A semantic proofreading tool for all languages based on a text repository[1]

Kai A. Olsen
Molde University College and Department of Informatics, University of Bergen
Norway
kai.olsen@himolde.no

Bård Indredavik
Technical Manager, Oshaug Metall AS, Molde, Norway
bard@oshaug.no

Abstract

A method for finding lexical, syntactic or semantic errors in text in any language is introduced. It is based on a large text repository. The idea is to "follow everybody else", that is, to compare the sentence offered by the user to the similar sentences in the text repository and suggesting alternative words when appropriate. This concept offers the possibility of taking proofreading further than what is possible with standard spell- and grammar checkers.

A prototype for such a system has been developed. This includes a spider that traverses the Web and stores the retrieved text; a builder that creates a convenient index structure for the text repository and the part that analyses the user's sentence, offering suggestions for improvement.

1 Introduction

Natural language text can be viewed at different levels. At the bottom level we have the characters and symbols, perhaps coded as ASCII or Unicode. On the next level we put these characters together to make words. Then words are combined into sentences, sentences into paragraphs and paragraphs into complete documents. In philology one would also view the text on different levels, i.e., as lexical (characters and words), syntactic (grammar), semantic (meaning) or pragmatic (the discussion of meaning, or the meaning of the meaning).

Text has been an ideal application for computers. The bottom level, the characters, is clearly formalized. This opens for automatic processing. At the same time one can make use of the computer's ability to handle large amounts of data, here large numbers of characters. With a word processor users can enjoy the freedom of being able to edit the text, and at any time have a clear, "print" version of the document on the screen. Computer networks make it easy to disseminate documents in electronic form; better displays make it possible to avoid printing and to read books and other documents directly from the screen; and with improved storage techniques we can build huge repositories of documents.

In this paper we shall study how these text repositories can be used for proofreading. That is, to aid users in producing sentences with fewer lexical or syntactic errors, and also, aid in improving the semantic part.

---

[1] This paper was presented at the NIK-2010 conference. For more information see www.nik.no

2 Spell- and grammar checker

For each level of text we have a set of tools. At the bottom level a simple editor as Notepad can be used to enter and edit a sequence of symbols. It the simplest form the text is viewed as a character sequence only, a string, without any notion of words or sentences. A modern word processor, however, embodies a higher level concept of a document, including terms as word, sentence, paragraph and section. It can then avoid splitting words at the end of a line, capitalizing the first word in a sentence, using larger fonts for headings, avoid headings at the bottom of a page, etc. With a dictionary the system can indicate unknown words, i.e., words not found in the vocabulary. It may be a name, a word unknown to the system or, most often, a typographical error or spelling mistake. Of course, typos that results in a correct word, but not the word the author intended are not found. For example, if "I have a car" becomes "I have a far" a standard spell checker will not find the error.

On the next level a grammar checker is employed. In their simplest form they look for missing punctuation, double words, and sentences that start in lower case. They use pattern matching to indicate clichés and other forms of poor writing. More advanced systems parse each sentence using a dictionary and compare the results to a set of grammatical rules. For example, it will detect errors in sentence such as these "I is a man", "an language", "he had two car", but will often not find grammatical errors in more complex sentences. For example, the English grammar checker in Microsoft Word 2007 will find the error in "Apples and oranges is used as dessert." However, the sentence "Apples and oranges, the latter served in a bowl, is used as dessert." passes without comment.

While a spelling checker is valuable for catching typographical errors; and while both spelling and grammar checkers are important tools for poor writers or when writing in a foreign language they only do a part of the proofreading. They do not, and can not, aid in correcting errors in the semantic parts. Thus the sentence "I have a far" stands uncorrected, just as "A forest has many threes" and "They lived at London". To be able to find these errors we need smarter systems, i.e., systems that can detect semantic errors.

3 Semantic error detection

There have been many attempts to put "intelligence" into computer systems. The first chess playing programs was based on such an approach. Later, efforts were made to give computers expert knowledge, for example in medicine. Some of these applications were successful within a formalized environment (see for example Buchanan and Shortliffe, 1984). A medical expert system could diagnose a patient based on test results with a high degree of accuracy. In the real world, however, these systems fail in most normal cases. A human doctor can achieve better results faster by a combination of examination, tests and experience.

Language is similar. The dictionaries and grammatical rules only take us a short way. There will always be errors that are not detected and correct phrases that are marked as errors by automatic systems. The meaning of a sentence can be hidden within the words and will often require some world knowledge to be understood. Hubert Dreyfus (1972, 1992) said that we all know a chair when we see it, but it is not easy to define. Many of our everyday concepts are similar; it is not easy to give a formal definition of concepts such as dinners, friends, and jokes. Poets utilize the deeper meanings of words, also their rhythm when they are pronounced to tell the reader something more than what is in

each word. However, most users do not need a system to analyze poems. What many need is a system that can take typos such as "I have a far", spelling mistakes such as "We had ice cream for desert" or correcting prepositions in "we live at the West Coast." This is especially the case when we write in a foreign language. Not only do we make more mistakes than writing in our native language, but we also make mistakes that should be easy to correct, such as mixing words, wrong prepositions, etc. Improved spelling and grammar checkers may take some of these errors, but not all.

Perhaps we do not have to program computers to *understand* language. Another approach is to use computers to *compare*. That is, to compare what we have written to what everybody else writes. Googles use this method, a statistical approach, for their translator. In addition to dictionaries and grammatical rules it also has huge text repositories that can be applied to the task. For example, within the European Union the same document may exist in many different languages, translated by humans. This offers Google a base for comparison: This sentence in German has this counterpart in English, Italian, French, etc. The term collective intelligence is often used to describe these methods (Segaran, 2007).

Machine translation, also translation based on statistical methods, works exceptional well on simple texts and exceptional bad on more complex texts. This is what we should expect. In simple texts many of the phrases follow syntactical rules, there is a high probability that the phrases may be found in the text repository and also that there exist translations to other languages. In the more complex texts it will be more difficult to find useful counterparts in the repository. Hidden bindings or references between words may also make the sentences difficult to parse.

Google also makes use of their text repository when they suggest "Did you mean", that is, based on data from other searches and from the text repository they will suggest an improvement of a user query. For example, if we misspell "Pitsburg" they will ask "did you mean: Pittsburgh?" In situations where they are quite sure that the user has made an error they will show results for the improved query directly without asking the user. Thus "rok and rol" will result in Google showing results for "rock and roll."

Without touching the "meaning" of a sentence this approach allow us to find semantic errors. The idea is, of course, based on the supposition that the majority is correct. That is, if a majority of the sentences in the text repository have "Pittsburgh" and not "Pitsburg", only a few mentions the latter, and based on the similarity between words one can assume that the user really meant "Pittsburgh". Feedback from users can tell if we were correct. If most users that got this "Did you really mean Pittsburgh"-question ended up by accepting the suggestion, we know that we are on the right way. After a while we can skip the "Do you really mean" and just explain that we offer search results for Pittsburgh, as in the "rok and rol" example.

This idea of following the majority is not new. Take the idea of a path in the mountains. When following a path we exploit the cumulative knowledge and experience of all individuals that have been walking this way. The path avoids steep cliffs, leads us to a ford at the river, and will follow an effective and not too strenuous route. If one person takes off from the path no impression will be made on the terrain. If many go in this new direction, a branch to the path can appear. The system we describe here works in the same manner. Modern computer systems with fast processors and huge data storage allow us to exploit the idea in a language-based context. Using statistics we will follow the main path, or perhaps suggest an alternative (branch) if this occurs frequently (the impression is strong enough).

4 The semantic parser

The idea of using text repositories for semantic parsing and proofreading was introduced in Olsen and Williams, 2004. The concept was then to test each alternative using a search engine. Thus one could try "we live at [on/in] the West Coast" or "we had ice cream for desert [dessert]", noting the number of search results in each case. With the apostrophes we ensure that the search engine looks at the whole phrase, and not only for single keywords. The authors compare this method to asking a friend native to the language what alternative to use; just that in the Web-based case you could ask millions instead of one. While this is a useful method in many cases the clear disadvantage is that we will have to know the alternatives. It is therefore a method for the more fluent writers that use this method to check a detail, which preposition to use, which phrase is most common, etc.

We can find the alternatives by inserting a wild-card (most often represented by an asterisk) in the query. This symbol indicates any word, e.g., "We live * the West Coast". By looking at the results we could see that the prepositions *at*, *in*, *on*, and *by* are used here. Now we have the alternatives, and we can proceed as above – checking the frequency of each. While quite cumbersome, this method would work. But we still have the problem that the user must suspect a certain word in the sentence. Many write "we had ice cream for desert" and are quite happy with their spelling.

The solution is a tool that finds the set of alternatives for each word in a sentence, and which retrieves the frequency for each alternative. We have a possible mistake where the users alternative (e.g., desert) have a low score and another (e.g., dessert) have a high score in a given sentence. In this case a suggestion for replacing the one with the other will also be strengthened by the similarity of the two words.

What we present here is a tool based on these principles. In the prototype version the user can type in a sentence on a Web page, and get suggestions for improving her text. Some examples are shown in Table 1 below.

User's sentence	Suggested sentence	Confidence
We had ice cream for desert.	We had ice cream for dessert.	Strong
A red far.	A red car.	Medium
	A red farm.	Medium
She lived on London.	She lived in London.	Strong
Håkonshallen ligger på Bergen.	Håkonshallen ligger i Bergen.	Strong
Tickets were reserved.	Tickets were booked.	Medium
Mount Everest is 8000 meter high.	Mount Everest is 8848 meter high.	Strong
Bergen is the capital of Norway.	Oslo is the capital of Norway.	Strong

Table 1. Examples of system suggested improvements.

As one sees the examples cover any type of blunder. For the two last examples one could also use the "Wiki answer" features found on the Web, or, of course, just use a search engine to get the answer. The advantage with our approach is that this is embedded into the proofreading automatically. The "confidence" attribute is based on a simple comparison of the frequency between the suggested and the original variant.

Note that the system is language independent. This is indicated in Table 1 where we find examples both in English and Norwegian. While one could add all texts, in any

language, to the same text repository it is more practical to have a database for each language. This enables the system to run faster, and one avoids collisions where there are similarities between languages. Google, for example, will often suggest a did-you-mean word in English, when we search in Norwegian. We will therefore let the spider follow links with ".no" endings for building a Norwegian text repository, ".es" for Spanish, etc. We expect that links with ".com", ".edu" and ".org" endings direct to pages in English. However, it is quite easy to detect language based on the occurrence of certain keywords in the document. This is especially the case here, where the consequences of making a wrong decision are minor.

5   Prototype implementation

A prototype of the system has been implemented. Initially we used the Google search engine to test our ideas. Then we build a complete system consisting of three parts, a spider or crawler that traverses the Web and build a text repository, a builder that creates an efficient index structure and an analyzing part that looks for alternative words in a sentence offered by the user, offering suggestions for improvements if found.

5.1   Prototype using Google

Figure 1. Using Google to test sentences.

Our very first version was just a front end to Google, to test the concept. We entered phrases or complete sentences and noted the number of results, as seen in Figure 1. Here we only got 191 results. The system then automatically replaced one and one word by an asterisk and parsed the text to find the most frequent alternatives for each wild card. Each alternative was then inserted in the sentence, another search was made, and the number of results was registered.

Figure 2. Replacing "desert" by a wild card (*).

In Figure 2 we see that "dessert" is an alternative for the wild card. Using this word instead of "desert" in the query we go from 191 to 4,860 results, as seen in Figure 3.

Figure 3. 4,860 results when replacing "desert" with "dessert".

We can therefore, with a high confidence, suggest that the user replace "desert" by "dessert". However, Google block automatic searches. There are also other limitations when using Google, for example that punctuation may be removed. We therefore had to build our own text repository. The advantages are that we have full control; can handle punctuation and also speed up the process by running all searches on our own server. In later versions it will then also be possible to integrate the proofreading system with a word processor.

To build our own text repository we need a spider or a crawler, a program that traverses the Web, parsing text, and that follows all URL's that it encounter.

5.2 The spider

The spider is offered a list of URL's – the seed. In this list we include links to home pages of universities, newspapers and government institutions. The spider traverses the list, retrieve a page at a time, parse the HTML, register all new URL's and clean the text. Care is taken to spread the load to different sites. URL's are stored in files, one for HTML-links, another for pdf-links and a third for doc-links (links to Microsoft Word files, .doc and .docx). In the cleaning process formatting information is removed, HTML-codes are replaced by their respective symbols and other types of noise are removed. The cleaned text is stored in simple ASCII-files (.txt), with approx one megabyte in each file. Files are numbered consecutively.

Separate PDF- and Doc-parsers retrieve links of each type, load the pages, clean the text and store the results in one megabyte text files. While there are few doc-files there are many pdf. Both file types have the advantage that they often include large amounts of text, for example as books, reports and other documents. Also, text from these sources may have a higher quality with regard to spelling, grammar and semantics than text represented as HTML.

A hash table is used to remember links that have already been traversed. A simple formula will hash each URL and store the resulting value (a long integer) in this table. The spider will, of course, only follow URL's that have not been encountered before. In the prototype setup the spider, html-, pdf- and doc-parsers runs on a multi processor PC. With this implementation we manage to retrieve approx one megabyte of text every few minutes. This value is, of course, dependent on the response times of the servers hosting the different URL's.
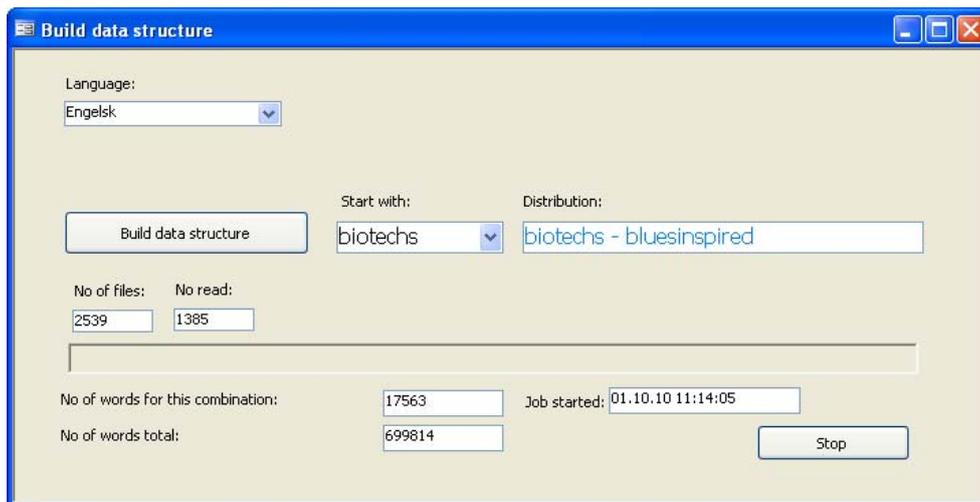
5.3 Building the index structure

Figure 4. Building the index structure

A program (Figure 4) will traverse the text files and build an elaborate index structure. For each word a reference to the files where this word occurs will be stored. In addition, for each file and each word a reference to the lines (or sentences) where the word occurs will also be saved. The idea is that we, when analyzing a sentence offered by the user, will only have to parse the files where words in the query occur in the same lines.

The build operation is memory demanding and the operation is therefore performed in many iterations, looking at only a part of the alphabet in each. This index is build once, but can be updated whenever there is more source text to consider. It is stored as simple .txt files.

5.4 Analyzer

In the prototype version the user can enter a sentence or a phrase at a time. The analyzing part of the system will find file references for each word from the index. For each file and word the line-information (the line or sentence numbers holding the word) is retrieved. Only files where the words occur in the same line are then parsed.

As a first step we find the frequency of the complete sentence offered by the user. Then the system will examine each part of the sentence, replacing each word with a wild card. With N words in the sentence we will now get the frequency for each sequence of $N-1$ words. That is, due to the wild cards we must also find alternatives for each word. For example, with the sentence "We had ice cream for desert", we have the following variants:

- we, had, ice, cream, for, desert
- *, had, ice, cream, for, desert
- we, *, ice, cream, for, desert
- we, had, *, cream, for, desert
- we, had, ice, *, for, desert
- we, had, ice, cream, *, desert
- we, had, ice, cream, for, *

For each variant the system will find the files where the words occur in the same line. For example, in the case above we will retrieve all files that have at least five of the six

words in the same line, parse these looking for the different word combinations and recording the words representing the wild cards. For the last word combination this will probably give us the word "dessert".

The decision-part of the system will now compute a simple score, based on the frequency for each alternative compared to the frequency of the original statement. If the score of an alternative sentence is higher than the score of the original, a suggestion for a replacement is made. If the difference between scores is high we will classify this improvement with a "high" confidence level, else "medium" or "low".

Since many mistakes are made by mixing similar words, such as "desert" and "dessert", the score is also based on the similarity between words. Similarity is defined as the edit distance from the one word to the other, i.e., the number of steps (changing or moving a character at the time) one would have to perform to change the first word into the second. We use Hirschberg's algorithm (Hirschberg, 1975), adapted to the edit-distance problem, and a variation introduced by Powell et al (1999).

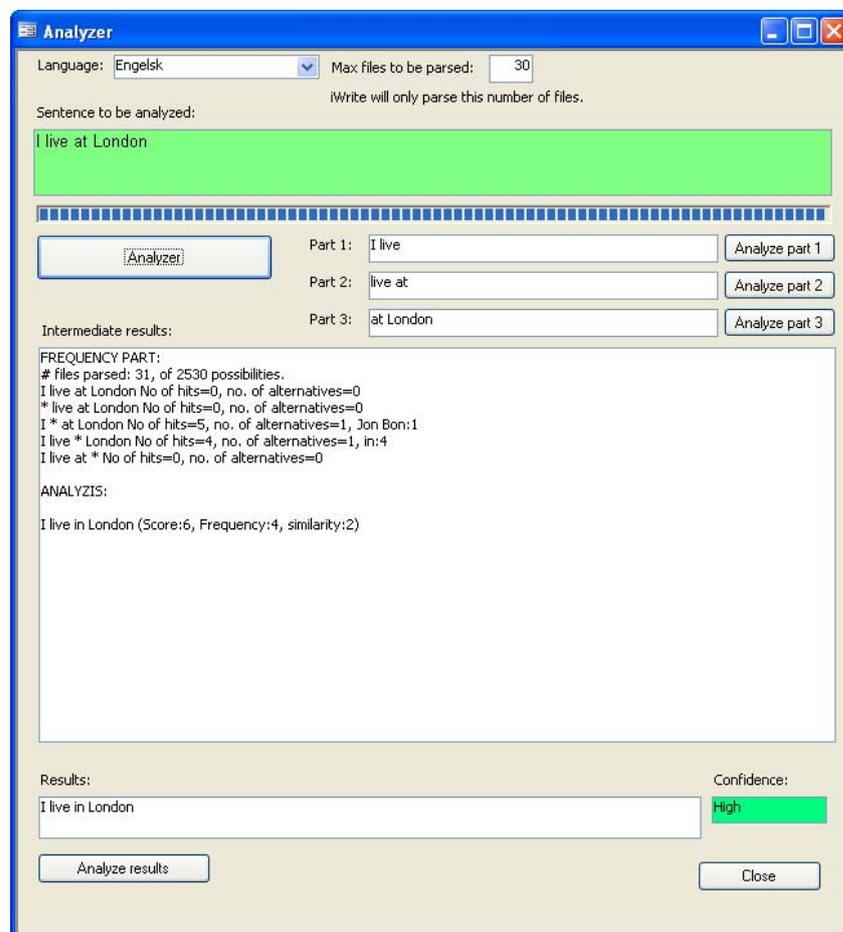## 5.5 The prototype user interface

Figure 5. The user interface

The user interface of the prototype is presented in Figure 5, here with an example in English. The user has typed the sentence "I live at London", and the result "I live in London" is presented. As seen, the system has replaced the wrong preposition ("at") with a correct ("in"). In this case the confidence level is marked as high since the correct sentence has a relatively high score, the original a zero score.

The user have the possibility to analyze parts of the sentence (the options on the top). Intermediate results are shown in the prototype version, both the frequency data and the background of the detailed analyzes that the system has performed.

5.6 Prototype performance

A clear limitation of using the Web to create a text repository is that this is a very time consuming way of retrieving text. The problem is on the server side, it takes some time to download the contents of a page from a remote server. In practice this limited the size of repositories to a few gigabytes, which we used weeks to collect. This is not large enough to get a good sample of all possible sentences. For example, while getting hits on "I live * London" we found no samples of "I live * Oslo" in an English repository of 2.5 gigabytes. In a Norwegian one gigabyte repository we found "Jeg bor * Oslo" (I live * Oslo), but not "Jeg bor * Bergen".

Another drawback is the linearly process to build the index structure. Since it is dependent on holding at least parts of the index in memory when building the structure we had to make many passes through the data files, just looking at a part of the alphabet in each pass. For example, the system used ten days to build a complete index structure for 2,500 one megabyte text files.

Based on the users query the analyzer will parse all the files where the words in the sentence occur in the same line. Since we must include wild cards the files that have at least *N-1* of the words from the user's query in the same line are parsed. For queries with very few words, and common words only, there may be many files to parse. With a maximum of 30 files to parse, as in the example in Figure 5, we have limited maximum processing time to one minute. However, queries with at least two rare words will be analyzed in just a few seconds.

Thus, while we have a working prototype that can correct many types of syntactic and semantic errors it is clear that a product version would have to use a different approach than the linear model that is applied here. Clearly the solution is to use parallel processing and a large number of computers, i.e., the same techniques that are used by Google, Yahoo and others.

6 A product version – going parallel

In a product version there are many options for letting this process go in parallel. Now, parallel processing involves complex algorithms, a scheme for data distribution and for handling failures, all of which obscure the logic of the computation. As an alternative, the MapReduce method (Dean and Ghemawat, 2004, 2010) seems a promising candidate.

The *map* operation is applied to each record and produces a set of intermediate key/value pairs. For example, in an application to count the number of words in a document collection the operation can map from documents to a list of value pairs, each a word and the number of occurrences of this word in the document. The *reduce* operation is then applied to all pairs that share the same key, giving us the total frequency for each word in the collection. The advantage of this set up is that the operations can be performed on large sets of simple computers, also that the "parallelism" is hidden from the application developer.

In our case we could choose to apply the user's queries directly to the text repository, i.e., to the one megabyte text files, *mapping* the occurrence of each alternative sentence

to an intermediate sentence/frequency pair and then running a *reduce* to get the total frequencies. Each single computer would then be handled a map or a reduce operation. It should then be possible to serve a large number of user queries in real time, given that one had a large cluster of servers.

7 Discussion

Many types of mistakes, such as a bad choice of preposition, wrong word, and erroneous data are found, even with our limited prototype version. But the "follow the crowd" idea breaks down in certain situations. For example, "We eat avocado" may be suggested replaced by "We eat apples." However, this is a tool for users that have knowledge of the language that they use. In fact, what this tool does, similar to spelling and grammar checkers, is transforming writing- to reading knowledge. While a person using English as second language would read and understand the sentence "We had ice cream for dessert" the same person could perhaps spell the last word with only one s. A suggestion of replacing "desert" with "dessert", for changing "reserve a ticket" into "book a ticket" or to use another preposition in "we live at the West Coast" will most often be understood.

One may criticize the system that the idea of following everybody else does not open for creativity and new ways of expression. However, when writing in a foreign language the aim of most users will be that of following the norm. The same is the case for users that know they are not proficient when writing in their own language. Also in this case "commonality" will be the goal. At last, the system only makes suggestions; it is up to the user if these are to be followed.

The prototype text repository was built by traversing the Web. The disadvantage with this approach is that on the Web everybody is a writer. Thus we find slang, misspellings and wrong grammar. Had this been the exception the correct writing of the majority would have been of help, but these misspellings are so frequent that they may influence the decisions made by the system. For example, even when ignoring words that occur just once we find that a major part of the "words" found on the Web are misspellings and constructs that is not a part of the language, at least of the official language as defined by dictionaries. Since we also want to register names and numbers there it is no solution to do a dictionary lookup. We could eliminate all texts that had a large fraction of misspellings but this would again increase the time used to build the repository.

The best solution would probably be to get access to a large high quality text repository, for example bibliographical databases for books, journal papers, official reports, etc. Then one could build a high quality repository with little effort. There are also open sources that could be explored, e.g., from project Gutenberg (www.gutenberg.org) and similar sites.

Even with large high quality repositories there will be many sentences that are not found. We can use the last sentence as an example. Even Google have no match for this. A remedy is to split the sentence into parts, checking each part by itself. As we have seen, the prototype has functionality for doing this. Another approach would be to convert the sentence into a more generic form, for example by replacing "I live at Oslo" with "I live at <city>". But this would require a dictionary to classify words, e.g. Oslo as a city. Further we may find that different prepositions are used for different cities.

8 Future work

The current system takes a sentence at a time. A product version could be integrated with the word processor and analyze whole documents. Perhaps a similar interface that are used by the spell- and grammar checker can be used, i.e., to underline words or structures that should be changed. With a color scheme one could distinguish between suggestions for correcting errors or suggestions for improvements only, perhaps also visualizing the strength of the suggestion. It may be difficult to perform the analysis on the fly, but even a system where a document could be submitted for proofing and returned later would be useful. However, spelling and grammar checking was also a separate task in the beginning. With faster networks, more computing power and parallel execution it should be possible to give feedback as one writes.

Feedback from users can open up for a system that can learn. For example, the suggestion of changing the proposition in "lived at London" will be strengthened if many users follow the advice of the system. Similar, if very few change avocado to apples in the sentence "We eat avocado", the system can deduct that avocado is not a mistake for apples.

If we could find a convenient method for classifying the raw material, the texts found on the Web, the system could let the user choose style. For example, we could have a *literature* based style where the text repository is built on books; an *academic* where we had traversed scientific journals to build a repository; or a *modern* based on a collection of Web pages such as online-newspapers, discussion forums and blogs. The classification could be based on a distinction of source, as suggested here; or by analyzing each document, basing the style attribute on average word length, average sentence length, vocabulary, phrases used, etc. Given that there are enough raw materials within each group this should be possible to achieve.

References

Buchanan, B. G., Shortliffe, E. H. (1984) Rule Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project, Addison-Wesley.

Dean, J. and Ghemawat, S. (2004) MapReduce: Simplified data processing on large clusters. Proc. Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, Dec 6-8 http://labs.google.com/papers/mapreduce.html. Also in Communications of the ACM, 2008, Vol. 51, No 1.

Dean, J. and Ghemawat, S. (2010) MapReduce: A Flexible Data Processing Tool, Communications of the ACM, Vol. 53, No 1.

Dreyfus, Hubert (1972), *What Computers Can't Do*, New York: MIT Press

Dreyfus, Hubert (1992), *What Computers Still Can't Do: A Critique of Artificial Reason*, New York: MIT Press

Hirschberg, S. (1975) A linear space algorithm for computing maximal common subsequences. Communication of the ACM. 18(6) p 341-343.

Olsen, K.A., Williams, J. G. (2004). Spelling and Grammar Checking Using the Web as a Text Repository, Journal of the American Society for Information Science and Technology (JASIST), vol. 55, no 11.

Segaran, T. (2007) *Programming Collective Intelligence*, O´Reilly.

Powell, D. R., Allison, L., Dix, T. I. (1999). A versatile divide and conquer technique for optimal string alignment. Inf. Proc. Lett. 70 p 127-139.