03_머신러닝 분류 평가

머신러닝
분류 평가

• 평가(Evaluation)

> 머신러닝은 데이터 가공/변환(전처리), 모델 학습/예측, 평가의 프로세스로 구성된다.

> 성능 평가 지표는 일반적으로 모델이 분류냐 회귀냐에 따라 여러 종류로 나뉜다.

> 분류의 평가방법은 실제 결과 데이터와 예측 결과 데이터가 얼마나 정확하고 오류가 적게 발생하는가에 기반한다.

> 회귀의 평가방법은 실제값과 예측값의 오차 평균값에 기반한다.

머신러닝
분류 평가

• 평가(Evaluation)

> **분류 모델의 성능 평가 지표의 종류**
- 정확도(Accuracy)
- 오차행렬(Confusion Matrix)
- 정밀도(Precision)
- 재현율(Recall)
- F1 스코어
- ROC AUC가 있다.

> **회귀 모델의 성능 평가 지표의 종류**
- MAE(Mean Absolute Error)
- MSE(Mean Squared Error)
- RMSE(Root Mean Squared Error)
- R^2

머신러닝 분류 평가

• 정확도

> 정확도는 실제 데이터에서 예측 데이터가 얼마나 같은지를 판단하는 지표

$$\text{정확도}(Accuracy) = \frac{\text{예측 결과가 동일한 데이터 건수}}{\text{전체 예측 데이터 건수}}$$

> 가장 직관적으로 모델 예측 성능을 나타내는 평가 지표

> 그러나 이진 분류의 경우 데이터의 구성에 따라 성능을 왜곡할 수 있기에 정확도 수치 하나만 가지고 모델 성능을 평가하지 않는다.

머신러닝 분류 평가

- 정확도

> 성능 왜곡 실습

```
from sklearn.base import BaseEstimator
# 남성이면 사망, 여성이면 생존으로 예측하는 모델
class MyDummyClassifier(BaseEstimator):
    def fit(self, X, y = None):
        pass
    def predict(self, X):
        pred = np.zeros((X.shape[0], 1))
        for i in range(X.shape[0]):
            if X['Sex'].iloc[i] == 'male':
                pred[i] = 0
            else:
                pred[i] = 1
        return pred
```

머신러닝 분류 평가

- 정확도

> 성능 왜곡 실습

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

X_titanic_df = pd.read_csv('titanic_X.csv')
y_titanic_df = pd.read_csv('titanic_y.csv')

X_train, X_test, y_train, y_test = train_test_split(X_titanic_df, y_titanic_df,
                                                    test_size = 0.2, random_state = 0)
```

머신러닝 분류 평가

- 정확도

> 성능 왜곡 실습

```
# 생성한 모델을 사용하여 학습/예측/평가
myclf = MyDummyClassifier()
myclf.fit(X_train, y_train)
mypredictions = myclf.predict(X_test)
print('정확도 :', accuracy_score(y_test, mypredictions))
```

Out : 정확도 : 0.7877094972067039

머신러닝 분류 평가

- 정확도

> 불균형 데이터 예측

```
# 학습시 아무런 행동을 하지않고 0으로 예측하는 모델 생성
class MyFakeClassifier(BaseEstimator):
    def fit(self,X,y):
        pass
    def predict(self,X):
        return np.zeros( (len(X), 1))
```

• 정확도

> 불균형 데이터 예측

```
digits = load_digits()
print(digits.data)
print(digits.target)
# 번호가 7이면 True==1, 그 외는 False==0
y = (digits.target == 7).astype(int)
X_train, X_test, y_train, y_test = train_test_split( digits.data, y, random_state=11)
```

Out :

```
[[ 0.  0.  5. ...  0.  0.  0.]
 [ 0.  0.  0. ... 10.  0.  0.]
 [ 0.  0.  0. ... 16.  9.  0.]
 ...
 [ 0.  0.  1. ...  6.  0.  0.]
 [ 0.  0.  2. ... 12.  0.  0.]
 [ 0.  0. 10. ... 12.  1.  0.]]
[0 1 2 ... 8 9 8]
```

머신러닝 분류 평가

- 정확도

> 불균형 데이터 예측

```
# 불균형한 레이블 데이터 분포도 확인
print('레이블 테스트 세트 크기 :', y_test.shape)
print('테스트 세트 레이블 0 과 1의 분포도')
print(pd.Series(y_test).value_counts())

# 생성한 모델로 학습/예측/평가
fakeclf = MyFakeClassifier()
fakeclf.fit(X_train , y_train)
fakepred = fakeclf.predict(X_test)
print(f'정확도:{accuracy_score(y_test , fakepred):.3f}')
```

• 정확도

> 불균형 데이터 예측

Out :
```
레이블 테스트 세트 크기 : (450,)
테스트 세트 레이블 0 과 1의 분포도
0    405
1     45
dtype: int64
모든 예측을 0으로 하여도 정확도는:0.900
```

결과값이 전부 0임에도 불구하고 테스트 데이터의 90%가 0을 가지고 있어 정확도는 90%에 해당한다.

이러한 한계점을 극복하기 위해서 여러 가지 분류 지표와 함께 사용한다.

머신러닝 분류 평가

• 오차 행렬

> 학습된 분류 모델이 예측한 분류와 실제 데이터의 분류 범주를 교차 표로 나타낸 것이다.

> 4분면 행렬에서 왼쪽, 오른쪽을 예측 값 기준으로 Negative, Positive로 분류하고, 위, 아래를 실제 값 기준으로 Negative, Positive로 분류한다.

실제 클래스 (Actual Class) \ 예측 클래스 (Predicted Class)	Negative(0)	Positive(1)
Negative(0)	TN (True Negative)	FP (False Positive)
Positive(1)	FN (False Negative)	TP (True Positive)

• 오차 행렬

> confusion_matrix()

```
# 앞에서 만든 MyFakeClassifier의 예측 결과의 오차행렬
from sklearn.metrics import confusion_matrix


confusion_matrix(y_test, fakepred)
```

Out : array([[405, 0],
[45, 0]], dtype=int64)

> TP, TN, FP, TN 값을 조합하여 주요 지표인 정확도, 정밀도, 재현율 값을 알 수 있다.

$$\text{정확도} = \frac{\text{예측 결과와 실제 값이 동일한 건수}}{\text{전체 데이터 수}} = \frac{(TN + TP)}{(TN + FP + FN + TP)}$$

• 정밀도와 재현율

> 정밀도와 재현율은 Positive 데이터 세트의 예측 성능에 초점을 맞춘 평가 지표이다.
> 정밀도와 재현율은 다음과 같은 공식으로 계산한다.

$$\text{정밀도} = \frac{TP}{(FP + TP)} \qquad \text{재현율} = \frac{TP}{(FN + TP)}$$

> 정밀도는 예측 값이 Positive일 때, TP 비율
  - 모델의 결과가 얼마나 정밀한가?
  - 모델이 말하는 Positive는 정말 Positive한가?
> 재현율은 실제 값이 Positive일 때, TP 비율
  - 모델이 얼마나 일관적인가?
  - Positive한 결과를 넣으면 정말 Positive로 나오는가?

• 정밀도와 재현율

> **정밀도와 재현율 중 중요시 해야하는 점**

> **정밀도는 실제 Positive 데이터를 Negative로 잘못 판단하게 되면 업무상 큰 영향이 발생하는 경우에 더 중요시 한다.**
  - 실제 Positive인 암 환자를 Negative로 잘못 판단하게 되면 사망에 이를 수 있다.

> **재현율은 실제 Negative 데이터를 Positive로 잘못 판단하게 되면 업무상 큰 영향이 발생하는 경우에 더 중요시 한다.**
  - 스팸 여부가 Negative인 메일을 Positive로 잘못 판단하게 되면 중요한 메일을 못 받을 수 있다.

• 정밀도와 재현율

> precision_score, recall_score

```
from sklearn.metrics import precision_score, recall_score

def get_clf_eval(y_test , pred):
    confusion = confusion_matrix( y_test, pred)
    accuracy = accuracy_score(y_test , pred)
    precision = precision_score(y_test , pred)
    recall = recall_score(y_test , pred)
    print('오차 행렬')
    print(confusion)
    print(f'정확도: {accuracy:.4f}, 정밀도: {precision:.4f}, 재현율: {recall:.4f}')

get_clf_eval(y_test, fakepred)
```

• 정밀도와 재현율

> precision_score, recall_score

```
# 타이타닉 데이터
X_titanic_df = pd.read_csv('titanic_X.csv')
y_titanic_df = pd.read_csv('titanic_y.csv')

X_train, X_test, y_train, y_test = train_test_split(X_titanic_df, y_titanic_df,
                                                    test_size = 0.2, random_state = 0)
```

• 정밀도와 재현율

> precision_score, recall_score

```
from sklearn.linear_model import LogisticRegression


lr_clf = LogisticRegression(solver='liblinear')
lr_clf.fit(X_train , y_train)
pred = lr_clf.predict(X_test)
get_clf_eval(y_test , pred)
```

Out :
```
오차 행렬
[[92 18]
 [20 49]]
정확도: 0.7877, 정밀도: 0.7313, 재현율: 0.7101
```

• 정밀도와 재현율

> **정밀도/재현율 트레이드오프**

- 업무의 특성상 정밀도 또는 재현율이 특별히 강조돼야 할 경우 임곗값(Threshold)을 조정하여 수치를 높일 수 있다.
- 하지만 정밀도와 재현율은 상호 보완적인 평가 지표이기에 한쪽을 높이면 다른 하나의 수치는 떨어지기 쉽다.

> **개별 레이블 별로 결정 확률을 구하고 임곗값을 변경하여 예측 값의 결과를 바꿀 수 있다.**

머신러닝
분류 평가

- 정밀도와 재현율

> predict_proba

```
pred_proba = lr_clf.predict_proba(X_test)
pred  = lr_clf.predict(X_test)
print('pred_proba()결과 Shape :', pred_proba.shape)
print('pred_proba array에서 앞 3개만 샘플로 추출 ₩n:', pred_proba[:3])

# 예측 확률과 예측 결과값을 한눈에 확인
pred_proba_result = np.concatenate([pred_proba, pred.reshape(-1,1)],axis=1)
print('두개의 class 중에서 더 큰 확률을 클래스 값으로 예측 ₩n', pred_proba_result[:3])
```

• 정밀도와 재현율

> predict_proba

Out :
```
pred_proba()결과 Shape : (179, 2)
pred_proba array에서 앞 3개만 샘플로 추출
: [[0.85016752 0.14983248]
 [0.87512771 0.12487229]
 [0.9254905  0.0745095 ]]
두개의 class 중에서 더 큰 확률을 클래스 값으로 예측
 [[0.85016752 0.14983248 0.        ]
 [0.87512771 0.12487229 0.        ]
 [0.9254905  0.0745095  0.        ]]
```

• 정밀도와 재현율

> 임곗값 조정하기

```
from sklearn.preprocessing import Binarizer

X = [[ 1, -1,  2],
     [ 2,  0,  0],
     [ 0,  1.1, 1.2]]
binarizer = Binarizer(threshold= 1.1)
print(binarizer.fit_transform(X))
```

Out : [[0. 0. 1.]
[1. 0. 0.]
[0. 0. 1.]]

• 정밀도와 재현율

> 임곗값 조정하기

```
custom_threshold = 0.4
pred_proba_1 = pred_proba[:,1].reshape(-1,1)

binarizer = Binarizer(threshold = custom_threshold ).fit(pred_proba_1)
custom_predict = binarizer.transform(pred_proba_1)

get_clf_eval(y_test, custom_predict)
```

Out :

```
오차 행렬
[[86 24]
 [17 52]]
정확도: 0.7709, 정밀도: 0.6842, 재현율: 0.7536
```

• 정밀도와 재현율

> 여러 개의 임곗값 조정하기

```
thresholds = [0.4, 0.45, 0.50, 0.55, 0.60]

def get_eval_by_threshold(y_test ,pred_proba_c1,thresholds):
    for custom_threshold in thresholds:
        binarizer = Binarizer(threshold = custom_threshold).fit(pred_proba_c1)
        custom_predict = binarizer.transform(pred_proba_c1)
        print('임곗값:',custom_threshold)
        get_clf_eval(y_test , custom_predict)

get_eval_by_threshold(y_test ,pred_proba[:,1].reshape(-1,1), thresholds )
```

• 정밀도와 재현율

> 여러 개의 임곗값 조정하기

Out :

```
임곗값: 0.4
오차 행렬
[[86 24]
 [17 52]]
정확도: 0.7709, 정밀도: 0.6842, 재현율: 0.7536
임곗값: 0.45
오차 행렬
[[88 22]
 [20 49]]
정확도: 0.7654, 정밀도: 0.6901, 재현율: 0.7101
임곗값: 0.5
오차 행렬
[[92 18]
 [20 49]]
정확도: 0.7877, 정밀도: 0.7313, 재현율: 0.7101
임곗값: 0.55
오차 행렬
[[98 12]
 [23 46]]
정확도: 0.8045, 정밀도: 0.7931, 재현율: 0.6667
임곗값: 0.6
오차 행렬
[[101   9]
 [ 26  43]]
정확도: 0.8045, 정밀도: 0.8269, 재현율: 0.6232
```

• 정밀도와 재현율

> precision_recall_curve

```
from sklearn.metrics import precision_recall_curve


pred_proba_class1 = lr_clf.predict_proba(X_test)[:, 1]
precisions, recalls, thresholds = precision_recall_curve(y_test, pred_proba_class1 )
print('반환된 분류 결정 임곗값 배열의 Shape:', thresholds.shape)
print('반환된 precisions 배열의 Shape:', precisions.shape)
print('반환된 recalls 배열의 Shape:', recalls.shape)
print("thresholds 5 sample:", thresholds[:5])
print("precisions 5 sample:", precisions[:5])
print("recalls 5 sample:", recalls[:5])
```

• 정밀도와 재현율

> precision_recall_curve

Out :
```
반환된 분류 결정 임곗값 배열의 Shape: (172,)
반환된 precisions 배열의 Shape: (173,)
반환된 recalls 배열의 Shape: (173,)
thresholds 5 sample: [0.04310391 0.0572492  0.06058916 0.06971505 0.07001447]
precisions 5 sample: [0.38547486 0.38764045 0.38983051 0.39204545 0.39428571]
recalls 5 sample: [1. 1. 1. 1. 1.]
```

• 정밀도와 재현율

> precision_recall_curve

```
thr_index = np.arange(0, thresholds.shape[0], 15)
print('샘플 추출을 위한 임계값 배열의 index 10개:', thr_index)
print('샘플용 10개의 임곗값: ', np.round(thresholds[thr_index],2))


print('샘플 임계값별 정밀도: ', np.round(precisions[thr_index], 3))
print('샘플 임계값별 재현율: ', np.round(recalls[thr_index], 3))
```

Out :

```
샘플 추출을 위한 임계값 배열의 index 10개: [  0  15  30  45  60  75  90 105 120 135 150 165]
샘플용 10개의 임곗값:  [0.04 0.1  0.13 0.15 0.18 0.26 0.36 0.5  0.61 0.71 0.84 0.92]
샘플 임계값별 정밀도:  [0.385 0.415 0.455 0.504 0.562 0.619 0.695 0.731 0.827 0.919 1.    1.   ]
샘플 임계값별 재현율:  [1.    0.986 0.957 0.942 0.913 0.87  0.826 0.71  0.623 0.493 0.319 0.101]
```

• 정밀도와 재현율

> precision_recall_curve를 사용한 정밀도, 재현율 곡선

```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

def precision_recall_curve_plot(y_test , pred_proba_c1):
    precisions, recalls, thresholds = precision_recall_curve(y_test, pred_proba_c1)
    plt.figure(figsize=(8,6))
    threshold_boundary = thresholds.shape[0]
    plt.plot(thresholds, precisions[0:threshold_boundary], '--', label='precision')
    plt.plot(thresholds, recalls[0:threshold_boundary], label='recall')
```

• 정밀도와 재현율

> precision_recall_curve를 사용한 정밀도, 재현율 곡선

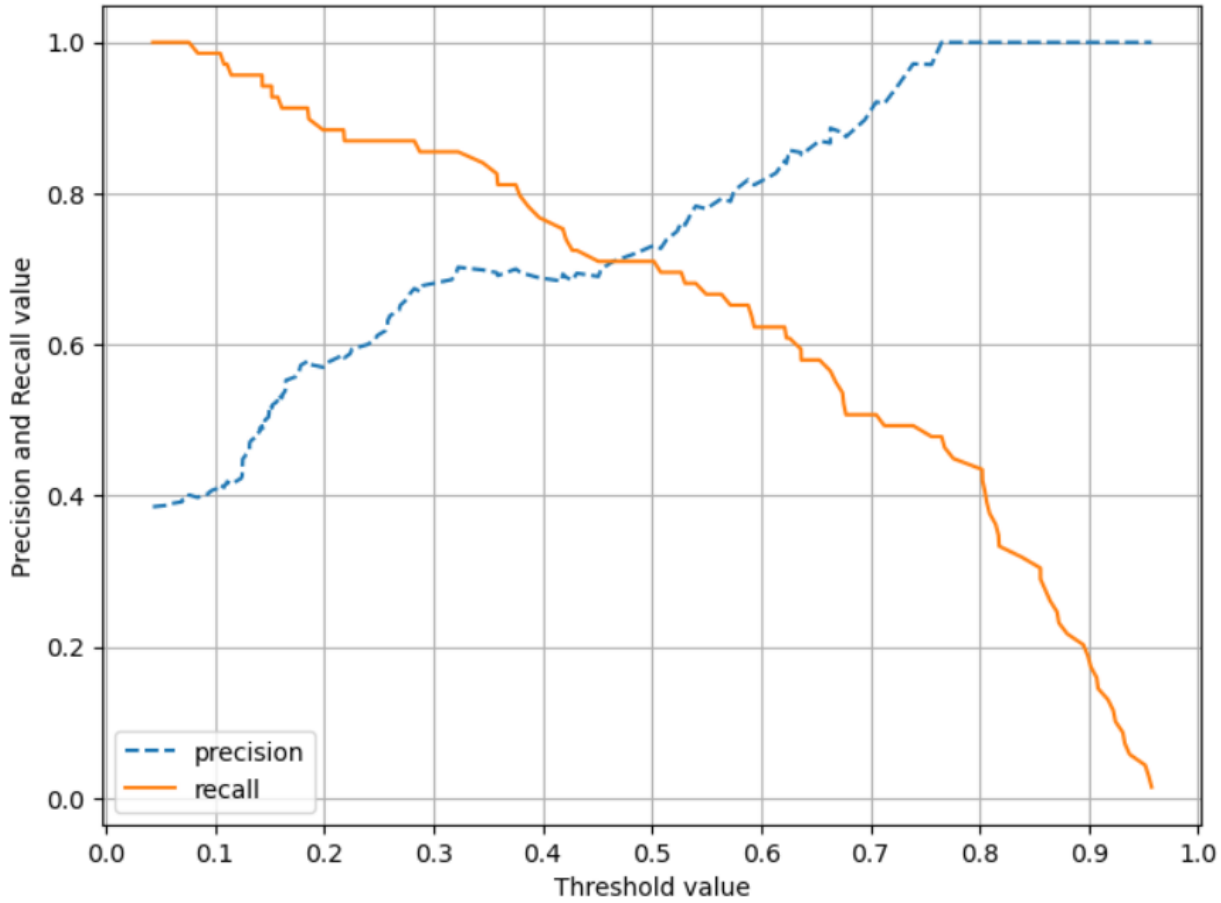
```
    # 이전 코드에 이어서 작성
    start, end = plt.xlim()
    plt.xticks(np.round(np.arange(start, end, 0.1),2))
    plt.xlabel('Threshold value')
    plt.ylabel('Precision and Recall value')
    plt.legend()
    plt.grid()
    plt.show()

precision_recall_curve_plot(y_test, lr_clf.predict_proba(X_test)[:, 1] )
```

• 정밀도와 재현율

> precision_recall_curve를 사용한 정밀도, 재현율 곡선

Out :



• F1 스코어

> F1 스코어는 정밀도와 재현율을 결합한 지표이다.

$$F1 = \frac{2}{\frac{1}{recall} + \frac{1}{precision}} = 2 * \frac{precision * recall}{precision + recall}$$

> 정밀도, 재현율 어느쪽으로도 치우치지 않을 때, 상대적으로 높은 값을 가진다.

```
from sklearn.metrics import f1_score
f1 = f1_score(y_test, pred)
print(f'F1 스코어 : {f1:.4f}')
```

Out : F1 스코어 : 0.7206

• F1 스코어

> get_eval_by_threshold

```
# F1 스코어 추가
def get_clf_eval(y_test , pred):
    confusion = confusion_matrix( y_test, pred)
    accuracy = accuracy_score(y_test , pred)
    precision = precision_score(y_test , pred)
    recall = recall_score(y_test , pred)
    f1 = f1_score(y_test,pred)
    print('오차 행렬')
    print(confusion)
    print(f'정확도: {accuracy:.4f}, 정밀도: {precision:.4f}, 재현율: {recall:.4f}, F1:{f1:.4f}')
```

• F1 스코어

> get_eval_by_threshold

```
thresholds = [0.4 , 0.45 , 0.50 , 0.55 , 0.60]
pred_proba = lr_clf.predict_proba(X_test)
get_eval_by_threshold(y_test, pred_proba[:,1].reshape(-1,1), thresholds)
```

Out :

```
임곗값: 0.4
오차 행렬
[[86 24]
 [17 52]]
정확도: 0.7709, 정밀도: 0.6842, 재현율: 0.7536, F1:0.7172
임곗값: 0.45
오차 행렬
[[88 22]
 [20 49]]
정확도: 0.7654, 정밀도: 0.6901, 재현율: 0.7101, F1:0.7000
임곗값: 0.5
오차 행렬
[[92 18]
 [20 49]]
정확도: 0.7877, 정밀도: 0.7313, 재현율: 0.7101, F1:0.7206
임곗값: 0.55
오차 행렬
[[98 12]
 [23 46]]
정확도: 0.8045, 정밀도: 0.7931, 재현율: 0.6667, F1:0.7244
임곗값: 0.6
오차 행렬
[[101   9]
 [ 26  43]]
정확도: 0.8045, 정밀도: 0.8269, 재현율: 0.6232, F1:0.7107
```

• ROC 곡선과 AUC

> ROC 곡선과 이에 기반한 AUC 스코어는 이진 분류의 예측 성능 측정에서 중요하게 사용되는 지표이다.

> ROC 곡선(Receiver Opertaion Characteristic Curve)은 수신자 판단 곡선으로 불린다.

> 임계값에 따라 FPR(False Positive Rate)이 변할 때, TPR(True Positive Rate)이 어떻게 변하는지를 나타내는 곡선이다.

> 임계값을 바꿔도 전반적인 성능이 좋고 나쁘고를 비교하고 싶을 때 사용한다.

$$TPR = \frac{TP}{(FN + TP)} = \text{재현율}$$

$$FPR = \frac{FP}{(FP + TN)} = 1 - \text{TNR} = 1 - \frac{TN}{(FP + TN)}$$

• ROC 곡선과 AUC

> roc_curve

```
from sklearn.metrics import roc_curve


pred_proba_class1 = lr_clf.predict_proba(X_test)[:, 1]
fprs , tprs , thresholds = roc_curve(y_test, pred_proba_class1)
thr_index = np.arange(1, thresholds.shape[0], 5)
print('샘플 추출을 위한 임곗값 배열의 index:', thr_index)
print('샘플 index로 추출한 임곗값: ', np.round(thresholds[thr_index], 2))
print('샘플 임곗값별 FPR: ', np.round(fprs[thr_index], 3))
print('샘플 임곗값별 TPR: ', np.round(tprs[thr_index], 3))
```

Out :
```
샘플 추출을 위한 임곗값 배열의 index: [ 1  6 11 16 21 26 31 36 41 46 51 56]
샘플 index로 추출한 임곗값:  [0.96 0.71 0.64 0.59 0.53 0.43 0.36 0.22 0.16 0.14 0.13 0.08]
샘플 임곗값별 FPR:  [0.    0.027 0.064 0.091 0.136 0.2   0.227 0.391 0.509 0.618 0.791 0.936]
샘플 임곗값별 TPR:  [0.014 0.507 0.58  0.652 0.681 0.725 0.812 0.884 0.913 0.942 0.957 0.986]
```

• ROC 곡선과 AUC

> roc_curve

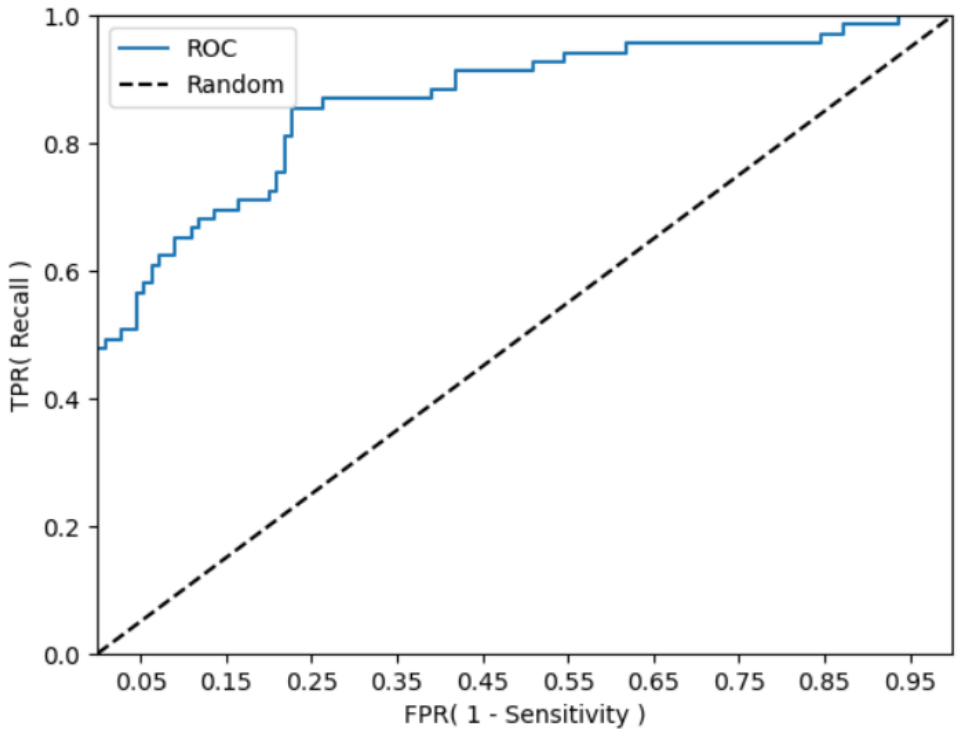
```
def roc_curve_plot(y_test , pred_proba_c1):
    fprs , tprs , thresholds = roc_curve(y_test ,pred_proba_c1)
    plt.plot(fprs , tprs, label='ROC')
    plt.plot([0, 1], [0, 1], 'k--', label='Random')
    start, end = plt.xlim()
    plt.xticks(np.round(np.arange(start, end, 0.1),2))
    plt.xlim(0,1); plt.ylim(0,1)
    plt.xlabel('FPR( 1 - Sensitivity )')
    plt.ylabel('TPR( Recall )')
    plt.legend()
    plt.show()
```

• ROC 곡선과 AUC

> roc_curve

```
roc_curve_plot(y_test, lr_clf.predict_proba(X_test)[:, 1] )
```

Out :



• ROC 곡선과 AUC

> roc_curve

```
from sklearn.metrics import roc_auc_score
pred_proba = lr_clf.predict_proba(X_test)[:, 1]
roc_score = roc_auc_score(y_test, pred_proba)
print(f'ROC AUC 값: {roc_score:.4f}')
```

Out : ROC AUC 값: 0.8671

• ROC 곡선과 AUC

> roc_curve

```
# roc_auc 추가
def get_clf_eval(y_test, pred=None, pred_proba=None):
    confusion = confusion_matrix( y_test, pred)
    accuracy = accuracy_score(y_test , pred)
    precision = precision_score(y_test , pred)
    recall = recall_score(y_test , pred)
    f1 = f1_score(y_test,pred)
    roc_auc = roc_auc_score(y_test, pred_proba)
    print('오차 행렬')
    print(confusion)
    print(f'''정확도: {accuracy:.4f}, 정밀도: {precision:.4f}, 재현율: {recall:.4f},
    F1: {f1:.4f}, AUC:{roc_auc:.4f}''')
```

• ROC 곡선과 AUC

> roc_curve

```
get_clf_eval(y_test, pred, pred_proba )
```

Out : 오차 행렬
[[92 18]
 [20 49]]
정확도: 0.7877, 정밀도: 0.7313, 재현율: 0.7101,
F1: 0.7206, AUC:0.8671

• 피마 인디언 당뇨병 예측

> **피쳐 구성**
  - Pregnancies : 임신 횟수
  - Glucose : 포도당 부하 검사 수치
  - BloodPressure : 혈압
  - SkinThickness : 팔 삼두근 뒤쪽의 피하지방 측정값
  - Insulin : 혈청 인슐린
  - BMI : 체질량 지수
  - DiabetesPedigreeFunction : 당뇨 내력 가중치 값
  - Age : 나이
  - Outcome : 클래스 결정 값(0 또는 1)

• 피마 인디언 당뇨병 예측

> 필요 라이브러리 호출

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score, roc_auc_score
from sklearn.metrics import f1_score, confusion_matrix, precision_recall_curve, roc_curve
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
```

• 피마 인디언 당뇨병 예측

> 데이터 불러오기

```
diabetes_data = pd.read_csv('diabetes.csv')
print(diabetes_data['Outcome'].value_counts())
diabetes_data.head(3)
```

Out :

```
0    500
1    268
Name: Outcome, dtype: int64
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1

• 피마 인디언 당뇨병 예측

> 데이터 살펴보기

```
diabetes_data.info()
```

Out :

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               768 non-null    int64
 1   Glucose                   768 non-null    int64
 2   BloodPressure             768 non-null    int64
 3   SkinThickness             768 non-null    int64
 4   Insulin                   768 non-null    int64
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64
 8   Outcome                   768 non-null    int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

• 피마 인디언 당뇨병 예측

> 데이터 학습/예측/평가

```
X = diabetes_data.iloc[:, :-1]
y = diabetes_data.iloc[:, -1]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
                                                    random_state = 156, stratify=y)
lr_clf = LogisticRegression(solver='liblinear')
lr_clf.fit(X_train , y_train)
pred = lr_clf.predict(X_test)
pred_proba = lr_clf.predict_proba(X_test)[:, 1]
get_clf_eval(y_test , pred, pred_proba)
```

Out :

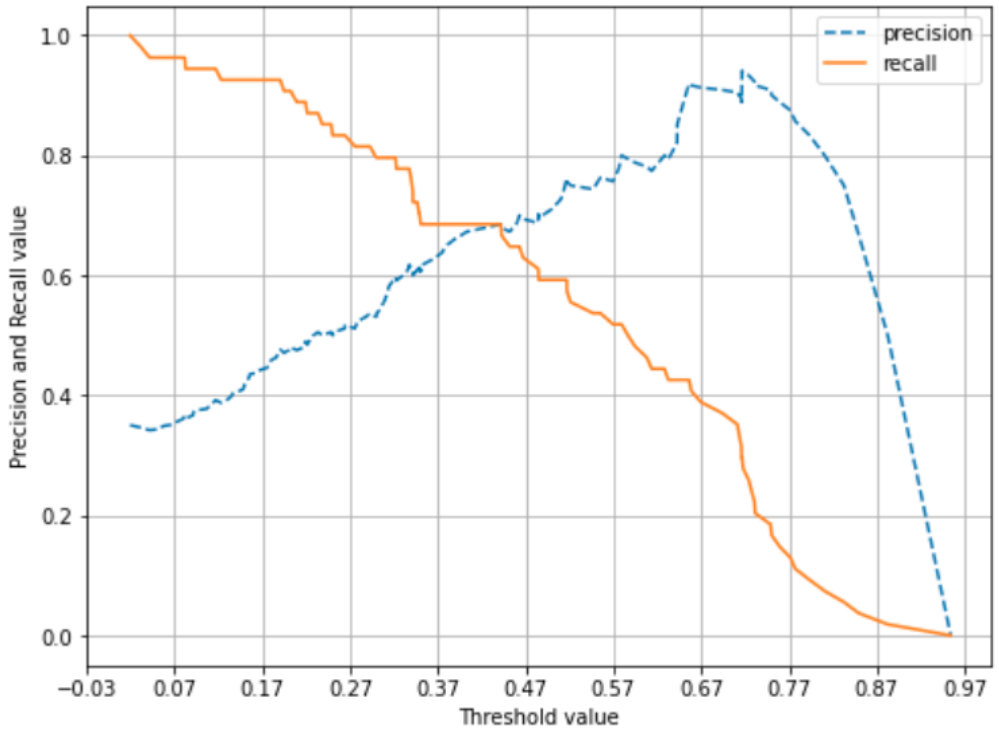
```
오차 행렬
[[87 13]
 [22 32]]
정확도: 0.7727, 정밀도: 0.7111, 재현율: 0.5926, F1: 0.6465, AUC:0.8083
```

- 피마 인디언 당뇨병 예측

> 데이터 학습/예측/평가

```
pred_proba_c1 = lr_clf.predict_proba(X_test)[:, 1]
precision_recall_curve_plot(y_test, pred_proba_c1)
```

Out :



머신러닝 분류 평가

- 피마 인디언 당뇨병 예측

> 피쳐 다시 살피기

```
diabetes_data.describe()
```

Out :

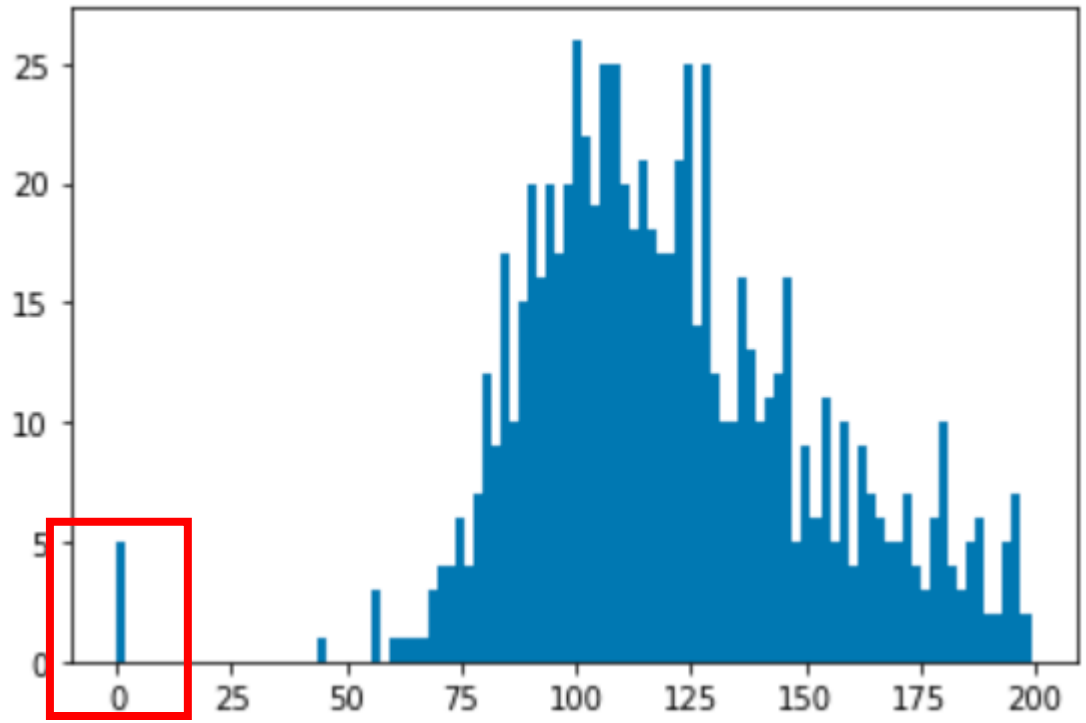
	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

• 피마 인디언 당뇨병 예측

> 피쳐 다시 살피기

```
plt.hist(diabetes_data['Glucose'], bins = 100)
```

Out :



• 피마 인디언 당뇨병 예측

> 피쳐 다시 살피기

```
zero_features = ['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin','BMI']


total_count = diabetes_data['Glucose'].count()
for feature in zero_features:
    zero_count = diabetes_data[ diabetes_data [feature] == 0][feature].count()
    print(f'{feature} 0 건수는 {zero_count}, 퍼센트는 {100*zero_count/total_count:.2f} %')
```

Out :

```
Glucose 0 건수는 5, 퍼센트는 0.65 %
BloodPressure 0 건수는 35, 퍼센트는 4.56 %
SkinThickness 0 건수는 227, 퍼센트는 29.56 %
Insulin 0 건수는 374, 퍼센트는 48.70 %
BMI 0 건수는 11, 퍼센트는 1.43 %
```

• 피마 인디언 당뇨병 예측

> 피쳐 다시 살피기

```
mean_zero_features = diabetes_data[zero_features].mean()
diabetes_data[zero_features]=diabetes_data[zero_features].replace(0, mean_zero_features)
diabetes_data.describe()
```

Out :

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	121.681605	72.254807	26.606479	118.660163	32.450805	0.471876	33.240885	0.348958
std	3.369578	30.436016	12.115932	9.631241	93.080358	6.875374	0.331329	11.760232	0.476951
min	0.000000	44.000000	24.000000	7.000000	14.000000	18.200000	0.078000	21.000000	0.000000
25%	1.000000	99.750000	64.000000	20.536458	79.799479	27.500000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	79.799479	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

• 피마 인디언 당뇨병 예측

> 다시 학습/예측/평가

```
X = diabetes_data.iloc[:, :-1]
y = diabetes_data.iloc[:, -1]
scaler = StandardScaler( )
X_scaled = scaler.fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size = 0.2,
                                                    random_state = 156, stratify=y)
lr_clf = LogisticRegression(solver='liblinear')
lr_clf.fit(X_train , y_train)
pred = lr_clf.predict(X_test)
pred_proba = lr_clf.predict_proba(X_test)[:, 1]
get_clf_eval(y_test , pred, pred_proba)
```

Out :

```
오차 행렬
[[90 10]
 [21 33]]
정확도: 0.7987, 정밀도: 0.7674, 재현율: 0.6111,
    F1: 0.6804, AUC:0.8433
```

• 피마 인디언 당뇨병 예측

> 임곗값 변화시키며 성능 평가

```
from sklearn.preprocessing import Binarizer
# 함수 생성
def get_eval_by_threshold(y_test ,pred_proba_c1,thresholds):
    for custom_threshold in thresholds:
        binarizer = Binarizer(threshold = custom_threshold ).fit(pred_proba_c1)
        custom_predict = binarizer.transform(pred_proba_c1)
        print('임곗값:',custom_threshold)
        get_clf_eval(y_test , custom_predict, pred_proba_c1)
```

머신러닝 분류 평가

- 피마 인디언 당뇨병 예측

> 임곗값 변화시키며 성능 평가

```
thresholds = [0.3 , 0.33 ,0.36,0.39, 0.42 , 0.45 ,0.48, 0.50]
pred_proba = lr_clf.predict_proba(X_test)
get_eval_by_threshold(y_test, pred_proba[:,1].reshape(-1,1), thresholds )
```

Out :

```
임곗값: 0.3
오차 행렬
[[65 35]
 [11 43]]
정확도: 0.7013, 정밀도: 0.5513, 재현율: 0.7963,
    F1: 0.6515, AUC:0.8433
임곗값: 0.33
오차 행렬
[[71 29]
 [11 43]]
정확도: 0.7403, 정밀도: 0.5972, 재현율: 0.7963,
    F1: 0.6825, AUC:0.8433
임곗값: 0.36
오차 행렬
[[76 24]
 [15 39]]
정확도: 0.7468, 정밀도: 0.6190, 재현율: 0.7222,
    F1: 0.6667, AUC:0.8433
임곗값: 0.39
오차 행렬
[[78 22]
 [16 38]]
정확도: 0.7532, 정밀도: 0.6333, 재현율: 0.7037,
    F1: 0.6667, AUC:0.8433
임곗값: 0.42
오차 행렬
[[84 16]
 [18 36]]
정확도: 0.7792, 정밀도: 0.6923, 재현율: 0.6667,
    F1: 0.6792, AUC:0.8433
임곗값: 0.45
오차 행렬
[[85 15]
 [18 36]]
정확도: 0.7857, 정밀도: 0.7059, 재현율: 0.6667,
    F1: 0.6857, AUC:0.8433
임곗값: 0.48
오차 행렬
[[88 12]
 [19 35]]
정확도: 0.7987, 정밀도: 0.7447, 재현율: 0.6481,
    F1: 0.6931, AUC:0.8433
임곗값: 0.5
오차 행렬
[[90 10]
 [21 33]]
정확도: 0.7987, 정밀도: 0.7674, 재현율: 0.6111,
    F1: 0.6804, AUC:0.8433
```

- 피마 인디언 당뇨병 예측

> 임곗값 변화시키며 성능 평가

```
binarizer = Binarizer(threshold=0.48)
pred_th_048 = binarizer.fit_transform(pred_proba[:, 1].reshape(-1,1))


get_clf_eval(y_test , pred_th_048, pred_proba[:, 1])
```

Out :

```
오차 행렬
[[88 12]
 [19 35]]
정확도: 0.7987, 정밀도: 0.7447, 재현율: 0.6481,
    F1: 0.6931, AUC:0.8433
```