Politecnico di Torino

Machine Learning for Networking

Pandas – Extra slides

Andrea Pasini
Flavio Giobergia
Elena Baralis
Gabriele Ciravegna

DataBase and Data Mining Group

Missing values

- Represented with **sentinel** value
  - **None**: Python null value
  - **np.nan**: Numpy Not A Number
- None is a Python **object**:
  - np.array([4, None, 5]) has dtype=Object
- np.NaN is a **Floating point** number
  - np.array([4, np.nan, 5]) has dtype=Float

- Using **nan** achieves better **performances** when performing numerical computations

Missing values

- Pandas supports both **None** and **NaN**, and automatically converts between them when appropriate
- Example:

In [1]:
```
pd.Series([4, None, 5, np.nan])
```

Out[1]:
```
0    4.0
1    NaN
2    5.0
3    NaN
dtype=float64
```

Missing values

- Operating on missing values (for Series and DataFrames)
  - isnull()
    - Return a boolean mask indicating null values
  - notnull()
    - Return a boolean mask indicating not null values
  - dropna()
    - Return filtered data containing null values
  - fillna()
    - Return new data with filled or input missing values

Missing values

- Operating on missing values: **isnull, notnull**
  - Return a new Series/DataFrame with the same shape as the input

In [1]:
```
s1 = pd.Series([4, None, 5, np.nan])
s1.isnull()
```

Out[1]:
```
0     False
1     True
2     False
3     True
dtype=bool
```

Missing values

- Operating on missing values: **dropna**
  - For Series it removes null elements

In [1]:

```
s1 = pd.Series([4, None, 5, np.nan])
s1.dropna()
```

Out[1]:

```
0    4.0
2    5.0
dtype=float64
```

- Operating on missing values: **dropna**
  - For DataFrames it removes **rows** that contain at least a missing value (default behaviour)
    - Passing `how=all` removes rows if they contain all NaN's

Index	Total	Quantity
a	1	2
b	3	NaN
c	5	6

→

Index	Total	Quantity
a	1	2
c	5	6

  - Alternatively, it is possible to remove columns

```
dropped_df = df.dropna(axis='columns')
```

- Operating on missing values: **fillna**
  - Fill null fields with a specified value (for both Series and DataFrames)

In [1]:
```
s1 = pd.Series([4, None, 5, np.nan])
s1.fillna(0)
```

Out[1]:
```
0    4.0
1    0.0
2    5.0
3    0.0
dtype=float64
```

Missing values

- Operating on missing values: **fillna**
  - The parameter **method** allows specifying different filling techniques
    - **ffill**: propagate last valid observation forward
    - **bfill**: use next valid observation to fill gap

In [1]:
```
s1 = pd.Series([4, None, 5, np.nan])
s1.fillna(method='ffill')
```

Out[1]:
```
0    4.0
1    4.0
2    5.0
3    5.0
```

Pivoting

- Pivoting allows inspecting relationships within a dataset
- Suppose to have the following dataset:

```
df = pd.DataFrame({'type':['a','b','b','a','b','a','b','a'],
                   'class':[3,2,3,3,2,1,1,2],
                   'fail':[1,1,1,0,1,0,0,0]})
```

Index	type	class	fail
0	a	3	1
1	b	2	1
2	b	3	1
3	a	3	0
4	b	2	1
5	a	1	0
6	b	1	0
7	a	2	0

- that shows **failures** for sensors of a given type and class during some test

In [1]:
```
df.pivot_table('fail', index='type',
                    columns='class', aggfunc='sum')
```

- Shows the number of **failures** for all the combinations of **type** and **class**

Index	type	class	fail
0	a	3	1
1	**b**	**2**	**1**
2	b	3	1
3	a	3	0
4	**b**	**2**	**1**
5	a	1	0
6	b	1	0
7	a	2	0

Out[1]:
```
class  1  2  3
type
a      0  0  1
b      0  2  1
```

2 sensors of type b and class 2 had some failure

Pivoting

```
In [1]: df.pivot_table('fail', index='type',
                        columns='class', aggfunc='mean')
```

- Shows the percentage of **failures** for all the combinations of **type** and **class**

```
Out[1]: class    1    2    3
        type
        a        0.0  0.0  0.5
        b        0.0  1.0  1.0
```

50% of sensors of type a and class 3 had some failure

Index	type	class	fail
0	**a**	**3**	**1**
1	b	2	1
2	b	3	1
3	**a**	**3**	**0**
4	b	2	1
5	a	1	0
6	b	1	0
7	a	2	0

Multi-Index

- **Multi-Index** allows specifying an index hierarchy for
  - Series
  - DataFrames
- Example: index a Series by city and year

index	city	Rome	Rome	Turin	Turin
	year	2018	2019	2018	2019
	values	10	13	7	9

- Building a **multi-indexed Series**

In [1]:

```
ix = [['Rome', 'Rome', 'Turin', 'Turin'],
      ['2018', '2019', '2018', '2019']]
s1 = pd.Series([10,13,7,9], index=ix)
s1 = s1.sort_index()  # Multi-Index must be sorted
                      # if you want to use slicing
print(s1)
```

Out[1]:

```
Rome   2018    10
       2019    13
Turin  2018     7
       2019     9
```

Multi-Index

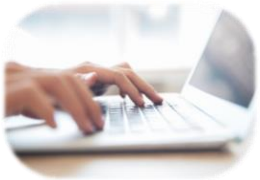
- **Naming** index levels

In [1]:

```
s1.index.names=['city', 'year']
print(s1)
```

Out[1]:

```
city   year
Rome   2018    10
       2019    13
Turin  2018     7
       2019     9
```



- **Accessing index levels**
  - **Slicing** and **simple indexing** are allowed
  - Slicing on index levels follows Numpy rules

In [1]:

```
print(s1.loc['Rome'])       # Outer index level
print(s1.loc[:,'2018'])     # All cities, only 2018
```

Out[1]:

```
year
2018     10
2019     13

city
Rome       10
Turin       7
```

Rome	Rome	Turin	Turin
2018	2019	2018	2019
10	13	7	9

- **Accessing index levels (Examples)**

In [1]:

```
print(s1.loc['Turin', '2018':'2019'])
print(s1[s1>10])     # Masking
```

Out[1]:

```
city    year
Turin   2018     7
        2019     9

city   year
Rome   2019     13
```

Rome	Rome	Turin	Turin
2018	2019	2018	2019
10	13	7	9

- **Multi-indexed DataFrame**
  - Specify a multi-index for **rows**
  - **Columns** can be multi-indexed as well

		Humidity		**Temperature**	
		max	min	max	min
Turin	2018	33	48	6	33
	2019	35	45	5	35
Rome	2018	40	59	2	33
	2019	41	57	3	34

- **Multi-indexed DataFrame: creation**

In [1]:

```
ix = [['Rome', 'Rome', 'Turin', 'Turin'],
      ['2018', '2019', '2018', '2019']]
cols = [['c1','c1','c2','c2'],['a','b','a','b']]
data = np.arange(16).reshape((4,4))
df = pd.DataFrame(data, index=ix, columns=cols)
print(df)
```

Out[1]:

```
             c1      c2
              a   b   a   b
Rome   2018   0   1   2   3
       2019   4   5   6   7
Turin  2018   8   9  10  11
       2019  12  13  14  15
```

- **Multi-indexed DataFrame:** access with **outer** index level

In [1]:
```
print(df['c1'])                # Access by column
print(df.loc['Rome', 'c1']) # Access rows and cols
```

Out[1]:
```
             a   b
Rome  2018   0   1
      2019   4   5
Turin 2018   8   9
      2019  12  13

      a  b
2018  0  1
2019  4  5
```

		c1		c2	
		a	b	a	b
Rome	2018	0	1	2	3
	2019	4	5	6	7
Turin	2018	8	9	10	11
	2019	12	13	14	15

Multi-Index

- **Multi-indexed DataFrame:** access with **outer** and **inner** index levels

In [1]:
```
df['c1', 'a']                # Access by column
```

Out[1]:
```
Rome   2018    0
       2019    4
Turin  2018    8
       2019   12
```

		c1		c2	
		a	b	a	b
Rome	2018	0	1	2	3
	2019	4	5	6	7
Turin	2018	8	9	10	11
	2019	12	13	14	15

- **Multi-indexed DataFrame:** access with **outer** and **inner** index levels

In [1]:

```
ix = pd.IndexSlice
df.loc[ix['Rome', '2018'], ix['c1':'c2', 'a']]
```

Out[1]:

```
c1  a    0
c2  a    2
```

		c1		c2	
		a	b	a	b
Rome	2018	0	1	2	3
	2019	4	5	6	7
Turin	2018	8	9	10	11
	2019	12	13	14	15

Multi-Index

- **Reset Index:** transform index to DataFrame columns and create new (single level) index

In [1]:

```
df.index.names = ['city', 'year']
df_reset = df.reset_index()
print(df_reset)
```

Out[1]:

	city	year	c1		c2	
			a	b	a	b
0	Rome	2018	0	1	2	3
1	Rome	2019	4	5	6	7
2	Turin	2018	8	9	10	11
3	Turin	2019	12	13	14	15

New index

- **Set Index:** transform columns to Multi-Index
  - Inverse function of reset_index()

```
In [1]: df_reset.set_index(['city', 'year'])
```

	city	year	c1		c2	
			a	b	a	b
0	Rome	2018	0	1	2	3
1	Rome	2019	4	5	6	7
2	Turin	2018	8	9	10	11
3	Turin	2019	12	13	14	15

→

city	**year**	c1		c2	
		a	b	a	b
Rome	**2018**	0	1	2	3
	2019	4	5	6	7
Turin	**2018**	8	9	10	11
	2019	12	13	14	15

New index

- **Unstack:** transform multi-indexed Series to a Dataframe

```
myseries.unstack()
```

city	year	
Rome	2018	0
	2019	4
Turin	2018	8
	2019	12

→

	2018	2019
Rome	0	4
Turin	8	12

Multi-Index

- **Stack:** inverse function of unstack()
  - From DataFrame to multi-indexed Series

```
mydataframe.stack()
```

	2018	2019
Rome	0	4
Turin	8	12

→

Rome	2018	0
	2019	4
Turin	2018	8
	2019	12

PoliTo DBMG

- **Aggregates on multi-indices**
  - Allowed by passing the **level** parameter
  - Level specifies the **row granularity** at which the result is computed

```
my_dataframe.max(level='city')
```

city	year	c1		c2	
		a	b	a	b
Rome	2018	0	1	2	3
	2019	4	5	6	7
Turin	2018	8	9	10	11
	2019	12	13	14	15

→

city	c1		c2	
	a	b	a	b
Rome	4	5	6	7
Turin	12	13	14	15

Aggregates on multi-indices

```
my_dataframe.max(level='year')
```

city	year	c1		c2	
		a	b	a	b
Rome	2018	0	1	2	3
	2019	4	5	6	7
Turin	2018	8	9	10	11
	2019	12	13	14	15

→

year	c1		c2	
	a	b	a	b
2018	8	9	10	11
2019	12	13	14	15

Multi-Index

- **Aggregates on multi-indices**
  - Can also aggregate columns
    - Specify axis=1

```
my_dataframe.max(axis=1, level=0)
```

city	year	c1		c2	
		a	b	a	b
Rome	2018	0	1	2	3
	2019	4	5	6	7
Turin	2018	8	9	10	11
	2019	12	13	14	15

→

city	year	c1	c2
Rome	2018	1	3
Rome	2019	5	7
Turin	2018	9	11
Turin	2019	13	15