Report for Information Retrieval

Yuanqing Jiang S3289950
Yuwei Zhang S3492095

The indexing and searching mechanics are all implemented in Python, with a clean and well designed OOP model to allow reusing code and further editing. Our custom data structures and class definitions can be found in util.py.

1. Indexing Construction

1.1 Parsing the information collection

Our parsing module builds the inverted index based on the following algorithm [2]:

The documents are read line by line, and tags are escaped during the iteration, except that when encountering a <DOC> tag, the module detects a new document is being read, hence increase the document count by 1, which is later one used as document ID.

Every ordinary plain line will be split by whitespace by .split(). The returned array of raw terms will then be passed to a tokenize function, which starts to process each of the passed terms with a sequence of operations.

After processing the terms, all finalized terms will be appended to a list, which will be return from the ParseMod to the entry point of the program for index construction.

1.2 Tokenize Terms

The tokenize behavior is modeled with 2 connected functions in our program: tokenize and normalize.

Once the tokenize function receive the list of unprocessed terms, it will first use .strip() to clean up new line characters and escape any left tags, then it will use .lower() and .split('-') to normalize term to lower case and break hyphenated word into two. Then by calling str.translate() in Python, all punctuations will be removed.

Furthermore, each tokenized term will be passed to normalize() for further normalization.

The entire process will return a list of tokenized and normalized terms. As we have given more thoughts in how to handle numbers and a better normalization of terms, the scope of the assignment did not leave us much rooms for a detailed implementation of normalization.

1.3 Information merge

The previous operations will return a list of tuple<term, docID>, which will then be sorted first.

As the program was well constructed in a OOP model, we have build classes for most entities involved in indexing. Therefore we merge our processed information by initializing a Lexicon object that holds a dictionary mapping term strings to Termitem object that has a instance variable of InvertedList.

The program will do that by going through each term in the sorted list and call lexicon.updateTerm() to either add or update terms and its document frequency, this function will invoke other functions in TermItem object and InvertedList in a chain of responsibility.

In the end there will be a lexicon object with an list of lexicons and inverted lists in the memory. And will be output to files.

2. Stoplist

The data structure of stoplist module is built-in dictionary. Since we are using python, we decided to use the built-in dictionary which has an efficient hash function and can be easily accessed.

Dictionary is one of Python's built-in data types, which defines a one-to-one relationship between keys and values. As only standard mapping type of Python, the dictionary supports an associated container that stores a pair of key-values (mapped by keys to values). The implementation of dictionary in Python provides hash function by requiring key objects to reduce the complexity of the dictionary finding O(1) average. Also the solution to the hash conflict is open addressing. When a conflict occurs, a probe (or measured) technique is used to form a probe (test) sequence in the hash table. Look up the units one by one until the given keyword is found, or an open address (the address unit is

empty). Searching failure means find the open address when the search shows that the table no pending keywords.

3. Index Search

3.1 Data Structures

We have implemented our very own data structures for lexicon, term items, and inverted list.

The lexicon object will be the entry point for accessing data in memory, which holds a dictionary that maps term strings to the Termitem object, allowing quick retrieval of information. The Termitem object holds the term string, document frequency, and a instance variable of InvertedList object.

The Inverted List object models the abstract inverted list, it contains a dictionary with pairs of <documentID, inDocumentFrequency>, it also allows quick retrieval of frequency information when document ID is known.

This OOP approach greatly enhanced understandability and reusability of code.

3.2 Searching of the corresponding word

As the inverted list file is written in binary form, using Python's struct object and pack() and unpack() functions, we are storing and looking up information in invlists file in the following steps:

When writing to file, the offset position will be calculated by the number of bytes being written in the previous iteration, and will be recorded as a 'pointer' with integer value in the lexicon file.

When querying occurs, the program will read from the specific position in the invlists file according to the offset position stored in lexicon, by invoke .seek() in Python, and the number of bytes being read into the memory will be determined by the frequency information stored and mapped to each term in the lexicon. The total number of bytes to read for each term can be denoted as such:

For every document it occurs in, there will be 2 integers, and each takes 4 bytes in binary file

totalBytes = docFreq * 2 * 4

4. Index size

The size of our inverted index is:
Invlists: 13KB
Lexicon: 26KB
Map: 80 Bytes
testDoc: 31KB
(To reduce the time of computing during development, we made a short version of the original document collection)
The combined size of our index seems to be a bit bigger than the original file, even though the invlists file is written in binary form.
I believe the reason behind it is that we have added much more additional information on top of the original documents, such as the term frequency, document frequency, pointer / offset information, which would naturally increase the size of the inverted index.

5. Limitations

I believe there's no significant bugs or error that I am aware of after the final bug fixing before the submission. And I am particularly satisfied with the OOP model design and the way the code and components are structured and written after putting in a lot of effort trying to achieve perfection.

However, if more time is given, I believe improvements can be made in the following aspects:

1. Team management: having one person stressing and writing up all the code and design is not a good idea, the other person ends up in a situation where he/she could not effectively understand the code hence can not come up with a report to hand in
2. The command line module: the current modules assumes that the input given by the user is correct and valid. Although I have implemented some kind of error displaying function but I believe it's not entirely sufficient to keep the system from crashing.

3. Compression: The current inverted index is still bigger than the original documents, which is not optimal, I believe implementing a compression module will help a lot.

6. Contributions

Well, I, Yuanqing Jiang, do not want to be the bad person because at the beginning of the semester I did agree to do most of the heavy lifting work and I did, all the code.

However after spending hours and hours coding I don't get to see my partner that often during lectures and tutes. I would confirm that my partner wrote 50% of the report which I later on had to come back to edit for grammar and technical errors.ß

S3289950 Yuanqing Jiang	**70%**
S3492095 Yuwei Zhang	**30%**