The Design of a Proofreading Software Service

Raphael Mudge
Automattic
Washington, DC 20036
raffi@automattic.com

Abstract

Web applications have the opportunity to check spelling, style, and grammar using a software service architecture. A software service authoring aid can offer contextual spell checking, detect real word errors, and avoid poor grammar checker suggestions through the use of large language models. Here we present After the Deadline, an open source authoring aid, used in production on WordPress.com, a blogging platform with over ten million writers. We discuss the benefits of the software service environment and how it affected our choice of algorithms. We summarize our design principles as speed over accuracy, simplicity over complexity, and do what works.

1 Introduction

On the web, tools to check writing lag behind those offered on the desktop. No online word processing suite has a grammar checker yet. Few major web applications offer contextual spell checking. This is a shame because web applications have an opportunity to offer authoring aids that are a generation beyond the non-contextual spell-check most applications offer.

Here we present After the Deadline, a production software service that checks spelling, style, and grammar on WordPress.com[1], one of the most popular blogging platforms. Our system uses a software service architecture. In this paper we discuss how this system works, the trade-offs of the software service environment, and the benefits. We conclude with a discussion of our design principles: speed over accuracy, simplicity over complexity, and do what works.

1.1 What is a Software Service?

A software service (Turner et al., 2003) is an application that runs on a server. Client applications post the expected inputs to the server and receive the output as XML.

Our software service checks spelling, style, and grammar. A client connects to our server, posts the text, and receives the errors and suggestions as XML. Figure 1 shows this process. It is the client's responsibility to display the errors and present the suggestions to the user.

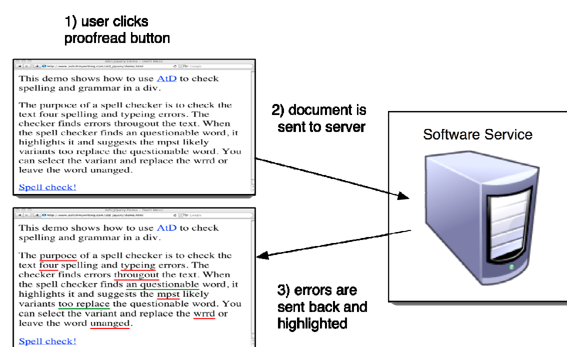

Figure 1. After the Deadline Client/Server Interaction.

[1] An After the Deadline add-on for the Firefox web browser is available. We also provide client libraries for embedding into other applications. See http://www.afterthedeadline.com.

Proceedings of the NAACL HLT 2010 Workshop on Computational Linguistics and Writing, pages 24–32,
Los Angeles, California, June 2010. ©2010 Association for Computational Linguistics

1.2 Applications

One could argue that web browsers should provide spell and grammar check features for their users. Internet Explorer, the most used browser (StatCounter, 2010), offers no checking. Firefox offers spell checking only. Apple's Safari web browser has non-contextual spell and grammar checking. Application developers should not wait for the browsers to catch up. Using a software service architecture, applications can provide the same quality checking to their users regardless of the client they connect with. This is especially relevant as more users begin to use web applications from mobile and tablet devices.

1.3 Benefits

A software service application has the advantage that it can use the complete CPU and memory resources of the server. Clients hoping to offer the same level of proofreading, without a software service, will use more resources on the local system to store and process the language models.

Our system uses large memory-resident language models to offer contextually relevant spelling suggestions, detect real word errors, and automatically find exceptions to our grammar rules.

On disk our language model for English is 165MB uncompressed, 32MB compressed. We use hash tables to allow constant time access to the language model data. In memory our English language model expands to 1GB of RAM. The memory footprint of our language model is too large for a web browser or a mobile client.

A software service also has maintenance advantages. The grammar rules and spell checker dictionary are maintained in one place. Updates to these immediately benefit all clients that use the service.

In this environment, users lose the ability to update their spell checker dictionary directly. To compensate, clients can offer users a way to always ignore errors. Our WordPress plugin allows users to ignore any error. Ignored errors are not highlighted in future checks.

1.4 Operating Requirements

A software service authoring aid must be able to respond to multiple clients using the service at the same time. Our service regularly processes over 100,000 requests a day on a single server.

Our goal is to process one thousand words per second under this load.

Since our system works in the web environment, it must process both text and HTML. We use a regular expression to remove HTML from text sent to the service.

It's important that our service report errors in a way that the client can locate them. The error phrase alone is not enough because suggestions may differ based on the context of the error.

We take a shortcut and provide clients with the text used to match the error and the word that precedes the error phrase. For example, for indefinite article errors, the text used to match the error is the misused article and the word following it. The client searches for this marker word followed by the error text to find the error and present the correct suggestions. This scheme is not perfect, but it simplifies our client and server implementations.

2 Language Model

Our system derives its smarts from observed language use. We construct our language model by counting the number of times we see each sequence of two words in a corpus of text. These sequences are known as bigrams. Our language model is case sensitive.

We trained our bigram language model using text from the Simple English edition of Wikipedia (Wikimedia, 2010), Project Gutenberg (Hart, 2008), and several blogs. We bootstrapped this process by using Wikipedia and Project Gutenberg data. We then evaluated the contents of several blogs looking for low occurrences of commonly misspelled words and real word errors. Blogs that had a low occurrence of errors were then added to our corpus. Our corpus has about 75 million words.

We also store counts for sequences of three words that end or begin with a potentially confused word. A potentially confused word is a word associated with a confusion set (see section 4.1). The real word error detector feature relies on these confusion sets. These counts are known as trigrams. We limit the number of trigrams stored to reduce the memory requirements.

2.1 Functions

Throughout this paper we will use the following functions to refer to our language model.

P(word): This function is the probability of a word. We divide the number of times the word occurs by the total number of words observed in our corpus to calculate the probability of a word.

$P(word_n, word_{n+1})$: This function is the probability of the sequence $word_n\ word_{n+1}$. We divide the number of times the sequence occurs by the total number of words observed in our corpus to calculate the probability of the sequence.

$Pn(word_n|word_{n-1})$: This function is the probability of a word given the previous word. We calculate this with the count of the $word_{n-1}\ word_n$ sequence divided by the count of the occurrences of $word_n$.

$Pp(word_n|word_{n+1})$: This function is the probability of a word given the next word. We use Bayes' Theorem to flip the conditional probability. We calculate this result as: $Pp(word_n|word_{n+1}) = Pn(word_{n+1}|word_n) * P(word_n) / P(word_{n+1})$.

$Pn(word_n|word_{n-1}, word_{n-2})$: This function is the probability of a word given the previous two words. The function is calculated as the count of the $word_{n-2}\ word_{n-1}\ word_n$ sequence divided by the count of the $word_{n-2}\ word_{n-1}$ sequence.

$Pn(word_{n+1}, word_{n+2}|word_n)$: is the probability of a sequence of two words given the word that precedes them. This is calculated as the count of $word_n\ word_{n+1}\ word_{n+2}$ sequence divided by the count of the occurrences of $word_n$.

$Pp(word_n|word_{n+1}, word_{n+2})$: This function is the probability of a word given the next two words. We calculate this result with $Pn(word_{n+1}, word_{n+2}|word_n) * P(word_n) / P(word_{n+1}, word_{n+2})$.

3 Spell Checking

Spell checkers scan a document word by word and follow a three-step process. The first step is to check if the word is in the spell checker's dictionary. If it is, then the word is spelled correctly. The second step is to generate a set of possible suggestions for the word. The final step is to sort these suggestions with the goal of placing the intended word in the first position.

3.1 The Spell Checker Dictionary

The dictionary size is a matter of balance. Too many words and misspelled words will go unnoticed. Too few words and the user will see more false positive suggestions.

We used public domain word-lists (Atkinson, 2008) to create a master word list to generate our spell checker dictionary. We added to this list by analyzing popular blogs for frequently occurring words that were missing from our dictionary. This analysis lets us include new words in our master word list of 760,211 words.

Our spell checker dictionary is the intersection of this master word list and words found in our corpus. We do this to prevent some misspelled words from making it into our spell checker dictionary.

We only allow words that pass a minimal count threshold into our dictionary. We adjust this threshold to keep our dictionary size around 125,000 words.

Threshold	Words	Present Words	Accuracy
1	161,879	233	87.9%
2	116,876	149	87.8%
3	95,910	104	88.0%
4	82,782	72	88.3%
5	73,628	59	88.6%

Table 1. Dictionary Inclusion Threshold.

Table 1 shows the effect of this threshold on the dictionary size, the number of present words from Wikipedia's List of Common Misspellings (Wikipedia, 2009), and the accuracy of a non-contextual version of our spell checker. We will refer to the Wikipedia Common Misspellings list as WPCM through the rest of this paper.

3.2 Generating Suggestions

To generate suggestions our system first considers all words within an edit distance of two. An edit is defined as inserting a letter, deleting a letter, substituting a letter, or transposing two letters (Damerau, 1964).

Consider the word *post*. Here are several words that are within one edit:

cost	substitute p, c	pose	substitute t, e
host	substitute p, h	posit	insert i
most	substitute p, m	posts	insert s
past	substitute o, a	pot	delete e
pest	substitute o, e	pots	transpose s, t
poet	substitute s, e	pout	substitute s, u

The naïve approach to finding words within one edit involves making all possible edits to the misspelled word using our edit operations. You may remove any words that are not in the dictionary to arrive at the final result. Apply the same algorithm to all word and non-word results within one edit of the misspelled word to find all words within two edits.

We store our dictionary as a Trie and generate edits by walking the Trie looking for words that are reachable in a specified number of edits. While this is faster than the naïve approach, generating suggestions is the slowest part of our spell checker. We cache these results in a global least-recently-used cache to mitigate this performance hit.

We find that an edit distance of two is sufficient as 97.3% of the typos in the WPCM list are two edits from the intended word. When no suggestions are available within two edits, we consider suggestions three edits from the typo. 99% of the typos from the WPCM list are within three edits. By doing this we avoid affecting the accuracy of the sorting step in a negative way and make it possible for the system to suggest the correct word for severe typos.

3.3 Sorting Suggestions

The sorting step relies on a score function that accepts a typo and suggestion as parameters. The perfect score function calculates the probability of a suggestion given the misspelled word (Brill and Moore, 2000).

We approximate our scoring function using a neural network. Our neural network is a multi-layer perceptron network, implemented as described in Chapter 4 of *Programming Collective Intelligence* (Segaran, 2007). We created a training data set for our spelling corrector by combining misspelled words from the WPCM list with random sentences from Wikipedia.

Our neural network sees each typo ($word_n$) and suggestion pair as several features with values ranging from 0.0 to 1.0. During training, the neural network is presented with examples of suggestions and typos with the expected score. From these examples the neural network converges on an approximation of our score function.

We use the following features to train a neural network to calculate our suggestion scoring function:

editDistance(suggestion, $word_n$)
firstLetterMatch(suggestion, $word_n$)
Pn(suggestion|$word_{n-1}$)
Pp(suggestion|$word_{n+1}$)
P(suggestion)

We calculate the edit distance using the Damerau–Levenshtein algorithm (Wagner and Fischer, 1974). This algorithm recognizes insertions, substitutions, deletions, and transpositions as a single edit. We normalize this value for the neural network by assigning 1.0 to an edit distance of 1 and 0.0 to any other edit distance. We do this to prevent the occasional introduction of a correct word with an edit distance of three from skewing the neural network.

The firstLetterMatch function returns 1.0 when the first letters of the suggestion and the typo match. This is based on the observation that most writers get the first letter correct when attempting to a spell a word. In the WPCM list, this is true for 96.0% of the mistakes. We later realized this corrector performed poorly for errors that swapped the first and second letter (e.g., oyu → you). We then updated this feature to return 1.0 if the first and second letters were swapped.

We also use the contextual fit of the suggestion from the language model. Both the previous and next word are used. Consider the following example:

```
The written wrd.
```

Here *wrd* is a typo for *word*. Now consider two suggestions *word* and *ward*. Both are an edit distance of one from *wrd*. Both words also have a first letter match. *Pp(ward|written)* is 0.00% while *Pp(word|written)* is 0.17%. Context makes the difference in this example.

3.4 Evaluation

To evaluate our spelling corrector we created two testing data sets. We used the typo and word pairs from the WPCM list merged with random sentences from our Project Gutenberg corpus. We also used the typo and word pairs from the ASpell data set (Atkinson, 2002) merged with sentences from the Project Gutenberg corpus.

We measure our accuracy with the method described in Deorowicz and Ciura (2005). For comparison we present their numbers for ASpell and several versions of Microsoft Word along with ours in Tables 2 and 3. We also show the number of misspelled words present in each system's spell checker dictionary.

	Present Words	Accuracy
ASpell (normal)	14	56.9%
MS Word 97	18	59.0%
MS Word 2000	20	62.6%
MS Word 2003	20	62.8%
After the Deadline	53	66.1%

Table 2. Corrector Accuracy: ASpell Data.

	Present Words	Accuracy
ASpell (normal)	44	84.7%
MS Word 97	31	89.0%
MS Word 2000	42	92.5%
MS Word 2003	41	92.6%
After the Deadline	143	92.7%

Table 3. Corrector Accuracy: WPCM Data.

The accuracy number measures both the suggestion generation and sorting steps. As with the referenced experiment, we excluded misspelled entries that existed in the spell checker dictionary. Note that the present words number from Table 1 differs from Table 3 as these experiments were carried out at different times in the development of our technology.

4 Real Word Errors

Spell checkers are unable to detect an error when a typo results in a word contained in the dictionary. These are called real word errors. A good overview of real word error detection and correction is Pedler (2007).

4.1 Confusion Sets

Our real word error detector checks 1,603 words, grouped into 741 confusion sets. A confusion set is two or more words that are often confused for each other (e.g., right and write). Our confusion sets were built by hand using a list of English homophones as a starting point.

4.2 Real Word Error Correction

The real word error detector scans the document finding words associated with a confusion set. For each of these words the real word error detector uses a score function to sort the confusion set. The score function approximates the likelihood of a word given the context. Any words that score higher than the current word are presented to the user as suggestions.

When determining an error, we bias heavily for precision at the expense of recall. We want users to trust the errors when they're presented.

We implement the score function as a neural network. We inserted errors into sentences from our Wikipedia corpus to create a training corpus. The neural network calculates the score function using:

$Pn(suggestion|word_{n-1})$
$Pp(suggestion|word_{n+1})$
$Pn(suggestion|word_{n-1}, word_{n-2})$
$Pp(suggestion|word_{n+1}, word_{n+2})$
$P(suggestion)$

With the neural network our software is able to consolidate the information from these statistical features. The neural network also gives us a back-off method, as the neural network will deal with situations that have trigrams and those that don't.

While using our system, we've found some words experience a higher false positive rate than others (e.g., to/too). Our approach is to remove these difficult-to-correct words from our confusion sets and use hand-made grammar rules to detect when they are misused.

4.3 Evaluation

We use the dyslexic spelling error corpus from Pedler's PhD thesis (2007) to evaluate the real word error correction ability of our system. 97.8%

of the 835 errors in this corpus are real-word errors.

Our method is to provide all sentences to each evaluated system, accept the first suggestion, and compare the corrected text to the expected answers. For comparison we present numbers for Microsoft Word 2007 Windows, Microsoft Word 2008 on MacOS X, and the MacOS X 10.6 built-in grammar and spell checker. Table 4 shows the results.

Microsoft Word 2008 and the MacOS X built-in proofreading tools do not have the benefit of a statistical technique for real-word error detection. Microsoft Word 2007 has a contextual spell-checking feature.

	Precision	Recall
MS Word 07 - Win	90.0%	40.8%
After the Deadline	89.4%	27.1%
MS Word 08 - Mac	79.7%	17.7%
MacOS X built-in	88.5%	9.3%

Table 4. Real Word Error Correction Performance.

Most grammar checkers (including After the Deadline) use grammar rules to detect common real-word errors (e.g., a/an). Table 4 shows the systems with statistical real-word error correctors are advantageous to users. These systems correct far more errors than those that only rely on a rule-based grammar checker.

5 Grammar and Style Checking

The grammar and style checker works with phrases. Our rule-based grammar checker finds verb and determiner agreement errors, locates some missing prepositions, and flags plural phrases that should indicate possession. The grammar checker also adds to the real-word error detection, using a rule-based approach to detect misused words. The style checker points out complex expressions, redundant phrases, clichés, double negatives, and it flags passive voice and hidden verbs.

Our system prepares text for grammar checking by segmenting the raw text into sentences and words. Each word is tagged with its relevant part-of-speech (adjective, noun, verb, etc.). The system then applies several grammar and style rules to this marked up text looking for matches. Grammar rules consist of regular expressions that match on parts-of-speech, word patterns, and sentence begin and end markers.

Our grammar checker does not do a deep parse of the sentence. This prevents us from writing rules that reference the sentence subject, verb, and object directly. In practice this means we're unable to rewrite passive voice for users and create general rules to catch many subject-verb agreement errors.

Functionally, our grammar and style checker is similar to Language Tool (Naber, 2003) with the exception that it uses the language model to filter suggestions that don't fit the context of the text they replace, similar to work from Microsoft Research (Gamon, et al 2008).

5.1 Text Segmentation

Our text segmentation function uses a rule-based approach similar to Yona (2002) to split raw text into paragraphs, sentences, and words. The segmentation is good enough for most purposes.

Because our sentence segmentation is wrong at times, we do not notify a user when they fail to capitalize the first word in a sentence.

5.2 Part-of-Speech Tagger

A tagger labels each word with its relevant part-of-speech. These labels are called tags. A tag is a hint about the grammatical category of the word. Such tagging allows grammar and style rules to reference all nouns or all verbs rather than having to account for individual words. Our system uses the Penn Tagset (Marcus et al, 1993).

```
The/DT little/JJ dog/NN
laughed/VBD
```

Here we have tagged the sentence *The little dog laughed*. *The* is labeled as a determiner, *little* is an adjective, *dog* is a noun, and *laughed* is a past tense verb.

We can reference little, large, and mean laughing dogs with the pattern *The .*/JJ dog laughed*. Our grammar checker separates phrases and tags with a forward slash character. This is a common convention.

The part-of-speech tagger uses a mixed statistical and rule-based approach. If a word is known and has tags associated with it, the tagger tries to

find the tag that maximizes the following probability:

$$P(tag_n|word_n) * P(tag_n|tag_{n-1}, tag_{n-2})$$

For words that are not known, an alternate model containing tag probabilities based on word endings is consulted. This alternate model uses the last three letters of the word. Again the goal is to maximize this probability.

We apply rules from Brill's tagger (Brill, 1995) to fix some cases of known incorrect tagging. Table 5 compares our tagger accuracy for known and unknown words to a probabilistic tagger that maximizes $P(tag_n|word_n)$ only.

Tagger	Known	Unknown
Probability Tagger	91.9%	72.9%
Trigram Tagger	94.0%	76.7%

Table 5. POS Tagger Accuracy.

To train the tagger we created training and testing data sets by running the Stanford POS tagger (Toutanova and Manning, 2000) against the Wikipedia and Project Gutenberg corpus data.

5.3 Rule Engine

It helps to think of a grammar checker as a language for describing phrases. Phrases that match a grammar rule return suggestions that are transforms of the matched phrase.

Some rules are simple string substitutions (e.g., utilized → used). Others are more complex. Consider the following phrase:

```
I wonder if this is your companies way of providing support?
```

This phrase contains an error. The word companies should be possessive not plural. To create a rule to find this error, we first look at how our system sees it:

```
I/PRP wonder/VBP if/IN
this/DT is/VBZ your/PRP$ companies/NNS way/NN of/IN providing/VBG support/NN
```

A rule to capture this error is:

```
your .*/NNS .*/NN
```

This rule looks for a phrase that begins with the word *your*, followed by a plural noun, followed by another noun. When this rule matches a phrase, suggestions are generated using a template specified with the rule. The suggestion for this rule is:

```
your \1:possessive \2
```

Suggestions may reference matched words with *n*, where *n* is the *n*th word starting from zero. This suggestion references the second and third words. It also specifies that the second word should be transformed to possessive form. Our system converts the plural word to a possessive form using the *\1:possessive* transform.

Phrase	Score
your companies way	0.000004%
your company's way	0.000030%

Table 6. Grammar Checker Statistical Filtering.

Before presenting suggestions to the user, our system queries the language model to decide which suggestions fit in the context of the original text.

Rules may specify which context fit function they want to use. The default context fit function is: $Pn(word_n|word_{n-1}) + Pp(word_n|word_{n+1}) > (0.5 \times [Pn(word_n|word_{n-1}) + Pp(word_n|word_{n+1})]) + 0.00001$.

This simple context fit function gets rid of many suggestions. Table 6 shows the scores from our example. Here we see that the suggestion scores nearly ten times higher than the original text.

This statistical filtering is helpful as it relieves the rule developer from the burden of finding exceptions to the rule. Consider the rules to identify the wrong indefinite article:

```
a [aeiouyhAEIOUYH18]\w+
an [^aeiAEIMNRSX8]\w+
```

One uses *a* when the next word has a consonant sound and *an* when it has a vowel sound. Writing rules to capture this is wrought with exceptions. A rule can't capture a sound without hard coding each exception. For this situation we use a context

fit function that calculates the statistical fit of the indefinite article with the following word. This saves us from having to manually find exceptions.

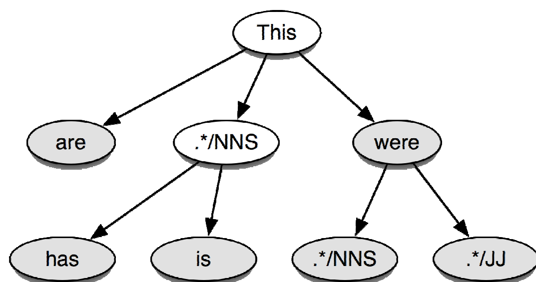

Figure 2. Rule Tree Example.

Each rule describes a phrase one word and tag pattern at a time. For performance reasons, the first token must be a word or part-of-speech tag. No pattern matching is allowed in the first token. We group rules with a common first word or tag into an *n*-ary rule tree. Rules with common pattern elements are grouped together until the word/tag patterns described by the rule diverges from existing patterns. Figure 2 illustrates this.

When evaluating text, our system checks if there is a rule tree associated with the current word or tag. If there is, our system walks the tree looking for the deepest match. Each shaded node in Figure 2 represents a potential match. Associated with each node are suggestions and hints for the statistical checker.

We measure the number of rules in our system by counting the number of nodes that result in a grammar rule match. Figure 2 represents six different grammar rules. Our system has 33,732 rules to check for grammar and style errors.

The capabilities of the grammar checker are limited by our imagination and ability to create new rules. We do not present the precision and recall of the grammar checker, as the coverage of our hand-made rules is not the subject of this paper.

6 Conclusions

Our approach to developing a software service proofreader is summarized with the following principles:

- Speed over accuracy
- Simplicity over complexity
- Do what works

In natural language processing there are many opportunities to choose speed over accuracy. For example, when tagging a sentence one can use a Hidden Markov Model tagger or a simple trigram tagger. In these instances we made the choice to trade accuracy for speed.

When implementing the smarts of our system, we've opted to use simpler algorithms and focus on acquiring more data and increasing the quality of data our system learns from. As others have pointed out (Banko and Brill, 2001), with enough data the complex algorithms with their tricks cease to have an advantage over the simpler methods.

Our real-word error detector is an example of simplicity over complexity. With our simple trigram language model, we were able to correct nearly a quarter of the errors in the dyslexic writer corpus. We could improve the performance of our real-word error corrector simply by adding more confusion sets.

We define "do what works" as favoring mixed strategies for finding and correcting errors. We use both statistical and rule-based methods to detect real word errors and correct grammar mistakes.

Here we've shown a production software service system used for proofreading documents. While designing this system for production we've noted several areas of improvement. We've explained how we implemented a comprehensive proofreading solution using a simple language model and a few neural networks. We've also shown that there are advantages to a software service from the use of large language models.

After the Deadline is available under the GNU General Public License. The code and models are available at http://open.afterthedeadline.com.

Acknowledgements

The author would like to acknowledge the review committee for their questions and suggestions. The author would also like to acknowledge Nikolay Bachiyski, Michael Yoshitaka Erlewine, and Dr. Charles Wallace who offered comments on drafts of this paper.

References

Kevin Atkinson. 2008, Kevin's Wordlist Page. http://wordlist.sourceforge.net/, last accessed: 4 April 2010.

Kevin Atkinson, Spellchecker Test Kernel Results. 2002. http://aspell.net/test/orig/, last accessed: 28 February 2010.

Michele Banko and Eric Brill. 2001. Scaling to very very large corpora for natural language disambiguation. *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics and the 10th Conference of the European Chapter of the Association for Computational Linguistics*, Toulouse.

Eric Brill. 1995. Transformation-based error-driven learning and natural language processing: a case study in part of speech tagging. *Computational Linguistics*, 21:543–565.

Eric Brill and Robert C. Moore. 2000. An improved error model for noisy channel spelling correction. *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics*, Hong Kong, pp. 286–293.

Fred J. Damerau. 1964. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3): 659-664.

Sebastian Deorowicz and Marcin G. Ciura. 2005. Correcting spelling errors by modelling their causes. *International Journal of Applied Mathematics and Computer Science*, 15(2):275–285.

Michael Gamon, Jianfeng Gao, Chris Brockett, Alexander Klementiev, William Dolan, Dmitriy Belenko, and Lucy Vanderwende. 2008. Using Contextual Speller Techniques and Language Modeling for ESL Error Correction. *Proceedings of IJCNLP*, Hyderabad, India, Asia Federation of Natural Language Processing.

Michael Hart. 2008. *Project Gutenberg*. http://www.gutenberg.org/, last accessed: 28 February 2010.

Abby Levenberg. 2007. *Bloom filter and lossy dictionary based language models*. Master of Science Dissertation, School of Informatics, University of Edinburgh.

Mitchell Marcus, Beatrice Santorini, and Maryann Marcinkiewicz. 1993. Building a large annotated corpus of English: the Penn Treebank. *Computational Linguistics*, 19(2).

Daniel Naber. 2003. *A Rule-Based Style and Grammar Checker*. Diplomarbeit Technis Fakultät, Universität Bielefeld, Germany.

Jennifer Pedler. 2007. *Computer Correction of Real-word Spelling Errors in Dyslexic Text*. PhD thesis, Birkbeck, London University.

Segaran, T. 2007 *Programming Collective Intelligence*. First. O'Reilly. pp. 74-85

StatCounter, 2010. *Top 5 Browsers from Feb 09 to Mar 10*. http://gs.statcounter.com/, last accessed: 28 February 2010.

Kristina Toutanova and Christopher D. Manning. 2000. Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger. *Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora* (EMNLP/VLC-2000), pp. 63-70.

Mark Turner, David Budgen, and Pearl Brereton. 2003. Turning software into a service. *Computer*, 36(10):38–44.

Robert A. Wagner and Michael J. Fischer. 1974. The string-to-string correction problem. *Journal of ACM*, 21(1):168–173.

Wikipedia, 2009. *List of Common Misspellings*. http://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings, last accessed: 28 February 2010.

Wikimedia Inc. 2010. *Wikimedia Downloads*. http://download.wikipedia.org/, last accessed: 28 February 2010.

Shloma Yona, 2002. *Lingua::EN::Sentence Module*, CPAN. http://search.cpan.org/~shlomoy/Lingua-EN-Sentence-0.25/lib/Lingua/EN/Sentence.pm, last accessed: 28 February 2010.