THE UNIVERSITY OF EDINBURGH

TEXT TECHNOLOGIES FOR DATA SCIENCE

FIRST ASSIGNMENT

Pavlos Gogousis
p.gogousis@sms.ed.ac.uk
Student ID: s1884197

Approach and Structure

Since the proposed programming language to complete the assignment was Python, the approach follows an object oriented model. The classes created to develop the basic IR system are the following.

- Invoker: Python script that invokes necessary modules.
- Preprocessor: Class providing methods for text preprocessing.
- InvertedIndex: Positional inverted index structure.
- QueryProcessor: Class implementing the retrieval module.

Each of the modules is analyzed in more detail below.

Invoker

Simple script that initializes and invokes necessary data types and methods.

Preprocessor

Package built to provide a toolkit for text processing in the Inverted Index and Query Preprocessor modules. It supports:

- **Tokenization**: The tokenization method used was based on the regular expression `(?!\&\b)\W+` which basically splits on every multiple non-alphanumerical character occurrence (`\W+` part), except the case where a "&" character is surrounded by alphanumerical characters (`(?!\&\b)` - negative lookahead part). For instance, the word "AT&T" should not be split, whereas words like "game's" or "state-of-the-art" should. The basis for this is that after testing many different tokenization techniques, this one seemed to retrieve the best results, since it creates a more broad index (term-wise).
- **Stemming**: Words were stemmed using the Porter stemmer library, due to the fact that it was the proposed method to stem terms. The Preprocessor package implemented also supports stemming with the Snowball library. The results were not significantly different, though.
- **Lowercasing**: Word characters were converted to lower case.
- **Stopword removal**: Stopwords were removed by using the proposed stopword list. Other packages were also used, but the results were not as good, since the other stopword lists were quite shorter.

Inverted Index

Class that implements the positional inverted index data structure. The data structure used to store the index in memory is as follows:
`{term, {document ID, [positions]}}`, which basically means that it is a dictionary whose keys are terms and values are nested dictionaries, whose keys are document IDs and values are lists of positions. The reason this structure was chosen is because we need to be able to retrieve term values instantly (`O(1)`), document values also instantly, and lastly positions can be stored in a list since we have to iterate over them.

The class provides methods that:

- **Build index from file**: The given file contains the collection of documents. The file used is of the TREC format, so it is parsed with the help of the Python xml parsing package. In the implementation it is assumed that the collection does not fit in memory, so the safest approach is to parse the document collection file in a sequential manner. This means that the text is retrieved element by element (in the xml tree). ”Headline” and ”Text” tags’ contents are merged and the Preprocessor module is used to preprocess the combined text.
- **Insert to and retrieve from index**: Insertion and retrieval methods are implemented to insert occurrences of terms, as well as to retrieve the document IDs and positions of a given term.
- **Export index to file**: Function that stores the positional inverted index to file following a certain structure. Afterwards, the file can be loaded from the QueryProcessor module, since reading a file containing the index is much faster than building the index from scratch.

Query Processor

Class that functions as the engine that processes queries and retrieves documents from the already built positional inverted index. This class loads and maintains a positional inverted index from a file so that it can perform boolean search and calculate TF-IDF scores for given queries. All query content is tokenized and preprocessed the exact same way as the positional inverted index is built.

Methods implemented within the class are described below:

- **Boolean search**: Boolean search supports 3 main operations:
    1. Expressions with `AND,` `OR` and `NOT` operators
    2. Proximity search for two terms

3. Exact phrase search for two terms

The implementation considers two types of expressions.

– **Simple expressions**: Simple expressions consist of just a term or an exact phrase (phrase handler invokes proximity handler) with or without the `NOT` operator infront of them. Also, another type of a simple expression is a proximity query, though it is a standalone expression, meaning it cannot be part of a complex expression.
– **Complex expressions**: Complex expressions are combinations of simple expressions (except proximity) with an `AND` or `OR` operator.

Therefore, we consider the following: `expA (op expB)`, where `expA` can be any of the following type: term, phrase or proximity query, and optionally an operator (`op`) can follow `expA` with an additional simple expression `expB`, which in turn can be another term or phrase.

The query processor parses each query and determines whether the expression is of simple or complex type. If it is simple the corresponding set is retrieved and returned. In case it is a complex expression, it is broken into parts and afterwards treated as an intersection (`AND`) or a union (`OR`) of simple expressions.

Lastly, the proximity query format is `#n(term1, term2)`, which means we want `term1` and `term2` to be at most `n` positions apart from each other. The phrase search is handled exactly like a proximity query where the distance is set to exactly `1` and the order in which the terms occur is strict. This means that `term1` should appear before `term2`, not the other way around. Within the method, the comparison is implemented using the linear merging algorithm.

In all cases the result is a set of documents that satisfy the query.

- **TF-IDF ranked retrieval**: This module calculates the TF-IDF score for each of the documents that correspond to the term listings. To be more specific, for a query such as "`term1 term2 term3 ... termN`", the engine will retrieve all documents with these postings and calculate for each of them the TF-IDF score based on the provided formula. As mentioned before, the query is preprocessed before retrieving results.

- **Write results to file**: Both the boolean and the TF-IDF query processors export their results to a file. In case a query does not match any documents, therefore cannot retrieve any results, "`null`" is written at the document ID and value fields. Additionally, as asked, only the first 1000 results are exported - booleans are in ascending order based on the document ID and TF-IDF are sorted in descending order based on the document score.

- **Import queries from files**: The query processor has a method to import the boolean and the TF-IDF queries from given paths leading to the corresponding files.
- **Import index from file**: Method that parses a positional inverted index file and stores it in an Inverted Index data structure. Again, the parsing is implemented sequentially, since we cannot assume the whole file fits in memory to load at once, as in the Inverted Index module.

Comments and Improvements

Modern information retrieval systems are very efficient and precise. Thus, building such system is very challenging. While implementing the tasks various difficulties were faced, such as how to handle complex queries, thinking of the most efficient ways to perform various tasks, as well as choosing the appropriate data structures and algorithms.

The engine built performs quite simple tasks. There could be many improvements, such as involving threads to process queries in parallel, dynamically extending the collection and the inverted index, and providing more flexibility to the way the user generates queries.