기계학습 중간고사 대체 과제

1. Clustering
2. Decision Tree

제 출 일 2022-05-08
학 번 2018204072
이 름 김재윤
담 당 교 수 이상민
담 당 조 교 강민정, 이유진

광운대학교
KwangWoon University

“ Clustering ”

1. 데이터 수집

Clustering 실습을 위해 사용된 데이터는 “Dry Bean Dataset” (https://archive.ics.uci.edu/ml/datasets/Dry+Bean+Dataset) 이다. 해당 Dataset은 건조한 콩 13,611알에 대하여 성분 정보를 담고 있다. 콩의 품종을 결정하는 16개의 독립변수를 가지고 있으며, 콩의 품종을 나타내는 7개의 고유하는 값은 해당 데이터셋의 Class이다.

Dry Bean Dataset Data Set

Download: Data Folder, Data Set Description

Abstract: Images of 13,611 grains of 7 different registered dry beans were taken with a high-resolution camera. A total of 16 features; 12 dimensions and 4 shape forms, were obtained from the grains.

Data Set Characteristics:	Multivariate	**Number of Instances:**	13611	**Area:**	Computer
Attribute Characteristics:	Integer, Real	**Number of Attributes:**	17	**Date Donated**	2020-09-14
Associated Tasks:	Classification	**Missing Values?**	N/A	**Number of Web Hits:**	2002000

2. 데이터 형태 파악

필수 라이브러리

```
import os
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
from sklearn.cluster import AgglomerativeClustering
from sklearn.cluster import DBSCAN
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.metrics import silhouette_score
import collections
import mglearn
import matplotlib.pyplot as plt
import random

import warnings
warnings.filterwarnings(action='ignore')

random.seed(2020)
from sklearn.decomposition import PCA
from yellowbrick.cluster import KElbowVisualizer
from yellowbrick.cluster.elbow import kelbow_visualizer
```

1) 데이터 셋 소개

```
# 데이터 로드
df = pd.read_csv('Dry_Bean_Dataset.csv')
df.head()
```

	Area	Perimeter	MajorAxisLength	MinorAxisLength	AspectRation	Eccentricity	ConvexArea	EquivDiameter	Extent	Solidity	roundness	Compactness	ShapeFactor1	ShapeFactor2	ShapeFactor3	ShapeFactor4	Class
0	28395	610.291	208.178117	173.888747	1.197191	0.549812	28715	190.141097	0.763923	0.988856	0.958027	0.913358	0.007332	0.003147	0.834222	0.998724	SEKER
1	28734	638.018	200.524796	182.734419	1.097356	0.411785	29172	191.272751	0.783968	0.984986	0.887034	0.953861	0.006979	0.003564	0.909851	0.998430	SEKER
2	29380	624.110	212.826130	175.931143	1.209713	0.562727	29690	193.410904	0.778113	0.989559	0.947849	0.908774	0.007244	0.003048	0.825871	0.999066	SEKER
3	30008	645.884	210.557999	182.516516	1.153638	0.498616	30724	195.467062	0.782681	0.976696	0.903936	0.928329	0.007017	0.003215	0.861794	0.994199	SEKER
4	30140	620.134	201.847882	190.279279	1.060798	0.333680	30417	195.896503	0.773098	0.990893	0.984877	0.970516	0.006697	0.003665	0.941900	0.999166	SEKER

- 위 두 그림은 Clustering 실습에 사용될 필수 라이브러리의 import 과정과 Pandas를 활용해 데이터셋을 프레임의 형태로 가져오는 것을 보여준다.

```
print(df.info())
df.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13611 entries, 0 to 13610
Data columns (total 17 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Area             13611 non-null  int64
 1   Perimeter        13611 non-null  float64
 2   MajorAxisLength  13611 non-null  float64
 3   MinorAxisLength  13611 non-null  float64
 4   AspectRation     13611 non-null  float64
 5   Eccentricity     13611 non-null  float64
 6   ConvexArea       13611 non-null  int64
 7   EquivDiameter    13611 non-null  float64
 8   Extent           13611 non-null  float64
 9   Solidity         13611 non-null  float64
 10  roundness        13611 non-null  float64
 11  Compactness      13611 non-null  float64
 12  ShapeFactor1     13611 non-null  float64
 13  ShapeFactor2     13611 non-null  float64
 14  ShapeFactor3     13611 non-null  float64
 15  ShapeFactor4     13611 non-null  float64
 16  Class            13611 non-null  object
dtypes: float64(14), int64(2), object(1)
memory usage: 1.8+ MB
None
```

	Area	Perimeter	MajorAxisLength	MinorAxisLength	AspectRation	Eccentricity	ConvexArea
count	13611.000000	13611.000000	13611.000000	13611.000000	13611.000000	13611.000000	13611.000000
mean	53048.284549	855.283459	320.141867	202.270714	1.583242	0.750895	53768.200206
std	29324.095717	214.289696	85.694186	44.970091	0.246678	0.092002	29774.915817
min	20420.000000	524.736000	183.601165	122.512653	1.024868	0.218951	20684.000000
25%	36328.000000	703.523500	253.303633	175.848170	1.432307	0.715928	36714.500000
50%	44652.000000	794.941000	296.883367	192.431733	1.551124	0.764441	45178.000000
75%	61332.000000	977.213000	376.495012	217.031741	1.707109	0.810466	62294.000000
max	254616.000000	1985.370000	738.860154	460.198497	2.430306	0.911423	263261.000000

- info() 함수를 활용해 변수의 type을 포함한 요약정보를 확인할 수 있다.
- describe() 함수를 통해 연속형 변수들의 분포와 통계량을 확인할 수 있다.

```
print("데이터 셋의 관측치 수와 Class 수는 아래와 같다.")
print("Dry Bean :",np.shape(df)[0],"/ Class : ", df['Class'].nunique())
```

```
데이터 셋의 관측치 수와 Class 수는 아래와 같다.
Dry Bean : 13611 / Class :  7
```

- 위 그림의 코드를 통해 콩의 품종에 해당하는 종속변수 'Class'의 수는 7개가 있음을 확인할 수 있다.

3. 전처리

2-1) 범주형 변수에 대한 처리

```
Label = df['Class'].to_frame()
df.drop(['Class'], axis=1, inplace=True)
```

```
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()
Label['Class'] = encoder.fit_transform(Label['Class'].values)
```

- 해당 데이터셋에서는 종속변수인 'Class'만 범주형 변수에 해당된다. 이는 단지 Label을 의미하기 때문에 **Label Encoding**을 적용하여 숫자값으로 mapping하였다.

2-2) Scailing

```
# 표준화, 정규화
col_names = df.columns
from sklearn.preprocessing import StandardScaler
scaler_std = StandardScaler()
df_standard = scaler_std.fit_transform(df)

from sklearn.preprocessing import MinMaxScaler
scaler_nor = MinMaxScaler()
norm_array = scaler_nor.fit_transform(df)
df_norm = pd.DataFrame(norm_array, columns=col_names)

df_norm
```

	Area	Perimeter	MajorAxisLength	MinorAxisLength	AspectRation	Eccentricity	ConvexArea	EquivDiameter	Extent	Solidity	roundness	Compactness	ShapeFactor1	ShapeFactor2	ShapeFactor3	ShapeFactor4
0	0.034053	0.058574	0.044262	0.152142	0.122612	0.477797	0.033107	0.070804	0.671024	0.922824	0.934823	0.786733	0.593432	0.833049	0.750996	0.980620
1	0.035500	0.077557	0.030479	0.178337	0.051577	0.278472	0.034991	0.073577	0.735504	0.871514	0.793138	0.903549	0.547447	0.967316	0.884987	0.974979
2	0.038259	0.068035	0.052633	0.158190	0.131521	0.496448	0.037126	0.078816	0.716671	0.932141	0.914511	0.773514	0.582016	0.800942	0.736200	0.987196
3	0.040940	0.082942	0.048548	0.177691	0.091623	0.403864	0.041389	0.083854	0.731365	0.761614	0.826871	0.829912	0.552408	0.854744	0.799846	0.893675
4	0.041504	0.065313	0.032862	0.200679	0.025565	0.165680	0.040123	0.084906	0.700538	0.949832	0.988408	0.951583	0.510741	1.000000	0.941770	0.989116
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
13606	0.092559	0.160862	0.189318	0.187843	0.375584	0.788553	0.089967	0.172180	0.512286	0.942381	0.852151	0.465175	0.531785	0.382135	0.412185	0.974113
13607	0.092576	0.159358	0.176450	0.201964	0.321303	0.746241	0.089910	0.172207	0.786890	0.947954	0.862952	0.523974	0.509582	0.426233	0.470848	0.970912
13608	0.092739	0.160605	0.176384	0.203370	0.318558	0.743877	0.090219	0.172463	0.561689	0.936648	0.855785	0.525351	0.508683	0.427019	0.472240	0.943025
13609	0.092773	0.163657	0.179703	0.200669	0.330472	0.753971	0.090623	0.172517	0.482741	0.908991	0.834795	0.510145	0.514216	0.415330	0.456919	0.913342
13610	0.092824	0.169448	0.200882	0.176768	0.423337	0.819877	0.090347	0.172598	0.751569	0.933322	0.795826	0.416526	0.550320	0.346892	0.364762	0.970162

13611 rows × 16 columns

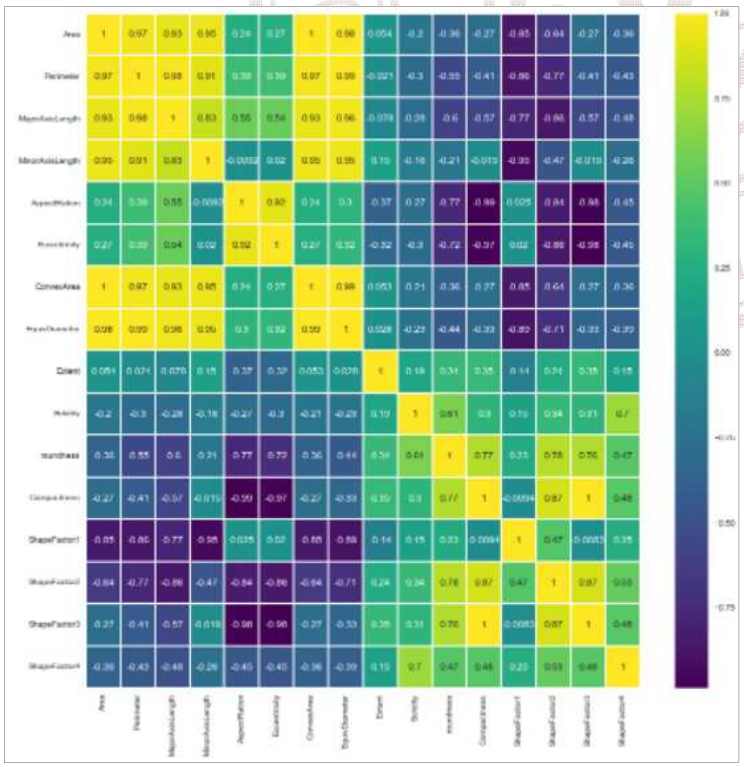
!! describe() 함수를 통해 데이터 분포를 확인하였으며, 해당 데이터셋은 Scailing 과정이 필요하고 판단할 수 있다. 또한 Clustering을 위한 많은 모델은 거리정보를 기반으로 군집화하기 때문에 해당 실습을 진행하기 위해서는 Scailing 과정이 **필수적**이라고 말할 수 있다.

2-3) 독립변수들간 상관성 분석

```
corr = df_norm.corr()
fig, ax = plt.subplots(figsize = (15,15))
sns.heatmap(corr, ax = ax, cmap = 'viridis', linewidth = 0.1, annot=True)
```

```
<AxesSubplot:>
```

```
df_norm.drop(['ConvexArea', 'ShapeFactor3'], axis=1, inplace=True)
```



- Seaborn라이브러리에서 제공하는 Heatmap을 통해 독립변수들간의 **상관성을 파악**할 수 있다.
- 많은 변수들은 비슷한 특징과 의미를 가진다는 사실을 알 수 있었으며, 상관계수를 1로 갖는 변수들에 대해서는 오른쪽 상단의 코드와 같이 하나씩 제거하는 과정을 진행하였다.

4. Clustering Analysis

4.1) K-means

3-1) K-Means

```
# K-means 모델링 / Scatter plot 함수 구현
def k_Means_Plot(Data, Select_k, NAME, Init_Method = 'k-means++', Num_Init=10):
    Data2 = Data[['P1', 'P2']]
    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(df_norm)
        mglearn.discrete_scatter(Data2['P1'], Data2['P2'], Kmeans_Clustering.labels_, ax=axes[i - 4], s=5)
        mglearn.discrete_scatter(Kmeans_Clustering.cluster_centers_[:, 0],
                                 Kmeans_Clustering.cluster_centers_[:, 1],
                                 list(range(i)),
                                 markeredgewidth=3,
                                 ax=axes[i - 4], s=10)
        Score = np.round(silhouette_score(Data2, Kmeans_Clustering.labels_),3)
        axes[i - 4].set_title( NAME + ' / k = ' + str(i)+' / S_Score:'+str(Score))
```

- 위 그림의 코드는 실습 데이터셋에 대하여 k-means 모델을 구축하고, 구축된 모델을 mglearn 라이브러리에서 제공하는 scatterplot을 통해 Test해보고 실제 Label과 비교해볼 수 있도록 구현된 함수이다.
- 모델링에 적용되는 데이터프레임은 df_norm이라는 것을 확인할 수 있다.

K-means 모델을 구축하기에 앞서 모델링 결과를 가시화할 수 있도록 PCA를 진행한다. 이 과정은 여러 독립변수들로 이루어진 고차원 데이터를 2차원의 형태로 차원축소하는 과정이다. 차원축소된 2차원의 데이터는 df_pca로 설정하였으며, 추가적으로 실제 라벨에 대한 분류를 확인하기 위한 df_test를 df_pca와 Label 데이터프레임을 Pandas의 concat() 함수를 통해 생성하였다.

```
# 차원 축소
pca = PCA(n_components=2)
pca.fit(df_norm)
pca_array = pca.transform(df_norm)
df_pca = pd.DataFrame(pca_array)
df_pca = df_pca.rename(columns={0:'P1', 1:'P2'})
df_pca
```

	P1	P2
0	-0.759084	0.192877
1	-0.883743	0.362877
2	-0.720518	0.198503
3	-0.745542	0.276044
4	-0.982273	0.472887
...	...	...
13606	-0.112706	-0.092588
13607	-0.202975	0.053085
13608	-0.180033	-0.004420
13609	-0.146816	-0.046207
13610	-0.063494	-0.095763

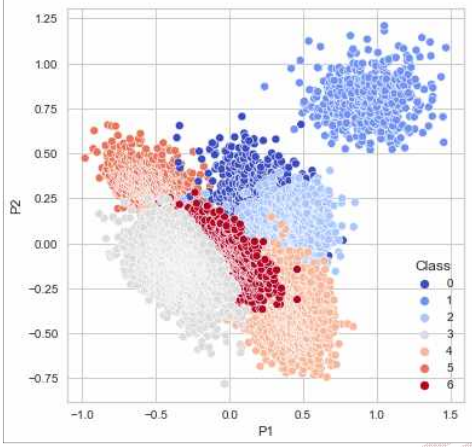
```
df_test = pd.concat([df_pca, Label],axis=1)
df_test
```

	P1	P2	Class
0	-0.759084	0.192877	5
1	-0.883743	0.362877	5
2	-0.720518	0.198503	5
3	-0.745542	0.276044	5
4	-0.982273	0.472887	5
...	...	...	...
13606	-0.112706	-0.092588	3
13607	-0.202975	0.053085	3
13608	-0.180033	-0.004420	3
13609	-0.146816	-0.046207	3
13610	-0.063494	-0.095763	3

13611 rows × 3 columns

```
plt.figure(figsize=(20, 6))
plt.subplot(132)
sns.scatterplot(x=df_test['P1'], y=df_test['P2'], data=df_test, hue=df_test['Class'], palette='coolwarm')
```

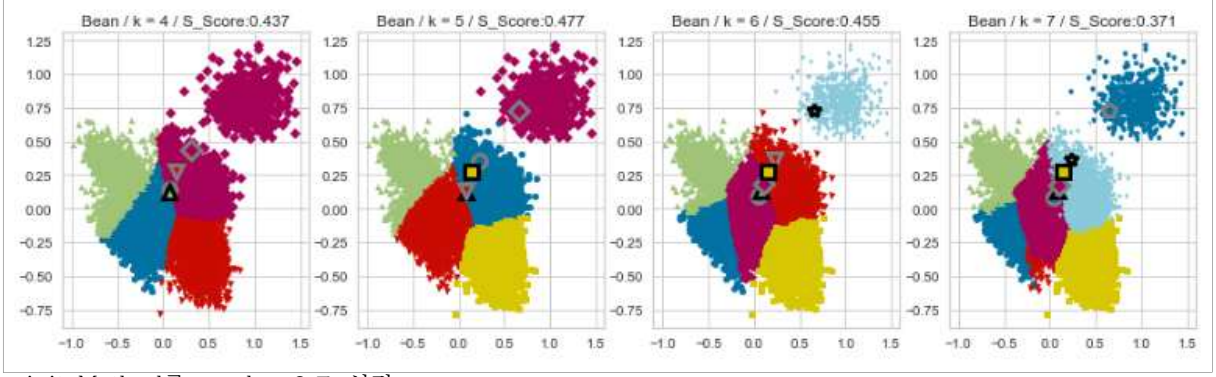
```
<AxesSubplot:xlabel='P1', ylabel='P2'>
```



- 위 코드와 시각화 자료는 df_test의 Column을 통해 df_pca에 대한 scatterplot을 Labeling하여 시각화하여 나타낸 것이며, hue=df_test['Class']를 설정함으로써 **실제 7종류의 Class에 대하여 각각 다른 색상을 입혀 Sample을 나타냈다.** 결과적으로 실제 Label에 대한 Sample의 군집과 분포를 확인할 수 있다. 해당 시각화 자료는 다양한 Clustering 모델들에 대한 External 평가지표가 될 수 있다.

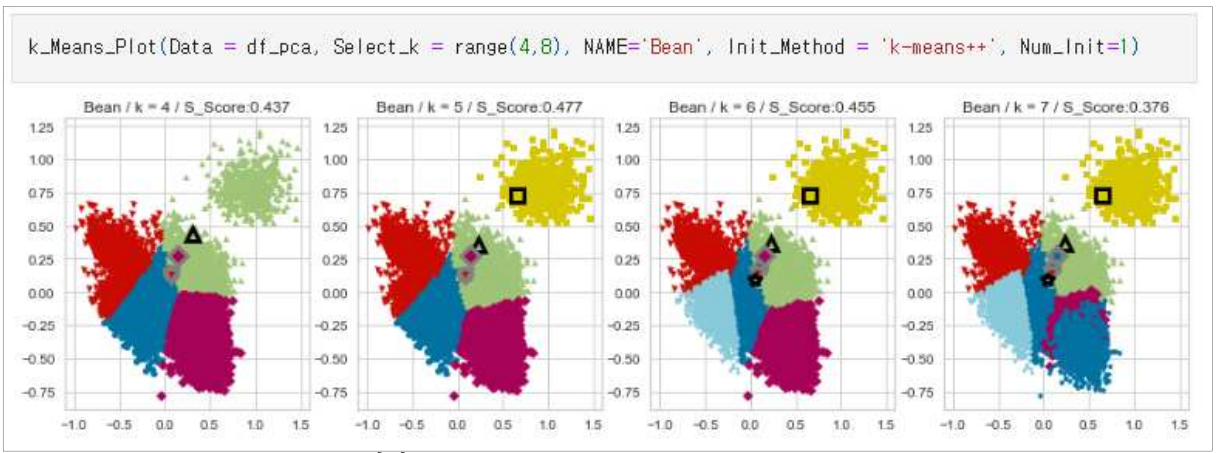
아래의 코드와 시각화 자료는 해당 데이터셋에 대하여 k-means 모델이 얼만큼 잘 분류할 수 있는지를 보여준다.

```
k_Means_Plot(Data = df_pca, Select_k = range(4,8), NAME='Bean', Init_Method = 'random', Num_Init=1)
```



- init_Method를 random으로 설정

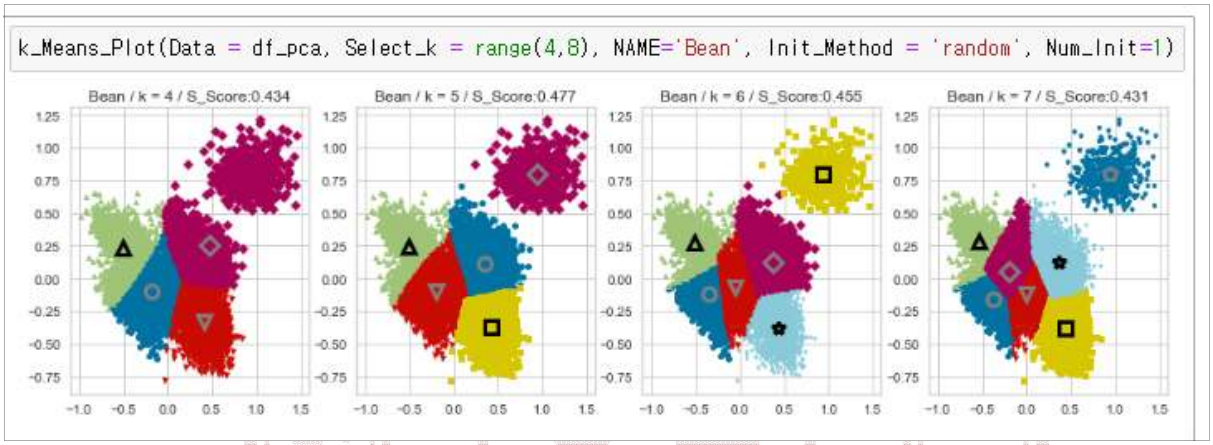
```
k_Means_Plot(Data = df_pca, Select_k = range(4,8), NAME='Bean', Init_Method = 'k-means++', Num_Init=1)
```



- init_Method를 k-means++로 설정

init-Method를 두 종류로 지정하여 Modeling을 진행해보았으며, **random**인 경우와 **k-means++**인 경우의 분포와 군집의 형태가 약간 다른 것을 확인할 수 있다. k-means 모델은 **초기 중심점을 설정**하는 것이 무척 중요한데, random은 랜덤하게 중심점을 설정하지만, k-means++는 초기 중심점을 조금 더 신중하게 설정한다. 초기 군집을 설정할 때 첫 번째 중심점을 찍고 두 번째 중심점부터는 이미 지정된 중심점으로부터 최대한 먼 곳으로 설정한다. 이는 초기 중심점을 전략적으로 배치하기 때문에, 일반적인 K-means의 random한 초기 중심점 설정으로 가끔 발생하는 문제를 조금은 완화시켜주는 장점이 있다고 말할 수 있다.

!! 위 그림에서는 중심점이 각각의 군집 내에 찍혀있지 않고, 엉뚱한 곳에 표시되어있다. 이는 고차원의 데이터인 df_norm으로 모델링을 진행하고나서 2차원 평면상에 투영시켰기 때문이라고 판단할 수 있다. PCA를 우선적으로 진행하고 fit(df_pca)로 함수내 코드를 수정하면 군집에 대한 적절한 중심점으로 표시되는 것을 아래와같이 확인할 수 있다.



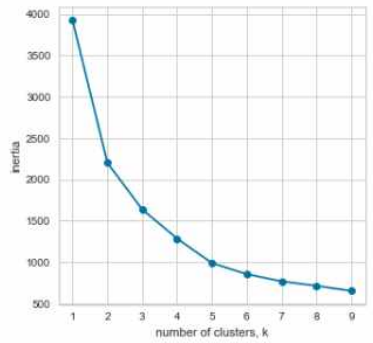
- Clustering은 비지도 학습이므로 군집 수와 군집의 형태에 대하여 정확한 평가를 내릴 수는 없다. 하지만 실습을 위해서 Label을 포함한 데이터셋을 선정하였기 때문에 실제의 군집형태와 비교하여 **External 평가**를 진행할 수 있다.

- 위 세 개의 시각화 자료에서 군집수 k 설정에 따라 각각 다른 **Silhouette Score**를 확인할 수 있다. 이 평가 지표는 k-means 군집화의 가장 보편적인 평가지표로 사용하며, 일반적으로 0.5보다 클 때 타당한 군집 결과라고 이야기한다.

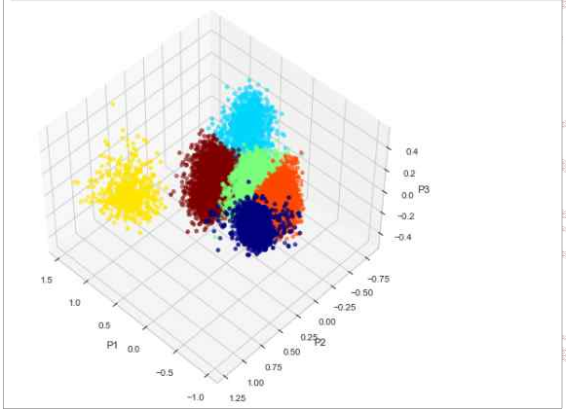
- 해당 데이터셋에 대한 k-means Clustering 결과는 그렇게 좋은 군집화 결과라고 말할 수는 없다. 하지만 실제 Class로 Labeling된 군집의 형태와 비교하는 External 평가를 해보았을 때, 나쁘지 않은 군집화 결과를 보여준다고 판단된다.

- 추가적으로 군집의 중심축으로부터 군집 내의 모든 관측치까지의 거리를 합한 값도 군집화의 평가지표가 될 수 있다. 이 값은 **SSE(Sum of Squared Error)**라고 한다. 이는 정성적인 평가 지표이므로 이를 통해 군집수에 대한 확실한 정답을 판단할 수는 없으며, 보편적으로 **elbow point**에서 합리적인 군집수 k를 갖는다고 판단한다. k값에 따른 SSE값의 변화를 보여주는 아래의 그래프를 통해 이를 확인할 수 있으며, k=4 혹은 k=5인 지점에서 더 이상 큰 폭으로 SSE가 감소하지 않기 때문에 해당 k값이 적절한 군집수가 될 수 있을 것이라고 판단된다.

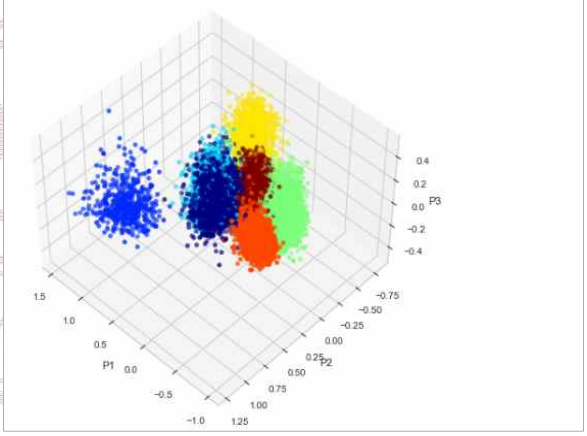
```python
ks = range(1,10)
inertias = []

for k in range(1,10):
    model = KMeans(n_clusters=k)
    model.fit(df_norm)
    inertias.append(model.inertia_)

# Plot ks vs inertias
plt.figure(figsize=(5, 5))

plt.plot(ks, inertias, '-o')
plt.xlabel('number of clusters, k')
plt.ylabel('inertia')
plt.xticks(ks)
plt.show()
```




```python
# 초기 중심점 설정, 그룹 수, random_state 설정
k=7
model = KMeans(init='k-means++', n_clusters = k, random_state=42)

# 학습
kmeans_plus = model.fit(df_pca)

fig = plt.figure(figsize=(6,6))
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)
ax.scatter(df_pca['P1'],df_pca['P2'],df_pca['P3'],c=kmeans_plus.labels_, cmap='jet')
ax.set_xlabel('P1')
ax.set_ylabel('P2')
ax.set_zlabel('P3')
plt.show()
```




```python
df_test = pd.concat([df_pca, Label],axis=1)
df_test


fig = plt.figure(figsize=(6,6))
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)
ax.scatter(df_test['P1'],df_test['P2'],df_test['P3'], c=df_test['Class'], cmap='jet')
ax.set_xlabel('P1')
ax.set_ylabel('P2')
ax.set_zlabel('P3')
plt.show()
```



- 3차원으로 차원축소하여서도 모델링 및 External 평가를 진행하였다. **k-means 군집화는 Sphere한 형태의 데이터 분포에 적합**하다는 사실을 더 직관적으로 알 수 있다.

4.2) Hierarchical clustering

```
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()
```

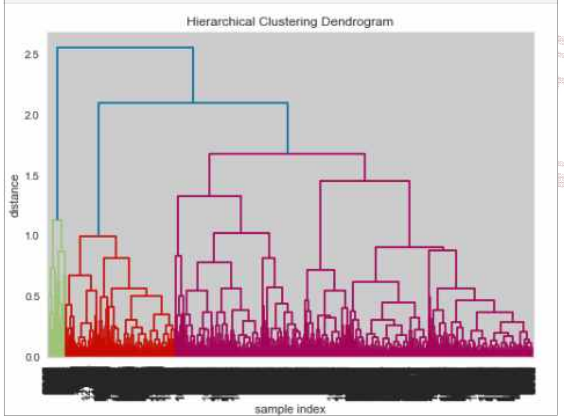
- 위 함수는 데이터를 계층적으로 군집화하는 과정에 대해서 **Dendrogram**을 나타내주기 위한 코드이다.

!! Dendrogram을 통해 **Greedy하게 sample혹은 군집의 유사도를 측정하며 Bottom Up 방식으로 군집을 생성해가는 과정을 확인**할 수 있으며, 하나의 군집이 될 때까지의 과정을 나타낸다. 이 자체만으로도 유의미한 계통 체계를 파악할 수 있다.

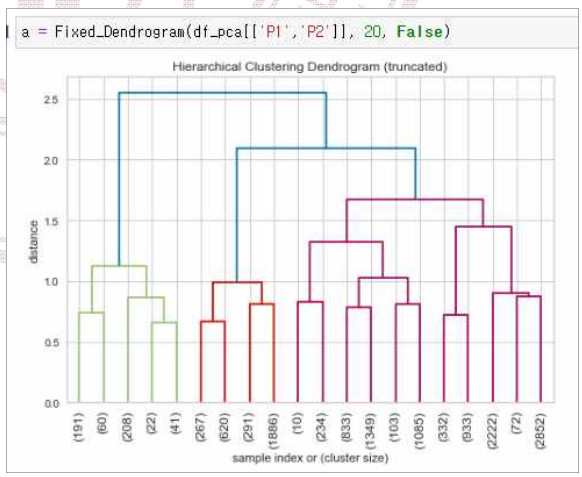
- 군집 사이의 유사도 거리는 매번 측정해야하는 핵심 수행 절차이다. 위 함수에서는 거리측정의 방식을 **Complete linkage**로 설정하였으며, 이는 두 군집이 포함하는 sample에 대해 가장 멀리 떨어진 sample로 군집간의 거리를 측정하는 방식이다.

아래의 시각화 자료는 위에서 구현한 함수를 활용하여 df_pca의 Dendrogram을 보여준다.

```
# 모든 리프노드에 대한 dendrogram 생성
Fixed_Dendrogram(df_pca[['P1','P2']], -1, True) # -1: nonsensingful value
```



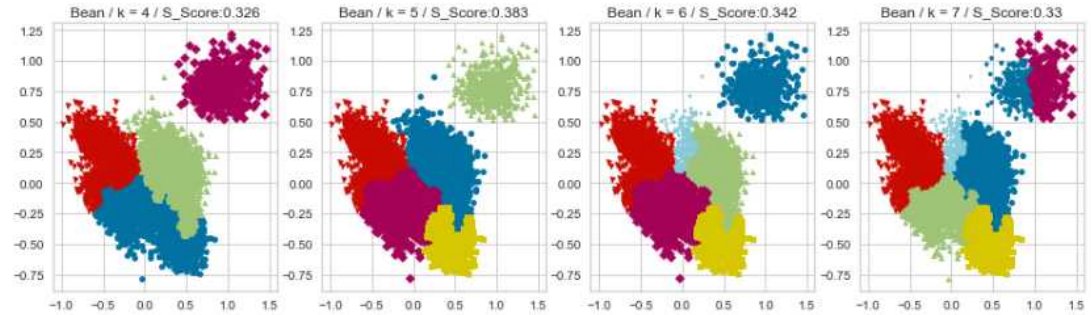

```
a = Fixed_Dendrogram(df_pca[['P1','P2']], 20, False)
```



- 왼쪽의 경우 **Num_of_p를 -1로 설정**해줌으로써 시작이되는 가장 하단의 노드를 모든 sample로 설정하여 유사도 거리 측정을 통해 군집화가 진행된다. sample의 수가 많은 경우 실행 시간이 무척 오래 걸린다.
- 오른쪽의 경우 시작이 되는 하단 노드를 20개의 임의 Cluster로 설정하고 Full Use는 False로 설정하여 20개의 Cluster로부터 유사도 거리 측정으로 군집화를 해나가는 과정이다. 더욱 간결해진 Dendrogram을 확인할 수 있으며, Dendrogram에서 **Threshold를 잘 설정하여 적절한 군집의 수를 결정**할 수 있다.

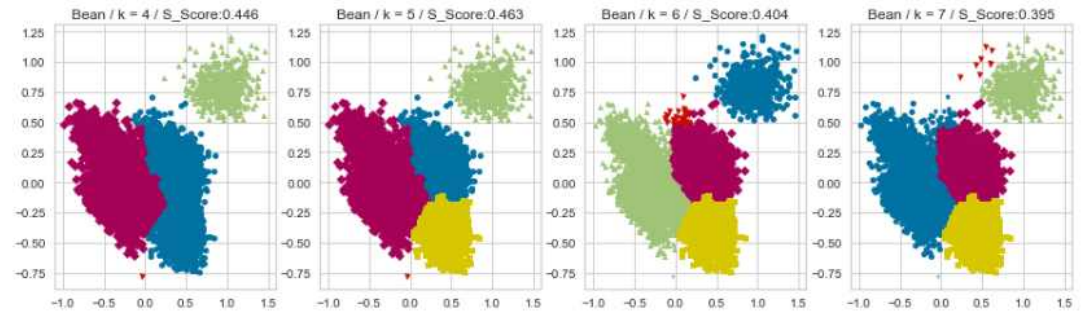
```
def Hclust_Plot(Data,Select_k,NAME):
    Data2 = Data[['P1', 'P2']]
    fig, axes = plt.subplots(1, (np.max(list(Select_k))-np.min(list(Select_k)))+1, figsize=(15, 4))
    for i in Select_k:
        H_Clustering = AgglomerativeClustering(n_clusters=i,linkage="complete")
        P_Labels = H_Clustering.fit_predict(Data2)
        mglearn.discrete_scatter(Data2['P1'], Data2['P2'], P_Labels, ax=axes[i - 4], s=5)
        axes[i - 4].set_title("Data:" + NAME + ' / k = ' + str(i))
        Score=np.round(silhouette_score(Data2,P_Labels),3)
        axes[i - 4].set_title( NAME + ' / k = ' + str(i)+' / S_Score:'+str(Score))
```

```
Hclust_Plot(df_pca, range(4, 8), 'Bean')
```



- 아래의 그림은 Average linkage 방식으로 거리측정.

```
Hclust_Plot(df_pca, range(4, 8), 'Bean')
```



- 위 그림은 계층적 군집화의 결과를 가시화 시키는 함수와 시각화 자료이다. k-means에서와 마찬가지로 k=4 ~ k=7로 설정했을 때, 군집의 형태가 어떠한지 확인할 수 있다. 일반적으로 k=5인 경우의 Silouoette Score가 가장 높게 나타났다. 이 점은 k-means 알고리즘과 동일하다.

- 계층적 군집화의 유사도 거리 측정의 4가지 방식으로 **Single, Colmplet, Average, Centroid** linkage를 배웠으며, 이들의 군집 간의 거리를 측정하는 방식은 각각 다르다. 무엇이 가장 낫다고 말할 수 없으며, 데이터의 분포에 따라 그에 맞는 기법을 적용해야한다.

- 해당 데이터셋에도 네가지 기법을 적용해서 결과를 확인했으며, 위 그림은 **Average linkage** 방식으로 모델링한 경우를 보여주며, 4가지 거리 측정 방식 중 가장 좋은 Score를 가졌다. **!!**하지만 위 경우에서는 Score에 기반한 Relative 평가보다 실제 라벨링된 군집의 형태와 비교하는 **External 방식의 평가**를 진행하는 것이 더 좋을 것이라고 판단된다.

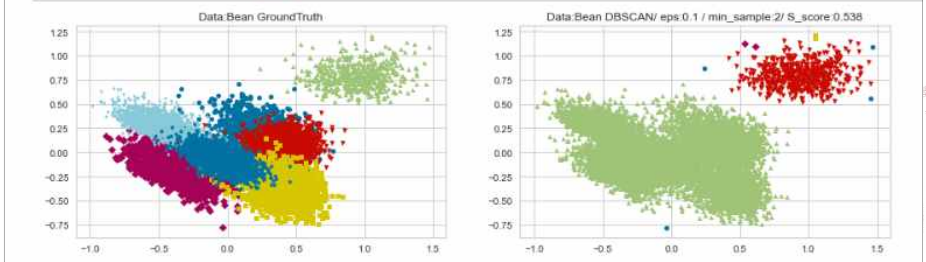
- 실제 class로 라벨링 된 scatterplot과 비교해 보았을 때도 k-means에 비해 좋지 못한 결과를 보인다. 해당 데이터셋은 계층적 군집화보다는 k-means로 모델링한 경우에 더 좋은 성능을 보이는 것으로 판단된다.

4.2) DBSCAN clustering

- DBSCAN은 Hyperparameters에 대한 설정이 무척 중요한 군집 모델이다. Hyperparameter로는 ϵ과 **minPts**를 가진다. ϵ는어떠한 점에 대해 이웃점을 탐색할 수 있는 최대 반경을 의미하며, minPts는 ϵ(최대 반경)내 포함해야하는 최소한의 이웃점 개수를 의미한다. 이 둘에 대한 설정이 **모델 구축 자체에 영향을 준다.**
- DBSCAN에서는 점들에대해 **core, border, noise**로 분류하여 정의한다. 이웃점이 설정한 minPts이상으로 충분한 중심점을 core라고 하며, 해당 중심점에서 ϵ내에 있는 점을 border라고 하며, 둘에 속하지 않다면 noise로 정의한다.

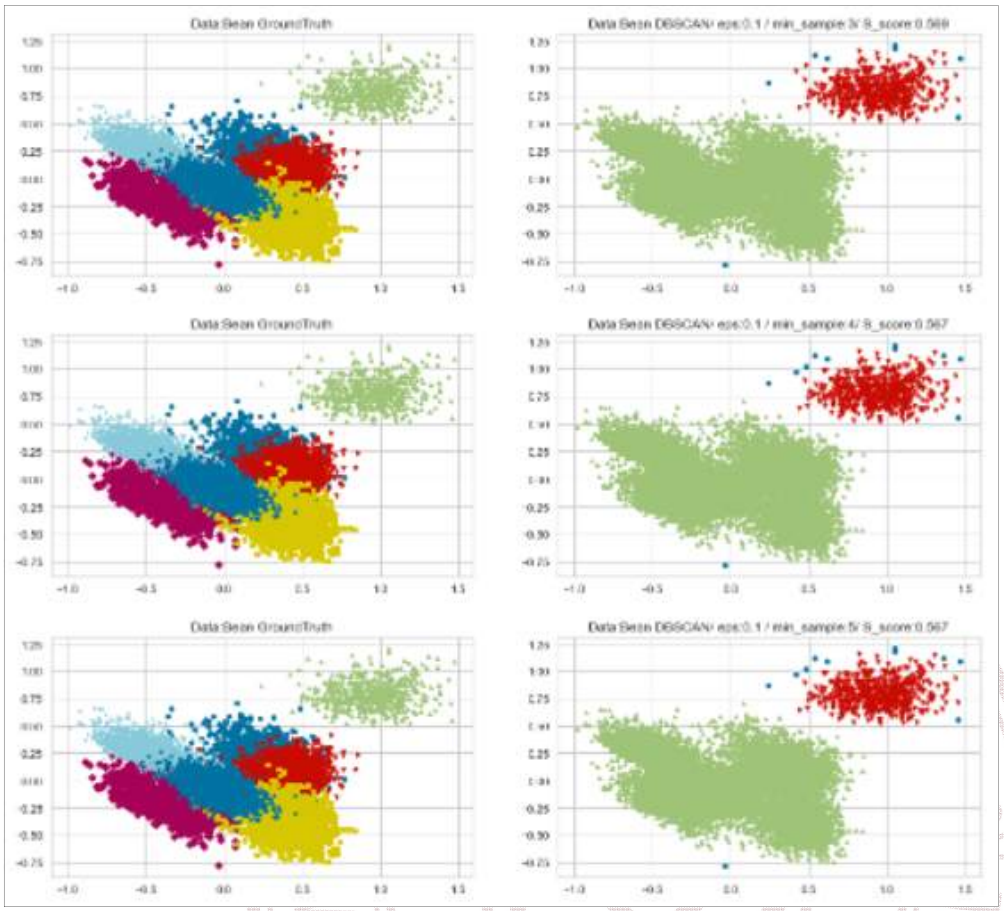
DBSCAN clustering

```
def DBSCAN_Plot(Data,NAME,min_samples=2,eps=0.1):
    Data2 = Data[['P1', 'P2']]
    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['P1'], Data2['P2'], Data['Class'], ax=axes[0], s=5)
    axes[0].set_title("Data:" + NAME + ' GroundTruth')

    mglearn.discrete_scatter(Data2['P1'], Data2['P2'], 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=df_test, NAME="Bean")
```



- 해당 데이터셋은 2차원 평면에서 오른쪽 상단의 군집을 제외한 나머지 군집들이 아주 밀도있게 뭉쳐있는 형태를 가진다.
- DBSCAN은 군집수에 대한 설정을 따로 할 수 없다. 대부분의 Hyperparameters 설정에 크게 두 군집으로 나누어진 결과를 보여주었다. 그나마 나은 Silhouette Score를 가지도록 ad-hoc 방식으로 초기 파라미터를 설정하였다.

```
DBSCAN_Plot(Data=df_test,NAME="Bean",min_samples=1,eps=0.1)
DBSCAN_Plot(Data=df_test,NAME="Bean",min_samples=2,eps=0.1)
DBSCAN_Plot(Data=df_test,NAME="Bean",min_samples=3,eps=0.1)
DBSCAN_Plot(Data=df_test,NAME="Bean",min_samples=4,eps=0.1)
DBSCAN_Plot(Data=df_test,NAME="Bean",min_samples=5,eps=0.1)
```



- Hyperparameter에 대한 설정을 다양하게 해가면서 군집의 형태를 확인하였다. 모든 경우에 크게 두 군집으로 분류하는 것을 확인할 수 있었고, Silhouette Score는 계속해서 낮게 나타났다.

- 해당 데이터셋에서 DBSCAN 모델의 한계이자 단점을 확인할 수 있다. Cluster의 개수를 미리 지정할 수 없는 점이 장점이 될 수도 있지만, 해당 데이터셋을 평가하기 위해서는 이를 설정해주는 것이 필요해 보인다. DBSCAN은 밀도가 높은 지점에서 낮은 지점으로 중심점을 설정하면서 군집화를 해나가는데, 해당 데이터셋은 군집간의 밀도마저 너무 높기 때문에 이를 제대로 분류해내지 못한다는 것을 확인할 수 있었다. 이 경우, Hyperparameters 설정에도 큰 어려움이 따랐다.

!! 최종적으로 Dry Bean 데이터셋에 세가지 모델이 어떻게 반응하였는지 비교해보자면, 분리형 군집화인 k-means cluster를 적용했을 때 다양한 관점의 평가에서 가장 합리적인 결과를 보였다고 판단된다. 계층적 군집화인 Hierarchical cluster를 적용한 경우의 Relative한 평가는 k-means와 비슷한 결과를 보였지만, 실제 라벨과 비교해보는 평가에서는 k-means에 비해 덜 좋은 성능을 보이는 것으로 판단하였다. 밀도 기반의 군집화를 하는 DBSCAN의 경우 보통 이전 두 모델에 비해 장점이 많다고 하지만, 해당 실습의 데이터에서는 좋은 성능을 보였다고 말할 수는 없다.

!! 모든 데이터에 좋은 성능을 보장하는 모델은 없으므로, 데이터의 특성을 잘 파악하여 그에 적합한 알고리즘을 선정하여 적용하는 것이 무척 중요하다는 것을 알 수 있다.

“ Decision Tree ”

1. 데이터 수집

Decision Tree 실습을 위해 사용된 데이터는 “Obesity DataSet” (https://archive.ics.uci.edu/ml/datasets/Estimation+of+obesity+levels+based+on+eating+habits+and+physical+condition+) 이다. 해당 Dataset은 멕시코, 페루, 콜롬비아 국적을 가진 사람들의 식습관과 신체정보에 기반한 비만도 정보를 제공한다. 식습관, 신체정보와 관련된 변수는 15개이며, 비만정도를 나타내는 7개의 고유한 값은 해당 데이터셋의 Class이다.

Estimation of obesity levels based on eating habits and physical condition Data Set

Download: Data Folder, Data Set Description

Abstract: This dataset include data for the estimation of obesity levels in individuals from the countries of Mexico, Peru and Colombia, based on their eating habits and physical condition.

Data Set Characteristics:	Multivariate	**Number of Instances:**	2111	**Area:**	Life
Attribute Characteristics:	Integer	**Number of Attributes:**	17	**Date Donated**	2019-08-27
Associated Tasks:	Classification, Regression, Clustering	**Missing Values?**	N/A	**Number of Web Hits:**	80493

2. 데이터 형태 파악

필수 라이브러리

```
# 라이브러리 로드
import pandas as pd
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split # 학습, 테스트set 구분
from sklearn.tree import export_graphviz # tree 시각화를 위해
# export_graphviz : 의사결정나무에 대한 graphviz dot data 생성하는 함수
import graphviz # tree 시각화
import sklearn.metrics as mt # 성능지표를 계산하기 위해 import
from sklearn.model_selection import cross_val_score, cross_validate # 교차검증

import warnings
warnings.filterwarnings('ignore')
```

1) 데이터 셋 소개

```
df = pd.read_csv('obesity.csv')
df.head()
```

	id	Gender	Age	Height	Weight	family_history_with_overweight	FAVC	FCVC	NCP	CAEC	SMOKE	CH2O	SCC	FAF	TUE	CALC	MTRANS	NObeyesdad
0	1	Female	21.0	1.62	64.0	yes	no	2.0	3.0	Sometimes	no	2.0	no	0.0	1.0	no	Public_Transportation	Normal_Weight
1	2	Female	21.0	1.52	56.0	yes	no	3.0	3.0	Sometimes	yes	3.0	yes	3.0	0.0	Sometimes	Public_Transportation	Normal_Weight
2	3	Male	23.0	1.80	77.0	yes	no	2.0	3.0	Sometimes	no	2.0	no	2.0	1.0	Frequently	Public_Transportation	Normal_Weight
3	4	Male	27.0	1.80	87.0	no	no	3.0	3.0	Sometimes	no	2.0	no	2.0	0.0	Frequently	Walking	Overweight_Level_I
4	5	Male	22.0	1.78	89.8	no	no	2.0	1.0	Sometimes	no	2.0	no	0.0	0.0	Sometimes	Public_Transportation	Overweight_Level_II

- 위 두 그림은 Decision Tree 실습에 사용될 필수 라이브러리의 import 과정과 Pandas를 활용해 데이터셋을 프레임의 형태로 가져오는 것을 보여준다.
- id 컬럼을 제거하고, 해당 데이터셋이 결측치를 포함하는지 확인할 수 있다.

```
# id 컬럼 제거 / 결측치 확인
df.drop('id', axis=1, inplace=True)
df.isnull().sum()
```

```
Gender                            0
Age                               0
Height                            0
Weight                            0
family_history_with_overweight    0
FAVC                              0
FCVC                              0
NCP                               0
CAEC                              0
SMOKE                             0
CH2O                              0
SCC                               0
FAF                               0
TUE                               0
CALC                              0
MTRANS                            0
NObeyesdad                        0
dtype: int64
```

```
print(df.info())
df.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2111 entries, 0 to 2110
Data columns (total 17 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   Gender                          2111 non-null   object
 1   Age                             2111 non-null   float64
 2   Height                          2111 non-null   float64
 3   Weight                          2111 non-null   float64
 4   family_history_with_overweight  2111 non-null   object
 5   FAVC                            2111 non-null   object
 6   FCVC                            2111 non-null   float64
 7   NCP                             2111 non-null   float64
 8   CAEC                            2111 non-null   object
 9   SMOKE                           2111 non-null   object
 10  CH2O                            2111 non-null   float64
 11  SCC                             2111 non-null   object
 12  FAF                             2111 non-null   float64
 13  TUE                             2111 non-null   float64
 14  CALC                            2111 non-null   object
 15  MTRANS                          2111 non-null   object
 16  NObeyesdad                      2111 non-null   object
dtypes: float64(8), object(9)
memory usage: 280.5+ KB
```

```
print("데이터 셋의 관측치 수와 Class 수는 아래와 같다.")
print("Bank :",np.shape(df)[0],"/ Class : ", df['NObeyesdad'].nunique())
```

```
데이터 셋의 관측치 수와 Class 수는 아래와 같다.
Bank : 2111 / Class :  7
```

- info() 함수를 활용해 변수의 type을 포함한 요약정보를 확인할 수 있다.
- 추가적으로 describe() 함수를 통해 연속형 변수들의 분포와 통계량을 확인할 수 있다.
- 종속변수 'NObeyesdad'의 Class 수를 확인할 수 있다.

3. 전처리

2-1) Decision Tree 실습을 위한 종속 변수 처리

```
# 종속 변수의 class
df['NObeyesdad'].unique()
```

```
array(['Normal_Weight', 'Overweight_Level_I', 'Overweight_Level_II',
       'Obesity_Type_I', 'Insufficient_Weight', 'Obesity_Type_II',
       'Obesity_Type_III'], dtype=object)
```

```
# obesity / non-obesity 로 분류
df['NObeyesdad'] = df['NObeyesdad'].replace({'Normal_Weight': 0})
df['NObeyesdad'] = df['NObeyesdad'].replace({'Overweight_Level_I': 0})
df['NObeyesdad'] = df['NObeyesdad'].replace({'Overweight_Level_II': 0})
df['NObeyesdad'] = df['NObeyesdad'].replace({'Insufficient_Weight': 0})


df['NObeyesdad'] = df['NObeyesdad'].replace({'Obesity_Type_I': 1})
df['NObeyesdad'] = df['NObeyesdad'].replace({'Obesity_Type_II': 1})
df['NObeyesdad'] = df['NObeyesdad'].replace({'Obesity_Type_III': 1})

df.rename(columns={'NObeyesdad' : 'class'},inplace=True)
df
```

	Gender	Age	Height	Weight	family_history_with_overweight	FAVC	FCVC	NCP	CAEC	SMOKE	CH2O	SCC	FAF	TUE	CALC	MTRANS	class
0	Female	21.000000	1.620000	64.000000	yes	no	2.0	3.0	Sometimes	no	2.000000	no	0.000000	1.000000	no	Public_Transportation	0
1	Female	21.000000	1.520000	56.000000	yes	no	3.0	3.0	Sometimes	yes	3.000000	yes	3.000000	0.000000	Sometimes	Public_Transportation	0
2	Male	23.000000	1.800000	77.000000	yes	no	2.0	3.0	Sometimes	no	2.000000	no	2.000000	1.000000	Frequently	Public_Transportation	0
3	Male	27.000000	1.800000	87.000000	no	no	3.0	3.0	Sometimes	no	2.000000	no	2.000000	0.000000	Frequently	Walking	0
4	Male	22.000000	1.780000	89.800000	no	no	2.0	1.0	Sometimes	no	2.000000	no	0.000000	0.000000	Sometimes	Public_Transportation	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2106	Female	20.976842	1.710730	131.408528	yes	yes	3.0	3.0	Sometimes	no	1.728139	no	1.676269	0.906247	Sometimes	Public_Transportation	1
2107	Female	21.982942	1.748584	133.742943	yes	yes	3.0	3.0	Sometimes	no	2.005130	no	1.341390	0.599270	Sometimes	Public_Transportation	1
2108	Female	22.524036	1.752206	133.689352	yes	yes	3.0	3.0	Sometimes	no	2.054193	no	1.414209	0.646288	Sometimes	Public_Transportation	1
2109	Female	24.361936	1.739450	133.346641	yes	yes	3.0	3.0	Sometimes	no	2.852339	no	1.139107	0.586035	Sometimes	Public_Transportation	1
2110	Female	23.664709	1.738836	133.472641	yes	yes	3.0	3.0	Sometimes	no	2.863513	no	1.026452	0.714137	Sometimes	Public_Transportation	1

2111 rows × 17 columns

- 우선적으로 Decision Tree 알고리즘 적용과 평가를 위한 형태의 데이터셋으로 변형시켰다.
- 세부적인 Class를 Obesity와 Non-Obesity로 재분류하기 위해 (Insufficient_Weight, Normal_Weight, Overweight_Level_Ⅰ, Overweight_Level_Ⅱ)는 0으로, Obesity에 해당하는 (Obesity_Type_Ⅰ, Obesity_Type_Ⅱ, Obesity_Type_Ⅲ) 는 1로 mapping 시켰다.
- Column 명은 class로 변경하였다.

2-2) 범주형 변수에 대한 처리

```
# 범주형 변수의 처리를 위해 feature의 type 확인
categorical = df.select_dtypes('object')
print(categorical.columns.to_list())
print('₩n범주형 변수 각각의 고유한 값의 개수는 아래와 같습니다.₩n')

for col_name in categorical:
    print(col_name, ':', categorical[col_name].nunique())
```

```
['Gender', 'family_history_with_overweight', 'FAVC', 'CAEC', 'SMOKE', 'SCC', 'CALC', 'MTRANS']

범주형 변수 각각의 고유한 값의 개수는 아래와 같습니다.

Gender : 2
family_history_with_overweight : 2
FAVC : 2
CAEC : 4
SMOKE : 2
SCC : 2
CALC : 4
MTRANS : 5
```

- 해당 데이터셋은 8개의 범주형 독립변수를 가지고 있으며, 각각의 변수가 가지는 고유한 값에따라 Encoding 기법을 달리 해주기로 결정하였다. 2개의 고유한 값을 가지는 변수들에 대해서는 0과 1로 Label Encoding을 적용해도 모델 구축에 문제가 되지 않으므로 **Label Encoding**을 적용시키고, 3개 이상의 고유한 값을 가지는 변수들에 대해서는 dummies() 함수를 통해 **One-Hot Encoding**을 적용하기로 하였다.

```
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()
df['Gender'] = encoder.fit_transform(df['Gender'].values)
df['family_history_with_overweight'] = encoder.fit_transform(df['family_history_with_overweight'].values)
df['FAVC'] = encoder.fit_transform(df['FAVC'].values)
df['SMOKE'] = encoder.fit_transform(df['SMOKE'].values)
df['SCC'] = encoder.fit_transform(df['SCC'].values)
df.head()
```

	Gender	Age	Height	Weight	family_history_with_overweight	FAVC	FCVC	NCP	CAEC	SMOKE	CH2O	SCC	FAF	TUE	CALC	MTRANS	class
0	0	21.0	1.62	64.0	1	0	2.0	3.0	Sometimes	0	2.0	0	0.0	1.0	no	Public_Transportation	0
1	0	21.0	1.52	56.0	1	0	3.0	3.0	Sometimes	1	3.0	1	3.0	0.0	Sometimes	Public_Transportation	0
2	1	23.0	1.80	77.0	1	0	2.0	3.0	Sometimes	0	2.0	0	2.0	1.0	Frequently	Public_Transportation	0
3	1	27.0	1.80	87.0	0	0	3.0	3.0	Sometimes	0	2.0	0	2.0	0.0	Frequently	Walking	0
4	1	22.0	1.78	89.8	0	0	2.0	1.0	Sometimes	0	2.0	0	0.0	0.0	Sometimes	Public_Transportation	0

```
# 나머지 독립 변수에 대하여 One-Hot Encoding
df = pd.get_dummies(df)
df
```

	Gender	Age	Height	Weight	family_history_with_overweight	FAVC	FCVC	NCP	SMOKE	CH2O	...	CAEC_no	CALC_Always	CALC_Frequently	CALC_Sometimes	CALC_no	MTRANS_Automobile	MTRANS_Bike	MTRANS_Motorbike	MTRANS_Public_Transportation	MTRANS_W
0	0	21.000000	1.620000	64.000000	1	0	2.0	3.0	0	2.000000	...	0	0	0	0	1	0	0	0	1	
1	0	21.000000	1.520000	56.000000	1	0	3.0	3.0	1	3.000000	...	0	0	0	1	0	0	0	0	1	
2	1	23.000000	1.800000	77.000000	1	0	2.0	3.0	0	2.000000	...	0	0	1	0	0	0	0	0	1	
3	1	27.000000	1.800000	87.000000	0	0	3.0	3.0	0	2.000000	...	0	0	1	0	0	0	0	0	0	
4	1	22.000000	1.780000	89.800000	0	0	2.0	1.0	0	2.000000	...	0	0	0	1	0	0	0	0	1	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
2106	0	20.976842	1.710730	131.408528	1	1	3.0	3.0	0	1.728139	...	0	0	0	1	0	0	0	0	1	
2107	0	21.982942	1.748584	133.742943	1	1	3.0	3.0	0	2.005130	...	0	0	0	1	0	0	0	0	1	
2108	0	22.524036	1.752206	133.689352	1	1	3.0	3.0	0	2.054193	...	0	0	0	1	0	0	0	0	1	
2109	0	24.361936	1.739450	133.346641	1	1	3.0	3.0	0	2.852339	...	0	0	0	1	0	0	0	0	1	
2110	0	23.664709	1.738836	133.472641	1	1	3.0	3.0	0	2.863513	...	0	0	0	1	0	0	0	0	1	

2111 rows × 27 columns

- 위 그림은 **Label Encoding**과 **One-Hot Encoding**이 적용된 데이터의 형태를 보여준다. Column의 수가 27개로 늘어난 것을 확인할 수 있다.

2-3) 성능 평가를 위한 Dataset Split

```
x_train, x_test, y_train, y_test = train_test_split(df, df['class'], test_size=0.3, random_state=1024)
x_train = x_train.drop(['class'], axis=1)
x_test = x_test.drop(['class'], axis=1)
y_train = y_train.to_frame()
y_test = y_test.to_frame()
```

- Decision Tree 모델을 구축하고, 모델에 대한 평가를 하기 위해서는 **Training Set과 Test Set으로 분리하는 과정**이 필요하다. 이 과정은 sklearn의 train_test_split() 함수를 통해 아주 쉽게 진행된다. y에 해당되는 변수를 class로 지정하고, 두 Dataset의 비율은 7:3으로 설정하였다.

3. Decision Tree

3-2) Training

```
dt_clf = DecisionTreeClassifier(random_state=5764)
dt_clf.fit(x_train, y_train)
```

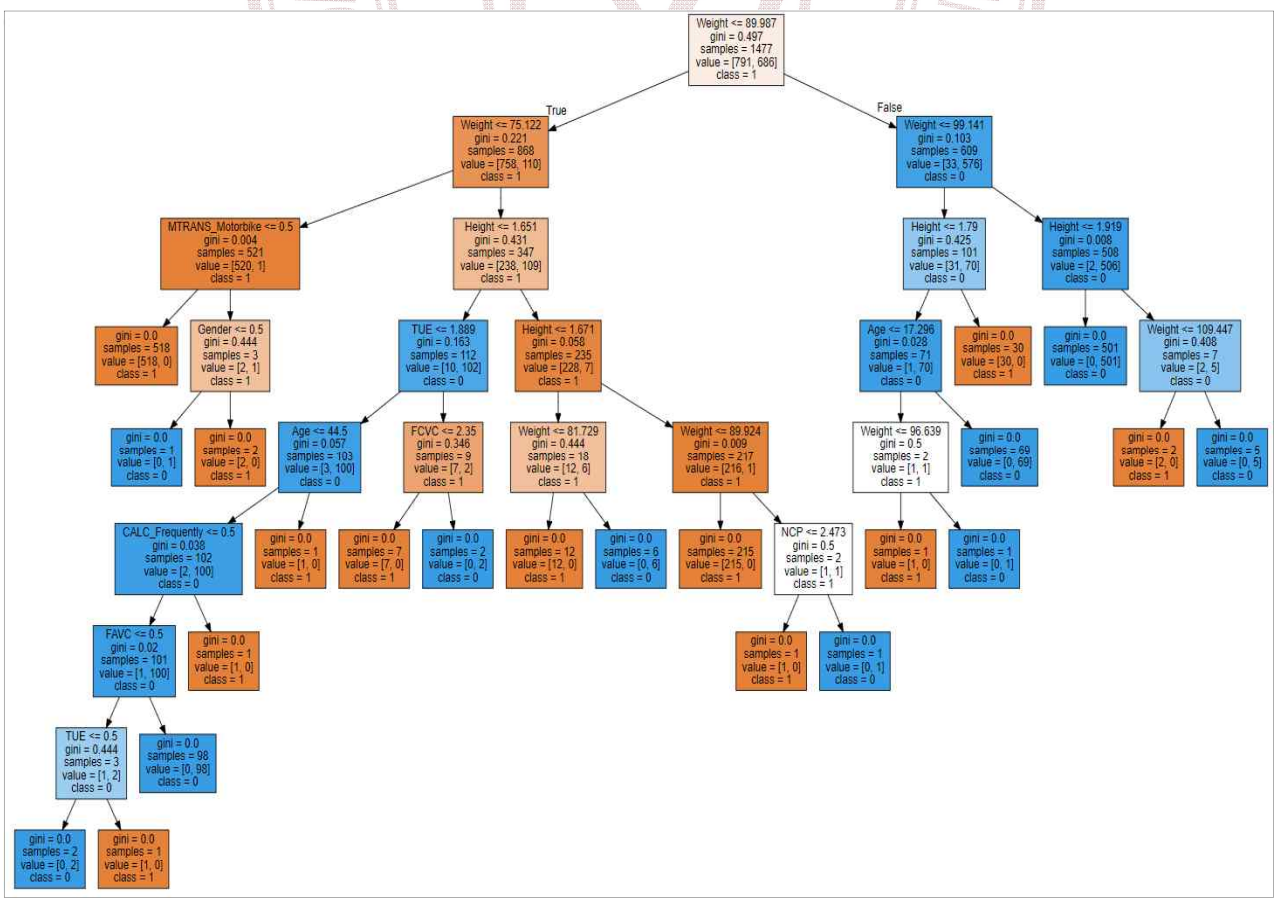
```
DecisionTreeClassifier(random_state=5764)
```

- 위 그림은 Training Dataset에 대한 DecisionTree모델을 구축하는 과정을 보여준다. 이 또한 sklean의 fit() 함수를 통해 아주 쉽게 training 시킬 수 있다.

3-3) Graphviz -> Tree 가시화

```
export_graphviz(dt_clf, out_file="tree.dot", class_names = ['1', '0'], feature_names = x_train.columns, impurity=True, filled=True)
```

```
import graphviz
print('[ max_depth의 제약이 없는 경우의 Decision Tree 시각화 ]')
# 위에서 생성된 tree.dot 파일을 Graphviz 가 읽어서 시각화

with open("tree.dot") as f:
    dot_graph = f.read()
graphviz.Source(dot_graph)
```



- 위 그림의 단계는 Decision Tree Model 구축이 **어떠한 과정으로 진행되는지를 파악**하기 위해 무척 중요하다. 가장 위 Root node에 위치한 Weight는 구축된 Tree모델에서 가장 중요하게 판단한 변수라고 말할 수 있다. 그 아래의 Internal node를 통해서는 다양한 중요변수에 대한 조건에 의해 정상범주와 이상범주를 분류한다. 노드의 색을통해 계속해서 **재귀적인 분리**를 해나가는 특징을 확인할 수 있고, 가장 하단에 위치한 Leaf node는 부모 노드들에 의해 반복적으로 분류된 데이터를 최종적으로 정상 혹은 이상으로 판단하여 예측한다. Graphviz 그림을 통해 Decision Tree Model이 어떠한 형태로 데이터를 분류하는지에 대한 원리를 **규칙 기반으로 이해**할 수 있다. 사람이 이해할 수 있는 친숙한 매커니즘을 제공한다는 것 자체만으로 이는 아주 큰 의미를 가지며, **Decision Tree를 통해 상당히 다양한 Insight를 도출할 수 있다고 말할 수 있다.**

- 아래의 그림은 위 Decision Tree Model에 대한 평가 지표이다. 별도의 Hyper Parameter를 설정하지 않았으며, 다양한 측면의 평가를 위해서 교수님께서 사용하신 print_pred_result() 함수로 구현하였다.

4-1) Scoring and Confusion Matrix

```
# 학습결과 평가
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')

print_pred_result(y_test, y_pred)
```

```
Train_Accuracy :  1.0

Test_Accuracy :  0.9794952681388013

Accuracy:  0.98

Recall:  0.98

Precision:  0.98

F1_score:  0.98

Confusion Matrix:
 [[341   7]
 [  6 280]]
```

```
def print_pred_result(y_test, y_pred):
    accuracy = mt.accuracy_score(y_test, y_pred)
    recall = mt.recall_score(y_test, y_pred)
    precision = mt.precision_score(y_test, y_pred)
    f1_score = mt.f1_score(y_test, y_pred)
    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)
```

- 1.0 의 Train_Accuracy Score를 갖는 것을 확인할 수 있다. 이는 Train Set에서의 완전한 모델을 구축하였기 때문에 발생하는 일이다. 다른 평가 지표도 다양하게 확인할 수 있다. Confusion Matrix를 통해 모델의 성능을 쉽게 확인할 수 있는데, 위에서 전처리된 데이터셋의 class에 대하여 '이상을 정상으로 오분류'한 샘플의 수 7개, '정상을 이상으로 오분류'한 샘플의 수 수는 6개로 확인된다.

- 추가적으로 정확도(Accuracy), 재현률(Recall), 정밀도(Precision), F1 score도 확인할 수 있다. Hyper Parameter 설정 없이 모델을 구축한 결과에도 Decision Tree는 나쁘지 않은 성능을 보여준다고 판단된다.

***
- Accuracy는 Decision Tree가 0을 0으로, 1을 1로 잘 분류한 비율을 의미한다.
- Recall은 실제 0을 0으로 잘 분류한 비율을 의미한다.
- Precision은 0으로 분류한 관측치들 중 실제로 그러한 비율을 의미한다.
- F1 Score는 Precision과 Recall의 조화 평균이다. 이는 Label이 불균형할 때 좋은 지표가 된다.
***

하지만 Train set에 과적합된 Decision Tree를 다양한 측면으로 구축해보기 위해 아래와 같이 Hyper parameter를 설정하여 Decision Tree를 구축하였다.

4. Hyper parameter를 사용하여 Decision Tree 구축

5-1) Grid Search 수행

```
from sklearn.model_selection import GridSearchCV
dt_clf2 = DecisionTreeClassifier(random_state=5)
parameters = {'splitter': ['best', 'random'],
              'max_depth': [3, 5, 6],
              'min_samples_split': [2, 3, 4]}

dt_grid = GridSearchCV(dt_clf2, param_grid = parameters, cv = 5)

dt_grid.fit(x_train, y_train)

result = pd.DataFrame(dt_grid.cv_results_['params'])
result['mean_test_score'] = dt_grid.cv_results_['mean_test_score']
result.sort_values(by='mean_test_score', ascending=False)
```

	max_depth	min_samples_split	splitter	mean_test_score
14	6	3	best	0.987803
16	6	4	best	0.987128
6	5	2	best	0.987126
10	5	4	best	0.986448
8	5	3	best	0.986448
12	6	2	best	0.985772
0	3	2	best	0.976312
2	3	3	best	0.976312
4	3	4	best	0.976312
11	5	4	random	0.922162

- 위 그림은 교수님께서 Grid Search를 통해 적절한 Hyper Parameter를 찾기 위해 사용하신 코드이다.

5-2) 하이퍼 파라미터 튜닝으로 Decsion Tree 구축

```
# 학습결과 평가
dt_grid = DecisionTreeClassifier(random_state=2, splitter='best', max_depth=6, min_samples_split=3)
dt_grid.fit(x_train, y_train)
y_pred = dt_grid.predict(x_test)

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

print_pred_result(y_test, y_pred)
```

```
Train_Accuracy :  0.997968855788761

Test_Accuracy :  0.9810725552050473

Accuracy:  0.98

Recall:  0.99

Precision:  0.97

F1_score:  0.98

Confusion Matrix:
 [[340   8]
 [  4 282]]
```

- GridSearchCV() 를 통해 좋은 Score를 낼 수 있는 두가지 Hyper Parameter를 확인하였고, 이를 사용하여 Decision Tree를 구축한 결과, 기존의 Tree보다 조금 더 좋은 성능을 가진 Model을 구축할 수 있었다. 기존의 Tree가 오분류한 샘플의 수가 많지 않았기 때문에 상대적인 비교가 쉽지 않지만, 위 과정에서 눈여겨 보아야 할 부분은 **Train Score에 대한 규제를 주어 Test Score의 점수를 높일 수 있었다**는 점이다.

5. 가지치기 (Pruning phase) 적용

Decision Tree에서 가지치기를 적용하는 과정또한 모델 구축에 대한 Hyper Parameter 설정 과정이라고 할 수 있다. 위에서 확인할 수 있었듯이, 재귀적인 분리는 Training set에서의 샘플을 완전하게 분류할 수 있었으며, 이는 과적합 문제라고 볼 수 있다. 일반화 성능을 높이기 위해서, 즉 Test set에서도 좋은 성능을 가지기 위해서 가지치기를 수행할 수 있다. 이는 max_depth를 직접 설정하여 아래의 그림과 같이 진행된다.

6) 가지치기 수행 (Pruning phase)

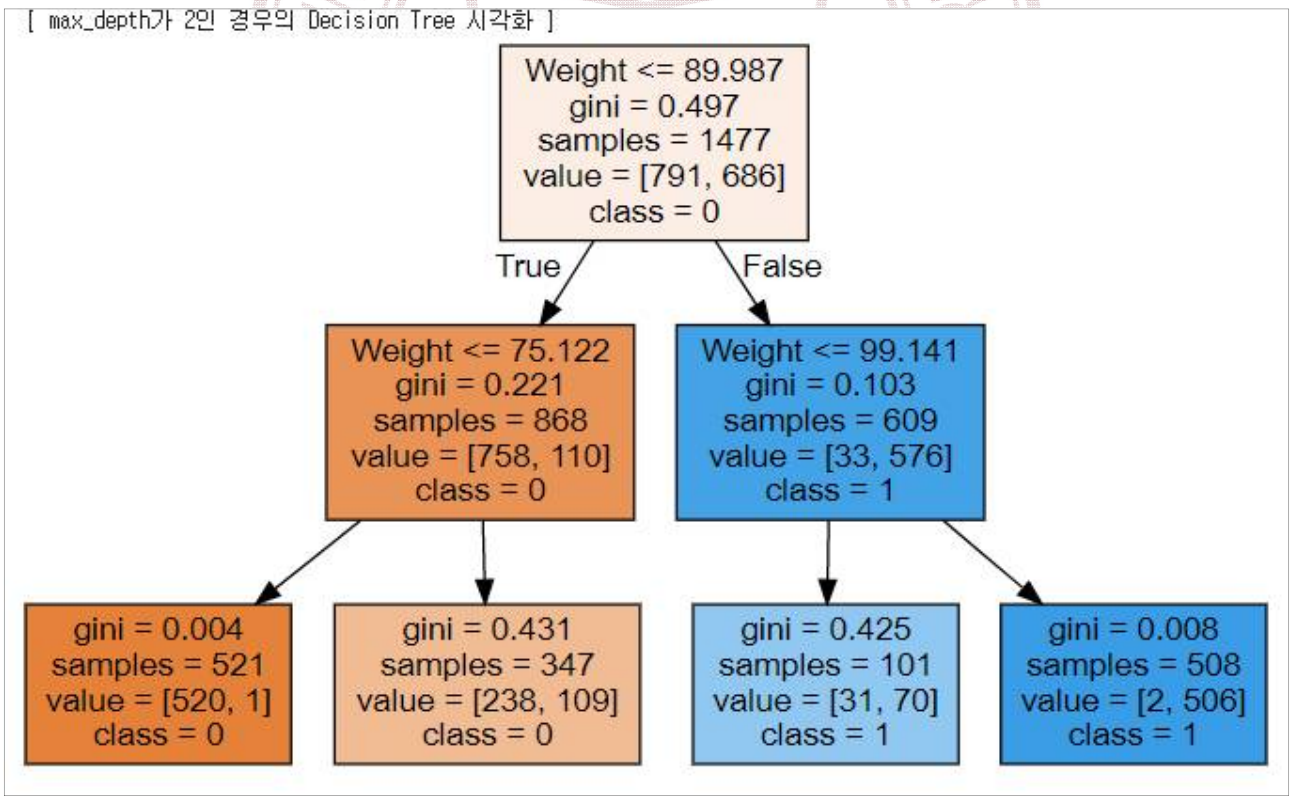
6-1) max_depth = 2

```
pruned_dt_clf = DecisionTreeClassifier(random_state=2022, max_depth=2) # 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: 0.903
Accuracy of test set: 0.905
```

```
# export_graphviz( )의 호출 결과로 out_file로 지정된 tree.dot 파일을 생성함
export_graphviz(pruned_dt_clf, out_file="prunedtree.dot", class_names = ['0','1'], feature_names = x_train.columns, impurity=True, fil
```

```
print('[ max_depth가 2인 경우의 Decision Tree 시각화 ]')
# 위에서 생성된 tree.dot 파일을 Graphviz 가 읽어서 시각화
with open("prunedtree.dot") as f:
    dot_graph = f.read()
graphviz.Source(dot_graph)
```



```
# 학습결과 평가
y_pred = pruned_dt_clf.predict(x_test)
print("Train_Accuracy : ", pruned_dt_clf.score(x_train, y_train), '₩n')
print("Test_Accuracy : ", pruned_dt_clf.score(x_test, y_test), '₩n')

print_pred_result(y_test, y_pred)
```

```
Train_Accuracy :  0.980365605958023

Test_Accuracy :  0.9842271293375394

Accuracy:  0.98

Recall:  0.98

Precision:  0.99

F1_score:  0.98

Confusion Matrix:
 [[345   3]
 [  7 279]]
```

- max_depth를 2로 설정하여 규제를 가한 위 코드와 Graphviz 시각화 자료를 확인할 수 있다. 이 때 변수 Weight만으로도 충분히 Sample을 분류해냈다. 예상대로 몸무게가 비만도를 판단하는데, 가장 중요한 변수가 된다는 것을 확인할 수 있다.
- Train Accuracy의 Score가 낮아졌지만, Test Accuracy의 Score는 더 높아졌고, 더 정확한 분류를 하는 것을 확인할 수 있다.

5. Decision Tree with Outliers

2-3) 이상치 랜덤하게 생성

```
y_df = df['class']
y_df = y_df.to_frame()
df.drop('class', axis=1, inplace=True)
```

```
outlier = -1
```

```
df.count().sum() * 0.05
```

```
2744.3
```

```
import random

for i in range(2744):
    r = random.randrange(0, 2111)
    c = random.randrange(0, 25)
    df.iloc[r,c] =  outlier
```

```
df = pd.concat([df, y_df],axis=1)
df.head()
```

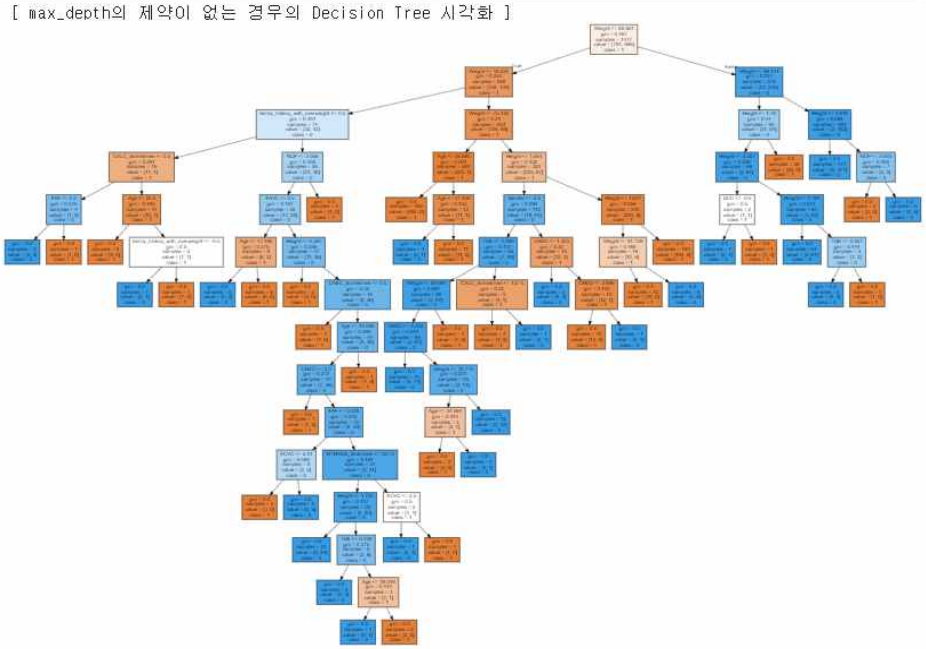
- 위 그림은 전체 관측치의 0.5% 수 만큼 random한 위치에 이상치(-1)을 생성해주는 코드이다. 이전의 전처리 과정은 동일하며 모델 구축 전 진행된 과정이다. 이는 기존의 모델 구축과 무엇이 달라지는지 실험하기 위한 과정이다. Dataset의 Split 과정과 Training 과정은 생략하였다.

3-3) Graphviz -> Tree 가시화

```
export_graphviz(dt_clf, out_file="tree.dot", class_names = ['1', '0'], feature_names = x_train.col
```

```
import graphviz
print('[ max_depth의 제약이 없는 경우의 Decision Tree 시각화 ]')
# 위에서 생성된 tree.dot 파일을 Graphviz 가 읽어서 시각화

with open("tree.dot") as f:
    dot_graph = f.read()
graphviz.Source(dot_graph)
```

```
[ max_depth의 제약이 없는 경우의 Decision Tree 시각화 ]
```



- 위 그림은 이상치를 포함하고 있는 Train Set에 대하여 Decision Tree 모델이 샘플을 분리해가는 과정을 보여준다. 이상치가 존재하지 않았던 데이터셋보다 더욱 복잡하고 가지를 많이 친 형태의 그림을 확인할 수 있다. **이상치 혹은 결측치를 포함하더라도,** Decision Tree는 **재귀적인 알고리즘**으로 완벽한 분류를 해나간다고 볼 수 있다.

4-1) Scoring and Confusion Matrix

```
# 학습결과 평가
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')

print_pred_result(y_test, y_pred)
```

```
Train_Accuracy :  1.0

Test_Accuracy :  0.9495268138801262

Accuracy:  0.95

Recall:  0.95

Precision:  0.94

F1_score:  0.94

Confusion Matrix:
 [[330  18]
 [ 14 272]]
```

- Train set이 이상치를 포함하고 있었음에도 불구하고, Train set에서의 정확도는 완벽하다. 하지만 이상치를 포함한 Test set에 대해서는 기존의 Decision Tree 모델과 비교했을 때 정확도가 많이 낮아진 것을 볼 수 있다. Confusion Matrix를 통해서도 오분류의 양이 늘어난 것을 확인할 수 있다.

5-2) 가지치기 (Pruning phase) 적용

6-2) max_depth = 3

```
pruned_dt_clf = DecisionTreeClassifier(random_state=2022, max_depth=3) # 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)))
```

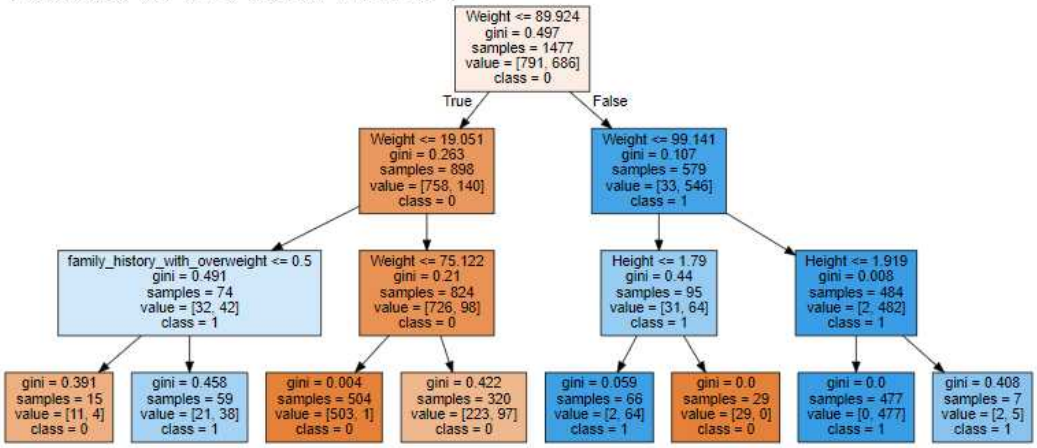
```
Accuracy of training set: 0.914
Accuracy of test set: 0.912
```

```
# export_graphviz( )의 호출 결과로 out_file로 지정된 tree.dot 파일을 생성함
export_graphviz(pruned_dt_clf, out_file="prunedtree.dot", class_names = ['0','1'], feature_names = x_train.columns, impurity=True, filled=True)
```

이상치를 포함한 데이터에 대해서도 마찬가지로 Hyper Parameter를 설정하여 모델링을 진행하였다.

```
print('[ max_depth가 3인 경우의 Decision Tree 시각화 ]')
# 위에서 생성된 tree.dot 파일을 Graphviz 가 읽어서 시각화
with open("prunedtree.dot") as f:
    dot_graph = f.read()
graphviz.Source(dot_graph)
```

```
[ max_depth가 3인 경우의 Decision Tree 시각화 ]
```




```
# 학습결과 평가
y_pred = pruned_dt_clf.predict(x_test)
print("Train_Accuracy : ", pruned_dt_clf.score(x_train, y_train), '₩n')  # train 에 과적합 시켰으니깐.
print("Test_Accuracy : ", pruned_dt_clf.score(x_test, y_test), '₩n')

print_pred_result(y_test, y_pred)
```

```
Train_Accuracy :  0.985781990521327

Test_Accuracy :  0.9763406940063092

Accuracy:  0.98

Recall:  0.98

Precision:  0.97

F1_score:  0.97

Confusion Matrix:
 [[339   9]
 [  6 280]]
```

- 위 그림은 max_depth를 3으로 설정하여 구축한 모델의 분류 과정과 그에 따른 성능을 보여준다. depth를 설정함으로써 일반화 성능을 높였기 때문에 Training Accuracy가 낮아진 것이 확인된다. 그에 반면, Test set의 Accuracy는 크게 향상되었고, 오분류의 양도 많이 감소했다. F1 score 등 다른 성능도 나쁘지 않은 것을 확인할 수 있다.

- 결과적으로 Decision Tree는 Training과정이 조금 길어지더라도 이상치, 결측치에 민감하게 반응하지 않고, 잘 분류해낼 수 있다는 장점을 가지고 있다고 판단할 수 있다. 추가적으로 위 과정을 통해 Decision Tree는 연속형 변수들에 대한 상대적인 크기 차이와 범주형 변수의 어떠한 Encoding 기법에도 민감하게 반응하지 않을 것이라고 예상해볼 수 있다.

!! Decision Tree는 분류의 과정을 직관적으로 확인할 수 있게 해준다는 것이 가장 큰 장점이라고 생각된다. 모델 구축 자체만으로도 다양한 Insight를 제공할 수 있다는 점에서 실무에서 무척 중요한 모델로 쓰일 것으로 예상된다. 또한 하이퍼 파라미터 설정은 더욱 보기 좋은 시각화 자료를 제공하며, 중요 변수를 쉽게 파악하여 변수 선택에도 도움을 줄 수 있었다. 무엇보다 중요한 점은 하이퍼파라미터 설정을 통해 모델의 일반화 성능을 높여서 Train set의 과적합을 방지할 수 있었다는 점이었다.

회고

1. 과제를 진행하면서 정말 많은 데이터를 만났다. Clustering 실습에 적합한 데이터인지를 파악하기 위해서는 전처리 과정을 거쳐야 했으며, 이러한 작업에 시간이 많이 소요됐다. Clustering과 Decision Tree에 대한 실습이 중요한 과제였지만, 다양한 데이터를 접하고 형태를 파악해 보는 것도 이번 과제에서 상당히 핵심적인 부분이었던 것 같다.

2. 비지도 학습이 어떤 식으로 진행될 수 있는지 알 수 있었고, 과제를 진행하기 위해서는 강의 영상도 반복해서 보아야 했다. 다양한 분류 모델의 특징과 그 안의 알고리즘이 어떻게 구현되는지에 대해서도 알 수 있었다. 알아야 할 것이 아직 한참 남았다는 것을 느꼈고, 더 관심을 갖고 다양한 모델을 다양한 데이터를 통해 다뤄보고 친숙해지는 과정이 필요할 것 같다.

3. 모델이 어떠한 알고리즘으로 구현되어 있는지에 대한 더 확실한 이해가 필요할 것 같다. 이론적으로는 각각의 모델이 어떻게 분류를 할 수 있는지에 대한 원리를 배웠지만, 분명 좀 더 알아야 할 부분이 있다. 단지 라이브러리에서 제공하는 모델을 가지고 실험해 보는 것에 그치지 않고, 그 안의 구현이 어떠한 원리로 되어있는지를 좀 더 구체적으로 알 필요가 있다고 느꼈다.

4. 비지도 학습의 분류에 적용되는 정성적인 평가는 흥미로웠다. 정답이 없는 부분이기 때문에, 합리적인 군집의 수를 찾아가는 과정은 매력적인 부분이었다.

5. 선형대수학에 대한 공부를 소홀하게 해서는 안된다는 것을 느꼈다. 아직 수업에서 다루지 않은 부분인 PCA의 원리와 과정을 제대로 알기 위해서는 선형대수학에서의 기본적인 이해가 바탕이 돼야 할 것 같다.