Sep 2001

IR Course

Search Engine - Project Summary

Final assignment for course: Web based IR techniques/TAU 2001
By Prof. Eitan Ruppin

Submitted By:

Anatoli Uchitel
Tommer Leyvnad

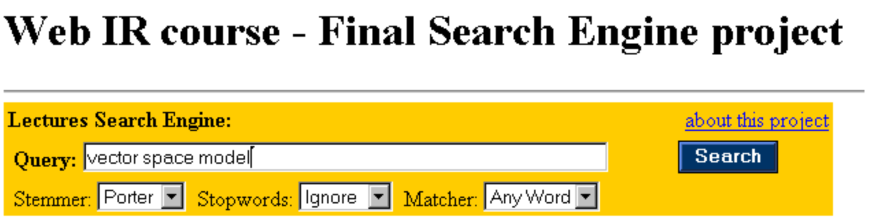
1. Purpose & Preface

This document describes our final work in the course "Web IR techniques". We've implemented a Search Engine (SE) with re-ranking that uses both stemming and stop-words.

This document first describes the algorithms that we've used along with the data-structures it operates on. We then explain the high-level design of our search engine. The appendixes contain rather technical data for compiling, running etc.

- We've implemented a web-interface for our Search Engine and we recommend you check it

Image 8.1: Online search engine interface at http://www.math.tau.ac.il/~tommer/web_se/se.html



Web IR course - Final Search Engine project

Lectures Search Engine: about this project

Query: vector space model

Search

Stemmer: Porter Stopwords: Ignore Matcher: Any Word

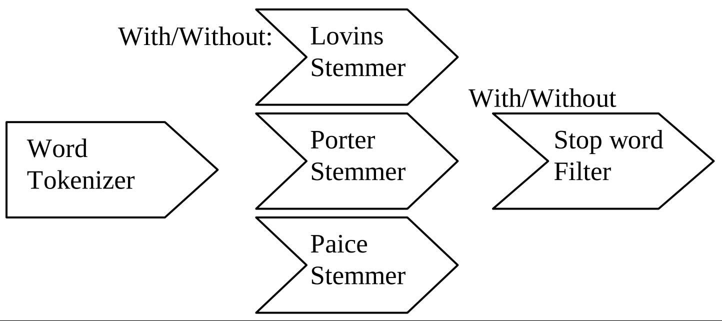
2. Algorithms & Data Structures

2.1. Algorithms

2.1.1. Tokenization

The process of parsing input text into terms is called tokenization. We've implemented our tokenizers in such a way that they can be combined in different orders to generate a tokenizers stream (figure 2.1).

Figure 2.1: Tokenizers streaming



The different tokenizers and their functionality are discussed in <ADD>.

2.1.1.1. Tokenization configuration

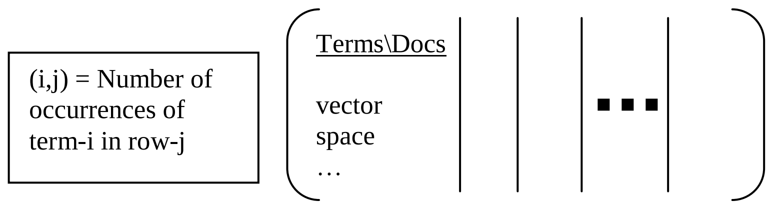
We call the tokenizers stream the configuration of our engine. There are 8 possible configurations:

- 4 (3 stemmers or no-stemming) x 2 (with or without stop-word filtering)

2.1.2. Indexing

Our SE uses a terms-by-documents table to represent indexed documents where each cell contains the number of occurrences of the appropriate term in the appropriate document (figure 2.2).

Figure 2.2: Terms by documents table



We've decided to store this information in order to separate the terms-by-documents table from the re-ranking algorithm. While re-ranking, the row occurrences frequencies written will be used to calculate the various re-ranking scores.

2.1.3. Querying

Querying, given some query string, is the process of finding the best matching previously indexed documents.

This process is separated into two:

- First: Match documents to the query string
- Second: Order the matched documents in a relevancy order (re-ranking)

2.1.3.1. Matching

Matching is the process of generating the list of documents that match the query terms. We've implemented two types of matchers:

- Match any query token – Documents that contain **any** term from the query are included in the matched list
- Match all query tokens – Only documents that contain **all** the query tokens are included in the matched list

Given the terms-by-documents table, implementing both matchers is easy and is not further discussed.

2.1.3.2. Re-ranking

Our re-ranking technique is based on TF-IDF. Given some matched document *d*, weight some term *t* in this document with respect to the frequency of *t* in this document versus the frequency of *t* in all documents.

The exact formula: $$weight(i,j) = \begin{cases} (1+\log(tf_{i,j}))\log\dfrac{N}{df_j} & if : tf_{i,j} \geq 1 \\ 0 & if : tf_{i,j} = 0 \end{cases}$$

Where $tf_{i,j}$ is the document frequency for term *i* in document *j*, meaning the value in the terms-by-document table at (i,j)

Notice that a term that occurred only in 1 document is given its full logarithmic frequency weight whereas a term that appears in all documents gets a zero weight.

We generate such a weight vector for each of the matched documents. The re-ranking score between these vectors is calculated as the cosine between each such vector and the weighted query vector. Documents are then re-ranked according to that score in descending order.

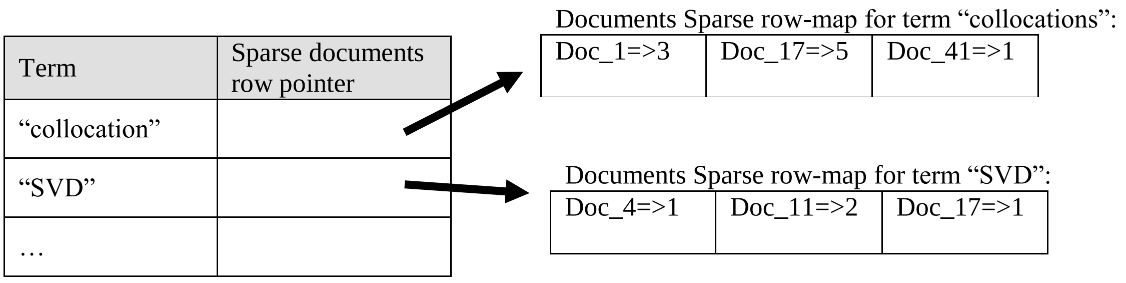
2.2. Data Structures

2.2.1. Vectors and Matrices

To represent the various sparse vectors needed for both the query, weighted terms (for re-ranking) and term-by-documents table we've used maps. Instead of saving frequency/value 0 at non-existing terms we simply store a mapping from the documents that do contain the term to their frequency/value.

The term-by-documents table itself is also a map from the various terms to the sparse vector map. This technique simplifies indexing new documents into the terms-by-documents table since we don't need to reallocate the matrix with a different size but rather simply add the document to the sparse row maps for existing table terms and add new terms to the global terms by document map itself (figure 2.3).

Figure 2.3: Term-by-documents tables implemented as map of sparse row maps



Term	Sparse documents row pointer
"collocation"	
"SVD"	
…	

Documents Sparse row-map for term "collocations":

Doc_1=>3	Doc_17=>5	Doc_41=>1

Documents Sparse row-map for term "SVD":

Doc_4=>1	Doc_11=>2	Doc_17=>1

The only major difference is that generating the column document vector (terms distribution within some document needed for generating the score re-ranking vector) is slightly more complex, though it has the same O(n) computational complexity (assuming the map/hash is O(1)) as the regular two-dimensional array matrix implementation.

2.2.2. Stored disk collections

We define a collection of indexed documents to be all the necessary data needed for successfully querying, indexing new documents and re-indexing existing documents.

In our implementation this translates to:

a. The configuration used for this collection (the tokenizers stream)

b. The mapping from actual document URI to its document ID[1]

c. The terms-by-document table

The mapping (b) is stored simply by storing its elements whereas the terms-by-documents table is stored by its individual rows: first the row's term (e.g., "collocation") followed by the sparse row vector as a list of pairs (document ID and the term's frequency). The configuration is encoded in the collection filename[2].

We've decided to store the collections on disk in a plain text format for educational purposes (it easily shows the terms-by-document map).

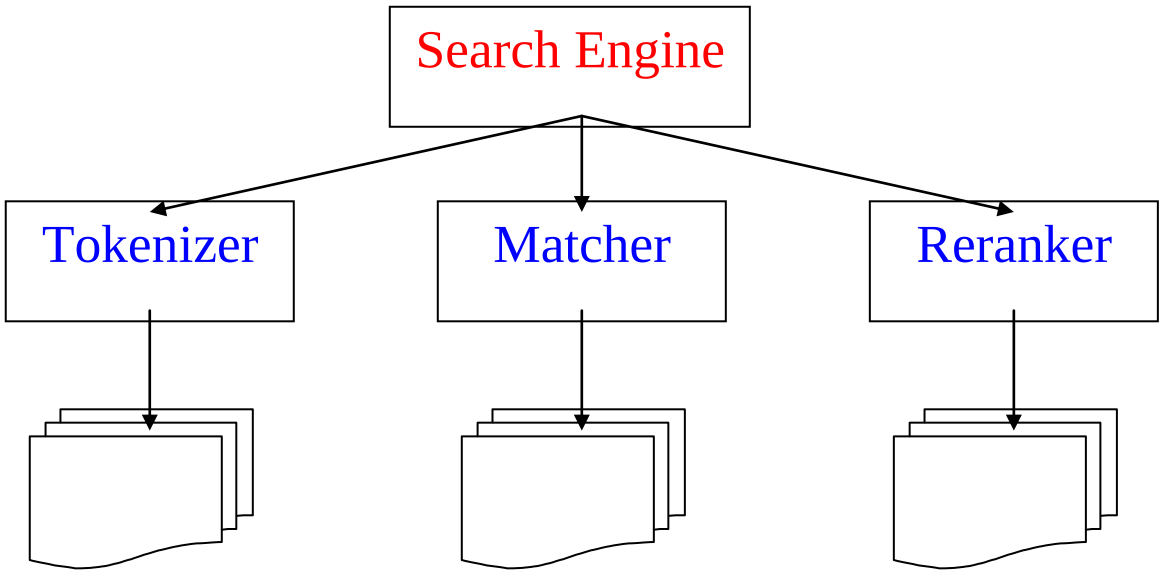
[1] This mapping translated a given document ID used in the terms-by-document table to the original input URI, thus enabling us to be more compact in the terms-by-documents table (saving integers instead of URI strings).

[2] For example: The collection *collection_porter_stopwords.col* was created and is used when using a porter-stemmer combined with stop-words filtering

3. High Level design

This project was designed in the object oriented methodology and implemented in C++.

3.1. The class diagram:



3.2. Concrete implementation classes

The search engine chooses by the command line parameters the implementation of the following super abstract classes.

3.2.1. Tokenizer

This class implementations, implement the token readers, which read both the documents to be indexed and the query string to produce terms.
Available implementations are:

3.2.1.1. WordTokenizer

The simplest tokenizer, which retrieves terms (words) as they are, considering most of the punctuation marks and symbols as white-space.

3.2.1.2. *StopWordTokenizer*

Retrieves terms as they are using an inner tokenizer, filtering out stop words.

3.2.1.3. *LovinsWordTokenizer*

*R*etrieves terms and performs stemming using Lovins'stemming algorithm.

3.2.1.4. PorterWordTokenizer

retrieves terms and performs stemming using Porter's stemming algorithm.

3.2.1.5. PaiceTokenizer

retrieves terms and performs stemming using Paice's stemming algorithm.

3.2.2. Matcher

This class implementations, implement the method in which the matching is performed.
Available implementations are:

3.2.2.1. AnyMatcher

*C*onsiders a successful match if at least one term from the query appears in the document.

3.2.2.2. *AllMatcher*

*C*onsiders a successful match only if all the terms from the query appear in the document.

3.2.2.3. *Reranker*

This class implementations, implement the method by which the matching documents found by the Matcher class are reranked so the best document gets the highest score.
We have implemented the TF/IDF reranking technique, which is considered to be a good weightening scheme.

3.3. CGI implementation details

The CGI itself (written in Perl) simply extract the various configuration parameters and invokes the html-output version of the search engine executable.

3.4. Implementation summary

Our implementation is highly documented and can easily be understood. We've included in appendix D a short description of all the implementation files.

This implementation can be easily extended to add further implementations to these abstract classes , and can be seen as a good basis for further implementations , beside of it being a working complete search engine .

A. Appendix A – Using the Search Engine

A.1 Running from the command line (in brief)

Usage: SE [options] file1 file2 ...

- **file1 file2... : The files to index (may be empty in order not to index new files)**
- **Options**:

  -q *queryString: Run the supplied query on the collection*

  -s *porter/paice/lovins/none: Use the given stemmer (default is porter(*

  -m *all/any: Match any query token (default) or all query tokens*

  -n *: Do not ignore stopwords (default is to ignore stopword)*

A.1.1 Behind the scenes

For every combination (stemmer type and stopwords option = 4x2) there is a collection file which name advices the combinatio (as discussed in 2.1.1.1).
These collection files are appended by indexing new documents. Specifying existing documents will re-index them.
Whenever a query is searched it should be specified using the -q flag (to include multiple words query in you should use quotes/double-quotes).
Note: We allow both querying and indexing at the same command-line run (indexing is done first).

A.1.2 Examples:

SE lecture1.txt lecture2.txt
- This will index the specified files in the collection

SE -q 'IR LF IDF'
- This will query the collection for IR and/or LF and/or IDF

SE -s none -n -m any -q 'vector space model'
- This will query the no-stemmer no-stopword collection for documents with all the query terms (i.e, vector & space & model)

A.2. Using the Web Interface

As mentioned before, we've created a web-interface for our project using a CGI. The web interface allows online querying against all possible collections that contain the lecture summaries of all student groups.

To access the web interface go to http://www.math.tau.ac.il/~tommer/web_se/se.html.

B. Appendix B – Compiling the Search Engine

Our implementation was compiles on both Windows and Linux platforms.

In order to support the use of this engine inside a Web CGI we've created two sets of output routines: regular console output and html format output. The actual output desired is controlled at compile-time by defining/not-defining the HTML_OUT flag.

B.1. Compiling on Windows

We support compiling with MS-VC6. To compile simply open the supplied project (.dsp) file and build the project. Notice that the default output is regular console output.

B.2. Compiling on Linux

We support compiling on Linux using g++[3]. To compile a console output version simply run make. To compile HTML output version run "make HTMLOUT=1".

[3] It might also compile on other Unix platforms using either g++ or the native C++ compiler. Notice that we haven't tested this.

C. Appendix C - Work division

C.1. Lecture Summaries

Written by both Tommer & Anatoli:

- Tommer: lecture_01-05-01.txt, lecture_15-05-01.txt, lecture_17-04-01.txt, lecture_20-3-01.txt, lecture_24-04-01.txt, lecture_27-03-01.txt, text-summarization.txt
- Anatoli: Link analysis.txt, handle browsing.txt, important notes.txt, lecture030401.txt, lecture080501.txt, lecture130301.txt, lecture290501.txt, svm2.txt

C.2. Source code

- **Search Engine**

  Written by Tommer

- **Tokenizer**

  WordTokenizer – Written by Tommer

  StopWordTokenizer , PorterWordTokenizer , LovinsWordTokenizer , PaiceWordTokenizer – Written by Anatoli

- **Matcher**

  AnyMatcher – Written by Anatoli

  AllMatcher – Written by Tommer

- **Reranker**

  Written by Tommer

- **Main and Web Interface**

  Written by Tommer

- **Indexing Script**

  Written by Anatoli

C.3. Documentation

Includes this document, howto.txt file, students.txt file Written by both Tommer and Anatoli.

D.Appendix D – Source files

This appendix first lists the various implementation files and their functionality. A complete source-code printout then follows.

D.1. File listing

Directory/Filename	Brief Description
./	Main directory contains the main file and various building files
./se	Linux SE binary
./main.cpp	The main function implementation file (responsible for parsing the configuration and initializing the search engine)
./stopwords.txt	The stopwords list
./SE.dsp	The MSVC6 project file for compiling on Windows platform
./Makefile	The makefile for compiling on Linux platform
./SE Project Summary.doc	This documentation file
./index-sub-tree	A shell script that indexes a specified directory hierarchy using find and xargs for all possible configurations
./howto.txt, students.txt	The required text files
Common/	This directory contains general, project wide, source files
Common/getopt.c, h	GNU's getopt implementation for command line argument parsing (included since it's not available on windows)
Common/SEGlobal.h	Global includes and typedefs. This also defines our stl-based Vector, Map containers (these containers can be easily printed using >> operator)
Tokenizers/	This directory contains the tokenizers base abstract class definition and the actual concrete implementations
Tokenizers/Tokenizer.cpp, h	The tokenizers base-classes (both leaf and stream node)
Tokenizers/WordTokenizer.cpp, h	The basic (leaf) word tokenizer
Tokenizers/StopWordTokenizer.cpp, h	The stop word filter (stream node)
Tokenizers/…	The remaining files are the stream-node wrappers to the downloaded stemmers (each downloaded implementation is in its own sub-directory)
Matchers/	This directory contains the matchers base-class with its supported concrete implementations
Matchers/Matcher.h	The Matcher base-class definition
Matchers/Any(All)Matcher.cpp, h	The implementations for the two support matchers
Rerankers/	This directory contains the re-rankers base-class with its supported concrete implementation
Rerankers/Reranker.h	The Reranker base-class definition
Rerankers/TFIDFReranker.cpp, h	The implementation of our TF.IDF re-reranker
Engine	This directory contains the Search Engine files
Engine/SearchEngine.cpp, h	The Search Engine implementation files
stlfix/	A directory the holds the sstream file missing from g++ std-library

D.2. Source Files