Assignment 2

Information Retrieval and Web Search
Winter 2024
Total points: 90
Issued: 01/31/2024 Due: 02/15/2024

All the code has to be your own (exceptions to this rule are specifically noted below). The code must run on the CAEN environment without additional installation or additional files (except for the data files specified in the assignment).

You can discuss the assignment with others, but the code is to be written individually. You are to abide by the University of Michigan/Engineering honor code; violations will be reported to the Honor Council.

General submission instructions:

- Include all the files for this assignment in a folder called *[your-uniqname].Assignment2/*
  Do not include the data folder, i.e., *cranfieldDocs/* (you can assume that this folder will be available in the same folder as your vectorspace.py program), cranfield.reljudge or cranfield.queries.
  Make sure to include an .output file for each of the two weighting schemes.
  For instance, mihalcea.Assignment2/ will contain vectorspace.py, two cranfield.*.*.output files, cranfield.answers
  You may include Assignment 1 files (e.g. preprocess.py) or external code you need for preprocessing (e.g., stemmer) as needed.
  Archive the folder using tgz or zip and submit on Canvas by the due date.
- Make sure you include your name and uniqname in each program and in the cranfield.answers file
- Make sure all your programs run correctly on the CAEN machines using python3.

[90 points] Vector-space model.

Write a Python program that implements the vector-space model. You will test this program on the Cranfield dataset, which is a small standard Information Retrieval text collection, consisting of 1400 documents from the aerodynamics field. The dataset cranfield.zip is available from the Files/ section under Canvas. The dataset is associated with 225 queries, along with human relevance judgments. These are also provided on Canvas as two files, cranfield.queries and cranfield.reljudge.

Note on relevance judgments: They are provided as a list of document numbers associated with queries. For instance, the following lines in the cranfield.reljudge file

```
4 236
4 166
5 552
5 401
5 1297
5 1296
```

indicate that for query 4, documents 236 and 166 are relevant; for query 5, documents 552, 401, 1297, and 1296 are relevant.

Programming guidelines:

Write a program called *vectorspace.py* that indexes the collection and returns a ranked list of documents for each query in a list of queries. The program will receive (at least) four arguments on the command line, in the following order. Other arguments can be added, if necessary.

- Argument 1: indicating the weighting schemes to be used for the **documents**. For the purpose of this assignment, your program should be able to "understand" at least two values for the term weighting schemes specified in this argument:
    - tfc (corresponding to the traditional tf.idf for documents)
    - <your-own-weighting scheme> (different from tfc)
- Argument 2: indicating the weighting schemes to be used for the **query**. For the purpose of this assignment, your program should be able to "understand" at least two values for the term weighting schemes specified in this argument:
    - tfx (corresponding to the traditional tf.idf for queries)
    - <your-own-weighting-scheme> (different from tfx).
- Argument 3: indicating the name of the folder containing the collection of documents to be indexed. For testing purposes, use the *cranfieldDocs/* folder that you will obtain after unpacking the cranfield.zip archive.
- Argument 4: indicating the name of the file with the test queries. For testing purposes, use the *cranfield.queries* query file that you will get from the class webpage.

Include the following two functions in *vectorspace.py*. For preprocessing in both functions, you are encouraged (but not required) to use the functions you implemented for Assignment 1. (Should you decide to use your functions from Assignment 1, please include the relevant files in your submission):

a. Function that adds a document to the inverted index:

Name: *indexDocument*; input: document id or tuple consisting of (document id, the content of the document); input: weighting scheme for documents (string); input: weighting scheme for query (string); input/output: inverted index (your choice of data structure)

Given the name of a file, this function will:

- preprocess the content provided as input, i.e., apply removeSGML, tokenizeText, stemWords.
  Notes: you can use the removeSGML and tokenizeText from Assignment 1, or you can replace them with external tools available on CAEN, if you prefer (eg, the numpy tokenizer available on CAEN).
  Do not apply the BPE tokenizer.
  For stemming words, you can use an existing implementation of the Porter stemmer, e.g., https://tartarus.org/martin/PorterStemmer/python.txt
    - If you use this code, you may need to make the following edit to the file to run it with Python3. Change on line 366 "print output" to "print(output)"
- add the tokens to the inverted index provided as input and compile the counts necessary to calculate the term weights for the given weighting schemes. Note: the inverted index will be updated with the tokens from the document being processed.

b. Function that retrieves information from the index for a given query.

Name: *retrieveDocuments*; input: query (string); input: inverted index (your choice of data structure); input: weighting scheme for documents (string); input: weighting scheme for query (input); output: ids for relevant documents, along with similarity scores (dictionary)

Given a query and an inverted index, this function will:

- preprocess the query, i.e., remove SGML, tokenize the text (no BPE), stem the words
- determine the set of documents from the inverted index that include at least one token from the query.
- calculate the similarity between the query and each of the documents in this set, using the given weighting schemes to calculate the document and the query term weights

The main program should perform the following sequence of steps:
i. open the folder containing the data collection, provided as the third argument on the command line (e.g., *cranfieldDocs/),* and read one file at a time from this folder.
ii. for each file, obtain the content of the file, and add it to the index with *indexDocument*
iii. if necessary for the term weighting schemes, calculate and store the length of each document
iv. open the file with queries, provided as the fourth argument on the command line (e.g., cranfield.queries), and read one query at a time from this file (each line is a query)
v. for each query, find the list of documents that are relevant, along with their similarity scores.

The *vectorspace.py* program should be run using a command like this:
% *python3 vectorspace.py tfc tfx cranfieldDocs/ cranfield.queries*

It should produce an output file called
cranfield.[DocumentWeightingNameHere].[QueryWeightingNameHere].output
consisting of pairs of query ids, along with the ids of the documents that are relevant and their similarity score (for each query, list in descending order of similarity score):

```
queryId1 documentId1 similarityScore1
queryId1 documentId2 similarityScore2
....
queryId1 documentIdx similarityScorex
queryId2 documentId1 similarityScore1
....
....
```

For instance, when you use tfc for the documents, and tfx for the queries, your output file will be called cranfield.tfc.tfx.output, and will include lines such as

```
1  4 0.23
1 123 0.16
...
```

Notes:

- The terminology used for the weighting schemes is based on the Salton & Buckley notation, discussed in class under the 'Term Weighting Approaches" section. This includes different variations for: term frequency; inverse document frequency; cosine normalization with document length.
- For your own weighting scheme, the weighting scheme for documents and queries can be any combination as long as they are different from tfc.tfx (corresponding to the traditional tf.idf) scheme. You are allowed to use the weighting schemes presented in lecture.
- As mentioned previously, it is acceptable to use an SGML tag remover or a tokenizer different from your own implementation in Assignment 1 (e.g., the numpy tokenizer available on CAEN).
- For stemming words, you can use an existing implementation of the Porter stemmer, e.g., https://tartarus.org/martin/PorterStemmer/python.txt Make sure you include any external code you use with your submission, so your assignment can be graded.
- If a term appears in the query but does not appear in any document, which leads to a document frequency of zero, you can choose to ignore it or do smoothing (e.g., add one to the df value).
- For the definition of precision, recall, and macro-averaging, please refer to lecture slides.
- The number of retrieved documents in part(b) will be at the scale of several hundreds.
- If you are using os.listdir, the implementation may differ on different operating systems (this means Windows and CAEN may return different lists). To have consistent results on both environments, you may need to sort the filenames.
- Make sure your code runs in Python 3.6 (this is the Python version on CAEN!)

Write-up guidelines:

Use the cranfield.reljudge file as a gold standard to measure the performance of your system. Use the following two weighting scheme combinations: (1) tfc for documents, tfx for query; (2) your choice of weighting scheme for documents and your choice of weighting scheme for query.

Create a file called cranfield.answers. Include in cranfield.answers:

* A description of the weighting schemes you used. Include a brief statement on how you came up with the second weighting scheme, and why you thought it might be effective.

* For each of the two weighting scheme combinations, calculate and include the macro-averaged precision and recall when you use:
- top 10 documents in the ranking (for each query)
- top 50 documents in the ranking (for each query)
- top 100 documents in the ranking (for each query)
- top 500 documents in the ranking (for each query)
If there are not enough documents for "top N", then pad with documents that have 0 similarity with the query, randomly selected without repeats.

* Which weighting scheme provides better results? Include a one paragraph discussion of how the two weighting schemes compare to one another.

* If your program receives more than the four specified arguments from the command line, please provide documentation on how to properly run your code.