[Data Preprocessing]

1. target 별 파일 분할 & 각각 다른 메모리로 저장

targert=1, 2, 3, 4, 5, 6로 구성되어 있는 모든 파일들을 각각의 target 값 별로 파일을 구분하여 6개의 메모리에 나누어 저장하였다.

```
In [2]: path = './mlpr20_project_data_new/'
        file_list = os.listdir(path)

        file_list_1 = [file for file in file_list if file.endswith("1.txt")]
        file_list_2 = [file for file in file_list if file.endswith("2.txt")]
        file_list_3 = [file for file in file_list if file.endswith("3.txt")]
        file_list_4 = [file for file in file_list if file.endswith("4.txt")]
        file_list_5 = [file for file in file_list if file.endswith("5.txt")]
        file_list_6 = [file for file in file_list if file.endswith("6.txt")]
```

2. target 별 파일 모두 불러오기 & Dataframe으로 병합

target 별로 구성된 파일들을 불러와서 target 별 파일의 데이터들을 하나의 dataframe으로 병합하였다. 이렇게 총 6개의 dataframe을 생성하였다.

```
In [8]: data_1 = []

        for file in file_list_1:
            path = './mlpr20_project_data_new/' + file
            data = pd.read_csv(path, sep="₩t", names=['acc_x','acc_y','acc_z','ang_x','ang_y','ang_z'])
            data_1.append(data)

        dataframe_1 = pd.concat(data_1,ignore_index=True)
```

3. target column 추가

각 target 별 dataframe에 새로운 column을 만들어 해당 target 값을 입력해주었다.

```
In [26]: dataframe_1['target'] = 1
         dataframe_1.head(10)
```

Out[26]:

	acc_x	acc_y	acc_z	ang_x	ang_y	ang_z	target
0	1.471144	-0.348910	-0.127496	-0.861457	1.006809	0.257263	1
1	1.265634	-0.310253	-0.155189	-0.639981	2.044750	0.052422	1
2	1.123337	-0.214267	-0.095852	-0.773039	1.121777	-0.139702	1
3	0.633964	-0.088872	-0.062499	-1.172391	-0.404259	0.197850	1
4	0.721722	-0.138351	-0.146895	-1.527402	-0.857769	0.087846	1
5	0.928545	-0.116090	-0.113833	-1.372721	-0.544605	0.083144	1
6	0.982685	-0.316336	-0.188282	-1.416664	-0.849395	0.127491	1
7	0.955245	-0.350209	-0.224500	-0.985089	-0.637406	0.117450	1
8	0.866943	-0.424898	-0.082767	-0.355277	-0.822416	-0.137980	1
9	0.977844	-0.365781	-0.127592	-0.723367	-0.861126	0.043706	1

4. 모든 dataframe 병합

모든 dataframe을 다시 하나의 dataframe으로 병합하였다.

```
In [32]: dataframe_tot = pd.concat([dataframe_1, dataframe_2, dataframe_3, dataframe_4, dataframe_5, dataframe_6])
```

5. dataframe 분할 (train data / target data)

dataframe의 데이터를 train data와 target data로 나누었다. 이는 train data만 따로 standardization을 진행하기 위함이다.

```
In [36]: df_split = np.split(dataframe_tot, [6], axis=1)
```

```
In [37]: df_split[0]
```

Out[37]:

	acc_x	acc_y	acc_z	ang_x	ang_y	ang_z
0	1.471144	-0.348910	-0.127496	-0.861457	1.006809	0.257263
1	1.265634	-0.310253	-0.155189	-0.639981	2.044750	0.052422
2	1.123337	-0.214267	-0.095852	-0.773039	1.121777	-0.139702
3	0.633964	-0.088872	-0.062499	-1.172391	-0.404259	0.197850
4	0.721722	-0.138351	-0.146895	-1.527402	-0.857769	0.087846
...	...	...	...	...	...	...
171131	0.108195	0.851331	0.526397	-0.000174	0.004444	0.007352
171132	0.108080	0.853172	0.525278	-0.000484	0.000105	0.006160
171133	0.112025	0.852866	0.521009	-0.000423	0.004464	-0.002110
171134	0.112034	0.854851	0.531059	0.000244	0.003737	-0.003800
171135	0.112305	0.856970	0.532894	0.000152	0.008915	-0.006741

892032 rows × 6 columns

```
In [38]: df_split[1]
```

Out[38]:

	target
0	1.0
1	1.0
2	1.0
3	1.0
4	1.0
...	...
171131	6.0
171132	6.0
171133	6.0
171134	6.0
171135	6.0

892032 rows × 1 columns

6. Train data standardization

모델은 데이터의 standardization의 여부에 영향을 많이 받는다. 따라서 standardization을 진행해주었으며 이때 PCA를 이용하였다.

```
In [43]: from sklearn.preprocessing import StandardScaler

         # Standardizing the features
         x_std = StandardScaler().fit_transform(df_x)
```

7. Dataframe 병합 (train data / target data)

Standardization까지 완료된 train data와 target data를 다시 하나의 dataframe으로 병합하였다. 이는 훈련/검증 세트를 나누기 위함이다.

```
In [55]: df_x_std.reset_index(drop=True, inplace=True)
         df_y.reset_index(drop=True, inplace=True)
         df_tot = pd.concat([df_x_std, df_y], axis=1)
```

8. 훈련/검증 데이터셋 준비

train data : validation data = 0.7 : 0.3 의 비율로 두 데이터셋을 나누었다. 7:3 비율은 모델을 훈련하고 검증할 때 가장 최적의 비율이라고 알려져 있다.

9. Dataframe 분할 (train data / target data)

추후 모델 training 시에 각각의 데이터를 활용하기 편하도록 train data와 target data를 나누어 각각 dataframe으로 만들었다. Train dataset, validation dataset 모두에 진행하였다. 그 후 4개의 csv 파일로 각각의 dataframe을 저장하였다.

```
In [63]: train_split = np.split(train_set, [6], axis=1)
In [64]: train_split[0]
```

[k-Nearest Neighbors (k-NN)]

1. Hyperparameter tuning – k (Feature Extraction (O))

데이터 preprocessing 과정에서 feature extraction을 한 데이터셋을 이용하여 kNN 모델을 이용한 분류작업을 진행하였다. 첫 모델로는 일단 파라미터 k=5로 설정하여 실험을 진행했다.

1) k=5 (Accuracy: 61.28%)

정확도가 61.28로 상당히 낮은 정확도를 보였다. k를 더 낮추면 overfitting의 우려가 되어 일단 첫 모델과 비교 차 k값을 조금만 증가시켜 비교해보기로 했다.

Generating Model for k=5

```
In [16]: #Import knearest neighbors Classifier model
         from sklearn.neighbors import KNeighborsClassifier
```

```
In [17]: #Create KNN Classifier
         knn = KNeighborsClassifier(n_neighbors=5)

         #Train the model using the training sets
         knn.fit(x_train_FE, y_train)
```

```
C:₩Users₩JinsolKim₩Anaconda3₩lib₩site-packages₩ipykernel_launcher.py:5: DataConversi
onWarning: A column-vector y was passed when a 1d array was expected. Please change
the shape of y to (n_samples, ), for example using ravel().
  """
```

```
Out[17]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                              metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                              weights='uniform')
```

```
In [19]: #Predict the response for test dataset
         y_pred = knn.predict(x_test_FE)
```

Model Evaluation for k=5

```
In [20]: #Import scikit-learn metrics module for accuracy calculation
         from sklearn import metrics

         # Model Accuracy, how often is the classifier correct?
         print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.6128395799858002
```

2) k=7 (Accuracy: 62.00%)

k=5에서 7로 k값을 증가시켰음에도 불구하고 정확도가 상승하였다. 이는 k값의 변화가 크지 않아 training 과정에서 달라질 수 있는 차이라 생각이 되어 k값을 더 증가시켜 보았다.

Re-generating Model for K=7

In [21]:
```
#Import knearest neighbors Classifier model
from sklearn.neighbors import KNeighborsClassifier

#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=7)

#Train the model using the training sets
knn.fit(x_train_FE, y_train)

#Predict the response for test dataset
y_pred = knn.predict(x_test_FE)
```

```
C:₩Users₩JinsolKim₩Anaconda3₩lib₩site-packages₩ipykernel_launcher.py:8: DataConversi
onWarning: A column-vector y was passed when a 1d array was expected. Please change
the shape of y to (n_samples, ), for example using ravel().
```

In [22]:
```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.6200291468928665
```

3) k=10 (Accuracy: 62.59%)

k값을 증가시켰는데 적은 차이지만 정확도가 또 올랐다. 그래서 아예 k값을 크게 증가시켜 보기로 했다.

Re-generating Model for K=10

In [23]:
```
#Import knearest neighbors Classifier model
from sklearn.neighbors import KNeighborsClassifier

#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=10)

#Train the model using the training sets
knn.fit(x_train_FE, y_train)

#Predict the response for test dataset
y_pred = knn.predict(x_test_FE)
```

```
C:₩Users₩JinsolKim₩Anaconda3₩lib₩site-packages₩ipykernel_launcher.py:8: DataConversi
onWarning: A column-vector y was passed when a 1d array was expected. Please change
the shape of y to (n_samples, ), for example using ravel().
```

In [24]:
```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.6259781024625388
```

4) k=100 (Accuracy: 64.74%)

k값을 100으로 증가시켰더니 전보다는 상대적으로 큰 정확도가 상승했다. k값을 훨씬 크게 증가시키기로 했다.

Re-generating Model for K=100

In [25]:
```
#Import knearest neighbors Classifier model
from sklearn.neighbors import KNeighborsClassifier

#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=100)

#Train the model using the training sets
knn.fit(x_train_FE, y_train)

#Predict the response for test dataset
y_pred = knn.predict(x_test_FE)
```

```
C:₩Users₩JinsolKim₩Anaconda3₩lib₩site-packages₩ipykernel_launcher.py:8: DataConversi
onWarning: A column-vector y was passed when a 1d array was expected. Please change
the shape of y to (n_samples, ), for example using ravel().
```

In [26]:
```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.6473637009080379
```

5) k=500 (Accuracy: 65.01%)

마찬가지로 k값을 500으로 증가시켰더니 정확도가 증가했다.

Re-generating Model for K=500

In [27]:
```
#Import knearest neighbors Classifier model
from sklearn.neighbors import KNeighborsClassifier

#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=500)

#Train the model using the training sets
knn.fit(x_train_FE, y_train)

#Predict the response for test dataset
y_pred = knn.predict(x_test_FE)
```

```
C:₩Users₩JinsolKim₩Anaconda3₩lib₩site-packages₩ipykernel_launcher.py:8: DataConversi
onWarning: A column-vector y was passed when a 1d array was expected. Please change
the shape of y to (n_samples, ), for example using ravel().
```

In [28]:
```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.6501251821680805
```

6) k=1000

k값을 1000으로 설정했더니 '메모리 부족'으로 실행이 되지 않아 아쉬웠다.

Re-generating Model for K=1000

In [30]:
```
#Import knearest neighbors Classifier model
from sklearn.neighbors import KNeighborsClassifier

#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=1000)

#Train the model using the training sets
knn.fit(x_train_FE, y_train)

#Predict the response for test dataset
y_pred = knn.predict(x_test_FE)
```

```
C:₩Users₩JinsolKim₩Anaconda3₩lib₩site-packages₩ipykernel_launcher.py:8: DataConversi
onWarning: A column-vector y was passed when a 1d array was expected. Please change
the shape of y to (n_samples, ), for example using ravel().
```

```
---------------------------------------------------------------------------
-
MemoryError                               Traceback (most recent call last)
<ipython-input-30-91c05e7acb23> in <module>
      9
     10 #Predict the response for test dataset
---> 11 y_pred = knn.predict(x_test_FE)
```

2. Hyperparameter tuning – k (Feature Extraction (X))

이번에는 feature extraction을 진행하지 않은 데이터셋을 이용하여 kNN 분석을 진행해보았다. feature extraction을 진행했을 경우와 비교를 하기 위해 파라미터 k=[5, 7, 10, 100, 500, 1000]로 순차적으로 변경해가며 실험을 진행하였다.

1) k=5 (Accuracy: 79.08%)

Feature extraction을 안 한 모델로 kNN을 적용해봤더니 예상과는 다르게 feature extraction을 했을 때보다 오히려 정확도가 훨씬 올라간 79.08%를 기록하였다.

Generating Model for k=5

In [7]:
```
#Import knearest neighbors Classifier model
from sklearn.neighbors import KNeighborsClassifier
```

In [8]:
```
#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=5)

#Train the model using the training sets
knn.fit(x_train, y_train)
```

```
C:₩Users₩Owner₩anaconda3₩lib₩site-packages₩ipykernel_launcher.py:5: DataConversionWa
rning: A column-vector y was passed when a 1d array was expected. Please change the
shape of y to (n_samples, ), for example using ravel().
  """
```

Out [8]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

In [9]:
```
#Predict the response for test dataset
y_pred = knn.predict(x_test)
```

Model Evaluation for k=5

In [10]:
```
#Import scikit-learn metrics module for accuracy calculation
from sklearn import metrics

# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.7908037816225104
```

2) k=7 (Accuracy: 79.57%)

Feature extraction을 안 했을 때와 마찬가지로 k값을 증가시키니 정확도가 향상하였다.

Re-generating Model for K=7

In [11]:
```
#Import knearest neighbors Classifier model
from sklearn.neighbors import KNeighborsClassifier

#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=7)

#Train the model using the training sets
knn.fit(x_train, y_train)

#Predict the response for test dataset
y_pred = knn.predict(x_test)
```

```
C:₩Users₩Owner₩anaconda3₩lib₩site-packages₩ipykernel_launcher.py:8: DataConversionWa
rning: A column-vector y was passed when a 1d array was expected. Please change the
shape of y to (n_samples, ), for example using ravel().
```

In [12]:
```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.7956503867568476
```

3) k=10 (Accuracy: 79.93%)

Feature extraction을 하지 않았을 때와 마찬가지로 k값이 증가하면서 정확도가 향상되는 경향을 보이고 있다.

Re-generating Model for K=10

In [13]:
```
#Import knearest neighbors Classifier model
from sklearn.neighbors import KNeighborsClassifier

#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=10)

#Train the model using the training sets
knn.fit(x_train, y_train)

#Predict the response for test dataset
y_pred = knn.predict(x_test)
```

```
C:₩Users₩Owner₩anaconda3₩lib₩site-packages₩ipykernel_launcher.py:8: DataConversionWa
rning: A column-vector y was passed when a 1d array was expected. Please change the
shape of y to (n_samples, ), for example using ravel().
```

In [14]:
```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.7992862748028848
```

4) k=100 (Accuracy: 80.40%)

k=100일 때 가장 높은 정확도인 80.40%를 기록하였다.

Re-generating Model for K=100

In [15]:
```
#Import knearest neighbors Classifier model
from sklearn.neighbors import KNeighborsClassifier

#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=100)

#Train the model using the training sets
knn.fit(x_train, y_train)

#Predict the response for test dataset
y_pred = knn.predict(x_test)
```

```
C:₩Users₩Owner₩anaconda3₩lib₩site-packages₩ipykernel_launcher.py:8: DataConversionWa
rning: A column-vector y was passed when a 1d array was expected. Please change the
shape of y to (n_samples, ), for example using ravel().
```

In [16]:
```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.8040170397219835
```

5) k=500 (Accuracy: 79.07%)

k=500으로 설정하니 처음으로 k값이 커지면서 성능이 낮아지는 상황이 발생하였다.

Re-generating Model for K=500

In [17]:
```
#Import knearest neighbors Classifier model
from sklearn.neighbors import KNeighborsClassifier

#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=500)

#Train the model using the training sets
knn.fit(x_train, y_train)

#Predict the response for test dataset
y_pred = knn.predict(x_test)
```

```
C:₩Users₩Owner₩anaconda3₩lib₩site-packages₩ipykernel_launcher.py:8: DataConversionWa
rning: A column-vector y was passed when a 1d array was expected. Please change the
shape of y to (n_samples, ), for example using ravel().
```

In [18]:
```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.7907103620940922
```

6) k=1000 (Accuracy: 78.07%)

k를 500과 1000으로 설정하니 처음에 'k값이 커질수록 정확도가 낮아질 것이다.'라고 원래 예상했던 대로 값이 낮아지는 경향을 보이기 시작했다. 이에 따라 본 데이터셋을 kNN으로 분석하고자 할 때는 k가 대략 100일 때 가장 최적의 성능을 기록하는 것을 알 수 있었다.

Re-generating Model for K=1000

In [19]:
```
#Import knearest neighbors Classifier model
from sklearn.neighbors import KNeighborsClassifier

#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=1000)

#Train the model using the training sets
knn.fit(x_train, y_train)

#Predict the response for test dataset
y_pred = knn.predict(x_test)
```

```
C:₩Users₩Owner₩anaconda3₩lib₩site-packages₩ipykernel_launcher.py:8: DataConversionWa
rning: A column-vector y was passed when a 1d array was expected. Please change the
shape of y to (n_samples, ), for example using ravel().
```

In [20]:
```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.7807368932401629
```

Feature extraction을 통해 유용한 정보를 추출해 이를 바탕으로 학습 성능을 올려줄 새로운 feature를 생성한다고 알고 있어서, feature extraction을 하면 무조건 성능이 좋아질 줄로만 생각하고 있었다. 하지만 원본 데이터의 특성에 따라 feature extraction를 통해 데이터를 축소시키는 것보다 기존의 원본 데이터를 그대로 사용하는 것이 모델의 성능 향상에 도움이 될 수도 있다는 사실을 깨달았다.

3. Hyperparameter tuning – weights

지금까지 feature extraction을 하지 않은 데이터셋을 이용해서 k=100일 때 가장 최적의 성능을 보인다는 사실을 알아냈다. 이제 이러한 조건을 유지한 채로 kNN의 다른 parameter인 weights를 조정해 모델의 성능이 어떻게 변하는지 확인하고자 하였다. kNN에서 데이터 간의 거리를 어떠한 방식으로 기준을 잡고 측정하느냐에 따라 새로 들어올 데이터의 분류가 달라지기에 distance parameter tuning은 매우 중요하다.

아무 설정을 하지 않았던 기존에는 default값인 'uniform'으로 설정되어 있었지만, 이번에는

직접 파라미터를 'distance'를 지정하여 실험을 진행하였다. 이 때, 'uniform'은 이웃 간에 가중치가 동일하게 매겨지는 방식이며, 'distance'는 가까운 이웃이라면 먼 이웃보다 가중치를 더 크게 부여하는 방식이다. 따라서 'distance'으로 파라미터를 변경하면 가까운 이웃끼리 서로를 더욱 잘 분류하는 학습을 하게 됨에 따라 학습의 성능이 더 높아질 것으로 예상하고 실험을 진행하였다.

1) weights='distance' (Accuracy: 80.56%)

실험 결과, 예상대로 weights='uniform'이었을 때 (Accuracy: 80.40%)보다 'distance'일 때 정확도가 80.56%로 더 상승한 것을 확인할 수 있었다. 하지만 예상한 것보다는 그렇게 큰 차이를 보이지는 않았다.

```
In [12]: #Import knearest neighbors Classifier model
         from sklearn.neighbors import KNeighborsClassifier

         #Create KNN Classifier
         knn = KNeighborsClassifier(n_neighbors=100, weights='distance')

         #Train the model using the training sets
         knn.fit(x_train, y_train)

         #Predict the response for test dataset
         y_pred = knn.predict(x_test)
```

```
C:₩Users₩Owner₩anaconda3₩lib₩site-packages₩ipykernel_launcher.py:8: DataConversionWa
rning: A column-vector y was passed when a 1d array was expected. Please change the
shape of y to (n_samples, ), for example using ravel().
```

```
In [13]: print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.805552856769179
```

4. Hyperparameter tuning – algorithm

kNN의 파라미터 중 실제로 이웃을 결정하는 알고리즘의 종류가 달라지면 혹시 성능에 차이가 나타날까 싶어 algorithm에 따른 성능변화를 살펴보았다. 기존에는 별다른 설정을 하지않아 default 값인 'auto'로 설정되어 모든 알고리즘 중 가장 최적이라 판단되는 알고리즘을 자동으로 적용해주도록 하였었다. 이번에는 직접 'kd_tree', 'ball_tree', 'brute' 3가지 알고리즘을 변화시켜 가며 성능의 차이를 비교하였다.

1) algorithm='kd_tree' (Accuracy: 80.56%)

K-D tree (K-Dimensinal Tree) 알고리즘의 기본 아이디어는 'A가 B와 아주 멀고, B와 C가 가깝다면, A와 C의 거리는 아주 멀다'라는 개념을 바탕으로 설계된다. K-D Tree의 경우 일부 특정 속성을 대상으로 데이터의 이진 분류가 이루어진다. 따라서 실제로는 유클리디안 거리가 더 먼 training 데이터끼리 함께 분류될 수도 있기에 다른 algorithm에 비해 학습 성능이 낮을 것이라 예측하였다. 하지만 성능은 algorithm='auto'일 때와 다름없이 정확도가 정확히 똑같이 나왔다.

In [14]:
```
#Import knearest neighbors Classifier model
from sklearn.neighbors import KNeighborsClassifier

#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=100, weights='distance', algorithm='kd_tree')

#Train the model using the training sets
knn.fit(x_train, y_train)

#Predict the response for test dataset
y_pred = knn.predict(x_test)
```

```
C:₩Users₩Owner₩anaconda3₩lib₩site-packages₩ipykernel_launcher.py:8: DataConversionWa
rning: A column-vector y was passed when a 1d array was expected. Please change the
shape of y to (n_samples, ), for example using ravel().
```

In [15]:
```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.805552856769179
```

2) algorithm='ball_tree' (Accuracy: 80.56%)

algorithm='auto', 'kd_tree'일 때와 같은 정확도 수치를 기록하였다.

In [20]:
```
#Import knearest neighbors Classifier model
from sklearn.neighbors import KNeighborsClassifier

#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=100, weights='distance', algorithm='ball_tree')

#Train the model using the training sets
knn.fit(x_train, y_train)

#Predict the response for test dataset
y_pred = knn.predict(x_test)
```

```
C:₩Users₩Owner₩anaconda3₩lib₩site-packages₩ipykernel_launcher.py:8: DataConversionWarning: A column-vector y
was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using rav
el().
```

In [21]:
```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.805552856769179
```

3) algorithm='brute' (Accuracy: 80.56%)

algorithm='auto', 'kd_tree', 'brute'일 때와 같은 정확도 수치를 기록하였다.

```
In [22]: #Import knearest neighbors Classifier model
         from sklearn.neighbors import KNeighborsClassifier

         #Create KNN Classifier
         knn = KNeighborsClassifier(n_neighbors=100, weights='distance', algorithm='brute')

         #Train the model using the training sets
         knn.fit(x_train, y_train)

         #Predict the response for test dataset
         y_pred = knn.predict(x_test)

         C:₩Users₩Owner₩anaconda3₩lib₩site-packages₩ipykernel_launcher.py:8: DataConversionWarning: A column-vector y
         was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using rav
         el().

In [23]: print("Accuracy:",metrics.accuracy_score(y_test, y_pred))

         Accuracy: 0.805552856769179
```

algorithm 파라미터를 무엇을 선택하든 'auto', 'kd_tree', 'brute', 'auto'일 때의 모델 모두 같은 성능을 보였다. 따라서 algorithm 파라미터는 모델의 성능에 영향을 끼치지 않는 것으로 나타났다. 단지 트리를 구성하는 방식에 따라 모델 training에 속도 차이가 있지 않을까 하는 생각이 든다.

5. Hyperparameter tuning – leaf_size

Leaf size는 Ball Tree나 K-D Tree에 전달하는 leaf의 크기를 말하며 즉, 트리에서 각각 몇 대로 뻗어 나갈지 나타내는 값이다. Leaf size가 너무 작으면 가지수가 많아져 노이즈가 끼기 쉽고 속도가 느려진다. 반대로 Leaf size가 너무 크면 분류를 정확하게 하지 않아 예측 성능이 낮아지게 된다. Default 값은 30으로 leaf size를 기존의 실험보다 낮은 10과 높은 50으로 변경하여 실험하였다.

1) leaf_size=10 (Accuracy: 80.56%)

Leaf size를 30에서 10으로 낮추어 줬으나 leaf size가 30일 때인 정확도 80.56%와 같은 성능을 보였다.

In [18]:
```
#Import knearest neighbors Classifier model
from sklearn.neighbors import KNeighborsClassifier

#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=100, weights='distance', algorithm='kd_tree', leaf_size=10)

#Train the model using the training sets
knn.fit(x_train, y_train)

#Predict the response for test dataset
y_pred = knn.predict(x_test)
```

```
C:₩Users₩Owner₩anaconda3₩lib₩site-packages₩ipykernel_launcher.py:8: DataConversionWarning: A column-vector y
was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using rav
el().
```

In [19]:
```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.805552856769179
```

2) leaf_size=50 (Accuracy: 80.56%)

Leaf size를 30에서 50으로 증가시켰을 때에도 정확도에 변화가 없었다.

In [22]:
```
#Import knearest neighbors Classifier model
from sklearn.neighbors import KNeighborsClassifier

#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=100, weights='distance', algorithm='brute')

#Train the model using the training sets
knn.fit(x_train, y_train)

#Predict the response for test dataset
y_pred = knn.predict(x_test)
```

```
C:₩Users₩Owner₩anaconda3₩lib₩site-packages₩ipykernel_launcher.py:8: DataConversionWarning: A column-vector y
was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using rav
el().
```

In [23]:
```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.805552856769179
```

6. Conclusion

No.	Feature Extraction	Parameter Settings	Accuracy (%)
1	O	k=5	61.28
2		k=7	62.00
3		k=10	62.59
4		k=100	64.74
5		k=500	(Memory Error)
6		k=1000	79.08
7	X	k=5	79.57

8		k=7	79.93
9		k=10	80.40
10		k=100	79.07
11		k=500	78.07
12		k=1000	80.40
13		**k=100, weights='distance'**	**80.56**
14		k=100, weights='distance', algorithm='kd_tree'	80.56
15		k=100, weights='distance', algorithm='ball_tree'	80.56
16		k=100, weights='distance', algorithm='brute	80.56
17		k=100, weights='distance', leaf_size=10	80.56
18		k=100, weights='distance', leaf_size=50	80.56

(주) Parameter settings 중 특별히 표시하지 않은 parameter들은 default 값

k-Nearest Neighbors에서 k 값이 증가할 수록 높은 정확도를 기록하였으며, feature extraction을 하지 않았을 때 정확도가 훨씬 개선되었다. 또한 weights는 'uniform'이 아닌 'distance'일 때 더 높은 성능을 보였다. 이 밖에 algorithm과 leaf_size는 정확도 개선에 별다른 영향을 주지 않았다.

가장 높은 성능은 k=100, weights='distance'인 파라미터를 가진 모델에서 나왔다. 물론 이러한 파라미터 조건에서 algorithm과 leaf_size의 파라미터를 default값이 아닌 변화를 주어도 default값일 때와 같이 최고 성능을 보였다. 하지만, 최종 모델 선정시에는 최적의 algorithm을 자동으로 선정해주는 'auto'인 default값을 그대로 유지하기로 하였고, 또한 leaf_size도 default로 유지하기로 결정하였다.

[Random Forest]

1. Hyperparameter tuning – n_estimators (Feature Extraction (O))

n_estimators 파라미터는 random forest 안의 결정트리 갯수를 결정한다. 결정트리가 많을수록 더 깔끔한 decision boundary가 나오므로 n_estimators는 클수록 좋다. 하지만 그만큼 메모리와 훈련시간이 증가하고 overfitting이 될 가능성이 있는 단점이 있다. 따라서 우선 n_estimators를 100부터 시작해서 최적의 값을 찾고자 하였다.

1) n_estimators=100 (Accuracy: 61.5%)

n_estimators=100인 경우 61.5%의 매우 낮은 성능을 보였다. 혹시 k-NN처럼 feature extraction을 하지 않으면 성능이 높아질까 생각하여 feature extraction을 하지 않고 모델을 테스트해보기로 하였다.

In [15]:
```
from sklearn.ensemble import RandomForestClassifier
```

In [17]:
```
rf = RandomForestClassifier(n_estimators=100, oob_score=True, random_state=0)
```

3. Model Training

In [21]:
```
rf.fit(x_train_FE, y_train)
```

```
C:₩Users₩JinsolKim₩Anaconda3₩lib₩site-packages₩ipykernel_launcher.py:1: DataConversionWarning: A co
lumn-vector y was passed when a 1d array was expected. Please change the shape of y to (n_sample
s,), for example using ravel().
  """Entry point for launching an IPython kernel.
```

Out[21]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=True, random_state=0, verbose=0,
                       warm_start=False)
```

4. Model Testing

In [20]:
```
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
```

In [23]:
```
pred = rf.predict(x_test_FE)
accuracy = accuracy_score(y_test, pred)

print(f'Out-of-bag score estimate: {rf.oob_score_:.3}')
print(f'Mean accuracy score: {accuracy:.3}')
```

```
Out-of-bag score estimate: 0.614
Mean accuracy score: 0.615
```

2. Hyperparameter tuning – n_estimators (Feature Extraction (X))

1) n_estimators=100 (Accuracy: 81.3%)

Feature extraction을 진행하지 않은 데이터셋을 이용해 random forest를 진행하였더니 예상했던 바와 같이 정확도가 61.5%에서 81.3%로 크게 상승하였다.

In [7]:
```
from sklearn.ensemble import RandomForestClassifier
```

In [8]:
```
rf = RandomForestClassifier(n_estimators=100, oob_score=True, random_state=0)
```

3. Model Training

In [9]:
```
rf.fit(x_train, y_train)
```

```
C:₩Users₩JinsolKim₩Anaconda3₩lib₩site-packages₩ipykernel_launcher.py:1: DataConversionWarning: A co
lumn-vector y was passed when a 1d array was expected. Please change the shape of y to (n_sample
s,), for example using ravel().
  """Entry point for launching an IPython kernel.
```

Out[9]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=True, random_state=0, verbose=0,
                       warm_start=False)
```

4. Model Testing

In [10]:
```
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
```

In [11]:
```
pred = rf.predict(x_test)
accuracy = accuracy_score(y_test, pred)

print(f'Out-of-bag score estimate: {rf.oob_score_:.3}')
print(f'Mean accuracy score: {accuracy:.3}')
```

```
Out-of-bag score estimate: 0.808
Mean accuracy score: 0.813
```

2) n_estimators=200 (Accuracy: 81.41%)

n_estimators의 값 100에서 200으로 증가시켰더니 정확도가 소폭 상승하였다.

```
In [7]: from sklearn.ensemble import RandomForestClassifier

In [9]: rf = RandomForestClassifier(n_estimators=200)
```

3. Model Training

```
In [10]: rf.fit(x_train, y_train)

C:₩Users₩Owner₩anaconda3₩lib₩site-packages₩ipykernel_launcher.py:1: DataConversionWarning: A column-vector y
was passed when a 1d array was expected. Please change the shape of y to (n_samples,), for example using rave
l().
  """Entry point for launching an IPython kernel.

Out[10]: RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=200,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

4. Model Testing

```
In [11]: from sklearn.metrics import accuracy_score
         from sklearn.metrics import confusion_matrix

In [12]: pred = rf.predict(x_test)
         accuracy = accuracy_score(y_test, pred)

         print(f'Mean accuracy score: {accuracy:.5}')

         Mean accuracy score: 0.81413
```

3) n_estimators=500 (Accuracy: **81.49%**) – *Best Performance!!*

성능을 더 높이고자 n_estimators 파라미터의 값을 크게 높여주어 실험하였다. 예상대로 정확도는 올라갔지만 생각만큼 크게 오르지는 않았다.

In [8]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix

rf = RandomForestClassifier(n_estimators=500)
rf.fit(x_train, y_train)
```

```
C:₩Users₩Owner₩anaconda3₩lib₩site-packages₩ipykernel_launcher.py:6: DataConversionWarning: A column-vector y
was passed when a 1d array was expected. Please change the shape of y to (n_samples,), for example using rave
l().
```

Out[8]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=500,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

In [9]:
```
pred = rf.predict(x_test)
accuracy = accuracy_score(y_test, pred)

print(f'Mean accuracy score: {accuracy:.5}')
```

```
Mean accuracy score: 0.81488
```

4) n_estimators=800 (Accuracy: 81.48%)

더 큰 비율로 n_estimators의 값을 높이고자 1000으로 설정하여 실험을 진행하였으나 'Memory Error'가 뜨면서 중단되었다. 따라서 살짝 낮추어 800으로 진행하였다. 그 결과 예상외로 n_estimators가 500일 때보다 정확도가 조금 떨어졌다. 따라서 본 학습데이터에 가장 최적인 n_estimators의 값을 500으로 선정하였다.

In [8]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix

rf = RandomForestClassifier(n_estimators=800)
rf.fit(x_train, y_train)
```

```
C:₩Users₩Owner₩anaconda3₩lib₩site-packages₩ipykernel_launcher.py:6: DataConversionWarning: A column-vector y
was passed when a 1d array was expected. Please change the shape of y to (n_samples,), for example using rave
l().
```

Out[8]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=800,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

In [9]:
```
pred = rf.predict(x_test)
accuracy = accuracy_score(y_test, pred)

print(f'Mean accuracy score: {accuracy:.5}')
```

```
Mean accuracy score: 0.81481
```

3. Hyperparameter tuning – min_samples_leaf & min_samples_split

Random forest의 hyperparameter 중 min_samples_leaf와 min_samples_split의 값을 조정해 보았다. 각각 하나의 node를 가를 때 필요한 최소한의 sample 수와 하나의 leaf에 존재해야 할 sample의 수를 의미한다. 우석 default값이 min_samples_leaf는 2, min_samples_split는 1이므로 한 단위씩 변경하여 성능의 변화를 관찰하였다.

1) min_samples_leaf=2 (default), min_samples_split=2 (Accuracy: 81.47%)

min_samples_split을 1에서 2로 증가시켜 모델을 학습시켰더니 성능이 81.49%에서 81.47%로 미세하게 떨어진 것을 확인하였다.

```
In [9]: from sklearn.ensemble import RandomForestClassifier

In [13]: rf = RandomForestClassifier(n_estimators=500, min_samples_leaf=2, min_samples_split=2)
```

3. Model Training

```
In [14]: rf.fit(x_train, y_train)

C:₩Users₩JinsolKim₩Anaconda3₩lib₩site-packages₩ipykernel_launcher.py:1: DataConversionWarning: A co
lumn-vector y was passed when a 1d array was expected. Please change the shape of y to (n_sample
s,), for example using ravel().
  """Entry point for launching an IPython kernel.

Out[14]: RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                        criterion='gini', max_depth=None, max_features='auto',
                        max_leaf_nodes=None, max_samples=None,
                        min_impurity_decrease=0.0, min_impurity_split=None,
                        min_samples_leaf=2, min_samples_split=2,
                        min_weight_fraction_leaf=0.0, n_estimators=500,
                        n_jobs=None, oob_score=False, random_state=None,
                        verbose=0, warm_start=False)
```

4. Model Testing

```
In [15]: from sklearn.metrics import accuracy_score

In [16]: pred = rf.predict(x_test)
         accuracy = accuracy_score(y_test, pred)

         print(f'Mean accuracy score: {accuracy:.5}')

         Mean accuracy score: 0.81468
```

2) min_samples_leaf=1, min_samples_split=2

'Memory Error'로 학습이 중단되었다.

```
In [10]: from sklearn.ensemble import RandomForestClassifier

In [11]: rf = RandomForestClassifier(n_estimators=500, min_samples_leaf=1, min_samples_split=2)
```

3. Model Training

```
In [12]: rf.fit(x_train, y_train)

C:₩Users₩JinsolKim₩Anaconda3₩lib₩site-packages₩ipykernel_launcher.py:1: DataConversionWarning: A co
lumn-vector y was passed when a 1d array was expected. Please change the shape of y to (n_sample
s,), for example using ravel().
  """Entry point for launching an IPython kernel.

---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-12-fc0aeedd5571> in <module>
----> 1 rf.fit(x_train, y_train)
```

4. Conclusion

No.	Feature Extraction	Parameter Settings	Accuracy (%)
1	O	n_estimators=100	61.5
2	X	n_estimators=100	81.3
3		n_estimators=200	81.41
4		n_estimators=500	81.49
5		n_estimators=800	81.48
6		n_estimators=500, min_samples_leaf=2 (default), min_samples_split=2	81.47
7		n_estimators=500, min_samples_leaf=2, min_samples_split=2	(Memory Error)

(주) Parameter settings 중 특별히 표시하지 않은 parameter들은 default 값

Random Forest에서는 파라미터 n_estimators 값이 증가할수록 모델의 성능이 높아지는 것으로 나타났다. k값이 너무 작으면 거리가 너무 가까운 데이트들을 기준으로 새로운 데이터를 분류하므로 overfitting이 일어날 가능성이 커진다. 그러나 k값이 너무 크면 분류 기준이 과하게 일반화되어 새로운 데이터를 정밀하게 분류하지 못해 underfitting이 일어나 분류 모델의 성능이 낮아질 수 있다. 본 모델에서는 n_estimators=500일 때 가장 높은 성능을 보였고, 그 이상으로는 낮아지는 모습을 보였다. 따라서 모델은 n_estimators=500인 파라미터를 가진 모델로 선정하였다.

[Gaussian Mixture Model (GMM)]

1. Hyperparameter tuning (Feature Extraction (O))

1) n_components=6, covariance_type='tied' (Accuracy: 10.05%)

n_components를 6으로, covariance_type은 'tied'로 설정하여 GMM을 수행하였다. 결과는 10.05%로 매우 낮은 정확도를 보였다.

```
In [12]: from sklearn.mixture import GaussianMixture

In [33]: # gmm = GaussianMixture(n_components=6, covariance_type='tied', random_state=0).fit(x_train) # 군집
         gmm = GaussianMixture(n_components=6, covariance_type='tied').fit(x_train_FE) # 군집화 알고리즘 gmm

In [35]: gmm.fit(x_train_FE, y_train)
         pred = gmm.predict(x_test_FE)

In [36]: from sklearn.metrics import accuracy_score

In [37]: accuracy_score(pred,y_test)
Out[37]: 0.10049699189118494
```

2. Hyperparameter tuning (Feature Extraction (X))

1) n_components=6, covariance_type='tied' (Accuracy: 19.16%)

앞선 모델들과 마찬가지로 feature extraction을 하지 않은 데이터셋을 이용해 성능에 변화를 줘보고자 하였다. 그 결과 정확도가 feature extraction을 하지 않았을 때의 정확도 10.05%에서 19.16%로 거의 100%가 증가하였으나 모델의 성능 자체로는 너무나 좋지 않은 성능을 보여 결론적으로는 GMM 모델은 이 데이터셋을 분석하기에 적합하지 않은 것으로 판단하여 더 이상의 분석을 중단하게 되었다.

```
In [7]: from sklearn.mixture import GaussianMixture

In [8]: # gmm = GaussianMixture(n_components=6, covariance_type='tied', random_state=0).fit(x_train) # 군집
        gmm = GaussianMixture(n_components=6, covariance_type='tied').fit(x_train) # 군집화 알고리즘 gmm 만

In [9]: gmm.fit(x_train, y_train)
        pred = gmm.predict(x_test)

In [10]: from sklearn.metrics import accuracy_score

In [11]: accuracy_score(pred,y_test)
Out[11]: 0.19157729531781323
```

[Stochastic Gradient Descent (SGD)]

1. Hyperparameter tuning (Feature Extraction (O)) (Accuracy: 53.2%)

Stochastic Gradient Descent 모델로 multiclass를 분류하기 위해 ovo (one vs. one) 방식을 이용하였다. ovo 방식은 분류를 진행한 후 가장 많은 판별값을 얻은 클래스를 선택하는 방법이다. SGD 모델 생성 후 성능을 확인해보니 53.2%로 낮은 정확도를 보여 feature extraction을 하지 않은 데이터셋으로 모델을 학습시키기로 하였다.

```
In [100]: from sklearn.multiclass import OneVsOneClassifier

In [122]: ovo_sgd_1 = OneVsOneClassifier(SGDClassifier(max_iter=5))

In [123]: ovo_sgd_1.fit(x_train_FE, y_train)

C:\Users\JinsolKim\Anaconda3\lib\site-packages\sklearn\utils\validation.py:760: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
C:\Users\JinsolKim\Anaconda3\lib\site-packages\sklearn\linear_model\_stochastic_gradient.py:557: ConvergenceWarning: Maximum number of iteration reached before convergence. Consider increasing max_iter to improve the fit.
  ConvergenceWarning)
C:\Users\JinsolKim\Anaconda3\lib\site-packages\sklearn\linear_model\_stochastic_gradient.py:557: ConvergenceWarning: Maximum number of iteration reached before convergence. Consider increasing max_iter to improve the fit.
  ConvergenceWarning)
C:\Users\JinsolKim\Anaconda3\lib\site-packages\sklearn\linear_model\_stochastic_gradient.py:557: ConvergenceWarning: Maximum number of iteration reached before convergence. Consider increasing max_iter to improve the fit.
  ConvergenceWarning)
C:\Users\JinsolKim\Anaconda3\lib\site-packages\sklearn\linear_model\_stochastic_gradient.py:557: ConvergenceWarning: Maximum number of iteration reached before convergence. Consider increasing max_iter to improve the fit.
  ConvergenceWarning)

In [124]: ovo_pred_1 = ovo_sgd_1.predict(x_test_FE)

In [125]: accuracy = accuracy_score(y_test, ovo_pred_1)

          print(f'Mean accuracy score: {accuracy:.3}')

Mean accuracy score: 0.532
```

2. Hyperparameter tuning (Feature Extraction (O)) (Accuracy: 53.2%)

Feature extraction을 진행하지 않은 데이터셋을 이용해 SGD 모델을 학습하였다. 학습한 모델의 성능을 확인해보니 feature extraction을 진행했을 때보다는 약간 성능이 높아졌지만, 56.4%로 낮은 정확도를 보여 SGD도 본 데이터셋에 적합하지 않은 것으로 판단되었다.

In [11]:
```python
from sklearn.linear_model import SGDClassifier
from sklearn.multiclass import OneVsOneClassifier
```

In [12]:
```python
ovo_sgd_1 = OneVsOneClassifier(SGDClassifier(max_iter=5))
```

In [13]:
```python
ovo_sgd_1.fit(x_train, y_train)
```

```
C:₩Users₩JinsolKim₩Anaconda3₩lib₩site-packages₩sklearn₩utils₩validation.py:760: DataConversionWa
rning: A column-vector y was passed when a 1d array was expected. Please change the shape of y t
o (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
C:₩Users₩JinsolKim₩Anaconda3₩lib₩site-packages₩sklearn₩linear_model₩_stochastic_gradient.py:557:
ConvergenceWarning: Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  ConvergenceWarning)
C:₩Users₩JinsolKim₩Anaconda3₩lib₩site-packages₩sklearn₩linear_model₩_stochastic_gradient.py:557:
ConvergenceWarning: Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  ConvergenceWarning)
C:₩Users₩JinsolKim₩Anaconda3₩lib₩site-packages₩sklearn₩linear_model₩_stochastic_gradient.py:557:
ConvergenceWarning: Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  ConvergenceWarning)
C:₩Users₩JinsolKim₩Anaconda3₩lib₩site-packages₩sklearn₩linear_model₩_stochastic_gradient.py:557:
ConvergenceWarning: Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  ConvergenceWarning)
```

In [15]:
```python
ovo_pred_1 = ovo_sgd_1.predict(x_test)
```

In [17]:
```python
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_test, ovo_pred_1)

print(f'Mean accuracy score: {accuracy:.3}')
```

```
Mean accuracy score: 0.564
```

[Multi-Layer Perceptron (MLP)]

1. Hyperparameter tuning (Feature Extraction (O))

1) 2 hidden layers (dense 32, relu), output layer (dense 6, softmax) (Accuracy: 63.12%)

MLP 모델 구성을 위한 dense 층의 크기를 32로 시작하였다. 즉 입력 sample에 대해서 32 차원의 벡터를 출력하도록 하였다. 또한 마지막 층에 softmax 활성화 함수를 사용하여 각 입력 sample마다 46개의 출력 클래스에 대한 확률 분포를 출력하도록 하였다.

추가로 본 보고서에 표기를 하지는 않았지만 dropout을 넣을지 모델의 성능이 더 떨어지는 것을 확인했다.

In [26]:
```
from keras import models
from keras import layers

model = models.Sequential() # 비어 있는 모델
model.add(layers.Dense(32, activation = 'relu', input_shape=(2,)))
model.add(layers.Dense(32, activation = 'relu'))
model.add(layers.Dense(6, activation = 'softmax'))
```

```
Using TensorFlow backend.
```

In [27]:
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 32)                96
_________________________________________________________________
dense_2 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_3 (Dense)              (None, 6)                 198
=================================================================
Total params: 1,350
Trainable params: 1,350
Non-trainable params: 0
_________________________________________________________________
```

2) Model Compile

In [28]:
```
from keras import optimizers

model.compile(optimizer = 'rmsprop',
              loss = 'categorical_crossentropy',
              metrics = ['accuracy'])
```

```
In [36]: history = model.fit(x_train_FE, y_train_enc, epochs = 100)
```

```
Epoch 1/100
624422/624422 [==============================] - 124s 199us/step - loss: 0.8240 - accuracy: 0.64
17
Epoch 2/100
624422/624422 [==============================] - 108s 174us/step - loss: 0.8255 - accuracy: 0.64
11
Epoch 3/100
624422/624422 [==============================] - 110s 176us/step - loss: 0.8247 - accuracy: 0.64
16
Epoch 4/100
624422/624422 [==============================] - 111s 178us/step - loss: 0.8238 - accuracy: 0.64
23
Epoch 5/100
624422/624422 [==============================] - 110s 176us/step - loss: 0.8237 - accuracy: 0.64
22
Epoch 6/100
624422/624422 [==============================] - 110s 176us/step - loss: 0.8242 - accuracy: 0.64
20
Epoch 7/100
624422/624422 [==============================] - 108s 173us/step - loss: 0.8230 - accuracy: 0.64
```

```
In [37]: history_dict = history.history
```

```
In [38]: history_dict.keys()
```

```
Out[38]: dict_keys(['loss', 'accuracy'])
```

4. Model Testing

```
In [39]: results = model.evaluate(x_test_FE, y_test_enc)

         results
```

```
267610/267610 [==============================] - 22s 83us/step
```

```
Out[39]: [0.8854483913267875, 0.6312245726585388]
```

2. Hyperparameter tuning (Feature Extraction (X))

1) 2 hidden layers (dense 32, relu), output layer (dense 6, softmax)

'Memory Error'로 학습이 중단되었다.

In [16]:
```
from keras import models
from keras import layers

model = models.Sequential() # 비어 있는 모델
model.add(layers.Dense(32, activation = 'relu', input_shape=(6,)))
model.add(layers.Dense(32, activation = 'relu'))
model.add(layers.Dense(6, activation = 'softmax'))
```

```
Using TensorFlow backend.
```

In [17]:
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 32)                224
_________________________________________________________________
dense_2 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_3 (Dense)              (None, 6)                 198
=================================================================
Total params: 1,478
Trainable params: 1,478
Non-trainable params: 0
_________________________________________________________________
```

2) Model Compile

In [18]:
```
from keras import optimizers

model.compile(optimizer = 'rmsprop',
              loss = 'categorical_crossentropy',
              metrics = ['accuracy'])
```

In [19]:
```
history = model.fit(x_train, y_train_enc, epochs = 100)
```

```
~₩Anaconda3₩lib₩site-packages₩scipy₩sparse₩compressed.py in _major_index_fancy(self, idx)
    685             return self.__class__(new_shape)
    686
--> 687         row_nnz = np.diff(self.indptr)
    688         idx_dtype = self.indices.dtype
    689         res_indptr = np.zeros(M+1, dtype=idx_dtype)

<__array_function__ internals> in diff(*args, **kwargs)

~₩Anaconda3₩lib₩site-packages₩numpy₩lib₩function_base.py in diff(a, n, axis, prepend, append)
   1267     op = not_equal if a.dtype == np.bool_ else subtract
   1268     for _ in range(n):
-> 1269         a = op(a[slice1], a[slice2])
   1270
   1271     return a

MemoryError: Unable to allocate 2.38 MiB for an array with shape (624422,) and data type int32
```

Feature extraction을 하지 않았을 경우와 비교하기 위해 feature extraction을 하지 않은 데이터셋을 이용하여 모델을 학습시키고자 하였다. 하지만 아쉽게도 'Memory Error'로 학습 중간에 중단되었다. 앞선 random forest와 kNN 모델과 같이 feature extraction을 하지 않은 모델이 더욱 성능이 좋을 것으로 예상하였었다. 우선 feature extraction을 진행한 데이터셋으로 모델을 학습시킨 결과가 63.12%로 굉장히 낮은 성능을 보여 feature extraction을 하지 않은 데이터셋으로 모델이 학습이 안된다면 더 이상 MLP를 통해 모델의 성능을 개선시킬 수 있는 가능성이 보이지 않아 여기서 MLP 모델 학습을 마치게 되었다.

[Conclusions]

No.	ML/DL	Algorithm	Feature Extraction	Parameter Settings	Accuracy (%)
1	ML	k-Nearest Neighbors	O	k=5	61.28
2				k=7	62.00
3				k=10	62.59
4				k=100	64.74
5				k=500	(Memory Error)
6				k=1000	79.08
7			X	k=5	79.57
8				k=7	79.93
9				k=10	80.40
10				k=100	79.07
11				k=500	78.07
12				k=1000	80.40
13				**k=100, weights='distance'**	**80.56**
14				k=100, weights='distance', algorithm='kd_tree'	80.56
15				k=100, weights='distance', algorithm='ball_tree'	80.56
16				k=100, weights='distance', algorithm='brute	80.56
17				k=100, weights='distance', leaf_size=10	80.56
18				k=100, weights='distance', leaf_size=50	80.56
19		Random Forest	O	n_estimators=100	61.5
20			X	n_estimators=100	81.3
21				n_estimators=200	81.41
22				**n_estimators=500**	**81.49**
23				n_estimators=800	81.48

No.	ML/DL	Algorithm	Feature Extraction	Parameter Settings	Accuracy (%)
24				n_estimators=500, min_samples_leaf=2 (default), min_samples_split=2	81.47
25				n_estimators=500, min_samples_leaf=2 (default), min_samples_split=1 (default)	(Memory Error)
26		Gradient Mixture Model	O	n_components=6, covariance_type='tied'	10.05
27			X	n_components=6, covariance_type='tied'	19.16
28		Stochastic Gradient Descent	O	(default)	53.2
29			X	(default)	56.4
30	DL	Multi-Layer Perceptron	O	2 hidden layers (dense 32, relu), output layer (dense 6, softmax)	63.12
31			X	2 hidden layers (dense 32, relu), output layer (dense 6, softmax) (Accuracy: 63.12%)	(Memory Error)

(주) Parameter settings 중 특별히 표시하지 않은 parameter들은 default 값

Activity 데이터셋을 이용하여 6가지 클래스의 activity를 분류하는 프로젝트를 진행하였다. 이를 위해 머신러닝 알고리즘 중 k-Nearest Neighbors, Random Forest, Gradient Mixture Model, Stochastic Gradient Descent을 사용하였고, 딥러닝 알고리즘으로 Multi-Layer Perceptron을 사용하였다. 각 모델을 처음 생성할 때는 default로 설정되어 있는 파라미터를 기준으로 생성한 뒤 학습된 모델의 성능과 데이터셋과의 적합성을 확인하여 파라미터를 점차 변화시켜 본 데이터셋에 최적의 성능을 보이는 모델을 선정하였다.

그 결과 n_estimators=500의 random forest 모델과 k=100, weights='distance'인 파라미터를 가진 k-NN 모델이 각각 1순위, 2순위로 가장 높은 성능을 보여 이와 같은 random forest 모델을 본 프로젝트의 최종 모델로 선정하였다. 또한 k-NN 모델을 추가로 제안한다. k-NN 모델 중 가장 높은 정확도인 80.56%를 기록한 모델이 6가지 있었지만, 실험을 진행한 모델 중 가장 심플한, k=100, weights='distance'의 파라미터를 가진 모델을 추가로 제안한다. 최종 선정한 random

forest 모델은 81.49%, 추가로 제안하는 k-NN 모델은 80.56% 성능의 결과를 보였다. 물론 이는 검증데이터로 나타난 결과이므로 프로젝트의 test 데이터로는 어떠한 성능을 나타낼지 기대가 된다.

본 프로젝트를 진행하면서 대용량의 데이터셋을 이용해 분류 알고리즘을 만들어 분석하고 분류하는 실질적인 데이터분석/머신러닝,딥러닝 분석 할 수 있어서 매우 의미 깊었다. 특히 파라미터를 변경해가면서 성능이 개선되고, 성능이 낮아지면 어떠한 부분이 잘 못 되었는지 찾아가는 과정이 매우 재미있었다. 머신러닝과 딥러닝 작업에 매우 큰 흥미를 갖게 되고, 이론으로만 배우던 알고리즘들을 직접 만들어 구현할 수 있다는 자신감이 생기게 된 점이 개인적으로 가장 뿌듯하다. 물론 중간중간에 training이 다 되길 오랫동안 기다리다 메모리에러로 인해 학습이 중단되면 너무 아쉽고, 그 이상의 더 무거운 파라미터 값을 넣어 확인하지 못한다는 점이 아쉬었다. 특히 MLP 모델에 여러 파라미터 변경을 통해 많은 실험을 하여 높은 성능을 찾기를 제일 기대했었는데 학습이 되지 않아 속상했다. 하지만, 현재 시스템 상황과 데이터셋 그리고 알고리즘의 파라미터들 간에 최적화를 찾아가는 과정이 즐거웠다.