Week2_사이킷런, 평가

사이킷런 소개와 특징

사이킷런(scikit-learn)

: 파이썬 머신러닝 라이브러리 중 가장 많이 사용되는 라이브러리

- 쉽고 파이썬스러운 API 제공
- ML을 위한 다양한 알고리즘, 편리한 프레임워크, API 제공
- 오랜 기간 다양한 실전 환경에서 사용됨

첫 번째 머신러닝 만들어 보기 - 붓꽃 품종 예측하기

붓꽃 데이터 세트 → 붓꽃 품종 **분류(Classification)**

- 분류 = 대표적인 지도학습(Supervised Learning) 방법의 하나
  - 지도학습
    = 학습을 위한 다양한 피처(Feature)와 분류 결정값
    *(기준)*인 레이블(Label) 데이터 - 학습 데이터 세트 - 로 모델을 학습한 뒤, 별도의 테스트 데이터 세트에서 미지의 레이블 예측

사용할 사이킷런 패키지 내의 모듈들

- sklearn.datasets: 사이킷런에서 자체적으로 제공하는 데이터 세트를 생성하는 모듈의 모임

- sklearn.tree: 트리 기반 ML 알고리즘을 구현한 클래스의 모임
- sklearn.model_selection: 데이터를 분리(학습 데이터와 검증 데이터, 예측 데이터)하거나 최적의 하이퍼 파라미터로 평가하기 위한 모듈의 모임
    - 하이퍼 파라미터 = 학습을 위해 직접 입력하는 파라미터. 하이퍼 파라미터를 통해 ML 알고리즘의 성능을 튜닝할 수 있다.

사용할 ML 알고리즘: 의사 결정 트리(Decision Tree) - DecisionTreeClassifier 적용

```
from sklearn.datasets import load_iris # 붓꽃 데이터
from sklearn.tree import DecisionTreeClassifier # 의사 결정 트리를 구현한 함수
from sklearn.model_selection import train_test_split # 데이터 세트 분리하는 함수

import pandas as pd
```

필요한 모듈 import

붓꽃 품종 예측하기

붓꽃 데이터 세트 로딩 → DataFrame으로 변환해 데이터 확인

- 피처: sepal length, sepal width, petal length, petal width
- 레이블(결정값): 0 - Setosa, 1 - Vericolor, 2 - Virginica

```
# 붓꽃 데이터 세트 로딩
iris=load_iris()

# iris.data: Iris 데이터 세트에서 피처(feature)만으로 된 데이터를 가지고 있는 numpy
iris_data = iris.data

# iris.target: 붓꽃 데이터 세트에서 레이블(결정 값) 데이터를 가지고 있는 numpy
iris_label = iris.target
print('iris target값:', iris_label)
print('iris target명:', iris.target_names)

# 붓꽃 데이터 세트를 자세히 보기 위해 DataFrame으로 변환
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 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 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 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

학습용 데이터와 테스트용 데이터 분리

- train_test_splite(): 학습 데이터와 테스트 데이터를 test_size 파라미터 입력값의 비율로 분할
  - iris_data: 피처 데이터 세트
  - iris_label: 레이블 데이터 세트
  - text_size=0.2: 전체 데이터 세트 중 테스트 데이터 세트의 비율. 테스트 데이터 20%, 학습 데이터 80%
  - random_state: 호출할 때마다 같은 학습/테스트용 데이터 세트를 생성하기 위해 주어지는 난수 발생 값(seed). train_test_split은 random_state를 지정하지 않으면 호출 시 무작위로 데이터를 분리해서, 수행할 때마다 다른 데이터를 만들 수 있다.

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

- X_train: 학습용 피처 데이터 세트
- X_test: 테스트용 피처 데이터 세트
- y_train: 학습용 레이블 데이터 세트
- y_test: 테스트용 레이블 데이터 세트

의사 결정 트리를 이용해 학습 & 예측 수행

- 학습: DecisionTreeClassifier 클래스의 객체를 생성하고,
  객체의
  fit() 메소드에 학습용 피처 데이터 속성과 결정값 데이터 세트를 입력해 호출하면, 학습을 수행한다.

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

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

학습이 완료된 DecisionTreeClassifier 객체

- 예측: 학습된 DecisionTreeClassifier 객체를 이용해 예측을 수행한다. 예측은 반드시 학습 데이터와 다른 데이터를 이용해야 하며, 일반적으로 테스트 데이터 세트를 이용한다.

  객체의 **predict() 메소드**에 테스트용 피처 데이터 세트를 입력해 호출
  → 학습된 모델 기반에서 테스트 데이터 세트에 대한 예측값을 반환한다.

```
# 학습이 완료된 DecisionTreeClassifier 객체에서 테스트 데이터 세트로 예측 수행
pred = dt_clf.predict(X_test)
```

예측 성능 - 정확도 평가하기

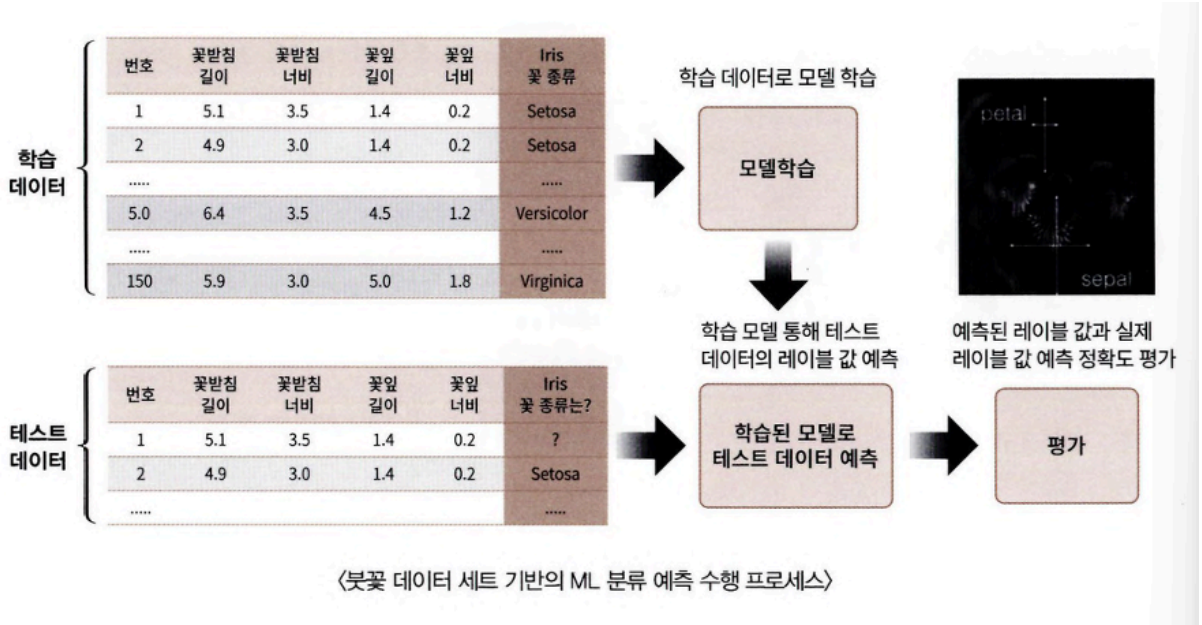
- 정확도 = 예측 결과가 실제 레이블 값과 얼마나 정확하게 맞는지를 평가하는 지표

  → 예측한 붓꽃 품종과 실제 테스트 데이터 세트의 붓꽃 품종이 얼마나 일치하는지 확인하기

  ⇒ 사이킷런의 **accuracy_score(***실제 레이블 데이터 세트, 예측 레이블 데이터 세트***)** 함수를 사용

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

```
예측 정확도: 0.9333
```



〈붓꽃 데이터 세트 기반의 ML 분류 예측 수행 프로세스〉

사이킷런 기반 프레임워크 익히기

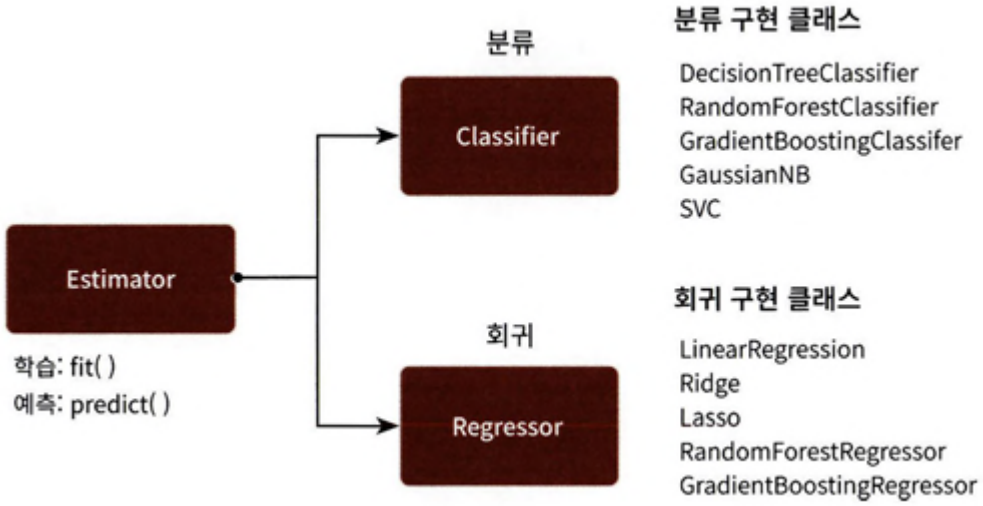
Estimator 이해 및 fit(), predict() 메소드

1. 지도학습 구현 클래스

   Estimator 클래스 = Classifier(분류) + Regressor(회귀) 클래스 = 지도학습 클래스

   - evaluation 함수, 하이퍼 파라미터 튜닝을 지원하는 클래스의 경우 Estimator를 인자로 받는다.

   fit()-학습, predict()-예측: Estimator 클래스는 fit(), predict()만을 이용해 학습과 예측 결과를 반환함



2. 비지도학습 구현 클래스

   = 차원 축소, 클러스터링, 피처 추출(Feature Extraction) 등을 구현한 클래스

   fit(): 입력 데이터의 형태에 맞춰 데이터를 변환하기 위한 사전 구조를 맞추는 작업

   transform(): 입력 데이터의 차원 변환, 클러스터링, 피처 추출 등의 실제 작업

   → fit_transform(): fit() + transform()

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(문자열을 숫자 형 코드 값으로 인코딩, 정규화, 스케일링 등)
	sklearn.feature_selection	알고리즘에 큰 영향을 미치는 피처를 우선순위대로 셀렉션 작업을 수행하는 다양한 기능 제공

피처 처리	sklearn.feature_extraction	텍스트 데이터나 이미지 데이터의 벡터화된 피처를 추출하는 데 사용됨. 예를 들어 텍스트 데이터에서 Count Vectorizer나 Tf-Idf Vectorizer 등을 생성하는 기능 제공. 텍스트 데이터의 피처 추출은 sklearn.feature_extraction.text 모듈에, 이미지 데이터의 피처 추출은 sklearn.feature_extraction.image 모듈에 지원 API가 있음.
피처 처리 & 차원 축소	sklearn.decomposition	차원 축소와 관련한 알고리즘을 지원하는 모듈임. PCA, NMF, Truncated SVD 등을 통해 차원 축소 기능을 수행할 수 있음
데이터 분리, 검증 & 파라미터 튜닝	sklearn.model_selection	교차 검증을 위한 학습용/테스트용 분리, 그리드 서치(Grid Search)로 최적 파라미터 추출 등의 API 제공
평가	sklearn.metrics	분류, 회귀, 클러스터링, 페어와이즈(Pairwise)에 대한 다양한 성능 측정 방법 제공 Accuracy, Precision, Recall, ROC-AUC, RMSE 등 제공
ML 알고리즘	sklearn.ensemble	앙상블 알고리즘 제공 랜덤 포레스트, 에이다 부스트, 그래디언트 부스팅 등을 제공
	sklearn.linear_model	주로 선형 회귀, 릿지(Ridge), 라쏘(Lasso) 및 로지스틱 회귀 등 회귀 관련 알고리즘을 지원. 또한 SGD(Stochastic Gradient Descent) 관련 알고리즘도 제공
	sklearn.naive_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

ML 모델을 구축하는 주요 프로세스

: 피처 처리 → ML 알고리즘 학습/예측 수행 → 모델 평가 (→ 반복)

내장된 예제 데이터 세트

분류나 회귀 연습용 예제 데이터

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

사이킷런에 내장된 데이터 세트는 일반적으로 **딕셔너리** 형태

데이터 세트의 주요 key들

- data: 피처의 데이터 세트 (ndarray)
- target: 분류-레이블 값, 회귀-숫자 결괏값 데이터 세트 (ndarray)
- target_names: 개별 레이블 이름 (ndarray OR list)
- feature_names: 피처 이름 (ndarray OR list)
- DESCR: 데이터 세트에 대한 설명과 각 피처의 설명 (string)

```
from sklearn.datasets import load_iris

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

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

Bunch 클래스 = 파이썬 딕셔너리 자료형. 이 딕셔너리의 키 값이 위의 5개인 것.

```
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'])
```

이 키들을 이용해 딕셔너리를 사용하듯 *datasets.data* 또는 *datasets*['data'] 등등.. 으로 이용하면 된다.

```
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(' data의 shape:', iris_data.data.shape)
print(iris_data['data'])

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

```
 feature_names의 type: <class 'list'>
 feature_names의 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']

 data의 type: <class 'numpy.ndarray'>
 data의 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]
 [4.6 3.4 1.4 0.3]
```

```
 [6.9 3.1 5.1 2.3]
 [5.8 2.7 5.1 1.9]
 [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]]

 target의 type: <class 'numpy.ndarray'>
 target의 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 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]
```

- fetch 계열: 데이터 크기가 커서 패키지에 처음부터 저장되어 있지 않고, *인터넷→홈 디렉터리 아래의 scikit_learn_data라는 서브 디렉터리*→나중에 불러들이는 데이터
- 표본 데이터 생성기
  - datasets.make_classifications(): 분류를 위한 데이터 세트를 만듦. 높은 상관도, 불필요한 속성 등의 노이즈 효과를 위한 데이터를 무작위로 생성함.

- datasets.make_blobs(): 클러스터링을 위한 데이터 세트를 만듦. 군집 지정 개수에 따라 여러 가지 클러스터링을 위한 데이터 세트를 생성함.

Model Selection 모듈 소개

model_selection 모듈: 학습 데이터와 테스트 데이터 세트 분리, 교차 검증 분할 및 평가, Estimator의 하이퍼 파라미터 튜닝을 위한 다양한 함수와 클래스 제공

학습/테스트 데이터 세트 분리 - train_test_split()

⚠️ 학습 & 예측을 동일한 데이터 세트로 수행할 때 발생하는 문제

```
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 데이터 세트로 예측 수행
pred = dt_clf.predict(train_data)
print('예측 정확도:', accuracy_score(train_label, pred))
```

```
예측 정확도: 1.0
```

→ 이미 학습한 학습 데이터를 기반으로 예측했기 때문에 당연히 정확도가 100%가 나온다. 따라서 예측을 수행하는 데이터 세트는 학습에 사용되지 않은 다른 데이터 세트여야 함.

train_test_split()를 이용해 붓꽃 데이터 세트를 학습/테스트 데이터 세트로 분리하기

train_test_split()의 파라미터들

- test_size: 전체 데이터에서 테스트 데이터 세트 크기를 얼마로 샘플링할지 결정 (default=0.25)
- train_size: 전체 데이터에서 학습용 데이터 세트 크기를 얼마로 샘플링할지 결정 (잘 사용하지 않는다)
- shuffle: 데이터를 분리하기 전에 데이터를 미리 섞을지(데이터가 더 분산되게 할지) 결정 (default=True)
- random_state: 호출 시 동일한 학습/테스트용 데이터 세트를 생성하기 위해 주어지는 난수 값. random_state를 지정하면 수행할 때마다 동일한 데이터 세트로 분리됨.

+) train_test_split()의 리턴값 = 튜플 (train 피처 세트, test 피처 세트, train 레이블 세트, test 레이블 세트)

```
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)
```

eature → label(결과)라서 X→y라고 이해함

학습 데이터의 양을 일정 수준 이상으로 보장하는 것도 중요하고, 최대한 다양한 데이터를 기반으로 예측 성능을 평가하는 것도 중요하다!

교차 검증

⚠️ 과적합(Overfitting) 문제: 모델이 학습 데이터에만 과도하게 최적화되어, 실제 예측을 다른 데이터로 수행할 경우에는 예측 성능이 과도하게 떨어지는 문제.
→ 고정된 학습&테스트 데이터로 평가하다 보면 테스트 데이터에만 최적의 성능을 발휘할 수 있도록 편향되게 모델을 유도하는 경향이 생기게 됨
⇒ 교차 검증 필요!

- 교차 검증 = 데이터 편중을 막기 위해, 많은 학습/테스트 데이터 세트에서 알고리즘 학습과 평가를 수행하는 것

K 폴드 교차 검증

: 먼저 K개의 데이터 폴드 세트를 만들어서, K번만큼 각 폴드 세트에 학습과 검증 평가를 반복적으로 수행하는 방법. K개의 평가를 평균한 결과로 예측 성능을 평가한다.

→ 사이킷런에서 KFold, StratifiedKFold 클래스 제공

KFold 클래스를 이용해 붓꽃 데이터 세트를 교차 검증하고 예측 정확도 알아보기

KFold()로 KFold 객체 kfold를 생성하기

```
from sklearn.model_selection import KFold
import numpy as np

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

# 5개의 폴드 세트로 분리하는 KFold 객체와 폴드 세트별 정확도를 담을 리스트 객체 생성
kfold = KFold(n_splits=5)
cv_accuracy = []
print('붓꽃 데이터 세트 크기:', features.shape[0])
```

```
붓꽃 데이터 세트 크기: 150
```

kfold의 split()을 호출해 전체 붓꽃 데이터를 5개의 폴드 데이터 세트로 분리하기

전체 붓꽃 데이터 세트 = 150개
학습용 데이터 세트 = 150 * 4/5 = 120개
검증 테스트 데이터 세트 = 150 * 1/5 = 30개

KFold 객체는 split()을 호출하면 학습용/검증용 데이터로 분할할 수 있는 인덱스를 반환한다. 이 인덱스로 데이터 추출을 수행해야 한다.

```
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 # 1회 학습 및 예측 완료
  # 반복마다 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
```

Stratified K 폴드

: 불균형한(imbalanced) 분포도를 가진 레이블 데이터 집합을 위한 K 폴드 방식

- 불균형한 분포도: 특정 레이블 값이 특이하게 많거나 매우 적어서 값의 분포가 한쪽으로 치우치는 것

  → 원본 데이터와 유사한 '중요 레이블' 값의 분포를 학습/테스트 세트에 유지하는 게 중요함

⇒ Stratified K 폴드는 원본 데이터의 레이블 분포를 먼저 고려한 뒤, 이 분포와 동일하게 학습과 검증 데이터 세트를 분배한다.

K 폴드가 가지고 있는 문제 알아보기

붓꽃 데이터 세트를 DataFrame으로 생성한 뒤 레이블 값의 분포도 확인

```python
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()
```

label	count
0	50
1	50
2	50

dtype: int64

이슈 발생 현상 도출하기

```
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
```

→ 매 검증마다 특정 레이블의 분포가 전혀 반영되지 않는 상황

동일한 데이터 분할을 StratifiedKFold로 수행하고 학습/검증 레이블 데이터의 분포도 확인하기

```
from sklearn.model_selection import StratifiedKFold

skf = StratifiedKFold(n_splits=3)
n_iters=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: 4
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: 5
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: 6
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
```

→ 학습 레이블과 검증 레이블 데이터 값의 분포도가 거의 동일하게 할당됨

StratifiedKfold로 교차 검증하기

Stratified K 폴드는 원본 데이터의 레이블 분포도 특성을 반영해서 데이터를 분리해준다. 따라서 왜곡된 레이블 데이터 세트에서는 반드시 Stratified K 폴드를 이용해 교차 검증해줘야 한다.

```
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)

  n_iter += 1 # 1회 학습 및 예측 완료
  # 반복마다 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)

# 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
```

** np.round 찾아보기

분류 교차 검증: 일반적으로 Stratified K 폴드로 분할되어야 함

회귀 교차 검증: 회귀의 결정값은 연속된 숫자값이기 때문에, Stratified K 폴드가 지원되지 X

간편한 교차 검증 - cross_val_score()

: 교차 검증을 더 편리하게 수행할 수 있게 해주는 사이킷런의 API

(1) 폴드 세트 설정 (2) for문으로 학습 및 테스트 데이터의 인덱스 추출 (3) 학습과 예측 반복 수행 및 예측 성능 반환 을 한꺼번에 수행해준다.

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: 사이킷런 분류 알고리즘 클래스 Classifier, 회귀 알고리즘 클래스 Regressor
- X: 피처 데이터 세트
- y: 레이블 데이터 세트
- scoring: 예측 성능 평가 지표
- cv: 교차 검증 폴드 수

+) 리턴값: scoring 파라미터로 지정된 성능 지표 측정값을 배열 형태로 반환

+) classifier가 입력되면 Stratified K 폴드 방식으로 데이터를 분할함

cross_val_score() 사용해보기

위에서 StratifiedKFold를 이용한 것과 동일한 결과가 나오게 된다.

```
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 세트 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
```

cross_val_score()은 cv로 지정된 횟수를 크기로 가지는 평가 결괏값 배열을 반환하므로, 이 배열에 mean() 함수를 적용하면 평가 수치를 얻을 수 있다.

- cross_validate(): 여러 개의 평가 지표를 반환할 수 있으며, 성능 평가 지표와 수행 시간을 함께 제공

GridSearchCV - 교차 검증+최적 하이퍼 파라미터 튜닝

GridSearchCV는 격자처럼 촘촘하게 파라미터를 입력하면서 테스트를 하는 방식이다. 사이킷런의 GridSearchCV API는 분류나 회귀 알고리즘에 사용되는 하이퍼 파라미터를 순차적으로 입력하면서 편리하게 최적의 파라미터를 도출할 수 있는 방안을 제공한다.

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

이 경우 for문으로 모든 파라미터 조합을 실행하고 최적의 파라미터와 수행 결과를 도출해냄

→ '모든 파라미터 조합'을 테스트해 보기 때문에 수행 시간이 상대적으로 오래 걸린다

GridSearchCV 클래스의 생성자 주요 파라미터

- estimator: classifier, regressor, pipeline
- param_grid: key+리스트 값의 딕셔너리. estimator의 튜닝을 위해 파라미터 지정
- scoring: 예측 성능을 측정할 평가 방법
- cv: 교차 검증을 위해 분할되는 학습/테스트 개수 지정
- refit: True 생성 시 최적 하이퍼 파라미터를 찾은 뒤, 입력된 estimator 객체를 해당 하이퍼 파라미터로 재학습시킴 (default=True)

GridSearchCV로 붓꽃 데이터 예측 분석하기

train_test_split()을 이용해 학습 데이터와 테스트 데이터를 분리하고,
학습 데이터에서 GridSearchCV를 이용해 최적 하이퍼 파라미터를 추출한다.

- 결정 트리 알고리즘 DecisionTreeClassifier의 중요 하이퍼 파라미터: max_depth, min_samples_split

```
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)
dtree = DecisionTreeClassifier()

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

GridSearchCV 객체의 fit(*학습 데이터 세트*) 메소드 수행
→ 학습 데이터를 폴딩 세트로 분할해 param_grid에 기술된 하이퍼 파라미터를 순차적으로 변경하면서 학습/평가를 수행하고 그 결과를 cv_results_ 속성(딕셔너리)에 기록

```
import pandas as pd

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

# 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']]
```

	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

- params 칼럼: 수행할 때마다 적용된 하이퍼 파라미터 값
- rank_test_score: 예측 성능 순위
- mean_test_score: 개별 하이퍼 파라미터별로 CV의 폴딩 테스트 세트에 대해 총 수행한 평가 평균값

fit() 수행 → 최고 성능을 나타낸 하이퍼 파라미터의 값-best_params_, 그때의 평과 결과-best_score_ 속성에 기록됨

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

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

refit=True이므로 GridSearchCV가 최적 성능 하이퍼 파라미터로 Estimator를 학습해 best_estimator_로 저장한다.

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

# GridSearchCv의 best_estimaotor_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)))
```

```
테스트 데이터 세트 정확도: 0.9667
```

⇒> 일반적으로 학습 데이터를 GridSearchCV를 이용해 최적 하이퍼 파라미터 튜닝을 수행한 뒤, 별도의 테스트 세트에서 이를 평가하는 것이 일반적인 머신러닝 모델 적용 방법!

데이터 전처리 (Data Preprocessing)

“Garbage In, Garbage Out”

데이터에 대해 미리 처리해야 할 기본 사항

1. 결손값(NaN, Null)은 허용 X → 다른 값으로 변환해야 한다
    - Null값 처리
        - 피처 값 중 Null 값이 얼마 되지 않는다면 피처의 평균값 등으로 대체
        - Null 값이 대부분이라면 해당 피처는 드롭
        - Null 값이 일정 수준 이상 된다면, 업무 로직 등을 상세히 검토해 더 정밀한 대체 값을 선정해야 함
2. 사이킷런의 머신러닝 알고리즘은 문자열 값을 입력값으로 허용 X → 인코딩돼서 숫자 형으로 변환
    - 카테고리형 피처
    - 텍스트형 피처 → 피처 벡터화(feature vectorization) 등 기법으로 벡터화 또는 불필요한 피처라면 삭제

데이터 인코딩

레이블 인코딩 (Label encoding)

: 카테고리 피처 → 코드형 숫자 값

LabelEncoder 클래스로 구현한다. 객체 생성 후 fit()과 transform()을 호출해 레이블 인코딩을 수행한다.

LabelEncoder로 레이블 인코딩하기

```python
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)

# 데이터의 값이 많은 경우 LabelEncoder 객체의 classes_ 속성값으로 확인
print('인코딩 클래스:', encoder.classes_)

# inverse_transform()으로 인코딩된 값을 다시 디코딩할 수 있음
print('디코딩 원본값:', encoder.inverse_transform([4,5,2,0,1,1,3,3]))
```

```
인코딩 변환값: [0 1 4 5 3 3 2 2]
인코딩 클래스: ['TV' '냉장고' '믹서' '선풍기' '전자레인지' '컴퓨터']
디코딩 원본값: ['전자레인지' '컴퓨터' '믹서' 'TV' '냉장고' '냉장고' '선풍기' '선풍기']
```

⚠️ 레이블 인코딩: 문자열 → 숫자형 카테고리 값

숫자 값에는 크고 작음이라는 특성이 작용하기 때문에, 특정 ML 알고리즘에서 가중치가 더 부여되거나 더 중요하게 인식할 가능성이 생긴다. 그러나 '카테고리'는 순서나 중요도가 없는 값이므로, 레이블 인코딩은 선형 회귀 등의 ML 알고리즘에는 적용하지 않아야 한다.

원-핫 인코딩 (One-Hot Encoding)

: 피처 값의 유형에 따라 새로운 피처(ex. 상품분류_TV)를 추가해 고유 값에 해당하는 칼럼에만 1 표시, 나머지 0 표시

OneHotEncode 클래스로 변환이 가능하다. 단, 입력값으로 2차원 데이터가 필요하며, 변환값이 희소 행렬(Sparse Matrix) 형태이므로 다시 toarray() 메소드를 이용해 밀집 행렬(Dense Matrix)로 변환해야 한다.

OneHotEncoder를 이용해 원-핫 인코딩으로 변환하기

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

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

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

# 원-핫 인코딩 적용
oh_encoder = OneHotEncoder()
oh_encoder.fit(items)
oh_labels = oh_encoder.transform(items)

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

```
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)
```

8 레코드 1 칼럼 → 8 레코드 6 칼럼

get_dummies()를 이용해 원-핫 인코딩하기

```
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	True	False	False	False	False	False
1	False	True	False	False	False	False
2	False	False	False	False	True	False
3	False	False	False	False	False	True
4	False	False	False	True	False	False
5	False	False	False	True	False	False
6	False	False	True	False	False	False
7	False	False	True	False	False	False

→ 숫자형 값으로 변환 없이도 바로 변환이 가능하다!

피처 스케일링과 정규화

피처 스케일링(feature scaling): 서로 다른 변수의 값 범위를 일정한 수준으로 맞추는 작업

- 표준화(Standardization): 데이터의 피처 각각이 평균이 0이고 분산이 1인 가우시안 정규 분포를 가진 값으로 변환하는 것

$$x_i\_new = (x_i - mean(x))/stdev(x)$$

- 정규화(Normalization): 서로 다른 피처의 크기를 통일하기 위해 크기를 변환해주는 개념. 개별 데이터의 크기를 모두 똑같은 크기 단위로 비교하는 것이다.

$$x_i\_new = (x_i - min(x))/(max(x) - min(x))$$

  - **벡터 정규화**- 사이킷런의 전처리에서 제공하는 Normalizer 모듈: 선형대수의 정규화 개념이 적용됨 → 개별 벡터의 크기를 맞추기 위해 변환하는 것. 개별 벡터를 모든 피처 벡터의 크기로 나누면 된다.

$$x_i\_new = x_i/\sqrt{x_i^2 + y_i^2 + z_i^2}$$

StandardScaler

: 표준화를 쉽게 지원하기 위한 클래스

** 사이킷런의 Support Vector Machine, Linear Regression, Logistic Regression은 데이터가 가우시안 분포를 가지고 있다고 가정하고 구현됐기 때문에 사전에 표준화를 적용하는 것이 예측 성능 향상에 중요한 요소가 된다.

StandardScaler로 데이터 값 변환하기

```python
# 붓꽃 데이터 세트 로딩하고 DataFrame으로 변환
iris = load_iris()
iris_data = iris.data
iris_df = pd.DataFrame(data=iris_data, columns=iris.feature_names)

print('feature들의 평균 값')
print(iris_df.mean())
print('₩nfeature들의 분산 값')
print(iris_df.var())

from sklearn.preprocessing import StandardScaler

# StandardScaler 객체 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 변환됨 -> df로 변환
iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_names)
print('feature들의 평균 값')
print(iris_df_scaled.mean())
print('₩nfeature들의 분산 값')
print(iris_df_scaled.var())
```

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

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

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

모든 칼럼 값의 평균은 0, 분산은 1에 아주 가까운 값으로 변환되었다!

MinMaxScaler

: 데이터값을 0~1 사이의 범위(음수 값이 있을 경우 -1~1 범위)로 변환. 데이터 분포가 가우시안 분포가 아닐 경우에 Min, Max Scale을 적용해볼 수 있다.

```
from sklearn.preprocessing import MinMaxScaler

# MinMaxScaler 객체 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 변환됨 -> df로 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())
```

```
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
```

학습 데이터와 테스트 데이터의 스케일링 변환 시 유의점

Scaler 객체를 사용할 때

- fit(): 데이터 변환을 위한 기준 정보 설정
- transform(): 이렇게 설정된 정보를 이용해 데이터를 변환
- fit_transform(): fit()+transform()

⚠️ Scaler 객체를 이용해 학습 데이터 세트로 fit(), transform()을 적용하면 테스트 데이터 세트로는 다시 fit()을 수행하지 않고, 학습 데이터 세트의 fit() 결과를 이용해 transform() 변환을 적용해야 한다. 그렇지 않는다면, 학습 데이터와 테스트 데이터의 스케일링 기준 정보가 달라지기 때문에 올바른 예측 결과를 도출하지 못할 수 있다.

테스트 데이터에 fit()을 적용할 때 발생하는 문제 알아보기

```
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 가능
# -> reshape(-1,1)로 차원 변경
train_array = np.arange(0,11).reshape(-1,1)
test_array = np.arange(0,6).reshape(-1,1)
```

train_array에 MinMaxScaler의 fit()을 적용하면 최솟값 0, 최댓값 10이 설정되고 1/10 스케일이 적용된다. 여기에 transform()을 호출하면 1/10 scale로 학습 데이터를 변환하며, 원본 데이터 1→0.1, ...로 변환된다.

```
# MinMaxScaler 객체에 별도의 feature_range 값을 지정하지 않으면 0~1 값으로 변환
scaler = MinMaxScaler()

# fit()하게 되면 train_array 데이터의 최솟값이 0, 최댓값이 10으로 설정
scaler.fit(train_array)

# 1/10 scale로 train_array 데이터 변환함. 원본 10->1로 변환됨
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))

# MinMaxScaler에 test_array를 fit()하게 되면 원본 데이터 값이 0~5로 설정됨
scaler.fit(test_array)

# test_array의 scale 변환 출력
print('원본 test_array데이터:', np.round(test_array.reshape(-1), 2))
print('Scale된 test_array데이터:', np.round(test_array.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_array데이터: [0 1 2 3 4 5]
Scale된 test_array데이터: [0 1 2 3 4 5]
```

⚠️ 학습 데이터와 테스트 데이터의 스케일링이 맞지 않는다. 테스트 데이터는 1/5로, 학습 데이터는 1/10으로 스케일링되었다. 이렇게 되면 학습 데이터와 테스트 데이터의 서로 다른 원본값이 동일한 값으로 변환될 수도 있다. ML 모델은 학습 데이터를 기반으로 학습되기 때문에 반드시 테스트 데이터는 학습 데이터의 스케일링 기준에 따라야 한다.

학습 데이터로 fit()을 수행한 MinMaxScaler 객체의 transform()을 이용해 테스트 데이터 변환하기

```
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('₩n 원본 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() = fit() + transform()이므로 테스트 데이터에서는 절대 사용하면 안 되는 메소드

1. 가능하다면 전체 데이터의 스케일링 변환을 적용한 뒤 학습과 테스트 데이터로 분리
2. 아니면 학습 데이터로 이미 fit()된 Scaler 객체를 이용해 transform()으로 변환

사이킷런으로 수행하는 타이타닉 생존자 예측

- Passengerid: 탑승자 데이터 일련번호
- survived: 생존 여부, 0 = 사망, 1 = 생존
- pclass: 티켓의 선실 등급, 1 = 일등석, 2 = 이등석, 3 = 삼등석
- sex: 탑승자 성별
- name: 탑승자 이름
- Age: 탑승자 나이
- sibsp: 같이 탑승한 형제자매 또는 배우자 인원수
- parch: 같이 탑승한 부모님 또는 어린이 인원수
- ticket: 티켓 번호
- fare: 요금
- cabin: 선실 번호
- embarked: 중간 정착 항구 C = Cherbourg, Q = Queenstown, S = Southampton

타이타닉 탑승자 데이터

분석에 필요한 라이브러리 import,
타이타닉 탑승자 파일을 판다스의 read_csv()를 이용해 DataFrame으로 로딩하기

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
# ipython에서 rich output(그림 등)을 인라인에서 볼 수 있게 하는 코드

titanic_df = pd.read_csv('/content/drive/MyDrive/Euron Homework/titanic_train.csv') # 경로 복사
titanic_df.head(3)
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S

```
print('₩n ### 학습 데이터 정보 ### ₩n')
print(titanic_df.info())
```

```
 ### 학습 데이터 정보 ###

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
None
```

- RangeIndex: DataFrame 인덱스의 범위 = 전체 row 수 = 891개
- 칼럼 수 = 12

사이킷런 머신러닝 알고리즘은 Null 값을 허용하지 않으므로 Null 값을 처리해야 한다.

Null 값이 있는 칼럼에 대해 **DataFrame의 fillna() 함수를 사용해 Null 값을 평균 또는 고정 값으로 변경하기**

*** 책에서 말한 대로 구현하면 아래와 같은 경고가 뜸*

<ipython-input-52-f438379f6575>:6: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained assignment using an inplace method.

The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, to perform the operation inplace on the original object.

→ df['Age']가 해당 데이터프레임의 오리지널 뷰인지, 복사본인지 구분하지 못 한다. 판다스 3.0부터 inplace=True로 지정한다고 하더라도 이 inplace는 무시될 것(복사본에 inplace를 적용하게 되는 것)이므로, 다른 방법으로 값을 변경할 것을 권장한다.

df.fillna({'Cabin': 'N'}, inplace=True) 또는 df['Cabin'] = df['Cabin'].fillna('N')

```
titanic_df.fillna({'Age': titanic_df['Age'].mean()}, inplace=True)
titanic_df.fillna({'Cabin': 'N'}, inplace=True)
titanic_df.fillna({'Embarked': 'N'}, inplace=True)
print('데이터 세트 Null 값 개수', titanic_df.isnull().sum().sum())
# df에서 NULL 값의 개수를 각각 세고, 칼럼별 null 값의 개수를 추산한 뒤, 모든 null 값의 합을 합함
```

```
데이터 세트 Null 값 개수 0
```

Sex, Cabin, Embarked 문자열 피처들 처리하기

```
print('Sex 값 분포 :₩n', titanic_df['Sex'].value_counts())
print('₩n Cabin 값 분포 :₩n', titanic_df['Cabin'].value_counts())
print('₩n Embarked 값 분포:₩n', titanic_df['Embarked'].value_counts())
```

```
Sex 값 분포 :
 Sex
male      577
female    314
Name: count, dtype: int64

 Cabin 값 분포 :
 Cabin
N              687
C23 C25 C27      4
G6               4
B96 B98          4
C22 C26          3
               ...
E34              1
C7               1
C54              1
E36              1
C148             1
Name: count, Length: 148, dtype: int64

 Embarked 값 분포:
 Embarked
S    644
C    168
Q     77
N      2
Name: count, dtype: int64
```

→ Cabin 값의 경우 N이 가장 많고, 'C23 C25 C27' 등 여러 Cabin 값이 한꺼번에 표기된 Cabin 값이 있는 등 속성값이 제대로 정리되지 않았다. Cabin의 경우 첫 번째 알파벳이 중요한 데이터이므로, 앞 문자만 추출한다.

Cabin 속성 앞 문자만 추출하기

```
titanic_df['Cabin'] = titanic_df['Cabin'].str[:1] # 첫 번째 문자만 추출
print(titanic_df['Cabin'].head(3))
```

```
0    N
1    C
2    N
Name: Cabin, dtype: object
```

성별에 따른 생존자 수 비교하기

```
titanic_df.groupby(['Sex', 'Survived'])['Survived'].count()
```

		Survived
Sex	Survived	
female	0	81
	1	233
male	0	468
	1	109

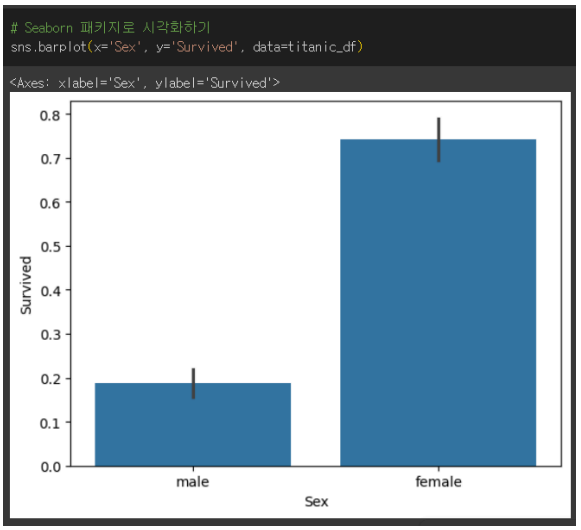
dtype: int64

성별과 생존 여부를 기준으로 groupby한 뒤, Survived 속성을 기준으로 count()함

titanic_df 데이터에서 X 축에 'Sex' 칼럼, Y 축에 'Survived' 칼럼을 입력하여 막대 차트 그리기

```
# Seaborn 패키지로 시각화하기
sns.barplot(x='Sex', y='Survived', data=titanic_df)
```

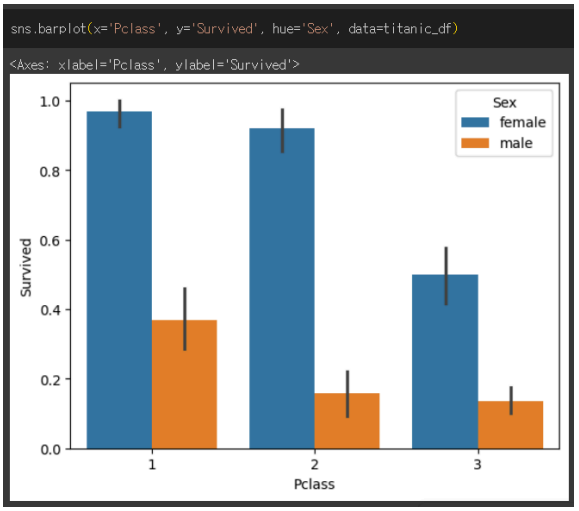
```
<Axes: xlabel='Sex', ylabel='Survived'>
```



seaborn을 이용하여 객실 등급과 성별에 따른 생존 확률 표현하기

```
sns.barplot(x='Pclass', y='Survived', hue='Sex', data=titanic_df)
```

```
<Axes: xlabel='Pclass', ylabel='Survived'>
```

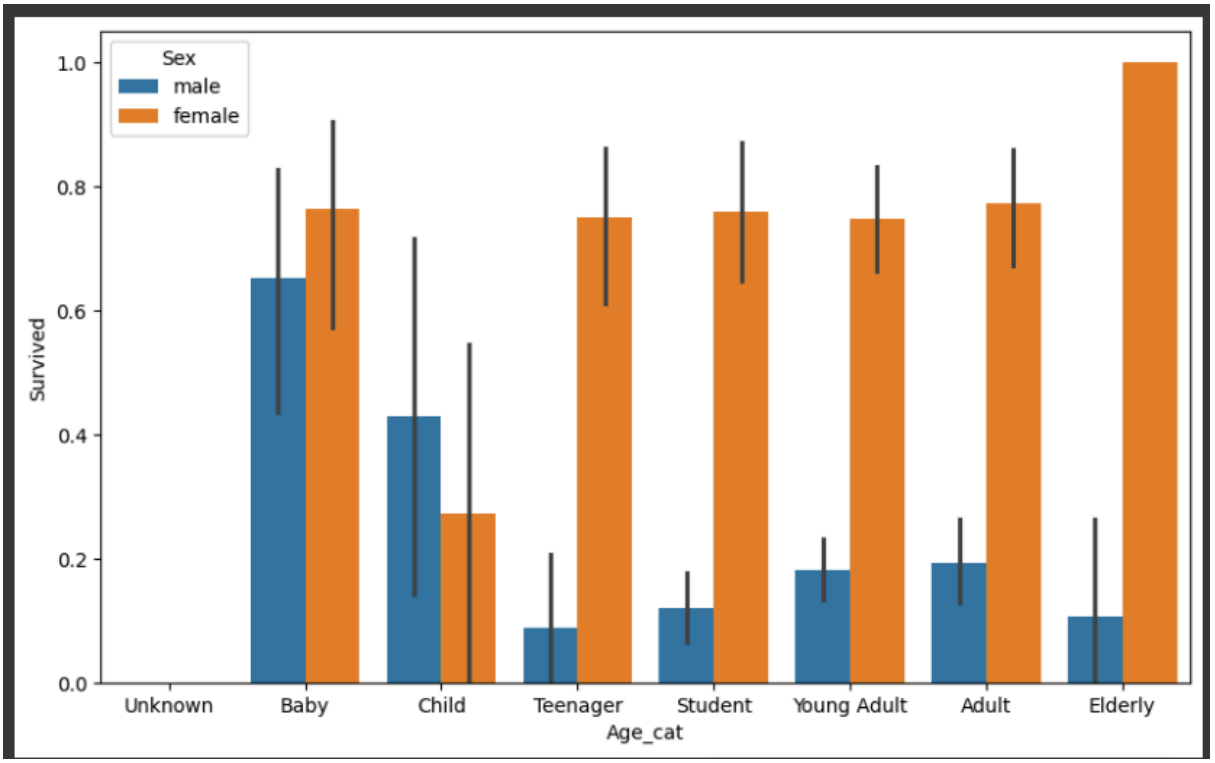


Age에 따른 생존 확률 알아보기

Age는 값 종류가 많은 데이터이므로, 범위별로 분류해 카테고리 값을 할당한다.

```
# 입력 age에 따라 구분 값을 반환하는 함수 설정.DataFrame의 apply lambda 식에 사용
def get_category(age):
  cat = ''
  if age <= -1: cat='Unknown'
  elif age<=5: cat='Baby'
  elif age<=12: cat='Child'
  elif age<=18: cat='Teenager'
  elif age<=25: cat='Student'
  elif age<=35: cat='Young Adult'
  elif age<=60: cat='Adult'
  else: cat='Elderly'

  return cat

# 막대그래프의 크기 figure을 더 크게 설정
plt.figure(figsize=(10, 6))

# X 축의 값을 순차적으로 표시하기 위한 설정
group_names = ['Unknown', 'Baby', 'Child', 'Teenager', 'Student', 'Young Adult', 'Adult', 'Elderly']

# lambda 식에 get_category() 함수를 반환값으로 지정
# get_category(x) 'Age' -> 'cat' 변환
titanic_df['Age_cat'] = titanic_df['Age'].apply(lambda x: get_category(x))
sns.barplot(x='Age_cat', y='Survived', hue='Sex', data=titanic_df, order=group_names)
titanic_df.drop('Age_cat', axis=1, inplace=True)
```

카테고리 값을 get_category()를 통해 할당한 뒤, 막대그래프로 표현한다.



LabelEncoder를 이용해 남아있는 문자열 카테고리 피처를 숫자형 카테고리 피처로 변환하기

LabelEncoder 객체는 카테고리 값의 유형 수에 따라 0~(카테고리 유형 수-1)까지의 숫자 값으로 변환한다. 사이킷런 전처리 모듈의 대부분의 인코딩 API는 사이킷런의 기본 프레임워크 API인 fit(), transform()으로 데이터를 변환한다.

```
from sklearn.preprocessing import LabelEncoder

def encode_features(dataDF):
  features=['Cabin', 'Sex', 'Embarked']
  for feature in features:
    le = LabelEncoder()
    le=le.fit(dataDF[feature])
    dataDF[feature]=le.transform(dataDF[feature])

  return dataDF

titanic_df = encode_features(titanic_df)
titanic_df.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	1	22.0	1	0	A/5 21171	7.2500	7	3
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	0	38.0	1	0	PC 17599	71.2833	2	0
2	3	1	3	Heikkinen, Miss. Laina	0	26.0	0	0	STON/O2. 3101282	7.9250	7	3
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	0	35.0	1	0	113803	53.1000	2	3
4	5	0	3	Allen, Mr. William Henry	1	35.0	0	0	373450	8.0500	7	3

데이터 전처리 함수 정리하기

- transform_features(): 데이터 전처리를 전체적으로 호출하는 함수
- drop_features(df): 불필요한 피처(PassengerId, Name, Ticket) 제거
- fillna(df): Null 값이 있는 칼럼을 다른 값으로 대체하는 함수

- format_features(df): 레이블 인코딩(문자형 카테고리→숫자형 카테고리) 수행하는 함수

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

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

# 레이블 인코딩 수행
def format_features(df):
  df['Cabin']=df['Cabin'].str[:1] # 첫 글자만 뽑아오기
  features = ['Cabin', 'Sex', 'Embarked']
  for feature in features:
    le = LabelEncoder() # 인코딩 초기화?를 위해 반복문 안에 넣은 것 같다
    le=le.fit(df[feature])
    df[feature]=le.transform(df[feature])
  return df

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

원본 데이터 다시 가공하기

클래스 결정값 데이터 세트: 타이타닉 생존자 데이터 세트의 레이블인 Survived 속성만 별도 분리

피처 데이터 세트: Survived 속성을 드롭한 DF

→ 피처 데이터 세트 가공(=전처리)

```
# 원본 데이터를 재로딩하고, 피처 데이터 세트와 레이블 데이터 세트 추출
titanic_df = pd.read_csv('/content/drive/MyDrive/Euron Homework/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)
```

train_test_split() API를 이용해 테스트 데이터 세트 추출하기

전체의 20%로 추출

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_titanic_df, y_titanic_df,
                                                    test_size=0.2, random_state=11)
```

결정 트리, 랜덤 포레스트, 로지스틱 회귀를 이용해 타이타닉 생존자 예측하기

** 로지스틱 회귀는 분류 알고리즘임

- 결정 트리 → DecisionTreeClassifier
- 랜덤 포레스트 → RandomForestClassifier
- 로지스틱 회귀 → LogisticRegression

학습 데이터와 테스트 데이터를 기반으로 머신러닝 모델을 학습(fit)라고, 예측(predict)한다.

예측 성능 평가는 정확도로 한다. - accuracy_score() API 사용

```python
from sklearn.tree import DecisionTreeClassifier # 결정 트리
from sklearn.ensemble import RandomForestClassifier # 랜덤 포레스트
from sklearn.linear_model import LogisticRegression # 로지스틱 회귀
from sklearn.metrics import accuracy_score # 정확도 성능 평가

# 결정 트리, Random Forest, 로지스틱 회귀를 위한 사이킷런 Classifier 클래스 생성
dt_clf = DecisionTreeClassifier(random_state=11) # 매번 같은 결과를 내기 위한 파라미터 설정
rf_clf = RandomForestClassifier(random_state=11)
lr_clf = LogisticRegression(solver='liblinear') # 작은 데이터 세트에서의 이진 분류는 liblinear가 성능(

# DecisionTreeClassifier 학습/예측/평가
dt_clf.fit(X_train, y_train)
dt_pred = dt_clf.predict(X_test)
print('DecisionTreeClassifier 정확도: {0:.4f}'.format(accuracy_score(y_test, dt_pred)))

# RandomForestClassifier 학습/예측/평가
rf_clf.fit(X_train, y_train)
rf_pred = rf_clf.predict(X_test)
print('RandomForestClassifier 정확도: {0:.4f}'.format(accuracy_score(y_test, rf_pred)))

# LogisticRegression 학습/예측/평가
lr_clf.fit(X_train, y_train)
lr_pred = lr_clf.predict(X_test)
print('LogisticRegression 정확도: {0:.4f}'.format(accuracy_score(y_test, lr_pred)))
```

```
DecisionTreeClassifier 정확도: 0.7877
RandomForestClassifier 정확도: 0.8547
LogisticRegression 정확도: 0.8659
```

최적화 작업을 수행하지 않았고, 데이터양도 충분하지 않기 때문에 이 평가로 알고리즘의 성능을 평가하긴 어려움

교차 검증으로 결정 트리 모델 더 평가하기

model_selection 패키지의 KFold 클래스, cross_val_score(), GridSearchCV 클래스 사용

1. KFold 클래스로 교차 검증 수행, # of Fold=5

```
from sklearn.model_selection import KFold

def exec_kfold(clf, folds=5):
  # 폴드 세트가 5개인 KFold 객체 생성, 폴드 수만큼 예측 결과를 저장하기 위해 리스트 객체 생성
  kfold=KFold(n_splits=folds)
  scores=[]

  # KFold 교차 검증 수행
  for iter_count, (train_index, test_index) in enumerate(kfold.split(X_titanic_df)):
    # X_titanic_df 데이터에서 교차 검증별로 학습과 검증 데이터를 가리키는 index 생성
    ## 이 인덱스를 통해 학습&검증 데이터 뽑기
    X_train, X_test = X_titanic_df.values[train_index], X_titanic_df.values[test_index]
    y_train, y_test = y_titanic_df.values[train_index], y_titanic_df.values[test_index]
    # Classifier 학습, 예측, 정확도 계산
    clf.fit(X_train, y_train)
    predictions = clf.predict(X_test)
    accuracy = accuracy_score(y_test, predictions)
    scores.append(accuracy)
    print("교차 검증 {0} 정확도: {1:.4f}".format(iter_count, accuracy))

  # 5개 fold에서의 평균 정확도 계산
  mean_score = np.mean(scores)
  print("평균 정확도: {0:.4f}".format(mean_score))

# exec_kfold 호출
exec_kfold(dt_clf, folds=5)
```

```
교차 검증 0 정확도: 0.7542
교차 검증 1 정확도: 0.7809
교차 검증 2 정확도: 0.7865
교차 검증 3 정확도: 0.7697
교차 검증 4 정확도: 0.8202
평균 정확도: 0.7823
```

2. cross_val_score(), 즉 StratifiedKFold로 교차 검증 수행

```
from sklearn.model_selection import cross_val_score # StratifiedKFold를 이용해 폴드 세트 분할

scores = cross_val_score(dt_clf, X_titanic_df, y_titanic_df, cv=5)
for iter_count, accuracy in enumerate(scores):
  print("교차 검증 {0} 정확도: {1:.4f}".format(iter_count, accuracy))

print("평균 정확도: {0:.4f}".format(np.mean(scores)))
```

```
교차 검증 0 정확도: 0.7430
교차 검증 1 정확도: 0.7753
교차 검증 2 정확도: 0.7921
교차 검증 3 정확도: 0.7865
교차 검증 4 정확도: 0.8427
평균 정확도: 0.7879
```

3. GridSearchCV 클래스로 최적 하이퍼 파라미터를 찾은 뒤 예측 성능 측정

   CV는 5개의 폴드 세트를 지정하고 하이퍼 파라미터는 max_depth, min_samples_split, min_samples_leaf를 변경하면서 성능을 측정한다. 최적 하이퍼

파라미터와 그때의 예측을 출력하고, 최적 하이퍼 파라미터로 학습된 Estimator를 이용해 위의 train_test_split()으로 분리된 테스트 데이터 세트에 예측을 수행해 예측 정확도를 출력한다.

```
from sklearn.model_selection import GridSearchCV

parameters = {'max_depth':[2, 3, 5, 10], 'min_samples_split':[2, 3, 5], 'min_samples_leaf':[1, 5, 8]}

grid_dclf = GridSearchCV(dt_clf, param_grid=parameters, scoring='accuracy', cv=5)
grid_dclf.fit(X_train, y_train)

print('GridSearchCV 최적 하이퍼 파라미터 :', grid_dclf.best_params_)
print('GridSearchCV 최고 정확도: {0:.4f}'.format(grid_dclf.best_score_))
best_dclf = grid_dclf.best_estimator_

# GridSearchCV의 최적 하이퍼 파라미터로 학습된 Estimator로 예측 및 평가 수행
dpredictions = best_dclf.predict(X_test)
accuracy = accuracy_score(y_test, dpredictions)
print('테스트 세트에서의 DecisionTreeClassifier 정확도: {0:.4f}'.format(accuracy))
```

```
GridSearchCV 최적 하이퍼 파라미터 : {'max_depth': 3, 'min_samples_leaf': 5, 'min_samples_split': 2}
GridSearchCV 최고 정확도: 0.7992
테스트 세트에서의 DecisionTreeClassifier 정확도: 0.8715
```

테스트용 데이터 세트가 작기 때문에 수치상으로 예측 성능이 많이 증가한 것처럼 보이는 것이다.

정리

머신러닝 애플리케이션

1. 데이터의 가공 및 변환 과정의 전처리 작업
2. 데이터를 학습 데이터와 테스트 데이터로 분리하는 데이터 세트 분리 작업
3. 학습 데이터를 기반으로 머신러닝 알고리즘을 적용해 모델 학습시킴
4. 학습된 모델을 기반으로 테스트 데이터에 대한 예측 수행
5. 예측된 결괏값을 실제 결괏값과 비교해 머신러닝 모델에 대한 평가를 수행

데이터 전처리 작업

머신러닝 알고리즘이 최적으로 수행되도록 데이터를 사전 처리하는 것

- 오류 데이터 보정, 결손값(Null) 처리 등 데이터 클렌징 작업
- 레이블 인코딩, 원-핫 인코딩 등 인코딩 작업

- 데이터의 스케일링/정규화 작업

- 머신러닝 모델은 학습 뒤 반드시 별도의 테스트 데이터 세트로 평가되어야 한다. 고정된 테스트 데이터 세트를 이용한 반복적인 모델의 학습 및 평가는 편향된 머신러닝 모델을 만들 가능성이 높다!

  → 이 문제를 해결하기 위해 KFold, StratifiedKFold, cross_val_score() 등 교차 검증 클래스 및 함수 제공

  ** 최적 하이퍼 파라미터를 교차 검증을 통해 추출: GridSearchCV

평가

머신러닝: 데이터 가공/변환 → 모델 학습/예측 → 평가

- 성능 평가 지표
  - 회귀: 실제값과 예측값의 오차 평균값에 기반
  - 분류: 일반적으로는 실제 결과 데이터와 예측 결과 데이터가 얼마나 정확하고 오류가 적게 발생하는가에 기반하지만, 정확도만 가지고 평가하면 잘못된 평가가 될 수 있음
    - 분류의 성능 평가 지표: 정확도(Accuracy), 오차행렬(Confusion Matrix), 정밀도(Precision), 재현율(Recall), F1 스코어, ROC AUC

분류는 결정 클래스 값 종류 유형에 따라 이진 분류(긍정/부정), 멀티 분류(여러 값)로 나뉨

정확도(Accuracy)

실제 데이터에서 예측 데이터가 얼마나 같은지 판단하는 지표

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

정확도 지표가 ML 모델의 성능을 왜곡하는 예제

오리지널 데이터를 반영하지 않고, 같은 결과 분포가 나오는 분석이면 정확도가 비슷할 수 있다.

사이킷런의 BaseEstimator 클래스를 상속받아 아무런 학습 없이, 성별에 따라 생존자를 예측하는 단순한 Classifier를 생성한다.

- BaseEstimator: Customized 형태의 Estimator를 개발자가 생성할 수 있음
- MyDummyClassifier: fit()은 아무것도 수행하지 않고, predict()는 Sex 피처가 1이면 0, 0이면 1로 예측하는 단순한 Classifier

```
from sklearn.base import BaseEstimator

class MyDummyClassifier(BaseEstimator):
  # fit() 메소드는 아무것도 학습하지 않음
  def fit(self, X, y=None):
    pass
  # predict()
  def predict(self, X):
    pred = np.zeros((X.shape[0], 1))
    for i in range(X.shape[0]): # 데이터 수만큼 수행
      if X['Sex'].iloc[i]==1:
        pred[i]=0
      else:
        pred[i]=1
    return pred
```

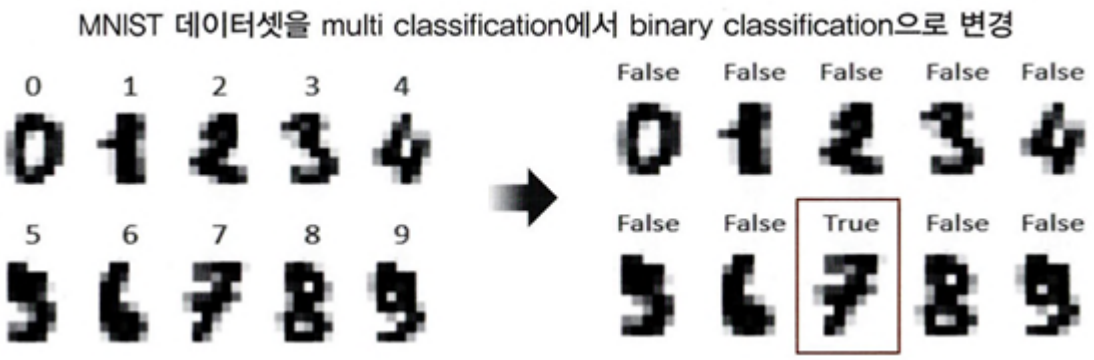
```
titanic_df = pd.read_csv('/content/drive/MyDrive/Euron Homework/titanic_train.csv')
y_titanic_df = titanic_df['Survived']
X_titanic_df = titanic_df.drop('Survived', axis=1)
X_titanic_df = transform_features(X_titanic_df)
X_train, X_test, y_train, y_test=train_test_split(X_titanic_df, y_titanic_df,test_size=0.2, random_sta

# 위에서 생성한 Dummy Classifier를 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)))
```

```
Dummy Classifier의 정확도는: 0.7877
```

⚠️ 이렇게 단순한 알고리즘으로 예측을 해도 데이터의 구성에 따라 정확도 결과가 꽤 높을 수 있다. 정확도는 불균형한 레이블 값 분포에서 ML 모델의 성능을 판단할 땐 적합한 평가 지표가 아니다.

MNIST 데이터 세트를 변환해 불균형한 데이터 세트로 만든 뒤, 정확도 지표 적용해보기

** MNIST 데이터셋: 0~9 숫자 이미지 픽셀 정보 - 사이킷런은 load_digits() API로 제공



전체 데이터의 10%만 True, 90%는 False인 불균형한 데이터 세트로 변형함

→ 모든 데이터를 False, 0으로 예측하는 classifier를 이용해 정확도를 측정하면 90%

```
from sklearn.datasets import load_digits # 데이터 import
from sklearn.model_selection import train_test_split
from sklearn.base import BaseEstimator # 개발자가 커스터마이즈할 수 있는 Estimator
from sklearn.metrics import accuracy_score  # 정확도 성능 평가

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

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

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

# digits 번호가 7번이면 True, 이를 astype(int)로 1로 변환, 아니면 False고 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 0과 1의 분포도')
print(pd.Series(y_test).value_counts())

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

```
레이블 테스트 세트 크기: (450,)
테스트 세트 레이블 0과 1의 분포도
0    405
1     45
Name: count, dtype: int64
모든 예측을 0으로 하여도 정확도는:0.900
```

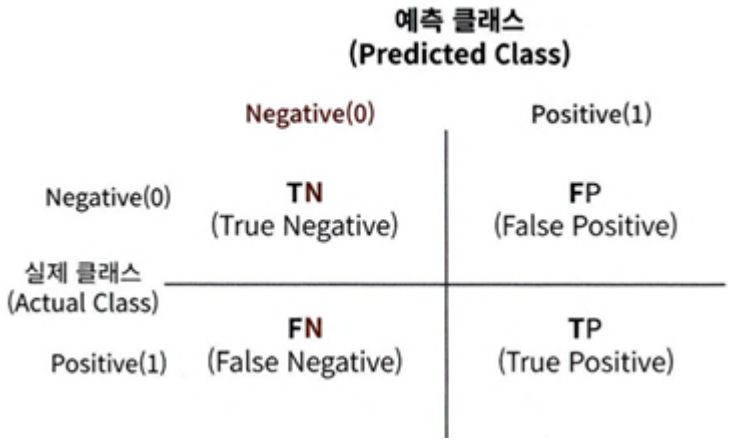
단순히 predict()의 결과를 np.zeros()로 모두 0으로 반환하는데, 450개 테스트 데이터 세트에 수행한 예측 정확도가 90%

→ 불균형한 레이블 데이터 세트에서는 성능 수치로 정확도 평가 지표를 사용하면 안 된다!

오차 행렬

학습된 분류 모델이 예측을 수행하며 얼마나 헷갈리고(confused) 있는지도 함께 보여주는 지표

True/False, Positive/Negative의 4분면 행렬에서 실제 레이블 클래스 값과 예측 레이블 클래스 값이 어떠한 유형을 가지고 매핑되는지, 분류 모델 예측 성능의 오류가 어떤 모습으로 발생하는지를 나타낸다.



네 가지 유형은 예측 클래스와 실제 클래스의 Positive 결정 값(1)과 Negative 결정 값(0)의 결합에 따라 결정

- TN: 예측값 Negative 0, 실제 값 Negative 0
- FP: 예측값 Positive 1, 실제 값 Negative 0
- FN: 예측값 Negaitve 0, 실제 값 Positive 1
- TP: 예측값 Positive 1, 실제 값 Positive 1

사이킷런 - confusion_matrix() API 제공

confusion_matrix()를 이용해 MyFakeClassifier의 예측 성능 지표를 오차 행렬로 표현하기

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_test, fakepred)
```

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

**이진 분류의 TN, FP, FN, FP는 도표와 동일한 위치로 array에서 추출할 수 있음

→ TN=405, FP=0, FN=45, TP=0

→ 전체 450건 데이터 중 Negative 0로 예측해서 True가 된 결과 405건

Negative 0으로 예측해서 False가 된 결과 45건

⇒ 정확도, 정밀도(Precision), 재현율(Recall) 값을 알 수 있음

오차 행렬의 정확도

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

불균형한 레이블 클래스를 가지는 이진 분류 모델
: 목표가 되는 적은 수의 결괏값에 Positive 설정, 그렇지 않은 일반적인 경우 Negative로 설정하는 경우가 많다. 불균형한 이진 분류 데이터 세트에서는 Positive 데이터 건수가 매우 작으므로, 데이터에 기반한 ML 알고리즘은 Positive보다는 Negative로 예측 정확도가 높아지는 경향이 발생한다.
대부분이 Negative라서 Negative로 예측한 것인데, 정작 목표가 되는 데이터 예측에 실패해도 TN 수치 때문에 정확도 자체는 높아지는 현상 발생

정밀도와 재현율

Positive 데이터 세트의 예측 성능에 더 초점을 맞춘 평가 지표

- 정밀도 (양성 예측도) = TP/(FP+TP)
  : 예측을 Positive로 한 대상 중 예측과 실제 값 모두 Positive로 일치한 데이터의 비율
  - 실제 Negative 음성 데이터를 Positive로 잘못 판단하는 경우 큰 문제가 생길 때 정밀도가 중요!
  - 사이킷런 API: precision_score()
- 재현율 = TP/(FN+TP)
  : 실제 값이 Positive인 대상 중 예측과 실제 값이 Positive로 일치한 데이터의 비율
  - 실제 Positive 양성 데이터를 Negative로 잘못 판단하는 경우 큰 문제가 생길 때 재현율이 중요!
  - 사이킷런 API: recall_score()

→ Positive로 예측하고 실제로 Positive인 데이터가 많아야 정밀도와 재현율이 올라감

타이타닉 예제로 오차 행렬, 정밀도, 재현율을 모두 구해서 예측 성능 평가하기

모든 평가 지표를 한꺼번에 호출하는 get_clf_eval()

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

def get_clf_eval(y_test, pred):
  confusion = confusion_matrix(y_test, pred)
  accuracy = accuracy_score(y_test, pred)
  precision = precision_score(y_test, pred)
  recall = recall_score(y_test, pred)
  print('오차 행렬')
  print(confusion)
  print('정확도: {0:.4f}, 정밀도: {1:.4f}, 재현율: {2:.4f}'.format(accuracy, precision, recall))
```

로지스틱 회귀 기반으로 타이타닉 생존자를 예측하고 각종 평가 수행

```
from sklearn.linear_model import LogisticRegression

# 원본 데이터를 재로딩, 데이터 가공, 학습 데이터/테스트 데이터 분할
titanic_df = pd.read_csv('/content/drive/MyDrive/Euron Homework/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_
lr_clf = LogisticRegression(solver='liblinear')

lr_clf.fit(X_train, y_train) # 학습
pred = lr_clf.predict(X_test) # 예측
get_clf_eval(y_test, pred) # 평가
```

```
오차 행렬
[[108  10]
 [ 14  47]]
정확도: 0.8659, 정밀도: 0.8246, 재현율: 0.7705
```

정밀도/재현율 트레이드오프(Trade-off)

정밀도와 재현율은 상호 보완적인 평가 지표이기 때문에, 분류의 결정 임곗값(Threshold)을 조정해 정밀도 또는 재현율 하나의 수치를 높이면 다른 하나의 수치는 떨어지기 쉽다는 것

- 사이킷런의 분류 알고리즘
  : 개별 레이블 별로 결정 확률 계산
  predict_proba() → 예측 확률이 큰 레이블 값으로 예측
- predict_proba()

입력 파라미터	predict( ) 메서드와 동일하게 보통 테스트 피처 데이터 세트를 입력
반환 값	개별 클래스의 예측 확률을 ndarray m x n (m: 입력값의 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 array와 예측 결괏값 array를 병합해 예측 확률과 결괏값을 한눈에 확인
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])
```

```
pred_proba() 결과 Shape: (179, 2)
pred_proba array에서 앞 3개만 샘플로 추출
: [[0.44935227 0.55064773]
 [0.86335512 0.13664488]
 [0.86429645 0.13570355]]
두 개의 class 중에서 더 큰 확률을 클래스 값으로 예측
 [[0.44935227 0.55064773 1.        ]
 [0.86335512 0.13664488 0.        ]
 [0.86429645 0.13570355 0.        ]]
```

첫 번째 칼럼 값+두 번째 칼럼 값 = 1

두 칼럼 중 더 큰 확률 값으로 predict()가 최종 예측함

사이킷런의 정밀도/재현율 트레이드오프 방식을 이해하기

```
from sklearn.preprocessing import Binarizer

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

# X의 개별 원소들이 threshold 값보다 같거나 작으면 0을, 크면 1을 반환
binarizer = Binarizer(threshold=1.1)
print(binarizer.fit_transform(X))
```

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

predict()의 의사코드 만들기

```
# Binarizer의 threshold 설정값. 분류 결정 임곗값
custom_threshold = 0.5

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

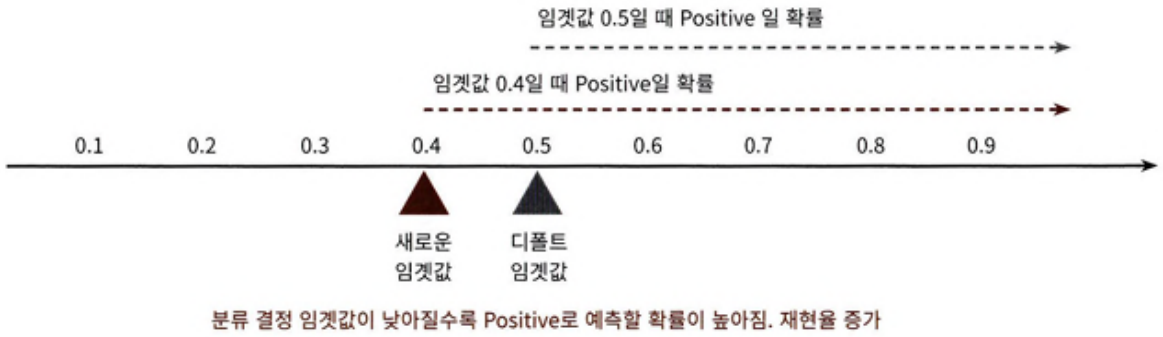
```
오차 행렬
[[108  10]
 [ 14  47]]
정확도: 0.8659, 정밀도: 0.8246, 재현율: 0.7705
```

분류 결정 임곗값을 낮춰보기

```
custom_threshold=0.4
pred_proba_1 = pred_proba[:, 1].reshape(-1,1)
binarizer = Binarizer(threshold=custom_threshold).fit(pred_proba_1)
custom_predict = binarizer.transform(pred_proba_1)

get_clf_eval(y_test, custom_predict)
```

```
오차 행렬
[[97 21]
 [11 50]]
정확도: 0.8212, 정밀도: 0.7042, 재현율: 0.8197
```

→ 임곗값을 낮추면 재현율 값이 올라가고 정밀도가 떨어짐: 분류 결정 임곗값은 Positive 예측값을 결정하는 확률의 기준이 되므로, 임곗값을 낮추면 Positive 예측이 많아짐.



분류 결정 임곗값이 낮아질수록 Positive로 예측할 확률이 높아짐. 재현율 증가

[임곗값 0.5일 때 오차 행렬]

TN	FP
108	10
FN	TP
14	47

[임곗값 0.4일 때 오차 행렬]

TN	FP
97	21
FN	TP
11	50

F1 스코어 (Score)

: 정밀도와 재현율이 어느 한쪽으로 치우치지 않을 때 상대적으로 높은 값을 가짐

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

→ 사이킷런 API f1_score()

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

```
F1 스코어: 0.7966
```

```
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, re

thresholds = [0.4, 0.45, 0.50, 0.55, 0.60]
pred_proba = lr_clf.predict_proba(X_test)
get_eval_by_threshold(y_test, pred_proba[:, 1].reshape(-1, 1), thresholds)
```

```
임곗값: 0.4
오차 행렬
[[97 21]
 [11 50]]
정확도: 0.8212, 정밀도: 0.7042, 재현율: 0.8197, F1:0.7576
임곗값: 0.45
오차 행렬
[[105  13]
 [ 13  48]]
정확도: 0.8547, 정밀도: 0.7869, 재현율: 0.7869, F1:0.7869
임곗값: 0.5
오차 행렬
[[108  10]
 [ 14  47]]
정확도: 0.8659, 정밀도: 0.8246, 재현율: 0.7705, F1:0.7966
임곗값: 0.55
오차 행렬
[[111   7]
 [ 16  45]]
정확도: 0.8715, 정밀도: 0.8654, 재현율: 0.7377, F1:0.7965
임곗값: 0.6
오차 행렬
[[113   5]
 [ 17  44]]
정확도: 0.8771, 정밀도: 0.8980, 재현율: 0.7213, F1:0.8000
```

ROC 곡선과 AUC

- ROC(Receiver Operation Characteristic Curve)=수신자 판단 곡선

  : FPR(False Positive Rate)에 따른 TPR(True Positive Rate)의 변화를 나타내는 곡선

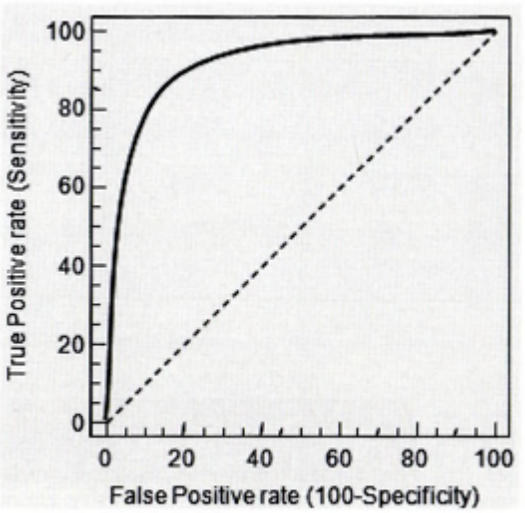
$$FPR = FP/(FP + TN) = 1 - TNR(\text{특이성})$$

  ** TPR(True Positive Rate)=재현율=민감도
  : 실제값 Positive가 정확히 예측돼야 하는 수준

  ** TNR(True Negative Rate)=특이성(Specificity)
  : 실제값 Negative가 정확히 예측돼야 하는 수준

  *** 사이킷런 API roc_curve()

$$TNR = TN/(FP + TN)$$



〈 ROC 곡선 예시 〉

- 대각선=완전 랜덤 수준의 이진 분류 ROC 직선 (AUC: 0.5)
  → ROC 곡선이 가운데 직선에 가까울수록 성능이 떨어지는 것, 멀수록 성능이 뛰어난 것

ROC 곡선은 FPR을 0부터 1까지 변경하면서 TPR의 변화 값을 구한다!

- FPR을 0으로 만들기=분류 결정 임곗값을 1로 지정
- FPR을 1로 만들기=TN을 0로 만들기 (→FP/FP=1)

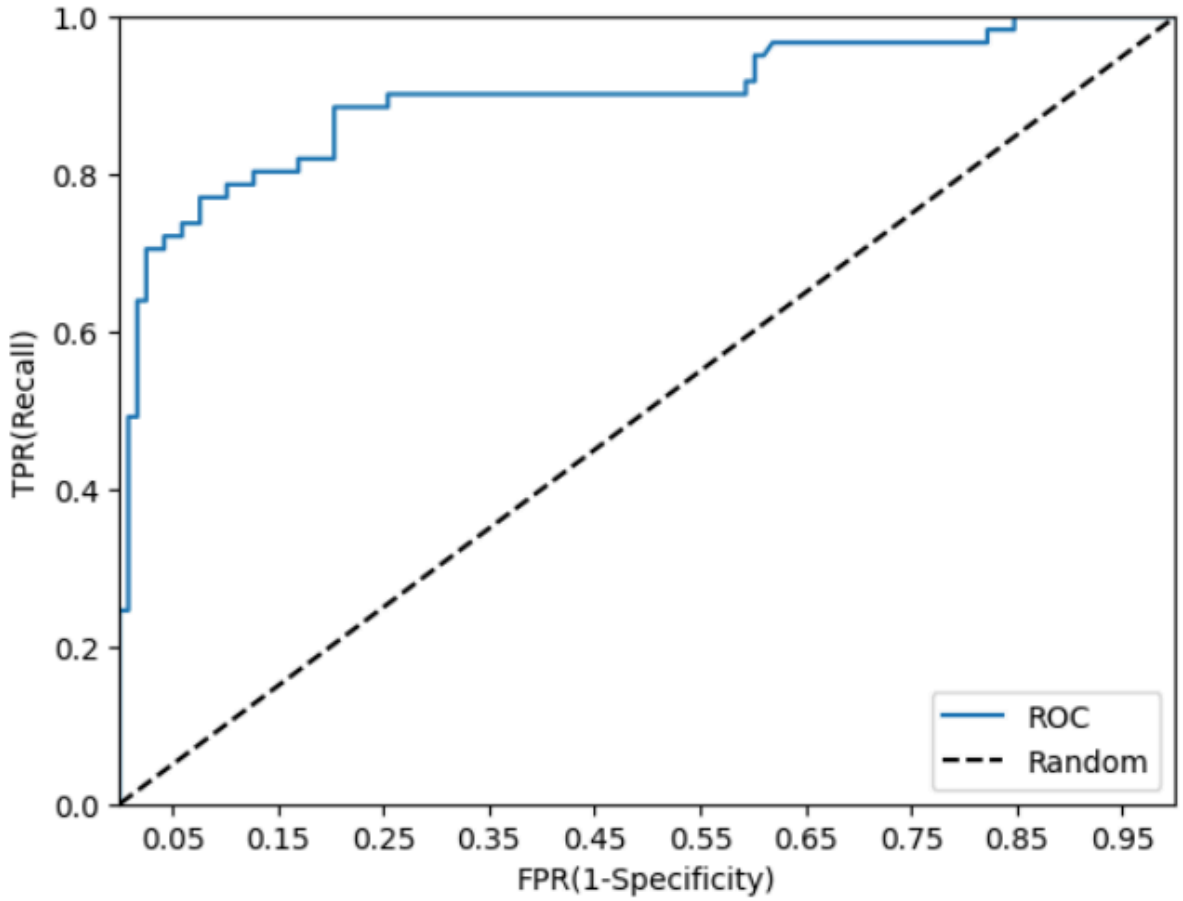
roc_curve() API를 이용해 타이타닉 생존자 예측 모델의 FPR, TPR, 임곗값 구하기

```
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 임곗값 배열에서 샘플로 데이터를 추출하되, 임곗값을 5 Step으로 추출
# thresholds[0]은 max(예측확률)+1로 임의 설정됨. 이를 제외하기 위해 np.arange는 1부터 시작
thr_index = np.arange(1, thresholds.shape[0], 5)

print('샘플 추출을 위한 임곗값 배열의 index:', thr_index)
print('샘플 index 로 추출한 임곗값:', np.round(thresholds[thr_index], 2))

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

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

- 임곗값: 1→작아짐
- FPR: 0→커짐
- TPR: FPR이 조금씩 커질 때 TPR이 가파르게 커짐

ROC 곡선으로 시각화하기



ROC 곡선 면적에 기반한 AUC 값을 분류의 성능 지표로 사용함

- AUC(Area Under Curve): ROC 곡선 밑의 면적을 구한 것. 1에 가까울수록(→대각선과 멀어짐) 좋은 수치
    - 대각선 직선의 랜덤 수준 이진 분류 AUC 값=0.5

```
from sklearn.metrics import roc_auc_score
pred_proba = lr_clf.predict_proba(X_test)[:, 1]
roc_score = roc_auc_score(y_test, pred_proba)
print('ROC AUC 값: {0:.4f}'.format(roc_score))
```

```
ROC AUC 값: 0.8987
```

get_clf_eval() 함수에 ROC AUC 값 추가하기

```python
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)
  # AUC score 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))
```

정리

✅ 성능 평가 지표: 정확도, 오차 행렬, 정밀도, 재현율, F1 스코어, ROC-AUC

→ 데이터 분류의 목적에 따라 정밀도, 재현율, F1, ROC-AUC 지표를 적절하게 사용해야 한다