A Personalisable Hierarchical Intranet Document Categoriser

A thesis submitted to the University of Manchester
for the degree of Master of Science
in the Faculty of Science and Engineering

2005

James William Furness

School of Computer Science

http://www.base6.com/dagama/

Contents

1 Introduction 9
1.1 Background 9
1.2 Motivation 10
1.3 Project Goals 10
1.4 Report Structure 11

2 Background 13
2.1 Introduction 13
2.2 Overview of Information Retrieval 15
2.2.1 Introduction 15
2.2.2 Measuring Effectiveness 16
2.2.3 Document Representation 17
2.2.4 Query Processing 22
2.3 Overview of Text Mining 24
2.3.1 Introduction 24
2.3.2 Unsupervised Classifiers 27
2.3.3 Supervised Classifiers 29
2.4 Overview of HTML 41
2.5 Overview of HTTP 42
2.5.1 Protocol Overview 42
2.5.2 HTTP 1.1 43
2.5.3 CGI Scripts 44
2.6 Summary 44

3 Design Overview 45
3.1 Introduction 45
3.2 Requirements 45
3.2.1 Introduction 45
3.2.2 Document Acquisition 45
3.2.3 Document Transformation 48
3.2.4 Document Classification 49
3.3 System Architecture 53
3.3.1 Introduction 53
3.3.2 Document Acquisition 55
3.3.3 Document Transformation 55
3.3.4 Document Classification 57
3.4 Summary 57

4 Detailed Design 59
4.1 Introduction 59
4.2 Implementation Tools and Techniques 59
4.3 Portability 62
4.4 Design Highlights 62
4.4.1 System Configuration 62
4.4.2 General Architecture 62
4.4.3 SQL Server Data Access Components 66
4.4.4 Web Crawler 74
4.5 Summary 75

5 Testing 77
5.1 Introduction 77
5.2 Unit testing 77
5.3 Integration testing 77
5.3.1 Document Acquisition Process 77
5.3.2 Term Stats Update Process 78
5.3.3 Classifier Training Process 78

5.3.4 Classification Process 79
5.3.5 Web Crawler Plugin 79
5.4 Effectiveness testing 79
5.4.1 Document Acquisition Scalability 80
5.4.2 Term Statistics Updating 81
5.4.3 Hierarchical PrTFIDF Training 83
5.4.4 Classification 84
5.4.5 Conclusion 85
5.5 Further Testing 85
5.6 Summary 86
6 Evaluation 87
6.1 Introduction 87
6.2 Summary of Goals 87
7 Conclusion 89
7.1 Limitations 89
7.2 Extensions 89
7.3 Summary of achievements 90
8 References 91
I Implementation Details 95
I.1 Introduction 95
I.2 System Framework 95
I.3 System Plugins 105

Table of Figures

Figure 1 Growth of the World-Wide Web 1995-2005 [2] 9
Figure 2 The business opportunity in text mining [34] 13
Figure 3 "Search" versus "Discover" [9].. 14
Figure 4 A general model of information retrieval [5] 15
Figure 5 The 'contingency' table [6] 16
Figure 6 The "bag of words" model [31] 20
Figure 7 Luhn's Word-Frequency diagram [19] (Adapted in [6]) 21
Figure 8 Two-dimensional vector space with terms as basis 23
Figure 9 The Text Mining Process [9]... 24
Figure 10 Taxonomy of Web Mining (Adapted from [20]) 25
Figure 11 Taxonomy of Classification Techniques (Examples from [22]).............. 27
Figure 12 A dendogram [27] 29
Figure 13 6-Nearest Neighbour Classification [27].. 30
Figure 14 Illustration of Rocchio Text Categorisation [27] 31
Figure 15 Illustration of a Disjunctive Category under Rocchio [27].................... 32
Figure 16 Category Hierarchy 38
Figure 17 Breadth-first search of Hierarchy 40
Figure 18 Pipelining [29]... 44
Figure 19 System Architecture.. 54
Figure 20 ThreadPool illustration [37]... 60
Figure 21 HTTP connection performance counters 63
Figure 22 SQL Server Database Schema 66
Figure 23 Node training data update 71
Figure 24 Hierarchical PrTFIDF subtree training propagation............................. 73
Figure 25 Concurrency limiter performance counters 102

Abstract

A novel system is proposed for the indexing, searching and browsing of an intranet document repository for use as part of a corporate extranet. The system allows users to browse a hierarchically organised collection of documents. The hierarchy is automatically maintained by the system after a minimum of training. Users additionally have the option to personalise the hierarchy in order to organise the documents in any way they see fit. This document presents an overview of the design of the system and the reasons for the various design choices.

The system has been implemented and initial tests on the system have been conducted which show that the system would comfortably be able to handle a repository sized between 3,000 and 30,000 documents.

Declaration

No portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Acknowledgements

I would like to thank my supervisor David Brée and my second marker Mary McGee Wood for their supervision and invaluable assistance during the course of this project, providing encouragement, ideas and reference sources.

I am also thankful to EDF for providing resources during the development and testing of this project, in particular Gertjan Schuurmans and Daniel Peters for initially proposing the project and for their guidance during discussions about the project.

Finally, I would also like to thank Alvaro Fernandes and Craig Jones for their assistance during discussions about the project.

1 Introduction

1.1 Background

Since its birth in 1990, the world-wide web has shown phenomenal growth, due to the huge growth of the internet, combined with its ideal suitability[1] as a mechanism for the rapid dissemination of information over the internet. The mass availability of information through the world-wide web has spearheaded the growth of Internet access and this has in turn further encouraged the growth of the world-wide web.

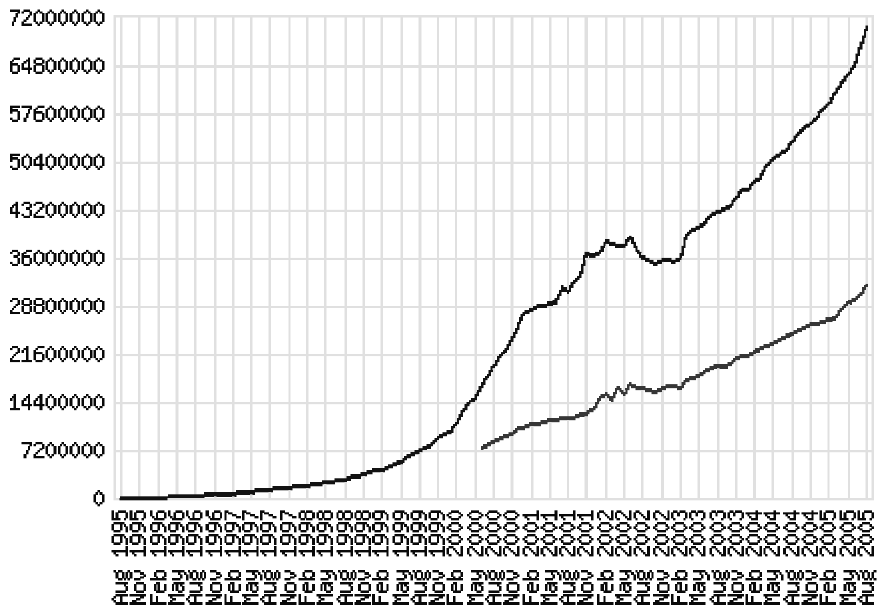


Figure 1 Growth of the World-Wide Web 1995-2005 [2]
(Upper line indicates hostnames, lower indicates unique hosts)

In the late 1990s, web portals such as Yahoo! [3] began to appear, and many companies tried to build or acquire a portal. Services such as email, chat, games and news were provided in order to secure the user-base by encouraging repeat visits, and to increase the length of time spent on the portal by users. [1]

During the early 2000s, the focus of the web portal industry shifted towards the corporate intranet portal, or "enterprise web". Although the expectation of repeat visits from millions of unaffiliated users and thus the generation of advertising

[1] It provided a simple and universal means for document retrieval (*HTTP*), a HyperText system for document markup (HTML) and a means for uniquely and efficiently addressing all documents (*URLs*).

revenue has been largely unfulfilled, the use of a portal to unite the web communications and thinking within a large organisation has begun to be seen as both money-saving and labour-saving. Some corporate analysts have predicted that corporate intranet web portal spending will be one of the top five areas for growth in the Internet technologies sector during the first decade of the 21st century. [1]

Corporate intranet portals are now beginning to adapt to provide access to a company's business partners, suppliers and customers as well as employees (sometimes referred to as a corporate extranet or corporate portal). These portals aim to provide a virtual workspace for each individual using them, providing the individual with access to all of the information, business applications and services needed to perform their jobs. [4]

1.2 Motivation

A key component of corporate extranet technology is often a content and document management system, providing services to support the full life cycle of document creation and providing mechanisms for authoring, approval, version control, scheduled publishing, indexing and searching. [1]

In particular the indexing and searching aspect of a document management system is a problem which is traditionally tackled in a similar way to internet searching, i.e. via a keyword search facility. Although this is an effective approach for most applications, the problems are not entirely similar, and some of the information harnessed in internet searching is unavailable in intranet searching and vice-versa. An approach tailored specifically to searching a document management system rather than one targeted at generic documents spread across one or many websites could potentially provide a more effective solution.

This project was proposed by a company specialising in web technology and knowledge management, EDF [10], in response to a need for a novel method of organising documents on medium to large intranets.

1.3 Project Goals

This project attempts to create a novel system for indexing, searching and browsing of an intranet document repository for use as part of a corporate extranet. The system should provide the following features:

- **Hierarchical structure**: The system should organise the document repository as a hierarchy by grouping related documents together.

- **Browsable**: The primary means for retrieving documents should be browsing through the document hierarchy.
- **Automated**: The system should organise documents automatically with a minimum of training.
- **Personalisable**: Ideally, the system should support some degree of personalisation for individual users.

According to EDF [10], the intended size of document repositories is expected to be between 3,000 and 30,000 documents.

1.4 Report Structure

Chapter 2	provides an overview of the technologies involved and the current state of the art.
Chapter 3	contains a high-level overview of the chosen design.
Chapter 4	contains a detailed view of the chosen design.
Chapter 5	details the initial testing carried out on the system and proposes further testing required.
Chapter 6	evaluates the performance of the system.
Chapter 7	summarises the achievements of the project, identifies limitations and possible further work.

2 Background

2.1 Introduction

Traditional *Data Mining* techniques operate on structured data such as corporate databases; this has been an active area of research for many years. More recently with the advent of the World Wide Web, a rapidly growing repository of *unstructured* data (in the form of text documents) has become available.

Information Retrieval is the science of searching for information in documents. Research began in the 1980s in response to a need for automatic methods of locating documents in large collections of texts. The commercial importance of this area grew massively following the advent of the World Wide Web in 1991 and subsequent exponential growth in the number of web pages.

Text Mining is the science of extracting *novel, interesting* and *non-trivial* information from text. It is a much younger field than both information retrieval and data mining, but is believed to have high commercial potential value, particularly compared to data mining due to the fact that most information (over 80%) is stored as text, and this area is currently largely unexploited as shown in Figure 2. [30]

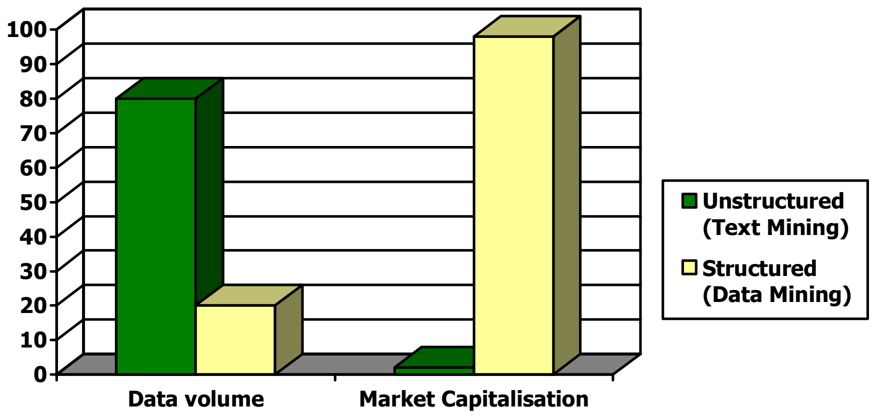


Figure 2 The business opportunity in text mining [34]

The commercial importance of text mining is further increasing due to the exponential growth of the number of information sources available on the web and the recent interest in e-commerce. [7]

Text Mining is an interdisciplinary research area and attracts interest from a number of research areas such as Databases, Information Retrieval and Natural Language Processing.

	Search *(Goal-oriented)*	***Discover*** *(Opportunistic)*
Structured Data	Data Retrieval	Data Mining
Unstructured Data	Information Retrieval	Text Mining

Figure 3 "Search" versus "Discover" [9]

Although Text Mining and Information Retrieval operate on the same type of data they are clearly distinguished by their methodology. Information Retrieval is *goal-oriented* and aims to locate as many relevant documents and as few non-relevant documents as possible in response to a *query*, whereas Text Mining is opportunistic and aims to discover new information in a dataset. These approaches can be compared to their counterparts in the more established topic of Data Mining, as shown in Figure 3.

This chapter presents an overview of the key technologies, concepts and existing research related to the project:

- Firstly an overview of Information Retrieval techniques is presented.
- Secondly an overview of Text Mining is presented, building upon the concepts of Information Retrieval and supervised classification algorithms are discussed in detail.
- Finally an overview of protocols and technologies related to the markup and delivery of internet and intranet documents is presented.

2.2 Overview of Information Retrieval

2.2.1 Introduction

The purpose of a traditional automatic information retrieval strategy is to retrieve all relevant documents whilst retrieving as few non-relevant documents as possible. This is illustrated in Figure 4.

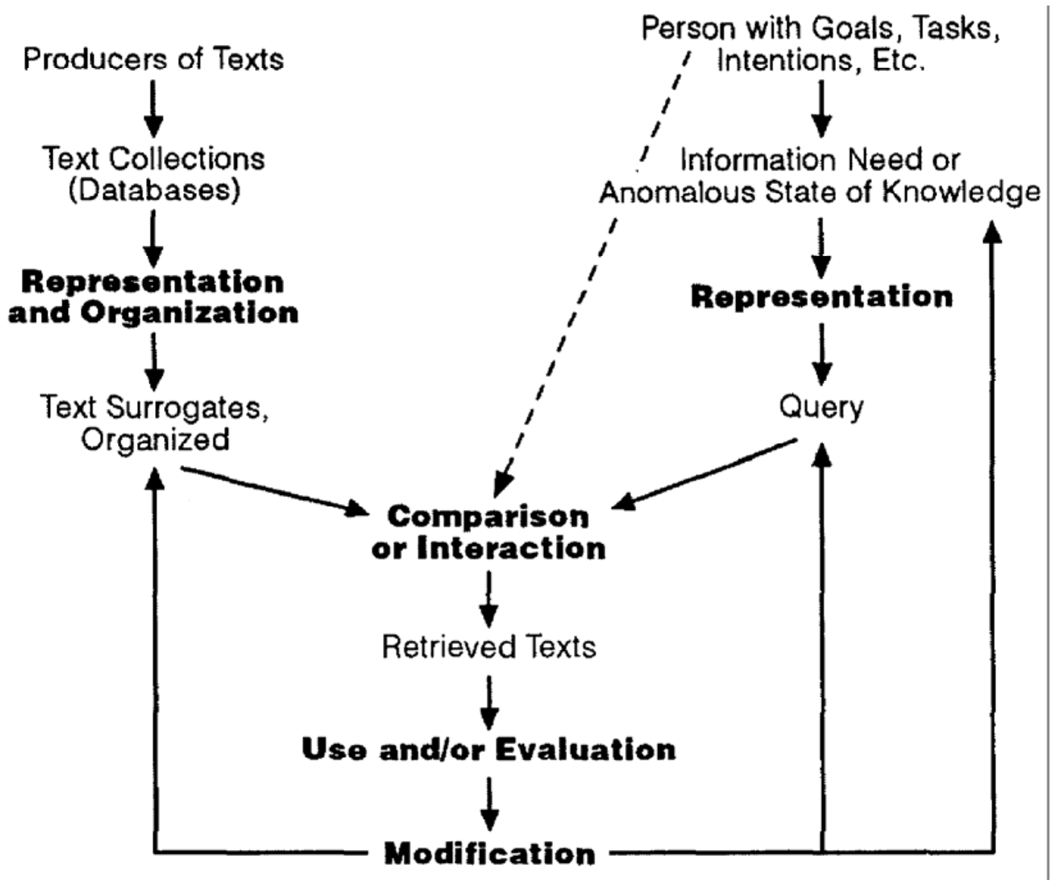

Producers of Texts
Text Collections (Databases)
Representation and Organization
Text Surrogates, Organized
Person with Goals, Tasks, Intentions, Etc.
Information Need or Anomalous State of Knowledge
Representation
Query
Comparison or Interaction
Retrieved Texts
Use and/or Evaluation
Modification

Figure 4 A general model of information retrieval [5]

The left-hand side of the diagram represents the process of turning texts into a form which is amenable to automatic processing, or *text surrogate*, consisting of index *terms*, *keywords* or *descriptors*.

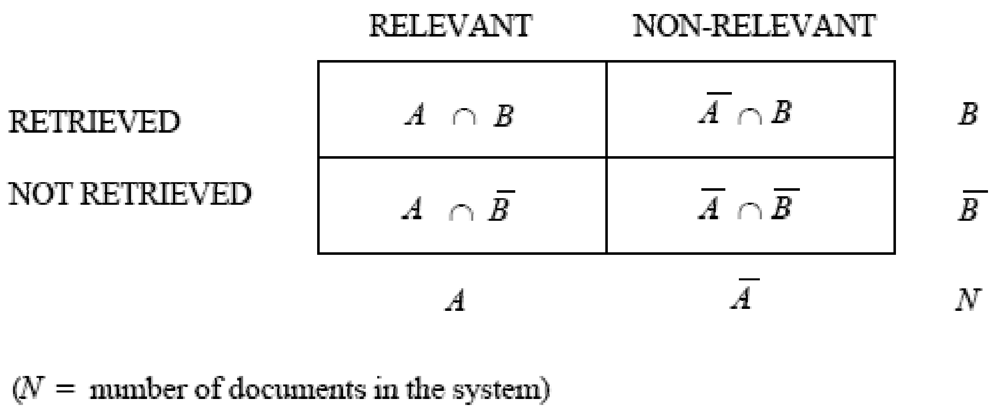
The right-hand side of the diagram represents the processing of a *query* arising due to the user's anomalous state of knowledge or information need. The query is then turned into a representation the system can understand.

The query is then compared to the collection of text surrogates and the texts thought to be relevant are returned to the user, who evaluates the texts by comparing them with the information the user expected to retrieve. This then often leads to modification of the query or possibly the information need or some of the surrogates. Effectiveness of the system is determined by the extent to which modification of the query is required.

2.2.2 Measuring Effectiveness

2.2.2.1 Precision and Recall

The results which are retrieved in response to a query can be split into two sets, those relevant to the user's query and those not relevant. The results not retrieved in response to the query can also be split similarly as shown in Figure 5.



	RELEVANT	NON-RELEVANT	
RETRIEVED	$A \cap B$	$\bar{A} \cap B$	B
NOT RETRIEVED	$A \cap \bar{B}$	$\bar{A} \cap \bar{B}$	$\bar{B}$
	A	$\bar{A}$	N

(N = number of documents in the system)

Figure 5 The 'contingency' table [6]

From this table, two of the most common measures of effectiveness can be derived. [6]

Recall is defined as the ratio of relevant documents retrieved for a given query over the number of relevant documents known to the system:

$$\text{Recall} = \frac{|A \cap B|}{|A|} \tag{1}$$

Precision is the ratio of relevant documents retrieved over the total number of documents retrieved:

$$\text{Precision} = \frac{|A \cap B|}{|B|} \tag{2}$$

Generally, a user wishes to achieve both high recall and high precision, but in practice both cannot be simultaneously optimised.

2.2.2.2 User-Oriented Measures

Precision and Recall fail to take into account individual users' different interpretations of what is relevant; as a result various user-oriented measures have been proposed:

$$\text{Coverage Ratio} = \frac{R_K}{U} \tag{3}$$

(Fraction of documents known to be relevant, which have been retrieved)

$$\text{Novelty Ratio} = \frac{R_U}{(R_U + R_K)} \tag{4}$$

(Fraction of relevant documents retrieved, which were previously unknown to the user)

$$\text{Relative Recall} = \frac{(R_U + R_K)}{U} \tag{5}$$

(Ratio of relevant documents found by the system over the number of documents the user expected to find)

where

U	=	Number of relevant documents known to the user
R_K	=	Number of relevant documents known to the user, which were retrieved
R_U	=	Number of relevant documents previously unknown to the user, which were retrieved

2.2.3 Document Representation

The information retrieval process, as shown above in Figure 4 requires text surrogates to be derived from the collection of documents to be searched, and organised in a way that facilitates queries to be processed quickly and efficiently. In addition to this, if the collection of documents is large, the amount of space required to store each surrogate may need to be minimised.

This requires a method of extracting from each document the *words* or *terms* which best summarise the meaning of the document. This creates a number of obstacles both due to the rather ambiguous nature of human language and also due to the problems of minimising storage space whilst optimising query processing accuracy and speed. Some of the most common solutions to these problems are presented below.

2.2.3.1 Tokenisation

The first step of processing is generally to divide the input text into *tokens* (or *unigrams*), or *terms*, where each is either a word or something else like a number or a punctuation mark. Often *tokenisers* split words where punctuation or white space occurs, however this also breaks up hyphenated word pairs such as "hard-disk".

To further complicate matters, not all white space indicates a word break. In the previous example "hard-disk" may also be written "hard disk". Similarly "database" and "data base" are both common written forms of the same concept. This is known as a *collocation* or *n-gram* (in this case a *bigram* since it contains two *unigrams*), where one or more words combined have a different meaning together than individually. [21]

2.2.3.2 Stoplist Elimination

An extremely common approach in most information retrieval systems is to eliminate very frequently occurring words since their ability to discriminate between documents is low (for example "the" is contained in an extremely high proportion of documents and generally provides little insight as to the meaning of the document).

This is achieved by maintaining a list of terms known as a *stoplist* or *negative dictionary*. This should take into account the type of documents the system is intended to process. These terms are stripped out of the document and discarded during processing.

2.2.3.3 Stemming

Stemming is the process of automatically *conflating* related terms, usually by reducing them to a common form. One of the most well known stemming techniques is *Porter's algorithm* [11]. For example Porter's algorithm would reduce the words 'engineer', 'engineers', 'engineered' and 'engineering' to the common root form 'engineer'.

This serves to reduce the dimensionality of the document, providing a more statistically accurate estimate for the number of occurrences of a given term. However this has the disadvantage that information is lost due to this dimensionality reduction. Another disadvantage is that the *stemmer* is a simple algorithm and can *overstem* words, for example words such as 'generate', 'general', 'generic' and 'generous' are all stemmed to the same common form 'gener'.

2.2.3.4 WordNet

WordNet [12] is a lexical database of the English language. It contains entries for over 155,000 words, providing data on *polysemy* (words with more than one meaning) and defines *synsets* of words representing a similar lexical concept. These synsets are organised into a *conceptual hierarchy* similar to a thesaurus and defines links between words according to a number of relations including:

- *Hyponyms* ("is a") e.g. cat is a hyponym of animal
- *Hypernyms* (opposite of a hyponym) e.g. animal is a hypernym of cat
- *Meronyms* ("part of") e.g. tail is a meronym of cat
- *Antonyms* (opposite) e.g. hot is an antonym of cold

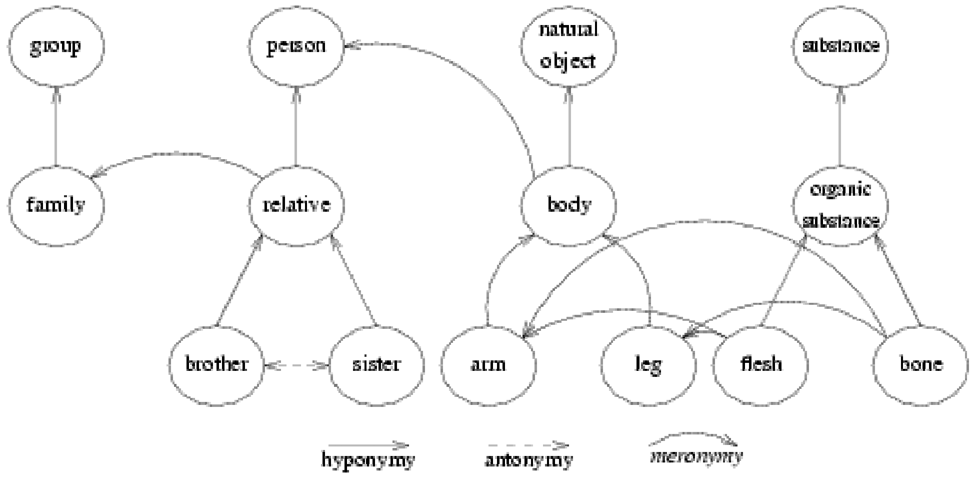


Figure 6 A WordNet conceptual hierarchy [42]

The use of WordNet provides a much more advanced solution to automated conflation through using data specific to each word rather than an algorithm such as Porter's algorithm which exploits patterns in English words which are only true in the majority of cases.

Additionally the use of WordNet provides for more advanced functionality than conflation, for example the use of context to disambiguate polysemy or the use of WordNet's conceptual hierarchy to either add hypernyms of terms to the surrogate or replace terms in the surrogate with their hypernym. This enables a specific query to retrieve documents relating to a similar but more general concept and vice versa. [13][14]

2.2.3.5 Term Weighting

In the *"bag of words"* model, the representation of the document contains no information about the relative positioning of the words, only how many times they occur in the document (similar to a set except that items contained in a bag can occur more than once). Despite this loss of data, a good approximation of the meaning of the document is still retained, and both the space to store documents and the processing power required for queries is reduced dramatically.

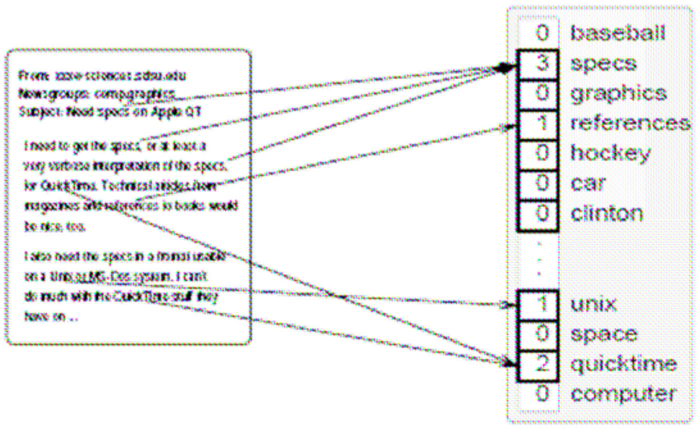


Figure 7 The "bag of words" model [31]

Hence, the document can be represented as a *term vector* of the form [15]:

$$D = (n_1, n_2, \ldots, n_i) \tag{6}$$

where each n_i is equal to the number of times the corresponding term (or word) appears in the original document.

In this model, the importance of a term in a document is assumed to be proportional to how many times it appears. Clearly this model will be unfairly biased towards large documents which contain more terms and therefore would have a higher importance attached to their terms than a small document.

The above model can be refined to eliminate this bias by normalising the terms in equation (6) into the range 0..1 (0 being a term that never occurs in the document, 1 being a term that accounts for 100% of the document):

$$D = (tf_1, tf_2, \ldots, tf_i)$$
where (7)
$$tf_x = \frac{n_x}{\sum_{k=1}^{i} n_k}$$

However, *term frequency* factors alone cannot ensure acceptable retrieval performance since when high frequency terms are not concentrated in a few particular documents but occur frequently in the whole collection (for example words such as "the" or "a"), all documents tend to be retrieved and this affects the query precision. *Term discrimination* conditions suggest that the best terms for document content identification are those able to distinguish certain individual documents from the remainder of the collection. [15]

This mathematical basis for this theory is Zipf's law [18]. Zipf proposed that the product of the frequency of use of words and the rank order is approximately constant. Zipf verified his law on American Newspaper English. Luhn, in his work on automatic text summarisation [19], used Zipf's law as a null hypothesis to enable him to specify two cut-offs to exclude non-significant words that were either too rare or too commonly used to contribute significantly to the meaning of the article. He then assumed that the *resolving power*, i.e. the ability of words to discriminate content reached a peak half way between the two cut-offs and fell in either direction to approximately zero at the cut-off points as shown below in Figure 8.

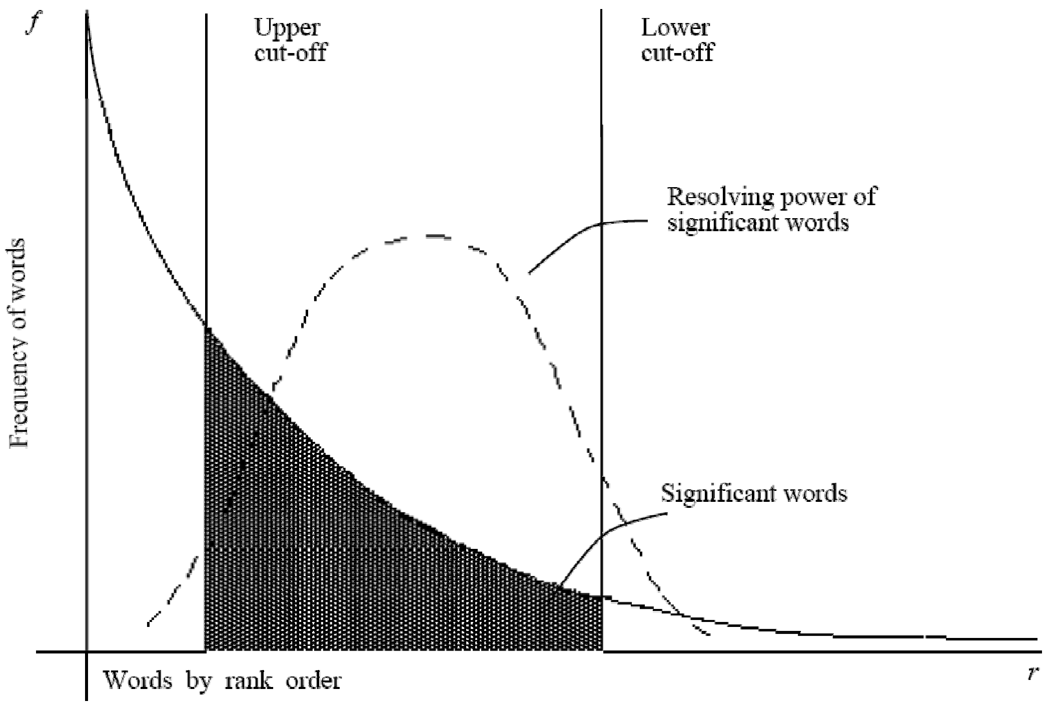


Figure 8 Luhn's Word-Frequency diagram [19] (Adapted in [6])

The most common approach to term discrimination is known as *TF-IDF* or *Term Frequency – Inverse Document Frequency*. In this approach, the elements of equation (7) are replaced with term frequency multiplied by an *inverse document* *frequency* which is inversely proportional to the prevalence of the term across the document collection as a whole. The most common variant of this is TF-LogIDF [15]:

$$D = (tf - idf_1, tf - idf_2, \ldots, tf - idf_i)$$

$$\text{where} \tag{8}$$

$$tf - idf_x = tf_x \times \log\left(\frac{N}{df_x}\right)$$

where *N* is the total number of documents in the collection and df_x is the number of documents in which the term *x* occurs at least once.

Salton & Buckley [15] also propose a third term weighting factor for information retrieval systems using variable length *document vectors* (where documents with less unique words have a shorter vector) which attempts to compensate for documents with a large number of terms having an increased chance of term matches with queries and therefore a better chance of being retrieved than shorter ones.

2.2.4 Query Processing

After the text surrogates have been created from the document collection to be searched, the system is ready to accept queries. When a query is entered into the system it undergoes a similar transformation to that involved in creating a text surrogate from a document. This ensures that terms in the query are conflated with equivalent terms in the text surrogate collection.

The transformed query is then compared to the collection of text surrogates and matching results are returned, usually in order of relevance to the query. However in a large collection of documents with a large collective vocabulary of terms it is difficult to optimise the query to run both quickly and accurately (with both high precision and recall). A number of common query processing strategies are discussed below:

2.2.4.1 Boolean Model

The Boolean model is the simplest query processing model. Query terms are connected by 'AND', 'OR' and 'NOT'. The query only returns exact matches where all terms in the query are in the document.

Ordering of returned results is achieved by adding up the weights of the terms used in the query for each document.

2.2.4.2 Vector Space Model

The *Vector Space Model* uses the principle that the document vector in equation (8) can be used to represent documents as vectors in a multidimensional space. Similarly a query can also be represented as a vector in the same multidimensional space.

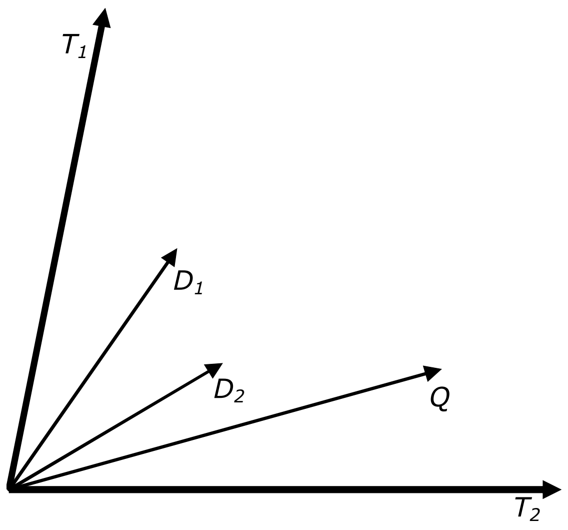


Figure 9 Two-dimensional vector space with terms as basis

The diagram above shows a simple two-dimensional space. The terms (T_1, T_2) form the axes; documents (D_1, D_2) are represented in this space as vectors relative to the axes. A query, Q, is represented similarly.

Under this model, the similarity of documents and queries can be compared using the cosine similarity measure:

$$Sim(\vec{i},\vec{k}) = \frac{i \cdot k}{|i| \times |k|} = \frac{|i||k|\cos\theta}{|i| \times |k|} \tag{9}$$

Hence documents are compared based on the "best match" principle rather than the "exact match" principle used in the Boolean model. However this lacks the expressiveness of the Boolean model's operators (AND/OR/NOT).

Wong et al. [17] propose an extension to the vector space model which allows Boolean term correlations to be imported into the retrieval process whilst still maintaining the "best match" approach.

2.2.4.3 Relevance Feedback

Relevance Feedback is a common process in Information Retrieval where a user refines a query by marking returned documents as relevant or non-relevant. The system then computes a better representation of the user's need based upon the original query and the relevance feedback provided by the user. This process is repeated until the user is satisfied with the results. One of the best known relevance feedback algorithms is the *Rocchio Algorithm* [33].

2.3 Overview of Text Mining

2.3.1 Introduction

Text Mining is the process of extracting novel, meaningful and useful data, or *knowledge,* from unstructured text. It is similar to the process of Data Mining, but is a more interdisciplinary field drawing also on linguistics, information retrieval, statistics and computational linguistics to solve problems caused by the unstructured nature of text.

2.3.1.1 The Text Mining Process

The steps involved in the Text Mining process is summarised below in Figure 10.

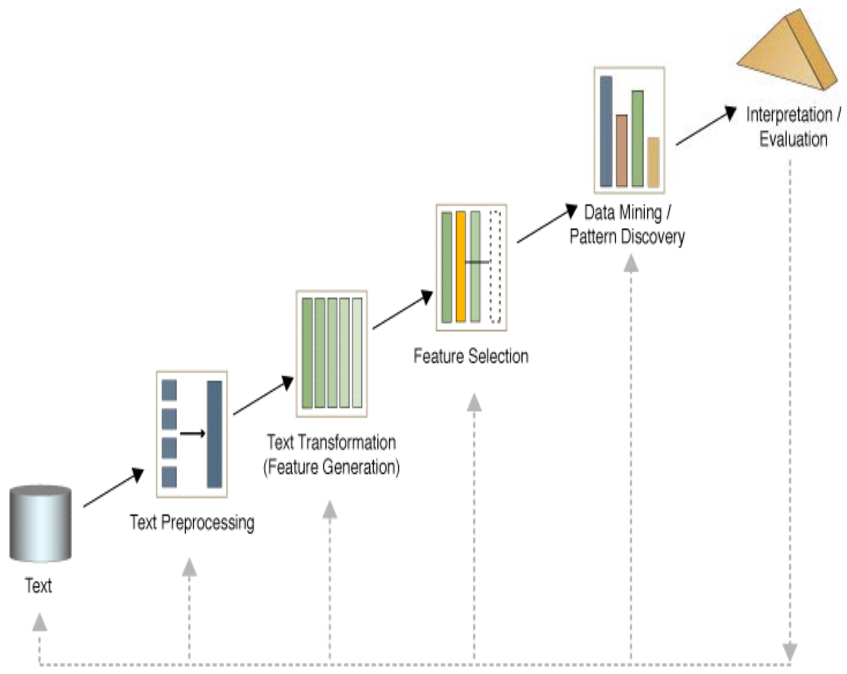

Text
Text Preprocessing
Text Transformation (Feature Generation)
Feature Selection
Data Mining / Pattern Discovery
Interpretation / Evaluation

Figure 10 The Text Mining Process [9]

The initial steps in the process convert a document from raw text into a consistent internal representation, and are exactly the same as the steps used in Information Retrieval to produce a Text Surrogate:

- **Text Pre-Processing** involves syntactic and semantic analysis on the text, for example stemming or the use of WordNet.
- **Text Transformation** converts the pre-processed document text into a consistent internal representation so that documents can be compared to each other. In general the internal representation utilises the "bag of words" model.
- **Feature Selection** performs statistical analysis of the transformed text and selects a subset of the terms available (often by selecting a percentage of the highest TF-IDF weightings) in order to reduce processing power and to reduce problems of estimation when the number of terms present is much larger than the number of documents.

The raw text has now been transformed into a consistent internal representation, and the **Data Mining/Pattern Discovery** process can take place. This is the major component of the Text Mining process, taking pre-processed data and turning it into knowledge. In Text Mining this generally takes the form of a *learning classifier*.

2.3.1.2 Web Mining

The application of Text Mining specifically to the domain of Web Pages is known as *Web Mining*. Additional information available specifically in this domain allows knowledge discovery through other means than those available in standard text mining (due to the availability of additional data such as usage patterns and link structure); it also can provide additional information for use in classification.

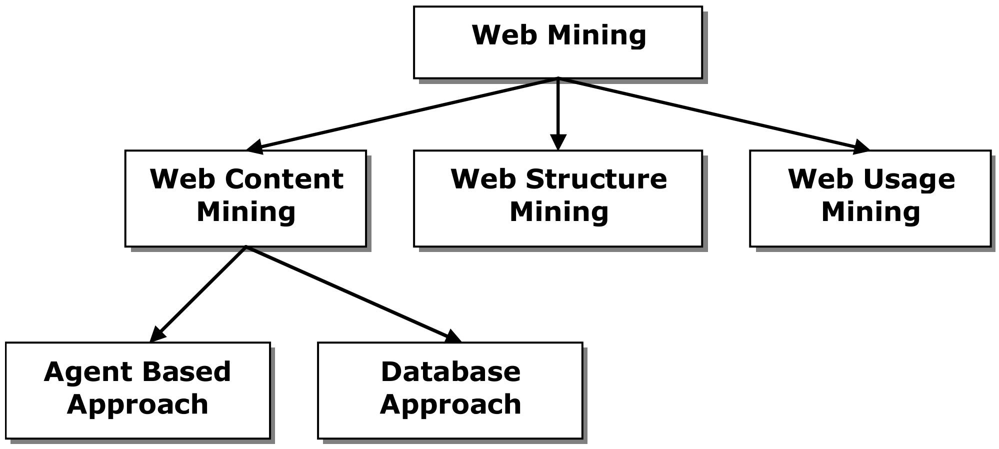


Figure 11 Taxonomy of Web Mining (Adapted from [20])

There are 3 major categories of Web Mining:

- **Web Content Mining** utilises the content of individual pages:
    - An **Agent Based Approach** uses an intelligent agent which is individual to the user and performs web mining on the user's behalf.
    - A **Database Approach** is generally a more centralised approach, for example a search engine using web content mining to group related web pages together.
- **Web Structure Mining** utilises the structure formed by hyperlinks between documents.
- **Web Usage Mining** utilises user access patterns from web server access logs.

A prominent example of using Web Structure mining to boost Text Mining is Google [23]. Google uses a system known as *PageRank* [24] as one of the main factors in ordering returned results. In essence, PageRank utilises the principle that if a page is important or useful people will create links to this page, hence pages with the most links leading to them are the most useful pages. A page's PageRank is proportional to the number of links pointing at that page. Also the text of links is taken into account and if a number of links with the same text point to the same page, this page will appear near the top of searches using the same keywords as the link text.

It is interesting to note that a famous vulnerability of this algorithm is that a large number of sites collaborating and creating links to a particular site can influence the ordering of results, known as a "*Google Bomb*" or "*Google Wash"*. For example in February 2005 this technique was used to place George W. Bush's biography page as the first result returned when searching for the keywords "miserable failure". [25]

Web Mining will not be discussed further since the intention of this project is to concentrate on a repository of Intranet documents which may not necessarily have links between each other and may not have access logs available.

2.3.1.3 Classification Algorithms

As mentioned previously, the major component of the Text Mining process is generally a *learning classifier*. This takes pre-processed data and turns it into knowledge. There are two types of *learning classifier:*

- A **Supervised Learning Classifier** (referred to in the remainder of this document as a *classification algorithm*) automatically places new documents into groups or *classes* based upon statistical characteristics of the new document and a *training set* of previously labelled documents.

- An **Unsupervised Learning Classifier** (referred to in the remainder of this document as a *clustering algorithm*) automatically places documents into groups or *classes* '*a priori'* (without any prior training data).

Another classification technique is also available, a **fixed classifier**. This is normally rule based and does not have the capacity to learn and therefore is "unintelligent". This makes it unable to discover knowledge per se, and so is not generally used in Text Mining; however it is included in the taxonomy presented in Figure 12 below for completeness.

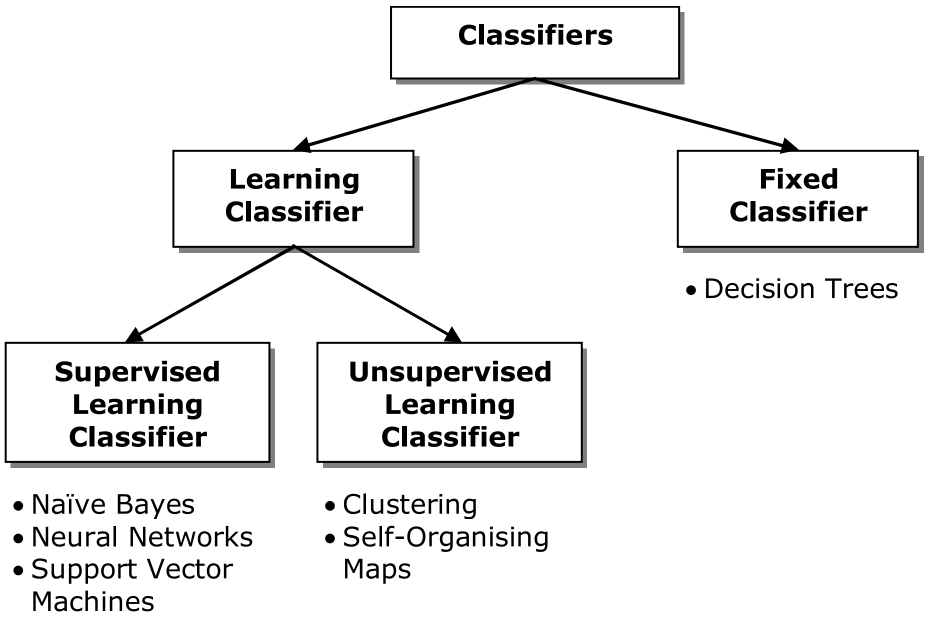


Figure 12 Taxonomy of Classification Techniques (Examples from [22])

2.3.2 Unsupervised Classifiers

The majority of clustering or unsupervised classification algorithms operate by attempting to partition a dataset into *clusters* such that all documents in a cluster share some common trait, i.e. inter-cluster similarity is minimised and intra-cluster similarity is maximised.

Clustering algorithms have a major advantage over supervised classifiers in that they have no preconceptions whatsoever about data, and therefore are able to identify patterns that training would inhibit a supervised classifier from identifying. However this is also a disadvantage in that by definition clustering algorithms do not allow training and so it is not possible (without modification to the algorithm) for the user to influence the process and personalise the output.

Because this limitation is contrary to the goals of this project (personalisable categorisation of documents), unsupervised classifiers are discussed only in brief to provide a comparison to supervised classifiers.

2.3.2.1 k-Nearest Neighbour

The *k-nearest neighbour algorithm* utilises the vector space model (As defined in section 2.2.4). Clusters are based upon *centroids* (which act as the "centre of gravity" for a cluster).

The algorithm operates as follows [27]:

- Let d be the distance measure between instances.
- Select k random instances $\{s_1, s_2, \ldots s_k\}$ as seeds.
- Until clustering converges or other stopping criterion:
    - For each instance x_i:
        - Assign x_i to the cluster c_j such that $d(x_i, s_j)$ is minimal.
    - (*Update the seeds to the centroid of each cluster*)
    - For each cluster c_j
        - $\mathrm{s}_\mathrm{j} = \vec{\mu}(\mathrm{c}) = \frac{1}{|c|}\sum_{\vec{x} \in c} \vec{x}$

2.3.2.2 Hierarchical

Hierarchical clustering algorithms are grouped into two broad categories:

- *Agglomerative* or *bottom-up hierarchical clustering algorithms* start with all instances in separate clusters and repeatedly join the two most similar clusters until there is only one cluster.
- *Divisive* or *top-down hierarchical clustering algorithms* start with all instances in the same cluster, and divide the clusters until each instance forms a cluster on its own.

Hierarchical clustering algorithms have the advantage that the number of clusters does not need to be known in advance, and termination is guaranteed (*k-NN* is not guaranteed to converge).

In the case of agglomerative clustering, the definition of "most similar" affects the type of clusters that are produced: [27]

- *"Centre of gravity"* defines similarity as the distance between centroids.

- *Average-link* defines similarity as the average *cosine similarity* between pairs of elements.
- *Single-link* defines similarity as the *cosine similarity* of the most cosine similar (Produces long, thin clusters).
- *Complete-link* defines similarity as the *cosine similarity* of the least cosine similar (Produces tight, spherical clusters).

The output of hierarchical clustering can be represented as a dendogram as shown in Figure 13 (below).

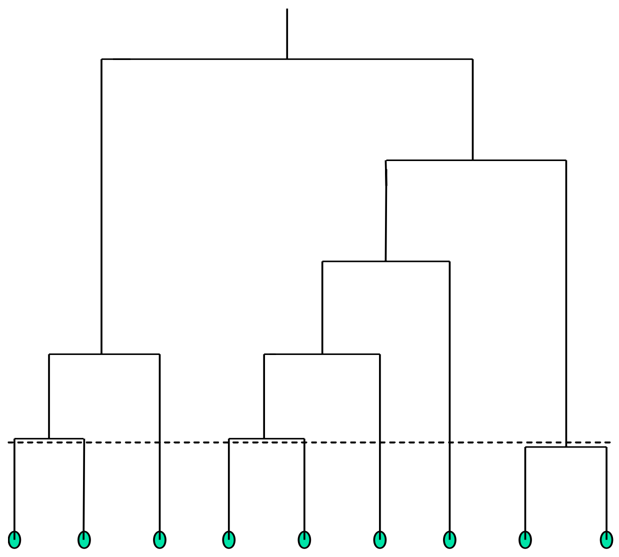


Figure 13 A dendogram [27]

This is then cut at the desired level (as indicated by the dotted line) to get the final clusters; each connected component at the level of the cut forms a cluster.

2.3.3 Supervised Classifiers

Supervised classification algorithms generally operate through finding the pre-existing class most similar to a new example. A classifier must be trained before it can be used; a priori a classifier is unable to operate.

Formally stated, given training data $\{(x_1, y), \ldots, (x_n, y)\}$ where x_n are *features* or terms in the document and y is the class assigned to the training instance, a classifier $h: X \rightarrow Y$ maps an object $x \in X$ to a classification label $y \in Y$. [26]

2.3.3.1 k-Nearest Neighbour

Introduction

The *k-nearest neighbour algorithm* is a classification variant of the k-nearest neighbour clustering algorithm. It attempts to find the *k* nearest examples in the training set to a new instance. The new instance is assigned to whichever class has the highest number of documents in the *k* nearest examples. Training occurs by simply adding examples to the relevant class in the training set.

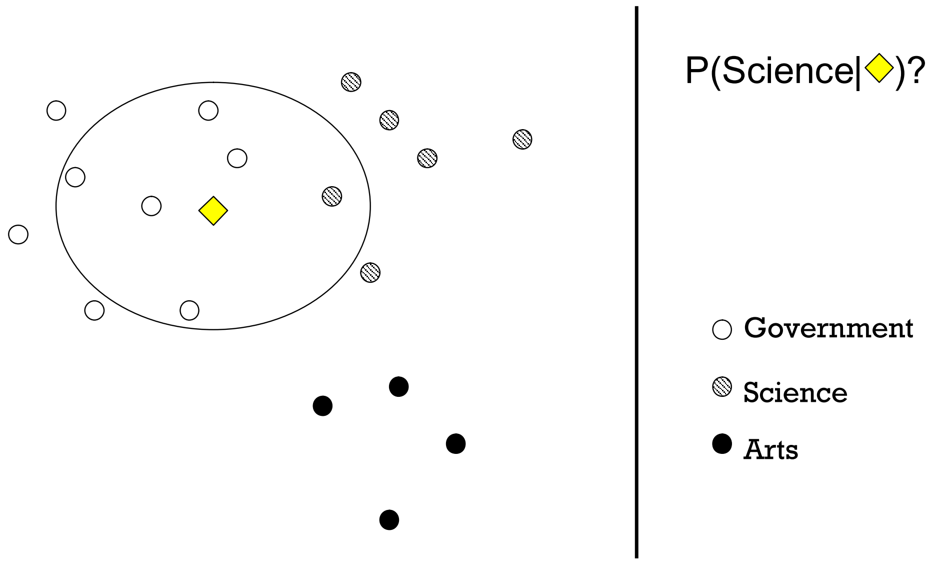


Figure 14 6-Nearest Neighbour Classification [27]

Time Complexity

Training time complexity is of order $\vartheta(1)$ since it simply requires adding an instance to the training set.

Searching for nearest neighbours naively would require the entire collection to be searched, however by utilising a standard vector space *inverted index* (Which optimises retrieval of all documents containing a given term); testing time complexity is of order $\vartheta(B|V_t|)$ where B is the average number of training documents in which a word in the test document occurs and $|V_t|$ is the average vocabulary size for a test document. Typically $B << |D|$. [27]

2.3.3.2 Relevance Feedback (Rocchio Text Classifier)

Introduction

The *rocchio algorithm* incorporates *relevance feedback* into the vector space model (both defined in section 2.2.4). Based upon the training set, a *prototype vector* for each category is computed. New instances are assigned to the category with the closest prototype vector based cosine similarity. Incorrectly classified instances are manually classified and added to the training set. [6]

Figure 15 (below) shows an example of Rocchio Text Categorisation. The long-dashed lines represent instances from one category with the longer bold line being the prototype vector. Similarly the dotted lines represent instances from another category. In the middle the solid line represents a new instance being classified, the two arcs show the cosine similarity between it and the two prototype vectors.

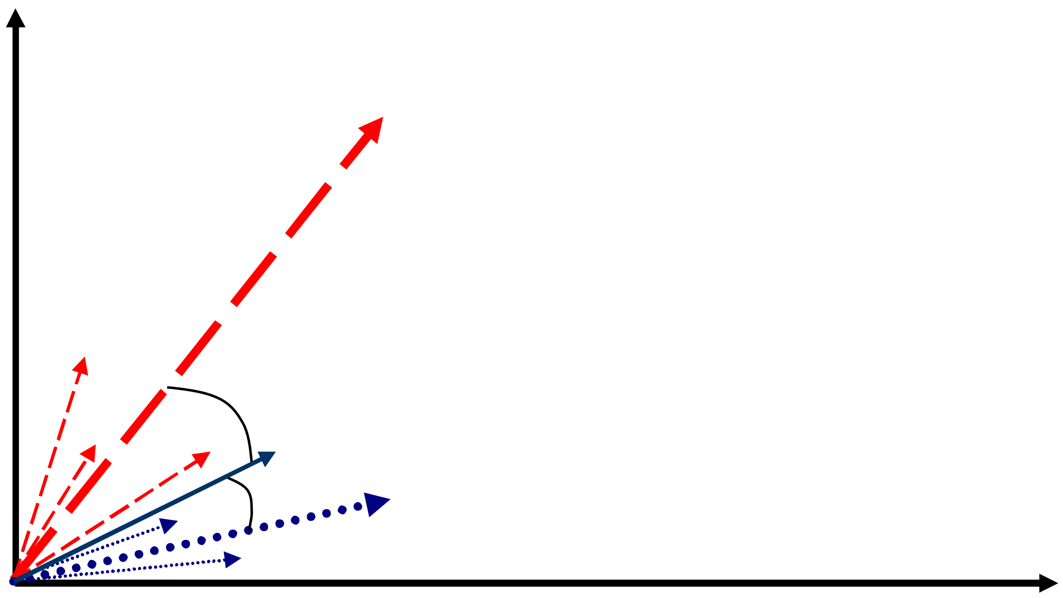


Figure 15 Illustration of Rocchio Text Categorisation [27]

However with a *polymorphic* or *disjunctive category*, where documents are not all clearly grouped together in vector space, a single prototype vector cannot represent the category accurately as shown by the long-dashed lines in Figure 16 (below).

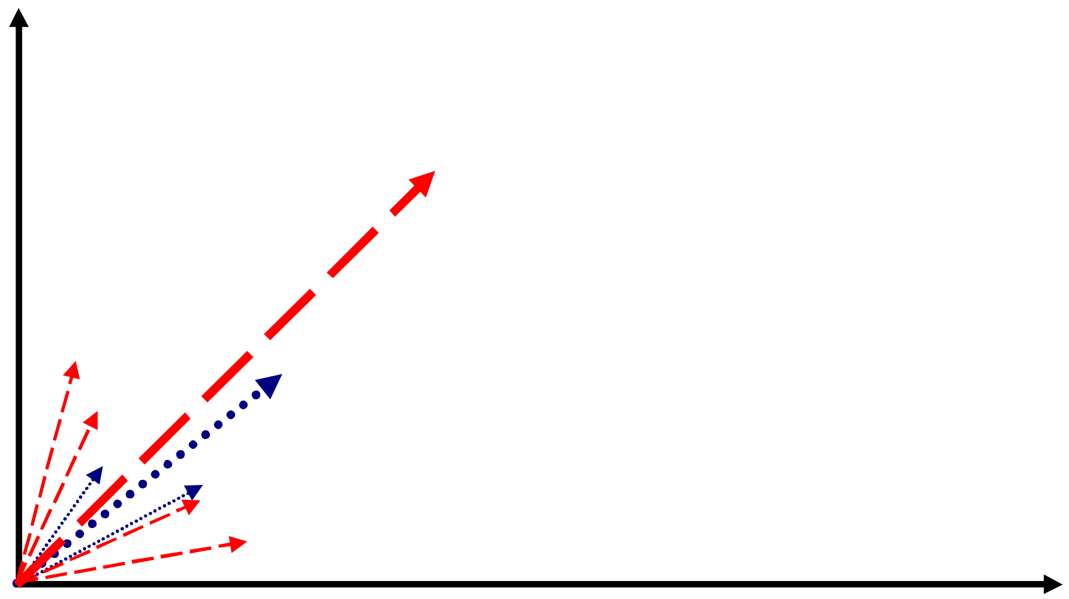


Figure 16 Illustration of a Disjunctive Category under Rocchio [27]

In this example a nearest neighbour classifier would be able to cope better.

Time Complexity

The Rocchio algorithm decreases testing time compared to k-nearest neighbour if the number of classes is much less than the number of documents, since only one comparison to the prototype vector is required per class during testing.

Training time complexity is of order $\vartheta(|D|(L_D + |V_D|)) = \vartheta(|D|L_D)$ where $|D|$ is the number of documents in the system, L_D is the average length of a document in *D* and V_D is the average vocabulary size for a document in *D*. [27]

Testing time complexity is of order $\vartheta(L_t + |C||V_t|)$ where L_t is the average length of a test document, $|C|$ is the number of classes and $|V_t|$ is the average vocabulary size for a test document. [27]

2.3.3.3 The Naïve-Bayes Classifier

Introduction

A *naïve Bayes classifier* is a simple classifier based on a probability model that incorporates strong independence assumptions; namely that the presence or absence of a given term in a document is completely independent of the presence of absence of any other (non-identical) term.

This assumption of independence is clearly false, and hence the classifier is deliberately naïve. Despite this over-simplified assumption, the naïve Bayes classifier works much better than would be expected from its simplistic design.

Probability Model

This section describes the mathematical foundation for the probability model underlying the naïve Bayes classifier. It is adapted from [28].

Bayes' theorem relates the conditional and marginal probabilities of two independent random variables:

$$\begin{aligned} P(A \mid B)P(B) &= P(B \mid A)P(A) \\ \therefore P(A \mid B) &= \frac{P(B \mid A)P(A)}{P(B)} \end{aligned} \tag{10}$$

where $P(A)$ is the *prior* or *marginal probability* of *A,* $P(A \mid B)$ is the *posterior probability* of *A* given *B*, $P(B \mid A)$ for a specific value of *B* is the *likelihood function* for *A* given *B* and $P(B)$ is the *prior* or *marginal probability* of *B* and acts as the *normalising constant*.

The probability model for a classifier is a conditional model:

$$P(C \mid x_1, \ldots, x_n) \tag{11}$$

over a dependent class variable C with a small number of outcomes or classes which are conditional on several feature (or term) variables x_n. Using Bayes theorem this can be written:

$$P(C \mid x_1, \ldots, x_n) = \frac{P(C)P(x_1, \ldots, x_n \mid C)}{P(x_1, \ldots, x_n)} \tag{12}$$

The denominator is constant and can be estimated from the frequency of terms in the training data. The numerator is equivalent to the joint probability model:

$$P(C)P(x_1, \ldots, x_n \mid C) = P(C, x_1, \ldots, x_n) \tag{13}$$

Using repeated applications of the definition of conditional probability this can be rewritten to:

$$\begin{aligned} P(C, x_1, \ldots, x_n) &= P(C)P(x_1, \ldots, x_n \mid C) \\ &= P(C)P(x_1 \mid C)P(x_2 \ldots, x_n \mid C, x_1) \\ &= P(C)P(x_1 \mid C)P(x_2 \mid C, x_1)P(x_3 \mid C, x_1, x_2)\ldots \end{aligned} \tag{14}$$

Because the model is "naïve", it is assumed that each feature x_n is conditionally independent from other features. Therefore:

$$P(x_i \mid C, x_j) = P(x_i \mid C) \tag{15}$$

Hence the joint probability model in equation (13) can be rewritten as follows:

$$\begin{aligned} P(C, x_1, \ldots, x_n) &= P(C)(x_1 \mid C)(x_2 \mid C)\ldots \\ &= P(C)\prod_{i=1}^{n} P(x_i \mid C) \end{aligned} \tag{16}$$

Therefore, equation (11) can be rewritten as follows:

$$\begin{aligned} P(C \mid x_1, \ldots, x_n) &= \frac{P(C)P(x_1, \ldots, x_n \mid C)}{P(x_1, \ldots, x_n)} \\ &= \frac{P(C)\prod_{i=1}^{n} P(x_i \mid C)}{P(x_1, \ldots, x_n)} \end{aligned} \tag{17}$$

Note that the denominator is constant as previously stated.

Decision Rule

In order to turn the above into a classifier, the model above is combined with a decision rule. The most common decision rule is the *maximum a posteriori,* which picks the most probable hypothesis.

Because this looks at relative values rather than absolute values, the constant denominator in equation (17) can be omitted:

$$\text{classify}(x_1, \ldots, x_n) = \text{argmax}_c \, P(C = c)\prod_{i=1}^{n} P(X_i = x_i \mid C = c) \tag{18}$$

This classifier can be easily adapted for text classification by using maximum likelihood estimates of $P(C = c)$ and $P(X_i = x_i \mid C = c)$ over the training set:

$$\hat{P}(C=c)=\frac{N(C=c)}{N}$$
$$\hat{P}(X_i=x_i \mid C=c)=\frac{1+N(X_i=x_i,C=c)}{k+N(C=c)} \qquad (19)$$

where $N(C=c)$ is the number of documents assigned to class *C* in the training set, *N* is the total number of documents in the training set and *k* is the number of features (i.e. the maximum value of *i*).

1 and *k* in the bottom equation act as smoothing constants to prevent the predicted probability from reaching zero since this would cause equation (17) to be equal to zero, therefore preventing any class missing just one feature from being selected.

Note that the maximum a posteriori decision rule makes the classifier robust to serious deficiencies of its underlying naïve probability model – probabilities do not have to be estimated correctly; the classifier's decision remains correct as long as the correct class is more probable than any other.

Time Complexity

Training time complexity is of order $\vartheta(|D|L_D+|C||V|)$ where $|D|$ is the number of documents in the system, L_D is the average length of a document in *D,* $|C|$ is the number of classes and $|V|$ is the vocabulary size. [27]

Testing time complexity is of order $\vartheta(|C|L_t)$ where L_t is the average length of a test document and $|C|$ is the number of classes. [27]

2.3.3.4 The PrTFIDF Classifier

Joachims [31] presents a probabilistic analysis of the Rocchio Text Classifier (described above; also known as a *TFIDF Classifier*) which makes the implicit assumption of the Rocchio classifier as explicit as for the Naïve Bayes Classifier. Joachims identifies a number of problems which lead to "comparatively low classification accuracy", and proposes a probabilistic version of the Rocchio Classifier, called *PrTFIDF*, which eliminates the inefficient parameter tuning and design choices of the Rocchio Classifier.

Probability Model

Naïve-Bayes computes an estimate of $P(C_j \mid d')$ (the probability that document d' is in class C_j) using equation (17) and by making a simplifying assumption of independence.

The PrTFIDF Classifier uses a different means of estimating $P(C_j \mid d')$, inspired by the "*retrieval with probabilistic indexing*" approach [41]. A set of descriptors X is used to represent the content of documents. A descriptor x is assigned to a document d with a certain probability $P(x \mid d)$.

Using the theorem of total probability this can be written as:

$$P(C_j \mid d) = \sum_{x \in X} P(C_j \mid x, d) P(x \mid d) \tag{20}$$

This can be rewritten using Bayes' theorem:

$$P(C_j \mid d) = \sum_{x \in X} \frac{P(C_j \mid x)}{P(d \mid x)} P(C_j \mid x) P(x \mid d) \tag{21}$$

To make this tractable, the simplifying assumption that $P(d \mid C_j, x) = P(d \mid x)$ is made:

$$P(C_j \mid d) \approx \sum_{x \in X} P(C_j \mid x) P(x \mid d) \tag{22}$$

This implies that $P(C_j \mid d)$ is approximated by the expectation of $P(C_j \mid x)$, where x consists of a sequence of n words drawn randomly from document d. For $n = |d|$, $P(C_j \mid d)$ equals $P(C_j \mid x)$, but with decreasing n this simplifying assumption (like the independence assumption in the naïve-Bayes classifier) will be violated in practice.

In the simplest case, $n = 1$, equation (22) can be written as:

$$P(C_j \mid d) \approx \sum_{w \in F} P(C_j \mid w) P(w \mid d) \tag{23}$$

The two probabilities in this equation can be estimated as follows:

$$\hat{P}(w \mid d) = \frac{TF(w,d)}{\sum_{w' \in F} TF(w',d)} = \frac{TF(w,d)}{|d|} \qquad (24)$$

where $|d|$ denotes the number of words in document d and $TF(w,d)$ is the number of times a word w occurs in document d.

$P(C_j \mid w)$, the probability that C_j is the correct category of d given that we only know the randomly drawn word w from d, can be rewritten using Bayes' theorem:

$$P(C_j \mid w) = \frac{P(w \mid C_j)P(C_j)}{\sum_{C' \in C} P(w \mid C')P(C')} \qquad (25)$$

$P(C_j)$ can be estimated from the fraction of training documents that are assigned to class C_j:

$$\hat{P}(C_j) = \frac{|C_j|}{\sum_{C' \in C} |C'|} = \frac{|C_j|}{|D|} \qquad (26)$$

$P(w \mid C_j)$ can be estimated as:

$$\hat{P}(w \mid C_j) = \frac{1}{C_j} \sum_{d \in C_j} \hat{P}(w \mid d) \qquad (27)$$

Hence:

$$P(C_j \mid d') = \sum_{w \in F} \frac{P(w \mid C_j)P(C_j)}{\sum_{C' \in C} P(w \mid C')P(C')} \times P(w \mid d') \qquad (28)$$

Decision Rule

This can be turned into a classifier decision rule as follows:

$$H_{\mathrm{Pr}\,TFIDF}(d') = \arg\max_{C_j \in C} \sum_{w \in F} \frac{P(w \mid C_j)P(C_j)}{\sum_{C' \in C} P(w \mid C')P(C')} \times P(w \mid d') \qquad (29)$$

Joachims then goes on to prove the relationship between PrTFIDF and TFIDF.

Effectiveness

In testing the classifier showed performance improvements of up to 40% reduction in error rate on five out of six tasks.

2.3.3.5 Hierarchical PrTFIDF Classifier

Introduction

Peng and Choi [32] propose a hierarchical classification algorithm which utilises the hierarchical structure of categories to optimise both time complexity and accuracy.

Training

Initially the classifier starts with a predefined hierarchy of categories or classes, and training set of instances, each instance associated with one category in the hierarchy:

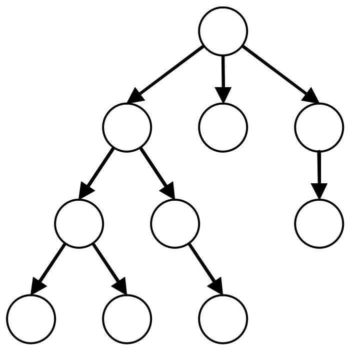


Figure 17 Category Hierarchy

Initially a *feature vector* is generated for each category (equivalent to the prototype vector in the Rocchio text classifier algorithm), by counting the number of occurrences of each feature *w* to form the term frequency $TF(w,C)$ and then normalising this value using the formula:

$$P(w \mid C) = \frac{1 + TF(w,C)}{|F| + \sum_{f \in F} TF(f,C)} \tag{30}$$

where F is the set of all features in the current category C, $|F|$ is the number of elements in set F and $TF(w,C)$ is the total number of appearances of a feature w in a category C.

The feature vectors are then propagated up the tree from the leaf nodes to the root using the following formula:

$$P(w \mid T) = \sum_{i=1}^{k} P(w \mid SubTree_i) P(SubTree_i \mid T) + P(w \mid N) P(N \mid T) \qquad (31)$$

The formula effectively takes a weighted average of the feature vector of the node *N*, and the feature vectors of it's *k* subtrees $\{SubTree_1, \ldots, SubTree_k\}$, where $P(w \mid SubTree_i)$ and $P(w \mid N)$ are calculated using equation (30). $P(SubTree_i \mid T)$ and $P(N \mid T)$ are calculated using equation (32) and serve to weight the average.

$$P(SubTree_i \mid T) = \begin{cases} 0 & \text{if leaf} \\ \dfrac{\sum_{i=1}^{k} \ln(1 + Ex(SubTree_i))}{\sum_{i=1}^{k} \ln(1 + Ex(SubTree_i)) + \ln(1 + Ex(Node))} & \text{otherwise} \end{cases}$$

$$P(N \mid T) = \begin{cases} 1 & \text{if leaf} \\ \dfrac{\ln(1 + Ex(Node))}{\sum_{i=1}^{k} \ln(1 + Ex(SubTree_i)) + \ln(1 + Ex(Node))} & \text{otherwise} \end{cases} \qquad (32)$$

where $Ex(Node)$ is the number of instances in the current node and $Ex(SubTree_i)$ is the number of instances in the sub-tree *i*.

Finally a *uniqueness ranking* is generated for every feature of every node, using the features of the parent node as negative examples to determine this ranking:

$$R(w \mid Node) = \frac{P(w \mid Node) W(nodeAsSubtree)}{P(w \mid ParentNode)} \qquad (33)$$

where *Node* is the current node and *ParentNode* is the parent of the current node. $W(nodeAsSubtree)$ is the weight factor assigned to the current node when it is propagated to the parent. If a feature is unique to one child of the parent node, this formula returns its maximum value, 1.

Classification

Thanks to the feature propagation, checking all the parent nodes at a given level of the tree is sufficient to identify which branch of the tree the new instance belongs to.

Classification simply involves a breadth-first search of the tree, starting at the root, each child node is considered and the search then recurses down the child node with the maximum PrTFIDF probability using equation (28). The search continues until it reaches a leaf of the tree as illustrated in Figure 18 (below).

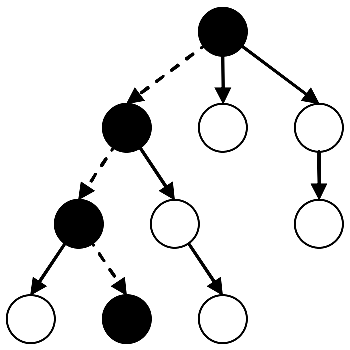


Figure 18 Breadth-first search of Hierarchy

It is now known that the new instance belongs to one of the categories visited by the search; and the PrTFIDF classifier is applied a second time for each category visited, this time using the category's feature vector as in equation (30), and only considering features with a ranking of 1 in equation (33), which indicates a unique feature. The node with the maximum PrTFIDF probability is where the new instance is placed.

Time Complexity

The Naïve-Bayes, Rocchio and PrTFIDF classifiers search all classes to identify a potential match (thorough search), and so are of order $\vartheta(n)$. Comparatively this classifier is of $\vartheta(\log n)$ in the case of a *balanced tree*. Additionally Peng and Choi. quote an increase in accuracy compared to a thorough search algorithm (85% accuracy with tree search cf. 78% accuracy with thorough search).

2.4 Overview of HTML

HyperText Markup Language is a markup language designed for the creation of web pages. It is text-based, but allows structure information such as headings, paragraphs and lists to be defined and can also define semantic information about a document.

A simple *HTML* document is shown below:

```
<html>
<head>
<title>Example HTML Document</title>
</head>
<body>
<h1>This is a heading</h1>
<p>This is a paragraph. <b>This is bold.</b></p>
</body>
</html>
```

2.5 Overview of HTTP

This section presents a brief overview of *HTTP*, the underlying protocol that drives the *world-wide web*. Additionally some more advanced points relevant to this project have been included.

HTTP stands for *HyperText Transfer Protocol* and is the network protocol used to deliver files and data over the world-wide web. The first version was developed in 1990 at CERN by Tim Berners-Lee.

HTTP was originally designed to transmit HTML documents, but it is now used to transmit all types of files.

2.5.1 Protocol Overview

The standard method of addressing files is to use a *Uniform Resource Locator* (*URL*) to identify a location on the server. This is a specific type of *Uniform Resource Indicator* (*URI*). URIs are typically of the form `service:parameters`. URLs are typically of the form `http://host:port/path/file.html`. Often the port is omitted and defaults to the standard HTTP port, 80.

HTTP generally communicates over a *TCP/IP socket connection* and is connectionless and stateless. It is based upon a request/response paradigm, and in its most basic form consists of the following steps:

1. Client establishes a TCP connection to the server host and port given in the URL
2. Send the HTTP Request to the server
3. Receive the HTTP Response
4. Close the TCP connection

The *HTTP Request* consists of a request line specifying the operation (most commonly *GET*, *HEAD* or *POST*), requested path and protocol version. This is followed by zero or more request headers specifying additional information and then a blank line. In the case of a *POST* request the headers are followed by data. A typical request might look like this:

```
GET /test.txt HTTP/1.1
Host: www.cs.manchester.ac.uk
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT)
```

The *HTTP Response* is structured similarly, with the first line specifying the protocol version, a numeric status code and description. This is followed by response headers, a blank line and then the content of the response. A typical response might look like this:

```
HTTP/1.1 200 OK
Date: Fri, 20 Feb 2004 13:31:00 GMT
Server: Apache/1.2.0
Content-Type: text/plain

This is a test document.
```

Note that the numeric status code is machine-readable and the first digit corresponds to the category of response (for example *2xx* indicates a success). Additionally a *MIME Type* is returned (in the *Content-Type:* header) which identifies what format the document is in, for example *text/plain*, *text/html*, *application/msword* or *application/pdf*. MIME Types are widely used to identify file types, and are also used for email attachments and by operating systems.

2.5.2 HTTP 1.1

The initial version, *HTTP 0.9* only supported raw data transfer, and rapidly became a de-facto standard on the Internet. The first official version, *HTTP 1.0* was defined by RFC 1945 in 1996 and added content type negotiation.

Several major problems existed in this version and in 1999 *HTTP 1.1* was defined by RFC 2616. Improvements include:

- **Persistent connections**: Most HTML pages reference other objects such as images; under HTTP 1.0 a new connection is created for each object so a page with N referenced objects requires N+1 connections. Setting up a new TCP/IP connection causes an unnecessary overhead; HTTP 1.1 uses persistent connections which allow several requests to be sent over one connection.
- **Hostname identification**: A *Host:* header is added to all requests allowing one IP address to be allocated to multiple domain names.
- **Proxy support**: HTTP 1.1 adds additional headers to help proxies determine how long to keep documents in their cache.
- **Byte ranges**: The client can specify a byte range to be retrieved instead of a whole document.

- **If modified since**: The client can specify a time and date in an *If-Modified-Since:* header. The server responds as normal except in the case that the request results in a normal *200* (*OK*) response, and the page has not been modified since the time specified where, a *304* (*Not Modified*) response is returned with no data. This enables bandwidth consumption to be reduced by not fetching pages already cached locally unless the remote copy has been updated.
- **Compression**: Compression of documents can be negotiated between client and server.
- **Pipelining**: Several requests can be sent on a persistent connection without waiting for responses. The responses can then be sent together, maximising packet sizes and increasing network efficiency.

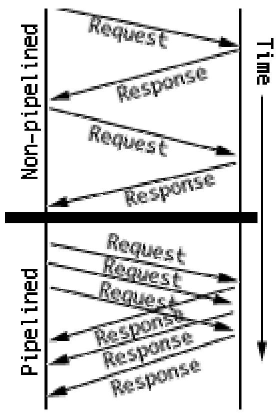


Figure 19 Pipelining [29]

2.5.3 CGI Scripts

A *CGI script* is a program that runs on the web server and generates a *dynamic* response to the client's request. A CGI script runs as a separate process to the web server and interfaces with the web server via the *Common Gateway Interface* (*CGI*) standard [40]. A *servlet* is a similar concept, but runs in the web server process, increasing the efficiency of requests since no additional processes need be created to serve requests.

2.6 Summary

This section has presented a detailed overview of supervised learning classifier algorithms related to this project, in addition to a broad overview of the underlying theory and background. Particularly of note is the Hierarchical PrTFIDF classifier which appears to be ideal for the requirements of this project.

In addition an overview of protocols and technologies related to the markup and delivery of internet and intranet documents has been given.

3 Design Overview

3.1 Introduction

This chapter provides a high-level overview of the design of the system, which aims to fulfil the goals set out in section 1.3. Details of the high-level design decisions made are given in addition to the rationale behind these decisions.

The aim was to design a system which was as close as possible (within the timeframe available) to a production system. Due to the potentially huge scope of the project and the limited available time in which to complete it, strong emphasis has been placed upon developing a modular architecture in order to facilitate the easy extension and modification of the system.

3.2 Requirements

3.2.1 Introduction

This section sets out high-level requirements to constrain the design process and discusses the thought process underlying the decisions made. The requirements aim to fulfil the goals set for the project, and are scoped so as to be realistically achievable within the time available.

Since the project is scoped to focus purely on document classification, clearly a Text Mining algorithm must be core to the solution. As such the discussion of requirements will use the steps set out in Figure 10 as a framework.

Requirement: The system should be made modular wherever possible to facilitate easy extension and modification.

3.2.2 Document Acquisition

Introduction

The Document Acquisition process deals with acquiring documents from a multitude of potential sources and converting them into a text-based format which the Text Mining process is able to understand.

Document Delivery

The most common means of intranet document delivery is HTTP (Section 2.5). In the case of web crawlers operating over the internet HTTP would be the only mechanism guaranteed to be available to retrieve remote documents.

In the case of an intranet document management system this is not necessarily so due to the lower number of web servers involved (often only one) and also the fact that all web servers would most likely be under the administrative control of the same entity responsible for operating the document management system. Hence it is plausible that any necessary means of acquiring documents could be made available to the document management system, for example documents could be accessed through a *file share*.

An additional possible method for acquiring documents would be to build a *custom server* which runs on the web server and transmits documents to the document management system via a *custom protocol*.

A brief comparison of three possible methods for delivering documents to the acquisition component of the document management system is presented below:

	HTTP	*File Share*	*Custom Server*
Cross-Platform Compatibility	Yes	No	Possible
Access to underlying filesystem metadata	Partial	Yes	Yes
Incremental Transfer	HTTP/1.1	Yes	Yes
Dynamic Content	Yes	No	Difficult
Directory Enumeration	No	Yes	Yes
Pull/Push	Pull	Pull	Both

Cross-Platform Compatibility: HTTP has the advantage that it is implicitly cross-platform compatible; any intranet web server can be accessed via HTTP. A file sharing protocol is not guaranteed to be available on all web server configurations and a custom server would require porting to all platforms in use.

Access to underlying filesystem metadata: HTTP servers generally provide the last modified date with the response, however not all servers obtain this from the file on disk, and in systems with dynamically generated content the last modified date is generally set to the current time. File sharing protocols generally provide full access to filesystem metadata and make available the creation, modification and last-accessed dates. Similarly a custom server would have access to any filesystem metadata available.

Incremental Transfer: HTTP/1.1 provides an If-Modified-Since: header which allows only documents which have changed to be retrieved. However every document must be polled individually. Not all web servers support this header and dynamically generated documents generally ignore this header. In the case of file sharing protocols or a custom server, incremental transfer is supported through the access given to the underlying filesystem data – this can be polled and the document only read if it has been updated.

Dynamic Content: HTTP servers generally support dynamic content in the form of servlets or CGI scripts, where instead of returning a file from disk, a program is run and the output of this program is returned, allowing greater interactivity with the user since user input can be taken into account. The user cannot generally tell whether content returned is dynamic or static since the source of the content is obscured by the HTTP layer, and therefore dynamic content requires no special treatment.
File sharing protocols provide no execution support, and attempting to retrieve a document which would be dynamically generated over HTTP would return the source code or binary source of the program responsible for generating the page. Even with the source, execution would be difficult since CGI scripts or servlets are designed to run over HTTP, and on the same platform and configuration as the web server. A custom server could potentially execute CGI scripts or servlets but it would be difficult to account for all possible configurations of scripts without extensive manual tuning.

Directory Enumeration: HTTP servers generally provide no directory enumeration for security reasons. In cases where directory listings are provided, this is in a HTML format depending on the web server and its configuration and therefore cannot be parsed without extensive manual tuning. File sharing protocols implicitly support directory enumeration, and a custom server could easily implement directory enumeration.
Without directory enumeration support the only reliable way to discover new HTTP documents is using a *'web spider'* which *'crawls'* documents and identifies any new links present in them. Newly discovered links are queued and subsequently visited. Eventually a *spider* should be able to find all pages providing there is a link to the page (i.e. when visualised as a graph of pages as nodes and links as arcs, the spider will be able to discover any page in a section of the graph connected to the section in which it began *crawling*).

Pull/Push: HTTP and File Sharing protocols are implicitly Pull based – documents must be requested. A custom server could implement a publication/subscription model allowing document management systems to register to receive updates when a document has been modified or created.

HTTP has huge advantages in terms of cross-platform compatibility and reduced maintenance despite the advantages of the other methods. Additionally some of these advantages such as bandwidth reduction are negated in an intranet environment where it can be assumed ample bandwidth is available to counteract the lack of incremental transfer.

> **Requirement:** In order to support a full range of potential delivery methods, a generic format for a document acquisition plugin should be defined. However in the case of the initial implementation of the system, HTTP should be the default acquisition method.

Document Decoding

Once the document has been acquired, it may be in a number of markup formats. Common formats in use on an intranet include HTML (Section 2.4), *Adobe Acrobat* (*PDF*) and *Microsoft Word*.

> **Requirement:** The document decoding stage of the system should have a capability to detect the format of the document, and decode it into a generic format upon which the Text Mining process is able to operate. In the initial implementation of the system the input format will be limited to HTML.
>
> Conversion to a generic format implies the loss of markup data; this loss should be minimised if possible.

3.2.3 Document Transformation

Introduction

The raw text of the source document is now available to the system. This text must be transformed into a consistent internal representation before it can be classified.

Text Pre-Processing

The pre-processing stage generally involves attempting to conflate the words in the document such that words of a similar meaning in other documents will be considered equivalent. The most basic form of pre-processing is stemming (Section 2.2.3.3), often combined with stoplist elimination (Section 2.2.3.2); however more advanced forms are possible such as converting words to hypernyms and meronyms (Section 2.2.3.4) or alternatively adding these terms to the document.

Requirement: The pre-processing stage of this system should support pre-processing components which are able to change, remove or add terms. The initial implementation should use stoplist elimination and a simple Porter stemmer.

Text Transformation

The conflated text is then generally tokenised if the system's internal representation uses the "bag of words" model.

This system will be constrained to use the "bag of words" model internally since it provides a good processing speed and reduced storage space through the loss of positional data whilst retaining word frequency data.

Requirement: The text transformation state of this system should allow a user-defined tokenisation algorithm to be used to split terms into a "bag of words" representation. In the initial implementation of the system this algorithm should split words into tokens by punctuation or spaces. Hyphenated words should not be split.

Feature Selection

The conflated and transformed text is then statistically analysed; unimportant terms may be discarded during this stage.

Requirement: Since this is the component the text mining components will build upon, its design will be fixed to ensure consistent data. It should operate by calculating weights for each transformed/collated term in the document. TF, TF-IDF and TF-LogIDF weights should be stored for each term.

3.2.4 Document Classification

Introduction

Once the documents have been acquired and transformed into a consistent internal representation they must be classified.

The goals of the project require the classification to be hierarchical, mostly automated and also for it to be personalisable by users of the system.

Classification Algorithm

A number of classification algorithms have been presented in section 2.3. As shown in Figure 12, learning classification algorithms can be grouped into supervised and unsupervised learning classifiers.

As stated in section 2.3.2, unsupervised classifiers (or clustering algorithms) are by definition not trainable and hence unsuitable for the user to personalise.

Even if the personalisation goal of the project is ignored it can be seen that an unsupervised algorithm is unlikely to be as useful as a supervised algorithm since the unsupervised algorithm is designed to group data together according to patterns that it discovers whereas the intended use of the system is to be trained by its administrators to group documents as they see fit.

However there is one point at which an unsupervised algorithm would be useful – in the setup of a system initially containing a large number of documents (this is known as a *semi-supervised classifier* or *bootstrapping*). In this case it would help the system administrators to have a hierarchy automatically generated which they can subsequently tune to their requirements. However this is not a core requirement and would only be of use during the initial setup of the system.

Hence the choice of the main classification algorithm has been constrained to supervised learning algorithms. Of the algorithms presented in section 2.3.3, the Hierarchical PrTFIDF classifier is the only classifier inherently supporting a hierarchy, and utilising this information to decrease the time complexity of the classification process.

The Rocchio Text Classifier is efficient but has difficulties coping with disjunctive categories, which would require the user to be careful when creating categories.

The k-NN classifier has *high variance* and *low bias*, which means it has a tendency to *overfit* itself to the data and have difficulty classifying new examples slightly different to the training data. A naïve-Bayes classifier has *low variance* and *high bias* and has the opposite tendency; to accept examples as part of a class they should not be.

Overall, the hierarchical PrTFIDF classifier seems to be best suited to the design goals of the system since it is inherently hierarchical and it has a much faster training time than most 'flat' classifiers because it uses the hierarchy to reduce the number of comparisons from *n* to *log n* on average. It is also based upon the PrTFIDF classifier which has a strong probabilistic foundation and in tests [31] was found to be more accurate than the TFIDF classifier in almost all cases, and generally more accurate than the naïve-Bayes classifier especially in examples with low amounts of training data (which would make the system easier to maintain). In tests of the Hierarchical PrTFIDF classifier, its classification accuracy was quoted as being better than PrTFIDF [32].

Requirement: The system should use a hierarchical PrTFIDF classifier to classify documents. It should support the later addition of a clustering algorithm to bootstrap the initial classification of documents.

Hierarchy Personalisation

A generic hierarchy is defined by two relations, *membership* of an instance in a category and *inclusion* of a category in another. In a generic classification system, a *canonical* classifier and hierarchy is provided which the user sees by default. If the user wishes to personalise the system they must reject one or both of these relations:

	Definition	*Personalisation*
Membership relation $\in$	*'is-a'* Relates instance to category	Reject *membership relation* but accept *topology*. Must reject/replace *canonical classifier.*
Inclusion relation $\subset$	*'a kind of'* Relates category to other categories	Accept *membership relation* but reject *topology*. Accept *canonical classifier* but remap *topology*.

The simplest case is rejection of the *inclusion relation* – this simply requires a one-to-one mapping between the *canonical topology* and the *user topology*.

More difficult is the rejection of the *membership relation* – this implies the user wishes to classify documents differently to the system defaults and hence the *canonical classifier* is no longer of use to the user. Note that if the user also wishes to reject the *inclusion relation* this does not add to the difficulty since once the *canonical classifier* has been rejected the replacement classifier can be applied to any hierarchy desired.

If the membership relation is rejected, the scalability of the system is affected because instead of providing one classifier for *n* users, between 1 and *n* classifiers must be provided.

By providing many classifiers, the maintenance of the system becomes more complex. Although a user may create their classifier and train it to satisfaction, it will require occasional maintenance from time to time due to *concept drift.* This is a term used in data mining referring to changes over time in the concepts associated with terms, for example new fields of research would warrant the addition of new categories into a system and certain terms would become associated with the new categories. For example, "Clinton" was once a good term association with "president of the united states"; now it is not. [27]

One idea that was initially considered was to try to provide some kind of means to connect the canonical hierarchy to each user's hierarchy. For example in a naïve-Bayes classifier it would theoretically be possible to modify the classifier in equation (18) to take into account both the terms in the document, but also the position of the document in the canonical hierarchy. For example:

$$\text{classify}(x_1,\ldots,x_n) = \text{argmax}_c\, P(UC=c)P(UC=c \mid GC=c)\prod_{i=1}^{n} P(X_i = x_i \mid UC=c) \quad (34)$$

This equation aims to classify documents but also take into account their canonical classification. *UC* represents the user category assigned to the document and *GC* represents the canonical category assigned to the document. Initially:

$$P(UC=c \mid GC=c) = \begin{cases} 1 & \text{if } c = UC = GC \\ \approx 0 & \text{otherwise} \end{cases} \quad (35)$$

Initially this is 1 where the user category matches the global category and approximately (but not equal to) 0 in all other cases. As the user moves documents around between categories these probabilities would be adjusted by the training component and once the user had broken a canonical category up sufficiently these probabilities would all be approximately equal and hence would become insignificant compared to the product on the right side of the equation, effectively disconnecting the user hierarchy from the canonical hierarchy.

However this is more based upon hunch than theory, and also raises additional questions as to how the topology would map onto the canonical topology when categories can be added, deleted or moved in both. Solving these problems would take more time than is available due to the already complex architecture proposed.

A simpler and more reliable solution is proposed – to allow any user wishing to customise their view of a hierarchy to clone the hierarchy and then alter the training data and topology themselves. This is simpler to implement and provides sufficient personalisation capability for most user's needs.

One problem with this personalisation approach is that having potentially one hierarchy per registered user means that every new document will have to be classified several times; the complexity of the classification algorithm becomes critical. Fortunately the hierarchical PrTFIDF classifier is very efficient.

> **Requirement:** Any user of the system should be able to clone a hierarchy they are viewing and personalise it by altering the topology and changing the training data to alter the classification of documents. One hierarchy should be flagged as the canonical or global hierarchy which is the default view. Users should be able to make their hierarchy public so that other users can view it if desired.

3.3 System Architecture

3.3.1 Introduction

As stated in the requirements the system is to be made as modular as possible to facilitate easy modification and extension.

This section presents an architecture which aims to fulfil the requirements set out in the previous section whilst retaining as much modularity as possible.

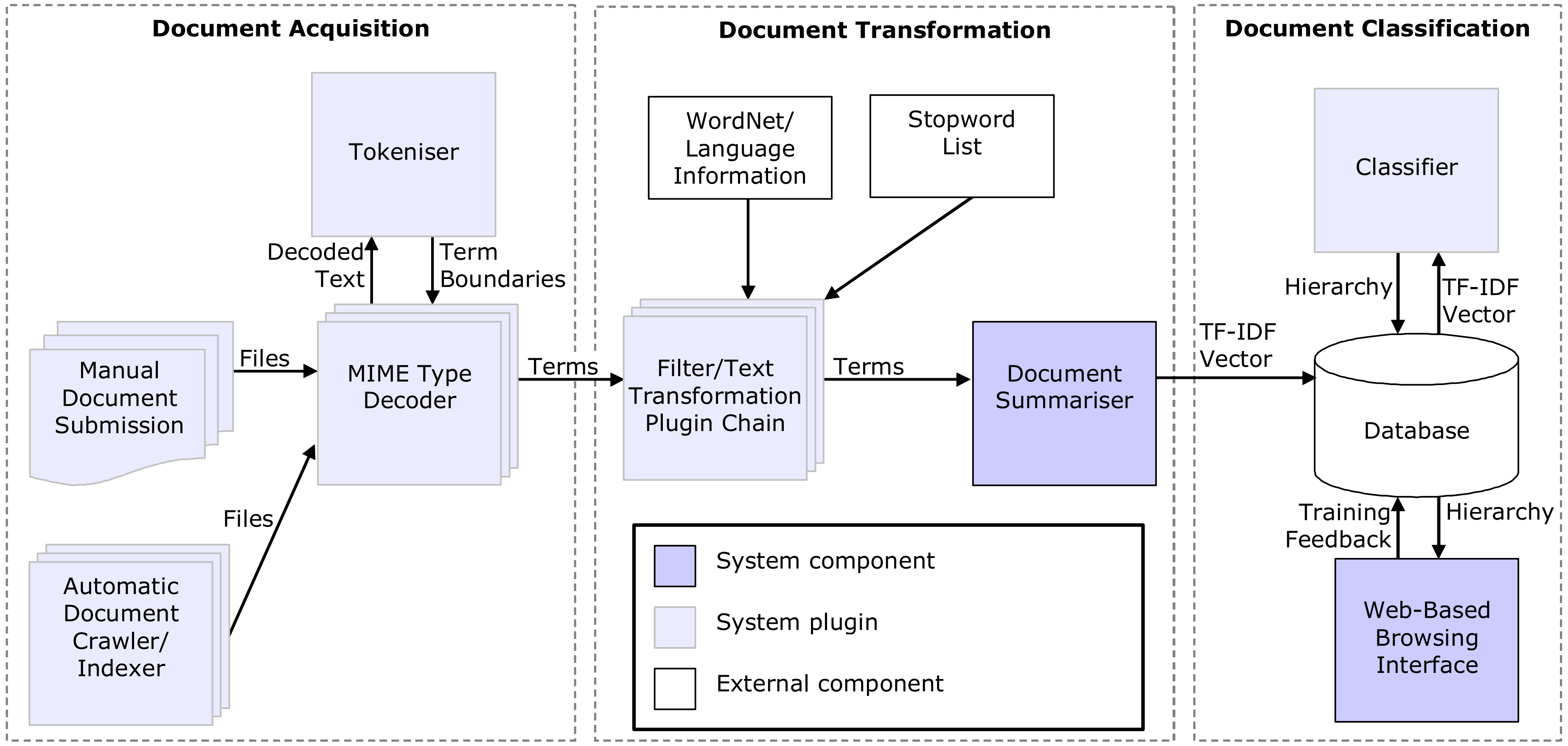


Figure 20 System Architecture

3.3.2 Document Acquisition

Document Submission

The *manual document submission* and the *automatic document crawler* plugins support the entry of documents into the system. The plugin should specify an URI to uniquely identify the document in the system and the MIME type of the document. Documents should be identified by the URI and the ID of the plugin that submitted them.

MIME Type Decoder

The system should poll all loaded *document decoder plugins* until it finds one which accepts the document's MIME Type (section 2.5.1), or raise an error if a plugin could not be found.

The decoder plugin selected should then convert the document into plain text and call the *tokeniser plugin* in order to identify token boundaries in the document. It then passes an array of term objects onto the *filter/text transformation plugin chain*.

The reason that the tokenisation is performed by the decoder rather than plain text being output and tokenisation occurring after the decoder is to allow the decoder to attach *metadata* to the term object, for example formatting metadata (bold/title etc). A *filter/text transformation plugin* that understands this metadata can use it to alter *term bias* or act upon the information in some other way.

Tokeniser

The *tokeniser plugin* accepts text input one character at a time from the decoder plugin and returns a Boolean value identifying whether the character is a term boundary. The decoder plugin uses this information to break the document up into an array of term objects.

3.3.3 Document Transformation

Filter/Text Transformation Plugin Chain

All text transformation and pre-processing operations are essentially accomplished by the addition, deletion or modification of the terms in a document. Hence this stage of the processing is represented by a filter chain. Plugins are placed into this chain in an explicit order and each plugin has access to an ordered array of terms and can add, modify or delete these as it sees fit.

The term object contains the string that it represents, a bias value (which defaults to 1) and an optional *term context object* that can be attached by the decoder plugin to provide additional metadata about the term. If the filter plugin understands the attached context object, it can use the additional metadata.

Examples of filter plugins include:

- A *format weighting plugin* could use knowledge of the term context objects attached to terms by a *HTML decoder plugin* to double the importance of terms which were bold by multiplying their bias by 2.
- A *stemming filter plugin* could replace terms with stemmed versions.
- A *WordNet filter plugin* could replace terms with hypernyms or it could append hypernyms and meronyms to the array of terms
- A *n-gram generator plugin* could generate all collocated n-grams in the document (All possible pairings of words immediately next to each other). It could append either all of these to the array of terms, or it could be more space efficient by appending only n-grams known to be meaningful. Alternatively it could replace the terms list with a list of n-grams, converting the internal representation from the "bag of words" model to the "bag of n-grams" model

Document Summariser

The document summariser component counts the number of occurrences of each term in the array of terms.

It also averages the *bias* associated with each term. Note that *term bias* and *term frequency* are different terms; *term frequency* being the proportion of the document made up by the term and *term bias* being the average bias value for the term. For example a term which occurred 2 times in bold and 1 time in normal text would (with a format weighting plugin assigning double importance to bold text) have a summarised bias of $\frac{1+2+2}{3} \approx 1.6$.

Term count, Term frequency, TF-IDF and TF-LogIDF weightings are then calculated for each term and the data added to the database. Note that term frequency, TF-IDF and TF-LogIDF values are not required by the classification algorithm used but are calculated for the use of search plugins or clustering algorithms used to bootstrap the classifier.

3.3.4 Document Classification

Classifier

The classifier classifies documents into all hierarchies in use with a hierarchical PrTFIDF classifier.

Web-Based Browsing Interface

The *web-based browsing interface* allows users to browse any of the hierarchies available. It also allows users to clone any available hierarchy and if the hierarchy is owned by the user it allows modification of the topology and pinning/unpinning documents to a particular point in the category (pinned documents constitute the training set for the classifier)

3.4 Summary

An architecture has been described for a modular, flexible and extensible document management system.

The document acquisition architecture allows plugins to be added without change to the core system to support the gathering of documents via multiple delivery methods and to allow the decoding of multiple formats of document.

The filter plugin chaining system allows a great number of text transformation and pre-processing algorithms to be used without change to the core system.

The architecture fulfils the goals set out in 1.3 and provides a browsable document hierarchy to which new documents are automatically added and classified. It supports user-personalisation through the cloning and modification of the hierarchy. The classification algorithm in use is extremely efficient and also robust to poorly created hierarchies (for example disjoint categories).

4 Detailed Design

4.1 Introduction

This chapter builds upon the requirements and architecture set out in the high-level design overview (chapter 3). It details the tools that were used to implement the system, the rationale behind major design decisions that were taken and the problems that arose during implementation.

4.2 Implementation Tools and Techniques

This section provides an overview of the tools and techniques that should be used in implementing the system.

Microsoft .NET Framework 1.1

Microsoft's .NET framework is a development platform similar to Sun Microsystems' Java. It provides a number of programming languages (such as C#, Visual Basic .NET, J#, C++, Perl, Python etc) which are compatible with a defined *Common Language Specification* (*CLS*).

These languages are compiled into *Intermediate Language* (*IL*), which is itself a language and is both (source) language and platform neutral. An application is then distributed in IL format, which can then be run by a *Just-In-Time* (*JIT*) compiler which converts the IL into platform specific code. JIT compilers are available for a number of platforms.

The primary reason for use of the .NET framework was that this is the preferred development platform of EDF [10] and hence the resources they were able to provide for the project are compatible with .NET.

Additionally .NET has a number of advantages over Java, for example the class library provided with .NET is better developed in some areas, for example it supports *asynchronous socket I/O* and provides a *ThreadPool* to optimise some multi-threaded applications. Also despite the .NET JIT currently being available for less platforms, it has a major advantage in that multiple programming languages support the CLS allowing programmers to choose the language that suits them, whereas Java only supports one language. .NET also has better support for *SOAP* (*Simple Object Access Protocol*) web services than Java.

The choice of which language to implement in is entirely down to personal taste since all CLS compliant languages are capable of producing identical IL after compilation, and because all languages use the same *Base Class Library* (*BCL*), the helper functions/classes available are identical. In this instance *C#* was chosen as the implementation language.

.NET Thread Pool

The .NET framework *ThreadPool* is the basis for all asynchronous programming in the .NET framework.

Its operation is simple – it provides a pool of threads which are able to execute any function specified as a work item. Submitted work items are queued, and a pool of threads (usually 25 threads per processor) continuously poll for queued work items and execute any work items found.

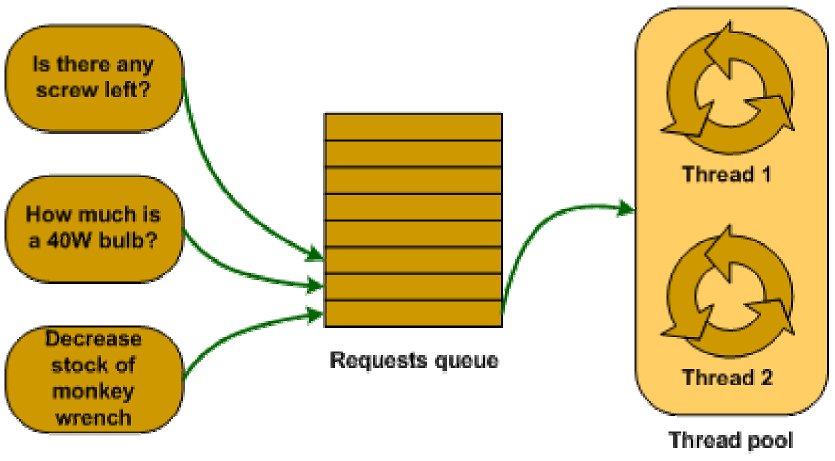


Figure 21 ThreadPool illustration [37]

This has a huge advantage by eliminating the overhead of threads starting up and shutting down, especially useful if the work item is a very short function in which case starting a thread just for this is very inefficient. It also removes the burden of thread management from the programmer. The thread management built into the *ThreadPool* optimises the number of threads to ensure enough threads are running to maximise CPU usage but not so many threads that a lot of time is being wasted in context switches.

Another huge advantage of the ThreadPool is the optimisation of wait operations. Traditionally any multithreaded program needing to wait for events would use some form of sleep call to suspend the thread. This left a thread running and using up system resources whilst doing nothing. This is particularly wasteful in network I/O operations where a number of threads may all be suspended for a long time and using system resources. The ThreadPool has the capability to queue wait requests, and execute them once the wait has completed. In the case of a server this increases capacity and also neatly sidesteps the problem of needing to limit the maximum number of connections to prevent denial of service.

All I/O waits (for example reading documents from remote web servers) use the ThreadPool. Additionally the ThreadPool is used wherever possible for background execution of tasks.

Microsoft SQL Server 2000

Microsoft SQL Server 2000 was chosen as the backend DBMS. This is because this is the database used by EDF [10] and hence they were able to provide access to this for testing purposes. Data access is compartmentalised into a *Data Access Layer* (*DAL*), so switching to an alternative DBMS would only require changes to the DAL code.

SQL Server *stored procedures* are batches of *T-SQL* statements (*T-SQL* provides ANSI SQL functionality with some enhancements such as flow control statements). An effort has been made to use these for performance-critical operations, since they provide faster performance because the execution plan for all queries inside the procedure are precompiled (although this can be a hindrance if they are not recompiled if the database statistics, i.e. relative sizes of tables and index densities, change significantly). Stored procedures are also faster because the code executes on the database server, this means that only the parameters and the results are sent over the network, all processing happens on the database server.

Subversion

Subversion [35] was chosen for source code control. This is a greatly improved version of Concurrent Versions System (CVS) and provides an additional backup of code in addition to a full revision history allowing changes to be reverted without requiring code to be rewritten. Code was stored in the EDF [10] Subversion repository.

Coding Standards

Microsoft's Internal Coding Guidelines [36] were chosen as the standard for source code formatting to ensure all code is presented in a consistent format.

Particular attention was paid to including XML documentation comments on all public methods and properties of public classes. This both allows automatic generation of help pages for the system's API and also allows auto completion and pop-up help for programmers using the API.

Also `try/catch` blocks with appropriate error logging were used as frequently as possible, particularly around calls to user plugins to prevent errors in these from causing a fatal error in the system.

4.3 Portability

The system should be developed in pure .NET with no native code. This allows the system to be used on any platform with a .NET JIT.

4.4 Design Highlights

This section presents an overview of the major achievements of the design and implementation of the system. For a more detailed account of the implementation of the system please refer to the implementation details section in Appendix 0.

4.4.1 System Configuration

The system supports configuration through .NET's XML application config. This supports storage of basic name/value pairs. The dagama configuration manager supports reading of configured settings, the name of each setting is prefixed by the fully qualified name of the class it refers to in order to ensure that the names of plugin configuration settings do not conflict. A sample configuration file is shown below:

```
<?xml version="1.0" encoding="utf-8" ?>

<configuration>
  <appSettings>
     <add key="Dagama.Acquire.DecoderPlugin.DecoderManager.RegisteredPlugin0"
    value="Dagama.Acquire.DecoderPlugin.TextHtml.TextHtmlDecoder"></add>
    <add key="Dagama.Acquire.DecoderPlugin.DecoderManager.RegisteredPlugin1"
    value="Dagama.Acquire.DecoderPlugin.TextPlain.TextPlainDecoder"></add>
  </appSettings>
</configuration>
```

4.4.2 General Architecture

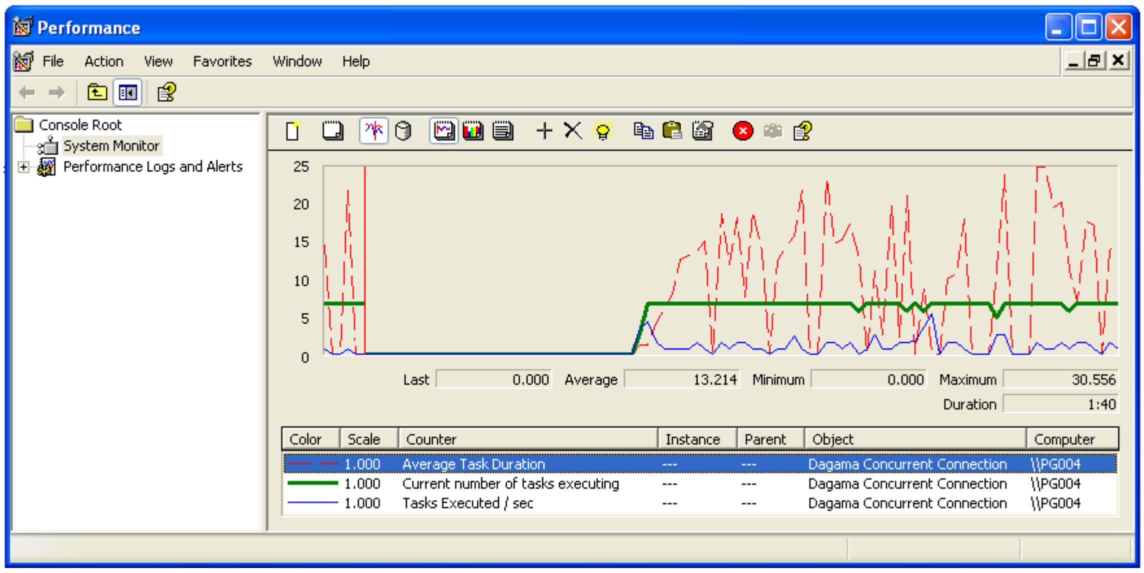
The architecture is designed to be as modular as possible; in particular all data access code is separated into a Data Access Layer. This ensures that switching the database backend of the system only requires rewrites to the DAL. All communication between the DAL and other parts of the application is achieved through abstract data types which are not specific to the database used.

The architecture is also designed to be as robust as possible; particularly around plugins where careful error handling is used to ensure that errors in plugins are logged but do not cause a fatal error in the whole application.

The plugin architecture is a compromise between efficiency and extensibility, and the use of plugins prevents some possible optimisations of the system which would require each component's interface to be less generic (and hence not a plugin) in order to optimise data structures and operations performed. However, every effort has been made to maximise the efficiency of the architecture. For example when filter plugins process the terms array, instead of copying the array and passing it to each individually, a pointer to an array modification interface is passed to each which enables them to seek through and modify the array with constant memory usage.

The document summariser which converts from a list of terms to the internal "bag of words" representation uses the quick sort algorithm to sort the list alphabetically and then loops through this sorted list to enable the summarisation to be performed in-place with constant memory usage and without the requirement for hash tables.

Performance monitoring is provided through the Windows performance counters API. This allows the current state of the application to be monitored both locally and remotely. It also allows this data to be logged for later replay.



Last	0.000	Average	13.214	Minimum	0.000	Maximum	30.556
						Duration	1:40

Color	Scale	Counter	Instance	Parent	Object	Computer
🟥	1.000	Average Task Duration	---	---	Dagama Concurrent Connection	\\PG004
🟩	1.000	Current number of tasks executing	---	---	Dagama Concurrent Connection	\\PG004
🟦	1.000	Tasks Executed / sec	---	---	Dagama Concurrent Connection	\\PG004

Figure 22 HTTP connection performance counters

Classes requiring resources to be released or specialised cleanup implement the *IDisposable* interface which is a standard .NET interface which allows resources to be guaranteed to be released at a specific point rather than depending upon the garbage collector (which is "lazy" and therefore cannot guarantee when resources will be released, and also does not guarantee a class destructor function will ever be called).

A number of helper classes are provided by the framework. These are used in various other parts of the application. For example:

- **Concurrency Limiter**: This class supports the limiting of the number threads executing a particular task. This is achieved by having a counter which is checked and incremented/decremented inside a *mutex* block (to ensure only one thread can access the counter at a time). If the maximum number of slots are currently in use, the *mutex* is released and then the thread blocks on an *AutoResetEvent* object.

  When a slot is freed, the *AutoResetEvent* object is signalled, releasing exactly one waiting thread which then fills the slot. The advantage of this is it maximises efficiency (compared with using a timer to repeatedly sleep then poll for a free slot), the thread simply sleeps until a slot being freed wakes it.

- **Continuous Processing Thread Base Class**: Base class for a thread which is designed to repeatedly execute a work function which returns true if work was found or false if it was not.

  If false was returned, the thread sleeps using a truncated binary exponential back off algorithm (1 second, 2 seconds, 4 seconds, 8 seconds etc) up to a defined maximum. The purpose of this is to reduce the load if the task that is continuously being executed is not doing anything (for example polling for documents to be classified but none are being added).

- **Periodic Processing Thread Base Class**: Base class for a thread which is designed to execute a work function every *n* seconds. Rather than sleeping for *n* seconds after the work function has been executed, the duration which work function took to execute is calculated, and the thread sleeps *n – duration*. This ensures that the work function runs exactly every *n* seconds where possible, if the function takes longer than *n* seconds it is run continuously.

- **Rate Limiter**: This class is designed to limit the rate at which a task is being executed. It achieves this by monitoring the number of tasks executed over a given window, for example enforcing a maximum of 5 tasks/minute averaged over the last 10 minutes.

  If the current rate is over the acceptable maximum, the thread sleeps until the next logged task run is due to be expired (i.e. has become older than the current time minus the window length), then recomputes the average rate and sleeps again if necessary. The average over data currently within the window is recomputed within a *`mutex`* block for thread safety.

4.4.3 SQL Server Data Access Components

Schema

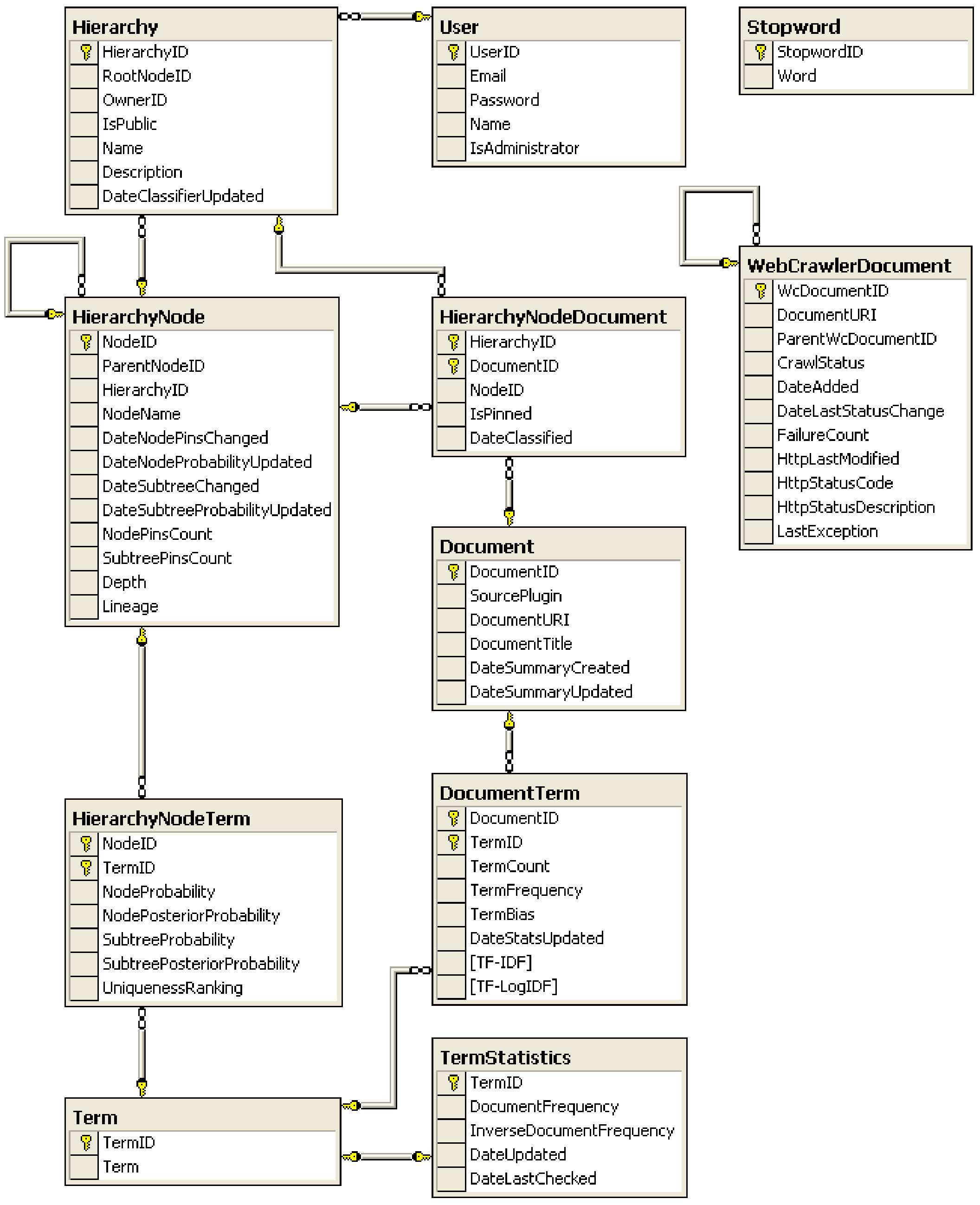


Figure 23 SQL Server Database Schema

The SQL Server schema is shown in Figure 23 (above). The function of each table is as follows:

- **Document** contains one row per document in the system.
- **DocumentTerm** contains one row per term in each document.
- **Hierarchy** contains one row per hierarchy in the system, each row has a reference to the hierarchy's root node.

- **HierarchyNode** contains one row per node in each hierarchy.
- **HierarchyNodeDocument** associates documents with hierarchy nodes. `IsPinned` is set to 1 if this document has been pinned (is part of the training set)
- **HierarchyNodeTerm** contains statistics computed over the training set by the hierarchical PrTFIDF classifier.
- **Stopword** contains the stopword list.
- **Term** contains all terms known to the system.
- **TermStatistics** is a vertical partition of the Term table and contains statistics computed about each term.
- **User** contains details about users of the system.
- **WebCrawlerDocument** contains the web crawler plugin's crawl queue and the status of all documents known to the web crawler.

The SQL Server Data Access Components use transactions where appropriate to ensure that updates occur atomically. In some places due to high concurrency the locking and transactions are tuned manually for maximum throughput.

The following sections give an overview of the more complex database algorithms designed and problems that arose during design and testing.

The Data Access Components are a key factor in the efficiency of the system since all operations depend upon the underlying database for storage and retrieval of data. The efficiency with which this data is stored and (most importantly) retrieved directly impacts the efficiency of the system.

In designing the database structure and algorithms involved, the aim has been to maximise the efficiency of browsing through the system. As a result, all information regarding document vectors, category vectors, classification decisions etc. has been cached where possible. Updates of this cached information are designed to be as efficient as possible by only updating the minimum amount of data required by a change. Calculation algorithms have been converted from mathematical equations to relational algebra where possible to utilise the highly optimised processing capabilities of the underlying relational database management system.

Note that the following sections refer to *term statistics* and *document term statistics*. The difference between these is that *term statistics* refers to the calculated document frequency and inverse document frequency of terms over all documents, *document term statistics* refers to the TF-IDF and TF-LogIDF weightings calculated for a term specific to a given document.

Issues with concurrency on Terms table

Particular problems were found with high levels of concurrency on the *Term* table – *term statistics update threads* are continuously updating the term statistics, *document term statistics threads* are continuously reading the term statistics and document processing threads are continuously both looking up term IDs for terms found in the document and adding new terms not currently existing in the database.

The major problem with this table in particular is that unlike other tables, where generally only one thread is trying to access the row at a time (a document has to be inserted into the system before the TF-IDF weightings can be calculated etc), with the *Term* table a number of threads are liable to be accessing the same row at the same time because most documents will use the same words. As a result of this the locking had to be controlled by vertically fragmenting the *Term* table.

This ensures that the term lookup part associating strings with TermID values is read-only (and occasionally an insert occurs when a new term is found). Additionally instead of updating term statistics every time a document is added or updated in the system (which would require lots of continuous updates to the table by all document processing threads), one thread is used to periodically re-calculate term statistics (using a stored procedure) for all terms in the database. If the term statistics differ from their previous values, a date updated timestamp is set to the current time.

The procedure which performs inserts into the *Term* table also inserts a stub row into the *TermStatistics* table with just the term ID and all other values *null* to cause the term statistics update procedure to process this term – this also means that the term statistics update procedure never performs reads on the *Term* table itself, reducing locking conflicts. There is potentially a delay between statistics updates where a newly added term in the *TermStatistics* table has no statistics present, in this case the missing term will be ignored by the classification process and the document will be initially classified, once the term has been propagated into the *TermStatistics* table its newer timestamp will cause the document to be re-classified taking the term into account.

Additionally the term statistics update process had to be further modified to update one term per execution rather than updating all terms in one execution. This is because with large numbers of documents and terms in the system the execution was taking longer than the command execution timeout. Rather than setting this to be infinite and risking problems due to a command getting stuck in an infinite loop or the network connection inadvertently becoming disconnected and causing the command to wait forever, the command was broken up into smaller chunks of work. This was achieved by setting a variable to the start time of the process, and repeatedly re-executing the stored procedure passing the start time as an argument. This also splits each update into a single transaction, and ensures that row locks are released immediately after the row has been updated (at the expense of a slower query). The stored procedure updates only rows which were last updated before the stated timestamp, and execution eventually stops when there are no rows matching this criteria.

Another problem occurred during the initial setup of the system, due to a large number of new terms being discovered in the first 10-20 minutes the system is running (since it is building up its core vocabulary), lots of locking timeouts (where the database waits longer than a specified timeout to obtain a lock, so gives up) were occurring on inserts to the *`Term`* table. To overcome this error handling was added to catch this type of error and sleep for a random delay between 10 seconds and 1 minute, and then retry. This process is repeated up to 3 times before finally raising an error.

Document Term Statistics Update Algorithm

The process of performing document term statistics updates is designed to be as efficient as possible by using timestamps. Every execution looks for one document term with invalid TF-IDF value, either because the TF portion (the document itself) has changed or the IDF portion (term statistics) has changed. This is determined through timestamps.

The TF-IDF and TF calculations are not required by the classification algorithm used but are provided calculated for the use of search plugins or clustering algorithms used to bootstrap the classifier.

The process is continually executed ensuring document term statistics are updated as quickly as possible when they become out of date. A *truncated binary exponential backoff algorithm* is used to reduce load on the database, when this procedure has no document term statistics requiring update it sleeps for an increasing amount of time (1 second, 2 seconds, 4 seconds, 8 seconds etc) up to a defined maximum wait period. New work arriving will reset the delay to 1 second. Whilst work is available, the thread sleeps for 0 seconds between each item being updated, this reduces load on the system by encouraging the operating system to make a context-switch allowing another thread to execute in the foreground before the current thread is re-scheduled into the foreground again.

Again this process uses a stored procedure for efficiency, the stored procedure updates one document term and then returns, it is called repeatedly to update all document term statistics. It is also run as a separate thread to the term statistics updating procedure to ensure that a constant influx of work for this procedure would not prevent the term statistics thread from running.

Hierarchy Maintenance

SQL server triggers are used to maintain the `Depth` and `Lineage` fields on each node. `Depth` indicates the node's depth in the hierarchy, 0 being the root, 1 being adjacent to the root etc. `Lineage` indicates the path from the node to the root, e.g. /1/2/3/.

A SQL server trigger is simply a stored procedure (batch of T-SQL statements) which executes when the data in the database is altered.

Hierarchical PrTFIDF Training

The hierarchical PrTFIDF training procedures are designed to be as efficient as possible to reduce the overhead of having multiple hierarchies in the system due to users being able to create personalised hierarchies.

In this section, *training set* refers to documents which have been "pinned" to the node by the user. *Node training data* refers to the feature vector calculated for a particular node given its training set. *Subtree training data* refers to the feature vector calculated for a particular node given both its node training data and the *subtree training data* of all of its children.

Firstly, node training data update occurs when the either the user has added or removed documents from the training set, or a document's term weightings have changed:

1. The process initially identifies one node in any hierarchy requiring update (by checking the node's last updated timestamp, the maximum last updated timestamp of any document in the training set and the timestamp the node's training set was modified).

2. It then re-calculates node training data for documents pinned to (in the training set for) that given node.
3. It also updates the node pins count, which is the number of documents in the training set for that node.
4. The algorithm is run repeatedly until no nodes are found requiring update.

This is a difficult process to optimise in a relational database since the size of the feature vector is not fixed for the node's training data or for the documents that are pinned to the node, and additionally the update process cannot delete terms from the node's training data because the given term may be zero for a particular node, but still present because it has been propagated down from the subtree. This is optimised using a complex outer join (see Figure 24 below) between the tables, pairing together aggregated terms from the training set for this node (using the SQL COUNT and SUM functions over a GROUP BY clause) with terms currently in the node training data for the node. Terms found in the training set but not in the node training data are added to the node training data, other terms are updated. Finally the node's last updated timestamp is set to the current time to indicate the node training data has changed.

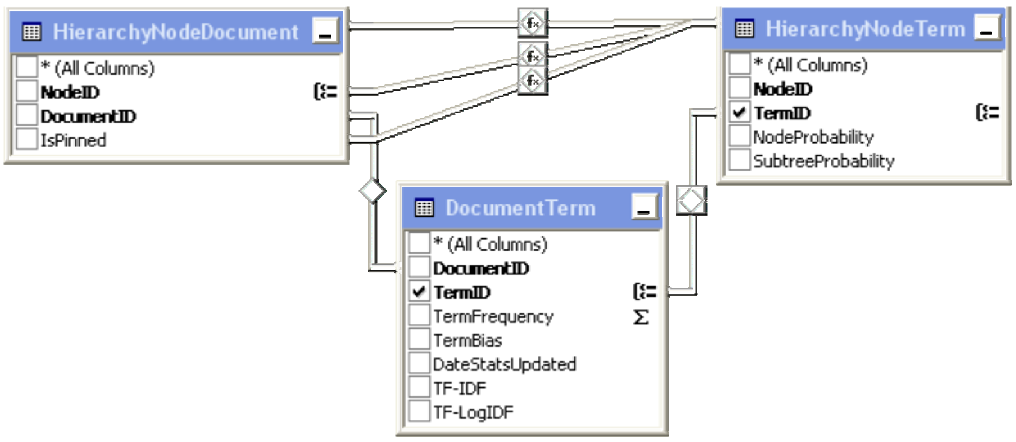


Figure 24 Node training data update

The outer join optimises this query because the database performs a hash join internally between aggregated terms in pinned documents and terms currently associated with the node. The database is highly optimised to perform this sort of operation, and returns a list of paired terms which the application can then quickly compare.

Additionally the use of aggregate SQL functions (COUNT and SUM functions over a GROUP BY clause) also optimises the query, again because the database is highly optimised to perform this sort of operation, and also because it reduces the amount of data transferred over the network from the database.

Secondly, subtree training data is propagated up the hierarchy.

1. Initially a query selects a node to update by comparing a given node's subtree training data last update timestamp, its node training data last update timestamp and the maximum subtree training data last update timestamp amongst all of its children. The algorithm prioritises nodes closer to the leaves of the tree.
2. The node pins count is summed across all child nodes of the current node to get the subtree pins count.
3. An outer join query similar to the one used for node training data is used to pair terms in the node training data with terms aggregated from the subtree training data of all child nodes:
    a. The training data from the current node and aggregated from child nodes of the current node becomes the subtree training data for the current node. Terms are added to the current node's subtree feature vector if not present. Additionally if a term has been removed from both this node's node feature vector and all of its child node subtree feature vectors it is removed completely to reduce storage space.
    b. All terms propagated from all child nodes of the current node have their uniqueness ranking updated.
4. The current node's subtree training data last update timestamp is set to the current time.
5. The algorithm is run repeatedly until no nodes are found requiring update.

As before, the outer join optimises this query because the database performs a hash join internally between aggregated terms in pinned documents and terms currently associated with the node. The database is highly optimised to perform this sort of operation, and returns a list of paired terms which the application can then quickly compare.

The reason for prioritising nodes closest to the leaves of the tree, is to ensure that updates are efficiently propagated from the leaves down to the root, requiring on average $\log n$ updates. The process is illustrated below in Figure 25, where nodes with updated node training data are shown diagonally striped and black nodes represent the nodes updated by the algorithm in the 6 iterations it takes to propagate 2 updates through the entire tree (c.f. 10 if all nodes were exhaustively updated).

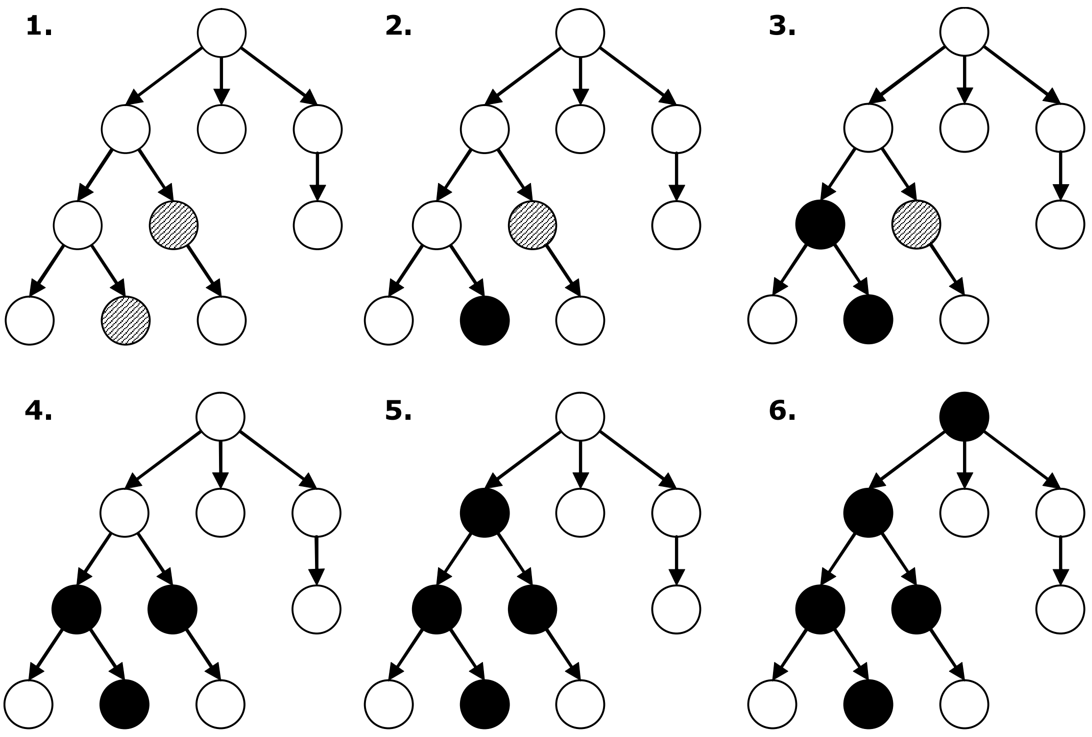


Figure 25 Hierarchical PrTFIDF subtree training propagation

Thirdly, $P(C_j \mid w)$ is recalculated for all terms in the updated classifier. This is termed the 'posterior probability' in the system. A changed classifier is detected by comparing the subtree training data last update timestamp of the root node in the hierarchy with the hierarchy's classifier last update timestamp. The root node's subtree training data last update timestamp will only be updated once all changes have been propagated through to the root node by the previous process.

When the above circumstances have been detected, $P(C_j \mid w)$ is recalculated for all vectors. Equation (25) can be rewritten as:

$$P(C_j \mid w) = \frac{P(w \mid C_j)P(C_j)}{\sum_{C' \in C} P(w \mid C')P(C')} = \frac{P(w \mid C_j)\frac{|C_j|}{|D|}}{\sum_{C' \in C} P(w \mid C')\frac{|C'|}{|D|}} \tag{36}$$

The cached value of this is calculated in two phases. Firstly the numerator of the equation is calculated, secondly all values of the numerator are summed, since this provides the denominator. The two are then divided and stored in the database. A value is computed across both all subtree probabilities and all individual node probabilities. Finally the hierarchy's *classifier updated timestamp* is set to the current time which causes all documents in the hierarchy to be re-classified.

Note these procedures run repeatedly to process one node at a time, in addition to the specific reasons stated in this section, the more general reasons stated in the previous section ("*Document Term Statistics Update Algorithm*") are also equally valid. As before, a truncated binary exponential backoff algorithm is used to control the frequency of polling the database during periods where no nodes are found requiring update. Additionally both procedures also use transactions and locking to ensure that the data is never left in an inconsistent state.

Hierarchical PrTFIDF Classification

The classification algorithm identifies documents requiring update by identifying documents which not pinned to the hierarchy and are either currently not classified, or the hierarchy has changed since the document was last classified, or the document has changed since it was last classified.

Unfortunately because all changes to the hierarchy involve changes to the root nodes, a change to the hierarchy requires reclassification of all non-pinned documents, fortunately the hierarchical algorithm is efficient at classifying documents.

Thanks to the precalculated values of $P(C_j \mid w)$, the decision rule in equation (29) can be simplified to:

$$H_{\Pr TFIDF}(d') = \underset{C_j \in C}{\arg\max} \sum_{w \in F} P(C_j \mid w) \times P(w \mid d') \qquad (37)$$

The classifier performs a breadth first search to identify a path from root to leaf most similar to the document, using the above formula and only considering features with a uniqueness ranking of 1, indicating a feature unique to the particular branch of the tree. The path through the tree is stored, and then the document is compared to each node along this path again using the formula above, but this time considering all features, to identify which node it is most similar to. The document is then assigned to this node by the classifier.

4.4.4 Web Crawler

A web crawler plugin was provided as the default document acquisition method. This repeatedly checks the crawl queue database for new documents to crawl, and retrieves these.

It limits the load on remote servers by limiting the number of documents that can be simultaneously downloaded, and also by limiting the rate at which documents are downloaded. HTTP/1.1 connections to hosts are pooled by the .NET framework, which allows subsequent requests to these hosts to reuse already established connections.

The queue management process uses locking which would allow a cluster of web crawlers to be operated if necessary, increasing the throughput of the system.

4.5 Summary

The finalised design conforms to the architecture set out in the high level design. It also has the potential to fulfil all of the project goals providing the implementation works as specified in this design section.

Some of the key achievements of this design are:

- Automatic, fault-tolerant background operation.
- Supports plugins to allow user modification of system configuration without code changes.
- Highly optimised process for updating documents in the database and various summaries calculated from these documents.
- Highly optimised classifier training process to reduce load of having many hierarchies.
- Highly optimised classification process to reduce load of having many documents and many hierarchies.

5 Testing

5.1 Introduction

This chapter details the testing that was undertaken, and proposed further testing to be conducted on the system.

Firstly tests were conducted to verify the functionality of the system. Secondly tests were conducted to determine its effectiveness.

5.2 Unit testing

Thanks to the modular design of the system, it was possible to perform some tests on each part of the system individually after it had been implemented by creating simple test harness classes.

This provides basic tests to ensure that each component of the system is bug free and that it functions as its specification states.

For example, the startup and shutdown of classes using threads were tested and plugins were tested to ensure they perform correctly on predefined test data (such as a predefined list of test terms for a filter plugin).

5.3 Integration testing

Once all components of the system were completed, an integration test was conducted by running the application and conducting a number of test scenarios designed to simulate both normal operation and error conditions.

This ensures that the components of the system integrate correctly together and the system as a whole is bug free.

5.3.1 Document Acquisition Process

Tests of the document acquisition process and plugins were conducted by creating a test acquisition class which submits a file on disk into the system. The file was a short text file, designed to be easily checked manually.

Firstly, the summarisation process was tested by ensuring that the term frequency given to each word in the test document was correctly set to the number of occurrences divided by the total number of words.

Secondly, each plugin was added to the system and tested in turn - the stopword filter was verified to correctly remove words on the stopword list, the Porter stemmer was verified to correctly stem words in the document, the lower case filter was verified to correctly convert words to lower case.

Finally, the term bias averaging feature was tested by creating a test plugin to assign a bias to particular words and the averaged term bias was compared with the results of applying the algorithm manually.

5.3.2 Term Stats Update Process

The term statistics updating process was verified by creating documents with a small number of terms, the calculated document frequency of each term was calculated manually and compared with the system result.

The document term statistics updating process was also verified, by checking that the results for a number of terms by hand. The algorithm was also tested to ensure updates only occur when necessary by manually adjusting the timestamp on rows to ensure that flagging a row as needing update causes only one row to be updated, and that this row was updated only once.

5.3.3 Classifier Training Process

The classifier training process was verified by pinning a single document to a node and running the node training data update algorithm. It was then verified that the node’s feature vector was identical to the document’s feature vector.

The outer join used in the algorithm was verified by deleting some terms from the document and some from the node feature vectors to ensure that three cases were tested – where the term was present only in the node feature vector, only in the document feature vector and present in both. After executing the algorithm again it was verified that both feature vectors were the same.

The hierarchical propagation was then tested by creating a parent category for the node previously tested and ensuring that the subtree feature vector was correctly calculated by the subtree training data update algorithm. This was then further tested by creating simulating the hierarchy shown in Figure 25 and verifying that the algorithm operates in the predicted way.

Additional documents were then added to the system and the above tests were re-run with multiple documents and categories.

The training was also tested with multiple hierarchies to ensure that each hierarchy's training data was kept correctly isolated.

5.3.4 Classification Process

The classification process was tested initially by creating a hierarchy with a number of levels but only one node with training data, it was verified that the algorithm correctly recurses down the tree and classifies all the documents as part of that node.

Next two simple documents which were identical apart from a small number of words were put into separate nodes as training data, it was then verified that documents identical to the training data were placed in the category which was trained with the identical document.

The classifier was also tested to correctly classify each document exactly once per hierarchy present.

5.3.5 Web Crawler Plugin

The web crawler plugin was tested to correctly handle and log in the database a number of error cases such as 404 document not found, response timeouts and disallowed MIME Types. It was also tested on cases such as HTTP redirects and "Document not modified" responses.

"Crawling" was tested by ensuring that the web crawler correctly received (from the decoder plugin), and subsequently queued for crawling, all links found in a given test document.

The concurrency and rate limiters were tested to correctly block and wait in cases where the limit had been exceeded.

5.4 Effectiveness testing

Effectiveness testing was conducted to determine how suitable the system is in processing a collection of documents of the intended size.

Note that all tests were run with the intranet web server on the same network as the test machine. However, due to a Microsoft SQL Server database not being available locally, the database connection was made via VPN from Manchester to EDF's network in Holland, hence all database I/O was much slower than would be normal with the database on the same network as the test machine. Hence all test results can be taken to be a worst-case indication and expected to improve when run with a local database server.

Initial testing simply left the crawler running for long periods of time, and retrieved over 20,000 documents from the departmental intranet. A number of problems due to database locking and query optimisation were discovered due to this, and resulting from this some of the optimisations discussed in the previous section, for example vertically fragmenting the `Term` table were introduced.

After this, all major queries were run through the SQL server execution plan analyser so that a large volume of data was present when checking execution plans - the execution plan takes account of statistics generated from data, and as such is not accurate without a realistic volume of data in the system. This information was used to create additional indexes on columns that were found to be reducing query performance.

Once the system performance had been tuned, formal testing was conducted. All figures below are quoted to 3 significant figures.

5.4.1 Document Acquisition Scalability

Document acquisition effectiveness was measured as an end-to-end benchmark of the system's document acquisition capability. This measures how many documents the system is able to acquire via HTTP and then process them through MIME Type decoding, tokenisation, stopword lists, stemming and summarisation before finally storing them in the database.

The crawler was targeted at the entire School of Computer Science website and over three tests lasting half an hour, using a maximum of 7 concurrent HTTP connections, 10 simultaneously processed documents and a rate limit of 150 connections per minute (the rate limit was set high to prevent it from affecting statistics gathered), the total number of URLs crawled and the number of URLs successfully retrieving a document was noted.

Note the difference between total URLs crawled and URLs successfully retrieving a document is due to factors such as:

- Invalid links resulting in 404 document not found errors.
- Redirects where the web server replies that the document has been moved to a different URL.
- MIME-Types not understood by the set of decoder plugins available.
- Connection timeouts.
- TCP/IP errors.
- HTTP errors.

However these are still counted since it provides an indication of the web crawler's performance in a real environment.

Between each test the web crawler was stopped and started again but the database was not reset. This better simulates real conditions since the database will run more slowly as the number of documents it contains grows.

	Test 1	Test 2	Test 3	Average
Total Processed (Documents)	1584	5478	3483	3515
Total Successful (Documents)	1266	2046	1611	1641
Processing Rate (Documents/s)	0.88	3.04	1.935	1.953
Successful Processing Rate (Documents/s)	0.703	1.137	0.895	0.912

Scaling the average URL processing rate up to the target repository sizes:

Time to crawl documents	Value
Rate (seconds/URL)	1.953
3,000 documents projection (hours)	**1.628**
30,000 documents projection (hours)	**16.275**

This figure is quite acceptable since it is unlikely documents would need to be re-indexed more than daily, and as a result the web crawler has been set not to crawl documents more than once every 24 hours. Additionally this process would be faster partially due to the slow test database connection and partially due to the fact that this test only processes new documents, so none of the *If-Modified-Since* optimisations are used.

5.4.2 Term Statistics Updating

Note: These two processes are not currently used by the system, this data is made available for future plugins to use. Therefore little effort has been made to optimise these other than to ensure that they do not slow down system-critical processes (at the expense of the speed of these processes).

The efficiency of the two term statistics update procedures was tested by adding instrumentation to the system which uses the .NET Tick counter (a high resolution timer) to time a batch of queries. The total time to execute the batch is then divided by the number of operations to get a rate. The instrumentation was set to log data into a .CSV file which can then be imported into Microsoft excel and the timings analysed. To get a true indication of performance times this logging was operated over a long period of using the system, and the rates averaged.

Term Statistics Update

Firstly the term statistics update process was examined. The speed of this process is dependant on the number of terms in the database, i.e. the system's vocabulary, and the number of documents in the system. In the test system this was 6578 words.

Over 30 updates, this took on average 6.738 x 10^{-5} seconds/document/term, with a standard deviation of 7.987 x 10^{-6}. A worst-case projection is shown below assuming a vocabulary of 10,000 terms and a repository size between 3,000 and 30,000.

Time to update *term statistics*	Value
Rate (seconds/document/term)	6.738 x 10^{-5}
10,000 terms projected rate (seconds/document)	0.674
3,000 documents projection (minutes)	**33.691**
30,000 documents projection (hours)	**5.615**

This is quite acceptable since this is a periodic cached data update process which does not block other processes, used only by the subsequent calculation of TF-IDF values. Additionally this process is not currently used by any parts of the system, and could easily be optimised further by updating a number of terms in a batch rather than one per execution of the stored procedure. Unfortunately because the process runs on the entire database at once it is not optimised to isolate just newly added documents. Additionally note this process would be much faster with a database server on the same network.

Document Term Statistics Update

Secondly the document term statistics update process was examined. The speed of this process is dependant on the number of documents in the database and the number of terms that each document contains.

In the test system a document vector contained an average of 175 terms, with a standard deviation of 256.5 terms.

In the test system over 10 iterations of updating 500 terms this took on average 0.042 seconds/term, with a standard deviation of 0.004. A worst-case projection is shown below for average sized document vectors and a repository size between 3,000 and 30,000.

Time to update *document term statistics*	Value
Rate (seconds/term)	0.042
Average-sized document rate (seconds/document)	7.312
3,000 documents projection (hours)	**6.093**
30,000 documents projection (hours)	**60.934**

This shows the time that the *entire* document collection could have its TF-IDF and TF-LogIDF vectors. Because only newly submitted documents need processing this figure is reasonably acceptable, however this process would definitely be a candidate for further optimisation. Additionally note this process would be much faster with a database server on the same network.

5.4.3 Hierarchical PrTFIDF Training

Again the following tests were conducted by adding instrumentation to the system, set to log data into a .CSV file which was then imported into Microsoft excel analysed.

Node Training Data

Firstly the node training data update process was examined. The speed of this process is dependant on the average number of documents pinned to a node and the size of each document's vector.

Over 50 node updates, with 10 documents pinned to each node, this took on average 1.771 seconds/node, with a standard deviation of 0.109. Assuming each hierarchy has 40 nodes, the following projection can be made:

Time to update *node training data*	Value
Rate (seconds/node)	1.771
Projected rate (seconds/hierarchy)	**70.84**

Hence each reasonably-sized hierarchy would require about a minute to have its node probabilities updated. Also note this is a worst-case figure – the operation normally only happens on nodes which been modified, so simple updates would take about 2 seconds.

Subtree Training Data

Secondly the subtree training data update process was examined. The speed of this process is dependant on the depth of the hierarchy and the size of each document's vector.

Over 10 hierarchy updates, with a hierarchy depth of 6 nodes, this took on average 35.479 seconds/node updated, with a standard deviation of 3.609. In the previously assumed 40 node hierarchy the first update to establish subtree probabilities would therefore require 23.652 minutes. However after this, assuming an average depth of 5 nodes, updates would require 2.957 minutes.

This is a reasonable figure since hierarchy alterations are likely to happen infrequently once established, and when an update is made the number of calculations made is minimised.

Precalculation of posterior probabilities

Thirdly the posterior probability precalculation process was examined. The speed of this process is dependant on the number of nodes in the hierarchy and the size of each node's feature vector

Over 10 precalculations, in a hierarchy of 40 nodes, this took on average 1.096 seconds per hierarchy, with a standard deviation of 0.517.

This is clearly a very acceptable figure, however some problems due to deadlock between this process and the previous two processes was encountered.

5.4.4 Classification

Again the following tests were conducted by adding instrumentation to the system, set to log data into a .CSV file which was then imported into Microsoft Excel to be analysed.

In the previously described 40 node hierarchy, 1000 document classifications took on average 0.161 seconds/document, with a standard deviation of 0.003. For the target repository size, the following projection can be made:

Time to classify documents	Value
Rate (seconds/hierarchy/document)	0.161
3,000 documents projection (minutes/hierarchy)	**8.05**
30,000 documents projection (hours/hierarchy)	**1.342**

This shows the time required to re-classify all documents (for example after a change to the hierarchy required a complete re-classification of all documents). However in the case of adding a new document only 0.161 seconds would be required per hierarchy.

5.4.5 Conclusion

The tests shown above prove that the system is capable of handling a document repository of the target size with reasonable delays in document acquisition, reclassification etc. However it also shows that if users are allowed to create large numbers of hierarchies, scalability would potentially be a problem if large numbers of users changed their hierarchies simultaneously.

It is probably reasonable to assume that for a small to medium number of hierarchies it would be unlikely that several users would change their hierarchies simultaneously, and if they did a delay of several minutes to hours for updates to propagate may be acceptable.

In the case of new document submissions the number of hierarchies is unlikely to be a problem since the document acquisition and classification process can operate very quickly. Hence providing users make few changes to their hierarchy once initially set up the number of hierarchies would not be a limiting factor.

5.5 Further Testing

The algorithms have been shown to perform correctly on a small scale, and have been tested on a larger scale to find points requiring further optimisation.

However there has not been enough time to test the quality of the classification on a large scale, since it is very time consuming to build up suitable training data for a very large collection of documents. This is a key factor in usability of the system and further testing should aim to build up a realistic collection of documents, encourage a number of users to create hierarchies and then obtain feedback from users as to the quality of automatic classification after sufficient training. This testing should be conducted over the long-term and ensure that the system remains stable in the long-term and that new documents are accurately classified with a minimum of training.

5.6 Summary

The testing has shown that the system operates correctly over a small test collection of documents. It has also verified the scalability of the algorithm over a large collection of documents.

However, within the timeframe available it was not possible to conduct a large-scale evaluation over a large collection of documents with a number of users maintaining personal hierarchies. Further testing is necessary to ensure the system operates correctly under these circumstances and to outline any improvements necessary.

6 Evaluation

6.1 Introduction

This section evaluates the project in terms of its goals, and attempts to evaluate how well each goal has been fulfilled.

6.2 Summary of Goals

In section 1.3 the goals of the project were defined to be producing a system which fulfils the following criteria:

- **Hierarchical**
- **Browsable**
- **Automated**
- **Personalisable**

The system that has been designed fulfils these goals. By testing it has been verified that the system implementation correctly operates as designed.

Unfortunately in the personalisation step it was not possible to come up with a solution for linking user's personalised hierarchies to the canonical hierarchy within the timeframe available due to the huge complexity of achieving this. However the requirement for personalisation has been fulfilled with a less complex method of cloning hierarchies. Moreover this less complex method is more robust and affords the user greater control over their personal hierarchy.

Some issues were encountered during implementation and testing particularly with respect to database locking and concurrency, but it is believed that all of these issues have now been resolved. However large-scale tests are necessary to fully verify this.

The plugin architecture proposed is a trade-off between extensibility and speed, however in the case of an intranet document repository the number of documents requiring classification is relatively small, so speed is not as much of an issue as it would be in an internet system. Scalability testing undertaken so far has proven that the system is more than capable of handling the specified repository sizes.

7 Conclusion

7.1 Limitations

This section identifies the limitations of the system and highlights possible further work.

As highlighted previously, this system makes no effort to connect changes in the canonical hierarchy with the user's hierarchy. Future enhancements to the system could either inform the user of changes to the canonical hierarchy to prompt them to change their personal hierarchy, or a system for linking the two hierarchies as outlined in section 3.2.4 could be utilised. However a system like this presents a number of difficult challenges.

The web crawler does not currently support the *robots.txt* exclusion standard which allows the administrator of a website to prevent web crawlers from visiting certain parts of the site. For the system to be used to crawl any websites not controlled by the administrator of the system, the robots.txt exclusion standard ideally should be supported.

From the testing, some of the processes involved in the system have been shown to be slow. A lot of this is due workarounds to reduce concurrency in the database backend. Most commercial web crawlers such as Google utilise custom-built data storage which is optimised for the operations performed. The best possible improvement of this system to increase the speed of the various processes would probably be to replace the backend database with a custom-built data storage system.

7.2 Extensions

One of the major extensions to the system noted previously would be the use of a clustering algorithm to bootstrap the initial hierarchy. The major problem with bootstrapping in this way is the danger of *overfitting*. This is where the classifier is fitted to the training data in such a way that it performs perfectly over the training data, but is unable to classify any new examples correctly.

Additionally there are a number of additional extensions possible in terms of plugins, for example collecting bigram collocations, the creation of a WordNet stemming plugin, or the creation of a WordNet plugin to add hypernyms to the terms contained in a document.

Peng and Choi [32] also propose a solution for automatically detecting the need for a new category upon the arrival of documents significantly different to the categories existing in the hierarchy, this would possibly be another useful extension to the project to reduce requirements for hierarchy maintenance.

Finally, it is interesting to note that a browsing-based system such as this has the effect of increasing the novelty ratio (equation (4) in section 2.2.2.2) for a user trying to find information compared with a keyword search since documents are found by concept rather than by a query. A useful extension of the system would be to incorporate a keyword search, using the TF-IDF data already present in the database. Results retrieved could allow the user to either browse into the category containing the result, or jump directly to the document.

7.3 Summary of achievements

A system has been constructed and partially tested which fulfils the initial goals set. The system builds upon existing research, but implements the algorithms in a highly optimised and relational database specific manner to maximise the efficiency of the system. It provides for user-personalisation through the cloning and subsequent personalisation of hierarchies.

The system supports plugins for future administrators to be able to modify the operation of the system and extend the system without modifying the code of the system itself.

Through testing the system has been shown to be scalable enough to cope with the sizes of document repository targeted. The system should be capable of being deployed commercially after a phase of further detailed testing.

8 References

[1] http://en.wikipedia.org/wiki/Web_portal, 06/07/2005.

[2] http://news.netcraft.com/archives/web_server_survey.html, 10/08/2005.

[3] http://www.yahoo.com/, 06/07/2005.

[4] http://www.sapdesignguild.org/editions/edition3/portal_definition.asp, 06/07/2005.

[5] Belkin, N. J., Croft, W. B. (1992)
Information filtering and information retrieval: two sides of the same coin?
Communications of the ACM, 35(12), 29-38.

[6] Rijsbergen, C. J. van (1979)
Information Retrieval.
London: Butterworths.

[7] Kosala, R., Blockeel, H. (2000)
Web Mining Research: A Survey.
SIGKDD Explorations, 2(1), 1-15.

[8] Etzioni, O. (1996)
The World-Wide Web: Quagmire or Gold Mine?
Communications of the ACM, 39 (11), 65-68.

[9] http://www.database.cis.nctu.edu.tw/seminars/2003F/TWM/slides/p.ppt, 10/08/2005.

[10] http://www.edf.nl/, 10/08/2005.

[11] Porter, M. F. (1980)
An algorithm for suffix stripping.
Program; automated library and information systems, 14(3), 130-137.

[12] http://wordnet.princeton.edu/, 31/08/2005.

[13] Sedding, J., Kazakov, D. (2004)
WordNet-based Text Document Clustering.
Proceedings of the Third Workshop on Robust Methods in Analysis of Natural Language Data (ROMAND), 104-113.

[14] Hotho, A., S. Staab, S., Stumme, G. (2003)
Wordnet improves text document clustering.
Proceedings of the SIGIR 2003 Semantic Web Workshop.

[15] Salton, G., Buckley, C. (1988)
Term-Weighting Approaches in Automatic Text Retrieval.
Information Processing and Management: an International Journal, 24(5), 513-523.

[16] Raghavan, V. V., Wong, S. K. M. (1986)
A Critical Analysis of the Vector Space Model for Information Retrieval.
Journal of the American Society for Information Science, 37(5), 279-287.

[17] Wong, S. K. M., Ziarko, W., Raghavan, V. V., Wong, P. C. N. (1986)
On extending the vector space model for Boolean query processing.
Proceedings of the 9th annual international ACM SIGIR conference on Research and development in information retrieval, 175-185.

[18] Zipf, H. P. (1949)
Human Behaviour and the Principle of Least Effort.
Cambridge, Massachusetts: Addison-Wesley.

[19] Luhn, H. P. (1958)
The automatic creation of literature abstracts.
IBM Journal of Research and Development, 2, 159-165.

[20] Cooley, R., Srivastava, J., Mobasher, B. (1997)
Web Mining: Information and Pattern Discovery on the World Wide Web.
Proceedings of the 9th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'97).

[21] Manning, C. D., Schütze, H. (2003)
Foundations of Statistical Natural Language Processing.
Cambridge, Massachusetts: The MIT Press.

[22] http://en.wikipedia.org/wiki/Classifier_%28mathematics%29, 03/09/2005.

[23] http://www.google.com/, 03/09/2005.

[24] Page, L., Brin, S., Motwani, R., Winograd, T. (1998)
The PageRank citation ranking: Bringing order to the web.
Technical Report, Stanford Digital Libraries.

[25] http://en.wikipedia.org/wiki/Google_bomb, 03/09/2005.

[26] http://en.wikipedia.org/wiki/Statistical_classification, 03/09/2005.

[27] Manning, C., Raghavan, P. (2004)
Text Retrieval and Mining.
Stanford University Course CS276A Lecture Slides
(http://www.stanford.edu/class/cs276a/syllabus2004.html, 03/09/2005).

[28] http://en.wikipedia.org/wiki/Naive_Bayes_classifier, 04/09/2005.

[29] http://webmonkey.wired.com/webmonkey/templates/print_template.htmlt?meta=/webmonkey/geektalk/97/12/index4a_meta.html, 04/09/2005.

[30] http://en.wikipedia.org/wiki/Text_mining, 04/09/2005.

[31] Joachims, T. (1997)
A Probabilistic Analysis of the Rocchio Algorithm with TFIDF for Text Categorization.
Proceedings of the 14th International Conference on Machine Learning (ICML-97).

[32] Peng, X., Choi, B. (2002)
Automatic Web Page Classification in a Dynamic and Hierarchical Way.
Second IEEE International Conference on Data Mining (ICDM'02), 386-393.

[33] Rocchio, J. J. (1971)
Relevance feedback in Information Retrieval.
The SMART Retrieval System – Experiments in Automatic Document Processing (Salton, G. ed.), Englewood Cliffs: Prentice Hall, 313-323.

[34] Manning, C., Raghavan, P. (2005)
Web Search and Mining.
Stanford University Course CS276B Lecture Slides
(http://www.stanford.edu/class/cs276b/syllabus.html, 07/09/2005).

[35] http://subversion.tigris.org/, 07/09/2005.

[36] http://blogs.msdn.com/brada/articles/361363.aspx, 07/09/2005.

[37] http://msdn.microsoft.com/library/d/fault.asp?url=/library/en-us/dndotnet/html/progthrepool.asp, 08/09/2005.

[38] http://support.microsoft.com/default.aspx?scid=kb;en-us;830118, 09/09/2005.

[39] http://ndoc.sourceforge.net/, 09/09/2005.

[40] http://www.w3.org/CGI/, 11/09/2005.

[41] Fuhr, N. (1989)
Models for retrieval with probabilistic indexing.
Information Processing and Management: an International Journal, 25 (1), 55-72.

[42] Miller, G., Beckwith, R., Fellbaum, C., Gross, D. (1990)
Introduction to WordNet: an on-line lexical database.

International Journal of Lexicography, 3(4), 235-244.

Implementation Details

I.1 Introduction

This section provides an overview of all the major classes in the system implementation:

- Firstly, the system framework is discussed. This is the set of supporting classes which make up the core components of the system and connect the plugins together.
- Secondly, the set of plugins which have been provided with the initial implementation of the system are discussed.

The system has been named "Dagama" and all classes are inside a similarly named namespace.

For a more detailed account of all classes and the methods, properties and variables each provide, NDoc [39] format documentation generated from XML comments is provided at http://www.base6.com/dagama/.

I.2 System Framework

This section provides a summary of all major classes in the system framework implementation and their function.

Acquisition Management (Dagama.Acquire)

`Dagama.Acquire.AcquireManager`

This class manages the entire acquisition and document summarisation process. It implements the singleton design pattern. On startup it initialises the document processor pool to prepare for the processing of documents and then the acquisition manager to begin acquiring documents. On shutdown it stops both of these.

`Dagama.Acquire.DocProcessorPool`

This class manages the pool of document processing threads and implements the *`IAcquire`* interface. On startup it creates a *`ConcurrencyLimiter`* to limit the number of concurrent documents being processed and calls the static *`GlobalInit`* function of the *`DocProcessor`* class. On shutdown it calls the *`GlobalFree`* function of the *`DocProcessor`* class.

Upon notification that a document has been updated or deleted, a new slot is requested from the *ConcurrencyLimiter* and once the slot has been obtained it creates a new *DocProcessor* object to process the document and queues its processing function for execution on the *ThreadPool*.

.NET uses a database connection pool which reduces the need to repeatedly disconnect and reconnect to the database, the idea being instead of being closed a connection is returned to the pool after use, and a new connection will be supplied from the pool instead of making another connection to the database. During testing the application was exhausting the connection pool causing errors, so this class was re-engineered to use a fixed array of *Thread* objects (instead of using the *ThreadPool*) each of which each hold a permanent database connection. This is less efficient but appeared to reduce the problem. Later it was discovered that this problem is actually caused by a bug in Microsoft Visual Studio .NET's debugger [38], and the design was reverted to use the *ThreadPool* again.

Configuration Settings

Name	Value
MaxConcurrentSubmissions	The maximum number of documents to process concurrently before forcing new submissions to queue.

Dagama.Acquire.DocProcessor

This class manages the processing of an individual document. It holds a global (static) instance of the *DecoderManager*, *FilterManager*, *TokeniserManager* and *DocumentSummariser*. *GlobalInit* and *GlobalFree* functions initialise and destroy these classes on startup/shutdown.

An *UpdateDocumentCallback* function is provided to process new documents (which are decoded, tokenised, filtered and summarised before being passed to the DAL to update in the database). A *DeleteDocumentCallback* function is also provided to pass a document to the DAL to be deleted from the database.

Acquisition Plugins (Dagama.Acquire.AcquistionPlugin)

Dagama.Acquire.AcquisitionPlugin.AcquisitionManager

This class manages all acquisition plugins. On startup it sends a start message to all registered plugins and on shutdown it sends a shutdown message to all running plugins. Each plugin is also provided with an *Dagama.Acquire.IAcquire* interface which provides methods to submit and remove documents from the system.

Configuration Settings

Name	Value
RegisteredPlugin[0..n]	The fully qualified name of an acquisition plugin to be instantiated by the framework.

Dagama.Acquire.AcquisitionPlugin.IAcquire

Acquisition system interface. Provides methods to submit and remove documents from the system.

Dagama.Acquire.AcquisitionPlugin.IAcquisitionPlugin

Interface which all document acquisition plugins must implement. Has start/stop methods to start and stop the plugin.

Decoder Plugins (Dagama.Acquire.DecoderPlugin)

Dagama.Acquire.DecoderPlugin.DecoderManager

This class manages all decoder plugins. On startup it instantiates all registered plugins and on shutdown it destroys all registered plugins.

Has a decode document method which accepts a raw document and an *ITokeniser* object, and searches registered plugins for a plugin capable of decoding it and returns the decoded array of terms. Throws an exception if no plugin was found.

The decoder plugin is also supplied with a link callback function passed from the acquisition plugin. This allows the acquisition plugin to be notified of new URIs discovered in the documents (for example in the case of a web crawler this allows newly discovered documents to be crawled).

Configuration Settings

Name	Value
RegisteredPlugin[0..n]	The fully qualified name of a decoder plugin to be instantiated by the framework.

Dagama.Acquire.DecoderPlugin.IDecoderPlugin

Interface which all decoder plugins must implement. Provides two methods, one which returns a Boolean specifying whether a stated MIME Type can be decoded by this plugin, one which performs decoding of a document given a document and an *ITokeniser* object.

Filter Plugins (Dagama.Acquire.FilterPlugin)

Dagama.Acquire.FilterPlugin.FilterManager

This class manages the chain of filter plugins. On startup it instantiates all registered plugins and on shutdown it destroys all registered plugins.

Has a filter method which accepts the array of decoded tokens and passes it to each filter plugin in turn for modification. The array is actually an *ITokenList* object, ensuring that only a reference to the array is passed around (as opposed to passing copies around) in order to reduce memory requirements.

Configuration Settings

Name	Value
RegisteredPlugin[0..n]	The fully qualified name of a filter plugin to be instantiated by the framework. Documents are passed to each plugin in order from *RegisteredPlugin0* to *RegisteredPluginN*.

Dagama.Acquire.FilterPlugin.IFilterPlugin

Interface which all filter plugins must implement. Has a filter method which accepts an *ITokenList* interface through which to enumerate and modify terms.

Document Summariser (Dagama.Acquire.Summariser)

Dagama.Acquire.Summariser.DocumentSummariser

Converts an array of tokens into a "bag of words" model list of tokens and occurrences. This is achieved by using a quick sort algorithm to sort the list alphabetically and then stepping through the list counting occurrences of each term and averaging term bias.

Tokeniser Plugin (Dagama.Acquire.TokeniserPlugin)

Dagama.Acquire.TokeniserPlugin.TokeniserManager

This class manages the chain of tokeniser plugins. Tokeniser plugins use the factory design pattern (to enable each individual tokeniser to maintain state whilst tokenising a given document), and on startup this class instantiates the registered tokeniser factory and on shutdown it destroys the tokeniser factory.

Has a *GetTokeniser* method which obtains a new *ITokeniser* object from the registered factory.

Configuration Settings

Name	Value
RegisteredPlugin	The fully qualified name of the tokeniser factory plugin to be instantiated by the framework.

Dagama.Acquire.TokeniserPlugin.ITokeniserFactory

Interface which all tokeniser factory plugins must implement. This has a *GetTokeniser* method which must return a new *ITokeniser* object.

Dagama.Acquire.TokeniserPlugin.ITokeniser

Interface which all tokenisers must implement. This has a *IsTokenBoundary* method which accepts a character and returns True if it is a token boundary.

Acquisition Data Types (Dagama.Acquire.Types)

Provides a number of supporting data type classes, for example *ITokenList*.

Classifier (Dagama.Classifier)

Dagama.Classifier.ClassifierTrainUpdateNodeProbability

Derived from *ContinuousThreadBase*, causes the DAL to periodically check for node training vectors requiring update.

Configuration Settings

Name	Value
MaxNoUpdateSleepDelaySeconds	Maximum delay for the *ContinuousThreadBase* truncated binary exponential backoff algorithm to wait if no training vectors need updating.

Dagama.Classifier.ClassifierTrainUpdateSubtreeProbability

Derived from *ContinuousThreadBase*, causes the DAL to periodically check for subtree training vectors requiring update.

Configuration Settings

Name	Value
MaxNoUpdateSleepDelaySeconds	Maximum delay for the *ContinuousThreadBase* truncated binary exponential backoff algorithm to wait if no training vectors need updating.

Dagama.Classifier.ClassifyCalcPosteriorProbabilities

Derived from *ContinuousThreadBase*, causes the DAL to periodically check for hierarchies requiring their precalculated posterior probabilities to be updated.

Configuration Settings

Name	Value
MaxNoUpdateSleepDelaySeconds	Maximum delay for the *ContinuousThreadBase* truncated binary exponential backoff algorithm to wait if no documents need classifying.

Dagama.Classifier.ClassifierClassify

Derived from *ContinuousThreadBase*, causes the DAL to periodically check for documents requiring classification.

Configuration Settings

Name	Value
MaxNoUpdateSleepDelaySeconds	Maximum delay for the *ContinuousThreadBase* truncated binary exponential backoff algorithm to wait if no documents need classifying.

Dagama Service (Dagama.Service)

Dagama.Service.DagamaService

This is the framework's startup class. It is written as a Microsoft Windows system service which allows it to run in the background on any windows machine and to be started and stopped automatically.

The class simply communicates windows start/stop requests to *Dagama.Acquire.AcquireManager*, *Dagama.Stats.StatsManager* and *Dagama.Classifier.ClassifierManager*.

Statistics Updater (Dagama.Stats)

Dagama.Stats.TermStatsUpdaterThread

Derived from *PeriodicThreadBase*, causes the DAL to periodically re-calculate both the document frequency and inverse document frequency of known terms.

Configuration Settings

Name	Value
UpdateIntervalMinutes	How often to update the term statistics in minutes.

Dagama.Stats.DocumentTermStatsUpdaterThread

Derived from *ContinuousThreadBase*, causes the DAL to periodically check for document term statistics (*TF-IDF* and *TF-LogIDF*) requiring update.

Configuration Settings

Name	Value
MaxNoUpdateSleepDelaySeconds	Maximum delay for the *ContinuousThreadBase* truncated binary exponential backoff algorithm to wait if no document term statistics need updating.

Thread Management (Dagama.Threading)

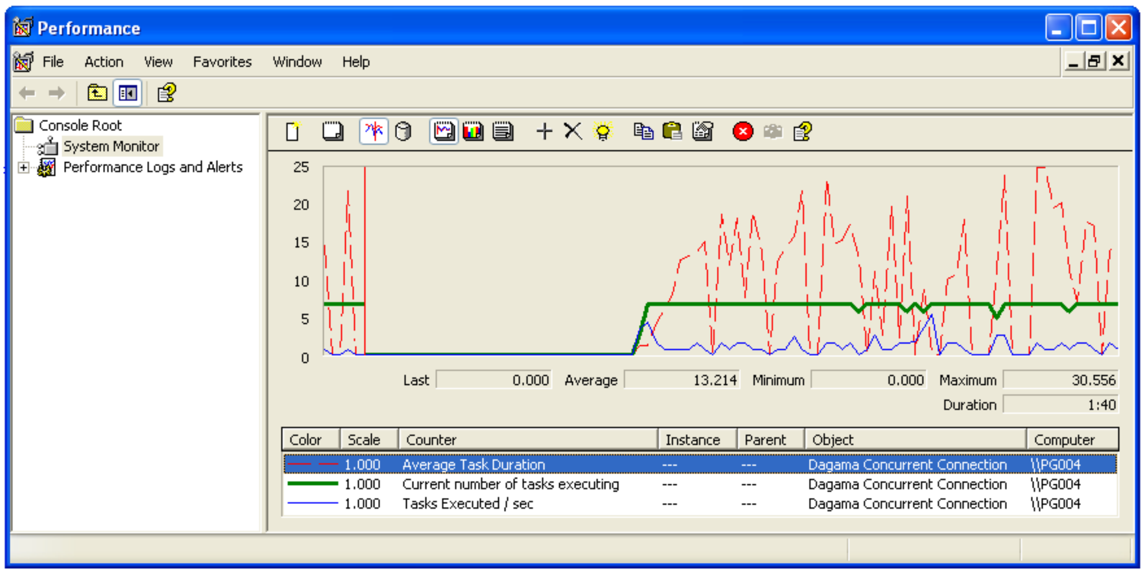
Dagama.Threading.ConcurrencyLimiter

This class supports the limiting of the number threads executing a particular task. This is achieved by having a counter which is checked and incremented/decremented inside a *mutex* block (to ensure only one thread can access the counter at a time). If the maximum number of slots are currently in use, the *mutex* is released and then the thread blocks on an *AutoResetEvent* object.

When a slot is freed, the *AutoResetEvent* object is signalled, releasing exactly one waiting thread which then fills the slot. The advantage of this is it maximises efficiency since no timers are required to repeatedly poll for a free slot, the thread simply sleeps until one being freed wakes it.

This class also creates performance monitors to show the current number of concurrent tasks executing and the average duration and rate of execution.

This allows the current system activity to be currently viewed, for example the number of web pages the web crawler is currently downloading or the number of documents currently being processed.



Last 0.000 Average 13.214 Minimum 0.000 Maximum 30.556
Duration 1:40

Color	Scale	Counter	Instance	Parent	Object	Computer
🟥	1.000	Average Task Duration	---	---	Dagama Concurrent Connection	\\PG004
🟩	1.000	Current number of tasks executing	---	---	Dagama Concurrent Connection	\\PG004
🟦	1.000	Tasks Executed / sec	---	---	Dagama Concurrent Connection	\\PG004

Figure 26 Concurrency limiter performance counters

Dagama.Threading.ContinuousThreadBase

Derives from *ThreadBase*. Abstract base class for a thread which is designed to repeatedly execute a *DoWork* function (which is overridden). The *DoWork* function returns true if work was found, false if it was not. If false is returned, the thread sleeps using a truncated binary exponential back off algorithm, so initially it would sleep for 1 second, then 2 seconds, then 4, then 8, then 16 etc up to a defined maximum. The purpose of this is to reduce the load if the task that is continuously being executed is not doing anything (For example polling for documents to be classified but none are being added).

Dagama.Threading.PeriodicThreadBase

Derives from *ThreadBase*. Abstract base class for a thread which is designed to execute a *DoWork* function every *n* seconds. Rather than sleeping for *n* seconds after the *DoWork* function has been executed, the duration which the *DoWork* function took to execute is calculated, and the thread sleeps *n – duration*. This ensures that if the *DoWork* function is taking longer than the specified duration to execute it will be run continuously.

Dagama.Threading.RateLimiter

This class is designed to limit the rate at which a task is being executed. It achieves this by monitoring the number of tasks executed over a given window, for example enforcing a maximum of 5 tasks/minute averaged over the last 10 minutes.

If the current rate is over the acceptable maximum, the thread sleeps until the next logged task run is due to be expired (i.e. has become older than the current time minus the window length). The average over data currently within the window is recomputed within a *mutex* block for thread safety.

Dagama.Threading.ThreadBase

Abstract base thread class, provides an abstract *Run* function to be overridden, also *StartThread*/*StopThread* functions which manage the startup/shutdown of the thread and a *KeepRunning* function for the *Run* function to poll to see whether it should shut down (calling *StopThread* also causes the thread to be awoken if it is sleeping, if it does not shut down within 5 minutes it is terminated).

Configuration Manager (Dagama.Configuration)

Provides a *ConfigurationManager* class which allows configuration settings to be loaded from the XML application config file.

Data Access Layer (Dagama.Data)

The Data Access Layer separates data access code from the main application.

Dagama.Data.DataFactoryManager

This class which implements the singleton design pattern and provides one method to return the registered data factory object.

Configuration Settings

Name	Value
RegisteredPlugin	The data factory plugin to instantiate on startup.

Data Access Layer Interfaces (Dagama.Data.Interfaces)

Provides one factory interface and a number of data service interfaces. Each database-specific implementation must implement all of these.

SQL Server Data Access Components (Dagama.Data.Sql)

`Dagama.Data.Sql.SqlClassifier`

Invokes stored procedures to update the classifier training data. One stored procedure updates node training data on nodes which have either had their training set altered, or where documents in the training set have changed. When training data is updated in this way a datestamp is set to the current time. The other stored procedure checks for nodes where this datestamp is more recent than the date the node's probability was updated. It processes nodes closest to the root of the tree first and recurses down to the leaves of the tree calculating the subtree probability and node counts using the hierarchical PrTFIDF algorithm.

Also contains a procedure to classify documents.

`Dagama.Data.Sql.SqlConn`

Manages database connections.

Configuration Settings

Name	Value
`ConnectionString`	The SQL Server connection string to use.

`Dagama.Data.Sql.SqlDocument`

Provides methods to add, delete and update documents in the database.

`Dagama.Data.Sql.SqlStatsUpdate`

Provides a method to invoke a stored procedure update term statistics, namely document frequency and inverse document frequency. If these values change from the previously computed values a datestamp is updated which causes the document's term weightings to be recomputed.

A second method invokes the stored procedure to update the document's term weightings in the case that the term's document frequency has been changed by the first method or the term's term frequency has changed in the document.

`Dagama.Data.Sql.SqlStopWords`

Provides a method to retrieve a list of stopwords from the database.

`Dagama.Data.Sql.SqlWebCrawlerDocument`

Provides methods to add URLs and update the status of URLs in the web crawler queue. Also provides a method to dequeue a URL for crawling. This method uses SQL Server locking to prevent updates whilst the status of the URL is set to "in progress". The row is then unlocked but any other threads executing this method will ignore rows marked as "in progress", allowing web crawlers to be run on a cluster of machines.

Data Access Layer Types (Dagama.Data.Types)

Provides supporting abstract data representations for use in communication between the data access layer and the application.

Logging (Dagama.Logging)

Provides a common library for the logging of all exceptions and messages, complete with source information and a severity level.

I.3 System Plugins

The following plugins have been supplied with the initial implementation of the system.

Dagama.Acquire.AcquisitionPlugin.TestAcquire.TestAcquire

Test acquire plugin, on startup loads a file from disk and submits it into the system.

Dagama.Acquire.AcquisitionPlugin.WebCrawler.WebCrawler

Web crawler plugin, crawls documents using asynchronous I/O and the *ThreadPool*. Extends *ContinuousThreadBase*.

The web crawler uses a *ConcurrencyLimiter* and a *RateLimiter* to limit load placed on remote servers. It maximises network efficiency by discarding unsuitable MIME-Types after the HTTP header is received but before the response is sent, also uses HTTP/1.1 connection re-use to limit overhead of reconnecting to the same server for each request.

It uses the HTTP/1.1 *If-Modified-Since:* header to only download documents which have been updated. Also uses regular expressions to limit which URIs are crawled.

All URIs have their query string and bookmark information stripped, ensuring the crawler does not get stuck in any infinite loops caused by CGI scripts. The crawler does not obey the robots exclusion protocol since it is designed for intranet use, but this could easily be added later.

The web crawler handles redirects by marking the original URI as processed and then queuing the redirect URI in the database for crawling.

Configuration Settings

Name	Value
`HttpAcceptHeader`	HTTP *Accept:* header to be sent to remote servers.
`HttpRefererHeader`	HTTP *Referer:* (sic.) header to be sent to remote servers.
`HttpFromHeader`	HTTP *From:* header to be sent to remote servers.
`RequestTimeoutSeconds`	HTTP request timeout in seconds.
`RateMonitorWindowMinutes`	*`RateLimiter`* window in minutes.
`RateMonitorMaxConnectionsPerMinute`	*`RateLimiter`* maximum allowed number of connections/minute.
`MaxConcurrentConnections`	*`ConcurrencyLimiter`* maximum number of simultaneous *HTTP* connections.
`MaxEmptyQueueSleepDelaySeconds`	Maximum delay for the *`ContinuousThreadBase`* truncated binary exponential backoff algorithm to wait if no web pages are queued for crawling.
`MaxAcquireFailuresBeforeRemove`	Maximum number of failures to acquire a document before it is removed from the database.
`AllowedURI[0..n]`	List of regular expressions representing *URIs* which are allowed to crawl, anything not in this list will be ignored.
`AllowedMIMEType[0..n]`	List of regular expressions representing *MIME types* which are allowed to crawl, anything not in this list will be ignored.

`Dagama.Acquire.DecoderPlugin.TextPlain.TextPlain`

Decodes *text/plain* documents using the system tokeniser.

Dagama.Acquire.DecoderPlugin.TextHTML.TextHTML

Decodes *text/html* documents using the system tokeniser. Uses regular expressions to strip text out of the HTML document, and to strip all links, which are reported back to the acquisition plugin.

Dagama.Acquire.FilterPlugin.HyphenFilter.HyphenFilter

Filters out words consisting entirely of hyphens (since the tokeniser counts hyphens to be part of a word).

Dagama.Acquire.FilterPlugin.LowerCaseFilter.LowerCaseFilter

Converts all terms to lower case.

Dagama.Acquire.FilterPlugin.PorterStemmerFilter.PorterStemmerFilter

Replaces all terms with Porter-stemmed versions (hyphenated words have each part separated by hyphens stemmed individually).

Dagama.Acquire.FilterPlugin.StopWordfilter.StopWordfilter

Holds a sorted stopword list in memory, uses binary search to compare each term in the document with this list and removes words matching a word on the stoplist.

Dagama.Acquire.TokeniserPlugin.BasicTokeniserFactory

Creates *BasicTokeniser* objects.

Dagama.Acquire.TokeniserPlugin.BasicTokeniser

Ignores context information, simply treats any character not matching [\-A-Za-z0-9] (letters, numbers and '-') to be a token boundary.

THE UNIVERSITY OF MANCHESTER

ABSTRACT OF THESIS/DISSERTATION submitted by James William Furness for the Degree of Master of Science and entitled "A Personalisable Hierarchical Document Categoriser".

Month and Year of submission: September 2005.

A novel system is proposed for the indexing, searching and browsing of an intranet document repository for use as part of a corporate extranet. The system allows users to browse a hierarchically organised collection of documents. The hierarchy is automatically maintained by the system after a minimum of training. Users additionally have the option to personalise the hierarchy in order to organise the documents in any way they see fit. This document presents an overview of the design of the system and the reasons for the various design choices.

The system has been implemented and initial tests on the system have been conducted which show that the system would comfortably be able to handle a repository sized between 3,000 and 30,000 documents.