Data Preprocessing with Python Pandas — Part 2 Data Formatting

Angelica Lo Duca Nov 20, 2020 · 5 min read ★

Image by Pexels from Pixabay

This tutorial explains how to preprocess data using the Pandas library. Preprocessing is the process of doing a pre-analysis of data, in order to transform them into a standard and normalized format. Preprocessing involves the following aspects:

- missing values
- data formatting
- data normalization
- data standardization
- data binning

In this tutorial we deal only with data formatting. In my previous tutorial I dealt with missing values.

Data formatting is the process of transforming data into a common format, which helps users to perform comparisons. An example of not formatted data is the following: the same entity is referred in the same column with different values, such as New York and NY.

You can download the source code of this tutorial as a Jupyter notebook from my Github Data Science Repository.

Import data

In this tutorial we will use the dataset related to Twitter, which can be downloaded from this link.

Firstly, import data using the pandas library and convert them into a dataframe. Through the `head(10)` method we print only the first 10 rows of the dataset.

```
import pandas as pd
df = pd.read_csv('tweets.csv')
df.head(5)
```

In this tutorial, we drop all the missing values through the `dropna()` function.

```
df.dropna(inplace=True)
```

Incorrect data types

First of all, we should make sure that every column is assigned to the correct data type. This can be checked through the property `dtypes`.

```
df.dtypes
```

which gives the following output:

```
Tweet Id                      object
Tweet URL                     object
Tweet Posted Time (UTC)       object
Tweet Content                 object
Tweet Type                    object
Client                        object
Retweets Received              int64
Likes Received                 int64
Tweet Location                object
Tweet Language                object
User Id                       object
Name                          object
Username                      object
User Bio                      object
Verified or Non-Verified      object
Profile URL                   object
Protected or Non-protected    object
User Followers                 int64
User Following                 int64
User Account Creation Date    object
Impressions                    int64
dtype: object
```

In our case we can convert the column `Tweet Location` to `string` by using the function `astype()` as follows:

```
df['Tweet Location'] = df['Tweet Location'].astype('string')
```

We can convert all the objects to strings by running the following statements:

```
import numpy as np
obj_columns = df.select_dtypes(include=np.object).columns.tolist()
df[obj_columns] = df[obj_columns].astype('string')
```

The `astype()` function supports all datatypes described at this link.

Make the data homogeneous

This aspect involves categorical and numeric data. Categorical data should have all the same formatting style, such as lower case. In order to format all categorical data to lower case, we can use the following statement:

```
df['Tweet Content'] = df['Tweet Content'].str.lower()
```

Other techniques to make homogeneous categorical data involve the following aspects:

- remove white space everywhere: `df['Tweet Content'] = df['Tweet Content'].str.replace(' ', '')`
- remove white space at the beginning of string: `df['Tweet Content'] = df['Tweet Content'].str.lstrip()`
- remove white space at the end of string: `df['Tweet Content'] = df['Tweet Content']].str.rstrip()`
- remove white space at both ends: `df['Tweet Content'] = df['Tweet Content'].str.strip()`.

Numeric data should have for example the same number of digits after the point. For example, if we want 2 decimal points, we can run the following command: `df['User Following'] = df['User Following'].round(2)`.

Other techniques to make homogeneous numeric data include:

- Round up — Single DataFrame column — `df['User Following'] = df['User Following'].apply(np.ceil)`
- Round down — Single DataFrame column — `df['User Following'] = df['User Following'].apply(np.floor)`.

Different values for the same concept

It may happen that the same concept is represented in different ways. For example, in our dataset, the column `Twitter Location` contains the values `Columbus,OH` and `Columbus, OH` to describe the same concept. We can use the `unique()` function to list all the values of a column.

```
df['Tweet Location'].unique()
```

which gives the following output:

```
<StringArray>
[                                  'Brussels',
                              'Pill, Bristol',
                                  'Ohio, USA',
                                        <NA>,
                             'Cincinnati, OH',
                                   'WKRC TV',
                             'Scottsdale, AZ',
                                'Columbus,OH',
                                 'Columbus, OH',
                                 'DK Diner, USA',
 ...
                                'Kampala, Uganda',
                            'ilorin,kwara state',
                                'Nigeria, Lagos',
                                        'Kigali',
                            'Towcester, England',
 'Heart of the EU (the clue is in the name)',
                           'South West, England',
                                   'Manchester',
```

```
                                'Seattle, WA',
                            'in my happy place']
Length: 106, dtype: string
```

In order to deal with different values representing the same concept, we should manipulate each type of error separately. For example, we can manipulate every string `word,word` in order to insert a space after the comma and have the following output `word, word`. We can define a function, called `set_pattern()` which searches for a specific pattern into a string and then it performs some replacement in the same string, if the pattern is found. In our case we search for all the patterns having the structure `word,word` and then we replace the `,` with `, `. Finally we return the result.

```
def set_pattern(x):
    pattern = r'[(A-Z)]\w+,([A-Z])\w+'
    res = re.match(pattern, x)
    if res:
        x = x.replace(',', ', ')
    return x
```

Now we can apply the function to every value in the column `Tweet Location`. This can be achieved by using the function `apply()` combined with the operator `lambda`. We can specify that the function `apply()` must be applied to every row (through the parameter `axis = 1`) and then through the `lambda` operator we can select the specific row and apply it the function `set_pattern()`.

```
df['Tweet Location'] = df.apply(lambda x: set_pattern(x['Tweet
Location']), axis=1)
```

Summary

In this tutorial I have illustrated how to perform data formatting with Python Pandas. The following three steps are suggested:

- put data in the correct format — this is required when further analysis is required, such as statistical analysis
- make data homogeneous — this is useful especially for textual data which need further analysis, such as sentiment analysis

- use a single value to represent the same concept — this is useful when data grouping is required.

If you would like to learn about the other aspects of data preprocessing, such as data standardization and data normalization, stay tuned…

If you wanted to be updated on my research and other activities, you can follow me on Twitter, Youtube and and Github.

Data Science Data Preprocessing Python Pandas Pandas Dataframe