USING GENETIC PROGRAMMING TO QUANTIFY THE EFFECTIVENESS OF SIMILAR USER CLUSTER HISTORY AS A PERSONALIZED SEARCH METRIC

Except where reference is made to the work of others, the work described in this thesis is my own or was done in collaboration with my advisory committee.

Brian David Eoff

Certificate of Approval:

Juan E. Gilbert
Assistant Professor
Department of Computer Science and Software Engineering

John A. Hamilton Jr.
Associate Professor
Department of Computer Science and Software Engineering

W. Homer Carlisle
Associate Professor
Department of Computer Science and Software Engineering

Stephen L. McFarland
Acting Dean, Graduate School

Using Genetic Programming to Quantify the Effectiveness of Similar User Cluster History as a Personalized Search Metric

Brian David Eoff

A Thesis

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Master of Science

Auburn, Alabama

December 16, 2005

USING GENETIC PROGRAMMING TO QUANTIFY THE EFFECTIVENESS OF SIMILAR USER CLUSTER HISTORY AS A PERSONALIZED SEARCH METRIC

Brian David Eoff

_____
Signature of Author

_____
Date

Copy sent to:

Name Date

THESIS ABSTRACT

USING GENETIC PROGRAMMING TO QUANTIFY THE EFFECTIVENESS OF SIMILAR USER CLUSTER HISTORY AS A PERSONALIZED SEARCH METRIC

Brian David Eoff

Master of Science, December 16, 2005
(B.E., Auburn University, 2003)

100 Typed Pages

Directed by John A. Hamilton Jr.

Online search is the service that pushes the Internet. One must only look at the success of a company such as Google, an idea from a 1998 graduate research paper that has in 2005 not only become a wildly successful company, but whose very name Google had become synonymous with the verb search, to realize how important search is.

Many IR researchers have suggested that the next great step in search is to make the process more personal. Search results should be tailored to the individual user. Early attempts at personalization such as relevance feedback have never gained popularity with users due to the need for further interaction. The end goal is personalization without the user having to contribute more of their attention.

I propose that personalization can be accomplished by observing a user's document selections. That a page's overall popularity is important, but more important is the pages that users similar to the primary user find popular. I also propose that history should not be based solely on a listing of prior documents a user has found relevant, but on the clusters of documents a user has found relevant. Clusters allow for pockets of information to be observed, and thus a fuller understanding of the user can be determined.

How then do I determine if this new metric is usable short of implementing a search engine using the metric, putting it online, and hoping users flock to it? Genetic programming was created to solve such problems. Genetic programming can be used to determine if a newly proposed information retrieval metric (collaborative filtering based on cluster history) is effective. By giving a genetic programming framework a training set containing documents, queries, and relevance judgments an optimal ranking function can be found. The genetic programming framework could incorporate the new metrics proposed along with traditional search metrics such as term frequency and document length. If these proposed metrics survived the evolution process they can be determined to be effective in the returning of relevant documents to a user's query.

ACKNOWLEDGMENTS

For Mom, Dad, Sarah, and the rest of my people. They know who they are.

Style manual or journal used IEEE Standards Style Manual (together with the style known as “aums”).

Computer software used The document preparation package TEX (specifically LATEX) together with the departmental style-file `aums.sty`.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Google, the most popular online search engine with a 15.3% share of visits receives 250 million search requests per day, and has indexed 8.1 billion web pages [7][8][9]. Yahoo [10], Google's closest competitor, receives a mere 10% of all search requests. In seven years Google has gone from a graduate research paper to a billion dollar company, primarily due to PageRank, a single search metric that uses incoming page links to determine a page's popularity [11]. PageRank was the most significant advantage it had over its competition. Google was five years late to the start of online search engines, yet they were able to make up the distance due to their search algorithm metric.

Information overload has been predicted since the 1940's [12]. The number of new documents created for the web is growing at a substantial rate. If the internet is going to continue being a useful source of information, online search engines not only have to keep pace with the new document bulk, but also improve their performance in returning relevant documents to a user's query. Without search engines the internet becomes an unnavigable mess.

Current search engines give little or no consideration to a user's past queries. Past user histories should be used in determining the relevance of a document to a user's query. If a user searches for the term "Java" based on their past queries and page choices, a search engine should be able to determine if they are interested in the programming language or coffee. Google's PageRank algorithm pushes popular documents higher in the order of returned documents to a user's query. Instead of using what the whole online community considers to be a popular page, a search engine might perform better if it returned pages

that are popular with other users that have similar histories to that of the user. Then small pockets of interest could be developed.

Search algorithm designers are unsure of what metrics to use and the way in which to combine and balance different metrics. There are a variety of metrics dealing with link structure of web pages, term weighting and popularity. The goal of these designers is to create an algorithm that will give them optimal performance, since error will always exist in these algorithms. There is no perfect search algorithm. The human creation of the queries, and the difference between the perceived and actual relevance will always be an issue.

Genetic programming can be used to create an optimal solution to the search algorithm problem. For the genetic programming framework to function efficiently, large amounts of data are necessary, which can be used as training sets to learn the correct documents for a query. The data could be available to any large search engine, documents, user queries, and depending on what documents the users selected after their query-relevancy judgements. A small sample of this information could be fed into a genetic programming framework with the hope of producing an algorithm that would return relevant results. A designer could also remove metrics and determine which were the most useful. They could also test new metrics quickly without having to inflict a possibly poor idea on real users.

The research conducted through this thesis attempts to establish that a genetic programming framework can be used to determine the usability of various search metrics and to also demonstrate that the user histories of similar users is a successful metric in determining relevancy of documents to a user's search.

This thesis will examine the various techniques used in information retrieval. Chapter One will give the reader background information on the field of information retrieval as it applies to online search engines. The goal is to show the advancement through the history and demonstrate how personalized search is the next step in information retrieval. Chapter

Two will be a complete literature survey of the information retrieval and genetic programming cannon. Chapter Three will discuss the implementation and reasoning behind the genetic programming framework and the communal personalized search metric. Chapter Four will describe the experiments conducted to prove that communal personalized searching is able to return more relevant results to the user's queries. Chapter Five will conclude the thesis with an overview of how genetic programming and personalized search fit into online information retrieval and a reflection on the findings.

CHAPTER 2

BACKGROUND INFORMATION

> "Although information retrieval has lately become quite a fad, I intend in this paper to stand back and take an unhurried look at what is going on, and try to predict where this field must go and what it must do in the future." - Calvin N. Mooers (1959) [13]

2.1 What is Information Retrieval?

Information retrieval (IR) was a term coined by Calvin Mooers in 1950. The goal of an IR system is to organize data in such a way that a user can quickly gain access to the knowledge they desire. Van Rijsbergen, a predominant IR researcher, stated that the problem inherent in information retrieval is, "we have vast amounts of information to which accurate and speedy access is becoming ever more difficult" [14]. A traditional library card catalog is an example of an information retrieval system. It is an attempt to condense a collection in such a way that a user can better access what they are interested in without having to look through all documents.

2.2 History of Information Retrieval

The problem of searching raw data for information has been around for over one hundred years. In 1897 a concordance, or index, of every meaningful word in the Bible was published [15]. And while that might seem a trivial task today (simply input a copy, parse it, and build a count of the words) it was a lifelong task in 1897. These concordances were early examples of organizing information in such a way that a user could quickly access

what they were interested in. The indexes created by these early concordance makers could be viewed as the ancestors of the inverted index structure used in most search engines. An inverted index is a hash data structure where the hash key is a word, and inside the hash is a linked list with two fields. The two fields contain the number of occurrences and the document location. After the computing revolution of the 1960's concordances were no longer created by hand, and the process was significantly quickened. The last handmade concordance was a collection of Byron's works; it took a mere twenty-five years [15]. With computer assistance the task would take minutes.

Some of the earliest discussion of digital search came from Vannevar Bush, one time director of the Office of Scientific Research and Development. Bush invented the concept for what he would later call "Memex" in the 1930's. He described Memex as "a device in which an individual stores all his books, records and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility" [12]. Also, Bush described an idea of documents that were interconnected with each other through "trails," this concept is considered to be the inspiration for hypertext. Bush created the Memex concept (though he never actually constructed it) to counteract what he viewed as information overload. Researchers at Microsoft are currently attempting to construct a Memex like system, MyLifeBits [16].

In the 1960's Cornell professor Gerald Salton began focusing on IR research. Salton led the group that developed SMART, jokingly known as Salton's Magical Retriever of Text, but commonly known as System for the Manipulation and Retrieval of Text. Salton contributed to the discovery of a variety of IR techniques: vector space model, term weighting, relevance feedback, clustering, extended Boolean retrieval, term discrimination value,

dictionary construction, term dependency, text understanding and structuring, passage retrieval, and automatic text processing using SMART. The vector space model was particularly ground breaking. In a vector space system both the query and the document were treated as vectors, and their relevancy to each other was determined by the distance apart.

The first internet search engine was Archie, created in 1990 by Alan Emtage [17]. This was before Tim Berners-Lee's creation of HTTP. All documents on the internet were stored on FTP servers. Archie's crawler probed the various FTP servers for listings of their files and indexed each of those files.

Wandex was the first WWW (World Wide Web) search engine, it was created by Matthew Gray in 1993. Earlier in the year Gray created the "World Wide Web Crawler," or Wanderer, an early crawling robot. The Wanderer caused a small amount of controversy, the release of the bot caused a notable loss in network performance. Wanderer mistakenly accessed the same pages repeatedly, often a hundred times in a single hour [17].

In 1992 the Department of Defense, along with NIST co-sponsored the Text REtrieval Conference (TREC) [18]. The aim was to promote research in the information retrieval community by supplying the infrastructure that was needed for such a huge evaluation of text retrieval methods. The TREC conference contained a variety of tracks focusing on question-answering systems, multimedia search, search involving structured data and many others are added on a yearly basis. At the 2003 TREC confernce ninety-three groups from twenty-two countries participated [18]. Also of note is that in the first six years of the TREC conference the effectiveness of retrieval systems presented doubled [18].

During the early nineties bots were only recording the title of web pages, the first hundred or so words and their locations. In 1994 Brian Pinkerton created a crawler that recorded all of the text of each document [17]. This was the first time full-text search was available on the WWW. Pinkerton entitled his system WebCrawler. At this time it only

contained the documents from six thousand servers. WebCrawler was eventually sold in 1997 to American Online (AOL).

Yahoo was the first online search engine to gain widespread popularity. Yahoo was not exactly a search engine in the traditional sense. Yahoo was a hierarchal arranged catalog, that was not fed by a crawler or bot, but was inputted by Yahoo's editors.

In the past small innovations in the search community, if implemented well, have lead to substantial returns in terms of popularity. Each of the discussed projects had their moments, and there abilities attracted users. There seems to be a natural progression. There are no huge innovations, just bit by bit improvements. Users will not move on to a new search engine unless that engine offers a significant improvement over what they are currently using. It is not enough for a company to get on even footing with their competitor, they must surpass them. Conveniently, if a new company does surpass the competition then users tend to flock to their product.

The history of information retrieval section of this thesis is not a complete history of internet search, but the reader can quickly see the progression of the technology. The University of Nevada's Veronica, Thinking Machine Corporation's WAIS, UCSTRI, Netfind and Lycos have been left out [3].

2.3 Charles Darwin, Programmer

> "Computer programs that evolve in ways that resemble natural selection can solve complex problems even their creators do not fully understand." - John Koza [1]

Since the early nineties a sub-discipline of Artificial intelligence known as evolutionary computing has gained popularity. The idea that evolution can be applied to creating solutions was first proposed by John Holland. Evolutionary computing used the theories of evolution, survival of the fittest, selective breeding and mutation to answer difficult questions. Instead of simply attempting to brute-force a solution, evolutionary computing allowed an intelligent selective search of the solution space of a problem [1][19]. Given sufficient computational resources a genetic programming solution could yield results that compete with those of the best domain experts [20].

Genetic Programming was developed principally by John Koza. The individual was no longer a coded representation of the problem, the individual was a computer program. The goal of a genetic programming system was to discover a program that produced some desired output for a particular set of inputs [1]. These programs were represented as a tree structure, as shown in Figure 2.1. This structure allowed for the various programs to be easily manipulated from generation to generation. The trees consisted of functions and terminals. The functions could be programming operations, arithmetic, mathematical, logical or domain specific. The terminals could be numerical constants or variables.

The initial population was created pseudo-randomly. Koza described this step as a "primordial ooze of thousands of randomly created programs" [20]. Then the population was ranked according to the fitness of each individual using the fitness function. The goal of the fitness function was to determine a score for each individual that reflected how well their possible solution performs. The fitness function allowed the programs to be ranked, thus determining who is the "fittest" and will survive. The designer of the system had a choice he must make: what individuals get to breed, what individuals would simply enter the next generation, and which individuals would be killed off. Randomness had a role

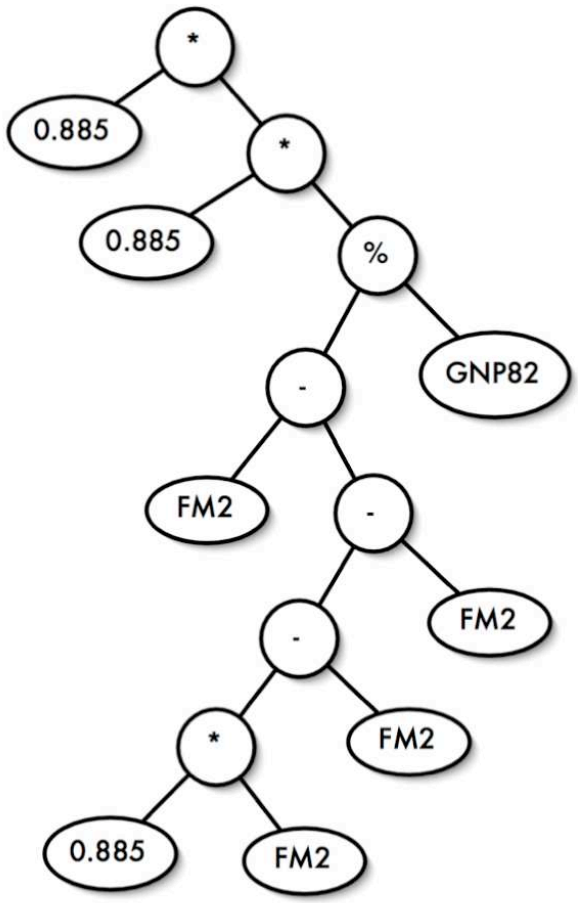


Figure 2.1: A Genetic Programming Tree [1]

to play in this. It was not wise to remove all the weakest individuals, due to the possibility that those trees might move towards a local optimum. There were various selection techniques; they include elitism, fitness-proportionate, roulette-wheel, scaling, tournament, rank, generation, steady-state, and hierarchal selection. Once the individuals had been selected they could either be mutated, where they were simply altered, or they could have been crossovered with another individual. Crossover was metaphoric reproduction. And like reproduction, two individuals could produce multiple children. An example would be if the best two individuals were crossovered four different ways to produce four children.

Genetic programming was used to solve a variety of problems in fields ranging from electrical circuit design to biochemistry and microbiology. Genetic programming has had great success with solving problems that have large solution spaces, and require an optimal solution.

2.4 Information Retrieval is Difficult

> "The simple reason: even humans are poor at deciding what information is relevant to a particular question. Trying to get a computer to figure it out is nearly impossible." [21]

Information retrieval (IR) is an unsolved problem in multiple disciplines of study, and it is not unsolved because of lack of interest. The scientist who comes up with a correct system, one that always return the document that the user needs and does it in a reasonable amount of time, will surely enjoy great personal wealth and numerous accolades. Due to the high profitability of this research, search engine algorithms are kept secret, and thusly advances become less likely [22]. The vast majority of the techniques used by search engines were discovered in the 1970's by IR researchers such as Salton. The seventies

were a time when researchers were not as concerned with the profitability of their ideas, and openly published their findings.

Patterson best describes the inherent difficulty in building search engines: “At serve time, you have to get the results out of the index, sort them as per their relevancy to the query and stick them in a pretty Web page and return them. If it sounds easy, then you haven’t written a search engine” [23]. The need for disk space and processing power is staggering. Those needs are slightly outweighed by the need for large amounts of bandwidth. There is often no way of knowing if what you are doing will fully work. There is no testing suite for determining how well a search engine operates. Scalability becomes an issue with large online search engines. The search engine is a real-time system; it needs to respond quickly to a user’s request.

Another issue is that web site owners want their sites featured highly by search engines, and they will often attempt to abuse the search engine ranking functions to achieve higher status. For example, if word count highly affects the ranking of a page, a web site creator might repeatedly insert a word they want to be associated with their site in hidden text. A normal user will not be able to see the word, but a crawler will see it. A site owner might pay a “link farm” service to boost its search placement in query results [22]. A link farm has the ability to create thousands of pages with links directed towards a single web site. This will give the illusion of popularity, and can cause metrics such as PageRank to be inaccurate. These techniques are known as “spamming” search engines. An entire industry of search engine optimizers (SEO) has been created willing to sell their services of boosting a web sites search ranking [24]. If a search engine does not protect itself from these tricks their search results will become useless, and thusly they will lose users. Techniques have come out to beat search engine spam, but often times those techniques are also

quickly beaten. It is a constant arms race, and often requires human involvement in the page rankings.

CHAPTER 3

LITERATURE REVIEW

Information retrieval is a popular research subject. The goal of most information retrieval research is simply about giving users the document most relevant to their queries. In the search for a solution to that problem researchers have studied various ways of sorting documents, indexing documents, compressing various parts of the document set, and applying reasoning to the query. The goal of this literature review was to find information on how to best return the most relevant documents to a user's query.

The papers that influenced and shaped the work conducted for this thesis are described in the following literature survey. The first section will be an overview of literature dealing with the concept of relevance. Relevance is a difficult thing to comprehend due to its abstract nature. The second section deals with research into the clustering of similar documents, followed by relevance feedback. Research into term-weighting strategies, various ranking functions, personalized search and the use of genetic programming to tune search algorithms will also be discussed.

3.1 Relevance

Relevance is a central idea in IR research. Van Rijsbergen went so far as to claim relevance is the notion at "the centre of information retrieval" [14]. Despite its significance, relevancy is a hard concept to quantify. Relevance has been declared to be the "most fundamental and much debated concern for information science." Despite the importance of relevancy there is little agreement about its exact nature. Moreover a proper way to evaluate systems making relevancy judgements has yet to be determined. The debate over relevance gets even more convoluted: a document can be perceived as being relevant to the

system based on an entered query, but not relevant to the user. Prior research has show that thirty-eight variables affect the relevancy that a human judges a document on, including the style of the document, visual layout, and difficulty of language. These variables are incredibly demanding relevancy judgements for a search engine to consider, which is what makes relevancy difficult; it goes far beyond the mere content of the document.

The most recent work on the concept of relevancy as applied to IR has been done by Stefano Mizzaro. Mizzaro discusses the confusion about relevance by researchers, stating "a great deal of such problems are caused by the existence of many relevances, not just one, and by an inconsistently used terminology: the terms 'relevance', 'topicality', 'utility', 'usefulness', 'system relevance', 'user relevance', and others are given different meanings by different authors: sometimes they are used as synonyms (two or more terms for the same concept) sometimes in an ambiguous manner (the same term for two or more concepts)" [2].

In his paper [2] Mizzaro describes four dimensions of relevancy. The first dimension is "information resources." In this dimension exists a group of three entities: document, surrogate, and information. Mizzaro defines document as "the physical entity that the user of an IR system will obtain after his seeking of information". Surrogate means "a representation of the document," to which Mizzaro gives examples such as a keyword list, an abstract or bibliographic information. The last member of the group is information, which Mizzaro acknowledges is not a physical concept, but the "entity that the user receives/creates when reading a document." The second dimension is the representation of the user's problem. In this dimension Mizzaro makes a distinction between the Real Information Need (RIN) and the Perceived Information Need (PIN) [2]. The distinction must be noted that what the user wants, what they truly want, might not be the same as what they request. This user issue is often referred to as Anomalous State of Knowledge (ASK). Another issue is the vocabulary

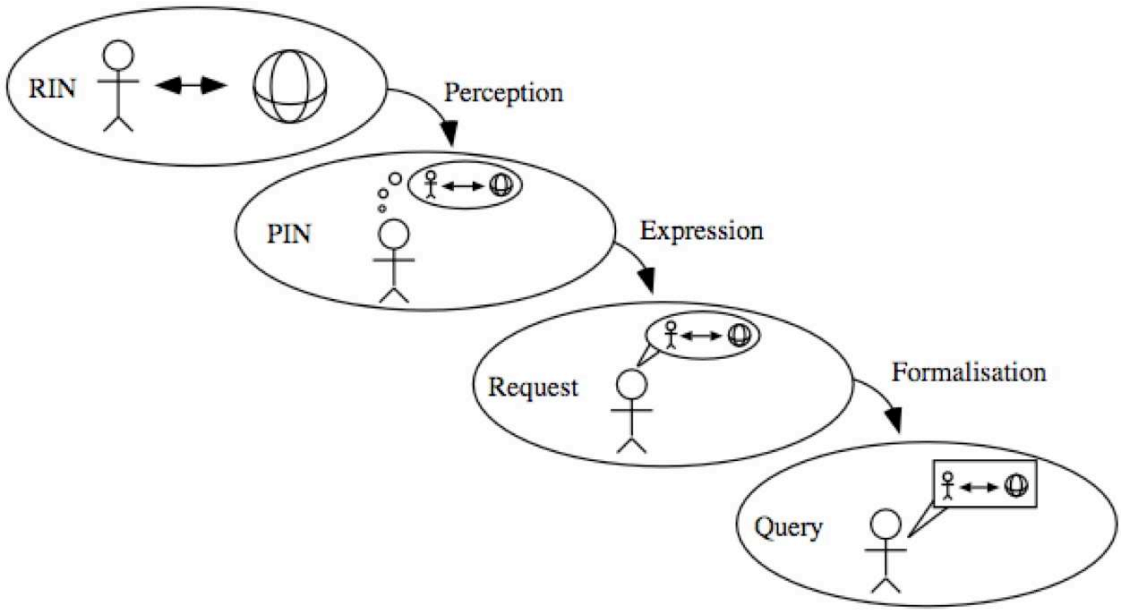


Figure 3.1: Relevance And The User [2]

problem, which is a mismatch between the terms used in the document, and the terms used in the request. The user mentally creates a request for the system, but the user must enter this request in as a query, which might require a boolean like language. The RIN, PIN, request and query are the parts of the second dimension. These four items can be viewed as states. A user must go from as RIN to PIN to a request to a query, and at many times the process can be flawed.

The third dimension is time, which isn't discussed much in IR research. A document might not be relevant to a user's query at a certain time, but may be considered so in the future. The change is often brought about by a user having a greater understanding of the material after a period of time, and a document that wasn't relevant to the user when her of she was less knowledgeable could be later on once a base of knowledge has been gained. The fourth dimension is "components" [2]. There are three components in the fourth dimension: topic, task and context. Topic is the subject area that the user is interested

in. Task is the activity that the user will use the information they receive for, ie. writing a term paper or studying for an exam. The third component is context. Context includes everything that affects the search and the decision of relevance that is not covered in topic or task [2].

Mizzaro in essence has created a framework for the discussion of relevance. In his articles he also proposes graphing relevance in a four-dimension space to fully understand it. This seems to be an unnecessary attempt to further quantify relevance into a measurable form, but Mizzaro's contribution to the relevance discussion cannot be overlooked. Relevance as a measurement must take into consideration numerous factors, and relevance is not a permanent statistic between a document and a query. Relevance must be considered in context with the user. Mizzaro also notes that relevance can change over time, something search designers rarely give consideration to. The importance of this is that a search engine's performance is judged on its ability to return a maximum number of relevant documents and a minimum number of non-relevant documents to a user's query. To create a well designed search ranking function one must fully understand the meaning of relevance. Just as important as returning relevant documents and not returning non-relevant documents is to also minimize the number of relevant documents that are not retrieved.

3.2 Document Clustering

Clustering is the grouping of similar objects. It is a research subject in a variety of scientific disciplines, due to the need to make sense of large collections of data. Many terms are synonymous with clustering such as unsupervised learning, numerical taxonomy, vector quantization and learning by observation [4]. IR researcher van Rijsbergen proposed a clustering hypothesis which stated, "closely associated documents tend to be relevant to the same requests" [14]. Clustering was originally applied to IR research as a means for

	Relevant	Non-relevant	
Retrieved	a	b	(a+b) all retrieved documents
Not Retrieved	c	d	(c+d) documents left out
	(a+c) all relevant documents in collection	(b+d) all non relevant documents in collection	(a+b+c+d) total collection

Figure 3.2: Relevance is critical in determining ranking function performance [3]

improving efficiency. Instead of determining the relevance of a query to each document in the data set, the query was compared to the centroid of the cluster, and if it was deemed relevant all documents in said cluster were also deemed relevant. This drastically reduced the number of comparisons that need to be made. Unfortunately, in most studies it was determined that retrieving the contents of the clusters whose centroids most closely matched the query did not perform as well as retrieving the top rank documents from the entire collection. Other researchers have studied the performance improvements by using clusters in the search process, but their results only showed occasional gains [25].

In current IR research, clustering usually falls into two categories: document clustering or search result clustering. Document clustering is done prior to a user ever being involved in the system. All documents are clustered based on their similarity, and as new documents are discovered they are added to a cluster. For a large document set the initial clustering can be very computationally expensive. Search result clustering tries to counteract that cost. Clustering only occurs once search results are returned, and the clustering

only applies to those results. The clustering is based on the small snippets that search engines return with their result listings. So the end results are grouped together to allow a user to browse the categories made on the fly. No record of these clusters are kept.

To accomplish document clustering, numerous algorithms have been created. Clustering algorithms fall into two categories, hierarchical or partitioning [4]. Hierarchical algorithms begins with an initial clustering, and then merges or splits the clusters until a measure of similarity has been met. The first step of a hierarchical clustering algorithm is to place each document into its own cluster. Next, the closest clusters are combined. This is done until the appropriate number of clusters has been found. That is the bottom-up approach to hierarchical clustering. In the top-down approach all document are placed into a single cluster. The two documents in the cluster that are farthest apart are the centroids of the two new clusters. All the documents that were in the original cluster now become a member of which ever of the two clusters they are closer to. Which ever cluster is the largest gets split in the next cycle.

The most common example of a partitional cluster algorithm is k-means. In k-means a number of clusters are chosen, and then each document is randomly assigned to a cluster. Next, the centroid for each cluster is calculated, and each document is assigned to the cluster that the centroid is nearest. This last step is repeated until clustering converges, and documents no longer switch clusters. Due to its random nature, various runs of the k-means algorithm will have varied results on the exact same document set. Figure 3.3 presents the tree of clustering algorithms, all the algorithms are either partitional of hierarchical.

The question of which clustering algorithm is the best performer is often debated. The speed in which the clustering algorithm performs and the quality of the clusters must be considered. The partitional algorithm K-means is faster than hierarchical algorithms [26]. It can compute in O(K*N). The hierarchical algorithms produce a higher quality of clusters.

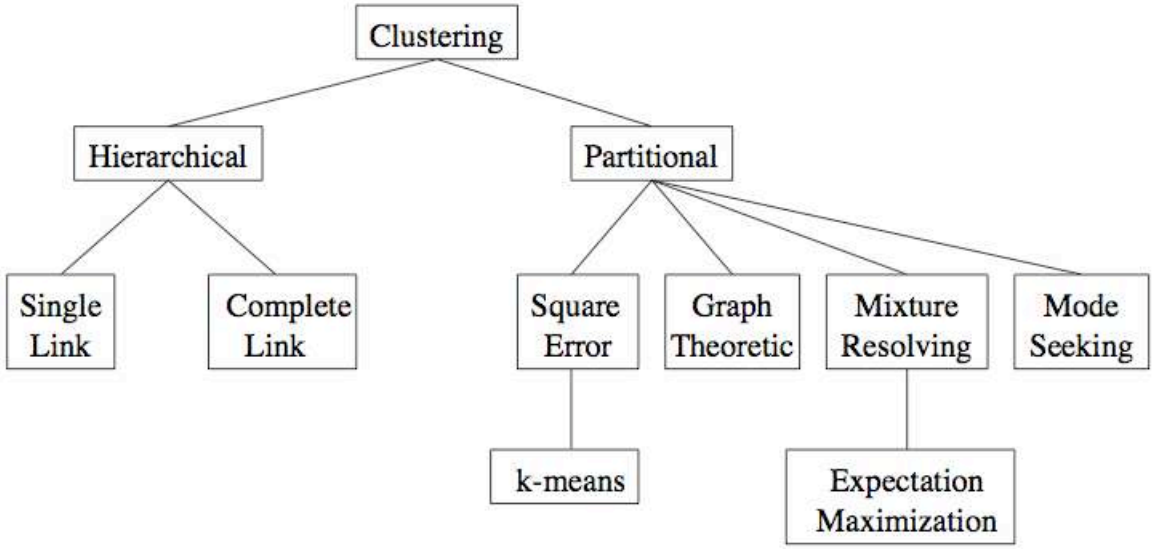


Figure 3.3: Jain's Taxonomy of Clustering Algorithms [4]

When a document set is extremely large it is wiser to use the K-means algorithm for the sake of efficiency. If the data set is small, or if the cluster must be of a high quality hierarchical algorithms make more sense [26]. In their article, "Evaluation of Clustering Algorithms for Document Datasets," Zhao and Karypis observed that partitional clustering algorithms outperform agglomerative when dealing with document datasets [27]. Quality in terms of clustering is a subjective measurement of if the clusters contain similar documents. It should also be pointed out that research has shown that only the fifty to a hundred most frequent terms in the document are needed to cluster documents [28] . The same research pointed out that clustering based on all terms actually degraded the system performance.

Clustering could be used to return documents to a query that does not contain any of the words in the query. If a document is deemed very relevant to a query, then all the documents in that document's cluster would also be deemed to have a relevance to the query. No longer will it be a simple keyword search. The beauty of this type of mechanism is that it could potentially alleviate certain issues with the ambiguity of language. An example would be if a user searched for "Search Engine Research," and the system might find the

cluster that contains documents on "Search Engine Research," and would notice that the cluster also contained "IR Research." It then might return documents that do not contain words from the original query, but returns documents that were similar to documents that were relevant to the query. This technique could alleviate the problem of a user not fully knowing the colloquial of a field.

A few online search engines are utilizing clustering in the search process. The online search engines Visimo and the Clusty use clustering extensively. The notoriously close-lipped Google labs commented in a recent article about the possible use of clustering in future incarnations of their search engine. Urs Hoelzle, VP of Engineering at Google, commented that academic implementations of clustering had little success due to lack of data, and went on to further comment, "If you have enough data, you get reasonably good answers out of it" [29].

3.3 Relevance Feedback

Relevance feedback was a technique proposed by Gerald Salton. [30] defines relevance feedback as "an interaction cycle in which a user selects a small set of documents that appear to be relevant to the query, and the system then uses features derived from these selected relevant documents to revise the original query." In the paper, "Improving Retrieval Performance by Relevance Feedback," Salton and Buckley list the main advantages of relevance feedback as: "It shields the user from the details of the query formulation process, and permits the construction of useful search statements without intimate knowledge of collection make-up and search environment," "It breaks down the search operation into a sequence of small steps, designed to approach the wanted subject gradually," and "It provides a controlled query alteration process designed to emphasize some terms and to de-emphasize others, as required in particular search environments" [31].

The question of what is the best way to implement relevance feedback has been widely researched. Relevance feedback can be implemented as term reweighing or query expansion. Term reweighing is "modifying term weights based on term use in retrieved relevant and non-relevant document" [32]. Query expansion is the taking of the most frequent terms from a document that has been deemed relevant, adding them to the query, and then resubmitting the query. Query expansion cannot take into consideration negative feedback; a term cannot be removed from the query that is not there. Term reweighing can take into consideration negative relevance. The common words from a document with a negative relevance receive a lower term weight. In the same way that common words in the document set receive a lower weight, common terms in the document which the user has described an nonrelevant get lower weight. Harman's research shows that query expansion performs better in average precision than term reweighing [32]. Query expansion results in a 27% improvement over the base search. Also Harman observes that the greatest results are achieved by adding twenty terms to the query. Both query expansion and term reweighing see performance improvements in [32]. Term reweighing performs best when not taking negative feedback into consideration according to the experiments performed in [32]. The improvement though is a marginal 0.2% improvement.

Relevance feedback is an interesting technique due to its simplicity. With the help of the user, the search engine can personalize the documents that are retrieved. Many users already use a form of relevance feedback. Once a user receives a listing of documents, occasionally they will notice a word in the descriptions that they hadn't used in their original query. They will then re-enter the query with this new information. The concept is not unfamiliar to a user, and is not terribly invasive. That does not make it easy. The user must quantify a document, often on the basis of little more that a brief description. Also this

quantification might be on a scale or declaring the document relevant or non-relevant, and both ways have flaws.

Pseudo relevance feedback or automatic query expansion works on the assumption that the top few returned pages are often the most relevant, and that pages that are similar to those are also important [33]. Pseudo relevance feedback in essence creates a new query based on the most common terms in the first few returned documents. The technique has been very successful at TREC, and the OKAPI search algorithm uses a form of it [34]. The beauty of these feedback systems is that they do not require the user to actively participate.

The reason for the rise in research on pseudo-relevance feedback is due to the unwillingness of everyday users to use relevance feedback features. The study conducted by Jansen et. al. found that less than five percent of users utilize the relevance feedback option in online search engines [35]. The authors go on to comment that "the question of actual low use of this feature should be addressed in contrast to assumptions about high usefulness of this in IR research" [35]. The authors question the direction of the research of relevance feedback commenting, "This is one of the examples where users are voting with their fingers, and research is going the other way" [35].

3.4 Term-Weighting and Ranking Functions

The ranking functions of commercial online search engines are closely guarded secrets. Little is known about the ranking function of the current leader Google beyond a graduate student conference paper. However, papers presented at TREC conferences have provided a wealth of information about the internals of many experimental ranking functions. Also, most of the ranking functions are derived from the vector space model (VSM) created by Gerald Salton.

The vector space model views both the query and the documents as a vector. The document is already in this form if indexing has occurred. The vectors will be the size of the combined vocabulary of the query and the document. If a word is contained in the query or the document, and not in the other a zero is placed in the query for that word location. The similarity is computed by taking the inner product of the two vectors [15]. If the query and the document have no words in common this product will be zero. The inner product is not the only means of measuring similarity.

Term weighting goes beyond considering the count of the query terms in a document. If a term occurs in many documents, then it gets a lower weight. This is the idea behind term-frequency inverse document frequency (TFIDF). Figure 3.4 shows the ranking function Pivoted TFIDF, Okapi[34] and INQUERY [36]. All the ranking functions have similar form, and often use the exact same variables.

Other ranking functions are web specific, and are primarily based on the linked structure of the web, such as Google's PageRank and Hyper-Interlaced Topic Selection (HITS). PageRank uses the number of pages that link to a document as a reflection of relevance, and those documents are higher ranked. Google recently filed for a patent that discussed the using of time metrics such as how often a web site's content changes, and when the domain name for the site was registered [37]. This metric takes into consideration how long ago a domain name was registered. Most malicious web sites do not hold their domain names for more than a year. Google uses this metric to reduce the number of junk sites that are returned in their search results. Also Google takes into consideration the location of the query terms in the page and the size of the text containing the query terms [38].

- Okapi BM25

$$\sum_{T \in Q} \frac{3 \times tf}{0.5 + 1.5 \times \frac{length}{length_{avg}} + tf} \times \log \frac{N - df + 0.5}{df + 0.5} \times QTW$$

- Pivoted TFIDF

$$\sum_{T \in Q} \frac{1 + \log(tf)}{1 + \log(tf_{avg})} \times \log\left(\frac{N+1}{df}\right) \times \frac{1}{0.8 + 0.2 \times \frac{length}{length_{avg}}} \times QTW$$

- INQUERY

$$\sum_{T \in Q} 0.4 + 0.6 \times \left(0.4 \times H + 0.6 \times \frac{\log(tf + 0.5)}{\log(tf_{\max} + 1.0)}\right) \times \frac{\log\left(\frac{N}{df}\right)}{\log(N)} \times QTW$$

Figure 3.4: Ranking Functions [5]

3.5 Genetic Programming And Ranking Functions

> "In the struggle for survival, the fittest win out at the expense of their rivals because they succeed in adapting themselves best to their environment." - Charles Darwin [39]

The authors of [40] propose using genetic programming to build search engine ranking functions. The IR ranking function can easily be represented as a tree, which is the data structure used in genetic programming. Their trees consist of eleven terminals, four operators, and a constant value between 0 and 1. The size of the trees are limited to a depth of no more than ten for their experiment. For the fitness function the authors use average precision. A set number of the top trees in the current generation become members of the next generation, and tournament selection is used to determine which trees would crossover and have their offspring continue into the next generation. To determine the success of their solution they compare each individual's performance against Okapi BM25, a ranking function that has consistently performed well in TREC evaluations. The corpus is the Associate Press collection spanning three years. The document set is split into three sets; one for

training, one for evaluation and one for testing. The split is done to prevent the GP from over-fitting itself to one document set, thus not producing accurate results. The result of their experiment is that they were able to discover a ranking function that outperformed Okapi by a significant percentage. One of the intriguing results of the study is that the GP framework discovers some commonly known ranking functions such as TFIDF.

The research in [40] further expands on the experiment from [6]. It is no longer a nameless GP framework; Fan names the system ARRANGER (Automatic Rendering of Ranking Functions by Genetic Programming) [6]. Fan et. al. goes on to propose using the ARRANGER system and pseudo relevance feedback (Fan refers to the technique as "blind feedback") to gain further performance improvements.

The researchers from [40] present results from another experiment based on the genetic framework they propose. Part of the results, though, consist of creating individual ranking functions for each query, and then comparing them to Okapi and TFIDF. In this portion of the experiment when using short queries they were able to gain a increase in average precision by 16.19% and 10.71% against PTFIDF and Okapi respectively[5]. When using longer queries they were able to perform 32.97% and 17.01% greater. This portion of the experiment seems frivolous. It is unfair to create a function for each query, and it seems to be unreasonable in a normal IR environment. Also, TREC does provide long queries, which can range between seventeen and ninety words. Rarely do search engines encounter queries of this size. Also the researchers removed fifteen queries out of the set of fifty, due to the queries not having a significant number of relevant documents (they all had less than four relevant documents). The more important portion of the experiment though is using the genetic programming framework to create a consensus ranking function, one that will work well over all fifty queries. The results of this experiment are when dealing with short queries the ranking function discovered was able to out perform Okapi and PTFIDF 3.13%

$$\frac{\log\left(tf \times \left(tf\_avg + \frac{tf}{\log(tf^2 \times tf\_avg)} + \frac{tf \times N}{df} \times \frac{tf\_avg \times (tf\_doc\_max + n)}{df}\right)\right)}{n + 2 \times tf\_doc\_max + 0.373}$$

Figure 3.5: Best Ranking Function Discovered by Fan et. al. ARRANGER [6]

and 10.75% in average precision [5]. On long queries the gain in average precision grew to 18.4% and 3.13% against PTFIDF and Okapi respectively. Figure 3.5 below shows the best ranking function that was discovered by the genetic programming framework.

Fan et. al. also studied the affects of a variety of fitness functions when using genetic programming to discover ranking functions [41]. In early experiments Fan and his colleagues limited the ranking function to average precision, and only considered the top twenty documents in evaluation fitness. In this article the fitness functions take into consideration the order of the documents. Relevant documents should be higher in the order than the non-relevant documents. Unfortunately relevancy judgments provided by TREC do not provide a rank of relevance order for queries, only whether a document is relevant or not. The idea of this research [41] is, for example, if only fifteen relevant documents exist for a query, the first fifteen of the documents retrieved should be relevant and the last five of the twenty should not be. Relevant and non-relevant documents should not be mixed throughout the returned set; relevant documents first, non-relevant documents last. This is referred to as precision. The authors state why they view this fitness evaluation to be of importance due to the notion that "the utility of a relevant document decrease with ranking order" [41]. The authors of [41] propose four new fitness functions based on this concept. Their experiment also contains three other fitness functions: average precision, the CHK fitness function developed by Chang and Kwok and the LGM fitness function created by Lopez-Puljate [41]. The corpus used for the experiment was the TREC 10GB collection from 2000. The corpus was randomly split into training, validation and test sections. The

results were measured by the average-precision and the precision of the top ten hits. The results of the experiment show that three of the four fitness functions created by the authors were able to create ranking functions that outperformed Okapi BM24. These three ranking functions also outperform PAVG and CHK as fitness functions. The LGM fitness function was unable to be used in creating a ranking function that outperforms Okapi BM25, as was one of the fitness functions the authors propose [41].

3.6 Personalized Search

Personalized search is user specific unlike traditional search. The promise of personalized search has always been results that are tailored to the individual. This would be accomplished by developing a model or a knowledge of a user. Over time the software would be able to recognize preferences, and incorporate that into the ranking. Currently a semi-personalized search is available, but often it requires the user to fill out a survey, and often the results are more tailored to the person's sex, location, income, and age. These qualities do not encompass who a person is. Personal search is a difficult concept to implement, the system must learn about a person, and then apply that information to the ranking of documents.

In the paper, "Context in Web Search," Steve Lawrence makes a distinction between a document being valuable as opposed to simply being relevant. Lawrence states that a documents value "depends on the context of the query - for example, the education, interests and previous experience of a user along with information about the current request" [42]. This idea is similar to Mizzaro's take on relevance which must consider the user's background and needs, even if the user is not fully knowledgeable of those needs. Other researchers have referred to the traditional results returned by search engines as "consensus relevant," and the results of a personalized system as "personal relevancy" [43]. Lawrence

goes on to describe Inquirius2, a search engine which requests the user to select a context for their query. The example given in [42] is that of a user searching for a document on "machine learning" might want to limit the context of the search to research articles. Inquirius2 requires explicit information from the user, but Lawrence also mentions the Watson project that does not require such information. The Watson project is able to observe the documents currently opened and being edited, and uses information gained from those documents. The query is modified to include this new information prior to being submitted to a search engine. Lawrence goes on to define a personalized search engine as "a search engine that knows all your previous requests and interests, and uses that information to tailor results" [42]. Also noted is that a personalized search engine will not return the same results to a query for different users, and also over time the results a user received to a past query may differ from the results they receive to the same query in the future.

The Haystack project [44] is an attempt to create a personalized search engine that does not require the user to explicitly state context. The Haystack search engine does not focus on the web, but is concerned with the user's desktop system. Haystack is capable of observing the user's interaction with the system through a variety of proxies (web and email). By using a proxy, Haystack can take into consideration temporal effects with relation to the user's interaction with web documents. The longer a user spends at a particular site, the more likely the content of that site interests the user. This information can be used to more accurately return results to a user's query for information stored on their system.

The primary concern of a personalized search engine must be the way in which they model a user's interest. Tandujaja and Mui take issue with the schemes most personalized search engines use to store user info, and refer to it in jest as "a bag of words" [45]. The authors point out that the "bag of word" approach does not provide a proper context for the words, and could be confused due to synonyms. The authors give the example of the word

"rose", which could easily be applied to flowers, or possible wine [45]. The "bag of words" approach provides no means of determining which. They also comment that this approach focuses on the user's likes, and does not take into consideration dislikes. Tandujaja and Mui propose Persona, a personalized search system combining filtering and user profiling [45]. Persona uses a tree-coloring technique to create a user profile. The tree is the Open Directory Project (ODP) taxonomy. The Open Directory Project is very similar to Yahoo in its structure. Persona incorporates the HITS algorithm discussed earlier. The Persona system does require explicit relevance feedback from the user. Users are suppose to rate the context, not the individual documents, negatively or positively. This feedback is reflected on the user's tree by a color-coding of the node. This coding reflects the "number of times it has been visited, rated positive, negative and associated URL's" [45]. Once the user enters a quer,y Persona (if the context is found in the user's profile) gives the associated documents more or less weight, depending on the color. If the context is not in the user's profile, Persona tries the ODP taxonomy. It then checks if the context has the same parent as any node in the user's profile, or if it is a child. If so, their weights are adjusted accordingly [45]. This tree based user profile is what makes Persona interesting. The use of a tree, which can easily be compared to a maintained taxonomy makes the user profiling system powerful in the face of synonyms and other ambiguities. Liu et. al. notes that a user profile combined with a general profile or a categorical hierarchy similar to the ODP can be used to better map a user's query to a set of categories, and thus return more relevant results [46]. In their experiment they found the combine system outperforms a simple user profile. There findings show that a user history needs to be put into a context.

Collaborative filtering has primarily been used in recommendation systems in online commerce sites. Often collaborative filtering is referred to as a "word of mouth" systems [47] . An example is Amazon's book recommender service, which uses feedback on books

a user has ordered in the past to find similar users, and recommend books that a similar user has enjoyed that the primary user has yet to purchase [48]. Collaborative filtering has not been fully implemented into ranking functions. The goal of collaborative filtering is to use other people's preferences, determine how similarly they are to the user, and then decide whether a user would be interested in the item.

3.7 Conclusion

The work of Weiguo Fan, Gerald Salton, van Rijsbersen and Mizzaro are pivotal to this thesis on a variety of levels. Salton and van Rijsbergen are the early innovators. They understood the idea that if information can be measured, then similarity can be discovered. Mizzaro took on the difficult task of actually defining relevance, a word that is thrown around far too often in IR literature, with little consideration of what it means. Relevance incorporates more than just a measurement between a document and a query, the user should also be considered when computing relevance. The work of Weiguo Fan and his colleagues showed that GP ideas can be applied to IR. All of these articles, and the work of these scientist specifically, lead me to my idea of using GP to determine the usefulness of history and social preference as a search metric.

CHAPTER 4

RESEARCH DESCRIPTION

> ”If it doesn’t work right, we can always try something else.” - John McCarthy [49]

It is my belief that the next logical step for search engines is towards personalization. If a search engine uses information gained about a user it would be more capable to return relevant results to a query. This personalization should mature over time. The more a user uses the search engine the better the results should be. Also, a search engine should be able to gain this information by observation. A user should not have to answer a survey or explicitly give over information.

The question arises on how best to use the history of a user to return relevant documents. Simply giving documents that a user has visited in the past a higher rank is not enough. Ranking highly documents similar to those that a user has visited in the past could provide a benefit in the usefulness of the returned documents. This technique could be used to determine the topics of interest in the past, and thusly would be beneficial in alleviating ambiguities of language. All documents in the collection could be grouped together based on their similarities. The clusters would form ad-hoc topic categories. Thus, a more useful image of the user could be developed based on the content type of documents they have visited and not simply the document. Based on the clustering hypothesis, if a document in a cluster is relevant to a query, it is probable that other documents in that cluster would be relevant.

I propose that keeping a history of the clusters a user has visited is more advantageous than simply maintaining a document history. Clusters are in essence a grouping of similar

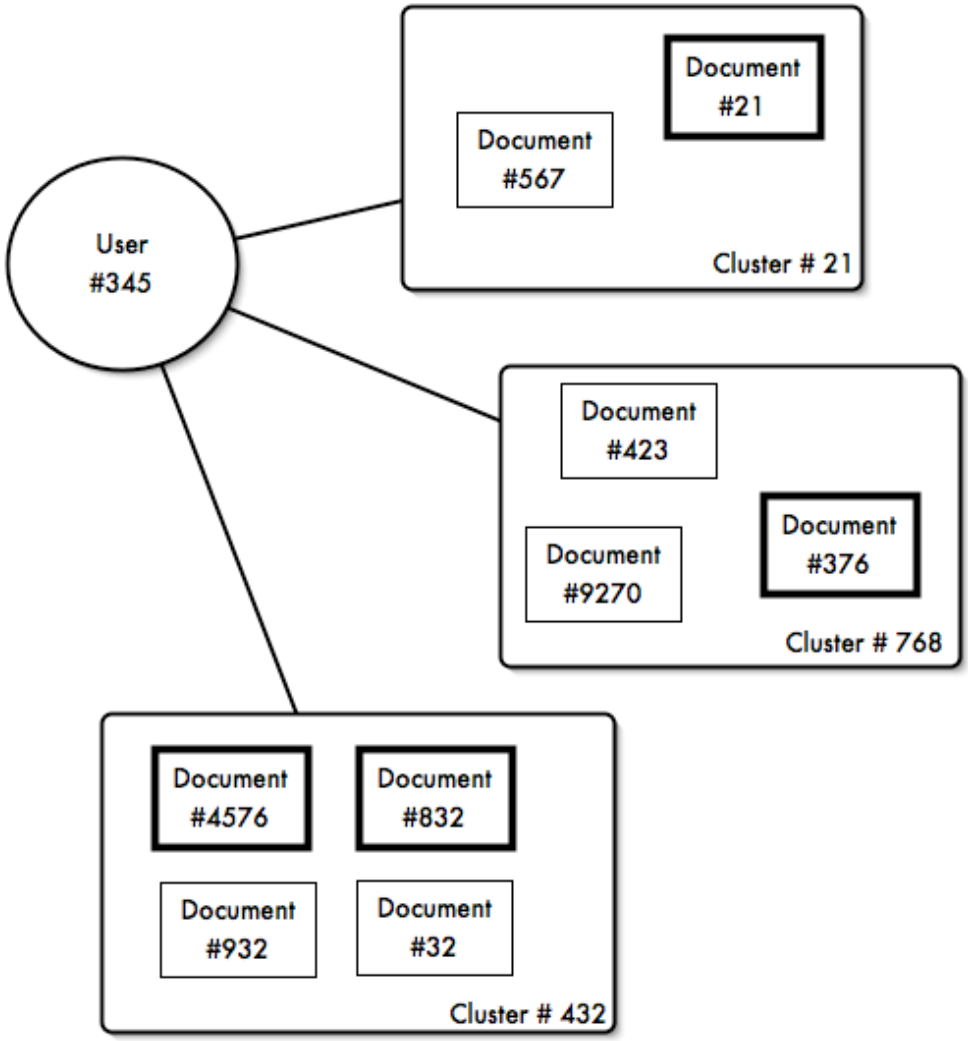


Figure 4.1: Cluster History

documents. Often this type of information is provided by search engines to give the user documents similar to a chosen document if the user so desires. Instead of an added feature, this information should be a direct part of the ranking function. In effect the cluster a user has selected documents from in the past should be used in determining a document's relevance. Figure 4.1 visualizes the way in which the history of the user is recorded as a listing of their cluster history, and the clusters that contain documents that the user has and has not visited. In Figure 4.1, the documents the user visited are in a bold square.

Many current search engines use the popularity of a document to determine its ranking. While this metric has had success, it seems that a better metric would be to reward the document that users similar to the principal user have viewed in the past to be relevant. This would allow community knowledge to participate in the ranking function. It would provide it a social element. Once a user begins to use the search engine, it will discover other users that have searched and found similar pages, and use this information as another metric. Figure 4.2 shows how by utilizing a user's cluster history similar preferences between users could be discovered. In a brief example the users Amy and Barry are similar due to the number of shared clusters between them. Instead of the ranking function limiting its knowledge to just the user's past, it could also use the past of other like-minded users. As IR researcher Keith Stirling wrote, "relevance estimates from past users can be used to rank documents for future users who may submit similar information requests" [50].

This technique is known as "collaborative filtering." It is often utilized in recommendation systems. Amazon uses this technique to recommend books that similar customers have bought in the past or have a favorable feedback of. In a collaborative filtering system, similarity between users is determined in the same manner that similarity between a document and a query are in the vector space model [51]. This collaborative filtering score is calculated by adding up the similarity scores of all the users who have accessed the particular document or object. This function can be normalized to take into consideration the size of the document, and differences in the number of documents visited. This is done so as not to corrupt the results if a user has a particularly vast history, or the document contains many more words than others. The score is an attempt to quantify how similar the users that have visited this document are to the user. Most collaborative filtering functions are not based simply on visiting the document but on the user giving explicit feedback of the usefulness of the document. This can be an impediment to the user. I believe that instead

of the collaborative filtering being based on explicit score, or document history it would be better served to be based on cluster history. This is a natural conclusion based on what can be gained by recording a cluster history versus a document history. The goal of this research is to determine if collaborative filtering based on cluster history is a usable search metric.

The issue is how this personalization fits into the search engine algorithm. Throughout this thesis a variety of search metrics have been discussed: word count, term weights, and link structure. Personal preference should be a metric integrated into the ranking function. I have proposed three new metrics, personal history, personal cluster history, and cluster history of similar users. These metrics need to be integrated into the ranking function, and balanced to return the most relevant results. To balance this ranking function it must be determined which metrics are the most important, and the ranking function must reflect that. Word count will always be an important metric, and it is unthinkable that another metric such as personal history would affect the ranking function more. If that was the case no matter what the user's query, the pages they have already visited would be returned.

The question arises, though, how best to integrate this new metric into a ranking function. The designers of ranking functions must carefully balance the importance of each metric, and fine tune the function. Traditionally it appears this is caused by trial-and-error and gradual refinement. It also seems to involve a large amount of intuitive creation. If a creator proposed a new metric, how could they tell if that metric was successful if the ranking function they have created was created in such a matter? Perhaps the metric was superior, but the integration of the metric into the ranking function was flawed. Genetic programming is a way to determine such things. Researchers have proposed using GP to create better ranking functions, but this research involves taking the current metrics and

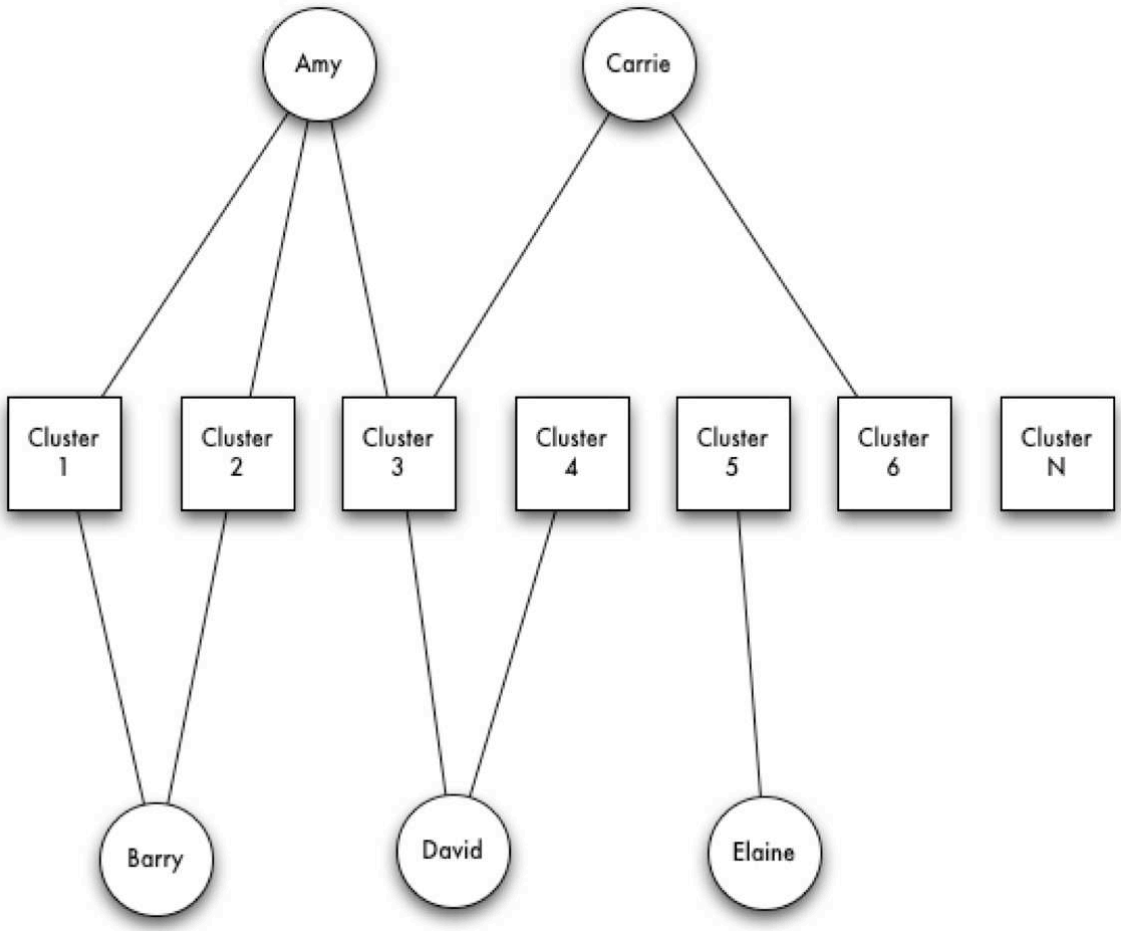


Figure 4.2: Sharing of Clusters Among Users

rearranging them and the operators to produce better functions. As of yet, no one has proposed using this technique to determine whether or not a new metric is valid.

Given the GP all the search metrics, and the new proposed metric, over time the GP would create a ranking function that included the most usable metrics. Poor metrics would be lost in the evolution process that takes place between generations. To be successful all that would be needed would be a document set, queries, relevancy judgements, and user histories. This would be automated testing of the ranking function. The fitness function associated with the GP would give the creator insight into the performance of the function.

CHAPTER 5

EXPERIMENT AND RESULTS

The goal of this experiment was to demonstrate that clustered history of similar users can be integrated into a ranking function and return relevant documents to a query. To accomplish this, a set of documents, a set of queries, and a set of relevancy judgements were needed. Relevancy judgements are the determined relevance between a document and a query, usually of a binary nature; either the document is relevant or it is not. The TREC conference (previously mentioned in Chapter 2) provided all this necessary material for their participants. This resource allowed for the comparison of ranking functions to determine which performs better. I choose to use the TREC dataset because it is the preeminent conference in information retrieval research, and also because most of the past literature in IR that I have studied during this research has exclusively used TREC datasets to test the ability of their retrieval techniques.

The list below provides an overview of the steps that were taken to accomplish the experiment.

1 The dataset was parsed, portered, and stopwords were removed. The dataset was then indexed and stored in a MySQL database. The queries and relevance judgements were also stored in a database.

2 User histories were created based on a portion of the available queries and a portion of the available relevance judgements.

3 The dataset was clustered using a K-means hierarchical clustering algorithm. The number of clusters was chosen to insure that each cluster contained on average ten documents.

4 A GP framework was constructed. The goal of this framework was to create a ranking function that had optimal performance. The GP had terminals available to it consisting of search metrics from the Okapi, INQUERY and Pivoted TFIDF ranking functions. Also included in the possible terminals were the personal history of the user, the clustering history, and a collaborative filtering metric based on the cluster history.

5 Once the optimal ranking function was discovered by the GP framework all the queries were run through using the Okapi, INQUERY and Pivoted TFIDF ranking function. This allowed the performance of the GP created framework to be compared to other common search engine ranking functions.

6 The ranking function that was created using the GP framework was then tested on two other datasets to insure over training did not occur. Okapi, INQUERY and Pivoted TFIDF were also tested to allow for more comparisons.

The TREC HARD data set is referred to as CR99. The data set is part of the Congressional Record (CR). The CR99 data set is 146.6 MB in size. The CR data set is further divided into three data sets CRE, CRS and CRH. These sets are divided so one is for the Senate, one is for the House, and the third is for extension. The CRE, CRS and CRH sets contain 4126, 6339, and 6146 documents respectively. The data sets are not perfect though; all contain duplicate copies of documents. The data set also contains a listing of forty queries, and relevance judgements for each document to each query.

The first step of the experiment was to get the data set into the form of an inverted index. This was accomplished using a Perl script, and the inverted index was stored in a MySQL database. The index is essential to the performance of the ranking function. It is simply a large two key hash table; the first key is the documents ID, the second key

is a term from the document and the value is the word count of said term. Before each term was entered in the database it was portered, and all capitalization and whitespace was removed. Also common stopwords such as “the” were not indexed. TREC contains two different types of queries, long and short. The short query is a three to five word description of the information similar to what users normally enter into a search engine. The long query is a paragraph or more detailed description of what the user is interested in. For the purposes of this experiment only the short queries were used, because that is most similar to what a common online search engine would face. Studies have shown that the average query contains only 2.21 terms, and that less than four percent of queries contain more than six terms [35]. The queries and the relevancy judgements were also stored in a MySQL database for quick access. The queries received the same portering and removing of common words that the index received. These practices are consistent with normal search engine operation.

At this point the documents were clustered using a K-means partitional algorithm. The clustering program was a Perl script that accessed the inverted index stored in MySQL. The source code for the clustering program is available in the appendix. The number of clusters were chosen so that each cluster would average ten documents. Research has shown that having many clusters with a small number of documents is ideal in document retrieving systems [52]. The goal of the clustering was to group similar documents, to in effect create pockets of information. The data sets were clustered individually. The way the K-means partitional algorithm worked was that all documents were placed into a single cluster. Next the largest cluster, which in this case was the initial cluster was split. A sample of all the documents in the cluster were taken and the two documents furthest apart based on vector space similarity were chosen to be the initial members of the two new clusters created by

splitting the largest cluster. Then all the documents in the largest cluster were compared to each of the two to determine which was closer, and thusly joined that cluster.

Unfortunately user histories were not available from TREC. For the purposes of this experiment user histories were pseudo-randomly created. Each user was given between five and twenty of the forty available queries. They then had between two and ten of the relevant documents for each query added to their personal history. Ten users were created. It was briefly considered to have the users also select some non-relevant documents, but it did not seem logical for a user to go out of their way to visit documents not relevant to their query. In a traditional search engine the user is given a small preview of the page, and normally they can determine if the document is relevant to their interest. It should be kept in mind that TREC does not provide a degree of relevance between a document and a query. It is a simple binary distinction; either the document is relevant or it is not. Also for each query there can be up to one hundred relevant documents. The hypothetical user only choose a small number of those documents.

A variety of known ranking functions were tested using the TREC data set. This was done to get a base for performance, with the intent of performing at a higher level. Given a query the ten documents with the highest scores were returned. Then using the relevancy judgements it was determined how many of these documents were relevant, this gave us a percentage. The percentage was then averaged over the entire set of queries. The ranking functions tested were TFIDF, INQUERY, and Okapi BM25. None of these ranking functions consider user history as a component of their ranking criteria.

It is impossible to prove a ranking function is correct. The goal is to simply create an optimal solution. In this experiment the trial and error of creating a ranking function was simply replaced with genetic programming. All the variables in the previous ranking functions were given to the GP framework to create a ranking function. The proposed

personal history functions were available. By using this setup a ranking function could be found that outperformed the commonly accepted functions. Instead of creating a ranking function by hand, and tweaking it until it performs well, I simply chose to let evolution accomplish it.

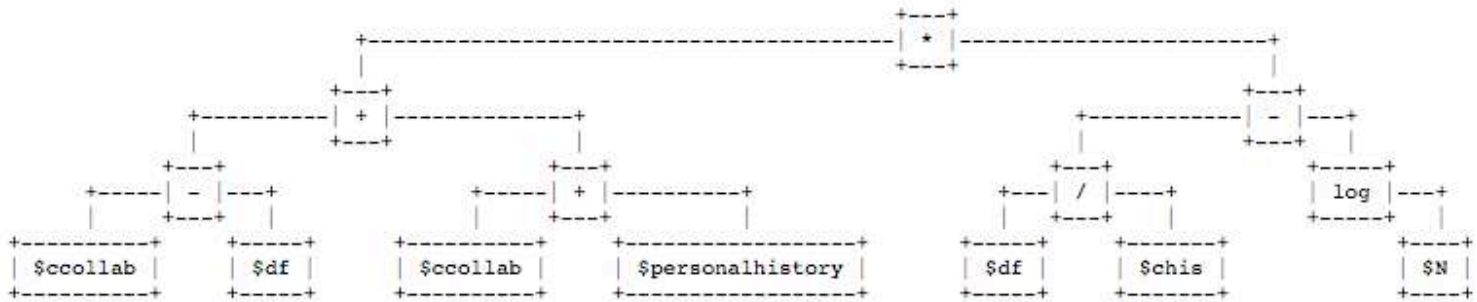
A genetic programming framework was created to develop the most optimal ranking function. The framework was written in the Perl programming language. The choice of using Perl instead of Common Lisp in which Koza developed his GP framework was due to complications in getting Lisp to communicate to a MySQL database. Also Perl contains an eval function, which allows a string to be evaluated as source code. The subroutine to calculate the rank of each document could be created on the fly. This was necessary to determine the fitness of the newly created ranking functions. In the GPIR framework the trees, or functions had a limited number of terminals and operators. The operators consisted of log, addition, subtraction, multiplication, and division. The available operators were term frequency, the number of documents in the collection in which the term is present, average term frequency of the collection, the maximum term frequency, the average term frequency in the collection, the maximum of the number of documents in the collection in which a term is present, number of document in the collection, word count of the document, personal history, cluster history, and the collaborative filtering cluster score describe in the prior section. These metrics were also used by Fan et. al. in their various experiments [5]. The framework could compose the tree consisting of any of these terminals or operators. The max depth of each tree was limited to seven, and the minimum depth was limited to two. This was done to insure the ranking functions didn't become unruly, and possibly unusable, but on the lower side, to insure that the ranking function was an equation and not simply a variable. Certain rules were also defined in the creation of the trees to insure that correct equations were created, to avoid division by zero for example. The GP

framework pseudo-randomly created an initial population of trees, consisting of pseudo-randomly chosen terminals and operators. Once created, the fitness of each of these trees was evaluated. The evaluation uses the tree to return the top ten documents for each of the forty queries provided by TREC. Limiting it to the top ten documents was done because studies have shown that 58% of users do not look past the first ten results [35]. Using the relevance judgements, it was determined how many of these ten documents were relevant to the query. A percentage of relevant documents was recorded for each query, and the fitness was the average of these percentages over all forty queries. On my test machine (G4 1Ghz, 1GB RAM) it took approximately one minute to test each tree, a reasonable time considering that processing each tree was basically submitting forty queries to a search engine. Once an initial population of trees was tested they were ordered according to their fitness. At this point the population evolved into the next generation. The size of the population remained consistent from generation to generation. The top $\sqrt{populationsize} - 1$ functions in the current generation automatically advanced into the next generation without change. Each of these select few were crossovered with each other. Crossover consisted of taking a random portion of one tree, and combining it with a random portion of another tree to produce a new tree. The figure below is an example of two trees being crossovered.

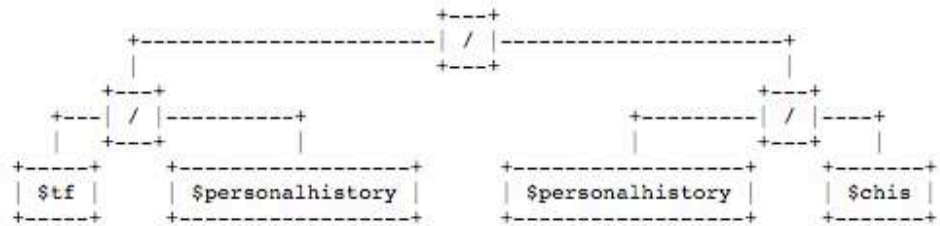
To fill out the remaining members of the population trees were chosen at random and crossovered. This insured that weaker performing members of the population could exist in further generations. The population was evolved for a number of generations. At the end of each generation the best tree was printed out. Also, the average fitness of each population was recorded. This was used to determine if the population became stronger through the steps of evaluation.

For the purpose of this experiment the population size was fifty trees, and with twenty generations. Preliminary runs of the genetic programming framework showed that an initial

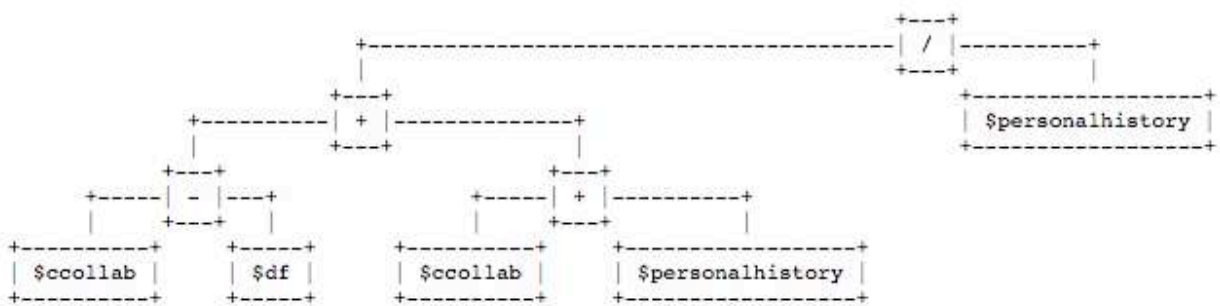
Parent 1



Parent 2



Child Crossover(Parent1 and Parent2)



population size of fifty was sufficient enough to produce a good variety of ranking functions. Having a larger initial population did not produce better results, and the run-time of the experiment was significantly increased. The choice of twenty populations was due to initial observations that showed the optimal ranking function being discovered prior to the tenth generation. The use of twenty was to provide a buffer. The fitness of the trees was only determined against one of the datasets, CRE. Once the GP run was completed the best trees were taken and tested against the other two datasets to insure that over training did not occur. Not all terminals were used in each tree, and just as in biological evolution where unwanted traits are lost, weaker metrics were evolved out of the population. By shared cluster history surviving the process of evolution, it was viewed as a useful metric. Also, comparing the GP created ranking function containing shared cluster history versus common ranking functions performance gains were further observed.

When the experiment began problems with the dataset began to surface. It was known that the dataset had duplicate copies of documents, each with a different document identification code. The issue was that when the ranking function would deem both these documents relevant to a query (after all they were identical) the relevance judgements provided by TREC would only list one of the documents as relevant. This led to the percentage score being low, but since this was a problem the GP ranking function, Okapi, INQUERY, and inverted TFIDF all experienced evenly it allowed the experiment to progress, knowing that it could be shown which ranking function performed the best. Unfortunately these scores, while they allowed rank to be assigned, were not an accurate representation of the ranking function's true performance. In generation four of the run the tree that would be the best performer was discovered. The tree is shown in Figure 5.1. The tree used the cluster collaborative filter score metric, the term frequency of the query terms in the document, the number of documents in the collection and the document length. The metrics personal

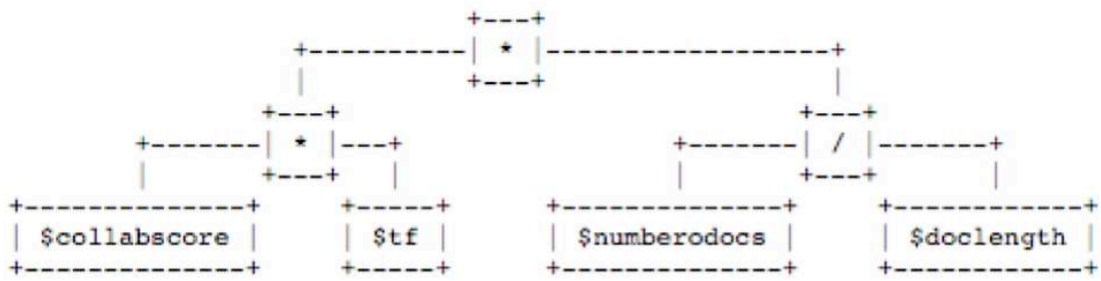


Figure 5.1: Best Tree

history and cluster history, which were simply lists of the documents and the clusters the user had visited in the past, were available, but they did not survive the evolution process. The complete run of the experiment is available in the appendix, including a record of all the trees in the final population and their scores.

Figure 5.2 is a graph containing the best fitness of each generation, and how it progressed over the entire run. Also in the graph is the average fitness of the population which peeked at generation nine, and proceeded to plateau at that point with little change.

The ranking function in Figure 5.1 was developed using the CRE dataset, but to insure no over-training was committed, the function was tested against the other two datasets. Also the INQUERY, Pivoted TFIDF, and Okapi ranking functions were tested against the three data sets to allow a comparison to the GP created function. These ranking functions were simply the equations used to rank the documents and did not take into consideration the use of thesaurus or pseudo-relevance feedback. These are techniques outside of the ranking function that can be further used to improve the quality of results.

As the graph in Figure 5.3 shows the GP created ranking function using the collaborative filtering based on clusters was able to out perform the three other ranking functions on two out of three of the data sets. The function showed the greatest performance gain on the CRE data set in which it was trained, but also performed well on the CRS dataset, which

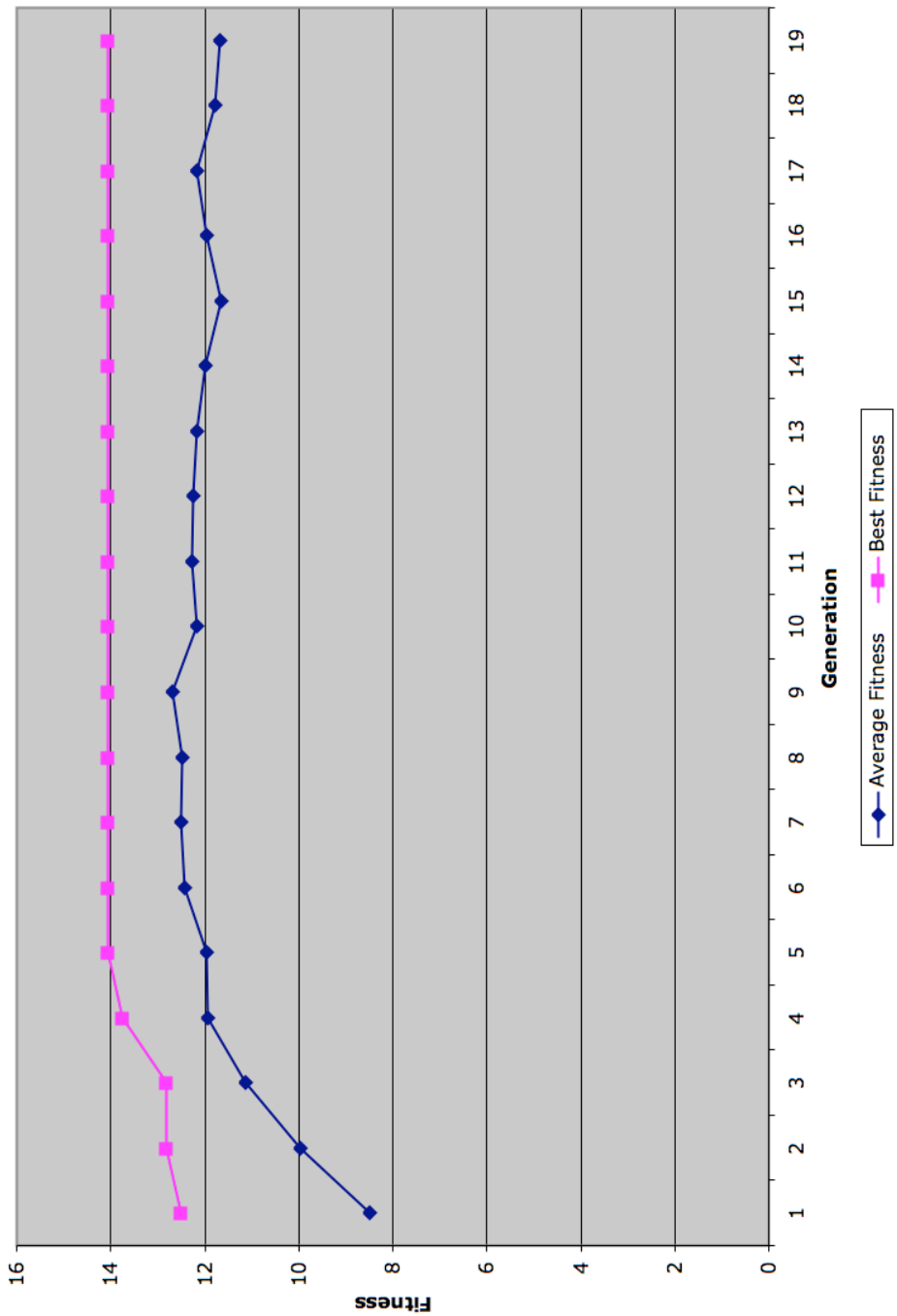


Figure 5.2: Fitness of Trees over GPIR Run

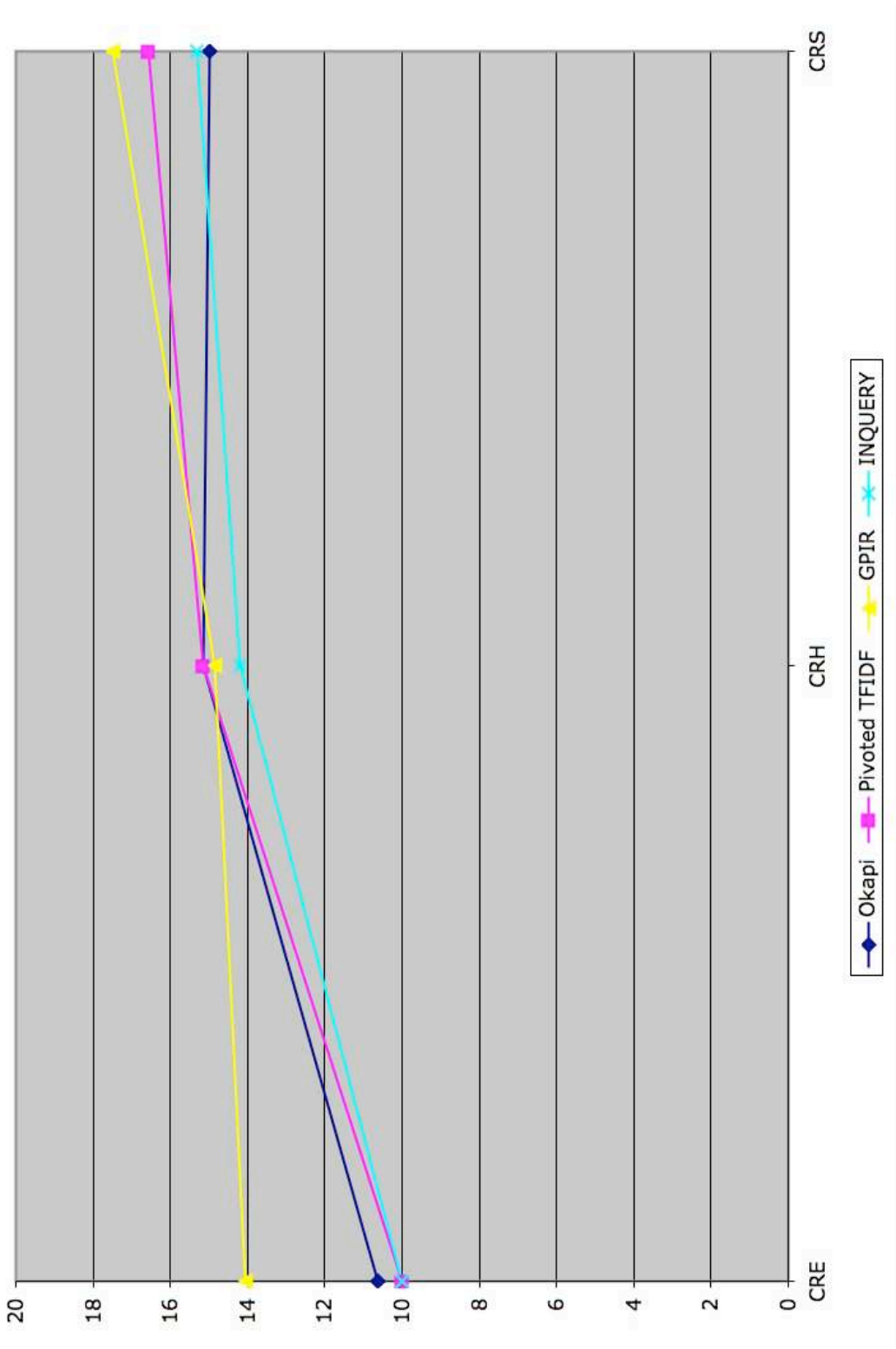


Figure 5.3: Graph of CR99 Performance

	CRE	CRS	CRH
Pivoted TFIDF	10	16.5625	15.16129
INQUERY	10	15.625	14.19
Okapi	10.625	15	15.16129
GPIR	14.0625	17.5	14.8387

Figure 5.4: Chart of Performance

the GP framework had no contact with. The complete source code of the GP framework is provided in the appendix.

Figure 12 is a table containing the scores of each of the ranking functions. The scores were calculated in the same manner as the fitness is in the GP framework.

On two of the three datasets the ranking function created by the GP incorporating the new metric outperformed the more seasoned ranking functions. On the CRH dataset where it placed third it was still close to the top, with Okapi and INQUERY tying in score, and only being roughly .33 above. On average the GP created ranking function outperformed its closest rival Pivoted TFIDF by 1.5. The only significant advantages this ranking function had were the new personalized metric, and being constructed using the genetic programming framework.

CHAPTER 6

CONCLUSION

Information retrieval experts claim that the next great advance will be personalized search, that the search engine will actually know the user. That is not enough. The search engine should also know people like the user. This information helps put a user's interests in a better light.

A metaphor for this new personal search would be if a person went to the bookstore and picked out a book. They knew the basic subject that they were interested in, and they've had past experience buying books, they know authors they've enjoyed, and publishers who create handsome books. Using this information they could make a better purchase than if they simply walked in only knowing the subject matter they were interested in. Consider though if this shopper also had recommendations from their friends, family and colleagues, people of similar intelligence, interest and location. Given this information a customer would be far more likely to be satisfied with the purchase they made.

Search is a product. It is a service. The system that gives the user what they want, with as little hassle as possible will win. For an engine to not use all the information available is foolish. The only question is how to balance all these metrics into an equation that will produce optimal results, without torturing the user through a period of test and fix. The creators of genetic programming envisioned such problems when they created the concept.

Once a creator has proposed a new metric, how can they determine whether or not it actually works? Even if the metric has merit, how do they implement it into the ranking function? GP provides a way for a designer to discover how to integrate this new metric into a ranking function. Collaborative filtering can be helpful in the process or returning useful documents to a user's query.

With a much larger data set, access to user's queries and their subsequent document selections to said queries, a better ranking function could be designed with the help of genetic programming, and also new metrics could be tested. If with a new metric, the fitness of the final ranking function increases then it can be deemed that metric is useful in the ranking the documents.

Once a new metric has been proposed the question of how to best integrate it into the ranking function, and also if the proposed metric is useful will arise. The solution to both questions is the use of genetic programming. Considering the impossibility of proving a search metric works, the best alternative is to determine a function that incorporates the metric that outperforms the best performing ranking function not containing the metric.

The user is the ultimate determiner of relevancy. Two users enter in the exact same query and the document that the first user believes is relevant the second user might disagree. The inexact nature of this field forces researchers to search for the most optimal solution, since a totally correct solution seems to be unlikely.

Given more information, the performance of the ranking function can only improve. With larger data sets, more queries, more relevancy judgements, and more user histories the genetic programming frameworks' ability to create a successful ranking function will be helped significantly.

Consider the possibility of a search engine that constantly changes. Slowly over time the ranking function is adapted to give better results. The ranking function used two years prior, will not be the same as the current because of the knowledge gained. A genetic programming application would run concurrently beside the search engine, constantly attempting to discover a better ranking function. The genetic program is always there to allow a designer to easily add a new metric into the fold, and then discover if the metric improves the ranking process.

When given the choice of which metrics to use the genetic programming framework chose, term frequency, the collaborative score proposed, number of documents and the length of the document. All of the other metrics were deemed to be unneeded to produce the results, or possibly harmful. The ranking function created by the GP framework had the best performance increase over the other ranking functions on the dataset that it was tested. This was to be expected. On the CRH dataset the GP created ranking function was bested by Okapi and PivotedTFIDF, yet the distance was very minimal. The GP created function well outperformed all the functions on the CRS dataset. The GP function had only one real advantage over the other functions, and that was the ability to use the collaborative filtering metric I earlier proposed. If the GP had been allowed to train on all three datasets it would have been able to improve the score on the CRH dataset. The point of training it on only one of the data sets was to support the notion that it would perform well on other datasets. The genetic programming framework was a tool. The single day run of the genetic programming framework was able to outperform a variety of ranking function that were created over months using heuristic techniques. The run was able to determine which metrics were usable by their appearance in the final ranking function. If these metrics were without merit they would have been quickly removed during the evolution process.

The best ranking function was discovered after the fourth generation. Longer runs, and a larger initial population would not have improved the final performance of the ranking function. The best way to improve the results would have been to have had more queries and more relevancy judgements. That would have allowed the GP framework to perfect the ranking function that much more.

On average the ranking function created by the GP framework, which included the collaborative filtering based on clustering metric, performed at 15.467. This is compared to PivotedTFIDF, OKAPI, and INQUERY which performed at 13.9, 13.587 and 13.271

respectively. The addition of this metric caused a significant improvement in the performance of the ranking function. The performance gain was not due to over training, as demonstrated by good performance on the other two datasets which the GP had no contact with.

The research conducted in this thesis has shown that genetic programming can be used to determine if a newly proposed information retrieval metric (collaborative filtering based on cluster history) is effective. The framework also integrates this new metric into the ranking function in an optimal way, and fully automates the heuristic process of building a ranking function. A technique such as this can allow researchers to gain insight into the performance of their metrics, and also significantly shorten the time it takes to create these search engine ranking functions.

CHAPTER 7

FUTURE WORK

In the future I hope to apply the techniques proposed into a true online search engine. A search engine that could adjust the ranking function over time would be a significant improvement over current implementations. With larger datasets, more users, and thusly more queries and relevance judgements, an even more significant improvement could be had over the modest gains demonstrated in this text. With nothing more than twenty users, and a dataset that did not exceed five thousand documents I was able to create a system that could find the optimal ranking function.

Some might contend that adding larger numbers of documents and larger number of user's would not help the process, that the computation needed to test each function on each query would become too much. It should be viewed as a benefit, not a challenge. The dataset could be randomly divided, also the users. Each time a function was created it would test it against a portion of the queries, a portion of the users and a portion of the documents.

I imagine the genetic programming framework running in the background able to integrate a better ranking function into the search engine if it is found. Also new metrics could be quickly added to the search, without prior testing to decide if they are actually useful. The framework will determine their usefulness.

In the future I also hope to look into new metrics that further use community information to improve search engine results. Hopefully this research will lead me to a greater understanding of what can be inferred from observation of a user's search history, and how these observations can best be integrated into the search process while maintaining the user's privacy.

BIBLIOGRAPHY

[1] J. R. Koza, *Genetic Programming : On the Programming of Computers by Means of Natural Selection*. MIT Press, 1993.

[2] S. Mizzaro, "How many relevances in information retrieval?," *Interacting with Computers*, vol. 10, no. 3, pp. 303–320, 1998.

[3] O. R. Zaïane, "From resource discovery to knowledge discovery on the internet," tech. rep., TR 1998-13, Simon Fraser University, 1998.

[4] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: A review," *ACM Computing Surveys*, vol. 31, no. 2, pp. 264–323, 1999.

[5] W. Fan, M. D. Gordon, and P. Pathak, "A generic ranking function discovery framework by genetic programming for information retrieval," *Information Processing and Management*, vol. 40, no. 4, pp. 587–602, 2004.

[6] W. Fan, M. Luo, L. Wang, W. Xi, and E. A. Fox, "Tuning before feedback: combining ranking discovery and blind feedback for robust retrieval," in *SIGIR '04: Proceedings of the 27th annual international conference on Research and development in information retrieval*, pp. 138–145, ACM Press, 2004.

[7] Google, "http://www.google.com," Date Accessed - June 2005.

[8] D. Sullivan, "Search engine size wars v erupts," November 2004. http://blog.searchenginewatch.com/blog/041111-084221.

[9] D. Sullivan, "Searches per day," February 2003. http://searchenginewatch.com/reports/article.php/2156461.

[10] Yahoo, "http://www.yahoo.com," Date Accessed - June 2005.

[11] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[12] V. Bush, "As we may think," *The Atlantic Monthly*, July 1945.

[13] C. N. Mooers, "The next twenty years in information retrieval: Some goals and predictions," *American Documentation*, vol. 11, pp. 229–236, March 1960.

[14] C. J. Van Rijsbergen, *Information Retrieval, 2nd edition*. Dept. of Computer Science, University of Glasgow, 1979.

[15] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[16] J. Gemmell, G. Bell, R. Lueder, S. Drucker, and C. Wong, "Mylifebits: fulfilling the memex vision," in *MULTIMEDIA '02: Proceedings of the tenth ACM international conference on Multimedia*, pp. 235–238, ACM Press, 2002.

[17] W. Sonnenreich and T. Macinta, *Web Developer.com Guide to Search Engines*. Wiley, Febuary 1998.

[18] NIST, "Text retrieval conference (trec) home page," Date Accessed - June 2005. http://trec.nist.gov.

[19] J. R. Koza, "Genetic programming," in *Encyclopedia of Computer Science and Technology* (J. G. Williams and A. Kent, eds.), pp. 29–43, Marcel-Dekker, 1998.

[20] H. Hirsh, W. Banzhaf, J. R. Koza, C. Ryan, L. Spector, and C. Jacob, "Genetic programming," *IEEE Intelligent Systems*, vol. 15, pp. 74–84, May - June 2000.

[21] S. G. Steinberg, "Seek and ye shall find (maybe)," *Wired*, vol. 4, May 1996.

[22] M. Cafarella and D. Cutting, "Building nutch: Open source searches per day," *Queue*, vol. 2, no. 2, pp. 54–61, 2004.

[23] A. Patterson, "Why writing your own search engine is hard," *Queue*, vol. 2, no. 2, pp. 48–53, 2004.

[24] A. L. Penenberg, "Search rank easy to manipulate," March 2005. http://www.wired.com/news/culture/0,1284,66893,00.html.

[25] R. M. Losee and J. Lewis Church, "Are two document clusters better than one? the cluster performance question for information retrieval: Brief communication," *Journal of the American Society for Information Science and Technology*, vol. 56, pp. 106–108, January 2005.

[26] A. Leuski, "Evaluating document clustering for interactive information retrieval," in *CIKM '01: Proceedings of the tenth international conference on Information and knowledge management*, pp. 33–40, ACM Press, 2001.

[27] Y. Zhao and G. Karypis, "Evaluation of hierarchical clustering algorithms for document datasets," in *CIKM '02: Proceedings of the eleventh international conference on Information and knowledge management*, pp. 515–524, ACM Press, 2002.

[28] A. Leouski and W. Croft, "An evaluation of techniques for clustering search results," tech. rep., IR-76, Department of Computer Science University of Massachusetts, Amherst, 1996.

[29] S. Kuchinskas, "Peeking into google," March 2005. http://www.internetnews.com/xSP/article.php/3487041.

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

[31] G. Salton and C. Buckley, "Improving retrieval performance by relevance feedback," *Journal of the American Society of Information Science*, vol. 41, pp. 286–297, June 1990.

[32] D. Harman, "Relevance feedback revisited," in *SIGIR '92: Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 1–10, ACM Press, 1992.

[33] M. S. Khan and S. Khor, "Enhanced web document retrieval using automatic query expansion," *Journal of the American Society for Information Science and Technology*, vol. 55, pp. 29–49, January 2004.

[34] S. E. Robertson, S. Walker, M. Hancock-Beaulieu, A. Gull, and M. Lau, "Okapi at trec-2," in *Text REtrieval Conference*, pp. 21–30, 1992.

[35] B. J. Jansen, A. Spink, and T. Saracevic, "Real life, real users, and real needs: a study and analysis of user queries on the web," *Information Processing and Management: an International Journal*, vol. 36, no. 2, pp. 207–227, 2000.

[36] J. P. Callan, W. B. Croft, and S. M. Harding, "The INQUERY retrieval system," in *Proceedings of DEXA-92, 3rd International Conference on Database and Expert Systems Applications*, pp. 78–83, 1992.

[37] A. Acharya, M. Cutts, J. Dean, P. Haahr, M. Henzinger, U. Hoelzle, S. Lawrence, K. Pfleger, O. Sercinoglu, and S. Tong, "United states patent application 20050071741," 2003.

[38] S. Dominich and A. Skrop, "Pagerank and interaction information retrieval: Research articles," *Journal of the American Society for Information Science and Technology*, vol. 56, no. 1, pp. 63–69, 2005.

[39] WikiQuote, "Charles darwin - wikiquote," Date Accessed - June 2005. http://en.wikiquote.org/wiki/Charles_Darwin.

[40] W. Fan, M. D. Gordon, and P. Pathak, "Discovery of context-specific ranking functions for effective information retrieval using genetic programming," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 4, pp. 523–527, 2004.

[41] W. Fan, E. A. Fox, P. Pathak, and H. Wu, "The effects of fitness functions on genetic programming-based ranking discovery for web search," *Journal Of The American Society For Information Science And Technology (JASIST)*, vol. 55, pp. 628–636, 2004.

[42] S. Lawrence, "Context in web search," *IEEE Data Engineering Bulletin*, vol. 23, no. 3, pp. 25–32, 2000.

[43] J. Pitkow, H. Schütze, T. Cass, R. Cooley, D. Turnbull, A. Edmonds, E. Adar, and T. Breuel, "Personalized search," *Commun. ACM*, vol. 45, no. 9, pp. 50–55, 2002.

[44] E. Adar, D. Kargar, and L. A. Stein, "Haystack: per-user information environments," in *CIKM '99: Proceedings of the eighth international conference on Information and knowledge management*, pp. 413–422, ACM Press, 1999.

[45] F. Tanudjaja and L. Mui, "Persona: A contextualized and personalized web search," in *In Proceedings of the 35 Annual Hawaii International Conference on System Sciences (HICSS'02)*, HICSS, 2002.

[46] F. Liu, C. Yu, and W. Meng, "Personalized web search for improving retrieval effectiveness," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 1, pp. 28–49, 2004.

[47] U. Shardanand and P. Maes, "Social information filtering: Algorithms for automating "word of mouth"," in *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, vol. 1, pp. 210–217, ACM Press, 1995.

[48] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: item-to-item collaborative filtering," *Internet Computing, IEEE*, vol. 7, pp. 76–80, Jan-Feb 2003.

[49] WikiQuote, "John mccarthy - wikiquote," Date Accessed - June 2005. http://en.wikiquote.org/wiki/John_McCarthy.

[50] K. H. Stirling, "On the limitations of document ranking algorithms in information retrieval," in *SIGIR '81: Proceedings of the 4th annual international ACM SIGIR conference on Information storage and retrieval*, pp. 63–65, ACM Press, 1981.

[51] I. Soboroff and C. Nicholas, "Collaborative filtering and the generalized vector space model (poster session)," in *SIGIR '00: Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 351–353, ACM Press, 2000.

[52] A. Griffiths, H. C. Luckhurst, and P. Willett, "Using interdocument similarity information in document retrieval system," *Journal of the American Society for Information Science and Technology*, vol. 37, pp. 3–11, 1986.

APPENDICES

APPENDIX A

GENETIC PROGRAMMING FRAMEWORK EXPERIMENT RUN

```
Connected to DB
Inverted Index Loaded from DB
Queries Loaded from DB
Relevance Judgements Loaded from DB
User Profiles Loaded from DB
Building Similarity Matrix
Loaded Cluter Information
Number of Documents 4125

Generation 1
12.5
$collabscore * $tf
Average Fitness of Generation 8.49375

Generation 2
12.8125
$collabscore * $dfmaxcol
Average Fitness of Generation 9.95710784313725

Generation 3
12.8125
log $tfavgcol + $collabscore - $tfavg / $tfavgcol -
   log $doclengthavg + $dfmaxcol * $tf + $clhistory /
   $numberodocs / $doclength / $doclength / $dfmaxcol
Average Fitness of Generation 11.1397058823529

Generation 4
13.75
$collabscore * $tf * $collabscore * $dfmaxcol *
   $numberodocs / $doclength
Average Fitness of Generation 11.9362745098039

Generation 5
14.0625
$collabscore * $tf * $numberodocs / $doclength
Average Fitness of Generation 11.9607843137255

Generation 6
14.0625
```

```
$collabscore * $tf * $collabscore / $collabscore /
   $doclength
Average Fitness of Generation 12.4203431372549

Generation 7
14.0625
$numberodocs * $collabscore * $tf * $numberodocs /
   $doclength
Average Fitness of Generation 12.5122549019608

Generation 8
14.0625
$collabscore / $collabscore * $tf * $numberodocs /
   $doclength * $collabscore
Average Fitness of Generation 12.4877450980392

Generation 9
14.0625
$collabscore * $tf * $collabscore / $collabscore /
   $doclength
Average Fitness of Generation 12.6899509803922

Generation 10
14.0625
$collabscore * $tf * $collabscore / $collabscore /
   $doclength
Average Fitness of Generation 12.156862745098

Generation 11
14.0625
$collabscore * $tf * $collabscore / $collabscore /
   $doclength
Average Fitness of Generation 12.2732843137255

Generation 12
14.0625
$collabscore * $tf * $numberodocs / $doclength *
   $numberodocs * $numberodocs
Average Fitness of Generation 12.2549019607843

Generation 13
14.0625
$collabscore * $tf * $collabscore / $collabscore /
   $doclength
Average Fitness of Generation 12.156862745098

Generation 14
```

```
14.0625
$numberodocs * $collabscore * $tf * $numberodocs /
   $doclength * $numberodocs * $numberodocs
Average Fitness of Generation 11.9791666666667

Generation 15
14.0625
$collabscore * $tf * $collabscore / $collabscore /
   $doclength * $numberodocs
Average Fitness of Generation 11.6421568627451

Generation 16
14.0625
$collabscore * $tf * $numberodocs / $doclength *
   $numberodocs * $numberodocs
Average Fitness of Generation 11.9669117647059

Generation 17
14.0625
$numberodocs * $numberodocs * $numberodocs *
   $collabscore * $tf * $numberodocs / $doclength *
   $numberodocs
Average Fitness of Generation 12.1629901960784

Generation 18
14.0625
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs
Average Fitness of Generation 11.7708333333333

Generation 19
14.0625
$numberodocs * $numberodocs * $numberodocs *
   $collabscore * $tf * $numberodocs / $doclength *
   $numberodocs * $numberodocs
Average Fitness of Generation 11.6666666666667


Final Generation
-----------------------------------------
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs *
   $numberodocs * $numberodocs
Fitness 14.0625
-----------------------------------------
$numberodocs * $numberodocs
Fitness 9.375
```

```
$numberodocs * $numberodocs * $numberodocs *
   $collabscore * $tf * $numberodocs / $doclength *
   $numberodocs * $numberodocs
Fitness 14.0625
```

```
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs
Fitness 14.0625
```

```
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs *
   $numberodocs
Fitness 14.0625
```

```
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs *
   $numberodocs
Fitness 14.0625
```

```
$collabscore * $tf * $numberodocs / $doclength *
   $numberodocs * $numberodocs
Fitness 14.0625
```

```
$tf * $numberodocs
Fitness 9.375
```

```
$numberodocs * $numberodocs
Fitness 9.375
```

```
$numberodocs * $collabscore * $tf * $numberodocs /
   $doclength * $numberodocs * $numberodocs *
   $numberodocs
Fitness 14.0625
```

```
$numberodocs * $numberodocs
Fitness 9.375
```

```
$collabscore * $tf * $numberodocs
Fitness 12.5
```

```
$numberodocs * $collabscore * $tf * $numberodocs /
   $doclength * $numberodocs * $numberodocs *
   $numberodocs
Fitness 14.0625
```

```
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs *
   $numberodocs * $numberodocs
Fitness 14.0625
```

```
$numberodocs * $numberodocs
Fitness 9.375
```

```
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs *
   $numberodocs * $numberodocs
Fitness 14.0625
```

```
$numberodocs * $numberodocs
Fitness 9.375
```

```
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs
Fitness 14.0625
```

```
$numberodocs * $numberodocs
Fitness 9.375
```

```
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs *
   $numberodocs
Fitness 14.0625
```

```
$numberodocs * $numberodocs
Fitness 9.375
```

```
$numberodocs * $collabscore * $tf
Fitness 12.5
```

```
$numberodocs * $numberodocs
Fitness 9.375
```

```
$numberodocs * $numberodocs
Fitness 9.375
```

```
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs *
   $numberodocs
Fitness 14.0625
```

```
$numberodocs * $numberodocs
Fitness 9.375
```

```
$collabscore * $tf * $numberodocs / $doclength *
   $numberodocs * $numberodocs
Fitness 14.0625
```

```
$numberodocs / $doclength * $numberodocs *
   $numberodocs * $collabscore * $tf * $numberodocs /
   $doclength * $numberodocs * $numberodocs
Fitness 13.125
```

```
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs *
   $numberodocs
Fitness 14.0625
```

```
$numberodocs * $numberodocs
Fitness 9.375
```

```
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs
Fitness 14.0625
```

```
$numberodocs * $numberodocs
Fitness 9.375
```

```
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs *
   $numberodocs
Fitness 14.0625
```

```
$numberodocs * $numberodocs
Fitness 9.375
```

```
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs
Fitness 14.0625
```

```
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs *
   $numberodocs
Fitness 14.0625
```

```
$numberodocs * $numberodocs * $numberodocs *
   $collabscore * $tf * $numberodocs / $doclength *
   $numberodocs * $numberodocs
Fitness 14.0625
```

```
$numberodocs * $numberodocs * $numberodocs
Fitness 9.375
_____________________________________
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs *
   $numberodocs * $collabscore * $tf * $numberodocs /
   $doclength
Fitness 13.75
_____________________________________
$numberodocs * $numberodocs * $numberodocs *
   $collabscore * $tf * $numberodocs / $doclength *
   $numberodocs * $numberodocs
Fitness 14.0625
_____________________________________
$numberodocs * $numberodocs * $numberodocs *
   $collabscore * $tf * $numberodocs / $doclength *
   $numberodocs * $numberodocs
Fitness 14.0625
_____________________________________
$numberodocs * $numberodocs * $numberodocs *
   $collabscore * $tf * $numberodocs / $doclength *
   $numberodocs * $numberodocs
Fitness 14.0625
_____________________________________
$numberodocs * $collabscore * $tf * $numberodocs /
   $doclength * $numberodocs * $numberodocs *
   $numberodocs
Fitness 14.0625
_____________________________________
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs *
   $numberodocs
Fitness 14.0625
_____________________________________
$numberodocs * $collabscore * $tf * $numberodocs /
   $doclength * $numberodocs
Fitness 14.0625
_____________________________________
$numberodocs * $numberodocs * $collabscore * $tf *
   $numberodocs / $doclength * $numberodocs *
   $numberodocs
Fitness 14.0625
_____________________________________
$numberodocs * $numberodocs
Fitness 9.375
_____________________________________
$numberodocs * $numberodocs
```

```
Fitness 9.375
_____________________________________________

$numberodocs * $numberodocs * $collabscore * $tf *
    $numberodocs / $doclength * $numberodocs *
    $numberodocs
Fitness 14.0625
_____________________________________________

$numberodocs * $numberodocs
Fitness 9.375
_____________________________________________

$numberodocs * $numberodocs * $collabscore * $tf *
    $numberodocs / $doclength * $numberodocs *
    $numberodocs
Fitness 14.0625


Best FINAL Tree
_____________________________________________

$collabscore * $tf * $numberodocs / $doclength
Best Fitness: 14.0625
```

APPENDIX B

GENETIC PROGRAMMING FRAMEWORK SOURCE CODE

```
#! /opt/local/bin/perl -w

#GPIR Framework
#Able to create ranking functions, test them for their
    ability
#and evolve the ranking functions over generations

use DBI;
use porter;
use strict;
use warnings;
use Tree::Binary;
use Tree::Binary::Search;
use Tree::Visualize;
use Tree::Binary::Visitor::InOrderTraversal;
use Tree::Binary::Visitor::
   InOrderTraversalExpressionTree;
use Tie::Hash::StructKeyed;
use Carp;
use Storable;
$SIG{__WARN__} = \&carp;
$SIG{__DIE__} = \&confess;


use vars qw(%collabscoretable @stopwords %clusterinfo
    %sim %clusterhistory %population $tfavgcol
   $doclengthavg %nkhash $popsize @terminals
   @operators @userprofile $dbh %queries %
   relevancejudgement %docscores $query %invertedindex
    $numberodocs %tfmaxhash %dfmaxcolhash %
   doclengthhash %tfavghash);
@operators = ("+", "-", "*", "/", "log", "null");
@terminals = ('$tf', '$Nk', '$tfavgcol', '$tfmax', '
   $tfavg', '$dfmaxcol', '$numberodocs', '$doclength'
    , '$doclengthavg', '$phistory', '$collabscore', '
   $clhistory', 'null');

my $dsn = 'DBI:mysql:thesis:localhost';
my $db_user_name = 'root';
my $db_password = 'dragon';
my ($id, $password);
```

```
$dbh = DBI->connect($dsn, $db_user_name, $db_password)
    ;
print "Connected to DB\n";

loadInvertedIndex();
print "Inverted Index Loaded from DB\n";
loadQueries();
print "Queries Loaded from DB\n";
loadRelevanceJudgements();
print "Relevance Judgements Loaded from DB\n";
loadUserProfiles();
print "User Profiles Loaded from DB\n";
buildSimMatrix(1);
print "Building Similarity Matrix \n";
loadClusterInfo();
print "Loaded Cluter Information \n";


my @docids = keys(%invertedindex);
$numberodocs = $#docids;
print "Number of Documents ".$numberodocs."\n";
@docids = ();
$tfavgcol = tfavgcol();
$doclengthavg = doclengthavg();
$popsize = $ARGV[0];
my $numgeneration = $ARGV[1];


tie %population, 'Tie::Hash::StructKeyed';
createInitialPopulation();

my $bestfitness = 0;
my $besttree = ();

for(my $x = 1; $x < $numgeneration; $x++) {
        print "\nGeneration ".$x."\n";
        #Lets Print The Population Here
        my $holder = getBestFromPopulation();       #
            PrintBest Member of Current Population
        print "Best Fitness ".$population{$holder}."\n
            ";
        printTree($holder);
        if ($population{$holder} > $bestfitness) {
                $besttree = $holder;
                $bestfitness = $population{$holder};
        }
```

```
        print "Average Fitness of Generation ".
            avgFitnessPopulation()."\n\n";
        evolvePopulation();
}
        print "Final Generation \n";
        printPopulation();
        print "Best FINAL Tree\n--------------------\n";
        printTree($besttree);
        print "Best Fitness: ".$bestfitness."\n";

        #Output The Graphs Here

exit;

sub createInitialPopulation {

        for(my $x = 0; $x < $popsize; $x++) {
                my $treesize = int rand(5) + 2;
                my $parent = $operators[int rand($#
                    operators)];
                my $tree = Tree::Binary->new($parent);
                $tree = buildTree($tree, $treesize,
                    $parent);
                $population{$tree} = getFitnessOfTree(
                    $tree);
        }

}

sub evolvePopulation {

        tie my %newpopulation, 'Tie::Hash::
            StructKeyed';


        for(my $x=0; $x < ((int(sqrt($popsize))) - 1)
            ; $x++) {
                my $besttree = getBestFromPopulation()
                    ;
                $newpopulation{$besttree} = ();
                delete $population{$besttree}; #remove
                     besttree from population
        }


        my @poparr = (keys %newpopulation);
```

```
        foreach my $doca (@poparr) {
                foreach my $docb (@poparr) {

                                my $temptree =
                                    crossoverTree($doca
                                    , $docb);
                                $newpopulation{
                                    $temptree} = ();
                }
        }

        my @tempary = keys (%newpopulation);
        my $tempsize = $#tempary;
        for(my $y=0; $y < ($popsize - $tempsize); $y
            ++) {
                my $temptree = crossoverTree(
                    getRandomFromPopulation(),
                    getRandomFromPopulation());
                $newpopulation{$temptree} = ();
        }

        %population = ();

        for my $tree (keys %newpopulation) {
                $population{$tree} = getFitnessOfTree(
                    $tree);
        }
}

sub getBestFromPopulation {
        my $bestfitness = 0;
        my $besttree = ();
        for my $tree (keys %population) {
                if($population{$tree} > $bestfitness)
                    {
                        $bestfitness = $population{
                            $tree};
                        $besttree = $tree;
                }
        }

        return $besttree;
}

sub getRandomFromPopulation {
        my @populationlist = (keys %population);
```

```
        my $randtree = $populationlist[int(rand($#
            populationlist))];
        return $randtree;
}

sub printPopulation {

        for my $tree (keys %population) {
                print "------------------\n";
                printTree($tree);
                print "Fitness ".$population{$tree}."\
                    n";
        }
}

sub printTree {
        my ($tree) = @_;
        print Tree::Visualize->new($tree, 'ASCII', '
            Diagonal')->draw() . "\n";
        print "\n" x 2;
        print Tree::Visualize->new($tree, 'ASCII', '
            TopDown')->draw() . "\n";
        print "\n" x 2;
        my $visitor = Tree::Binary::Visitor::
            InOrderTraversalExpressionTree->new();
        $tree->accept($visitor);
        my $equation = (join " ", $visitor->getResults
            ());
        print $equation."\n\n";
}

sub getFitnessOfTree {
        my ($tree) = @_;
        my %docscores = ();
        my $visitor = Tree::Binary::Visitor::
            InOrderTraversalExpressionTree->new();
        $tree->accept($visitor);

        my $statement = (join " ", $visitor->
            getResults());
        eval('sub rankingfunction { my ($tf, $dfmaxcol
            , $Nk, $clhistory, $phistory, $tfavg, $
            $doclength, $tfmax, $collabscore) = @_; my $
            $score = 0; eval { $score=' .$statement.'
            ; }; return $score; } ');

        foreach my $query (keys %queries) {
```

```
my @queryterms = split(" ", $query);


for my $document (keys %invertedindex)
    {

        my $score = 0;
        my $check = 1;
        my $tfmax = tfmax($document);
        my $doclength = doclength(
            $document);
        my $tfavg = tfavg($document);
        my $phistory = personalHistory
            ($document);
        my $clhistory = clusterHistory
            ($clusterinfo{$document});
        my $collabscore =
            collabClusterScore(
            $clusterinfo{$document}, 1)
            ;

        foreach my $term (@queryterms)
            { #Because the query
            contains multiple terms

                my $Nk = Nk($term);
                my $dfmaxcol =
                    dfmaxcol($term);

                if(exists(
                    $invertedindex{
                    $document}{$term}))
                    {
                        my $tf = $invertedindex
                            {$document}{
                            $term};
                        eval {
```

```
                                                $score =
                                                    $score
                                                     + rankingfunction
                                                    ($tf
                                                    , $dfmaxcol
                                                    , $Nk
                                                    , $clhistory
                                                    , $phistory
                                                    , $tfavg
                                                    , $doclength
                                                    , $tfmax
                                                    ,$collabscore
                                                    ) *
                                                    $tf;

                                                };
                                        }
                                        else {
                                                $check = 0;
                                        }
                                }

                                if($check != 0) {
                                        $docscores{$queries{
                                            $query}}{$document
                                            } = $score;
                                }
                        }
                }

my $count = 0;
my $overallpercent = 0;


for my $query (keys %docscores) {
        $count++;
        my @topdocs = ();
        my $percent = 0;
        foreach my $did (sort {$docscores{$query}{$b
            } <=> $docscores{$query}{$a}} (keys %{
            $docscores{$query}})) {


                if(@topdocs <= 9) {
                        push(@topdocs, $did);
                        if(exists($relevancejudgement{
                            $query}{$did}))
```

```
                        {
                                $percent = $percent
                                    + 10;
                        }
                }
        }
        $overallpercent = $overallpercent + $percent;
}

return ($overallpercent/$count);

}

sub buildTree {
        my ($tree, $n, $p) = @_;
    return $tree unless $n > 0;
    $n--;
    if($n > 1&& $p ne "log"){
    $p = $operators[int rand($#operators)];
        $tree->setLeft(buildTree(Tree::Binary->new($p)
            , $n, $p));
        $p = $operators[int rand($#operators)];
        $tree->setRight(buildTree(Tree::Binary->new($p
            ), $n, $p));
        }
        elsif ($n == 1 && $p ne "log"){
                $p = $terminals[int rand($#terminals)
                    ];
                $tree->setLeft(buildTree(Tree::Binary
                    ->new($p), $n, $p));
                $p = $terminals[int rand($#terminals)
                    ];
                $tree->setRight(buildTree(Tree::Binary
                    ->new($p), $n, $p));
        }
        elsif ($n == 1 && $p eq "log"){
                $p = $terminals[int rand($#terminals)
                    ];
                $tree->setRight(buildTree(Tree::Binary
                    ->new($p), $n, $p));
        }
        elsif ($n > 1 && $p eq "log"){
                $p = $operators[int rand($#operators)
                    ];
                $tree->setRight(buildTree(Tree::Binary
                    ->new($p), $n, $p));
        }
```

```
        return $tree;
}

sub crossoverTree {
        my($treea, $treeb) = @_;
        my $parent = ();
        my $parenttest = int(rand(2));
        if($parenttest == 1) {
                $parent = $treea->getNodeValue();
        }
        else {
                $parent = $treeb->getNodeValue();
        }
        my $tree = Tree::Binary->new($parent);
        #Make precaution if root is log
        if($parent ne "log") {

                $tree->setLeft(treeGoDown($treea, int(
                   rand($treea->height()))));
                $tree->setRight(treeGoDown($treeb, int
                   (rand($treeb->height()))));

        }
        else {
                $tree->setRight(treeGoDown($treeb, int
                   (rand($treeb->height()))));
        }

        return $tree;

}

sub treeGoDown {
        my($tree, $steps) = @_;
        for(my $x = 0; $x < $steps; $x++) {
                if($tree->hasLeft() == 1 && $tree->
                   hasRight() == 1) {
                        if(int(rand(2)) == 0) {
                                $tree = $tree->
                                   getRight();
                        }
                        else {
                                $tree = $tree->getLeft
                                   ();
                        }
                }
                elsif($tree->hasRight() == 1) {
```

```
                        $tree = $tree->getRight;
                }
        }

        return $tree;
}

sub loadInvertedIndex {

        my $sthdoc = $dbh->prepare("select * from
            CREOccurance");

        $sthdoc->execute or die "Unable to execute
            query: $dbh->errstr\n";

        %invertedindex = ();

        while(my $row = $sthdoc->fetchrow_hashref)
        {
                $invertedindex{$row->{"DID"}}{$row->{"
                    Term"}} = $row->{"Count"};
        }

        $sthdoc->finish();

}

sub loadQueries {
        my $sthquery = $dbh->prepare("select * from
            Queries");
        $sthquery->execute or die "Unable to execute
            query\n";
        %queries = ();

        while (my $row = $sthquery->fetchrow_hashref)
            {
                $queries{$row->{"Query"}} = $row->{"ID
                    "}
        }

        $sthquery->finish();
}

sub loadRelevanceJudgements
{
        my $sthreljud = $dbh->prepare("select * from
            RelevanceJudgements");
```

```
        $sthreljud->execute or die "Unable to execute
            query: $dbh->errstr\n";

        %relevancejudgement = ();

        while(my $row = $sthreljud->fetchrow_hashref)
        {
                $relevancejudgement{$row->{"Query"}}{
                    $row->{"DID"}} = $row->{"Count"};
        }

        $sthreljud->finish();

}

sub loadUserProfiles
{
        @userprofile = ();

        my $sthuprofile = $dbh->prepare("select DID
            from CREUserProfiles where UserID='1'");
        $sthuprofile->execute or die "Unable to
            execute query: $dbh->errstr\n";

        while (my $ary = $sthuprofile->fetchrow)
        {
        push(@userprofile, $ary);
        }

        $sthuprofile->finish();

}

sub fixQueries {
        my ($query) = @_;
        my @queryterms = ();
        $query =~ tr/[A-Z]/[a-z]/;
        my @tempqueryterms = split(" ", $query);

        foreach my $word (@tempqueryterms) {
                my $holder = 0;
                foreach my $t (@stopwords) {
                        if($word eq $t) {
                                $holder = 1;
                        }
                }
                if ($holder == 0) {
```

```
                        push(@queryterms, porter($word
                            ))
                }
        }

        return @queryterms;
}

sub Nk {
                my $word = $_[0];
                my $nkcount = 0;

                if(!exists($nkhash{$word}))
                {

                for my $document (keys %invertedindex)
                    {
                        if(exists($invertedindex{
                            $document}{$word})) {
                                $nkcount++;
                        }
                }
                        $nkhash{$word} = $nkcount;
                }
                else
                {
                 $nkcount = $nkhash{$word} ;
                }
                return $nkcount;
}

sub tfmax { # The Maximum Term Frequecny in a Document
        my $did = $_[0];
        my $largest = 0;

        if(!exists($tfmaxhash{$did})) {
                for my $word (keys %{$invertedindex{
                    $did}}) {
                        if($largest < $invertedindex{
                            $did}{$word}) {
                                $largest =
                                    $invertedindex{$did
                                    }{$word};
                        }
                }
                $tfmaxhash{$did} = $largest;
        }
```

```
        else
        {
                $largest = $tfmaxhash{$did};
        }
        return $largest;
}

sub doclength {
        my $did = $_[0];
        my $total = 0;
        if(!exists($doclengthhash{$did})) {
                for my $word (keys %{$invertedindex{
                   $did}}) {
                        $total = $total +
                           $invertedindex{$did}{$word
                           };
                }
                $doclengthhash{$did} = $total;
        }
        else {
                $total = $doclengthhash{$did};
        }
        return $total;
}

sub doclengthavg {
        my $total = 0;
        for my $did (keys %invertedindex) {
                for my $word (keys %{$invertedindex{
                   $did}}) {
                        $total = $total +
                           $invertedindex{$did}{$word
                           };
                }
        }
        return $total/$numberodocs;

}

sub doccount {
        my @docids = (keys %invertedindex);
        return $#docids;
}

sub tfavg { #Average Term Frequency in the current
   document
```

```
        my $did = $_[0];
        my $count = 0;
        my $total = 0;
        if(!exists($tfavghash{$did})) {
                for my $word (keys %{$invertedindex{
                   $did}}) {
                        $count++;
                        $total = $total +
                           $invertedindex{$did}{$word
                           };
                }
                $tfavghash{$did} = $total/$count;
        }

        #return $total/$count;
        return $tfavghash{$did};
        #Average Term Frequency in the current
           document
}

sub tfavgcol {
        my $total = 0;
        for my $did (keys %invertedindex) {
                $total = $total + tfavg($did);
        }

        return $total/doccount();
}

sub dfmaxcol {
        my $word = $_[0];
        my $largest = 0;
        if(!exists($dfmaxcolhash{$word})) {
                for my $did (keys %invertedindex){
                        if(exists($invertedindex{$did
                           }{$word})) {
                                if($invertedindex{$did
                                   }{$word}) {
                                        $largest = $invertedindex
                                           {$did}{$word
                                           };
                                }
                        }
                }
        $dfmaxcolhash{$word} = $largest;
        }
        else {
```

```
                $largest = $dfmaxcolhash{$word};
        }
        return $largest;
}

sub df { #Number of Documents in A Collection That
    Contain A Word (SAME AS NK)
        my $word = $_[0];
        my $total = 0;
        for my $did (keys %invertedindex)
        {
                if(exists($invertedindex{$did}{$word})
                    ) {
                        $total++;
                }
        }

        return $total;
}

sub hashValueDescending {
    $docscores{$query}{$b} <=> $docscores{$query}{$a};
}

sub personalHistory {

        my $document = $_[0];
        my $check = 0;
        foreach my $doc (@userprofile) {
                if($document eq $doc) {
                        $check = 1;

                }
        }

        return $check;
}

sub clusterHistory {
        my $cluster = $_[0];

        if(exists($clusterhistory{$cluster})){
                return 1;
        }
        else {
                return 0;
        }
```

```
}

sub collabClusterScore {
        my($clusternum, $userid) = @_;

        if(exists($collabscoretable{$clusternum})){
                return $collabscoretable{$clusternum};
        }
        else{
        my $score = 0;

        foreach my $id (keys %clusterhistory) {
                if(exists $clusterhistory{$id}{
                   $clusternum}) {
                        if($id != $userid){
                                $score = $score + $sim
                                   {$id}{$userid};
                        }
                }
        }

        $collabscoretable{$clusternum} = $score;

        return $score;
        }
}

sub buildSimMatrix {

my $userid = $_[0];

my $sthcluhistory = $dbh->prepare("select
   CREUserProfiles.UserID, CRECluster.Cluster from
   CREUserProfiles, CRECluster where CRECluster.DID =
   CREUserProfiles.DID");
$sthcluhistory->execute or die "Unable to execute
   query\n";
%clusterhistory = ();

while (my $row = $sthcluhistory->fetchrow_hashref) {
        $clusterhistory{$row->{"UserID"}}{$row->{"
           Cluster"}} = 1;
}

$sthcluhistory->finish();
```

```
foreach my $user (keys %clusterhistory) {
        if($userid != $user) {
        $sim{$user}{$userid} = 0;
                foreach my $cluster (keys %{
                   $clusterhistory{$userid}}) {
                        if(exists $clusterhistory{
                           $user}{$cluster}) {
                                $sim{$user}{$userid
                                   } = $sim{$user}{
                                   $userid} + 1;
                        }
                }
        }
}

}

sub avgFitnessPopulation {
        my $count = 0;
        my $total = 0;

        for my $tree (keys %population) {
                $count++;
                $total = $total + $population{$tree};
        }

        return $total/$count;
}

sub loadClusterInfo {
        %clusterinfo = ();

        my $sthcluinfo = $dbh->prepare("select * from
           CRECluster");

        $sthcluinfo->execute or die "Unable to execute
           query: $dbh->errstr\n";

        while(my $row = $sthcluinfo->fetchrow_hashref)
        {
                $clusterinfo{$row->{"DID"}} = $row->{"
                   Cluster"};
        }

        $sthcluinfo->finish();

}
```

```
sub loadClusterHistory {
        %clusterhistory = ();

        my $sthcluhistory = $dbh->prepare("select
            CRECluster.Cluster from CREUserProfiles,
            CRECluster where CRECluster.DID =
            CREUserProfiles.DID and CREUserProfiles.
            UserID = 1");

        $sthcluhistory->execute or die "Unable to
            execute query: $dbh->errstr\n";


        while(my $row = $sthcluhistory->
            fetchrow_hashref)
        {
                $clusterhistory{$row->{"Cluster"
                    }} = 1;
        }

        $sthcluhistory->finish();


}
```

APPENDIX C

CLUSTERING SOURCE CODE

```
#! /opt/local/bin/perl -w

# K-Means Document Clustering Program
#Size of the Cluster Taken From the Command Line

use DBI;

($sec,$min,$hour,$mday,$mon,$year,$wday,
$yday,$isdst)=localtime(time);
printf "%4d-%02d-%02d %02d:%02d:%02d\n",
$year+1900,$mon+1,$mday,$hour,$min,$sec;

#Hash of {documentID}{term} = Count
%occurance = ();
#Hash of {documentID}{documentID} = Distance
%distances = ();
#Hash of {documentID} = Cluster
%cluster = ();
$clustersize = $ARGV[0];

print "Final Cluster Size: ".$clustersize."\n";

my $dsn = 'DBI:mysql:thesis:localhost';
my $db_user_name = 'root';
my $db_password = '';
my ($id, $password);
my $dbh = DBI->connect($dsn, $db_user_name,
    $db_password);

@frdocs = ();
@biggestcluster = ();

$sth = $dbh->prepare("select * from CRSOccurance");
$sth->execute or die "Unable to execute query: $dbh->
    errstr\n";

while($row = $sth->fetchrow_hashref)
{
        $occurance{$row->{"DID"}}{$row->{"Term"}} =
            $row->{"Count"};
}
```

```
$sth->finish();

for $document (keys %occurance) {
        push(@frdocs, $document);
}

%documentwordcount = ();

for $document (keys %occurance) {
        for $term (keys %{$occurance{$document}}) {
                if(exists($documentwordcount{$document
                    })) {
                        $documentwordcount{$document
                            } = $documentwordcount{
                            $document} + $occurance{
                            $document}{$term};
                }
                else {
                        $documentwordcount{$document
                            } = $occurance{$document}{
                            $term};
                }
        }
}

print "StepOne Finished \n";

#This Hash Keeps Track of How Large Each Cluster Is
%clustersizes = ();

foreach $document (@frdocs) {
        $cluster{$document} = 1;

        if(exists($clustersizes{1})) {
                $clustersizes{1} = $clustersizes
                    {1} + 1;
        }
        else {
                $clustersizes{1} = 1;
        }
}

$numberofsplits = 1;

while($numberofsplits < $clustersize) {
```

```
$largest = 0;
$numberofsplits++;

foreach $key (keys %clustersizes) {
        if($clustersizes{$key} > $largest) {
                $largest = $clustersizes{$key
                    };
                $largestCluster = $key;
        }
}

print "Splitting Largest Cluster ".
    $largestCluster."\n";

print "Cluster Size ".$clustersizes{
    $largestCluster}."\n";


@biggestcluster = ();
foreach $id (keys %cluster) {
        if($cluster{$id} == $largestCluster) {
                push(@biggestcluster, $id)
        }
}


@twodocs = greatestdistance(@biggestcluster);
$pointA = $twodocs[0];
$pointB = $twodocs[1];
#

foreach $id (keys %cluster){
                if($cluster{$id} ==
                    $largestCluster) {
                                $dista =
                                    distance(
                                    $id,$pointA
                                    );
                                $distb =
                                    distance(
                                    $id,$pointB
                                    );
                                if( $dista <
                                    $distb) {
```

```
                                                $cluster
                                                    {$id
                                                    } =
                                                    $largestCluster
                                                    ;
                                            }
                                            else {
                                                $cluster
                                                    {$id
                                                    } =
                                                    $numberofsplits
                                                    ;
                                            }
                                    }
                            }
                            print $cluster{$pointA}."\t".$cluster{$pointB
                                }."\n";

                            %clustersizes = ();
                            foreach $id (keys %cluster) {
                                    $clLoc = $cluster{$id};
                                    if (exists($clustersizes{$clLoc})) {
                                            $clustersizes{$clLoc} =
                                                $clustersizes{$clLoc} + 1;
                                    }
                                    else {
                                                    $clustersizes{$clLoc
                                                        } = 1;
                                    }
                            }
}

$sthb = $dbh->prepare("CREATE TABLE CRSCluster (DID
    VARCHAR(30), Cluster INT, PRIMARY KEY('DID', '
    Cluster'))");
$sthb->execute;
$sthb->finish();

foreach $id (keys %cluster) {
        $sthc = $dbh->prepare("insert into CRSCluster
            values('".$id."','".$cluster{$id}."')");
        $sthc->execute;
        $sthc->finish();

}

                foreach $thing(keys %clustersizes){
```

```
                        print $thing."\t".
                            $clustersizes{$thing}."\n";
                }


($sec,$min,$hour,$mday,$mon,$year,$wday,
$yday,$isdst)=localtime(time);
printf "%4d-%02d-%02d %02d:%02d:%02d\n",
$year+1900,$mon+1,$mday,$hour,$min,$sec;
exit;


sub distance {
        $distance = 0;
        $difference = 0;
        $didx = $_[0];
        $didy = $_[1];

        if(!exists($distances{$didx}{$didy})) {

        if($didx ne $didy) {

                for $term (keys %{$occurance{$didx}})
                    {

                                if(exists($occurance{
                                    $didy}{$term})) {
                                        $difference
                                            = (
                                            $occurance{
                                            $didx}{
                                            $term} -
                                            $occurance{
                                            $didy}{
                                            $term})**2;
                                }
                                elsif (!exists(
                                    $occurance{$didy}{
                                    $term})) {
                                        $difference
                                            = (
                                            $occurance{
                                            $didx}{
                                            $term} - 0)
                                            **2;
                                }
```

```
                        if(exists($distance{
                            $didx}{$didy})) {
                                $distance =
                                    $distance
                                     +
                                    $difference
                                    ;
                        }
                        else {
                                $distance =
                                    $difference
                                    ;
                                }
                    }

                    for $term (keys %{$occurance{
                        $didy}}) {
                                if(!exists($occurance{
                                    $didx}{$term}))
                                {
                                        $difference
                                             = (
                                            $occurance{
                                            $didy}{
                                            $term} - 0)
                                            **2;
                                        $distance =
                                            $distance
                                             +
                                            $difference
                                            ;
                                }


                    }
            }
    else {
                    $distance = 0;
    }

            $distances{$didx}{$didy} = $distance;
            $distances{$didy}{$didx} = $distance;
    }
    else {
            $distance = $distances{$didx}{$didy};
    }
```

```
        return $distance;
}

sub greatestdistance {
        @holderlist = @_;
        if($#holderlist > 200) {
                $startpoint = int (($#holderlist)/100)
                    ;
                @documentlist = @holderlist[0..
                    $startpoint];
        }
        else {
                @documentlist = @holderlist;
        }

        $greatestdistance = 0;
        @furthesttwo = ();

        foreach $documentone (@documentlist) {
                foreach $documenttwo (@documentlist) {
                        $distance = distance(
                            $documentone, $documenttwo)
                            ;
                        if($distance >
                            $greatestdistance        )
                             {
                                $greatestdistance =
                                    $distance;
                                $pointa = $documentone
                                    ;
                                $pointb = $documenttwo
                                    ;
                        }
                }
        }
        print $pointa."\t".$pointb."\n";

        push(@furthesttwo, $pointa);
        push(@furthesttwo, $pointb);
        return @furthesttwo;
}
```