

Assignment 1 (Due: March 3) (30 pts)

In this project you will design and implement your own information retrieval system. The project has two phases. In Phase I, you will build the indexing component, which will take a large collection of text and produce a searchable, persistent data structure. In Phase II, you will add the searching component, according to Vector Space Model.

The project may be done individually, or in a group of two. Both members of a group are expected to contribute to all aspects of the project: design, implementation, documentation, and testing.

Phase I

Phase I of your project will read a set of files, parse them into documents (in case multiple logical documents are in the same physical file) and terms, and produce an inverted index associated data structures. The latter will be stored on disk and used in Phase II.

Phase I will have two major components: the *lexical analyzer* and the *inverter*. You need to make explicit what assumptions you make about the structure and words in the documents. You might choose to have your lexer configurable at run-time; the configuration file would then specify how to segment terms, what tag indicates the start of a new document, how to treat numbers, etc. Pay particular attention to the choice of data structures and algorithms.

Your program will need to save several data structures to disk. At the minimum, these will include the lexicon (the table of words occurring in the text, appropriate metadata/statistics, and pointers into the inverted file), the document location list (a table indicating where to find the files on disk at retrieval time), and the inverted file itself.

Note that in order to fully implement the vector space model you may need to retain several pieces of information about the documents in the dictionary / document location list / inverted index data structures such as the total number of documents, the maximum term frequency for each document, the length of the weight vector for each document, etc. Try to save values that you anticipate are needed to calculate cosine similarity that can be generated only with high computational cost if not present.

Phase I resources

Here are some resources you might find useful in Phase I:

Stop Lists

These are some commonly-used stoplists. You may find that you want to edit them somewhat.

- SMART's stop list
<http://www.lextek.com/manuals/onix/stopwords2.html>
- Stop list from van Rijsbergen's textbook
http://www.dcs.gla.ac.uk/idom/ir_resources/linguistic_utils/stop_words
- Stop list from Frakes and Baeza-Yates' textbook
<ftp://sunsite.dcc.uchile.cl/pub/users/rbaeza/irbook/>

Stemmer

Martin Porter's web page <http://tartarus.org/~martin/PorterStemmer/> contains implementations of the Porter Stemmer in a number of languages.

Milestones for Phase I

1. Parse the files containing the documents into individual documents and words.
2. Invert the collection in memory.
3. Store the inverted file and associated data structures to disk.

Benchmarks for Phase I

1. Which collection are you using for your project?
2. Corpus statistics:
 - How many documents are there in the collection?
 - How many words are there in the collection and how many unique words do you index from the collection?
 - How many postings (inverted file entries) did you create for this collection? What are the lengths of the shortest and longest postings lists? What is the average postings list length? (State your answers in terms of number of document entries per posting.)
3. Inverting collection:
 - How much time did it take to index the collection?
 - How much memory is needed to index the collection?
 - How much total disk space is required for your index and related data structures? This total should include everything created in your indexing process that is needed later to answer queries (lexicon, inverted file, document location table, etc).

Phase II

For Phase II, you will write a program which will accept queries from the user and search for documents using the data structures produced in Phase I. Implement the inverted search algorithm using the vector space model to rank the documents, carefully choosing the weighting function.

Your search interface must allow the user to:

1. Input a search query. The query should be a free-text or keyword-based query.
2. View a list of search results. Each result should display an internal document ID (sequence number), a title or URL, and possibly a snippet of text from the document.
3. Choose a document from the list to view. This should fetch the document from where it is stored and display it to the user. In other words, implement the basic, single-query-and-result-list search process. Beyond these central requirements, you are free to make design and interface decisions as you see fit.

Milestones for Phase II

1. Implement the vector space model and inverted file based search algorithm.
2. Take a query interactively and display ranked results. Allow the user to select documents to read from the list.
3. Evaluate your search engine in terms of standard metrics such as precision and recall.

Benchmarks for Phase II

- Process a user query. Take a (single or multi-word) query interactively from the user, retrieve the top 100 documents from the collection, show the top 20 document identifiers to the user with scores, and let the user choose one to display. Report the amount of time needed for the entire interaction, from after the user first enters their query to when they can see a full document on the screen.

Now repeat this for a number of queries tabulating the queries used and the (wall-clock) time it took to search the collection, display the top 20 results, allow the user to select one document, and display the selected document.

- Evaluate the search engine. For each test query, provide a table showing precision, recall, F-measure, etc. and/or a variant of precision-recall graph.

Deliverables

You will turn in the following *four* items for each phase in the form of zip-archives: `PhaseI.zip` and `PhaseII.zip`.

Design document Describe your application's architecture and major data structures. Describe how your system implements the indexing and search operations (i.e., give pseudo-code for your search algorithm). List any external libraries or resources required by your application.

You are encouraged to write the design document prior to writing the code for planning purposes, although it will certainly change during the course of the project. Aim to make the design decisions explicit enough so that one of your classmates could implement your program from it (e.g., What is a token? What is the format of stored structures?).

Benchmark output The output of your program for each step in the benchmark. The specific output required is discussed in each benchmark.

Code and accompanying documentation Your code needs to be well documented with comments. The comments should not literally verbalize what the code is doing, instead, it should provide a high-level, abstract summary of what the code is supposed to accomplish, listing aspects such as tacit assumptions, invariants, etc.

You also need to include a `README.txt` file that describes how to compile your program and run it on datasets.

To turn in your solution in `*.zip` with the accompanying `README*.txt`, run the following command on `unixapps1.wright.edu`:

```
/common/public/tkprasad/cs707/turnin-pa1 *.zip README*.txt
```

You are also expected to demo your program to me.

Implementation

You may code your project in any programming language (such as C++, C#, Java or Python). You can choose any standard document collection to test your program but your code should be adaptable easily to be run on a new test collection for scalability testing.