1️⃣

week1_2장:사이킷런으로 시작하는 머신러닝

01. 사이킷런 소개와 특징

- 파이썬 머신러닝 라이브러리 중 가장 많이 사용되는 라이브러리
- 머신러닝을 위한 다양한 알고리즘과 개발을 위한 편리한 프레임워크와 API를 제공함

02. 첫 번째 머신러닝 만들어 보기 - 붓꽃 품종 예측하기

: 붓꽃 데이터 세트는 꽃잎의 길이와 너비, 꽃받침의 길이와 너비(**피처)**를 기반으로 꽃의 품종 **(label, target)**을 예측(분류)하기 위한 것

> 💡 **분류:** 대표적 지도학습 방법 중 하나
>
> **지도학습:** 학습을 위한 다양한 **피처**와 분류 결정값인 **레이블(Label)** 데이터로 모델을 학습한 뒤, 별도의 테스트 데이터 세트에서 미지의 레이블을 예측함.
> ⇒ 즉, 지도학습은 명확한 정답이 주어진 데이터를 먼저 학습한 뒤 미지의 정답을 예측하는 방식
> ⇒ 학습을 위해 주어진 데이터 세트 - 학습 데이터 세트 / 머신러닝 모델의 예측 성능 평가 위해 별도로 주어진 데이터 세트 - 테스트 데이터 세트

`sklearn.datasets` 사이킷런에서 자체적으로 제공하는 데이터 세트를 생성하는 모듈

`sklearn.tree` 트리 기반 ML 알고리즘을 구현하는 클래스의 모임

`sklearn.model_selection` 학습 데이터와 검증 데이터, 예측 데이터로 데이터를 분리하거나 최적의 하이퍼 파라미터로 평가하기 위한 다양한 모듈의 모임

(`하이퍼파라미터` 란 머신러닝 알고리즘별로 최적의 학습을 위해 직접 입력하는 파라미터들을 통칭, 하이퍼 파라미터를 통해 머신러닝 알고리즘의 성능 튜닝 가능함)

붓꽃 데이터 세트를 생성하기 위해 `load_iris()` 이용. ML 알고리즘은 의사 결정 트리 알고리즘으로, 이를 구현한 `DecisionTreeClassifier`를 적용함

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split  # 데이터 세트 분리를 위한 것
```

```
import pandas as pd

# 붓꽃 데이터 세트 로딩
iris = load_iris()
```

```
# iris.data는 Iris 데이터 세트에서 피처(feature)(꽃잎의 길이와 너비, 꽃받침의 길이와 너비)만으로 된 데이터
를 numpy로 가지고 있습니다.
iris_data = iris.data
print(iris_data)
''' iris dataset의 피처: 꽃잎의 길이와 너비, 꽃받침의 길이와 너비
[[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]
 [5.  3.4 1.5 0.2]
 [4.4 2.9 1.4 0.2]
 [4.9 3.1 1.5 0.1]
 [5.4 3.7 1.5 0.2]
 [4.8 3.4 1.6 0.2]
 [4.8 3.  1.4 0.1]
 [4.3 3.  1.1 0.1]
 [5.8 4.  1.2 0.2]
 [5.7 4.4 1.5 0.4]
'''

# iris.target은 붓꽃 데이터 세트에서 레이블(결정 값) 데이터를 numpy로 가지고 있습니다.
iris_label = iris.target
print('iris target값:', iris_label)
''' 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] '''
print('iris target명:', iris.target_names)
# iris target명: ['setosa' 'versicolor' 'virginica'] -> 순서대로 0, 1, 2

# 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)
```

	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

column에 feature 변수와 label(결정값 or target_names)이 들어감

`df = pd.DataFrame(data, index, columns, dtype)`

- data - DataFrame을 생성할 데이터
- index - 각 Row에 대해 Label을 추가 (옵션)
- columns - 각 Column에 대해 Label을 추가 (옵션)
- dtype - 각 Column의 데이터 타입 명시 (옵션)

https://wooono.tistory.com/80 (출처)

학습용 데이터와 테스트용 데이터 분리(학습 데이터로 학습된 모델의 성능 평가하려면 테스트 데이터 세트 필요함)

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

`train_test_split(target, label, test_size, random_state)`

- target - 피처 데이터 세트 (여기선 iris_data)
- label - 레이블 데이터 세트 (여기선 iris_label)
- test_size - 전체 데이터 세트 중 테스트 데이터 세트의 비율
- random_state - 호출할 때마다 같은 학습/테스트 용 데이터 생성 위해 주어지는 난수 발생 값

위에서 확보한 학습 데이터 기반으로 머신러닝 분류 알고리즘의 하나인 의사결정 트리를 이용해 학습과 예측 수행.

```
# DecisionTreeClassifier 객체 생성
dt_clf = DecisionTreeClassifier(random_state = 11)
# 동일한 학습/예측 결과 출력을 위해 random_state 설정

# 학습 수행
dt_clf.fit(X_train, y_train)
# DecisionTreeClassifier(random_state=11)
```

학습된 객체를 이용해 예측 수행. (예측은 학습 데이터가 아닌 다른 데이터 사용해야함) → predict( ) 메서드에 테스트용 피처 데이터 세트 입력 → 테스트 데이터 세트에 대한 예측값 반환

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

예측 결과를 기반으로 모델의 예측 성능 평가 → 정확도 측정(예측 결과가 실제 레이블 값과 얼마나 정확하게 맞는지 평가)

`accuracy_score(실제 레이블 데이터 세트(실제값), 예측 레이블 데이터 세트(예측값))`

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

💡 붓꽃 데이터 세트로 분류를 예측한 프로세스 정리
1. 데이터 세트 분리: 데이터를 학습 데이터와 테스트 데이터로 분리
2. 모델 학습: 학습 데이터를 기반으로 ML 알고리즘을 적용해 모델을 학습시킴
3. 예측 수행: 학습된 ML 모델을 이용해 테스트 데이터의 분류(즉, 붓꽃 종류)를 예측함
4. 평가: 이렇게 예측된 결괏값과 테스트 데이터의 실제 결과값을 비교해 ML 모델의 성능을 평가함

03. 사이킷런의 기반 프레임워크 익히기

Estimator 이해 및 fit( ), predict( ) 메서드

모델 학습: `fit()` 모델 예측: `predict()`

사이킷런에서는 분류 알고리즘을 구현한 클래스를 Classifier로, 회귀 알고리즘을 구현한 클래스를 Regressor로 지칭함. → 지도학습의 모든 알고리즘을 구현한 클래스를 통칭해 **Estimator**라고 부름

`cross_val_score()` (evaulation 함수), `GridSearchCV` (하이퍼 파라미터 튜닝 지원 클래스) → **Estimator**를 인자로 받음→ 인자로 받은 Estimator에 대해서 `cross_val_score()`, `GridSearchCV.fit()` 함수 내에서 이 Estimator의 fit()과 predict()를 호출해서 평가를 하거나 하이퍼 파라미터 튜닝을 수행함.

사이킷런에서 **비지도학습**인 차원 축소, 클러스터링, 피처 추출(Feature Extraction) 등을 구현한 클래스 역시 대부분 fit()(-지도학습의 fit()과 달리, 입력 데이터의 형태에 맞춰 데이터를 변환하기 위한 사전 구조를 맞추는 작업)과 transform()(-fit으로 사전 구조를 맞추면 이후 입력 데이터의 차원 변환, 클러스터링, 피처 추출 등의 실제 작업은 transform()으로 수행함)을 적용함.

사이킷런의 주요 모듈

분류	모듈명	설명
예제 데이터	`sklearn.datasets`	사이킷런에 내장되어 **예제로 제공하는 데이터 세트**
피처 처리	`sklearn.preprocessing`	**데이터 전처리**에 필요한 다양한 가공 기능 제공(문자열을 숫자형 코드 값으로 **인코딩, 정규화, 스케일링** 등)
	`sklearn.feature_selection`	알고리즘에 큰 영향을 미치는 피처를 우선순위대로 셀렉션 작업을 수행하는 다양한 기능 제공
	`sklearn.feature_extraction`	텍스트 데이터나 이미지 데이터의 벡터화된 피처를 추출하는데 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 모듈. 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(GridSearch)로 최적 파라미터 추출 등의 API 제공
평가	`sklearn.metrics`	분류, 회귀, 클러스터링, 페어와이즈에 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_mode`	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)

분류	모듈명	설명
유틸리티	`sklearn.pipeline`	피처 처리 등의 변환과 ML 알고리즘 학습, 예측 등을 함께 묶어서 실행할 수 있는 유틸리티 제공

일반적으로 머신러닝 모델을 구축하는 주요 프로세스는 피처의 가공, 변경, 추출을 수행하는 **피처 처리(feature processing)**, **ML 알고리즘 학습/예측 수행**, 그리고 **모델 평가**의 단계를 반복적으로 수행하는 것.

내장된 예제 데이터 세트

회귀용 예제와, 분류나 클러스터링을 위한 예제로 나뉨

fetch 계열의 명령은 데이터의 크기가 커서 패키지에 처음부터 저장돼 있지 않고 인터넷에서 내려받아 저장한 후 불러들이는 데이터이다.

⬇️ 분류와 클러스터링을 위한 표본 데이터 생성기

API 명	설명
`datasets.make_classifications()`	분류를 위한 데이터 세트를 만듦. 특히 높은 상관도, 불필요한 속성 등의 노이즈 효과를 위한 데이터를 무작위로 생성해 줌.
`datasets.make_blobs()`	클러스터링을 위한 데이터 세트를 무작위로 생성해 줌. 군집 지정 개수에 따라 여러가지 클러스터링을 위한 데이터 세트를 쉽게 만들어줌.

⬇️ 분류나 회귀 연습용 예제 데이터

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

→ 사이킷런에 내장된 데이터 세트는 일반적으로 딕셔너리 형태로 돼 있음. 키는 보통 data, target, target_name, feature_names, DESCR로 구성됨.

- data: 피처의 데이터 세트를 가리킴 → 넘파이 배열(ndarray) 타입
- target: 분류 시 레이블 값, 회귀일 때는 숫자 결괏값 데이터 세트 → 넘파이 배열(ndarray) 타입

- target_names: 피처의 이름을 나타냄 → 넘파이 배열 또는 파이썬 리스트 형태
- DESCR: 데이터 세트에 대한 설명과 각 피처의 설명을 나타냄 → 넘파이 배열 또는 파이썬 리스트 형태

→ 피처 데이터 값을 반환받기 위해서는, 내장 데이터 세트 api를 호출한 뒤 그 key 값을 지정하면 됨.

```
from sklearn.datasets import load_iris

iris_data = load_iris()
print(type(iris_data))
# <class 'sklearn.utils.Bunch'> - 파이썬 딕셔너리와 비슷한 형태 -> 즉, key값 확인 가능
```

```
데이터 세트의 key 값 확인
keys = iris_data.keys()
print('붓꽃 데이터 세트의 키들:', keys)
# 붓꽃 데이터 세트의 키들: dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'fea
ture_names'])
```

데이터 세트가 딕셔너리 형태이기 때문에 피처 데이터 값을 추출하기 위해서는 데이터세트.data(혹은 데이터세트['data'])를 이용하면 됨. 나머지 키 값들도 동일하게 추출하면 됨.

```
각 키가 가리키는 값 출력
print('\n feature_names 의 type:',type(iris_data.feature_names))
print(' feature_names 의 shape:',len(iris_data.feature_names))
print(iris_data.feature_names)

print('\n target_names 의 type:',type(iris_data.target_names))
print(' feature_names 의 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_names 의 type: <class 'list'>
 feature_names 의 shape: 4
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

```
 target_names 의 type: <class 'numpy.ndarray'>
 feature_names 의 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]
...
 [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 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]
'''
```

4. Model Selection 모듈 소개

학습/테스트 데이터 세트 분리 - train_test_split()

`train_test_split()`

첫 번째 파라미터: 피처 데이터 세트, 두번째: 레이블 데이터 세트, 이후는 파라미터 선택

- test_size: 전체 데이터에서 테스트 데이터 세트 크기를 얼마로 샘플링할 것인가 결정. 디폴트는 0.25
- train_size: 전체 데이터에서 학습용 데이터 세트 크기를 얼마로 샘플링할 것인가를 결정. 주로 test_size를 사용하기에, 잘 사용되지 않음.
- `shuffle`: 데이터를 분리하기 전에 데이터를 미리 섞을지를 결정함. 디폴트는 True. 데이터를 분산시켜서 좀 더 효율적인 학습 및 테스트 데이터 세트를 만드는 데 사용됨.
- random_state: 호출할 때마다 동일한 학습/테스트용 데이터 세트를 생성하기 위해 주어지는 난수 값. train_test_split( )는 호출할 때마다 무작위로 데이터 분리해서 이 값 지정해주지 않으면 수행할 때마다 다른 데이터 생성하게 됨.
- train_test_split( )의 반환값은 튜플 형태. 순차적으로 학습용 데이터의 피처 데이터 세트, 테스트용 데이터의 피처 데이터 세트, 학습용 데이터의 레이블 데이터 세트, 테스트용 데이터의 레이블 데이터 세트가 반환됨

```
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)))
```

→ 붓꽃 데이터 150개의 데이터로 데이터 양이 크지 않아서, 테스트 데이터는 45개밖에 되지 않음. 이를 통해 알고리즘의 예측 성능을 판단하기에는 적절하지 않음. 학습을 위한 데이터의 양을 일정 수준 이상으로 보장하는 것도 중요하지만, 학습된 모델에 대해 다양한 데이터를 기반으로 예측 성능을 평가하는 것도 중요함.

교차 검증

별도의 테스트용 데이터로 예측 성능 평가 → **과적합(Overfitting)**의 문제 발생 가능성 있음

과적합(Overfitting): 모델이 학습 데이터에만 과도하게 최적화되어, 실제 예측을 다른 데이터로 수행할 경우 예측 성능이 과도하게 떨어지는 것.

⇒ **개선: 교차 검증**

데이터의 편증을 막기 위해서 별도의 **여러 세트로 구성된 학습 데이터 세트와 검증 데이터 세트에서 학습과 평가를 수행**하는 것. 그리고 각 세트에서 수행한 평가 결과에 따라 하이퍼 파라미터 튜닝 등의 모델 최적화 진행.

대부분의 ML 모델의 성능 평가는 **1) 교차 검증 기반으로 1차 평가**를 한 뒤에 최종적으로 **2)테스트 데이터 세트에 적용해 평가**하는 프로세스. → ML에 사용되는 데이터 세트를 세분화해서 학습, 검증, 테스트 데이터 세트로 나눌 수 있음. 테스트 데이터 세트 외에 별도의 검증 데이터 세트를 둬서 최종 평가 이전에 학습된 모델을 다양하게 평가하는데 사용함. (즉, **학습데이터를 다시 분할**하여 학습 데이터와 학습된 모델의 성능을 1차 평가하는 **검증 데이터로 나눔**. 테스트 세트는 모든 학습/검증 과정이 완료된 후 최종적으로 성능을 평가하기 위한 데이터 세트)

k 폴드 교차 검증 (보편적 교차 검증 기법)

(훈련 데이터 k개로 쪼개로 나머지 1/k를 훈련하는 과정에서 검증용으로 쓰는)

K개의 데이터 폴드 세트를 만들어서 K번만큼 각 폴드 세트에 학습과 검증 평가를 반복적으로 수행하는 방법.

예를 들어 K가 5라고 하면, 5개의 폴드된 데이터 세트를 학습과 검증을 위한 데이터 세트로 변경하면서 5번 평가를 수행한 뒤, 이 5개의 평가를 평균한 결과를 가지고 예측 성능을 평가함

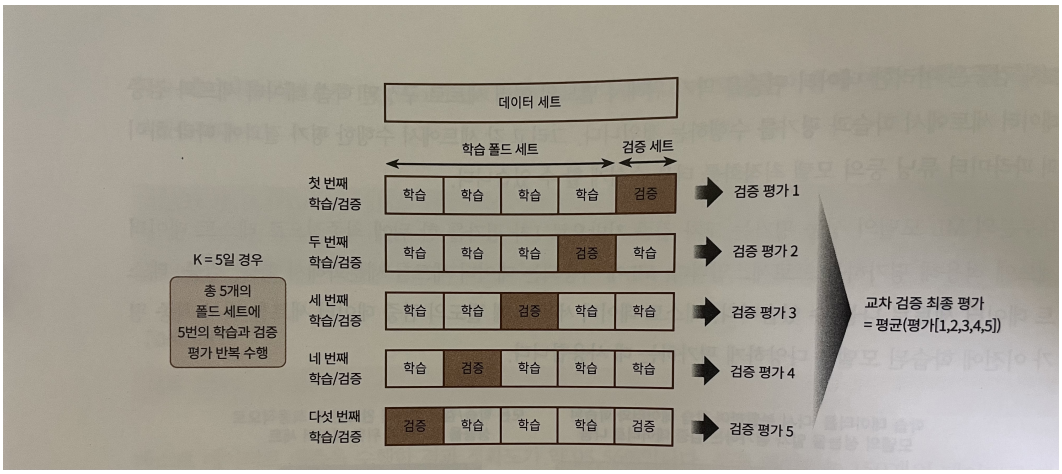
1) 데이터 세트를 K등분 함

2) 첫 번째 반복에서는 처음부터 4개의 등분을 학습 데이터 세트, 마지막 5번째 등분 하나를 검증 데이터 세트로 설정 → **학습 데이터 세트에서 학습 수행**, **검증 데이터 세트에서 평가 수행**

3) 첫 번째 수행 후 두번째 반복에서 다시 비슷한 학습과 평가 작업 수행함. 단, 학습 데이터와 검증데이터를 변경함. (4번째 등분 하나를 검증 데이터 세트로 결정)

4) 이렇게 학습 데이터와 검증 데이터 세트를 점진적으로 변경하면서 마지막 K번째까지 학습과 검증을 수행하는 것이 K 폴드 교차 검증임.

⇒ 5개(K개)의 예측 평가를 구하면, 이를 평균해서 K 폴드 평가 결과로 반영




```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
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 폴트 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
```

KFold 객체를 생성했으니, 이 생성된 객체의 `split()`을 호출해 전체 붓꽃 데이터를 5개의 폴드 데이터 세트로 분리함. 전체 붓꽃 데이터는 150개 → 학습용 데이터 세트는 4/5인 120개, 검증 테스트 데이터 세트는 1/5인 30개로 분할됨. `split()`을 호출하면, 학습용/검증용 데이터로 분할할 수 있는 인덱스를 반환함.

아래 코드는 5개의 폴드 세트를 생성하는 KFold 객체의 split()을 호출해 교차 검증 수행 시마다 학습과 검증을 반복해 예측 정확도를 측정함. 그리고 split()이 어떤 값을 실제로 반환하는 지도 확인하기 위해 검증 데이터 세트의 인덱스도 추출.

```
n_iter = 0

# KFold객체의 split( ) 호출하면 폴드 별 학습용, 검증용 테스트의 로우 인덱스를 array로 반환
for train_index, test_index  in kfold.split(features):
    # kfold.split( )으로 반환된 인덱스를 이용하여 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 시 마다 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(결정 클래스) 데이터 집합을 위한 K 폴드 방식.

(불균형한 분포도? 특정 레이블 값이 특이하게 많거나 매우 적어서 값의 분포가 한쪽으로 치우치는 것. e.g. 대출 사기 데이터 → 랜덤하게 학습 및 테스트 세트의 인덱스를 고르더라도 레이블 값의 비율을 적절하게 반영하지 못하는 경우가 쉽게 발생함. ⇒ 원본 데이터와 유사하게 레이블 값의 분포를 학습/테스트 세트에도 유지하는게 매우 중요함)

⇒ 이를 위해, **원본 데이터의 레이블 분포를 먼저 고려**한 뒤 이 **분포와 동일하게 학습과 검증 데이터 세트를 분배** 함.

StratifiedKFold: KFold로 분할된 레이블 데이터 세트가 전체 레이블 값의 분포도를 반영하지 못하는 문제 해결!

`split()` 인자 - 피처 데이터 세트 + 레이블 데이터 세트도 필요함! (레이블 데이터 분포도에 따라 학습/검증 데이터 나누기 때문)

```
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:
 2    34
0    33
1    33
```

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

→ 학습 레이블, 검증 레이블 데이터 값의 분포도 동일하게 할당됨

```
dt_clf = DecisionTreeClassifier(random_state=156)

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

# StratifiedKFold의 split( ) 호출시 반드시 레이블 데이터 셋도 추가 입력 필요
for train_index, test_index  in skfold.split(features, label):
    # split( )으로 반환된 인덱스를 이용하여 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 시 마다 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.mean(cv_accuracy))
'''
#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: 0.9666666666666667
'''
```

일반적으로 분류에서의 교차 검증은 K 폴드보다 Stratified K 폴드로 분할돼야 함. 회귀에서는 지원되지 않음.

교차검증을 보다 간편하게 - cross_val_score()

교차검증을 편리하게 수행할 수 있게 도와주는 API.

KFold에서 1) 폴드 세트르 설정 2) for 루프에서 반복으로 학습 및 테스트 데이터의 인덱스 추출 3) 반복적으로 학습과 예측 수행, 예측 성능 반환 → 이 과정을 한 번에 수행해줌.

`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 - 분류 또는 회귀 알고리즘(분류 알고리즘이 입력되면 Stratified K 폴드 방식으로 분할함)
- X - 피처 데이터 세트
- y - 레이블 데이터 세트
- scoring - 예측 성능 평가 지표

- cv - 교차 검증 폴드 수

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

iris_data = load_iris()
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))  # 교차 검증별 정확도: [0.98 0.94 0.98]
print('평균 검증 정확도:', np.round(np.mean(scores), 4))  # 평균 검증 정확도: 0.9667
```

`cross_val_score` 는 cv로 정해진 횟수만큼 scoring 파라미터로 지정된 평가 지표로 평가 결과값을 배열로 반환함. 그리고 이를 평균해 평가 수치로 사용함.

api 내부에서 estimator를 학습(fit), 예측(predict), 평가(evaluation) 시켜주어 간단하게 교차 검증 수행이 가능함. (내부적으로 Stratified KFold 이용)

비슷한 api로 `cross_validate()`가 있음. cross_val_score는 단 하나의 평가 지표만 사용 가능하지만, `cross_validate()`는 여러 개의 평가 지표 반환 가능, 성능 평가 지표와 수행 시간도 같이 제공함.

GridSearchCV - 교차 검증과 최적 하이퍼 파라미터 튜닝을 한 번에

하이퍼파라미터 값을 조정해 예측 성능 개선 가능함.

사이킷런은 GridSearchCV API를 이용해 분류나 회귀 알고리즘에 사용되는 최적의 파라미터 도출 방안을 제공함.

예를 들어, 결정 트리 알고리즘의 하이퍼 파라미터를 순차적으로 변경하면서 최고 성능을 가지는 파라미터 조합을 찾고자 한다면, 파라미터 집합을 만들고 이를 순차적으로 적용하면서 최적화 수행 가능함.

GridSearchCV는 교차 검증을 기반으로 하이퍼 파라미터의 최적 값을 찾게 해줌. 사용자가 튜닝하고자 하는 여러 종류의 하이퍼 파라미터를 다양하게 테스트하면서 최적의 파라미터를 편리하게 찾게 해주지만 동시에 순차적으로 파라미터를 테스트하므로 수행시간이 상대적으로 오래 걸림.

⇒ 즉, 데이터 세트를 cross-validation을 위한 학습/테스트 세트로 자동으로 분할한 뒤에 하이퍼 파라미터 그리드에 기술된 모든 파라미터를 순차적으로 적용해 최적의 파라미터 찾음.

- estimator - classifier, regressor, pipeline이 사용될 수 있음
- param_grid - key + 리스트 값을 가지는 딕셔너리가 주어짐. estimator 튜닝을 위해 파라미터명과 사용될 여러 파라미터 값을 지정함.
- scoring: 예측 성능을 측정할 평가 방법을 지정함. 보통은 사이킷런의 성능 평가 지표를 지정하는 문자열로 지정하거나 별도의 성능 평가 지표 함수 지정할 수 있음.
- cv - 교차검증을 위해 분할되는 학습/테스트 세트의 개수
- refit: 디폴트는 true, true로 생성시 가장 최적의 하이퍼 파라미터를 찾은 뒤 입력된 estimator 객체를 해당 하이퍼파라미터로 재학습시킴.

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

# 데이터 로딩 후 학습 데이터와 테스트 데이터 분리
iris = 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()

### parameter 들을 dictionary 형태로 설정
parametes = {'max_depth':[1,2,3], 'min_samples_split':[2,3]}
```

1) 학습 데이터 세트를 GridSearchCV 객체에 fit(학습데이터 세트) 메서드에 인자로 입력함.

2) GridSearchCV 객체의 fit(학습데이터 세트) 메서드를 수행하면 학습 데이터를 cv에 기술된 폴딩 세트로 분할해 param_grid에 기술된 하이퍼 파라미터를 순차적으로 변경하면서 학습/평가를 수행

3) 그 결과를 cv_results_ 속성에 기록

4) cv_results_는 GridSearchCV의 결과 세트로 딕셔너리 형태로 key값과 리스트 형태의 value 값을 가짐.

5) cv_ results_를 pandas의 df으로 변환하면 내용을 좀 더 쉽게 볼 수 있음.

```
import pandas as pd

# param_grid의 하이퍼 파라미터들을 3개의 train, test set fold로 나누어서 테스트 수행 설정
```

```
### refit=True가 디폴트. True면 가장 좋은 파라미터 설정으로 재학습시킴
grid_dtree = GridSearchCV(dtree, param_grid=parameters, cv=3, refit=True)

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

# GridSearchCV 결과 추출하여 DF으로 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}	0.966667	1	0.950	1.0	0.95
5	{'max_depth': 3, 'min_samples_split': 3}	0.966667	1	0.950	1.0	0.95

- params 칼럼에는 수행할 때마다 적용된 개별 하이퍼 파라미터값을 나타냄
- rank_test_score 하이퍼 파라미터별로 성능이 좋은 score 순위를 나타냄. 1이 가장 뛰어난 순위이며 최적의 하이퍼 파라미터
- mean_test_score는 개별 하이퍼 파라미터별로 CV의 폴딩 테스트 세트에 대해 총 수행한 평가 평균값.

`best_params_` 최고 성능을 나타낸 하이퍼파라미터 값

`best_socre_` 최고 성능을 나타낸 하이퍼파라미터의 평가 결과 값(cv_results_의 rank_test_score가 1일 때의 값)

```
print('GridSearchCV 최적 파라미터:', grid_dtree.best_params_)  # {'max_depth': 3, 'min_sampl
es_split': 2}
print('GridSearchCV 최고 정확도: {0:.4f}'.format(grid_dtree.best_score_))  # 0.9750
```

refit=True(default)면, GridSearchCV가 최적 성능을 나타내는 하이퍼 파라미터로 estimator를 학습해 best_estimator_로 저장함. 이미 학습된 best_estimator_를 이용해 앞에서 train_test_split()으로 분리한 테스트 데이터 세트에 대해 예측하고 성능을 평가해 보겠음.

```
# GridSearchCV의 refit으로 이미 학습이 된 estimator 반환
estimator = grid_dtree.best_estimator_

# GridSearchCV의 best_estimator_는 이미 최적 하이퍼 파라미터로 학습이 됨
pred = estimator.predict(X_test)
print('테스트 데이터 세트 정확도: {0:.4f}'.format(accuracy_score(y_test,pred)))
# 테스트 데이터 세트 정확도: 0.9667
```

<정리>

1. 데이터 로딩 후 학습/테스트 데이터 분리
2. parameter 설정
3. gridsearchcv 객체에 학습
4. 결과 최적의 하이퍼파라미터 구하고 -> 최적의 파라미터로 학습된 estimator 반환
5. 위의 estimator로 예측 및 성능 평가 진행

05. 데이터 전처리

- 결손값(NaN, Null 값) 허용되지 않음 → 대체 or 해당 피처 드롭 등..
- 문자열 값(카테고리 형 or 텍스트 형) → 숫자형 변환
  (사이킷런의 머신러닝 알고리즘은 문자열 값을 입력값으로 허용하지 않음)
  - 카테고리 형 → 코드 값으로 표현
  - 텍스트 형 → 피처 벡터화

데이터 인코딩(레이블 인코딩, 원-핫 인코딩)

레이블 인코딩(Label Encoding)

: 카테고리 피처를 코드형 숫자 값으로 변환 (간단하게 문자열 값을 숫자형 카테코리 값으로 변환)

`LabelEncoder` 를 객체로 생성한 후 `fit()`과 `transform()`을 호출해 레이블 인코딩 수행함.

```
from sklearn.preprocessing import LabelEncoder

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

```
# LabelEncoder를 객체로 생성한 후, fit()과 transform()으로 Label 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]
```

위와 달리, 데이터가 많은 경우에는 문자열이 어떤 숫자 값으로 인코딩 됐는지 알기 어려움. → LabelEncoder 객체 classes_ 속성값으로 확인해야 함.

```
print('인코딩 클래스:',encoder.classes_)  # 인코딩 클래스: ['TV' '냉장고' '믹서' '선풍기' '전자렌지'
 '컴퓨터']
```

`classes_` 속성은 0번부터 순선대로 변환된 인코딩 값에 대한 원본을 가짐.

`inverse_transform()` 을 통해 인코딩된 값을 다시 디코딩할 수 있음.

```
print('디코딩 원본 값:',encoder.inverse_transform([4, 5, 2, 0, 1, 1, 3, 3]))
# 디코딩 원본 값: ['전자렌지' '컴퓨터' '믹서' 'TV' '냉장고' '냉장고' '선풍기' '선풍기']
```

label encoding 문제) 일괄적인 숫자 값으로 변환이 되면서 몇몇 ml 알고리즘에 적용할 경우 예측 성능이 떨어지는 문제 발생. 숫자 값의 크고 작음에 대한 특성이 작용하기 때문... 더 큰 값에 가중치 부여되거나 더 중요하게 인식될 가능성 있음. → 레이블 인코딩은 선형 회귀와 같은 알고리즘에는 적용 x 트리 계열의 알고리즘은 숫자의 특성을 반영하지 않아 레이블 인코딩 사용 가능

⇒ 원-핫 인코딩이 이러한 문제점 해결해줌.

원-핫 인코딩(One-Hot Encoding) - get_dummies( )

: 피처 값의 유형에 따라 새로운 피처를 추가해 **고유 값에 해당하는 칼럼에만 1을 표시하고 나머지 칼럼에는 0을 표시**하는 방식. 즉, 행 형태로 돼 있는 피처의 고유 값을 열 형태로 차원을 변환 환 뒤, 고유 값에 해당하는 칼럼에만 1을 표시하고 나머지 칼럼에는 0을 표시함.

6개의 상품 분류 고유 값에 따라 상품 분류 피처를 6개의 상품 분류 고유 값 피처로 변환함. 해당 고유 값에 매칭되는 피처만 1을 표시하고, 나머지 피처는 0을 입력함. 여러 개의 속성 중 단 한 개의 속성만 1로 표시하여, 원-핫 인코딩이라고 함.

레이블 인코더와 달리 주의해야할 점

- OneHotEncoder로 변환하기 전에 모든 문자열 값이 숫자형 값으로 변환돼야 함

- 입력 값으로 2차원 데이터가 필요함

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

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

# 먼저 숫자값으로 변환을 위해 LabelEncoder로 변환
encoder = LabelEncoder()
encoder.fit(items)
labels = encoder.transform(items)
# 2차원 데이터로 변환
labels = labels.reshape(-1,1)

# 원-핫 인코딩 적용
oh_encoder = OneHotEncoder()
oh_encoder.fit(labels)
oh_labels = oh_encoder.transform(labels)
print('원-핫 인코딩 데이터')
print(oh_labels.toarray())
'''
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.]]
'''
print('원-핫 인코딩 데이터 차원')
print(oh_labels.shape)  # (8, 6)
```

원-핫 인코딩 더 쉽게 → `get_dummies()` 이용 ⇒ 숫자형 변환 없이 바로 사용 가능함

```
import pandas as pd

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

피처 스케일링과 정규화

서로 다른 **변수의 값 범위를 일정한 수준으로 맞추는 작업**을 피처 스케일링이라고 한다. 대표적인 방법으로는 **표준화(Standardization)와 정규화(Normalization)**가 있다.

- **표준화:** 데이터의 피처 각각이 **평균이 0이고 분산이 1인 가우시안 정규 분포**를 가진 값으로 변환되는 것. 표준화를 통해 변환될 피처 x의 새로운 i번째 데이터를 x_inew라고 한다면, 이 값은 원래 값에서 피처 x의 평균을 뺀 값을 피처 x의 표준편차로 나눈 값으로 계산할 수 있음.

  »

- **정규화:** 서로 다른 피처의 **크기를 통일하기 위해 크기를 변환**해주는 개념. 동일한 크기 단위로 비교하기 위해 값을 모두 최소 0 ~ 최대 1의 값으로 변환하는 것. 즉, **개별 데이터의 크기를 모두 똑같은 단위**로 변경하는 것. 새로운 데이터 x_inew는 원래 값에서 피처 x의 최댓값과 최솟값의 차이로 나누 값으로 변환될 수 있음. ⇒ **피처 스케일링**

  $$x_inew = x_i - min(x)/max(x) - min(x)$$

  그런데 사이킷런의 전처리에 제공되는 Normalizer 모듈과 일반적인 정규화는 약간의 차이가 있음. 사이킷런에서는 선형대수에서의 정규화 개념이 사용되었으며, 개별 벡터의 크기를 맞추기 위해 변환하는 것을 의미함. 즉, 개별 벡터를 모든 피처 벡터의 크기로 나눠줌. 세개의 피처가 있다고 하면, 원래 값에서 세개의 피처의 i번째 피처 값에 해당하는 크기를 합한 값으로 나눠 줌. ⇒ **벡터 정규화**

  $$x_inew = x_i/\sqrt{(x_i^2 + y_i^2 + z_i^2)}$$

StandardScalerd(표준화 지원 클래스)

표준화 지원 클래스. 즉, **개별 피처를 평균이 0이고, 분산이 1인 값으로 변환**해 줌. 가우시안 정규 분포를 가질 수 있도록 데이터 변환하는 것은 중요함. 특히 사이킷런에서 구현한 RBF 커널을 이용하는 서포트 벡터 머신, 선형 회귀, 로지스틱 회귀는 데이터가 가우시안 분포를 가지고 있다고 가정하고 구현되어, 사전에 표준화를 적용하는 것은 예측 성능 향상에 중요한 요소가 됨.

```
from sklearn.datasets import load_iris
import pandas as pd
# 붓꽃 데이터 셋을 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())
'''
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
'''
print('\nfeature 들의 분산 값')
print(iris_df.var())
'''
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
'''
```

StandardScaler를 이용해 각 피처 표준화. StandardScaler 객체 생성 → fit, transform(스케일 변환된 데이터 세트가 넘파이의 ndarray이므로 DataFrame으로 변환해 평균값과 분산 확인)

```
from sklearn.preprocessing import StandardScaler

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

#transform( )시 scale 변환된 데이터 셋이 numpy ndarry로 반환되어 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())
'''
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
'''
print('\nfeature 들의 분산 값')
print(iris_df_scaled.var())
'''
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
'''
```

→ 모든 칼럼 값의 평균 0에 아주 가깝게, 분산은 1에 아주 가까운 값으로 변환됨.

MinMaxScaler

: 데이터값을 0과 1사이의 범위 값으로 변환. (데이터의 분포가 가우시안 분포가 아닐 경우 사용)

```
from sklearn.preprocessing import MinMaxScaler

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

# transform()시 scale 변환된 데이터 셋이 numpy ndarry로 반환되어 이를 DataFrame으로 변환
iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_names)
print('feature들의 최소 값')
print(iris_df_scaled.min())
'''
feature들의 최소 값
sepal length (cm)    0.0
sepal width (cm)     0.0
petal length (cm)    0.0
petal width (cm)     0.0
dtype: float64
'''
print('\nfeature들의 최대 값')
print(iris_df_scaled.max())
'''
feature들의 최대 값
sepal length (cm)    1.0
sepal width (cm)     1.0
petal length (cm)    1.0
petal width (cm)     1.0
dtype: float64
'''
```

→ 모든 피처에 0에서 1 사이의 값으로 변환되는 스케일링이 적용됨

학습 데이터와 테스트 데이터의 스케일링 변환 시 유의점

StandardScaler나 MinMaxScaler와 같은 Scaler 객체를 이용해 데이터의 스케일링 변환 시 fit(), transform(), fit_transform() 메서드를 이용함.

- `fit()` 데이터 변환을 위한 기준 정보 설정(예를 들어, 데이터 세트의 최댓값/최솟값 설정 등)
- `transform()` 이렇게 설정된 정보를 이용해 데이터 변환
- `fit_transform()` 이 둘을 한번에 적용하는 기능 수행

학습 데이터와 테스트 데이터에 fit()과 transform() 적용할 때 주의가 필요함. scaler 객체를 이용해 학습 데이터로 위 메서드를 적용하면 테스트 데이터 세트로는 다시 fit()을 수행하지 않고 학습 데이터 세트로 fit()을 수행한 결과를 이용해 transform() 변환을 적용해야 함.
⇒ 즉, 학습 데이터로 `fit()`이 적용된 스케일링 기준 정보를 그대로 테스트 데이터에 적용해야 함! (그렇지 않고 테스트 데이터로 다시 새로운 스케일링 기준 정보를 만들게 되면 학습 데이터와 테스트 데이터의 스케일링 기준 정보가 서로 달라짐)

↓ 학습 데이터로 fit( ) 수행한 scaler 객체의 transform( ) 이용해 데이터 변환

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

# 학습 데이터는 0 부터 10까지, 테스트 데이터는 0 부터 5까지 값을 가지는 데이터 세트로 생성
# Scaler클래스의 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)
```

```
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))
'''
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. ]
'''

# test_array에 Scale 변환을 할 때는 반드시 fit() 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 test_array 데이터: [0 1 2 3 4 5]
Scale된 test_array 데이터: [0.  0.1 0.2 0.3 0.4 0.5]
'''
```

마찬가지로 fit_transform()도 사용을 주의해야 함. 테스트 데이터에서는 사용하면 안됨.

⇒ 그냥 학습/테스트 데이터로 **분리하기 전에!** 전체 데이터 세트에 스케일링 적용한 뒤 학습/테스트 데이터로 분리하는 것이 더 바람직함.

> 💡 1. 가능하다면 **전체 데이터의 스케일링 변환을 적용한 뒤** 학습과 테스트 데이터로 분리
> 2. 1이 여의치 않다면 테스트 데이터 변환 시에는 fit()이나 fit_transform()을 적용하지 않고, 학습 데이터로 이미 fit() 된 scaler 객체를 이용해 transform()으로 변환

6. 사이킷런으로 수행하는 타이타닉 생존자 예측

타이타닉 탑승자 데이터를 기반으로 생존자 예측 진행.

변수명	설명
Passengerid	탑승자 데이터 일련변호
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

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

```
titanic_df = pd.read_csv('titanic_train.csv')
titanic_df.head(3)
```

```
# 데이터 칼럼 타입 확인
titanic_df.info()
'''
<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
'''
```

- RangeIndex: DataFrame 인덱스의 범위를 나타냄 → 전체 로우 수 알 수 있음
- Data columns: 전체 칼럼 수
- object 타입의 칼럼은 string 타입으로 봐도 무방함

사이킷런 머신러닝 알고리즘은 Null 값을 허용하지 않아서 이 값들을 어떻게 처리할지 결정해야 함.

→ 여기서는 `fillna()` 함수를 사용해서 간단하게 Null 값을 평균 또는 고정 값으로 변경하겠음.

- Age → 평균 나이
- 나머지 칼럼 → 'N' 값으로 변경

```
titanic_df['Age'].fillna(titanic_df['Age'].mean(), inplace=True)
titanic_df['Cabin'].fillna('N', inplace=True)
titanic_df['Embarked'].fillna('N',inplace=True)
```

모든 칼럼의 Null 값 없는지 확인

```
print('데이터 세트 Null 값 갯수 ', titanic_df.isnull().sum().sum())  # 0

# df.isnull().sum() -> 칼럼 별 null 값의 개수
# titanic_df.isnull().sum().sum() -> 총 null 값의 개수
```

문자열 피처인 Sex, Cabin, Embarked 값 분류를 살펴보겠음.

```
# sex 값 분포
titanic_df['Sex'].value_counts()
'''
male      577
female    314
Name: Sex, dtype: int64
'''
# Cabin 값 분포
'''
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: Cabin, Length: 148, dtype: int64
'''
# Embarked 값 분포
'''
S    644
C    168
Q     77
N      2
Name: Embarked, dtype: int64
'''
```

→ Cabin의 경우 속성값이 제대로 정리되지 않은 것 같음. Cabin의 경우, 선실 번호 중 선실 등급 나타내는 첫번째 알파벳이 중요해 보임 → 앞 문자만 추출

```
titanic_df['Cabin'] = titanic_df['Cabin'].str[:1]
titanic_df['Cabin'].head(3)
'''
```

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

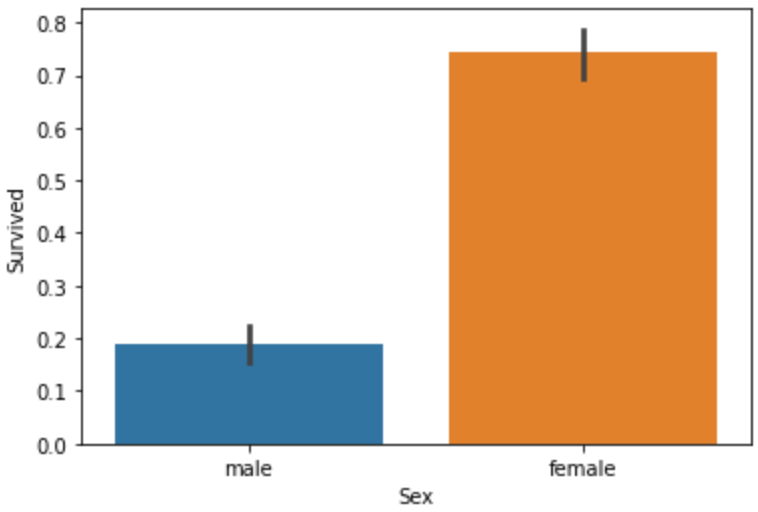
머신러닝 알고리즘으로 예측을 수행하기 전에 데이터 탐색을 먼저 하겠음.

1) 어떤 유형의 승객이 생존 확률이 높았는지 확인 - **성별**에 따른 생존자 수

```
titanic.groupby(['Sex', 'Survived'])['Survived'].count()
'''
Sex     Survived
female  0            81
        1           233
male    0           468
        1           109
Name: Survived, dtype: int64
'''
```

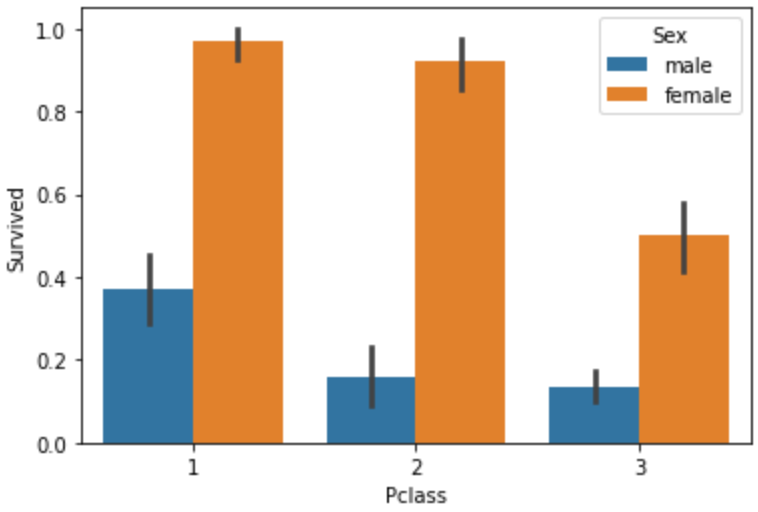
탑승객은 남자 557명, 여자 314명으로 남자가 더 많음. 시본 패키지를 이용해서 시각화해보겠음. 시본은 기본적으로 맷플롯립에 기반하고 있지만, 좀 더 세련된 비주얼과 쉬운 API, 편리한 판다스 df과의 연동 등으로 데이터 분석을 위한 시각화로 애용되는 패키지

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



2) 어떤 유형의 승객이 생존 확률이 높았는지 확인 - **부**에 따른 생존자 수(부 측정 - 객실 등급으로 확인 가능함) + 성별도 함께 고려하겠음

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



https://hleecaster.com/python-seaborn-barplot/

estimation, hue에 대한 설명 등

((**여러 열에서 집단 묶어서 세부 집단 시각화 하기 (hue)** → 예를 들어 “Gender”별로 차트를 그릴 때 그 안에 “Age Range”를 각각 보고 싶다고 하면 `hue`를 쓰면 된다.))

3) 어떤 유형의 승객이 생존 확률이 높았는지 확인 - **나이**에 따른 생존자 수

Age 값의 종류가 많기 때문에 범위별로 분류해 카테고리 값을 할당하겠음.

나이	값
0~5	Baby
6~12	Child
13~18	Teenager
19~25	Student
26~35	Yound Adult

36~60	Adult
61~	Elderly
-1 이하 오류값	Unknown

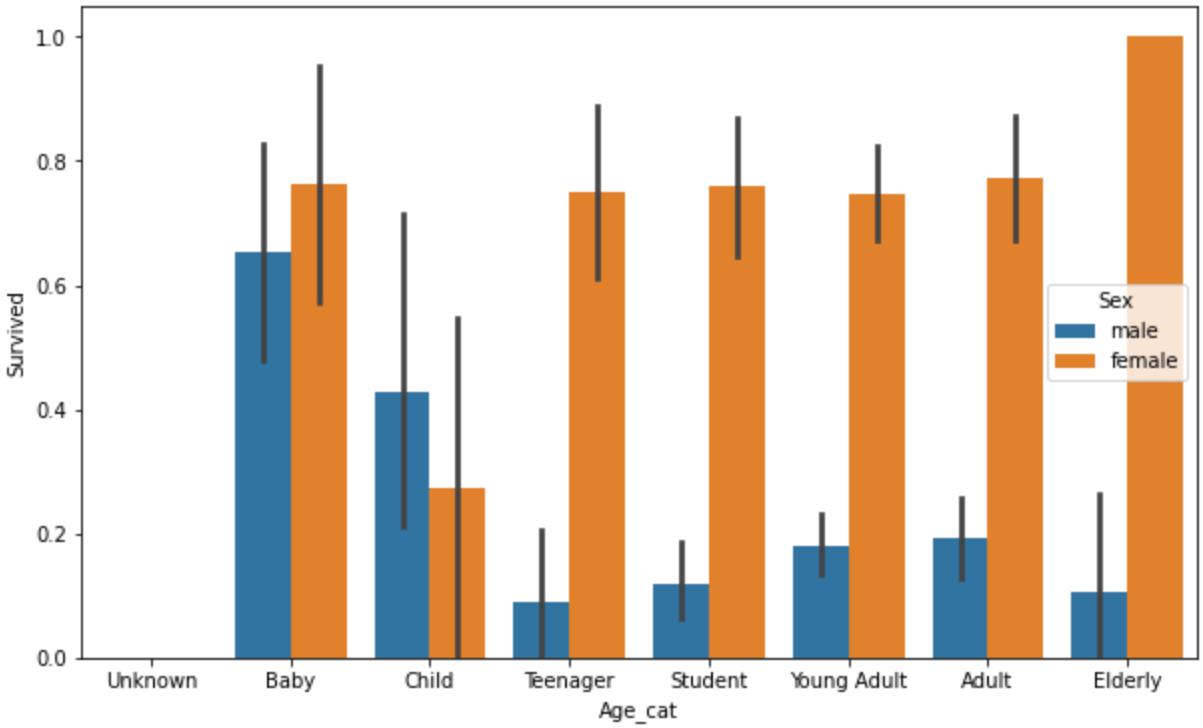
```
# 입력 age에 따라 구분값을 반환하는 함수 설정. DataFrame의 apply lambda식에 사용.
def get_category(age):
    cat = ''
    if age <= -1: cat = 'Unknown'
    elif age <= 5: cat = 'Baby'
    elif age <= 12: cat = 'Child'
    elif age <= 18: cat = 'Teenager'
    elif age <= 25: cat = 'Student'
    elif age <= 35: cat = 'Young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

    return cat

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

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

# lambda 식에 위에서 생성한 get_category( ) 함수를 반환값으로 지정.
# get_category(X)는 입력값으로 'Age' 컬럼값을 받아서 해당하는 cat 반환
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)
```



→ 분석 결과, Sex, Age, PClass가 생존을 좌우하는 중요 피처임을 어느 정도 확인함.

이제 남아있는 문자열 카테고리 피처를 숫자형 카테고리 피처로 변환하겠음. 인코딩은 사이킷런의 LabelEncoder 클래스를 이용해 레이블 인코딩 적용하겠음.

LabelEncoder 객체는 카테고리 값의 유형 수에 따라 0~(카테고리 유형 수 -1)까지의 숫자 값으로 변환함.

~~여기서부터 def 함수 인수로 쓰이는 dataDF나 df는 어디서 나온건지 궁금함 아니 이해가 안됨~~

```
from sklearn import preprocessing

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

    return dataDF

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

지금까지 피처를 가공한 내역을 정리하고 이를 함수로 만들어 쉽게 재사용할 수 있도록 하겠음.
(

데이터의 전처리를 전체적으로 호출하는 함수는 transform_features()이며 Null 처리, 포매팅, 인코딩을 수행하는 내부 함수로 구성함.

```
from sklearn.preprocessing import LabelEncoder

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

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

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

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

이 함수 이용해 다시 원본 데이터를 가공해 보겠음. 원본 CSV 파일 다시 로딩하고,
생존자 데이터 세트의 레이블인 **Survived 속성만 별도 분리해 클래스 결정값(label/target)** 데이터 세트로 만들겠음. 그리고 **Survived 속성을 드롭해 피처 데이터 세트(feature)**를 만들겠음. 이렇게 생성된 피처 데이터 세트에 transform_features()를 적용해 데이터를 가공해보겠음.

```
# 원본 데이터 재로딩하고, feature 데이터 셋과 label 데이터 셋 추출
titanic_df = pd.read_csv('titanic_train.csv')
y_titanic_df = titanic_df['Survived']  # 레이블 데이터 셋
X_titanic_df = titanic_df.drop('Survived', axis=1)  # 피처 데이터 셋
```

```
X_titanic_df = transform_features(X_titanic_df)
```

테스트 데이터 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)
```

ML 알고리즘인 결정 트리, 랜덤 포레스트, 로지스틱 회귀를 이용해 타이타닉 생존자를 예측하겠음.

위의 코드에서 학습/테스트 세트로 분리된 데이터를 기반으로, 머신러닝 모델을 학습하고(fit), 예측(predict)할 것. 예측 성능 평가는 정확도로 할 것.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# 결정트리, Random Forest, 로지스틱 회귀를 위한 사이킷런 Classifier 클래스 생성
dt_clf = DecisionTreeClassifier(random_state=11)
rf_clf = RandomForestClassifier(random_state=11)
lr_clf = LogisticRegression()

# 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)))  #
 0.7877

# 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)))
# 0.8547

# 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)))
# 0.8492
```

→ 아직 최적화 작업 수행하지 않았고, 데이터 양 충분하지 않아 어떤 알고리즘이 가장 높은 성능 가지는지 평가할 수 없음.

교차 검증으로 결정 트리 모델을 좀 더 평가해 보겠음.

1) KFold

```
from sklearn.model_selection import KFold

def exec_kfold(clf, folds=5):
    # 폴드 세트를 5개인 KFold객체를 생성, 폴드 수만큼 예측결과 저장을 위한  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
        X_train, X_test = X_titanic_df.values[train_index], X_titanic_df.values[test_inde
x]
        y_train, y_test = y_titanic_df.values[train_index], y_titanic_df.values[test_inde
x]

        # 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() → stratifiedkfold 이용해 폴드 분할

```
from sklearn.model_selection import cross_val_score

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 → 최적 하이퍼파라미터 찾기 → 예측 성능 측정

```
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_spli
t': 2}
GridSearchCV 최고 정확도: 0.7992
테스트 세트에서의 DecisionTreeClassifier 정확도 : 0.8715
'''
```

07. 정리

지금까지 사이킷런을 기반으로 머신러닝 애플리케이션을 쉽게 구현해 보았음. 머신러닝 애플리케이션의 수행 과정은 아래와 같음.

- 데이터의 가공 및 변환 과정의 **전처리 작업**
  - 오류 데이터 보정, 결손값 처리 → 데이터 클렌징 작업
  - 레이블 인코딩, 원-핫 인코딩 → 인코딩 작업

    - 데이터의 스케일링/정규화

  ⇒ 머신러닝 알고리즘이 최적으로 수행될 수 있도록 데이터 사전 처리

- 테이터를 학습 데이터와 테스트 데이터로 분리하는 **데이터 세트 분리** 작업
- 학습 데이터를 기반으로 **머신러닝 알고리즘을 적용**해 모델 학습
- 학습된 모델을 기반으로 **테스트 데이터에 대한 예측** 수행
- **예측된 결괏값을 실제 결괏값과 비교**해 모델에 대한 **평가 수행**