Ch.2 사이킷런으로 시작하는 머신러닝
Ch.3 평가

2조 박나림, 강민서, 이가은

EWHA EURON

Ch02. 사이킷런으로 시작하는 머신러닝

EWHA EURON

#2.1 사이킷런 소개와 특징

사이킷런(scikit-learn): 대표적인 파이썬 머신러닝 라이브러리

<특징>

가장 파이썬스러운 API

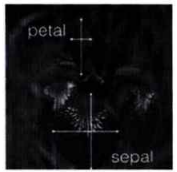
머신러닝을 위한 매우 다양한 알고리즘과 개발을 위한 프레임워크/API 제공

매우 많은 실전 환경에서 사용

#2.2 첫 번째 머신러닝 만들어 보기 – 붓꽃 품종 예측하기

붓꽃 품종 예측: 붓꽃 데이터 세트 속 **Feature**(꽃잎 길이, 너비, 꽃받침 길이, 너비)를 기반으로 붓꽃의 품종을 분류(**Classification**)

붓꽃 데이터 피처



- Sepal length
- Sepal width
- Petal length
- Petal width

붓꽃 데이터 품종(레이블)

Setosa

Vesicolor

Virginica

*Classification: 대표적인 supervised learning의 방법.
*Supervised learning: Label데이터로 모델 학습 후 별도의 test set으로 label을 예측해 모델 성능을 평가함

#2.2 첫 번째 머신러닝 만들어 보기 – 붓꽃 품종 예측하기

sklearn.datasets: 사이킷런에서 자체적으로 제공하는 데이터 세트를 생성하는 모듈의 모임.
sklearn.tree: 트리 기반 ML 알고리즘을 구현한 클래스의 모임.
sklearn.model_selection: 학습, 검증, 예측 데이터로 데이터를 분리하거나 최적의 하이퍼 파라미터로 평가하기 위한 다양한 모듈의 모임.

하이퍼 파라미터: 머신러닝 알고리즘별로 최적의 학습을 위해 직접 입력하는 파라미터들

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

* load_iris(): 붓꽃 데이터 세트, Decision TreeClassifier 채택, train_test_split()함수로 데이터 세트 분리

#2.2 첫 번째 머신러닝 만들어 보기 – 붓꽃 품종 예측하

```python
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)
```

```
iris target값: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
iris target명: ['setosa' 'versicolor' 'virginica']
```

	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

1. **데이터 세트 분리** –학습 데이터와 테스트용 데이터

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

2. **모델 학습** – 학습 데이터를 기반으로 ML 알고리즘을 이용해 모델 학습

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

#학습 수행
dt_clf.fit(X_train, y_train)
```

3. **예측 수행** – 학습된 ML 모델을 이용해 테스트 데이터의 분류를 예측

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

4. **성능 평가** –예측된 결괏값과 테스트 데이터의 실제 결괏값을 비교해 ML 모델 성능을 평가

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

#2.3 사이킷런 기반 프레임워크 익히기

지도 학습

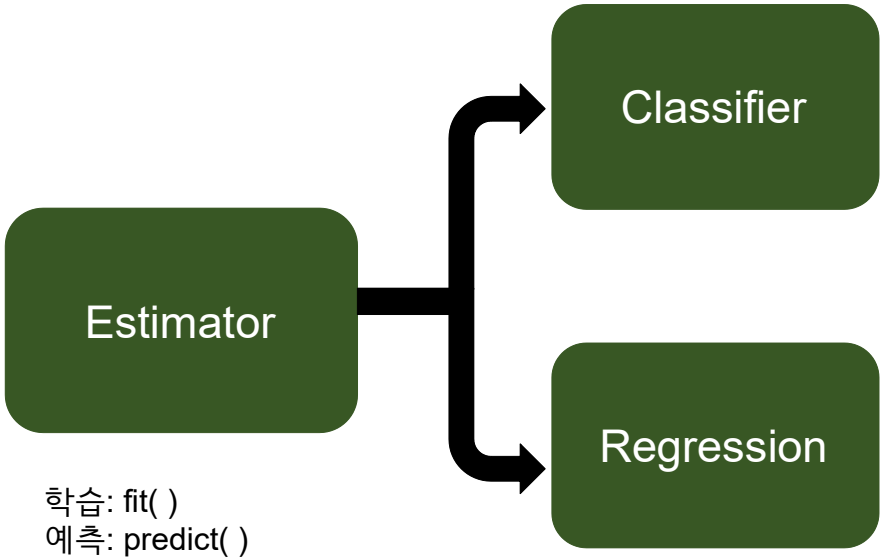
fit( ) – ML 모델 학습
predict( ) – 학습된 모델의 예측

Classifier: 분류 알고리즘 클래스
Regressor: 회귀 알고리즘 클래스
-> Estimator 클래스 – 지도학습의 모든 알고리즘을 구현한 클래스

비지도 학습 / 피처 추출

fit( ) – 입력 데이터의 형태에 맞춰 데이터를 변환하기 위한 사전 구조를 맞추는 작업
transform( ) – 입력 데이터의 차원 변환, 클러스터링, 피처 추출
fit + transform = fit_transform( )

#2.3 사이킷런 기반 프레임워크 익히기



분류 구현 클래스

DecisionTreeClassifier
RandomForestClassifier
GradientBoostingClassifer
GaussianNB
SVC

회귀 구현 클래스

LinearRegression
Ridge
Lasso
RandomForestRegressor
GradientBoostingRegressor

#2.3 사이킷런의 기반 프레임워크 익히기

사이킷런의 주요 모듈

분류	모듈명	설명
예제 데이터	sklearn.datasets	예제 데이터셋
피처 처리	sklearn.preprocessing	데이터 전처리
	sklearn.feature_selection	피처를 우선순위대로 선택
	sklearn.featre_extraction	텍스트, 이미지 데이터의 벡터화된 피처 추출
피처 처리&차원 축소	sklearn.decomposition	차원축소
데이터 분리,검증&파라미터 튜닝	sklearn.model_selection	데이터 분리, 최적 파라니터 추출 API
평가	skleanr.metrics	성능측정법 제공
ML 알고리즘	sklearn.ensemble	앙상블 알고리즘
	sklearn.linear_model	회귀 관련 알고리즘&SGD
	sklearn.naïve_bayes	나이브 베이즈 알고리즘
	sklearn.neighbors	최근접 이웃 알고리즘
	sklearn.svm	서포터 벡터 머신 알고리즘
	sklearn.tree	의사결정트리 알고리즘
	sklearn.cluster	비지도 클러스터링 알고리즘
유틸리티	skleanr.pipeline	변환, 학습, 예측을 함께 묶음

#2.3 사이킷런의 기반 프레임워크 익히기

내장된 예제 데이터 셋

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

fetch_ 인터넷에서 내려받아 scikit_learn_data라는 서브 디렉터리에 저장 후 호출

표본 데이터 생성기

분류와 클러스터링을 위한 표본 데이터 생성기

API 명	설명
datasets.make_classifications( )	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 생성해 줍니다. 군집 지정 개수에 따라 여러 가지 클러스터링을 위한 데이터 세트를 쉽게 만들어 줍니다.

#2.3 사이킷런의 기반 프레임워크 익히기

딕셔너리 형 데이터셋의 키
값

Key	
Data	피처의 데이터 세트
Target	분류 시 레이블 값 / 회귀 시 숫자 결괏값 데이터 세트
Target_name	개별 레이블의 이름
Feature_name	피처의 이름
DESCR	데이터 셋에 대한 설명과 각 피처의 설명

```
from sklearn.datasets import load_iris

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

```
<class 'sklearn.utils._bunch.Bunch'>
```

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

```
붓꽃 데이터 세트의 키들: dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names', 'filename', 'data_module'])
```

EWHA EURON

#2.3 사이킷런 기반 프레임워크 익히기

Load_iris()가 반환하는 붓꽃 데이터 세트의 각 키가 의미하는 값

target_names

setosa, versicolor, virginica

(0 , 1 , 2)

feature_names	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
data	5.1	3.5	1.4	0.2
	4.9	3.0	1.4	0.2
	.....	.....	.....	.....
	4.6	3.1	1.5	0.2
	5.0	3.6	1.4	0.2

target
0
1
.....
2
0

#2.3 사이킷런 기반 프레임워크 익히기

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

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

```
feature_names의 type: <class 'list'>
feature_names 의 shape: 4
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']

target_names의 type: <class 'numpy.ndarray'>
target_names의 shape: 3
['setosa' 'versicolor' 'virginica']

data 의 type: <class 'numpy.ndarray'>
data 의 shape: (150, 4)
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [6.8 3.2 5.9 2.3]
 [6.7 3.3 5.7 2.5]
 [6.7 3.  5.2 2.3]
 [6.3 2.5 5.  1.9]
 [6.5 3.  5.2 2. ]
 [6.2 3.4 5.4 2.3]
 [5.9 3.  5.1 1.8]]

target 의 type: <class 'numpy.ndarray'>
target 의 shape: (150,)
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

#2.4 Model Selection 모듈 소개

1) train_test_split(): 학습/테스트 데이터 세트 분리

(feature data set, label data set, test_size, train_size, shuffle, random_state)

Test_size: 테스트 데이터 세트 크기를 얼마로 샘플링할 것인가(default=0.25)

Train_size: 학습용 데이터 세트 크기를 얼마로 생플링할 것인가(보통 사용 안함)

Shuffle: 데이터 분리 전 데이터를 미리 섞을 지 결정(default=True)

Random_state: 호출할 때마다 동일한 분리 세트 생성을 위해 주어지는 난수 값

#2.4 Model Selection 모듈 소개

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

dt_clf = DecisionTreeClassifier( )
iris_data = load_iris()

X_train, X_test, y_train, y_test = train_test_split(iris_data.data, iris_data.target, \
                                                    test_size = 0.3, random_state = 121)
```

```
dt_clf.fit(X_train, y_train)
pred = dt_clf.predict(X_test)
print('예측 정확도: {0:.4f}'.format(accuracy_score(y_test, pred)))
```

```
예측 정확도: 0.9556
```

#2.4 Model Selection 모듈 소개

2) 교차 검증

:별도의 여러 세트로 구성된 학습 데이터 세트와 검증 데이터 세트에서 학습과 평가를

수행함

*과적합(overfitting)문제 해결을 위해 교차 검증을 이용해 다양한 학습과 평가 수행

#2.4 Model Selection 모듈 소개

k 폴드 교차 검증

: 가장 보편적인 교차 검증 기법으로,

k개의 데이터 폴드 세트를 만들어 k번 각 폴드 세트에 학습,검증 평가 반복

#2.4 Model Selection 모듈 소개

```
n_iter = 0

# KFold 객체의 split()를 호출하면 폴드별 학습용, 검증용 테스트의 로우 인덱스를 array로 반환
for train_index, test_index in kfold.split(features):
    # kfold.split( )으로 반환된 인덱스를 이용해 학습용, 검증용 테스트 데이터 추출
    X_train, X_test = features[train_index], features[test_index]
    y_train, y_test = label[train_index], label[test_index]
    #학습 및 예측
    dt_clf.fit(X_train, y_train)
    pred = dt_clf.predict(X_test)
    n_iter += 1
    # 반복 시마다 정확도 측정
    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: {3}'
          .format(n_iter, accuracy, train_size, test_size))
    print('#{0} 검증 세트 인덱스:{1}'.format(n_iter, test_index))
    cv_accuracy.append(accuracy)
#개별 iteration별 정확도를 합하여 평균 정확도 계산
print('\n## 평균 검증 정확도:', np.mean(cv_accuracy))
```

```
#1 교차 검증 정확도 :1.0, 학습 데이터 크기: 120, 검증 데이터 크기: 30
#1 검증 세트 인덱스:[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29]

#2 교차 검증 정확도 :0.9667, 학습 데이터 크기: 120, 검증 데이터 크기: 30
#2 검증 세트 인덱스:[30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53
 54 55 56 57 58 59]

#3 교차 검증 정확도 :0.8667, 학습 데이터 크기: 120, 검증 데이터 크기: 30
#3 검증 세트 인덱스:[60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
 84 85 86 87 88 89]

#4 교차 검증 정확도 :0.9333, 학습 데이터 크기: 120, 검증 데이터 크기: 30
#4 검증 세트 인덱스:[ 90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107
 108 109 110 111 112 113 114 115 116 117 118 119]

#5 교차 검증 정확도 :0.7333, 학습 데이터 크기: 120, 검증 데이터 크기: 30
#5 검증 세트 인덱스:[120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137
 138 139 140 141 142 143 144 145 146 147 148 149]

## 평균 검증 정확도: 0.9
```

#2.4 Model Selection 모듈 소개

stratified K 폴드: 불균형한 분포도를 가진 레이블 데이터 집합을 위한 K 폴드

원본 데이터의 레이블 분포를 먼저 고려한 후 이 분포와 동일하게 데이터 세트를 분배

```
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())
```

```
## 교차 검증: 1
학습 레이블 데이터 분포:
 label
1    50
2    50
Name: count, dtype: int64
검증 레이블 데이터 분포:
 label
0    50
Name: count, dtype: int64
## 교차 검증: 2
학습 레이블 데이터 분포:
 label
0    50
2    50
Name: count, dtype: int64
검증 레이블 데이터 분포:
 label
1    50
Name: count, dtype: int64
## 교차 검증: 3
학습 레이블 데이터 분포:
 label
0    50
1    50
Name: count, dtype: int64
검증 레이블 데이터 분포:
 label
2    50
Name: count, dtype: int64
```

이렇게 되면 교차 검증마다 3개의 폴드 세트로 만들어지는

학습 레이블과 검증 레이블이 완전히 다른 값으로 추출된다.

->검증 예측 정확도가 0임.

#2.4 Model Selection 모듈 소개

```python
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())
```

```
## 교차 검증: 1
학습 레이블 데이터 분포:
 label
2    34
0    33
1    33
Name: count, dtype: int64
검증 레이블 데이터 분포:
 label
0    17
1    17
2    16
Name: count, dtype: int64
## 교차 검증: 2
학습 레이블 데이터 분포:
 label
1    34
0    33
2    33
Name: count, dtype: int64
검증 레이블 데이터 분포:
 label
0    17
2    17
1    16
Name: count, dtype: int64
## 교차 검증: 3
학습 레이블 데이터 분포:
 label
0    34
1    33
2    33
Name: count, dtype: int64
검증 레이블 데이터 분포:
 label
1    17
2    17
0    16
Name: count, dtype: int64
```

출력 결과를 보면 학습 레이블과 검증 레이블 데이터 값의 분포도가 거의 동일하게 할당됐음을 알 수 있다.

#2.4 Model Selection 모듈 소개

```
dt_clf = DecisionTreeClassifier(random_state=156)

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

# StratifiedKFold의 split( ) 호출시 반드시 레이블 데이터 세트도 추가 입력 필요
for train_index, test_index in skfold.split(features, label):
    # split( )으로 반환된 인덱스를 이용해 학습용, 검증용 테스트 데이터 추출
    X_train, X_test = features[train_index], features[test_index]
    y_train, y_test = label[train_index], label[test_index]
    #학습 및 예측
    dt_clf.fit(X_train, y_train)
    pred = dt_clf.predict(X_test)

    #반복 시마다 정확도 측정
    n_iter += 1
    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: {3}'
          .format(n_iter, accuracy, train_size, test_size))
    print('#{0} 검증 세트 인덱스:{1}'.format(n_iter, test_index))
    cv_accuracy.append(accuracy)

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

```
#1 교차 검증 정확도 :0.98, 학습 데이터 크기: 100, 검증 데이터 크기: 50
#1 검증 세트 인덱스:[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  50
  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66 100 101
 102 103 104 105 106 107 108 109 110 111 112 113 114 115]

#2 교차 검증 정확도 :0.94, 학습 데이터 크기: 100, 검증 데이터 크기: 50
#2 검증 세트 인덱스:[ 17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  67
  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82 116 117 118
 119 120 121 122 123 124 125 126 127 128 129 130 131 132]

#3 교차 검증 정확도 :0.98, 학습 데이터 크기: 100, 검증 데이터 크기: 50
#3 검증 세트 인덱스:[ 34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  83  84
  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 133 134 135
 136 137 138 139 140 141 142 143 144 145 146 147 148 149]

## 교차 검증별 정확도: [0.98 0.94 0.98]
## 평균 검증 정확도: 0.9667
```

#2.4 Model Selection 모듈 소개

3) cross_val_score(): 교차 검증을 간편하게

폴드 세트 설정

For문으로 인덱스 추출

반복 학습, 예측 후 성능 반환

Cross_val_score
()

#2.4 Model Selection 모듈 소개

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

estimator: 사이킷런의 Classifier 또는 Regressor

X: feature set

y: label data set

Scoring: 예측 성능 평가 지표

cv: 교차 검증 폴드 수

#2.4 Model Selection 모듈 소개

```
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()
dt_clf = DecisionTreeClassifier(random_state=156)

data = iris_data.data
label = iris_data.target

#성능 지표는 정확도(accuracy), 교차 검증 세트는 3개
scores = cross_val_score(dt_clf, data, label, scoring='accuracy', cv=3)
print('교차 검증별 정확도:', np.round(scores, 4))
print('평균 검증 정확도:', np.round(np.mean(scores), 4))
```

```
교차 검증별 정확도: [0.98 0.94 0.98]
평균 검증 정확도: 0.9667
```

02.Model Selection(2)와 데이터 전처리

EWHA EURON

#2-4 GridSearchCV 이해

#1 GridSearchCV란?

- 머신러닝 모델의 하이퍼파라미터를 자동으로 탐색해주는 도구

*하이퍼 파라미터 : 머신러닝 알고리즘을 구성하는 주요구성요소로, 이 값을 조정해 알고리즘의 예측 성능을 개선할 수 있음.

- Grid는 격자라는 뜻으로, 촘촘하게 파라미터를 입력하면서 테스트를 하는 방식

- 여러 파라미터 조합을 시험하며
교차검증(Cross Validation)으로 성능 평가

- 최적의 파라미터 조합과 그 성능을 찾아줌.

#2 GridSearchCV 주요 파라미터

-**estimator**(어떤 모델을 사용할지) : classfier, regressor, pipleline이 사용될 수 있다.

-**param_grid** (탐색할 하이퍼 파라미터 후보 딕셔너리): key + 리스트 값을 가지는 딕셔너리가 주어짐.

-**scoring** : 성능 평가 지표
예) accuuacy, 직접 만든 함수도 가능

-**cv** : 교차 검증을 위해 분활되는 학습/테스트 세트의 개수

- **refit**
: True 일 경우, 최적 파라미터로 전체 데이터 재학습
False일 경우, 성능 평가는 하지만 최종 학습된 모델은 없음. 디폴트는 True이다.

#2-4 GridSearchCV 이해

#3 GridSearchCV API 의 사용법

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

# 데이터를 로딩하고 학습 데이터와 테스트 데이터 분리
iris_data = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris_data.data, iris_data.target,
                                                    test_size=0.2, random_state=121)
```

1. load_iris()로 아이리스 데이터셋을 불러와서 iris_data에 저장.

2. train_test_split은 데이터를 훈련용(train)과 테스트용(test)으로 나누는 함수

iris_data.data → 입력 데이터(features)

iris_data.target → 레이블(target, 꽃 종류)

test_size=0.2 → 테스트 데이터 비율을 20%로 지정

random_state=121 → 난수 시드를 고정해서 매번 같은 결과가 나오도록 함

반환값:

X_train → 훈련용 특징 / X_test → 테스트용 특징

y_train → 훈련용 레이블 / y_test → 테스트용 레이블

```
dtree = DecisionTreeClassifier()

### 파라미터를 딕셔너리 형태로 설정
parameters = {'max_depth':[1, 2, 3], 'min_samples_split':[2, 3]}
```

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

#2-4 GridSearchCV 이해

```
import pandas as pd

# param_grid의 하이퍼 파라미터를 3개의 train, test set fold로 나누어 테스트 수행 설정.
### refit=True가 default임. True이면 가장 좋은 파라미터 설정으로 재학습시킴.
grid_dtree = GridSearchCV(dtree, param_grid=parameters, cv=3, refit=True)
```

데이터를 3등분해서 교차검증

```
# 붓꽃 학습 데이터로 param_grid의 하이퍼 파라미터를 순차적으로 학습/평가 .
grid_dtree.fit(X_train, y_train)

# GridSearchCV 결과를 추출해 DataFrame으로 변환
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']]
```

.fit : 실행시 CV * 파라미터 조합의 수 로 학습, 평가 진행

-params : 수행할 때마다 적용된 개별 하이퍼 파라미터값.
-mean_test_score : 교차검증3번의 평균 정확도
-rank_test_score 는 성능 순위를 나타내며 '1'이 가장 성능이 좋음을 나타냄.
-'split0_test_score', 'split1_test_score', 'split2_test_score' : 각 fold별 점수

☞ **하이퍼파라미터 조합마다 각 fold에서 나온 점수와 평균, 순위를 한눈에 확인할 수 있다.**

	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

#2-4 GridSearchCV 이해

#4 확인절차

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

1. grid_dtree.best_params_

GridSearchCV가 찾은 최적 하이퍼파라미터 조합을 딕셔너리 형태로 반환

2. grid_dtree.best_score_

교차 검증에서 최고 평균 점수를 반환

.format()을 사용해서 소수점 4자리까지 출력

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

※ 개별 하이퍼 파라미터값이 갖는 성능이 같으면
GridSearchCV에서는 먼저 나온 조합을 최적 파라미터로 정의
하지만, 실제 연구에서는 같은 성능을 가진 파라미터값을 모두 테스트 데이터에 적용해보고 성능이 더 좋은 쪽을 선택

```
# GridSearchCV의 refit으로 이미 학습된 estimator 반환
estimator = grid_dtree.best_estimator_

# GridSearchCV의 best_estimator_는 이미 최적 학습이 됐으므로 별도 학습이 필요 없음
pred = estimator.predict(X_test)
print('테스트 데이터 세트 정확도: {0:.4f}'.format(accuracy_score(y_test, pred)))
```

1. grid_dtree.best_estimator_

GridSearchCV에서 최적 하이퍼파라미터로 이미 학습된 모델을 반환
별도로 fit()을 다시 할 필요가 없음.

2. predict()

테스트 데이터 X_test에 대한 예측값 계산.

3. accuracy_score(y_test, pred)

실제값 y_test와 예측값 pred를 비교해 정확도를 계산

```
테스트 데이터 세트 정확도: 0.9667
```

#2-5 데이터 전처리

#1 데이터 전처리(Data Preprocessing)란?

데이터 전처리(Data Preprocessing)
: 머신러닝을 하기 전에 데이터를 깨끗하게 준비하는 과정

☆중요한 포인트☆
1. 결측값(NaN, Null)
결측값이란 “데이터가 없는 부분”을 의미

머신러닝에서는 그대로 두면 안 되고 평균값이나 다른 적절한 값으로 채워서 처리

단, 중요한 데이터라면 단순히 평균으로 채우면 예측이 틀릴 수 있으니 신중히 선택해야 함

2. 문자형 데이터 처리

머신러닝 알고리즘은 숫자를 이해하지만 문자는 이해하지 못하기 때문에 문자 데이터를 숫자로 바꾸는 과정이 필요

예) ‘TV’, ‘냉장고’, ‘선풍기’ → 숫자로 변환

이 과정이 바로 **데이터 인코딩**이다.

3. 특징(feature)과 벡터화(vectorization)

머신러닝에서는 데이터를 벡터(숫자 리스트)로 만들어서 모델에 삽입

예) ‘문자열 분석’, ‘숫자 벡터화’ 등

#2-5 데이터 전처리

#2 데이터 인코딩 - 레이블 인코딩

레이블 인코딩(Label Encoding): 문자 데이터를 0,1,2,…처럼 숫자로 바꾸는 것.

```
from sklearn.preprocessing import LabelEncoder

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

# LabelEncoder를 객체로 생성한 후, fit( )과 transform( )으로 레이블 인코딩 수행.
encoder = LabelEncoder()
encoder.fit(items)
labels = encoder.transform(items)
print('인코딩 변환값:', labels)
```

```
인코딩 변환값: [0 1 4 5 3 3 2 2]
```

TV → 0
냉장고 → 1
전자레인지 → 4
컴퓨터 → 5
선풍기 → 3
믹서 → 2

숫자는 단순히 문자를 구분하기 위한 코드

숫자가 크거나 작다고 중요도가 높다/낮다는 의미는 아님

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

classes_ 속성

classes_ 속성에는 각 숫자가 어떤 원래 값에 대응되는지가 순서대로 들어 있다.

```
인코딩 클래스: ['TV' '냉장고' '믹서' '선풍기' '전자레인지' '컴퓨터']
```

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

inverse_transform() 메서드

inverse_transform()은 숫자 인코딩을 다시 원본 범주형 값으로 되돌리는 함수

```
디코딩 원본값: ['전자레인지' '컴퓨터' '믹서' 'TV' '냉장고' '냉장고' '선풍기' '선풍기']
```

#2-5 데이터 전처리

#2 데이터 인코딩 - 레이블 인코딩

원본 데이터

상품 분류	가격
TV	1,000,000
냉장고	1,500,000
전자레인지	200,000
컴퓨터	800,000
선풍기	100,000
선풍기	100,000
믹서	50,000
믹서	50,000

상품 분류를 레이블 인코딩한 데이터

상품 분류	가격
0	1,000,000
1	1,500,000
4	200,000
5	800,000
3	100,000
3	100,000
2	50,000
2	50,000

※레이블 인코딩의 문제점□

1. 문제 발생 원인

레이블 인코딩은 단순히 "문자를 숫자로 변환"할 뿐인데 이 숫자를 컴퓨터는 **수학적 의미로 해석**할 수 있다.

“냉장고(1)는 TV(0)보다 크다. / 전자레인지(2)는 냉장고(1)보다 크다.” 와 같이

인공지능 모델이 잘못 이해할 가능성이 있다.

2. **실제 의미와의 불일치**

"상품 종류"라는 데이터는 순서나 크기 비교 개념이 없는 범주형 데이터

TV가 냉장고보다 "작다" 또는 "크다"는 의미는 없다.

단지 "다른 종류"일 뿐이다.

#2-5 데이터 전처리

#3 데이터 인코딩 - 원-핫 인코딩(One-Hot Encoding)

원-핫 인코딩: 상품 종류의 개수만큼 새로운 열을 만들고 해당 상품에 해당하는 위치에는 1, 나머지는 모두 0을 표시하는 방식

원본 데이터

상품 분류
TV
냉장고
전자레인지
컴퓨터
선풍기
선풍기
믹서
믹서

원-핫 인코딩

상품분류_ TV	상품분류_ 냉장고	상품분류_ 믹서	상품분류_ 선풍기	상품분류_ 전자레인지	상품분류_ 컴퓨터
1	0	0	0	0	0
0	1	0	0	0	0
0	0	0	0	1	0
0	0	0	0	0	1
0	0	0	1	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	0	0	0

※ 원-핫 인코딩 특징

1. Scikit-learn 라이브러리 활용
cikit-learn의 OneHotEncoder 클래스를 사용

2. OneHotEncoder를 이용해 변환한 값이 희소 행렬(Sparse Matrix) 형태이므로 이를 다시 **toarray() 메서드를 이용**해 밀집 행렬(Dense Matrix)로 변환해야 한다

* 희소 행렬(Sparse Matrix)
: 값 대부분이 0인 행렬

장점: 0이 너무 많을 경우, 0을 일일이 저장하지 않고, "어디에 1이 있는지만" 저장

*밀집 행렬(Dense Matrix)
: 0이든 1이든 모든 값을 빠짐없이 저장하는 방식

→ OneHotEncoder는 메모리 효율성을 위해 희소 행렬을 기본 출력으로 함
이는 메모리 효율성을 위해 설계된 것인데, 사람이 결과를 확인하거나 단순한 계산에 활용하려면 이해하기 어려움.

따라서, 사람이 보기 쉽게 또는 다른 연산에 바로 활용하기 위해 **toarray() 메서드**를 사용하여 희소 행렬을 밀집 행렬로 바꾸는 과정 필요

#2-5 데이터 전처리

#3 데이터 인코딩 - 원-핫 인코딩(One-Hot Encoding)

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

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

```
# 2차원 ndarray로 변환합니다.
items = np.array(items).reshape(-1, 1)

# 원-핫 인코딩을 적용합니다.
oh_encoder = OneHotEncoder()
oh_encoder.fit(items)
oh_labels = oh_encoder.transform(items)

# OneHotEncoder로 변환한 결과는 희소행렬이므로 toarray()를 이용해 밀집 행렬로 변환.
print('원-핫 인코딩 데이터')
print(oh_labels.toarray())
print('원-핫 인코딩 데이터 차원')
print(oh_labels.shape)
```

.fit() : item 데이터를 바탕으로 인코더를 학습

학습된 인코더를 이용해 item 데이터를 원-핫 인코딩으로 변환

```
원-핫 인코딩 데이터
[[1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]]
원-핫 인코딩 데이터 차원
(8, 6)
```

원-핫 인코딩 과정

1. 상품 리스트 준비
2. numpy 배열로 변환 → 2차원 구조로 reshape
3. OneHotEncoder 객체 생성
4. fit()으로 고유한 상품 종류 학습
5. transform()으로 데이터를 원-핫 인코딩 변환
6. .toarray()로 희소 행렬을 사람이 보기 쉬운 밀집 행렬로 변환

#2-5 데이터 전처리

#3 데이터 인코딩 - 원-핫 인코딩(One-Hot Encoding)

원본 데이터

상품 분류	가격
TV	1,000,000
냉장고	1,500,000
전자레인지	200,000
컴퓨터	800,000
선풍기	100,000
선풍기	100,000
믹서	50,000
믹서	50,000

숫자로 인코딩

상품 분류	가격
0	1,000,000
1	1,500,000
4	200,000
5	800,000
3	100,000
3	100,000
2	50,000
2	50,000

원-핫 인코딩

TV	냉장고	믹서	선풍기	전자레인지	컴퓨터	가격
1	0	0	0	0	0	1,000,000
0	1	0	0	0	0	1,500,000
0	0	0	0	1	0	200,000
0	0	0	0	0	1	800,000
0	0	0	1	0	0	100,000
0	0	0	1	0	0	100,000
0	0	1	0	0	0	50,000
0	0	1	0	0	0	50,000

사이킷런의 OneHotEncoder : 원본 데이터를 숫자로 인코딩한 뒤, 원-핫 인코딩 형태로 변환

pandas 라이브러리에 있는 get_dummies()라는 함수를 이용하면 더 쉽게 원-핫 인코딩이 가능
이 함수는 “문자열 카테고리”를 숫자로 바꾸는 과정 없이 곧바로 원-핫 인코딩 형태로 변환한다.

```
import pandas as pd

df = pd.DataFrame({'item':['TV', '냉장고', '전자레인지', '컴퓨터', '선풍기', '선풍기', '믹서', '믹서']
})
pd.get_dummies(df)
```

	item_TV	item_냉장고	item_믹서	item_선풍기	item_전자레인지	item_컴퓨터
0	1	0	0	0	0	0
1	0	1	0	0	0	0
2	0	0	0	0	1	0
3	0	0	0	0	0	1
4	0	0	0	1	0	0
5	0	0	0	1	0	0
6	0	0	1	0	0	0
7	0	0	1	0	0	0

#2-5 데이터 전처리

#4 피처 스케일링과 정규화

1. 피처 스케일링이란?
서로 다른 변수(피처)의 값 범위를 일정한 수준으로 맞추는 작업

이유: 머신러닝 알고리즘은 값의 크기에 민감할 수 있기 때문에 크기가 큰 피처가 결과를 지배하지 않도록 크기를 맞춰 주는 과정이 필요

구분	표준화	정규화	벡터 정규화
목적	평균과 분산을 기준으로 데이터 분포를 조정	값의 범위를 0~1 사이로 맞춤	벡터의 길이를 1로 맞춤 → 서로 비교할 때, 방향만 비교
계산식	$x_i\_new = \frac{x_i - mean(x)}{stdev(x)}$	$x_i\_new = \frac{x_i - \min(x)}{\max(x) - \min(x)}$	$x_i\_new = \frac{x_i}{\sqrt{x_i^2 + y_i^2 + z_i^2}}$
출력 결과	평균 0, 분산 1인 값	0 ~ 1 범위의 값	길이가 1인 벡터
사용 예시	데이터가 평균을 중심으로 퍼진 정도(분산)를 같게 만들어야 할 때	- 값을 0과 1 사이로 제한해야 할 때 - 모델이 입력의 상대적 크기보다 절대적인 범위(0~1)를 필요로 할 때	문서 벡터, 코사인 유사도 계산
특징	데이터 분포를 정규 분포에 가깝게 조정	단위 차이를 없애고 범위를 통일	방향은 유지하고 크기만 1로 맞춤

#2-5 데이터 전처리

4 StandardScaler

StandardScaler : 표준화를 쉽게 지원하기 위한 클래스. 데이터를 평균 0, 분산 1로 변환

```
from sklearn.datasets import load_iris
import pandas as pd
# 붓꽃 데이터 세트를 로딩하고 DataFrame으로 변환합니다.
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())
```

```
feature 들의 평균 값
sepal length (cm)    5.843333
sepal width (cm)     3.057333
petal length (cm)    3.758000
petal width (cm)     1.199333
dtype: float64
```

```
feature 들의 분산 값
sepal length (cm)    0.685694
sepal width (cm)     0.189979
petal length (cm)    3.116278
petal width (cm)     0.581006
dtype: float64
```

```
from sklearn.preprocessing import StandardScaler

# StandardScaler객체 생성
scaler = StandardScaler()
# StandardScaler로 데이터 세트 변환. fit( )과 transform( ) 호출.
scaler.fit(iris_df)
iris_scaled = scaler.transform(iris_df)

# transform( ) 시 스케일 변환된 데이터 세트가 NumPy ndarray로 반환돼 이를 DataFrame으로 변환
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())
```

- fit() : 데이터의 평균과 표준편차 계산
- transform() : 데이터를 평균 0, 분산 1로 변환

```
feature 들의 평균 값
sepal length (cm)   -1.690315e-15
sepal width (cm)    -1.842970e-15
petal length (cm)   -1.698641e-15
petal width (cm)    -1.409243e-15
dtype: float64
```

```
feature 들의 분산 값
sepal length (cm)    1.006711
sepal width (cm)     1.006711
petal length (cm)    1.006711
petal width (cm)     1.006711
dtype: float64
```

#2-5 데이터 전처리

4 MinMaxScaler

MinMaxScaler : 데이터를 0과 1 사이로 변환(가장 작은 값은 0, 가장 큰 값은 1이 되도록 변환)

-필요한 이유 : 데이터의 범위가 서로 다르면 머신러닝 모델이 특정 큰 값에만 집중되는데
MinMaxScaler를 적용하면 모든 값이 같은 범위(0~1)에 들어가기 때문에 모델이 값의 비율에 집중 가능

※ 만약 데이터에 음수 값이 있으면 -1~1 범위로 변환

```
from sklearn.preprocessing import MinMaxScaler

# MinMaxScaler객체 생성
scaler = MinMaxScaler()
# MinMaxScaler로 데이터 세트 변환. fit()과 transform() 호출.
scaler.fit(iris_df)
iris_scaled = scaler.transform(iris_df)

# transform() 시 스케일 변환된 데이터 세트가 NumPy ndarray로 반환돼 이를 DataFrame으로 변환
iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_names)
print('feature들의 최솟값')
print(iris_df_scaled.min())
print('\nfeature들의 최댓값')
print(iris_df_scaled.max())
```

- fit() : 데이터에서 최솟값과 최댓값을 계산해서 기억

- transform() : 계산한 최솟값과 최댓값을 이용해 모든 데이터를 0~1 범위로 변환

원래 데이터: 2, 5, 10, 20
0~1로 변환 값: 0, 0.16, 0.42, 1
가장 작은 값 2 → 0 / 가장 큰 값 20 → 1
나머지 값은 비율에 맞게 변환

```
feature들의 최솟값
sepal length (cm)    0.0
sepal width (cm)     0.0
petal length (cm)    0.0
petal width (cm)     0.0
dtype: float64

feature들의 최댓값
sepal length (cm)    1.0
sepal width (cm)     1.0
petal length (cm)    1.0
petal width (cm)     1.0
dtype: float64
```

#2-5 데이터 전처리

#5 학습 데이터와 테스트 데이터의 스케일링 변환 시 유의점

머신러닝에서 데이터를 표준화(StandardScaler)나 정규화(MinMaxScaler) 할 때 주의할 점

1) 학습 데이터(train)로 기준(평균, 표준편차, 최소값, 최대값 등)을 먼저 계산

2) 테스트 데이터(test)는 학습 데이터로 계산한 기준을 그대로 사용

함수	역할
fit()	학습 데이터에서 기준 정보(평균, 표준편차, 최소값, 최대값) 계산
transform()	계산된 기준을 이용해 데이터를 변환
fit_transform()	fit()과 transform()을 한 번에 수행

☆요약☆
- 학습 데이터 → fit_transform()
- 테스트 데이터 → 학습 데이터로 계산된 기준 → transform()

#2-5 데이터 전처리

#5 학습 데이터와 테스트 데이터의 스케일링 변환 시 유의점

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

# 학습 데이터는 0부터 10까지, 테스트 데이터는 0부터 5까지 값을 가지는 데이터 세트로 생성
# Scaler 클래스의 fit(), transform()은 2차원 이상 데이터만 가능하므로 reshape(-1, 1)로 차원 변경
train_array = np.arange(0, 11).reshape(-1, 1)
test_array =  np.arange(0, 6).reshape(-1, 1)
```

MinMaxScaler :값을 0~1 사이로 변환

```
원본 train_array 데이터: [ 0  1  2  3  4  5  6  7  8  9 10]
Scale된 train_array 데이터: [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
```

```
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_array에 Scale 변환을 할 때는 반드시 fit()을 호출하지 않고 transform()만으로 변환해야 함.
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))
```

* scaler.fit(train_array)
→ 스케일링 '기준' : 학습 데이터에서 최솟값과 최댓값(여기서는 0과 10)을 계산해서 기억

* scaler.transform(train_array)
→ 기억한 기준(0~10)을 사용해 학습 데이터를 0~1로 변환
→ 예: 1 → 0.1, 2 → 0.2, 10 → 1.0

#2-5 데이터 전처리

#5 학습 데이터와 테스트 데이터의 스케일링 변환 시 유의점

테스트 데이터 세트를 변환하는데, fit()을 호출해 스케일링 기준 정보를 다시 적용한 뒤 transform()을 수행한 결과

```
# MinMaxScaler에 test_array를 fit()하게 되면 원본 데이터의 최솟값이 0, 최댓값이 5로 설정됨
scaler.fit(test_array)

# 1/5 scale로 test_array 데이터 변환함. 원본 5->1로 변환.
test_scaled = scaler.transform(test_array)

# test_array의 scale 변환 출력.
print('원본 test_array 데이터:', np.round(test_array.reshape(-1), 2))
print('Scale된 test_array 데이터:', np.round(test_scaled.reshape(-1), 2))
```

```
원본 test_array 데이터: [0 1 2 3 4 5]
Scale된 test_array 데이터: [0.  0.2 0.4 0.6 0.8 1. ]
```

fit(): 스케일러에게 기준(min, max 또는 mean, std 등) 을 학습 데이터로만 계산해서 기억

왜 절대 테스트에 fit()을 쓰면 안 되나? (간단한 숫자 예시)

학습 데이터가 0 ~ 10이면 MinMaxScaler는 min=0, max=10을 기억
1 → 0.1, 2 → 0.2, 10 → 1.0

테스트 데이터가 0 ~ 5이면 (테스트로 fit()하면) min=0, max=5 기준
1 → 0.2, 2 → 0.4, 5 → 1.0

같은 원본값(예: 2) 이라도 학습기준에서는 0.2, 테스트기준에서는 0.4가 되어 같은 값이 다르게 해석됨 → 모델은 학습 때 본 것과 다른 스케일의 입력을 받는 것과 같음.

테스트를 다시 fit() 하면 학습(훈련)과 테스트의 기준이 달라져서 모델이 제대로 평가되지 않음

☞테스트는 scaler.transform(test_array) 만 호출해야 학습과 같은 기준으로 변환됨.

#2-5 데이터 전처리

#5 학습 데이터와 테스트 데이터의 스케일링 변환 시 유의점

테스트 데이터 세트를 변환하는데, fit()을 호출하지 않고 transform()을 수행한 결과

```
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_array에 Scale 변환을 할 때는 반드시 fit()을 호출하지 않고 transform()만으로 변환해야 함.
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))
```

```
원본 train_array 데이터: [ 0  1  2  3  4  5  6  7  8  9 10]
Scale된 train_array 데이터: [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]

원본 test_array 데이터: [0 1 2 3 4 5]
Scale된 test_array 데이터: [0.  0.1 0.2 0.3 0.4 0.5]
```

fit_transform()에 대한 주의

fit_transform()은 내부적으로 fit() → transform()을 연속으로 실행함.

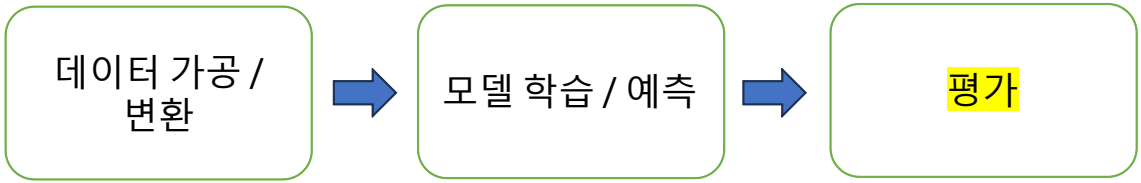
따라서 학습 데이터에서는 편리하게 사용해도 되지만,
테스트 데이터에서는 절대 사용하면 안 됨 (테스트 기준으로 다시 fit하므로).

☆요약☆
1. 전체 데이터의 스케일링 변환을 적용한 뒤 학습과 테스트 데이터 분리
2. 먼저 데이터를 train/test로 나누기 → 그 다음에
MinMaxScaler().fit_transform(X_train) → scaler.transform(X_test)

03. 평가

#3-1 정확도

#1 머신러닝의 프로세스



#2 성능 평가 지표 (Evaluation Metric)

: 모델 예측 성능을 평가하는 지표

회귀: 실제값과 예측값의 오차 평균값에 기반
분류: 정확도 / 오차행렬 / 정밀도 / 재현율 / F1 스코어 / ROC AUC

-> 이진 분류: 2개의 결괏값만 가짐
-> 멀티 분류: 여러 개의 결괏값을 가짐

#3 정확도 : accuracy_score()

: 실제 데이터에서 예측 데이터가 얼마나 같은지 판단하는 지표

정확도=(예측 결과가 동일한 데이터 건수) / (전체 예측 데이터 건수)

⇒ 불균형한 레이블 값 분포에서 적합하지 않음!

예) 100개의 데이터 중 0이 90개, 1이 10개일 때
무조건 0으로 예측하는 모델의 정확도는 90%

#3-1 정확도

예시 1) 타이타닉 생존자 예측 – Sex가 1이면 0, 0이면 1로 예측

- 클래스 상속: 이미 만들어진 클래스(=부모 클래스)에 속성, 메소드를 추가해서 새로운 클래스(=자식 클래스)를 생성하는 것
- BaseEstimator: 모든 모델의 기본이 되는 부모 클래스 -> fit(), predict() 등의 구조를 정의

=> BaseEstimator 클래스를 상속받아 직접 모델을 생성할 수 있다

```
from sklearn.base import BaseEstimator

# BaseEstimator를 상속받아 MyDummyClassifier라는 클래스 생성
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
```

이렇게 단순한 알고리즘도 정확도 결과가 약 78.88%로 높게 나온다!

```
# 위에서 생성한 Dummy Classifier를 이용해 학습/예측/평가 수행.
myclf = MyDummyClassifier()  # 클래스 객체 생성
myclf.fit(X_train, y_train)  # 학습
mypredictions = myclf.predict(X_test)   # 예측
print('Dummy Classifier의 정확도는: {0:.4f}'.format(accuracy_score(y_test, mypredictions)))
```

#3-1 정확도

예시 2) MNIST 데이터 중 7 찾기 - 모든 데이터를 0으로 예측

- 0~9 중 7인 것만 True이기 때문에 전체 데이터의 10%만 True인 불균형 데이터 세트

```python
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):  # 학습하지 않음
        pass

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

# MNIST 데이터 로딩
digits = load_digits()

# digits 번호가 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)
```

```python
# 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())
```

```
레이블 테스트 세트 크기 : (450,)
테스트 세트 레이블 0 과 1의 분포도
0    405
1     45
Name: count, dtype: int64
```

: 불균형 데이터 세트

```python
# 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)))
```

```
모든 예측을 0으로 하여도 정확도는:0.900
```

=> 이진 분류에서 정확도는 모델의 성능을 왜곡할 수 있기 때문에 여러 가지 분류 지표와 함께 적용해야 한다!

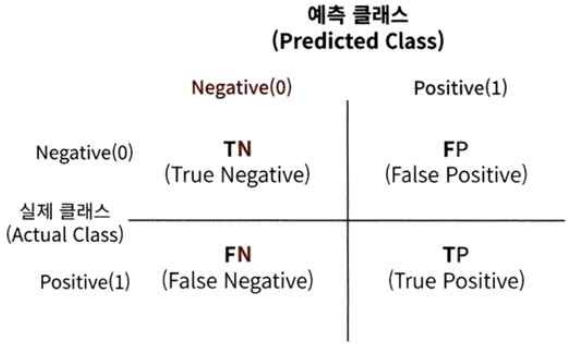
#3-2 오차 행렬

#1 오차 행렬이란

: 예측 오류가 얼마인지와 더불어 어떤 유형의 예측 오류가 발생하고 있는지 함께 나타내는 지표

- 앞 문자 T/F: 예측값과 같은지/틀린지 나타냄
- 뒤 문자 N/P: 예측 결과값이 부정인지/긍정인지 나타냄

ex) TN: T-> 예측값=실제값 / N-> 예측값이 Negative



#2 confusion_matrix()

```
1 from sklearn.metrics import confusion_matrix
2
3 confusion_matrix(y_test, fakepred)
```

```
array([[405,   0],
       [ 45,   0]])
```

- ndarray 형태로 오차 행렬 출력
- 도표와 동일한 위치로 출력
- 이 값들을 조합해 정확도, 정밀도, 재현율 값을 알 수 있음

정확도=(예측 결과와 실제 값이 동일한 건수) / (전체 데이터 수)

=(TN+TP) / (TN+FP+FN+TP)

불균형한 이진 분류 데이터 세트에서는 Negative로 예측하는 경향이 강해짐.

-> TN는 매우 커지고 TP는 매우 작아짐. FN이 매우 작고 FP는 매우 작아짐

-> Positive에 대한 예측 정확도를 판단하지 못한 채 Negative에 대한 예측 정확도만으로 분류의 정확도를 높게 평가하게 됨.

#3-3 정밀도와 재현율

: Positive 데이터 세트의 예측 성능에 초점을 두기 때문에 불균형한 데이터 세트에서 정확도보다 더 선호된다.

#1 정밀도 : precision_score()

= TP/(FP+TP)

= 양성 예측도

= 예측을 positiv로 한 대상 중 실제 positive인 대상의 비율

실제 positive 데이터를 negative로 잘못 판단하면 큰 영향이 생기는 경우 중요하다

Ex) 암 판단 모델, 금융 사기 적발 모델

#2 재현율 : recall_score()

= TP/(FN+TP)

= 민감도, TPR (True Positive Rate)

= 실제 값이 positive인 대상 중 예측도 positive로 일치한 비율

실제 negative 데이터를 positive로 잘못 판단하면 큰 영향이 생기는 경우 중요하다

Ex) 스팸메일 여부

재현율과 정밀도 모두 TP를 높이는데 초점을 두지만, 재현율은 FN을 낮추는데, 정밀도는 FP를 낮추는데 초점을 맞춘다.

=> 두 지표는 상호 보완적이다

#3-3 정밀도와 재현율

#3 정밀도 / 재현율 트레이드오프

: 결정 임곗값(Threshold)을 조정해 정밀도 또는 재현율의 수치를 올릴 수 있지만 어느 한쪽을 높이면 다른 하나는 떨어진다.

> 임곗값을 낮추면 재현율 값이 올라가고 정밀도가 떨어진다.
> (임곗값: positive 예측값을 결정하는 확률의 기준)
> 임곗값을 낮출수록 Positive 예측값이 많아짐 -> 실제 양성값을 음성으로 예측하는 횟수가 줄어듦 -> 재현율 값이 높아짐

사이킷런의 분류 알고리즘은 개별 레이블별로 결정 확률을 구하고 예측 확률이 큰 레이블값으로 예측한다.

➔ 이진 분류에서는 임곗값을 0.5로 정하고 이 값보다 크면 positive, 작으면 negative로 결정한다.

predict_proba(): 테스트 피처 데이터를 입력 받아 개별 클래스의 예측 확률을 ndarray (데이터 수 x 클래스 수) 형태로 반환한다.

➔ 이진 분류에서는 첫 칼럼이 0에 대한 예측 확률, 두 번째 칼럼이 1에 대한 예측 확률이다.

predict()메서드는 predict_proba()메서드에 기반한 API임

➔ 분류 결정 임곗값을 조절해 정밀도와 재현율을 조정할 수 있다.

#3-3 정밀도와 재현율

코드로 직접 구현해 보자!

: pred_proba() 메서드로 구한 객체 변수에 Binarizer 클래스를 적용하면 predict()메서드와 결과가 같다.

* Binarizer 객체:

수치형 데이터를 특정 임곗값을 기준으로 0과 1로 변환해줌

```
from sklearn.preprocessing import Binarizer
X=[[1,-1,2],[2,0,0],[0,1.1,1.2]]

# 임곗값을 1.1로 설정하고 클래스를 객체로 생성
binarizer=Binarizer(threshold=1.1)
# fit_transform 메서드를 이용해 값이 임곗값보다 작으면 0, 크면 1로 변환
print(binarizer.fit_transform(X))
```

```
[0. 0. 1.]
[1. 0. 0.]
[0. 0. 1.]]
```

```
from sklearn.preprocessing import Binarizer
from sklearn.linear_model import LogisticRegression

# 로지스틱 회귀 모델 객체 생성 후 학습
lr_clf = LogisticRegression(solver='liblinear')
lr_clf.fit(X_train, y_train)

pred_proba = lr_clf.predict_proba(X_test)

# Binarizer의 threshold 설정값. 분류 결정 임곗값임.
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)

print(confusion_matrix(y_test, custom_predict))
```

```
[[108  10]
 [ 14  47]]
```

같은 것을 확인할 수 있다.

```
pred = lr_clf.predict(X_test)
print(confusion_matrix(y_test, pred))
```

```
[[108  10]
 [ 14  47]]
```

EWHA EURON

#3-3 정밀도와 재현율

: 임곗값을 바꿔가며 정확도, 정밀도, 재현율의 변화를 살펴보자.

```python
# 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 list객체 내의 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)
```

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

precision_recall_curve()

: 임곗값 변화에 따른 정밀도와 재현율 값을 반환

```python
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)

# 샘플로 10건만 추출해보기
thr_index = np.arange(0, thresholds.shape[0], 15)
print('샘플 추출을 위한 임곗값 배열의 index 10개:', thr_index)
print('샘플용 10개의 임곗값: ', np.round(thresholds[thr_index], 2))
print('샘플 임계값별 정밀도: ', np.round(precisions[thr_index], 3))
print('샘플 임계값별 재현율: ', np.round(recalls[thr_index], 3))
```

```
샘플 추출을 위한 임곗값 배열의 index 10개: [  0  15  30  45  60  75  90 105 120 135 150]
샘플용 10개의 임곗값:  [0.02 0.11 0.13 0.14 0.16 0.24 0.32 0.45 0.62 0.73 0.87]
샘플 임계값별 정밀도:  [0.341 0.372 0.401 0.44  0.505 0.598 0.688 0.774 0.915 0.968 0.938]
샘플 임계값별 재현율:  [1.    1.    0.967 0.902 0.902 0.902 0.869 0.787 0.705 0.492 0.246]
```

#3-3 정밀도와 재현율

#4 정밀도/재현율의 맹점

- 정밀도가 100%가 되는 방법:
  - 확실한 기준이 되는 경우만 Positive로 예측하고 나머지는 모두 Negative로 예측
  - 정밀도=TP/(TP+FP)
  - 정말 확실한 Positive 한 명만 Positive로 예측해도 FP는 0, TP는 1이 되므로 1/(1+0)으로 100%가 된다.

- 재현율이 100%가 되는 방법:
  - 모두 positive로 예측
  - 재현율=TP/(TP+FN)
  - 실제 positive인 사람이 1000명 중 30명이라도 30/(30+0)으로 100%가 된다.

정밀도와 재현율은 극단적인 수치 조작이 가능하다.
⇒ 두 수치가 적절하게 조합되는 평가 지표가 필요!

#3-4 F1 Score

: 정밀도와 재현율을 결합한 지표

- 정밀도와 재현율이 어느 한 쪽으로 치우치지 않을 때 높은 값을 가진다.

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

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

```
F1 스코어: 0.7966
```

```python
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 = 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

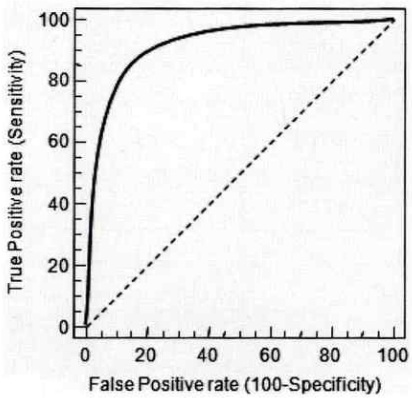
#3-5 ROC 곡선과 AUC

#1 ROC 곡선이란?

(=수신자 판단 곡선)

- FPR(False Positive Rate)이 변할 때 TPR(True Positive Rate, 재현율/민감도)이 어떻게 변하는지 나타내는 곡선
  - TPR(민감도) = TP / (FN+TP), positive가 정확히 예측되는 수준
  - TNR(특이성) = TN / (FP+TN), negative가 정확히 예측되는 수준
  - FPR = FP / (FP+TN) = 1-TNR

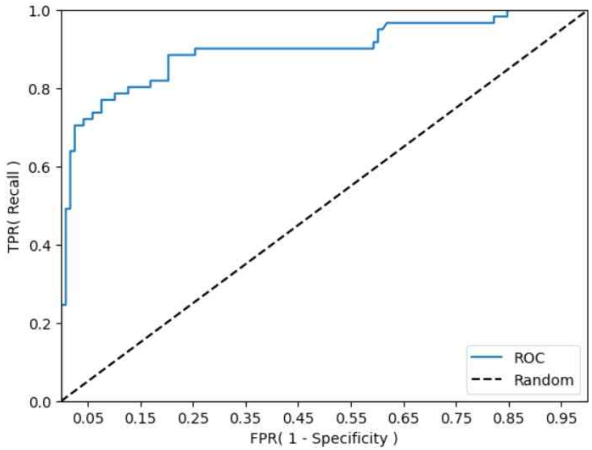
- 가운데 직선은 ROC 곡선의 최저 값
- ROC 곡선이 가운데 직선과 멀어질수록 성능이 뛰어난 것
- 곡선은 분류 결정 임곗값을 바꿔 FPR을 0부터 1까지 변경하면서 TPR의 변화값을 구한다.
- 임곗값 1 → FPR 0
- 임곗값0 → FPR 1



〈 ROC 곡선 예시 〉

#3-5 ROC 곡선과 AUC

```python
import matplotlib.pyplot as plt
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)

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)
    # ROC 곡선을 그래프로 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])
```



#2 AUC 값이란?

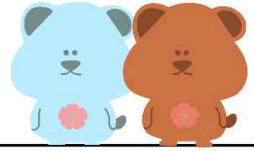
: ROC 곡선 밑의 면적

-> 1에 가까울수록 좋다

```python
from sklearn.metrics import roc_auc_score

# 양성 클래스 확률값 추출
pred_proba = lr_clf.predict_proba(X_test)[:, 1]
# AUC 값 계산
roc_score = roc_auc_score(y_test, pred_proba)
print('ROC AUC 값: {0:.4f}'.format(roc_score))
```

```
ROC AUC 값: 0.8987
```

THANK YOU



EWHA EURON