02 사이킷런으로 시작하는 머신러닝

01 사이킷런 소개와 특징

사이킷런의 소개

- 파이썬 기반의 머신러닝을 위한 가장 쉽고 효율적인 개발 라이브러리

사이킷런의 특징

- 쉽고 가장 파이썬스러운 API 제공
- 머신러닝을 위한 매우 다양한 알고리즘과 개발을 위한 편리한 프레임워크와 API 제공
- 오랜시간, 매우 많은 환경에서 사용되는 성숙한 라이브러리

02 첫 번째 머신러닝 만들어 보기 - 붓꽃 품종 예측하기

지도학습(Supervised Learning)

- 명확한 정답이 주어진 데이터를 먼저 학습한 뒤 미지의 정답을 예측하는 방식
- 학습 데이터 세트 : 학습을 위해 주어진 데이터 세트
- 테스트 데이터 세트 : 머신러닝 모델의 예측 성능을 평가하기 위해 별도로 주어진 데이터 세트

붓꽃 데이터 생성 및 ML알고리즘 DecisionTreeClassifier 적용

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

붓꽃 데이터 세트 로딩 및 DataFrame 변환

```
import pandas as pd

#붓꽃 데이터 세트 로딩
iris = load_iris()

#iris.data는 데이터 세트에서 feature만으로 된 데이터를 numpy형태로 보유
iris_data = iris.data

#iris.target은 데이터 세트에서 label(결정 값) 데이터를 numpy로 보유
iris_label = iris.target
print('iris target 값:',iris_label)
print('iris tagert 명:',iris.target_names)

#붓꽃 데이터 세트를 DataFrame으로 변환
iris_df = pd.DataFrame(data=iris_data, columns = iris.feature_names)
iris_df['label'] = iris.target
iris_df.head(3)
```

train_test_split()를 이용해 학습용 데이터와 테스트용 데이터를 분리

```
#iris_data : feature data set
#iris_label : label data set
#test_size : test data set의 비율
#random_state : random 값을 고정시키기 위해서 존재
X_train, X_test, y_train, y_test = train_test_split(iris_data, iris_label, test_size = 0.2, random_state = 11)
```

의사 결정 트리를 이용한 학습과 예측 수행

```
#DecisionTreeClassifier 객체 생성
dt_clf = DecisionTreeClassifier(random_state = 11)
#학습 수행
dt_clf.fit(X_train,y_train)
#학습이 완료된 DecisionTreeClassifier 객체에서 테스트 데이터 세트로 예측 수행.
pred = dt_clf.predict(X_test)
#DecisionTreeClassifier의 예측 성능 평가
from sklearn.metrics import accuracy_score
print('예측 정확도: {0:.4f}'.format(accuracy_score(y_test,pred))
```

붓꽃 데이터 세트로 분류를 예측한 프로세스를 정리

1. **데이터 세트 분리** : 데이터를 학습 데이터와 테스트 데이터로 분리
2. **모델 학습** : 학습 데이터를 기반으로 ML 알고리즘(의사 결정 트리)를 적용해 모델 학습
3. **예측 수행** : 학습된 ML 모델을 이용해 테스트 데이터의 분류(즉, 붓꽃 종류)를 예측
4. **평가** : 예측된 결괏값과 테스트 데이터의 실제 결괏값을 비교해 ML 모델 성능 평가

03 사이킷런의 기반 프레임워크 익히기

Estimator 이해 및 fit(), predict() 메서드

Estimator 클래스

- **Classifier**(분류 알고리즘을 구현한 클래스) + **Regressor**(회귀 알고리즘을 구현한 클래스)
- **fit()** (ML 모델 학습), **predict()** (학습된 모델의 예측) 을 내부에서 구현함

분류나 회귀를 위한 데이터 세트의 keys

- data는 피처의 데이터 세트를 가리킨다
- target은 분류 시 레이블값, 회귀 일 때는 숫자 결괏값 데이터 세트이다
- target_names는 개별 레이블의 이름을 나타냅니다
- feature_names는 피처의 이름을 나타낸다
- DESCR은 데이터 세트에 대한 설명과 각 피처의 설명을 나타낸다.

붓꽃 데이터 세트 생성

```
from sklearn.datasets import load_iris

iris_data = load_iris()
print(type(iris_data)) # Bunch 클래스(Dict 자료형과 유사함)

keys = iris_data.keys()
print(keys)
#dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names', 'filename'])
```

load_iris()가 반환하는 객체의 키인 feature_names, target_name, data,target이 가리키는 값

Input

```
print(type(iris_data.feature_names))
print(len(iris_data.feature_names))
print(iris_data.feature_names)

print(type(iris_data.target_names))
print(len(iris_data.target_names))
print(iris_data.target_names)

print(type(iris_data.data))
```

```
print(iris_data.data.shape)
print(iris_data.data)

print(type(iris_data.target))
print(iris_data.target.shape)
print(iris_data.target)
```

Output

```
 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]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]



 [6.3 2.5 5.  1.9]
 [6.5 3.  5.2 2. ]
 [6.2 3.4 5.4 2.3]
 [5.9 3.  5.1 1.8]]

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

04 Model Selection 모듈 소개

학습/테스트 데이터 세트 분리 - train_test_split()

학습 데이터로만 학습하고 예측했을 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)

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

```
예측 정확도:  1.0
```

정확도가 100%인 이유

이미 학습한 학습 데이터 세트를 기반으로 예측했기 때문

따라서 예측을 수행하는 데이터 세트는 **전용의 테스트 데이터 세트**여야함

train_test_split()을 통해 원본 데이터 세트에서 학습 및 테스트 데이터 세트를 쉽게 분리할 수 있음

train_test_split()을 이용한 붓꽃 데이터 세트 분리

- **test_size** : 전체 데이터에서 **테스트** 데이터 세트 크기를 얼마로 샘플링 할 것인가를 결정함. defalut 값은 25%이다.
- **train_size** : 전체 데이터에서 **학습용** 데이터 세트 크기를 얼마로 샘플링할 것인가를 결정함. 잘 사용되지는 않는다.
- **shuffle :** 데이터를 분리하기 전에 데이터를 미리 섞을지를 결정한다. default는 True이다. 데이터를 분산시켜서 좀 더 효율적인 학습 및 테스트 데이터 세트를 만드는데 사용됩니다.
- **random_state :** 호출할 떄마다 동일한 학습/테스트용 데이터 세트를 생성하기 위해 주어지는 난수 값. train_test_split()은 호출 시 무작위로 데이터를 분리하므로 random_state를 지정하지 않으면 수행할 때마다 다른 학습/테스트 용 데이터를 생성합니다. 책에서 소개하는 예제는 ranom_state를 동일한 데이터 세트로 분리하기 위해 random_state를 일정한 숫자 값으로 부여한다.
- train_test_split()의 반환값은 튜플 형태이다. 순차적으로 학습용 데이터의 피처, 테스트용 데이터의 피처, 학습용 데이터의 레이블, 테스트용 데이터의 레이블 데이터 세트가 반환된다.

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

dt_clf = DecisionTreeClassifier()
iris_data = load_iris()

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

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

예측 정확도 :0.9556

1. 붓꽃 데이터, 의사결정 트리 호출
2. 학습 데이터와 테스트 데이터 분리
3. 의사결정트리에 학습 데이터 모델 학습
4. 테스트 데이터의 피처 데이터를 이용해 테스트 데이터의 레이블값 예측
5. 테스트 데이터의 레이블과 피처데이터 비교를 통한 예측 정확도 측정

교차 검증

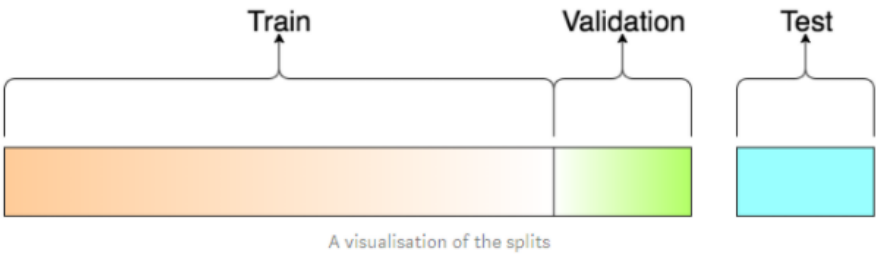
과적합(Overfitting)

모델이 학습 데이터에만 과도하게 최적화되어, 실제 예측을 다른 데이터로 수행할 경우에는 예측 성능이 과도하게 떨어지는 것.

해당 테스트 데이터에만 과적합되는 학습 모델이 만들어져 다른 테스트용 데이터가 들어올 경우에는 성능이 저하된다.

→ 따라서 **교차 검증**을 이용해 더 다양한 학습, 평가를 수행

여러 세트로 구성된 학습 데이터 세트와 검증 데이터 세트에서 학습과 평가를 수행



A visualisation of the splits

K 폴드 교차 검증

K개의 데이터 폴드 세트를 만들어서 K번만큼 각 폴드 세트에 학습과 검증 평가를 반복적으로 수행

K폴드 교차 검증 프로세스 구현

```
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 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

```
import numpy as np
n_iter = 0

#Kfold 객체의 split()를 호출하면 폴드 별 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
    #반복시마다 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 iteration 별 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]

## 평균 검증 정확도: 1.0

#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]

## 평균 검증 정확도: 0.98335

#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]

## 평균 검증 정확도: 0.9444666666666667

#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]

## 평균 검증 정확도: 0.941675

#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 분포도를 가진 레이블 데이터 집합을 위한 K 폴드 방식

대출 사기 데이터 예측

대출 사기 레이블 : 전체의 0.0001% → K 폴드로 제대로 된 비율 반영 X

K 폴드의 문제점

```
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()
```

```
0    50
1    50
2    50
Name: label, dtype: int64
```

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

교차 검증시 학습 레이블이 1,2 밖에 없는 경우에는 0의 경우 전혀 학습하지 못함

반대로 검증 레이블은 0 밖에 없으므로 학습 모델은 절대 0을 예측하지 못함

StratifiedKFold

레이블 데이터에 따라 학습/검증 데이터를 나누기 때문에, split() 메서드에 피처 뿐만 아니라 **레이블 데이터도 반드시 필요함**

```
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']):
    print(train_index, test_index)
    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
```

- **여기서 한가지 의문이 들었던건, iris_df에서 밑의 DataFrame처럼 label을 포함하고 있는데, 왜 이를 피처 데이터 세트라고 설명할까 의문이 들었다.**

	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
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	2
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2
149	5.9	3.0	5.1	1.8	2

그러나 피처 데이터 세트일 수 밖에 없다. 다시 말하면 label 데이터 세트는 의미가 없다! label 데이터는 0, 1, 2로 나눠지며, 각각의 값들이 0 50개, 1 50개, 2 50개로 나눠지는 label data set를 의미한다. 이 때, feature data set는 0~149까지의 3으로 split했을 때 0~49 50~99 100 149로 나뉘는 반면 lable data set은 각 value에 따라 나누어진다. 따라서 0~16, 50~66, 100~115 // 17~33, 67~82, 116~132 // 34~49, 83~99, 133~149로 test_index가 설정되는 것이다.

- 그럼 여기서 또 의문점, 왜 split함수는 label 데이터 세트를 저렇게 분할하는 것일까?

StratifiedKFold를 이용한 붓꽃 데이터 분리

```
# 붓꽃 데이터 세트에서 Stratified Kfold 를 이용한 검증
dt_clf = DecisionTreeClassifier(random_state=156)

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

# StratifiedKFold의 split 호출 시 반드시 레이블 데이터 세트도 추가 입력 필요
for train_index, test_index in skfold.split(features, label):
    #split()으로 반환된 인덱스를 이용해 학습용, 검증용 테스트 데이터 추출
    X_train, X_test = features[train_index], features[test_index]
    y_train, y_test = label[train_index], label[test_index]
    # 학습 및 예측
    dt_clf.fit(X_train, y_train)
    pred = dt_clf.predict(X_test)
    # 반복 시마다 정확도 측정
    n_iter += 1
    accuracy = np.round(accuracy_score(y_test, pred),4)
    train_size = X_train.shape[0]
    test_size = X_test.shape[0]
    print('{0} 교차검증 정확도: {1}, 학습데이터 크기: {2}, 검증데이터 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)

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

```
1 교차검증 정확도: 0.98, 학습데이터 크기: 100, 검증데이터 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 교차검증 정확도: 0.94, 학습데이터 크기: 100, 검증데이터 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 교차검증 정확도: 0.98, 학습데이터 크기: 100, 검증데이터 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]

 교차검증별 정확도:  [0.98 0.94 0.98]
## 평균 검증 정확도:  0.9666666666666667
```

왜곡된 레이블 데이터 세트에서는 반드시 Stratified K 폴드를 이용해 교차 검증해야함

교차 검증을 보다 간편하게 - cross_val_score()

KFold로 데이터를 학습하고 예측하는 코드

1. 폴드 세트 설정
2. for 루프에서 반복으로 학습 및 테스트 데이터의 인덱스를 추출
3. 반복적으로 학습과 예측을 수행하고 예측 성능 반환

cross_val_score() API 선언 형태

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

- estimator : 예측모델
- X : 피처 데이터 세트
- y : 레이블 데이터 세트,
- scoring : 예측성능평가 지표
- cv : 교차검증 폴드 수 (estimator로 classifier가 입력되면 Stratified Kfold 방식으로 학습/테스트 데이터 세트 분할)

- 수행 후 scoring 파라미터 값으로 지정된 성능 지표를 배열 형태로 반환

cross_val_score()의 자세한 사용법

```
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

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

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

GridSearchCV - 교차 검증과 최적 하이퍼 파라미터 튜닝을 한 번에

GridSearchCV

촘촘하게 파라미터를 입력하면서 테스트 하는 방식

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

GridSearch 클래스의 생성자로 들어가는 주요 파라미터

- estimator : 적용 알고리즘 모델로 classifier, regressor, pipeline 등이 사용
- param_grid : ket+리스트 값을 가지는 딕셔너리가 주어짐. estimator의 튜닝을 위해 파라미터명과 사용될 여러 파라미터 값을 지정
- scoring : 예측 성능을 평가할 평가 방법을 지정, 문자열로 사이킷런의 성능평가 지표를 입력하나 별도의 함수 지정도 가능
- cv : 교차 검증을 위해 분할되는 학습/테스트 세트의 개수를 지정
- refit : 기본값은 True이며 True로 생성시 가장 최적의 하이퍼 파라미터를 찾은 뒤 입력된 estimator 객체를 해당 하이퍼 파리미터로 재학습시킴

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
# 데이터를 로딩하고 학습데이터와 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()

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

```
import pandas as pd
# param_grid의  하이퍼 파라미터를 3개의 train, test set fold로 나누어 테스트 수행 설정
## refit=True가 기본 설정으로 True이면 가장 좋은 파라미터 설정으로 재학습 시킴
```

```
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 데이터 프레임으로 반환
# cv_results_는 gridsearchcv의 결과 세트로서 딕셔너리 형태로 key와 리스트 형태의 value 값을 가진다.
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 칼럼에는 수행할 때 마다 적용된 개별 하이퍼 파라미터값을 나타냅니다.
- rank_test_score는 하이퍼 파라미터별로 성능이 좋은 score 순위를 나타냄.
- mean_test_score는 개별 하이퍼 파라미터별로 CV의 폴딩 테스트 세트에 대해 총 수행한 평균값

GridSearchCV 객체의 fit()을 수행하면 최고 성능을 나타낸 하이퍼 파라미터의 값과 평가 결과 값이 각각 best_params, best_score_ 속성에 기록된다.

```
print('GridSearch 최적 파라미터: ', grid_dtree.best_params_)
print('GridSearch 최고 점수: ', grid_dtree.best_score_)
```

```
GridSearch 최적 파라미터:  {'max_depth': 3, 'min_samples_split': 2}
GridSearch 최고 점수:  0.975
```

- **이 때 파라미터 선정 기준은 무엇이 되는것인가?**

refit = True이면 GridSearchCV가 최적 성능을 나타내는 하이퍼 파라미터로 Estimator를 학습해 best_estimator로 저장함

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

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

```
테스트 데이터세트 정확도:  0.9667
```

05 데이터 전처리

데이터 인코딩

레이블 인코딩 : 카테고리 피처를 코드형 숫자 값으로 변환하는 것

LabelEncoder(클래스)를 객체로 생성한 후 fit()과 transform()을 호출해 레이블 인코딩을 수행함

```
from sklearn.preprocessing import LabelEncoder

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

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

```
인코딩 변화값: [0 1 4 5 3 3 2 2]
```

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

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

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

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

레이블 인코딩이 일괄적인 숫자 값으로 변환이 되면서 **크고 작음에 대한 특성**이 작용할 수 있음

따라서, 레이블 인코딩은 선형회귀와 같은 ML알고리즘에는 적용하지 않아야 함

원-핫 인코딩 : 피처 값의 유형에 따라 새로운 피처를 추가해 고유 값에 해당하는 칼럼에만 1을 표시하고 나머지 칼럼에는 0을 표시하는 방식

행 형태로 되어 있는 피처의 고유값을 열 형태로 변환한 뒤, 고유값에 해당하는 칼럼에만 1, 나머지는 0을 표시

⭐ **주의할 점**

1. OneHotEncoder로 변환하기 전에 모든 문자열 값이 **숫자형 값**으로 변환돼야 한다는 것
2. 입력 값으로 **2차원 데이터**가 필요하다는 점

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

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

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

#원-핫 인코딩을 적용한다.
oh_encoder = OneHotEncoder()
oh_encoder.fit(labels)
oh_labels = oh_encoder.transform(labels)
print('원-핫 인코딩 데이터')
print(oh_labels.toarray())
print('\n원-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)
```

위 예제 코드의 변환 절차

원본 데이터 → 숫자로 인코딩(2차원 데이터로 변환) → 원-핫 인코딩

판다스의 get_dummies() : 문자열 카테고리를 숫자 형으로 변환할 필요 없이 바로 변환 가능

```
import pandas as pd

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

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

피처 스케일링과 정규화

피처 스케일링(feature scaling)

서로 다른 변수의 값 범위를 일정한 수준으로 맞추는 작업

대표적으로 **표준화(Standardization)**, **정규화(Normalization)**이 있다.

표준화(Standardization)

데이터의 피처 각각이 평균이 0이고 분산이 1인 가우시안 정규 분포를 가진 값으로 변환하는 것

$$x_{i\_}\,new = \frac{x_i - mean(x)}{stdev(x)}$$

정규화(Normalization)

서로 다른 피처의 크기를 통일하기 위해 크기를 변환해주는 개념

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

그런데 **사이킷런의 Normalizer 모듈은 개별 벡터의 크기를 맞추기 위해 변환**하는 것을 의미함

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

StandardScaler

```
from sklearn.datasets import load_iris
import pandas as pd
#붓꽃 데이터 세트를 로딩하고 DataFrame으로 변환합니다
iris = load_iris()
iris_data = iris.data
iris_df = pd.DataFrame(data=iris_data,columns = iris.feature_names)

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

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

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

StandardScaler를 이용해 각 피처를 한 번에 표준화해 변환

이때, **transform()을 호출**할 때 **스케일 변환된 데이터 세트가 넘파이의 ndarray**이므로 이를 DataFrame으로 변환한다.

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

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

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

모든 칼럼의 값의 **평균이 0에 아주 가까운 값**으로, **분산은 1에 아주 가까운 값**으로 변환됐다!

MinMaxScaler

데이터값을 0과 1사이의 범위 값으로 변환함

```
from sklearn.preprocessing import MinMaxScaler

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

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

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

feature 들의 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

fit()은 데이터 변환을 위한 기준 정보 설정(데이터 세트의 최소/최댓값 설정)

transform()은 이렇게 설정된 정보를 이용해 데이터를 변환

fit_transform()은 한번에 적용하는 기능 수행

그런데 학습 데이터로 fit()이 적용된 스케일링 기준 정보를 그대로 테스트 데이터에 적용해야 하며, 그렇지 않고 테스트 데이터로 다시 새로운 스케일링 기준 정보를 만들게 되면 학습 데이터와 테스트 데이터의 스케일링 기준 정보가 서로 달라지기 때문에 올바른 예측 결과를 도출하지 못할 수 있다

학습 데이터를 0~10, 테스트 데이터를 0~5의 값을 가지는 ndarray 생성

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

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

학습 데이터

```
sclaer=MinMaxScaler()
# fit 하게 되면 최소값이 0, 최댓값이 10으로 설정
scaler.fit(train_array)
# 1/10 scale로 train_array 데이터 변환함. 10 -> 1로 변환됨
train_scaled=scaler.transform(train_array)

print('원본 train_array 데이터 :', np.round(train_array.reshape(-1),2))
print('scale이 적용된 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]
scale이 적용된 train_array 데이터: [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
```

테스트 데이터

```
#MinMaxScaler에 test_array를 fit하면 최대 5 최소 0으로 설정됨
scaler.fit(test_array)
#1/5 scale로 test_array 변환. 원본 5->1 변환
test_scaled=scaler.transform(test_array)

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

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

학습 데이터와 테스트 데이터의 스케일링이 맞지 않음을 알 수 있다.

따라서 테스트 데이터에 다시 fit()을 적용해서는 안 되며 학습 데이터로 이미 fit()이 적용된 Scaler 객체를 이용해 transform()으로 변환해야 한다.

테스트 데이터에 fit()을 호출하지 않고 학습 데이터로 fit()을 수행한 MinMaxScaler

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

test_scaled=scaler.transform(test_array)
print('원본 test_array:', np.round(test_array.reshape(-1),2))
print('scale이 적용된 test_array:', np.round(test_scaled.reshape(-1),2))
```

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

따라서, 전체 데이터 세트에 스케일링을 적용한 뒤 학습과 테스트 데이터 세트로 분리하는 것이 더 바람직하다.

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 transform으로 변환