사이킷런

1. 사이킷런 소개와 특징

```
import sklearn

print(sklearn.__version__)
```

2. ML 실습- 붓꽃 품종 예측하기

- **분류(Classification)**: 지도학습(Supervised Learning) 방법의 하나

▼ **지도학습(Supervised Learning)**

: 정답이 주어진 데이터를 학습한 후, 미지의 정답을 예측하는 방식의 학습

- 학습 데이터 세트: 학습을 위한 데이터 세트
- 테스트 데이터 세트: 모델 성능 예측을 평가하기 위한 데이터 세트

▼ module

이름	설명
sklearn.datasets	dataset 생성
sklearn.tree	tree 기반 ML 알고리즘 구현 class
sklearn.model_selection	데이터 분리 or 최적 하이퍼 파라미터로 모델을 평가하기 위한 module

*하이퍼 파라미터: 알고리즘별 최적의 학습을 위해 직접 입력하는 파라미터

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

▼ dataset

```
import pandas as pd

iris = load_iris  # load dataset

iris_data = iris.data  # feature of Iris dataset

iris_label = iris.target  # label(결정값) of Iris dataset
print('iris target값: ',iris_label)
print('iris target명: ',iris.target_names)

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

1. 데이터 세트 분리하기
   - train_test_split()

```
X_train, X_test, y_train, y_test = train_test_split(iris_data, iris_label,
                                                    test_size=0.2, random_
```

이름	설명
iris_data	feature dataset
iris_label	label dataset (결정값)
test_size	테스트 데이터 세트의 비율
random_state	매번 같은 학습/테스트 데이터 세트 생성 (seed)
X_train, X_test	학습/테스트 feature dataset
y_train, y_test	학습/테스트 label dataset

2. 모델 학습하기

```
dt_clf = DecisionTreeClassifier(random_state=11)  # DecisionTreeClassifier
```

   - fit()

```
dt_clf.fit(X_train, y_train)  # 학습하기
```

3. 예측하기
   - predit()

```
pred = dt_clf.predict(X_test)
```

4. 예측 성능 평가하기
   - accuracy_score() (정확도) : 예측 값이 실제 값과 얼마나 일치하는지
     - accuracy_score(실제 dataset, 예측 dataset)

```
from sklearn.metrics import accuracy_score

print('예측 정확도: {0:.4f}.format(accuracy_score(y_test,pred)))
```

3. 사이킷런의 기반 프레임워크

▼ Estimator, fit() & predict() 메서드

- Estimator
  - 지도 학습의 모든 알고리즘을 구현한 클래스

    cf. 비지도 학습
    - ex. 차원 축소, 클러스터링, 피처 추출(Feature Extraction) 등

        - fit(), transform() 적용
          (+ fit_transform() 적용 가능)
    - Classifier: 분류 알고리즘 구현 클래스
      Regressor: 회귀 알고리즘 구현 클래스
    - fit(), predict() 구현
        - fit(): ML 모델 학습
          predict(): 학습된 모델의 예측

▼ 사이킷런 주요 module

분류	이름	설명
예제 데이터	sklearn.datasets	내장된 데이터 세트
피처 처리	sklearn.preprocessing	데이터 전처리에 필요한 가공 기능 (정규화, 스케일링, 데이터 유형 변환 etc)
	sklearn.feature_selection	알고리즘에 영향을 미치는 피처 순으로 셀렉션 작업을 수행하는 기능
	sklearn.feature_extraction	텍스트 / 이미지 데이터의 벡터화된 피처 추출하는데 사용
피처 처리 & 차원 축소	sklearn.decomposition	차원 축소 기능 수행
데이터 분리, 검증 & 파라미터 튜닝	sklearn.model_selection	교차 검증을 위한 학습/테스트 데이터 분리 최적 파라미터 추출 by GridSearch
평가	sklearn.metrics	분류, 회귀, 클러스터링, 페어와이즈에 대한 성능 측정 기능 제공
ML 알고리즘	sklearn.ensemble	앙상블 알고리즘 - 랜덤 포레스트, 에이다 부스트, 그래디언트 부스팅 등
	sklearn.linear_model	회귀 관련 알고리즘(선형, 로지스틱 회귀, RIdge, Lasso 등), SGD(Stochastic Gradient Descent) 관련 알고리즘 제공
	sklearn.naive_bayes	나이브 베이즈 알고리즘 제공
	sklearn.neighbors	최근접 이웃 알고리즘 제공
	sklearn.svm	서포트 벡터 머신 알고리즘 제공
	sklearn.tree	의사 결정 트리 알고리즘 제공
	sklearn.cluster	비지도 클러스터링 알고리즘 제공
유틸리티	sklearn.pipeline	변환, ML 알고리즘 학습, 예측 등 함께 묶어서 실행하는 유틸리티 제공

▼ 내장된 예제 데이터 세트

API 명	설명
datasets.make_classifications()	분류 데이터 세트 생성
datasets.make_blobs()	클러스터링 데이터 세트 무작위 생성

key 명	설명
data	피처 데이터 세트
target	분류: 레이블 값 회귀: 결과값 데이터
target_names	개별 레이블 이름
feature_names	피처 이름
DESCR	dataset에 대한 설명, feature에 대한 설명

```
from sklearn.datasets import load_iris

iris_data = load_iris()
print(type(iris_data))
```

```
keys = iris_data.keys()
print('붓꽃 데이터 세트의 키들:', keys)
```

```
print('\n feature_names의 type:', type(iris_data.feature_names))
print(' feature_names의 shape:', len(iris_data.feature_names))
print(iris_data.feature_names)

print('\n target_names의 type:', type(iris_data.target_names))
print(' target_names의 shape:', len(iris_data.target_names))
print(iris_data.target_names)

print('\n data의 type:', type(iris_data.data))
print(' data 의 shape:', iris_data.data.shape)
print(iris_data['data'])

print('\n target의 type:', type(iris_data.target))
print(' target 의 shape:', iris_data.target.shape)
print(iris_data.target)
```

4. Model Selection module

▼ train_test_split()

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

iris = load_iris()
```

```
dt_clf = DecisionTreeClassifier()
train_data = iris.data
train_label = iris.target
dt_clf.fit(train_data, train_label)

# 학습 데이터로 예측 수행
pred = dt_clf.predict(train_data)
print('예측 정확도:', accuracy_score(train_label, pred))
```

- parameter

이름	설명
feature dataset	(필수 입력값)
label dataset	(필수 입력값)
test_size	테스트 데이터 세트 크기
train_size	학습용 데이터 세트 크기
shuffle	분리하기 전 데이터를 섞을지 여부 (데이터 분산 시키기) * True (default)
random_state	매번 동일한 학습/테스트 데이터 생성

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

iris_data = load_iris()
dt_clf = DecisionTreeClassifier()

X_train, X_test, y_train, y_test = train_test_split(iris_data.data, iris_d
```

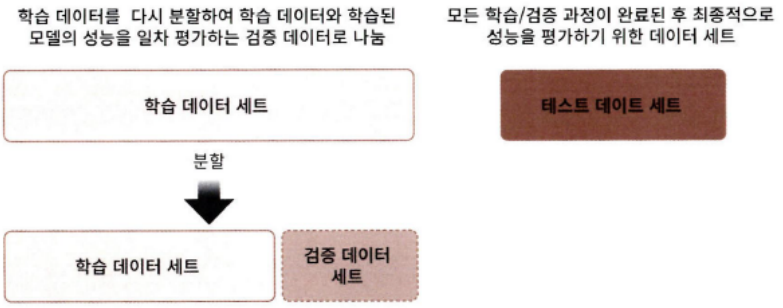
```
# 테스트 데이터로 예측 수행
dt_clf.fit(X_train, y_train)
pred = dt_clf.predict(X_test)
print('예측 정확도:', accuracy_score(y_test, pred))
```

▼ 모델 평가

1. **교차 검증**

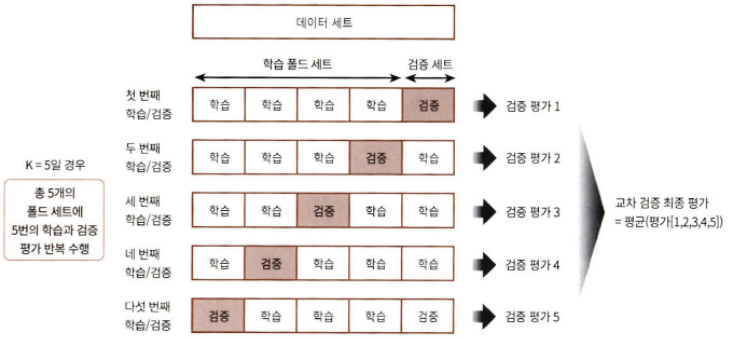
- 문제점: 고정된 학습/테스트 데이터로 평가 → 학습/테스트 데이터에만 적합
  ⇒ 교차 검증!

  * 과적합: 모델이 학습 데이터에만 과도하게 맞춰짐



a. **K 폴드 교차 검증**

- K개의 데이터 폴드 세트 생성 → K번 학습&검증 반복
- Step:
  1. 데이터 세트 K등분
  2. K개의 등분 중 하나씩 돌아가며 검증 데이터로 설정
  3. K개의 예측 평가 평균을 구함




```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import KFold
import numpy as np

iris_data = load_iris()
features = iris.data
label = iris.target
df_clf = DecisionTreeClassifier(random_state=156)

# KFold: 5개의 폴드 세트로 분리, list: 세트별 정확도 담기
kfold = KFold(n_splits=5)
cv_accuracy = []
print('붓꽃 데이터 세트 크기:', features.shape[0])
```

```
n_iter = 0
```

```
# .split(): fold별 학습용, 검증용 row index를 array로 변환함
for train_index, test_index in kfold.split(features):
  # .split()으로 반환된 인덱스로 학습용, 검증용 data 추출
  X_train, X_test = features[train_index], features[test_index]
  y_train, y_test = label[train_index], label[test_index]
  # 학습, 예측
  dt_clf.fit(X_train, y_train)
  pred = dt_clf.predict(X_test)
  n_iter += 1
  # 정확도
  accuracy = np.round(accuracy_score(y_test, pred),4)
  train_size = X_train.shape[0]
  test_size = X_test.shape[0]
  print('\n#{0} 교차 검증 정확도 :{1}, 학습 데이터 크기: {2}, 검증 데이터 크기
  print('#{0} 검증 세트 인덱스:{1}'.format(n_iter, test_index))
  cv_accuracy.append(accuracy)

# iteration별 정확도의 평균 계산
print('\n## 평균 검증 정확도:', np.mean(cv_accuracy))
```

- 문제점: 전체 레이블의 분포 반영하지 못함
  ⇒ Stratified K 폴드!

b. **Stratified K 폴드**

- 불균형한 분포일 때 주로 사용
  *불균형 분포: 특정 label이 매우 많거나 적은 분포
- 분류(Classification)에서 사용되어야 함

```
import pandas as pd

iris = load_iris()
iris_df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
iris_df['label']=iris.target
iris_df['label'].value_counts()
```

```
# KFold로 데이터 분할
kfold = KFold(n_splits=3)
n_iter = 0

for train_index, test_index in kfold.split(iris_df):
  n_iter += 1
  label_train = iris_df['label'].iloc[train_index]
  label_test = iris_df['label'].iloc[test_index]
  print('## 교차 검증: {0}'.format(n_iter))
  print('학습 레이블 데이터 분포:\n', label_train.value_counts())
  print('검증 레이블 데이터 분포:\n', label_test.value_counts())
```

```
# StratifiedKFold로 데이터 분할
from sklearn.model_selection import StratifiedKFold

skf = StratifiedKFold(n_splits=3)
n_iter = 0

for train_index, test_index in skf.split(iris_df, iris_df['label']):
  n_iter += 1
  label_train = iris_df['label'].iloc[train_index]
  label_test = iris_df['label'].iloc[test_index]
  print('## 교차 검증: {0}'.format(n_iter))
  print('학습 레이블 데이터 분포:\n', label_train.value_counts())
  print('검증 레이블 데이터 분포:\n', label_test.value_counts())
```

```
dt_clf = DecisionTreeClassifier(random_state=156)

skfold = StratifiedKFold(n_splits=3)
n_iter = 0
cv_accuracy = []


for train_index, test_index in skfold.split(features, label):
  # .split()으로 반환된 인덱스로 학습용, 검증용 data 추출
  X_train, X_test = features[train_index], features[test_index]
  y_train, y_test = label[train_index], label[test_index]
  # 학습, 예측
  dt_clf.fit(X_train, y_train)
  pred = dt_clf.predict(X_test)
  n_iter += 1
  # 정확도
  accuracy = np.round(accuracy_score(y_test, pred),4)
  train_size = X_train.shape[0]
  test_size = X_test.shape[0]
  print('\n#{0} 교차 검증 정확도 :{1}, 학습 데이터 크기: {2}, 검증 데이터 크기
  print('#{0} 검증 세트 인덱스:{1}'.format(n_iter, test_index))
  cv_accuracy.append(accuracy)

# iteration별 정확도의 평균 계산
print('\n## 교차 검증별 정확도:', np.round(cv_accuracy,4))
print('## 평균 검증 정확도:', np.mean(cv_accuracy))
```

▼ cross_val_score()

- 다음 기능을 한꺼번에 수행해줌
  1. fold 세트 설정
  2. for 루프로 반복하여 학습/테스트 데이터 인덱스 추출

3. 학습/예측 수행 → 성능 반환

cross_val_score(estimator, X, y=None, scoring=None, cv=None, n_jobs=1, verbose=0, fit_params=None, pre_dispatch='2*n_jobs'). 이 중 estimator, X, y, scoring, cv가 주요 파라미터입니다.

- parameter

이름	설명
estimator	Classifier/Regressor
X, y	feature, label dataset
scoring	예측 성능 평가 지표
cv	교차 검증 fold 수

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score, cross_validate
from sklearn.datasets import load_iris

iris_data = load_iris()
df_clf = DecisionTreeClassifier(random_state=156)

data = iris_data.data
label = iris_data.target

scores = cross_val_score(dt_clf, data, label, scoring='accuracy', cv
print('교차 검증별 정확도:', np.round(scores,4))
print('평균 검증 정확도:', np.round(np.mean(scores),4))
```

cf. cross_validate(): 여러 개의 평가 지표 반환

2. **GridSearchCV**

- 교차 검증 & 최적 하이퍼 파라미터 튜닝

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

- parameter

이름	설명
estimator	classifier/regressor/pipeline
param_grid	key + list - 파라미터 이름과 값을 지정
scoring	예측 성능 평가 지표
cv	분할 학습/테스트 개수
refit	최적의 하이퍼 파라미터 찾은 뒤 estimator 객체를 해당 파라미터로 학습 시킴 (default: True)

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

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris_data.data, iri
                                        test_size=0.2, random_state=121)
dtree = DecisionTreeClassifier()

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

```
import pandas as pd

# param_grid의 하이퍼 파라미터를 3개의 train, test fold로 나눠서 테스트 수행함
# refit=True: 가장 좋은 파라미터로 재학습 (default)
grid_dtree = GridSearchCV(dtree, param_grid=parameters, cv=3, refit=Tru

grid_dtree.fit(X_train, y_train)

# GridSearchCV 결과 추출
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']]
```

【Output】

	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}	0.966667	1	0.950	1.0	0.95
5	{'max_depth': 3, 'min_samples_split': 3}	0.966667	1	0.950	1.0	0.95

- column 별 의미

이름	설명
params	적용된 하이퍼 파라미터 값
rank_test_score	성능이 좋은 score 순위 - rank=1: 최적의 하이퍼 파라미터
mean_test_score	하이퍼 파라미터별 CV 폴딩 테스트 세트의 평가 평균값

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

```
# refit=True 로 이미 학습된 estimator 반환
estimator = grid_dtree.best_estimator_
```

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

5. 데이터 전처리

- 문자열 값은 입력 값으로 허용하지 않음
  ⇒ 숫자 형으로 바꿔주기
- 문자열 피처: 카테고리형 / 텍스트형 피처
  ex. 주민번호, 문자열 아이디 (식별자 피처) - 인코딩하지 않고 삭제함

▼ **데이터 인코딩**

1. **레이블 인코딩 (Label encoding)**
   - 카테고리 피처 → 숫자 값

```
from sklearn.preprocessing import LabelEncoder

items=['TV','냉장고','전자레인지','컴퓨터','선풍기','선풍기','믹서','믹서']

encoder = LabelEncoder()
encoder.fit(items)
labels = encoder.transform(items)
print('인코딩 변환값:', labels)
```

```
print('인코딩 클래스:', encoder.classes_)
```

```
print('디코딩 원본값:', encoder.inverse_transform([4,5,2,0,1,1,3,3]))
```

   - 문제점: 변환된 숫자 값이 더 크면 가중치가 부여되거나 중요하게 인식 ⇒ 원-핫 인코딩
     - 선형 회귀에 적용 no!
     - 트리 계열 알고리즘은 문제 없음
2. **원-핫 인코딩 (One Hot encoding)**

   a. OneHotEncoder
   - 고유 값에 해당하는 칼럼: 1, 나머지 칼럼: 0
   - 주의점:
     a. 숫자형으로 변환해야 함
     b. 입력값: 2차원 데이터

```
from sklearn.preprocessing import OneHorEncoder
import numpy as np

items=['TV','냉장고','전자레인지','컴퓨터','선풍기','선풍기','믹서','믹서']
```

```
# 숫자 값으로 변환
encoder = LabelEncoder()
encoder.fit(items)
# 2차원 데이터로 변환
labels = labels.reshape(-1,1)

# 원-핫 인코딩 적용
oh_encoder = OneHotEncoder()
oh_encoder.fit(labels)
oh_labels = oh_encoder.transform(labels)
print('원-핫 인코딩 데이터')
print(oh_labels.toarray())
print('원-핫 인코딩 데이터 차원')
print(oh_labels.shape)
```

b. get_dummies()

- 숫자형으로 변환하지 않아도 됨

```
import pandas as pd

df = pd.DataFrame({'items':['TV','냉장고','전자레인지','컴퓨터','선풍기','
pd.get_dummies(df)
```

▼ 피처 스케일링과 정규화

- 피처 스케일링 (feature scaling)
  - 서로 다른 변수의 값 범위를 일정 수준으로 맞춤

표준화(Standardization)	정규화(Normalization)
데이터 값이 가우시안 정규분포 가진 값으로 변환 * 가우시안 분포: 평균=0, 분산=1	서로 다른 데이터 크기를 모두 같은 단위로 변경
x_new = (x - mean(x))/sd(x)	x_new = (x - min(x))/(max(x) - min(x))

▼ StandardScaler & MinMaxScaler

StandardScaler	MinMaxScaler
표준화	0~1 사이의 값으로 변환
데이터가 가우시안 분포를 갖는다고 가정하고 구현할 때	데이터의 분포가 가우시안 분포가 아닐 때
- 선형 회귀(Linear Regression) - 로지스틱 회귀(Logistic Regression) - 서포트 벡터 머신 (Support Vector Machine)	

```
# StandardScaler

from sklearn.datasets import load_iris
import pandas as pd
```

```
iris = load_iris()
iris_data = iris.data
iris_df = pd.DataFrame(data=iris_data, columns=iris.feature_names)

print('feature 들의 평균 값')
print(iris_df.mean())
print('\nfeature 들의 분산 값')
print(iris_df.var())
```

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(iris_df)
iris_scaled = scaler.transform(iris_df)

iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_names
print('feature 들의 평균 값')
print(iris_df_scaled.mean())
print('\nfeature 들의 분산 값')
print(iris_df_scaled.var())
```

```
# MinMaxScaler

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
scaler.fit(iris_df)
iris_scaled = scaler.transform(iris_df)

# transform() -> 변환된 데이터 세트는 ndarray로 반환됨 => DataFrame으로 변환
iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_names
print('feature 들의 최솟값')
print(iris_df_scaled.min())
print('\nfeature 들의 최솟값')
print(iris_df_scaled.max())
```

- 데이터 스케일링 변환 시 주의점:
  - 학습/테스트 데이터 각각 fit() 적용하면 스케일링 기준이 달라짐
    ⇒ 학습 데이터에 적용된 기준으로 테스트 데이터에 적용해야 함

```
from sklearn.preprocessing import MinMaxScaler
import numpy as np

# scaler 클래스의 fit(), transform()은 2차원 이상 데이터만 가능함
```

```
train_array = np.arange(0,11).reshape(-1,1)
test_array = np.arange(0,6).reshape(-1,1)
```

```
scaler = MinMaxScaler()

# fit(): train_array 최댓값 10, 최솟값 0으로 설정
scaler.fit(train_array)

# 1/10 scale로 train_array 변환
train_scaled = scaler.transform(train_array)

print('원본 train_array 데이터:', np.round(train_array.reshape(-1),2))
print('Scale된 train_array 데이터:', np.round(train_scaled.reshape(-1),2))
```

```
# fit(): test_array 최댓값 5, 최솟값 0으로 설정
scaler.fit(test_array)

# 1/5 scale로 test_array 변환
test_scaled = scaler.transform(test_array)

print('원본 test_array 데이터:', np.round(test_array.reshape(-1),2))
print('Scale된 test_array 데이터:', np.round(test_scaled.reshape(-1),2))
```

```
scaler = MinMaxScaler()
scaler.fit(train_array)
train_scaled = scaler.transform(train_array)
print('원본 train_array 데이터:', np.round(train_array.reshape(-1),2))
print('Scale된 train_array 데이터:', np.round(train_scaled.reshape(-1),2))

# test_array Scale 변환 - transform()만으로 변환
test_scaled = scaler.transform(test_array)
print('\n원본 test_array 데이터:', np.round(test_array.reshape(-1),2))
print('Scale된 test_array 데이터:', np.round(test_scaled.reshape(-1),2))
```

- fit_transform()은 학습 데이터에만 사용!
- 스케일링 변환 시 tip
  - 스케일링 변환 후 학습/테스트 데이터로 분리
  - 변환을 먼저 할 수 없다면 테스트 데이터에 transform()만 적용

평가

분류 모델 성능 평가

- 분류 문제는 범주형 클래스 레이블 예측함

1. 정확도(Accuracy)

- 정확도(Accuracy) = (예측 결과가 동일한 데이터 수)/(전체 예측 데이터 수)
- if 불균형한 레이블 값을 갖는 분포 → 지표로 적합하지 않음
  - ex. data: 0 (90개/100개), 1 (10개/100개)
    무조건 0으로 예측 → 정확도 90%

```
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=None):
    pass
  def predict(self, X):
    return np.zeros((len(X),1), dtype=bool)

digits = load_digits()

y = (digits.target == 7).astype(int)
X_train, X_test, y_train, y_test = train_test_split(digits.data, y, random
```

```
# 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())

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, fa
```

2. 오차 행렬(Confusion Matrix)

	예측 N	예측 P
실제 N	TN (True Neg)	FP (False Pos)
실제 P	FN (False Neg)	TP (True Pos)

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_test, fakepred)
```

cf. 정확도(Accuracy) = (TN+TP)/(TN+FP+FN+TP)

3. 정밀도와 재현율

이름	정밀도	재현율
수식	TP/(FP+TP)	TP/(FN+TP)
설명	P로 예측한 데이터 중 실제로 P인 비율	실제로 P인 데이터 중 P로 예측한 비율
사례	스팸 메일 분류	암 진단
동의어	양성 예측도	민감도(Sensitivity), TPR(True Positive Rate)

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

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, pr
```

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

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_d

lr_clf = LogisticRegression()

lr_clf.fit(X_train, y_train)
```

```
pred = lr_clf.predict(X_test)
get_clf_eval(y_test, pred)
```

▼ 정밀도/재현율 트레이드오프(Trade-off)

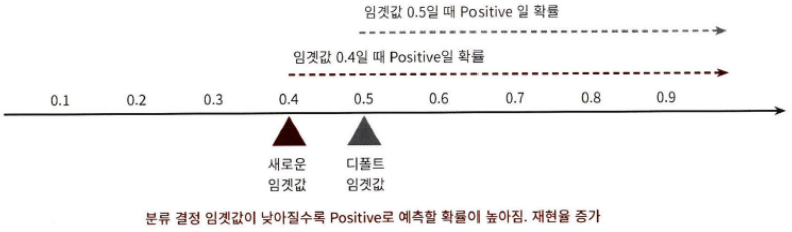
- 어느 한 지표를 강제로 높이면 다른 지표 수치가 떨어진다!
- predict_proba()

입력 파라미터	predict( ) 메서드와 동일하게 보통 테스트 피처 데이터 세트를 입력
반환 값	개별 클래스의 예측 확률을 ndarray m x n (m: 입력 값의 레코드 수, n: 클래스 값 유형) 형태로 반환. 입력 테스트 데이터 세트의 표본 개수가 100개이고 예측 클래스 값 유형이 2개(이진 분류)라면 반환 값은 100 x 2 ndarray임. 각 열은 개별 클래스의 예측 확률입니다. 이진 분류에서 첫 번째 칼럼은 0 Negative의 확률, 두 번째 칼럼은 1 Positive의 확률입니다.

```
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 arrray, 예측 결과값 array 병합
pred_proba_result = np.concatenate([pred_proba, pred.reshape(-1,1)], ax
print('두 개의 class 중에서 더 큰 확률을 클래스 값으로 예측 \n', pred_proba_res
```

- 분류 결정 임곗값 (Threshold) 조절 → 지표 수치 조절함
  - 임곗값 감소 → P 예측값 많아짐 → 재현율 높아짐



분류 결정 임곗값이 낮아질수록 Positive로 예측할 확률이 높아짐. 재현율 증가

- precision_recall_curve()

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

▼ 정밀도/재현율 맹점

1. 정밀도 = 100% ?
   - 확실하게 P일 때만 P로 예측하고 나머지는 모두 N으로 예측한다
     → TP = 1이더라도 FP = 0
     ⇒ 정밀도 = 1/(1+0) = 1

2. 재현율 = 100% ?
    - 모두 P로 예측한다
      → FN = 0
      ⇒ 재현율 = TP/(TP+0) = 1
- 두 지표 모두 높은 수치 얻는다 → Good
  둘 중 하나만 높은 수치를 얻는다 → Not Good
    - 아래의 경우, 임곗값=0.45 가 적당해 보임

평가 지표	분류 결정 임곗값				
	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

4. F1 score

- F1 = 2/(1/정밀도 + 1/재현율)
- 정밀도, 재현율 결합 지표 ⇒ 둘 중 어느 한쪽으로 치우치는 것 막음

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

```
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, precision, 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)
```

평가 지표	분류 결정 임곗값				
	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

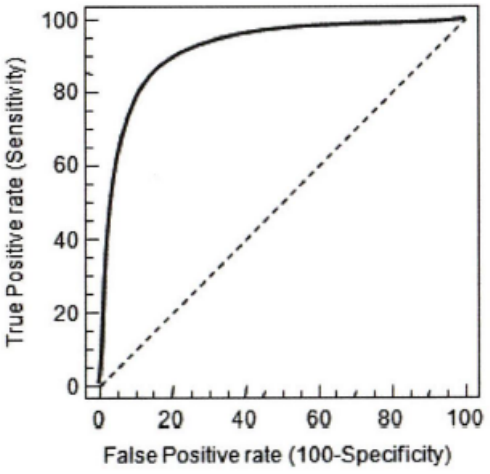
*임곗값 = 0.6 일 때 F1 최댓값 갖지만, 재현율 감소함

5. ROC 곡선, AUC

1. **ROC 곡선**

- FPR이 변할 때 TPR 어떻게 변하는지 나타내는 곡선
  * FPR (False Positive Rate) = 1 - TNR
  * TPR (True Positive Rate, 재현율)

민감도(TPR)	특이성(TNR)
P가 정확히 예측되어야 하는 수준	N이 정확히 예측되어야 하는 수준
TP/(FN+TP)	TN/(FP+TN)
ex. 암에 걸린 사람은 양성 판정	ex. 건강한 사람은 음성 판정



〈 ROC 곡선 예시 〉

- roc_curve()

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

2. **AUC (Area Under Curve)**

- ROC 곡선 아래 면적
  - 1에 가까울 수록 좋은 수치
  - FPR 작을 때 TPR 클 때 → AUC 커짐
- 불균형 데이터에서 주로 사용

```
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('오차 행렬')
    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))
```