

02장 - 사이킷런으로 시작하는 머신러닝

02-01 사이킷런 소개와 특징

- **사이킷런(Scikit-learn)** : 파이썬 머신러닝 라이브러리 중 가장 많이 사용되는 라이브러리
 - 파이썬 기반의 머신 러닝을 위한 가장 쉽고 효율적인 개발 라이브러리 제공.

```
1 !pip install scikit-learn
```

```
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/di
```

```
1 import sklearn
```

```
1 sklearn.__version__ # 사이킷런 버전 확인
```

```
'1.2.2'
```

02-02 첫 번째 머신러닝 만들어보기 - 붓꽃 품종 예측하기

붓꽃 데이터 세트로 붓꽃의 품종을 분류(Classification)하는 것.

- **분류(Classification)**
 - 대표적인 **지도학습(Supervised Learning)** 방법 중 하나
 - 레이블(Label; 다양한 피처와 분류 결정값) 데이터로 모델 학습, 테스트 데이터 세트에서 미지의 레이블 예측

```
1 from sklearn.datasets import load_iris # 데이터 세트 생성
2 from sklearn.tree import DecisionTreeClassifier # 알고리즘 의사 결정 트리 불러오기
3 from sklearn.model_selection import train_test_split # 데이터 세트를 학습 데이터+테스트 데이터
```

```
1 import pandas as pd
```

```
1 iris = load_iris() # 붓꽃 데이터 세트 불러오기
2
3 iris_data = iris.data # 피처 데이터
4 iris_label = iris.target # 레이블 데이터
5
6 iris_df = pd.DataFrame(data = iris_data, columns = iris.feature_names) # 데이터 프레임
7 iris_df['label'] = iris.target
8
9 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

```

1 # 학습(train) 데이터와 테스트(test) 데이터로 분리
2 # x: 피쳐, y: 레이블
3
4 # 학습 데이터 80%, 테스트 데이터 20%로 데이터 분할
5 X_train, X_test, y_train, y_test = train_test_split(iris_data, iris_label, test_size=
6
7 # test_size = n 에서 n은 테스트 데이터 세트의 비율
8 # random_state = n 에서 n은 어떤 값이든 상관없음(난수 발생 값)

```

```

1 # DecisionTreeClassifier 객체 생성
2 dt_clf = DecisionTreeClassifier(random_state = 11)

```

```

1 # 학습 수행
2 dt_clf.fit(X_train, y_train)

```

```

▼ DecisionTreeClassifier
DecisionTreeClassifier(random_state=11)

```

```

1 # 예측 수행
2 pred = dt_clf.predict(X_test)

1 # 예측 성능 평가
2 from sklearn.metrics import accuracy_score
3 print('예측 정확도: {0:.4f}'.format(accuracy_score(y_test, pred)))

```

예측 정확도: 0.9333

분류 예측 프로세스 정리

- 데이터 세트 분리
- 모델 학습
- 예측 수행
- 평가

02-03 사이킷런의 기반 프레임워크 익히기

- **Estimator 클래스** : 지도학습의 모든 알고리즘을 구현한 클래스
 - **Classifier** : 분류 알고리즘을 구현한 클래스
 - **Regressor** : 회귀 알고리즘을 구현한 클래스
 - `fit()`, `predict()`를 내부에서 구현
 - `evaluation` 함수, 하이퍼 파라미터 튜닝 지원 클래스에서 인자로 받음

◦ 비지도학습(차원 축소, 클러스터링, 피쳐 추출 등) 구현 클래스에서도 적용.

- `fit()` : 입력 데이터의 형태에 맞춰 데이터를 변환하기 위한 사전 구조를 맞추는 작업
- `transform()` : 입력 데이터의 차원 변환, 클러스터링, 피쳐 추출 등 실제 작업 수행

• 사이킷런의 주요 모듈

분류	모듈명	설명
예제 데이터	<code>sklearn.datasets</code>	사이킷런에 내장되어 예제로 제공하는 데이터 세트
피쳐 처리	<code>sklearn.preprocessing</code>	데이터 전처리에 필요한 다양한 가공 기능 제공(문자열을 숫자 형 코드 값으로 인코딩, 정규화, 스케일링 등)
	<code>sklearn.feature_selection</code>	알고리즘에 큰 영향을 미치는 피쳐를 우선순위대로 선택 작업 수행하는 다양한 기능 제공

피처 처리	<code>sklearn.feature_extraction</code>	<p>텍스트 데이터나 이미지 데이터의 벡터화된 피처를 추출하는 데 사용됨.</p> <p>예를 들어 텍스트 데이터에서 Count Vectorizer나 Tf-Idf Vectorizer 등을 생성하는 기능 제공.</p> <p>텍스트 데이터의 피처 추출은 <code>sklearn.feature_extraction.text</code> 모듈에, 이미지 데이터의 피처 추출은 <code>sklearn.feature_extraction.image</code> 모듈에 지원 API가 있음.</p>
피처 처리 & 차원 축소	<code>sklearn.decomposition</code>	차원 축소와 관련한 알고리즘을 지원하는 모듈임. PCA, NMF, Truncated SVD 등을 통해 차원 축소 기능을 수행할 수 있음
데이터 분리, 검증 & 파라미터 튜닝	<code>sklearn.model_selection</code>	교차 검증을 위한 학습용/테스트용 분리, 그리드 서치(Grid Search)로 최적 파라미터 추출 등의 API 제공
평가	<code>sklearn.metrics</code>	<p>분류, 회귀, 클러스터링, 페어와이즈(Pairwise)에 대한 다양한 성능 측정 방법 제공</p> <p>Accuracy, Precision, Recall, ROC-AUC, RMSE 등 제공</p>
ML 알고리즘	<code>sklearn.ensemble</code>	<p>앙상블 알고리즘 제공</p> <p>랜덤 포레스트, 에이다 부스트, 그래디언트 부스팅 등을 제공</p>
	<code>sklearn.linear_model</code>	주로 선형 회귀, 릿지(Ridge), 라쏘(Lasso) 및 로지스틱 회귀 등 회귀 관련 알고리즘을 지원. 또한 SGD(Stochastic Gradient Descent) 관련 알고리즘도 제공
	<code>sklearn.naive_bayes</code>	나이브 베이즈 알고리즘 제공. 가우시안 NB, 다항 분포 NB 등.
	<code>sklearn.neighbors</code>	최근접 이웃 알고리즘 제공. K-NN 등
	<code>sklearn.svm</code>	서포트 벡터 머신 알고리즘 제공
	<code>sklearn.tree</code>	의사 결정 트리 알고리즘 제공
유틸리티	<code>sklearn.cluster</code>	비지도 클러스터링 알고리즘 제공 (K-평균, 계층형, DBSCAN 등)
	<code>sklearn.pipeline</code>	피처 처리 등의 변환과 ML 알고리즘 학습, 예측 등을 함께 묶어서 실행할 수 있는 유틸리티 제공

내장된 예제 데이터 세트

분류나 회귀 연습용 예제 데이터

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

분류와 클러스터링을 위한 표본 데이터 생성기

- `datasets.make_classifications()` : 분류를 위한 데이터 세트 생성
- `datasets.make_blobs()` : 클러스터링을 위한 데이터 세트 무작위 생성

키(Key)

- `data`: 피쳐의 데이터 세트
 - 넘파이 배열(ndarray) 타입
- `target`: 분류 시 레이블 값, 회귀 시 숫자 결괏값 데이터 세트
 - 넘파이 배열(ndarray) 타입
- `target_names`: 개별 레이블의 이름
 - 넘파이 배열(ndarray) 타입
 - 파이썬 리스트(list) 타입
- `feature_names`: 피쳐의 이름
 - 넘파이 배열(ndarray) 타입
 - 파이썬 리스트(list) 타입
- `DESCR`: 데이터 세트에 대한 설명과 각 피쳐의 설명
 - 스트링 타입

```
1 from sklearn.datasets import load_iris
```

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

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

```
1 keys = iris_data.keys()
2 print(keys)
```

```
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names', 'fi
```

```
1 print("\n feature name:", iris_data.feature_names) # feature name
2 print("\n shape:", len(iris_data.feature_names)) # feature name의 shape
3 print("\n type:", type(iris_data.feature_names)) # feature name의 type
4
5 print("\n target name:", iris_data.target_names) # target name
6 print("\n shape:", len(iris_data.target_names)) # target name의 shape
7 print("\n type:", type(iris_data.target_names), '\n') # target name의 type
8
9 print("\n data:", iris_data.data) # data
10 print("\n shape:", len(iris_data.data)) # data의 shape
11 print("\n type:", type(iris_data.data), '\n') # data의 type
12
13 print("\n target:", iris_data.target) # target
14 print("\n shape:", len(iris_data.target)) # target의 shape
15 print("\n type: ", type(iris_data.target), '\n') # target의 type
```



```

1 # K 폴드 교차 검증 프로세스 구현 - KFold & StratifiedKFold 클래스 활용
2 from sklearn.tree import DecisionTreeClassifier
3 from sklearn.metrics import accuracy_score
4 from sklearn.model_selection import KFold
5 import numpy as np
6
7 iris = load_iris()
8 features = iris.data
9 label = iris.target
10 dt_clf = DecisionTreeClassifier(random_state=156)
11
12 # 5개의 폴드 세트로 분리하는 KFold 객체와 폴드 세트별 정확도를 담을 리스트 객체 생성.
13 kfold = KFold(n_splits=5)
14 cv_accuracy = []
15 print('붓꽃 데이터 세트 크기: ', features.shape[0])

```

붓꽃 데이터 세트 크기: 150

```

1 n_iter = 0
2
3 # KFold 객체의 split()를 호출하면 폴드 별 학습용, 검증용 테스트의 로우 인덱스를 array로 반환
4 for train_index, test_index in kfold.split(features):
5     # kfold.split()으로 반환된 인덱스를 이용해 학습용, 검증용 테스트 데이터 추출
6     X_train, X_test = features[train_index], features[test_index]
7     y_train, y_test = label[train_index], label[test_index]
8
9     # 학습 및 예측
10    dt_clf.fit(X_train, y_train)
11    pred = dt_clf.predict(X_test)
12    n_iter += 1
13
14    # 반복 시마다 정확도 측정
15    accuracy = np.round(accuracy_score(y_test, pred), 4)
16    train_size = X_train.shape[0]
17    test_size = X_test.shape[0]
18    print ("\n#{0} 교차검증정확도:{1}, 학습데이터 크기: {2}, 검증데이터 크기: {3}".format(n_iter, accuracy, train_size, test_size))
19    print("#{0} 검증 세트 인덱스:{1}".format(n_iter, test_index))
20    cv_accuracy.append(accuracy)
21
22 # 개별 iteration별 정확도를 합하여 평균 정확도 계산
23 print("\n## 평균 검증 정확도:", np.mean(cv_accuracy))

```

#1 교차검증정확도:1.0, 학습데이터 크기: 120, 검증데이터 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 24 25 26 27 28 29]

#2 교차검증정확도:0.9667, 학습데이터 크기: 120, 검증데이터 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 54 55 56 57 58 59]

#3 교차검증정확도:0.8667, 학습데이터 크기: 120, 검증데이터 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 84 85 86 87 88 89]

#4 교차검증정확도:0.9333, 학습데이터 크기: 120, 검증데이터 크기: 30

#4 검증 세트 인덱스: [90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 108 109 110 111 112 113 114 115 116 117 118 119]

#5 교차검증정확도:0.7333, 학습데이터 크기: 120, 검증데이터 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

- 특정 레이블 값이 특이하게 많거나 매우 적어서 값의 분포가 한쪽으로 치우치는 것을 말함.
- K 폴드가 레이블 데이터 집합이 원본 데이터 집합의 레이블 분포를 학습 및 테스트 세트에 제대로 분배하지 못하는 경우의 문제를 해결해줌.

```
1 import pandas as pd
2 iris = load_iris()
3 iris_df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
4 iris_df['label'] = iris.target
5 iris_df['label'].value_counts()
```

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

```
1 kfold = KFold(n_splits=3)
2 n_iter = 0
3 for train_index, test_index in kfold.split(iris_df):
4     n_iter += 1
5     label_train = iris_df['label'].iloc[train_index]
6     label_test = iris_df['label'].iloc[test_index]
7     print('##교차검증:{0}'.format(n_iter))
8     print('학습 레이블 데이터 분포 : \n' , label_train.value_counts())
9     print('검증 레이블 데이터 분포 : \n' , label_test.value_counts())
10    iris_df['label'].value_counts()
```

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

```

2    50
Name: label, dtype: int64

```

```

1 # StratifiedKFold를 수행하여 데이터 분할 후, 학습/검증 레이블 데이터의 분포도 확인하기
2 from sklearn.model_selection import StratifiedKFold
3
4 skf = StratifiedKFold(n_splits=3)
5 n_iter = 0
6
7 for train_index, test_index in skf.split(iris_df, iris_df['label']):
8     n_iter += 1
9     label_train= iris_df['label'].iloc[train_index]
10    label_test= iris_df['label'].iloc[test_index]
11    print('\n 교차검증:{0}'.format(n_iter))
12    print('학습레이블데이터분포:\n', label_train.value_counts())
13    print('검증레이블데이터분포:\n', label_test.value_counts())

```

```

교차검증:1
학습레이블데이터분포:
2    34
0    33
1    33
Name: label, dtype: int64
검증레이블데이터분포:
0    17
1    17
2    16
Name: label, dtype: int64

```

```

교차검증:2
학습레이블데이터분포:
1    34
0    33
2    33
Name: label, dtype: int64
검증레이블데이터분포:
0    17
2    17
1    16
Name: label, dtype: int64

```

```

교차검증:3
학습레이블데이터분포:
0    34
1    33
2    33
Name: label, dtype: int64
검증레이블데이터분포:
1    17
2    17
0    16
Name: label, dtype: int64

```

```

1 # StratifiedKFold를 이용한 데이터 분리
2
3 dt_clf = DecisionTreeClassifier(random_state=156)
4
5 skfold = StratifiedKFold(n_splits=3)
6 n_iter = 0
7 cv_accuracy = []
8
9 # StratifiedKFold의 split( ) 호출 시 반드시 레이블 데이터 세트도 추가 입력 필요
10 for train_index, test_index in skfold.split(features, label):
11     # split( )으로 반환된 인덱스를 이용해 학습용, 검증용 테스트 데이터 추출
12     X_train, X_test = features[train_index], features[test_index]
13     y_train, y_test = label[train_index], label[test_index]
14
15     # 학습 및 예측
16     dt_clf.fit(X_train, y_train)
17     pred = dt_clf.predict(X_test)
18
19     # 반복 시마다 정확도 측정
20     n_iter += 1
21     accuracy = np.round(accuracy_score(y_test, pred), 4)
22     train_size = X_train.shape[0]
23     test_size = X_test.shape[0]
24
25     print('\n#{0} 교차 검증 정확도:{1}, 학습데이터크기:{2}, 검증데이터크기:{3}'.format(n_iter, ac
26     print('#{0} 검증 세트 인덱스:{1}'.format(n_iter, test_index))
27     cv_accuracy.append(accuracy)
28
29 # 교차 검증별 정확도 및 평균 정확도 계산
30 print('\n## 교차 검증별 정확도: ', np.round(cv_accuracy, 4))
31 print('## 평균 검증 정확도: ', np.mean(cv_accuracy))

```

#1 교차 검증 정확도:0.98, 학습데이터크기:100, 검증데이터크기:50

#1 검증 세트 인덱스: [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 1
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, 학습데이터크기:100, 검증데이터크기:50

#2 검증 세트 인덱스: [17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 3
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, 학습데이터크기:100, 검증데이터크기:50

#3 검증 세트 인덱스: [34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 8
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

교차 검증을 보다 간편하게- cross_val_score()

: 교차 검증을 좀 더 편리하게 수행할 수 있게 해주는 API

```
cross_val_score(estimator, X, y=None, scoring=None, cv=None, n_jobs=1, verbose=0, fit_params=None)
```

- 주요 파라미터: estimator, X, y, scoring, cv

```
1 from sklearn.tree import DecisionTreeClassifier
2 from sklearn.model_selection import cross_val_score, cross_validate
3 from sklearn.datasets import load_iris
4
5 iris_data = load_iris()
6 dt_clf = DecisionTreeClassifier(random_state=156)
7 data = iris_data.data
8 label = iris_data.target
9
10 # 성능 지표는 정확도(accuracy), 교차 검증 세트는 3개
11 scores = cross_val_score(dt_clf, data, label, scoring='accuracy', cv=3)
12 print('교차 검증별 정확도: ', np.round(scores, 4))
13 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

```
1 # 파라미터 집합 만들고 순차적으로 적용하며 최적화 수행하기
2 grid_parameters = {'max_depth': [1, 2, 3],
3                    'min_samples_split': [2, 3]}
```

```

1 # GridSearch API 사용법
2
3 from sklearn.datasets import load_iris
4 from sklearn.tree import DecisionTreeClassifier
5 from sklearn.model_selection import GridSearchCV
6
7 # 데이터를 로딩하고 학습 데이터와 테스트 데이터 분리
8 iris = load_iris()
9 X_train, X_test, y_train, y_test = train_test_split(iris_data.data, iris_data.target)
10 dtree = DecisionTreeClassifier()
11
12 # 파라미터를 딕셔너리 형태로 설정
13 parameters = {'max_depth':[1, 2, 3], 'min_samples_split':[2, 3]}
14
15 # param_grid의 하이퍼 파라미터를 3개의 train, test set fold로 나누어 테스트 수행 설정.
16 # refit = True가 default임. True이면 가장 좋은 파라미터 설정으로 재학습시킴.
17 grid_dtree = GridSearchCV(dtree, param_grid=parameters, cv=3, refit=True)
18
19 # 붓꽃 학습 데이터로 param_grid의 하이퍼 파라미터를 순차적으로 학습/평가.
20 grid_dtree.fit(X_train, y_train)
21 # GridSearchCV 결과를 추출해 Dataframe으로 변환
22 scores_df = pd.DataFrame(grid_dtree.cv_results_)
23 scores_df[['params', 'mean_test_score', 'rank_test_score', 'split0_test_score', 'spli

```

	params	mean_test_score	rank_test_score	split0_test_score	spli
0	{'max_depth': 1, 'min_samples_split': 2}	0.700000	5	0.700	
1	{'max_depth': 1, 'min_samples_split': 3}	0.700000	5	0.700	
2	{'max_depth': 2, 'min_samples_split': 2}	0.958333	3	0.925	
3	{'max_depth': 2, 'min_samples_split': 3}	0.958333	3	0.925	
4	{'max_depth': 3, 'min_samples_split': 2}	0.975000	1	0.975	
5	{'max_depth': 3, 'min_samples_split': 3}	0.975000	1	0.975	

```

1 # 최적 하이퍼 파라미터의 값과 그때의 정확도 알아보기
2 print('GridSearchCV 최적 파라미터:', grid_dtree.best_params_)
3 print('GridSearchCV 최고 정확도: %.4f%%' % format(grid_dtree.best_score_))

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

```

```

1 # 데이터 세트 예측, 성능 평가
2
3 # Gridsearchcv의 refit으로 이미 학습된 estimator 반환
4 estimator = grid_dtrees.best_estimator_
5
6 # Gridsearchcv의 best_estimator_는 이미 최적 학습이 됐으므로 별도 학습이 필요없음
7 pred = estimator.predict(X_test)
8 print('테스트 데이터 세트 정확도: {0:.4f}'.format(accuracy_score(y_test, pred)))

```

테스트 데이터 세트 정확도: 0.9667

02-05 데이터 전처리(Data Preprocessing)

- 결손값(NaN, Null)은 허용되지 않음
 - 고정된 다른 값으로 변환해야 함
- 문자열 값을 입력 값으로 허용하지 않음

데이터 인코딩

- 레이블 인코딩(Label encoding)
 - : 카테고리 피처를 코드형 숫자 값으로 변환하는 것
- 원-핫 인코딩(One Hot encoding)

레이블 인코딩

- LabelEncoder 클래스로 구현
 - fit()
 - transform()

```

1 from sklearn.preprocessing import LabelEncoder
2
3 items = ['TV', '냉장고', '전자레인지', '컴퓨터', '선풍기', '선풍기', '믹서', '믹서']
4
5 #LabelEncoder를 객체로 생성한 후, fit( )과 transform( )으로 레이블 인코딩 수행.
6 encoder = LabelEncoder()
7 encoder.fit(items)
8 labels = encoder.transform(items)
9 print('인코딩 변환값: ', labels)

```

인코딩 변환값: [0 1 4 5 3 3 2 2]

```

1 # 속성값 확인
2 print('인코딩 클래스: ', encoder.classes_)

```

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

```

1 # 디코딩 원본값 확인
2 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 인코딩은 간단하게 문자열 값을 숫자형 카테고리 값으로 변환.

원-핫 인코딩(One-Hot Encoding)

: 피쳐값의 유형에 따라 새로운 피쳐를 추가해 고유값에 해당하는 칼럼에만 1을 표시하고 나머지 칼럼에는 0을 표시하는 방식

: 행 형태로 돼 있는 피쳐의 고유값을 열 형태로 차원을 변환한 뒤, 고유값에 해당하는 칼럼에만 1을 표시하고 나머지 칼럼에는 0을 표시

```

1  # OneHotEncoder를 이용해 앞의 데이터를 원-핫 인코딩으로 변환해보겠습니다.
2  from sklearn.preprocessing import OneHotEncoder
3  import numpy as np
4
5  items=['TV', '냉장고', '전자레인지', '컴퓨터', '선풍기', '선풍기', '믹서', '믹서']
6
7  # 먼저 숫자값으로 변환을 위해 LabelEncoder로 변환합니다.
8  encoder = LabelEncoder()
9  encoder.fit(items)
10 labels = encoder.transform(items)
11
12 # 2차원 데이터로 변환합니다.
13 labels = labels.reshape(-1, 1)
14
15 # 원- 핫 인코딩을 적용합니다.
16 oh_encoder = OneHotEncoder()
17 oh_encoder.fit(labels)
18 oh_labels = oh_encoder.transform(labels)
19
20 print('원-핫 인코딩 데이터')
21 print(oh_labels.toarray())
22
23 print('원-핫 인코딩 데이터 차원')
24 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)

```

1 # 원-핫 인코딩을 더 쉽게 지원하는 API - get_dummies()
2
3 import pandas as pd
4
5 df = pd.DataFrame({'item': ['TV', '냉장고', '전자레인지', '컴퓨터', '선풍기', '선풍기', '믹서', '믹서']})
6 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	

피쳐 스케일링과 정규화

- 표준화를 통해 변환될 피쳐 x 의 새로운 i 번째 데이터를 $x_{i\text{new}}$ 라고 하자.

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

- 새로운 데이터 $x_{i\text{new}}$ 는 원래 값에서 피쳐 x 의 최솟값을 뺀 값을 피쳐 i 의 최댓값과 최솟값의 차이로 나눈 값으로 변환 가능

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

- 정규화

$$x_{i\text{new}} = \frac{x_i}{\sqrt{(x_i^2 + y_i^2 + z_i^2)}}$$

StandardScaler

: 표준화를 쉽게 지원하기 위한 클래스

- 개별 피쳐를 평균이 0이고, 분산이 1인 값으로 변환해줌.
- 가우시안 정규 분포를 따름.

```

1 # StandardScaler가 어떻게 데이터 값을 변환하는지 확인하기
2
3 from sklearn.datasets import load_iris
4 import pandas as pd
5
6 # 붓꽃 데이터 세트를 로딩하고 DataFrame으로 변환합니다.
7 iris = load_iris()
8 iris_data = iris.data
9 iris_df = pd.DataFrame(data=iris_data, columns=iris.feature_names)
10
11 print('feature들의 평균값')
12 print(iris_df.mean())
13
14 print('\n feature들의 분산값')
15 print(iris_df.var())

```



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

```
feature들의 분산값
sepal length (cm)    0.685694
sepal width (cm)     0.189979
petal length (cm)    3.116278
petal width (cm)     0.581006
dtype: float64
```

```
1 # DataFrame으로 변환해 재확인하기
2 from sklearn.preprocessing import StandardScaler
3 # StandardScaler 객체 생성
4 scaler = StandardScaler()
5 # StandardScaler로 데이터 세트 변환. fit( )과 transform( ) 호출.
6 scaler.fit(iris_df)
7 iris_scaled = scaler.transform(iris_df)
8 # transform( )시 스케일 변환된 데이터 세트가 Numpyndarray로 반환돼 이를 DataFrame으로 변환
9 iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_names)
10 print('feature들의 평균값')
11 print(iris_df_scaled.mean())
12 print('\nfeature들의 분산값')
13 print(iris_df_scaled.var())
```

```
feature들의 평균값
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 분산값
sepal length (cm)    1.006711
sepal width (cm)     1.006711
petal length (cm)    1.006711
petal width (cm)     1.006711
dtype: float64
```

MinMaxScaler

: 데이터값을 0과 1 사이의 범위값으로 변환

```
1 # MinMaxScaler 작동 메커니즘 확인
2 from sklearn.preprocessing import MinMaxScaler
3
4 # MinMaxScaler 객체 생성
5 scaler = MinMaxScaler()
6
7 # MinMaxScaler로 데이터 세트 변환. fit()과 transform() 호출.
8 scaler.fit(iris_df)
9 iris_scaled = scaler.transform(iris_df)
10
11 # transform( ) 시 스케일 변환된 데이터 세트가 NumPyndarray로 반환돼 이를 Dataframe으로 변환
12 iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_names)
13
14 print('feature들의 최솟값' )
15 print(iris_df_scaled.min())
16
17 print('\nfeature들의 최댓값' )
18 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

: 학습 데이터로 fit()이 적용된 스케일링 기준 정보를 그대로 테스트 데이터에 적용해야 하며, 그렇지않고 테스트 데이터로 다시 새로운 스케일링 기준 정보를 만들게 되면 학습 데이터와 테스트 데이터의 스케일링 기준 정보가 서로 달라지기 때문에 올바른 예측 결과를 도출하지 못할수있음.

```

1 # 테스트 데이터에 fit()을 적용할 때 생기는 문제점
2
3 from sklearn.preprocessing import MinMaxScaler
4 import numpy as np
5
6 # 학습 데이터는 0부터 10까지, 테스트 데이터는 0부터 5까지 값을 가지는 데이터 세트로 생성
7 # Scaler 클래스의 fit(), transform()은 2차원 이상 데이터만 가능하므로 reshape(-1, 1)로 차원 변경
8 train_array = np.arange(0, 1).reshape(-1, 1)
9 test_array = np.arange(0, 6).reshape(-1, 1)
10
11 # MinMaxScaler 객체에 별도의 feature range 파라미터값을 지정하지 않으면 0~1 값으로 변환
12 scaler = MinMaxScaler()
13
14 # fit() 하게 되면 train_array 데이터의 최솟값이 0, 최댓값이 10으로 설정.
15 scaler.fit(train_array)
16
17 # 1/10 scale로 train_array 데이터 변환함. 원본 10 -> 1 로 변환됨.
18 train_scaled = scaler.transform(train_array)
19 print('원본 train_array 데이터:', np.round(train_array.reshape(-1), 2))
20 print('Scale된 train_array 데이터:', np.round(train_scaled.reshape(-1), 2))

```

원본 train_array 데이터: [0]
Scale된 train_array 데이터: [0.]

```

1 # MinMaxScaler에 test_array를 fit() 하게 되면 원본 데이터의 최솟값이 0, 최댓값이 5로 설정됨
2 scaler.fit(test_array)
3
4 # 1/5 scale로 test_array 데이터 변환함. 원본 5->1로 변환.
5 test_scaled = scaler.transform(test_array)
6
7 # test_array의 scale 변환 출력.
8 print('원본 test_array 데이터 : ', np.round(test_array.reshape(-1), 2))
9 print('Scale된 test_array 데이터:', np.round(test_scaled.reshape(-1), 2))

```

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

```

1 scaler = MinMaxScaler()
2 scaler.fit(train_array)
3 train_scaled = scaler.transform(train_array)
4
5 print('원본 train_array 데이터:', np.round(train_array.reshape(-1), 2))
6 print('Scale된 train_array 데이터:', np.round(train_scaled.reshape(-1), 2))
7
8 # test_array에 Scale 변환을 할 때는 반드시 fit()을 호출하지 않고 transform()만으로 변환해야함.
9 test_scaled = scaler.transform(test_array)
10 print('\n원본 test_array 데이터: ', np.round(test_array.reshape(-1), 2))
11 print('Scale된 test_array 데이터:', np.round(test_scaled.reshape(-1), 2))

```

원본 train_array 데이터: [0]
Scale된 train_array 데이터: [0.]

원본 test_array 데이터: [0 1 2 3 4 5]
Scale된 test_array 데이터: [0. 1. 2. 3. 4. 5.]

02-06 사이킷런으로 수행하는 타이타닉 생존자 예측(실습)

```

1 from sklearn.preprocessing import LabelEncoder
2
3 # Null 처리 함수
4 def fillna(df):
5     df['Age'].fillna(df['Age'].mean(), inplace=True)
6     df['Cabin'].fillna('N', inplace=True)
7     df['Embarked'].fillna('N', inplace=True)
8     df['Fare'].fillna(0, inplace=True)
9     return df
10
11 # 머신러닝 알고리즘에 불필요한 속성 제거
12 def drop_features(df):
13     df.drop(['PassengerId', 'Name', 'Ticket'], axis=1, inplace=True)
14     return df
15
16 # 레이블 인코딩 수행.
17 def format_features(df):
18     df['Cabin'] = df['Cabin'].str[:1]
19     features = ['Cabin', 'Sex', 'Embarked']
20     for feature in features:
21         le = LabelEncoder()
22         le = le.fit(df[feature])
23         df[feature] = le.transform(df[feature])
24     return df
25
26 # 앞에서 설정한 데이터 전처리 함수 호출
27 def transform_features(df):
28     df = fillna(df)
29     df = drop_features(df)
30     df = format_features(df)
31     return df

```

✓ 03장 예측

03-01 정확도(Accuracy)

$$\text{정확도(Accuracy)} = \frac{\text{예측결과가동일한데이터건수}}{\text{전체예측데이터건수}}$$

: 직관적으로 모델 예측 성능을 나타내는 평가 지표

```

1 # 타이타닉 생존자 예측 수행
2
3 from sklearn.base import BaseEstimator
4 class MyDummyClassifier(BaseEstimator):
5     # fit( ) 메서드는 아무것도 학습하지않음.
6     def fit(self, X, y=None):
7         pass
8
9 # predict() 메서드는 단순히 Sex 피처가 1이면 0, 그렇지 않으면 1로 예측함.
10 def predict(self, X):
11     pred = np.zeros((X.shape[0], 1))
12     for i in range(X.shape[0]) :
13         if X['Sex'].iloc[i] == 1:
14             pred[i] = 0
15         else :
16             pred[i] = 1
17     return pred

```

```

1 import pandas as pd
2 from sklearn.model_selection import train_test_split
3 from sklearn.metrics import accuracy_score
4 import numpy as np
5
6 # 원본 데이터 재로딩 , 데이터 가공 , 학습 데이터/테스트 데이터 분할.
7 titanic_df = pd.read_csv('/content/train.csv')
8 y_titanic_df = titanic_df['Survived']
9 X_titanic_df = titanic_df.drop(['Survived'], axis=1)
10 X_titanic_df = transform_features(X_titanic_df)
11 X_train, X_test, y_train, y_test = train_test_split(X_titanic_df, y_titanic_df, test
12
13 # dummy classifier를 이용한 학습/예측/평가 수행
14 myclf = MyDummyClassifier()
15 myclf.fit(X_train, y_train)
16
17 mypredictions = myclf.predict(X_test)
18 print('DummyClassifier의 정확도는:{0:.4f}'.format(accuracy_score(y_test, mypredictions))

```

DummyClassifier의 정확도는:0.7877

```

1 from sklearn.datasets import load_digits
2 from sklearn.model_selection import train_test_split
3 from sklearn.base import BaseEstimator
4 from sklearn.metrics import accuracy_score
5 import numpy as np
6 import pandas as pd
7
8 class MyFakeClassifier(BaseEstimator):
9     def fit(self, X, y):
10         pass
11 # 입력값으로 들어오는 X 데이터 세트의 크기만큼 모두 0값으로 만들어서 반환
12     def predict(self, X):
13         return np.zeros((len(X), 1), dtype=bool)
14
15 # 사이킷런의 내장 데이터 세트인 load_digits()를 이용해 MNIST 데이터 로딩
16 digits = load_digits()
17 # digits 번호가 7이면 True이고 이를 astype(int)로 1로 변환, 7번이 아니면 False이고 0으로 변환.
18 y = (digits.target == 7).astype(int)
19 X_train, X_test, y_train, y_test = train_test_split(digits.data, y, random_state=11)
20
21
22 # Dummy Classifier로 학습/예측/정확도 평가
23 fakeclf = MyFakeClassifier()
24 fakeclf.fit(X_train, y_train)
25
26 fakepred = fakeclf.predict(X_test)
27 print('모든 예측을 0으로 하여도 정확도는:{:.3f}'.format(accuracy_score(y_test, fakepred)))

```

모든 예측을 0으로 하여도 정확도는:0.900

03-02 오차 행렬(Confusion Matrix)

: 학습된 분류모델이 예측을 수행하면서 얼마나 헛갈리고(confused) 있는지도 함께 보여주는 지표

: 이진분류의 예측 오류가 얼마인지와 더불어 어떠한 유형의 예측 오류가 발생하고 있는지를 함께 나타내는 지표

- TN: 예측값을 Negative값 0으로 예측했고 실제값 역시 Negative값 0
- FP: 예측값을 Positive값 1로 예측했는데 실제값은 Negative값 0
- FN: 예측값을 Negative값 0으로 예측했는데 실제값은 Positive값 1
- TP: 예측값을 Positive값 1로 예측했는데 실제값 역시 Positive값 1

```

1 # MyFakeClassifier의 예측 성능 지표를 오차 행렬로 표현해보기
2 from sklearn.metrics import confusion_matrix
3 confusion_matrix(y_test, fakepred)

```

```

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

```

정확도 = 예측 결과와 실제값이 동일한 건수/전체 데이터수 = $(TN+TP)/(TN+FP+FN+TP)$

03-03 정밀도와 재현율

정밀도 = $TP / (FP + TP)$

재현율 = $TP / (FN + TP)$

- 정밀도: 예측을 Positive로 한 대상 중에 예측과 실제값이 Positive로 일치한 데이터의 비율
- 재현율: 실제값이 Positive인 대상 중에 예측과 실제값이 Positive로 일치한 데이터의 비율
 - 모두 TP를 높이는 데 초점을 둬
 - 재현율을 FN을 낮추는데 초점을 둬
 - 정밀도는 FP를 낮추는데 초점을 둬

```
1 from sklearn.metrics import accuracy_score, precision_score, recall_score, confusion_matrix
2
3 def get_clf_eval(y_test, pred):
4     confusion = confusion_matrix(y_test, pred)
5     accuracy = accuracy_score(y_test, pred)
6     precision = precision_score(y_test, pred)
7     recall = recall_score(y_test, pred)
8     print('오차행렬')
9     print(confusion)
10    print('정확도:{0:.4f}, 정밀도:{1:.4f}, 재현율:{2:.4f}'.format(accuracy, precision, recall))
```

```
1 # 로지스틱 회귀 기반 타이타닉 생존자 예측, confusion matrix, accuracy, precision, recall 평가 :
2 import pandas as pd
3 from sklearn.model_selection import train_test_split
4 from sklearn.linear_model import LogisticRegression
5
6 # 원본 데이터를 재로딩, 데이터 가공, 학습 데이터/테스트 데이터 분할.
7 titanic_df = pd.read_csv('/content/train.csv')
8 y_titanic_df = titanic_df['Survived']
9 X_titanic_df = titanic_df.drop(['Survived'], axis=1)
10 X_titanic_df = transform_features(X_titanic_df)
11
12 X_train, X_test, y_train, y_test = train_test_split(X_titanic_df, y_titanic_df, test_size=0.2)
13
14 lr_clf = LogisticRegression()
15
16 lr_clf.fit(X_train, y_train)
17 pred = lr_clf.predict(X_test)
18 get_clf_eval(y_test, pred)
```

오차행렬

```
[[104  14]
```

```
 [ 13  48]]
```

정확도:0.8492, 정밀도:0.7742, 재현율:0.7869

/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:458: Conv
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:

<https://scikit-learn.org/stable/modules/preprocessing.html>

Please also refer to the documentation for alternative solver options:

https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression

n_iter_i = _check_optimize_result(

정밀도/재현율 트레이드오프

: 정밀도와 재현율은 상호 보완적인 평가지표이기 때문에 어느 한쪽을 강제로 높이면 다른 하나의 수치는 떨어지기 쉽다
는 것.

```
1 pred_proba = lr_clf .predict_proba(X_test)
2 pred = lr_clf.predict(X_test)
3 print('pred_proba() 결과 Shape : {0}'.format(pred_proba.shape))
4 print('pred_proba array에서 앞 3개만 샘플로 추출 \n : ' , pred_proba[:3])
5
6 # 예측 확률 array와 예측 결과값 array를 병합(concatenate)해 예측 확률과 결과값을 한눈에 확인
7 pred_proba_result = np.concatenate([pred_proba, pred.reshape(-1, 1)], axis=1)
8 print('두 개의 class 중에서 더 큰 확률을 클래스값으로 예측 \n',pred_proba_result[:3])
```

```
pred_proba() 결과 Shape : (179, 2)
pred_proba array에서 앞 3개만 샘플로 추출
: [[0.46197474 0.53802526]
   [0.87872398 0.12127602]
   [0.87719492 0.12280508]]
두 개의 class 중에서 더 큰 확률을 클래스값으로 예측
[[0.46197474 0.53802526 1.
   [0.87872398 0.12127602 0.
   [0.87719492 0.12280508 0.]
```

```
1 # 정밀도와 재현율 곡선 시각화
2 import matplotlib.pyplot as plt
3 import matplotlib.ticker as ticker
4 %matplotlib inline
```

```
1 from sklearn.metrics import precision_recall_curve
2 pred_proba_class1 = lr_clf.predict_proba(X_test)[: , 1]
3 precisions, recalls, thresholds = precision_recall_curve(y_test, pred_proba_class1)
4
5 def precision_recall_curve_plot(y_test, pred_proba_c1):
6 # threshold ndarray와 이 threshold에 따른 정밀도, 재현율 ndarray 추출.
7     precisions, recalls, thresholds = precision_recall_curve(y_test, pred_proba_c1)
8
9 # X축을 threshold 값으로, Y축은 정밀도, 재현율값으로 각각 Plot 수행.
10 # 정밀도는 점선으로 표시
11     plt.figure(figsize=(8, 6))
12     threshold_boundary = thresholds.shape[0]
13     plt.plot(thresholds, precisions[0: threshold_boundary], linestyle='-', label='pr')
14     plt.plot(thresholds, recalls[0: threshold_boundary], label='recall')
15
16 # threshold 값 X축의 Scale을 0.1 단위로 변경
17     start, end = plt.xlim()
18     plt.xticks(np.round(np.arange(start, end, 0.1), 2))
19
20 # X축, y축 label과 legend, 그리고 grid설정
21     plt.xlabel('Threshold value'); plt.ylabel('Precision and Recall value')
22     plt.legend(); plt.grid()
23     plt.show()
24
25 precision_recall_curve_plot(y_test, lr_clf.predict_proba(X_test)[: , 1])
```