FEATURES: Real-time Adaptive Feature Learning and Document Learning for Web Search

Zhixiang Chen[*§] **Xiannong Meng**[*] **Richard H. Fowler**[*] **Binhai Zhu**[†]
[*]Department of Computer Science, University of Texas – Pan American
1201 West University Drive, Edinburg, TX 78539–2999, USA
chen@cs.panam.edu, meng@cs.panam.edu, fowler@panam.edu
[†] Department of Computer Science, City University of Hong Kong
Kowloon, Hong Kong. bhz@cs.cityu.edu.hk

[§]**Contact Information**: Department of Computer Science
University of Texas–Pan American, 1201 West University Drive
Edinburg, TX 78539–2999, USA. Email: chen@cs.panam.edu
Phone: (956)381-2667. Fax: (956)384-5099

Abstract

In this paper we report our research on building FEATURES - an intelligent web search engine that is able to perform real-time adaptive feature (i.e., keyword) and document learning. Not only does FEATURES learn from the user's document relevance feedback, but also automatically extracts and suggests indexing keywords relevant to a search query and learns from the user's keyword relevance feedback so that it is able to speed up its search process and to enhance its search performance. We design two efficient and mutual-benefiting learning algorithms that work concurrently, one for feature learning and the other for document learning. FEATURES employs these algorithms together with an internal index database and a real-time meta-searcher so to perform adaptive real-time learning to find desired documents with as little relevance feedback from the user as possible. The architecture and performance of FEATURES are also discussed.

1 Introduction

As the world wide web rapidly evolves and grows, web search has come to provide an interface between the human users and the vast information of the web in people's daily life. There have been a number of popular and successful general-purpose or meta search engines such as AltaVista[1], Yahoo! [2], Google [3], MetaCrawler [4], Dogpile [5], and Inference Find [6]. Many of the existing engines support personalization (or customization) with the help of predefined user profiles or a collection of customizable parameters such as suggestions about keywords to include or exclude, language choices, document locations, etc. These functions can help a search engine find more relevant documents for the user. User profiles are often automatically created by means of *cookies*, client-side *digital traces* or tracking of user's browsing patterns (e.g., [5, 27, 21], or manually created by users themselves. Profiles can be used either at the server side or client side. In most

cases the collection of customizable parameters is fixed. In other cases some of those parameters can be automatically generated by tracking the recent browsing processes of a user. For example, a search engine can compile a list of suggested keywords based on its internal ranking and the user's most recent browsing contents for use in future search. In essence, the nature of the personalization (or customization) are *static* in the sense that it is defined before a search process and is not able to support real-time adaptive learning from the user's relevance feedback through interactive refinements.

One approach for the next generation intelligent search engines is that they be built on top of existing search engine design and implementation techniques. They may be built by integrating intelligent components with one general-purpose search engine or with a collection of general-purpose search engines through meta-searching. An intelligent search engine would use the search results of the general-purpose search engines as its starting search space, from which it would adaptively learn from the user's feedback to boost and to enhance the search performance and the relevance accuracy. It may use feature extraction, document clustering and filtering, and other methods to help an adaptive learning process. Recent research on web communities [17, 13, 7] has used a short list of hits returned by a search engine as a starting set for further expansion of search. There have been considerable efforts applying machine learning to web search related applications, for example, scientific article locating and user profiling [4, 5, 18], focused crawling [23], collaborative filtering [22, 3], and user preference boosting [12]. An adaptive real-time search algorithm without an index, which is basically a focused search starting at some given url and crawling within some neighboring documents, is given in [15],

FEATURES is part of our research on building an intelligent search engine [10, 8]. In this paper, document features are limited to keywords that are used to index documents, but our approach may be applied to other cases of document features. Given a search engine S, we use two new concepts, dynamic features and dynamic vector space, to explore the search result $R(q, u)$ returned by S for any query q and any user u. Our strategy is that we use the dynamic features that are relevant to the query q to map the whole document search space to a substantially smaller subspace - the dynamic vector space - that is relevant to the query. We present a feature learning algorithm that extracts a small set of the dynamic features, i.e., the most relevant indexing keywords to the search query at the moment, and suggests those keywords to the user for her to judge whether they are indeed relevant or not. The feature learning algorithm works concurrently with the document learning algorithm operating on relevance judgments to retrieve relevant documents for the user. Both learning algorithms help each other to speed up the search process and enhance search performance. An internal index database is used in which each document is indexed using about 300 keywords. We also design and implement a meta-searcher for FEATURES through real-time meta-searching, parsing, and indexing. FEATURES uses an internal index database, a real-time meta-searcher, and learning algorithms to perform real-time adaptive learning for web search and retrieve relevant documents requiring the least possible feedback.

2 Dynamic Features vs. Dynamic Vector Space

In spite of the World Wide Web's size and the high dimensionality of web document indexing features, the traditional vector space model in information retrieval [26, 24, 2] has been used for web document representation and search. However, to implement real-time adaptive learning with limited computing resource, here, an Ultra one Sun workstation, we cannot apply the traditional vector space model directly. Recall that back in 1998, the AltaVista system was running on 20 multi-processor machines, all of them having more than 130 Giga-Bytes of RAM and over 500 Giga-Bytes of disk space [2]. We need a new model

that is efficient enough both in time and space for FEATURES and other web search implementations with limited computing resources. The new model may also be used to enhance the computing performance of a web search system even if enough computing resources are available.

We now examine indexing in web search. Again, in this paper we use keywords as document indexing features. Let X denote the set of all indexing keywords for the whole web (or, practically, a portion of the whole web). Given any web document d, let $I(d)$ denote the set of all indexing keywords in X that are used to index d with non-zero values. Then, we have the following two properties:

(a) The size of $I(d)$ is substantially smaller than the size of X. Practically, $I(d)$ can be bounded by a constant. The rationale behind this is that in the simplest case we do not need to use all the keywords in d to index it. In our implementation of FEATURES, we use about 300 keywords to index documents in its internal database and at most 64 automatically generated keywords to index a document retrieved through meta-search.

(b) For any search process related to the search query q, let $D(q)$ denote the collection of all the documents that match q, then the set of indexing keywords relevant to q, denoted by $F(q)$, is

$$F(q) = \bigcup_{d \in D(q)} I(d).$$

Although the size of $F(q)$ varies from different queries, it is still substantially smaller than the size of X, and might be bounded by a few hundreds or a few thousands in practice.

Definition 2.1. *Given any search query q, we define $F(q)$, which is given in (b) above, as the set of dynamic features relevant to the search query q.*

Definition 2.2. *Given any search query q, the dynamic vector space $V(q)$ relevant to q is defined as the vector space that is constructed with all the documents in $D(q)$ (as given in (b) above) such that each of those document is indexed by the dynamic features in $F(q)$.*

For any query q FEATURES first finds the set of documents $D(q)$ that match the query q. It finds $D(q)$ with the help of a general-purpose search strategy through searching its internal database, or through meta-searching AltaVista [1] when no matches are found within its internal database. It then finds the set of dynamic features $F(q)$, and later constructs the dynamic vector space $V(q)$. Once $D(q), F(q)$ and $V(q)$ have been found, FEATURES starts its adaptive learning process from the user's document and feature relevance feedback. More precisely, it uses two learning algorithms in parallel to achieve its goal of learning: One algorithm adaptively learns from the documents judged by the user as relevant or irrelevant examples; the other extracts and suggests a small set of keywords that are most relevant to the search query up to the moment, and learns from the keywords judged by the user as relevant or irrelevant. The feature learning algorithm helps the document learning algorithm to increase the ranks of relevant documents, while the document learning algorithm helps the feature learning algorithm to increase the ranks of the relevant features.

3 Feature Learning and Document Learning

As we have investigated in [9, 10, 8], intelligent web search can be modeled approximately as an adaptive learning process such as on-line learning [1, 20], when the search engine acts as a learner and the user as a

teacher. The user sends a query to the engine, the engine uses the query to search the index database, and returns a list of document url's that are ranked according to a ranking function. Then, the user provides relevance feedback, and the engine uses the feedback to improve its next search and returns a refined list of document url's. The learning (or search) process ends when the engine finds the desired documents for the user. Conceptually, a query entered by the user can be understood as the logical expression of the collection of the documents the user wants. A list of document url's returned by the engine can be interpreted as an approximation to the collection of the desired documents.

Rocchio's similarity-based relevance feedback algorithm, one of the most popular query reformation method in information retrieval [16, 14, 24, 2], is in essence adaptive supervised learning from examples [25, 19]. we showed in [11] that for any of the four typical similarity measurements (inner product, cosine coefficient, dice coefficient, and Jaccard coefficient) listed in [24], Rocchio's similarity-based relevance feedback algorithm has a lower bound that is at least linear in the dimensionality of the Boolean vector space. Our linear lower bounds hold for arbitrary zero-one initial query vectors, and for arbitrary classification threshold and updating coefficients used at each step of the algorithm. Because the linear lower bounds were proved based on the worst case analysis, they *may not* affect the effective applicability of the similarity-based relevance feedback algorithm. On the other hand, the lower bounds help us understand the algorithm so that we may find new strategies to improve its performance or design new learning algorithms with better performance.

3.1 The General Setting of Learning

For each particular search query q, with the help of certain general-purpose search strategy FEATURES first finds the three sets, the general matching document set $D(q)$, the dynamic feature set $F(q)$, and the dynamic vector space $V(q)$. Let $F(q) = \{K_1, \ldots, K_n\}$ such that each K_i denotes a dynamic feature (i.e., an indexing keyword). The two learning algorithms of FEATURES maintain a common weight vector $\mathbf{w} = (w_1, \ldots, w_n)$ for dynamic features in $F(q)$. The components of $\mathbf{w}$ have non-negative real values. The feature learning algorithm uses $\mathbf{w}$ to extract and learn the most relevant features. The document learning algorithm also uses $\mathbf{w}$ to classify documents in $D(q)$ as relevant or irrelevant.

One should note that during the learning process both learning algorithms update the common weight vector $\mathbf{w}$ concurrently. Of course, both algorithms need to be equipped with efficient ranking functions. Moreover, we also need to provide a good strategy to simulate the equivalence query for the document learning algorithm, because the user in reality cannot serve as a *real* teacher as modeled in on-line learning.

3.2 The Feature Learning Algorithm FEX (Feature EXtraction)

For any dynamic feature $K_i \in F(q)$ with $1 \leq i \leq n$, we define the rank of K_i as

$$h(K_i) \quad = \quad h_0(K_i) + w_i.$$

$h_0(K_i)$ is the initial rank for K_i. Recall that K_i is some indexing keyword. With the feature ranking function h and common weight vector $\mathbf{w}$, FEX extracts and learns the most relevant features as follows.

Algorithm FEX. *At stage $s \geq 0$, it first sorts all the dynamic features in $F(q)$ with the ranking function h and extracts 10 top-ranked features and suggests them to the user for her to judge their relevance to the query q. When it receives the feature relevance feedback from the user, then for each feature K_i judged by*

the user as relevant it promote K_i by setting $w_i = pw_i$; for each feature judged by the user as irrelevant it demotes it by setting $w_i = w_i/d$. Here, both p and d are respectively feature promotion and demotion parameters, and they are tunable.

3.3 The Document Learning Algorithm TW2

The algorithm TW2, a tailored version of Winnow2 [20], was designed and successfully implemented in our recent projects WebSail and Yarrow [10, 8]. Winnow2 sets all initial weights to 1, but TW2 sets all initial weights to 0 and has a different promotion strategy accordingly. The rationale behind setting all the initial weights to 0 is not as simple as it looks. The motivation is to focus attention on the propagation of the influence of the relevant documents, and use irrelevant documents to adjust the focused search space. Moreover, this approach is computationally feasible because existing effective document ranking mechanisms can be coupled with the learning process.

Algorithm TW2 (The tailored Winnow2). *TW2 uses the common weight vector* $\mathbf{w}$ *and a real-valued threshold* θ *to classify documents in* $D(q)$. *Initially, all weights have value* 0. *Let* $\alpha > 1$ *be the promotion and demotion factor. TW2 classifies documents whose vectors* $x = (x_1, \ldots, x_n)$ *satisfy* $\sum_{i=1}^{n} w_i x_i > \theta$ *as relevant, and all others as irrelevant. If the user provides a document that contradicts to the classification of TW2, then we say that TW2 makes a mistake. Let* $w_{i,b}$ *and* $w_{i,a}$ *denote the weight* w_i *before the current update and after, respectively. When the user responds with a document which contradicts to the current classification, TW2 updates the weights in the following two ways:*

- **Promotion**: *For a document judged by the user as relevant with vector* $x = (x_1, \ldots, x_n)$, *for* $i = 1, \ldots, n$, *set*

$$w_{i,a} = \begin{cases} w_{i,b}, & if\ x_i = 0, \\ \alpha, & if\ x_i = 1\ and\ w_{i,b} = 0, \\ \alpha w_{i,b}, & if\ x_i = 1\ and\ w_{i,b} \neq 0. \end{cases}$$

- **Demotion**: *For a document judged by the user as irrelevant with vector* $x = (x_1, \ldots, x_n)$, *for* $i = 1, \ldots, n$, *set* $w_{i,a} = \frac{w_{i,b}}{\alpha}$.

In contrast to the linear lower bounds proved for Rocchio's similarity-based relevance feedback algorithm [11], the above learning algorithm has surprisingly small mistake bounds for learning any collection of documents represented by a disjunction of a small number of relevant features. The mistake bounds are independent of the dimensionality of the indexing features. For example,

- *To learn a collection of documents represented by a disjunction of at most* k *relevant features (or indexing keywords) over the* n*-dimensional boolean vector space, TW2 makes at most* $\frac{\alpha^2 A}{(\alpha-1)\theta} + (\alpha + 1)k \ln_\alpha \theta - \alpha$ *mistakes, where* A *is the number of dynamic features occurred in the learning process.*

- *When in average* l *out of* k *relevant features (or indexing keywords) appear as dynamic features for any relevant document judged by the user during the learning process, the bound in Theorem 4.3.1 is improved to* $\frac{\alpha^2 A}{(\alpha-1)\theta} + \frac{(\alpha+1)k}{l} \ln_\alpha \theta - \alpha$ *in average, where* A *is the number of dynamic features occurred in the learning process.*

Theoretical analysis of the above mistake bounds is left to the full version of this paper. The actual implementation of the learning algorithm TW2 requires the help of document ranking and equivalence query simulation given in the following two subsections.

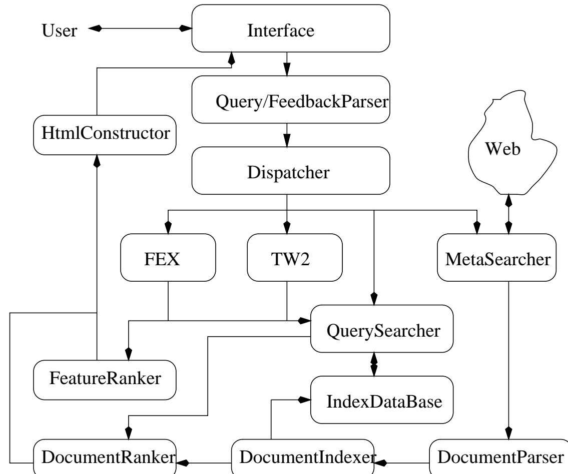

Figure 1: Architecture of FEATURES

3.4 Document Ranking

Let g be a ranking function independent of TW2 and FEX. We define the ranking function f for TW2 as follows. For any web document $d \in D(q)$ with vector $x_d = (x_1, \ldots, x_n) \in V(q)$,

$$f(d) = \gamma_d[g(d) + \beta_d] + \sum_{i=1}^{n} w_i x_i.$$

g remains constant for each document d during the learning process of TW2. Various strategies can be used to define g, for example, PageRank [6], classical tf-idf scheme, vector spread, or cited-based rankings [28]. The two additional tuning parameters are used to do individual document promotions or demotions of the documents that have been judged by the user as feedback. Initially, let $\beta_d \geq 0$ and $\gamma_d = 1$. γ_d and β_d can be updated in the similar fashion as w_i is updated by TW2.

3.5 Equivalence Query Simulation

The DocumentRanker of FEATURES uses the ranking function f to rank the documents and the HtmlConstructor returns the top 10 ranked documents to the user. These top 10 ranked documents represent an approximation to the classification made by TW2. The quantity 10 can be replaced by, say, say, 25 or 50. But it should not be too large for two reasons: The user may only be interested in a very small number of top ranked documents; and the display space is limited for visualization. The user can examine the short list of documents and can end the search process, or if some documents are judged misclassified, document relevance feedback can be provided. Sometimes, in addition to the top 10 ranked documents the system may also provide the user a short list of other documents below the top 10. Documents in the second short list may be selected randomly. The motivation for the second list is to give the user some better view of the classification made by the learning algorithm.

4 The FEATURES

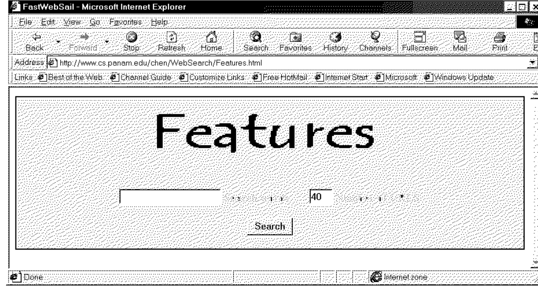

Figure 2: Interface of FEATURES

4.1 The Architecture

FEATURES is implemented on an Ultra one Sun Workstation with a storage of 27 Giga-bytes hard disk on an IBM R6000 workstation. It is a multi-threaded program coded in C++. Its architecture is shown in Figure 1. FEATURES maintains an internal index database with about 834,000 documents, each of which is indexed with about 300 indexing keywords. Besides its internal index database, it has a MetaSearcher that queries AltaVisa when needed. The documents retrieved through meta-search are parsed and indexed by the DocumentParser and the DocumentIndexer that work in real-time. The two learning algorithms FEX and TW2 update the common weight vector $\mathbf{w}$ concurrently. The major components and their functions are explained in the next subsection.

4.2 How FEATURES Works

FEATURES has an interface as shown in Figure 2. Using this interface, the user can enter a query and specify the number of document urls to be returned. Having entered query information, she then starts FEATURES. FEATURES invokes its Query/FeedbackParser to parse the query information, document relevance feedback, or feature relevance feedback. Then, Dispatcher decides whether the current task is an initial search process or a learning process. If it is an initial search process, Dispatcher first calls QuerySearcher to find the relevant documents within its internal index database. The relevant documents found by QuerySearcher are then passed to DocumentRanker, which ranks the documents and sends them to HtmlConstructor. HtmlConstructor finally generates html content to be shown to the user.

If QuerySearcher fails to find any documents relevant to the query within IndexDataBase, FEATURES calls its MetaSearcher to query AltaVista and retrieve a list of documents. The length of the list is determined by the user. Once the list of the top matched documents is retrieved, FEATURES calls its DocumentParser and DocumentIndexer to parse retrieved documents, collate them, and index them with at most 64 indexing keywords. The indexing keywords are automatically extracted from the retrieved documents by DocumentParser. The indexed documents will be cached in IndexDataBase and also sent to DocumentRanker and later to HtmlConstructor to be displayed to the user.

Usually, HtmlConstructor shows the top 10 ranked documents, plus the top 10 ranked features, to the user for her to judge the document relevance and the feature relevance. But, for the initial search HtmlConstructor shows only the top ranked documents. The format of presenting the top 10 ranked documents together with the top 10 ranked features is shown in Figure 3. In this format, each document url and each feature are preceded by radio buttons for the user to indicate whether the document or the

feature is relevant[1]. The clickable urls may be selected to view the documents so that the user can make her judgment more accurately. After provides relevance feedback, she can submit the feedback to FEATURES, view all the document urls, or enter a new query to start a new search process.

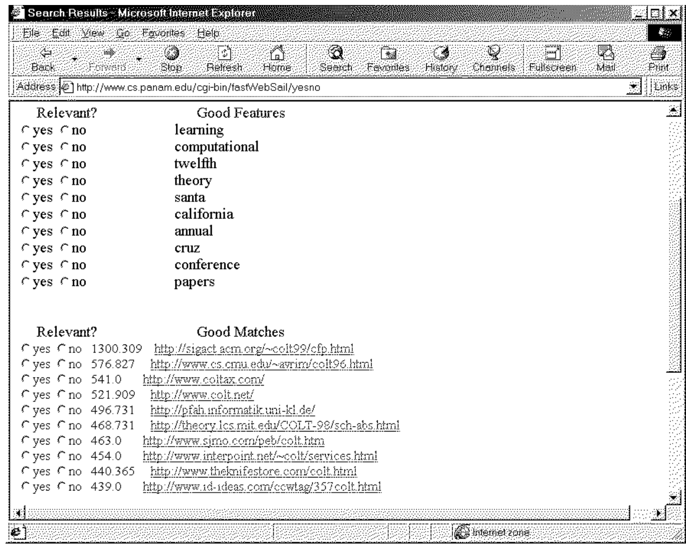

Figure 3: Initial Query Result for "colt"

If the current task is a learning process from the user's document and feature relevance feedback, Dispatcher sends the feature relevance feedback information to the feature learner FEX and the document relevance feedback information to the document learner TW2. FEX uses the relevant and irrelevant features as judged by the user to promote and demote the related feature weights in the common weight vector **w**. TW2 uses the relevant and irrelevant documents judged by the user as positive and negative examples to promote and demote the weight vector. TW2 also performs individual document promotion or demotion for those judged documents. Once FEX and TW2 have finished promotions and demotions, the updated weight vector **w** is sent to QuerySearcher and to FeatureRanker. FeatureRanker re-ranks all the dynamic features that are then sent to HtmlConstructor. QuerySearcher searches IndexDataBase to find the matched documents that are then sent to DocumentRanker. DocumentRanker re-ranks the matched documents and then sends them to HtmlConstructor to select documents and features to be displayed.

4.3 The Performance: FEATURES vs. AltaVista

We have made FEATURES open for public access. Interested readers can access it via the url given at the end of the paper and check its performance. Although FEATURES is still in its early stages, its actual performance is promising. In order to provide some measure of system performance we have made a comparison between AltaVista [1] and FEATURES. Our evaluation is based on the following approach similar to Recall and Precision.

For a search query q, let A denote the set of documents returned by the search engine (either AltaVista or FEATURES), and let R denote the set of documents in A that are relevant to q. For any integer m with

[1]The search process shown in Figures 3 and 4 was performed on March 7, 2000. The query word is *"colt"* and the desired web documents are those related to "computational learning theory". After two interactions with a total of 4 relevant and irrelevant documents and 5 relevant and irrelevant features judged by the user as feedback, all the "colt" related web documents among the initial 100 matched documents were moved to the top 10 positions.

$1 \leq m \leq |A|$, define R_m to be the set of documents in R that are among the top m ranked documents according to the search engine. Now, we define the relative Recall R_{recall} and the relative Precision $R_{precision}$ as follows.

$$R_{recall} = \frac{|R_m|}{|R|}$$
$$R_{precision} = \frac{|R_m|}{m}$$

We have conducted experiments for 100 search queries, each of which was sent to both AltaVista and FEATURES. The results returned from the two engines were examined manually to calculate R_{recall} and $R_{precision}$. We summarize the the average relative Recall R_{recall} and the average relative Precision $R_{precision}$ in the following tables. One may notice that FEATURES has better performance.

$\mathbf{R_{precision}}$	(50,10)	(50,20)	(100,10)	(100,20)	(150,10)	(150,20)	(200,10)	(200,20)
AltaVista	0.36	0.30	0.36	0.30	0.36	0.30	0.36	0.30
FEATURES	0.48	0.40	0.51	0.44	0.52	0.46	0.52	0.46

$\mathbf{R_{recall}}$	(50,10)	(50,20)	(100,10)	(100,20)	(150,10)	(150,20)	(200,10)	(200,20)
AltaVista	0.37	0.51	0.30	0.42	0.29	0.39	0.29	0.39
FEATURES	0.67	0.96	0.42	0.80	0.37	0.71	0.37	0.67

In the above table, each column is labeled by a pair of integers $(|A|, m)$, where A is the total number of documents retrieved by the engine for the given search query and m is used to define the relative Recall and Precision. Because of the non-adaptive nature of AltaVista, it is obvious that AltaVista has the same relative Precision for each query when the value of m is the same. The average relative Recall and Precision values are calculated for FEATURES after an average of 5 interactive refinements.

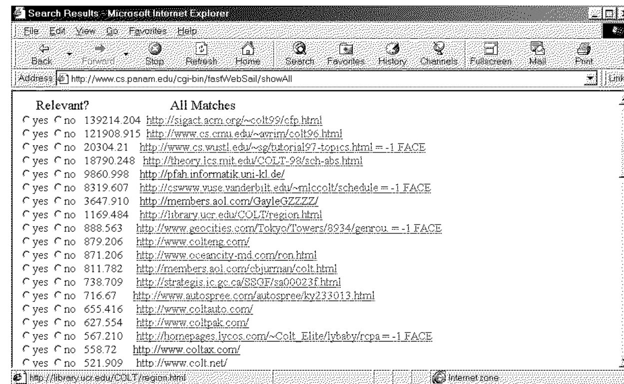

Figure 4: Result for "colt" after 2 Interactions, 4 Examples and 5 Features Judged

5 Concluding Remarks

As part of our efforts towards building an intelligent search engine, we have designed and implemented FEATURES. It utilizes two adaptive learning algorithms that work concurrently in real-time, one of which extracts and learns the most relevant dynamic features from the user's feature relevance judgments, and the

other learns the most relevant documents from the user's document relevance judgments. FEATURES is still in its early stages and needs to be improved and enhanced in many aspects. For example, in the near future we plan to improve its learning algorithms with some real-time clustering methods.

URL References:

[1] AltaVista: www.altavista.com.
[2] Yahoo!: www.yahoo.com.
[3] Google: www.google.com.
[4] MetaCrawler: www.metacrawler.com.
[5] Inference Find: www.infind.com.
[6] Dogpile: www.dogpile.com.
[7] WebSail: www.cs.panam.edu/chen/WebSearch/WebSail.html.
[8] Yarrow: www.cs.panam.edu/chen/WebSearch/Yarrow.html.
[9] Features: www.cs.panam.edu/chen/WebSearch/Features.html.

References

[1] D. Angluin. Queries and concept learning. *Machine Learning*, 2:319–432, 1987.

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

[3] D. Billsus and M. J. Pazzani. Learning collaborative information filters. In *Proceedings of the Fifteenth International Conference on Machine Learning*, 1998.

[4] K. Bollacker, S. Lawrence, and C. Lee Giles. Citeseer: An autonomous web agent for automatic retrieval and identification of interesting publications. In *Proceedings of the Second International Conference on Autonomous Agents*, pages 116–113, New York, 1998. ACM Press.

[5] K. Bollacker, S. Lawrence, and C. Lee Giles. A system for automatic personalized tracking of scientific literature on the web. In *Proceedings of the Fourth ACM Conference on Digital Libraries*, pages 105–113, New York, 1999. ACM Press.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh World Wide Web Conference*, 1998.

[7] S. Chakrabarti, B. Dom, P. Raghavan, S. Rajagopalan, D. Gibson, and J. Kleinberg. Automatic resource compilation by analyzing hyperlink structure and associated text. In *Proceedings of the Seventh World Wide Web Conference*, pages 65–74, 1998.

[8] Z. Chen and X. Meng. Yarrow: A real-time client site meta search learner. accepted by the AAAI 2000 Workshop on Artificial Intelligence for Web Search, July 2000.

[9] Z. Chen, X. Meng, and R. H. Fowler. Searching the web with queries. *Knowledge and Information Systems*, 1:369–375, 1999.

[10] Z. Chen, X. Meng, B. Zhu, and R. Fowler. Websail: From on-line learning to web search. accepted by the 2000 International Conference on Web Information Systems Engineering, June 2000.

[11] Z. Chen and B. Zhu. Some formal analysis of the rocchio's similarity-based relevance feedback algorithm. Technical Report CS-00-22, Dept. of Computer Science, University of Texas-Pan American, March 2000.

[12] Y. Freund, R. Iyer, R. Schapire, and Y. Singer. An efficient boosting algorithm for combining preferences. In *Machine Learning: Proceedings of the Fifteenth International Conference*, 1998.

[13] D. Gibson, J. Kleinberg, and P. Raghavan. Inferring web communities from link topology. In *Proceedings of the Ninth ACM Conference on Hypertext and Hypermedia*, 1998.

[14] E. Ide. New experiments in relevance feedback. In G. Salton, editor, *The Smart System - Experiments in Automatic Document Processing*, pages 337–354, Englewood Cliffs, NJ, 1971. Prentice-Hall Inc.

[15] A. Ikeji and F. Fotouhi. An adaptive real-time web search engine. In *Proceedings of the ACM CIKM'99 Workshop on Web Information and Data Management*, 1999.

[16] Jr. J.J. Rocchio. Relevance feedback in information retrieval. In G. Salton, editor, *The Smart Retrieval System - Experiments in Automatic Document Processing*, pages 313–323, Englewood Cliffs, NJ, 1971. Prentice-Hall, Inc.

[17] J. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of ACM*, 46(5):604–632, 1999.

[18] S. Lawrence, K. Bollacker, and C. Lee Giles. Indexing and retrieval of scientific literature. In *Proceedings of the Eighth ACM International Conference on Information and Knowledge Management*, 1999.

[19] D. Lewis. Learning in intelligent information retrieval. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 235–239, 1991.

[20] N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2:285–318, 1988.

[21] X. Meng and Z. Chen. Personalize web search using information on client's side. In *Advances in Computer Science and Technologies*, pages 985–992. International Academic Publishers, 1999.

[22] A. Nakamura and N. Abe. Collaborative filtering using weighted majority prediction algorithms. In *Machine Learning: Proceedings of the Fifteenth International Conference*, 1998.

[23] J. Rennie and A. McCallum. Using reinforcement learning to spider the web efficiently. In *Proceedings of the Sixteenth International Conference on Machine Learning*, 1999.

[24] G. Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, 1989.

[25] G. Salton and C. Buckley. Improving retrieval performance by relevance feedback. *Journal of the American Society for Information Science*, 41(4):288–297, 1990.

[26] G. Salton, A. Wong, and C.S. Yang. A vector space model for automatic indexing. *Comm. of ACM*, 18(11):613–620, 1975.

[27] D.H. Widyantoro, T.R. Ioerger, and J. Yu. An adaptive algorithm for learning changes in user interests. In *Proceedings of the Eight ACM International Conference on Information and Knowledge Management*, pages 405–412, 1999.

[28] B. Yuwono and D.L. Lee. Search and ranking algorithms for locating resources on the world wide web. In *Proceedings of the International Conference on Data Engineering*, pages 164–171, New Orleans, USA, 1996.