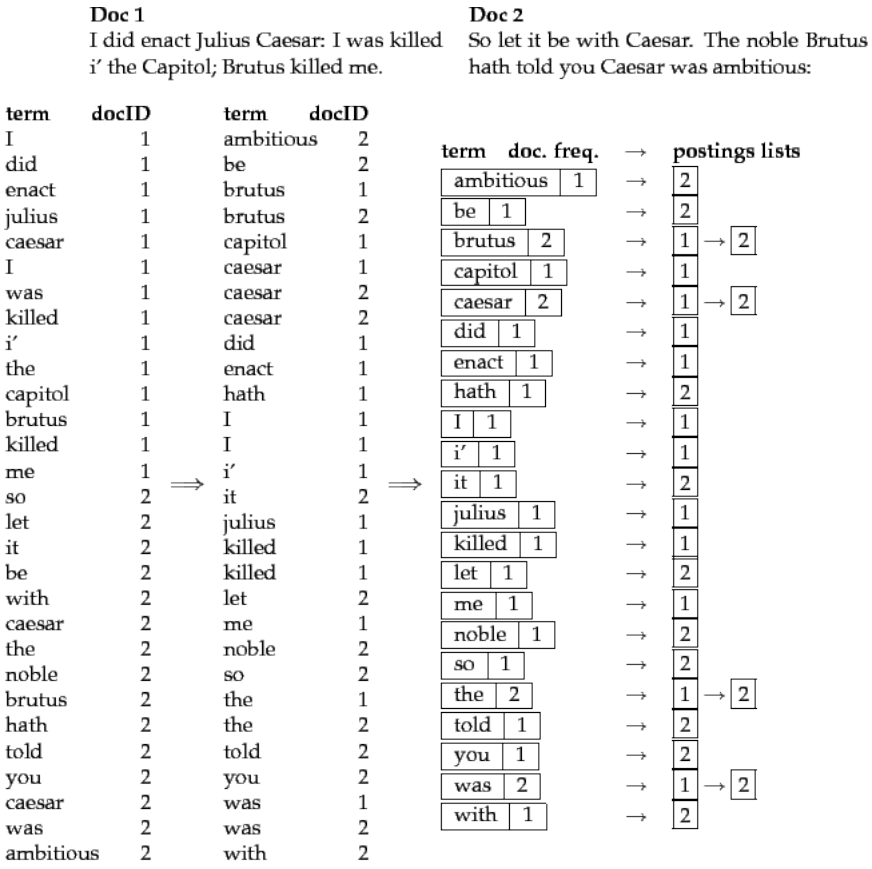
Web Search using Inverted Index
MTL342 Project Report

Ankit Kumar (2017MT10727)

October 20, 2019

1 Introduction

Information retrieval is the activity of obtaining information system resources that are relevant to an information need from a collection of those resources. In this project I focused on using Inverted Indexes (mostly) in text based information retrieval. The techniques I worked on and the results I obtained are for text based systems, and may not hold exactly for Multimedia Systems (although, similarity in trends can be expected). The data set I have used for testing my implementations is described in section 3.
The general procedure for inverted index creation is descried loosely in below pic:



- Create unsorted vocabulary containing (term, docId)
- Sort the pairs
- Merge to get doc-frequency and posting list

For stemming, I have used porter stemmer (present in `stem` directory in the zip). I have also used the boost library heavily for string processing, so it is necessary in order to test the codes.

2 Implementations

- Boolean Retrieval
  - Basic & Advanced Boolean Retrieval Model
  - PageRank Algorithm
- TF-IDF Model
- BM-25 Model
- Phrase Queries
- Inverted Index for Large Scale Systems

Note that for the sake of testing, appropriate `Makefile` are present for easy compilation.

3 Data Set

The dataset is present in `testCases` folder. For the very last implementation (Section 8), I made multiple copies of the webpages present in `testCases` and are present in `testCasesLarge`. It's size is close to 410 mb, and can be generated by running `copyDataset.cpp` file (present in the `largeCollectionInvertedIndex` folder). I did not include this in the zip file because of the large size!
The dataset was obtained from: http://www.gutenberg.org/browse/scores/top
Unless otherwise stated, consider $d = \#Documents$, $q = \#$ terms in query and $V =$ vocabulary size.

4 Boolean Retrieval

This is the very first and most basic implementation in IR. In Boolean Retrieval, we can pose any query which is in the form of a Boolean expression of terms, that is, in which terms are combined with the operators AND, OR, and NOT.

4.1 "Basic" Boolean Retrieval

In this part, I'll discuss the implementation with only a single Boolean operator.

Let's take the example of AND. When user gives the query: A AND B, what he wants to know are the documents that contain both the terms, A and B.
We create a $V * d$ matrix (call BooleanIndex) that contains 1 at (i, j) position if i^{th} term in vocabulary is present in j^{th} document. So, **Space Complexity** $= O(Vd)$. Since we need to fill the matrix for each term for each document, the **Time Complexity** $= O(Vd)$.

Similarly, it can be seen that for both OR and NOT operators too, the space and time complexity will remain the same.

4.2 "Advanced" Boolean Retrieval

In this part, I'll discuss all the three operators together. Suppose the user wants to retrieve a document that has both A and B, may have C, and should not have D. Since the user cannot write English to explain his needs to computer, I implemented the system where this query can be written as:

$$AND \text{ A } AND \text{ B } OR \text{ C } NOT \text{ D}$$

All possible exceptions are explicitly handled in the code: and/or/not are missing, some other term is used for and/or/not, stop words are inserted, query terms are missing, etc. Then, for each query term, I find the subset of results at the current stage that satisfy the given query, using individual functions for and, or and not, & the final output is then displayed. The Space Complexity and Time Complexity remain the same.

4.3 Ranking: PageRank

We need to rank pages in some order when shown to the user. Boolean retrieval, as one of its disadvantage, doesn't provide any numerical relevance judgements. Thus, we do not have a standard ranking method for Boolean Retrieval as such. However, we can use PageRank, a ranking algorithm by Google, to rank our results.
PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.

4.3.1 Algorithm

- Given: Adjacency matrix M, damping factor d, number of iterations
- n be number of webpages, $n = \#\text{rows in } M$
- Generate a random vector v with values in $(0, 1)$. Make l_1 norm of v by taking: $v = \frac{v}{||v||}$
- Repeat for number of iterations:
  - $v = d * M * v + (1 - d)/n$
- v now contains final scores of the pages

Space Complexity = $O(N^2)$, where N = # pages retrieved, because we want to know whether there's a directed link between pages (i, j).
Time Complexity = $O(N^2)$ because a major step involved in this is matrix multiplication ($M_{N*N} * M_{N*1}$).

4.4 Code

`Boolean Retreival` contains this in the files `booleanIndex.cpp` and `booleanQuery.cpp`. `booleanIndex.cpp` creates the $V * d$ matrix and writes it to disk, which is then read by `booleanQuery.cpp` which gives relevance judgements for input queries. Note that this file also contains the `pageRank` function, but it isn't used because I didn't have links present in the data set (I verified the function to be correct anyway by generating appropriate test samples).

5 TF-IDF Model

Boolean Retrieval has certain drawbacks. Partial matches are not allowed. Query vocabulary is precisely the same as collection vocabulary. Provides no "good" ranking as such. A file with 100 "stack" is as good as a file with 1 "stack". These issues can be resolved to a good extent by using TF-IDF model.

5.1 TF-IDF

In TF-IDF model, we can assign non-binary weights to index-terms in queries and documents. TF refers to term frequncy, and is the frequency of a term in a particular document. IDF refers to Inverse Document Frequency. It is a measure of the "uniqueness" of a term. If a term is present too many doucments, it is probably not that important and can be ignored (Zipf's Law).
The query is no longer a boolean expression , but a free-form sequence of words. We compute the similarity of a document w.r.t. to the query and rank the documents based on their similarity score. The query can match partially too.
There are two ways to determine the TF corresponding to a term:

$$\text{Raw frequency: TF} = f_{i,j}$$

$$\text{Log Normalized: TF} = 1 + log f_{i,j}$$

where $f_{i,j}$ is the frequency of i^{th} term in j^{th} file.
For IDF:

$$\text{IDF} = log(1 + \frac{d}{df_i})$$

where df_i is the number of documents the i^{th} term is present in.
Space Complexity = $\sum_k O(occ_k)$ where occ_k is the number of occurrences of the k^{th} term.
In worst case, **Time Complexity** = $O(Vd)$, when each term is present in each document. In practice, the number of documents in which a particular term appears is very less and hence the time taken will be much lower than that.

5.2 Code

`TF-IDF` contains this in the files `Indexer.cpp` and `tfIdf.cpp`. `Indexer.cpp` creates the inverted index according to the TF-IDF model which is then read by `tfIdf.cpp`. Note that this file contains both the raw frequency TF(`rawTFIDF`) and log normalized TF(`logNormTFIDF`) for the sake of implementation, but I have used `logNormTFIDF` function while creating the inverted index. Output of a particular query are webpages in decreasing order of relevance (relevance is also shown). The number of results shown = min(50, result).

6 BM25 Model

TF-IDF model works pretty well, but there are certain disadvantages. First of all, it doesn't have any mathematical explanation on why it is related to relevance. Also, it assumes independence of terms. It may not always be true, for eg., names, organisations, technical terminology, etc. BM25 on the other hand, provides a much better explanation.

6.1 BM25

Okapi BM25 (BM is an abbreviation of best matching) is a ranking function used by search engines to estimate the relevance of documents to a given search query.
Relevance of a document for i^{th} query term is given by:

$$qf_i.\ log(1 + \frac{d}{df_i}).\ \frac{tf_i(1+k_1)}{x},\ x = k_1 * ((1-b) + \frac{b*dl}{dl_{avg}}) + tf_i$$

Note that k_1 and b are hyper-parameters, to be adjusted according to performance. Usually, k_1 is kept between 1.2 & 2.0 while b is kept 0.75. In my implementation, I have taken $k_1 = 1.25$ and $b = 0.75$.

Explanation:

- qf_i, frequency of the term in query, is in multiplication because if a term is present multiple times in the query, it clearly means that the user wants to see results more related to that term compared to other terms in the query.
- IDF term has the same explanation as in TF-IDF section (Zipf's Law).
- Term frequency, tf_i, is in multiplication since more times a document contains a particular term, higher is the probability of it being relevant to the topic represented by the document.
- If a document is spam, contains unnecessary repetitions, then its length dl is expected to be relatively much larger compared to the average document length (dl_{avg}). Thus, both term frequency and the relative document length can be used to identify a spam document. Since we want to push such a document down in the results, there is a division with x.

Note that since BM25 is just providing a ranking function, the space and time complexity of both BM25 and TF-IDF is the same. It's the evaluation on which BM25 wins over TF-IDF model (and every other model, except Language models).
Side note: At the extreme values of the coefficient b BM25 turns into ranking functions known as BM11 (for $b = 1$) and BM15 (for $b = 0$).

6.2 Code

`BM25` contains this in the files `Indexer.cpp` and `bm25.cpp`. `Indexer.cpp` creates the inverted index according to the BM25 model which is then read by `bm25.cpp`. Output of a particular query are webpages in decreasing order of relevance (relevance is also shown). The number of results shown = min(50, result).

7 Phrase Queries

A phrase search matches only those documents that contain a specified phrase, such as `"Sherlock Holmes"`. For ranking, I have used the number of occurrences of the phrase in the document. Higher number of occurrences impliy higher rank.

7.1 Using AVL tree

Using AVL tree, I did the insertions as: (`wordIndex` is the index of word in the file)

- If at any node, the wordIndex of term to be inserted is lesser than the index of current node, insert in left AVL subtree.
- If wordIndex is greater, insert in right AVL subtree.
- If wordIndex is same, it means the terms are from different files. In this case, I put them according to their file name.
  - If file name of current lexicographically is smaller than current node's file name, insert in left AVL subtree.
  - Else, insert in right AVL subtree.
  - Names of 2 files can't be same, and hence no conflicts occur.

7.2 Using HashMap

AVL is a little slower due to $O(logn)$ time is search (ignoring the index creation speed- it's query answering speed that matters the most), where n is the total number of words present in the entire collection in the worst case– n can be huge! Hash map provides a much faster alternative- $O(1)$ on average. In the file `phraseQuery.cpp`, both hash map and AVL tree implementations are present, but I have used only the hash map one.

7.2.1 Algorithms

- Create inverted index in similar manner as TF-IDF, but, instead of storing the number of occurrences, store the actual occurrences (positions) of each term for each page.
- For each query
  - Remove stop words from the query, apply stemmer
  - If any of the query term does not exist in the inverted index, this phrase won't exist in the any of the files. So, no results found.
  - Otherwise, for each term, take the files it is present in. Take the intersection of all these sets to find the set of files all of these terms are present in.
  - For each page, start from the very first query term, and check if the consecutive terms present are the once in query. If all terms are present, increase phrase occurrence counter for this file by 1.
- Return the files, in decreasing order of number of occurrences of the query phrase.

7.3 Code

`Phrase Query` contains this in the files `phraseQuery.cpp`. I did not have another file to create the index offline because in case of AVL tree, we would need to re-create the entire tree again, and is a large overhead. Index creation and query answering are both done in the same file. Output of a particular query are webpages in decreasing order of relevance (relevance- no of phrase occurrences- is also shown). The number of results shown = min(50, result).

8 Inverted Index for Large Scale Systems

The size of dataset is usually huge- of the order of terabytes! A common step in inverted index creation, irrespective of the implementation, is sorting the vocabulary to get proper postings lists. It isn't possible if the dataset is this big. In such cases, we split the data into many smaller files that can be processed in memory and later merge them.

8.1 Algorithm

- Create an unsorted file (leftmost file in Section 1)
- Split this file into n files, of almost equal size, where n is chosen such that the individual files can be processed in-memory
- Within each file:
  - Sort the file in-memory
  - Generate (termId, ¡docId¿) pairs. ¡docId¿ is vector storing files the term is present in
  - Store positional information, document-term frequency etc. according to the implementation
  - write to disk
- Merge the sorted runs
  - Take m files (OS-dependent)
  - For each file, pick the first line, choose the ASCII-based lowest string. Write to new file
  - Continue above step until all lines in all files are processed. Delete the m files afterwards (Memory efficient- these files aren't needed later)
  - Continue iterating until one file remains

The time complexity is dominated by the sorting step theoretically. But in practice, is the step 1 and step 4 (merge) that take the major time. Time Complexity = $O(occ)$, where occ = number of words present in the entire collection.

8.2 Code

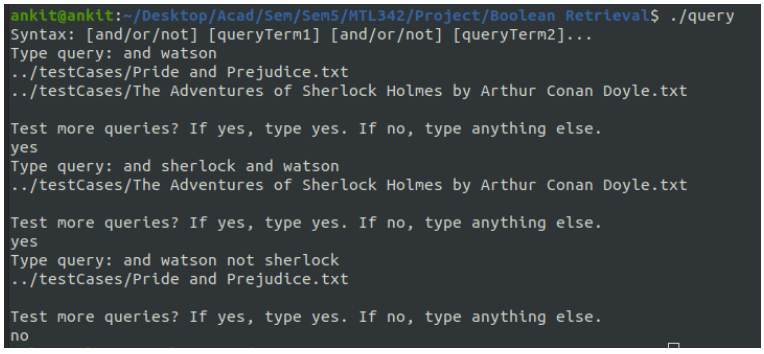
`largeCollectionInvertedIndex` contains this in the files `Indexer.cpp` and `copyDataset.cpp`. `copyDataset.cpp` duplicates the dataset. `Indexer.cpp` creates the inverted index.

9 Sample Results

9.1 Boolean Model

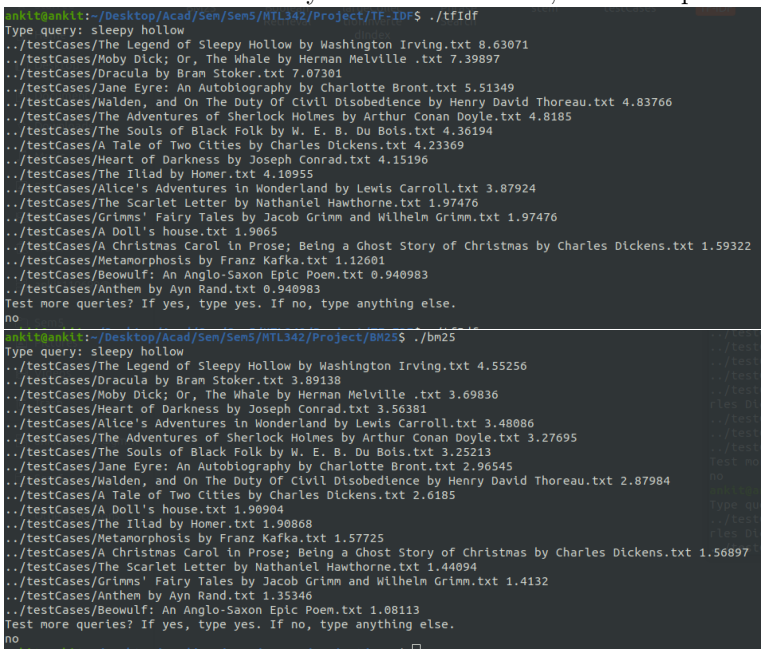
Queries are:

- All files with watson
- All files with sherlock and watson
- Files that contain watson and not sherlock



9.2 TF-IDF Model v/s BM25 Model

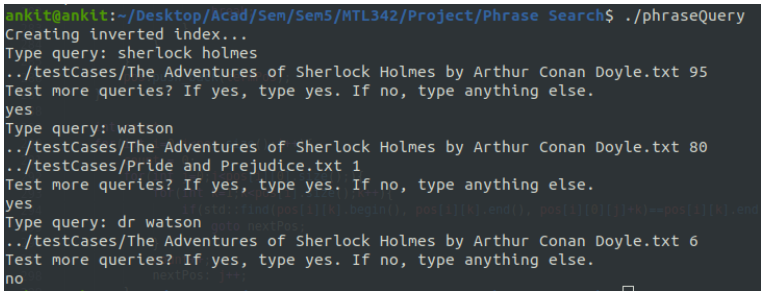
It shows the search reults corresponding to the search term `sleepy hollow`. The very first file is obviously the most relevant one. This query also shows the difference between BM25 and TF-IDF. Even though the first document retreived by these 2 is the same, the subsequent ones differ significantly.



9.3 Phrase Queries

Queries are:

- `Sherlock Holmes`: Retrieves the only file containing this phrase. There was one another file that contained `Holmes`, but not the complete phrase.
- `Watson`: Retrieves all files containing the term `Watson`.
- `Dr Watson`: In `Pride and Prejudice`, the `Watson` referred isn't a doctor, and hence there is no occurrence of this phrase. Thus, the novel on Sherlock Holmes is rightly retrieved in this case.



9.4 Large Collection Inverted Index

The below pic shows the index creation and time taken in the same for the large collection.

10 References

- Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze. An Introduction to Information Retrieval [major reference]
- https://en.wikipedia.org/wiki/Full-text_search
- https://en.wikipedia.org/wiki/Boolean_model_of_information_retrieval
- https://en.wikipedia.org/wiki/Okapi_BM25
- https://en.wikipedia.org/wiki/Tf%E2%80%93idf
- https://en.wikipedia.org/wiki/External_sorting