1. 군집화 분석 수행

1.1 데이터 수집

- 데이터 : HCV Data (출처 : https://archive.ics.uci.edu/ml/datasets/HCV+data)

- 데이터 설명 : 헌혈자와 C 형 간염 환자의 실험실 값과 인구 통계학적 값이 포함되었다.

- 변수 설명 : X (환자 ID),
  Category (진단) (값: '0=헌혈자', '0s=의심 헌혈자', '1=간염', '2=섬유증', '3=간경변증'),
  Age, Sex (f,m), ALB(알부민), ALP(알칼리인산분해효소), ALT(알라닌아미노전이효소),
  AST(아스파테이트아미노전이효소), BIL(빌리루빈), CHE(주화성), CHOL(콜레스테롤),
  CREA(크레아티닌), GGT(감마글루타밀전이효소), PROT(프로트롬빈)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1		Category	Age	Sex	ALB	ALP	ALT	AST	BIL	CHE	CHOL	CREA	GGT	PROT
2	1	0=Blood D	32	m	38.5	52.5	7.7	22.1	7.5	6.93	3.23	106	12.1	69
3	2	0=Blood D	32	m	38.5	70.3	18	24.7	3.9	11.17	4.8	74	15.6	76.5
4	3	0=Blood D	32	m	46.9	74.7	36.2	52.6	6.1	8.84	5.2	86	33.2	79.3
5	4	0=Blood D	32	m	43.2	52	30.6	22.6	18.9	7.33	4.74	80	33.8	75.7
6	5	0=Blood D	32	m	39.2	74.1	32.6	24.8	9.6	9.15	4.32	76	29.9	68.7
7	6	0=Blood D	32	m	41.6	43.3	18.5	19.7	12.3	9.92	6.05	111	91	74
8	7	0=Blood D	32	m	46.3	41.3	17.5	17.8	8.5	7.01	4.79	70	16.9	74.5
9	8	0=Blood D	32	m	42.2	41.9	35.8	31.1	16.1	5.82	4.6	109	21.5	67.1
10	9	0=Blood D	32	m	50.9	65.5	23.2	21.2	6.9	8.69	4.1	83	13.7	71.3

- 데이터 구성 : 615 개의 행과 13 개의 컬럼으로 이루어져있다. 실제 LABEL 이 들어있는 Category 와 범주형 변수인 Sex 와 11 개의 연속형 변수가 있다.

```
hcvdat.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 615 entries, 0 to 614
Data columns (total 14 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Unnamed: 0  615 non-null    int64
 1   Category    615 non-null    object
 2   Age         615 non-null    int64
 3   Sex         615 non-null    object
 4   ALB         614 non-null    float64
 5   ALP         597 non-null    float64
 6   ALT         614 non-null    float64
 7   AST         615 non-null    float64
 8   BIL         615 non-null    float64
 9   CHE         615 non-null    float64
 10  CHOL        605 non-null    float64
 11  CREA        615 non-null    float64
 12  GGT         615 non-null    float64
 13  PROT        614 non-null    float64
dtypes: float64(10), int64(2), object(2)
memory usage: 67.4+ KB
```

1.2 전처리

- 결측치가 있는 행을 삭제하였다. 약 20 개의 데이터만 삭제되었다.

```
hcvdat = hcvdat.dropna(axis=0)
hcvdat
```

	Unnamed: 0	Category	Age	Sex	ALB	ALP	ALT	AST	BIL	CHE	CHOL	CREA	GGT	PROT
0	1	0=Blood Donor	32	m	38.5	52.5	7.7	22.1	7.5	6.93	3.23	106.0	12.1	69.0
1	2	0=Blood Donor	32	m	38.5	70.3	18.0	24.7	3.9	11.17	4.80	74.0	15.6	76.5
2	3	0=Blood Donor	32	m	46.9	74.7	36.2	52.6	6.1	8.84	5.20	86.0	33.2	79.3
3	4	0=Blood Donor	32	m	43.2	52.0	30.6	22.6	18.9	7.33	4.74	80.0	33.8	75.7
4	5	0=Blood Donor	32	m	39.2	74.1	32.6	24.8	9.6	9.15	4.32	76.0	29.9	68.7

- Csv 파일을 읽어오며 생긴 index 열을 삭제하였다.

```
hcvdat = hcvdat.drop(columns = hcvdat.columns[0])
hcvdat.head()
```

	Category	Age	Sex	ALB	ALP	ALT	AST	BIL	CHE	CHOL	CREA	GGT	PROT
0	0=Blood Donor	32	m	38.5	52.5	7.7	22.1	7.5	6.93	3.23	106.0	12.1	69.0
1	0=Blood Donor	32	m	38.5	70.3	18.0	24.7	3.9	11.17	4.80	74.0	15.6	76.5
2	0=Blood Donor	32	m	46.9	74.7	36.2	52.6	6.1	8.84	5.20	86.0	33.2	79.3
3	0=Blood Donor	32	m	43.2	52.0	30.6	22.6	18.9	7.33	4.74	80.0	33.8	75.7
4	0=Blood Donor	32	m	39.2	74.1	32.6	24.8	9.6	9.15	4.32	76.0	29.9	68.7

- 실제 Label 이 들어있는 Category 는 뒤에 값 설명을 제외한 숫자 부분만 추출하였다.

```
hcvdat['Category'] = hcvdat['Category'].str[0:1]
hcvdat.head()
```

	Category	Age	Sex	ALB	ALP	ALT	AST	BIL	CHE	CHOL	CREA	GGT	PROT
0	0	32	m	38.5	52.5	7.7	22.1	7.5	6.93	3.23	106.0	12.1	69.0
1	0	32	m	38.5	70.3	18.0	24.7	3.9	11.17	4.80	74.0	15.6	76.5
2	0	32	m	46.9	74.7	36.2	52.6	6.1	8.84	5.20	86.0	33.2	79.3
3	0	32	m	43.2	52.0	30.6	22.6	18.9	7.33	4.74	80.0	33.8	75.7
4	0	32	m	39.2	74.1	32.6	24.8	9.6	9.15	4.32	76.0	29.9	68.7

- 범주형 변수인 Sex 는 원핫인코딩을 사용하여 변환하였다.

```
ohe = OneHotEncoder()
hcvdat['Sex'] = ohe.fit_transform(hcvdat[['Sex']]).toarray()
hcvdat.head()
```

	Category	Age	Sex	ALB	ALP	ALT	AST	BIL	CHE	CHOL	CREA	GGT	PROT
0	0	32	0.0	38.5	52.5	7.7	22.1	7.5	6.93	3.23	106.0	12.1	69.0
1	0	32	0.0	38.5	70.3	18.0	24.7	3.9	11.17	4.80	74.0	15.6	76.5
2	0	32	0.0	46.9	74.7	36.2	52.6	6.1	8.84	5.20	86.0	33.2	79.3
3	0	32	0.0	43.2	52.0	30.6	22.6	18.9	7.33	4.74	80.0	33.8	75.7
4	0	32	0.0	39.2	74.1	32.6	24.8	9.6	9.15	4.32	76.0	29.9	68.7

- 앞서, 결측치가 데이터에서 삭제되며 index 가 연속적이지 않아,
index 를 리셋시켰다.

```
hcvdat = hcvdat.reset_index()
hcvdat = hcvdat.drop(columns = hcvdat.columns[0])
hcvdat.head()
```

	Category	Age	Sex	ALB	ALP	ALT	AST	BIL	CHE	CHOL	CREA	GGT	PROT
0	0	32	0.0	38.5	52.5	7.7	22.1	7.5	6.93	3.23	106.0	12.1	69.0
1	0	32	0.0	38.5	70.3	18.0	24.7	3.9	11.17	4.80	74.0	15.6	76.5
2	0	32	0.0	46.9	74.7	36.2	52.6	6.1	8.84	5.20	86.0	33.2	79.3
3	0	32	0.0	43.2	52.0	30.6	22.6	18.9	7.33	4.74	80.0	33.8	75.7
4	0	32	0.0	39.2	74.1	32.6	24.8	9.6	9.15	4.32	76.0	29.9	68.7

- Category 변수의 이름을 Class 로 변경하였다.

```
hcvdat.rename(columns={'Category':'Class'}, inplace=True)
hcvdat
```

	Class	Age	Sex	ALB	ALP	ALT	AST	BIL	CHE	CHOL	CREA	GGT	PROT
0	0	32	0.0	38.5	52.5	7.7	22.1	7.5	6.93	3.23	106.0	12.1	69.0
1	0	32	0.0	38.5	70.3	18.0	24.7	3.9	11.17	4.80	74.0	15.6	76.5
2	0	32	0.0	46.9	74.7	36.2	52.6	6.1	8.84	5.20	86.0	33.2	79.3
3	0	32	0.0	43.2	52.0	30.6	22.6	18.9	7.33	4.74	80.0	33.8	75.7
4	0	32	0.0	39.2	74.1	32.6	24.8	9.6	9.15	4.32	76.0	29.9	68.7

- 실제 라벨이 있는 Class 변수를 제외한 모든 변수들을 StandardScaler 로 전처리하여 PCA 로 차원을 축소하였다.

```
X_features = hcvdat[['Age','Sex','ALB','ALP','ALT','AST','BIL','CHE','CHOL','CREA','GGT','PROT']].values
X_features_scaled = StandardScaler().fit_transform(X_features)

pca = PCA(n_components=2)
pca_transformed = pca.fit_transform(X_features_scaled)
dataframe = pd.DataFrame(pca_transformed, columns=['PCA1','PCA2'])
dataframe['Class']=hcvdat['Class']
```

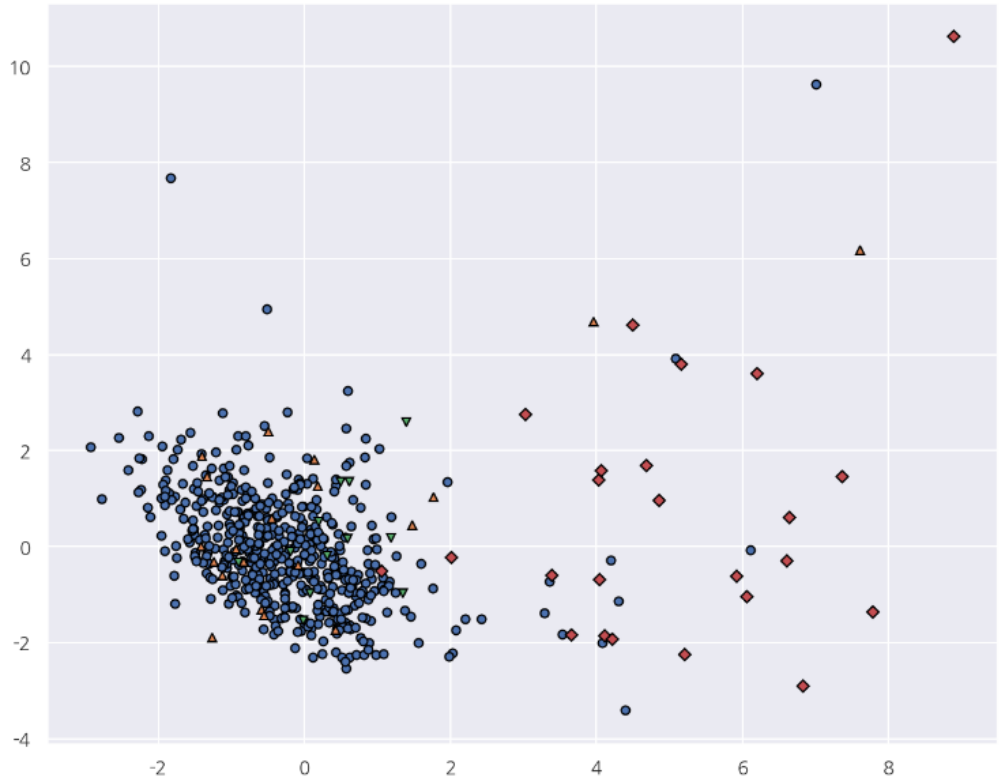
- 군집화에 필요한 열들을 데이터프레임으로 만들고, 열 이름을 각각 X, Y, Class 로 변경하였다.

```
dataframe.columns=['X','Y','Class']
Artificial_Dataset={'HCVData':dataframe}
```

1.3 군집화 분석 결과 및 가시화

1.3.1 산점도 가시화

군집화에 앞서, 2 차원 상에 산점도를 가시화하였다.



가시화 결과 Label 0 의 수가 나머지 Label 들의 수에 비해 높다는 것을 확인하였다.

```
hcvdat.Category.value_counts()
```

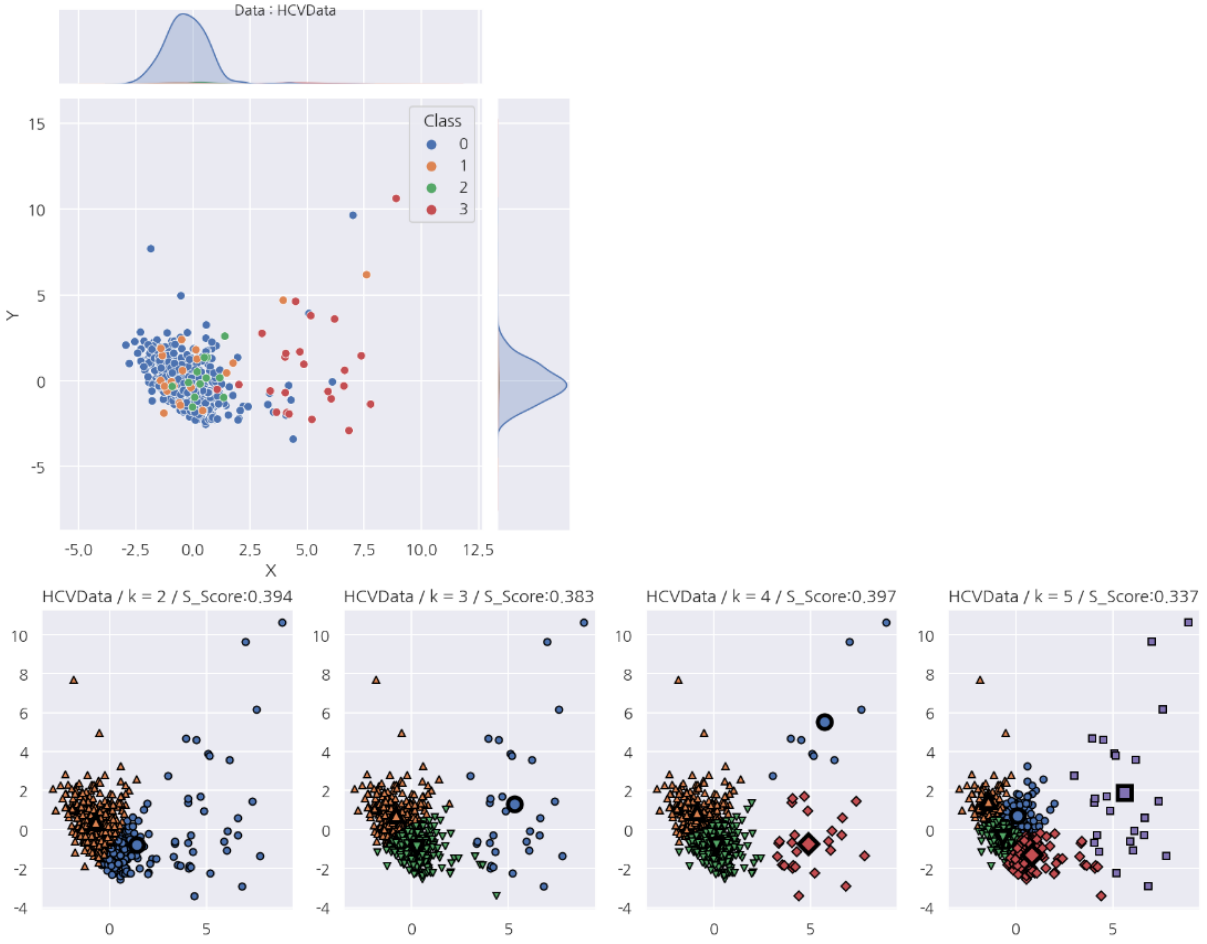
```
0=Blood Donor                533
3=Cirrhosis                   30
1=Hepatitis                   24
2=Fibrosis                    21
0s=suspect Blood Donor         7
Name: Category, dtype: int64
```

1.3.2 K-Means

```
def k_Means_Plot(Data, Select_k, NAME, Init_Method = 'k-means++', Num_Init=10):
    Data2 = Data[['X', 'Y']]
    fig, axes = plt.subplots(1, (np.max(list(Select_k))-np.min(list(Select_k)))+1, figsize=(15, 4))
    for i in Select_k:
        Kmeans_Clustering = KMeans(n_clusters=i, init=Init_Method, random_state=2020, n_init=Num_Init)
        Kmeans_Clustering.fit(Data2)
        mglearn.discrete_scatter(Data2['X'], Data2['Y'], Kmeans_Clustering.labels_, ax=axes[i - 2], s=5)
        mglearn.discrete_scatter(Kmeans_Clustering.cluster_centers_[:, 0],
                                 Kmeans_Clustering.cluster_centers_[:, 1],
                                 list(range(i)),
                                 markeredgewidth=3,
                                 ax=axes[i - 2], s=10)
        Score = np.round(silhouette_score(Data2, Kmeans_Clustering.labels_),3)
        axes[i - 2].set_title( NAME + ' / k = ' + str(i)+' / S_Score:'+str(Score))
```

- 'n_init' = 1 일 때, k 가 4 일 때의 silhouette score 가 0.397 로 가장 높았지만, 다른 군집화 기법들에 비해 낮은 silhouette score 로 군집화 결과가 좋진 않다.

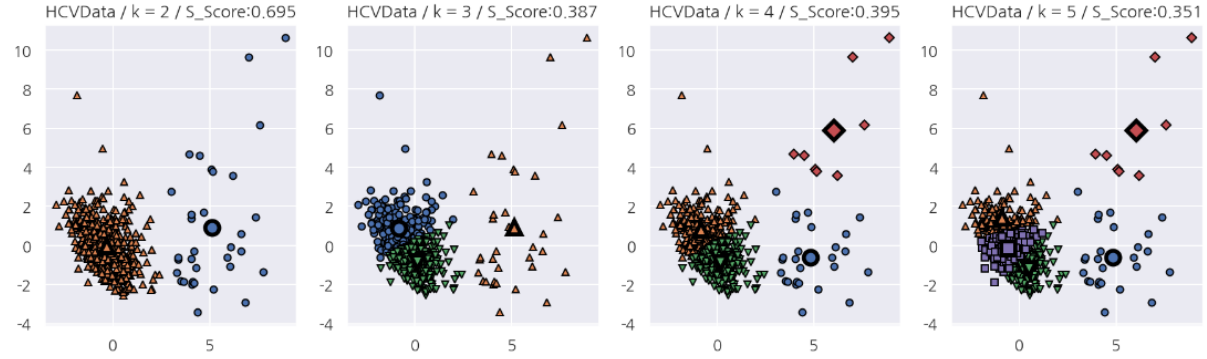
```
for i in range(0,1):
    Simple_Scatter(i, list(Artificial_Dataset.keys())[i])

    k_Means_Plot(Data = Artificial_Dataset[list(Artificial_Dataset.keys())[i]],
                 Select_k = range(2, 6),
                 NAME = list(Artificial_Dataset.keys())[i],
                 Init_Method='random',Num_Init=1)
```



- 'n_init' = 10 일 때, k 가 2 일 때의 silhouette score 가 0.695 로 가장 높아 좋은 결과를 보인다. 하지만 실제 label 엔 4 개의 label 중 한 개의 label 의 비율이 너무

높아, k 가 2 일 때의 silhouette score 가 높게 나온 것으로 판단되어 군집화가 잘 된 것은 아니다.

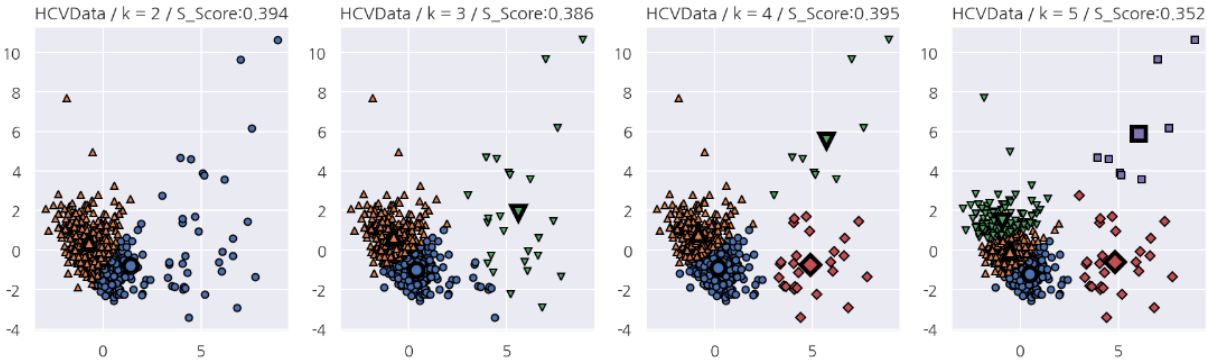
```
for i in range(0,1):
    Simple_Scatter(i, list(Artificial_Dataset.keys())[i])

    k_Means_Plot(Data = Artificial_Dataset[list(Artificial_Dataset.keys())[i]],
                 Select_k = range(2, 6),
                 NAME = list(Artificial_Dataset.keys())[i],
                 Init_Method='random',Num_Init=10)
```



1.3.3 K-Means++

k 가 4 일 때의 silhouette score 가 0.395 로 가장 높아 좋은 결과를 보인다. K-Means++보단 K-Means 를 사용하였을 때의 결과가 더 좋았다.

```
for i in range(0,1):
    Simple_Scatter(i, list(Artificial_Dataset.keys())[i])

    k_Means_Plot(Data = Artificial_Dataset[list(Artificial_Dataset.keys())[i]],
                 Select_k = range(2, 6),
                 NAME = list(Artificial_Dataset.keys())[i],
                 Init_Method='k-means++',Num_Init=1)
```

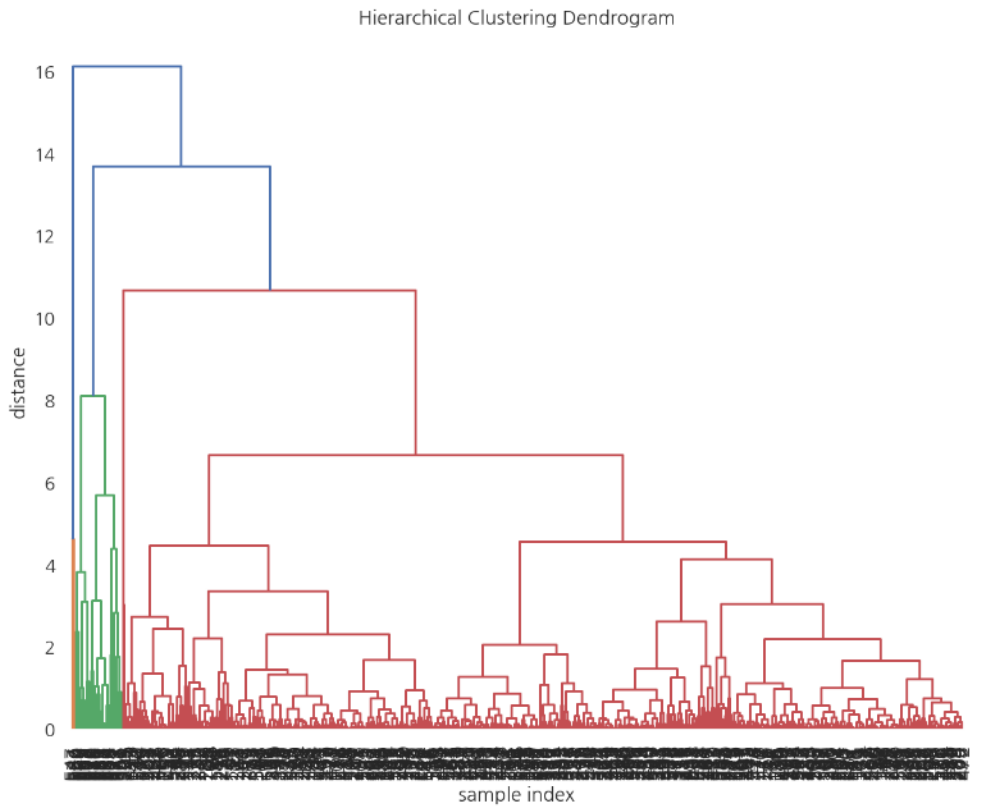


1.3.4 Hierarchical clustering

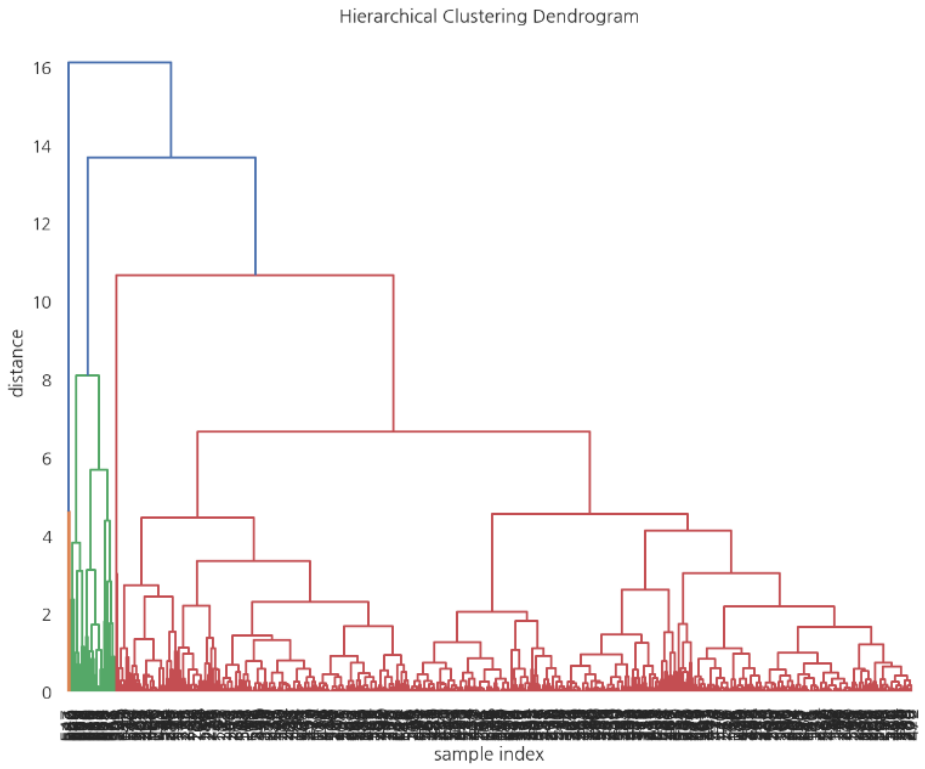
p 가 2 일 때의 silhouette score 가 0.811 로 가장 높아 좋은 결과를 보인다. 하지만 K-Means 를 사용할 때와 마찬가지로 실제 label 엔 4 개의 label 중 한 개의 label 의 비율이 너무 높아, k 가 2 일 때의 silhouette score 가 높게 나온 것으로 판단되어 군집화가 잘 된 것은 아니다.

```
def Fixed_Dendrogram(Data, Num_of_p, Full_Use):
    Linkage_Matrix = linkage(Data,'complete')
    if(Full_Use == True):
        Num_of_p = np.shape(Data)[0]
        plt.title('Hierarchical Clustering Dendrogram')
        plt.xlabel('sample index')
    else:
        plt.title('Hierarchical Clustering Dendrogram (truncated)')
        plt.xlabel('sample index or (cluster size)')
    plt.ylabel('distance')
    dendrogram(
        Linkage_Matrix,
        truncate_mode = 'lastp',
        p = Num_of_p,
        leaf_rotation = 90.,
        leaf_font_size = 12.,
        color_threshold = 'default'
    )
    plt.show()
```

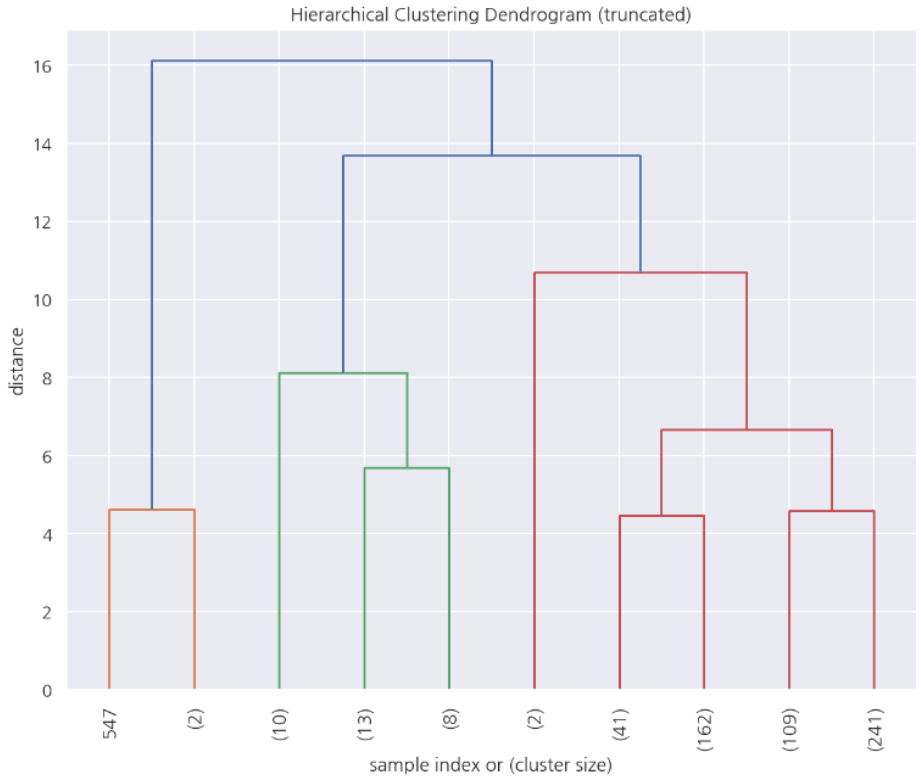
```
Fixed_Dendrogram(dataframe[['X','Y']], -1, True) # -1: nomeaningful value
```



```
Fixed_Dendrogram(dataframe[['X','Y']], 20, True)
```

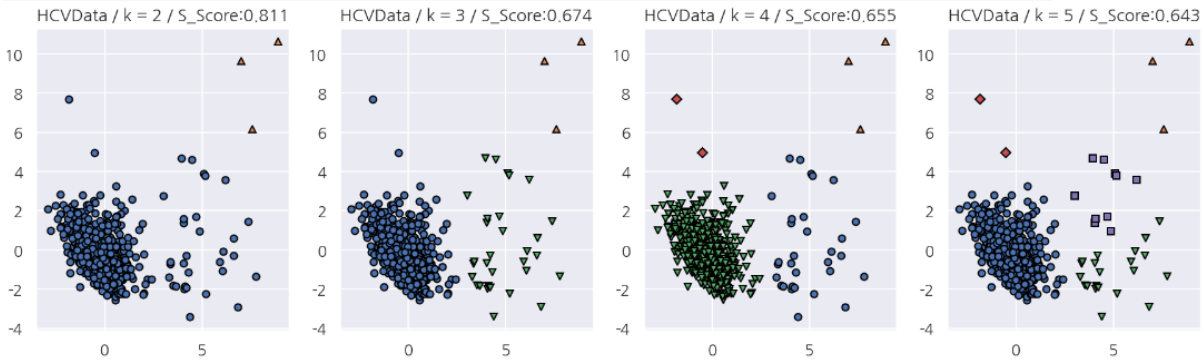



```
a = Fixed_Dendrogram(dataframe[['X','Y']], 10, False)
```



```
def Hclust_Plot(Data,Select_k,NAME):
    Data2 = Data[['X', 'Y']]
    fig, axes = plt.subplots(1, (np.max(list(Select_k))-np.min(list(Select_k)))+1, figsize=(15, 4))
    for i in Select_k:
        H_Clustering = AgglomerativeClustering(n_clusters=i,linkage="complete")
        P_Labels = H_Clustering.fit_predict(Data2)
        mglearn.discrete_scatter(Data2['X'], Data2['Y'], P_Labels, ax=axes[i - 2], s=5)
        axes[i - 2].set_title("Data:" + NAME + ' / k = ' + str(i))
        Score=np.round(silhouette_score(Data2,P_Labels),3)
        axes[i - 2].set_title( NAME + ' / k = ' + str(i)+' / S_Score:'+str(Score))
```

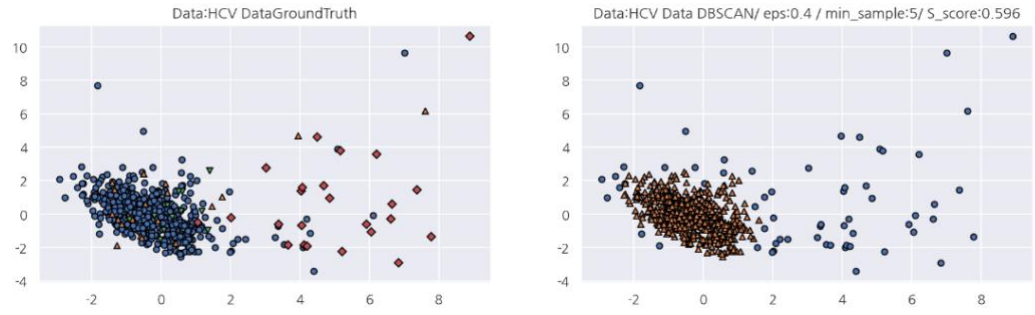
```
for i in range(0,1):
    Simple_Scatter(i, list(Artificial_Dataset.keys())[i])
    Hclust_Plot(Artificial_Dataset[list(Artificial_Dataset.keys())[i]],
                range(2, 6), list(Artificial_Dataset.keys())[i])
```



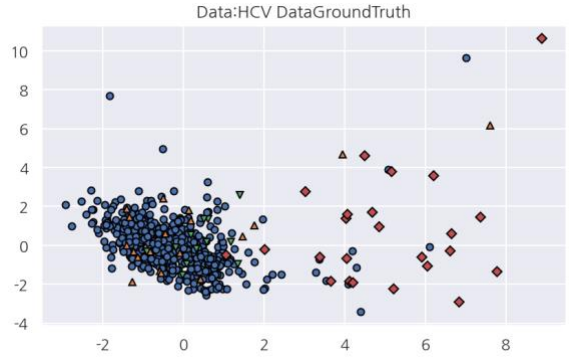
1.3.5 DBSCAN

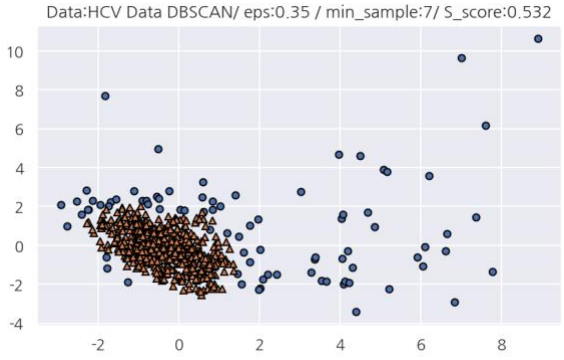
Eps 가 0.4, min_samplaes 가 5 일 때의 silhouette score 가 0.596 으로 가장 좋은 결과를 보인다. 전체적으로 앞의 다른 군집화 기법들보다 군집화 결과가 좋았다.

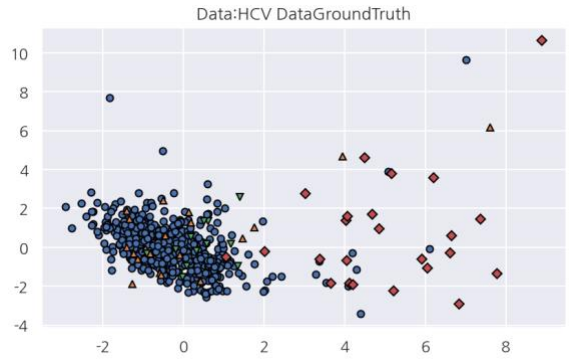
```
def DBSCAN_Plot(Data,NAME,min_samples=5,eps=0.4):
    Data2 = Data[['X', 'Y']]
    Append_k_Means_Results = list()
    fig, axes = plt.subplots(1, 2, figsize=(15, 4))
    Set_DBSCAN_Hyperparameter=DBSCAN(min_samples=min_samples,eps=eps)
    Results = Set_DBSCAN_Hyperparameter.fit_predict(Data2)
    Score=np.round(silhouette_score(Data2,Results),3)
    mglearn.discrete_scatter(Data2['X'], Data2['Y'], Data['Class'], ax=axes[0], s=5)
    axes[0].set_title("Data:" + NAME + 'GroundTruth')
    mglearn.discrete_scatter(Data2['X'], Data2['Y'], Results, ax=axes[1], s=5)
    axes[1].set_title("Data:" + NAME + ' DBSCAN/ eps:'+str(eps)+' / min_sample:'+str(min_samples)+'/ S_score:'+str(Score))


DBSCAN_Plot(Data=dataframe,NAME="HCV Data")
```

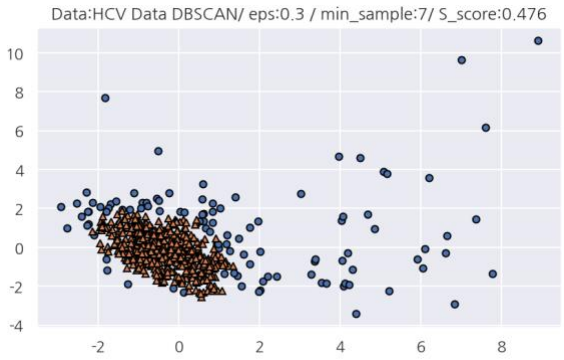


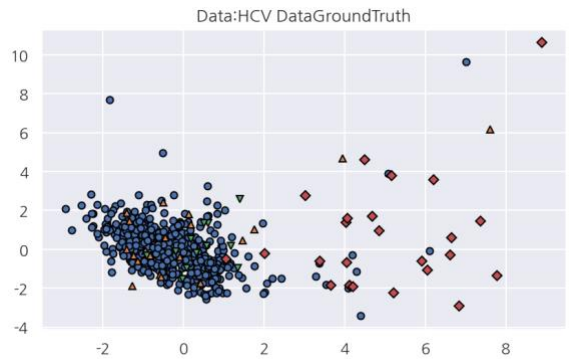
```
DBSCAN_Plot(Data=dataframe,NAME="HCV Data",min_samples=7,eps=0.35)
DBSCAN_Plot(Data=dataframe,NAME="HCV Data",min_samples=7,eps=0.30)
DBSCAN_Plot(Data=dataframe,NAME="HCV Data",min_samples=7,eps=0.25)
DBSCAN_Plot(Data=dataframe,NAME="HCV Data",min_samples=7,eps=0.40)
DBSCAN_Plot(Data=dataframe,NAME="HCV Data",min_samples=5,eps=0.35)
DBSCAN_Plot(Data=dataframe,NAME="HCV Data",min_samples=5,eps=0.30)
DBSCAN_Plot(Data=dataframe,NAME="HCV Data",min_samples=5,eps=0.25)
DBSCAN_Plot(Data=dataframe,NAME="HCV Data",min_samples=5,eps=0.40)
```

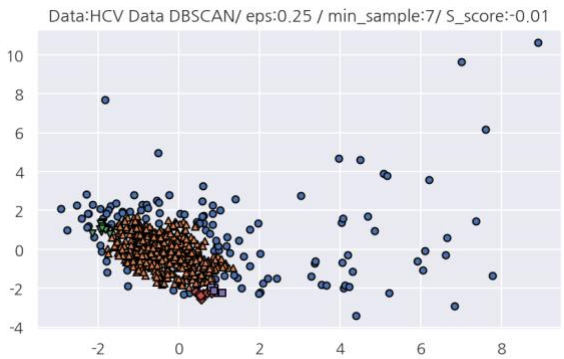


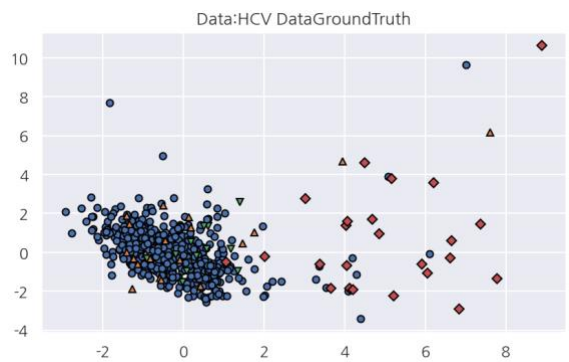


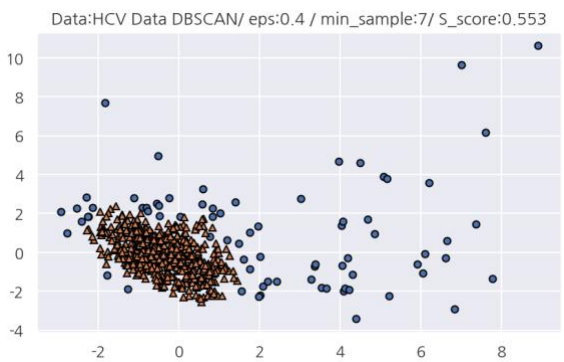


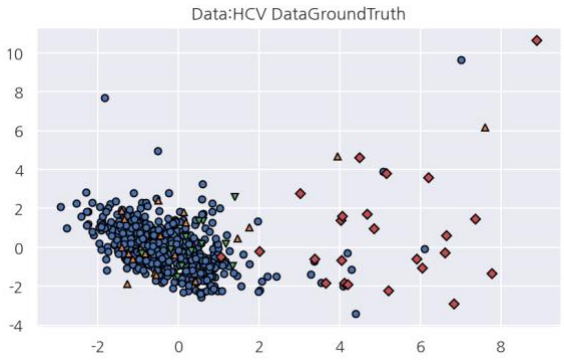


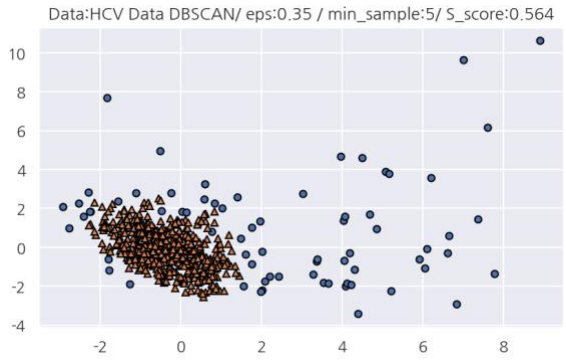


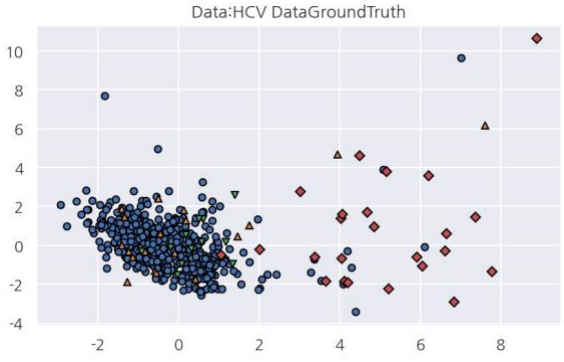


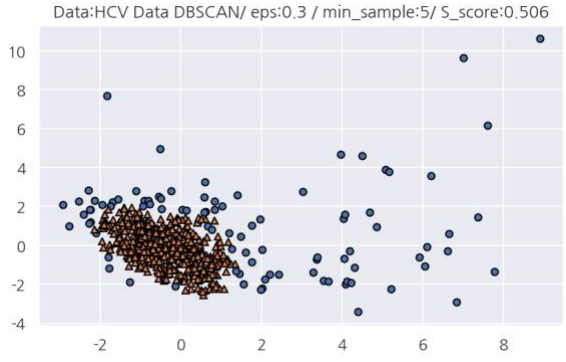


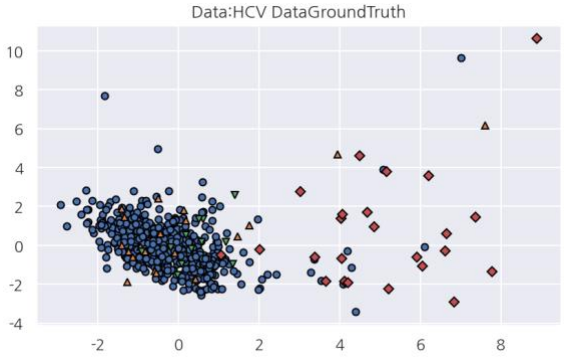


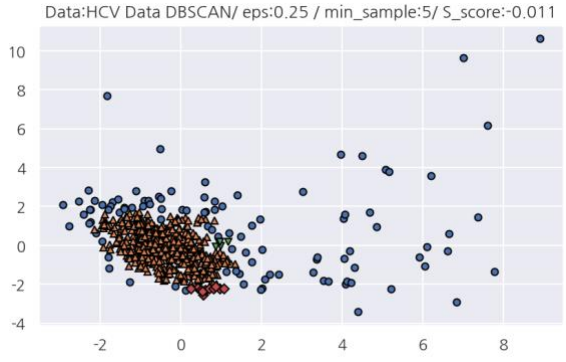

Data:HCV DataGroundTruth

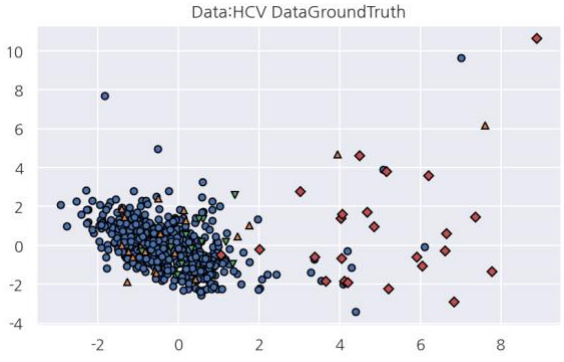

Data:HCV Data DBSCAN/ eps:0.35 / min_sample:5/ S_score:0.564

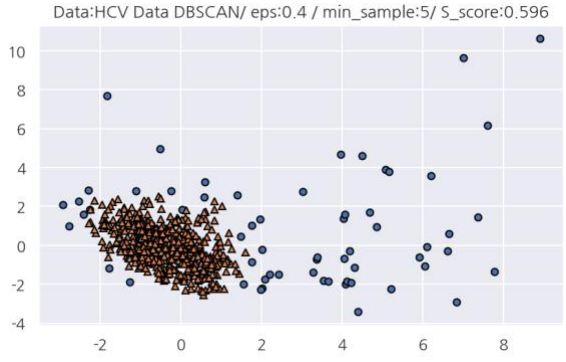

Data:HCV DataGroundTruth


Data:HCV Data DBSCAN/ eps:0.3 / min_sample:5/ S_score:0.506


Data:HCV DataGroundTruth


Data:HCV Data DBSCAN/ eps:0.25 / min_sample:5/ S_score:-0.011


Data:HCV DataGroundTruth


Data:HCV Data DBSCAN/ eps:0.4 / min_sample:5/ S_score:0.596

2. 의사결정나무 분석 수행

2.1 데이터 수집 및 전처리

- 데이터 : Wall-Following Robot Navigation Data (출처 : https://archive.ics.uci.edu/ml/datasets/Wall-Following+Robot+Navigation+Data)

- 데이터 설명 : SCITOS G5 로봇이 허리 주위에 원형으로 배열된 24 개의 초음파 센서를 사용하여 4 라운드 동안 벽을 따라 시계 방향으로 방을 탐색하면서 수집된 데이터이다.

- 변수 설명 : US1(로봇 전면의 초음파 센서, 180Â°), US2(-165Â°), US3(-150Â°), US4(-135Â°), US5(-120Â°), US6(-105Â°), US7(-90Â°), US8(-75Â°), US9(-60Â°), US10(45Â°), US11(-30Â°), US12(-15Â°), US13(로봇 전면의 초음파 센서, 0Â°, US14(15Â°), US15(30Â°), US16(45Â°), US17(60Â°), US18(75Â°), US19(90Â°), US20(105Â°), US21(120Â°), US22(135Â°), US23(150Â°), US24(165Â°)Class(Move-Forward, Slight-Right-Turn, Sharp-Right-Turn, Slight-Left-Turn)

	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2.918	5	2.351	2.332	2.643	1.698	1.687	1.698	1.717	1.744	0.593	0.502	0.493	0.504	0.445	0.431	0.444	0.44	0.429	Slight-Right-Turn	
2	2.918	5	2.637	2.332	2.649	1.695	1.687	1.695	1.72	1.744	0.592	0.502	0.493	0.504	0.449	0.431	0.444	0.443	0.429	Slight-Right-Turn	
3	2.918	5	2.637	2.334	2.643	1.696	1.687	1.695	1.717	1.744	0.593	0.502	0.493	0.504	0.449	0.431	0.444	0.446	0.429	Slight-Right-Turn	
4	2.918	5	2.353	2.334	2.642	1.73	1.687	1.695	1.717	1.744	0.593	0.502	0.493	0.504	0.449	0.431	0.444	0.444	0.429	Slight-Right-Turn	
5	2.918	5	2.64	2.334	2.639	1.696	1.687	1.695	1.717	1.744	0.592	0.502	0.493	0.504	0.449	0.431	0.444	0.441	0.429	Slight-Right-Turn	

- 데이터 구성 : 5456 개의 행과 25 개의 컬럼으로 이루어져있다. 실제 LABEL 이 들어있는 Class 와 24 개의 연속형 변수가 있다.

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5456 entries, 0 to 5455
Data columns (total 25 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   US1     5456 non-null   float64
 1   US2     5456 non-null   float64
 2   US3     5456 non-null   float64
 3   US4     5456 non-null   float64
 4   US5     5456 non-null   float64
 5   US6     5456 non-null   float64
 6   US7     5456 non-null   float64
 7   US8     5456 non-null   float64
 8   US9     5456 non-null   float64
 9   US10    5456 non-null   float64
 10  US11    5456 non-null   float64
 11  US12    5456 non-null   float64
 12  US13    5456 non-null   float64
 13  US14    5456 non-null   float64
 14  US15    5456 non-null   float64
 15  US16    5456 non-null   float64
 16  US17    5456 non-null   float64
 17  US18    5456 non-null   float64
 18  US19    5456 non-null   float64
 19  US20    5456 non-null   float64
 20  US21    5456 non-null   float64
 21  US22    5456 non-null   float64
 22  US23    5456 non-null   float64
 23  US24    5456 non-null   float64
 24  Class   5456 non-null   object
dtypes: float64(24), object(1)
memory usage: 1.0+ MB
```

- 기존 데이터에 변수의 이름이 저장되어있지 않아 열 이름을 따로 지정하였다.

```
data.columns = ['US1', 'US2', 'US3', 'US4', 'US5', 'US6', 'US7', 'US8', 'US9', 'US10', 'US11', 'US12', 'US13', 'US14', 'US15', 'US16
data
```

	US1	US2	US3	US4	US5	US6	US7	US8	US9	US10	...	US16	US17	US18	US19	US20	US21	US22	US23	US24	Class
0	0.438	0.498	3.625	3.645	5.000	2.918	5.000	2.351	2.332	2.643	...	0.593	0.502	0.493	0.504	0.445	0.431	0.444	0.440	0.429	Slight-Right-Turn
1	0.438	0.498	3.625	3.648	5.000	2.918	5.000	2.637	2.332	2.649	...	0.592	0.502	0.493	0.504	0.449	0.431	0.444	0.443	0.429	Slight-Right-Turn
2	0.438	0.498	3.625	3.629	5.000	2.918	5.000	2.637	2.334	2.643	...	0.593	0.502	0.493	0.504	0.449	0.431	0.444	0.446	0.429	Slight-Right-Turn
3	0.437	0.501	3.625	3.626	5.000	2.918	5.000	2.353	2.334	2.642	...	0.593	0.502	0.493	0.504	0.449	0.431	0.444	0.444	0.429	Slight-Right-Turn
4	0.438	0.498	3.626	3.629	5.000	2.918	5.000	2.640	2.334	2.639	...	0.592	0.502	0.493	0.504	0.449	0.431	0.444	0.441	0.429	Slight-Right-Turn

- Target 데이터를 기존 데이터와 분리하였다.

```
target=data['Class']
target
```

```
0         Slight-Right-Turn
1         Slight-Right-Turn
2         Slight-Right-Turn
3         Slight-Right-Turn
4         Slight-Right-Turn
                ...
5451          Move-Forward
5452     Sharp-Right-Turn
5453     Sharp-Right-Turn
5454          Move-Forward
5455     Sharp-Right-Turn
Name: Class, Length: 5456, dtype: object
```

```
data = data.drop(columns = data.columns[24])
data
```

	US1	US2	US3	US4	US5	US6	US7	US8	US9	US10	...	US15	US16	US17	US18	US19	US20	US21	US22	US23	US24
0	0.438	0.498	3.625	3.645	5.000	2.918	5.000	2.351	2.332	2.643	...	1.744	0.593	0.502	0.493	0.504	0.445	0.431	0.444	0.440	0.429
1	0.438	0.498	3.625	3.648	5.000	2.918	5.000	2.637	2.332	2.649	...	1.744	0.592	0.502	0.493	0.504	0.449	0.431	0.444	0.443	0.429
2	0.438	0.498	3.625	3.629	5.000	2.918	5.000	2.637	2.334	2.643	...	1.744	0.593	0.502	0.493	0.504	0.449	0.431	0.444	0.446	0.429
3	0.437	0.501	3.625	3.626	5.000	2.918	5.000	2.353	2.334	2.642	...	1.744	0.593	0.502	0.493	0.504	0.449	0.431	0.444	0.444	0.429
4	0.438	0.498	3.626	3.629	5.000	2.918	5.000	2.640	2.334	2.639	...	1.744	0.592	0.502	0.493	0.504	0.449	0.431	0.444	0.441	0.429

- 이상치(극단값) 확인 : 이상치가 이미 데이터에 5% 이상 존재하여 따로 생성하지 않았다.

```
quartile_1 = data[:].quantile(0.25)
quartile_3 = data[:].quantile(0.75)
IQR = quartile_3 - quartile_1
IQR
```

```
US1     0.89300
US2     1.31950
US3     1.20050
US4     2.36250
US5     2.54050
US6     2.04950
US7     3.06925
US8     1.57500
US9     3.20025
US10    1.89025
US11    1.63725
US12    0.98850
US13    1.16700
US14    1.70825
US15    3.57625
US16    0.46900
US17    0.33200
US18    0.30725
US19    0.33400
US20    0.32125
US21    0.43525
US22    1.32525
US23    0.76750
US24    0.77325
dtype: float64
```

```
outlier_data = []

for i in range(0,24) :
    outlier = data[(data[data.columns[i]] < (quartile_1[i] - 1.5 * IQR[i])) | (data[data.columns[i]] > (quartile_3[i] + 1.5 * IQR[i]
    outlier_ratio = len(outlier) / len(data)
    outlier_data.append(outlier_ratio)

outlier_data
```

```
[0.03610703812316716,
 0.17705278592375367,
 0.16000733137829912,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.0,
 0.15175953079178886,
 0.16788856304985336,
 0.0,
 0.0,
 0.09200879765395895,
 0.09567448680351906,
 0.10245601173020528,
 0.13306451612903225,
 0.13874633431085043,
 0.08797653958944282,
 0.18328445747800587,
 0.1315982404692082,
 0.10043988269794721]
```

2.2 의사결정나무 구축 및 실험비교, 가시화 및 설명

- 의사결정나무 구축 : 데이터가 과적합되어있어 depth 의 수가 너무 많아 해당 의사결정나무에 대한 해석은 생략하고, 가지치기 후의 의사결정나무에 대한 해석을 아래에서 서술하였다.

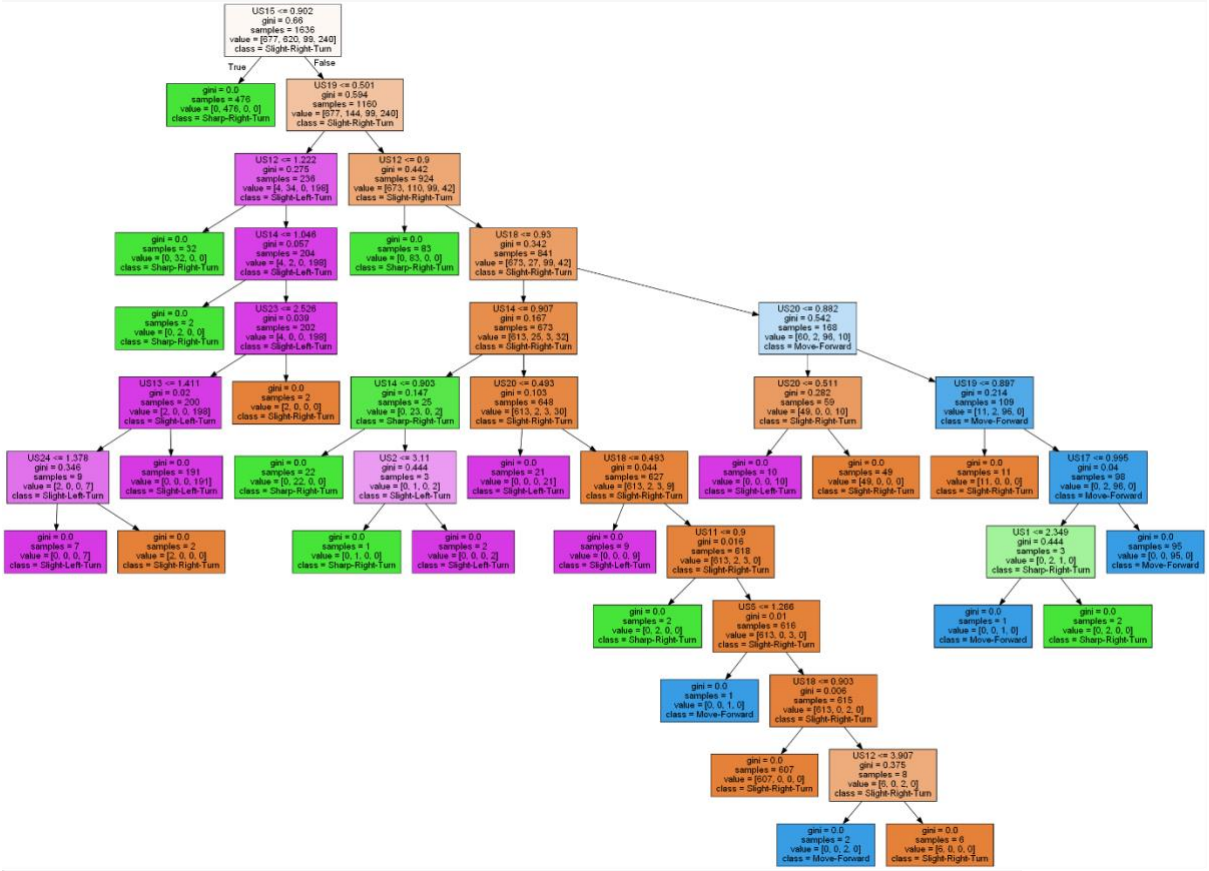
```
seed = 5456

x_train, x_test, y_train, y_test = train_test_split(data, target, test_size=0.7, random_state=seed)
```

```
dt_clf = DecisionTreeClassifier(random_state=5456)
dt_clf.fit(x_train, y_train)
```

```
DecisionTreeClassifier(random_state=5456)
```

```
export_graphviz(dt_clf, out_file="tree.dot", class_names = target_names, feature_names = feature_names, impurity=True, filled=True)
```

```
print('[ max_depth의 제약이 없는 경우의 Decision Tree 시각화 ]')
with open("tree.dot") as f:
    dot_graph = f.read()
graphviz.Source(dot_graph)
```



```
graphviz.Source(dot_graph).render('tree', format="png")
```

```
'tree.png'
```

```
y_pred = dt_clf.predict(x_test)
```

```
print("Train_Accuracy : ", dt_clf.score(x_train, y_train), '₩n')
print("Test_Accuracy : ", dt_clf.score(x_test, y_test), '₩n')

accuracy = mt.accuracy_score(y_test, y_pred)
recall = mt.recall_score(y_test, y_pred, average='micro')
precision = mt.precision_score(y_test, y_pred, average='micro')
f1_score = mt.f1_score(y_test, y_pred, average='micro')
matrix = mt.confusion_matrix(y_test, y_pred)

print('Accuracy: ', format(accuracy,'.2f'),'₩n')
print('Recall: ', format(recall,'.2f'),'₩n')
print('Precision: ', format(precision,'.2f'),'₩n')
print('F1_score: ', format(f1_score,'.2f'),'₩n')
print('Confusion Matrix:','₩n', matrix)
```

```
Train_Accuracy :  1.0

Test_Accuracy :  0.9863874345549738

Accuracy:  0.99

Recall:  0.99

Precision:  0.99

F1_score:  0.99

Confusion Matrix:
 [[1504    1   13   10]
 [   5 1471    1    0]
 [   0    1  228    0]
 [  20    1    0  565]]
```

```
# 교차검증

x = data
y = target

# 각 폴드의 스코어
scores = cross_val_score(dt_clf, x, y, cv = 5)

print('Averaged results of cross validation: ', scores.mean())
```

```
Averaged results of cross validation:  0.9770934687066678
```

```
pd.DataFrame(cross_validate(dt_clf, x, y, cv =5))
```

	fit_time	score_time	test_score
0	0.041938	0.001999	0.955128
1	0.039901	0.001996	0.980752
2	0.043881	0.001991	0.971586
3	0.040891	0.002003	0.994500
4	0.039825	0.003048	0.983501

```
# test set에 대한 스코어(정확도)
dt_clf.score(x_test, y_test)
```

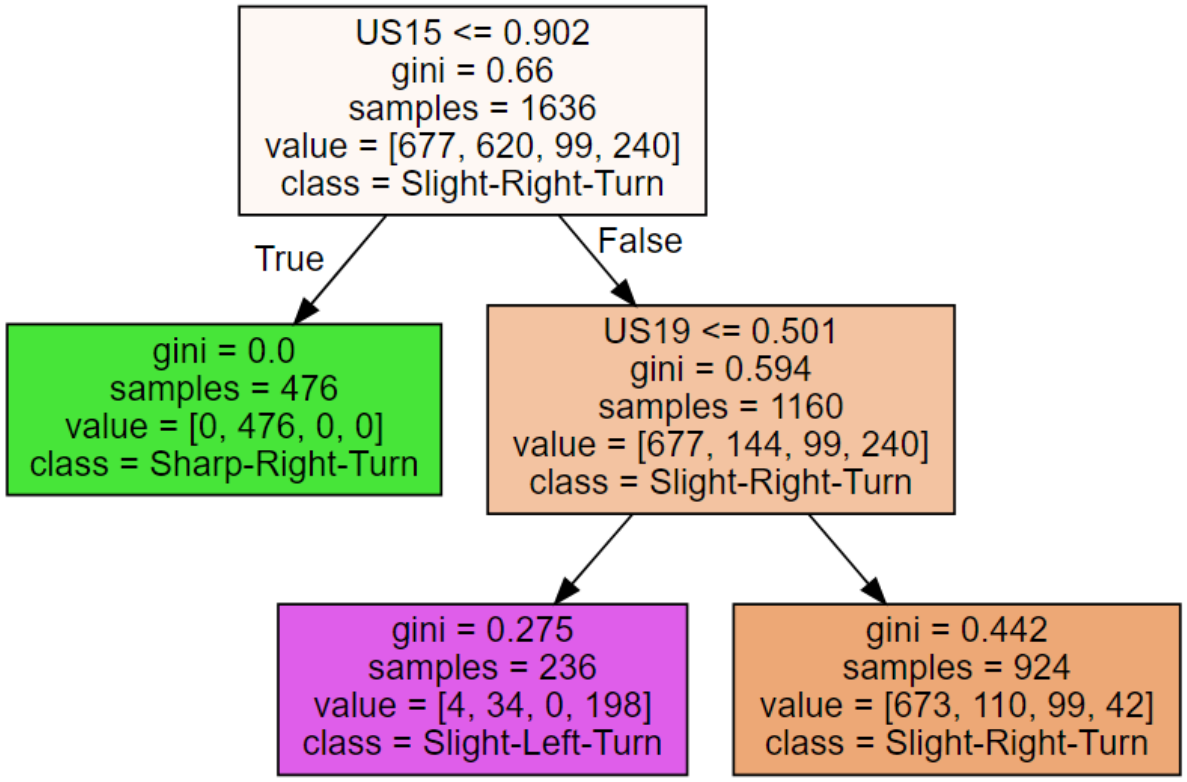
```
0.9863874345549738
```

- 가지치기 수행 : 가지치기 수행 전보다 test set 에 대한 정확도가 떨어졌지만, 가지치기 수행 전엔 과적합이 되어있어 정확도가 높았다고 판단된다. (0.98 -> 0.82) 이 의사결정 나무에 따르면, 많은 데이터가 Slight-Right-Turn 으로 분류되고, 476 개의 데이터가 Sharp-Right-Turn(US15 센서의 값이 0.902 보다 작거나 같을 경우)으로 분류되며, 236 개의 데이터만이 Slight-Left-Turn(US15 센서의 값이 0.902 보다 크고, US19 센서의 값이 0.501 보다 작거나 같을 경우)으로 분류된다.

```
pruned_dt_clf = DecisionTreeClassifier(max_depth=2, random_state=156) # max_depth=3으로 제한
pruned_dt_clf .fit(x_train, y_train)

print("Accuracy of training set: {:.3f}".format(pruned_dt_clf.score(x_train, y_train)))
print("Accuracy of test set: {:.3f}".format(pruned_dt_clf.score(x_test, y_test)))
```

```
Accuracy of training set: 0.823
Accuracy of test set: 0.821
```

```
# export_graphviz( )의 호출 결과로 out_file로 지정된 tree.dot 파일을 생성함
export_graphviz(pruned_dt_clf, out_file="prunedtree.dot", class_names = target_names, feature_names = feature_names, impurity=True,
print('[ max_depth가 2인 경우의 Decision Tree 시각화 ]')
# 위에서 생성된 tree.dot 파일을 Graphiviz 가 읽어서 시각화
with open("prunedtree.dot") as f:
    dot_graph = f.read()
graphviz.Source(dot_graph)
```



```
Train_Accuracy :  0.8233496332518337

Test_Accuracy :  0.8206806282722513

Accuracy:  0.82

Recall:  0.82

Precision:  0.82

F1_score:  0.82

Confusion Matrix:
 [[1523    0    0    5]
 [ 269 1136    0   72]
 [ 229    0    0    0]
 [ 109    1    0  476]]
 Averaged results of cross validation:  0.8200150750563215
```

	fit_time	score_time	test_score
0	0.021550	0.001994	0.817766
1	0.017954	0.002032	0.834097
2	0.015960	0.001011	0.813016
3	0.017965	0.001948	0.818515
4	0.016987	0.001006	0.816682

```
# test set에 대한 스코어(정확도)
pruned_dt_clf.score(x_test, y_test)
```

```
0.8206806282722513
```

2.3 의사결정나무의 특징 코멘트

- 데이터에서 이상치를 확인해본 결과, 데이터의 절반이 넘는 행이 이상치를 최소 한 개 이상 갖고 있었음에도 불구하고, 정확도가 높은 것을 보아, 의사결정나무는 rank 를 이용한 모델이기 때문에 이상치에 민감하지 않다.
- 가지치기 수행 전의 의사결정나무는 과적합이 있어 정확도가 굉장히 높았는데, 가지치기를 수행하며 과적합이 해결되었다.
- 가지치기 수행 결과 max_depth 에 낮은 수를 제한할수록 과적합이 해결되었지만 max_depth 가 2 일 경우엔 정확도가 너무 낮아졌고 모델이 적합하지 않을 것으로 예상하여, max_depth 가 3~4 일 경우가 가장 이상적일 것이다.

```
pruned_dt_clf = DecisionTreeClassifier(max_depth=3, random_state=156) # max_depth=3으로 제한
pruned_dt_clf .fit(x_train, y_train)

print("Accuracy of training set: {:.3f}".format(pruned_dt_clf.score(x_train, y_train)))
print("Accuracy of test set: {:.3f}".format(pruned_dt_clf.score(x_test, y_test)))
```

```
Accuracy of training set: 0.894
Accuracy of test set: 0.896
```

```
pruned_dt_clf = DecisionTreeClassifier(max_depth=4, random_state=156) # max_depth=4으로 제한
pruned_dt_clf .fit(x_train, y_train)

print("Accuracy of training set: {:.3f}".format(pruned_dt_clf.score(x_train, y_train)))
print("Accuracy of test set: {:.3f}".format(pruned_dt_clf.score(x_test, y_test)))
```

```
Accuracy of training set: 0.917
Accuracy of test set: 0.915
```

```
pruned_dt_clf = DecisionTreeClassifier(max_depth=5, random_state=156) # max_depth=5으로 제한
pruned_dt_clf .fit(x_train, y_train)

print("Accuracy of training set: {:.3f}".format(pruned_dt_clf.score(x_train, y_train)))
print("Accuracy of test set: {:.3f}".format(pruned_dt_clf.score(x_test, y_test)))
```

```
Accuracy of training set: 0.962
Accuracy of test set: 0.955
```