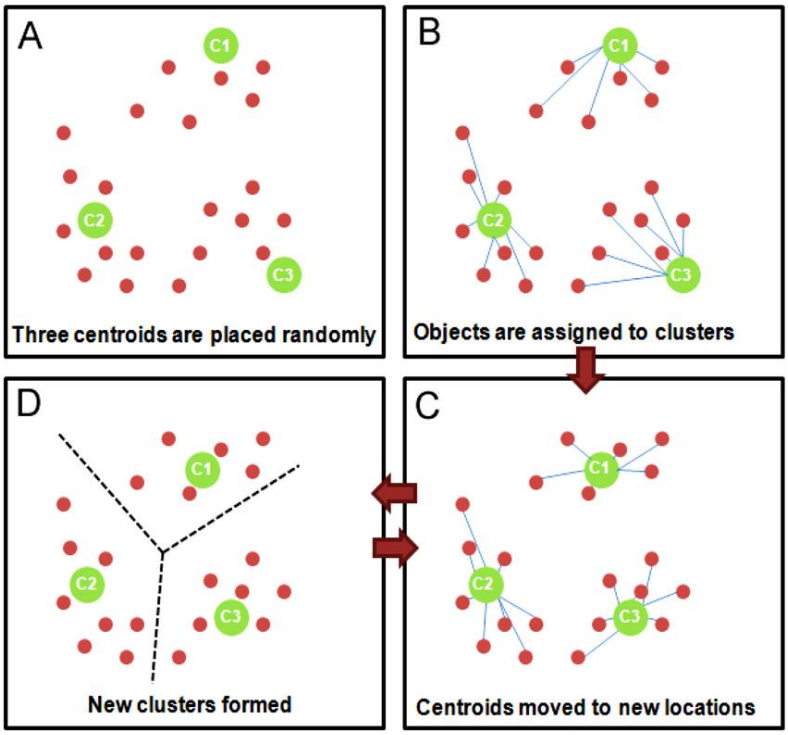
07

군집화

K-means 알고리즘

K-means 알고리즘은 군집화(clustering)에서 가장 일반적으로 사용되는 알고리즘입니다. K-means 는 군집 중심(centroid)이라는 특정한 임의의 지점을 선택해 해당 중심에 가장 가까운 포인트들을 선택하는 군집화 기법입니다.

Centroid는 선택된 포인트의 평균 지점으로 이동하고, 이동된 중심점에서 다시 가까운 포인트를 선택, 다시 중심점을 평균 지점으로 이동하는 프로세스를 반복적으로 수행합니다. 모든 데이터 포인트에서 더 이상 중심점의 이동이 없을 경우에 반복을 멈추고 해당 중심점에 속하는 데이터들을 군집화하는 기법입니다.



이러한 k-means 알고리즘은 거리 기반 알고리즘으로 속성의 개수가 매우 많을 경우 군집화 정확도가 떨어지게 됩니다. 이를 위해 PCA 로 차원 감소를 적용해야할 수도 있습니다. 또한 hyper-parameter 에 해당되는 k, 즉 군집의 개수를 어떻게 선택할지 가이드하기가 어려울 수 있습니다.

Scikit-learn 은 k-means 를 구현하기 위해 Kmeans 클래스를 제공합니다. Kmeans 클래스는 아래와 같은 초기화 parameter 를 가지고 있습니다.

```
class sklean.cluster.Kmeans(n_clusters=8, init='k-means++', n_init=10,
                            max_iter=300, tol=0.0001, precompute_distances='auto',
                            verbose=0, random_state=None, copy_x=True, n_jobs=1,
                            algorithm='auto')
```

이중 중요한 parameter 는 아래와 같습니다.

- n_clusters: centroid 의 개수
- init: 초기에 centroid 좌표를 설정할 방식, 보통 k-means++으로 최초 설정.
- max_iter: 최대 반복횟수, 이 횟수 이전에 모든 데이터의 centroid 이동이 없으면 종료

Kmeans 객체는 군집화 수행이 완료되면 아래와 같은 주요 속성을 알 수 있게 됩니다.

- labels_: 각 데이터포인트가 속한 레이블
- cluster_centers_: 각 centroid 의 좌표, 이를 통해 centroid 좌표가 어디인지 시각화할 수 있음.

K-means 알고리즘을 이용한 붓꽃데이터 군집화

붓꽃 데이터를 이용해 k-means 군집화르 수행해 보겠습니다. 붓꽃의 꽃받침(sepal)과 꽃잎(petal) 길이와 너비에 따른 품종을 분류하는 데이터셋입니다. 붓꽃 데이터셋을 3 개의 그룹으로 군집화해 보겠습니다. 이를 위해 n_clusters 는 3, 초기 중심 설정 방식은 default 값인 k-means++, 최대 반복 횟수 역시 default 값인 300 으로 설정하여 KMeans 객체를 만들고 여기에 fit( )을 수행하겠습니다.

```
kmeans = KMeans(n_clusters=3, init='k-means++', max_iter=300,random_state=0).fit(irisDF)
```

kmeans 의 labels_ 속성 값을 확인해보고, 이 labels_값을 'cluster' column 으로 irisDF 에 추가하겠습니다.

```
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 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 2 0 2 0 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 0 2 0 0 0 0 2 0 0 0 2 0 0 0 2 0
 0 2]
```

```
irisDF['target'] = iris.target
```

실제 붓꽃 품종 분류 값과 얼마나 차이가 나는지로 군집화가 효과적으로 됐는지 확인해 보겠습니다. DataFrame 에 group by 연산을 실제 분류 값인 target 과 군집화 분류값인 cluster 레벨로 적용해 target 과 cluster 값 개수를 비교할 수 있습니다.

```
iris_result = irisDF.groupby(['target','cluster'])['sepal_length'].count()
print(iris_result)
```

```
target  cluster
0       1          50
1       0          48
        2           2
2       0          14
        2          36
```

분류 타깃이 0 인 데이터는 1 번 군집으로 모두 잘 그룹화 되었습니다. target1 값 데이터는 2 개만 2 번 군집으로 분류되었고, 나머지 48 개는 모두 0 번 군집으로 그룹화되었습니다. 하지만 target2 의 경우 0 번, 2 번에 분산되어 그룹화되었습니다.

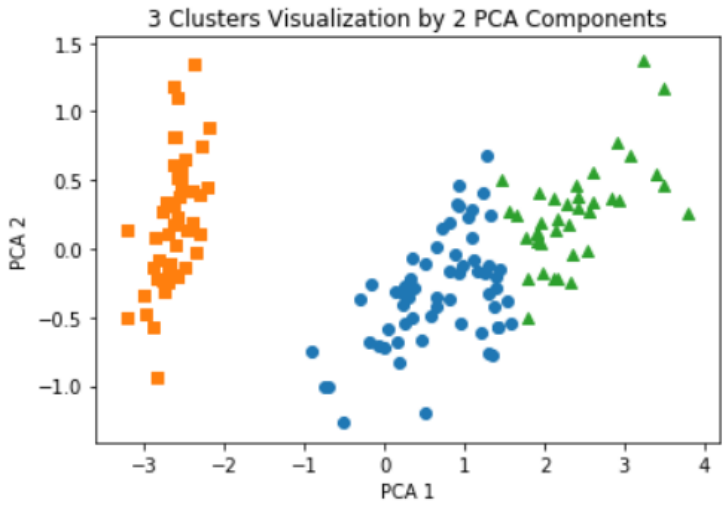
이번에는 붓꽃 데이터셋의 군집화를 시각화해보겠습니다. 붓꽃 데이터셋의 속성이 4 개이므로 2 차원 평면에 적합치 않아 PCA 를 이용해 4 개의 속성을 2 개로 차원축소한 뒤에 X 좌표, Y 좌표로 개별 데이터를 표현하도록 하겠습니다.

```
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)
```

	sepal_length	sepal_width	petal_length	petal_width	cluster	target	pca_x	pca_y
0	5.1	3.5	1.4	0.2	1	0	-2.684207	0.326607
1	4.9	3.0	1.4	0.2	1	0	-2.715391	-0.169557
2	4.7	3.2	1.3	0.2	1	0	-2.889820	-0.137346

이제 각 군집별로 cluster 0 은 마커 'o', cluster 1 은 마커 'S', cluster 2 는 마커 '^'로 표현합니다.

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

# cluster값 0, 1, 2에 해당하는 Index로 각 cluster 레벨의 pca_x, pca_y 값 추출. o, s, ^ 로
marker 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 1 을 나타내는 네모는 명확히 다른 군집과 잘 분리돼 있습니다. cluster 0 을 나타내는 동그라미와 cluster 2 를 나타내는 세모는 네모만큼 명확하게는 분리돼 있지 않음을 알 수 있습니다.

군집화 알고리즘 테스트

Scikit-lean 은 다양한 유형의 군집화 알고리즘을 테스트해 보기 위한 간단한 데이터 생성기를 제공합니다. 대표적인 군집화용 데이터 생성기로는 make_blobs( )와 make_classification( ) API가 있습니다.

make_blobs( ) 의 간략한 사용법을 알아보면서 군집화를 위한 테스트 데이터셋을 만드는 방법을 살펴보겠습니다. make_blobs( )를 호출하면 feature dataset와 target dataset가 tuple로 반환됩니다. 호출 parameter는 아래와 같습니다.

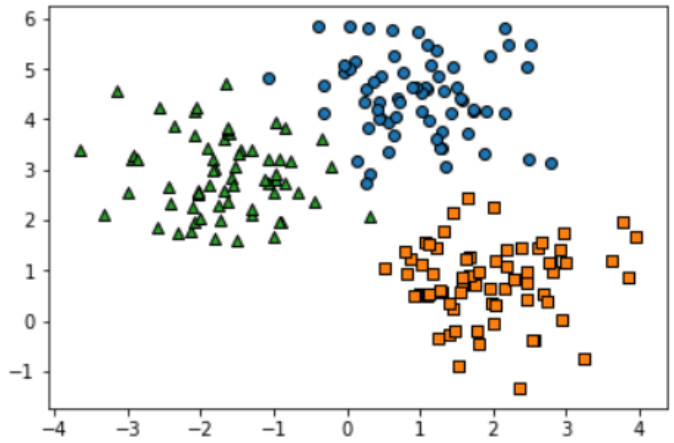
- n_samles: 생성할 총 데이터의 개수, default는 100개
- n_features: 데이터의 feature 개수, 시각화를 목표로 할 경우 2 개로 설정해 보통 첫 번째 feature는 x좌표, 두 번째 feature는 y좌표 상에 표현함
- centers: int 로 설정하면 군집의 개수, 그렇지 않고 ndarray 로 표현할 경우 개별 centroid의 좌표를 의미함
- cluster_std: 생성될 군집데이터의 표준편차

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

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]
```

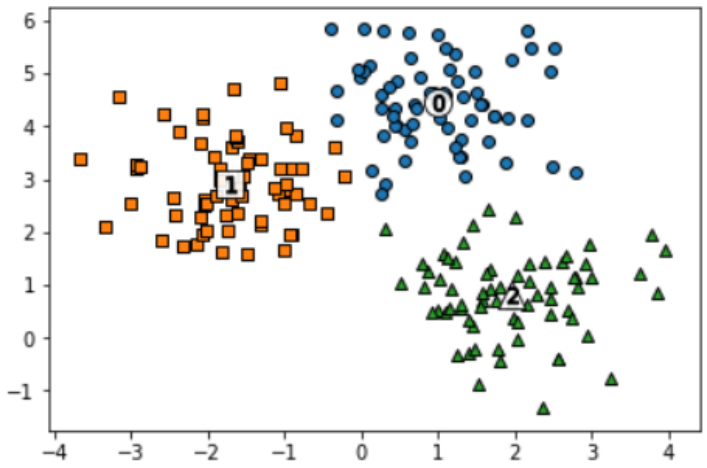
feature dataset 는 200 개의 record 와 2 개의 feature 를 가지므로 shape 은 (200,2), target dataset 의 shape 은 (200,), 그리고 3 개의 cluster 의 값은 [0,1,2]이며 각각 67, 67,66 개로 균일하게 구성돼 있습니다. 시각화해보면 아래와 같습니다.

```
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()
```



이번에는 이렇게 만들어진 데이터셋에 Kmeans 군집화를 수행한 뒤에 군집별로 시각화해보겠습니다. 먼저 Kmeans 객체에 fit_predict(X)를 수행해 make_blobs( )의 feature dataset 인 X 데이터셋을 군집화합니다.

```
# 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()
```

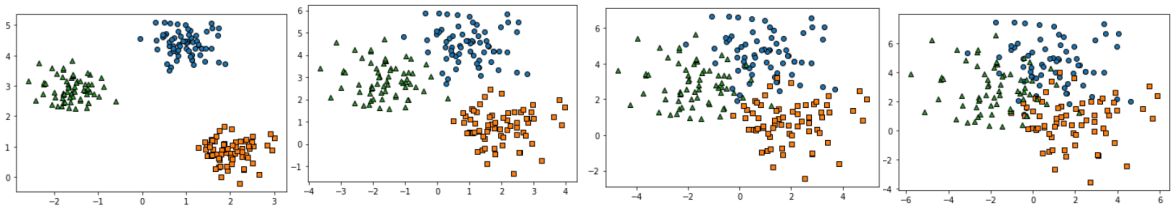



```
print(clusterDF.groupby('target')['kmeans_label'].value_counts())
```

```
target  kmeans_label
0       0              66
        1               1
1       2              67
2       1              65
        2               1
```

대부분 매핑이 잘 된 모습을 확인할 수 있습니다.

make_blobs( )는 cluster_std parameter 로 데이터의 분포도를 조절합니다. 다음 그림은 cluster_std 가 0.4, 0.8, 1.2, 1.6 일 때를 시각화한 것입니다. cluster_std 가 작을수록 군집 중심에 데이터가 모여있으며, 클수록 퍼져있음을 알 수 있습니다.



군집평가 (Cluster Evaluation)

앞의 붓꽃 데이터셋의 경우 결과값에 품종을 뜻하는 target label 이 있었고, 군집화 결과를 이 레이블과 비교해 군집화가 얼마나 효율적으로 되었는지 짐작할 수 있었습니다. 하지만 대부분의 군집화 데이터셋은 이렇게 비교할 만한 타깃 레이블을 가지고 있지 않습니다. 그렇다면 군집화가 얼마나 효율적으로 잘 됐는지 평가할 수 있는 지표로서 실루엣 분석을 이용합니다.

실루엣 분석은 각 군집 간의 거리가 얼마나 효율적으로 분리돼 있는지를 나타냅니다. 효율적으로 잘 분리됐다는 것은 다른 군집과의 거리는 떨어져 있고 동일한 군집끼리의 데이터는 서로 가깝다는 뜻입니다. 실루엣 분석은 실루엣 계수를 기반으로 합니다. 실루엣 계수는 개별 데이터가 가지는 군집화 지표입니다. 개별 데이터가 가지는 실루엣 계수는 해당 데이터가 같은 군집 내의 데이터와 얼마나 가깝게 군집화 되어 있고, 다른 군집과는 얼마나 멀리 분리돼있는지를 나타내는 지표입니다.

실루엣 계수 값은 해당 데이터 포인트와 같은 군집 내에 있는 다른 데이터포인트와의 거리를 평균한 값 a(i), 해당 데이터 포인트가 속하지 않은 군집 중 가장 가까운 군집과의 평균 거리 b(i)를 기반으로 계산됩니다. 두 군집이 얼마나 떨어져 있는가의 값은 b(i)-a(i)이며, 이 값을 정규화 하기 위해 MAX(a(i), b(i)) 값으로 나눕니다.

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

실루엣 계수는 -1 에서 1 사이의 값을 가지며, 1 로 가까워질수록 근처의 군집과 더 멀리 떨어져 있다는 것이고, 0 에 가까울수록 근처의 군집과 가까워진다는 것입니다. - 값은 아예 다른 군집에 데이터 포인트가 할당됐음을 뜻합니다.

Scikit-learn 은 이러한 실루엣 분석을 위해 다음과 같은 method 를 제공합니다.

- sklearn.metrics.silhouette_samples(X, labels, metric='euclidean', **kwds): X feature dataset 와 각 feature data 가 속한 군집 레이블 값인 labels 를 입력해주면 실루엣 계수를 반환합니다.
- sklearn.metrics.silhouette_score(X, labels, metric='euclidean', sample_size=None, **kwds): 전체 데이터의 실루엣 계수 값의 평균

붓꽃 데이터셋의 군집화 결과를 실루엣 분석으로 평가해 보겠습니다. 이를 위해 sklearn.metrics 모듈의 silhouette_samples( )와 silhouette_score( )를 이용합니다.

```
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)

# 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))

irisDF.head(3)
```

```
silhouette_samples( ) return 값의 shape (150,)
붓꽃 데이터셋 Silhouette Analysis Score:0.553
```

	sepal_length	sepal_width	petal_length	petal_width	cluster	silhouette_coeff
0	5.1	3.5	1.4	0.2	1	0.851573
1	4.9	3.0	1.4	0.2	1	0.817887
2	4.7	3.2	1.3	0.2	1	0.830087

평균 실루엣 계수 값이 약 0.553 입니다. 1 번 군집의 경우 0.8 이상의 높은 실루엣 계수를 나타내는데, 전체적인 평균 값이 낮은 이유는 다른 군집의 실루엣 계수 값이 작기 때문입니다.

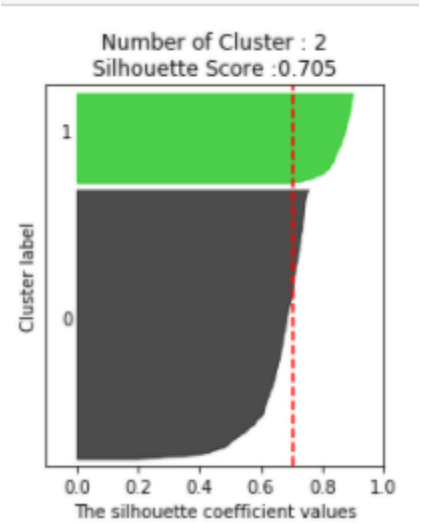
```
irisDF.groupby('cluster')['silhouette_coeff'].mean()
```

```
cluster
0    0.417182
1    0.797630
2    0.451105
```

군집 개수 최적화 방법

앞서 본 예제에서도 확인가능하듯, 전체 데이터의 평균 실루엣 계수가 높다고 반드시 최적의 군집 개수로 군집화가 잘 됐다고 볼 수는 없습니다.

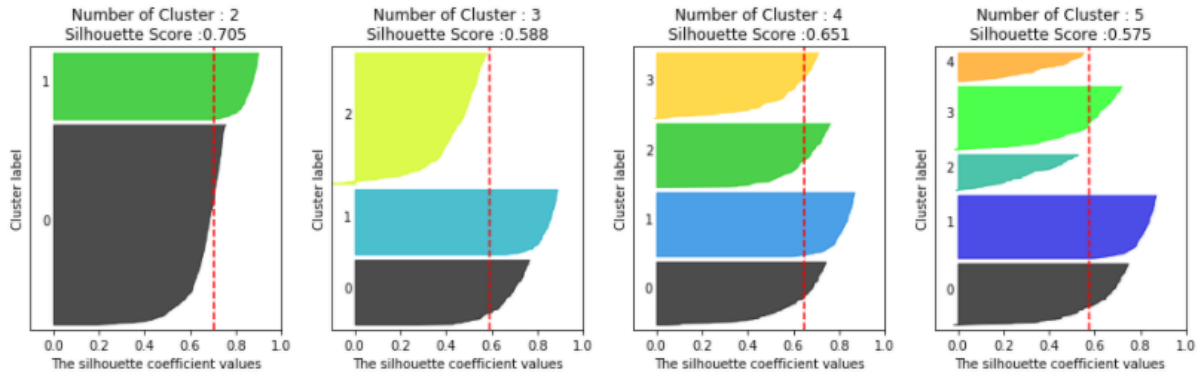
첫번째 예시는 주어진 데이터에 대해 군집 개수를 2 개로 설정했을 때입니다. 이때 평균 실루엣 계수는 0.704 로 매우 높게 나타났습니다. 하지만 이렇게 2 개로 군집화하는 것이 최적의 방법일까요? 아래 그림에서 x 축은 실루엣 계수 값이고, y 축은 개별 군집과 이에 속하는 데이터입니다. 그리고 점선은 전체 평균 실루엣 계수입니다.



위 예시는 1 번 군집의 모든 데이터는 평균 실루엣 계수 값 이상이지만, 2 번 군집의 경우는 평균보다 적은 데이터 값이 매우 많습니다.

아래는 차례대로 2, 3, 4, 5 개로 군집 개수를 다르게 했을 때 군집화 결과입니다.

```
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)
```

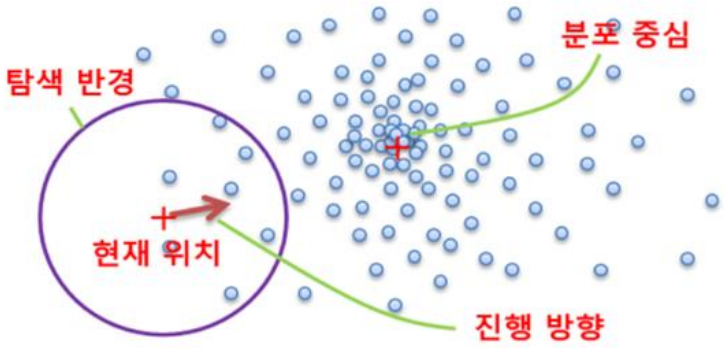


이 그래프 분석을 통해 평균 실루엣 값은 작지만 군집 개수가 4 개일 때 가장 이상적인 군집화가 진행되었음을 알 수 있습니다.

실루엣 계수를 통한 군집 평가 방법은 직관적으로 이해하기 쉽지만 각 데이터별로 계산량이 발생하므로 데이터 양에 따라 수행 시간이 크게 늘어날 수 있습니다. 특히 몇 만 건 이상의 데이터에 대해 개인용 pc 에서 실루엣 계수 평가를 수행하면 메모리 부족 등의 에러가 발생할 수 있습니다. 이 경우 군집별로 임의의 데이터를 샘플링해 실루엣 계수를 평가하는 방안을 고민해봐야 합니다.

Mean Shift

Mean shift 는 k-means 와 유사하게 중심을 군집의 중심으로 지속적으로 움직이면서 군집화를 수행합니다. 하지만 k-means 가 소속된 데이터의 평균 거리를 중심으로 이동하는 데 반해, mean shift 는 데이터가 모여있는 밀도가 가장 높은 곳으로 이동시킵니다.



이를 위해 확률 밀도 함수를 이용합니다. 가장 집중적으로 데이터가 모여 있어 확률 밀도 함수가 피크인 점을 centroid 로 설정합니다. 일반적으로 주어진 모델의 확률 밀도 함수를 찾기 위해서 KDE (Kernel Density Estimation)를 이용합니다. Mean shift 는 임의의 포인트에서 시작해 이러한 피크 포인트를 찾을 때까지 KDE 를 반복적으로 이용하며 군집화를 수행합니다.

Mean shift 는 k-means 와 다르게 군집의 개수를 지정할 필요가 없습니다. 대역폭에 따라 알고리즘 자체에서 군집의 개수를 최적으로 정합니다. 하지만 이 때문에 대역폭의 크기를 어떤 값으로 정하느냐에 따라 군집화의 품질이 달라지게 됩니다.

Scikit-learn 은 mean shift 를 위해 MeanShift 클래스를 제공합니다. 다음은 make_blobs( )의 cluster_std 를 0.8 로 정한 3 개의 군집에 대해 bandwidth 를 0.9 로 설저한 mean shift 알고리즘을 적용한 예시입니다.

```
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=0.8, random_state=0)

meanshift= MeanShift(bandwidth=0.9)
cluster_labels = meanshift.fit_predict(X)
print('cluster labels 유형:', np.unique(cluster_labels))
```

```
cluster labels 유형: [0 1 2 3 4 5 6 7]
```

군집이 8 개로 분리되었습니다. 일반적으로 bandwidth 의 값을 작게 할수록 군집의 개수가 많아집니다. 이번에는 bandwidth 를 살짝 높인 1.0 으로 mean shift 를 수행하겠습니다.

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

```
cluster labels 유형: [0 1 2]
```

3 개의 군집으로 잘 군집화됐습니다. 따라서 mean shift 에서는 bandwidth 를 최적화 값으로 설정하는 것이 매우 중요합니다. scikit-learn 은 최적화된 bandwidth 를 찾기 위해서 estimate_bandwidth( )를 제공합니다.

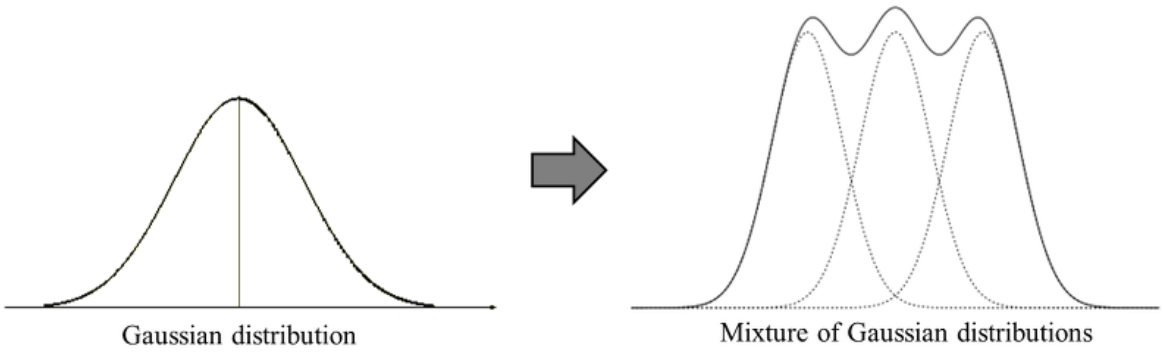
```
from sklearn.cluster import estimate_bandwidth

bandwidth = estimate_bandwidth(X,quantile=0.2)
print('bandwidth 값:', round(bandwidth,3))
```

```
bandwidth 값: 1.444
```

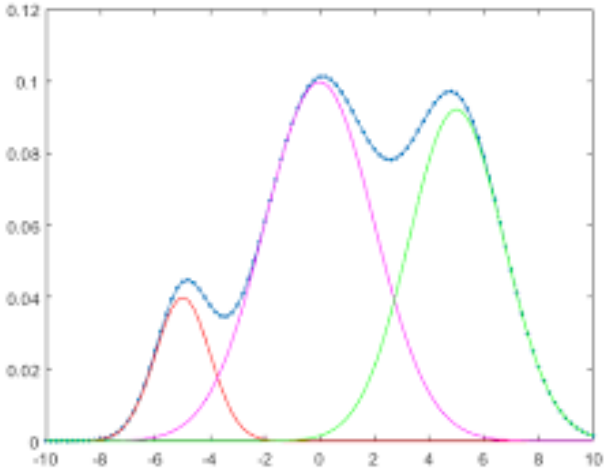
Mean shift 의 장점은 좀 더 유연한 군집화가 가능하다는 점입니다. 이상치의 영향력도 크지 않으며 미리 군집의 개수를 정할 필요도 없습니다. 다만, 알고리즘의 수행 시간이 오래 걸리고 bandwidth 에 따른 군집화 영향도가 매우 큽니다. 때문에 일반적으로 mean shift 는 computer vision 영역에서 잘 사용됩니다. 이미지에서 특정 개체를 구분하거나 움직임을 추적하는 데 뛰어난 기능을 발휘하기 때문입니다.

GMM (Gaussian Mixture Model)

GMM 군집화는 군집화를 적용하고자 하는 데이터를 여러 개의 Gaussian 분포가 섞인 것으로 간주합니다.



먼저 다음과 같이 세 개의 gaussian 분포를 가진 A, B, C 데이터셋이 있다고 가정하겠습니다. 이 세 개의 정규분포를 합치면 아래 형태가 될 것입니다.



군집화를 수행하려는 실제 데이터셋의 분포도가 위와 같다면 이 데이터셋이 정규 분포 A, B, C 가 합쳐서 된 분포도임을 알 수 있습니다. 이처럼 전체 데이터셋은 서로 다른 gaussian 분포를 가진 여러 가지 확률 분포 곡선으로 구성될 수 있으며, 이러한 사실에 기반해 군집화를 수행하는 것이 GMM 군집화입니다. 가령 100 개의 데이터셋이 있다면 이를 구성하는 여러 개의 정규 분포 곡선을 추출하고, 개별 데이터가 이 중 어떤 정규 분포에 속하는지 결정하는 방식입니다. 이와 같은 방식은 GMM 에서는 모수 추정이라고 하는데, 모수 추정은 대표적으로 2 가지를 추정하는 것입니다.

- 개별 정규 분포의 평균과 분산

- 각 데이터가 어떤 정규 분포에 해당되는지의 확률

이러한 모수 추정을 위해 GMM 은 EM (Expectation and Maximization) 방법을 적용합니다.

붓꽃 데이터셋으로 GMM 을 통한 군집화를 진행해보도록 하겠습니다. scikit-learn 은 이러한 GMM 을 지원하기 위해 Gaussian Mixture 클래스를 지원합니다. Gaussian Mixture 객체의 가장 중요한 초기화 parameter 는 n_components 입니다. n_components 는 gaussian mixture 모델의 총 개수입니다. 이를 3 으로 설정하고 군집화를 수행해보도록 하겠습니다.

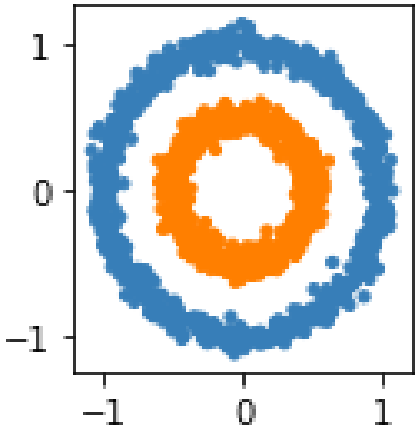
```
from sklearn.mixture import GaussianMixture

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
```

Target 0 는 cluster 0 으로 , Target 2 는 cluster 1 으로 모두 잘 매핑이 됐습니다. Target 1 만 분산되어 군집화되었으나 앞 절의 k-means 보다는 더 효과적인 분류 결과가 도출되었습니다. 하지만 이는 GMM 이 더 뛰어나다는 의미가 아니라 붓꽃 데이터셋에는 GMM 이 더 효과적이라는 의미입니다. K-means 는 평균 거리 중심으로 군집화를 수행하므로 개별 군집 내의 데이터가 원형으로 흩어져 있는 경우에 매우 효과적으로 군집화가 수행될 수 있습니다.

DBSCAN

DBSCAN 은 대표적인 밀도 기반 군집화 알고리즘입니다. DBSCAN 은 아래와 같이 데이터의 분포가 기하학적인 복잡한 데이터셋에도 효과적인 군집화가 가능합니다.



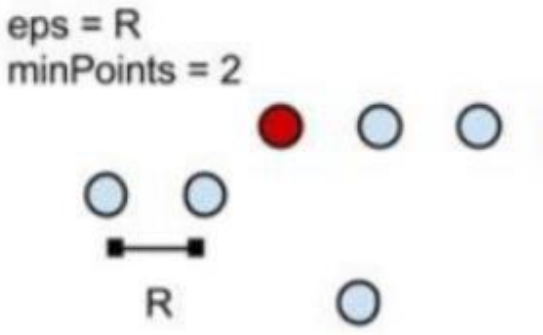
DBSCAN 을 구성하는 가장 중요한 두 가지 parameter 는 epsilon 과 min points 입니다.

- epsilon: 개별 데이터를 중심으로 입실론 반경을 갖는 원형의 영역
- min points: 개별 데이터의 입실론 주변 영역에 포함되는 타 데이터의 개수

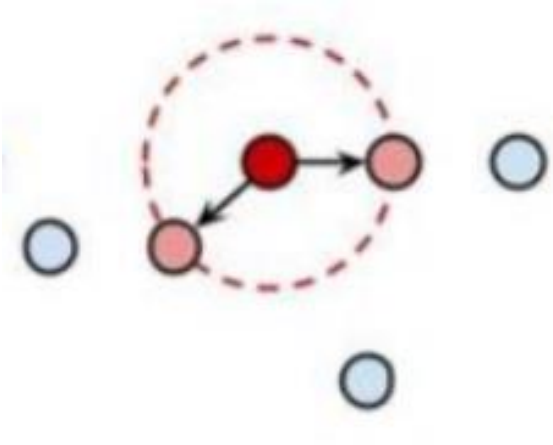
epsilon 주변 영역 내에 포함되는 최소 데이터 개수를 충족시키는가 아닌가에 따라 데이터 포인트를 다음과 같이 정의합니다.

- core point: 주변 영역 내에 min points 이상의 타 데이터를 가지고 있을 경우
- neighbor point: 주변 영역 내에 위치한 타 데이터
- border point: 주변 영역 내에 min points 이상의 neighbor point 를 가지고 있진 않지만 core point 를 neighbor point 로 가지고 있는 데이터
- noise point: min points 이상의 neighbor point 를 가지고 있지 않으며, core point 도 neighbor point 로 가지고 있지 않은 데이터

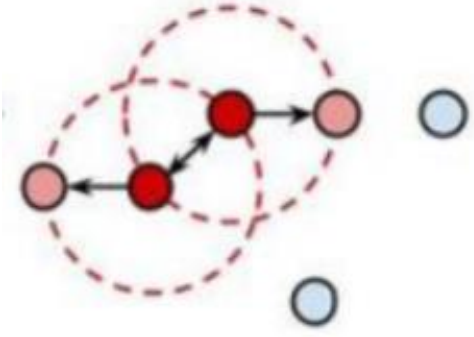
이를 보다 시각적으로 쉽게 설명하기 위하여 아래 예제를 보도록 하겠습니다. epsilon = R, min points = 2 인 상황을 가정하겠습니다.



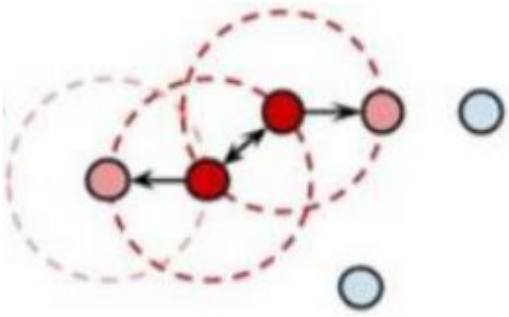
아래 p1 는 해당 기준을 만족하므로 core point 입니다.



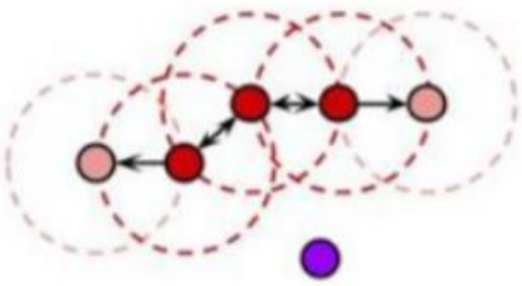
p2 역시 min points 기준을 만족하여 core point 입니다. core point p1 의 neighbor point p2 역시 core point 일 경우 p1 과 p2 를 연결해 직접 접근이 가능합니다.



p3 은 min points 기준을 만족시키지는 못하지만 neighbor point 로 core point 를 가지고 있습니다. 이러한 데이터를 border point 라고 하며, 군집의 외곽을 형성하게 됩니다.



p4 의 경우 noise point 라고 합니다.



scikit-learn 은 DBSCAN 클래스를 통해 DBSCAN 알고리즘을 지원합니다. 해당 클래스는 아래와 같은 주요한 초기화 parameter 를 가지고 있습니다.

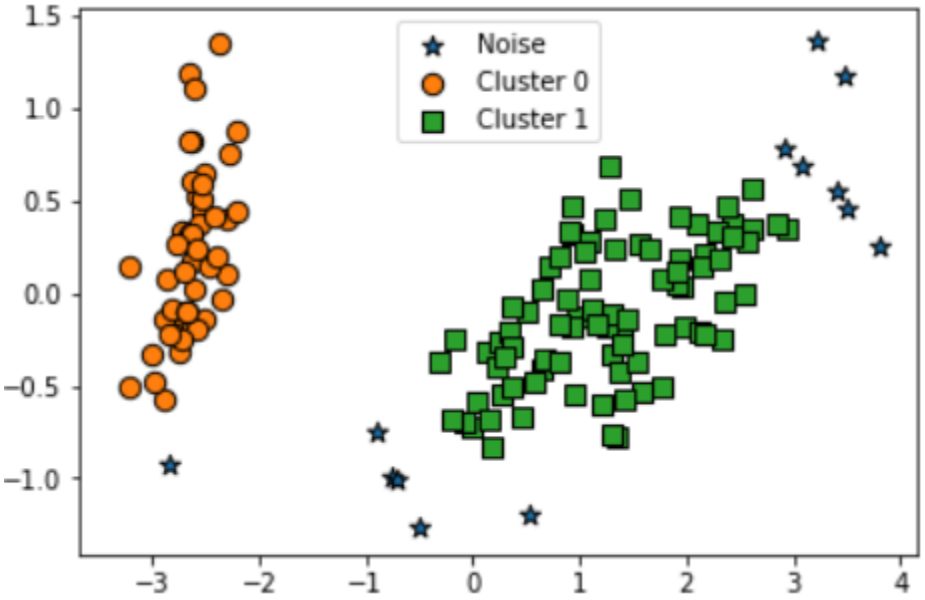
- eps: epsilon 주변 영역의 반경
- min_samples: core point 가 되기 위해 epsilon 주변 영역 내에 포함돼야할 최소 데이터 개수, 자기 자신을 포함하여 min points+1 에 해당

DBSCAN 알고리즘으로 붓꽃데이터셋을 군집화해보겠습니다. 이 예시에서는 eps=0.6, min_samples = 8 로 설정하겠습니다(일반적으로 eps 값으로는 1 이하의 값을 설정합니다).

```
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)
```

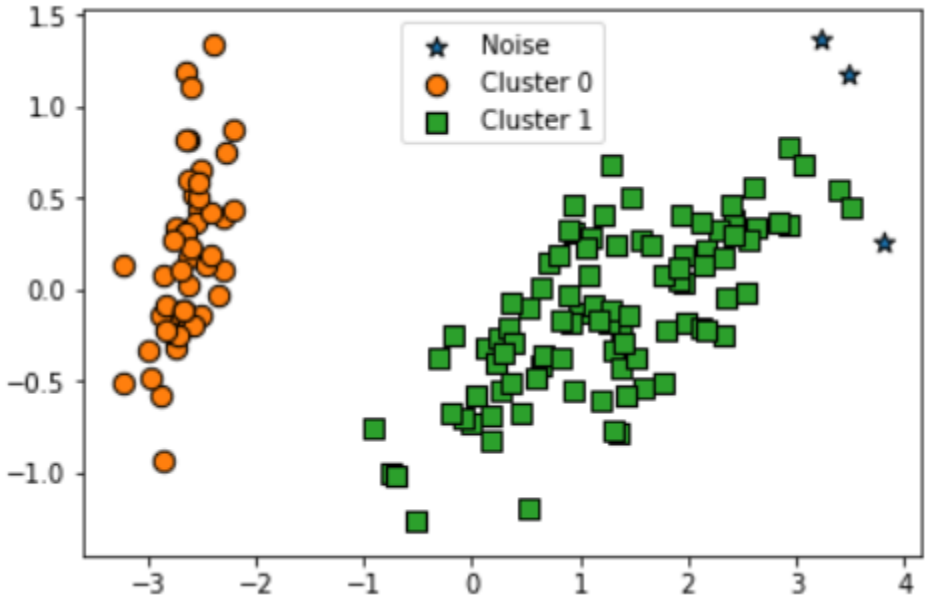
```
target  dbscan_cluster
0        0               49
        -1                1
1        1               46
        -1                4
2        1               42
        -1                8
```

dbscan_cluster 값을 살펴봤을 때, 특이하게 -1 이 군집 레이블로 있는 것을 확인할 수 있습니다. 군집 레이블이 -1 인 것은 노이즈에 속하는 군집을 의미합니다. 따라서 위 붓꽃데이터셋은 두 개의 군집으로 군집화됐습니다. 클러스터 결과를 표현하기 위해 붓꽃데이터셋을 PCA 를 이용해 2 개의 feature 로 압축한 뒤 시각화해 보겠습니다.

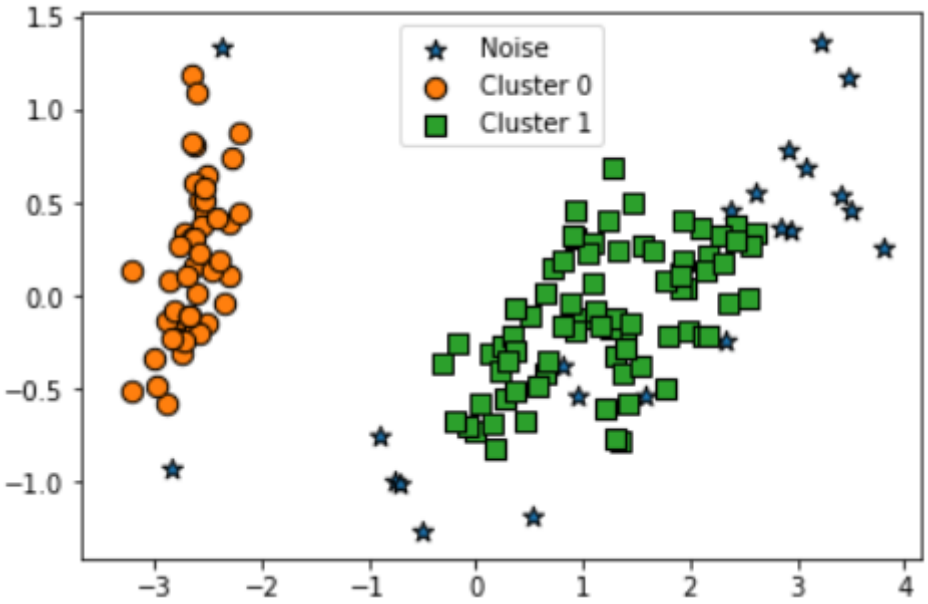


별표로 표현된 값은 모두 노이즈입니다. 일반적으로 eps 의 값을 크게 하면 반경이 커져 포함하는 데이터가 많아지므로 노이즈 데이터 개수가 작아집니다. min_samples 를 크게 하면

주어진 반경 내에 더 많은 데이터를 포함해야 하므로 노이즈 데이터 개수가 커지게 됩니다. 우선 앞 예시에서 epsilon 을 0.8 로 증가시켰을 때의 결과입니다.



노이즈가 줄어든 모습입니다. 다음은 min_samples 를 16 으로 늘린 결과입니다.

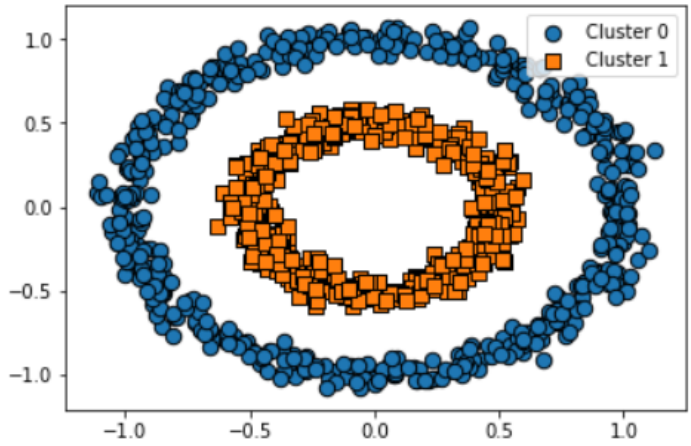


노이즈가 기존보다 증가였음을 확인할 수 있습니다.

이번에는 기하학적 분포를 가지는 데이터셋에서 DBSCAN 과 타 알고리즘을 비교해 보겠습니다. 먼저 make_circles( )함수를 이용해 내부 원과 외부 원으로 돼 있는 2 차원 데이터셋을 만들어

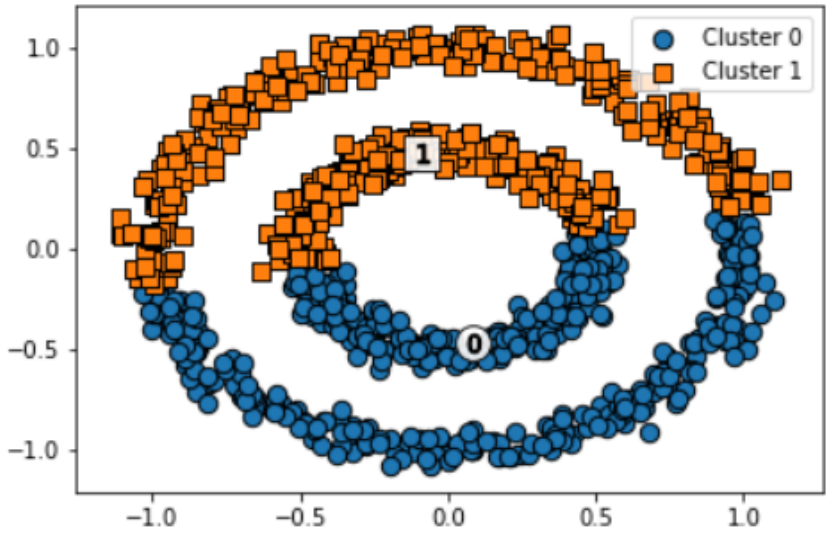
보겠습니다. factor parameter 는 내부 원과 외부 원의 scale 비율, noise parameter 는 노이즈 데이터셋의 비율입니다.

```
from sklearn.datasets import make_circles

X, y = make_circles(n_samples=1000, shuffle=True, noise=0.05, random_state=0, factor=0.5)
clusterDF = pd.DataFrame(data=X, columns=['ftr1', 'ftr2'])
clusterDF['target'] = y

visualize_cluster_plot(None, clusterDF, 'target', iscenter=False)
```



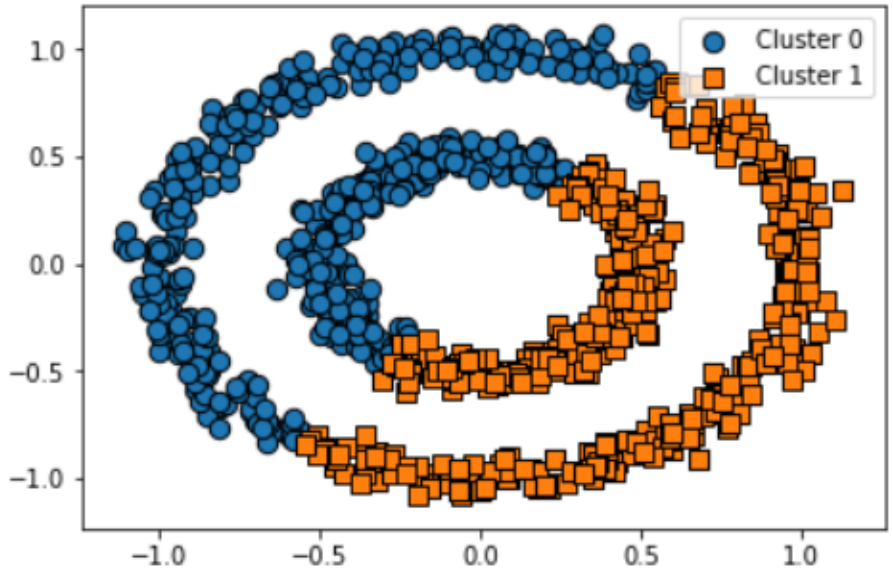
먼저 k-means 를 통한 군집화 결과입니다.

```python
# KMeans로 make_circles( ) 데이터 셋을 클러스터링 수행.
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=2, max_iter=1000, random_state=0)
kmeans_labels = kmeans.fit_predict(X)
clusterDF['kmeans_cluster'] = kmeans_labels

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



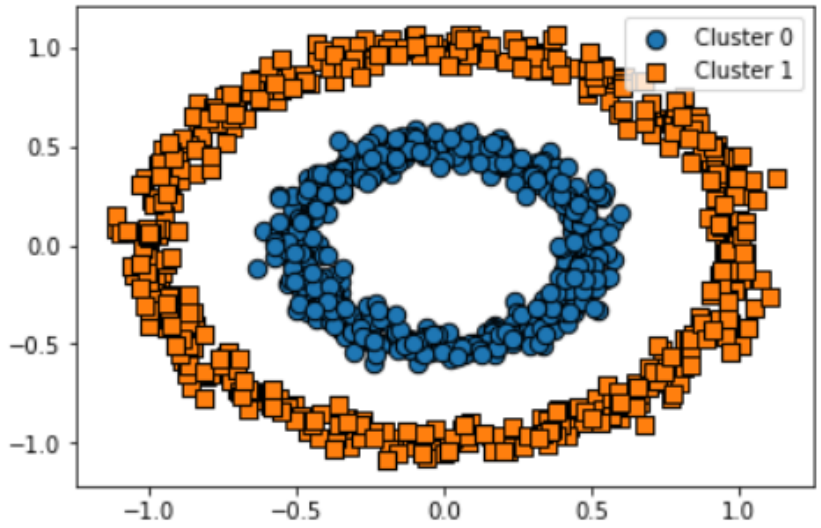
다음으로는 GMM 을 적용해 보겠습니다.

```
# GMM으로 make_circles( ) 데이터 셋을 클러스터링 수행.
from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n_components=2, random_state=0)
gmm_label = gmm.fit(X).predict(X)
clusterDF['gmm_cluster'] = gmm_label

visualize_cluster_plot(gmm, clusterDF, 'gmm_cluster', iscenter=False)
```



마지막으로 DBSCAN을 적용한 결과입니다.

```
# DBSCAN으로 make_circles( ) 데이터 셋을 클러스터링 수행.
from sklearn.cluster import DBSCAN

dbscan = DBSCAN(eps=0.2, min_samples=10, metric='euclidean')
dbscan_labels = dbscan.fit_predict(X)
clusterDF['dbscan_cluster'] = dbscan_labels

visualize_cluster_plot(dbscan, clusterDF, 'dbscan_cluster', iscenter=False)
```



DBSCAN 을 통해 원형태의 기하학적인 모형이 정확히 군집화가 되었음을 확인할 수 있습니다.

DBSCAN 실습 – 고객 segmentation

https://archive.ics.uci.edu/ml/datasets/online+retail