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

4. Query processing and retrieval

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

1 Introduction

Previously, we saw how we could go through the following stages of indexing on a small collection of text document.

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

We will continue our exploration in that direction with the help of Lemur Toolkit. In addition, we will look at how to process a query, and perform matching and retrieval.

2 Indexing revisited

The last time we saw how we could take a collection of documents and represent them in an index by processing the collection step-by-step. That process was cumbersome and inefficient. Fortunately for us, we do not have to do so much of "manual" work. Lemur has an application called *ParseToFile*, which accomplishes almost the same in a fraction of a second. The first argument to this application is the name of a parameter file (see Code 1), followed by input file name(s).

* These notes for INLS 490-154 Spring 2009 by Chirag Shah (http://www.unc.edu/∼chirags) are licensed under a Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License.

Code 1: Parameter file for ParseToFile application (parsing documents)

```
<parameters>
    <outputFile>parsed.txt</outputFile>
    <docFormat>trec</docFormat>
    <stopwords>stopwords.list</stopwords>
    <stemmer>porter</stemmer>
</parameters>
```

The output of *ParseToFile* should contain one word per line, and the information about the documents in which those words occur. In addition, stop words may have been removed and the rest of the words may be stemmed depending up on the parameters. Compare these results to those obtained by the previous processes.

Here, of course, our ultimate goal is to index a document and therefore, the missing last step is actually storing the collection in a reasonable format. *BuildIndex* application in Lemur does just that. *BuildIndex* does all that *ParseToFile* does, plus converting that parsed file to an index format. See Code 2 for a sample parameter file for *BuildIndex*.

Code 2: Parameter file for BuildIndex application

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

In the above example, we chose *indri* to be the type of index and called our index *myindex*. Thus, *BuildIndex* will create a directory named 'myindex' with subdirectories 'collection' and 'index'. The last time we also saw how to analyze this index using *dumpindex*. We will skip those details as they were already covered and move on to processing the queries.

3 Processing queries with Lemur

Let us now move to the user side of IR. The user expresses his information need in terms of a query. Before we process this query for matching and retrieval, we need to parse it and represent it in a suitable format. Code 3 shows a sample query file with three queries. You would notice that the queries are written as documents in TREC format. This means we could use *ParseToFile* application to parse this query file. A sample parameter file is shown in Code 4. Run *ParseToFile* with the name of this parameter file as the first argument, and the name of the query file (Code 3) as the second argument. The output should be in 'query_parsed.txt' file.

Code 3: Query file

```
<DOC>
<DOCNO>1</DOCNO>
<TEXT>
columbus islands
</TEXT>
</DOC>
<DOC>
<DOCNO>2</DOCNO>
<TEXT>
british colony
</TEXT>
</DOC>
<DOC>
<DOCNO>3</DOCNO>
<TEXT>
independance
</TEXT>
</DOC>
```

Code 4: Parameter file for ParseToFile application (parsing queries)

```
<parameters>
    <outputFile>query_parsed.txt</outputFile>
    <docFormat>trec</docFormat>
    <stopwords>stopwords.list</stopwords>
    <stemmer>porter</stemmer>
</parameters>
```

4 Retrieval with Lemur

Having represented a collection of documents in an index, and parsed the input queries, we are now ready to perform actual matching and retrieval. We will use an application from Lemur, called *RetEval*. A sample parameter file for this application is shown in Code 5. Here, `<index>` is the name of the index, `<textQuery>` is the file that has parsed queries stored, `<resultFile>` is where you want the output of the retrieval be stored, `<resultFormat>` indicates how you want your retrieval results be formatted, `<resultCount>` tells Lemur up to how many documents you want to retrieve for a given query, and `<retModel>` indicates the retrieval model to use, which in this case is 0=TFIDF.

Code 5: Parameter file for RetEval application

```
<parameters>
    <index>myindex</index>
    <textQuery>query_parsed.txt</textQuery>
    <resultFile>results.txt</resultFile>
    <resultFormat>3col</resultFormat>
    <resultCount>5</resultCount>
    <retModel>0</retModel>
</parameters>
```

Since we chose `3col` as the output format, we would see the output formatted in this way:

```
<query_id> <doc_id> <score>
```

Here, the score indicates the similarity score between the query (as given by `<query_id>`) and the document (as given by `<doc_id>`). The actual values of these scores are probably not that useful, but their relative values are, since they allow us to rank the retrieved list of documents. The actual scores also vary based on the retrieval model chosen. More details on the calculation of such scores are given in the next section.

It should also be noted that the maximum results (here, 5) applies to *each* query. Since we used three queries here to be run, our total results could be up to 15.

5 Vector space model

There are several models of retrieval, but none is as popular as the vector space model in IR. And it is not only in IR, but several other fields where this model is widely used. The basic concept of the model is really simple. It is based on a key idea that a piece of information (document or query) is a vector in high-dimensional space.

Let us understand this by a toy example. Imagine our vocabulary is {dog,cat}, mere two words. This means we are working on a two-dimensional space when it comes to the vector space model. These two dimensions are shown in Figure 1. Now, let us assume we have one document with five occurrences of 'dog', two occurrences of 'cat', and nothing else. In the space given, this vector $\{5, 2\}$ is represented in Figure 1 as V_d. And let us assume that we have a query with one occurrence of 'dog' and one occurrence of 'cat. This vector $\{1, 1\}$ is represented in Figure 1 as V_q.

In order to evaluate the matching of these two vectors, we need to see how "close" they are in the given space. One way of doing this is by looking at the angle θ between these two vectors. This brings us to the most common way of finding similarity among two vectors in the vector space model - cosine similarity. The function for this similarity is defined in Equation (1).

$$Sim(V_d, V_q) = cosine\theta = \frac{V_d V_q}{|V_d||V_q|} \tag{1}$$

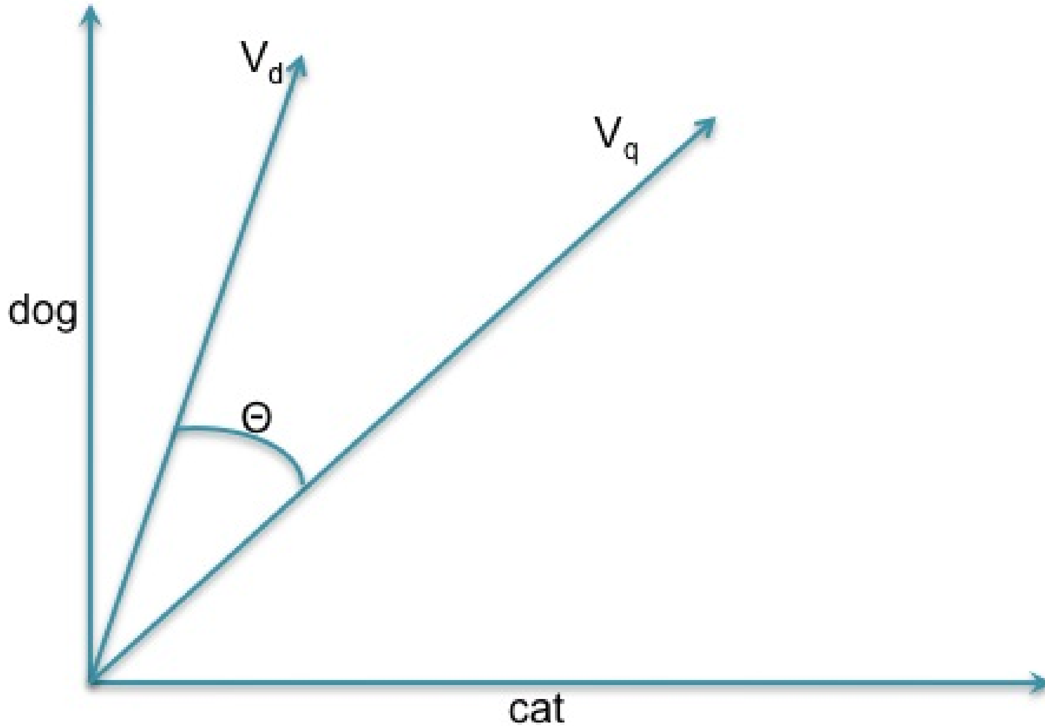

Figure 1: A two-dimensional vector space

Plugging in the values $V_d = \{5, 2\}$ and $V_q = \{1, 1\}$ in this equation, we get 0.92 as the similarity score. Since cosine scores range from 0 to 1, we can see that we have a very high matching score here.

Now let us talk about those values in the vector. For the above example, we simply used the frequencies of the terms as their *weights*, but there are better ways of weighing these dimensions. One popular way is TFIDF weighing. TFIDF is based on two simple intuitions:

1. The more the word occurs in a given document, the more important it is.
2. The less the word is common in the collection, the more important it is.

The first intuition is translated in the following formulation.

$$TF(t_i, d_j) = \frac{freq(t_i, d_j)}{\sum_k freq(t_i, d_j)} \tag{2}$$

Here, $freq(t_i, d_j)$ is the frequency of term t_i in document d_j, and the denominator calculates the size of that document. Thus, the frequency of a term is normalized by the size of a document.

The second intuition is translated can be translated as - the importance of a term is inversely proportional to its frequency in the collection. This formulation of Inverse Document Frequency (IDF) is given below.

$$IDF(t_i) = log\frac{|D|}{|d_j : t_i \in d_j|} \tag{3}$$

Here, $|D|$ is the total number of documents and $|d_j : T_i \in d_j|$ is the number of documents in which term t_i occurs.

Finally, we can combined these two calculations in to one score by multiplying them as shown below.

$$TFIDF(t_i, d_j) = TF(t_i, d_j)IDF(t_i) \tag{4}$$

6 Summary

- One way of looking at information retrieval is the matching of information to the information need.
- For the unstructured textual information domain, information is in the form of a collection of text documents, information need in terms of a keywords-based query, and the results are a set of text documents.
- Vector space model is the most popular model for IR, TFIDF is the most common scheme for weighing the terms, and cosine similarity is the most prevalent measure for computing similarity between two vectors.