Initial analysis of the text document

After visual inspection of the text document, **a short script was created to understand the basic structure of the .txt file** that is going to be the core of this project.

Very briefly, the data sctructure is as follows: every entry on the document has a maximum of 3 fields (name, URL and description). Some of the entries are missing a field.

The information that was extracted in this initial analysis is:

- Total number of lines: 1975
- Total number of blank spaces: 128
- Total number of comments: 57

After cleaning it is expected that the document will have **1790 valid entries**.

```
with open("Files/data.txt", "r") as f:

    # A list of all the entries is stored in lst variable
    lst = f.readlines()

    # Several counters are initialize
    blank_counter = 0
    comment_counter = 0
    line_counter = 0

    for item in lst:

        line_counter += 1

        #Blank lines always have a new line character (/n)
        if item == "\n":
            blank_counter += 1

        # Comments always start with %, # or * characters
        elif item[0] in "%#*":
            comment_counter += 1

# General information is printed on screen
print(f'Lines: {line_counter}\nBlank spaces: {blank_counter}\
nComments: {comment_counter}\n\nLines after cleaning: {line_counter-
blank_counter-comment_counter}')
```

Data cleaning and transforming

To ensure a correct data processing later on, **document must first be cleaned and transformed** into a manageable object type.

After visual inspection of the text document, it is decided to remove the following elements:

- Comments. All comments start with one of this characters: '%', '#' or '*'
- Blank lines. All blank lines have only a new line character in them: '/n'

After cleaning the data, **it is decided to transfer the document information into a list of sublist**, were each sublist matches an entry of the original document and each item on that sublist matches a field (Name, URL and Description).

```
from string import punctuation, whitespace

def read_and_clean(file, *args):

    '''
    Takes a file path and a variable number of characters (*args).
When executed, it removes any invalid entry (blank
    lines and lines starting with any of the characters introduced as
*args) from the original file. It returns all
    valid entries of the database grouped all together in a list of
lists.
    '''

    # File is opened in reading mode
    with open(file, "r", encoding="utf-8") as f:

        # readlines() converts all lines on a file in a list where
each line is an item. This list is stored in the variable lines.
        lines = f.readlines()

    lst = []

    # Every valid entry is appended into a new list
    for line in lines:
        if not line[0] in args and not line == "\n":
            lst.append(line)

    # The list of strings is converted into a list of lists for later
use in other functions
    lst = list(map(lambda item: item.split(','), lst))

    # Unnecessary characters are removed from each item of the list
    lst_cleaned = []
    strip_chars = punctuation + whitespace

    for index, item in enumerate(lst):
        lst[index] =  list(map(lambda s: s.strip(strip_chars) , item))

    # The list with the valid entries is written into the file,
eliminating the previous content
    with open('Files/data.txt', "w", encoding="utf-8") as f:
        for item in lst:
            f.writelines(f'{item[0]}, {item[1]}, {item[2]}\n')

    return lst

cleaned_data = read_and_clean("Files/data.txt", '#', '*', '%')

print(cleaned_data)
```

Auxiliary functions

To help debbuging and simplifying code in more advanced functions, some auxiliary functions are firstly defined.

features_picker_cleaner function

This function takes the list of lists that was generated and cleaned in the previous function and returns a list of strings containing only one particular feature of the original document. Each feature is given an identifier number as follows:

- 1 for the Name field
- 2 for the URL field
- 3 for the Description field

Also, this function allows to introduce, some optional characters to be removed from each particular feature. For example the name “burrrd.”, which corresponds to one of the entries is replaced by “burrrd” if "." is introduced as an optional argument.

All empty values are stored as 'n/a'.

```
def features_picker_cleaner(entries, feature, *args):

    '''
    Takes a list of sublists (every sublist represents a field in a
database), an integer representing a feature
    (Name, URL or Description) to be filtered of the sublist and a
variable number of leftover characters to be
    cleaned from every individual field. Returns only the filtered
field in the form of a list of strings.
    '''

    lst = []

    # Every item on the list is iterated
    for line in entries:
        # The field of interest is accessed and stored in a variable
        field = line[feature-1]
        #print(field)
        #We add the fields that aren't empty to a new list that will
```

```
only contain the selected feature
        if field:
            lst.append(field)
        else:
            # Empty fields are appended as n/a
            lst.append('n/a')

    # Leftover characters are removed from the items on the feature
list
    for arg in args:
        lst = list(map(lambda s: s.replace(arg, ""), lst))

    return lst

filtered_field = features_picker_cleaner(cleaned_data, 3)
print(filtered_field)
```

counter function

To count the occurrences of a value inside a particular feature, a custom function is created. This function takes the list of strings created in the previous function and returns the number of occurrences of a particular value inside that list of strings.

```
from functools import reduce

def counter(value, where_to_look_for):

    '''
    Takes a list of strings and a value. Returns the number of
occurrences of that value inside the list.
    '''

    # A boolean list is generated, where values that match the
searched value return a 1 and the rest return 0. This boolean
    # list is then summed to obtain the total of occurrences of that
value.
    return sum(list(map(lambda field: 1 if field == value else 0,
where_to_look_for)))


print(counter('Herramientas Instagram', filtered_field))
```

values_and_frequencies function

This function uses the previous function (occurrences counter function) to create a dictionary that maps each unique value of a feature with its number of occurrences.

```
def values_and_frequencies(where_to_look_for, min_num_repetitions=0):
    '''
    Takes a list of strings and a minimum number of repetitions.
```

```
Returns a dictionary that maps each item of the list of strings
    with its frequence of occurrence.

    '''
    dict_freq = dict()

    # A unique values list can also be obtained by using high order
functions as follows:
    # once_list = reduce(lambda x,y: [*x,y] if not y in x else x,
where_to_look_for, [])

    once_list = set(where_to_look_for)

    # The list of unique values is iterated and the fields are passed
to the counter function
    for field in once_list:

        # The number of occurrences of a particular field are
calculated with counter function and stored in a variable
        n_rep = counter(field, where_to_look_for)

        # If the number of occurrences of a particular field surpasses
the defined threshold, then it is stored inside
        # a diccionary
        if(n_rep >= min_num_repetitions):
            dict_freq[field] = n_rep

    return dict_freq

print(values_and_frequencies(filtered_field))
```

filter_names function

This function returns only the second half of a list of strings (filtered feature) after ordering it alphabetically.

```
def filter_names(num, where_to_look_for):

    '''
    Takes a number (that represents the field to be filtered) and a
list of lists. Returns the second half of the filtered field,
    after ordering it alphabetically.

    '''

    # The filtered field is obtained with the features_picker_cleaner
    lst = features_picker_cleaner(where_to_look_for, num)
    len_lst = len(lst)
    # If the length of the list is odd, the floor division by 2 is
used to obtain the starting index of the returned list.
    # Otherwise, if legth is even, normal division is used.
    if len_lst % 2 == 0:
        index = len_lst / 2
    else:
        index = len_lst // 2

    index = int(index)

    # To ensure a correct sorting, lower method is passed as key in
sorted function to ensure that lower and upper cases are
    # sorted simultaneously.
    return sorted(lst[index:], key=lambda word: word.lower())

print(filter_names(2, cleaned_data))
```

domain_extraction function

Used to obtain the unique values for the website domains on the URL feature. Uses the features_picker_cleaner function to obtain a list of strings for a particular feature and to eliminate some leftover characters, such as the Hypertext Transfer Protocol (HTTP). Web paths are also eliminated. Finally, some general information about the number of repeated domains is printed.

```
def domain_extraction(where_to_look_for):
    '''
    Takes a list of lists and extracts all the non-repeating webpage
domains from the second field. Non-repeating
    Webpage domains are finally returned in a list of strings.
    '''

    # Webpage domain field is first filtered from the original list of
lists. Leftover characters to be cleaned are also
    # passed into the function.
    lst = features_picker_cleaner(where_to_look_for, 2, 'http://',
'https://', 'www.')

    #Path is removed to obtain the domain only
    for i in range(len(lst)):
        lst[i] = lst[i].split('/')[0]

    #Information on repeted domains is calculated with the
values_and_frequencies function and is then displayed on screen.
    dic = values_and_frequencies(lst, 2)
    print(f'Number of repeated domains: {sum(dic.values())}')

    return set(lst)
```

```
domain_extraction(cleaned_data)
```

Characteristics_organization function

Similarly to the values_and_frequencies function, this function returns a dictionary that maps every unique value with its number of occurrences. The difference is that this function is made specifically to generate a mapping for the third feature (Description) and does not allow a minimum number of repetitions.

```
def characteristics_organization(where_to_look_for):

    '''
    Takes a list of lists (every sublist represents a field in a
database) and generates a dictionary that maps the names of the
    third field with its occurrences. Entries with no information on
the third field are stored as n/a.
    '''
    # The field is first filtered from the document. A list of strings
containing the field of interest is generated.
    descriptions = features_picker_cleaner(where_to_look_for, 3)

    # Frequency of occurrence is calculated by passing the filtered
field into the values_and_frequencies function.
    return values_and_frequencies(descriptions)

print(characteristics_organization(cleaned_data))
```