Integrating Information Retrieval & Neural Networks

by

Victoria J. Hodge.

A dissertation submitted in partial fulfillment of the
requirements for the degree
Doctor of Philosophy.

University of York,
Department of Computer Science.

Sept, 2001

Abstract

Due to the proliferation of information in databases and on the Internet, users are overwhelmed leading to *Information Overload*. It is impossible for humans to index and search such a wealth of information by hand so automated indexing and searching techniques are required. In this dissertation, we explore current Information Retrieval (IR) techniques and their shortcomings and we consider how more sophisticated approaches can be developed to aid retrieval. Current techniques can be slow due to the sheer volume of the search space although faster ones are being developed. Matching is often poor, as the quantity of retrievals does not necessarily indicate quality retrievals. Many current approaches simply return the documents containing the greatest number of 'query words'. A methodology is desired to: process documents unsupervised; generate an index using a data structure that is memory efficient, speedy, incremental and scalable; identify spelling mistakes in the query and suggest alternative spellings; handle paraphrasing of documents and synonyms for both indexing and searching; to focus retrieval by minimising the search space; and, finally calculate the query-document similarity from statistics autonomously derived from the text corpus. We describe our IR system named MinerTaur, developed using both the AURA modular neural system and a hierarchical, growing self-organising neural technique based on Growing Cell Structures which we call TreeGCS. We integrate three modules in MinerTaur: a spell checker; a hierarchical thesaurus generated from corpus statistics inferred by the system; and, a word-document matrix to efficiently store the associations between the documents and their constituent words. We describe each module individually and evaluate each against comparative data structures and benchmark implementations. We identify improved memory usage, spelling recall accuracy, cluster quality and training and recall times for the modules. Finally we compare MinerTaur against a benchmark IR system, SMART developed at Cornell University, and reveal superior recall and precision for MinerTaur versus SMART.

Contents

List of Figures

List of Tables

Acknowledgements

I wish to thank my supervisor Jim Austin for his invaluable assistance and support throughout my research. I would like to thank my assessor Suresh Manandhar for his feedback and assistance with my manuscripts.

I would also like to thank all members of the A.C.A.G. group, Department of Computer Science at the University of York, past and present for their invaluable comments and assistance with my work leading to this dissertation.

Author's Declaration

We include a list of our published papers below and indicate the chapters to which they are relevant.

1. V.J. Hodge & J. Austin,
   Hierarchical Growing Cell Structures: TreeGCS,
   In, Proceedings of the Fourth International Conference on Knowledge-Based Intelligent Engineering Systems, August 30th to September 1st 2000,

2. V.J. Hodge & J. Austin,
   Hierarchical Growing Cell Structures: TreeGCS,
   IEEE Transactions on Knowledge and Data Engineering, 13(2), Special Issue on Connectionist Models for Learning in Structured Domains, 2001,

3. V.J. Hodge & J. Austin,
   An Evaluation of Standard Retrieval Algorithms and a Weightless Neural Approach,
   In, Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks, 24-27 July, 2000,

4. V.J. Hodge & J. Austin,
   An Evaluation of Standard Retrieval Algorithms and a Binary Neural Approach,
   Neural Networks, 14(3), Elsevier Science Ltd, 2001,

5. V.J. Hodge & J. Austin,
   An Integrated Neural IR System.
   Proceedings of the 9th European Symposium on Artificial Neural Networks, Bruges, April 2001.

6. V.J. Hodge & J. Austin,
   A Novel Binary Spell Checker.
   In, Proceedings of the International Conference on Artificial Neural Networks (ICANN2001), Vienna, August 2001. Eds G. Dorffner, H. Bischof, & K. Hornik, LNCS 2130, pp 1199-1205: Springer-Verlag.

7. V.J. Hodge & J. Austin,
   A Comparison of a Novel Spell Checker and Standard Spell Checking Algorithms.
   Accepted for publication, Pattern Recognition, Elsevier Science.

8. V.J. Hodge & J. Austin,
   An Evaluation of Standard Spell Checking Algorithms and a Binary Neural Approach.,
   Accepted for publication, IEEE Transactions on Knowledge and Data Engineering.

9. V.J. Hodge & J. Austin,
   Hierarchical Word Clustering - automatic thesaurus generation.
   Accepted for publication, NeuroComputing, Elsevier Science.

Chapter 2 builds upon the TreeGCS algorithm developed in paper 2 and paper 1. Paper 1 is a concise version of paper 2. Chapter 2 is largely based on paper 9 with minor changes to the presentation. Chapter 5 is largely based on paper 4 with small amendments. Paper 3 is an abridged version of paper 4. Chapter 4 covers papers 6, 7 and 8. Paper 7 describes an earlier version of the spell checker while paper 6 is a concise version of paper 8.

Chapter 1

Introduction

1.1 Introduction

In this chapter, we initially state the contributions of the thesis. We then identify the challenges for Information Retrieval (IR) systems, we categorise and assess current IR approaches and outline and compare our IR system that we are developing (named MinerTaur) to the existing techniques. The name is derived from: "a **Miner** for **T**ext built using the **aur**a modular neural system".

1.2 Thesis Contributions

The main contribution of the thesis is the implementation and analysis of an integrated modular neural Information Retrieval architecture. We seamlessly integrate a binary matrix-based neural network architecture with a dynamic growing self-organising neural network architecture to implement an Information Retrieval system.

We implement and analyse a novel binary neural spell checker which validates the query words, suggests correct spellings and provides a user-driven stemming capability. We use a binary matrix architecture for the spell checker and integrate three different spelling approaches with a novel scoring scheme to seamlessly amalgamate the sets of matching words retrieved by each individual approach.

We employ a dynamic growing neural module to hierarchically cluster words according to their co-occurrence statistics in the text corpus. We develop a novel hierarchical clustering method 'TreeGCS' which extends an existing flat clustering approach by dynamically superimposing a hierarchy. We can then cluster word co-occurrence vectors from the text corpus to generate a synonym hierarchy which we use at various levels of abstraction to score word similarity in MinerTaur.

We implement and analyse a binary neural matrix which indexes the words and documents allowing the query words and their respective synonyms to be matched against the stored documents and a ranked list of matching documents retrieved.

Our overall system is a completely integrated, modular IR system. The organisation and technique developed is unique as far as we know with respect to the modules integrated and the model produced. The individual modules can be used stand-alone or may be plugged into other systems - all three modules are self-contained. The only processing necessary for the modules is to produce the context vectors for the hierarchical clustering and the binary vectors for the word-document matrix.

1.3 Information Retrieval

Over the years the amount of electronic information stored in databases and on the Internet has expanded rapidly to the current state where vast repositories of information on almost every conceivable subject are available, overwhelming users and producing *Information Overload*. Information Retrieval from these repositories is based on the pinpointing of words in the text using word-based queries generated by the users. It is impossible for humans to index and search such a proliferation of information by hand so automated indexing and searching techniques are required. It is certainly impractical and probably impossible to index the repositories dynamically while the user query is serviced due to the immense volume of data so the index must be pre-compiled and stored in an appropriate data structure.

Much of the text in the repositories is unstructured beyond the grammatical level (sentences and paragraphs) and provides little assistance for indexing other than word and document associations. Producing a suitable representation for the index is difficult due to the colossal number of associations between terms and documents that must be stored. The data structure needs to have both a compact, memory efficient representation and permit rapid retrieval. The index must have broad coverage (*indexing coverage*), recognising most of the words and concepts stored in the documents yet also permit various levels of indexing (*term specificity*) from general high-level concepts to specific words and allow the words and concepts to be linked to documents and thus located by the user. A simple flat index with no abstraction would be insufficient for the enormous repositories as searching a flat index would not be feasible and cross-referencing between indexed terms impossible for the user. The information in the repositories is often human generated and grammatical and spelling errors are common

so any indexing or searching technique must be able to overcome such errors. Document repositories are dynamic so any index must be incremental. If a new document is added or a document is altered, processing the entire repository to recompile the whole index is intractable.

Although humans have excellent information processing and error correcting capabilities, the sheer volume prevents a user being able to search the repositories and locate their required information even when hierarchical repository indexes are available. Sophisticated search tools are required to dovetail with the index data structure and filter the information reducing the search to a manageable size and facilitating the retrieval task. The search tools must be able to cope with the vast quantities of information, the vagaries and ambiguities of language, mistakes in the query and the possibility that the user may have only a vague description of the information they seek. Current search techniques can be slow due to the sheer volume of the search space although faster ones are being developed. Matching is often poor, as the quantity of retrievals does not necessarily indicate quality retrievals. Searches can return millions of potential matches to user queries and many matches retrieved are irrelevant. Many current search engines use simple Boolean keyword pattern matching that takes no account of spelling mistakes in the query terms, alternative spelling of the query terms (for example 'grey' and 'gray'), possible synonyms, alternative word meanings or the relevance of each query term. They simply return the document containing the greatest number of 'query words'.

There are three methods of retrieval tools that have been developed:

ad hoc retrieval uses a static corpus but allows dynamic queries,

filtering employs a dynamic corpus but uses a static query and the matching is Boolean where documents either match or they do not match,

routing may be viewed as filtering with ranked matching - all matching documents are ranked by a similarity metric according to their similarity to the query.

In this dissertation, we focus on techniques that automatically index the repositories and employ ad hoc retrieval - matching documents in a static corpus against dynamic user queries. There are three main approaches for ad hoc Information Retrieval: Boolean searches approach the text retrieval problem at the character string level, statistical methods approach it at the word unit level and knowledge based methods approach it at the word meaning level. The three categories are not strictly delineated. There is a degree of overlap between the categories. We have indicated where we feel systems fall into more than one category in the system descriptions in section 1.4.

1.3.1 Boolean

Boolean searches approach the text retrieval problem at the character-string level by treating query terms as symbols linked by logical functions. Boolean systems match the terms using string-matching algorithms and apply the appropriate logical conjunctives to the sets of documents retrieved from each string match. The documents may simply be stored in an indexed data structure (for example, a conventional hash table [44] or an inverted file list as in Glimpse [70]) and searched by Boolean keyword matching. However, Boolean searching cannot rank documents according to a relevance metric; the documents either match or they do not. We desire a ranked match list to score documents for the user. Boolean searching cannot account for word meaning or word variations; it takes no account of possible synonyms, alternative spellings, alternative forms such as singular and plural nouns or the polysemy of language as the most frequently used words in the English language have multiple meanings [104]. It matches purely on the words in the query. Boolean matching usually returns too many low quality matches. Simply retrieving a list of matching documents with no finer-grained differentiation quickly becomes unmanageable, as the number of matching documents for standard Boolean querying is often incomprehensibly large for the gigabytes or even terabytes of documents stored in many corpora.

1.3.2 Statistical

Statistical methods approach retrieval at the word-unit level, frequently linking words to documents in an appropriate data structure such as a matrix weighted according to a function of the frequency of occurrence of the word in the document (see for example [43] or [92]). This weighted approach suffers the difficulty of selecting the term weight parameters correctly. We demonstrate in our empirical evaluation in chapter 6 the criticality of the vector representation for the SMART [92] system. If the weights need to be extracted from a knowledge engineer then the process is too time-consuming and subjective. Alternative stochastic systems store n-gram to document associations [16], [17] and [28]. We described n-grams in chapter 4. A vector represents each document with each attribute denoting the presence or absence of a particular n-gram in the document.

All vector-based approaches return a document that most closely approximates the query given that both query and documents are vectors defined in the same vector space. The storage overhead can be large depending on the efficiency of the representation. Word-vector approaches frequently return very short documents that are the query plus a few words as the degree of vector similarity is much higher than for a longer document that will have many other words present. Word-based vector approaches cannot handle spelling errors, alternative spellings, synonyms, word stemming, (for example, 'engine', 'engines',

'engineer' etc.) or higher-level concepts. The n-gram approach allows spelling errors to be overcome, does not favour short documents but has a large storage overhead if all possible n-grams are stored, as many never occur in the English language. Many implementations eliminate any atypical n-grams from the storage and thus perform dimensionality reduction. However, storage is usually higher than a comparative word-document association system as there are generally more n-grams than words in a document. For all approaches, the matrix is a single level representation that does not account for any semantic relationships of the words; there is no synonymy.

Many neural network systems use the multi-layer perceptron network (see for example [25], [8] or [87], [88]). The network is divided into three layers with an input layer of nodes representing the words, an output layer of nodes representing the documents and a layer of hidden nodes between them with weighted links connecting the layers. The link weights represent the relevance of the words to the documents as measured by some relevance metric. The appropriate input nodes are activated during retrieval, the activation propagates through the network to the document nodes. This produces a network representation that is essentially similar to the word-document matrix described above. The approach suffers similar limitations to the word-document matrix.

The statistical vector space approach has been extended to a more knowledge-based approach by methods such as Latent Semantic Indexing (LSI) [30]. Through matrix decomposition, LSI aims to represent meta-level information rather than the purely text information of the standard vector approach and reveal the terms' higher-level semantic relationships. As the document representation is decomposed, low-level differences in the documents are factored out and LSI enables topics to be extracted from the data. The vector space may then be queried through topic rather than pure keyword matching. LSI increases recall and precision compared to conventional vector-based approaches but at a very high computational cost as noted by [59].

Another approach is a Bayesian Network of nodes, for example [15] representing words, concepts and documents linked by weighted arcs. The knowledge representation is not hierarchical, it is still essentially a flat representation - the only meta-level information is provided by concept nodes representing numbers, dates and company names. Bayesian networks cannot handle spelling errors or synonyms unless additional modules are added to the system. The approach also suffers from an exponential growth of nodes; as the number of documents grows a new node must be added for each new document and also any new concepts relevant to the new document. The number of document grows linearly with the size of the document repository and the number of concept nodes may

grow exponentially.

1.3.3 Knowledge-Based

Knowledge-based methods approach retrieval at the word meaning level, frequently integrating thesauri or knowledge hierarchies to permit more sophisticated and flexible matching strategies at variable levels of concept abstraction. The search space can be partitioned during retrieval and the focus of retrieval varied between very specific concepts to very general concepts. They elicit knowledge from source data rather than relying on knowledge extracted from human experts.

A knowledge-based approach that satisfies the criteria of knowledge abstraction, high flexibility and knowledge partitioning is an automatically generated hierarchical representation of the domain. The advantage of the hierarchy lies in its multi-level expressive power, its simplicity and flexibility and its ability to partition the index and searching, discrete regions of the hierarchy can be stored and searched separately. There are two alternative hierarchical topologies: a hierarchy of documents or a hierarchy of words.

The documents are clustered into a document hierarchy using a suitable unsupervised clustering algorithm see [45], [26], [108], [89], [21] or [71] Alternatively, the corpora may be classified using a suitable train and test hierarchical classification algorithm such as those in [52] or [116]. The words present in the documents generate word probability distributions for each document and these form the input space for the clustering algorithm. Clustering and classification induce a hierarchical abstraction model of the document collection allowing browsing of the repository by the user at their preferred level of abstraction from specific words to high-level concepts and the retrieval of similar documents when a document is supplied as a query to the system. The user's search can be progressively refined and focussed according to the hierarchal structure and the users can also browse the structure to identify individual or groups of documents on their required subject. It provides a unique insight into the topological structure of the document collection allowing high-level (topic) based relationships to be revealed as with LSI. However, it is difficult to get clean clusters as there is usually a great deal of overlap in the data. For a large document collection there will be so many document vectors in the repository's data space that clustering is intractable. A document hierarchy does not account for synonyms, misspellings (unless the entire corpus is spell checked), parts-of-speech, tenses and plurals in the document text. Each word has unique attribute in the document vector unless the vectors are pre-processed with singular valued decomposition [30] for example, to obtain a meta-level representation of the documents, to factor out low-level differences and to focus on higher-level

similarities. To permit word-based querying a separate data structure and retrieval algorithm is necessary to index the corpus and associate the documents with their constituent words.

The complexity of Information Retrieval resides in the complex semantic relationships rather than in the data itself. In a word hierarchy, the semantics of the words are realised by the cluster relationships. A word hierarchy reveals much about the information and topics covered by the collection from high-level concepts down to the fine-grained level of words and phrases that a document hierarchy cannot extract. The similarity of two concepts in the hierarchy is a function of the reciprocal (1/x) distance between them. Using a hierarchical representation, the focus of word retrieval can be enlarged or restricted by expanding or contracting the portion of the hierarchy examined. In the hierarchy, specific terms produce maximal precision but lower recall as similar concepts may be missed by the narrow focus of retrieval. Conversely, high-level concepts produce higher recall with lower precision as the documents retrieved are more diverse but are less tightly coupled. We need a hierarchical clustering method to automatically generate the hierarchy, as it would be impossible for a human to generate such a word hierarchy rapidly, domain-dependently and objectively. The word hierarchy may then be linked to a separate document data structure at the fine-grained level to store the word-document associations. Many systems use the pre-generated semantic hierarchy of WordNet. However, WordNet subdivides word senses, is subjective and requires supervised association of the corpus words to the relevant WordNet categories. The WEBSOM [48],[59] system generates a semantic category map using statistics extracted from the document corpus but the representation is flat. Our MinerTaur system generates a semantic category hierarchy automatically and from statistics extracted from the corpus requiring no human intervention.

1.4 Current IR Systems

The motivations of the systems detailed below encompass word sense disambiguation, synonym inferencing, query-document and document-document similarity estimation and both classification and clustering. Most of the methods described below are vector-based or knowledge-based with the exception of Chen & Honavar [19] and Glimpse [70] which are Boolean. The methods include contextual methods, n-gram methods, WordNet hierarchy methods, sentential structure methods, Bayesian network methods, document and term frequency, machine learning and neural network methods. The following list is not exhaustive but is intended to be broad and provide reasonable coverage and insight into the premises underlying each system. Van Rijsbergen [107] described the state-of-the-art IR systems at the time of writing (1979). The number of sys-

tems has increased exponentially since the book was written but the central tenets and concepts outlined in the book still hold. Chen [20] provides a concise description of more contemporary systems.

1.4.1 Boolean

Boolean IR systems suffer the disadvantages described above in section 1.3.1. Also, many users consider Boolean logical conjunctives as 'unfriendly', unintuitive and confusing and do not properly understand their implications to the matching process.

Glimpse [70] is not limited to solely Boolean querying. It allows full-text retrieval using Boolean queries, approximate query word matching and even regular expressions. It is essentially an approximate matching front-end preprocessor using Agrep (see [115], [114] and chapter 4) for approximate string matching of the query words linked to a two-level document index. The two-level index is extremely small, only 2-4% of the original text size. The index is a hybrid between a full text index and sequential searching through the data with no indexing. The index stores pointers to areas (blocks) where the data may be found. The system initially pinpoints the appropriate block to search and sequentially searches the indicated block to exactly locate the required information. This two-level approach keeps the index size to a practical minimum while limiting the search time to an acceptable length (the time for a block search). If a word is found more than once in a block only one association needs to be stored in the index as the sequential search will find all occurrences. The two-level index scheme is a trade-off: it needs to minimise storage by maintaining large block sizes but needs the block size to be sufficiently small to permit tractable sequential search. The system is essentially Boolean but the approximate word matching in Agrep allows best match querying; the system can retrieve the document with the minimum number of errors compared to the user query. The system is relatively slow for large databases and is limited to primitive querying. It is ideal for a personal file collection but probably too limited for a full IR system where a greater degree of query sophistication (such as synonym matching) is required to pinpoint the exact information and minimise false matches.

Chen & Honavar [19] use an associative memory based on a 2-layer perceptron with a layer of input neurons, a layer of hidden neurons and a layer of output neurons; and hence, two layers of connection weights. Bipolar input vectors representing the information are associated with binary output vectors in a supervised training process. When the input pattern is presented to the network during recall, one of the stored input patterns is returned as the binary output of the network. If multiple patterns match the input, then the output pattern

will be a superposition of the associated output patterns. With the addition of appropriate control circuitry the network may be modified to produce sequential recall of one or more stored, matching patterns. The network has a high fan-out for input neurons and high fan-in for output neurons, so for hardware realisations the network may be subdivided into modules linked by a shared input and output bus allowing simple expansion of the database. The approach is not robust. It is susceptible to spelling errors and cannot accommodate synonymy or paraphrasing. The approach is somewhat similar to our word-document matrix module but we incorporate further spelling and hierarchical synonymy modules into our MinerTaur system.

1.4.2 Statistical

The statistical techniques suffer the limitations outlined in section 1.3.2.

Vector-Based Approaches

The N most frequent words in a corpus form the n dimensions of a document vector producing a uniform representation for all documents. The attribute value is a function of the term frequency or a Boolean denoting the presence/absence of the word in the document for Boolean vectors.

SMART [92], [93], [94] was developed as one of the first IR systems. It uses a vector space model to represent documents. SMART performs automatic indexing by removing stop-words using a pre-determined list, stemming via suffix deletion, (optionally terms may be grouped using statistical word co-occurrence) and weighting. Various vector-weighting schemes have been utilised in SMART [93]. The most common variant is the *term frequency * inverse document frequency (tfidf)* weighting configuration. For tfidf, terms are assigned higher weights if they occur frequently in any particular document (term frequency), as they are considered important to that document but the weight is penalised if the term occurs in many documents (inverse document frequency). Terms that occur infrequently across a corpus are deemed relatively unimportant as they contribute little to document similarities. This contradicts Shannon's Theory that states that *the more infrequent a word the more information it conveys.* The vector weights are normalised to penalise long documents, as the weighting would unfairly favour them with their high term frequency.

$$weight\ of\ i^{th}\ term\ in\ document = \frac{term\ freq * inv\ doc\ freq}{Euclidean\ length\ of\ doc} \tag{1.1}$$

Given a new query, it converts it to a vector, and then uses a similarity measure such as cosine normalisation [93] to compare it to the documents in the vector space.

$$Sim(Q, D_i) = \sum_{Common\ terms\ t_j} q_j * d_{ij} \tag{1.2}$$

SMART ranks the documents, and returns the top ranked documents to the user. SMART can perform relevance feedback based on the user feedback of the quality of the retrieval. We identify in our later empirical evaluation that the SMART system is susceptible to the choice of vector configuration. SMART also implements global stemming that may introduce errors into the representation. We feel stemming should be supervised and specific to the query to prevent errors such as *'trainers' = sports' shoes* being stemmed to *'train' = mode of transport.*

Goldszmidt & Sahami's [43] approach uses document vector overlap to estimate document similarity. The authors remove the most infrequent words. Each document is then associated with an N-dimensional vector d_i. Each dimension corresponds to a distinct word in the union of all words in the corpus. The value of the jth component corresponds to the number of times the associated word appears in the document. The similarity of two documents is calculated as the degree of overlap of their vectors using normalised geometric mean and a scaling factor for the axes of the word space. This ensures that long documents will not be favoured by the word frequency attribute of the document vector over shorter and possibly more relevant texts. The documents vectors may also be clustered using hierarchical agglomerative clustering followed by iterative optimisation to generate a document hierarchy. From Zipf's Law [120], there are very few frequent terms and many infrequent terms in a corpus so the document vectors will comprise mainly 0's even after the most infrequent words have been eliminated. The few frequent terms occur in most documents so will have a vector attribute value > 0 for most document vectors. This produces a low similarity between document vectors and thus paraphrased documents are assigned low similarities. The method takes no account of semantic similarity and synonyms.

In **Koller & Sahami's** system [61], each document is represented by a Boolean vector where each vector element denotes the presence or absence of all words that appear in the corpus following the removal of infrequent words. Boolean vectors should have a lower storage overhead than the standard term weight vectors as a Boolean is cheaper to store than an integer or floating-point term weight. A hierarchy of classifiers is generated using feature selection to tailor the feature set of each classifier to its task. The approach is utilised in Sahami, Dumais, Heckerman & Horvitz [91] to filter junk e-mail from other desired e-mail. Here, additional features are assimilated into the vectors - words, phrases and domain specific features. Such a hierarchy of classifiers is susceptible to classification errors higher in the hierarchy. It is impossible to generate new classes for documents that do not fit any existing class. The methodology may also suffer the scalability problem. The approach is supervised; for the e-mail filter the

system must be trained with pre-classified exemplars to enable the classifiers to learn and classify. The method suffers the same problem as [43] with respect to Zipf's Law and the frequency of term occurrence. It takes no account of semantic similarity and paraphrased documents have low vector overlap.

Deerwester et al. [30] have developed **Latent Semantic Indexing (LSI)** that aims to determine structure in the pattern of word usage in documents. It uses statistical techniques to estimate the structure. The approach is more knowledge-based but uses global context statistics. Hence we have chosen to classify LSI as statistical although it may also be considered knowledge-based. LSI exploits dependencies between terms and explicitly utilises them in the representation and during retrieval. Terms that occur in similar contexts receive similar vector representations; context is document wide, not just local neighbourhoods as with WEBSOM [84] or MatchPlus [13]. LSI uses a rectangular matrix of terms and documents. The matrix is decomposed through singular-value decomposition (SVD - see [30]). The dimensionality is reduced (minimising storage) as many matrix components are small and may be ignored. Due to the dimensionality reduction, documents with differing term usage profiles may be mapped onto the same vector of factor values. A query is represented as the weighted sum of its component vectors and compared using the dot-product against all document vectors. The n documents with the highest cosines are returned as the best matches. LSI increases the recall and precision but at a high computational cost to the system. The SVD analysis is time consuming and computationally expensive [59] so updating the database of document representations is inefficient. LSI is best suited to static document collections. If the document collection is dynamic or even if there is only a slight change, the compressed representation has to be recomputed using SVD. The method handles synonymy but only partially accommodates polysemic word disambiguation. Polysemic words are represented by a weighted average of the different meanings. If none of the real meanings are similar to the average, the semantics are distorted. We noted a similar deficiency with our average context vector generation method in chapter 2 but posit possible improvements to surmount the problem in chapter 7

N-Gram Methods

N-grams are flexible, they require no linguistic pre-processing of the corpus and are language-independent. However, n-gram approaches have low recall and precision and return many false positives. We demonstrate in chapter 4 the low recall and precision of a conventional n-gram spelling technique.

Damashek [50], [28] initially pre-processes the text to transform all alphabetic characters to upper-case. Damashek employs 5-gram subsets. The text

is converted to 5 character sequences using a moving window that is advanced through the text one character at a time until all text has been processed. This stage produces a hash table indexed by the 5-grams and storing their respective frequencies in the text. When the entire corpus has been processed the frequency count stored in the hash table is normalised to replace the absolute count with a relative count. Normalisation ensures that different length but similar documents have similar hash tables. The difference between two documents is their scalar product divided by their lengths. The scalar product is the sum of the products of the common elements (5-grams). This is identical to the cosine of the document vectors used in SMART [93] described earlier. The main drawback of the approach is the need to limit the hash table size. There are 27*27*27*27*27=14,348,906 possible 5-grams in purely alphabetical text (26 letters plus [*space*]). To reduce the storage the hash table is limited to 262,144 positions. Many 5-grams map onto the same position and hashing collisions inevitably arise. Even though many 5-grams never occur in the language there are sufficient present to create collisions. This slows both insertion and retrieval and produces false positive matches due to the 5-gram hashing overlap.

Cavnar [16], [17] proposes an n-gram based document representation coupled with a vector processing hybrid approach. Each document is divided in to its constituent n-grams (4-grams) to produce a frequency histogram representing the document. All document vectors are normalised to unit length. A dictionary denoting the number of occurrences of each 4-gram in the entire corpus and how many documents contain the 4-gram is also generated. If the 37 characters [a-z] [0-9] [*space*] are represented, there are 37*37*37*37=1,874,161 possible 4-grams. To reduce the document vector dimensionality only 4-grams with an inverse document frequency of > 2.75 are stored in each document vector, compressing the storage. Cavnar notes that this essentially equates to stop-word removal and word stemming in a conventional vector-based approach. In a system query the 4-gram weight is the normalised frequency*inverse document frequency. The similarity of the document to a query is the cosine of the angle between the vectors - as the vectors are normalised this similarity measure reduces to computing the dot product. The fundamental drawbacks of Cavnar's method are the generation of false positive matches which necessitate a post-processing stage to eliminate the false positives. Cavnar's n-gram method is not positional [106] and matches the n-grams anywhere in the words regardless of their respective positions; one n-gram at the beginning of the query word will match the same n-gram even of it is situated at the end of a long lexicon word. N-grams introduce a huge storage overhead and require the storage of an association for every n-gram from every document. Even though Cavnar eliminates some n-grams by using the IDF factor the storage overhead is still prohibitive.

Bayesian Network

INQUERY [15], [14], [11] forms a network from the documents and the queries (see figure 1.1). The value of a document node (true or false) represents the

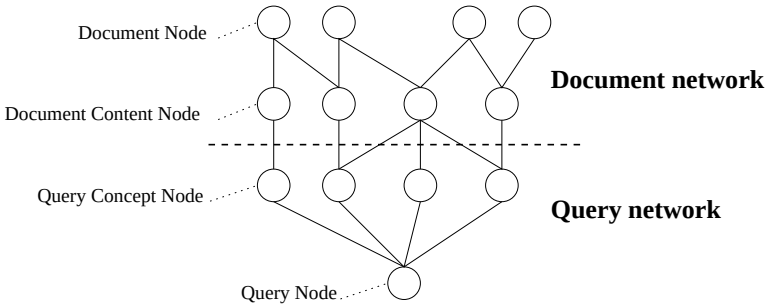

Figure 1.1: Figure illustrating the INQUERY Bayesian network

proposition that a document satisfies a query passed to the network during query processing. Content nodes represent meta-level information ranging from words to dates to company names. The content nodes are generated automatically by the system so there is no requirement for human intervention or knowledge engineering. Query concept nodes represent the concepts passed from the query. Query nodes are linked to their corresponding query concept nodes and are always true. All other arc weights are calculated from corpus statistics or estimated by the system. These weights need to be estimated accurately as it is critical to the retrieval process. The arcs linking document and content nodes are stored as an inverted index that is sorted to speed retrieval. We show in chapter 5 that Inverted File Lists are slower for retrieval than our binary matrix index structure. The arc weight must be estimated at query time and thus relies on the statistical model used to generate the estimate. During retrieval, the INQUERY system evaluates the query node and returns a belief list cataloguing the documents and their corresponding beliefs, i.e., the probabilities of the documents meeting the query. INQUERY cannot handle synonyms. However, a synonym module has been amalgamated to expand queries but it is not generated by the system and must to be produced in advance. There is no spell checking mechanism to handle typographical and grammatical spelling errors in the documents or queries although again such a module could be incorporated to spell check the query terms. INQUERY suffers the explosion of information as the number of document nodes grows linearly with the corpus size and the number of concept nodes could grow exponentially as new documents are added to the corpus. The representation is essentially flat, although meta-level information is extracted during the lexical and syntactic analyses, there is no knowledge abstraction to produce a hierarchical representation.

Statistical Neural Network Methods

Statistic-based neural network methods are exemplified by Multi-Layer Perceptron (MLP) networks. In addition to the limitations described in section 1.3.2, the statistic-based neural network methods all suffer from the small training problem, i.e., there is only a small amount of training data available and it is likely to represent only a small portion of the corpus - much generalisation is required to extrapolate the training data to encompass the entire corpus. Yang [117] observed inferior performance for MLPs compared to Support Vector Machines [52] or k-Nearest Neighbour, where the performance is calculated as a function of the training set category frequency, system robustness and the ability to handle skewed category distributions. Node-based neural networks also suffer from the computational complexity explosion due to the large number of documents in text corpora. One node in the network represents one document in the corpus and one node represents every unique word in the corpus. For a large text collection there will be millions of document containing millions of unique words so the network size may be intractable.

Belew's AIR [8] employs a weighted graph as the fundamental representation. The representation is modified via user feedback with users recommending their likes and dislikes and suggesting where to expand/ prune the search. The system comprises three layers with nodes in the three layers representing documents, keywords and authors respectively. All links are symmetrical and weighted. They are formed between documents and their associated keywords and documents and their authors. The weights of the document-keyword links are set according to the inverse frequency equation. The system relies on sub-symbolic, weak information to ensure stability and robustness against input errors, noise and system failure. The system is totally supervised and requires teaching at all stages.

Rose [87], [88] employs Belew's [8] AIR system as a foundation for the **SCALIR** legal retrieval system. SCALIR incorporates spreading activation into interleaved connectionist and semantic networks upholding the belief that activity is proportional to relevance. System nodes represent terms and documents (both legal cases and legal statutes). They may be hierarchically structured representing meta-terms (concepts) and meta-statute sections (chapters). They form tree structures and the structural information has to be manually entered although the actual nodes representing the hierarchy are created automatically by the system. All system links are directed, non-symmetrical and weighted. The connectionist links illustrate statistical regularities discovered by the system during indexing. The connectionist link weights are adjusted according to user feedback. The connectionist network represents micro-features (sub-symbolic information) and represents indexed links between words and documents (both legal

cases and statutes) that contain them using the *TFIDF* weighting scheme. The symbolic links represent knowledge generated by knowledge engineers - features and symbolic information. They represent case-to-case links. The methodology extracts a hierarchical knowledge representation using corpus statistics. The network construction is essentially supervised, a limiting factor of the underlying AIR [8] network.

Crestani [25] describes an experimental system using a 3-layer feed-forward model. The three layers correspond to the query descriptor, concepts and documents. The structure of the query layer is derived directly from queries and there is an indirect connection (via the concept layer) between the query descriptor and any relevant document in the final layer - the document layer. The connection is weighted according to the similarity between the query descriptor and the document. Documents have weighted links to other documents if they are cited by that document. The concept layer forms an intermediate layer representing meta-knowledge that is independent from the language and symbolism. Queries and documents are linked to concepts they describe. Concepts are inter-related by weighted connections. Learning is performed via the Back-Propagation rule. The network has very similar topology to the INQUERY Bayesian network [15] and suffers similar limitations.

Gedeon [42] employs a neural approach to learn synonyms and related clusters of words defining similar concepts from a text corpus in the legal domain. The MLP network has an input and output node for each keyword selected from text corpus. A layer of hidden units in the neural network connects the input nodes to the output nodes. For each keyword in a document, the corresponding input node is activated and the output vector trained as the word frequency of all keywords in the document under consideration (i.e., each corresponding output node is activated with respect to the frequency of that keyword in the document). The Back Propagation algorithm produces generalisation of outputs for each input word. The usefulness of the system remains to be tested in practise to assess the value of the associations. The central tenet of the approach is the neural network's ability to generalise to encompass unseen concepts. The clusters are not synonyms but in fact semantic associations specific to the corpus. Common words are more often included in clusters than infrequent words and this trait needs further investigation.

1.4.3 Knowledge-Based

The principal advantages and disadvantages of the knowledge-based were described in section 1.3.3.

Contextual Methods

These approaches utilise the local neighbourhood of words to generate vectors to represent the word and its context. The systems use stochastic and neural network clustering to establish the similarity of words. They can thus impute document similarity from the similarity of their contained words. They are essentially statistical techniques but to some degree they are knowledge-based as many generate a thesaurus from the word contexts.

MatchPlus [13] uses local context (to ensure neighbouring words have more influence) rather than the document wide context of LSI. Stop-words are initially discarded from the document and the remaining words are stemmed. MatchPlus represents each stem by a context vector of about 300 numeric features. A bootstrapping step uses a neural network to generate the context vectors sensitive to similar word usage but requires several passes to discover such higher order relationships i.e., synonymy. Once a dictionary of context vectors is generated, document representations can be calculated from the weighted sum of the term context vectors. Context vectors are used for terms, documents, and queries, allowing comparisons between any pair using a neural network. Similarity is determined by the dot product of context vectors. MatchPlus intuitively captures higher order relationships and light paraphrasing through similarity. It is fully automated but the bootstrapping requires several passes over the entire corpus. Global stemming introduces errors, for example, '*glasses*' meaning '*spectacles*' is stemmed to '*glass*' and confused with '*a drinking vessel*' or '*a vitreous substance*'.

Cheng & Wilensky's IAGO! [22] is an Internet search engine and categorises documents by exploiting thesaurus categories. A lexical disambiguation algorithm establishes a moving window of 50 words around a target word and associates each thesaurus category to which the target word belongs with each word in the surrounding window. The algorithm produces a matrix of normalised word category associations. To assign a category to a document, the classifier generates a vector of all thesaurus categories and each word in the document adds its category association vector. The algorithm outputs the strongest component as the category of the document. The algorithm will only classify text into the categories given by a thesaurus that are sometimes questionable - they fragment single senses and omit other senses. It only assigns one category per document although this could easily be extended by, for example, assigning the N strongest components as the topic.

WordNet Hierarchy

These methods rely on the pre-compiled hierarchical categorisation of synonyms, hyponyms (IS-A) and metonyms (PART-OF) of WordNet to estimate word sim-

ilarity and the most appropriate word sense (WordNet lists all senses of words with the most frequently occurring listed first). These techniques use the semantic hierarchy of WordNet so may be categorised as knowledge-based although they do not actually derive the knowledge-based thesaurus themselves.

Scott & Matwin's [97] approach exploits WordNet categories to induce document-document similarity. The methodology comprises three phases: Assign part-of-speech tags to each word in the corpus and discard all words except nouns and verbs. Use WordNet to assemble a global list of synonym and hyponym sets. Calculate the density of each synonym set present in each document (number of occurrences / number of words in document) to produce a set of numerical features. This can be utilised to determine the document-document similarity - the authors do not specify a methodology for this step. The method described requires a pre-determined hierarchy to operate and is susceptible to any errors or inconsistencies in the hierarchy. The method is optimal for classes that are broadly defined and/or semantically related but less than optimal otherwise.

Sentential Structure

Sentences are parsed to generate parse trees that are used in isomorphic matching to estimate query-document similarity. This technique is knowledge based as similar words will have similar parse trees so the technique takes account of some semantics although no word sense disambiguation is performed.

Smeaton, Donnell & Kelledy [100] use a linguistic approach to IR. Each clause in a text corpus generates a list where each item is a verb, adverb, adjective, stop-word or TSA (tree structures derived from syntactic analysis and used to encode ambiguities). Each user query is scored against each document by matching at three levels: leaf-node level - identical or related syntactic function labels; structural level - scores for inexact matches caused by ambiguities and combining node scores in to an overall TSA score for clause matches; document level - addressing how clause TSA scores are combined to produce document scores. The empirical evaluation indicated that the method functioned poorly. The approach suffers the inherent problem of complexity. The language differences between queries and documents, i.e., interrogative versus descriptive language combine to inhibit the methodology.

Document Classification

The approach is supervised and needs a train and validate learning phase to generate the hierarchy. Such pre-classified corpora are limited to specific pre-classified collections such as the Reuters collection used for our evaluations [82]. Pre-classified documents may be inconsistent and subjective due to the inconsistency and subjectivity of the human's specifying the document categories. The

Reuters collection contains many such anomalies, acknowledged in the documentation accompanying the collection. Classification systems are totally reliant on the categories awarded.

Joachim's [52] aim is a robust document classifier that can handle non-separable problems. The system uses Support Vector Machines (SVMs) to determine hypotheses that guarantee the lowest true error. They find a set of coefficients that separates the training data and has the shortest weight vector. Only examples where the co-efficient is greater then zero contribute - *support vectors*. SVMs can additionally learn non-linear hypotheses by incorporating convolution functions, learning (polynomial classifiers, Radial Basis Function classifiers [9] or 2-layer sigmoid neural networks [7]). They can handle non-separable problems by discarding training examples that contribute to inseparability. SVMs eliminate the need for feature selection which is required in other classifiers for instance inductive learning systems such as Yang's system [116] described below. SVMs are robust and do not require parameter tuning. However, they are purely supervised - they require a fixed number of predetermined categories to classify. The methodology used by Joachims is slow due to the quadratic optimisation (hypothesis) problem. However, Dumais et al. [31] integrate a method that subdivides the large QP problem into smaller problems that may be solved analytically and produces a 30-fold speed-up.

Yang's system [116] uses a symbolic learning, decision tree approach adapted from the C4.5 algorithm. Inductive learning induces a decision tree using the attributes of pre-classified exemplars, systematically maximising the information gain on each iteration of the algorithm. A binary vector represents each document with attributes set to 1 denoting the presence of a specific word in the document and attributes set to 0 where the corresponding word is absent. The attribute with maximal information gain forms the decision tree root and the training exemplars are subdivided according to the attribute. Then for each branch, the algorithm recursively determines the attribute that maximises the information gain until the training corpus is hierarchically categorised. The decision tree produced correctly categorises all exemplars. Although Yang [116] postulates that the number of training examples required is small to ensure correct classification, the approach still entails a pre-classified corpus for training.

Document Clustering

Document clusters are intended for browsing, identifying concepts and domain knowledge in the document collection and searching the corpus rather than purely for matching user queries. The document cluster topology is not incremental and must be recompiled if the knowledge base changes, i.e. the system must re-cluster the entire corpus.

Hofmann's [45] system hierarchically clusters documents from statistical analyses of word distributions. The text corpus is pre-processed with a word stemmer and the most and least frequent words are eliminated. A cluster abstraction model, annealed EM (a statistical mixture model similar to deterministic annealing see [45] which prevents over-fitting, reduces sensitivity to local maxima and specifies the maximum number of terminal cluster nodes) organises groups of documents into a hierarchy. The probability document i belongs to cluster C is dependent on the cluster specific word probability distributions in conjunction with the words present in the document. The word probability of document i is modelled as a mixture of exemplars from different abstraction levels. Each document contains a mixture of hierarchical mappings ranging from general language to highly specific technical terminology. The hierarchy allows users to browse or query the document collection at the desired level of abstraction. The approach has only been tested on small to medium sized collections.

HyPursuit [108] is a hierarchical network search engine that clusters Web documents and identifies the structure of the domain to allow user browsing and searching at their required level of abstraction. The hierarchical clustering is based on the documents contents and also the link structure of the hypertext links that link the document to other documents on the Web. HyPursuit uses the complete-link dendrogram method [33] to cluster the documents. The similarity of documents is calculated using a function of both hyper-links and the document term vector (as used in SMART [93]) given in equation 1.3. The current system uses the max function in equation 1.3.

$$S_{ij} = f(S_{ij}^{terms}, S_{ij}^{links}) \tag{1.3}$$

The link similarity S_{links}^{ij} in equation 1.4 is a weighted function of the number of descendant documents that documents i and j refer to S_{ij}^{dsc}, the shortest path between the two documents S_{ij}^{spl} and the number of ancestor documents that refer to both documents S_{ij}^{anc}.

$$S_{ij}^{links} = W_d \cdot S_{ij}^{dsc} + W_a \cdot S_{ij}^{anc} + W_s \cdot S_{ij}^{spl} \tag{1.4}$$

The document similarity is calculated from the normalised dot product of the term vectors for each document and uses the SMART [93] term similarity function given in equation 1.5.

$$S_{ij}^{terms} = \sum_{k} w_{ik} \cdot w_{jk} \tag{1.5}$$

The system is scalable. The user is able to interact at their required level of specificity. However, only a limited evaluation has been performed on the system by the authors and there have been no comparisons against existing systems or methodologies.

The Scatter/Gather system [26] has two components: a hierarchical clustering of the document corpus to enable user browsing and a k-nearest neighbour word-based text search to query the corpus. A vector represents each document with each attribute representing the frequency of occurrence of a specific word in that document. The paper presents two possible hierarchical clustering methods which are used in Scatter/Gather to generate a hierarchical cluster topology representing the text corpus at various levels of abstraction. A more accurate but slower clustering algorithm, fractionation [26] followed by group average agglomerative clustering, groups the document corpus off-line. For interactive user clustering, the user browses the pre-compiled hierarchical document cluster, selects the clusters they wish to explore further and Scatter/Gather employs a cheaper less accurate clustering algorithm, Buckshot [26] followed by group average agglomerative clustering to iteratively re-cluster the selected documents to the user's desired level of specificity. The approach imputes a hierarchical topology to identify the structure of the corpus but is limited to pre-classified datasets.

Knowledge-Based Neural Network Methods

Knowledge-based neural network approaches are exemplified by systems based on self-organising neural networks. Word vectors or document vectors form the input vector space of the neural network to infer similarity and categorise words and documents. WEBSOM [48],[59] and our methodology (see section 6.2) aim to progress one step further towards a knowledge-based approach by generating a thesaurus representation from the word contexts to incorporate synonyms into the retrieval process although neither uses word sense disambiguation.

Wermter, Arevian & Panchev [109], [110] use a recurrent network with internal recurrent hysteresis connections augmented with a dynamic short-term memory that is trained using supervised learning of semantic category vectors. A document title is represented by a sequence of significance vectors concatenated together with the outputs set as the topic of the document in the corpus. The recurrent connections are multiply layered and encode contexts based on the frequency of the words in different semantic categories normalised against the frequency of occurrence of the target word in the corpus. The context near the output of the network is more general and less dynamic than the contexts near the input. These networks are used for the text routeing of news wire documents. The networks have a high recall and precision rate, above 92 percent. The internal representation may also be analysed. The neural network performs supervised classification as the document topics form the outputs with the significant words forming the inputs to the neural network. This relies on pre-classified data being available to train and test the system during the

learning phase. We desire an unsupervised, autonomous learning approach that does not require pre-classified data but learns purely from statistics gathered from unstructured text. However, the approach uses semantic categories and knowledge abstraction and we feel such higher-level knowledge is essential for contemporary systems. The system lacks a spell-checking module so spelling errors in the corpus or a query may detrimentally affect the retrieval rates.

Wiener et al. [111] present a system using non-linear neural networks to model higher-order relations between document words and simultaneously predict multiple document subjects from common hidden features. The data must be pre-processed to reduce the dimensionality to ensure tractability of the input vectors for the neural network. The neural network relates the input vectors (reduced document features) to a binary output representing the topic assignment. There is one network per category. The use of a non-linear neural network permits multiple subjects to be assigned. The paper [111] describes prototypes utilising both an MLP neural network with a hidden layer and an MLP network with no hidden layer. The system is limited to pre-categorised document collections to enable the network topology to be derived and requires a highly supervised approach. The dimensionality reduction phase can factor out fine-grained knowledge from the system which may be important in ranking retrieved documents.

Roussinov & Chen [89] and **Chen, Shuffels & Orwig** [21] employ a SOM-based approach. The implementation adapts SOMs to produce a hierarchical taxonomy of maps representing document clusters down to concept clusters. The aim is a hierarchical clustering of progressively finer-grained concepts. Each document is represented by an N-dimensional vector denoting whether the N most frequently occurring terms are present or absent. Documents belonging to the same category are recursively decomposed to produce finer-grained maps corresponding to deeper levels in the hierarchy. The same process clusters both documents and concepts thus eliminating any bias. The concepts represent a spectrum of semantically similar words with an associated label. The authors improve the efficiency of the traditional SOM by computing the Euclidean distance to all nodes and updating the weights proportional to the number of non-zero co-ordinates rather than the total number of co-ordinates. The methodology uses the N most frequent terms to describe documents but such an approach plays in to the hands of the vocabulary problem (paraphrased documents would be less likely to be judged similar as their most frequent terms would vary). SOMs map more than one input (document or concept) onto each node so this precludes unique identification of documents and concepts.

Merkl [71], [72] aims to identify similarities between documents and produce

a hierarchical representation and visualisation of the document collection. The representation gets progressively more specialised top-down. A representative number of terms (keywords) are identified for the document collection. Binary vectors are produced for each document where each attribute represents the presence or absence of the corresponding keyword in that particular document. The vectors are then trained into the highest layer SOM. For each output unit in the SOM, a map is added to the layer below and so on down through the hierarchy. Any vectors that map onto the unit in one layer form the input to the map on the layer below. Each map is trained with specialised data enabling a hierarchical document taxonomy to be inferred. The document similarities are mapped on to the 2-D maps and thus are easily visualised. The representation also reduces training time compared to the single-layered SOM. For each more specialised layer, the common attributes from the vectors for each input vector space may be removed. Only varying attributes need be trained. The dimensionality reduction reduces the training time as the similarities are calculated on a per dimension basis so any reduction in dimensionality simplifies the calculation. The training time is further reduced, as there is no neighbourhood interference. In a single-layered SOM each cluster interferes with the self-organisation of its topological neighbours, particularly boundary units. Due to the hierarchical topology, much of the overall structure is maintained by the architecture. However, the topology is very rigid. The author concedes that the hierarchy needs to be pre-determined. The map dimensions and hierarchical topology must be preset which requires knowledge of the data and its required structure and tends towards a supervised approach. Our TreeGCS clustering approach is both dynamic and hierarchical, with very few structure parameters, only the desired number of GCS cells needs to be specified.

WEBSOM [49], [64], [47], [46], [48], [59] [65] categorises over one million documents. The system discards punctuation, stop-words, numbers, the most common words and the most infrequent words. The system initially produces a semantic category map using 270-dimensional average word context vectors input to a SOM as described in chapter 2. Document vectors are generated from the word locations in the semantic category map. The vectors represent a histogram of word categories identified by SOM grid location. The word attributes in the document vector can be weighted by inverse document frequency. The word category histograms are then projected randomly to form 315-dimensional statistical document vectors that form the inputs to a document SOM. The SOM arranges the statistical document vectors in to the document map. The word category histograms can be computed much faster than the topic vectors of LSI [59]. The indexing data structure is incremental in WEBSOM. WEBSOM is entirely unsupervised and parallelisable enabling computational speedup. However, the topography is single-layered and the SOMs cannot form

discrete clusters; the cluster boundaries have to be determined by a posteriori inspection. Our TreeGCS clustering algorithm for producing synonym networks (thesauri) is implicitly hierarchical and forms discrete clusters with no recourse to a posteriori inspection.

1.5 Current IR System Analysis

We feel the systems described in the previous section all suffer from at least one deficiency. Many of the methodologies exposited above eliminate high frequency and low frequency words that have low discriminatory power according to Luhn and noted in [107]. For systems that represent the corpus as a matrix of document vectors with a vector attribute representing each unique vocabulary word it is imperative to reduce the vocabulary to be indexed to minimise the dimensionality of the document vectors and reduce the storage requirement of the matrix. However, our MinerTaur system uses binary matrices (see chapter 3) to store the word-document associations which do not store null attributes, if a specific word does not occur in a particular document the storage for the association is nil. If we store infrequent words we do not unduly increase the overall system storage as these words occur in only a few documents so only a few index locations need be stored. We in fact only eliminate a small set of stop-words (extremely common words) such as {'and' or 'the'} that have absolutely minimal discriminatory power. We feel that elimination of high or low frequency words may discard essential information. We note that although low frequency words have a low discriminatory power with respect to the entire corpus, they have a high discriminatory power for the documents that contain them. They often uniquely differentiate a particular document so we feel they are essential to the retrieval process. Shannon's theory concurs that singleton words are important for indexing. Some systems even implement global word stemming to reduce the vocabulary yet further and thus reduce the number of terms to be indexed further. Van Rijsbergen quotes Salton [107]

> "... on the average the simplest indexing procedures which identify a given document or query by a set of terms, weighted or unweighted, obtained from a document or query text are also the most effective."

For most systems, the index is the portal to the text corpus, so the index must be complete and comprehensive. The searching and matching algorithms exploit the index so any errors or omissions due to word elimination will reduce the effectiveness of the system to the user.

Few systems have a hierarchical knowledge representation or hierarchical thesaurus integrated into the system. The systems that incorporate a knowledge or word abstraction generally rely on a pre-compiled hierarchy such as WordNet which may be too general, including many word senses not relevant to the

corpus while omitting domain specific terminology. Many systems extract some meta-level information such as dates and company names in INQUERY or topic extraction in LSI but neither are hierarchical. We generate a word hierarchy solely from corpus word co-occurrence statistics in chapter 2.

Few systems spell check the input or accommodate spelling errors. The Glimpse system uses the Agrep approximate string-matching algorithm to identify the best matching words and can retrieve the best matching documents from approximate matching words. N-gram based systems can also perform approximate string matching to identify the best matching word and overcome spelling errors in the query or the corpus. INQUERY incorporates a spell checker as a plug-in module but it is not an integral part of the system. Our spell checker is implemented in the same architecture as the word-document matrix to allow a simple transition from spell checking to document retrieval. The binary vectors representing the matching set of words from the spell checker form the inputs to the word-document matrix with no translation necessary thus maximising retrieval speed for this stage of system retrieval.

Few systems allow local, user-driven stemming of query words. Many systems implement global word stemming. Gale, Church & Yarowsky [41] noted that

> "the use of three general purpose stemming algorithms did not result in improvements in retrieval performance in the three text collections examined, as measured by classical evaluation techniques."

and van Rijsbergen [107] observed that stemming introduced errors. We allow a query word stemming stage in our retrieval process using local stemming selected by the user. We detail our spell checker's ability to match word stems in chapter 4. The Glimpse system could exploit Agrep's word stemming capabilities but none of the other systems discussed incorporates such a user-driven facility.

Many systems employ inefficient data structures. Glimpse uses a two-level index that is only suitable for small system storage. Bayesian networks and node-based neural-networks suffer the explosion of nodes as the number of nodes increases exponentially with the number of documents stored unless the network is subdivided and stored and processed in separate modules. However, differentiated modules need to be re-integrated (the results need to be amalgamated) and this amalgamation process can slow retrieval. Other systems minimise storage by incorporating various data compression techniques. LSI reduces the size of the word-document matrix through singular valued decomposition and in the process extract meta-level information. However, low-level information may be factored out, simple word-document associations are lost. The process is also exceedingly time consuming and computationally expensive [59]. The WEBSOM

system compresses the dimensionality of the word histograms that represent each document to a suitable level for processing. Again much low-level information may be lost. We use an efficient and accurate binary matrix data structure described in chapter 3 for the spelling and indexing modules. Our approach has faster training, faster retrieval and equivalent storage requirements compared to other similar data structures used in IR systems, see chapter 5. Our approach retrieves all expected matches and does not retrieve any false matches unlike similar compressed representations described in chapter 5.

1.6 The MinerTaur Integrated, Modular IR System

We elected to store the data repository using a word hierarchy for synonym retrieval linked to a word-document matrix to index the repository with a front-end spell checker to validate the query terms. See figure 1.2 for an overview of the MinerTaur system architecture during training and figure 1.3 for an overview of the MinerTaur system architecture during query matching.

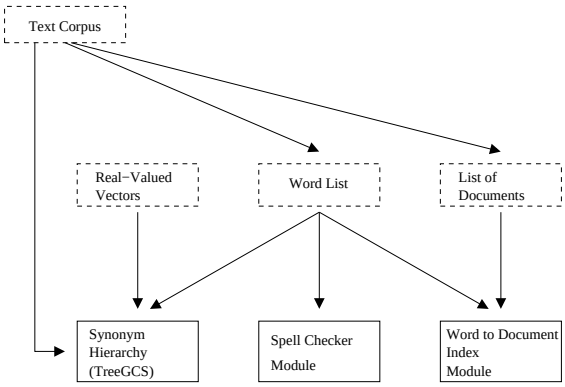

Figure 1.2: Figure illustrating our integrated modular MinerTaur system during training. The dashed boxes indicate artefacts and the solid boxes illustrate the three system modules.

1.7 Outline of Dissertation

We describe each of the modules that integrate to produce the overall MinerTaur system in separate chapters of the dissertation. Chapter 2 describes 'TreeGCS' and evaluates our technique for context vector generation and clustering against contemporary systems and techniques. TreeGCS processes the semantics of the words by exploiting the patterns present in text. These patterns produce statistical correlations in the context patterns of individual words. We represent the word context as real-valued vectors and thus infer the similarities of words

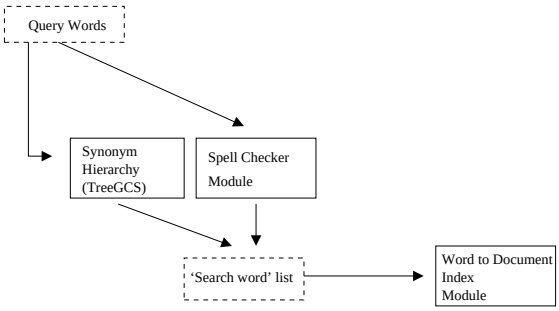

Figure 1.3: Figure illustrating our integrated modular MinerTaur system for querying. The dashed boxes indicate artefacts and the solid boxes illustrate the three system modules.

through vector similarity as similar words (synonyms) will have similar contexts and thus similar vectors. TreeGCS is an unsupervised, growing, self-organising hierarchy of nodes able to form discrete clusters. TreeGCS refines and extends the Growing Cell Structure algorithm of Fritzke [34], [36], [35], [37], [38]. We superimpose a hierarchy on top of the existing GCS neural network. The hierarchy forms as the GCS network grows and splits beneath. In TreeGCS the word context vectors are mapped onto a 2-D hierarchy reflecting the topological ordering of the input space. TreeGCS maps the data space in an evolutionary fashion, starting with a small number of cells and gradually generating new cells, validating the accuracy of the topological representation and removing superfluous cells while building a hierarchy on top of the evolving cell structure. The hierarchy branches and divides with the cell structure beneath. The word hierarchy acts as a pre-processing module for MinerTaur, see figure 1.4 for an overview. Proponents posit that humans intuitively cluster information such as words and concepts into a generalisation hierarchy [62] to allow understanding and both matching and retrieval of concepts. Manual creation of such a thesaurus would be formidable challenge; it is extremely time consuming and has massive complexity. The MinerTaur approach is entirely automated and uses only unstructured text corpora as data; no knowledge engineering is required and there is no requirement to pre-process the data into a structured format.

The hierarchy is searchable as soon as the context vectors have been generated and the TreeGCS hierarchical thesaurus neural network run for a small number of epochs to produce an initial hierarchy. TreeGCS may be interrupted at any time and the hierarchy exploited at that stage. Obviously the quality of mapping from the input vectors to the nodes in the hierarchy will improve as the number of epochs completed increases and the hierarchy settles into an accurate topological representation. Each node in the hierarchy represents a small group of synonyms at the lowest level and progressively larger groups of

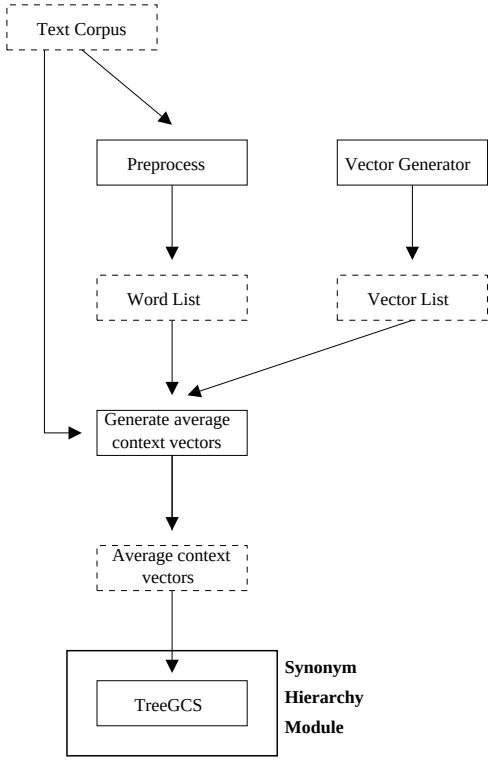

Figure 1.4: Figure illustrating our semantic hierarchy generation module. The dashed boxes indicate artefacts and the solid boxes illustrate the system modules and processes.

related words up through the hierarchy.

Our approach does not have to use its own word hierarchy. We could simply plug-in a pre-specified hierarchy. It is readily applicable to existing hierarchies as long as some scoring metric is available e.g. WordNet can produce scoring [97] so would be suitable. We demonstrate the necessity of using high-dimensional vectors to represent the individual words in the documents. High dimensional vectors ensure that the word vectors are approximately orthogonal and there are no implicit word dependencies or relationships in the vectors representing the individual words. Therefore all dependencies and relationships are imputed purely from the relationships between the document words. We demonstrate the superiority of a wider context window when generating the context vectors, illustrating the superior quality clusters and higher stability of the cluster topology produced. Finally we establish the higher quality of the clusters produced by TreeGCS compared to contemporary clustering approaches. The clusters produced from TreeGCS are similar to the clusters extracted from a benchmark human generated thesaurus.

Chapter 3 is a description of the AURA neural network architecture that forms the foundation for our MinerTaur system. AURA is a modular neural network allowing high flexibility of the system architecture produced. AURA provides rapid, single epoch training of associations and single step matching of inputs. AURA uses binary, unsupervised matrix memories. New associations are superimposed, i.e., new inputs are overlaid over existing trained associations so the network size is not altered allowing new associations to be added to the existing structure incrementally without any recompilation. The matrix may be very sparse or saturated as required by the application. AURA allows us to use multiple bits set in the bit vectors to increase storage capacity but in the system described in this dissertation we use orthogonal (single-bit set) vectors as the system memory usage is relatively low and there are no false positives when orthogonal bit vectors are used. AURA allows multiple input vectors to be overlaid, input to the system and matched simultaneously. We can even perform multiple, simultaneous partial matching.

A further pre-processing module is a spell-checking algorithm: to verify the query terms, identify spelling mistakes, to make recommendations for spelling corrections and to suggest alternative spellings. A spell checking pre-processor introduces fault tolerance as user errors can be corrected at source and robustness as the system will still function even with user errors. As any reader who has misspelt their query can attest, current search techniques are unforgiving of query spelling errors and do not attempt to spell check the individual query terms before commencing searching. The spell checker prevents user frustration as we are able to validate query terms before they are input to the system and recommend alternative spellings for the user, saving user time and processing time. Our innovative spell checker also allows word stemming to suggest all alternative stemming variants from a word stem supplied by user, for example if the user requests 'engine' the stemmer will suggest 'engines', 'engineer', 'engineering' etc. We implement such a spell checking methodology by developing an associative-memory based hybrid spelling technique in chapter 4. The approach uses the AURA associative memory used in the word-document association matrix so it integrates elegantly. We demonstrate the high recall of the methods integrated into the spell checker and also the high recall and precision achieved by our integrated, hybrid spell checker in comparison to a selection of benchmark algorithms. The words retrieved from the spell checker form part of the input to the word-document association matrix.

In chapter 4 we also empirically evaluate the affect of lexicon size on the retrieval efficiency of the spelling CMM to assess the scalability of our MinerTaur system. IR systems have to store vast repositories of textual information so a scalable system is imperative to accommodate huge vocabularies and lists of

word-document associations. We tabulate the memory usage statistics and the retrieval times for a range of vocabulary sizes. We identify a critical lexicon size where the time to retrieve the spelling candidate matches from a misspelt input exceeds our desired maximum of 2 seconds [74]. We postulate options for the storage of lexicons that exceed the maximum threshold and identify two preferred solutions depending on the constraints imposed. The choice is a trade-off between memory requirements against rapid retrieval with no false matches.

All IR systems need a method to index the data repository and to identify the word-document associations - a list of which words occur in each document to allow user queries to be serviced. This list of associations is extremely large in even a moderate IR system so the storage technique must be memory-efficient yet allow rapid and simple training and retrieval. LSI [30] decomposes the word-document matrix to decrease storage and in the process extracts the meta-level semantic relationships. However, the low-level word-document information is lost. In chapter 5, we develop an association matrix in the associative memory-based AURA neural network architecture to store the associations. By implementing our word-document matrix in AURA we can utilise AURA's single epoch training capabilities, superimposed coding, associative matching and AURA also enables partial matching in a single retrieval step. We provide a detailed comparison of our data structure against a series of benchmark data structures commonly used in IR systems. In our data structure, we maintain all necessary information unlike LSI yet we can also utilise the word hierarchy to introduce high-level categorisation (topic-based information) into MinerTaur.

In chapter 6, we describe our integrated, modular MinerTaur system. The words extracted from the word hierarchy are amalgamated with the words retrieved from the spell checker to form the input to the word-document association matrix. The outputs from the matrix are the best matching documents. We evaluate the memory usage statistics and the retrieval time for a range of queries applied to MinerTaur. We compare the training time of MinerTaur to the SMART benchmark system. We then compare the recall and precision figures of various configurations of MinerTaur against various configurations of the SMART benchmark system.

We begin our final chapter with the overall conclusions for our MinerTaur system and its empirical and comparative evaluations. We analyse MinerTaur against our criteria for an ideal system. Finally we posit recommendations for future improvements and expansions for our MinerTaur system.

Chapter 2

TreeGCS

In this chapter, we propose a hierarchical, word clustering neural network algorithm that we call TreeGCS. Our TreeGCS technique automatically generates a hierarchical thesaurus (synonym abstraction) using purely stochastic information derived from unstructured text corpora and requiring no prior classifications. The synonym hierarchy forms one of the query processing modules in our MinerTaur IR system (see figures 1.2 and 1.3). The synonym cluster abstraction overcomes the *Vocabulary Problem* as the words contained in any paraphrased documents fall into identical or similar synonym clusters where similarity is measured by distance in the synonym hierarchy. The synonym abstraction also overcomes *Information Overload* by focusing the search during query matching within cohesive clusters thus narrowing retrieval to very similar words. We describe existing word categorisation methods, identifying their respective strengths and weaknesses and evaluate our proposed hierarchical clustering approach against existing statistical and neural approaches using a human generated thesaurus for comparison. We demonstrate the effectiveness of our approach and its superiority to existing techniques.

2.1 Introduction

Word categorisation (encompassing both unsupervised clustering and supervised classification) enables the words to be associated or grouped according to their meaning to produce a thesaurus. In this chapter we focus solely on word clustering as this approach is unsupervised. Clustering does not require pre-generated human classifications to train the algorithm and is therefore less subjective and more automated as it learns from text corpus knowledge only. Word clustering can also overcome the *Vocabulary Problem* cited by Chen et al. [21]. They posit that through the diversity of expertise and background of authors and the polysemy of language, there are many ways to describe the

same concept; there are many synonyms. In fact, Stetina et al. [104] postulate that polysemous words occur most frequently in text corpora even though most words in a dictionary are monosemous. Humans are able to intuitively cluster documents from imputed similarity and are thus able to accommodate the differing vocabularies of authors and the inherent synonymy and polysemy of text. A computerised IR system must be able to match this ability. For computerised document similarity calculation, an underlying hierarchical synonym clustering is required to enable differing vocabularies to be accommodated. The distances in the hierarchy may be used for word similarity estimation and to score document similarity, thus allowing paraphrased documents to be awarded high similarity scores as their contained words fall into identical or neighbouring synonym clusters. The hierarchy allows us to focus query matching by limiting searching to cohesive clusters and therefore minimises the search space for each query.

An automated thesaurus approach is desirable as human generated thesauri are too general; they encompass all senses of words even though many are redundant for a particular domain. Human generated thesauri are expensive with respect to construction time particularly if a single human knowledge engineer generates the hierarchy. If multiple experts are consulted then it is very difficult to obtain a single unified hierarchy. Human thesauri omit domain specific terminology and also omit certain word senses while subdividing others where there is little distinction; they are rather subjective. Automatic methods can be trained generally or domain specifically as required. In this chapter we analyse current word categorisation approaches and describe and evaluate our method with respect to the current implementations.

Clustering may be defined as a process of partitioning a multi-dimensional input data space into groups of similar objects. Grouping similar data objects reduces the amount of data in the input space, it allows easier comprehension of the underlying structure and identifies similarities. A cardinal similarity measurement over the object attributes determines the similarity of the objects. A frequently used similarity measure is Euclidean distance. Hierarchical clustering builds a tree of nested clusters arranged according to the proximity of the clusters where proximity is determined by the similarity metric. Hierarchical clustering progressively merges the closest pair of clusters or splits large clusters into smaller sub-clusters to produce a tree structure. A hierarchical clustering allows the data to be organised and presented at varying levels of abstraction, identifying the underlying data structure at the requisite level of abstraction, categorising higher-level concepts and simplifying the input space. The structure may be traversed or queried at the desired level of data generality.

We use a self-organising hierarchical unsupervised neural network approach that progressively models the context data topology. Self-organising is systematic and objective compared to statistical and conventional neural network approaches. Traditional statistical clustering approaches require prior knowledge of the data model that may not be available or determinable. Statistical and neural approaches, for example k-means clustering [33] or multi-layer perceptron neural networks [7] require the number of clusters (k) or the neural network topology to be stipulated in advance. This would be impossible for an unknown distribution such as word context data. Our approach systematically evolves the cluster structure. Statistical and conventional neural approaches often cannot handle outliers (isolated data points separated from the other data points). Statistical and conventional neural approaches often do not scale well as the training time is exponential with respect to the number of input vectors. Some statistical approaches, for example ID3 [79] or C4.5 [80], are dependent on both the number of attributes and the number of values for each attribute. The self-organising neural network detailed here requires only minimal prior knowledge of the data distribution and only a minimal set of parameter settings is required for our approach. The results obtained from self-organising techniques are

> "on average as good as or better than results of other modelling techniques. Sometimes self-organising modelling is the only way to get results for a problem at all [66]."

Our approach is entirely automated and uses only unstructured text corpora as data. The motivation for our approach derives from the patterns present in text. These patterns produce statistical correlations in the context patterns of individual words. We can thus infer the similarities of words from their contexts, as similar words (synonyms) will have similar contexts due to their correlations. Through unsupervised text processing we represent semantic relationships by hierarchically categorising similar words according to their co-occurrence statistics. We automatically infer a domain-specific or generalised hierarchical thesaurus as required. We therefore surmount the *Vocabulary Problem* [21] by permitting synonym retrieval to match paraphrased documents. We can use the thesaurus in our IR system, MinerTaur, to award scores to synonyms using the intra-cluster distances and the inter-cluster distances in the hierarchy.

2.1.1 Current Methods

Current approaches for textual analysis are multifarious and diverse. The motivations encompass word sense disambiguation, synonym inferencing and both classification and clustering. They include (the following list is not exhaustive but is intended to be broad): contextual methods, WordNet hierarchy methods, clustering methods and SOM methods.

Contextual Methods

These approaches utilise the local neighbourhood of words in a document (the context) to establish lexical similarity and impute synonym groups or disambiguate polysemic words.

Yarowsky [118] employs two phases: an iterative bootstrapping procedure and an unsupervised categorisation phase. All instances of a polysemous word are identified in the text corpus. A number of representative samples are selected from each sense set and used to train a supervised classification algorithm. The remainder of the sense sets are trained into the supervised classifier. The classifier may additionally be augmented with one sense per discourse information, i.e., document topic. The classifier can then be used in an unsupervised mode to categorise new exemplars. Stetina et al [104] postulate that one sense per discourse holds for nouns but evidence is much weaker for verbs. The approach therefore is only suitable for nouns and requires an appraisal of the text corpus before processing commences to identify the nouns. The method is only partially unsupervised requiring a supervised initial training method; i.e. human intervention which can be time consuming. We use a totally unsupervised approach.

The motivation for Schütze & Pederson [96] is a lexical hierarchy exploiting contextual statistics and requiring no prior data knowledge. The algorithm collects a symmetric, term-by-term matrix recording the number of times that words i and j co-occur in a symmetric window centred about word i in the text corpus, where i and j are any random word indices from the list of all corpus words. Singular-valued decomposition (SVD) is used to reduce the dimensionality of the matrix to produce a dense vector for each item that characterises its co-occurrence neighbourhood. The dense co-occurrence vectors are clustered using an agglomerative clustering algorithm to generate a lexical hierarchy. The method groups words according to their similarity unsupervised rather than some pre-computed thesaurus categories. However, vector dimensionality reduction introduces computational complexity and may cause information loss as the vectors induced represent the meta-concepts and not individual words. Shannon's Theory states that *the more infrequent a word the more information it conveys.* These may well be discarded by SVD whereas out method produces average context vectors for all words or a subset of the words as required. SVD does not account for the proximity of the word co-occurrences (co-occurrence is considered from a purely binary perspective). There is no weighting of the co-occurrence according to the two terms' proximity in the context window.

WordNet Hierarchy

These methods utilise the human-generated hierarchical categorisation of synonyms, hyponyms (IS-A) and metonyms (PART-OF) of WordNet to estimate word similarity and the most appropriate word sense (WordNet lists all senses of words with the most frequently occurring listed first).

Li, Szapakowicz & Matwin's [67] method utilises the WordNet synonym, hyponym and metonym hierarchy to assign word similarity according to the distance in the hierarchy. Similarity is inversely proportional to distance. However, the distance of taxonomic links is variable, due to certain sub-taxonomies being much denser than others. Again the technique relies on an underlying predetermined word hierarchy and can only process words present in the hierarchy; it could not extrapolate similarities to new words. Human generated thesauri are subjective and rely on sense categorisation decisions made by the human constructor.

Clustering

An unsupervised clustering algorithm derives the word clusters and models of association directly from distributional data rather than pre-determined classes as in Yarowsky.

Pereira, Tishby & Lee [78] employ a divisive clustering algorithm for probability distributions to group words according to their participation in particular grammatical relations with other words. In the paper, nouns are classified according to their distribution as direct objects of verbs with cluster membership defined by $p(c|w)$ (the probability a word belongs to a cluster) for each word rather than hard Boolean classification. Deterministic annealing finds the sets of clusters by starting with a single holistic cluster and increasing the annealing parameter (see paper [78]). As the annealing parameter increases, the clusters split producing a hierarchical data clustering. The approach is limited to specific grammatical relations, requiring a pre-processor to parse the corpus and tag the part-of-speech. We desire an approach able to induce the hierarchy from unstructured text where no complicated syntactic pre-processing is necessary. At the time of writing, the authors felt their technique required further evaluation.

Self-Organising Map (SOM) Methods

Word vectors or document vectors form the input vector space of the SOM [58] to permit topological mapping, to infer similarity and categorise words and documents.

The aim of Lowe [68] is a topological mapping of contextual similarity exploiting

contextual information to derive semantic relationships. Each word in a 29-word vocabulary is associated with a 58-element co-occurrence vector. The value of the nth attribute in the co-occurrence vector reflects the number of times the nth word of the vocabulary has preceded and the $(n+29)$th attribute represents the number of times the nth word has succeeded the keyword where $1 \leq n \leq 29$. The 58 element vectors form the input vectors for a SOM network. The SOM is labelled by determining the best matching unit for each input vector. The word contexts (labels) are arranged topologically according to lexical and semantic similarity by the SOM. However, the method is inherently susceptible to the scalability problem; vector length grows linearly in relation to lexical size and thus the method is not feasible for a large vocabulary.

Ritter & Kohonen's [84] approach provides the motivation for our system. A topological map of semantic relationship among words is developed on a self-organising feature map. In the initial implementation, each word has a unique, seven-dimensional, unit length, real-valued vector assigned, for example "size"$\rightarrow \vec{x1}$ "three"$\rightarrow \vec{x2}$ "window"$\rightarrow \vec{x3}$. The input vector space is formed from the average context in which each word occurs in the text corpus. Semantic similarity is induced from context statistics, i.e., word neighbourhoods using a window of size three, one word either side of the target word, (only nouns verbs and adverbs are used in the method). For the three word window "size three window", each word has its 7-D vector attached "size$\rightarrow \vec{x1}$ three$\rightarrow \vec{x2}$ window$\rightarrow \vec{x3}$". The vectors are concatenated to form a 21-D context vector for the central word "three"$\rightarrow \vec{x1}\vec{x2}\vec{x3}$. All context vectors generated from the text corpus for "three" are summed and the summed vector is divided by the frequency of occurrence of "three" producing an average context vector for "three". If "three" occurred three times in the corpus then $\vec{Avg}_{three} = \frac{\vec{x1}\vec{x2}\vec{x3}+\vec{x8}\vec{x2}\vec{x4}+\vec{x5}\vec{x2}\vec{x3}}{3}$. The method has been extended to WEBSOM [49], [48], [59] that categorises over one million documents using a window size of three and 90-dimensional, unit length, real-valued word vectors. The approach is entirely unsupervised requiring no human intervention and parallelisable enabling computational speedup. However, SOMs cannot form discrete (disconnected) clusters thus inhibiting the data representation. Our TreeGCS approach forms separate clusters. The clusters have to be determined after the algorithm terminates by hand and this introduces the innate subjectivity of human judgements. Also, the word topography in WEBSOM is single-layered compared to our hierarchical clustering.

Therefore, SOMs have been extended to Hierarchical SOMs (HSOMs) [72], [71] which are multi-layered SOMs and thus allow a cluster hierarchy to be formed. Each neuron in the lattice on a meta-layer points to an entire SOM on the layer below. The sub-layer is a finer grained representation of the knowledge in the

higher layer. However, HSOMs generally require prior knowledge of the input distribution to predict a suitable topology. For example, in [73] the author describes the DISCERN system and states that one of the main liabilities of the DISCERN system is that *"much of the internal knowledge structures are fixed and specified in advance"*. Therefore, Song & Lee [101] have produced the structurally adaptive intelligent neural tree (SAINT) system see figure 2.1 that automatically determines a cluster hierarchy composed of SOM networks removing the need to pre-specify the structure. Where the mapping error from the input

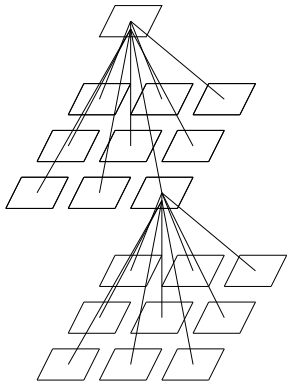

Figure 2.1: Figure showing SAINT's hierarchical architecture.

vector to the best matching node is high, new nodes are added. Any superfluous nodes are deleted and nodes that are very similar are merged. This dynamically creates a tree-structured SOM. The approach is similar to our TreeGCS method for generating cluster hierarchies. However, the lattice topology within the SOM sub-networks fixes the number of neighbours attached to each node and loses flexibility. Our TreeGCS method is more flexible as the number of neighbours attached to each node can vary between 2 (the minimum) and the maximum number of connections specified by the user and so is less limited by pre-specified topological constraints than a SOM lattice.

2.1.2 Summary of Current Methods

Many of the methods exposited use Zipf's Law[1] and stop-word elimination to reduce the vocabulary of the text corpus to be processed, some even implement word stemming. We feel that this may discard essential information. Although we do not generate context averages for frequent words, e.g., {the, and, but, etc.}, we include these words in the context averaging of the keywords. For this reason we use a context size of seven (three words either side of the target

[1] Zipf's Law [120] implies that a significant portion of the words in a corpus constitute the words that appear most infrequently whereas frequently occurring words comprise a relatively small portion of the corpus. Many approaches eliminate the most infrequent words to minimise vector dimensionality and computational requirements. The designers of these approaches deem that such words provide little discriminatory power for document similarity assessment.

word). We demonstrate in sections 2.3 and 2.4 the qualitative improvement of word clustering against a human-generated thesaurus and Euclidean distance-based vector approach of a size seven window compared to size three. Ritter & Kohonen [84] and their extrapolations [49], [48], [59], fix the context window at three and thus have to discard frequent terms, infrequent terms and punctuation etc. We feel these provide much information and are certainly employed by a human reader when parsing text. Dagan, Lee & Pereira [27] empirically demonstrated that singleton words (words occurring once) were important for parsing concurred by Shannon's theory. An infrequent word occurring only once in two documents may be the key to identifying those documents and should not be discarded from the indexing. The larger window allows us to maximise the lexical information used and minimise the amount of pre-processing required.

2.2 Our MinerTaur Method

We cluster words into a synonym hierarchy using the TreeGCS hierarchical clustering that we have developed and detail later. TreeGCS is an unsupervised growing, self-organising hierarchy of nodes able to form discrete clusters. Similar high-dimensional inputs are mapped onto a two-dimensional hierarchy reflecting the topological ordering of the input vector space. See figure 2.2 for an overview of our method. We assume a latent similarity in word co-occurrences and use the TreeGCS hierarchical neural clustering technique to estimate word similarity from contextual statistics without resorting to a human-generated thesaurus or WordNet. We categorise all keywords as discussed previously and perform no dimensionality reduction thus minimising information loss. The process is fully automated, requires no human intervention or data processing as the context vectors are generated automatically from unstructured text data and the clustering requires minimal a priori knowledge of the data distribution due to the self-organising, hierarchical neural network. Each node in the hierarchy represents a small group of synonyms at the lowest level and progressively larger groups of related words up through the tree. The distance between the nodes in the tree is directly proportional to the similarity of the word sets they represent.

2.2.1 Pre-processing

All upper-case letters are converted to lower-case to ensure matching. A list of all words and punctuation marks in the text corpus is generated and random, real-valued, unit-length m-dimensional vectors assigned to each word as in equation 2.1.

$$Word \rightarrow \vec{x} \in \Re^m \tag{2.1}$$

Stop-words are removed to create a second list of keywords. A moving window of size n is passed across the text corpus, one word at a time (see figure 2.3). Ritter & Kohonen use a context window of size three, we use size seven and illustrate

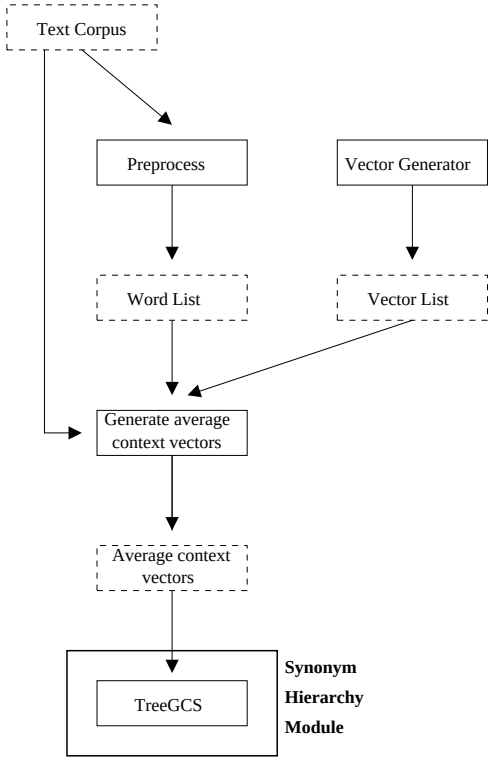

Figure 2.2: Figure illustrating the average context vector and synonym hierarchy production process in our hierarchy generation module. The dashed boxes indicate artefacts and the solid boxes illustrate modules and processes.

the qualitative improvement this generates in sections 2.3 and 2.4. If the word in the window centre is a keyword ($\vec{x}^{middle} \in \{keyword\}$)then the unique, random, real-valued, unit length m-dimensional vectors representing each word in the window of size n ($\{\vec{x}^1, \vec{x}^2, ..., \vec{x}^n\}$) are concatenated ($\{\vec{x}^1\vec{x}^2...\vec{x}^n\}$) and added to the $m * n$ dimensional context vector for the keyword ($\vec{y}^{keyword}$) see equation 2.2).

$$\vec{y}^{keyword} \in \Re^{m*n} = \vec{y}^{keyword} + \vec{x}^1\vec{x}^2...\vec{x}^n \text{ where } \vec{x}^{middle} \in \{keyword\} \quad (2.2)$$

When the entire corpus has been processed, all context vectors generated for each keyword are averaged (total for each dimension / frequency of keyword), see equation 2.3.

$$\vec{Avg}_{keyword} = \forall i \frac{\vec{y}_i^{keyword}}{frequency} \quad (2.3)$$

$$\vec{Avg}_{keyword} = symFact * \vec{y}_i^{keyword} \text{ for keyword attributes} \quad (2.4)$$

The keyword attributes in the average vector are finally multiplied by the symbol factor ($symFact$ in equation 2.4). The keyword is multiplied by a symbol factor of value 0.2 in Ritter & Kohonen's method for average context vector generation and also in the WEBSOM average context vector generation technique.

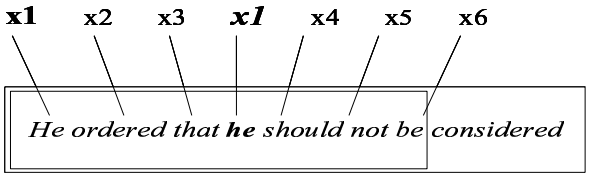

Figure 2.3: Figure illustrating the moving word window. The initial capital letter will be converted to lower case to ensure the 'he's match. Both instances of 'he' are represented by the same vector. The vectors associated with each word are concatenated to form the context vector for the target word 'he'.

The symbol factor diminishes the relative influence of the keyword (the central word in the context window) in relation to the surrounding words in the context window for the average context vectors. This prevents the actual keyword over-influencing the topological mapping formation and places the emphasis for topology and semantic similarity inferral on the context vector attributes, the surrounding words. We empirically determined the optimum factor value for our approach with a context window size of seven and found for seven-dimensional vectors that a symbol factor of 0.4 produced the optimal cluster quality (as judged by the authors). For 90-dimensional vectors the symbol factor has far less influence over the Euclidean distances between the context averages and thus the clusters generated, so we elected to use a symbol factor of 0.4, as this was more effective for the seven-dimensional vectors. This prevents the keyword over-influencing the context average but still provides sufficient influence for a context window of size seven.

The average context vectors form the input space of the GCS that underpins the hierarchy in our word categorisation technique. Due to the randomness and orthogonality of the word vectors, the averaged context vectors reflect purely symbolic associations and statistical relations and the individual word vectors are independent of any word ordering. We treat the text corpus as an ordered set of words and generate the synonym hierarchy from the intrinsic word statistics. We do not need to generate any linguistic structures, which are complex to produce and tend to be susceptible to linguistic ambiguities.

It is imperative that the vectors ascribed to the individual words in the text corpus imply no ordering of the words. Text processing is thus based purely on the processing of sequences of words. During trials with seven dimensional vectors, we observed that the vectors ascribed to the words do affect the clustering - the Euclidean distances are altered between the context averages. This is particularly germane for low frequency words where the context average is biased by the vectors assigned. Even for words occurring greater than 10 times, the vector

assignment influences the similarities. Kaski [54] showed that there is a direct correlation between vector dimensionality and orthogonality - the higher the dimensionality the greater the orthogonality. We empirically evaluated various dimensionalities for consistency with respect to cluster content when different vectors are ascribed to the words in the corpus. We used a centroid clustering dendrogram (see section 2.3) that clusters according to Euclidean distance to provide the most similar 25 words when the vectors were assigned to the words in two different orders. We found that the cluster sets were identical for 90-dimensional vectors over a set of experiments but varied for all dimensionalities tested below 90. The order of the cluster set varied slightly even for 90-dimensional but we concluded that this was sufficiently consistent and stable.

As with Deerwester [30], we handle synonymy but only partially accommodate polysemy. Polysemic words are again represented by a weighted average of their contexts but we only generate one context for polysemic words (the context is the mean context of all word senses biased by the frequency of occurrence of each sense). For example, *plant* may be *a living organism* or *heavy machinery*. Only one context average would be produced for *plant*.

2.2.2 GCS Algorithm

Our TreeGCS method is based on the Growing Cell Structure (GCS) method that is described next and is adapted from [35]. GCS networks form discrete clusters unlike SOMs where the SOM cells remain connected in a lattice structure. The dimensions of the SOM lattice have to be pre-specified (such as the 9x9 grid used in our evaluation later in this chapter). Contrastingly only the maximum number of cells needs to be pre-specified in GCS and the network grows dynamically by adding new cells and deleting superfluous cells until the maximum number of cells is reached. The number of neighbouring cells connected to a particular cell is not fixed in GCS unlike SOMs.

The initial topology of GCS is a 2-dimensional structure (triangle) of cells (neurons) linked by vertices. Each cell has a *neighbourhood* defined as those cells directly linked by a vertex to the cell. The input vector distribution is mapped onto the cell structure by mapping each input vector to the best matching cell. Each cell has a $contextWindow * wordVectorDimensionality$-dimensional vector attached denoting the cell's position in the input vector space; topologically close cells have similar attached vectors. On each iteration, the attached vectors are adapted towards the input vector. The adaptation strength is constant over time and only the *best matching unit (bmu)* and its direct topological neighbours are adapted unlike SOMs where the adaptation occurs in a progressively reducing radius of neurons around the *bmu*. Cells are inserted where the cell structure under-represents the input vector distribution and superfluous cells

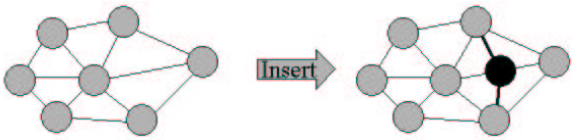

Figure 2.4: Figure illustrating cell insertion. A new cell and associated connections are inserted at each step.

that are furthest from their neighbours are deleted. Each cell has a '*winner counter*' variable denoting the number of times that cell has been the *bmu*. The winner counter of each cell is reduced by a predetermined factor on every iteration. The aim of the GCS method is to evenly distribute the winner counter values so that the probability of any cell being a *bmu* for a random input is equal, i.e., the cells accurately represent the input space.

The GCS learning algorithm is described below, the network is initialised in point 1 and points 2 to 7 represent *one iteration*. An *epoch* constitutes one iteration (points 2 to 7) for each input vector in the dataset, i.e. one pass through the entire dataset.

1. A random triangular structure of connected cells with attached vectors ($w_{c_i} \in \Re^n$) and E representing winner counter (the number of times the cell has been the winner) is initiated.

2. The next random input vector ξ is selected from the input vector density distribution. The input vector space is represented as real-valued vectors of identical length.

3. The best matching unit (bmu) is determined for ξ and the *bmu*'s winning counter is incremented.
   $bmu = ||\xi - w_c||_{min_{c \in network}}$ where $||\ ||$ = Euclidean distance
   $\Delta E_{bmu} = 1$

4. The best matching unit and its *neighbours* are adapted towards ξ by adaptation increments set by the user.
   $\Delta w_{bmu} = \epsilon_{bmu}(\xi - w_{bmu})$
   $\Delta w_n = \epsilon_i(\xi - w_n) \quad (\forall n \in neighbourhood)$

5. If the number of input signals exceeds a threshold set by the user a new cell (w_{new}) is inserted between the cell with the highest winning counter (w_{bmu}) and its farthest *neighbour* (w_f) - see figure 2.4,
   The weight of the new unit is set according to:
   $w_{new} = (w_{bmu} + w_f)/2$.
   Connections are inserted to maintain the triangular network configuration.
   The winner counter of all *neighbours* of w_{new} is redistributed to donate

Figure 2.5: Figure illustrating cell deletion. Cell A is deleted. Cells B and C are within the neighbourhood of A and would be left dangling by the removal of the five connections surrounding A so B and C are also deleted.

fractions of the neighbouring cells' winning counters to the new cell and spread the winning counter more evenly,
$\Delta E_n = -\frac{1}{|n|}E_n \quad (\forall n \in neighbourhood\ of\ w_{new})$.
The winner counter for the new cell is set to the total decremented from the winning counters of the neighbouring cells.
$E_{new} = \sum(\frac{1}{|n|}E_n \quad (\forall n \in neighbourhood\ of\ w_{new})$.

6. After a user-specified number of iterations, the cell with the greatest mean Euclidean distance between itself and its *neighbours* is deleted and any cells within the neighbourhood that would be left 'dangling' are also deleted (see figure 2.5). Any trailing edges are deleted to maintain the triangular configuration.
$Del = max_{c \in network}(\frac{\sum ||w_c - w_n||}{card(n)} \forall n \in neighbourhood)$

7. The winning counter variable of all cells is decreased by a user-specified factor to implement temporal decay.
$\Delta E_c = -\beta E_c \quad \forall c \in network$

The user-specified parameters are: the dimensionality of GCS which is fixed at 2 here, the maximum number of neighbour connections per cell, the maximum cells in the structure, ϵ_{bmu} the adaptation step for the winning cell, ϵ_i the adaptation step of the neighbourhood, β the temporal decay factor, the number of iterations for insertion and the number of iterations for deletion.

The algorithm iterates until a specified performance criterion is met, such as the network size. If the maximum number of epochs and the maximum number of cells are specified as the termination criteria then new cells are inserted until the maximum number of cells is reached. Once the maximum has been reached, adaptation continues each iteration and cells may be deleted. The cell deletion reduces the number of cells to below the maximum allowing one or more new cells to be inserted until the maximum number of cells is reached again. Deletion removes superfluous cells while creating space for new additions in under-represented regions of the cell structure so the input distribution mapping is improved while the maximum number of cells is maintained.

Fritzke has demonstrated superior performance for the GCS over SOMs [37]. Superiority with respect to:

- topology preservation, similar input vectors are mapped onto identical or closely neighbouring neurons ensuring robustness against distortions; this ensures that semantically similar words (words represented by similar average context vectors) will be mapped to identical or closely neighbouring GCS cells and thus identical or closely neighbouring synonym clusters when we superimpose our cluster hierarchy on top of the GCS network.
- neighbouring cells have similar attached vectors, ensuring robustness. If the dimensionality of the input vectors is greater than the network dimensionality then the mapping usually preserves the local similarities among the input vectors better in GCS compared to SOMs. This is particularly germane for the high-dimensional vectors produced by our average context vector generation technique which form the input vector space for the GCS clustering algorithm, if the corpus vocabulary is limited there may be more dimensions than vectors as with the evaluation in this chapter;

2.2.3 TreeGCS Algorithm

The TreeGCS is superimposed onto the standard GCS algorithm exposited above. A tree root node points to the initial cell structure and incorporates a list of all cells from the GCS. As the GCS splits or clusters are deleted, the tree divides and removes leaf nodes to parsimoniously summarise the disjoint network beneath and the GCS cell lists are updated with each leaf node holding a list of all GCS cells in its associated cluster. Only leaf nodes maintain a cluster list. A parent's cluster list is implicitly a union of the children's cluster lists and is not stored for efficiency - minimising memory usage. No constraints are imposed on the tree hence it is dynamic and requires no prior data knowledge - the tree progressively adapts to the underlying cell structure. The hierarchy generation is run once after each GCS epoch. The running time per hierarchy generation iteration is $O(cells)$ as we essentially breadth-first search through the entire cell structure.

A conceptual hierarchy of word synonym clusters is generated. The distance in the hierarchy between two concepts is inversely proportional to the similarity. Concepts are progressively more general and the cluster sets become larger towards the root of the hierarchy.

The underlying GCS's algorithm is susceptible to the ordering of the input vector space, if we alter the order of the input vectors in the dataset, a different cluster topology is generated for each unique input vector order. Thus, in TreeGCS we only commence cell deletion once 90 % of the total cells required

in the cell structure have been added. This delayed deletion prevents premature cluster committal and ensures the GCS network has evolved sufficiently before cluster splitting commences. In addition, we also iterate between different orders of the input vector space to ameliorate the order susceptibility (the x dimensional vectors that represent the context averages are rearranged to generate different data orders). Iterating between different orders cancels out the variance in the hierarchical structures generated by the different orders, vastly improving the algorithm's qualitative performance. The algorithm for the tree superimposition is detailed below in pseudocode.

For each epoch,

Execute the GCS epoch, forming an unconnected graph representing the disjoint clusters.

Breadth first search from the final winning cell for the epoch to determine which cells are present in the cluster.

While some cells remain unprocessed,

Breadth first search from the next unprocessed cell to determine which cells are present in the cluster.

If the number of clusters has increased from the previous epoch, **then** any tree nodes that point to multiple clusters are identified and child nodes are added for each new cluster formed (see figure 2.6). The cluster list of the parent is deleted and cluster lists are updated for the child nodes. If a cluster is formed from new cells (cells inserted

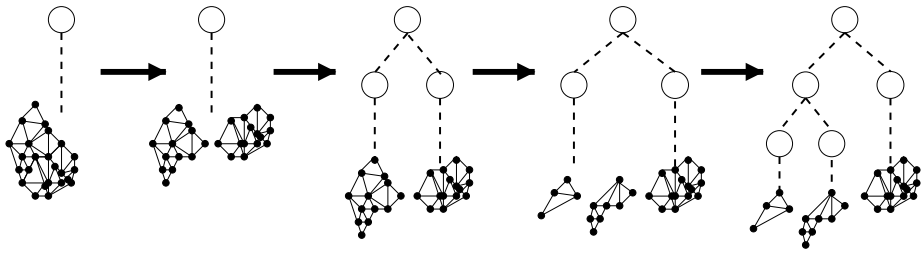

Figure 2.6: Figure illustrating cluster subdivision. One cluster splits to form two clusters and the hierarchy is adjusted. The leftmost cluster then splits again.

during the current epoch) then a new tree node is added as a child of the root and the new cluster cells added to the new node's list.

Elsif the number of clusters has decreased, a cluster has been deleted and the associated tree node is deleted. The tree is tidied to remove any redundancy (see figure 2.7).

For each unprocessed cluster, the tree node that points to that cluster is determined, the cluster list emptied and the new cells are added.

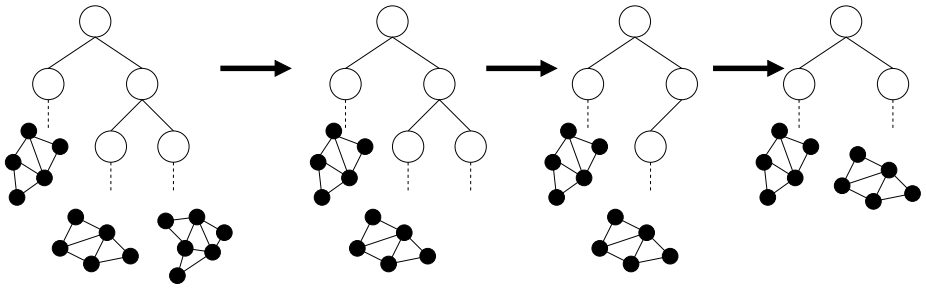

Figure 2.7: Figure illustrating cluster deletion. The rightmost cell cluster is deleted during an epoch (step 2) - this leaves a dangling pointer. The node with the dangling pointer is removed (step 3), leaving redundancy in the hierarchy. The redundancy is removed in the final step.

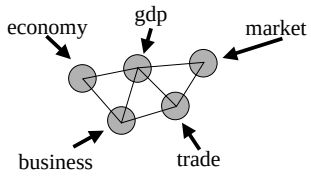

Figure 2.8: The cells in the GCS layer are labelled with the words they represent.

The GCS cells are labelled, see figure 2.8. Each input vector is input to the GCS and the cell identifier of the *bmu* is returned. The cell can then be labelled with the appropriate word. Words that occur in similar contexts map to topologically similar GCS cells thus reflecting syntactic and semantic similarity through purely stochastic background knowledge. This enables the induction of higher order relations: bus is similar to car; train is similar to bus allowing us to infer that train is similar to car. Tree nodes are merely pointers to GCS cells. All nodes except leaf nodes have only an identifier and pointers to their children. The leaf nodes have an identifier but also point to the GCS cell clusters and implicitly the GCS cell labels (they maintain a list of the identifiers of the GCS cells in their respective clusters). When the GCS *bmu* is identified, the associated tree node can also be identified and the tree can be traversed to find all word distances from the distances between the clusters (leaf nodes) in the tree.

The superimposed tree will be symmetrically equivalent; it is right/left independent. Trees that appear superficially dissimilar can in fact represent identical cluster topologies.

2.3 Evaluation

In section 2.3.1, we demonstrate the qualitative effectiveness of our average vector generation method against the two approaches described in sections 2.3.5 and 2.3.6. We demonstrate the qualitative effectiveness of our TreeGCS clustering algorithm against a SOM clustering in section 2.3.2.

Human clustering is innately subjective. In an experiment by Macskassy et al. [69], no two human subjects produced 'similar' clusters when clustering the information contained in a set of Web pages. This creates difficulties for cluster set evaluation and determining whether computerised methods are qualitatively effective. We clusters sets of average context vectors using TreeGCS and a SOM and compare the TreeGCS and SOM clusters against a human-generated thesaurus and a dendrogram clustering to ensure human clustering is being emulated while preserving the mathematical clustering of the dendrogram. Both TreeGCS and the SOM map inputs to nodes to approximate the input data whereas the dendrogram uses all input data points with no mapping. We can verify that the TreeGCS and SOM approximations are optimally preserving the Euclidean distances illustrated in the dendrogram and not distorting the input data space. Fritzke has previously demonstrated GCS's superior performance with respect to correctly classified test patterns over 6 common neural network approaches and the nearest neighbour statistical classifier for mapping the vowel recognition dataset [35].

In this chapter, we use a small dataset comprising 51 words to enable visualisation of the cluster structures and cluster contents and to permit a qualitative comparison of the cluster structures and cluster contents. A larger dataset would preclude visualisation of the cluster structures as they would be too complex to draw and a qualitative comparison of the cluster structures generated would thus be extremely difficult for a larger dataset. In chapter 6, we cluster 2,192 words using our context vector generation method coupled with our TreeGCS clustering technique. We list three clusters generated and also demonstrate an improvement in the query-to-document recall accuracy of our MinerTaur IR system when the cluster hierarchy generated from the 2,192 words using our word categorisation approach is included compared to the recall figure when the synonym module is not included.

2.3.1 Three Methods for Context Vector Generation

We use the Ritter & Kohonen method (see sections 2.1.1 and 2.3.5), WEBSOM method (see sections 2.1.1 and 2.3.6) and our MinerTaur method for average context vector generation (see sections 2.2 and 2.3.7) to produce three sets of context average vectors. We train each of the three average context vector

sets in turn into a standardised benchmark Euclidean distance-based clustering algorithm, the dendrogram, to derive three dendrogram clusterings, one for each context vector generation method. We can then compare each dendrogram cluster topology with a human generated word hierarchy (MS Bookshelf[2]). We identify the context vector generation method that generates a dendrogram with the most similar cluster topology and cluster contents to the human-generated Bookshelf clusters. The evaluation is thus based purely on the Euclidean distances of the average context vectors produce by each method.

2.3.2 TreeGCS versus SOM Clustering Comparison

We then train the three sets of average context vectors generated by the three methods into a SOM and TreeGCS for comparison of the accuracy of the two clustering algorithms. The clusters generated by the TreeGCS and SOM algorithms are compared with the synonym sets derived from Bookshelf. This provides a human-oriented comparison for TreeGCS versus SOM. We also compare TreeGCS versus SOM purely on vector distances, using the dendrogram cluster sets as benchmarks, to ensure that the mapping of input vectors to cells for both TreeGCS and SOMs are preserving the vector distances. We also note that this second comparison allows us to verify further the accuracy of our context vector generation method versus the other two approaches. We validate that the TreeGCS clusters more accurately emulate both the Bookshelf and dendrogram cluster sets when trained using the vectors from our method.

2.3.3 Text Corpus, Dendrogram and Thesaurus

The text corpus for the evaluation was taken from the economic data in the World Factbook [113] for each of the countries in Europe. This corpus is written in correct English, the vocabulary is reasonably small allowing a compact thesaurus to be generated with many words that have similar meanings allowing the cluster quality to be readily evaluated. For R & K and WEBSOM, we removed punctuation, numbers and common-words from the text corpus as described in [84]. The context vectors for all words were then generated as per equations 2.2, 2.3 and 2.4. For our method, only numbers were removed from the corpus. Context averages were calculated for keywords only (words not in the common word list) as per equations 2.2, 2.3 and 2.4. We cluster the context averages of the words that occur ten times or more in the text corpus for all approaches. This emulates R & K and WEBSOM that remove infrequent terms and maintains a consistency of words to be clustered for our approach to ensure a correlation and permit comparison.

[2]We were unable to use the WORDNET hierarchy as it does not contain all of the words from the text corpus. This precludes the use of synSet distances from the WORDNET hierarchy [67] (described previously) as an evaluation tool. Bookshelf allows us to generate clusters distances but no word similarity distances.

The dendrogram hierarchically illustrates similarities and is ideal for structure comparison. The dendrogram uses the centroid-clustering algorithm where the algorithm iteratively merges clusters. Initially there is one data point per cluster. Each cluster is represented by the average of all its data points, the mean vector; the inter-cluster distance is defined as the distance between pairs of mean vectors. The algorithm iteratively determines the smallest distance between any two clusters, (using the Euclidean distance metric) and the two clusters are merged producing a branch in the cluster tree. The merging is repeated until only one cluster is left. However, dendrograms have problems with identical similarities as only two clusters may be merged at each iteration, so if there are two pairs of clusters with equal distances, one pair has to be merged on one iteration and the other pair on the next iteration, the order being arbitrary. Dendrograms are not incremental, the structure has to be rebuilt if new data is assimilated and visualisation is difficult for a large dataset - there is one leaf node for each data point so it is very difficult to view for e.g., more than 500 data points. Therefore we feel a dendrogram would be unsuitable as the underlying mechanism for a lexical clustering method but is relevant for structure and cluster comparisons. Both TreeGCS and SOMs use Euclidean distance when mapping the inputs on to the output topology so we feel the dendrogram is consistent with these approaches. We take the *25* most similar words from the dendrogram and compare where these are mapped by the SOM and TreeGCS approaches (shown in bold in the evaluations). We highlight the most similar 25 of the 51 words to allow visualisation of whether the Euclidean distance-based similarity identified by the dendrogram is indeed being emulated by the SOM and TreeGCS mappings. We can also visualise where the remaining 26 words are mapped.

Also, we produced synonym sets from the MS Bookshelf thesaurus and compared the TreeGCS and SOM for the three methods of vector generation (see sections 2.3.5, 2.3.6 and 2.2) to these. We consulted Bookshelf for each of the terms that occur 10 times or more in the corpus. Where the definition of the term included one of the other terms we grouped these terms in to a synonym set. The synonym sets are arranged in similarity order, the closer together the more similar and the greater the distance the more dissimilar the words. The synonym sets from MS Bookshelf are:

1. {economy, system, market, budget, policies, program, government, account}
2. {investment, resources, welfare, privatization, reform}
3. {output, energy, exports, gdp, trade}
4. {industry, agriculture}

5. {growth, progress, inflation}
6. {debt, deficit},
7. {economic, financial, industrial, monetary}
8. {agricultural}
9. {currency}
10. {capita, percent, sector}
11. {substantial, large, highly}
12. {small}
13. {foreign, private, public}
14. {countries, republic, state}
15. {european, eu, europe, union, western}
16. {unemployment }
17. {years}

2.3.4 Settings

All settings are summarised in tables 2.1, 2.2 and 2.3. Table 2.1 compares the settings for the generation of the averaged context vectors from the word contexts in the text corpus for each of the three methods evaluated.

Method	Vector Dimensionality	Context Window	Symbol Factor
R+K	7	3	0.2
WEBSOM	90	3	0.2
MinerTaur	90	7	0.4

Table 2.1: Table comparing the settings for the context vector generation in each of the three methods evaluated.

Table 2.2 compares the settings for the SOM for each method of word context vector generation. We use the SOM-PAK [60] SOM implementation (as used in WEBSOM [48]). We use the parameter settings that produced the minimal quantisation error for a 9x9 map of rectangular topology, using the neighbourhood kernel 'bubble', (where the neighbourhood function refers to the set of array points around the node). The context vectors generated according to the WEBSOM method necessitated a different setting for α (the cell vector adaptation parameter) compared to the vectors generated by the other two methods to minimise the quantisation error of the topological mapping from the

Method	α for x epochs	Radius for x epochs	α for next y epochs	Radius for next y epochs
R+K	0.9 for 3000	10 for 3000	0.5 for 27000	3 for 27000
WEBSOM	0.75 for 3000	10 for 3000	0.5 for 27000	3 for 27000
MinerTaur	0.9 for 3000	10 for 3000	0.5 for 27000	3 for 27000

Table 2.2: Table comparing the parameter settings for the SOM algorithm to generate the map for each of the three context vector generation methods evaluated. α is the initial learning rate parameter which reduces to 0 during training and the *radius* is the neighbourhood of cells that are adapted in the SOM adaptation phase. The radius iteratively reduces to 0 during training.

input space to the 9x9 map.

Table 2.3 compares the settings for the TreeGCS for each method of word context vector generation. We set the parameters to produce the 'best' quality

Method	ϵ_{bmu}	ϵ_i	β	Cells	Max Conns	Insertion	Deletion	Epochs
R+K	0.1	0.01	0.001	81	25	10	810	30000
WEBSOM	0.02	0.002	0.0002	81	25	10	810	30000
MinerTaur	0.02	0.002	0.0002	81	25	10	810	30000

Table 2.3: Table comparing the parameter settings for the TreeGCS algorithm to generate the cluster hierarchy for each of the three context vector generation methods evaluated. N.B. Conns is an abbreviation for connections.

clusters as judged by the authors. The seven-dimensional vector evaluation required different parameters from the 90-dimensional trial.

We describe the three methods of average context vector generation and detail the results of the comparison between TreeGCS, SOM and dendrogram for each. Comparing all results to the MS Bookshelf clusters.

2.3.5 Ritter & Kohonen Method

We emulate the Ritter & Kohonen method as faithfully as possible. We remove common words, punctuation and numbers from the text corpus. We select the vectors from a distribution of random numbered, seven dimensional vectors. We use a context window of size three. We multiply the keyword vector by a symbol factor of 0.2. The following cluster topologies were generated from the text corpus using words that occurred ten times or more. We chose to only cluster word frequency 10 words to ensure the context vectors were truly averaged and not biased by limited exposure and also to eliminate infrequent terms as R & K.

- From the dendrogram clustering using the R & K average context vectors, the 25 most similar words are:
  {system union output industry substantial policies exports european privatization countries sector inflation percent economic foreign financial resources government growth large economy unemployment gdp years eu}

- For the TreeGCS hierarchy generated using the R & K average context vectors see figure 2.9. The words in bold are the 25 most similar words

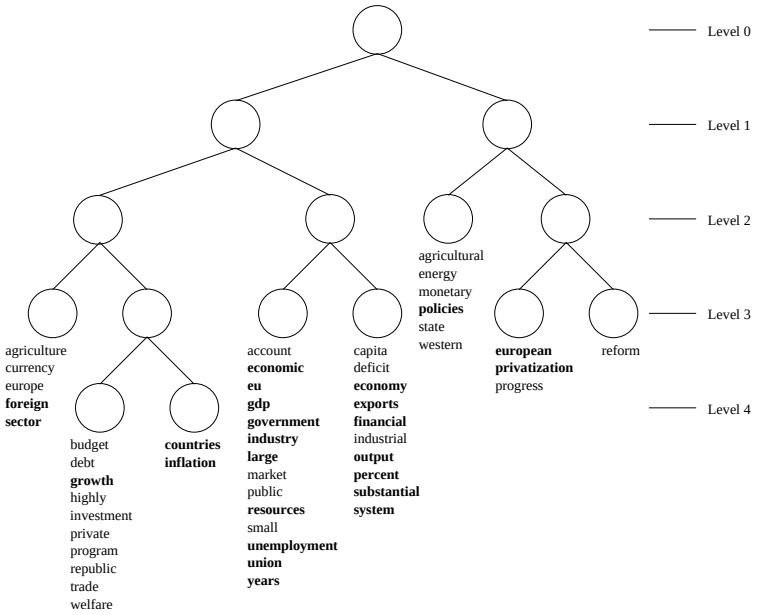

Figure 2.9: Figure depicting the TreeGCS cluster topology produced using the Ritter & Kohonen method of average context vector generation. The words in bold indicate the top 25 words selected by the dendrogram.

identified by the dendrogram generated using the R & K average context vectors.

- For the SOM cluster topology (see figure 2.10), again the 25 most similar words from the dendrogram are highlighted in bold.

2.3.6 WEBSOM

WEBSOM, the new development of the R & K approach, uses 90-dimensional real-valued random vectors for the words. Kaski [54] showed that the orthogonality is proportional to vector dimensionality and we have found that for seven-dimensional vectors, the actual vector assigned to each word in the corpus affects the context averages and thus the similarity and clustering produced. The seven-dimensional approach is also more susceptible to the symbol factor as the multiplier has more effect on the Euclidean distance than for 90-dimensional

monetary		program		**inflation** **output**		republic trade		
	reform		**economy growth system**		**percent** **substantial**		budget welfare	
		deficit				highly **sector**		agriculture currency market
	industrial		**eu** **member**		europe investment			
european privatization progress		**financial**		account		debt		
economic			**government** public		**exports industry union**			
		large **resources**		**gdp** **years**		agricultural **policies**		private
			foreign small		low **unemployment**		state	
capita						**countries** energy		western

Figure 2.10: Figure depicting the SOM mapping produced from the Ritter & Kohonen method for average context vector generation. The words in bold indicate the top 25 words selected by the dendrogram

where the effect is less. WEBSOM extends R & K and uses 90-dimensional word vectors, context window of size three and symbol factor 0.2

- From the dendrogram generated using the WEBSOM average context vectors, the 25 most similar words are:
  {countries european budget exports industry sector agricultural industrial large system trade output gdp financial eu economic economy government growth inflation percent privatization unemployment years foreign }
- For TreeGCS see figure 2.11 for the hierarchy generated using the WEBSOM average context vectors. The words in bold are the 25 most similar words identified by the dendrogram generated using the WEBSOM average context vectors.
- For the SOM, the topology is illustrated in (see figure 2.12), again the 25 most similar words from the dendrogram are highlighted in bold.

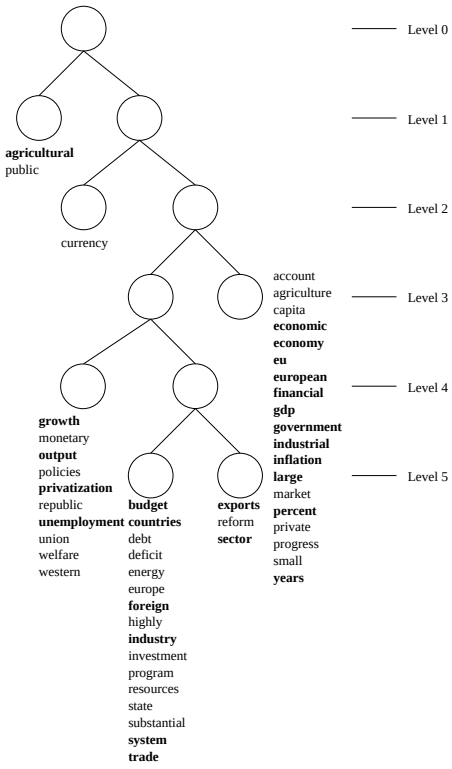

Figure 2.11: Figure showing the TreeGCS cluster generated from the WEBSOM method for average context vector generation. The words in bold are the top 25 words selected by the dendrogram.

2.3.7 Our MinerTaur method

Our method (see section 2.2) varies slightly from the previous two. We only remove numbers from the corpus, the previous two methods also remove common words and punctuation. We do not generate context averages for common words and punctuation but use them in the context window of other words, hence we have a larger context window of size seven. Our method uses 90-dimensional vectors and symbol factor 0.4. Again only words occurring ten or more times are shown in the clusters to ensure the contexts were averaged and not biased by infrequency due to the small size of our test corpus. Although normally we would include these words, we wanted a valid comparison to the previous methods.

- For the dendrogram generated using average context vectors produced by our MinerTaur method, the cluster of the 25 most similar terms is:
  { small reform percent exports gdp output market system agricultural budget industrial financial foreign large industry privatization inflation growth economic eu economy government trade unemployment energy }
- For TreeGCS see figure 2.13 for the hierarchy generated from average

capita		deficit				progress		agriculture market welfare
	eu						**industrial**	
currency debt investment **trade**		**exports** republic **sector**		**european** **large**		**growth** **percent** **privatization**		**agricultural** public
	economy		**economic**		**government**		**financial**	
foreign resources				**gdp** **output** **unemployment**		energy **industry**		program
	budget				**inflation**			
account union		**countries** europe		highly **system**		private small substantial		years
			state					
		policies		western		low		monetary reform

Figure 2.12: Figure showing the SOM mapping for the average context vectors generated using the WEBSOM method. The words in bold indicate the top 25 words selected by the dendrogram

context vectors produced by our method. The words in bold are the 25 most similar words identified by the dendrogram generated from the average context vectors produced by our method and listed above.

- For the SOM the cluster topology is shown in figure 2.14. The 25 most similar words from the dendrogram are shown in bold. We have also included the Sammon mapping (see [60]) for the SOM (See figure 2.15). The Sammon mapping maps the n-dimensional input vectors onto 2-dimensional points on a plane.

2.4 Analysis

2.4.1 Three Methods for Context Vector Generation

Figure 2.16 compares the cluster numbers for each of the top 25 words identified by the dendrogram generated from the average context vectors for each method. The graph shows the number of words in each of the 17 Bookshelf clusters for the

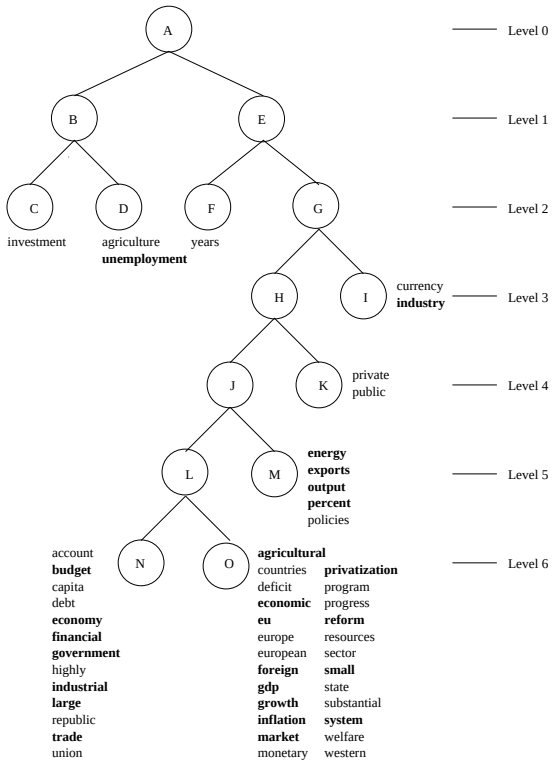

Figure 2.13: Figure illustrating the TreeGCS cluster generated from the average context vectors produced using our method. The words in bold are the top 25 words selected by the dendrogram.

top 25 words from each dendrogram. For our context vector method, 19 of the top 25 words are grouped in clusters 1 to 7 whereas the words are spread across the 17 clusters for the other two methods. Thus we feel that our vector generation method is producing vectors which cluster closer to a human generated clustering compared to the other methods which use a smaller context window. The clustering is obviously dependent on the corpus and the word co-occurrence statistics as these may distort word usage, for example, none of the methods has any words from cluster 6 {debt, deficit} in their respective top 25 dendrogram words. However, we feel a context window of size 7 with 90-dimensional word vectors achieves the closest arrangement to a human generated clustering.

With respect to the input data space, the 90-dimensional vectors are far less susceptible to parameter settings and vector assignments than the seven-dimensional vectors. The higher dimensionality vectors also increase word vector orthogonality; a prerequisite for the 'bag of words' average context vector generation approach. The word similarity is measured purely by semantic similarity of the word contexts and is not influenced by the vectors ascribed to represent the individual words. We feel similar methods, using self-organising maps or growing

debt deficit republic sector		public		**large** substantial		**eu**	european	
					currency			**government**
percent years		countries europe		resources **small**		market		**budget economy**
			growth		**energy**		**industrial**	
policies progress		**privatization reform**		**exports**		**gdp output**		**agricultural financial**
			inflation		**trade**		state	
industry program		**system**		agriculture **unemployment**		**economic** private		**foreign** low
					highly		welfare	
monetary		investment		account union		capita		western

Figure 2.14: Figure illustrating the SOM mapping of the average context vectors generated with our method. The words in bold indicate the top 25 words selected by the dendrogram.

cell structures, should use vectors of this dimensionality or greater to ensure orthogonality and maintain consistency and stability of the lexical clusters regardless of initial word-vector assignments.

2.4.2 TreeGCS versus SOM Clustering Comparison

For all three evaluations in sections 2.3.5, 2.3.6 and 2.3.7, the top 25 words from the dendrogram are spread across the SOM (as can be seen from the spread of bold text) but tend to be in closely related clusters in the TreeGCS hierarchy with just the odd exception (the bold text occurs in clusters that are near neighbours in the hierarchy). For example, for our method (see sections 2.2 and 2.3.7), the dendrogram words, shown in bold text, are predominantly in clusters M, N and O. If we examine the TreeGCS hierarchy (see figure 2.13), these clusters are very closely related. Only 'industry' and 'unemployment' are clustered elsewhere. With respect to Euclidean distance, the TreeGCS emulates the nearest neighbour approach of the dendrogram far better than the SOM. The Sammon mapping produced from the SOM using our method to derive the

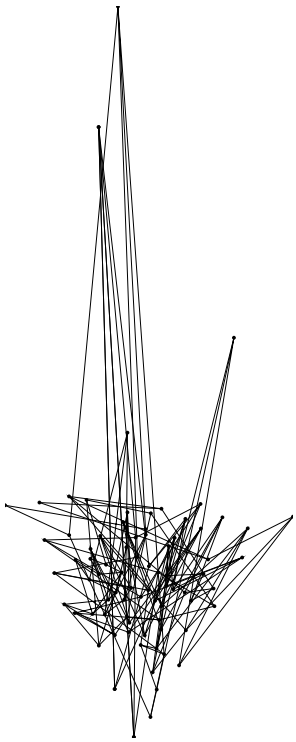

Figure 2.15: Figure illustrating the Sammon map generated for 90-dimensional vectors with context window = 7.

context vectors is extremely distorted (see figure 2.15). SOMs are criticised in the literature [9] for distorting high dimensional inputs when they map onto the 2-dimensional representation.

If we consider the TreeGCS and SOM generated from our context vector method (see sections 2.2 and 2.3.7), for TreeGCS, Bookshelf sets 1 to 7 are in M, N, O where N and O (see figure 2.13) are siblings and M is a sibling of their union. The remaining Bookshelf synonym sets are grouped very similarly to Bookshelf although they are within other sets, i.e., the groupings are consistent with Bookshelf but are subsets of larger sets. So we feel that the groupings are emulated. For the SOM (see figure 2.14), the words from the Bookshelf synonym sets are spread across the SOM. Therefore, we feel that our TreeGCS emulates the human generated clustering accurately and more faithfully than the SOM when both algorithms are trained with an identical input data space.

We noted that this second evaluation also allowed a further comparison of the vector generation methods as we can compare the quality of the TreeGCS clusters produced from each of the three input vector sets. With respect to the size of the context window, we feel that our size seven-context window produces superior quality TreeGCS clusters to WEBSOM's context window of size three.

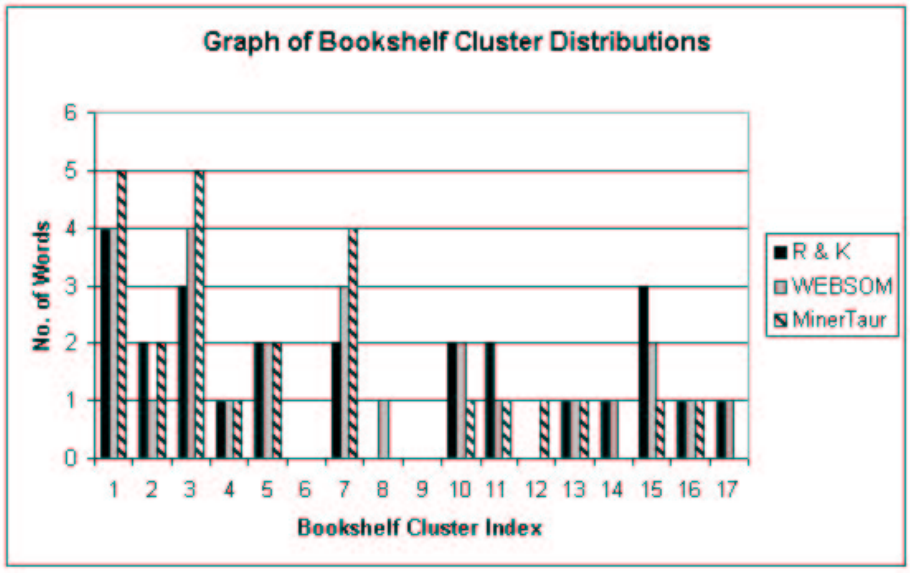

Figure 2.16: Graph of the distribution of words through the Bookshelf sets using the top 25 words identified by the dendrogram for each of the three average vector generation methods. For example, 5 of the top 25 words from our context vector dendrogram are found in Bookshelf cluster 1. The numbers on the x-axis are the cluster numbers from section 2.3.3.

The TreeGCS clusters produced from the average context vectors produced by our method emulate both the nearest neighbour (dendrogram) and human generated thesaurus more accurately than the TreeGCS cluster produced from the WEBSOM average context vectors. The vast majority of terms from the dendrogram and MS Bookshelf are in the three clusters (M, N and O) (see figure 2.13) for our vector generation method but are spread across four clusters with many of the other words also within these clusters for the WEBSOM method of vector generation (see figure 2.11). For the TreeGCS clusters, the input vector set produced from our method most accurately emulates both the dendrogram and MS Bookshelf cluster sets compared with the vector set generated from the WEBSOM method.

2.5 Conclusion and Future Work

We feel that our method, 90-dimensional vectors, symbol factor of 0.4, context window of seven is superior to the R & K and WEBSOM methods. Our method for context vector generation enables the dendrogram (Euclidean distance) to be more similar to the human generated thesaurus of Bookshelf than either the R & K or WEBSOM approaches. We also demonstrated that the TreeGCS algorithm emulates both Euclidean vector-distance based and human generated cluster sets more faithfully than the SOM algorithm. Therefore, we feel the

optimum approach for synonym clustering of the methods evaluated is to generate the average context vectors using our method and train these in to the TreeGCS cluster algorithm. TreeGCS not only emulates the nearest neighbour and human generated clusters more faithfully, it forms discrete clusters and dynamically forms a lexical hierarchy.

There are two main drawbacks to our current method. The first is the inability to disambiguate words. All senses of a polysemic word are averaged together during the context average formation, distorting the averaged context vectors produced. We intend to improve this by including part-of-speech tagging to differentiate identical words which represent different parts-of-speech, for example *spring: noun, a water source* and *spring: verb, to jump*. However, autonomously differentiating word senses is currently intractable and relies on a knowledge engineer to tag the senses. This is infeasible for a large IR corpus. We discuss possible extensions and improvements in chapter 7

The second main drawback lies in the underlying GCS algorithm and is a speed problem. The algorithm is dependent on the winner search - finding the best matching unit. This involves comparing the input vector to the vector attached to each cell, calculating the difference for each vector dimension. This must be repeated for each vector in the input vector space to complete each epoch. This search is therefore, (number of input vectors * vector dimensionality * number of cells) $\approx O(n^3)$ for each GCS epoch. For the small vocabulary evaluated in this chapter the speed problem was not apparent. However, for a large IR corpus, with an intrinsically extensive vocabulary the speed is prohibitively slow and the algorithm running time is infeasible. In chapter 7, we detail approaches we have tested and other possible amendments and extensions to the underlying GCS algorithm to speed the algorithm, reduce the running time and hence remove the bottleneck.

Chapter 3

AURA

3.1 Introduction to the AURA Neural Network

Neural networks map inputs onto outputs (the best matching item(s)) rather than performing a search of the entire input data space for the best match. In this dissertation we utilise the AURA neural network system. AURA is a collection of neural network-based components that may be implemented in a modular fashion. The building blocks of the AURA system are Correlation Matrix Memories (CMMs). CMMs are binary associative $m*n$ memory structures. The AURA CMMs use binary weights to produce an index data structure storing a linear mapping μ between a binary input vector of length m and a binary output vector of length n as in equation 3.1.

$$\mu : \{0,1\}^m \rightarrow \{0,1\}^n \tag{3.1}$$

AURA uses a supervised learning rule, similar to a hash function, to simply and rapidly map inputs to outputs. In figure 3.1, the input vector i addresses the m rows of the CMM and the output vector o addresses the n columns of the CMM. AURA implements Hebbian learning by reinforcing active connections during network training.

Unfortunately in conventional neural networks and the Hopfield binary network [7] training takes time. Conventional neural networks are also often unable to deal with missing inputs during training and they cannot match noisy or partial inputs during retrieval. Associative binary neural networks such as AURA do not suffer from the lengthy training problem. In AURA, training is a single epoch process with one training step for each input-output association trained preserving AURA's high speed. In contrast, our TreeGCS method detailed in chapter 2 requires approximately 10,000 epochs with 2,192 iterations per epoch to train 2,192 input vectors onto a network of 1,000 cells. Storage is efficient in associative binary networks as the size of the matrix can be determined when it is instantiated. Thus, new input patterns do not require additional memory

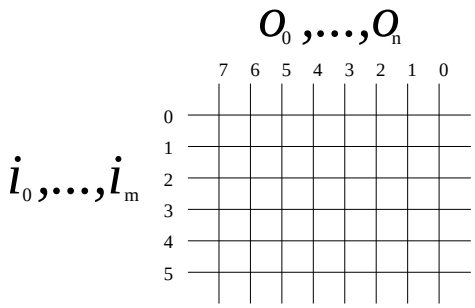

Figure 3.1: The input vector i addresses the rows of the CMM and the output vector o addresses the columns.

allocation as they are overlaid with existing trained patterns, [105]. An indexing data structure, such as the spelling-word index or word-document index described in this thesis, may be pre-structured for more rapid searching such as tree-structured, but this process is memory and processor demanding and time consuming. AURA exploits the simple linear mapping of binary vectors to produce rapid training and retrieval.

The AURA architecture contrasts with the TreeGCS neural approach described in chapter 2. AURA may be considered a long-term storage structure where static associations are stored to be retrieved at any time in the future as required. TreeGCS is more comparable to a learning memory where knowledge is gradually learned and a network structure grows and evolves to represent the data to be stored. AURA is aimed at storage efficiency, rapid single-step storage and rapid single-step retrieval but conversely, TreeGCS requires much longer to train due to its evolutionary nature and must be given time to evolve satisfactorily to represent the knowledge. TreeGCS stores and matches at a symbolic level whereas AURA stores and matches at the individual binary-bit level. TreeGCS maps inputs onto nodes so multiple inputs map to an individual TreeGCS node but AURA uses rows and columns in a matrix structure where each location in the matrix is indexed by one row and one column. TreeGCS is suited to the word hierarchy evolution whereas AURA is appropriate for the associative retrieval required by the spell checker and word-document index.

AURA extends previous work on binary associative matrices by Amari [2], Palm [77] and Willshaw [112]. AURA has augmented these antecedent approaches by adding a partial match capability where only a subset of the active inputs are matched and can determine the best matching output for a partially matched input. For example if three inputs are set then AURA can retrieve any output that matches two or less inputs. This capability is described in more detail in section 3.1.4. AURA has also added a multiple input and multiple output capability where more than one input vector can be gathered into a single cumulative

input vector and matched. Multiple output vectors may also be gathered into a single, cumulative output vector and retrieved. This process is described in section 3.1.4.

For our MinerTaur system, we develop CMM modules for a hybrid dual-CMM spell checker detailed in chapter 4 and also to produce an indexing data structure to store word to document associations described in chapter 5. The partial matching of multiple inputs where multiple best matching outputs are retrieved forms the foundation for our spell checking modules and word-document index. In chapter 5, we evaluate a CMM word-document indexing structure against a standard inverted file index, a standard hashing algorithm and a compressed hashing storage technique. We evaluate all four data structures storing 1.25 million identical word-document associations for training speed, memory usage, serial recall speed and partial match recall speed. We note that the CMM has vastly superior training speed and partial match recall speed. The CMM has a comparative memory usage but is slightly slower for serial recall.

3.1.1 Input and Output Vectors

In the AURA modular neural network architecture the tokens (spellings, spelling codes, word and documents) are initially translated to binary patterns as the AURA modules require binary data inputs. The binary pattern for each token has a preset constant number of bits set to 1 in a fixed-length binary array. The binary vectors may have a single bit set (orthogonal) vectors or multiple bits set.

For the hybrid spell checker described in chapter 4, the input spellings are character based and the binary vectors are divided into a series of concatenated 'chunks' with each chunk representing a character from the input word. The output vectors represent the corresponding words and have only a single-bit set to uniquely identify each word from the lexicon of all words. Each word vector is an m-dimensional binary vector where m equals the number of words in the list. Where the value of an attribute is not known the binary pattern is set to all 1's indicating that all attributes are a valid match as in the UNIX '?' any single character matching facility.

For the word-document association matrix in chapter 5, the words form the inputs and the documents the outputs of the neural network. The words are represented by the identical vector that represented them in the spell checker to allow the vector to uniquely identify the word and so that the word vector output of the spell checker may form the input of the word-to-document matrix. The documents are represented by an orthogonal vector to uniquely identify them with n-dimensions and n unique documents in the repository.

3.1.2 Vector Representations

There are three alternative strategies for representing binary vectors and CMMs in AURA. The binary vectors may be stored as Binary Bit Vectors (BBVs) or Compact Bit Vectors (CBVs) and CMMs as BBVs, CBVs or they may be represented as Efficient Bit Vectors (EBVs). BBVs store all p bits in the binary vector, storing a 0 or 1 as appropriate for each position. CBVs store a list of the locations of the set bits in the binary vector; this is ideal for sparse binary vectors, as only a few positions need to be stored in the list. CMMs may be stored as EBVs that enable a switch at a predefined weight (number of bits set) for each CMM row. If the row vector has a lower or equal weight to the switch value then the particular row is stored as a CBV but for higher weights the row is stored as a BBV - this enables the most efficient storage implementation to be used for each individual row vector. If a row is empty then for an efficient CMM the storage overhead will simply be a NULL pointer, i.e., a pointer to an empty list so again this minimises the storage requirements. In the spell checker the rows represent the positions of letters in the words in the lexicon. Certain letters may never occur in certain positions, for example there may be no words with the second letter 'z' so this row would simply be stored as a NULL pointer. In section 5.4.1, we compare the memory usage for binary, compact and efficient CMMs storing 1.25 million input-output associations.

For the orthogonal word and document identification binary vectors we use CBVs as only one bit position needs to be stored in the bit set list which is far more memory efficient than BBVs, i.e., storing the entire bit list. The spelling and code vectors are also sparse with one bit set per letter 'chunk' (see chapter 4) so again we use CBVs. The CMMs are set to Efficient for all modules so that the switch value can be pre-set and the most efficient representation selected for each row of the CMM.

3.1.3 Training the Network

AURA uses a supervised Hebbian learning rule during training. The binary output vector associated with each binary input vector is known thus guiding the learning process and producing supervised learning. Active connections are reinforced during training emulating Hebbian Learning. The diagram (figure 3.2) shows the network after 1, 2 and 3 patterns have been trained. The input vectors are 01000000, 00100000 and 00001000 and their respective output vectors (denoted class pattern in figure 3.2) are 01000000, 00010000 and 00000100. The output vectors are fixed-length and uniquely identify their associated inputs. The CMM is set to one where an input row and an output column are both set (see figure 3.2). There is one association in the CMM per input-output pair. After storing all input-output associations, the CMM weights w_{kj} for row

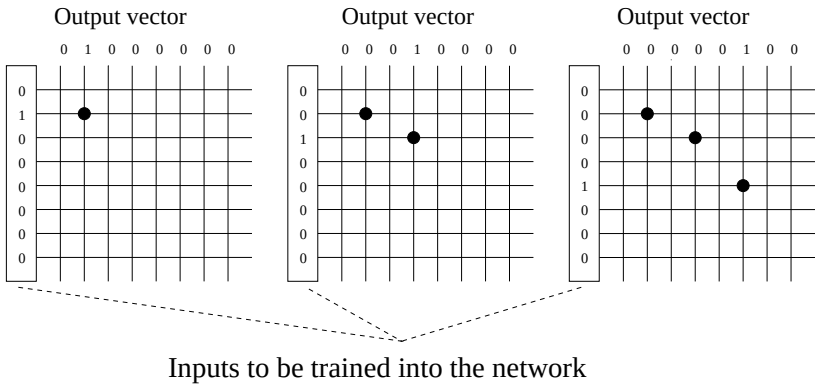

Figure 3.2: Diagram showing three stages of network training

j column k where $\vee$ and $\wedge$ are logic 'or' and 'and' respectively is given by:

$$w_{kj} = \bigvee^{\text{all } i} input_j^i \wedge output_k^i \tag{3.2}$$

3.1.4 Recalling from the Network

This process is essentially similar to the training process except the binary output vector is not applied to the CMM only the binary input vector. The columns are summed as in equation 3.3

$$act_j = \sum^{\text{all } i} input_i \wedge w_{ji} \tag{3.3}$$

and the output activation vector (denoted '*act*' in equations 3.3, 3.4 and 3.5) is thresholded to produce a thresholded, binary output vector. We use the Willshaw threshold with a threshold value equivalent to the number of bits set in the input vector to retrieve all matches (see figure 3.3). The Willshaw

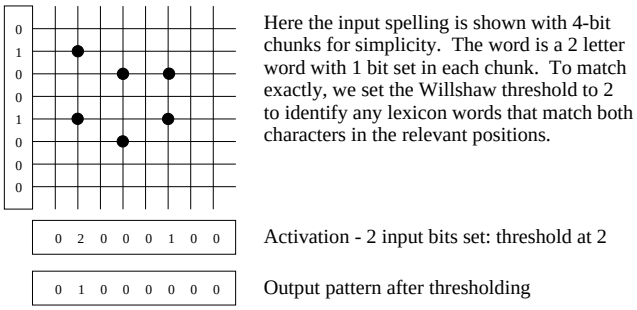

Figure 3.3: Diagram showing system recall. The input pattern has 1 bit set so the CMM is thresholded at 1.

threshold produces a binary bit vector (denoted '*threshold*' in equations 3.4 and 3.5) with a bit set to 1 where the corresponding activation output vector attribute is greater than or equal to the predefined Willshaw threshold value

(denoted 'WT' in equation 3.5) and sets the remaining bits to 0, see equation 3.5.

$$act \in \aleph^n \wedge thresholded \in \{0,1\}^n \tag{3.4}$$

$$thresholded^n = Willshaw(act^n) \text{ where}$$
$$thresholded_j = 1 \text{ iff } act_j > WT \text{ else } thresholded_j = 0 \tag{3.5}$$

We wish to retrieve all outputs that match the input. If the input has one bit set we retrieve all columns that sum to one. The thresholded output vector represents the binary output vector trained into the network and associated with the specific input vector. Even with an error in the input (see figure 3.4) the correct output vector is generated. This illustrates the ability of AURA to

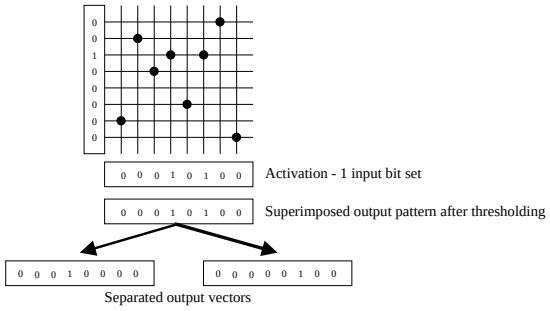

Figure 3.4: Diagram showing system recall. The input pattern is corrupted and should be 00101000. However, the output vector pattern generated by AURA is still correct.

overcome imperfect input patterns. Again the columns are summed and the activation thresholded to produce a thresholded output vector.

Multiple Inputs - Words

For the word-to-document matrix, we often need to match multiple words simultaneously. If we wish to retrieve multiple word matches, rather than serially matching the binary vectors, AURA replicates parallel matching. The binary vectors for the required words are superimposed, forming a single input vector (see equation 3.6 and figure 3.5), which may then be presented to the CMM in a single stage process.

$$inputVector = \bigvee^{\text{all } i} inputVector^i \tag{3.6}$$

The binary vectors representing the input words are logically ORed to permit commutativity thus making the order of the input words irrelevant. The input words are treated as a 'bag of words' allowing partial matching of the requisite number of words. This is essential for our spell checker and word-document matrix where we need to retrieve output vectors (words or documents respectively)

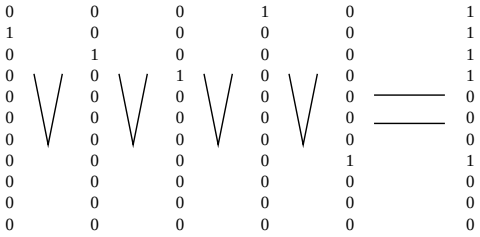

Figure 3.5: Diagram showing the superimposition of input vectors.

that match unordered subsets of the input vectors (partial spellings or subsets of words respectively).

Partial Match

If only a partial match of the input is required then this combinatorial problem is easily resolved due to the superimposition of the vectors in the input and the overlaid storage representation in the CMM. We exploit this facility to retrieve the best matching spelling from the lexicon in the hybrid spell checker in chapter 4 and to recall the documents that match subsets of input words in the word-to-document association matrix in chapter 5. For the word-to-document association matrix we can use partial match to recall any documents that match M of the N superimposed word vectors where $(M < N)$, for example, 'find all documents matching at least two of a set of four words'. An orthogonal vector represents each word. We superimpose the word vectors to form a single input vector as in figure 3.5. The input vector is presented to the CMM and the Willshaw threshold is set at $M \cdot B$ where B is the number of bits set in each separate input vector, for orthogonal word vectors B is 1. To retrieve 'all documents matching at least two of a set of four words', we set the threshold at two, see figure 3.6. This generalised combinatorial partial match provides a

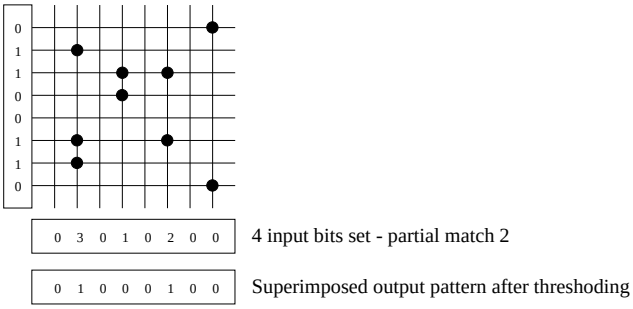

Figure 3.6: Figure showing a partial match retrieval. We want to find all outputs matching at least two of the four superimposed inputs so we threshold the output activation vector at 2.

very efficient mechanism for partial match and resembles parallel matching as

we are able to present the entire set of words to be matched in a single input step and perform the partial match in a single threshold step.

Multiple Outputs - Words and Documents

The output from a match performed by the network (a partial match or multiple input vector match) may be a single or multiple binding of individual orthogonal output vectors that match, see figure 3.7. A bit will be set in the thresholded output vector for each orthogonal output vector that matches at or above the Willshaw threshold value when the output activation vector is thresholded.

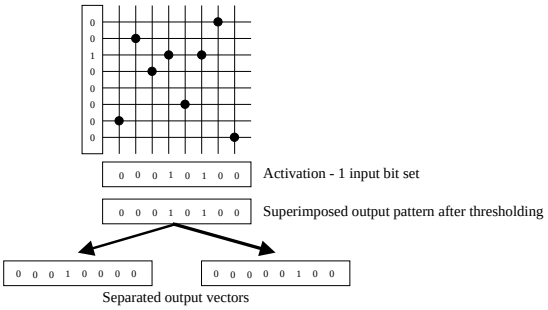

Figure 3.7: Figure showing CMM recall where thresholded output vector is a superimposition of multiple orthogonal output vectors. As the all output vectors in the system described in this dissertation are orthogonal the individual output vectors are easily identified from the superimposed output vector.

$$outputVector = \bigvee^{\text{all } i} matchingOutputVector^i \tag{3.7}$$

The superimposed outputs must be identified. All binary vectors used to represent individual output data items are orthogonal so we may simply obtain a list of the bits set in the superimposed output vector. The number of bits set corresponds to the number of matches i.e., each bit set represents a separate orthogonal output vector. The time for this retrieval of matching vectors process is proportional to the number of bits set in the output vector O(bits set), there is one matching output vector per bit set for orthogonal output vectors.

3.1.5 Orthogonal Codes

In the IR system described in this dissertation, all output vectors are orthogonal to ensure that the each output can be uniquely identified and the system will not return false positives. The advantage of using vectors with multiple bits set is data compression, the length of the binary vector may be truncated and the representation compressed so the storage overhead of the CMM is reduced. However, if more than one bit is set in the output vectors then we can get bit clashes and false positives will be returned from the retrieval (see also chapters

4, 5 and [57]) as 'ghost' matches are returned. For example if we have three words A, B and C represented by 01001000, 01000001 and 00001001 and the output should match A and C, then the superimposed output vector will be 01001001. However, B is a subset of the output vector so even though A and C are the correct match, B is a false positive. The data we employ to evaluate our IR system in this dissertation has low memory usage, a combined total of approximately 23MB for the three CMMs, so there is no need to compress the data representation.

If we increased the information stored dramatically then we may need to subdivide the data or to compress the stored representation. We provide some analyses in chapter 4 for speed of retrieval and CMM memory usage when the dataset is extremely large. We train up to 1.5 million words into a CMM to form a dictionary. We evaluate the speed of retrieval for exact and best match word spellings and the memory usage of the CMM. We note that such a large dataset would require subdivision or compression, we outline three possible methodologies for generating the binary vectors. We conclude that the optimal compression technique depends on the memory constraints of the system so we pinpoint two possible techniques that a user may select according to their memory requirements.

3.2 Conclusion

The AURA neural network architecture provides a simple, one-shot training process, a fast and highly flexible match procedure that permits partial matching - this is particularly applicable for spell checking and best matching document selection. AURA can handle multiple data types; such as word spellings, phonetic codes, word identifiers and document identifiers. AURA exhibits scalability, we demonstrate AURA's ability to handle approximately 1,250,000 word to document associations in chapter 4. These are all key requirements for our proposed IR system making AURA an ideal foundation for the modular IR system detailed in the dissertation.

Chapter 4

AURA Hybrid Spell Checker

In this chapter, we propose a simple, flexible and efficient hybrid spell checking methodology incorporating phonetic matching, supervised learning and associative matching in the AURA neural system. In our spell checker, we integrate two CMM modules: a Hamming Distance and n-gram algorithm CMM with high *recall* for typing errors and a phonetic spell-checking CMM module with high *recall* for phonetic spelling errors. Our phonetic spell checking module is a novel integration of Soundex-type codes and transformation rules. We have developed a single synergistic spell checker, integrating the retrieved words from the three spelling algorithms in the two separate CMM modules by using a novel scoring scheme to calculate an overall score for each matched word for each algorithm. From the overall scores we can rank the possible matches. In this chapter, we provide memory usage and speed of retrieval statistics for our hybrid spell checker and we evaluate our approach against several benchmark spell-checking algorithms for recall accuracy. Our proposed hybrid methodology has the joint highest recall rate of the techniques evaluated coupled with low computational cost.

We also analyse the retrieval time and memory usage statistics for the spell checker trained with various lexicon sizes. We pinpoint the lexicon size when the retrieval time increases to above 2 seconds, which is the maximum retrieval time we have deemed feasible for a spell checker [74]. The analysis also identifies the memory usage for the lexicon at this point. From the retrieval time and memory usage analyses, we make recommendations for accommodating larger lexicon sizes in CMMs, we identify the problems larger lexicons will introduce and posit recommendations for overcoming the problems identified. We have identified three possible solutions : modularity, compression and binning. Modularity subdivides the dataset so only a subsection of the total data storage needs

to be memory resident at any one time. The other two suggestions compress the representation and thus reduce memory usage.

4.1 Introduction

Errors, particularly spelling and typing errors are abundant in human generated electronic text. For example, Internet search engines are criticised for their inability to spell check the user's query which would prevent many futile searches where the user has incorrectly spelled one or more query terms. An approximate word-matching algorithm is required to identify errors and using some measure of similarity, recommend words that are most similar to each misspelt word. This error checking would prevent wasted computational processing, prevent wasted user time and make any system more robust as spelling and typing errors can prevent the system identifying the required information. An ideal spell checker could also suggest word stemming variants, if the user supplies a word stem the spell checker can return the set of variants for the word stem, expanding the query as desired and ensuring a more robust and thorough search.

We describe an interactive spell checker that performs a presence check on words, identifies spelling errors, recommends alternative spellings and suggests stemming sets from a word stem supplied by the user. The basis of the system is the AURA modular neural network described in chapter 3. The spell checker uses a hybrid approach to overcome phonetic spelling errors and the four main forms of typing errors: insertion, deletion, substitution and transposition (double substitution). We use a Soundex-type coding approach [63] coupled with transformation rules to overcome phonetic spelling errors. Phonetic spelling errors are the most difficult to detect and correct as they distort the spelling more than other error types such as typographical errors [63] so a phonetic component is essential. We use an n-gram approach [106] to overcome the first two forms of typing error and integrate a Hamming Distance approach to overcome substitution and transposition errors. N-gram approaches match small character subsets of the query term. They incorporate statistical correlations between adjacent letters and are able to accommodate letter omissions or insertions. Hamming Distance matches words by left aligning them and matching letter for letter. Hamming Distance does not work well for insertion and deletion where the error prevents the letters aligning with the correct spelling but works well for transposition and substitutions where most characters are still aligned. We have developed a novel scoring system for our spell checker. We separate the Hamming Distance and n-gram scores so the hybrid system can utilise the best match from either and overcome all four typing-error types. We add the Soundex score to the two separate scores to produce two word scores to introduce a phonetic component to both scores. The overall word score is the maximum of these two values.

Our approximate matching approach is simple and flexible. We assume the query words and lexicon words comprise sequences of characters from a finite set of 30 characters (26 alphabetical and 4 punctuation characters). The approach maps characters onto binary vectors and two storage-efficient Correlation Matrix Memories from AURA which represent the lexicon. The system is not language-specific so may be used on other languages; the phonetic codes and transformation rules would just need to be adapted to the new language. Our spell checker aims to high recall[1] accuracy possibly at the expense of precision[2]. However, the scoring allows us to rank the retrieved matches so we can limit the number of possibilities suggested to the user to the top 10 matches, giving both high recall and precision.

Some alternative spelling approaches include the Levenshtein Edit distance [63], Agrep [115] [114], Aspell [4], n-gram matching (see section 4.5 and [18]) and the two benchmark approaches MS Word 97 and MS Word 2000. We evaluate our approach against the alternatives listed above. We compare our hybrid approach against Edit Distance and n-gram for memory usage. We provide the training time for our hybrid approach and calculate the characters per second processing rate for training. We compare against n-gram, Agrep and Edit Distance for retrieval times. The n-gram, Agrep and Edit Distance algorithms were the only ones from the list above that ran on the same architecture as our system, the other techniques ran on different architectures so we could not produce a valid timing comparison for those approaches. We compare our hybrid system with all of the afore-mentioned approaches for quality of retrieval - the percentage of correct words retrieved from 600 misspelt words giving a figure for the recall accuracy with noisy inputs (misspellings).

4.2 Levenshtein Edit Distance

Levenshtein edit distance produces a similarity score for the query term against each lexicon word in turn. The score is the number of single character insertions, deletions or substitutions required to alter the query term to produce the lexicon word, for example to go from '*him*' to '*ham*' is 1 substitution or '*ham* to '*harm* is 1 insertion. The word with the lowest score is deemed the best match.

$$f(0,0) = 0 \tag{4.1}$$

$$f(i,j) = min[(f(i-1,j)+1, f(i,j-1)+1, f(i-1,j-1)+d(q_i,l_j)] \\ \text{where } d(q_i,l_j) = 0 \text{ if } q_i = l_j \text{ else } d(q_i,l_j) = 1 \tag{4.2}$$

[1] The percentage of correct spellings retrieved.

[2] The percentage of words other than the correct spelling retrieved.

For all word comparisons a function $f(0,0)$ is set to 0 as in equation 4.1 as all letters in the words are indexed from $1...n$. This forms the basis for the recursion in equation 4.2. A function $f(i,j)$ in equation 4.2 is calculated for all i query word characters and all j lexicon word characters, iteratively counting the string difference between the query $q_1q_2...q_i$ and the lexicon word $l_1l_2...l_j$. Each insertion, deletion or substitution is awarded a score of 1 (see equation 4.2). Edit distance is $O(mn)$ for retrieval as it performs a brute force comparison with every character (*all m characters*) of every word (*all n words*) in the lexicon and therefore can be slow for large dictionaries.

4.3 Agrep

Agrep [115] [114] is based upon Edit Distance and finds the best match; the word with the minimum single character insertions, deletions and substitutions. Agrep uses several different algorithms for optimal performance with different search criteria. For simple patterns with errors, Agrep uses the Boyer-Moore algorithm with a partition scheme (see [115] for details). Agrep essentially uses arrays of binary vectors and pattern matching, comparing each character of the query word in order, to determine the best matching lexicon word. The binary vector acts as a mask so only characters where the mask bit is set are compared, minimising the computation required. There is one array for each error number for each word, so for k errors there are $k+1$ arrays ($R^0...R^k$) for each word. The following two equations describe the matching process for up to k errors $0 < d \leq k$. R_j denotes step j in the matching process of each word and R_{j+1} the next step. RShift is a logical right shift and S_c is a binary vector representing the character being compared c. $\vee\wedge$ denote logical OR and AND respectively.

$$R_0^d = 11...100...000 \text{ with d bits set} \tag{4.3}$$

$$R_{j+1}^d = \text{Rshift}[R_j^d] \wedge S_c \vee \text{Rshift}[R_j^{d-1}] \vee \text{Rshift}[R_{j+1}^{d-1}] \vee R_j^{d-1} \tag{4.4}$$

$$\tag{4.5}$$

For a search with up to k errors permitted there are $k+1$ arrays as stated previously. There are 2 shifts, 1 AND and 3 OR operations for each character comparison (see [115]) so the running time quoted by Wu & Manber is $O((k+1)n)$ for an n word lexicon.

4.4 Phonetic Spell Checkers

Soundex and Phonix have been designed specifically for phonetic spell checking and produce a four-character to code to represent each word. The words are then matched by counting the number of corresponding code characters.

```
01230120022455012623010202
abcdefghijklmnopqrstuvwxyz
```

Figure 4.1: Figure listing the Soundex code mappings.

```
code[1] := word[1]; //where code and word are indexed from 1 to m
j := 2;
for n in 2 to lengthOf(word){
    If Soundex(word[n]) == 0 then skip;
    If Soundex(word[n]) == Soundex(word[n-1]) then skip;
    code[j] := Soundex(word[n]);
    j := j + 1;
}
If code has less than 3 digits then pad with 0s.
Truncate code at 4 characters to leave letter|digit|digit|digit.
```

Figure 4.2: Figure listing the Soundex algorithm in pseudocode. Skip jumps to the next loop iteration. The function Soundex() returns the appropriate code mapping for the letter.

4.4.1 Soundex

Soundex was developed early in the 20th century (see [63] for details) and maps the letters of the alphabet on to a series of numeric codes with the exception of the first letter of the word which is mapped to itself (see figure 4.1 for the codes and figure 4.2 for the algorithm). Each word is encoded by concatenating the codes of its constituent letters, ignoring 0's and only indexing the first letter if adjacent letters in the word map to identical letter code values. In Soundex the word code is limited to 4 characters (first letter|0-6|0-6|0-6). If there are less than 4 non-zero characters in the word code then the code is padded with 0's. Words can then be compared for phonetic similarity by counting the number of corresponding code characters; the same code character *AND* the same code position. However, Soundex retrieves many false positives during matching due to a large set of letters mapping to each letter code and many of these false positives do not sound-like the target word. Soundex also does not rank the retrieved matches, words either match or they do not; the only score available is a count of the number of matches in the word code (from 0 to 4).

4.4.2 Phonix

Phonix [39], [40] is an extension of Soundex, generating 4 character word codes using the same basic algorithm as Soundex (listed in figure 4.2) but mapping the letters to different letter codes compared to Soundex. The letter encodings are listed in figure 4.3. The Phonix letter codes are numbered from 0-8

```
01230720022455012683070808
abcdefghijklmnopqrstuvwxyz
```

Figure 4.3: Figure listing the Phonix codes for each letter.

to allow more letters to be distinguished and to reduce the number of words mapping to each 4-character word code. Prior to producing the word code, the letters are processed using transformation rules, see [39] for the complete list of transformation rules. There are approximately 90 rules, each with a given priority and any that apply are implemented in priority order [39]. The rules are designed to implement context where letter pronunciation differs according to the surrounding letters. The increased number of letter codes and the use of transformation rules improves the quality and uniqueness of the word codes generated and thus produces less false positives during matching compared to Soundex coding alone. However, [121] showed that Phonix was scarcely better than Soundex for recall and precision accuracy. We note that Phonix was designed for South African name matching and includes many rules which are redundant for our English language spell checker, hence we do not include the Phonix system in our later evaluation. We use a much smaller rule base for our spell checker focussed solely on the etymology covered by the Concise English Dictionary. Rogers and Willett [86] extended Phonix but their implementation was designed for 17th century spellings which would not be applicable for our modern IR system.

4.5 Binary N-Grams

An n-gram is an ordered subset of the letters of a word with cardinality n. For example, 'the', 'her', 'ere' are the 3-grams of 'there'. A binary n-gram maps each individual n-gram onto one attribute of a binary vector as in equation 4.7. A binary vector representing a word is thus a logical OR of the binary vectors for all n-grams in the word. For each lexicon word, all n-grams are determined, the respective bits set in a binary vector and stored in a dictionary D in equation 4.6 where the set of n-grams present in the word is associated with the word.

$$D = \forall_i word_i \rightarrow \{nGram_{i1}, nGram_{i2}, ..., nGram_{ix}\} \tag{4.6}$$

Misspelt words can then be converted to their constituent n-grams and compared with the dictionary. The dictionary word with the highest number of common n-grams to the misspelt word is retrieved as the best matching spelling. N-gram spell checking offers reasonable recall performance, but pays a large storage penalty. There are two variants for binary n-gram matching where each word is represented by an equal length binary vector and the n-grams present in a word are denoted by setting the appropriate bit in the binary vector [106], [83]:

Positional. Each n-gram in the word is mapped onto a binary vector that represents both the n-gram and its position in the word. The bit to set is determined by equation[3] 4.7 which calculates the bit to set from both the letters of the n-gram and the n-gram's position in the word [106].

$$bitset = 26^n\mu + 26^{n-1}x_1 + 26^{n-2}x_2 + ... + x_n \tag{4.7}$$

$x_1, x_2, ..., x_n$ are integers representing the n characters of the n-gram (usually their position in the alphabet) and $\mu = 0$ for the first n-gram in the word, $\mu = 1$ for the second and so on. Hence each n-gram in the word is associated with a separate array of 26^m bits in the binary vector. The positional n-gram has superior detection and correction of spelling errors due to the positional storage [51]. However, storing the position creates a huge storage overhead, positional n-grams require 64 times the storage of non-positional n-grams for an equivalent word [51]. More systems use the non-positional approach due to this large storage overhead, mitigating the slightly lower recall by the huge storage reduction.

Non-positional. The non-positional variant does not store the position of the n-gram only setting a bit in the binary vector if the corresponding n-gram is present. The bit to set is determined by equation 4.7 with $\mu = 0$ for all n-grams. By not storing the positional information, the non-positional approach vastly reduces the storage overhead but will inevitably retrieve false positives. For example, for the non-positional approach both 'other' and 'there' have two 3-grams in common with 'ther'. However, for the positional approach only 'there' has two common 3-grams due to the additional positional information

Our n-gram technique described later is a hybrid of the positional and non-positional approaches, we do not note the position of the n-gram within a word but we limit matching to the length of the spelling plus 2 characters. The n-gram is shifted along in a moving window from the beginning to the end of the word and we detect any n-gram matches during shifting. This allows us to use the same binary vectors to represent the spellings and their associated words for the n-gram as the Hamming Distance approach. This synergy minimises the storage overhead of the spell checker. The CMM storing vectors for both our shifting n-gram and Hamming Distance uses one third of the storage of the standard non-positional technique, see section 4.7.1. The non-positional n-gram approach in turn has a much lower storage requirement than the positional technique as described above. However, the shifting n-gram is slower to match than the non-positional approach due to the overhead of shifting each n-gram along the length of the input spelling, see section 4.7.3 and table 4.4.

[3]The equation applies to the 26 alphabetic characters only. If the space character is included then 27 must be substituted for 26.

Cavnar [18] uses a non-positional superimposed binary n-gram encoding scheme mapping the n-dimensional n-grams onto a single dimension binary vector but with each bit representing multiple n-grams rather than just one n-gram as described above. The system arranges the n-grams into sets ('bins') with an equal n-gram frequency distribution between the sets. Each set can then be mapped onto a single bit in the binary vector. This reduces the storage required for a dictionary of word $\rightarrow$ n-gram associations but pays the penalty of introducing more false positive retrievals than the standard non-positional approach. Consequently, many words will be represented by identical binary vectors as they contain n-grams that map to the same bins. The approach does not store any frequency or positional information which would both help reduce the false positives. The technique may be further extended to encode every n-gram with 'an ensemble of binary vectors and yield even greater space savings'. Our shifting n-gram technique uses 1-grams, 2-grams and 3-grams depending on the length of the spelling to improve matching. Cavnar [18] similarly uses both 2-grams and 3-grams.

An alternative n-gram approach using only 3-grams and implemented in AURA builds upon Cherkassky [23]. For the analysis here we use the AURA 3-gram methodology as it is essentially similar to both Cherkassky [23] and Cavnar's [18] methodologies - it does not implement the binning step of Cavnar but rather maintains separate bits for each distinct 3-gram so retrieves less false positive matches than Cavnar's binning approach.

In our AURA 3-gram technique evaluated in this chapter, a 3-dimensional representation of all of the possible 3-grams is produced with the x-dimension representing the first letter of the 3-gram, the y-dimension the second letter and the z-dimension the third. The characters are represented by 30-bit chunks

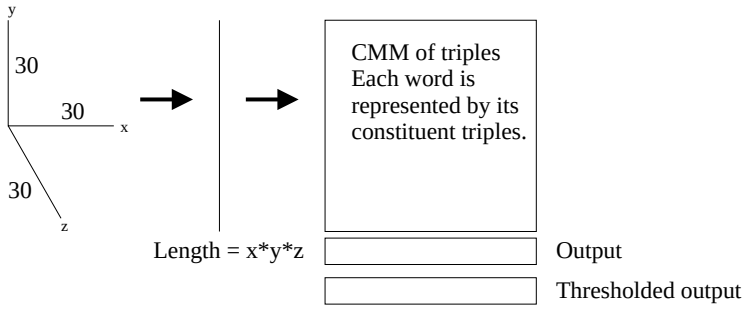

Figure 4.4: Diagram showing the triple mapping process

to allow the 26 alphabetic characters, space, and three punctuation characters to be represented, see figure 4.5. The three-dimensional representation is then mapped onto a single-dimension vector of length 30x30x30 with one bit position

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	-	'	&	/

Figure 4.5: Table indicating which bit is set in the 30-bit chunk representing each character.

for each possible triple (see figure 4.4 and equation 4.8).

$$bitset = 30^n + 30^{n-1}x_1 + 30^{n-2}x_2 + ... + x_n \tag{4.8}$$

Each word to be trained is subdivided into its constituent 3-grams and the appropriate bits are set in the 1-D vector. The output vector to be trained into the AURA 3-gram CMM is again a single-bit set (orthogonal) binary vector. There is no length limit imposed while matching the non-positional n-gram. Thus, an n-gram from the first three letters of the mis-spelling would match characters 48, 49, and 50 of a word in the lexicon if the respective n-grams were identical. Our shifting n-gram approach is a hybrid of the positional and non-positional techniques and limits matching to the length of the input plus two characters. The positional n-gram approach aligns the n-grams exactly by position and is thus too exact and susceptible to insertion or deletion spelling errors as the n-grams will no longer align by position in the mis-spelling and correct lexicon word. Also, the strict 3-gram approach does not work well for less than 5 character words as there may be no common triples between the query and the correct spelling. Therefore, for our shifting n-gram we use 1-grams for up to 4 letters and 2-grams for 4 to 6 letter words and 3-grams only for words longer than 6 letters and Cavnar combines 2-grams and 3-grams in his approach.

4.6 AURA

AURA, described in chapter 3 and evaluated in chapters 4 and 5, is ideal for spell checking compared to other neural networks as AURA does not suffer from the lengthy training problem of other neural networks. Storage is efficient in the CMMs as new inputs are incorporated into the matrix and do not require additional memory allocation. AURA is also able to partially match inputs, which is a prerequisite for any spell checker to permit the retrieval of the best matching word.

4.6.1 Our Methodology

In our spelling system, we use two CMMs: one CMM stores the words for n-gram[4] and Hamming Distance matching and the second CMM stores phonetic

[4]N.B. Hamming Distance and n-gram cannot be used simultaneously we have to perform Hamming Distance match, retrieve the candidate matches then perform n-gram match and

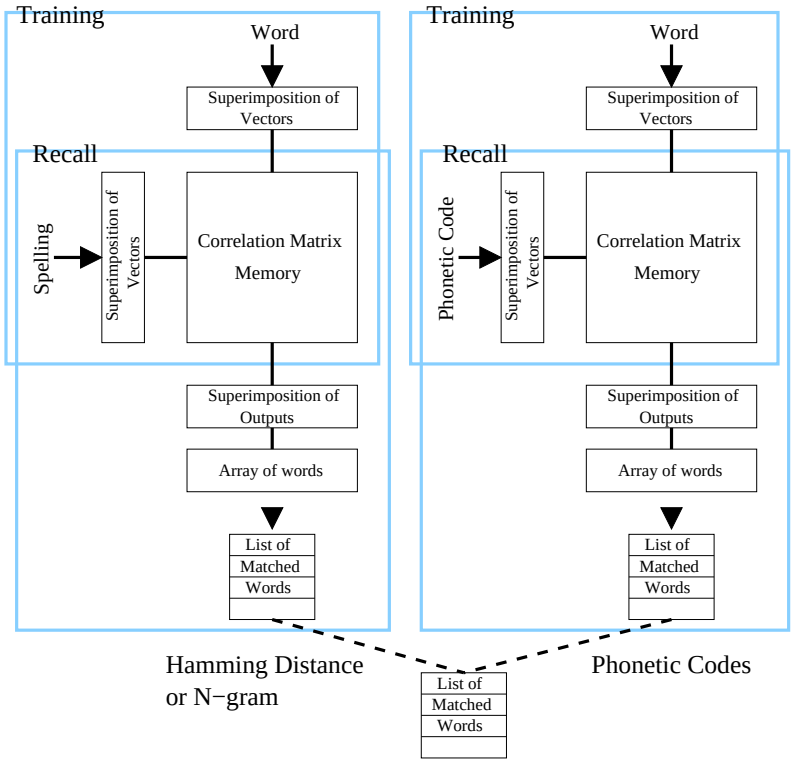

Figure 4.6: Diagram of the hybrid spell checker as implemented in the AURA modular system.

word codes for homophone matching. The CMMs are used independently but the results are combined during the scoring phase of the spell checker to produce an overall score for each word.

4.6.2 Hamming Distance and N-Gram

For Hamming Distance and n-gram, the word spellings form the inputs and the matching words from the lexicon form the outputs of the CMM. As stated previously in section 4.5, we use an identical representation for the spellings and associated matching words for both Hamming Distance and n-gram. For the inputs, we divide a binary vector of length 960 into a series of 30-bit chunks. This allows us to store up to 30 characters for each letter (a-z and 4 punctuation characters). Words of up to 30 characters may be represented and the longest word in the lexicon for our evaluation in this thesis is 29 characters. For the shifting n-gram described later, we need two additional character chunks to allow shifting so we need a vector of length 30 x (30+2) giving 960 bits. Each word is divided into its constituent characters. The appropriate bit is set in the chunk to represent each character in order of occurrence (we use the same

retrieve the candidate matches before combining all candidate matches using our scoring scheme

mappings as for the n-gram bit setting, see figure 4.5). The chunks are concatenated to produce a binary vector to represent the spelling of the word and form the input to the CMM. Any unused chunks are set to all zero bits. We use Compact Bit Vectors for the word spelling vectors as there is only a single bit set in the vector for each character in the word so the vector is sparse and CBVs will provide the most efficient storage.

Each word in the alphabetical list of all words in the text corpus has a unique orthogonal binary vector to represent it, which forms the output from the CMM (see equation 4.9). Essentially, we associate the spelling of the word to an orthogonal output vector to uniquely identify it.

$$bitVector^{p} = p^{th}\text{bit set } \forall p \text{ for p = position\{words\}} \tag{4.9}$$

We use Compact Bit Vectors for the word output vectors as there is only a single bit set in the vector so CBVs will provide the most efficient storage representation. The CMM represents the lexicon of all words in the corpus; the inputs are the spellings and the outputs the matching words. We use an efficient CMM for the lexicon. The spelling-word matrix is reasonably sparse so we can exploit the most memory efficient representation for each row, storing sparse rows as Compact Bit Vectors and dense rows as Binary Bit Vectors. We empirically evaluated the switch value (the number of bits set when the row is converted to a binary representation from a compact representation) for the CMM to minimise memory usage.

4.6.3 Phonetic Spell Checking

Our methodology combines Soundex-type word codes with phonetic transformation rules as with Phonix [40] to produce a four-character code for each word. In their paper [86], Rogers & Willett evaluated the recall and precision performance for their phonetic approach with varying length phonetic codes. They noted that four-character codes had higher recall with longer codes having higher precision. We combine the scores for the best matching words with the scores from the Hamming Distance and n-gram best matches so we feel higher recall is more important than higher precision. The integrated scoring mechanism will relegate many false positives from the phonetic matching to the foot of the ranking list. We studied the etymology of the English Language given in the preface to the Concise Oxford Dictionary to generate our phonetic rule base and integrated a few rules from Phonix and Aspell that we deemed the most important. However, we minimise our rule base as checking each constituent letter of the word against the rule base is computationally time consuming. We also use a larger number of letter codes than Soundex and Phonix. We use fourteen codes compared to seven for Soundex and nine for Phonix to preserve the letter similarities yet prevent dissimilar letters mapping to the same code character,

^hough → *h5*	^rough → *r3*
^cough → *k3*	^tough → *t3*
^chough → *s3*	^enough → *e83*
^laugh → *l3*	^trough → *tA3*
^ps → s	^wr → r
^pt → t	^kn → n
^pn → n	^gn → n
^mn → n	^x → z
sc(e|i|y) → s	1. (i|u)gh(¬a) → _ 2. gh → g
+ti(a|o) → s	gn$ → n
ph → f	gns$ → ns
1. c(e|i|y|h) → s 2. c → k	q → k
mb$ → m	+x → ks

Table 4.1: Table of the phonetic transformation rules in our system. We obtained a few rules from Aspell [4] and Phonix [40], for the remaining rules we studied the lexicon and English grammar. Italicised letters are phonetic codes - all other letters are standard alphabetical letters. ^ indicates 'the beginning of a word', $ indicates 'the end of the word' and + indicates '1 or more letters'. Rule 1 is applied before rule 2.

which is a problem for both Soundex and Phonix. The letters are translated to code characters indexed from 0-D in hexadecimal as we need single character representations for each word code character to guarantee that all word codes are four characters. We can minimise false positives as only very similar words map to the same word code. Both Soundex and Phonix have low recall precision as they generate many false positive matches [121] due to the limited number of possible word code permutations.

For our method, any applicable transformation rules are applied to the word. The phonetic transformation rules are given in table 4.1. The code for the word is generated according to the algorithm in figure 4.7 using the letter mappings listed in figure 4.8. If the phonetic code produced by the algorithm has less than 4 characters then the word code is padded with 0's to ensure all word codes are exactly four characters. The prefixes {hough, cough, chough, laugh, rough, tough, enough, trough } are converted to their respective phonetic codes given in table 4.1 and our phonetic algorithm then converts any remaining letters in the word to their phonetic mapping. For example, *laughs*, the prefix 'laugh' is converted to '*l*3' and the remaining letter 's' is converted to '*B*' using the algorithm in figure 4.7 producing a phonetic code '*l*3*b*0' for 'laughs' after the spare code character has been padded with a 0.

For phonetic spelling, the phonetic word codes form the inputs and the matching

```
Apply all applicable phonetic transformation rules;
code[1] := word[1]; // where code and word are indexed from 1 to m
j := 2;
for n in 2 to lengthOf(word){
    If Soundex(word[n]) == 0 then skip;
    If Soundex(word[n]) == Soundex(word[n-1]) then skip;
    code[j] := Soundex(word[n]);
    j := j + 1;
}
If code has less than 3 digits then pad with 0's.
Truncate code at 4 characters to leave letter|digit|digit|digit.
```

Figure 4.7: Figure giving our algorithm in pseudocode. The function Soundex() returns the code value for the letter as detailed in figure 4.8. Skip jumps to the next loop iteration.

```
01-2034004567809-ABC0D0-0B0000
abcdefghijklmnopqrstuvwxyz-'&/
```

Figure 4.8: Figure giving our codes for each letter: (c, q and x do not have a code as they are always mapped to other letters by the transformation rules: c→s or c→k, q→k and x→z or x→ks).

words from the lexicon form the outputs of the CMM. Each word is converted to its 4 character phonetic code. For the input (phonetic) vectors, we divide a binary vector of length 62 into an initial alphabetical character representation (23 characters as c, q and x are not used) and three 13-bit chunks. Each of the three 13-bit chunks represents a phonetic code from figure 4.8 where the position of the bit set is the hexadecimal value of the letter code. The chunks are concatenated to produce a binary vector to represent the phonetic code of the word and form the input to the CMM. We use CBVs for the phonetic word code binary vectors as they are sparse with only four bits set in a vector of length sixty-two. CBVs are the most efficient representation for sparse vectors as there are only four positions stored in the list of bits set. A BBV would necessitate storing a binary value for all sixty-two bits which would increase the storage overhead considerably.

As with the Hamming Distance and n-gram module, each word in the alphabetical list of all words in the text corpus has a unique orthogonal binary vector to represent it and form the output from the phonetic CMM. A single bit is set corresponding to the position of the word in the alphabetical list of all words (see equation 4.10). We use the same representation for the word vector as used by the Hamming Distance and n-gram CMM module detailed in this chapter

and also the word-to-document association CMM detailed in chapter 5. This maintains uniformity and allows simple processing; we can combine binary vector outputs from the two spelling CMMs and use the superimposed outputs as inputs to the word-to-document association CMM.

$$bitVector^{p} = p^{th}\text{bit set } \forall p \text{ for p = position\{words\}} \tag{4.10}$$

The phonetic CMM represents the lexicon as 4-character phonetic codes. Again we employ an efficient CMM with an empirically evaluated switch value to minimise memory usage for the phonetic lexicon. The CMM switches between the most efficient representations for each row.

4.6.4 Training the Network

The binary patterns representing the word spellings or phonetic codes are input to the appropriate CMM and the binary patterns for the matching words form the outputs from the respective CMM. The two spell checker CMMs are trained as described in section 3.1.3. After training, there is one association in the n-gram & Hamming Distance CMM per spelling-word pair. There is one association in the phonetic CMM per phonetic code-word pair.

4.6.5 Recalling from the Network - Hamming Distance

For recall only the spelling pattern is applied to the network. The columns are summed to produce an output activation vector.

$$output_j = \sum^{\text{all } i} input_i \wedge w_{ji} \tag{4.11}$$

The output activation vector is thresholded to produce a binary output vector (see figure 4.9). The binary output vector represents the word originally trained

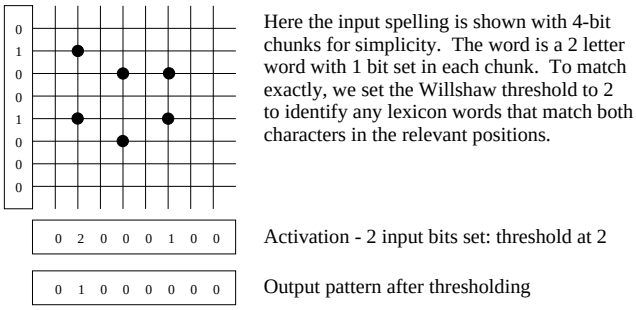

Figure 4.9: Diagram showing system recall. The input pattern has 1 bit set so the CMM is thresholded at 1.

into the network matching the input spelling just presented to the network. We use the Willshaw threshold (see chapter 3) set to the highest activation value to retrieve the best matches(see figure 4.9). We wish to retrieve all outputs that

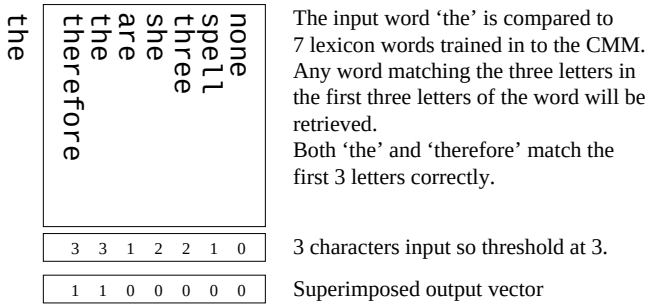

Figure 4.10: Diagram showing Hamming Distance matching

match as many characters in the input as possible (see figure 4.10). The output vector is a superimposition of the orthogonal vectors representing the words that match the maximum number of the input word's characters. This partial match provides a very efficient, single-step mechanism for selecting those words that best match.

We are able to use the '?' convention from UNIX for unknown characters in our Hamming Distance matching by setting all bits in the chunk (representing a *universal OR*), i.e., the chunk represents a 'don't care' during matching and will match any letter or punctuation character for the particular letter slot. For example, if the user is unsure whether the correct spelling is 'separate' or 'seperate' they may input 'sep?rate' to the system and the correct match will be retrieved as the chunk with all bits set will match 'a' in the lexicon entry 'separate'.

4.6.6 Word Stems

For word stem matching, we implement a similar process to the Hamming Distance match but we set the Willshaw threshold value to the number of bits in the input vector, i.e., the number of characters in the input word. Thus the

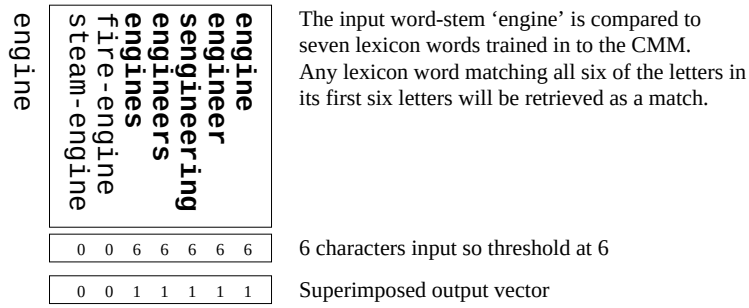

Figure 4.11: Diagram showing word stem matching

input word forms a stem and all word stemming variants for which the input word is a stem are retrieved. Figure 4.11 shows a stemming recall. The input

word stem '*engine*' is converted to a binary vector using the same technique as the Hamming Distance conversion, converting each letter to a binary chunk using the position of the word in the alphabet as an index to set the bit and concatenating all chunks in the order of the constituent letters to generate a vector for the word. The binary word vector is input to the standard Hamming Distance & n-gram CMM. The threshold is set to the number of letters in the word stem (6 for '*engine*') and any word matching all 6 input letters in its first 6 letters is retrieved as a matching stemming variant.

4.6.7 Recalling from the Network - Shifting N-Grams

We utilise the same CMM for n-gram matching that we use for Hamming Distance matching. We use three n-gram approaches switching between the three approaches as appropriate. We use 1-grams for spellings with less than four characters, 2-grams for 4 to 6 characters and 3-grams for spellings with more than 6 characters. Misspelt words with less than four characters are unlikely to have any 2-grams or 3-grams found in the correct spelling, for example 'teh' for 'the' have neither common. Spellings with four to six characters may have no common 3-grams but should have common 2-grams and words with more than 6 characters should match 3-grams. We experimented with just using 1-grams and 2-grams but we found that 1-grams and 2-grams produced too many false positive matches during our empirical evaluation so we opted to employ 3-grams particularly as the module is incorporated with other matches in our hybrid system so we did not want too many low scoring matches to process and slow retrieval.

Again we can employ the '?' convention from UNIX by setting all bits in the appropriate input chunk. We describe a recall for a 7-letter word ('theatre') using 3-grams below and in figure 4.12. All our n-gram techniques (1-gram, 2-gram and 3-gram) operate on the same tenet, we just vary the size of the comparison window.

We take the first three characters of the spelling 'the' and input these left aligned to the spelling CMM as in the left-hand CMM of figure 4.12. We wish to find lexicon words matching all three letters of the 3-gram, i.e., all words with an output activation of three for their first three letters. In the left-hand CMM of figure 4.12, 'theatre' and 'the' match the first three letters so their corresponding output activations are three. When we threshold the output activation vector at the value three, the bits set in the thresholded output vector correspond to a superimposition of the bits set to uniquely identify 'the' and 'theatre'. We then slide the 3-gram one place to the right, input to the CMM and threshold at three to find any words matching the 3-gram. We continue sliding the 3-gram to the right until the first letter of the 3-gram is in the position

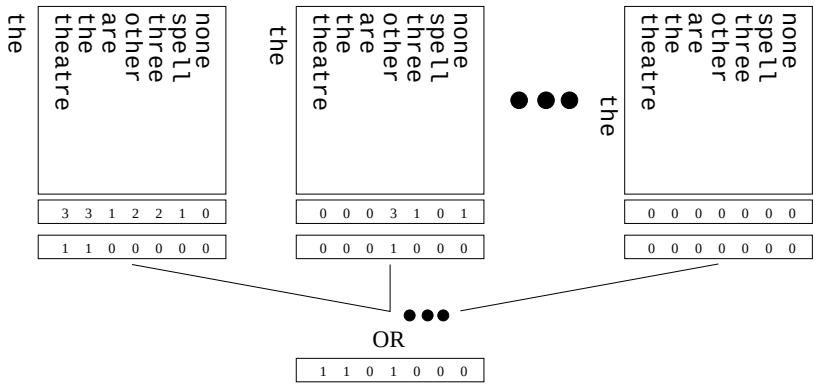

Figure 4.12: Diagram showing a 3-gram shifting right

of the last character of the spelling and the third letter of the n-gram is aligned with the last letter of the input plus two characters. We match the length of the input plus two characters as nearly all spelling mistakes are within two letters of the correct spelling [63]. We logically OR the output vector from each 3-gram position to produce an output vector denoting any word that has matched any of the 3-gram positions for this 3-gram, see figure 4.12. We then move onto the second 3-gram 'hea', left align, input to the CMM, threshold and slide to the right producing a second 3-gram vector of words that match this particular 3-gram in any place. When we have matched all m 3-grams {'the', 'hea', 'eat', 'atr', 'tre'} from the spelling, we will have m output vectors representing the words that have matched each 3-gram respectively. We sum these output vectors to produce an output activation vector representing a count of the number of 3-grams matched for each word. We then threshold at the maximum value of the output activation vector to produce a thresholded output vector with bits set corresponding to the best matching CMM columns (word), i.e., the columns (words) with the highest activation.

4.6.8 Recalling from the Network - Phonetic

The recall from the phonetic CMM is essentially similar to the Hamming Distance recall. We input the 4-character phonetic code for the search word into the CMM and recall a vector representing the superimposed outputs of the matching words. The Willshaw threshold is set to the maximum output activation to retrieve all words that phonetically best match the input word.

4.6.9 Superimposed Outputs

Partial matching generates multiple word vector matches superimposed in a single output vector after thresholding (see figure 4.10). These outputs must be identified. A list of all words in the lexicon is held in an array. The position of any set bits in the output vector corresponds to that word's position in the

array (see equations 4.9 and 4.10). By retrieving a list of the bits set in the output vector, we can retrieve the matched words from the corresponding array positions (see figure 4.13). The time for this process is proportional to the number of matching words Θ(words) for orthogonal output vectors.

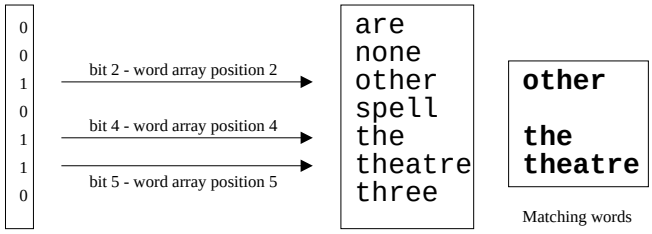

Figure 4.13: Diagram showing matching word retrieval using the bits set in the thresholded output vector as positional indices in to the word array.

4.6.10 Integrating the Modules

For exact matching (checking whether a word is present in a lexicon) we use the Hamming Distance and a length match. We perform the Hamming Distance (see section 4.6.5), thresholding at the length of the input spelling (number of bits set) to find all words beginning with the input spelling. In figure 4.10, the input would be thresholded at three (length of 'the') to retrieve {'the','therefore'}. To count the number of characters in each stored word, we input a binary vector with all bits set to one and threshold the output at the exact length of the input spelling (number of bits set). There is one bit per character so if all bits are activated and summed we effectively count the length of the word. N.B., we could have stored the length of each word with the word array. However, we felt the additional storage overhead created was unnecessary particularly as our exact match is rapid (0.03 seconds for a 29-letter word from a 29,187 word lexicon) so this additional length match step in the CMM is negligible with respect to time and creates no additional storage. From figure 4.10, all bits are set in the input and the output thresholded at exactly 3 to retrieve the 3 letter words {'the', 'are', 'she'}. We can then logically AND the Hamming Distance output vector with the length vector to retrieve the exact match if one exists, i.e., matching all input characters AND the exact length of the input. A list of the bits set in the logically ANDed binary vector is generated to index the matching words in the word array (position of bit set equals location of matching word in word array). The single matching word is {'the'}.

If the exact match returns *false* (no matches), we assume the query word is spelt incorrectly and we can then spell check using the query word to produce a list of alternative spellings. We translate the word to the spelling binary vector for Hamming Distance and present this binary vector to the Hamming Distance

& n-gram CMM. We then present the shifting n-gram vectors to this CMM. We translate the word to the phonetic code, transform this word code to a binary vector and present the binary vector to the phonetic CMM. See section 4.6.1 for details of how the input vectors are produced. We generate a separate output vector for each method with an output activation set for each word in the lexicon. We threshold the output activation vector for the Hamming Distance method at the highest attribute value to retrieve the best matching words. We also threshold the output activation vectors for the shifting n-gram method and the phonetic spelling method at their respective highest attribute values to identify their respective best matches. The resultant best matching words are retrieved from the word array for each of the three thresholded vectors in turn, i.e., we identify the array locations of the best matching words from the positions of the bits set in each thresholded vector

With our maximum value thresholding, we only score the absolute best matching words for each of the three spelling approaches in turn. It is possible that the overall best matching word may be the second best match for n-gram and second best for phonetic and would thus not receive a score from our approach. To overcome this, we could calculate a Hamming Distance score, an n-gram score and a phonetic score for every lexicon word. However, this would be computationally too expensive as we would also need to normalise all scores to ensure that none of the three methods biases the overall score calculation. This would inevitably slow the spell checking process too much. We have demonstrated the high recall of our approach so we feel the negligible increase in recall introduced by calculating scores for all words cannot mitigate the increase in retrieval time.

We produce three separate scores: one for the Hamming Distance best match (see equation 4.12), one for the shifting n-gram best match (see equation 4.13) and one for the phonetic best match (see equation 4.14) which we integrate to produce an overall word score (see equation 4.15). We evaluated various scoring mechanisms. Our first approach was to simply sum all three scores to produce a word score but this favours words that match reasonably well on both n-gram and Hamming Distance but they are usually complementary, as stated earlier in this chapter. We evaluated the n-gram scoring mechanisms in [63] ($2 * c/(n+n')$ and $c/max(n,n')$ where c is the number of common n-grams and n and n' are the lengths of the query word and lexicon word respectively) but found all were inferior to our scoring in equation 4.13. We keep the Hamming Distance and n-gram scores separate as they are complementary and we add the Soundex score to each of these two scores. We normalise the Soundex and Hamming Distance score to give a theoretical maximum score for each, equivalent to the maximum score possible for the n-gram so none of the three methods integrated biases the overall word score; they are all equally influential. In the following equations,

diff = ($strlen$(queryWord) − $strlen$(lexiconWord)) where $strlen()$ returns the length of the word

$$\text{ScoreHamming} = 2 * (\text{WillThresh} - \text{diff} - ((2 * |\text{n-gram}|) - 1)) \tag{4.12}$$

$$\text{ScoreN-gram} = 2 * (\text{WillThresh} - \text{diff}) \tag{4.13}$$

$$\text{ScorePhonetic} = \frac{2 * (\text{WillThresh} - \text{diff})}{(2 * |\text{phoneticCode}|)} * (strlen(\text{queryWord}) - (|\text{n-gram}| - 1)) \tag{4.14}$$

The score for the word is then given by equation 4.15:

$$\begin{aligned} \text{Score} \quad = \quad & max((\text{ScoreHamming} + \text{ScorePhonetic}), \\ & (\text{ScoreN-gram} + \text{ScorePhonetic})) \end{aligned} \tag{4.15}$$

The word score exploits the complementary n-gram and Hamming Distance scores and introduces a phonetic aspect to both n-gram and Hamming Distance. Hamming Distance and n-gram work well for typographical errors but less well for phonetic errors. By adding the Soundex score we can introduce homophone scoring and thus balance the word scores equally between all forms of spelling error (typographical and phonetic).

We can also provide stemming sets from word stems to allow the user to see all word forms derived from their word stem that are present in the lexicon. This can assist matching by expanding the query and providing more flexibility and robustness. Documents that may have been missed because they contain alternative stemming variants will now be retrieved. For example, if the user supplies 'engine' we can retrieve 'engines', 'engineer', 'engineering' as in figure 4.11 and allow the user to select from the stemming set any words they feel are relevant to the query.

4.7 Evaluation

In this dissertation we evaluate our 'Hybrid CMM' spell checker against the benchmark spell checkers detailed in this chapter for memory use, training time, retrieval time and spelling recall. For all evaluations we use the lexicon from the UNIX 'spell' program supplemented with the correct spellings of our test words and comprising 29,187 words and 242,819 characters in total. All analyses were performed on a SGI Origin 2000 with the following specifications (taken from the IRIX *hinv* command):

- 32 X 180 MHZ IP27 Processors
  This allows us to evaluate the various spell checking techniques in parallel, each technique evaluated on one processor simultaneously to provide identical system conditions for each technique evaluated.

Method	Size (bytes)
Spelling CMM	1,480,144
Triple mapping method	3,827,680
Edit Distance word array	1,475,000
Phonetic CMM	460,464

Table 4.2: Table listing the memory sizes of comparable data structures used to store words and their spellings.

- CPU: MIPS R10000 Processor Chip Revision: 2.6
- FPU: MIPS R10010 Floating Point Chip Revision: 0.0
- Main memory size: 8192 MBytes
- Instruction cache size: 32 KBytes
- Data cache size: 32 KBytes

4.7.1 Memory Use

We compare the memory usage of the two trained CMMs (one storing word spellings for n-gram and Hamming Distance matching and the other storing the 4-character phonetic codes) against a standard static word array data structure of 29,187 array elements with up to 50 characters in each element - declared in C as *char wordArray[29187][50]* and against the 3-gram approach (denoted 'triple mapping' and described in section 4.5) where all possible 3-grams are mapped onto a 1-D vector and the appropriate bits are set for the 3-grams present in each spelling.

From table 4.2, our spelling CMM uses approximately one third of the memory used for the triple mapping approach that stores the same information (word spellings). The triple mapping requires vectors of length 2,700 (there are 30 possible characters for each attribute of the 3-gram giving 30*30*30 possible character permutations). Our spelling approach requires vectors of length (30 bits per character * maximum word length to be stored). Here the maximum word length is 29 so we elected to use 960-bit vectors (30 characters + 2 extra for the shifting n-gram). The matrix for the triple mapping is very sparse as many of the triple possibilities never occur in the English language, for example 'xyz', 'aaa', 'zzz'. The memory usage of the Edit distance word array and the spelling CMM are approximately equivalent although of course we need to store a data structure that maps the words to the binary vectors. We have elected to use the word array as described in section 4.6.1 which is identical to the Edit Distance array so our memory usage is actually double the Edit Distance (i.e., CMM plus word array). The phonetic CMM memory usage is much lower than

the other data structures as all words are represented by 4 character codes so the CMM is much smaller (number of words *by* 4 x 24 bits per character) and thus the CMM memory usage is much lower. The total memory usage for the 'Hybrid CMM' spell checker is 1,480,144 + 460,464 + 1,475,000 = 3,415,608 bytes.

4.7.2 Training Time

For the spelling CMM, the training time was: the time to produce the binary vectors for each word spelling, to associate each word with an orthogonal output vector and train the input spelling-output word vector association into the memory matrix. For the phonetic CMM, the training time was the time to produce the phonetic word code, to produce the binary vector for the phonetic word code, to associate each phonetic code vector with an orthogonal output vector and train the phonetic code-output word vector association into the memory matrix. We train the two CMMs consecutively. We generate 1. the orthogonal output vector, 2. the spelling vector, 3. the phonetic code and from the phonetic code we generate 4. the phonetic vector, training 1 and 2 into the spelling CMM and 1 and 4 into the phonetic CMM for each word in turn. Our 'Hybrid CMM' approach processes the 242,819 characters of the UNIX 'spell' dictionary and trains the two CMMs in 7.2 seconds. Therefore the approach may process 33,725 characters per second for training. Training is only performed once and is a single epoch process.

4.7.3 Retrieval Time

We compare the retrieval times for an exact match search and a best match search for each of the approaches described in the chapter. For comparison we use a short word 'the' comprising one 3-gram and a very long word that we added to the lexicon for evaluation purposes 'floccinaucinihilipilification' with twenty-seven 3-grams. The length variation enables us to compare the running time for each algorithm and see whether they are closer to $O(n)$ or $O(1)$, i.e., dependent on the length of the input or independent and approaching constant time for retrieval.

Exact Match

Our methodology (denoted 'Hybrid CMM') implements exact match through a Hamming Distance and length comparison as this is faster than the shifting n-gram approach from empirical comparison ($O(1)$ compared to $O(n)$). The triple mapping approach retrieves the words matching all triples and retrieves the words of equivalent length to the input in a second output vector and similarly to our approach, forms a logical AND of the two output vectors to retrieve the word that have all triples in common AND are equivalent length to the

Method	Time (secs) for 'flocci...'	Time (secs) for 'the'
'Hybrid CMM'	0.04	0.04
Triple mapping	0.03	0.02
Agrep	0.01	0.01
Levenshtein Edit Distance	1.93	0.22

Table 4.3: Table listing the exact match retrieval times for the systems evaluated. Our hybrid system uses the logical AND of the output vectors from the Hamming Distance and length match for the exact match procedure.

Method	Time (secs) for 'flocci...'	Time (secs) for 'the'
'Hybrid CMM'	1.41	0.22
Phonetic CMM	<0.01	0.01
Hamming Distance	<0.01	0.01
Shifting n-gram	1.06	0.03
Triple mapping	<0.01	0.03
Agrep	0.01	0.03
Levenshtein Edit Distance	1.93	0.22

Table 4.4: Table listing the best match retrieval times for the systems evaluated. Our 'Hybrid CMM' approach combines the results of the phonetic match, n-gram match and Hamming Distance match using our scoring equation to identify the best matching lexicon words.

input, i.e., the exact match. From table 4.3, our 'Hybrid CMM' approach, the triple mapping and Agrep are all $\Theta(1)$ as the retrieval is independent of the length of the input, only Levenshtein Edit Distance is $O(inputLength)$. For our hybrid, retrieval time depends only on the number of matches as these must be retrieved from the word array, so more matches will entail slower retrieval. Our hybrid matches against the 242,819 characters of the UNIX 'spell' lexicon in 0.04 seconds so extrapolating would be able to process approximately 6,070,475 characters per second.

Best Match

For best match (see table 4.4), our 'Hybrid CMM' approach is dependent on the shifting n-gram which is $O(n)$. Our combined 'Hybrid CMM' approach took 1.41 seconds to retrieve the best match for 'floccinaucinihilipilification' compared with a time of 1.06 seconds for the shifting n-gram alone so the shifting n-gram occupies 75% of the total retrieval time for our 'Hybrid CMM'. For a 3-letter word input, our 'Hybrid CMM' approach took 0.22 seconds for 242,819 characters so could process 1,103,723 characters per second. For the 29-letter word the retrieval took 1.41 seconds so we can process 172,212 characters per

second. We could speed this retrieval by having three CMMs, one with 1-grams stored, one 2-grams and the third 3-grams. This would be faster, equivalent to the n-gram triple mapping, i.e., <0.01 seconds for the 29-character word but the memory storage would be approximate 6 times higher. Three times for the 3-gram CMM (as seen in the memory use evaluation in section 4.7.1), twice for the 2-gram and equivalent for the 1-gram. We feel the slightly slower speed is preferable to the higher memory requirement of the alternative representation. Most searches are likely to be approximate 7 or 8 characters. N.B. for the Hamming Distance and triple mapping the search for 'the' took longer than the 29-letter word. This is because there are many possible matches for 'the' which must be retrieved from the word array using the bits set in the thresholded output vector as indices into the word array but there is only one word to be retrieved for the longer word. The retrieval speed of the bit indexing mechanism is proportional to the number of bits set and thus words retrieved.

4.7.4 Quality of Retrieval

We evaluate our 'Hybrid CMM' approach against the benchmark spell checkers described in this chapter for spelling recall. We extracted 583 spelling errors from the Aspell [4] and Damerau [29] word lists and incorporated 17 additional spelling errors from the MS Word 2000 auto-correction list to give 600 mis-spelt words in total[5]. The results are given in table 4.5. We counted the number of times each approach suggested the correct spelling among the top 10 matches and also the number of times the correct spelling was placed first. N.B. We counted strictly so even if a word was tied for first place but was listed third alphabetically then we counted this as third. However, if a word was tied for tenth place (where the algorithm produced a score) but was listed lower, we counted this as top 10. We include the recall scores for MS Word 97, 2000 and Aspell spell-checkers for a benchmark comparison. We used the standard supplied dictionaries for both MS Word[6] and 'Aspell' [4]. For all other evaluated methodologies we used the standard UNIX dictionary augmented with the correct spelling for each of our phonetic misspellings giving 29,178 words in total[7]. The final column of table 4.5 gives the average number of words returned per query for the four approaches using the UNIX dictionary. For all queries when the approach returned the correct word in the top 10, we counted the position of the correct word. We then totalled the positions for all correct queries and divided the total by the number of correctly found queries to give

[5] We implemented all aspects of our spell checker before extracting the mis-spelt words to ensure no biasing of the results.

[6] We also checked that the correct spelling of each of the words not correctly matched was present in the Word dictionary and the 'Aspell' dictionary before counting the correct matches.

[7] All spell checkers included some of the misspellings as variants of the correct spelling, for example 'miniscule' was stored as a variant of 'minuscule', 'imbed' as a variant of 'embed' in MS Word. We counted these as 'PRESENT'.

Method	Found (Top 10)	First Place	Present	Not Found	% Recall	Precision
'Hybrid CMM'	558	368	6	36	93.9	2.13
Aspell	558	429	6	36	93.9	
Word 2k	510	432	17	73	87.5	
Word 97	504	415	15	81	86.1	
Edit Dist.	510	367	6	84	85.9	1.61
Agrep	481	303	6	113	80.1	2.57
Triple Mapping	449	115	6	149	75.6	2.51

Table 4.5: The table indicates the recall accuracy of the methodologies evaluated. Column 1 gives the number of correct matches within the top 10 matches returned for each word, column 2 shows the number of correct matches in first place, column 3 gives the number of spelling variants already present in the lexicon, column 4 gives the number of words not correctly found in the top 10. Column 5 provides a recall accuracy percentage (the number of top 10 matches / (600 - number of words present)) and column 6 lists the average number of words returned per query.

an average number of words returned.

Recall and Precision

If we consider the final column of table 4.5, the recall percentage (the number of top 10 matches / (600 - number of words present)), we can see that our hybrid implementation has the joint highest recall with Aspell. Our stated aim in the Introduction was high recall. We have achieved the joint highest recall of the methodologies evaluated, even higher than the MS Word benchmark algorithms. The MS Word and Aspell spell checkers are optimised for first place spelling and achieve more first place results than our hybrid. Both MS Word spell checkers and the Aspell system were using their respective standard dictionaries. This makes comparison of the first place spelling more contentious. However, assuming the comparison is valid, our approach has higher overall recall accuracy than both MS Word systems. Aspell uses a version of the UNIX 'spell' dictionary similar to our dictionary used in this evaluation. Aspell is a rule-based system based on a technique that is similar in implementation to Phonix [39]. The approach would not integrate neatly with our MinerTaur system architecture. Our spelling CMM produces superimposed output vectors that may be input straight into the word-document matrix described in chapter 5 in a single step. If we used Aspell as our spell checker, we would also need to compose the vectors for the matched words after spell checking, inevitably slowing retrieval. The user will see the correct spelling in the top ten an equal number of times for both Aspell and our system so we feel the ability to integrate with the system architecture offsets our hybrid approach's slightly lower first place retrieval.

Our hybrid spell checker retrieves 2.13 words on average, the correct word plus 1.13 incorrect words. Our precision figure is slightly below the Edit Distance figure but above both the Agrep and Triple Mapping figures. Our recall figure is superior to the Edit Distance value and we feel that recall is more important than precision. It is more important for the user to see the correct word possibly in third place rather than to receive a minimal list of words, for example a list of just 2 words with the correct word omitted because precision was the focus.

4.8 CMM Size Evaluation

For our size evaluation, we use the 29,187 and 9,485 word datasets from the evaluation of the spell checker in this chapter and the word-document index in chapter 5 respectively. We also use a large lexicon obtained by combining two word lists to give 1,423,237 words: one downloaded from [32] and the other downloaded from [76]. Both word lists are intended for password cracking programs which exploit a vast word base to compare and decipher user passwords. We sorted the large 1,423,237 word lexicon alphabetically and prepared the smaller datasets by extracting the first 1 million, 750,000, 500,000, 250,000 and 100,000 words from the large alphabetically sorted lexicon. We trained the words from the various-sized lexicons into the spelling CMM and phonetic CMM as per the methodology described in this chapter using orthogonal output vectors in all cases. Hence the output vector length is equivalent to the size of the lexicon to enable orthogonal vectors that uniquely identify each word. We calculated the memory usage for the spelling CMM and the phonetic CMM for each lexicon size. We noted the time for three matches for each lexicon size using our hybrid spelling methodology. We exactly matched a small word 'and', we exactly matched a long everyday word 'applications' and we performed a best match search using a long uncommon word 'floccinaucinihilipilification' ensuring that the correct spelling was not present in any lexicon so we could just retrieve a set of best matching spellings. We noted in this chapter that the best match retrieval was slower than exact match due to our use of the shifting n-gram so we can evaluate the worst-case scenario of a very long word best match retrieval against large lexicon sizes.

4.8.1 CMM Size

Table 4.6 details the memory usage statistics and figure 4.14 provides a graph of the memory usage for the spelling and phonetic CMMs with each of the lexicon sizes. The memory usage grows linearly with the lexicon size due to the use of orthogonal vectors - the output vector size increases linearly with the lexicon size. The spelling CMM memory grows more rapidly as the input size is larger (32 x 30 bit letter chunks see section 4.6.2 for the details) compared to the pho-

Lexicon size words	Spelling CMM memory usage	Phonetic CMM memory usage
1,423,237	147,313,072	23,134,392
1,000,000	96,725,812	14,761,992
750,000	68,814,124	10,510,632
500,000	42,034,544	6,266,048
250,000	18,071,472	2,895,124
100,000	6,069,600	1,054,432
29,187	1,480,144	460,464
9,485	971,256	214,112

Table 4.6: Table of the CMM sizes in bytes for the various lexicon sizes trained in to the CMM.

netic input size of (30 bits + 3 * 16 bits see section 4.6.3 for the details). Due to

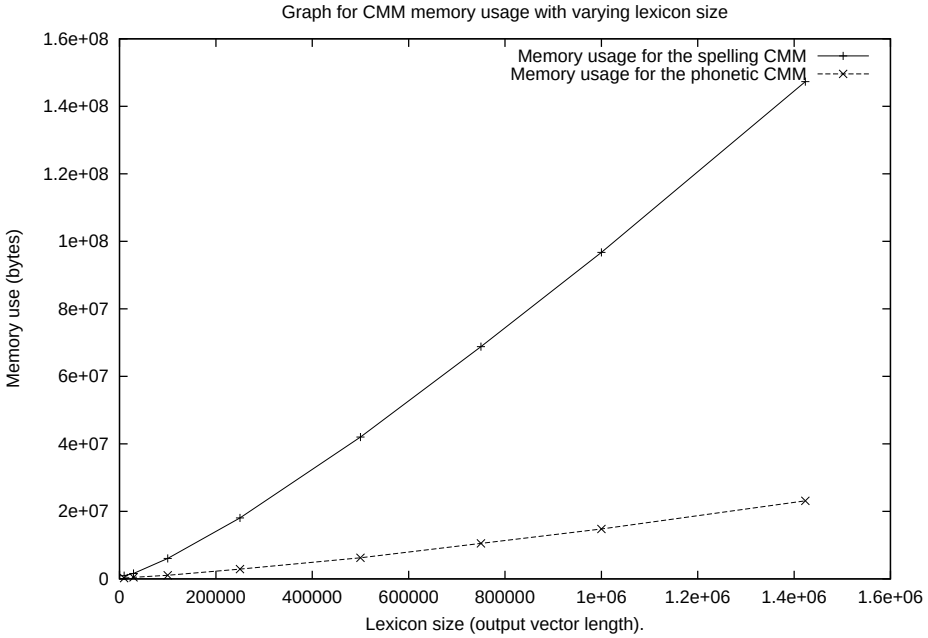

Figure 4.14: Graph of the CMM memory usage for each of the lexicon sizes.

the larger input size, the total number of bits in the spelling matrix grows more rapidly hence the memory usage for the spelling CMM increases more rapidly.

Figure 4.15 provides a graph of the retrieval times for the various lexicon sizes for the three test matches. The time for retrieval grows linearly with the lexicon size for each of the three test matches. The two exact match retrievals are extremely rapid even for the large lexicon size. The twelve-letter word 'applications' requires 4.2 seconds for an exact match with a trained lexicon size of

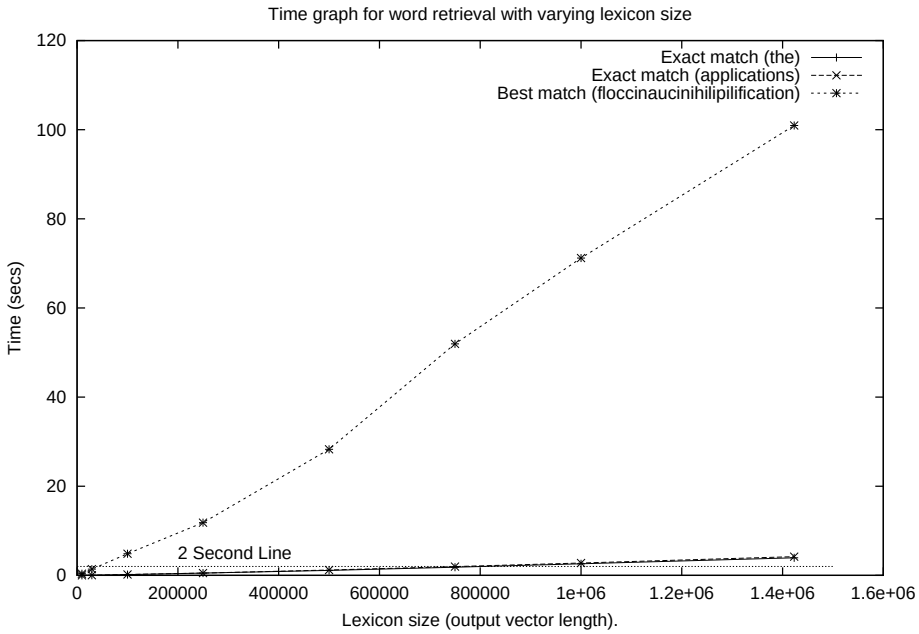

Figure 4.15: Graph of the retrieval times for the three benchmark words with each of the lexicon sizes. The best match search for a long word (floccinaucinihilipilification) not present in the stored lexicon is the worst case retrieval time and hence the retrieval time is much higher than an exact match for shorter words present in the lexicon which do not require the slower shifting n-gram retrieval.

1,423,237 words. The times for the three-letter word exact match are very similar to the twelve letter exact match. The times are just slightly less due to the lower word size as the input bit vector representing the three-letter spelling has less bits set (three compared to twelve) so less CMM rows need to be activated and checked during a match. However, from the graph the best match is much slower for the long twenty-nine-letter word. The slowness is due to our choice of the shifting n-gram rather than the alternative approach of non-positional n-grams as described in section 4.5. Our shifting n-gram approach has much higher recall than the non-positional n-gram approach, see table 4.5, but is slower particularly for longer words. However, we decided that the higher recall of the shifting n-gram mitigates the slower retrieval as most query words are likely to be seven letters or less and will have acceptable retrieval times whereas a twenty-nine-letter word is likely to be encountered very rarely in user queries.

From the graph in figure 4.15, we can see that the intersection of the 2-second line (i.e., the maximum feasible retrieval time for a spell checker) and the lexicon size axis for the 29-letter word best match retrieval is approximately 40,000 words. Hence, for a very long, uncommon word best match retrieval to take less than 2 seconds we can train up to 40,000 words into the hybrid system's two

lexicon CMMs using orthogonal output codes. From the memory usage graph, for a lexicon size of 40,000 words the spelling CMM memory usage is 2MB. This is not an unreasonable figure for a spell checker with a 40,000-word lexicon. However, 40,000 words is a small lexicon, password cracking dictionaries can store in excess of 1 million words as exemplified by the lexicon used in this evaluation.

4.9 Proposed Solutions for Large Datasets

We propose and analyse three solutions to the problem of storing larger lexicons in CMMs, using a one million-word lexicon as an example. We desire a uniform representation for each word vector in both the spelling and phonetic CMMs to allow our integrated hybrid spell checker to easily derive matches. If we permitted different representations we would require a complicated post match processing module to retrieve the matches from both the spelling and phonetic CMMs, to identify, to integrate and to score the matches. We also keep identical bit vectors so we only need to store the bit vector $\rightarrow$ word associations once rather than storing two separate associations; one association list for the spelling CMM and one for the phonetic CMM and hence this lowers the memory usage. A further manifestation of using identical word vectors is the ability to feed the output vectors from the phonetic and spelling CMMs directly into the word-document CMM to retrieve matching documents without recourse to computationally costly vector processing and translation. We note that the memory usage reduction is less important for the phonetic CMM as all words are translated to four-character codes which involves a much shorter bit vector length (augmented by the fact that there are only 16 different codes per character for the phonetic coding compared to the 30 possible characters for the spellings). These shorter input vectors produce lower memory usage for the phonetic CMM compared to the spelling CMM and accordingly a slower memory usage expansion than the spelling CMM where all characters in the word are translated into the bit vector.

4.9.1 Modularity

One possibility would be to alphabetically divide the lexicon into subsets of 40,000 words (first 40,000 words alphabetically, second 40,000 words etc) and train each 40,000-word block into a separate CMM. We could use a series of bits, one for each CMM, in the word vectors to denote in which CMM that word is stored. We could then search the CMMs in parallel and each CMM would have a maximum best match retrieval time of 2 seconds. This option would not generate any false positives as each separate CMM would generate only correct matches, the word vector output vectors would still be orthogonal. However, all CMMs would need to be memory resident to enable parallel searching for

spelling matches so this may be infeasible for some computer systems due to the total memory use. Furthermore, all CMMs would need to be memory-resident, as we cannot know in advance in which CMM the correct spelling is stored. If memory conservation is not of overriding importance then this option will provide the quickest retrieval due to the parallel search capability and the guarantee of no false positive matches. The other two options postulated here both generate false positives that need to be validated in a post-match verification step.

We could also subdivide the word-document CMM equivalently using the 40,000 word blocks. Each word-document CMM would have an input size of 40,000 and an output equivalent to the number of documents. This subdivision would speed the document retrieval phase as smaller CMMs are faster for matching than a single large CMM as tabulated in table 4.15. Only the word-document CMMs that are necessary for matching need be memory resident, i.e., those that contain the words to be matched and known in advance if we use bits in the word vectors to index the relevant CMM. The spelling CMMs would feed directly into the corresponding word-document matrices in a 1-1 mapping.

4.9.2 Compression

An alternative solution using a single spelling CMM but reducing its memory overhead is to use a compressed vector representation, reducing the width of the CMM by employing output vectors with multiple bits set to represent each word. There would be insufficient output vector attributes in the compressed CMM to permit an orthogonal representation with a unique attribute denoting each word. The word-to-document matrix could also use the same word vector representation for simplicity and ease of use, allowing the memory usage for the word-to-document matrix to be reduced accordingly as the input vector dimensionality will be reduced due to the word vector compression. Using multiple bits set in the vectors nullifies the storage advantage of compact bit vector CMMs so a binary bit vector CMM representation is preferred. For example, a two-bits set vector doubles the number of bits set for each output vector trained into the CMM during the learning phase. There will be a degree of overlap between the output bit vectors forced by the multiple bit set representation, only orthogonal vectors guarantee zero bit overlap, i.e., the same bit is never used more than once in different vectors. Thus, the total number of bits set for the spelling CMM will not be exactly double the number of bits set for the conventional orthogonal efficient CMM. However, the increased number of bits set on each row negates the storage efficiency of using compact CMMs or efficient CMMs that switch between compact where only the list of bits set is stored and binary bit representations where all bits are stored for the rows. The bit set list for the compact form becomes too long to introduce any storage gain and the

efficient CMM reverts to a binary representation for all rows.

However, using multiple-bits set output vectors introduces a storage gain for binary CMMs compared to orthogonal vectors as the bit vectors are shorter than the equivalent orthogonal bit vectors so there are less columns in the CMM reducing the size and thus the CMM storage requirement. The number of binary bits that need to be stored for each row is vastly reduced and thus the overall storage of the binary CMM is reduced compared to a binary CMM with orthogonal output vectors. For example to store 1,000 words in orthogonal bit vectors requires a 1,000 dimension bit vector with a unique attribute for each word. The equivalent two-bit set vector need only be 46-dimensional as there are $C_2^{46} = 1035$ different combinations of two bits set in a 46-dimensional vector. We note that 46 is the theoretical minimum dimensionality. The practical minimum dimensionality would in fact be higher to prevent too high a degree of overlap between the word vectors and thus prevent too many false positive matches being retrieved. Turner and Austin [105] investigated the performance of binary CMMs with non-sparse vector encoding and introduced probability distributions for modelling storage overlap due to the superpositioning of stored vectors and false positive matches due to the overlap of output vectors. The probability information may be used to determine the chance of false matches and an optimal representation size for vectors and CMMs may then be selected according to the system resources available and the maximum acceptable level of false positive matches.

There is a point where a binary CMM with multiple bits set becomes more efficient with respect to storage than the compact equivalent with orthogonal outputs as there are many more columns in the orthogonal CMM and this increases the storage too much. However, the switch point is very complex to calculate for any specific dataset, as we cannot know beforehand the degree of overlap of the bits in the associations to be stored in the multiple-bits set CMM. We do not have prior knowledge of the spelling-word associations to be stored and we can only generalise. An empirical evaluation would be necessary using the theoretical framework introduced in Turner and Austin [105] to pinpoint the most suitable configurations, comparing the configurations' respective storage overheads and selecting the most efficient configuration with respect to the specific dataset, the level of false positive matches deemed acceptable for the system and the system resources available. This optimal multiple-bit set representation could then be compared for storage overhead and retrieval time against an orthogonal representation.

Using multiple bits set inevitably introduces the problem of false positive matches due to bit clashes and introduces the necessity of a post-match back-check to

eliminate the false positives. Knuth [57] details superimposed coding and the intrinsic problem of false positives. For example if we use two-bit set vectors and we have the following three bit vectors each representing a unique word:

$$A: \ 10001000 \ \ B: \ 01001000 \ \ C: \ 01000100 \tag{4.16}$$

If the superimposed thresholded output bit vector from the spelling CMM is:

$$output: 11001100 \tag{4.17}$$

The output vector could denote $A+C$ or $A+B+C$. We would need to validate A, B and C against the input to establish the correct matches. This introduces an extra step into the matching procedure that slows the process. The three words, A, B and C may in fact be very similar and difficult to separate. We cannot possibly anticipate the false matching words or documents using this approach as they are totally dependent on the input to be matched, for example an input word may match 50% of the documents and the matching document vectors may overlap to form many false positives so separation of the superimposed output vectors may be difficult and slow due to many false positive matches. We can model the general framework for false matches as in Turner and Austin [105] but the specific false matches will vary greatly according to the inputs used. Turner and Austin assume a generalised number of query words and a generalised partial match level to generate the distribution model. In the approach described next, we are able to completely control the false positives produced with each correct match and can guarantee simple and efficient separation.

4.9.3 Binning

Our preferred option if memory conservation is of paramount importance is to employ '*binning*' as with Cavnar's system [18] with single bit set (orthogonal) vectors. The technique is also closely related to Aho and Ullman's Address Generation Hashing [1], described in section 5.1 and the superimposed coding techniques also detailed in section 5.1. Cavnar uses binning to represent multiple n-grams but we prefer to bin words. Binning maintains orthogonal vectors which preserves the storage advantage of efficient CMMs as there are few bits set in each CMM row and thus the bit set list to be stored is minimal. Binning with orthogonal output vectors should have a lower storage requirement than using multiple bits set output vectors as we can use efficient CMMs with their intrinsically lower storage requirement switching between the most storage efficient method as opposed to the binary representation required for multiple bits set that stores all bits in each row. Another advantage of binning is that we will get the correct match plus the remaining words from the bin, which we can control, rather than the uncontrolled false positives of multiple bit set false positives. We can use a post match back check to verify the correct word from the bin.

If we can have an orthogonal output vector size of 40,000 for a guaranteed 2-second maximum retrieval then we would need to have 25 words in each bin for a 1 million word lexicon. Therefore, 25 words map on to each bit (there are 25 words in each bin and each bin is represented by an orthogonal vector). The approach equates to superimposing the 25 separate CMMs of our first recommended option on top of each other. If we ensure that the 25 words in the bin are very different, with different lengths and different constituent characters we can simplify the back check. For each bit set in the thresholded output bit vector, all words in the bin are retrieved as matches to be verified and we can easily distinguish the correct match from the 25 words retrieved from each bin. As there are only 25 words to match against the input we can perform a character-based comparison relatively cheaply, for example, using Edit Distance as the words are very dissimilar the correct match from the bin should be easily retrieved. It is also a relatively simple procedure to form the bin sets. If we sort the input lexicon into alphabetical order then we can train each consecutive word in to each consecutive bin in a '*round-robin binning*' technique. This assumes that word length and the constituent characters of the words are randomly distributed through the alphabetically sorted list, which we feel is a reasonable assumption. It should also ensure dissimilarity in the bin sets. We assume that similar words tend to occur close to each other alphabetically so these will be spread across the bins before we cycle back to the first bin during round robin binning.

4.10 Conclusion

Spell checkers are somewhat dependent on the words in the lexicon. Some words have very few words spelt similarly, so even multiple mistakes will retrieve the correct word. Other words will have many similarly spelled words so one mistake may make correction difficult or impossible. Of the techniques evaluated, our hybrid approach had the joint highest recall rate at 93.9%. Humans averaged 74% for isolated word-spelling correction [63] (where no word context is included and the subject is just presented with a list of errors and must suggest a best guess correction for each word). Kukich [63] posits that ideal isolated word-error correctors should exceed 90% when multiple matches may be returned.

There is a trade-off when developing a phonetic spell checker between including adequate phonetic transformation rules to represent the grammar and maintaining an acceptable retrieval speed. To represent every possible transformation rule would produce an unfeasibly slow system yet sufficient rules need to be included to provide an acceptable quality of retrieval. We feel that our rule base is sufficient for the spell checker that we are developing as the phonetic module

is integrated with alternative spelling approaches (n-gram and Hamming Distance). We cannot account for exceptions without making the transformation rule base intractably large; we have mapped sc(e|i|y) → s which applies with the exception of sceptic pronounced *skeptic*. However, we feel our recall and coverage are generally high for the system developed and that the combination of the three methods should overcome any limitations of the individual methods.

Our exact match procedure is very rapid and independent of the length of the input spelling. The best match process is slower because the shifting triple is slower. However, we feel the superior quality of the shifting triple and the lower memory requirement as compared to the regular 3-gram offsets the lower speed. Our tripling approach exploits the spelling-word matrix used for the Hamming Distance match and also allows the value of n to be varied according to the length of the misspelt input. To implement variable length n-grams with the conventional n-gram approach would require three CMMS, one for the 1-gram, 2-gram and 3-gram increasing the memory requirement and the additional Hamming Distance CMM would also need to be stored.

For our system evaluation in this thesis detailed in chapter 6, the lexicon is approximately 40,000 words so we can maintain orthogonal vectors. For the spelling module we maintain a dual CMM representation, a spelling CMM and a complementary phonetic CMM. When MinerTaur develops to encompass larger datasets we will need to employ a more efficient representation with respect to retrieval time and memory overheads. From our discussion in this chapter, the solution chosen will depend on various hardware parameters of the underlying computer such as available memory. We will therefore choose the best option, either binning or multiple CMMs as hardware limits dictate.

A possible improvement to our spelling module would be to add a facility to handle the UNIX wildcard character '*' to allow more flexible word matching for the user. We have already implemented the UNIX '?' convention of allowing any single character to match. However, the wildcard is more complicated as the letters in the spelling do not align with those in the matches, for example, 'el*t' would match 'elephant' but only two characters are aligned and there is a large difference in the lengths. We could not use the Hamming Distance approach due to the non-alignment. We would also be limited with our shifting n-gram as this shifts along the length of the input spelling so 'el*t' would not match elephant if we restricted the length of the shifting. However, shifting along the maximum possible length (30 characters) is not really an option as the shifting would be slow and was not implemented with this facility in mind. We have acknowledged that it can be slow for long words but as most words in common usage are twelve characters or less, for example most words in this chapter, we

adopted the higher recall and lower memory overhead of the shifting n-gram as opposed to the higher speed of the non-positional n-gram. The phonetic spelling again aligns the phonetic code characters so this would not allow matching for wildcards. If we were to implement the wildcard in a CMM we would need the standard non-positional n-gram as this allows n-gram matching anywhere in the lexicon word up to the maximum length of 30 characters. This would require an extra CMM to store the n-gram binary vectors and lexicon word associations and thus introduce a higher storage overhead. A more feasible alternative would be to exploit the Unix 'grep' facility or possibly 'Agrep' as both allow wildcard matching, are fast and do not store the lexicon during processing so impose only a minimal storage overhead.

Chapter 5

AURA Word-Document Association Data Structure

Many computational implementations require algorithms that are storage efficient, may be rapidly trained with data and allow fast retrieval of selected data. IR requires the storage of massive sets of word to document associations. The data structure must allow the documents matching query terms to be retrieved using some form of indexing. This inevitably requires an algorithm that is efficient for storage, allows rapid training of the associations, fast retrieval of documents matching the query terms, allows new associations to be added without a recompilation of the entire data structure and additionally, permits partial matching (M of N) query terms where $M \leq N$ and N is the total number of query terms and M is the number of those terms that must match. Many methodologies have been posited for storing these associations, see for example [57]: including inverted file lists, hash tables, document vectors and superimposed coding techniques. In this chapter, we only compare 'Perfect' techniques, i.e. those that preserve all word to document associations as opposed to 'Imperfect' methodologies such as Latent Semantic Indexing (LSI) [30] which compresses a document vector matrix to reduce the storage overhead but factors out many word-document associations in the resultant compressed representation.

5.1 Introduction

A representation strategy used in many systems is document vectors. There are various adaptations of the underlying strategy but fundamentally, vectors that have an attribute for each word in the corpus represent the documents. The document vectors form a matrix representation of the corpus with one row per document vector and one column per word attribute. For binary document vectors, used in for example Koller & Sahami [61], the weights are Boolean so if

a word is present in a document the appropriate bit in the document vector is set to 1. The matrix may also be integer-based as used in Goldszmidt & Sahami [43] where w_{jk} represents the number of times $word_j$ is present in $document_k$. By activating the appropriate columns (words) the documents containing those words may be retrieved from the matrix. LSI decomposes a word-document matrix to produce a meta-level representation of the corpus with the aim of correlating terms and extracting document topics. LSI reduces the storage using a factor analysis algorithm called Singular Valued Decomposition (SVD) [30]. Although this serves to reduce storage it also discards information and compresses out the word-document associations we need for our evaluation.

There are alternative hashing strategies (as compared to the standard hash structure evaluated in this chapter). They are aimed at partial matching and minimise file accesses to speed retrieval by pre-selecting matching records or groups of records but tend to retrieve false positives (documents appear to match that should not) and are thus 'Imperfect'. The techniques are similar to the Binning technique we recommended in chapter 4 to accommodate large lexicons and large numbers of word-document associations. There may be insufficient dimensions for uniqueness in hashing so many words may hash to the same bits and thus false matches will be retrieved. The extra matches then have to be re-checked for correctness and the false matches eliminated thus slowing retrieval. They are described in [56] and include:

Address Generation Hashing. Each document is subdivided into n fields of words. A separate hash function is applied to each distinct word field to produce a hash key (binary address) for the document. Each word in the corpus activates pre-determined bits in the binary document address. The document corpus is subdivided into equal size subsections (buckets) by grouping all documents that hash to the same binary address. During access, a binary address is generated from the user query by hashing the fields. The binary address of the query is used to index the buckets and identify any buckets (groups of documents) relevant to the query. All documents within the relevant buckets are checked against the query in a finer-grained match to determine the best matching document. Any bits not set in the query are assumed to be multi-mode both 1 and 0 so $2^x \times R$ where x is the number of unknown bits and R is the number of items per bucket will be retrieved and checked. A large amount of items are retrieved where many are false positives that have to be back-checked for correctness thus slowing retrieval by a significant degree. There may be insufficient attributes for uniqueness so many words may hash to the same bits. Determining the optimal size for the fields is NP hard. (See also [1]).

Hashing with Descriptors A hash code is generated for each attribute of a

record to be stored. The hash codes are then concatenated to produce a vector representing the record. Records are subdivided into groups (pages) and the record vectors are bitwise ORed to produce a page descriptor, a fixed length vector determined by the records in the page. The pages do not need to be stored contiguously on physical storage. During a user query, the page descriptors are checked to verify if any records match the query. Only the pages matching the query vector are accessed. Ramamohanarao et al. [81] demonstrated that hashing with descriptors has a significant performance advantage compared to standard hashing such as the hashing scheme evaluated in this chapter. Though hashing with descriptors produces less false positives than address generation hashing it still requires a time consuming back-check of the documents retrieved to eliminate false matches. (See also [81]).

There are also superimposed coding (SIC) techniques (described in [56]) for hash table partial matching applications but again these are 'Imperfect' and tend to over-retrieve, for example:

One-Level Superimposed Coding One vector is generated for each document by converting the constituent words into unique vectors with k bits set where k is pre-specified. The vectors from all of the document's words are bitwise ORed to produce a single vector representing the document, the *signature*. Signature files have a lower storage requirement than comparative inverted file lists evaluated in this chapter. The document signatures are stored using a 'bit-slice' mechanism, where the first bits of all document vectors are stored contiguously, then the second bits of all document vectors and so on. This reduces the cost of retrieval as the query bits can be compared with the stored document vectors in bit order, iteratively eliminating unmatched objects. However, the retrieval speed gain introduced by bit-slicing is at the expense of incrementality. The storage cannot be augmented with new document vectors so the bit-sliced storage structure would need to be rebuilt and stored again. Additionally, there will be false positives retrieved as the approach uses multiple-bit set, word vectors and a back check of the retrieved documents is required making the process significantly slower and computationally expensive. (See also [85]).

Two-Level Superimposed Coding The document signatures are generated as with one level SIC. The set of all document signatures is then subdivided into fixed length partitions (blocks). A descriptor vector is constructed for each block by bitwise ORing the attribute vectors of the indexed words from all documents in the block. This block descriptor vector is larger than an individual document vector but much smaller than the sum of all document vectors in the block. A block descriptor vector thus has a

lower storage overhead than the individual document vectors in the block. The query vector is generated by bitwise ORing the attribute vectors for the query words. The query vector selects the appropriate blocks for any matching documents. The query vector is then applied to the retrieved blocks to select the appropriate documents. However, again a post match back check of the documents retrieved is necessitated due to false positive matching. (See also [90]).

We wish to avoid the information loss inherent in LSI and also the false positives of SIC with their intrinsic requirement for a post-match re-check to eliminate false matches. Therefore we concentrate on 'Perfect' lexical techniques. We analyse an **inverted file list** used in Agrep [115], [114] and a slightly more sophisticated version of which is used in both the Google search engine [10] and in Inquery [15]), against **hash tables** used by Higgins [44] - in this chapter we evaluate two hashing functions, against the **binary associative memory** of AURA described in [5] and chapter 3. We implement a novel binary matrix version of document indexing using AURA where the word-document associations are added incrementally and superimposed so new associations may be added as and when required. Incremental knowledge addition, the ability to incorporate new data in to the existing data structure without an entire recompilation, is a key requirement of an IR indexing technique as identified in the Introduction in chapter 1. In AURA training requires only a single-pass through the word-document association list and is incremental with new associations simply incorporated into the binary CMM.

In all cases we evaluate the algorithms in their standard form without any sophisticated improvements to provide a valid comparison of the approaches. We evaluate the algorithms for storage use; training speed[1], retrieval speed, and partial matching capabilities. Knuth [57] posits that hash tables are superior to inverted file lists with respect to speed but the inverted file list uses slightly less memory. Knuth also details superimposed coding techniques but focuses on approaches with multiple bits set as described above, which inevitably generate false positives. We implement *orthogonal* single-bit set vectors for individual words and documents that produce no false positives. AURA may be used in 'Imperfect' mode where multiple bits are set to represent words and documents, this reduces the vector length required to store all associations but generates false matches as described previously and in Knuth [57]. In this chapter, we focus on using AURA in 'Perfect' mode as the size of the dataset evaluated permits both orthogonal input and output vectors. Orthogonal vectors produce perfect retrieval; there are no false positives so we do not require a back-check

[1] training time and thus speed is implementation dependent. To minimise variance, we preserve as much similarity between the data structures as possible particularly during training - see section 5.2 (subsections 5.2.1, 5.2.2, 5.2.3 and 5.2.4) for details

procedure to validate the matches as the extra elimination step would increase the retrieval time.

For our evaluation we use the Reuters-21578 Newswire text corpus [82] as the dataset is a standard Information Retrieval benchmark. It is also large, consisting of 21,578 documents with on average approximately 100 words per document. This allows the extraction of a large set of word-to-document associations for a thorough evaluation of the algorithms. We discarded any documents that contained little text and any documents that were repetitions of previous documents. After pruning 18,249 documents remained that were further tidied to leave lower-case alphabetical and six standard punctuation characters only. All alphabetical characters were changed to lower case to ensure matching. We felt numbers and the other punctuation characters added little value to the dataset and are unlikely to be stored in an IR system. We left 6 punctuation characters as control (verification) values. We extracted all 9,491 remaining words including the six punctuation symbols " : ; & ( ) from the 18,249 documents and derived 1,230,893 word-to-document associations. We created a file containing the document IDs, which are represented by integers indexed from 0 to 18,248, and a list of each word that occurs in that document. The word-document association is binary, we only store whether a specific word occurs in a specific document and we do not store a word frequency figure.

The fractions of documents that contain each individual word are shown in the graph (see figure 5.1). The minimal fraction is 0.000055 (1 document) and the maximal fraction is 0.8138 (14,850 documents). There is an even distribution of common and uncommon words when the words are sorted alphabetically. Hence, when we extract the first 100 words and every 50th word from the alphabetical list we should extract a representative sample of common and uncommon words correlated with the overall distribution.

We analyse the three data structures for multiple single query term retrievals and also for partial match retrievals where documents matching N of M query terms are retrieved. We assume that all words searched are present; we do not consider error cases in this chapter, e.g., words not present or spelling errors.

5.2 Data Structures

5.2.1 Inverted File List (IFL)

For the inverted file list compared in this chapter, we use an array of words, sorted alphabetically and linked to an array of lists. This data structure minimises storage and provides flexibility. The array of words provides an index into the array of lists (see figure 5.2 and section A.1.2 for the C++ implemen-

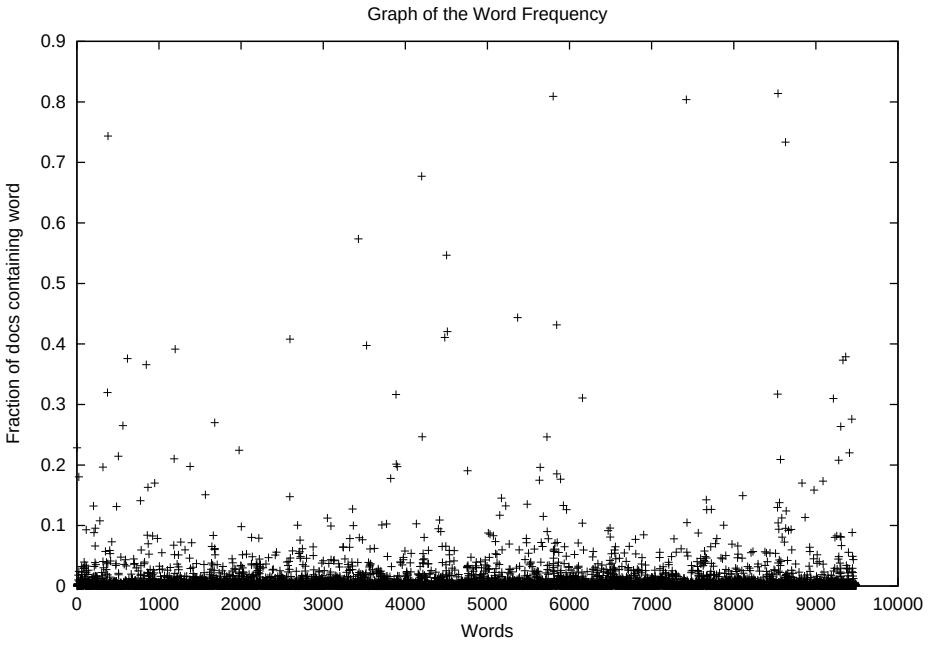

Figure 5.1: Diagram showing the fraction of all documents that contain each word. Each word has an integer ID indexed from 0 to 9,490 according to its position in the alphabetically sorted list of all words.

tation). A word is passed to an indexing function (binary search through the alphabetically sorted array of words) that returns the position of the word in the array. The document list stored at array position X represents the documents associated with the word at position X in the alphabetic word array. The lists are only as long as the number of documents associated with the words to minimise storage but yet can easily be extended to incorporate new associations. The document ID is appended to the head of the list for speed, appending at the tail requires a traversal of the entire list and thus slows training. The approach requires minimal storage for the word array (the array need only be as long as the number of words stored as compared to the hash table below, for example, that requires the word storage array to be only 50% full) but additions of new words would require the array to be reorganised alphabetically and the pointers to the array of lists updated accordingly. The retrieval of the index into the array of lists, binary search, is $O(\log\ n)$ so the design of the data structure is a trade-off between minimising storage but providing slower access, in comparison the hash table below is $O(1)$ for access.

The inverted file list achieves partial matching through the addition of a supplementary data structure - an array of documents and counters. In this chapter, the documents were identified by integer IDs so we exploit this to provide an index into an array of counters that count the number of words matched by

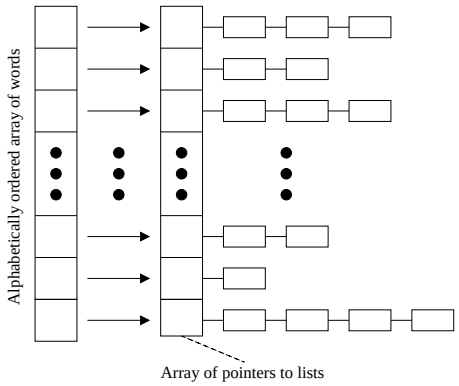

Figure 5.2: Diagram showing the inverted file list data structure. We implemented two separate, linked data structures to preserve similarity between this data structure and the hash table structure. The speed of training and retrieval are dependent on implementation. By maintaining similarity, we attempt to eliminate as many differences as possible to permit comparisons between the different data structures.

each document. The counter stored at position ID is the number of times that document ID has been retrieved. During retrieval, each time a document is retrieved from the document list of a query word, the corresponding document counter is incremented. A single pass through the array, once retrieval is complete, will retrieve the documents that have matched the required number of words (N of M).

For the inverted file list, the training time was the time to input the words into the alphabetically sorted word array and the time to add the word-document associations, i.e. to add the document IDs to each respective word's linked list. The memory used equals the size of the word array plus the size of the array of lists. For the partial match an additional data structure, an array of document counters, was incorporated so the memory usage for partial matching is also given.

5.2.2 Hash Table

The hash table employed in this chapter is used to maximise the retrieval speed of the documents associated with a particular word. An array of words is again linked to an array of pointers to linked lists (see figure 5.3 and section A.1.3 for the C++ implementation) to minimise storage and maintain flexibility while a hash function generates the indices. A hash function determines the location at which a given item is stored based on some attribute of the item to be stored. This computed location is called the item's hash value. To insert a new item into the hash array, it is necessary to compute the hash value of the item. If

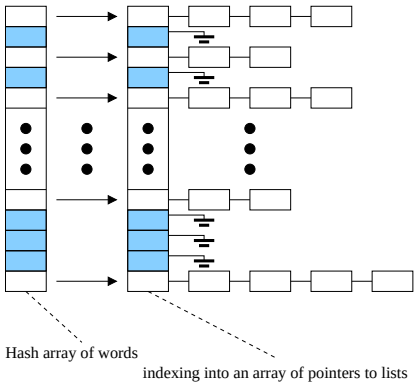

Figure 5.3: Diagram showing the hash table data structure.

this location is empty, the new item can be inserted. However, if this location is already occupied (a collision), an alternative strategy must be used. The complexity of an ideal hash operation with no collisions is constant time $\Theta(1)$. The worst-case upper bound for the hash operation complexity is $O(n)$ as the table fills. The price for such a speed up is memory usage. The hash table used in this chapter degrades as the table fills and thus must be kept less than 50% occupied otherwise there are too many collisions degrading both retrieval and insertion speeds through the additional calculations that must be employed. There are two methods for handling collisions known as collision rules; they are *Open Addressing* (also known as *Chaining*) and *Closed Hashing*.

- In *Open addressing*, each hash location is a pointer to a linked list. All items that hash to a particular location are added to the location's linked list. The approach is not feasible for the data representation required in this chapter. We must have a unique hash location for each word to indicate which documents belong to which words. We would need to store both the words and the documents as two attribute records in the list to identify the associations. This would waste storage space and slow retrieval as the algorithm searched through the entire list of all documents in the chain where many may be associated with the other words in the word chain.

- In this chapter we use *closed hashing* where no more than one item is stored at each hash location. The collision rule generates a succession of locations until an empty array item is discovered.

It is desirable, for efficiency reasons, to have as few collisions as possible. To achieve this, we employ an initial hash function that is not pre-disposed to favour any particular location or set of locations. We want to spread the hash values across the array as evenly as possible to minimise collisions and prevent clustering [24]. This is most easily achieved if the hash function depends on all parts of the item, computing a large integer value from the item, dividing

this integer by the table size, and using the remainder as the hash value. We evaluated three common hash functions for strings given in equations 5.1 and 5.2 and the C code below, where $a[n]$ is the character string of length n and $a[0]$ denotes the first character's ASCII value, $a[1]$ the second and so on:

$$a[0] * 1 + a[1] * 2 + a[2] * 3 + ... + a[n-1] * n, \tag{5.1}$$

$$a[0] * 2^0 + a[1] * 2^1 + a[2] * 2^2 + ... + a[n] * 2^n \tag{5.2}$$

and Horner's rule given below in C code (adapted from Sedgewick [98]).

```
1 unsigned hash:: horner(char * word)
2    {
3        for (sum=0; *word; word++){
4            sum = (sum*131) + *word;
5        }
6        return (sum \% hashTableSize);
7    }
```

Horner's Rule produced the least number of collisions of the three functions during insertion of 9,491 words into the hash table. We then empirically evaluated Horner's rule and found a factor value of 131 (line 4) minimised the collisions during insertion. We selected a hash table size that was a prime number to ensure that the modulo function (%) is not followed by an integer with small divisors. If the integer has small factors then the hash algorithm will be pre-disposed to clustering [3]. We selected the prime number 20,023 for the table size, as this was the first prime number greater than double the length of the word list (to ensure the hash table is less than 50% full). It also has $(hashTableSize - 2 = 20,021)$ as prime. This ensures the hash increment (described below) does also not have any small divisors. If $hashTableSize$ and $hashTableSize - 2$ are not relative primes at the least, they will have a greatest common divisor (gcd) and only $\frac{1}{gcd}$ of the table will be probed (see [24]).

There are various methods that may be used to generate additional locations if a collision occurs at the hash value location: *Linear probing*, *Quadratic probing* and *Double hashing*.

- *Linear probing*, the next available space is allocated to the item whose hash value caused a collision. The approach tends to cause clustering.
- *Quadratic probing*, this method iteratively generates locations exponential distances apart until a free location is discovered; the offset quadratically depends on the probe number. Again this method tends to cause secondary clustering and bouncing. Quadratic probing is not guaranteed to find an empty cell even if one exists (see [24]).

- *Double hashing* is used for our investigations. Double hashing computes a displacement directly from the key using another function of the key and adds this displacement to the table position until a vacant position is found. The C code is given below where *sum* is the value returned from the Horner function listed above.

```
unsigned hash:: hashIncr(void)
    {
        return 1 + sum \% (hashTableSize-2);
    }
```

  For double hashing, $\Omega(n^2)$ displacements are produced rather than $\Omega(n)$ for linear or quadratic probing improving the algorithm's performance (more possible free locations are examined), see [24].

Again for partial match we employ the additional document data structure used for the inverted file list. The training time was the time to input the words into the word hash array and then to add the word-document associations by prefixing the document IDs to their respective lists. The memory used is the size of the word array (50% full) plus the size of the array of lists (50% full). For the partial match an additional data structure was incorporated so the memory usage for partial matching is again given.

5.2.3 Two Stage Hashing - Hash Table Compact

The previous hash table in section 5.2.2 used a word array and a list array that were both only 50% full thus wasting storage. For comparison, we utilise the data structures from the previous hash table algorithm. However, we compact the array of lists so that all array locations are used (see figure 5.4 and appendix A.1.5 for the C++ implementation). The initial hash array of words stores both

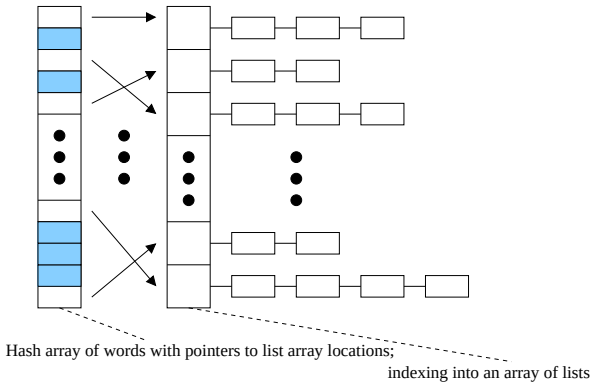

Figure 5.4: Diagram showing the hash table data structure. The integer location of the word's list is stored with the word in the first data structure and may then be used to access the contents of the list.

the word and an integer identifying the location of the word's document ID list in the list array. When the word string is passed to the hash function, the string is hashed and an integer retrieved from the hash location giving the location of the document list in the list array. The documents may then be read from that location. The complexity of the algorithm is identical to the previous $O(1)$ degrading to $O(n)$ for inserting a word into the word array or retrieving the location of the word or its associated document list. We compare the two data structures for memory use, training speed and retrieval speed. We evaluate the additional overhead of storing an integer to point to the word's document list versus storing only the word but requiring gaps in the array of lists (empty array elements) for the previous hash table.

Again for partial match we employ the additional document data structure used for the inverted file list. The training time was the time to input the words into the word hash array and then to insert the document IDs of the word-document associations in to their respective lists. The memory used is the size of the word array (50% full) plus the size of the array of lists (100% full). For the partial match an additional data structure was incorporated so the memory usage for partial matching is again given.

5.2.4 AURA

The words form the inputs and the documents form the outputs of the binary matrix. Each word in the word list is represented by a unique m-dimensional orthogonal (single bit set) binary vector where m equals the number of words in the list. The first word's vector has the first bit set and the last word in the word list has the last vector bit set (see equation 5.3). We can then use the word list as an index, setting the bit in the binary vector that correlates to the word's position in the word list. The documents are represented likewise with n-dimensional single bit set binary vectors, the first document's vector has the first bit set through to the last document's vector has the last bit set where n equals the number of documents (see equation 5.3). We can use the document ID (integer) as an index to identify which bit to set in the document binary vector during training and also to identify the matched documents during retrieval.

$$bitVector^{p} = p^{th}\text{bit set; } \forall p \text{ where p = position\{words\} or p = position\{documents\}} \tag{5.3}$$

We require a 'Perfect' recall technique as stated in the 'Introduction' 5.1. Orthogonal vectors ensure that unique vectors identify all words and all documents. Orthogonal vectors ensure that the CMM does not return any false positives, i.e., any erroneous matches. If more than one bit is set in the input vectors or output vectors then we can get bit clashes and false positives will be returned from the system as with the alternative hashing and superimposed coding approaches listed in the 'Introduction' and [57].

For the word-to-document matrix we use Compact Bit Vectors (CBVs, see chapter 3) as only one bit is set in each word or document binary vector. Therefore, this representation is the most storage efficient for the word or document orthogonal binary vectors as only one position index is stored in the list. We use efficient CMMs with a switch value of 300, if more than 300 bits are set the CMM row is stored as binary otherwise it will be stored as compact. We empirically derived the optimal switch value where the CMM memory usage was minimal. A comparison of the memory usage of the CMM using binary CMMs, compact CMMs and efficient CMMs (see section 3.1.2) is provided in the results section (see section 5.4.1).

Training the Network

The binary patterns representing the tokens are input to the network and the binary patterns for the documents form the outputs for the CMM in a single epoch supervised training process. The training process is $\Omega(n)$ as there is one association in the CMM per word-document pair producing a total of n associations.

Recalling from the Network

For recall only the required word vector is input to the network. If more than one word needs to be match then the required word vectors are superimposed and the superimposed vector is input to the network. The columns are summed

$$output_j = \sum^{\text{all } i} input_i \wedge w_{ji} \tag{5.4}$$

and the output of the network thresholded to produce a binary output vector (see figure 5.5). The thresholded output vector represents the superimposition

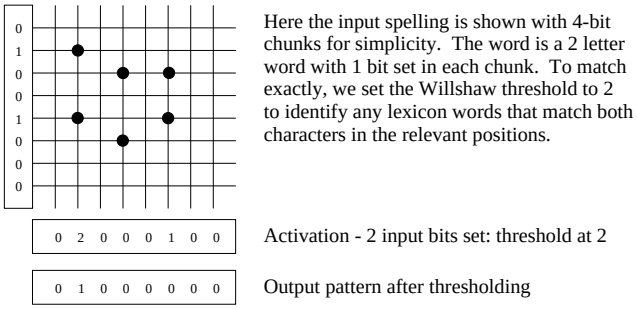

Figure 5.5: Diagram showing system recall. The input pattern has 1 bit set so the CMM is thresholded at 1.

of the matching documents trained into the network. We use the Willshaw threshold (see [5]) set to the number of matches required for example if ten

words are input but we only need to match seven then we threshold at 7, (see figure 5.5). The Willshaw threshold sets to 1 all the values in the output vector greater than or equal to a predefined threshold value (seven in the example described above) and sets the remainder to 0. We can identify the matched documents by retrieving a list of the bits set in the thresholded output vector. Each bit set is the ID of a document (represented by integers). The time to retrieve the matching documents is thus proportional to the number of bits set in the output vector Θ(bits set), there will be one matching document per bit set for orthogonal (single bit set) vectors.

The training time was the time to input the words into the word array, read in the list of documents and to store the word-document associations in the memory matrix. The memory used equals the size of the word array plus the size of the document array plus the size of the memory matrix.

5.3 Analyses

All analyses were performed on a SGI Origin 2000 with the following specifications (taken from the IRIX *hinv* command):

- 32 X 180 MHZ IP27 Processors
  As in chapter 4, the 32 processors allow us to run our evaluations in parallel, with one identical evaluation of each data structure running on a separate processor simultaneously (4 processors in total) to provide an identical system state for each data structure (IFL, hash, hash compact, CMM) during each separate evaluation and thus minimise timing variance.
- CPU: MIPS R10000 Processor Chip Revision: 2.6
- FPU: MIPS R10010 Floating Point Chip Revision: 0.0
- Main memory size: 8,192 MBytes
- Instruction cache size: 32 KBytes
- Data cache size: 32 KBytes

All data structures were compiled with the CC compiler using -Ofast (fast binaries) and -64 (64-bit) - the AURA library requires 64-bit compilation so we compiled the other data structures likewise for consistency. If the code had been compiled as 32-bit for the inverted file list and hash tables the memory usage would be less, the actual values are given in section 5.5.1 for comparison. The algorithms were run with the command *runon x <algorithm>* and one identical evaluation for each data structure was run in parallel.

The following analyses were performed on the data structures:

1. The memory usage for each algorithm was calculated using the C/C++ sizeof() utility.

2. The training time for each algorithm was calculated using the C/C++ clock() function. We described the components of the training time in the description of each data structure.

3. Serial Match - the words are matched one at a time with the matching documents retrieved after each word match. A canonical list of all words occurring in 10 or more different documents was generated and sorted alphabetically. For all data structures an identical output (the word to be matched and the list of matching documents) was generated. Two serial match investigations were run on each data structure.

   (a) For each of the first 100 words in turn taken from the alphabetically sorted canonical list, retrieve the matching documents (all documents that contain that word). This provides an iterative analysis:
   $\text{word}_1 \rightarrow \{\text{matching documents}\}_1$ then
   $\text{word}_2 \rightarrow \{\text{matching documents}\}_2$ then
   ...
   $\text{word}_{100} \rightarrow \{\text{matching documents}\}_{100}$.
   The graph is given in figure 5.6. As each word is read, the matching documents are retrieved and written to an output file, then the next word is retrieved and matched etc. For all data structures the output to the file is identical. We read all words in to an array of words then read the first 100 words from the array using a 'for' loop. Obviously we could have just read the first 100 words from the file but to maintain consistency with the next evaluation we employed the array here (we need to retrieve every 50th word from the alphabetical list of all words and hence need to read in the entire list for the next evaluation). This analysis aims to evaluate the access speed of each data structure and should favour the hash table as the first 100 words are less likely to have suffered collisions. The hash table was virtually empty as these words were being inserted therefore the retrieval time will be approximately $\Omega(1)$. The entire 100-word match was performed ten times consecutively and the overall average time calculated as averaging should eliminate any timing fluctuations.

   (b) Retrieve the documents that match every 50th word in the canonical word list in turn (iterative).
   $\text{word}_{50} \rightarrow \{\text{matching documents}\}_{50}$ then
   $\text{word}_{100} \rightarrow \{\text{matching documents}\}_{100}$ then
   ...
   $\text{word}_{9450} \rightarrow \{\text{matching documents}\}_{9450}$.
   This matches the documents against 189 words (9,491 words /50).

We select every 50th word as the graph of the number of words in each document is similar to the graph for the 'first 100' word retrieval (see figures 5.6 and 5.7). This second analysis aims to negate any bias present in the first analysis, for example, using the first 100 words may favour the hashing algorithms as the first words to be inserted in to the hash data structures will be less likely to collide, therefore less hash table elements have to be probed to match these words, thus speeding retrieval. Again all words are read from the word file into an array and every 50th word is retrieved from the array in a 'for' loop. The matching documents for each word are written to an output file in turn. This analysis will again evaluate the retrieval speeds and should not favour any data structure as the words are evenly spread through the alphabetically sorted list of words. The match was performed ten times consecutively and the time averaged for each of the ten retrievals.

4. Partial Match. For all data structures the output generated is identical to maintain consistency. The following three partial match evaluations were performed on each method.

   (a) Retrieve the documents that match at least 0, at least 1, ... at least 9 of the first 100 words taken from the alphabetically sorted canonical word list. Figure 5.6 is a graph of the frequency distribution of the first 100 words occurring in each document. There is only 1 document matching at least 9 words so we cease partial matching at this value.

   (b) Retrieve the documents that match at least 0, at least 1, ... at least 9 of every 50th word taken from the alphabetically sorted canonical list of words. Figure 5.7 is a graph of the frequency distribution of the number of words occurring in each document taking every 50th word of the alphabetical list of all words. Again there are only a few documents matching at least 9 words so we maintain the same evaluation values as with the 'first 100' evaluation.

   (c) Retrieve the documents that match at least 10, at least 11, ... at least 37 of the most frequently occurring words - the words that occur in at least 20% of all documents. There are up to 38 words in the 20% word set to match and from the graph (see figure 5.8) the documents match between 0 and 38 words from this set.

5.4 Results

5.4.1 Memory Usage

The memory usage for each of the constituent sub-structures for each data structure is given below (in bytes).

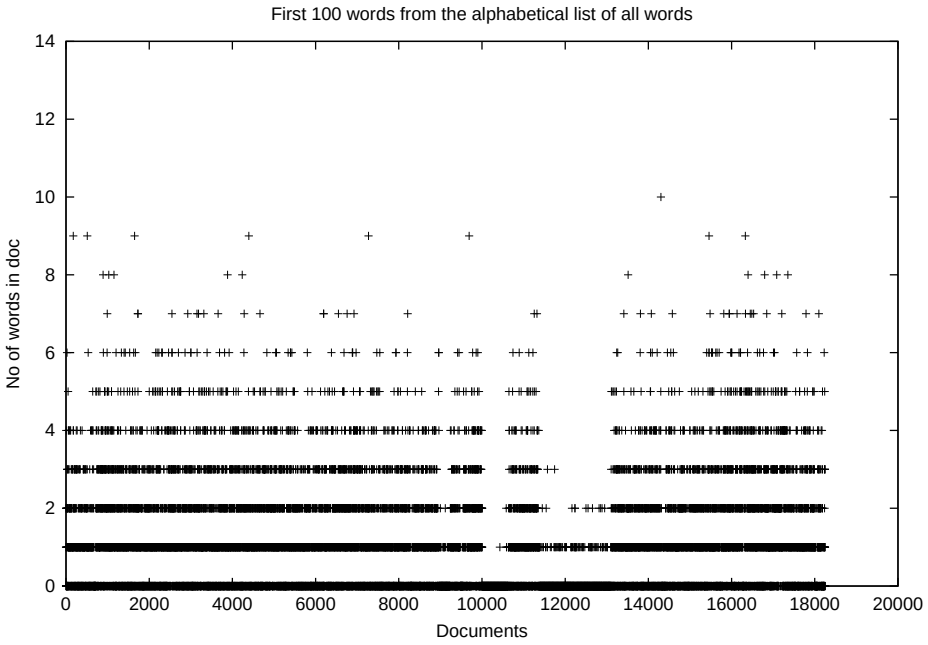

Figure 5.6: Diagram showing the total number of the first 100 words from the canonical word list present in each document. A word that occurs multiple times in a particular document is only counted once for that document. Each document has an integer ID.

1. Inverted File List Length: 9,491.
   List memory use for all document lists is 9,847,144.
   Array (array of lists) memory use is 75,928.
   WordTable (array of words) memory use is 475,000.
   Memory use for array of documents (counter for partial match) 73,000.
   Memory use for array of words (to get every 50th word etc.) 474,550.
   Total memory use is 10,945,622.

2. Hash Table Length: 20,023.
   List memory use for all document lists is 9,847,144.
   Array (array of lists with empty locations) memory use is 160,184.
   WordTable (hash table of words length 20,023) memory use is 1,001,150.
   Memory use for array of documents (counter for partial match) 73,000.
   Memory use for array of words (to get every 50th word etc.) 475,000.
   Total Memory use is 11,556,478.

3. Hash Table Compact Length: 9,492.
   List memory use is 9,847,144. All three data structures have identical memory usage for the lists.

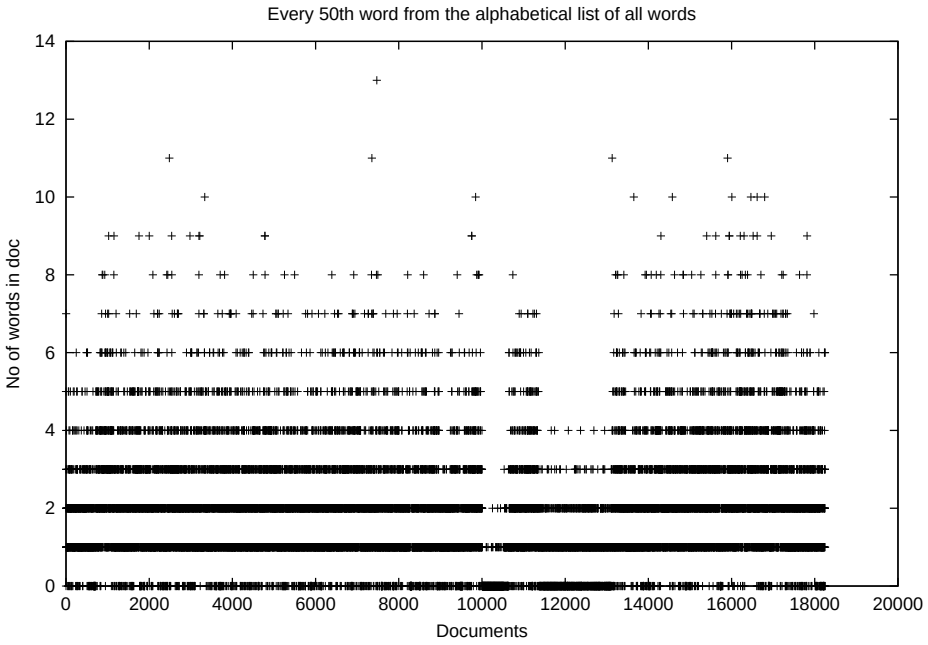

Figure 5.7: Diagram showing the number of words counted in each document when taking every 50th word from the list of all words. Again, any word that occurs multiple times in a particular document is only counted once for that document. Each document has an integer ID.

Array (array of lists with length 9,492) memory use is 75,936.
WordTable (hash table of words with length 20,023) memory use is 1,121,288.
The memory use is higher than the hash table as this structure stores the integers to indicate the word's list.
Memory use for array of documents for partial match 73,000.
Memory use for array of words to get every 50th word etc. 475,000.
Total Memory use is 11,002,632.

4. CMM

   - Binary - storing the CMM as a binary representation (see section 3.1.2).
     CMM saturation is 0.007.
     CMM uses 44,283,360 bytes. The memory usage is very high in comparison to the other data structures.

   - Compact -storing the CMM as a compact representation (see section 3.1.2).
     CMM saturation is 0.711.
     CMM uses 21,378,128 bytes. The memory usage is high in com-

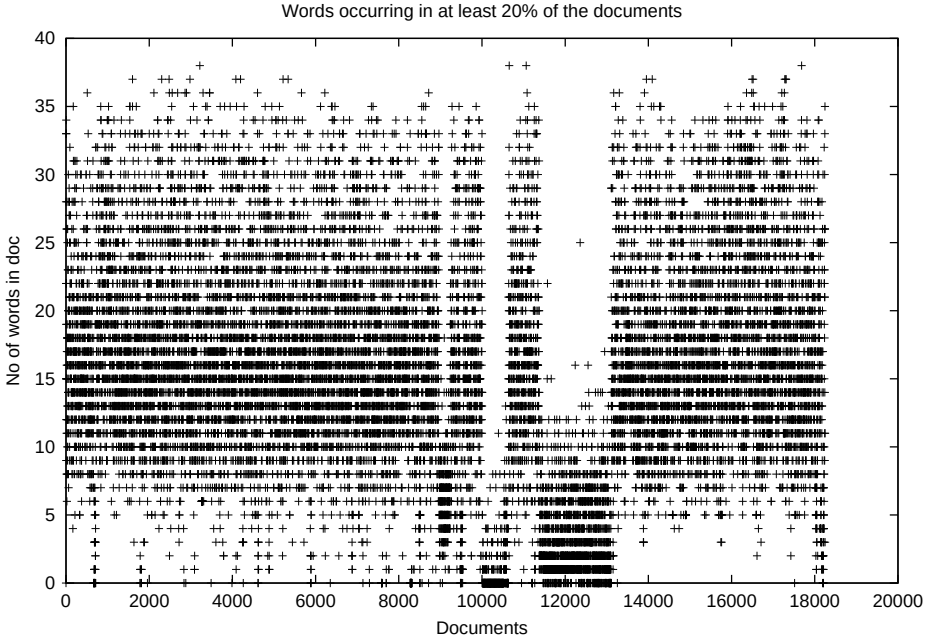

Figure 5.8: Diagram showing the number of frequent words (those in at least 20% of the documents) occurring in each document. Again, any word that occurs multiple times in a particular document is only counted once for that document. Each document has an integer ID indexed from 0 to 18,248. N.B. Figure 5.8 has three lower frequency troughs between document ID 9000 and ID 13000. This is due to the documents in this section of the Reuters corpus containing relatively few of the common words and many abbreviations; there are many shorter documents between 9,000 and 13,000.

parison to the other data structures.

- Efficient - storing the CMM as an efficient representation with a switch value of 300 (see section 3.1.2).
  CMM saturation is 0.007.
  CMM uses 10,958,614 bytes. The memory usage is much lower than the compact or binary representations.
  Memory use for array of words 475,000 to retrieve every 50th word etc. and to retrieve the word position for binary vector generation.
  Memory use for array of document IDs 73,000 to store the integer index for binary vector generation.
  Total memory use is 11,079,114.

5.4.2 Training Times

Twenty training times were noted for each data structure technique and the mean time for training was calculated. The table below gives the mean training

time in seconds for each algorithm.

	IFL	Hash Table	Hash Compact	efficient CMM
Training time (secs.)	1,965.365	1,570.48	1,568.96	98.31

5.4.3 Serial Match

The words are read from the data file and for each of the selected words in turn, the associated documents are retrieved and written to an output file. Ten times were recorded for each serial match for each data structure method and the mean time calculated. The mean retrieval time in seconds is given in the table below.

	IFL	Hash Table	Hash Compact	CMM - efficient
First 100	0.143	0.128	0.13	0.904
Every 50	0.253	0.223	0.202	1.907

5.4.4 Partial Match

A graph is given for each partial match evaluation, listing the number of words to be retrieved (at least M) on the x-axis and the mean time in seconds for 20 retrievals on the y-axis. A separate plot is shown for each data structure on each graph.

Partial Match on the First 100 Words

The graph of the mean time in seconds for the retrieval of M of N matching words from the set of the first 100 words taken from the canonical list is given in figure 5.9 for each data structure.

Partial Match on Every 50th Word

The graph of the mean time in seconds for the retrieval of M of N matching words from the set of every 50th word taken from the canonical list is given in figure 5.10 for each data structure.

Partial Match on Words Occurring in at Least 20% of the Documents

The graph of the mean time in seconds for the retrieval of M of N matching words from the set of the words present in at least 20% of the documents is given in figure 5.11 for each data structure.

5.5 Analysis

5.5.1 Memory Usage

If we disregard the final array of words from the memory totals of the inverted file list, hash table and hash table compact (the array was only included for consistency) then the memory totals in bytes for the four data structures in 64-bit

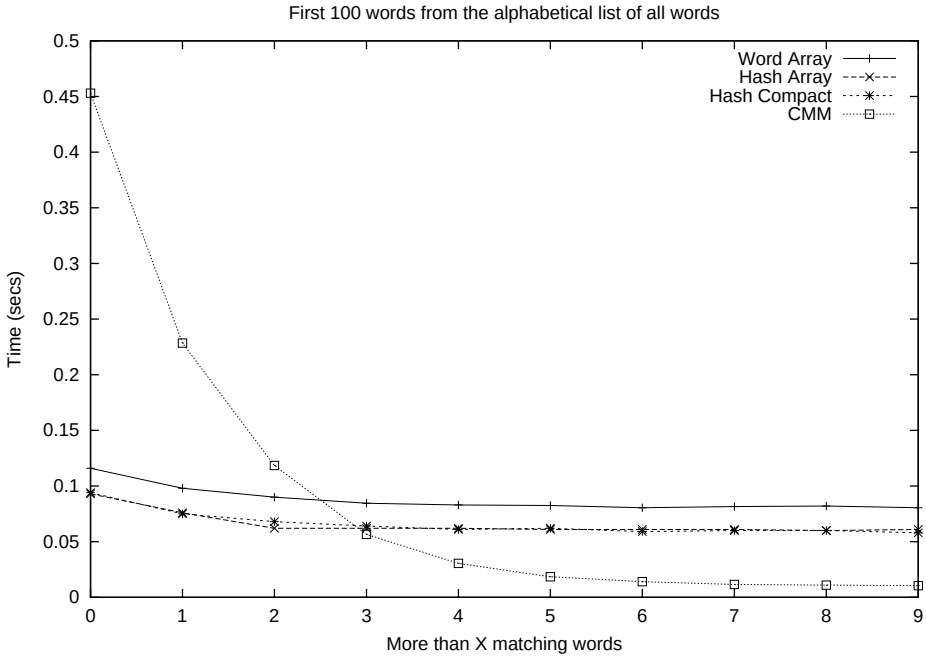

Figure 5.9: Graph of the retrieval time for M of N matching with the first 100 words from the list of all words.

mode are in ascending order: IFL (10,471,072), hash table compact (10,527,632), CMM (11,079,114) and hash table (11,081,478). If the IFL and two hashing data structures are compiled using 32-bit compilation then their respective sizes in bytes are: IFL (5,509,536), hash table compact (6,077,814) and hash table (6,155,828). As stated previously, the IFL requires the least memory but, as can be seen from the subsequent analyses, is slower for retrieval than the hash tables. The hash table compact (storing an integer index to the document lists) is more memory efficient than having empty list array elements in the standard hash table. The CMM has the second highest memory usage but lower than the conventional hash table. None of the memory usage statistics is significantly higher than the other data structures; they are all within 10% of each other. Therefore, we feel the timing comparisons for training and retrieval provide the most informative insight into the optimum data structure for an IR system of the structures evaluated.

We can see that the efficient CMM has a far lower memory usage than a binary or compact CMM due to the ability to switch to the most memory efficient representation for each row. The efficient CMM requires only 25% of the memory required for the binary representation and 50% of the compact memory requirement.

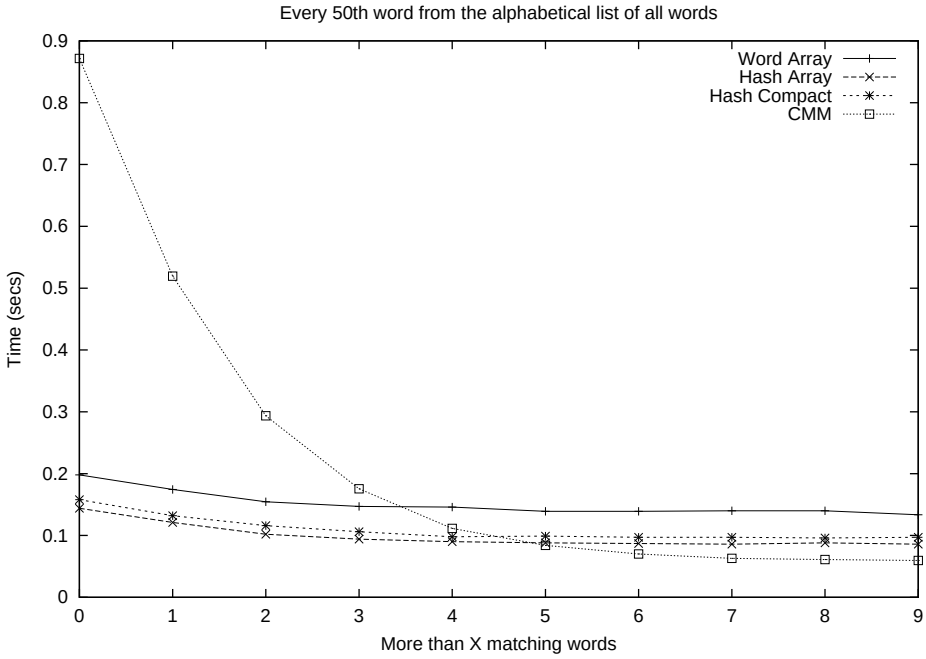

Figure 5.10: Graph of the retrieval times for M of N matching when taking every 50th word from the list of all words.

5.5.2 Training Times

The IFL is the slowest to train as expected as each search to locate the position of the word to link to the document requires a binary search through the entire array of words $O(\log n)$ before the document may be inserted in the word's list. The training times for the two hash table approaches are similar, as we would expect with both using the same document insertion procedure. The method calculates the hash location for the word and prefixes the document ID to the corresponding list of documents. However, by far the quickest to train is the CMM. The CMM does not require the lengthy search for the word to add the document ID to the word's list. Although the hash table is $\Theta(1)$ for word search initially, if there are collisions, it will degenerate to $O(n)$. The CMM simply associates the word with the document in a single step association procedure so when the word is input, the document will be recalled. The CMM takes 0.063 as long to train as the hash tables. This is a significant difference. The CMM reads in the same data as the others so the file access can only constitute a minor part of the IFL and hash table times. The majority is employed inserting the words and word-to-document associations in the data structure whereas this time is minimal for the CMM.

5.5.3 Serial Match Recall

The two hash tables are the quickest for the serial match as the match for each word will be approximately $\Theta(1)$. The IFL is marginally slower due to

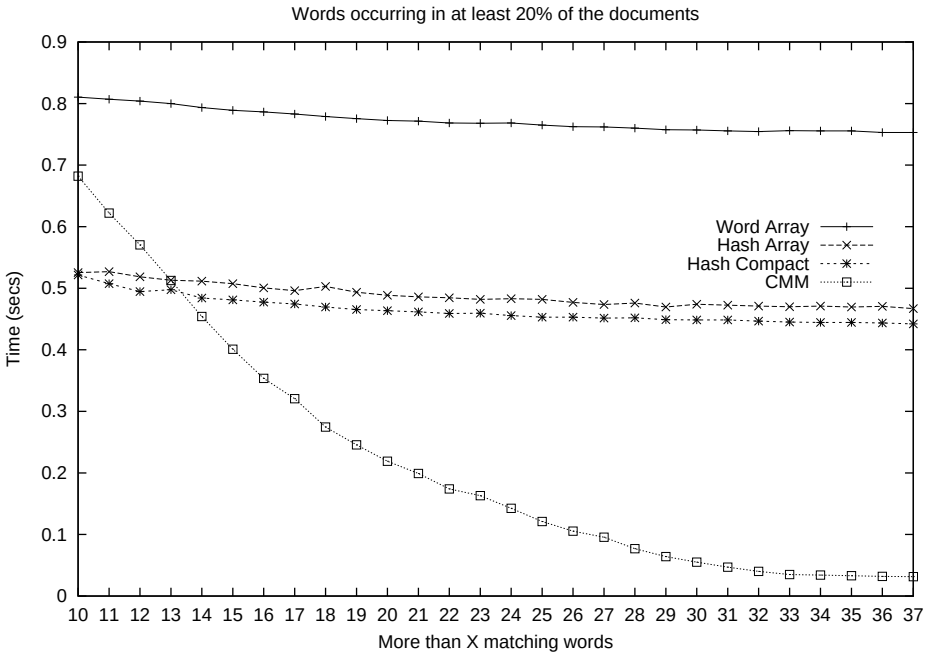

Figure 5.11: Graph of the retrieval times for M of N matching with frequent words (those in at least 20% of the documents).

the $O(\log n)$ binary search through the word array. The CMM is significantly slower for serial match. For each word, the word has to be translated to a binary vector, input to the CMM, the binary output vector retrieved and the set bits in the output vector identified to retrieve the IDs of the matching documents. The CMM takes 7.06 times longer for the first 100 word serial retrieval and 8.55 times longer for every 50th word serial retrieval than the standard hash table.

5.5.4 Partial Match Recall

The IFL is slower than the hash tables in all instances of partial matching. We would expect this due to the binary search $O(\log n)$ through the IFL to locate the words prior to finding the associated documents. The two hash tables are both faster than the IFL in all instances and produce very similar results, the differences between the graphs are negligible. The hash table compact does appear slightly faster although the difference may be attributed to the slight variation in the C/C++ timing utility and processor operation. The CMM is slower than the hash tables for low frequency partial match but for higher frequency partial match the CMM excels. The time curve for the CMM starts above the hash table for all graphs but falls below at 'at least 3' words matched on the 'first 100' word graph and at 'at least 5' on the 'every 50th' word graph. For the words present in at least 20% of the documents (see figure 5.12), the CMM is faster than the hash table with an increasing difference from 'at least

13' or more. The difference will level off and eventually fall as the y-axis forms

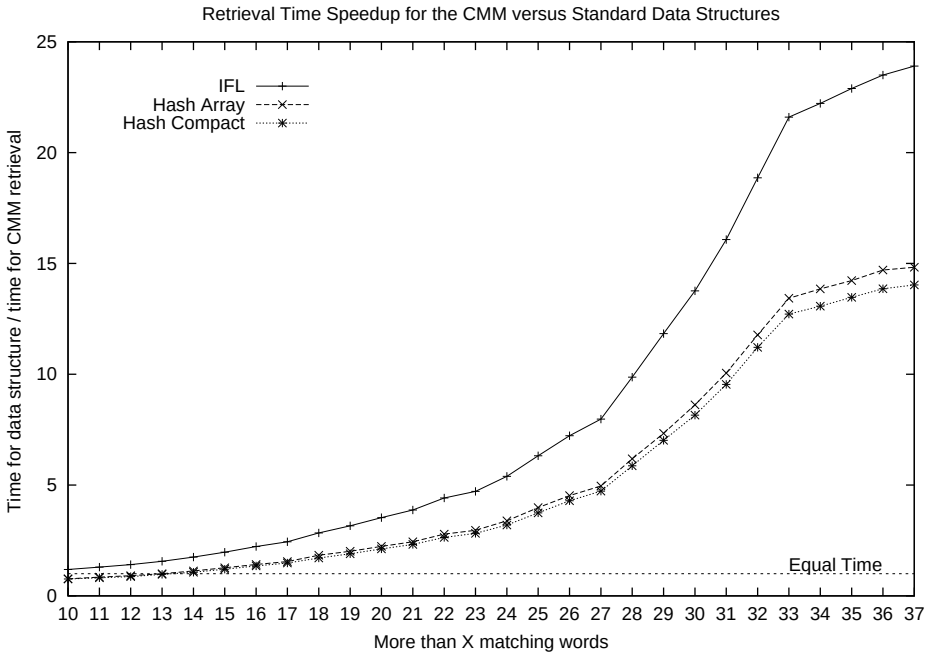

Figure 5.12: Graph of the speedup of the CMM versus the other three data structures when retrieving frequent words (those in at least 20% of the documents). The CMM is up to 24 times faster than the other data structures for identical partial matching.

a lower bound asymptote to the CMM curve and the retrieval speed of the hash table will continue to fall thus slowly approaching the CMM curve. However, the hash table will only approach the CMM slowly and may never reach it due to the intrinsic overheads of the data structure (collision resolution) so the CMM is preferable for partial matching.

In all cases the number of words to be matched affects the times, as we would expect. The 'every 50' match is slower for all data structures than the 'first 100'. This is because there are 189 words in the 'every 50' set compared to 100 words in the first 100' set to be matched to retrieve the documents containing the required number of words. Also, the 'first 100' words are retrieved more quickly by the 'for' loop from the word list. For the three data structures: IFL, hash and hash compact, the retrieval time for M words from the 'first 100' set is *just over 50%* of the time for the equivalent number of words from the 'every 50' set. For the CMM, the retrieval time for M words from the 'first 100' set is *just under 50%* of the time for the equivalent number of words from the 'every 50' set. The quicker word access time for the 'first 100' match has more effect on the overall retrieval time of the CMM than the other three data structures.

The match operation for the hash tables is very similar in all instances so for each evaluation 'first 100 words', 'every 50th word' and 'words in 20% of the documents' the timing only reduces slightly as the number of partial matches reduces. The procedure for the hash table matching varies little. The method reads in the words to be matched one at a time. For each word, the word is hashed to find the associated document list; the documents are retrieved from the list; for each document retrieved, the counter is incremented in the array of document counters to indicate the retrieval; and finally, the array is traversed writing to a file all documents that exceed the number of matches required. The only variation is in the final step where, as the number of matches required increases, the retrieval quickens as less documents are written to file. For the CMM the match operation timing reduces rapidly as the number of partial matches increases. The initial step is to read in the words to be matched, generate their associated binary vectors from the word array position, superimpose and input to the CMM. This is identical in all instances. To vary the number of matches the threshold is adjusted. When the threshold is low, the binary vector retrieved from the thresholded output will have more bits set. Thus when this binary vector is matched against the document vectors there will be more matching documents and retrieval will be slower than for higher thresholds where the thresholded output will be relatively sparse with fewer matches required.

5.6 Conclusion

We introduced a novel neural approach for storing word-document associations in an IR system. The neural approach uses single-epoch training, superimposed storage and associative partial matching in a binary matrix where new associations are added incrementally. We compared this novel approach to existing standard data structures for storing word-document associations.

For repeated serial, single word matching the compacted hash table is probably the most efficient methodology, the storage is second best for the compacted representation and the retrieval speed is fastest and second fastest for the two retrievals. The standard hash table produced the fastest retrieval time for the serial retrieval of the documents matching the first 100 words with the compacted variant second fastest. The compact hash table was the fastest to serially retrieve the documents matching every 50th word from the alphabetically sorted list of all words. The CMM is comparatively slow for the serial retrieval taking up to nine times longer to retrieve the documents than the fastest data structure.

For partial matching the CMM performs best. The memory usage is higher than the IFL, and hash compact but lower than the conventional hashing ap-

proach. The retrieval speed is superior to the other three approaches for partial match; the greater the number of words to be matched at each retrieval, the more superior the CMM is.

We recommend the CMM for Information Retrieval word-document association storage, as the approach is superior with respect to training time and retrieval time with comparable memory usage. The superimposition of the input vectors allows one-shot multiple word retrieval and partial match. We also note that more word to document associations could be added to the existing association CMM without significantly increasing the memory usage due to the superimposed storage. The CMM could therefore be used in an incremental system, although we do not evaluate incremental data structures here we noted the incremental training approach where associations are incorporated sequentially and overlaid with the existing trained associations. The efficient CMM can switch to the most memory efficient storage representation for each row to minimise the storage requirement of the word-document CMM as new associations are added. For the other data structures evaluated, additional associations would increase the memory use of the list array with one additional list node for each additional association. If additional words were added the word arrays would need to be incremented by two locations for the hash tables to keep the array less than 50% full. The word array would need to be extended by a single location for the IFL but the new word would need to be inserted in the correct alphabetical location and not just appended to the end of the array. We can add word-document associations to the CMM until the matrix is saturated, i.e., every word is linked with every document.

For our evaluation in this chapter, we assumed that all words searched were present and there were no spelling errors. We demonstrated our spell checker in chapter 4 and we also evaluate our overall MinerTaur system incorporating both the spell checker and word-document association matrix using mis-spelt words in chapter 6. The inverted file list, the two hashing algorithms and the word-document matrix evaluated here have no error correction facilities included. If a match was attempted on a mis-spelt word or a word not present in the stored associations then all four techniques would search the stored word list examining the maximal number of words for the respective search techniques: binary search for the inverted file list and word-document matrix and hashing for the two hash algorithms. They would all fail to match the search term and hence would retrieve no matching documents. The time for this word list retrieval would equate to the maximum word list search time for the respective data structures as the binary search is maximal for an item not present and the maximum number of hashing locations would be probed for the hashing algorithms.

Chapter 6

Information Retrieval

In this chapter, we describe out integrated modular Information Retrieval system: MinerTaur. We evaluate memory usage statistics for the various modules. We compare the training time for MinerTaur to a benchmark IR system: SMART. We evaluate the retrieval time for a wide range of queries supplied to MinerTaur using various configurations of the system modules. We then compare the recall and precision figures of various configurations of MinerTaur against various configurations of the SMART benchmark system.

6.1 Introduction

A methodology is desired to: process documents unsupervised and generate a multi-level and compact index using a data structure that is memory efficient, speedy, incremental and scalable; overcome spelling mistakes in the query; suggest alternative spellings for query terms; handle paraphrasing of documents and synonyms for both indexing and searching; to focus retrieval by progressively minimising the search space and finally calculate the document similarity from statistics autonomously derived from the text corpus. Documents may be retrieved according to the user's exact requirements by progressively refining the search and iteratively employing finer-grained and more specific matching techniques.

Document and text processing may ultimately be considered a natural language processing task. In our methodology described in this thesis we simplify the process. We extract semantic word relationships by using averaged word contexts (statistical co-occurrence vectors) to generate the hierarchical thesaurus from which we can score the similarity of the words as described in chapter 2. We store all word-document associations in a matrix structure as full text indexing is more comprehensive and accurate [6]. We use the scores from the query words and the thesaurus search to feed into the matrix and accumulate document scores. The document with the highest cumulative score

denotes the best matching document. The technique is wholly data-driven and unsupervised; the only knowledge regarding the words are the statistics of their co-occurrence to infer semantic relationships and the associations between the words and the documents containing them, we require no domain-dependent background knowledge or prior classification.

6.2 The MinerTaur System

Our MinerTaur system comprises three modules, a spell checker, a hierarchical

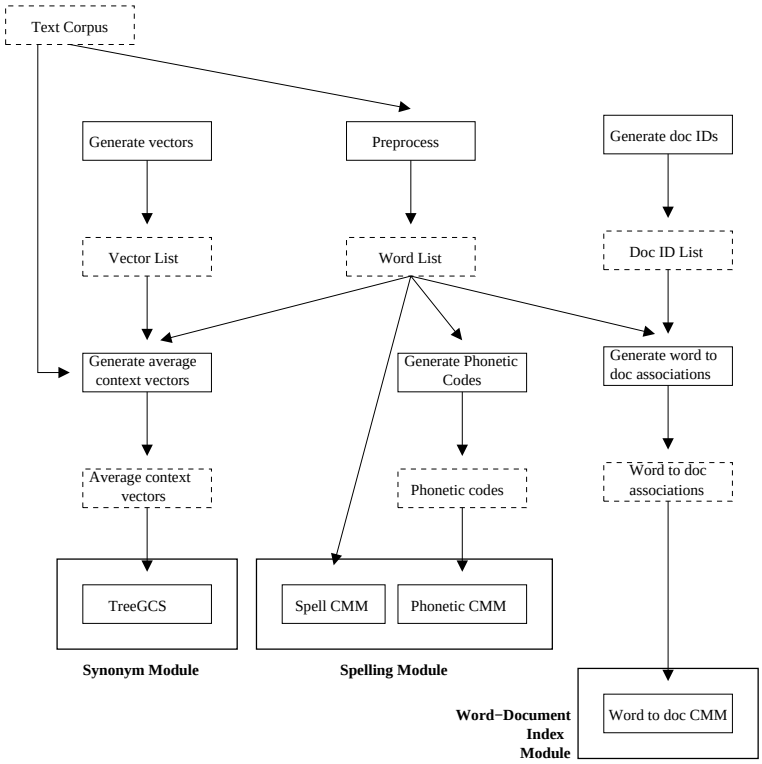

Figure 6.1: Figure illustrating the modular architecture of our MinerTaur system and illustrating the training process. The dashed boxes indicate artefacts and the solid boxes illustrate system processes and modules.

thesaurus and finally a word-document association index (see figures 6.1 and 6.2 for an overview of the system during training and querying respectively). In the training figure 6.1, the hierarchical thesaurus is represented by the TreeGCS process and the preceding processes and artefacts derived from the text corpus. The spelling system comprises the Spell CMM and Phonetic CMM processes and their ancestor artefacts and processes again derived from the text corpus. The word to document CMM represents the final system module and is derived from the text corpus and a list of the document identifiers. In the query figure 6.2, each query word in turn (iteratively) passes through each of the three

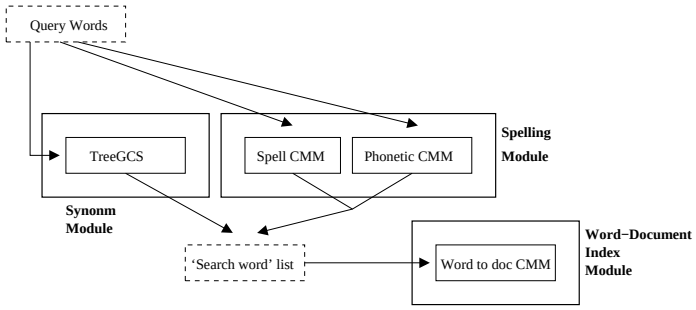

Figure 6.2: Figure illustrating the modular architecture of our MinerTaur system and illustrating the querying process. The dashed boxes indicate artefacts and the solid boxes illustrate system processes and modules.

modules, which are used serially. The query word is first validated against the lexicon in the Spelling CMM. If there is a spelling error, the query word is passed to the Spelling CMM and Phonetic CMM to generate a list of candidate matches from the lexicon. The candidate matches from the spelling module are amalgamated in a scoring process to rank the matches and the user selects the correct spelling from the ranked list. The Spelling CMM also generates a list of word stems if required. The TreeGCS module is used to determine any synonyms at the user's behest. Finally the query words, stems and synonyms are passed to the word to documents CMM in a search word list to retrieve and score all documents that match.

Front-end spell checking module to isolate any mis-spelt query words. We only perform isolated word error correction [63] and identify query terms not present in the lexicon. Our spell checker does not perform context-based spell checking [63] to identify substitution errors where one correctly spelled word is substituted for another for example 'there' and 'their'. This context-based error correction necessitates complete natural language processing capabilities and may not be feasible within a terse, term-list IR query. Our spell checker validates each query word against the lexicon. If a word is not matched we assume a spelling error and our spell-checking module provides a list of the best candidate matches for the user to select from. Our spell-checking module also provides a word stemming capability so the user can input a word stem and the spell checker returns a set of matching stemming variants from the stored lexicon. For example, if the user inputs 'engine' our word stemmer will suggest 'engines', 'engineer' and 'engineering' from the lexicon. This allows the user to ascertain all appropriate words in the corpus without requiring knowledge of the corpus vocabulary and to expand their search to all apposite words.

Hierarchical thesaurus. We employ the statistical gathering and inference

methodology described in chapter 2 to automatically develop a hierarchical thesaurus from word context correlations in the text corpus. TreeGCS evolves an unsupervised, hierarchical semantic map of the corpus using word co-occurrence statistics. We can then exploit the intra-cluster distances and the inter-cluster distances to ascribe scores to query words and their synonyms. Once the hierarchy has been generated the hierarchical data structure evolved can be written to a file and stored. The stored data structure can be read in by the system when the IR system is initialised. Hence, hierarchical thesaurus generation is a one-shot process and only the data structure then needs to be read from a file which is a rapid procedure. This also allows incremental training. We can write the data structure to the file as it evolves during training and the system can read in the data structure from the file to obtain the most contemporary hierarchy each time the system is initialised.

Word-document matrix implemented using orthogonal word vectors and orthogonal document vectors to ensure perfect matching with no false positives or false negatives. As the Reuters' dataset [82] used in our retrieval evaluation comprises 18,249 documents (the same documents used in our evaluation in chapter 5), indexing 48,766 words with 890,071 word-document associations to be stored, we are able to use the orthogonal binary vectors. Each row of the matrix effectively represents and indexes a particular word and each column effectively indexes a specific document. We described the word-document matrix implementation in chapter 5. We index all words except a small set of stop-words (i.e., very common words with a very low discriminatory power with respect to text corpora such as 'the' and 'and' taken from the SMART stop-word list[1] [99]). We have reduced the number of word-document associations to be stored compared to chapter 5 where we stored the associations for commonly occurring words. As we have removed the stop-words we have removed many word-document associations. Unlike many other systems described previously in this chapter, we store word-document associations for infrequent corpus words. We only remove very common words and hypothesise that low frequency words may have a low discriminatory power across the corpus but they have a high discriminatory power with respect to the documents that contain them; they uniquely identify a single or small set of documents and should be indexed by the IR system.

6.2.1 Query Processing

We provide a walk-through for an example query *"grain, wheat, oatts, barley, soy"* which includes a spelling error for demonstrative purposes. For a multiple

[1] As we compare our MinerTaur system against the SMART system we used the SMART stop-word list to process our system and thus to maintain standardisation across the evaluation.

word query, each individual query word is processed in order *grain* then *wheat* then *oatts* then *barley* then *soy* and an accumulator vector is produced with the scores for each document with respect to each individual query word. After all query words have been processed the document accumulator vectors are added and a total score for each document with respect to all query words is produced.

The first question presented to the user is ''Do you want word stemming?''. This will perform word stemming matching on all query words and present the user with a set of word stemming variants for each query word input. If the user selects '*yes*' then this facility is activated for all query words otherwise it is switched off for all query words.

The first step for our MinerTaur system is for the user to input their first query word. The query word is converted to the spelling binary vector, as described in section 4.6.1, and input to the spelling CMM. We perform an *exact match retrieval* (described in section 4.7.3) to identify any lexicon words that match all letters of the input spelling and have an identical length. If there is an exact match then the output from the spelling CMM is a single binary vector representing the matching word. The first, second, fourth and fifth words in our example query *grain*, *wheat*, *barley* and *soy* are correctly spelt and present in the lexicon so all represent an exact match. The binary vectors representing the words trained in to the spell checker are identical to the binary vectors representing the same word trained in the word-document matrix. Therefore, we can use the thresholded output vector from the spell checker as the input vector to the word-document matrix. The user is asked to supply a score (userScore) for the word between 1 and 10 if they wish to denote the relative importance of the word to the query. A higher score signifies higher importance. For our evaluations in this dissertation we did not input user scores so all *userScores* were automatically set to the default value of 1. The topics assigned to the Reuters document collection do not indicate any degree of importance for each topic word so we left all word scores equal. The exact matching word is assigned a score of 1000*userScore. We describe why we use a scale factor of 1,000 later.

If there is no exact match from the spelling CMM the word is not present in the corpus so we assume the query word is spelt incorrectly and perform a *best match retrieval*. We identify the best matching words from the lexicon trained into the spelling CMM using the query word as input. The user is presented with the top 10 matches (or less if fewer matches) and allowed to select the correct word from the list. From our example query *oatts* is incorrectly spelt and MinerTaur presents the user with the top matches in order: *oates, oats, boats, coats, floats, outs*. The correct match is the second word in the list which the user may select. This corrected word will now be used throughout

the further query processing. If the user does not select any words from the top 10 matches and there are more matches available in the candidate set retrieved then the user can select from the next 10 best matches and so on until all possible matches are exhausted or the user selects a matching word. If the user does not select a matching word then no further retrieval is performed for this query word. Otherwise the selected word is passed to the word-document matrix and also to the synonym hierarchy traversal if this option is selected. Again the user can input a score (userScore) for the word if they desire to signify the relative importance of each query word between 1 and 10. The best matching word are all assigned a score of 1000*userScore. For our evaluations in this dissertation, we left all user scores at 1 as stated previously.

If the user has activated the stemming facility, then we perform the word stemming operation described in chapter 4 and produce a list of stemming variants for the query word. From our query example, *grain* will produce the stemming variants {*grains, grain-filling, grain-producing, grain-quality, grains-oilseeds*} while *soy* will produce the stemming variants {*soya, soyabean, soyabeans, soybean, soybean-planted, soybean-producing, soybean-specific, soybeans, soybns, soyfood, soymeal, soyoil, soyproduct, soyproducts*}. The user can select any variants they wish and these will be passed to the word-document matrix to retrieve matching documents and also to the synonym hierarchy of this option is selected. The stemming variants are all assigned a score of 1000*userScore where userScore is the value assigned to the word stem by the user, a value between 1 and 10 to signify word importance. We left all user scores at 1 for the evaluation in this dissertation.

The user is asked if they wish to perform synonym matching. If they do then all variants (spelling or word stemming variants) selected for the query word are matched against the synonym hierarchy in turn. For the system evaluated here, we only clustered 2,192 frequently occurring words from the total Reuters' vocabulary of 48,766 words to ensure the average context vectors were close to the true mean and not unduly biased by a lack of averaging. Thus not all query words will have a match in the synonym hierarchy. An empty set is returned for an unmatched word. Each word is input to the TreeGCS hierarchy and the best matching cell is located if the word is present. Any other words (synonyms) mapping onto this GCS cell are assigned a score of 0.5. At this stage we have not yet empirically derived an optimum value for this setting but as can be seen from the later recall results, adding in the synonym hierarchy increases the recall significantly so the setting may be near optimal. From our running example query, the query word *barley* maps onto the same TreeGCS node as {*sorghum, maize*} so *sorghum* and *maize* each score 0.5.

We then perform a Dijkstra shortest path traversal [24] through the cluster containing the best matching cell using the connections in the GCS network, see figure 6.3. The cumulative distance is calculated to each cell in the cluster,

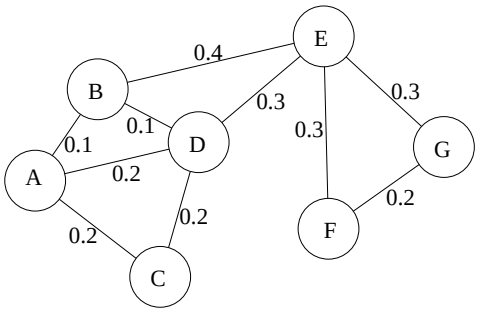

Figure 6.3: Figure illustrating a GCS cluster and the cell distances.

all distances are normalised in a range 0-0.25 and the normalised distance is subtracted from 0.5 to ascribe a score to each cell in the cluster. Any corpus words mapping on to those cells are thus ascribed the respective scores. From the example query, {*wheat, corn, cotton, grain*} are in the same cluster as *barley* and score 0.45, 0.43, 0.37 and 0.25 respectively. The scores for the synonyms derived from the hierarchical thesaurus are floating point numbers in the range 0 to 1 with accuracy dependent on the local C++ installation (floating point accuracy). We need to convert all word scores to integer values for inputting to the word-document CMM during the integer vector accumulation phase of retrieval (as the integer vector requires integer values). The range 0 to 1 floating point needs to be converted to a suitable integer range. All scores are multiplied by 1,000 to give scores ranging from 500 for words mapping on to the same cell as the query word down to 250. These scores are then multiplied by the userScore (importance) for the original query word. We score from 500 for this cluster to ensure that synonyms do not over influence the document scoring (they are half as important as exact matching words). We score down to 250 as the all words in the next matching cluster in the scoring process are awarded 250 so we score down to this value in the best matching cluster.

The integer selected is a trade-off between speed and accuracy. The CMM processes integer vector inputs by repeatedly activating the CMM row a number of times equivalent to the value of the integer. The more times the row is activated (i.e., the higher the integer) the slower the retrieval but paradoxically the more accurate the word scoring phase of our IR system. The greater integer range gives higher scoring precision thus percolating greater precision to the cumulative document scores. We selected an integer range of 0 to 1,000 as we felt this range provides acceptable precision while maintaining a high retrieval speed. The range is arbitrary and selectable for the installation. A faster pro-

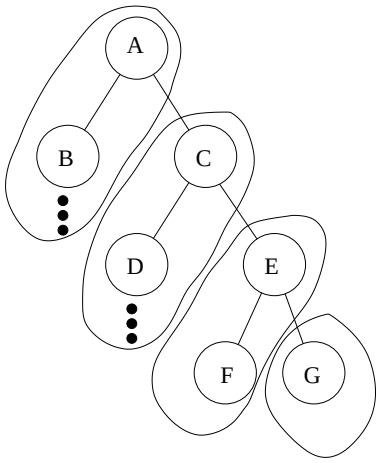

Figure 6.4: Figure illustrating the TreeGCS hierarchy scoring traversal. The word we are searching for is in cluster G. We assign scores to all words in cluster G according to the cumulative Euclidean distances in the cluster. We then score all other words in the hierarchy by assigning scores to the clusters. All clusters and cells categorised together are awarded identical scores. All cells in descendent clusters of node D would be awarded identical scores.

cessor would permit faster retrieval so a higher precision (larger integer) range could be used. For the 180MHz processors used for our evaluations and analyses, we felt 0 to 1,000 was the optimum range. We did not empirically evaluate the value with respect to speed of retrieval or scoring precision but may do so at a later date to verify our range or indeed select a new optimal range.

Once all cells in the query word's cluster have been assigned a value using the reciprocal of the *Dijkstra shortest cumulative path distance* algorithm [24], we assign scores to the remaining clusters in the hierarchy. The scores are the reciprocal tree distance and all cells in each cluster are assigned the same score. As the clusters are disjoint we felt it was inappropriate and also computationally expensive to calculate cell distances from the original query word's cell to all cells in the GCS network. We exploit the grouping advantages of the hierarchy and its ease of traversal by assigning scores on a per cluster basis. As mentioned in chapter 2 our hierarchical tree structure is symmetric. In figure 6.4, we do not know which descendent of node D is closer to G so we assign all cells in all descendent clusters of D (and thus all lexicon words mapping on to those cells) the same score. The algorithm traverses the tree moving from parent to parent, halving the score on each step and assigning that score to all successor clusters not already scored.

```
Score := 0.5
While unprocessed cluster {
    Score := Score / 2;
```

```
    Move up from current node to parent;
    Depth first search from parent to all successor nodes;
    For each successor node not yet processed{
        If cluster is a leaf
            Assign current score to all cells in cluster;
        Mark cluster as processed;
    }
}
```

All cluster scores are multiplied by 1,000 and converted to integer values as we require integer inputs to the word-document matrix giving a score range of 500 to 0 for the synonyms. From the example query, {*rice, sugar, soybeans*} are in the next hierarchy cluster to *barley* and all score 0.25. {*land, coal, products, silver, gold*} are in the next hierarchy cluster and all score 0.125 and so on through the hierarchy. This score value is further multiplied by the userScore for the query word to signify relative importance of the query term. All words are assigned the score of their best matching cell's cluster. As the C++ language only processes numerical values (floats and integers) to a specific degree of precision, many scores will be rounded to 0 during conversion so only semantically close synonyms (those with stochastically similar co-occurrence vectors) will score highly. Many will be too distant in the hierarchy to score or will score very low values.

Each input row of the word document matrix is activated with the integer score of the corresponding word from the initial spelling match, the stemming match if selected and the synonym hierarchy traversal if selected. We sequentially input each word that scored more than 0 to the word-document matrix to keep the scores awarded to documents separate. The matrix precludes a parallel score match where the scores differ as we do not know which documents have matched which specific words and which specific scores. We would just retrieve an output activation of the total match score. Each attribute of the integer vector represents the score for that document with respect to the query word, each document takes the highest integer value with respect to the query word: a query word match, a stem match or a synonym match. We threshold the CMM output activation vector from the query word input, stemming input or synonym input using the Willshaw threshold set to the score value to produce a binary thresholded vector. We multiply the binary vector by the input integer score to produce a *current score vector*. This awards each document containing that word a score equivalent to the input word score. We use an *accumulation integer vector* to collate the output for each query word in turn (collating the query word, its stems and its synonyms). We compare the *current score vector* for each retrieval against the *accumulation integer vector*. If the value of the *current score vector* is higher than the *accumulation integer vector* for

any attribute, the *accumulation integer vector* attribute is set to that score for that document. We essentially maintain a separate accumulating score vector for each query word with each document attribute accumulating the maximum score for that word.

We note that serial matching is slow as we demonstrated in chapter 5 but we have not yet optimised the synonym module while we assess the merit of the hierarchical clustering technique and consider alternative options so we have not yet optimised word retrieval and thus employ serial matching. We can group the synonyms and their respective vectors according to the word scores, i.e., all vectors for identically scoring words are superimposed to speed and parallelise retrieval. The single superimposition vector may then be input to the word-document CMM and the CMM thresholded at one to retrieve any document matching any input word. The resultant thresholded binary vector can then be multiplied by the current synonym score and added to the accumulation vector if the current synonym score is greater than the existing score for that document attribute.

The whole process of exact match, best match if no exact match, word stemming if selected, synonym hierarchy traversal if selected and passing the selected words to the word-document matrix to generate an *accumulation integer vector* is repeated for each query word. This produces one *accumulation integer vector* per query word. We then add all *accumulation integer vectors* together to give a single *total score vector* for the particular user query. Each attribute of the *total score vector* indexes a document, analogous to the output vectors for the word-document CMM and the value of the attribute represents the total accumulated score for all query words for the corresponding document. We can then select the best matching document(s) by identifying the attribute(s) with the highest values and returning the document(s) indexed by the maximal attribute(s).

From our example query *"grain, wheat, oatts, barley, soy"* -
If we have no stemming or synonyms then there are 2 top matching documents after spelling correction each with a score of 5000 indicating that they match all query words (each query word carries a score of 1000 in our evaluations in this chapter). The best matches are:
15762, 17936 @score5000
We list document 17936 below with the five query words in bold to allow the reader to view an example document's contents:
17936: winnipeg oilseeds were higher , with rapeseed gaining 2.10 to 2.50 dollars a bushel and flaxseed up 1.50 to 2.70 . oilseeds were supported by the rally in the chicago **soy** complex , trade sources said . further , recently active producer selling subsided , while exporters provided underlying support in flaxseed and crushers

bought rapeseed , they said . feed **grain** trade was quiet with locals and commission houses dominating activity . rye fell 0.10 to 0.20 . **barley** ranged 0.40 lower to 0.40 higher . **oats** ranged unchanged to 0.30 lower . **wheat** fell 0.20 to 0.50 .

If we have no stemming but activate synonyms then the top 15 matching documents are:
15762, 17936 @score5000
6232, 14593, 15768, 15769, 15770 @score4000
2060, 4020, 8663, 10878, 11222, 13425, 13426, 14266 @score3500
We list document 14266 below with the three of the five matched query words in bold along with the synonyms *maize, sorghum* from the query word *barley*. Both synonyms score 500 as they map onto the same TreeGCS cell as barley giving a total document score of 3500. We take the highest score for each query word. Barley is not matched so the highest score for barley derives from its synonyms, so the document scores 500 with respect to this query word plus 3000 for the three matching query words:
14266: the argentine **grain** market was quiet in the week to wednesday , with prices rising slightly on increased interest in **wheat** , millet and birdseed . **wheat** for domestic consumption rose six australs per tonne to 118 . for export it rose eight to 108 per tonne from bahia blanca , increased 0.50 to 104 at necochea and was unchanged at rosario at 108.30 . **maize** increased one to 90 per tonne at buenos aires , was unchanged at 82 in bahia blanca , increased 0.50 to 85 at necochea and fell one to 88 at parana river ports . **sorghum** from bahia blanca increased 0.50 australs to 76.50 per tonne and dropped one to 75 at rosario . it was quoted at 75 at villa constitucion , san nicolas and puerto alvear . **oats** were unchanged at 168 per tonne at buenos aires . millet from buenos aires and rosario rose five per tonne to 140 and birdseed rose 15 to 205 at buenos aires .

If we have stemming but no synonyms then the top 15 matching documents are:
15762, 17936 @score5000
2060, 6232, 14593, 15746, 15768, 15769, 15770, 16773, 17492, 17935 @score4000
1259, 1835, 1836 @score3000
We list document 2060 below with the three of the five matched query words in bold along with the stemming variant *soybeans* derived from *soy* which scores 1000 as stemming variants score the same as query words giving a total document score of 4000:
2060: best basis bids posted by **grain** dealers at the close - chicago corn processors spot , 4 under may - unc l/h april , 1 under may - unc f/h may , may price - up 1 exporters april , 4 under may - unc new crop , 10 under dec - unc burns harbor april , 5 under may - unc merchandiser spot , 4 under may - unc chicago **oats** merchandiser spot , 20 over may - unc chicago **soybeans** exporters april , 3 under

may - unc new crop , 10 under nov - unc burns harbor april , 3 under may - unc merchandiser spot , 3 under may - unc chicago srw **wheat** millers 57 lb spot , 35 over may - unc elevator 58 lb spot , may price - unc new crop , 7 under july - unc exporters spot , ua new crop , 5 under july - unc merchandiser spot , 35 over may - unc toledo corn , spot , 5 under may - up 1 **soybeans** , spot , 1 under may - unc srw **wheat** , spot 58 lbs , 35 over may - unc illinois river - seneca corn , spot , 2-1/2 under may - up 1-1/2 **soybeans** , spot , 5 under may - dn 1-1/2 cif gulf - barge corn , april , 21 over may - nc **soybeans** , april , 24 over may offered - dn 2 srw **wheat** , unquoted pik certificates - percent of face value out by april 7 , 104-1/2 percent bid 105-1/4 asked - dn 1/2 nc - no comparison unc - unchanged

If there are too many matching documents for a particular query, we allow a more refined search to minimise the matching document set. We exploit the synonym hierarchy further by utilising any synonyms that map onto the same GCS cells as the query terms; we exploit the statistically closest words to enhance the search by expanding the query. N.B. we treat all query terms as equally weighted. At present we do not differentiate using the word weights initially assigned by the user. We pass each query word in turn to the TreeGCS hierarchy, we return any words that map onto the same GCS cell as the query word and set the corresponding bit representing that word in the single input vector. We use a single input vector to the word-document matrix representing the superimposition of all query words and statistically close synonyms by setting the vector bit representing each query word and their respective synonyms. When we have processed all query words, we present the input vector to the word-document matrix. We threshold the output activation vector at its highest attribute value to retrieve the best matching documents. The best matching document will be the document or small set of documents that contains the maximal number of query terms and statistically correlated synonyms. We only use synonyms that map onto the same cells as the query words as these are very statistically similar synonyms and we also want to minimise the search word set. If there are too many words in the search word set we may well retrieve too dissimilar documents, we want to expand the search slightly from the original query words but still maintain a tightly focussed search. This way we hope to reduce the number of best matching documents while preserving our search focus. This process has been implemented in the MinerTaur system but not yet fully evaluated to empirically assess the effect on the precision and recall calculated from the documents retrieved. The procedure should improve precision without detrimentally affecting recall. We also need to evaluate the procedure against a benchmark IR system such as SMART [99] to compare the recall and precision for queries which retrieve many documents that have identical scores with respect to matching words.

6.3 Evaluation

We evaluate our MinerTaur system against the benchmark IR system SMART version 11 from Cornell University [99]. We measure our system against SMART for training time. We compare the retrieval time for our 'basic' system against system configurations with each module (spelling, stemming, synonym hierarchy) activated in turn for a series of different length queries to demonstrate how the times compare, how the number of query words affects the system, whether it is O(1) or O(n) with respect to query words and the proportion of the overall system retrieval time occupied by each module in turn. We evaluate our MinerTaur system for memory usage and compare the memory usage of the spelling and word-document CMMs against the stored data files to evaluate the overheads introduced by the CMM data structure. Finally, we evaluate our MinerTaur system against SMART for *recall* and *precision* using 18,249 documents and 66 queries we extracted from the Reuters-21578 dataset. We provide statistical analyses of the 66 queries to investigate statistical correlations and system biasing of the query set to ensure impartiality of the query test set with respect to the systems evaluated. [82]. We describe how we pre-processed the corpus below. All analyses were performed on a SGI Origin 2000 with the following specifications (taken from the IRIX *hinv* command):

- 32 X 180 MHZ IP27 Processors
  Again the multi-processor system allows us to assess the respective systems in parallel. We can have one process from each evaluated system running an identical assessment simultaneously and hence, we can minimise timing variance.
- CPU: MIPS R10000 Processor Chip Revision: 2.6
- FPU: MIPS R10010 Floating Point Chip Revision: 0.0
- Main memory size: 8,192 MBytes
- Instruction cache size: 32 KBytes
- Data cache size: 32 KBytes

We generated the memory usage statistics for our MinerTaur system using the C/C++ sizeof() utility. We calculated the timing values using the C/C++ clock() function and the UNIX /bin/time function.

6.3.1 Memory Usage Evaluation

We derived the memory usage figures for the phonetic spelling CMM, the integrated Hamming Distance and n-gram CMM and the word-document association CMM using the C++ sizeof() function. The figures are tabulated in table

6.5. We do not include an exact figure for the growing hierarchical cell structure as the size is dependent on the complexity of the hierarchy; a more complex hierarchy signifies a higher memory usage. Paradoxically, a more complex hierarchy indicates more refined clustering which is more desirable for scoring the words as the scores are much finer-grained and fewer words are grouped on the same score. Hence, words awarded the same score will be more similar. The hierarchy is written to a file periodically and the most contemporary hierarchy is stored on disk and read in to the system during system initialisation so we have not included the memory usage figure of the hierarchy as it is constantly evolving. However, we do include the size of the tree and GCS files written to disk after a training run of 10,000 epochs with 1,000 GCS cells underpinning the superimposed hierarchy and 2,192 average word context vectors in the input space. The files contain a representation of the tree hierarchy and the structure and parameters of the GCS network, respectively.

6.3.2 Training Time Evaluation

We do not include a precise TreeGCS hierarchy generation timing as the generation process is not currently optimised while we consider possible improvements or replacements (see chapter 7 for a discussion) and also the hierarchy generation is a separate and incremental process. The hierarchy is written to disk periodically during generation and each time the IR system is initialised the most contemporary hierarchy is read in from the file. However, we do include an approximate figure for a training run of 3,000 epochs on a 1,000 cell GCS network with a hierarchy dynamically superimposed and with 2,192 average word context vector inputs. We measured the training time for our MinerTaur system as: the time to read in the GCS structure and superimposed hierarchy from a file stored on disk; the time to label the TreeGCS cells with the synonym hierarchy words using the average context vectors; the time to read in the spelling-word associations for the lexicon plus the time to read in the word-document associations. To label the GCS cells, the system reads in a word and its associated average context vector, finds the best matching GCS cell and labels that cell with the word. The figures are given in table 6.7. In table 6.8 we provide a comparative training time for the SMART system using the most commonly used SMART vector format and no stemming. MinerTaur trains with no stemming so for an authentic comparison we compare the non-stemmed SMART variant. We timed the SMART system by calling the training routine from within a C++ program and using the C++ clock() function to sample the system clock as the training routine was called and immediately as training finished. By timing from within a C++ program we are replicating our timing method.

6.3.3 Retrieval Time Evaluation

We perform timing comparisons for a series of retrievals with MinerTaur. We provide a series of comparative figures: a bare-bones ('basic') system time with no spell checking or synonym hierarchy traversal; a system with spell checking incorporated; a system with word stemming activated and a system with synonym traversal activated. This allows us to ascertain the proportions of the overall system retrieval time attributable to each system module. We timed our system by adding two clock() lines to the C++ code, one clock() that sampled the system time after the last word of the user query was entered and a second clock() that sampled the system time after the last matching document had been output. The results are detailed in section 6.4.2.

6.3.4 Recall and Precision Evaluation

We pre-processed the Reuters' dataset by removing all meta-information and just storing the body text[2] (demarcated by the <BODY> <\BODY> tags) for each document. We stored all documents regardless of whether they were from the test or training set in the Reuters database. We use the 18,249 documents from chapter 5 stored in a text file with one document per line. We could then use the line number as the identifier for the document allowing us to use the UNIX 'lines' function to retrieve the required documents. The SMART system uses the line number as the document identifier so this numbering approach ensured regularity across all systems. SMART outputs the line identifier for the matching documents and we could also exploit the line identifier to determine the bit to set in the binary vector representing each document and output the numerical identifier from the bits set in the thresholded output vector from MinerTaur. We converted all text in the document file to lower-case as our spell checker requires lower case. We ensured that all words and all punctuation marks were separated by spaces to allow words and punctuation to be discriminated by any text pre-processor. All systems are processing an identical text file to maintain equivalence.

We compiled a list of all topic sets for all 18,249 documents using the UNIX 'grep' facility, sorting the list into order using UNIX 'sort' and then applying the UNIX 'uniq' command to ensure no topic repetitions. We then selected 82 distinct topic sets as queries generally containing three words or more to ensure that the matching document set is reasonably small and compact. Topic sets with fewer than three topic terms produce too many matching documents. We did include 2 queries of two words but the words were very specific and each word matched few documents providing a narrow focus of retrieval. We ensured minimal repetition of topics and ensured that each query was unique. We se-

[2] We stored the meta-information in a separate file to allow us to generate the topics and queries for our evaluation.

lected all topic sets before commencing the evaluation to ensure impartiality over the systems, we felt we may favour our MinerTaur system if we selected any topic sets after evaluation had commenced. We also only selected queries where at least one of the evaluated systems retrieved at least one correct match in the top 15 matches and this left 66 queries from the initial set of 82 queries.

The default SMART system retrieves 15 best match documents for each query so we commenced our retrieval and precision calculation by retrieving the top 15 documents. We also tried to limit the matching document set to fewer than 15 documents for each query as we felt that searching for more than 15 matching documents was likely to produce human errors in the matching and counting process. We adhered to the 15-match threshold for all but two queries that had 16 and 23 matching documents respectively. We also felt that collating more than 50 retrieved documents for each system configuration investigated was likely to introduce human errors. We therefore calculate recall and precision figures for the systems for the top 15, top 20, top 30, top 40 and top 50 retrieved documents.

By evaluating a range of best matching documents, we eradicate any statistical irregularities in the queries and eliminate any bias towards a particular system. We ensured that we had maximal coverage of all topic sets and did not favour any particular topic from the set of all topics. We also ensured that for example, documents assigned the topics {*copper, gold, lead, zinc*} were retrieved with documents assigned the topics {*copper, lead, gold, zinc*} and all other permutations of the four topic terms. We also matched topic subsets so for example, when searching for topics {*copper, gold, lead, zinc*} we also retrieved documents that matched {*copper, gold, lead, zinc, silver, platinum*} and any other supersets of the query words. We input identical queries to all systems evaluated, just a simple list of the topic words. All words were awarded a score of 1 in MinerTaur to ensure equality as the Reuters' dataset does not differentiate between the topics so we do not attempt to impose any inequity.

The Reuters' dataset was designed for classification where the documents are trained against the topic sets. The terms used for the topics do not necessarily occur in the document body texts that we are using for our evaluation, many are hyphenated abbreviations, for example *sun-oil* for *sunflower oil* or *veg-oil* for *vegetable oil*. We therefore decided to convert the topics to terms present in our body text file. This benefits all systems equally as SMART and MinerTaur both index using words in the text corpus only. For example, we converted *sun-oil* to {*sunflower*} {*oil*}, *pork-belly* to {*pork*} {*belly*}, *nat-gas* to {*natural*} {*gas*} and searched for the new terms separately. We did not conjoin {*pork + belly*} as a phrase but searched for {*pork*} *OR* {*belly*}.

	RELEVANT	NOT RELEVANT	
RETRIEVED	$A \cap B$	$\bar{A} \cap B$	B
NOT RETRIEVED	$A \cap \bar{B}$	$\bar{A} \cap \bar{B}$	$\bar{B}$
	A	$\bar{A}$	

Table 6.1: Table from [107] identifying the relevant and retrieved sets.

We input each topic set (query) to each system in turn and counted the number of correct matches and the number of false positive matches retrieved by each system in the top 15, top 20, top 30, top 40 and top 50 best candidate matches. We produced *recall* figures for each system evaluated, given in table 6.10 for the top 15, top 20, top 30, top 40 and top 50 documents retrieved. We totalled the number of correct matches retrieved (RELEVANT $\wedge$ RETRIEVED: $|A \cap B|$ from table 6.1) for all queries and divided this figure by the expected number of relevant documents for all queries (RELEVANT $|A|$) giving equation 6.1 where $|\ |$ is the modulus function.

$$RECALL = \frac{|A \cap B|}{|A|} \tag{6.1}$$

We also produced a *precision* figure for matching, given in table 6.11. We calculated precision as the number of correct matches retrieved (RELEVANT $\wedge$ RETRIEVED $|A \cap B|$) for all queries divided by the number of matches retrieved (RETRIEVED $|B|$) giving equation 6.2 for the precision.

$$PRECISION = \frac{|A \cap B|}{|B|} \tag{6.2}$$

If all matches are correct then the precision will be 1.0. All SMART system configurations and the two MinerTaur configurations with synonymy activated ('syn' and 'synStem' described below) return the top 15 to top 50 documents. For the third MinerTaur configuration, 'basic', we only retrieve the set of highest scoring documents, for example if there are 6 documents with the highest score we return the top 6. We only return the top X when there are X or more best matching documents. We found the sets of lower scoring documents were often too large and this approach does not provide finer-grained scoring differentiation so we only return the set of top matching documents. Documents are simply scored according to the number of query words present. SMART and our synonym system allow a finer-grained match due to their respective scoring systems which permit a higher degree of scoring differentiation. If all correct matches are retrieved by any configuration in the top X (where X ranges from 15 up to 50), we count the number of retrieved documents (RETRIEVED $|B|$) as the number of documents up to and including the lowest ranked correct matching document. For example if there are three matching documents assigned the topic set represented by the query and they are retrieved at positions 1, 2 and

4 in the top 15 matches then we count four documents as retrieved; we exclude the lower matches from the precision calculation when all correct matches are found. For all systems we calculate precision from the top X documents in all other cases.

The recall and precision figures for the Reuters-21578 Dataset [82] should be considered from a relative perspective, system against system and not from their absolute values. The topics attributed to each document are somewhat arbitrary, inconsistent and objective. For example, document 2,085 (from our numbering convention) contains the words *copper, gold, lead, silver* but has the topic {*earn*} ascribed. Whereas the other documents ascribed the topics {*copper, gold, lead, silver*} all contain the words and are in fact very similar in content to 2,085. Another example anomaly are documents 3,311 and 14,684 from our numbering scheme that have the topic {*oilseed*} ascribed but *oilseed* or *rapeseed* or similar are never mentioned in the documents. We also noted that some documents do not have topics ascribed so may be recalled and denoted as incorrect matches for a particular query, even though, of course, they may match the query if they had their topics assigned. There are also spelling errors in the dataset. For example, document 13,274, from our numbering, has the topic {*aluminium*} ascribed but the only occurrence of *aluminium* in the document is incorrectly spelled. We did not attempt to spell check the corpus due to the sheer size of the corpus and the number of proper names that would render spell checking intractable. These anomalies all serve to reduce the recall and precision values for all systems evaluated and we feel an absolute value should only be considered for a consistent topic allocation. We do note though, that all systems should be affected equally, so a relative comparison is still valid. Therefore we ask the reader to compare the relative recall and precision values of: our system with no synonyms (we call this variant 'basic' in our analyses, it is similar to a UNIX 'grep' best partial match); our system with synonymy incorporated, we denote 'syn' in our analyses; our system with stemming in conjunction with synonymy, we denote 'synStem' for our analyses and the SMARTv11.0 system configurations.

We evaluate our system with the synonym hierarchy traversal switched off ('basic'), with synonym traversal activated ('syn') and with the word stemming activated in conjunction with the synonym traversal ('synStem') to compare the effects on the *recall* and *precision* figures. We demonstrate the need for synonym matching to permit paraphrased documents to be retrieved by noting the higher recall and precision for the system with the synonym traversal activated. We also note the improved recall and precision when the word stemming is activated in conjunction with the synonym hierarchy allowing word stemming variants not specified in the query to be included in the retrieval along with their

synonyms, for example, if the query word is 'grain', the stemming variants are {'grain', 'grains'}.

For our evaluation, we clustered the average context vectors of the 2,192 most frequently occurring words in the Reuters-21578 database [82]; words that occur more than 100 times. This ensures that the context vectors generated for these words are fully averaged and unbiased. If the word seldom occurs then the average context vector can be biased towards the initial vector representations and is not a true mean. The more frequent the word, the closer the average context vector will be to the true mean. We also reduced the input size for the TreeGCS algorithm, as the underlying GCS algorithm is slow. It took approximately 220 hours to perform 9,000 epochs using the 2,192 average context vectors as inputs. Even though we can read in the data structure incrementally, sampling the most contemporary hierarchy each time the IR system is run, we need to allow the hierarchy to evolve sufficiently before it can be initially sampled. This however, means that it is not always possible for us to generate synonyms for query words. In our evaluation we generate synonyms where they are available but otherwise just have to rely on the actual query words. In chapter 7 we discuss possible adaptations and alternative approaches for our hierarchical thesaurus production to speed the training and evolution process. The hierarchy generated from the 2,192 words is too complex to illustrate but we illustrate three clusters below, (two complete clusters and a subset of a third cluster). The first two complete clusters demonstrate the general qualitative excellence of the clustering process we employ to generate the hierarchy. The final partial cluster demonstrates one of the main problems of the approach, the inability to differentiate word senses and the resultant clustering ambiguities produced. In chapter 7, we discuss possible adaptations to the clustering process aimed at overcoming such problems.

1. {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}
2. {January, February, March, April, May, June, July, August, September, October, November, December, Calendar, Fiscal, Chapter}
3. {Act, Block, Expire, Guarantee, Pay, Become, Operate, Remain, Serve, Post, Stand, Use, Yield, Advance, Design, Apply, Begin, Continue, Delay, Exercise, Test, Consider, Decide, Go, Resume, Be ...}

The first two clusters can be deemed excellent quality regardless of the analysis method used. The final partial cluster contains a mixture of words, many polysemic or representing multiple parts-of-speech. All senses or parts-of-speech are merged in to a single average context vector for the word and thus distort the average vector to misrepresent the word. If all senses or at the least all parts-of-speech could be separated and a vector produced for each, then the vectors would not be distorted and the clustering would be improved. We also

note that we have not separated any proper names and these are all included in the average context vector generation and synonym hierarchy production. We posit recommendations in the final chapter for possible solutions to this problem.

We elected to compare our MinerTaur system against the SMART system as we felt SMART is a benchmark system, extensively examined in the IR literature. It was documented initially in the late 1960's and has been developed and refined since. The source code for the SMART system is available for download from the World Wide Web and is thus well known. The SMART system has always placed highly in the TREC conference performance comparisons and is place consistently in the top 5 systems evaluated [102] and [103].

For the SMART system the user must write a specification file to overwrite the default specifications with their own requirements. We wrote a minimal specification file to preserve SMART default settings for most SMART parameters. For all evaluations, any reader familiar with the SMART system and the settings will know the default specifications we have used. They are listed in the documentation accompanying the system downloaded from [99]. The only settings we specified were for the document and query vectors, the vector matching process and whether word stemming should be activated. We also elected to index only words and proper nouns, which is an identical indexing approach to MinerTaur for the evaluated corpus. We hope that by maintaining the default values we have not biased the investigation and have used a known benchmark system. All configurations eliminate stop-words from the indexing and matching processes. We used the standard SMART stop-word list; we also used this stop-word list as the stop-word list for our system to maintain uniformity. We used the SMART system in interactive mode for all retrievals, entering each query word separated by a space and terminating the query with a '.'. All configurations of the SMART system used for our empirical evaluation are detailed below. We describe the SMART configuration using identical notation to the SMART documentation [99] and the published SMART papers for the vector formats.

1. 'SMARTnnn.nnn' We deactivated word stemming so all document vector terms are indexed exactly as the words appear in the text corpus. The documents and queries are converted to vectors represented by the nnn format and the inverted file index is constructed in the nnn format. nnn format vectors are:

   (a) **n**one - the term frequency (tf) component is calculated as:

   $$\text{new tf} = \text{current tf} \tag{6.3}$$

   (b) **n**one - the document weight is calculated as:

   $$\text{new weight} = \text{new tf} \tag{6.4}$$

(c) **n**one - there is normalisation:

$$\text{result weight} = \text{new weight} \tag{6.5}$$

All documents and queries are matched to the nnn format. This format essentially counts the number of times any query word occurs in each document and assigns the document score as the matching query word count. The documents are assigned integer scores. We included this configuration as a control mechanism to demonstrate the inadequacy of merely using a count of the number of matched words as a similarity metric for an IR system. The format scores comparatively poorly for both recall and precision.

2. 'SMARTatc.atc' Again we deactivated word stemming for this configuration. The documents and queries are converted to vectors represented by the atc format and the inverted file index is constructed in the atc format. Atc format vectors are:

   (a) **a**ugmented normal form - the tf component is calculated as:

   $$\text{new tf} = 0.5 + 0.5 * \frac{\text{tf}}{max(\text{tf in vector})} \tag{6.6}$$

   (b) **t**fidf - the document weight is calculated as:

   $$\text{new weight} = \text{new tf} * log\frac{\text{num docs}}{\text{tf in entire corpus}} \tag{6.7}$$

   (c) **c**osine - the entire vector is normalised:

   $$\text{result weight} = \frac{\text{new weight}}{\sqrt{\sum_{i=1}^{n}(\text{new weights}^2)}} \tag{6.8}$$

   Given a new query, SMART converts it to an atc vector, and then uses a cosine vector similarity measure to compare it to the documents in the vector space. The resultant similarity is scored between 0 and 1. We included this configuration in our empirical investigation as it is commonly used and quoted in the literature.

3. 'SMARTatc.atcStem' - this format is similar to SMARTatc.atc but SMART word stemming is activated. In SMART stemming, all corpus words are stemmed prior to indexing and only the stems are indexed. The query is also stemmed and matching performed according to matching word stems. This contrasts to our stemming approach where the system suggests to the user a list of stemming variants generated from a stem initially input by the user. The user may then select which stemming variants they wish to use in the matching procedure. We hope this user selection will eliminate such anomalies as '*trainers*' being stemmed to '*train*' and the user presented with train timetables rather than sports shop links as occurs in unconstrained stemming.

4. SMARTatc.ntcStem - this format is analogous to SMARTatc.atcSTEM but the queries are converted to the ntc vector format. Both the documents and the inverted file list remain as atc vectors.

   (a) **n**one - the term frequency (tf) component is calculated as:

   $$\text{new tf} = \text{current tf} \tag{6.9}$$

   (b) **t**fidf - the document weight is calculated as:

   $$\text{new weight} = \text{new tf} * log\frac{\text{num docs}}{\text{tf in entire corpus}} \tag{6.10}$$

   (c) **c**osine - the entire vector is normalised:

   $$\text{result weight} = \frac{\text{new weight}}{\sqrt{\sum_{i=1}^{n}(\text{new weights}^2)}} \tag{6.11}$$

   Salton & Buckley [93] recommend this format in their paper. For the document vectors they recommend using the augmented normalisation for documents comprising technical vocabularies and meaningful terms such as the Reuters' dataset, they recommend the *tfidf* weighting format and using the cosine normalisation when the documents vary in length as they do in the Reuters' dataset. For the queries they recommend using the word frequency count to provide greater discrimination among query terms, the *tfidf* weighting format and any normalisation as this is unimportant for overall performance.

5. SMARTlnc.ltcSTEM - this vector format was used in the SMART system for the TREC 2 evaluation [12] where the SMART system was ranked fourth of the systems evaluated [102]. The document vectors and inverted file are stored in the lnc variant and the queries are converted to the ltc variant. Stemming is applied to all document words and all query words.

   (a) **l**og - the term frequency (tf) component is calculated as:

   $$\text{new tf} = \ln(\text{tf}) + 1.0 \tag{6.12}$$

   (b) **n**one - the document weight is calculated as:

   $$\text{new weight} = \text{new tf} \tag{6.13}$$

   (c) **c**osine - the entire vector is normalised:

   $$\text{result weight} = \frac{\text{new weight}}{\sqrt{\sum_{i=1}^{n}(\text{new weights}^2)}} \tag{6.14}$$

   (a) **l**og - the term frequency (tf) component is calculated as:

   $$\text{new tf} = \ln(\text{tf}) + 1.0 \tag{6.15}$$

Meta-Topic	Matching Queries	Correctly Matching Documents
Agriculture	25	81
Finance	12	12
Metallurgy	11	27
Energy	8	14
Miscellaneous	10	21

Table 6.2: Table listing the meta-topic categories, the number of queries in each category and the number of documents assigned to the queries in the category.

(b) **t**fidf - the document weight is calculated as:

$$\text{new weight} = \text{new tf} * log\frac{\text{num docs}}{\text{tf in entire corpus}} \tag{6.16}$$

(c) **c**osine - the entire vector is normalised:

$$\text{result weight} = \frac{\text{new weight}}{\sqrt{\sum_{i=1}^{n}(\text{new weights}^2)}} \tag{6.17}$$

We use default settings for SMARTv11 in all other cases.

Query Analysis

We investigate the statistical correlations and thus the statistical validity of the 66 Reuters' topic queries by first grouping the queries according to the meta-topic of the set of topic words. We use a majority voting system to assign the meta-topic categories, i.e., the category is the meta-level topic represented by the majority of the topic words in the query. If there is no obvious majority we assign the query to the miscellaneous category. We evaluate the recall figures for each meta-topic category and for each system configuration by evaluating the top 15 documents retrieved by each system for all queries belonging to each category. We input each query in the meta-topic category to each system configuration in turn and retrieved the top 15 matches. We counted the number of correctly matching documents in the top 15 matches and calculated the recall for each system as the total correct matches retrieved for all queries in the category divided by the expected number of correct matches as assigned by the Reuters' dataset. We can pinpoint whether the queries in each meta-category are favouring or penalising any system. The meta-topics, the number of matching queries and the number of documents correctly matching those queries according to the Reuters' topic assignments are given in table 6.2.

We further investigate the statistical correlations of the Reuters' query set by categorising the queries according to the number of correctly matching documents for each query. We calculate the recall figures for each system configuration for each category of matching documents (1, 2 and 3 or more) using the

No. of Matching Documents	Queries in Set	Correctly Matching Documents
1	40	40
2	11	22
3 or more	15	99

Table 6.3: Table listing the categories, the number of queries in each category and the number of documents assigned to the queries in the category.

Number of Words in Query	Queries in Set	Correctly Matching Documents
2 or 3	11	34
4	17	46
5	16	43
6	13	21
7 or more	9	11

Table 6.4: Table listing the number of words in each query, the number of queries in each category and the number of documents assigned to the queries in the category.

top 15 matches returned by each system for each query input. We can identify if any system is unduly favoured or unduly disadvantaged by queries with a specific number of matching documents. The number of queries and the number of correct matching documents are tabulated in table 6.3.

Our final investigation of the statistical correlations of the Reuters' query set is by categorising the queries according to the number of words in each query. We calculate the recall figures for each system configuration for each category of query words (2 or 3, 4, 5, 6 and 7 or more) using the top 15 matches returned by each system for each query input. We can identify if any system is unduly favoured or unduly disadvantaged by queries with a specific number of query words. The number of queries and the number of correct matching documents are tabulated in table 6.4.

Spelling Analysis

We perform a series of retrievals using MinerTaur with spelling switched off to compare the effect on the *recall* and *precision* figures. We compared the effectiveness of our spell checker against comparative and benchmark spell checkers in chapter 4 and noted the superior recall of our integrated modular approach, superior to benchmark spell checkers such as MS Word 2k and equivalent to the highest performing alternative, Aspell. In this chapter, we effectively demonstrate the necessity of a spelling pre-processor module to identify query word spelling errors. We reveal the effect on recall and precision of not detecting and

Component	Information Capacity	Memory Usage (bytes)
Spelling CMM	48,766 spelling-word associations	3,025,892
Phonetic CMM	48,766 phonetic codes	780,868
Word-Document CMM	890,071 word-document associations	19,464,422

Table 6.5: Table listing the memory usage statistics for the modules in our system.

correcting spelling errors prior to the document retrieval phase. We repeat the query run of 66 topic queries and retrieving the top 15 candidate documents but with one word incorrectly spelt in each query. We studied each query and pre-selected the word we felt would be most likely misspelled by a user. We note that we are performing partial match for all queries so the query may not depend on the misspelt word. The misspelt word may not discriminate in the query; it may not be present in any of the set of best matches or may be present in all. The discriminatory power of the correctly spelled word with respect to the best matching documents influences the misspelt word's effect on the recall and precision figures. However, the importance of the spell checker should still be revealed across the 66 queries.

6.4 Results

6.4.1 Memory Usage

For our integrated, modular MinerTaur system we calculated the memory usage with the C++ 'sizeof()' function. The statistics are given in table 6.5.

We compare the file size for the word file and word-document association list file against the spelling (Hamming + n-gram) CMM and the word-document CMM to evaluate the memory requirement of the data structure against the data stored. An ideal data structure should have an equivalent or lower overhead without introducing any storage errors such as false positive matches. We could subdivide the vector sets and use multiple smaller CMMs for each system module, we could reduce individual CMM storage by using binning or we could use multiple bits set to compress the CMM size and thus the CMM's storage overhead (see chapter 4 for a discussion of the alternative representations). However, to maintain perfect retrieval with no false matches we use orthogonal vectors and single CMMs for each module (a dual CMM for the hybrid spell checker and a single word-document CMM) for the data capacity of the Reuters corpus. A larger corpus would necessitate subdivided or compacted data structures. The storage overhead of the SMART system may become intractable

Component	Memory Usage	File size
Spelling CMM	3,025,892	445,087
Word-document CMM	19,464,422	14,469,449

Table 6.6: Table listing the CMM sizes and the file sizes of the corresponding data.

Component	Information Capacity	Training Time (seconds)
Spelling CMM	48,766 spelling-word associations	25
Word-Document CMM	890,071 word-document associations	18
TreeGCS hierarchy	GCS + hierarchy and 2,192 words	87

Table 6.7: Table listing the training times for the modules in our system.

without modification for a very large dataset as the matrix of document vectors and the inverted file list of words become too large to read into memory as single structures. Finally we provide an indication of the file size for the stored TreeGCS hierarchy for a network of 1,000 GCS cells after 10,000 training epochs with 2,192 input vectors. This is a notional figure dependent on the complexity of the hierarchy dynamically imputed by the TreeGCS algorithm but is included as an indicative value. The size was 8,672,517 bytes for the GCS network and 16,861 bytes for our superimposed tree. The GCS file includes the parameters for each GCS cell (ID, winning count, square deviation, 630-D vector, number of connections, list of cells connected to). The tree file contains a representation of the cluster tree (parent-child and sibling-sibling links) and a list of the GCS cells in each cluster.

6.4.2 Timing Statistics

Training Time

In table 6.7 we detail the training times for each component of our integrated, modular MinerTaur system. We compare the total training time for MinerTaur versus SMART using the SMARTatc.atc configuration. The SMARTatc.atc training time includes the time to produce the nnn representation and convert all vectors to the atc format with no word stemming. The figures are listed in table 6.8. We note that the training time to infer the synonym hierarchy using the TreeGCS algorithm with 1,000 cells and 2,192 input vectors for 3,000 epochs is approximately 70 hours. This is a one-off process, run to generate a synonym hierarchy that is stored on disk and read in to the MinerTaur system during the training time (87 secs) in table 6.7. We discuss potential speed enhancements for the entire TreeGCS methodology in chapter 7.

System	Training Time (seconds)
MinerTaur	130
SMART	289

Table 6.8: Table listing the training times for our system compared to SMART.

Retrieval Time

We input a series of queries with 2, 4, 6, 8, 10 and 12 words in the queries and compared the retrieval times for our 'basic' system. We input each query ten times and calculated the average retrieval time over the 10 retrievals. We also activated and deactivated the synonym traversal, word stemming and spell checking to measure the affect on the overall retrieval time of each component. We note that not all words are present in the synonym hierarchy so we record the number of words that were present in the synonym column as this affects the number of times the synonym hierarchy is traversed and hence the retrieval time for the query. We also note that we indicate the number of misspelt words as again this impacts on the retrieval time. We perform the 6-word match with spelling activated and with one to six misspelt query words to note the affect of the spelling module exact match in proportion to the overall system retrieval. All retrieval figures are listed in table 6.9.

6.4.3 Recall and Precision

MinerTaur versus SMARTv11

We input the 66 topic queries we extracted from the Reuters corpus to the systems in turn and counted the correct number of documents returned (A) from table 6.1 and the number of false positives returned for each query in the top 15 to top 50 matches. These two figures added gives the total retrieved for each query (B) from table 6.1. If all correct matches were found inside the top X (where X ranges from 15 to 50) then we calculated the false positives as the number of incorrect matches between the first document and the last correct match. There are a total of 155 correct matching documents for all queries as identified by the Reuters' topic assignments for the top 15 matches, 161 for the top 20 and 164 for the other evaluations. We list the recall and precision figures in table 6.10 and table 6.11 respectively and include graphs for the recall and precision in figures 6.5 and 6.6 respectively for easy comparison.

Query Analysis

We identify any statistical correlations and biases in the queries by categorising the queries by meta-topic, the number of matching documents and the number of words in the query and calculating recall figures for the top 15 documents retrieved by each configuration of SMART and our MinerTaur system. We

Query words	Synonyms (Present)	Stemming	Spelling (Incorrect)	Retrieval Time (seconds)
2 word	N	N	N	0.225
4 word	N	N	N	0.422
6 word	N	N	N	0.621
8 word	N	N	N	0.823
10 word	N	N	N	1.035
12 word	N	N	N	1.277
2 word	Y(0)	N	N	0.227
4 word	Y(4)	N	N	23.363
6 word	Y(5)	N	N	35.328
8 word	Y(4)	N	N	47.676
10 word	Y(6)	N	N	59.445
12 word	Y(4)	N	N	77.020
2 word	N	Y	N	0.222
4 word	N	Y	N	0.445
6 word	N	Y	N	0.661
8 word	N	Y	N	0.875
10 word	N	Y	N	1.109
12 word	N	Y	N	1.360
6 word	N	N	Y(1)	0.775
6 word	N	N	Y(2)	0.917
6 word	N	N	Y(3)	1.006
6 word	N	N	Y(4)	1.136
6 word	N	N	Y(5)	1.432
6 word	N	N	Y(6)	1.579

Table 6.9: Table listing the retrieval times of the various modules of our system. The synonym column denotes whether synonym traversal was activated and how many words were present in the synonym hierarchy (i.e., how many times the hierarchy was traversed), the stemming column denotes whether stemming was activated and the spelling column denotes whether input words were misspelt and how many were misspelt.

System	15	20	30	40	50
SMARTnnn.nnn	0.245	0.292	0.402	0.494	0.506
SMARTatc.atc	0.413	0.447	0.518	0.579	0.646
SMARTatc.atcSTEM	0.452	0.484	0.567	0.622	0.652
SMARTatc.ntcSTEM	0.464	0.511	0.575	0.634	0.694
SMARTlnc.ltcSTEM	0.477	0.534	0.610	0.677	0.713
Basic	0.555				
Syn	0.665	0.667	0.683	0.738	0.762
SynStem	0.729	0.745	0.762	0.780	0.805

Table 6.10: Table listing the recall figures for the systems and their respective configurations. For 'basic' we only retrieved the single set of best partial matching documents so only a top 15 figure is given.

System	15	20	30	40	50
SMARTnnn.nnn	0.044	0.041	0.042	0.041	0.036
SMARTatc.atc	0.089	0.077	0.066	0.059	0.056
SMARTatc.atcSTEM	0.100	0.089	0.078	0.070	0.062
SMARTatc.ntcSTEM	0.110	0.101	0.085	0.075	0.069
SMARTlnc.ltcSTEM	0.111	0.102	0.086	0.078	0.071
Basic	0.340				
Syn	0.229	0.206	0.165	0.150	0.136
SynStem	0.282	0.255	0.215	0.186	0.167

Table 6.11: Table listing the precision figures for the systems and their respective configurations. For 'basic' we only retrieved the single set of best partial matching documents so only a top 15 figure is given.

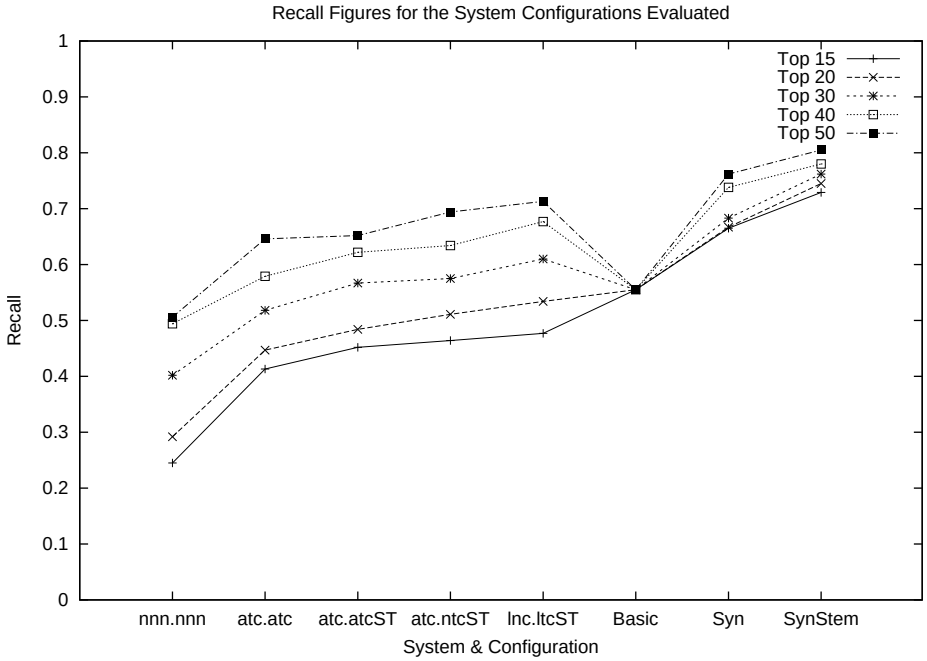

Figure 6.5: Graph illustrating the recall figures for all system configurations evaluated for the top 15 to top 50 matches retrieved. The recall figures are given in table 6.10.

can thus pinpoint whether any category is favouring or penalising any system configurations. We input each query to each system variant in turn. We counted the number of correct matches (as identified by the Reuters' dataset's topic assignments) in the top 15 documents. The recall graph for the queries grouped by meta-topic is given in figure 6.7, the graph for the recall figures for the queries grouped by the number of matching documents is given in figure 6.8 and the graph for the recall figures for the queries grouped by the number of query words is given in figure 6.9. We also include three further stacked column graphs in figures 6.10, 6.11 and 6.12 illustrating the contribution of each category to the total documents retrieved for each system configuration for the meta-topic categories, the number of matching document categories and the number of words in the query categories, respectively. The final column in each graph illustrates the maximum possible retrieval for each category as defined by the Reuters' topic assignments and forms a benchmark to illustrate the expected contributions of the categories to the total documents retrieved for each system configuration.

Spelling

We investigated the importance of our spell checker by noting the effect on the recall and precision figures with the spell checker activated and deactivated, given in table 6.12. We repeated the 66 topic queries used for our previous recall and precision investigations. We input each query to the system in turn with

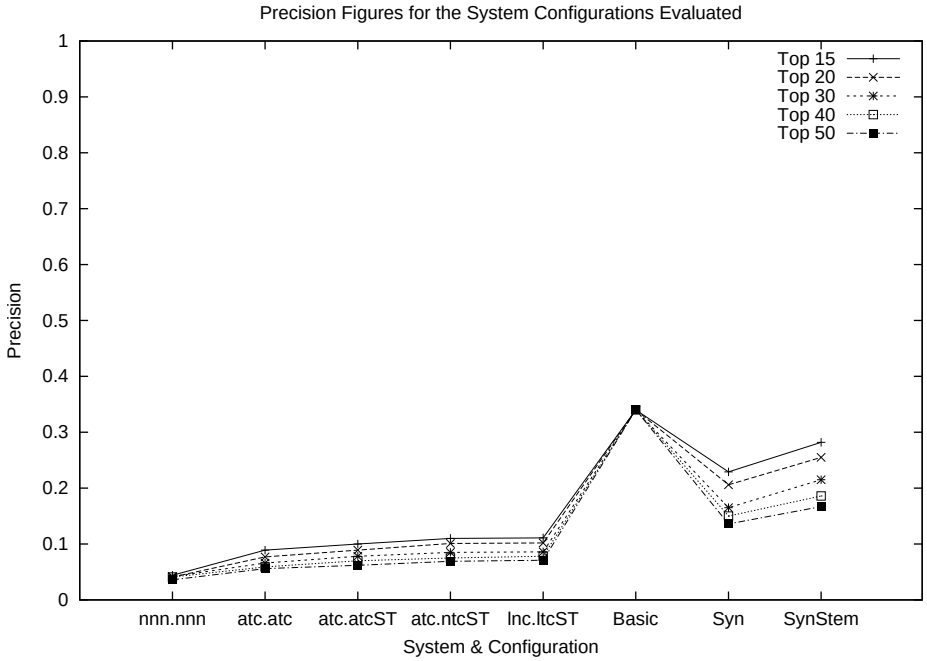

Figure 6.6: Graph illustrating the precision figures for all system configurations evaluated for the top 15 to top 50 matches retrieved. The precision figures are given in table 6.11.

System	Relevant $\|A\|$	False Matches	Total Retrieved $\|B\|$	Recall $\|A/155\|$	Precision $\|A/B\|$
SpellingOff	58	363	421	0.374	0.138
SpellingOn	86	171	253	0.555	0.340

Table 6.12: Table listing the recall and precision figures for our system with spelling activated and deactivated.

spelling activated and deactivated. We counted the number of correct matches and the number of false positives in the top 15 documents. If all correct matches are found inside the top 15 then we counted the number of false positives as the number of incorrect matches in between the first matching document and the last correct match as previously.

6.5 Analysis

6.5.1 Memory Usage

We demonstrated in chapter 5 that our word-document CMM compared favourably with an inverted file index and hash table representations for memory usage. We demonstrated in chapter 4 that our CMM lexicon representations compared favourably with other lexicon storage techniques with respect to their memory overheads. From table 6.6, the spelling CMM requires 6.8 times more memory

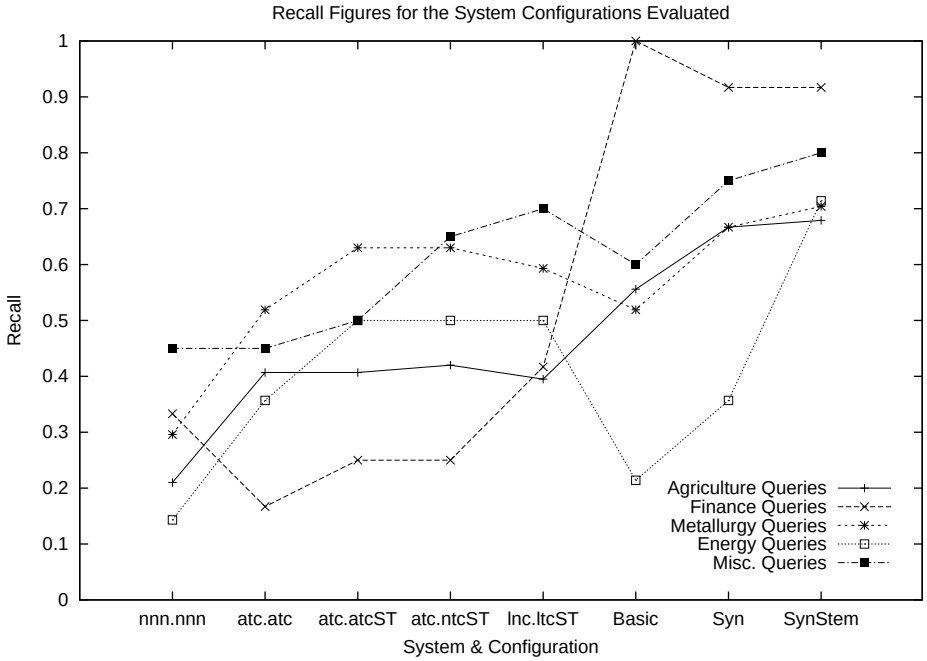

Figure 6.7: Graph illustrating the recall figures for all system configurations evaluated for the top 15 matches with queries categorised by meta-topic.

than the lexicon data file. This is mainly due to the overheads of the CMM data structure. The memory requirement for the word-document CMM and the corresponding data file are much closer, the CMM having 1.35 times the memory usage of its corresponding data file. The CMM structure overheads form a much lower proportion of the overall storage and are much less significant in the storage requirement. Even though CMMs are memory conserving, the data structure overheads when only a small data file is stored and the data file requires relatively many input rows in comparison to its size can exacerbate the memory expenditure. We can see from the file sizes that our superimposed hierarchy requires minimal storage with only a 16,861 byte file to represent the hierarchy. The GCS structure needs to store a 630-D vector for each cell so the storage overhead is higher with a corresponding file size of 8,672,517 bytes. Nevertheless, we feel this overhead is acceptable.

6.5.2 Training Time

From table 6.8, the SMART system takes 222% longer to train than MinerTaur from the prepared data. We should note that once both systems are trained, they could both perform multiple retrieval runs. SMART can be run repeatedly without recourse to retraining the document vectors and inverted index unless the system needs altering in some way such as a new database or a change of vector format. SMART writes the document vectors and inverted index to disk in a binary format that may be read rapidly into the system. In fact, once the document vectors and inverted index are generated, SMART is ready to query

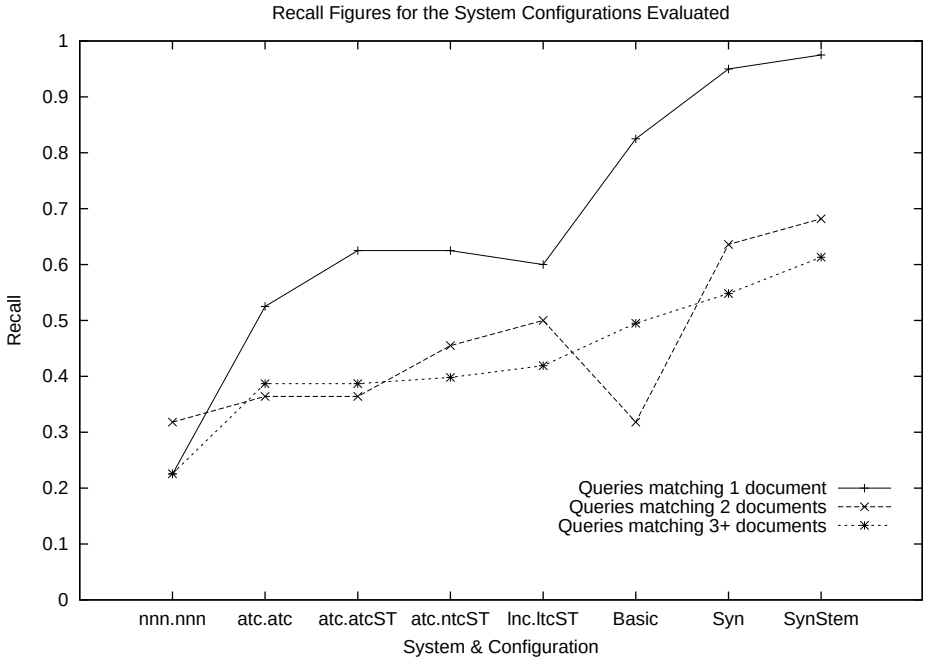

Figure 6.8: Graph illustrating the recall figures for all system configurations evaluated for the top 15 matches with queries categorised by the number of matching documents.

in less than 1 second. MinerTaur needs to be trained on each initiation taking 130 seconds; we intend the system to be left running and only retrained if any information such as the word-document association list changes. We have mentioned previously that we did not include the TreeGCS training in the training time for our MinerTaur system as we have not yet optimised this module and it may also be trained incrementally with the system sampling the most contemporary hierarchy each run-time. We feel MinerTaur is thus faster to train unless the system needs to be stopped and started repeatedly, requiring multiple training runs where SMART would be faster to train.

We have tried various approaches to speed the training of the algorithm including using the previous winner approach which is similar to a speedup exploited for SOMs [58], [55]. We store the identification of the winning cell for each input vector in an array. On the next epoch, if the previous winning cell (stored in the array) was closer to the input vector than any of its directly connected GCS neighbours then we set the new winner as the previous winning node. If the previous winner is not closer than any neighbour then we perform the usual traversal through the GCS topology to determine the winner. Again, we store all new winning cell identifiers in the array and repeat the process on each successive epoch. This assumes that if the previous winner is closer than its neighbours to the input vector then the previous winner is the closest node to the input vector. However, this had an adverse effect on the hierarchy quality

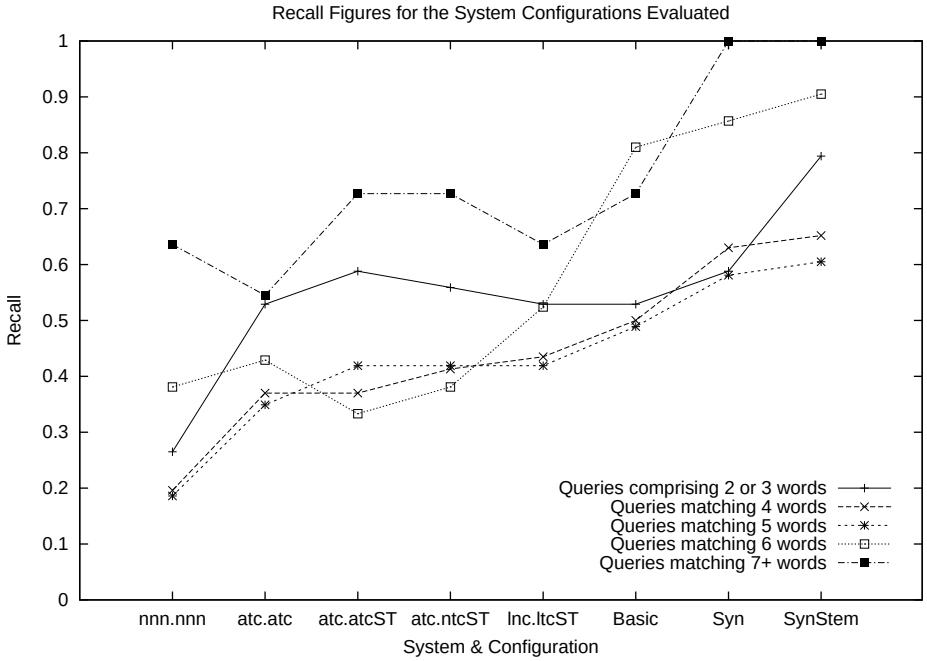

Figure 6.9: Graph illustrating the recall figures for all system configurations evaluated for the top 15 matches with queries categorised by the number of matching documents.

and the topology generated was very dissimilar to the standard approach. In the SOM there is a more regular topological configuration of neighbouring cells such as a square or hexagonal configuration that surrounds the central cell. In the GCS structure all neighbours may be situated in the vector space on one side of the previous winning node so the node may appear closer than the neighbours to the input vector but yet not be the closest node as another indirectly connected is closer. We also tried reducing the precision of the average context vectors processed by the TreeGCS algorithm from 'doubles' to 'floats' but this introduced only a negligible speedup. We attempted to convert the real-valued average context vectors to a binary representation but this lost too much accuracy. The hierarchical cluster topology generated by the binary vectors was affected unduly with little similarity between the real-valued cluster topology and the binary topology and also little similarity between the clusters produced, both in the number of clusters and their contents.

6.5.3 Retrieval Time

Examining table 6.9, our retrieval is $O(n)$ where n is the number of query words. For a simple query, the retrieval time is approximately $0.1 * \text{numberOfWords}$ seconds. We can see that the increase in retrieval time when stemming is added to a simple query is negligible. This is to be expected, as the algorithm we use for stemming is extremely rapid. The system only requires a single input to the lexicon CMM and the retrieval of all matched words by using the indexes

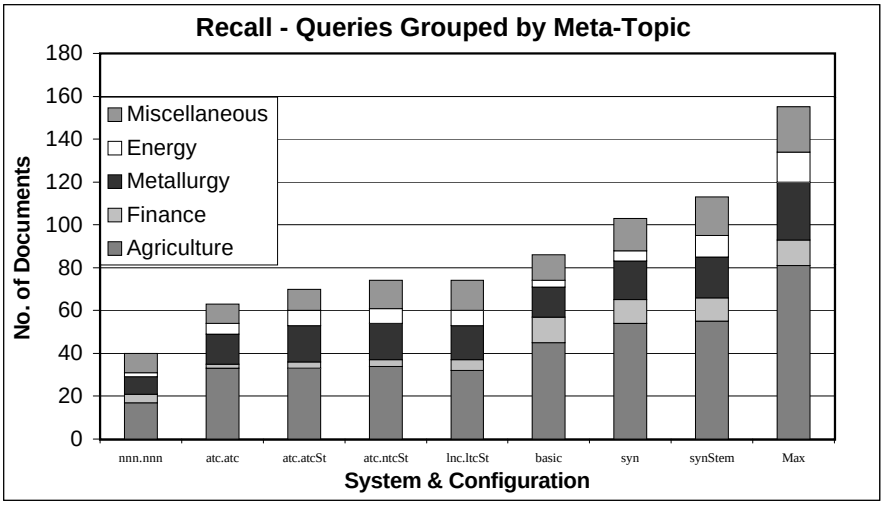

Figure 6.10: Stacked column graph illustrating the contribution of each query category for the top 15 matches with queries categorised by their respective meta-topics. The right-hand column represents the maximum number of documents associated with each query category as defined by the Reuters' topic assignments.

of the bits set in the thresholded output vector to retrieve the list of stemming variants. The spell checking retrieval adds a small proportion of additional time but the overhead, while more than the exact match required for stemming, is still acceptable. Our spell checking module is slowed by our shifting n-gram as we stated in chapter 4. This is slower than the conventional non-positional n-gram but we have demonstrated the higher recall of our shifting n-gram and also noted that the shifting n-gram integrates with the Hamming Distance whereas the conventional non-positional n-gram would require a separate CMM to function, increasing the system storage overhead. We therefore feel the advantages of the shifting n-gram mitigate the speed penalty. The number of matching words affects the spelling retrieval time. If there are many possible matches then retrieval is slowed more than for a misspelt word with few possible matches. There will be more bits set in the thresholded output vectors of the spelling module's two CMMs so more bit position indexes will have to be matched to determine the matched words. However, we feel that a retrieval time of less than 2 seconds per query is a suitable benchmark [74] and for a 6-word query with 6 misspelt words, retrieval was still below this benchmark.

The only really slow retrieval speed derives from the synonym hierarchy traversal. We feel the speed of retrieval is unacceptably slow for this method. As we

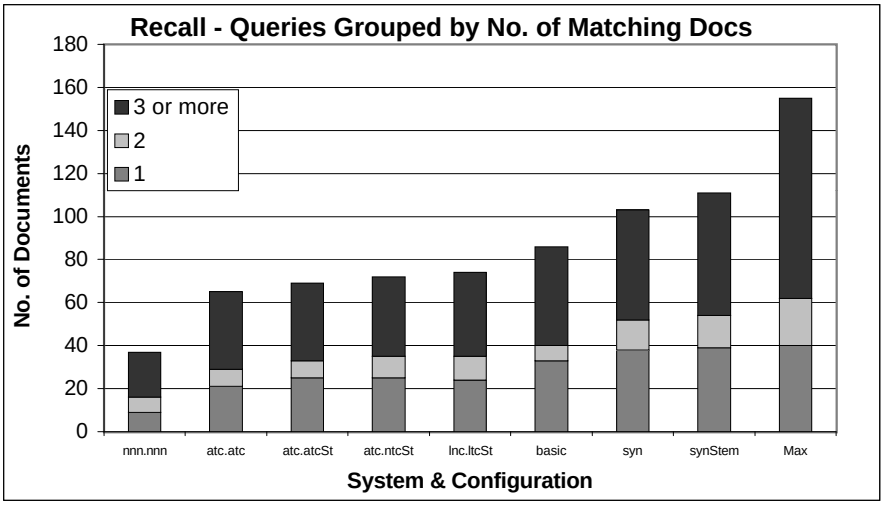

Figure 6.11: Stacked column graph illustrating the contribution of each query category for the top 15 matches with queries categorised by the number of matching documents. The right-hand column represents the maximum number of documents associated with each query category as defined by the Reuters' topic assignments.

have many criticisms of the speed of this module we have not yet optimised the retrieval. We can identify many possible speed optimisations that will improve retrieval speed. An obvious adjunct would be an array of pointers containing the alphabetically sorted array of all words in the synonym hierarchy. Each array word could point to the GCS cell or tree node that represents that word. At present we have to search to locate initially whether a word is present in the synonym hierarchy. If the word is present, we have to return the GCS cell that represents that word. We can then commence hierarchy traversal from that GCS cell. The array would eliminate the searching and hence speed the retrieval time. If two query words map to the same GCS cell or even the same GCS cluster, we can eliminate repetitious searches and use the scores from the previous search to speed the new traversal. If the new word maps to the same GCS cell, we can reproduce the scoring exactly. If the new word maps to the same cluster but not the same cell, we can calculate the scores for the word cluster but re-use the synonym hierarchy traversal scores as these are identical. We have already noted that we can superimpose the vectors for all words in each cluster, which would speed retrieval rather than serially inputting each vector. We noted in chapter 5 that CMMs were slower than comparable data structures for serial retrievals but much faster for parallel partial matching. By superimposing vectors, we can replicate parallel matching and thus speed the

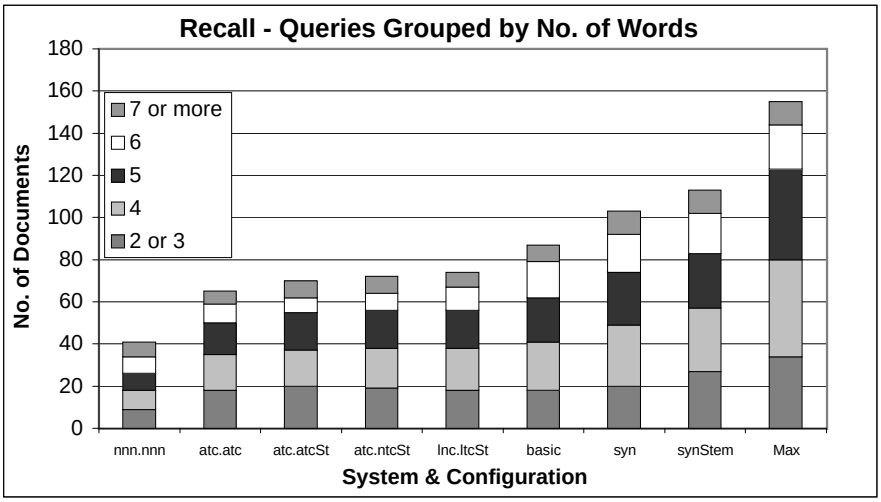

Figure 6.12: Stacked column graph illustrating the contribution of each query category for the top 15 matches with queries categorised by the number of words in the query. The right-hand column represents the maximum number of documents associated with each query category as defined by the Reuters' topic assignments.

retrieval process.

6.5.4 Recall and Precision

MinerTaur versus SMARTv11

The first point we would like to make is how the evaluation highlights the criticality of the vector format selection for document and query vectors in vector methods such as the SMART system. The vector configuration has a profound influence on the recall and precision figures and may be cited as a criticism of this technique as the user may not know in advance which configuration to select for their data instead having to rely on a heuristic evaluation to identify the optimal vector format.

From tables 6.10 and 6.11 and figures 6.5 and 6.6, we can see the higher recall and precision figures achieved by our MinerTaur system compared to the SMART configurations for 66 Reuter's queries. There is a significant improvement in the recall and precision figures for MinerTaur compared with all SMART variants. We can also observe the improvement in recall and precision when the synonym traversal 'syn' and when the synonym traversal in conjunction with the stemming module 'synStem' is added to the 'basic' system.

We only retrieved the single set of top matching documents for our 'basic' system so the recall and precision figure remains static across all evaluations. Often there are many documents in the second best set of matches. The 'basic' system cannot discriminate between these, as the scoring is not sufficiently fine-grained so we elected to just retrieve the single best matching set of documents. The evaluation was intended to demonstrate the expected recall and precision achieved by retrieving the set of documents representing the best partial match of the query words. We note that the 'basic' system has the highest precision figure of the systems evaluated. For most queries (61 of 66) this returned less than 15 documents in the set. All other systems were returning the top 15 and up to the top 50 matches except where all correct matches were found in which case the document up to and including the lowest ranked correct match were returned. This gives the 'basic' system a higher precision due to the smaller matching sets. The highest-ranking SMART variant, 'SMARTlnc.ltcSTEM', only surpasses the basic system's top 15 recall figure when the top 30 documents retrieved, double the number of documents retrieved for our 'basic' system.

Analysing the recall and precision graphs in figures 6.5 and 6.6 respectively, all system variants (excluding our 'basic' system) have an increasing recall and conversely a decreasing precision as the number of documents retrieved for each query increases from 15 to 50. This is exactly as we would expect as the higher number of retrievals increases the probability of retrieving more correct matches but paradoxically increases the number of false positives if the correct match is not found and thus decreases the precision figure. The highest ranked SMART variant, 'SMARTlnc.ltcSTEM', has a lower recall figure for the top 50 documents than our 'synStem' configuration achieves by retrieving only the top 15 documents. Our 'synStem' correctly identifies 113 of 155 correct matches in the top 15 with 'SMARTlnc.ltcSTEM' retrieving 117 of 164 documents in the top 50. For the precision, all variants of our MinerTaur system exceed the highest-ranking SMART precision figure for all evaluations. For the top 50 matches, our 'synStem' variant retrieves 789 documents of which 132 are correct matches. 'SMARTlnc.ltcSTEM' retrieves 1,654 documents of which 117 are correct matches for the top 50 documents retrieved per query. Our 'synStem' variant retrieves fewer documents in total for the top 50 matches (789 documents in total) than 'SMARTlnc.ltcSTEM' retrieves in total for the top 20 matches, retrieving 841 documents. We also note that adding global stemming to the 'SMARTatc.atc' configuration increased the recall and precision figures slightly by 0.039 and 0.011 respectively but adding user-selected stemming into our MinerTaur system induced a more marked increase in both recall and precision of 0.064 and 0.053 respectively. We feel our stemming approach of allowing the user to select from a list is more effective than global word stemming that can often introduce errors into the stemming process, such as stemming 'train-

ers' to 'train'. We note that we did not select the word stems with any bias towards the dataset, we attempted to select the stemming variants that we felt were most similar to the word stem, such as plurals.

Next we analyse just the top 15 matches and the queries where the MinerTaur or SMART systems failed to find a single correct match. For the 66 queries, there was 1 query when all SMART configurations found one correct match (of two correct matches) and all configurations of our system failed to find a single correct match. Conversely, there were 17 queries when 'synStem' found correct matches but 'SMARTlnc.ltcSTEM' failed, the best performing of the SMART configurations. The vast majority (14) of these queries had just one correct matching document as indicated by the Reuters' topic assignments. Paradoxically, figure 6.8 illustrates that 'SMARTlnc.ltcSTEM' has highest recall for queries matching 1 document so these queries are not disadvantaging the SMART system. Essentially, 'synStem' identified matches on 16 more queries than 'SMARTlnc.ltcSTEM'. There were 21 queries where 'synStem' found more correct matches then 'basic' and 2 queries where conversely 'basic' found more correct matches than 'synStem'. For the latter, the 'basic' system found the single correct matches for two queries where the addition of synonym traversal and stemming boosted the scores of other documents and caused the single correct match to fall out of the top 15 matches. 'SynStem' outperforms the 'basic' configuration by identifying correct matches when 'basic' fails to find any on 19 queries

Analysing the graphs in figures 6.8 and 6.11, for all systems with the exception of 'SMARTnnn.nnn' the queries matching 1 document (as assigned by the Reuters' dataset) induced the highest recall figure. For our 'synStem' configuration the recall was 0.975 as this configuration retrieved 39 of the 40 matching documents in the top 15 retrieved documents. We would expect the systems to have the highest recall figure when only a single document correctly matches the query as only one of the 15 documents retrieved needs to match, so the probability of a correct match and 100% recall in the top 15 matching documents is high. The 'SMARTatc.ntcSTEM' and 'SMARTlnc.ltcSTEM' variants, our 'syn' and our 'synStem' configurations all produce the second highest recall figure for the queries matching 2 documents and have their respective lowest recall figures for the queries matching 3 or more documents. Conversely, 'SMARTatc.atc', 'SMARTatc.atcSTEM' and our 'basic' variant all have their respective lowest recall figures for the queries matching 2 documents and their second highest recall figures for the queries matching 3 or more documents. We feel the recall graph indicates that the queries are not favouring any particular system when grouped by the number of matching documents, they affect the SMART variants and indicate the criticality of the vector configuration selected for the Reuters'

dataset. The graph does not highlight any statistical correlations with respect to the query groups. The only anomaly derives from the 'SMARTnnn.nnn' configuration that produces the lowest recall from the queries matching 1 document and the highest recall from the queries matching 2 documents. We included this configuration as a control mechanism to illustrate that merely counting matching words is not an apposite metric for document-query similarity estimation in an IR system and this is born out by the recall figures.

The graphs illustrating the recall figures for the respective system variants when the queries are categorised by meta-topic in figures 6.7 and 6.10 indicate that again there are no overall statistical correlations and system biasing for the queries. The agriculture queries comprise the largest category of queries and the plot for the agriculture recall in figure 6.7 essentially mirrors the overall recall plot for all systems in figure 6.5. The finance query plot indicates that queries falling into the finance category favour our system and in particular the 'basic' variant which has 100% recall. We feel this 100% match is due to the documents in the 'finance' category using a focussed vocabulary and tending to contain the subject words in the text of the document, for example {*money, dollar, yen, german, mark*} are all contained in the single document assigned this subject. MinerTaur matches on the percentage of the query words present so if all query words are present the document represents a high match. SMART matches on both the percentage present and the number of each word present so the frequency of each query word will affect the documents retrieved and hence may lower both recall and precision. Conversely, the energy and miscellany plots indicate that queries falling in to those two categories slightly favour the SMART systems as here the subject words tend to occur many times in the document but not all terms may be present. The recall figures for these categories are relatively higher compared to our system than the overall recall figures. We feel that these slight biases cancel each other in the overall recall and thus there are no overall statistical biases with respect to the query categories. We can also see from figure 6.7 that our 'synStem' system has the highest recall figure for all categories irrespective of any statistical favouritism for any specific categories.

The graphs illustrating the recall figures for the respective system configurations when the queries are categorised by the number of query terms in figures 6.9 and 6.12 indicate that again there are no overall statistical correlations and system biasing for the queries. All systems except the 'basic' variant have their highest recall percentages for the 'seven or more' category as we would expect as these queries are the most specific and narrow the focus of retrieval maximally, facilitating the correct recall. For the 'syn' and 'synStem' configurations of our system, the 6 word queries produced the next highest recall paradoxically the

2 or 3 word queries produced the second highest recall for all SMART configurations except 'SMARTnnn.nnn'. We stated in section 6.3.4 that we selected the short queries carefully to ensure specific topics and minimise the focus of retrieval to very specific words so we would expect the 2 or 3 word queries to have a high recall but probably not higher than the much more specific 6 word queries that produced the higher recall in our system variants. The 5-word category produced the lowest recall for all variants of our MinerTaur system and the 'SMARTlnc.ltcSTEM', 'SMARTnnn.nnn' and 'SMARTatc.atc' configuration. The 4-word category produced the second lowest recall for all systems except 'syn'. These 4 or 5 word queries are the least specific as the terms may be general or only m of the n query topics are matched where $m < n$ producing more candidate matches and hence less specific retrieval. We can also see from figure 6.9 that our 'synStem' system has the highest recall figure for all categories irrespective of any statistical favouritism for any system configurations across the categories.

We feel the top 15 recall figure of 0.729 and top 50 figure of 0.805 from table 6.10 for synonym traversal in conjunction with stemming ('synStem') in MinerTaur is commendable particularly with the inconsistencies and anomalies of the Reuters' topic assignments with respect to retrieval. The topics were assigned for a train and test classification task and as such probably have vagaries and objectivity deliberately included. A 73% success rate for retrieving correct match is very high particularly as this far exceeds the SMART system and our 'basic' system. We feel we have validated the accuracy of our implemented system and demonstrated the necessity of our synonym traversal and stemming modules. We also note that the synonym traversal we employed for this evaluation only clustered 2,192 frequently occurring words of the approximately 49,000 words in the documents. Therefore, we have demonstrated the necessity of such a module and can also surmise that a complete hierarchy, clustering all words in the corpus (minus non-essential words such as stop-words) would improve the recall further.

Spelling Analysis

Our spell checker detected and suggested the correct spelling to the user in the top 10 ranked spelling matches for all spelling mistakes investigated. The spell checker activated test run has identical recall and precision figures to our standard system 'basic' evaluated previously. The merit of the spell checking module is revealed by the lower recall and precision figures in table 6.12 produced when the spell checker was deactivated. Even though the mis-spelt word may not figure in the partial match for best matching documents and thus the word may not discriminate for this specific query, across a series of queries the importance of a spell-checking module is apparent. For the 66 queries, the spell checking

influenced the number of correct matches retrieved in 18 queries, i.e., when the spell checker was deactivated the system retrieved less correct matches in 18 queries. The spell checker influenced the number of false positives in 35 queries. For 34 queries the lack of spell checking increased the number of false positives. However, for one query the spell checker did actually decrease the number of false positives as the correctly spelt word was causing false positive matches and these were eliminated by the misspelling. However, such an eventuality is rare and we have shown from table 6.12 the importance of an accurate spell-checking module for the recall and precision of an IR system.

6.6 Conclusion

In this chapter, we have empirically demonstrated the superior training time for MinerTaur versus the SMART benchmark IR system. We denoted the training time as the time to learn the corpus vocabulary and word-document associations. We have empirically evaluated the retrieval times for the various modules comprising our system in chapters 4 and 5. The exact word match to validate query words against the stored lexicon in the Hamming Distance/n-gram CMM is very fast. The word-document association retrieval from the CMM is very fast. The combined time for these two stages is approximately 0.1 seconds per query word. It is very rare that a user employs more than 10 query words so query word validation and matching document retrieval will generally be less than 1 second on the hardware evaluated. This is below our threshold of 2-second retrieval [74]. Our query word stemming retrieval is also fast. The stemming uses the rapid Hamming Distance retrieval technique. The spelling retrieval employed if a query word is not verified by the exact match, is slightly slower due to the shifting n-gram retrieval which is O(length of query word), i.e., the number of shifts is proportional to the number of letters. However, the retrieval time is still acceptable, below 2 seconds when six of six query words are misspelt.

In this chapter, we have demonstrated the superior recall and precision displayed by our MinerTaur system compared to the benchmark SMART system. We also observed how recall was improved in MinerTaur by incorporating the synonym traversal module and was further improved by amalgamating the synonym traversal and user-selected stemming with the basic system. We have investigated the potential statistical correlations and biases in the query set but none were apparent. We have identified several areas that we feel contribute most to our MinerTaur system outperforming the SMART system for recall and precision. SMART uses a global stemming algorithm compared to the user-selected local stemming employed in MinerTaur. Global stemming is omnipotent but conversely can lead to incorrect stemming such as stemming 'glasses (spectacles)' to 'glass (vitreous substance)'. The user directs the stem-

ming in MinerTaur by initially supplying the stem in the list of query terms and then selecting their required stemming variants from a list of all possible stemming variants identified by MinerTaur for the particular stem supplied by the user. This ring-fences the stemming and prevents spurious errors such as those introduced by global stemming. MinerTaur has a synonym hierarchy available to pinpoint synonyms and hence include the synonym distances in the document scoring process. The SMART configurations evaluated in this chapter did not have synonymy incorporated. We should note that MinerTaur outperforms SMART for our recall and precision evaluations in this chapter even when synonymy is deactivated.

MinerTaur uses a binary word-document association representation. If a specific word is present in a particular document the association is denoted by a binary 1. The frequency of occurrence is not stored. The SMART configurations evaluated use a word frequency count or normalised word frequency count for the word-document association. SMARTnnn.nnn uses purely a count of the number of query words present in each document to score the documents. This distorts the scoring and produces poor recall and precision through favouring longer documents with more words present even though a shorter document may be a 'better' match for the user's criteria. In chapter 7, we discuss a possible extension of our word-document representation to encompass 0, 1 and 2 or more words present. If a word occurs twice in a document the document will receive double the score of a document where the word occurs only once. However, we limit this scoring to 2 or more words present to prevent the scoring distortions inherent in word frequency counting.

One potential adjunct to our recall and precision evaluation would be a larger query set. We could not glean any more queries from the Reuters' topics as we felt the remaining query sets were too similar to the queries already used in our evaluation and hence may favour certain systems. We wanted a good spread of topics with minimal similarity to evaluate all systems across all topics and to minimise partiality.

The one real weakness of the current system, which we have alluded to previously, lies with the TreeGCS hierarchy. The training time is very protracted for the large corpus vocabularies mandatory for an IR system. We have also not optimised the retrieval phase from the hierarchy due to the problem as we felt it unrealistic to speed the retrieval before we solve the more important training time problem. Even though we have identified the weakness with the training time for TreeGCS, we have identified the qualitative effectiveness of the TreeGCS algorithm against the SOM algorithm and also the quality of the TreeGCS system module with respect to system recall. In this chapter adding

synonym traversal increased the recall by 0.11 compared to the recall for the basic system. Ideally we need to speed the TreeGCS algorithm while maintaining the qualitative effectiveness of the cluster sets. Many techniques are faster than TreeGCS but at the expense of cluster quality. We need to find an algorithm with a minimal trade-off between speed and quality.

Chapter 7

Overall Conclusions and the Future

In this chapter we summarise our criteria for an ideal IR system, relate our evaluated MinerTaur system against our ideal technique, identify weaknesses in the current implementation described in this dissertation and posit some recommendations for possible improvements and expansions of the modules and integrated system.

In the introductory chapter and at the beginning of chapter 6 we outlined our necessary criteria for an ideal IR system

> "A methodology is desired to: process documents unsupervised and generate a multi-level and compact index using a data structure that is memory efficient, speedy, incremental and scalable; overcome spelling mistakes in the query; suggest alternative spellings for query terms; handle paraphrasing of documents and synonyms for both indexing and searching; to focus retrieval by progressively minimising the search space and finally calculate the document similarity from statistics autonomously derived from the text corpus. Documents may be retrieved according to the user's exact requirements by progressively refining the search and iteratively employing finer-grained and more specific matching techniques."

Our word context vector generation method is unsupervised, requiring only unstructured text as data, gathering statistical correlations from the corpus and inferring the word similarities purely from their contexts. We produce a multi-level (hierarchical) thesaurus that we can use to infer synonyms in the corpus using multi-level clusters of similar words and to thus handle paraphrasing of documents at query time. The hierarchical thesaurus represents the semantic relationships of the text corpus and permits the system to vary the degree of specificity of the search using both fine-grained and high-level categorisations of

the words as the user query is serviced. We showed in chapter 2 that TreeGCS coupled with our context vector generation process using a wider context window and high dimensionality word vectors produced high quality clusters - more similar to human generated vocabulary or Euclidean distance clusters of a dendrogram than the equivalent SOM vector generation approach or WEBSOM method.

In chapters 4 and 5 we demonstrated equivalent memory requirements for the spelling and word-document matrix modules of our system compared to corresponding techniques and benchmark data structures. In chapter 5 we empirically demonstrated that our word-document indexing structure was faster than comparable benchmark data structures for the partial match retrieval necessary in an IR system. We noted that CMMs are trained incrementally and are scalable with new word-document associations trained incrementally and overlaid with the existing associations allowing the CMM to train until saturation is reached when every word is linked to every document. We can also extend CMMs by copying an existing CMM into a new, larger CMM with additional rows or columns to permit new words and documents to be added. This is facilitated by our use of orthogonal vectors where each row represents a word and each column a document so we need only add a new row at the bottom or a new column on the left to extend the data structure to a new word or document respectively. The data structure does no have to be recompiled to assimilate new words or documents unlike may data structures used in IR systems such as the compressed word-document matrix used in LSI systems.

We empirically demonstrated the superior recall of our spell checker compared to analogous techniques and benchmark spell checkers in chapter 4. Our spell checker integrates elegantly with the architecture of the word-document matrix allowing us to replicate the vector representations of the lexicon words in both modules and thus speed retrieval, as no vector conversions are necessary. We demonstrated in the previous chapter the system recall improvement when the spell checker module is activated compared with the recall when the spell checker is not included in MinerTaur. For 66 queries, including the spell checker increased recall by 0.181. Even though we are performing partial matching in the IR system and the incorrectly spelled word may not be discriminating for the specific query, across a series of queries the benefit of the spell checker is validated. We also note that the word stemming capability we have incorporated into the spell checker further improves the recall figure. Rather than performing global stemming as SMARTv11.0, we allow the user to select from a series of stemming variants retrieved by the spell-checking module for a word stem input by the user. This eradicates the errors of global word stemmers which, for example, stem 'glasses (spectacles)' to 'glass (vitreous substance)', and our

technique includes only the query word stemming variants desired by the user. Incorporating the stemmer into our MinerTaur system increased the recall figure more than incorporating the global word stemmer into the SMART system. For the top 15 matches, the SMART global stemmer improved the SMART system recall by 0.039 (SMARTatc.atc versus SMARTatc.atcSTEM) compared with a 0.064 recall improvement when we incorporated our local stemming technique.

In the previous chapter, we demonstrated the superior recall and precision figures for our MinerTaur system compared to all configurations of the SMART system. We calculate document scores purely from statistics gathered from the text corpus. We employ both fine-grained and abstract matching techniques using the word relationships identified by the synonym hierarchy. The similarity of a document to the user query takes account of the spelling, word stemming and synonyms. Scores are awarded for each factor according to a diminishing return, the more similar a word to the query words, the higher the score awarded to that word and thus the higher the score propagated to any document containing that word.

The TreeGCS algorithm we developed from Fritzke's Growing Cell Structures [34], [36], [35], [37], [38] and described in chapter 2 is extremely slow to train and is also slow for retrieval. The latter problem is less significant as we have not optimised the retrieval due to the inherent problems with the slow training time. We have tried various approaches to speed the training of the algorithm but none have yet proved successful. We will continue to investigate further enhancements.

One possibility would be to use an alternative technique to cluster the average context vectors. We could use the AURA system to perform the clustering introducing the advantage of total synergistic modular integration; all system modules would be derived from an identical architecture. We could perform a k-nearest-neighbour matching process using the average context vectors converted to a binary vector format. AURA has been used successfully to implement the k-nearest neighbour algorithm. We have previously experimented with converting the context vectors to a binary format including adapting Zhou & Austin's [119] k-nearest-neighbour binning approach but the binary conversion process lost too much accuracy due to the high vector dimensionality. If we could successfully determine a suitable binary conversion function that maintains sufficient accuracy then we could exploit the speed of AURA. We may need to use higher dimensionality word vectors and thus higher dimensionality average context vectors compared to the 630-dimensional average context vectors used previously. AURA matching is a single step O(1) and not dependent on the dimensionality of the input or output vectors. Retrieval in TreeGCS is dependent on the

vector dimensionality. We needed to minimise dimensionality while ensuring maximal orthogonality of the vectors ascribed to the words to generate the average context vectors to ensure no similarity is imputed by the vectors ascribed. Increasing the dimensionality would introduce more accuracy for the binary method; our previous 630-dimensional binary conversions introduced too much error. We can use AURA for k-nearest neighbour by systematically reducing the threshold value for the output activation vector to retrieve the nearest vectors with respect to Euclidean distance; we do not need to specify the value of k in advance. We can either iteratively reduce the threshold until we have retrieved the k nearest neighbours or we can systematically reduce the threshold by x stages and retrieve any neighbours that lie within the threshold. We could also exploit the approach to generate a hierarchical representation by progressively determining the most similar clusters and merging them in an agglomerative clustering process analogous to a dendrogram. A successful implementation would introduce a massive speedup from months of training for the TreeGCS algorithm to seconds of real-valued to binary conversion and nearest neighbour matching in AURA. The only time penalty for the AURA k-nearest neighbour technique is the need to iteratively reduce the threshold and this time constraint is negligible in comparison to training TreeGCS.

We feel the average context vector generation methodology also needs improving and the approaches we discuss below may have an added bonus of speeding the TreeGCS clustering (or an alternative approach) by allowing us to subdivide the input space to allow parallel clustering on the subdivisions or to reduce the input space markedly by just clustering nouns. The average context vector generation and hierarchical clustering process differentiate many sets of synonyms. When we clustered the 2192 words for our system, we found the methodology had identified a cluster containing all and only the days of the week. A further cluster contained the months of the year along with the words 'calendar' and 'fiscal'. However, other words, particularly polysemic or those with multiple parts of speech, are not separated.

One possibility would be to include some form of part-of-speech (POS) tagging in the average context vector generation process prior to clustering to group the different parts of speech separately for greater differentiation, for example `supply<noun>` and `supply<verb>` would be separated by the tagging and an average context vector generated for each, which would then be clustered separately. For the current approach, all meanings of polysemic word are averaged together. POS tagging would introduce some separation although of course the different meaning but same part of speech word senses would still be clustered together, for example `bank<verb>` would be distinguished from `bank<noun>` but `bank<noun>` meaning *river bank* and `bank<noun>` meaning *financial institution*

would still be averaged together. The only way to distinguish the different senses would be to employ a human knowledge engineer to mark the different occurrences with separate tags to differentiate the word senses. Automated word sense disambiguation is very complex, and still in its infancy. The only viable methods are supervised and human implemented but these are human-intensive and time-consuming for both the knowledge engineer and the system so probably still intractable. If the corpus is restricted to a specific domain many word senses may not be pertinent anyway. We feel a POS tagger would be the preferred option as it is automated, POS tagging is tried and tested and highly accurate and any mistakes would probably be nullified in the context vector averaging procedure. We would just need a separate word identification vector (90-D vector) for each distinct POS tag for each unique word. We could also cluster each POS set separately which would also allow parallel clustering runs, speeding the clustering process greatly. We have noted that the clustering process needs speeding so any acceleration is vital. We could separate the average context vector sets with one set for each part of speech, this would also reduce the input space for each cluster run from one large input space to several smaller subsets. The smaller subsets would enable the cluster algorithm to run much faster i.e., the training would be much less time-consuming than a single large input space. One cluster algorithm process could run for each part of speech simultaneously providing a large speed increment.

Another option would be to just cluster average context vectors for nouns in the hierarchical thesaurus generation as in other systems, for example Yarowsky [118]. We would still use all words in the average context vector generation seven-word windows but vectors would only be produced for words tagged as nouns. This would eliminate some ambiguity and users generally search using nouns very rarely do they need to search using verbs. The POS tagger could isolate nouns in an automated and accurate tagging stage. This would also greatly reduce the quantity of average context vectors in the input space for the clustering algorithm, which is desirable as clustering process (discussed above) is very time-consuming and computationally intensive.

The hierarchical clustering process we have developed produces high quality cluster topologies as we demonstrated in chapter 2 and in the higher recall figure when the synonym hierarchy was incorporated into our MinerTaur system in chapter 6. The GCS algorithm that underlies TreeGCS is parameter sensitive. Our hierarchical construction process that overlies the GCS network is flexible and requires only the original GCS parameter settings; we have introduced no additional parameters with our hierarchical addition to GCS. However, selecting suitable parameters for GCS is a heuristic process. It is very time-consuming to implement a train and test parameter selection approach on a large vocabulary

(a large input space of context vectors as required for an IR system). TreeGCS needs to be run with each set of parameter settings for at least 10000 epochs to allow the hierarchy to evolve and settle. All hierarchies produced must then be assessed on completion to select the optimal parameter combination. We need a superior parameter setting approach. One possibility would be to exploit a *simple ensemble* and/or *bagging* methodology see [75]. During our evaluations, we have observed that altering the input data order and altering the GCS parameters produces different cluster topologies. We would run TreeGCS with different parameters for the simple ensemble approach and/or with different arrangements of training set for bagging to produce an ensemble of different hierarchies. We could then generate an average clustering from all hierarchies produced. However, the hierarchical thesaurus is a very complex data structure with many branches and child clusters so averaging may be difficult. The ensemble approaches are more aimed at classifiers than clustering algorithms as they use the average classifier vote to produce the ensemble. Generating an average vote would be more difficult for a clustering algorithm.

Another problem with TreeGCS is the difficulty of assessing the stopping point of the cluster topology evolution. When do we stop? One possibility is to determine when the number of clusters remains static for a specific predetermined number of epochs. However, there are two problems. The cluster number can remain static for many epochs and then commence changing again. A further problem is an oscillation problem where the same cluster is constantly deleted and reinstated so although stability has essentially been reached it is not apparent particularly in a large and complex cluster structure like a hierarchical thesaurus. One advantage of our current implementation is that we have written TreeGCS so we can cluster for a specific number of epochs, run the IR system using the cluster topology evolved, continue the clustering process and then run the IR system. We can cluster incrementally until we have a satisfactory cluster topology or we can be certain that stability has been reached.

Our current MinerTaur system makes no attempt to identify proper names in the text corpus. We converted the entire document collection to lower case and all company names were added to the average context vector generation process. We feel we need to isolate any proper names in a linguistic pre-processing phase similar to the pre-processing in the INQUERY [15] system. We can then remove the proper names from the average context vector generation and clustering process. We could even cluster them separately if we felt this was propitious thus reducing the input space for the clustering algorithm (by removing many context averages) and allowing the TreeGCS algorithm to run faster. We could replace all company names by 'companyName' and all personal names by 'personalName' in the text corpus for the average context vector generation process.

Tag Pattern	Example
A N	Allied Zurich
N N	Burger King
A A N	British American Tobacco
A N N	International Business Machines
N A N	South African Breweries
N N N	Shell Oil Company
N P N	Smith and Nephew

Table 7.1: POS combinations from [53] for detecting phrases and names.

These replacements would then be included in the average context vector generation for other words but no context vectors would be produced for proper names. Cavnar [16] postulated the use of capital letters to identify such names in a proposed extension to his IR system. This method works well but is dependent on the names being capitalised in the text corpus, i.e., it is reliant on the diligence of the author. However, it would be helpful to separate e.g., 'Shell' *oil company* from 'shell' *exoskeleton*. The Reuters 21578 dataset we have used for our evaluation in this dissertation has some meta-tags included to denote for example, proper names but they are not consistently applied. If each name is tagged once we can extrapolate and tag all occurrences but vitally each name needs to be tagged once and the name typed unchanged e.g., ABC Company is not changed to ABC Co. on some untagged occurrences later in the corpus, as these will be missed when we extrapolate the tags. If we were to implement the POS tagging discussed above, another option would be to modify Justeson and Katz's approach [53] for phrase identification to encompass name identification. They suggest that the POS combinations listed in table 7.1 identify collocations for phrases and we could use these to detect names. The identification would need to be supervised but could be conflated with a phrase identification step. Any possible phrases or proper names could be flagged to a knowledge engineer who could tag them appropriately.

Name identification is very problematic if for example '*shell*' is not initially capitalised due to a typographical error and is not listed as the phrase '*shell oil company*'. How do we know whether an occurrence of '*shell*' is '*the company*' or '*an exoskeleton*'? It would be intractable to achieve 100% reliability for name identification with such inconsistencies inherent in unstructured text. We would need computationally complex procedure using contextual information and these are rarely reliable. However, we should be able to achieve a sufficiently high reliability rate possibly up to 90% by using a hybrid procedure with a fast text scan to identify all capitalised names or meta-tagged names if available, extrapolating the tags to untagged occurrences of the proper names

and then employing a slower more computationally intensive and supervised phase using the POS tags and Justeson & Katz's phrase identification technique to identify any names missed by the initial capitalisation or tag-detection processes.

We could also use Justeson & Katz's [53] phrase identification technique to identify phrases in the corpus to allow phrase-based searching by users. As we mentioned in the previous paragraph, this stage could be amalgamated with the proper name identification in to a single supervised phase. Currently MinerTaur only permits term-based searches and we feel it would be beneficial to extend searching to phrases. We could identify phrases in the lexical preprocessing phase where we generate our context averages. We could assimilate the POS tags and the context vector generation phase to identify any words that regularly co-occur and thus indicate possible phrases. We could tabulate the word co-occurrences in a matrix when generating our average context vectors so co-occurrence statistics would be readily available. We would need to treat the phrases as single, conjoined units for the spell checker and also for the word-document association matrix. We would augment the existing term-based approach with the phrase identification facility rather than replacing any terms. We could use a separate CMM to store phrase-document associations. Bates [6] and Salton & Buckley [93] concur that single word indexing maximises IR accuracy so we feel the term-based index should be augmented with a phrase-based index rather than replacing any conjoined terms with their respective phrases in the index.

We feel another useful adjunct to the system would be to add Boolean querying capabilities such as '*AND*', '*OR*', '*NOT*' to the querying process. Our MinerTaur system described in this dissertation currently just performs partial '*AND*' match; identifying the best match according to the word scores. An '*OR*' or '*NOT*' capability is not currently included. As we use binary vectors for word-document matching a both '*OR*' and '*NOT*' capabilities will be straightforward to introduce. To implement '*NOT*' is merely a case of inverting vector bits. Boolean logic is considered esoteric and unfriendly for novice users but for experienced users it allows highly accurate matching. The experienced user can pinpoint their required information accurately minimising the false positive matches and ensuring the correct match is always retrieved.

We could extend querying even further to encompass SQL and possibly even NLP queries for novice users who may feel that Boolean logic or SQL is unfriendly, esoteric and unintuitive. Users could then select whether to query using Boolean logic for experienced users, SQL for users familiar with conventional database techniques or natural language querying for novice users. Many

natural language interfaces have been produced for IR systems (SMART version 11.0 [99] evaluated in this dissertation handles natural language queries) or many databases have natural language interfaces [95]. We could either incorporate off-the-shelf SQL and NLI packages or develop our own interface in the AURA system to seamlessly integrate with the existing IR modules and permit both SQL and natural language querying.

A further extension to the current system could be to amalgamate the CMM row saturation from the word-document matrix into the scoring process. The saturation is high for common words as many document bits are set (where each document is represented by a column) and the row saturation is low for infrequent words. This equates to the inverse document frequency (idf) of systems such as SMART, for example we could use the following equation to determine the word score analogous to the *idf* score,

$$wordScore = currentScore * (1 - rowSaturation) \tag{7.1}$$

High frequency words would score lower and low frequency words that differentiate between documents more according to the SMART idf approach would score more highly using the row saturation factor.

We infer from our evaluations that we may need a more complex frequency count for the word-document associations. This would help differentiate candidate matches more finely when ranking the matches for retrieval. Cavnar [16] suggested counting 0, 1 and 2 word associations per document separately as a word occurring twice in a particular document is twice as important as if it occurred just once. We can ignore higher frequency occurrences as that is irrelevant according to Cavnar and anyway just biases the system towards longer documents that have higher frequency word histograms. We could use different arity CMMs, providing a separate CMM for single-occurrence word-document associations and a CMM for multiple-occurrence word-document associations (where multiple denotes two or more). If the word were in the multiple-occurrence then it would score double than if in single-occurrence when calculating the document scores. The CMMs could be searched in parallel so retrieval would not be slowed in fact the retrieval speed may be increased as the retrieval time in CMMs is largely dependent on the number of matches. If the word-document associations were split into two, the number of matching documents retrieved from the CMMs would be approximately half the cardinality retrieved from a single CMM assuming a roughly equal distribution of associations between the two CMMs and thus this large slice of retrieval time could be halved. It is a simple process to amalgamate the outputs from two CMMs particularly as we are using orthogonal binary vectors to represent the documents. The vectors can simply be logically ORed to retrieve the matches.

In conclusion, we have described, compared and implemented a modular, integrated IR architecture. We have evaluated each individual module and the integrated system against comparative benchmark approaches identifying the strengths and any potential weaknesses. We have recommended possible extensions and improvements for the future development of the system.

Appendix A

Code Listing

A.1 Code Listing for the Data Structures Evaluated

All routines are adapted from the hash tables in [3].

A.1.1 Array of Lists

Used in word array, hash array and hash compact.

```
struct elem{char * name; elem *next;};
class list {
public:
   list(){pStart = NULL;}
   ~list();
private:
   elem *pStart;
};

void list::ListInsert(const char *s){
//insert a document ID into the list
   elem *p = new elem;
   elem *q = new elem;
   if (FindPosition(s) == NULL) {
       int len = strlen(s);
       p->name = new char[len + 1];
       strcpy(p->name, s);
       p->next = pStart;
       pStart = p;
   }
}
```

```
void list::writeList(const char * wordFile) {
//write to file entire list (all docs that match a particular word)
   FILE* F = fopen(wordFile, "w");
   elem *p = pStart;
   while (p){
       fprintf(F, "Doc is: %s ", p->name);
       p = p->next;
   }
   fprintf(F, "\n");
   fclose(F);
}
```

A.1.2 Inverted File List - Word Array

```
class StringHash {
   StringHash(unsigned len=1021): N(len) {
       a = new list[len];//array of lists of document IDs
   }

   void insert(const char *s, const char * doc) {
   //add a doc ID to a word's list - the location is found by hashing the word
       a[hash(s)].ListInsert(doc);
   }

   void writeDocsToFile(const char * wordFile, const char *s) {
   //write the documents associated with a particular word
       a[hash(s)].writeList(wordFile);
   }
private:
   unsigned N;
   list *a;
};

void StringHash::getWords(const char * wordFile)const{
//read in a file of words to initialise the array of words
   count = 0;
   FILE* F = fopen(wordFile, "r");
   while (!feof(F)) {
       fscanf(F, "%s", wordLabel);
       strcpy(wordArray[count++], wordLabel);
   }
   fclose(F);
```

```
}

unsigned StringHash::hash(const char *s)const{
//find the location of a word in the array (binary search)
   unsigned sum = 0;
   int middle;
   int left = 0;
   int right = N-1;
   while (right-left > 1) {
       middle = (right+left) / 2;
       (strcmp(s,wordArray[middle]) <= 0 ? right : left) = middle;
   }
   if ((strcmp(s,  wordArray[middle]) == 0)){
       sum = middle;
   }
   if ((strcmp(s,  wordArray[left]) == 0)){
       sum = left;
   }
   if ((strcmp(s,  wordArray[right]) == 0)){
       sum = right;
   }
   return sum;
}
```

A.1.3 Hash Table of Words : Length 20023

```
struct elem2 {char name[50];};
int collisionCounter;

class HashTable{
public:
   HashTable(unsigned len = 1021);
   elem2 *a;
};

HashTable::HashTable(unsigned len){
//initialise the hash table
   N = (len > 3 ? len : 3);
   a = new elem2[N];
   for (unsigned i=0; i < N; i++){
       a[i].name[0] = '\0';
   }
   collisionCounter = 0;
```

```
}

unsigned HashTable::hash(const char *s)
//Horner's hash function
{
   for (sum=0; *s; s++){
       sum = (sum*131 + *s);
   }
   return (sum % N);
}

int HashTable::h2(const char *t, unsigned &i)const{
//secondary hash function
   unsigned count = 0, incr;
   if (strcmp(a[i].name, t)) {
       incr = HashIncr();
       do {
           if (++count == N) return 0; // Failure
           i = (i + incr) % N;
       } while (strcmp(a[i].name, t));
   }
   return 1;  // Success
}

void HashTable::insert(const char *s){
//insert a word in to the hash table
   unsigned i = hash(s);
   if (!strcmp("", a[i].name) == 0)
       ++collisionCounter;
   if (!h2("", i)){
       cout << "Hash table full" << endl;
       exit(1);
   }
   strcpy(a[i].name, s);
}

unsigned HashTable::getPos(const char *s){
//return the position of a particular word
   unsigned i = hash(s);
   if (h2(s, i)) {
       return i;
   }
```

```
   else return 0;
}
```

A.1.4 Array of Lists

Used in both hash table: length 20023 and hash table compact: length 9491.

```
class StringHash {
public:

//Hash table implementation - length 20023
   StringHash(unsigned len = 1021) : N(len){
       a = new list[len];
       wordTable = new HashTable(len);
   }

//Hash table compact implementation len = 20023, hlen = 9491
   StringHash(unsigned len = 1021, unsigned hlen 1021) : N(len){
       a = new list[len];
       wordTable = new HashTable(len);
   }

   ~StringHash(){delete[] a;}

   void writeDocsToFile(const char * wordFile, const char *s){
//write all docs associated with a particular word
       a[hash(s)].writeList(wordFile);
   }

   void insert(const char *s, const char * doc){
//insert a docID in a particular word's list
       a[hash(s)].ListInsert(doc);
   }

private:
   unsigned N;
   list *a;
   HashTable *wordTable;
};

int StringHash::getAllCollisions(void){
//return the number of collisions
   return wordTable->getCollisions();
```

```
}

void StringHash::getWords(const char * wordFile)const{
//read in all words and insert them into the hash table
   count=0;
   FILE* F = fopen(wordFile, "r");
   while (!feof(F)) {
       fscanf(F, "%s", wordLabel);
       wordTable->insert(wordLabel);
   }
   fclose(F);
}

unsigned StringHash::hash(const char *s)const{
//return the position of a word in the hash table
   return wordTable->getPos(s);
}
```

A.1.5 Hash Table Compact

All routines are identical to the hash table except those given below.

```
struct elem2 {char name[50];int listPos;};
int collisionCounter;

class HashTable {
public:
   HashTable(unsigned len=1021);
   elem2 *a;
};

HashTable::HashTable(unsigned len) {
//initialise the hash table
   N = (len > 3 ? len : 3);
   a = new elem2[N];
   for (unsigned i=0; i < N; i++){
       a[i].name[0] = '\0';
   }
   collisionCounter=0;
   listNum = 0; //set list ID counter to 0
}


void HashTable::insert(const char *s) {
```

```
//insert a word in the hash table and add an integer to identify the
//word's list in the compact list array
   unsigned i = hash(s);
   if (!strcmp("", a[i].name) == 0){
       ++collisionCounter;
   }
   if (!h2("", i)){
       cout << "Hash table full" << endl;
       exit(1);
   }
   strcpy(a[i].name, s);
   a[i].listPos = listNum++;
//the location for the word's list in the list array
}

unsigned HashTable::getPos(const char *s){
//return the position of a particular word's list
//NB this is the list pos and not the word's position in the hash table
   unsigned i = hash(s);
   if (h2(s, i)) {
       return a[i].listPos;
   }
   else return 0;
}
```

Bibliography

[1] A. V. Aho and J. D. Ullman. Optimal Partial-Match Retrieval When Fields are Independently Specified. *ACM Transactions on Database Systems*, 4(2):168–179, 1979.

[2] S. Amari. Characteristics of sparsely encoded associative memory. *Neural Networks*, 2(6):451–457, 1989.

[3] L. Ammeraal. *Algorithms and Data Structures in C++*. John Wiley & Sons, Chichester, England, 1998.

[4] Aspell. Web page http://aspell.sourceforge.net/.

[5] J. Austin. Distributed associative memories for high speed symbolic reasoning. In R. Sun and F. Alexandre, editors, *IJCAI '95 Working Notes of Workshop on Connectionist-Symbolic Integration: From Unified to Hybrid Approaches*, pages 87–93, Montreal, Quebec, Aug. 1995.

[6] M. J. Bates. Indexing and Access for Digital Libraries and the Internet: Human, Database, and Domain Factors. *Journal of the American Society for Information Science*, 49(13):1185–1205, Nov. 1998.

[7] R. Beale and T. Jackson. *Neural Computing: An Introduction*. Institute of Physics Publishing, Bristol, U.K. and Philadelphia, PA, 1990.

[8] R. Belew. A Connectionist Approach to Conceptual Information Retrieval. In *Proceedings of the International Conference on Artificial Intelligence and Law*, 1987.

[9] C. M. Bishop. *Neural networks for pattern recognition*. Oxford, Clarendon P., 1995.

[10] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *7th International World Wide Web Conference*, 1998.

[11] J. Broglio, J. Callan, and B. Croft. An Overview of the INQUERY System as Used for the TIPSTER Project. Technical Report UM-CS-1993-085, University of Massachusetts, Amherst, Computer Science, 1993.

[12] C. Buckley and J. Walz. SMART in TREC 8. In *Proceedings of the Eighth Text Retrieval Conference (TREC 8)*. NIST Special Publication, 1999.

[13] W. R. Caid, S. T. Dumais, and S. I. Gallant. Learned vector-space models for document retrieval. *Information Processing and Management*, 31(3):419–429, 1995.

[14] J. Callan and W. Croft. An Evaluation of Query Processing Strategies using the TIPSTER collection. In *Proceedings of the 16th ACM SIGIR International Conference on Research and Development in Information Rerieval*, pages 347–356. ACM, 1993.

[15] J. Callan, W. Croft, and S. Harding. The INQUERY Retrieval System. In *Proceedings of the 3rd International Conference on Database and Expert Systems Applications*, pages 78–83, 1992.

[16] W. B. Cavnar. N-Gram-Based Text Filtering for TREC-2. In D. K. Harman, editor, *Proceedings of the 2nd Text REtrieval Conference (TREC-2)*, pages 171–179, NIST, 1994.

[17] W. B. Cavnar. Using an N-Gram-Based Document Representation with a Vector Processing Retrieval Model. In D. K. Harman, editor, *Proceedings of the Third Text REtrieval Conference (TREC-3)* , Gaithersburg, Maryland, 1994.

[18] W. B. Cavnar and A. J. Vayda. Using Superimposed Coding of N-Gram Lists for Efficient Inexact Matching. In *Proceedings of the 5th USPS Advanced Technology Conference*, pages 253–267, Washington, DC, 1992.

[19] C.-H. Chen and V. Honavar. A Neural Network Architecture for High-Speed Database Query Processing. In Dale, Moisl, and Somers, editors, *Handbook of Natural Language Processing*. Marcel Dekker, New York, 1999.

[20] H. Chen. Machine Learning for Information Retrieval: Neural Networks, Symbolic Learning, and Genetic Algorithms. *Journal of the American Society for Information Science*, 46(3):194–216, Apr. 1995.

[21] H. Chen, C. Schuffels, and R. Orwig. Internet Categorization and Search: A Machine Learning Approach. *Journal of Visual Communication and Image Representation, Special Issue on Digital Libraries*, 7(1):88–102, 1996.

[22] I. Cheng and R. Wilensky. An Experiment in Enhancing Information Access by Natural Language Processing. Technical Report CSD-97-963, University of California, Berkeley, Division of Computer Science, University of California, Berkeley, CA, 1997.

[23] V. Cherkassky, N. Vassilas, G. Brodt, and H. Wechsler. Conventional and Associative Memory Approaches to Automatic Spelling Correction. *Engineering Applications of Artificial Intelligence*, 5(3):223–237, 1992.

[24] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[25] F. Crestani. A Model for Adaptive Information Retrieval . *Journal of Intelligent Information Systems*, 1997.

[26] D. Cutting, D. Karger, J. Pedersen, and J. W. Tukey. Scatter/Gather: A Cluster-based Approach to Browsing Large Document Collections. In *Proceedings of the 15th Annual International ACM/SIGIR Conference*, Copenhagen, 1992.

[27] I. Dagan, L. Lillian, and F. C. N. Pereira. Similarity-Based Methods for Word Sense Disambiguation. In *35th Annual Meeting of the Association for Computational Linguistics*, San Francisco, California, 1997. Morgan Kaufmann.

[28] M. Damashek. Gauging Similarities with n-grams: Language-Independent Categorization of Text. *Science*, 267(10):843–848, 1995.

[29] F. Damerau. A technique for Computer Detection and Correction of Spelling Errors. *Communications of the ACM*, 7(3):171–176, 1964.

[30] S. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by Latent Semantic Analysis. *Journal of the Society for Information Science*, 1(6):391–407, 1990.

[31] S. Dumais, J. Platt, D. Heckerman, and M. Sahami. Inductive Learning Algorithms and Representations for Text Categorization. In *CIKM-98: Proceedings of the 7th International Conference on Information and Knowledge Management*, page to appear, 1998.

[32] Elcom Ltd - Password Recovery Software. Web page http://www.elcomsoft.com/prs.html.

[33] B. S. Everitt. *Cluster Analysis*. Edward Arnold, 1993. 3rd Edition.

[34] B. Fritzke. Unsupervised Clustering With Growing Cell Structures. In *Proceedings of the IJCNN-91*, Seattle, WA, 1991.

[35] B. Fritzke. Growing Cell Structures - a Self-organizing Network for Unsupervised and Supervised Learning. Technical Report TR-93-026, International Computer Science Institute, Berkeley, CA, 1993.

[36] B. Fritzke. Growing Cell Structures - a Self-organizing Network in k Dimensions. In I. Aleksander and J. Taylor, editors, *Artificial Neural Networks II*, pages 1051–1056. North-Holland, Amsterdam, 1993.

[37] B. Fritzke. Kohonen Feature Maps and Growing Cell Structures - a Performance Comparison. In C. Giles, S. Hanson, and J. Cowan, editors, *Advances in Neural Information Processing Systems - 5 -(NIPS*92)*. Morgan Kaufmann, San Mateo, CA, 1993.

[38] B. Fritzke. Growing Self-organizing Networks - Why? In M. Verleysen, editor, *ESANN'96: European Symposium on Artificial Neural Networks*, pages 61–72. D-Facto Publishers, Brussels, 1996. Invited paper.

[39] T. Gadd. 'Fisching for Werds'. Phonetic Retrieval of written text in Information Retrieval Systems. *Program*, 22(3):222–237, 1988.

[40] T. Gadd. PHONIX: The Algorithm. *Program*, 24(4):363–366, 1990.

[41] W. Gale, K. Church, and D. Yarowsky. Discrimination Decisions for 100,000-dimensional spaces. In A. Zampolli, N. Calzolari, and M. Palmer, editors, *Current Issues in Computational Linguistics: Essays in Honour of Don Walker*, pages 429–450. Giardini Editori e Stampatori and Kluwer Academic Publishers, Pisa and Dordrecht, 1994.

[42] T. D. Gedeon, R. A. Bustos, B. J. Briedis, G. Greenleaf, and A. Mowbray. Word-Concept Clusters in a Legal Document Collection. In B. Reusch, editor, *Computational Intelligence - Theory and Applications*, Lecture Notes in Computer Science, 1226. Springer, 1997.

[43] M. Goldszmidt and M. Sahami. A Probabilistic Approach to Full-Text Document Clustering. Technical Report ITAD-433-MS-98-044, SRI International, 1998.

[44] L. D. Higgins and F. J. Smith. Disc Access Algorithms. *Comp. Journal*, 14(3):249–253, Aug. 1971.

[45] T. Hofmann. Learning and Representing Topic. A Hierarchical Mixture Model for Word Occurrences in Document Databases . In *Conference for Automated Learning and Discovery, Workshop on Learning from Text and the Web, CMU 1998 (accepted for presentation)*, 1998.

[46] T. Honkela. *Self-Organizing Maps in Natural Language Processing.* PhD thesis, Helsinki University of Technology, Neural Networks Research Centre, PO Box 2200, FIN-02015, FINLAND, Dec. 1997.

[47] T. Honkela, S. Kaski, K. Lagus, and T. Kohonen. Newsgroup Exploration with WEBSOM Method and Browsing Interface. Technical Report Tech Report: A32, Helsinki University of Technology, Faculty of Information Technology, Espoo, Finland, 1996.

[48] T. Honkela, S. Kaski, K. Lagus, and T. Kohonen. WEBSOM–self-organizing maps of document collections. In *Proceedings of WSOM'97,*

Workshop on Self-Organizing Maps, Espoo, Finland, June 4-6, pages 310–315. Helsinki University of Technology, Neural Networks Research Centre, Espoo, Finland, 1997.

[49] T. Honkela, V. Pulkki, and T. Kohonen. Contextual Relations of Words in Grimm Tales, Analyzed by Self-Organizing Map. In F. Fogelman-Soulie and P. Gallinari, editors, *Proceedings of the International Conference on Artificial Neural Networks (ICANN-95)*, volume 2, pages 3–7. EC2 et Cie, Paris, 1995.

[50] S. Huffman and M. Damashek. Acquaintance: A Novel Vector-Space N-Gram Technique for Document Categorization. In D. K. Harman, editor, *Proceedings of TREC-3, 3rd Text Retrieval Conference*, pages 305–310, Gaithersburg, US, 1994. National Institute of Standards and Technology, Gaithersburg, US.

[51] J. J. Hull and S. N. Srihari. Experiments in Text Recognition with Binary-Grams and Viterbi Algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4(5):520–530, Sept. 1982.

[52] T. Joachims. Text Categorization with Support Vector Machines: Learning with Many Relevant Features. In *European Conference on Machine Learning (ECML)*, 1998.

[53] J. Justeson and S. Katz. Technical terminology: some linguistic properties and an algorithm for identification in text. *Natural Language Engineering*, 3(2):259–289, 1996.

[54] S. Kaski. Dimensionality reduction by random mapping: Fast similarity computation for clustering. In *Proceedings of IJCNN'98, International Joint Conference on Neural Network*, volume 1, pages 413–418, IEEE Service Center, Piscataway, NJ, 1998.

[55] S. Kaski. Fast Winner Search for SOM-Based Monitoring and Retrieval of High-Dimensional Data. In *Proceedings of IJCNN'98, International Joint Conference on Neural Network*, Edinburgh, UK, Sept. 1999.

[56] J. Kennedy. *The Design of a Scalable and Applications Independent Platform for Binary Neural Networks.* PhD thesis, Department of Computer Science, University of York, Heslington, York, UK. YO10 5DD, Dec. 1997.

[57] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, MA, 1968.

[58] T. Kohonen. *Self-Organizing Maps*, volume 2. Springer-Verlag, Heidelberg, 1997.

[59] T. Kohonen. Self-organization of very large document collections: State of the art. In L. Niklasson, M. Bodén, and T. Ziemke, editors, *Proceedings of ICANN98, the 8th International Conference on Artificial Neural Networks*, volume 1, pages 65–74. Springer, London, 1998.

[60] T. Kohonen, J. Hynninen, J. Kangas, and J. Laaksonen. Som_pak, the self-organizing map program package.

[61] D. Koller and M. Sahami. Hierarchically Classifying Documents Using Very Few Words. In *ICML-97: Proceedings of the Fourteenth International Conference on Machine Learning*, pages 170–178, San Francisco, CA, 1997. Morgan Kaufmann.

[62] M. Kubat, I. Bratko, and R. Michalski. A Review of Machine Learning Methods. In R. Michalski, I. Bratko, and M. Kubat, editors, *Machine Learning and Data Mining: Methods and Applications*, pages 3–69. John Wiley & Sons, Ltd, London, 1998.

[63] K. Kukich. Techniques for Automatically Correcting Words in Text. *ACM Computing Surveys*, 24(4):377–439, 1992.

[64] K. Lagus, T. Honkela, S. Kaski, and T. Kohonen. Self-Organizing Maps of Document Collections: A New Approach to Interactive Exploration. In E. Simoudis, J. Han, and U. Fayyad, editors, *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 238–243. AAAI Press, Menlo Park, California, 1996.

[65] K. Lagus and S. Kaski. Keyword selection method for characterizing text document maps. In *Proceedings of ICANN'99, Ninth International Conference on Artificial Neural Networks*, Edinburgh, UK, Sept. 1999.

[66] F. Lemke. Knowledge Extraction from Data Using Self-Organizing Modeling Techniques. In *eSEAM'97*, MacSciTech Organization, 1997.

[67] X. Li, S. Szpakowicz, and S. Matwin. A WordNet-based Algorithm for Word Sense Disambiguation. In *Proceedings of IJCAI-95*, Montréal, Canada, 1995.

[68] W. Lowe. Semantic Representation and Priming in a Self-organizing Lexicon. In *Proceedings of the 4th Neural Computation Psychology Workshop*, pages 227–239. Springer Verlag, 1997.

[69] S. A. Macskassy, A. Banerjee, B. D. Davison, and H. Hirsh. Human Performance on Clustering Web Pages: A Preliminary Study. In *The Fourth International Conference on Knowledge Discovery and Data Mining*, 1998.

[70] U. Manber and S. Wu. GLIMPSE: A Tool to Search Through Entire File Systems. In *1994 Winter USENIX Technical Conference*, 1994.

[71] D. Merkl. Exploration of Text Collections with Hierarchical Feature Maps. In *Int'l ACM SIGIR Conference on Research and Development in Information Retrieval,*, pages 186–195, Philadelphia. PA, July 1997.

[72] D. Merkl. Lessons Learned in Text Document Classification. In *Proceedings of WSOM'97, Workshop on Self-Organizing Maps, Espoo, Finland*, pages 316–321, Helsinki University of Technology, Neural Networks Research centre, Espoo, Finland., June 1997.

[73] R. Miikkulainen. Script-Based Inference and Memory Retrieval in Subsymbolic Story Processing. *Applied Intelligence*, 5:137–163, 1995.

[74] R. Miller. Response time in man-computer conversational transactions. In *Proceedings of Spring Joint Computer Conference, 33*, pages 267–277, Montvale, NJ, 1968. AFIPS Press.

[75] D. Opitz and R. Maclin. Popular Ensemble Methods: An Empirical Study. *Journal of Artificial Intelligence Research*, 11:169–198, 1999.

[76] Outpost9. Web page http://www.outpost9.com/files/wordlists.html.

[77] G. Palm. On Associative Memory. *Biological Cybernetics*, 36:19–31, 1980.

[78] F. Pereira, N. Tishby, and L. Lee. Distributional Clustering of English Words. In *Proceedings of ACL-93*, Columbus, Ohio, 1993.

[79] J. R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1(1):81–106, 1986.

[80] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1993.

[81] K. Ramamohanorao, J. W. Lloyd, and J. Thom. Partial Match Retrieval using Hahsing and Descriptors. *ACM Transactions on Database Systems*, 8(84):552–576, 1983.

[82] Reuters-21578. The Reuters-21578, Distribution 1.0 test collection is available from David D. Lewis' professional home page, currently: http://www.research.att.com/~lewis.

[83] E. M. Riseman and A. R. Hanson. A Contextual PostProcessing System for Error Correction Using Binary N-Grams. *IEEE Transactions on Computers*, 23:480–493, May 1974.

[84] H. Ritter and T. Kohonen. Self-Organizing Semantic Maps. *Biological Cybernetics*, 61:241–254, 1989.

[85] C. S. Roberts. Partial Match Retrieval via the Method of Superimposed Codes. *Proceedings of IEEE*, 67(12):1624–1642, 1979.

[86] H. J. Rogers and P. Willett. Searching for historical word forms in text databases using spelling-correction methods: Reverse error and phonetic coding methods. *Journal of Documentation*, 47(4):333–353, Dec. 1991.

[87] D. E. Rose. Appropriate Uses of Hybrid Systems. In D. Touretzky, J. Elman, T. Sejnowski, and G. Hinton, editors, *Connectionist Models: Proceedings of the 1990 Summer School*, pages 277–286. Morgan Kaufmann, Inc, 1990.

[88] D. E. Rose. *A Symbolic and Connectionist Approach to legal Information Retrieval.* Lawrence Earlbaum, Hillsdale, 1994.

[89] D. G. Roussinov and H. Chen. A Scalable Self-organizing Map Algorithm for Textual Classification: A Neural Network Approach to Thesaurus Generation. *Communication and Cognition – Artificial Intelligence*, 15(1-2):81–112, 1998.

[90] R. Sacks-Davis and R. A. K. A Two Level Superimposed Coding Scheme for Partial Match Retrieval. *Information Systems*, 8(4):273–280, 1983.

[91] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A Bayesian Approach to Filtering Junk E-Mail. In *Learning for Text Categorization: Papers from the 1998 Workshop.* AAAI Technical Report, No. WS-98-05, 1998.

[92] G. Salton. Interactive Information Retrieval. Technical Report TR69-40, Cornell University, Computer Science Department, Aug. 1969.

[93] G. Salton, J. Allan, and C. Buckley. Term-Weighting Approaches in Automatic Text Retrieval. *Information Processing & Management*, 24(5):513–523, 1988.

[94] G. Salton, J. Allan, and C. Buckley. Automatic Structuring and Retrieval of Large Text Files. *Communications of the ACM*, pages 97–108, Feb. 1994.

[95] B. Schneiderman. *Designing the User Interface: Strategies for Effective Computer Interaction.* Addison-Wesley, Reading, Massachusetts, 2nd edition, 1992.

[96] H. Schütze and J. O. Pederson. Information Retrieval Based on Word Senses. In *Fourth Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175, Las Vegas, NV, 1995.

[97] S. Scott and S. Matwin. Use of Lexical Knowledge in Text Classification. Technical Report TR-98-03, University of Ottawa, 150 Louis Pasteur, Ottowa, Ontario, K1N 6N5, Canada, 1998.

[98] R. Sedgewick. *Algorithms in C++.* Addison-Wesley, Reading, MA, 1992.

[99] v. s. c. SMART. ftp://ftp.cs.cornell.edu/pub/smart.

[100] A. F. Smeaton, R. O'Donnell, and F. Kelledy. Indexing Structures Derived from Syntax in TREC-3: System Description. In D. K. Harman, editor, *NIST Special Publication 500-226: Overview of the Third Text REtrieval Conference (TREC-3)* , pages 277–286, Gaithersburg, Maryland, Nov. 1994. Department of Commerce, National Institute of Standards and Technology.

[101] H.-H. Song and S.-W. Lee. A Self-Organizing Neural Tree for Large-Set Pattern Classification. *IEEE Transactions on Neural Networks*, 9(3):369–380, 1998.

[102] K. Sparck-Jones. Summary Performance Comparisons TREC-2 Through TREC-7. In *Proceedings of the Seventh Text Retrieval Conference (TREC 7) : Appendix B.* NIST Special Publication 500-242, 1999.

[103] K. Sparck-Jones. Summary Performance Comparisons TREC-2 Through TREC-8. In *Proceedings of the Eighth Text Retrieval Conference (TREC 8) : Appendix B.* NIST Special Publication, 1999.

[104] J. Stetina, S. Kurohashi, and M. Nagao. General Word Sense Disambiguation Method Based on a Full Sentential Context. In *Coling-ACL '98 Workshop "Usage of WordNet in Natural Language Processing Systems"*, Université de Montréal, Montréal, Canada, Aug. 1998.

[105] M. Turner and J. Austin. Matching Performance of Binary Correlation Matrix Memories. *Neural Networks*, 10(9):1637–1648, 1997.

[106] J. R. Ullman. A Binary n-Gram Technique for Automatic Correction of Substitution, Deletion, Insertion and Reversal Errors in Words. *Computer Journal*, 20(2):141–147, May 1977.

[107] C. J. Van Rijsbergen. *Information Retrieval.* Butterworths, London & Boston, 1979.

[108] R. Weiss, B. Vélez, M. A. Sheldon, C. Namprempre, P. Szilagyi, A. Duda, and D. A. Gifford. HyPursuit: A Hierarchical Network Search Engine that Exploits Content-Link Hypertext Clustering. In *Proceedings of the Seventh ACM Conference on Hypertext*, Washington, DC, Mar. 1996.

[109] S. Wermter, G. Arevian, and P. C. Recurrent Neural Network Learning for Text Routing. In *Proceedings of the International Conference on Artificial Neural Networks.*, Edinburgh, UK, Sept. 1999.

[110] S. Wermter, P. C., and G. Arevian. Hybrid Neural Plausibility Networks for News Agents. In *Proceedings of the National Conference on Artificial Intelligence*, Orlando, FL, July 1999.

[111] E. D. Wiener, J. Pedersen, and A. S. Weigend. A neural network approach to topic spotting. In *Proceedings of Fourth Annual Symposium on Document Analysis and Information Retrieval (SDAIR'95), Las Vegas, NV, April 24-26*, pages 317–332, 1995.

[112] D. J. Willshaw, O. P. Buneman, and H. C. Longet-Higgins. Non-Holographic Associative Memory. *Nature*, 222:960–962, 1969.

[113] . World FactBook. http://www.odci.gov/cia/publications/factbook/country-frame.html.

[114] S. Wu and U. Manber. AGREP - A Fast Approximate Pattern Matching Tool. In *Usenix Winter 1992 Technical Conference*, pages 153–162, San Francisco, CA, Jan. 1992.

[115] S. Wu and U. Manber. Fast Text Searching With Errors. *Communications of the ACM*, 35, Oct. 1992.

[116] Y. Yang, J. Carbonell, R. Brown, T. Pierce, B. Archibald, and X. Liu. Learning approaches for detecting and tracking news events. *IEEE Intelligent Systems*, 14(4):32–43, 1999.

[117] Y. Yang and X. Liu. A re-examination of text categorization methods. In *Proceedings of ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, 1999.

[118] D. Yarowsky. Unsupervised Word Sense Disambiguation Rivaling Supervised Methods. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics. Cambridge, MA,*, pages 189–196, 1995.

[119] P. Zhou and J. Austin. A Binary Correlation Matrix Memory k-NN Classifier. In *International Conference on Artificial Neural Networks*, Sweden, 1998.

[120] G. K. Zipf. *Human Behaviour and the Principle of Least Effort: an Introduction to Human Ecology.* Addison-Wesley, Cambridge, MA, 1949.

[121] J. Zobel and P. Dart. Phonetic String Matching: Lessons from Information Retrieval. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Zurich, Switz, 1996.