스터링을 적용하고 실루엣 개수를 구함
    n_cols = len(cluster_lists)
```

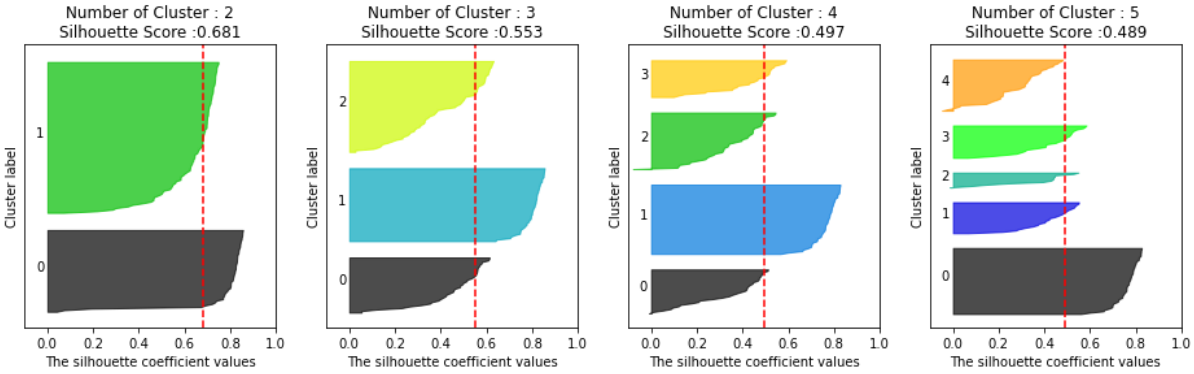
```
    # plt.subplots()으로 리스트에 기재된 클러스터링 수만큼의 sub figures를 가지는 axs 생성
    fig, axs = plt.subplots(figsize=(4*n_cols, 4), nrows=1, ncols=n_cols)

    # 리스트에 기재된 클러스터링 갯수들을 차례로 iteration 수행하면서 실루엣 개수 시각화
    for ind, n_cluster in enumerate(cluster_lists):

        # KMeans 클러스터링 수행하고, 실루엣 스코어와 개별 데이터의 실루엣 값 계산.
        clusterer = KMeans(n_clusters = n_cluster, max_iter=500, random_state=0)
        cluster_labels = clusterer.fit_predict(X_features)

        sil_avg = silhouette_score(X_features, cluster_labels)
        sil_values = silhouette_samples(X_features, cluster_labels)

        y_lower = 10
        axs[ind].set_title('Number of Cluster : '+ str(n_cluster)+'\n' \
                          'Silhouette Score :' + str(round(sil_avg,3)) )
        axs[ind].set_xlabel("The silhouette coefficient values")
        axs[ind].set_ylabel("Cluster label")
        axs[ind].set_xlim([-0.1, 1])
        axs[ind].set_ylim([0, len(X_features) + (n_cluster + 1) * 10])
        axs[ind].set_yticks([])  # Clear the yaxis labels / ticks
        axs[ind].set_xticks([0, 0.2, 0.4, 0.6, 0.8, 1])

        # 클러스터링 갯수별로 fill_betweenx( )형태의 막대 그래프 표현.
        for i in range(n_cluster):
            ith_cluster_sil_values = sil_values[cluster_labels==i]
            ith_cluster_sil_values.sort()

            size_cluster_i = ith_cluster_sil_values.shape[0]
            y_upper = y_lower + size_cluster_i

            color = cm.nipy_spectral(float(i) / n_cluster)
            axs[ind].fill_betweenx(np.arange(y_lower, y_upper), 0, ith_cluster_sil_values, \
                                   facecolor=color, edgecolor=color, alpha=0.7)
            axs[ind].text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))
            y_lower = y_upper + 10

        axs[ind].axvline(x=sil_avg, color="red", linestyle="--")
# make_blobs 을 통해 clustering 을 위한 4개의 클러스터 중심의 500개 2차원 데이터 셋 생성
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=500, n_features=2, centers=4, cluster_std=1, \
                  center_box=(-10.0, 10.0), shuffle=True, random_state=1)

# cluster 개수를 2개, 3개, 4개, 5개 일때의 클러스터별 실루엣 계수 평균값을 시각화
visualize_silhouette([ 2, 3, 4, 5], X)
```



⇒ 4개의 군집일 때 가장 최적이 됨

➕붓꽃 데이터를 이용해 K-평균 수행 시 최적의 군집 개수

```
visualize_silhouette([2,3,4,5], iris.data)
```



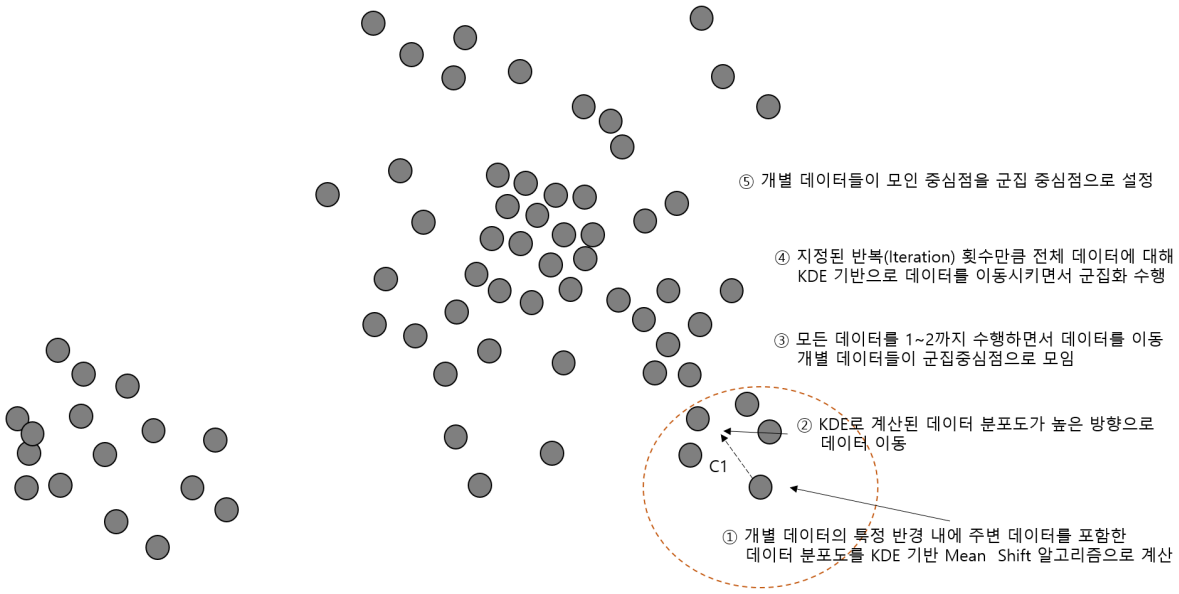
⇒ 2개의 군집일 때 가장 좋아보임

실루엣 게수를 통한 k-평균 군집 평가 방법은 직관적으로 이해하긴 쉽지만, 각 데이터별로 다른 데이터와의 거리를 반복적으로 계산해야 하므로 데이터 양이 늘어나면 수행시간이 크게 늘어남

→ **군집별로 임의의 데이터를 샘플링**해 실루엣 계수를 평가하는 방안 고려

3. 평균 이동(Mean Shift)

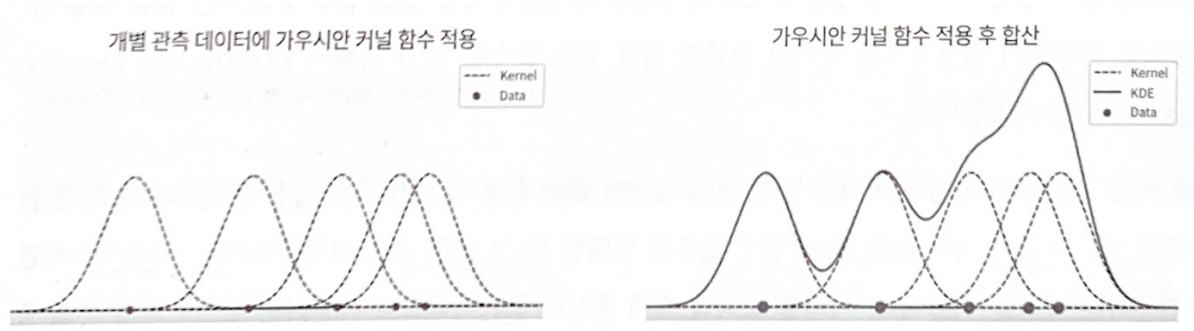
→ K-평균은 중심에 소속된 데이터의 평균 거리 중심으로 이동 VS **평균 이동은 중심을 데이터가 모여 있는 밀도가 가장 높은 곳으로 이동**



→ 데이터를 반경 내의 데이터 분포 확률 밀도가 가장 높은 곳으로 이동하기 위해 주변 데이터와의 거리 값을 KDE(Kernel Density Estimation) 함수 값으로 입력한 뒤 그 반환 값을 현재 위치에서 업데이트 하면서 이동

→ 지정된 반복(Iteration) 횟수만큼 전체 데이터에 대해서 KDE 기반으로 데이터를 이동시키면서 군집화 시킨 뒤, 집중적으로 데이터가 모여있어 확률 밀도 함수(probability denstiy function)가 피크인 점을 군집 중심점으로 선정

(KDE는 커널 함수를 통해 어떤 변수의 PDF를 추정하는 대표적인 방법)



➕KDE의 커널 함수식과 h

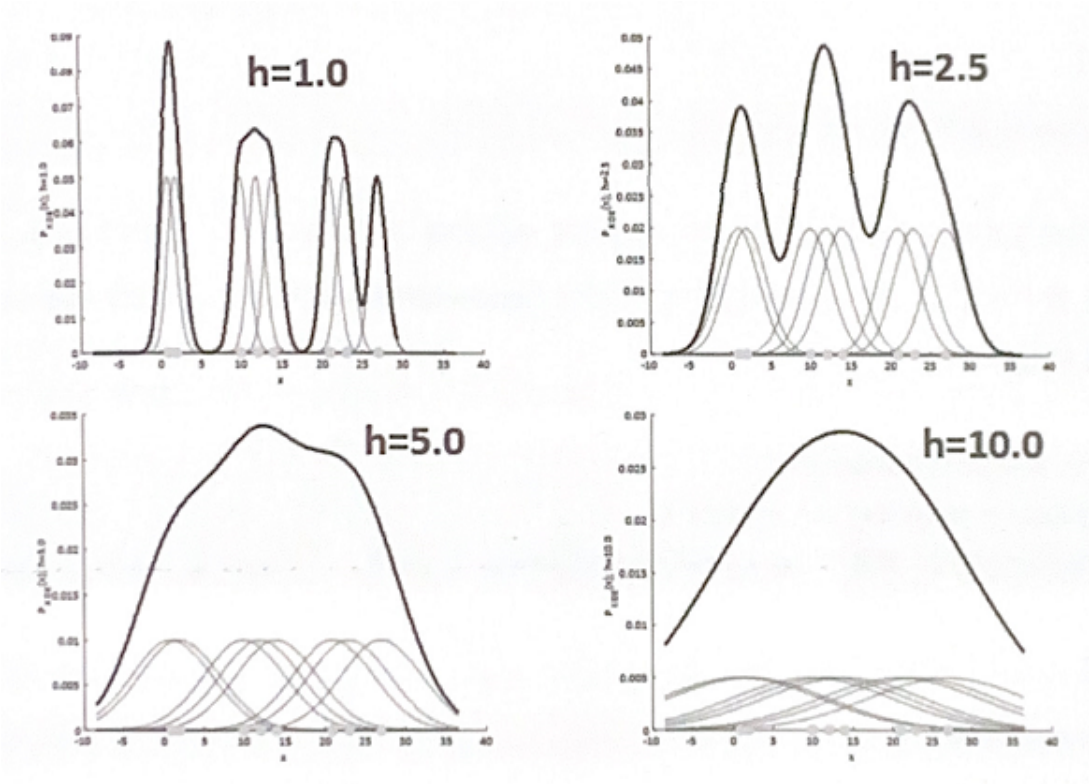
$$KDE = \frac{1}{n}\sum_{i=1}^{n} K_h(x - x_i) = \frac{1}{nh}\sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

→ 대역폭 h는 KDE 형태를 부드러운(또는 뾰족한) 형태로 평활화(Smoothing)하는데 적용

→ 작은 h는 좁고 뾰족한 KDE(과적합 위험 有)

→ 큰 h는 과도하게 평활화된 KDE로 인해 지나치게 단순화된 방식으로 PDF 추정(과소적합)

⇒ 대역폭 h를 계산하는 것은 KDE 기반의 평균 이동 군집화에서 매우 중요!



→ 대역폭 h가 클수록 적은 수의 군집 중심점, h가 작을수록 많은 수의 군집 중심점을 가짐

⇒ 평균 이동 군집화는 군집 개수 지정 X, 오직 대역폭의 크기에 따라 군집화 수행!

- 사이킷런 MeanShift 클래스

→ 중요 파라미터 : bandwidth(KDE의 대역폭 h와 동일)

→ 최적의 대역폭 계산을 위해 estimate_bandwidth()함수 제공

```
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.cluster import MeanShift

X, y = make_blobs(n_samples=200, n_features=2, centers=3, cluster_std=0.7, random_state=0)

#bandwidth를 0.8로 지정
meanshift = MeanShift(bandwidth=0.8)
cluster_labels = meanshift.fit_predict(X)
print('cluster labels 유형:', np.unique(cluster_labels))
```

```
cluster labels 유형: [0 1 2 3 4 5]
```

⇒ 군집이 0부터 5까지 6개로 분류(지나치게 세분화됨)

일반적으로 bandwidth값을 작게 할수록 군집 개수가 많아짐

```
#bandwidth를 1로 지정
meanshift = MeanShift(bandwidth=1)
cluster_labels = meanshift.fit_predict(X)
print('cluster labels 유형:', np.unique(cluster_labels))
```

```
cluster labels 유형: [0 1 2]
```

⇒ 3개의 군집으로 잘 군집화됨

- 최적화된 bandwidth 값 찾기(estimate_bandwidth())

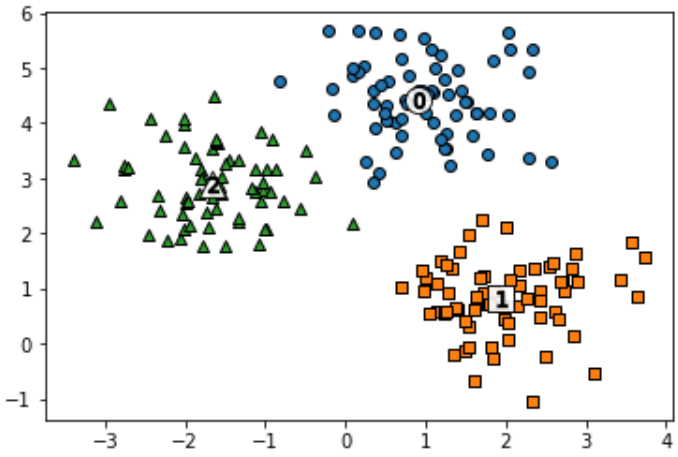
```
from sklearn.cluster import estimate_bandwidth

bandwidth = estimate_bandwidth(X)
print('bandwidth 값:', round(bandwidth, 3))
```

```
bandwidth 값: 1.816
```

→ 이를 이용해 다시 make_blobs()데이터 세트에 군집화 수행

```
import pandas as pd

clusterDF = pd.DataFrame(data=X, columns=['ftr1', 'ftr2'])
clusterDF['target'] = y

#estimate_bandwidth()로 최적의 bandwidth 계산
best_bandwidth = estimate_bandwidth(X)

meanshift = MeanShift(bandwidth=best_bandwidth)
cluster_labels = meanshift.fit_predict(X)
print('cluster labels 유형:', np.unique(cluster_labels))
```

```
cluster labels 유형: [0 1 2]
```




```
#타겟값과 군집 label값 비교
print(clusterDF.groupby('target')['meanshift_label'].value_counts())
```

```
target  meanshift_label
0       0                 67
1       1                 67
2       2                 66
Name: meanshift_label, dtype: int64
```

→ 1:1로 잘 매칭됨

🤩MeanShift의 장점

→ 데이터 세트의 형태를 특정 형태로 가정한다든가, 특정 분포도 기반의 모델로 가정하지 않기 때문에 좀 더 유연한 군집화 가능

→ 이상치의 영향력 크지 않음, 미리 군집의 개수를 정할 필요 X

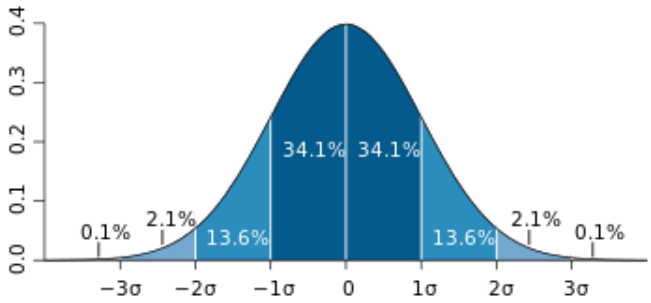
BUT 알고리즘의 수행 시간이 오래 걸림, bandwidth의 크기에 따른 군집화 영향도 매우 큼

⇒ 분석 업무보다는 컴퓨터 비전 영역에서 더 많이 사용됨

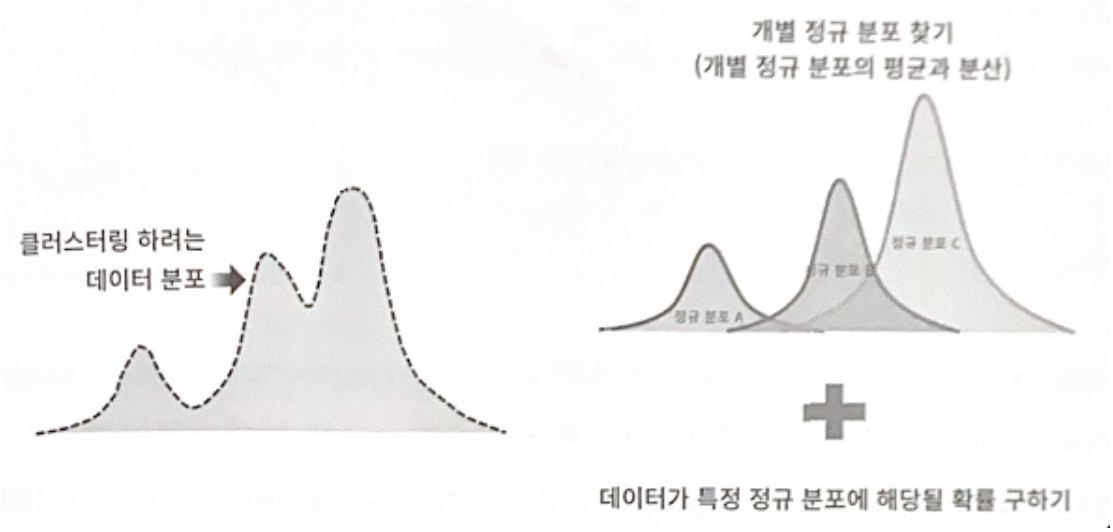
4. GMM(Gaussian Mixture Model)

GMM 소개

→ 군집화를 적용하고자 하는 데이터가 여러 개의 가우시안 분포(GaussianDistribution)를 갖는 데이터 집합들이 섞여 생성된 것이라는 가정하에 군집화를 수행하는 방식



GMM은 섞인 데이터 분포에서 개별 유형의 가우시안 분포를 추출함



전체 데이터 세트는 서로 다른 정규 분포 형태를 가진 여러 가지 확률 분포 곡선으로 구성

→ **이러한 서로 다른 정규 분포에 기반해 군집화를 수행하는 것이 GMM 군집화 방식**

- GMM의 모수추정

1. 개별 정규 분포의 평균과 분산
2. 각 데이터가 어떤 정규 분포에 해당되는지의 확률

⇒ 이를 위해 EM(Expectation and Maximization) 방법 적용

GMM을 이용한 붓꽃 데이터 세트 군집화

GMM은 확률 기반 군집화, K-Means는 거리 기반 군집화

```
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
```

```
iris = load_iris()
irisDF = pd.DataFrame(data=iris.data, columns=feature_names)
irisDF['target'] = iris.target
```

GaussianMixture 객체의 가장 중요한 초기화 파라미터 : **n_components**

→ **gaussian mixture의 모델의 총 개수**(K-Means의 n_clusters와 같이 **군집의 개수를 정하는 데** 중요한 역할 수행)

```
gmm = GaussianMixture(n_components=3, random_state=0).fit(iris.data)
gmm_cluster_labels = gmm.predict(iris.data)

#군집화 결과를 irisDF의 'gmm_cluster'칼럼명으로 저장
irisDF['gmm_cluster'] = gmm_cluster_labels

#target값에 따라 gmm_cluster 값이 어떻게 매핑되었는지 확인
iris_result = irisDF.groupby('target')['gmm_cluster'].value_counts()
print(iris_result)
```

```
1       1              45
        2               5
2       2              50
Name: gmm_cluster, dtype: int64
```

⇒ K-Means 군집화 결과보다 더 효과적인 분류 결과 도출

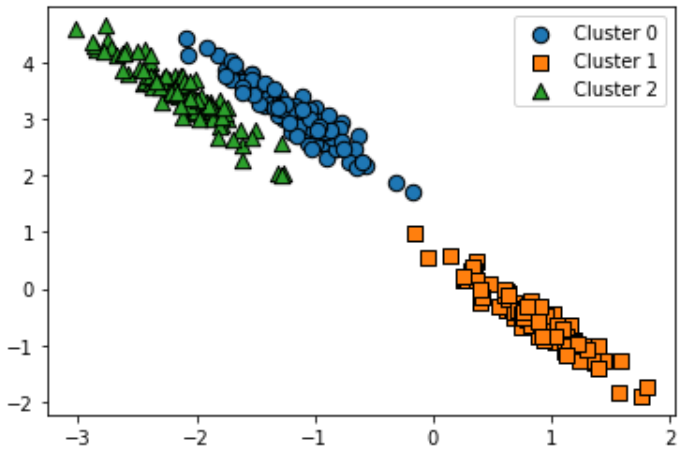
```
target  cluster
0       1          50
1       0           2
        2          48
2       0          36
        2          14
Name: sepal length (cm), dtype: int64
```

⇒ 붓꽃 데이터 세트엔 k-means보단 GMM이 더 효과적

→ 개별 군집 내의 데이터가 원형으로 흩어져 있는 경우에 K-Means가 효과적으로 군집화 수행 가능

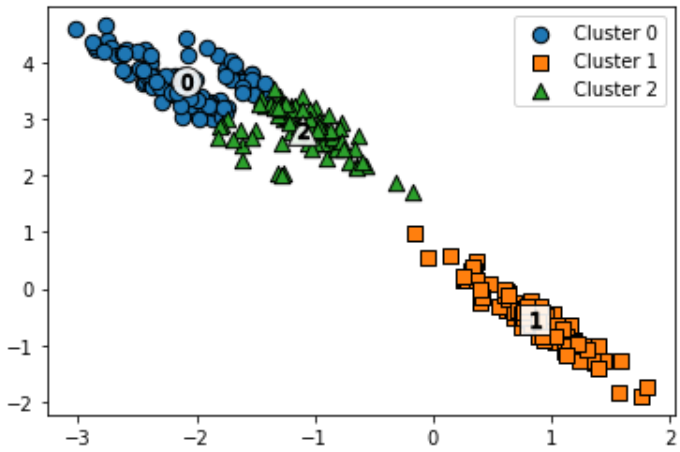
GMM과 K-평균 비교

k-means는 데이터가 원형으로 퍼져 있지 않은 경우에는 군집화 잘 수행 X(길쭉한 타원형일 경우)



이 데이터 세트에 K-Means를 적용해보자

```
from sklearn.cluster import KMeans

#3개 군집 기반 KMeans를 X_aniso 데이터 세트에 적용
kmeans = KMeans(3, random_state=0)
kmeans_label = kmeans.fit_predict(X_aniso)
clusterDF['kmeans_label'] = kmeans_label

visualize_cluster_plot(kmeans, clusterDF, 'kmeans_label', iscenter=True)
```

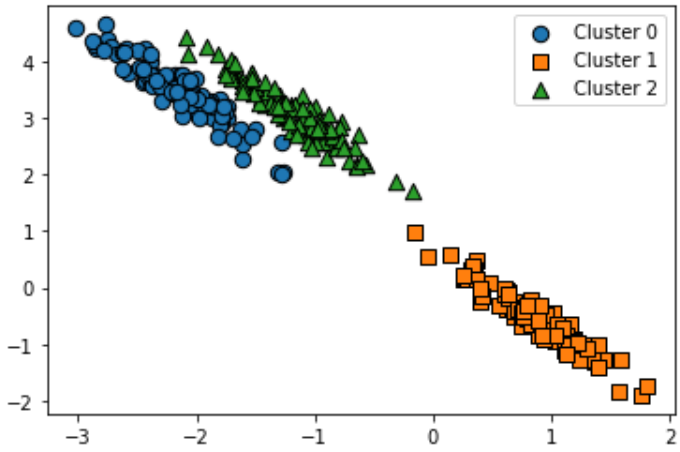


⇒ 주로 원형 영역 위치로 개별 군집화 수행, 원하는 방향으로 군집이 구성되지 않음

- GMM으로 군집화 수행

```
from sklearn.mixture import GaussianMixture

#3개의 n_components 기반 GMM을 X_aniso 데이터 세트에 적용
gmm = GaussianMixture(n_components=3, random_state=0)
gmm_label = gmm.fit(X_aniso).predict(X_aniso)
clusterDF['gmm_label'] = gmm_label
```

```
#GaussianMixture는 cluster_centers_ 속성이 없으므로 iscenter를 False로 설정
visualize_cluster_plot(gmm, clusterDF, 'gmm_label', iscenter=False)
```



⇒ 데이터가 분포된 방향에 따라 정확하게 군집화가 되었다!

- K-Means와 GMM의 군집화 효율 차이

```
print('### KMeans Clustering ###')
print(clusterDF.groupby('target')['kmeans_label'].value_counts())
print('\n### Gaussian Mixture Clustering ###')
print(clusterDF.groupby('target')['gmm_label'].value_counts())
```

```
### KMeans Clustering ###
target  kmeans_label
0       2               73
        0               27
1       1              100
2       0               86
        2               14
Name: kmeans_label, dtype: int64

### Gaussian Mixture Clustering ###
target  gmm_label
0       2            100
1       1            100
2       0            100
Name: gmm_label, dtype: int64
```