Data Cleaning Using Python

An Inncretech Publication

In this post, we will describe efficient ways to clean your data. We've broken the information down into 13 steps, including how to find missing values, deleting blank columns, working on strings, removing special characters, and more.

InnCreTech
Innovation Creativity Technology

Table of Contents

Step 1: Import required library

In this blog, we used pandas for basic work on data

```
In [50]: import pandas as pd
```

We also use basic numpy library functionalities for basic data processing

```
In [51]: import numpy as np
```

Step 2: Create a data frame

```
In [52]: df  = pd.DataFrame()
```

Step 3: Import the data file

Pandas support several kinds of data file types like csv, json, xls, html, fwf, hdf, pickle etc and can be used by many engineers to convert from one format to the other as well. Pandas gives this flexibility to the users to work on other python library by converting different file types to the supported one's. For example, if you are using Graphlab library and want to work on xls file type, graphlab may not support, so you need to convert xls file to supported file type like csv to get started.

```
In [53]: df = pd.read_csv('file.csv') # you may import other file format instead of csv

         # df = pd.read_csv('/path/file.csv') # you may mention file path before file n
         ame based on its location

         # df = pd.read_csv('/path/file.csv', encoding = 'utf-8')  # you can replace en
         coding type as per the need
```

Please note that you may have to mention file path and encoding type as per the need.

Step 4: Have first look of the data

Depending on the type of data, we will see head and tail of data to observe at basic level what all kind of data are present in the file. At this point, you may also use other libraries of python and see visuals from basic data.

```
In [54]: df.head()  # shows the first five rows

         # df.head(1) # Equivalent to df.head(n=1)

         # df.tail() # shows the last five rows

         #df.head(n=0) # gives you just columns but not any rows, n=0 means no of rows
          to be displayed is zer0
```

Step 5: Describe individual columns to see more details

Each column can have possible dirty data or duplicated data. With dirty data, I mean missing, unstructured or incomplete data. Any individual column can have many number of inconsistent data that may create problem when analyzing and in order to get rid of them, we first need to figure out what kind of inconsistency the data have.

```
In [55]: df.column_name.describe()

         #df.Business_Name.describe()

         # df.Address_1.describe() # gives detailed information about
```

Describe function can help you analyse the kind of count: 100000
unique: 84505
top: ---- / ? ...
freq: 1607
Name: Column_name, dtype: object

Step 6: Finding missing values

Most of the dataset have some missing values that are termed as Nan, null, not available or missing values. Term NaN is in numeric arrays, None/NaN in object arrays. We first check if we have missing values and then we find some replacing techniques to go further.

```
In [56]: df.isnull().values.any()
```

This will give you boolean result, True or False and based on the result, you may decide further operations on data.

Step 7: Swapping missing values with default values

Missing values can create several problems in later phase of data engineering if not handled efficiently. So, we fill all the missing values with default values. Dropping rows with missing values at this phase can be dangerous and can be done after this step.

```
In [57]: df.column_name = df.column_name.fillna('')

         # df.column_name = df.column_name.fillna('Missing')   # This will put 'Missin
         g' at spot of all missing data.
```

Now, repeat step 6 and see if you still have missing values, hope you get False.

Step 8: Check data type of each column and change if necessary

In step 5, we can observe dtype at the end of description that we get using describe() function and we can see if the data type of each column matches the needed data type, we keep otherwise change. In usual practice, sometimes int/float datatypes are represented as str/objects, in that case, we need to change the data type and convert it into the required format.

```
In [58]: df['column_name'] = df['column_name'].astype(str)

         #df['column_1'] = df['column_1'].astype(str)

         #df['column_2'] = df['column_2'].astype(int)
```

Another way to get all the columns data type together can be as follows:-

```
In [59]: df.dtypes
```

This will give you list of all columns and corresponding data types. Please note that in Python 3, str data type is referred as object data type. Repeat Step 5 and see if values changes and keep comparing dataframe with its previous state and new state.

Step 9: Delete Blank Columns

For large datasets, having columns with all entries blank or null or no data can be deleted. You first need to see if a particular column is useful or not and then decide this step.

```
In [60]: del df['column_name']
```

Another way of doing this is with dropna() function where we can define threshold that we will discuss in later steps. We drop the column having all null values.

```
In [61]: df = df.dropna(axis=1, how='all')

         #df = df.dropna(axis=1, how='any')
```

Now, we get rid of all such columns having almost no data and proceed further.

Step 10: Delete Blank Rows

Depending on the required threshold, we may delete rows with values less than our expected threshold value. For example, if we have thresh = 5, means, we should have at least 5 not-null values in the given row. By default, dropna() function deletes all the rows having at least one null value.

```
In [62]: df.dropna() # Drops all rows with null values

         #df.dropna(thresh=10)  # Drops all rows with less than 10 values
```

Step 11: Working on Strings

Columns with string data types can undergo several data processing. Various methods can be applied that can go beyond the scope of this blog, we will try to briefly explain how things work with strings.

a) Removing white spaces

White spaces can potentially create a major issue of disturbance during processing, cleaning and observing data. In string columns, we need to cleanup white spaces on edges i.e trailing whitespace and replace with default values.

```
In [63]: df['String_column_name'] = df['String_column_name'].str.strip()
```

b) Making string cases consistent

Keeping data in one case, either upper or lower can have many benefits. Usually we keep all column values in one case to perform efficient string matching.

```
In [64]: df['String_column_name'] = df['String_column_name'].str.lower() # converts all
          data in the column in lower case
```

c) Introducing binding in strings

We replace all the white spaces between words and attach them using underscore _ to bing all the strings together with no empty space. We may also use some other key to bind data together and work.

```
In [65]: df['String_column_name'] = df['String_column_name'].str.replace(' ', '_')
```

d) String matching

String matching can be performed to deduplicate data. One value in a string column can be present in many different ways. Ex- location of states of US can be represented in a dataset as `New York` and `New York, NY` . Several algorithms can sort out these problems of string matching like fuzzy matching, similarity scores of column elements etc. and resulting data processing. We will simply keep this task for readers to try as it goes beyond the scope of this blog.

e) Drop duplicates

In most of datasets, there are chances of duplication and after all the required processing, we can further drop all duplicates and get unique non repeating data.

In [45]:
```
df = df.drop_duplicates(subset=['Dup_column1', 'Dup_column2'], keep=False) # drop all duplicates

#df = df.drop_duplicates(subset=['Dup_column1', 'Dup_column2'], keep= 'first') #keeps first occurance of duplicates
#df = df.drop_duplicates(subset=['Dup_column1', 'Dup_column2'], keep= 'last') # keeps last occurance of duplicates
```

f) Remove special characters

In string column, we may have garbage values, Chinese scripts, special characters and many other abnormal text that can create problems in observing data. Occurrences of normal integer values can also create trouble for data scientists to observe. So, its always a good practice to check for columns with string data type if they have integer values or special character. We can replace all such values with default values and again go to previous steps and run the process of cleaning data simultaneously.

In [46]:
```
df['String_column_name'] = df['String_column_name'].str.replace('1','')
df['String_column_name'] = df['String_column_name'].str.replace('2','')
df['String_column_name'] = df['String_column_name'].str.replace('3','')
df['String_column_name'] = df['String_column_name'].str.replace('4','')
df['String_column_name'] = df['String_column_name'].str.replace('0','')
#...
df['String_column_name'] = df['String_column_name'].str.replace('-','')
df['String_column_name'] = df['String_column_name'].str.replace('"','')
df['String_column_name'] = df['String_column_name'].str.replace(' ','')
df['String_column_name'] = df['String_column_name'].str.replace('(','')
df['String_column_name'] = df['String_column_name'].str.replace('.','')
df['String_column_name'] = df['String_column_name'].str.replace('/','')
#...
#...
#...
```

The above process can be done initially or in later stages of data cleaning as per the need. String data processing can be tedious as well as conceptual when it comes to larger datasets. Many algorithms can also play a role in string cleaning within the datasets.

Step 12: Working on integers

Many missing integer data sets can be replaced in several situations with the help of predictions. For example, if we have carbon datasets and we want to replace missing data based on year of emission given that we have trend of 40 years of data, we may give a value that is consistent with the trend based on some algorithm and replace missing data with the trend data. The same thing follows with the integer values.

Step 13: Output and save the cleaned dataframe to the require file format

This step can be done at several stages of the project depending on size of the data. Above mentioned steps can take longer time and its always a good idea to keep saving data in some file if processing takes longer time than usual and terminates in the middle (very less chances but happens)

```
In [47]: df.to_csv('cleaned_data.csv', encoding = 'utf-8') # you may change file type d
         epending on the requirements.
```

The above mentioned steps may or may not be applicable depending on the circumstances and the dataset. We assume all possible ways to clean data based on the datasets and start processing. Many other ways are there to clean data and several processing may further be added to the above mentioned steps before saving. You may try several other ways to come up with new ideas of data cleaning and make good decisions on ways to clean. We hope this blog will help you get started with data cleaning and exploring the ocean of Data Science.

Thank You

If you'd like to learn more, or get in touch
with our experts:
Email us: info@inncretech.com
Skype us: @voipgeek