EDAN20 - Assignment 1

Oliver Cosic

September 2021

1 Introduction

This assignment is about creating an inverted index. The objectives of the assignment is to:

- Write a program that collects all the words from a set of documents
- Build an index from the words
- Represent a document using the Tf-idf values.

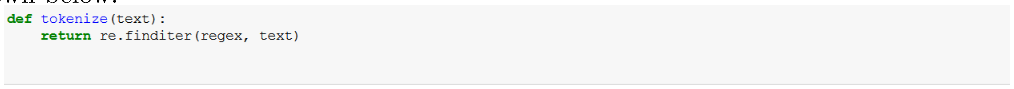
One could say that we split the process of building the inverted index into five steps:

- Index one file
- Read the content of a folder
- Create a master index for all the files
- Use tfidf to represent the documents
- Compare the documents of a collection

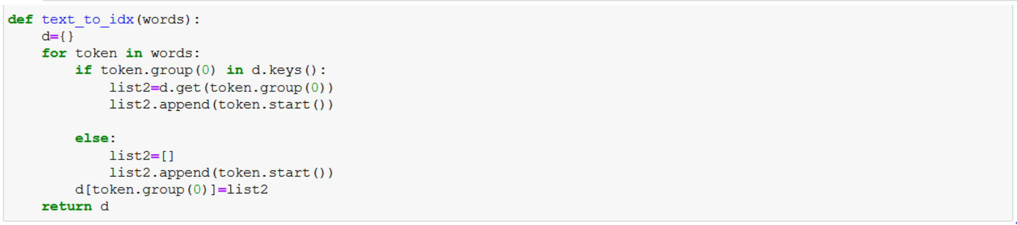
And in this report wil comment on each of these steps briefly.

2 Indexing one file

To index one file we basically wrote two functions called *tokenize* and *texttoidx* shown below:



```
def tokenize(text):
    return re.finditer(regex, text)
```



```
def text_to_idx(words):
    d={}
    for token in words:
        if token.group(0) in d.keys():
            list2=d.get(token.group(0))
            list2.append(token.start())

        else:
            list2=[]
            list2.append(token.start())
        d[token.group(0)]=list2
    return d
```

Where tokenize takes a text and with the help of the regex:

```
regex='\p{L}+'
```

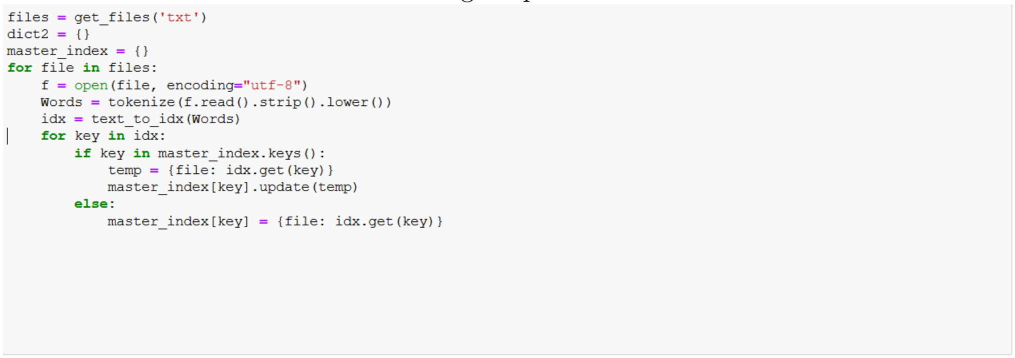
it returns the tokens/words in the text provided and also their positions in the text. The other function takes a list of words and returns a dictionary with the words as keys and the values are the positions in the text.

3 Read the content of a folder

Will not comment much on this section since we used a prewritten function to extract the files from a folder. I only did a small change to get the files from the current folder instead of a specific one.

4 Create a master index for all the files

To create the master index the following script was used:

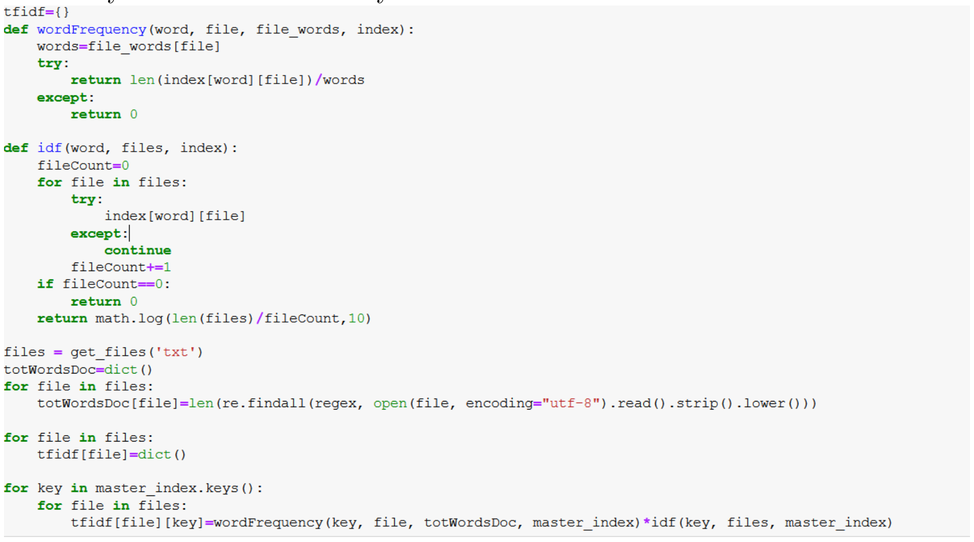


```
files = get_files('txt')
dict2 = {}
master_index = {}
for file in files:
    f = open(file, encoding="utf-8")
    Words = tokenize(f.read().strip().lower())
    idx = text_to_idx(Words)
    for key in idx:
        if key in master_index.keys():
            temp = {file: idx.get(key)}
            master_index[key].update(temp)
        else:
            master_index[key] = {file: idx.get(key)}
```

Where we use get files to get all the files with .txt as a suffix in the current folder. We use tokenize and text to idx as before to index each file. Then we build the master index with the help of dictionaries. The master index has the words as keys and as values a nested dictionary which has the documents as keys and positions of the words as values.

5 Use tfidf to represent the documents

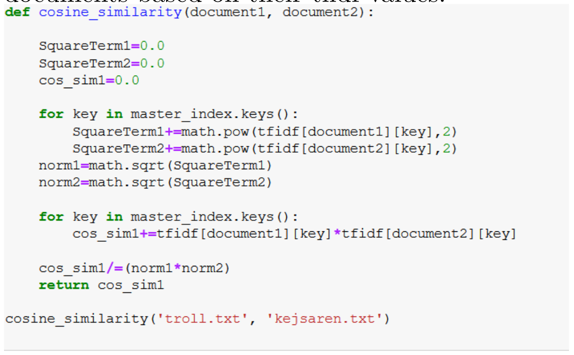
As the title of the section says we use the tfidf method to represent the documents. This is short for term frequency–inverse document frequency which is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. The following helper functions and the following script is how it was put together. Again by the use of dictionaries and the result is represented in a dictionary with the files as keys and a nested dictionary with the words as keys and the tfidf value for each word as values.



```
tfidf={}
def wordFrequency(word, file, file_words, index):
    words=file_words[file]
    try:
        return len(index[word][file])/words
    except:
        return 0

def idf(word, files, index):
    fileCount=0
    for file in files:
        try:
            index[word][file]
        except:
            continue
        fileCount+=1
    if fileCount==0:
        return 0
    return math.log(len(files)/fileCount,10)

files = get_files('txt')
totWordsDoc=dict()
for file in files:
    totWordsDoc[file]=len(re.findall(regex, open(file, encoding="utf-8").read().strip().lower()))

for file in files:
    tfidf[file]=dict()

for key in master_index.keys():
    for file in files:
        tfidf[file][key]=wordFrequency(key, file, totWordsDoc, master_index)*idf(key, files, master_index)
```

6 Compare the documents of a collection

Using the cosine simularity we compared all the documents and represented the result in a matrix. To do this the following function was used. It compares two documents based on their tfidf values.

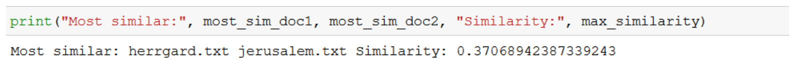


```
def cosine_similarity(document1, document2):

    SquareTerm1=0.0
    SquareTerm2=0.0
    cos_sim1=0.0

    for key in master_index.keys():
        SquareTerm1+=math.pow(tfidf[document1][key],2)
        SquareTerm2+=math.pow(tfidf[document2][key],2)
    norm1=math.sqrt(SquareTerm1)
    norm2=math.sqrt(SquareTerm2)

    for key in master_index.keys():
        cos_sim1+=tfidf[document1][key]*tfidf[document2][key]

    cos_sim1/=(norm1*norm2)
    return cos_sim1

cosine_similarity('troll.txt', 'kejsaren.txt')
```

Computing the similarity matrix the following result was obtained:

	bannlyst.txt	gosta.txt	herrgard.txt	jerusalem.txt	kejsaren.txt	marbacka.txt	nils.txt	osynliga.txt	troll.txt
bannlyst.txt	1.0	0.049	0.0009	0.0065	0.024	0.0368	0.051	0.0521	0.0886
gosta.txt	0.049	1.0	0.0031	0.0043	0.048	0.0802	0.1048	0.1248	0.1957
herrgard.txt	0.009	0.0031	1.0	0.3707	0.0007	0.0036	0.0051	0.0048	0.0041
jerusalem.txt	0.0065	0.0043	0.3707	1.0	0.0018	0.0049	0.0045	0.0283	0.0071
kejsaren.txt	0.024	0.048	0.0007	0.0018	1.0	0.0711	0.0497	0.0511	0.1813
marbacka.txt	0.0368	0.0802	0.0036	0.0049	0.0711	1.0	0.0847	0.0932	0.1472
nils.txt	0.051	0.1048	0.0051	0.0045	0.0497	0.0847	1.0	0.1106	0.1885
osynliga.txt	0.0521	0.1248	0.0048	0.0283	0.0511	0.0932	0.1106	1.0	0.1926
troll.txt	0.0886	0.1957	0.0041	0.0071	0.1813	0.1472	0.1885	0.1926	1.0

And printing the two most similar documents:



```python
print("Most similar:", most_sim_doc1, most_sim_doc2, "Similarity:", max_similarity)
```

```
Most similar: herrgard.txt jerusalem.txt Similarity: 0.37068942387339243
```

7 Comparing to google

Comparing our method to the text *Challenges in Building Large-Scale Information Retrieval Systems* which is about the history of google indexing.
As far as I can tell google use shards (that can be compared to our words in this assignment) to index. Since the volume of shards was increased so rapidly they improved their performance by improving the index encoding among other things. On slide *45* one can see similarities to our method in their index encoding. They search for a word and when they get a hit they store the position and some attributes. We did the same only that our attributes was the tfidf value of the word and in googles case they used stuff like font size and title as attributes.