CS F469 INFORMATION RETRIEVAL

AN ANALYSIS REPORT
ON
ASSIGNMENT - 1
SEARCH ENGINE FOR DOCUMENTS

BITS PILANI, HYDERABAD CAMPUS

UNDER THE SUPERVISION OF
DR. ARUNA MALAPATI

Group members :

Name	ID No.
Shashwat Anand	2019B3A70718H
Sailesh Duddupudi	2019B3A70632H
Parth Kulkarni	2019B3A70706H
Vineeth Kumar	2019B3A70220H

Table of contents

1. Architecture of our Search engine

In our implementation of the search engine, we used the Inverted table index. Inverted Index is a data structure used by search engines to optimize search performance by quickly locating relevant documents. It works by mapping the unique terms in the documents to the documents that contain them. In this report, we will explain how to implement a document search engine using Python and an inverted index table.

The architecture of our search engine is briefed as follows :

1) Converting Documents to text files
2) Preprocessing the text
3) Building the Inverted Table Index
4) Processing the Query

In the following pages, we will explain each of the points thoroughly.

2. Converting the Documents to text files

We initially tried using the PyPDF2 function in Python to read the pdfs but this caused a lot of issues and didn't read the files properly. Hence we used a pdf reader to manually convert all the pdfs to .txt files. Another issue we faced was that the spacing between each paragraph wasn't captured. So to split the text into paragraphs, we used an algorithm to determine what consists of a paragraph which is discussed later in this paper.

3. Preprocessing Document Text

We do the following operations for preprocessing the text to add to build the Inverted table index :

1) Remove Punctuation : We removed all types of punctuation which consists of “!()-[]{};:'"\, <>./?@#$%^&*_~”. Note that we didn’t remove periods for now since we will be using them to figure out what consists of a paragraph.
2) Lowercase : We converted all the alphabets to lowercase so the index will be made case insensitive.
3) Tokenizer : We used the NLTK Tokenizer (word_tokenize) to tokenize the preprocessed text.
4) Stopwords : We downloaded the stopwords set from the NLTK Corpus which we then used to filter out while forming the index and for the query as well
5) Stemming ** : Here, we used the Stemmer made by Martin Porter for Python (PorterStemmer library) to perform stemming on the previously processed text as well as the query text.
6) Storing text : We stored the text of all the documents in a single text file and also stored the starting and ending line numbers for each document. This way we can easily identify which paragraph belongs to which document.

** - We implemented stemming but it led to a lot of false positives in the results. So we chose to comment it out, but the implementation can still be found in our code.

4. Inverted Table Index

Inverted index is a data structure used in information retrieval to quickly find all documents that contain a specific word or set of words. It is created by analyzing all documents in a corpus (collection of documents) and extracting the unique words (terms) from them.

For each term, an inverted index stores a list of all the documents in which the term appears, along with the positions of the term in those documents. This enables efficient searching of the corpus, as the search engine can quickly locate all documents containing a particular query term by looking up the corresponding list in the inverted index.

In our implementation we used the data structure Dictionary in Python. This essentially a map where the key is the preprocessed word / token and the value is a list of sentence numbers where the word occurs. We know which document this belongs to since we stored that information separately. We then use an algorithm to find the paragraph that this sentence belongs to.

To find which paragraph the sentence belongs to, we first calculate the number of characters in each line then calculate the average value for characters in each line. Once we have the average we iterate through each line checking which line is less than average number of characters and ending with a full stop, thus this line becomes the end of the paragraph and the next line is the start of a new paragraph. This process is done through the document to get a list of starting and ending lines of paragraphs.

This dictionary is then stored as a json object which can be then used for fast and efficient search and retrieval.

5. Processing the Query

The query is inputted through the query.txt file. The query can be multi-word as well. We do the same preprocessing on the query, that we do on the document text, which includes removing punctuation, converting to lowercase, removal of stopwords, tokenization and stemming**. Once this is done, we search for each word in the Inverted table index and find all the lines it's present in and hence the paragraph. We do this for all the words in the query. We then do an intersection operation between the sets of paragraphs returned for each word of the query. This will return the paragraphs containing all the non-stopwords used in the query.

6. Run/Execution Time

Building the Inverted Table Index took a long time to execute (around 30-35 minutes). But the search and retrieval for the query is quite fast. The following is the time data for the corresponding queries :

S No	Number of Words in Query	Time (sec)
1	4	1.6
2	10	1.3
3	3	2.6
4	3	1.1
	Average	1.65