GradientBoost

잔차를 이용하여 이전 모형의 약점을 보완하는 새로운 모형을 순차적으로 적합한 뒤 이들을 선형 결합하여 얻어진 모형을 생성하는 지도 학습 알고리즘

특징	내용
장점	1. 구현이 쉽다 2. 정확도가 좋다. 3.유연하다(의사결정나무 이외에 다른 알고리즘 적용 가능. 여러가지 손실함수 적용 가능)
단점	1. 과적합 발생 가능성 큼 2. 메모리 사용량이 큼 3. 해석이 어려움

GBM은 과적합에도 강한 뛰어난 예측 성능을 가진 알고리즘이지만 메모리 소비가 크고, 수행 시간이 오래 걸린다는 단점이 있다. (하드웨어 요구사항이 다른 알고리즘에 비해 높음)

이 알고리즘이 처음 소개되고 이를 기반으로한 많은 알고리즘이 나왔으며 가장 각광 받고 있는 ML 패키지는 XGBoost와 LightGBM이다.

#01. 패키지 가져오기

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

from matplotlib import pyplot as plt
from pandas import read_excel
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
from sklearn.metrics import roc_curve, roc_auc_score, auc, RocCurveDisplay
from sklearn.model_selection import GridSearchCV
from imblearn.over_sampling import SMOTE
from sklearn.tree import DecisionTreeClassifier

from sklearnex import patch_sklearn
from daal4py.oneapi import sycl_context
patch_sklearn()
```

```
Intel(R) Extension for Scikit-learn* enabled (https://github.com/intel/scikit-learn-in
```

#02. 데이터 가져오기

```
origin = read_excel('https://data.hossam.kr/G02/breast_cancer.xlsx')
origin.head()
```

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symme
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809

5 rows × 31 columns

#03. 데이터 전처리

독립/종속 변수 분리

```
x = origin.drop('target', axis=1)
y = origin['target']
x.shape, y.shape
```

```
((569, 30), (569,))
```

훈련, 검증 데이터 분리

```
x_train, x_test, y_train, y_test = train_test_split(x, y, random_state = 123)
x_train.shape, x_test.shape, y_train.shape, y_test.shape
```

```
((426, 30), (143, 30), (426,), (143,))
```

데이터 불균형 처리

```
smote_sampler = SMOTE(sampling_strategy="minority", random_state=777)
x_sm, y_sm = smote_sampler.fit_resample(x_train, y_train)
print(x_sm.shape, y_sm.shape)

y_sm.value_counts().sort_index()
```

```
(536, 30) (536,)
```

```
0    268
1    268
Name: target, dtype: int64
```

#04. 훈련 모델 적합

단일 모형

하이퍼파라미터

파라미터	설명
loss	경사 하강법에서 사용할 비용 함수를 지정 `deviance`(기본값), `exponential`
learning_rate	GBM이 학습을 진행할 때마다 적용하는 학습률 `0 ~ 1`의 값(기본값=0.1)
n_estimators	반복수 또는 base_estimator 개수(기본값=100)
subsample	weak learner가 학습에 사용하는 데이터의 샘플링 비율 (기본값=1) 과적합이 염려되는 경우 subsample을 1보다 작은 값으로 설정

learning_rate

Weak learner가 순차적으로 오류 값을 보정해 나가는데 적용하는 계수.

너무 작은 값을 적용하면 업데이트 되는 값이 작아져서 최소 오류 값을 찾아 예측 성능이 높아질 가능성이 높지만 많은 weak learner는 순차적인 반복이 필요해서 수행 시간이 오래 걸리고, 모든 weak learner의 반복이 완료되어도 최소 오류 값을 찾지 못할 수 있다.

반대로 큰 값을 적용하면 최소 오류 값을 찾지 못하고 그냥 지나쳐 버려 예측 성능이 떨어질 가능성이 있지만 빠른 수행이 가능하다.

이러한 특성때문에 learning_rate는 n_estimators와 상호 보완적으로 조합해 사용하는데 작은 learning_rate를 사용하고 n_estimator를 크게 하면 최고의 성능을 보일 수 있는 지점까지 학습이 가능하겠지만 수행 시간이 너무 오래 걸리고 예측 성능도 들이는 시간 만큼 현격하게 좋아지지는 않는다.

```
ada = GradientBoostingClassifier(
        loss='deviance',
        learning_rate=1,
        n_estimators=10,
        subsample=0.8,
        min_samples_leaf=5,
        max_depth=3,
        random_state=100)

ada.fit(x_sm, y_sm)

print("훈련 정확도: ", ada.score(x_sm, y_sm))

y_pred = ada.predict(x_test)
print("테스트 정확도: ", accuracy_score(y_test, y_pred))
```

```
훈련 정확도:  1.0
테스트 정확도:  0.9790209790209791
```

분류 보고서

```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.98      0.96      0.97        54
           1       0.98      0.99      0.98        89

    accuracy                           0.98       143
   macro avg       0.98      0.98      0.98       143
weighted avg       0.98      0.98      0.98       143
```

ROC 곡선

각 클래스에 속할 확률에서 `1`에 속할 확률만 구함

```
score1 = ada.predict_proba(x_test)[:, 1]
score1[:5]
```

```
array([9.99782582e-01, 9.95270649e-01, 3.25112138e-04, 9.99803551e-01,
       3.25112138e-04])
```

ROC 점수 구하기

실제 Label과 Positive Label에 대한 예측 확률을 전달하여 roc 곡선 표현에 필요한 값들을 리턴받는다.

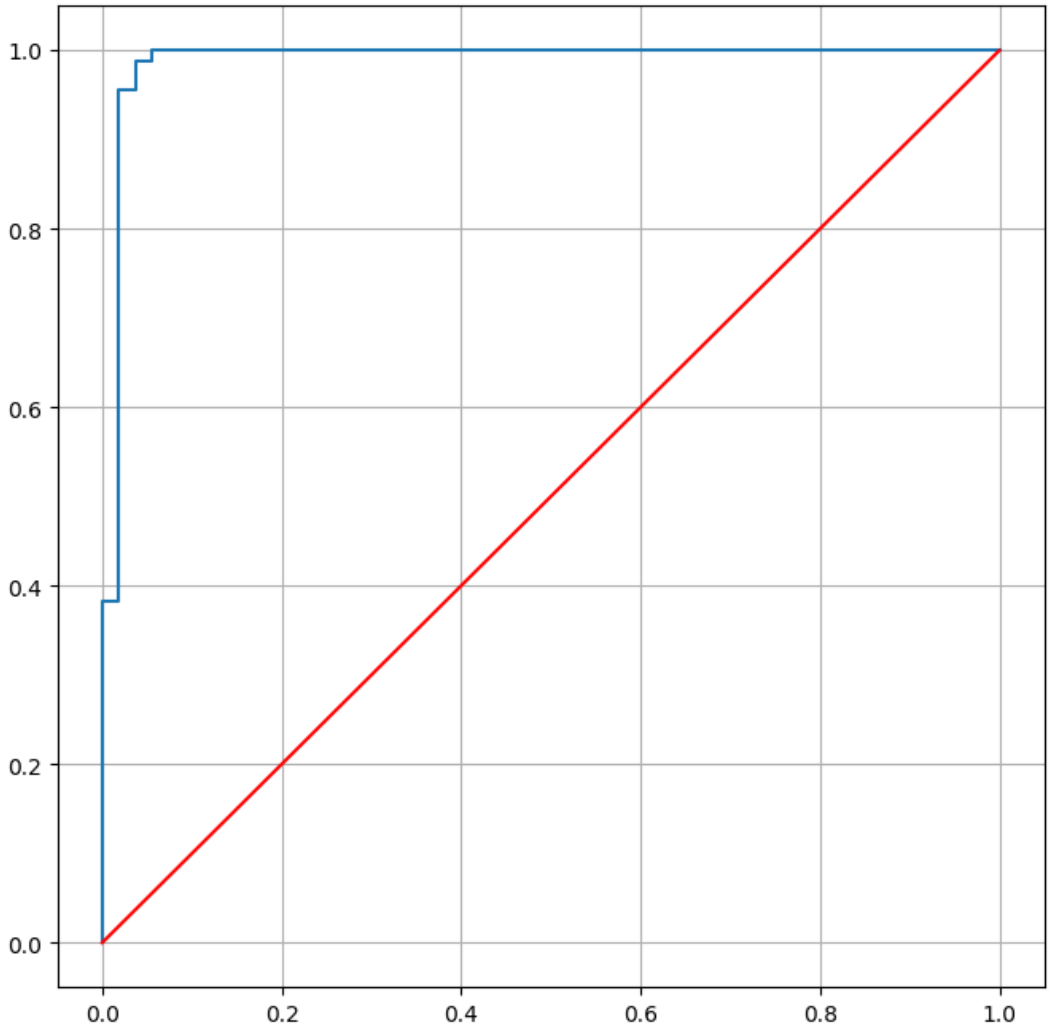
- 첫 번째 리턴값 : False Positive Rate(민감도)
- 두 번째 리턴값 : True Positive Rate(재현율)
- 세 번째 리턴값: 절단값(ROC커브 구현에 사용되지 않음)

```
fpr1, tpr1, cut1 = roc_curve(y_test, score1)
```

ROC 곡선 시각화

가운데 직선에 가까울 수록 분류 성능이 떨어지는 것이다.

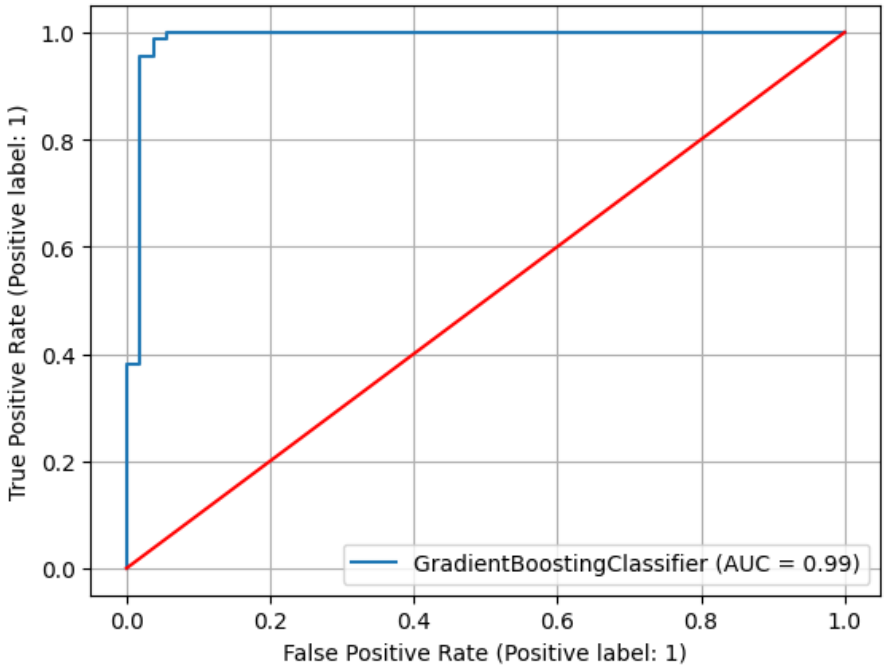
```
plt.figure(figsize=(8,8))
plt.plot(fpr1, tpr1)
plt.plot([0, 1], [0, 1], color='red')
plt.grid()
plt.show()
```



좀 더 이쁜 ROC 곡선

```
plt.figure(figsize=(8,8))
RocCurveDisplay.from_estimator(ada, x_test, y_test)
plt.plot([0, 1], [0, 1], color='red')
plt.grid()
plt.show()
plt.close()
```

```
<Figure size 800×800 with 0 Axes>
```



AUC 값 직접 계산하기

실제 Label과 Positive의 예측확률로 계산

```
print('roc_auc_score 함수 결과:', roc_auc_score(y_test, score1))
```

```
roc_auc_score 함수 결과: 0.9875156054931336
```

False Positive Rate와 True Positive Rate로 계산

```
print('auc 함수 결과:', auc(fpr1, tpr1))
```

```
auc 함수 결과: 0.9875156054931336
```