파머완 2장 - 사이킷런

Course	파머완 예습과제
Lesson Date	@2024년 3월 1일
Status	In Progress
Type	Lesson

01. 사이킷런 소개와 특징

- 머신러닝을 위한 매우 다양한 알고리즘과 개발을 위한 편리한 프레임워크와 API 제공
- 오랜 기간 실전 환경에서 검증되었으며, 매우 많은 환경에서 사용되는 라이브러리

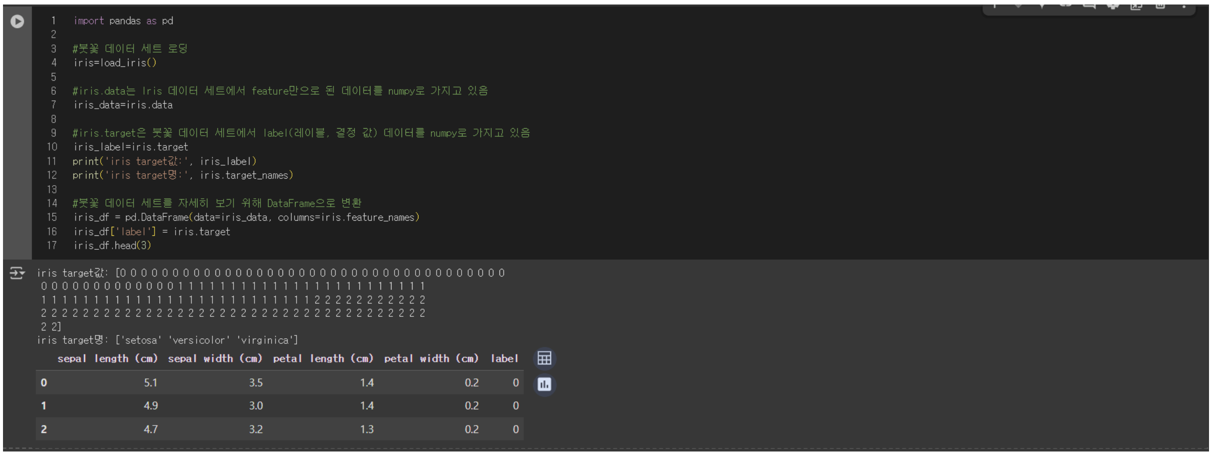
```
pip install -U scikit-learn
```

```
import sklearn

print(sklearn.__version__)
```

02. 첫 번째 머신러닝 만들어보기 - 붓꽃 품종 예측하기

- 붓꽃 데이터 세트로 붓꽃의 품종을 분류 (Classification)
- 꽃잎의 길이와 너비, 꽃받침의 길이와 너비 피처(Feature) 기반으로 꽃의 품종을 예측
- 분류 (Classification) - 지도학습 (Supervised Learning)
- 지도 학습 : 학습을 위한 다양한 Feature와 분류 결정값인 Label 데이터로 모델을 학습한 뒤, 별도의 테스트 데이터 세트에서 미지의 Lable을 예측함 → 명확한 정답이 주어진 데이터를 학습 후, 미지의 정답을 예측 !
- 사이킷런 패키지 내의 모듈명 : sklearn으로 시작

- 붓꽃 데이터 로딩 : load_iris(), ML 알고리즘 : 의사결정 트리 (Decision Tree) 알고리즘 → DecisionTreeClassifier
- train_test_split() : 데이터 세트를 학습 데이터와 테스트 데이터로 분리 → 학습된 데이터로 학습된 모델이 얼마나 뛰어난 성능을 가지는지 평가



```
import pandas as pd

#붓꽃 데이터 세트 로딩
iris=load_iris()

#iris.data는 Iris 데이터 세트에서 feature만으로 된 데이터를 numpy로 가지고 있음
iris_data=iris.data

#iris.target은 붓꽃 데이터 세트에서 label(레이블, 결정 값) 데이터를 numpy로 가지고 있음
iris_label=iris.target
print('iris target값:', iris_label)
print('iris target명:', iris.target_names)

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

```
iris target값: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 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]
iris target명: ['setosa' 'versicolor' 'virginica']
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	label
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0

- Feature : sepal length, sepal width, petal length, petal width
- Label : 0 (Setosa 품종), 1(versicolor 품종), 2(Virginica 품종)
- 학습 데이터와 테스트 데이터를 test_size 파라미터 입력값의 비율로 쉽게 분할

```
X_train, X_test, y_train, y_test = train_test_split(iris_data, iris_label, test_size=0.
```

- [파라미터] iris_data : 피처 데이터 세트, iris_label : Label 데이터 세트, test_size =0.2 : 전체 데이터 세트 중 테스트 데이터 세트의 비율, random_state : 난수 발생 값 (random_state 미지정 시 수행할 때마다 다른 학습/테스트 용 데이터를 만들 수 있음)
-

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

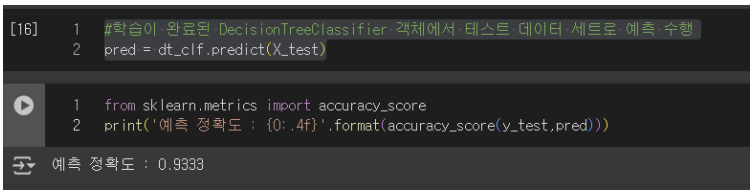
- DecisionTreeClassifier를 객체로 생성 (사이킷런의 의사 결정 트리 클래스)

```
#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)
```

- predict 메서드에 피처 데이터 세트를 입력해서 호출 → 학습된 모델 기반에서 테스트 데이터 세트에 대한 예측값을 반환
- 예측 결과를 기반으로 의사 결정 트리 기반의 DecisionTreeClassifier의 예측 성능 평가 : accuracy_score() 이용 → 정확도 측정

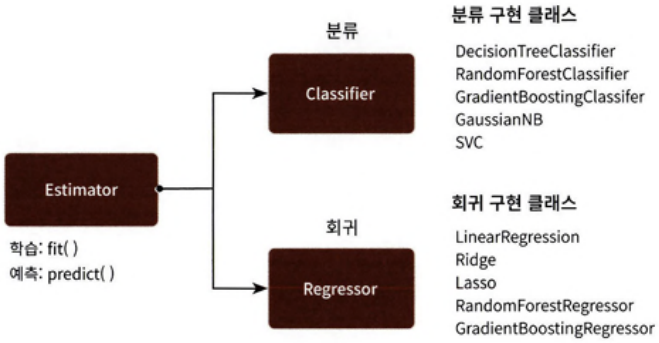


[붓꽃 데이터 세트 예측 프로세스 정리]

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 분류를 예측
4. 평가 : 예측된 결괏값과 테스트 데이터의 실제 결괏값을 비교해 ML 모델 성능 평가

03. 사이킷런의 기반 프레임워크 익히기

- 사이킷런은 ML 모델 학습을 위해서 fit()을 제공, 학습된 모델의 예측을 위해 predict() 제공
- Classifier : 분류 알고리즘을 구현한 클래스
- Regressor : 회귀 알고리즘을 구현한 클래스
- → Estimator 클래스 : Classifier + Regressor



- 사이킷런의 주요 모듈

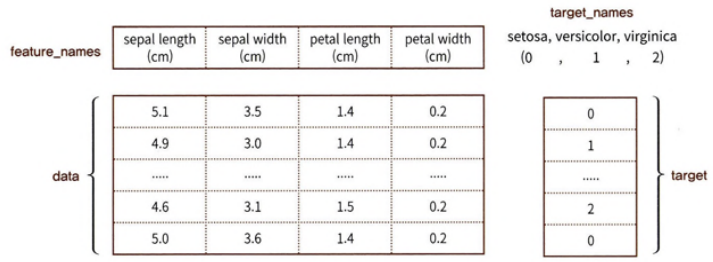
예제 데이터	sklearn.datasets	사이킷런에 내장되어 예제로 제공하는 데이터 세트
피처 처리	sklearn.processing	데이터 전처리에 필요한 다양한 가공 기능 제공
피처 처리	sklearn.feature_selection	알고리즘에 큰 영향을 미치는 피처를 우선순위대로 셀렉션 작업을 수행하는 다양한 기능 제공
피처 처리	sklearn.feature_extraction	텍스트 데이터나 이미지 데이터의 벡터화된 피처를 사용하는 데에 사용됨
피처 처리 & 차원 축소	sklearn.decomposition	차원 축소와 관련한 알고리즘 지원
데이터 분리, 검증 & 파라미터 튜닝	sklearn.model_selection	교차 검증을 위한 학습용/테스트용 분리, Grid search
평가	sklearn.metrics	분류, 회귀, 클러스터링, Pairwise에 대한 다양한 성능 측정 방법 제공
ML 알고리즘	sklearn.ensemble	앙상블 알고리즘 (랜덤 포레스트, 에이다 부스트, 그래디언트 부스트)
ML 알고리즘	sklearn.linear_model	회귀 관련 알고리즘 (선형 회귀, Ridge, Lasso, 로지스틱 회귀 등)
ML 알고리즘	sklearn.naive_bayes	나이브 베이즈 알고리즘
ML 알고리즘	sklearn.neighbors	최근접 이웃 알고리즘 제공 (K-NN)
ML 알고리즘	sklearn.svm	서포트 벡터 머신 알고리즘
ML 알고리즘	sklearn.tree	의사결정트리 알고리즘
ML 알고리즘	sklearn.cluster	비지도 클러스터링 알고리즘 (K-means, 계층형, DBSCAN 등)

- 내장된 예제 데이터 세트 : 일반적으로 딕셔너리 형태

- fetch 계열의 명령은 패키지에 처음부터 저장 X, 인터넷에서 내려받아 서브 디렉터리(scikit_learn_data)에 저장 후 추후 불러들이는 데이터

data	feature의 데이터셋	넘파이 (ndarray) 배열 타입
target	분류 : Label, 회귀 : 숫자 결괏값 데이터 세트	넘파이 (ndarray) 배열 타입
target_names	개별 label의 이름	넘파이 배열 or 파이썬 리스트 (list) 타입
feature_names	피처의 이름	넘파이 배열 or 파이썬 리스트 (list) 타입
DESCR	데이터 세트에 대한 설명, 각 feature에 대한 설명	스트링 타입

- 피터의 데이터 값 반환 : 내장 데이터 세트 API 호출 후, Key 값 지정
- load_iris() API의 반환 결과는 sklearn.utils.Bunch 클래스
- 데이터 키 - 피처들의 데이터 값
- [피처 데이터값 추출] 데이터 세트가 딕셔너리 형태 → 데이터세트.data 이용



```
[21]  1  #load_iris( )가 반환하는 객체의 키인 feature_names, target_name, data, target이 가리키는 값 출력
      2  print('₩n feature_names의 type:', type(iris_data.feature_names))
      3  print('feature_names의 shape:', len(iris_data.feature_names))
      4  print(iris_data.feature_names)
      5
      6  print('₩n target_names의 type:', type(iris_data.target_names))
      7  print('target_names의 shape:', len(iris_data.target_names))
      8  print(iris_data.target_names)
      9
     10  print('₩n data의 type:', type(iris_data.data))
     11  print('data의 shape:', iris_data.data.shape)
     12  print(iris_data['data'])
     13
     14  print('₩n target의 type:', type(iris_data.target))
     15  print('target의 shape:', iris_data.target.shape)
     16  print(iris_data.target)
```

```
 feature_names의 type: <class 'list'>
feature_names의 shape: 4
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']

 target_names의 type: <class 'numpy.ndarray'>
target_names의 shape: 3
['setosa' 'versicolor' 'virginica']

 data의 type: <class 'numpy.ndarray'>
data의 shape: (150, 4)
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
```

→ load_iris( )가 반환하는 객체의 키인 feature_names, target_name, data, target이 가리키는 값을 출력함. (아래부분 생략)

04. Model Selection 모듈 소개

학습/테스트 데이터 세트 분리 - train_test_split()

- 이미 학습된 데이터 세트를 기반으로 예측 시 정확도가 100% → NO
- 예측을 수행하는 데이터 세트는 학습을 수행한 학습용 데이터가 아니라, 전용의 테스트 데이터 세트여야 함
- train_test_split() : 원본 데이터 세트에서 학습 및 테스트 데이터 세트를 쉽게 분리 가능

```
1  from sklearn.tree import DecisionTreeClassifier
2  from sklearn.metrics import accuracy_score
3  from sklearn.datasets import load_iris
4  from sklearn.model_selection import train_test_split
5
6  dt_clf = DecisionTreeClassifier( )
7  iris_data = load_iris()
8
9  X_train, X_test, y_train, y_test = train_test_split(iris_data.data, iris_data.target, test_size=0.3, random_state=1
```

```
[38]
1  dt_clf.fit(X_train, y_train)
2  pred = dt_clf.predict(X_test)
3  print('예측 정확도: {0:.4f}'.format(accuracy_score(y_test, pred)))
```

```
예측 정확도: 0.9556
```

교차 검증

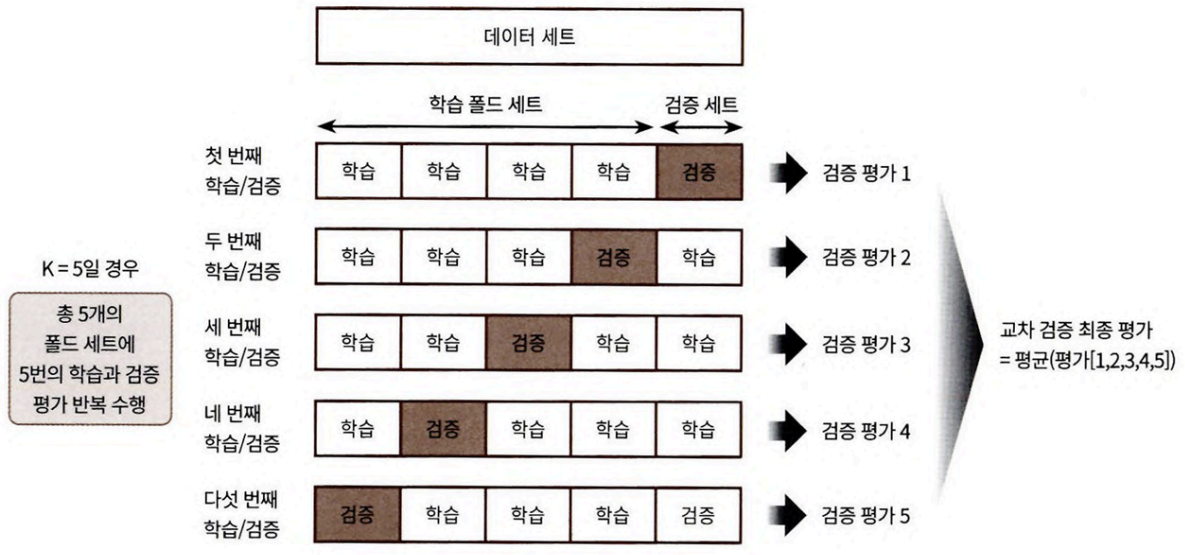
- 과적합(Overfitting)에 취약할 수 있음
- Overfitting이란 ?

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

- 교차 검증 : 데이터 편중을 막기 위해 별도의 여러 세트로 구성된 학습 데이터 세트와 검증 데이터 세트에서 학습과 평가를 수행하는 것
- 대부분의 ML 모델의 성능 평가는 교차 검증 기반으로 1차 평가 → 최종적으로 테스트 데이터 세트에 적용해 평가

K 폴드 교차 검증

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



- 데이터 세트를 K등분 → 마지막 등분 하나를 검증 데이터로 설정, 나머지는 학습 데이터 세트로 설정 → 학습 데이터 세트에서 학습 수행, 검증 데이터 세트에서 평가 수행 → 첫 번째 평가 수행 후 두번째 반복에서 평가 수행 (학습 데이터와 검증 데이터를 변경) , … → 마지막 K번째까지 학습과 검증 수행 → 예측 평가를 구하여 평균하고, 이를 K 폴드 평가 결과로 반영
- 사이킷런 : KFold, StratifiedKFold

```python
#K 폴드 교차 검증
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
```

→ KFold 객체를 생성함 : 이후 생성된 KFold 객체의 split()을 호출해 전체 붓꽃 데이터를 5개의 폴드 데이터 세트로 분리

```python
n_iter = 0

#KFold 객체의 split()를 호출하면 폴드별 학습용, 검증용 테스트의 로우 인덱스를 array로 반환
for train_index, test_index in kfold.split(features):
    #kfold.split()으로 반환된 인덱스를 이용해 학습용, 검증용 테스트 데이터 추출
    X_train, X_test = features[train_index], features[test_index]
    y_train, y_test = label[train_index], label[test_index]
    #학습 및 예측
    dt_clf.fit(X_train, y_train)
    pred = dt_clf.predict(X_test)
    n_iter += 1
    #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 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 분포도를 가진 레이블 데이터 집합을 위한 K 폴드 방식

- 원본 데이터의 레이블 분포를 고려한 뒤, 분포와 동일하게 학습과 검증 데이터 세트를 분배함
- 레이블 데이터 분포도에 따라 학습/검증 데이터를 나누기 때문에 split() 메서드에 인자로 피처 데이터 세트뿐만 아니라 레이블 데이터 세트도 반드시 필요 !

```
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]
    train_size = X_test.shape[0]
    print('₩n#{0} 교차 검증 정확도 :{1}, 학습 데이터 크기: {2}, 검증 데이터 크기: {3}'
        .format(n_iter, accuracy, train_size, test_size))
    print('#{0} 검증 세트 인덱스:{1}'.format(n_iter, test_index))
    cv_accuracy.append(accuracy)

# 교차 검증별 정확도 및 평균 정확도 계산
print('₩n## 교차 검증별 정확도:', np.round(cv_accuracy, 4))
print('## 평균 검증 정확도:', np.round(np.mean(cv_accuracy), 4))
```

```
#1 교차 검증 정확도 :0.98, 학습 데이터 크기: 50, 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  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: 50, 검증 데이터 크기: 30
#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: 50, 검증 데이터 크기: 30
#3 검증 세트 인덱스:[ 34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  83  84
  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 133 134 135
 136 137 138 139 140 141 142 143 144 145 146 147 148 149]

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

→ 3개의 Stratified K 폴드로 교차 검증한 결과 평균 검증 정확도가 약 96.67%로 측정

- 왜곡된 레이블 데이터 세트에서는 반드시 Stratified K 폴드를 이용해 교차 검증
- 회귀 : Stratified K 폴드 지원 X

교차 검증을 보다 간편하게 : cross_val_score()

- 교차 검증을 좀 더 편리하게 수행할 수 있게 해주는 API

```
cross_val_score(estimator, Xz y=None, scoring=None, cv=Nonez n_jobs=1, ve
pre_dispatch='2*n_jobs')
```

- estimator - Classifier or Regressor, X - 피처 데이터 세트, y - 레이블 데이터 세트, scoring - 예측 성능 평가 지표, cv - 교차 검증 폴드 수
- cross_val_score() API는 내부에서 Estimator 학습(fit), 예측(predict), 평가(evaluation) 시킴 → 간단한 교차 검증 가능

- 비슷한 API : cross_validate()→ 여러 개의 평가 지표 반환 가능

GridSearchCV - 교차 검증과 최적 하이퍼 파라미터 튜닝을 한 번에

- 하이퍼 파라미터 : 머신러닝 알고리즘 구성 요소 → 조정 : 알고리즘의 예측 성능 개선

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

- GridSearchCV : 교차 검증을 기반으로 하이퍼 파라미터의 최적 값을 찾아줌
- 주요 파라미터 : estimator, param_grid, scoring, cv, refit
- rank_test_socre → 예측 성능 순위
- mean_test_score → 개별 하이퍼 파라미터별로 CV의 폴딩 테스트 세트에 대해 총 수행한 평가 평균값

05. 데이터 전처리

- 결손값(NaN, NULL)은 허용 X → 다른 값으로 반환 필요
- 피처의 평균값으로 대체, 피처 드롭 등
- 텍스트형 피처 : 벡터화 OR 삭제

데이터 인코딩 - 레이블 인코딩, 원-핫 인코딩

[레이블 인코딩]

- Label encoding : 카테고리 피처를 코드형 숫자 값으로 변환 (EX-TV： 1, 냉장고: 2, 전자레인지: 3, 컴퓨터: 4, 선풍기: 5, 믹서: 6과 같은 숫자형 값으로 변환)
- LabelEncoder 클래스로 구현 : fit() & transform() 호출
- 문자열 값 → 숫자형 카테고리 값
- 숫자값의 대소 관계로 인한 특성 → ML 알고리즘 적용 X !

[원-핫 인코딩]

- 레이블 인코딩의 문제를 해결하기 위한 인코딩 방식
- 피처 값의 유형에 따라 새로운 피처 추가 → 고유 값에 해당하는 칼럼에만 1 표시, 나머지 칼럼에는 0 표시

원본 데이터

상품 분류
TV
냉장고
전자레인지
컴퓨터
선풍기
선풍기
믹서
믹서

원-핫 인코딩

상품분류_ TV	상품분류_ 냉장고	상품분류_ 믹서	상품분류_ 선풍기	상품분류_ 전자레인지	상품분류_ 컴퓨터
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

- OneHotEncoder 클래스로 변환 가능 (2차원 데이터의 입력값 필요, 변환값이 희소행렬 → toarray 이용해서 밀집 행렬로 변환 필요)

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

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

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

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

#OneHotEncoder로 변환힌 결과는 희소행렬이므로 toarray()를 이용해 밀집 행렬로 변환
print('원-핫 인코딩 데이터')
print(oh_labels.toarray())
print('원-핫 인코딩 데이터 차원')
print(oh_labels.shape)
```

```
원-핫 인코딩 데이터
[[1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]]
원-핫 인코딩 데이터 차원
(8, 6)
```

- get_dummies() : 원-핫 인코딩을 더 쉽게 지원하는 API → 문자열 카테고리 값을 숫자형으로 변환할 필요 없이 바로 변환

```python
import pandas as pd

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

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

피처 스케일링과 정규화

- 피처 스케일링 : 서로 다른 변수의 값 범위를 일정한 수준으로 맞추는 작업 (표준화, 정규화)
- 표준화 - 데이터의 피처 각각이 평균이 0이고 분산이 1 인 가우시안 정규 분포를 가진 값으로 변환하는 것
- 정규화 - 서로 다른 피처의 크기를 통일하기 위해 크기를 변환
- Noramalizer 모듈

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

StandardScaler

- 표준화를 지원하기 위한 클래스 : 개별 피처를 평균이 0, 분산이 1인 값으로 변환
- StandardScaler 객체를 생성 → fit()과 transform() 메서드에 변환 대상 피처 데이터 세트를 입력하고 호출

MinMaxScaler

- 데이터값을 0과 1사이의 범위 값으로 변환 (음수 : -1에서 1로 변환)

- 데이터 분포가 가우시안 분포가 아닐 때 적용

학습 데이터와 테스트 데이터의 스케일링 변환 시 유의점

- 학습 데이터 세트와 테스트 데이터 세트에 fit()과 transform()을 적용할 때 주의해야 함
- 학습 데이터로 fit()이 적용된 스케일링 기준 정보를 그대로 테스트 데이터에 적용해야 함
- if not : 학습 데이터와 테스트 데이터의 스케일링이 맞지 X
- 테스트 데이터에 다시 fit()을 적용해서는 안 됨, 학습 데이터로 이미 fit()이 적용된 Scaler 객체를 이용해 transform()으로 변환

07. 정리

- 사이킷런 : 다양한 머신러닝 기능을 제공하는 패키지, API → 머신러닝 애플리케이션을 쉽게 구현
- 머신러닝 애플리케이션 : 전처리 작업, 데이트 세트 분리 작업, 머신러닝 알고리즘 적용 → 모델 학습
- 머신러닝 모델은 학습 데이터 세트로 학습 → 별도의 테스트 데이터 세트로 평가되어야 함
- 교차 검증 : KFold, StratifiedKFold, cross_val_score
- GridSearchCV
- 파이썬 기반에서 머신러닝을 경험하는 필수 패키지

파머완 3장 - 평가

Course	파머완 예습과제
Lesson Date	@2024년 3월 13일
Status	In Progress
Type	Lesson

분류의 성능 평가 지표 - 정확도, 오차행렬, 정밀도, 재현율, F1 스코어, ROC AUC

분류 : 이진 분류 ↔ 멀티 분류

01. 정확도 (Accuracy)

정확도 : 예측 결과가 동일한 데이터 건수 / 전체 예측 데이터 건수

→ 직관적으로 모델 예측 성능을 나타내는 평가 지표

- BaseEstiamator 클래스 상속 → 학습X, 단순한 Classifier를 생성

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

#원본 데이터를 재로딩, 데이터 가공, 학습 데이터 / 테스트 데이터 분할
from google.colab import drive
drive.mount('/content/drive/')

titanic_df = pd.read_csv("/content/drive/MyDrive/titanic_train.csv")
y_titanic_df = titanic_df['Survived']
X_titanic_df = titanic_df.drop('Survived', axis=1)
```

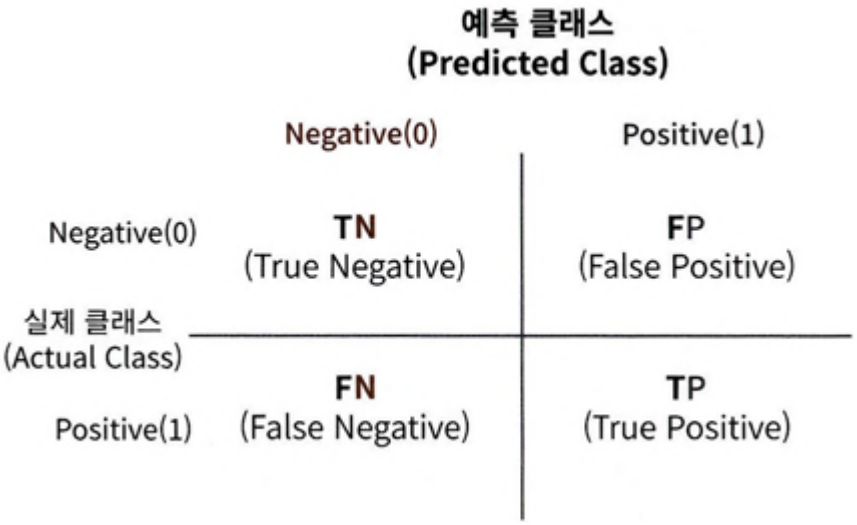
```
X_titanic_df = transform_features(X_titanic_df)
X_train, X_test, y_train, y_test=train_test_split(X_titanic_df, y_titanic_df, test_siz
```

[Output] Dummy Classifier의 정확도는: 0.7877

- MNIST 데이터 세트 (0~9까지의 숫자 이미지 픽셀 정보)
- 데이터 분포도가 균일하지 않은 경우 높은 수치가 나타날 수 있다는 것이 정확도 평가 지표의 맹점
- 예시 : 불균형한 데이터 세트 , Dummy Classifier 생성 → 예측 및 평가

02. 오차 행렬

- 오차행렬 : 학습된 분류 모델이 예측을 수행하면서 얼마나 헷갈리고 있는지 함께 보여줌 (→ 이진 분류의 예측 오류가 얼마인지, 어떤 유형의 예측 오류가 발생하고 있는지)



- Negative, Positive → TN, FP, FN, TP : 분류 모델 예측 성능의 오류가 어떤 모습으로 발생하는지
- TN : True Negative (0으로 예측, 실제로도 negative인 0)
- FP : False Positive (1로 예측, 실제는 negative인 0)
- FN : False Negative (0으로 예측, 실제는 positive인 1)
- TP : True Positive (1로 예측, 실제로도 positive인 1)
- confusion_matrix() API
- 정확도 = 예측 결과와 실제 값이 동일한 건수 / 전체 데이터 수 = (TN+TP)/(TN+FP+FN+TP)

- 불균형한 이진 데이터 분류 세트 → Negative로 예측 정확도가 높아지는 경향 발생

03. 정밀도와 재현율

- 정밀도(양성 예측도) = TP / (FP+TP) → 예측을 positive로 한 대상 중 예측과 실제값이 positive로 일치한 데이터의 비율
- 재현율(민감도) = TP / (FN+TP) → 실제 값이 positive인 대상 중 예측과 실제 값이 positive로 일치한 데이터의 비율
- 분류 모델의 업무 특성에 따라서 특정 평가 지표가 더 중요한 지표로 간주될 수 있음
- 정밀도 계산 - precision_score(), 재현율 계산 - recall_score()

정밀도/재현율 트레이드오프 : 정밀도와 재현율은 상호 보완적인 평가 지표 → 한 쪽을 강제로 높이면 다른 쪽은 떨어지기 쉬움

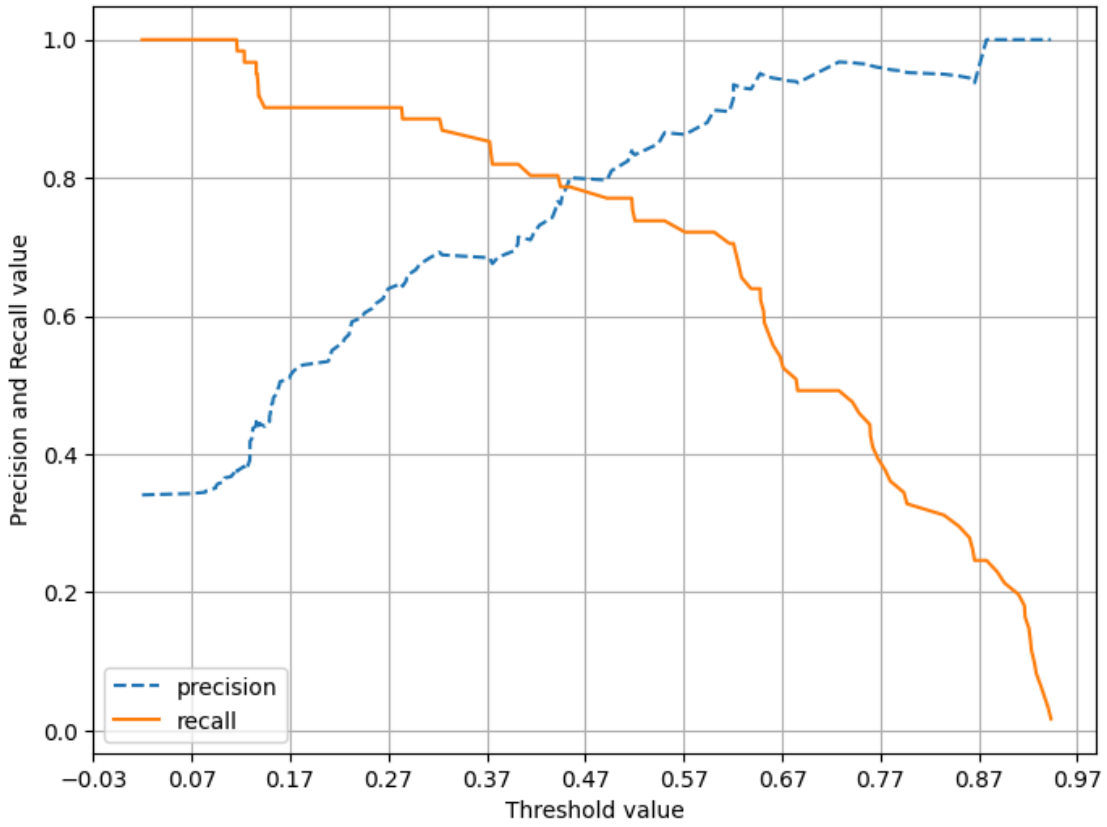
- predict_proba(): 테스트 피처 데이터 세트를 파라미터로 입력 → 테스트 피처 레코드의 개별 클래스 예측 확률을 반환

```
from sklearn.preprocessing import Binarizer

#Binarizer의 threshold의 설정값. 분류 결정 임곗값임
custom_threshold = 0.5

#predict_proba() 반환값의 두번째 칼럼, 즉 positive 클래서 칼럼 하나만 추출해 Binari
pred_proba_1 = pred_proba[:,1].reshape(-1,1)

binarizer = Binarizer(threshold = custom_threshold).fit(pred_proba_1)
custom_predict = binarizer.transform(pred_proba_1)

get_clf_eval(y_test, custom_predict)
```

- 분류 결정 임곗값을 낮추면 재현율 up, 정밀도 down

precision_recall_curve()

- 입력 파라미터 : y_true, probas_pred / 반환 값 : 정밀도, 재현



→ precision_recall_curve() API를 이용한 정밀도와 재현율 곡선 시각화

정밀도와 재현율의 맹점

- 정밀도가 100%가 되는 방법 : 확실한 기준인 경우만 Positive, 나머진 Negative
- 재현율이 100%가 되는 방법 : 모든 환자를 Positive로 예측

04. F1 스코어

F1 스코어

- 정밀도와 재현율을 결합한 지표 → 정밀도와 재현율이 어느 한쪽으로 치우치지 않는 수치를 나타낼 때 상대적으로 높은 값을 가짐

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

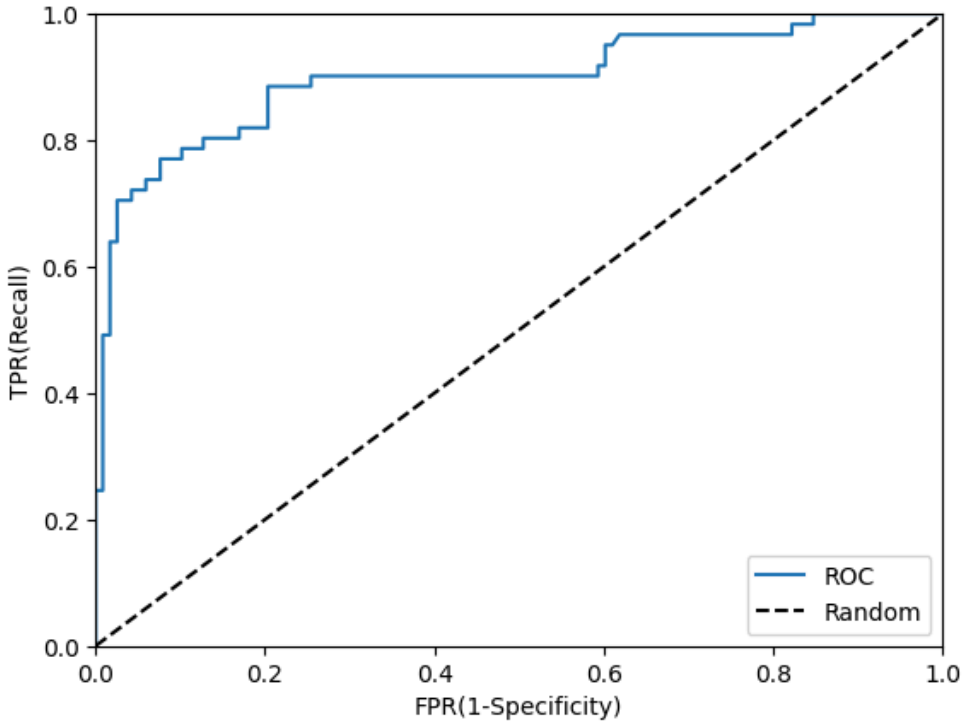
- f1_score() : 사이킷런의 F1 스코어를 구하기 위한 API

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

```
def get_clf_eval(y_test,pred):
    confusion = confusion_matrix(y_test, pred)
    accuracy = accuracy_score(y_test, pred)
    precision = precision_score(y_test, pred)
    recall = recall_score(y_test, pred)
    #F1 스코어 추가
    f1 = f1_score(y_test, pred)
    print('오차 행렬')
    print(confusion)
    #f1 score print 추가
    print('정확도:{0:.4f}, 정밀도:{1:.4f}, 재현율:{2:.4f}, F1:{3:.4f}'.format(accuracy,

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

05. ROC 곡선과 AUC

- ROC 곡선(Receiver Operation Characteristic Curve) : FPR이 변할 때 TPR이 어떻게 변하는지를 나타내는 곡선 (FPR - False Positive Rate, TPR = True Positive Rate, 재현율) → FPR을 0부터 1까지 변경하며 TPR의 변화값을 구함
- TNR (특이성, True Negative Rate) : 실제값 Negative가 정확히 예측되어야 하는 수준
- TNR = TN / (FP + TN)
- FPR = FP / (FP + TN) = 1 - TNR = 1 - 특이성
- ROC 곡선이 가운데 직선에 가까울수록 성능이 떨어지고, 멀어질수록 성능이 뛰어남
- roc_curve() API : 입력 파라미터 - y_true, y_score / 반환값 - fpr, tpr, thresholds



→ FPR 변화에 따른 TPR 변화를 ROC 곡선으로 시각화

- AUC 값 : ROC 곡선 밑의 면적을 구한 것 → 1에 가까울수록 좋은 수
- ROC AUC → 예측 확률값을 기반으로 계산됨 : get_clf_eval(y_test, pred=None, pred_proba = None)로 함수형 변경

07. 정리

- 성능 평가 지표 : 정확도, 오차 행렬, 정밀도, 재현율, F1 스코어, ROC-AUC
- 오차 행렬 : 실제 클래스 (Negative, Positive) & 예측 클래스 (True, False) → TN, FP, FN, TP
- 정밀도 & 재현율 : Positive 데이터 세트 예측 성능에 초첨을 맞춘 평가 지표, 임곗값인 Threshold를 조정해 정밀도 / 재현율 수치 높일 수 있음
- F1 스코어 : 정밀도와 재현율 결합한 평가 지표, 어느 한 쪽으로 치우치지 않을 때 높은 지표값
- ROC-AUC : 이진 분류의 성능 평가 지표, AUC는 ROC 곡선 밑면적