Learning Query Behavior in the Haystack System

by

Wendy S. Chien

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

Author ....
Department of Electrical Engineering and Computer Science
May 22, 2000

Certified by ....
David Karger
Associate Professor
Thesis Supervisor

Certified by ....
Lynn Andrea Stein
Associate Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Learning Query Behavior in the Haystack System

by

Wendy S. Chien

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2000, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science

Abstract

Haystack is a personalized information retrieval system that allows users to store, maintain, and query for information. This thesis describes how learning is added to the system so that when a user makes a query on a topic similar to a previous query, the system can use the relevance feedback information from before to provide an improved result set for the current query. The learning module was designed to be modular and extensible so more sophisticated learning algorithms and techniques can be easily implemented in the future. Testing of our system showed that learning based on relevance feedback somewhat improved the results of the queries.

Thesis Supervisor: David Karger
Title: Associate Professor

Thesis Supervisor: Lynn Andrea Stein
Title: Associate Professor

Acknowledgments

I would like to thank Professors David Karger and Lynn Andrea Stein for their support and guidance on this project. Their advice and suggestions were invaluable.

I would especially like to thank Svetlana Shnitser for her help in various parts of the project from the work on the kernel during the summer to the work on improving the query process. I learned a lot from our brainstorming sessions and working together. I would also like to thank Ziv Bar-Joseph for adding the relevance buttons to the WebGUI and to Jamie Teevan for her contribution to the feature vector discussions.

I would also like to acknowledge the other students who worked on Haystack, including Adam Glassman, Adam Holt, Will Koffel, and Ian Lai.

I am also grateful to Emily Chang for answering all my questions about using Xfig, and Jonathan Lie for answering my questions about LaTeX.

Finally, I would like to thank my family and friends for all their support during this past year.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As we advance further in the information age, the amount of data we keep on our personal computers or accounts is increasing at a rapid rate. This phenomenon is the result of several factors including the declining price of storage, the increase in popularity of the Internet, and new forms of digital media. Because storage is becoming increasingly inexpensive, we are able to store massive amounts of data. The vastness of the Internet also gives us access to a large amount of information that only increases as more people go online. The emergence of digital forms of photographs, audio clips, and video clips only adds to the mass of data available and collected. The combined result of a large storage space and access to a great number of files on a wide range of topics and formats is a body of information so large that we will no longer be able to easily keep track of all of our own data. Not only will we need tools to search through the Internet as we do now, but soon we will also need tools to search through our own data repositories. Unfortunately, most current search engines are geared towards a generic user and make no effort to utilize the characteristics of the user to increase query performance.

Like these traditional search engines, the main goal of the Haystack System is to solve all the above mentioned problems, but unlike the other systems, Haystack seeks to improve the search experience by personalizing the process. Personalization occurs by maintaining a data repository, or what is called a haystack in the Haystack system, for each user that contains all the documents archived, all the queries made and any other manipulation of data made by the user.

1.1 Personalization in the Haystack System

By keeping a personal haystack for each user, we can improve the search process in two main ways. The first method is by running the query first locally on the user's own haystack and then gradually increasing the scope of the files being queried. For example, the query is initially run on the user's own haystack, then on the user's associates' haystacks, and finally it is run on a universal set (the web). This sequence is based on the assumption that the user would like and trust, and therefore find more useful, documents collected by themselves or people they know more than a random document found elsewhere.

A second way in which personalization helps improve searching, and the path that we will explore in this thesis, is by learning about the characteristics of each user through which documents the user says he finds relevant and useful. Because the Haystack system is designed for individual users, we are able to observe the query behavior of each user over the entire time the user uses the system. This means that we can collect a large amount of data unique to each user regarding how he evaluates the results from each query he makes. Using this user-specific data we can learn about the user's interests and improve the result sets of all the future queries this user makes. Most other systems which attempt learning either learn over only a single query session for a user, which means they have a limited amount of data, or generalize and apply what they have learned for all users to each individual user, meaning their data is not necessarily accurate. Personalization in Haystack allows us to provide more accurate query results because we are able to amass a large amount of data specific to each user to help in improving the results of their own queries.

1.2 Learning

With all this data about the user, the question is, how can we use it to improve the query results we return to the user when the user makes a similar query in the future? (Improving the query results means returning more relevant documents than before and returning relevant documents with higher rankings than before.) To answer this question, let us step back and examine the idea of information retrieval. In the problem of information retrieval, we are given a query that is a description of a set of documents the user wishes us to return. Unfortunately, usually the description is not specific enough or clear enough to uniquely identify a set of documents. Thus, we are left trying to compare documents to a query, even though the query string is too short to be a good example of a relevant document.

Here is where relevance feedback, the relevance ratings the user gave the documents, can help us. By marking a document as relevant, the user is actually describing the query with the document because he is saying that that document matches the query. This new description can help us in multiple ways. First we can expand the query description to include relevant attributes so when issuing the query again, documents that have the new attributes will be returned. Secondly, now that our query also has a document-formulated description, we can produce a ranked list of relevant documents by comparing the documents in the corpus to our query to see if they match and therefore are relevant to the query.

The question that follows is then, how do we know how to match documents to a query even if it is described by documents? Again, if we step back and think about exactly what we are trying to accomplish, we notice that we would create the ideal result set by dividing the documents into relevant and non-relevant sets and only returning the relevant set. This problem then is a perfect example of a standard machine learning classification problem. A classification problem is one where we have a set of data made up of several classes or types and after being given "training" examples of each type, we need to be able to classify any new data point we are given. In our case, the data set is the set of all the documents in the user's corpus, the two classes are relevant and non-relevant documents, and the training examples are the relevance data that we are given by the user. Therefore, in order to solve our problem, we need only to apply classification algorithms.

1.3 Learning in Haystack

As we have described above, we can apply the information we gather about users from the the previous queries they made to improving the results of all their future queries in a variety of ways. In this thesis, we will focus on learning from previous queries on a topic how to improve the rankings of documents when given a query on the same or a similar topic. In this case we can use the information about which subset of documents the user has found relevant in the previous queries that were similar to the current one to improve the results returned for the current query. For example, if the user has queried on the topic of "cars" in the past and said they liked documents that were mostly on German cars then when they query for cars again, we can use the information that they were interested in German cars to give documents about German cars higher rankings.

Another idea, which we will not pursue in this thesis, but would be interesting to study later, is learning from all previous queries how to improve the rankings of files for a query on a brand new topic. We would still be able to improve the search results in this case by finding common traits between this query and previous ones. An example of such a case is if the user makes a query on airplanes. Suppose in the past, the user has searched for cars and motorcycles, and for each he was only interested in articles which discussed the engines of these vehicles. From this we can guess that the user is interested in engines in general, so when he looks for articles on airplanes, he is most interested in articles about airplane engines.

1.4 Learning Module

To implement the learning tasks described above, we created a new module in the system because Haystack currently does not support learning. The module works closely with the module that handles the query processing. The responsibilities of the learning module include gathering the relevance information from the user, storing it, and using several basic relevance feedback techniques and classification algorithms to produce a better result set. Because this is the initial attempt at learning in the system, much of the work of this thesis is not focused on the actual algorithms we use, but rather on the design of infrastructure in Haystack that supports learning and future more sophisticated algorithms.

1.5 Problem Definition and Thesis Overview

The problem we are trying to solve is given a query from the user, how should we modify it based on past information, and how should we provide a better set of rankings from that information. In this thesis, we will only concentrate on using past data on queries that are on topics the system has seen before. Because none of the infrastructure needed for learning is in place, much of the work in this thesis is in designing the learning module. Testing is done to show that the implementation is correct and to show the usefulness of having machine learning in the system.

In the next chapter, we cover in more detail the relevant work done in machine learning research. In chapter 3 we discuss the Haystack System. Chapter 4 describes the design of the learning module, while chapter 5 goes over the actual implementation of the learning module. In chapter 6 we review the correctness testing done. Chapter 7 suggests future work and draws conclusion from this research.

Chapter 2

Machine Learning Background and Related Work

Many approaches have been tried in applying machine learning to information retrieval. In this section we review some of the standard ideas in using the vector space model, relevance feedback, and query expansion.

2.1 Vector Space Model

Many learning algorithms for information retrieval represent documents using the vector space model. In this model, each document is represented by a vector where each element of the vector maps to an attribute or feature of the document. An example of an attribute is the number of times a particular word appears in the document, though we should be careful to note that features need not be only words. Choosing which features to put in the vector is an important decision because it determines which attributes of a document the learning algorithm will consider. Also, since many learning algorithms assume independence between features, not including too many similar dependent attributes is also important.

Two techniques are commonly used in selecting the terms used in the vector when dealing with text queries. The first method is the elimination of stop words. Stop words are words like “the”, “and”, “of”, etc. that appear in many documents and do not provide any insight about the content of a document. Removing these words removes any distortion in the results that might occur because of these words. The second technique is called stemming. In stemming, only one term is used to represent words that have the same stem, but are in different forms (e.g., eat vs. eating). This method therefore maintains a more accurate count of the number of times a word appears because it counts all the different forms of a word as the same word. Stemming is especially important because of the term independence assumption many learning algorithms use. This assumption is clearly incorrect, but is applied because it greatly simplifies the problem and reduces the computation needed. So if stemming were not used, and a query term was “eat”, then all instances of “eating” in a document would have no bearing on the query score of the document because the algorithm would not recognize that “eat” and “eating” are the same word. Since we know we are working with this independence assumption, we ought to remove cases of dependence which are easy to detect and rectify. This includes the counting of the same word in two different forms as two separate unrelated words. The final words found in the vector are all of those that appear in the documents and remain after applying these two techniques.

2.2 Relevance Feedback

Relevance feedback refers to the situation where the system receives feedback from the user about which documents the user found to fit the query. Previous studies [H92, SB90] have shown that users tend to input queries which often do not provide enough information to uniquely and accurately identify the topic for which the user is searching. User queries can be inadequate for a variety of reasons, including being too short so not all the features pertinent to the query are present, the user and the system interpret the features in two different ways, or simply query formulations and documents are different concepts and cannot be easily compared. For all these reasons, having examples of documents the user finds relevant to the query is useful to further describe the topic the user means to define with the query.

Traditionally, relevance feedback has been used in two ways to help improve the result set for future queries, query expansion and reranking the documents returned. Query expansion is the process of adding terms to the query to refine or redefine the scope of the results. Reranking is generally done by using the relevant documents as examples of what the user is looking for and then ranking the rest of the documents based on how similar they are to the relevant ones.

2.2.1 Query Expansion

Query expansion is the process where important features found in the relevant documents are added to the original query so when the query is issued again, the topic described by the new query more accurately reflects the user's information need. It is helpful because often the original query does not accurately describe the topic the user has in mind.

There are two main types of query expansion, automatic query expansion (AQE) and interactive query expansion (IQE). As the names suggest, in AQE, the machine tries to expand on the query without the help of the user, while in IQE, the user is asked for help in deciding which terms are added to the query. A simple version of AQE working with relevance feedback is where the algorithm finds the N most frequently occurring terms in the relevant documents and adds them to the query, where N is some preset parameter. A basic version of IQE behaves similarly in selecting the N frequently appearing terms, but instead of automatically adding them to the query, it presents these N terms to the user and allows the user to select which are relevant and should be added.

Past research has shown that if the user is knowledgeable and active in his participation, the results of using IQE are superior to AQE. The main problem with AQE is accidentally choosing to add a non-relevant term to the query. Because documents are not all equally relevant and even a relevant document can contain non-relevant terms, it is possible for a non-relevant term to be added. If this happens, then "query drift," where the results returned by the query drifts apart from the topic the user is actually interested in, can occur. This problem can be mitigated by methods which aim to determine which documents and terms are more relevant. There is ongoing research to find such methods. The main drawback of IQE is that we do not want to burden the user with the responsibility of choosing the relevant terms. Also, if the user is not active in providing feedback then the system will see no improvement.

2.2.2 Reranking

Relevance feedback from the user also allows us to treat the information retrieval problem as more of a traditional classifier problem. Information retrieval is different from classification in the sense that we are mapping query words which we use to define a topic to documents which might fit that topic, instead of mapping documents to topics. Using feedback, we obtain training data because we now have examples of both positive and negative documents. With this new information, we turn the retrieval problem into a classification problem and can use standard machine learning algorithms for classification. Below, we discuss two algorithms which can be used for relevance feedback ranking.

Rocchio's Algorithm

A basic relevance feedback technique in information retrieval is Rocchio's algorithm, which was developed in 1961, and has been shown to perform relatively well despite its simplicity [SMB97]. The idea behind Rocchio's algorithm is to find what is called the *optimal query*, which in theory ideally describes the representation of a query that would return the relevant documents. Since the optimal query represents what the user is looking for in a document, we can use it to calculate, by some similarity measure such as the dot product, how closely documents match it, and therefore how well they fit what the user is looking for.

In order to determine the optimal query, we use the following equation.

$$Q_{opt} = \frac{1}{n} \sum_{D_i \in R} \frac{D_i}{|D_i|} - \frac{1}{N-n} \sum_{D_i \in NR} \frac{D_i}{|D_i|} \tag{2.1}$$

where Q_{opt} is the optimal query and represented by a word vector, D_i is a document, also represented by a vector, $|D_i|$ is the Euclidean length of the vector D_i, R is the set of relevant documents, NR is the set of non-relevant documents, n is the number of relevant documents, and N is the total number of documents. Because the Q_{opt} cannot be calculated exactly (we do not have the sets of relevant and non-relevant documents), we will use the following equation to approximate it.

$$Q_i = \alpha Q_{i-1} + \beta \sum_{D_i \in R-seen} \frac{D_i}{|D_i|} - \gamma \sum_{D_i \in NR-seen} \frac{D_i}{|D_i|} \tag{2.2}$$

where $R - seen$ and $NR - seen$ are the sets of relevant and non-relevant documents that the user has specified, and α, β, and γ are parameters chosen, through calibration, to optimize the approximation. Now we can compare document vectors to Q_i, and update Q_i every time this query is run. Each document vector is normalized by its Euclidean vector length to keep from overweighting longer documents. Long documents tend to have higher term frequencies (because the same term can be repeated more in a longer document) and more distinct terms (because long documents contain more terms overall). Both of these characteristics can increase a document's score in a similarity function and give long documents an advantage over shorter ones. Normalization by vector length reduces the effect of both characteristics because an increase in either the term frequencies or number of distinct words will increase the length of the vector. The the relevant and non-relevant sums are normalized by the number of documents in each category so a difference in the number of documents marked relevant versus non-relevant will also not affect the optimal query.

Thus, we can see that the optimal query is essentially the average characteristics of the relevant documents minus the average characteristics of the non-relevant documents. In the equation, each relevant document is added to the optimal query while each non-relevant document is subtracted. Looking at it from the vector space view, we are moving the optimal query vector closer to the relevant document vectors and away from the non-relevant document vectors. The optimal query tries to capture all the important characteristics of the relevant documents by adding all the relevant document vectors to the optimal query and assuming the important characteristics are the ones which appear in many of the relevant documents. By the same argument, the optimal query also captures, but negatively, the characteristics that appear often in non-relevant documents. Intuitively this makes sense because if we use similarity to the optimal query as a measure of how relevant a new document is to the query, then if it shares many of the characteristics of the relevant documents but not many of the non-relevant documents, then it is most likely a relevant document.

Improvements on Rocchio: Since the invention of the algorithm, several improvements have been made on Rocchio. One type of improvement involves using a more advanced document length normalization on the vector while another changes the domain of the non-relevant documents used.

A possible normalization technique is pivoted document length normalization [SBM96]. This approach tries to reweight the terms in the vector so that for every given document length, the probability of retrieving a document of that length is equal to the probability of finding a relevant document of that length. It first determines the relevance and retrieval probabilities of documents as functions of document length using a another normalization technique (like the Euclidean vector length). It then finds the pivot point, the point at which the two functions intersect. On one side of the pivot point, the retrieval probability is higher than the relevance probability, while on the other side of the point, the opposite is true. By increasing the normalization factor (the amount the raw value is divided by) on the side with a higher probability of retrieval, we lower the probability of retrieval for those lengths. By the same reasoning, if we lower the normalization factor on the opposite side of the pivot point, we increase the probability of retrieval. So we see that in order to equalize the probabilities of retrieval and relevance, we want to tilt the normalization at the pivot point. The equation for finding the tilted or pivoted normalization factor is below.

$$\textit{pivoted normalization} = (1.0 \text{ - } \textit{slope}) \times \textit{old normalization}$$

where the slope is the amount to tilt.

In addition to normalizing the document vectors, training Rocchio on only the query domain (the set of documents in the topic of the query) instead of the entire corpus will also yield better results [SMB97]. In the original algorithm, non-relevant documents outside the query topic affected the formulation of the optimal query, but this could lead to problems. Consider the case where the user asks "Which Volkswagon models have performed well in crash safety tests?" The word "car" points us to the right domain, but will not help in determining which articles are good for learning about the safety of Volkswagons. If we run Rocchio's original algorithm the number of occurrences of "car" in the non-relevant documents will be low because of all the non-relevant documents outside the query domain, but it will be high among the relevant documents because it is likely to appear in any document about Volkswagon safety. Therefore, the word "car" will appear to be a good word for determining relevance when it is not. If, however, we only use non-relevant documents within the query domain, then we will not encounter this problem. Any word that describes the domain, but is not relevant to the query itself, will appear in both the relevant and non-relevant examples used in formulating the optimal query. Its high occurrence in both sets will cancel out so the word will hopefully be used as neither an indicator of relevance nor non-relevance. Unfortunately, this algorithm can also accidentally overshoot and mark the terms in the query string as being indicators of non-relevance, because non-relevant documents in the query domain will likely have a occurrence of the original query terms.

Boosting

While Rocchio is typically used only for relevance feedback in information retrieval, boosting is a generic machine learning technique that can be applied to a wide variety of problems. The idea behind boosting is to create a highly accurate classifier (or hypothesis) by combining the opinions of several different "weak" classifiers, thereby "boosting" the overall accuracy. These classifiers are made by training a weak learning algorithm on different data sets. A weak learning algorithm is defined to be one which is only required to be correct slightly more than half of the time, meaning it only needs to perform better than random guessing does. To classify a future document each of the classifiers classifies the new document as a positive or negative example of the query and their combined votes determine the overall classification.

Boosting originated from work on the "PAC" (Probably Approximately Correct) learning model. In the early 1980s, the idea of "boosting" a weak learning algorithm to be an arbitrarily accurate "strong" learning algorithm was invented. By the end of that decade, several more boosting algorithms emerged but each with some practical drawback. In 1995, AdaBoost, which solved many of the problems that plagued its predecessors, was introduced by Freund and Schapire [S99]. Currently is it one of the most effective boosting algorithms used.

AdaBoost is given a training set $(x_1, y_1), ..., (x_m, y_m)$ where $x_i \in X, y_i \in Y$ and X is some domain space and $Y = \{-1, 1\}$. The number of weak classifiers, h_t, used is dependent on the number of training rounds, T, used. In each round, the weak learning algorithm is called on some distribution $D_t, t = 1 \cdots T$ determined in the previous round based on the previous error of the learner. This distribution is really a length m array of weights (whose sum is 1). When the weak learner is called on the distribution D_t, the pair (x_i, y_i) is scaled by the weight held at $D_t(i)$. The resulting trained weak learner is one of the T classifiers used in the end. Below is the pseudocode for AdaBoost.

```
  Given: $(x_1, y_1), ..., (x_m, y_m)$ where $x_i \in X, y_i \in Y = \{-1, 1\}$
Initialize $D_1(i) = 1/m$.
```

For $t = 1, ..., T$:

- Train weak learner using distribution D_t.
- Get weak hypothesis $h_t : X \to \{-1, 1\}$ with error

$$\epsilon_t = \sum_{i:h_t(x_i) \neq y_i} D_t(i). \tag{2.3}$$

- Choose $\alpha_t = \frac{1}{2} \ln(\frac{1-\epsilon_t}{\epsilon_t})$.
- Update:

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} e^{-\alpha_t} & \text{if } h_t(x_i) = y_i \\ e^{\alpha_t} & \text{if } h_t(x_i) \neq y_i \end{cases} \tag{2.4}$$

  where Z_t is a normalization factor (chosen so that D_{t+1} will be a distribution.

Ouput the final hypothesis:

$$H(x) = sign(\sum_{t=1}^{T} \alpha_t h_t(x)). \tag{2.5}$$

When using boosting in relevance feedback, the training set $(x_1, y_1), ..., (x_m, y_m)$ is a list of document vector-relevance rating pairs.

2.3 Related Work

Many studies have been done in applying machine learning to information retrieval. In addition to the algorithms mentioned above, some of the techniques that have been used include neural nets, naive Bayes, decision trees, k-nearest neighbor, genetic algorithms. Because there are too many to discuss here, we will only talk about studies which are closely related to what we are trying in Haystack.

2.3.1 Boosting and Rocchio in Text Filtering

In their paper *Boosting and Rocchio Applied to Text Filtering*, Schapire, Singer, and Singhal, compare the performance of a modified Rocchio algorithm to their own algorithm AdaBoost on the problem of text filtering. Text filtering is only a slightly different problem from ranked retrieval, which is the problem we are trying to solve.

In text filtering the system is trying to decide whether or not to send the user a document, meaning it has to make a binary decision. In ranked retrieval we do not need to make a strict decision on the relevance of a document, but instead can give it a value for how relevant we think the document is.

The boosting algorithm they use is AdaBoost, the same as the one described here earlier, and the version of Rocchio they use is one enhanced with query zones and term reweighting. Their experiments with TREC data show that unless there is a large number (hundreds to thousands) of relevant documents, there is no noticeable difference in the performance of AdaBoost and Rocchio.

2.3.2 Okapi

Okapi is an information retrieval system developed at City University in London. It is accessible by academic users at the university and uses the library catalog and a scientific abstracts journal as databases. Okapi is a family of bibliographic retrieval systems and suitable for searching files of records whose fields contain textual data. In currently runs many different retrieval techniques, including relevance feedback applied to query expansion. Results from their studies showed that query expansion does improve performance, with very few queries returning no relevant documents. A note to make here is that the system does not store user relevance opinions beyond the current query session.

2.3.3 Direct Hit and Ant World

Direct Hit [DH00] and Ant World [AW00] are two systems for web searches which use relevance feedback from all users to improve query results to each user. For example, if a user makes a queries on "German cars," then the system will see what documents other users liked in the past and will use those to recommend documents to this user. So if other users thought the official site for "Volkswagon" was an excellent page, then it would appear with a high ranking for this user even though this user has made no relevance judgments on this query.

Chapter 3

Haystack Background

In this chapter we will describe the Haystack System in more detail by examining the information that is stored in Haystack, the types of queries accepted by Haystack, and finally the architecture of the system.

3.1 Information Stored in Haystack

The Haystack system provides its users a personal environment to store and manage their data, but the main focus in the system is in providing powerful searching services for the user. Nevertheless, the searching ability lies largely in what data the system chooses to store.

When a file is archived, not only is the content of the file stored, but meta-data about the file is also stored. For example, if the user archives an email from Fred dated yesterday, the content of Fred's email will be kept, but so will the related pieces of information that the type of the file is an email, the author is Fred, and the date it was received was yesterday. By also storing the meta-data associated with the file, we enable ourselves to expand the scope of the queries to include searches over the attributes of the file, not just the content. This also means any of meta-data can be returned as a result of a query. A user might find this feature useful if they are looking for an author who has written a lot about a topic the user is interested in. For simplicity, we will define a document to be an information item that can be described and returned by a query.

In addition to storing files that the user archives, the system also stores information about all the queries the user has made. Kept with the query is the query description (or string), and the entire result set. Like the meta-data, this means queries can also be returned as results of other queries. A previous query is probably only returned if it is somewhat similar to the current one, so the results of the previous query may be of interest to the user for this query.

3.2 Types of Queries in Haystack

As mentioned in the previous section, the real power of Haystack comes from the searches it is able to run. Currently Haystack accepts two types of queries, textual and structured queries. Textual queries have been available since the beginning of the project [A98], Structured queries have only recently been added [S00].

3.2.1 Text Queries

In a text query, the user gives the system a string of words that describes the topic on which they want information. The system then tries to match the query to the documents in the user's haystack and returns a list of documents relevant to the topic. For instance, the user might input the string "German cars" and the system would return documents on Mercedes, BMW, Volkswagon, or any other articles that talk about German cars.

In Haystack, we use an off the shelf information retrieval (IR) system to do the basic text search. We are using a system called ISearch, but we treat it as a black box. The query module passes the query string to the IR system and receives in return a ranking of documents. Haystack knows nothing about how the IR system decided that these documents were the most relevant. The decision to treat the IR system we use as a black box was made to allow the IR system to be easily replaced so users could use whichever search engine they like best. The black box model keeps our design modular, but because we have no access to the internal data structures, we have had to replicate some of the functionality in our own system [LI99]. In the future we will use a different IR system called Lucene and fully integrate it with Haystack, allowing us access to the data structures and algorithms used, thereby improving our ability to learn and rank documents.

3.2.2 Structured Queries

A structured query is one where the user is able to specify attributes they are specifically looking for in a document. Often these attributes will be meta-data that the services were able to glean from the document. For example, suppose the user is looking for a piece of email that was sent to them last month. While the text query might be able to return the right piece of email if the user was able to describe the content of the email correctly, it would be much easier for both the system and the user if the user were able to say that he wants all documents where the date is April 2000 and the type of the document is email. Using a structured search, the user gets exactly the type of document he is searching for. In addition, the user is allowed to combine a textual and a structured query so the user can specify both the content and the type of document.

Structured queries are handled by a completely different module than the textual queries. The query module separates the types of queries and issues them to the different modules assigned to handling them. The structured query string is sent to a query parser which translates the string to SQL. The SQL description is sent to a relational database which analyzes and returns the result. The score given to a document is between 0 and 1 and is equal to the fraction of the number of parts of the query it satisfies. For example, for the April 2000 email query described above, if a document is of type email but was received in March 2000, then the score of this document for this query would be $\frac{1}{2}$. A listing of the documents with their scores is then returned to the query module.

3.3 Architecture of Haystack

The main tasks of the Haystack System are archiving data and allowing the user to search through the data stored. These abilities are achieved mostly through the use of the data objects (which we call straws) and the services which manipulate the straws.

3.3.1 User Interfaces

Haystack has three user interfaces: the graphical user interface (GUI), the web interface (WebGUI), and the command line interface. Currently, each is implemented separately and there is no shared code between the interfaces.

3.3.2 Data Objects (Straws)

The data in the system are kept in objects called straws (`class Straw`), that are arranged in a directed graph used to represent the information and relationships between the data. The nodes in the graph are either pieces of information or hubs for information and the edges represent the relationship between the nodes.

The class Straw has three main subclasses, `Needle`, `Tie`, and `Bale`, each of which has a different role in the data model. A `Needle` object is responsible for storing the actual piece of information. For example, the title or body of a document is wrapped in a Needle object. A `Bale` object acts as a hub, and represents a collection of information, such as a query (a collection of query results) or a document (a collection of the body and the meta-data of the document). A `Tie` object describes the relationship between two other `Straws`. Therefore, a `Tie` would connect a `Needle` containing an author name to the bale representing a document written by that author. Figure 3-1 shows an example of how a document and a query are represented. The document is Shakespeare's tragedy *Romeo and Juliet* as we can see from the title, author, and text needles. The type of the document is Postscript. It matched a query on "Romeo" with a score of .75. We can see how powerful the data graph is because of its ability not only to hold information but also to express the relationships between the data.

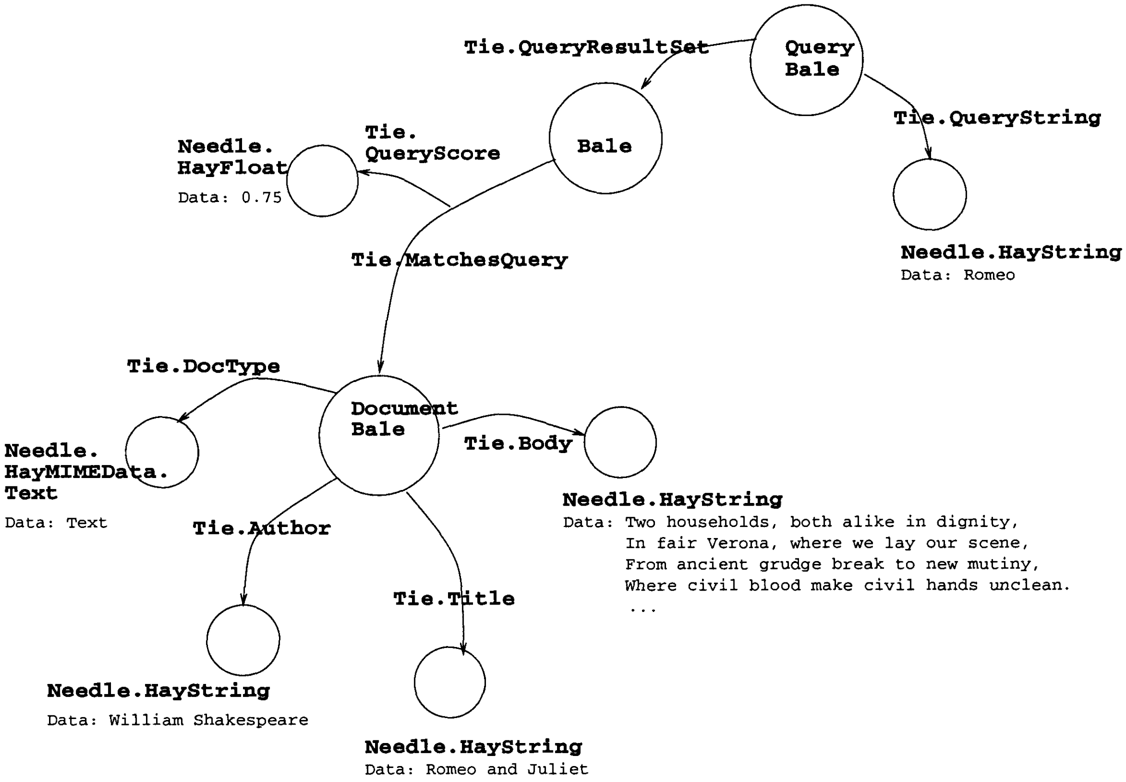


Figure 3-1: Straw Representation of a Query and a Document.

3.3.3 Services

The services in Haystack that are responsible for all the actions done or based on the data model consist of two types. The first type of service is one which is triggered off of an event in the data model, such as the creation of a tie between two straws. The second type is one that is called from a user interface and needs to interact with the data.

Services triggered on events in the data model make up the majority of the services in Haystack and perform most of the actions on the data in a haystack. Examples of this type of service are the ones that examine a document and try to extract metadata, such as an author extractor. A service in this category first registers an event interest (e.g., the appearance of a new document bale) with the service dispatcher. If this event occurs in the data model, the dispatcher notifies the service, which then carries out its task for that event. For example, the document type guesser is interested whenever a document is archived. When a new document is archived, it examines the document, determines the type of the document, and then creates a new document type tie pointing at a new needle containing the document type and attaches the tie to the document bale in the data model (See Figure 3-1).

Services run from the user interface are responsible for taking user input and putting this information into the data model so other services can trigger off the new data. The query service is an example of such a service. When the user indicates he wants to make a query in the user interface, the query method in the query service is called. After the query method calls outside modules for the ranking of documents, it creates a query bale that contains the query string, the results, and the scores.

Chapter 4

Design of the Learning Module

In order to improve Haystack's searching capability, we decided to introduce machine learning to the system. Our goal in using learning is to present a better set of results to the user by improving the rank of the relevant documents and presenting relevant documents that would not have been shown without learning. We created the learning module to be the component of the system responsible for learning in the system. Our initial design focuses on using relevance feedback to do query expansion and ranking. However, in order to implement relevance feedback, we needed to first change the user interface and the query module. We also had to create several other modules to help us find similar queries, aggregate results, and a vector class based on the vector space model.

Throughout the design, the main consideration was for extensibility. Since this was the initial attempt for learning in the system, we expect that in the future more sophisticated learning algorithms will be tried and the functionality of the learning module will grow. Therefore the main infrastructure of the learning module must be modular and flexible enough to easily handle changing algorithms and the addition of new learning abilities.

4.1 User Interface

The WebGUI and the GUI have buttons that allow the user to rate the relevance of documents returned by queries. After the user marks the relevant documents in the user interface, the user's preference must then be submitted to the system in a way the learning module can find it and use it. The most logical solution is to put the relevance information into the data model so the learning module can be triggered off the new data. Nevertheless, the user interface should not need to understand the anything about how relevance feedback is done in the system, even including how relevance is represented. If the user interface is unaware of all the underlying details then it does not need to change even when the learning algorithm or requirements change. To achieve this modularity, the user interface calls a component in the learning module which takes care of properly putting objects representing user stated relevance in the data model so that learning can be triggered.

4.2 Working with the Query Service

Originally the query service (implemented in HsQuery) supported no learning and simply called the IR system and returned the results. We wanted to add a learning component as well as expand on the types of queries performed to include database queries and any others that might be used in the future. With these additions we have complicated the query process, which can now be viewed as consisting of three major steps: pre-processing, dispatching, and post-processing. Figure 4-1 below depicts the flow of information in completing a query in the original model (on the left) and the revised model (on the right).

The pre-processing that occurs includes both splitting up the query into the different types (text and database) as well as performing any expansion on the query before dispatching it to the appropriate system. The issuing of the queries remains the same except there are more query systems to use. Notice that the query is also

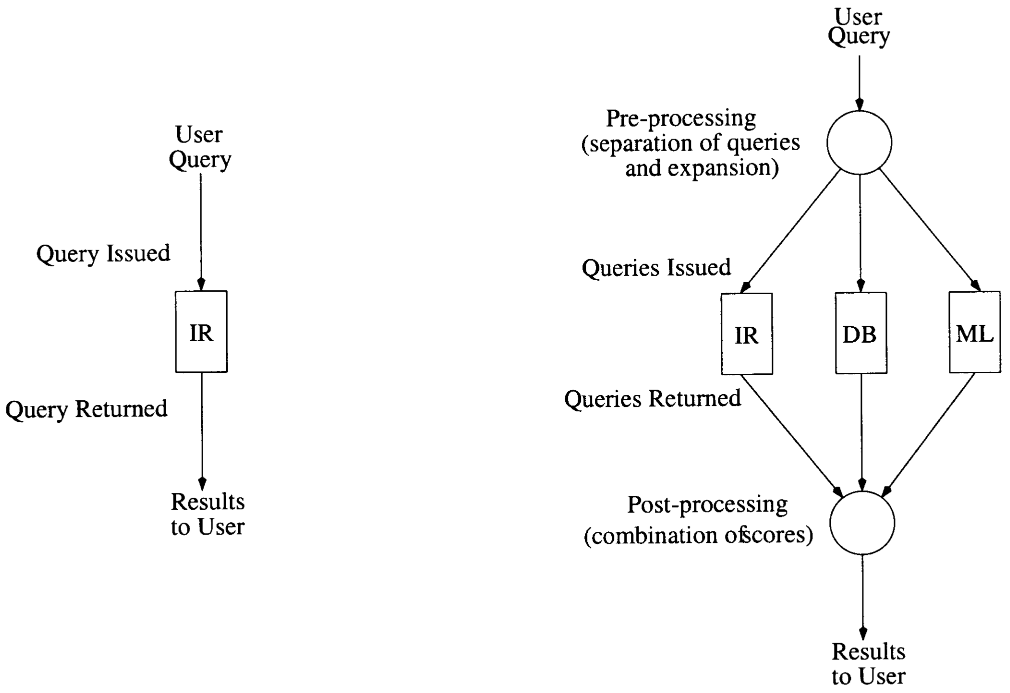


Figure 4-1: Original and New Query Process Models

issued to the machine learning module in parallel with the IR and DB modules. This is because the machine learning module also contributes directly to the final ranking. The feedback the machine learning module gets from the user is not shown in the diagram because it is only showing the data flow from the current query. Post-processing includes any re-evaluation of the results of the systems and aggregating the results from each of the systems. Because the query process has changed, we must revise the role of the query service, HsQuery.

4.2.1 HsQuery as a Query Manager

We have chosen to change the role of HsQuery to that of a query manager, which means that it is responsible for calling the appropriate modules for each step in the query process and for maintaining the flow of information between each of the query and learning modules.

The pre-processing that occurs includes both splitting up the query into the different types as well as performing any expansion on the query before dispatching it to the appropriate system. Because the query manager should not be concerned with the specifics of the individual query systems, the expansion of the queries should be left to an outside subsystem, such as the learning module. The issuing of the queries remains the same except there are more query systems to use than before. Post-processing includes any re-evaluation of the results of the systems and aggregating the results from each of the systems. Combining the different results is a non-trivial task because it is hard to know in advance how much the user will value the rankings of each query system. Therefore the weights on the different results must change over time to accommodate the user's tastes. For example, if the user views textual searches as guides, but structured searches as definitions (they only want documents which satisfy all parts of their query) then the structured search should be weighted more heavily because the user prefers documents with high structured search scores. Again the learning module can be called in to apply the knowledge the system has about the user's opinions on documents to determine how much each query system should be weighted. Thus, the learning module is closely tied to the query manager. Figure 4-2 shows the interaction of the query manager with the learning module.

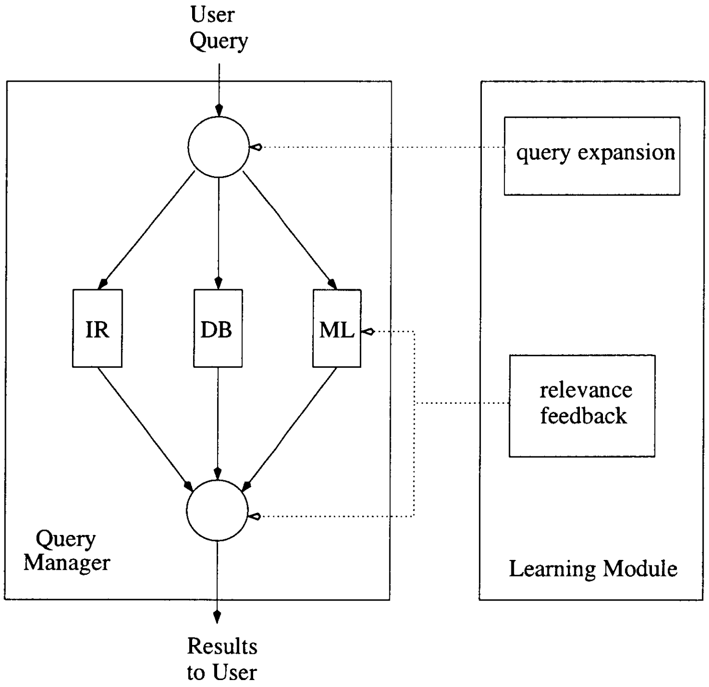


Figure 4-2: Learning Module Interaction with the Query Manager

In preprocessing, we use the query expansion capability of the learning module. In issuing the query, we use relevance feedback to generate a ranking of the documents, and in post-processing we also use relevance feedback to rank the query systems.

4.2.2 Alternative roles for HsQuery

Several other designs were considered for how the query service (HsQuery) should interact with the learning module. The first is to leave the role of HsQuery the same as it was initially implemented, where it is only responsible for acting as the interface between the user and the ranking system. The second is remove HsQuery entirely and have learning be completely integrated in the same module as the query service.

HsQuery as a Dispatcher

Under this design, HsQuery's tasks would include only interacting with the user and the data model while leaving all the work of ranking to a different module. (This was the role of HsQuery prior to the addition of learning and database queries.) The other module would need to work as a query manager, calling and managing the query systems.

This design is probably preferable to the one we chose to implement because it is more modular but does not limit the functionality. The reason we chose grouping the query manager with the other tasks of HsQuery was mostly for simplicity and the desire to get a working model done first with all the extensions of the query service. In the future, the manager can be separated from HsQuery.

HsQuery and the Learner as One

Another design idea was to merge the query service entirely with the learning module, opposite of the previous design. This combined service would take care of all aspects of querying and the user would be able to specify which algorithms they wanted to run by picking which version of the learning query service to use. For example, there could be a RocchioQuery or a BoostingQuery which ran their respective algorithms.

An advantage to this design is that multiple learning query services can run at the same time and the user would be able to pick which they wanted to view. However, this design was conceived before the system was capable of making database queries. With the additional query types now, too many possible configurations of the learning algorithm and the use of the database exist for this design to be practical.

4.3 Relevance Feedback and Query Expansion

Because relevance feedback is applied to query expansion and to ranking documents, we decided to combine both features in one module. Therefore this module is responsible for maintaining the information on the user's relevance ratings for each query and for deciding how to use this information. This module must support the ability to retrieve the user's relevance ratings from the data model, organize and store the data, give terms to expand the query, and to rank a list of documents based on the query's previous relevance ratings.

Since this module is responsible for managing so much information, we decided to separate the actual learning algorithm into a different module which is called by the relevance feedback module. This way we can easily change the algorithm without changing any of this module or duplicating the management of information. Nevertheless, because all the relevance feedback information is stored in this module, it is responsible for compiling a training set from the data and passing it to the module which does the actual learning. The outside module only needs to return scores for documents passed to it by the relevance feedback module after it has been stored.

4.4 Factories and Scorers

We have decided to divide the task of training and scoring into two separate modules because we wanted to emphasize the difference in the two steps. Because training produces a scorer (like a classifier, but returning a value, not just a boolean), the module that does the training will actually create the module that does the scoring. Therefore, we call the training modules factories and the scoring modules scorers.

The factories are the ones to receive training data and use it to train and return a scorer. The factory can also add training data to the data in a scorer and generate a new scorer. A scorer stores the training data used to make it, but is unable to add new data. So a scorer has a state, but it is unable to change its state.

Changing Algorithms

A major concern in the design was the ability to easily switch algorithms without disrupting the flow of the relevance feedback module. Several problems arise while trying to achieve this goal. Because retraining the scorers each time we want to provide test data to the scorers is too inefficient, we want to have a factory train them in advance. However, if we switch factories, then all the scorers from the previous factory become obsolete because they are running the old algorithm. New scorers with the training data from the previous scorers must be made as transparently as possible to the relevance feedback module. To help with this, a scorer can be given to a factory other than its parent factory and give its training data so a new scorer of the new factory type can be created and returned. One issue with this setup is that the relevance feedback module will not know when to resubmit scorers to the new factory. One possible solution is to invalidate the scorers, but this requires some overhead on the part of the relevance feedback module. Another solution would be to move the method of scoring a document to the factory and have the scorer only maintain the training set. This way, if the factory changes, the scoring will also be changed to using the new algorithm. However, this no longer has the desired effect of separating the training from the testing.

Updating Scorers

All of the discussion about how to get the latest scorer might be moot depending on when we choose to update the scorers. Because we are constantly adding documents to the repository, the possible features and their values (those that depend on corpus data) will be changing. With the addition of all types of features, such as database queries, the end result could be a scorer which has been trained on data that was accurate when it was trained, but is now very different. If this is the case then the scorer will act differently than it should if it were given the same training samples (but with additional features and different values) as before. If we choose to update the scorers every time we start a haystack session, the only time a new factory can be loaded, then the scorers will automatically update themselves. We were unable to reach a conclusion about the best time to update the scorers because on one hand, if we are using outdated vectors, then our scorer might also be outdated, but on the other hand, training the scorer is an expensive task especially if the user has accumulated a lot of relevance data in the system. Currently the scorer are not being updated at all. Once we have come to a consensus about the design, we will make the change.

4.5 Similar Queries

We need a similarity module to help us find the previous queries we want to learn from. Initially we want to learn only on the queries that are an "exact match" to the current one, meaning they have the exact same query string. Looking only at these queries is generally what is done in systems, however, we also want to add the ability to do *cross-query learning*, that is learning from queries that are similar, but not exactly the same as the current query. However because the data from these other queries is not in response to this query exactly, our confidence in how applicable they are will not be as high as our confidence in data about the current query. The similarity function we use and how we use it need to address these issues.

Since there are many different types of similarity measures, we decided to make the similarity check a separate module that can be easily switched. This module only needs the ability to say if the current query is an exact match on a previous query or is considered similar to a previous query. An exact match method is needed because if the query is the same as, not just similar to one before, then we do not want to create another query bale for it, but rather we want to use the same bale as before. This method will return the appropriate bale if one exists. We also need a separate method which determines if two bales are considered similar so the query manager can pass the similar bales to the relevance feedback module, which may use their previous relevance ratings.

4.6 Aggregator

The role of the aggregator is important because it is the last step in learning and the results of the aggregator are the results shown to the user. Thus, it needs to accurately decide how valuable each query system is and how much the final ranking should depend on each system. The aggregator gets a list of rankings from the query systems and needs to combine the scores to produce a final ranking. We have decided to start with a simple idea where each ranking is given a weight and the overall ranking is the sum of the weighted scores. In order to determine the weights, we will use relevance feedback where each feature is the score assigned by one of the querying systems (currently these are the IR system, the database query system, and the learned results). So if the user consistently likes documents which satisfy a particular type of query, then the corresponding query system will get a higher weight in the vector.

4.7 Feature Vectors

We will represent documents and the set of query systems in learning with feature vectors. When limiting the type of queries to textual queries, the features are words (or terms) only. The set of words used is the set of all words found in documents in the corpus after stemming and removing stop words as described in section 2. However, to expand the capabilities of the learning system to cover non-textual queries as well, we can generalize the components of the vectors describing the documents. The only constraints on the features in a vector are that we must be able to uniquely identify the feature and obtain its value from the vector. Specifically this means we will need to be able to get its identification number, the value of the feature in that vector at that time, and the type of feature. These requirements are needed to ensure that the learning algorithm can learn on the features. Nevertheless, they are flexible enough to allow several different combination of features. Below, in figure 4-3 we show several possible ways to combine the features into vectors.

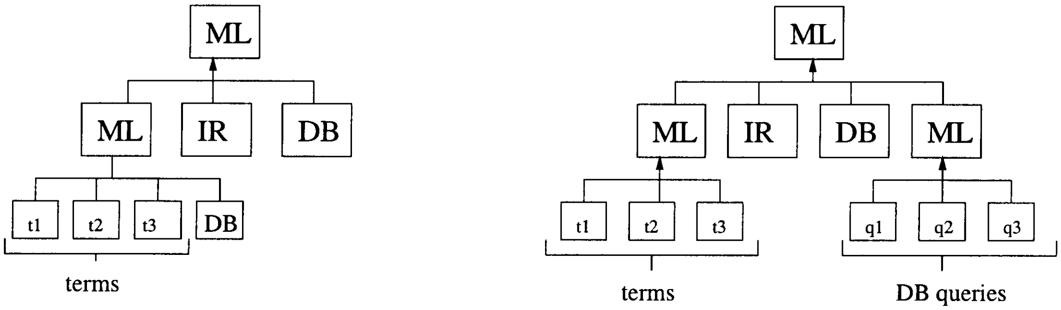


Figure 4-3: Possible Feature Vector Configurations

The boxes going into the ML box represent the different features used in the feature vector for machine learning.

The tree on the left has words as well as the results of all the database queries in the same vector. The advantages of this design is that it is simple to implement and we capture the characteristics of the database query part in learning as well. For instance, imagine the user is looking for articles on "President Clinton's inauguration" and selects "date = 1992." When the results are returned, the user marks that he or she found only the articles from 1992 relevant. If later they only input "President Clinton's inauguration" without a database component, the system would return the articles where "date = 1992" with a higher score than if the system did not consider the date.

The tree on the right shows a design where we separate the word features from the database query features. This design might be preferable to the previous design because we are unsure how practical comparing terms and database queries is. For example, unless we properly weight the features, if the number of queries is much higher than the number of terms, then the impact of the words in the document will be almost negligible and the learning will depend solely on the DB query scores in the vector. Other difficulties might also arise because the space of terms and of queries is inherently different so combining them in one vector space and comparing their values to each other might not make sense. While the design on the right separates these types of features, it does not rule out learning on the DB portion of the query. Instead of having one large learning module for all types of queries, we could have learning modules specific to the type of the query. The drawback to separating the query types though, is that we lose some of the importance that a document satisfies all parts of the query if we evaluate the scores for each query separately.

Our current design is the design on the left. Other designs which combine queries, query systems, and terms together in feature vectors in different ways are also plausible. Since we cannot predict which design will work best in practice, we have designed the feature vector to be flexible enough to handle either design we have mentioned and most others using the vector space model.

Chapter 5

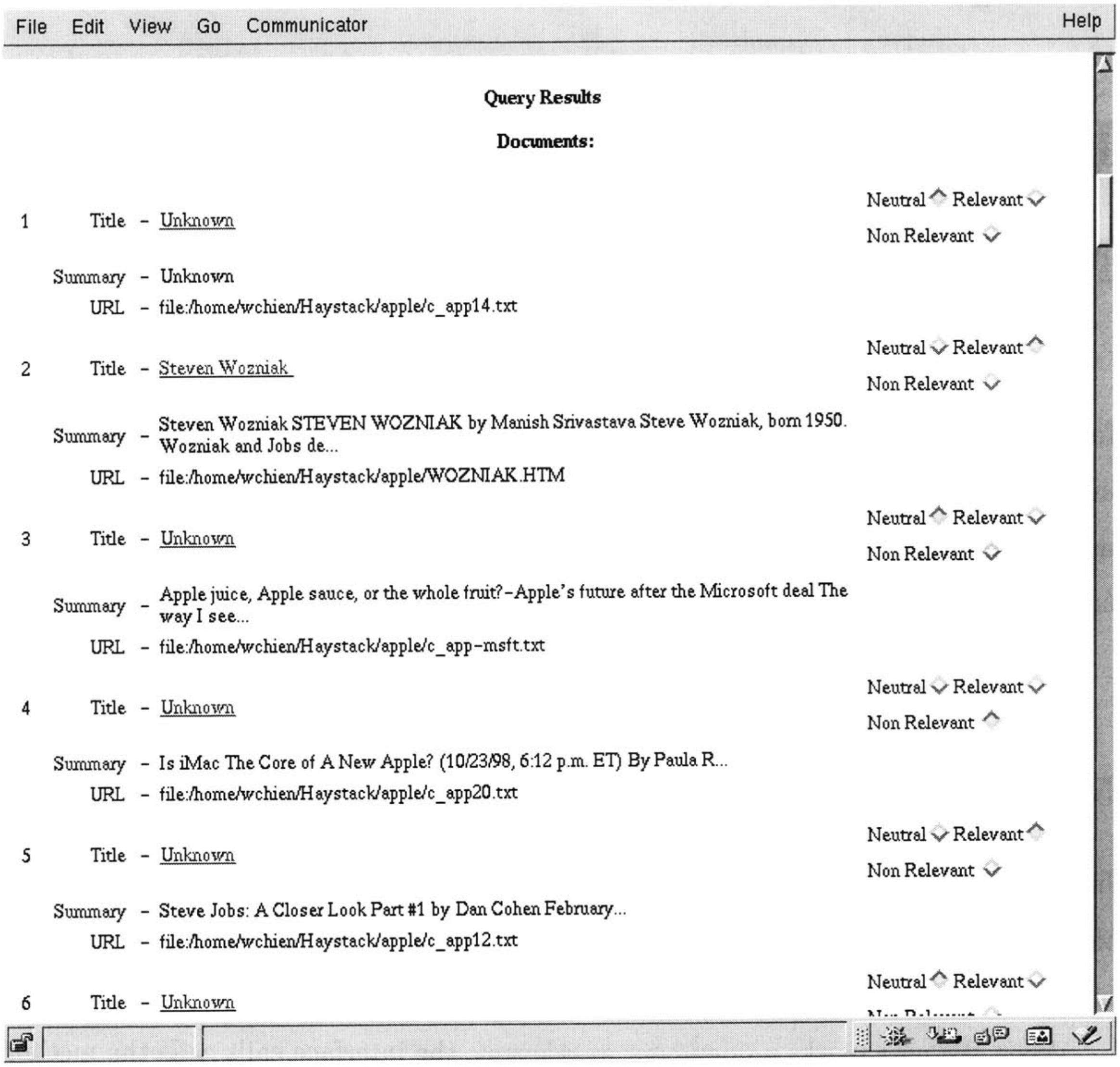
Implementation of the Learning Module

In this chapter we discuss the details of the implementations of the designs described in the previous chapter. We will review all the classes and interfaces involved in each module as well as the specific implementations of the algorithms we have chosen to use.

5.1 User Interface

In order to have relevance feedback, the user interfaces only need to include buttons that let the user to mark documents as being relevant or non-relevant when a query is returned. Figure 5-1 shows the WebGUI with the new buttons.

When the user marks a document as relevant, the interface calls calls the method `HsLearner.setRelevance(HaystackID, HaystackID, boolean)` which is given the query bale id, the document id, and a positive or negative rating on that document for that query. HsLearner acts as the interface to the learning module and is responsible for putting in ties of label "Tie.PosSample" or "Tie.NegSample" between the query bale and the document straw in the data model. In order not to over-count the documents if the user repeatedly rates the same document, `HsLearner` ensures that



Query Results

Documents:

1 Title - Unknown — Neutral ◆ Relevant ◇ Non Relevant ◇

Summary - Unknown

URL - file:/home/wchien/Haystack/apple/c_app14.txt

2 Title - Steven Wozniak — Neutral ◇ Relevant ◆ Non Relevant ◇

Summary - Steven Wozniak STEVEN WOZNIAK by Manish Srivastava Steve Wozniak, born 1950. Wozniak and Jobs de...

URL - file:/home/wchien/Haystack/apple/WOZNIAK.HTM

3 Title - Unknown — Neutral ◆ Relevant ◇ Non Relevant ◇

Summary - Apple juice, Apple sauce, or the whole fruit?-Apple's future after the Microsoft deal The way I see...

URL - file:/home/wchien/Haystack/apple/c_app-msft.txt

4 Title - Unknown — Neutral ◇ Relevant ◇ Non Relevant ◆

Summary - Is iMac The Core of A New Apple? (10/23/98, 6:12 p.m. ET) By Paula R...

URL - file:/home/wchien/Haystack/apple/c_app20.txt

5 Title - Unknown — Neutral ◇ Relevant ◆ Non Relevant ◇

Summary - Steve Jobs: A Closer Look Part #1 by Dan Cohen February...

URL - file:/home/wchien/Haystack/apple/c_app12.txt

6 Title - Unknown — Neutral ◆ Relevant ◇

Figure 5-1: WebGUI Screen Shot

only the first rating given to the document from that query is used. It is also possible to only use the last rating given but because using the first rating has a much simpler implementation, it was chosen. Now that we have gotten a first version of the learning module running, we can change this aspect of the interface to using the last rating because it is more user friendly and correct.

So far, the only interface which supports learning is the WebGUI. When the user marks a document as being relevant, the webscript `relevantResult.java` calls `HsLearner` with the data. The GUI has the markers in place, but has not been changed to call HsLearner. The command line interface must be expanded to accept commands which set the relevance ratings as well as call HsLearner.

5.2 Query Manager

We implemented the query manager in the query method of `HsQuery`. It is responsible for calling the database query system, the IR system, the similar query checking service (`HsSimilarQuery`), the relevance feedback module (`HsRelevanceFeedback`) for both query expansion and ranking, and the aggregator (`Aggregator`). Figure 5-2 shows the control flow of the query manager.

The data from each module is returned to HsQuery which then decides which other modules need that information. This design was chosen over having the modules communicate with each other to simplify the implementation as well as reduce the dependence of the modules on each other.

Future designs of the query manager might reduce the responsibility of the query manager and have separate pre-processing, dispatching, and post-processing modules which handle the passing of data. So instead of passing the data through the query manager, these smaller modules would have knowledge about to which module it needs to pass its data. For example, the preprocessing module is given a query by the query manager. It separates the query into its different parts and makes calls to outside systems to expand the parts of the query. Since it knows the dispatching

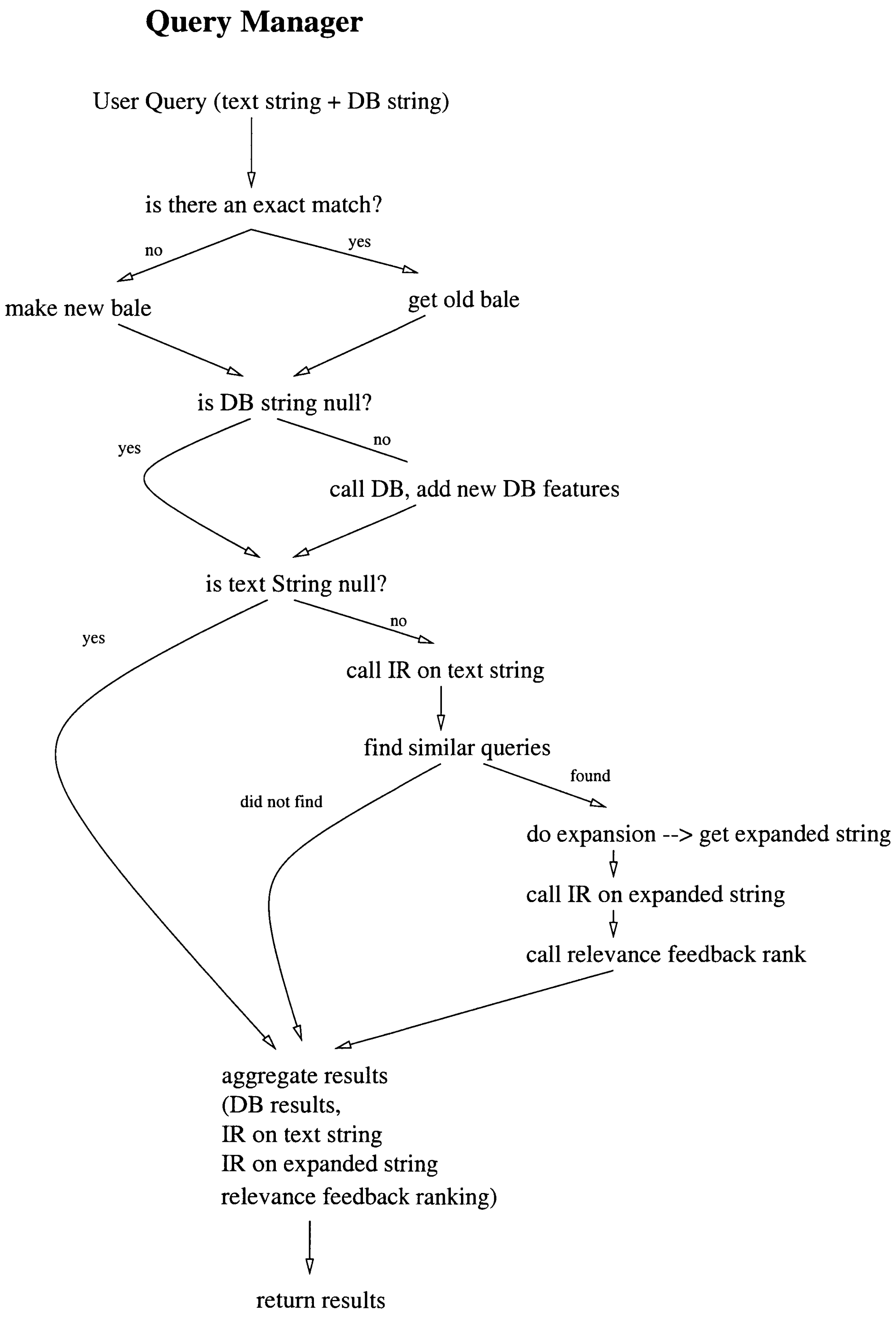


Figure 5-2: Query Manager Control Flow

module is next in line in the query process, it passes its separated expanded queries to the dispatching module. After the dispatching module has issued all the query parts to the query systems, and received the results from each system, it passes them to the post-processing module, where the results are combined. The post-processing module then returns the combined results to the query manager.

5.3 Relevance Feedback

The Relevance Feedback module is responsible for getting the data on which documents the user finds relevant and not relevant, as well as containing a method which ranks a list of documents and a method which returns the new terms to add. Because there are several ways to use learning algorithms to come up with a ranking of documents, we designed an interface `HsRelevanceFeedback` which all classes providing relevance feedback need to implement. `HsRelevanceFeedback` itself extends `HaystackEventService` in order to detect when relevance ratings are added to the data model.

5.3.1 HsRelevanceFeedback

The required methods of any class that provides relevance feedback are listed below.

- **addInterest**: The class must register its interest of a bale (the query) attached to a straw (the document) with a relevance label tie ("Tie.PosSample" or "Tie.NegSample.").

- **handleHaystackCreateEvent**: When this method is called, we know new relevance information has appeared in the data model. The module needs to read the new relevance information and store it so it can be used the next time a similar query is issued.

- **rank**: This method will take in a vector of RankedHaystackIDs, a list of similar query bales and the current query bale. The similar bales and current bale provide the relevance information we need. Because we cannot afford, nor do we have the ability to run our scorer on the entire corpus, we use a filter to limit the number of documents we rank. We use the vector of the other systems' rankings as a filter and only score documents contained in the ranking of one of the query systems. This is a pretty safe choice to make because it is unlikely there is a relevant document not retrieved by one of the query systems after query expansion is done. The rank method returns a ranking of documents based on the relevance information. No restrictions on which documents are included or how they are ranked are made.

- **getImportantFeatures**: This takes in the number of terms to return as well as a vector of the similar query bales to use for the relevance information. When called, this method returns a list of terms that can be added to the query.

5.3.2 Implementations

So far two implementations of this interface exist, `HsSingleScorer` and `HsBoosting`. The only difference between them are how they handle ranking documents and all the various data structures they need in order to implement the two algorithms.

HsSingleScorer.rank

As the name suggests, `HsSingleScorer` is the implementation of `HsRelevanceFeedback` which uses only one scorer to help in ranking the documents. So for each query topic, one scorer keeps track of all the training data and is responsible for returning the score for each document being ranked. Therefore in the rank method, all the documents being ranked are passed to the same single scorer and are ranked by the score returned. In order to increase efficiency a table is used to keep track of all the current scorers. Because of this, we must be careful that we are using a scorer running the correct algorithm (as described in the previous chapter). The current implementation does not address this problem because we have not yet come to a conclusion about the appropriate time to update the scorers.

HsBoosting.rank
This class is the implementation of the AdaBoost boosting algorithm. When the rank method is called, it creates several scorers of the same type (running the same learning algorithm) by varying the weights used on the training set. A document is given a final score based on the opinions of the all the scorers.

HsSingleScorer.getImportantFeatures and
HsBoosting.getImportantFeatures:
This method is the same in both implementations because both are currently using automatic query expansion where we add the most frequently appearing words in the relevant documents. Right now this method is implemented in the relevance feedback module, though in the future it should be moved to a separate module, in the same way as the ranking algorithm, to ease in switching between algorithms.

5.4 Scorer Factories and Document Scorers

The factory-scorer pair represents a learning algorithm we want to use. The purpose of the scorer factory is to return scorers when given the training data. The document scorer is able to take in a document and return a score based on how it was trained. Because we would like to be able to change algorithms easily, we have designed an interface called `ScorerFactory` which details the methods that a scorer factory needs as well as an interface `DocumentScorer` which describes all the methods a document scorer needs.

5.4.1 Interface ScorerFactory

In the scorer factory, we only need the ability to create a new document scorer when given training data and an old document scorer. The method declared by this interface is getScorer(training set, document scorer) and should work as follows:

- **Case: training set = null, document scorer = null** If both are null, then getScorer also returns null.

- **Case: training set = null, document scorer ≠ null** If there is no training data and the scorer was not made by this factory, then a new scorer from this factory is returned, otherwise the same document scorer is returned.

- **Case: training set ≠ null, document scorer = null** If there is no previous document scorer, then a new document scorer with the training data is made.

- **Case: training set ≠ null, document scorer ≠ null** If both the training data and the old document scorer exist, then a new document scorer is returned with the new training data combined with the old training data.

5.4.2 Interface DocumentScorer

The main task of the document scorer is to give an opinion about a document based on the training data with which it is created. The specific methods are described below.

- **score**: score(document) This method returns a score when given a document in the form of a Straw object or a FeatureVector. The score needs to be a value between 0 and 1.

- **classification**: In addition to giving a score, this module also needs to be able to state a boolean classification (relevant or non relevant) for a document in both representations (straws and vectors). This method is used by algorithms that require a boolean decision.

- **getTrainingData**: In the case that a scorer is returned to a factory other than the one which created it, this method allows the new factory to get the training information and create a new scorer of its own type.

The class which implements the `DocumentScorer` interface is an inner class of the class which implements the `ScorerFactory` interface for the same algorithms. Under this design, the parent factory of the scorer will be able to use any methods of the scorer that are specific to the algorithm being implemented by the pair. If in the algorithm, adding a training sample is possible without reviewing all the previous training examples, then the scorer can supply a method to make updating the scorer more efficient. For example, in Rocchio when we want to add a new piece of training data, we only need to add the vector of the piece of training data to the optimal query.

5.4.3 Rocchio Implementation

One implementation of the `ScorerFactory/DocumentScorer` interfaces currently exists. We have chosen to use Rocchio's algorithm to create the two classes `RocchioFactory` which implements `ScorerFactory` and `RocchioScorer` which implements `DocumentScorer`. In addition to the methods of `DocumentScorer`, `RocchioScorer` also contains a method `getVector()` that returns the vector that represents the "optimal query," and a method `setVector()` that allows the factory to set a new vector.

5.5 Similarity

The checking of similarity between queries is done by the class `HsSimilarQuery`. To keep a list of all the queries that have been made, it extends `HaystackEventService` and registers an interest with the dispatcher for the appearance of query bales. When a new query bale appears, the module adds it to the list of bales that already exist in the data model. This class also offers three methods for similarity.

- **isExactMatch(string)**: This method is needed to check if a query already exists in the data model. It is given the query string of the current query and compares it to all the previous query strings to see if it is exactly the same query as a previous one. If it is, it returns the query bale of the query.

- **isSimilarTo**: This method is used to give a confidence rating for how similar a bale is to another query bale. When given two sets of rankings, this method returns a value between 0 and 1 of how similar they are, with 0 being not similar at all and 1 being exactly the same.

- **hasSimilarResults**: This method is called to given two sets of rankings and it returns true if the two lists are considered "similar," and false otherwise. Similarity in this case is measured by how similar are the sums of the feature vectors. Similarity between the vectors is computed in the standard way in this system, using the normalized dot product.

5.6 Aggregator

The aggregator module is implemented by the class `Aggregator`. It has one main method called combine which is given a list of rankings from the different query systems and combines them to return a single final ranking. In the current implementation, combine produces a final set of rankings by finding the weighted sum of the rankings it has been passed, where the weights are preset. When the `FeatureVector` interface is implemented and query systems can be represented as features in a vector, relevance feedback will be used to adjust the weights dynamically, so query systems which have higher scores on relevant documents will be given a higher weight.

5.7 Features and Feature Vectors

We currently have not implemented the FeatureVector class, however we do have a design, shown in Figure 5-3, for the classes which would make up the Feature and the FeatureVectors. This design is a result of collaboration with Jaime Teevan, Svetlana Shnitser, and Ziv Bar-Joseph, who are also working on the Haystack project.

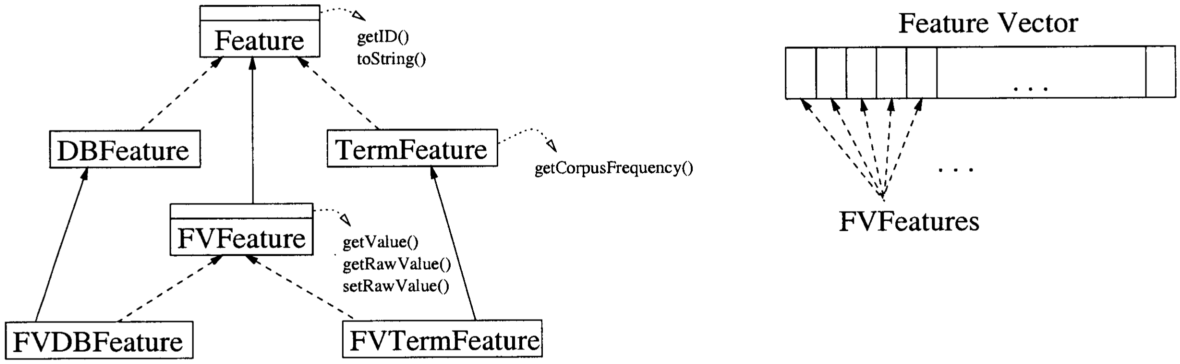


Figure 5-3: Feature and Feature Vector Module Dependency Diagram.

In the figure we see the different methods contained by each of the classes for the features and the feature vectors.

5.7.1 Features

The key idea in this model for features is that they are themselves objects with an identity and methods, but their values are dependent on the specific vector which contain them. In order to model this behavior we create two interfaces, one that describes the methods of a stand alone feature, and one that describes the methods of a feature in a vector.

The interface `Feature` maps out the methods required for a feature. Because a feature has an identity on its own without the presence of a vector, the `Feature` interface requires a `getID()` method and a `toString()` method. These are implemented by the classes `DBFeature`, `TermFeature`, `QuerySystemFeature`, and any other types of feature we want to create. In addition to the two methods of the `Feature` interface, the classes that implement the interface can have other methods specific to the type of `Feature` it is. For example, the `TermFeature` class also has the method getCorpusFrequency() that returns the frequency of this term in the whole corpus. This method appears at this level because the values returned by them still have meaning even if the feature does not appear in a vector. Each of these implementing classes also have a subclass that represents that type of feature in a vector and implements the `FVFeature` interface.

The `FVFeature` interface describes the methods that each feature in a vector needs for learning. As we described in the design sections, the only additional methods needed for learning is the `getValue()` method (because the id and the type of feature can be found from the classes which implement `Feature`). The `getValue()` method is dependent on the type of feature. For instance, the value of a term feature might be affected by the number of times that term appears in the corpus, where a DB feature value might not take that into account.

5.7.2 Feature Vectors

Feature vectors are, as one would expect, vectors made up of objects of classes implementing the `FVFeature` interface. The methods in `FeatureVector` are ones that deal with the vector as a whole but most use the values of the features it contains to return answers. The methods of the `FeatureVector` will at least include `getNormalizedVector`, `scale`, `add`, `clone`, `dot`, `equals`, and will probably increase as more applications are added that need to use them.

5.7.3 Word Vectors

As we stated above, Features and Feature Vectors have not yet been implemented. Until they are, we are using the classes `Term`, `WVTerm`, and interface `WordVector` which have almost the same functionality as `Feature` and `FeatureVector` described above. Originally `WordVector` was designed for features that were only the words in documents so it holds elements of the class `WVTerm`. A `WVTerm` contains a `Term` object, which is a string and id pair, as well as the frequency of that term in the vector. `WordVector` itself has many of the methods that `FeatureVector` will have, including `add`, `clone`, `scale`, `equals`, `getNormalizedVector`, `getRawVector`, `dot`, `normalizedDot`, `getSize`, and `getLength` methods. `WordVector` is implemented by the class `Profile` originally designed by Lisanskiy [LI99] and modified to fit the `WordVector` interface. It is important to note that the only normalization done by profiles is by the length of the vector. Profiles currently have no knowledge of the corpus as the implementations of FeatureVectors should have.

Chapter 6

Testing and Results

We have completed two types of tests on the learning module in Haystack. The first set of tests use TREC data and provide assurance that our module is working correctly. The second set of tests are run on a hypothetical example which describes a situation where the personalization in Haystack allows us to make a distinction where other search engines would not.

6.1 TREC Tests

We ran our system on a small subset of the TREC data. More specifically, we archived 200 Wall Street Journal articles from 1990 found on TIPSTER disk 2 and queried them with TREC Topic 1. The reason we only used a fraction of the data available from TREC was that our system was unable to run efficiently when given too much data to store. Without enough time to determine the reason for the slowdown in our system, we attempted to run a set of smaller tests on the data we could store.

6.1.1 Data Preprocessing

The TREC articles are stored in files which contain around 130-190 distinct articles each. The data has also been changed to standard Standard Generalized Markup

key	TREC Document Number (DOCNO)
DOC1	WSJ900625-0090
DOC2	WSJ900629-0045
DOC3	WSJ900702-0034
DOC4	WSJ900703-0065
DOC5	WSJ900706-0008
DOC6	WSJ900713-0067
DOC7	WSJ900716-0039
DOC8	WSJ900719-0126
DOC9	WSJ900803-0089
DOC10	WSJ900807-0031
DOC11	WSJ900808-0106
DOC12	WSJ900810-0086

Table 6.1: List of Relevant Documents

Language (SGML) format. We ran the files through a program that extracted the articles from the file and removed the SGML tags leaving plain text.

Because of our limitation of only being able to store about 200 documents, we needed to select the files we chose to archive carefully. For each query, the percentage of the TREC articles which are considered relevant is extremely low (around .1%). Therefore in order for us to have several relevant documents in our corpus for a particular query, we had to increase the percentage of relevant documents for our corpus. Unfortunately, changing the composition of the corpus can skew the results. Imagine a corpus where 1% of the documents are relevant to a query, but it is artificially increased to 50%. When the retrieval algorithm is run and a lot of relevant documents are returned, we do not know if the good results are from a good algorithm, or if it is just a result of returning a lot of documents. With this in mind, we chose to archive twelve documents relevant to topic 1 (total percetage 6.0%), giving us enough relevant documents to do some testing. The twelve documents we chose are listed in Table 6.1.

The TREC document number is listed (DOCNO) along with a name we will use in the results below.

6.1.2 Testing Procedures

We wanted to test both the Rocchio and the boosting algorithms with our tests. In general, to test a learning algorithm, data with the correct classifications attached is used. The data is split into two sets, one for training and one for testing. The training data is used to train the classifier running the algorithm. The testing data is then given to the classifier to classify. Since we have the correct classifications for the test data, we can see how much of the test data the classifier classified correctly.

We use the same basic strategy for our tests. But since we have so few relevant examples, we performed cross validation. Cross validation is for situations like ours where there is a limited amount of training and test data. In cross validation, we split all the data we have for training and testing into several smaller sets. We designate one subset to be our test set while the other subsets make up the training set. Then we test using the general methodolgy described above. We repeat this process until every subset has been used as the test set. For our tests we are going to do leave-one-out cross-validation, which is where the subsets are of size one.

The outline of our testing procedure is as follows:

1. Start with a clean haystack.
2. Archive the 200 selected documents.
3. Select one document to leave out.
4. Submit the query (Topic 1: "Antitrust Cases Pending"). No learning will be done on this query because there are no previous queries.
5. Mark relevant all the relevant documents except for the one selected above. (none marked non-relevant)
6. Rerun the query.
7. Examine the rankings of the unmarked relevant documents.
8. Go back to the first step, until all of the documents have been left out once.

	unmarked document											
	orig	none	DOC1	DOC3	DOC4	DOC5	DOC6	DOC7	DOC8	DOC9	DOC10	DOC12
DOC1	27	13	50	12	13	13	12	13	13	12	12	13
DOC2	-	8	10	7	8	24	7	7	8	8	11	11
DOC3	5	6	6	11	6	4	6	5	7	5	5	4
DOC4	10	19	20	19	33	16	19	19	21	18	18	25
DOC5	11	2	2	2	2	6	2	2	2	2	2	2
DOC6	4	9	9	9	9	5	14	10	10	9	8	8
DOC7	16	5	5	5	5	9	5	8	5	6	3	5
DOC8	12	12	13	16	12	11	11	12	15	13	9	9
DOC9	28	52	45	51	47	38	52	62	52	102	35	31
DOC10	23	3	3	3	3	2	3	3	3	3	6	3
DOC11	-	-	-	-	-	-	-	-	-	-	-	-
DOC12	1	1	1	1	1	1	1	1	1	1	1	1

Table 6.2: Rankings using Rocchio's algorithm.

6.1.3 Results and Analysis

We chose to run four sets of tests. We ran tests with both query expansion and ranking from relevance feedback for both Rocchio and boosting. In these tests, the weights for the IR rankings, the IR rankings with the expanded query, and the relevance feedback ranking were 0.5, 0.5, and 1.0, respectively. We also ran tests without using query expansion, but still using ranking from relevance feedback running for both Rocchio and boosting. In these tests the weight for both the the IR ranking and the relevance feedback ranking was 1.0. The database queries were neither tested nor used here.

Rocchio

We begin by looking at the tests for using a single Rocchio scorer. The results of these tests are shown in Table 6.2. Each column represents a different test done. The original query (one without learning) is labeled "orig." The rest of the columns are the tests where the query was performed after documents were marked relevant. The test labeled "none" is the one where all ten of the relevant documents returned by the original query are marked relevant. The rest of the tests are labeled by the relevant document that was used as the test point and therefore was not marked relevant. (The reason there are fewer documents in the columns than in the rows is that only ten of the relevant documents were returned by the original query, so only those ten could be marked relevant.) The rows represent the relevant documents in the corpus, and therefore a value in a box is the rank of that row's relevant document for that column's test.

We can see from the table that the overall rankings of the document is better.

	unmarked document											
	orig	none	DOC1	DOC3	DOC4	DOC5	DOC6	DOC7	DOC8	DOC9	DOC10	DOC12
DOC1	27	14	24	14	13	16	14	13	15	12	12	12
DOC2	-	-	-	-	-	-	-	-	-	-	-	-
DOC3	5	6	6	9	6	4	6	5	8	5	5	4
DOC4	10	13	13	12	20	12	13	14	13	13	13	15
DOC5	11	2	2	2	2	6	2	2	2	2	2	2
DOC6	4	9	9	8	9	9	11	8	9	9	8	7
DOC7	16	5	5	5	5	5	5	9	5	7	5	6
DOC8	12	10	10	13	10	10	10	10	11	10	10	10
DOC9	28	24	23	24	24	23	25	27	24	30	23	22
DOC10	23	3	4	4	4	3	4	3	4	4	9	4
DOC11	-	-	-	-	-	-	-	-	-	-	-	-
DOC12	1	1	1	1	1	1	1	1	1	1	1	1

Table 6.3: Rankings using Rocchio's algorithm with no query expansion.

DOC2, which was not returned in the original query, is returned for all the learned queries because the IR system called on the expanded query string retrieved it.

If we look at the ranking of the each document in the original query vs when it is the test document (the one not marked relevant) nine documents became worse, three improved, and one stayed the same. We think that these results are due to the fact that we are only giving it nine training documents and because TREC judges a document as "being relevant if any piece of it is relevant (regardless of how small the piece is in relation to the rest of the document)" [NST00]. This means that the non-relevant parts of the documents could be very different from each other. If this is the case, then nine documents will not be enough to train a learner.

Another oddity was that DOC9, even when marked relevant, would still drop in the rankings. When we examined the documents, we noticed that most of the relevant documents contained the words "law," "court," "judge," "FTC," as well as the words in the query. Unlike most of the other relevant documents, DOC9 is not about an antitrust case, but instead is about a company's financial status. The reason this document is retrieved is because it also mentions the company is involved in an antitrust case.

After seeing how much worse some of the rankings became because many new documents are retrieved as a result of the expanded query, we decided to also try running the same tests with query expansion disabled. Table 6.3 shows the results of these tests. Looking the results in Table 6.3 we see that again the overall rankings are better. If we look at the ranking of the each document in the original query vs when it is the test document, as we did before, we see that this time nine documents

	unmarked document											
	orig	none	DOC1	DOC3	DOC4	DOC5	DOC6	DOC7	DOC8	DOC9	DOC10	DOC12
DOC1	27	15	31	15	15	82	14	19	15	15	18	16
DOC2	-	28	28	35	31	146	31	147	34	31	143	139
DOC3	5	7	6	9	9	15	9	12	9	9	14	14
DOC4	10	16	15	16	13	12	15	14	16	16	15	12
DOC5	11	4	7	4	4	56	4	11	4	4	11	10
DOC6	4	5	5	5	5	6	5	6	5	5	6	6
DOC7	16	18	14	23	22	133	22	63	23	22	130	104
DOC8	12	12	12	11	12	25	12	15	12	14	16	17
DOC9	28	34	32	33	34	48	34	47	33	29	59	73
DOC10	23	13	11	21	16	145	17	138	20	13	63	112
DOC11	-	-	-	-	-	-	-	-	-	-	-	-
DOC12	1	1	1	1	1	2	1	2	1	1	2	1

Table 6.4: Rankings using AdaBoost.

improved, only three became worse, and one stayed the same.

Boosting

Boosting in our system did not work as well as we had hoped. Table 6.4 shows the results of our tests.

We think the reason why these results are quite poor are partly because we used so few training points and boosting requires a lot more training data [SSS98]. Not only are the rankings of the documents that are being left out not improving, but even the rankings of the training data points not improving. Once the efficiency problem is solved in the system, more boosting tests should be done.

6.2 Example

The second type of tests we ran were not to test the correctness, so much as to give an example where the personalization of Haystack can help improve a query result set. We archived thirty-five documents about Apple computers and thirty-five recipes containing apples. When the query is run initially, the majority of the top ranked documents are about Apple computers. We selected ten apple recipes to mark as relevant and reran the query. After the last run (which uses learning), the top thirty ranked documents were all apple recipes. This example shows that if there is a word with two meanings, like apple, the personalization of Haystack will allow the user to say which meaning they are interested in, and so the system will be able to provide better results.

Chapter 7

Conclusions and Extensions

The goal of this thesis was to take advantage of the personalized aspect of Haystack to learn about the individual users' query behavior. By learning about the interests of the user, we will be able to better predict what the user is searching for when he makes a query and therefore be able to produce better results. To achieve this goal we designed and implemented a learning module that uses relevance feedback and applies it to query expansion and scoring documents. We chose Rocchio's algorithm and boosting as our first algorithms to implement because both are relatively simple and have been shown to perform will in studies. Our tests show that the algorithms are running correctly in our system and that we are able to learn from past queries.

In addition to writing a working system, we have also been careful to design the learning module so that it is extensible. Since this is only the first pass at adding learning to the system, we expect that in the future more sophisticated learning algorithms will be tried as well as different approaches to learning.

This work is only the beginning of many possibilities for learning in Haystack. Future work in this area of Haystack include many projects. Some of the more immediately feasible work is in adding learning to the Aggregator, extending query expansion to become interactive, trying more sophisticated algorithms. More involved projects include adding the abiblity to intelligently learn from all previous queries, not just ones that are similar and developing learning algorithms specifically for Haystack.

With the addition of structured queries and machine learning rankings to our longtime textual queries, we need a way to intelligently aggreagate the scores of the different systems to produce the best combination for the user. Learning can help in this situation because from the user's relevance ratings and from the scores that each of the ranking systems gave each relevant document, we will get an idea of which system the user relies on the most. By giving those systems the user agrees with a higher weight, we change the aggregation to also rely more on those systems.

Extending the system to support interactive query expansion should be fairly straightforward because it can use a lot of the infrastructure in place for automatic query expansion. The main change will occur in the user interfaces becaue we will need to display the possible expansion terms. The words we choose to display to the user can be drawn from the list of words that is already compiled by automatic query expansion.

Because of the modular design of the learning module, adding new algorithms is very simple. This means it should be easy to implement more learning algorithms to use for Haystack. Some possible algorithms to try are naive Bayes and k-nearest neighbor.

Adding the ability to learn from previous queries for a topic the system has never seen before is a much more involved problem. We first need to be able to know how the current topic is related to previous topics. We then must apply this knowledge about previous related topics to to our current query.

Haystack has several distinctive characteristics which could be used in developing a learning algorithm specifically for it. For instance, the range of topics found in a user's haystack is much more likely to be clustered than a corpus like the World Wide Web because a user is likely to only have a certain number of interests. Also, the personalized nature of Haystack and the involvement of the system in everything the user does allows us to gather much more information about the user than most other systems.

Bibliography

[A98] E. Adar. Hybrid-Search and Storage of Semi-structured Information. Master's Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1998.

[AW00] AntWorld webpage `http://aplab.rutgers.edu/ant/`

[DH00] Direct Hit webpage `http://www.directhit.com`

[G98] G. Greiff. A Theory of Term Weighting Based on Exploratory Data Analysis. *Proceedings of SIGIR '98*, August 1998, pp. 11-19.

[H92] D. Harman. Relevance Feedback and Other Query Modification Techniques. *Information Retrieval: Data Structures & Algorithms*, pp. 241-263, Upper Saddle River, NJ, 1992. Prentice Hall, Inc.

[HRA92] D. Harman. Ranking Algorithms. *Information Retrieval: Data Structures & Algorithms*, pp. 363-392, Upper Saddle River, NJ, 1992. Prentice Hall, Inc.

[LI99] I. Lisanskiy. A Data Model for he Haystack Document Management System. Master's Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 1999.

[LOW99] A. Low. A Folder-Based Graphical Interface for an Information Retrieval System. Master's Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1999.

[MSB98] M. Mitra, A. Singhal, C. Buckley. Improving Automatic Query Expansion. *Proceedings of SIGIR '98*, August 1998, pp. 206-214.

[NST00] NIST TREC webpage `http://nist.trec.gov`.

[RW92] S. Robertson, S. Walker, M. Hancock-Beaulieu, A. Gull, M. Lau. Okapi at TREC *NIST Special Publication 500-207: The First Text REtrieval Conference (TREC-1)*, November 1992, pp. 21-30.

[RW93] S. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, M. Gatford. Okapi at TREC-2 *NIST Special Publication 500-215: The Second Text REtrieval Conference (TREC-2)*, August-September 1993, pp. 21-34.

[R71] J.J. Rocchio. Relevance Feedback in Information Retrieval. *The SMART Retrieval System–Experiements in Automatic Document Processing*, pp. 313-323, Englewood Cliffs, NJ, 1971. Prentice Hall, Inc.

[SB90] G. Salton, C. Buckley. Improving Retrieval Performance by Relevance Feedback *Journal of the American Society of Information Science*, 41(4):288-297, 1990.

[S99] R. Schapire. A Brief Introduction to Boosting *Proceedings of the Sixteenth International Conference on Artificial Intelligence*, 1999.

[SSS98] R. Schapire, Y. Singer, A. Singhal. Boosting and Rocchio Applied to Text Filtering *Proceedings of SIGIR '98*, August 1998, pp. 215-223.

[S00] S. Shnitser. Integrating Structural Search Capabilities into Project Haystack Master's Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 2000.

[SBM96] A. Singhal, C. Buckley, M. Mitra. Pivoted Dcoument Length Normalization *Proceedings of SIGIR '96*, August 1996, pp. 21-29.

[SMB97] A. Singhal, M. Mitra, C. Buckley. Learning Routing Queries in a Query Zone. *Proceedings of SIGIR '97*, August 1997, pp. 25-32.