파이썬 라이브러리를 활용한 데이터 분석

7장 데이터 정제 및 준비

2020.06.26금 2h

데이터 정제 및 준비

- **데이터를 합치고 재배열 필요**
  - 원천 데이터는 분석하기 어려운 형태로 기록되어 제공
- **주요 내용**
  - 누락된 데이터 처리
    - **isnull()**
    - **np.nan**
    - **dropna(), fillna()**
  - 데이터 변형
    - **duplicated()**
    - **drop_duplicates()**
    - **map()**
    - **replace()**
    - **cut(), qcut()**
  - 문자열 다루기
    - **정규 표현식**

파일 ch07-study.ipynb

7장 데이터 정제 및 준비

누락 데이터 처리
테이터 변형

7.1 누락된 데이터 처리

- **전처리**
  - 누락된 데이터(결측치) 처리가 중요
    - **판다스는 기술 통계에서는 누락 데이터 제외**
    - **출력 표기: NaN**
    - **코드: np.nan**
      - **R에서 NA(not available) 차용**
      - **파이썬 None도 결측치로 취급**
- **NA 처리 메소드**

Table 7-1. NA handling methods

Argument	Description
`dropna`	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
`fillna`	Fill in missing data with some value or using an interpolation method such as `'ffill'` or `'bfill'`.
`isnull`	Return boolean values indicating which values are missing/NA.
`notnull`	Negation of `isnull`.

메소드 dropna()

- **기본**
  - 결측치가 하나라도 있는 행이나 열을 제거
    - **시리즈에서는 NA 제거**
  - 옵션 how='all'
    - **모두 NA이면 행 제거**
  - 옵션 axis=1
    - **열에 대한 제거**
  - 옵션 thresh=2
    - **NA가 2개 이상이면 행 제거**

```
In [27]: df = pd.DataFrame(np.random.randn(7, 3))

In [28]: df.iloc[:4, 1] = NA

In [29]: df.iloc[:2, 2] = NA

In [30]: df
Out[30]:
          0         1         2
0 -0.204708       NaN       NaN
1 -0.555730       NaN       NaN
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741

In [31]: df.dropna()
Out[31]:
          0         1         2
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741

In [32]: df.dropna(thresh=2)
Out[32]:
          0         1         2
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

결측치 채우기, fillna()

- **메소드 df.fillna(값)**
  - 값은 사전으로도 가능
    - **키가 정수로 열 첨자가 됨**
  - 새로운 객체를 반환
    - **인자 inplcae=True로 기존 객체 수정 반영이 가능**
  - 평균 mean()으로 결측 값을 대입
    - **data.fillna(data.mean())**

```
In [19]: df
```

Out[19]:

	0	1	2
0	0.476985	3.248944	-1.021228
1	-0.577087	0.124121	0.302614
2	0.523772	NaN	1.343810
3	-0.713544	NaN	-2.370232
4	-1.860761	NaN	NaN
5	-1.265934	NaN	NaN

```
In [20]: df.fillna({1: 0.5, 2: 0})
```

Out[20]:

	0	1	2
0	0.476985	3.248944	-1.021228
1	-0.577087	0.124121	0.302614
2	0.523772	0.500000	1.343810
3	-0.713544	0.500000	-2.370232
4	-1.860761	0.500000	0.000000
5	-1.265934	0.500000	0.000000

결측치 채우기, fillna()

- 보간(interpolation) 방법, 옵션 method=
  - 결측치를 채워 넣는 방법
  - 옵션 method='ffill': forward fill
    - 바로 이전 열 값(위 값)으로 삽입
- 결측 치를 최대 몇 개까지 대입, 옵션 limit=

```
In [21]: df = pd.DataFrame(np.random.randn(6, 3))
         df.iloc[2:, 1] = NA
         df.iloc[4:, 2] = NA
         df
```

Out[21]:

	0	1	2
0	0.332883	-2.359419	-0.199543
1	-1.541996	-0.970736	-1.307030
2	0.286350	NaN	-0.753887
3	0.331286	NaN	0.069877
4	0.246674	NaN	NaN
5	1.327195	NaN	NaN

```
In [22]: df.fillna(method='ffill')
```

Out[22]:

	0	1	2
0	0.332883	-2.359419	-0.199543
1	-1.541996	-0.970736	-1.307030
2	0.286350	-0.970736	-0.753887
3	0.331286	-0.970736	0.069877
4	0.246674	-0.970736	0.069877
5	1.327195	-0.970736	0.069877

```
In [23]: df.fillna(method='ffill', limit=2)
```

Out[23]:

	0	1	2
0	0.332883	-2.359419	-0.199543
1	-1.541996	-0.970736	-1.307030
2	0.286350	-0.970736	-0.753887
3	0.331286	-0.970736	0.069877
4	0.246674	NaN	0.069877
5	1.327195	NaN	0.069877

Table 7-2. fillna function arguments

Argument	Description
value	Scalar value or dict-like object to use to fill missing values
method	Interpolation; by default 'ffill' if function called with no other arguments
axis	Axis to fill on; default axis=0
inplace	Modify the calling object without producing a copy
limit	For forward and backward filling, maximum number of consecutive periods to fill

df.fillna() 예제

```
In [24]: data = pd.Series([1., NA, 3.5, NA, 7])
         data

Out[24]: 0    1.0
         1    NaN
         2    3.5
         3    NaN
         4    7.0
         dtype: float64

In [25]: data.fillna(data.mean())

Out[25]: 0    1.000000
         1    3.833333
         2    3.500000
         3    3.833333
         4    7.000000
         dtype: float64
```

7.2 데이터 변형

- **중복 제거하기**
  - 메소드 duplicated()
    - **각 로우가 중복인지를 알려주는 불리안 시리즈를 반환**
  - 메소드 drop_duplicates()
    - **중복인 행을 제거하고 데이터프레임을 반환**
  - 메소드 dropduplicates(['k1'])
    - **중복 체크 열 k1을 지정**
  - 메소드 dropduplicates(['k1', 'k2'], keep = 'last')
    - **마지막 중복된 행을 남긴 결과를 반환**

메소드 duplicated()

- 각 행이 중복인지를 알려주는 논리 Series 반환
- 메소드 drop_duplicates()
  - duplicated 배열이 False인 데이터프레임 반환

```
In [26]: data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
                              'k2': [1, 1, 2, 3, 3, 4, 4]})
         data
```

Out[26]:

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4
6	two	4

```
In [27]: data.duplicated()
```

```
Out[27]: 0    False
         1    False
         2    False
         3    False
         4    False
         5    False
         6     True
         dtype: bool
```

```
In [28]: data.drop_duplicates()
```

Out[28]:

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4

duplicated(['키1', '키2', ...])

- 여러 키로 중복 검사
- 옵션 keep='last'
  - 마지막으로 발견된 값을 반환
    - **기본은 처음 발견된 값을 유지**

```
In [22]: data.drop_duplicates(['k1', 'k2'], keep='last')
```

Out[22]:

	k1	k2	v1
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
6	two	4	6

5가 지워지고 6이 남음

```
In [29]: data['v1'] = range(7)
         data
```

Out[29]:

	k1	k2	v1
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
5	two	4	5
6	two	4	6

```
In [31]: data.duplicated(['k1'])
```

Out[31]:

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

```
In [32]: data.drop_duplicates(['k1'])
```

Out[32]:

	k1	k2	v1
0	one	1	0
1	two	1	1

함수나 매핑을 이용한 데이터 변형

- **series.map(사전)**
  - 사전에 따라 값을 변경
    - **데이터의 요소별 변환 및 데이터를 다듬는 작업을 편리하게 수행**

```
[33]: data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon',
                                    'Pastrami', 'corned beef', 'Bacon',
                                    'pastrami', 'honey ham', 'nova lox'],
                           'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
      data
```

[33]:

	food	ounces
0	bacon	4.0
1	pulled pork	3.0
2	bacon	12.0
3	Pastrami	6.0
4	corned beef	7.5
5	Bacon	8.0
6	pastrami	3.0
7	honey ham	5.0
8	nova lox	6.0

```
In [34]: meat_to_animal = {
           'bacon': 'pig',
           'pulled pork': 'pig',
           'pastrami': 'cow',
           'corned beef': 'cow',
           'honey ham': 'pig',
           'nova lox': 'salmon'
         }
```

```
In [35]: lowercased = data['food'].str.lower()
         lowercased
```

```
Out[35]: 0          bacon
         1    pulled pork
         2          bacon
         3       pastrami
         4    corned beef
         5          bacon
         6       pastrami
         7     honey ham
         8       nova lox
         Name: food, dtype: object
```

```
In [36]: data['animal'] = lowercased.map(meat_to_animal)
         data
```

Out[36]:

	food	ounces	animal
0	bacon	4.0	pig
1	pulled pork	3.0	pig
2	bacon	12.0	pig
3	Pastrami	6.0	cow
4	corned beef	7.5	cow
5	Bacon	8.0	pig
6	pastrami	3.0	cow
7	honey ham	5.0	pig
8	nova lox	6.0	salmon

```
In [37]: data['food'].map(lambda x: meat_to_animal[x.lower()])
```

```
Out[37]: 0      pig
         1      pig
         2      pig
         3      cow
         4      cow
         5      pig
         6      cow
         7      pig
         8   salmon
         Name: food, dtype: object
```

변형 메소드

- **함수 적용**
  - df.map(meat_to_animal)
- **값 치환**
  - df.replace(-999, np.nan)
    - **-999를 mp.nan으로 수정**
  - df.replace([-999, -1000], [np.nan, 0])
    - **-999를 mp.nan으로, -1000을 0으로 수정**
  - df.replace({-999: np.nan, -1000: 0})
    - **-999를 mp.nan으로, -1000을 0으로 수정**

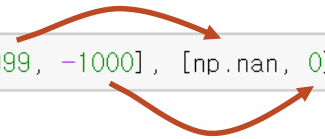
replace(from, to)

- 값 치환

```
In [38]: data = pd.Series([1., -999., 2., -999., -1000., 3.])
         data
Out[38]: 0       1.0
         1    -999.0
         2       2.0
         3    -999.0
         4   -1000.0
         5       3.0
         dtype: float64
```

```
In [39]: data.replace(-999, np.nan)
Out[39]: 0       1.0
         1       NaN
         2       2.0
         3       NaN
         4   -1000.0
         5       3.0
         dtype: float64
```

```
In [40]: data.replace([-999, -1000], np.nan)
Out[40]: 0    1.0
         1    NaN
         2    2.0
         3    NaN
         4    NaN
         5    3.0
         dtype: float64
```



```
In [41]: data.replace([-999, -1000], [np.nan, 0])
Out[41]: 0    1.0
         1    NaN
         2    2.0
         3    NaN
         4    0.0
         5    3.0
         dtype: float64
```

```
In [42]: data.replace({-999: np.nan, -1000: 0})
Out[42]: 0    1.0
         1    NaN
         2    2.0
         3    NaN
         4    0.0
         5    3.0
         dtype: float64
```

축 색인 이름 바꾸기

- 축 색인에 적용하는 map()
- 새로운 객체를 생성: rename()

In [43]:
```
data = pd.DataFrame(np.arange(12).reshape((3, 4)),
                    index=['Ohio', 'Colorado', 'New York'],
                    columns=['one', 'two', 'three', 'four'])
data
```

Out[43]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
New York	8	9	10	11

In [44]:
```
transform = lambda x: x[:4].upper()
data.index.map(transform)
```

Out[44]:
```
Index(['OHIO', 'COLO', 'NEW '], dtype='object')
```

In [45]:
```
data.index = data.index.map(transform)
data
```

Out[45]:

	one	two	three	four
OHIO	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

In [46]:
```
data.rename(index=str.title, columns=str.upper)
```

Out[46]:

	ONE	TWO	THREE	FOUR
Ohio	0	1	2	3
Colo	4	5	6	7
New	8	9	10	11

In [47]:
```
data.rename(index={'OHIO': 'INDIANA'},
            columns={'three': 'peekaboo'})
```

Out[47]:

	one	two	peekaboo	four
INDIANA	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

In [48]:
```
data.rename(index={'OHIO': 'INDIANA'}, inplace=True)
data
```

Out[48]:

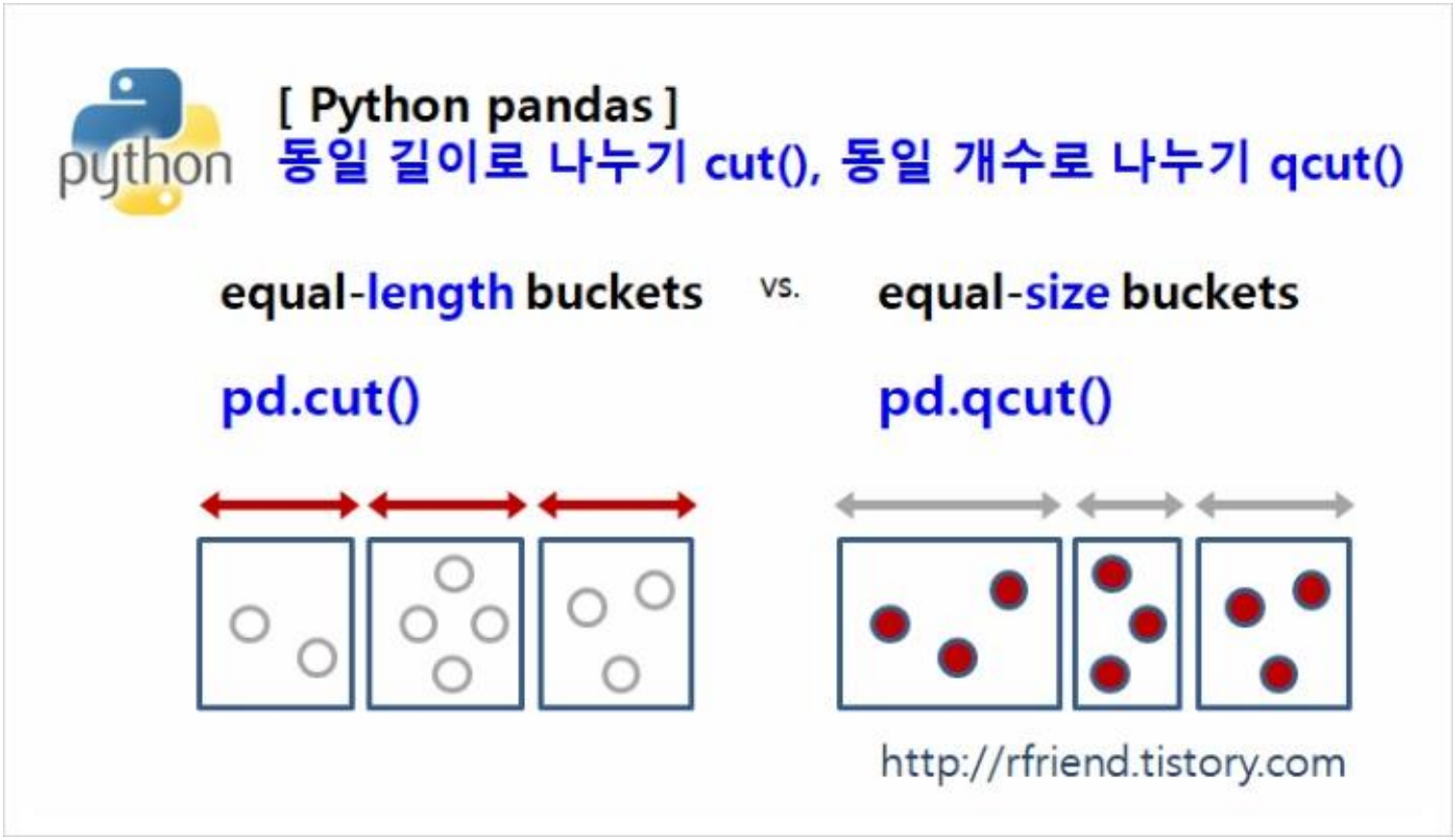
	one	two	three	four
INDIANA	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

7장 데이터 정제 및 준비

구간 구분, 특잇값
원핫 인코딩(표시자/더미 표기)

범주(categories)로 구분

- 함수 cut()과 qcut()



7.2.5 개별화와 양자화

- 함수 pd.cut()
  - 구간 리스트를 사용하여 그에 맞는 구간 그룹으로 나누는 함수
  - 반환 자료
    - **Categorical**
  - 자료형 Categorical의 속성
    - **codes**
    - **categories**
  - 인자 labels
    - **그룹의 이름 지정**

Discretization and Binning

```
In [79]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]

In [80]: bins = [18, 25, 35, 60, 100]
         cats = pd.cut(ages, bins)
         cats

Out[80]: [(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35,
         60], (35, 60], (25, 35]]
         Length: 12
         Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]

In [81]: cats.codes

Out[81]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)

In [82]: cats.categories

Out[82]: IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]],
                       closed='right',
                       dtype='interval[int64]')

In [83]: pd.value_counts(cats)

Out[83]: (18, 25]     5
         (35, 60]     3
         (25, 35]     3
         (60, 100]    1
         dtype: int64
```

N등분 구간으로

- **pd.cut(data, 4, presision=2)**
  - 데이터에서 최대값과 최소값을 기준으로 균등한 길이의 그룹을 자동으로 계산
    - **4등분의 소수점 아래 2자리로 계산**

```
In [86]: data = np.random.rand(20)
         data

Out[86]: array([0.3825, 0.576 , 0.7902, 0.6191, 0.7396, 0.9701, 0.2311, 0.8269,
                0.8363, 0.3881, 0.684 , 0.0145, 0.1015, 0.4436, 0.1118, 0.4374,
                0.8535, 0.7608, 0.3181, 0.8869])

In [87]: print(data.max(), data.min())

         0.9701144223774759 0.014497691036871374

In [88]: pd.cut(data, 4, precision=2)

Out[88]: [(0.25, 0.49], (0.49, 0.73], (0.73, 0.97], (0.49, 0.73], (0.73, 0.97], ..., (0.25,
         0.49], (0.73, 0.97], (0.73, 0.97], (0.25, 0.49], (0.73, 0.97]]
         Length: 20
         Categories (4, interval[float64]): [(0.014, 0.25] < (0.25, 0.49] < (0.49, 0.73] <
         (0.73, 0.97]]
```

함수 qcut()

- **pd.qcut(data, 4)**
  - 4 분위로 분류: 구간 소속 분포의 등분
    - **Data 분포 수가 똑같은 구간으로 나눔**
  - 만일 data의 수가 1000개라면
    - **4개의 구간으로 나누는데 각 구간은 250개씩의 데이터를 가짐**

```
In [91]: data = np.random.randn(1000)  # Normally distributed
         print(data.max(), data.min())

         3.168232720688797 -3.457223616687585

In [92]: cats = pd.qcut(data, 4)  # Cut into quartiles
         cats

Out[92]: [(0.65, 3.168], (-3.4579999999999997, -0.686], (-0.0547, 0.65], (-3.4579999999999997, -0.686], (-0.686, -0.0547], ..., (-0.0547, 0.65], (-3.4579999999999997, -0.686], (-0.686, -0.0547], (-0.0547, 0.65], (0.65, 3.168]]
         Length: 1000
         Categories (4, interval[float64]): [(-3.4579999999999997, -0.686] < (-0.686, -0.0547] < (-0.0547, 0.65] < (0.65, 3.168]]

In [93]: pd.value_counts(cats)

Out[93]: (0.65, 3.168]                    250
         (-0.0547, 0.65]                  250
         (-0.686, -0.0547]                250
         (-3.4579999999999997, -0.686]    250
         dtype: int64
```

함수 qcut()로 분포의 수 비율을 지정

- **분포의 변 위치를 리스트로 직접 지정**
  - 0에서 1 사이로
    - **[0, 0.1, 0.5, 0.9, 1]**
    - **100개라면, 10, 40, 40, 10개의 위치의 그룹으로 나눔**

```
In [100]: cat = pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
          cat

Out[100]: [(0.00401, 1.324], (0.00401, 1.324], (-1.357, 0.00401], (-1.357, 0.00401], (1.324,
          2.82], ..., (-1.357, 0.00401], (1.324, 2.82], (-1.357, 0.00401], (-1.357, 0.0040
          1], (0.00401, 1.324]]
          Length: 1000
          Categories (4, interval[float64]): [(-3.153, -1.357] < (-1.357, 0.00401] < (0.0040
          1, 1.324] < (1.324, 2.82]]
```

```
In [102]: pd.value_counts(cat)

Out[102]: (0.00401, 1.324]      400
          (-1.357, 0.00401]     400
          (1.324, 2.82]         100
          (-3.153, -1.357]      100
          dtype: int64
```

```
In [105]: pd.value_counts(cat, sort=False)

Out[105]: (-3.153, -1.357]      100
          (-1.357, 0.00401]     400
          (0.00401, 1.324]      400
          (1.324, 2.82]         100
          dtype: int64
```

7.2.6 특잇값 찾고 수정하기

- **Outlier**
  - 특잇값을 제외하거나 적당한 값으로 대체
- **칼럼 2에서 2을 초과하는 수 찾기**

In [74]:
```
data = pd.DataFrame(np.random.randn(1000, 4))
data.describe()
```

Out[74]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.004013	0.011546	0.032839	-0.005274
std	0.977132	0.986132	1.017371	1.022018
min	-3.151758	-3.457224	-3.657136	-3.415988
25%	-0.698974	-0.660708	-0.671805	-0.678644
50%	0.006938	0.004542	0.034299	-0.026265
75%	0.665461	0.696627	0.783997	0.701892
max	2.913081	2.774446	4.104784	3.168233

In [75]:
```
data.head()
```

Out[75]:

	0	1	2	3
0	-0.457931	0.072866	-1.044340	1.827439
1	-0.011175	0.263976	-1.364311	0.962825
2	0.528517	-0.589931	0.724138	-0.951306
3	1.226877	1.207842	-0.777700	0.820984
4	-1.664715	0.277523	1.335159	1.225852

In [76]:
```
col = data[2]
col
```

Out[76]:
```
0      -1.044340
1      -1.364311
2       0.724138
3      -0.777700
4       1.335159
          ...
995    -2.322698
996    -0.219481
997     0.243410
998     1.002830
999     1.034200
Name: 2, Length: 1000, dtype: float64
```

In [77]:
```
col[np.abs(col) > 3]
```

Out[77]:
```
148   -3.657136
969    4.104784
Name: 2, dtype: float64
```

In [76]:
```
col = data[2]
col
```

Out[76]:
```
0      -1.044340
1      -1.364311
2       0.724138
3      -0.777700
4       1.335159
          ...
995    -2.322698
996    -0.219481
997     0.243410
998     1.002830
999     1.034200
Name: 2, Length: 1000, dtype: float64
```

any(1)

- 조건을 하나라도 만족하는 값이 있는 모든 행 검색
  - 절대 값이 3을 초과
- 3을 초과하는 수를 -3 또는 3으로 지정

In [78]:
```
data[(np.abs(data) > 3).any(1)]
```

Out[78]:

	0	1	2	3
30	1.197119	0.961372	0.260748	-3.415988
148	1.287969	0.273283	-3.657136	0.574901
264	-0.905403	0.858331	0.553523	3.168233
488	-1.843102	-3.457224	-0.376997	-1.934466
557	-3.151758	-1.799562	-1.221104	1.661051
969	-0.247168	-1.145995	4.104784	0.298836

In [81]:
```
data[np.abs(data) > 3] = np.sign(data) * 3
data[30:31]
```

Out[81]:

	0	1	2	3
30	1.197119	0.961372	0.260748	-3.0

In [79]:
```
data[(np.abs(data) > 3).any(1)]
```

Out[79]:

0	1	2	3

In [61]:
```
np.sign(data).head()
```

Out[61]:

	0	1	2	3
0	-1.0	1.0	-1.0	1.0
1	1.0	-1.0	1.0	-1.0
2	1.0	1.0	1.0	-1.0
3	-1.0	-1.0	1.0	-1.0
4	-1.0	1.0	-1.0	-1.0

In [60]:
```
data.describe()
```

Out[60]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.050286	0.025567	-0.001399	-0.051765
std	0.992920	1.004214	0.991414	0.995761
min	-3.000000	-3.000000	-3.000000	-3.000000
25%	-0.599807	-0.612162	-0.687373	-0.747478
50%	0.047101	-0.013609	-0.022158	-0.088274
75%	0.756646	0.695298	0.699046	0.623331
max	2.653656	3.000000	2.735527	3.000000

7.2.7 치환과 임의 샘플링

- **np.random.permutation()**
  - 행을 임의 순서로 재배치
    - **pd.take(로우_순서_리스트)**

In [82]:
```
df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))
df
```

Out[82]:

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15
4	16	17	18	19

In [83]:
```
sampler = np.random.permutation(5) #0~4의 임의의 순서 조합을 하나 반환
sampler
```

Out[83]:
```
array([0, 1, 3, 4, 2])
```

In [84]:
```
df.take(sampler) #행 순서를 수정
```

Out[84]:

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
3	12	13	14	15
4	16	17	18	19
2	8	9	10	11

임의의 행 선택

- **df.sample(n)**
  - 옵션 replace=True
    - **동일 행의 반복 선택을 허용**

```
In [91]: choices = pd.Series([5, 7, -1, 6, 4])
         choices

Out[91]: 0    5
         1    7
         2   -1
         3    6
         4    4
         dtype: int64

In [96]: draws = choices.sample(n=10, replace=True)
         #draws = choices.sample(n=10) 오류 발생
         draws

Out[96]: 1    7
         1    7
         3    6
         4    4
         4    4
         0    5
         0    5
         4    4
         4    4
         3    6
         dtype: int64

In [94]: choices

Out[94]: 0    5
         1    7
         2   -1
         3    6
         4    4
         dtype: int64
```

원 원소 수가 샘플 수 보다 적으면 반드시 기술

7.2.8 표시자 / 더미 변수 계산

- **함수 get_dummies(키)**
  - 키 열의 모든 종류를 K개의 열로 구성된 데이터프레임을 만들고 속한 값을 표현하는 1과 0을 저장
    - **Convert categorical variable into dummy/indicator variables**
  - 인자 prefix
    - **열 이름의 접두어를 지정해 수정**

In [140]:
```
dummies = pd.get_dummies(df['key'], prefix='key')
dummies
```

Out[140]:

	key_a	key_b	key_c
0	0	1	0
1	0	1	0
2	1	0	0
3	0	0	1
4	1	0	0
5	0	1	0

Computing Indicator/Dummy Variables

In [136]:
```
df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
                   'data1': range(6)})
df
```

Out[136]:

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	b	5

In [137]:
```
pd.get_dummies(df['key'])
```

Out[137]:

	a	b	c
0	0	1	0
1	0	1	0
2	1	0	0
3	0	0	1
4	1	0	0
5	0	1	0

MovieLens 영화 평점 자료 p292

- **첨자 함수 pandas.Index.get_indexer()**
  - Compute indexer and mask for new index given the current index. The indexer should be then used as an input to ndarray.take to align the current data to the new index.
    - **현재 인덱스에서 지정된 새 인덱스에 대한 인덱서 및 마스크 계산**

Examples

```
>>> index = pd.Index(['c', 'a', 'b'])
>>> index.get_indexer(['a', 'b', 'x'])
array([ 1,  2, -1])
```

Notice that the return value is an array of locations in `index` and `x` is marked by -1, as it is not in `index`.

영화 자료에서 장르 추출

- **장르 열에서**
  - |로 구분되어 있는 장르 추출
  - 배열 genres에 저장

In [133]:
```
mnames = ['movie_id', 'title', 'genres']
movies = pd.read_table('datasets/movielens/movies.dat', sep='::',
                       header=None, names=mnames, engine='python')
movies[:10]
```

Out[133]:

	movie_id	title	genres
0	1	Toy Story (1995)	Animation|Children's|Comedy
1	2	Jumanji (1995)	Adventure|Children's|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama
4	5	Father of the Bride Part II (1995)	Comedy
5	6	Heat (1995)	Action|Crime|Thriller
6	7	Sabrina (1995)	Comedy|Romance
7	8	Tom and Huck (1995)	Adventure|Children's
8	9	Sudden Death (1995)	Action
9	10	GoldenEye (1995)	Action|Adventure|Thriller

In [134]:
```
movies.shape
```

Out[134]:
```
(3883, 3)
```

In [135]:
```
all_genres = []
for x in movies.genres:
    all_genres.extend(x.split('|'))
genres = pd.unique(all_genres)
genres
```

Out[135]:
```
array(['Animation', "Children's", 'Comedy', 'Adventure', 'Fantasy',
       'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror',
       'Sci-Fi', 'Documentary', 'War', 'Musical', 'Mystery', 'Film-Noir',
       'Western'], dtype=object)
```

빈 장르 DataFrame 제작

- 행은 영화 수(3883), 열은 모든 장르 수(18)

In [137]:
```
zero_matrix = np.zeros((len(movies), len(genres)))
zero_matrix
```

Out[137]:
```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

In [138]:
```
zero_matrix.shape
```

Out[138]:
```
(3883, 18)
```

In [139]:
```
dummies = pd.DataFrame(zero_matrix, columns=genres)
dummies
```

Out[139]:

	Animation	Children's	Comedy	Adventure	Fantasy	Romance	Drama	Action	Crime	Thriller	Horror	Sci-Fi	Documer
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
...	...	...	...	...	...	...	...	...	...	...	...	...	
3878	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
3879	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
3880	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
3881	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
3882	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	

3883 rows × 18 columns

첫 영화의 장르를 추출해 모든 장르의 열 번호로 색인

- 각 장르의 컬럼을 색인
  - dummies.columns.get_indexer(gen.split('|'))

```
In [140]: gen = movies.genres[0]
          gen.split('|')

Out[140]: ['Animation', "Children's", 'Comedy']

In [141]: dummies.columns

Out[141]: Index(['Animation', 'Children's', 'Comedy', 'Adventure', 'Fantasy', 'Romance',
                 'Drama', 'Action', 'Crime', 'Thriller', 'Horror', 'Sci-Fi',
                 'Documentary', 'War', 'Musical', 'Mystery', 'Film-Noir', 'Western'],
                dtype='object')

In [142]: dummies.columns.get_indexer(gen.split('|'))

Out[142]: array([0, 1, 2], dtype=int64)
```

장르 데이터프레임: 속하는 자기 장르의 칼럼에 1을 저장

In [143]:
```
for i, gen in enumerate(movies.genres):
    indices = dummies.columns.get_indexer(gen.split('|'))
    dummies.iloc[i, indices] = 1
```

In [144]:
```
dummies
```

Out[144]:

	Animation	Children's	Comedy	Adventure	Fantasy	Romance	Drama	Action	Crime	Thriller	Horror	Sci-Fi	Documer
0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
1	0.0	1.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
2	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	
3	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	
4	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
...	...	...	...	...	...	...	...	...	...	...	...	...	
3878	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
3879	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	
3880	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	
3881	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	
3882	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	

3883 rows × 18 columns

함수 get_dummies()와 cut() 활용

- 값의 분포를 표시자/더미(원 인코딩)로 표현

In [170]:
```
np.random.seed(12345)
values = np.random.rand(10)
values
```

Out[170]:
```
array([0.9296, 0.3164, 0.1839, 0.2046, 0.5677, 0.5955, 0.9645, 0.6532,
       0.7489, 0.6536])
```

In [176]:
```
bins = [0, 0.2, 0.4, 0.6, 0.8, 1]
pd.cut(values, bins)
```

Out[176]:
```
[(0.8, 1.0], (0.2, 0.4], (0.0, 0.2], (0.2, 0.4], (0.4, 0.6], (0.4, 0.6], (0.8, 1.
0], (0.6, 0.8], (0.6, 0.8], (0.6, 0.8]]
Categories (5, interval[float64]): [(0.0, 0.2] < (0.2, 0.4] < (0.4, 0.6] < (0.6,
0.8] < (0.8, 1.0]]
```

In [177]:
```
pd.get_dummies(pd.cut(values, bins))
```

Out[177]:

	(0.0, 0.2]	(0.2, 0.4]	(0.4, 0.6]	(0.6, 0.8]	(0.8, 1.0]
0	0	0	0	0	1
1	0	1	0	0	0
2	1	0	0	0	0
3	0	1	0	0	0
4	0	0	1	0	0
5	0	0	1	0	0
6	0	0	0	0	1
7	0	0	0	1	0
8	0	0	0	1	0
9	0	0	0	1	0