Danmarks Tekniske Universitet

01415
Computational Tools for Big Data

Final Project

Anonymous

December 2016

1 Introduction

In this project, we will use the Project Gutenberg Corpus documents, which contains more than 150 GB(500 GB with audio and images) of uncleared data. We will process the text documents of this dataset to build a system that is able to estimate the metadata (topic, popularity, author...) of a given document. Also, we will implement an Information Retrieval System, in which, given a document as a query, the system will output a list of the most similar documents, this way, a person can input its favourite novel, and find more books of the same kind.

1.1 Structure of the report

This report is separated in sections which are describing the different parts of the system. In each section we are representing our ideas but we are not including complete code samples in the sections, just a simplified pseudocode functions which are more accessible for a reader. If the reader is in interested in complete source code he can find it in the appendices.

1.2 Development environment

We are working on a desktop computer which is now running as a server for the calculations. All work is done in a virtual container with access to 8GB RAM and quad core processor and the team members have access to the resources via ssh and rdp protocols.

2 Dataset

2.1 Data acquisition

We used for our project a dataset from Project Gutenberg. We downloaded the data by mirroring host server. [1] This can be achvied by running following command.

```
rsync -av --del ftp@ftp.ibiblio.org::gutenberg /gutenbergDataRsync
```

Size of the downloaded date is around 150 GB excluding an audio and images files. Because our aim for this project is to work with text files we are not using them in our analyse.

2.2 Data explanation

The downloaded data consist of ebooks files mostly in txt, pdf, html and epub file formats. The book files are mapped to unique *ID* which is connecting them with other information. For example the book *Frankenstein; Or, The Modern Prometheus* belongs to the id 84 and therefore there exists files 84.txt, 84.epub, 84.rdf and etc. We are using this ID to access multiple format for a single book and also for extracting meta-data information.

Metadata are stored in RDF format and contain most important facts about the book as author, title, downloads and topic, we are parsing those files to be able train our prediction.

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//
EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html lang="en" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Zero the Slaver: A Romance of Equatorial Africa</title>
<meta content="text/html; charset=utf-8" http-equiv="Content-Type"/>
<meta content="Lawrence Fletcher" name="Author"/>
<meta content="Zero the Slaver" name="Description"/>
<style type="text/css">
```

Listing 1: Selection from a XHTML book file

```xml
<pgterms:downloads rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">2611
</pgterms:downloads>
   <dcterms:hasFormat>
     <pgterms:file rdf:about="http://www.gutenberg.org/ebooks/10.txt.utf-8">
       <dcterms:format>
         <rdf:Description rdf:nodeID="Nbe07b875f95b483fbc4dead440533f41">
           <rdf:value rdf:datatype="http://purl.org/dc/terms/IMT">
           text/plain</rdf:value>
           <dcam:memberOf rdf:resource="http://purl.org/dc/terms/IMT"/>
         </rdf:Description>
       </dcterms:format>
```

Listing 2: Selection from a RDF metadata file

3 Data Preprocessing

3.1 Data standardization

As first step for data preparation we are converting different data formats to cleaned text. Even though the DDBB (corpus) of our system is based on the Gutenberg project database, we can process files from different kinds of sources, so that our document query can be a wide range of file type. We can process:

- Plain text files
- HTML documents stored in the computer DDBB
- URLs
- pdfs

Support for different format types can be easily added.

3.2 Text cleaning

Once the relevant plain text is obtained the NLTK 3.0 Python library is used to preprocess the data, performing a correction and normalization of the words and expressions which we will further be used for representing each document. As we will explain later, this preprocessing will be performed in parallel for each document for obvious reasons (the corpus does not fit into memory). This preprocessing stage consists of the following steps:

- Removing accents: In case there are special characters for some reason, we first remove them.
- Tokenization: It divides the whole plain text String into separate tokens, in this case String words.
- Lower Case: Since upper letters are not important for the purposes of the task, every letter is turned into lower case so that the IR system is not case sensitive.
- Removing Alpha-numeric words: It removes all the punctuation and words that are not a combination of letters and numbers.
- Removing stopwords: It removes common words such as articles or junctions that are used in language to glue subjects. These words are not very relevant to the IR system because they don't offer information about the context themselves. The NLTK corpus of English stopwords is used in order to identify them.
- Steaming: Normalize verbs, plurals and compositions so that they are treated as the same. The Python implementation of the Snowball steamers is used.

3.3 Metadata extraction

As we mentioned before the metadata are stored in a RDF format. We are loading the RDF file as XML document, defining namespace to be able to access elements and extracting the desired information from data. The implementation is described in following function.

```
import xml.etree.ElementTree as et #XML parser

def extract_metadata(filename):
    #load rdf file as tree
    tree = et.parse(filename)
    #namespace definition
    ns = {"dcterms": "http://purl.org/dc/terms/", "pgterms":
↪   "http://www.gutenberg.org/2009/pgterms/"}
    #extract data
    for entry in tree.findall(".//dcterms:subject/rdf:Description/rdf:value", ns):
        subjects.append(entry.text)
    return {'Subject':subjects}

#example call
id = '10015'
extract_metadata(id+'/pg'+id +'.rdf')
```

Listing 3: A simplified function to extract information from a RDF metadata file

4 Implementation

The following figure is showing how the implemented system works. As it can be observed, our system has an initial preprocessing step in which the documents of the system are treated in a parallel manner, using MRjob library. In this preprocessing, we extract the relevant information of the documents to have a standardized representation of them. When a query is given to the system, it is transformed to the same representation and compared in parallel with the documents in the dataset. We will go through each part separately in the subsections.

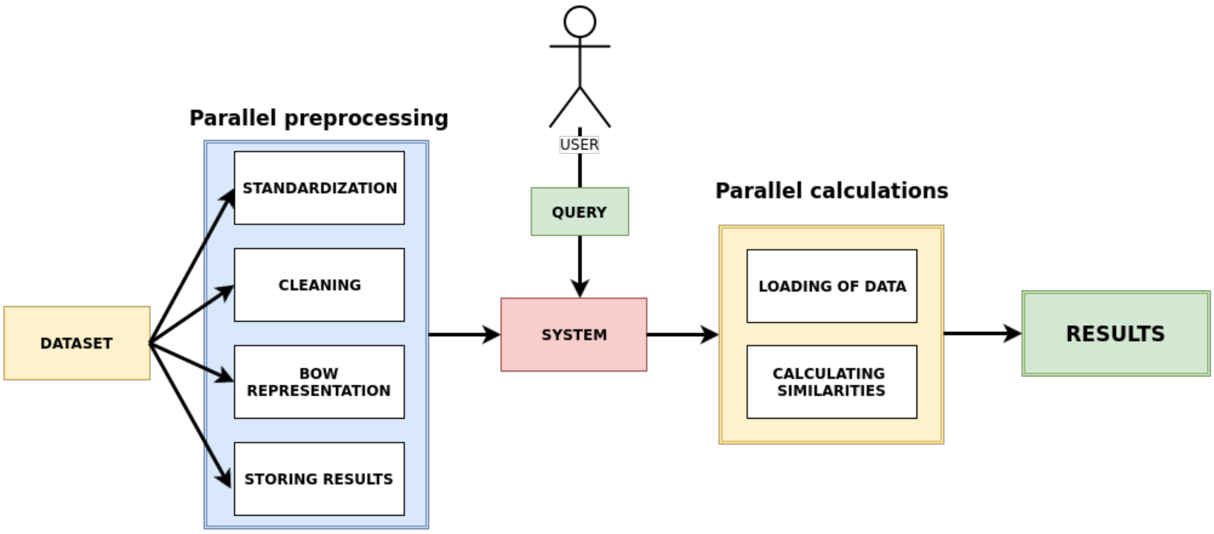

Figure 1: Diagram of the system

4.1 Document Representations

Every document is represented as the count of a set of normalized relevant words. This representation could be use for classification and regression techniques directly such as Naive Bayes or Random Forest in order to obtain patterns in documents. Furthermore we will transform each document into a TF-IDF vector that contains, for every different word in the set, a floating number obtained as the product of two terms:

- $TF_{(t,d)}$: Term Frecuency (BoW). Frecuency of term 't' in the document 'd'.
- $IDF_{(t,c)}$: Inverse Document Frecuency of term 't' in corpus 'c',

$$TF_{(t,d)} = \frac{\text{Ocurrences of term t in document d}}{\text{Number of total terms in d}}$$

$$IDF_{(t,c)} = log_2\left(\frac{\text{Total number of documents in c}}{\text{Number of documents that contain t}}\right)$$

The resulting TF-IDF vector representing each document in the corpus has the form:

$$TF - IDF_{(t,d,c)} = TF_{(t,d)} \times IDF_{(t,c)}$$

The IDF term is a measure of how discriminative a word is in the whole corpus and the TF term is a measure of how representative a word is for a document.

After testing a few queries, we have found that the $TF_{(t,d)}$ representation alone has a better retrieval of documents so it is the one we use in the final implementation.

The following figure shows the Bag of words for different files in a bar chart. We can appreciate that the most common enough could have enough representative power to characterize the documents to be able to draw similarities between them. Also, the next image, shows the Bag of Words of another document in a picture format to have a bigger overview of the weight of the words.

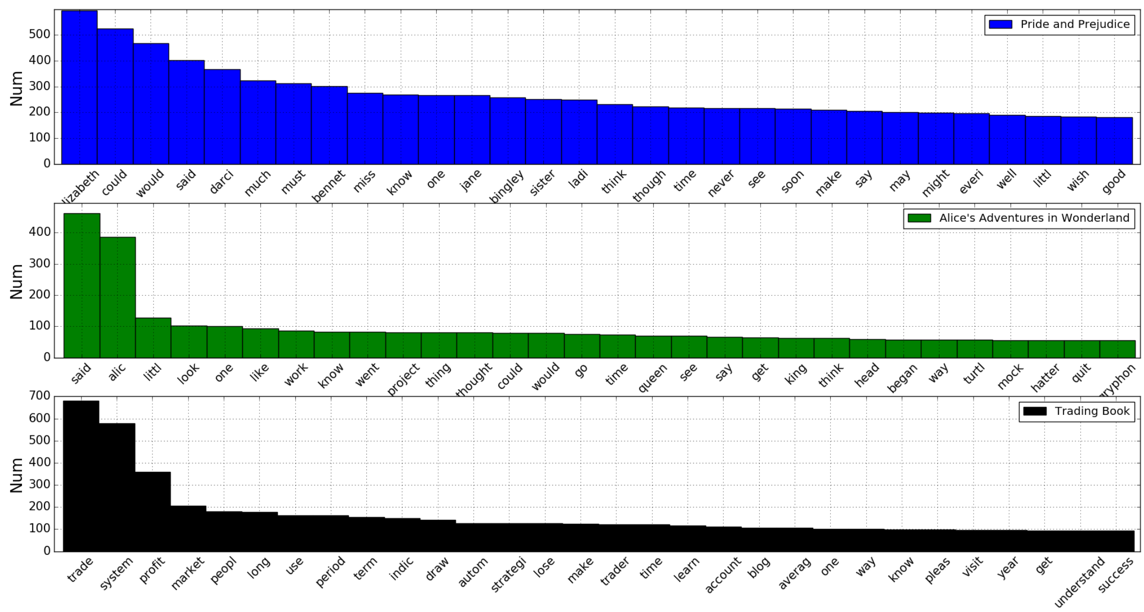

Figure 2: Bag of Words chart representation

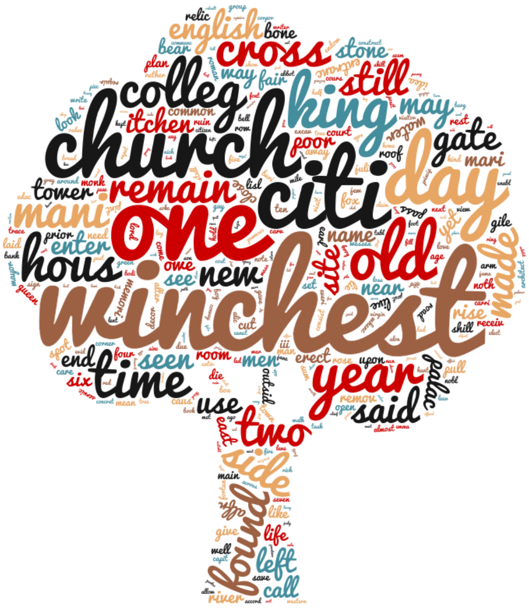

Figure 3: Bag of Words CloudWord representation

4.2 Similarity measures

The TF-IDF representation of the documents, allows them to be represented as vectors in N-dimensional space, where every dimension corresponds to a possible term. This representation can be used directly into a machine learning algorithm to find the relation between the vector and the objective function (popularity, author, downloads...). In the future work we will use them to try to estimate those objective functions.

Different similarity measures can be used in order to draw patterns in the text and be able to generate clusters that will help us find patterns and perform information retrieval system in which, given an example document, the system outputs a set of documents similar to the one presented. The Euclidean distance and cosine similarity measure have been used. The total rank has been obtained by mixing both ranks.

$$EuclidDist(Q, d) = |\vec{q} - \vec{d}|$$

$$CosSim(Q, d) = \frac{|\vec{q} - \vec{d}|}{|\vec{q}||\vec{d}|}$$

After a few trials of the system, the Euclidean Distance is the one that retrieves the most similar documents, it is the one finally implemented.

4.3 Scalability

Due to the high volume of data we are managing, it seemed logic to transform the documents into a simpler representation in a first stage of the process and store the results into disk. This way we will not need to recompute the preprocessing every time we try new algorithms, parameters or queries.

So, after preparing the data, being able to detect the text documents and obtain their metadata, the following step is to extract their Bag of Words representation, BoW, and store it into disk.
There are some challenges to face here:

- The documents are too big in order to make a BoW of the whole dataset at once.
- Every document will have a different dictionary for the BoW.

The solution we propose is to preprocess every file separately in parallel, using a **MRJob step** and store its BoW into disk, using a CSV file. In this way we have a sparse representation of all the files in the dataset into disk. Furthermore, we also limit the number of words in each dictionary to $N_w = 1000$.
In a later step, all the CSV files are read in parallel using another **MRJob step** whose combiners will merge the dictionaries of all the documents into a single one and finally transform the documents into

the common final sparse representation. Once this is done, we have a common sparse representation of all the documents, we use the sparse matrix library of python in order to make operations faster.

These stages are not so trivial and we use the **pandas** library in order to deal with CSVs and merge the dictionaries in the right way. Once we obtain the common words between the query and the documents of the dataset, we can compute the euclidean distance measure between it and the rest of the documents to obtain the documents. Then we find the top 5 files with the highest similarity measure and check its resemblance to the query. We need to do this ourselves

4.4 Running time

Running our system on data-set of this size was challenging. We tried to implemented it as parallel as possible. Time for preprocessing the data was based with linear dependence on number of records loaded. For 1000 records preprocessing is taking around 1 hour. Because in our dataset we have around 50000 books in different formats, creating the BOW representation for whole dataset is slow process. We can increase speed by using a cluster or by upgrading our hardware. This should be reasonable simple because all code is prepared to be run distributively. When the representation is done calling a query can took from 3-15 seconds based on the number of record preprocessed.

5 Testing queries

We have performed several tries with different queries in order to tune the parameters and models (similarity functions, words per document...). The performance of the system has been tested by us, giving a query to the system and checking if the retrieved documents are related. Taking into account both precision and recall. Of course the methodology can be improved in a lot of points but we managed to create this full first implementation.
As an example query we use the document Pride and Prejudice, the 3 most similar documents retrieved by the system, that are not from the same author are:

- The Emancipated by George Gissing
- Denzil Quarrier by George Gissing
- Demos by George Gissing

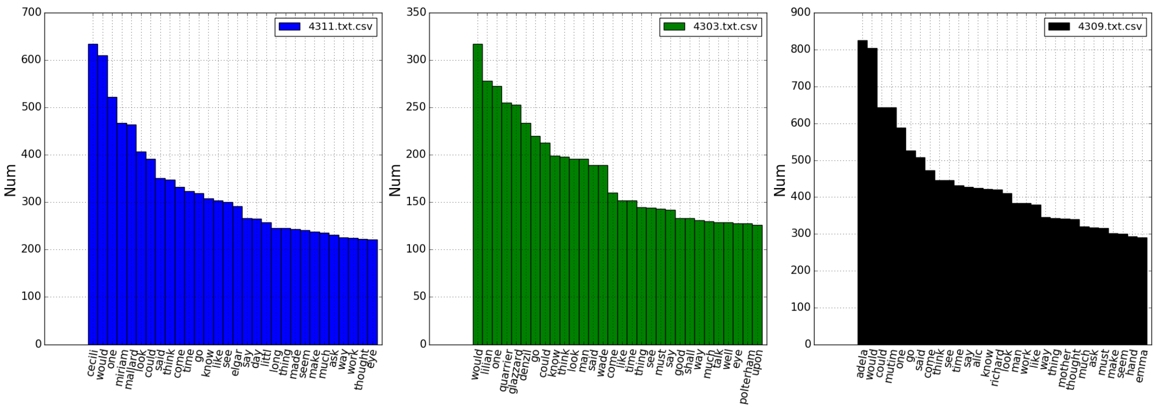

Figure 4: Bag of Words of the retrieved documents for Pride and Prejudice

All the retrived books are similar novels to the query one, but apparently the style of writting of the author has a lot of influence in this retrieval system. So it may be used for detecting plagiarism for example.

It seems like in order the improve the system, we should remove the common verbs and words that are non-informative. This is partially done by the IDF component, but in our dataset we found it more misleading than helping.

When queries not related to the dataset were given, such as a book about trading or a Python tutorial, the results of the retrieval system performed badly, because all the retrieved documents have very little to do with the query, but there is little we can do about it. Since the system is very scalable, there would be no problem, adding a lot of more documents into it. For example, the retrieved documents for the trading book are:

- The Black Experience in America by Norman Coombs
- West Indian Fables by James Anthony Froude Explained by J. J. Thomas
- Political Ideals by Bertrand Russell

The first two documents are actually about slavery, but since the words ”work”, ”black”, ”white”, ”time”, ”power” are very common, then the similarity is high (in trading the candlestick charts are common and they are in black and white, so it is a common word). The third document is actually about economy so it is a reasonable retrieval.

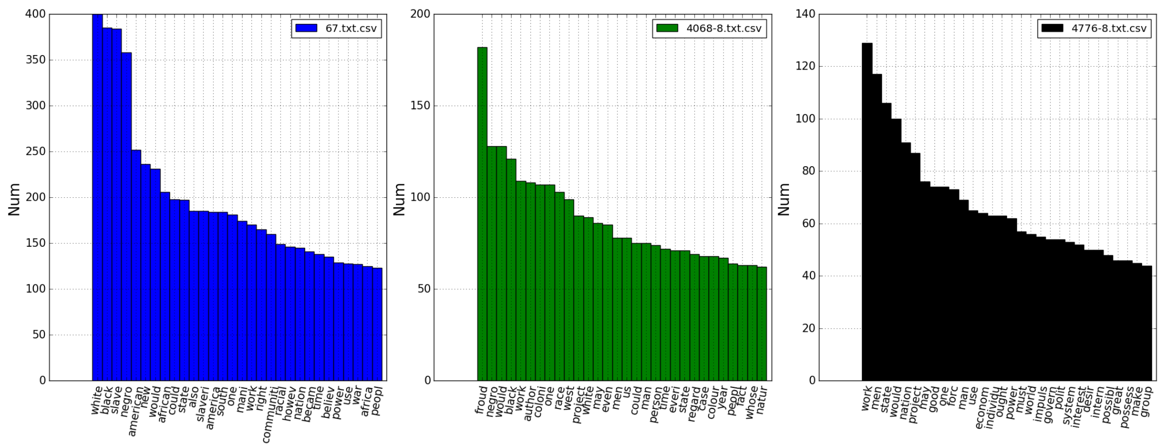

Figure 5: Bag of Words of the retrieved documents for a Trading Book

```
get_similar('Pride and Prejudice')
#Results:
['The Emancipated by George Gissing',
'Denzil Quarrier by George Gissing',
'Demos by George Gissing']
#Running time 6.25s
```

Listing 4: Pseudofunction queries

5.1 Topic detection and downloads prediction

We also tried to develop a system to predict the numbers of downloads and topic from the BOW representation. About the downloads, we tried basic regression algorithms such as linear regression but the results were just random. This could be expected because the number of downloads we have is only for the last month and it is not something that intuitively seem related to the frequency of certain words.

About topic detection we ran into the problem that the documents were categorized in slightly different manner and also, they had multiple tags, so it made it hard for us to model it and create a good system. We leave this feature as a possible future line of development.

6 Conclusion

During the development of this project we have seen the special challenges that Big Data presents. We solved the computational limitations by paralleling the processing of the data using the library MRjob and we pretransformed and permanently stored the dataset in order to not have to recompute it every time there is a new query.

We believe that the overall performance of the system is acceptable, the system has many limitations but we were able to tackle the problems and come up with a feasible and time efficient solution. More work needs to be done regarding the representation and similarity measures of the system but this could be the starting point of a high quality information retrieval system.

References

[1] "Gutenberg:mirroring_how-to," 2013 (accessed November 25, 2016). [Online]. Available: http://www.gutenberg.org/wiki/Gutenberg:Mirroring_How-To

Appendices

Code

.1 main.py

```
import numpy as np
import utilfunc as uf
import textprocesslib as tplib
import highfunc as hf
from subprocess import call  # Execute commands
import pandas as pd

from scipy.sparse import csr_matrix
import pickle_lib as pkl

from graph_lib import gl
#rootFolder = "./GutenbergData"
rootFolder = "./GutenbergData/A"
outFolder = "./books"
csvFolder = "./csvs"

## Read the first wrong library
read_preproc = 0

if (read_preproc == 1):
    # If we need to read and preprocess the raw files
    # and put then in the output folder

    hf.read_and_preprocess(rootFolder, outFolder)

compute_BoW = 0
if (compute_BoW == 1):
    ## Read the preprocessed data, preprocess the text and metadata
    # Creating a bag of word for every file and pickleing it
    # The files are already in the desired HTML format or any
    # other that we modify later

    use_MrJob = 1
    if(use_MrJob == 0):
        hf.compute_and_save_BoW(outFolder, csvFolder, MaxWords = 1000)
    ## USE MRJOB INSTEAD !!!

    elif (use_MrJob == 1):
        allPaths = uf.get_allPaths(outFolder)
        pd.DataFrame(allPaths).to_csv("./paths.txt")

        call("python ./precompute_DDBB_MrJob.py ./paths.txt", shell=True)

load_BoWs = 0
if (load_BoWs == 1):
    # Load all the BoWs and join them into a common format ?
    All_bow, dict_files = hf.load_and_join_BoWs(csvFolder, MaxFiles = 500)

######################################################################
################## Query Formatting ###########################
#################################################################
# Now we can enter a Query, then it will be transformed in the same way as the
# corpus and a similarity measure between it and all the documents will be done
# The system will output the N most similar elements.
```

```python
import copy
query_f = 1
if(query_f == 1):

    query_dirs = ["./books/0/Winchester","Yorkshire Battles", "1342-0.txt", "11-0.txt"]
    query_dir = query_dirs[2]
#    BoW = hf.preprocess_file("https://docs.python.org/2/howto/urllib2.html",
#                             typefile = "URL",MaxWords = 1000)

#    BoW = hf.preprocess_file("./free_ebook.pdf", typefile = "pdf",MaxWords = 1000)
#    BoW = hf.preprocess_file("1342-0.txt", typefile = "text",MaxWords = 1000)
    BoW = hf.preprocess_file("11-0.txt", typefile = "text",MaxWords = 1000)

    BoW = BoW.set_index(['word'])  # Set the index
    BoW.index = BoW.index.str.encode('utf-8') # Changing

    eu_dist = hf.get_doc_sims(All_bow, BoW)

    best_common,best_common_i  = uf.sort_and_get_order(eu_dist[:-1].tolist(), reverse = False)
#    print best_common_i[:5]
#    print dict_files
#    plt.close("all")
    gl.set_subplots(1,3)
    for i in range(3):
        BoW_values = All_bow[str(best_common_i[i])].values
        BoW_index = All_bow.index
        Nwords = 30
        b_v, b_vi = uf.sort_and_get_order(BoW_values, reverse = True)
        gl.bar(BoW_index[b_vi[0:Nwords]].tolist(),
               BoW_values[b_vi[0:Nwords]],
               legend = [dict_files[best_common_i[i]].split("/")[-1]],
                labels = ["", "", "Num"],
                nf = 1)

    ## Get the BoW of the most common:
plotting_thins = 0
if(plotting_thins == 1):
    query_dirs = ["./books/0/Winchester","Yorkshire Battles"]
    query_dirs = ["1342-0.txt", "11-0.txt"]
    labels = [ "Pride and Prejudice", "Alice's Adventures in Wonderland", "Trading Book"]

    gl.set_subplots(3,1)
    for i in range(3):


        # HTML  text
    #    BoW = hf.preprocess_file(query_dir, typefile = "text",MaxWords = 1000)
    #    BoW = hf.preprocess_file("https://docs.python.org/2/howto/urllib2.html",
   #     typefile = "URL",MaxWords = 1000)
        if(i < 2):
            query_dir = query_dirs[i]
            BoW = hf.preprocess_file(query_dir, typefile = "text",MaxWords = 1000)
        else:
            BoW = hf.preprocess_file("./free_ebook.pdf", typefile = "pdf",MaxWords = 1000)

        BoW = BoW.set_index(['word'])  # Set the index
        BoW.index = BoW.index.str.encode('utf-8') # Changing

#        BoW = hf.load_Bow()

        Nwords = 30
```

```
        BoW = BoW.sort(['num'], ascending=[0])
        print BoW.index[0:Nwords].shape
        print BoW["num"].values[1:Nwords]


        gl.bar(BoW.index[0:Nwords].tolist(), BoW["num"].values[0:Nwords],
               legend = [labels[i]],
                labels = ["", "", "Num"],
                nf = 1)

    caca  = open("./BoW.txt","w+")
    for i in range(len(BoW.index.tolist())):
        for j in range(BoW.iloc[i]["num"]):
            caca.write(BoW.iloc[i].name + " ")
    caca.close()

```

.2 precompute/$_{D}DBB_{M}rJob.py$

```
from mrjob.job import MRJob
from mrjob.step import MRStep
import utilfunc as uf
import highfunc as hf
from subprocess import call
import os


## This file is meant to be called from the main program
# in order to fucking ppreprocess all the DDBB obtaining their
## BoW and storing it into different files.

rootFolder = "./books"

class MRWordVC(MRJob):

    def mapper_VC(self, _, line):
        # This mapper obtains all the paths of all the files in the DDBB
        # and then fucking fucking yields a task for all of them.
        print os.getcwd()
        outFolder = "./csvs"
#         outFolder = "/home/montoya/Desktop/DTU Lec/1st Semester/5. Computational Tools for Big D

        filedir = line
        BoW = hf.preprocess_file(filedir, typefile = "HTML", MaxWords = 1000)
        file_name = filedir.split("/")[-1]
        filedir = filedir.split(file_name)[0]
```

.3 textprocesslib.py

```
import unicodedata
from nltk.tokenize import word_tokenize, wordpunct_tokenize, sent_tokenize
from nltk.stem import SnowballStemmer
from nltk.corpus import stopwords

import numpy as np
import pandas as pd
## TEXT PREPROCESSING FUNCTIONS

def doc_preprocess(document, mode = 0):
    # Preprocesees a document:  Tokenization, lemmatization...
    if (mode == 1):print document
```

```python
    document = strip_accents(document)
    if (mode == 1):print document
    document = doc_tokeniz(document,mode)
    if (mode == 1):print document
    document = doc_lowercase(document,mode)
    if (mode == 1):print document
    document = doc_rem_punctuation(document,mode)
    if (mode == 1):print document
    document = doc_rem_stopwords(document,mode)
    if (mode == 1):print document
    document = doc_stem(document,mode)
    if (mode == 1):print document
    return document

def strip_accents(s):
#   s = s.decode('utf-8').encode('ascii', 'replace')
#   s = s.encode('utf-8', 'replace')

   return ''.join(c for c in unicodedata.normalize('NFD', s)
                  if unicodedata.category(c) != 'Mn')

def doc_tokeniz(document, mode):
    tokens = word_tokenize(document)
    return tokens

def doc_lowercase (document, mode):
    low_text = [w.lower() for w in document]
    return low_text

def doc_rem_stopwords(document, mode):
    stopwords_en = stopwords.words('english')
    clean_text = [word for word in document if not word in stopwords_en]
    return clean_text

def doc_stem(document, mode):
    stemmer = SnowballStemmer('english')
    steammed_text = [stemmer.stem(word)for word in document]
    return steammed_text

def doc_rem_punctuation(document, mode):
    clean_text = [w for w in document if w.isalnum()]
    return clean_text

#####################################################
############ Bag of Words ##########################

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
# Initialize the "CountVectorizer" object, which is scikit-learn's
# bag of words tool.


def obtain_BoW(document, MaxWords = 1000):
    # Obtains the BoW and transforms it to a pandas Dataframe
    # Document is a list of words
    vectorizer = CountVectorizer(analyzer = "word",   \
                                 tokenizer = None,    \
                                 preprocessor = None, \
                                 stop_words = None,   \
                                 max_features = MaxWords)

```

```python
    BoWdoc = vectorizer.fit_transform(document)
#     print BoWdoc

    # BoW =  Pairs that tell you for each word,
    # the index file it belongs to
#    ## Create the dictionary with the words
#     vectorizer.fit(document)
#     # Transform the file to Sparse form matrix
#     BoWdoc = vectorizer.transform(document)

    # Get the bocabulary
    vocab = vectorizer.get_feature_names()
#     print vocab
    # Transform it to an array form
    BoWdoc_array = BoWdoc.toarray()
    # Sum up the counts of each vocabulary word
    count = np.sum(BoWdoc_array, axis=0)
#     print count
    BoW = dict()
    BoW["word"] = vocab
    BoW["num"] = count

    datFr = pd.DataFrame(BoW)

    return datFr

def obtain_Tfidf(document):
    # Document is a list of words
    vectorizer = TfidfVectorizer(analyzer = "word",    \
                                 tokenizer = None,     \
                                 preprocessor = None, \
                                 stop_words = None,    \
                                 max_features = 50,   \
                                 use_idf= False)

    ## Create the dictionary with the words
    vectorizer.fit(document)

    # Get the bocabulary
    vocab = vectorizer.get_feature_names()
    print vocab

    # Transform the file to Sparse form matrix
    BoWdoc = vectorizer.transform(document)
    # Pairs that tell you for each word, the index file it belongs to

    # Transform it to an array form
    BoWdoc_array = BoWdoc.toarray()
    # Sum up the counts of each vocabulary word
    dist = np.sum(BoWdoc_array, axis=0)

    print BoWdoc
    print dist
```

.4 utilfunc.py

```python
import os
import magic
import shutil
from bs4 import BeautifulSoup  # For HTML web treatment.
import json
```

```python
import numpy as np
################################################
############### Files library ################
################################################

def create_folder_if_needed (folder):
    # This function creates a path if it does not exist
    if not os.path.exists(folder):
        os.makedirs(folder)

def get_allPaths(rootFolder, fullpath = "yes"):
    ## This function finds all the files in a folder
    ## and its subfolders

    allPaths = []

    for dirName, subdirList, fileList in os.walk(rootFolder):  # FOR EVERY DOCUMENT
#       print "dirName"
       for fname in fileList:
            # Read the file
            path = dirName + '/' + fname;
            if (fullpath == "yes"):
                allPaths.append(os.path.abspath(path))
            else:
                allPaths.append(path)

    return allPaths

def type_file(filedir):
    mime = magic.Magic()
    filetype = mime.id_filename(filedir)
#    filetype = mime.id_filename(filedir, mime=True)

    # This will be of the kind "image/jpeg" so "type/format"
    filetype = filetype.split(",")[0]
    return filetype

def copy_file(file_source, file_destination, new_name = ""):
    # Copies a file into a new destination.
    # If a name is given, it changes its name

    file_name = ""
    file_path = ""

    file_name = file_source.split("/")[-1]
    file_path = file_source.split("/")[0]

    if (len(new_name) == 0): # No new name specified
        file_name = file_source.split("/")[-1]
    else:
        file_name = new_name

    create_folder_if_needed(file_destination)

    shutil.copy2(file_source, file_destination + "/" + file_name)

def loadJsonFromFile(filename):
    try:
        with open(filename) as data_file:
            jsonData = json.load(data_file)
        # return created json object
```

```
        return jsonData

    except IOError:
        print "Error: File does not appear to exist."
        return 0

###############################################
############### HTML treating ###############
###############################################

### AUTHOR AND TITLE
#<div class="dochead">
#<h2 class="author">Lawrence Fletcher</h2>
#<h2 class="title">"Zero the Slaver"</h2>
#<hr/>
#</div>

### CHAPTER ARE LIKE
#<div class="bodytext">
#<a href="" name="chap02"></a>
#<h3>Chapter Two.</h3>
#<h4>A Night of Horror.</h4>


def check_cover(filedir):
    # This function checks if the file has the words cover
    # If it does, it is just the data of it,
    # I couldnt find a proper way to separate them really
    # They do not follow an easy format

    filename = filedir.split("/")[-1]

    if (filename.find("(cover)") != -1):
        return 1
    else:
        return 0

def process_HTML_doc(filedir):
    ## Read the file with HTML and close it
    fd = open(filedir, 'r')
    doc_HTML = fd.read()
    fd.close()
    # Use BeutifulSoup to process the HTML, get tittle, plain text...
    soup = BeautifulSoup(doc_HTML)  # Transform plain text HTML into soup structure

    ## Estructure of the HTMLS !!

    ## First check that it is a file of a document
    # <meta content="Zero the Slaver" name="Description"/>
    # <meta name="description" content="Project Gutenberg Ebooks." />
#    content_type = soup.find("meta", {"name": "Description"})
#    if(content_type == None):
#        content_type = soup.find("meta", {"name": "description"})
#
#    if(content_type == None):
#        return None
#    content_type = content_type["content"]
#    if (content_type == "Project Gutenberg Ebooks."):
#        return None

    useful_text= ""
```

```
    #### FIND ALL CHAPTER ####

     useful = soup.findAll("p")  # Directori (Dessert > Bannaan )
     for elem in useful:
         useful_text += " " + elem.text
#         keywords += " " + elem.text

#    useful = soup.findAll("div", {"class": "bodytext"})  # Directori (Dessert > Bannaan )
#    for elem in useful:
#        useful_text += " " + elem.text
##        keywords += " " + elem.text


     return useful_text

def sort_and_get_order (x, reverse = True ):
    # Sorts x in increasing order and also returns the ordered index
    x = np.array(x)
    x = x.flatten()  # Just in case we are given a matrix vector.
    order = range(len(x))

    if (reverse == True):
        x = -x

    x_ordered, order = zip(*sorted(zip(x, order)))

    if (reverse == True):
        x_ordered = -np.array(x_ordered)

    return np.array(x_ordered), np.array(order)

```

.5 highfunc.py

```
import os
import magic
import shutil
from bs4 import BeautifulSoup  # For HTML web treatment.
import json
import utilfunc as uf
import textprocesslib as tplib
import pickle_lib as pkl
import pandas as pd
import urllib2
from pyPdf import PdfFileWriter, PdfFileReader
from textract import process
def getURLContent(path = 'http://python.org/'):
    response = urllib2.urlopen(path)
    html = response.read()
    soup = BeautifulSoup(html, 'html.parser')
    return soup.get_text()

def getPDFContent(path = "../../../P3.pdf"):
#    content = ""
#    # Load PDF into pyPDF
#    input_file = file(path, "rb")
#    pdf = PdfFileReader(input_file)
#    # Iterate pages
#    for i in range(0, pdf.getNumPages()):
#        # Extract text from page and add to content
#        content += pdf.getPage(i).extractText() + "\n"
```

```python
#        print pdf.getPage(i).extractText()
#    # Collapse whitespace
#    content = " ".join(content.replace(u"\xa0", " ").strip().split())
#    input_file.close()
##    print content
    content = process(path)
    return content

def read_and_preprocess(rootFolder, outFolder, MaxFiles = 1000):
    # This function reads the documents and preprocess them
    # It only selects the valid files and extracts the
    # important metainformation and text.
    # It also renames them and puts them in other folders.

    # Max   MaxFiles files per forlder
    i = 0
    aux_f = 0

    ## Get all the paths of the folder
    allPaths = uf.get_allPaths(rootFolder)

    # For every path  TODO in map-reduce
    for filedir in allPaths:
        filetype = uf.type_file(filedir)

        print filedir
        print filetype
        # For all files, we check the extension
        if (filetype == "HTML document"):
            # Copy the file
            coverf = uf.check_cover(filedir)
            if (coverf == 1):
            #Copy the file in the new destination
                uf.copy_file(filedir,outFolder+"_cover" + "/" + str(aux_f))
            else:
                uf.copy_file(filedir,outFolder + "/" + str(aux_f))

            i = i + 1

            if(i >= MaxFiles):
                i = 0
                aux_f += 1

def preprocess_file(filedir, typefile = "text", MaxWords = 1000):
    # This function will preprocess one query, obtaining its BoW
    # text:     If the file is a text file in the system.
    # HTML:     If the files an HTML document in your computer
    # URL:      If the file is an URL
    # pdf:      If the file is a pdf

    ########## READ THE PROPER TEXT OF THE DOCUMENT ########
    if (typefile == "text"):
        caca = open(filedir)
        document = caca.read()
        document = document.decode('utf-8', errors='replace').encode('utf-8')
        document = unicode(document, "utf-8")
        caca.close()

    elif (typefile == "HTML"):
        document = uf.process_HTML_doc(filedir)
        print filedir
```

```
        print len(document)
        ## If the HTML file is not a book
        if (type(document) == type(None)):
            return pd.Dataframe([])  # Exit this loop execution
        if(len(document) == 0):
            return pd.Dataframe([])

    elif (typefile == "URL"):
        document = getURLContent(filedir)
#         document = unicode(document, "utf-8")

    elif (typefile == "pdf"):
        document = getPDFContent(filedir)
        document = unicode(document, "utf-8")
#     print document
    ########## Preprocess document and obtain BoW ########
    processed = tplib.doc_preprocess(document)
    BoW = tplib.obtain_BoW(processed, MaxWords)
    return BoW

def compute_and_save_BoW(rootFolder, outFolder, MaxFiles = 1000, MaxWords = 1000):
    ## Get all the paths of the folder
    allPaths = uf.get_allPaths(rootFolder)
    # For every path  TODO in map-reduce
    for filedir in allPaths:
        BoW = preprocess_file(filedir, typefile = "text", MaxWords = 1000)
#         print ' '.join(processed)
#         BoW = tplib.obtain_BoW([' '.join(processed[1:1000]),
#         ' '.join(processed[1000:2000])], MaxWords)
#         BoW = tplib.obtain_Tfidf([processed[:1000],processed[1000:2000]])
        ## Now we pickel the processed data into the machine disk

        file_name = filedir.split("/")[-1]
        uf.create_folder_if_needed(outFolder)
#         print BoW
#         pkl.store_pickle(outFolder + "/" + file_name + ".pkl",BoW,1)
        BoW.to_csv(outFolder + "/" + file_name + ".csv",
                   encoding = "utf-8",
                   index = False)  # Do not write the index number

def load_Bow(filedir):
        BoW = pd.DataFrame.from_csv(filedir, sep=',',
                                    index_col=1)
        return BoW

def load_and_join_BoWs(rootFolder, MaxFiles = 100):
    ## Get all the paths of the folder
    allPaths = uf.get_allPaths(rootFolder)

    # For every path  TODO in map-reduce
    Total_BoW = pd.DataFrame()

    index_doc = 0

    # Dictionary between the file and its number
    dict_files = {}
    for filedir in allPaths:
        # Load the BoW
        BoW = pd.DataFrame.from_csv(filedir, sep=',',
                                    index_col=1)
        # filedir.split("/")[-1]
```

```
        BoW = BoW.rename(columns = {'num':str(index_doc)})
        dict_files[index_doc] = filedir

        # Concatenate them !
        Total_BoW = pd.concat([Total_BoW, BoW], axis=1)
#        print BoW
#        print Total_BoW
        index_doc += 1
#        print dict_files
        if (index_doc >= MaxFiles): # If we are done loading
            # Fill the NaNs with 0
            Total_BoW = Total_BoW.fillna(0)
#            print Total_BoW.shape
            return Total_BoW, dict_files  # Return the total BoW
        Total_BoW = Total_BoW.fillna(0)

    return Total_BoW, dict_files

from scipy.sparse import csr_matrix
import numpy as np
def get_doc_sims(All_bow, BoW):
    # This function gives back the similarity measures of the documents,
    # compared to the database.
#    print BoW
    joined_inner = pd.concat([All_bow, BoW], axis=1, join='inner')
#    print joined_inner.shape
    A = csr_matrix(joined_inner)

    v = np.array(joined_inner["num"].values)
#    print v.shape
    v = v/float(v.sum())
#    print v.shape
    A = A.T

#    print A.sum(axis = 1).shape
    A = A/A.sum(axis = 1)  # Get the frequencies

    A = A - v
    A = np.multiply(A,A)
#    print v.shape
    eu_dist = A.dot(np.ones((v.size,1)))

    return eu_dist
```

.6 pickle_lib.py

```
import pickle
import gc
import os

# Library for  loading and storing big amounts of data into
# different files because pickle takes a lot of RAM otherwise.
# If the number of partitions = 1, then it just loads like a
# regular pickle file.
# It uses gc also to remove garbage variables.

def store_pickle (filename, li, partitions = 1, verbose = 1):
    gc.collect()
    splitted = filename.split(".")
    if (len(splitted) == 1):  # If there was no extension
        fname = filename
```

```
        fext = ""
    else:
        fname = '.'.join(splitted[:-1])   # Name of the file
        fext = "." + splitted[-1]     # Extension of the file

    # li: List of variables to save.
    # It saves the variables of the list in "partitions" files.
    # This function stores the list li into a number of files equal to "partitions" in pickle form
    # If "partitions" = 1 then it is a regular load and store
    num = int(len(li)/partitions);

    if (partitions == 1):  # Only 1 partition
        if (verbose == 1):
            print "Creating file: " + fname + fext
        with open(fname + fext, 'wb') as f:
            pickle.dump(li, f)
    else:                   # Several partitions
        for i in range(partitions - 1):
            if (verbose == 1):
                print "Creating file: " + fname + str(i) + fext
            with open(fname + str(i)+ fext, 'wb') as f:
                pickle.dump(li[i*num:(i+1)*num], f)
                # We dump only a subset of the list
        # Last partition to create
        if (verbose == 1):
            print "Creating file: " + fname + str(partitions -1) + fext

        with open(filename + str(partitions - 1)+ fext, 'wb') as f:
                pickle.dump(li[num*(partitions - 1):], f)
                # We dump the last subset.
    gc.collect()

def load_pickle (filename, partitions = 1, verbose = 0):

    gc.collect()
    total_list = []
    splitted = filename.split(".")
    if (len(splitted) == 1):  # If there was no extension
        fname = filename
        fext = ""
    else:
        fname = '.'.join(splitted[:-1])   # Name of the file
        fext = "." + splitted[-1]     # Extension of the file

    if (partitions == 1):  # Only 1 partition
        if (verbose == 1):
            print "Loading file: " + fname + fext

        if (os.path.exists(fname + fext) == True):   # Check if file exists !!

            with open(fname + fext, 'rb') as f:
                total_list = pickle.load(f)    # We read the pickle file
        else:
            print "File does not exist: " + fname + fext
            return []
    else:                           # Several partitions
        for i in range(partitions):
            if (verbose == 1):
                print "Loading file: " + fname + str(i)+ fext

            if (os.path.exists(fname + str(i)+ fext) == True):   # Check if file exists !!
```

```
                with open(fname + str(i)+ fext, 'rb') as f:
                    part = pickle.load(f)     # We read the pickle file
                total_list.extend(part)

            else:
                print "File does not exist: " + fname + str(i)+ fext
                return []

    gc.collect()
    return total_list

#n = 3
#lista = [10, 23, 43, 65, 34, 98, 90, 84, 98]
#store_pickle("lista",lista,n)
#lista2 = load_pickle("lista",n)
```

.7 simlib.py

```
#### SIMILARITY LIBRARY #####

import numpy as np
from utilfunc import *
from textprocesslib import *
# Define similarity values between the query and the corpus.
def get_similarities(query,database):
    query_words = doc_preprocess(query,1)  # Preprocess query
    query_words_tfidf = tf_idf_doc(query_words, database) # Get tfidf of query

    for i in range (len(query_words_tfidf[0])):
        print str(query_words_tfidf[1][i]) + "\t\t" + query_words_tfidf[0][i]
    print " "
    N_docs_database = len(database)

    eu_dists = np.zeros([N_docs_database,1])
    cos_sims = np.zeros([N_docs_database,1])
    n_commun = np.zeros([N_docs_database,1])
    tfidf_sum = np.zeros([N_docs_database,1])

    for i in range (N_docs_database):
        # Common words
        (vq ,vd) = get_common_tfidf_vectors(query_words_tfidf,database[i])
        # Non-common words
        (vnq ,vnd) = get_noncommon_tfidf_vectors(query_words_tfidf,database[i])

        # If no words in commun
        if(vq.size == 0):
             eu_dists[i] = -1
             cos_sims[i] = -1
             n_commun[i] = vq.size
        else:
            eu_dists[i] = get_euc_dist(vq,vd,vnq,vnd)
            cos_sims[i] = get_cos_sim(vq,vd)
            n_commun[i] = vq.size
            tfidf_sum[i] = np.sum(vd)

    similarities = (eu_dists,cos_sims, n_commun, tfidf_sum)
    # Similatiries obtained are:
    # - The euclidean distance
    # - The cosine similarity
    # - The number of words in commun.
```

```
    #      (Important coz, they can have very high values in the eu y cos
    #      but just because they have very few words in commun)
    # - tfidf_sum: Sum of the tdifd of the values

    return similarities

def rank_documents(Similarity, N_top, type_simi):
    # RANK WITH EVERY INDEPENDENT SIMILARITY MEASURE.
    n_doc = len(Similarity)
    # 1) Euclidean distance: The closer, the better, so the smaller the value, the better
    if (type_simi == "euclidean"):
        # For this similarity, the vectors that did not have words in common have
        # similarity -1, so we cannot order them directly like this.
        for i in range(len(Similarity)):
            if (Similarity[i] == -1):
                Similarity[i] = 10000000000;
        indexes = np.array(range(0,n_doc))
        together = zip(Similarity, indexes)  # Zip both arrays together
        sorted_together =  sorted(together) # Sorted Increasing order
        ordered_indexes = [x[1] for x in sorted_together]
        # Since ordered_indexes contains the indexes of the best documents
        # (obtained from decreasing similarity)

    # 2) Cosine distance: The biger the value, the better
    if (type_simi == "cosine"):
        # Wrong vectors have similarity -1 so no problem, the bigger the better.
        indexes = np.array(range(0,n_doc))
        together = zip(Similarity, indexes)  # Zip both arrays together
        sorted_together =  sorted(together, reverse=True) # Sorted Decreasing order
        ordered_indexes = [x[1] for x in sorted_together]
        #print ordered_indexes

    # 3) Use the number of words in common:
    if (type_simi == "common"):
        indexes = np.array(range(0,n_doc))
        together = zip(Similarity, indexes)  # Zip both arrays together
        sorted_together =  sorted(together, reverse=True) # Sorted Decreasing order
        ordered_indexes = [x[1] for x in sorted_together]

    # ) Use the sum of tfidf values:
    if (type_simi == "tdidf_sum"):
        indexes = np.array(range(0,n_doc))
        together = zip(Similarity, indexes)  # Zip both arrays together
        sorted_together =  sorted(together, reverse=True) # Sorted Decreasing order
        ordered_indexes = [x[1] for x in sorted_together]

    top_indexes = ordered_indexes[0:N_top]
    return top_indexes

def get_combined_rank(Ranks, N_top, type_comb):
# Ok... we have several similarity measures, now we have to output the ranking
# based on those similarities. How do we combine them to get the best ranking ?
    n_rank,n_doc = np.shape(Ranks)

    eu_w = 0.5; # Weight of the euclidean distance importance
    cos_w = 0.5; # Weight of the cosine distance importance
    n_w = 1;   # Weight of the nomber of common words importance

    total_similarity  = np.zeros([n_doc,1])

    for i in range(n_doc):
```

```
        total_similarity [Ranks[i]] += eu_w * Ranks[0][i]
        total_similarity [Ranks[i]] += cos_w * Ranks[1][i]
        total_similarity [Ranks[i]] -= n_w * Ranks[2][i]

    # Get the top indexes rank from the combined similarity methods:
    indexes = np.array(range(0,n_doc))
    together = zip(total_similarity, indexes)  # Zip both arrays together
    sorted_together =  sorted(together) # Sorted Increasing order
    ordered_indexes = [x[1] for x in sorted_together]
    top_indexes = ordered_indexes[0:N_top]
    return top_indexes

def get_euc_dist(vq,vd,vnq,vnd):
    distance = np.sqrt(((vq - vd)*(vq - vd)).sum(axis=0))


    return distance

def get_cos_sim(vq,vd):
    modules = (np.sqrt((vq*vq).sum(axis=0)) * np.sqrt((vd*vd).sum(axis=0)))

    if (modules == 0):
        print "Error, cosine similarity. One vector is all 0s."
        return -1
    cos_sim = np.dot(vq.transpose(),vd)
    cos_sim = cos_sim / modules

    return cos_sim
```