

◆ 분석 개요 =====

1. 사용 언어 : Python
2. 분석 환경 : [CPU] Apple M2, [RAM] 8GB, [GPU] Colab T4

3. 분석 데이터 정보

- ① 수집 기관 : Ford 자동차 제조사
- ② 데이터 내용 : 엔진 상태와 관련 있는 센서 500개의 측정 값
- ③ 데이터 파일 : Train Dataset 1file, Test Dataset 1file

4. 분석 데이터 구조

- ① Features : 500 columns (Sensor values)
- ② Samples : 4921 raw (Train 3601, Test 1320)
- ③ Labels : 1 or 0 (Normal or Abnormal)

5. 학습 및 예측 목표

- ① 500개 각 센서 값을 학습하여 그 상태가 Normal(1)인지 Abnormal(0)인지 추측
- ② Train 3601개의 샘플을 학습 후 Test 1320개 샘플의 상태를 추측

6. 데이터 학습에 사용된 모델

- ① LogisticRegression
- ② TabularPredictor (Auto ML)
- ③ XGBClassifier
- ④ LGBMClassifier
- ⑤ CatBoostClassifier
- ⑥ RandomForestClassifier
- ⑦ **CNN (Convolutional Neural Network)**

7. 데이터 학습 및 예측 결론

- ① 학습이 가장 잘 되는 모델 : CNN (Convolutional Neural Network)
(정확도 97.2 ~ 97.5%, loss : 0.08 ~ 0.09)
- ② CNN 모델은 주로 이미지나 지역적 패턴이 있는 데이터를 학습 및 예측할 때 사용되는 모델로, 분석 대상인 데이터셋도 인접 하는 센서 값과 함께 패턴을 이루는 특성이 있기 때문에 CNN 모델에서 효과적으로 학습 되는 것으로 보임

◆ 데이터 분석 =====

1. 데이터를 불러와서 DataFrame 형식으로 변환

```
# 드라이브에서 train, test 데이터셋 불러오기

# arff.loadarff() 함수는 두개의 값을 반환. (데이터 numpy array, meta 데이터 객체)
data_train, meta_train = arff.loadarff('./FordA/FordA_TRAIN.arff')
data_test, meta_test = arff.loadarff('./FordA/FordA_TEST.arff')

# 데이터를 DataFrame 형식으로 변환
train_df = pd.DataFrame(data_train)
test_df = pd.DataFrame(data_test)
```

✓ 0.8s

MagicPython

2. train, test 데이터셋 확인

```
# train_df 구조 확인 (3601 rows x 501 columns)
train_df
```

✓ 0.0s

MagicPython

	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10
0	-0.797172	-0.664392	-0.373015	0.040815	0.526936	0.984288	1.353120	1.578108	1.659251	1.640809
1	0.804855	0.634629	0.373474	0.038343	-0.340988	-0.740860	-1.109667	-1.395357	-1.570192	-1.619951
2	0.727985	0.111284	-0.499124	-1.068629	-1.578351	-1.990534	-2.302031	-2.503403	-2.585211	-2.550600
3	-0.234439	-0.502157	-0.732488	-0.946128	-1.139739	-1.323336	-1.490243	-1.607077	-1.620430	-1.506933
4	-0.171328	-0.062285	0.235829	0.710396	1.239969	1.649823	1.876321	1.865535	1.703751	1.466467
...
3596	0.196022	-0.070102	-0.336226	-0.516799	-0.555282	-0.442793	-0.221369	0.025217	0.233320	0.350545
3597	0.041994	0.422255	0.740529	0.975426	1.109891	1.137270	1.058349	0.894955	0.671224	0.421544
3598	-0.570054	-0.333165	-0.293519	-0.425344	-0.590869	-0.615648	-0.348033	0.275412	1.153586	2.035725
3599	2.006732	2.079150	2.022036	1.867560	1.648112	1.379446	1.093717	0.784057	0.434150	0.029284
3600	-0.125241	-0.325363	-0.488237	-0.599045	-0.651111	-0.647106	-0.595040	-0.502922	-0.388109	-0.259946

3601 rows x 501 columns

```
# test_df 구조 확인 (1320 rows x 501 columns)
test_df
```

✓ 0.1s

MagicPython

	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10
0	-0.140402	0.171641	0.302044	0.232804	0.033853	-0.224183	-0.469987	-0.645396	-0.617700	-0.367280
1	0.334038	0.322253	0.453844	0.671852	0.887897	1.020469	1.059750	1.030290	0.950746	0.858436
2	0.716686	0.744367	0.725913	0.661325	0.555217	0.413585	0.246580	0.065273	-0.121109	-0.301032
3	1.240282	1.331189	1.386596	1.383220	1.305979	1.142784	0.878613	0.532291	0.140025	-0.258262
4	-1.159478	-1.204174	-1.167605	-1.033518	-0.818166	-0.558119	-0.299291	-0.093691	0.022770	0.044337
...
1315	0.143630	-0.135823	-0.510278	-0.850804	-1.058080	-1.082756	-0.961845	-0.748399	-0.575669	-0.569500
1316	-0.165568	-0.504614	-0.780065	-0.937044	-0.950518	-0.854054	-0.701736	-0.544270	-0.424473	-0.357913
1317	0.710084	0.593979	0.381886	0.127285	-0.112304	-0.274140	-0.312698	-0.195008	0.063567	0.398281
1318	0.006847	-0.140624	-0.270594	-0.378835	-0.461983	-0.515125	-0.538119	-0.532769	-0.495602	-0.436697
1319	-0.541355	-0.241723	0.100741	0.468953	0.830632	1.146251	1.392024	1.555571	1.642761	1.666174

1320 rows x 501 columns

3. train, test 데이터셋 info()

```
# train, test 데이터셋 정보
print(train_df.info())
print(test_df.info())
✓ 0.0s
```

MagicPython

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3601 entries, 0 to 3600
Columns: 501 entries, att1 to target
dtypes: float64(500), object(1)
memory usage: 13.8+ MB
None
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1320 entries, 0 to 1319
Columns: 501 entries, att1 to target
dtypes: float64(500), object(1)
memory usage: 5.0+ MB
None
```

- train_df : 3601 samples, 501 columns (Include target 1)
- test_df : 1320 samples, 501 columns (Include target 1)
- 500 feature type : float64, target type : object

4. Train, test 데이터셋 요약 (describe)

```
# train_df의 features(각 센서 값) 요약
train_df.describe()
✓ 0.3s
```

MagicPython MagicPython

	s1	s2	s3	s4	s5	s6	s7	s8	
count	3601.000000	3601.000000	3601.000000	3601.000000	3601.000000	3601.000000	3601.000000	3601.000000	3601.000000
mean	-0.016708	-0.015270	-0.013605	-0.011943	-0.009966	-0.007062	-0.003495	0.000568	
std	1.058455	1.051904	1.044560	1.042573	1.046933	1.052829	1.055439	1.054305	
min	-3.933454	-3.656912	-3.479467	-3.595350	-3.773891	-3.914729	-3.855301	-4.497360	
25%	-0.685693	-0.699526	-0.688302	-0.709732	-0.694991	-0.691035	-0.690316	-0.701244	
50%	-0.007573	-0.003044	-0.003066	0.008598	0.001042	0.000280	-0.011988	-0.026672	
75%	0.660360	0.674898	0.683698	0.679474	0.697530	0.688802	0.721442	0.705021	
max	3.503936	3.369278	3.498286	3.493830	3.293318	3.605585	3.895870	3.741035	

8 rows × 501 columns

```
# test_df의 features(각 센서 값) 요약
test_df.describe()
✓ 0.3s
```

MagicPython

	s1	s2	s3	s4	s5	s6	s7	s8	
count	1320.000000	1320.000000	1320.000000	1320.000000	1320.000000	1320.000000	1320.000000	1320.000000	1320.000000
mean	0.056002	0.049911	0.038884	0.024069	0.007442	-0.008919	-0.023495	-0.033577	-0.033577
std	1.030455	1.027540	1.031559	1.037225	1.040995	1.041183	1.039772	1.037450	1.037450
min	-3.295308	-3.114238	-3.143402	-3.210066	-3.488749	-3.297407	-3.530643	-3.614145	-3.614145
25%	-0.615181	-0.622534	-0.647919	-0.703829	-0.710411	-0.704157	-0.701900	-0.726534	-0.726534
50%	0.060755	0.063215	0.072926	0.046643	-0.001561	-0.033439	-0.069241	-0.038386	-0.038386
75%	0.725193	0.683444	0.708510	0.721956	0.711720	0.667813	0.662852	0.670717	0.670717
max	3.972283	3.854763	3.500758	3.425150	3.603918	3.498317	3.202631	3.149192	3.149192

8 rows × 501 columns

5. Target value 1과 0으로 변환

```
# target 값은 [b'-1', b'1'] byte 형식으로 인코딩 되어 있음
print(train_df.target.unique())

# target 값을 일반 문자열로 디코딩 한 후, int 형으로 변환
train_df['target'] = train_df['target'].apply(lambda x: int(x.decode()))
test_df['target'] = test_df['target'].apply(lambda x: int(x.decode()))

# target 값 -1을 0으로 변환
train_df['target'] = train_df['target'].replace(-1, 0)
test_df['target'] = test_df['target'].replace(-1, 0)

print(train_df.target.unique()) # 변환 후 target 값 출력
```

✓ 0.0s MagicPython

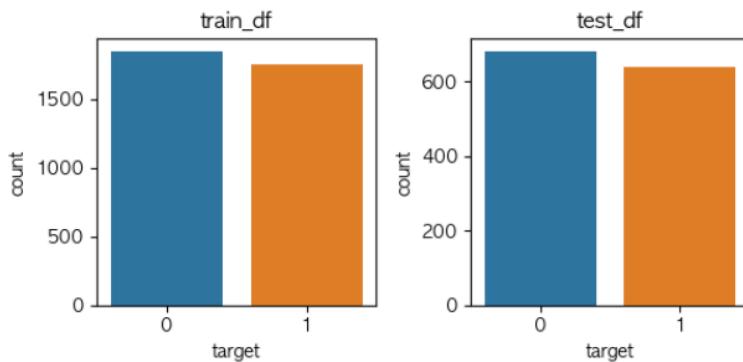
```
[b'-1' b'1']
[0 1]
```

6. Target value 빈도의 균형이 맞는지 확인

```
# 데이터셋의 target값 비중 불균형이 있는지 확인
#   -> sns.countplot: 범주형 데이터의 분포, sns.histplot: 연속형 데이터의 분포
fig, axs = plt.subplots(1,2, figsize=(6,3))
sns.countplot(data=train_df, x='target', ax=axs[0])
sns.countplot(data=test_df, x='target', ax=axs[1])
axs[0].set_title('train_df')
axs[1].set_title('test_df')
fig.tight_layout()
plt.show()

print('train_df_0_class : ', len(train_df.loc[train_df.target==0]))
print('train_df_1_class : ', len(train_df.loc[train_df.target==1]))
print('test_df_0_class : ', len(test_df.loc[test_df.target==0]))
print('test_df_1_class : ', len(test_df.loc[test_df.target==1]))
```

✓ 0.2s MagicPython



```
train_df_0_class : 1846
train_df_1_class : 1755
test_df_0_class : 681
test_df_1_class : 639
```

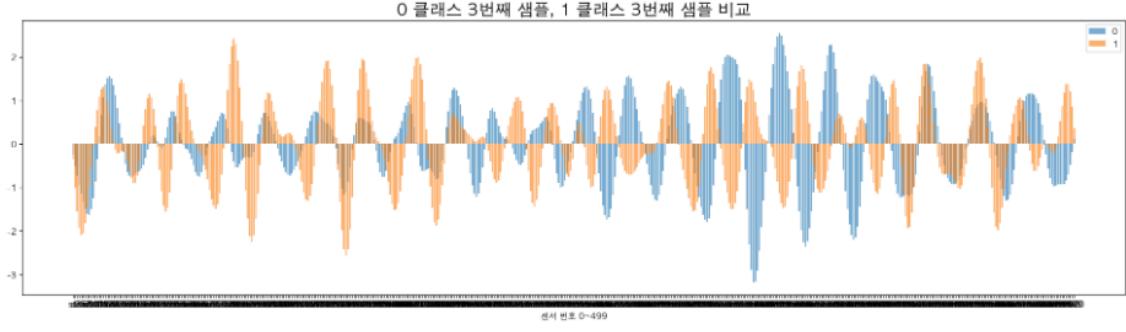
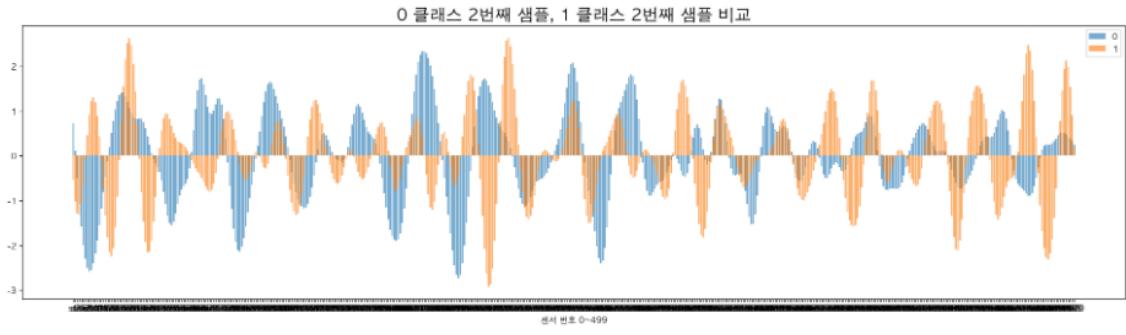
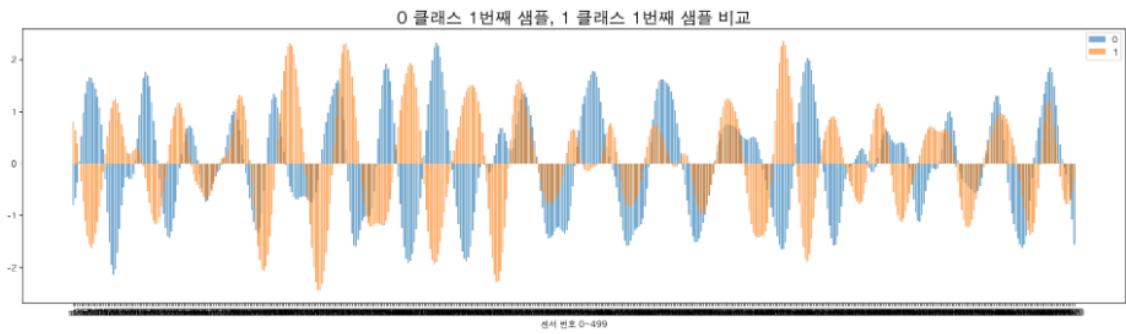
➤ train, test dataset 의 target value 빈도는 균형이 맞다.

7. 0과 1 클래스 각 샘플의 센서 값 분포 비교 (각 3개 샘플 비교)

```
# 클래스가 0, 1인 인덱스 추출
idx_0 = train_df.loc[train_df.target==0].index
idx_1 = train_df.loc[train_df.target==1].index

# 가로: 센서 번호, 세로: 센서 값
for i in range(3): # 비교하고 싶은 개수 설정 (3개 비교)
    plt.figure(figsize=(24,6))
    plt.bar(train_df.columns[:-1], train_df.iloc[idx_0[i], :-1], label='0', alpha=0.6)
    plt.bar(train_df.columns[:-1], train_df.iloc[idx_1[i], :-1], label='1', alpha=0.6)
    plt.title(f'0 클래스 {i+1}번째 샘플, 1 클래스 {i+1}번째 샘플 비교', fontsize=20)
    plt.xlabel('센서 번호 0~499')
    plt.legend()
    plt.show()
```

3.9s MagicPython



- 클래스별 센서 값에 차이가 있음 -> 학습 가능할 것으로 보임
- 전체 샘플의 센서 별 평균도 볼 필요 있음 (클래스 구분)

8. 센서 별 클래스 평균 값 분포

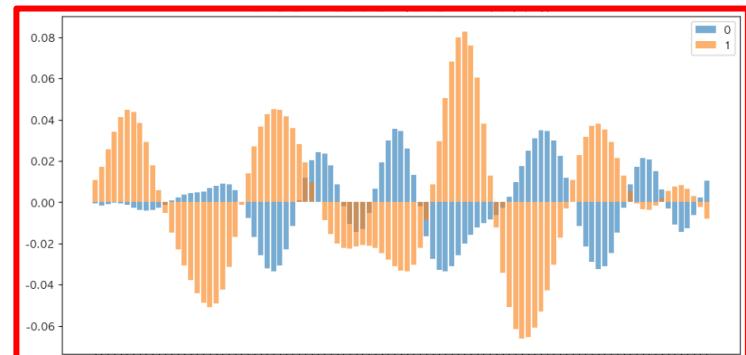
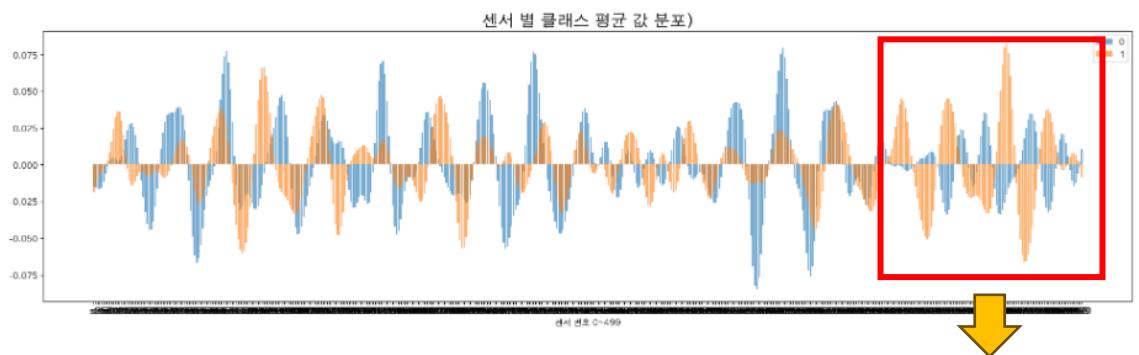
```
# 각 센서의 클래스별(0,1) 평균 값  
s_means = train_df.groupby(by='target').mean()  
✓ 0.0s
```

MagicPython

```
# 가로: 센서 번호, 세로: 센서 별 클래스 평균 값  
# -> 센서 별로 2개의 클래스(0, 1)  
plt.figure(figsize=(20,6))  
plt.bar(s_means.columns, s_means.iloc[0,:], alpha=0.6, label='0')  
plt.bar(s_means.columns, s_means.iloc[1,:], alpha=0.6, label='1')  
plt.title('센서 별 클래스 평균 값 분포', fontsize=20)  
plt.xlabel('센서 번호 0~499')  
plt.legend()  
  
plt.show()
```

✓ 1.2s

MagicPython



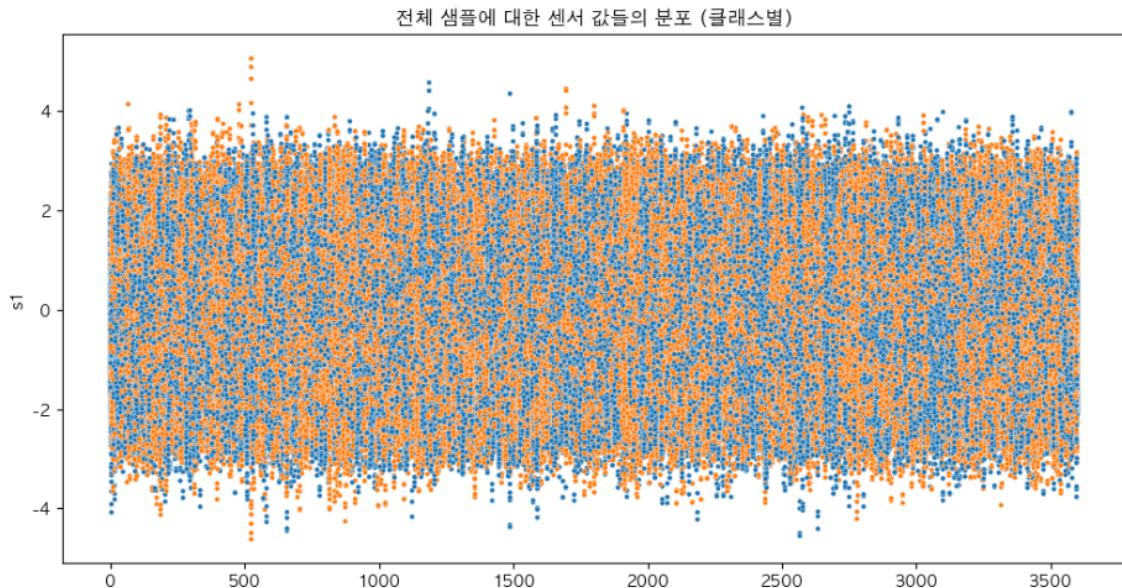
- 센서 후반부 (s403 ~) 클래스에 따라 센서 값이 양수와 음수로 나눠짐 -> 후반부 센서만 학습시켜 볼 필요 있음

9. 전체 샘플에 대한 센서 값들의 분포 (클래스별)

```
# 가로: 샘플, 세로: 모든 센서들의 값 (클래스별)

plt.figure(figsize=(12,6))
plt.title('전체 샘플에 대한 센서 값들의 분포 (클래스별)')
for i in range(train_df.shape[1]-1): #target열 제외 모든 센서값 인덱싱
    sns.scatterplot(data=train_df, x=train_df.index, y=train_df.iloc[:,i], s=10,
                     hue='target', legend=False)
```

✓ 19.7s MagicPython



- 각 샘플 값이 시계열 데이터일 경우 시간의 흐름에 따른 패턴은 보이지 않음
- 클래스별 평균 값으로 다시 시각화 해볼 필요 있음 (다음 장)

10. 각 샘플 별 500개 센서 값의 평균

```
# 가로: 샘플 번호, 세로: 500개 센서 값 평균

# 클래스별 데이터프레임 생성
train_0 = train_df.loc[train_df.target == 0]
train_1 = train_df.loc[train_df.target == 1]

# 샘플별 500개 센서 값의 평균 (bar)
plt.figure(figsize=(12,4))
plt.bar(train_0.index, train_0.iloc[:, :-1].mean(axis=1), label='0')
plt.bar(train_1.index, train_1.iloc[:, :-1].mean(axis=1), label='1')
plt.title('샘플별 500개 센서 값의 평균 (bar)')
plt.xlabel('샘플 번호')

# 샘플별 500개 센서 값의 평균 (scatter)
plt.figure(figsize=(12,4))
sns.scatterplot(x=train_df.index, y=train_df.iloc[:, :-1].mean(axis=1),
                 hue=train_df.target)
plt.title('샘플별 500개 센서 값의 평균 (scatter)')
plt.xlabel('샘플 번호')

# 클래스별 센서 평균 값의 분포 (stripplot)
plt.figure(figsize=(12,4))
sns.stripplot(data=train_df, x=train_df.target, y=train_df.iloc[:, :-1].mean(axis=1),
               jitter=True, marker='o', alpha=0.5)
plt.title('클래스별 센서 평균 값의 분포 (stripplot)')
plt.xlabel('target')

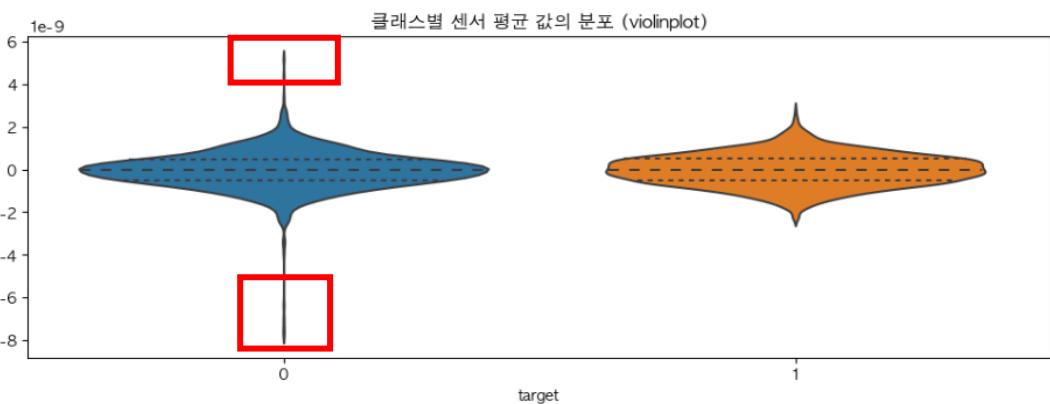
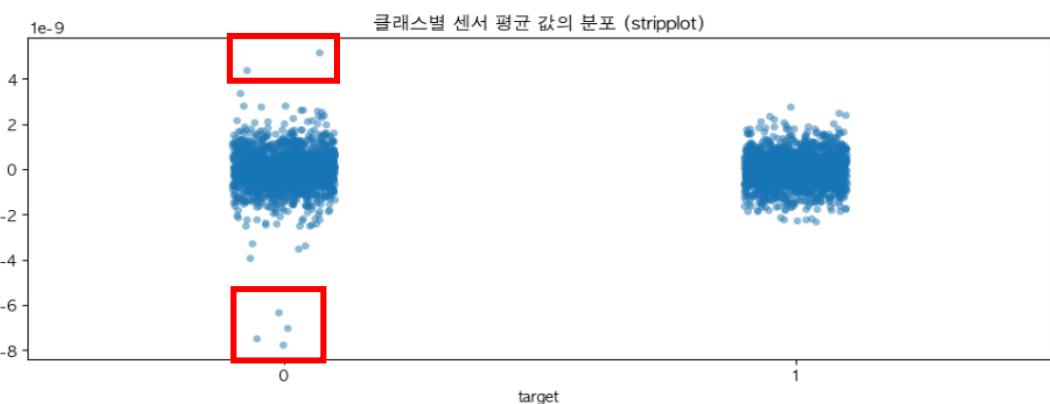
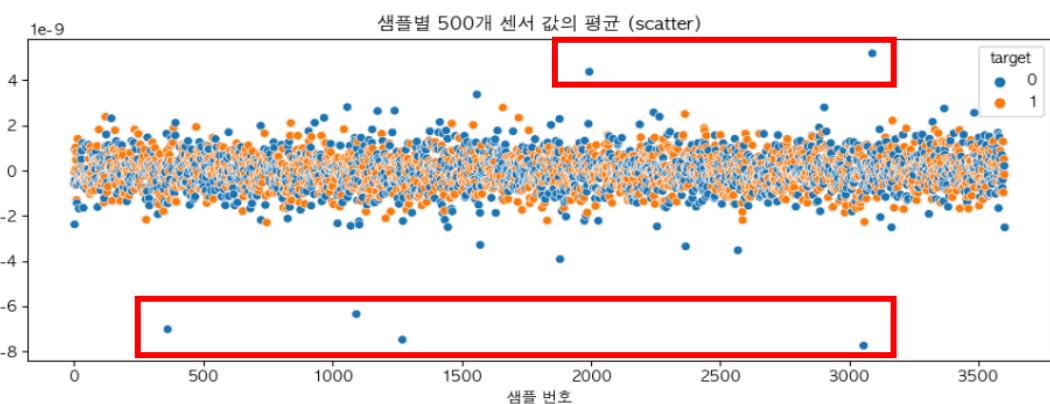
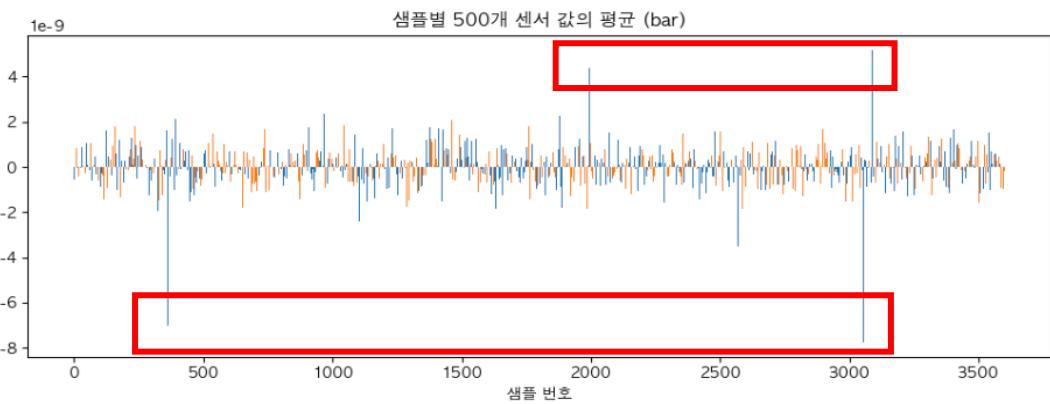
# 클래스별 센서 평균 값의 분포 (violinplot)
plt.figure(figsize=(12,4))
sns.violinplot(x=train_df.target, y=train_df.iloc[:, :-1].mean(axis=1), inner="quart")
plt.title('클래스별 센서 평균 값의 분포 (violinplot)')
plt.xlabel('target')

plt.show()
```

✓ 2.1s

MagicPython

➤ 출력 결과 다음 페이지 참조

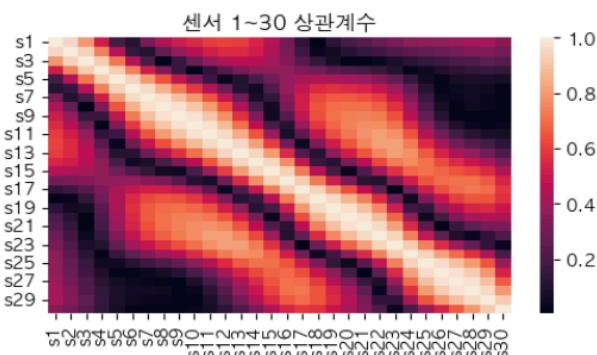
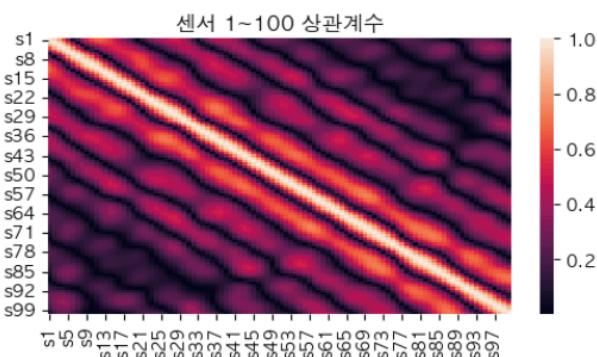
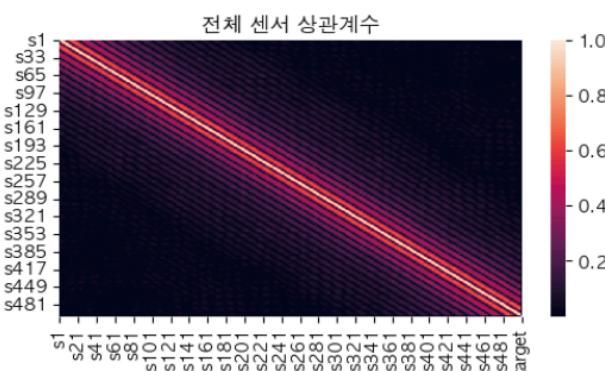


- 0 클래스에 노이즈 샘플이 있음.
- 노이즈 샘플 제거 후 학습 해볼 필요 있음.

11. 센서 별 상관관계 분석

```
# target과 각 센서 값의 상관관계 분석  
# 전체 센서 상관계수  
plt.figure(figsize=(6,3))  
plt.title('전체 센서 상관계수')  
train_corr = abs(train_df.corr())  
sns.heatmap(train_corr)  
  
# 센서 1~100 상관계수  
plt.figure(figsize=(6,3))  
plt.title('센서 1~100 상관계수')  
corr_100 = abs(train_df.iloc[:100, :100].corr())  
sns.heatmap(corr_100)  
  
# 센서 1~30 상관계수  
plt.figure(figsize=(6,3))  
plt.title('센서 1~30 상관계수')  
corr_30 = abs(train_df.iloc[:30, :30].corr())  
sns.heatmap(corr_30, xticklabels=1)
```

MagicPython



인접한 센서와 상관계수가 높음
→ CNN으로 학습 가능 할
것으로 보임
→ kernel_size를 3으로하면
학습이 잘될 것으로 예상

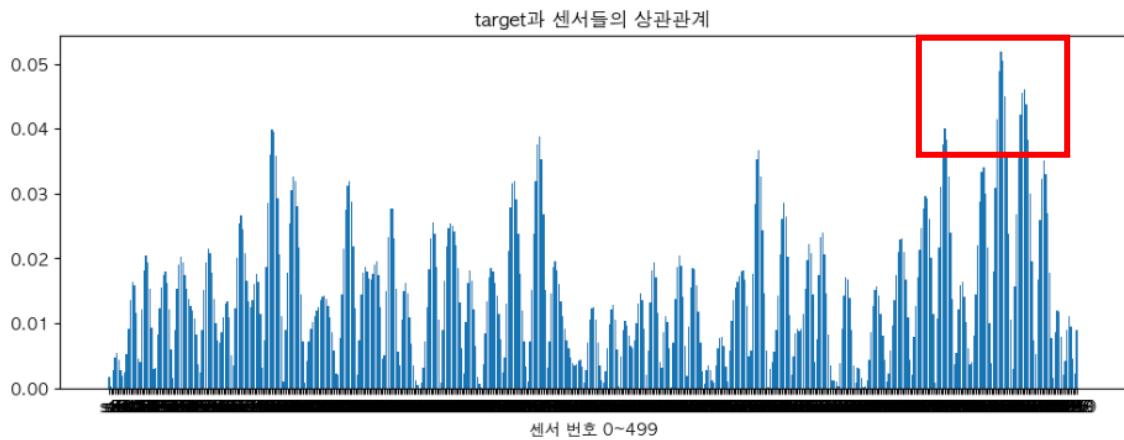
12. Target에 영향을 주는 센서 확인

```
# 센서별 target과 상관계수 분포 (target에 영향을 주는 센서 확인)
#   -> 가로: 센서, 세로: target과 상관계수 값
plt.figure(figsize=(12,4))
plt.title('센서별 target과 상관계수 분포')
plt.xlabel('센서 번호 0~499')
plt.bar(train_df.columns[:-1], train_corr.target[:-1])

fig.tight_layout()
plt.show()

# target과 센서간의 상관계수를 내림차순 정리
#   -> target에 영향을 주는 센서 확인
print('target과 상관관계 상위 센서')
print(train_corr.target.sort_values(ascending=False).head(11).drop('target',axis=0))
```

MagicPython



```
target과 상관관계 상위 센서
s461 0.051789
s462 0.050375
s460 0.048899
s473 0.045961
s472 0.045462
s463 0.044951
s474 0.043726
s471 0.042066
s459 0.041494
s432 0.040031
Name: target, dtype: float64
```

- 앞서 확인 한대로 후반부(s403 ~)에 target에 영향을 주는 센서가 있음.
- Target과 상관계수가 높은 센서들만 학습해볼 필요 있음.

◆ 학습 및 결과 예측 =====

1. Auto ML(AutoGluon)을 사용하여 최적의 머신러닝 모델 확인하기

```
# Auto ML로 정확도 높은 ML 모델 확인하기
from autogluon.tabular import TabularPredictor

# AutoGluon ML로 학습 및 예측, 정확도 출력
#   -> TabularPredictor.fit() 함수는 자동으로 훈련 데이터를 훈련 데이터와 검증 데이터로 분할(기본: 20%)
predictor = TabularPredictor(label='target').fit(train_df)
y_pred = predictor.predict(test_df)
# 정확도 계산, True는 1, False는 0이므로 mean()값은 예측이 맞은 비율
accuracy = (test_df.target == y_pred).mean()
print(f"Best model을 test 데이터셋으로 평가했을 때 정확도 : {accuracy}")

✓ 1m 30.s
```

MagicPython

[학습 및 예측 결과]

```
AutoGluon training complete, total runtime = 90.28s ... Best model: "WeightedEnsemble_L2"
TabularPredictor saved. To load, use: predictor = TabularPredictor.load("AutogluonModels/ag-20230816_071132/")

Best model 을 test 데이터셋으로 평가했을 때 정확도 : 0.8469696969696969
```

[Leaderboard 확인]

```
leaderboard = predictor.leaderboard(test_df)
leaderboard
```

✓ 0.5s

MagicPython

	model	score_test	score_val	pred_time_test	pred_time_val	fit_time	pred_time_test_marginal	pre
0	WeightedEnsemble_L2	0.846970	0.878	0.029192	0.017365	26.427780	0.001605	
1	CatBoost	0.833333	0.844	0.014374	0.010121	23.466851	0.014374	
2	LightGBMXT	0.823485	0.846	0.020239	0.012881	5.647985	0.020239	
3	NeuralNetTorch	0.815152	0.852	0.013213	0.005081	2.685123	0.013213	
4	XGBoost	0.805303	0.832	0.039410	0.017883	16.978330	0.039410	
5	NeuralNetFastAI	0.798485	0.822	0.022106	0.012711	2.083403	0.022106	
6	LightGBMLarge	0.788636	0.798	0.028383	0.015502	21.067012	0.028383	
7	LightGBM	0.788636	0.820	0.038047	0.019144	9.474849	0.038047	
8	ExtraTreesEntr	0.767424	0.788	0.043199	0.024052	0.511025	0.043199	
9	RandomForestEntr	0.749242	0.766	0.052784	0.022886	2.842844	0.052784	
10	ExtraTreesGini	0.747727	0.748	0.054983	0.023285	0.463099	0.054983	
11	RandomForestGini	0.735606	0.772	0.052099	0.024083	2.202469	0.052099	
12	KNeighborsDist	0.718939	0.718	0.049868	0.023038	0.051463	0.049868	
13	KNeighborsUnif	0.718939	0.718	0.067474	0.085666	1.338704	0.067474	

➤ 14 개 학습 모델 중 WeightedEnsemble_L2 모델의 정확도가 가장 높음 (84.7%)

➤ train, test 데이터셋 라벨 분리 (x, y) – 개별 ML 학습을 위해 (다음 페이지)

```
# AutoML에서는 라벨열을 지정해서 라벨을 인식했지만, 개별 ML에서는 라벨을 분리해서 학습
x_train, y_train = train_df.iloc[:, :-1], train_df.iloc[:, -1]
x_test, y_test = test_df.iloc[:, :-1], test_df.iloc[:, -1]

print(x_train.shape, y_train.shape, x_test.shape, y_test.shape)
```

✓ 0.0s

MagicPython

(3601, 500) (3601,) (1320, 500) (1320,)

2. 머신러닝 4 가지 모델로 학습 및 예측

```
# ML 모델별로 학습 및 예측 결과

# 학습 및 예측 모델 4가지 리스트
model = [xgb.XGBClassifier(),
          LGBMClassifier(),
          CatBoostClassifier(verbose=0),
          RandomForestClassifier()]

# 모델명 리스트 (출력시 모델명 텍스트로 사용)
model_list = []
for i in range(len(model)):
    model_list.append(type(model[i]).__name__)

# 캔버스 4개 만들기 (혼동행렬 / ROC / Feat.Imp. 상위 10항목 / Feat.Imp. 전체 분포)
fig, axs = plt.subplots(1, len(model), figsize=(3*len(model), 2.5))
fig2, axs2 = plt.subplots(1, len(model), figsize=(3*len(model), 2.5))
fig3, axs3 = plt.subplots(1, len(model), figsize=(3*len(model), 2.5))
fig4, axs4 = plt.subplots(1, len(model), figsize=(3*len(model), 2.5))
# 캔버스 4개의 title
fig.suptitle('각 모델의 혼동 행렬', fontsize=16)
fig2.suptitle('각 모델의 ROC Curve', fontsize=16)
fig3.suptitle('각 모델의 Feature importances 상위10 센서', fontsize=16)
fig4.suptitle('각 모델의 Feature importances 센서별 분포', fontsize=16)

# 4개 ML 모델 학습 후 정확도, 혼동행렬, ROC, Feature Importance 출력하기
for i in range(len(model)): # 모델 리스트 크기만큼 반복
    model[i].fit(x_train, y_train) # 모델 한개씩 불러와서 학습
    y_pred = model[i].predict(x_test) # 각 모델 예측값 변수에 할당
    accuracy = accuracy_score(y_test, y_pred) # 각 모델 정확도 변수에 할당
    # 기본 threshold에서 정확도 구하기
    print(f'threshold 0.5 -> {model_list[i]}_acc : {round(accuracy*100,2)}%')

# 각 모델의 혼동 행렬을 히트맵으로 그리기
conf_mat = confusion_matrix(y_test, y_pred) # 혼동 행렬 만들기 (2,2)
sns.heatmap(conf_mat, annot=True, fmt='d', cmap='Blues', ax=axs[i])
axs[i].set_title(model_list[i])
axs[i].set_ylabel('Actual')
axs[i].set_xlabel('Predicted')

# ROC 그래프 그리기
probs = model[i].predict_proba(x_test)[:, 1] # 양성 클래스에 대한 예측 확률
fpr, tpr, thresholds = roc_curve(y_test, probs) # fpr, tp, thresholds 리턴
roc_auc = auc(fpr, tpr) # auc 계산 함수 호출
axs2[i].plot(fpr, tpr, color='darkorange',
             lw=2, label=f'AUC = {roc_auc:.2f}') # fpr, tpr로 ROC 그래프 그리기
axs2[i].plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
axs2[i].set_title(model_list[i])
axs2[i].set_xlabel('FPR')
axs2[i].set_ylabel('TPR')
axs2[i].legend(loc="lower right")
```

```

# 최적의 threshold 값 찾기
optimal_idx = np.argmax(tpr - fpr) # tpr-fpr 차이가 가장 큰 인덱스 찾기
optimal_threshold = thresholds[optimal_idx] # 찾은 인덱스에 해당하는 threshold

# optimal threshold 값을 사용하여 예측
y_pred_optimal = np.where(probs > optimal_threshold, 1, 0)

# 새로운 예측 값을 사용하여 정확도 계산
accuracy_optimal = accuracy_score(y_test, y_pred_optimal)
print(f'threshold {optimal_threshold} -> \
      {model_list[i]}_acc : {round(accuracy_optimal*100, 2)}%')

# Feature importances 상위10 항목 구하기
imp_df = pd.DataFrame(model[i].feature_importances_)
imp_df.rename(columns={0:'imp'}, inplace=True) # 열 이름 변경
imp_df.index = 's' + imp_df.index.astype(str) # 인덱스명 변경 ('s' 추가하여 str으로)
imp_df_sort = imp_df.sort_values(by='imp', ascending=False) # 내림차순 정렬

# Feature importances 상위10 항목 내림차순 bar 그래프 그리기
axs3[i].bar(imp_df_sort.head(10).index, imp_df_sort.head(10).iloc[:, 0])
axs3[i].set_title(model_list[i])
axs3[i].tick_params(axis='x', rotation=90)
axs3[i].set_xlabel('Sensor_no.')
axs3[i].set_ylabel('Importances')

# Feature importances 전체 센서별 분포
axs4[i].bar(imp_df.index, imp_df.imp.values)
axs4[i].set_title(model_list[i])
axs4[i].set_xlabel('Sensor_no.')
axs4[i].set_ylabel('Importances')

fig.tight_layout()
fig2.tight_layout()
fig3.tight_layout()
fig4.tight_layout()

plt.show()

```

✓ 33.4s

```

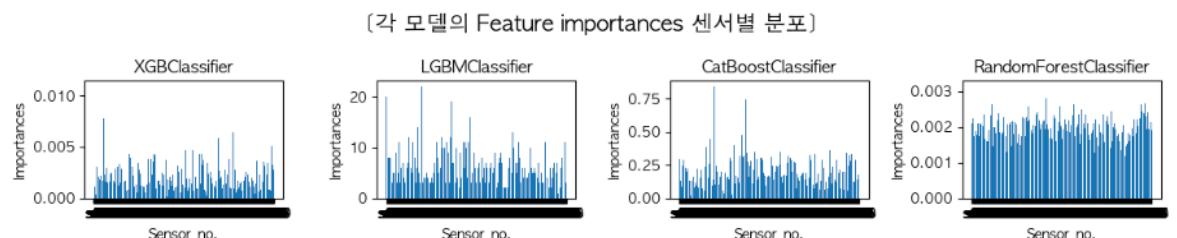
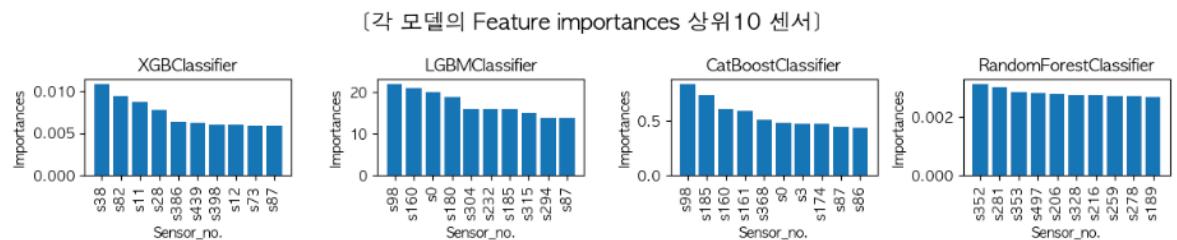
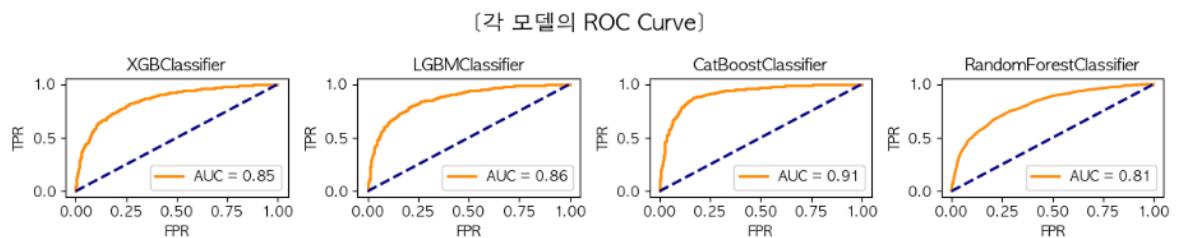
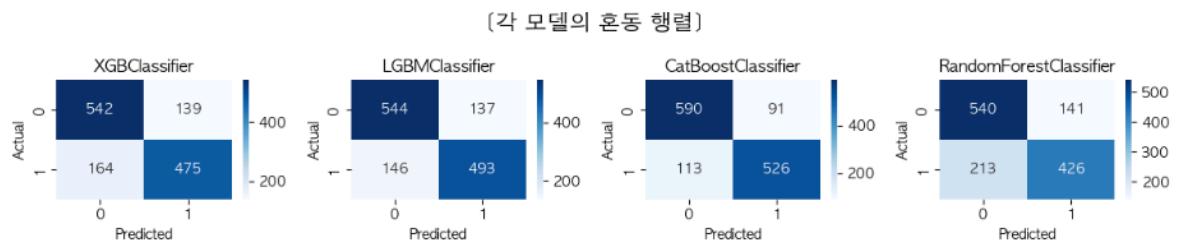
threshold 0.5 -> XGBClassifier_acc : 77.05%
threshold 0.4227582812309265 -> XGBClassifier_acc : 77.42%
threshold 0.5 -> LGBMClassifier_acc : 78.56%
threshold 0.4867063949630347 -> LGBMClassifier_acc : 79.09%
threshold 0.5 -> CatBoostClassifier_acc : 84.55% [Red Box]
threshold 0.4621984395664772 -> CatBoostClassifier_acc : 85.3%
threshold 0.5 -> RandomForestClassifier_acc : 73.94%
threshold 0.5 -> RandomForestClassifier_acc : 73.94%

```

MagicPython

➤ CatBoost 모델의 성능이 가장 좋음

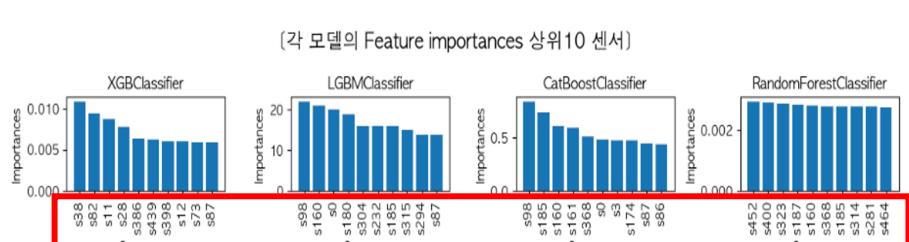
- 기본 threshold(0.50)에서 Accuracy 84.55%
- 최적의 threshold(0.46)에서 Accuracy 85.30%



- 혼동 행렬의 4 가지 항목 (TP, TN, FP, FN) 모두 CatBoost 성능이 가장 좋음
- 모델별 ROC Curve에서 AUC 값
 - XGBoost : 0.85, LGBM : 0.86, CatBoost : 0.91, RandomForest : 0.81
- Feature_importances_ 값은 각 특성의 중요도를 나타냄
 - 각 특성이 모델의 예측에 얼마나 기여했는지
- Feature_importances_ 값은 target 상관관계 값과 관련 없음
 - 각 상위 10 개 센서 중 중복 되는 센서 없음

target과 상관관계 상위 센서

```
s461    0.051789
s462    0.050375
s460    0.048899
s473    0.045961
s472    0.045462
s463    0.044951
s474    0.043726
s471    0.042066
s459    0.041494
s432    0.040031
Name: target, dtype: float64
```



3. CNN 학습 및 결과 예측

```
# CNN 학습 및 예측 결과

# CNN 학습을 위해 데이터 shape 변환
x_train_exp = np.expand_dims(x_train, -1) # x_train 배열의 마지막 차원에 새로운 축을 추가
x_test_exp = np.expand_dims(x_test, -1) # x_train 배열의 마지막 차원에 새로운 축을 추가

# 하이퍼파라미터 설정
epoch = 400 # 학습 반복 횟수
act = 'LeakyReLU' # 활성화 함수
opt = 'adam' # optimizer
filter = 32 # Conv1D Layer의 filter수
batch = 32 # batch size
val_rate = 0.2 # 검증데이터 비율 20%
kernel = 3 # filter의 kernel size

# 모델 정의(구조)

def make_cnn_model():
    model = Sequential()
    # Conv1D 1Layer
    model.add(Conv1D(filters=filter, kernel_size=kernel, activation=act,
                     padding='same', input_shape=(500, 1)))
    model.add(BatchNormalization()) # batch 데이터의 분포를 정규화

    # Conv1D 2Layer
    model.add(Conv1D(filters=filter, kernel_size=kernel, activation=act,
                     padding='same'))
    model.add(BatchNormalization()) # batch 데이터의 분포를 정규화

    # Conv1D 3Layer
    model.add(Conv1D(filters=filter, kernel_size=kernel, activation=act,
                     padding='same'))
    model.add(BatchNormalization()) # batch 데이터의 분포를 정규화

    # GlobalAveragePooling1D Layer
    model.add(GlobalAveragePooling1D())
    # - 각 feature map의 평균을 계산하여 차원을 축소
    # - Flatten() 대신 사용하여 파라미터 수를 줄이고 과적합을 방지

    # Dense Layer
    model.add(Dense(2, activation='softmax'))

    return model

cnn_model = make_cnn_model() # 모델 초기화
```

```

# 모델 컴파일
cnn_model.compile(optimizer=opt, # optimizer: 학습 최적화 알고리즘
                  loss='sparse_categorical_crossentropy', # 사용할 손실 함수
                  metrics=['sparse_categorical_accuracy']) # 모델의 평가를 위해 사용할 지표

# monitor 지표를 기준으로 베스트 모델을 저장
callbacks = [
    ModelCheckpoint('best_model.h5', save_best_only=True,
                    monitor='val_sparse_categorical_accuracy', mode='max')
]

로드 중...

history = cnn_model.fit(x_train_exp, y_train, # 학습 및 라벨 데이터셋
                         batch_size=batch, # batch 크기 지정
                         validation_split=val_rate, # 검증용 데이터의 비율 지정
                         epochs=epochs, # 학습 반복 횟수
                         callbacks=callbacks) # 검증 정확도가 올라갈때만 모델 저장

## 학습 중 정확도 베스트 정확도
max_train_accuracy = max(history.history['sparse_categorical_accuracy'])
max_val_accuracy = max(history.history['val_sparse_categorical_accuracy'])
max_tra_acc_idx = np.argmax(history.history['sparse_categorical_accuracy']) + 1
max_val_acc_idx = np.argmax(history.history['val_sparse_categorical_accuracy']) + 1

# 학습 중 손실 베스트
min_train_loss = min(history.history['loss'])
min_val_loss = min(history.history['val_loss'])
min_tra_loss_idx = np.argmin(history.history['loss']) + 1
min_val_loss_idx = np.argmin(history.history['val_loss']) + 1

print(f'Max train acc : {max_tra_acc_idx}_epoch_{max_train_accuracy}')
print(f'Max val acc: {max_tra_acc_idx}_epoch_{max_val_accuracy}', '\n')
print(f'Min train loss : {min_tra_loss_idx}_epoch_{min_train_loss}')
print(f'Min val loss: {min_val_loss_idx}_epoch_{min_val_loss}', '\n')

# 정확도 베스트 모델로 평가 (베스트 모델은 callbacks에 설정 됨)
model = load_model('best_model.h5')
scores = model.evaluate(x_test_exp, y_test)
print('test acrr ', scores[1])
print('test loss ', scores[0], '\n')

# 정확도, 손실 그래프
plt.figure(figsize=(10, 5))
plt.plot(history.history['sparse_categorical_accuracy'], label='Training accuracy')
plt.plot(history.history['val_sparse_categorical_accuracy'], label='Validation accuracy')
plt.plot(history.history['loss'], label='Training loss')
plt.plot(history.history['val_loss'], label='Validation loss')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(loc='upper left')
plt.show()

# 모델 구조 / 파라미터 개수 확인
model.summary()

```

[CNN 학습 결과]

--- 하이퍼파라미터 최적 조건 (요약)

하이퍼파라미터	조건1	조건2	조건3	조건4
Conv 1D 레이어수	2	3	4	5
활성함수	relu	swish	softplus	LeakyReLU
optimizer	adam	SGD		
필터 수	16	32	64	
Batch_size	16~24	32	64	128
최적모델 저장 기준 (monitor)	val_sparse_categorical_accuracy	val_loss		
검증 데이터 비중	10%	16~20%	30%	
kernel_size	2	3	4	5
데이터 정규화 StandardScaler()	O	X		
배치 정규화 BatchNormalization()	O	X		
epoch	200	300	400	

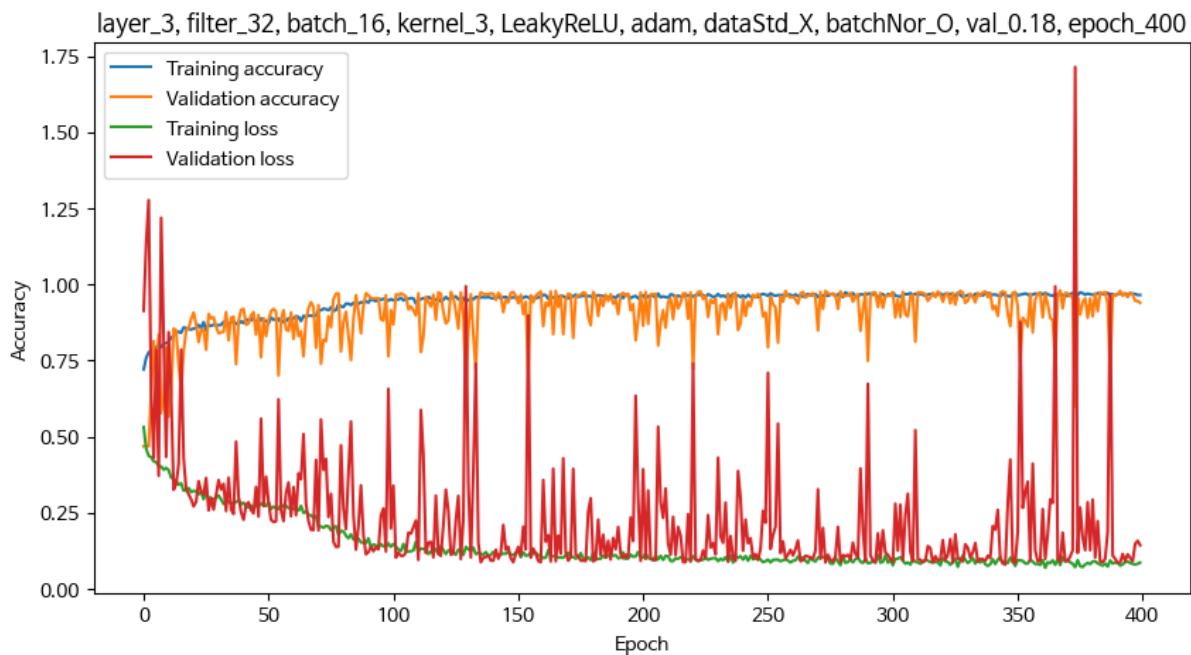
→ 최적 조건으로 테스트 데이터셋 정확도 97.2 ~ 97.5%, loss 0.08 ~ 0.09

- Conv 1D Layer
 - 최적 조건 : 3 Layer
 - 2Layer 또는 4Layer 일때 정확도 떨어짐(95% 이하), 5Layer 일때 과적합 발생
- 활성 함수
 - 최적 함수 : LeakyReLU
 - Relu, swish, softplus 에 비해 과적합이 덜 일어나고, 정확도도 가장 좋게 나타남
- Optimizer
 - 최적 함수 : adam
 - SGD 일때 400epoch 기준 정확도 떨어짐(92%), 2000epoch 까지 늘려도 97%에 도달하지 못함.
- 필터 수
 - 최적 필터 수 : 32
 - 필터 수가 많으면 특징은 더 많이 추출하겠지만 많을수록 성능이 좋아지는 것은 아님.
최적의 필터 수는 실험 결과 32 가 가장 적합함.
- Batch size
 - 최적 배치 수 : 16~24
 - 배치 수가 많을 수록 학습 속도는 빨라지나 정확도가 떨어지는 경향이 있음
 - 실험 결과 과소적합, 과적합이 일어나지 않으면서 정확도나 loss 값이 가장 잘 나오는 조건은 16~24 배치수

- 최적모델 저장 기준(monitor)
 - 최적 기준 : 'val_sparse_categorical_accuracy'
 - 'val_loss' 를 기준으로 할 경우 정확도가 대체로 0.5% 정도 떨어짐
- 검증 데이터 비중
 - 최적 비중 : 15~20% 일때 성능이 가장 좋게 나타남
- Kernel_size
 - 최적 커널 수 : 3
 - 커널 수가 2, 4, 5 일 경우 정확도가 떨어짐 (94% 이하)
 - 센서 간 상관계수에서 살펴본대로 인접 센서와 연관성이 높아 길이 3 의 패턴이 의미있는 특성을 가지고 있는 것으로 보임
- 데이터 정규화(StandardScaler)
 - 최적 조건 : 데이터 정규화 불필요
 - 정규화를 진행했을 경우 머신러닝, CNN 모델 모두 전/후 정확도가 비슷하거나 약간 낮아지는 정도
- 배치 정규화(BatchNormalization)
 - 최적 조건 : 배치 정규화 필요
 - 배치 정규화 안했을 경우 성능 크게 떨어짐 (92% 이하)
- Epoch
 - 최적 조건 : 400 회
 - Epoch 400 회 초과 되어도 성능이 더 이상 개선 되지 않음
(단, SGD optimizer 사용했을 경우는 예외)

[훈련/검증 정확도, 훈련/검증 손실 그래프]

---- 하이퍼파라미터 최적 조건으로 학습 했을 때



[CNN 혼동행렬과 ROC Curve]

```
# CNN 모델의 혼동행렬, ROC Curve

pred_prob = model.predict(x_test_exp) # 클래스 예측 확률
pred_class = np.argmax(pred_prob, axis=1) # 2개의 확률 중 높은 값을 클래스 값으로

fig, axs = plt.subplots(1, 2, figsize=(12,4))
# 혼동 행렬(Confusion Matrix)
confusion = confusion_matrix(y_test, pred_class)
# 히트맵으로 시각화
sns.heatmap(confusion, annot=True, fmt="d", cmap="Blues",
            xticklabels=['0', '1'],
            yticklabels=['0', '1'], ax=axs[0])
axs[0].set_ylabel('Actual')
axs[0].set_xlabel('Predicted')
axs[0].set_title('혼동 행렬(Confusion Matrix)')

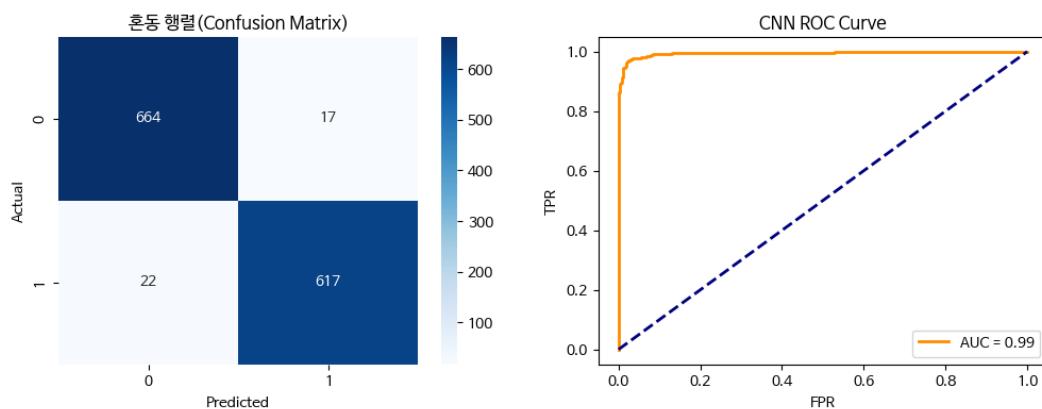
# ROC 그래프 그리기
probs = pred_prob[:, 1] # 클래스 1에 대한 예측 확률
fpr, tpr, thresholds = roc_curve(y_test, probs) #모든 threshold를 구하고,
#각각에 상응되는 fpr, tpr을 array로 반환
roc_auc = auc(fpr, tpr) # ROC Curve의 면적 AUC 계산하기
axs[1].plot(fpr, tpr, color='darkorange', lw=2, label=f'AUC = {roc_auc:.2f}')
axs[1].plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
axs[1].set_title('CNN ROC Curve')
axs[1].set_xlabel('FPR')
axs[1].set_ylabel('TPR')
axs[1].legend(loc="lower right")

# 최적의 threshold 값 찾기
optimal_idx = np.argmax(tpr - fpr) # tpr과 fpr의 차이가 가장 큰 인덱스 찾기
optimal_threshold = thresholds[optimal_idx] # 찾은 인덱스로 최적의 threshold 찾기

# 최적의 threshold 값을 사용하여 다시 예측
y_pred_optimal = np.where(probs > optimal_threshold, 1, 0)
# 새로운 예측 값을 사용하여 정확도 계산
accuracy_optimal = accuracy_score(y_test, y_pred_optimal)
print(f'threshold {round(optimal_threshold,2)} ->\nCNN_acc : {round(accuracy_optimal*100, 2)}%' ) # 최적의 threshold에 대한 정확도 출력
```

✓ 0.4s

MagicPython



➤ CNN 모델 정확도 97.2 ~ 97.5%, AUC = 0.99로 학습이 잘 됨

4. 데이터 전처리 후 재학습(1)

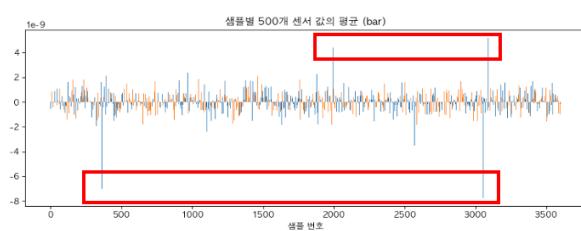
--- 500 개 전체 센서의 평균 값이 높은 샘플(노이즈) 데이터 제거 후 학습

■ 평균 값 상위 10 개 샘플 제거 코드

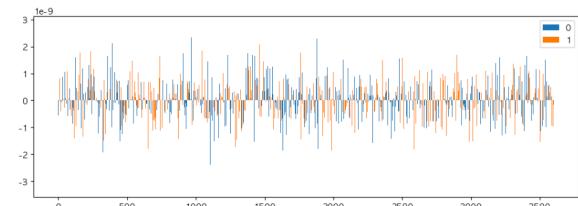
```
# 센서 평균값이 높은 샘플(인덱스) 추출, 제거
drop_idx = abs(train_0.iloc[:, :-1].mean(axis=1)).sort_values(ascending=False).head(10).index
train_0_drop = train_0.drop(drop_idx)
```

■ 상위 10 개 샘플(노이즈) 제거 후 시각화

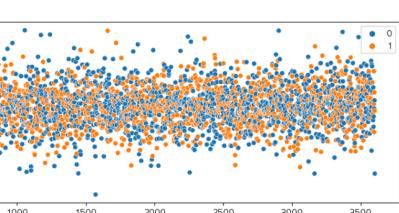
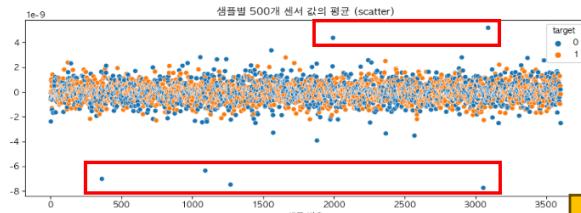
[노이즈 제거 전]



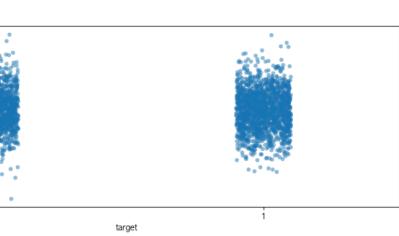
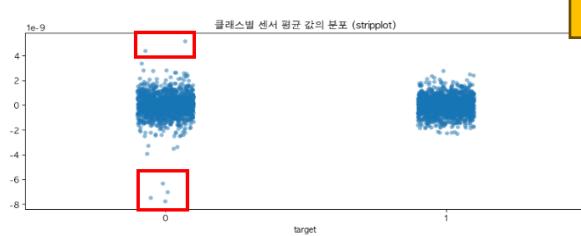
[노이즈 제거 후]



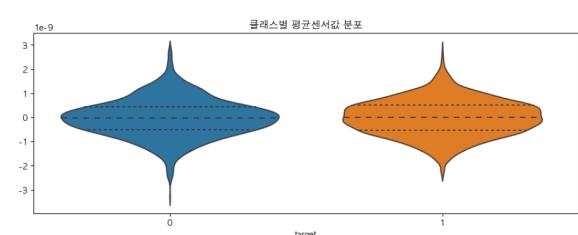
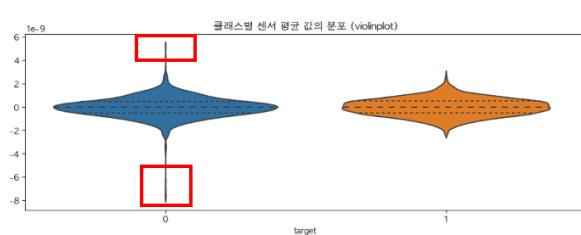
샘플별 500개 센서 값의 평균 (scatter)



클래스별 센서 평균 값의 분포 (stripplot)



클래스별 센서 평균 값의 분포 (violinplot)



■ 상위 5, 10, 20 개 샘플 제거 후 학습 및 예측 결과

→ 성능 개선 되지 않음

- 머신 러닝 모델 : 예측 정확도 비슷하거나 약간 낮은 수준
- CNN 모델 : 예측 정확도 비슷한 수준

5. 데이터 전처리 후 재학습 (2)

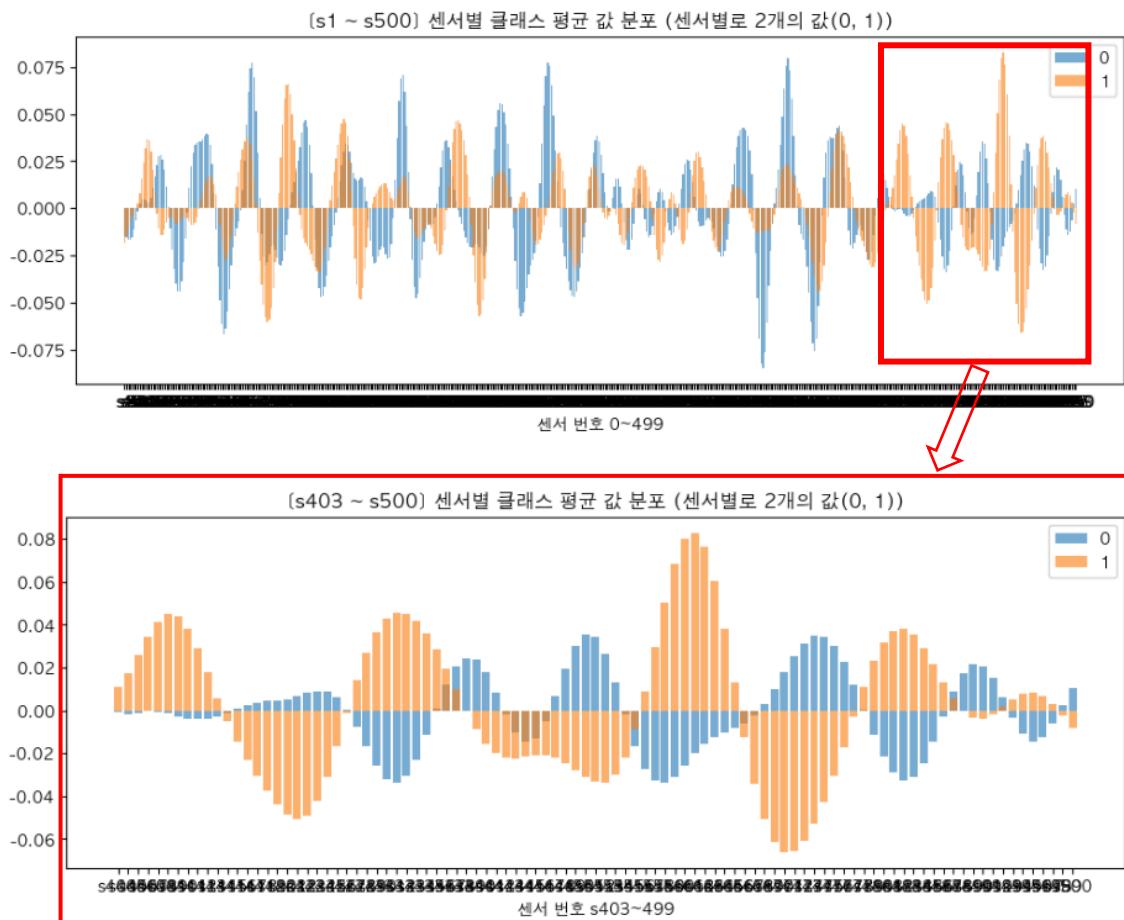
--- S403 센서 이후 데이터만 학습

```
# 가로:센서, 세로:클래스별(0,1) 모든 샘플의 평균값(scatter)
# [s1 ~ s500] 센서별 클래스 평균 값 분포 (센서별로 2개의 값(0, 1))
plt.figure(figsize=(12,4))
plt.bar(s_means.columns, s_means.iloc[0,:], alpha=0.6, label='0')
plt.bar(s_means.columns, s_means.iloc[1,:], alpha=0.6, label='1')
plt.title(' [s1 ~ s500] 센서별 클래스 평균 값 분포 (센서별로 2개의 값(0, 1))')
plt.xlabel('센서 번호 0~499')
plt.legend()

# [s403 ~ s500] 후반부(s403~)만 시각화
plt.figure(figsize=(12,4))
col_i = 403
plt.bar(s_means.columns[col_i:], s_means.iloc[0,col_i:], alpha=0.6, label='0')
plt.bar(s_means.columns[col_i:], s_means.iloc[1,col_i:], alpha=0.6, label='1')
plt.title(' [s403 ~ s500] 센서별 클래스 평균 값 분포 (센서별로 2개의 값(0, 1))')
plt.xlabel('센서 번호 s403~499')
plt.legend()
plt.show()
```

✓ 1.9s

MagicPython



- 403 번 센서부터 0 클래스와 1 클래스의 평균 값이 음수와 양수로 명확히 갈리는 현상. 이부분이 target에 영향을 크게 준다고 추측 할 수 있음
- 403 번 이후 센서 데이터만 학습하여 예측 해보기 (다음 페이지)

■ s403 ~ s500 데이터셋, 라벨 추출하기

```
# s403~s500 센서만 잘라서 새로운 데이터셋과 라벨 만들기
train_403 = train_df.iloc[:, 402:]
test_403 = test_df.iloc[:, 402:]

x_train_403 = train_403.iloc[:, :-1]
y_train_403 = train_403.iloc[:, -1]
x_test_403 = test_403.iloc[:, :-1]
y_test_403 = test_403.iloc[:, -1]

print(x_train_403.shape, y_train_403.shape)
print(x_test_403.shape, y_test_403.shape)
```

[0.0s

MagicPython

```
(3601, 98) (3601,)
(1320, 98) (1320,)
```

■ s403 ~ s500 센서만 학습하여 예측한 결과

- 성능 개선 되지 않음 : 데이터 부족으로 과소적합 발생되는 듯
 - 머신 러닝 모델 : 예측 정확도가 모델별로 2~10% 감소
 - CNN 모델 : 예측 정확도 89~90%로 7% 정도 감소 (loss는 0.09에서 0.25로 증가)

6. 데이터 전처리 후 학습 (3)

- Target 과 상관계수 값이 높은 센서로만 학습
- 상관 계수 낮은 센서 drop (10 개, 30 개 drop 이후 학습 및 예측)
- target 과 상관계수 낮은 센서 drop 시키는 코드

```
# target과 상관계수 낮은 10개 센서 drop 이후 데이터셋 만들기

train_corr = abs(train_df.corr())
corr_target = train_corr.target.sort_values(ascending=False).tail(10)
print(corr_target.index)

train_corr = train_df.drop(corr_target.index, axis=1)
test_corr = test_df.drop(corr_target.index, axis=1)

x_train = train_corr.iloc[:, :-1]
y_train = train_corr.iloc[:, -1]
x_test = test_corr.iloc[:, :-1]
y_test = test_corr.iloc[:, -1]

print(x_train.shape, y_train.shape)
print(x_test.shape, y_test.shape)
✓ 1.4s
```

MagicPython

```
Index(['s160', 's391', 's264', 's377', 's2', 's291', 's341', 's193', 's390',
       's161'],
      dtype='object')
(3601, 490) (3601,)
(1320, 490) (1320,)
```

■ target 과 상관계수 낮은 센서는 제거 후 학습 및 예측 결과

→ 성능 개선 되지 않음

- 머신 러닝 모델 : 모델별로 예측 정확도 비슷하거나, 1~2% 정도 떨어짐
 - 딥러닝 모델 : 하위 10 개 제거 시 정확도 2% 정도 떨어짐, 하위 30 개 제거 시 과소적합 발생 (정확도 90%, loss 0.24)
- 인접 하는 센서와 함께 특징을 구성하지만 중간 중간 데이터가 빠질 경우 특징을 추출하는데 악영향을 주는 것으로 추측