




7장. 군집화



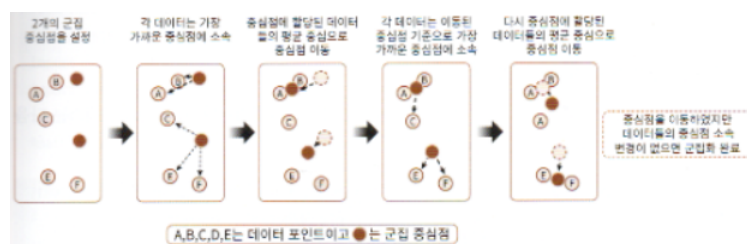
분류와 유사해보일 수 있지만 성격이 다르다. 데이터 내에 숨어있는 별도의 그룹을 찾아서 의미를 부여하거나, 동일한 분류값에 속하더라도 그 안에서 더 세분화된 군집화를 추구하거나, 서로 다른 분류값의 데이터도 더 넓은 군집화 레벨화 등의 영역을 가진다.

1 K-평균 알고리즘 이해

 군집 중심점(centroid)이라는 특정한 임의의 지점을 선택해 해당 중심에 가장 가까운 포인트들을 선택해 군집화하는 방식

기본 원리

1. 2개의 군집 중심점 설정
2. 각 데이터는 가장 가까운 중심점에 소속
3. 중심점에 할당된 데이터들의 평균 중심으로 중심점 이동
4. 각 데이터는 이동된 중심점 기준으로 가장 가까운 중심점에 소속
5. 다시 중심점에 할당된 데이터들의 평균 중심으로 중심점 이동
6. 중심점을 이동했지만 데이터들의 중심점 소속 변경이 없으면 군집화 완료



K-평균의 장점

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

K-평균의 단점

1. 거리 기반 알고리즘으로, 속성의 개수가 매우 많을 경우 군집화 정확도가 떨어짐

(이를 위해 PCA로 차원 감소를 적용해야 할 수도 있음)

1. 반복을 수행하기 때문에 반복 횟수가 많을 경우 수행 시간이 매우 느려짐
2. 몇 개의 군집을 선택해야 할 지 모를 때도 있음

1-1. 사이킷런 KMeans 클래스 소개

하이퍼 파라미터	설명
n_clusters	군집화할 개수(군집 중심점의 개수)
init	초기에 군집 중심점의 좌표를 설정할 방식. 보통은 임의로 설정하지 않고 K-Means++ 방식으로 설정

- 임의로 설정하고 싶으면 init='random'
- K-means++ 방식
- `max_iter` : 최대 반복 횟수. 이 횟수 이전에 모든 데이터의 중심점 이동이 없으면 종료 |
- 속성
 - labels_ : 각 데이터 포인트가 속한 군집중심점 레이블
 - clustercenters : 각 군집 중심점 좌표(shape=[군집개수, 피쳐개수]). 이를 이용해 시각화 가능

1-2. 군집화 알고리즘 테스트를 위한 데이터 생성

- 사이킷런의 데이터 생성기: 여러 개의 클래스에 해당하는 데이터 세트를 만드는데, 하나의 클래스에 여러 개의 군집이 분포될 수 있게 데이터를 생성한다.
 - `make_blobs()` : 개별 군집의 중심점과 표준 편차 제어 기능이 추가되어 있다. 피쳐 데이터 세트, 타깃 데이터 세트가 튜플로 반환

파라미터	설명
n_samples	디폴트 = 100
생성할 총 데이터의 개수	
n_features	데이터의 피쳐 개수
centers	int로 입력: 군집의 개수
ndarray로 입력: 개별 군집 중심점의 좌표	
cluster_std	생성될 군집 데이터의 표준편차

float로 입력: 군집 내 데이터의 표준 편차

[float, ...]로 입력: 각 군집의 순서대로 각각의 표준편차가 만들어짐.

⇒ 군집별로 서로 다른 표준편차를 가진 데이터 세트를 만들 때 사용 |

- `make_classification()`: 노이즈를 포함한 데이터를 만든다.
- `make_circle()`, `make_moon()`: 중심기반의 군집화로 해결하기 어려운 데이터 세트를 만듦

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

# 테스트 데이터 생성
X, y = make_blobs(n_samples=200, n_features=2, centers=3, cluster_std=0.8, random_state=0)
print(X.shape, y.shape)

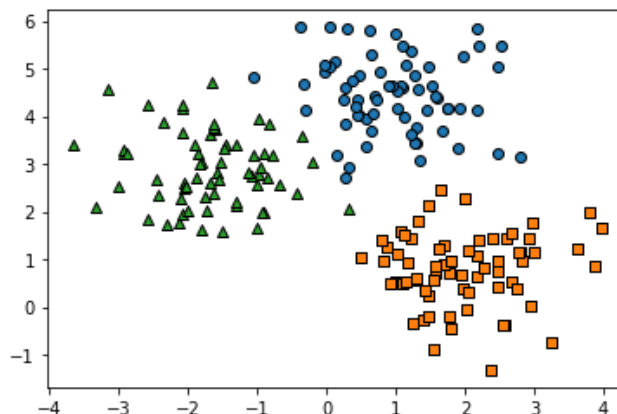
# y target 값의 분포를 확인
unique, counts = np.unique(y, return_counts=True)
print(unique, counts)

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

# DataFrame에 적용
import pandas as pd
clusterDF = pd.DataFrame(data=X, columns=['ftr1', 'ftr2'])
clusterDF['target'] = y

target_list = np.unique(y)
# 각 target별 scatter plot 의 marker 값들.
markers=['o', 's', '^', 'P', 'D', 'H', 'x']
# 3개의 cluster 영역으로 구분한 데이터 셋을 생성했으므로 target_list는 [0,1,2]
# target==0, target==1, target==2 로 scatter plot을 marker별로 생성.
for target in target_list:
    target_cluster = clusterDF[clusterDF['target']==target]
    plt.scatter(x=target_cluster['ftr1'], y=target_cluster['ftr2'], edgecolor='k', marker=markers[target] )

plt.show()
```



- KMeans 객체를 이용하여 X 데이터를 K-Means 클러스터링 수행 후, 시각화

```
# KMeans 객체를 이용하여 X 데이터를 K-Means 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']

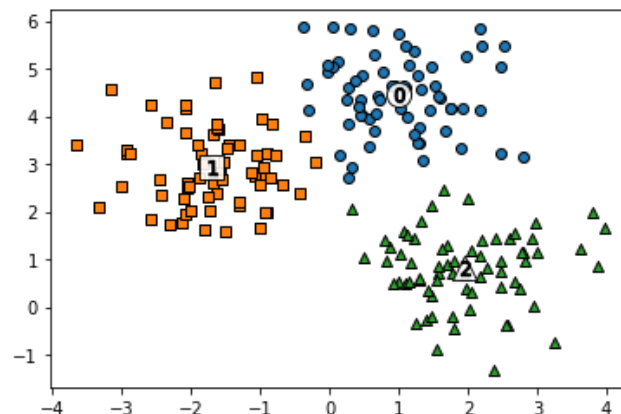
# 군집된 label 유형별로 iteration 하면서 marker 별로 scatter plot 수행.
for label in unique_labels:
    label_cluster = clusterDF[clusterDF['kmeans_label']==label]
    center_x_y = centers[label]
    plt.scatter(x=label_cluster['ftr1'], y=label_cluster['ftr2'], edgecolor='k',
                marker=markers[label]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

    # 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
>      1              1
> 1      2             67
> 2      1             65
>      2              1
```

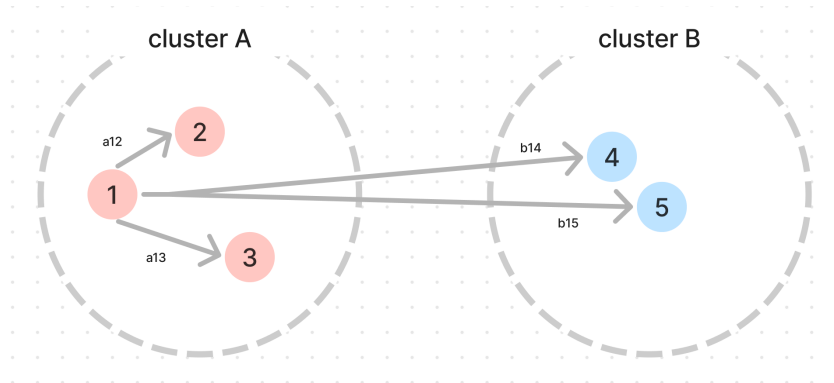


2 군집 평가 Cluster Evaluation - 실루엣 분석

: 대부분의 군집화 데이터 세트는 타깃 레이블을 가지고 있지 않다.

: 그래서 비지도 학습의 특성상 정확한 성능 평가는 어렵지만 군집화의 성능을 평가하는 방법으로는 실루엣 분석이 있다.

- 실루엣 분석: 각 군집 간의 거리가 얼마나 효율적으로 분리되어 있는지 나타냄
 - 효율적 분리 \Rightarrow 다른 군집과는 떨어져 있고, 동일 군집끼리의 데이터는 서로 가깝게 잘 뭉쳐 있는 것.
 - 군집화가 잘 될수록 개별 군집은 비슷한 정도의 여유공간을 가지고 떨어져 있다.
- 실루엣 계수(silhouette coefficient) : 개별 데이터가 가지는 군집화 지표
 - 해당 데이터가 같은 군집 내의 데이터와 얼마나 가깝게 군집화 돼있고, 다른 군집에 있는 데이터와는 얼마나 멀리 분리되어 있는지 나타내는 지표



- a_{ij} : i 번째 데이터에서 [자신이 속한 클러스터 내]의 [다른 데이터 포인트]까지의 거리
- $a(i)$: i 번째 데이터에서 [자신이 속한 클러스터 내]의 [다른 데이터 포인트]들의 [평균] 거리 $\Rightarrow a(1) = \text{avg}(a_{12}, a_{13} \dots)$
- $b(i)$: i 번째 데이터에서 [가장 가까운 타 클러스터 내]의 [다른 데이터 포인트]들의 [평균] 거리 $\Rightarrow b(1) = \text{avg}(b_{14}, b_{15} \dots)$
- 실루엣계수 $S(i) = \max(a(i), b(i)) - a(i)$
- 실루엣 계수는 -1~1 사이의 값을 가짐
 - 1로 가까울 수록, 근처의 군집과 더 멀리 떨어져 있다는 것
 - $b(i)$ 가 압도적으로 크면 $\rightarrow b(i) - a(i) \rightarrow 11 - 0.00 \dots \rightarrow 1$ 에 가까워짐
 - 0에 가까울 수록, 근처의 군집과 가까워진다는 것
 - $b(i) - a(i) = 0 \rightarrow b(i) = a(i)$: 클러스터내 거리랑, 타 클러스터내 거리랑 차이가 없다는 거니깐
 - 값이면 다른 군집에 데이터 포인트가 할당되었다는 것 $b(i) < a(i) \rightarrow$ 클러스터내 거리가 타 클러스터내 거리보다 크다 \rightarrow 다른 군집 데이터가 할당됐다고 볼 수 있음
- 사이킷런의 실루엣 분석 메소드
 - `silhouette_sample(X, labels, metric='euclidean', **kwargs)`
 - 인자로 X_feature 데이터 세트, 군집 레이블 값(labels) \Rightarrow 각 데이터의 실루엣 계수를 계산하여 반환
 - `silhouette_score(X, labels, metric='euclidean', sample_size=None, **kwargs)`

- 인자로 X feature 데이터 세트, 군집 레이블 값(labels) \Rightarrow 전체 데이터의 실루엣계수 값을 평균하여 반환
- `np.mean(silhouette_samples())` 랑 같음
- 일반적으로 이 값이 높을수록 군집화가 어느정도 잘 됐다고 판단할 수 있지만, 무조건 그런건 아니다.

◦ 사용

```
from sklearn.preprocessing import scale
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
# 실루엣 분석 metric 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

%matplotlib inline

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_

# iris 의 모든 개별 데이터에 실루엣 계수값을 구함.
score_samples = silhouette_samples(iris.data, irisDF['cluster'])
print('silhouette_samples( ) return 값의 shape' , score_samples.shape)
print(np.mean(silhouette_samples(iris.data, irisDF['cluster'])))
print(silhouette_score(iris.data, irisDF['cluster']))

> silhouette_samples( ) return 값의 shape (150,)
> 0.5528190123564095
> 0.5528190123564095

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

# 모든 데이터의 평균 실루엣 계수값을 구함.
average_score = silhouette_score(iris.data, irisDF['cluster'])
print('붓꽃 데이터셋 Silhouette Analysis Score:{0:.3f}'.format(average_score))

> 붓꽃 데이터셋 Silhouette Analysis Score:0.553

# 군집별 평균 실루엣 계수
print(irisDF.groupby('cluster')['silhouette_coeff'].mean())

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

2-3. 군집별 평균 실루엣 계수의 시각화를 통한 군집 갯수 최적화 방법

- 전체 데이터의 평균 실루엣 계수 값이 높다고 해서, 반드시 최적의 군집 개수로 군집화가 잘 됐다고 볼 수 없음

- 특정 군집만 실루엣 계수가 엄청 높고 나머지 군집들은 낮아도, 평균 실루엣 계수 자체는 높게 나올 수 있기 때문
- 따라서, 좋은 군집의 조건으로
 1. 전체 실루엣 계수의 평균값(`silhouette_score()`)은 0~1 사이의 값을 가지며, 1에 가까울 수록 좋다.
 2. 하지만 전체 실루엣 계수의 평균값과 더불어, 개별 군집의 평균값의 편차가 크지 않아야 한다.
 3. 즉, 개별 군집의 실루엣 계수 평균값이 전체 실루엣 계수 평균값에서 크게 벗어나지 않는 것이 중요하다.
- `visualize_silhouette([군집 갯수 list], X_feature)` 을 통한 실루엣 시각화 분석

```
### 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, 10), nrows=2, 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)
        centers = clusterer.cluster_centers_

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

        y_lower = 10
        axs[0,ind].set_title('Number of Cluster : '+ str(n_cluster)+'\n' \
                             'Silhouette Score : ' + str(round(sil_avg,3)) )
        axs[0,ind].set_xlabel("The silhouette coefficient values")
        axs[0,ind].set_ylabel("Cluster label")
        axs[0,ind].set_xlim([-0.1, 1])
        axs[0,ind].set_ylim([0, len(X_features) + (n_cluster + 1) * 10])
        axs[0,ind].set_yticks([]) # Clear the yaxis labels / ticks
        axs[0,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[0,ind].fill_betweenx(np.arange(y_lower, y_upper), 0, ith_cluster_sil_values, \
                             facecolor=color, edgecolor=color, alpha=0.7)
    axs[0,ind].text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))
    y_lower = y_upper + 10

    # 클러스터링된 데이터 시각화
    axs[1,ind].scatter(X_features[:, 0], X_features[:, 1], marker='.', s=30, lw=0, alpha=0.7, \
                       c=cluster_labels)
    axs[1,ind].set_title("Clustered data")
    axs[1,ind].set_xlabel("Feature space for the 1st feature")
    axs[1,ind].set_ylabel("Feature space for the 2nd feature")

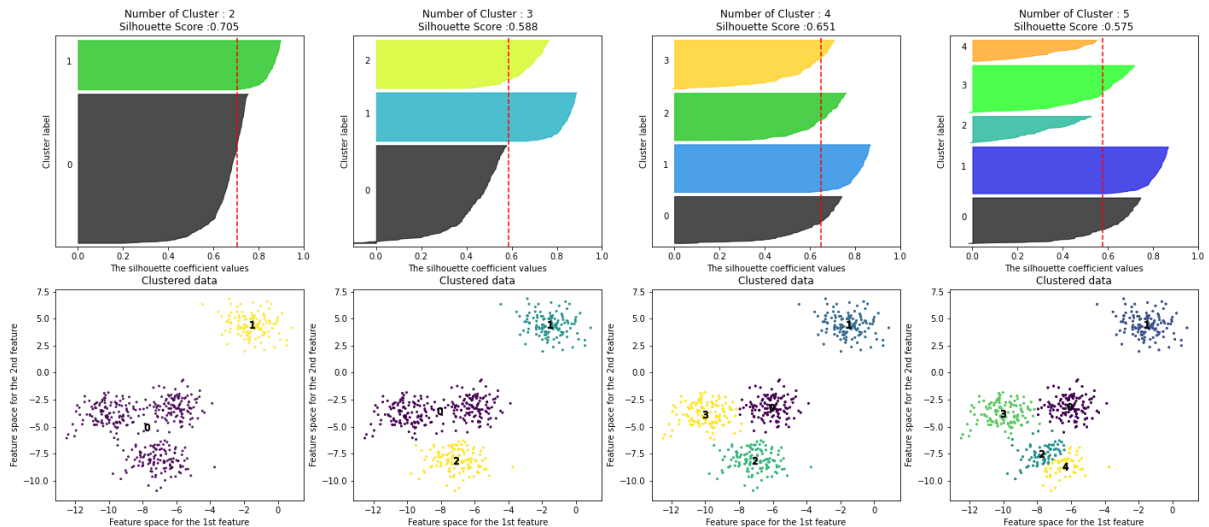
    # 군집별 중심 위치 좌표 시각화
    unique_labels = np.unique(cluster_labels)
    for label in unique_labels:
        center_x_y = centers[label]
        axs[1,ind].scatter(x=center_x_y[0], y=center_x_y[1], s=70, color='k', edgecolor='k',
                           marker='s' % label)

    axs[0,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)

```



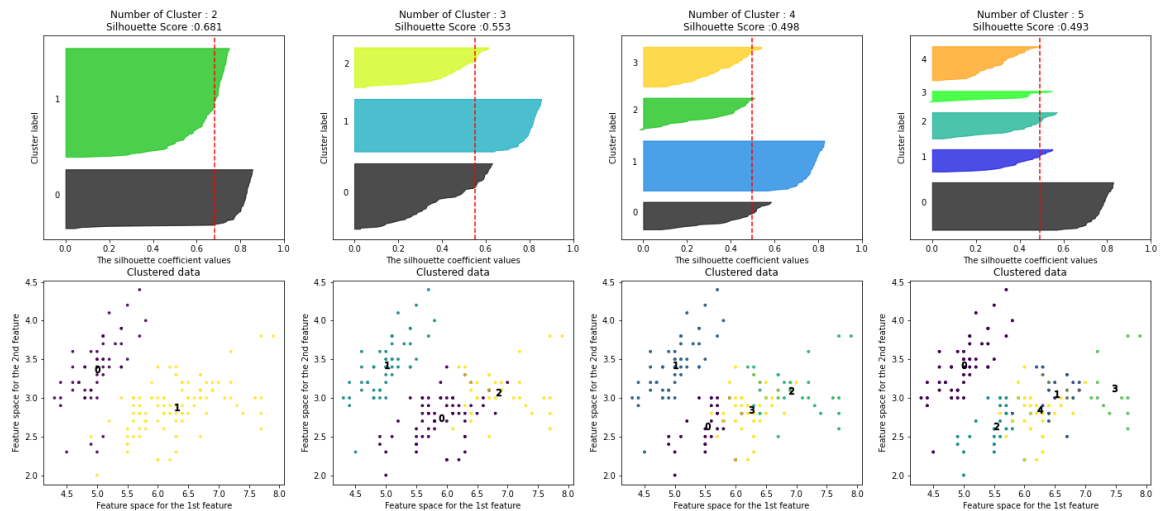
- iris 데이터로 실루엣 시각화 분석

```

from sklearn.datasets import load_iris

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

```

• 단점

- (직관적으로 이해하기 쉽지만) 각 데이터별로 다른 데이터와의 거리를 반복적으로 계산해야 하므로, 데이터 양이 늘어나면 수행시간이 크게 늘어난다.
- 또한 메모리 부족 등의 에러가 발생하기 쉬우며, 이 경우 군집별로 임의의 데이터를 샘플링 해 실루엣 계수를 평가하는 방안을 고민해야 한다.

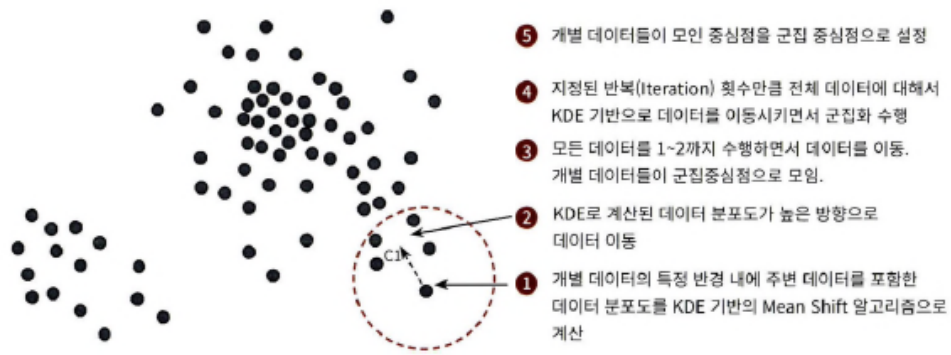
3 평균 이동 Mean Shift

: K-평균과 유사하게, 군집의 중심을 지속적으로 움직이면서 군집화를 수행함

: 그러나, K-평균이 중심에 소속된 데이터의 평균 거리 중심으로 이동하는데 반해,

: 평균 이동은 데이터가 모여있는 밀도가 가장 높은 곳으로 이동시키면서 군집화하는 방법

- 평균 이동 군집화는 데이터의 분포도를 이용해 군집 중심점을 찾음
 - 군집 중심점은 데이터 포인트가 모여있는 곳이라는 생각에서 착안
 - 이를 위해 확률 밀도 함수를 이용 함
 - 확률 밀도 함수가 피크인 점(가장 집중적으로 데이터가 모여 있을)을 군집 중심점으로 선정하며
 - 주어진 모델의 확률 밀도 함수를 찾기 위해서 KDE(Kernel Density Estimation)를 이용
 - 주변 데이터와의 거리 값을 KDE 함수 값으로 입력한 뒤, 그 반환 값을 현재 위치에서 업데이트하면서 이동하는 방식



- KDE(Kernel Density Estimation)

: 커널 함수를 통해 어떤 변수의 확률 밀도 함수를 추정하는 대표적인 방법

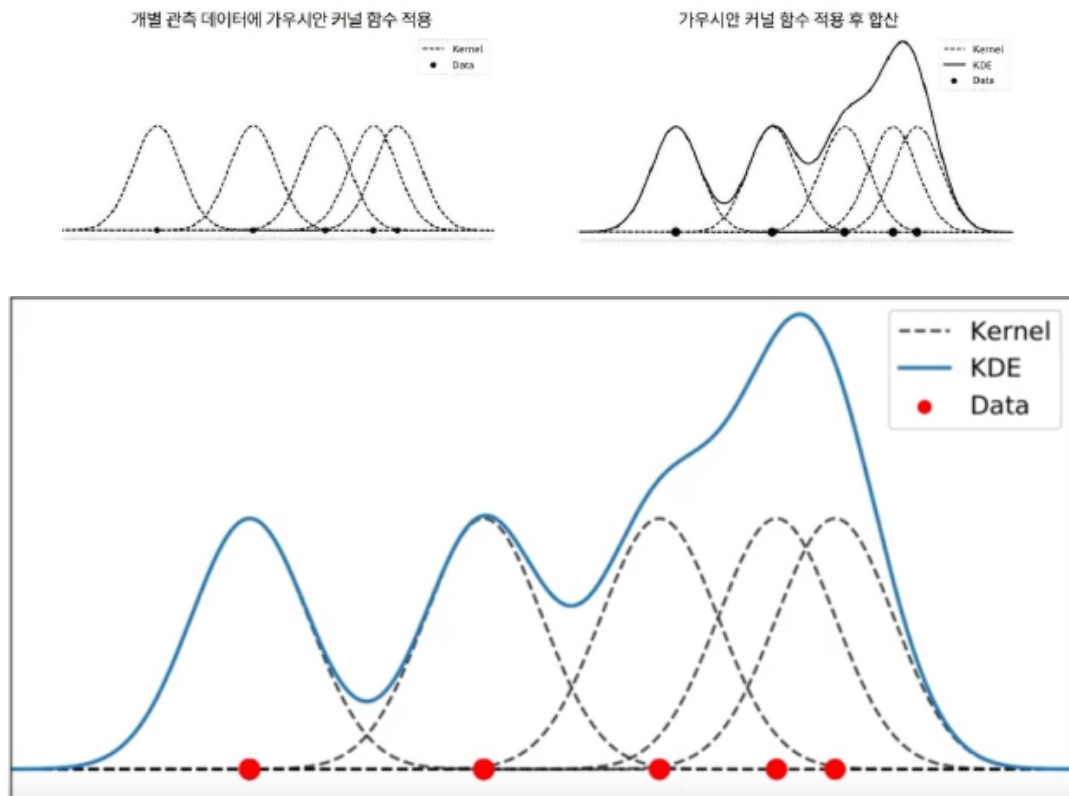
: 개별 데이터 각각에, 커널 함수를 적용한 값을 모두 더한 뒤 데이터 건수로 나눠 확률 밀도 함수를 추정한다.

- 확률 밀도 함수 PDF(Probability Density Function)

: 확률 변수의 분포를 나타내는 함수 (정규 분포, 감마 분포, t-분포 등)

- 확률 밀도 함수를 알면 특정 변수가 어떤 값을 갖게 될지에 대한 확률을 알게 되므로, 이를 통해 변수의 특성, 확률 분포 등 변수의 많은 요소를 알 수 있다.

- 커널 함수의 예시) 가우시안 커널 함수 적용



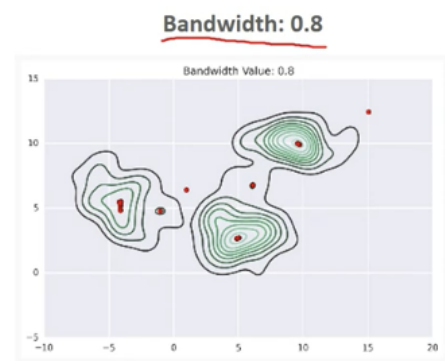
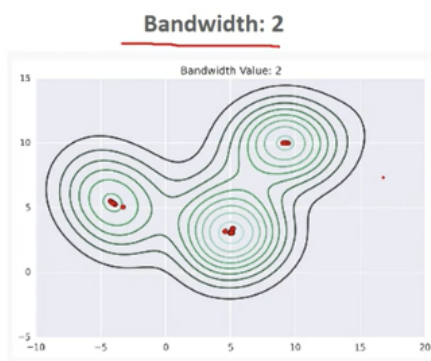
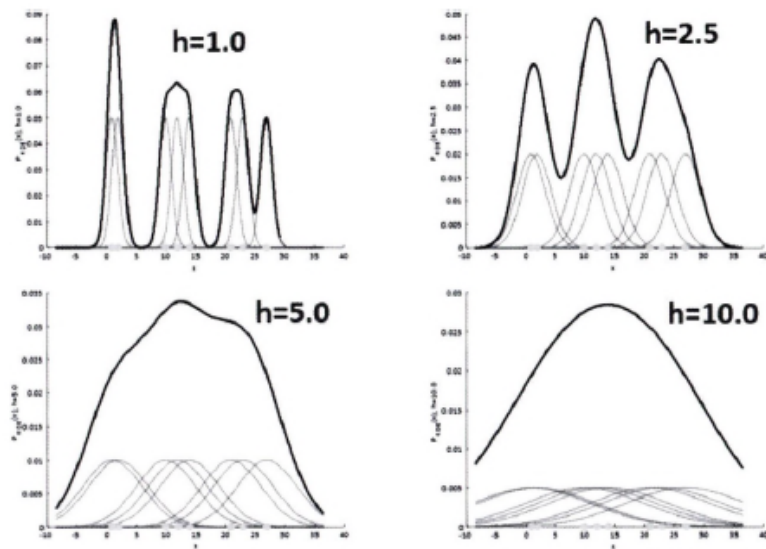
- 수식

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

- K : 커널 함수, x : 확률 변수 값, x_i : 관측값, h : 대역폭(bandwidth)
- 대역폭 h : KDE 형태를 부드럽거나 뾰족한 형태로 평활화(smoothing)하는데 적용
 - 작은 h 값: 좁고 뾰족한 KDE를 가짐. 과적합 되기 쉬움. 많은 수의 군집 중심점을 가짐
 - 큰 h 값: 과도하게 평활화된 KDE를 가짐. 과소적합 되기 쉬움. 적은 수의 군집 중심점을 가짐



적절한 h 를 계산하는 것이 KDE 기반의 평균 이동에서 매우 중요하다.



• 사용

```
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)
```

```

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]

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

> cluster labels 유형: [0 1 2]

```

- bandwidth (=KDE의 h) 값을 작게 할수록 군집 개수가 많아진다.
- ★ `estimate_bandwidth(X)` ★ : 최적의 대역폭 h 찾아줌. 파라미터로 피쳐 데이터 세트(X) 입력

```

from sklearn.cluster import estimate_bandwidth

# estimate_bandwidth()로 최적의 bandwidth 계산
bandwidth = estimate_bandwidth(X)
print('bandwidth 값:', round(bandwidth,3))

> bandwidth 값: 1.816

-----

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]

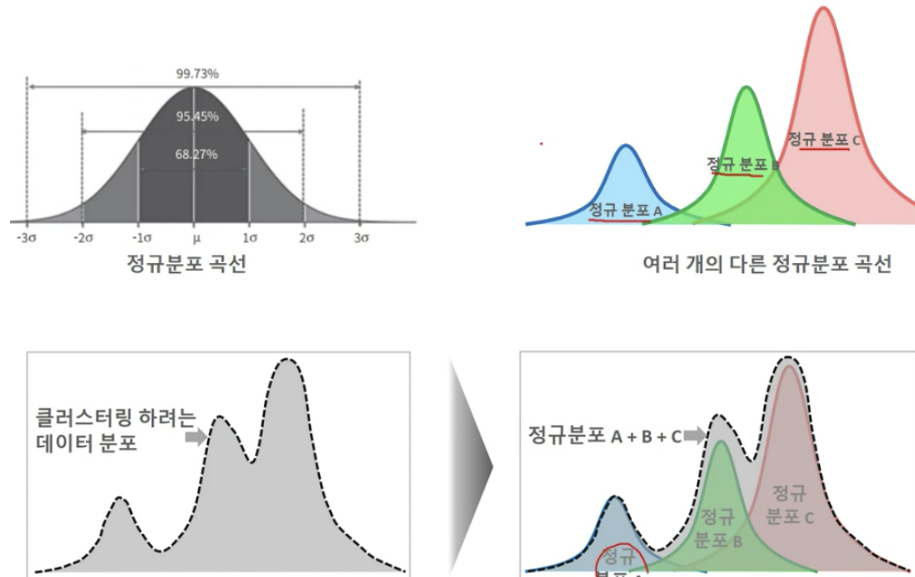
```

- 평균 이동의 장점
 - 데이터 세트의 형태를 특정 형태로 가정한다든가, 특정 분포 기반의 모델로 가정하지 않기 때문에 유연한 군집화 가능
 - 이상치의 영향력도 크지 않으며, 미리 군집의 개수를 정하지 않아도 된다.
- 평균 이동의 단점
 - 수행 시간이 오래 걸리고, bandwidth의 크기에 따른 군집화 영향도가 크다.
- 활용
 - 컴퓨터 비전 영역에서 많이 사용
 - 이미지나 영상 데이터에서, 특정 개체를 구분하거나 움직임을 추적하는데 뛰어난 역할

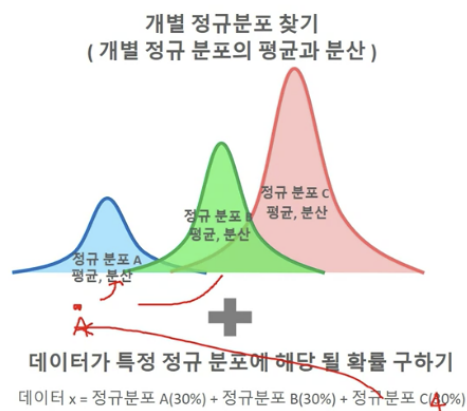
4 GMM(Gaussian Mixture Model) - 확률 기반 군집화



군집화를 적용하고자 하는 데이터가, 여러 개의 가우시안 분포를 가진 데이터 집합들이 섞여서 생성된 것이라는 가정하에, 군집화를 수행하는 방식



- 정규분포: 평균 μ 를 중심으로 높은 데이터 분포도를 가지고 있으며, 좌우 표준편차 1에 전체 데이터의 68.27%, 좌우 표준편차 2에 전체 데이터의 95.45%를 갖고 있다.
- 표준 정규 분포: 평균이 0, 표준편차가 1인 정규분포
- 섞인 데이터 분포에서 개별 유형의 가우시안 분포 추출, 개별 데이터가 이 중 어떤 정규분포에 속하는지 결정하는 방식



- 모수 추정 : 개별 정규 분포의 평균과 분산 추정, 각 데이터가 어떤 정규분포에 해당되는지의 확률 추정
 - 가령 1,000개의 데이터 세트 있다면, 이를 구성하는 여러 개의 정규 분포 곡선을 추출
 - 개별 데이터가 이 중 어떤 정규 분포에 속하는지 결정하는 방식
- 사용

```

from sklearn.mixture import GaussianMixture

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

# 클러스터링 결과를 irisDF 의 'gmm_cluster' 컬럼명으로 저장
irisDF['gmm_cluster'] = gmm_cluster_labels
irisDF['target'] = iris.target

# target 값에 따라서 gmm_cluster 값이 어떻게 매핑되었는지 확인.
iris_result = irisDF.groupby(['target'])['gmm_cluster'].value_counts()
print(iris_result)

```

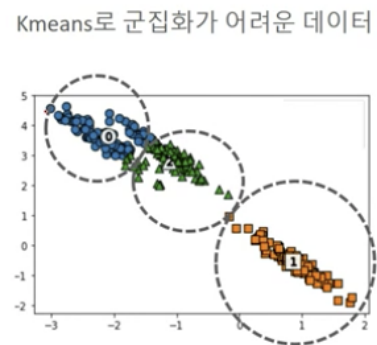
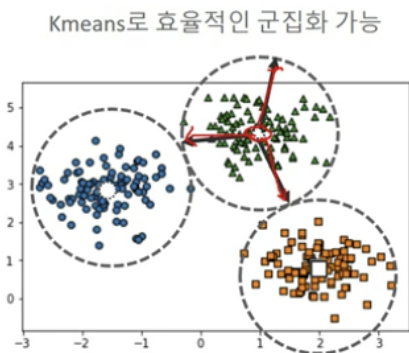
```

> target  gmm_cluster
> 0        0           50
> 1        2           45
>        1            5
> 2        1           50
> Name: gmm_cluster, dtype: int64

```

- **n_components** : 모델의 총 개수. 군집의 개수를 정하는데 중요한 역할 수행
- fit(피쳐 데이터 세트), predict(피쳐 데이터 세트)를 수행해 군집을 결정
- 장점 : KMeans보다 유연하게 다양한 데이터 세트에 잘 적용될 수 있다.

(Not 원형 범위여도 작동 잘 함)



- 성능이 더 좋다는 뜻이 아니라, K-평균은 평균 거리 중심을 이동하면서 군집화를 수행하여, 개별 군집 내의 데이터가 원형으로 흩어져 있는 경우에 매우 효과적으로 군집화가 수행될 수 있음⇒ 데이터 세트 구성에 따라 성능 달라짐따라서 K-평균은 길쭉한 타원형으로 늘어선 경우에는 군집화를 잘 수행하지 못함
- 단점 : 수행시간이 오래 걸림

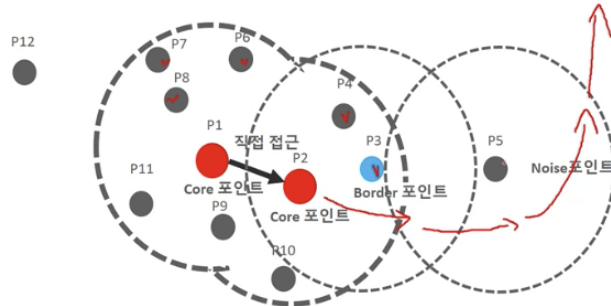
5 DBSCAN(Density Based Spatial Clustering of Applications with Noise) - 밀도 기반 군집화



입실론 주변 영역의 최소 데이터 갯수를 포함하는 밀도 기준을 충족시키는 데이터인, 핵심 포인트를 연결하면서 군집화를 구성하는 방식

→ 데이터의 분포가 기하학적으로 복잡한 데이터 세트에도 효과적으로 군집화 가능!

핵심 포인트(Core Point): 주변 영역 내에 최소 데이터 개수 이상의 타 데이터를 가지고 있을 경우 해당 데이터를 핵심 포인트라고 합니다.
이웃 포인트(Neighbor Point): 주변 영역 내에 위치한 타 데이터를 이웃 포인트라고 합니다.
경계 포인트(Border Point): 주변 영역 내에 최소 데이터 개수 이상의 이웃 포인트를 가지고 있지 않지만 핵심 포인트를 이웃 포인트로 가지고 있는 데이터를 경계 포인트라고 합니다.
잡음 포인트(Noise Point): 최소 데이터 개수 이상의 이웃 포인트를 가지고 있지 않으며, 핵심 포인트도 이웃 포인트로 가지고 있지 않는 데이터를 잡음 포인트라고 합니다



• 사용

```
from sklearn.cluster import DBSCAN

dbscan = DBSCAN(eps=0.6, min_samples=8, metric='euclidean')
dbscan_labels = dbscan.fit_predict(iris.data)

irisDF['dbscan_cluster'] = dbscan_labels
irisDF['target'] = iris.target

iris_result = irisDF.groupby(['target'])['dbscan_cluster'].value_counts()
print(iris_result)
```

```
> target  dbscan_cluster
> 0        0             49
>      -1             1
> 1        1            46
>      -1             4
> 2        1            42
>      -1             8
> Name: dbscan_cluster, dtype: int64

# 군집 레이블이 -1인 것은 노이즈에 속하는 군집을 의미
```

- 군집 레이블이 -1인 것은 노이즈에 속하는 군집을 의미
- Target 유형이 3가지 인데, 군집이 2개가 됐다고 군집화 효율이 떨어진다는 의미는 아님
 - DBSCAN은 군집의 갯수를 알고리즘에 따라 자동으로 지정하므로, DBSCAN에서 군집의 갯수를 지정하는 것은 무의미

- 원래 iris 데이터의 경우는 군집을 3개로 하는 것 보다, 2개로 하는 것이 군집화의 효율로서 더 좋은 면도 실제로 있음

• 파라미터

파라미터	설명
eps	입실론(epsilon)

개별 데이터를 중심으로 입실론 반경을 가지는 원형의 영역

일반적으로 1 이하의 값 설정

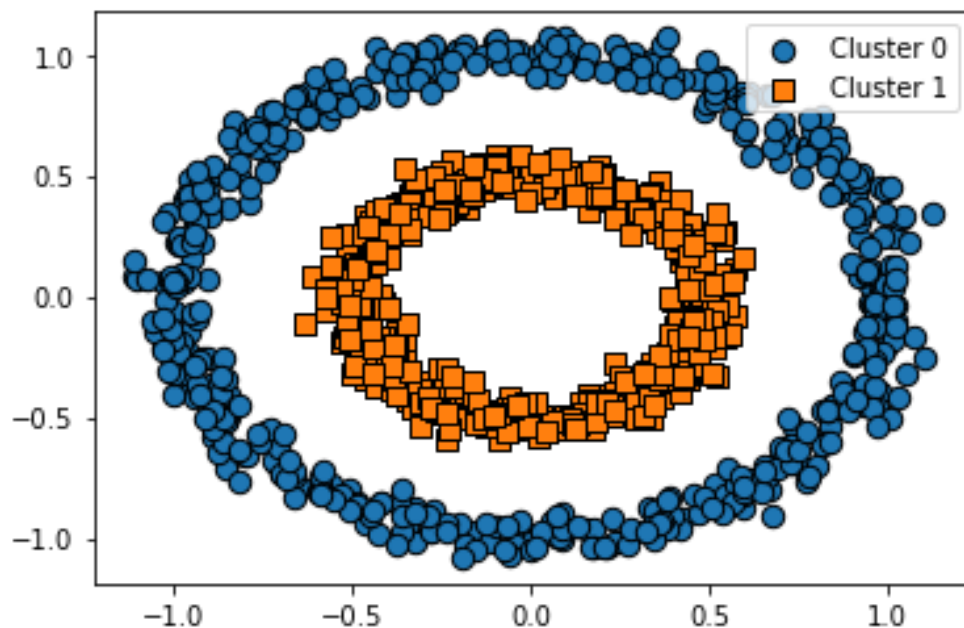
| min_samples | 최소 데이터 개수(min points)

개별 데이터의 입실론 주변 영역에 포함되는 타 데이터의 개수 (자기 자신 포함) |

- eps 값을 크게 하면, 반경이 커져 포함하는 데이터가 많아지므로 노이즈 데이터 개수가 작아진다.
- min_samples를 크게 하면, 주어진 반경 내에서 더 많은 데이터를 포함시켜야 하므로 노이즈 데이터 개수가 커진다.

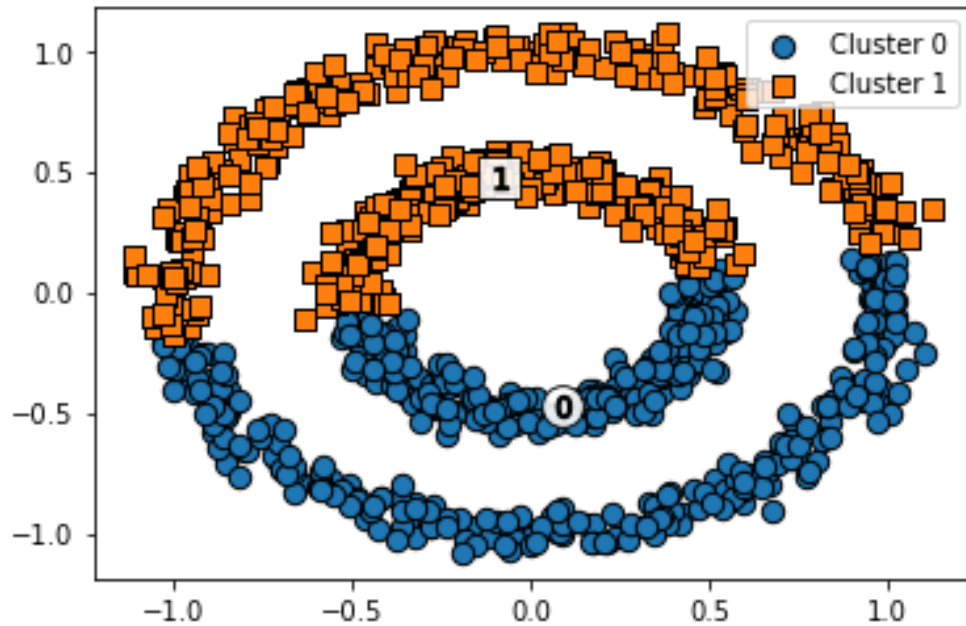
DBSCAN 적용하기 - make_circles() 데이터 세트 +- 비교하기

• make_circles() 원본



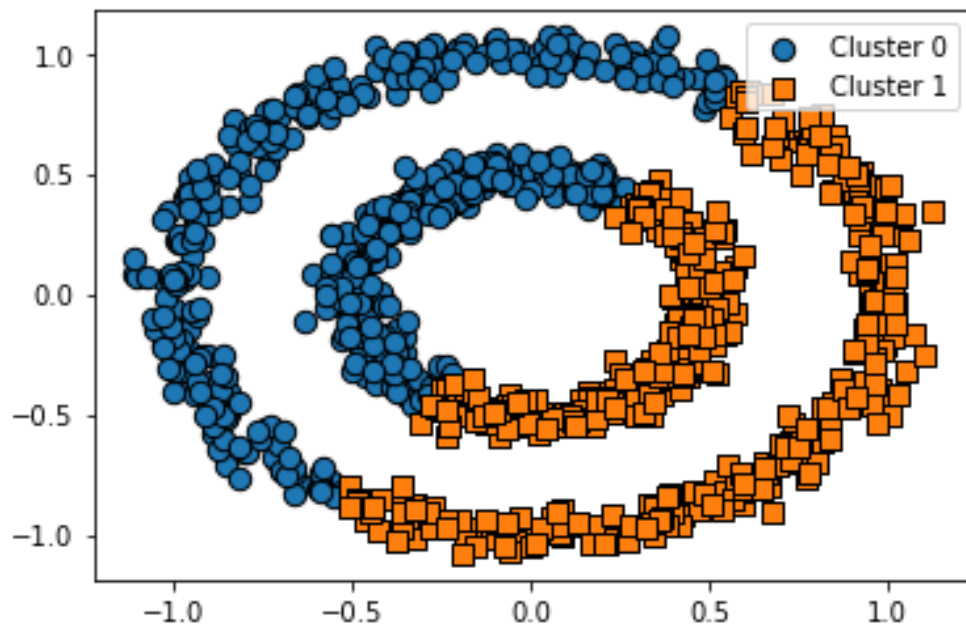
- make_circles 는 내부 원과 외부 원으로 구분되는 데이터 세트를 생성해줌

• Kmeans (거리기반 군집화)



◦ 거리를 기반으로, 위, 아래 군집 중심을 기반으로 군집화 됨

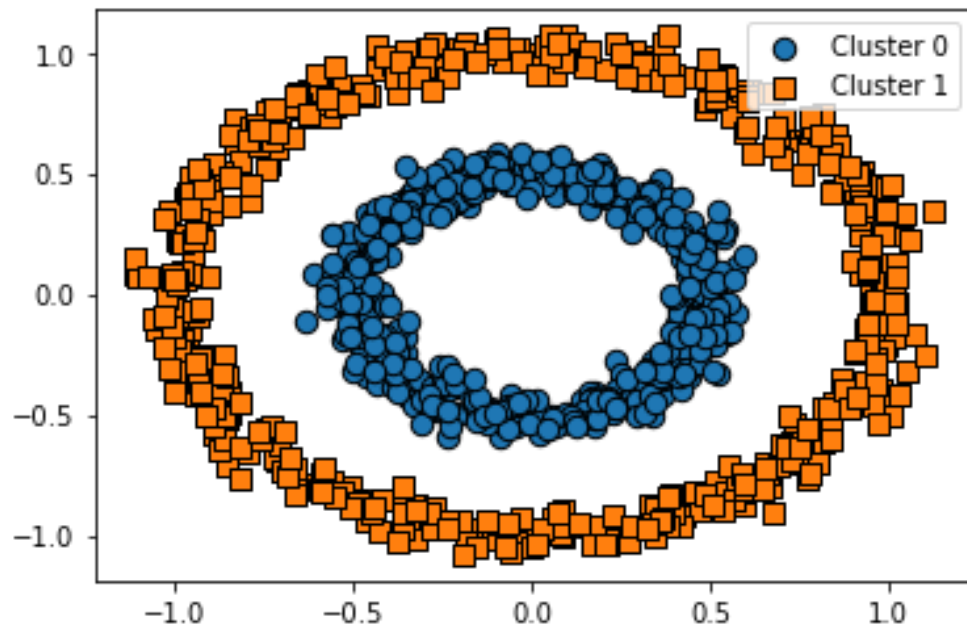
- **GMM (확률 기반 군집화)**



◦ 일렬로 늘어선 데이터 세트(타원형)에서는 효과적으로 군집화 적용이 가능했으나,

◦ 내부와 외부의 원형으로 구성된 더 복잡한 형태의 데이터 세트에서는 군집화가 원하는 방향으로 되지 않았음

- **DBSCAN**



→ 정확한 군집화