6. Practical week 5A (Preparing "Raw" Data for Analysis)

6.1. The Data Science things that everybody does but no one really talks about: Cleaning/Filtering/Formatting

In this practical we will cover some basic operations with data frames:

- How to identify the "dirty" (problematic) areas of a data repository
- How to identify the magnitude of the problem
- How to decide about what needs to be done
- A few tricks of how to "clean/fix" the fields that need "repair/small changes"
- Why this part of Data Science is important

6.2. Cleaning real-life data points: It is mostly a "String affair"

Like Cinderella for outsiders in the fairy tale, maybe the most important things are not so obvious in the Data Science "household". However, without her, the bad-mother's household will collapse. Similarly, data cleaning can make or break a Data Science project. Indeed, the professional data scientists spend a very large portion of their time on this task (sometimes 90% - and btw, do not worry, we will give you a clean data repository for your assignment – however, out in the real world this would be completely unrealistic).

There is a simple reason for this: collecting the data is always a messy and error prone process. One of the wisdom sayings in Data Science is that "Better data beats fancier algorithms". There is also a very old adage in Computer Science: "Garbage in… garbage out". If one has a clean data repository, many more tools can be applied to analyze it, and also simpler algorithms need to be devised for further analysis. In this chapter/practical, we will focus on some of the cleaning operations – that will use mostly the Python string class methods. But before that, we will shortly summarize the other previous steps that are important in preparing data for analysis.

The first step is to remove unwanted data points. For example, typical unwanted data are the **duplicates**, which can result when various separate datasets are combined. Duplicates can refer to rows in the table, but also to columns. Another example of unwanted data is the **irrelevant** data, which do not contribute to our analysis. Or, it can be that some entries are just unlikely to be real data (**outliers** or out-of-range data). If these can be identified, they should be eliminated (however, one has to have a good reason to remove an outlier).

The second step is to handle **missing** data. One may have records or columns that contain too often the `NaN` or `None` values. Two decisions can be taken: to **drop** the record or column altogether, or to input values that are **inferred** based on the distribution of the other existing values.

Because this is important, in the exercise below, we will discuss about such an inference, related to one specific data value.

However, the bulk of the work in this practical will be about changing strings of the same data field into a uniform format. You have to download first from Nestor a .py generator file (`fifthWeek_generate_data.py`) and run it; that will generate the .csv file that is named: `phdThesesFranceThirteesToNineties.csv`. You notice that this is a relatively big file. Our newly created data repository has a "**neat** structure", that is, we have a .csv file that has **all its records of the same length** and encoding, and it is (mostly) complete. That means that we can directly read the data into a data frame object in the memory. Before we do that, for the sake of just showing what happens, try to read this file into Excel, by using the import facility via the menu entry **Data/From Text**, indicating the comma as separator. You will have an Excel sheet with the file's records as rows. Observe the columns and try to understand their meaning (as usual for a .csv file you have the column names in the first row of the file/table). Observe the total number of rows/records. Is this the total number? (check with Notepad if Excel coped with all the rows in the file – Notepad also indicates the line number).

NOTE: as you probably can immediately notice, the data is generated by using the `faker` class code, presented in the previous chapter. You can study later the fake data generation code in `fifthWeek_generate_data.py` and play with it, changing it to create other kinds of fake test data.

Any data analysis exercise has a **purpose**, based on a research goal, business problem or opportunity, human curiosity or boredom, etc. Let us assume that our purpose here is to determine, based on the data available:

1. ***What is the average age of when a PhD candidate defended its thesis?***
2. ***What is the average age of female candidates, and also what is the average age of male candidates?***

Let's proceed!

6.3. Identifying the problems with the given data and making initial decision for the cleaning process

First, the file's content has to be read into a data frame object. Start a .py file named `fifthWeekEx1` (for example). Write the code that displays the shape, the columns, and the data types of the data frame. Run the code. This will enable you to see the (true) size and structure of data (number of rows/records, number of columns, and the name and types of the columns). To answer the analysis questions above, we need data only from three of these columns:

- the gender, and
- the year of birth, and
- the year of the PhD thesis defense.

From the data types of the column, you can see that the year of defense is an integer number, which can be used directly in the analysis. For example we can explore the range of this year of defense (as usual, the chosen name of the data frame object is `df`):

```
print('year range:',
      df['year_of_defense'].min(),
      '-',
      df['year_of_defense'].max())
```

Let us explore the gender column. When looking into the data with Notepad or its similar tools on Mac or Linux (this is the so-called preliminary visual inspection) one can immediately notice that there is a variation of the ways this field was completed. It could be possible that such a file was collected from various universities, which each decided to use a specific coding for the gender. Because there are so many rows, we have to find out first how many ways of coding the gender as a string are in this whole data repository. Before we do that we should sort the data frame by the values of the gender column:


```
odf = df.sort_values(by = ['gender'])
```


To identify the variety of the gender strings used, we can use two data frame class methods: one that counts how many kinds of representation are (a method named `.nunique()`) and another that gives the counts for each individual kind of representation (a method named `.value_counts()`):


```
print('ways the gender field appears:',
      df['gender'].nunique(), 'kinds\n',
      'each occurs:\n kind   occurences\n',
      df['gender'].value_counts())
```

We can see now how many kinds of gender strings exists throughout the data, and how many of each.

For the date of birth, do the same thing as above, using the same methods for the respective column in the data-frame. You will notice that the variety of dates of birth is much higher than the gender (almost two degrees of magnitude). A wise decision is to start with the easiest task. Therefore, first we will clean the gender column (follow the guiding text in 6.4), and only after the year of birth column (explained succinctly in 6.5). To finish coding this first .py file, write code that saves the data frame object ordered by gender data into a new. csv file, without the index (index = False, if you remember this from the previous practical). Run the code and remember the name of this output .csv file.

6.4. How to clean a field that has some string variety, but can have only simple (binary) values

Start a new .py file, named for example `fifthWeekEx2`. To keep file sizes manageable, the code in this file will repair the gender field only. First, read the csv file you just saved previously into a date frame object. This cleaning operation is rather simple because the gender field has a **binary**-like value. The column for gender will contain either the string "Female", either the string "Male" (in more modern times that the file contains data about, maybe more gender types are needed).

By looking at the possible values that are now in the column, we notice that there is a "?" value. That clearly says nothing useful about the gender. Because the number of these occurrence is rather low compared with the total number of records, for the time being we will ignore these rows (we come back to it for homework). To drop these few rows now is a typical decision for the Data Science process.

The rest of values can clearly indicate the gender. To replace the strings, we will use the `.apply()` method, like in the previous practical. This method needs always a function (either a defined one, either an anonymous **lambda** function). In the .py file, before the `read_csv()` containing line (as a good structured code should look like), you have to define a function named for example, `repairGender()`, that takes as argument a string named `gender`, and returns a string named `repairedGender`. The repair in this function, can be done by the following code:


```
    repairedGender = gender
# if the string is not repaired after the code below,
# the same value that was sent is returned
    if gender == 'f' or gender == 'female' or gender == 'F' or gender == 'fem.':
        repairedGender = 'Female'
    if gender == 'm' or gender == 'male' or gender == 'M' or gender == 'man':
        repairedGender = 'Male'
```

In the "main" part of the code (after you have to code that reads the csv file), this function can be used as:

```
df['gender'] = df['gender'].apply(repairGender)
```

Which will replace all the strings in the column with the repaired strings.
QUESTION: what happens to the rows that have the `'?'` value for the gender field. Check.

Write code that saves the data frame into an indexless .csv file. Run the code and remember the name of this second csv file you created from the original one.

HOMEWORK 1: the code above is rather clumsy and too big for the task. A good Python programmer could write a much more compact and elegant solution. Try to find a smaller solution. HINT: use the `title()` string method. Make a separate .py file for this solution.

6.5. How to clean a data field that is needed in a computation, but it is a messy string

By visually inspecting the year-of-birth field, you will notice that actually there are only a few characters that have to be cleaned around the year(s): `['c', 'C','.', '(', ')', '-']`

Start a new .py file, named for example `fifthWeekEx3`. The code in this file will repair the gender field only. First, read the file you just saved previously into a date frame object. Define a new function, named for example `repairYearOfBirth()`, very similar in structure with the previous function, which will return a repaired string representing the year of birth in a simple format, like `'1909'` – which can be easily converted into an integer usable in the age calculations. To clean unwanted characters, you can for example use the `replace()` string method, for example like this:


```
repairedYear = repairedYear.replace('c.', '')
```

That replaces the whole string `'c.'` with an empty string in the `repairedYear` string. Write code that replaces all the unwanted characters, except the `'-'`. This appears in the middle of a string in between two substring values that define a year range (the precise year of birth is not known). You can detect the dash (minus) character with the following statement:


```
if '-' in repairedYear:
```

…and use the `split()` string method to get the two strings that represent the year range. Compute the average year value (use `int()` and `round()`), and make sure that the final value is a string. Finish the defined function by returning the repaired year value – as a string.

In the main part of the program, use `apply()` to change the defective year of birth strings with the repaired ones. Write code that saves the data frame into an indexless csv file. Run the code and remember the name of this third file you created from the original one.

6.6. More to do

Start a new .py file, named for example `fifthWeekExercise3`. The code in this file will read the third csv file you saved previously and compute the answers to the two (actually three) questions for the intended data analysis:

1. *What is the average age of when a PhD candidate defended its thesis?*
2. *What is the average age of female candidates, and also what is the average age of male candidates?*

Finally, we have to address the ignored rows (those with `'?'` in the gender field). Given the amount of records and the relatively small number of these rows, the average values will not be really affected, therefore to ignore them is a rather sound Data Science cleaning decision in this particular case.

However, it may be that the number of these rows is higher - you can change the appropriate parameter in the faker code in `fifthWeek_generate_data.py` and have more of these. We could complete the field by hand, in Notepad (after the sorting by the gender value, these rows are nicely grouped together). For a few hundred entries, we could do this manually because we have a pretty good idea which first names denote a French female or male name – and it would take maybe a couple of hours. Naturally, this would be more difficult for a European if the names would be from the Indian Subcontinent, China, Japan, or Asia in general. Also, if the number of rows is in thousands, the manual task becomes untenable.

6.7. For the curious student

Automatic gender recognition is possible to some extent. You may study this further, for example one current project in the ever growing Python modules development ecosystems is:
https://pypi.org/project/gender-guesser/

You could attempt a third homework after this practical, intended for the ambitious students, to use the `Detector()` class developed by Jorg Michael (described for use on the webpage linked above) and automatically complete the gender field for the those rows where we can rely only on the name of the thesis author.

IN MORE DEPTH: We have written a paper in the past, based on our experience on a specific data aggregation process related to a healthcare research project in London. Here, the building of the data repository for the purpose of analysis is theorized into six steps. The paper is well cited and other researchers built their new ideas on our proposed aggregation process. As a student, you have free access to this paper, published in the Journal of Medical Informatics:
https://www.sciencedirect.com/science/article/pii/S1386505606000864