An Evaluation of Standard Retrieval Algorithms and a Binary Neural Approach

Victoria J. Hodge
Dept. of Computer Science,
University of York, UK

Jim Austin
Dept. of Computer Science,
University of York, UK

This research was supported by an EPSRC studentship.

Contact Address:
Victoria J. Hodge
Dept. of Computer Science,
University of York,
Heslington,
York,
UK
YO10 5DD
Tel: +44 1904 432729
Fax: +44 1904 432767
vicky@cs.york.ac.uk

Running Title: A Neural Retrieval Approach

An Evaluation of Standard Retrieval Algorithms and a Binary Neural Approach

Abstract

In this paper we evaluate a selection of data retrieval algorithms for storage efficiency, retrieval speed and partial matching capabilities using a large information retrieval dataset. We evaluate standard data structures, for example inverted file lists and hash tables but also a novel binary neural network that incorporates: single-epoch training, superimposed coding and associative matching in a binary matrix data structure. We identify the strengths and weaknesses of the approaches. From our evaluation, the novel neural network approach is superior with respect to training speed and partial match retrieval time. From the results we make recommendations for the appropriate usage of the novel neural approach.

Keywords: Information Retrieval Algorithm, Binary Neural Network, Correlation Matrix Memory, Word-Document Association, Partial Match, Storage Efficiency, Speed of Training, Speed of Retrieval

Many computational implementations require algorithms that are storage efficient, may be rapidly trained with data and allow fast retrieval of selected data. Information Retrieval (IR) requires the storage of massive sets of word to document associations. The underlying principle of most IR systems is to retrieve stored data on the basis of queries supplied by the user. 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 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 [11]: including inverted file lists, hash tables, document vectors, superimposed coding techniques and Latent Semantic Indexing (LSI) [8]. In this paper we compare 'Perfect' techniques, i.e. those that preserve all word to document associations as opposed to 'Imperfect' methodologies such as LSI.

A representation strategy used in many systems are document vectors. There are various adaptations of the underlying strategy but fundamentally the documents are represented by vectors with elements representing an attribute for each word in the corpus. 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 [12], 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 [9] 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 Singular Valued Decomposition [8] (SVD) - factor analysis. 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 paper). They are aimed at partial matching but tend to retrieve false positives (documents appear to match that should not) and are thus 'Imperfect'. There may be insufficient dimensions for uniqueness so many words may hash to the same bits and thus false matches will be retrieved. The extra matches then have to be rechecked for correctness and the false matches eliminated thus slowing retrieval. They are described in [10] and include: address generation hashing (see also [1]), and hashing with descriptors (see also [13]). There are also superimposed coding (SIC) techniques (described in [10]) for hash table partial matching applications but again these are 'Imperfect' and tend to over-retrieve, for example: one-level superimposed coding (see also [14]), and two-level superimposed coding (see also [15]).

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 recheck to eliminate false matches. Therefore we concentrate on 'Perfect' lexical techniques. We analyse an inverted file list (a slightly more sophisticated version of which is used in the Google search engine [6]), against hash tables using various hash functions, against binary associative memory: AURA [4]. We implement a novel binary matrix version of document vectors using AURA where the word-document associations are added incrementally and superimposed. Therefore training requires only a single-pass through the word-document association list. 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 [11] 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, that 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 [11]. In this paper, we focus on using AURA in 'Perfect' mode.

For our evaluation we use the Reuters 21578 Newswire text corpus as the dataset is a standard Information Retrieval benchmark. It is also large, consisting of 21578 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 documents that were repetitions. This left 18249 documents 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, i.e. 'The' becomes 'the' to ensure all instances match. 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 9491 remaining words including the six punctuation symbols " : ; & ( ) from the 18249 documents and derived 1230893 word-to-document associations. We created a file with the document

[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 1 (subsections 1.1, 1.2, 1.3 and 1.4) for details

ID (integers from 0 to 18248) and a list of each word that occurs in that document. The fraction of documents that contain each individual word are shown in the graph (see figure 1). The minimal fraction is 0.000055 (1 document) and the maximal fraction is 0.8138 (14850 documents). 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 paper, e.g., words not present or spelling errors.

1 Data structures

1.1 Inverted File List (IFL)

For the inverted file list compared in this paper, 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 2 and appendix A.2 for the C++ implementation). 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 $\Omega(1)$ 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 paper 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 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 words array and then the word-document associations in to their respective lists. The memory used = word array + 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.

1.2 Hash Table

The hash table employed in this paper 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 3 and appendix A.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 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 the paper degrades as the table fills and thus must be kept less than 50% occupied otherwise there are too many collisions degrading retrieval and insertion speed through the additional calculations that must be employed. There are 2 ways of handling collisions - collision rules; they are *Open Addressing or Chaining* and *Closed Hashing*.

- *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 paper. We must have a unique hash location for each word to indicate which documents belong to which of the words. We would need to store both the words and the documents 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 paper 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 [7]. 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, 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:

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

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

and Horner's rule given below in C code (adapted from Sedgewick [16]).

```
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 during insertion of 9491 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 also selected a prime number (20023) 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 ($hashTableSize - 2 = 20021$) 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 [7]). 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 [7]).

- *Double hashing* is used for our investigations. A far better rule is to compute a displacement directly from the key using another function of the key and add this displacement to the table position until a vacant position is found. The C code is given below and *sum* is the value returned from the Horner function.

  ```
  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).

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 the word-document associations in to their respective lists. The memory used = word array (50% full) + 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.

1.3 Two stage hashing - hash table compact

The previous hash table used a word array and list array that were both only 50% full thus wasting storage. For comparison, we use the data structures used in the previous hash table algorithm. However, we compact the array of lists so that all array locations are used (see figure 4 and appendix A.6 for the C++ implementation). The initial hash array of words stores both the word and an integer identifying the location of the word's 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 $\Omega(1)$ 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: 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 the word-document associations in to their respective lists. The memory used = word array (50% full) + 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.

1.4 AURA

AURA [4] is a collection of neural networks that may be implemented in a modular fashion and utilises Correlation Matrix Memories (CMMs) [4] to map inputs to outputs (see fig 5) through a learning rule, similar to a hash function. The AURA modular neural network uses binary weights and inputs. AURA implements the Hebbian learning rule, reinforcing active connections, to train the neural network. AURA does not suffer from the lengthy training problem of other neural networks; training is a one-pass (single epoch) process preserving the network's high speed. The technique is a supervised learning method as the outputs are known and guide the learning process. Storage is efficient in the CMMs as the matrix size is defined when the CMM is instantiated, new inputs do not require additional memory allocation as they are overlaid with existing trained patterns. AURA is able to partially match inputs. AURA has been used previously for symbolic, numeric and image data applications (see for example [5], [17] and [2]).

The words form the inputs and the documents the class patterns (outputs) of the neural network. The words and documents are translated to their respective binary bit vectors by the data-to-binary lexical token converter (see figure 5). Each word in the word list is represented by a unique m-dimensional orthogonal binary vector with a single bit set and 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 3). 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 and n equals the number of documents (see equation 3).

$$bitVector^p = p^{th}\text{bit set } \forall p \text{ for p = position\{words\} or p = position\{documents\}} \quad (3)$$

We require a 'Perfect' recall technique as stated in the 'Introduction'. Orthogonal vectors ensure that the CMM does not return false positives. If more than one bit is set in the input vectors or class patterns then we can get bit clashes and false positives will be returned from the lexical token converter as with the alternative hashing and superimposed coding approaches listed in the 'Introduction' and [11].

There are three alternative strategies for representing bit vectors and CMMs in AURA. They may be represented as binary bit vectors (BBVs), compact bit vectors (CBVs) or efficient bit vectors (EBVs). BBVs store all p bits in the bit vector, storing a 0 or 1 as appropriate for each position. CBVs store a list of the locations of the set bits in the bit vector, this is ideal for sparse bit vectors as only a few positions need to be stored in the list. EBVs enable a switch at a predefined weight (number of bits set). If the vector has a lower or equal weight then it is stored as a CBV but for a higher weight then it is stored as a BBV - this enables the most efficient storage implementation to be used for each individual vector. In this paper we use CBVs as only one bit is set in each word or document bit vector so this representation is the most storage efficient as only one position 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. A comparison of the memory usage of the CMM using binary CMMs, compact CMMs and efficient CMMs is provided in the results section (see section 3.1).

1.4.1 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. The diagram (figure 6) shows a CMM after 1, 2 and 3 patterns have been trained. The input patterns (words) are 01000000, 00100000 and 00001000 and their respective class patterns (documents) are 01000000, 00010000 and 00100000. The correlation matrix memory (CMM) is set to one where an input row (word bit) and an output column (document bit) are both set (see figure 6). The training process is $\Omega(n)$ as there is one association in the CMM per word-document pair. After storing all word-document associations, the CMM weights w_{kj} for row j column k where $\vee$ and $\wedge$ are logic 'or' and 'and' respectively is given by

$$w_{kj} = \bigvee^{\textit{all}\ \mathrm{i}} \textit{input}_j^i \wedge \textit{class}_k^i = \left\lceil \sum^{\textit{all}\ \mathrm{i}} \textit{input}_j^i \wedge \textit{class}_k^i \right\rfloor \tag{4}$$

1.4.2 Recalling from the network

For recall only the word pattern is applied to the network. The columns are summed

$$\textit{output}_j = \sum^{all\, i} \textit{input}_i \wedge w_{ji} \tag{5}$$

and the output of the network thresholded to produce a binary output vector (see figure 7). The vector represents the document trained into the network as matching the input word presented to the network for recall. We use the Willshaw threshold (see [4]) set to the number of bits in the input vector to threshold the output vector (see figure 7). Willshaw threshold sets to 1 all the values in the output vector greater than or equal to a predefined threshold value and sets the remainder to 0. We wish to retrieve all outputs that match all inputs. If the input has one bit set we retrieve all columns that sum to one.

If we wish to retrieve multiple word matches, rather than serially matching the bit vectors, AURA replicates parallel matching. The bit vectors for the required words are superimposed, forming a single input vector (see equation 6 and figure 8). If the input has two bits set, representing two words, then we retrieve all columns summing to 2, i.e., all documents matching both input words. Thus multiple word matching is $\Theta(1)$ with respect to input presentations as the inputs are superimposed and input as one vector. For the other data structures evaluated in this paper to retrieve documents matching N words requires N separate word input presentations with each word in order.

$$inputVector = \bigvee^{all i} inputVector^i \qquad (6)$$

If only a partial match of the input is required, i.e., only M of the N words ($M < N$) in the input must match exactly then this combinatorial problem is easily resolved due to the superimposition of the inputs. The input is sent to the network and the Willshaw threshold is set at $M \cdot B$ where B is the number of bits set in each input vector (1 for orthogonal vectors). This generalised combinatorial partial match provides a very efficient mechanism for selecting those documents that best match.

Partial matching generates multiple document vector matches superimposed in a single output vector after thresholding. These outputs must be identified. A list of valid outputs is held in a content-addressable memory and matched in the lexical token converter - binary to data. The outputs are passed to the lexical token converter (that converts the internal token representations back into the input data used by the external interface - binary to data). The time for this process is 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 lexical token converter, input the list of documents into a second lexical token converter and to store the word-document associations in the memory matrix. The memory used = word lexical token converter (data-to-binary) + document lexical token converter (data-to-binary) + document lexical token converter (binary to data) + memory matrix.

2 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
- 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

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 4.1 for comparison. The algorithms were run with the command *runon x <algorithm>* and one process from each data structure was run in parallel. This ensured that each data structure was evaluated simultaneously while the other processors in the Origin were free to furnish other processes and the timings should therefore be more reliable with less variance.

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. The training time was the time to read in the list of

words, input them to the data structure as appropriate and read in each of the word-to-document associations adding the appropriate links in the data structures. For the CMM the list of documents was also read into an array prior to adding the associations.

3. Serial Match - the words are matched one at a time with the matching documents retrieved after each word. The alphabetically sorted list of words was read into an array included in the timing. For all data structures an identical output (the word to be matched and the list of matching) was generated. Two serial match investigations were run on all methods.

   (a) Retrieve the documents that match each of the first 100 words in turn (iterative) from the alphabetically sorted array of all words. 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 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 of the data structures and should favour the hash table as the first 100 words are less likely to have suffered collisions as the hash table was virtually empty as these word 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 word list in turn (iterative). This matches the documents against 189 words

(9491 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 9 and 10). 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 - this compares the retrieval speed of CMMs compared to alternative methods that were implemented for partial matching. 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, 1, ... and at least 9 of the first 100 words. Figure 9 is a graph of the frequency distribution of the number 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, 1, ... and at least 9 of every 50th word. Figure 10 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, 11, ... and at least 37 of the most frequently occurring words - the words that occur in at least 20% of the documents. There are up to 38 words in the word set to match and from the graph (see figure 11) the documents match between 0 and 38 words.

3 Results

3.1 Memory Usage

The memory usage for each of the constituent substructures for each methodology is given below (in bytes).

1. Inverted File List Length: 9491.

   List memory use for all document lists is 9847144.

   Array (array of lists) memory use is 75928.

   WordTable (array of words) memory use is 475000.

   Memory use for array of documents (counter for partial match) 73000.

   Memory use for array of words (to get every 50th word etc.) 474550.

   Total memory use is 10945622.

2. Hash Table Length: 20023.

   List memory use for all document lists is 9847144.

   Array (array of lists with empty locations) memory use is 160184.

   WordTable (hash table of words length 20023) memory use is 1001150.

   Memory use for array of documents (counter for partial match) 73000.

   Memory use for array of words (to get every 50th word etc.) 475000.

   Total Memory use is 11556478.

3. Hash Table Compact Length: 9492.

   List memory use is 9847144. All three data structures have identical memory usage for the lists.

Array (array of lists with length 9492) memory use is 75936.

WordTable (hash table of words with length 20023) memory use is 1121288. 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 73000.

Memory use for array of words to get every 50th word etc. 475000.

Total Memory use is 11002632.

4. CMM

   - Binary - storing the CMM as a binary representation.
     CMM saturation is 0.007.
     CMM uses 44283360 bytes. The memory usage is very high in comparison to the other data structures.

   - Compact -storing the CMM as a compact representation.
     CMM saturation is 0.711.
     CMM uses 21378128 bytes. The memory usage is high in comparison to the other data structures.

   - Efficient - storing the CMM as an efficient representation with a switch value of 300.
     CMM saturation is 0.007.
     CMM uses 10958614 bytes. The memory usage is much lower than the compact or binary representations.
     Document decoder (binary to data) memory use is 1095132.
     Document encoder (data to binary) memory use is 1387156.
     Word encoder (data to binary) memory use is 759816.
     Memory use for array of words 475000 to retrieve every 50th word

etc. and to maintain simple data types for the encoder.

Total memory use is 14675718.

3.2 Training Times

Twenty training times were noted for each algorithm 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	CMM - efficient
Training time	1965.365	1570.48	1568.96	98.31

3.3 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 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

3.4 Partial Match

A graph is given for each partial match evaluation, listing the number of words to be retrieved - at least X 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.

3.4.1 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 is given in figure 12 for each data structure.

3.4.2 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 is given in figure 13 for each data structure.

3.4.3 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 14 for each data structure.

4 Analysis

4.1 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 algorithms in 64-bit mode are in ascending order: IFL (10471072), hash table compact (10527632), hash table (11081478) and CMM (14675718 excluding the array of documents). If the first three data structures are compiled as 32-bit then their respective sizes are: 5509536, 6077814 and 6155828. As stated previously, the inverted file list will use 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) is more memory efficient than having empty list array elements in the standard hash table. The CMM has the highest memory usage. The CMM memory usage is 1.4 times higher than the IFL when both are compiled in 64-bit form or 2.66 times higher when the IFL is compiled as 32-bit and AURA as 64-bit. However, none of the memory usage statistics is significantly larger than the others. 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.

4.2 Training Times

The IFL is the slowest to train as expected as each search for the word linked to the document to be trained requires a binary search through the entire array of words $O(\log n)$ before the document may be appended to 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 appends the document 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 append the document to the word's list. Although the hash table is $\Theta(1)$ for word search initially, if there are collisions, it will $\rightarrow O(n)$. The CMM simply associates the word with the document 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.

4.3 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 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 bit vector, input to the CMM, the output bit vector retrieved and the output matched against the documents in the document decoder to retrieve the matching documents. The CMM takes 7.06 times longer for the first 100 and 8.55 times longer for every 50th word than the standard hash table.

4.4 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 ascribed 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 but falls below at 'at least 3' words matched on the 'first 100' graph and 'at least 5' on the 'every 50th' word graph. For the words present in at least 20% of the documents (see figure 15), the CMM is faster than the hash table with an increasing difference for 'at least 13' or more. The difference will level off and eventually fall as the y-axis forms 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 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 compared to 100 words to be matched to retrieve the documents containing the required number of words.

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, retrieve their associated binary bit vectors from the lexical token converter, 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 bit vector retrieved from the thresholded output will have more bits set. Thus when this bit vector is matched against the document vectors in the lexical token converter (binary-to-data) 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 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 hash table is the most efficient methodology, the storage is slightly higher than the word array but the retrieval speed is much higher.

For partial matching the CMM performs best. Although the memory usage is higher (but not significantly so) the retrieval speed is superior; 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 only a slightly higher 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. 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.

References

[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. Alwis and J. Austin. A Neural Network Architecture for Trademark Image Retrieval. In *International Workshop on Artificial Neural Networks*, Spain, 1999.

[3] L. Ammeraal. *Algorithms and Data Structures in C++*. John Wiley & Sons, Chichester, England, 1998.

[4] 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.

[5] J. Austin, J. Kennedy, and K. Lees. A Neural Architecture for Fast Rule Matching. In *Artificial Neural Networks and Expert Systems Conference*, June 1995.

[6] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *7th International World Wide Web Conference*, 1998.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[8] 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.

[9] M. Goldszmidt and M. Sahami. A Probabilistic Approach to Full-Text Document Clustering. Technical Report ITAD-433-MS-98-044, SRI International, 1998.

[10] 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.

[11] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, MA, 1968.

[12] 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.

[13] 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.

[14] C. S. Roberts. Partial Match Retrieval via the Method of Superimposed Codes. *Proceedings of IEEE*, 67(12):1624–1642, 1979.

[15] R. Sacks-Davis and R. A. K. A Two Level Superimposed Coding Scheme for Partial Match Retrieval. *Information Systems*, 8(4):273–280, 1983.

[16] R. Sedgewick. *Algorithms in C++.* Addison-Wesley, Reading, MA, 1992.

[17] P. Zhou and J. Austin. A Binary Correlation Matrix Memory k-NN Classifier. In *International Conference on Artificial Neural Networks*, Sweden, 1998.

A Appendix.

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

A.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.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.3 Hash Table.

A.4 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\n"; 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.5 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.6 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\n"; 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;
}
```

B Figure Legends

Figure 1: Diagram showing the fraction of all documents that contain each word. Each word has an integer ID (0-9490) from its position in the alphabetically sorted list of all words.

Figure 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.

Figure 3: Diagram showing the hash table data structure.

Figure 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.

Figure 5: Diagram of the AURA modular system.

Figure 6: Diagram showing three stages of network training

Figure 7: Diagram showing system recall. The input pattern has 1 bit set so the CMM is thresholded at 1.

Figure 8: Diagram showing super-positioning of input (word) vectors.

Figure 9: Diagram showing the number of the first 100 words from the list

of all words occurring in each document. Each document has an integer ID.

Figure 10: Diagram showing the number of words counted in each document when taking every 50th word from the list of all words. Each document has an integer ID.

Figure 11: Diagram showing the number of frequent words (those in at least 20% of the documents) occurring in each document. Each document has an integer ID from 0-18248. N.B. Figure 11 has three lower frequency troughs between document ID 9000 and ID 13000. This is due to the documents in the Reuters dataset having fewer words in this section of documents.

Figure 12: Graph of the retrieval time for M of N matching with the first 100 words from the list of all words.

Figure 13: Graph of the retrieval times for M of N matching when taking every 50th word from the list of all words.

Figure 14: Graph of the retrieval times for N of M matching with frequent words (those in at least 20% of the documents).

Figure 15: 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).

C Figures

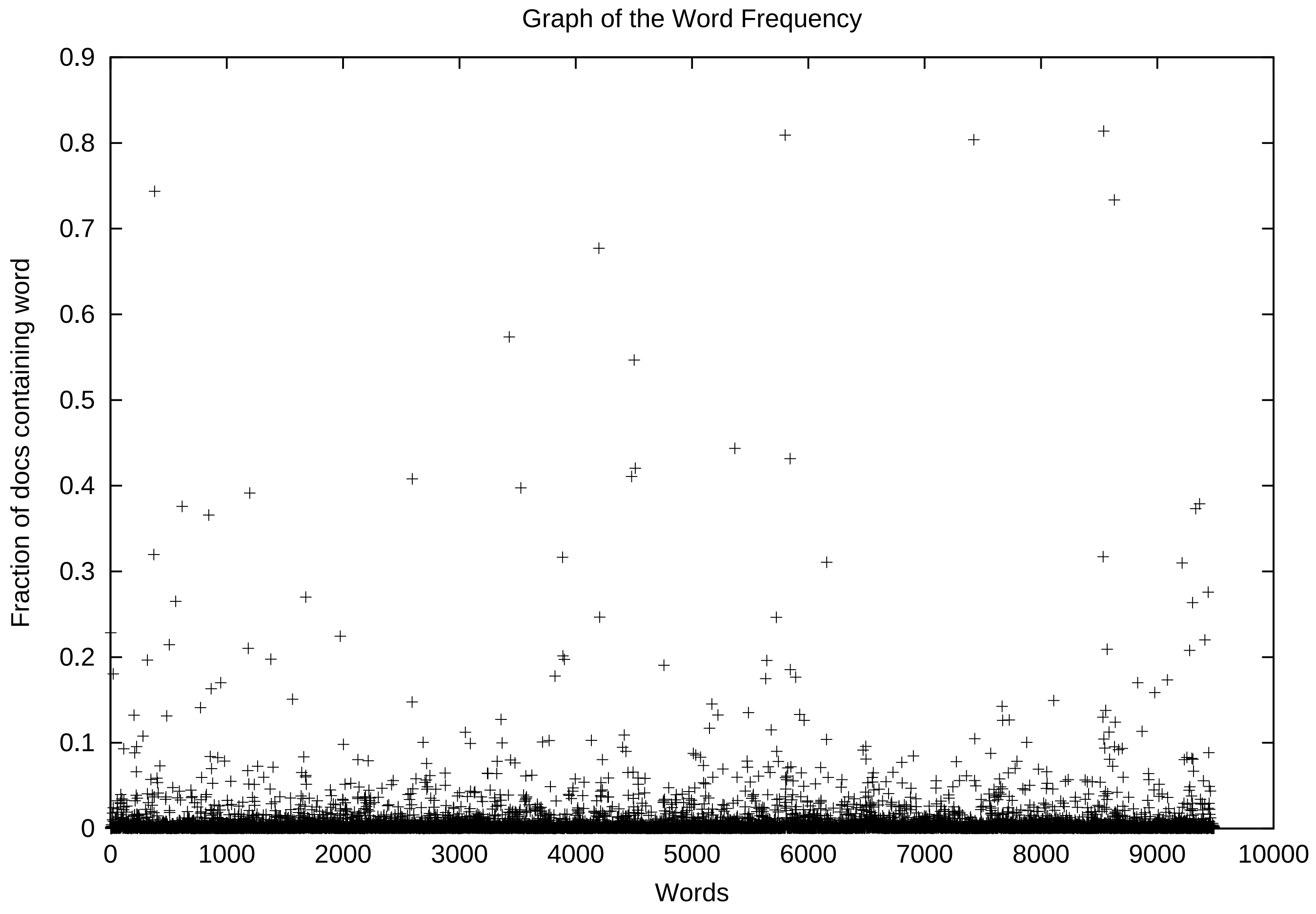

Graph of the Word Frequency
Fraction of docs containing word
Words
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

Figure 1:

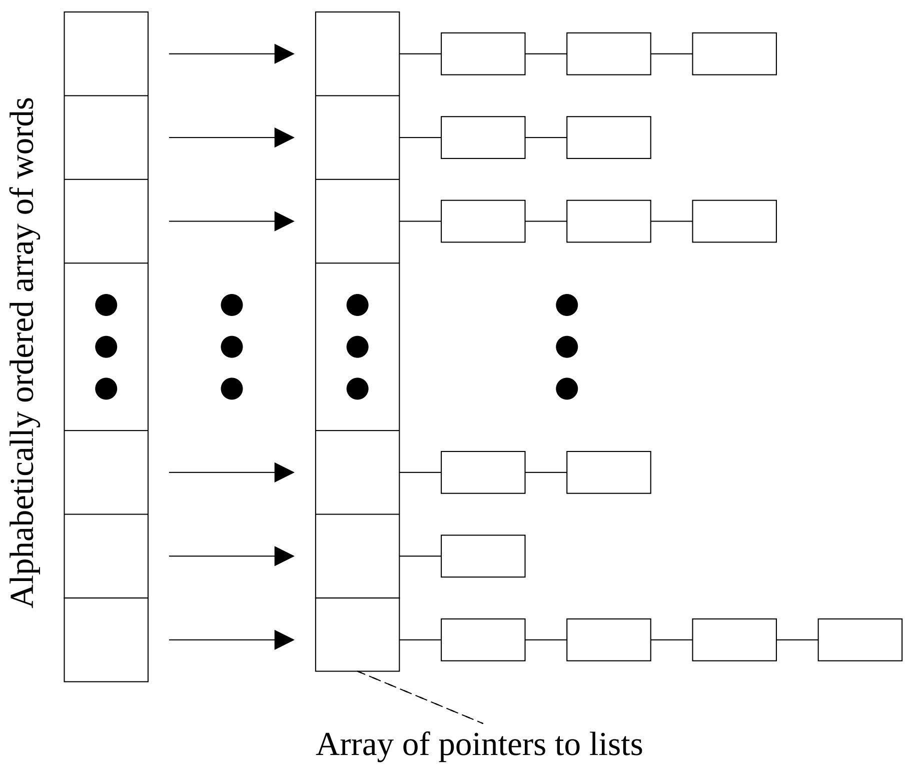

Alphabetically ordered array of words
Array of pointers to lists

Figure 2:

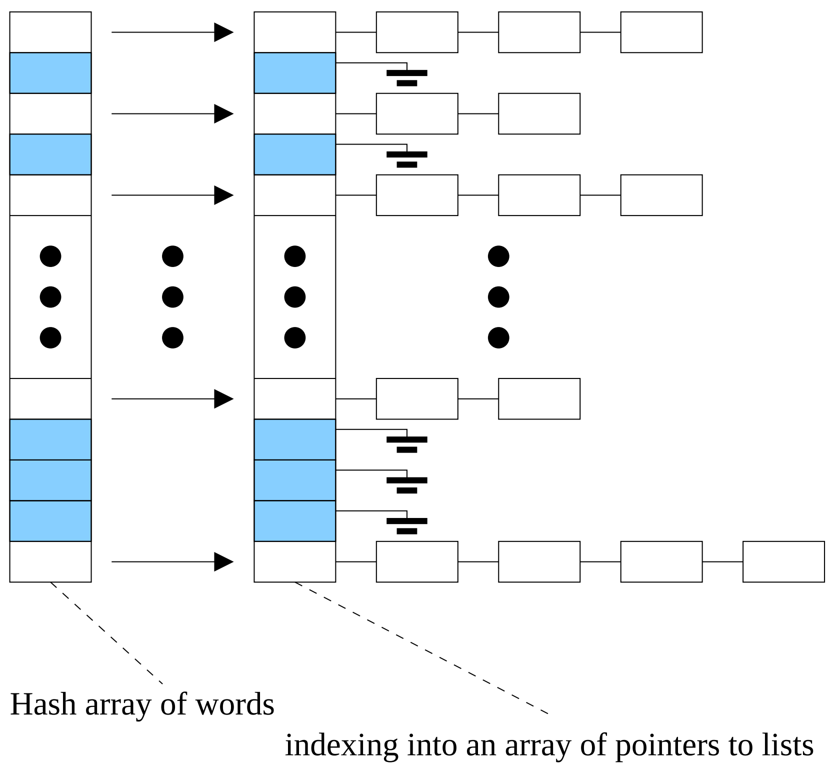

Hash array of words
indexing into an array of pointers to lists

Figure 3:

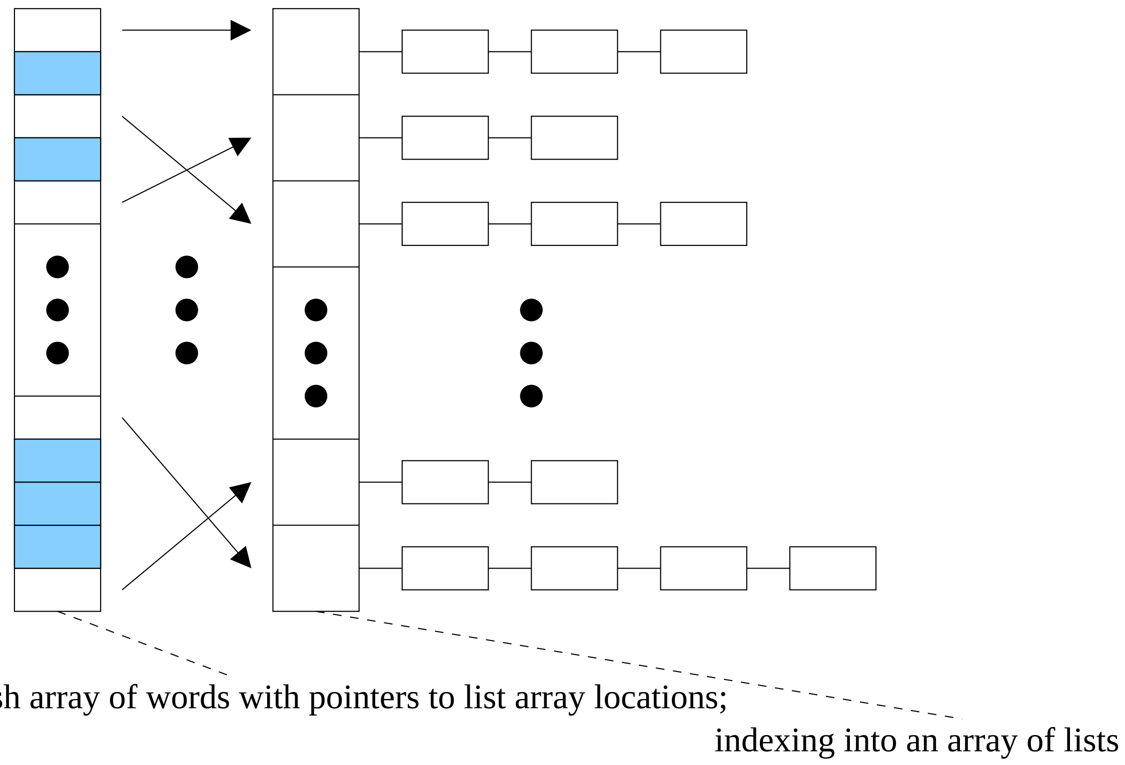

Hash array of words with pointers to list array locations;
indexing into an array of lists

Figure 4:

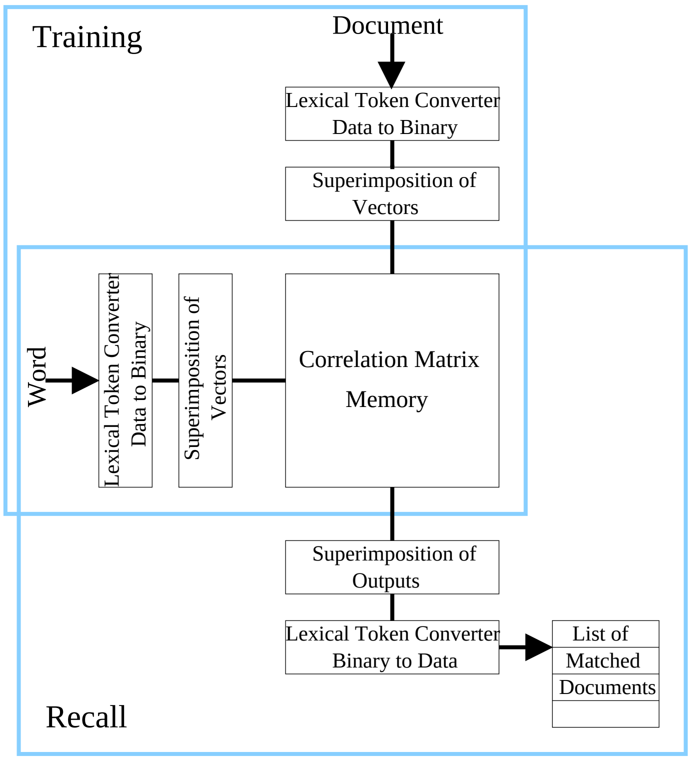

Training
Document
Lexical Token Converter Data to Binary
Superimposition of Vectors
Word
Lexical Token Converter Data to Binary
Superimposition of Vectors
Correlation Matrix Memory
Superimposition of Outputs
Lexical Token Converter Binary to Data
List of Matched Documents
Recall

Figure 5:

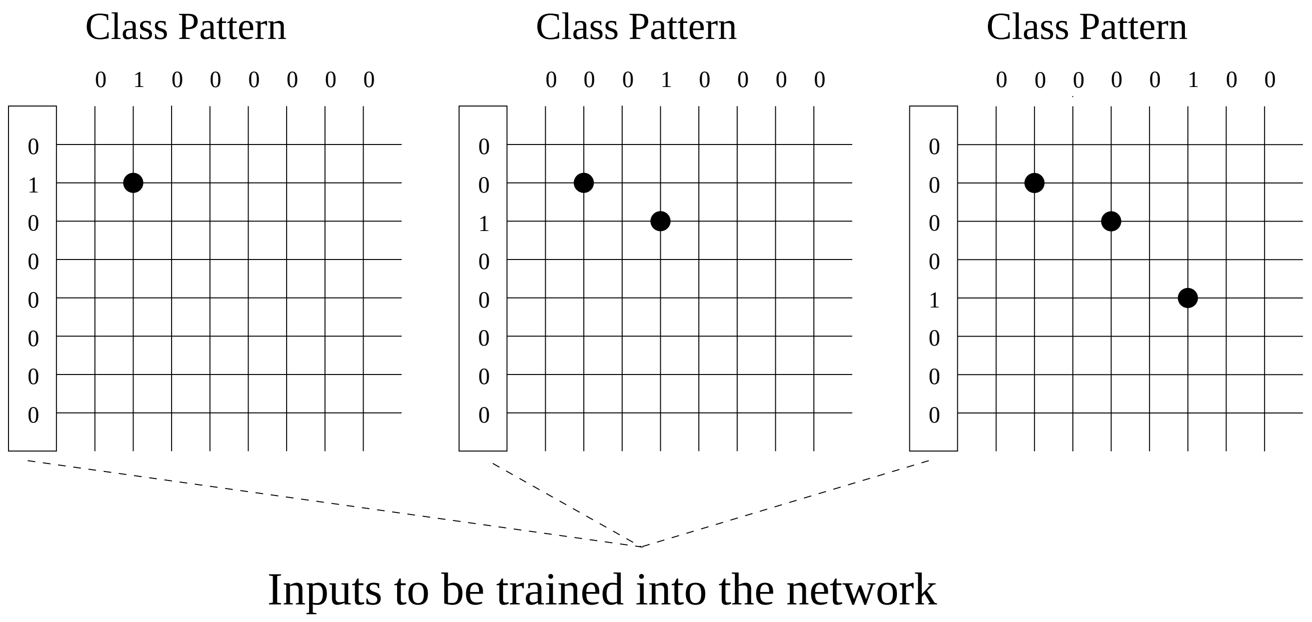

Class Pattern
0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0
Class Pattern
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0
Class Pattern
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
Inputs to be trained into the network

Figure 6:

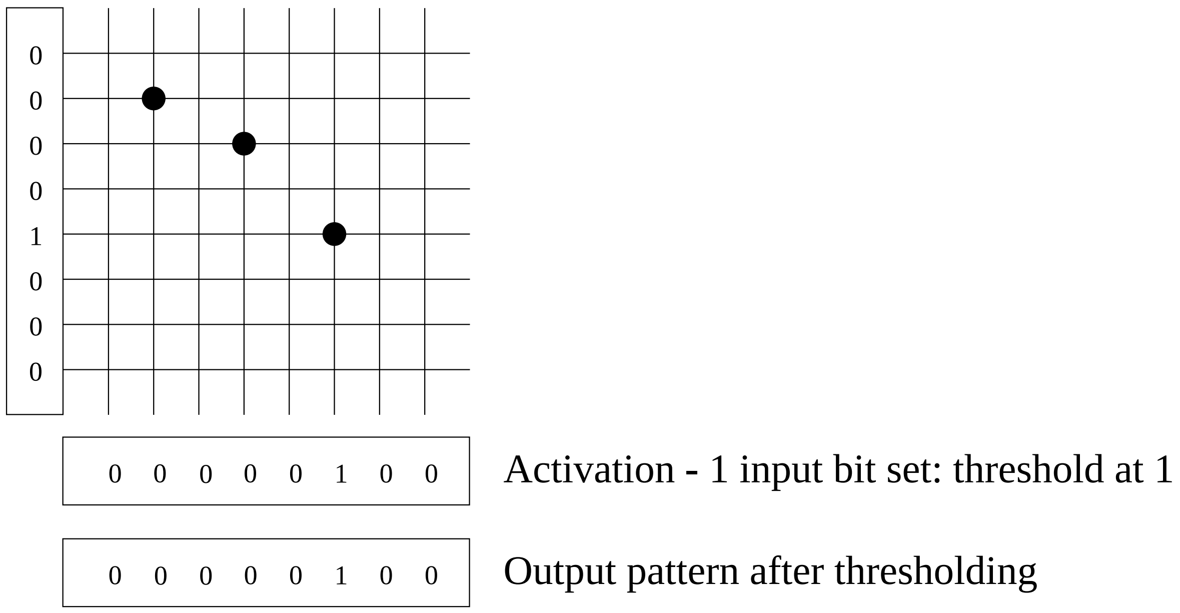

0
0
0
0
1
0
0
0
0 0 0 0 0 1 0 0
Activation - 1 input bit set: threshold at 1
0 0 0 0 0 1 0 0
Output pattern after thresholding

Figure 7:

$$\begin{array}{c}0\\1\\0\\0\\0\\0\\0\\0\\0\\0\\0\end{array} \bigvee \begin{array}{c}0\\0\\1\\0\\0\\0\\0\\0\\0\\0\\0\end{array} \bigvee \begin{array}{c}0\\0\\0\\1\\0\\0\\0\\0\\0\\0\\0\end{array} \bigvee \begin{array}{c}1\\0\\0\\0\\0\\0\\0\\0\\0\\0\\0\end{array} \bigvee \begin{array}{c}0\\0\\0\\0\\0\\0\\0\\1\\0\\0\\0\end{array} = \begin{array}{c}1\\1\\1\\1\\0\\0\\0\\1\\0\\0\\0\end{array}$$

Figure 8:

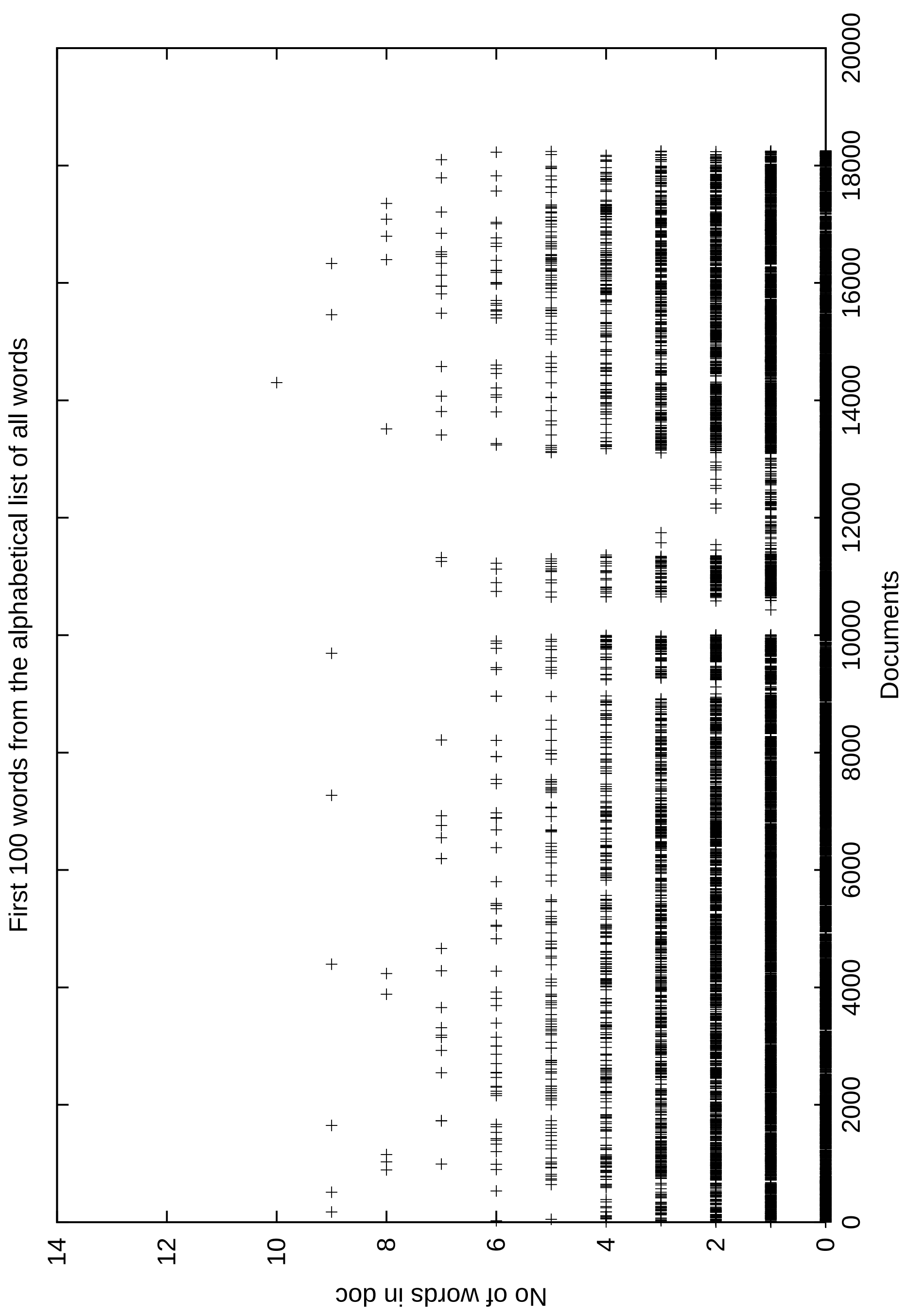

First 100 words from the alphabetical list of all words
No of words in doc
Documents
0
2
4
6
8
10
12
14
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000

Figure 9:

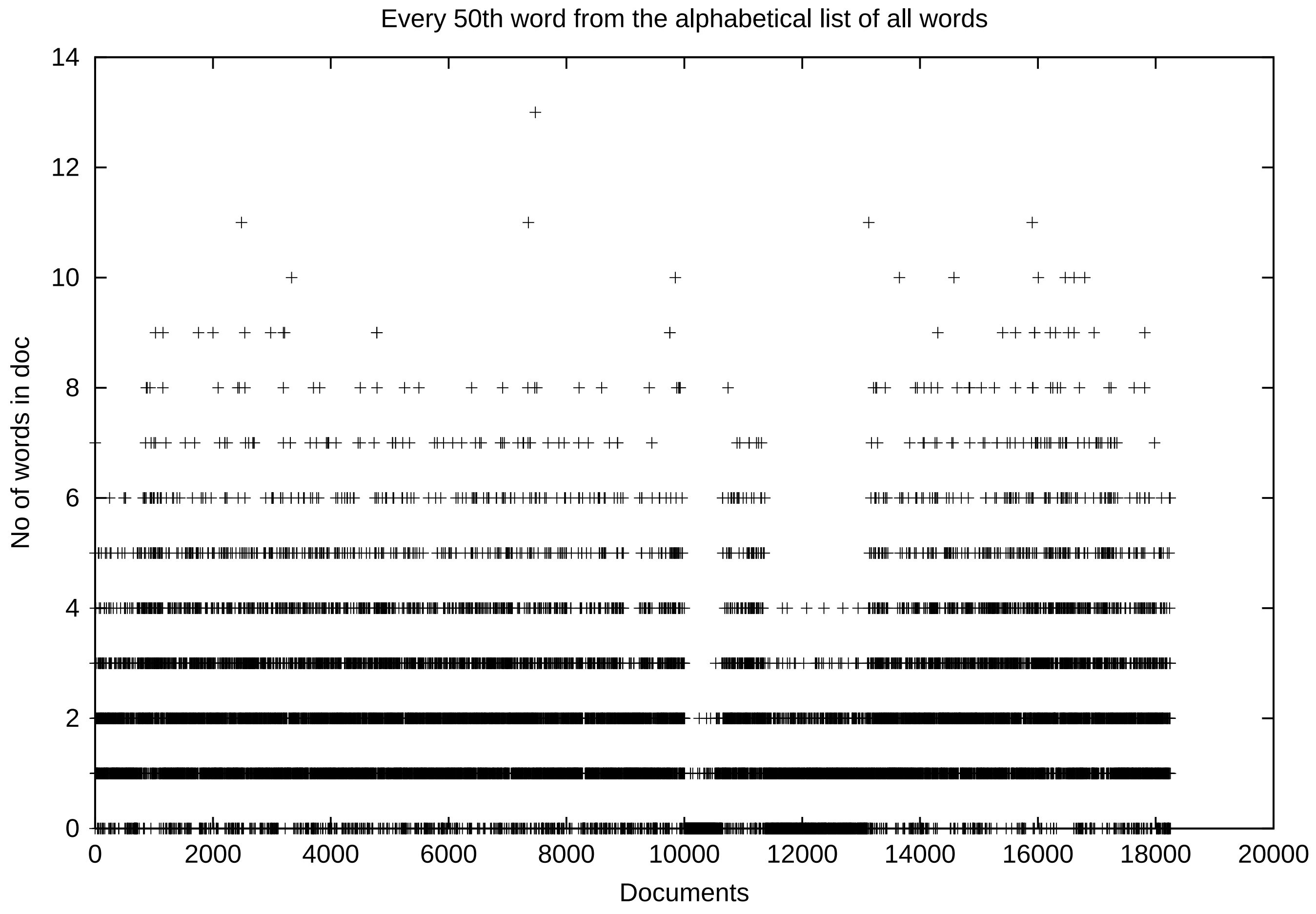
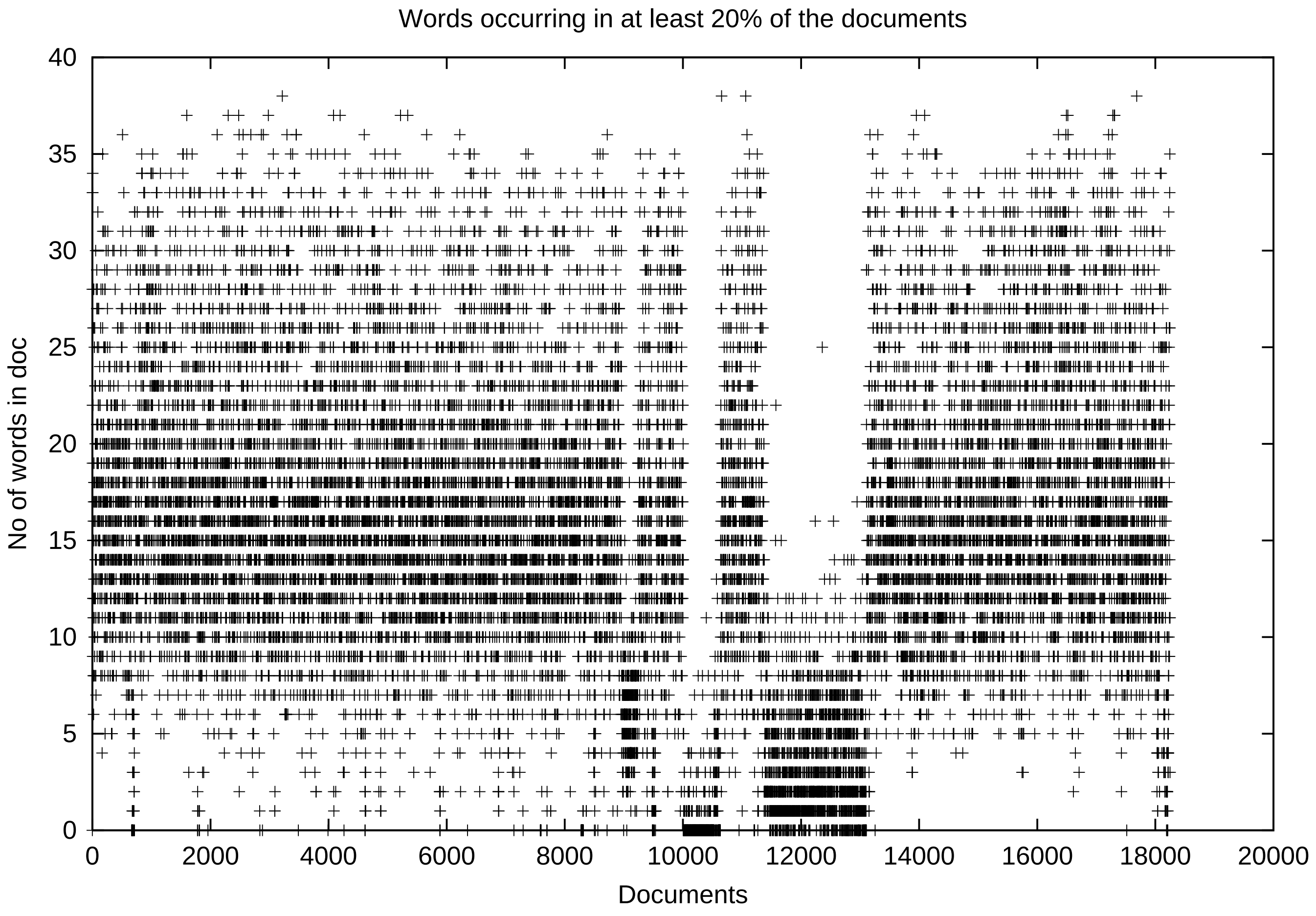

Every 50th word from the alphabetical list of all words
No of words in doc
Documents
0
2
4
6
8
10
12
14
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000

Figure 10:



Words occurring in at least 20% of the documents
No of words in doc
Documents
0
5
10
15
20
25
30
35
40
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000

Figure 11:

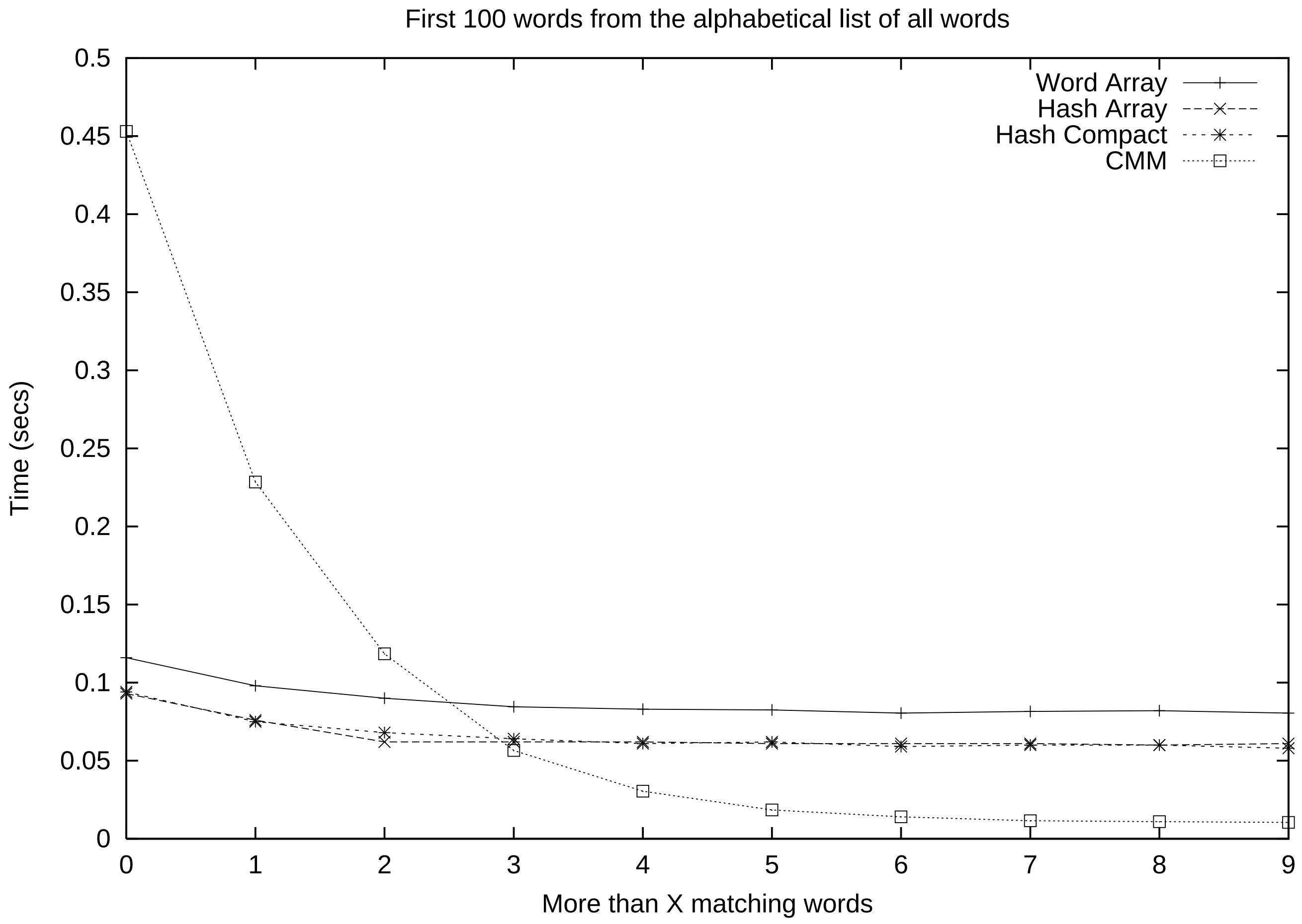

First 100 words from the alphabetical list of all words
Time (secs)
More than X matching words
Word Array
Hash Array
Hash Compact
CMM
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
1
2
3
4
5
6
7
8
9

Figure 12:

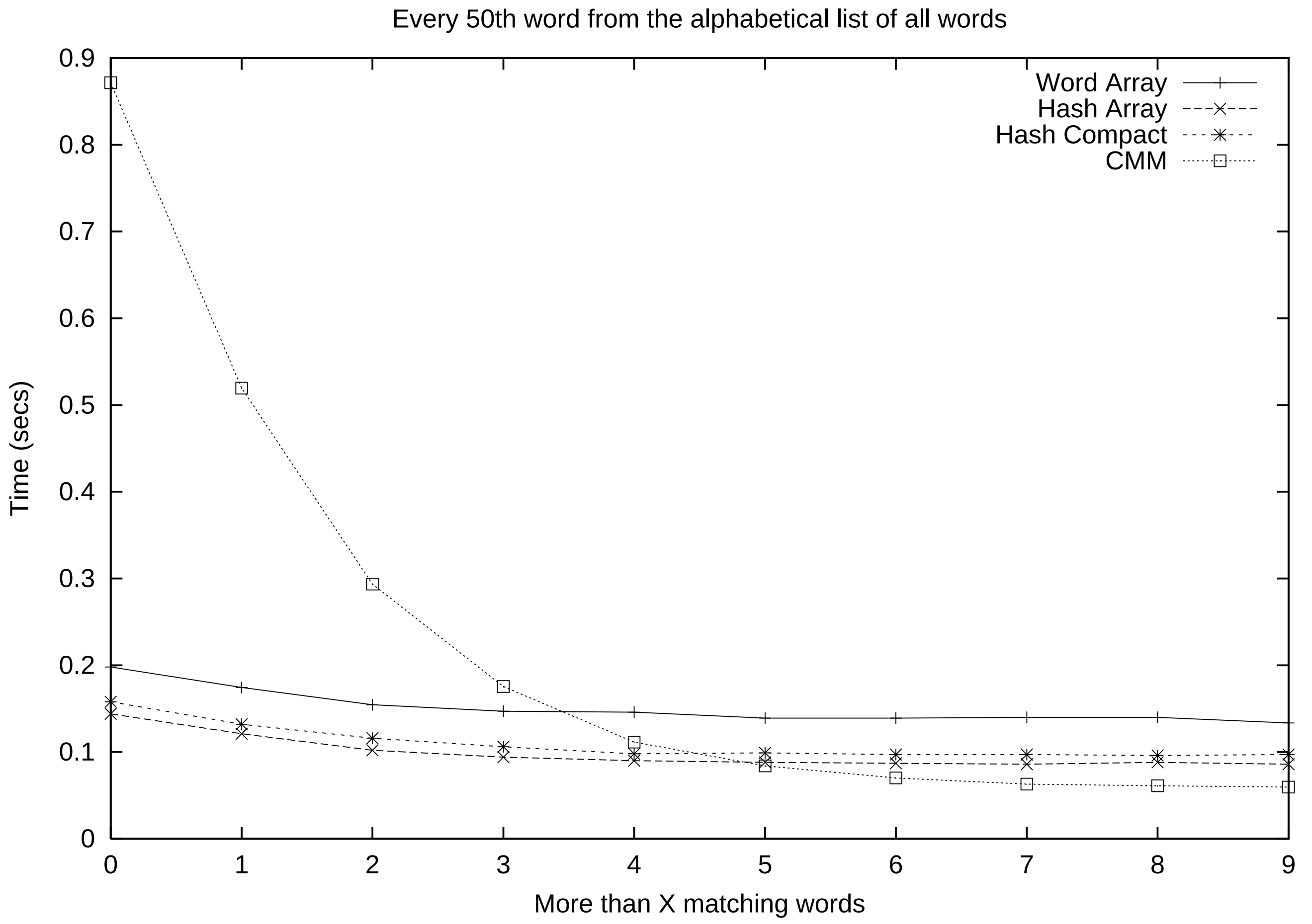

Every 50th word from the alphabetical list of all words
Word Array
Hash Array
Hash Compact
CMM
Time (secs)
More than X matching words
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
2
3
4
5
6
7
8
9

Figure 13:

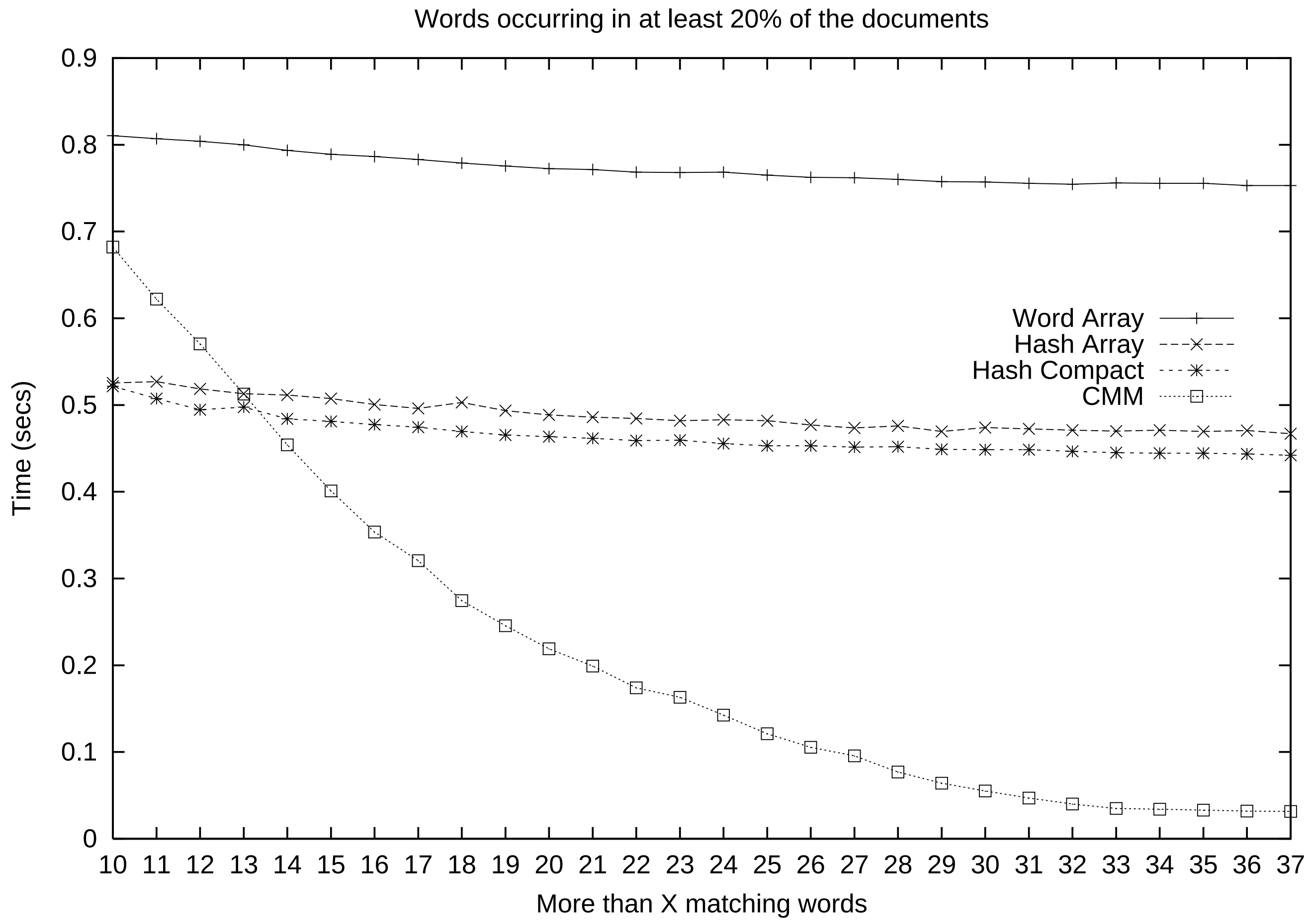

Words occurring in at least 20% of the documents
Time (secs)
More than X matching words
Word Array
Hash Array
Hash Compact
CMM
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37

Figure 14:

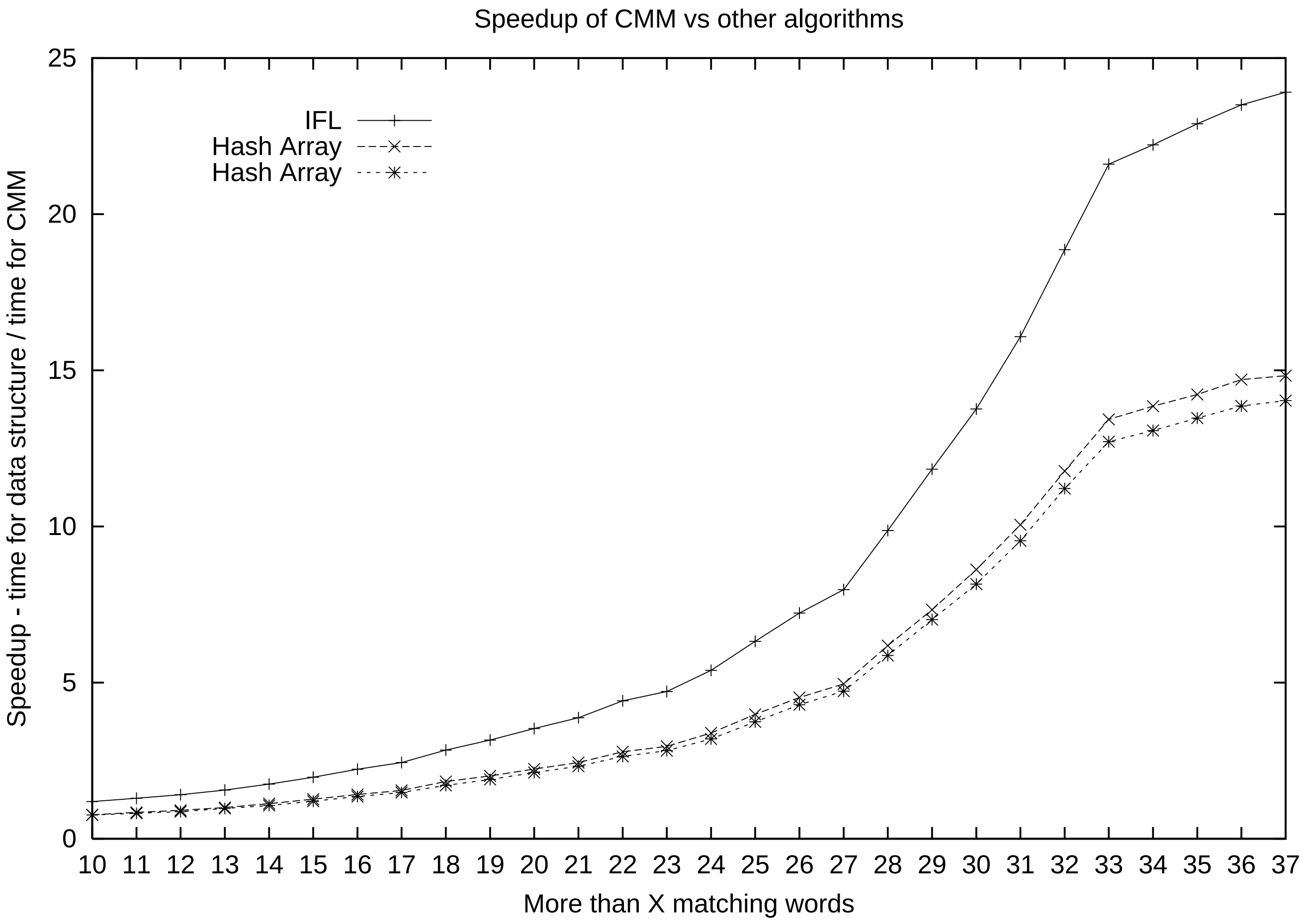

Speedup of CMM vs other algorithms
Speedup - time for data structure / time for CMM
More than X matching words
IFL
Hash Array
Hash Array
0
5
10
15
20
25
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37

Figure 15: