wbs
WARWICK BUSINESS SCHOOL
THE UNIVERSITY OF WARWICK

For the Change Makers

Programming for Data Analytics

Week 7 : Data Processing
Information Systems and Management
Warwick Business School

Remove Duplicate

- If you data contains duplicates, then you may consider removing those data.
- As always, think about the causes of duplicates first.
- To remove duplicated rows, you can use .drop_duplicates().

```
df.drop_duplicates() # By defaults, it removes rows with same values in all columns. It keeps the first occurrence and drops others.
```

subset to specify a set of certain columns to identify duplicates.

keep to specify whether to keep first, last occurrence or none

```
df.drop_duplicates(subset=['Age', 'Cabin', 'Sex'], keep=False)
```

Derive and transform columns

- Sometimes, you may want to create new columns derived from existing ones or transform existing ones.

* Normalization and standardization.

* Log transformation.

➢Continuous to categorial.

➢Dummy variables.

Transformation

- Normalization (more in next week)

```
df_titan['FareNor']=(df_titan['Fare']-
                     df_titan['Fare'].mean())/df_titan['Fare'].std()
```

- Log transformation

df_titan['FareLog'] = np.log(df_titan['Fare']) # zero division

Categorical to Numeric

- get_dummies(column) is a Pandas function used to create dummy variables columns based on unique values in current column. This process is also called one-hot encoding. This function returns a DataFrame.
- `pd.get_dummies(df_titan['Sex'])`
- `df_titan[['Female','Male']] = pd.get_dummies(df_titan['Sex'])`
- You can convert all categorical columns at once by passing DataFrame as argument.
- `pd.get_dummies(df_titan, columns=[])`

Continuous to categorical

- We can group a range of continuous values into a category by using cut(**column**,**catergory**,labels) function. For category, you can pass three types of values:
  1. An integer: defines the number of equal-width categories.
  2. sequence of scalars : Defines the category boundaries allowing for non-uniform width.
  3. IntervalIndex : Defines the exact categories to be used.
- pd.cut(df_titan['Age'],3) # 3 groups.
- pd.cut(df_titan['Age'],[0,19,61,100]) # 3 groups with boundaries.
- df_titan['AgeGroup'] = pd.cut(df_titan['Age'], [0,19,61,100], labels = ['Minor', 'Adult','Elder'])

Derived columns

- Instead of differentiating parent/children and sibling/spouse, we are only interested in family relationships.
- df_titan['Family'] = df_titan["Parch"] + df_titan["SibSp"]
- df_titan.loc[df_titan['Family'] > 0, 'Family'] = 1
- df_titan.loc[df_titan['Family'] == 0, 'Family'] = 0

Split multi-value columns

- Sometimes, you may want to split one column into multiple ones.
  - Datetime -> Year, Month, Date, Hour, Mins, Secs.
  - Name -> Title, First Name, Last Name.
  - Email -> Username, Domain.
- Regular expression can often do the trick.
  - Series.str can be used to access the values of the series as strings and apply several methods to it.
  - extract() is a Series.str method to capture groups in the regex pat as columns in a DataFrame.

Regular Expression

- A Regular Expression, or RegEx, is a sequence of characters that specifies a pattern to be searched.
- RegEx is like a mini "programming language" that embedded in Python, as well as other languages (more or less).
- For example, `\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b` is a regular expression to match valid email addresses.
- Very useful for data collection, extraction and cleaning.
- But requires practice and "trial and error".

Sets []

[] is used to match a single character specified in the brackets..

- [abcd]: Matches either a, b, c or d. It does not match "abcd".
- [a-d]: Matches any one alphabet from a to d.
- [a-] and [-a] | Matches a or -, because - is not being used to indicate a series of characters.
- [a-z0-9] | Matches any character from a to z and also from 0 to 9.

[^] is used to match a single character not specified in the brackets

- [^abc] matches any character that is not a, b and c.

RegEx in Python

- Python has build-in module re for regular expression operation.

re.findall(A, B) will matches all instances of a string or an expression A in a string B and returns them in a list.

```
print(re.findall("o","I love python")) # ['o','o']
```

- Add r before string A to indicate a regular expression.

```
print(re.findall(r"[a-p]","I love python")) # ['l', 'o', 'e', 'p', 'h', 'o', 'n']
print(re.findall(r"[lop]","I love python")) # ['l', 'o', 'p', 'o']
print(re.findall(r"[o-t][v-z]","I love python")) # ['ov', 'py']
```

Special Sequences

\w Matches alphanumeric characters, which means a-z, A-Z, and 0-9. It also matches the ideogram and underscore, _.

print(re.findall(r"\w","**I love**爱 **python3**")) # ['I', 'l', 'o', 'v', 'e', '爱', 'p', 'y', 't', 'h', 'o', 'n', '3']

\W matches any character not included in \w.

\d Matches digits, which means 0-9.

print(re.findall(r"\d","I love python**3**")) # ['3']

print(re.findall(r"\w\d","I love python**n3**")) #['n3']

\D Matches any non-digits.

print(re.findall(r"\D","I love python3")) # ['I', ' ', 'l', 'o', 'v', 'e', ' ', 'p', 'y', 't', 'h', 'o', 'n']

Regular Expression Quantifiers

*	0 or more
+	1 or more
?	0 or 1
{2}	Exactly 2
{2, 5}	Between 2 and 5
{2,}	2 or more
(,5}	Up to 5

Example

- Find all WBS student id in a text, such as u1888888.

```
re.findall(r'u1\d\d\d\d\d\d')
re.findall(r'u1\d{6}') #re.findall(r'u1\d+')
```

\s | Matches whitespace characters, which include the \t (tab space), \n (new line), \r (return), and space characters.

\S | Matches non-whitespace characters.

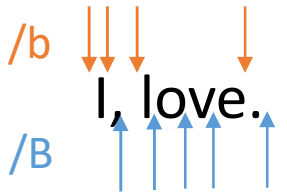
print(re.findall(r"\S","I love python3.")) # ['I', 'l', 'o', 'v', 'e', 'p', 'y', 't', 'h', 'o', 'n', '3', '.']

\b | matches the empty string (zero-width character, not blank space) at the beginning or end, i.e. boundary of a word (\w), in other words, between \w and \W.

\B | matches the any position that is not a word boundary \b.

print(re.findall(r"\w\b","I, love.")) #['I', 'e']

print(re.findall(r"\w\B","I, love.")) # ['l', 'o', 'v']



Special Characters

^ | matches the starting position of the string.

```
print(re.findall(r'^\w','I, love, python')) # ['I']
```

$ | matches the ending position of the string.

```
print(re.findall(r'\w$','I, love, python'))  # ['n']
```

- . | matches any character except line terminators like \n.
- \ | Escapes special characters or denotes character classes.
- A|B | Matches expression A or B.