IN104: CatchMe $\sim$ *OurSearchEngine*

Clément CAPORAL and Corentin SOUBEIRAN

April 2019

specific search engine in catchme folder on github link
https://github.com/Coohrentiin/IN104_CAPORAL_Clement_SOUBEIRAN_Corentin

Contents

1 Text transformation

1.1 Home made nltk : How to loose time?

Before using directly the nltk library, we tried to code our own text processor, and compared it with ntlk library and classic "regular expression" method.

Our class "text_transformation" in the text.py file splits the text in words, pop stops words and lemmatize verbs.

Function:	Home made	regular expression	nltk
time:	231s	3.6s	153s
treated words:	3 542 183	5 027 515	4 056 125
word per seconds:	6,52e-5	7,16e-7	3,77e-5

Regular expression doesn't re-shape words, that why it appears like the fastest method. In fact we will use nltk: it is the fastest method to re-shape document and treat words.
NLTK stands for Natural Language Toolkit. This toolkit is one of the most powerful NLP library which contains packages to make machines understand human language and reply to it with an appropriate output. Tokenization, Stemming, Lemmatization, Punctuation, Character count, word count are some of these packages which will be discussed in this tutorial.
In our code, these method have been implemented in the method clean_text.

1.2 Text transformation class

To manage the text transformation we created a class 'text transformation' will all what we need to transform the text, thats mean we apply (whitout using nltk library) the:

- Tokenization
- stop words remove
- lemmatization

This class includes:

- init
- split function
- tokenisation: suppression of possesive marks in a new word "not" for the first one and deleted for the second.
- stop words: In nltk script we isolated a list of stop words. So this function check if the word appears in this list.
- irregular verb list
- lemmatization: this function delete grammar by supression of "ed" or "ing" in end of words. We also check if the word is an irregular verbs or not and if so remplace the verb by its infinitive form. This function is clearly not optimized and makes many mistakes for instance by remplacing "Boeing" by "Boe", but we get an order of magnitude for the complexity of such functions

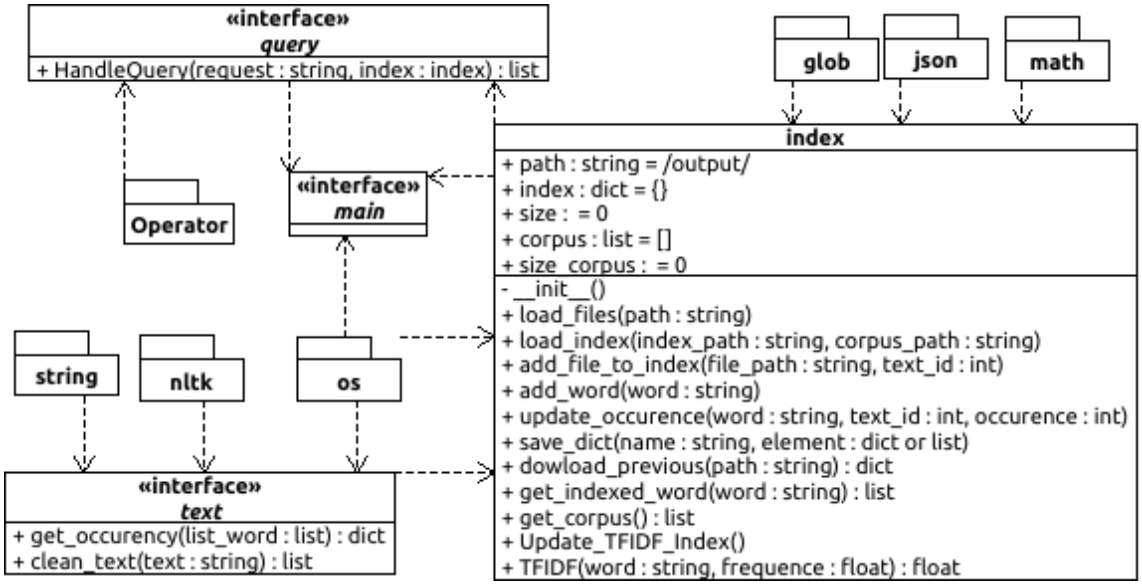

Figure 1: UML Diagram of catch me. The only class is index. We can notice that the different interface are more for code readability than performance as they contain only function.

In the rest of our project, this text_transformation class is remplaced by clean_text function that uses nltk. We combined the text transfomation manage by NLTK in a function "clean text" in "text.py". It takes a text (type:string) and answer "filtered text" (type: list) it's a list of word treated.

2 Algorithmic structure

2.1 UML

In figure 1 we illustrate our algorithme by an UML (Unified Modeling Language). That is representing the links between the different components of the algorithm (libraries, classes, functions ...).

2.2 Index creation

2.2.1 Content

To manage an optimized searching method in a corpus we create a class "index" with serevals functions:

- init : private method that creates basic attributes of the class
- load_files : use to build an index from text files
- load_image : use to build an index from image files
- load index : use to build an index from json file

- add_file_to_index : use to add one file to the current index
- add_word : use to add one word to index
- update_occurence : compute occurence of one word in the index and text
- save_dict : use to save the index dictionnary in json file
- download_previous : load previous json file
- get_indexed_word : use to get occurences of a word in the current index
- get_corpus : return the corpus of the index
- Update_TFIDF_Index : transforms occurences to TFIDF

Here are example of two functions (load_files and add_files_to_index in pseudo code:

```
load_files(path)
    for each document
        add the document to the corpus by using "add\_file\_to\_index"
    transform the occurences to TFIDF value using the current index
    save index in output folder in .jason
    save corpus (correspondance table) in output folder in .json
    return state (0 by default, -1 if it fails

add_files_to_index(file_path,text_id)
    Try to open file, in case of truble: return state=-1.
    read text, and stock it in variable "text".
    close file
    clean "text" by using "clean_text".
    calculate words occurency in a dictionary.
    for each word in the dictionary (that min in the file):
        if this word isn't already in the index:
            add it in the index by using "add_word"
            update occurency in the index by using "update_occurence"
    return 0 (state)
```

2.2.2 Why use a class for index

Using a class for the index offers many advantages : * dynamic add of new text index. If in the future we make a user interface the user could add a text dynamically * we can build different index at the same time using the same class and so have a index for text and an index of image for example * we can access from different interfaces the attributes of the class and update it everywhere for all other methods The method managing the index creation is "load_files(path)" (object of index class). This function complete the corpus with the file found in the default folder (input/) and load each document

2.2.3 Computing time and memory space

2.3 Query

"HandleQuery" is the function to process an answer it take on arguement a user request and an index, it answers a sorted list of documents from the most relevant to least relevant. Order of the documents is based on the calculation of the TF IDF by applying the formula of "cosine similarity" based on the comparison of the angle between the request and the vectorial space formed by the set of words which compose the texts of the corpus.

TF IDF representation: $w_{i,j} = tf_{i,j} * log(N/df_i)$

cosine similarity: $similarity = cos\theta = \frac{A.B}{||A||.||B||} = \frac{\sum_{i=0} A_i.B_i}{\sum_{i=0} A_i^2 \sum_{i=0} B_i^2}$
Where A is TFIDF of the query
B is TFIDF of the document.

The method handling the query is "HandleQuery(request,index)". This function return the ordered list of text based on the similarities with the request and the index. Here is its psuedo code:

```
load_files(path):
    Cleanning request and creation of a list of words by
        using "clean\_text"
    Calculation of words occurency in the request ("get\_occurency").
    for each word in the request:
        calculate TFIDF ($w_{i,q}$ where i is word indexation and q
            the query, it is $A_i$ in cosine similarity)
        for each document where the word appear:
            add to a variable the multiplication $A_i*B_i$ where
                $B_i$ is the TFIDF of the word in the documents studied.
            add $B_i^2$ to a variable
        for each document where the word appear:
            calculate $similarity=cos(\theta)$
        sort document by the similarity value.
```

2.4 Main

"Main" contains the interface between the user and index class. It is decribed in the figure 2 flowchart diagram.

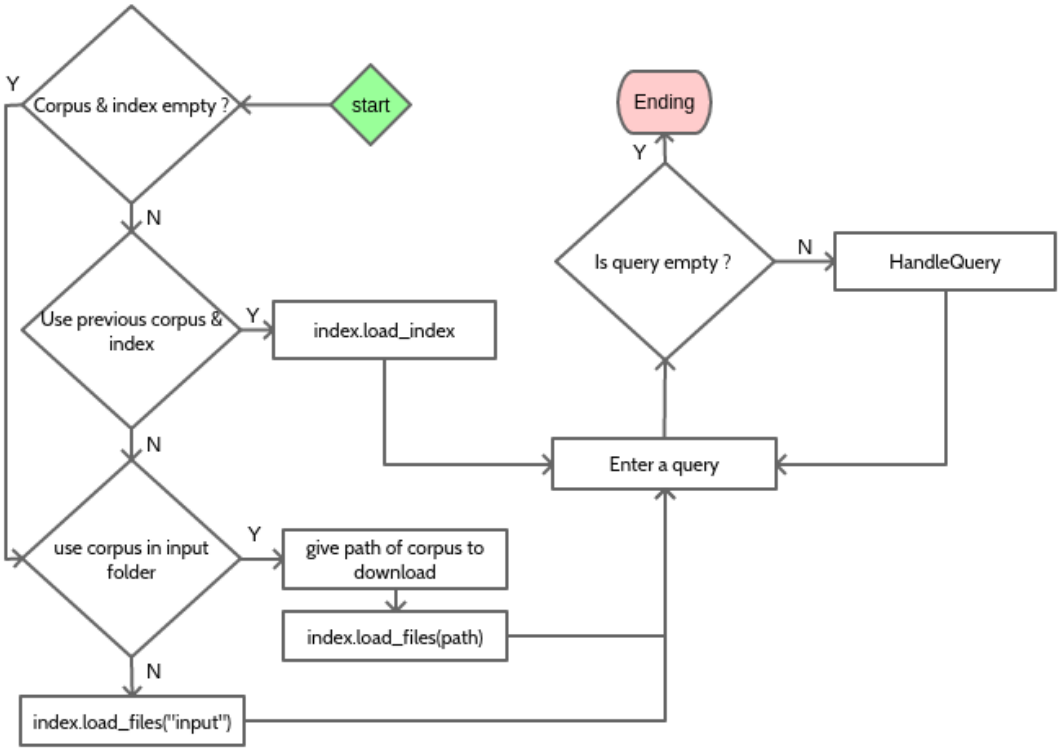

Figure 2: Flowchart of catch me. A user can use different method to build his or her database before handling queries

3 Complements

3.1 Error case management

In order to offer a better user experience, catch me handles the following error cases :

- Index unreadable : return "in loading index: index and corpus unreadable. Please build again a text corpus to create a new index"
- Input file is empty return "in loading files : problems ecountered during the creation of the index from default input/*. Check the following and try again : Check if there are only text files in index/*, Do not manipulate files during the index creation, check if the program has the rights to read the texts"
- Given corpus folder is empty : return "in loading files : problems encountered during the creation of the index. Check the following and try again : Check if there are only text files in the given path, Do not manipulate files during the index creation, try to use the default input/* folder to check if the problem comes from somewhere else"
- Corpus and index are not coherent return "Error: during the processing the manipulation of the index for request treatment. Index doesn't match with corpus. Please quit and rebuild index"

- File unreadable : "Error : path/to/file not found or not *.txt"

For each error we added a potential solution to the user to correct the error. In addition, the location of the error is a human-readable error message.

3.2 Complexity

To possibly compare our algorithm with those of our classmates we propose an estimation of the complexity of our algorithm in the following table (interrogation point mark unkown complexity:

Function:	**Global complexity:**	**Majoration:**	**M:**
load_Files	N+N*O(Nn+n^2) + $O(N^2n^2)$	O(N^2n^2)	O(M^2)
load_index	O(1)	O(1)	O(1)
add_File_ToIndex	O(n)+n^2 + $O(n) * (O(1) or O(nN))$	O(Nn+n^2)	O(M^2)
add_Word	O(1)	O(1)	O(1)
update_occ	nN+O(1)	O(nN)	O(M)
save_dict	? jason lib	? jason lib	X
download_previous	? jason lib	? jason lib	X
Update_TFIDF_index	$N^2n^2 + N$	O(N^2n^2)	O(M^2)
TFIDF	O(1)	O(1)	O(1)
get_Occurency(list)	n^2	O(n^2)	O(1)
tokenization(text)	n^2	O(n^2)	O(M)
clean_text(text)	n	O(n)	O(1)

- n: average word number by document
- N: number of document
- M: number of word in index

Illustrations: figure 3 and 4. This figures shows that theoretical and pratical complexities are good. You can find the code we used for the estimation in test_text.py

3.3 Extension of "CatchMe" for image

Catch me program can also be used to create an index of image.

3.3.1 Similarities

In order to use the same logic for images than text, we select features in the image that are encoded with number. Then one text becomes one image and one word becomes one number. Then we can use the same function (except load image/ load files) than for the text. To create the query we also use a different method and we can use the flowchart diagram as a "normal" user can't create its own image in input.

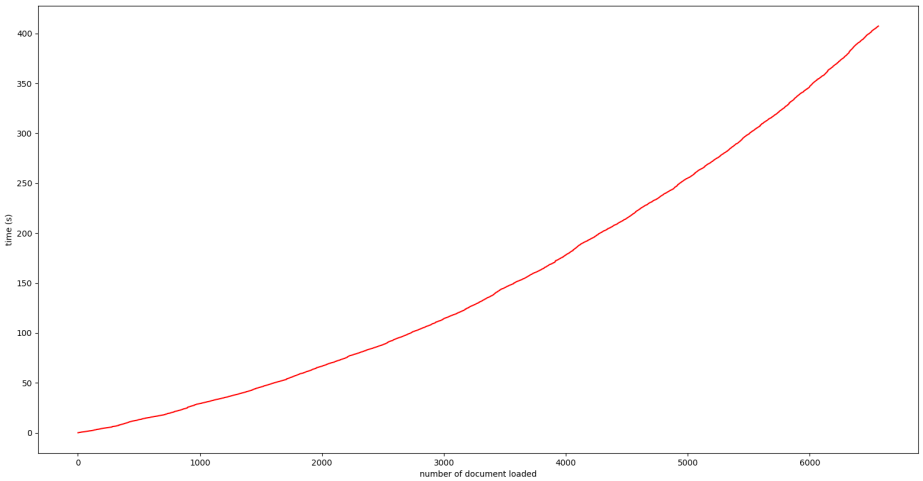

Figure 3: representation of time processing in fonction of the number of document processed

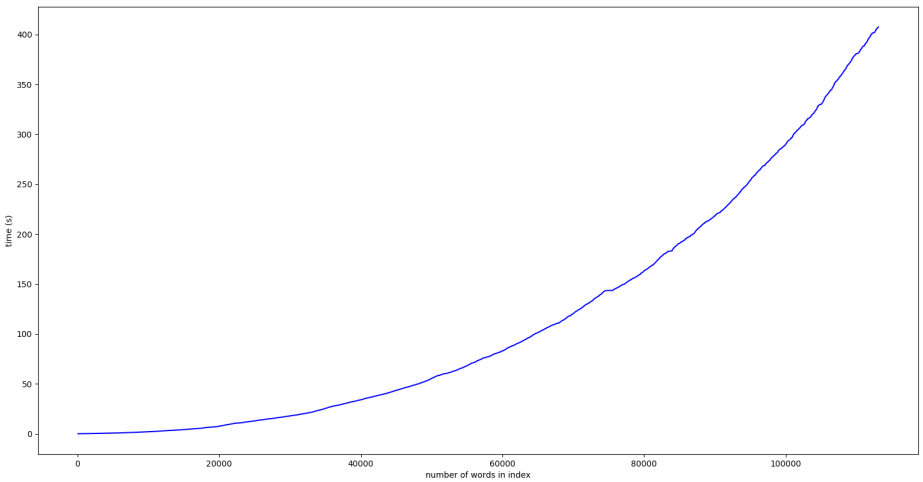

Figure 4: representation of time processing in fonction of the number of words in index

3.3.2 Result's examples

We created an index and launch our main_image to get the following result in figure 5:

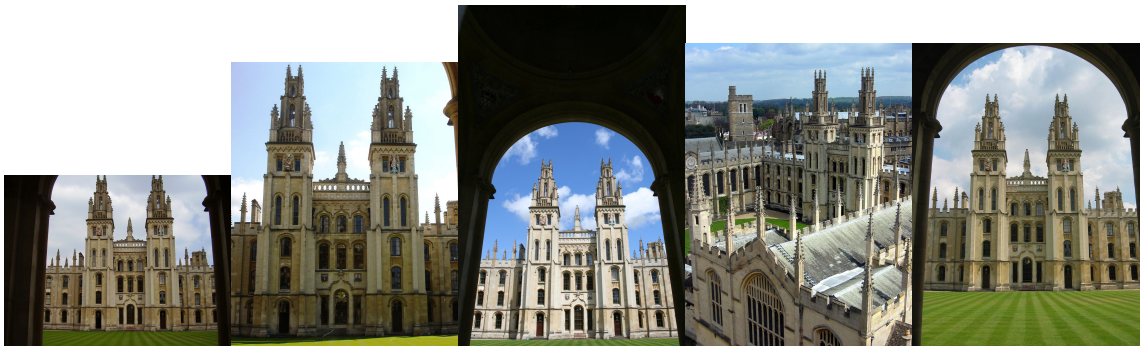

Figure 5: First image is the image we are looking for. From left to right, result from the most relevant to the less relevant. Notice that the fourth image seems strange to be here as it doesn't look like the one we are looking for. Maybe it is due to the fact that our test is based on a very little amount of images (33) and needs more data to better differentiate the images.

4 Development:

4.1 Tools

For this project to code we used Visual Studio Code. The live share help us to code together on the same program. The rest of the time we used git to push and pull the folder, nevertheless we mainly used liveshare. finally for this article we choosed to do it on Sharelatex, to have a commun file, and also to improve our skills with latex.

4.2 TDD: test driven development

For the development of catch me we tried to follow as much as possible the test driven development method. You can see the test we built in testIndex.py and testText.py