Information Retrieval: Simple Search Engine

The primary purpose of this assignment is to get familiar with the basic term generation, indexing, and query processing algorithms and use them to implement a simple search engine.

You will build your search engine with the given skeleton code in Python and test on the Cranfield Dataset. You can work individually or in a two-person team.

Setup:

- You will need to use the NLTK toolkit for stemming. For Ubuntu, it is easy to install
  ```
  sudo pip install -U nltk
  ```
  Install the python pip package first if you have not yet. For other platforms, please check the web site for instructions.
- Download the Cranfield dataset. It contains three files: cran.all - the document collection, query.text - the sample queries, qrels.text - the relevant query-document pairs. Check README.txt for more details.
- Download the skeleton code. You should use the skeleton code for easier grading.

Part 1: Building Inverted Index

The preprocessing steps include

- split a document to a list of tokens and lowercase the tokens.
- remove the stopwords. A list of stopwords has been provided - check the file "stopwords" in the directory.
- stemming. Use a stemmer in NLTK.

These steps are shared by both indexing and query processing. Thus, it's better to put these common functions in one place. Check the util.py file.

The Cranfield document file has a special format. cran.py has been provided to simplify your work in reading the documents.

Once you get a list of terms, use the steps to put the terms in the inverted index and sort the lists correspondingly. Check the files: doc.py and index.py to understand how the classes are organized, and then implement the necessary methods - you are free to add more methods if necessary.

Finally, the index should be saved to a file. You can use any serialization method you like (e.g., JSON or any Python built-in libraries). Make sure the index can be saved/loaded correctly.

You should provide the executable indexing program that can be run as follows

```
python index.py cran.all index_file
```

It builds the index for the cran.all file and saves the index into the index_file.

Part 2: Query Processing

The query processing component contains the following steps.

- Query preprocessing, which includes all the steps used for preprocessing documents in indexing. In addition, you should also use the spelling

corrector. Please check the Norvig's implementation and understand the algorithm. The python code of the spelling corrector has been included in the skeleton code: norvig_spell.py (need some testing to make sure it work as expected).
- Process queries with the Boolean model. Check the textbook or slides for the processing algorithm.
- Process queries with the vector model. Use the standard TFIDF to represent the weights in the vector representation. Use the cosine similarity for ranking.

Your standard query processing program should be run as follows

```
python query.py index_file model_selection query.text query_id
```

where model_selection indicates the Boolean or vector model: 0 - Boolean, and 1 - vector, query.text contains the sample queries (included in the Cranfield dataset), and query_id is the specific query you choose. cranqry.py has been provided for reading the special format used by query.text. The output will be a list of document IDs for the Boolean model, and the top 3 ranked results for the vector model. For vector model, choose one of the TFIDF scoring methods, e.g., lnc.ltc, mentioned in Figure 6.15 at the page 118 of the textbook (or the same Figure in slides "scoring_idf.ppt").

Part 3: Evaluation

Check qrels.text in the Cranfield dataset for evaluating the quality of search results. You can use the ndcg_score function in metrics.py to evaluate the quality. Please finish the program "batch_eval.py" to compute the average NDCGs for the boolean-model and vector-model based query processing results, respectively, and use t-test or wilcoxon-test to get the p-value for comparison (i.e., whether one ranking result is *statistically-significantly* better than the other).

```
python batch_eval.py index_file query.text qrels.text n
```

where n is the number of randomly selected queries from query.text. The output should be the average NDCGs (for n queries) for the boolean model and the vector model respectively, and the p-value for the comparison.

(Extra credit (5%)) You can extend this query evaluation program to compare different settings of vector models (i.e., TFIDF scoring methods mentioned in Part 2): implement some different query processing methods. Then compare and report the quality of ranking results.

Please test/debug all your programs thoroughly. By designing tests, ask yourself questions like

- Are the stopwords really removed? Are the terms in index all stemmed?
- What is the number of terms in the dictionary and what is the size of postings? Do they make sense?
- Are index saving and loading working as expected?
- Do you also convert queries to terms?
- How do you confirm that TFIDF values are computed correctly?
- How do you confirm cosine similarity is computed correctly?
- Are your selected sample queries getting the same results as you expect (manually computed) for boolean model processing?
- Are your selected sample queries getting the same results as you expect for vector model processing?
- Do the NDCGs for selected sample queries match your manually computed results?

- etc.

Deliverables

Turn in three files (DO NOT zip them into one zip file) to Pilot: (1) a brief documentation about your design, implementation, and tests you have done, (2) paste your well-commented source code in one PDF file, and (3) the zipped whole source code directory. You should put your team members names in the beginning of each PDF file.

Plagiarism will get 0 directly. No deadline extension will be given.

This page modified: Feb 9, 2019