1️⃣

week1_3장:평가

머신러닝은 데이터 가공/변환, 모델 학습/예측, 그리고 평가의 프로세스로 구성됨.

모델의 예측 성능을 평가하는 **성능 평가 지표(Evaluation Metric)**는 일반적으로 모델이 분류나 회귀냐에 따라 여러 종류로 나뉨.

- 회귀 → 예측값의 오차 평균값
- 분류 → 실제 결과 데이터와 예측 결과 데이터가 얼마나 정확하고 오류가 적게 발생하는지 + a
    - 정확도(Accuracy)
    - 오차행렬(Confusion Matrix)
    - 정밀도(Precision)
    - 재현율(Recall)
    - F1 스코어
    - ROC AUC

    ⇒ 분류는 결정 클래스 값 종류에 따라 긍정/부정과 같은 두개의 결괏값만을 가지는 **이진 분류**와 여러 개의 결정 클래스 값을 가지는 **멀티 분류**로 나뉨. 위의 지표는 모두에 적용되지만, 특히 이진 분류에서 더욱 중요하게 강조되는 지표임.

01. 정확도(Accuracy)

:실제 데이터에서 예측 데이터가 얼마나 같은지를 판단하는 지표

> 💡 정확도 = 예측 결과가 동일한 데이터 건수 / 전체 예측 데이터 건수

- 직관적으로 모델 예측 성능을 나타내는 평가 지표
- 하지만, **이진 분류의 경우 정확도 지표가 모델 성능을 왜곡**할 수 있음
    - 타이타닉 예제 정확도 80%. 탑승객이 여자일때 생존률이 높아서, 별다른 알고리즘의 적용 없이 무조건 여자-생존 남자-사망으로 예측해도 비슷한 수치가 나올 수 있음.

      → 단지 성별 조건 하나만을 가지고 결정하는 별거 아닌 알고리즘도 높은 정확도를 나타내는 상황 발생.

⇒ **정확도**는 **불균형한(imbalanced) 레이블 값 분포**에서 ML 모델의 성능을 판단할 경우, 적합한 평가 지표가 아님.

MNIST 데이터 세트를 변환해 불균형한 데이터 세트로 만든 뒤에 정확도 지표 적용시 발생할 문제를 보겠음.
(MNIST는 0~9까지의 숫자 이미지의 픽셀 정보 가지고 잇으며, 이를 기반으로 숫자 Digit를 예측하는데 사용됨. 원래 MNIST 데이터셋은 레이블 값이 0~9까지 있는 멀티 레이블 분류를 위한 것인데,)
이걸 레이블 값이 7인 것만 True, 나머지 값들은 모두 False로 변환해 이진 분류 문제로 바꿔보겠음. (즉, 전체 데이터의 10%만 T, 90%는 F인 불균형한 데이터셋으로 변형) : multi classification → binary classification

이렇게 불균형한 데이터셋에 모든 데이터를 F(0)으로 예측하는 Classifier를 이용해 정확도 측정하면 90%가 나옴. ⇒ 아무것도 하지 않고 무조건 특정한 결과로 찍어도 데이터 분포도가 균일하지 않은 경우 **높은 수치가 나타날 수 있다는 것**이 정확도 평가 지표의 맹점!

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.base import BaseEstimator
from sklearn.metrics import accuracy_score
import numpy as np
import pandas as pd

class MyFakeClassifier(BaseEstimator):
    def fit(self,X,y):
```

```
        pass

    # 입력값으로 들어오는 X 데이터 셋의 크기만큼 모두 0값으로 만들어서 반환
    def predict(self,X):
        return np.zeros( (len(X), 1) , dtype=bool)

# 사이킷런의 내장 데이터 셋인 load_digits( )를 이용하여 MNIST 데이터 로딩
digits = load_digits()

print(digits.data)
print("### digits.data.shape:", digits.data.shape)
print(digits.target)
print("### digits.target.shape:", digits.target.shape)
'''
[[ 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.]]
### digits.data.shape: (1797, 64)
[0 1 2 ... 8 9 8]
### digits.target.shape: (1797,)
'''
```

```
digits.target == 7  # array([False, False, False, ..., False, False, False])
```

```
# digits번호가 7번이면 True이고 이를 astype(int)로 1로 변환, 7번이 아니면 False이고 0으로 변환.
y = (digits.target == 7).astype(int)
X_train, X_test, y_train, y_test = train_test_split( digits.data, y, random_state=11)
```

```
# 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())

# Dummy Classifier로 학습/예측/정확도 평가
fakeclf = MyFakeClassifier()
fakeclf.fit(X_train , y_train)
fakepred = fakeclf.predict(X_test)
print('모든 예측을 0으로 하여도 정확도는:{:.3f}'.format(accuracy_score(y_test , fakepred)))
'''
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
'''
```

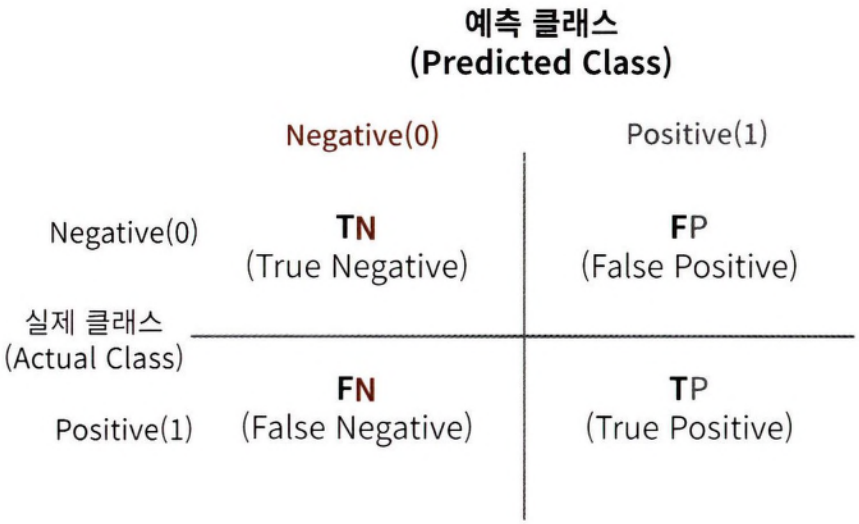
→ 단순히 predict( )의 결과를 np.zeros( )로 모두 0값으로 반환함에도 불구하고 예측 정확도가 90%가 나옴. → 말도 안되는 결과...

⇒ 정확도 평가 지표는 불균형한 레이블 데이터 세트에서는 성능 수치로 사용되어서 안됨!

⇒ 정확도가 가지는 분류 평가 지표로서의 한계점을 극복하기 위해 여러 가지 분류 지표와 함께 사용해야함.

02. 오차 행렬(Confusion Matrix)

: 이진분류에서 성능 지표로 잘 활용되는 오차행렬은 학습된 분류 모델이 예측을 수행하면서 얼마나 헷갈리고 있는지도 함께 보여주는 지표임. ⇒ 이진 분류의 **예측 오류가 얼마**인지 + **어떤 유형의 예측 오류** 발생하는지를 함께 나타냄!



출처: https://iphoong.tistory.com/7

4분면 행렬에서 실제 레이블 클래스 값과 예측 레이블 클래스 값이 어떠한 유형을 가지고 매핑되는지를 나타냄.

TN, FP, FN, TP는 예측 클래스와 실제 클래스의 Positive 결정 값(값1)과 Negative 결정값(값0)의 결합에 따라 결정됨.

- TN: True Negative, 앞 True는 예측 클래스 값과 실제 클래스 값이 같다는 의미. 뒤의 Negative는 예측값이 Negative 값이라는 의미. 즉, N값 0으로 예측했는데, 실제 값도 N값 0이라는 의미.
- 앞 문자 True/False는 예측값과 실제값이 '같은가/틀린가'를 의미하고, 뒤의 Negative/Positive는 예측 결과 값이 부정(0)/긍정(1)을 의미함

사이킷런은 오차행렬을 위한 API `confusion_matrix()` 를 제공함. 앞에서 만든, MyFakeClassifier의 예측 성능 지표를 오차 행렬로 표현하고, 이 모델의 예측 결과인 fakepred와 실제 결과인 y_test를 `confusion_matrix()`의 인자로 입력해 오차 행렬을 구해보겠음.

```
from sklearn.metrics import confusion_matrix

# 예측 결과인 fakepred와 실제 결과인 y_test의 Confusion Matrix출력
confusion_matrix(y_test , fakepred)
'''
array([[405,   0],
       [ 45,   0]])
'''
```

⇒ 오차 행렬은 ndarray 형태로 출력, 이진 분류의 TN, FP, FN, TP는 상단 도표와 동일한 위치를 가지고 array에서 가져올 수 있음.

TN, FP, FN, TP 값을 조합 → 정확도(Accuracy), 정밀도(Precision), 재현율(Recall) 값 알 수 있음.
정확도는, 예측값과 실제값이 얼마나 동일한가에 대한 비율로만 결정됨. 즉, 오차 행렬에서 True에 해당하는 값인 TN과 TP에 좌우됨.

💡 정확도 = 예측 결과와 실제 값이 동일한 건수 / 전체 데이터 수
= (TN+TP) / (TN + TP + FN + FP)

일반적으로 불균형한 레이블 클래스를 가지는 이진 분류 모델에서는 많은 데이터 중 중점적으로 찾아야 하는 매우 적은 수의 결괏값에 Positive를 설정해 1 부여, 그렇지 않은 경우 Negative로 0을 부여함. → 이 경우, P보다 N으로 예측 정확도가 높아지는 경향 발생(= 수치적인 판단 오류 발생)

03. 정밀도(Precision)와 재현율(Recall)

불균형한 데이터 세트에서 정확도보다 더 선호되는 평가 지표, Positive 데이터 예측 성능에 좀 더 초점을 맞춘 평가 지표.

> 💡 **정밀도** = TP / (FP + TP) - 양성이라고 나온 것 중 진짜 양성 확률
> **재현율** = TP / (FN + TP) - 진짜 양성 중 양성이라고 나온 것

→ (공통) TP(예측과 실제 모두 p) 높이는데 초점 (+ 정밀도: FP 낮추는데 초점, 재현율: FN 낮추는데 초점 ⇒ 정밀도와 재현율은 서로 보완적임)

- **정밀도:** 예측을 P라고 한 데이터 중, 예측과 실제 모두 P인 데이터 비율 → P 예측 성능을 더욱 정밀하게 측정하기 위한 평가 지표, 양성 예측도라고도 부름
  (보통은 재현율이 정밀도보다 중요한 경우 많음) 정밀도 중요 지표인 경우: 실제 음성 데이터를 양성으로 잘못 판단하게 되면 업무상 큰 영향이 발생하는 경우(스팸 메일)
- **재현율:** 실제 값이 P인 데이터 중, 예측과 실제 모두 P인 데이터 비율 → **민감도 (Sensitivity)** 또는 TPR(True Positive Rate)라고도 부름
  재현율이 중요 지표인 경우: 실제 양성 데이터를 음성으로 잘못 판단하게 되면 업무상 큰 영향이 발생하는 경우(암 판단 모델)

```
# 오차행렬, 정확도, 정밀도, 재현율 한꺼번에 계산하는 함수 생성
from sklearn.metrics import accuracy_score, precision_score, recall_score, confusion_matri
x

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('정확도: {0:.4f}, 정밀도: {1:.4f}, 재현율: {2:.4f}'.format(accuracy , precision ,recal
l))
```

```
# 로지스틱 회귀 기반으로 타이타닉 생존자 예측하고 평가 수행

import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

```
# 원본 데이터 재로딩, 데이터 가공, 학습/테스트 데이터 분할
titanic_df = pd.read_csv('titanic_train.csv')
y_titanic_df = titanic_df['Survived']
X_titanic_df= titanic_df.drop('Survived', axis=1)
X_titanic_df = transform_features(X_titanic_df)

X_train, X_test, y_train, y_test = train_test_split(X_titanic_df, y_titanic_df, \
                                                    test_size=0.20, random_state=11)

lr_clf = LogisticRegression()

lr_clf.fit(X_train , y_train)
pred = lr_clf.predict(X_test)
get_clf_eval(y_test , pred)  # 평가 한꺼번에 수행하는 함수
'''
오차 행렬
[[104  14]
 [ 13  48]]
정확도: 0.8492, 정밀도: 0.7742, 재현율: 0.7869
'''
```

정밀도/재현율 트레이드오프

분류 결정 임곗값(Threshold)을 조정해 정밀도 또는 재현율의 수치를 조절할 수 있음. 하지만 정밀도와 재현율은 상호보완적 관계라 한 쪽을 높이면 한쪽이 떨어지게 됨 ⇒ **정밀도/재현율 트레이드오프**

사이킷런 분류 알고리즘은 예측 데이터가 특정 레이블에 속하는지를 계산하기 위해, 먼저 개별 레이블별로 결정 확률을 구함. 그리고 예측 확률이 큰 레이블값으로 예측하게 됨. 일반적으로 이진 분류에서는 임계값을 0.5로 설정하고, 이 기준값보다 확률이 크면 P, 작으면 N으로 결정함.

`predict_proba()`: **개별 데이터별로 예측 확률 반환.** 학습이 완료된 사이킷런 Classifier 객체에서 호출 가능하며, 테스트 피처 데이터 세트를 파라미터로 입력해주면 테스트 피처 레코드의 개별 클래스 예측 확률을 반환함. (`predict()` 와 유사하지만, 예측 결과 **클래스**가 아닌 **예측 확률** 결과가 출력됨)

- 입력 파라미터: 테스트 피처 데이터 세트 입력
- 반환 값: 개별 클래스의 **예측 확률을 ndarray** 형태로 반환. (m입력값의 레코드 수 * n 클래스 값 유형)

```
# predict_proba와 predict() 차이

# predict_proba() -> 각각 두 개의 클래스 나올 확률(이진분류니까 클래스 두개, 합은 1)
pred_proba = lr_clf.predict_proba(X_test)
'''
```

```
[[0.46191519 0.53808481]
 [0.878675   0.121325  ]
 [0.87716185 0.12283815]
...
[0.8589115  0.1410885 ]
 [0.45503082 0.54496918]
 [0.37286468 0.62713532]]
'''
# predict() -> 확률 값이 아닌 예측 결과 클래스가 나옴(0인지 1인지) => 두 개의 칼럼 중 더 큰 확률 값으로 최종 예측
pred = lr_clf.predict(X_test)
'''
[1 0 0 0 0 0 ... 0 1 1] '''

print('pred_proba()결과 Shape : {0}'.format(pred_proba.shape))  # pred_proba()결과 Shape : (179, 2)
print('pred_proba array에서 앞 3개만 샘플로 추출 \n:', pred_proba[:3])
'''
 pred_proba array에서 앞 3개만 샘플로 추출
: [[0.46191519 0.53808481]
 [0.878675   0.121325  ]
 [0.87716185 0.12283815]]
'''
```

분류 결정 임계값을 조절해 정밀도와 재현율의 성능 수치를 상호 보완적으로 보완할 수 있음.

사이킷런의 predict( )는, predict_proba( ) 메서드가 반환하는 확률 값을 가진 ndarray에서 정해진 임계값을 만족하는 ndarray의 칼럼 위치를 최종 클래스로 결정함.

Binarizer 클래스를 이용할건데, Binarizer 클래스 객체로 생성하고, 입력된 ndarray의 값이 지정된 임계값 보다 같거나 작으면 0으로, 크면 1로 반환.

```
from sklearn.preprocessing import Binarizer

X = [[ 1, -1,  2],
     [ 2,  0,  0],
     [ 0,  1.1, 1.2]]

# threshold 기준값(여기서는 1.1)보다 같거나 작으면 0을, 크면 1을 반환
binarizer = Binarizer(threshold=1.1)
print(binarizer.fit_transform(X))
'''
[[0. 0. 1.]
 [1. 0. 0.]
 [0. 0. 1.]]
'''
```

이 Binarizer를 이용해 분류 결정 임계값을 0.5로 지정한 Binarizer 클래스를 적용해 최종 예측값을 구해보겠음.

```
from sklearn.preprocessing import Binarizer

# Binarizer의 threshold 설정값. 분류 결정 임계값임.
custom_threshold = 0.5

# predict_proba( ) 반환값의 두번째 컬럼(positive 클래스일 확률값), 즉 Positive 클래스 컬럼 하나만 추출하여 Binarizer를 적용
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)

# 구한 최종 예측값에 대해 평가 지표도 출력
get_clf_eval(y_test, custom_predict)
'''
오차 행렬
[[104  14]
 [ 13  48]]
정확도: 0.8492, 정밀도: 0.7742, 재현율: 0.7869
'''
```

→ 타이타닉 데이터로 학습된 로지스틱 회귀 분류 객체에서 호출된 predict()로 계산된 지표 값과 정확히 같음. predict()가 predict_proba()에 기반함을 알 수 있음.

분류 결정 임계값 0.4 기반에서 Binarizer를 이용하여 예측값 변환

```
# Binarizer의 threshold 설정값을 0.4로 설정. 즉 분류 결정 임곗값을 0.5에서 0.4로 낮춤
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)
'''
오차 행렬
[[98 20]
 [10 51]]
정확도: 0.8324, 정밀도: 0.7183, 재현율: 0.8361
'''
```

→ 임계값 ↓ ⇒ 재현율 ↑, 정밀도 ↓ ➡️ P 예측값이 많아지면 상대적으로 재현율 값이 높아짐. 양성 예측을 많이 하다 보니 실제 양성을 음성으로 예측하는 횟수가 상대적으로 줄어들기 때문.

여러개의 분류 결정 임곗값을 변경하면서 Binarizer를 이용하여 예측값 변환

```
# 테스트를 수행할 모든 임곗값을 리스트 객체로 저장.
thresholds = [0.4, 0.45, 0.50, 0.55, 0.60]

def get_eval_by_threshold(y_test , pred_proba_c1, thresholds):
    # thresholds list객체내의 값을 차례로 iteration하면서 Evaluation 수행.
    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 )
'''
임곗값: 0.4
오차 행렬
[[99 19]
 [10 51]]
정확도: 0.8380, 정밀도: 0.7286, 재현율: 0.8361
임곗값: 0.45
오차 행렬
[[103  15]
 [ 12  49]]
정확도: 0.8492, 정밀도: 0.7656, 재현율: 0.8033
임곗값: 0.5
오차 행렬
[[104  14]
 [ 13  48]]
정확도: 0.8492, 정밀도: 0.7742, 재현율: 0.7869
임곗값: 0.55
오차 행렬
[[109   9]
 [ 15  46]]
정확도: 0.8659, 정밀도: 0.8364, 재현율: 0.7541
임곗값: 0.6
오차 행렬
[[112   6]
 [ 16  45]]
정확도: 0.8771, 정밀도: 0.8824, 재현율: 0.7377
'''
```

⇒ 지금까지 임곗값 변화에 다른 평가 지표 값을 알아보는 코드를 작성했음. 사이킷런은 이와 유사한 `precision_recall_curve()` 제공함

- 입력 파라미터
  - y_true: 실제 클래스값 배열(배열 크기=[데이터 건수])

    - probas_pred: Positive 칼럼 예측 확률 배열
- 반환 값
    - 정밀도: 임곗값별 정밀도 값을 배열로 반환
    - 재현율: 임곗값별 재현율 값을 배열로 반환

`precision_recall_curve()` 를 이용해서 타이타닉 예측 모델의 임곗값별 정밀도와 재현율 구해보겠음 → 임계값을 담은 넘파이 ndarray와 이 임계값에 해당하는 정밀도 및 재현율 값을 담은 넘파이 ndarray 반환함

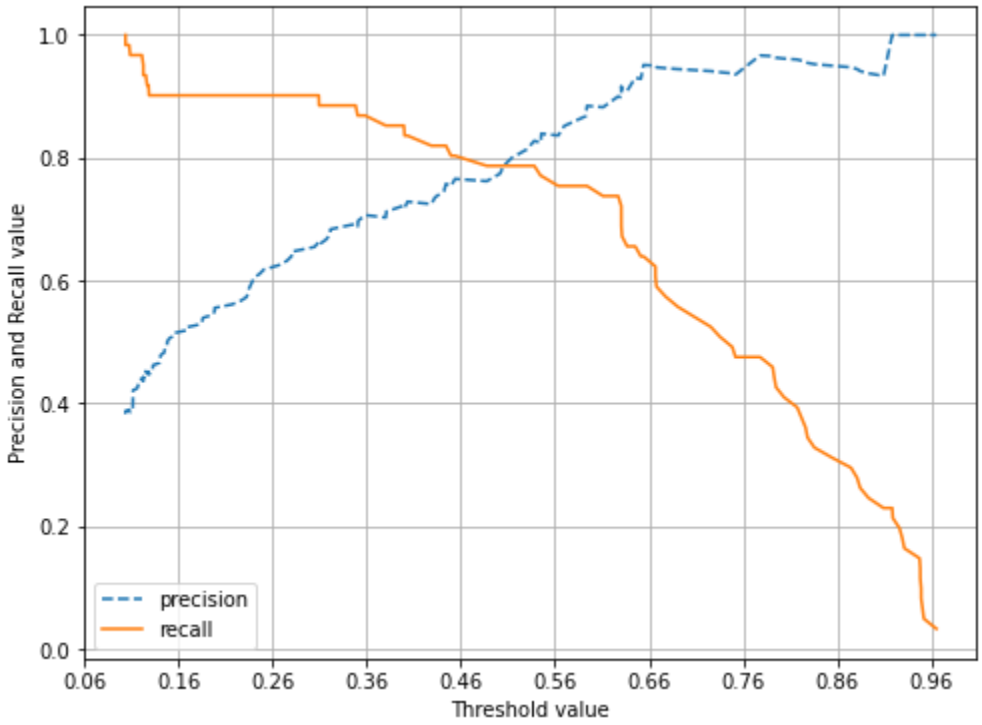
반환되는 임계값이 너무 작은 값 단위로 많이 구성되어있음(반환된 분류 결정 임곗값 배열의 Shape: (143,)) → 샘플로 10건만 추출하되, 임계값을 15단계로 추출해 좀 더 큰 값의 임곗값과 그때의 정밀도와 재현율 값 살펴봄

```
from sklearn.metrics import precision_recall_curve

# 레이블 값이 1일때의 예측 확률을 추출 - 두번쨰 칼럼 값에 해당하는 데이터 세트
pred_proba_class1 = lr_clf.predict_proba(X_test)[:, 1]

# 실제값 데이터 셋과 레이블 값이 1일 때의 예측 확률을 precision_recall_curve 인자로 입력
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)
'''
반환된 분류 결정 임곗값 배열의 Shape: (143,)
반환된 precisions 배열의 Shape: (144,)
반환된 recalls 배열의 Shape: (144,)
'''
print("thresholds 5 sample:", thresholds[:5])
print("precisions 5 sample:", precisions[:5])
print("recalls 5 sample:", recalls[:5])
'''
thresholds 5 sample: [0.10390978 0.10391203 0.1039372  0.10786305 0.10888586]
precisions 5 sample: [0.38853503 0.38461538 0.38709677 0.38961039 0.38562092]
recalls 5 sample: [1.         0.98360656 0.98360656 0.98360656 0.96721311]
'''

#반환된 임계값 배열 로우가 143건이므로 샘플로 10건만 추출하되, 임곗값을 15 Step으로 추출.
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))
'''
샘플 추출을 위한 임계값 배열의 index 10개: [  0  15  30  45  60  75  90 105 120 135]
샘플용 10개의 임곗값:  [0.1  0.12 0.14 0.19 0.28 0.4  0.56 0.67 0.82 0.95]'''

# 15 step 단위로 추출된 임계값에 따른 정밀도와 재현율 값
```

```
print('샘플 임계값별 정밀도: ', np.round(precisions[thr_index], 3))
print('샘플 임계값별 재현율: ', np.round(recalls[thr_index], 3))
'''
샘플 임계값별 정밀도:  [0.389 0.44  0.466 0.539 0.647 0.729 0.836 0.949 0.958 1.   ]
샘플 임계값별 재현율:  [1.    0.967 0.902 0.902 0.902 0.836 0.754 0.607 0.377 0.148]
'''
```

임곗값의 변경에 따른 정밀도-재현율 변화 곡선을 그림

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

def precision_recall_curve_plot(y_test , pred_proba_c1):
    # threshold ndarray와 이 threshold에 따른 정밀도, 재현율 ndarray 추출.
    precisions, recalls, thresholds = precision_recall_curve( y_test, pred_proba_c1)

    # X축을 threshold값으로, Y축은 정밀도, 재현율 값으로 각각 Plot 수행. 정밀도는 점선으로 표시
    plt.figure(figsize=(8,6))
    threshold_boundary = thresholds.shape[0]
    plt.plot(thresholds, precisions[0:threshold_boundary], linestyle='--', label='precisio
n')
    plt.plot(thresholds, recalls[0:threshold_boundary],label='recall')

    # threshold 값 X 축의 Scale을 0.1 단위로 변경
    start, end = plt.xlim()
    plt.xticks(np.round(np.arange(start, end, 0.1),2))

    # x축, y축 label과 legend, 그리고 grid 설정
    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] )
```



→ 임계값 증가할수록 재현율 값 낮아지고, 정밀도 값 높아짐. 임계값 0.45에서 둘의 값 비슷해짐.

정밀도와 재현율의 맹점

(P 예측의) 임계값을 변경함에 따라 정밀도와 재현율이 변경됨. 두 개의 수치를 상호 보완할 수 있는 수준에서 적용해야 함.

정밀도가 100%가 되는 방법

확실한 기준이 되는 경우만 P로 예측, 나머지는 모두 N으로 예측

재현율이 100%가 되는 방법

모든 데이터를 P로 예측.

⇒ 어느 한쪽만 참조하면 정밀도, 재현율 극단적인 수치로 조작 가능함. 정밀도 또는 재현율 중 하나에 상대적인 중요도를 부여해 알고리즘을 튜닝할 순 있지만, 그렇다고 하나만 강조해선 안 됨.

04. F1 스코어

: 정밀도와 재현율을 결합한 지표. 정밀도와 재현율이 어느 한쪽으로 치우치지 않는 수치를 나타낼 때 상대적으로 높은 값을 가짐.

→ `f1.score()`

```
from sklearn.metrics import f1_score
f1 = f1_score(y_test , pred)
print('F1 스코어: {0:.4f}'.format(f1))  # F1 스코어: 0.7805
```

임계값 변화시키면서 F1 스코어 포함한 평가 지표 구해보겠음

```
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 스코어 추가
    f1 = f1_score(y_test,pred)
    print('오차 행렬')
    print(confusion)
    # f1 score print 추가
    print('정확도: {0:.4f}, 정밀도: {1:.4f}, 재현율: {2:.4f}, F1:{3:.4f}'.format(accuracy, pre
cision, recall, f1))

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)

'''
임곗값: 0.4
오차 행렬
[[98 20]
 [10 51]]
정확도: 0.8324, 정밀도: 0.7183, 재현율: 0.8361, F1:0.7727
임곗값: 0.45
오차 행렬
[[103  15]
 [ 12  49]]
정확도: 0.8492, 정밀도: 0.7656, 재현율: 0.8033, F1:0.7840
임곗값: 0.5
오차 행렬
[[104  14]
 [ 13  48]]
정확도: 0.8492, 정밀도: 0.7742, 재현율: 0.7869, F1:0.7805
임곗값: 0.55
오차 행렬
[[109   9]
 [ 15  46]]
정확도: 0.8659, 정밀도: 0.8364, 재현율: 0.7541, F1:0.7931
```

```
임곗값: 0.6
오차 행렬
[[112   6]
 [ 16  45]]
정확도: 0.8771, 정밀도: 0.8824, 재현율: 0.7377, F1:0.8036
'''
```

05. ROC 곡선과 AUC

- ROC: FPR이 변할 때 TPR이 어떻게 변하는지를 나타내는 곡선(x축: FPR, y축: TPR)
    - TPR: 재현율, 민감도, 특이성(TNR)

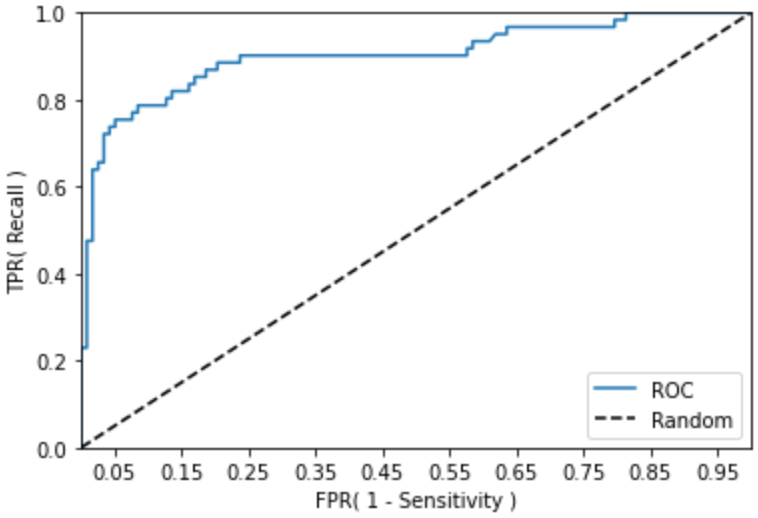
ROC 곡선의 가운데 직성은 ROC 곡선의 최저 값. 곡선이 직선에 가까울 수록 성능 떨어지고 멀수록 성능 좋음.

```
from sklearn.metrics import roc_curve

# 레이블 값이 1일때의 예측 확률을 추출
pred_proba_class1 = lr_clf.predict_proba(X_test)[:, 1]

fprs , tprs , thresholds = roc_curve(y_test, pred_proba_class1)
# 반환된 임곗값 배열에서 샘플로 데이터를 추출하되, 임곗값을 5 Step으로 추출.
# thresholds[0]은 max(예측확률)+1로 임의 설정됨. 이를 제외하기 위해 np.arange는 1부터 시작
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))

# 5 step 단위로 추출된 임계값에 따른 FPR, TPR 값
print('샘플 임곗값별 FPR: ', np.round(fprs[thr_index], 3))
print('샘플 임곗값별 TPR: ', np.round(tprs[thr_index], 3))
'''
샘플 추출을 위한 임곗값 배열의 index: [ 1  6 11 16 21 26 31 36 41 46 51]
샘플 index로 추출한 임곗값:  [0.97 0.65 0.63 0.56 0.45 0.4  0.35 0.15 0.13 0.11 0.11]
샘플 임곗값별 FPR:  [0.    0.017 0.034 0.076 0.127 0.169 0.203 0.466 0.585 0.686 0.797]
샘플 임곗값별 TPR:  [0.033 0.639 0.721 0.754 0.803 0.836 0.885 0.902 0.934 0.967 0.984]
'''
```

```
from sklearn.metrics import roc_curve

# 레이블 값이 1일때의 예측 확률을 추출
pred_proba_class1 = lr_clf.predict_proba(X_test)[:, 1]
print('max predict_proba:', np.max(pred_proba_class1))
```

```
fprs , tprs , thresholds = roc_curve(y_test, pred_proba_class1)
print('thresholds[0]:', thresholds[0])
# 반환된 임곗값 배열 로우가 47건이므로 샘플로 10건만 추출하되, 임곗값을 5 Step으로 추출.
thr_index = np.arange(0, thresholds.shape[0], 5)
print('샘플 추출을 위한 임곗값 배열의 index 10개:', thr_index)
print('샘플용 10개의 임곗값: ', np.round(thresholds[thr_index], 2))

# 5 step 단위로 추출된 임계값에 따른 FPR, TPR 값
print('샘플 임곗값별 FPR: ', np.round(fprs[thr_index], 3))
print('샘플 임곗값별 TPR: ', np.round(tprs[thr_index], 3))
```

```
def roc_curve_plot(y_test , pred_proba_c1):
    # 임곗값에 따른 FPR, TPR 값을 반환 받음.
    fprs , tprs , thresholds = roc_curve(y_test ,pred_proba_c1)

    # ROC Curve를 plot 곡선으로 그림.
    plt.plot(fprs , tprs, label='ROC')
    # 가운데 대각선 직선을 그림.
    plt.plot([0, 1], [0, 1], 'k--', label='Random')

    # FPR X 축의 Scale을 0.1 단위로 변경, X,Y 축명 설정등
    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_curve_plot(y_test, lr_clf.predict_proba(X_test)[:, 1] )
```



일반적으로 ROC 곡선 자체는 FPR과 TPR의 변화 값을 보는 데 이용하며 분류의 성능 지표로 사용되는 것은 ROC 곡선 면적에 기반한 AUC값으로 결정함. → 1에 가까울수록 좋은 수치.

06. 피마 인디언 당뇨병 예측

07. 정리

성능 평가 지표: 정확도, 오차행렬, 정밀도, 재현율, F1 스코어, ROC-AUC