Effective Spell Checking by Learning User Behavior*

by

Y. Zhao** and K. Truemper***

March, 1997; revised December 1997

* This research was supported in part by the Office of Naval Research under Grant N00014-93-1-0096.

** Ericsson, Inc.
1010 E. Arapaho Road
Richardson, Texas 75081
U.S.A.

*** University of Texas at Dallas
Computer Science Program
Box 830688
Richardson, Texas 75083-0688
U.S.A.

Send all communications to:
K. Truemper
University of Texas at Dallas
Computer Science Program EC31
Box 830688
Richardson, Texas 75083-0688
U.S.A.
email: truemper@utdallas.edu

Abstract

This paper describes a spell checking system that learns user behavior. Based on that insight, the system with high likelihood suggests correct replacements for incorrect words and declares unknown but correct words to be correct. The system relies on three dictionaries, a so-called user history file, and two logic modules to carry out the learning and spell checking.

Tests have proved that the system is very fast and highly reliable. Specifically, the top ranked replacement word for an incorrect word was the correct word 96% of the time. Words that were not in the large dictionary but that nevertheless were correct, for example, persons' names, compound words, and control commands, were declared to be correct 82% of the time. It was never observed that an incorrect word was accepted as correct.

Keywords: Spell checking, learning, logic application.

1. Introduction

This paper describes a spell checking system for English texts. A key element that sets the system apart from other systems, for example, Ispell (1993) or Microsoft Word (1994), is learning of user behavior. Altogether, the system has the following features.

(1.1) Each word that the user considers to be incorrect is almost surely flagged as incorrect.
(1.2) For each word flagged as incorrect, a replacement word is suggested that very likely is the one desired by the user.
(1.3) Each word that the system does not know but that the user considers to be correct, is very likely declared to be unknown but correct.
(1.4) Each word that the user previously defined to be incorrect is always declared to be incorrect.
(1.5) The spell checking is done rapidly and in a way that places no undue burden on the user.

Features (1.2)–(1.4) are user and domain specific and cannot possibly be achieved unless the system learns user behavior and preferences.

The paper proceeds as follows.

Section 2 reviews prior work on the detection and correction of spelling errors.

Section 3 summarizes the spell checking system of this paper. The system relies on three dictionaries, a user history file, and two logic modules to make its decisions.

Sections 4–6 describe the three dictionaries, called general dictionary, user dictionary, and excluded words dictionary. The general dictionary contains in excess of 200,000 words. The user dictionary lists all words the user has employed so far. With each word, the user dictionary contains, amongst other data, a word usage index that tells how often the word has been encountered in recently processed files. The excluded words dictionary consists of all words that the user considers to be incorrect regardless of circumstances.

Section 7 introduces the updating procedure for the word usage index.

Section 8 covers the user history file, which is a list of the spelling and typing errors most recently made by the user.

Section 9 discusses the construction of words from root words. The main tool is a logic formulation that defines frequently utilized construction rules. Reasoning based on that

logic formulation establishes whether a given word is constructible and, in the affirmative case, identifies the construction rule.

Section 10 concerns error detection and correction. Those tasks are carried out as follows. Suppose a given word is not in the dictionaries and is not constructible from some root word. We define such a word to be unknown. Similarly to the approach in Section 9 for word construction, a logic formulation relates various characteristics of the given unknown word and the user's recent spelling and typing behavior, to spelling or typing errors at certain likelihood levels. Reasoning based on that logic formulation identifies applicable spelling and typing errors and associated likelihood values, and eventually produces up to three replacement words.

If the systems finds replacement words, it ranks them according to weights that express the likelihood of applicability. The user accepts one of the proposed replacement words, or supplies another word, or declares the word in question to be actually correct.

If the system does not find any replacement words, it assumes that the unknown word is actually correct. The user confirms that assumption or supplies another word.

Regardless of which case applies, the system analyzes the user response, correspondingly updates the user dictionary and user history file, and thus learns user behavior and preferences. Goal of that learning is an improvement of the accuracy with which the system produces and ranks replacement words for incorrect words and declares unknown words to be actually correct.

Section 11 covers a convenient way in which the system learns special vocabularies.

Section 12 describes test results. They show that the system is very fast and highly reliable and that it computes correct replacements words and identifies unknown, correct words as correct with impressive accuracy.

Further details may be found in Zhao (1996), which also describes a companion system for syntax checking.

2. Prior Work

We review prior results on spell checking.

Error Detection

Efficient pattern matching and string comparison techniques have been explored for deciding whether an input string appears in a predefined dictionary. Two widely used techniques for error detection are *n-gram analysis* (Morris and Cherry (1975), Zamora, Pollock, and Zamora (1981)) and *dictionary lookup* (Knuth (1973), Aho and Corasick (1975), Peterson (1980)). We summarize the two approaches.

An *n-gram* is an n-letter subsequence of a string, where n usually is 1, 2, or 3. In general, *n-gram analysis techniques* check each n-gram in an input string against a precompiled table of n-gram statistics to determine whether the n-gram can occur in a word. If it does, its frequency of occurrence in the words of the language is computed. Strings containing n-grams that do not occur in words or occur very infrequently are considered to be possible misspellings.

Dictionary lookup techniques check whether an input string appears in a dictionary. Response time may become a problem as the size of the dictionary grows. The most common technique for gaining fast access to a dictionary utilizes a hash table (Knuth

(1973)), McIlroy (1982)). Other standard search techniques, such as tries (Knuth (1973)), frequency ordered binary search trees (Knuth (1973)), and finite state automata (Aho and Corasick (1975)), have been used to reduce dictionary search time. Dictionary partitioning schemes were suggested by Peterson (1980) to store a small set of the most frequently used words in cache or regular memory, and tens of thousands of less frequently used words in secondary memory. Many spell checkers only contain root forms of words to avoid storage of all possible morphological variants of individual words. However, simple affix stripping methods may lead to false acceptances when affixes are stripped without regard to syntax.

Error Correction

Most prior error correction techniques focus on isolated words, without taking into account information that might be extracted from the linguistic or textual context in which the string appears. Since about 80% of errors tend to be single error misspellings such as insertion, deletion, substitution, or transposition of letters (Damerau (1964)), current techniques concentrate on correcting single error misspellings. The main techniques for error correction are as follows.

Minimum edit distance techniques that compute a minimum edit distance between a misspelled string and a dictionary entry (Damerau (1964), Levenshtein (1966), Wagner (1974)): These techniques are employed by most error correction algorithms.

Similarity key techniques that map every string into a key such that similarly spelled strings have identical or similar keys (Pollock and Zamora (1984)): A key is computed for a misspelled string and provides a pointer to all similarly spelled words in the dictionary.

Rule based techniques that attempt to represent knowledge of common spelling error patterns in the form of rules: The rules provide a guideline for transforming misspellings into valid words (Yannakoudakis and Fawthrop (1983a), (1983b)). Multiple candidates are ranked by predefined estimates based on the applicability of the rules.

Other techniques, including *n-gram based techniques*, *probabilistic techniques* (Toussaint (1978), Hull and SriHari (1982)), and *neural net techniques*: The techniques are derived from methods for optical character recognition (Riseman and Hanson (1974)), command language interfaces (Durham, Lamb, and Saxe (1983), Hawley (1982)), database retrieval (Parsaye, Chignell, Khoshafian, and Wong (1990)), name and address correction (Cherkassky and Vassilas (1989), Gersho and Reiter (1990)), and text-to-speech synthesis (Kukich (1992), Tsao (1990)).

The spell checking method described in this paper employs dictionary lookup techniques and logic computations to detect and correct errors. A key difference to prior methods is that our spell checking method learns individual user behavior and preferences. The next section gives an overview of the method.

3. Spell Checking System

The spell checking system relies on the following approach.

When a user invokes the system in a given domain for the first time, the system relies on a *general dictionary* to determine spelling errors and suggest corrections. The dictionary has in excess of 200,000 entries and thus contains most common words of the English language. The spell checking process creates and maintains a second dictionary called *user dictionary*. There is a third, typically very small, *excluded words dictionary*

that is defined and updated by the user. It contains words that the user will not accept as correct regardless of circumstances. We present details about the three dictionaries in Sections 4–6.

Suppose the system processes some text. For each word, it checks whether the word

(3.1) occurs in the excluded words dictionary, or
(3.2) occurs in the user dictionary, or
(3.3) occurs in the general dictionary, or
(3.4) can be constructed from a root word in the general dictionary, or
(3.5) likely is the result of a spelling or typing error, or
(3.6) likely is considered correct by the user.

The actions taken in the six cases are as follows.

Case (3.1): Word in Excluded Words Dictionary

If the word is found in the excluded words dictionary, it is known to be incorrect by user definition. We indicate that fact to the user, ask for a replacement word, and check (3.1)–(3.6) for the replacement word.

Case (3.2): Word in User Dictionary

If (3.1) does not apply and the word is in the user dictionary, it is considered to be correct. That conclusion triggers updating of a *word usage index*, but does not involve any interaction with the user. The updating formula for the index is given in Section 7.

Case (3.3): Word in General Dictionary

Suppose (3.1) and (3.2) do not apply. If the word is found in the general dictionary, we store the word and the related syntactic classification such as adjective, noun, transitive or intransitive verb, etc. in the user dictionary. We also check with the procedure for (3.4) below whether the word can be constructed from some root word. If that is so, we amend the user dictionary as described for (3.4).

Case (3.4): Construction of Word

Suppose (3.1)–(3.3) do not apply. We determine with the aid of a logic module whether some construction rule can produce the given word from some root words of the general dictionary. We only consider the most frequently used construction rules, for example, the rules for adding the ending “er” or “ier” to express comparative form, or for adding “est” or “iest” for superlative form. On the other hand, we do not consider the construction that adds the ending “er” to convert a verb to a noun.

Once we have identified construction rules that derive the given word from some root words, we store the given word, the root words, and the syntactic classification implied by the constructions in the user dictionary.

The restricted use of construction rules has the positive effect that it is highly unlikely that we derive words that actually are not part of the English language. Thus, it is equally unlikely that we contaminate the user dictionary with incorrect words. On the negative side, we may not recognize a valid construction. But that is a small problem, since the general dictionary is large and since we may fail to recognize a valid construction at most once, when the given word is encountered for the first time.

Case (3.5): Spelling or Typing Error

Define a word to be *unknown* if it is not in any one of the dictionaries and cannot be constructed from some root word. Thus, a word is unknown if (3.1)–(3.4) do not apply. Suppose such a word is at hand.

We check whether the word likely is the result of a spelling or typing error. The evaluation is carried out via a logic module that considers a number of possible errors and the associated corrections. Assume that at least one such error is found. The correct words corresponding to the errors are sorted according to weights that depend on the likelihood that the user has made such errors and on the word usage index of the correct words. If there are more than three correct words, all but the three most likely ones are discarded. The remaining words are proposed to the user in the just determined order. The user either accepts one of these words, or types in some other word, or declares that the word supposedly in error is correct. The three subcases lead to the following system actions.

In the first subcase, the system records the error associated with the accepted word in a *user history file*. We explain the role of that file in Section 8.

In the second subcase, we treat the word supplied by the user as another given word and check (3.1)–(3.5). The resulting actions are as described above, with one exception. If (3.1)–(3.4) do not apply, the system warns the user that the replacement word possibly is incorrect, but otherwise does not carry out any error analysis. The user may choose to ignore the warning, or may offer a different replacement word.

The third subcase does not trigger any error analysis.

Regardless of which subcase applies, an additional step is taken if the correct word, that is, the word accepted or typed in by the user, is not in the user dictionary. In that situation, we add the correct word to the user dictionary and check whether the correct word can be created by one of the construction rules from some root words as described in (3.4). If a construction rule applies, the applicable root words are stored together with the given word in the user dictionary.

Choice (3.6): Word Unknown but Likely Correct

If none of the preceding cases applies, we guess that the user considers the given unknown word to be correct and ask the user to confirm that conjecture. The user may agree or type in another word. In the first subcase, we add the word to the user dictionary. In the second subcase, we check (3.1)–(3.4) for the replacement word. If (3.1)–(3.4) do not apply, the system warns the user that the replacement possibly is incorrect. The user may choose to ignore the warning or may offer a different replacement word.

The above summary of the various system actions is not yet complete, since it does not provide details about the three dictionaries, the word usage index, the user history file, the word construction rules, and the error detection and correction process. We fill that gap in the subsequent sections.

4. General Dictionary

At present, the general dictionary contains 215,707 words and their syntactic classification. The dictionary originally was created using *MobyWords* (1991) and *MobyParts-of-Speech* (1991). We have made changes by manual efforts and have also checked parts of the dictionary against *Webster's Ninth New Collegiate Dictionary* (1989).

The classification of a word is encoded as a character string such that each character represents one possible part of speech for the word. A total of 47 classification cases are used.

A multi-hashing search algorithm is used to access the general dictionary. The general dictionary is partitioned into buckets so that words beginning with the same three characters are located in the same bucket. The first level hashing maps an input string into its bucket. The second level hashing maps a string within a bucket into the possible location in the general dictionary.

We allow different hash functions for the various buckets. To simplify the recreation of the entire access process, we have designed software that automatically chooses the bucket sizes and the hash functions so that worst-case access time is very low. This feature has proved to be handy for other applications where we sometimes increased the general dictionary by more than 50%.

Access time for a word is about 0.15 milliseconds on a computer with 42 mips. That speed is ample for the spell checking application.

5. User Dictionary

The user dictionary contains each word that has been encountered so far and is correct for one of the following two reasons: It is a word or a derivative of a word in the general dictionary or has been declared to be correct by the user. Together with each word, the user dictionary contains the following information.

(5.1) If applicable, a root word that via some construction rule can be converted to the given word.
(5.2) A word usage index that measures the frequency with which the given word has occurred in texts recently processed by the system. The computations producing the index are explained in Section 7.

The data listed below are not used for spell checking, but are utilized and updated during syntax checking and semantical checking. We will describe details in subsequent publications.

(5.3) The syntactic classification of the given word.
(5.4) The relative frequencies with which the given word previously has been classified as adjective, noun, transitive or intransitive verb, etc.

We use a standard balanced tree structure to store, augment, and retrieve entries of the user dictionary. Since that dictionary for a given domain rarely has more than 20,000 entries, storage and retrieval of entries is very fast.

6. Excluded Words Dictionary

The excluded words dictionary contains words that the user considers incorrect regardless of circumstances. For example, if the user favors American English over British English, the dictionary might contain “colour”.

During spell checking, the dictionary is stored and accessed like the user dictionary, using a standard balanced tree structure.

Updating of the excluded words dictionary is controlled by the user. In a typical situation, the user discovers a word in a file that should not have been used but has been accepted by the spell checking system. The user then adds that word to the excluded words dictionary to prevent future acceptance of the word.

7. Word Usage Index

The word usage index, say q, of a word is utilized in case (3.5) when the weights of replacement words for an incorrect word are computed. Details of the weight calculations are included in Section 10. Here we describe how q is interpreted and updated. Generally, the index lies between -1000 and 1000.

If the word is not in the user dictionary, the value of q is taken to be zero.

If the word is in the user dictionary and q is nonpositive, the word has not occurred during the processing of the last $|q| + 1$ files.

If the word is in the dictionary and q is positive, the word has occurred with a certain frequency during the processing of recent files. Generally, the larger the value of q, the higher the frequency.

The updating of q is as follows. Suppose a text is being processed. Each time a given word is encountered, its index q is updated, say to q', according to the formula

$$q' = \begin{cases} 2 & \text{if } q \le 0 \\ q+1 & \text{if } 0 < q < 1000 \\ 1000 & \text{if } q = 1000 \end{cases} \tag{7.1}$$

When processing of a text is completed, the current index q of each word is reduced to another value, say q'', according to an empirically determined formula that has worked well for the weight calculations of Section 10. The formula is

$$q'' = \begin{cases} -1000 & \text{if } q = -1000 \\ q-1 & \text{if } -1000 < q < 100 \\ \lfloor q - 0.0005q^2 \rfloor; & \text{if } 100 \le q \le 1000 \end{cases} \tag{7.2}$$

Evidently, for $q = -1000$, there is no reduction, and for $-1000 < q < 100$, the index is reduced by 1. For $100 \le q \le 1000$, the size of the reduction grows with q, and reaches 500 for $q = 1000$.

8. User History File

The user history file lists the types of spelling and typing errors made by the user. Since a user may change spelling and typing behavior over time, the user history file is configured as a sliding window that contains the most recently made errors. The file is utilized to compute the likelihoods with which the user makes various types of errors. These likelihoods enter the computations of the logic module for determining spelling and typing errors; see Section 10.

9. Construction of Words

In the English language, a number of construction rules can derive words from root words. We have selected a certain subset so that we can derive most if not all words that are not in the general dictionary but that can be derived from a root word in the dictionary.

Selected Construction Rules

The selected rules cover the addition of "ed" (for past tense of verbs), "er" (for comparative form of adjectives and adverbs), "est" (for superlative form of adjectives and adverbs), "ing" (for present participle of verbs), and "s" (for plural of nouns and third person singular of verbs). Other rules—for example, for addition of "ly" to an adjective to derive an adverb—are not needed since the general dictionary already contains all words that may be derived by them. We have encoded the selected rules in a logic formulation that has 255 propositional variables, 434 clauses in conjunctive normal form, and 1012 literals. Besides the construction rules, the logic formulation also defines the related syntactic classifications.

Space constraints prohibit a detailed discussion of the logic encoding of the selected rules, so we just include three example rules and display the logic clauses that represent a portion of one of the rules.

(9.1) A transitive verb suffixed with "d", "ed", or "ied" is a transitive verb past tense and past participle tense.

(9.2) A root word ending with "s", "sh", "x", or with "y" preceded by a consonant, should not be directly suffixed with "s".

(9.3) A root word ending with "e" and with a preceding consonant or "u", or containing only one vowel and ending with one consonant, should not be directly suffixed with "ing".

We discuss the logic clauses representing the following portion of the rule (9.2).

(9.4) A root word ending with "s", "sh", "x" should not be suffixed with "s".

We define a predicate $root()$, which represents the classification of a given root word, on the set $root\_class = \{noun, verb\_transitive, verb\_intransitive\}$. Since the set is finite, the predicate $root()$ effectively represents three propositional variables, which are $root(noun)$, $root(verb\_transitive)$, and $root(verb\_intransitive)$.

We need a second predicate, $root\_end()$, which represents the letter or string of letters terminating the root word and which is defined on the set $end\_string = \{s, sh, x\}$. Effectively, the predicate defines the three propositional variables $root\_end(s)$, $root\_end(sh)$, and $root\_end(x)$.

In addition, we need three propositional variables: $word\_end\_s$, which tells whether the given word ends in "s"; $wrong\_end\_s$, which specifies whether the word ends incorrectly in "s"; and $violate\_rule$, which represents whether the word is incorrectly constructed.

With these predicates and propositional variables, (9.4) may be encoded as follows.

FOR ALL c IN $root\_class$
FOR ALL e IN end_string
  IF $root(c)$ AND
    $root\_end(e)$ AND
(9.5) $word\_end\_s$
  THEN $wrong\_end\_s$.

IF $wrong\_end\_s$ OR ... (other variables connected by OR)
THEN $violate\_rule$.

Solution via Logic Minimization

Of the 255 variables of the logic formulation, 23 variables represent conclusions about construction of the given word from some root word. For example, the variable *violate_rule* defined above is one of the conclusion variables. The other 232 variables represent the input variables describing characteristics of the given word or are intermediate variables. Examples of input variables are the variable *word_end_s* and the predicates *root*() and *root_end*(). An example of the intermediate variables is *wrong_end_s*.

One may decide whether a given word is improperly constructed as follows. First, the input variables are assigned *True/False* values depending on the word. Second, each conclusion variable, among them *violate_rule*, is processed in turn, by first assigning *False* to the conclusion variable and then checking the resulting logic formulation for satisfiability. For each conclusion variable so processed, the conclusion is valid if the logic formulation, with *True/False* values assigned as just described, is unsatisfiable, and is not valid otherwise.

We given an example. Suppose that the given word is "blesss" and that we partition that word into the terminating letter "s" and the root "bless". Note that "bless" is a transitive verb and is not a noun or intransitive verb. Furthermore, both the given word "blesss" and the root word "bless" end in "s". Hence, the input variables *root*(*verb_transitive*), *root_end*(*s*), and *word_end_s* receive the value *True*. All other input variables receive the value *False*.

To settle whether the word "blesss" is improperly or properly constructed, we take each conclusion variable in turn, set it to *False*, and decide satisfiability of the resulting logic formulation. For the case of the conclusion variable *violate_rule*, that process results in unsatisfiability, as the reader may easily verify using the given *True/False* values in the clauses of (9.5). Hence, the word "blesss" violates a construction rule.

Since there are 23 conclusion variables, in principle one would have to solve up to 23 satisfiability (SAT) instances to decide which ones of the conclusions do apply. It turns out that we can obtain the same information much faster by solving a certain logic minimization (MINSAT) instance, as described in Truemper (1998). We sketch the approach following the definition of MINSAT.

A MINSAT instance is a SAT instance where two cost values have been assigned to each variable. For a given variable, one of the two costs is incurred if the variable takes on the value *True*, while the other cost is incurred if the value takes on the value *False*.

One solves a MINSAT instance as follows. One either produces a satisfying solution that minimizes the total cost resulting from the *True/False* values of the satisfying solution,

or concludes that the instance is unsatisfiable.

For the case at hand, we assign to each conclusion variable—for example, to the variable *violate_rule*—a cost of 1 if the variable takes on the value *True*, and a cost of 0 for taking on *False*. For each one of the remaining variables, we declare the cost values for both *True* and *False* to be 0.

The selected cost values have the effect that any MINSAT solution assigns *False* to the conclusion variables as much as possible. Furthermore, if in a MINSAT solution a conclusion variable has the value *False*, then that variable cannot be proved to have necessarily the value *True*. Thus, we can ignore all conclusion variables with value *False*, and need to carry out the usual theorem proving procedure via SAT instances only for the remaining conclusion variables, that is, for those with value *True*.

Implementation

We solve the above described logic problems as follows. The logic formulation is compiled and evaluated by the *Leibniz System* (1997). That system contains a so-called *program generator* that derives from the logic formulation a solution algorithm. With that algorithm, the Leibniz System supplies a worst-case bound on the solution time. For example, for the logic formulation containing the construction rules, that bound is 0.02 sec, assuming a machine speed of 42 mips. The solution algorithm is executed via the *execution module* of the Leibniz System, which may be called from any C program.

10. Error Detection and Correction

Recall that a word is unknown if (3.1)–(3.4) do not apply. Deciding whether a given unknown word is incorrect is a nontrivial matter. We answer that question approximately by determining whether the word can be explained as a misspelled or mistyped word. That is, if we find such an explanation, we declare the word to be in error. Otherwise, we deem the word to be correct.

The typical spelling and typing errors are well known, see for example Berry (1961), Shaw (1962), and Blumenthal (1981).

The most frequently occurring spelling errors consist of violations of general rules for adding a suffix (rules for doubling final consonant, dropping silent "e", retaining silent "e", replacing "y" by "i") and particular spelling mistakes ("ei" $\leftrightarrow$ "ie", "de" $\leftrightarrow$ "di", omitting double consonant, doubling single consonant).

We consider the following typing errors, where *neighbor letter* refers to any letter that on the keyboard is close to a given letter: Transposing two letters, transposing a blank space with a letter, repeating a letter, omitting a letter, inserting a letter that is a neighbor of a given letter, typing an incorrect letter that is a neighbor of the required letter.

The user history file contains the types of spelling and typing errors most recently made by the user. We rely on that information to decide which type of error, if any, has been made when a given unknown word is suspected to be in error.

Define a *candidate word* to be any word the user may have intended to type. We determine and rank candidate words as follows. First, we identify errors that most likely produced the incorrect word and search for candidate words in light of those errors. Second, we rank candidate words according to certain weights. Details are as follows.

Finding Likely Errors and Candidate Words

We use an empirically derived logic formulation to determine for a given incorrect word the possible spelling and typing errors and their likelihoods of occurrence. The logic formulation relates the recent error history of the user to the possible spelling and typing errors. Space constraints prevent a complete description of the formulation, so we just summarize three facts represented by some of the logic clauses.

(10.1) Suppose that the incorrect word does not contain repeated letters, or consecutive letters that are neighbors, or "de", or "di", or "ie", or "ei". Further, suppose that the word does not end in a silent "e", that the text does not have two consecutive unknown words including the one being investigated, and that the word does not occur elsewhere in the text. Then the error almost surely is a transposition of two letters.

(10.2) Suppose that the word contains "de", or "di", or "ie", or "ei", or ends in a silent "e", or that the text contains two consecutive unknown words including the given one. Further, suppose that the word contains two consecutive letters that are neighbors. Then the error sometimes is an insertion of an additional letter.

(10.3) If the incorrect word contains repeated letters and if the user history file contains several cases where the user erroneously typed repeated letters, then the word frequently contains an erroneous repetition of letters.

Similarly to the logic formulation of the word construction rules, we use logic minimization and algorithms compiled by the Leibniz System (1997) to deduce a list of the errors that may have produced the given word, and the likelihoods with which these errors apply. The logic formulation has 116 propositional variables, 328 clauses in conjunctive normal form, and 982 literals. The Leibniz System produces for that logic minimization problem a solution algorithm with worst-case time bound of 0.01 sec, assuming a machine speed of 42 mips.

The conclusions derived via the logic formulation define the desired list of errors. We sketch how those conclusions and the associated likelihoods are computed.

First, terms of the logic facts expressing the degree of certainty such as "almost surely", "frequently", and "sometimes" are converted to numerical likelihood values; for example, the cited example terms result in 100%, 75%, and 25%, respectively. These numerical values are associated with the logic clauses representing the logic facts.

Second, the likelihood with which a conclusion holds is determined by the following reasonable rule adapted from Fuzzy Logic: It is the largest number l such that the clauses with likelihood value greater than or equal to l prove the conclusion. Then the conclusion is declared to hold with likelihood l.

Third, the Leibniz System relies on the just cited rule to determine conclusions and the associated likelihood values by the following iterative process, where initially $l = 100\%$.

> Delete all clauses with likelihood less than l. Solve the resulting MINSAT instance. Declare each conclusion that is proved by the solution of that instance and has not been proved for a higher likelihood value, to hold at level l. If l is at the lowest level, stop; otherwise, reduce l to the next lower value and begin the next iteration.

Using the list of errors and the associated likelihoods so computed, we derive from the given word candidate words. That is, we postulate each error of the list in turn and, with the aid of the user dictionary and the general dictionary, establish candidate words. To assure a speedy derivation of the candidate words, we do not construct candidate words from root words of the general dictionary. As a result, we may not obtain a needed candidate word. We claim that, over time, occurrences of that event become less frequent and asymptotically cease entirely. Indeed, the user dictionary contains all words ever employed by the user and asymptotically contains the entire vocabulary of the user. Thus, all needed candidate words eventually will be found in the user dictionary.

Ranking Candidate Words

We assign to each candidate word a weight w that is computed from the word usage index q, the likelihood l with which the user made the error that converts the candidate word to the given word, and a certain scaling factor α. The formula for w is

$$w = (1 + \alpha)q + 100l \tag{10.4}$$

The value of α is dynamically adjusted so that, according to past experience, the candidate word with the highest weight most likely is the correct replacement word.

If more than three candidate words are found, all but the three with largest weights are discarded. The remaining words are suggested to the user, in the order given by the weights. The user may accept one of the candidate words, type in some other word, or may declare the given word to be correct. In our implementation of the spell checking process, acceptance of the candidate word with highest weight is particularly convenient; it just requires pressing of the "Return" key.

11. Learning Special Vocabularies

Some text files, for example TEX files, contain special control words that the system might diagnose as incorrect. To speed up learning of such special vocabularies, one may declare a given text to be entirely correct. The system processes such a text without any user interaction, by assuming that any word not falling into one of the cases (3.1)–(3.4) is correct.

12. System Implementation and Test Results

The spell checking system has been implemented and used for several years at a number of universities and research institutes. It is called the *Laempel System*. We have carried out extensive testing of the system using various files. In this section, we summarize the conclusions.

Execution Times

The system has extraordinary speed. For example, spell checking a TEX file with 24,541 words for the first time, with no initial user dictionary available, required about 20 sec on a 42 mips computer. Afterward, each spell checking run of that file required about 2 sec.

Error Detection

We have compared the Laempel System with the program Ispell (1993). The conclusions are as follows.

In a number of test runs, the Laempel System found about 4% more spelling and typing errors than Ispell. This conclusion is confirmed qualitatively by R. Borndörfer of the Konrad-Zuse-Institut for Information Technology, Berlin, Germany, who performed extensive tests comparing the Laempel System with Ispell. He applied the Laempel System to a number of files that had been declared correct by Ispell, and found several mistakes in virtually everyone of these files.

On the other hand, we have yet to encounter the situation where the Laempel System accepted an erroneous word. This is not surprising, since the system has a highly reliable general dictionary and utilizes word construction rules very conservatively.

A good spell checker not only identifies all erroneous words, but also guesses most words that it does not know but that are correct, to be correct. In tests using persons' names, compound words, and control commands that are not in the general dictionary or the user dictionary, the Laempel System regarded such words as correct 82% of the time, while Ispell did so 19% of the time.

Error Correction

We evaluated the ability of the two systems for suggesting correct replacement words with test files typed by two users. The files contained a total of about 15,500 words, of which 53 were wrongly spelled everyday words. We purposely ignored special words such as names, compound names, and control commands of the files since we knew already that the Laempel System is more adept at handling such unknown but correct words than Ispell.

To make the comparison as fair as possible, we started the Laempel System on the files of each user with a vacuous user dictionary and error history file. Hence, all learning had to take place during the processing of the files of each user.

The Laempel System suggested correct replacement words for 51 of the 54 erroneous words. It could not find a replacement word for 3 of the 54 words.

Ispell suggested correct replacement words for 52 of the errors. It could not find a replacement word for 2 of the 54 words. This is better than the Laempel System by one word.

The Laempel System proposed on average 1.2 replacement words for each one of the 51 erroneous words, and, by design, never more than 3 words. The suggested first replacement choice was correct in 96% of the cases.

In contrast, Ispell suggested on average 3.0 words for each one of the 52 erroneous words, and in one case as many as 31 words. In fact, in 16% of the cases, Ispell proposed at least 4 words. The suggested first replacement choice was correct in 79% of the cases.

Given these results, we are justified in claiming that the Laempel System is fast and that, based on learning of user behavior and preferences, the system reliably identifies incorrect words, very frequently proposes correct replacements, and very frequently identifies unknown correct words as correct.

References

Aho, A. V., and Corasick, M. J. (1975), Efficient string matching: An aid to bibliographic search, *Communications of the ACM* 18 (1975) 333-340.

Berry, T. E. (1961), *The Most Common Mistakes in English Usage*, Chilton, Philadelphia, 1961.

Blumenthal, J. C. (1981), *English 3200*, 3rd edition, Harcourt Brace Jovanovich, New York, 1981.

Cherkassky, V., and Vassilas, N. (1989), Back-propagation networks for spelling correction, *Neural Net* 1 (1989) 166-173.

Damerau, F. J. (1964), A technique for computer detection and correction of spelling errors, *Communications of the ACM* 7 (1964) 171-176.

Durham, I., Lamb, D. A., and Saxe, J. B. (1983), Spelling correction in user interfaces, *Communications of the ACM* 26 (1983) 764-773.

Gersho, M., and Reiter, R. (1990), Information retrieval using self-organizing and heteroassociative supervised neural networks, *Proceedings of IJCNN*, San Diego, California, June 1990.

Hawley, M. J. (1982), Interactive spelling correction in Unix: The METRIC Library, *AT&T Bell Labs Tech. Mem.*, August 1982.

Hull, J. J., and SriHari, S. N. (1982), Experiments in text recognition with binary *n*-gram and Viterbi algorithms, *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-4 (1982) 520-530.

Ispell (1993), version 4.0, Free Software Foundation, Cambridge, Massachusetts, 1993.

Knuth, D. E. (1973), *The Art of Computer Programming: Sorting and Searching*, (Volume 3), Addison-Wesley, Reading, Massachusetts, 1973.

Kukich, K. (1992), Spelling correction for the telecommunications network for the deaf, *Communications of the ACM* 35 (1992) 80-90.

Leibniz System (1997), Version 4.2, Leibniz, Plano, Texas, 1997.

Levenshtein, V. I. (1966), Binary codes capable of correcting deletions, insertions and reversals, *Soviet Physics, Doklady* 10 (1966) 707-710.

McIlroy, M. D. (1982), Development of a spelling list, *IEEE Transactions on Communications* COM-30 (1982) 91-99.

Microsoft Word (1994), version 6.0 for Windows, Microsoft, Redmond, Washington, 1994.

MobyWords (1991), (MS-DOS Version 1.3), Grady Ward, 1991.

MobyParts-of-Speech (1991), (MS-DOS Version 1.3), Grady Ward, 1991.

Morris, R., and Cherry, L. L. (1975), Computer detection of typographical errors, *IEEE Transactions on Professional Communication* PC-18 (1975) 54-63.

Parsaye, K., Chignell, M., Khoshafian, S., and Wong, H. (1990), Intelligent database, *AI Expert* 5 (1990) 38-47.

Peterson, J. L. (1980), Computer programs for detecting and correcting spelling errors, *Communications of the ACM* 23 (1980) 676-684.

Pollock, J. J., and Zamora, A. (1984), Automatic spelling correction in scientific and scholarly text, *Communications of the ACM* 27 (1984) 358-368.

Riseman, E. M., and Hanson, A. R. (1974), A contextual postprocessing system for error correction using binary n-grams, *IEEE Transactions on Computers* C-23 (1974) 480-493.

Shaw, H. (1962), *Errors in English and Ways to Correct Them*, Barnes & Noble, New York, 1962.

Toussaint, G. T. (1978), The use of context in pattern recognition, *Pattern Recognition* 10 (1978) 189-204.

Truemper, K. (1998), *Effective Logic Computation*, Wiley-Interscience, New York, 1998.

Tsao, Y. C. (1990), A lexical study of sentences typed by hearing-impaired TDD users, *Proceedings of the 13th International Symposium on Human Factors in Telecommunications*, Turin, Italy, 1990, pp. 197-201.

Wagner, R. A. (1974), Order-n correction for regular languages, *Communications of the ACM* 17 (1974) 265-268.

Webster's Ninth New Collegiate Dictionary (1989), Macintosh CD-ROM, Merriam-Webster, Inc., and Highlighted Data, Inc., Washington, DC, 1989.

Yannakoudakis, E. J., and Fawthrop, D. (1983a), The rules of spelling errors, *Information Processing & Management* 19 (1983) 87-99.

Yannakoudakis, E. J., and Fawthrop, D. (1983b), An intelligent spelling error corrector, *Information Processing & Management* 19 (1983) 101-108.

Zamora, E. M., Pollock, J. J., and Zamora, A. (1981), The use of trigram analysis for spelling error detection, *Information Processing & Management* 17 (1981) 305-316.

Zhao, Y. (1996), *Intelligent Text Processing*, thesis, University of Texas at Dallas, Richardson, Texas, August 1996.