



김범준



박소영



오세정



이현진

1

# 3주차 정규 Session

비타민 6기 3주차 정규세션 1조가 준비했습니다!

## - 목차 -

### 1. 누락 데이터 처리

- 누락 데이터 확인
- 누락 데이터 제거
- 누락 데이터 치환

### 2. 중복 데이터 처리

- 중복 데이터 확인
- 중복 데이터 제거

### 3. 데이터 표준화

- 단위 환산
- 자료형 변환

### 4. 범주형(카테고리) 데이터 처리

- 인코딩
- 레이블 인코딩
- 원핫 인코딩

### 5. 피처 스케일링

- 표준화
- 정규화

### 6. fit(), transform(), fit\_transform()

## 데이터 전처리의 중요성

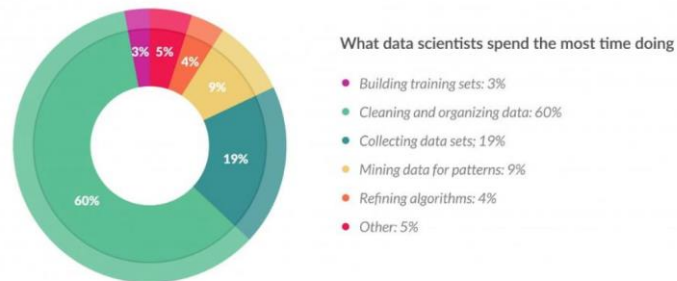
데이터 전처리란?

수집한 데이터를 분석에 적합한 형태로 만드는 과정

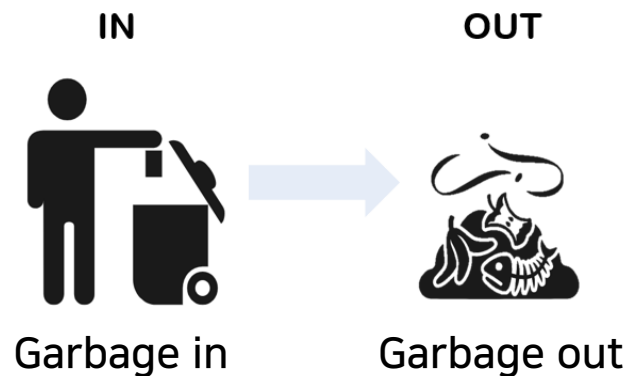
기계학습 알고리즘이 얼마나 학습 잘하는지?

⇒ 분석 데이터 품질이 결정!

-누락 데이터, 중복 데이터 등의 오류 수정 필요



데이터 사전처리: 약 80%차지



## 누락 데이터 처리

누락 데이터란?

NaN으로 표시 - Not a Number

```
# 라이브러리 불러오기
import seaborn as sns

# titanic 데이터셋 가져오기
df = sns.load_dataset('titanic')
df.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

- 누락 데이터가 많아지면 데이터 품질이 떨어짐
- 누락 데이터 제거 or 치환 작업 필요

## 누락 데이터 처리

누락 데이터 확인

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 891 entries, 0 to 890
```

```
Data columns (total 15 columns):
```

#	Column	Non-Null Count	Dtype
0	survived	891 non-null	int64
1	pclass	891 non-null	int64
2	sex	891 non-null	object
3	age	714 non-null	float64
4	sibsp	891 non-null	int64
5	parch	891 non-null	int64
6	fare	891 non-null	float64
7	embarked	889 non-null	object
8	class	891 non-null	category
9	who	891 non-null	object
10	adult_male	891 non-null	bool
11	deck	203 non-null	category
12	embark_town	889 non-null	object
13	alive	891 non-null	object
14	alone	891 non-null	bool

```
dtypes: bool(2), category(2), float64(2), int64(4), object(5)
```

```
memory usage: 80.6+ KB
```

- df.info()  
: 각 열에 속하는 non-null (NaN이 아닌) 값의 개수
- RangeIndex: 각 열에 있는 데이터의 개수
- Rangeindex 수 - non-null 수 = 누락 데이터 수

ex) Deck

- $891 - 203 = 688$

## 누락 데이터 처리

누락 데이터 확인

Value\_counts(normalize=False, sort=True, ascending=False, bins=None, dropna=True)

```
# deck 열의 NaN 개수 계산하기
```

```
nan_deck = df['deck'].value_counts(dropna=False)  
print(nan_deck)
```

```
NaN    688  
C       59  
B       47  
D       33  
E       32  
A       15  
F       13  
G        4  
Name: deck, dtype: int64
```

- value\_counts()  
: 특정 column/series의 unique value별로 count
- dropna = False  
: NaN 값 구할 수 있음!!
- dropna = True  
: NaN 값을 제외한 유효한 데이터 개수 구함

## 누락 데이터 처리

### 누락 데이터 확인

### 누락 데이터를 찾는 직접적인 방법

# isnull() 메소드로 누락 데이터 찾기

```
print(df.head().isnull())
```

	survived	pclass	sex	age	...	deck	embark_town	alive	alone
0	False	False	False	False	...	True	False	False	False
1	False	False	False	False	...	False	False	False	False
2	False	False	False	False	...	True	False	False	False
3	False	False	False	False	...	False	False	False	False
4	False	False	False	False	...	True	False	False	False

True: 누락 데이터  
False: 유효한 값

# notnull() 메소드로 누락 데이터 찾기

```
print(df.head().notnull())
```

	survived	pclass	sex	age	...	deck	embark_town	alive	alone
0	True	True	True	True	...	False	True	True	True
1	True	True	True	True	...	True	True	True	True
2	True	True	True	True	...	False	True	True	True
3	True	True	True	True	...	True	True	True	True
4	True	True	True	True	...	False	True	True	True

True: 유효한 값  
False: 누락 데이터

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

개수를 구하기 위해서는  
sum 함수 사용!!

## 누락 데이터 처리

누락 데이터 확인

### 누락 데이터의 개수 구하기

# isnull()메소드로 누락 데이터 개수 구하기

```
print(df.head().isnull().sum(axis=0))
```

```
survived      0
pclass        0
sex           0
age           0
sibsp         0
parch         0
fare          0
embarked      0
class         0
who           0
adult_male    0
deck         3
embark_town   0
alive         0
alone         0
dtype: int64
```

sum 함수 사용시

- True값 = 1

- False값 = 0

axis = 0 :  
행들을 더하기

	survived	pclass	sex	age	...	deck	embark_town	alive	alone
0	False	False	False	False	...	True	False	False	False
1	False	False	False	False	...	False	False	False	False
2	False	False	False	False	...	True	False	False	False
3	False	False	False	False	...	False	False	False	False
4	False	False	False	False	...	True	False	False	False

axis = 1  
: 열들을 더하기

axis=1로 계산 할 경우

# isnull()메소드로 누락 데이터 개수 구하기

```
print(df.head().isnull().sum(axis=1))
```

```
0    1
1    0
2    1
3    0
4    1
```



## 누락 데이터 처리

누락 데이터 확인

“각 column의 누락 데이터 수 확인”

```
# 라이브러리 불러오기
import seaborn as sns

# titanic 데이터셋 가져오기
df = sns.load_dataset('titanic')

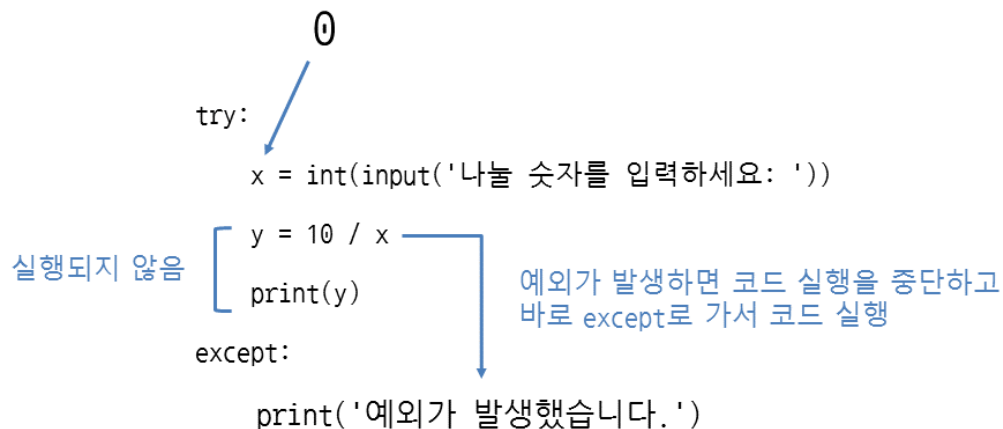
# for 반복문으로 각 열의 NaN 개수 계산하기
missing_df = df.isnull()
for col in missing_df.columns:
    missing_count = missing_df[col].value_counts() # 각 열의 NaN 개수 파악

    try:
        print(col, ': ', missing_count[True]) # NaN 값이 있으면 개수 출력
    except:
        print(col, ': ', 0) # NaN 값이 없으면 0 개 출력
```

```
survived : 0
pclass : 0
sex : 0
age : 177
sibsp : 0
parch : 0
fare : 0
embarked : 2
class : 0
who : 0
adult_male : 0
deck : 888
embark_town : 2
alive : 0
alone : 0
```

- try & except

: try 하위 명령에서 오류가 발생할 경우,  
except 실행



## 누락 데이터 처리

누락 데이터 제거

`df.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)`

### “열 삭제”

*# NaN 값이 500개 이상의 열을 모두 삭제 - deck 열 (891개 중 688개의 NaN값)*

```
df_thresh = df.dropna(axis=1, thresh=500)
print(df_thresh.columns)
```

```
Index(['survived', 'pclass', 'sex', 'age', 'sibsp', 'parch', 'fare',
      'embarked', 'class', 'who', 'adult_male', 'embark_town', 'alive',
      'alone'],
      dtype='object')
```

- axis=1 : 열(column) ↔ axis=0 : 행(row)
- thresh = 500  
: NaN 값을 500개 이상 갖는 경우 삭제  
임계값(threshold)을 설정한다고 생각하면 됨

### “행 삭제”

*# age 열에 나이 데이터가 없는 모든 행 삭제 - age 열 (891개 중 177개의 NaN값)*

```
df_age = df.dropna(axis=0, subset = ['age'], how = 'any')
print(len(df_age))
```

- subset : 열을 list 형태로 입력
- how = 'any' : NaN값이 하나라도 존재하면 삭제  
= 'all' : 모든 데이터가 NaN값일 때 삭제

## 누락 데이터 처리

누락 데이터 치환

### “NaN값을 평균값으로 치환하기”

```
# age 열의 첫 10개 데이터 출력 (5행에 NaN값)
print(df['age'].head(10))
print('\n')

# age 열의 NaN값을 다른 나이 데이터의 평균으로 변경하기
mean_age = df['age'].mean(axis=0) # age column의 평균 계산(NaN값 제외)
df['age'].fillna(mean_age, inplace=True)

# age 열의 첫 10개 데이터 출력 (5행에 NaN값이 평균으로 대체)
print(df['age'].head(10))
```

```
0    22.0
1    38.0
2    26.0
3    35.0
4    35.0
5     NaN
6    54.0
7     2.0
8    27.0
9    14.0
Name: age, dtype: float64
```

```
0    22.000000
1    38.000000
2    26.000000
3    35.000000
4    35.000000
5    29.699118
6    54.000000
7     2.000000
8    27.000000
9    14.000000
Name: age, dtype: float64
```

`df.fillna(value=None, method=None, axis=None, inplace=False)`

- 원본 객체 변경을 위해서 `inplace = True` 옵션 추가
- `mean()` : NaN을 제외한 값들의 평균 계산
- `median()` : 중간값 계산

### “dataframe 만들고 NaN값 치환하기”

```
import numpy as np
import pandas as pd

# NaN값을 갖고있는 dataframe 만들기
df_ex = pd.DataFrame([[np.nan, 1, 3],
                       [5, np.nan, 4],
                       [8, 6, np.nan]],
                      columns=('A', 'B', 'C'))

print(df_ex)

# A열의 NaN값을 해당 열의 중간값으로 채우기
df_ex['A']
median_ex = df_ex['A'].median(axis=0)
df_ex['A'].fillna(median_ex, inplace=True)

# B열의 NaN값을 0으로, C열의 NaN값은 9로 채우기
values = {"B": 0, "C": 9}
df_ex.fillna(value = values, inplace=True)

df_ex
```

	A	B	C
0	NaN	1.0	3.0
1	5.0	NaN	4.0
2	8.0	6.0	NaN

	A	B	C
0	6.5	1.0	3.0
1	5.0	0.0	4.0
2	8.0	6.0	9.0

## 누락 데이터 처리

누락 데이터 치환

```
# embark_town 열의 829행의 NaN 데이터 출력
print(df['embark_town'][825:830])
print('\n')

# embark_town 열의 NaN값을 승선도시 중에서 가장 많이 출현한 값으로 치환하기
most_freq = df['embark_town'].value_counts(dropna=True).idxmax()
print(most_freq)
print('\n')

df['embark_town'].fillna(most_freq, inplace=True)

# embark_town 열 829행의 NaN 데이터 출력(NaN값이 most_freq 값으로 대체)
print(df['embark_town'][825:830])
```

```
825    Queenstown
826    Southampton
827     Cherbourg
828    Queenstown
829         NaN
Name: embark_town, dtype: object
```

Southampton

```
825    Queenstown
826    Southampton
827     Cherbourg
828    Queenstown
829    Southampton
Name: embark_town, dtype: object
```

“가장 많이 나타나는 값으로 바꾸기”

- idxmax() : 가장 큰 값 찾기

\* 누락 데이터가 NaN으로 표시되지 않은 경우  
누락 데이터가 : ?, -, unkown 등으로 표시  
→replace()를 활용하여 NaN으로 변경

ex. '?'을 np.nan으로 치환

- Numpy에서 지원하는 np.nan으로 변경

df.replace("?", np.nan, inplace=True)

## 누락 데이터 처리

누락 데이터 치환

```
# 라이브러리 불러오기
import seaborn as sns

# titanic 데이터셋 가져오기
df = sns.load_dataset('titanic')

# embark_town 열의 829행의 NaN 데이터 출력
print(df['embark_town'][825:830])
print('\n')

# embark_town 열의 NaN값을 바로 앞에 있는 828행의 값으로 변경하기
df['embark_town'].fillna(method='ffill', inplace=True)
print(df['embark_town'][825:830])
```

```
825    Queenstown
826    Southampton
827     Cherbourg
828    Queenstown
829           NaN
Name: embark_town, dtype: object
```

```
825    Queenstown
826    Southampton
827     Cherbourg
828    Queenstown
829    Queenstown
Name: embark_town, dtype: object
```

“이웃하고 있는 값으로 바꾸기”

- method = 'ffill' : NaN 행의 직전 행 값으로 바꿈
- method = 'bfill' : NaN 행의 다음 행 값으로 바꿈

## 중복 데이터 처리 중복 데이터 확인

```
# 라이브러리 불러오기
import pandas as pd

# 중복 데이터를 갖는 데이터프레임 만들기
df = pd.DataFrame({'c1': ['a', 'a', 'b', 'a', 'b'],
                   'c2': [1, 1, 1, 2, 2],
                   'c3': [1, 1, 2, 2, 2]})

print(df)
print('\n')

# 데이터프레임 전체 행 데이터 중에서 중복값 찾기
df_dup = df.duplicated()
print(df_dup)
print('\n')
```

	c1	c2	c3
0	a	1	1
1	a	1	1
2	b	1	2
3	a	2	2
4	b	2	2

```
0    False
1     True
2    False
3    False
4    False
dtype: bool
```

### “특정 열에서 중복 데이터 확인”

```
# 데이터프레임의 특정 열 데이터에서 중복값 찾기
col_dup = df['c2'].duplicated()
print(col_dup)
```

```
0    False
1     True
2     True
3    False
4     True
Name: c2, dtype: bool
```

	c1	c2	c3
0	a	1	1
1	a	1	1
2	b	1	2
3	a	2	2
4	b	2	2

df.duplicated()

-전에 나온 행들과 비교!

중복되는 행: True

처음 나오는 행 : False

## 중복 데이터 처리 중복 데이터 제거

`dataframe.drop_duplicates(subset=None, keep='first', inplace=False)`

```
print(df)
print('\n')

# 데이터프레임에서 중복 행 제거
df2 = df.drop_duplicates()
print(df2)
print('\n')
```

	c1	c2	c3
0	a	1	1
1	a	1	1
2	b	1	2
3	a	2	2
4	b	2	2

	c1	c2	c3
0	a	1	1
2	b	1	2
3	a	2	2
4	b	2	2

Subset 옵션에 해당하는 열 기준으로 판단

```
# c2, c3열을 기준으로 중복 행 제거
df3 = df.drop_duplicates(subset=['c2', 'c3'])
print(df3)
```

	c1	c2	c3
0	a	1	1
2	b	1	2
3	a	2	2

## 데이터 표준화 단위 환산

### 단위 환산이 필요한 이유

같은 데이터셋 안에서  
서로 다른 측정 단위를 사용하는 경우

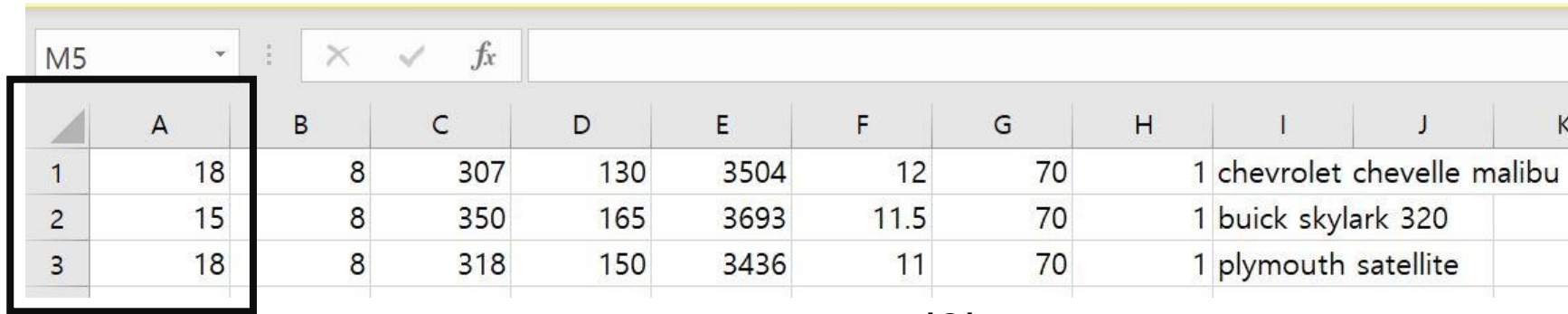
전체 데이터의 일관성에서 문제가  
발생하기 때문에 측정 단위를 맞추어 줄 필요가 있음

외국데이터를 불러왔을 때  
국내에서 잘 사용하지 않는  
마일, 야드, 온스 등의 도량형 단위가  
포함되어 있는 경우

한국에서 사용하는 단위인  
미터, 평, 그램으로 변환하는 것이 좋음



## 데이터 표준화 단위 환산



	A	B	C	D	E	F	G	H	I	J	K
1	18	8	307	130	3504	12	70		1	chevrolet chevelle malibu	
2	15	8	350	165	3693	11.5	70		1	buick skylark 320	
3	18	8	318	150	3436	11	70		1	plymouth satellite	

auto-mpg.csv 파일

A열은 mpg (mile per gallon)로 되어 있음.  
이를 한국에서 익숙한 표기법인 kpl (kilometer per liter)로 변환

## 데이터 표준화 단위 환산

### 데이터 불러오기

```
▶ import pandas as pd

df = pd.read_csv('auto-mpg.csv', header = None)

df.columns = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
              'acceleration', 'model year', 'origin', 'name']
```

	A	B	C	D	E
1	18	8	307	130	3504
2	15	8	350	165	3693
3	18	8	318	150	3436

auto-mpg.csv 파일

원본데이터에 열 이름이 없기 때문에  
Header = None을 입력한 후

df.columns에 열 이름을 직접 입력해줌.

## 데이터 표준화 단위 환산

```
▶ # mpg(mile per gallon)를 kpl(kilometer per liter)로 변환 (mpg_to_kpl = 0.425)
mpg_to_kpl = 1.60934/3.78541

# mpg 열에 0.425를 곱한 결과를 새로운 열에 추가
df['kpl'] = df['mpg'] * mpg_to_kpl
print(df.head(3))
print('/n')
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	₩
0	18.0	8	307.0	130.0	3504.0	12.0	70	
1	15.0	8	350.0	165.0	3693.0	11.5	70	
2	18.0	8	318.0	150.0	3436.0	11.0	70	

	origin	name	kpl
0	1	chevrolet chevelle malibu	7.652571
1	1	buick skylark 320	6.377143
2	1	plymouth satellite	7.652571

/n

1 mile은 1.60934km이고 1 gallon은 3.78541임.  
Mpg를 kpl로 변환하기 위하여 mpg\_to\_kpl을 계산하고  
mpg열에 mpg\_to\_kpl을 곱한 결과를 새로운 열(kpl)로 추가함

## 데이터 표준화 단위 환산

▶ # kpl 열을 소수점 아래 둘째자리에서 반올림

```
df['kpl'] = df['kpl'].round(2)
print(df.head(3))
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	#
0	18.0	8	307.0	130.0	3504.0	12.0	70	
1	15.0	8	350.0	165.0	3693.0	11.5	70	
2	18.0	8	318.0	150.0	3436.0	11.0	70	

	origin	name	kpl
0	1	chevrolet chevelle malibu	7.65
1	1	buick skylark 320	6.38
2	1	plymouth satellite	7.65

round(n)으로 소수점 아래 n자리에서 반올림 가능

Df['kpl'].round(2)로 kpl열을 소수점 아래 둘째자리에서 반올림

## 데이터 표준화 자료형 변환

### 각 열의 자료형 확인하기

▶ # 각 열의 자료형 확인

```
print(df.dtypes)
print('/n')
```

```
mpg          float64
cylinders     int64
displacement float64
horsepower   object
weight       float64
acceleration  float64
model year    int64
origin        int64
name          object
dtype: object
/n
```

숫자가 객체형(object)로 저장된 경우에는  
숫자형 (int or float)으로 변환해야 함.

먼저 df.dtypes 메소드를 통해서  
각 열의 자료형을 확인

엔진 출력의 크기를 나타내는 데이터인 horsepower가  
객체형인 object로 저장된 것을 확인할 수 있음.

## 데이터 표준화 자료형 변환

### 각 열의 자료형 확인하기

▶ # 각 열의 자료형 확인

```
print(df.dtypes)
print('/n')
```

```
mpg          float64
cylinders     int64
displacement float64
horsepower   object
weight       float64
acceleration float64
model year    int64
origin        int64
name          object
dtype: object
/n
```

Everything is object in Python!

파이썬에서는 객체(object)라는 단위로 메모리 위의 정보를 보관함.

다시 말해 객체란 메모리에 저장된 자료라는 뜻.

숫자가 객체형(object)로 저장된 경우에는  
숫자형(int or float)으로 변환해야 함.

먼저 df.dtypes 메소드를 통해서  
각 열의 자료형을 확인

엔진 출력의 크기를 나타내는 데이터인 horsepower가  
객체형인 object로 저장된 것을 확인할 수 있음.

## 데이터 표준화 자료형 변환

### 각 열의 자료형 확인하기

▶ # horsepower 열의 고유값 확인

```
print(df['horsepower'].unique())  
print('/n')
```

```
['130.0' '165.0' '150.0' '140.0' '198.0' '220.0' '215.0' '225.0' '190.0'  
'170.0' '160.0' '95.00' '97.00' '85.00' '88.00' '46.00' '87.00' '90.00'  
'113.0' '200.0' '210.0' '193.0' '?' '100.0' '105.0' '175.0' '153.0'  
'180.0' '110.0' '72.00' '86.00' '70.00' '76.00' '65.00' '69.00' '60.00'  
'80.00' '54.00' '208.0' '155.0' '112.0' '92.00' '145.0' '137.0' '158.0'  
'167.0' '94.00' '107.0' '230.0' '49.00' '75.00' '91.00' '122.0' '67.00'  
'83.00' '78.00' '52.00' '61.00' '93.00' '148.0' '129.0' '96.00' '71.00'  
'98.00' '115.0' '53.00' '81.00' '79.00' '120.0' '152.0' '102.0' '108.0'  
'68.00' '58.00' '149.0' '89.00' '63.00' '48.00' '66.00' '139.0' '103.0'  
'125.0' '133.0' '138.0' '135.0' '142.0' '77.00' '62.00' '132.0' '84.00'  
'64.00' '74.00' '116.0' '82.00']  
/n
```

고유값을 확인해봤을 때  
중간에 '?' 문자열이 섞여있어서  
CSV 파일을 데이터프레임으로  
변환하는 과정에서 문자열로 인식

## 데이터 표준화 자료형 변환

### 누락데이터 삭제 후 자료형 변경

▶ # 누락 데이터('?') 삭제

```
import numpy as np
```

```
df['horsepower'].replace('?', np.nan, inplace=True) # '?'을 np.nan으로 변경  
df.dropna(subset=['horsepower'], axis=0, inplace=True) # 누락 데이터 행 삭제  
df['horsepower'] = df['horsepower'].astype('float') # 문자열을 실수형으로 변경
```

```
# horsepower 열의 자료형 확인  
print(df['horsepower'].dtype)
```

```
float64
```

고유값의 '?'를 NaN값으로 변경한 후 누락데이터가 들어있는 행을

df.dropna 메소드로 삭제하고 객체형(object)를 실수형(float)으로 변경

Df['horsepower'].dtype으로 horsepower의 자료형이 실수형으로 변경되었음을 확인할 수 있음



## 데이터 표준화 자료형 변환

```
▶ # origin 열의 고유값 확인  
print(df['origin'].unique())
```

[1 3 2]

Origin 열에는 정수형 데이터인 1, 2, 3이 들어있지만 실제로는 국가 이름인 USA, EU, JPN을 뜻함.

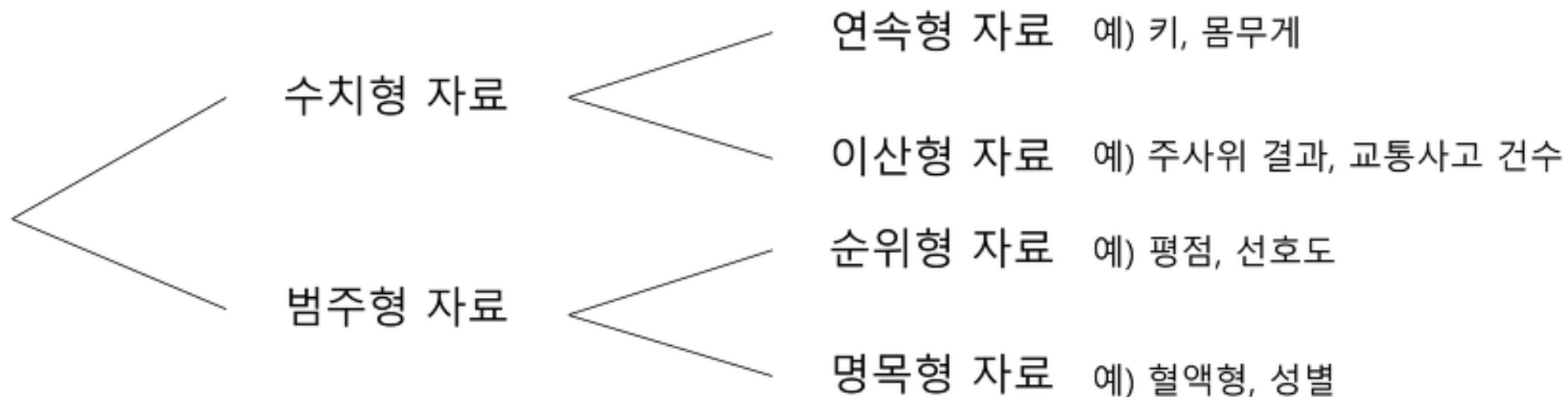
```
▶ # 정수형 데이터를 문자열 데이터로 변환  
df['origin'].replace({1:'USA', 2:'EU', 3:'JPN'}, inplace=True)  
  
# origin 열의 고유값과 자료형 확인  
print(df['origin'].unique())  
print(df['origin'].dtype)  
  
['USA' 'JPN' 'EU']  
object
```

Replace() 메소드를 사용하여 각 숫자 데이터를 국가 이름으로 바꿔줄 수 있음.  
Inplace = True를 통해 원본데이터를 변경할 수 있음.  
숫자 데이터를 문자열 데이터로 변환하면서 자료형이 object로 바뀐 것 확인 가능.

## 데이터 표준화 자료형 변환

객체형 데이터(object)를 범주형 데이터(category)로 변경

→ 유한 개의 고유값이 반복적으로 나타내는 경우



## 데이터 표준화 자료형 변환

객체형 데이터(object)를 범주형 데이터(category)로 변경

→ 유한 개의 고유값이 반복적으로 나타내는 경우

```
▶ # 문자열을 범주형으로 변환  
df['origin'] = df['origin'].astype('category')  
print(df['origin'].dtypes)
```

category

astype('category') 메소드를 이용하면  
범주형 데이터로 변환할 수 있음

```
▶ # 범주형을 문자형으로 다시 변환  
df['origin'] = df['origin'].astype('str')  
print(df['origin'].dtype)
```

object

astype('str') 메소드를 이용하면  
문자형 데이터로 변환할 수 있음

## 데이터 표준화 자료형 변환

숫자형 데이터(int64)를 범주형 데이터(category)로 변경

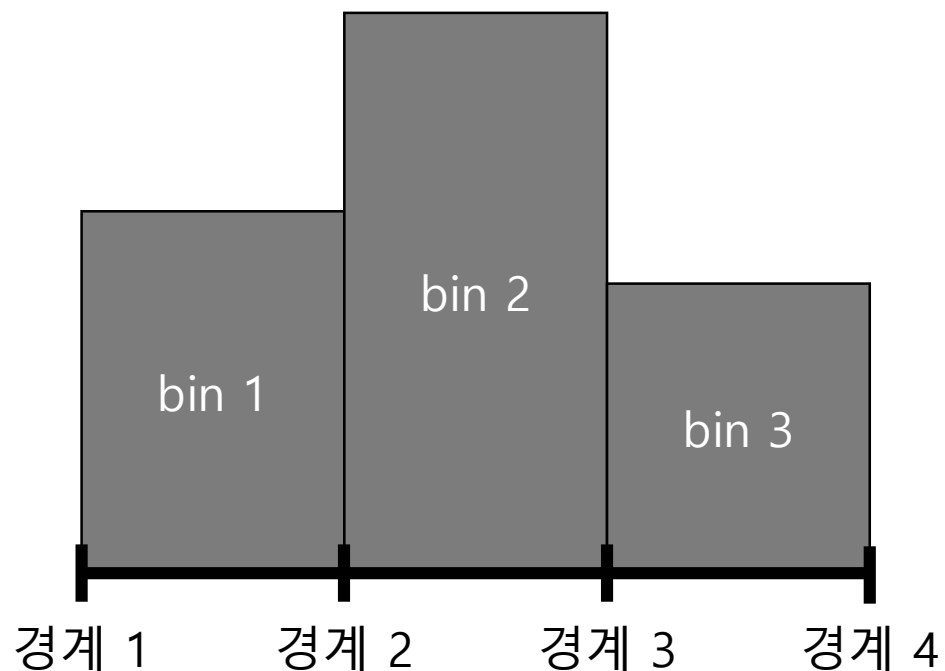
```
# model year 열의 정수형을 범주형으로 변환  
print(df['model year'].sample(3))  
df['model year'] = df['model year'].astype('category')  
print(df['model year'].sample(3))
```

```
346    81  
8      70  
319    80  
Name: model year, dtype: int64  
362    81  
210    76  
131    74  
Name: model year, dtype: category  
Categories (13, int64): [70, 71, 72, 73, ..., 79, 80, 81, 82]
```

출시년도를 나타내는 'model year'의 경우  
시간적인 순서의 의미는 있으나  
숫자의 상대적 크기는 별 의미가 없음

astype('category') 메소드를 이용하면 범주형 데이터로 변환할 수 있음

## 범주형(카테고리) 데이터 처리



데이터 분석 알고리즘에 따라서는  
연속 데이터를 그대로 사용하기 보다는  
일정한 구간(bin)으로 나눠서  
분석하는 것이 효율적인 경우가 있음.

연속 변수를 일정한 구간으로 나누고,  
각 구간을 범주형 이산 변수로 변환하는 과정을  
구간 분할(binining)이라고 함.

## 범주형(카테고리) 데이터 처리

엔진 출력을 나타내는 연속 변수  
'horsepower'



저출력 / 보통출력 / 고출력으로  
구간 분할

```
▶ # np.histogram 함수로 3개의 bin으로 구분할 경계값의 리스트 구하기  
count, bin_dividers = np.histogram(df['horsepower'], bins=3)  
print(bin_dividers)
```

```
[ 46.          107.33333333 168.66666667 230.          ]
```

저출력, 보통출력, 고출력을 구분할 bin값의 리스트가 출력됨

## 범주형(카테고리) 데이터 처리

```

▶ # 3개의 bin에 이름 지정
bin_names = ['저출력', '보통출력', '고출력']

# pd.cut 함수로 각 데이터를 3개의 bin에 할당
df['hp_bin'] = pd.cut(x = df['horsepower'],      # 데이터 배열
                      bins = bin_dividers,       # 경계값 리스트
                      labels = bin_names,        # bin 이름
                      include_lowest = True)      # 첫 경계값 포함

# horsepower 열, hp_bin 열의 첫 15행 출력
print(df[['horsepower', 'hp_bin']].head(15))

```

‘horsepower’열의 숫자 데이터를 3개 구간에 할당하고  
 각 구간의 이름(저출력/보통출력/고출력)을 hp\_bin에 저장  
 Include\_lowest=True로 각 구간의 낮은 경계값을 포함해주면  
 각 데이터가 해당하는 구간이 출력됨

	horsepower	hp_bin
0	130.0	보통출력
1	165.0	보통출력
2	150.0	보통출력
3	150.0	보통출력
4	140.0	보통출력
5	198.0	고출력
6	220.0	고출력
7	215.0	고출력
8	225.0	고출력
9	190.0	고출력
10	170.0	고출력
11	160.0	보통출력
12	150.0	보통출력
13	225.0	고출력
14	95.0	저출력

## 범주형(카테고리) 데이터 처리 인코딩

### 인코딩이란?

코드화, 암호화

컴퓨터에서 인코딩 - 사람이 인지할 수 있는 형태의 데이터를 약속된 규칙에 의해

컴퓨터가 사용하는 0과 1로 변환하는 과정

⇒ ML알고리즘에서 사용가능하도록 데이터를 변환하는 것

# 디코딩 : 부호화된 정보를 부호화되기 전으로 되돌리는 처리 혹은 그 처리 방식



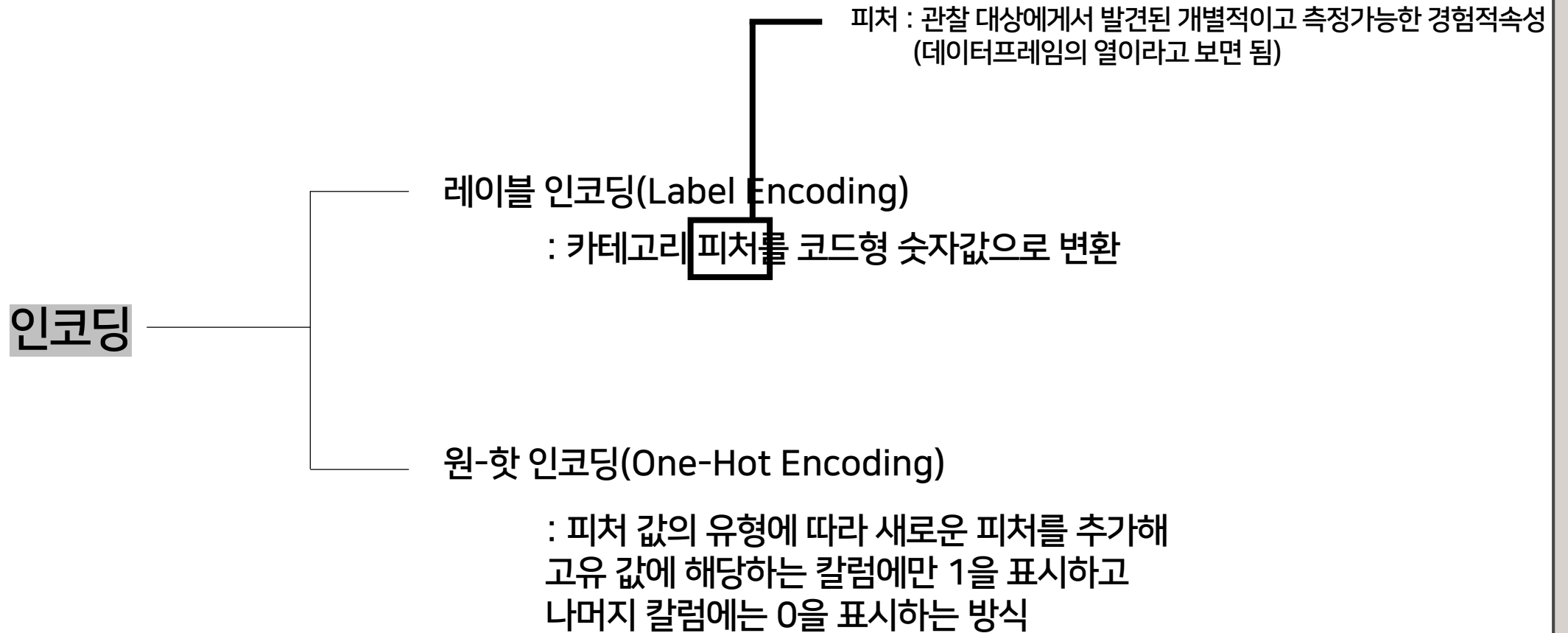
## 범주형(카테고리) 데이터 처리 인코딩



무료 소프트웨어 기계학습 라이브러리

사이킷런 머신러닝 알고리즘과 같이  
문자열 값을 입력 값으로 허용하지 않는 경우

## 범주형(카테고리) 데이터 처리 인코딩



## 범주형(카테고리) 데이터 처리 레이블 인코딩

[상품구분]

TV  
냉장고  
전자레인지  
컴퓨터  
선풍기  
믹서



1 : TV  
2 : 냉장고  
3 : 전자레인지  
4 : 컴퓨터  
5 : 선풍기  
6 : 믹서

“숫자형 값으로 변환”

[주 의]

'01', '02'와 같은 코드값도 문자열

▶ 숫자형으로 변환

## 범주형(카테고리) 데이터 처리

레이블 인코딩

LabelEncoder 클래스로 구현

```
from sklearn.preprocessing import LabelEncoder

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

#LabelEncoder를 객체로 생성한 후, fit()과 transform()으로 레이블 인코딩 수행.
encoder=LabelEncoder()
encoder.fit(items)
labels=encoder.transform(items)
print('인코딩 변환값:',labels)
```

인코딩 변환값: [0 1 4 5 3 3 2 2]

items리스트의 원소들이 숫자 값으로 변환되어 출력

## fit(), transform(), fit\_transform()

fit()

데이터 변환을 위한 기준 정보 설정  
매개변수를 이용해서 내부개체상태로 저장  
예를 들면, 데이터 세트의 최대값/최소값 설정 등

+

transform()

fit()정보를 이용해서 데이터를 변환

=

fit\_transform()

## 범주형(카테고리) 데이터 처리

레이블 인코딩

LabelEncoder 클래스로 구현

```
print('인코딩 클래스:', encoder.classes_)
```

인코딩 클래스: ['TV' '냉장고' '믹서' '선풍기' '전자레인지' '컴퓨터']

```
print('디코딩 원본값:', encoder.inverse_transform([4,5,2,0,1,1,3,3]))
```

디코딩 원본값: ['전자레인지' '컴퓨터' '믹서' 'TV' '냉장고' '냉장고' '선풍기' '선풍기']

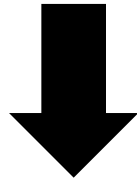
데이터가 매우 많을 경우 어떤 문자열이 어떤 숫자로 변환되었는지 알기 어려움

- ▶ class\_속성값은 0번부터 순서대로 어떤 문자열이 변환된 것인지 보여줌
- ▶ inverse\_transform() 은 인코딩된 값을 디코딩해줌

## 범주형(카테고리) 데이터 처리

레이블 인코딩

회귀와 같은 머신러닝 알고리즘에서  
숫자값의 크고 작음에 대한 특성이 작용 → 예측 성능 저하



1과 0으로만 표현  
원-핫 인코딩 (One-Hot Encoding)

## 범주형(카테고리) 데이터 처리 원핫 인코딩

상품분류	상품분류_ TV	상품분류_ 냉장고	상품분류_ 믹서	상품분류_ 선풍기	상품분류_ 전자레인지	상품분류_ 컴퓨터
TV	1	0	0	0	0	0
냉장고	0	1	0	0	0	0
전자레인지	0	0	0	0	1	0
컴퓨터	0	0	0	0	0	1
선풍기	0	0	0	1	0	0
선풍기	0	0	0	1	0	0
믹서	0	0	1	0	0	0
믹서	0	0	1	0	0	0



## 범주형(카테고리) 데이터 처리 원핫 인코딩

### OneHotEncoder 클래스로 구현

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

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

#먼저 숫자 값으로 변환을 위해 LabelEncoder로 변환합니다.
encoder=LabelEncoder()
encoder.fit(items)
labels=encoder.transform(items)
#2차원 데이터로 변환합니다.
labels=labels.reshape(-1,1)

#원-핫 인코딩을 적용합니다.
oh_encoder=OneHotEncoder()
oh_encoder.fit(labels)
oh_labels=oh_encoder.transform(labels)
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)

첫 번째 값이 인코딩 0(첫번째)값인 TV이므로 첫 번째 레코드의 첫 번째 칼럼이 1

## 범주형(카테고리) 데이터 처리 원핫 인코딩

### 판다스의 get dummies() 구현

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

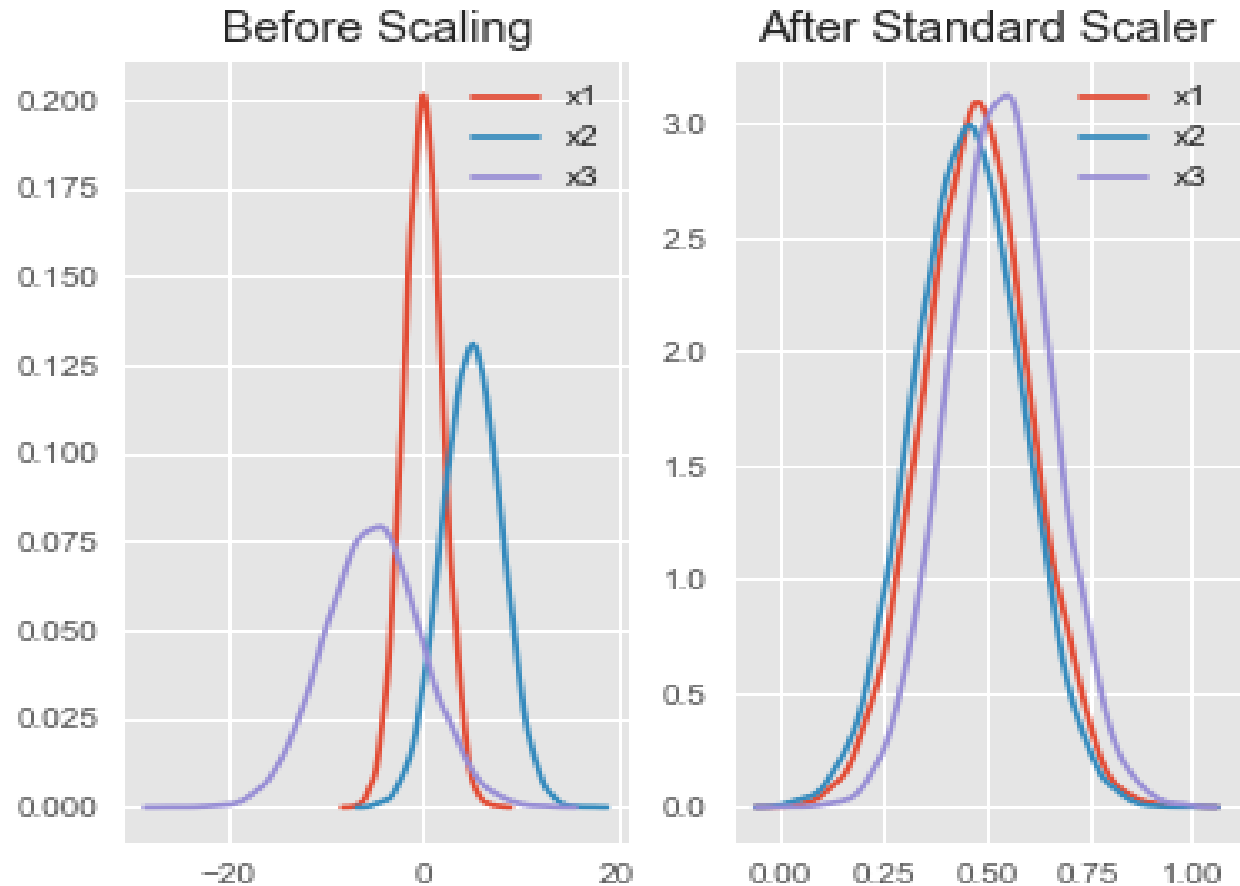
	item_TV	item_냉장고	item_믹서	item_선풍기	item_전자레인지	item_컴퓨터
0	1	0	0	0	0	0
1	0	1	0	0	0	0
2	0	0	0	0	1	0
3	0	0	0	0	0	1
4	0	0	0	1	0	0
5	0	0	0	1	0	0
6	0	0	1	0	0	0
7	0	0	1	0	0	0

숫자형 값 변환 없이  
바로 변환 가능

## 피쳐 스케일링

피쳐 스케일링이란?

서로 다른 변수의 값 범위나  
데이터의 피처를 동일한 수준으로  
맞추는 작업.  
표준화와 정규화가  
피쳐 스케일링의 대표적인 방법이다.



## 피쳐 스케일링 표준화

### 표준화

- 데이터의 피쳐 각각이 평균이 0, 분산이 1인 가우시안 정규 분포를 가진 값으로 변환하는 것.
- 공식:  $x_{i_{new}} = \frac{x_i - \text{mean}(x)}{\text{std}(x)}$
- 원 데이터에 해당 공식을 대입하면 표준화된 데이터가 생성됨!
- 서포트 벡터 머신, 선형 회귀, 로지스틱 회귀 등은 데이터가 정규분포를 가진다고 가정하기 때문에 전처리 과정에서 표준화는 매우 중요!

## 피쳐 스케일링 표준화

StandardScaler 표준화를 위한 클래스 사용법

1. 원 데이터의 평균, 분산값 확인
2. StandardScaler 클래스를 사용하여 표준화된 데이터 생성

## 피쳐 스케일링 표준화

```
from sklearn.datasets import load_iris
import pandas as pd
iris=load_iris()
iris_data=iris.data
iris_df=pd.DataFrame(data=iris_data,columns=iris.feature_names) #iris data set를 로딩하고 iris_df에 DataFrame형태로 저장.
print("<데이터의 평균 값>")
print(iris_df.mean())
print("\n<데이터의 분산 값>")
print(iris_df.var())
iris_df.head()
```

```
<데이터의 평균 값>
sepal length (cm)    5.843333
sepal width (cm)     3.057333
petal length (cm)    3.758000
petal width (cm)     1.199333
dtype: float64
```

```
<데이터의 분산 값>
sepal length (cm)    0.685694
sepal width (cm)     0.189979
petal length (cm)    3.116278
petal width (cm)     0.581006
dtype: float64
```

### 원 데이터의 평균 분산 확인 결과

sepal length, sepal width, petal length, petal width의 단위는 cm로 동일하나 각 데이터 피쳐들의 평균과 분산이 모두 다르고 가우시안 정규분포를 따르지도 않음 → **표준화 필요성 확인**

## 피처 스케일링 표준화

```
from sklearn.preprocessing import StandardScaler #StandardScaler를 import
scaler=StandardScaler()
scaler.fit(iris_df) #iris_df를 StandardScaler의 기준으로 fit
iris_scaled=scaler.transform(iris_df) #iris_scaled에 표준화된 iris_df의 데이터를 할당

#scaler를 통해 변환된 데이터는 ndarray형식이므로 DataFrame형식으로 변환
iris_df_scaled=pd.DataFrame(data=iris_scaled,columns=iris.feature_names)

print("<데이터의 평균 값>")
print(iris_df_scaled.mean())
print("\n<데이터의 분산 값>")
print(iris_df_scaled.var())
```

### 표준화된 데이터 생성 결과

<데이터의 평균 값>

```
sepal length (cm)    -1.690315e-15
sepal width (cm)     -1.842970e-15
petal length (cm)    -1.698641e-15
petal width (cm)     -1.409243e-15
dtype: float64
```

<데이터의 분산 값>

```
sepal length (cm)    1.006711
sepal width (cm)     1.006711
petal length (cm)    1.006711
petal width (cm)     1.006711
dtype: float64
```

데이터 피처가 평균이 0에 매우 가까운 값으로, 분산은 1에 매우 가까운 값으로 변환되었음. (e-15는  $10^{-15}$ 를 의미합니다!)

※ 주의 ※

Scaler를 통해 변환된 데이터는 ndarray형식이므로 반드시 DataFrame으로 변환해주어야 함!

## 피쳐 스케일링 정규화

### 정규화

- 서로 다른 피쳐의 크기를 통일하기 위해 크기를 변환해주는 작업

ex) 피쳐 A(금액): 1~100,000,000원의 범위를 가짐

피쳐 B(길이): 1~100cm의 범위를 가짐

피쳐 C(거리): 1~1000KM의 범위를 가짐

→ 각 피쳐를 동일한 크기 단위로 비교하기 위해 모두 최소0 최대1의 값으로 변환해주는 것이 정규화 작업!

- 공식:  $x_{i_{new}} = \frac{x_i - \min(x)}{\max(x) - \min(x)}$  or  $x_{i_{new}} = \frac{x_i}{\sqrt{X^2 + Y^2 + Z^2 \dots}}$  (벡터 정규화)

- 벡터 정규화는 개별 피쳐 벡터를 모든 피쳐 벡터의 크기로 나눠주는 작업.(선형대수에서의 정규화 개념)

일반 적으론 전자의 공식을 정규화에 활용. 이외에 각 열 데이터의 최대값의 절대값으로 열 데이터를 나눠주는 방법도 존재(예시 설명)



## 피쳐 스케일링 정규화

```
from sklearn.preprocessing import MinMaxScaler #MinMaxScaler를 import
scaler=MinMaxScaler() #iris_df를 MinMaxScaler의 기준으로 fit
scaler.fit(iris_df) #iris_scaled에 정규화된 iris_df의 데이터를 할당
iris_scaled=scaler.transform(iris_df)
#scaler를 통해 변환된 데이터는 ndarray형식이므로 DataFrame형식으로 변환
iris_df_scaled=pd.DataFrame(data=iris_scaled,columns=iris.feature_names)
print("<정규화된 데이터의 최댓값>")
print(iris_df_scaled.max())
print("\n<정규화된 데이터의 최솟값>")
print(iris_df_scaled.min())
```

<정규화된 데이터의 최댓값>

```
sepal length (cm)    1.0
sepal width (cm)     1.0
petal length (cm)    1.0
petal width (cm)     1.0
dtype: float64
```

<정규화된 데이터의 최솟값>

```
sepal length (cm)    0.0
sepal width (cm)     0.0
petal length (cm)    0.0
petal width (cm)     0.0
dtype: float64
```

### MinMaxScaler

정규화를 위한 클래스  $x_{i_{new}} = \frac{x_i - \min(x)}{\max(x) - \min(x)}$  공식을 활용하여 정규화 진행

적용결과: 피쳐들의 최댓값, 최솟값이 각각 1,0 이므로 정규화가 잘 되었음을 알 수 있다.

## 피쳐 스케일링 정규화

#각 열의 데이터를 각 열 데이터의 최댓값의 절대값으로 나눠주는 방법

```
iris_df_abs_scaled=iris_df/abs(iris_df.max())  
print("<abs 정규화된 데이터의 최댓값>")  
print(iris_df_abs_scaled.max())  
print("\n<abs 정규화된 데이터의 최솟값>")  
print(iris_df_abs_scaled.min())
```

<abs 정규화된 데이터의 최댓값>

```
sepal length (cm)    1.0  
sepal width (cm)     1.0  
petal length (cm)    1.0  
petal width (cm)     1.0  
dtype: float64
```

<abs 정규화된 데이터의 최솟값>

```
sepal length (cm)    0.544304  
sepal width (cm)     0.454545  
petal length (cm)    0.144928  
petal width (cm)     0.040000  
dtype: float64
```

각 열 데이터의 최댓값의 절대값으로 나눠주는 정규화 방법

적용결과: 정규화된 데이터 피쳐들의 값 범위가 모두  
0~1사이임을 확인 할 수 있으므로 정규화가 잘 되었음을 알 수 있다.

## 피쳐 스케일링 정규화

### 벡터 정규화

- 어떤 벡터를 단위 벡터  $u$  (크기가 1인 벡터)로 만드는 작업.

▶ 벡터를 벡터 자기 자신의 크기(Norm)로 나눠주면 된다!

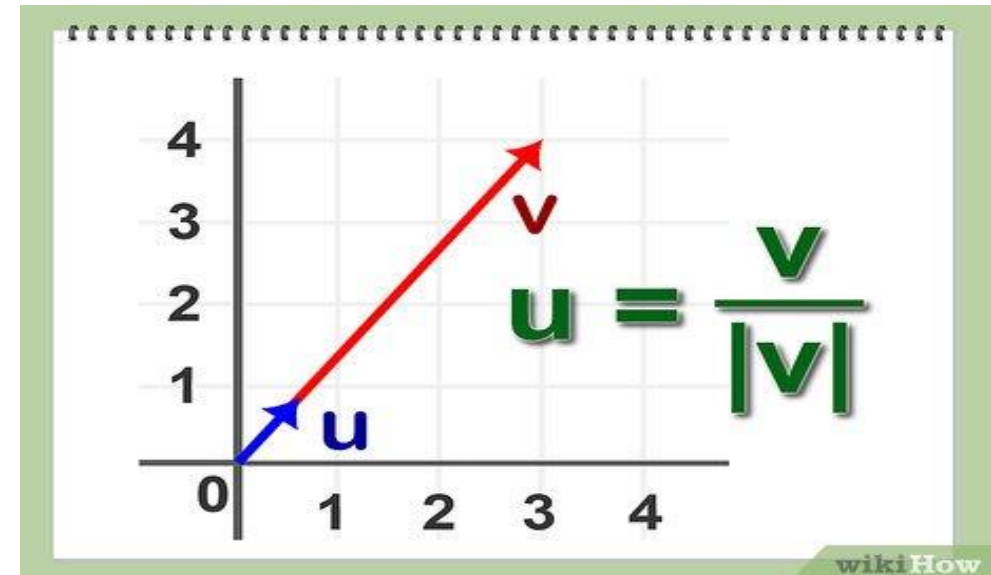
\*벡터의 Norm 공식(참고)

$$\|\mathbf{x}\|_2 = \left( \sum_{i=1}^N |x_i|^2 \right)^{1/2} = \sqrt{x_1^2 + x_2^2 + \dots + x_N^2}$$

$x_1, x_2 \dots x_N$ 은  $X$  벡터의 성분!

this vector has a length of 10  $\Rightarrow$  this vector has a length of 1

the process of normalization



## 피쳐 스케일링 정규화

### 벡터 정규화 예시!

```
import numpy as np
an_array = np.random.rand(10)*10 #(0,1)사이의 난수를 추출하고 각 난수에 10을 곱한 배열 생성
print("<원 배열> \n",an_array)
norm = np.linalg.norm(an_array) #np.linalg.norm()은 벡터의 Norm을 구하는 numpy의 함수
print("<원 배열의 Norm> \n", norm)
normal_array = an_array/norm #원 배열의 norm으로 원 배열을 나눠서 벡터 정규화 진행!(벡터 정규화 공식)
print("<벡터 정규화된 배열> \n",normal_array)
print("<벡터 정규화된 배열의 Norm> \n", np.linalg.norm(normal_array))
```

<원 배열>

```
[7.81187419 5.3971178  3.16955673 8.72617439 7.42493813 1.18129267
 0.95576481 0.48588728 0.70588013 2.76477932]
```

<원 배열의 Norm>

```
15.561602462968326
```

<벡터 정규화된 배열>

```
[0.50199677 0.34682275 0.20367804 0.56075037 0.47713198 0.07591073
 0.06141815 0.03122347 0.04536038 0.17766675]
```

<벡터 정규화된 배열의 Norm>

```
1.0
```

## 피쳐 스케일링 정규화

```
iris_df_vector_scaled=iris_df/np.linalg.norm(iris_df) #iris_df의 각 열 데이터를 데이터 프레임의 norm으로 나눠주기
print("<벡터 정규화된 데이터의 최댓값>")
print(iris_df_vector_scaled.max())
print("\n<벡터 정규화된 데이터의 최솟값>")
print(iris_df_vector_scaled.min())
print("\n<원 데이터 프레임의 Norm>", np.linalg.norm(iris_df))
print("\n<벡터 정규화된 데이터 프레임의 Norm>", np.linalg.norm(iris_df_vector_scaled))
```

```
<벡터 정규화된 데이터의 최댓값>
sepal length (cm)    0.080885
sepal width (cm)     0.045050
petal length (cm)    0.070647
petal width (cm)     0.025597
dtype: float64
```

```
<벡터 정규화된 데이터의 최솟값>
sepal length (cm)    0.044026
sepal width (cm)     0.020477
petal length (cm)    0.010239
petal width (cm)     0.001024
dtype: float64
```

```
<원 데이터 프레임의 Norm> 97.66928892952994
```

```
<벡터 정규화된 데이터 프레임의 Norm> 1.0
```

iris 데이터 프레임에 벡터 정규화 적용하기!

각 열 데이터를 데이터 프레임 전체  
열 벡터의 Norm으로 나눠 주기

- 공식: 
$$x_{i_{new}} = \frac{x_i}{\sqrt{X^2+Y^2+Z^2...}}$$

벡터 정규화 결과 각 열의 데이터가 0~1사이의 값을  
가지게 되었고 데이터 프레임의 Norm이 1로 조정되었다.

**fit(), transform(), fit\_transform()**

딥러닝 : 분류에 사용할 데이터를 스스로 학습

머신러닝 : 학습 데이터를 수동으로 제공

머신러닝



지도학습 : 훈련 데이터로부터 하나의 함수를 유추해서 테스트 데이터를 대입하여 분석

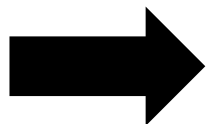
비지도학습 : 데이터 구성에 대해 알아보는 것

`fit()`, `transform()`, `fit_transform()`

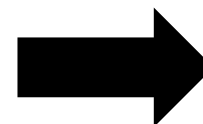
training data

test data

↓  
학습

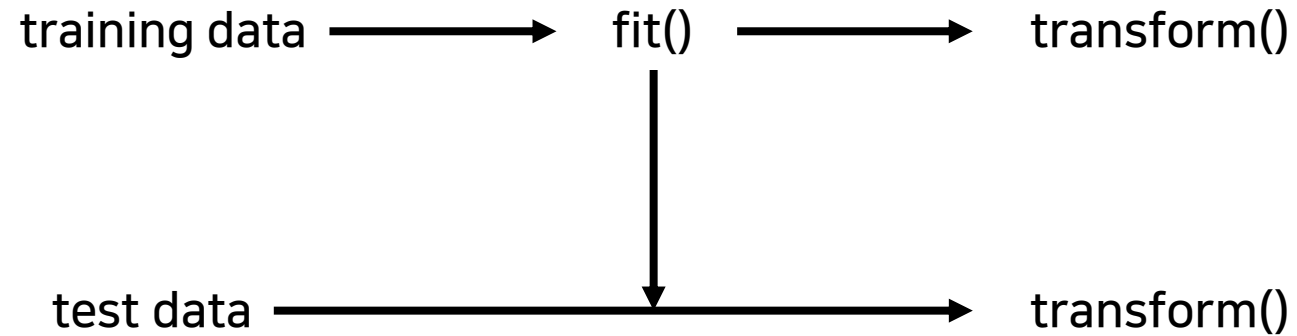


↓  
모델



분석

## fit(), transform(), fit\_transform()



test data에는 fit\_transform() 메소드를 이용하여 변환하면 안됨



## fit(), transform(), fit\_transform() Testdata에 fit()을 적용할 때 발생하는 문제 예시

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

# 학습 데이터는 0부터 10까지, 테스트 데이터는 0부터 5까지 값을 가지는 데이터 세트로 생성
# Scaler 클래스의 fit(), transform()은 2차원 이상 데이터만 가능하므로 reshape(-1,1)로 차원 변경
train_array = np.arange(0,11).reshape(-1,1)
test_array = np.arange(0,6).reshape(-1,1)

# MinMaxScaler 객체에 별도의 feature_range파라미터 값을 지정하지 않으면 0~1 값으로 변환
scaler = MinMaxScaler()

# fit()하게 되면 train_array 데이터의 최솟값이 0, 최대값이 10으로 설정.
scaler.fit(train_array)

# 1/10 scale로 train_array 데이터 변환함. 원본10->1 로 변환됨.
train_scaled = scaler.transform(train_array)

print('원본 train_array 데이터:', np.round(train_array.reshape(-1),2))
print('Scale된 train_array 데이터:', np.round(train_scaled.reshape(-1),2))

원본 train_array 데이터: [ 0 1 2 3 4 5 6 7 8 9 10]
Scale된 train_array 데이터: [0. 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
```

1 → 0.1

## fit(), transform(), fit\_transform() Testdata에 fit()을 적용할 때 발생하는 문제 예시

### \* test\_data에 fit()을 적용한 결과

```
# MinMaxScaler에 test_array를 fit()하게 되면 원본 데이터의 최솟값이 0, 5로 설정됨  
scaler.fit(test_array)
```

```
# 1/5 scale로 test_array 데이터 변환함. 원본 5->1 로 변환.  
test_scaled = scaler.transform(test_array)
```

```
print('원본 test_array 데이터:', np.round(test_array.reshape(-1), 2))  
print('Scale된 test_array 데이터:', np.round(test_scaled.reshape(-1), 2))
```

```
원본 test_array 데이터: [0 1 2 3 4 5]  
Scale된 test_array 데이터: [0.  0.2 0.4 0.6 0.8 1. ]
```

1 → 0.2

### \* train\_data의 fit()을 이용한 결과

```
# test_array에 Scale 변환을 할 때는 반드시 fit()을 호출하지 않고 transform()만으로 변환해야 함  
test_scaled = scaler.transform(test_array)  
print('\n원본 test_array 데이터:', np.round(test_array.reshape(-1), 2))  
print('Scale된 test_array 데이터:', np.round(test_scaled.reshape(-1), 2))
```

```
원본 test_array 데이터: [0 1 2 3 4 5]  
Scale된 test_array 데이터: [0.  0.1 0.2 0.3 0.4 0.5]
```

1 → 0.1

## `fit()`, `transform()`, `fit_transform()`

1. 가능한 전체 데이터의 스케일링 변환을 적용한 후 테스트 데이터 분리
2. 1이 불가능하다면 테스트 데이터 변환 시에는 `fit()`이나 `fit_transform()` 적용하지 않고 학습데이터로 이미 `fit()`된 `Scalar`객체를 이용하여 `transform()`으로 변환

# Q & A

**감사합니다 ^^**