

# 1주차 연습과제

범위

파머완 1장-3장

## 넘파이 (Numpy)

```
import numpy as np
```

- **as np**를 추가하여 약어로 모듈을 표현해주는 것이 관례
- 넘파이의 기반 데이터 타입은 **ndarray**
- ndarray를 이용하여 넘파이에서 다차원 배열을 쉽게 생성하고 다양한 연산을 수행할 수 있다.

```
array1 = np.array([1, 2, 3])
print('array1 type:', type(array1))
print('array1 array 형태:', array1.shape)

array2 = np.array([[1, 2, 3],
                   [2, 3, 4]])
print('array2 type:', type(array2))
print('array2 array 형태:', array2.shape)

array3 = np.array([[1, 2, 3]])
print('array3 type:', type(array3))
print('array3 array 형태:', array3.shape)
```

```
array1 type: <class 'numpy.ndarray'>
array1 array 형태: (3,)
array2 type: <class 'numpy.ndarray'>
array2 array 형태: (2, 3)
array3 type: <class 'numpy.ndarray'>
array3 array 형태: (1, 3)
```

### np.array() 사용법

- ndarray로 변환을 원하는 객체를 인자로 입력하면 ndarray를 반환함.

### ndarray.shape

★ **ndarray.shape**는 ndarray의 **차원**과 **크기**를 튜플(Tuple) 형태로 나타내 준다.

- ex. [1,2,3]인 array1의 shape : (3,)
  - 1차원 array로 3개의 데이터를 가지고 있음을 뜻함.
- ex. [[1,2,3], [2,3,4]]인 array2의 shape : (2, 3)
  - 2차원 array로 2개의 로우와 3개의 칼럼으로 이루어짐을 뜻함.
- ex. [[1,2,3]]인 array3의 shape : (1, 3)
  - 2차원 array로 1개의 로우와 3개의 칼럼으로 이루어짐을 뜻함.
  - ★ array3와 array1은 동일한 데이터 건수를 가지고 있지만, **array1은 명확하게 1차원임을 (3,) 형태로 표현한 것이며, array3 역시 명확하게 로우와 칼럼으로 이루어진 2차원 데이터임을 (1,3)으로 표현한 것.**

➡ 이러한 차이를 명확히 이해하는 것이 중요. 데이터 값으로는 동일하나 차원이 달라서 오류가 발생하는 경우가 빈번하기 때문. 차원의 차수를 변환하는 방법을 알아야 이런 오류를 막을 수 있음. (reshape 함수)

- array() 함수의 인자로써 파이썬의 리스트 객체가 주로 사용됨.
- 리스트 []는 1차원이고, 리스트의 리스트 [[]]는 2차원과 같은 형태로 배열의 차원과 크기를 쉽게 표현할 수 있기 때문.

## ndarray의 데이터 타입

- ndarray내의 데이터값은 숫자 값, 문자열 값, 불 값 등이 모두 가능함.
- 숫자형의 경우 int형(8bit, 16bit, 32bit), float형(16bit, 32bit, 64bit, 128bit), 그리고 이보다 더 큰 숫자 값이나 정밀도를 위해 complex 타입도 제공함.

ndarray내의 데이터 타입은 **같은 데이터 타입만 가능함**.<br>

(= 한 개의 ndarray 객체에 int와 float가 함께 있을 수 X.)

★ ndarray 내의 데이터 타입은 **dtype** 속성으로 확인할 수 있음.

```
list1 = [1, 2, 3]
print(type(list1))
array1 = np.array(list1)
print(type(array1))
print(array1, array1.dtype)
```

```
<class 'list'>
<class 'numpy.ndarray'>
[1 2 3] int64
```

dtype 속성

- 리스트 자료형인 list1은 ndarray로 쉽게 변경 가능하며 변경된 ndarray 내의 값은 모두 int32형임.<br>
- ★ 만약 다른 데이터 유형이 섞여 있는 리스트를 ndarray로 변경한다면?
  - 데이터 크기가 더 큰 데이터 타입으로 형 변환을 일괄 적용함.

```
list2 = [1, 2, 'test']
array2 = np.array(list2)
print(array2, array2.dtype)

list3 = [1, 2, 3.0]
array3 = np.array(list3)
print(array3, array3.dtype)
```

```
['1' '2' 'test'] <U21
[1. 2. 3.] float64
```

데이터 크기가 더 큰 데이터 타입으로 변환된 모습

서로 다른 데이터 타입이 섞여 있는 경우 데이터 타입이 더 큰 데이터 타입으로 변환됨.

- array2는 int형이 유니코드 문자열 값으로 변환됨.
- array3는 int형과 float형이 섞여 있었으므로 int 1, 2가 모두 1. 2.인 float64형으로 변환됨.<br><br>
- ndarray 내 데이터값의 타입 변경 : **astype()** 메서드
  - 메모리를 더 절약해야 할 때 이용됨.
  - ex) int형으로 충분할 때 float 데이터 타입이라면 int형으로 바꿔 메모리 절약 가능.


```

▶ array_int = np.array([1, 2, 3])
array_float = array_int.astype('float64') #int -> float
print(array_float, array_float.dtype)

array_int1 = array_float.astype('int32') #float -> int
print(array_int1, array_int1.dtype)

array_float1 = np.array([1.1, 2.1, 3.1])
array_int2 = array_float1.astype('int32') #float -> int (소수점 이하 없어짐)
print(array_int2, array_int2.dtype)

```

 [1. 2. 3.] float64  
 [1 2 3] int32  
 [1 2 3] int32

## ndarray를 편리하게 생성하기 - arange, zeros, ones

- `arange()` 는 파이썬 표준 함수인 `range()` 와 유사한 기능을 가짐.
- 0부터 함수 인자 값 -1까지의 값을 순차적으로 ndarray의 데이터값으로 변환해 줌.
- `zeros()` 는 함수 인자로 튜플 형태의 shape 값을 입력하면 모든 값을 0으로 채운 해당 shape를 가진 ndarray를 반환함.
- `ones()` 는 함수 인자로 튜플 형태의 shape 값을 입력하면 모든 값을 1로 채운 해당 shape를 가진 ndarray를 반환함.
- 함수 인자로 dtype을 정해주지 않으면 default로 **float64**형의 데이터로 ndarray를 채움.

## ndarray의 차원과 크기를 변경하는 reshape()

- `reshape()` 메서드는 ndarray를 특정 차원 및 크기로 변환함.
- 변환을 원하는 크기를 함수 인자로 부여하면 됨.
- -1을 인자로 사용하면 원래 ndarray와 호환되는 새로운 shape로 변환해줌.
- array1은 1차원 ndarray로 0~9까지의 데이터를 갖고 있음.
- array1.reshape(-1, 5)는 **array1과 호환될 수 있는 2차원 ndarray로 변환하되, 고정된 5개의 칼럼에 맞는 로우를 자동으로 새롭게 생성해 변환하라는 의미.**
- 반대로 reshape(5, -1)은 **10개의 1차원 데이터와 호환될 수 있는 고정된 5개의 로우에 맞는 칼럼인 2를 가진 2차원 ndarray로 변환하라는 의미.**
- -1을 사용하더라도 호환될 수 없는 형태는 변환할 수 없음.(array4)
- -1 인자는 reshape(-1, 1) 형태로 자주 사용됨.
- `reshape(-1, 1)` 은 원본 ndarray가 어떤 형태라도 **2차원이고, 여러 개의 로우를 가지되 반드시 1개의 칼럼을 가진 ndarray로 변환됨을 보장함.**

## 넘파이의 ndarray의 데이터 세트 선택하기 - 인덱싱(Indexing)

- 특정한 데이터만 추출
- 슬라이싱(Slicing)
- 팬시 인덱싱(Fancy Indexing)
- 불린 인덱싱(Boolean Indexing)

```
[21] # 단일 값 추출
# 1부터 9까지의 1차원 ndarray 생성
array1 = np.arange(start=1, stop=10)
print('array1:', array1)
# index는 0부터 시작하므로 array1[2]는 3번째 index 위치의 데이터값을 의미
value = array1[2]
print('value:', value)
print(type(value))
```

```
array1: [1 2 3 4 5 6 7 8 9]
value: 3
<class 'numpy.int64'>
```

```
[24] # 인덱스에 마이너스 기호를 이용하면 맨 뒤에서부터 데이터를 추출할 수 있음
print('맨 뒤의 값:', array1[-1], ' 맨 뒤에서 두 번째 값:', array1[-2])
```

```
맨 뒤의 값: 9 맨 뒤에서 두 번째 값: 8
```

```
[25] # 단일 인덱스를 이용해 ndarray 내의 데이터값도 간단히 수정 가능
array1[0] = 9
array1[8] = 0
print('array1:', array1)
```

```
array1: [9 2 3 4 5 6 7 8 0]
```

```
[27] # 2차원 ndarray에서 인덱스는 [row, col]을 이용해 접근함
# 정확히 얘기하면 ndarray에서는 [axis0, axis1] 인덱스가 맞음. 이해를 돕기 위해 row, col 형
# 3차원 ndarray에서는 axis0, axis1, axis2 3개의 축을 가짐. (행, 열, 높이)
array1d = np.arange(start=1, stop=10)
array2d = array1d.reshape(3, 3)
print(array2d)

print('(row=0, col=0) index 가리키는 값:', array2d[0, 0])
print('(row=0, col=1) index 가리키는 값:', array2d[0, 1])
print('(row=1, col=0) index 가리키는 값:', array2d[1, 0])
print('(row=2, col=2) index 가리키는 값:', array2d[2, 2])
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
(row=0, col=0) index 가리키는 값: 1
(row=0, col=1) index 가리키는 값: 2
(row=1, col=0) index 가리키는 값: 4
(row=2, col=2) index 가리키는 값: 9
```

- axis 0이 로우 방향 축, axis 1이 칼럼 방향 축

## 슬라이싱

- 단일 데이터 값 추출을 제외하고 슬라이싱, 팬시 인덱싱, 불린 인덱싱으로 추출된 데이터 세트는 모두 ndarray 타입.
- ':' 사이에 시작 인덱스와 종료 인덱스를 표시하면 시작 인덱스에서 종료 인덱스-1의 위치에 있는 데이터의 ndarray를 반환함.
- ':' 사이의 시작, 종료 인덱스는 생략 가능함.

2차원 ndarray에서 뒤에 오는 인덱스를 없애면 1차원 ndarray를 반환함.

- array2d[0]과 같이 2차원에서 뒤에 오는 인덱스를 없애면 로우 축(axis 0)의 첫 번째 로우 ndarray를 반환하게 됨. 반환되는 ndarray는 1차원.

3차원 ndarray에서 뒤에 오는 인덱스를 없애면 2차원 ndarray를 반환함.

## 팬시 인덱싱

- 리스트나 ndarray로 인덱스 집합을 지정하면 해당 위치의 인덱스에 해당하는 ndarray를 반환하는 인덱싱 방식

```
▶ array1d = np.arange(start=1, stop=10)
  array2d = array1d.reshape(3, 3)

  array3 = array2d[[0, 1], 2]
  print('array2d[[0, 1], 2] => ', array3.tolist())

  array4 = array2d[[0, 1], 0:2]
  print('array2d[[0, 1], 0:2] => ', array4.tolist())

  array5 = array2d[[0, 1]] #((0,:), (1,:)) 인덱싱 적용
  print('array2d[[0, 1]] => ', array5.tolist())

☞ array2d[[0, 1], 2] => [3, 6]
  array2d[[0, 1], 0:2] => [[1, 2], [4, 5]]
  array2d[[0, 1]] => [[1, 2, 3], [4, 5, 6]]
```

## 불린 인덱싱

- 조건 필터링과 검색을 동시에 할 수 있기 때문에 자주 사용

```
▶ array1d = np.arange(start=1, stop=10)
  # [ ] 안에 array1d > 5 Boolean indexing 적용
  array3 = array1d[array1d > 5]
  print('array1d > 5 불린 인덱싱 결과 값 :', array3)

☞ array1d > 5 불린 인덱싱 결과 값 : [6 7 8 9]
```

## 불린 인덱싱 동작 과정

1. array1d > 5와 같이 ndarray의 필터링 조건을 안에 기재
2. False 값은 무시하고 True 값에 해당되는 index 값만 저장 (True 값을 가진 인덱스를 저장)
3. 저장된 index 데이터 세트로 ndarray 조회

내부적으로 여러 단계를 거치지만 코드 자체는 [ ] 안에 필터링 조건만 넣으면 되기 때문에 사용자는 이를 신경 쓸 필요 없음.

## 행렬의 정렬 - sort()와 argsort()

### 행렬 정렬

- np.sort() : 원 행렬은 그대로 유지한 채 원 행렬의 정렬된 행렬을 반환
- ndarray.sort() : 원 행렬 자체를 정렬한 형태로 변환하며 반환 값은 None

```
[40] org_array = np.array([3, 1, 9, 5])
print('원본 행렬:', org_array)
# np.sort()로 정렬
sort_array1 = np.sort(org_array)
print('np.sort() 호출 후 반환된 정렬 행렬:', sort_array1)
print('np.sort() 호출 후 원본 행렬:', org_array)
# ndarray.sort()로 정렬
sort_array2 = org_array.sort()
print('org_array.sort() 호출 후 반환된 정렬 행렬:', sort_array2)
print('org_array.sort() 호출 후 원본 행렬:', org_array) # 원본 행렬도 변환
```

```
원본 행렬: [3 1 9 5]
np.sort() 호출 후 반환된 정렬 행렬: [1 3 5 9]
np.sort() 호출 후 원본 행렬: [3 1 9 5]
org_array.sort() 호출 후 반환된 정렬 행렬: None
org_array.sort() 호출 후 원본 행렬: [1 3 5 9]
```

- 둘 모두 기본적으로 오름차순으로 정렬.
- 내림차순으로 정렬하기 위해선 `[::-1]`을 적용함.

```
[41] sort_array1_desc = np.sort(org_array)[::-1]
print('내림차순으로 정렬:', sort_array1_desc)
```

```
내림차순으로 정렬: [9 5 3 1]
```

```
[42] # 행렬이 2차원 이상일 경우에 axis 축 값 설정을 통해 로우 방향, 또는 칼럼 방향으로 정렬 수
array2d = np.array([[8, 12],
                    [7, 1 ]])
sort_array2d_axis0 = np.sort(array2d, axis=0)
print('로우 방향으로 정렬:\n', sort_array2d_axis0)

sort_array2d_axis1 = np.sort(array2d, axis=1)
print('칼럼 방향으로 정렬:\n', sort_array2d_axis1)
```

```
로우 방향으로 정렬:
[[ 7  1]
 [ 8 12]]
칼럼 방향으로 정렬:
[[ 8 12]
 [ 1  7]]
```

## 정렬된 행렬의 인덱스를 반환하기

- `np.argsort()` : 정렬 행렬의 원본 행렬 인덱스를 ndarray 형으로 반환함.

```
# 시험 성적순으로 학생 이름을 출력하고자 할 때 사용할 수 있다
# 넘파이는 메타 데이터를 가질 수 없기 때문에 이름 ndarray, 성적 ndarray를 각각 가져야 하기 때문.
import numpy as np

name_array = np.array(['John', 'Mike', 'Sarah', 'Kate', 'Samuel'])
score_array = np.array([78, 95, 84, 98, 88])

sort_indices_asc = np.argsort(score_array)
print('성적 오름차순 정렬 시 score_array의 인덱스:', sort_indices_asc)
print('성적 오름차순으로 name_array의 이름 출력:', name_array[sort_indices_asc]) # 이 때 사용
```

↗ 성적 오름차순 정렬 시 score\_array의 인덱스: [0 2 4 1 3]  
성적 오름차순으로 name\_array의 이름 출력: ['John' 'Sarah' 'Samuel' 'Mike' 'Kate']

## 선형대수 연산 - 행렬 내적과 전치 행렬 구하기

### 행렬 내적(행렬 곱)

- `np.dot()` 을 통해 행렬 곱을 수행할 수 있음.

### 전치 행렬

- 원 행렬에서 행과 열 위치를 교환한 원소로 구성된 행렬을 그 행렬의 전치 행렬이라고 함.
- `transpose()` 를 이용해 전치 행렬을 구할 수 있음.

## 데이터 핸들링 - 판다스(Pandas)

- 판다스는 행과 열로 이뤄진 2차원 데이터를 효율적으로 가공/처리할 수 있는 다양하고 훌륭한 기능을 제공함.
- 판다스의 핵심 객체는 DataFrame
  - 데이터프레임은 여러 개의 행과 열로 이뤄진 2차원 데이터를 담는 데이터 구조체
  - 판다스가 다루는 대부분의 영역은 이 데이터프레임에 관련된 부분임.
- Index, Series 객체
  - Index = RDBMS의 PK처럼 개별 데이터를 고유하게 식별하는 Key 값
  - Series와 DataFrame은 모두 Index를 key 값으로 가지고 있음.
- Series vs DataFrame
  - Series는 칼럼이 하나뿐인 데이터 구조체
  - DataFrame은 칼럼이 여러 개인 데이터 구조체

⇒ 한마디로 데이터프레임은 여러 개의 시리즈로 이뤄졌다고 할 수 있음.

## 판다스 API

판다스는 다양한 포맷으로 된 파일을 데이터프레임으로 로딩할 수 있는 편리한 API 를 제공함.

- `read_csv()` : 디폴트 필드 구분문자가 콤마(,)
- `read_table()` : 디폴트 필드 구분문자가 탭(\t)
- `read_fwf()` : 고정 길이 기반의 칼럼 포맷을 데이터프레임으로 로딩
- `read_csv()` 는 csv 뿐만 아니라 어떤 필드 구분 문자 기반의 파일 포맷도 DataFrame으로 변환이 가능함. `read_csv()`의 인자인 `sep`에 해당 구분 문자를 입력해주면 됨.
  - ex) `read_csv('파일명', sep='\t')` - 탭으로 필드 구분
- `read_csv()`와 `read_table()`은 기능상 큰 차이가 없음
- `read_csv()` 는 호출 시 파일명 인자로 들어온 파일을 로딩해 데이터프레임 객체로 반환함.
- 또한 별다른 파라미터 지정이 없으면 파일의 맨 처음 로우를 칼럼명으로 인지하고 칼럼으로 변환함.



	PassengerId	Survived	Pclass	Name
0	1	0	3	Braund, Mr. Owen Harris
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...
2	3	1	3	Heikkinen, Miss. Laina
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)
4	5	0	3	Allen, Mr. William Henry
...	...	...	...	...

- 맨 왼쪽에 원본 파일에 기재되지 않았는데 0, 1, 2, 3... 이렇게 순차적으로 적혀있는 데이터값은 판다스의 **Index 객체 값**이다.
- 모든 데이터프레임 내의 데이터는 생성되는 순간 고유의 Index 값을 가지게 됨. → RDBMS의 PK와 유사하게 고유의 레코드를 식별하는 역할

- **DataFrame.head(N)** : DataFrame의 맨 앞에 있는 N개의 로우를 반환.

◦ ex) head(3) : 맨 앞 3개의 로우를 반환 (디폴트는 5개)

- **DataFrame.shape** : DataFrame의 행과 열 크기를 튜플 형태로 반환.


◦ shape는 DataFrame 객체의 변수

```
[55] print('DataFrame 크기: ', titanic_df.shape)
```

DataFrame 크기: (891, 12)

891개의 로우와 12개의 칼럼으로 이루어져 있음

- **info()** : 총 데이터 건수, 데이터 타입, Null 건수 파악 가능.
- **describe()** : 칼럼별 숫자형 데이터값의 n-percentile 분포도, 평균값, 최댓값, 최솟값을 나타냄.
  - 오직 숫자형(int, float 등) 칼럼의 분포도만 조사하며 자동으로 object 타입의 칼럼은 출력에서 제외시킴
- 데이터의 분포도를 아는 것 = 머신러닝 알고리즘의 성능을 향상시키는 중요한 요소.
- describe() 메서드만으로 정확한 분포도를 알기에는 무리지만 개략적인 분포도를 확인할 수 있어서 유용함.

 titanic\_df.describe()

	PassengerId	Survived	Pclass	Age	SibSp	Parch
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000

describe() 결과 값. 25%는 25 percentile 값.

- describe()는 해당 숫자 칼럼이 숫자형 카테고리 칼럼인지를 판단할 수 있게 도와줌.



- Pclass의 경우 min이 1, 25%~75%가 2와 3, max가 3이므로 1, 2, 3으로 이뤄진 숫자형 카테고리 칼럼일 것.
- 반환 값은 많은 건수 순서대로 정렬된다.

- ★ `value_counts()` : 해당 칼럼 값의 유형과 건수를 반환하는 메서드

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

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

value\_counts() 결과 값. 3이 491개, 1이 216개, 2가 184개.

- DataFrame의 [] 연산자 내부에 칼럼명을 입력하면 해당 칼럼에 해당하는 Series 객체를 반환함. → 데이터프레임은 시리즈의 집합이다

```
titanic_pclass = titanic_df['Pclass']
print(type(titanic_pclass))
```

```
<class 'pandas.core.series.Series'>
```

DataFrame 안에 'Pclass'라는 Series.

```
titanic_pclass.head()
```

*Series의 index*

0	3
1	1
2	3
3	1
4	3

*Series의 데이터 값*

*\* Series의 index = DataFrame의 index*

```
Name: Pclass, dtype: int64
```

Pclass Series 내부 모습.

- Series는 Index와 단 하나의 칼럼으로 구성된 데이터 세트
- Series와 DataFrame은 인덱스를 반드시 가짐
- 인덱스 값은 단순히 0부터 순차적으로 의미없는 식별자만 할당하는 것이 아니라 고유성이 보장된다면 의미 있는 데이터값 할당도 가능하다.
- value\_counts는 칼럼 값별 데이터 건수를 반환하므로 고유 칼럼 값을 식별자로 사용할 수 있는 것
- 인덱스는 데이터프레임, 시리즈가 만들어진 후에도 변경될 수 있고 숫자형, 문자열도 가능하지만 모든 인덱스는 고유성이 보장되어야 함.
- `dropna` : 기본값 True, Null 값 무시하고 개별 데이터 값의 건수 계산함.
- null값 포함하여 value\_counts() 적용하고자 하면 value\_counts(dropna=False)로 하면 됨.

## DataFrame과 리스트, 딕셔너리, 넘파이 ndarray 상호 변환

- DataFrame은 리스트, 넘파이, ndarray와 다르게 칼럼명을 갖고 있기 때문에 칼럼명을 따로 지정해준다(지정하지 않으면 자동 할당)
- DataFrame은 기본적으로 행/열을 가지는 2차원 데이터이기 때문에 2차원 이하의 데이터들만 DataFrame으로 변환될 수 있음.

- `pd.DataFrame( 리스트or넘파이or ndarray , columns= col_name )` 형태

#### • 딕셔너리 → DataFrame 변환

- 딕셔너리의 키(Key)는 칼럼명, 딕셔너리의 값(Value)은 키에 해당하는 칼럼 데이터로 변환됨.
- 키는 문자열, 값은 리스트 (or ndarray) 형태로 딕셔너리를 구성함.
- `pd.DataFrame(딕셔너리)` 형태

#### • DataFrame → 넘파이 ndarray, 리스트, 딕셔너리로 변환

- 넘파이 ndarray로 변환 : DataFrame 객체의 values를 이용
- 리스트로 변환 : values로 얻은 ndarray에 tolist() 호출
- 딕셔너리로 변환 : DataFrame 객체의 to\_dict() 메서드를 호출하는데, 인자로 'list'를 입력하면 딕셔너리 값이 리스트형으로 반환됨.

## DataFrame의 칼럼 데이터 세트 생성과 수정

- [] 연산자를 이용해 생성과 수정이 가능함.

## DataFrame 데이터 삭제

- drop() 메서드를 이용해 데이터를 삭제할 수 있다.
- axis1을 입력하면 칼럼 축 방향으로 드롭을 수행하므로 칼럼을 드롭하겠다는 의미
- axis0을 입력하면 로우 축 방향으로 드롭을 수행하므로 로우를 드롭하겠다는 의미
- DataFrame의 특정 로우를 가리키는 것은 인덱스기 때문에 axis를 0으로 지정하면 DataFrame은 자동으로 labels에 오는 값을 인덱스로 간주함.

## Index 객체

- 판다스의 Index 객체는 RDBMS의 PK(Primary Key)와 유사하게 DataFrame, Series의 레코드를 고유하게 식별하는 객체
- DataFrame, Series에서 Index 객체만 추출하려면 DataFrame.index 또는 Series.index 속성을 통해 가능하다.
- 한 번 만들어진 DataFrame 및 Series의 Index 객체는 함부로 변경할 수 없음.
- Series 객체는 Index 객체를 포함하지만 Series 객체에 연산 함수를 적용할 때 Index는 연산에서 제외된다. Index는 오직 식별용으로만 사용되기 때문.
- `reset_index()` 메서드 : 새롭게 인덱스를 연속 숫자 형으로 할당함. 기존 인덱스는 'index'라는 새로운 칼럼명으로 추가함.
  - Series에 reset\_index()를 적용하면 새롭게 연속 숫자형 인덱스가 만들어지고 기존 인덱스는 'index'칼럼명으로 추가되면서 DataFrame으로 변환됨.
  - 파라미터를 Drop=True로 설정하면 기존 인덱스는 그대로 삭제되고 새로운 칼럼이 추가되지 않으므로 그대로 Series로 유지됨.

## 데이터 선택 및 필터링

- DataFrame의 [] 연산자 (넘파이, Series에서의 [] 연산자와는 다름!)
  - 칼럼명 문자(또는 칼럼명의 리스트 객체), 또는 인덱스로 변환 가능한 표현식이 들어감.
  - 칼럼만 지정할 수 있는 칼럼 지정 연산자
  - 슬라이싱도 가능하고 불린 인덱싱도 적용할 수 있음
    - 슬라이싱은 권장되지 않음.
- DataFrame의 iloc[] 연산자
  - iloc : 위치 기반 인덱싱, loc : 명칭 기반 인덱싱
  - DataFrame의 로우나 칼럼을 지정하여 데이터를 선택할 수 있는 인덱싱 방식
- 불린 인덱싱
  - 가져올 값을 조건으로 [] 내에 입력하면 자동으로 원하는 값을 필터링해줌 (Numpy 참조)

## 정렬, Aggregation 함수, GroupBy 적용

- **DataFrame, Series 정렬** - `sort_values()`
  - by로 특정 칼럼을 입력하면 해당 칼럼으로 정렬을 수행함.
  - 기본적으로 오름차순이고, `ascending=False`로 내림차순 수행.
  - `inplace=False`로 설정하면 기존 DataFrame은 그대로 유지되며 정렬된 DataFrame을 결과로 반환함. 기본은 `inplace=False`.
- **Aggregation 함수 적용**
  - DataFrame에서 min, max, sum, count 등의 aggregation을 호출할 경우 모든 칼럼에 해당 aggregation을 적용한다.
- **Groupby() 적용**
  - 입력 파라미터 by에 칼럼을 입력하면 대상 칼럼으로 groupby가 됨.
  - DataFrame에 groupby를 호출하면 DataFrameGroupBy라는 또 다른 형태의 DataFrame을 반환함.
  - 데이터프레임에 groupby를 호출하여 반환된 결과에 aggregation 함수를 호출하면 groupby 대상 칼럼을 제외한 모든 칼럼에 해당 aggregation 함수를 적용함.

## 결손 데이터 처리하기

- **isna() 로 결손 데이터 여부 확인**
  - 모든 칼럼의 값이 NaN인지 아닌지를 True나 False로 알려줌
  - 결손 데이터 개수는 isna() 결과에 sum() 함수를 추가해 구할 수 있음
  - ex ) `titanic_df.isna().sum()`
- **fillna()로 결손 데이터 대체하기**
  - 주의할 점은 fillna()를 이용해 반환값을 다시 받거나 `inplace=True` 파라미터를 추가해야 실제 데이터셋 값이 변경됨.

## apply lambda 식으로 데이터 가공

- 일괄적인 데이터 가공을 하는 것이 속도 면에서 더 빠르나 복잡한 데이터 가공이 필요한 경우 apply lambda를 이용한다.
- 람다 식은 if else를 지원한다. → 주의: if 식보다 반환값을 먼저 기술해야함.

## 첫 번째 머신러닝 만들어 보기 - 붓꽃 품종 예측하기

- 분류(Classification)는 대표적인 지도학습 방법의 하나.
- 지도학습(Supervised Learning)
  - 학습을 위한 다양한 피처와 분류 결정값인 레이블(Label) 데이터로 모델을 학습한 뒤, 별도의 테스트 데이터 세트에서 미지의 레이블을 예측.
  - 명확한 정답이 주어진 데이터를 먼저 학습한 뒤, 미지의 정답을 예측하는 방식.
- 학습을 위해 주어진 데이터 세트 : `학습 데이터 세트`
- 머신러닝 모델의 예측 성능을 평가하기 위해 별도로 주어진 데이터 세트 : `테스트 데이터 세트`

## 학습용 데이터와 테스트용 데이터 분리하기

- 사이킷런에서 제공하는 `train_test_split()` API 이용
- 학습 데이터와 테스트 데이터를 `test_size` 파라미터 입력값의 비율로 쉽게 분할함.
- `train_test_split`(피처 데이터 세트, 레이블 데이터 세트, `test_size`=테스트 데이터셋 비율, `random_state`=난수 발생 값)
  - 난수 발생 값을 설정하지 않으면 수행할 때마다 다른 학습/테스트 용 데이터를 만들 수 있음(설정하면 호출할 때마다 같은 학습/테스트 용 데이터셋을 만듦)
- **학습 수행** : `fit(학습용 피처 데이터셋, 결정값)` 메서드
- **예측** : `predict(테스트 피처 데이터셋)` 메서드
- 머신러닝 모델의 성능 평가 방법 - 정확도

- 정확도는 예측 결과가 실제 레이블 값과 얼마나 정확하게 맞는지를 평가함.
- 사이킷런은 정확도 측정을 위해 `accuracy_score()` 함수를 제공함.

## 붓꽃 데이터셋 분류 예측 프로세스

### 데이터셋 분리 → 모델 학습 → 예측 수행 → 모델 평가

1. **데이터 세트 분리** : 데이터를 학습 데이터와 테스트 데이터로 분류한다.
2. **모델 학습** : 학습 데이터를 기반으로 ML 알고리즘을 적용해 모델을 학습시킨다.
3. **예측 수행** : 학습된 ML 모델을 이용해 테스트 데이터의 분류(즉, 붓꽃 종류)를 예측한다.
4. **평가** : 이렇게 예측된 결과값과 테스트 데이터의 실제 결과값을 비교해 ML 모델 성능을 평가한다.

## 사이킷런의 기반 프레임워크 익히기

### Estimator 이해 및 `fit()`, `predict` 메서드

- 분류와 회귀의 다양한 알고리즘을 구현한 모든 사이킷런 클래스는 `fit()`과 `predict()`만을 이용해 간단하게 학습과 예측 결과를 반환한다.
- **Classifier** : 분류 알고리즘을 구현한 클래스
- **Regressor** : 회귀 알고리즘을 구현한 클래스
- → 둘 모두를 합쳐서 **Estimator** 클래스라고 부름.
  - 지도학습의 모든 알고리즘을 구현한 클래스
- 비지도학습인 차원 축소, 클러스터링, 피쳐 추출 등을 구현한 클래스 역시 대부분 `fit()`과 `transform()`을 적용함.
- 여기서의 `fit()`은 지도학습에서의 `fit()`과 같이 학습을 의미하는 것이 아닌 입력 데이터의 형태에 맞춰 데이터를 변환하기 위한 사전 구조를 맞추는 작업.
- `fit()`으로 변환을 위한 사전 구조를 맞추면 이후 **실제 작업은 `transform()`으로 이뤄짐.**

### 내장된 예제 데이터 세트

- 사이킷런에 내장된 데이터 셋은 일반적으로 딕셔너리 형태로 되어있음.
- 키는 보통 `data`, `target`, `target_name`, `feature_names`, `DESCR`로 구성됨.
- `data` : 피쳐의 데이터셋을 가리킴
- `target` : 분류 시 레이블 값, 회귀일 때는 숫자 결과값 데이터 셋
- `target_names` : 개별 레이블의 이름
- `feature_names` : 피쳐의 이름
- `DESCR` : 데이터 셋에 대한 설명과 각 피쳐의 설명
- `data`, `target`은 넘파이 배열(`ndarray`) 타입이며, `target_names`, `feature_names`는 넘파이 배열 또는 파이썬 list 타입. `DESCR`은 스트링 타입
- 피쳐의 데이터 값을 반환받기 위해선 내장 데이터셋 API를 호출한 뒤 그 키값을 지정하면 됨.
- **Bunch** 클래스 : 파이썬 딕셔너리 자료형과 유사함
- 대부분의 데이터셋은 이와 같이 딕셔너리 형태의 값을 반환함.

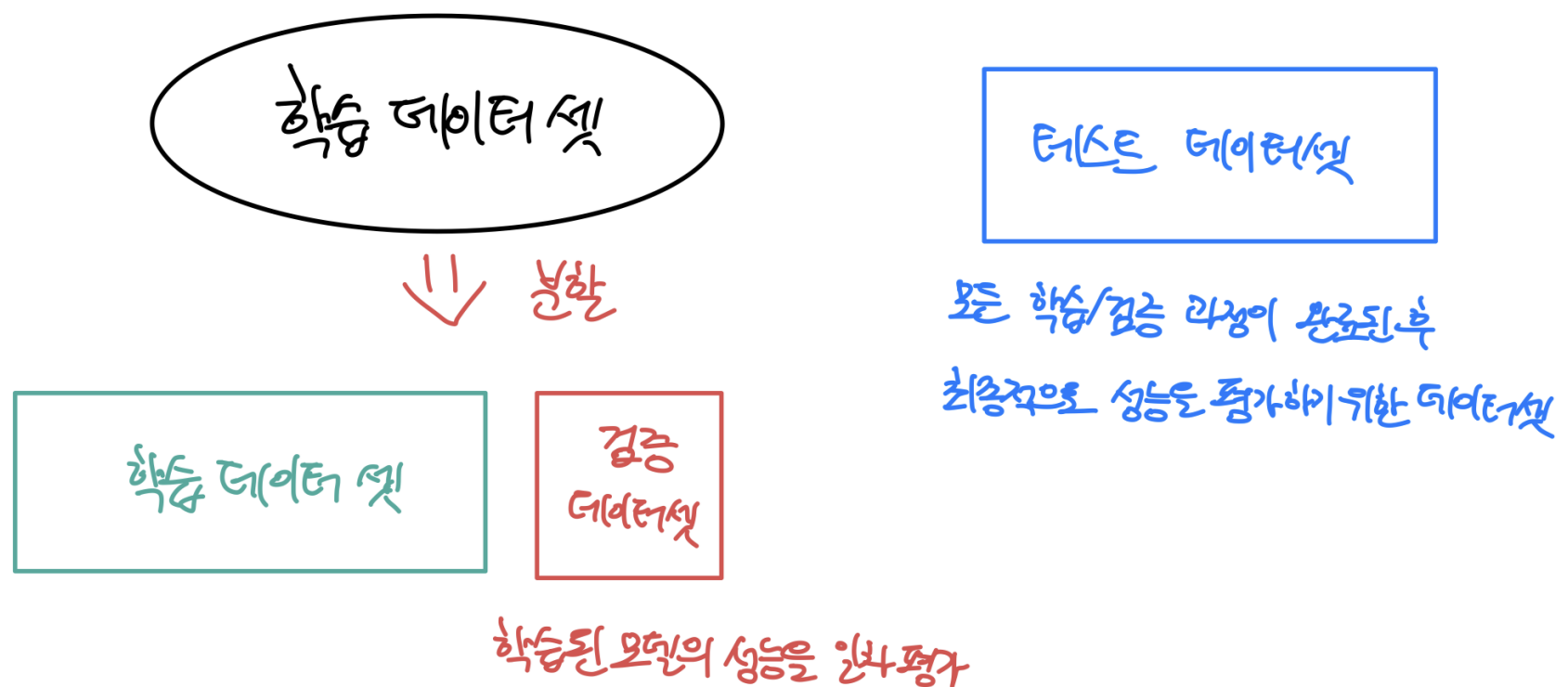
## Model Selection 모듈 소개

### 학습/테스트 데이터 분리 - train\_test\_split()

- 학습과 예측을 동일한 데이터셋으로 수행하게 되면 이미 학습한 데이터를 기반으로 예측하기 때문에 정확도가 100%가 나온다.
- 따라서 예측을 수행하는 데이터셋은 학습을 수행한 학습용 데이터셋이 아닌 전용의 데이터셋이어야함.
- train\_test\_split()을 통해 학습 데이터와 테스트 데이터셋을 분리 가능.
- 학습을 위한 데이터 양을 일정 수준 이상 보장하는 것도 중요하지만, 학습된 모델에 대해 다양한 데이터를 기반으로 예측 성능을 평가하는 것도 매우 중요함.

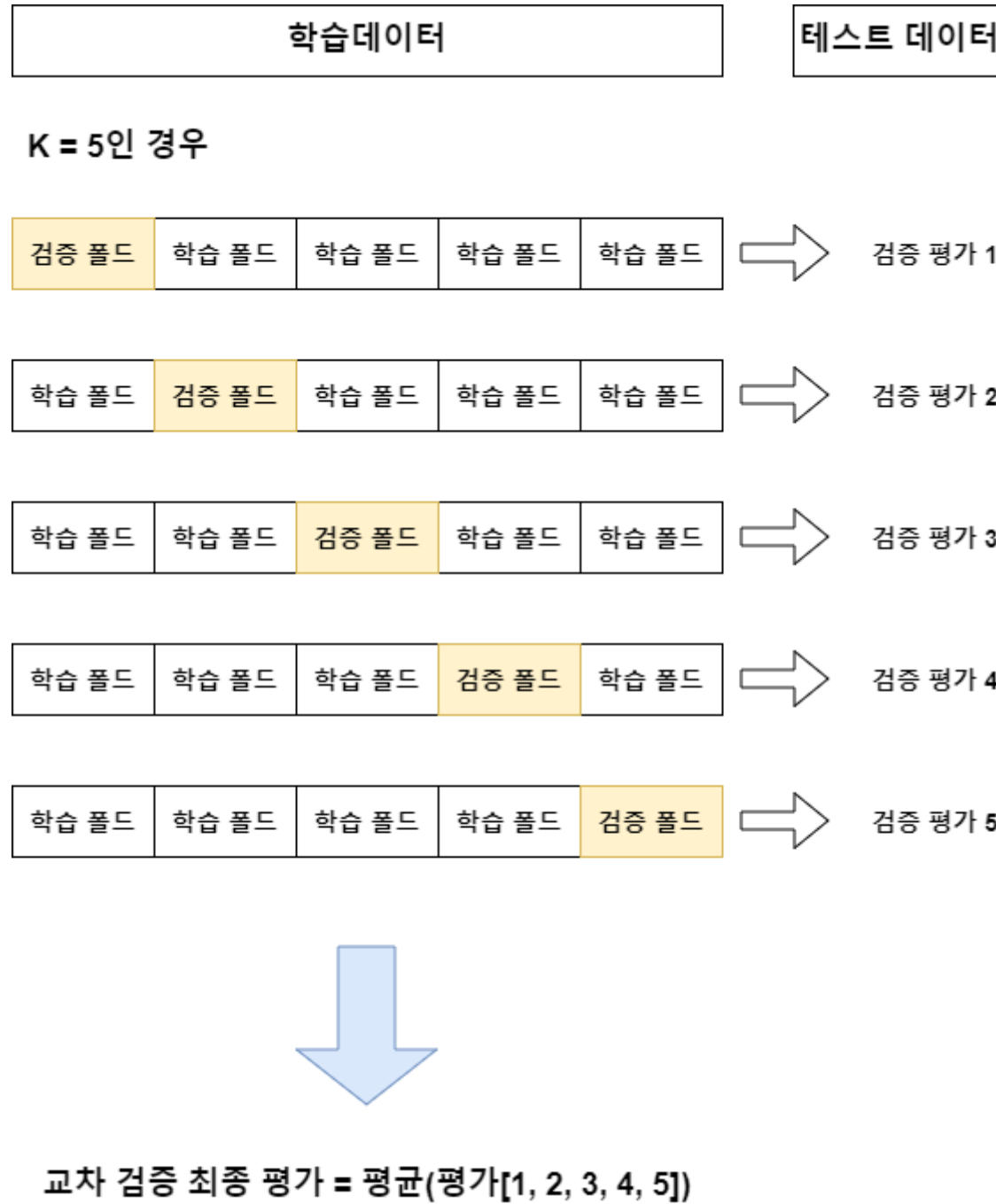
### 교차 검증

- 고정된 학습 데이터와 테스트 데이터로만 평가를 하다 보면 테스트 데이터에만 최적의 성능을 발휘하도록 **과적합(Overfitting)** 되는 경우가 생겨 실제 예측을 다른 데이터로 수행할 시 예측 성능이 과도하게 떨어질 수 있음.
- 이를 예방하기 위해 교차 검증을 이용해 더 다양한 학습과 평가를 수행함.



### K 폴드 교차검증

- K개의 데이터 폴드 세트를 만들어서 K번만큼 각 폴드 세트에 학습과 검증 평가를 반복적으로 수행하는 방법
- 가장 보편적으로 사용되는 교차 검증 기법



- 교차검증은 sklearn의 KFold에 구현되어 있으며, KFold 객체 생성 후에 split() 메소드를 활용하여 각각 반환된 train index와 test index를 이용해 학습용, 검증용 테스트 데이터를 추출한다. (이때, K만큼 for loop 내에서 반복)

## Stratified K 폴드

- 불균형한(imbalanced) 분포도를 가진 레이블(결정 클래스) 데이터 집합을 위한 K 폴드 방식**
- 불균형한 분포도 = 특정 레이블 값이 특이하게 많거나 매우 적어서 값의 분포가 한쪽으로 치우치는 것을 의미함.
- Stratified K 폴드는 원본 데이터의 레이블 분포를 먼저 고려한 뒤 이 분포와 동일하게 학습과 검증 데이터 세트를 분배하여 KFold로 분할된 레이블 데이터 세트가 전체 레이블 값의 분포도를 반영하지 못하는 문제를 해결해줌.
- 실제 데이터 분포가 불균형할 때 데이터셋을 잘못 쪼개게 되면 어떤 레이블은 학습 데이터셋에 아예 속하지 않게 된다거나 하는 문제가 생길 수 있기 때문.
- K폴드 방식과 Stratified K폴드 방식은 거의 비슷하지만 큰 차이는 **StratifiedKFold는 레이블 데이터 분포도에 따라 학습/검증 데이터를 나누기 때문에 split() 메서드에 인자로 피쳐 데이터셋 뿐만 아니라 레이블 데이터셋도 반드시 필요하다.**
  - K폴드의 경우 레이블 데이터셋은 split 메서드의 인자로 입력하지 않아도 무방함.

## 교차 검증을 보다 간편하게 - cross\_val\_score()

- K 폴드 데이터 학습 및 예측 코드 방식**
  - 폴드 세트 설정 → for loop에서 반복으로 학습 및 테스트 데이터의 인덱스 추출 → 반복 학습 및 예측을 수행하고 예측 성능 반환
- cross\_val\_score() 방식**
  - train index, test index를 발생시키지 않고 바로 교차 검증 후 score를 리턴해주는 API

- 내부에서 Estimator를 학습(fit), 예측(predict), 평가(evaluation) 시켜주므로 간단하게 교차 검증을 수행할 수 있음.
- classifier가 입력되면 내부적으로 Stratified K 폴드 방식으로 레이블값의 분포에 따라 학습/테스트 셋을 분할함 (회귀인 경우 Stratified K 폴드 방식으로 분할할 수 없으므로 K 폴드 방식으로 분할)
- 하지만 scoring에 있어서 지원하는 평가 지표가 제한되어 있기 때문에 앞의 KFold를 알아둘 필요가 있음.
- 비슷한 API로 cross\_validate()가 있는데 cross\_val\_score()가 단 하나의 평가 지표를 반환하는 것과 달리 cross\_validate()는 여러 개의 평가 지표를 반환할 수 있음. 또한 학습 데이터에 대한 성능 평가 지표와 수행 시간도 같이 제공함. (그러나 보통 cross\_val\_score() 하나로 대부분의 경우 쉽게 사용함)

## GridSearchCV - 교차 검증과 최적 하이퍼 파라미터 튜닝을 한 번에

- **하이퍼 파라미터** : 머신러닝 알고리즘을 구성하는 주요 구성 요소이며, 이 값을 조정해 알고리즘의 예측 성능을 개선할 수 있음.
- 사이킷런은 GridSearchCV API를 이용해 분류나 회귀와 같은 알고리즘에 사용되는 하이퍼 파라미터를 **순차적으로 입력하면서 편리하게 최적의 파라미터를 도출할 수 있는 방안**을 제공함.
- 파라미터 집합 만들기 → 순차적으로 파라미터를 바꿔 실행하며 최적화 수행
- GridSearchCV는 교차 검증을 기반으로 이 하이퍼 파라미터의 최적 값을 찾게 해줌.
  - 사용자가 튜닝하고자 하는 여러 종류의 하이퍼 파라미터를 다양하게 테스트하면서 최적의 파라미터를 편리하게 찾게 해주지만 동시에 순차적으로 파라미터를 테스트하므로 수행 시간이 상대적으로 오래 걸림.

GridSearchCV는 사용자가 하이퍼 파라미터마다 몇 가지 값을 가진 리스트를 입력하면, 가능한 하이퍼 파라미터의 경우의 수마다 예측 성능을 측정하여 사용자가 일일이 하이퍼 파라미터를 설정하고, 예측 성능을 비교하여 최적의 파라미터를 찾는 수고를 줄이고 이 과정을 한꺼번에 진행한다.

## GridSearchCV API 동작 과정

1. train\_test\_split()을 이용하여 학습 데이터셋과 테스트 데이터셋을 분리한다.
2. 하이퍼 파라미터 세트를 딕셔너리 형태로 변수에 저장한다
  - a. ex) parameters = {'max\_depth':[1, 2, 3], 'min\_samples\_split':[2, 3]}
3. GridSearchCV 객체의 fit(학습 데이터 세트) 메서드에 인자로 입력한다.
  - a. fit 메서드를 수행하면 학습 데이터를 cv에 기술된 폴딩 세트로 분할해 param\_grid에 기술된 파라미터를 순차적으로 변경하며 학습과 평가를 수행한다. 그 후, cv\_result\_라는 속성에 기록한다.
  - b. cv\_result\_는 gridsearchcv의 결과 세트로서 딕셔너리 형태를 가진다.
4. cv\_result\_가 딕셔너리 형태를 가진 것을 이용하여 Pandas의 DataFrame으로 변환하면 내용을 더 쉽게 볼 수 있음.

## GridSearchCV API 결과값 칼럼별 의미

- params 칼럼 : 수행할 때마다 적용된 개별 하이퍼 파라미터 값
- rank\_test\_score : 하이퍼 파라미터별 성능이 좋은 score순위 1이 가장 뛰어난 순위이며 이때의 파라미터가 최적의 하이퍼 파라미터임.
- mean\_test\_score : 개별 하이퍼 파라미터별로 CV의 폴딩 테스트 세트에 대해 총 수행한 평가 평균값
- **best\_params\_ 속성** : 최고 성능을 나타낸 하이퍼 파라미터의 값
- **best\_score\_ 속성** : 최고 성능을 나타낸 하이퍼 파라미터의 평가 결과값 (정확도)
- GridSearchCV 객체의 생성 파라미터로 refit=True가 디폴트인데, refit=True이면 GridSearchCV가 최적 성능을 나타내는 하이퍼 파라미터로 Estimator를 학습해 **best\_estimator\_** 로 저장한다.



- 일반적으로 학습 데이터를 **GridSearchCV**를 이용해 최적 하이퍼 파라미터 튜닝을 수행한 뒤에 별도의 테스트에서 이를 평가하는 것이 일반적인 머신러닝 모델 적용 방법.

## 데이터 전처리

- ML알고리즘은 데이터를 기반으로 하기 때문에 어떤 데이터를 입력으로 받느냐에 따라 결과도 크게 달라질 수 있음.



사이킷런의 ML 알고리즘을 적용하기 전 미리 데이터에 대해 처리해야 할 사항이 있다.

- **결손값, 즉 NaN, Null 값은 허용되지 않으므로 고정된 다른 값으로 변환해야함.**
  - 피쳐 값 중 Null 값이 얼마 되지 않는다면 **피쳐의 평균값** 등으로 간단히 대체 가능하지만 Null 값이 대부분이라면 오히려 해당 피쳐는 드롭하는 것이 더 좋음
  - 일정 수준 이상 되는 경우 정확한 수치는 없지만 해당 피쳐가 중요도가 높은 피쳐이고 Null을 단순히 피쳐의 평균값으로 대체할 경우 예측 왜곡이 심할 수 있다면 업무 로직 등을 상세히 검토해 더 정밀한 대체 값을 선정해야 함.
- **사이킷런의 ML 알고리즘은 문자열 값을 입력값으로 허용하지 않으므로 모든 문자열 값은 인코딩되어 숫자 형태로 변환되어야 함.**
- **문자열 피쳐**
  - 카테고리형 피쳐
  - 텍스트형 피쳐
    - 피쳐 벡터화(feature vectorization) 기법으로 벡터화하거나 삭제 권장

## 데이터 인코딩

- 레이블 인코딩
- 원-핫 인코딩(One-Hot Encoding)

## 레이블 인코딩

- 카테고리형 피쳐를 코드형 숫자 값으로 변환하는 것
- 레이블 인코딩은 **LabelEncoder** 클래스로 구현함. LabelEncoder를 객체로 생성한 후 fit()과 transform()을 호출해 레이블 인코딩을 수행.
- 문자열 값이 어떤 숫자 값으로 인코딩됐는 지 알기 위해선 LabelEncoder 객체의 **classes\_** 속성 값 확인하기
  - 0번부터 순서대로 변환된 인코딩 값에 대한 원본값을 가짐
- **inverse\_transform()** 을 통해 인코딩된 값을 다시 디코딩할 수 있음

인코딩 : transform(), 디코딩 : inverse\_transform()

- 레이블 인코딩을 통해 일괄적으로 숫자 값으로 변환되면서 숫자 값의 크고 작음에 대한 특성이 작용해 특정 ML 알고리즘에서 가중치가 더 부여되거나 더 중요하게 인식할 가능성이 발생함
- → 하지만 숫자 값에 따른 순서나 중요도로 인식되어서는 안되기 때문에 레이블 인코딩은 선형 회귀와 같은 ML 알고리즘에는 적용하지 않아야 함.
- 트리 계열의 ML 알고리즘은 숫자의 특성이 반영되지 않으므로 별 문제가 없음.
- 이 문제를 해결한 것이 바로 **원-핫 인코딩**

## 원-핫 인코딩(One-Hot Encoding)

- 피쳐 값의 유형에 따라 새로운 피쳐를 추가해 **고유 값에 해당하는 칼럼에만 1을 표시하고 나머지 칼럼에는 0을 표시하는 방식**
- 행 형태로 되어 있는 피쳐의 고유 값을 열 형태로 차원을 변환한 뒤, 고유 값에 해당하는 칼럼에만 1을 표시하고 나머지 칼럼에는 0을 표시함.

ID	과일
1	사과
2	바나나
3	체리

One-Hot Encoding

ID	사과	바나나	체리
1	1	0	0
2	0	1	0
3	0	0	1

LabelEncoder

ID	과일
1	0
2	1
3	2

두 인코딩 방식의 차이점

- 원-핫 인코딩은 사이킷런에서 OneHotEncoder 클래스로 변환이 가능함.
- 단, **입력 값으로 2차원 데이터**가 필요하며, OneHotEncoder를 이용해 변환한 값이 **희소 행렬(Sparse Matrix) 형태**이므로 이를 다시 **toarray() 메서드**를 이용해 **밀집 행렬(Dense Matrix)로 변환해야 함**.
- 판다스에서는 원-핫 인코딩을 더 쉽게 지원하는 get\_dummies()라는 API가 있음.
  - 사이킷런의 OneHotEncoder와 다르게 문자열 카테고리 값을 숫자 형으로 변환할 필요 없이 바로 변환 가능.

## 피처 스케일링과 정규화

- 하지만 숫자형 데이터들도 피처에 따라 단위가 다르기 때문에, 분포가 다를 것이고 이를 제대로 고려하지 않으면 예측 성능이 떨어지는 원 인 중 하나가 되므로 **피처 스케일링**이 필요함.
- **피처 스케일링 : 서로 다른 변수의 값 범위를 일정한 수준으로 맞추는 작업**
  - 표준화(Standardization)
  - 정규화(Normalization)

## 표준화

- 데이터의 피처 각각이 평균이 0이고 분산이 1인 가우시안 정규 분포를 가진 값으로 변환하는 것을 의미.
- 표준화를 통해 변환될 피처 x의 새로운 i번째 데이터를 xi\_new라고 한다면 이 값은 원래 값에서 피처 x의 평균을 뺀 값을 피처 x의 표준편 차로 나눈 값으로 계산할 수 있음.

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

## 정규화

- 분포 보다는 서로 다른 피처의 크기를 통일하기 위해 크기를 변환해주는 개념.
- 개별 데이터의 크기를 모두 똑같은 단위로 변경하는 것
- ex) 피처 A는 KM, 피처 B는 원 단위라면 동일한 크기 단위로 비교하기 위해 값을 모두 최소 0~ 최대 1의 값으로 변환하는 것.

- 새로운 데이터  $x_{i\_new}$ 는 원래 값에서 피쳐  $x$ 의 최솟값을 뺀 값을 피쳐  $x$ 의 (최댓값 - 최솟값)으로 나눈 값으로 변환할 수 있음.

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

- 사이킷런의 전처리에서 제공하는 Normalizer 모듈과 일반적인 정규화는 약간 차이가 있는데, 사이킷런의 모듈에서는 선형대수에서의 정규화 개념이 적용됐으며, 개별 벡터의 크기를 맞추기 위해 변환하는 것을 의미한다.
- = 개별 벡터를 모든 피쳐 벡터의 크기로 나눠 준다.

## StandardScaler

- StandardScaler은 표준화를 쉽게 지원해 주는 함수
- 즉, 피쳐들을 평균이 0이고 분산이 1인 값으로 변환을 시켜줌.
- 서포트 벡터 머신(Support Vector Machine), 선형 회귀(Linear Regression), 로지스틱 회귀(Logistic Regression)과 같은 ML 알고리즘들은 가우시안 분포를 가지고 있다고 가정을 하고 구현되었기 때문에 표준화를 적용하는 것이 예측 성능 향상에 중요한 요소로 작용될 수 있다.

## MinMaxScaler

- MinMaxScaler은 데이터의 값들을 0과 1사이의 범위 값으로 변환하여 줍니다. (음수 값이 있다면 -1에서 1값으로 변환됨) 데이터들의 분포가 가우시안 분포가 아닐 경우에 적용을 해볼 수 있다.

## 학습 데이터와 테스트 데이터의 스케일링 변환 시 유의점

- 학습 데이터셋과 테스트 데이터셋에 fit()과 transform()을 적용할 때, Scaler 객체를 이용해 학습 데이터셋으로 fit()과 transform()을 적용하면 **테스트 데이터셋으로는 다시 fit()을 수행하지 않고 학습 데이터셋으로 fit()을 수행한 결과를 이용해 transform() 변환을 적용해야 함.**

→ 테스트 데이터로 다시 새로운 스케일링 기준을 만들어 버리면 학습 데이터와 테스트 데이터의 **스케일링 기준 정보가 달라지기 때문에 올바른 예측 결과가 도출되지 못할 수 있음.**

- 머신러닝 모델은 학습 데이터를 기반으로 학습되기 때문에 반드시 테스트 데이터는 학습 데이터의 기준에 따라야 하며, 테스트 데이터의 1값은 학습 데이터와 동일하게 0.1 값으로 변환되어야 함.

- → **테스트 데이터에 다시 fit()을 적용하면 안 되며 학습 데이터로 이미 fit()이 적용된 Scaler 객체를 이용해 transform()으로 변환해야 함.**

- 마찬가지로 fit\_transform()을 적용할 때도 주의가 필요함.
- fit\_transform()은 fit()과 transform()을 순차적으로 수행하는 메서드이므로 학습 데이터에서는 상관 없지만 테스트 데이터에서는 절대 사용해서는 안 됨.



학습과 테스트 데이터에 fit()과 transform()을 적용할 때 주의 사항이 발생하므로 **학습과 테스트 데이터로 분리하기 전에 먼저 전체 데이터셋에 스케일링을 적용한 뒤** 학습과 테스트 데이터셋으로 분리하는 것이 더 바람직하다.

## 유의할 점 요약

1. 가능하다면 전체 데이터의 스케일링 변환을 적용한 뒤 학습과 테스트 데이터로 분리
2. 1이 여의치 않다면 테스트 데이터 변환 시에는 fit()이나 fit\_transform()을 적용하지 않고 학습 데이터로 이미 fit() 된 Scaler 객체를 이용해 transform()으로 변환

## 평가

### 분류 - 성능 평가 지표(Evaluation Metric)

- 정확도(Accuracy)
  - 오차행렬(Confusion Matrix)
  - 정밀도(Precision)
  - 재현율(Recall)
  - F1 스코어
  - ROC AUC
- 분류는 결정 클래스 값 종류의 유형에 따라 긍정/부정과 같은 2개의 결괏값만을 가지는 이진 분류와 여러 개의 결정 클래스 값을 가지는 멀티 분류로 나눌 수 있음. 위의 분류의 성능지표는 두 분류 모두에 적용되는 지표이지만, 특히 이진 분류에서 더욱 중요하게 강조되는 지표다.

### 정확도(Accuracy)

- 실제 데이터에서 예측 데이터가 얼마나 같은지를 판단하는 지표 (맞은 예측의 수 / 전체 예측의 수)로 나타낼 수 있음.
- 하지만 불균형한 레이블 값 분포에서 ML 모델의 성능을 판단할 경우 부적절한 평가 지표임.
- 비대칭한 데이터 세트에서 Positive에 대한 예측 정확도를 판단하지 못한 채 Negative에 대한 예측 정확도만으로도 분류의 정확도가 매우 높게 나타나는 수치적인 판단 오류가 일어나기 때문.
- 이러한 한계를 극복하기 위해 여러 가지 분류 지표와 함께 적용하여 ML 모델 성능을 평가해야 하며, **오차 행렬**에 대해 이해해야 함.

```
[33] # 불균형한 레이블 데이터 분포도 확인.
      print('레이블 테스트 세트 크기 :', y_test.shape)
      print('테스트 세트 레이블 0 과 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)))
```

```
레이블 테스트 세트 크기 : (450,)
테스트 세트 레이블 0 과 1의 분포도
0      405
1       45
dtype: int64
모든 예측을 0으로 하여도 정확도는:0.900
```

불균형한 레이블 데이터에서 정확도의 오류

### 오차 행렬(confusion matrix, 혼동행렬)

		예측 클래스 (Predicted Class)	
		Negative(0)	Positive(1)
실제 클래스 (Actual Class)	Negative(0)	TN (True Negative)	FP (False Positive)
	Positive(1)	FN (False Negative)	TP (True Positive)

- 학습된 분류 모델이 예측을 수행하면서 얼마나 헷갈리고(confused) 있는지도 함께 보여주는 지표.
- 즉, 이진 분류의 예측 오류가 얼마인지와 더불어 어떠한 유형의 예측 오류가 발생하고 있는지를 함께 나타내는 지표.
- TN, FP, FN, TP는 예측 클래스와 실제 클래스의 Positive 결정 값(값 1)과 Negative 결정 값(값 0)의 결합에 따라 결정된다.
- 앞 문자 True/False는 예측값과 실제값이 '같은가/틀린가'를 의미하고 뒤 문자 Negative/Positive는 예측 결과 값이 부정(0)/긍정(1)을 의미한다.

1. TN은 예측값을 Negative 값인 0으로 예측했고 실제값 또한 Negative 값인 0일 때
2. FP은 예측값을 Positive 값인 1로 예측했는데 실제값은 Negative 값인 0일 때
3. FN은 예측값을 Negative 값인 0으로 예측했는데 실제값은 Positive 값인 1일 때
4. TP은 예측값을 Positive 값인 1로 예측했는데 실제값 역시 Positive 값인 1일 때

- 사이킷런은 오차 행렬을 구하기 위해 confusion\_matrix() API 를 제공한다.
- 출력된 오차 행렬은 ndarray 형태
- TP, TN, FP, FN 값은 Classifier 성능의 여러 면모를 판단할 수 있는 기반 정보를 제공하며, 이 값을 조합해 Classifier의 성능을 측정할 수 있는 주요 지표인 정확도(Accuracy), 정밀도(Precision), 재현율(Recall)을 알 수 있다.
- **정확도 = 예측 결과와 실제 값이 동일한 건수 / 전체 데이터 수 = (TN + TP) / (TN + FP + FN + TP)**

$$\frac{TP + TN}{TP + TN + FP + FN}$$

- 비대칭한 데이터셋에서 정확도는 판단 오류를 일으키기 때문에 정밀도와 재현율이 더 선호된다.

## 정밀도와 재현율

- 정밀도와 재현율은 Positive 데이터셋의 예측 성능에 좀 더 초점을 맞춘 평가 지표임.
- **정밀도 = TP / (FP + TP)**

- **재현율** =  $TP / (FN + TP)$

- **정밀도** : **예측을 Positive로 한 대상** 중에 예측과 실제 값이 Positive로 일치한 데이터의 비율을 뜻함. Positive 예측 성능을 더욱 정밀하게 측정하기 위한 평가 지표로 양성 예측도라고도 불림.
- **재현율** : **실제 값이 Positive인 대상** 중에 예측과 실제 값이 Positive로 일치한 데이터의 비율을 뜻함. 민감도 또는 TPR(True Positive Rate)라고도 불림.

- 정밀도와 재현율 지표 중에 이진 분류 모델의 업무 특성에 따라 특정 평가 지표가 더 중요한 지표로 간주될 수 있다.

- **재현율이 상대적으로 더 중요한 지표인 경우는 실제 Positive 양성 데이터를 Negative 음성으로 잘못 판단하게 되면 업무상 큰 영향이 발생하는 경우**

- ex) 암 판단 모델, 금융 사기 적발 모델

- 보통 재현율이 정밀도보다 상대적으로 중요 업무가 많지만 정밀도가 더 중요 지표인 경우도 있음.

- **정밀도가 상대적으로 더 중요한 지표인 경우는 실제 Negative 음성인 데이터 예측을 Positive 양성으로 잘못 판단하게 되면 업무상 큰 영향이 발생하는 경우**

- ex) 스팸메일 여부 판단 모델,

- 재현율과 정밀도 모두 TP를 높이는 데 동일하게 초점을 맞추지만, 재현율은 FN을 낮추는 데, 정밀도는 FP를 낮추는 데 초점을 맞춘다.
- 이 같은 특성 때문에 재현율과 정밀도는 서로 보완적인 지표로 분류의 성능 평가 지표로 사용됨.
- **가장 좋은 성능 평가는 재현율과 정밀도 모두 높은 수치를 얻는 것.**
- 어느 한 쪽이 매우 높고 다른 수치는 매우 낮은 결과는 바람직하지 않음.
- 사이킷런은 정밀도 계산을 위해 `precision_score()` 를, 재현율 계산을 위해 `recall_score()` 를 API로 제공한다.

## 정밀도/재현율 트레이드오프

- 분류하려는 업무의 특성상 정밀도 혹은 재현율이 특별히 강조돼야 할 경우 분류의 결정 임계값을 조정해 정밀도 또는 재현율의 수치를 높일 수 있다.
- 하지만 둘은 상호 보완적인 평가 지표이기 때문에 어느 한 쪽을 높이면 다른 하나의 수치가 떨어지기 쉽다. 이를 **정밀도/재현율의 트레이드오프**라고 부른다.

## 분류 결정 임계값이 낮아질수록 Positive로 예측할 확률이 높아짐

- 이진 분류에서는 기본 임계값이 0.5인데 0.5보다 크면 Positive로 예측
- 0.5보다 작으면 Negative로 예측

→ 분류 결정 임계값이 0.4로 낮아진다면? Positive로 예측할 확률이 높아짐

→ **재현율 증가 / 정밀도 낮아짐**

- 정밀도와 재현율 중에 더 중요한 지표가 무엇인지 판단하고 임계값을 조정한다.

## 정밀도와 재현율의 맹점

- Positive 예측의 임계값을 변경함에 따라 정밀도와 재현율의 수치가 변경되므로 임계값의 변경은 업무 환경에 맞게 두 개의 수치를 상호 보완할 수 있는 수준에서 적용돼야 함.
- 단순히 하나의 수치를 높이기 위한 수단으로 사용하면 안 됨.

$$Precision = \frac{TP}{FP + TP}$$

$$Recall = \frac{TP}{FN + TP}$$

## F1 스코어

- 어느 한쪽으로 치우치지 않고, 정밀도와 재현율을 결합한 지표
- 정밀도와 재현율이 어느 한쪽으로 치우치지 않는 수치를 나타낼 때 상대적으로 높은 값을 가짐.
- 정밀도와 재현율의 조화평균

$$F1\ Score = \frac{2}{\frac{1}{Recall} + \frac{1}{Precision}} = \frac{2 * Precision * Recall}{Precision + Recall}$$

## ROC 곡선과 AUC

- ROC 곡선과 이에 기반한 AUC 스코어는 이진 분류의 예측 성능 측정에서 중요하게 사용되는 지표
- ROC 곡선(Receiver Operation Characteristic Curve) : 수신자 판단 곡선
- ROC Curve는 FPR(False Postive Rate)이 변할 때, TPR (True Positive Rate)이 어떻게 변하는지 나타내는 곡선이다.
- FPR을 X축, TPR을 Y축으로 잡으면 FPR의 변화에 따른 TPR의 변화가 곡선으로 나타나게 됨.



## ROC 곡선

: FPR (False Positive Rate) 이 변할 때 TPR (True Positive Rate)이 어떻게 변하는지를 나타내는 곡선.

FPR을 x축, TPR을 y축으로 잡으면 FPR의 변화에 따른 TPR의 변화가 곡선 형태로 나타남.

$$\text{TPR} = \text{재현율} = \frac{\text{TP}}{(\text{FN} + \text{TP})} \begin{matrix} \leftarrow \text{예측값, 실제값 모두 positive} \\ \leftarrow \text{실제 positive 값} \end{matrix} = \text{민감도}$$

→ 민감도가 대응하는 지표로 TNR (True Negative Rate)으로 불리는 **특이성**이 있다.

• **민감도** : 실제값 positive (양성)가 정확히 예측되어야 하는 수준.

**TPR** (진병이 있는 사람은 진병이 있는 것으로 양성 판정)

• **특이성** : 실제값 Negative (음성)가 정확히 예측되어야 하는 수준.

**TNR** (진병이 없는 사람은 진병이 없는 것으로 음성 판정)

$$\text{특이성 (TNR)} = \frac{\text{TN}}{(\text{FP} + \text{TN})}$$

ROC 곡선의 x축 기준인 FPR은  $\frac{\text{FP}}{(\text{FP} + \text{TN})}$  이므로  $1 - \text{TNR}$  또는  $1 - \text{특이성}$ 으로 표현

$$\therefore \text{FPR} = \frac{\text{FP}}{(\text{FP} + \text{TN})} = 1 - \text{TNR}$$

- ROC 곡선이 가운데 직선에 가까울수록 성능이 떨어지는 것이고, 멀어질수록 성능이 뛰어난 것.
- ROC 곡선은 FPR을 0부터 1까지 변경하며 TPR의 변화 값을 구하는데, 이 FPR을 변경하는 방법은 **분류 결정 임계값**을 변경하면 된다.
- 분류 결정 임계값은 Positive 예측값을 결정하는 확률의 기준이므로
- FPR을 0으로 만들려면 → 임계값을 1로 지정하면 된다.**
- 임계값이 1이 되면 positive 예측 기준이 매우 높아서 분류기가 임계값보다 높은 확률을 가진 데이터를 positive로 예측할 수 없기 때문.
- 아예 양성으로 예측하지 않으므로 FP = 0, 따라서 FPR도 0이 된다.
- 반대로 FPR을 1로 만들기 위해선 → 임계값을 0으로 지정하면 된다.**
- positive 예측 허들이 너무 낮아서 모두 양성 판정을 한다.
- 아예 음성으로 예측하지 않으므로 TN = 0, FPR 값이 1이 된다.
- 사이킷런은 ROC 곡선을 구하기 위해 `roc_curve()` API를 제공함.
- 사용법은 `precision_recall_curve()` API와 유사함.

## AUC

- AUC (Area Under Curve) 값 : ROC 곡선 밑의 면적을 구한 것으로서 일반적으로 **1에 가까울수록 좋은 수치**.

- AUC 수치가 커지려면 **FPR(x축)이 작은 상태에서 얼마나 큰 TPR(y축)을 얻느냐가 관건.**
- 가운데 직선에서 멀어지고 왼쪽 상단 모서리 쪽으로 가파르게 곡선이 이동할수록 직사각형에 가까운 곡선이 되며 면적이 1에 가까워지는 좋은 ROC AUC 성능 수치를 얻게 된다.
- 가운데 대각선 직선은 랜덤 수준의 (동전 던지기 수준) 이진 분류 AUC 값으로 0.5이다. 따라서 보통의 분류는 0.5 이상의 AUC 값을 가지게 된다.

