Context-sensitive spell checking based on word trigram probabilities

Suzan Verberne

Context-sensitive spell checking based on word trigram probabilities

Master thesis Taal, Spraak & Informatica
University of Nijmegen

Suzan Verberne
February – August 2002

Supervisors:
Dr. Inge de Mönnink (KUN)
Drs. Remco van Veenendaal (Polderland)

Index

Voorwoord

De Volkskrant van 11 juli 2002 bericht van een spelfout op het graf van de op 6 mei 2002 overleden politicus Pim Fortuyn. Op zijn graf stond in eerste instantie de volgende tekst, uit een gedicht van Roland Holst:

> *Ik zal de halmen niet meer zien*
> *nog binden ooit de volle schoven*
> *maar doe mij in de oogst geloven*
> *waarvoor ik dien...*

De Volkskrant schrijft:

> *Marten Fortuyn erkent dat hij verantwoordelijk is voor de fout. Het was hem niet opgevallen dat het woordje 'nog' eigenlijk 'noch' moest zijn: 'De spellingscorrector op mijn computer zag het natuurlijk niet.'*

Marten Fortuyn heeft gelijk: de spellingscorrector kon niet detecteren dat *nog* in dit geval een spelfout was, omdat het zelf een bestaand woord is en dus voorkomt in het woordenboek van de spellingscorrector. Dit spellingscontroleprobleem – spel- en typfouten die resulteren in een bestaand woord – is het onderwerp van mijn scriptie.

Ik heb mijn scriptie geschreven als afsluiting van de vierjarige opleiding Taal, Spraak en Informatica aan de Katholieke Universiteit Nijmegen. In de onderzoeksvariant van deze studie wordt de scriptie gecombineerd met een stage. Gedurende mijn studie was ik met name geïnteresseerd in de taaltechnologische onderdelen, zoals automatisch vertalen, information retrieval en formele grammatica's. Omdat ik gedurende de laatste twee en een half jaar van mijn studie als student-assistent heb gewerkt bij de afdeling Taal en Spraak, was ik al bekend met de gang van zaken binnen de faculteit. Daarom koos ik ervoor extern stage te lopen, met als doel erachter te komen hoe het is om in een bedrijf te werken.
Toen ik dr. Peter-Arno Coppen vertelde over mijn stageplannen, noemde hij Polderland Language & Speech Technology als mogelijk bedrijf om stage bij te lopen. De informatie op de website van Polderland sprak me erg aan; mijn keuze voor Polderland als stagebedrijf was dan ook niet moeilijk. Ik hoopte tijdens mijn stage uit te vinden of ik het leuk vind om te werken aan praktisch onderzoek en praktische toepassingen. Bovendien wilde ik graag kennis maken met de rol die een taal- en spraaktechnoloog kan spelen in het moderne toegepaste onderzoek.

Na een paar gesprekken met Polderland groeide het idee om onderzoek te doen in het kader van spellingscorrectie. Dit leek me een interessant onderzoeksgebied. Na wat denkwerk en met de hulp van mijn begeleider dr. Inge de Mönnink, besloot ik om onderzoek te doen naar het gebruik van contextinformatie voor het oplossen van het probleem van fouten die resulteren in bestaande woorden.

Van 1 februari tot 1 juni 2002 heb ik stage gelopen bij Polderland en ben ik praktisch bezig geweest met mijn onderzoek. Daarna heb ik – ook bij Polderland – deze scriptie geschreven. Gedurende mijn periode bij Polderland heb ik ook praktische werkervaring kunnen opdoen doordat ik heb meegedraaid met de dagelijkse gang van zaken en heb meegewerkt aan enkele projecten. Dit is mij allemaal erg goed bevallen.

Met uitzondering van dit voorwoord heb ik mijn scriptie in het Engels geschreven. Argumenten daarvoor waren ten eerste dat mijn onderzoek een spellingscontrole voor het Engels bestudeert en ten tweede dat alle literatuur over het onderwerp in het Engels was.

Dan rest mij nu nog het bedanken van enkele mensen die mij hebben geholpen bij het schrijven van mijn scriptie. Ten eerste natuurlijk mijn begeleider dr. Inge de Mönnink, die zeer betrokken is geweest bij mijn scriptie door alle tekst zowel inhoudelijk als tekstueel kritisch te bekijken en steeds nuttig commentaar te geven. Ook drs. Remco van Veenendaal, mijn stagebegeleider bij Polderland, is zeer behulpzaam geweest met allerlei praktische en inhoudelijke zaken, met name gedurende mijn stageperiode. Daarnaast wil ik mijn Polderland-kamergenoot drs. Olaf Seibert bedanken voor zijn praktische hulp en Perl-vaardigheden, die onmisbaar zijn gebleken tijdens mijn onderzoek. Tot slot bedank ik dr. Nelleke Oostdijk, die tweede lezer is geweest van mijn scriptie.

Wie na dit voorwoord nog steeds geïnteresseerd is in mijn scriptie, wens ik veel plezier met het lezen ervan.

Suzan Verberne
Nijmegen, september 2002

1. Introduction

Ode to the Spell Checker

Eye halve a spelling checker
It came with my pea sea
It plainly marques four my revue
Miss steaks eye kin knot sea.

Eye strike a key and type a word
And weight four it two say
Weather eye am wrong oar write
It shows me strait a weigh.

As soon as a mist ache is maid
It nose bee fore two long
And eye can put the error rite
Its rare lea ever wrong.

Eye have run this poem threw it
I am shore your pleased two no
Its letter perfect awl the weigh
My checker tolled me sew.

Norman Vandal

When a modern spell checker for English would be used to spell check the poem of Norman Vandal above, it would find no errors. This is not because the text contains no errors, but because all errors resulted in existing words. In line 7 the clause *Whether I am wrong or right* is intended, but the words *whether*, *I* and *right* are replaced by respectively *weather*, *eye* and *write*. All three words sound like the intended word but have a different meaning. The words are considered erroneous because they are different from the intended word, in spite of the fact that they appear in every dictionary. This type of error is referred to as *real-word errors*. All modern commercial spelling error detection and correction tools work on word level and use a lexicon. Every word from the text is looked up in the speller lexicon. When a word is not in the lexicon, it is detected as an error. In order to correct the error, a spell checker searches the lexicon for words that resemble the erroneous word most. These words are then suggested to the user who chooses the word that was intended. Since a lexicon-based spell checker cannot detect real-word errors, these errors are not corrected. This is referred to as *the problem of real-word errors*. Since spelling error detection and correction on word level cannot solve this problem, research into automatic

spelling error detection and correction now focuses on the development of spell-checking algorithms that make use of context. Spelling error detection and correction techniques that aim at detecting and correcting real-word errors are thus also referred to as *context-sensitive spell checking techniques*.
In the current research, I aim at creating a context-sensitive spell checking method that is able to detect and correct all kinds of human-generated real-word errors. Starting point for the context-sensitive spell checker is a lexicon-based spelling error detection and correction application that detects and corrects non-word errors (errors not yielding a lexicon entry). The context-sensitive spell checker can be combined with this lexicon-based spelling error detection and correction application in order to create an application that is able to detect and correct non-word errors as well as real-word errors. In the current research, I do not aim to create such an application. In fact, I will design and build both a lexicon-based spell checker and a stand-alone context-sensitive spell checker, in order to be able to test them separately.

The method for context-sensitive spelling error detection and correction that is used in the current research considers three-word sequences (*word trigrams*) instead of single words. The main idea behind this is that the misspelling of a word often results in an unlikely sequence of (three) words. For example, suppose one misspells *of* as *off* in the sequence *one of them*. I expect the relative frequency of the trigram *one off them* to be much smaller than the relative frequency of *one of them*. This probability information can be used for detecting the trigram *one off them* as wrong. In order to correct an erroneous trigram, the context-sensitive spell checker tries to find suggestion trigrams. This is done by changing the original words in the trigram into relatively similar words. These new words are then combined in every way possible, resulting in various suggestion trigrams.

The main goal of this research is finding out whether context-sensitive spell checking based on word trigram probability information is a valid solution to the problem of real-word errors. For this purpose, I have formulated two main research questions:

1. What proportion of real-word errors can be detected and corrected using context-sensitive spell checking based on word trigram probability information?

2. To what extent do the results of a lexicon-based spell checker improve when the context-sensitive spell checker is combined with the lexicon-based spell checker?

In chapter 2, *error classifications*, I discuss various error classifications that can be found in the literature and introduce the error classification that is used in the current research for building and testing a spell checking application that combines a context-sensitive spell checker with a lexicon-based spell checker. Chapter 3, *automatic spelling error detection and correction*, deals with all aspects of automatic spelling error detection and correction. The process of detection and correction in general and the performance of spell checking techniques are described first. After that, a frequently used modern-spell checking technique is described in detail, followed by a

discussion of some problems for modern spell checking applications. One of these problems is the problem of real-word errors. Subsequently, some existing techniques for detecting and correcting real-word errors are discussed.

In chapter 4, *building a context-sensitive spell checker*, I describe the context-sensitive spell checking method and algorithm that are used in the current research. First, the lexicon-based spell checker, which is the starting point for the spelling error detection and correction tool, is described in detail. Then I describe the method for building the context-sensitive spell checker, followed by a comparison to other context-sensitive spell checking methods.

Chapter 5, *performance of the spell checking application*, describes and discusses the results of the current research. First, some general evaluation techniques for spelling error detection and correction systems are described. Then the performances of both the lexicon-based spell checker and the context-sensitive spell checker are described and discussed.

In chapter 6, *conclusion*, I draw some conclusions from the current research. The two main research questions are answered first, followed by an evaluation of the current research. Then the results are compared to the results of other studies. Finally, I give some recommendations for further research.

2. Error classifications

As described in chapter 1, context-sensitive spell checking aims at detecting and correcting real-word errors. In order to create a valid method for context-sensitive spell checking, it is important to define real-word errors properly. Moreover, it is important to know what kinds of errors occur and how errors can be classified. For the current research, I aim at making a valid error classification for building and testing a spell checking application that extends a lexicon-based spell checker with context-sensitive error detection and correction. In order to make a good classification of errors, several error classifications as found in the literature are discussed first (section 2.1). In section 2.2, the classification that is used for the current research is described.

2.1. Error classifications in the literature

In the literature, errors have been classified in different ways. One classification does not exclude another. In this section, I consider the different classifications in order to find a classification that can describe all errors. The classifications that are described in this section come from several studies. In the next subsection (2.1.1), these studies are described. In section 2.1.2, the error classifications from these studies are discussed.

2.1.1. Relevant studies

Kukich (1992b) describes many error classifications that have been explored by several studies. I consider seven of these studies relevant for the current research. All of the studies that are mentioned in this section are taken from Kukich (1992b).
Studies often present different results when determining the frequencies of occurrence of several error classes. These differences can be explained by the characteristics of the studies: the size of the corpus from which the errors have been taken, the text type, the error types taken into account and the number of errors considered. Table 2-1 gives an overview of these features for the relevant studies. A question mark indicates that the information is not available for that particular study. The column *# of errors* contains two values: the absolute number of errors and (between brackets) the percentage of erroneous words of all words in the corpus. These percentages indicate that the error rate varies with text type. Unfortunately, the error rates for Kukich (1990) and Young, Eastman and Oakman (1991) are unknown. Probably, the error rate is high in the written conversations studied by Kukich (1990) and low in the natural language queries studied by Young, Eastman and Oakman, but this is just a presumption.
Considering the differences between the studies shown in table 2-1, it is plausible that the frequencies of particular error types differ. This is described in the next section.

Table 2-1 Short description of relevant studies into error classification (all described by Kukich (1992b)

Study	*Corpus size*	*Text type*	*# of Errors*	*Error types*
Wing and Baddeley (1980)	80,000 words	Handwritten essays of Cambridge college applicants	1,185 (1.5%)	No typing errors[1]
Pollock and Zamora (1983 and 1984)	25,000,000 words	Scientific and scholarly text	50,000 (0.2%)	No real-word errors
Yannakoudakis and Fawthrop (1983)	1,014,312 + 60,000 words	Brown corpus and texts written by high-educated bad spellers	1,377 (0.1%)	All sorts of errors
Mitton (1987)	170,016 words	Handwritten student compositions	4218 (2.5%)	No typing errors[1]
Kukich (1990)	?	Written conversations between deaf people	2,000	All sorts of errors
Young, Eastman and Oakman (1991)	426 natural language queries	Natural language queries to a document retrieval system	?	?
Kukich (1992a)	40,000 words	Typed textual conversations	?	All sorts of errors

[1] The corpus of this study does not contain typing errors since the text was hand-written

2.1.2. Classifications described in the literature

In this section, six classifications are described that have been taken from Kukich (1992b) and that have been in described the studies mentioned in section 2.1.1

Typing errors vs. spelling errors

A first error classification that is found in the literature is that between typing errors (sometimes referred to as *typos*) and spelling errors[2]. Typing errors are slips of the keyboard, analogous to slips of the pen. A writer who produces a typing error or slip of the pen knows the correct spelling of the word but makes a mistake when typing or writing the word. Since it is not likely for a writer to make many mistakes within one word, most erroneous words resulting from a typing error orthographically resemble the intended word (e.g. *front* → *fron*). Spelling errors, on the other hand, result from the writer's ignorance of the correct spelling. There are three possible causes for spelling errors. First, the erroneous word phonetically resembles the intended word (e.g. *memories* → *memerys*). In case of these phonetic errors, the resulting string is a homophone or near-homophone of the intended word. Second, the erroneous word bears a semantic similarity to the intended word (e.g. *council* → *consul*). Third, the writer is ignorant of a grammatical rule that should be applied. This can lead to incorrect use of subject-verb congruence (e.g. *he does* → *he do*), wrong forms of irregular verbs (e.g. *I ran* → *I runned*), or other grammatical mistakes.[3] Although many studies mention the distinction between typing errors and spelling errors, no research that I know of has been done into the frequency of these two classes. There has only been some research into the frequency of phonetic errors: Mitton (1987) found that 44% of all errors in his corpus involved homophones or near-homophones.

Single error words vs. multiple error words

A second error classification that is found in the literature is the distinction between erroneous words containing a single error and erroneous words containing multiple errors. In all studies described by Kukich (1992b), the definition of Damerau (1964) is used. He states that single-error misspellings are those erroneous words that contain a single instance of one of the following four character transformations: one character inserted, one character deleted, one character substituted or a transposition of two characters. These single transformations are sometimes referred to as *Damerau transformations*. Misspellings that contain more than one of these character transformations are referred to as multi-error misspellings.

---

[2] Often, the term *spelling error* is used for both spelling errors and typing errors. This is also the case in the current research. Where the term can yield confusion, the meaning is explicitly mentioned. The term *automatic spelling error detection and correction* always refers to both spelling errors and typing errors.

[3] Sometimes, grammatical errors are not considered spelling errors but a separate category of errors. One could argue that grammatical errors are therefore not in the scope of spelling error detection and correction. In the current research, all grammatical errors that yield only one erroneous word (i.e. the sentence is corrected by changing just one word) are considered a subcategory of spelling errors.

It is not exactly clear what the frequency of single errors and multiple errors is, since several studies obtained different results. Damerau (1964) found that approximately 80% of all misspelled words are single-error misspellings; Mitton (1987) found that 69% of the misspellings in his corpus were single errors. And Pollock and Zamora (1984) found that 94% of all spelling errors they studied were single-error misspellings.
The large difference between the percentages found by Mitton (1987) and Pollock and Zamora (1984) (respectively 69% and 94%) can be explained by the fact that Mitton did not consider typing errors since he studied handwritten texts, whereas Pollock and Zamora used spelling errors as well as typing errors. Assuming that slips of the pen occur less frequently than slips of the keyboard, Pollock and Zamora's corpus probably contained relatively more slips than Mitton's corpus. As described above, erroneous words due to typing errors (i.e. slips) orthographically resemble the intended word more than erroneous words due to spelling errors. This implies that the frequency of single-error misspellings is higher for typing errors than for spelling errors. This explains the lower single-error rate in the corpus used by Mitton. The difference in frequencies may also be explained by corpus size, error type and text type: Pollock and Zamora used a much larger corpus and considered only non-word errors, whereas Mitton used a much smaller corpus and considered real-word errors as well as non-word errors. Their corpora also consisted of different text types. Unfortunately, there is no research I know of that shows a connection between corpus size, error type and text type on the one hand and single-error rates on the other.

Errors in short words vs. errors in long words
A third distinction in errors is that between errors occurring in short words and errors occurring in long words. What is the relation between word length and error frequency? There is a wide variance in the proportion of errors that occur in short words according to different studies.
Pollock and Zamora (1983) found that errors in words of three and four characters constitute 9.2% of all errors. Yannakoudakis and Fawthrop (1983) found that the frequency of errors occurring in short words was about 1.5%, whereas Kukich (1990) found that over 63% of the errors occurred in words of two, three and four characters. The question arises where this large difference between 1.5%, 9.2% and 63% comes from. Firstly, the definition of short words is not the same for all three studies. Kukich considered words of length two, three and four characters, whereas Pollock and Zamora and Yannakoudakis and Fawthrop considered words of three and four characters. However, the different definitions of short words is not expected to fully explain the differences in short-word error frequencies that were found. Secondly, the text types that have been used for counting the frequencies are very different: Yannakoudakis and Fawthrop used texts from the Brown corpus and texts written by high-educated adults and Pollock and Zamora used a scientific and scholarly corpus of texts, whereas Kukich used written conversations between deaf people. The written data used by Kukich probably resembles spoken language much more than the written data used by Yannakoudakis and Fawthrop or Pollock and Zamora. Therefore, in Kukich's data, average word length is expected to be smaller than in Yannakoudakis and Fawthrop's and Pollock and Zamora's data. If the frequency of short words in Kukich's texts is indeed higher than in Yannakoudakis and Fawthrop's and Pollock and Zamora's texts, the frequency of errors in short words will be higher too. Thus, text type plays an

important role in the frequency of occurrence of short-word errors. To know whether the large difference could be explained by text type only, more information on the methods of both studies is needed.

First character errors vs. nth character errors

Another possible classification of errors is that between errors occurring in the first character of a word and errors occurring in the rest of the word. The first character of a word can be deleted (*from* → *rom*), substituted (*from* → *grom*), transposed with the second character (from → *rfom*) or another character can be inserted before the first character (*from* → *dfrom*). An nth character error is an error that occurs in a character that is not the first character of the word.

There has been some research into the percentage of errors that occur in the first character of a word. Pollock and Zamora (1983) found that 3.3% of their 50,000 misspellings involved first characters; Yannakoudakis and Fawthrop (1983b) found a first-position typing error rate of 1.4%; Mitton (1987) found that 7% of all misspellings involved first-position errors. And Kukich (1992a) observed a 15% first-position error rate.

The differences between these results (respectively 3.3%, 1.4%, 7% and 15%) can first of all be explained by the different text types: Yannakoudakis and Fawthrop used texts from the Brown corpus and texts written by highly-educated adults; Pollock and Zamora used scientific and scholarly texts, whereas Mitton studied handwritten texts from fifteen-year-olds and Kukich used written conversations between deaf people. Probably, the difference between the results of Yannakoudakis and Fawthrop (1.4%) and Pollock and Zamora (3.3%) is caused by the difference in corpus size. Especially the large amount of first-character errors found by Kukich is striking, but it can be explained by text type: as the corpus used by Kukich probably mostly resembles spoken language, it contains relatively short words. This can probably explain the high amount of first-character errors: an error in a three-character word is more likely to be a first-character error than an error in a ten-character word since the error can be in fewer different positions.

Errors within one word vs. word boundary errors

A fifth classification of errors that has been mentioned in the literature is that between errors within a word and word boundary errors. Word boundary errors are those errors that result from incorrect spacing. There are two types of word boundary error: incorrect splits (e.g. *together* → *to gether*) and run-ons (e.g. *a lot* → *alot*). There has been little research into the frequency of word boundary errors. Kukich (1992a) found that 15% of all errors were word boundary errors. Mitton (1987) found that 13% of his 4218 errors were word boundary errors.

Non-word errors vs. real-word errors

Finally, a classification can be made between non-word errors and real-word errors. Non-word errors are those errors that yield a character string that is not a valid word (e.g. *from* → *grom*). Real-word errors yield a character string that is a valid and correct word itself (e.g. *from* → *form*). There have been several studies into the frequency of real-word errors. Peterson (1986) studied the relation between lexicon size and real-word error rate. He only considered single error misspellings (Damerau, 1964). When a larger lexicon is used (and therefore more strings are

considered real words), a word is more likely to yield another lexicon entry when misspelled. Peterson found that the real-word error rate ranged from 2% for a small lexicon to 10% for a 50,000-word lexicon and almost 16% for a 350,000-word lexicon. Mitton (1987) found that 40% of all 4218 errors were real-word errors. Young, Eastman and Oakman (1991) found a real-word spelling error rate of 25%. Wing and Baddeley (1980) found that 30% of the errors were real-word errors. The percentage found by Peterson (16%) is much lower than the percentages found by others. This is because Peterson only considered errors resulting from single-error misspellings and the other studies also addressed errors resulting from more than one character transformation. The differences between the other three results (Mitton; Young, Eastman and Oakman and Wing and Baddeley) can mainly be explained by text type. Mitton and Wing and Baddeley both used a corpus of handwritten texts. The handwritten corpus used by Mitton probably contained relatively many short words, because it consisted of compositions written by fifteen-years-old students. Short words are more likely to yield a real word when misspelled (see section 3.5.2). This probably explains the high real-word error rate found by Mitton.
Unfortunately, no text types that are relevant for the current research have been studied: handwritten texts are never the input of a spelling error detection and correction system and natural language queries are much shorter than a text that usually is spell checked. Therefore, no choice is made for one of the studies' results, but the whole range from 25% to 40% is used as real-word error rate for the current research.

Classifications within real-word errors
Above, errors have been classified into non-word errors and real-word errors. In the literature, some subclassifications within the class of real-word errors are considered as well. Mitton (1987) divides real-word errors into three classes: wrong-word errors, wrong-form-of-word errors and word-division errors. Wrong-word errors are those errors where a semantically and grammatically different word was written instead of the correct one (e.g. *know* → *now*). Wrong-form-of-word errors are incorrect grammatical forms of the intended word, such as using incorrect tense (*yesterday I went* → *yesterday I go*) or incorrect number (*dozens of things* → *dozens of thing*). Word-division errors are those word boundary errors that happen to result in one or more correct words (e.g. *inside* → *in side*). Mitton found that 44% of all real-word errors were wrong-word errors; 24% were wrong-form-of-word errors and 32% were word-division errors. Within the class of word-division errors, most errors were incorrect splits; it was very rare for run-ons to produce a real word.
Kukich (1992b) chooses another classification of real-word errors. She distinguishes between the cause of an error and the result of an error. When discussing causes of errors, Kukich mentions six types:

1. Simple typos (e.g. *from* → *form*);
2. Cognitive or phonetic lapses (e.g. *ingenious* → *ingenuous*, *there* → *their*);
3. Syntactic or grammatical mistakes (e.g. *he arrives* → *he arrive*);
4. Semantic anomalies (e.g. *in five minutes* → *in five minuets*, *leave a message* → *lave a message*);

5. Insertions or deletions of whole words (e.g. *the system has been operating system for almost three years*);
6. Improper spacing (e.g. *myself* → *my self*, *ad here* → *adhere*).

Typos (class 1) have been described earlier in this section, when the classification into typing errors and spelling errors was discussed. In that classification, spelling errors were described as those errors that result from the writer's ignorance of the correct spelling: they can have been caused by phonetic resemblance, semantic resemblance or ignorance of a grammatical rule. This class of errors corresponds to the classes 2 and 3 of Kukich's cause-classification.
Semantic anomalies, class 4 in Kukich's classification, are sometimes called *malapropisms*. Sentences resulting from this kind of error are not syntactically incorrect but only semantically. When someone mistypes *minutes* as *minuets* in the context *in five minutes*, the resulting word sequence is not syntactically incorrect, but it is semantically incorrect, because it has another meaning than the user had intended.
Insertions or deletions of whole words (Kukich's class 5) always yield a real-word error, unless a writer inserts an incorrectly spelled word. This class of errors differs from the classes 1 to 4 in that errors due to word insertions or deletions do not yield an incorrect word that has to be corrected, but a sentence that contains one word too much or too few. Errors due to improper spacing (class 6) have a similar property: the resulting sentence also contains an incorrect number of words. These word-division errors can be incorrect splits as well as run-ons.
The classes 1, 2 and 4 in Kukich's classification correspond to the class of wrong-word errors in Mitton's classification. Grammatical errors (Kukich's class 3) correspond to Mitton's wrong-form-of-word errors. Both Kukich and Mitton consider a separate class of errors due to improper spacing: word-division errors (class 6). Mitton does not consider Kukich's class 5, insertions or deletions of whole words, as a class of real-word errors.

Kukich does not only classify real-word errors according to cause. She also classifies real-word errors according to possible results. When considering results of real-word errors, Kukich mentions four classes:

1. Syntactic errors (e.g. *The students are doing <u>there</u> homework*);
2. Semantic errors (e.g. *He spent his summer travelling around the <u>word</u>*);
3. Structural errors (e.g. *I need <u>three</u> ingredients: red wine, sugar, cinnamon and cloves*);
4. Pragmatic errors (e.g. *He studies at the University of Toronto in <u>England</u> and she studies at Cambridge*).

Sentences that contain a syntactic error (class 1) are syntactically incorrect. The cause of this incorrectness is substituting a word by another word that has another part of speech. When someone substitutes *their* by *there* in the sentence *The students are doing their homework*, a possessive pronoun is substituted by an adverb. This substitution results in a grammatically incorrect sentence.

Sentences that contain a semantic error (class 2) are semantically incorrect, but syntactically correct. This is the case when the writer substitutes a word by another word that has the same part of speech. When someone mistypes *world* as *word* in the context *travelling around the world*, the resulting word sequence is not syntactically incorrect, but it is semantically incorrect, because it has another meaning than the user had intended.
Structural errors (class 3) violate the inherent coherence relations in a text, such as an enumeration violation. These errors are sometimes referred to as discourse structure errors. Pragmatic errors (class 4) reflect some anomaly related to the goals and plans of the discourse participants. In case of structural and pragmatic errors, the writer makes an error of thought. It is not necessarily the case that the resulting word was not intended by the writer.

2.2. A two-dimensional classification

For the current research, I need an unambiguous classification of (real-word) errors. The classification I have chosen is based on Kukich's classification of real-word errors described above. Kukich distinguishes between the cause of an error and the result of an error. I use both classifications to arrive at a two-dimensional classification for the current research. However, Kukich's classification is not unambiguous. For this reason, I had to make some changes.
Kukich mentions six classes when discussing causes of errors:

1. Simple typos (e.g. *from* → *form*);
2. Cognitive or phonetic lapses (e.g. *ingenious* → *ingenuous*, *there* → *their*);
3. Syntactic or grammatical mistakes (e.g. *he arrives* → *he arrive*);
4. Semantic anomalies (e.g. *in five minutes* → *in five minuets*, *leave a message* → *lave a message*);
5. Insertions or deletions of whole words (e.g. *the system has been operating system for almost three years*);
6. Improper spacing (e.g. *myself* → *my self*, *ad here* → *adhere*).

In my opinion, the last three categories are not causes of errors themselves but can all be placed in one of the first two categories. Semantic anomalies (class 4) are the result of either a typo (*in five minuets, lave a message*) or a cognitive or phonetic lapse (*an ingenuous machine*, *Can I have a peace of cake please?*). Insertions or deletions of whole words (class 5) cannot be caused by a typo and have to be caused by a cognitive lapse. Improper spacing (class 6) can be caused by either typos (*I refer red to yesterday's meeting*) or cognitive lapses (*The students are doing their home work*). For my own classification, I will consequently only adopt Kukich's categories 1, 2 and 3 as causes of errors.
Kukich distinguishes four types of real-word error results:

1. Syntactic errors (e.g. *The students are doing <u>there</u> homework*);
2. Semantic errors (e.g. *He spent his summer travelling around the <u>word</u>*);

3. Structural errors (e.g. *I need three ingredients: red wine, sugar, cinnamon and cloves*);
4. Pragmatic errors (e.g. *He studies at the University of Toronto in England and she studies at Cambridge*).

In my opinion, structural and pragmatic errors are not spelling or typing errors but can perhaps be referred to as knowledge errors: the writer makes an error of thought. It is not necessarily the case that the resulting word was not intended by the writer, whereas in the current research, a word is considered erroneous if it is not the word that the user had intended. Consequently, the distinction between the intended word and the erroneous word has been made in this section and is also made in following sections. Therefore, in my classification I will not consider the categories 3 and 4. I classify real-word errors in two categories: syntactic errors and semantic errors. The former category contains those errors that yield a syntactically incorrect sentence, whereas the latter category contains those errors that yield a syntactically correct but semantically incorrect sentence.
Obviously, the two classifications suggested by Kukich are not mutually exclusive, but can be combined. Every (real-word) error is caused by either a typing error, a cognitive or phonetic lapse, or a grammatical mistake, and the result can be classified as either a syntactic error or a semantic error. In table 2-2 on page 15, this two-dimensional approach, which I adopt for my own research, is presented with possible causes of errors in rows and possible results in columns. Although Kukich's classification is a classification of real-word errors, it is equally applicable to non-word errors. Therefore, non-word errors have been added to the scheme. Non-word errors can also be caused by either a typo, a phonetic lapse or a grammatical mistake. The cells of table 2-2 contain an example of the category.

2.3. Conclusion

In this chapter, I aimed at finding a unambiguous classification of errors that can be used for the current research. In section 2.1, six classifications of errors have been described that can be found in the literature: spelling errors vs. typing errors, single errors vs. multiple errors, errors in short words vs. errors in long words, first character errors vs. nth character errors, errors within one word vs. word boundary errors and non-word errors vs. real-word errors. For the current research, especially the classification into real-word errors and non-word errors is important, since context-sensitive spell checking aims at detecting and correcting particularly this type of error. I studied the other five classifications in order to make an unambiguous classification for the current research. Moreover, several error classifications explored by others are important for their research into spelling error detection and correction techniques. Some of this research is described in chapter 3.
After considering several error classifications, I chose a two-dimensional classification in which the distinction between typing errors and spelling errors and the distinction between non-word errors and real-word errors play an important role.

Classifications within the class of real-word errors have also been proposed in the literature. Mitton (1987) and Kukich (1992b) did important research on this subject. For the current research, I adopted Kukich's classification into syntactic errors and semantic errors and added it to the two-dimensional classification. The classification that I adopted for my own research can be found in table 2-2.

Table 2-2 Classification of errors with some examples

Cause	Non-word errors	Real-word syntactic errors	Real-word semantic errors
Typo	*could* → *xould*	*from* → *form*	*minutes* → *minuets*
Cognitive or phonetic lapse	*among* → *amoung*	*their* → *there*	*ingenious* → *ingenuous*
Grammatical mistakes	*ran* → *runned*	*you arrive* → *you arrives*	*he arrived* → *he arrives*

3. Automatic spelling error detection and correction

3.1. Introduction

This chapter deals with automatic spelling error detection and correction. But what exactly is automatic spelling error detection and correction? This question comprises in four subquestions:

1. What is a spelling error?
2. What is detection?
3. What is correction?
4. What is automatic?

What is a spelling error?
In the current research spelling errors are defined as human-generated writing errors. As described in section 2.1, the term *spelling error* sometimes refers to both spelling errors and typing errors: automatic spelling error detection and correction aims at detecting and correcting both spelling errors and typing errors. In the current research this ambiguous denotation of the term *spelling error* is adopted. Also, the current research considers only human-generated spelling errors. While some techniques for detecting and correcting errors of optical character recognition (OCR) devices have been studied in the literature, most research has been done into techniques for detecting and correcting human-generated errors.

What is detection?
Error detection is the procedure of finding incorrectly spelled words in a text. A word that is considered incorrect is flagged by the spell checking application. If the word was indeed erroneous, the error has been detected.

What is correction?
Error correction is the procedure of correcting an error once it has been detected. An error is corrected when the spell checking application or the user replaces an erroneous word by the word that the user intended.[4] Sometimes, the term *error correction* is used to refer to the processes of error detection and correction together. In the current research, I consistently adopt the distinction between error detection and error correction.

What is automatic?
The term *Automatic spelling error detection and correction* can refer to both fully automatic spell checking techniques and spell checking techniques that interact with the user. Fully automatic

[4] In the current research it is assumed that in all cases the user of a spell checking application is the writer of the text that is spell checked. Therefore, the term *user* refers to the writer of the text.

systems detect and correct errors without any intervention by the user. Interactive systems detect errors, then suggest a number of possible corrections and allow the user to choose the word that should replace the erroneous word. In the current research the term *automatic error detection and correction* is used for both fully automatic error correction and interactive error detection and correction.
Automatic spelling error detection and correction is often referred to by using the shorter term *spell checking*. An application for spelling error detection and correction can then be referred to as *a spell checking application*.

In the current chapter, I aim to make a complete description of automatic spelling error detection and correction. In section 3.2, the process of automatic spelling error detection and correction is described. Section 3.3 discusses the performance of modern spell checking applications. Section 3.4 contains a detailed description of a modern spell checker. In section 3.5, three types of problem for spelling error detection and correction are discussed, followed by a description of techniques for real-word error detection and correction in section 3.6.

3.2. The process of automatic spelling error detection and correction

Research into automatic spelling error detection and correction has been carried out since the 1960s. Several methods for automatic spelling error detection and correction have been explored. The current research only addresses methods that aim at detecting and correcting human-generated errors. In techniques for human-generated error detection and correction a distinction is made between techniques that focus on error detection only and techniques that aim at detecting and correcting errors. The latter class can be subdivided into fully automatic correction techniques and interactive correction techniques. In case of a fully automatic system, error correction involves finding candidate corrections and choosing the most likely one. In case of an interactive system, candidate corrections are identified and then ranked. The user then chooses the correct word. Figure 3-1 on page 18 shows a graphical representation of this classification of error detection and correction techniques. This classification is applicable for both non-word error detection and correction and real-word error detection and correction. I will come back to this distinction in section 3.5.3.
As figure 3-1 shows, error correction consists of two tasks: first, finding candidate corrections and second, choosing the most likely correction (in case of a fully automatic system) or ranking the candidate corrections (in case of an interactive system). However, choosing the most likely correction also involves ranking, because the task is determining which candidate correction is the most likely to be intended by the user. The process of spelling error correction (fully automatic as well as interactive) can thus be divided into three tasks:

1. Error detection
2. Finding candidate corrections
3. Ranking candidate corrections

In the next three subsections, methods for these three tasks are described.

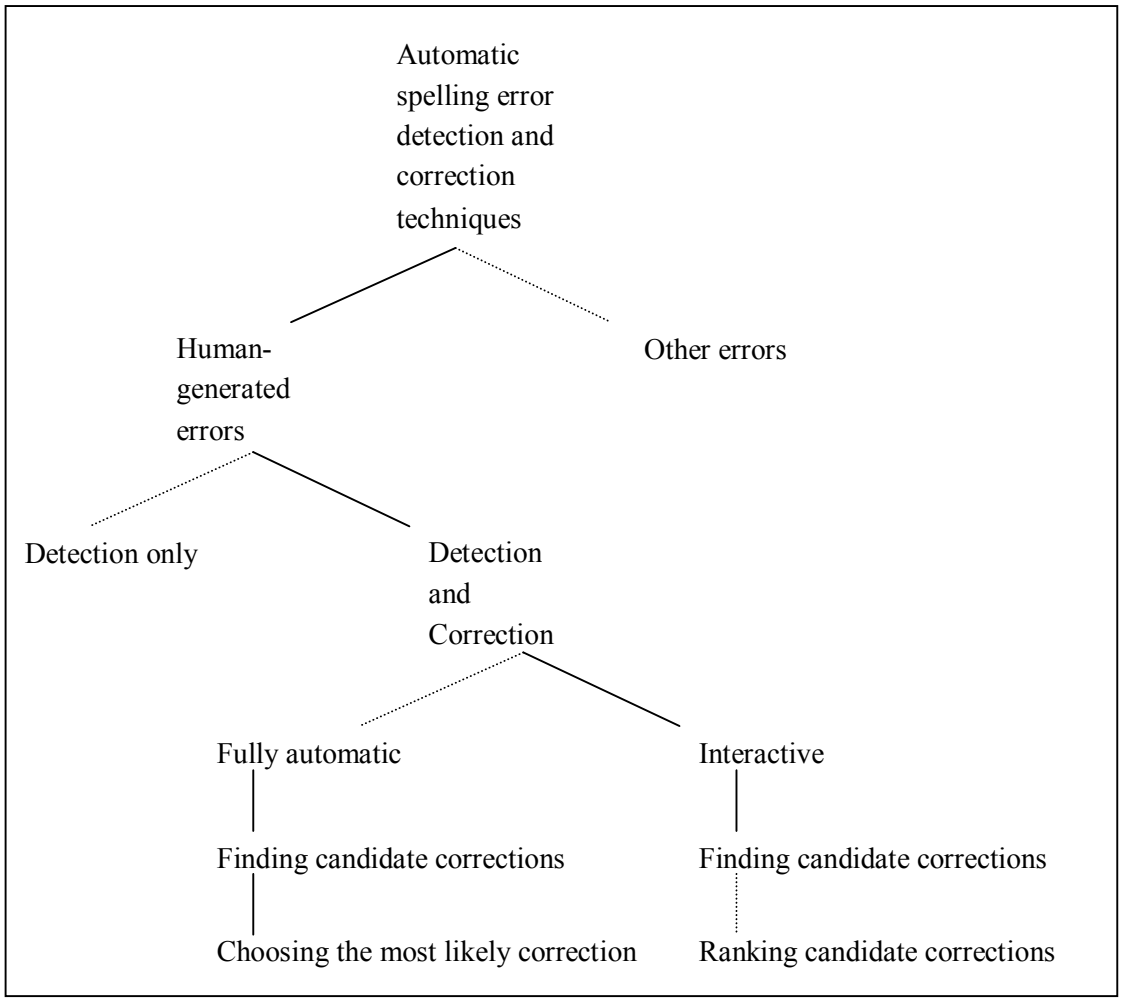


Figure 3-1 Classification of automatic spelling error detection and correction techniques

3.2.1. Error detection

In error detection, a subdivision is made between *non-word error detection* and *real-word error detection*. The distinction between non-word and real-word errors has been explained in section 2.1. Non-word error detection aims at detecting errors that do not result in lexicon entries. Techniques for non-word error detection are referred to as *isolated-word error detection techniques*. Real-word error detection is described in section 3.6.

Research into non-word error detection has mainly been carried out from the early 1970s until the early 1980s. Two main techniques have been explored for non-word error detection: character n-gram analysis and lexicon lookup. Character n-gram techniques work by examining each character n-gram in an input string and looking it up in a precompiled table of n-gram frequencies. Strings that contain non-existent or very infrequent n-grams (like *jtg* or *bkm*) are

detected as probable erroneous words. Character n-gram techniques require a large lexicon or corpus in order to compile the table of n-gram frequencies. Lexicon lookup techniques check for each word in the input text whether it is present in the spell checker lexicon. If the input word is not in the lexicon, it is detected as an error. Character n-gram techniques have mainly been used for detecting and correcting errors made by optical character recognition (OCR) devices. Lexicon-lookup techniques are more accurate than character n-gram techniques for detecting human-generated errors (Kukich, 1992b). Therefore, most techniques for detecting and correcting human-generated errors are lexicon-based. As the current research only addresses human-generated errors, I will only consider lexicon-based spell checking techniques in the rest of this chapter.

3.2.2. Finding candidate corrections

Once a string has been detected as an error, an error correction technique aims at finding candidate corrections for the erroneous word. Several algorithms for finding candidate corrections have been explored. The most popular method by far is computing the *minimum edit distance* between the detected string and a lexicon entry. The minimum edit distance has been defined as the minimum number of editing operations (i.e. insertions, deletions and substitutions) that is required for transforming one string into another. The first minimum edit distance spelling correction algorithm based on these three types of character transformation was implemented by Damerau (1964). Levenshtein (1966) developed a similar algorithm for correcting deletions, insertions and transpositions. Other researchers developed variants of the algorithms that were developed by Damerau and Levenshtein: Wagner and Fischer (1974) generalized it to cover also multi-error misspellings and Lowrance and Wagner (1975) extended the algorithm to account for some additional transformations, such as the exchange of nonadjacent characters. Some minimum edit distance algorithms that have been explored do not only use orthographic distance scores, but also phonetic similarities. Veronis (1988) devised an algorithm that calculates weights for the orthographic edit distance based on phonetic similarity. These weights are important to be able to find phonetic misspellings, because often, phonetic misspellings are a large number of editing operations removed from the intended word (see also section 2.1). If only orthographic information is taken into account, the intended word will most probably not be among the candidate corrections.
Minimum edit distance techniques have been applied to virtually all spelling correction tasks. An advantage of using a minimum edit distance measure is the fact that ranking can be performed easily. This is described in the next subsection.

3.2.3. Ranking candidate corrections

The goal of ranking is putting the most likely correction of the detected word at the top of the list of candidate corrections. The candidate corrections are presented to the user as ranked suggestions.

Ranking can be based on a similarity measure between the erroneous word and the suggestion or on word frequencies. The former possibility for ranking is useful in a system that uses a minimum edit distance technique for finding suggestions. Ranking can be based on this minimum edit distance. The candidate correction that has the smallest minimum edit distance to the detected string is put at the top of the suggestion list.
The latter method, word frequency, could also be taken into consideration for ranking. For example, suppose suggestions *anther*, *another* and *antihero* are found for the string *anither*. The relative frequency of *another* probably is higher than the relative frequency of *anther* and *antihero*. Based on word frequency, *another* could be ranked higher than *anther* and *antihero*.
A third possibility for ranking is combining minimum edit distance and word frequency. Some of the suggestions may have the same minimum edit distance. Then word frequency could be taken into consideration in order rank the suggestions properly.
If no ranking based on penalty value or word frequency is performed, the suggestions could be shown in alphabetical order.

3.3. Performance of automatic spelling error detection and correction techniques

All modern commercial spell checking techniques use a minimum edit distance algorithm for finding and ranking suggestions. To give a general idea of the performance of modern spell checkers, I will briefly describe the performance of three minimum edit distance algorithms. In order to interpret the results of these spell checking evaluations properly, it is important to know how correction is defined. In general, when measuring the performance of an interactive spelling correction algorithm that performs ranking, an error is defined as corrected when the correct suggestion is at the top of the suggestion list.
Damerau (1964) tested his minimum edit distance algorithm on a test set of 964 misspellings of words longer than five characters, using a lexicon of 1,593 words. He obtained a correction rate of 84%. Kukich (1990) used a test set of 170 misspellings, of which 25% involved multiple errors and 63% were words of two, three and four characters. She observed an overall correction rate of 62%, using a lexicon of 1,142 words. Muth and Tharp (1977) found a correction rate of 97% on a 1,487-word test set. Unfortunately, they did not specify the size of the lexicon they used or the characteristics of their test set.
Kukich (1992b) did some research into the performance of isolated-word correction techniques in general. She found that approximately 78% of non-word errors could be corrected by isolated-word correction techniques. I will return to the performance of isolated-word correction techniques in section 3.5.3.

3.4. A modern spell checking technique

The previous two sections give a general idea of the techniques used in modern spell checking applications and of their performance. In this section, a popular spell checking technique based on

a minimum edit distance algorithm is described in detail. Specifically, the algorithm that is used in the Polderland spell checkers is described. The design of the Polderland spell checkers functions as an example for modern commercial spell checking applications in general. Polderland spell checkers are interactive systems for detecting and correcting human-generated errors.[5]
In a lexicon-based spell checking application, the lexicon has two functions. First, the content of the spell checker lexicon defines which words are considered correct and which are not. If a word in the text is not in the spell checker lexicon, it is considered an error and therefore it is flagged by the spell checking application. Second, when a word is detected as an error, the spell checker lexicon is used for generating possible corrections. A frequently used technique for implementing the minimum edit distance algorithm in a spelling error correction application is the *trie*-structure lexicon. All Polderland spell checkers have a trie-structure lexicon. This lexicon structure is described in section 3.4.1. After that, the three basic steps of error detection and correction are described: detecting errors (3.4.2), finding candidate corrections (3.4.3) and ranking candidate corrections (3.4.4).

3.4.1. Structure of the spell checker lexicon

As mentioned above, the structure of the lexicon in Polderland spell checkers is a trie. A trie (from *retrieval*) is a frequently used technique for implementing the minimum edit distance algorithm in a spelling error correction application, because it is useful for storing strings over an alphabet. It has been used for storing large dictionaries in spell checking and natural language understanding applications.
An example of a trie structure is given in figure 3-2 on page 22. The words that have been stored in this trie are *an*, *ant*, *all*, *allot*, *alloy*, *aloe*, *are*, *ate* and *be*. All strings that share a common beginning hang off a common node. The top node in the picture represents the beginning of the string. By following a path in the trie, a string can be constructed. From the top node, the potential first characters are *a* and *b*. From the *a* node, the characters *l*, *n*, *r* and *t* are possible second characters. From *n* two paths can be chosen: the path to *t* and the path to the terminator node. If the terminator is chosen, the string is completed and the resulting word is *an*. When the strings are words with an {a..z} alphabet a node has at most 27 daughters – one for each following character plus a terminator. All strings in the trie can be found by a depth-first scan of the trie (Allison, 1999).
In the following three sections, error detection, finding candidate corrections and ranking candidate corrections using a trie-structure lexicon are described.

3.4.2. Error detection

In order to detect errors in the text, every word from the text is searched in the lexicon trie. When the trie has been entered, all possible daughters from the start node are visited until the first

[5] See figure 3-1 for the classification of automatic spelling error detection and correction techniques

character of the word is found. Then, the same is done for the second character and so on. If a terminator node and the end of the word are both encountered, the string is recognised as a lexicon entry. If there are no other possible paths from a node, the word cannot be found in the lexicon and is detected as an error.

For example, suppose a user writes the string *alliy*. When arriving at this word, the lexicon trie is entered. The first three characters (*all)* match a path in the trie (see figure 3-2). When arriving at the second *l*, the path to *i* is chosen (see figure 3-3 on page 23). Then a node is encountered where

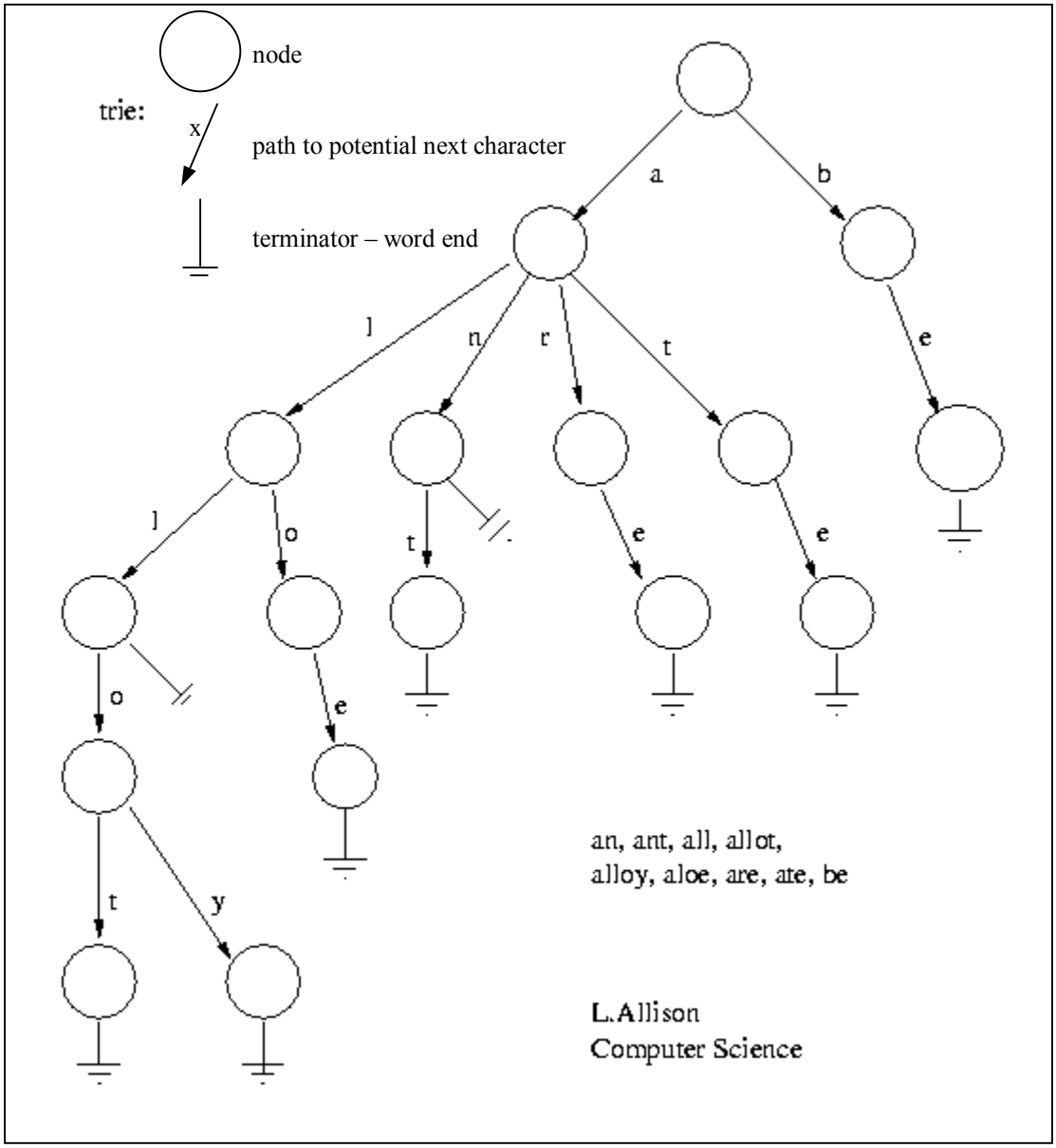


Figure 3-2 A *trie* example

two paths can be chosen: to *g* or to *e*. As the next character of the word under consideration (*y*) is not one of these characters, no path can be chosen. This implies that the word *alliy* is detected as an error and the user is notified.

3.4.3. Finding candidate corrections

When a word is detected as an error, the spell checking application tries to find suggestions for the word in the lexicon. Once a word has been flagged and the user has asked the system to generate suggestions, the lexicon trie is entered. All possible paths from the start node are tried until the first character of the flagged word is found. Then the same is done for the second character, and so on. As the word was misspelled and therefore is not present in the spell checker lexicon, there is a node from which the next character path cannot be found. Then paths to other nodes are followed from the current node and from previous nodes in order to find a string that is a candidate correction. When choosing a path to a character node that is not the next character in the original word, a penalty is given. This is done for each character node that is not equal to the corresponding character in the word. If the penalty passes a predefined border, then no further

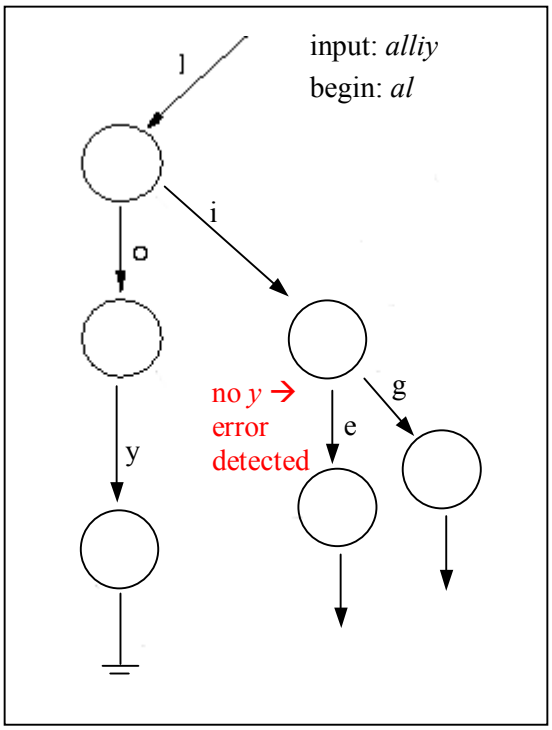


Figure 3-3 A detection example

daughters are checked and the process is continued from a higher node. All resulting strings that have a lower penalty than the predefined border value are suggested to the user.

The process of finding suggestions would not function properly without penalties, because if no penalty would be given when choosing a path to a different character node, candidate corrections would be found that are unlikely to be the intended word. For example, suppose that a user writes the string *aloy*. As the string is not present in the spell checker lexicon, it is detected as an error. The lexicon trie is then entered for finding suggestions. The part *alo* matches a path in the trie. From the node *o* several paths can be chosen, such as the path to *n* and the path to *u*. But there is no path to the next character *y*. In order to find a suggestion, the path to *n* is followed, from which

the path to *g* can be chosen. Then a terminator is found and the string *along* has been completed. If no penalty is given, *along* consequently would be a suggestion for *aloy*, although it is not very likely to be the intended word. If a penalty is given for each divergent character, the penalty for the string *along* would probably be too high and therefore it would not be a suggestion.
In fact, giving penalties for choosing another path in the trie corresponds to calculating the minimum edit distance; the Polderland spell checking applications use a minimum edit distance technique for finding suggestions. Figure 3-4 gives an example of finding suggestions. Here, the user wrote the string *alliy*. The part *all* matches a path in the trie. When arriving at the second *l*, the path to *i* is chosen (see figure 3-4), but there appears to be no daughter *y*. Then paths to other nodes are followed from the current node and from previous nodes. From the node *l* the link to *o* is chosen. Because a node is chosen that is not the next character in the word, a penalty is given. From *o* the correct character *y* can be chosen and therefore the penalty is not increased. From *y* a terminator is found and the string *alloy* has been formed. As the penalty is low, *alloy* will probably be a suggestion for the erroneous *alliy*. Similarly, the strings *alley*, *allay* and *ally* could be suggestions for *alliy*.

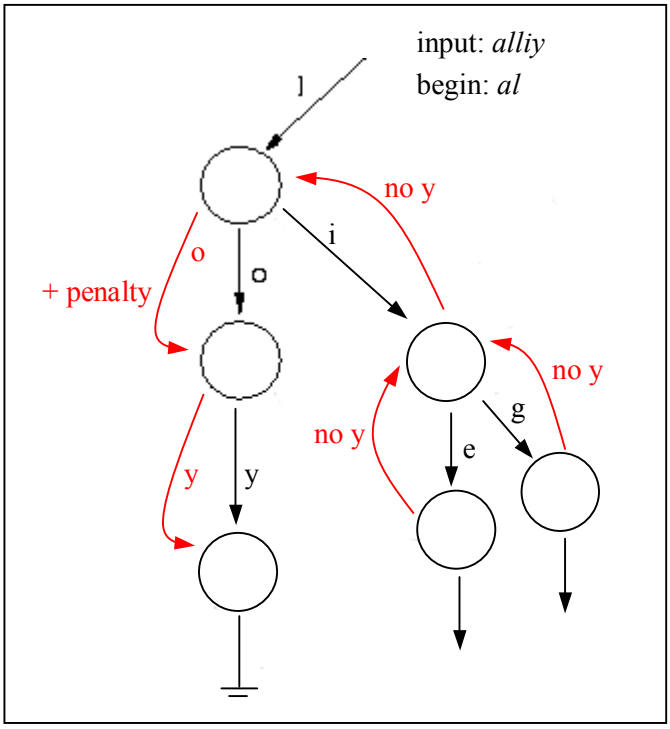


Figure 3-4 An example of finding suggestions

If all sorts of single character transformation (insertion, deletion and substitution) yield an equal penalty value, the penalty value corresponds exactly to the minimum edit distance. However, in Polderland spell checking applications, it is possible to vary penalty values with various character transformations. For example, it is possible to predefine that the penalty that is given when performing substitution (*alloy* → *alliy*) is higher than when performing deletion (*alloy* → *aloy*) of

a character. This information on penalty values is added to the lexicon. In the lexicon, following a sequence of nodes during the process of finding suggestions corresponds to a specific character transformation. For example, suppose the spell checker is searching suggestions for the string *alliy*. Then following the path from the second *l* via *o* to *y* yields a string (*alloy*) that can be transformed into *alliy* by substituting *o* by *i*. Thus, choosing this path corresponds with a substitution. Similarly, when searching suggestions for *aloy*, the path from *l* to another *l* corresponds with a deletion: the word that is found by following this path (*alloy*) can be transformed into *aloy* by deleting one character. Varying penalty values with various character transformations is in fact a variant on the minimum edit distance algorithm. The resulting penalty value is not equal to just the minimum number of editing operations but to a *weighted* minimum number of character transformations that is required for transforming one string into another.
Not only orthographic character transformations, but also phonetic resemblance plays a role in weighting minimum edit distances for Polderland spell checking applications: substituting a character or character sequence with a phonetically resembling character (sequence) yields a lower penalty than substituting it with a character (sequence) that has no phonetic resemblance. For example, when searching for suggestions for the string *bel*, *bell* would get a lower penalty than *belt*. In the Polderland spell checkers, phonetic information consists of sets of characters or character sequences that phonetically resemble each other. This information, which varies with the language of the spell checking application, is added to the spell checker lexicon when building the spell checker.
Other possibilities for varying penalty values, such as taking into account nearness on the keyboard (giving a lower penalty for character nodes that are more likely to be exchanged because the characters are close to each other on the keyboard) or knowledge on the probability of specific error types (e.g. first-character errors), are not used within the Polderland spell checkers.

3.4.4. Ranking candidate corrections

Having a list of alternatives for an incorrect string, the suggestions have to be ranked, in order to put the candidate correction that is most likely to be the intended word at the top of the list. As described in section 3.2.3, ranking could be based on minimum edit distance or word frequency. As explained in section 3.4.3, the minimum edit distance has been weighed based on specific character transformations and phonetic resemblance, resulting in the penalty value. Since penalty value aims to be a criterion for the probability of a suggestion being the correct suggestion, ranking is performed based on this penalty value. All suggestion strings have a specific penalty value that is lower than a predefined border value. However, not all suggestions necessarily have the same penalty value. The higher the penalty value of a suggestion, the lower this suggestion is ranked.

3.5. Problems for modern spell checking applications

In the previous sections it was described how spelling error detection and correction works in modern spell checking applications in general and in Polderland spell checkers specifically. However, this method appears to be insufficient for some classes of errors. In this section, three problems for modern spell checking applications are discussed: the word boundary problem, the problem of short word errors and the problem of real-word errors.

3.5.1. The word boundary problem

One of the classifications that were described in section 2.1 is the distinction between errors within one word and word boundary errors. There are two sorts of word boundary errors: incorrect splits (e.g. *together* → *to gether*) and run-ons (e.g. *a lot* → *alot*). Approximately 15% of all errors are word boundary errors (Kukich, 1992b; Mitton, 1987). What exactly is the problem of word boundary errors?
In nearly all spelling error detection and correction techniques, word boundaries are defined by white space characters. Incorrect splits and run-ons therefore yield a deviant number of words in the resulting sentence. This difference in number of words can give problems for error detection and/or correction. Run-ons are mostly a problem for correcting errors, whereas incorrect splits are a problem for both detecting and correcting.
Incorrectly putting two words together, like mistyping *of the* as *ofthe* or misspelling *kitchen door* as *kitchendoor*, often yields a string that is not a lexicon entry (Mitton, 1987). Therefore, the word is detected as an error. In order to correct this error the spell checking application should be able to add white spaces at any position within the incorrect string. If adding a white space yields two lexicon entries, a valid suggestion has been found. Unfortunately, adding white spaces at any position within the incorrect string results in many possible combinations of words that have to be checked against the spell checker lexicon. This decreases the speed of the application.
If a word has incorrectly been split up and results in two strings, detecting and correcting the error is more difficult. Incorrectly splitting a word often results in one or more strings that are lexicon entries themselves. Suppose an incorrect split results in two words of which one is a lexicon entry. Then this string is not detected, but the other string – which was not a lexicon entry – is. However, it is very difficult to correct this erroneous string, since neighbouring words are not taken into consideration for finding suggestions. For example, suppose *together* had been written as *to gether*. Since g*ether* is not a lexicon entry, the string is detected as an error. But when searching suggestions the preceding word *to* is normally not taken into account, as a result of which *together* probably is not found as a suggestion because its penalty is too high. A solution could perhaps be found in taking into consideration adjacent words when searching suggestions. Unfortunately, this can also yield incorrect suggestions. For example, suppose a user misspells *gather* as *gether* in the sentence *I try to gather all leaves*. Then *together* is a suggestion for *to gether*. More research needs to be done in order to find out how big this problem is. Moreover, taking into account adjacent words will decrease the speed of the application.

When an incorrect split yields two lexicon entries instead of one, the error cannot be detected by lexicon-based spell checking techniques at all.
Some applications that can correct word boundary errors have been studied. Kernighan (1991) designed a corrector for a text-to-speech application that handles run-ons by checking for deletions of white spaces at the same time it checks for other character deletions. Carter (1992) designed an application that was explicitly designed for correcting word boundary errors. This system was able to find the correct suggestion for 55% of 108 errors. Possibly, a large portion of word boundary errors involves a relatively small set of high-frequency function words (Kukich, 1992b). Pollock and Zamora (1984) used this assumption for implementing the SPEEDCOP corrector, which checked for run-on errors involving function words.
Despite these studies, word boundary errors are still a problem for modern commercial spell checking applications.

3.5.2. The problem of short word errors

Errors in short words were described in section 2.1. Short words were defined as words of two, three and four characters. Short word errors are a problem for automatic error detection and correction. This problem was noticed by Pollock and Zamora (1983). They found that misspellings in three and four character words constitute 9.2% of all misspellings but as much as 42% of the wrong corrections. This means that 42% of all erroneous strings that do not get the correct suggestion are short strings. Thus finding the correct suggestion for an erroneous string is more difficult for short words than for long words. Landauer and Streeter (1973) argued that this is because high-frequency words are more likely to yield another lexicon entry when applying a single character transformation than low-frequency words. Since short words occur more frequently than long words (Park and Gero, 1999), short words are more likely to yield another lexicon entry. As errors resulting in lexicon entries cannot be detected using a lexicon-based spell checking technique, they cannot be corrected either. This results in a larger error correction problem for short words.
Landauer and Streeter (1973) argued that short words are more likely to result in real-word errors when misspelled. Real-word errors themselves are also a problem for error detection and correction techniques. This problem is described in the next subsection.

3.5.3. The problem of real-word errors for spelling error detection and correction

Real-word errors are a problem for error detection and correction techniques. All modern commercial spell checking applications are lexicon-based: a lexicon is used to define which words are correct and which are not. Errors that yield a lexicon entry (i.e. a real word) therefore cannot be detected using a lexicon-based spell checking method. As real-word errors are defined as those errors that yield a lexicon entry, the amount of real-word errors varies with the size of the lexicon (Peterson, 1986). The larger the lexicon, the more real-word errors will occur.
As described in section 2.1, the frequency of real-word errors is between 25% and 40% of all errors, dependent on text type. This means that, if a spell checking tool can detect and correct

real-word errors, its performance will improve drastically. This becomes even more clear when the frequency of real-word errors is combined with the performance of the modern isolated-word error correction techniques. Kukich (1992b) found that 15% of all non-word errors are word-boundary errors and therefore cannot be corrected by isolated-word correction techniques. Thus, 85% of all non-word errors can be corrected by isolated-word correction techniques. Kukich also found that the best isolated-word correction techniques correct approximately 78% of this 85%. This means that 66% of all non-word errors are corrected. Now suppose a real-word error rate of 25% (Young et al, 1991). This means that 75% of all errors are non-word errors. 66% of this 75% is corrected by a relatively good isolated-word correction technique. This yields a total correction rate of 50%. Supposing a real-word error rate of 40%, (Mitton, 1987) only 66% of 60% of all errors is corrected, yielding a total correction rate of 40%.
Supposing a real-word error rate of 25% and a correction rate for real-word errors equal to that for non-word errors (78%), a method for detecting and correcting real-word errors could theoretically raise the performance of a system with 19.5% (78% of 25%). This would improve the system's correction rate from 50% to 69.5%.

3.6. Techniques for real-word error detection and correction

Context-sensitive error detection and correction aims at detecting and correcting real-word errors, which cannot be detected by isolated-word detection and correction techniques. As described in the previous section, modern lexicon-based spelling error detection and correction systems correct approximately 50% of all errors. As 25% to 40% of all errors are real-word errors, a method for detecting and correcting real-word errors would be useful. Several context-sensitive spell checking techniques have been suggested in the literature. There are two main types: methods based on probability information and methods based on semantic information. The study of Mays, Damerau and Mercer (1991) and the study of Golding and Schabes (1996) address methods based on trigram probabilities. The studies of St-Onge (1995) and Hirst and Budanitsky (2001) address methods based on semantic information. In this section, I describe and discuss these four studies.

3.6.1. Methods based on probability information

The study of Mays, Damerau and Mercer (1991) addresses the detection and correction of real-word errors using word trigrams. They use a word trigram statistical language model based on a 20,000-word lexicon. Unfortunately, their paper does not contain any information about the size of the corpus that they used for extracting trigram probabilities. As a test set, they use 100 correct sentences containing only lexicon words. Then they create a cohort for each sentence existing of similar sentences containing exactly one real-word error, generated by basic character transformations (insertion, deletion or substitution of one character or transposition of two characters). They compute the probability for each sentence in a cohort based on the probability of the trigrams of which the sentence consists. If the erroneous sentence does not have the highest

probability of its cohort then the error is detected. If the correct sentence has the highest probability of its cohort then the error is corrected. They obtain a detection score of 76% and a correction score of 74%.
I have two points of criticism on the method for obtaining these results. First, the relatively small difference between detection score and correction score has been created by the presence of the correct sentence in the cohort. Suggestions are not generated from the corpus but are already present as alternatives. This could never be done in a real application, because the correct sentence would have to be known in advance. Second, it is not clear what results could be achieved if the test set contained real errors instead of artificially generated erroneous sentences all containing only one error generated by basic character transformations.
Golding and Schabes (1996) created a method called Tribayes. For this method, they used the confusion sets from the list *Words commonly confused* in the back of the Random House Unabridged Dictionary (Flexner, 1983). Given a target occurrence of a word to correct, Tribayes substitutes in turn each word from the confusion set into the sentence. For each substitution, it calculates the probability of the resulting sentence based on part-of-speech trigrams. It selects as its suggestion the word that yields the sentence having the highest probability of all confusion sentences. This method has two disadvantages: the limited type of errors described by the confusion sets and the use of part-of-speech trigrams. The confusion list only contains sets of words that are commonly confused because of their similar meaning or form. Therefore, uncommon errors and typing errors are not considered. Furthermore, the list is finite: new confusable errors will never be corrected. The disadvantage of using part-of-speech trigrams is that only errors that yield syntactically incorrect sentences can be solved. A semantic error, like mistyping *minutes* as *minuets*, could never be corrected by using part-of-speech trigrams, since the part of speech is the same.

3.6.2. Methods based on semantic information

Real-word errors that result in a semantic anomaly without yielding a syntactically incorrect sentence are sometimes referred to as *malapropisms*. David St-Onge (1995) tries to solve the malapropism problem by using semantic word chains based on Wordnet. Every content word from the text is considered; function words are ignored because their semantic content is very low. The spell checker tries to add the word under consideration to a chain of semantically related words. If the word cannot be added, then it is regarded a malapropism. The program changes the word by inserting, deleting or moving one character and then tries to fit in the new word. The word that can be added to the chain is suggested to the user.
Hirst and Budanitsky (2001) also try to solve the problem of malapropisms using semantic chains. They work with the following algorithm. The spell checker marks a word as confirmed if (1) it occurs in the text more than once, (2) it occurs in the text as part of a known phrase or (3) within a window of *n* paragraphs there are one or more words with a sense related to the sense of the word under consideration. If an unconfirmed word *w* has a spelling variation (a valid word that is one or few basic transformations away from the original word) *w'* that would have been

confirmed if it had appeared in the text instead of *w*, the user is alerted to the possibility that *w'* is intended where *w* appears.
A major disadvantage of methods based on semantic information, like the methods used by St-Onge and Hirst and Budanitsky, is the fact that they ignore function words. Since function words are high-frequency short words, spelling and typing errors occur relatively often on these words (section 2.1).

3.7. Conclusion

In this chapter automatic spelling error detection and correction techniques have been described. After a definition and explanation of the term *automatic spelling error detection and correction*, the process of spelling error detection and correction and the performance of spell checking techniques have been described. All modern spell checking applications are lexicon-based and use a minimum edit distance algorithm. The performance of the spell checking application varies with lexicon size and the characteristics of the test set. Kukich (1992b) found that approximately 78% of non-word errors could be corrected by isolated-word correction techniques in general. The performance of isolated-word correction techniques shows that there is a need for real-word error detection and correction techniques.
To be able to build an application for context-sensitive spell checking, it is useful to know exactly how modern spell checking techniques function. Therefore, the spell checking algorithm of the Polderland spell checkers, which functions as an example for modern commercial spell checking applications in general, has been described. The Polderland spell checkers use a trie-structure lexicon that applies a technique based on a weighted minimum edit distance for finding suggestions. The fact that modern spelling error detection and correction techniques use a lexicon for detecting errors makes them incapable of detecting and correcting real-word errors. This problem has been discussed together with two other problems for automatic spelling error detection and correction techniques: the word boundary problem and the problem of short word errors. The problem of real-word errors is the main problem for the current research. As 25% to 40% of all errors are real-word errors, a good method for detecting and correcting real-word errors would improve the performance of modern spell checkers drastically. Two methods for solving the problem of real-word errors have been described: methods based on probability information and methods based on semantic information. Altogether, four methods for context-sensitive spell checking have been discussed. I had some criticisms on three of these methods: they exclude specific error types or specific word classes. The methods of St-Onge (1995) and Hirst and Budanitsky (2001) excluded errors in function words; Golding and Schabes (1996) excluded semantic errors. Moreover, Golding and Schabes used a precompiled, finite set of common spelling errors, resulting in a method that cannot detect and correct less common errors and (incidentally occurring) typing errors. Consequently, these three methods are not valid for solving the problem of real-word errors entirely. In my opinion, a method for context-sensitive spell checking should aim at detecting and correcting all types of real-word error regardless of the

word class of the (resulting) word. In chapter 4, my solution to the problem of real-word errors is described.

4. Building a context-sensitive spell checker

4.1. Introduction

In section 3.5.3, the problem of real-word errors for automatic spelling error detection and correction was described: isolated-word (i.e. lexicon-based) detection and correction techniques cannot detect and correct real-word errors. Spelling error detection and correction methods that aim at detecting and correcting real-word errors are referred to as *context-sensitive error detection and correction methods*. In section 3.6, four methods for context-sensitive spell checking were discussed. I criticized three of these methods: they exclude specific error types or specific word classes. The methods of St-Onge (1995) and Hirst and Budanitsky (2001) excluded errors in function words; Golding and Schabes (1996) excluded semantic errors. Moreover, Golding and Schabes used a precompiled, finite set of common spelling errors, resulting in a method that cannot detect and correct less common errors and (incidentally occurring) typing errors. Consequently, these three methods cannot solve the problem of real-word errors entirely.

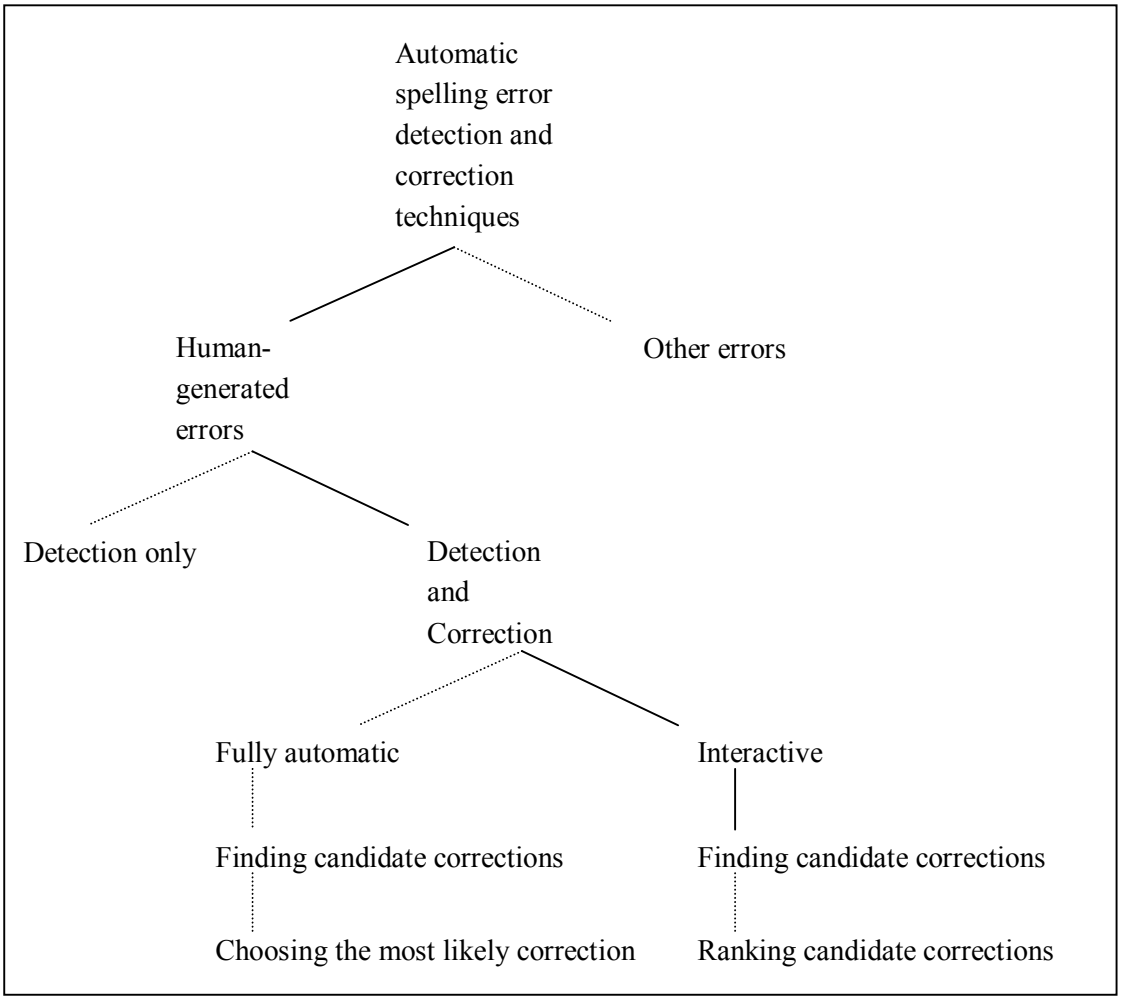


Figure 4-1 The class of the intended context-sensitive spell checking application

In my opinion, a method for context-sensitive spell checking should aim at detecting and correcting all types of real-word error regardless of the word class of the (resulting) word. In section 2.2, I defined which error types are considered as real-word errors in the current research. A context-sensitive spell checking method should aim at detecting and correcting all of these real-word error types. In the current research, I aim at creating an interactive system for detecting and correcting human-generated real-word errors. In order to restrict the building process, the correction part of the algorithm is restricted to finding candidate corrections: no ranking is performed. An error is considered corrected if the correct suggestion is in the list of suggestions. The continued lines in figure 4-1 on page 32 illustrate where the intended context-sensitive spell checking application stands in the classification of automatic spelling error detection and correction techniques that was defined in section 3.2. The spell checking application should be able to detect and correct all classes of real-word errors in all classes of words. In this chapter, the spell checking method and algorithm that were developed for the current research are described.
Starting point for building the context-sensitive spell checker is a lexicon-based spell checker. One of the main questions of the current research is "To what extent do the results of a lexicon-based spell checker improve when the lexicon-based spell checker is combined with the context-sensitive spell checker?" (See also chapter 1). Therefore, it is important to start from a well-performing lexicon-based spell checker. The lexicon-based spell checker that is used in the current research is described in section 4.2. After that, the context-sensitive spell checking method is described in sections 4.3 and 4.4. In section 4.5, a comparison is made to the other context-sensitive spell checking methods which were described in section 3.6.

4.2. Building a lexicon-based spell checker

As mentioned above, a lexicon-based spell checker is used as the starting point for building the context-sensitive spell checker. But the lexicon-based spell checker also has a function within the context-sensitive spell checking algorithm. This function is explained in section 4.3. In this section, the lexicon-based spell checker that is the basis for the context-sensitive spell checker is described. It is important to have a well-performing lexicon-based spell checker, in order to prevent the performance of the context-sensitive spell checker to be negatively influenced by the performance of the lexicon-based spell checker. As described in section 3.4, the lexicon is the most important part of a lexicon-based spell checker. For a spell checking purpose, a lexicon has to have the following characteristics. First, it has to be large. In general, a lexicon has to contain at least several ten thousands of words to be suitable for spell checking purposes (naturally, lexicon size is dependent of language). Second, the lexicon has to contain words from the language (variety) for which the spell checker is built. Third, it is useful when a lexicon contains part-of-speech information of all words. This information can be used to add some linguistic information to the spell checker. For the current research, I chose to use the British English Source Lexicon (BESL) from Oxford University Press (OUP) because it is large, contains British English words and part-of-speech information. Moreover, it was available at that moment. Starting from BESL, a spell checker has been built according to the Polderland method for

building lexicon-based spell checkers. Since all Polderland spell checkers are suitable for spell checking in Microsoft Office, the BESL spell checker can be used in Microsoft Office too. In this section, it is described how the BESL spell checker has been built, starting from the BESL lexicon. In the first subsection (4.2.1), the format and content of BESL are described. Section 4.2.2 describes how a spell checker lexicon has been created from BESL and in section 4.2.3, the features of the resulting spell checker are discussed.

4.2.1. The British English Source Lexicon

The British English Source Lexicon (BESL) has been created by Oxford University Press (OUP). BESL is a collection of seven British English lexicons published by OUP, which are:

- Shorter Oxford English Dictionary (SOED),
- Concise Oxford Dictionary (COD),
- Pocket Oxford Dictionary (POD),
- New Oxford Dictionary of English (NODE),
- Concise Pronunciations Dictionary (CPD),
- Oxford Spelling Dictionary (OSD),
- Inhouse Resources (INC).

The number of entries in BESL is 156,932. Within each entry in BESL it is indicated from which source(s) the word is taken. Many entries contain a number of subentries, which are declensions, conjugations or derivatives of the main entry. When all subentries are included, the total number of entries in BESL is 421,745.
Figure 4-2 on page 35 contains a sample of the format of BESL. The example contains four entries. Between the tags <hw> and </hw> is the entry's headword. Between every pair of <sube> and </sube> is a subentry; <pr> and <ph> mean pronunciation and phonetics respectively. <src> indicates the source(s) from which this entry is taken.

4.2.2. Creating a spell checker lexicon from BESL

A spell checker lexicon that is suitable for attaching to a Polderland spell checker (which is in turn suitable to be used within Microsoft Office) needs to have a fixed format. Starting from BESL, a word list can be extracted which has the right format to generate a Polderland spell checker lexicon from it. The structure of the Polderland spell checker lexicons was described in section 3.4.1. To extract an appropriate word list from BESL, a few steps have to be taken.
The first step is extracting from BESL a word list that only contains word forms and their parts of speech. Therefore, all entries and subentries and their parts of speech have to be extracted from BESL. In appendix IV-1, a Perl program is presented for extracting this information from BESL. The resulting word list has the format <*word*><*tab*><*category*>.

The resulting word list contains a few difficulties for converting it into a Polderland spell checker lexicon. Therefore, the second step is editing it. This editing exists of removing multi-word entries, replacing hyphens by double hyphens and filling gaps in the part-of-speech field. Multi-word entries like *age range* and *city center* have to be removed from the word list because a text is checked on word level and not beyond word borders. Hyphens within words (*A-road* and *spin-off*) have to be replaced because in a Polderland spell checker lexicon a hyphen means a possible word break at the end of the line. Double hyphens (*A--road*, *spin--off*) in the lexicon are interpreted as real hyphens by the spell checker. Gaps in the part-of-speech field have to be filled with a dummy tag W (for *word*). The Perl program that was used to execute these three editing steps is shown in appendix IV-2.

```
<se ty=s id=53893><hw>handle</hw><ps>n.<hm>2</hm> &amp. m4.1</ps><sube>
<form>handle</form><sps>n.</sps><pr or=Br><ph>"handl</ph></pr></sube><sube>
<form>handles</form><sps>pl.</sps><pr or=Br><ph>"handlz</ph></pr></sube>
<srcs><src>SOED</src><src>CPD</src><src>COD</src><src>POD</src></srcs></se>

<se ty=s id=53895><hw>handle</hw><ps>v.<hm>2</hm><gr>t.</gr> &amp.
m4</ps><sube><form>handle</form><sps>v.</sps><pr or=Br>
<ph>"handl</ph></pr></sube><sube><form>handles</form><sps>present</sps><pr or=Br>
<ph>"handlz</ph></pr></sube><sube><form>handled</form><sps>past</sps><pr or=Br>
<ph>"handld</ph></pr></sube><sube><form>handling</form><sps>gerund</sps><pr or=Br>
<ph>"handl=IN</ph><vph>"handlIN</vph></pr></sube><srcs><src>SOED</src>
<src>CPD</src><src>COD</src><src>POD</src></srcs></se>

<se ty=s id=53896><hw>handleability</hw><ps>n. &amp. m9</ps><sube>
<form>handleability</form><sps>n.</sps><pr
or=Br><ph>%handl@"bIl1ti</ph><vph>%handl=@"bIl1ti</vph></pr></sube>
<srcs><src>CPD</src></srcs></se>

<se ty=s id=53897><hw>handleable</hw><ps>a. &amp. m9</ps><sube>
<form>handleable</form><sps>a.</sps><pr
or=Br><ph>"handl@bl</ph><vph>"handl=@bl</vph></pr></sube>
<srcs><src>CPD</src></srcs></se>
```

Figure 4-2 Format of BESL

After editing, the field <word> in the word list contains all entries and subentries from BESL and the field <category> contains the corresponding part-of-speech tags from BESL. As a Polderland spell checker cannot interpret these part-of-speech tags, the BESL tags have to be replaced by Polderland tags. For Polderland spell checker lexicons, the field <category> contains two items: a part-of-speech tag and a position tag. The possible part-of-speech tags are V (verb), N (noun), J (adjective) and U (number). All remaining words have W (word). The position tag indicates whether a word can be an independent word (I), the left part of a compound (L) and/or the right part of a compound (R). English does not have the possibility of making one-word compounds: semantic compounds are written as two words. Therefore, all words in the word list get the position tag I. Thus, the third step that has to be taken in order to create an appropriate word list is replacing the BESL part-of-speech tags (adv, pron, prep, conj, int, a, n and v) by the Polderland

part-of-speech tags and position tags. If a word has more than one part-of-speech tag, this results in more than one entry in the spell checker lexicon.
The final step is replacing ASCII codes (e.g. *&agra.*, *&eacu.* and *&iuml.*) within the word list by diacritics (e.g. *à*, *é* and *ï*).
Figure 4-3 contains a sample of the format of the resulting word list.

```
handle      NI
handle      VI
handleability     NI
handleable  JI
handlebar   NI
handlebars  NI
handled     JI
handled     VI
handleless  JI
handler     NI
handlers    NI
handles     NI
handles     VI
handless    JI
handline    VI
handlined   VI
handlines   VI
handling    VI
```

Figure 4-3 Format of the word list extracted from BESL

The word list now has a suitable format to generate a Microsoft spell checker lexicon from it. For this purpose, Polderland has an application called mklex. Input for mklex is a text-readable file and output is a trie-structure lexicon with extension .lex. This output file, which is the BESL lexicon in the format of a Polderland spell checker (i.e. a trie structure), can be attached to Microsoft Office.

4.2.3. Features of the BESL spell checker

The BESL spell checker was built according to the Polderland method for building lexicon-based spell checkers. Therefore, the BESL spell checker is in the same spell checking application class as the Polderland spell checkers, which were described in section 3.4. The structure of the Polderland spell checkers and how they function were also described in section 3.4. In that section, the three basic steps of error detection and correction were denoted: detecting errors, finding candidate corrections and ranking candidate corrections. For the BESL spell checker these three steps function as described in section 3.4. Some choices still have to be made regarding penalty definitions. It is possible to predefine various penalty values for various kinds of paths that are chosen when generating a suggestion. In the BESL spell checker, variant penalty

values for different character transformations are taken into account as well as phonetic information. The penalty configurations for different character transformations and the phonetic information are both defined when creating the spell checker lexicon. It is possible to run mklex on a word list using an extra parameter, which is the name of a file containing these penalty configurations and the phonetic information. This information is then included in the resulting spell checker lexicon. Appendix I shows part of the content of phonrulesEN.txt, which is the configurations file that was used for the BESL spell checker. In the first part of the file, Match Penalties, the penalty values for various character transformations are defined. These are relative values. They mean for example that doubling (*doub*) a character (*chosen* → *choosen*) yields a penalty that is twice as high as substituting (*sub*) a character (*alloy* → *alliy*). The rest of the file, Match Rules, consists of phonetic rules for English. For several sets of character sequences, the phonetic resemblance is defined. These definitions are used for giving a relatively low penalty when choosing a character sequence that phonetically resembles the original character sequence. For example, rule 27 indicates that the character sequences *Mc* and *Mac* are phonetically exactly equal; hence the declaration *twin* in the middle column. The third column gives an example of a context in which the character sequences are phonetically equal. Rule 28 indicates that the sequences *sch* and *sk* are phonetically similar but not exactly equal; hence the declaration *kin* in the middle column. In the first part of the phonrules-file, Match Penalties, it has been declared that exchanging a character sequence by its twin, kin or cousin yields a relatively low penalty. In case of *sch,* this means that replacing the sequence by *sk* yields a penalty value 1, whereas replacing it by *sl* yields a penalty value 2 (i.e. one substitution and one deletion).
In the current research, the penalty configurations of the lexicon-based spell checker should be chosen in a way that optimises the performance of the context-sensitive spell checker[6]. Context, a program developed by Polderland, enables the user to test the consequences of changing the spell checker's penalty configurations. When a spell checker lexicon is attached to Context, a number of candidate corrections (suggestions) for an input word in Context are given. It is also possible to give a lexicon entry as an input word; then the words are found that the most resemble the input word. The program Context is also useful for testing the consequences of changing the penalty configurations for the performance of the spell checker in general.
By changing the lexicon configurations, a spell checker lexicon can be created that optimises the performance of the context-sensitive spell checker. The results of optimising these configurations are described in section 5.3.

4.3. The context-sensitive spell checking method and algorithm

The BESL spell checker, which was described in the previous section, is an important component of the context-sensitive spell checker. In this section the context-sensitive spell checking method is described. In section 4.3.1, the choice for the spell checking method that is used in the current

[6] How the features of the lexicon-based spell checker can influence the performance of the context-sensitive spell checker is described in section 4.3.

research is motivated. In section 4.3.2, the resulting context-sensitive spell checking algorithm is described.

4.3.1. Motivation for the context-sensitive spell checking method

As mentioned in the introduction to the current chapter, a method for context-sensitive spell checking should aim at detecting and correcting all types of real-word error, regardless of the word class of the (resulting) words. Therefore, I aim at creating a context-sensitive spell checking method that can detect and correct typing errors as well as spelling errors and syntactic errors as well as semantic errors. This demand has three consequences. First, detecting and correcting all classes of errors in all classes of words implies detecting and correcting all errors that could possibly be made. The spell checking algorithm should therefore not only be designed for detecting and correcting a predefined, finite set of errors, such as a set of commonly made errors, but it should also be able to detect and correct errors that are less common. Second, a context-sensitive spell checking method that is able to detect and correct semantic errors as well syntactic errors should not be based exclusively on syntactic features of words. Suppose we have a spell checking algorithm that uses part-of-speech information for determining whether a specific sentence is correct or not. Such an algorithm could use a database containing relative frequencies of part-of-speech trigrams. For example, in English, the part-of-speech trigram *ART N V* probably has a higher relative frequency than *ART PREP V*. In a context-sensitive method based on syntactic features, this probability information is used to determine whether a specific sequence of parts of speech is correct. For example, the trigram *the from is* could be detected since *ART PREP V* probably is a low-frequent part-of-speech trigram. Changing one character yields *ART N V* (*the form is*), which probably has a higher relative frequency. Such an algorithm would not be able to detect semantic errors, because in case of a semantic error the erroneous word has the same part of speech as the intended word. For example, suppose someone mistypes *minutes* as *minuets*. Since both words have the same part of speech (i.e. *N plural*), the erroneous and intended part-of-speech trigram are exactly the same. Therefore, a method based exclusively on syntactic features, is not able to detect and correct semantic errors.
Third, a spell checking method that is able to detect and correct errors in all classes of words should not exclude function words. This implies that the method should not be based on semantic features of words. Since function words have very low semantic content, a semantic method would not be able to detect and correct errors in function words. In section 3.6, a distinction was made between methods based on probabilistic information and methods based on semantic information. The third demand implies that a method for detecting and correcting all types of real-word error in all classes of words should not be based on semantic information, thus it should be based on probabilistic information. The second demand implies that syntactic features of words should not play an exclusive role in determining whether a specific string is correct or not. These two assumptions (the method should be based on probabilistic information and not exclusively on syntactic information) result in a context-sensitive spell checking method based on word (form) probability information. I chose to create a detection and correction algorithm that

uses information on probabilities of word trigrams[7]. This means that sequences of three words are considered instead of words in isolation. To check whether a specific trigram in the text contains a real-word error, the probability of that trigram is determined. If its probability is very low, then the trigram is considered erroneous. For example, suppose someone misspells *of* as *off* in the trigram *in case of*. Then the resulting trigram *in case off* will have a lower frequency of occurrence than the intended trigram *in case of*. The information on trigram probabilities is extracted from a large corpus. This is described in section 4.4. First, the context-sensitive spell checking algorithm is described in detail in the next subsection.

4.3.2. The context-sensitive spell checking algorithm

As described above, the context-sensitive spell checking algorithm of the current research uses probability information to determine whether a specific word trigram contains a real-word error. The context-sensitive spell checking algorithm performs detection and correction. Both steps are described separately in this section.

The detection algorithm performs three main steps. First, the text that has to be spell checked is split up in trigrams. At every word a new trigram starts, resulting in a number of trigrams equal to the number of words in the text minus two. For example, the five-word sentence *Please fill in the form* is split up in the three trigrams *please fill in*, *fill in the* and *in the form*.

Second, for each trigram it is checked whether all three words are in the BESL spell checker lexicon. This check would not have to be executed when the lexicon-based spell checker and the context-sensitive spell checker would have been combined into one spell checking application. In that case, the lexicon-based spell checker would perform non-word error detection and correction before the context-sensitive spell checker would perform real-word error detection. Then the input of the context-sensitive spell checker would not contain non-word errors and this second step would not have to be executed. However, in the current research, a stand-alone context-sensitive spell checker is built in order to be able to test it separately from the lexicon-based spell checker. Thus, the lexicon check is performed: if one or more words from the trigram are not in the spell checker lexicon, the trigram contains a non-word error and is it not considered further, because non-word errors are not in the scope of context-sensitive spell checking. This means that in the current research, there are still non-word errors in the text after the context-sensitive spell checker checked it for real-word errors. Looking up each word of every trigram in the lexicon implicates that most words from the text are checked three times (once for every trigram it is part of). This way, the program does not have to 'remember' which word is correct and which one is not. This is done to save memory space. In the chosen algorithm, the 'memory' of the system is restricted to the trigram under consideration.

Third, every trigram is looked up in a precompiled database containing a list of trigrams and their number of occurrence in the corpus used for compiling the database. If the trigram is in the trigram database, the trigram is regarded correct and it is not considered further. If the trigram is

[7] A trigram is a sequence of three units. These units can be characters, phonemes, words or parts of speech. In the current research, the term *trigram* refers to a *word trigram*.

not in the trigram database, then the trigram is considered too unlikely and therefore detected as an erroneous trigram containing a real-word error.
The correction algorithm performs an additional three steps. When a trigram has been detected, one or more of the three words is considered erroneous, but which of the three is not known. Therefore, candidate corrections for all three words are sought. The BESL spell checker lexicon is used to find candidate corrections for all three words of the trigram. This is the first step of the correction algorithm. When all possible candidate corrections for all three words have been found, these are all put together resulting in candidate corrections for the trigram as a whole. The third step is looking up each of these candidate correction trigrams in the trigram database. The trigrams that are in the database are considered more likely to be intended by the user than the detected trigram and are therefore suggested to the user.
See figure 4-4 on page 41 for a graphical representation of this context-sensitive spell checking algorithm with the example sentence *Would you please fill in the from?*, in which *from* is erroneous because *form* was intended. For implementing this spell checking algorithm, a Perl program was written. This Perl program is described in more detail in section 5.3.

4.4. Building the trigram database

As a source for compiling a trigram database, I chose to use the written part of the British National Corpus (BNC), which contains 90 million words of written British English text. The trigrams from the BNC and their numbers of occurrence are stored in a large database.

4.4.1. The British National Corpus

For building a trigram database that contains a large amount of English trigrams, a corpus of written text is needed. As the lexicon-based spell checker uses a lexicon of British English, the corpus for extracting trigrams should also be a corpus of British English. The British National Corpus (BNC) has been chosen for this purpose.
The British National Corpus is a 100 million-word corpus of British text. Ninety percent (90 million words) of the corpus consists of written texts and ten percent (10 million words) consists of spoken texts. The BNC is a sample corpus, which means that it is composed of text samples instead of whole texts. These text samples are generally no longer than 45,000 words. The corpus is not restricted to any particular subject field, register or genre, but it is restricted to one language: British English. According to the designers of the BNC, the size of the corpus as a whole, as well as the size of respectively the written and the spoken parts would be large enough to yield valuable empirical statistical data.

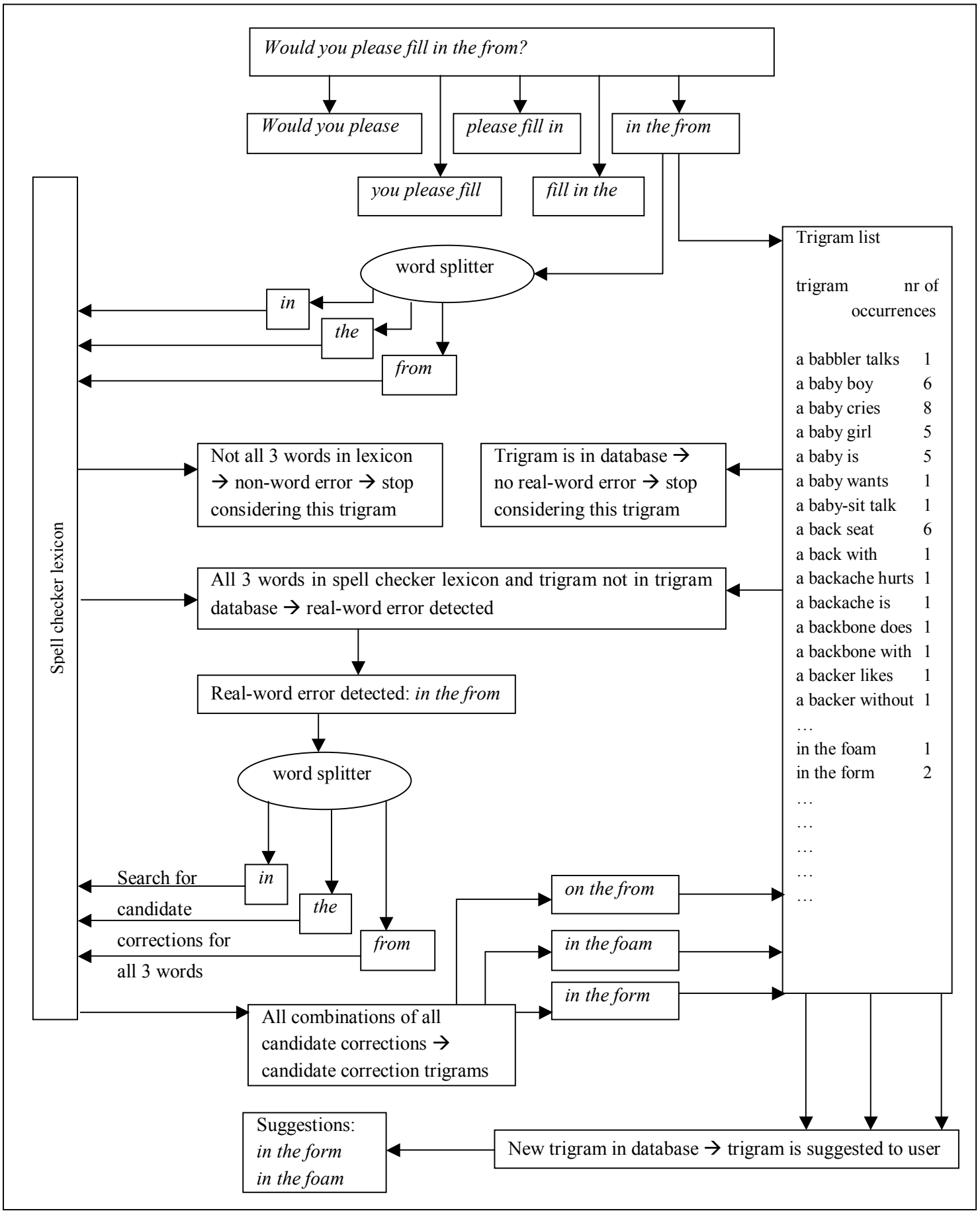


Figure 4-4 A graphical representation of the context-sensitive spell checking algorithm

4.4.2. The algorithm for extracting trigrams

Extracting trigrams from the written part of the BNC is done by means of a Perl script. In the algorithm for extracting trigrams, a word is defined as a character sequence bordered by white spaces. Semantic compounds (e.g. *kitchen door*) are therefore considered as two words. The algorithm removes all punctuation and capitals, as a result of which the trigrams *he asked, 'why'* and *He asked why* are the same (i.e. *he asked why*). As the algorithm removes punctuation, sentence borders are removed too. Moreover, text borders are also removed, in order to facilitate the extracting process. As a result, some trigrams consist of one or two words from one text and one or two words from another text, but this is a relatively small number of trigrams. The written part of the BNC consists of approximately 90 million words, which corresponds with 90 million minus two trigrams. These words are in 4424 texts. This means that there are 4423 text borders. For each text border, there are two trigrams that consist of words from two texts (one has two words from the first text and another has two words from the second text). This means that in the trigram database, there are 8846 trigrams that cross a text border. On 90 million trigrams, this is one out of every ten thousand. In fact, the number of trigrams that cross a sentence border is not negligible. Supposing an average of twenty words per sentence, the written part of the BNC contains 4,5 million sentences, which corresponds to 9 million trigrams that cross a sentence border. This is one in every ten trigrams. Fortunately, this is unproblematic, since in the text that is spell checked sentence borders are also removed.

Each trigram from the BNC is stored in the trigram database. If the trigram is already present in the database, its number of occurrences is increased with one. There appear to be 6,132,751 different trigrams in the written part of the BNC.

Obviously, not all trigrams are unique. 1,687 of the trigrams occur more than one hundred times and nineteen of them occur more than one thousand times. The ten most frequently occurring trigams from the BNC and their numbers of occurrence are given in table 4-1.

Table 4-1 The ten most frequently occurring trigrams

Trigram	number of occurrence
one of the	3467
the end of	2173
out of the	1729
as well as	1680
part of the	1395
it was a	1372
end of the	1370
there is a	1347
per cent of	1305
some of the	1293

4.5. Comparison to other context-sensitive spell checking methods

As described in section 4.3.1, the method for context-sensitive spell checking that I chose for the current research aims at detecting and correcting all types of error regardless of the word class of the (resulting) word. Three of the four methods discussed in section 3.6 are not able to detect and correct all classes of real-word errors in all classes of words. Only Mays, Damerau and Mercer (1991) included all types of errors in all classes of words. They created a method based on word trigram probabilities. Therefore, my method resembles theirs most. However, there are some important differences between my method and their method.
First, a method based on word trigram probabilities requires a database of trigrams and their relative frequency. The source that Mays et al. used for compiling such a database is not clear. For compiling my trigram database, I used the British National Corpus (BNC). Second, the algorithm of Mays, Damerau and Mercer works at sentence level: the likelihood of a sentence is computed from the probabilities of all trigrams in the sentence. In my algorithm, every trigram is considered separately: the probabilities of sentences are not calculated. Third and most important: the algorithm of Mays, Damerau and Mercer uses sentence cohorts from which the sentence that has the highest likelihood is chosen. A cohort contains the correct sentence and a number of constructed erroneous deviations from the correct sentence. Thus, the correct sentence is always present in the cohort and therefore it is always chosen when its probability is higher than the other sentences in the cohort. In my case, the correct trigram is not necessarily present in the trigram database, thus the chance that it is chosen as a suggestion is much smaller. On the other hand, my method can be used on real (i.e. unknown) data whereas the method of Mays, Damerau and Mercer only functions in a test environment where the correct sentence is known in advance. Obviously, this is never the case in a real application.

4.6. Conclusion

In section 3.6, four methods for context-sensitive spell checking were discussed. Three of these methods are not very suitable in my opinion, because they are not able to detect and correct all types of real-word errors in all classes of words. In the current research, I aim at creating a context-sensitive spell checking method that can detect and correct semantic real-word errors as well as syntactic real-word errors, those resulting from typos and those resulting from spelling errors.
A method that is able to detect and correct all types of real word errors in all classes of words has to satisfy three demands. First, the method should not be able to detect and correct a finite set of errors only; second, the method should not be based exclusively on syntactic features of words; third, the method should not be based on semantic information. These three demands lead to a method based on word trigram probability information. Starting point for this context-sensitive spell checking method is a lexicon-based spell checker. By changing the configurations of the lexicon-based spell checker, a lexicon can be created that optimises the performance of the context-sensitive spell checker. The results of optimising these configurations are described in

section 5.3. Another important element of the context-sensitive spell checking application is a database containing trigrams and their relative frequencies. This database has been extracted from the written part (90 million words) of the BNC.

Compared to other context-sensitive spell checking methods, I concluded that the only method besides mine that is able to detect and correct all classes of real-word errors in all classes of words is the method of Mays, Damerau and Mercer (1991). However, there are some important differences between their method and mine. The most important difference is that my method can be used on real (i.e. unknown) data whereas the method of Mays, Damerau and Mercer only functions in a test environment where the correct sentence is known in advance. Obviously, this is never the case in a real application. Therefore, the method of Mays, Damerau and Mercer is less valid for detecting and correcting errors in real human-written texts than mine.

In chapter 5, the performance of my method for context-sensitive spell checking is evaluated.

5. The performance of the spell checking application

In this chapter the performance of the spell checking application, which was described in chapter 4, is evaluated. To be able to obtain valid results and to interpret them properly, it is important to know how to evaluate a spelling error detection and correction system in general, i.e. how to measure the performance of a spell checking application. Therefore, some general methods for evaluation are described in section 5.1. After that, the results of the current research are described and discussed. The results can be divided into two parts: the performance of the lexicon-based BESL spell checker (section 5.2) and the performance of the context-sensitive spell checker (section 5.3).

5.1. Evaluating a spelling error detection and correction system

In the literature, several methods for evaluating a spelling error detection and correction system have been proposed. The most frequently used methods for measuring this performance are described in section 5.1.1. After that, the method used for the evaluation of Polderland spell checkers is described in section 5.1.2. This method is a good example of how the evaluation of spell checkers is carried out in practice and I will use (parts of) this evaluation for the evaluation of the BESL spell checker and the context sensitive spell checker.

5.1.1. Performance measures

Two frequently used measures for evaluating a spell checking application are precision and recall. In general, precision denotes a system's accuracy and recall indicates a system's coverage. However, the definitions of both measures vary between studies. To illustrate this, I first describe the definitions used by TEMAA, a Danish project for natural language processing evaluation, before I describe my own definitions.

The definitions for recall and precision used by TEMAA
TEMAA defines recall as *the degree to which the checker accepts all the valid words of a language* and precision as *the degree to which the checker rejects all the invalid words*. These definitions for precision and recall are used in other studies too. TEMAA also uses a third measure, suggestion adequacy, which is defined as *in case of invalid words, does the checker provide correct suggestions*.
The definitions used by TEMAA are not used in many studies. Most studies use definitions for precision and recall that are similar to the definitions that are used in the current research. These definitions are described below.

My definitions for recall and precision
First, I make a distinction between *detection recall* and *correction recall*. As described in section 3.1, detection is defined as an incorrect string being flagged by the system. For an interactive system, correction is defined as the intended word being at the top of the list of candidate corrections, or – when no ranking is performed – the intended word being present in the list of candidate corrections. Following these definitions, I define detection recall, correction recall and precision as follows:

- Detection recall: the number of correctly detected strings divided by the total number of erroneous strings in the text.
- Correction recall: the number of corrected erroneous strings divided by the total numbers of erroneous strings in the text.
- Precision: the number of correctly detected strings divided by the total number of detected strings.

For computing detection recall as well as precision the number of correctly detected strings is required. A string is correctly detected if it has been detected and it is indeed erroneous. An incorrectly detected string is referred to as a *false hit*.
The definitions used by TEMAA are opposite to mine: in TEMAA's definition, recall denotes the coverage of correctly spelled words, whereas in my definition, detection recall denotes the coverage of errors. In TEMAA's definition set, precision has this latter denotation.
I prefer my set of definitions over TEMAA's because TEMAA's definition set needs a third measure (suggestion adequacy), whereas my definition set has been expanded by dividing recall into detection recall and correction recall, which is less complicated and more intuitive. Moreover, my definitions of precision and recall correspond to the definitions that are normally used in the evaluation of information retrieval systems. That makes my definitions less confusing than those used by TEMAA.
In practice, precision and detection recall are oppositely related to each other: a system that has a high precision has a relatively low recall and vice versa.
Detection recall, correction recall and precision are used in the current research for measuring the performance of both the lexicon-based BESL spell checker and the context-sensitive spell checker. How this is done, is described in the sections 5.2 and 5.3. First, the Polderland method for measuring the performance of a lexicon-based spell checker is described.

5.1.2. Polderland's methods for measuring performance

Polderland has developed two methods for testing the performance of a lexicon-based spell checker. The tests have been built in Microsoft Visual Basic and can be executed with a macro in Microsoft Word. Therefore, only lexicon-based spell checkers that are accessible in Microsoft Office can be tested by these methods. As described in section 4.2, all Polderland spell checkers are suitable for spell checking in Microsoft Office.

Both testing methods, the coverage test and the suggestion test, calculate the performance of a spell checker and give the possibility to compare the results to the performance of other lexicon-based spell checkers for the same language. The two tests are described below.

Coverage test
The coverage test aims to measure the spell checker's coverage of correctly spelled words. In order to perform a valid coverage test, a large amount (more than 100.000 words) of text is required. The coverage test is executed for two or more spell checkers for the same language consecutively. For each spell checker the same (Microsoft Word) documents are used. The test counts the total number of words in the text and the number of words that are flagged by the spell checker. Coverage is determined from the number of flagged words divided by the total number of words. The spell checker that flags the highest number of words has the lowest coverage. Therefore, it is important that the test texts contain no or very few errors. The larger the amount of test texts is, the more reliable the coverage test will be.

Suggestion test
The suggestion test can be used for measuring detection recall, correction recall and precision. As described in section 5.1.1, the number of correctly detected errors is required for computing detection recall and precision. For computing correction recall, the number of erroneous words that got the correct suggestion is required. For that reason, a text containing errors is not sufficient for performing the suggestion test. The erroneous words and their corrections have to be known in order to be able to compute recall and precision. Therefore, a test document for the suggestion test consists of pairs of erroneous words and their corresponding correct words. The test can be executed for both spell checkers at the same time: all word pairs occur twice in the test document and for each half of the document another spell checker can be active. Since the spell checkers check the same pairs of words, the results can legitimately be compared.
The following counts are performed. The text between brackets refers to the labels shown in appendix II-1, which is the user interface of the suggestion test.

- The number of words that have been detected by one spell checker (*A errors*);
- The number of words that have only been detected by one spell checker (*just A*);
- The number of words that received a correct suggestion (*correct suggestions*);
- The number of words that have incorrectly been detected (*falsely false*[8]);
- The number of words that have been detected by both spell checkers (*spelling errors both for A and B*).

The suggestion test does not count the total number of errors in the test document, thus this quantity has to be counted separately. When the total number of errors in the test document and

[8] The suggestion test does not only count the number of falsely false words, it also gives a list of falsely false words (see in appendix II-1 the label *falsely false words*). *Falsely false words* are referred to as *false hits* in the current research.

the number of correctly detected strings are known, detection recall, correction recall and precision can be computed. For computing detection recall and precision, the number of correctly detected errors is required. This is the total number of detected errors minus the number of incorrectly detected errors, the latter being the number of false hits. Thus, the number of correctly detected errors by spell checker A can be computed as follows (expressed using the labels from appendix II-1):

(1) Nr correctly detected A = A errors – falsely false

Detection recall, correction recall and precision can then be defined as follows (expressed using the labels from appendix II-1):

(2) Detection recall A = Nr correctly detected A / Total nr errors in document

(3) Correction recall A = Nr correct suggestions A / Total nr errors in document

(4) Precision A = Nr correctly detected A / A errors

5.2. The performance of the BESL spell checker

Section 4.2 described how the lexicon-based BESL spell checker was built according to the Polderland method for building spell checkers. The BESL spell checker was built in service of the context-sensitive spell checker. The lexicon of the BESL spell checker is used twice within the context-sensitive spell checker (see section 4.3.2). There are two important reasons for measuring the performance of the BESL spell checker. In the first place, it is important to know the performance of the lexicon-based spell checker itself in order to be able to answer one of the main questions: to what extent does the performance of a lexicon-based spell checker improve when the context-sensitive spell checker would be combined with it? Secondly, it is important to know the performance of the BESL spell checker because the performance of the context-sensitive spell checker is (partly) dependent on it, since the suggestions made by the lexicon-based spell checker are the basis of the suggestions for the detected trigram as a whole.
For these two reasons, two separate tests are needed. First, a test is required that can test the performance of the BESL spell checker when used independent of the context-sensitive spell checker. This is the performance of the BESL spell checker for detecting and correcting errors in general. Second, a test for measuring the performance of the spell checker within the context of the context-sensitive spell checker is required. Within the context-sensitive spell checker, the BESL spell checker is used for finding suggestions for real words. Therefore, the second test should be able to measure the performance of the BESL spell checker for generating suggestions for real-word errors. Both tests have been executed. In the next two subsections, both tests and the results are discussed respectively.

5.2.1. Performance of the BESL speller for detecting and correcting errors in general

In order to know the performance of the BESL spell checker, a test is required that can determine values for detection recall, correction recall and precision. In section 5.1.2, the suggestion test was described. From the results of this test detection recall, correction recall and precision can be computed. Therefore, I chose to execute a suggestion test for measuring the performance of the BESL spell checker. As described in section 5.1.2, the suggestion test requires sets of erroneous words and their corresponding correct words. For the current test, these sets have been taken from the British English part of the International Corpus of English (ICE-GB), in which errors have been marked. There are 454 sets in the test document. It contains instances of all classes of errors that have been defined in section 2.2, table 2-2. These classes are repeated below, including an example from the ICE-GB test set for each class.

- Typing errors, e.g. *with* → *wih*
- Errors due to a cognitive or phonetic lapse, e.g. *competent* → *competant*
- Grammatical errors, e.g. *this* → *these*
- Non-word errors, e.g. *their* → *thier*
- Real-word syntactic errors, e.g. *they* → *the*
- Real-word semantic errors, e.g. *complementary* → *complimentary*

The suggestion test has been executed on these 454 sets of words. The user interface of the test can be found in appendix II-2. This picture shows the results of two spell checkers, called X and POLDERLAND. X is the Microsoft spell checker for British English and POLDERLAND is the BESL spell checker. Appendix II-2 shows that the BESL spell checker flagged more strings than the Microsoft spell checker, but less of the detected strings received the correct suggestion and there are more false hits.
From the results of this test, detection recall, correction recall and precision can be computed:

(5) Nr correctly detected Microsoft = 293 – 25 = 268
(6) Detection recall Microsoft = 268 / 454 = 0.59
(7) Correction recall Microsoft = 226 / 454 = 0.50
(8) Precision Microsoft = 268 / 293 = 0.91

(9) Nr correctly detected BESL = 306 – 39 = 267
(10) Detection recall BESL = 267 / 454 = 0.59
(11) Correction recall BESL = 214 / 454 = 0.47
(12) Precision BESL = 267 / 306 = 0.87

The performance measures for both the BESL spell checker and the Microsoft spell checker are also given in table 5-1.

Table 5-1 Results of the macro test for BESL spell checker and Microsoft spell checker

Measure	BESL spell checker	Microsoft spell checker
Detection recall	0.59	0.59
Correction recall	0.47	0.50
Precision	0.87	0.91

The results for correction recall and precision show that the Microsoft spell checker performs a little better than BESL spell checker, but the difference is small (respectively 0.50 and 0.47 correction recall). To know whether the correction recalls of both spell checkers really differ, a significance test is required. This test shows that the spell checkers perform equally well.[9]
However, recall is small for both spell checkers: 0.59 detection recall and about 0.50 correction recall. This corresponds with the expectation that 50% of all errors are corrected by lexicon-based non-word error techniques (see section 3.5.3). This expectation is based on a real-word error rate of 25% of all errors.

5.2.2. Performance of the BESL speller for generating suggestions for real-word errors

The lexicon of the lexicon-based spell checker is used twice within the context-sensitive spell checker. First, the lexicon-based spell checker checks for all three words in a trigram whether they are lexicon entries. Second, when an erroneous trigram is detected and suggestion trigrams have to be found, the lexicon-based spell checker generates suggestions for each word in the trigram, which can then be combined to form new trigrams.
As the suggestions generated by the lexicon-based spell checker lead to the suggestion trigrams presented by the context-sensitive spell checker, the performance of the lexicon-based spell checker should be optimised for giving suggestions for real-word errors. This optimisation is obtained by repeated testing. How this is done and what the results are is described below.

Testing material
In order to test the performance of the BESL spell checker for giving suggestions for real-word errors, a list of real-word errors and their corrections is required. I compiled a list of 134 real-

[9] On the sample (n) of 454 words, the Microsoft spell checker corrects 226 words and the BESL spell checker corrects 214 words.
The null hypothesis is "both proportions come from the same population", which means "both spell checkers perform equally well".
Proportion A: Pa=226/454 = 0.498
Proportion B: Pb=214/454 = 0.471
Expected proportion: P=(226+214)/(454+454) = 0.485
The z-score can be computed:
$Z=(Pa-Pb)/sqrt(P.(1-P).(1/454+1/454)) = (0.47 – 0.50) / sqrt(0.250 . 4.405.10^{-3}) = –0.904$
Supposing either a level of significance $\alpha = 0.05$ or $\alpha = 0.10$, z is not in the rejection area of the normal distribution. Therefore, the null hypothesis is not rejected; the spell checkers perform equally well.

word errors from three sources: Vivian Cook's list of L2 English spelling mistakes, Mitton's (1987) sets of common wrong-word errors and the list *Words Commonly Confused*, used by Golding and Schabes (1996).

Vivian Cook compiled a list of 1400 learner English spelling mistakes from written work by L2 learners of English, taken from the Longman Corpus of Student English and his own collections. Since I am only interested in the real-word errors, I reduced the list of 1400 spelling errors into a list of 89 real-word errors by removing all sets of words of which one was marked as erroneous by the lexicon-based BESL spell checker in Microsoft Word. I also removed duplicates (some errors had been made by more than one student). In Cook's data, two words in a confusion set are not exchangeable: in all cases, the left-hand word was the misspelling of the right-hand word.

Mitton (1987), whose research was described in section 2.1.1, collected (among other data) nine common wrong-word errors. In Mitton's data, two words in a confusion set are exchangeable: the test data contained occurrences of a left-hand word being misspelled as the corresponding right-hand word and occurrences of the right-hand word being misspelled as the left-hand word.

Golding and Schabes (1996), whose research was also described in section 2.1.1, used the 71 confusion sets from the list *Words Commonly Confused* in the back of the Random House Unabridged Dictionary (Flexner, 1983). Like the words from Mitton's confusion sets, words from the *Words Commonly Confused* sets are exchangeable.

Combining the three confusion sets led to the introduction of some duplicates in the new set. I removed these manually.

The errors on the list have been classified into the classes described in chapter 2. The list contains instances of four of the six error classes that were defined in section 2.2, table 2-2. It contains typing errors, errors due to a cognitive or phonetic lapse, real-word syntactic errors and real-word semantic errors. There are no grammatical errors in the list and obviously no non-word errors. The four classes included in the real-word error list are listed below, with an example from the real-word error list for each class.

- Syntactic error due to a typo, e.g. *four* → *hour*
- Syntactic error due to a cognitive or phonetic lapse e.g. *here* → *hear*
- Semantic error due to a typo, e.g. *play* → *pay*
- Semantic error due to a cognitive or phonetic lapse, e.g. *piece* → *peace*

The complete classified list can be found in appendix III.

Testing method

The list of 134 real-word errors is used for optimising the performance of the BESL spell checker for giving suggestions for real-word errors. As described in section 4.2.3, the program Context can be used for testing the quality of the suggestions generated by a spell checker. The number of suggestions generated by Context can be changed manually. This feature is used during the process of optimising performance: the number of suggestions is varied in order to increase the number of correct suggestions. However, varying the penalty configurations in phonrulesEN.txt is

even more important for optimising the suggestion performance.[10] As described in section 4.2.3, it is possible to vary the penalty of different character transformations. For example, it is possible to give a higher penalty for substitution (*alloy* → *alliy*) than for deletion (*alloy* → *aloy*) of a character. Thus, changing the penalty configurations of a lexicon-based spell checker can change its performance for different kinds of errors.
By repeatedly changing the configurations of the phonrules file and Context, the suggestion performance of the BESL spell checker has been optimised. The performance is measured by giving the right-hand words from the 134 real-word error sets as input in Context and checking whether the left-hand words are among the suggestions.

Results
For the optimal configuration of the BESL spell checker, I computed which fraction of the words got a correct suggestion for each class of real-word errors. The results are shown in table 5-2.

Table 5-2 Fraction of correct suggestions for real-word errors from all four classes

Cause	Real-word syntactic errors	Real-word semantic errors	Total
Typo	0.83	0.89	0.86
Cognitive or phonetic lapse	0.66	0.63	0.65

The total correction recall for real-word errors is 0.72.
The results show that the BESL spell checker more often gives the good suggestion for typing errors than for cognitive or phonetic lapses. The reason for this is that in general typing errors orthographically resemble the original word more closely than errors due to phonetic lapses (see chapter 2, section 2.1.2). The question now arises where the large difference between the 0.72 correction recall for real-word errors and the 0.47 total correction recall from the first performance test comes from. The answer lies in the difference between Microsoft Word and Context as 'entrance' to the spell checker. Microsoft Office spell checkers generate at most six suggestions for a word, whereas the number of suggestions generated by Context can be changed manually and was set on thirty in order to increase the correction recall.[11]

[10] Part of phonrulesEN.txt is given in appendix I. The denotations of the rules in this file are explained in section 4.2.3.

[11] In an earlier version of the BESL spell checker, the maximum number of suggestions of Context was set on eight, resulting in a correction recall for real-word errors of 0.41. However, the two figures cannot be compared directly, since the penalty configurations were also different between the two versions.
Unfortunately, increasing the number of suggestions given by Context to thirty decreases the system's speed enormously, since every word in the detected trigram gets at most thirty suggestions. The three words are combined, resulting in a lot of trigrams. All those trigrams are searched in the trigram database, which takes much time.

5.3. The performance of the context-sensitive spell checker

In this section, the performance of the context-sensitive spell checker is described. In subsections 5.3.1 and 5.3.2, the test material and the test method for testing the performance of the context-sensitive spell checker are described respectively. In subsection 5.3.3, the results of the performance test are described and discussed, followed by a computation of the (expected) improvement of the BESL spell checker for non-word errors when it would be combined with the context-sensitive spell checker (subsection 5.3.4).

5.3.1. Test material

In order to test the performance of the spell checker properly, I created two test corpora. First, a part of the malapropisms corpus of Hirst and Budanitsky (2001) was used. Hirst and Budanitsky created a corpus containing malapropisms artificially. They took 500 articles and replaced one word in every 200 words by a spelling variation (a valid word that is one or a few basic transformations away from the original word). To be a candidate for replacement, a word:

- had to be present in the spell checker lexicon,
- needed to have at least one spelling variation that was also in the lexicon, and
- should not be a function word or a proper noun.

Using the Hirst and Budanitsky malapropisms corpus as test corpus for the current research has three disadvantages. First, the corpus does not contain syntactic errors. Second, it does not contain errors in function words. Third, the corpus does not contain errors that are more than a few basic character transformations away from the original word, like some errors resulting from phonetic lapses (e.g. *cite* → *sight*). Since Hirst and Budanitsky used a context-sensitive spell checking method based on semantic information, excluding function words and syntactic errors in their test corpus is not a problem for their own research. However, since I aim at creating a method for detecting and correcting all classes of real-word errors in all classes of words, the Hirst and Budanitsky corpus does not suffice.

Thus, I created another test corpus from part of the BNC combined with the 134 real-word errors sets, which were described in section 5.2.2 (see also appendix III). In the BNC texts, all occurring right-hand words from this list have been replaced by their corresponding left-hand word.[12] Since the list of 134 real-word errors contains instances of all classes of real-word errors (except grammatical errors) in all classes of words, the BNC test corpus does too. However, a large disadvantage of the BNC test corpus is that the trigram database has been compiled from the BNC too. Therefore, all correct trigrams in the test corpus are present in the trigram database. Since the spell checking algorithm can find all correct trigrams in the database, there will not be any false hits. In the Hirst and Budanistky test corpus there can be false hits, since a trigram in this corpus is not necessarily present in the BNC, even though it is correct. Thus, the Hirst and Budanitsky test corpus is more representative for 'real' data than the BNC test corpus.

[12] The Perl program that was used to replace words in the corpus can be found in appendix IV-6.

From eight randomly chosen texts from both test corpora, a test corpus of about 12,500 words was compiled. In table 5-3, the composition of the test corpus is shown.

Table 5-3 Composition of the test corpus for context-sensitive spell checking

Source corpus	Nr of texts	Total nr of words
BNC	8	5476 words
Hirst & Budanitsky	8	7142 words

The Hirst and Budanitsky test corpus contains 93 erroneous trigrams, which corresponds with 31 errors since every error in the text is included in three trigrams. The BNC test corpus contains 1818 erroneous trigrams, which corresponds with 606 errors.

5.3.2. Testing method

In order to measure the performance of the context-sensitive spell checker, I aim at computing detection recall, correction recall and precision. In section 5.1.1, I gave my definitions for these measures. In the current section, these definitions are adopted, but trigrams are considered instead of words. Thus, detection recall is the number of correctly found erroneous trigrams divided by the total number of erroneous trigrams in the text, correction recall is the number of correct suggestions divided by the total number of erroneous trigrams in the text and precision is the number of correctly detected erroneous trigrams divided by the total number of detected trigrams.
For computing detection recall, the total number of errors in the text and the number of correctly detected strings are required. For correction recall, the total number of errors in the text and the total number of errors that have been corrected are required. For computing precision, the number of detected strings and the number of correctly detected strings are required. For these counts, two files are required. First, a list of erroneous and corresponding correct trigrams from the test texts is needed. This list is compiled by comparing the original texts and the texts containing errors.[13] Second, the output (result) files from the context-sensitive spell checker are required. An example of the format of such an output file is given in appendix V. Each report of a detected trigram is followed by one or more suggestions. The number of suggestions for a detected trigram is not limited: all combinations of three word suggestions are presented. Because of that, some trigrams get many suggestions, in particular those trigrams that consist of short words. The trigram that has the highest number of suggestions is *her ad it* (intended trigram: *her and it*), which got 152 suggestions. The high number of candidate corrections for all three words caused this high number of suggestions. Fortunately, such a high amount of suggestions is exceptional: the average number of suggestions per detected trigram is approximately eight[14].
The output of the context-sensitive spell checker was evaluated using the list of erroneous and corresponding correct trigrams. The Perl program for executing this evaluation is given in appendix IV-7. The result of this Perl program is an output file containing the false hits from the

[13] The Perl program for compiling this trigram list is given in appendix IV-5.
[14] Appendix IV-6 contains the Perl program that performs this count.

text, all required counts (the number of erroneous trigrams, the number of detected trigrams, the number of correct suggestions, the number of correctly detected trigrams and the number of false hits), detection recall, correction recall and precision. When all output files were evaluated, the averages for the two test corpora are computed. The results of the evaluation are described in the next subsection.

5.3.3. Results

Tables 5-4, 5-5 and 5-6 give the scores for detection recall, correction recall and precision respectively, per test corpus. BNC is the BNC test corpus; BUD is the Hirst and Budanitsky test corpus. Text minimum is the lowest score for a text of the test corpus; text maximum is the highest score. The average is calculated over all eight texts of one test corpus. In figures 5-1, 5-2 and 5-3 on page 56, the same results are shown graphically.

Table 5-4 Detection recall per test corpus

Test corpus	Average	Text minimum	Text maximum
BNC	0.72	0.61	0.85
BUD	0.51	0.42	0.67

Table 5-5 Correction recall per test corpus

Test corpus	Average	Text minimum	Text maximum
BNC	0.68	0.58	0.85
BUD	0.33	0.11	0.67

Table 5-6 Precision per test corpus

Test corpus	Average	Text minimum	Text maximum
BNC	0.98	0.95	1.00
BUD	0.05	0.03	0.07

The results for the Hirst and Budanitsky corpus are the most important for a discussion of the performance of the context-sensitive spell checker, since these texts and the errors in these texts are completely independent of the trigram database. As the trigram database is extracted from the BNC, all correct trigrams from the BNC test texts are in the database and no false hits are expected.[15]

[15] The absence of false hits implicates a precision of 1.00. As figure 5-3 shows, this is not the case for BNC texts. This means that some correct trigrams were not present in the trigram database. How is this possible? After studying the output files from the spell checker, I concluded that there was nothing wrong with either the spell checking algorithm or the algorithm for evaluating the output files. The most suitable explanation seems the incompleteness of the trigram database. Somehow, a number of trigrams must have disappeared from the database.

The first point of discussion is the large range in recall values for the Hirst and Budanitsky texts (see figure 5-2 and to a lesser extent figure 5-1). This large range is caused by the relatively small number of errors in the Hirst and Budanitsky test corpus. The number of erroneous trigrams per text varies from six to 21. Because of these small numbers, coincidence has a large influence and causes a large range between texts.

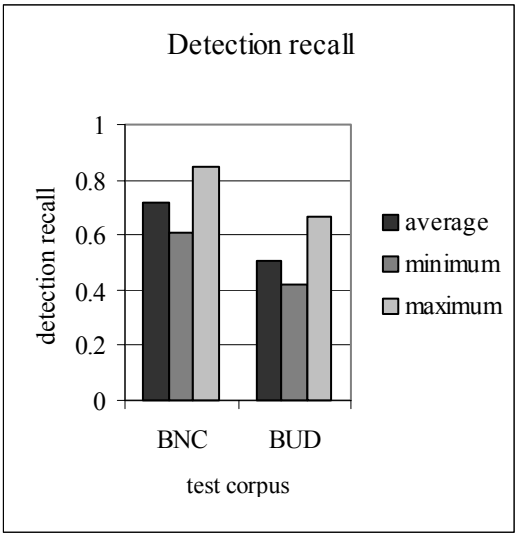


Figure 5-1 Detection recall

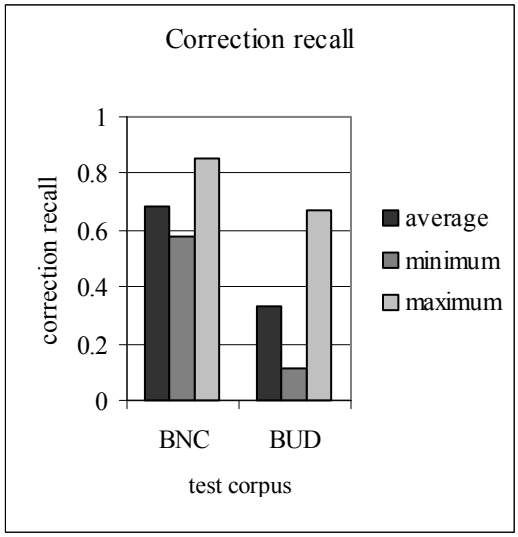


Figure 5-2 Correction recall

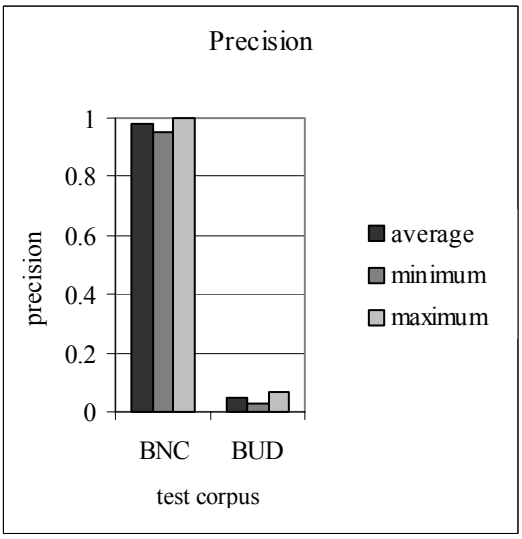


Figure 5-3 Precision

A second point of discussion is the striking difference between the error detection rate (figure 5-1) and the error correction rate (figure 5-2) for the Hirst and Budanitsky texts (0.51 and 0.33 respectively). This difference is much bigger than the difference between the detection and correction rate for BNC texts (0.72 and 0.68 respectively). This can probably be explained by the

fact that all correct trigrams in the BNC texts are present in the trigram database. Therefore, once an erroneous trigram has been detected, the correct trigram is likely to be suggested. On the other hand, the correction rate for the Hirst and Budanitsky texts is very low when compared to the detection rate, because the correct trigram is not necessarily present in the trigram database.
The question now arises why not all of the detected trigrams in the BNC texts get the correct suggestion. This can be explained by the fact that the BESL spell checker does not always generate a correct suggestion for the words in a trigram. As a result, the correct combination cannot be made.
Another point of discussion is the large number of false hits and the corresponding low precision for the Hirst and Budanitsky texts. This has the same reason as the large difference between detection recall and correction recall: trigrams not containing errors are not necessarily present in the trigram database and therefore likely to be falsely detected. Figure 5-3 shows that the precision for the Hirst and Budanitsky texts is very low. In fact, only 5% of all detected trigrams are rightly detected. The remaining 95% are false hits.

5.3.4. Improvement with respect to the BESL spell checker

Now that the fraction of real-word errors that is detected and corrected by the context-sensitive spell checker is known, we can compute the (expected) improvement in the performance of the lexicon-based BESL spell checker when it would be combined with the context sensitive spell checker. For this computation, I use the results for the Hirst and Budanitsky texts, since these are the most representative for real data. As table 5-1 shows, the BESL spell checker detects 59% of all errors. According to table 5-4, the context-sensitive spell checker detects 51% of real-word errors. Supposing a real-word error rate of 25% of all errors (see section 2.1.2), this implicates that the context-sensitive spell checker detects 12.8% (51% of 25%) of all errors. Adding 59 and 12.8 yields a total detection rate of 72% for the spell checking application. This is an improvement of 22% (i.e. 12.8/59 * 100%) with respect to the original lexicon-based spell checker.
For correction, a similar computation can be done. As table 5-1 shows, the BESL spell checker corrects 47% of all errors. According to table 5-5, the context-sensitive spell checker corrects 33% of real-word errors. Again, supposing a real-word error rate of 25% of all errors, this implicates that the context-sensitive spell checker corrects 8.3% (33% of 25%) of all errors. Adding 47 and 8.3 yields a total correction rate of 55% for the spell checking application. This is an improvement of 17% (i.e. 8.3/47 * 100%) with respect to the original lexicon-based spell checker.

6. Conclusion

The concluding chapter is divided into four parts. First, the main research questions as stated in the Introduction are answered (section 6.1). Then the current research is evaluated (section 6.2). In section 6.3, a comparison is made to the results of other studies on context-sensitive spell checking. Finally, some suggestions for further research are proposed in section 6.4.

6.1. Answering the main questions

In the Introduction, the following main research questions were formulated:

1. What proportion of real-word errors can be detected and corrected using context-sensitive spell checking based on word trigram probability information?

2. To what extent do the results of a lexicon-based spell checker improve when the context-sensitive spell checker is combined with the lexicon-based spell checker?

In this section I will try to answer these questions and to evaluate the results in the light of a possible application in a state of the art spell checker.
First, as described in section 5.3.3, 51% of all real-word errors were detected using context-sensitive spell checking based on word trigram probability information. 33% of all real-word errors have been corrected. Especially the detection rate seems high enough for a real application. As a comparison, a lexicon-based spell checker like the Microsoft spell checker or the BESL spell checker has a detection rate of 59%. Unfortunately, the correction rate is low: only one out of three erroneous trigrams gets the correct suggestion, whereas one out of two is detected.
Second, the expected improvement of the spell checking application with respect to the lexicon-based spell checker has been computed in the previous section. The detection rate rises from 59% to 72%. The correction rate rises from 47% to 55%. This means that the performance of the lexicon-based spell checker improves with 22% for detection and 17% for correction when the context-sensitive spell checker is combined with the lexicon-based spell checker.
Detection and correction recall suggest that this context-sensitive spell checking algorithm (possibly with some small adaptations) performs well enough for a real application. Unfortunately, precision is much too low: only 5% of all detected trigrams are rightly detected. An application that has this performance is not reliable enough for a user to work with.
Some suggestions for changes that are expected to improve the system's performance are given in section 6.4.

6.2. Evaluation of the current research

In retrospect, some parts of the current research should have been done differently. In this section, three criticisms on the current research are described.

The first point of criticism concerns the validity of the test corpus for testing the performance of the context sensitive spell checker. First of all, the overall performance results could only be based on the Hirst and Budanitsky corpus, since the BNC corpus was already used as the training corpus. However, the Hirst and Budanitsky corpus is not very suitable for testing purposes for various reasons. First, the error density of the Hirst and Budanitsky corpus is very low: only one in every 200 words contains an error. Therefore, the corpus is not very representative and the range between the performances for the eight texts was very large. In order to handle the low error density in the Hirst and Budanitsky corpus, I probably should have used a larger subset of their corpus. Second, the research aimed at testing all classes of real-word errors. To achieve this, all classes of errors had been put in the BNC test corpus. However, the Hirst and Budanitsky corpus contains only semantic errors (malapropisms). Therefore, only the performance for semantic errors is measured. I do not expect the results for syntactic errors to be very different from those for semantic errors, but it would have been better to test on all classes of real-word errors.

The second point of criticism concerns the trigram extraction algorithm. As described in section 4.4.2, sentence borders are removed when extracting trigrams from the BNC. The number of trigrams that cross a sentence border is not negligible: one in every ten trigrams crosses a sentence border. Unless the fact that in the text that is spell checked sentence borders are removed too, it would have been better not to ignore sentence borders, because it results in one meaningless trigram in every ten trigrams.

The third point of criticism is that there is a small deficiency in the context-sensitive spell checking algorithm. In order to increase precision, a trigram is only detected when it has one or more suggestions. Therefore, the rule that prints the detected trigram is in the same loop as the rule that prints the possible suggestion. Unfortunately, this means that in the output file the detected trigram is printed for every suggestion. This had to be transformed in a format in which the detected trigram is printed once and all suggestions are printed under it (see appendix V), in order to make the file valid as input for the evaluation program.

A fourth point of criticism concerns the general architecture of the spell checker. For the current research, I decided to build a ‘stand alone’ context sensitive spell checker, i.e. a spell checker that would only correct real-word errors and not non-word errors. However, a check for non-word errors has to be performed all the same in order to ignore those trigrams that include a non-word error (see section 4.3, figure 4-4). In retrospect, it seems nonsensical not to correct the non-word error at the same time. In other words, it makes sense to combine the detection and correction of non-word errors and real-word errors in one system from the start. This way, no detection steps have to be made twice. As a result, the whole process would be much faster.

6.3. Comparing the results of the current research to those of other studies

It would be interesting to compare results of the current research to the results of Golding and Schabes (1996), St-Onge (1995) or Hirst and Budanitsky (2001), because they try to solve the problem of real-word errors using other methods than I did. Unfortunately, none of these methods consider all classes of real-word errors in all classes of words (see chapter 3, sections 3.6 and 3.7). Therefore, my results cannot be compared one on one to the results reported for these studies. In spite of that, I will shortly discuss their results and try to make a comparison to my results in so far as possible.

6.3.1. Golding and Schabes (1996)

Golding and Schabes (1996) consider a finite number of confusion sets (18). They compute the performance of their system for each confusion set separately. In their test corpus, they replaced instances of words from the confusion sets by another word from the same set. Then they computed for each confusion set how often the system suggests replacing the erroneous word by the original word. For example, they replaced several instances of *their* by *there*. In 87.6% of the occurrences, the system suggests replacing the incorrect *there* by *their*. Unfortunately, I cannot compare the results of the current research to the results obtained by Golding and Schabes because I did not consider separate confusion sets in computing my results.

6.3.2. Hirst and Budanitsky (2001)

Hirst and Budanitsky (2001) measure the performance of their system for detecting and correcting malapropisms using detection recall, correction recall and precision. They found a detection recall varying from 23.1% to 50%, a precision varying from 18.4% to 24.7% and a correction recall varying from 2.6% to 8%.
Since I used a test corpus that contains the same class of real-word errors as the test corpus used by Hirst and Budanitsky, it is possible to compare my results to theirs. In table 6-1, the results are compared.

Table 6-1 Comparison of the results of the current research to those of Hirst and Budanitsky

	Hirst and Bud. (min.[16])	Hirst and Bud. (max.[16])	Current research
Detection recall	23.1%	50%	51%
Correction recall	2.6%	8%	33%
Precision	18.4%	24.7%	5%

Table 6-1 shows that Hirst and Budanitsky's method, which is based on semantic information, yields a higher precision than my method, which is based on probabilistic information. This

[16] In evaluating their system, Hirst and Budanitsky (2001) used different search scopes in determinations of semantic relatedness: just the paragraph containing the target word (scope = minimum); that paragraph plus one or two adjacent paragraphs on each side; and the complete article (scope = maximum).

means that their system detects fewer strings incorrectly. On the other hand, Hirst and Budanitsky's method performs much worse when considering correction recall. Apparently, it is very hard for a system based on semantic information to find the appropriate correction for a detected error.
From their results, Hirst and Budanitsky conclude that their system approaches practical usability. They state:

> *It is not realistic to expect absolute correctness, 100% precision and recall, nor is this level of performance necessary for the system to be useful. In conventional interactive spelling correction, it is generally assumed that very high recall is imperative but precision of 25% or even less is acceptable – that is, the user may reject more than 3 out of 4 of the system's suggestions* (words detected by the system) *without deprecating the system as 'dumb' or not worth using*
> (Hirst and Budanitsky, 2001, p. 24).

However, I concluded that the performance of the system that was described in the current research is much too poor for a real application (section 6.1). I argued that this is mainly because of the low precision. The precision of Hirst and Budanitsky's system is higher, but in my opinion still too low for a real application. After all, with a precision of approximately 20%, the user will reject four out of five of the system's flaggings. Moreover, with a correction recall of approximately 5%, the system suggests the correct suggestion for only one out of twenty words. An application that has this performance is not reliable enough for a user.

6.3.3. St-Onge (1995)

St-Onge (1995) also computed detection recall, correction recall and precision for his method for detecting and correcting malapropisms. He found a detection recall of 28.5%, a correction recall of 24.8% and a precision of 12.5%. These results are compared to the results of the current research and those of Hirst and Budanitsky in table 6-2.

Table 6-2 Comparison of the results of St-Onge and Hirst and Budanitsky to the current research

	St-Onge	Hirst and Bud.[17]	Current research
Detection recall	28.5%	35%	51%
Correction recall	24.8%	5%	33%
Precision	12.5%	20%	5%

Table 6-2 shows that the system of St-Onge is more reliable than Hirst and Budanitsky's method for suggesting corrections. Unfortunately, like in the current research, precision is much too low.

[17] The results of Hirst and Budanitsky are indications, based on the minimum and maximum values shown in table 6-1.

St-Onge states that his method could be used in a real application when combined with other methods:

> *Full malapropism detection with no false-alarms at all cannot be expected with the approach proposed here. However, I believe that better performances and the integration of this malapropism detection algorithm to a spelling checker could make a commercializable product.*
> (St-Onge, 1995, p. 49)

I consider the view of St-Onge to be far more realistic than that of Hirst and Budanitsky.
After all, I think that methods based on probabilistic information are more promising than methods based on semantic information, because I prefer a method that aims at detecting and correcting all classes of real-word errors in all classes of words. Moreover, the current research shows that in a probabilistic method recall can be reasonably high. The methods of St-Onge and Hirst and Budanitsky can detect and correct all classes of real-word errors but by using semantic information, they ignore errors in function words. However, I have to make one note at this conclusion: the algorithm of the current research performs spell checking much too slow for a real application. Therefore, I conclude that probabilistic methods are preferable above semantic methods, under the condition that the algorithm can be executed in real-time.

6.3.4. Mays, Damerau and Mercer (1991)

As the current research has most in common with the study of Mays, Damerau and Mercer (1991), it would also be interesting to compare my results to their results. But unfortunately, they did not use real test data to measure the performance of their spell checking application. Therefore, I cannot really compare my detection recall (51%) to theirs (76%) and my correction recall (33%) to theirs (74%).
The study of Mays, Damerau and Mercer (1991) is the only other research in the literature that uses word trigram probability information as a possible solution to the problem of real-word errors. However, they built an application of which the performance could not be measured on real data, since their algorithm uses predefined sentence cohorts (see chapter 3, section 3.6.1). Thus, the method used by Mays, Damerau and Mercer is not valid for checking a real text. The method in the current research is therefore the first context-sensitive spell checking method that can be used for all types of real-word errors in all word classes and that is valid for checking real text. However, as concluded in section 6.1, the performance of the current method is not good enough for a real application. The changes that should be made in order to make the system perform well enough, are described in section 6.4.

6.4. Further research

As described in section 6.1, there are two main reasons why the current spell checker is not appropriate for a real application. First, the spell checking program runs much too slow. This problem cannot be solved easily. One could wait for computers to be faster or one could change the algorithm. Whether changing the algorithm can solve the speed problem enough needs further research.
Second, precision is much too low: only 5% of all detected trigrams are rightly detected. An application that has this performance is not reliable for a user. I do not expect the precision to get much higher when a larger corpus for extracting trigram probabilities is used: there will always be many correct trigrams in a text that are not in the trigram database.
I expect the improvement of the system has to be found in changing the algorithm. There are several ways for possibly improving the spell checking method. A variant of the original algorithm could be useful for a real application. Three different variants could be studied.
First, a variant of the algorithm, in which a trigram is only detected if it has a very high-probability suggestion, could be feasible. For example, when a trigram is not in the trigram database but one of its suggestion trigrams occurs in the BNC over a hundred times, then the original trigram is very likely to be an misspelling of this high-frequent trigram. A trigram should only be detected when it has such a high-frequent variant. Detection rate will be lower but more of the detected trigrams will be correctly detected. A good balance between precision and recall should be found in order to create a useful application. The possibilities for such an algorithm should be studied.
Another possibility for improving performance could perhaps be using word probability information. Maybe a trigram that only contains very low-frequent words is less likely to be the correct suggestion than a trigram of high-frequent words. This should be found out. Possibly, word probability information could also be used for detecting trigrams that are low frequent but occur in the BNC. When a trigram contains three high-frequent words but the trigram itself has a low number of occurrences in the database, it may be more likely to contain an error than when the words in the trigram are low frequent words themselves. Whether this is indeed true should be found out.
Research could also be done into the use of text type information. For a specific text type with a specific subject, a trigram database could be built from a corpus containing this kind of texts. Then less incorrect suggestions are possibly given. For example, when building a context-sensitive spell checking application for children, vocabulary would be smaller and therefore the trigram database would be smaller. Only trigrams that are likely to be used by children could then be suggested.
There should be more research into all these variants in order to find out what kind of application could be derived from the current research.

References

Allison, L. (1999), *Trie,* Department of Computer Science and Software Engineering, Monash

Burnard, L. (2000), *Reference Guide for the British National Corpus (World Edition)*

Carter, D. M. (1992), *Lattice-based word identification in CLARE*, in 'Proceedings of the 30th Annual Meeting of the Association for Computation Linguistics', Newark, Del., June 28 – July 2, ACL: 159-166

Cook, V.J. (1997), *L2 users and English spelling*, in 'Journal of Multilingual and Multicultural Development', 18, 6: 474-488

Damerau, F. J. (1964), *A technique for computer detection and correction of spelling errors*, in 'Communications of the ACM', 7(3): 171-176

Damerau, F. J. and Mays, E. (1989), *An Examination of Undetected Typing Errors*, in 'Information Processing & Management', 25(6): 659-664

Flexner, S. B. (Ed.) (1983), *Random House Unabridged Dictionary*, 2nd edition. Random House, New York

Golding, A. R. and Schabes, Y. (1996), *Combining trigrambased and feature-based methods for contextsensitive spelling correction*, in 'Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics', Santa Cruz, CA

Grudin, J. (1983), *Error patterns in skilled and novice transcription typing*, in 'Cognitive Aspects of Skilled Typewriting', W. E. Copper, Ed. Springer-Verlag, New York

Hirst, G. J. and Budanitski, A. (2001), *Correcting Real-Word Spelling Errors by Restoring Lexical Cohesion*, Department of Computer Science, Toronto, Ontario, Canada

Kernighan, M. D. (1991), *Specialized spelling correction for a TDD system*, AT&T Bell Labs Tech. Mem., August 30.

Kukich, K. (1990), *A comparison of some novel and traditional lexical distance metrics for spelling correction*, in 'Proceedings of INNC-90-Paris', 309-313

Kukich, K. (1992a), *Spelling correction for the telecommunications network for the deaf*, in 'Communications of the ACM', 35(5): 80-90

Kukich, K. (1992b), *Techniques for automatically correcting words in text,* in 'ACM Computing Surveys', 24: 377-439

Landauer, T. K. and Streeter, L. A. (1973), *Structural differences between common and rare words*, in 'Journal of Verbal Learning and Verbal Behaviour', 12: 119-131

Leech, G. (1991), *The state of the art in corpus linguistics*, in Aijmer, K. and Altenberg, B. (eds.), 'Introduction to English Corpus Linguistics', Longman, London, 1991: 8-29

Levenshtein, V. I. (1966), *Binary codes capable of correcting deletions, insertions and reversals*, in 'Sov. Phys. Dokl'. 10 (Feb): 707-710

Lowrance, R. and Wagner, R. (1975), *An extension of the string-to-string correction problem*, in 'J. ACM', 22, 2 (Apr.): 177-183

Mays, E., Damerau, F. J. and Mercer, R. L. (1991), *Context based spelling correction*, in 'Proceedings of the IBM Natural Language ITL', 517-522, Paris, France

Mitton, R. (1987), *Spelling Checkers, Spelling Correctors, and the Misspellings of Poor Spellers*, in 'Information Processing & Management', 23(5): 495-505

Mitton, R. (1996), *English Spelling and the Computer Longman*, London

Muth, F. E. Jr. and Tharp, A. L. (1977), *Correcting human error in alphanumeric terminal input*, in 'Information Processing and Management', 13: 329-337

Park, S. H. and Gero, J. S. (1999), *Qualitative representation and reasoning about shapes*, in Gero, J. S. and Tversky, B.(eds.), 'Visual and Spatial Reasoning in Design, Key Centre of Design Computing and Cognition', University of Sydney, Sydney, Australia: 55-68

Peterson, J. L. (1986), *A note on undetected typing errors*, in 'Communications of the ACM', 29(7): 633-637

Pollock, J. J. and Zamora, A. (1983), *Collection and characterization of spelling errors in scientific and scholarly text*, in 'Journal of American Society of Informatics and Science', 34(1): 51-58

Pollock, J. J. and Zamora, A. (1984), *Automatic spelling correction in scientific and scholarly text*, in 'Communications of the ACM', 27(4): 358-368

St-Onge, D. (1995), *Detecting and correcting malapropisms with lexical chains,* Master's thesis, Department of Computer Science, University of Toronto. Also published as Technical Report CSRI-319

The TEMAA website: http://cst.dk/projects/temaa/D16/Heading15

Tillenius, M. (1996), *Efficient generation and ranking of spelling error corrections*, *Technical Report TRITA-NA-E9621*, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm

Veronis, J. (1988), *Computerized correction of phonographic errors*, in 'Comput. Hum.', 22: 43-56.

Wagner, R. A. and Fischer, M. J. (1974), *The string-to-string correction problem*, in 'J ACM' *21*, 1 (Jan.): 168-178

Wing, M. and Baddeley, A. D. (1980), *Spelling errors in handwriting: A corpus and distributional analysis*, in 'Cognitive Processes in Spelling', U. Frith. Ed. Academic Press, London

Wu, Z. B., Hsu, L. S., Chew, L. T. (1992), *A Survey on Statistical Approaches to Natural Language Processing*, National University of Singapore

Yannakoudakis, E. J. and Fawthrop, D. (1983), *The rules of spelling errors*, in 'Information processing and management', 19(12): 101-108

Young, C. W., Eastman, C. M. and Oakman, R. L. (1991), *An analyses of ill-formed input in natural language queries to document retrieval systems*, in 'Information Processing and Management', 27(6): 615-622

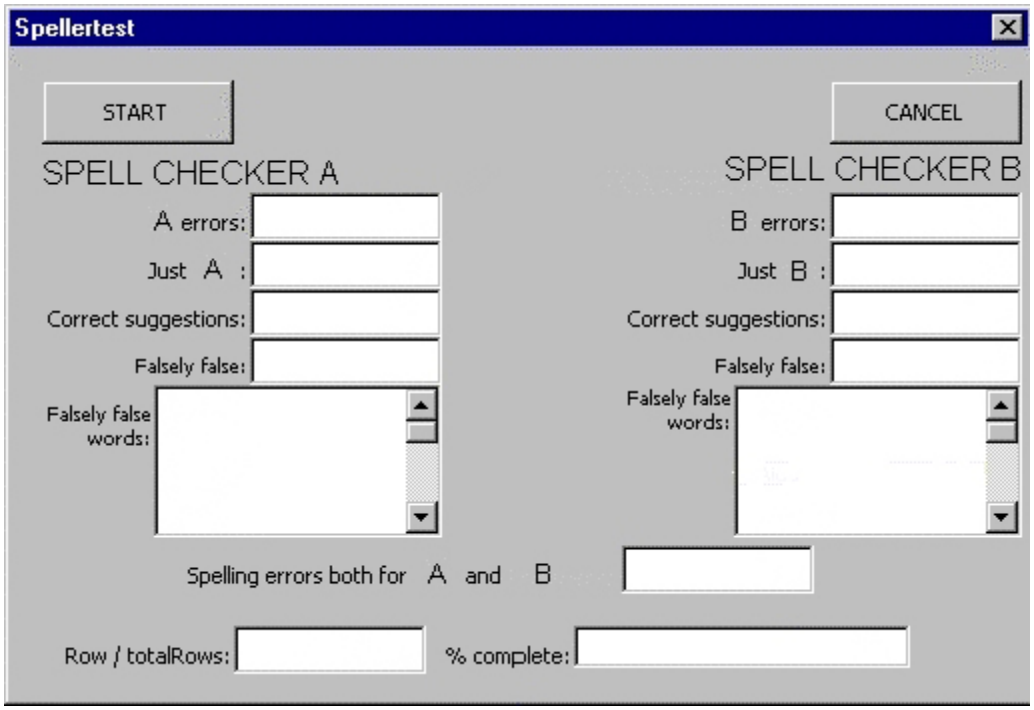
Appendices

Appendix I – a sample of phonrulesEN.txt

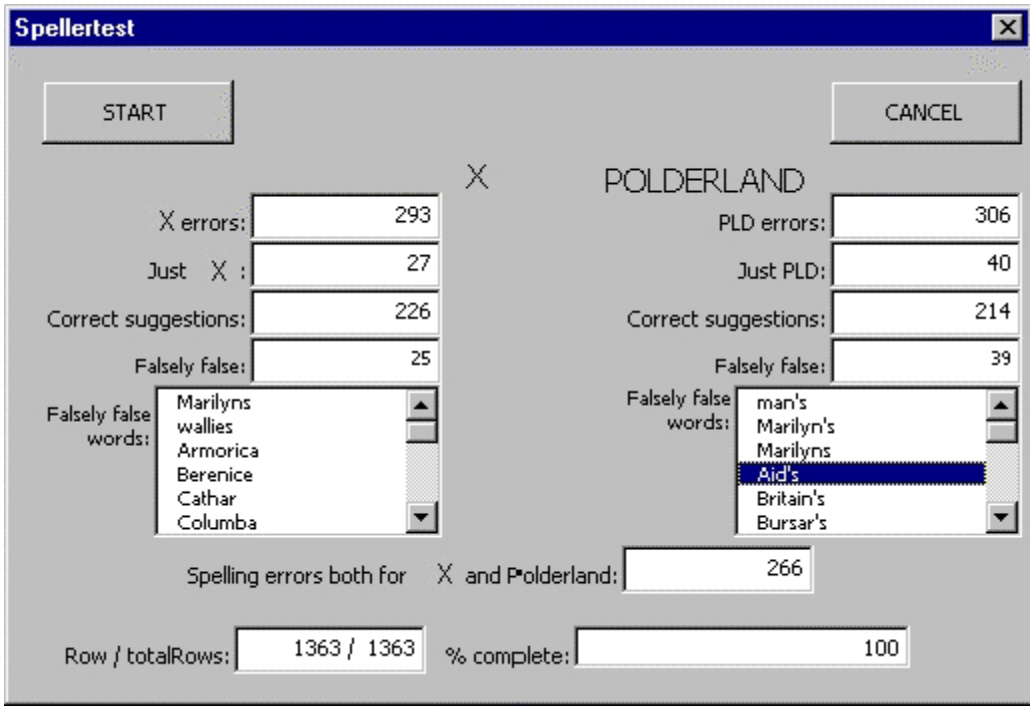
```
(1)   %#!MatchPenalties
(2)   ins=0001
(3)   del=0001
(4)   sub=0001
(5)   alt=0001
(6)   hyph=0000
(7)   doub=0002
(8)   undoub=0002
(9)   twin=0001
(10)  kin=0001
(11)  cousin=0001
(12)  %#!PenaltyLimits
(13)  limbase=12
(14)  limincr=2
(15)  limmax=50
(16)  maxdistbase=8
(17)  maxdistincr=0
(18)  maxdistmax=8
(19)  %#!MatchRules
(20)  ;RULE  CLASS  REMARK
(21)  /io<=>u/     kin     ;fashion
(22)  /io<=>e/     kin     ;fashion
(23)  /oe<=>u/     kin     ;Phoenician
(24)  /oi<=>u/     cousin ;porpoise,tortoise
(25)  /oi<=>e/     cousin  ;porpoise,tortoise
(26)  /ou<=>e/     cousin  ;famous
(27)  /$Mc<=>$Mac/ twin    ;McDonalds, MacIntosh
(28)  sch<=>sk/    kin    ;school, ski
(29)  /ce<=>sh/    kin    ;ocean
(30)  /ci<=>sh/    kin    ;special
(31)  /sc<=>sh/    kin    ;fascism
(32)  /si<=>sh/    kin    ;emulsion
(33)  /ti<=>sh/    kin    ;station
(34)  /sci<=>sh/   kin    ;conscious
(35)  /ssi<=>sh/   kin    ;mission
(36)  /ght<=>t/    kin    ;nought
(37)  /f<=>v/      twin   ;very, fairy
(38)  /ph<=>v/     kin    ;Stephen
(39)  /u<=>w/      kin    ;persuade
(40)  /y<=>j/      cousin ;yard
(41)  /s<=>z/      cousin ;haze
(42)  /ible<=>able/twin
(43)  /c=>$ç/      cousin ;garçon
(44)  /$&<=>and/   cousin
(45)  /$.=>/       twin
(46)  /$.=>dot/    kin
```

Appendix II – User interfaces

1. User interface of the macro test for comparing two spell checkers



2. User interface of the macro test for comparing the BESL speller and the MS speller



Appendix III – Classified list of 134 real-word errors

1 = typo, syntactic error
2 = cognitive / phonetic lapse, syntactic error
4 = typo, semantic error
5 = cognitive / phonetic lapse, semantic error

a-am	1	weeding-wedding	1
ad-and	1	ales-else	2
an-and	1	altar-alter	2
anther-another	1	an-a	2
ay-any	1	are-our	2
beg-big	1	brake-break	2
begin-being	1	Braun-brown	2
bough-bought	1	buy-by	2
bout-about	1	by-be	2
car-can	1	cent-sent	2
cold-could	1	cite-sight	2
Crete-create	1	coast-cost	2
fee-free	1	comment-common	2
form-from	1	cool-call	2
hat-had	1	descript-described	2
hold-old	1	draw-drawer	2
hour-four	1	dual-duel	2
lather-leather	1	extent-extend	2
Lear-learn	1	foreword-forward	2
loner-longer	1	good-goods	2
metal-mental	1	hear-here	2
on-an	1	its-it's	2
out-our	1	know-now	2
quite-quiet	1	life-live	2
re-are	1	lither-leather	2
run-rum	1	ma-my	2
stud-study	1	may-my	2
test-best	1	miner-minor	2
the-then	1	moral-morale	2
the-they	1	of-off	2
thing-think	1	off course-of course	2
though-thought	1	past-passed	2
tried-tired	1	pedal-peddle	2
wan-want	1	plain-plane	2

pleas-please	2
principal-principle	2
red-read	2
role-roll	2
see-sea	2
shear-sheer	2
staid-stayed	2
than-then	2
their-there	2
to-too	2
tree-three	2
trough-through	2
weather-whether	2
were-where	2
who's-whose	2
Wright-right	2
your-you're	2
bens-bends	4
chord-cord	4
country-county	4
derived-deprived	4
flounder-founder	4
hangar-hanger	4
heart-head	4
heat-heart	4
hopping-hoping	4
knee-kneel	4
law-lawn	4
litter-letter	4
mane-man	4
ordinance-ordnance	4
pasted-passed	4
pay-play	4
radial-radical	4
Walsh-Welsh	4
wards-words	4
breach-breech	5
accept-except	5
adapt-adopt	5
advice-advise	5
affect-effect	5
aid-aide	5
aloud-allowed	5
bloc-block	5
born-borne	5
bough-bow	5
complement-compliment	5
compose-comprise	5
consul-council	5
corps-corpse	5
desert-dessert	5
eminent-immanent	5
feel-fill	5
flair-flare	5
forceful-forcible	5
fortuitous-fortunate	5
idle-idol	5
lay-lie	5
lead-led	5
loose-lose	5
loosing-losing	5
massage-message	5
naval-navel	5
palate-palette	5
peace-piece	5
persecute-prosecute	5
perspective-prospective	5
precede-proceed	5
raise-rise	5
tanking-thanking	5
toke-took	5

Appendix IV – Perl programs

1. The Perl program for extracting relevant information from BESL

```
#!/usr/pkg/bin/perl
#perl program to extract a usable lexicon containing for each id every possible
word form from BESL.

use strict;
#use warnings;

my $regel;

while ($regel=<>) {
       my @forms;
       my $id;
       my $hw;
       my $pos;

       if ($regel =~ /<se.*id=([0-9]*)>/) {
              $id = $1;
       }
       if ($regel =~ /<hw.*>(.*)<\/hw>/) {
              $hw = $1;
       }
       if ($regel =~ /<ps>(.*?)[&<,(]/) {
              $pos = $1;
       }
#testing whether there is more than one occurrence of <form>

       while ($regel =~ /<form>([a-zA-Z]*)<\/form>/g) {
           push @forms, $1;
       }
#first occurrence of <form> contains the same content as <hw>,
#thus only printing second occurrence of <form>

       if ($hw ne $forms[0]) {
           unshift @forms, $hw;
       }

       for my $f (@forms) {
           print "$f\t$pos\n";
       }
}
```

2. The Perl program for editing the BESL word list

```
#! usr/bin/perl

# This program removes multi-word entries, replaces single hyphens by
# double hyphens and gives entries without POS-tag the tag “W”.

$lex = $ARGV[0];
open(IN,$lex) || die "cannot open lexicon $lex";

while ($rule = <IN>) {

# entries with no pos-tag should get the tag W

       @parts1 = split(/\t/,$rule);
       $word = $parts1[0];
       $postag = $parts1[1];
       if ($postag eq "\n") {
              $postag = "W\n";
       }

# multi-word entries are removed: the entry should not contain white-spaces
       if ($rule !~ /.+(\s.+)+\t/) {

# suffix:
              if ($rule =~ /[a-zA-Z]0(\-.+)\t/) {
                                  print "$1\tsuff.\n";
                 }
# prefix:
                 elsif ($rule =~ /(.+\-)\t/) {
                                  print "$1\tpref.\n";
                 }
# hyphened entries (unknown number of parts):

              elsif ($rule =~ /(.+\-.+)\t(.+)/) {
                       @parts = split(/\-/,$1);
                     $ne = @parts;
                     $i = 0;
                     while ($i != $ne-1) {
                            print "$parts[$i]--";
                            $i++;
                     }
                     print "$parts[$i]\t$2\n";
                 }
# all other rules:

              else {
                     print "$word\t$postag";
              }
       }
}
```

3. The Perl program for executing the context-sensitive spell checking algorithm

```
#!/usr/local/bin/perl

use strict;

use DB_File;
use IPC::Open2;
use Data::Dumper;

my %counts;         # counts of trigrams
my %index;          # map from word to index number
my $pid;
my $debug_ipc_in = 0;
my $debug_ipc_out = 0;

open_databases();
start_conventional_speller();
main();

sub main
{
    my $border_prob = 0;

    # from trigram.pl (Olaf): from here...

    my @gram = ("", "", "");
    my @index = (0, 0, 0);

    while (my $line = <>) {
       chomp $line;
       $line =~ s/.*/\L$&/;       # lowercase the text
       $line =~ s/[^\w']/ /g;     # change all non-word characters to space
       my @words = split /\s+/, $line;

       #print "l: $line\n";

       while (@words) {           # loop over all words of a text line
           my $w = shift @words;
           next if not $w;        # skip empty words
           shift @gram;           # drop old word
           push @gram, $w;        # add in new word

           my $index = $index{$w} || 0; # map from word to index,
           shift @index;
           push @index, $index;
           my $trigram = pack "w3", @index;

    # ... to here
       if ($gram[0] ne "") {  #otherwise, first word and first two words are
incomplete trigrams
```

```
           my $orig_prob = $counts{$trigram} || 0; # trigram_prob(@gram);
           my $sugg_prob;

           if ((all_in_lex(@gram)) && ($orig_prob <= $border_prob)) {
              print "real-word error detected: $gram[0] $gram[1] $gram[2]\n";

              my @combinations = get_all_combinations(@gram);
              foreach my $sugg_gram (@combinations) {
                  $sugg_prob = trigram_prob(@$sugg_gram);
                  # print "@$sugg_gram -> $sugg_prob\n";
                  # The suggestion-trigram by definition occurs in the lexicon
                  # so we don't need to test that.
                  if (# (all_in_lex(@sugg_gram)) &&
                     ($sugg_prob > $orig_prob)) {
                          print "suggestion: $sugg_gram->[0] $sugg_gram->[1]
$sugg_gram->[2]\n";
                  }
              }
           }
        }
       }
    }

    #waitpid $pid, 0;
}

sub expect
{
    my $expect = shift;
    my $resp;
    #print "$expect\n";

    do {
       $resp = <RDRFH>;
       print "<-< ", $resp if $debug_ipc_in;
    } until $resp =~ /$expect/;
    print " - \n" if $debug_ipc_in;
}

sub all_in_lex
{
    my @trigram = @_;
    my $ok = 1;

    # Feed all 3 words to the speller and check if it complains.
     while ($ok && (my $w = shift @trigram)) {
       # feed only second word of trigram to the speller:
       # my $w = $trigram[1];

       print ">-> ", $w, "\n" if $debug_ipc_out;
       print WTRFH $w, "\n";
       my $resp = <RDRFH>;
```

```
       print "<-< ", $resp if $debug_ipc_in;

       next if ($resp =~ /Prompt/);
       # Look for Check returns 0[0:0]/1, @0:3 for 'bla'
       # where /1 is the error code for "unknown word".
       $ok = 0 if ($resp =~ /]\/1,/);

       expect(qr/Prompt/);
     }

    return $ok;
}

sub get_all_variations
{
    my @trigram = @_;
    my @all_variations = ();

    while (my $w = shift @trigram) {
    #find variations for second word only:
    #my $w = $trigram[1];

       print ">-> ", ",", $w, "\n" if $debug_ipc_out;
       print WTRFH ",",$w, "\n";
       my $resp;

       expect(qr/Suggestions:/);

       # Keep all variations, including original word, without duplicates.
       my %variations;
       $variations{$w} = 1;
       while (my $resp = <RDRFH>) {
           print "<-< ", $resp if $debug_ipc_in;
           chomp $resp;
           my @parts = split /\t/, $resp;
           last if @parts != 2;
           $variations{$parts[1]} = 1;
       }

       push @all_variations, [ sort keys %variations ];

       expect(qr/Prompt/);
    }
    # print "All variations ", Dumper(\@all_variations), "\n";

    return @all_variations;

}

sub get_all_combinations
{
    my @trigram = @_;
```

```
    my @combinations = ();

    my @variations = get_all_variations(@trigram);

    foreach my $w1 (@{$variations[0]}) {
       foreach my $w2 (@{$variations[1]}) {
          foreach my $w3 (@{$variations[2]}) {
              push @combinations, [ $w1, $w2, $w3 ];
       #      push @combinations, [ $trigram[0], $w1, $trigram[2] ];
          }
       }
    }

    # print "All combinations: ", Dumper(\@combinations), "\n";
    return @combinations;
}

sub trigram_prob
{
    my @trigram = @_;

    my @indices;

    foreach my $w (@_) {
       my $index = $index{$w} || 0;
       return 0 if !$index;

       push @indices, $index;
    }

    my $index = pack "w3", @indices;

    return $counts{$index} || 0;
}

sub open_databases
{
    my $filename = "db.trigrams";
    my $wordfile = "db.words";
    my $invwordfile = "$wordfile-inv";
    #$wordfile = undef;    # if undef, use words as keys directly (takes more
space)
    my $nextindex;

    my $hashinfo = new DB_File::HASHINFO;
    $hashinfo->{'cachesize'} = 30_000_000;     # Use a bigger cache than normal
- hope it helps. The size is in bytes.

    my $db = tie %counts, "DB_File", $filename, O_RDONLY, 0640, $hashinfo
       or die "Cannot open file '$filename': $!\n";

    my $indexdb;
```

```
    my $invindexdb;

    if (defined $wordfile) {
       # The words are mapped to an index number, and the index is then
       # used in the trigram-database. This is because the number will
       # be shorter than the word, therefore the trigram-database will
       # be smaller. Especially since each word will occur at least 3 times
       # as a key, for each possible position in a trigram. Many words
       # also occur in different contexts, which is even more duplication.

       # Note: for this juggling with the keys, we could have used a
       # DBM filter: filter_fetch_key and filter_store_key.

       my $hashinfo_smaller = new DB_File::HASHINFO;
       $hashinfo_smaller->{'cachesize'} = 3_000_000;  # Use a bigger cache than
normal - hope it helps.

       $indexdb = tie %index, "DB_File", $wordfile, O_RDONLY, 0640,
$hashinfo_smaller
           or die "Cannot open file '$wordfile': $!\n";

       # Also the next index number is stored, so the script can be run
       # incrementally.

       $nextindex = $index{'#'} || 1;
    }

}

sub start_conventional_speller
{
    $pid = open2(\*RDRFH, \*WTRFH, "./context", "context");
    if (!$pid) {
       die "Cannot start context\n";
    }

    expect(qr/Prompt/);
}
```

4. The Perl program for replacing words in a BNC text by real-word errors

```
#! usr/bin/perl

# replaces in a BNC text all right-hand words from the real-word error list by
# the left-hand corresponding word.

$text = $ARGV[0];
$list = "confulist.txt";
#$newtext = "> result.txt";
$wordfile = "> $text."."replacements.txt";
```

```
open(INT, $text) || die "cannot open text $text\n";
open(INL, $list) || die "cannot open list $list\n";
#open(OUTT, $newtext) || die "cannot open outfile\n";
open(OUTW, $wordfile) || die "cannot open word outfile\n";

@lefts = @rights = "";

while ($listline=<INL>) {
       @parts = split(/\t/,$listline);
       $left = $parts[0];
       $right = $parts[1];
       @lefts = (@lefts, $left);
       @rights = (@rights, $right);
       }

#for ($i=0;$i<100;$i++) {
#      print "$lefts[$i]\t$rights[$i]\n";
#}

$ne = @rights;
$count = 0;
while ($textline=<INT>) {
       @words = split(/\s/,$textline);
       foreach $word(@words) {
              foreach ($i=0;$i<$ne;$i++) {
                     if ($word eq $rights[$i]) {
                            print OUTW "$word replaced by $lefts[$i] in\n
$textline\n";
                            $count++;
                            $textline =~ s/\b$word\b/$lefts[$i]/;
                            # print $textline;
                     }
              }
       }
print $textline;
}
#print "number of replacements: $count\n";
```

5. The Perl program for compiling a list of erroneous trigrams

```
#! usr/local/bin/perl

# select from the original text and the text containing errors all trigrams
# that are different from each other. Output: a file containing two
# columns. Left: erroneous trigram. Right: original trigram.

$forig = "$ARGV[0]";
$ferror = "$ARGV[1]";

print "$fnorm\n";
```

```
open(IN1,$forig);
open(IN2,$ferror);

while ($rule1 = <IN1>) {

       chomp $rule1;
       # lowercase the text:
        $rule1 =~ s/.*/\L$&/;
        #change all nonword characters to space:
        $rule1 =~ s/[^\w']/ /g;
        #remove double spacing:
        $rule1 =~ s/[ ]{2,4}/ /g;
       $long1 .= $rule1;
}

#print "\n long: $long1\n";

@words1 = split(/\s+/,$long1);
#foreach $i (@words1) {
#      print "$i\n";
#}

while ($rule2 = <IN2>) {

        chomp $rule2;
       # lowercase the text:
        $rule2 =~ s/.*/\L$&/;
        #change all nonword characters to space:
        $rule2 =~ s/[^\w']/ /g;
        #remove double spacing:
        $rule2 =~ s/[ ]{2,4}/ /g;
       $long2 .= $rule2;
}

@words2 = split(/\s+/,$long2);

#print "\n";
$ne1 = @words1;
#print "$ne1\n";
$ne2 = @words2;
#print "$ne2\n";


for ($i=0;$i<=$ne1;$i++) {
       if ($words1[$i] ne $words2[$i]) {
              print "$words2[$i-2] $words2[$i-1] $words2[$i]\t$words1[$i-2]
$words1[$i-1] $words1[$i]\n";
              print "$words2[$i-1] $words2[$i] $words2[$i+1]\t$words1[$i-1]
$words1[$i] $words1[$i+1]\n";
              print "$words2[$i] $words2[$i+1] $words2[$i+2]\t$words1[$i]
$words1[$i+1] $words1[$i+2]\n";
```

```
	}
}
```

6. The Perl program for counting the average number of suggestions

```
#! usr/bin/perl

# Counts the average number of suggestions per detected trigram

$file = "total.result2";
open(IN,$file);

$count_detect = 0;
$count_suggest = 0;

while ($rule = <IN>) {
	if ($rule =~ /error detected/) {
		$count_detect++;
	}
	if ($rule =~ /suggestion\:/) {
		$count_suggest++;
	}
}
print "detected: $count_detect\n";
print "suggestions: $count_suggest\n";
$average =  $count_suggest/$count_detect;
print "average: $average\n";
close(IN);
```

7. The Perl program for evaluating the output files of the context-sensitive spell checker

```
#! user/bin/perl

# compares the list of corresponding erroneous and original trigrams with
# the output file from the speller in order to interpret the output of the
# context-sensitive spell checker.

$errortrigramlist = $ARGV[0];
$result = $ARGV[1];

open(IN1,$errortrigramlist) || die "cannot open errortrigramlist";

$nr_errgrams = -1; # (there is always one empty rule)

while ($rule1=<IN1>) {

	# lowercase the text of the trigramlist:
	$rule1 =~ s/.*/\L$&/;
	#change all nonword characters to space:
```

```
	$rule1 =~ s/[^\w^\n^\t']/ /g;
	#remove double spacing:
	$rule1 =~ s/[ ]{2,4}/ /g;
	#print $rule1;

	@parts1 = split (/\t/,$rule1);
	$errorgram = $parts1[0];
	$correctgram = $parts1[1];
	$trigramset{$errorgram} = $correctgram;
	$nr_errgrams++;
}

#printing the associative array:
#foreach $key (keys(%trigramset)) {
#	print "$key $trigramset{$key}";
#}

open(IN2,$result) || die "cannot open resultfile";

$nr_correct_sugg = 0;

while ($rule2=<IN2>) {
	if ($rule2 =~ /real-word error detected:/) {
		@parts2 = split (/\:\s/,$rule2);
		$detectedgram = $parts2[1];
		#print "detected trigram: $detectedgram";
		chop ($detectedgram);

		#foreach $key (keys(%trigramset)) {
		#	if ($key eq $detectedgram) {
		#		print "$detectedgram\t $key\n";
		#	}
		#}
		#print "correct trigram: $trigramset{$detectedgram}";

		# find the correct suggestion with a detected gram:

		$found_correct = 0;
		open(IN3,$result) || die "cannot open resultfile second time";
		while ($rule3=<IN3>) {
			if ($rule3 =~ /$detectedgram/) { # when the point of
detection is encountered...
				$nextrule = <IN3>;
				# ... walk through all suggestions below:
				   while ($nextrule =~ /suggestion\:/) {
				    #print $nextrule;
				    @partssugg = split(/\:\s/,$nextrule);
				    $sugggram = $partssugg[1];
				    chop ($sugggram);
				    #print "suggestion: $sugggram\n";
				    # check whether the suggestion is the original
trigram:
```

```
                        if ($trigramset{$detectedgram} eq "$sugggram\n") {
                               #print "correct suggestion: $sugggram\n";
                               $nr_correct_sugg++;
                               $found_correct = 1;
                        }
                        $nextrule = <IN3>;
                     }

                }
             }
             close(IN3);
             if ($found_correct == 0) { # then the correct suggestion was not
among the suggestions
                           # print "no correct suggestion found\n";
               }

      }
}

close(IN2);


open(IN4,$result) || die "cannot open resultfile";

#put all detected trigrams in an array:
$i=0;
while ($rule4=<IN4>) {
      if ($rule4 =~ /real-word error detected:/) {
             @parts4 = split (/\:\s/,$rule4);
             $detectedgram2 = $parts4[1];
            chop($detectedgram2);
             $detectedarray[$i] = $detectedgram2;
            $i++
      }
}

$nr_detected = @detectedarray;

close(IN4);

# compute the number of detected trigrams that is indeed
# erroneous for computing the precision... from here...

$nr_detected_err = 0;
$nr_false_hits = 0;

foreach $i (@detectedarray) {
      $false_hit = 1;
      foreach $error (keys(%trigramset)) {
             if ($i eq $error) {
                    #print "$i correctly detected\n";
                    $nr_detected_err++;
```

```
                        $false_hit = 0;
                }
        }
        if ($false_hit == 1) {
                print "false hit: $i\n";
                $nr_false_hits++;
        }
}

# ... to here

print "number of erroneous trigrams: $nr_errgrams\n"; #number of UNIQUE
erroneous trigrams
print "number of trigrams detected: $nr_detected\n";
print "number of correct suggestions: $nr_correct_sugg\n";
print "number of detected trigrams that is indeed erroneous:
$nr_detected_err\n";
print "number of false hits: $nr_false_hits\n";

$recall = $nr_detected_err / $nr_errgrams;
$recall_suggestions = $nr_correct_sugg / $nr_errgrams;
$precision = $nr_detected_err / $nr_detected;
printf "recall (nr of correctly detected / nr of erroneous trigrams) is
%4.2f\n", $recall;
printf "recall_suggestions (nr of correct suggestions / nr of erroneous
trigrams) is %4.2f\n", $recall_suggestions;
printf "precision (nr of correctly detected / nr of total detected) is
%4.2f\n", $precision;
print "if there are no false hits, precision is 1.00 and vice versa\n";
```

Appendix V – Format of output file from context-sensitive spell checker

```
real-word error detected: describe an number
suggestion: decline a number
suggestion: describe a number
suggestion: described a number
suggestion: describes a number
real-word error detected: teaching methods ad
suggestion: teaching method and
suggestion: teaching method as
suggestion: teaching methods a
suggestion: teaching methods an
suggestion: teaching methods and
suggestion: teaching methods as
real-word error detected: methods ad media
suggestion: methods and media
real-word error detected: ad media these
suggestion: and edit these
suggestion: and media these
suggestion: as media there
real-word error detected: these will by
suggestion: chest wall by
suggestion: chose well my
suggestion: here bill my
suggestion: here well my
suggestion: here will be
suggestion: house wall b
suggestion: house wall by
suggestion: house will be
suggestion: house will my
suggestion: thee will be
suggestion: theme will be
suggestion: there swill be
suggestion: there will be
suggestion: there will my
suggestion: these will be
suggestion: those will be
real-word error detected: will by considered
suggestion: well be considered
suggestion: will be conditioned
suggestion: will be considered
suggestion: will be positioned
```