파이썬 머신러닝 완벽 가이드

▼ Chap1. 파이썬 기반의 머신러닝과 생태계 이해

> 💡 머신러닝 알고리즘이란?
> → 데이터를 기반으로 통계적인 신뢰도를 강화하고 예측 오류를 최소화하기 위한 다양한 수학적 기법을 적용해 데이터 내의 패턴을 스스로 인지하고 신뢰도 있는 예측 결과를 도출해 내는 것

- 머신러닝의 분류 (지도학습 vs 비지도학습)

1. 지도학습: 분류(Classification), 회귀(Regression), 추천 시스템, 텍스트 분석, NLP 등
2. 비지도학습: 클러스터링, 차원 축소, 강화학습 등

넘파이(Numpy)

→ 파이썬에서 선형대수 기반의 프로그램을 쉽게 만들 수 있도록 지원하는 패키지

```
import numpy as np

array = np.array([[1,2,3],
                  [4,5,6],
                  [7,8,9]])
print('array type:', type(array))
print('array 형태:', array.shape)
print('array 차원:', array.ndim)
```

```
<output>
array type: <class 'numpy.ndarray'>
array 형태: (3, 3)
array 차원: 2
```

3행 3열의 numpy array 생성 가능 (2차원 데이터)

- ndarray내의 데이터 타입은 두가지가 공존할 수 없음! (int, float 공존해있으면 int → float로 바뀜)

```
#astype()을 통해 데이터 타입 변경(메모리 절약, float -> int로 바꾸자)
array_float = np.array([1.,2.,3.])
print(array_float.dtype)
array_int = array_float.astype('int32')
print(array_int.dtype)
```

```
<output>
float64
int32
```

- arange, zeros, ones, reshape

```
a = np.arange(10)
print('array1:\n',a)
b = np.zeros((3,2))
print('array2:\n',b)
c = np.ones((3,2))
print('array3:\n',c)

a_reshape = a.reshape(2,5)
print('array1_reshape:\n',a_reshape)
```

```
<output>
array1:
 [0 1 2 3 4 5 6 7 8 9]
array2:
 [[0. 0.]
 [0. 0.]
 [0. 0.]]
array3:
 [[1. 1.]
 [1. 1.]
 [1. 1.]]
array1_reshape:
 [[0 1 2 3 4]
 [5 6 7 8 9]]
```

- 인덱싱(Indexing)

→ ndarray 내의 일부 데이터 세트나 특정 데이터만을 선택할 수 있도록 함.

```
#단일값 추출
array1 = np.arange(1,10)
print(array1)
print('0번째 값:',array1[0]) #0번째 값 추출
print('뒤에서 2번째 값:', array1[-2])
array1[-2] = -8 #값 수정
print(array1)
print('--------------2차원 array-------------')
array2 = np.arange(1,10).reshape(3,3)
print(array2)
print('0행 0열:', array2[0,0])
print('1행 2열:', array2[1,2])
```

```
<output>
[1 2 3 4 5 6 7 8 9]
0번째 값: 1
뒤에서 2번째 값: 8
[ 1  2  3  4  5  6  7 -8  9]
--------------2차원 array-------------
[[1 2 3]
 [4 5 6]
 [7 8 9]]
0행 0열: 1
1행 2열: 6
```

```
#슬라이싱
array1 = np.arange(1,10)
print(array1[:])
print(array1[0:3])
print(array1[3:])
print('--------------2차원 array-------------')
array2 = np.arange(1,10).reshape(3,3)
print(array2[:])
print('2번째 열만',array2[:,2])
print(array2[1:,1:])
```

```
<output>
[1 2 3 4 5 6 7 8 9]
[1 2 3]
[4 5 6 7 8 9]
--------------2차원 array-------------
[[1 2 3]
 [4 5 6]
 [7 8 9]]
2번째 열만 [3 6 9]
[[5 6]
 [8 9]]
```

```
#불린 인덱싱 (조건 필터링과 검색을 동시에 할 수 있음)
array1 = np.arange(1,10)
array1[array1>5]
```

`array([6, 7, 8, 9])`

- 행렬의 정렬 (sort, argsort)

sort → 행렬을 정렬, argsort → 정렬된 행렬의 인덱스를 반환

```
#sort
array1 = [3,1,9,5]
print(np.sort(array1)) #원본은 변경x
print(array1)
print('----------------------------------------')
print(array1.sort()) #원본 변경O
print(array1)
print('내림차순 정렬:', np.sort(array1)[::-1])
print('--------------2차원 array-------------')
array2 = np.array([[8,12],
                   [7,1]])
print('행 방향으로 정렬\n', np.sort(array2, axis=0))
print('열 방향으로 정렬\n', np.sort(array2, axis=1))
```

```
<output>
[1 3 5 9]
[3, 1, 9, 5]
----------------------------------------
None
[1, 3, 5, 9]
내림차순 정렬: [9 5 3 1]
--------------2차원 array-------------
행 방향으로 정렬
 [[ 7  1]
 [ 8 12]]
열 방향으로 정렬
 [[ 8 12]
 [ 1  7]]
```

```
#argsort
array1 = [3,1,5,0]
print(np.argsort(array1)) #[0,1,3,5]
print(np.argsort(array1)[::-1]) #[5,3,1,0]
```

```
<output>
[3 1 0 2]
[2 0 1 3]
```

- 행렬의 내적(np.dot)과 전치행렬(np.transpose)

전치행렬은 원 행렬에서 행과 열 위치를 교환한 우너소로 구성한 행렬

```
a = np.arange(1,7).reshape(2,3)
b = np.arange(7,13).reshape(3,2)
c = np.arange(1,5).reshape(2,2)
print('두 행렬의 내적\n',np.dot(a,b))
print('전치행렬\n',np.transpose(c))
```

```
<output>
두 행렬의 내적
 [[ 58  64]
 [139 154]]
전치행렬
 [[1 3]
 [2 4]]
```

판다스(Pandas)

→ 데이터 핸들링에 특화된 프레임워크, 핵심 객체는 DataFrame

- 데이터 불러오고 살펴보기

```
import pandas as pd
titanic_df = pd.read_csv('titanic.csv')
titanic_df.head(3)
titanic_df.shape
```

891행, 12열에 해당하는 데이터프레임

describe(), info() 메서드를 통해 각 열의 특징들을 알아 볼 수 있음

info() → 총 데이터 건수, 데이터 타입, null 건수 알 수 있음

describe() → 오직 숫자형 column의 분포도만 조사(분포도, 평균값, 최댓값, 최솟값 등)

```
titanic_df['Pclass'].value_counts()
```

```
<output>
3    491
1    216
2    184
Name: Pclass, dtype: int64
```

→ DataFrame의 [] 연산자 내부에 column 명을 입력하면 해당 column의 Series 객체 반환

→ value_counts()는 지정된 칼럼의 데이터값 건수 반환(많은 건수 순서로 정렬)

- DataFrame ↔ ndarray, 리스트, 딕셔너리

1. ndarray, 리스트, 딕셔너리 → DataFrame

DataFrame은 특별하게 칼럼명을 갖고 있어, DataFrame으로 변환을 해줄 때 칼럼명을 지정해주어야 한다.(지정 안해주면 자동으로 할당)

2차원 이하의 데이터들만 DataFrame으로 변환될 수 있음.

```
col_name = ['col1', 'col2', 'col3']
array = np.array([[1,2,3],
                  [4,5,6]])
#ndarray를 dataframe으로
df_array = pd.DataFrame(array, columns=col_name)
print(df_array)
```

```
<output>
  col1  col2  col3
0     1     2     3
1     4     5     6
```

딕셔너리의 경우에는 딕셔너리의 키(key)가 칼럼명으로, 값(value)이 데이터로 변환된다.

```
dict_ = {'col1':[1,11], 'col2':[2,22]}
df_dict = pd.DataFrame(dict_)
print(df_dict)
```

```
<output>
  col1  col2
0     1     2
1    11    22
```

2. DataFrame → ndarray, 리스트, 딕셔너리

```
#dataframe을 ndarray로
array_df = df_dict.values
print(type(array_df))
#dataframe을 리스트로
list_df = df_dict.values.tolist()
print(type(list_df))
#dataframe을 딕셔너리로
dict_df = df_dict.to_dict('list')
print(type(dict_df))
```

```
<class 'numpy.ndarray'>
<class 'list'>
<class 'dict'>
```

- DataFrame의 칼럼 생성, 수정 및 삭제

생성 및 수정은 [ ] 연산자를 통해 쉽게 할 수 있음

```
#칼럼 데이터 세트 생성
titanic_df['Age_0'] = 0
#기존 칼럼을 이용해 새로운 칼럼 생성
titanic_df['Family_No'] = titanic_df['SibSp'] + titanic_df['Parch'] + 1
#칼럼 데이터 세트 수정
titanic_df['Age_0'] = titanic_df['Age_0'] + 100
```

DataFrame에서 데이터의 삭제는 drop() 메서드를 이용한다.

⚠️ drop 옵션 중 axis=0(행 방향), axis=1(열 방향) 구분 주의 → 칼럼을 삭제할 땐 axis=1

```
titanic_drop_df = titanic_df.drop(['Age_0', 'Family_No'], axis=1)
titanic_drop_df.head(3)
```

→ 원본 DataFrame은 유지하고 드롭된 DataFrame을 새롭게 객체 변수로 받고 싶다면 inplace=False로 설정(디폴트), 원본에 드롭된 결과를 적용하고 싶을 땐 inplace=True

> 1. DataFrame 바로 뒤의 [ ] 연산자는 넘파이의 [ ]나 Series의 [ ]와 다르다.
> 2. DataFrame 바로 뒤의 [ ] 내 입력 값을 칼럼명을 지정해 칼럼 지정 연산에 사용하거나 불린 인덱스 용도로만 사용해야 함.

- iloc[ ]와 loc[ ]

1. iloc[ ]: 위치 기반 인덱싱만 허용하는 연산자
2. loc[ ]: 명칭 기반 인덱싱만 허용하는 연산자(행 위치에는 dataframe 인덱스 값을, 열 위치에는 칼럼 명을 입력해줌 ex. data_df.loc['one','Name']

- DataFrame, Series의 정렬 → sort_values()

  주요 파라미터는 by(특정 칼럼에 대한 정렬 수행), ascending(오름차순 내림차순 선택), inplace

```
#Pclass, Name을 기준으로 내림차순 정렬
titanic_sorted = titanic_df.sort_values(by=['Pclass', 'Name'], ascending=False)
titanic_sorted.head(3)
```

```
#모든 칼럼의 해당 aggregation 적용
titanic_df.count()
#지정한 칼럼만 aggregation 적용
titanic_df[['Age','Fare']].mean()
```

- groupby() 적용

  → 칼럼을 입력값으로 넣으면 해당 칼럼으로 group화됨

  → DataFrame에 groupby()를 호출해 반환된 결과에 aggregation 함수를 호출하면 groupby() 대상 칼럼을 제외한 모든 칼럼에 해당 aggregation 함수를 적용함

```
titanic_df.groupby('Pclass')[['PassengerId','Survived']].count()
```

  groupby()로 묶어주기 → 출력할 칼럼 지정해주기 → aggregation 함수 지정

  딕셔너리를 통해 각 컬럼에 대해 각기 다른 aggregation 함수를 지정해 줄 수 있다.

```
agg_format = {'Age':'max', 'SibSp':'sum', 'Fare':'mean'}
titanic_df.groupby('Pclass').agg(agg_format)
```

- 결손 데이터 처리하기

1. isna( ).sum( )로 결손 데이터 여부 확인
2. fillna( )로 결손 데이터 대체하기

```
titanic_df['Cabin'] = titanic_df['Cabin'].fillna('C000')
titanic_df['Age'] = titanic_df['Age'].fillna(titanic_df['Age'].mean())
titanic_df['Embarked'] = titanic_df['Embarked'].fillna('S')
titanic_df.isna().sum()
```

▼ Chap2. 사이킷런으로 시작하는 머신러닝

사이킷런

→ 가장 파이썬스러운 API 제공

→ 머신러닝을 위한 매우 다양한 알고리즘과 개발을 위한 편리함 프레임워크

→ 매우 많은 환경에서 사용되는 성숙한 라이브러리

첫번째 머신러닝 - 붓꽃 품종 예측

꽃잎의 길이, 너비, 꼴받침의 길이, 너비 feature를 기반으로 꽃의 품종 예측(Classification)

> 💡 지도학습: 학습을 위한 다양한 피처와 분류 결정값인 Label 데이터로 모델을 학습한 뒤 별도의 테스트 데이터 세트에서 미지의 레이블 예측
> → 명확한 정답이 주어진 데이터를 학습한 뒤 미지의 정답을 예측하는 것

```
import sklearn
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
import pandas as pd

#붓꽃 데이터셋 로딩
iris = load_iris()
iris_data = iris.data
iris_label = iris.target

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

→ feature에는 sepal length, sepal width, petal length, petal width가 있음.

→ Label에서 0은 Setosa, 1은 Versicolor, 2는 Virginica

머신러닝 알고리즘은 의사결정트리 사용

데이터셋 분리 → train_test_split( ) 사용

```
#학습데이터(80%), 테스트 데이터(20%) 분리
X_train, X_test, y_train, y_test = train_test_split(iris_data, iris_label, test_size=0.2, random_state=11)
#DecisionTreeClassifier 객체 생성
dt_clf = DecisionTreeClassifier(random_state=11)
#학습 수행
dt_clf.fit(X_train, y_train) #학습 데이터 기반으로 학습 완료
#학습이 완료된 DecisionTreeClassifier 객체에서 테스트 데이터 세트로 예측 수행
pred = dt_clf.predict(X_test)

from sklearn.metrics import accuracy_score
print('예측 정확도:', accuracy_score(y_test,pred))
```

```
예측 정확도: 0.9333333333333333
```

1. 데이터 세트 분리(학습 데이터, 테스트 데이터로 분리)

   → 첫번째 파라미터는 피처 데이터 세트, 두번째 파라미터는 Label 데이터 세트

   → test_size = 는 전체 데이터 세트 중 테스트 데이터 세트의 비율

   → random_size는 호출할 때마다 같은 데이터 세트를 생성하기 위해 지정

   → shuffle=True는 디폴트값(데이터를 분산시켜 좀 더 효율적인 데이터세트 만들기)

2. 모델 학습(학습 데이터를 기반으로 머신러닝 알고리즘을 적용해 모델 학습)

   → 의사결정트리 객체 생성(dt_clf)

   → fit( ) 메서드를 통해 학습용 피쳐 데이터(X_train)와 결정값 데이터 세트(y_train)를 입력해 호출함

3. 예측 수행(학습된 머신러닝 모델을 이용해 테스트 데이터의 분류를 예측)

   → 학습 데이터가 아닌 **반드시 테스트 데이터 이용**!

   → predict( ) 메서드에 테스트용 피쳐 데이터를 입력하면 학습된 모델 기반에서 테스트 데이터 세트에 대한 예측값 반환

4. 평가(예측된 결과값과 테스트 데이터의 실제 결과값을 비교해 머신러닝 모델 성능을 평가함)

   → 여러가지 성능 평가 방법이 있음 (여기서는 정확도 측정)

   → 첫번째 파라미터로 실제 Label 데이터 세트, 두번째 파라미터는 예측 Label 데이터 세트를 입력

사이킷런의 기반 프레임워크 익히기

- fit( ), predict( ) 메서드

  → ML **모델 학습**을 위해 **fit( )**, 학습된 **모델 예측**을 위해 **predict( )** 메서드 제공

  → Estimator 클래스에는 Classification(분류), Regressor(회귀) 존재

  → 분류 구현 클래스: DecisionTreeClassifier, RandomForestClassifier, GradientBoostingClassifier, GaussianNB, SVC 등등

  → 회귀 구현 클래스: LinearRegression, Ridge, Lasso, RandomForestRegressor, GradientBoostingRegressor

교차 검증

데이터의 수가 적으면 학습 데이터, 테스트 데이터로 나눠서 진행해도 overfitting 위험 有

→ **교차 검증** 이용! (본고사를 치르기 전에 모의고사를 여러 번 보는 것)

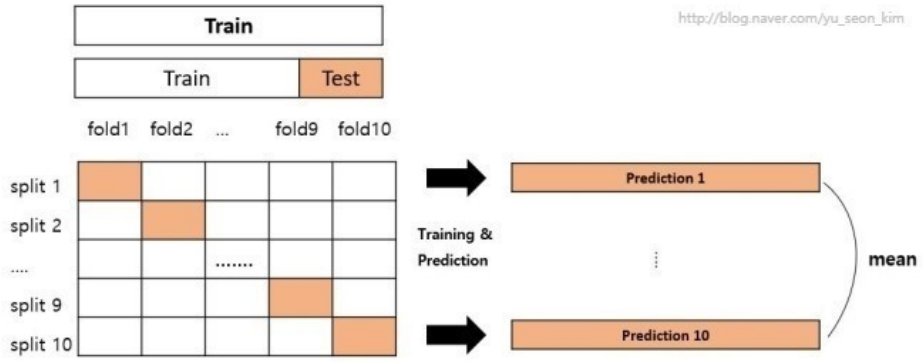
→ 교차 검증에서 많은 학습과 검증 세트에서 알고리즘 학습과 평가 수행 가능

(학습 데이터를 학습 데이터와 검증 데이터세트로 또 다시 나눔)

- K 폴드 교차 검증

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

→ K가 5라고 가정하면, 5개의 폴드된 데이터 세트를 학습과 검증을 위한 데이터 세트로 변경하며 5번 평가를 수행한 뒤 이 5개의 평가를 평균한 결과를 갖고 예측 성능 평가




```
#K-FOLD 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=123)

#5개의 폴드 세트로 분리하는 KFold 객체와 폴드 세트별 정확도를 담을 리스트 객체 생성
kfold = KFold(n_splits=5)
cv_accuracy = []
#전체 붓꽃 데이터는 모두 150개, 따라서 학습용 데이터 세트는 이 중 4/5인 120개, 검증용 데이터 세트는 30개로 분할

n_iter = 0
#KFold 객체의 split()을 호출하면 폴드 별 학습용, 검증용 테스트의 row index를 array로 반환
for train_index, test_index in kfold.split(features):
    #kfold.split으로 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
    #반복할때마다 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_iter, '교차 검증 정확도:',accuracy, '학습 데이터 크기',train_size, '검증 데이터 크기', test_size)
    print(n_iter, '검증 세트 인덱스:',test_index)
    cv_accuracy.append(accuracy)

#개별 iteration별 정확도를 합하여 평균 정확도 계산
print('평균 검증 정확도:', np.mean(cv_accuracy))
```

폴드 세트 설정 → for loop에서 반복으로 학습 및 테스트 데이터의 인덱스 추출 → 반복적으로 학습과 예측 수행 후 예측 성능 반환

- Stratified K 폴드

→ 불균형한 분포도를 가진 Label 데이터 집합을 위한 K 폴드 방식(특정 label 값이 특이하게 많거나 적어 값의 분포가 한쪽으로 치우치는 것) ex. 대출 사기 데이터 예측

→ 원본 데이터의 Label 분포를 먼저 고려한 뒤 이 분포와 동일하게 학습과 검증 데이터 세트를 분배함.

→ split( ) 메서드에 인자로 피쳐 데이터 세트 뿐만 아니라 **Label 데이터 세트도 필요**함!

```
#Stratified KFold
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('교차검증:',n_iter)
    print('학습 레이블 데이터 분포: \n', label_train.value_counts())
    print('검증 레이블 데이터 분포: \n', label_test.value_counts())
```

→ 기존 KFold와 다르게 학습 레이블과 검증 레이블 데이터 값의 분포도가 동일하게 할당됐음

```
dt_clf = DecisionTreeClassifier(random_state=156)

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

for train_index, test_index in skfold.split(features, label):
    #split으로 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
    #반복할때마다 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_iter, '교차 검증 정확도:',accuracy, '학습 데이터 크기',train_size, '검증 데이터 크기', test_size)
    print(n_iter, '검증 세트 인덱스:',test_index)
    cv_accuracy.append(accuracy)

#개별 iteration별 정확도를 합하여 평균 정확도 계산
print('평균 검증 정확도:', np.mean(cv_accuracy))
```

```
평균 검증 정확도: 0.9666666666666667
```

⚠️일반적으로 Classification에서의 교차 검증은 KFold가 아니라 Stratified KFold로 분할돼야 함.

⚠️Regression에서는 Stratified KFold가 지원되지 않음.(회귀의 결정값은 연속된 숫자값이기에)

- 교차 검증을 보다 간편하게 → cross_val_score( )

→ 사이킷런에서 제공하는 교차검증을 좀 더 편리하게 수행할 수 있게 하는 API

```
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 - Classification 또는 Regressor

X - 피쳐 데이터 세트, y - Label 데이터 세트

scoring - 예측 성능 평가 지표 기술, cv - 교차 검증 폴드 수

```
#cross_val_score() 사용하기
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=123)

data = iris_data.data
label = iris_data.target

#성능 지표는 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))
```

→ API 내부에서 Estimator를 학습(fit), 예측(predict), 평가(evaluation)시켜주므로 간단하게 교차검증 수행 가능

→ 비슷한 API인 cross_validate( )는 여러개의 평가 지표를 반환할 수 있음 (cross_val_score는 단 하나의 평가 지표만 가능)

GridSearchCV - 교차 검증과 최적 하이퍼 파라미터 튜닝을 한 번에!

→ 하이퍼 파라미터를 조정함으로써 알고리즘의 예측 성능을 개선할 수 있다.

→ 사이킷런은 GridSearchCV API를 이용해 Classifier나 Regressor와 같은 알고리즘에 사용되는 하이퍼 파라미터를 순차적으로 입력하며 편리하게 최적의 파라밑를 도출할 수 있는 방안을 제공함.

→ 교차 검증을 기반으로 하이퍼 파라미터의 최적 값을 찾게 해준다. (수행 시간이 상대적으로 오래 걸림)

주요 파라미터

-estimator, param_grid(키 + 리스트 값을 갖는 딕셔너리, estimator의 튜닝을 위해 파라미터명과 사용될 여러 파라미터 값을 지정해줌)

-scoring(예측 성능을 측정할 평가 방법 지정, 별도의 성능 평가 지표 함수도 지정할 수 있음)

-cv(교차 검증을 위해 분할되는 학습, 테스트 세트의 개수 지정)

-refit(디폴트가 True → 가장 최적의 하이퍼 파라미터를 찾은 뒤 입력된 estimator 객체를 해당 하이퍼파라미터로 재학습시킴)

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
import pandas as pd

#데이터를 로딩하고 학습 데이터와 테스트 데이터 분리
iris_data = 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=123)
dtree = DecisionTreeClassifier()

#파라미터를 딕셔너리 형태로 설정
parameters = {'max_depth':[1,2,3], 'min_samples_split':[2,3]}
#param_grid의 하이퍼 파라미터를 3개의 train, test set fold로 나누어 테스트 수행 설정
#refit=True가 디폴트 값. 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 DataFrame으로 변환
scores_df = pd.DataFrame(grid_dtree.cv_results_)
scores_df[['params', 'mean_test_score', 'rank_test_score', 'split0_test_score', 'split1_test_score', 'split2_test_score']]
```

→ params 칼럼에는 수행할 때마다 적용된 개별 하이퍼 파라미터 값을 나타낸다.

→ rank_test_score는 하이퍼 파라미터 별로 성능이 좋은 score 순위를 나타낸다. (1이 가장 뛰어난 순위)

→ mean_test_score는 개별 하이퍼 파라미터별로 CV의 폴딩 테스트 세트에 대해 총 수행한 평가 평균값

```
print('GridSearchCV 최적 파라미터:', grid_dtree.best_params_)
print('GridSearchCV 최고 정확도:', grid_dtree.best_score_)
```

데이터 전처리

→ 결손값은 허용되지 않음, 고정된 다른 값으로 변환해야 함

→ Null 값이 대부분인 피쳐값이라면 해당 피쳐는 드롭하는 것이 좋음

→ 사이킷런의 머신러닝 알고리즘은 문자열 값을 입력값으로 허용하지 않음. 따라서 모든 문자열 값을 인코딩해서 숫자 형으로 변환해야 함

데이터 인코딩

레이블 인코딩(Label encoding) : 카테고리 피쳐를 코드형 숫자 값으로 변환하는 것

원-핫 인코딩(One Hot encoding) : 피쳐 값의 유형에 따라 새로운 피쳐를 추가해 고유 값에 해당하는 칼럼에만 1을 표시하고 나머지 칼럼에는 0을 표시하는 방식

- 레이블 인코딩

→ LabelEncoder 클래스로 구현

```
#레이블 인코딩
from sklearn.preprocessing import LabelEncoder

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

#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_)
```

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

- 원-핫 인코딩(One-Hot Encoding)

→ 행 형태로 되어 있는 피쳐에 열 형태로 차원을 변환한 뒤, 고유 값에 해당하는 칼럼에만 1을 표시하고 나머지 칼럼에는 0을 표시

⚠️OneHotEncoder로 변환되기 전에 모든 문자열 값이 숫자형 값으로 변환돼야 함

⚠️입력값으로 2차원 데이터가 필요하다.

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

items = ['TV', '냉장고', '전자레인지', '컴퓨터', '선풍기', '선풍기']
#먼저 숫자 값으로 변환을 위해 LabelEncoder로 변환
encoder = LabelEncoder()
encoder.fit(items)
labels = encoder.transform(items)
#2차원 데이터로 변환
labels = labels.reshape(-1,1)

#OneHot Encoding
oh_encoder = OneHotEncoder()
oh_encoder.fit(labels)
oh_labels = oh_encoder.transform(labels)
print('One-Hot 인코딩 데이터')
print(oh_labels.toarray())
print('One-Hot 인코딩 데이터 차원')
print(oh_labels.shape)
```

💡 판다스에는 원-핫 인코딩을 더 쉽게 지원하는 API가 있음
→ **get_dummies( ) 이용!**

```
import pandas as pd

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

→ get_dummies( )를 이용하면 숫자형 값으로 변환 없이 바로 변환 가능하다.

피쳐 스케일링 & 정규화

표준화(Standardization) : 데이터의 피쳐 각각이 평균이 0이고 분산이 1인 가우시안 정규 분포를 가진 값으로 변환하는 것 ((x-mean(x))/std(x))

정규화(Normalization) : 서로 다른 피쳐의 크기를 통일하기 위해 크기를 변환해주는 개념 (다른 변수들을 모두 동일한 크기 단위로 비교하기 위해 0~1 사이의 값으로 변환하는 것)

((x-min(x))/(max(x)-min(x)))

- **StandardScaler**

→ 표준화를 쉽게 지원하기 위한 피쳐 스케일링 클래스

→ 개별 피쳐를 평균이 0이고 분산이 1인 값으로 변환(가우시안 분포)

→ SVM, 선형 회귀, 로지스틱 회귀는 데이터가 가우시안 분포를 가지고 있다고 가정하고 구현됐기 때문에 사전에 표준화를 적용해야 함

```
#StandardScaler
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
import pandas as pd

#데이터 세트 로드 후 데이터 프레임으로 변환
iris = load_iris()
iris_data = iris.data
iris_df = pd.DataFrame(data=iris_data, columns=iris.feature_names)

#StandardScaler 객체 생성
scaler = StandardScaler()
#StandardScaler로 데이터 세트 변환, fit()과 transform() 호출
scaler.fit(iris_df)
iris_scaled = scaler.transform(iris_df)

#transform() 했을 때 스케일 변환된 데이터 세트가 ndarray로 변환되어 이를 DataFrame으로 변환
iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_names)
print('feature들의 평균값:')
print(iris_df_scaled.mean())
print('feature들의 분산값:')
print(iris_df_scaled.var())
```

→ 모든 칼럼의 평균이 0에 가까운 값으로, 분산은 1에 가까운 값으로 변환 되었음

- **MinMaxScaler**

→ 데이터 값을 0~1 범위 값으로 변환(음수 값이 있으면 -1~1 범위 값으로 변환)

→ 데이터의 분포가 가우시안 분포가 아닐 경우에 적용해 볼 수 있음

```
#MinMaxScaler
from sklearn.preprocessing import MinMaxScaler
```

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

iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_names)
print('feature들의 최솟값:')
print(iris_df_scaled.min())
print('feature들의 최댓값:')
print(iris_df_scaled.max())
```

→ 모든 피처에 0~1 사이의 값으로 변환되는 스케일링이 적용됐음

⚠️Scaler 객체를 이용해 학습 데이터 세트로 fit()을 수행한 결과를 이용해 transform()변환을 적용해야 함

→ 학습 데이터로 fit()이 적용된 스케일링 기준 정보를 그대로 테스트 데이터에 적용해야 함!

→ 테스트 데이터로 다시 새로운 스케일링 기준 정보를 만들게 되면 학습 데이터와 테스트 데이터의 스케일링 기준 정보가 달라지기에 올바른 예측 결과 도출 불가능

→ **따라서 fit()은 학습 데이터에서 한번만 하면 되는 것!! (transform은 두번)**

(test에 scale 변환을 할 때는 반드시 fit( )을 호출하지 않고 transform( )만으로 변환해야 함)

💡 1. 가능하다면 전체 데이터의 스케일링 변환을 적용한 뒤 학습, 테스트 데이터로 분리
2. 여의치 않다면 테스트 데이터 변환 시에는 fit( )이나 fit_transform( )을 사용하지 않고 학습 데잍로 이미 fit( )된 Scaler 객체를 이용해 transform( )으로 변환

▼ Chap3. 평가

분류(Classification)의 성능 평가 지표

1. 정확도(Accuracy)
2. 오차행렬(Confusion Matrix)
3. 정밀도(Precision)
4. 재현율(Recall)
5. F1 스코어
6. ROC AUC

→ 모두 긍정/부정과 같은 2개의 결과값만 갖는 이진 분류에서 중요하게 강조하는 지표

정확도(Accuracy)

→ 실제 데이터에서 예측 데이터가 얼마나 같은지를 판단하는 지표

→ 예측결과가 동일한 데이터 건수 / 전체 예측 데이터 건수

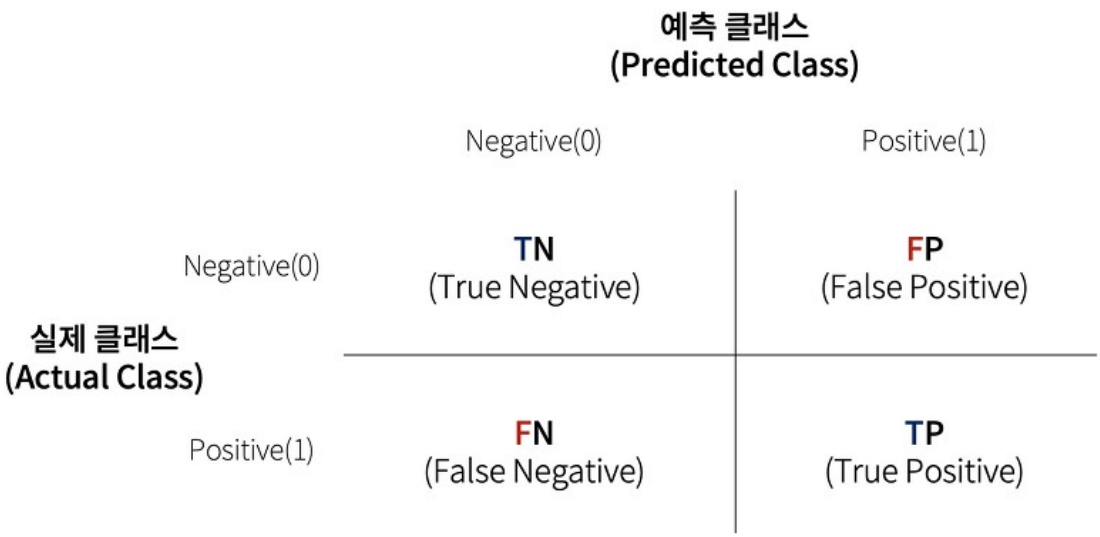
→ 이진 분류의 경우 데이터의 구성에 따라 머신러닝 **모델의 성능을 왜곡할 수 있기에 정확도만 갖고 성능을 평가하지 않음!**

→ 불균형한 레이블 값 분포에서 머신러닝 모델의 성능을 판단할 경우 적합한 평가 지표가 아님

→ 다른 여러가지 분류 지표와 함께 사용해야 한다!

오차 행렬(Confusion Matrix)

→ 이진 분류의 예측 오류가 얼마인지와 더불어 어떠한 유형의 예측 오류가 발생하고 있는지를 함께 나타내는 지표



TN, FP, FN, TP

앞문자 True/False → 예측값이 실제값과 같은가, 다른가

뒷문자 Negative/Positive → 예측 결과 값이 부정(0)인지 긍정(1)인지

사이킷런은 오차행렬을 구하기 위해 confusion_matrix( ) API를 제공함

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

TN은 array[0,0], FP는 array[0,1]로, FN은 array[1,0]으로. TP은 array[1,1]에 해당

→ 이 값들을 조합해 정확도(Accuracy), 정밀도(Precision), 재현율(Recall) 값을 알 수 있다.

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

→ 불균형한 데이터 세트에서 정확도보다 더 선호되는 평가 지표는 정밀도와 재현율임

정밀도와 재현율

→ Positive 데이터 세트의 예측 성능에 좀 더 초점을 맞춘 평가 지표

> 💡 **정밀도 = TP/(FP+TP)**
> **재현율 = TP/(FN+TP)**

정밀도: 예측을 Positive를 한 대상중에 예측값과 실제 값이 Positive로 일치한 데이터의 비율

재현율: 실제 값이 Positive인 대상 중에 예측과 실제 값이 Positive로 일치한 데이터의 비율(=민감도)

재현율이 중요 지표인 경우는 **실제 Positive 양성 데이터를 Negative로 잘못 판단하게 되면 업무상 큰 영향이 발생하는 경우** ex. 암 판단 모델, 보험사기 적발 모델

정밀도가 중요 지표인 경우는 **실제 Negative 음성 데이터를 Positive로 잘못 판단하게 되면 업무상 큰 영향이 발생하는 경우** ex. 스팸 메일 판단 모델

재현율과 정밀도 모두 TP를 높이는데 초점을 맞춤, 그리고 재현율은 FN을 낮추는데, 정밀도는 FP를 낮추는데 초점을 맞춤. → **재현율과 정밀도 모두 높은 수치를 얻는 것이 제일 바람직하다!**

BUT 정밀도와 재현율은 상호보완적인 평가 지표이기에 어느 한쪽을 강제로 높이면 다른 하나의 수치는 떨어지기 쉬움. (Trade-Off)

사이킷런은 정밀도 계산을 위해 precision_score()을, 재현율 계산을 위해 recall_score()을 API로 제공

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, confusion_matrix

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)
    print('오차행렬')
    print(confusion)
    print('정확도:',accuracy, '정밀도:',precision, '재현율:',precision)

#로지스틱 회귀 기반 타이타닉 생존자 예측
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

titanic_df = pd.read_csv('titanic.csv')
y_titanic_df = titanic_df['Survived']
X_titanic_df = titanic_df.drop('Survived', axis=1)
X_titanic_df = transform_features(X_titanic_df)
```

F1 스코어

→ 정밀도와 재현율을 결합한 지표

→ 정밀도와 재현율이 어느 한 쪽으로 **치우치지 않는 수치**를 나타낼 때 상대적으로 **높은 값을 가짐**

F1 = 2*(precision*recall)/(precision+recall)

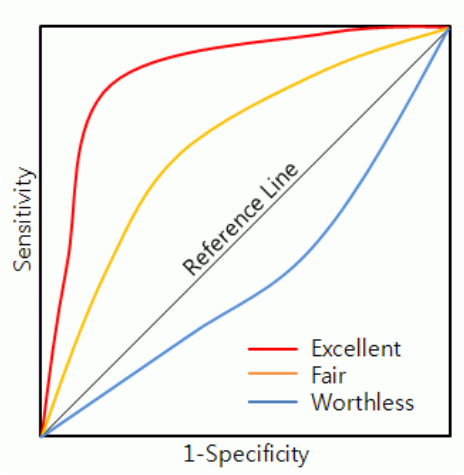
사이킷런에서는 F1 스코어를 구하기 위해 f1_score( )라는 API를 제공함.

```
from sklearn.metrics import f1_score
f1 = f1_score(y_test, pred)
print('F1 스코어:'. f1)
```

ROC 곡선과 AUC

→ ROC 곡선은 FPR(False Positive Rate)이 변할 때 TPR(True Positive Rate)이 어떻게 변하는지를 나타내는 곡선

→ FPR(1-TNR=1-특이성)을 X축으로, TPR(재현율, 민감도)을 Y축으로 둔다.



ROC곡선이 가운데 직선에 가까울수록 성능이 떨어지는 것이고, 멀어질수록 성능이 뛰어난 것