Chapter 11 [1]

LEARNING TO FIND CONTEXT BASED SPELLING ERRORS

Hisham Al-Mubaid*, Klaus Truemper**
** University of Houston - Clear Lake*
Department of Computer Science
email: hisham@cl.uh.edu
*** Department of Computer Science*
University of Texas at Dallas
Richardson, TX 75083-0688, U.S.A.
Email: truemper@utdallas.edu

Abstract A context-based spelling error is a spelling or typing error that turns an intended word into another word of the language. For example, the intended word "sight" might become the word "site." A spell checker cannot identify such an error. In the English language—the case of interest here—a syntax checker may also fail to catch such an error since, among other reasons, the parts-of-speech of an erroneous word may permit an acceptable parsing. This chapter presents an effective method called Ltest for identifying the majority of context-based spelling errors. Ltest learns from prior, correct text how context-based spelling errors may manifest themselves, by purposely introducing such errors and analyzing the resulting text using a data mining algorithm. The output of this learning step consists of a collection of logic formulas that in some sense represent knowledge about possible context-based spelling errors. When, subsequently, testing text is examined for context-based spelling errors, the logic formulas and a portion of the prior text are used to analyze the case at hand and to pinpoint likely errors. Ltest has been added to an existing software system for spell and syntax checking. We have conducted tests involving mathematical, technical, and general texts. On the average, Ltest found 68% of context-based spelling errors in large texts and 87% of such errors in small texts. These detection rates are relative to words for which training was possible using the prior text. On the other hand, the number of false-positive diagnoses was small, involving on average 23 word instances (= 0.7% of the possible error instances) of a large text and 1 word instance (= 8% of the possible error instances) of a small text.
These statistics indicate that the method is effective for the recognition of the majority of context-based spelling errors considered in the experimental tests.

Keywords Semantics of natural language, algorithms, learning typing/spelling errors, Representation of word context, data mining, learning logic.

[1] Triantaphillou, E. and G. Felici (Eds.), **Data Mining and Knowledge Discovery Approaches based on Rule Induction Techniques**, Massive Computing Series, Kluwer Academic Publishers, Dordrecht, The Netherlands, pp. ???-???, 2004.

1. INTRODUCTION

Finding a spelling or typing error is easy if the erroneous word is not part of the language, since then a spell checker can point out such a *non-word error*. The detection problem is harder if the erroneous word is part of the language. An example is the misspelling of the intended word "sight" as "site." Such an error can be detected by examining the context of the word. Accordingly, it has been called a *context-based spelling error* (Golding (1995), Golding and Roth (1996, 1999), Golding and Schabes (1996)).

It is convenient that throughout most of the chapter we make the following two assumptions. First, we assume that the event of a context-based spelling error is relatively rare, and that the user is unlikely to make the same mistake several times in the same document. For example, if the user misspells "sight" as "site," then this error is assumed to be due to a momentary lapse and not due to user ignorance regarding the spelling of "sight." Second, we assume that, for any erroneous word instance introduced by a context-based spelling error, the text also contains an instance of the correct word. The two assumptions are mostly but not always satisfied. For example, a person may confuse some words and make some errors repeatedly. For example, such confusion could exist about "its" versus "it's" or "complement" versus "compliment." As a second example, a word may occur just once in a text, and that single occurrence may be mistyped or misspelled. Toward the end of the chapter, in Sections 4 and 5, we describe extensions of the method that do not require the two assumptions.

In contrast to spell checkers, a syntax checker may possibly detect a context-based spelling error. However, there is no guarantee of such detection, since, among other reasons, the parts-of-speech of the erroneous word may permit an acceptable parsing of the sentence. For example, if "site" displaces "sight" in the sentence "It was a beautiful sight," then the resulting sentence "It was a beautiful site" has an acceptable parsing. Indeed, the latter sentence is meaningful and by itself gives no clue that the word "site" is out of place. Here are a few additional examples of intended and erroneous words: bay–pay, fair–fare, for–four, its–it's, lead–led, quiet–quite, them–then, there–three.

Error detection by a syntax checker likely is difficult if the text contains many special terms, symbols, formulas, or conventions whose syntactic contribution cannot be established without a complete understanding of the text; examples are mathematical TEX or LATEX texts. For such texts, as well as for texts that do not contain such complicating aspects, this chapter offers an effective technique for identifying the majority of context-based spelling errors without the need to fully understand the text. The main results are as follows.

(1) A new way of encoding, for a given occurrence of a word w, the structure of the neighborhood of the occurrence and the connection with other occurrences of w and their neighborhoods. The encoding uses the text under investigation as well as a second text that acts as a reference text.

(2) A new way of learning from prior, correct text how context-based spelling errors can be recognized. This step uses the encoding of (1) and an existing data mining algorithm. It produces a set of logic formulas that contain insight into context-based spelling errors.

(3) A new way of employing the logic formulas of (2) to identify likely context-based spelling errors in testing texts. The scheme accepts both small and large testing texts, and it handles unusual cases such as erroneous words never seen in the learning phase.

The method is called *Ltest* (*Logic test*) as it uses logic formulas to test for errors. Ltest has been added to an existing spell and syntax checking system. We have conducted tests involving mathematical book chapters typeset in $\TeX$, technical papers typeset in $\LaTeX$, and newspaper texts in two subject areas. For the learning step we randomly selected prior texts from the given domain. These texts averaged 30,012 word instances. The testing texts consisted of some large texts averaging 7,138 word instances and some small texts averaging 105 word instances. The latter texts were introduced to see if the method can find context-based spelling errors when a testing text does not provide much insight into the usage pattern of words. Although the average prior text had about four times the size of the average large testing text, roughly half the word usage in the testing texts was not sufficiently represented in the prior texts to allow learning of such usage and subsequent error checking. Such representation does not require much: If Ltest is to learn the difference between a given word and a given erroneous word, then both words must occur at least three times in both the *training text* and the *history text*, which Ltest obtains by splitting the prior text. Though this requirement is mild, for the average large text just 3,162 possible error cases out of a total of 7,360 possible error cases, or 43%, could be tested. For the average small text, 12 possible error cases out of a total of 28 possible error cases, or 43%, could be tested. It is shown later that these percentages can be boosted close to 100% by a suitable augmentation of the training text and the history text. We did not carry out such augmentation for the tests since such a change might have introduced a bias. Instead, we evaluated the performance of Ltest on the possible error cases for which the prior texts had allowed learning. We introduced such errors randomly into the testing texts. On average, Ltest detected 68% of these errors in large texts and 87% in small texts. The testing texts with the errors were also checked by the syntax checker of the system, in a separate step. The syntax checker, by itself, performed poorly, finding only 12% of the errors in large texts and 4% in small texts. Combined use of Ltest and the syntax checker—which is the way the entire process has been implemented—boosted the detection rate for large texts to 72%, but did not improve the rate of 87% for small texts. The difference in performance between samll and large texts is due to two factors: – large texts normally involve numerous special terms, symbols, formulas, and conventions that make error detection more complicated than small texts, –

small testing texts are examined by the classifiers that are created for large testing texts, so classifiers perform better on the small texts.

Define a diagnosis to be *false-positive* if the method estimates a correct word instance to be in error. Clearly, user acceptance of the method requires that at most a few false-positive diagnoses are made. This requirement was satisfied in the test cases, since false-positive diagnoses occurred on average for 23 word instances, or 0.7% of the 3,162 tested instances of a large text, and for 1 word instance of a small text. The 1 false-positive diagnosis for the average small testing text represents 8% of the tested instances, which is high but not important since the number of such cases, which is 1, is small. We ran another experiment on the method using texts for which the two leading prior methods, which are *BaySpell* (Golding (1995)) and *WinSpell* (Golding and Roth (1999)), had produced results. In the experiment, Ltest outperformed both methods by classifying 95.6% of the considered word instances correctly. BaySpell and WinSpell achieved 89.9% and 93.5% accuracy, respectively. Testing time is very low in these experiments: in the order of 2 minutes for large texts, and in order of 10 seconds for small texts, more details in Section 4 and in tables 3 and 6.

Taken together, the high detection rates and the low number of false-positive diagnoses for both large and small texts make the method an effective tool.

The rest of the chapter proceeds as follows. Section 2 discusses previous work. Section 3 describes the method. Section 4 discusses the implementation of the method and the computational results. Section 5 outlines extensions. Section 6 summarizes the main points of the chapter. Appendices A to D contain technical details of some of the steps.

2. PREVIOUS WORK

A number of methods have been developed for the detection of context-based spelling errors. The research up to 1992 is covered in the survey by Kukich (1992). The methods proposed since then use a Bayesian approach (Golding (1995)) that may be combined with part-of-speech trigrams (Golding and Schabes (1996)), transformation-based learning (Mangu and Brill (1997)), latent semantic analysis (Jones and Martin (1997)), differential grammars (Powers (1997)), lexical chains (St-Onge (1995), Hirst and St-Onge (1995), Budanitsky (1999), Budanitsky and Hirst (2001)), and Winnow-based techniques (Golding and Roth (1996, 1999), Roth (1998)). The two leading prior methods are the statistics-based *BaySpell* (Golding (1995)) and the Winnow-based *WinSpell* (Golding and Roth (1999)).

The Bayesian method (Golding (1995)) handles context-based spelling correction as a problem of ambiguity resolution. The ambiguity is modeled by confusion sets. The Bayesian method uses decision lists to choose the proper word from the confusion set. It also relies on classifiers for two types of features: context-words and collocations. The method learns these features from a training corpus of correct text. The testing process starts with a list of features sorted by decreasing strength and traverses the entire list to combine evidences from all matching features in a given context and target word. In the experiment reported in Golding (1995), 18 confusion sets are used. The performance ranges from 45% to 98% with an average of 82% of the words classified correctly. Golding uses 1-Million-Word Brown corpus and the 3/4-Million-Word corpus of the *Wall Street Journal*.

The *Winnow* approach of Golding and Roth (1996) uses a multiplicative weight update algorithm that achieves a good accuracy and handles a large number of features. The method learns large set of features with the corresponding weight. The method performs better than Bayesian. The multiplicative weight update algorithm represents the members of a confusion set as clouds of simple nodes corresponding to context words and collocation features. Winnow requires confusion sets to be known in advance. In the training phase, a feature extractor learns a set of features and produces a huge list of all features in the training text. Statistics of occurrence of features are also collected. Pruning is applied to eliminate unreliable features. The algorithm has been applied to 21 confusion sets taken from the list of "Words commonly confused" in the back of the Random House dictionary (Flexner (1983)).

3. DETAILS OF Ltest

For a given domain of texts, Ltest carries out two steps called the *learning step* and the *testing step*. In the learning step, Ltest learns from *prior text* that is known to be error-free how context-based spelling errors may manifest themselves. Ltest splits the prior text into a *training text* and a *history text*. We cover the splitting process in a moment.

The idea of training text and history text is based on the following intuitive idea. Suppose we are not experts in some field, say in law. We are given some correct legal document to read. As we scan the text, we may not really understand the sense in which some words are used. But we can learn how words are used in connection with other words. Next, we are given another legal document and are asked to check it for errors. Strictly speaking, we cannot do so since we are not experts. But we can read the second text and see whether some words are used out of context, relative to the word usage in the first text.

In terms of this intuitive discussion, let us view the history text as the first document and the training text as the second one. The reader may object to the latter choice since the training text is correct, as is, of course, the history text. But that changes now. We introduce errors into the training text, one at a time, and try to see how we could locate that error using both the training text and the history text. Using data mining, we compress that knowledge about finding errors into logic formulas. Later, when a new text that is not known to be correct is tested for errors, we analyze that new text using these logic formulas. At that time, the new text plays the role of the training text, while the history text plays the same role as before.

Throughout this section, w is a word that by a typical spelling or typing error may become another word, which we denote by v. We call the correct word w the *intended* word, while any incorrect v that is produced instead of w by a typical spelling or typing error, is an *error word* for w. We collect the error words v for a given intended word w in the *confusion set* for w. For example, if the intended word w is *there*, then the possible error words v for w are *three* and *their*, and hence $\{three, their\}$ is the confusion set for *there*.

We call the possible alteration of w to v a *substitution* and denote it by $v{\leftarrow}w$. The substitutions linking the just-mentioned correct *there* and the error words *their* and *three* are *three*←*there* and *their*←*there*. Other example substitutions are *must*←*just* and *its*←*it's*. To streamline the presentation, we skip here details of the construction of the substitutions. Those details are covered in Appendix A.

We employ the notation v_i to represent the ith instance of the word v in the given text. In connection with a given substitution $v{\leftarrow}w$, we use the adjectives *good* and *bad* in the obvious way. For example, if an instance v_i of v in a text was intended to be an instance w_j of w, then we say that the instance v_i is bad and that the instance w_j is good.

Next we discuss the learning step.

3.1 Learning Step

First, Ltest splits the prior text into a *training text* and a *history text* by assigning each sentence of the prior text to one of the two texts. The assignment is done by a heuristic method described in Appendix B. The method has the goal that, for each substitution $v{\leftarrow}w$ for which both v and w occur in the prior text, the training text and the history text contain about the same number of instances of v as well as w. Of course, that goal may not be reached for a particular v and w, due to the way these words may occur in the sentences of the prior text. But according to experiments, the method typically gets close to that goal.

With the training text and history text at hand, the learning step carries out the following process for each substitution $v{\leftarrow}w$. For each instance v_i of v in the training text, a *characteristic vector* is computed. The vector has a total of 18 ± 1 entries. The entries relate the words, parts-of-speech of words, punctuation marks, and special symbols near a given instance v_i in the training text to the words, parts-of-speech of words, punctuation marks, and special symbols near other instances v_j of v in either the training text or the history text. In terms of the earlier, intuitive discussion, the entries of the characteristic vector record the usage of the word v in the context of the training text and the history text.

For example, suppose that the instance v_i of v is preceded by two words p^1 and p^2, say in the sequence $p^2\ p^1\ v_i$. If some other instance v_j of v in the training text is preceded by the same two words, in the same sequence, that is, $p^2\ p^1\ v_j$, then the 4th entry of the characteristic vector is $+1$. If no such sequence $p^2\ p^1\ v_j$ exists in the training text, then the 4th entry is -1. Analogously, if the history text contains a sequence $p^2\ p^1\ v_j$, then the 13th entry of the characteristic vector is $+1$. If no such sequence exists in the history text, then that entry is -1. To unclutter the presentation, we omit here a detailed discussion of the remaining entries of the characteristic vector. Details are included in Appendix C.

The reader may wonder why we do not use 0 instead of -1 to record absence of the sequence $p^2\ p^1\ v_j$. The reason is the encoding convention of the data mining tool Lsquare introduced shortly. That tool interprets $+1$ to mean that a certain fact, say X, holds, -1 to mean that fact X does not hold, and 0 to mean that it is unknown whether fact X holds, Felici and Truemper (2002).

Suppose the characteristic vectors for each instance v_i of v have been computed. Then, for each instance w_j of w in the training text, w_j is replaced temporarily by an instance v_r of v, and a characteristic vector for that v_r is computed. Consistent with the earlier use of the terms *good* and *bad*, we are justified to call each v_i good and each v_r bad.

At this point, we have two classes of characteristic vectors. The first class consists of the vectors representing features of the good occurrences of v. Let us call this class $G(v)$. The second class consists of the vectors representing features of the bad occurrences of v generated from the occurrences of w in the text. Let us call the second class $B_{v\leftarrow w}(v)$. The subscript $v\leftarrow w$ in the notation $B_{v\leftarrow w}(v)$ is needed since the second class is the set of vectors of bad occurrences of v generated from occurrences of w according to the substitution $v\leftarrow w$.

With the two classes $G(v)$ and $B_{v\leftarrow w}(v)$ at hand, the learning step uses the data mining algorithm *Lsquare* to compute a set of logic formulas $L_{v\leftarrow w}(v)$ that correctly classify each characteristic vector as being in one of the two classes $G(v)$ or $B_{v\leftarrow w}(v)$. Details of Lsquare are given in the chapter "Learning Logic Formulas and Related Error Distributions" included in this volume. Thus, we only sketch here the features of Lsquare needed for the situation at hand.

Lsquare accepts as input two sets A and B of $\{0, \pm 1\}$ vectors, all having the same length, say n. An entry $+1$ means that a certain fact, say X, is known to hold, -1 means that fact X is known not to hold, and 0 means that it is unknown whether fact X holds. For the cases considered in this chapter, the vectors are the above defined characteristic vectors, and thus do not contain any 0s and are $\{\pm 1\}$ vectors. Lsquare outputs a set of 20 disjunctive normal form (DNF) logic formulas and 20 conjunctive normal form (CNF) logic formulas, each of which uses some subset of logic variables y_1, $y_2, \ldots,$ y_n. To classify an arbitrary $\{\pm 1\}$ vector x of length n, Lsquare first assigns *True/False* values to y_1, $y_2, \ldots, y_n$ according to the rule $y_i = True$ if $x_i = 1$ and $y_i = False$ if $x_i = -1$. The *True/False* values are used to evaluate each of the 20 DNF and 20 CNF formulas. If a formula evaluates to *True* (resp. *False*), then we say that the formula produces a vote of 1 (resp. -1). Summing up the 40 votes produced by the 40 logic formulas, we get a *vote-total* that is even and may range from -40 to 40. Lsquare guarantees that, for each vector x of A (resp. B), the vote-total is positive (resp. negative). When A and B are randomly drawn from two populations $\mathcal{A}$ and $\mathcal{B}$, then a vote-total for a record of $\mathcal{A} \cup \mathcal{B}$ close to 40 means that the vector is in $\mathcal{A}$ with high probability and thus is in $\mathcal{B}$ with very low probability. As the vote-total decreases from $+40$ and eventually reaches -40, the probability of membership in $\mathcal{A}$ decreases while that of membership in $\mathcal{B}$ increases.

We interrupt the discussion of the training step for a moment and sketch how the constructed logic formulas $L_{v \leftarrow w}(v)$ are used in the testing step. Suppose we have a testing text with instances of v and w. We want to know whether, relative to the substitution $v \leftarrow w$, an instance v_k of v is good or bad. We compute, for that instance, a characteristic vector $t(v_k)$ using the testing/history texts instead of the training/history texts, and apply to that vector the set of logic formulas of $L_{v \leftarrow w}(v)$. Suppose the vote-total exceeds an appropriately selected threshold. We then estimate the vector $t(v_k)$ to be in the class $G(v)$, which plays the role of $\mathcal{A}$ in the above discussion about Lsquare. Thus, we have evidence that the instance v_k may be good. On the other hand, if the vector $t(v_k)$ is declared to be in the class $B_{v \leftarrow w}(v)$, then this is evidence that the instance v_k may be bad.

We continue the discussion of the training step. So far, we have learned to differentiate between good and bad instances of v relative to the substitution $v \leftarrow w$. Next, the learning step trains how to classify the other word of the substitution, w, as good or bad. Analogously to the case of v, the training step constructs two classes of vectors for w. The first class, $G(w)$, contains one vector for each good occurrence of w in the training text. The second class, $B_{v \leftarrow w}(w)$, includes one vector for each bad occurrence of w generated from one occurrence of v in the training text. Once more, we use Lsquare to determine a set of 40 logic formulas $L_{v \leftarrow w}(w)$ that, using vote-totals, correctly assign the vectors to their sets $G(w)$ and $B_{v \leftarrow w}(w)$. One may employ $L_{v \leftarrow w}(w)$ for testing a text that contains both v and w, as follows. Take an instance v_k of v in the text. To see whether v_k was intended to be a w, temporarily replace v_k by an instance of w; let that instance be w_q. Compute a characteristic vector $f_{v \leftarrow w}(w_q)$ for w_q, and apply $L_{v \leftarrow w}(w)$ to the vector $f_{v \leftarrow w}(w_q)$. If $f_{v \leftarrow w}(w_q)$ is declared to be in $G(w)$ (resp. $B_{v \leftarrow w}(w)$) according to some appropriately selected threshold, then we have evidence that w_q likely is good (resp. bad) and thus v_k likely is bad (resp. good).

Here is an example, for the substitution $there \leftarrow three$. We assume that the training text contains instances of *there* and instances of *three*. The learning step builds four classes of characteristic vectors: $G(there)$, $B_{there \leftarrow three}(there)$, $G(three)$, and $B_{there \leftarrow three}(three)$. Using the first two classes, Lsquare creates the set of logic formulas $L_{there \leftarrow three}(there)$. This set is used to classify new vectors of *there* into the set $G(there)$ or $B_{there \leftarrow three}(there)$. Using the next two classes, namely $G(three)$ and $B_{there \leftarrow three}(three)$, Lsquare builds the set of logic formulas $L_{there \leftarrow three}(three)$. The latter set is used to classify vectors of *three* into the set $G(three)$ or $B_{there \leftarrow three}(three)$. Note that the class $G(there)$ consists of the vectors of the good occurrences of *there*, while the class $B_{there \leftarrow three}(there)$ consists of the vectors of the bad occurrences of *there* generated from the occurrences of *three*.

The above discussion several times explicitly or implicitly refers to appropriately selected thresholds for various vote totals. The computation of these thresholds is part of the testing step, which we cover next.

3.2 Testing Step

We assume that the testing text has been processed by a spell checker and that, therefore, it does not contain any illegal words. For each word v of the text, we find all possible words w that by misspelling or mistyping may become v. That is, we construct the confusion set for v. We process each substitution $v \leftarrow w$ so determined as follows. If the text does not contain w, we cannot test the instances v_k with respect to the substitution $v \leftarrow w$. So assume that at least one instance of w is present. The processing depends on how often v occurs in the testing text. Declare the case to be *regular* if v occurs at least twice in the testing text, and define it to be *special* otherwise. We first treat the regular case.

3.2.1 Testing regular cases

If we do not have both sets $L_{v \leftarrow w}(v)$ and $L_{v \leftarrow w}(w)$ of logic formulas, then the learning step did not provide sufficient insight into the relationship between v and w. Accordingly, we ignore each instance v_k of v in the testing text with respect to the substitution $v \leftarrow w$. We call each such ignored v_k instance relative to $v \leftarrow w$ a *discarded v($v \leftarrow w$) instance*.

Now suppose that both $L_{v \leftarrow w}(v)$ and $L_{v \leftarrow w}(w)$ are available. For each instance v_k of v in the testing text, we construct a characteristic vector $t(v_k)$ from the testing/history texts. For each instance w_l of w in the testing text, we replace w_l temporarily by v_p and construct a characteristic vector $f_{v \leftarrow w}(v_p)$. We handle each instance w_l of w in the testing text analogously to v_k. Thus, for each w_l, we construct a characteristic vector $t(w_l)$. For each v_k of the testing text, we replace v_k temporarily by w_q and construct a characteristic vector $f_{v \leftarrow w}(w_q)$. At this point, we have the characteristic vectors $t(v_k)$, $f_{v \leftarrow w}(v_p)$, $t(w_l)$, and $f_{v \leftarrow w}(w_q)$.

Let us assume that among the instances v_k in the testing text there is at most one in error. We make the corresponding assumption for w. Given these assumptions, we expect that, for all vectors $t(v_k)$ except at most one, the vote-total $r(t(v_k), L_{v \leftarrow w}(v))$ produced by $L_{v \leftarrow w}(v)$ is positive. Correspondingly, we expect that, for all vectors $f_{v \leftarrow w}(v_p)$ except at most one, the vote-total $s(f_{v \leftarrow w}(v_p), L_{v \leftarrow w}(v))$ computed via $L_{v \leftarrow w}(v)$ to be negative. Thus, we expect that there is a threshold value $\alpha_{v \leftarrow w}(v)$ such that almost all, if not all, vote-totals for the vectors $t(v_k)$ are greater than $\alpha_{v \leftarrow w}(v)$, and such that almost all, if not all, vote-totals for vectors $f_{v \leftarrow w}(v_p)$ are less than $\alpha_{v \leftarrow w}(v)$.

We calculate an odd-valued threshold $\alpha_{v \leftarrow w}(v)$ using the above considerations; the details of the computations are given in Appendix D. Given $\alpha_{v \leftarrow w}(v)$, we estimate an instance v_k of the testing text to be bad if its vote-total is less than the threshold, *i.e.*,
$r(t(v_k), L_{v \leftarrow w}(v)) < \alpha_{v \leftarrow w}(v)$ and estimate v_k to be good otherwise. In the former case, the difference $d_1(v_k)$ between the vote-total of $t(v_k)$ and the threshold is a reasonable measure of the likelihood that v_k is bad. That is, a large difference corresponds to a high likelihood.

We utilize $L_{v \leftarrow w}(w)$ in analogous fashion. Each instance v_k is temporarily replaced by an instance w_q of the word w, and we get the vote-total $s(f_{v \leftarrow w}(w_q), L_{v \leftarrow w}(w))$ for the characteristic vector $f_{v \leftarrow w}(w_q)$ of the generated w_q. The vote-total is computed by $L_{v \leftarrow w}(w)$. If the vote-total is above the threshold $\alpha_{v \leftarrow w}(w)$, then the generated occurrence w_q likely is good, the instance v_k is estimated to be bad, and the difference $d_2(v_k)$ between the vote-total and the threshold is a measure of the likelihood that v_k is bad. If the vote-total is less than the threshold, then the generated occurrence w_q likely is bad, and thus the instance v_k is estimated to be good. Notice that the threshold $\alpha_{v \leftarrow w}(w)$ is computed analogously to $\alpha_{v \leftarrow w}(v)$ described above.

The tests involving the two thresholds may produce agreeing or conflicting estimates for a given instance v_k. If at least one of the two tests estimates v_k to be good, then we estimate v_k to be good. On the other hand, if both tests estimate v_k to be bad, then we estimate v_k to be bad and take the sum $d_s(v_k)$ of $d_1(v_k)$ and $d_2(v_k)$ to be a measure of the likelihood that the estimate of v_k being bad is indeed correct. Accordingly, we sort all such bad instances v_k using their $d_s(v_k)$ values. The v_k with the largest $d_s(v_k)$ is the most likely one to be bad. In the implementation of the method, that instance v_k is posed to the user as a questionable word. If the user declares v_k to be correct, we assume that the other instances of v that we estimated to be bad, are actually good as well. On the other hand, if the user agrees that the v_k with largest $d_s(v_k)$ is indeed bad, then we pose to the user the case of the v_k with the second largest $d_s(v_k)$ as potentially bad and apply the above rule recursively.

3.2.2 Testing special cases

We have completed the discussion of the regular case where each of v and w occurs at least twice in the testing text. Now, we discuss two special cases where v occurs exactly once in the testing text. Since v occurs just once, it may well be that this instance of v is bad. Hence, this situation calls for careful analysis. Here are the two cases.

Case (1) The word v occurs exactly once but w occurs at least twice in the testing text: We construct for each instance w_l the characteristic vector $t(w_l)$, apply $L_{v \leftarrow w}(w)$, and get the vote-total $r(t(w_l), L_{v \leftarrow w}(w))$. Next, we temporarily replace the single v_k by w_q, construct the characteristic vector $f_{v\leftarrow w}(w_q)$, and apply $L_{v\leftarrow w}(w)$. If the resulting vote-total $s(f_{v\leftarrow w}(w_q), L_{v\leftarrow w}(w))$ for the generated w_q is greater than the smallest of the $r(t(w_l), L_{v\leftarrow w}(w))$, then we estimate the w_q that replaced v_k to be good and thus estimate v_k to be bad; otherwise, we estimate v_k to be good.

Case (2) The word v occurs exactly once and w occurs only once in the testing text: We would like to construct a characteristic vector $t(v_k)$ for v_k as in the regular case, apply $L_{v\leftarrow w}(v)$, and make a decision based on the vote-total. However, the rules for construction of $t(v_k)$ demand that v_k occurs at least twice in the testing text, which does not hold here. Hence, $t(v_k)$ cannot be computed. We overcome this difficulty by a seemingly inappropriate step where the testing text is for the moment replaced by the history text appended by v_k and its neighborhood of the testing text. That temporary substitution allows computation of $t(v_k)$, since existence of $L_{v\leftarrow w}(v)$ implies that v occurs at least three times in the history text. We apply $L_{v\leftarrow w}(v)$ to the vector $t(v_k)$, get a vote-total $r(t(v_k), L_{v\leftarrow w}(v))$, and estimate v_k to be good or bad using a threshold of $\alpha_{v\leftarrow w}(v) = -19$. That is, if $r(t(v_k), L_{v\leftarrow w}(v)) < -19$, then v_k is estimated to be bad. Otherwise, it is estimated to be good. The threshold choice is driven by the consideration that vote-totals below -19 almost always show v_k to be bad.

3.2.3 An example

Let us discuss an example where the word *there* is examined in a given testing text. First, Ltest constructs the set of confusion words for *there*. Let that set be {*three*, *their*}. Thus, we have two substitutions involving *there*: *there*←*three* and *there*←*their*. Each occurrence of *there* is examined twice. Once, *there* is examined relative to the substitution *there*←*three*. The second time, *there* is examined relative to the substitution *there*←*their*. Let us discuss the first case. When *there* is examined relative to the substitution *there*←*three*, each occurrence $there_k$ of *there* in the testing text is tested twice as follows:
(i) Compute the vector for $there_k$. Based on the testing technique described above, the occurrence $there_k$ is estimated to be good or bad.
(ii) Replace $there_k$ by an occurrence $three_q$ of the word *three*, and construct a vector for that generated occurrence $three_q$. Then $three_q$ can be classified as good (resp. bad), and thus the occurrence $there_k$ is estimated as bad (resp. good). If the two tests (i) and (ii) estimate $there_k$ as bad, then the occurrence $there_k$ is declared bad; otherwise $there_k$ is declared good.

Declare each instance of v that in the testing step is ignored relative to a substitution $v\leftarrow w$ to be a *discarded* $v(v\leftarrow w)$ *instance*. If an instance of v is not discarded relative to a substitution $v\leftarrow w$, declare it to be a *tested* $v(v\leftarrow w)$ *instance*. By these definitions, an instance of v may be discarded relative to a substitution $v\leftarrow w$ and may be tested relative to another substitution $v\leftarrow z$.

Let N_d (resp. N_t) be the total number of discarded (resp. tested) $v(v\leftarrow w)$ instances encountered in all iterations through the testing step. If the ratio

$N_t/(N_d + N_t)$ is close to 1, then the learning step has produced most of the logic formulas needed for checking the given testing text. On the other hand, a ratio close to 0 implies that the learning step has produced few of the logic formulas that are relevant for the testing text. For this reason, we call the ratio $N_t/(N_d + N_t)$ the *relevance ratio* of the given prior text and the given testing text. Section 5 shows that relevance ratios close to 1 can be achieved by a suitable augmentation of the given training and history texts.

4. IMPLEMENTATION AND COMPUTATIONAL RESULTS

The learning step and the testing step of Ltest have been added to an existing software system for spell and syntax checking called *Laempel*. In this section, we review that system, describe how the method has been inserted, and report computational results that include a comparison with the prior methods BaySpell and WinSpell.

The spell checker of Laempel is described in Zhao and Truemper (1999). The key feature setting it apart from other spell checkers is the high probability with which Laempel suggests correct replacement words for misspelled words (96% for the top-ranked replacement word) and recognizes correct words that are not in the dictionary, as correct (82%). Laempel achieves this performance by learning user behavior and using that insight to make decisions.

The syntax checker of Laempel is covered in Zhao (1996). It consists of three steps. In the first step, the given text is cleaned up by a screening process. In the second step, two logic modules check the cleaned text for local syntactic errors. A total of 27 different cases are considered. The third step is applied to each sentence that does not contain any local syntactic errors. A reasoning process involving 18 logic modules analyzes each such sentence for global syntactic errors. If no such error is determined, the process attempts to parse the sentence. We say "attempts" since the process gives up on parsing if the sentence is so complex that the 18 logic modules become bogged down in the parsing process. In tests, the percentage of sentences that were parsed by the syntax checker ranged from 100% for simple texts and 76% for a mathematical text to 61% for a TV network news text. For the sentences that have been parsed, Laempel records for each word the assigned part-of-speech. That information is utilized later to estimate whether a given word has a dominant part-of-speech.

We are ready to discuss the implementation of the learning step of Ltest. Recall that the learning algorithm splits the prior text into a training text and a history text, and then deduces from these two texts a collection of logic formulas. Prior to the computation of the formulas, Laempel carries out spell checking and syntax checking for the two texts and asks the user to make corrections as needed. The learning algorithm processes the corrected texts to obtain the collection of logic formulas.

We turn to the implementation of the testing step of Ltest. Let a testing text be given. Laempel first checks the text for spelling and syntax errors. Once the user has made appropriate corrections, the testing algorithm searches the text for context-based spelling errors. Whenever the algorithm has produced a list of likely errors for a substitution $v \leftarrow w$, Laempel poses the top-ranked instance of the list to the user as possibly in error. If the user declares the instance to be correct, Laempel assumes that all other instances of the list are correct as well. On the other hand, if the user declares the instance to be in error, Laempel records that fact, removes the instance from the list, and applies the above rule recursively; that is, Laempel poses the currently top-ranked instance to the user as possibly being in error, and so on. Once a testing text has been checked for context-based spelling errors, Laempel records all sentences that do not contain any error acknowledged by the user. When the text is processed again after changes by the user, those sentences are presumed to be correct, and checking focuses on modified or new sentences. This rule reduces subsequent processing times of the testing file.

We have evaluated the performance of Ltest. The texts consisted of mathematical book chapters formulated in TEX, technical papers in LATEX, and newspaper texts covering health and politics. Table 1 tells the number of words, the number of word instances of the texts, and the classification as training, history, or testing text. Note that the first group of texts consisting of texts 1–1 to 1–4 contains two history texts 1–2 and 1–3. The smaller of the two history texts, 1–2, has 8,291 word instances, while the larger history text, 1–3, has 16,443 instances. We see in a moment how the difference in size of the two history texts affects the learning of logic formulas.

Recall from the learning step that for a given substitution $v \leftarrow w$ we attempt to derive two sets $L_{v \leftarrow w}(v)$ and $L_{v \leftarrow w}(w)$ of logic formulas, by first replacing each instance of w by v, and then replacing each instance of v by w. Denote the first replacement by v–w and the second one by w–v. We need this notation for the next table, which summarizes the results of applying the learning algorithm to the combinations of training/history texts shown in Table 2. The statistics include the number of replacements v–w evaluated, the distribution of the number of instances of v in the training text connected with the replacements v–w, the total number of logic formulas learned from the texts, and the execution time. Computations were done on a Sun Ultra 1 (167 MHz) workstation, which by current standards is slow.

The training time ranges from 1h 17m to 6h 14m, with an average of 3h 43m. Most of that time is required for the computation of logic formulas by Lsquare.

On present day computers, say with 1000 MHz, training time would be at most 1h.

Line 1 of Table 2 shows that training text 1–1 and history text 1–2 led to learning of 4,760 logic formulas for 119 replacements. In contrast, the

Table 1: Text statistics

Text	Used for	Type	Number of different words	Number of word instances
1–1	training	math book	1,922	2,4581
1–2	history	chapters	1,143	8,291
1–3	history	in	1,827	1,6443
1–4	testing	TEX	1,100	9,744
2–1	training	math book	1,826	1,6444
2–2	history	chapters in	1,715	1,5098
2–3	testing	TEX	1,216	6,491
3–1	training	technical	2,029	1,8773
3–2	history	papers in	2,817	2,5881
3–3	testing	LATEX	1,318	6,456
4–1	training	newspaper	1,968	6,645
4–2	history	articles about	1,925	5,854
4–3	testing	health	1,334	4,667
5–1	training	newspaper	2,054	8,837
5–2	history	articles about	2,086	8,645
5–3	testing	politics	1,590	5,726

Table 2: Learning cases

Texts		Number of re-placements v–w	Distribution of number of instances of v in training text for replacements v–w (%)						Number of logic formulas learned	Training time
Train-ing	Hist-ory		3 to 10	11 to 20	21 to 50	51 to 100	101 to 200	>200		
1–1	1–2	119	11	13	17	27	18	14	4,760	4h 3m
1–1	1–3	176	27	16	15	18	13	11	7,040	5h 58m
2–1	2–2	178	24	16	21	19	12	8	7,120	3h 4m
3–1	3–2	250	31	19	15	18	6	10	10,000	6h 14m
4–1	4–2	48	40	15	33	2	6	4	1,920	1h 17m
5–1	5–2	64	30	23	25	8	11	3	2,560	1h 40m
Avrg.		139	27	17	21	15	11	8	5,567	3h 43m

same training text paired with history text 1–3 results in learning 7,040 logic formulas for 176 replacements, an increase of 48%. The large increase in learned information is due to the larger size of history text 1–3 compared with history text 1–2. Note that on average 27% of the training samples contained 3 to 10 instances of v in the training text, while 17% of the training samples contained 11 to 20 instances. Thus, roughly half of the training samples had

Table 3: Large testing text cases

Texts			Number of discarded $v(v \leftarrow w)$ instances	Number of tested $v(v \leftarrow w)$ instances	Number of false-positive diagnoses	Testing time
Testing	Training	History				
1–4	1–1	1–2	5,324	4,610 (46%)	18(0.4%)	2m 0s
1–4	1–1	1–3	4,535	5,399 (54%)	24(0.4%)	2m 35s
2–3	2–1	2–2	3,777	3,061 (45%)	28(0.9%)	2m 2s
3–3	3–1	3–2	2,911	3,258 (53%)	33(1.0%)	2m 32s
4–3	4–1	4–2	3,404	891 (21%)	20 (2.2%)	1m 6s
5–3	5–1	5–2	5,239	1,756 (25%)	17 (1.0%)	1m 25s
Average			4,198	3,162 (43%)	23.4(0.7%)	1m 57s

at most 20 instances.

Once training was completed, the testing algorithm was applied to both large and small testing texts in the domains of the training/history texts. The cases of large testing texts are given in Table 3. For each testing text, the table includes the related training/history texts, the number of discarded and tested $v(v \leftarrow w)$ instances, and the number of false-positive diagnoses incurred when algorithm processes the text. The percentage given with the number of tested $v(v \leftarrow w)$ instances is the relevance ratio, which in Section 3 is defined to be the number of tested $v(v \leftarrow w)$ instances divided by the total number of discarded and tested $v(v \leftarrow w)$ instances. For testing text 1–4, the relevance ratio is 46% when 1–1/1–2 are used as training/history texts. The ratio increases to 54% when 1–1/1–3 are used instead. The improvement is due to the fact that history text 1–3 leads to increased learning when compared with history text 1–2, as discussed in connection with Table 2. The average relevance ratio, which is 43%, is an undesirably small number that results from the random selection of training and history texts. In Section 5 it is described how relevance ratios close to 1 can be achieved by an appropriate augmentation of the training texts and history texts. We did not carry out such manipulation for the tests of this section so that the test results are unbiased.

The percentage listed in Table 3 with the number of false-positive diagnoses is the ratio of that number divided by the number of tested $v(v \leftarrow w)$ instances. That percentage is small and ranges from 0.4% to 2.2%, with an average of 0.7%. Much more important from a user standpoint is the fact that the number of false-positive diagnoses is uniformly small, ranging from 17 to 33, with an average of about 23. The testing time is on the order of 2m for each case. On current computers, that time would be on the order of 20s.

Into each large testing text of Table 3, we randomly introduced context-based spelling errors and, for each such error, checked if the syntax checker or Ltest detected that error and posed it to the user as top-ranked candidate. Thus, the results characterize the error detection capability of Ltest

Table 4: Error detection for large testing texts

Texts			Number of errors generated	Number of errors detected by		
Testing	Training	History		syntax checker alone	Ltest alone	syntax checker and Ltest
1–4	1–1	1–2	37	3 (8%)	26 (70%)	26 (70%)
1–4	1–1	1–3	163	16 (10%)	114 (70%)	120 (74%)
2–3	2–1	2–2	47	7 (15%)	34 (72%)	35 (74%)
3–3	3–1	3–2	53	9 (17%)	38 (72%)	39 (74%)
4–3	4–1	4–2	47	5 (11%)	27 (57%)	32 (68%)
5–3	5–1	5–2	33	5 (15%)	20 (61%)	22 (67%)
Average			63.3	7.5 (12%)	43.2 (68%)	45.7 (72%)

Table 5: Small testing texts

Text	Number of different words	Number of word instances
1–4′	81	184
1–4″	58	110
2–3′	36	50
3–3′	51	80
4–3′	78	106
5–3′	85	102

for cases where learning is possible from the training/history texts. Table 4 summarizes the performance. The percentage figures in parentheses represent the portion of generated errors detected by the syntax checker or Ltest, as applicable. Note that the syntax checker on average found only 12% of the errors, while Ltest identified 68%. Combined, the two checks located 72% of the errors.

We extracted several small testing texts consisting of at most a few sentences from the large testing texts. Table 5 contains the statistics about these small testing texts. The names of the texts are derived from those of the large ones by adding one or two primes. For example, the small testing texts 1–4′ and 1–4″ are derived from the large testing text 1–4.

Table 6 lists the training/history texts used in conjunction with the small testing texts and provides statistics analogously to Table 3. The average relevance ratio is 43% and thus equal to that for large testing texts. The number of false-positive diagnoses ranges from 0 to 2, with an average of 1. The average false-positive rate is 8%. That percentage may seem high, but this is not important since the number of false-positive diagnoses is small. The execution times of Table 6 are far below the roughly 2m required for large testing texts and average 6s. On current (2001) computers, the average time would be about 1s.

Table 6: Small testing text cases

Texts			Number of discarded $v(v{\leftarrow}w)$ instances	Number of tested $v(v{\leftarrow}w)$ instances	Number of false-positive diagnoses	Testing time
Testing	Training	History				
1–4′	1–1	1–2	43	32 (43%)	0 (0%)	12s
1–4″	1–1	1–3	11	20 (65%)	1 (5%)	7s
2–3′	2–1	2–2	3	3 (50%)	2 (67%)	4s
3–3′	3–1	3–2	7	4 (36%)	1 (25%)	5s
4–3′	4–1	4–2	23	7 (23%)	2 (29%)	3s
5–3′	5–1	5–2	7	7 (50%)	0 (0%)	5s
Average			16	12 (43%)	1 (8%)	6s

Table 7: Error detection for small testing texts

Texts			Number of errors generated	Number of errors detected by		
Testing	Training	History		syntax checker alone	Ltest alone	syntax checker and Ltest
1–4′	1–1	1–2	8	0 (0%)	6 (75%)	6 (75%)
1–4″	1–1	1–3	10	2 (20%)	10 (100%)	10 (100%)
2–3′	2–1	2–2	8	0 (0%)	6 (75%)	6 (75%)
3–3′	3–1	3–2	6	0 (0%)	5 (83%)	5 (83%)
4–3′	4–1	4–2	6	0 (0%)	6 (100%)	6 (100%)
5–3′	5–1	5–2	2	0 (0%)	2 (100%)	2 (100%)
Average			6.7	0.3 (4%)	5.8 (87%)	5.8 (87%)

As for the cases of large testing texts, we randomly inserted context-based spelling errors and determined how many of these errors were identified by the syntax checker or by Ltest. Table 7 contains the results. On average, the syntax checker finds only 4% of the errors, while Ltest locates 87%. In contrast to the large testing texts, the syntax checker does not help at all since Ltest finds all errors determined by the syntax checker.

Table 8 summarizes the performance of the leading prior methods BaySpell (Golding (1995)) and WinSpell (Golding and Roth (1999)) and Ltest on the same prior text and testing text. D. Roth kindly made these texts available. They were obtained by a 80/20 split of the 1-Million-Words Brown corpus (Kučera and Francis (1967)). The figures in the table represent the percentages of correctly classified word instances for the specified confusion sets. On average, Ltest achieved the best performance with 95.4% accuracy, compared with 89.9% for BaySpell and 93.5% for WinSpell. The testing times used by Ltest for the cases in Table 8 are comparable and very close to those reported in Table 3.

The detection rate of 95.4% for Ltest is much higher than the 68% found earlier for large texts. How is this possible? First, the two rates concern

Table 8: Performance of Ltest compared with BaySpell and WinSpell

Confusion set	BaySpell	WinSpell	Ltest
accept, except	92.0	96.0	94.0
affect, effect	98.0	100	97.9
being, begin	95.2	97.9	98.7
cite, sight	73.5	85.3	81.3
country, county	91.9	95.2	94.1
its, it's	95.9	97.3	98.3
lead, led	85.7	91.8	98.0
passed, past	90.5	95.9	94.7
peace, piece	92.0	88.0	92.6
principal, principle	85.3	91.2	94.7
quite, quiet	89.4	93.9	98.5
raise, rise	87.2	89.7	98.2
weather, whether	98.4	100	98.7
your, you're	90.9	97.3	95.9
average	89.9	93.5	95.4

different statistics. The 95.4% rate covers classification of correct words as correct and of erroneous words as incorrect. The 68% rate covers only the detection of erroneous words as incorrect. If we are to compare numbers, we must combine the 68% rate with the rate for classifying correct words as correct. The latter rate is $1-(\text{false-positive rate}) = 1-0.007 = 99.3\%$. Using a 50/50 weighting to combine rates, we see that $(0.68 + 0.993)/2 = 83.7\%$ should be compared with 95.4%. From our computational experience with Ltest, the gap between 83.7% and 95.4% is due to four factors:

– First, the test using the Brown corpus relies on much larger training texts than we used in the earlier tests.

– Second, most confusion words of Table 8 are *content* words—that is, nouns, verbs, adjectives, and adverbs. We have found confusion sets involving such words to be much easier to handle than sets involving *function* words such as prepositions, connectives, and articles. Such words were part of the earlier tests. Indeed, the tests even check for some errors in mathematical formulas such as misspelled mathematical variables.

– Third, some of the large testing texts considered earlier involve numerous special terms, symbols, formulas, and conventions, which complicate the search for errors.

– Fourth, the constraint of very low false-positive rate imposed on Ltest makes detection of errors much more difficult. It would be interesting to see how the two leading prior methods perform when they are adapted so that they take all of these aspects into account.

The experiments reported show that Ltest finds the majority of context-based spelling errors, provided the error instances v_k can be tested. This is so if two conditions are satisfied: (1) For each erroneous instance v_k, the correct word must occur in the testing text. (2) Logic formulas for the applicable substitutions $v \leftarrow w$ must have been learned.

The first condition is typically met in large testing texts, but is not necessarily satisfied in small testing texts. There is a simple way to avoid this shortcoming for small testing texts. We take an additional, large text in the same domain area, test it, and correct it if necessary. Let us call the resulting text the *core text.* Whenever a small text is to be tested, we adjoin it to the core text and test the resulting large *expanded text.* If an instance of v occurs in the small text portion, and if an instance of w occurs anywhere in the expanded text, then v is tested for possibly being the result of a context-based spelling error. As a result, almost any v that should be tested is indeed tested.

The second condition is satisfied if the training/history texts are representative of the testing texts. This is not the case for the above tests due to our random selection of training/history texts. In the next section, we see how representative texts can be obtained, as part of several extensions.

5. EXTENSIONS

Significant improvements in the error detection rate can be attained by a better syntax checker, since, in our tests, quite a few context-based spelling errors resulted in syntactically incorrect sentences that were not flagged by the Laempel syntax checker. A better syntax checker would also lower the false-positive rate, for the following reason. Let $v \leftarrow w$ be the currently processed substitution in the testing step. Suppose the testing step tentatively replaces an instance of v by w. If the syntax checker determines that the modified sentence is syntactically incorrect, then we need not consider v as a possibly misspelled or mistyped w, and thus eliminate a potential false-positive diagnosis. We tried this idea using the Laempel syntax checker and found that it reduced the number of false-positive diagnoses insignificantly. A better syntax checker should produce substantially better results.

Another improvement produces a relevance ratios very close to 1. Suppose we have sufficient text to determine the entire vocabulary used in the given domain. We compute the confusion set for each word of that vocabulary. Given a training text and a history text, we check if each of these texts contains, for each word occurring in one of the confusion sets, at least three instances each. If this is not the case for a given word, we add sentences from general text material to the training or history text, as needed, until each word is reasonably represented, say, by 10-20 instances. When training is done using the expanded training and history texts, then logic formulas are produced for each word of each confusion set. Accordingly, testing achieves a relevance ratio of close to 1.

It is possible that a person makes an error repeatedly, for example, by confusing "its" and "it's" or "complement" and "compliment." Such behavior is contrary to one of the two assumptions made in Section 1, and it affects the reliability with which errors are detected via thresholds. One may remedy this shortcoming of Ltest as follows. Whenever an error involving a given substitution $v \leftarrow w$ is found to occur more than once in a testing text, then that case is recorded as part of the *performance history* of the person who created the text. In subsequent tests, that fact is taken into account when characteristic vectors are constructed in connection with the substitution $v \leftarrow w$ and evaluated via logic formulas. Space constraints prevent a detailed discussion, but the main idea is that, for the evaluation of $v \leftarrow w$, each sentence with an instance of v is viewed as separate small text, and that v is tested for correctness as described in Section 4 using a core text. The use of a performance history of a person may seem far-fetched. But the spell and syntax checker of the Laempel System, of which Ltest is now part, already uses such history information, with good results.

This section discussed some of the future research directions which can be summarized as follows: –Improving the syntax checker to reduce the number of false-positive cases as mentioned earlier in this section. –Devise methods to derive certain *generic* formulas to be used whenever training is not possible due to small number instances. –Explore the usage of some *core* good text (see the last two paragraphs in section 4) to be adjoined to small testing texts where there may not be enough word instances to collect sufficient features of these words in the characteristics vector.

6. SUMMARY

The chapter describes the method Ltest for finding context-based spelling errors. The key elements are as follows.

(1) An encoding of the relationships of an instance of a word to other instances of that word, using the text under investigation and a history text that acts as a reference text for both training and testing. The encoding is based on neighborhoods of word instances and, if applicable, on the dominant parts-of-speech of such instances.
(2) Representation of the relationships between words instances and correct/incorrect use by logic formulas that are extracted by a data mining algorithm.
(3) A voting system based on the logic formulas.
(4) A calibration of the voting system via thresholds for each testing text.

Ltest has been added to an existing system for checking spelling and syntax errors. A number of tests have proved that the resulting system is effective and robust. It detects the majority of context-based spelling errors while committing few false-positive diagnoses. Execution times of the system are moderate for the learning step and are small for testing even large texts.

REFERENCES

Al-Mubaid, H., Identifying Inadvertent Semantic Errors in English Texts, Ph.D. Thesis, University of Texas at Dallas, 2000.

Bruce, R., and Wiebe, J., Word-sense disambiguation using decomposable models, Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics (ACL-94), 1994, 139–146.

Bruce, R., and Wiebe, J., Decomposable modeling in natural language processing, Computational Linguistics, 25 (1999) 195–207.

Budanitsky, A., Lexical semantics relatedness and its application in natural language processing, Technical Report CSRG-390, University of Toronto, 1999.

Budanitsky, A., and Hirst, G., Semantic distance in WordNet: An experimental, application-oriented evaluation of five measures, Workshop on WordNet and Other Lexical Resources, Second Meeting of the North American Chapter of the Association of Computational Linguistics, Pittsburgh, June, 2001.

Felici, G., Sun, F., and Truemper, K., A method for controlling errors in two-class classification, Proceedings of the 23rd Annual International Computer Software & Applications Conference COMPSAC 99, Phoenix, AZ, 1999, 186–191.

Felici, G., and Truemper, K., A Minsat approach for learning in logic domains, INFORMS Journal on computing, 14(1) 2002, 20–36.

Golding, A. R., A Bayesian hybrid method for context-sensitive spelling correction, Proceedings of the Third Workshop on Very Large Corpora, Cambridge, MA, 1995, 39–53.

Golding, A. R., and Roth, D., Applying Winnow to context-sensitive spelling correction, Machine Learning: Proceedings of the Thirteenth International Conference, San Francisco, CA, 1996, 182–190.

Golding, A. R., and Roth, D., A Winnow-based approach to context-sensitive spelling correction, Machine Learning, Special Issue on Machine Learning and Natural Language Processing, (34) 1999 107–130.

Golding, A. R., and Schabes, Y., Combining Trigram-based and feature-based methods for context-sensitive spelling correction, Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics, Santa Cruz, CA, 1996, 71–78 .

Hirst, G., and St-Onge, D., Lexical chains as representations of context for the detection and correction of malapropisms, Christaine Felbaum, editor, WordNet, MIT Press, MA, 1995.

Jones, M. P., and Martin, J. H., Contextual spelling correction using latent semantic analysis, Proceedings of the 5th Conference on Applied Natural Language Processing, Washington, DC, 1997.

Kučera, H., and Francis, W. N., *Computational Analysis of Present-Day American English*, Brown University Press, Providence, RI, 1967.

Kukich, K., Techniques for automatically correcting words in text, ACM Computing Survey, 24 (1992) 377–439.

Mangu, L., and Brill, E., Automatic rule acquisition for spelling correction, Proceedings of the International Conference on Machine Learning, 1997, 734–741.

Pedersen, T., Search techniques for learning probabilistic models of word sense disambiguation, Working Notes of the AAAI Spring Symposium on Search Techniques for Problem Solving Under Uncertainty and Incomplete Information, Palo Alto, CA, 1999.

Pedersen, T., and Bruce, R., Knowledge lean word-sense disambiguation, Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98), Madison, WI, 1998.

Pedersen, T., Bruce, R., and Wiebe, J., Sequential model selection for word sense disambiguation, Proceedings of the 1997 Conference on Applied Natural Language Processing (ANLP-97). Washington, DC, 1997, 388–395.

Powers, D., Learning and application of differential grammars, Proceedings of the ACL Special Interest Group in Natural Language Learning, Madrid, 1997.

Roth, D., Learning to resolve natural language ambiguities: A unified approach, Proceedings of National Conference on Artificial Intelligence, 1998, 806–813.

St-Onge, D., Detecting and correcting malapropisms with lexical chains, MS Thesis, University of Toronto, Department of Computer Science, 1995.

Webster's Ninth Collegiate Dictionary, Merriam-Webster, Inc., and Highlighted Data, Inc., Macintosh CD-ROM edition, 1989.

Zhao, Y., Intelligent text processing, Ph.D. Thesis, University of Texas at Dallas, 1996.

Zhao, Y., and Truemper, K., Effective spell checking by learning user behavior, Applied Artificial Intelligence, 13 (1999) 725–742.

Appendix A: Construction of substitutions

Recall that the substitution $v \leftarrow w$ represents that the word w may by misspelling or mistyping become v. In this appendix, we summarize how, given v, all words w that may give rise to a substitution $v \leftarrow w$ are computed. For complete details of the construction rules, see Al-Mubaid (2000).

We collect the misspelling cases in four groups that involve the following situations. For each situation, we include some examples.

- vowel combinations producing similar sounds: by↔buy, fair↔fare, week↔weak.
- consonants having similar sounds: bay↔pay, cine↔sine, hid↔hit.
- silent characters and substrings: knee↔nee, right↔rite, sight↔site, where↔were.
- apostrophe use: he's↔his, it's↔its, let's↔lets, they're↔there.

A total of 61 rules create all cases of these four groups. We determined these rules as follows. First, in a combination of manual and computer search, we extracted from Webster's Ninth Collegiate Dictionary (1989) a number of classes of different words with identical or nearly identical pronunciation. Second, we manually eliminated rare words. Third, we represented the remaining classes by rules. It turned out that 61 rules suffice to represent those classes. Due to space constraints, we omit a detailed listing of the rules; they are included in Al-Mubaid (2000). We use the shorthand notation $v \leftrightarrow w$ for the substitutions $v \leftarrow w$ and $w \leftarrow v$.

The mistyping cases are taken from Zhao and Truemper (1999). Define a *neighbor letter* to be any letter that on the keyboard is close to a given letter. Then the typing errors considered in the cited reference are as follows: transposing two letters, repeating a letter, omitting a letter, inserting a letter that is a neighbor of a given letter, and typing an incorrect letter that is a neighbor of the required letter. Here are some examples for each type of error.

1. Transposing two letters: bye↔bey, form↔from, goal↔gaol, trial↔trail.
2. Repeating a letter: latter←later, tiller←tiler.
3. Omitting a letter: cam←scam, met←melt, see←seem, tale←table.
4. Inserting a letter that is a neighbor of a given letter: defined←define, care←car,
   trash←rash.
5. Typing an incorrect letter that is a neighbor of the required letter: for↔foe, high↔nigh, into↔onto, just↔must.

Appendix B: Construction of training and history texts

Suppose we have a database of correct texts for the given domain. For example, the database may consist of a large number of working papers on graph theory, or of papers published in a combinatorics journal, or of some books on matrix algebra, or of a large collection of legal documents in one area of law. We assume that the words used in the database constitute the entire or almost entire vocabulary of the testing texts we intend to process in the same domain area. From the database, we want to derive reasonably sized training and history texts so that the learning step applied to them produces the logic formulas needed in subsequent testing of texts. The construction of the training text and history text from the given database proceeds as follows.

Construction of training/history texts

INPUT: A database of texts for the given domain area.

OUTPUT: Training/history texts for the domain area.

1. Initialize the training text and history text as empty texts. Determine the number $n(v)$ of instances of each word v in the database.
2. Derive all substitutions $v{\leftarrow}w$ for which both v and w occur in the database, and collect the words v and w of these $v{\leftarrow}w$ in a set V. Sort the words of V using the counts $n(\cdot)$ so that the topmost word has the smallest $n(\cdot)$ value.
3. Process the words v of V one by one and in the order determined in Step 2, as follows. Randomly select sentences of the database that contain at least one instance of v, and assign each selected sentence to the training text or the history text, whichever has at that time the fewest number of instances of v. Stop the processing of v when all sentences of the database containing instances of v have been assigned, or when both the training text and the history text contain at least 1,000 instances of v each.
4. Output the training/history texts on hand, and stop.

Appendix C: Structure of characteristic vectors

Both the learning step and the testing step use characteristic vectors to encode the relationships connecting a given word instance with other instances of the same word. The characteristic vectors are based on two texts. The first text is the training text or testing text, depending on whether we are in the learning step or in the testing step, respectively. We denote either one of the two texts by T. The second text is the history text, regardless of whether we are in the learning step or in the testing step. We denote that text by H. It acts as a correct reference text during both the learning step and the testing step.

Define a *non-word token* to be any symbol that occurs in text T or H that is not a word. Examples are the period, comma, exclamation mark, question mark, semicolon, colon, forward and backward slash, plus, minus, ampersand, and the signs for pound and dollar.

Define the *neighborhood* of an instance x_m of a word x in text T or H to consist of the two words or tokens immediately preceding x_m and of the two words or tokens immediately following x_m. Let p^1, p^2, f^1, and f^2 denote words or tokens. Then x_m and its neighborhood in the text T or H may be depicted as a sequence $p^2\ p^1\ x_m\ f^1\ f^2$ of words or tokens in the text. The neighborhood definition is modified in the obvious way if x_m occurs at or near the beginning or end of the text. That is, p^1 and p^2, or just p^2, or f^1 and f^2, or just f^2 are then absent from the sequence.

Define a part-of-speech of a word to be the *dominant* part-of-speech of the word if in past usage of the word in the given domain that part-of-speech was the correct syntactic interpretation at least 90% of the time. In our implementation, we use the output of the Laempel syntax checker to estimate whether a part-of-speech is dominant. Below, we assume that we have that estimate available.

Both the learning step and the testing step require characteristic vectors for instances x_m of words x in text T. Such a vector, denoted by Z^{x_m}, has 18 entries encoding 18 different features of that particular instance. The first half of the entries is produced from text T, while the second half is generated from text H. All but four of the entries of Z^{x_m} relate tokens of the neighborhood of the instance x_m of a given word x to the tokens of the neighborhood of other instances x_n of x. The remaining four entries of Z^{x_m} link the parts-of-speech of words in the neighborhood of x_m to the parts-of-speech of words in the neighborhood of instances x_n. The 18 features are the result of a long series of experiments involving many different rule sets. In those experiments, we started out with elaborate rule sets. We discovered that such sets tend to produce erratic detection results and are unsuitable when both large and small texts are to be processed. By gradual simplification we arrived at the current rule set. Here are the details.

Construction of the characteristic vector

Each of the entries $Z_1^{x_m}$, $Z_2^{x_m}$, $\ldots$ $Z_{18}^{x_m}$ is equal to ± 1. The rules below list explicitly for each entry the condition under which an entry takes on the value 1. If that condition is not satisfied, then the entry implicitly has the value -1. We use the above notation for neighborhoods—that is, p^2 p^1 x_m f^1 f^2—to denote the tokens immediately preceding or following a given instance x_m in a sentence. Here are the definitions.

1. (p^1 or p^2 in text T, word case) Define $Z_1^{x_m} = 1$ if (a) or (b) below hold.
   (a) p^1 is a word and, for another instance x_n of x, the sequence p^1 x_n occurs in text T.
   (b) p^1 is a non-word token and p^2 is a word, and, for another instance x_n of x and for another non-word token q, the sequence p^2 q x_n occurs in text T.
2. (f^1 or f^2 in text T, word case) Define $Z_2^{x_m} = 1$ if (a) or (b) below hold.
   (a) f^1 is a word and, for another instance x_n of x, the sequence x_n f^1 occurs in text T.
   (b) f^1 is a non-word token and f^2 is a word, and, for another instance x_n of x and for another non-word token q, the sequence x_n q f^2 occurs in text T.
3. (p^1 and f^1 in text T, word case) Define $Z_3^{x_m} = 1$ if both p^1 and f^1 are words and if, for another instance x_n of x, the sequence p^1 x_n f^1 occurs in text T.
4. (p^1 and p^2 in text T, word case) Define $Z_4^{x_m} = 1$ if both p^1 and p^2 are words and if, for another instance x_n of x, the sequence p^2 p^1 x_n occurs in text T.
5. (f^1 and f^2 in text T, word case) Define $Z_5^{x_m} = 1$ if both f^1 and f^2 are words and if, for another instance x_n of x, the sequence x_n f^1 f^2 occurs in text T.
6. (p^1 in text T, non-word token case) Define $Z_6^{x_m} = 1$ if p^1 is a non-word token and if, for another instance x_n of x, the sequence p^1 x_n occurs in text T.
7. (f^1 in text T, non-word token case) Define $Z_7^{x_m} = 1$ if f^1 is a non-word token and if, for another instance x_n of x, the sequence x_n f^1 occurs in text T.
8. (p^1 in text T, part-of-speech case) Define $Z_8^{x_m} = 1$ if the following two conditions are satisfied. First, p^1 must be a word and is estimated to have a dominant part-of-speech. Second, a sequence q x_n must occur in text T where x_n is another instance of x and where q is a word having an estimated dominant part-of-speech equa
9. (f^1 in text T, part-of-speech case) Define $Z_9^{x_m} = 1$ if the following two conditions are satisfied. First, f^1 must be a word and is estimated to have a dominant part-of-speech. Second, a sequence x_n q must occur in

text T where x_n is another instance of x and where q is a word having an estimated dominant part-of-speech equal to that estimated for f^1.

10.–18. (Text H) Define $Z_{10}^{x_m}$–$Z_{18}^{x_m}$ like $Z_1^{x_m}$–$Z_9^{x_m}$ except that, in each case, the specified sequences must be in text H instead of text T.

The characteristic vectors of the sets $G(v)$, $B_{v \leftarrow w}(v)$, $G(w)$, and $B_{v \leftarrow w}(w)$ are constructed by the above rules when one takes x_m of the above rules to be v_i, v_j, w_j, and w_i, respectively, and selects the applicable texts. The characteristic vectors needed in the testing step are constructed analogously.

Some previous work in word sense disambiguation (for example, see Bruce and Wiebe (1994, 1999), Pedersen, Bruce, and Wiebe (1997), Pedersen and Bruce (1998), Pedersen (1999)) uses similar encodings where words, parts-of-speech, and morphological features near a given word instance are recorded. Here, our list of parts-of-speech has 46 items that accommodate all morphological subcases. Ignoring that minor variation, the main difference between the cited methods and the one proposed here is the use of dominant parts-of-speech instead of just parts-of-speech, the use of a reference text for both learning and testing, and the way the characteristic vectors are evaluated.

Appendix D: Classification of characteristic vectors

We describe how the set $L_{v \leftarrow w}(v)$ (resp. $L_{v \leftarrow w}(w)$) is used to estimate whether a given characteristic vector is in $G(v)$ or $B_{v \leftarrow w}(v)$ (resp. $G(w)$ or $B_{v \leftarrow w}(w)$). It suffices to examine how $L_{v \leftarrow w}(v)$ is applied to the characteristic vector $t(v_k)$ of an instance v_k of v. The set $L_{v \leftarrow w}(v)$ consists of 20 disjunctive normal form (DNF) logic formulas and 20 conjunctive normal form (CNF) logic formulas. Each of the 40 formulas produces for $t(v_k)$ a *vote* of $+1$ or -1. A $+1$ (resp. -1) indicates that the logic formula estimates $t(v_k)$ to be in $G(v)$ (resp. $B_{v \leftarrow w}(v)$). Let the sum of these 40 votes be the *vote-total* $r(t(v_k), L_{v \leftarrow w}(v))$. Since each vote is equal to $+1$ or -1, the vote-total $r(t(v_k), L_{v \leftarrow w}(v))$ is even and may range from -40 to 40. Furthermore, if $r(t(v_k), L_{v \leftarrow w}(v))$ is close to 40 (resp. -40), then $t(v_k)$ is likely to be in $G(v)$ (resp. $B_{v \leftarrow w}(v)$). For example, a vote-total $r(t(v_k), L_{v \leftarrow w}(v))$ equal to 40 means that all of the 40 formulas has estimated $t(v_k)$ to be in $G(v)$. But how are positive or negative vote-totals near 0, or 0 itself, to be interpreted? The data mining algorithm Lsquare estimates probability distributions for the vote-totals that one may be tempted to use for the answer. But such use assumes that the testing text comes, statistically speaking, from the same population as the training text. But we only know that these two texts are in the same domain area. Thus, the two texts are not guaranteed to satisfy the assumption. A few test cases have confirmed that the assumption may indeed not be satisfied. For this reason, we do not make use of the probability distributions. Instead, we compute from the testing text an odd integer *threshold* $\alpha_{v \leftarrow w}(v)$, $-40 < \alpha_{v \leftarrow w}(v) < 40$, to decide if $t(v_k)$ should be declared to be in $G(v)$ or $B_{v \leftarrow w}(v)$. Recall that $r(t(v_k), L_{v \leftarrow w}(v))$ is even, so $r(t(v_k), L_{v \leftarrow w}(v)) = \alpha_{v \leftarrow w}(v)$ is not possible. We estimate $t(v_k)$ to be in $G(v)$, and thus to be good, if $r(t(v_k), L_{v \leftarrow w}(v)) > \alpha_{v \leftarrow w}(v)$, and estimate $t(v_k)$ to be in $B_{v \leftarrow w}(v)$, and thus to be bad, if $r(t(v_k), L_{v \leftarrow w}(v)) < \alpha_{v \leftarrow w}(v)$. The computation and use of $\alpha_{v \leftarrow w}(w)$ mimics that of $\alpha_{v \leftarrow w}(v)$, so we only discuss the case of $\alpha_{v \leftarrow w}(v)$.

Given a substitution $v \leftarrow w$, for each instance v_k of a word v in the given testing text, the testing step determines a characteristic vector $t(v_k)$. Furthermore, for each instance w_l of a word w in the given testing text, w_l is temporarily replaced by v_p and a characteristic vector $f_{v \leftarrow w}(v_p)$ is computed. Finally, the testing step applies $L_{v \leftarrow w}(v)$ to each $t(v_k)$ and to each $f_{v \leftarrow w}(v_p)$, getting vote-totals $r(t(v_k), L_{v \leftarrow w}(v))$ and $s(f_{v \leftarrow w}(v_p), L_{v \leftarrow w}(v))$, respectively. The threshold is derived from these vote-totals. Before we describe the computations, let us try to predict the behavior of the vote-totals.

Suppose no instance v or w in the testing text involves a context-based spelling error. Then the instances v_k are good, and the instances v_p, which are derived from instances w_l, are bad. The learning step has created logic formulas that produce positive vote-totals for good instances and negative vote-totals for bad instances. Assuming that the testing text is similarly structured as the training text, we therefore expect that the vote-totals

$r(t(v_k), L_{v\leftarrow w}(v))$ for the instances v_k are positive and that the vote-totals $s(f_{v\leftarrow w}(v_p), L_{v\leftarrow w}(v))$ for the instances v_p are negative. Of course, this need not be so. But at least one may reasonably expect that most if not all $r(t(v_k), L_{v\leftarrow w}(v))$ values are greater than most if not all $s(f_{v\leftarrow w}(v_p), L_{v\leftarrow w}(v))$ values.

The above discussion supposes that no v or w involves a context-based spelling error. In Section 1, we assumed that such errors are rare, so we may suppose rather reasonably that at most one instance of v and at most one instance of w is involved in a context-based spelling error. Regardless of the specific situation, any such error most likely involves a v_k with smallest vote-total $r(t(v_k), L_{v\leftarrow w}(v))$ or a v_p with largest vote-total $s(f_{v\leftarrow w}(v_p), L_{v\leftarrow w}(v))$. Of course, we do not know if such an error is present. But we do not want the threshold computations to be affected by such errors. So, as a precautionary measure, we delete the smallest vote-total from the list of $r(t(v_k), L_{v\leftarrow w}(v))$ and sort the remaining entries. We end up with a sorted list of vote-totals, say, r_1, r_2, … r_m with r_1 largest, and know that these vote-totals very likely correspond to good instances of v. Similarly, we delete the largest vote-total from the list of $s(f_{v\leftarrow w}(v_p), L_{v\leftarrow w}(v))$ and sort the remaining entries. We end up with a sorted list of vote-totals, say, s_1, s_2, … s_n with s_1 largest, and know that these vote-totals very likely correspond to bad instances of v. Note that the above arguments crucially depend on the assumption of Section 1 that errors involving a given word are rare. There may be situations where a person makes the same error repeatedly. For example, the person may repeatedly confuse "it" and "it's" or "complement" and "compliment." In that case, the threshold computed next may still allow such errors to be caught. But the probability that this will take place is reduced. In Section 5, a modification of Ltest is described that, over time, leads to improved detection of such systematic errors.

Recall that the testing step estimates an instance of v to be good (resp. bad) if the vote-total is above (resp. below) the threshold $\alpha_{v\leftarrow w}(v)$. Hence, if the smallest r_i, which is r_m, is larger than the largest s_j, which is s_1, then we pick $\alpha_{v\leftarrow w}(v)$ about halfway between r_m and s_1. If $r_m \leq s_1$, we want a compromise value for $\alpha_{v\leftarrow w}(v)$ that minimizes the sum of the number of r_i below $\alpha_{v\leftarrow w}(v)$ and the number of s_j above $\alpha_{v\leftarrow w}(v)$. The computations below reflect these ideas, but also rely on the notion that, in case several threshold values equally well achieve the stated goal, then, among these, the threshold value closest to 0 is preferred.

The above computations can be carried out only if each of the words v and w occurs at least twice in the testing text. In the situations where the testing step requires thresholds, two instances of v are guaranteed to exist. However, w may occur just once, and thus the vote-totals s_1, s_2, … s_n may not exist. In that exceptional case, the single instance of w may itself constitute a context-based spelling error, and we are reluctant to rely on that instance to make decisions regarding the instances of v. Instead, we define the threshold $\alpha_{v\leftarrow w}(v)$ to be equal to -39. This means that we are very conservative in estimating an instance of v to be in error and that we do so only if the vote-total is equal to -40.

Computation of threshold

INPUT: Sorted vote-totals $r_1, r_2, \ldots\ r_m$ and possibly $s_1, s_2, \ldots\ s_n$.

OUTPUT: Threshold $\alpha_{v \leftarrow w}(v)$.

1. If $s_1, s_2, \ldots\ s_n$ do not exist, define $\alpha_{v \leftarrow w}(v) = -39$, and stop.
2. If $r_m > s_1$: Define $\alpha_{v \leftarrow w}(v) = (r_m + s_1)/2$. Reduce (resp. increase) $\alpha_{v \leftarrow w}(v)$ by 1 if $(r_m + s_1)/2$ is even and greater than (resp. less than or equal to) 0. Stop.
3. $(r_m \leq s_1)$ Select an odd-valued $\alpha_{v \leftarrow w}(v)$ so that the number of r_i below $\alpha_{v \leftarrow w}(v)$ plus the number of s_j above $\alpha_{v \leftarrow w}(v)$ is minimum. If there is a choice, pick among them the value closest to 0. Stop.

Index