



2장. 사이킷런으로 시작하는 머신러닝

EURON 6기 초급세션 김수연

01. 사이킷런 소개와 특징

📌 사이킷런(scikit-learn)

```
import sklearn
```

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

02. 첫 번째 머신러닝 만들어 보기 - 붓꽃 품종 예측하기 (분류)



지도학습은 명확한 정답이 주어진 데이터를 먼저 학습한 뒤 미지의 정답을 예측하는 방식

지도학습은 학습을 위한 다양한 피쳐와 분류 결정값인 레이블(Label) 데이터로 모델을 학습한 뒤, 별도의 테스트 데이터 세트에서 미지의 레이블을 예측
이 때 학습을 위해 주어진 데이터 세트를 학습 데이터 세트, 머신러닝 모델의 예측 성능을 평가하기 위해 별도로 주어진 데이터를 테스트 데이터 세트로 지칭

사이킷런 패키지 내의 모듈명은 `sklearn` 으로 시작

`sklearn.datasets` 내의 모듈은 사이킷런에서 자체적으로 제공하는 데이터 세트를 생성하는 모듈의 모임

`sklearn.tree` 내의 모듈은 트리 기반 ML 알고리즘을 구현한 클래스의 모임

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

하이퍼 파라미터란 머신러닝 알고리즘별로 최적의 학습을 위해 직접 입력하는 파라미터들을 통칭, 하이퍼 파라미터를 통해 머신러닝 알고리즘 성능을 튜닝

```
# 붓꽃 데이터 세트 생성
from sklearn.datasets import load_iris

# ML 알고리즘은 의사 결정 트리 알고리즘으로 이를 구현한 DecisionTreeClassifier 적용
from sklearn.tree import DecisionTreeClassifier

#데이터 세스를 학습 데이터와 테스트 데이터로 분리
from sklearn.model_selection import train_test_split
```


```
# 붓꽃 데이터 세트를 로딩
iris = load_iris()

# iris.data는 데이터 세트에서 피쳐(feature)만으로 된 데이터를 numpy로 가지고 있다.
iris_data = iris.data

# iris.target은 붓꽃 데이터 세트에서 레이블(결정 값) 데이터를 numpy로 가지고 있다.
iris_label = iris.target
```

```
print('iris target값: ', iris_label)
print('iris target명: ', iris.target_names)


# 붓꽃 데이터 세트를 자세히 보기 위해 DataFrame으로 변환
iris_df = pd.DataFrame(data=iris_data, columns=iris.feature_names)
iris_df['label'] = iris.target # df에 'label' column 추가하여 target 값 삽입
iris_df.head(3)
```


 `train_test_split` : 학습 데이터와 테스트 데이터를 `test_size` 파라미터 입력 값의 비율로 분할

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

피쳐 데이터 세트, 레이블 데이터 세트, 전체 데이터 세트 중 테스트 데이터 세트의 비율을 파라미터로 입력

`random_state` : 호출할 때마다 같은 학습/테스트 용 데이터 세트를 생성하기 위해 주어지는 난수 발생 값. 지정하지 않을 경우 수행할 때마다 다른 데이터가 만들어진다. 숫자 자체는 어떤 값을 지정해도 무관하다.


 `fit()` : 학습용 피쳐 데이터 속성과 결정값 데이터 세트를 입력해 호출하면 학습을 수행

 `predict()` : 테스트용 피쳐 데이터 세트를 입력해 호출하면 학습된 모델 기반에서 테스트 데이터 세트에 대한 예측값 반환

```
# DecisionTreeClassifier(사이킷런의 의사결정 트리 클래스) 객체 생성
dt_clf = DecisionTreeClassifier(random_state=11)

# 학습 수행
dt_clf.fit(X_train, y_train)

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

 `accuracy_score()` : 실제 레이블 데이터 세트, 예측 레이블 데이터 세트를 입력하여 정확도를 평가

```
from sklearn.metrics import accuracy_score
print('예측 정확도: {0: .4f}'.format(accuracy_score(y_test, pred)))
# 예측 정확도: 0.9333
```

03. 사이킷런의 기반 프레임워크 익히기

Estimator 이해 및 fit(), predict() 메서드



Estimator 클래스란, 지도학습의 알고리즘을 구현한 모든 클래스를 통칭하여 말하는 것

모든 사이킷런 클래스는 `fit()` 와 `predict()` 만을 이용해 간단하게 학습과 예측 결과를 반환

- Classifier : 분류 알고리즘을 구현한 클래스
- Regressor: 회귀 알고리즘을 구현한 클래스

사이킷런의 주요 모듈

분류	모듈명	설명
예제 데이터	<code>sklearn.datasets</code>	사이킷런에 내장되어 예제로 제공하는 데이터 세트
피처 처리	<code>sklearn.preprocessing</code>	데이터 전처리에 필요한 다양한 가공 기능 제공(문자열을 숫자형 코드 값으로 인코딩, 정규화, 스케일링 등)
	<code>sklearn.feature_selection</code>	알고리즘에 큰 영향을 미치는 피처를 우선순위로 선택 작업 수행하는 다양한 기능 제공
피처 처리	<code>sklearn.feature_extraction</code>	텍스트 데이터나 이미지 데이터의 벡터화된 피처를 추출하는 데 사용됨. 예를 들어 텍스트 데이터에서 Count Vectorizer나 Tfidf Vectorizer 등을 생성하는 기능 제공. 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.
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>	분류, 회귀, 클러스터링, 페어와이즈(Pairwise)에 대한 다양한 성능 측정 방법 제공 Accuracy, Precision, Recall, ROC-AUC, RMSE 등 제공
ML 알고리즘	<code>sklearn.ensemble</code>	앙상블 알고리즘 제공 랜덤 포레스트, 에이다 부스트, 그래디언트 부스팅 등을 제공
	<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
	<code>sklearn.cluster</code>	비지도 클러스터링 알고리즘 제공 (K-평균, 계층형, DBSCAN 등)
유틸리티	<code>sklearn.pipeline</code>	피처 처리 등의 변환과 ML 알고리즘 학습, 예측 등을 함께 묶어서 실행할 수 있는 유틸리티 제공

04. Model Selection 모듈 소개

사이킷런의 `model_selection` 모듈은 학습 데이터와 데이터 세트를 분리하거나 교차 검증 분할 및 평가, 그리고 Estimator의 하이퍼 파라미터를 튜닝하기 위한 다양한 함수와 클래스를 제공

학습/테스트 데이터 세트 분리 - `train_test_split()`

```
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, ra

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: 0.9666
```

 **train_test_split** : 원본 데이터 세트에서 학습 및 테스트 데이터 세트를 분리

피쳐 데이터 세트, 레이블 데이터 세트를 입력받고, 선택적으로 다음과 같은 파라미터를 입력받을 수 있다.

- **test_size** : 전체 데이터에서 테스트 데이터 세트 크기를 얼마로 샘플링할 것인가
- **train_size** : 전체 데이터에서 학습용 데이터 세트 크기를 얼마로 샘플링할 것인가
- **shuffle** : 데이터를 분리하기 전에 섞을지 → 데이터를 분산시켜 효율적인 데이터 세트를 만들 수 있다.
- **random_state** : 호출할 때마다 동일한 학습/테스트용 데이터 세트를 생성하기 위해 주어지는 난수 값

교차 검증



교차검증이란 데이터 편증을 막기 위해 별도로 여러 세트로 구성된 학습 데이터 세트와 검증 데이터 세트에서 학습과 평가를 수행하는 것을 말한다.

고정된 학습 데이터와 테스트 데이터로 평가를 하다 보면 테스트 데이터에만 최적의 성능을 발휘할 수 있도록 편향되게 모델을 유도하는 경향이 생긴다. 이러한 문제를 해결하기 위해 **교차 검증**을 이용해 다양한 학습과 평가를 수행할 수 있다.

K 폴드 교차 검증



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

예를 들어 5 폴드 교차 검증의 경우, 5개의 폴드된 데이터 세트를 학습과 검증을 위한 데이터 세트로 변경하면서 5번 평가를 수행한 뒤, 이 5개의 평가를 평균한 결과를 가지고 예측 성능을 평가.

먼저 데이터 세트를 5등분하고, 첫 번째 반복에서는 처음부터 4등분을 학습 데이터 세트, 마지막 5번째 등분 하나를 검증 데이터 세트로 설정하고 학습 데이터 세트에서 학습 수행, 검증 데이터 세트에서 평가를 수행한다. 첫 번째 평가를 수행하고 나면 두 번째 반복에서 다시 비슷한 학습과 평가 작업을 수행한다.

```
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 담은 리스트 객체 생성
kfold = KFold(n_splits=5)
cv_accuracy = []
print('붓꽃 데이터 세트 크기: ', features.shape[0])

# 붓꽃 데이터 세트 크기: 150

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
```

```

# 반복 시마다 정확도 측정
accuracy = np.round(accuracy_score(y_test, pred), 4)
train_size = X_train.shape[0]
test_size = X_test.shape[0]
print('\n#{0} 교차 검증도: {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))

```

위의 예제는 5개의 폴드 세트로 분리하는 KFold 객체와 폴드 세트별 정확도를 담은 리스트 객체 생성하고, `split()` 를 호출하여 교차 검증 수행 시마다 학습과 검증을 반복해 예측 정확도를 측정한다.

Stratified K 폴드



특정 레이블 값이 특이하게 많거나 매우 적은 등 불균형한 분포도를 가진 레이블(결정 클래스) 데이터 집합을 위한 K 폴드 방식

→ 원본 데이터의 레이블 분포를 먼저 고려한 뒤 이 분포와 동일하게 학습과 검증 데이터 세트를 분배

!! KFold와 달리 `split()` 메서드에 인자로 피쳐 데이터 세트뿐만 아니라 레이블 데이터 세트도 반드시 필요하다는 점을 주의한다.

```

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())

```

```

dt_clf = DecisionTreeClassifier(random_state=156)

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

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

    # 반복 시마다 정확도 측정
    n_iter += 1
    accuracy = np.round(accuracy_score(y_test, pred), 4)
    train_size = X_train.shape[0]
    test_size = X_test.shape[0]
    print('\n#{0} 교차 검증 정확도: {1}, 학습 데이터 크기: {2}, 검증 데이터 크기: {3}'.format(n_iter, accuracy, train_size, test_size))
    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))
```

🔍 교차 검증을 보다 간편하게 - `cross_val_score()`

KFold의 경우 폴드 세트를 설정하고, for 루프에서 반복으로 학습 및 테스트 데이터의 인덱스를 추출한 뒤, 반복적으로 학습과 예측을 수행하고 예측 성능을 반환하였다.

📌 `cross_val_score()` : KFold가 데이터를 학습하고 예측하는 일련의 과정을 한 번에 수행하는 API

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

iris_data = load_iris()
dt_clf = DecisionTreeClassifier(random_state=156)

data = iris_data.data
label = iris_data.target

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

피쳐 데이터 세트, 레이블 데이터 세트, 예측 성능 평가 지표, 교차 검증 폴드 수를 파라미터로 입력한다. cv로 지정된 횟수만큼 scoring 파라미터로 지정된 성능 지표 측정값을 배열 형태로 반환하며, 이를 평균하여 평가 수치로 사용한다.

GridSearchCV - 교차 검증과 최적 하이퍼 파라미터 튜닝을 한 번에

사이킷런은 `GridSearchCV` API를 이용해 Classifier나 Regressor와 같은 알고리즘에 사용되는 하이퍼 파라미터를 순차적으로 입력하면서 편리하게 최적의 파라미터를 도출할 수 있는 방안을 제공한다.

GridSearchCV는 교차 검증을 기반으로 하이퍼 파라미터의 최적 값을 찾게 해준다. 즉, 데이터 세트를 cross-validation을 위한 학습/테스트 세트로 자동으로 분할한 뒤에 하이퍼 파라미터 그리드에 기술된 모든 파라미터를 순차적으로 적용해 최적의 파라미터를 찾을 수 있게 한다.

주요 파라미터는 다음과 같다.

- `estimator` : classifier, regressor, pipeline 등이 사용
- `param_grid` : key + 리스트 값을 가지는 딕셔너리. estimator의 튜닝을 위해 파라미터명과 사용될 여러 파라미터 값
- `scoring` : 예측 성능을 측정할 평가 방법
- `cv` : 교차 검증을 위해 분할되는 학습/테스트 세트의 개수
- `refit` : True로 생성 시 가장 최적의 하이퍼 파라미터를 찾은 뒤 입력된 estimator 객체를 해당 하이퍼 파라미터로 재학습

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

# 데이터를 로딩하고 학습 데이터와 테스트 데이터 분리
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2, random_state=156)
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 default임. True이면 가장 좋은 파라미터 설정으로 재학습시킴.
grid_dtree = GridSearchCV(dtree, param_grid=parameters, cv=3, refit=True)
```

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

# GridSearchCV 결과를 추출해 DataFrame으로 변환
scores_df = pd.DataFrame(grid_dtrees.cv_results_)
scores_df[['params', 'mean_test_score', 'rank_test_score', 'split0_test_score', 'split1_test_score', 's

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


# GridSearchCV의 refit으로 이미 학습된 estimator 반환
estimator = grid_dtrees.best_estimator_

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


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

05. 데이터 전처리

ML 알고리즘은 데이터에 기반하고 있기 때문에 어떤 데이터를 입력으로 가지느냐에 따라 결과가 크게 달라진다. 따라서 사이킷런의 ML 알고리즘을 적용하기 전 데이터에 대해 다음과 같은 사항을 미리 처리한다.


 결손값, 즉 NaN, Null 값은 허용되지 않는다.

이러한 Null 값은 다른 값으로 변환해야 한다. 피쳐 값 중 Null 값이 얼마 되지 않는다면 피쳐의 평균값 등으로 간단히 대체할 수 있으나, Null 값이 대부분이라면 오히려 해당 피쳐는 드롭하는 것이 좋다.

 문자열 값을 입력값으로 허용하지 않는다.

모든 문자열 값은 인코딩돼서 숫자 형으로 변환해야 한다. 문자열 피쳐는 일반적으로 카테고리형 피쳐와 텍스트형 피쳐로 나뉜다.

데이터 인코딩

 **레이블 인코딩**: `LabelEncoder`를 이용해 카테고리형 피쳐를 코드형 숫자 값으로 변환

```
from sklearn.preprocessing import LabelEncoder

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

# LabelEncoder를 객체로 생성한 후, fit()과 transform()으로 레이블 인코딩 수행
encoder = LabelEncoder()
encoder.fit(items)
labels = encoder.transform(items)
print('인코딩 변환값: ', labels)
print('인코딩 클래스: ', encoder.classes_)
print('디코딩 원본값: ', encoder.inverse_transform([4, 5, 2, 0, 1, 1, 3, 3]))

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

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

일괄적인 숫자 값으로 변환되며 특정 ML 알고리즘에서 가중치가 부여되거나 더 중요하게 인식할 가능성이 있으므로 선형 회귀와 같은 ML 알고리즘에는 적용하지 않아야 한다.

🔍 원-핫 인코딩

피쳐 값의 유형에 따라 새로운 피쳐를 추가해 고유 값에 해당하는 칼럼에만 1을 표시하고 나머지 칼럼에는 0을 표시

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

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

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

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

사이킷런에서 `OneHotEncoder` 클래스로 쉽게 변환 가능. `LabelEncoder`와 달리 모든 문자열 값이 숫자형으로 변환되어야 한다는 점, 입력 값으로 2차원 데이터가 필요하다는 점에 주의.

```
import pandas as pd

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

`get_dummies()` 를 이용해 더 쉽게 이용 가능. 이 경우, 문자열 카테고리 값을 숫자 형으로 변환할 필요가 없다.

피쳐 스케일링과 정규화



피쳐 스케일링(feature scaling)이란, 서로 다른 변수의 값 범위를 일정한 수준으로 맞추는 작업을 말한다. 대표적으로 표준화와 정규화가 있다.

표준화란, 데이터의 피쳐 각각이 평균이 0이고 분산이 1인 가우시안 정규 분포를 가진 값으로 변환하는 것을 말한다.

정규화란, 서로 다른 피쳐의 크기를 통일하기 위해 크기를 변환해주는 것을 말한다. 그러나 사이킷런의 전처리에서 제공하는 `Normalizer` 모듈과 일반적인 정규화는 약간의 차이가 있다. 사이킷런의 `Normalizer` 모듈은 선형대수에서의 정규화 개념으로 개별 벡터의 크기를 맞추기 위해 변환되는 것을 의미한다.

🔍 StandardScaler

표준화를 쉽게 지원하기 위한 클래스로, 개별 피쳐를 평균이 0이고 분산이 1인 값으로 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 변환돼 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('\n feature 들의 분산 값')
print(iris_df_scaled.var())
```

MinMaxScaler

데이터값을 0과 1사이의 범위 값으로 변환한다(음수 값이 있으면 -1에서 1값으로 변환한다). 데이터의 분포가 가우시안 분포가 아닌 경우 Min, Max Scale을 적용해볼 수 있다.

```
from sklearn.preprocessing import MinMaxScaler

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

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

학습 데이터와 테스트 데이터의 스케일링 변환 시 유의점

StandardScaler나 MinMaxScaler와 같은 Scaler 객체를 이용해 데이터의 스케일링 변환 시 fit(), transform(), fit_transform() 메소드를 이용한다. 일반적으로 fit()은 데이터 변환을 위한 기준 정보 설정을 적용하며 transform()은 이렇게 설정된 정보를 이용해 데이터를 변환한다. 그리고 fit_transform()은 fit()과 transform()을 한번에 적용한다.

머신러닝 모델은 학습 데이터를 기반으로 학습되기 때문에 반드시 테스트 데이터는 학습 데이터의 스케일링 기준에 따라야 한다. 따라서 테스트 데이터에 다시 fit()을 적용해서는 안 되며 **학습 데이터로 이미 fit()이 적용된 Scaler 객체를 이용해 transform()으로 변환해야 한다.**

가능하다면 전체 데이터의 스케일링 변환을 적용한 뒤 학습과 테스트 데이터로 분리하는 것이 바람직하다.

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

# test_array에 Scale 변환을 할 때는 반드시 fit()을 호출하지 않고 transform()만으로 변환해야 함
test_scaled = scaler.transform(test_array)
print('\n 원본 test_array 데이터: ', np.round(test_array.reshape(-1), 2))
print('Scale된 test_array 데이터: ', np.round(test_scaled.reshape(-1), 2))
```

fit_transform()을 적용할 때도 마찬가지로, 테스트 데이터에서 절대 사용해서는 안 된다.



3장. 평가

EURON 6기 초급세션 김수연

01. 정확도



정확도란 실제 데이터에서 예측 데이터가 얼마나 같은지를 판단하는 지표이다.

직관적으로 모델 예측 성능을 나타내는 평가 지표지만, **불균형한 레이블 값 분포에서 ML 모델의 성능을 판단하는 경우에는 적합하지 않다.**

```

from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.base import BaseEstimator
from sklearn.metrics import accuracy_score
import numpy as np
import pandas as pd

class MyFakeClassifier(BaseEstimator):
    def fit(self, X, y):
        pass
    # 입력값으로 들어오는 x 데이터 세트의 크기만큼 모두 0 값으로 만들어서 반환
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
# 사이킷런의 내장 데이터 세트인 load_digits()를 이용해 MNIST 데이터 로딩
digits = load_digits()

# digits 번호가 7이면 True이고 이를 astype(int)로 1로 변환, 7번이 아니면 False이고 0으로 변환
y = (digits.target == 7).astype(int)
X_train, X_test, y_train, y_test = train_test_split(digits.data, y, random_state=11)

# 불균형한 레이블 데이터 분포도 확인
print('레이블 테스트 세트 크기: ', y_test.shape)
print('테스트 세트 레이블 0과 1의 분포도')
print(pd.Series(y_test).value_counts())

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

```

위의 예제에서 불균형한 데이터 세트와 Dummy Classifier를 생성한 뒤 불균형한 데이터로 생성한 y_test의 데이터 분포도를 확인하고 MyFakeClassifier를 이용해 예측과 평가 수행하였다. 단순히 predict()의 결과를 np.zeros()로 모두 0 값으로 반환함에도 불구하고 테스트 데이터 세트에 수행한 예측 정확도가 90%이다.

이러한 정확도 평가 지표의 한계를 극복하기 위해 정확도는 여러 가지 분류 지표와 함께 적용되어야 한다.

02. 오차 행렬



오차행렬이란 이진 분류의 예측 오류가 얼마인지와 더불어 어떠한 유형의 예측 오류가 발생하고 있는지를 함께 나타내는 지표이다.

사이킷런은 오차행렬을 구하기 위해 `confusion_matrix()` API를 제공한다.

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

TP, TN, FP, FN 값은 Classifier 성능의 여러 면모를 판단할 수 있는 기반 정보를 제공하는데, 이 값을 조합해 Classifier의 성능을 측정할 수 있는 주요 지표인 정확도(Accuracy), 정밀도(Precision), 재현율(Recall) 값을 알 수 있다.

- TN: 예측값을 Negative 값 0으로 예측했고 실제값 역시 Negative 값 0
- FP: 예측값을 Positive 값 1로 예측했는데 실제값은 Negative 값 0
- FN: 예측값을 Negative 값 0으로 예측했는데 실제값은 Positive 값 1
- TP: 예측값을 Positive 값 1로 예측했는데 실제값 역시 Positive 값 1



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

정확도는 예측값과 실제 값이 얼마나 동일한가에 대한 비율로 결정되기 때문에 오차 행렬에서는 True에 해당하는 값인 TN과 TP에 좌우된다.

03. 정밀도와 재현율

정밀도와 재현율은 Positive 데이터 세트의 예측 성능에 더 초점을 맞춘 평가 지표이다.

정밀도는 예측을 Positive로 한 대상 중 예측과 실제 값이 Positive로 일치한 데이터의 비율을, 재현율은 실제 값이 Positive인 대상 중 예측과 실제 값이 Positive로 일치한 데이터의 비율을 뜻한다.



정밀도 = $TP/(FP+TP)$



재현율 = $TP/(FN+TP)$

정밀도와 재현율 지표 중 이진 분류 모델의 업무 특성에 따라서 특정 평가 지표가 더 중요한 지표로 간주될 수 있다. 재현율이 중요 지표인 경우는 실제 Positive 양성 데이터를 Negative로 잘못 판단하게 되면 업무상 큰 영향이 발생하는 경우이며, 정밀도가 상대적으로 더 중요한 지표인 경우는 실제 Negative 음성인 데이터가 예측을 Positive 양성으로 잘못 판단하게 되면 업무상 큰 영향이 발생하는 경우이다.

사이킷런은 정밀도 계산을 위해 `precision_score()` 를, 재현율 계산을 위해 `recall_score()` 를 API로 제공한다.

입력 파라미터	<code>predict()</code> 메서드와 동일하게 보통 테스트 피쳐 데이터 세트를 입력
반환 값	<p>개별 클래스의 예측 확률을 ndarray m x n (m: 입력 값의 레코드 수, n: 클래스 값 유형) 형태로 반환. 입력 테스트 데이터 세트의 표본 개수가 100개이고 예측 클래스 값 유형이 2개(이진 분류)라면 반환 값은 100 x 2 ndarray임.</p> <p>각 열은 개별 클래스의 예측 확률입니다. 이진 분류에서 첫 번째 칼럼은 0 Negative의 확률, 두 번째 칼럼은 1 Positive의 확률입니다.</p>

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# 원본 데이터를 재로딩, 데이터 가공, 학습 데이터/테스트 데이터 분할
titanic_df = pd.read_csv('./titanic_train.csv')
y_titanic_df = titanic_df['Survived']
X_titanic_df = titanic_df.drop('Survived', axis=1)
X_titanic_df = transform_features(X_titanic_df)
```

```
X_train, X_test, y_train, y_test = train_test_split(X_titanic_df, y_titanic_df, test_size=0.20, random_

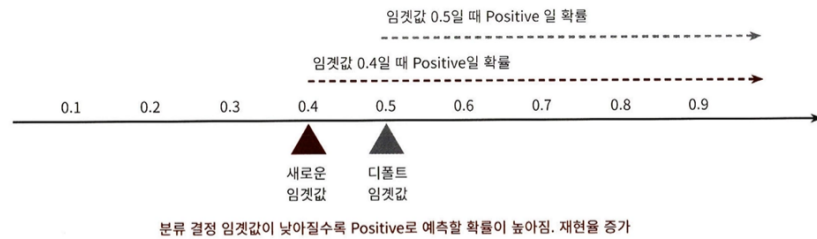
lr_clf = LogisticRegression()

lr_clf.fit(X_train, y_train)
pred = lr_clf.predict(X_test)
get_clf_eval(y_test, pred)
```

정밀도에 비해 재현율이 낮게 나왔다. 재현율 또는 정밀도를 좀 더 강화할 방법은 무엇일까?

정밀도/재현율 트레이드오프 : 정밀도와 재현율은 상호 보완적인 평가 지표이기 때문에 한쪽을 강제로 높이면 다른 하나의 수치는 떨어지기 쉽다.

사이킷런은 개별 데이터별로 예측 확률을 반환하는 메서드인 `predict_proba()` 를 제공한다. 해당 메서드는 학습이 완료된 사이킷런 Classifier 객체에서 호출이 가능하며 테스트 피쳐 데이터 세트를 파라미터로 입력해주면 테스트 피쳐 레코드의 개별 클래스 예측 확률을 반환한다.



Positive 예측값이 많아지면 상대적으로 재현율 값이 높아집니다. 양성 예측을 많이 하다 보니 실제 양성을 음성으로 예측하는 횟수가 상대적으로 줄어들기 때문입니다.

[임계값 0.5일 때 오차 행렬]

TN	FP
108	10
FN	TP
14	47

[임계값 0.4일 때 오차 행렬]

TN	FP
97	21
FN	TP
11	50

사이킷런은 임계값 변화에 따른 평가 지표 값을 알 수 있는 `precision_recall_curve()` API를 제공한다.

입력 파라미터	y_true: 실제 클래스값 배열 (배열 크기는 [데이터 건수])
	probas_pred: Positive 클래스의 예측 확률 배열 (배열 크기는 [데이터 건수])
반환 값	정밀도: 임계값별 정밀도 값을 배열로 반환
	재현율: 임계값별 재현율 값을 배열로 반환

```
from sklearn.metrics import precision_recall_curve

# 레이블 값이 1일 때의 예측 확률을 추출
pred_proba_class1 = lr_clf.predict_proba(X_test)[: , 1]

# 실제값 데이터 세트와 레이블 값이 1일 때의 예측 확률을 precision_recall_curve 인자로 입력
precisions, recalls, thresholds = precision_recall_curve(y_test, pred_proba_class1)
print('반환된 분류 결정 임계값 배열의 Shape: ', thresholds.shape)

# 반환된 임계값 배열 로우가 147건이므로 샘플로 10건만 추출하되, 임계값을 15 step으로 추출
thr_index = np.arange(0, thresholds.shape[0], 15)
print('샘플 추출을 위한 임계값 배열의 index 10개: ', thr_index)
print('샘플용 10개의 임계값: ', np.round(thresholds[thr_index], 2))

# 15 step 단위로 추출된 임계값에 따른 정밀도와 재현율 값
```

```
print('샘플 임계값별 정밀도: ', np.round(precisions[thr_index], 3))
print('샘플 임계값별 재현율: ', np.round(recalls[thr_index], 3))
```

04. F1 스코어



F1 스코어란, 정밀도와 재현율을 결합한 지표이다. F1 스코어는 정밀도와 재현율이 어느 한 쪽으로 치우치지 않는 수치를 나타낼 때 상대적으로 높은 값을 가진다.

사이킷런은 F1 스코어를 구하기 위해 `f1_score()` 라는 API를 제공한다.

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

05. ROC 곡선과 AUC

ROC 곡선과 이에 기반한 AUC 스코어는 이진 분류의 예측 성능 측정에서 중요하게 사용되는 지표이다.



ROC 곡선은 FPR(False Positive Rate)이 변할 때 TPR(True Positive Rate)이 어떻게 변하는지를 나타내는 곡선이다. FPR을 X 축으로, TPR을 Y 축으로 잡으면 FPR의 변화에 따른 TPR의 변화가 곡선 형태로 나타난다.



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

- 민감도(TPR)은 실제값 Positive(양성)가 정확히 예측되어야 하는 수준을 나타낸다.



$\text{민감도} = TP / (FN + TP)$

- 특이성(TNR)은 실제값 Negative(음성)가 정확히 예측되어야 하는 수준을 나타낸다.



$\text{특이성} = TN / (FP + TN)$

사이킷런은 ROC 곡선을 구하기 위해 `roc_curve()` API를 제공한다.

입력 파라미터	<code>y_true</code> : 실제 클래스 값 array (array shape = [데이터 건수])
	<code>y_score</code> : <code>predict_proba()</code> 의 반환 값 array에서 Positive 칼럼의 예측 확률이 보통 사용됨. array, shape = [n_samples]
반환 값	<code>fpr</code> : fpr 값을 array로 반환
	<code>tpr</code> : tpr 값을 array로 반환
	<code>thresholds</code> : threshold 값 array

```
from sklearn.metrics import roc_curve

# 레이블 값이 1일 때의 예측 확률을 추출
pred_proba_class1 = lr_clf.predict_proba(X_test)[: , 1]

fprs, tprs, thresholds = roc_curve(y_test, pred_proba_class1)
# 반환된 임계값 배열 로우가 47건이므로 샘플로 10건만 추출하되, 임계값을 5 step으로 추출
# thresholds[0]은 max(예측확률)+1로 임의 설정됨. 이를 제외하기 위해 np.arange는 1부터 시작
```

```
thr_index = np.arange(1, thresholds.shape[0], 5)
print('샘플 추출을 위한 임계값 배열의 Index 10개: ', thr_index)
print('샘플용 10개의 임계값: ', np.round(thresholds[thr_index], 2))
```

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

```
def roc_curve_plot(y_test, pred_proba_c1):
    # 임계값에 따른 FPR, TPR 값을 반환받음
    fprs, tprs, thresholds = roc_curve(y_test, pred_proba_c1)
    # ROC 곡선을 그래프 곡선으로 그림
    plt.plot(fprs, tprs, label='ROC')
    # 가운데 대각선 직선을 그림
    plt.plot([0, 1], [0, 1], 'k--', label='Random')

    #FPR X 축의 Scale을 0.1단위로 변경 X, Y축 명 설정 등
    start, end = plt.xlim()
    plt.xticks(np.round(np.arange(start, end, 0.1), 2))
    plt.xlim(0, 1); plt.ylim(0, 1)
    plt.xlabel('FPR(1-Sensitivity)'); plt.ylabel('TPR(Recall)')
    plt.legend()
```

```
roc_curve_plot(y_test, pred_proba[:, 1])
```

```
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score
from sklearn.metrics import recall_score, f1_score, roc_auc_score
import numpy as np
```

```
print(confusion_matrix(y_test, pred))
print("정확도: ", np.round(accuracy_score(y_test, pred), 4))
print("정밀도: ", np.round(precision_score(y_test, pred), 4))
print("재현도: ", np.round(recall_score(y_test, pred), 4))
```

