

# 1. 사이킷런



초급 세션 2팀 조주현

# #2.1 사이킷런 소개와 특징

# **사이킷런**: 대표적인 파이썬 머신러닝 라이브러리

# 사이킷런의 특징

- 1) 쉽고 가장 파이썬스러운 API 제공
- 2) 머신러닝을 위한 다양한 알고리즘 및 편리한 프레임워크, API 제공

# 설치하는 법

- 1) pip  
pip install scikit-learn
- 2) **conda (권장)**  
**conda install scikit-learn**

# 버전 확인하는 법

```
import sklearn  
print(sklearn.__version__)
```

# #2.2 첫 번째 머신러닝 만들어 보기 – 붓꽃 품종 예측하기

# 붓꽃 데이터 세트로 붓꽃의 품종을 분류(Classification)

# 꽃잎의 길이와 너비, 꽃받침의 길이와 너비 피처(Feature) 기반으로 꽃의 품종 예측

# 분류(Classification): 대표적인 지도학습 (Supervised learning) 방법

# 지도학습

다양한 피처와 분류 결정값인 레이블 데이터로 모델 학습 → 별도의 테스트 데이터 세트에서 미지의 레이블 예측

# 학습 데이터 세트 (학습을 위해 주어짐) vs 데이터 세트 (예측 성능 평가를 위해 별도로 주어짐)

# 모듈

sklearn.datasets: 사이킷런에서 자체제공하는 데이터 세트를 생성하는 모듈의 모임

sklearn.tree: 트리 기반 ML 알고리즘을 구현한 클래스의 모임

sklearn.model\_selection: 학습 데이터, 검증 데이터, 예측 데이터로 데이터를 분리하거나 최적의 하이퍼파라미터로 평가하기 위한 다양한 모듈의 모임

+) 하이퍼 파라미터: 알고리즘별 최적의 학습을 위해 직접 입력하는 파라미터들 통칭, 성능 튜닝 가능

## #2.2 첫 번째 머신러닝 만들어 보기 – 붓꽃 품종 예측하기

# 붓꽃 데이터 생성 – load\_iris()

# ML 알고리즘 – 의사 결정 트리 (Decision Tree) 알고리즘, 이를 구현한 DecisionTreeClassifier

# 데이터 세트 분리 (학습&테스트) – train\_test\_split() 함수 사용

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
```

EWCHA  
EUROPEAN

```
import pandas as pd

iris = load_iris()
iris_data = iris.data

iris_label = iris.target
print('iris target값:', iris_label)
print('iris target명:', iris.target_names)

iris_df = pd.DataFrame(data = iris_data, columns = iris.feature_names)
iris_df['label'] = iris.target
iris_df.head(3)
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	label
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0

## #2.2 첫 번째 머신러닝 만들어 보기 – 붓꽃 품종 예측하기

# 학습용 데이터 테스트용 데이터 분리

`train_test_split()` 이용

`test_size` 파라미터 입력 값의 비율로 분할

```
X_train, X_test, y_train, y_test = train_test_split(iris_data, iris_label, test_size = 0.2, random_state = 11)
```

## #2.2 첫 번째 머신러닝 만들어 보기 – 붓꽃 품종 예측하기

# 학습된 DecisionTreeClassifier 객체 이용하여 예측 수행

# DecisionTreeClassifier 객체의 `predict()` 메서드에 테스트용 피쳐 데이터 세트 입력하여 호출  
→ 학습된 모델 기반에서 테스트 데이터 세트에 대한 예측값을 반환

# 정확도를 통해 성능 측정

# **정확도**: 예측 결과가 실제 레이블 값과 얼마나 정확하게 맞는지 평가하는 지표

`accuracy_score()` – 첫 번째 파라미터에 **실제** 레이블 데이터 세트, 두 번째 파라미터에 **예측** 레이블 데이터 세트  
입력

# 데이터 세트 분리 → 모델 학습 → 예측 수행 → 평가

## #2.3 사이킷런의 기반 프레임워크 익히기

# `fit()`: ML 모델 학습

# `predict()`: 학습된 모델 예측

# 지도학습의 주요 두 축 - 분류(Classification)  
회귀(Regression)

# `Classifier`: 분류 알고리즘을 구현한 클래스

# `Regressor`: 회귀 알고리즘을 구현한 클래스

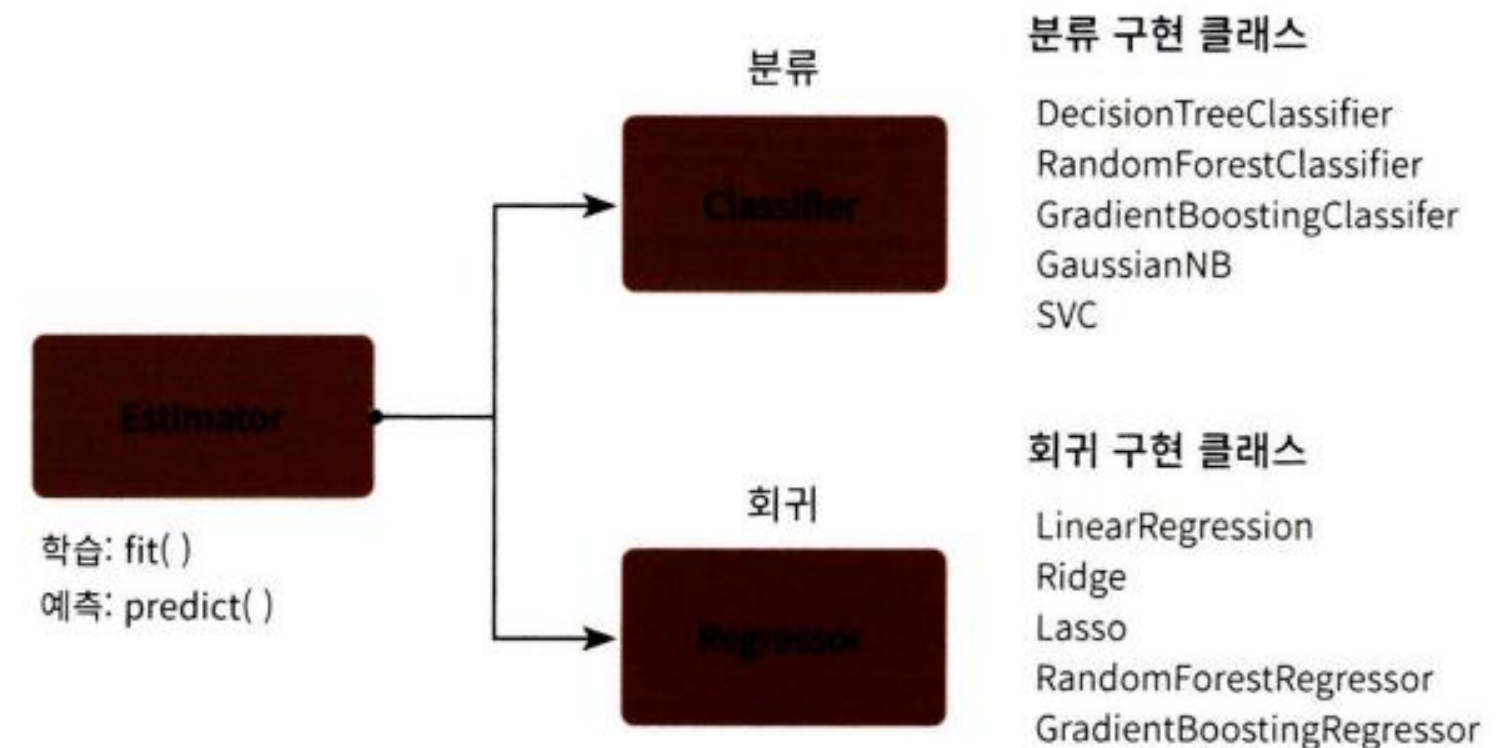
# `Classifier` + `Regressor` = `Estimator` 클래스

# 하이퍼 파라미터 튜닝을 지원하는 클래스의 경우  
`Estimator`를 인자로 받고, 함수 내에서 `Estimator`의 `fit()`와  
`predict()`를 호출해서 평가하거나 하이퍼 파라미터 튜닝 수행

# 비지도학습인 차원 축소, 클러스터링, 피쳐 추출 등을 구현한  
클래스에도 대부분 `fit()`과 `transform()` 적용

- 비지도학습과 피쳐 추출에서 `fit()`은 지도학습과 다르게 입력  
데이터의 형태에 맞춰 데이터를 변환하기 위한 사전 구조를  
맞추는 작업
- 실제 작업은 `transform()`으로 수행

# `fit()` + `transform()` = `fit_transform()`





# #2.3 사이킷런의 기반 프레임워크 익히기

## # 사이킷런의 주요 모듈

분류	모듈명	설명
예제 데이터	sklearn.datasets	사이킷런에 내장되어 예제로 제공하는 데이터 세트
피처 처리	sklearn.preprocessing	데이터 전처리에 필요한 다양한 가공 기능 제공
	sklearn.feature_selection	알고리즘에 큰 영향을 미치는 피처를 우선순위로 선택 작업 수행하는 다양한 기능 제공
	sklearn.feature_extraction	텍스트 데이터나 이미지 데이터의 벡터화된 피처를 추출하는 데 사용 텍스트 데이터의 피처 추출 - sklearn.feature_extraction.text 모듈에 지원 API 있음 이미지 데이터의 피처 추출 - sklearn.feature_extraction.image 모듈에 지원 API 있음
피처 처리 & 차원 축소	sklearn.decomposition	차원 축소와 관련한 알고리즘을 지원하는 모듈
데이터 분리, 검증 & 파라미터 튜닝	sklearn.model_selection	교차 검증을 위한 학습용/테스트용 분리, 그리드 서치로 최적 파라미터 추출 등 API 제공
평가	sklearn.metrics	분류, 회귀, 클러스터링, 페어와이즈에 대한 다양한 성능 측정 방법 제공, Accuracy, Preision, Recall, ROC-AUC, RMSE 등 제공

# #2.3 사이킷런의 기반 프레임워크 익히기

## # 사이킷런의 주요 모듈

분류	모듈명	설명
ML 알고리즘	sklearn.ensemble	앙상블 알고리즘 제공, 랜덤 포레스트, 에이다 부스트, 그래디언트 부스팅 등 제공
	sklearn.linear_model	주로 선형 회귀, 릿지, 라쏘 및 로지스틱 회귀 등 회귀 관련 알고리즘을 지원, 또한 SGD 관련 알고리즘도 제공
	sklearn.naïve_bayes	나이브 베이즈 알고리즘 제공, 가우시안 NB, 다항 분포 NB 등
	sklearn.neighbors	최근접 이웃 알고리즘 제공, K-NN 등
	sklearn.svm	서포트 벡터 머신 알고리즘 제공
	sklearn.tree	의사 결정 트리 알고리즘 제공
	sklearn.cluster	비지도 클러스터링 알고리즘 제공, K-평균, 계층형, DBSCAN 등
유틸리티	sklearn.pipeline	피처 처리 등의 변환과 ML 알고리즘 학습, 예측 등을 함께 묶어서 실행할 수 있는 유틸리티 제공

# #2.3 사이킷런의 기반 프레임워크 익히기

# 내장된 예제 데이터 세트 → 여러 API 호출로 만들 수 있음

# 분류나 회귀 연습용 예제 데이터

API 명	설명
datasets.load_boston()	회귀 용도, 미국 보스턴의 집 피쳐들과 가격에 대한 데이터 세트
datasets.load_breast_cancer()	분류 용도, 위스콘신 유방암 피쳐들과 악성/음성 레이블 데이터 세트
datasets.load_diabetes()	회귀 용도, 당뇨 데이터 세트
datasets.load_digits()	분류 용도, 0에서 9까지 숫자의 이미지 픽셀 데이터 세트
datasets.load_iris()	분류 용도, 붓꽃에 대한 피쳐를 가진 데이터 세트

# #2.3 사이킷런의 기반 프레임워크 익히기

# fetch 계열의 명령 - 데이터 크기가 커서 저장되어 있지 않고 인터넷에서 내려받아 서브 디렉터리에 저장 후 추후 불러들이는 데이터, 최초 사용 시 인터넷 연결 필수

fetch_covtype()	회귀 분석용 토지 조사 자료
fetch_20newsgroups()	뉴스 그룹 텍스트 자료
fetch_olivetti_faces()	얼굴 이미지 자료
fetch_lfw_people()	얼굴 이미지 자료
fetch_lfw_pairs()	얼굴 이미지 자료
fetch_rcv1()	로이터 뉴스 말뭉치
fetch_mldata()	ML 웹사이트에서 다운로드

# 분류와 클러스터링을 위한 표본 데이터 생성기 + etc...

API 명	설명
datasets.make_classification()	분류를 위한 데이터 세트를 만듭니다. 특히 높은 상관도, 불필요한 속성 등의 노이즈 효과를 위한 데이터를 무작위로 생성해 줍니다.
datasets.make_blobs()	클러스터링을 위한 데이터 세트를 무작위로 생성해줌 군집 지정 개수에 따라 여러 가지 클러스터링을 위한 데이터 세트를 쉽게 생성해줌

# #2.3 사이킷런의 기반 프레임워크 익히기

# 분류나 회귀를 위한 연습용 예제 데이터의 구성

- 일반적으로 딕셔너리 형태
- 키는 일반적으로 dat, target, target\_name, feature\_names, DESCR로 구성

# 개별 키가 가리키는 데이터 세트의 의미

data	뉴스 그룹 텍스트 자료
target	얼굴 이미지 자료
target_names	얼굴 이미지 자료
feature_names	얼굴 이미지 자료
DESCR	로이터 뉴스 말뭉치

- # data, target – 넘파이 배열
- # target\_names, feature\_names – 넘파이 배열 or 파이썬 리스트
- # DESCR – 스트링 타입

# 피처의 데이터 값 반환: 내장 데이터 세트 API 호출 → Key 값 지정

# #2.4 Model Selection 모듈 소개

# module\_selection: 학습 데이터, 테스트 데이터 세트 분리 or 교차 검증 분할 및 평가, Estimator의 하이퍼 파라미터 튜닝을 위한 다양한 함수, 클래스 제공

# 학습/테스트 데이터 세트 분리 - train\_test\_split()

# 테스트 데이터 세트를 이용하지 않고 학습 데이터 세트로만 학습하고 예측할 시 문제점?  
→ 이미 학습한 학습 데이터 세트를 기반으로 예측하여 정확도가 100%로 출력됨

# 학습 및 테스트 데이터 세트로 분리하는 법

sklearn.model\_selection 모듈에서 train\_test\_split 로드  
→ 첫 번째 파라미터로 피쳐 데이터 세트, 두 번째 파라미터로 레이블 데이터 세트 입력받음  
→ 이후 선택적으로 다음 파라미터 입력

test_size	전체 데이터에서 테스트 데이터 세트 크기를 얼마로 샘플링할지 결정, default = 0.25 (25%)
train_size	전체 데이터에서 학습용 데이터 세트 크기를 얼마로 샘플링할지 결정, 잘 사용 X
shuffle	데이터를 분리하기 전에 데이터를 미리 섞을지를 결정함, default = True, 데이터 분산시켜서 좀 더 효율적인 학습 및 테스트 데이터 세트를 만드는 데 사용
random_state()	호출할 때마다 동일한 학습/테스트용 데이터 세트를 생성하기 위해 주어지는 난수 값
train_test_split()	반환값은 튜플 형태, 순차적으로 학습용 데이터의 피쳐 데이터 세트, 테스트용 데이터의 피쳐 데이터 세트, 학습용 데이터의 레이블 데이터 세트, 테스트용 데이터의 레이블 데이터 세트가 반환됨



## #2.4 Model Selection 모듈 소개

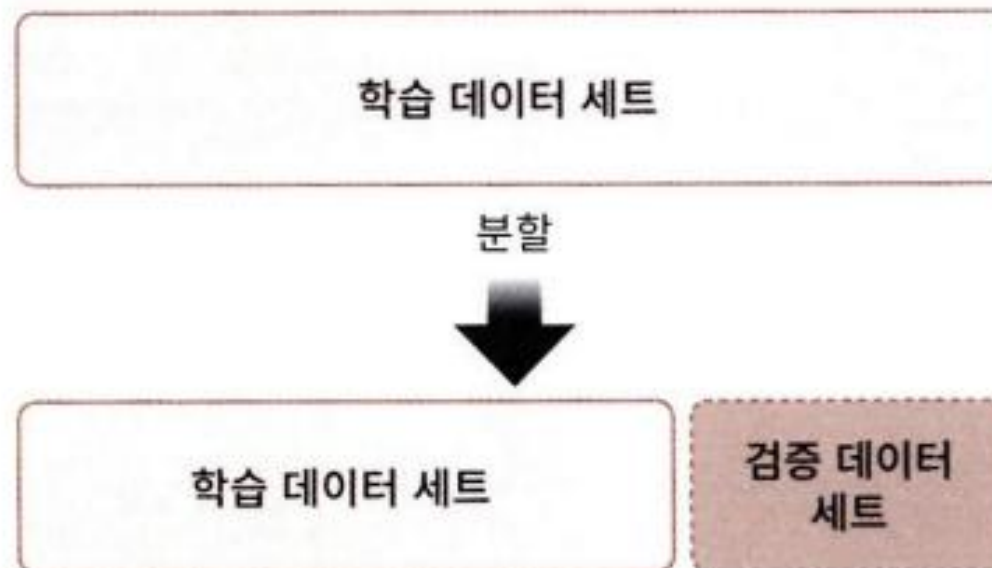
# 별도의 테스트용 데이터를 사용하는 것 역시  
과적합에 취약한 약점을 가질 수 있음

# **과적합**: 모델이 학습 데이터에만 과도하게  
최적화되어, 실제 예측을 다른 데이터로  
수행할 시 예측 성능이 과도하게  
떨어지는 것  
→ 해결을 위해 교차 검증 이용

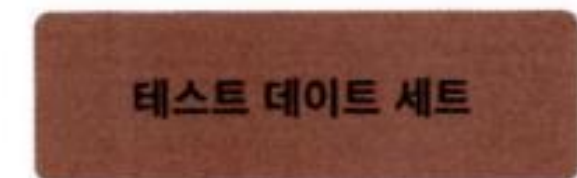
# **교차 검증**: 데이터 편종을 막기 위해 별도의  
여러 세트로 구성된 학습 데이터  
세트와 검증 데이터 세트에서 학습과  
평가를 수행하는 것

# ML 모델의 성능 평가  
교차 검증 기반으로 1차 평가  
→ 최종적으로 테스트 데이터 세트에 적용하여  
평가

학습 데이터를 다시 분할하여 학습 데이터와 학습된  
모델의 성능을 일차 평가하는 검증 데이터로 나눔



모든 학습/검증 과정이 완료된 후 최종적으로  
성능을 평가하기 위한 데이터 세트



## #2.4 Model Selection 모듈 소개

# **K 폴드 교차 검증**: K개의 데이터 폴드 세트를 만들어서 K번만큼 각 폴드 세트에 학습과 검증 평가를 반복적으로 수행하는 방법

ex) 5폴드 교차 검증 ( $K = 5$ )

5개의 폴드된 데이터 세트를 학습과 검증을 위한 데이터 세트로 변경하면서 5번 평가 수행 후 이 5개의 평가를 평균한 결과를 가지고 예측 성능을 평가

데이터 세트 5등분

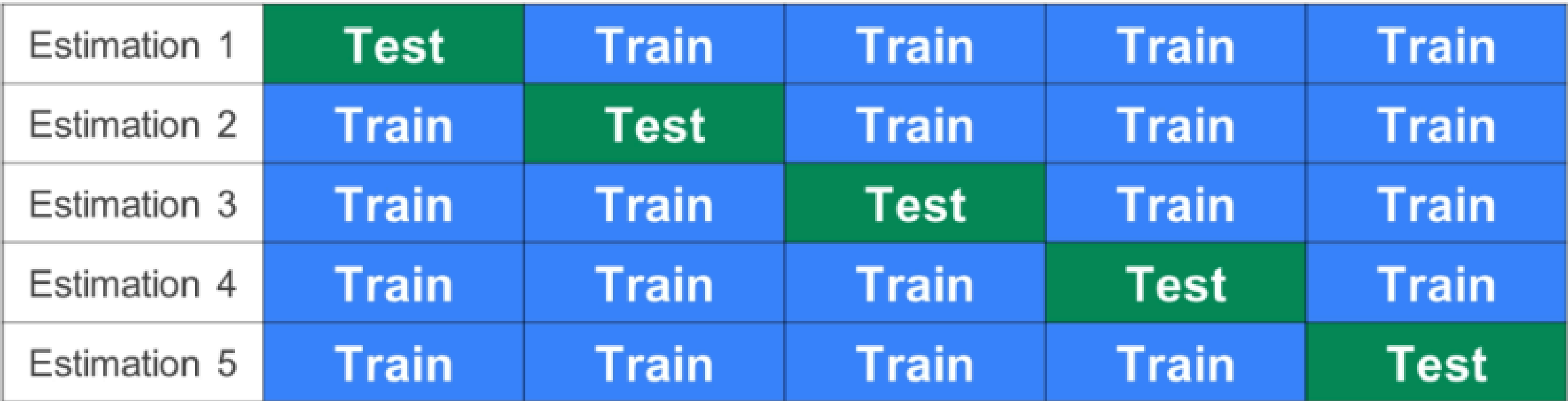
- 첫 번째 반복에서 처음부터 4개 등분을 학습 데이터 세트, 마지막 5번째 등분 하나를 검증 데이터 세트로 설정 후 학습 데이터 세트에서 학습 수행, 검증 데이터 세트에서 평가 수행
- 학습 데이터와 검증 데이터를 점진적으로 변경해서 다시 평가작업 반복 수행
- 마지막 5번째까지 학습과 검증을 수행하여 5개의 예측 평가를 구했다면 이를 평균하여 K 폴드 평가 결과로 반영



# #2.4 Model Selection 모듈 소개

## 5-fold CV

## DATASET



Estimation 1	Test	Train	Train	Train	Train
Estimation 2	Train	Test	Train	Train	Train
Estimation 3	Train	Train	Test	Train	Train
Estimation 4	Train	Train	Train	Test	Train
Estimation 5	Train	Train	Train	Train	Test

## #2.4 Model Selection 모듈 소개

# K 폴드 교차 검증 프로세스 구현 위해 `Kfold`와 `StratifiedKFold` 클래스 제공

# `Kfold` 클래스를 이용하여 교차 검증하고 예측 정확도 알아보기

붓꽃 데이터 세트와 `DecisionTreeClassifier` 생성

→ 5개의 폴드 세트로 분하는 `Kfold` 객체 생성

→ 생성된 객체의 `split()`을 호출하여 전체 붓꽃 데이터를 5개의 폴드 데이터 세트로 분리  
학습용/검증용 데이터로 분할할 수 있는 인덱스 반환

# 교차 검증 시마다 검증 세트의 인덱스가 달라짐

# `Stratified K 폴드`: 불균형한 분포도를 가진 레이블 데이터 집합을 위한 K 폴드 방식

→ K 폴드가 레이블 데이터 집합이 원본 데이터 집합의 레이블 분포를 학습 및 테스트 세트에 제대로 분배하지 못하는 경우의 문제를 해결해 줌

# 사용 방식은 `Kfold` 사용 방법과 거의 비슷하나 `StratifiedKFold`는 레이블 데이터 분포도에 따라 학습/검증 데이터를 나누기 때문에 `split()` 메서드에 인자로 피쳐 데이터 세트뿐만 아니라 레이블 데이터 세트도 반드시 필요로 함

# 회귀에서는 `StratifiedFold` 지원되지 않음

## #2.4 Model Selection 모듈 소개

# 교차 검증을 보다 간편하게 하는 법 - `cross_val_score()` - 하나의 평가 지표만 가능

# Kfold의 과정을 한꺼번에 수행해주는 API

# 선언 형태

```
cross_val_score(estimator, X, y=None, scoring=None, cv=None, n_jobs=1, verbose=0, fit_params=None,
pre_dispatch='2*n_jobs')    ///  
주요 파라미터
```

# estimator: Classifier or Regressor

# X: 피쳐 데이터 세트

# y: 레이블 데이터 세트

# scoring: 예측 성능 평가 지표

# cv: 교차 검증 폴드 수

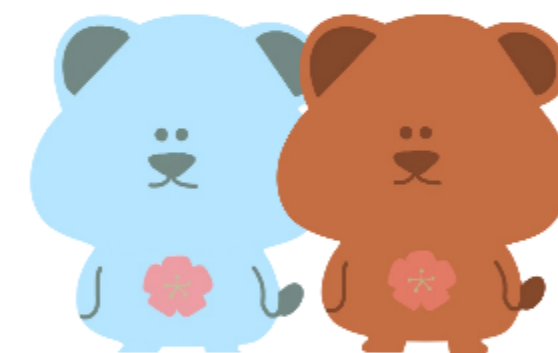
# 수행 후 반환값은 scoring 파라미터로 지정된 성능 지표 측정값을 배열 형태로 반환 (cv로 지정된 횟수만큼)

# classifier가 입력되면 Stratified K 폴딩 아식으로 레이블값 분포 따라서 학습/테스트 세트 분할 (회귀는 불가)

# `cross_val_score()` API는 내부에서 Estimator를 학습, 예측, 평가시켜줌 → 간단하게 교차 검증 수행 가능

# 비슷한 API로 `cross_validate()` 존재 - 여러 개의 평가 지표를 반환 가능 + 학습 데이터에 대한 성능 평가 지표 & 수행 시간 제공

## 2장 후반



# #02 2장 후반

---

#1 GridSearchCV

#2 데이터 인코딩

# #01 GridSearchCV

## GridSearchCV :

하이퍼 파라미터를 순차적으로 입력하면서 최적의 파라미터를 도출할 수 있는 방안을 제공.

ex) mission : 결정 트리 알고리즘의 최적의 파라미터 조합 찾기!

### 1. 파라미터의 집합 만들기 (예시)

```
grid_parameters = { 'max_depth' : [1, 2, 3],  
                    'min_samples_split' : [2, 3]  
                  }
```

# #01 GridSearchCV

## 2. 모든 파라미터를 순차적으로 성능 측정 과정에 적용하기 (예시)

순번	max_depth	min_samples_split
1	1	2
2	1	3
3	2	2
4	2	3
5	3	2
6	3	3

## 3. 파라미터, CV에 따라 학습/평가 후 평균값으로 성능을 측정하기

if)  $cv = 3$  :  
파라미터 조합마다 3개의 폴딩 세트를 3회에 걸쳐 학습/평가  
→ 총  $6 \times 3 = 18$ 회!

# #01 GridSearchCV

## 실제 실행 코드와 결과 & 해석

```
parameters = {'max_depth':[1, 2, 3], 'min_samples_split':[2, 3]} <- 파라미터, 딕셔너리로 선언

grid_dtree = GridSearchCV(dtree, param_grid=parameters, cv=3, refit=True)
grid_dtree.fit(X_train, y_train) <- estimator, parameter 선택.
                                cv(분할) = 3, 최적의 파라미터로 재학습.

scores_df = pd.DataFrame(grid_dtree.cv_results_)
scores_df[['params', 'mean_test_score', 'rank_test_score',
              'split0_test_score', 'split1_test_score', 'split2_test_score']]
```

	params	mean_test_score	rank_test_score	split0_test_score	split1_test_score	split2_test_score
0	{'max_depth': 1, 'min_samples_split': 2}	0.700000	5	0.700	0.7	0.70
1	{'max_depth': 1, 'min_samples_split': 3}	0.700000	5	0.700	0.7	0.70
2	{'max_depth': 2, 'min_samples_split': 2}	0.958333	3	0.925	1.0	0.95
3	{'max_depth': 2, 'min_samples_split': 3}	0.958333	3	0.925	1.0	0.95
4	{'max_depth': 3, 'min_samples_split': 2}	0.975000	1	0.975	1.0	0.95
5	{'max_depth': 3, 'min_samples_split': 3}	0.975000	1	0.975	1.0	0.95

적용된 파라미터

폴딩 테스트 결과 평균값

순위

각 폴딩 테스트 별 점수



# #01 GridSearchCV

## 실제 실행 코드와 결과 & 해석

```
print('GridSearchCV 최적 파라미터:', grid_dtrees.best_params_)
print('GridSearchCV 최고 정확도:{0:.4f}'.format(grid_dtrees.best_score_))

estimator = grid_dtrees.best_estimator_
pred = estimator.predict(X_test)
print('테스트 데이터 세트 정확도: {0:.4f}'.format(accuracy_score(y_test, pred)))
```

```
GridSearchCV 최적 파라미터: {'max_depth': 3, 'min_samples_split': 2}
GridSearchCV 최고 정확도:0.9750
테스트 데이터 세트 정확도: 0.9667
```

best\_params\_ : 학습/평가 후 저장된 최적의 파라미터  
best\_score\_ : 학습/평가 시 얻은 최고 정확도  
best\_estimator\_ : refit 옵션으로 학습된 최적의 estimator

# #02 데이터 인코딩

- ML : 데이터에 기반한 알고리즘
  - > 어떤 데이터를 입력으로 가지는지가 매우 중요!
- 결손값(NaN, NULL)은 허용 x
  - > 다른 값으로 대체하거나 일정 비율 이상일 시 해당 feature를 drop
- - 사이킷런 : 문자열 입력값으로 허용 X
  - > 문자열 값을 숫자 형으로 변환하는 과정 필요
- 문자열 피처의 종류 및 변환:

카테고리형 피처  
-> 코드 값으로  
표현

텍스트형 피처  
-> 벡터화  
or 삭제

# #02 데이터 인코딩

## 레이블 인코딩

- 카테고리 피처를 코드형 숫자 값으로 변환하는 인코딩
- LabelEncoder 클래스를 통해 구현
- 숫자의 크고 작음의 차이가 존재 -> 성능 저하를 일으킬 수 있음!  
-> 회귀에는 사용 x, 트리에는 0

## 원-핫 인코딩

- 피처 값의 유형에 따라 새로운 피처 추가 (행 형태인 고유 값을 열로 차원 변환)  
-> 고유 값 칼럼에 1 표시, 나머지 칼럼 0 표시
- 사이킷런: OneHotEncoder, 판다스 get\_dummies() 통해 구현
- 사이킷런 사용 시 : 문자열 -> 숫자형 변환 필요, 입력 값으로 2차원 데이터 필요

# #02 데이터 인코딩

- 피처 스케일링 : 서로 다른 변수의 값을 일정한 수준으로 맞추는 작업  
→ 표준화(Standardization), 정규화(Normalization)
- 표준화 : 데이터의 피처 각각이 평균이 0이고 분산이 1인  
가우시안 정규 분포를 가진 값으로 변환하는 것
- 정규화 : 서로 다른 피처의 크기를 통일하기 위해 크기를 변환해주는 것  
개별 데이터의 크기를 모두 똑같은 단위로 변경하는 것
- 사이킷런의 Normalizer : 선형대수의 정규화 개념 적용 (위의 정규화와 다름!)  
개별 벡터를 모든 피처 벡터의 크기로 나눠 줌.

$$x_{i\_new} = \frac{x_i - \text{mean}(x)}{\text{stdev}(x)}$$

표준화

$$x_{i\_new} = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

정규화

$$x_{i\_new} = \frac{x_i}{\sqrt{x_i^2 + y_i^2 + z_i^2}}$$

Normalizer

# #02 데이터 인코딩

- StandardScaler : 앞의 표준화를 쉽게 지원하기 위한 클래스  
데이터가 가우시안 정규 분포를 가질 수 있도록 변환해줌.
- MinMaxScaler : 데이터값을 0과 1 사이의 범위의 값으로 변환

## 학습 데이터와 테스트 데이터의 스케일링 변환 시 유의점

- Scaler 객체를 이용해 학습 데이터 세트로 fit()과 transform()을 적용했을 때  
-> 테스트 데이터 세트로 fit 수행 x! 학습 데이터의 기준 정보를 그대로 적용.  
이유 : 둘의 스케일링이 맞지 않을 수 있음.
- fit\_transform() : 같은 이유로 테스트 데이터에서는 절대 사용 x.

# THANK YOU





# 03. 평가

문원정

# # 분류의 성능 평가 지표

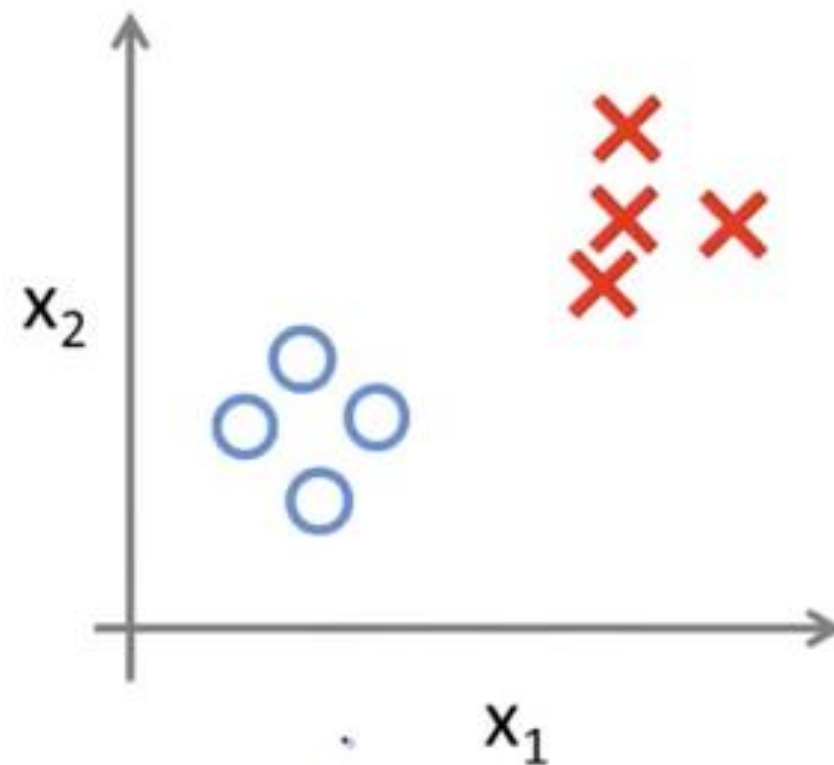
- 정확도 (Accuracy)
- 오차행렬 (Confusion Matrix)
- 정밀도 (Precision)
- 재현율 (Recall)
- F1 스코어
- ROU AUC



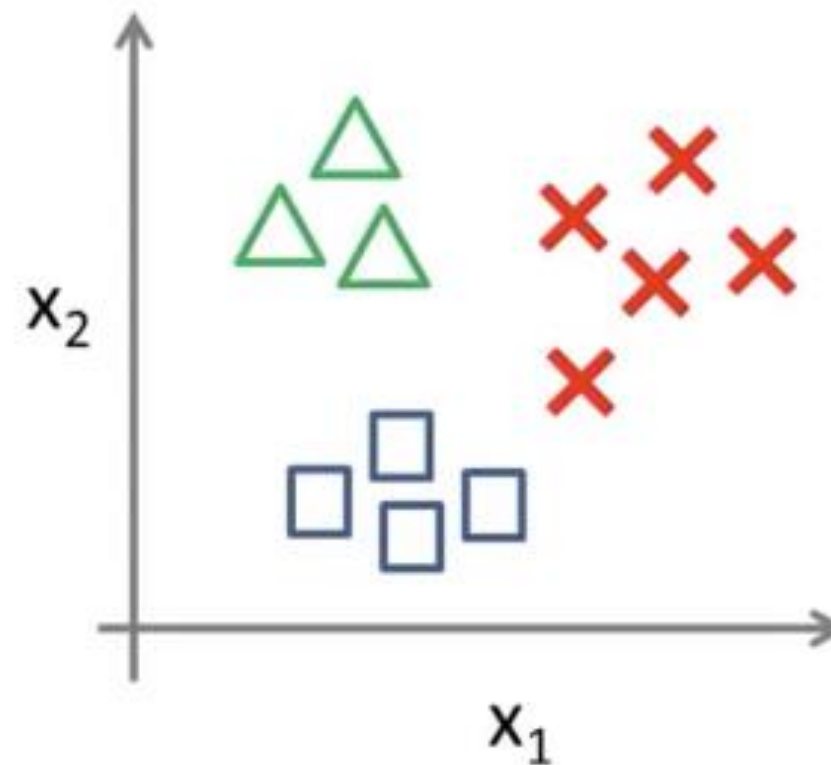
이진 분류 : 긍정/부정과 같은 2개의 결괏값만 가짐

멀티 분류 : 여러 개의 결정 클래스 값을 가짐

Binary classification:



Multi-class classification:



# 01. 정확도 (Accuracy)



# #01 정확도

**정확도** : 실제 데이터에서 예측 데이터가 얼마나 같은지 판단하는 지표

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

**!!** 하지만 이진 분류의 경우 데이터의 구성에 따라 ML 모델의 성능을 왜곡할 수 있기에 정확도 수치 하나만 가지고 성능을 평가하지 X

# #01 정확도

```
from sklearn.base import BaseEstimator class

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

# #01 정확도

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

#원본 데이터를 재로딩, 데이터 가공, 학습 데이터/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=0.2, random_state=0)

#위에서 생성한 DummyClassifier를 이용해 학습/예측/평가 수행
myclf = MyDummyClassifier()
myclf.fit(X_train, y_train)

mypredictions = myclf.predict(X_test)
print('Dummy Classifier의 정확도는: {0:.4f}'.format(accuracy_score(y_test, mypredictions)))
```

[output]

Dummy Classifier의 정확도는: 0.7877

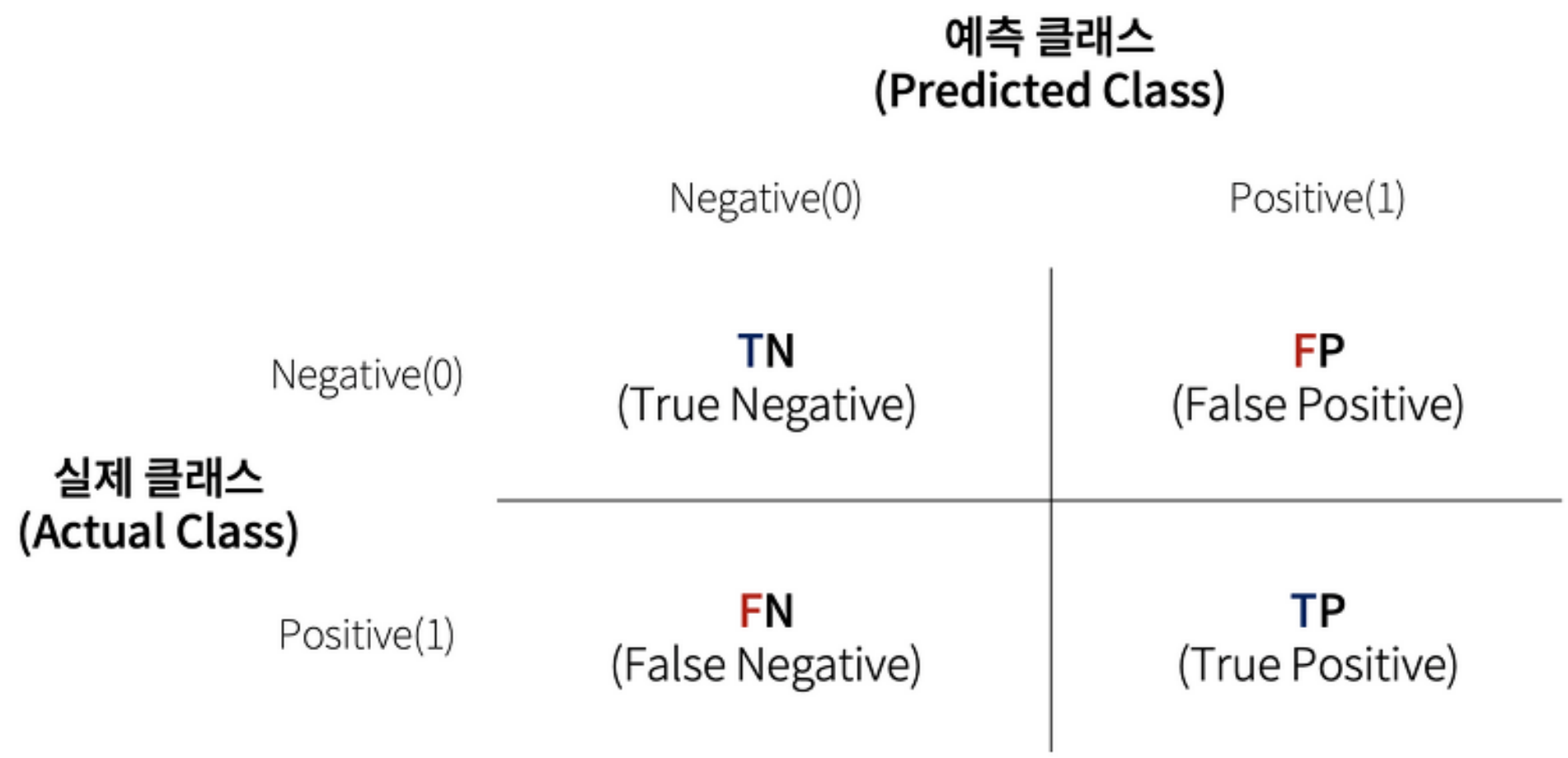
**!! 정확도 결과가 무려 78.77%**

## 02. 오차 행렬

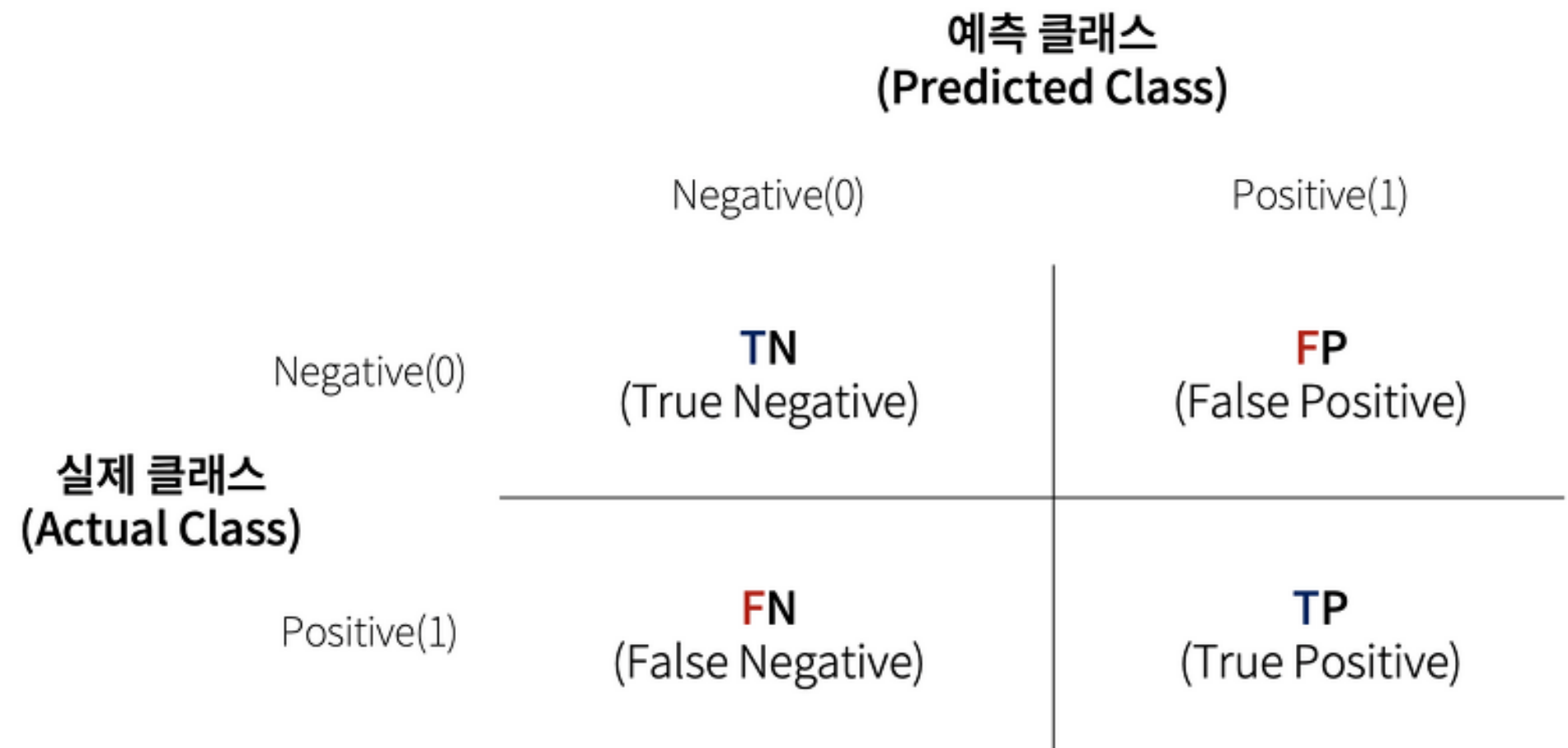


# #02 오차 행렬

**오차 행렬** : 이진분류의 예측 오류가 얼마인지와 더불어 어떠한 유형의 예측 오류가 발생하고 있는지를 함께 나타내주는 지표



# #02 오차 행렬



**TN**은 예측값을 Negative 값인 0으로 예측했고 실제값 또한 Negative 값인 0일 때

**FP**은 예측값을 Positive 값인 1으로 예측했는데 실제값은 Negative 값인 0일 때

**FN**은 예측값을 Negative 값인 0으로 예측했는데 실제값은 Positive 값인 1일 때

**TP**은 예측값을 Positive 값인 1으로 예측했고 실제값 또한 Positive 값인 1일 때



# #02 오차 행렬

```
from sklearn.metrics import confusion_matrix  
  
confusion_matrix(y_test, fakepred)  
  
[output]  
array([[405, 0],  
       [ 45, 0]], dtype=int64)
```

!! MyFakeClassifier의 예측 결과인 fakepred와 실제 결과인 y\_test를 confusion\_matrix()의 인자로 입력해 오차행렬을 배열 형태로 출력

!! 출력된 오차 행렬은 ndarray 형태

# #02 오차 행렬

		예측 클래스	
		Negative	Positive
실제 클래스	Negative	<b>TN</b> 예측: Negative(7이 아닌 Digit) 405개 실제: Negative(7이 아닌 Digit)	<b>FP</b> 예측: Positive(Digit 7) 0 실제: Negative(7이 아닌 Digit)
	Positive	<b>FN</b> 예측: Negative(7이 아닌 Digit) 45개 실제: Positive(Digit 7)	<b>TP</b> 실제: Positive(Digit 7) 0 실제: Positive(Digit 7)

MyFakeClassifier는 target이 7인지 아닌지에 따라 True/False 로 변경한 데이터 세트를 사용해 무조건 Negative로 예측하는 Classifier.

=> 따라서 TN은 전체 데이터 450건 중 무조건 Negative 0으로 예측해서 True가 된 결과 405건, FP,TP는 Positive 1로 예측한 건수가 없으므로 0건, FN은 Positive 1인 건수 45건을 Negative로 예측해서 False가 된 결과 45건

# #02 오차 행렬

!! TN,FP,FN,TP 값을 조합하여 Classifier의 성능을 측정할 수 있는 주요 지표인 정확도, 정밀도, 재현율 값을 알 수 있음.

$$\begin{aligned}\text{정확도} &= \text{예측 결과와 실제 값이 동일한 건수/전체 데이터 수} \\ &= (TN+TP)/(TN + FP + FN + TP)\end{aligned}$$

일반적으로 불균형한 레이블 클래스를 가지는 이진 분류 모델에서는 많은 데이터 중에서 중점적으로 찾아야 하는 적은 결괏값에 Positive를 설정해 1값을 부여하고, 그렇지 않은 경우 Negative로 0 값을 부여하는 경우가 많음.

⇒ 정확도 지표는 positive에 대한 예측 정확도를 판단하지 못한 채 negative에 대한 예측 정확도만으로 분류의 정확도가 매우 높게 나타나는 오류가 일어남

⇒ 정밀도와 재현율 더 선호

## 03. 정밀도와 재현율



# #03 정밀도와 재현율

**정밀도** : 예측을 Positive로 한 대상 중에 예측과 실제 값이 Positive로 일치한 데이터의 비율을 뜻한다.

Positive 예측 성능을 더욱 정밀하게 측정하기 위한 평가지표로 양성 예측도

**재현율** : 실제 값이 Positive한 대상 중에 예측과 실제 값이 Positive로 일치한 데이터의 비율을 뜻한다.

민감도 또는 TPR(True Positive Rate)

$$\text{정밀도} = \text{TP} / (\text{FP} + \text{TP})$$

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

# #03 정밀도와 재현율

!! 이진 분류 모델의 업무 특성에 따라 특정 평가 지표가 더 중요한 지표로 간주될 수 있음

1) **재현율**이 중요 지표인 경우: 실제 Positive 양성 데이터를 Negative로 잘못 판단하게 되면 업무상 큰 영향이 발생하는 경우

ex) 암 판단 모델, 보험, 금융 사기 적발 모델

2) **정밀도**가 중요 지표인 경우: 실제 Negative 음성인 데이터 예측을 Positive 양성으로 잘못 판단하게 되면 업무상 큰 영향이 발생하는 경우

ex) 스팸메일 여부 판단 모델

재현율과 정밀도 모두 TP를 높이는 데 동일하게 초점을 두지만, 재현율은 FN(실제 Positive, 예측 Negative)를 낮추는 데 정밀도는 FP를 낮추는 데 초점을 둔다.

# #03 정밀도와 재현율

!! 사이킷런은 정밀도 계산을 위해 `precision_score()`,  
재현율 계산을 위해 `recall_score()`를 API로 제공

```
from sklearn.metrics import precision_score, recall_score, f1_score
```

```
precision=precision_score(y_test,pred)  
recall=recall_score(y_test,pred)  
f1=f1_score(y_test,pred)
```



# 03. 평가

김정은



# 목차

---

#01 정밀도와 재현율 (2)

#02 F1 스코어

#03 ROC 곡선과 AUC



# #01 정밀도와 재현율 (2)

## #1 정밀도와 재현율 트레이드오프

### 정밀도와 재현율의 트레이드오프

- ➔ 정밀도와 재현율은 상호 보완적인 평가 지표
- ➔ 어느 한 쪽을 강제로 높이면 다른 하나의 수치는 떨어짐

### Predict\_proba( )

- ➔ 사이킷런 제공, 개별 데이터별로 예측 확률을 반환하는 메서드
- ➔ 학습이 완료된 사이킷런 Classifier 객체에서 호출
- ➔ 테스트 피쳐 데이터 세트를 파라미터로 입력
- ➔ 테스트 피쳐 레코드의 개별 클래스 예측 확률을 반환

입력 파라미터	테스트 피쳐 데이터 세트
반환값	개별 클래스의 예측 확률을 ndarray mxn 형태로 반환 m : 입력 값의 레코드 수   n : 클래스 값 유형

# #01 정밀도와 재현율 (2)

## Predict\_proba( ) 사용 예시

```
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 array와 예측 결과값 array를 병합(concatenate)해 예측 확률과 결과값을 한눈에 반환
pred_proba_result = np.concatenate([pred_proba, pred.reshape(-1,1)], axis=1)
print('두 개의 class 중에서 더 큰 확률을 클래스 값으로 예측 \n', pred_proba_result[:3])
```

- ➔ 반환 결과 ndarray는 0과 1에 대한 확률
- ➔ 첫번째 칼럼과 두번째 칼럼 값을 더하면 1
- ➔ 두 개의 칼럼 중 더 큰 확률로 predict( )가 최종 예측

- ➔ 사이킷런은 분류 결정 임계값을 조절
- ➔ 정밀도와 재현율의 성능 수치를 상호 보완적으로 조정

## Output

```
pred_proba()결과 shape : (179, 2)
pred_proba array에서 앞 3개만 샘플로 추출
: [[0.44935228 0.55064772]
   [0.86335513 0.13664487]
   [0.86429646 0.13570354]]
두 개의 class 중에서 더 큰 확률을 클래스 값으로 예측
[[0.44935228 0.55064772 1.
   [0.86335513 0.13664487 0.
   [0.86429646 0.13570354 0.]
```

# #01 정밀도와 재현율 (2)

## 사이킷런의 정밀도/재현율 트레이드오프 방식

```
from sklearn.preprocessing import Binarizer

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

#predict_proba() 반환값의 두 번째 칼럼, 즉 Positive 클래스 칼럼 하나만 추출해 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)
```

임계값 : 0.5

## Output

- 사이킷런의 predict( )는 predict\_proba( ) 메서드가 반환하는 확률값을 가진 ndarray에서 정해진 임계값을 만족하는 ndarray의 칼럼 위치를 최종 예측 클래스로 결정
- Binarizer 클래스 이용

오차 행렬  
[[108 10]  
[ 14 47]]

정확도: 0.8659, 정밀도: 0.8246, 재현율: 0.7705

# #01 정밀도와 재현율 (2)

## 사이킷런의 정밀도/재현율 트레이드오프 방식

```
# Binarizer의 threshold 설정값을 0.4로 설정. 즉 분류 결정 임계값을 0.5에서 0.4로 낮춘다.  
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)
```

- 임계값의 하락 -> 재현율 상승, 정밀도 하락
- 분류 결정 임계값은 Positive 예측값을 결정하는 확률의 기준이 됨
- 임계값의 하락 -> True 값의 개수가 증가
- Positive 예측값 개수의 증가 -> 재현율 상승
- 실제 양성을 음성으로 예측하는 횟수가 감소

임계값을 수정한다면?

임계값 : 0.4

### Output

오차 행렬

[[97 21]

[11 50]]

정확도: 0.8212, 정밀도: 0.7042, 재현율: 0.8197

# #01 정밀도와 재현율 (2)

## 임계값의 변화에 따른 오차 행렬의 변화

임계값 : 0.5

TN	FP
108	10
FN	TP
14	47

임계값 : 0.4

TN	FP
97	21
FN	TP
11	50

임계값을 0.4에서 0.6까지 0.05씩 증가시키면?

평가 지표	분류 결정 임계값				
	0.4	0.45	0.5	0.55	0.6
정확도	0.8212	0.8547	0.8659	0.8715	0.8771
정밀도	0.7042	0.7869	0.8246	0.8654	0.8980
재현율	0.8197	0.7869	0.7705	0.7377	0.7213

증가  
증가  
감소



# #01 정밀도와 재현율 (2)

## Precision\_recall\_curve()

→ 정밀도, 재현율 값을 반환

입력 파라미터	y_true : 실제 클래스값 배열 ( 배열 크기 = [ 데이터 건수 ] ) probas_pred : Positive 칼럼의 예측 확률 배열 ( 배열 크기 = [ 데이터 건수 ] )
반환값	정밀도 : 임계값별 정밀도 값을 배열로 변환 재현율 : 임계값별 재현율 값을 배열로 변환

## Output

반환된 분류 결정 임계값 배열의 Shape: (147,)

샘플 추출을 위한 임계값 배열의 index 10개: [ 0 15 30 45 60 75 90 105 120 135]

샘플용 10개의 임계값: [0.12 0.13 0.15 0.17 0.26 0.38 0.49 0.63 0.76 0.9 ]

샘플 임계값별 정밀도: [0.379 0.424 0.455 0.519 0.618 0.676 0.797 0.93 0.964 1. ]

샘플 임계값별 재현율: [1. 0.967 0.902 0.902 0.902 0.82 0.77 0.656 0.443 0.213]

임계값 증가 → 정밀도 상승, 재현율 하락

## 코드 예시

```
from sklearn.metrics import precision_recall_curve

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

# 실제값 데이터 세트와 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 임계값 배열의 Shape:', thresholds.shape)

# 반환된 임계값 배열 row가 147건이므로 샘플로 10건만 추출, 임계값을 15step으로 추출
thr_index = np.arange(0, thresholds.shape[0], 15)
print('샘플 추출을 위한 임계값 배열의 index 10개:', thr_index)
print('샘플용 10개의 임계값:', np.round(thresholds[thr_index], 2))

# 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))
```

# #01 정밀도와 재현율 (2)

## Precision\_recall\_curve( ) 를 이용한 정밀도/재현율 곡선

## Output

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

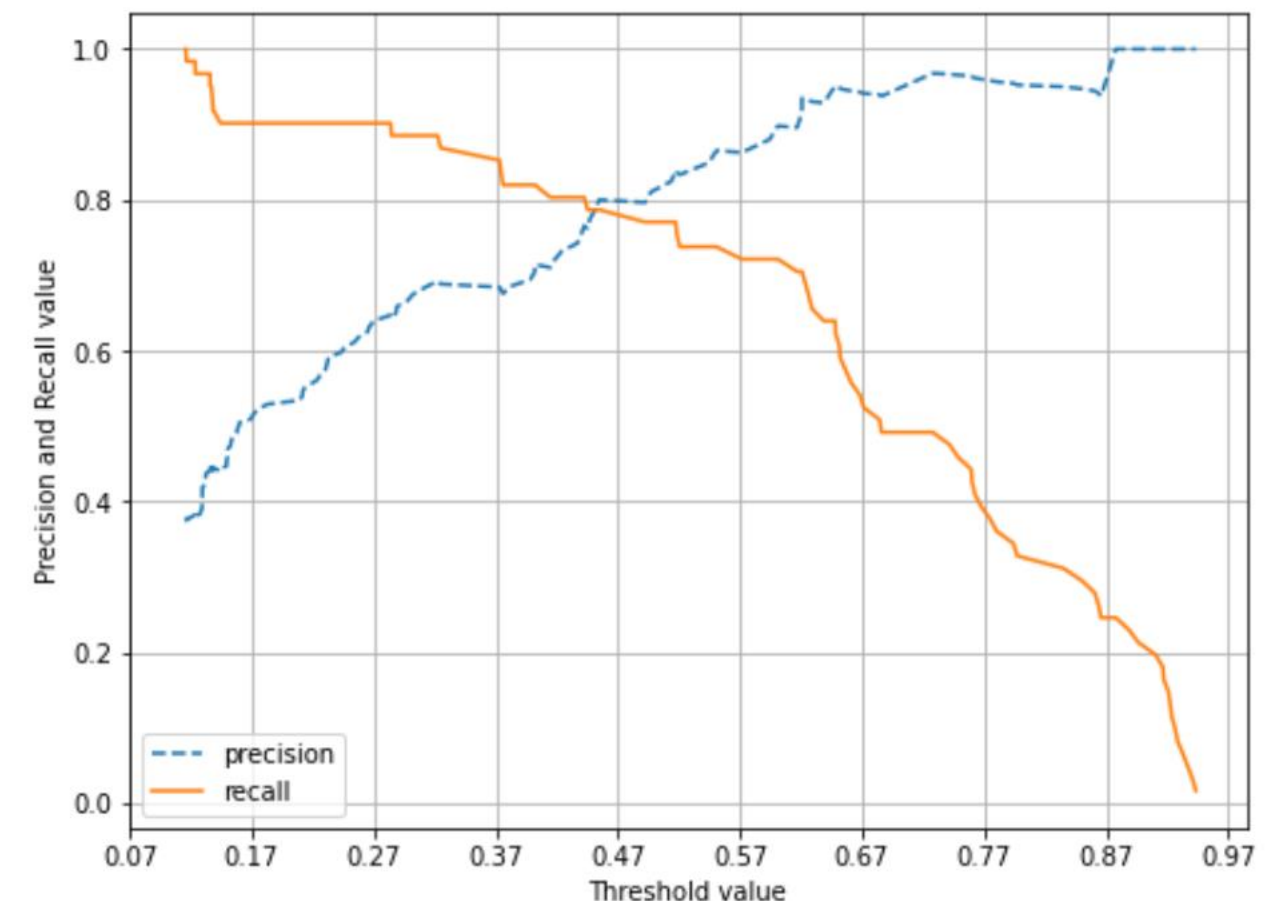
def precision_recall_curve_plot(y_test, pred_proba_c1):
    # threshold ndarray와 threshold에 따른 정밀도, 재현율 ndarray 추출.
    precisions, recalls, thresholds = precision_recall_curve( y_test, pred_proba_c1 )

    # X축을 threshold 값으로, 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] )
```



임계값 감소

→ 많은 수의 양성 예측

→ 재현율 상승, 정밀도 하락



# #01 정밀도와 재현율 (2)

## #2 정밀도와 재현율의 맹점

### 정밀도가 100%가 되는 방법

- 정밀도 =  $TP / (TP + NP)$
- 확실한 기준이 되는 경우만 Positive로 예측, 나머지는 모두 Negative로 예측

### 재현율이 100%가 되는 방법

- 재현율 =  $TP / (TP + FN)$
- 모든 경우를 Positive로 예측

정밀도와 재현율 역시, 어느 한 쪽만 참조하면 극단적인 수치 조작이 가능

정밀도 또는 재현율 중 하나만 스코어가 좋고, 하나는 스코어가 나쁜 경우

→ 성능이 좋지 않은 분류

# #02 F1 Score

## F1 스코어

- 정밀도와 재현율을 결합한 지표
- 어느 한 쪽으로 치우치지 않을 경우, 높은 값을 가짐

## F1 스코어 코드 예시

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

F1 스코어: 0.7966

임계값을 0.4에서 0.6까지 0.05씩 증가시키면?

평가 지표	분류 결정 임계값				
	0.4	0.45	0.5	0.55	0.6
정확도	0.8212	0.8547	0.8659	0.8715	0.8771
정밀도	0.7042	0.7869	0.8246	0.8654	0.8980
재현율	0.8197	0.7869	0.7705	0.7377	0.7213
F1	0.7576	0.7869	0.7966	0.7965	0.800

F1은 임계값 0.6일 때 가장 좋으나, 0.6에서 재현율은 크게 감소 ... 주의 필요

# #03 ROC 곡선과 AUC

## ROC 곡선

- FPR이 변할 때, TPR이 어떻게 변하는지를 나타냄
- $TPR = \text{재현율} = TP / (FN + TP) = \text{민감도}$
- 민감도란, 실제값 Positive가 정확히 예측되어야 하는 수준
- $TNR = \text{특이성} = TN / (FP + TN)$
- 특이성이란, 실제값 Negative가 정확히 예측되어야 하는 수준
- $FPR = FP / (FP + TN) = 1 - TNR = 1 - \text{특이성}$

## ROC 곡선

→ FPR을 0부터 1까지 변경하면서  
TPR의 변화 값을 구한 것

## roc\_curve ( )

입력 파라미터	$y\_true$ : 실제 클래스값 array ( array shape = [ 데이터 건수 ] ) $y\_score$ : predict_proba( )의 반환값 array에서 Positive 칼럼의 예측 확률이 보통 사용됨 array, shape = [ n_samples ]
반환값	$fpr$ : fpr 값을 array로 반환 $tpr$ : tpr 값을 array로 반환 thresholds = threshold 값 array

# #03 ROC 곡선과 AUC

## ROC 곡선 코드 예시(FPR, TPR, 임계값)

```
from sklearn.metrics import roc_curve

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

fprs, tprs, thresholds = roc_curve(y_test, pred_proba_class1)
# 반환된 임계값 배열에서 샘플로 데이터를 추출하되, 임계값을 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 임계값 배열의 index:', thr_index)
print('샘플 index로 추출한 임계값:', np.round(thresholds[thr_index], 2))

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

## Output

```
샘플 추출을 위한 임계값 배열의 index: [ 1  6 11 16 21 26 31 36 41 46]
샘플 index로 추출한 임계값: [0.94 0.73 0.62 0.52 0.44 0.28 0.15 0.14 0.13 0.12]
샘플 임계값별 FPR: [0.    0.008 0.025 0.076 0.127 0.254 0.576 0.61  0.746 0.847]
샘플 임계값별 TPR: [0.016 0.492 0.705 0.738 0.803 0.885 0.902 0.951 0.967 1.    ]
```

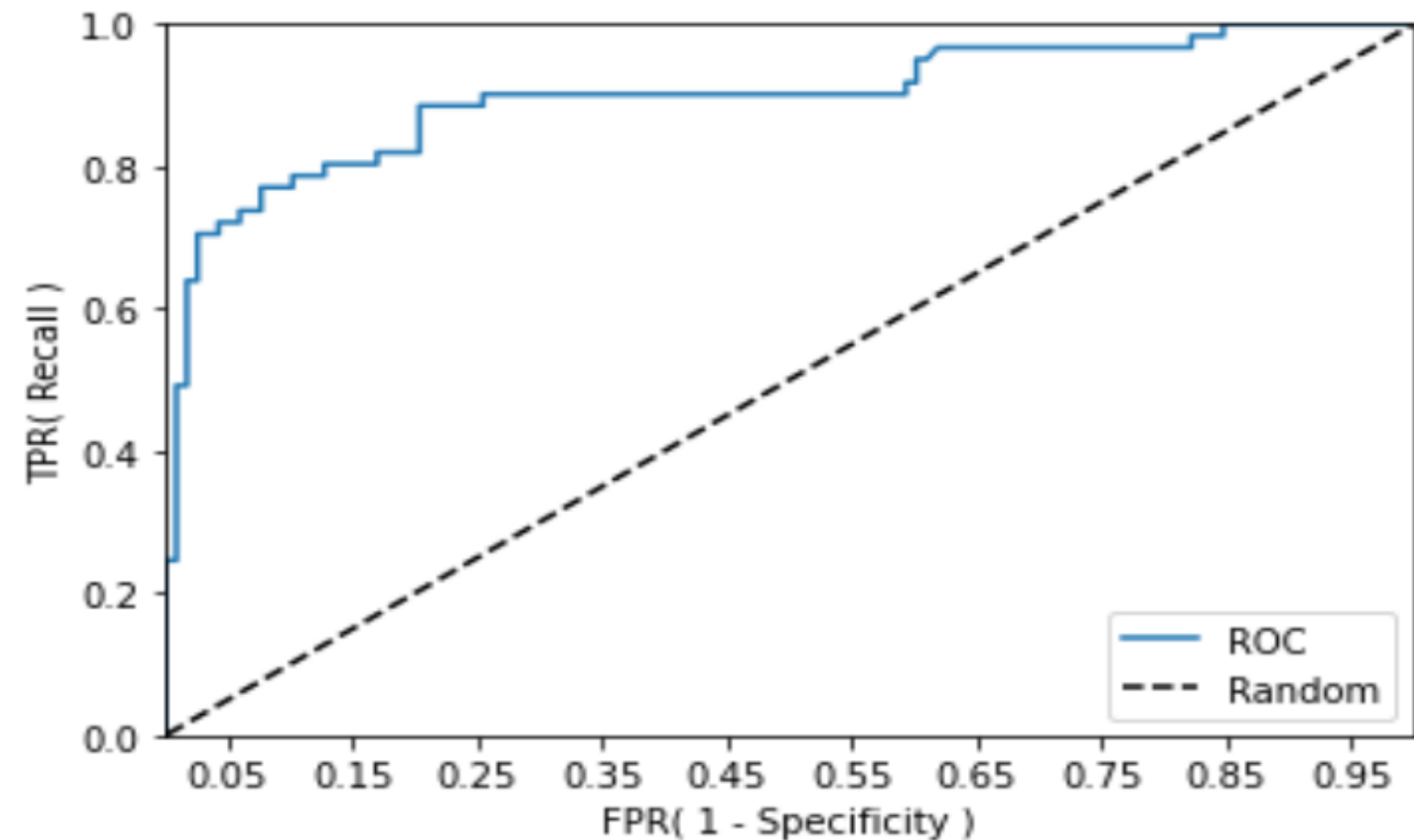
- 임계값이 1에 가까운 값에서 점점 감소
- FPR이 점점 증가
- FPR이 조금씩 커질 때, TPR은 가파르게 증가

# #03 ROC 곡선과 AUC

## ROC 곡선 시각화 코드

```
def roc_curve_plot(y_test, pred_proba_c1):  
    #임꺽값에 따른 FPR, TPR 값을 반환받음.  
    fprs, tprs, thresholds = roc_curve(y_test, pred_proba_c1)  
    #ROC 곡선을 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 0.1 단위로 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])
```

## Output



- ➔ 일반적으로, ROC 곡선은 FPR과 TPR의 변화를 보는 데 사용
- ➔ 분류의 성능 지표 -> ROC 곡선 면적에 기반한 AUC 값

## AUC(Area Under Curve)

- ➔ ROC 곡선 밑 면적, 일반적으로 1에 가까울수록 좋은 수치
- ➔ 보통 분류는 0.5 이상의 AUC 값을 가짐