A Comparison of Standard Spell Checking Algorithms and a Novel Binary Neural Approach

Victoria J. Hodge and Jim Austin

Abstract—In this paper, we propose a simple, flexible, and efficient hybrid spell checking methodology based upon phonetic matching, supervised learning, and associative matching in the AURA neural system. We integrate Hamming Distance and n-gram algorithms that have high *recall* for typing errors and a phonetic spell-checking algorithm in a single novel architecture. Our approach is suitable for any spell checking application though aimed toward isolated word error correction, particularly spell checking user queries in a search engine. We use a novel scoring scheme to integrate the retrieved words from each spelling approach and calculate an overall score for each matched word. From the overall scores, we can rank the possible matches. In this paper, we evaluate our approach against several benchmark spellchecking algorithms for recall accuracy. Our proposed hybrid methodology has the highest recall rate of the techniques evaluated. The method has a high *recall* rate and low-computational cost.

Index Terms—Binary neural spell checker, integrated modular spell checker, associative matching.

1 Introduction

Errors, particularly spelling and typing errors, are abundant in human generated electronic text. For example, Internet search engines are criticized 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 in queries where little or no contextual information is available and using some measure of similarity, recommend words that are most similar to each misspelled 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.

We describe an interactive spell checker that performs a presence check on words against a stored lexicon, identifies spelling errors, and recommends alternative spellings [9]. The basis of the system is the AURA modular neural network [3] described later. 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 (see Kukich [10]) coupled with transformation rules to overcome phonetic spelling errors. "Phonetic spelling errors are the most difficult to detect and correct" [10] as they distort the spelling more than the other error types so a phonetic component is essential. We use an n-gram approach [13] to overcome the first two forms of typing errors 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 utilize 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. 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 four punctuation characters). The approach maps characters onto binary vectors and two storage-efficient binary matrices (Correlation Matrix Memories described in Section 2.3) that 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 [10], Agrep [15], [14], aspell [2], and the two benchmark approaches MS Word 97 and MS Word 2000. We evaluate our approach against these alternatives. The reader is referred to Kukich [10] for a thorough treatise of spell checking techniques. We compare our hybrid

• *The authors are with the Department of Computer Science, University of York, UK YO10 5DD. E-Mail: {austin, vicky}@cs.york.ac.uk.*

Manuscript received 30 Oct. 2000; revised 25 May 2001; accepted 27 July 2001.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 113081.

1. The percentage of correct spellings retrieved.
2. The percentage of words other than the correct spelling retrieved.

system with all of the aforementioned approaches for quality of retrieval—the percentage of correct words retrieved from 600 misspelled words giving a figure for the recall accuracy with noisy inputs (misspellings).

The paper is organized into the following sections: In Section 2, we provide a brief description of the alternative techniques used for the empirical comparison against our technique. Section 2.3 is a detailed description of the AURA neural architecture and our hybrid spelling technique using integrated AURA modules. In Section 3, we provide a qualitative comparison of the recall accuracy of our technique and those alternative approaches described in Section 2. Section 4 provides our analysis of the recall results and Section 5 contains our conclusions and suggestions for possible further developments for our technique.

2 Some Alternative Spell Checking Approaches

2.1 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 one substitution or "*ham*" to "*harm*" is one insertion. The word with the lowest score is deemed the best match.

$$f(0,0) = 0 \tag{1}$$

$$\begin{aligned} f(i,j) = min[(f(i-1,j)+1, f(i,j-1)+1,\\ f(i-1,j-1)+d(q_i,l_j)]\\ \text{where } d(q_i,l_j) = 0 \text{ if } q_i = l_j \text{ else } d(q_i,l_j) = 1. \end{aligned} \tag{2}$$

For all words, a function $f(0,0)$ is set to 0 and then a function $f(i,j)$ is calculated for all query letters and all lexicon-word letters, iteratively counting the string difference between the query $q_1q_2 \ldots q_i$ and the lexicon word $l_1l_2 \ldots l_j$. Each insertion, deletion, or substitution is awarded a score of 1, see (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 when using large dictionaries.

2.2 Agrep

Agrep [15], [14] 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 [15] for details of partitioning). 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, minimizing the computation required. There is one array for each error number for each word, so for k errors there are $k+1$ arrays ($R^0 \ldots R^k$) for each word. 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 and OR denote logical AND and OR, respectively, and S_c is a bit vector representing the character being compared c. The following two equations describe the matching process for up to k errors $0 < d \leq k$.

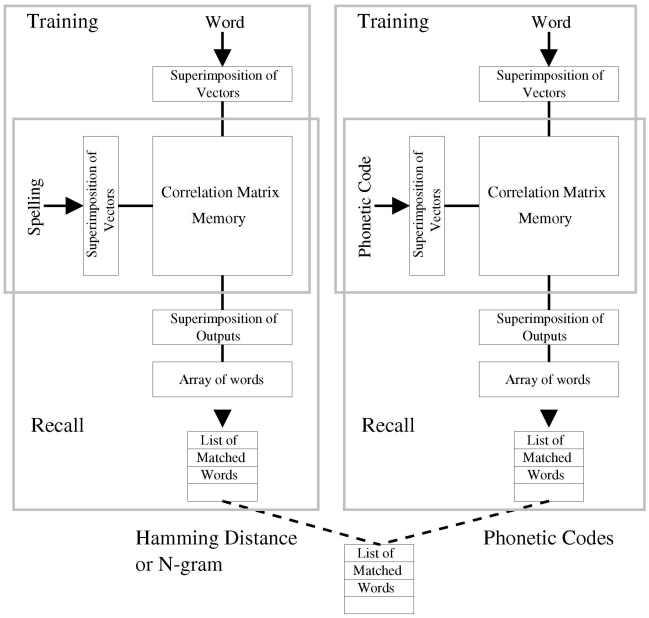

Fig. 1. Diagram of the hybrid spell checker as implemented in the AURA modular system.

$$R_0^d = 11 \ldots 100 \ldots 000 \text{ with d bits set} \tag{3}$$

$$\begin{aligned} R_{j+1}^d = &Rshift[R_j^d] \text{ AND } S_c \text{ OR}\\ &Rshift[R_j^{d-1}] \text{ OR } Rshift[R_{j+1}^{d-1}] \text{ OR } R_j^{d-1}. \end{aligned} \tag{4}$$

For a search with up to k errors permitted, there are $k+1$ arrays as stated previously. There are two shifts, one AND, and three OR operations for each character comparison (see [15]), so Wu and Manber quote the running time as $O((k+1)n)$ for an n word lexicon.

2.3 Our Integrated Hybrid Modular Approach

We implement all spelling modules in AURA. AURA [3] is a collection of binary neural networks that may be implemented in a modular fashion. AURA utilizes Correlation Matrix Memories (CMMs) [3] to map inputs to outputs (see Fig. 1) through a supervised learning rule, similar to a hash function. CMMs are binary associative $m * n$ matrix memory structures that store a linear mapping μ between a binary input vector of length m and a binary output vector of length n (see (5)). The input vector i addresses the m rows of the matrix and the output vector o addresses the n columns of the matrix (see Fig. 2).

$$\mu : \{0,1\}^m \rightarrow \{0,1\}^n. \tag{5}$$

AURA does not suffer from the lengthy training problem of other neural networks as training is a single epoch process (O(1) for each association) generating the network's high speed. Storage is efficient in the CMMs as new inputs

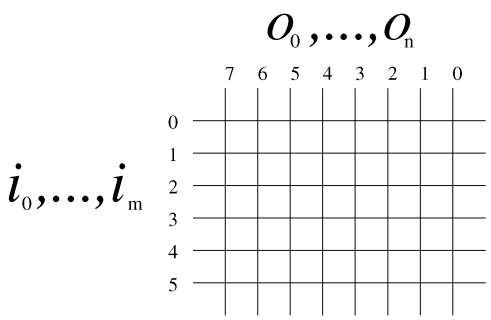

Fig. 2. The input vector i addresses the rows of the CMM and the output vector o addresses the columns.

are incorporated into the matrix and do not require additional memory allocation. AURA is also able to partially match inputs in a single step process and is much faster than conventional symbolic approaches [8]. In the implementation described in this paper, there are no false positive or false negative matches as CMMs retrieve all expected matches during single iteration partial matching [12]. We also use orthogonal vectors to uniquely identify each output word. Retrieval is faster for orthogonal vectors compared to the nonorthogonal compressed variant because no validation is necessary as there are no ghost matches caused by bit interference, which is a problem for nonorthogonal vectors where multiple bits are set [12]. Modern computers have sufficient memory to handle the small memory increase when using the longer orthogonal vectors. We provide the implementation details and theoretical running time for our AURA spell checker in the following sections.

2.4 Our Methodology

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

2.4.1 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. For the inputs, we divide a binary vector of length 960 into a series of 30-bit chunks. Words of up to 30 characters may be represented, (we need two additional character chunks for the shifting n-gram described later). Each word is divided into its constituent characters. The appropriate bit is set in the chunk to represent each character (see Fig. 3 for a table listing which bit represents which character), in order of occurrence. The chunks are concatenated to produce a binary bit vector to represent the spelling of the word and form the input to the CMM. Any unused chunks are set to all zero bits.

Each word in the alphabetical list of all words in the text corpus has a unique orthogonal binary vector to represent it. The binary vector forms the output from the CMM for that word so we can identify when the word has been retrieved as a match. A single bit is set corresponding to the position of the word in the alphabetical list of all words (see (6)). Essentially, we associate the spelling of the word to an orthogonal output vector to uniquely identify it and maintain the system's high speed as no output validation is necessary. When we output a vector from the CMM, it is a simple task to lookup the bit set and use the position to index the words in the alphabetical list of all words and return the matching word at that position.

$$bitVector^p = pth \text{ bit set } \forall p \text{ for p} = \text{position}\{\text{words}\}. \quad (6)$$

2.4.2 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 diagram (Fig. 4) shows a CMM after one, two, and three patterns have been trained. The CMM is set to one where an input row (spelling bit) and an output column (word bit) are both set (see Fig. 4). After storing all spelling-word associations, the CMM binary weight w_{kj}, where $w_{kj} \in \{0, 1\}$ for row j column k, where $\vee$ and $\wedge$ are logic "or" and "and," respectively, is given by (7)

$$\begin{aligned} w_{kj} &= \bigvee_{all\ \text{i}} inputSpelling_j^i \wedge outputWord_k^i \\ &= \left[\sum_{all\ \text{i}} inputSpelling_j^i \wedge outputWord_k^i \right]. \end{aligned} \quad (7)$$

2.4.3 Recalling from the Network—Binary Hamming Distance

We essentially count the number of aligned letters (same letter and same position in the word) between the input spelling and the lexicon words. For recall only, the spelling input vector is applied to the network. The columns are summed as in (8)

$$output_j = \sum_{alli} input_i \wedge w_{ji} \text{ where } w_{ji} \in \{0, 1\} \quad (8)$$

and the output of the network thresholded to produce a binary output vector (see Fig. 5). The output vector represents the words trained into the network matching the input spelling presented to the network for recall. We use the Willshaw threshold (see [3]) set to the highest activation value to retrieve the best matches (see Fig. 5). 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 (see Fig. 6). The output vector is a superimposition of the orthogonal vectors representing

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	-	'	&	/

Fig. 3. Table indicating which bit is set in the 30-bit chunk representing each character.

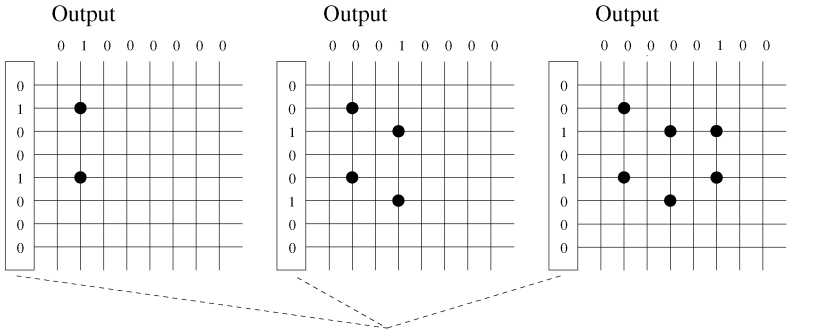

Fig. 4. Diagram showing three stages of network training.

the words that match the maximum number of the input word's characters. This partial match provides a very efficient and rapid single-step mechanism for selecting those words that best match. The CMM retrieval is $(O(1))$ as one input presentation retrieves all matching outputs.

We are able to use the "?" convention from UNIX for unknown characters 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."

2.4.4 Recalling from the Network—Shifting n-grams

This stage counts the number of n-grams in the input spelling present in each lexicon word. We use exactly the same CMM for the n-gram method as we use for the Hamming Distance retrievals. However, we use three n-gram approaches, unigrams for spellings with less than four characters, bigrams for four to six characters, and trigrams for spellings with more than six characters. Misspelled words with less than four characters are unlikely to have any bigrams or trigrams found in the correct spelling, for example, "teh" for "the" have neither common. Spellings with four to six characters may have no common trigrams but should have common bigrams and words with more than six characters should match trigrams. 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 trigrams below and in Fig. 7. All n-gram techniques used operate on the same principal, we essentially 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 Fig. 7. We wish to find lexicon words matching all three letters of the trigram, i.e., all words with an output activation of three for their first three letters. In the left-hand CMM of Fig. 7, "theatre" and "the" match the first three letters so their corresponding output activations are three. When we threshold the output activation at the value three, the bits set in the thresholded output vector correspond to a superimposition of the bits set to uniquely identify "theatre" and "the." We then slide the trigram one place to the right, input to the CMM, and threshold at three to find any words matching the trigram. We continue sliding the trigram to the right until the first letter of the trigram is in the position of the last character of the spelling. We match the length of the input plus two characters as nearly all spelling mistakes are within two letters of the correct spelling [10]. We logically OR the output vector from each trigram position to produce an output vector denoting any word that has matched any of the trigram positions. We then move onto the second trigram "hea," left align, input to the CMM, threshold, and slide to the right producing a second trigram vector of words that match this particular trigram in any place. When we have matched all n trigrams {"the," "hea," "eat," "atr," "tre"} from the spelling, we will have n output vectors representing the words that have matched each trigram, respectively. We sum these output vectors to produce an integer vector

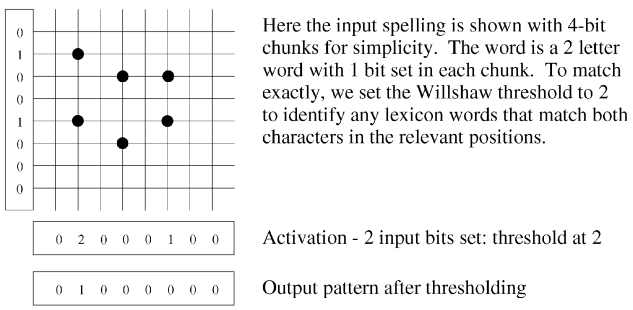

Fig. 5. Diagram showing system recall. The input pattern has 2 bits set so the CMM is thresholded at 2.

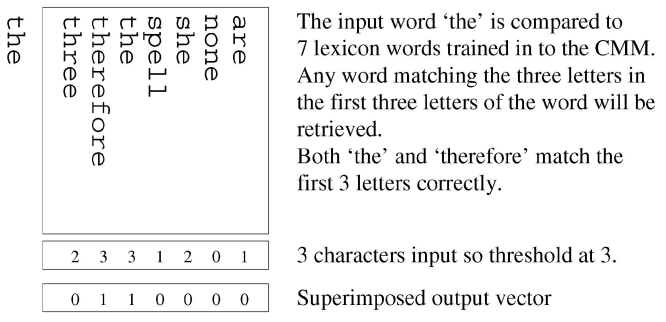

Fig. 6. Diagram showing Hamming Distance matching.

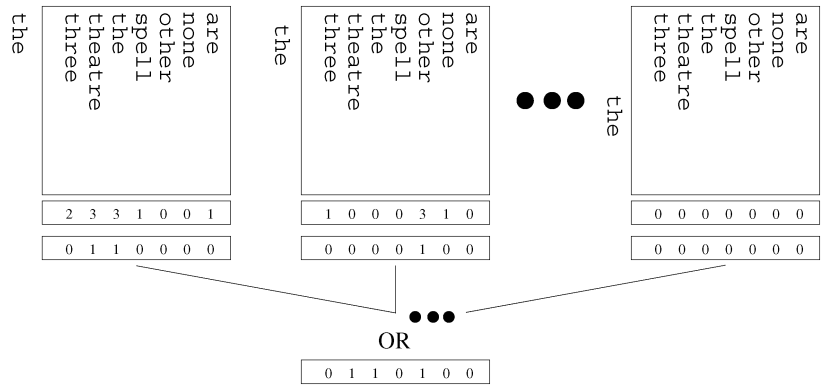

Fig. 7. Diagram showing a trigram shifting right.

representing a count of the number of trigrams 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 (words), i.e., the columns (words) with the highest activation.

2.4.5 Phonetic Spell Checking

Our methodology combines Soundex-type codes with phonetic transformation rules as with Phonix [6] to produce a four-character code for each word. However, we use fewer rules than Phonix which was designed for name matching and includes Dutch rules. We also wish to minimize the rule base as much as possible to maintain speed. We studied the etymology of the English language detailed in the Concise Oxford Dictionary and spelling errors garnered from Internet NewsGroups and hand-written text, and integrated a small subset of rules derived from Phonix and aspell [2] that we felt were particularly important for a generic spell checker. We also use a larger number of codes than Soundex and Phonix. We use 14 codes compared to seven in Soundex and nine in Phonix to preserve the letter pronunciation similarities while preventing too many dissimilar letters mapping to the same code value. The letters are translated to codes indexed from 0-D (hexadecimal) to allow a single character representation of each code value and ensure only four character codes are generated. We minimize false positives as only similar sounding words map to the same four-character code. Both Soundex and Phonix have low precision as they generate many false positives [16] due to the limited number of code permutations.

For our method, any applicable transformation rules are applied to the word. The phonetic transformation rules are given in Table 1. The code for the word is generated according to the algorithm[3] in Fig. 8 using the codes given in Table 2. The letters c, q, and x do not have a code as they are always mapped to other letters by transformation rules: $c \rightarrow s$ or $c \rightarrow k$, $q \rightarrow k$, and $x \rightarrow z$ or $x \rightarrow ks$. The function Soundex() returns the code value for the letter as detailed in the code table in Table 2. If the code produced from the algorithm contains less than four characters, then any empty characters are padded with 0s.

For phonetic spelling, the phonetic codes form the inputs and the matching words from the lexicon form the outputs of the CMM. Each word is converted to its four character phonetic code. For the input (phonetic) vectors, we divide a binary bit 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 Table 2 where the position of the bit set is the hexadecimal value of the code. The chunks are concatenated to produce a binary bit vector to represent the phonetic code of the word and form the input to the CMM.

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 bit vector to

TABLE 1
Table of the Phonetic Transformation Rules in Our System

^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

Italicized letters are Soundex 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.

3. The prefixes {hough, cough, chough, laugh, rough, tough, enough, trough} are converted to their respective phonetic codes and any remaining letters in the word are then converted as per the algorithm. For example, *laughs*, the prefix "laugh" is converted to "*l3*" and the remaining letter "s" is converted to "B" using the algorithm in Table 8 giving a phonetic code "*l3B*0" after padding with 0s to produce the four-character code.

```
First letter of code is set to first letter of word
For all remaining word letters {
   if letter maps to 0 then skip
   if letter maps to same code as previous letter then skip
   Set next code character to value of letter mapping in table 2
}
If the code has less than 4 characters then pad with 0s
Truncate the code at 4 characters
```

Fig. 8. Figure listing our code generation algorithm in pseudocode. Skip jumps to the next loop iteration.

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 (9)).

$$bitVector^p = pth \text{ bit set } \forall p \text{ for p } = \text{position}\{\text{words}\}. \quad (9)$$

2.4.6 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.

2.4.7 Superimposed Outputs

Partial matching generates multiple word vector matches superimposed in a single output vector after thresholding (see Fig. 6). 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 (6) and (9)). By retrieving a list of the bits set in the output vector, we can retrieve the matched words from the corresponding array positions (see Fig. 9). The time for this process is proportional to the number of bits set in the output vector $\Theta(\text{bits set})$, there is one matching word per bit set for orthogonal output vectors.

2.5 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 2.4.3), thresholding at the length of the input spelling (number of bits set) to find all words beginning with the input spelling. In Fig. 6, the input would be thresholded at 3 (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 1 and threshold the output at the exact length of the input spelling (number of bits set). There is 1-bit per character, so, if all bits are activated and summed, we effectively count the length of the word. We could have stored the length of each word with the word array, but we felt the additional storage overhead created was unnecessary particularly as our exact match is extremely rapid (0.03 seconds for a 29 letter word on a 180MHz MIPS R10000 processor and 30,000 word lexicon) using the CMM. From Fig. 6, if all bits are set in the input and the output thresholded at exactly 3 this will retrieve the three letter words {"the," "are," "she"}. We can then logically AND the Hamming Distance output vector with this vector (the length vector), to retrieve the exact match if one exists, i.e., matching all input characters AND the exact length of the input. The bit set in the ANDed bit vector indexes into the word array to retrieve the matching word {"the"}.

If the exact match process does not return any matches, we assume the query word is spelled incorrectly. We can then spell check using the query word and produce a list of alternative spellings. We present the input vector for Hamming Distance, then shift n-gram to the first CMM and phonetic spelling to the second CMM (see Section 2.4 for details of how the input vectors are produced), and generate a separate output vector for each method with an activation set for each word in the lexicon. We threshold the output activation vector from the Hamming Distance method at the maximum attribute value of the vector to recover the best matches. We threshold the output activation vector from the shifting n-gram method at the highest value and also the phonetic output. The corresponding best matching words are retrieved from the word array for each

TABLE 2
Table Giving Our Codes for Each Letter

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

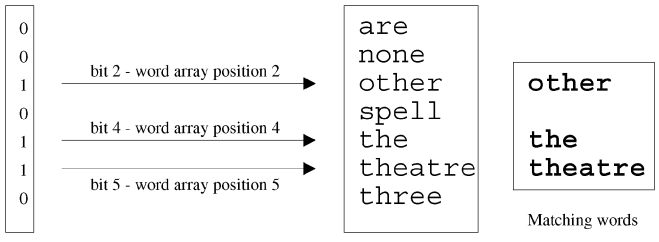

Fig. 9. Diagram showing matching word retrieval using the set bits in the thresholded output vector as positional indices into the word array.

thresholded vector, i.e., we identify the array locations of the best matching words from the positions of the bits set in the thresholded vectors. Retrieval times for Hamming Distance and the phonetic match are $O(1)$ taking < 0.01 seconds for both a 3-character and 29-character input and 30,000 word lexicon on a 180MHz MIPS R10000 processor [7]. Retrieval for the shifting n-gram is slower at $O(n)$ as each n-gram is shifted along the length of the input and the number of shifts is proportional to input length. Retrieval ranges from 0.01 to 1.06 seconds for a 3-character to 29-character input and 30,000 word lexicon on a 180MHz MIPS R10000 processor [7]. This compares to < 0.01 secs for a conventional n-gram or 1.53 secs for standard edit distance. The shifting n-gram is slower than a conventional n-gram approach where no shifting is required. However, we can exploit the same CMM for both Hamming Distance and the shifting n-gram which would not be possible with a conventional n-gram approach, so we minimize memory usage. Retrieval accuracy is also higher for the shifting n-gram compared to a conventional n-gram [7], so we feel the superior retrieval accuracy and memory conservation offered by the shifting n-gram mitigates the slightly slower retrieval speed. [7].

We produce three separate scores: one for the Hamming Distance best match (see (10)), one for the shifting n-gram best match (see (11)), and one for the phonetic best match (see (12)), which we integrate to produce an overall word score (see (13)). We evaluated various scoring mechanisms for recall accuracy using the UNIX lexicon and a word list from the Reuters corpus [11]. Our first approach was to simply sum all three scores to produce a word score, but this favors 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 [10] $(2 * c/(n + n'))$ 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 (11). 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 normalize both the Soundex and Hamming Distance score to give a theoretical maximum score for each, equivalent to the maximum score possible for the n-gram. We subtract $((2^*|\mathrm{n-gram}|) - 1)$ to normalize the Hamming Distance toward the n-gram score. We multiply the Soundex score by $(strlen(\mathrm{queryWord}) - (|\mathrm{n-gram}| - 1))$ to normalize to the n-gram score. This ensures that none of the three methods integrated biases the overall score, they should all be equally influential. In the following equations: $WillThresh$ is the Willshaw threshold setting; $strlen()$ is the string length;

$$\mathrm{diff} = (strlen(\mathrm{queryWord}) - \mathrm{strlen(lexiconWord)});$$

$\|$ is the length (modulus) of the n-gram or phonetic code.

$$\begin{aligned}&\mathrm{ScoreHamming} = \\ &2 * (\mathrm{WillThresh} - \mathrm{diff} - ((2 * |\mathrm{n\text{-}gram}|) - 1))\end{aligned} \tag{10}$$

$$\mathrm{ScoreN\text{-}gram} = 2 * (\mathrm{WillThresh} - \mathrm{diff}) \tag{11}$$

$$\begin{aligned}\mathrm{ScorePhonetic} = &\frac{2 * (\mathrm{WillThresh} - \mathrm{diff})}{(2 * |\mathrm{phoneticCode}|)} * \\ &(strlen(\mathrm{queryWord}) - (|\mathrm{n\text{-}gram}| - 1)).\end{aligned} \tag{12}$$

The score for the word is then given by (13).

$$\begin{aligned}\mathrm{Score} = max((&\mathrm{ScoreHamming} + \mathrm{ScorePhonetic}), \\ &(\mathrm{ScoreN\text{-}gram} + \mathrm{ScorePhonetic})).\end{aligned} \tag{13}$$

3 Evaluation

For our evaluations, we use three lexicons, each supplemented with the correct spellings of our test words: UNIX "spell" comprising 29,187 words and 242,819 characters; Beale [5] containing 7,287 words and 43,504 characters, and CRL [1] with 45,243 words and 441,388 characters.

3.1 Quality of Retrieval

We extracted 583 spelling errors from the aspell [2] and Damerau [4] word lists and combined 17 spelling errors we extracted from the MS word 2000 auto-correct list to give 600 spelling errors. The misspellings range from one to five error combinations with a reasonably even distribution of insertion, deletion, and substitution errors. Three hundred twenty-nine words have one error, 188 have two errors, 61 have three errors, 14 have four errors, and eight have five errors. We devised our phonetic rule, code base, and scoring mechanism before extracting the spelling error data set. For each lexicon in turn, we counted the number of times each algorithm suggested the correct spelling among the top 10 matches and also the number of times the correct spelling was placed first. 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 10th place (where the algorithm produced a score) but was listed lower, we counted this as top 10. In Table 3, we include the score for MS Word 97, MS Word 2000, and aspell [2] spell-checkers using their standard supplied dictionaries[4] for a benchmark comparison against the other algorithms which were all using the standard UNIX spell lexicon.[5] In Table 4, we compare our hybrid approach against the Hamming Distance, shifting n-gram, and phonetic components (scoreHamming, scoreNGram, and scorePhonetic from (13), respectively) that integrate to form our hybrid approach against agrep and Edit Distance on different lexicon sizes to investigate the effect of lexicon size on recall accuracy. We only compare the methods that can be trained on the different lexicons in Table 4.

4. 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.

5. All lexicons 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" in Table 3 and included them in the retrieved scores in Table 4.

TABLE 3
Table Indicates the Recall Accuracy for Our Hybrid, the Three Benchmark Approaches, Edit Distance, and Agrep Using the Standard Lexicons

Method	Found (Top 10)	Found (Pos. 1)	Present	Not Found	% Recall
Hybrid	558	368	6	36	93.9
Aspell	558	429	6	36	93.9
Word 2k	510	432	17	73	87.5
Word 97	504	415	15	81	86.1
Edit	510	367	6	84	85.6
Agrep	481	303	6	113	80.1

Column 1 gives the number of top 10 matches, column 2 the number of first place matches, column 3 the number of spelling variants (see footnote 5) present in the lexicon, column 4 gives the number of words not correctly found in the top 10, and the fifth column provides a recall accuracy percentage (the number of top 10 matches / (600 - number of words present)).

TABLE 4
Table Indicates the Recall Accuracy of the Methodologies Evaluated with the Beale and CRL Lexicons

	Beale			CRL		
Method	Found (Top 10)	Found (Pos. 1)	% Recall	Found (Top 10)	Found (Pos. 1)	% Recall
Hybrid	591	497	98.5	558	380	93
ScoreHamming	450	374	75	355	259	59.2
ScoreNGram	486	418	81	393	274	65.5
ScorePhonetic	529	410	88.2	488	252	81.3
Edit	524	408	87.3	500	330	83.3
Agrep	552	451	92	500	291	83.3

4 Analysis

If we consider the final column of Table 3, 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. Aspell and MS Word 2000 have more first place matches than our method. However, both aspell and MS Word were using their standard dictionaries which makes comparison contentious. Assuming a valid comparison, MS Word is optimized for first place retrieval and Aspell relies on a much larger rule base. We have minimized our rule base for speed and yet achieved equivalent recall. The phonetic component takes < 0.01 secs for a 29-letter input and 30,000 word lexicon. The user will see the correct spelling in the top 10 matches an equal number of times for both aspell and our method.

In Table 4, we can see that our integrated spell checker attains far higher recall than the three components (scoreHamming, scoreNGram, and scorePhonetic from (13)) for both lexicons so the integration and scoring mechanism is synergistic. We have achieved far higher recall than either agrep or Edit Distance for the all lexicon sizes. Of the eight words not found by our hybrid trained with the Beale lexicon, six were not found using the UNIX lexicon, one was present, and the other word was retrieved at position 11. The words not found by our spell checker trained with the UNIX lexicon were 72 percent similar to the words not found using the CRL lexicon. One word was present in the CRL lexicon and the remaining words were predominantly low top 10 matches. We feel this demonstrates consistency of retrieval across the lexicons. The words our hybrid spell checker fails to find are predominantly high error, highly distorted misspellings such as "invermeantial" for "environmental" missed for both the UNIX and CRL lexicons or misspelling that are more similar to many stored lexicon words than the true correct word, for example, our hybrid fails to find "daily" from the misspelling "daly" for the Beale lexicon as words such as "dally" and "dale" are retrieved as higher matches. We maintain 93 percent recall for the large 45,243 word lexicon which we note contains many inflected forms such as verb tenses, singular/plural nouns etc. and, thus, tests recall thoroughly as we only count exact matches as correct.

5 Conclusion

Spell checkers are somewhat dependent on the words in the lexicon. Some words have very few words spelled 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 in Table 3, our hybrid approach had the joint highest recall rate at 93.9 percent tied with aspell which uses a much larger rule base. We maintain high recall accuracy for large lexicons and increase recall to 98.5 percent for the smaller lexicon. The recall for our integrated system is far higher than the recall for any of the individual components. Humans averaged 74 percent for isolated word-spelling correction [10] (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 [10] posits that ideal isolated word-error correctors should exceed 90 percent 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 we are developing as the phonetic module will be 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 $\mathrm{sc(e|i|y)} \rightarrow \mathrm{s}$ 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 $O(1)$. The best match process is slower as the shifting triple is $O(n)$. However, we feel the ability to reuse the Hamming Distance CMM coupled with the superior quality of the shifting triple as compared to the regular nonpositional trigram [13] offsets the lower speed.

Some possible improvements include adding a facility to handle the UNIX wild-card character "*" to enable more flexible word matching. We have already implemented the UNIX "?" any single character match. The wild-card is more complicated as the letters in the word do not align. We could not use the Hamming Distance approach due to the nonalignment. We would also be limited with our shifting n-gram as this only shifts along the length of the input word. The shifting n-gram can be slow if shifted many places to the right, so this is not an option. The phonetic spelling match would also be problematic as this aligns the phonetic code characters and the wild-card would preclude alignment. If we were to implement the wild-card in a CMM, we would probably need to use the triple mapping approach as this matches the n-gram anywhere in the lexicon word. This would of course require an extra CMM to store the n-gram signature bit vectors for the lexicon words. An alternative would be to exploit the UNIX "grep" facility and use this or possibly "agrep" for wild-card matching as both are fast and do not store the lexicon thus minimizing the storage overhead.

Acknowledgments

This work was supported by an EPSRC studentship.

References

[1] Web page: ftp://ftp.ox.ac.uk/pub/wordlists, 2002.
[2] Aspell. Web page: http://aspell.sourceforge.net/, 2002.
[3] J. Austin, "Distributed Associative Memories for High Speed Symbolic Reasoning," *Proc. IJCAI '95 Working Notes of Workshop Connectionist-Symbolic Integration: From Unified to Hybrid Approaches,* R. Sun and F. Alexandre, eds., pp. 87-93, Aug. 1995.
[4] F. Damerau, "A Technique for Computer Detection and Correction of Spelling Errors," *Comm. ACM,* vol. 7, no. 3, pp. 171-176, 1964.
[5] Elcom Ltd—Password Recovery Software. Web page: http://www.elcomsoft.com/prs.html, 2002.
[6] T. Gadd, "PHONIX: The Algorithm," *Program,* vol. 24, no. 4, pp. 363-366, 1990.
[7] V. Hodge and J. Austin, "A Comparison of a Novel Spell Checker and Standard Spell Checking Algorithms," *Pattern Recognition,* vol. 35, no. 11, pp. 2571-2580, 2002.
[8] V. Hodge and J. Austin, "An Evaluation of Standard Retrieval Algorithms and a Binary Neural Approach," *Neural Networks,* vol. 14, no. 3, 2001.
[9] V. Hodge and J. Austin, "An Integrated Neural IR System," *Proc. Ninth European Symp. Artificial Neural Networks,* Apr. 2001.
[10] K. Kukich, "Techniques for Automatically Correcting Words in Text," *ACM Computing Surveys,* vol. 24, no. 4, pp. 377-439, 1992.
[11] 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, 2001.
[12] M. Turner and J. Austin, "Matching Performance of Binary Correlation Matrix Memories," *Neural Networks,* vol. 10, no. 9, pp. 1637-1648, 1997.
[13] J.R. Ullman, "A Binary n-Gram Technique for Automatic Correction of Substitution, Deletion, Insertion, and Reversal Errors in Words," *Computer J.,* vol. 20, no. 2, pp. 141-147, May 1977.
[14] S. Wu and U. Manber, "AGREP—A Fast Approximate Pattern Matching Tool," *Proc. Usenix Winter 1992 Technical Conf.,* pp. 153-162, Jan. 1992.
[15] S. Wu and U. Manber, "Fast Text Searching With Errors," *Comm. ACM,* vol. 35, Oct. 1992.
[16] J. Zobel and P. Dart, "Phonetic String Matching: Lessons from Information Retrieval," *Proc. 19th Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval,* 1996.

Victoria J. Hodge is a postgraduate research student in the Department of Computer Science, University of York. She is a member of the Advanced Computer Architecture Group investigating the integration of neural networks and information retrieval.

Jim Austin has the Chair of Neural Computation in the Department of Computer Science, University of York, where he is the leader of the Advanced Computer Architecture Group. He has extensive expertise in neural networks as well as computer architecture and vision. Dr. Austin has published extensively in this field, including a book on RAM-based neural networks.