COL764 Assignment 1

Vector Space Retrieval System

Prof. Srikanta B. Jagannath

Submitted by:
Mudit Soni (2017EE10463)

Implementation Details:

Language:

Python3

Files:

- invidx_cons.py: For constructing index and postings.
- printdict.py: Prints the index.
- utils.py: Utilities
- vecsearch.py: For searching queries and generating result file.
- stopwords.txt: File containing english stopwords from NLTK.
- readme.md
- setup.sh: For installing required packages.
- packages dir: Directory containing required packages.

Packages used:

- BeautifulSoup: For parsing xml documents.
- NLTK: For stemming.
- Pygtrie: Trie data structure for implementing the dictionary.
- Pickle: For serializing the dictionary.
- Regex: For parsing queries.
- Timeit: For measuring running time.

Algorithmic Details:

invidx_cons.py:

1. For each document in the directory, the text is preprocessed using the string library. Preprocessing involves: removing punctuations and digits, removing stop

words, converting to lowercase, tokenization and stemming. Stopwords file was manually downloaded from NLTK read while execution as NLTK's builtin function was slow. The tokens are stored in a list as tuples along with document IDs. Similar to normal tokens, named entities are also tokenized and stored with their identifiers in the list.

2. Now the document list is traversed and an inverted index dictionary is generated with tokens as keys and document ID lists as values. Dictionary is used to get O(1) access as keys are updated often.
3. The inverted index dictionary is now traversed and 'df' and 'idf' are calculated and stored in another dictionary.
4. Now, doclist is traversed and vector length for each document is calculated using tf idf scores and stored in another dictionary.
5. For each token, the postings list is written in a binary file and it's offset calculated. 20 bytes are used per element, 16 bytes for the document name padded with space and 4 bytes for score calculated as (tf*idf)/(doc_len). The score is multiplied by 1e8 and converted to int from float before storing.
6. The total number of documents n, to be needed for calculating query scores, is stored separately in the binary file.
7. Finally, trie data structure is used to store all tokens and their offsets and is pickled to a binary file.

vecsearch.py:

1. All the query IDs and query text are extracted from the file using regex. Dictionary (trie) is loaded in memory.
2. For each query, the query is tokenized and wildcards and regular+named tokens are obtained as separate lists.
3. Trie is accessed and relevant documents containing any of the token are obtained by seeking offsets in the posting lists binary file. A dictionary is used to store the document scores, read from binary file, multiplied by the query score, calculated for each token, for each document. Query vector length is calculated in the same loop. The total number of documents n required for calculating idf in query score is obtained from the binary file.
4. Finally normalized scores are obtained by dividing scores with query vector length.
5. In case of wildcards, recursion is used to generate all possible combinations of eligible tokens obtained by prefix search in the trie. For each document, only the maximum normalized score, from all possible queries for a prefix, is stored in the dictionary.
6. These scores are then sorted and elements above the specified cutoff are written to the result file.

Performance:

Size of Dictionary file	5.6 MB
Size of Index file	300 MB
Time for generating inverted index	1377s
Time for generating query results	6s
nDCG score for cutoff 10	0.2177
F1 score for cutoff 100	0.1536