COSC 488: Introduction to Information Retrieval Spring 2018
Project Part 1: Pre-Processing Documents & Building Inverted Index
Sabrina Ma slm243@georgetown.edu

Cover page

a. Status of this assignment: Complete

b. Time spent on this assignment. Number of hours

2/2/18: 12:00 - 1:00PM (1 hr)
2/3/18: 5:00 - 6:00PM (1 hr)
2/4/18: 2:00PM - 6:00PM (3 hrs)
2/5/18: 8:00AM - 1:00PM (6 hrs)
2/9/18: 7:00AM - 2:00PM (8 hrs)
2/10/18: 8:00PM - 11:00PM (4 hrs)
2/11/18: 12:00 - 5:00PM (5 hrs)
2/12/18: 7:00 - 11:00PM (4 hrs)
2/13/18: 5:00PM - 6:00PM (1 hr)
2/14/18: 4:00PM - 6:00pm (2 hrs)
2/15/18: 4 - 7PM, 10 - 12Pm (5 hrs)
2/16/18: 8:00AM - 6:00PM (10 hrs)
Total: 50 hours

c. Things you wish you had been told prior to being given the assignment

I wish I had more knowledge of software development project management tools (e.g., Trello) so I could have been more organized while working on this project. Aside from that, I thought the TAs did a very good job of informing us about the project requirements and answering our follow-up questions.

Design Document

Interface

Shell interface that accepts arguments in the following format:

```
python3 build.py [trec-files-directory-path] [index-type] [output-dir]
Example: python3 build.py data/ positional output/
```

- `[trec-files-directory-path]` is the directory containing the raw documents
- `[index-type]` can be one of the following: "*single*", "*stem*", "*phrase*", "*positional*"
- `[output-dir]` is the directory where index and lexicon files will be written

Note: Please run using Python 3! I received UnicodeDecode errors when I used Python 2.x.

main()

- Parse command-line arguments
- Read stop words file
- Create output directories where index and lexicon files will be written
- Read TREC data files and pass to `preProcess`
- Sort and merge temp files in `sortAndMerge`
- Build inverted index from merged list of triples

preProcess()

- Parse each file using BeautifulSoup, a Python package for text parsing
- For each document, extract the document id and preprocess/tokenize whatever is enclosed between the <TEXT> tags
- For each document, create a dictionary `termfreq` that store term's frequency.
- For each line in document, replace escape sequences
- To tokenize, use re.split() to split line on space, period, and symbols (^,*,#,@, $ …) for the single, positional, and stem index and used nltk's tokenizer `nltk.word_tokenize()` for the phrase index, because this tokenizer keeps punctuation after parsing, which is used to determine whether a sequence of tokens was a phrase.
- Perform case folding and change all to lower case
- Add token to `termfreq` if it is not a stop word
- After a document is processed, generate triples from `termfreq` in `createTriples`
- Write triples to disk in `writeToDisk`

Pre-processing specifics for each index

- Single - found and removed special tokens from line in `findSpecialTokens()`
- Positional - used a counter to assign position to every token
- Stem - used `nltk.stem.porter` to perform token stemming
- Phrase - `preProcessPhrase()` identifies two-term and three-term phrases that do not cross stop-words, punctuation marks, and special symbols

findSpecialTokens()

- Create regex patterns for all types of special tokens. If there is a match, perform special token-specific normalization, add token to termfreq, and then remove from line.
- Strange email addresses are truncated so they have a valid format (e.g, BARNES.Don@EPAMAIL.EPA.GOV@IN -> BARNES.Don@EPAMAIL.EPA.GOV)
- Abbreviations and acronyms were converted to lowercase and stripped of periods
- Dates converted to MM-DD-YYYY format in `normalizeDate()`
    - Date ranges are not handled
- Decimal and currency were converted to whole number and rounded up
- In single term index, parts of hyphenated terms stored as well as the entire term without hyphens (e.g., hello-world-123 stored as hello, world, 123, and helloworld123)
- File extensions and IP addresses stored with period (e.g., helloworld.pdf)
- Used list of prefixes from this website: http://www.grammar-monster.com/lessons/hyphens_in_prefixes.htm
- Original IP address regex `\b(\d{1,3}\.){3}\d{1,3}\b` incorrectly matched references to chapters/procedures (e.g., 4.3.3.1). Because these can be identified as single-digits, I changed my regex such that a digit had to repeat at least twice in order to be considered an IP address, i.e. `\b(\d{2,3}\.){3}\d{2,3}\b`

createTriples()

- Iterate through `termfreq` to create the (term, document id, term frequency) triples
- If there is a memory constraint, check length of list of triples. If the length equals memory capacity, write triples to disk and empty the list.

writeToDisk()

- Writes triples to temporary files in `output-dir/temp/`. Number of triples in each temp file based on declared memory constraint.

sortAndMerge()

- Implements sort/merge-based index construction, specifically:
- Sortes temp files by term and document id
- Perform m-way merging using priority queue of intermediate files in memory and write merged list of triples onto the disk
    - Received `OSError: [Errno 24] Too many open files` because my max number of open files limit was too low. To configure, ran "ulimit -n <limit>" in terminal

buildInvertedIndex()

- Converts list of triples to inverted index and lexicon using a dictionary
- Index files in `output-dir/indexes/`
- For the phrase index, also creates a `filtered-phrase` index that only contains terms with a df > 1.
- For the positional index, the first element is the document frequency, and the rest of the elements are the positions.

Report 1: Index statistics for each index

	Lexicon (# of terms)	Index size (bytes)	Max df	Min df	Mean df	Median df
Single term index	36,637	10.3 MB	1251 term: 1	1	9.979	1
Stem index	30,211	8.9 MB	1202 term: 1	1	10.513	1
Phrase index	167,405	11.1 MB	564 phrase: billing code	1	1.61	1
Filtered phrase index (df > 1)	34,866	4.5 MB	564 phrase: billing code	1	3.945	1
Single term positional index	38,311	20.8MB	1742 term: of	1	12.25	1

Report 2: Running time based on each given memory constraints and each index type using sort-based inversion method

Single term Index

Triple List Size	1,000	10,000	100,000	Unlimited
Time taken to create temp files	25.17s	20.82s	22.51s	19.14 (Time to create triples)
Time taken to merge temp files	2.32s	2.08s	1.65s	N/A
Time taken to build inverted index in ms	26.55s	23.677	22.66s	20.80s

Stem Index

Triple List Size	1,000	10,000	100,000	Unlimited
Time taken to create temp files (up to merging)	34.61s	34.38s	33.88s	33.81s
Time taken to merge temp files	2.73s	2.26s	2.06s	N/A
Time taken to build inverted index in ms	59.88s	54.00s	37.38s	34.04s

Phrase Index

Triple List Size	1,000	10,000	100,000	Unlimited
Time taken to create temp files	40.70s	38.64s	38.10s	37.89s
Time taken to merge temp files	2.44s	1.64s	1.88s	N/A
Time taken to build inverted index in ms	46.64s	45.86s	45.34s	43.77s

Positional Index

Triple List Size	1,000	10,000	100,000	Unlimited
Time taken to create temp files	16.32s	14.45s	12.16s	11.49s
Time taken to merge temp files	4.54s	3.28s	2.77s	N/A
Time taken to build inverted index in ms	22.38s	19.13s	17.23s	13.14s

Analysis

The index with the smallest lexicon was the stem index, which had ~30,000 terms. This is because the Porter Stemmer reduces variations of each word and create a common stem. According to the Pre-Processing lecture slides, stemming reduces the term index by ~17%. This is consistent with my results, because my single term index lexicon was 36,637 terms and 36,637 * .83 = ~30,000 terms. The index with the largest number of terms was the phrase index with ~167,405 terms. This was due to the larger number of special characters that exist in the file that the nltk tokenizer did not split on. After filtering the phrase index to only include terms with a document frequency of one or more, the lexicon size went down to ~34,000, which is comparable with the other indices' lexicon sizes. The single term positional index contains more terms than the single term index because the positional index includes stop words. Additionally, special token normalization was performed for the single term index. Therefore, tokens like date were standardized. This makes it more likely that tokens map to the same term.

The average index size was 11.12 MB. The single term positional index (20.8 MB) was twice as large as the single term index (10.3 MB) because the positional index stores not only each term's document frequency, but also each term's position for each document. In retrospect, I could have optimized the index sizes is by adding less formatting. The format of each index is `term -> [[doc1, df], [doc2, df], ... ]`. I could have saved space by removing the " -> " and the square brackets.

The maximum document frequencies of the single and stem indices were 1,251 and 1,202, both for the term "1". For the phrase index, the max df was 564 for the phrase "billing code." The max document frequency for the single term positional index was 1,742 for the term "of," which is a stop word. This indicates that stop words can occur many times in a collection and are often not discriminating or useful.

The mean document frequency for the single term, stem, and positional index were all around 10. Because the median document frequency for all indices is 1, I infer that the distribution of document frequencies across all terms would be skewed right.

Across all the indices, the fewer the temporary files (and larger the triple size), the faster the running time. This is because fewer temporary files means less overhead in the form of read, write, sort, and merge operations. Across all indices and memory constraints, the time taken to merge temp files was under 5s, which makes it a pretty cheap operation. The positional index running times was the fastest because it requires the least pre-processing. The single term index runtimes were several seconds longer because every file line had to be checked for special tokens using regular expressions and processed accordingly. The stem index took around 10s longer than the single and positional, perhaps because each token had to be processed by a stemmer object. Finally, the phrase index took the longest, possibly because of the additional logic in the preProcess method, larger lexicon size, and additional filtered index creation.