Reexamining *tf.idf* based information retrieval with Genetic Programming

NIR OREN
University of the Witwatersrand

The tf.idf family of vector based information retrieval schemes is very popular due to its simplicity and robustness, as well as its tractability to enhancements. This paper proposes a method to automatically perform a search for new tf.idf like schemes using genetic programming. The results of this automated search are then evaluated in a simple usage scenario. Also evaluated are the effects of using different fitness functions in the genetic programming phase.

Categories and Subject Descriptors: H3.3 [**Information Retrieval**]: Retrieval Models; I2.2 [**Automatic Programming**]: Program Synthesis—*Genetic Programming*; I2.1 [**Applications**]:

General Terms: Experimentation, Performance

Additional Key Words and Phrases: Genetic Programming, Information Retrieval, tf.idf

1. INTRODUCTION

The central focus of the field of Information Retrieval (IR) has always been the ability to search for information relevant to a user's needs within a collection of data. Most commonly, this involves finding a small set of documents within a large document collection, based on a set of search terms provided by the user. Many search methods have been suggested, but as yet, no clear winner has emerged, and the field of indexing and searching approaches is still an active research area within IR.

One of the most popular paradigms for indexing and searching, discussed further in the next section, is the vector based model of IR. Amongst the vector based models is a family of very popular schemes referred to as *tf.idf* methods. These schemes employ a small number of document, collection and query features to provide a measure of relevance for each document with respect to the user's query. While *tf.idf* approaches are simple, effective and popular, when used without any enhancements, they are not as powerful as more modern techniques.

This paper proposes using an automatic programming technique known as genetic programming to generate new *tf.idf* like information retrieval methods. Genetic programming involves the 'breeding' of new programs from old programs. The breeding process attempts to isolate the best traits of the parent programs, thus creating new programs with improved capabilities.

2. BACKGROUND

The aim of this section is to provide the reader with the background necessary to evaluate the remainder of the paper. It will also provide details on similar approaches to information retrieval.

2.1 Genetic Programming

Genetic programming (GP), first introduced by Koza [Koza 1992] adapts the approach used in genetic algorithms to search through program space in an attempt to find a program which solves a predefined problem as accurately as possible. the term 'Genetic Algorithms' describe a set of optimization techniques that are used to search a space for optimal points. The space is searched in a directed, stochastic manner, and the method of searching borrows ideas from the theory of evolution.

An implementation of GP contains a population of individuals, each representing a program. Each individual has an associated fitness value, which is calculated using a global fitness function, and reflects how close the individual program is to an optimal solution. The population is evolved through successive generations, with the population at each generation 'bred' from the fittest members of the previous generation.

An individual program is most commonly represented by its parse tree, due to the ease with which the breeding operations can be carried out on this structure. Each node in the parse tree represents an operation, which takes in values computed by its children as parameters. Terminal nodes commonly represent input values taken from the problem domain.

Author Address:
N Oren, School of Computer Science, University of the Witwatersrand, Private Bag 3, Wits, 2050, South Africa; nir@cs.wits.ac.za
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2002 ACM 0004-5411/2002/0100-0001 $5.00

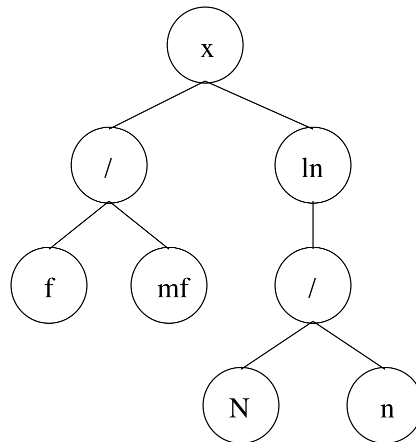

Figure 1. *A parse tree representing the function* $(f/mf) \times \log\frac{N}{n}$. *f is the raw term frequency, mf is the maximum raw term frequency, N is the total number of documents in the collection, and n is the number of documents containing the index term.*

Figure 1 illustrates a simple parse tree representing a basic *tf.idf* family indexer.

The most common breeding operations are:

Copying which takes the individual and moves it unchanged into the next generation.

Mutation where a node within the parse tree is chosen at random, and changed to another node. The new node is usually one that can take on the same number of children, i.e. a multiplication node may change to a division node, but not to a *log* node, as the *log* node can only have one child. A similar operation, also commonly referred to as mutation, involves selecting a mutation point as previously described, and then replacing the subtree below it with a new tree.

Crossover takes two (or, less commonly more) trees as inputs. A crossover node is then randomly selected from each tree. The subtrees rooted at each crossover node are then exchanged.

Other, less commonly used breeding operations involve permutation, where the subtree below a permutation point is reversed, and editing, which uses prespecified rules to simplify a program (for example, by changing all boolean disjunction expressions with identical child nodes to a single instance of the child node). It is also possible to prune trees, as well as to replace a tree with a completely new tree. Both of these can be considered specializations of the mutation operator.

A number of problems quickly become apparent when attempting to automatically create programs with a GP based approach. One issue involves the handling of types: how should a program act when a node that expects an integer is passed a boolean value? A number of methods have been proposed to deal with this problem. Firstly, it is possible to deem all programs with type conflicts as unfit, thereby minimizing their chances of propagating to the next generation. In some cases, it is possible to massage the returned types into a form suitable for consumption by the parent node, in the previous example, truth could be represented by 1, and falsehood by 0, allowing a node expecting integer values to successfully handle this situation.

A related problem, with similar solutions, involves dealing with illegal operations. A common example of this would involve a division node being asked to divide any number by 0. An alternative solution to this class of problem would involve modifying the operation to handle illegal data. For example, in the division by zero case, the division operator could be modified to set the denominator to a very small but non–zero value.

Due to the number of evaluations that must be undertaken, genetic programming is computationally intensive. Parallel techniques exist that provide a near–linear speedup based on the number of processors used. This is achieved by subdividing the population according to 'islands', with one island per processor. Fitnesses are computed on an island–wide basis. When a generation finishes running on its island, its fittest individuals are migrated to other islands. It is possible to run this approach asynchronously, thus keeping the processors busy at all times. The disadvantage of this approach is the reduction in the richness of the populations: crossover only occurs on an island-wide basis instead of across the whole system. The migration of individuals serves to counteract this penalty to some extent. Another parallel technique was used in the course of this research, and will be described in a later section.

2.2 Information Retrieval

An IR system is used to retrieve data applicable to a user's needs based on queries posed to the system by the user. Usually, the system consists of a number of components:

(1) A document store, often referred to as the document collection, where documents searchable by the system reside.
(2) A query formulation subsystem, which allows users to pose queries to the system.
(3) An indexing component, responsible for accepting raw documents and converting them into a form usable (i.e. searchable) by the system. Documents can only be added to the document collection by going through this component.
(4) A processing component which operates on query and document representations, deciding what to return in response to a query, and in what order.
(5) An output component, which displays the results of the user's query. This component is usually closely linked with the query formulation system so as to enable the user to further operate on the results of a query.

While much research is still underway regarding the various components of an IR system, within the context of this paper, we are only interested in the document retrieval mechanism, and will thus only focus on the indexing and processing component. Furthermore, while IR research deals with many different types of data such as audio, video and other mixed–media documents, we will assume that our document collection consists only of unstructured, unmarked, single–language text files. Similarly, queries are assumed to consist of a single natural language string.

A number of popular retrieval techniques exist, including the boolean model which returns only fully matching documents, and the probabilistic model, which computes the probability of a document being valid, and thus returns an ordered list of documents. The vector approach to information retrieval also returns a list of documents ordered by believed relevance, and is probably the most common way in which IR systems function.

A number of IR techniques represent a document by a vector, with element i within the vector taking on a value (usually 0 and 1) based on the presence or absence of the word indexed by i within a global word list. Queries are represented in a similar manner. Determining whether a document is relevant for a given query involves computing a similarity measure between the document and query. Most commonly, this is done by calculating the cosine of the angle between the two vectors. The resulting value, ranging between 0 and 1, is normally used to rank the documents by believed relevance, with higher ranked documents appearing before lower ranked documents. A retrieval value threshold is commonly set, if a document-query similarity value is below this threshold, the document is deemed irrelevant, and is not retrieved.

Vector based IR multiplies the document vector on a term–by–term basis with another vector, known as the weighting vector. This weighting vector represents the varying abilities of various terms to describe the document's contents. For example, the word 'the' is probably not very useful in discriminating whether a document is or is not relevant, while a word such as 'genetic' would normally better indicate the topic the document discusses. The resultant vector is then used to compute a similarity measure.

One of the earliest and most popular ways to create weighting vectors is the *tf.idf* family of weighting schemes. The term frequency component (*tf*) of a term t_i for a document d_j is calculated according to

$$tf_{i,j} = \frac{frequency_{i,j}}{max_l frequency_{l,j}} \tag{1}$$

i.e. the raw frequency of a term divided by the frequency of the most common term in the document. The *idf*, or interdocument frequency component is normally computed as follows:

$$idf_i = \log \frac{N}{n_i}$$

where N is the total number of documents in the collection, and n_i is the number of documents in which the term t_i appears.

In *tf.idf* weighting schemes, the component of the weighting vector for document d_j at position i (i.e. for term t_i) is of the form

$$w_{i,j} = tf_{i,j} \times idf_i$$

A number of variations have been proposed to this basic formula. Salton and Buckley [Salton and Buckley 1988] have experimented with a number of these variations, and found that the basic formula is sufficient for most collections.

The query term weights are often calculated in a different manner to document term weights. Salton and Buckley recommended using the formula

$$w_{i,q} = idf_i \times (0.5 + 0.5 \times tf_{i,q}) \tag{2}$$

where the term frequencies are computed over the query rather then over the original collection. Inverse document frequencies for the query weighting vector are still computed over the entire document collection.

Salton and Buckley show that different weighting vector schemes perform better or worse on collections and queries with different properties. For example, when short queries are most common (such as in a web search engine environment), query weighting vectors should be modified to increase the influence of the second component in equation 2. Similarly, retrieval of documents in collections with highly varied vocabularies seem to benefit from an increased *tf* weighting. This

result indicates that some sort of ‘adaptive’ information retrieval algorithm, which modifies itself to perform better on a specific collection, may be worth pursuing. The research described in this report suggests one approach to creating an adaptive information retrieval algorithm.

2.3 Evaluating IR systems

With the number of IR systems being researched and in operation, methods are needed so as to be able to compare their abilities. Two very different aspects of an IR system can be measured: efficiency, and effectiveness. Efficiency can be measured in terms of the resources required by the system, including the storage space required to store the document collection, and the computing resources needed to perform operations on the collection, such as the addition and removal of documents, and performing queries. While it is sometimes possible to compute the time and space complexity of the various aspects of the system, more concrete efficiency measures are often required when implementing a concrete system, but these measurements are difficult to perform in a machine independent manner.

Effectiveness attempts to measure, as the name implies, the effectiveness of an IR system at satisfying a set of queries. Given a sufficiently general document and query collection, the effectiveness should provide a domain neutral measure of the ability of the system. The measure of effectiveness is further complicated by the fact that it is dependent on the type of task being evaluated. Interactive systems must be evaluated in a different way to systems in which user feedback plays no role.

Many IR researchers [Rijsbergen 1979] believe that a satisfactory approach to the evaluation of an information retrieval system is yet to be found. Since this is still a rich and ongoing area of research, we will only examine the most common evaluation methods. Furthermore, it is assumed that the system under evaluation operates with minimal user interaction.

The most widespread method of evaluating an IR system involves providing precision–recall values for a set of queries posed on a specific document collection. Usually, a precision–recall diagram is plotted so that comparisons between IR systems can be made visually.

Given a document collection and a query, let R be the number of relevant documents in the set for this query, and A be the number of documents retrieved by the IR system. Finally, let I be the number relevant documents within the documents retrieved by the IR system. Recall and precision can then be defined as:

$$Recall = \frac{I}{R}$$

$$Precision = \frac{I}{A}$$

The basic precision and recall measures assume that the IR system returns an unsorted list of results, which is then evaluated in full. If this is not the case, recall and precision values change as more documents within the result list are examined (the assumption is made that the list is ordered from the highest to least believed relevance). This is done by recomputing precision over the seen documents whenever a relevant document is found in the retrieved document list. For example, assume the following documents are retrieved in response to a query, with the documents marked by an asterisk indicating relevant documents:

rank	document number	relevant	rank	document number	relevant
1	d_{23}		6	d_{100}	*
2	d_{42}	*	7	d_{43}	
3	d_{12}	*	8	d_{67}	
4	d_1		9	d_{94}	*
5	d_7		10	d_{10}	

Assume that this query has 5 relevant documents, of which four were retrieved as above. At a recall level of 20% (i.e. 1 out of the 5 relevant documents have been seen), precision is 50%, since one out of the two seen documents are relevant. At the 40% recall level, precision increases to 66%. At 60% and 80% recall, precision values are 50% and 44%. Finally, at 100% recall, precision drops to 0%, as all relevant documents were not retrieved.

Precision-recall curves are normally drawn by computing precision at 11 standard recall values, namely, 0%, 10%, 20%, ..., and 100%. If, as in the above example, insufficient relevant documents exist to compute recall at all these points, the values at the standard points are set as the maximum known precision at any known recall points between the current and next standard points. The example presented above would therefore yield the precision–recall curve illustrated in Figure 2.

Usually, precision–recall curves are computed by averaging the precisions obtained at the standard recall values over all queries posed to the system.

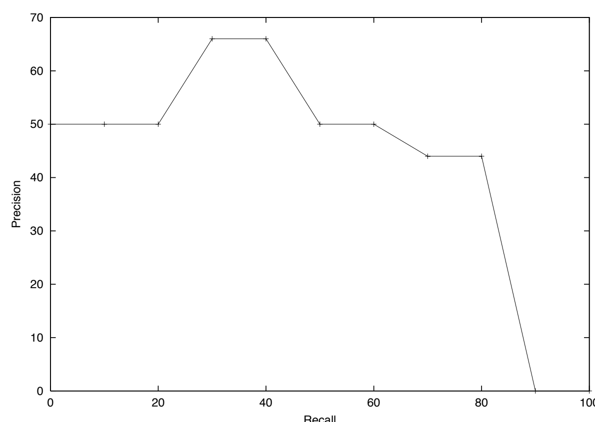

Figure 2. *A sample precision–recall diagram.*

When averaged over a number of queries, precision–recall curves tend to follow an exponential decay curve. Intuitively this results from the fact that an algorithm would generally rank at least a few relevant documents quite highly, therefore yielding high precision for low recall values. As the number of relevant documents returned by the system increases however, more and more irrelevant documents are returned within the results. Obviously, a perfect algorithm would have a flat precision–recall curve at the 100% level.

While precision–recall figures are often useful, many situations require a single figure as a measure of system performance. Baeza-Yates [Baeza-Yates and Ribeiro-Neto 1999] provides a number of measures appropriate for situations when one wishes to evaluate the performance of an algorithm over a single query.

A number of criticisms have been leveled against precision–recall based measures, and other measurement methods have thus been proposed [Hersh et al. 1995]. Despite its problems, Precision–recall still remains the most popular IR system evaluation scheme.

2.4 Related Research

Genetic Algorithms and genetic programming have been investigated for some time in the context of information retrieval. Gordon [Gordon 1988] describes a scheme which utilizes genetic algorithms to alter keywords associated with various documents as the user interacts with the system. By doing this, more accurate keywords can be assigned to each document, which can then be matched with keywords in user queries so as to provide improved performance. Yang and Korfhage [Yang and Korfhage 1993] investigated a similar method. Instead of altering keywords, they used genetic algorithms to alter the weighting assigned to various tokens. As the population evolved, the accuracy of the keyword weightings in representing the topic of a document in the collection improved.

Gordon, together with a number of collaborators, has continued to investigate evolutionary techniques for information retrieval. Pathak, Gordon, and Fan [Pathak et al. 2000] investigated using genetic algorithms to choose between a number of similarity functions. It was believed that since different similarity functions appear to greatly change the quality of results for different collections and query sets, it would be possible to search through the space of matching functions and find an optimal combination of these for a specific collection.

Fan, Gordon and Pathak have recently started to investigate the effectiveness of an approach similar to the one proposed in this research [Fan et al. 2000]. Briefly, genetic programming is used to generate a population of weighting vector generators. By evolving this population they attempted to create individuals well suited to answering specific queries. This research while proceeding in a similar manner attempts to achieve a different set of goals: we try to evolve individuals that will operate well for all queries posed to a specific document collection, rather then attempting to operate well for a small set of queries.

3. EXPERIMENTAL APPROACH

This research aimed to investigate the application of genetic programming to the creation of *tf.idf* like document evaluators. These evaluators can then be used to create document representation vectors, which are in turn utilized in similarity measurements to created a document list ordered by believed relevance to a user's query. This approach suggests the possibility of creating fine tuned retrieval tools for specific document collections, as well as the discovery of an evaluator superior to basic *tf.idf* when applied to general collections.

Finding an improved document evaluator allows one to leverage existing higher level techniques such as relevance feedback [Salton 1989] to further improve the quality of the IR system. These higher level methods will require no change

```
OperatorDepth:=3. MinTerminalDepth:=2. MaxTerminalDepth:=6.

function selectNode(depth)
  if (depth<OperatorDepth)
    return random operator.
  if (depth>=MaxTerminalDepth)
    return random terminal.
  return random operator or terminal
end.

function generateSubtree(depth)
  currentNode:=selectNode(depth)
  if currentNode has a possibly unlimited number of children,
     set the number of children to between 1 and 3
  for each child of currentNode
     generateTree(depth+1).
  return currentNode
end

function generateTree
  return generateSubTree(0).
end
```

Figure 3. *The algorithm used to generate trees in the initial population.*

as they operate well on any vector IR based scheme.

This paper reports back on the evaluation of our approach in a very simple usage scenario: we assume a constant document collection, to which different queries may be posed.

The remainder of this section describes the data on which experiments were run, as well as the implementation details of the experiments. Results will be discussed in the next section.

3.1 Data

Training and evaluation were performed on the Cystic Fibrosis database. This dataset consists of 1239 documents and 100 queries covering various aspects of Cystic Fibrosis research. The documents were published in various medical journals between 1974 and 1979. This dataset was chosen due to its easy availability, as well as its size.

Each document contains 11 fields, some of which are used to keep track of the document (fields such as the paper number, and a record number), with the remaining fields containing information that can be used in answering queries. The only field used for evaluation purposes within the context of this research was the field containing an extract of the text from the original papers.

Queries within this collection include, amongst other fields, the text of the query, together with a list of relevant papers. Relevance was determined by four human judges, with each judge assigning a number between 0 and 2 to the paper's relevance to the query. For the purposes of this research, a document was deemed relevant if a single judge assigned it a non-zero relevance score.

3.2 Implementation

Before the experiments were run, the documents and queries were preprocessed by removing stop words, and by performing stemming using Porter's algorithm [Porter 1980]. Some documents were also removed as they contained no text extracts.

70 randomly selected queries were used in training, with the remainder being used for the final evaluation of the resultant trees. By utilizing the same collection with different queries, we attempt to represent the case in which new queries are posed to an IR system operating on a static collection.

100 trees were created, 99 of which were random, with the final tree representing standard *tf.idf* indexing. This final tree was introduced as a form of a-priori knowledge for the genetic programming process. The system was then run over 150 generations. The algorithm used for tree creation is a variant of the standard 'grow' algorithm [Koza 1992] for tree creation, and is shown in figure 3.

Breeding was performed by directly copying the fittest 10% of the population, after which copying was performed with a 10% likelihood, the mutation of a single node occurred 25% of the time, and the mutation of an entire subtree occurred with a 20% likelihood. The chance of crossover was set at 35%. Finally, new trees were introduced into the new generation 10% of the time. The likelihood of an individual being chosen for a breeding operation was directly proportional to its fitness. In most GP experiments, the selection of breeding probabilities is done by trial and error. The breeding operation probabilities used here were intuitively selected in an attempt to fulfill a number of criteria:

— The direct copy means that successful trees will propagate to the next generation.

Node	Description
idf	Inverse document frequency
rtf	Raw term frequency
tf	Term frequency as calculated in Equation 1
N	Number of documents in collection
n	Number of document in which word appears
M	Term frequency of most common word
l	Document length
a	Average document length
c	A constant node
$+ - \times$ / log $\sqrt{}$	Standard mathematical operations

Table I. *Nodes used in the genetic programming.*

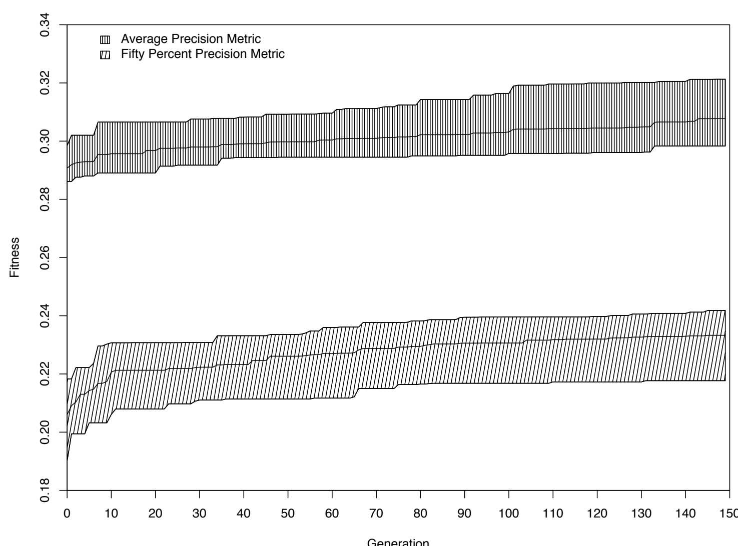

Figure 4. *The minimum, mean and maximum fitness of the best individual within each generation during training.*

—Mutation and the introduction of new trees means that large portions of the search space are explored.

—Crossover, historically seen as a very effective operation in improving the fitness of individuals, should occur quite often.

The modification of breeding probabilities changes the rate at which individuals explore the search space. Due to the speed in which overtraining appeared in the experimental runs different breeding parameters were not investigated.

An auxiliary research goal was to examine the effects of different fitness functions on the resulting population. To this end, two different fitness functions were used. The first involved computing the precision at the 50% recall level, with the second consisting of the average precision calculated at the 11 standard recall values. The first metric would thus give the example shown in Figure 2 a fitness of 50, while the second would result in a fitness of 43.8.

Since genetic programming techniques are inherently random, three runs were carried out for each fitness function. Within each run, all individuals were evaluated on all training data. Due to the computational cost of this approach, a cluster of approximately 40 computers was used. A single workstation was used to coordinate the work, with it handing out individuals for evaluation to the remaining computers. Results were collated on the central computer, which was also responsible for the creation of new individuals whenever all calculations for the previous generation had been completed.

Table I describes the terminal and non-terminal nodes used when evolving the programs.

4. RESULTS

Figure 4 illustrates the fitness of the fittest individual within each generation under both fitness schemes. While not obvious on the figure, standard *tf.idf* results in the same fitness as the first generation. Different fitnesses were obtained in each run due to the different queries used in training and evaluation. As expected, both fitness functions increase in a non-strictly monotonic manner as training progresses. According to this figure, it appears as if individuals evolved using the average precision fitness function will perform better then those created with the precision at 50% recall metric. Lastly, it can be seen that improvements of between five and eight percent in performance on training data appear once training is complete.

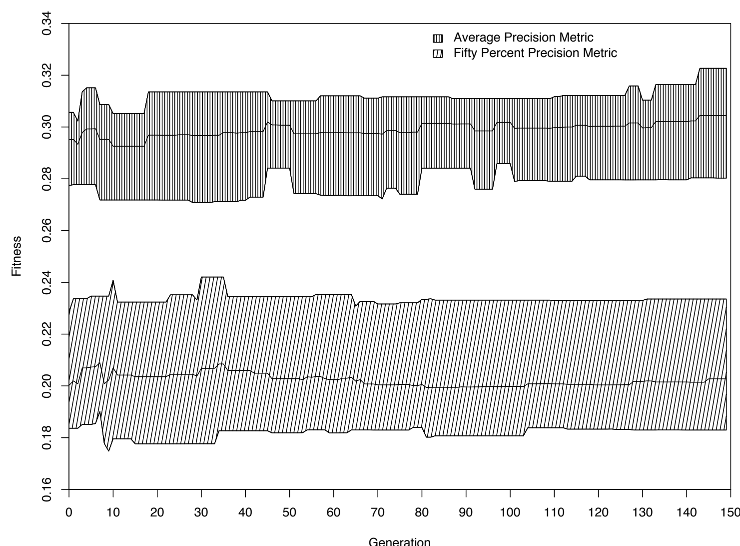

Figure 5. *The minimum, mean and maximum fitness of the best individual within each generation during testing.*

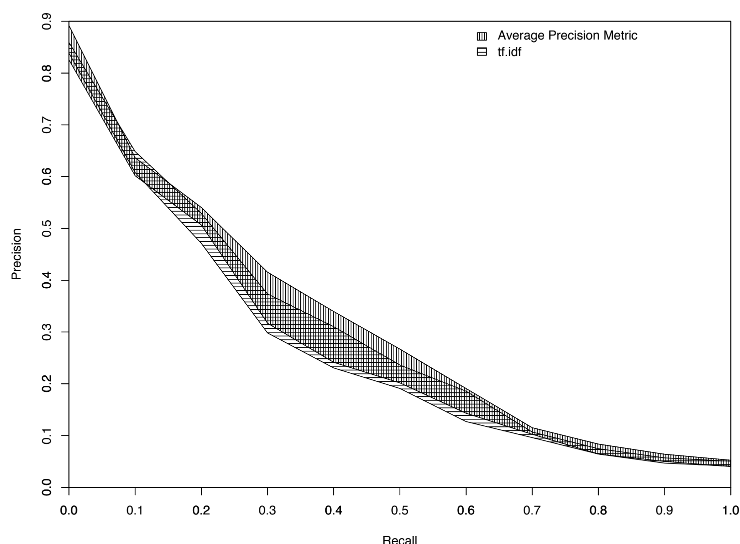

Figure 6. *Precision Recall curve for* tf.idf *and the best performing individual (as determined during the training phase) using the average precision metric, evaluated on the training dataset.*

Evaluating the best individuals from each generation on the remaining queries yields the results shown in Figure 5. These results indicate a mean improvement in fitness of approximately three percent. Both fitness functions appear to peak after less then ten generations, probably due to overtraining. While the precision at 50% recall fitness function continues to perform poorly after this stage, the average precision fitness function appears to improve again after a few more generations elapse. One possible reason for this may be that while further improvements at a certain recall percentage cannot be obtained, gains at other recalls may still be possible. Once more, it appears as if the average precision fitness function outperforms the precision at 50% recall metric.

Due to the nature of the chosen fitness functions, they are strong representatives of how well the individuals function in an IR setting. To obtain a proper indication of their ability, one needs to compare precision–recall values. Figure 6 illustrates the precision–recall curves for *tf.idf* and the average precision metric. Figure 7 does the same for the recall at 50% precision metric. For clarity, the mean value is not illustrated in both of these figures. Figure 8 shows the performance of both metrics when tested over the entire (training and testing) dataset. The aim of evaluating over the entire dataset was to eliminate the possibility of poor results due to a set of pathological queries.

These figures show that the effectiveness of our approach is highly dependent on the manner in which fitness is

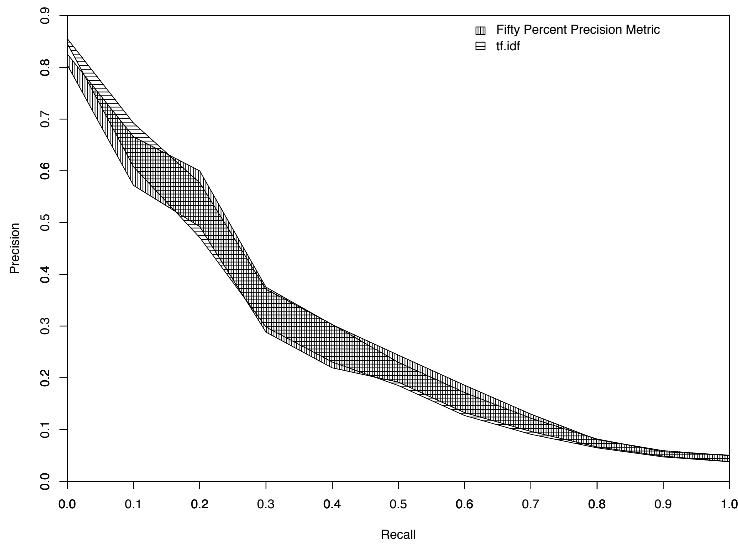

Figure 7. *Precision Recall curve for* tf.idf *and the best performing individual (as determined during the training phase) using the precision at 50% recall metric, evaluated on the testing dataset.*

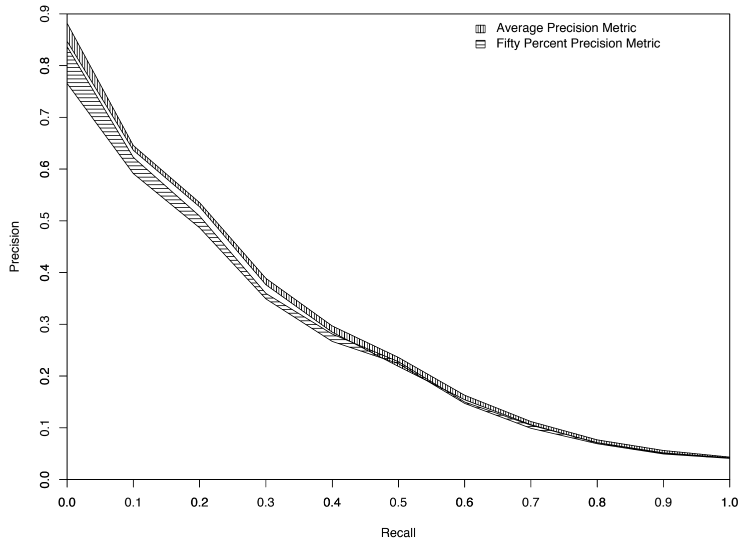

Figure 8. *Precision Recall curve for the best performing individuals (as determined during the training phase) using both metrics, evaluated over the entire (training and testing) dataset.*

calculated: the average precision fitness measure slightly outperforms standard *tf.idf*, while the precision at 50% recall fitness function performs almost identically to *tf.idf*. It is interesting to note that the average precision metric outperforms the other fitness function even at a recall level of 50%, as the latter attempts to optimize for that case. One possible reason for this behavior, as mentioned earlier, is that the average precision function somehow allows more flexibility to the individuals, thus reducing the chance of being trapped in a local fitness maximum.

Figure 9 illustrates a typical individual evolved in the course of this research. As is typical of programs generated using evolutionary techniques, understanding its operation is difficult, if not impossible. The individual contains a large amount of redundant information. While simplification is possible, this would affect the further evolution of the system.

Unfortunately, while improvements are seen over basic *tf.idf* they are not significant. Savoy and Vrajitoru [Savoy and Vrajitoru 1996] states that improvements in average precision of 5% are considered significant, with increases of greater then 10% seen as very significant. The increase of the mean average precision between *tf.idf* and the average precision

```
((( tfValue )*(( idfValue ) / ((( maxTfValue ) - (( wordDocumentCount ) - ( tfValue ))) - ( wordDocumentCount)))*
(documentCountValue))*((averageDocumentLengthValue) log )* (((( maxTfValue) - (averageDocumentLengthValue)) +
((((documentCountValue) + ( maxTfValue ) + ( tfValue )) + (0.33231506139613d)) / ( idfValue ))) log ))
```

Figure 9. *A typical evolved individual, represented as a string. Note that the log operation operates on the statement preceding it.*

metric was 4.4%. A number of measures could be taken to improve these results, and are discussed next.

5. FUTURE WORK AND CONCLUSIONS

Research is now underway to investigate this approach in more complex scenarios. In one of these scenarios, additional documents are added to the collection after training has completed. In the second, training is done on the complete dataset, with evaluation taking place on an unrelated dataset. The first extension to this work attempts to simulate the situation where a document collection housing related documents is modified as time progresses. Since the documents are related, the quality of the evaluator is expected to fall between the results described here, and those obtained when applying this approach to generate general purpose evaluators. The second extension aims to investigate the ability of this approach to function as a general–purpose evaluator.

Another future avenue of research involves investigating ways of improving the effectiveness of the approach. Possible techniques include:

— Additional terminals and operators. By adding new terminal nodes, individuals can be evolved that make use of additional semantic content. Also, by using a different set of terminals, it should be possible to create individuals that represent other information retrieval approaches. Additional nodes may increase the time taken to evolve useful individuals. Also, some terminal nodes may be very expensive to compute. The addition of new operators may increase the descriptive capability of the individuals, and may be required to support more complex terminal nodes.
— The combination of a number of genetic programs to form a voting system for document relevance may increase the quality of results. Also, by forming some sort of pipeline, with the output of one genetic program feeding into the next, and limiting the terminals available to each, it may be possible to reduce the computational costs of complex terminal nodes.
— Query weightings are usually calculated differently to document weightings. Large gains in retrieval performance may be obtained by allowing for this to occur, perhaps by evolving two populations, one for the queries, the other for documents, in parallel.

Other questions still open to investigation include the effects of modifying the likelihood of the various breeding operations, as well as changing the manner in which individuals are chosen for breeding. As this research has shown, the selection of a fitness function has large effects on the final population. It may thus be worth searching for more effective fitness functions.

This paper examines the possibility of using genetic programming to aid in vector based information retrieval. While the results presented here show only modest gains over *tf.idf*, a number of avenues of research remain that may significantly boost the effectiveness of this approach.

REFERENCES

BAEZA-YATES, R. AND RIBEIRO-NETO, B. 1999. *Modern Information Retrieval*. Addison-Wesley.

FAN, W., GORDON, M. D., AND PATHAK, P. 2000. Personalization of search engine services for effective retrieval and knowledge management. In *Proceedings of the 2000 International Conference on Information Systems*. 20–34.

GORDON, M. 1988. Probabilistic and genetic algorithms in document retrieval. *Communications of the ACM 31,* 10, 1208–1218.

HERSH, W. R., ELLIOT, D. L., HICKAM, D. H., WOLF, S. L., AND MOLNAR, A. 1995. Towards new measures of information retrieval evaluation. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 164–170.

KOZA, J. 1992. *Genetic Programming: On the programming of computers by means of natural selection*. MIT Press.

PATHAK, P., GORDON, M., AND FAN, W. 2000. Effective information retrieval using genetic algorithms based matching functions adaptation. In *Proceedings of the 33rd Hawaii International Conference on System Science*.

PORTER, M. F. 1980. An algorithm for suffix stripping. *Program 14*, 130–137.

RIJSBERGEN, C. V. 1979. *Information Retrieval, 2nd Ed.* Butterworth & Co.

SALTON, G. 1989. *Automatic Text Processing*. Addison-Wesley.

SALTON, G. AND BUCKLEY, C. 1988. Term-weighting approaches in automatic text retrieval. *Information Processing & Management 24,* 5, 513–523.

SAVOY, J. AND VRAJITORU, D. 1996. Evaluation of learning schemes used in information retrieval. Tech. Rep. CR-I-95-02, Universitié de Neuchâtel, Faculté de droit et des Sciences Ećonomiques.

YANG, J. AND KORFHAGE, R. R. 1993. Query improvement in information retrieval using genetic algorithms: A report on the experiments of the trec project. In *Proceedings of the 1st Text Retrieval Conference*. 31–58.