INLS 490-154: Information Retrieval Systems Design and Implementation. Spring 2009.

3. Learning to index

Chirag Shah*
School of Information & Library Science (SILS)
UNC Chapel Hill NC 27599
chirag@unc.edu

1 Introduction

Last time we started working with the text file and saw a toy-example that allowed us to look for a word in a text document. This time we will start making that simple process more sophisticated. At this point it is useful to revisit our model for IR shown in Figure 1. In this class we will focus on the part highlighted with a box, that is, the part that deals with representing textual information. This is really a critical part since it determines how the rest of the model for IR will work. The objective of this part is to represent a collection of documents in such a way that the retrieval process works as effectively and efficiently as possible. This process of *adequately* and *efficiently* representing textual information is called indexing.

Indexing typically involves the following stages.

1. Information collection
2. Tokenization
3. Stop words removal
4. Stemming
5. Storage

Not all of these steps are mandatory and their order could be changed depending on the design. In this class we will work through all these stages to index a small set of documents.

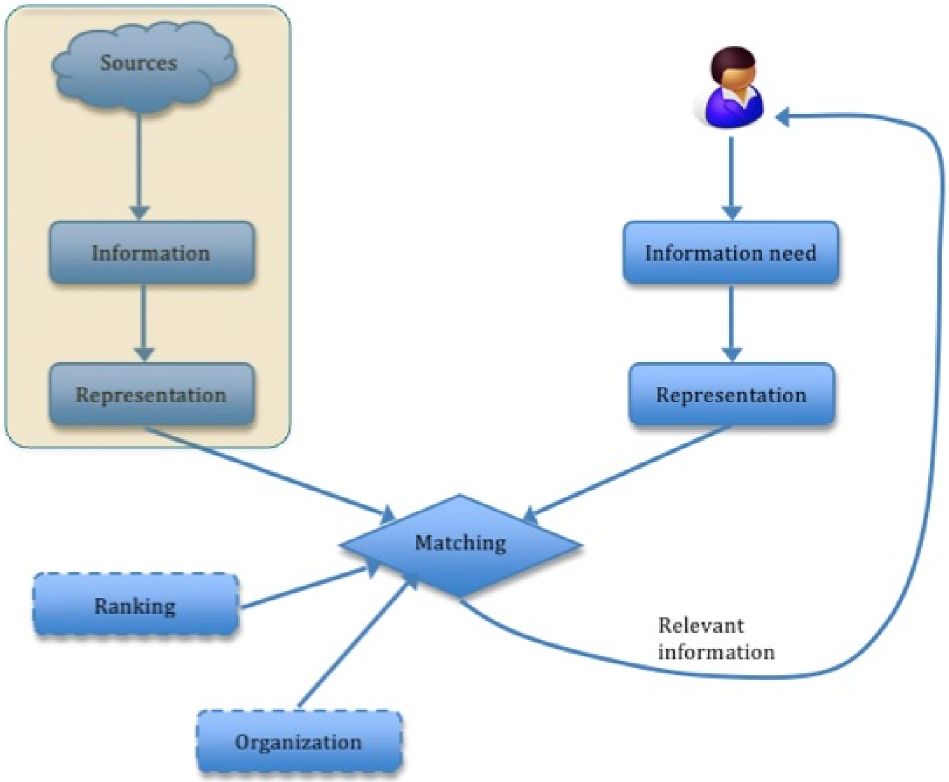

Figure 1: A model of Information Retrieval (IR)

2 Recall and precision

Before we talk about other concepts relating to indexing and work on building various subcomponents, it is important to learn about two ideas that are central to IR: recall and precision. In Figure 2, a van diagram is given showing a set of relevant documents (R) for an information need, and a set of retrieved documents (R') by some IR system. Recall is the portion of relevant documents returned, and precision is the portion of the returned document that is relevant. Using Figure 2, this can be formulated as

$$\text{Recall} = \frac{R \cap R'}{R} \tag{1}$$

$$\text{Precision} = \frac{R \cap R'}{R'} \tag{2}$$

In practice, it is usually found that improving one of these harms the other one. But this is an empirical observation, and not a mathematically proven theory (sounds like a nice research challenge!).

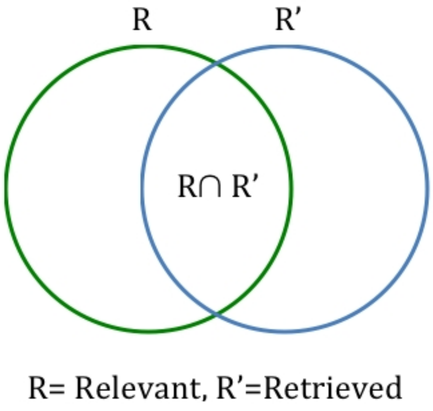

Figure 2: A model to understand recall and precision in IR

3 Stop words removal and stemming

For now, we will assume that you could easily collect some text documents manually. That completes our first stage of information representation. Once you have a set of these documents, let us put them all in one file while preserving the information about which text came from which document. The following Perl script (Code 1) combines a set of text documents and create a single file with SGML markup (TREC format).

Code 1: Perl code to combine a set of text documents in TREC format

```
# Usage: ./combineDocs.pl <input_file>
# <input_file> has the list of files, one name per line.
#!/usr/bin/perl

open(FILELIST, $ARGV[0]) || die "Error reading the input file.\n";
open(OUTFILE, ">documents.txt") || die "Error creating the output file.\n";

$docNo = 1;
while ($docName = <FILELIST>)
{
    print OUTFILE "<DOC>\n<DOCNO>$docNo</DOCNO>\n<TEXT>\n";
    open (INFILE, $docName);
    while ($line = <INFILE>)
    {
        print OUTFILE $line;
    }
    close(INFILE);
    print OUTFILE "</TEXT>\n</DOC>\n";
    $docNo++;
}
```

```
close(FILELIST);
close(OUTFILE);
```

Now we move on to the second stage, which involves preparing tokens. A token in IR could be anything - a word, a sentence, a paragraph, or even a document. For a keywords based search, we probably want to have words or phrases as tokens. For now, let us safely assume that our token (the smallest unit for processing) is word. Code 2 is a simple Perl script that breaks the input text document into one work per line.

Code 2: A tokenizer in Perl

```
# Usage: ./tokenize.pl < <input_file>
#!/usr/bin/perl

while ($line = <STDIN>)
{
    chop($line);
    @entry = split(/ /, $line);
    $i = 0;
    while ($entry[$i])
    {
        print "$entry[$i]\n";
        $i++;
    }
}
```

Now it is time to remove the stop words. Code 3 shows a Perl script that processes an input file, going through line by line,[1] matching it with a list of stop words, and removing that word if it exists in the stop words list.[2]

Code 3: Perl code to remove stop words

```
# Usage: ./removeStopWords.pl <input_file> <stopwords_file>
#!/usr/bin/perl

# Read all the stop words
open (STOPWORDS, $ARGV[1]) || die "Error opening the stopwords file\n";
$count = 0;
while ($word = <STOPWORDS>)
{
    chop($word);
    $stopword[$count] = lc($word);
    $count++;
```

[1] The script does NOT assume only one word per line.

[2] You need to have this file ready, with one stop word per line.

```
}
close(STOPWORDS);

# Process the input file line by line
open (INFILE, $ARGV[0]) || die "Error opening the input file\n";
while ($line = <INFILE>)
{
    chop($line);
    @entry = split(/ /, $line);
    $i = 0;
    while ($entry[$i])
    {
        $found = 0;
        $j = 0;
        while (($j<=$count) && ($found==0))
        {
            if (lc($entry[$i]) eq $stopword[$j])
            {
                    $found = 1;
            }
            $j++;
        }
        if ($found == 0)
        {
            print "$entry[$i]\n";
        }
        $i++;
    }
}
close(INFILE);
```

At this point you may want to check the number of lines for both your original document (combined file of all the text documents) and the one generated by removing stop words. You should see the reduction and there is our first advantage of removing stop words - saving the storage space!

Let us continue to the following stage and perform stemming, which is a process of reducing a word to its original form. For example, stemming ‘going’ or ‘gone’ will result in ‘go’. Thus, all ‘go’, ‘going’, and ‘gone’ can be represented as just one token - ‘go’. For the most retrieval purposes, this is alright, but you can imagine where it is not.

At this point, you can obtain Porter’s stemmer[3] and stem the output generated by the above process. Try running `sort` and `uniq` commands on the input as well as the output of the stemmer. You should see further reduction in the index size since stemming reduces many words to their common stems.

[3] http://tartarus.org/ martin/PorterStemmer/

We could now go on and store the generated output in an efficient data structure, but we will stop here with a note that one of the most common structure for storage in such situations is inverted file structure. This structure can be represented as

```
<term> <document> <positions>
<term> <document> <positions>
.
.
.
```

As you can see, each line contains the information about a term's occurrence in a given document along with its positions. This information can be used not only for retrieval, but also in matching and ranking.

Before we finish this section, let us note down the advantages and disadvantages of stop words removal and stemming.

Removing stop words saves space while indexing, and saves processing while searching. However, if we are not careful removing these words, we may end up throwing out the information that may affect our retrieval performance. For instance, if we remove *who* considering a stop word, we may not able to retrieve that document for a query *WHO* (a music band), thus reducing the recall. On the other hand, removing stop words such as '1' and '2' may not index 'Spiderman' and 'Spiderman 2' as different terms. This will result in documents about 'Spiderman' showing up for 'Spiderman 2' queries too. The effect of this issue is the decrease in precision.

Stemming also carries similar pros and cons. Reducing inflated words to their stemmed form will save space and processing, but the retrieval performance may be affected. For instance, 'runners' would be reduced to 'run' due to stemming; however, a query about 'Runners Magazine' may bring documents that are about run, but not the Running Magazine. IR expert Avi Rappoport notes,[4]

> "In many cases, the search engine can't show exact matches first, because the stemmed version is stored in the index. This may save disk space, but it annoys users no end. This is particularly a problem with verbs...It's easy enough to perform the stemming in the query and search for multiple versions of the word, rather than storing the stemmed version only."

In addition to this, the meaningfulness of the stemmed word depends on the kind of stemmer used. For instance, Porter stemmer will reduce 'democracy' to 'democraci', which is a meaningless word. Given that an index is not to be used directly by the users, such an incident may be fine, but it may not be in many situations.

To overcome this last issue for stemming, Bob Krovetz developed a stemmer that stems using dictionary words. In this case, a stemmed word will be guaranteed to be a meaningful word, as it comes from a dictionary. Thus, Krovetz stemmer will provide human-readable stems, but the reduction in index size may not be as large as given by Porter's stemmer. Krovetz stemmer is available as a part of Lemur.

[4] http://forums.searchenginewatch.com/showthread.php?s=6316bc62facd940a19c197eda891022&t=258&page=2&pp=20

4 Indexing with Lemur

Lemur toolkit,[5] developed by UMass Amherst and CMU, is a open-source toolkit designed to support research in language modeling and information retrieval. As a part of this course, we will be using this toolkit quite a bit. It is important to get familiarized with it as early as possible in the course to reap the maximum benefits of this toolkit.

Once you have Lemur toolkit installed, let us take it for a spin. Find 'BuildIndex' application and see if you can run it from the command line. If you do not pass any parameters to it, BuildIndex will list the possible options. This is true with almost all the applications in Lemur. Another characteristic that these applications have in common is the way their parameter files are structured in XML format. Code 4 shows an parameter file for BuildIndex.

Code 4: A parameter file for BuildIndex

```
<parameters>
    <index>myindex</index>
    <indexType>indri</indexType>
    <dataFiles>files.list</dataFiles>
    <docFormat>trec</docFormat>
    <stemmer>porter</stemmer>
    <stopwords>stopwords.list</stopwords>
</parameters>
```

To build an index with Lemur, follow these steps.

1. Create a file containing a list of files to index. Let us call it 'files.list'. Each line in 'files.list' will be the name of a text file that we want to include in the index.

2. If you want to use stop words removal, make sure you have a list of stop words ready in a file (here, 'stopwords.list').

3. Prepare a parameter file. You can copy Code 4. Here, the name of the index is given as 'myindex' and the type of the index is 'indri' (details later). Porter stemmer is used. One can also specify `krovetz` here to use Krovetz stemmer. Let us save this parameter file as 'myindex.param'.

4. Run 'BuildIndex myindex.param'. If everything goes right, you should see a directory named 'myindex', which contains your index as well as the original collection for retrieval.

Now that we have created an index, let us look inside it. An application called 'dumpindex' in Lemur helps us to that. Following are some useful options to use with this application. Each of these options go after 'dumpindex myindex'.

[5] http://www.lemurproject.org/

- *s*. Prints some statistics about the collection.

- *v*. Prints the vocabulary of the index in the following format:
  `<term> <term_count> <doc_count>`

- *tp <term>*. Prints positions of term occurrence. The first line of the output indicates:
  `<term> <stem> <count> <total_count>`
  The following lines of the output are in this format:
  `<doc_no> <count> <positions>`

- *dt <doc_no>*. Prints the original text of a document.

5 Summary

- Good representation of the collected information is essential for building a good IR system. This process is called indexing for textual documents.

- Indexing process typically involves (1) getting the information in the right format, (2) tokenization, (3) stop words removal, (4) stemming, and (5) storage.

- Both stop words removal and stemming have their pros and cons. In general, performing them saves space and processing, but may compromise the retrieval effectiveness.

- Two of the most common measures of evaluating retrieval effectiveness are recall and precision.

- In Lemur, use 'BuildIndex' to prepare the index and 'dumpindex' to analyze it.