



Discovering Document Semantics QBYS: A System for Querying the WWW by Semantics

MICHAEL JOHNSON

College of Science and Mathematics, Madonna University, Livonia, MI 48150, USA

mjohnson@madonna.edu

FARSHAD FOTOUHI

SORIN DRĂGHICI

MING DONG

DUO XU

Department of Computer Science, Wayne State University, Detroit, MI 48202, USA

fotouhi@cs.wayne.edu

sod@cs.wayne.edu

mdong@cs.wayne.edu

xuduo18@cs.wayne.edu

Abstract. This paper describes our research into a query-by-semantics approach to searching the World Wide Web. This research extends existing work, which had focused on a query-by-structure approach for the Web. We present a system that allows users to request documents containing not only specific content information, but also to specify that documents be of a certain type. The system captures and utilizes structure information as well as content during a distributed query of the Web. The system also allows the user the option of creating their own document types by providing the system with example documents. In addition, although the system still gives users the option of dynamically querying the web, the incorporation of a document database has improved the response time involved in the search process. Based on extensive testing and validation presented herein, it is clear that a system that incorporates structure and document semantic information into the query process can significantly improve search results over the standard keyword search.

Keywords: query-by-semantics, document features, document type, neural network

1. Introduction

The task of accessing and processing information in today's business world is a necessity. However, as more and more information is recorded, being able to sift through all this information to capture what is relevant has become increasingly more difficult. Traditional database systems have, for the most part, been able to keep up with the increased amount of data because hardware technology such as processor speed and memory size has increased as well. However, in recent years a significant paradigm shift towards relying on distributed information repositories has occurred, largely due to the phenomenal growth of the World Wide Web. Once again, database systems were able to keep abreast thanks in part to new networking technology and research into distributed database systems. However, the *new-style* distributed database systems still rely heavily on the *old-style* structure of tables, records, and fields that are inherent in traditional database systems. As a result, these systems cannot easily adapt to handling *unstructured* and *semi-structured* data.

From a *closed-world* perspective, not being able to adapt to unstructured or semi-structured data would not be an issue. In fact, it has always been the responsibility of

the data to adapt to the structure of the database system. However, for many database traditionalists who originally perceived the Web as the “goose that laid the golden egg”, it has become a realization that in actuality, the Web is much more like “Pandora’s Box”, and unfortunately, it has already been opened. In the time span of just a few short years, the number of web pages has grown exponentially. There are billions of pages on the Web (according to their web site, Google [9] alone has indexed over 3 billion web pages). Although some web sites use traditional-style databases to generate page content, most web pages are written in Hypertext Markup Language (HTML). When viewed in a web browser, these documents are essentially unstructured. However, the *source code* of an HTML document contains specific structure information relating to how the document *should* be rendered by the browser. Hence an HTML web page is actually a semi-structured document. Unfortunately, although these types of documents are *visually* appealing, to the traditional database query system, about the only useful information is the text within the page.

Early web search systems (many of which are still in use today) performed *queries* of the web by capturing and storing page information in a database. Users then search the database by providing *key* search words. The results of the search are simply a list of links to the pages that contain the specified words. However, these types of search engines are much less effective today because of the vast size of the Web. This is the case not because the query system cannot handle searching a large amount of data but rather because typical searches can result in thousands of purportedly *relevant* web documents. Hence, a significant amount of research has gone into improving web searches. However, most of the research into improving searches has involved analyzing the content of a document. Our approach, on the other hand, has been to take advantage of the fact that a web page is a semi-structured document, and that web designers utilize structure to emphasize particularly important aspects of their documents. Our prior research, documented in “A Neural Network Net Query-by-Structure Approach” [15] has shown that capturing and using structure information during the query process can significantly enhance performance.

2. Query-by-semantics system

Our research began with a query by-structure approach using a system we designed called CLaP system [7]. CLaP was designed in an attempt to enhance Web searches by allowing the user to search for web pages containing words that were structured and presented in a specific way within a web page. However, due to the extensive amount of information required of the user during the query, it became apparent that another approach was necessary. Hence, two new systems that employed neural networks to organize the information based on relevancy of both the content and the structure of the document were created [15]. The primary goal of these systems, called AN³ systems, was to test the feasibility of using neural networks to improve query performance while eliminating the need for the complex user queries. The initial results from the testing were promising enough to warrant further research. However, it was clear that significant changes were required. In particular, the system used prototype vectors that were handcrafted to represent the designer’s interpretation of a specific document type.

There are at least two major problems with this approach. First, users of the system may have entirely different opinions about how a specific type of document should be structured. Secondly, determining the components of the document types at system configuration time limits the number of document types, when in fact; there are an infinite number of different types of web documents. Clearly, a better approach, such as allowing the user to provide the details regarding a document type, was needed. But, so as not to repeat the drawbacks of the CLaP system, instead of asking the user to provide detailed structure information, a better approach is to have the system simply asks the user to provide example documents that are representative of the document types. These examples could then used to create the prototype vectors.

2.1. Overview of query-by-semantic system

A high-level overview of the query process for the query-by-semantic system can best be given by examining the architecture depicted figure 1. Prior to initiating a query, the system must be configured with sets of documents representing various document types. In the architecture, this is represented by the Document Type module, where the process for each different document type involves creating a document type name, providing a set of documents that represent examples of that document type, and then constructing a vector that represents the features of that document type. Once the document types have been defined in the systems, users initiate a query using keywords and a document type. The systems provide the option of initiating the query from an existing web search engine or querying a local database. This is reflected in the architecture by a query to the Document Repository component. However, regardless of the option selected, the documents that are returned from the Document Repository are simply a set of web documents that match the initial keyword search. These documents are then sent to the Machine Learning module along with the *document type* selected by the user. The output of the Machine Learning module is effectively the same documents input to the module from the Document Repository module but ordered or ranked based on how closely the documents match the type of document requested by the user. In fact, when a web query is performed, the output is simply the URLs and descriptions captured in the initial commercial search engine query but in the order specified by the Machine Learning module.

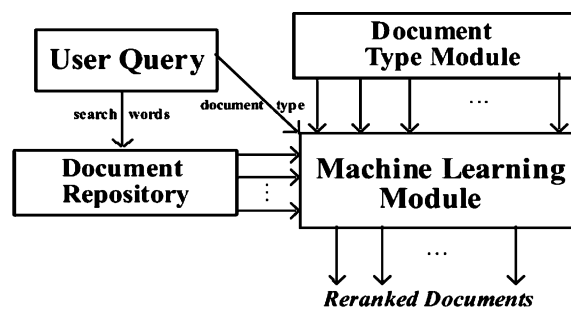


Figure 1. Query-by-semantic system architecture.

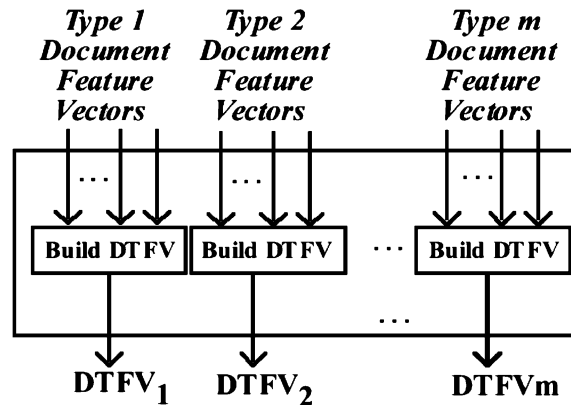


Figure 2. Document type module.

The system presented in this paper utilizes a modified Hamming neural network [10] within the Machine Learning module. A Hamming neural network is a supervised learning network that consists of two layers. The first layer (Feedforward Layer) performs a correlation between the input vector and the prototype vectors. The second layer (Recurrent Layer) performs a competition to determine which of the prototype vectors is closest to the input vector. The network is considered supervised because it utilizes prototype vectors, which are determined in advance and are designed to match the expected input vectors.

Hence, prior to being able to classify documents, the neural network system must first have initialized prototype vectors. These vectors are constructed from the *document type feature vectors* (DTFV). These vectors are obtained from the Document Type module. However, as mentioned earlier, the first step is to create a representation of the document types to be searched for. This is accomplished by providing the example documents to the Document Type module of the system.

As shown in figure 2, the output of the Document Type module is a set of document type feature vectors. Each vector is essentially a description of a document type based on a set of different, yet similar, documents. As its name suggests, this description is in the form of a vector, which is constructed using a set of *document feature vectors* (DFV). The document feature vectors, which are constructed from the example documents provided by the user, are first grouped into document types. These vectors are then input into a module that constructs the separate document type feature vector for each document type. The following sections describe the different implementations utilized in their construction of these vectors.

2.2. Document feature vector

The input to the Machine Learning module is a set of vectors. Hence, each example HTML document provided by the user must be converted into a vector. This vector makes up the *features* of the document. The concept of the document feature vector was first presented in the neural network query-by-structure systems. However, the feature vectors were based on a specific set of system-defined words contained within a specific set of tags. For example,

the feature vector for a document type considered to be a *news article* was constructed to represent a small set of relevant words (such as business, news, sports, article) and a set of HTML tags which those words should appear within (such as bold (), underline (<u>), font () and italics (<i>)).

However, for the query-by-semantic system, the document type feature vector must be dynamically constructed from the examples provided by the user. As a result, an entry in the document feature vector must allow for almost any word within almost any tag. This entry can best be described as a <tag, word, value> triple, where *tag* represents an HTML or XML tag, *word* represents a word in the document, and *value* represents the number of times that the word appeared within the document inside the specific tag. So, when provided with an example document, the system parses the document and creates a document feature vector as a collection of <tag, word, value> triples.

Like the neural network query-by-structure systems, not all tags and words are included in the document feature vector. However, instead of specifying which tags and words to include in the feature vector, the system specifies certain *stop* tags and *stop* words that are *not* to be included. In other words, all tags and words not labeled as *stop* tags or *stop* words are considered to be *valid* tags and words, respectively. In addition, since the system does not designate the *stop* tags when determining which tags to treat as *valid*, a mechanism was implemented to provide the user with the capability to dynamically add or remove tags from the list of *stop* tags between queries. Although not required, there are several tags that should be excluded since they do not provide any additional structure information (i.e. <body>, <script>, <style>, etc). However, this is left up to the discretion of the user. As a result, since the system does not have a predefined set of *acceptable* tags, any properly structured HTML and even XML tags will be captured by the system.

As far as identifying *valid* words, instead of predefining the list of *important* words, the system allows for any word to be included in the document feature vectors. The system does perform some *stemming* by removing all punctuation and numeric symbols from the words. Additionally, the system does incorporate two ways for the user to specify *invalid* words. First, the user can add words to a list of invalid *stop* words. Hence, words such as: them, that, the, you, etc. can be ignored by the parser when encountered in a document. Secondly, so as to allow the user to easily specify a much larger set of invalid words, the system incorporates a mechanism for the user to specify the size of the *valid* word. For example, the user can specify that the document feature vectors can only contain words of three characters or more. Although not as robust as utilizing a comprehensive list of *stop* words, this is clearly much less cumbersome for the user.

A document feature vector is constructed with a one-pass parser that identifies only tags and words allowed by the user-defined word size and list of invalid tags. The final result of the parsing is a collection of <tag, word, value> triples which make up a document feature vector.

2.3. Document type feature vector

The construction of the document type feature vector was an important component of the system so six different algorithms were evaluated. Each algorithm is fairly straightforward with

the fundamental objective being to determine a *value* to associate with each $\langle tag, word \rangle$ pair in the document type feature vector. This *value* plays an important role in determining the significance of each $\langle tag, word \rangle$ pair. However, it should be clear that the resultant document type feature vector from all six algorithms is simply a vector of $\langle tag, word, value \rangle$ triples that contains exactly one entry for each possible $\langle tag, word \rangle$ pair in all document feature vectors that belong to one document type, with no $\langle tag, word \rangle$ pair being repeated. Mathematically $\langle tag, word \rangle$ pairs for document type k can be described as the union of the $\langle tag, word \rangle$ pairs of all document feature vectors for type k :

$$\langle tag, word \rangle \text{ of DTFV}_k = \cup \langle tag_j, word_j \rangle \text{ of DFV}_i \quad (1)$$

where i is the index of document feature vectors for document type k and j is the index of $\langle tag, word \rangle$ pairs in document feature vector i . Once the $\langle tag, word \rangle$ pairs are fixed, the *value* component of DTFV could then be computed by different ways (see Method #1 to Method #6 below for details).

Each algorithm applies two functions, namely **val** and **modify**. The **val** function, when given a *tag* and *word* as input, simply returns the *value* associated with the corresponding $\langle tag, word, value \rangle$ triple in the document type feature vector. Likewise, the **modify** function, when given a *tag* and *word* as input, modifies the value associated with the corresponding $\langle tag, word, value \rangle$ triple in the document type feature vector based on a *value* provide as the third input. In addition, the **modify** function also returns the document type feature vector as output from the function.

Method #1: Incremental Sum Algorithm

Each document type feature vector (DTFV) is constructed by taking each document feature vector (DFV) of the same type i and counting the number of common $\langle tag, word \rangle$ pairs with this result being placed in the document type feature vector *value* of the $\langle tag, word, value \rangle$ triple having the same $\langle tag, word \rangle$ pair as follows:

$$\forall_i \text{ DFV } \forall_j \langle tag, word, value \rangle, \\ \text{DTFV} = \begin{cases} \text{DTFV} \cup \langle tag_j, word_j, 1 \rangle & \langle tag_j, word_j \rangle_i \notin \text{DTFV} \\ \text{modify}(tag_j, word_j, \text{val}(tag_j, word_j) + 1) & \langle tag_j, word_j \rangle_i \in \text{DTFV} \end{cases}$$

Note that in the Incremental Sum algorithm, the *value* (the number of occurrences of a word) in the document feature vector $\langle tag, word, value \rangle$ triple is not considered when creating the document type feature vector. Hence, when a new $\langle tag, word \rangle$ pair is encountered, a new $\langle tag, word, value \rangle$ triple is created in the document type feature vector with a *value* of 1 regardless of the *value* in the document feature vector $\langle tag, word, value \rangle$ triple. In addition, each subsequent occurrence of the same $\langle tag, word \rangle$ pair in a document feature vector $\langle tag, word, value \rangle$ triple results in the *value* of the corresponding $\langle tag, word, value \rangle$ triple of the document type feature vector being incremented.

Method #2: Value Sum Algorithm

Each document type feature vector (DTFV) is constructed by taking each document feature vector (DFV) of the same type i and adding the *values* in the $\langle tag, word, value \rangle$ triples whose document feature vectors have common $\langle tag, word \rangle$ pairs with this result being placed in the document type feature vector *value* of the $\langle tag, word, value \rangle$ triple as follows:

$$\forall_i \text{DFV} \forall_j \langle tag, word, value \rangle,$$

$$\text{DTFV} = \begin{cases} \text{DTFV} \cup \langle tag_j, word_j, value_j \rangle & \langle tag_j, word_j \rangle_i \notin \text{DTFV} \\ \text{modify}(tag_j, word_j, \text{val}(tag_j, & \langle tag_j, word_j \rangle_i \in \text{DTFV} \\ word_j) + value_j) & \end{cases}$$

As opposed to the Incremental Sum algorithm, in the Value Sum algorithm, the *value* in the document feature vector $\langle tag, word, value \rangle$ triple is considered when creating the document type feature vector. Hence, when a new $\langle tag, word \rangle$ pair is encountered, a new $\langle tag, word, value \rangle$ triple is created in the document type feature vector with the corresponding *value* being identical to the *value* of $\langle tag, word, value \rangle$ triple in document feature vector. Likewise, each subsequent occurrence of the same $\langle tag, word \rangle$ pair in a document feature vector results in the *value* of the $\langle tag, word, value \rangle$ triple of the document type feature vector being increased by the *value* of the $\langle tag, word, value \rangle$ triple in document feature vector.

Method #3: Saturated Incremental Sum Algorithm

Each document type feature vector (DTFV) is constructed by taking each document feature vector (DFV) of the same type i and counting the number of common $\langle tag, word \rangle$ pairs with this result being placed in the document type feature vector *value* of the $\langle tag, word, value \rangle$ triple just as in the Incremental Sum algorithm. However, a *upper bound* limit b_1 is place on the size of the *value* in the respective $\langle tag, word, value \rangle$ triple of the document type feature vector as follows:

$$\forall_i \text{DFV} \forall_j \langle tag, word, value \rangle,$$

$$\text{DTFV} = \begin{cases} \text{DTFV} \cup \langle tag_j, word_j, 1 \rangle & \langle tag_j, word_j \rangle_i \notin \text{DTFV} \\ \text{modify}(tag_j, word_j, \text{val}(tag_j, & \langle tag_j, word_j \rangle_i \in \text{DTFV} \\ word_j) + 1) & \wedge \text{val}(tag_j, word_j) \geq b_2 \end{cases}$$

Note that if $b_1 = i$ (the number of document feature vectors), the functionality of the Saturated Incremental Sum algorithm is identical to that of the Incremental Sum algorithm. Since neither considers the *value* of the $\langle tag, word, value \rangle$ triple in the document feature vector when creating the document type feature vector. Hence, setting the value of b_1 to anything greater i would be pointless.

Method #4: Saturated Value Sum Algorithm

Each document type feature vector (DTFV) is constructed by taking each document feature vector (DFV) of the same type i and adding the *values* in the $\langle \text{tag}, \text{word}, \text{value} \rangle$ triples whose document feature vectors have common $\langle \text{tag}, \text{word} \rangle$ pairs just as in the Value Sum algorithm. However, like the Saturated Incremental Sum algorithm, an *upper bound* limit b_2 is placed on the size of *value* in the respective $\langle \text{tag}, \text{word}, \text{value} \rangle$ triple in the document type feature vector as follows:

$$\forall_i \text{DFV} \forall_j \langle \text{tag}, \text{word}, \text{value} \rangle,$$

$$\text{DTFV} = \begin{cases} \text{DTFV} \cup \langle \text{tag}_j, \text{word}_j, b_2 \rangle & \langle \text{tag}_j, \text{word}_j \rangle_i \notin \text{DTFV} \\ & \wedge \text{val}(\text{tag}_j, \text{word}_j) \geq b_2 \\ \text{DTFV} \cup \langle \text{tag}_j, \text{word}_j, \text{value}_j \rangle & \langle \text{tag}_j, \text{word}_j \rangle_i \notin \text{DTFV} \\ & \wedge \text{val}(\text{tag}_j, \text{word}_j) < b_2 \\ \text{modify}(\text{tag}_j, \text{word}_j, b_2) & \langle \text{tag}_j, \text{word}_j \rangle_i \in \text{DTFV} \\ & \wedge \text{value}_j + \text{val}(\text{tag}_j, \text{word}_j) \geq b_2 \\ \text{modify}(\text{tag}_j, \text{word}_j, \\ \text{val}(\text{tag}_j, \text{word}_j) + \text{value}_j) & \langle \text{tag}_j, \text{word}_j \rangle_i \in \text{DTFV} \\ & \wedge \text{value}_j + \text{val}(\text{tag}_j, \text{word}_j) < b_2 \end{cases}$$

As opposed to the Saturated Incremental Sum algorithm, there is no limit to the size selected for b_2 . In fact, selecting too small a value for b_2 could result in the *value* of one $\langle \text{tag}, \text{word}, \text{value} \rangle$ triple from a document feature vector entirely saturating the *value* in the corresponding $\langle \text{tag}, \text{word}, \text{value} \rangle$ triple in the document feature type vector.

Method #5: Modified Saturated Value Sum Algorithm

Each document type feature vector (DTFV) is constructed by taking each document feature vector (DFV) of the same type i and adding the *values* in the $\langle \text{tag}, \text{word}, \text{value} \rangle$ triples whose document feature vectors have common $\langle \text{tag}, \text{word} \rangle$ pairs just as in the Value Sum algorithm. However, as opposed to the previous Saturated Value Sum algorithm, an *upper bound* limit b_3 is used to limit the size of the *value* of a $\langle \text{tag}, \text{word}, \text{value} \rangle$ triple in the document feature vector as follows:

$$\forall_i \text{DFV} \forall_j \langle \text{tag}, \text{word}, \text{value} \rangle,$$

$$\text{DTFV} = \begin{cases} \text{DTFV} \cup \langle \text{tag}_j, \text{word}_j, b_3 \rangle & \langle \text{tag}_j, \text{word}_j \rangle_i \notin \text{DTFV} \\ & \wedge \text{val}(\text{tag}_j, \text{word}_j) \geq b_3 \\ \text{DTFV} \cup \langle \text{tag}_j, \text{word}_j, \text{value}_j \rangle & \langle \text{tag}_j, \text{word}_j \rangle_i \notin \text{DTFV} \\ & \wedge \text{val}(\text{tag}_j, \text{word}_j) < b_3 \\ \text{modify}(\text{tag}_j, \text{word}_j, \text{val}(\text{tag}_j, \\ \text{word}_j) + b_3) & \langle \text{tag}_j, \text{word}_j \rangle_i \in \text{DTFV} \\ & \wedge \text{value}_j \geq b_3 \\ \text{modify}(\text{tag}_j, \text{word}_j, \text{val}(\text{tag}_j, \\ \text{word}_j) + \text{value}_j) & \langle \text{tag}_j, \text{word}_j \rangle_i \in \text{DTFV} \\ & \wedge \text{value}_j < b_3 \end{cases}$$

Note that the difference from the previous Saturated Value Sum algorithm is that the upper bound limit b_3 is placed on the *value* in the $\langle tag, word, value \rangle$ triple of each document type feature vector, rather than the *value* in the corresponding $\langle tag, word, value \rangle$ triple of the document type feature vector. Additionally, if $b_1 = 1$, the functionality of the Modified Value Sum algorithm is identical to that of the Incremental Sum algorithm since regardless of the *value* of the $\langle tag, word, value \rangle$ triple in the document feature vector, only a 1 is added to the corresponding *value* in the $\langle tag, word, value \rangle$ triple of the document type feature vector.

Method #6: Modified Saturated Base Value Sum Algorithm

Each document type feature vector (DTFV) is constructed by taking each document feature vector (DFV) of the same type i and adding the *values* in the $\langle tag, word, value \rangle$ triples whose document feature vectors have common $\langle tag, word \rangle$ pairs just as in the Modified Saturated Value Sum algorithm. However, in addition to the *upper bound* limit b_3 , a *base value* b_4 is also applied to the size of the *value* of a $\langle tag, word, value \rangle$ triple in the document feature vector as follows:

$$\forall_i \text{DFV} \forall_j \langle tag, word, value \rangle,$$

$$\text{DTFV} = \begin{cases} \text{DTFV} \cup \langle tag_j, word_j, b_3 + b_4 \rangle & \langle tag_j, word_j \rangle_i \notin \text{DTFV} \wedge \text{val}(tag_j, word_j) \geq b_3 + b_4 \\ \text{DTFV} \cup \langle tag_j, word_j, value_j + b_4 \rangle & \langle tag_j, word_j \rangle_i \notin \text{DTFV} \wedge \text{val}(tag_j, word_j) < b_3 + b_4 \\ \text{modify}(tag_j, word_j, \text{val}(tag_j, word_j) + b_3 + b_4) & \langle tag_j, word_j \rangle_i \in \text{DTFV} \wedge \text{value}_j \geq b_3 + b_4 \\ \text{modify}(tag_j, word_j, \text{val}(tag_j, word_j) + \text{value}_j + b_4) & \langle tag_j, word_j \rangle_i \in \text{DTFV} \wedge \text{value}_j < b_3 + b_4 \end{cases}$$

Note that the difference from the previous Modified Saturated Value Sum algorithm is that the inclusion of the base value b_4 imposes a *lower bound* on the *value* in the $\langle tag, word, value \rangle$ triple of each document type feature. The general idea here was to give more weight to the second occurrence of a $\langle tag, word \rangle$ pair, but not twice as much as the first occurrence of that same $\langle tag, word \rangle$ pair.

Analysis of each of the above algorithms was performed and is documented in the following chapter. As mention earlier, the objective was to determine the value to associate with each $\langle tag, word \rangle$ pair in the document type feature vector. However, in order to determine the actual document type feature vector used by the system, after each algorithm completed, the $\langle tag, word, value \rangle$ triples were sorted as only the $\langle tag, word \rangle$ pairs with the top n highest *values* of the document type feature vector are utilized by the prototype vectors.

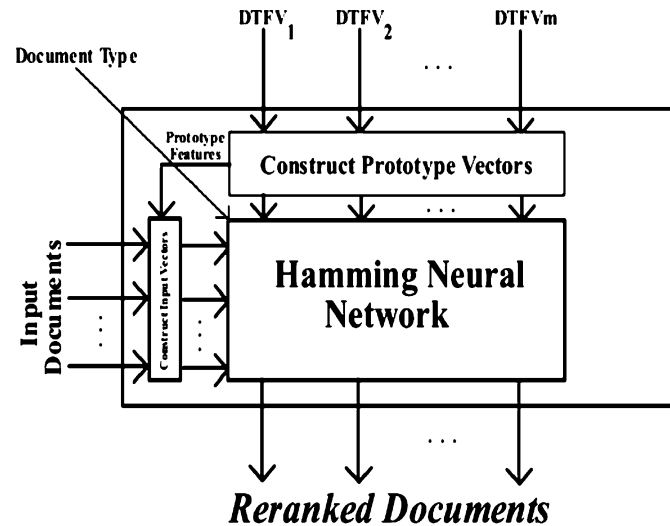


Figure 3. Machine learning module.

2.4. Machine learning module

The Machine Learning module for this system, as shown in figure 3, incorporates a modified version of the Hamming neural network that was originally used in by query-by-structure systems. The Hamming neural network is a supervised neural network because it utilizes prototype vectors that are determined prior to the processing of the input vectors. Ideally, the prototype vectors should be designed to resemble the expected input vectors. Hence, the first step in designing the Hamming neural network for the system was to initialize the prototype vectors.

The prototype vectors were created using the document type feature vectors. In the neural network, each document type has its own prototype vector. However, these vectors are not identical to the corresponding document type feature vector because the prototype vectors must contain not only entries for the document type that is being represented by that vector, but also by other document types that are not representative of the document type. Hence it is necessary to include the entries of the other document type feature vectors in each prototype vector.

One significant issue that was addressed was the number of entries per document type feature vector to include in the prototype vectors. Each document type feature vector is a sorted vector (by *value*) containing $\langle tag, word, value \rangle$ triples that includes the *tags* and *words* of a number of similar documents. However, the *value* components of the vector provide a measurement of the significance of each $\langle tag, word \rangle$ pair. Hence, constructing the prototype vector (PV) for each document type was simply a matter of combining the n most relevant entries in the document type feature vector with the top n entries in several other document feature vectors.

Hence, using the sorted document type feature vectors, a list of *words* and a list of *tags* is constructed based on the most relevant n entries. This list is comprised of the each unique $\langle tag, word \rangle$ pair in the i document type feature vectors being considered. Here, the number of unique *tags* multiplied by the number of unique *words* provides the size of a prototype vector where each entry in the prototype vector results in a unique $\langle tag, word \rangle$ pair. Since the number of document type feature vectors represents the number of prototype vectors, the last component of the construction of the prototype vectors is to simply create a prototype vector from each document type feature vector by setting an entry in the prototype vector to 1 if the equivalent $\langle tag, word \rangle$ is in one of the first n entries in the document type feature vector and setting it to 0 otherwise. One thing to note here is that regardless of whether a feature is the top feature or the n th feature, the value in corresponding value in the prototype vector is the same, namely 1. It should also be noted that the value n can be adjusted by the user.

The input vectors to the neural network were constructed from the documents identified as relevant by the initial user query and hence obtained from the Document Repository module. However, since the input vectors to the neural network must be of identical structure to that of the prototype vectors, the input needed to be modified. However, since the output of the document repository was a vector of $\langle tag, word, value \rangle$ triples, constructing the input vectors simply involved scanning this input to identify $\langle tag, word \rangle$ pairs within the input documents that corresponded to all possible $\langle tag, word \rangle$ pairs within the prototype vectors. If the $\langle tag, word \rangle$ pair appeared in an input document and prototype vector, its *value* was placed in the input vector. Any $\langle tag, word \rangle$ pairs in the prototype vectors that are not in the input document result in a 0 being inserted into the appropriate location in the input vector. And, any $\langle tag, word \rangle$ pairs in the input document that are not in the prototype vectors are merely discarded.

Once the input vectors and prototype vectors have been properly formed, they are input to the Hamming neural network. This network is virtually identical to the previous query-by-structure approach, the specific details of this network can be found in “Enhancing Query-by-Structure Neural Network Net Approach” [14]. Upon completion, the output from the neural network is a set of vectors representing which prototype vectors most closely matches each input vectors. Recall, though, that the objective of this system was to determine which of the web documents (*input vectors*) most closely matches the selected document type (*prototype vector*). Fortunately, the network output works rather well in solving this problem as our network not only returns the *score* of the winner, but also the *scores* of the losers. Hence, sorting the values in each of these vectors, from highest to lowest, provides a new ranking of the web documents. So, after sorting, the system can simply output the URLs and descriptions captured by the Document Repository but in the order specified by the neural network.

2.5. Document repository module

In order to improve query response time and to conduct worthwhile analysis, it was imperative to incorporate a document database into the system. Initially, an attempt was made to incorporate an existing database/search engine package called ht://dig [24]. However, since

the prototype vectors required that the input documents be mapped to vectors containing $\langle \text{tag}, \text{word}, \text{value} \rangle$ triples, it was still necessary to perform a retrieval of each of the web documents in order to capture document structure. As a result, at this time, a simplified document database was developed.

Currently, the document database, although functional, is somewhat small and rudimentary. The database is populated each time a user performs a dynamic web query by first sending the retrieved web pages to the Document Repository module before sending them to the neural network. In the initial implementation, instead of saving the web pages retrieved, the Document Database module saved the document feature vectors. By maintaining document feature vectors, not only is the time to dynamically retrieve the documents from the web eliminated, but the time to parse the documents as well. In addition, the amount of storage needed to maintain only the feature vectors is significantly less than would be needed to save entire documents.

However, by not maintaining local copies of the entire document, too much relevant information was lost. In particular, because the user has the capability to dynamically add or remove *stop* tags and *stop* words, any change to these system parameters could result in inconsistent future results. Hence, it was necessary for the system to capture the entire document and store in the database along with its corresponding URL. As a result, any query to the local database requires that each document be parsed prior to being sent to the Machine Learning module. Although not as significant as having to issue a query on the Web, this did result in a slower response time.

In an attempt to improve response time, a couple other modifications were made to the Document Repository module. First, when a document is first placed into the database, it is indexed by document type since the document type selected by the user is known. This allows for future local database queries to significantly prune down the search space by simply considering documents that have been indexed by the desired type. The obvious drawback is that any documents of that type that have been placed into another document types space will not be considered by that particular query. This is probably not too significant when, for example, considering two types such as resumes and recipes. Clearly, it is unlikely that many recipes will end up being returned from a Web query requesting resumes. However, if a user decides to add a new type that one would consider a subtype of an existing type (i.e. artist resumes as opposed to resumes), clearly some highly relevant documents will not be considered when a local database query is issued.

The second modification involved the issue of duplicate documents. If a user were to issue an identical query to the Web twice in a row, the exact same set of documents would be returned by the search engine and captured by the Document Repository module. The initial implementation handled this issue by simply replacing the existing copy with the most recently received copy. However, since this required examining all the captured URLs, as the database began to increase in size, a notable performance decrease was observed whenever a dynamic query to the Web was initiated. As a result, it was decided that the Document Repository module would simply capture and store duplicate copies. Hence, for now, the system handles the issue of duplicate copies *off-line* by periodically running a program to *clean* the database of any duplicate documents.

Figure 4. User interface.

2.6. User query module

The user interface to the system, shown in figure 4, allows to users initiate a query using key search words, the maximum number of URLs to be provided in the results, and a document type.

The user does not need to use the words in the document type because the system uses *query modification* [11] to add the document type to the query prior to issuance to the Document Repository module. The type field is used by the system to categorize document structure. In addition, the system provides the option of initiating the query from an existing web search engine (currently Altavista [1] or Google) or querying a local database.

If the user selects the local database, a basic *select* operation is performed to identify all appropriate documents. However, if the query is initiated on one of the search engines, the system performs a standard web search on the specified commercial search engine using the search words. Since the results of the search are simply a list of the pages deemed *relevant* by the search engine, the system must then retrieve the source code of each of the web pages and parse each document to capture and analyze the document structure. But, regardless of the option selected, the documents that are returned from the Document Repository are simply a set of web documents that match the initial search. Along with the *document type* selected by the user, these documents are then sent to the Machine Learning module as a vector of $\langle \text{tag}, \text{word}, \text{value} \rangle$ triples. The *value* component of this triple represents the number of times the corresponding *word* appeared between the associated begin/end *tag*.

2.7. Principle component analysis (PCA) system

The ultimate goal for Machine Learning module is to be able to provide the user with the capability to select different machine learning approaches for their queries. Currently, however the interface to provide this capability has not yet been designed. However, using the data gathered from the other system modules, various machine learning approaches can be tested *off-line*.

One such approach tested was using Principle Component Analysis (PCA). PCA is a well-known method for feature extraction, data compression, and multivariate data projection. Feature extraction can be used to reduce the dimensionality and hence improve the generalization ability of classifiers. Data projection methods enable the visualization of high dimensional data to better understand the underlying structure, explore the intrinsic dimensionality, and analyze clustering tendency of multivariate data [20].

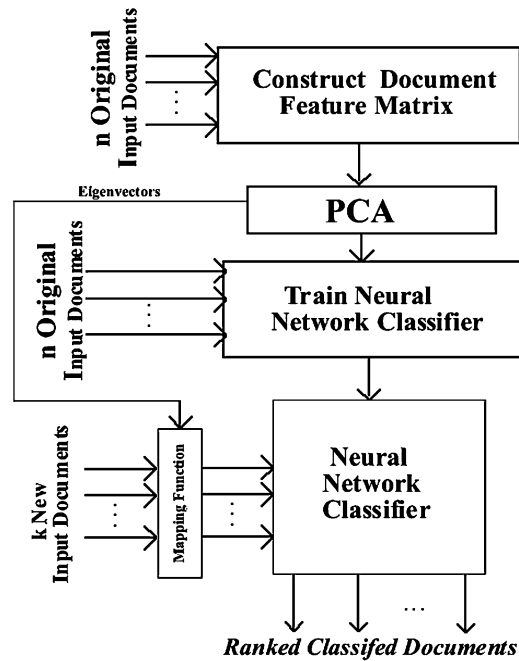


Figure 5. PCA architecture.

The architecture of the PCA system is shown in figure 5. In terms of the modules presented earlier, the Construct Document Feature Matrix module is essentially the Document Type Module and the n input documents are the *document feature vectors* labeled by document type. This module simply constructs a matrix of n rows with the number of columns defined, once again, to be the size of:

$$|\cup \langle \text{tag}_j, \text{word}_j \rangle \text{ of DFV}_i|$$

The entries of the matrix are simply the values of the $\langle \text{tag}, \text{word} \rangle$ pairs.

Once the feature matrix has been constructed, it is sent to the PCA module, which maps the original matrix to a new space and reduces the number of features. The end result is a new matrix containing n rows and m columns where the number of columns represents the new features defined by PCA. This matrix combined with the n original training documents is used to train the neural network classifier. For this system a two-layer backpropagation network [10] was used. Finally, after the neural network classifier has been trained, it is ready for user queries. Users can issue queries just as before, however apart from the neural network used, there are two significant differences. First, since PCA transforms the original input documents to a new space, using the eigenvectors created from PCA, a *mapping function* must be used transform the input documents to the new space. In addition, although the user does select a *document type* for their query, its value is not provided to the neural network. The final output is once again a set of ranked classified documents.

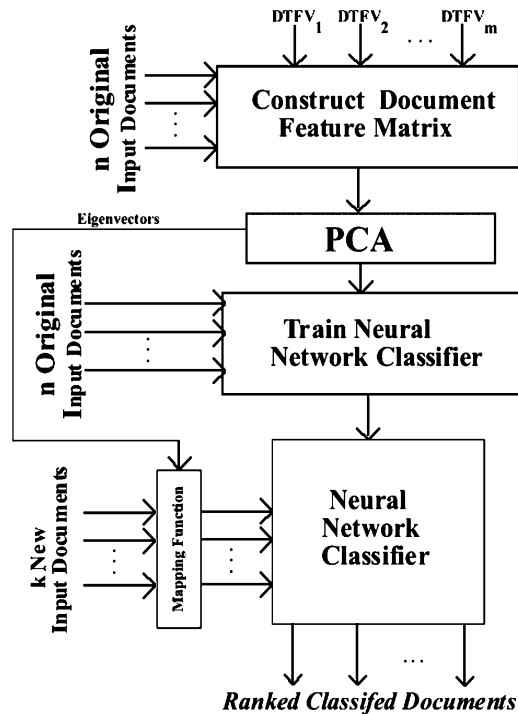


Figure 6. Hybrid PCA architecture.

2.8. Hybrid PCA system

Although PCA is a powerful technique for mapping large dimension matrices to much smaller dimension matrices, in order to work effectively, it typically requires a rather large sample set. However, the goal of the query-by-semantic system was to allow the user to dynamically add and remove document types during a query session. Since the larger the matrix, the longer PCA takes to compute, a hybrid approach was attempted in order to reduce the size of the original matrix. Figure 6 represents the architecture for this approach, and it is virtually identical the original PCA architecture with one subtle difference, specifically that the document type feature vectors are used in conjunction of the input documents to create the document feature matrix.

3. Analysis and results

Prior to the inclusion of the Document Database Module, the only way to validate the system had been to issue queries to the web and then analyze the results. This involved performing an exhaustive brute force approach of viewing each document to determine if it actually was of the desired type was necessary. It is our contention that the following analysis and results

not only validate the system but provide significant support to the quality of the results as well.

3.1. Document type feature vector analysis

The document feature vectors and document type feature vectors are dynamic in the sense that the user of the system can define them. However, they are static when related to the actual user query since both of these vectors can be constructed *offline* prior to the issuance of a query. Regardless, it should be clear based on simple analysis of all approaches, the time complexity of these algorithms is quadratic in terms of the words and tags.

Although the system allows the user much flexibility in setting parameters for things such as invalid tags, stop words, word size, etc., after analyzing preliminary results, it was decided that the Modified Saturated Base Value Sum algorithm should be used in the construction of the document type feature vector. The new system was originally constructed with the first method, the Incremental Sum algorithm. This method simply determined whether a $\langle \text{tag}, \text{word} \rangle$ pair existed in a document type feature vector with no consideration to the number of occurrences. However, by not counting the number of occurrences within a specific document, valuable content and structure information might be lost. For example, consider a document in which the word *University* was boldfaced and underlined three times. Clearly, this content and structure might be very relevant in a resume document. Hence, the Value Sum algorithm was designed to capture this information.

Unfortunately, the Value Sum algorithm had a major drawback. This was that one poorly selected document could severely corrupt a document type feature vector. As an example from actual testing, when constructing a document type feature vector for a document type representing “Faculty Home Page”, a well-structured faculty home page from a professor from South Carolina was utilized as one of the document feature vectors. However, this particular professor had a bullet list of published papers at the tail end of the document. The problem stemmed from the fact that most of these papers were published in conferences in North Carolina and South Carolina. As a result when the document type feature vector was constructed, the two most significant $\langle \text{tag}, \text{word} \rangle$ pairs were $\langle \text{ul}, \text{Carolina} \rangle$ and $\langle \text{i}, \text{Carolina} \rangle$. Although this content/structure combination might be relevant to a “South Carolina Faculty Home Page” document type, the word “Carolina” clearly should not be relevant to the more generic “Faculty Home Page” document type. As a result, it became apparent that this approach had the potential of adversely favoring one or two documents.

This above result led to the exploration of the Saturated Incremental Sum algorithm and Saturated Value Sum algorithm. The original intent was to limit the effects of specific $\langle \text{tag}, \text{word} \rangle$ pairs by placing an upper bound on the number of times they would be counted. Unfortunately, the Saturated Incremental Sum algorithm seemed to always perform worst than the original Incremental Sum algorithm and the Saturated Value Sum algorithm did not result in significantly improved results. If the upper bound was set too high, it performed just like the original Value Sum algorithm and if the upper bound was set too low, it performed very much like the Incremental Sum algorithm. In addition, the results varied with different document types, so finding a proper upper bound for all types was not possible.

It turned out that the Incremental Sum algorithm was actually the best approach of the first four algorithms. The logical explanation for this was that it actually identified the most common $\langle tag, word \rangle$ pairs among all example documents. In other words, it should be more relevant that a common $\langle tag, word \rangle$ pair appear in several different documents, than for a $\langle tag, word \rangle$ pair to appear several times in only one document. However, in another attempt to take advantage of the *value* information, the Modified Saturated Value Sum algorithm was designed. This algorithm placed the upper bound not on the total number of $\langle tag, word \rangle$ pairs in all documents, but on the total number of $\langle tag, word \rangle$ pairs within each document. The results observed when using this algorithm were a definite improvement over the other two algorithms that incorporated the value of $\langle tag, word \rangle$ pair provided the upper bound was not set too high. This was due to the fact that the algorithm limited the effect that a number of $\langle tag, word \rangle$ pairs could have on the document type feature vector. For example, if the $\langle tag, word \rangle$ pair $\langle ul, Carolina \rangle$ appeared twelve times in one document, instead adding a value of twelve for that pair to the document type feature vector, the value of the upper bound was assigned to that feature.

Unfortunately, although the results were favorable, for most cases this algorithm still did not perform as well as the Incremental Sum algorithm. The explanation for this was fairly simple. When using simply the *value*, two occurrences of a $\langle tag, word \rangle$ pair within a document results in a value of 2 whereas one occurrence results in a value of 1. The effect is that the algorithm identifies that having two occurrences of a feature within a document is *twice* as significant as simply one occurrence. Hence, this algorithm still placed too much importance on the multiple occurrences of a $\langle tag, word \rangle$ pair with a document. So, to solve this problem the Modified Saturated Base Value Sum algorithm was designed.

This algorithm was almost identical to the Modified Saturated Value Sum algorithm, but instead of directly using the *value* of a $\langle tag, word \rangle$ pair, a *base value* was added before inclusion into the document type feature vector. As an example, using a base value of 5, when a $\langle tag, word \rangle$ pair appeared only once in a document, the value associated with it would be 5. However, if it appeared twice, it would be given a value of 6. So, now the feature is identified as being more significant, but not twice as significant. And, as anticipated, this algorithm typically performed just as well, if not better, than the Incremental Sum algorithm in most cases.

Although the preceding paragraphs describe the process that was taken in determining how to construct the document type feature vector and provide a commonsense explanation as to the selection of which algorithm to incorporate into the system, in order to validate the decision, the following analysis was performed.

For each of the six algorithms, document type feature vectors were constructed for four different document types: Abstract, Recipe, Resume, and University Department Home page. In addition, the document type feature vectors were constructed from ten sample documents of each type. Using the Hamming neural network system, a *special query* was performed utilizing the local database, with the prototype vectors being constructed from the top twenty-five features of each document type. This special query was designed to ensure that a specific set of two hundred documents was used and returned as the results. These two hundred documents were a handpicked set of 50 *Abstract* documents, 50 *Recipe* documents, 50 *Resume* documents and 50 *University Department Home Page* documents.

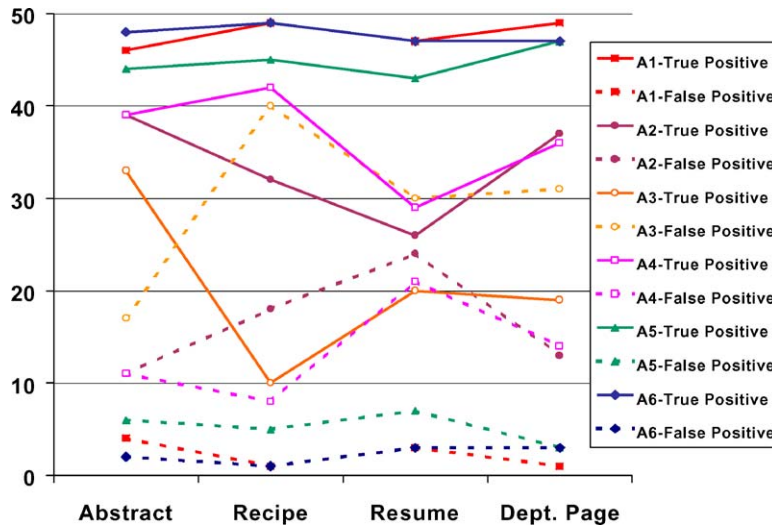


Figure 7. Document type feature vector algorithm analysis.

The results of the special query were the 200 documents classified into one of the four document types.

Figure 7 shows the results of the analysis and corroborates our initial observations when testing the different algorithms. For each of the six algorithms, the number of documents classified correctly was considered *true positive* results and the number of documents classified incorrectly was classified as *false positive* results. For this test, it turned out that both the Incremental Sum algorithm (A1) and the Modified Saturated Base Value Sum algorithm (A6) had *true positive* results of 95.5% and *false positive* results of 4.5%. The next best algorithm was the Modified Saturated Value Sum algorithm (A5), which had a 90 to 10% ratio of *true positives* to *false positives*. The Saturated Value Sum algorithm (A4) and the Value Sum algorithm (A2) both performed poorly having *true positive* results and *false positive* results of 73 to 27% and 67 to 33%, respectively. Finally, the Saturated Incremental Sum algorithm (A3) performed dismally with a higher *false positive* result of 55% than the *true positive* result of 45%.

Another issue that was studied was determining the optimal number of example documents (or document feature vectors) needed to create a document type feature vector. Optimal probably is not the best word to use here since the optimal number is *as many as possible*. But, since one of the fundamental goals of the system was to allow the user to create the document type feature vectors by providing example documents, the ideal number actually should be as small as possible. Clearly, requiring the user to provide hundreds of sample documents would be counterproductive to the whole idea of reducing the number of relevant documents returned in a query, since the user would have to spend time finding or generating these sample documents. However, the system is robust enough to create document type feature vectors from any number of sample documents.

To perform this analysis, using the Hamming neural network system, the same *special query* that was used to validate the algorithm selection was performed. In addition, the document type feature vectors were constructed from the same set of two hundred document type samples listed earlier, however, this time only the Modified Saturated Base Value Sum algorithm was used. The first test was performed using all two hundred sample documents (i.e. 50 documents of each type). Subsequent tests were then performed by creating document type feature vectors with sample sizes of 100, 60, 48, 40, 32, 20, and 12 documents, where an equal number of documents from each document type were selected in each test. In addition, for the construction of the prototype vectors for the Hamming neural network, the number of features selected from each document type feature vector was twenty-five. The choice of this value is validated in the following section.

Figures 8–15 show the classification success/failure rate when attempting to classify the original two hundred documents using the feature vectors constructed from the specified number of sample documents. In figure 8, when all 200 documents are used to create the document type feature vectors, a classification success rate of 97.5% was achieved. And, an even better classification success rate of 98% was observed when only 25 documents of each type were used, as can be observed in figure 9. This difference is not significant since

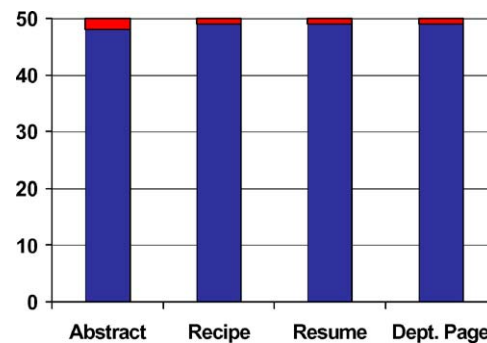


Figure 8. # Documents = 200.

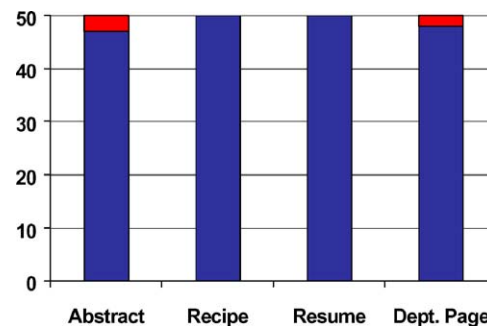


Figure 9. # Documents = 100.

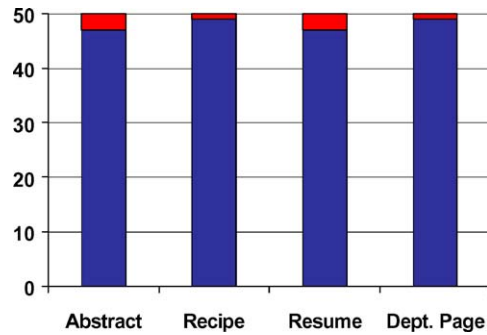


Figure 10. # Documents = 60.

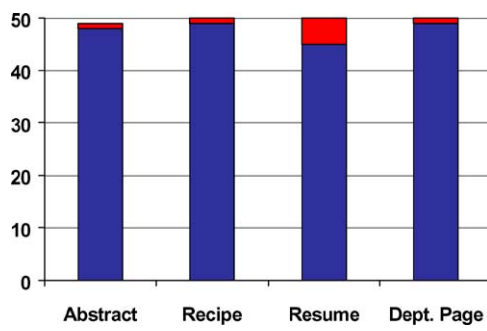


Figure 11. # Documents = 48.

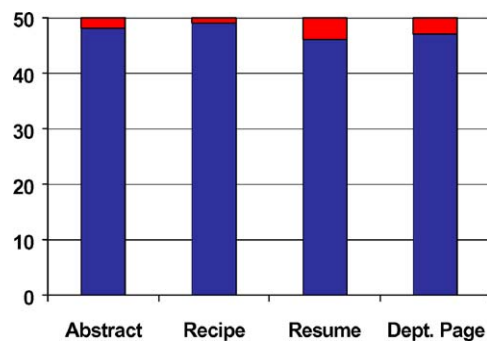


Figure 12. # Documents = 40.

it simply reflects the fact that only one more document was classified correctly, and so the real conclusion to draw here is that there is no significant difference between constructing the document type feature vectors from 200 documents as opposed to 100 documents.

In addition, as figures 10–13 show, there is very little difference between selecting a sample size of 60, 48, 40, or 32, as the classification success rate for these are 96%, 95.5%, 95% and 94.5%, respectively.

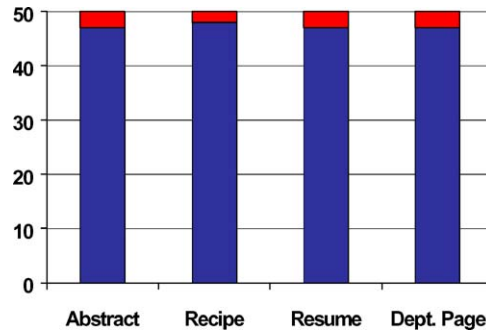


Figure 13. # Documents = 32.

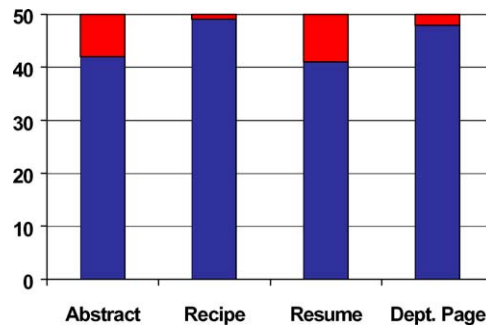


Figure 14. # Documents = 20.

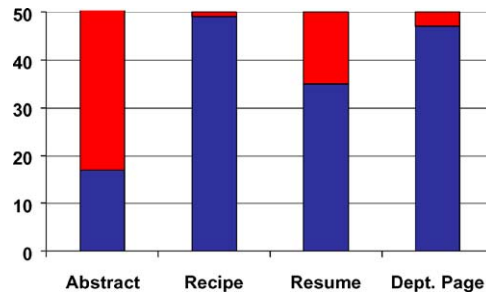


Figure 15. # Documents = 12.

In fact, even when only five document of each type are selected (twenty documents total), a classification success rate of 90% is still achieved as can be seen in figure 14. However, figure 15 shows that there is a significant drop off when only *three document type vectors* are selected from each category as the classification success rate falls to 74%. Hence, the results would indicate that selecting a sample size of approximately ten documents should provide fairly good results.

In addition, as figures 10–13 show, there is very little difference between selecting a sample size of 60, 48, 40, or 32, as the classification success rate for these are 96%, 95.5%, 95% and 94.5%, respectively.

3.2. *Prototype vector selection*

Another decision studied was determining how many features to select from each document type feature vector when constructing the prototype vectors for the system. Theoretically, the number of features in the document type feature vector could be nearly as large as the size of the set combining the entire document feature vectors. However, a majority of these features would likely be irrelevant, especially since a large percentage of the $\langle \text{tag}, \text{word} \rangle$ pairs appear in only one document. Since the size of this vector ultimately determines the number of features to be compared by the systems, it was important to select a large enough size to ensure high quality results but not so large as to significantly effect the performance of the system.

To test the prototype vector size, another analysis was performed, once again using the same *special query*, with the Modified Saturated Base Value Sum algorithm and the document type feature vectors constructed using the same two hundred document type samples utilized in earlier tests. The variable for this test was to run several queries adjusting the number of features used by the Hamming neural network system. In other words, although the document type feature vector maintained by the system may contain several thousand features for each document type, the number of features used in the construction of the prototype vectors should be considerably less. The objective of this analysis was simply to determine a reasonable lower bound.

Figures 16–21 display the results of the analysis of constructing the prototype vectors for each document type using n features from each document type feature vector. In each test, the value of n was decreased and the results were analyzed to determine the number of successful and unsuccessful classifications. However, regardless of the total number of features selected, the top n features of each document type were always selected.

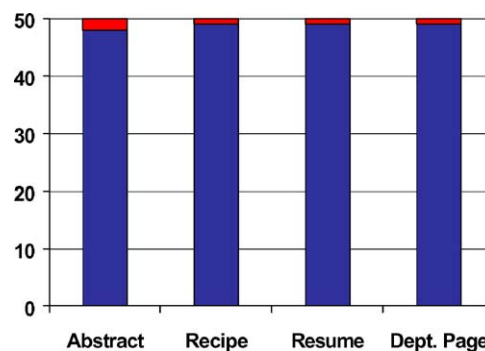


Figure 16. Top 100 features.

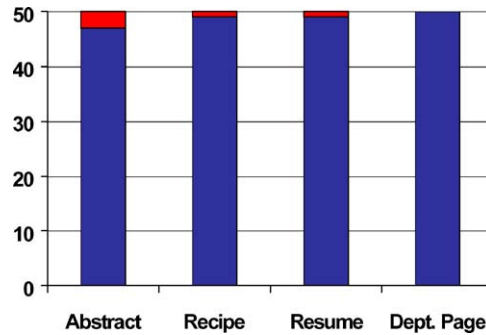


Figure 17. Top 50 features.

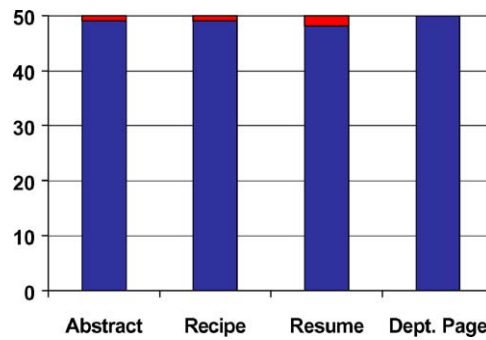


Figure 18. Top 25 features.

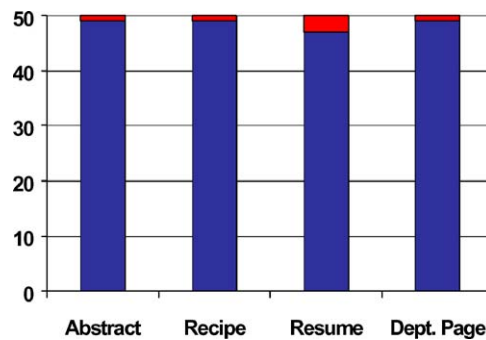


Figure 19. Top 20 features.

Clearly, the results from figures 17–21 shows that regardless of whether the top 100, 50, 25, 20, or even top 15 features are used in the construction of the prototype vectors, the system still classified successfully at or above 97%. In fact, as can be observed in figure 21, even when only the top 10 features were used, the system still achieved a classification success rate of almost 95%.

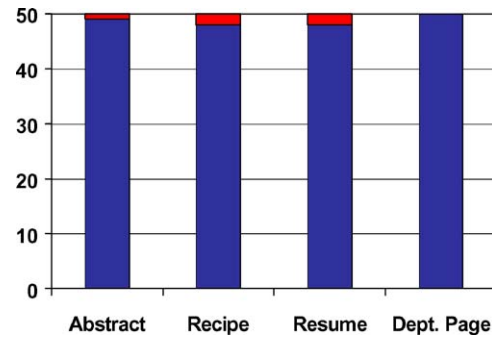


Figure 20. Top 15 features.

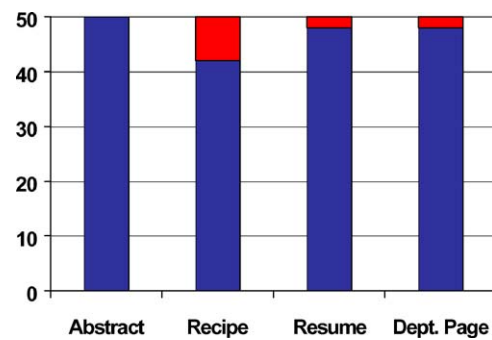


Figure 21. Top 19 features.

In the analysis of both the document type feature vector algorithms and the document type vector size, both tests selected twenty-five features from the document type feature vectors when constructing the prototype vectors of the system. Clearly, the above results show that twenty-five features were more than adequate. However, it should also be evident that the choice of both the number of sample documents used to create the document type feature vector as well as the number of features to select from this vector when creating the prototype vectors are highly related.

3.3. System test

The experiments in the previous section all utilized the same set of documents to both train the network as well as to perform the various validation tests. To truly test the system, a new set of 20 sample documents was obtained. Then a new *special query* designed to ensure that the new set of 20 documents was used and returned as the results was performed. Four tests were performed using a different sample size to create the *document type feature vectors*. The results are shown in figure 22 for sample sizes of 10, 15, 25, and 50.

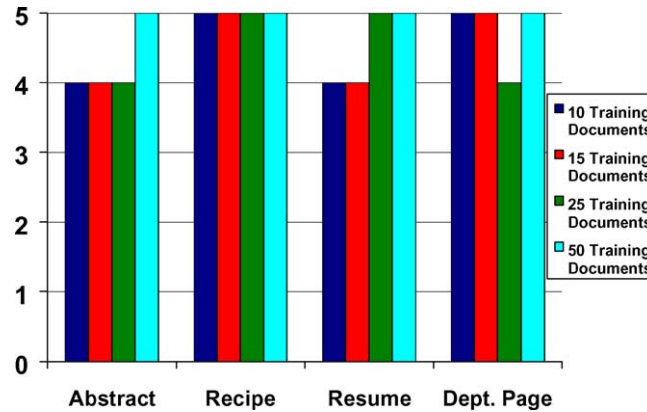


Figure 22. New sample test results.

As the results show, the classification for document type feature vectors created with 10, 15, and 25 sample documents all resulted in a 90% success rate. It required a sample set of 50 documents before the classification success rate reached 100%. The tests shown in figure 22 used the top 10 features of each *document type feature vector* to construct the prototype vectors. Other tests were performed where the number of features selected from the document type feature vectors, but there was virtually no change in the results.

3.4. PCA query-by-semantics systems analysis

Compared to the tests from the Hamming neural network system, the initial results of the PCA approaches were too dismal to even warrant much discussion. However, some positive analysis was obtained in validating the use of the document type feature vectors to improve the PCA system.

First, a document feature vector matrix was using the same two hundred sample documents used in the previous sections. This matrix contained two hundred rows and over fifty thousand columns which represented the total number of unique $\langle \text{tag}, \text{word} \rangle$ pairs among all two hundred documents. The entries in the matrix were the values associated with the $\langle \text{tag}, \text{word} \rangle$ pairs for each document. Figure 23 shows the process of taking this rather large matrix and using PCA to reduce the dimensionality. However, instead of using the PCA resultant matrix to train the neural network, the matrix was input into the Weka toolkit [12], after adding an extra column with a label to identify each row's document type.

The Weka toolkit is an open-source collection of machine learning algorithms written in Java designed to solving real-world data mining problems. Weka is well suited for developing new machine learning schemes and has several modules that can be applied directly to a data set. One such module implements an algorithm called 10-fold cross-validation. The 10-fold cross-validation algorithm takes a matrix with labeled rows and trains a classifier to be able to recognize which class a vector might belong to. The label on each row identifies the class.

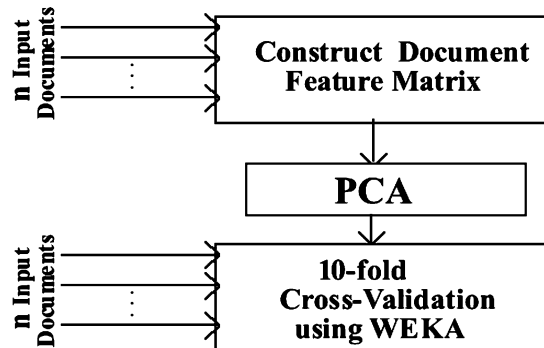


Figure 23. 10-fold cross validation.

The 10-fold cross validation algorithm works as follows. First, during the training phase, the algorithm takes as input 90% of the vectors in the matrix to train the classifier. In this step, the vectors that are used contain the class label. Next, during the testing phase, the remaining 10% of the vectors are input with the class label removed. The terminology “10-fold” represents that the process is repeated ten times, with different vectors selected for training and testing, with the results being averaged.

The general idea is that 10-fold cross-validation can be used to validate the input data. Hence, for the systems presented here, this algorithm can be used to determine whether the document feature vectors provide an adequate representation of the document type.

Figure 24 shows the results of the cross-validation of the original matrix. As can be observed, the results were not particularly good. One logical explanation for this is that the features of the original vectors input into the Document Feature Construction module contained the values associated with each *<tag, word>* pair. This is analogous to the use of the values in the Value Sum algorithm which during testing was shown to be an inferior approach and hence not as useful.

With that in mind, another test was performed using the PCA Hybrid approach of first building the document type feature vectors from the original input documents. After

- **Original Matrix Size** **200 x 50730**
- **Matrix Size after PCA applied** **200 x 133**
- **Correctly Classified Instances** **131 (65.5%)**
- **Incorrectly Classified Instances** **69 (34.5%)**

```

=== Confusion Matrix ===
  a  b  c  d  <-- classified as
40  2  4  4 | a = Abstract Document Type
 7 38  2  3 | b = Recipe Document Type
10 11 20  9 | c = Resume Document Type
 5  4  8 33 | d = Dept. Page Document Type
  
```

Figure 24. Cross validation results.

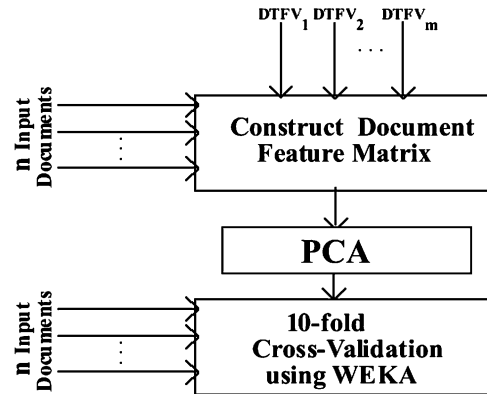


Figure 25. 10-fold cross validation with document type feature vectors.

construction of these vectors, the input vectors and the document type feature vectors were input into the Document Feature Construction Module, which then transformed the input vectors into new vectors in a fashion similar to the construction of the prototype vectors of the Hamming neural network. This process is shown in figure 25.

The results of this test are shown in figure 26. Clearly, there was a significant improvement over the previous cross-validation test. In fact, over 90% of the documents were classified correctly. Hence, if anything, this at least further validates the use of the document type feature vectors in performing classification, and demonstrates that even with a relatively small set of sample documents, classification can be effectively performed.

Finally, so as to perform a similar comparison between the modified feature matrix and the original feature matrix, another cross-validation test was performed using only the top 45 features from the matrix generated by original PCA matrix.

As can be seen from the results in figure 27, this clearly would not be an effective approach to reducing the number of features. However, although the cross-validation results of the straightforward PCA approach were not that promising, it does not necessarily mean that

- **Original Matrix Size** 200 x 351
- **Matrix Size after PCA applied** 200 x 45
- **Correctly Classified Instances** 181 (90.5%)
- **Incorrectly Classified Instances** 19 (9.5%)

```

=== Confusion Matrix ===
  a  b  c  d  <-- classified as
45  3  2  0 | a = Abstract Document Type
 4 44  1  1 | b = Recipe Document Type
 0  1 46  3 | c = Resume Document Type
 2  2  0 46 | d = Dept. Page Document Type
  
```

Figure 26. Cross validation results using document type feature vectors.

- Original Matrix Size 200 x 50730
- Matrix Size after PCA applied 200 x 45
- Correctly Classified Instances 89 (44.5%)
- Incorrectly Classified Instances 111 (55.5%)

```

=== Confusion Matrix ===
  a  b  c  d  <-- classified as
27  5  5 13 | a = Abstract Document Type
11 16  4 19 | b = Recipe Document Type
10  8 10 22 | c = Resume Document Type
 6  1  7 36 | d = Dept. Page Document Type

```

Figure 27. Cross validation results with reduced number of features.

the approach is not worth pursuing. The most likely problem is that for PCA to function optimally, it requires a very large set of sample documents. Although two hundred samples seem like a large number, PCA typically performs best with significantly larger sample sizes. However, this is a significant problem as it relates to the current approach being studied. Namely, the objective is to create a dynamic system that allows the users to provide a relatively small set of sample documents, and still be able to achieve fairly relevant queries with improved results. However, the cross-validation results of the hybrid approach did give fairly promising results, which definitely warrant further study.

3.5. Final remarks







The system is available for public use. To request an account, simply follow the instructions at: <http://acs.madonna.edu/qbys/>

4. Extension of our system in semantic multimedia document search

Recent development of digital media technology has generated a huge amount of multimedia documents on the Web. Effective and efficient search of multimedia documents has merged as an important and challenging area in multimedia computing. Research in human perception of multimedia content suggests that the semantic cues play an important role in multimedia document retrieval. Users usually pose and expect a retrieval to provide answers to semantic questions (for example, show me an image of the sky), and not low level (for example, show me a predominantly blue and white image). However, current state-of-art computer vision technology lags far behind the human's ability to assimilate information at a semantic level [22]. It is clear that retrieval by similarity of visual attributes when used arbitrarily cannot provide semantically meaningful information. For example, a search for a red flower by color red on a very heterogeneous database cannot be expected to yield meaningful results.

Many multimedia documents on the Web have some associated annotation information such as keywords, title, date, and authors. If not, we could still get similar information on the basis of the text surrounding the multimedia documents [5]. Our system could be easily adopted to produce *multimedia document type feature vector* based on the annotation

Table 1. 10 Multimedia document types.

				
Autumn	Cloud	Blue sky	Desert	Flower
				
Sea	Snow mountain	Snow Tree	China	Canadian

information and facilitate the multimedia document search. We collected a small set of images from Corel CD (200 landscape images). Each image comes with around 20 keywords annotated by Corel employees. Based on the annotations, we group the images into 10 multimedia document types (see Table 1) and use our system to generate corresponding feature vectors. Our preliminary results show that our approach could provide more meaningful retrieval results when compared with pure visual attribute search.

5. Related works

Popular World Wide Web search engines such as AltaVista and Yahoo! [25] allow users to search for web pages by providing search words which are used to query the search engine's database for URLs of web pages containing those search words. However, most search engines utilize very little, if any, detailed structure information. HotBot [13] is one search engine that does give the user the capability to specify that the documents they are searching for contain images, video, and/or Javascript code. In addition, Google is another search engine that utilizes structure information. The exceptional research paper, "The Anatomy of a Large-Scale Hypertextual Web Search Engine" [4], published on the inner workings of the Google search engine, discusses use of various HTML tags with emphasis on document linking. Due to Google's mainstream acceptance, this is a clear indication of the relevance of utilizing structure within a document search.

Most search engines dynamically search the web. However, these types of searches are typically done in the *background*. In other words, these search engines do not allow the user to directly *query* the web but rather to directly query the search engine's database. A real web query system locates documents by dynamically retrieving and scanning the

documents during the query process. Although there may not exist any mainstream search engine that can directly query the web, there has been a significant amount of research done in this area. WebSQL [21], WebLog [17], and W3QS [16] are examples of languages and systems used to directly query the Web. All three of these approaches use a limited amount of document structure information when analyzing documents retrieved from a web query.

There has also been a significant amount of research in an attempt to improve web searches using machine learning techniques. *WebWatcher* [5, 20] and LASER [3], for example, are systems that assist users in locating desired information on a specific site. These systems track the user's actions and utilize machine learning methods to acquire knowledge about user's goals, web pages users visit, and success or failure of the search. Rankings are based on keywords. In addition, LASER incorporated document structure into the search process. Another system, called Syskill and Webert [23], utilizes six different machine learning algorithms. This system also requires user interaction by requesting that the user rate pages on a three-point scale. After analyzing information on each page, individual user profiles are then created. Based on the user profile, the system suggests other links that might be of interest to the user.

Finally, two systems that are probably the most similar the research presented here are: Watson [22, 25] and Inquirus 2 [8, 23]. Both systems are currently accessible to users on the Web. In addition, both systems also utilize existing search engines and enhance the search with query modification techniques. And, both systems take advantage of document structure when determining what information to present to the user. However, there are some significant differences to the systems presented here.

Watson, for instance, only considers four kinds of structure: normal words emphasized words, de-emphasized words, and list item. The authors were not entirely clear on what constituted a word within each of the specific categories, although they did provide some basic explanations. However, it seems apparent that their use of structure is hard-coded and fixed by the system providing no mechanism for the user to specify what is emphasized and what is not. In addition, it does not appear that the system is using any type of machine learning techniques to analyze the documents. Finally, the system isn't designed so much to identify document types as much as it is attempt to identify information relevant the users current task at hand, which it accomplishes by monitoring what the user is currently doing or viewing and requesting additional feedback as well.

Inquirus 2, on the other hand, has several more things in common with the research presented here. First there is the use of document structure. Inquiris 2 converts document into vectors of words (or phrases) within specific structure elements. The structure elements include: title, 1st 75 terms within a document, anywhere in the text, in a heading or emphasized, in the anchor text and other special words, symbols or characters. Coincidentally, the authors have identified these structure elements as *document vector types*.

As the name implies, the Inquirus 2 system converts web documents into document type vectors. The system then attempts to classify the documents according to a *category*. The designers of this system chose to use SVM [6] to classify documents. In [8], the authors describe a test where they took a sample set of 2618 positive and negative examples of *personal home pages* web documents and 2703 positive and negative examples of *call for papers web* documents. These pages were manually gathered by exhaustively searching

various web sites. The authors claim to have achieved a better than 2% *false positive* rate for the classification of these document types.

There are two significant differences to the Inqiris 2 system and the research presented here. First, although document structure an integral part of the Inqiris 2 system, the document structure analyzed by the system is fixed. And, more significantly, the designers of the system also predetermine the document type thus limiting the number of possible document types. This concept alone was the fundamental reason for redesigning the query-by-structure system almost two years ago. Granted the system does create document types based on example documents instead of hard-coding the features. However, there is no mechanism in place for users of the system to create their own document types. Currently, on the most recent visit to the Iquiris 2 web site, the following categories were listed: *Personal Home Pages, Research Papers, Product Reviews, Guides and FAQ, Calls for Papers, and Genealogy*.

6. Conclusions and future research

The content of a document is clearly crucial in any type of query. However, the research presented here has shown that the query-by-structure and query-by-semantic approaches have potential as well. The CLaP system showed us that utilizing structure as well as content during a web search could significantly improve query performance. However, the system required that the user specify specific details about the document structure as part of their query. As a result, when performing even the simplest of queries, an inordinate amount of information was required from the user. In order to eliminate these complex queries, we chose to test the feasibility of a different approach that utilized neural networks within the query process. The neural network systems resulted in an equally good performance and demonstrated that the burden of identifying structure within the document can become the responsibility of the system rather than the user.

However, although the initial query-by-structure systems provided promising results, to produce a viable system, it was necessary to incorporate some modifications. Clearly, the database module needed to be implemented, as the query response turnaround time was much too long. In addition, the modification of the document feature vector solved the issue of the designer defining the document type. Now, the user basically has the ability to say “give me a document that has these words and looks like this.”

Finally, it should be noted *anointed* father of the World Wide Web, Hans Berners-Lee, claims that there is currently a paradigm shift taking place towards a Semantic Web [2]. The basic notion is that new web documents are being creating with specific semantic tags, primarily using XML. These tags provide additional meaning to the text within the document. In other words, the HTML tags within a document are used to describe the structure or layout of the document and the XML tags are used to provide a more detailed description of the content of the documents. New search tools are being created to identify documents containing these descriptive semantic tags that will potentially revolutionize the way web searches are performed and immensely increase the quality of these searches.

At first glance, the concept of the Semantic Web might lead one to think that the query-by-semantic approach presented here could become obsolete rather shortly. However, there

are at least two major implications that should lead one to conclude the exact opposite. First, what is to be done with the billions of already created or soon to be created web pages containing only HTML with no semantic information whatsoever. Clearly, it would be unreasonable to think that all of these documents must be redesigned to adapt to the Semantic Web. Secondly, if one thinks of these semantic tags as simply another form of structure (i.e. semantic structure instead of presentation structure), the query-by-semantics approach might, in fact, be extremely relevant to the Semantic Web.

In fact, in [18], Lu et al. describe some of the flaws in the current semantic web search engines and claim: "XML is a promising technique since it keeps content, structure, and representation apart and is a much more adequate means for knowledge representation. However, XML can represent only some semantic properties through its syntactic structure. XML queries need to be aware of syntactic structure". The authors then proceed to describe how Semantic Web search engines will be built with ontologies that will describe the context of a document. However, we contend that the concept of document semantics and document types, with some additional enhancements and modifications, the systems describe in this research will be able to thrive on the Semantic Web.

References

1. Altavista. The Internet. <http://www.altavista.com>
2. T. Berners-Lee, J. Hendler, and O. Lassila, The Semantic Web, Scientific American, May 2001, pp. 35–43.
3. J. Boyan, D. Freitag, and T. Joachims, "A machine learning architecture for optimizing web search engines," AAAI-96 Workshop on Internet-Based Information Systems, Portland, OR, 1996, pp. 334–335.
4. S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in Proceedings of the 7th International World Wide Web Conference, Brisbane, Australia, 1998, pp. 107–117.
5. S.F. Chang, J.R. Smith, M. Beigi, and A. Benitez, "Visual information retrieval from large distributed online repositories," *Comm. ACM*, Vol. 40, No. 12, pp. 63–71, 1997.
6. C. Cortes and V. Vapnik, "Support vector networks," *Machine Learning*, Vol. 20, pp. 273–297, 1995.
7. F. Fotouhi, W. Grosky, and M. Johnson, "CLaP: A system to query the web using content, link, and presentation information," in Proceedings of the 14th International Symposium on Computer and Information Sciences, Kusadasi, Turkey, 1999, pp. 214–221.
8. E. Glover, G. Flake, S. Lawrence, W. Birmingham, A. Kruger, C. Giles, and D. Pennock, "Improving category specific web search by learning query modifications," in Symposium on Applications and the Internet, SAINT San Diego, California, 2001.
9. Google. The Internet. (<http://www.google.com>)
10. M. Hagan, H. Demuth, and M. Beale, *Neural Network Design*, 1st ed., Boston, MA, 1996.
11. D. Harman, "Relevance feedback and other query modification techniques," *Information Retrieval: Data Structures and Algorithms*, Chapter 11, pp. 241–263, 1992.
12. G. Holmes, A. Donkin, and I. Witten, "Weka: A machine learning workbench," in Proceedings of the 1994 Second Australian and New Zealand Conference on Intelligent Information Systems, Brisbane, Australia, 1994, pp. 357–361.
13. HotBot. The Internet. <http://www.hotbot.com>
14. M. Johnson, F. Fotouhi, and S. Draghici, Query-by-Structure Approach for the Web. *Data Mining: Opportunities and Challenges*, Chapter 13, pp. 301–322, 2003.
15. M. Johnson, F. Fotouhi, and S. Draghici, "A neural network net query-by-structure approach," in 12th International Conference of the Information Resources Management Association, IRMA'01, Toronto, Canada, 2001, pp. 108–111.
16. D. Konopnicki and O. Shmueli, "W3QS: A query system for the world wide web," in Proceedings of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland, 1995, pp. 11–15.

17. L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian, "A declarative language for querying and restructuring the web," in Proceedings of the Sixth International Workshop on Research Issues in Data Engineering, New Orleans, LA, 1996, pp. 12–21.
18. S. Lu, M. Dong, and F. Fotouhi, "The semantic web: Opportunities and challenges for next-generation web applications," *Information Research*, 7(4). Special Issue on the Semantic Web, 2002.
19. S.K. Madria, S.S. Bhowmick, W.K. Ng, and E.P. Lim, "Research issues in web data mining," in Proceedings of Data Warehousing and Knowledge Discovery, First International Conference, DaWaK '99, 1999, pp. 303–312.
20. J. Mao and A. Jain, "Artificial neural networks for feature extraction and multivariate data projection," *IEEE Transactions on Neural Networks*, Vol. 6, No. 2, pp. 296–316, 1995.
21. A.O. Mendelzon, G.A. Mihaila, and T. Milo, "Querying the world wide web," in Proceedings of the International Conference on Parallel and Distributed Information Systems (PDIS'96), Miami, Florida, 1996, pp. 54–67.
22. M. Naphade and T. Huang, "Extracting semantics from audiovisual content: The final frontier in multimedia retrieval," *IEEE Trans. on Neural Networks*, Vol. 13, No. 4, pp. 793–809, 2002.
23. M. Pazzani, J. Muramatsu, and D. Billsus, "Syskill & Webert: Identifying interesting web sites," in Proceedings of the 13th National Conference on Artificial Intelligence, Menlo Park, CA, 1996, pp. 54–61.
24. The ht://dig Group. The Internet. (<http://www.htdig.org>)
25. Yahoo! The Internet. <http://www.yahoo.com>



Michael Johnson obtained his B.S. in computer engineering from the University of California, San Diego in 1987. After working in industry at GTE Communications and AT&T Bell Labs for four years, he returned to school and obtained his M.S. in computer science at Michigan State University in 1993. Dr. Johnson completed his Ph.D. studies at Wayne State University earlier this year. In addition, for the past nine years he has been a professor and head of the Computer Science Department at Madonna University in Livonia, Michigan.



Farshad Fotouhi received his Ph.D. in computer science from Michigan State University in 1988. He joined the faculty of Computer Science at Wayne State University in August 1988 where he is currently a Professor and Associate Chair of the department.

Dr. Fotouhi major area of research is databases, including relational, object-oriented, multimedia/hypermedia systems, and data warehousing. He has published over 80 papers in referred journals and conference proceedings,

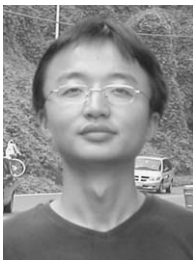
served as program committee members of various database related conferences. He is a member of the ACM and the IEEE Computer Society.



Sorin Drăghici received the Ph.D. degree in Computer Science from University of St. Andrews, St. Andrews, UK in 1995. From 1998 to present he is an Associate Professor in the Department of Computer Science, Wayne State University. He is also Director of the Intelligent Systems and Bioinformatics Laboratory and Scientific Director of the Bioinformatics Core of Wayne State University. He is a member of the IEEE, INNS and AAAS. His research interests include bioinformatics, neural networks, data analysis, gene expression and genomics. On these topics he has authored or co-authored over eighty papers in international journals and conferences.



Ming Dong received his B.S. degree from Shanghai Jiao Tong University, Shanghai, P.R. China in 1995 and the Ph.D. degree from University of Cincinnati, Cincinnati, Ohio, in 2001, all in electrical engineering. He joined the faculty of Wayne State University, Detroit, MI in 2002 and is currently an Assistant Professor in the Department of Computer Science. His research interests include pattern recognition, multimedia document retrieval, and financial engineering.



Duo Xu received the B.E. degree in Communication and Control Engineering in Xi'an Jiaotong University, China in 2001. He is going to graduate as M.S. in Fall 2003, in Machine Vision and Pattern Recognition Laboratory, Computer Science Department, Wayne State University. His research interests focus on image and document retrieval.