IMPLEMENTING A VECTOR SPACE DOCUMENT RETRIEVAL SYSTEM

Mark A. Greenwood

Dept. of Computer Science, University of Sheffield
Regents Court, 211 Portobello St, Sheffield. S1 4DP. UK

m.greenwood@dcs.shef.ac.uk

Electronic copy and resources available from
http://www.dcs.shef.ac.uk/~mark/ follow the PhD link

ABSTRACT

This paper describes the implementation of a vector space document retrieval system and the evaluation of this system using the CACM document collection. The system is evaluated using both traditional techniques i.e. precision-recall curves, and more modern techniques which are founded in the way people use Internet search engines. Both evaluation methods clearly show that retrieval systems, which make use of stemming and stop word removal, are superior to those that do not. Also, the TF×IDF algorithm of ranking documents is more successful than simply using Term Frequency. A flaw in the original implementation of the system, which leads to some query words being counted multiple times, is also evaluated with promising results. This shows that the vector space model in its traditional form is not necessarily the best document retrieval technique currently available.

1. INTRODUCTION

With the rapid growth in the availability of electronic documents, vast amounts of work have gone into the development of document retrieval systems. Originally these systems were developed to work within libraries, where the document collection (i.e. books, journal articles, etc) was relatively small. Over time, however, our information needs have changed, to the point where we frequently want to retrieve relevant documents from the World Wide Web - clearly the two situations are considerably different.

The problem of retrieving relevant documents from a large document collection can be approached using many different algorithms. The three classic models used in information retrieval (under which all these algorithms can be loosely grouped) are called Boolean, vector and probabilistic [1]. In a similar fashion there are a variety of ways in which to evaluate the effectiveness of a document retrieval system; historically these have been recall and precision, although these measures of effectiveness don't seem to be quite as relevant in today's world of the Internet [3].

This paper describes a specific implementation of an information retrieval system that is based upon the vector space model. The final system is highly configurable to allow us to observe how changes in the model affect the relevance of the retrieved documents. Due to an error in the original implementation an interesting configuration of the system is available, although this configuration means that the retrieval algorithm is no longer a vector space model in the traditional sense.

The retrieval systems are implemented in Perl, so that use can be made of the built in regular expressions which make parsing documents (especially those such as the CACM collection) very simple. For more information on Perl see [6].

For more information on information retrieval models and algorithms see [1] and [2].

2. DOCUMENT COLLECTIONS

Many different document collections are available for use in this kind of research. Unfortunately, by far the best way of evaluating any information retrieval system is to use a document collection which contains not only documents but also queries and relevance judgements. Preparing a set of queries and relevance judgements is a time consuming task and therefore these collections are often relatively small and much harder to find.

Two such collections do exist; the CACM collection and the Cystic Fibrosis Collection [5]. These are, however, both very small (consisting of a few thousand documents), compared to modern text collections such as those used at TREC (at the TREC-9 conference the document collection for the Question Answering track consisted of 979,000 documents [4]).

It was decided that since the CACM collection has been used for evaluating IR systems for a long time, then this study would continue the tradition and hence any results generated from this study would be comparable to previous work carried out using the collection. The collection consists of 3204 documents and is supplied with 64 natural language queries and the judgements for each (i.e. which documents are relevant to each query – interestingly there are 12 queries which have no relevant documents in the collection).

3. INDEXING A DOCUMENT COLLECTION

3.1. What to Index

Each document in the CACM collection is clearly split into numerous sections, including: title, author, abstract, keywords and citation. It was decided for this study to index the title, author and abstract (where available), the keywords were not indexed as these seemed, in most instances, to add no new words to the index, but would clearly increase the time required to process the document collection.

3.2. Data Structure

As was stated in the introduction, the IR system is implemented in Perl, and so extensive use of the hash data type was made to store and manipulate the index. Figure 3-1 shows the data-structure that is created by the indexer as it processes the documents in the collection.

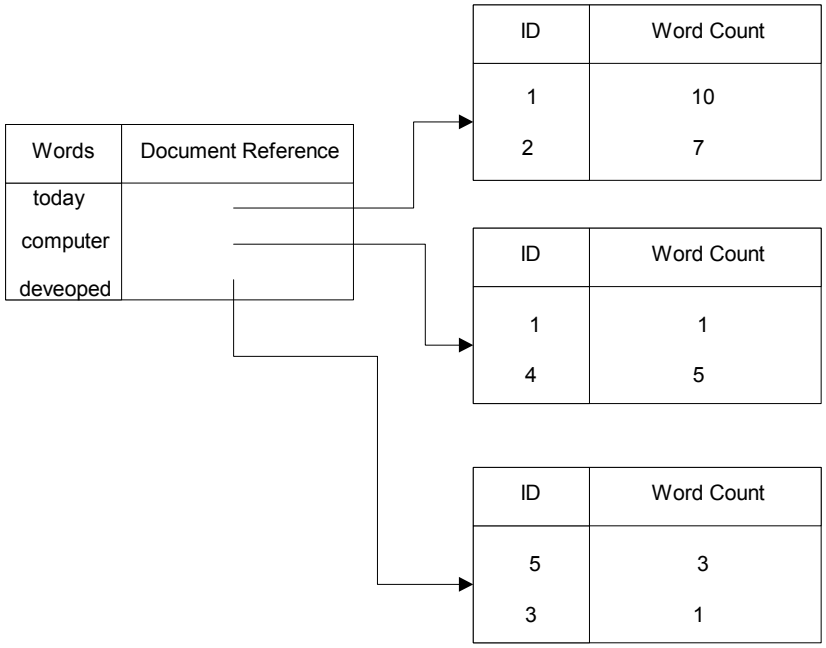

Figure 3-1: Data structure constructed by the indexer

This data structure is not only easy to construct during indexing, but also, as will be made clear later, it is beneficial to the implementation of the querying software. This type of data structure is known as a word-level inverted index [2].

The index is built using code which implements the following pseudo algorithm:

```
Read in a line at a time from the file containing the document collection, until an
entire document has been read in.
Split the document into tokens; remove any stop words present (if the user has
requested) and stem the resulting tokens (if the user has requested).
For each token in the document
        Get a reference to the hash of documents - word counts for this word
        If the hash exists then this word has been seen before so
                Get the word count for the current document, for this word
                If this exists
                        add one to it and put it back in the hash
                else
                        set the count to 1 and store it against this document in the
                        document – word count hash
        else
                store a reference to an anonymous hash containing this document and a
                count of 1 against this word in the index
Store the length of the document in terms as we will need this as well
Repeat until there are no more documents
```

Pseudo Code 3-1: Indexing Algorithm

As you can see, this algorithm clearly builds up the data structure described above. It does this by storing a reference to a hash of documents and word counts within a hash which is indexed by word.

3.3. Configuring the Indexer

The indexer can be configured using two different command line switches. Under standard operation the indexer stores every word in the index as it is found in the document. Clearly, some of the words in a document are irrelevant when you are retrieving information - such as the words: the, a, in, as, of, etc. These words (known as stop words) can be removed from the document before indexing so as to reduce the size of the index and also the time required to query it. The CACM collection comes with a supplied stop word list. This implementation will make use of this list if the `-w` command line switch is supplied to the indexer.

The other option for configuring the index involves stemming the words before indexing them. A Perl implementation of the Porter Stemmer is used for this. Clearly, a stemmer is never going to work 100% reliably, but this is not an issue, as long as all the words in the collection and the queries are stemmed using the same stemmer. So to use stemming when producing the index the program can be given the `-s` command line argument.

Clearly both removing stop words and stemming can be combined when producing an index, which means that there are four different combinations of index which can be generated (standard, stop words removed, words stemmed, stop words removed and words stemmed), and all four are generated and used in this study.

More details about command line switches and how to run the indexer can be found in the `README.TXT` file supplied with the system.

4. QUERYING A DOCUMENT COLLECTION

4.1. The Vector Space Model

The vector space model for document retrieval is based, upon building an *n* dimensional vector for the query and each document in the collection. The full non-optimised model causes *n* to be equal to the number of words in the language, whereas in reality *n* is usually equal to the number of different words in the document collection (words in the query are not important, because if they don't appear in the document then they are of no use for retrieval).

The following is an example vector for a document (taken from page 182 of [2]), in a language that only contains 10 words (i.e. there are only ten unique words in the collection):

$$D_1 = (1, 0, 0, 1, 0, 0, 0, 2, 2, 0)$$

The elements of the vectors are the frequency of occurrence of the word within the document; for example word 1 appears once in the document, whereas word 9 appears twice. The query vector is slightly different in that the presence, or not, of a word is indicated by a 1 or 0 respectively, counts of the words are not recorded.

Finding relevant documents using this model, involves generating a vector for the query and each document in the collection and then ranking the documents by similarity to the query. This is usually expressed as the difference, θ, in direction of the query vector and a document vector, as shown in the figure below.

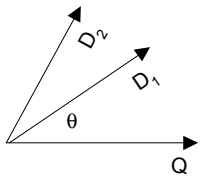

Figure 4-1: Difference in direction of two document vectors with a query vector

This is usually stated as the cosine rule, as in Equation 4-1 (see [2] for a fuller explanation of this equation).

$$cosine(Q, D_d) = \frac{1}{W_d W_q} \sum_{t \in Q \cap D_d} \left(1 + \log_e f_{d,t}\right) \cdot \log_e \left(1 + \frac{N}{f_t}\right)$$

Equation 4-1: Cosine rule for the Vector Space model

The above equation contains many variables, some of which may not be initially clear, and are described in Table 4-1. Equation 4-1is rarely implemented in full for reasons of efficiency. Section 4.3 details how this study implements the vector space model.

Variable	*Description*
Q	The vector representing the query
D_d	The vector representing the document, d
W_d	The weight of document, d
W_q	The weight of the query, q
t	The current term
$f_{d,t}$	The frequency of the term, t, within the document, d
f_t	The number of documents that contain the term, t
N	The number of documents within the collection

Table 4-1: Explanation of the variables in Equation 4-1

4.2. Data Structure

The query engine starts by reading the index from a file and uses this to re-create the data structure outlined in section 3.2. Each query is then read in and analysed to produce a data structure such as that shown in Figure 4-2.

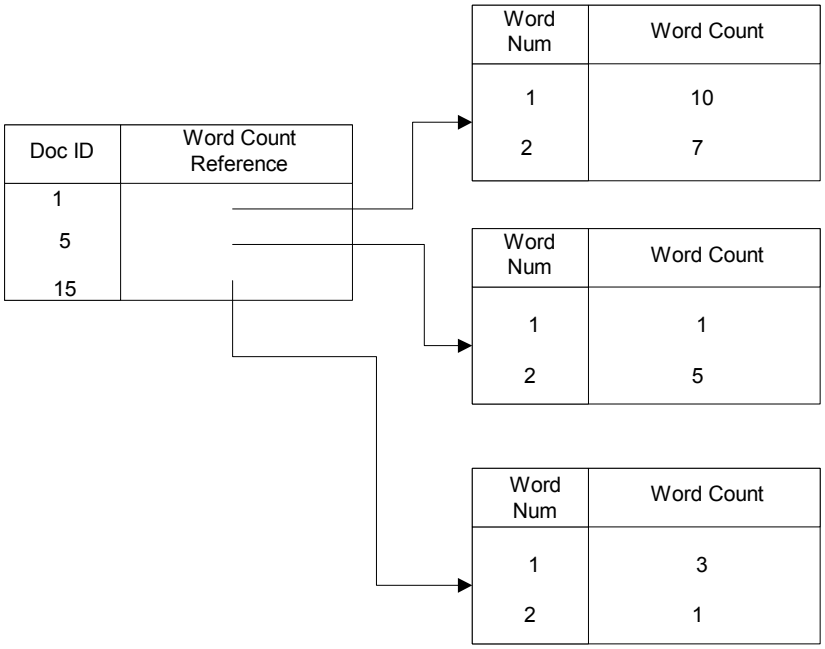

Figure 4-2: Data structure generated for each query

This structure is document based, rather than word based, but is easy to construct from the query words and the index. Pseudo code for generating this structure for a query is given in Pseudo Code 4-1.

It can be seen from the data structure used for the query and the algorithm for building this structure that the index is stored in a way which easily allows this algorithm to build up the data structure. Also the combination of indexing and querying structures contain all the data required for ranking the documents, using the TF or TF×IDF ranking algorithms.

```
Read in a line at a time from the file containing the queries, until an entire query has
been read in.

Split the query into tokens; remove any stop words present (if the user has requested)
and stem the resulting tokens (if the user has requested)

For each token in the query

        Get the documents which contain this token from the index

        Store the number of documents returned for use in calculating IDF later

        For each document which contains this token

                If we have seen this document before for this query then

                        Update the hash table of words-counts for this document and adding
                        this word, and the count for this document

                Else

                        Create a new hash table and store in it the token and the count
                        for this document, then store against the document in a hash table
```

Pseudo Code 4-1: Code for generating the data structure given in Figure 4-2

4.3. Implementing the Vector Space Model

Implementing the vector space model requires the use of two equations: to calculate Term Frequency, TF, and Inverse Document Frequency, IDF. Term frequency is related to how often the specific term appears within a document, and is calculated so to be independent of the length of the document. The equation used, in this study, to calculate TF is given below.

$$\mathrm{TF} = \frac{\log_e(t+1)}{\log_e(l)}$$

Equation 4-2: Equation used to calculate TF

In the equation t is the frequency of this term within the document, and l is the length of the document in terms. The term frequency is increased by 1 so that values of 0 are not produced by this equation. As an example if we have two documents both of which are ten terms in length, and both include the word *algorithm*, although the second document contains it twice, and the first only once then: $\mathrm{TF}_1 = 0.30$ and $\mathrm{TF}_2 = 0.48$. This clearly shows that documents which contain a term more frequently (with regard to document length), occur higher in the ranking.

IDF measures how frequent the current term is across the entire document collection. The equation used to calculate IDF, in this study, is given below.

$$\mathrm{IDF} = \log_e\left(\frac{N}{n}\right)$$

Equation 4-3: Equation used to calculate IDF

In this equation N is the number of documents in the collection, and n is the number of documents which contain the current term. As an example if we have two words *algorithm* and *evaluation*, and a document collection of 10 documents, then if algorithm appears in one document and evaluation in five: $\mathrm{IDF}_{\mathrm{algorithm}} = 2.30$ and $\mathrm{IDF}_{\mathrm{evaluation}} = 0.69$. This clearly shows that terms which only occur in a few documents are rated higher than more commonly occurring words.

The two ranking algorithms used in this study are TF and $\mathrm{TF}\times\mathrm{IDF}$. Using just TF leads to a very crude ranking algorithm which makes no real attempts to implement Equation 4-1. Using $\mathrm{TF}\times\mathrm{IDF}$, although it may not appear so at first glance, does implement the cosine equation given in section 4.1.

For a single term the $\mathrm{TF}\times\mathrm{IDF}$ can be defined and simplified as in Equation 4-4.

$$\mathrm{TF}\times\mathrm{IDF} = \frac{\log_e(t+1)}{\log(l)} \times \log_e\left(\frac{N}{n}\right) = \frac{\log_e(t+1)\log_e\left(\frac{N}{n}\right)}{\log(l)}$$

Equation 4-4: Reformulation of $\mathrm{TF}\times\mathrm{IDF}$

This is still not the same as Equation 4-1, but if we now generalise Equation 4-4 to all the tokens in a query-document combination we get Equation 4-5.

$$\mathrm{TF}\times\mathrm{IDF}(Q,D)=\sum_{w\in Q\cap D}\frac{\log_e(t+1)\log_e\left(\frac{N}{n}\right)}{\log(l)}=\frac{1}{\log(l)}\sum_{w\in Q\cap D}\log_e(t+1)\log_e\left(\frac{N}{n}\right)$$

Equation 4-5: Multiple token version of $\mathrm{TF}\times\mathrm{IDF}$

Now it should be clear to see that the above, is proportional to cosine(Q,D_d), with log(l) being the document weight, W_d, w being the terms in the query and document, and W_q being omitted as it is constant for a given query. So the equation given below is used in the implementation of the system to calculate $\mathrm{TF}\times\mathrm{IDF}$.

$$\mathrm{TF}\times\mathrm{IDF}(Q,D)=\frac{1}{\log(l)}\sum_{w\in Q\cap D}\log_e(t+1)\log_e\left(\frac{N}{n}\right)$$

Equation 4-6: The $\mathrm{TF}\times\mathrm{IDF}$ equation used in the implementation

4.4. Configuring the Query Engine

The query engine can be configured to work in multiple ways. Firstly the same command-line switches are available for the query engine as for the indexer (i.e. stop word removal and stemming) these work in the same way as with the indexer so see section 3.3 for more details on these options.

The other options available to configure the query engine are as follows. One obvious configuration is how many of the relevant documents the system should return. To specify a number of document simply specify the `-d` switch followed by the required number of documents. If you don't specify how many documents should be returned then the default is to return the top ten documents much like modern Internet search engines.

As was mentioned in section 4.3 there are two ranking algorithms which can be used, TF or $\mathrm{TF}\times\mathrm{IDF}$. The default is to use $\mathrm{TF}\times\mathrm{IDF}$, as this is the best performing algorithm, however, to use just TF simply supply the `-t` command line switch.

The final way of configuring the query engine is to specify the `-m` command line switch. This option means that the vector space model is no longer being implemented properly, as the vector used to represent the query can contain multiple instances of the same word, i.e. the length of the vector is no longer the number of words in the language. As an example if the word *algorithm* appears twice in the query then, the frequencies for this word in all documents will be used twice in ranking the documents. This option was originally implemented as a mistake in the code, but was actually observed to produce good results as will be illustrated in section 5, and so was retained as a configuration option.

5. EVALUATION AND RESULTS

5.1. Brief Description of the Evaluated Systems

All the systems (using all the combinations of configuration options to produce systems), have been evaluated against the supplied queries so as to be able to rank them, in order of retrieval performance.

The systems are named using up to four lower case letters. The coding scheme used to name the systems is shown in Table 5-1.

Symbol	*Description*
i	The $\mathrm{TF}\times\mathrm{IDF}$ ranking algorithm is being used
t	The TF ranking algorithm
m	Multiple instances of the same word are allowed in the queries
s	Stemming is used prior to processing
w	Stop words are removed prior to processing

Table 5-1: Description of the system naming scheme

As an example, consider two systems which both allow multiple words, use stemming and stop word removal and hence are only different in the ranking algorithm would be named *imsw* and *tmsw* (TF×IDF and TF respectively). This coding scheme will be used extensively throughout the following sections.

5.2. Traditional Evaluation Methods

The most common way to describe retrieval performance is to calculate how many of the relevant documents have been retrieved and how early in the ranking they were listed[1]. This leads to the following definitions:

The precision P_r of a ranking method, for a cut off point r, is the fraction of the top ranked documents that are relevant to the query:

$$P_r = \frac{\text{number retrieved that are relevant}}{\text{total number retrieved}}$$

Equation 5-1: Equation used to calculate precision

In contrast the recall R_r of a method at some value r is the proportion of the total number of relevant documents that were retrieved in the top r:

$$R_r = \frac{\text{number relevant that are retrieved}}{\text{total number relevant}}$$

Equation 5-2: Equation used to calculate recall

As an example if the document collection consists of 100 documents 10 of which are relevant to the query then if we retrieve top 15 ranked documents, containing 5 relevant documents then:

$$P_{15} = \frac{\text{number retrieved that are relevant}}{\text{total number retrieved}} = \frac{5}{15} = \frac{1}{3}$$

$$R_{15} = \frac{\text{number relevant that are retrieved}}{\text{total number relevant}} = \frac{5}{10} = \frac{1}{2}$$

Since recall is a non-decreasing function of rank, precision can be regarded as a function of recall rather than of rank. This relationship is formalized in a diagram known as a recall-precision curve, which plots precision as a function of recall (the values on a precision-recall graph are interpolated from values taken at different values of r).

Plotting precision-recall curves for different algorithms on the same graph allows you to easily rank the algorithms; if one curve is completely above another then it is the better algorithm. Unfortunately this rarely happens, as the curves usually intersect many times.

All the combinations evaluated for this study, were plotted on a precision-recall curve[2], and it was clear that those combinations which used both stemming and stop word removal were superior to those without. The graph showing all combinations was too cluttered to be displayed here and so Figure 5-1 shows only the precision-recall curves for the four combinations which use stemming and stop word removal (calculated when retrieving the top 100 relevant documents).

As you can easily see from this precision-recall curve, the systems which use the TF×IDF ranking algorithm clearly outperform the systems which only use the TF algorithm. Interestingly, however, the system which allows multiple instances of words in a query (weighting these words as more important to the query), manages to outperform the more traditional vector-space model (in which each word is only used once), although the performance difference is only small.

---

[1] The definitions of precision, recall and precision-recall curves in this section are taken from a great description which can be found in [2].

[2] The precision-recall values were calculated using a Perl evaluation scripted provided by R. Gaizauskas as part of the material prepared for [3].

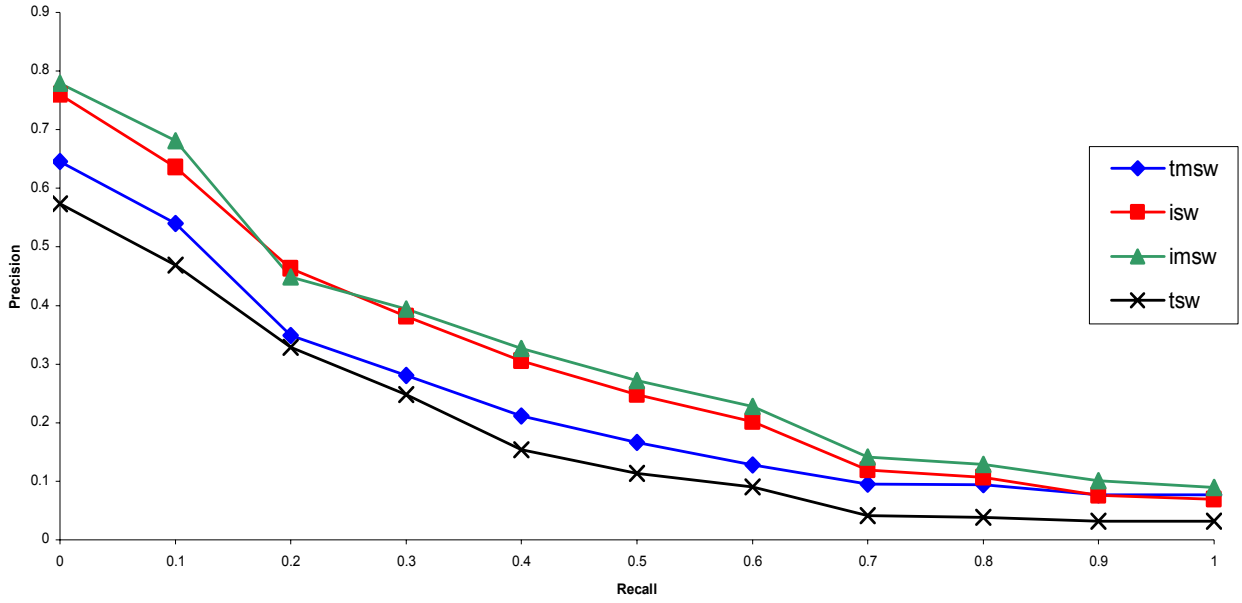

Figure 5-1: Precision-recall curves for all the systems which use stemming and stop word removal

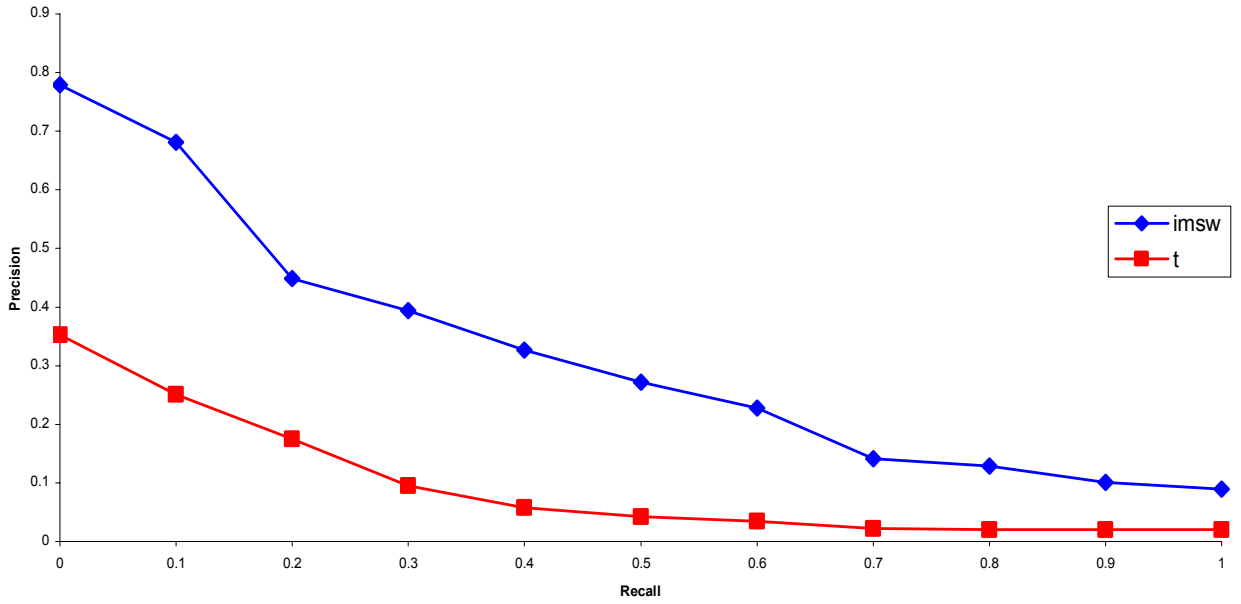

Figure 5-2: Precision-recall curves showing the best and worst performing systems

The precision-recall graph above shows the best and worst performing systems (the figures used are from a run where the top 100 documents were retrieved). It can clearly be seen from this graph that there is a vast difference in performance between these systems.

5.3. Internet Style Evaluation Methods

Although the evaluation measures used in the previous section are the traditional, well founded methods of evaluating document retrieval systems, some would say that they are now out-dated [3]. The reason that precision and recall may no longer be adequate measures of effectiveness is due to the way the information needs of society have changed over recent years, especially with the rapid growth of the Internet.

When work first started on document retrieval systems, they were designed to work in a library setting, where the user would want to know about every book that was at all relevant to their area of interest. Precision and recall are very good at measuring how good a system is at finding all the relevant documents within a collection (books in a library), but needs have changed and this situation is not what document retrieval systems are now being used for.

Currently most people instantly think of Internet search engines when they hear the phrase *document retrieval*, and they would of course be right. Internet search engines work across vast document collections, ranking and returning relevant documents in much the same way as the older library systems did. What has fundamentally changed is the way in which users use the results from search engines. Most users don't want to scroll through page after page looking through the top 100 relevant documents they would rather find just one document which is relevant and will answer their query. Therefore relevant documents must appear in the top ten (search engines usually display ten documents per page) so that a user can instantly see a document which appears to be relevant. The Google.com search engine has taken this one step further with their *I'm Feeling Lucky* method of searching[3]. This method, instead of displaying the traditional set of results, simply takes the user to the highest ranked document. Clearly this implies that users are quite happy with a single relevant document, and have no real use for all or even the top X relevant documents.

What is need then are newer evaluation measures that allow systems to be ranked according to how well they work in a search engine like fashion. The following introduces two search engine evaluation methods; position and answer. Position, is simply the position at which the first relevant document is ranked. Clearly the lower the position value the better the system. Answer on the other hand is the percentage of queries (for which there are relevant documents) where a relevant document is returned in the top ten, i.e. a document is returned which can answer the query. These measures are what we automatically associate with the performance of a good Internet search engine.

Figure 5-3 shows the average position values for the different systems which were evaluated (results were calculated from the positions within the top 100 documents).

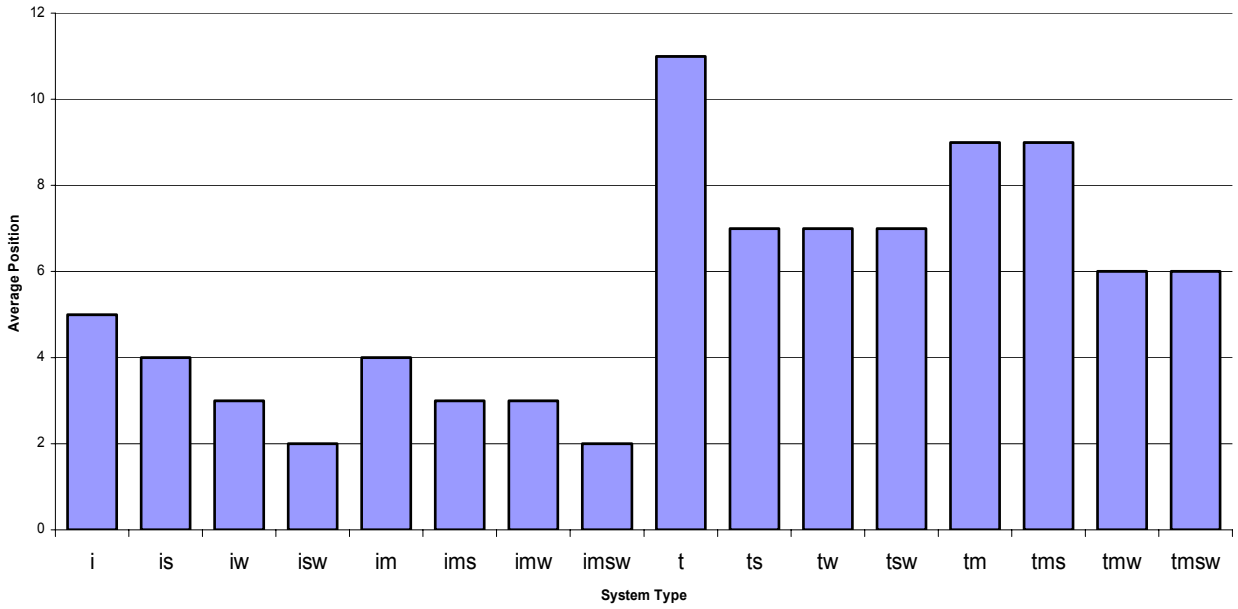

Figure 5-3: Average position of the first relevant document

It can clearly be seen that those system which use the TF×IDF ranking algorithm produce much better results with the best system, on average, returning a relevant document at position 2 in the ranked list. Comparing this with the systems which use the TF algorithm, the best of these systems only return documents as high as position 6, which is worse than the lowest performing TF×IDF system. The worst result is clearly the t system, which on average returns the document at position 11, which would be off the first page of results returned from a search engine.

The Figure 5-4 shows the results of the answer measure for all the systems evaluated. Again, it is clear to see that those systems which use the TF×IDF ranking algorithm outperform those which rely only on the TF algorithm. In fact all of the TF×IDF systems get 90% or more, with two systems reaching just over 98%, which corresponds to providing a relevant document in the top ten for 51 out of the 52 queries. Examination of the query which was not answered by these systems[4], shows that most of the documents consisted of very short titles and no abstracts where as the query is quite

[3] Go to http://www.google.com and click the *I'm Feeling Lucky* button instead of the *Google Search* button to try this feature for yourself.

[4] This was query number 22 in the CACM collection, and the first relevant document returned by the imsw system was at position number 26.

long - hence a lot of irrelevant words in the query are matched above those which would influence the retrieval of the relevant documents.

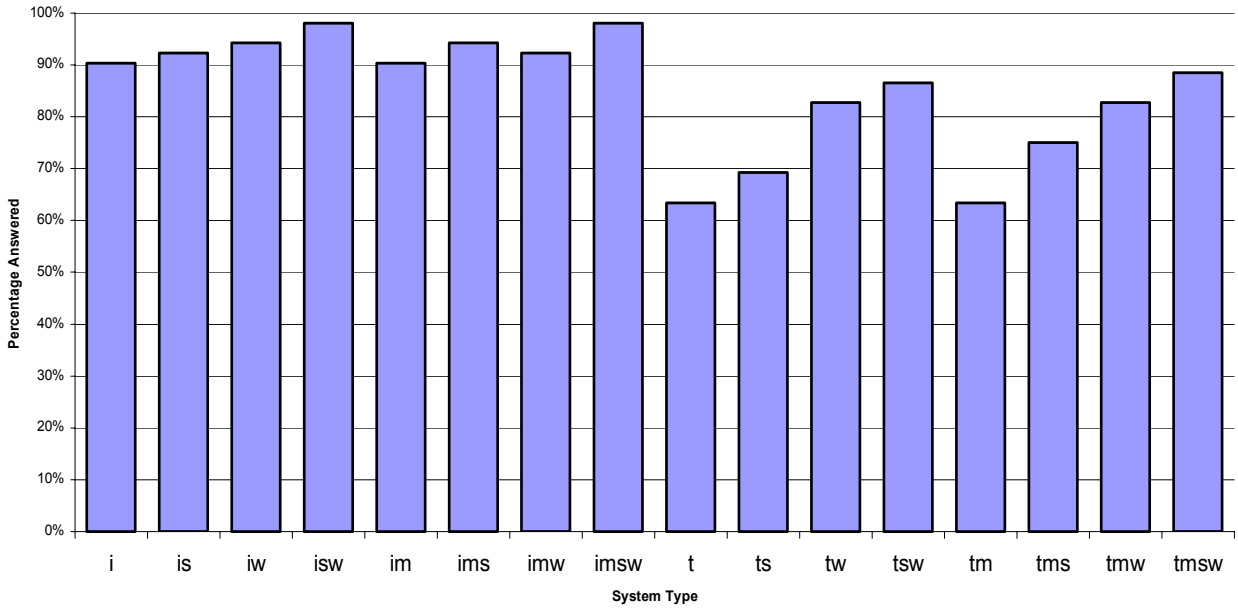

Figure 5-4: Percentage of queries which contain a relevant document in the first ten returned

With both of the new measures, the best performing systems were isw and imsw, both performing equally well, showing that the use of word weighting in the query (as provided by the `-m` command line switch) may not be as useful in boosting the retrieval performance as was initially suggested by the traditional evaluation methods.

6. FUTURE WORK

The main thrust of any future work in this area, should probably be concerned with working on the weighting of individual words within the query. This study has already shown that waiting terms which appear more often in a query higher than other words, produces better precision-recall figures than treating all the words in a query equally, as the vector space model chooses to do. Weighting of query words, could be based on frequency, as here, or could be more involved, maybe using part-of-speech information, to weight words (weighting words such as determiners lower than nouns and verbs).

Another area of work could be to look at exactly how tokens should be generated from plain text documents. The system implemented in this study simply breaks the text at white-space then removes punctuation from the beginning and end of the token. As an example “part of speech” will be treated as three separate tokens by this system, whereas “part-of-speech” will be treated as one token, and the system will never treat them as in anyway a like (for retrieval purposes) even though we would assume they have the same meaning.

An important area of any future work would be to generalise the indexer, so that other document collections could be indexed. This would allow a bigger document collection to be used (by combining collections and queries etc.), and research to be carried out to see if the performance of document retrieval systems is dependent upon the size of the collection over which it operates.

7. CONCLUSION

The best performing document retrieval system was the imsw system, which is interesting as it doesn’t implement a pure vector-space model, but rather weights terms within the query based on their frequency of occurrence within the query. Although only a small performance advantage is achieved when evaluated with the traditional evaluation approach of precision-recall curves, it is enough of an increase to warrant future work in this area. When evaluated with the new Internet search engine style evaluation methods imsw and isw performed equally well, showing that the weighting of query words may not be as important for modern retrieval as it would be for traditional retrieval systems.

The worst system evaluated was t, which is understandable, as it uses a very poor ranking algorithm, term frequency, and none of the other improvements used by the other systems, such as stemming or stop word removal.

8. REFERENCES

[1] R. Baeza-Yates, B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[2] I. Witten, A. Moffat, T. Bell. *Managing Gigabytes,* 2nd Edition. Morgan Kaufmann Publishers, 1999.

[3] R. Gaizauskas, L. Guthrie. *Text Processing (Lecture Notes)*. Presented at the University of Sheffield 2001

[4] E. Voorhees. *Overview of the Trec-9 Question Answering Track*. Available from http://trec.nist.gov/pubs/trec9/papers/qa_overview.pdf

[5] M. Shaw, J. Wood, R. Wood, H Tibbo. *The Cystic Fibrosis Database: Content and Research Opportunities*. LISR 13, pp. 347-366, 1991. The collection is available from http://www.dcc.ufmg.br/irbook/cfc.html

[6] L. Wall, T. Christiansen, J. Orwant. *Programming Perl*, 3rd Edition. O'Reilly & Associates Inc, 2000.