Data Cleaning and preparation

Week 4/5--Topic 4

jingyun.wang@durham.ac.uk

<<Python for Data Analysis : Data Wrangling with Pandas, NumPy, and Ipython>>

Handing missing data

https://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html

Pandas refers to missing data as **NA**, which stands for not available .

A sentinel value

- For numeric data, pandas uses the floating-point value NaN (Not a Number) to represent missing data.
- The built-in Python **None** value is also treated as NA in object arrays.

```
import numpy as np
import pandas as pd
string_data = pd.Series([None, 'artichoke', np.nan, 'avocado'])
string_data .isnull()
```

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

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`.

Filtering out missing data

On a Series, it returns the Series with only the non-null data and index values:

```
from numpy import nan as NA
data = pd.Series([1, NA, 3.5, NA, 7.9])
data.dropna()
```

→ `data[data.notnull()]`

```
0    1.0
2    3.5
4    7.9
dtype: float64
```

For dataframe, you can drop rows or columns that are all NA or only those containing any NAs. dropna by default drops any row containing a missing value:

```
data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA], [NA, NA, NA], [NA, 6.5, 3.]])
cleaned = data.dropna()
data
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

```
cleaned
```

	0	1	2
0	1.0	6.5	3.0

only want to drop rows that are all NA:

```
data.dropna(how="all")
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
3	NaN	6.5	3.0

Durham University

Examples

To drop columns in the same way, pass axis=1

```
data[4] = NA
data
```

	0	1	2	4
0	1.0	6.5	3.0	NaN
1	1.0	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN
3	NaN	6.5	3.0	NaN

```
data.dropna(axis=1, how='all')
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

Filling in missing data

Calling fillna() with a constant replaces missing values with that value:

```
data.fillna(0)
```

	0	1	2	4
0	1.0	6.5	3.0	0.0
1	1.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	6.5	3.0	0.0

Calling fillna() with a dict, you can use a different fill value for each **column**:

```
data.fillna({1: 5, 2: 0, 4:7})
```

	0	1	2	4
0	1.0	6.5	3.0	7.0
1	1.0	5.0	0.0	7.0
2	NaN	5.0	0.0	7.0
3	NaN	6.5	3.0	7.0

With fillna you can do lots of other things with a little creativity. For example, you might pass the mean or median value of a Series:

```
data=pd.Series([1.0,NA,3.5,NA,7])
data.fillna(data.mean())
```

```
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

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

fillna returns a new object, but you can modify the existing object in-place:

```
data.fillna(0, inplace=True)
```

Examples

The same interpolation methods available for reindexing

	0	1	2	4
0	1.0	6.5	3.0	NaN
1	1.0	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN
3	NaN	2.5	3.5	NaN

ffill (stands for ‘forward fill’):replaces the NULL values with the value from the previous row (or previous column, if the axis parameter is set to 'columns').

```
data.fillna(method="ffill")
```

	0	1	2	4
0	1.0	6.5	3.0	NaN
1	1.0	6.5	3.0	NaN
2	1.0	6.5	3.0	NaN
3	1.0	2.5	3.5	NaN

```
data.fillna(method="ffill",limit=1)
```

	0	1	2	4
0	1.0	6.5	3.0	NaN
1	1.0	6.5	3.0	NaN
2	1.0	NaN	NaN	NaN
3	NaN	2.5	3.5	NaN

Data transformation

Removing Duplicates

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

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

The DataFrame method duplicated() returns a boolean Series indicating whether each row is a duplicate (has been observed in a previous row) or not:

```
data.duplicated()
```

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

drop_duplicates() returns a DataFrame where the duplicated array is False

```
data.drop_duplicates()
```

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

	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

specify any subset of them to detect duplicates.

```
data.drop_duplicates(['k1'])
```

	k1	k2	v1
0	one	1	0
1	two	1	1

by default keep the first observed value combination

```
data.drop_duplicates(["k1","k2"])
```

	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

return the last one:

```
data.drop_duplicates(["k1","k2"], keep="last")
```

	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

Durham University

Transforming Data Using a Function or Mapping

```
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]})
meat_to_animal = { 'bacon': 'pig',
                  'pulled pork': 'pig',
                  'pastrami': 'cow',
                  'corned beef': 'cow',
                  'honey ham': 'pig',
                  'nova lox': 'salmon' }
lowercased = data['food'].str.lower()
lowercased
```

```
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
```

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

	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

Replacing values

```
data = pd.Series([1., -999., 2., -999., -1000., 3.])
data.replace(-999, np.nan)
```

```
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64
```

If you want to replace multiple values at once, you instead pass a list and then the substitute value:

`data.replace([-999, -1000], np.nan)`

To use a different replacement for each value, pass a list of substitutes:

`data.replace([-999, -1000], [np.nan, 0])`

The argument passed can also be a dict:

`data.replace({-999: np.nan, -1000: 0})`

Renaming Axis Indexes

```
data = pd.DataFrame(np.arange(12).reshape((3, 4)),
                    index=['Ohio', 'Colorado', 'New York'],
                    columns=['one', 'two', 'three', 'four'])
##Like a Series, the axis indexes have a map method
transform = lambda x: x[:4].upper()
data.index.map(transform)
```

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

```
##You can assign to index, modifying the DataFrame in-place:
data.index = data.index.map(transform)
data
```

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

```python
##If you want to create a transformed version of a dataset without modifying the original, a useful method is rename :
data.rename(index=str.title, columns=str.upper)
```

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

```python
##Notably, rename can be used in conjunction with a dict-like object providing new values for a subset of the axis labels:
data.rename(index={'OHIO': 'INDIANA'},
            columns={'three': 'peekaboo'})
```

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

```python
#Should you wish to modify a dataset in-place, :
data.rename(index={'OHIO': 'INDIANA'}, inplace=True)
data
```

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

Durham University

Discretization and binning

Continuous data is often discretized or otherwise separated into “bins” for analysis.

Suppose you have data about a group of people in a study, and you want to group them into discrete age buckets:

```
import pandas as pd
ages=[20,22,25,27,21,23,37,31,61,45,41,32]
bins=[18,25,35,60,100]
cats=pd.cut(ages,bins)
cats
```

special Categorical object

```
[(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]]
```

You can change which side is closed by passing right=False :

```
pd.cut(ages, [18, 26, 36, 61, 100], right=False)
```

Indicates whether bins includes the rightmost edge or not.

```
[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 100), [36, 61), [36, 61), [26, 36)]
Length: 12
Categories (4, interval[int64]): [[18, 26) < [26, 36) < [36, 61) < [61, 100)]
```

https://pandas.pydata.org/docs/reference/api/pandas.Categorical.html

```
type(cats)
```

```
pandas.core.arrays.categorical.Categorical
```

Attributes

`categories`	The categories of this categorical.
`codes`	The category codes of this categorical.
`ordered`	Whether the categories have an ordered relationship.
`dtype`	The `CategoricalDtype` for this instance.

```
cats.categories
```

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

```
cats.codes
```

```
array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)
```

```
pd.value_counts(cats)
```

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

Durham University

You can also pass your own bin names by passing a list or array to the labels options:

```
group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']
pd.cut(ages, bins, labels=group_names)
```

```
[Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior, MiddleAged, MiddleAged, YoungAdult]
Length: 12
Categories (4, object): [Youth < YoungAdult < MiddleAged < Senior]
```

Detecting and Filtering Outliers

Filtering or transforming outliers is largely a matter of applying array operations. Consider a DataFrame with some normally distributed data:

```
data = pd.DataFrame(np.random.randn(1000, 4))
data.describe()
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.004846	-0.021556	0.015159	0.039501
std	1.015053	0.981478	1.020946	1.008537
min	-3.259632	-2.760693	-2.890439	-3.276493
25%	-0.655477	-0.699397	-0.704225	-0.666109
50%	-0.022993	-0.016327	0.003854	0.087594
75%	0.668354	0.615626	0.719253	0.708329
max	3.393005	2.489701	3.240661	3.325200

find values in one of the columns exceeding 3 in absolute value:

```
col=data[2]
col[np.abs(col)>3]
```

```
675    3.099820
823    3.240661
Name: 2, dtype: float64
```

To select all rows having a value exceeding 3 or –3, you can use the boolean DataFrame:

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

	0	1	2	3
228	3.189155	-0.937648	-0.634416	-0.727816
437	-3.145205	-1.085325	-0.787895	-0.482588
440	-3.171539	-0.930211	0.108549	0.362863
488	-0.255291	-1.100287	-0.940546	-3.223819
556	1.089107	0.094619	0.342310	-3.276493
594	3.158020	1.849592	-1.689944	-1.264951
668	-0.759775	0.152895	1.591284	3.131849
675	-0.947180	-0.175683	3.099820	-0.652692
694	1.495299	-1.245924	-1.105193	3.325200
724	3.393005	-0.527329	-0.014588	1.153746
760	3.354250	-1.817975	0.163152	0.820375
823	-0.290860	0.862962	3.240661	0.632690
972	-3.259632	-0.901415	1.453389	1.262064

Values can be set based on these criteria. Here is code to cap values outside the interval –3 to 3:

```
data[np.abs(data) > 3] = np.sign(data) * 3
data.describe()
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.004328	-0.021556	0.014818	0.039544
std	1.009873	0.981478	1.019916	1.005556
min	-3.000000	-2.760693	-2.890439	-3.000000
25%	-0.655477	-0.699397	-0.704225	-0.666109
50%	-0.022993	-0.016327	0.003854	0.087594
75%	0.668354	0.615626	0.719253	0.708329
max	3.000000	2.489701	3.000000	3.000000

The statement np.sign(data) produces 1 and –1 values based on whether the values in data are positive or negative

Permutation and random sampling

```
df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))
df
```

	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

```
sampler = np.random.permutation(5)
sampler
```

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

DataFrame.take(indices, axis=0, **kwargs)

Return the elements in the given positional indices along an axis.

```
df.take(sampler)
```

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

To select a random subset without replacement, you can use the sample method on Series and DataFrame:

```
df.sample(n=3)
```

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

To generate a sample with replacement (to allow repeat choices), pass to sample :

```
choices = pd.Series([5, 7, -1, 6, 4])
draws = choices.sample(n=10, replace=True)
draws
```

```
4    4
3    6
0    5
0    5
2   -1
2   -1
3    6
1    7
0    5
1    7
dtype: int64
```

String manipulation

String Object Methods

Table 7-3. Python built-in string methods

Method	Description
`count`	Return the number of non-overlapping occurrences of substring in the string.
`endswith`	Returns `True` if string ends with suffix.
`startswith`	Returns `True` if string starts with prefix.
`join`	Use string as delimiter for concatenating a sequence of other strings.
`index`	Return position of first character in substring if found in the string; raises `ValueError` if not found.
`find`	Return position of first character of *first* occurrence of substring in the string; like `index`, but returns –1 if not found.
`rfind`	Return position of first character of *last* occurrence of substring in the string; returns –1 if not found.
`replace`	Replace occurrences of string with another string.
`strip`, `rstrip`, `lstrip`	Trim whitespace, including newlines; equivalent to `x.strip()` (and `rstrip`, `lstrip`, respectively) for each element.
`split`	Break string into list of substrings using passed delimiter.
`lower`	Convert alphabet characters to lowercase.
`upper`	Convert alphabet characters to uppercase.
`casefold`	Convert characters to lowercase, and convert any region-specific variable character combinations to a common comparable form.
`ljust`, `rjust`	Left justify or right justify, respectively; pad opposite side of string with spaces (or some other fill character) to return a string with a minimum width.

Regular Expressions

Table 7-4. Regular expression methods

Method	Description
`findall`	Return all non-overlapping matching patterns in a string as a list
`finditer`	Like `findall`, but returns an iterator
`match`	Match pattern at start of string and optionally segment pattern components into groups; if the pattern matches, returns a match object, and otherwise `None`
`search`	Scan string for match to pattern; returning a match object if so; unlike `match`, the match can be anywhere in the string as opposed to only at the beginning
`split`	Break string into pieces at each occurrence of pattern
`sub, subn`	Replace all (`sub`) or first `n` occurrences (`subn`) of pattern in string with replacement expression; use symbols `\1, \2, ...` to refer to match group elements in the replacement string

Vectorized String Functions in pandas

Table 7-5. Partial listing of vectorized string methods

Method	Description
`cat`	Concatenate strings element-wise with optional delimiter
`contains`	Return boolean array if each string contains pattern/regex
`count`	Count occurrences of pattern
`extract`	Use a regular expression with groups to extract one or more strings from a Series of strings; the result will be a DataFrame with one column per group
`endswith`	Equivalent to `x.endswith(pattern)` for each element
`startswith`	Equivalent to `x.startswith(pattern)` for each element
`findall`	Compute list of all occurrences of pattern/regex for each string
`get`	Index into each element (retrieve *i*-th element)
`isalnum`	Equivalent to built-in `str.alnum`
`isalpha`	Equivalent to built-in `str.isalpha`
`isdecimal`	Equivalent to built-in `str.isdecimal`
`isdigit`	Equivalent to built-in `str.isdigit`
`islower`	Equivalent to built-in `str.islower`
`isnumeric`	Equivalent to built-in `str.isnumeric`
`isupper`	Equivalent to built-in `str.isupper`
`join`	Join strings in each element of the Series with passed separator
`len`	Compute length of each string
`lower, upper`	Convert cases; equivalent to `x.lower()` or `x.upper()` for each element

Method	Description
`match`	Use `re.match` with the passed regular expression on each element, returning `True` or `False` whether it matches.
`extract`	Extract captured group element (if any) by index from each string
`pad`	Add whitespace to left, right, or both sides of strings
`center`	Equivalent to `pad(side='both')`
`repeat`	Duplicate values (e.g., `s.str.repeat(3)` is equivalent to `x * 3` for each string)
`replace`	Replace occurrences of pattern/regex with some other string
`slice`	Slice each string in the Series
`split`	Split strings on delimiter or regular expression
`strip`	Trim whitespace from both sides, including newlines
`rstrip`	Trim whitespace on right side
`lstrip`	Trim whitespace on left side

A data cleaning example

https://www.kaggle.com/parulpandey/2020-it-salary-survey-for-eu-region

IT Salary Survey EU 2020

In order to Explore the situation of IT salary, clean the dataset

To explore the following questions :

(1) Which are the top 20 positions among the respondents of the surveys? (Compare the results in 2019 and 2020.)

(2) Which are the top 10 main programming languages among the respondents? (Compare the results in 2019 and 2020.)

(3) What affect the annual income of the respondents

What happen if there are more than one that take the 10th place ?

Typical data cleaning in Natural Language Processing

https://www.machinelearningplus.com/nlp/natural-language-processing-guide/

More than 80% of the data available today is Unstructured Data.

The **texts**, **videos**, **images** which cannot be represented in a tabular form (or in any consistent structured data model) constitute unstructured Data.

Natural Language Toolkit (NLTK)

Text Pre-processing in NLP--nltk and spacy

The raw text data often referred to as text corpus has a lot of noise. There are punctuation, suffices and stop words that do not give us any information. Text Processing involves preparing the text corpus to make it more usable for NLP tasks.

For pre-processing of the raw text, follow the steps

1. Tokenization

The words of a text document/file separated by spaces and punctuation are called as tokens. The process of extracting tokens from a text file/document is referred as tokenization.

2. Sanitization of the strings - making text lowercase, removing all punctuation, replacing all whitespace/newlines with single spaces (' ')

3. Removal of stopwords (e.g. "and", "the")

5. Stemming --reducing a word to its 'root form'.

6. Lemmatization - reducing each word to a root/base form (*similar to stemming, except that the root word is correct and always meaningful.*)

Durham University