STAT4710J SP2024 Midterm RC Part 1

Author: Boyuan Zhang

Data Sampling and Probability (Lec 02)

Sampling

Probability/Random Sampling

- Capable of telling each individual's probability
- No need to have an equal probability for each candidate

Sampling Methods

- Simple random sampling: **Random**
- Random Sampling with replacement: **Random**
- Systematic sampling: **Random**
- Stratified sampling: **Random**
- Cluster sampling: **Random**
- Convenience sampling: **Not random**
- Quota sampling: **Not random**
- Voluntary response sampling and snowball sampling: **Not random**

Bias vs. Chance Error

- Bias: One direction
    1. Selection bias
    2. Response bias: People don't always tell the truth
    3. Non-response bias: No answers
- Chance error: Any direction

Population and Sample

- Target Population
- Sample frame
- Sample

Pandas (Lec 03, 04)

Series

- Creating a Series

```
s = pd.Series([4, -2, 0, 6], index=['a', 'b', 'c', 'd'])
```

- Accessing values

```
# single label
s['a']

# list of labels
s[['a', 'b']]
```

DataFrame

- Creating a DataFrame
    - Using lists and column names
    - From dictionaries
    - From Series

Slicing

- `loc`: Selecting by label, inclusive of both sides.

```
# select a single row
df.loc[0]

# select multiple rows
df.loc[0:4]

# select rows and columns
df.loc[0:4, 'Year':'Party']

# select specific rows and columns
df.loc[[87, 25, 179], ['Year', 'Candidate', 'Result']]
```

- `iloc`: Selecting by position, exclusive of the right index.

```
# select a single row
df.iloc[0]

# select multiple rows
df.iloc[0:4]

# select rows and columns
df.iloc[0:4, 0:4]

# select specific rows and columns
df.loc[[1, 2, 3], [0, 1, 2]]
```

Conditional Selection

- Create boolean arrays and pass them into slicing.

```
df[df['Party'] == 'Republic']
```

- Use `&` and `|` for boolean operations.

```
babynames[(babynames["Sex"] == "F") & (babynames["Year"] < 2000)]
```

Handy Utility Functions

- `df.size`: Returns the value of rows * columns.
- `df.shape`: Returns a tuple `(nrow, ncol)` containing the number of rows and columns, where `nrow = df.shape[0]` and `ncol = df.shape[1]`.
- `df.sample(n, replace=..., ...)`: Sampling without replacement by default. Set `replace=True` otherwise.
- `df.sort_values(by=..., key=..., ascending=..., ...)`: Sort values in ascending order by default. Set `ascending=False` otherwise.
- `series.value_counts()`: Counts unique values in descending order.
- `series.unique()`: Returns an array of unique values.
- `series.apply(func)`: Applies the function `func` to all the values in `Series`.
- String manipulation
  - `series.str.isin()`
  - `series.str.len()`
  - `series.str.startswith()`

Column Manipulation

- Addition: `df[newcol] = ...`
- Deletion: `df.drop(colname, axis=1)`
- Modification: `df.rename({col1: newcol1, col2: newcol2, ...})`

Groupby and Aggregation Functions

`groupby` creates sub-dataframes and `agg` aggregates all the columns with corresponding data types to one row of output to represent the group.

```
# syntax to aggregate all the columns
# may cause errors
df.groupby(colname).agg(aggfunc)

# select specific columns before aggregation
```

```
df.groupby(colname)[[col1, col2, ...]].agg(aggfunc)

# aggregate using lambda functions
df.groupby('Name')[['Count']].agg(lambda x: x.iloc[0])

# aggragate using customized functions
def ratio_to_peak(series):
    return series.iloc[-1]/max(series)

df.groupby('Name')[['Count']].agg(ratio_to_peak)

# can also directly use built-in aggregation functions
# mean(), median(), max(), min(), sum(), count(), size()...
df.groupby(...)[[...]].mean()

# filter subtables that satisfy certain conditions
df.groupby('Name').filter(lambda x: x['num'].sum() > 10)
```

Groupby and Pivot Table

```
# sometimes we want to group by first feature, and then by second feature
df.groupby(['Year', 'Sex']).agg(sum).head()

# use pivot table instead
df_pivot = df.pivot_table(index='Year',
                          columns='Sex',
                          values=['Count', 'Name'],
                          aggfunc=np.max
).head()
```

Join Tables

- Inner join: retains only **matched data** from both tables
- Left join: retains all the rows from the **left table** and sets NaN to the unmatched data in the right table
- Right join: retains all the rows from the **right table** and sets NaN to the unmatched data in the left table
- Outer join: retain all the rows from **both tables** and sets NaN to the unmatched data in the corresponding table

```
# basic syntax
pd.merge(
    left=left_df, # the left table to merge
    right=right_df, # the right table to merge
    how=..., # 'inner' by default
    on=None, # if the column names to merge are the same
    left_on=None, # column name to merge in the left table
    right_on=None, # column name to merge in the right table
    left_index=False, # whether to merge on the index of the left table
    right_index=False # whether to merge on the index of the right table
)
```

Regular Expressions (Lec 05)

```
import re
```

- Syntax

Basic Regex Syntax

The four basic operations for regular expressions.

You can technically do anything with just these basic four (albeit tediously).

|, *, () are **metacharacters**. They manipulate adjacent characters.

operation	order	example	matches	doesn't match
concatenation	3	AABAAB	AABAAB	every other string
or	4	AA|BAAB	AA BAAB	every other string
closure (zero or more)	2	AB*A	AA ABBBBBBA	AB ABABA
group (parenthesis)	1	A(A|B)AAB	AAAAB ABAAB	every other string
		(AB)*A	A ABABABABA	AA ABBA

AB*: A then zero or more copies of B: A, AB, ABB, ABBB

(AB)*: Zero or more copies of AB: ABABABAB, ABAB, ,AB, matches the empty string!

Expanded Regex Syntax

wildcard `.`
Consider: `.*`

character class:
Match one character in `[ ]`

Repeat preceding item `{…}` times

Compare/contrast: `o*`, `o+`, `o?`

operation	example	matches	doesn't match
any character (except newline)	`.U.U.U.`	`CUMULUS` `JUGULUM`	`SUCCUBUS` `TUMULTUOUS`
character class	`[A-Za-z][a-z]*`	`word` `Capitalized`	`camelCase` `4illegal`
repeated exactly a times: {a}	`j[aeiou]{3}hn`	`jaoehn` `jooohn`	`jhn` `jaeiouhn`
repeated from a to b times: {a,b}	`j[ou]{1,2}hn`	`john` `juohn`	`jhn` `jooohn`
at least one	`jo+hn`	`john` `jooooooohn`	`jhn` `jjohn`
zero or one	`joh?n`	`jon` `john`	any other string

23

CC BY NC SA

Convenient Regex Syntax

`\w` [A-Za-z0-9_]
`\d` [0-9]
`\s` whitespace
`+` at least one

`[^…]` negates entire character class

"take this next character literally"

operation	example	matches	doesn't match

built-in **character classes**	`\w+`
	`\d+`
	`\s+`
		`Fawef_03`
		`231231`
		whitespace
			`this person`
			`423 people`
			non-whitespace

character class **negation**	`[^a-z]+`	`PEPPERS3982` `17211!↑å`	`porch` `CLAmS`
escape character	`cow\.com`	`cow.com`	`cowscom`

33

CC BY NC SA

Even More Regular Expression Features

A few additional common regex features are listed above.

- Won't discuss these in lecture, but **might come up** in discussion or hw.
- There are even more features out there!

operation	example	matches	doesn't match
beginning of line	`^ark`	`ark two` `ark o ark`	`dark`
end of line	`ark$`	`dark` `ark o ark`	`ark two`
lazy version of zero or more *?	`5.*?5`	`5005` `55`	`5005005`

Again—The official guide is good!
https://docs.python.org/3/howto/regex.html

hell
Greedy: h.+l matches hell
Lazy: h.+?l matches hel.

Hello World
Greedy: <.+> will match Hello World
Lazy: <.+?> will match and

36

CC BY NC SA

- String Manipulation Functions

Python String	re	pandas Series
`s.lower()` `s.upper()`		`ser.str.lower()` `ser.str.upper()`
`s.replace(...)`	`re.sub(pattern, repl, text)`	`ser.str.replace(pattern, repl, regex=True)`
`s.split(...)`	`re.split(...)`	`ser.str.split(...)`
`s[1:4]`		`ser.str[1:4]`

	`re.findall(pattern, text)`	`ser.str.findall(pattern)`
		`ser.str.extract(pattern)`
		`ser.str.extractall(pattern)`

`'ab' in s`	`re.search(...)`	`ser.str.contains(...)`
`len(s)`		`ser.str.len()`
`s.strip()`		`ser.str.strip()`

- Raw String

Regular String	Raw String
`'ab*'`	`r'ab*'`
`'\\\\section'`	`r'\\section'`
`'\\w+\\s+\\1'`	`r'\w+\s+\1'`

- Capture Group

```
text = '''Observations: 03:04:53 - Horse awakens.
03:05:14 - Horse goes back to sleep.'''
pattern = r'(\d\d):(\d\d):(\d\d) - (.*)'
matches = re.findall(pattern, text)
```

Word Embedding (Lec 06)

- Bag-of-Words Encoding
- N-Gram Encoding
- TF-IDF

- Term Frequency (Importance in a single document)

$$tf(t, d) = \frac{\#\text{ of occurrences of } t \text{ in } d}{\text{total } \#\text{ of words in } d}$$

- Inverse Document Frequency (Rarity factor across documents)

$$idf(t) = \log\left(\frac{\text{total } \#\text{ of documents}}{\#\text{ of documents in which } t \text{ appears}}\right)$$

- TF-IDF

$$tfidf(t, d) = tf(t, d) \cdot idf(t)$$

Data on the Internet (Lec 07, 08)

Introduction to HTTP

- HTTP: Hypertext Transfer Protocol
- Request Methods
  - `GET`: Request data from a specified source
  - `POST`: Send data to the server

```
import requests

# get data
get_res = requests.get('https://bing.com')

# post data
post_res = requests.post('https://httpbin.org/post', data={'name': 'King
Triton'})
```

- HTTP Status Code
  - `200`: Successful request
  - `400`: Bad request
  - `404`: Page not found
  - `500`: Internal server error
  - ...

JSON

- JSON: JavaScript Object Notation
- Structure: Resemble Python dictionaries

```
family_tree =
{'name': 'Grandma',
 'age': 94,
 'children': [{'name': 'Dad',
   'age': 60,
   'children': [{'name': 'Me', 'age': 24}, {'name': 'Brother', 'age': 22}]},
  {'name': 'My Aunt',
   'children': [{'name': 'Cousin 1', 'age': 34},
    {'name': 'Cousin 2',
     'age': 36,
     'children': [{'name': 'Cousin 2 Jr.', 'age': 2}]}]}]}
```

- Accessing values

```
family_tree['children'][0]['children'][0]['age']
```

HTML

- HTML: HyperText Markup Language
- Useful Tags to Know

Element	Description
`<html>`	the document
`<head>`	the header
`<body>`	the body
`<div>`	a logical division of the document
`<span>`	an *inline* logical division
`<p>`	a paragraph
`<a>`	an anchor (hyperlink)
`<h1>, <h2>, ...`	header(s)
`<img>`	an image

- Attributes

```
<img src="king-selfie.png" alt="A photograph of King Triton.">
<!--`src`: source of image-->
<!--`alt`: text to display when the image fails to display-->

<a href="https://bing.com">this link</a> <!--destination of the hyperlink-->

<input id="username"> <!--usually unique-->

<div class="stat4710j">some text</div> <!--not necessarily unique-->
```

Parsing HTML Using Beautiful Soup

```
import requests
import bs4

html_string = requests.get(url)
soup = bs4.BeautifulSoup(html_string)

soup.find(tag, attrs={...}) # finds the first instance of a tag
soup.find_all(tag, attrs={...}) # finds all instances of a tag

# Attributes
soup.find('p').text
soup.find('div').attrs
soup.find('div').get('id')
```

Data Wrangling and EDA (Lec 09)

Structure

- The "shape" of a data file
- Variable Feature Types
    - Quantitative
        - Continuous
        - Discrete
    - Qualitative (Categorical)
        - Ordinal
        - Nominal
- Keys
    - Primary key: Unique, determines the values of the remaining columns
    - Foreign key: references primary keys in other tables

Granularity

- How fine/coarse is each datum?
- What does each record represent?

Scope

- How (in)complete is the data?
    - Does my data cover my area of interest?
    - Are my data too expensive?
    - Does my data cover the right time frame?

Temporality

- How is the data situated in time?
    - Data changes
    - Periodicity
    - Time zone
    - Null values: January 1st 1970, January 1st 1900, ...

Faithfulness (Missing Values)

- How well does the data capture "reality"?
- Deal with missing data
    - Drop records with missing values
    - Keep as `NaN`
    - Imputation/Interpolation
        - Average value
        - Random value
        - Predicted value

Reference

Jingye Lin, midtermRC_Part1, SP2023

Zhitong Tang, Mid_RC_Part1, SU2023

STAT4710J Slides, SP2024