# BITAMIN 8주차 정규 Session

## 머신러닝 평가 방법 및 이론

TEAM

비타민 6기 6조

TEAM MEMBERS

김중훈 김현정 백주은 정도현

Start

# *CONTENTS*

1. 정확도(Accuracy)

2. 오차 행렬

3. 정밀도와 재현율

4. F1 스코어

5. ROC 곡선과 AUC

6. 피마 인디언 당뇨병 예측

CHAPTER.0

# 머신러닝

## 머신러닝의 프로세스


## 성능 평가 지표의 유형

- 모델
  - 분류
    - 정확도(Accuracy)
    - 오차행렬(Confusion Matrix)
    - 정밀도(Precision)
    - 재현율(Recall)
    - F1 스코어
    - ROC AUC
  - 회귀

**분류**

**멀티 분류:** 여러 개의 결정 클래스 값

**이진 분류:** 긍정/부정 2개의 결과값

# 1. 정확도 (Accuracy)

CHAPTER.1

# 정확도(Accuracy)

## 정확도(Accuracy)란?

: 실제 데이터에서 예측 데이터가 얼마나 같은지를 판단하는 지표

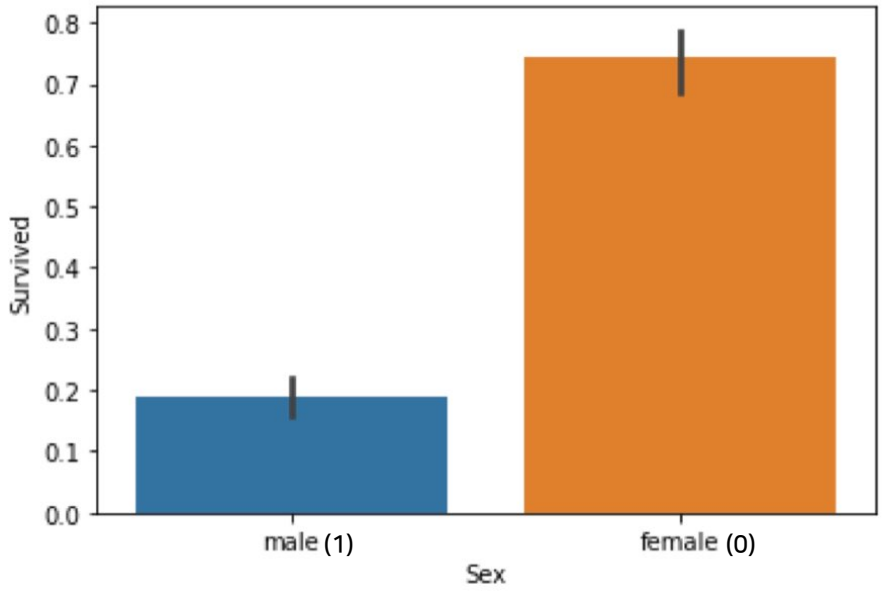
$$\text{정확도} = \frac{\text{예측 결과가 동일한 데이터 건수}}{\text{전체 예측 데이터 건수}}$$

BUT 이진 분류의 불균형한 레이블 데이터 세트의 경우 성능 왜곡 가능

따라서, 여러 가지 분류 지표와 함께 적용해야 함!

# 정확도(Accuracy)

## 타이타닉 생존자 예측




1) 사이킷런의 **BaseEstimator 클래스** 상속 받아 Customized 형태의 Estimator 생성

```
from sklearn.base import BaseEstimator

class MyDummyClassifier(BaseEstimator):
    # fit() 메서드는 아무것도 학습하지 않음
    def fit(self, X, y=None):
        pass
    # predict() 메서드는 단순히 Sex 피처가 1이면 0, 그렇지 않으면 1로 예측함
    def predict(self, X):
        pred = np.zeros((X.shape[0], 1))
        for i in range (X.shape[0]):
            if X['Sex'].iloc[i] == 1:
                pred[i] = 0
            else:
                pred[i] = 1

        return pred
```

**MyDummyClassifier 클래스** 생성

학습을 수행하는 **fit()** 메서드는 아무것도 수행하지 않음

예측을 수행하는 **predict()** 메서드는 성별에 따라 생존자 예측

# 정확도(Accuracy)

## 타이타닉 생존자 예측

### 2) transform_features() 함수 정의

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder

# Null 처리 함수
def fillna(df):
    df['Age'].fillna(df['Age'].mean(),inplace=True)
    df['Cabin'].fillna('N',inplace=True)
    df['Embarked'].fillna('N',inplace=True)
    df['Fare'].fillna(0,inplace=True)
    return df

# 머신러닝 알고리즘에 불필요한 속성 제거
def drop_features(df):
    df.drop(['PassengerId','Name','Ticket'],axis=1,inplace=True)
    return df

# 레이블 인코딩 수행.
def format_features(df):
    df['Cabin'] = df['Cabin'].str[:1]
    features = ['Cabin','Sex','Embarked']
    for feature in features:
        le = LabelEncoder()
        le = le.fit(df[feature])
        df[feature] = le.transform(df[feature])
    return df

# 앞에서 설정한 Data Preprocessing 함수 호출
def transform_features(df):
    df = fillna(df)
    df = drop_features(df)
    df = format_features(df)
    return df
```

### 3) 원본 데이터 재로딩, 데이터 가공, 학습/테스트 데이터 분할

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

# 데이터 로딩
titanic_df = pd.read_csv('./titanic_train.csv')

# 데이터 가공
y_titanic_df = titanic_df['Survived']
X_titanic_df = titanic_df.drop('Survived', axis=1)
X_titanic_df = transform_features(X_titanic_df)

# 학습/테스트 데이터 분할
X_train, X_test, y_train, y_test = train_test_split(X_titanic_df, y_titanic_df,
                                                    test_size=0.2, random_state=0)
```

### 4) MyDummyClassifier 이용해 학습/예측/평가

```
# MyDummyClassifier 이용
myclf = MyDummyClassifier()

# 학습 수행
myclf.fit(X_train, y_train)

# 예측 수행
mypredictions = myclf.predict(X_test)

# 평가 수행
print('Dummy Classifier의 정확도는: {0:.4f}'.format(accuracy_score(y_test, mypredictions)))
```

Dummy Classifier의 정확도는: 0.7877

단순한 알고리즘으로 예측을 하더라도
불균형한 레이블 데이트 세트에서는
높은 수치의 정확도 결과가 나올 수 있음!

CHAPTER.1

# 정확도(Accuracy)

## MNIST 데이터 세트

load_digits() API 통해 제공

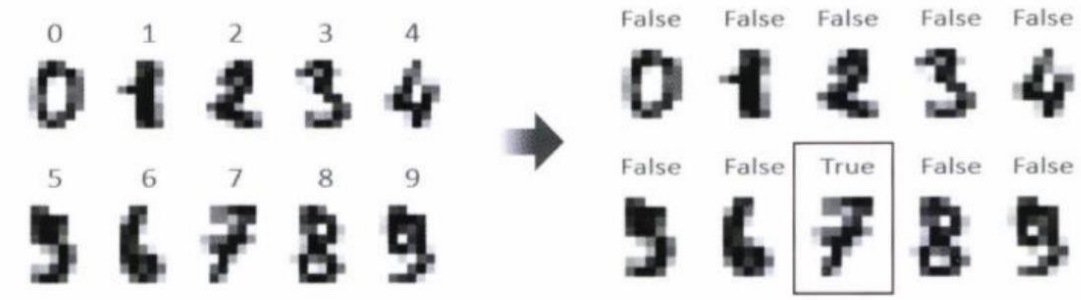
- 0부터 9까지의 숫자 이미지의 픽셀 정보를 가지고 있음
- 이를 기반으로 숫자 Digit 예측 가능

불균형한 데이터 세트로 만든 뒤에 정확도 지표 적용시 어떤 문제가 발생할 수 있을까?

**정확도 평가 지표의 맹점**

모든 데이터를 False(0)로 예측하는 classifier을 이용해 정확도 측정 시 90%에 가까운 예측 정확도

MNIST 데이터셋을 **멀티 분류**에서 **이진 분류**로 변경




전체 데이터의 10%만 True,
나머지 90%는 False인
불균형한 데이터 세트

CHAPTER.1

# 정확도(Accuracy)

## MNIST 데이터 세트

### 1) Classifier 생성

```
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 세트의 크기만큼 모두 0값으로 만들어서 변환
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
```

predict()의 결과를 np.zeros()로 모두 0값으로 반환

### 2) 불균형한 데이터 세트 생성

레이블 값이 7인 것만 True,
나머지 값은 모두 False로 변환

```
# 사이킷런의 내장 데이터 세트인 load_digits()를 이용해 MNIST 데이터 로딩
digits = load_digits()

# digits 번호가 7번이면 True이고 이를 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)
```

### 3) 불균형한 데이터로 생성한 y_test의 데이터 분포도 확인

MyFakeClassifier 이용해 예측과 평가 수행

```
# 불균형한 레이블 데이터 분포도 확인
print('레이블 테스트 세트 크기:', y_test.shape)
print('테스트 세트 레이블 0과 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 0과 1의 분포도
0    405
1     45
dtype: int64
모든 예측을 0으로 하여도 정확도는:0.900
```

**모든 예측을 0으로 하여도 예측 정확도 90%**

# 2. 오차 행렬

# (Confusion Matrix)

CHAPTER.2

# 오차 행렬

## 오차 행렬(confusion matrix)란?

: 이진 분류의 예측 오류가 얼마인지와 더불어 어떠한 유형의 예측 오류가 발생하고 있는지를 함께 나타내는 지표

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

True / False
예측값과 실제값이 '같은가/틀린가'

Negative/Positive
예측 결과 값이 부정(0)/긍정(1)

TN 예측값을 0으로 예측, 실제값도 0 (같음)
FP 예측값을 1로 예측, 실제값은 0 (틀림)
FN 예측값을 0으로 예측, 실제값은 1 (틀림)
TP 예측값을 1로 예측, 실제값도 1 (같음)

# 오차 행렬

MyFakeClassifier의 예측 성능 지표를 오차 행렬로 표현

confusion_matrix() API

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_test, fakepred)
```

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

| | | 예측 클래스 | |
|---|---|---|---|
| | | Negative<br>7이 아니라고 예측 | Positive<br>7이라고 예측 |
| 실제 클래스 | Negative<br>실제 7이 아닌 것 | **TN**<br>405개 | **FP**<br>0 |
| | Positive<br>실제 7인 것 | **FN**<br>45개 | **TP**<br>0 |

CHAPTER.2

# 오차 행렬

$$\text{정확도} = \frac{\text{TN + TP}}{\text{TN + FP + FN + TP}}$$

(예측 결과가 동일한 데이터 건수)

(전체 예측 데이터 건수)

불균형한 이진 분류 데이트 세트에서는
중점적으로 찾아야 하는 적은 수의 결과값에 Positive 설정해 1값 부여,
그렇지 않은 경우는 Negative로 0값 부여

ex. 사기 행위 예측 모델
사기 행위가 Positive(1), 정상 행위가 Negative(0)
ex. 암 검진 예측 모델
암이 양성일 경우 Positive(1), 음성일 경우 Negative(0)

Positive 데이터 건수 매우 적음
→ ML 알고리즘은 Negative로 예측하는 경향

**Positive에 대한 예측 정확도 판단 없이**
**Negative에 대한 예측 정확도만으로**
**분류의 정확도가 매우 높게 나타남**

# 3. 정밀도와 재현율

CHAPTER.3

# 정밀도와 재현율

## 정밀도와 재현율이란?

: Positive 데이터 세트의 예측 성능에 좀 더 초점을 맞춘 평가 지표

$$\text{정밀도} = \frac{TP}{FP + TP}$$

(=양성 예측도)

(예측과 실제 값이 Positive)

(예측을 Positive로 한 대상)

$$\text{재현율} = \frac{TP}{FN + TP}$$

(=민감도, TPR)

(예측과 실제 값이 Positive)

(실제 값이 Positive인 대상)

| | 예측 클래스 | |
|---|---|---|
| 실제 클래스 | Negative | Positive |
| Negative | TN | FP |
| Positive | FN | TP |

CHAPTER.3

# 정밀도와 재현율

## 재현율이 중요 지표인 경우

실제 Positive인 데이터를 Negative로
잘못 판단하면 업무상 큰 영향이 발생하는 경우

$$\frac{TP}{FN + TP}$$

예측: Negative
실제: Positive

예측: Positive
실제: Positive

**ex. 암 판단 모델**

실제 Positive인 암 환자를
Negative 음성으로 잘못 판단할 경우 심각

**ex. 금융 사기 적발 모델**

실제 금융거래 사기인 Positive 건을
Negative로 잘못 판단할 경우 큰 손해

VS

## 정밀도가 중요 지표인 경우

실제 Negative인 데이터를 Positive로
잘못 판단하면 업무상 큰 영향이 발생하는 경우

$$\frac{TP}{FP + TP}$$

예측: Positive
실제: Negative

예측: Positive
실제: Positive

**ex. 스팸메일 여부 판단 모델**

실제 Negative인 일반 메일을
Positive인 스팸 메일로 분류할 경우
아예 메일을 못 받음

CHAPTER.3

# 정밀도와 재현율

## 1) get_clf_eval() 함수 정의

```
from sklearn.metrics import accuracy_score, precision_score , recall_score , confusion_matrix

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 ,recall))
```

**Get_clf_eval() 함수**
confusion matrix, accuracy, precision, recall 등의 평가 동시 호출

정밀도 계산 **precision_score()**
재현율 계산 **recall_score()**

## 2) 로지스틱 회귀 기반으로 타이타닉 생존자 예측

```
import numpy as np
import pandas as pd

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

# 원본 데이터를 재로딩, 데이터 가공, 학습데이터/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)
```

**로직스틱 회귀란?**

**선형 회귀 방식을 분류에 적용한 알고리즘**

```
오차 행렬
[[104  14]
 [ 13  48]]
정확도: 0.8492, 정밀도: 0.7742, 재현율: 0.7869
```

정밀도 < 재현율

**재현율 또는 정밀도를 좀 더 강화할 방법은 무엇일까?**

# 정밀도와 재현율 - 정밀도/재현율 트레이드오프

## 정밀도/재현율의 트레이드오프(Trade-Off)이란?

분류의 결정 **임계값**(Threshold) 조정 → 정밀도 또는 재현율 수치 조정

정밀도 ⬆ → 재현율 ⬇

재현율 ⬆ → 정밀도 ⬇

| 실제 클래스 \ 예측 클래스 | Negative | Positive |
|---|---|---|
| Negative | TN | FP |
| Positive | FN | TP |

# 정밀도와 재현율 - 정밀도/재현율 트레이드오프

```
pred_proba = lr_clf.predict_proba(X_test)
pred  = lr_clf.predict(X_test)
print('pred_proba()결과 Shape : {0}'.format(pred_proba.shape))
print('pred_proba array에서 앞 3개만 샘플로 추출 \n:', pred_proba[:3])

# 예측 확률 array 와 예측 결과값 array 를 concatenate 하여 예측 확률과 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])
```

```
pred_proba()결과 Shape : (179, 2)
pred_proba array에서 앞 3개만 샘플로 추출
: [[0.46162417 0.53837583]
 [0.87858538 0.12141462]
 [0.87723741 0.12276259]]
두개의 class 중에서 더 큰 확률을 클래스 값으로 예측
 [[0.46162417 0.53837583 1.        ]
 [0.87858538 0.12141462 0.        ]
 [0.87723741 0.12276259 0.        ]]
```

predict_prob()
개별 클래스 예측 확률

predict()
두 개 칼럼 중 더 큰
확률값으로 최종 예측

첫 번째 칼럼: 클래스 값 0에 대한 예측 확률

두 번째 칼럼: 클래스 값1에 대한 예측 확률

첫 번째 칼럼 + 두 번째 칼럼 = 1

**predict_proba() 메서드**

테스트 피처 데이터 세트 입력

테스트 피처 레코드의 개별 클래스 예측 확률 반환

**predict() 메서드**

예측 결과 클래스 값 반환

**개별 레이블별로 예측 확률이 큰 레이블 값으로 예측**

ex. 이진 분류 모델에서 특정 데이터가

0이 될 확률이 10%, 1이 될 확률이 90%

→ 최종 예측은 90% 확률을 가진 1로 예측

**임계값=50% 보다 크면 Positive, 작거나 같으면 Negative로 결정**

# 정밀도와 재현율 - 정밀도/재현율 트레이드오프

## 1) Binarizer 클래스

```
from sklearn.preprocessing import Binarizer

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

# threshold 기준값보다 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.]]
```

ndarray의 값이 지정된 threshold(1.1)보다 같거나 작으면 0, 크면 1로 변환

## 2) 분류 결정 임계값 0.5 기반에서 Binarizer를 이용하여 예측값 변환

```
from sklearn.preprocessing import Binarizer

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

# predict_proba( ) 반환값의 두번째 컬럼 , 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)

get_clf_eval(y_test, custom_predict)
```

get_clf_eval()→ 평가 지표 출력

```
오차 행렬
[[104  14]
 [ 13  48]]
정확도: 0.8492, 정밀도: 0.7742, 재현율: 0.7869
```

↳ predict()로 계산된 지표값과 같음
즉, predict()는 predict_proba()에 기반

# 정밀도와 재현율 - 정밀도/재현율 트레이드오프

3) 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)
```

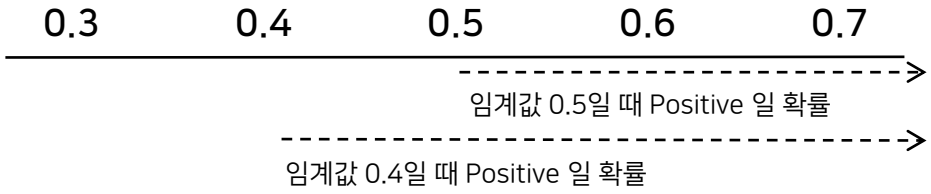
```
오차 행렬
[[99 19]
 [10 51]]
정확도: 0.8380, 정밀도: 0.7286, 재현율: 0.8361
```

CHAPTER.3

# 정밀도와 재현율 - 정밀도/재현율 트레이드오프

분류 결정 임계값은 **Positive 예측값을 결정**하는 확률의 기준

임계값보다 크면 positive 작거나 같으면 negative




임계값↓ → 재현율↑ 정밀도 ↓

(positive값 ↑) = TP / (FN+TP) = TP / (FP+TP)

Positive로 예측 ↓

실제 positive →

| TN | FP |
|---|---|
| 104 | 14 |
| FN | TP |
| 13 | 48 |

임계값 0.5일 때 오차 행렬

| TN | FP |
|---|---|
| 99 | 19 |
| FN | TP |
| 10 | 51 |

음성 예측 ↓ 양성 예측 ↑

임계값 0.4일 때 오차 행렬

# 정밀도와 재현율 - 정밀도/재현율 트레이드오프

4) 임계값을 0.4에서부터 0.6까지 0.05씩 증가 get_eval_by_threshold() 함수 정의

```
# 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
```

# 정밀도와 재현율 - 정밀도/재현율 트레이드오프

4) 임계값을 0.4에서부터 0.6까지 0.05씩 증가

| 평가 지표 | 분류 결정 임계값 | | | | |
|---|---|---|---|---|---|
| | 0.4 | 0.45 | 0.5 | 0.55 | 0.6 |
| 정확도 | 0.8380 | 0.8492 | 0.8492 | 0.8659 | 0.8771 |
| 정밀도 | 0.7286 | 0.7656 | 0.7742 | 0.8364 | 0.8824 |
| 재현율 | 0.8361 | 0.8033 | 0.7869 | 0.7541 | 0.7377 |

# 정밀도와 재현율 - 정밀도/재현율 트레이드오프

## 1) 타이타닉 예측 모델의 임곗값에 따른 정밀도-재현율 값 추출

```
from sklearn.metrics import precision_recall_curve

# 레이블 값이 1일때의 예측 확률을 추출
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)

#반환된 임계값 배열 로우가 143건이므로 샘플로 10건만 추출하되, 임곗값을 14 Step으로 추출.
thr_index = np.arange(0, thresholds.shape[0], 14)
print('샘플 추출을 위한 임계값 배열의 index 10개:', thr_index)
print('샘플용 10개의 임곗값: ', np.round(thresholds[thr_index], 2))

# 14 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))
```

```
반환된 분류 결정 임곗값 배열의 Shape: (143,)
샘플 추출을 위한 임계값 배열의 index 10개: [  0  14  28  42  56  70  84  98 112 126 140]
샘플용 10개의 임곗값:  [0.1  0.12 0.13 0.17 0.25 0.36 0.51 0.63 0.75 0.88 0.95]
샘플 임계값별 정밀도:  [0.389 0.437 0.458 0.524 0.618 0.707 0.787 0.915 0.938 0.944 1.   ]
샘플 임계값별 재현율:  [1.    0.967 0.902 0.902 0.902 0.869 0.787 0.705 0.492 0.279 0.066]
```

### precision_recall_curve() API

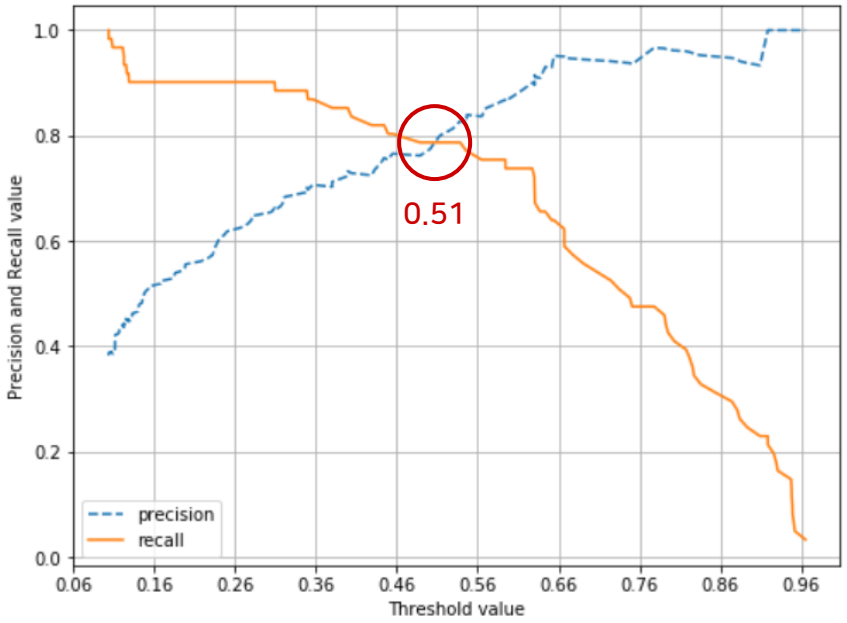
입력 파라미터:
**y_true** (실제 클래스값 배열),
**probas_pred** (positive 칼럼의 예측 확률 배열)

반환 값: 정밀도, 재현율

# 정밀도와 재현율 - 정밀도/재현율 트레이드오프

## 2) 임곗값의 변경에 따른 정밀도-재현율 변화 곡선 시각화

```
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='precision')
    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] )
```



CHAPTER.3

# 정밀도와 재현율 – 정밀도와 재현율 맹점

## 정밀도가 100%가 되는 방법

확실한 기준이 되는 경우만 Positive로
예측하고 나머지는 모두 Negative로 예측

ex. 전체 환자 1000명 중 확실한 Positive 징후만 가진 환자는 단 1명이라고 하면 이 한 명만 Positive로 예측하고 나머지는 모두 Negative로 예측

정밀도 = TP / (TP + FP) = 1/(1+0) = 100%

VS

## 재현율이 100%가 되는 방법

모두 Positive로 예측

ex. 실제 양성인 사람이 30명 정도라도 전체 환자 1000명을 다 Positive로 예측

재현율 = TP / (TP + FN) = 30 / (30 + 0) = 100%

주의!

정밀도/재현율 중 하나만 강조하는 상황이 돼서는 안 되며,
업무 환경에 맞게 정밀도와 재현율의 수치를 상호 보완할 수 있는 수준에서 임계값을 변경해야 한다.
정밀도 또는 재현율 중 하나만 스코어가 좋고 다른 하나는 스코어가 나쁜 분류는 성능이 좋지 않은 분류로 간주할 수 있다.

➡ 정밀도와 재현율 수치를 적절하게 조합돼 평가할 수 있는 지표 필요! F1 스코어

# 4. F1 Score

CHAPTER.4

# F1 Score

## 정밀도와 재현율을 결합한 지표

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

정밀도와 재현율의 조화평균

| 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) |

*If) A*

Precision = 0.9
Recall = 0.1 ➡ F1 score = 0.18

VS

*B*

Precision = 0.5
Recall = 0.5 ➡ F1 score = 0.5

# F1 Score

## f1_score()

➡ 로지스틱 회귀기반 타이타닉 생존자 모델로

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

```
F1 스코어 : 0.7805
```

```
임곗값: 0.4
오차 행렬
[[99 19]
 [10 51]]
정확도:0.8380, 정밀도:0.7286, 재현율:0.8361, F1:0.7786
임곗값: 0.45
```

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

✓ 앞의 get_clf_eval 함수에 F1 score 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 스코어 추가
    f1=f1_score(y_test, pred)
    print('오차 행렬')
    print(confusion)

    #F1 스코어 print 추가
    print('정확도:{0:.4f}, 정밀도:{1:.4f}, 재현율:{2:.4f}, F1:{3:.4f}'
          .format(accuracy, precision, recall, f1))

threshold = [0.4, 0.45, 0.5, 0.55, 0.6]
pred_proba=lr_clf.predict_proba(X_test)
get_eval_by_threshold(y_test, pred_proba[:,1].reshape(-1,1), thresholds)
```

# 5. ROC 곡선과 AUC

CHAPTER.5

# ROC 곡선과 AUC

## 이진분류기의 성능을 평가하는 지표

### ROC 곡선(Receiver Operation Characteristic Curve)

- ✓ FPR이 변할 때 TPR이 어떻게 변하는지를 나타내는 곡선
- ✓ TPR(True Positive Rate) : 재현율 or 민감도
- ✓ FPR(False Positive Rate) : 1-TNR(True Negative Rate)

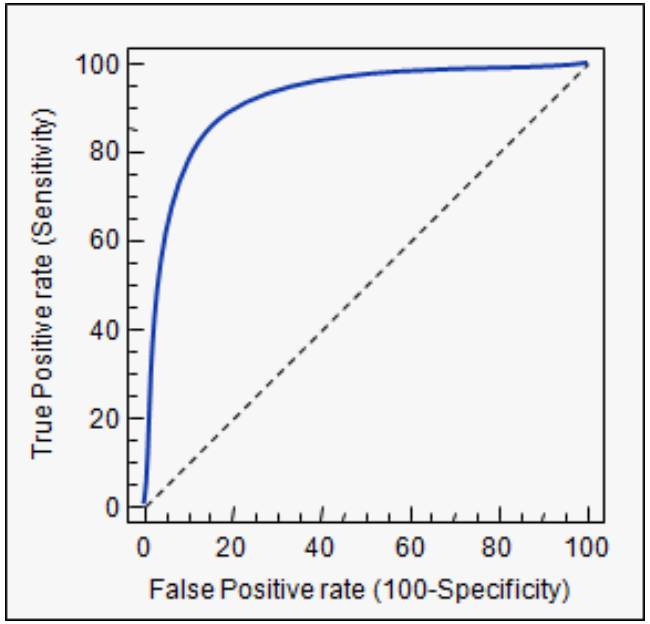
$$\frac{FP}{FP + TN}$$

**TPR(민감도)**

$$\frac{TP}{FN + TP}$$

**TNR(특이성)**

$$\frac{TN}{FP + TN}$$




| 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) |

# ROC 곡선과 AUC

## ROC 곡선(Receiver Operation Characteristic Curve)

✓ **TPR(True Positive Rate)** : 실제값 Positive가 정확히 Positive로 예측돼야 하는 수준
<Sensitivity, 민감도>
ex) 암환자를 암환자로 판정

$$\frac{TP}{FN+TP}$$

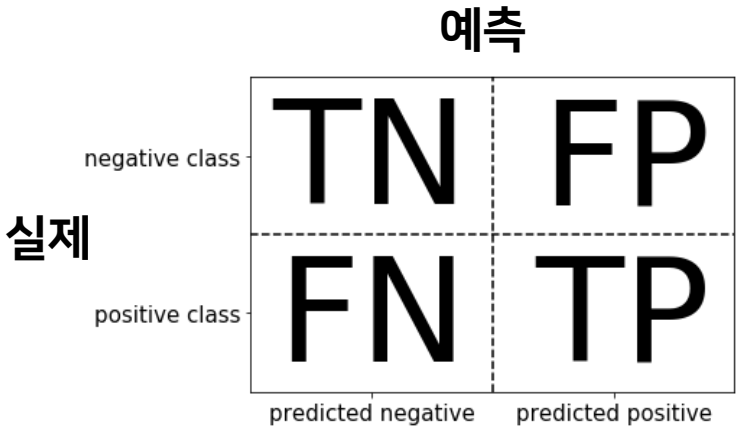
✓ **TNR(True Negative Rate)** : 실제값 Negative가 정확히 Negative로 예측돼야 하는 수준
<Specificity, 특이성>
ex) 건강한 사람을 건강한 사람으로 판정

$$\frac{TN}{FP+TN}$$

✓ **FPR(False Positive Rate)** : 실제값 Negative를 Positive로 예측
ex) 건강한 사람을 암환자로 판정→잘못 판단

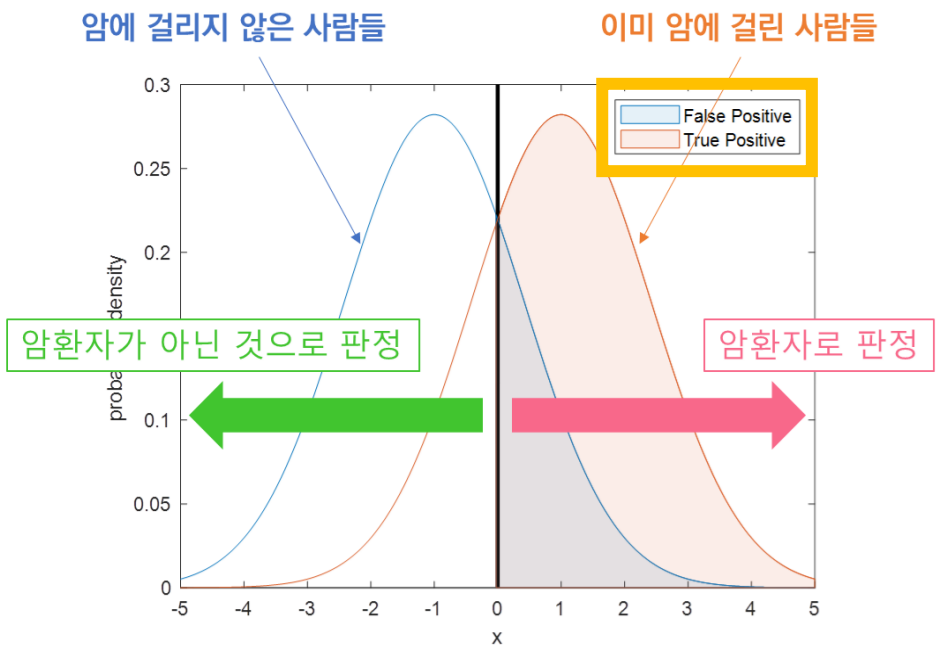
$$\frac{FP}{FP+TN}$$ **=1-TNR**

**=1-특이성**



# ROC 곡선과 AUC

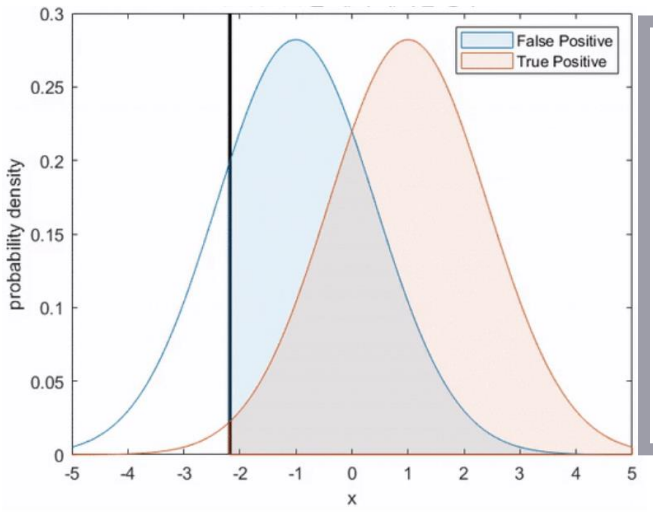
## TPR vs FPR




*If)*

의사가 성격이 급해 모든 환자들을 다 암환자로 판단

->임곗값(threshold)이 낮음



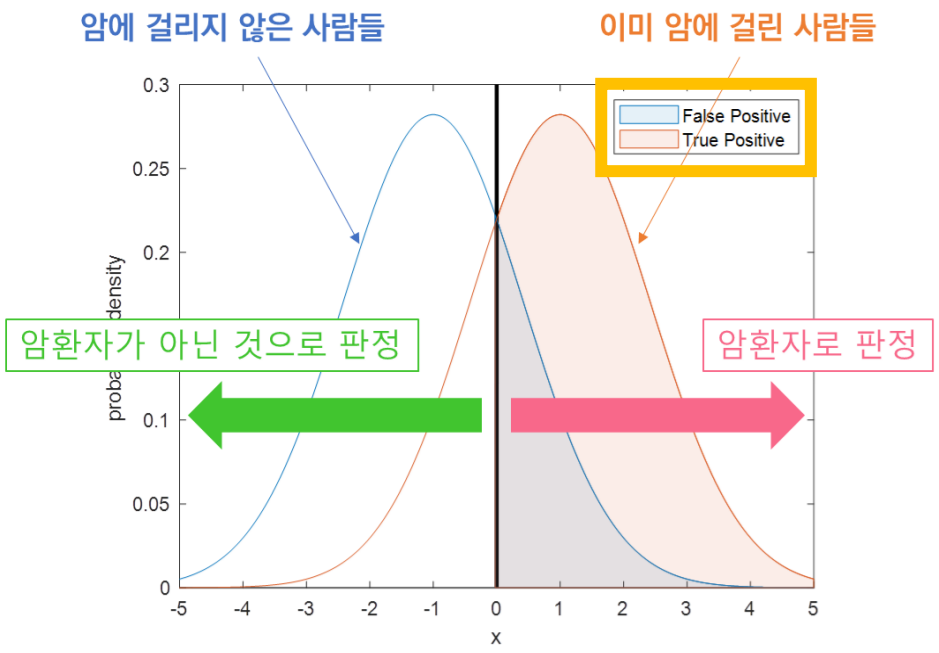

✓ 실제로 암에 걸린 환자들은 암환자 판정(TPR ↑)

✓ 암에 걸리지 않은 사람들도 암환자 판정(FPR ↑)

출처 : 공돌이의 수학노트

# ROC 곡선과 AUC

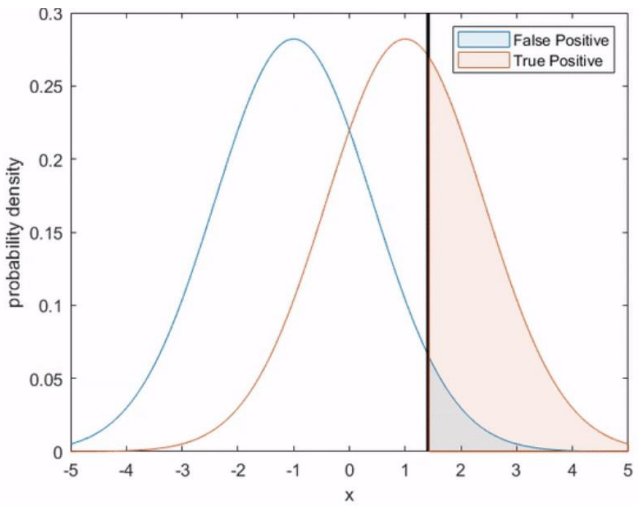
## TPR vs FPR




If)

의사가 겁이 많아 모든 환자들을 암이 아니라고

->임곗값(threshold)이 높음



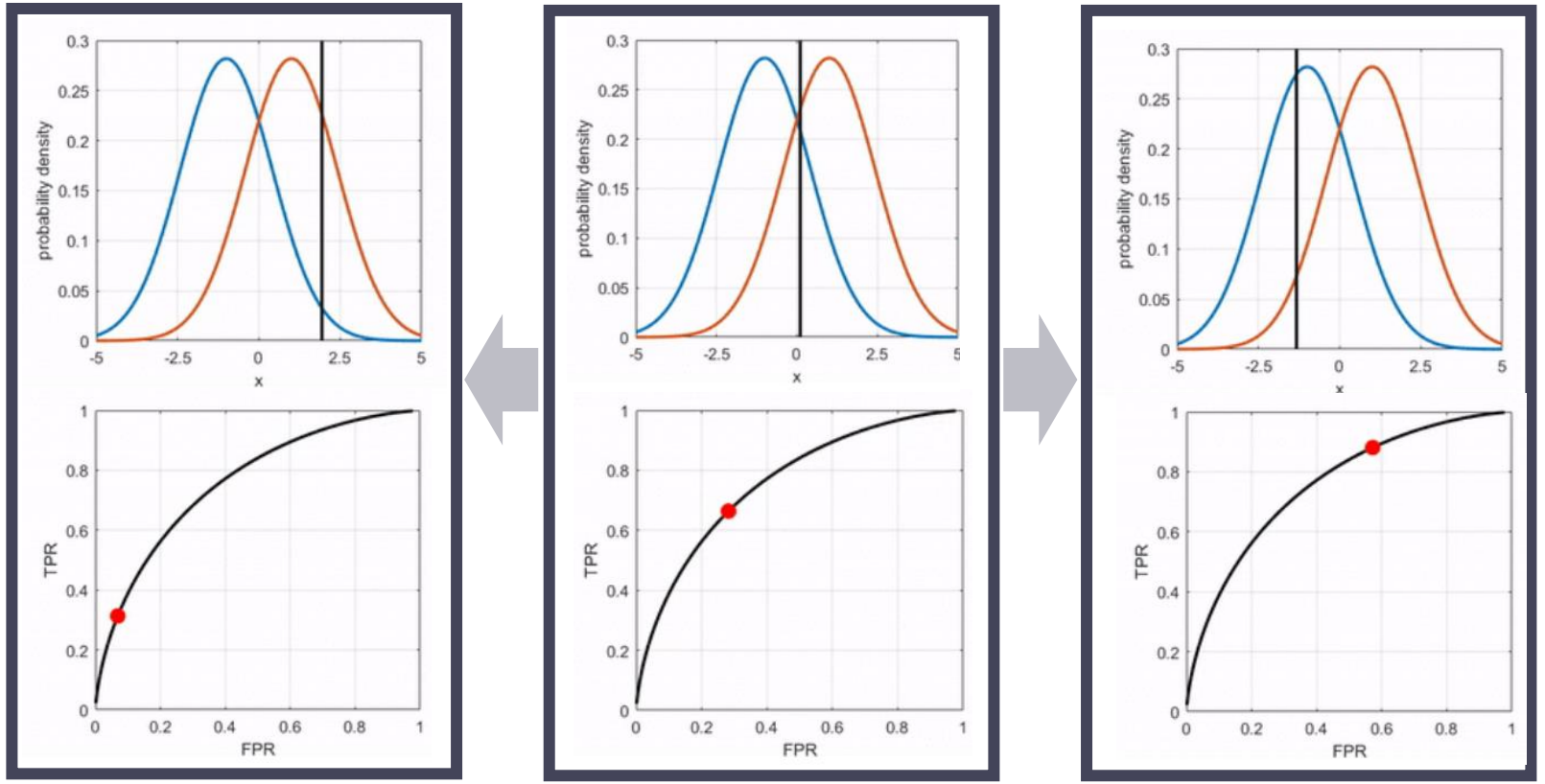

✓ 암에 걸리지 않은 내원자들 정상인 판정(FPR ↓)

✓ 암에 걸린 내원자들도 정상인(TPR ↓)

출처 : 공돌이의 수학노트

# ROC 곡선과 AUC

## ROC 곡선 위의 점




- ✓ TPR과 FPR은 어느정도 비례적으로 함께 커지거나 감소

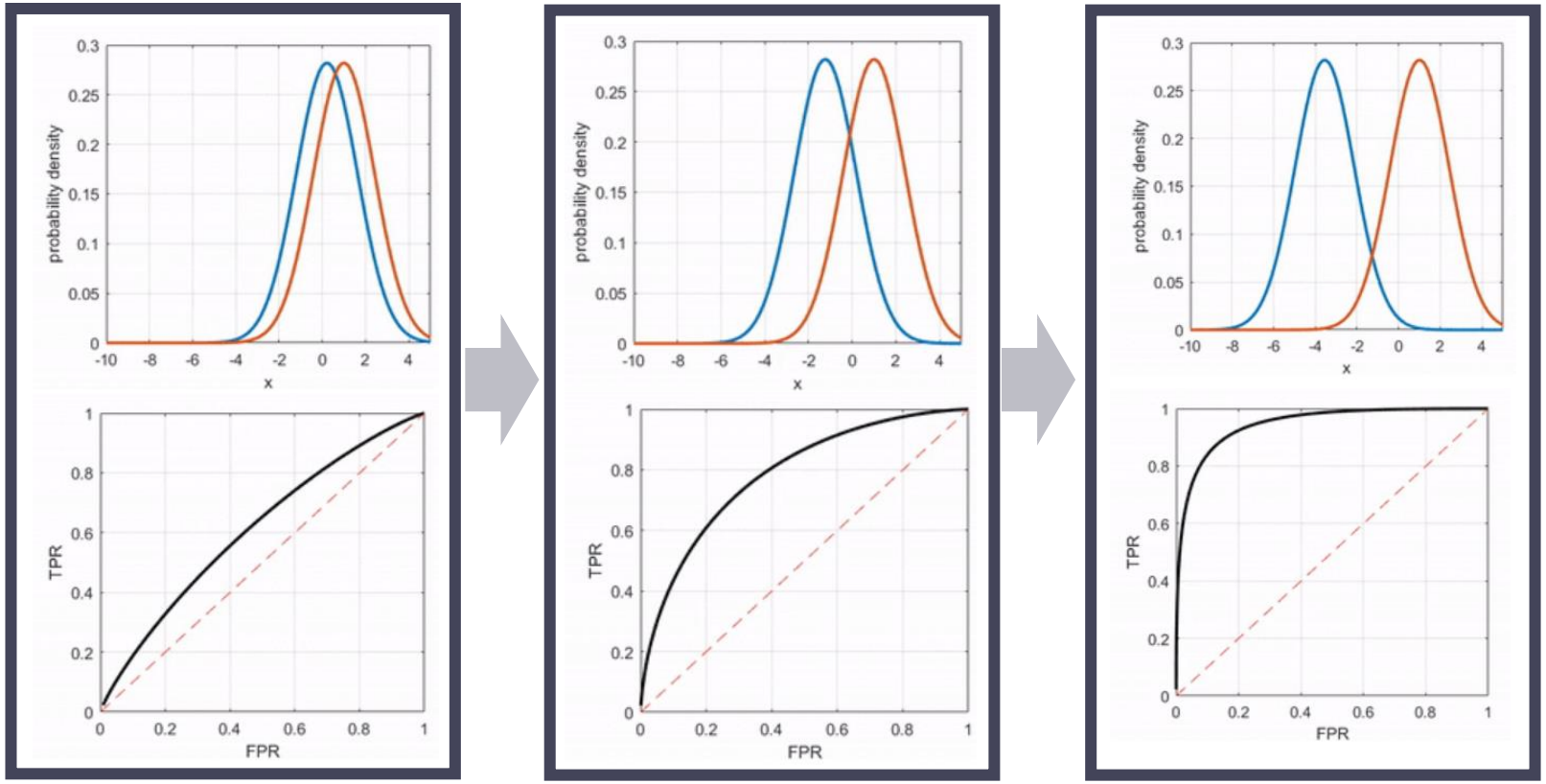
### 현 위의 점의 의미

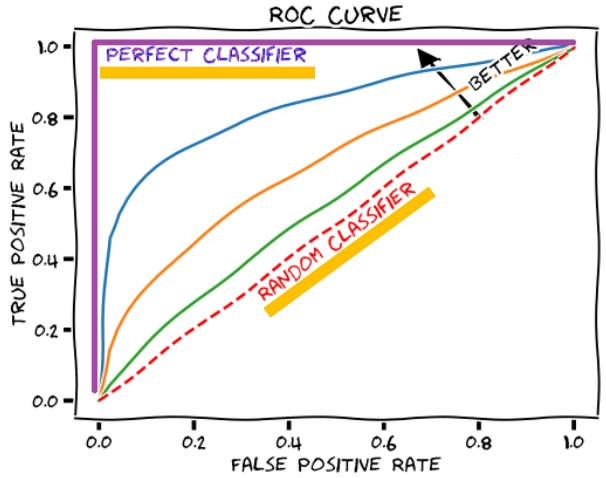
- ✓ 분류기의 분류성능은 변하지 않음
- ✓ 가능한 모든 임곗값 별 FPR과 TPR 표시

출처 : 공돌이의 수학노트

# ROC 곡선과 AUC

## 곡선의 휨 정도




좌측 상단에 곡선이
붙어있을수록 좋은 분류기!

출처 : 공돌이의 수학노트

# ROC 곡선과 AUC

## ROC 곡선 그리기

FPR을 0부터 1까지 변경하면서 TPR의 변화값을 구함

↓

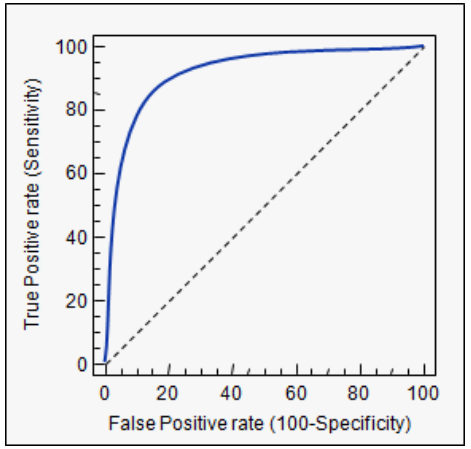
임곗값 변경

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

$$FPR = \frac{FP}{FP+TN}$$

임곗값 = 1 → 모두 Negative 예측 → FP=0 → FPR=0 → TPR

임곗값 = 0 → 모두 Positive 예측 → TN=0 → FPR=1 → TPR



# ROC 곡선과 AUC

## roc_curve()

```
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 5step으로 추출.
#thresholds[0]dms 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))

#5step단위로 추출된 임곗값에 따른 FPR, TPR 값
print('샘플 임곗값 별 FPR : ', np.round(fprs[thr_index], 3))
print('샘플 임곗값 별 TPR : ', np.round(tprs[thr_index], 3))
```

| | |
|---|---|
| 입력 파라미터 | y_true : 실제 클래스 값 array (array shape=[데이터 건수])<br>y_score : predict_proba()의 반환 값 array 에서 Positive 칼럼의 예측 확률이 보통 사용됨.<br>array.shape=[n_samples] |
| 반환 값 | fpr : fpr 값을 array로 반환<br>tpr : tpr 값을 array로 반환<br>thresholds : threshold 값 array |

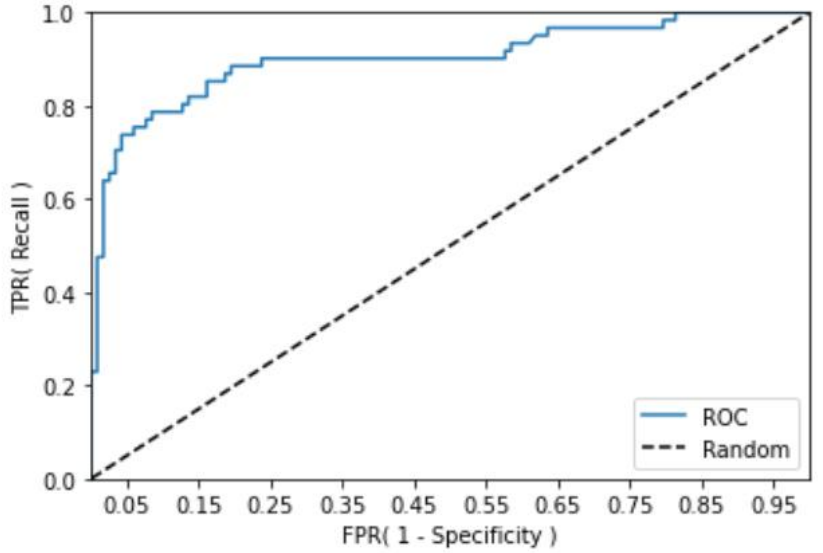
```
샘플 추출을 위한 임곗값 배열의 index :  [ 1  6 11 16 21 26 31 36 41 46 51]
샘플 index로 추출한 임곗값 :  [0.97 0.65 0.63 0.56 0.45 0.38 0.31 0.13 0.12 0.11 0.1 ]
샘플 임곗값 별 FPR :  [0.    0.017 0.034 0.076 0.127 0.186 0.237 0.576 0.619 0.754 0.814]
샘플 임곗값 별 TPR :  [0.033 0.639 0.705 0.754 0.803 0.852 0.902 0.902 0.951 0.967 1.   ]
```

- ✓ 임곗값이 점점 작아지면서 FPR 증가
- ✓ FPR이 조금씩 커질때 TPR은 가파르게 증가

# ROC 곡선과 AUC

## roc_curve()

```
import matplotlib.pyplot as plt

def roc_curve_plot(y_test, pred_proba_c1):
    # 임곗값에 따른 FPR, TPR 값을 반환받음
    fprs, tprs, thresholds = roc_curve(y_test, pred_proba_c1)
    # ROC곡선을 그래프 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 0.1단위로 변경, X, Y축 명 설정 등
    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 - Specificity )'); plt.ylabel('TPR( Recall )')
    plt.legend()

roc_curve_plot(y_test, pred_proba[:,1])
```




ROC곡선 자체는 FPR과 TPR의 변화 값을 보는데 이용.

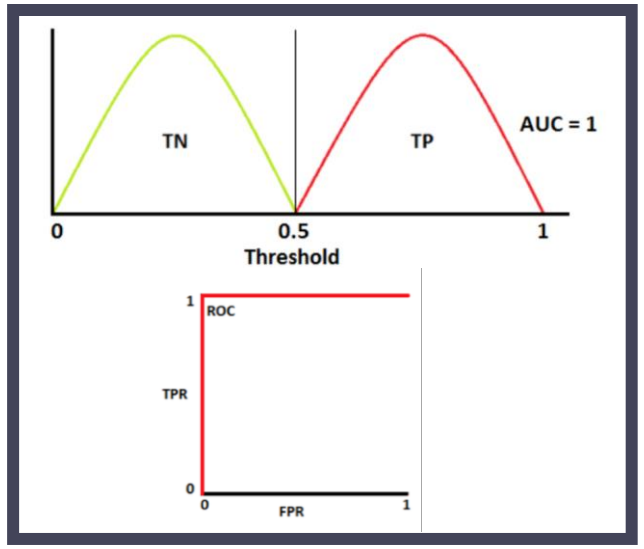
분류 성능 지표로 사용되는 것은 ROC 곡선 면적에 기반한 AUC 값

# ROC 곡선과 AUC

## AUC(Area Under Curve)

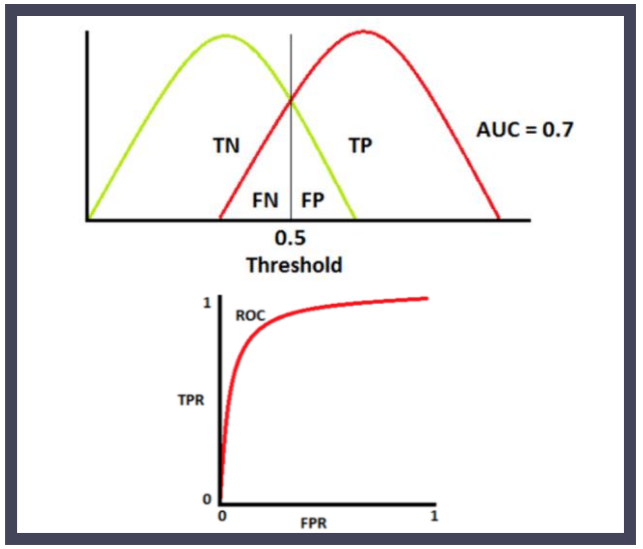
ROC 곡선 아래 면적 → 1에 가까울 수록 좋음 → FPR이 작을 때 얼마나 큰 TPR을 얻을 수 있는가

### 1) AUC=0



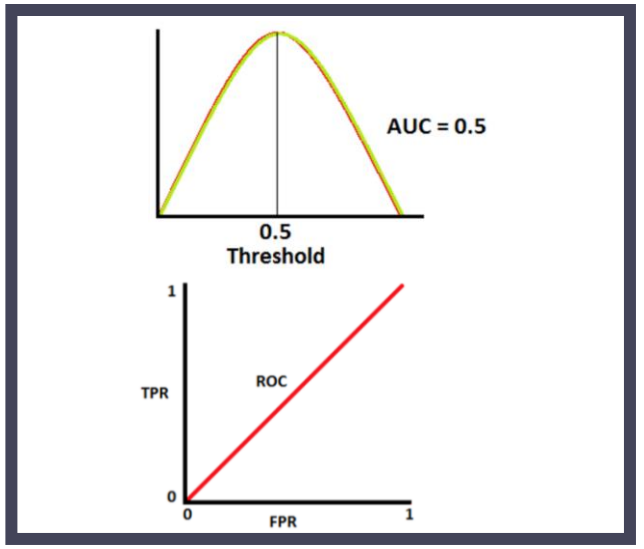

→ 이상적인 분류 성능
양성/음성 클래스 완벽 구분

### 1) AUC=0




→ threshold에 따른 오류 정도
양성/음성 구분 확률 70%

### 1) AUC=0




→ 최악의 모델 성능
양성/음성 구분 능력 없음

출처 : https://bioinformaticsandme.tistory.com/328

# ROC 곡선과 AUC

## AUC(Area Under Curve)

✓ 앞의 get_clf_eval 함수에 ROC AUC 값 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 추가

    roc_auc = roc_auc_score(y_test, pred_proba)
    print('오차행렬')
    print(confusion)

    #ROC-AUC print 추가
    print('정확도 : {0:.4f}, 정밀도 : {1:.4f}, 재현율 : {2:.4f}, F1 : {3:.4f}, AUC : {4:.4f}'
          .format(accuracy, precision, recall, f1, roc_auc))
```

ROC AUC는 예측 확률값을 기반으로 계산되므로

이를 get_clf_eval( )함수의 인자로 받을 수 있도록

get_clf_eval(y_test, pred=None, pred_proba=None)으로

함수형 변경

# ROC 곡선과 AUC

## AUC(Area Under Curve)

✓ 앞의 get_eval_by_threshold 함수 수정

```
def get_eval_by_threshold(y_test, pred_proba_c1, thresholds):
    #thresolds list객체 내의 값을 차례로 iteration 하면서 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, pred_proba_c1)
```

pred_proba 변수 추가

→ AUC값은 뒤의 예제에서 확인

# 6. 피마 인디언 당뇨병 예측

# 피마 인디언 당뇨병 예측

## 데이터 확인

diabetes.csv → 피마 원주민의 Type-2 당뇨병 결과 데이터

```
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

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

```
0    500  →Negative
1    268  →Positive
Name: Outcome, dtype: int64
```

# 피마 인디언 당뇨병 예측

## 데이터 확인

diabetes.csv → 피마 원주민의 Type-2 당뇨병 결과 데이터

```
diabetes_data.info()
```

```
<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
```

| | 임신횟수 | 포도당 수치 | 혈압 | 피하지방 측정값 |
|---|---|---|---|---|
| | **Pregnancies** | **Glucose** | **BloodPressure** | **SkinThickness** |
| **0** | 6 | 148 | 72 | 35 |
| **1** | 1 | 85 | 66 | 29 |
| **2** | 8 | 183 | 64 | 0 |

| 혈청 인슐린 | | 당뇨 내력 가중치 값 | | 클래스 결정 값 |
|---|---|---|---|---|
| **Insulin** | **BMI** | **DiabetesPedigreeFunction** | **Age** | **Outcome** |
| 0 | 33.6 | 0.627 | 50 | 1 |
| 0 | 26.6 | 0.351 | 31 | 0 |
| 0 | 23.3 | 0.672 | 32 | 1 |

# 피마 인디언 당뇨병 예측

## 로지스틱 회귀를 이용한 예측 모델 생성

```
# 피처 데이터 세트 x, 레이블 데이터 세트 y를 추출
# 맨 끝이 Outcome 칼럼으로 레이블 값임. 칼럼 위치 -1을 이용해 추출
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)

# 로지스틱 회귀로 학습, 예측 및 평가 수행
lr_clf = LogisticRegression()
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)
```

**stratify** : 지정한 Data의 비율을 유지.
예를 들어, Label Set인 Y가 25%의 0과 75%의 1로 이루어진 Binary Set일 때, stratify=Y로 설정하면 나누어진 데이터셋들도 0과 1을 각각 25%, 75%로 유지한 채 분할됨.

```
오차행렬
[[88 12]
 [23 31]]
정확도 : 0.7727, 정밀도 : 0.7209, 재현율 : 0.5741, F1 : 0.6392, AUC : 0.7919
```

**전체 데이터의 65%가 Negative이므로**
**정확도 보다는 재현율 성능에 초점을 맞춰보자!**

# 피마 인디언 당뇨병 예측

## Precision recall curve

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


0.42 임곗값에서 균형을
어느 정도 균형을 이루지만
두 값 모두 너무 낮음

→ 데이터에 개선할 부분이 있는지 확인

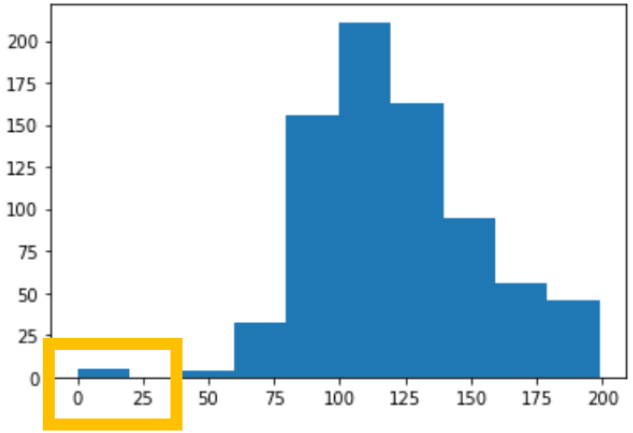
# 피마 인디언 당뇨병 예측

## 데이터 점검

```
diabetes_data.describe()
```

| | 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 |

```
plt.hist(diabetes_data['Glucose'], bins=10)
```




포도당 수치가 0일 수는 없음

# 피마 인디언 당뇨병 예측

## min = 0 데이터 확인

```
#0값을 검사할 피처 명 리스트
zero_features=['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI']

#전체 데이터 건수
total_count = diabetes_data['Glucose'].count()

#피처별로 반복하면서 데이터 값이 0인 데이터 건수를 추출하고, 퍼센트 계산
for feature in zero_features:
    zero_count = diabetes_data[diabetes_data[feature]==0][feature].count()
    print('{0} 0 건수는 {1}, 퍼센트는 {2:.2f} %'.format(feature, zero_count, 100*zero_count/total_count))
```

출산횟수 제외

```
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 %
```

→ 평균값으로 대체

```
# zero_features 리스트 내부에 저장된 개별 피처들에 대해서 0값을 평균값으로 대체
mean_zero_features = diabetes_data[zero_features].mean()
diabetes_data[zero_features]=diabetes_data[zero_features].replace(0, mean_zero_features)
```

# 피마 인디언 당뇨병 예측

## 피처 스케일링 → 다시 학습/테스트

```
X=diabetes_data.iloc[:, :-1]
y=diabetes_data.iloc[:,-1]

# StandardScaler 클래스를 이용해 피처 데이터 세트에 일괄적으로 스케일링 적용
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)

# 로지스틱 회귀로 학습, 예측 및 평가 수행.
lr_clf = LogisticRegression()
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)
```

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

# 피마 인디언 당뇨병 예측

## 임곗값 변화

```
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)
```

| 평가 지표 | 분류 결정 임곗값 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0.3 | 0.33 | 0.36 | 0.39 | 0.42 | 0.45 | 0.48 | 0.50 |
| 정확도 | 0.7013 | 0.7403 | 0.7468 | 0.7532 | 0.7792 | 0.7857 | 0.7987 | 0.7987 |
| 정밀도 | 0.5513 | 0.5972 | 0.6190 | 0.6333 | 0.6923 | 0.7059 | 0.7447 | 0.7674 |
| 재현율 | 0.7963 | 0.7963 | 0.7222 | 0.7037 | 0.6667 | 0.6667 | 0.6481 | 0.6111 |
| F1 | 0.6515 | 0.6825 | 0.6667 | 0.6667 | 0.6792 | 0.6857 | 0.6931 | 0.6804 |
| ROC AUC | 0.8433 | 0.8433 | 0.8433 | 0.8433 | 0.8433 | 0.8433 | 0.8433 | 0.8433 |

```
임곗값: 0.33
오차행렬
[[72 28]
 [12 42]]
정확도 : 0.7403, 정밀도 : 0.6000, 재현율 : 0.7778, F1 : 0.6774, AUC : 0.8433
```

재현율은 높지만

정밀도가 매우낮음

```
임곗값: 0.48
오차행렬
[[88 12]
 [19 35]]
정확도 : 0.7987, 정밀도 : 0.7447, 재현율 : 0.6481, F1 : 0.6931, AUC : 0.8433
```

전체적인 성능 평가 지표를 유지하면서

재현율 약간 상승

# 피마 인디언 당뇨병 예측

## 임곗값 = 0.48 에서 예측

```
#임곗값을 0.48로 설정한 Binarizer 생성
binarizer = Binarizer(threshold=0.48)

#위에서 구한 lr_clf의 predict_proba() 예측 확률 array 에서 1에 해당하는 칼럼값을 Binarizer로 변환
pred_th_048 = binarizer.fit_transform(pred_proba[:,1].reshape(-1,1))

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

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

# 감사합니다