Euron_ML

▼ 1. 파이썬 기반의 머신러닝과 생태계 이해

01. 머신러닝의 개념

> 애플리케이션을 수정하지 않고도 데이터를 기반으로 패턴을 학습하고 결과를 예측하는 알고리즘 기법

- 머신러닝의 분류

- 지도학습(Supervised Learning)
  - **분류**
  - **회귀**
  - 추천 시스템
  - 시각/음성 감지/인지
  - 텍스트 분석, NLP
- 비지도학습(Un-supervised Learning)
  - 클러스터링
  - 차원 축소
  - 강화학습
- 강화학습(Reinforcement Learning)

- 데이터 전쟁

: 데이터에 매우 의존적인 특성(Garbage in, Garbage out)

→ 데이터를 이해하고 효율적으로 가공, 처리, 추출해 최적의 데이터를 기반으로 알고리즘을 구동할 수 있도록 준비하는 능력이 중요

02. 파이썬 머신러닝 생태계를 구성하는 주요 패키지

- 머신러닝 패키지

: Scikit-Learn(데이터마이닝 기반의 머신러닝에서 독보적인 위치)

- 행렬/선형대수/통계 패키지
  - 넘파이(Numpy): 행렬과 선형대수를 다루는 대표적 패키지
  - 사이파이(SciPy): 자연과학과 통계를 위한 다양한 패키지
- 데이터 핸들링
  - 판다스

    : 대표적인 데이터 처리 패키지

    : 2차원 데이터에 처리에 특화

    : 맷플롯립(Matplotlib)을 호출해 쉽게 시각화 기능 지원
- 시각화
  - 맷플롯립(Matplotlib)

    : 대표적인 시각화 패키지

    <단점> 너무 세분화된 API, 시각적으로 투박, 작성해야 하는 코드가 과하게 길어져 효율이 떨어지고 불편함이 늘어남
  - 시본(Seaborn): 맷플롯립의 단점을 보완하기 위한 패키지 but 세밀한 제어는 맷플롯립의 API를 그대로 사용하기 때문에

    <장점> 판다스와의 쉬운 연동, 더 함축적인 API, 분석을 위한 다양한 유형의 그래프/차트 제공
- 서드파티 라이브러리

  : 파이썬 기반의 머신러닝을 편리하게 지원하기 위함
- 주피터 노트북

  : 대표적인 아이파이썬 툴 (대화형 파이썬 툴)→ 특정 코드 영역별로 개별 수행 지원

03. 넘파이

> 선형대수 기반의 프로그램을 쉽게 만들 수 있도록 지원하는 대표적인 패키지
>
> 루프를 사용하지 않고 대량 데이터의 배열 연산을 가능케 함

> 다양한 핸들링 기능 제공

- 넘파이 ndarray 개요

```
import numpy as np
```

: 넘파이 모듈 임포트→as np를 추가해 약어로 모듈을 표현

: ndarray는 넘파이 기반 데이터 타입→ 다차원 배열을 쉽게 생성하고 다양한 연산 수행

- **[np.array]**

```
# Input

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)

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

```
# Output

array1 type: <Class 'numpy.ndarray'>
array1 array 형태: (3, )
# 1차원 array로 3개의 데이터를 가지고 있음

array2 type: <Class 'numpy.ndarray'>
array2 array 형태: (2,3)
```

```
# 2차원 array로 2개의 로우와 3개의 칼럼으로 구성됨. 2*3=6개의 데이터를

array3 type: <Class 'numpy.ndarray'>
array3 array 형태: (1,3)
# 2차원 array로 1개의 로우와 3개의 칼럼으로 구성됨.
# array1은 명확하게 1차원임을 형태로 표현하였으므로 차이가 있다.
```

: array()함수는 다양한 인자를 입력 받아 ndarray로 변환

: ndarray의 shape변수는 ndarray의 크기(행과 열의 수)를 튜플 형태로 가짐→ 배열의 차원까지 알 수 있음

: 데이터값으로는 동일하나 차원이 달라서 발생하는 오류 주의 [reshpae() 함수 이용하여 해결]

- **[ndarray.ndim]**

```
    # Input

    print('array1: {:0}차원, array2: {:1}차원, array3: {:2}차원'
    array2.ndim, array3.ndim))
```

```
# Output

array1: 1차원, array2: 2차원, array3: 2차원
```

: 각 array의 차원을 알아볼 수 있는 함수

: array()함수의 인자는 파이썬의 리스트 객체가 주로 사용됨

- ndarray의 데이터 타입

: ndarray내의 데이터 타입은 같은 데이터 타입만 가능; 한 개의 ndarray 객체에 int와 float가 함께 있을 수 없음

: ndarray내의 데이터 타입은 dtype속성으로 확인

[같은 데이터 타입]

```
# Input

list1= [1,2,3]
print(type(list1))

array1= np.array(list1)
print(type(array1))

print(array1, array1.dtype)
```

```
# Output

<class 'list'>
<class 'numpy.ndarray'>
[1 2 3] int32
```

: list1은 리스트 자료형으로 숫자인 1,2,3을 값으로 가짐. 이를 ndarray로 변환한다면→ int32형

[다른 데이터 유형이 섞여 있는 리스트]

```
# Input

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)
```

```
# Output
```

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

: array2는 int형 값과 문자열이 섞여 있음. 이를 ndarray로 변환한다면→ 숫자형 1,2가 모두 문자열 값인 '1', '2' 로 변환됨

: ndarray는 모두 같은 데이터 타입이어야 하므로 서로 다른 데이터 타입이 섞여 있을 경우, *데이터 크기가 더 큰 데이터 타입으로 형 변환을 일괄 적용*

[ndarray 내 데이터값의 타입 변경: astype() 메서드 이용]

```
# Input

array_int= np.array([1,2,3])
array_float= array_int.astype('float64')
print(array_float, array_float.dtype)

array_int1= array_float.astype('int32')
print(array_int1, array_int1.dtype)

array_float1= np.array([1.1, 2.1, 3.1])
array_int2= array_float1.astype('int32')
print(array_int2, array_int2.dtype)
```

```
# Output

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

: astype()에 인자로 원하는 타입을 문자열로 지정

: 메모리 절약해야 할 때 이용

- ndarray를 편리하게 생성하기- arange, zeros, ones

> 특정 크기와 차원을 가진 ndarray를 연속값이나 0 또는 1로 초기화해 쉽게 생성해야 할 필요가 있는 경우
>
> 테스트용으로 데이터를 만들거나 대규모의 데이터를 일괄적으로 초기화해야 할 경우에 사용
>
> range()함수와 유사한 기능→ array를 range()로 표현하는 것으로 이해하기

```
# Input

sequence_array= np.arange(10)
print(sequence_array)
print(sequence_array.dtype, sequence_array.shape)
```

```
# Output

[0 1 2 3 4 5 6 7 8 9]
int64 (10,)
```

: default 함수 인자는 stop값→ 0부터 stop값인 10에서 -1을 더한 9까지의 연속 숫자 값으로 구성된 1차원 array 만듦 (range와 유사하게 start 값도 부여하여 다른 시작점을 가질 수 있음)

```
# Input

zero_array= np.zeros((3,2), dtype='int32')
print(zero_array)
print(zero_array.dtype, zero_array.shape)

one_array= np.ones((3,2))
```

```
print(one_array)
print(one_array.dtype, one_array.shape)
```

```
# Output

[[0 0]
 [0 0]
 [0 0]]
int32 (3, 2)
[[1. 1.]
 [1. 1.]
 [1. 1.]]
float64 (3, 2)
```

: zeros() 함수 인자로 튜플 형태의 shape 값 입력: 모든 값을 0으로 채운 해당 shape를 가진 ndarray를 반환

: 함수 인자로 dtype을 정해주지 않으면 float64형의 데이터로 채움(default)

- ndarray의 차원과 크기를 변경하는 reshape()

: ndarray를 특정 차원 및 크기로 변환하는 메서드

: 변환을 원하는 크기를 함수 인자로 부여

```
# Input

array1= np.arange(10)
print('array1:\n', array1)

array2= array1.reshape(2,5)
print('array2:\n', array2)

array3= array1.reshape(5,2)
print('array3:\n', array3)
```

```
# Output

array1:
 [0 1 2 3 4 5 6 7 8 9]
array2:
 [[0 1 2 3 4]
 [5 6 7 8 9]]
array3:
 [[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
```

: 1차원 ndarray를 2*5, 5*2 형태의 2차원 ndarray로 변환

```
# Input

array1.reshape(4,3)
```

```
# Output

ValueError                                Traceback (most rec
<ipython-input-7-a40469ec5825> in <cell line: 1>()
----> 1 array1.reshape(4,3)

ValueError: cannot reshape array of size 10 into shape (4,3)
```

: reshape()는 지정된 사이즈로 변경이 불가능하면 오류 발생

```
# Input
```

```
array1= np.arange(10)
print(array1)
array2= array1.reshape(-1,5)
# array1과 호환될 수 있는 2차원 ndarray로 변환하되, 고정된 5개의 칼럼에
print('array2 shape:', array2.shape)
array3= array1.reshape(5,-1)
# array1과 호환될 수 있는 2차원 ndarray로 변환하되, 고정된 5개의 로우에
print('array3 shape:', array3.shape)
```

```
# Output

[0 1 2 3 4 5 6 7 8 9]
array2 shape: (2, 5)
array3 shape: (5, 2)
```

: 인자로 -1을 적용하는 경우→ 원래 ndarray와 호환되는 새로운 shape로 변환

```
# Input

array1= np.arange(10)
array4= array1.reshape(-1,4)
```

```
# Output

ValueError                                Traceback (most rec
<ipython-input-9-5c7ca6326d7e> in <cell line: 2>()
      1 array1= np.arange(10)
----> 2 array4= array1.reshape(-1,4)

ValueError: cannot reshape array of size 10 into shape (4)
```

: 호환될 수 없는 형태는 변환 불가 (10개의 1차원 데이터를 고정된 4개의 칼럼을 가진 로우로는 변경할 수 없음!)

```
# Input

array1= np.arange(8)
array3d= array1.reshape((2,2,2))
print('array3d:\n', array3d.tolist())

# 3차원 ndarray를 2차원 ndarray로 변환
array5= array3d.reshape(-1,1)
print('array5:\n', array5.tolist())
print('array5 shape:', array5.shape)

# 1차원 ndarray를 2차원 ndarray로 변환
array6= array1.reshape(-1,1)
print('array6:\n', array6.tolist())
print('array6 shape:', array6.shape)
```

```
# Output

array3d:
 [[[0, 1], [2, 3]], [[4, 5], [6, 7]]]
array5:
 [[0], [1], [2], [3], [4], [5], [6], [7]]
array5 shape: (8, 1)
array6:
 [[0], [1], [2], [3], [4], [5], [6], [7]]
array6 shape: (8, 1)
```

: reshape(-1,1)은 원본 ndarray가 어떤 형태라도 2차원이고, 여러 개의 로우를 가지되 반드시 1개의 칼럼을 가진 ndarray로 변화됨을 보장

: stack이나 concat으로 결합할 때 형태를 통일해주어 유용하게 사용

: tolist() 메서드를 이용해 리스트 자료형으로 변환

- 넘파이의 ndarray의 데이터 세트 선택하기- 인덱싱(Indexing)

> ndarray내의 일부 데이터 세트나 특정 데이터만을 선택할 수 있도록 함

[특정한 데이터만 추출]

💡 원하는 위치의 인덱스 값을 지정하면 해당 위치의 데이터 반환

- 단일 값 추출

> ndarray 객체에 해당하는 위치의 인덱스 값을 [ ] 안에 입력

```
# Input

array1= np.arange(start=1, stop=10)
print('array1:', array1)

value= array1[2]
print('value:', value)
print(type(value))
```

```
# Output

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

: 인덱스는 0부터 시작하므로 array1[2]는 3번째 인덱스 위치의 데이터값을 의미

```
# Input

print('맨 뒤의 값:', array1[-1], '맨 뒤에서 두 번째 값:', array
```

```
# Output

맨 뒤의 값: 9 맨 뒤에서 두 번째 값: 8
```

: 인덱스 -1은 맨 뒤의 데이터값 의미

```
# Input

array1[0]= 9
array1[8]= 0
print('array1:', array1)
```

```
# Output

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

: 단일 인덱스를 이용한 ndarray내의 데이터값 수정

- 다차원 ndarray에서 단일 값 추출

> 콤마(,)로 분리된 로우와 칼럼 위치의 인덱스를 통해 접근

```
# Input

```

```
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])
```

```
# Output

[[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
```

: row와 col은 axis=0, axis=1을 의미

: 축 기반의 연산에서 axis가 생략되면 axis 0을 미

[슬라이싱 (Slicing)]

💡 연속된 인덱스상의 ndarray 추출

```
# Input

[[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
```

```
# Output
[1 2 3]
<class 'numpy.ndarray'>
```

: ':' 기호 사이에 시작 인덱스와 종료 인덱스 표시→ 시작 인덱스~종료 인덱스 -1 위치에 있는 데이터의 ndarray 반환

```
# Input

array1= np.arange(start=1, stop=10)
array4= array1[:3]
print(array4)

array5= array1[3:]
print(array5)

array6= array1[:]
print(array6)
```

```
# Output

[1 2 3]
[4 5 6 7 8 9]
[1 2 3 4 5 6 7 8 9]
```

: 슬라이싱 기호인 ':' 사이의 시작, 종료 인덱스는 생략이 가능

— 시작 인덱스 생략→ 맨 처음 인덱스 0으로 간주

— 종료 인덱스 생략→ 맨 마지막 인덱스로 간주

— 시작/종료 인덱스 생략→ 처음/맨 마지막 인덱스로 간주

- 2차원 슬라이싱

```
# Input

array1d= np.arange(start=1, stop=10)
array2d= array1d.reshape(3,3)
print('array2d:\n', array2d)

print('array2d[0:2, 0:2]\n', array2d[0:2, 0:2])
print('array2d[1:3, 0:3]\n', array2d[1:3, 0:3])
print('array2d[1:3, :]\n', array2d[1:3, :])
print('array2d[:, :]\n', array2d[:,:])
print('array2d[:2, 1:]\n', array2d[:2, 1:])
print('array2d[:2, 0]\n', array2d[:2, 0])
```

```
# Output

array2d:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
array2d[0:2, 0:2]
 [[1 2]
 [4 5]]
array2d[1:3, 0:3]
 [[4 5 6]
 [7 8 9]]
array2d[1:3, :]
 [[4 5 6]
 [7 8 9]]
array2d[:, :]
```

```
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
array2d[:2, 1:]
 [[2 3]
 [5 6]]
array2d[:2, 0]
 [1 4]
```

: 콤마로 로우와 칼럼 인덱스를 지칭하는 부분이 다름

```
# Input

print(array2d[0])
print(array2d[1])
print('array2d[0] shape:', array2d[0].shape, 'array2d[1] shap
```

```
# Output

[1 2 3]
[4 5 6]
array2d[0] shape: (3,) array2d[1] shape: (3,)
```

: 2차원 ndarray에서 뒤에 오는 인덱스를 없애면 1차원 ndarray를 반환한

[팬시 인덱싱 (Fancy Indexing)]

💡 일정한 인덱싱 집합을 리스트 또는 ndarray 형태로 지정해 해당 위치에 있는 데이터의 ndarray 반환

```
# Input

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

array3= array2d[[0,1], 2]
print('array2d[[0,1], 2] => ', array3.tolist())
: 로우 축에 팬시 인덱싱, 칼럼 축에 단일 값 인덱싱 2 적용→ 인덱스가 (0,2)

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

array5= array2d[[0,1]]
print('array2d[[0,1]] => ', array5.tolist())
: 인덱스가 ((0,:),(1,:))로 적용됨
```

```
# Output

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

: 로우 축에 팬시 인덱싱, 칼럼 축에 단일 값 인덱싱 2 적용→ 인덱스가 (0,2), (1,2)로 적용

[불린 인덱싱 (Boolean Indexing)]

💡 특정 조건에 해당하는지 여부인 True/False 값 인덱싱 집합을 기반으로 True에 해당하는 인덱스 위치에 있는 데이터의 ndarray 반환

조건 필터링과 검색을 동시에 할 수 있음

```
# Input

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

```
# Output

array1d>5 불린 인덱싱 결과 값: [6 7 8 9]
```

: 간단하게 조건 필터링이 가능하다

[ndarray 객체에 조건식을 할당하는 경우]

```
# Input

array1d>5
```

```
# Output

array([False, False, False, False, False,  True,  True,  True
```

: True로 이뤄진 ndarray객체 반환

: 조건에 맞는 값→ True, 그렇지 않은 경우→ False

[불린 인덱싱의 조건으로 반환된 ndarray객체를 [ ]내에 입력]

```
# Input

boolean_indexes= np.array([False, False, False, False, False,
```

```
array3= array1d[boolean_indexes]
print('불린 인덱스로 필터링 결과:', array3)
```

```
# Output

불린 인덱스로 필터링 결과: [6 7 8 9]
```

: True값이 있는 위치 인덱스 값으로 자동 변환해 해당하는 인덱스 위치의 데이터만 반환됨

[직접 인덱스 집합을 만들어 대입]

```
# Input

indexes= np.array([5,6,7,8])
array4= array1d[indexes]
print('일반 인덱스로 필터링 결과:', array4)
```

```
# Output

일반 인덱스로 필터링 결과: [6 7 8 9]
```

: 위의 결과와 동일

- 불린 인덱싱의 동작 단계

1. ndarray의 필터링 조건을 [ ] 안에 기재 (array1d>5)
2. True 값에 해당하는 인덱스값만 저장(값 자체인 1을 저장하는 것이 아니라, True값을 가진 인덱스를 저장)
3. 저장된 인덱스 데이터 세트로 ndarray 조회

⇒ 단순히 [ ] 내에 원하는 필터링 조건만 넣으면, 해당 조건을 만족하는 ndarray데이터 세트를 반환

- 행렬의 정렬 - sort()와 argsort()

[행렬 정렬]

> np.sort()_ 넘파이에서 호출 vs. ndarray.sort()_ 행렬 자체에서 호출

<np.sort()>

: 원 행렬은 유지한 채 원 행렬의 정렬된 행렬 변환

<ndarray.sort()>

: 원 행렬 자체를 정렬한 형태로 변환하며 변환 값은 None

```
# Input

org_array= np.array([3,1,9,5])
print('원본 행렬:', org_array)

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)

sort_array2= org_array.sort()
print('org_array.sort() 호출 후 반환된 행렬:', sort_array2)
print('org_array.sort() 호출 후 원본 행렬:', org_array)
```

```
# Output

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

: np.sort()는 원본 행렬을 변경하지 않고 정렬된 형태로 반환, ndarray.sort()는 원본 행렬 자체를 정렬한 값으로 변환

```
# Input

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

```
# Output

내림차순으로 정렬: [9 5 3 1]
```

: 모두 기본적으로 오름차순으로 행렬 내 원소를 정렬하므로, 내림차순으로 정렬하기 위해서는 [::-1] 적용

```
# Input

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)
```

```
# Output

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

```
 [[ 8 12]
 [ 1  7]]
```

: 행렬이 2차원 이상일 경우에 axis축 값 설정을 통해 로우/칼럼 방향으로 정렬 수행 가능

[정렬된 행렬의 인덱스를 반환하기]

> 기존 원본 행렬의 원소에 대한 인덱스를 필요로 할 때 np.argsort() 이용
>
> np.argsort(): 정렬 행렬의 원본 행렬 인덱스를 ndarray형으로 반환

```
# Input

org_array= np.array([3,1,9,5])
sort_indices= np.argsort(org_array)
print(type(sort_indices))
print('행렬 정렬 시 원본 행렬의 인덱스:', sort_indices)
```

```
# Output

<class 'numpy.ndarray'>
행렬 정렬 시 원본 행렬의 인덱스: [1 0 3 2]
```

: 원본 행렬의 정렬 시 행렬 인덱스 값 구하기

```
# Input

org_array=np.array([3,1,9,5])
```

```
sort_indices_desc= np.argsort(org_array)[::-1]
print('행렬 내림차순 정렬 시 원본 행렬의 인덱스:', sort_indices_desc
```

```
# Output

행렬 내림차순 정렬 시 원본 행렬의 인덱스: [2 3 0 1]
```

: 내림차순으로 정렬 시 원본 행렬 인덱스 구하기→ np.argsort( ) [::-1]

```
# Input

name_array= np.array(['John', 'Mike', 'Sarah', 'Kate', 'Samue
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_a
print('성적 오름차순으로 name_array의 이름 출력:', name_array[sort_
```

```
# Output

성적 오름차순 정렬 시 score_array의 인덱스: [0 2 4 1 3]
성적 오름차순으로 name_array의 이름 출력: ['John' 'Sarah' 'Samuel'
```

: ndarray는 TABLE, DATAFRAME 칼럼과 같은 메타 데이터를 가질 수 없기에, 실제 값과 그 값이 뜻하는 메타 데이터를 별도의 ndarray로 가져야 한다.

- 선형대수 연산- 행렬 내적과 전치 행렬 구하기

[행렬 내적(행렬 곱)]

> np.dot( ):두 행렬 A와 B의 내적, 즉 왼쪽 행렬의 로우와 오른쪽 행렬의 칼럼의 원소들을 순차적으로 곱한 뒤 결과를 모두 더함

```
# Input

A= np.array([[1,2,3],
             [4,5,6]])
B= np.array([[7,8],
             [9,10],
             [11,12]])

dot_product= np.dot(A,B)
print('행렬 내적 결과:\n', dot_product)
```

```
# Output

행렬 내적 결과:
 [[ 58  64]
 [139 154]]
```

[전치 행렬]

> 원 행렬에서 행과 열 위치를 교환한 원소로 구성한 행렬
>
> ex) A 행렬의 원소가 (1,2) 라면 이를 (2,1)로 교환
>
> transpose() 이용

```
# Input

A= np.array([[1,2],
             [3,4]])
transpose_mat= np.transpose(A)
print('A의 전치 행렬:\n', transpose_mat)
```

```
# Output

A의 전치 행렬:
 [[1 3]
 [2 4]]
```

04. 데이터 핸들링 - 판다스

- 판다스 시작- 파일을 DatatFrame으로 로딩, 기본 API

- 캐글 데이터 파일 내려받기 위한 사전 작업
  - 로그인
  - 경연 참가 버튼 클릭 후 데이터 내려받기
  - read_csv(filepath_or_buffer, sep=', ', ...)

    : filepath인자 가장 중요
- csv 파일을 dataframe으로 로딩한 후, 로딩된 dataframe의 출력값 확인하기
  - 데이터 확인
    - pd.read_csv( ): 파일명 인자로 들어온 파일을 로딩해 DataFrame 객체로 반환
    - DataFrame.head( ): DataFrame의 맨 앞 N개의 로우를 반환
  - 메타 데이터 등 조회 가능 (칼럼의 타입, Null 데이터 개수, 데이터 분포도 등)
    - info( ): 총 데이터 건수, 데이터 타입, Null 건수
    - describe( )

      : 칼럼별 숫자형 데이터값의 n-percentile분포도, 평균값, 최댓값, 최솟값

      : 오직 숫자형(int, float 등) 칼럼의 분포도만 조사 → object 타입의 칼럼은 출력에서 제외시킴
    - value_counts( )

      : 지정된 칼럼의 데이터값 건수 반환

: DataFrame의 [ ] 연산자 내부에 칼럼명 입력→Series 형태로 특정 칼럼 데이터 세트 반환

: Series 객체에서만 정의돼 있음

: 인덱스는 단순히 0부터 시작하는 순차 값이 아님. 고유성이 보장된다면 의미 있는 데이터값 할당도 가능

- DataFrame과 리스트, 딕셔너리, 넘파이 ndarray 상호 변환

> DataFrame이 2차원 데이터이기 때문에 2차원 이하의 데이터들만 DataFrame으로 변환 가능

[1차원 형태의 리스트와 넘파이 ndarray 변환]

: 1차원 형태의 데이터를 기반으로 DataFrame을 생성하므로 칼럼명은 한 개만 필요

[2차원 형태의 리스트와 넘파이 ndarray 변환]

: 2행 3열 형태의 리스트와 ndarray를 기반으로 생성하므로 칼럼명은 3개 필요

[딕셔너리를 DataFrame으로 변환]

: 딕셔너리의 키(Key)는 칼럼명으로, 딕셔너리의 값(Value)는 키에 해당하는 칼럼 데이터로 변환

: 키의 경우는 문자열, 값의 경우는 리스트(또는 ndarray)형태로 딕셔너리를 구성

[DataFrame을 넘파이 ndarray, 리스트, 딕셔너리로 변환하기]

- 넘파이 ndarray로 변환

> 머신러닝 패키지의 입력 인자 등에 적용하기 위함→ DaraFrame객체의 values를 이용

- 리스트와 딕셔너리로 변환

> values로 얻은 ndarray에 tolist( )를 호출

> 딕셔너리로의 변환은 to_dict( ) 메서드를 호출
>
> 인자로 'list'를 입력하면 딕셔너리의 값이 리스트형으로 반환됨

- DataFrame의 칼럼 데이터 세트 수정

> [ ]연산자를 이용해 쉽게 할 수 있음

[새로운 칼럼 Age_0을 추가하고 일괄적으로 0값을 할당]

: DataFrame [ ]내에 새로운 칼럼명을 입력하고 값을 할당하기

: titanic_df['Age_0']=0과 같이 Series에 상숫값을 할당하면 Series의 모든 데이터 세트에 일괄적으로 적용됨

[기존 칼럼 Series의 데이터를 이용해 새로운 칼럼 Series만들기]

[DataFrame 내의 기존 칼럼 값 일괄적으로 업데이트]

: 업데이트를 원하는 칼럼 Series를 DataFrame[ ] 내에 칼럼 명으로 입력한 뒤에 값을 할당해 주기

- DataFrame 데이터 삭제

> drop( ) 메서드 이용
>
> 원형: DataFrame.dtop(**labels=None, axis=0**, index=None, columns=None, level=none, **inplace=False**, errors='raise')

→ labels, axis, inplace 중요 파라미터

- axis 값에 따라 특정 칼럼 또는 특정 행을 드롭

  : 원하는 칼럼/로우 축 방향으로 드롭을 수행하겠다

- inplace는 디폴트 값이 Fasle이므로 파라미터를 기재하지 않으면 자동으로 False가 됨

  : inplace=False이면 자기 자신의 DataFrame의 데이터는 삭제하지 않으며, 삭제된 결과 DataFrame만 반환함

  : **<원본 DataFrame에서 드롭된 결과를 적용할 경우>** DataFrame의 데이터를 진짜 삭제하고 싶으면 inplace=True로 설정하기! → 반환 값은 None이 됨

- Index 객체

> DataFrame.index 또는 Series.index→ DataFrame, Series에서 Index객체만 추출

- 식별성 데이터를 1차원 array로 가지고 있음, ndarray와 유사하게 단일 값 반환 및 슬라이싱도 가능
- 한 번 만들어진 DataFrame 및 Series의 Index객체는 함부로 변경할 수 없음
- Series 객체는 Index객체를 포함하지만, Series 객체에 연산 함수를 적용할 때 Index는 연산에서 제외됨 → Index는 오직 식별용으로만 사용

> reset_index()메서드

: 새롭게 인덱스를 연속 숫자 형으로 할당, 기존 인덱스는 'index'라는 새로운 칼럼 명으로 추가함

: 연속된 int 숫자형 데이터가 아닐 경우에 다시 이를 연속 int숫자형 데이터로 만들 때 주로 사용

: drop=True로 설정하면 기존 인덱스는 새로운 칼럼으로 추가되지 않고 삭제됨 (새로운 칼럼이 추가되지 않으므로 Series는 유지된다)

- 데이터 셀렉션 및 필터링

> 넘파이와의 다른 기능 주의

> 넘파이: '[ ]' 연산자 내 단일 값 추출, 슬라이싱, 팬시 인덱싱, 불린 인덱싱
>
> 판다스: ix[ ], iloc[ ], loc[ ]

[DataFrame의 [ ] 연산자]

- [ ]→ 넘파이와 DataFrame 간 유의해야 할 부분
  - 넘파이의 [ ]: 행의 위치, 열의 위치, 슬라이싱 범위 등을 지정해 데이터 가져옴
  - DataFrame 의 [ ]: 안에 칼럼 명 문자, 인덱스로 변화 가능한 표현식이 들어갈 수 있음

    ⇒ [ ]: 칼럼만 지정할 수 있는 '칼럼 지정 연산자'

- 여러 개의 칼럼에서 데이터를 추출

  : ['칼럼1', '칼럼2']와 같이 리스트 객체 이용

  : DataFrame뒤의 [ ] 에는 칼럼명을 지정해야 하는 것 주의!→ 안에 숫자 인덱스를 사용하면 오류 발생

- 판다스의 인덱스 형태로 변환 가능한 표현식

  : [ ]내에 입력 가능 (예를 들면 슬라이싱; 사용하지 않는 걸 권장)

  : 불린 인덱싱 가능

[DataFrame ix[ ]연산자]

ix[0, 'Pclass']: 칼럼 명칭(label) 기반 인덱싱→ 사라짐 → loc[ ]

ix[0, 2]: 칼럼 위치(position) 기반 인덱싱 → iloc[ ]

- ix[ ] 연산자

  : 단일 지정, 슬라이싱, 불린 인덱싱, 팬시 인덱싱 모두 가능

  - ix[ ]연산자 인덱싱 방법으로 데이터 추출하기

ix[ ] 연산 유형	설명
data_df.ix[0, 0]	첫 번째 로우, 첫 번째 칼럼 위치에 있는 단일 값 반환
data_df.ix['one', 0]	인덱스 'one'에 해당하는 로우, 첫 번째 칼럼 위치의 단일 값 반환
data_df.ix[3, 'Name']	네 번째 로우, 칼럼명 Name 위치에 있는 단일 값 반환
data_df.ix[0:2, [0, 1]]	0:2 슬라이싱 (0,1) 범위의 로우와 첫번째, 두번째 칼럼 범위에 해당하는 DataFrame 반환
data_df.ix[0:2, [0:3]]	0:2 슬라이싱 범위의 로우와 0:3 슬라이싱 칼럼 범위에 해당하는 DataFrame 반환
data_df.ix[0:3, ['Name', 'Year']]	0:3 슬라이싱 범위의 로우와 Name, Year칼럼 범위에 해당하는 DataFrame 반환
data_df.ix[:]	전체 DataFrame 반환
data_df.ix[:, :]	전체 DataFrame 반환
data_df.ix[data_df.Year >= 2014]	Year 칼럼 값이 2014보다 크거나 같은 로우 인덱스를 가지는 DataFrame 반환

[명칭 기반 인덱싱과 위치 기반 인덱싱의 구분]

> 명칭 (Label) 기반 인덱싱: 칼럼의 명칭을 기반으로 위치 지정
>
> 위치 (Position) 기반 인덱싱: 0을 출발점으로 하는 가로축, 세로축 좌표 기반의 행과 열 위치를 기반으로 데이터 지정

- DataFrame의 인덱스값은 명칭 기반 인덱싱으로 간주
- ix[ ]의 경우 행과 열 위치에 명칭과 위치 기반 인덱싱 모두를 허용 → integer일 때 코드 작성에 혼선 초래 우려 → iloc[],loc[]

[DataFrame iloc[ ] 연산자]

: 위치 기반 인덱싱 허용

: 행과 열 값으로 integer 또는 integer형의 슬라이싱, 팬시 리스트 값 입력

: 위치 인덱싱이 아닌 명칭 또는 문자열 인덱스를 입력하면 오류 발생

[DataFrame loc[ ] 연산자]

: 명칭 기반 데이터 추출

: 행 위치에는 DataFrame index 값을, 열 위치에는 칼럼 명을 입력

: 무조건 문자열을 입력한다는 선입견은 위험→ reset_index의 예처럼 새로운 정수형 인덱스값으로 변경하면 output 도출

: loc[ ]에 슬라이싱 기호를 적용할 때 유의할 점 → 종료값-1이 아니라 종료 값까지 포함하는 것을 의미 (명칭은 숫자형이 아닐 수 있기 때문에 -1을 할 수가 없다)

[불린 인덱싱]

: 가져올 값을 조건으로 입력하면 자동으로 원하는 값을 필터링한다는 장점

: 반환된 객체의 타입은 DataFrame → 칼럼 명만 별도로 추출 가능

: 여러 개의 복합 조건 결합 적용 가능

1. and 조건→ &
2. or 조건→ |
3. Not 조건→ ~

- 정렬, Aggregation 함수, GroupBy 적용

[DataFrame, Series의 정렬 - sort_values()]

> sort_values()의 주요 파라미터: by, ascending, inplace

- by

  : 특정 칼럼 입력→ 해당 칼럼으로 정렬 수행

- ascending

  : ascending=True(기본)→ 오름차순 정렬

  :: ascending=False→ 내림차순 정렬

- inplace

: inplace=False(기본)→ sort_value()를 호출한 DataFrame은 그대로 유지하며 정렬된 DataFrame을 결과로 반환

: inplace=True→ 호출한 DataFrame의 정렬 결과를 그대로 적용

[Aggregation 함수 적용]

> min(), max(), sum(), count()

: DataFrame에서 aggregation을 호출할 경우 모든 칼럼에 해당 aggregation이 적용됨

: 대상 칼럼들만 추출해 aggregation 함수 적용→ 특정 칼럼에 aggregation함수 적용

[groupby() 적용]

: 입력 파라미터 by에 칼럼 입력→ 대상 칼럼으로 groupby됨

: DataFrame에 groupby()호출→ DataframeGroupBy라는 또 다른 형태의 DataFrame반환

- groupby() 대상 칼럼을 제외한 모든 칼럼에 해당 aggregation 함수 적용

  : DataFrame에 groupby() 호출해 반환된 결과에 aggregation 함수 호출
- DataFrame의 groupby()에 특정 칼럼만 aggregation 함수 적용

  : groupby()로 반환된 DataFrameGroupBy 객체에 해당 칼럼을 필터링한 뒤 aggregation 함수적용
- 서로 다른 aggregation 함수 적용

  : 적용하려는 여러 개의 aggregation 함수명을 DataFrameGroupBy 객체의 agg() 내에 인자로 입력해서 사용

  : agg() 내에 입력 값으로 딕셔너리 형태로 aggregation이 적용될 칼럼들과 aggregation 함수를 입력

- 결손 데이터 처리하기

> 결손데이터: 칼럼에 값이 없는, 즉 NULL인 경우 의미

> 넘파이의 NaN으로 표시
>
> → NaN값을 처리하지 않으므로&함수 연산 시 제외되므로 이 값을 다른 값으로 대체해야 함

[isna()로 결손 데이터 여부 확인]

> isna(): 데이터가 NaN인지 아닌지를 알려줌 (NaN 여부를 확인하는 API)
>
> → True나 False로 알려줌

- 결손데이터 개수 구하기

  : sum()함수 추가 → True는 내부적으로 숫자 1로, False는 숫자 0으로 변환

[fillna()로 결손 데이터 대기]

> fillna(): 결손 데이터를 편리하게 다른 값으로 대체 (NaN값을 다른 값으로 대체하는 API)

- (주의) 실제 데이터 세트 값을 변경하려면?

  : fillna()를 이용해 반환 값을 다시 받기

  : inplace=True 파라미터를 fillna()에 추가

- apply lambda 식으로 데이터 가공

> 복잡한 데이터 가공이 필요한 경우
>
> lambda식: 파이썬에서 함수형 프로그래밍을 지원하기 위함(함수의 선언과 함수 내의 처리를 한 줄의 식으로 쉽게 변환)

lambda x: x**2→ ':'로 입력 인자와 반환될 입력 인자의 계산식을 분리

입력인자 : 입력 인자의 계산식

- 여러 개의 값을 입력 인자로 사용해야 할 경우

  : map()함수 결합하여 사용
- 조금 더 복잡한 가공

  : if else절 사용

  → if절의 경우 if식보다 반환 값을 먼저 기술해야 함

  (틀린 식) x : if x <=15 ‘Child’ else ‘Adult

  (맞는 식) lambda x : ‘Child’ if x<=15 else ‘Adult’
- if,else만 지원

  : if, else if, else와 같이 else if는 지하지 않음
- else if가 많이 나와야 하는 경우나 switch cas문의 경우

  : else를 계속 내포해서 쓰는 게 아니라, 별도의 함수를 만드는 게 더 낫다