Machine Learning

▼ Chapter 1.1-1.2 | About Machine Learning

1. **머신러닝의 분류**
    a. 지도학습(Supervised Learning)
        - 분류(Classification)
        - 회귀(Regression)
        - 추천 시스템
        - 시각/음성 감지/인지텍스트 분석, NLP

    b. 비지도학습(Un-supervised Learning)
        - 클러스터링
        - 차원 축소
        - 강화학습

    c. 강화학습(Reinforcement Learning)
2. **데이터**
    a. 머신러닝은 데이터에 매우 의존적: Garbage In Garbage out
    b. 최적의 머신러닝 알고리즘과 모델 파라미터 구축 능력도 중요하지만, 데이터를 이해하고 효율적으로 가공, 처리, 추출하는 것이 중요
    c. 많은 회사의 경쟁력은 어떠한 품질의 데이터로 만든 머신러닝 모델이냐에 따라 결정될 수 있음
3. **파이썬 기반 머신러닝**
    a. 파이썬과 R을 머신러닝에 주로 사용
        - 타 언어보다 지원 패키지 및 생태계가 활발하기 때문
        - 타 언어는 개발 생산성이 떨어짐
        - TensorFlow, Keras, PyTorch 프레임워크 지원

4. **파이썬 머신러닝 주요 패키지**

    a. 머신러닝 패키지

    - 사이킷런(Scikit-Learn)
      비정형 데이터 분야에서 텐서플로, 케라스 등 라이브러리가 각광받고 있지만 사이킷런의 경우 여전히 데이터마이닝 기반의 머신러닝에서 독보적 위치를 차지하고 있음

    b. 행렬/선형대수/통계 패키지

    - 선형대수와 통계는 머신러닝의 이론적 백그라운드
    - NumPy
      행렬과 선형대수를 다루는 패키지
    - SciPy
      자연과학과 통계를 위한 패키지
    - Scikit-Learn 또한 SciPy패키지를 기반으로 구축된 여러 패키지를 가지고 있음

    c. 데이터 핸들링

    - Pandas
      파이썬의 대표적 데이터 처리 패키지 넘파이는 행렬 기반 데이터 처리에 특화되어 있어 일반적 데이터 처리에 부족한 반면 판다스는 2차원 데이터 처리에 특화되어 있어 편리
      Matplotlib을 호출하여 시각화도 가능

    d. 시각화

    - Matplotlib
      파이썬의 대표적 시각화 패키지
      세분화된 API로 익히기에는 어려운 편
      Matplotlib을 보와하기 위해 Seaborn을 사용하는 경우도 많음
      하지만 여전히 세밀한 부분의 제어는 Matplotlib의 API를 그대로 사용

▼ Chapter 1.3 | Basic NumPy for ML

1. **NumPy**

    - Numerical Python을 의미
    - 파이썬에서 선형대수 기반의 프로그램을 쉽게 만들 수 있도록 지원
    - 루프를 사용하지 않고 대량 데이터의 배열 연산을 가능하게 하므로 빠른 배열 연산 속도 보장
    - 하지만 Pandas를 이해하기 위해 Numpy는 필수적
    - 많은 머신러닝 알고리즘이 넘파이 기반으로 작성되어 있으며, 이들 알고리즘의 입력 데이터와 출력 데이터를 넘파이 배열 타입으로 사용
    - 편의성과 다양한 API 지원 측면에서 아쉬운 부분이 있기에 본 교재에서는 Pandas 데이터 프레임을 주로 사용

2. **ndarray**

    - 넘파이의 기본 데이터 타입은 ndarray
        - ndarray를 통해 넘파이에서 다차원 배열을 쉽게 생성하고 연산을 수행할 수 있다
    - ndarray 생성 및 shape
        - array()함수는 파이썬 리스트와 같은 다양한 인자를 입력받아 ndarray로 변환하는 기능 수행
        - 생성된 ndarray배열의 shape 변수는 ndarray의 크기(행과 열 수를 튜플 형태)를 튜플 형태로 가지고 있으며 이를 통해 ndarray 배열의 차원까지 알 수 있다

📓 실습 1.1 - ndarray 생성 및 shape 확인

```
[소스코드]
import numpy as np

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)
```

```
[결과]
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)
```

- 실습 1.1 결과 풀이
  - np.array()를 사용하기 위해 ndarray로 변환을 원하는 객체를 인자로 입력하면 ndarray를 반환
  - ndarray.shape는 ndarray의 차원과 크기를 tuple 형태로 나타낸다
  - array1 - [1, 2, 3]
    shape는 (3,)
    되며 이는 1차원 array로 3개의 데이터 존재
  - array2 - [[1, 2, 3], [2, 3, 4]]
    shape는 (2, 3)
    2차원 array, 2개의 로우와 3개의 컬럼으로 구성
    2*3=6개의 데이터 존재
  - array3 - [[1, 2, 3]]
    shape는 (1, 3)
    1개의 로우, 3개의 칼럼으로 구성된 2차원 데이터
- 데이터 값으로는 서로 동일하나 차원이 달라서 오류가 발생하는 경우가 빈번한 만큼 명확히 차원의 차수를 변환하는 방법을 알아야 오류를 막을 수 있다

📓 실습 1.2 - ndim을 통해 shape 확인

```
[소스코드]
import numpy as np

array1 = np.array([1, 2, 3])

array2 = np.array([[1, 2, 3],
                   [2, 3, 4]])

array3 = np.array([[1, 2, 3]])

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

```
[결과]
array1: 1차원, array2: 2차원, array3:  2차원
```

- array()함수의 인자로는 파이썬의 리스트 객체가 주로 사용
  리스트 []는 1차원, 리스트의 리스트 [[]]는 2차원과 같은 형태로 배열의 차원과 크기를 쉽게 표현할 수 있기 때문

3. ndarray의 DataType

- ndarray내의 데이터 값은 숫자, 문자열, 부울값 모두 가능, 정밀도를 위해 complex 타입도 제공
- ndarray 내의 데이터 타입은 연산 특성상 같은 데이터 타입만 가능
- ndarray내의 데이터 타입은 dtype 속성으로 확인 가능

📓 실습 1.3 - dtype 속성으로 데이터 타입 확인

```
[소스코드]
import numpy as np

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

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

- 실습 1.3 결과 풀이
  - 리스트 자료형인 list1은 integer 숫자인 1, 2, 3을 값으로 가지고 있으며, 이를 ndarray로 쉽게 변경할 수 있다. 변경된 ndarray 내의 데이터 값은 모두 int32형
  - 다른 데이터 유형이 섞여 있는 리스트를 ndarray로 변경하면 데이터 크기가 더 큰 데이터 타입으로 형 변환을 일괄 적용

📓 실습 1.4 - 데이터 타입이 섞인 경우 형 변환 형태

```
[소스코드]
import numpy as np

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)
```

```
[결과]
['1' '2' 'test'] <U11
[1. 2. 3.] float64
```

- 실습 1.4 결과 풀이
  - list2를 ndarray로 변환한 array2는 숫자형 값 1, 2가 모두 문자열 값인 '1', '2'로 변환되었다.

    - 이처럼 ndarray는 데이터 값이 모두 같은 데이터 타입이어야 하므로 서로 다른 데이터 타입이 섞여있을 경우 더 큰 데이터 타입으로 변환
    - 따라서 int형이 유니코드 문자열 값으로 변환되어 반환
- ndarray 내 데이터 타입 변경은 astype() 메서드에 인자로 원하는 타입을 문자열로 지정하여 할 수 있다
    - 이는 대용량 데이터의 ndarray를 만들 때 많은 메모리가 사용되는데, 메모리를 절약하고자 할 때 사용한다. int형으로 충분한데 데이터 타입이 float라면 int로 바꿔 메모리를 절약할 수 있다

📓 실습 1.5 - astype() 활용한 데이터 타입 변경

```
[소스코드]
import numpy as np

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)
```

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