데이터사이언스 TEAM PROJECT

TEAM 2
201700638 김비아
201700658 김서희
201701387 박보성
201501257 박세인

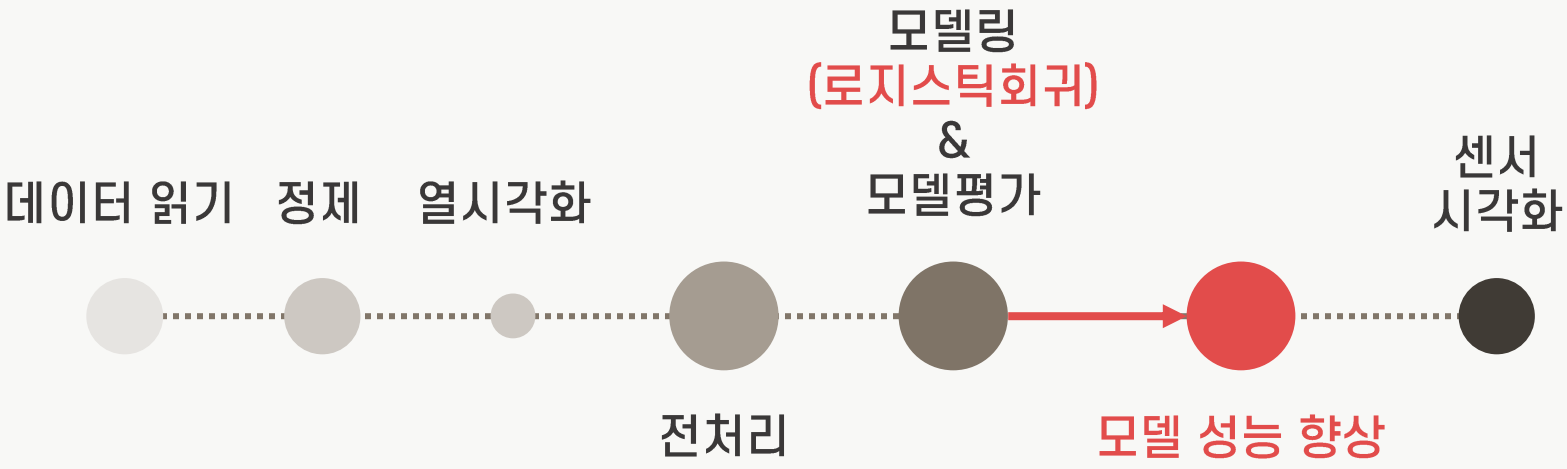
2팀의 주제와 분석 목표

“센서 데이터를 활용한 공정 이상 예측”

어떤 센서가 이상을 발생시키는가?

만든 모델이 Pass/ Fail의 여부를 잘 예측하는가?

목차



Feature:label

Train:test

정규화

- 불균형 데이터 해결 (SMOTE)
- 다양한 모델링 기법 & 평가
- GridSearchCV를 통한
  랜덤포레스트 하이퍼파라미터 튜닝

데이터 읽기

```
data = pd.read_csv('uci-secom.csv')
data.head(6)
```

	Time	0	1	2	3	4	5	6	7	8	...	581	582	583	584	585	586	587
0	2008-07-19 11:55:00	3030.93	2564.00	2187.7333	1411.1265	1.3602	100.0	97.6133	0.1242	1.5005	...	NaN	0.5005	0.0118	0.0035	2.3630	NaN	NaN
1	2008-07-19 12:32:00	3095.78	2465.14	2230.4222	1463.6606	0.8294	100.0	102.3433	0.1247	1.4966	...	208.2045	0.5019	0.0223	0.0055	4.4447	0.0096	0.0201
2	2008-07-19 13:17:00	2932.61	2559.94	2186.4111	1698.0172	1.5102	100.0	95.4878	0.1241	1.4436	...	82.8602	0.4958	0.0157	0.0039	3.1745	0.0584	0.0484
3	2008-07-19 14:43:00	2988.72	2479.90	2199.0333	909.7926	1.3204	100.0	104.2367	0.1217	1.4882	...	73.8432	0.4990	0.0103	0.0025	2.0544	0.0202	0.0149
4	2008-07-19 15:22:00	3032.24	2502.87	2233.3667	1326.5200	1.5334	100.0	100.3967	0.1235	1.5031	...	NaN	0.4800	0.4766	0.1045	99.3032	0.0202	0.0149

- Columns Info: Time, 0부터 589까지의 센서 번호, Pass/Fail
- Pass/Fail: –1 corresponds to a pass and 1 corresponds to a fail
- NaN, 즉 결측치가 관측되고 있음

데이터 정제

- 데이터 내 변수가 많기 때문에 관측을 통한 이상치 처리는 한계
- 따라서 결측치를 처리, 0으로 대체 `data = data.replace(np.NaN, 0)`

결측치 처리 전

```
Time          0
0             6
1             7
2            14
3            14
             ..
586           1
587           1
588           1
589           1
Pass/Fail     0
Length: 592, dtype: int64
```

결측치 처리 후

```
Time          0
0             0
1             0
2             0
3             0
             ..
586           0
587           0
588           0
589           0
Pass/Fail     0
Length: 592, dtype: int64
```

데이터 정제

- Time 변수 삭제

```
data = data.drop(columns = ['Time'], axis = 1)
```

	0	1	2	3	4	5	6	7	8	9	...	581	582	583	584	585	586	587
0	3030.93	2564.00	2187.7333	1411.1265	1.3602	100.0	97.6133	0.1242	1.5005	0.0162	...	0.0000	0.5005	0.0118	0.0035	2.3630	0.0000	0.0000
1	3095.78	2465.14	2230.4222	1463.6606	0.8294	100.0	102.3433	0.1247	1.4966	-0.0005	...	208.2045	0.5019	0.0223	0.0055	4.4447	0.0096	0.0201
2	2932.61	2559.94	2186.4111	1698.0172	1.5102	100.0	95.4878	0.1241	1.4436	0.0041	...	82.8602	0.4958	0.0157	0.0039	3.1745	0.0584	0.0484
3	2988.72	2479.90	2199.0333	909.7926	1.3204	100.0	104.2367	0.1217	1.4882	-0.0124	...	73.8432	0.4990	0.0103	0.0025	2.0544	0.0202	0.0149
4	3032.24	2502.87	2233.3667	1326.5200	1.5334	100.0	100.3967	0.1235	1.5031	-0.0031	...	0.0000	0.4800	0.4766	0.1045	99.3032	0.0202	0.0149
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1562	2899.41	2464.36	2179.7333	3085.3781	1.4843	100.0	82.2467	0.1248	1.3424	-0.0045	...	203.1720	0.4988	0.0143	0.0039	2.8669	0.0068	0.0138
1563	3052.31	2522.55	2198.5667	1124.6595	0.8763	100.0	98.4689	0.1205	1.4333	-0.0061	...	0.0000	0.4975	0.0131	0.0036	2.6238	0.0068	0.0138
1564	2978.81	2379.78	2206.3000	1110.4967	0.8236	100.0	99.4122	0.1208	0.0000	0.0000	...	43.5231	0.4987	0.0153	0.0041	3.0590	0.0197	0.0086
1565	2894.92	2532.01	2177.0333	1183.7287	1.5726	100.0	98.7978	0.1213	1.4622	-0.0072	...	93.4941	0.5004	0.0178	0.0038	3.5662	0.0262	0.0245
1566	2944.92	2450.76	2195.4444	2914.1792	1.5978	100.0	85.1011	0.1235	0.0000	0.0000	...	137.7844	0.4987	0.0181	0.0040	3.6275	0.0117	0.0162

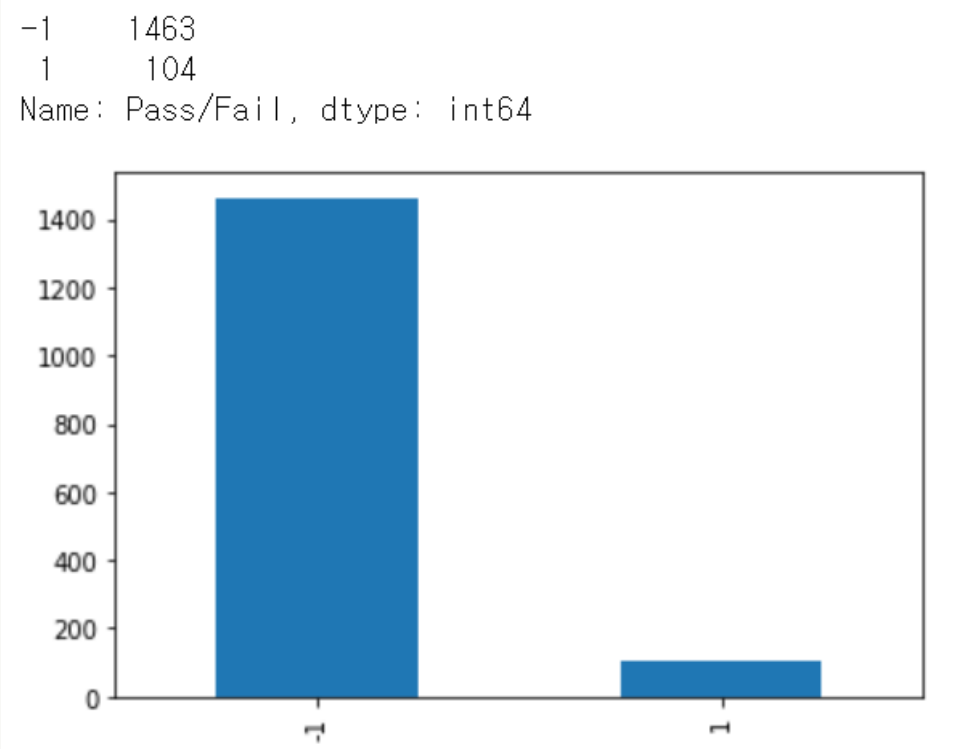
1567 rows × 591 columns

Pass/Fail 열 시각화

- Pass와 Fail 개수를 value_counts로 확인 및 분포 시각화

```
data['Pass/Fail'].value_counts().plot(kind='bar')
data['Pass/Fail'].value_counts()
```

```
-1    1463
 1     104
Name: Pass/Fail, dtype: int64
```

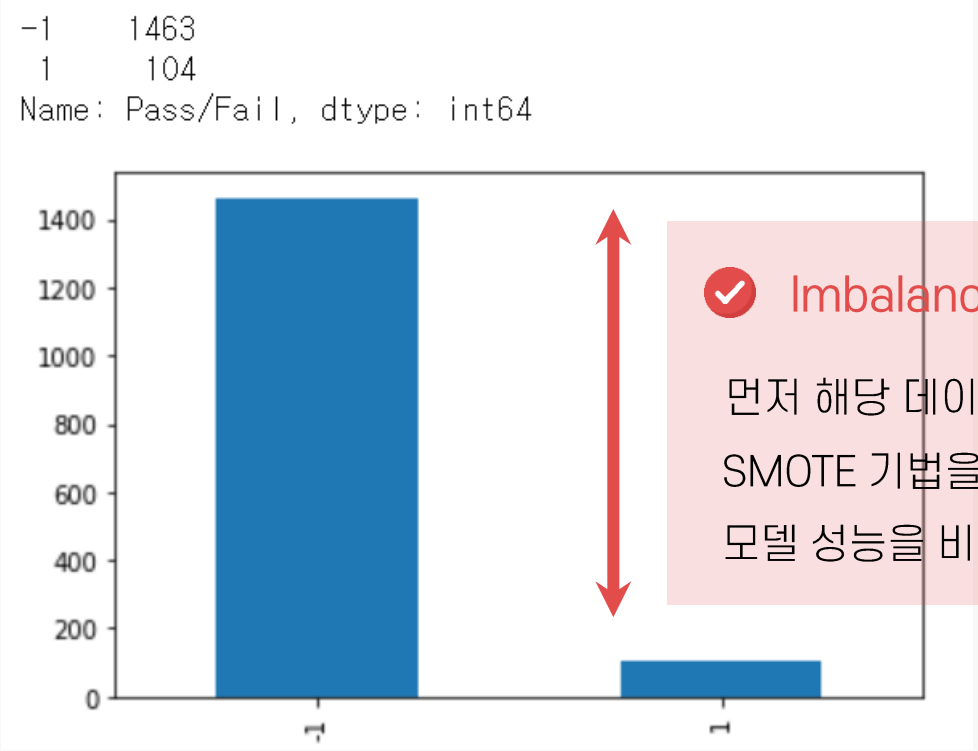


Pass/Fail 열 시각화

- Pass와 Fail 개수를 value_counts로 확인 및 분포 시각화

```
data['Pass/Fail'].value_counts().plot(kind='bar')
data['Pass/Fail'].value_counts()
```

```
-1    1463
 1     104
Name: Pass/Fail, dtype: int64
```



Imbalanced data

먼저 해당 데이터셋으로 모델링을 진행한 후,
SMOTE 기법을 통해 불균형을 해결한 다음
모델 성능을 비교할 것

데이터 전처리

(1) Feature : Label 데이터로 분리

- Feature 데이터인 x와 label 데이터인 y로 분리하기
- 예측해야 할 변수인 Pass/Fail 열을 제거하여 x에 저장
- Pass/Fail 열만을 선택하여 numpy 형태로 y에 저장

```
x = data.drop(columns = ['Pass/Fail'], axis = 1)
y = data['Pass/Fail']

# ravel은 "풀다"로 다차원을 1차원으로 푸는 것을 의미합니다.
# 1차원 벡터 형태로 출력하기 위해 ravel 사용합니다.

y = y.to_numpy().ravel()
y
```

```
array([-1, -1,  1, ..., -1, -1, -1], dtype=int64)
```

데이터 전처리

(2) Train : Test 데이터로 분리

- Sklearn에서 제공하는 train_test_split 사용 (무작위 데이터 분리)
- 트레인 데이터와 테스트 데이터를 8 : 2의 비율로 분리

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state = 0)
```

(3) 정규화

- StandardScaler
- 서로 다른 features의 크기를 맞춤

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()

# x_train에 있는 데이터에 맞춰 정규화를 진행합니다.

x_train = sc.fit_transform(x_train)
x_test = sc.transform(x_test)
x_train, x_test
```

모델링 & 모델 평가

-> 모델링 : **로지스틱 회귀**

- 시그모이드 함수 최적선을 찾고, 이 시그모이드 함수의 반환 값을 확률로 간주해 확률에 따라 분류를 결정
- 로지스틱 회귀는 주로 이진 분류에 사용되는데 예측 값 즉, 예측 확률이 0.5 이상이면 1로, 그렇지 않으면 0으로 예측

```
def modeling(model, x_train, x_test, y_train, y_test):
    model.fit(x_train, y_train)     # 데이터를 학습시킬 때는 fit 함수를 사용
    pred = model.predict(x_test)
    metrics(y_test,pred)

def metrics(y_test, pred):
    accuracy = accuracy_score(y_test,pred)
    precision = precision_score(y_test,pred) #zero_division=1
    recall = recall_score(y_test,pred)
    f1 = f1_score(y_test,pred)
    roc_score = roc_auc_score(y_test,pred,average='macro')
    print('정확도 : {0:.2f}, 정밀도 : {1:.2f}, 재현율 : {2:.2f}'.format(accuracy,precision,recall))
    print('f1-score : {0:.2f}, auc : {1:.2f}'.format(f1,roc_score,recall))
```

모델링 & 모델 평가

- 로지스틱 모델 평가

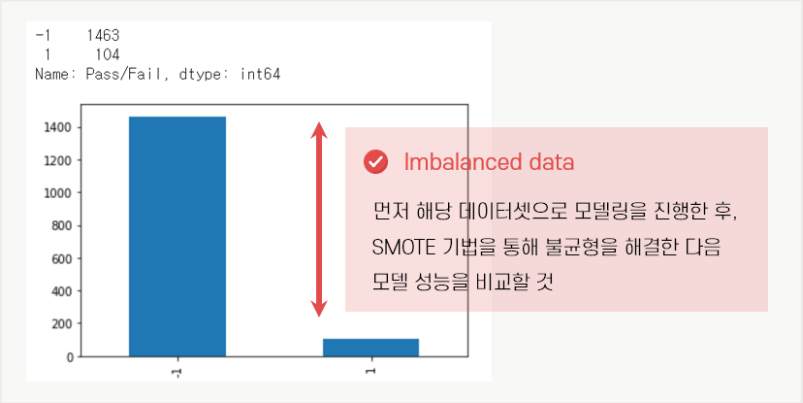
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score,precision_score,recall_score,f1_score,roc_auc_score

LR = LogisticRegression(max_iter =5000)
modeling(LR, x_train, x_test, y_train, y_test)
```

```
정확도 : 0.88, 정밀도 : 0.04, 재현율 : 0.08
f1-score : 0.05, auc : 0.50
```

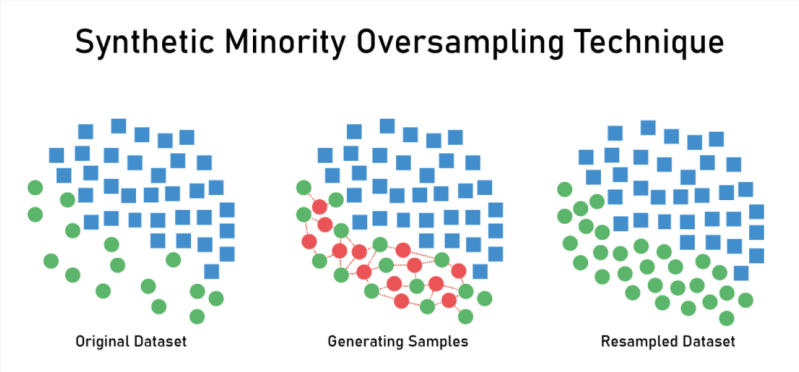
모델 성능 향상

1st trial: 불균형 데이터 해결하기



➢ **Imbalanced data**

SMOTE 기법을 통해 개수가 적은 라벨의 데이터를 오버샘플링하여 불균형을 해결하면 모델 성능이 더 좋아질 수 있을까?



➢ **SMOTE**

데이터의 개수가 적은 클래스의 표본을 가져온 뒤 임의의 값을 추가하여 새로운 샘플을 만들어 데이터에 추가하는 오버샘플링 방식

모델 성능 향상

```
from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=1)

x_train_over, y_train_over = smote.fit_resample(x_train, y_train)

print('SMOTE 적용 전 학습용 피처/레이블 데이터 세트: ', x_train.shape, y_train.shape)
print('SMOTE 적용 후 학습용 피처/레이블 데이터 세트: ', x_train_over.shape, y_train_over.shape)
print()
print('SMOTE 적용 후 레이블 값 분포: ₩n', pd.Series(y_train_over).value_counts())
```

```
SMOTE 적용 전 학습용 피처/레이블 데이터 세트:  (1253, 590) (1253,)
SMOTE 적용 후 학습용 피처/레이블 데이터 세트:  (2324, 590) (2324,)

SMOTE 적용 후 레이블 값 분포:
 1    1162
-1    1162
dtype: int64
```

```
modeling(LR, x_train_over, x_test, y_train_over, y_test)

# 이전 LR 모델
# 정확도 : 0.88, 정밀도 : 0.04, 재현율 : 0.08
# f1-score : 0.05, auc : 0.50
```

```
정확도 : 0.82, 정밀도 : 0.02, 재현율 : 0.08
f1-score : 0.03, auc : 0.47
```

모델 성능 향상

- 오버샘플링 전 `modeling(LR, x_train, x_test, y_train, y_test)`
- 오버샘플링 후 `modeling(LR, x_train_over, x_test, y_train_over, y_test)`

VS

평가지표	오버샘플링 전	오버샘플링 후
정확도	0.88	0.82
정밀도	0.04	0.02
재현율	0.08	0.08
F1-Score	0.05	0.03
AUC	0.50	0.47

=> SMOTE 기법이 유의미하지 않음

모델 성능 향상

2nd trial: 다른 다양한 모델링 기법 시도

- 로지스틱 회귀보다 더 좋은 성능을 내는 모델이 있을까?

로지스틱, KNN, 의사결정나무, 나이브베이즈, 랜덤포레스트, SVC, XGB, LGBM

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
import xgboost as xgb
from xgboost.sklearn import XGBClassifier
from lightgbm import LGBMClassifier
```

```
models = []
models.append(('LR', LogisticRegression(max_iter =5000)))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('RF', RandomForestClassifier()))          #
models.append(('SVM', SVC(gamma='auto')))                #
models.append(('XGB', XGBClassifier()))                  #
models.append(('LGBM', LGBMClassifier()))                #
```

모델 성능 향상

```
for name, model in models:
    model.fit(x_train, y_train)
    msg = "%s - train_score : %f, test score : %f" % (name, r
    print(msg)
```

```
LR - train_score : 0.988029, test score : 0.878981
KNN - train_score : 0.929769, test score : 0.958599
CART - train_score : 1.000000, test score : 0.920382
NB - train_score : 0.201915, test score : 0.152866
RF - train_score : 1.000000, test score : 0.958599
SVM - train_score : 0.932961, test score : 0.958599
XGB - train_score : 1.000000, test score : 0.955414
LGBM - train_score : 1.000000, test score : 0.955414
```

```
for name, model in models:
    model.fit(x_train_over, y_train_over)
    msg = "%s - train_score : %f, test score : %f" % (name, r
    print(msg)
```

```
LR - train_score : 0.990964, test score : 0.821656
KNN - train_score : 0.721601, test score : 0.305732
CART - train_score : 1.000000, test score : 0.831210
NB - train_score : 0.611015, test score : 0.238854
RF - train_score : 1.000000, test score : 0.952229
SVM - train_score : 0.996127, test score : 0.914013
XGB - train_score : 1.000000, test score : 0.942675
LGBM - train_score : 1.000000, test score : 0.945860
```

Oversampling	X		O	
Model	Train score	Test score	Train score	Test score
LR	0.98	0.87	0.99	0.82
KNN	0.92	0.95	0.72	0.30
CART	1.00	0.92	1.00	0.83
NB	0.20	0.15	0.61	0.23
RF	1.00	0.9585	1.00	0.9522
SVM	0.93	0.95	0.99	0.91
XGB	1.00	0.95	1.00	0.94
LGBM	1.00	0.95	1.00	0.94

모델 성능 향상

```
for name, model in models:
    model.fit(x_train, y_train)
    msg = "%s - train_score : %f, test score : %f" % (name, r
    print(msg)
```

```
LR - train_score : 0.988029, test score : 0.878981
KNN - train_score : 0.929769, test score : 0.958599
CART - train_score : 1.000000, test score : 0.920382
NB - train_score : 0.201915, test score : 0.152866
RF - train_score : 1.000000, test score : 0.958599
SVM - train_score : 0.932961, test score : 0.958599
XGB - train_score : 1.000000, test score : 0.955414
LGBM - train_score : 1.000000, test score : 0.955414
```

```
for name, model in models:
    model.fit(x_train_over, y_train_over)
    msg = "%s - train_score : %f, test score : %f" % (name, r
    print(msg)
```

```
LR - train_score : 0.990964, test score : 0.821656
KNN - train_score : 0.721601, test score : 0.305732
CART - train_score : 1.000000, test score : 0.831210
NB - train_score : 0.611015, test score : 0.238854
RF - train_score : 1.000000, test score : 0.952229
SVM - train_score : 0.996127, test score : 0.914013
XGB - train_score : 1.000000, test score : 0.942675
LGBM - train_score : 1.000000, test score : 0.945860
```

오버샘플링하지 않은 데이터를 사용한 RF의 성능이 제일 좋음

로지스틱 회귀 -> 랜덤 포레스트

KNN	0.92	0.95	0.72	0.30
CART	1.00	0.92	1.00	0.83
NB	0.20	0.15	0.61	0.23
RF	1.00	0.9585	1.00	0.9522
SVM	0.93	0.95	0.99	0.91
XGB	1.00	0.95	1.00	0.94
LGBM	1.00	0.95	1.00	0.94

모델 성능 향상

3rd trial: GridSearchCV를 통한 RF 하이퍼파라미터 튜닝

➢ 랜덤 포레스트의 성능을 더 향상시킬 수 있을까? (현재 0.9586)

```
rf_clf1 = RandomForestClassifier(n_estimators = 10,
                                 max_depth = 10,
                                 min_samples_leaf = 6,
                                 min_samples_split = 6,
                                 random_state = 0,
                                 n_jobs = -1)

rf_clf1.fit(x_train, y_train)
pred = rf_clf1.predict(x_test)
print('예측 정확도: {:.4f}'.format(accuracy_score(y_test,pred)))
```

```
예측 정확도: 0.9586
```

```
from sklearn.model_selection import GridSearchCV

params = { 'n_estimators' : [10, 200],
           'max_depth' : [10, 15, 20],
           'min_samples_leaf' : [6, 8, 10],
           'min_samples_split' : [6, 8, 10]
           }

# RandomForestClassifier 객체 생성 후 GridSearchCV 수행

rf_clf = RandomForestClassifier(random_state = 0, n_jobs = -1)
grid_cv = GridSearchCV(rf_clf, param_grid = params, cv = 5, n_jobs = -1)
grid_cv.fit(x_train, y_train)

print('최적 하이퍼 파라미터: ', grid_cv.best_params_)
print('최고 예측 정확도: {:.4f}'.format(grid_cv.best_score_))
```

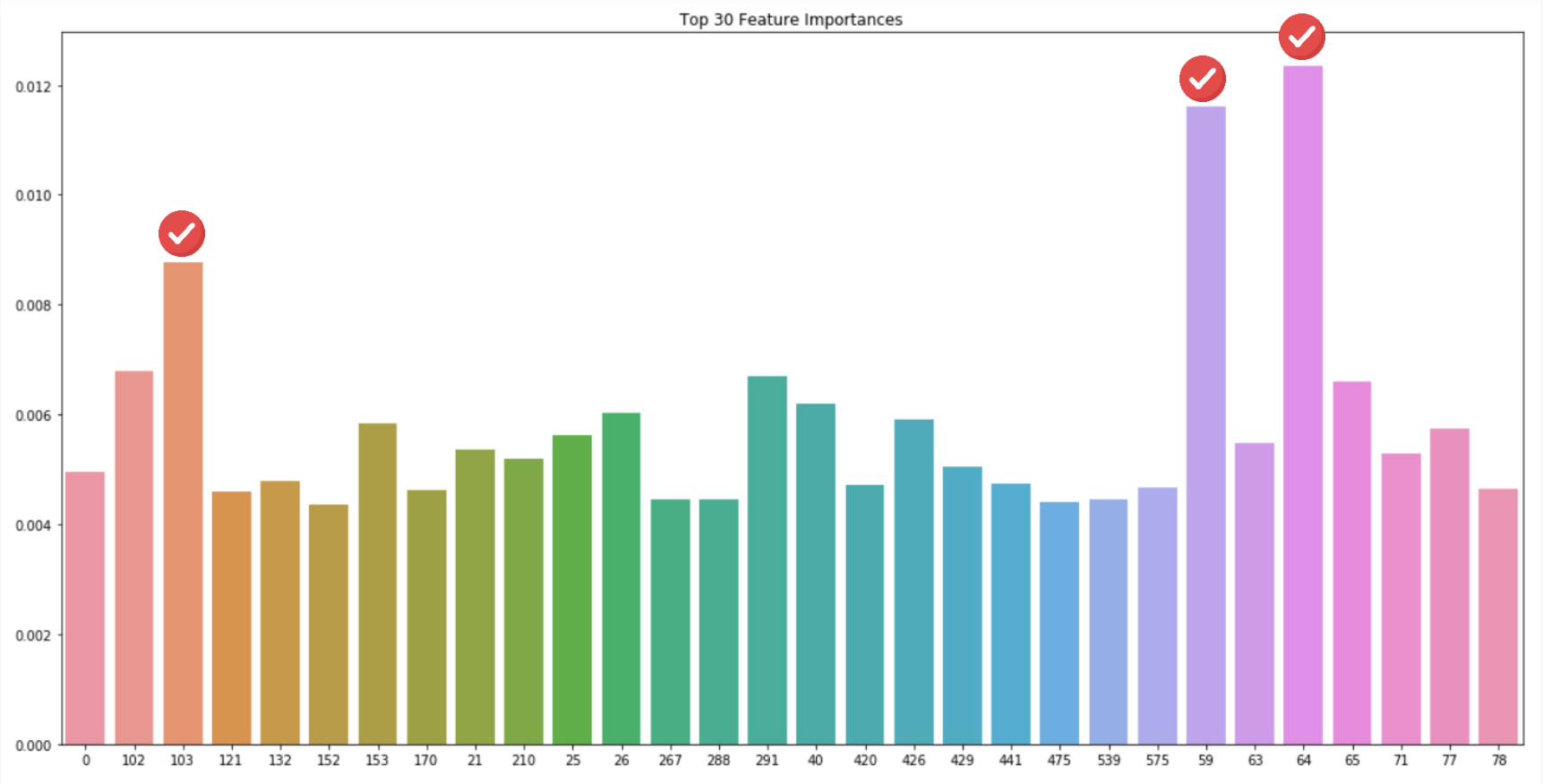
```
최적 하이퍼 파라미터:  {'max_depth': 10, 'min_samples_leaf': 6, 'min_samples_split': 6, 'n_estimators': 10}
최고 예측 정확도: 0.9274
```

➢ 성능 향상되지 않음

➢ 오버샘플링하지 않은 RF 모델 채택

유의미한 센서 시각화

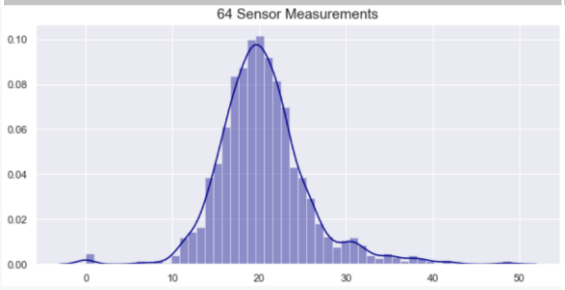
- 가장 중요한 변수 상위 30개 출력 : 64번, 59번, 103번 센서

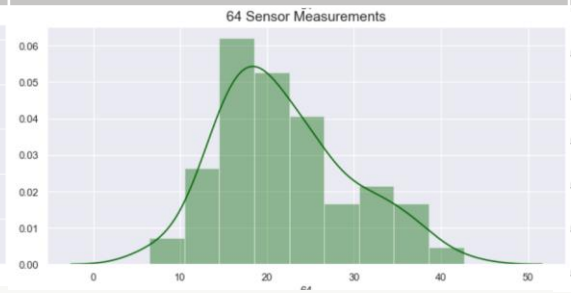


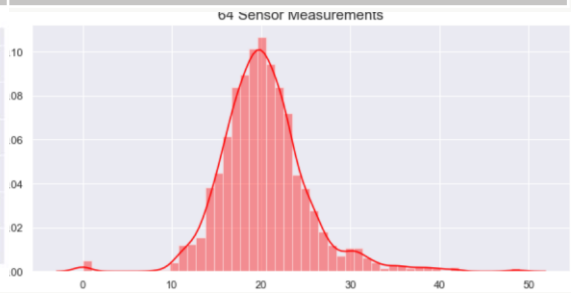
유의미한 센서 시각화

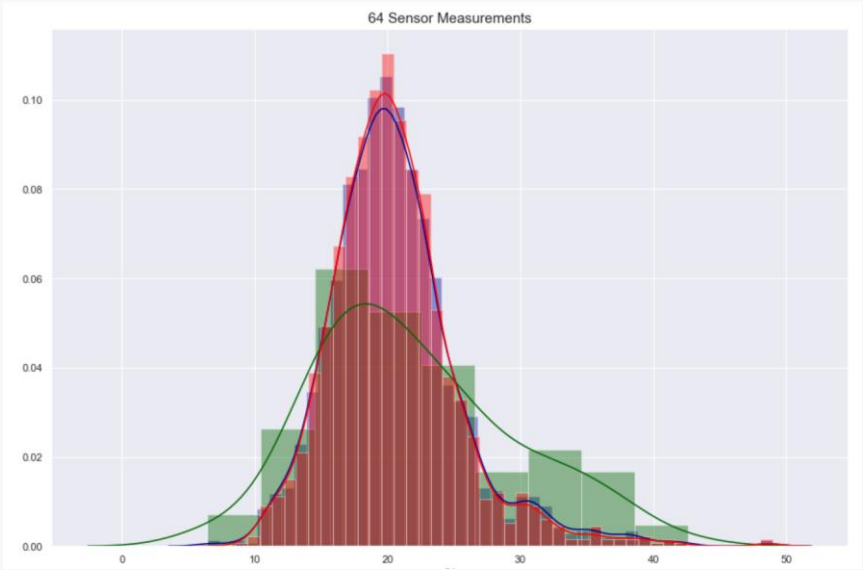
- 64번 센서 시각화

전체 분포	Fail(1)	Pass(-1)





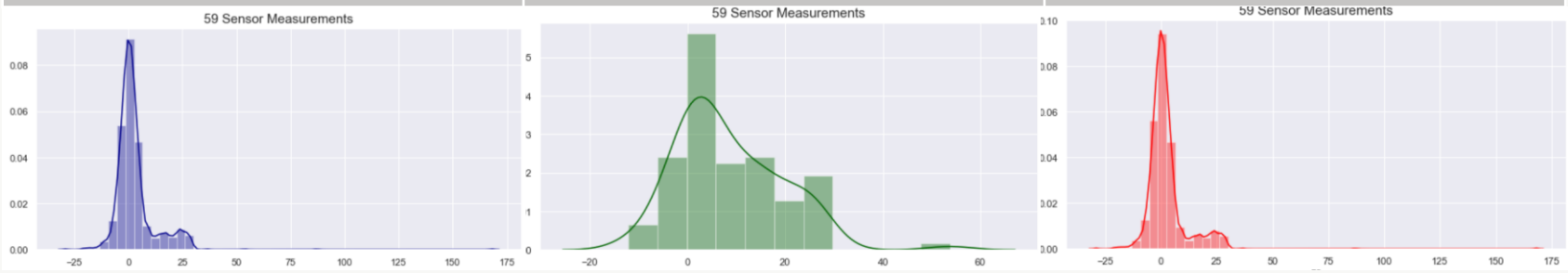


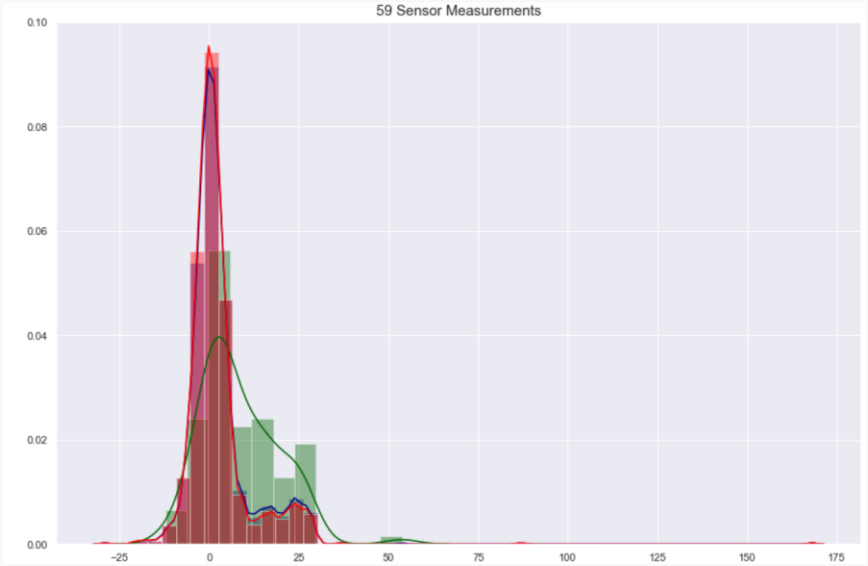


유의미한 센서 시각화

- 59번 센서 시각화

전체 분포	Fail(1)	Pass(-1)

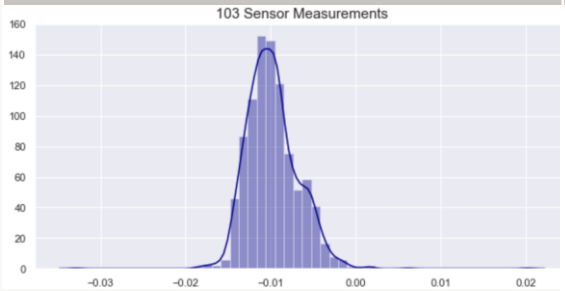


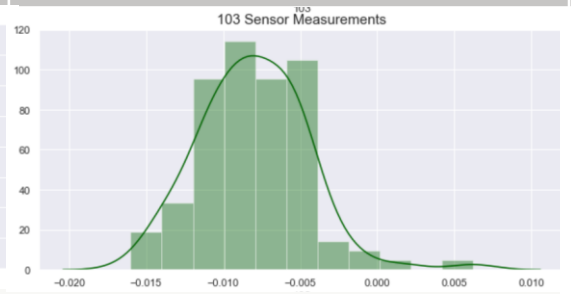


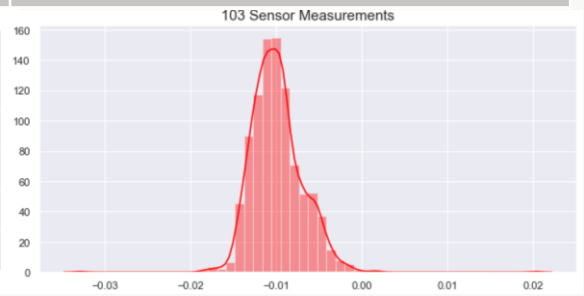
유의미한 센서 시각화

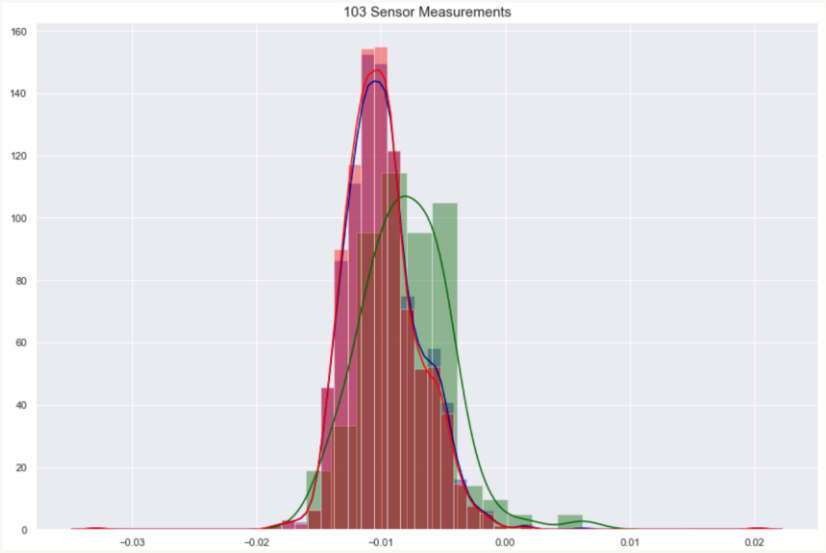
- 103번 센서 시각화

전체 분포	Fail(1)	Pass(-1)









감사합니다!