IB Extended Essay

Computer Science

Comparison of Jaro-Winkler and Ratcliff/Obershelp algorithms in spell check

Candidate Name: Ilya Ilyankou

Candidate Number: 000197-0031

Session: May 2014

Name of Supervisor: Julius Krajnak

Citation Style: IEEE

Word Count: 3494

Abstract

This Extended Essay focuses on comparison of two approximate string matching algorithms—Jaro-Winkler and Ratcliff/Obershelp—in their application in spell check. The essay starts with describing the theory lying behind both algorithms and illustrates them with examples.

For the comparison, a list of 53 misspelled words is created, and two databases of English words—with 58,000 and 236,000 entries—are used. The task for the two algorithms is to find three words with the highest similarity scores for each misspelled one. If the correct word appears in this top-three words list, the algorithm is awarded 1, 2 or 3 points according to the position of the correct word in the list.
Both algorithms are programmed in PHP and are run on the local server Apace with the PHP processor module.

Although both algorithms show a similar level of precision, the more accurate results are produced by the Ratcliff/Obershelp algorithm. Depending on the database, it shows a 4.0%—18.6% higher result than Jaro-Winkler algorithm.

[Word count: 163]

Acknowledgements

I would like to express my gratitude to Julius Krajnak, my former teacher of mathematics and an EE supervisor, and especially to Ivan Codarin, a former Information System Developer at the United World College of the Adriatic, for finding time to read and comment on my Extended Essay. I would also like to thank all my numerous former teachers of Computer Science, as well as my parents, for supporting my passion towards programming.

Contents

Introduction

Spell check is an important feature of modern software. It is widely used in computer programs such as text processors, email clients, dictionaries and searching engines.

After its introduction in 1980s, there have been debates whether users benefit from it or become less skillful in orthography. Researches show that with spell check on users have a tendency to rely on the computer applications and disregard the final check of their writing. Such writings end up containing more mistakes and mistypes than ones that had been written with the spell check off, when users tend to pay more attention to orthography [12].

However, spell checkers make users quicker and more confident when working with writing. The attention of users shifts from spelling rules of a particular language to a message and the usage of words in their writing. With spell check on, users might utilize unfamiliar or sophisticated words more intensively, without a fear of making a mistake. Additionally, spell check is useful when writing in a foreign language, because it suggests correct orthography for misspelled words and thus makes users memorize them. Personally I experience its usefulness every time I type an essay in Italian.

Since my first meeting with spell check around 10 years ago, I have been wondering how it works. One thing about spell check was obvious from the beginning: every typed word is compared with a previously arranged database of correctly spelled words. If a typed word is found in the database, it is considered correct and ignored, if no—it is marked as incorrect and is often underlined in text processors. However, the main curiosity was how checkers compose the list of words which might be substitutions of an incorrect word.

In Microsoft Office or LibreOffice, used by millions of users around the globe, this list appears after the right-click on the underlined in red word. For instance, Microsoft Office Word suggests the words “fish”, “flesh”, “fresh”, “fest” and “fess” as possible substitutes for an inexistent word “fesh”.

After the investigation into algorithms used for spell check, I decided to compare the effectiveness of the two string matching algorithms—Jaro-Winkler's distance and Ratcliff/Obershelp algorithm—which are widely used nowadays. Due to similar principles and levels of complexity, I was sure that both of them will show approximately the same level of accuracy in spell check.

Review of spell check algorithms

Most algorithms that are used by spell checkers can be divided into two groups.

Algorithms from the first group are called **phonetic.** They compare words according to pronunciation, taking into account sounds that can be misinterpreted. For English such sounds can be given by combinations “wr” and “r” (in words “wrong” and “right”), “u” and “oo” (“Luther” and “loop”).

However, due to differences in phonetic rules in every language, such algorithms cannot be used globally. Most of them were developed for English; thus, they will give inappropriate results when applied for other languages. The most popular phonetic algorithms are *Soundex* (used, inter alia, for the purposes of the United States Census) [9] and *Metaphone.*

The second group consists of **approximate string matching algorithms** that are based on finding the similarity score between two strings, one of which is inputted and another is an entry from a database [4]. Similarity score is a number, usually between 0 and 1, where 0 corresponds to no similarity between two strings, and 1 corresponds to the complete match of the two strings. Algorithms from this group calculate the similarity score according to a number of repeating symbols or blocks of symbols in two strings, their location and some other factors.

For my Extended Essay I decided to compare two approximate string matching algorithms—Jaro-Winkler and Ratcliff/Obershelp. As any approximate string matching algorithm, they have clear mathematical logic behind them and can be implemented universally in every language which uses alphabetical script. Moreover, they both combine efforts of two developers.

Jaro-Winkler distance algorithm

Jaro distance metric was introduced in 1989 by Matthew A. Jaro's as a comparator to be used in censuses and health data files. It was later modified by William E. Winkler, who believed that similarity score between two strings that have a longer set of symbols in common at their beginning should have a higher similarity score than those which contain a mistake in first few symbols. [13]

Being a similarity function, for the two strings S_1 and S_2 the algorithm returns a value from 0 to 1, where 0 corresponds to no similarity and 1 to a complete match.

Jaro distance is represented by the formula $D_j = \frac{1}{3} * (\frac{m}{|S_1|} + \frac{m}{|S_2|} + \frac{m-t}{m})$.

> Here, $|S_1|$ and $|S_2|$ are lengths of strings S_1 and S_2 respectively (in my case, S_1 is a misspelled/mistyped word and S_2 represents each word from a database);
> m is a number of **matching** symbols,
> t is a number of **transpositions**.

Two characters are called **matching** if the one from the string S_1 coincides with another one from the string S_2 which is located not farther than $\left\lfloor \frac{\max(|S_1|,|S_2|)}{2} \right\rfloor - 1$. For each **pair** of **matching** characters with different sequence order the number of transpositions ***t*** is increased by 1. [11]

For instance, in the words *HOUSE* and *HOME* the three matching symbols are *H*, *O* and *E*. Since these symbols appear in both strings in the same order, the number of transpositions for such strings is $t = \frac{0}{2} = 0$.
In contrast, matching symbols for words HOUSE and HOUES are *H, O, U, S, E*. But as the characters *S* and *E* appear in both strings in different order, the number of transpositions for such strings is $t = \frac{2}{2} = 1$.

If the number of **matching** symbols m equals to 0, the Jaro distance must be returned as 0 without calculations, as division by 0 mathematically cannot be carried out.

Example of Jaro distance calculations

Let's compare two words—MATHEMATICS and MATEMATICA using Jaro distance method:

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
S_1	M	A	T	H	E	M	A	T	I	C	S
S_2	M	A	T	E	M	A	T	I	C	A	

a) The length of the string S_1:
$|S_1| = 11$

The length of the string S_2:
$|S_2| = 10$

b) First three symbols *M, A, T* of each string coincide (therefore, are matching). Thus, the number of matching strings on this step m = 3.

c) Symbol [4] of the string S_1 —H— is not the same as the respective symbol of the string S_2 (E). Calculating the admissible distance d_m for the symbols to be called matching, we get:

$$d_m = \left\lfloor \frac{\max(11, 10)}{2} \right\rfloor - 1 = \left\lfloor \frac{11}{2} \right\rfloor - 1 = 5 - 1 = 4$$

d) In the string S_2, from the symbol [4] we process $d_m = 4$ symbols to the to the left and to the right in order to find the symbol *H*, but from the left we get *M-A-T* and from the right *M-A-T-I*. Therefore, there is no matching symbol for *H***.**

e) Continuing with the string S_1, we get *E* at the position of [5]. Although the symbol [5] of the string S_2 is *M*, not *E,* the one to the left from it is matching (string S_2, symbol [4], "*E*"). Thus, number m must be increased by 1.
Now, the number of matching symbols m = 4.

f) The same process described in *e)* will repeat with all symbols of the string S_2 with indices [6] to [10]. Thus, after we process the symbol [10], the number of matching symbols m = 9. String S_2 does not contain a matching symbol for

S (string S_1, [11]).

g) After the matching symbols are figured out, the number of transpositions must be calculated. In this example, all the **matching** symbols appear in the same **order** in both strings. Thus, the fact that they have different indices in two strings does not influence the number t of transpositions. $t = \frac{0}{2} = 0$.

h) Thus, all the data needed to calculate Jaro distance D_j is found. The Jaro distance D_j for the words MATHEMATICS and MATEMATICA is:

$$D_j = \frac{1}{3} * \left(\frac{9}{11} + \frac{9}{10} + \frac{9-0}{9}\right) = \frac{1}{3} * \left(\frac{9}{11} + \frac{9}{10} + \frac{1}{1}\right) = 0.906$$

Winkler's improvement

The main idea of Winkler's improvement in the algorithm was to give two comparing strings a higher score if they start with the same symbol(s). His theory was that mistypes are not usually made in the beginning of words.

To get a Jaro-Winkler score, the additional formula is used:
$D_{jw} = D_j + l * p * (1 - D_j)$.

Here, D_j is a Jaro distance for a pair of strings,
l is the number of coinciding words at the beginning of the two words,
p is a coefficient, p $\in[0,1]$.

Analyzing the Winkler formula, we see that if the product of l and p is equal to 1, the expression $l * p * (1 - D_j)$ gives the number needed for the sum $D_j + l * p * (1 - D_j)$ to equal 1. Thus, when $l * p$ = 1, the Jaro-Winkler function regards the two strings as perfectly matching. [11]

For $l * p$ to be equal to 1, we need the coefficient p to be equal to the inverse of the number of coinciding symbols at the beginning of the two strings. The coefficient p can be chosen depending on the specific problem the program must solve. For example, if I agree that two strings which start with 5 identical characters can be considered the same, I set the coefficient $p = \frac{1}{5}$.

However, in this case the Jaro-Winkler score will exceed 1 if two strings have 6 or more first characters in common, as $6 * \frac{1}{5} > 1$. Thus, to use Jaro-Winkler distance algorithm correctly, the coefficient p must be found.

William Winkler himself, after a series of experimentations, came to a conclusion that $\mathrm{p} = 0.1$ is the most appropriate coefficient for most cases. In this case, the two strings will get a maximum score of 1 if their 10 first characters coincide.

However, even when p is relatively small, there is no guarantee that the algorithm will get correct results. For instance, with $\mathrm{p} = 0.1$ the misspelled word CONSTITUTIOM will have the same similarity score of 1 with words CONSTITUTION and CONSTITUTIONAL, while the Jaro distance will award them different (although, with a small difference) similarity scores, making the comparison more precise.

For the example with MATEMATICA and MATHEMATICS, the Jaro-Winkler similarity score is:

$$D_{jw} = 0.906 + 3 * 0.1 * (1 - 0.906) = 0.934$$

Thus, the similarity score was increased by $\frac{(D_{jw} - D_j)}{D_j} * 100 = \frac{0.934 - 0.906}{0.906} * 100 = 3.09\%$

Ratcliff/Obershelp pattern-matching algorithm

Ratcliff/Obershelp pattern-matching algorithm was introduced by John W. Ratcliff and John A. Obershelp in 1983. This algorithm had an impact on the industry of educational software.

Before, educational software had often offered only multiple-choice tests, as for typed-by-user answers algorithms for processing and checking the inputted data were needed.

For example, for the question who the Egyptian pharaoh of the 18th dynasty was, the answers Tutankhamun, Tutenkhamun, Tutankhamen, Tutankhamon must be considered as correct. Additionally, a user could have inputted double “m” or made other sort of mistype.

The Ratcliff/Obershelp algorithm helped to solve this problem. As Jaro-Winkler distance algorithm, the Ratcliff/Obershelp returns the value from 0 to 1, where 1 is a complete match for two given strings.

The Ratcliff/Obershelp algorithm is expressed by the formula $D_{ro} = \frac{2*K_m}{|S_1|+|S_2|}$.
Here, K_m is a number of **matching** characters,
$|S_1|$ and $|S_2|$ are lengths of strings S_1 and S_2 respectively.

In Ratcliff/Obershelp algorithm, the concept of **matching** symbols is different from the one of Jaro-Winkler. First, the longest substring that strings S_1 and S_2 have in common is found. It is called an *anchor*. The value of K_m is increased by the length of the anchor. Then, the remaining parts of the string to the left and to the right of the anchor must be examined as if they were new strings (in other words, step 1 is repeated). The process is repeated till all the characters of the strings S_1 and S_2 are analyzed.

Example of Ratcliff/Obershelp score calculations

Let’s consider the same strings MATHEMATICS and MATEMATICA.

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
S_1	M	A	T	H	E	M	A	T	I	C	S
S_2	M	A	T	E	M	A	T	I	C	A	

a) The length of the string S_1:

$|S_1| = 11$

The length of the string S_2:

$|S_2| = 10$

b) The longest substring that the two strings have in common is *EMATIC*. Therefore, *EMATIC* is an anchor, and $K_m = |EMATIC| = 6$.

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
S_1	M	A	T	H	E	M	A	T	I	C	S
S_2	M	A	T	E	M	A	T	I	C	A	

c) To the left from the anchor there are sets of symbols *MATH* and *MAT* remaining. The longest common substring of those is *MAT*. Therefore, $K_m = 6 + |MAT| = 9$.

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
S_1	M	A	T	H	E	M	A	T	I	C	S
S_2	M	A	T	E	M	A	T	I	C	A	

d) As *MAT* substring is the beginning of both strings S_1 and S_2, there are no symbols to the left of it. On the right from *MAT*, where we have *E* in the string S_1 and no characters in the string S_2. Therefore, K_m remains the same and we go to the characters on the right from the anchor.

e) To the right from the anchor, there are characters *S* and *A* left. As they are different, they are not matching. Thus, the value of K_m remains 9 and all the characters in both strings are considered. Therefore, we have all the data needed to calculate the Ratcliff/Obershelp score.

The Ratcliff/Obershelp similarity score for the strings MATHEMATICS and MATEMATICA are:

$$D_{ro} = \frac{2 * 9}{10 + 11} = \frac{18}{21} = 0.857$$

The mathematical part of the Ratcliff/Obershelp algorithm does not look as sophisticated as the one of Jaro-Winkler algorithm. Although, in their formulae the same elements are used: lengths of strings S_1 and S_2 , and the number of **matching** symbols. However, Jaro-Winkler algorithm uses an additional variable t expressing the number of **transpositions**, as well as l and p (the number of repeating symbols at the beginning of two strings and the coefficient respectively).

Methodology of comparison

There is a dilemma in choosing a database of English words for the comparison. More entries increase the possibility that the database will contain the correct version of the misspelled word. On the other hand, a big database will contain more words which are used in modern English extremely rarely, and possibly more words will receive a high similarity score along with the real one.

There are roughly a million words in the modern English language. However, Oxford Dictionary contains slightly over 200,000 entries. It signifies that most words existing in the language are not widely used.

Thus, depending on the task we must carefully chose the size of the database depending on the range of vocabulary that might be utilized by users.

For the comparison, I chose two databases that can be used for free. FreeBSD list [14] contains around 236,000 entries and Mieliestronk's dictionary [15] has around 58,000. Both of them are "txt" files, containing each word on a new line and having different forms of nouns (singular and plural: "teacher" and "teachers"), verbs (present, past and gerund: "teach", "taught", "teaching"), prefixes ("overteach"). I decided to use both databases, as compare the results within them.

To get a list of misprinted words as realistic as possible, five people, of whom two are native speakers of English, were asked to type passages in English which they were dictated. The passages were prepared beforehand from up-to-date online sources, such as Wikipedia, the websites of Chicago Tribune and Forbes. Correcting the misprints was not permitted. It allowed obtaining a list of words which consisted of *mistyped*, as well as *misspelled* words.

Table 1: The list of misspelled and mistyped words in alphabetical order, in lowercase

No.	**Misspelled/mistyped words**	**Correct (meant) words**
1	acommodation	accommodation
2	bandadge	bandage
3	cathegory	category
4	collegue	colleague
5	coatia	croatia
6	definately	definitely
7	diarea	diarrhoea
8	diseace	disease
9	emberasment	embarrassment
10	enhansment	enhancement
11	intire	entire
12	equaterial	equatorial
13	exagurate	exaggerate
14	fittiest	fittest
15	formely	formerly
16	fourty	forty
17	garantee	guarantee
18	happend	happened
19	happilly	happily
20	harrased	harassed
21	kenedy	kennedy
22	lapyop	laptop
23	lisence	license
24	lollypop	lollipop
25	menkind	mankind
26	milenium	millennium
27	misundrestanding	misunderstanding
28	mosow	moscow
29	narow	narrow
30	nostalia	nostalgia
31	occured	occurred
32	passtime	pastime
33	percieve	perceive
34	persistant	persistent
35	poetty	poetry
36	polititian	politician
37	portugese	portuguese
38	propoganda	propaganda
39	publically	publicly
40	quizz	quiz
41	raiting	rating
42	reinessance	renaissance
43	rythm	rhythm
44	sence	sense
45	silouhetted	silhouetted
46	souverein	sovereign
47	spounge	sponge
48	squirel	squirrel
49	thoroly	thoroughly
50	tounge	tongue
51	triology	trilogy
52	truely	truly
53	whith	with

For each word from the left column in *Table 1*, the two comparing algorithms had to return the short-list of 3 words for which the calculated similarity score was the highest. Depending on whether or not short-lists contained the respective original (correct) word, algorithms received points from 0 to 3, depending on the position of the correct word on the short-list. If the correct word got the very high similarity score (and, therefore, was the first on the short-list), the algorithm received 3 points. For the 2nd position it received 2 points and for the 3rd position—1 point.

If the correct was not on a short-list, the algorithm received 0 points for that particular test.

Programming the algorithms

Jaro-Winkler distance algorithm [18], written in PHP, was found on open access under the GNU General Public License. I tested it manually, comparing results of the algorithm with previously calculated by me scores. After making sure that it works correctly, I decided to use it for my research.

I didn't manage to find the Ratcliff/Obsershelp algorithm written in any computing language on open access, therefore I wrote it myself. The algorithm of finding the longest substring of the two strings [16] in PHP was found online, and was used as a part of my program.

The program was launched on the local server with Denwer [17] (consisting of the web server Apache and the PHP processor module). To make the output of the program readable, I used a markup language for the web HTML and cascading style sheets CSS.

On the computer with 8 GB of RAM and Intel i7 processor, the runtime of the program was around 11 hours. During this time, the program compared 53 misspelled words with words from the two databases (58,000 and 236,000 words) and produced the resulting table in HTML format.

The listing of the entire program is represented in *Appendix A*, and the resulting table is represented in *Appendix B*.

Results

The points received by the algorithms are shown in *Table 2*. The '*' sign means that the right word was absent in the dictionary. The "**" sign means that the words which is supposed to be misspelled presents in a database as a correct one. The colored areas indicate significant differences in similarity scores given by two comparing algorithms.

Table 2: Points awarded to the algorithms

No.	Misspelled word	Database FreeBSD		Database Mieliestronk	
		Jaro-Winkler	Ratcliff/Obershelp	Jaro-Winkler	Ratcliff/Obershelp
1	acommodation	3	3	3	3
2	bandadge	3	3	3	3
3	cathegory	3	3	3	3
4	collegue	3	3	3	3
5	coatia	0*	0*	3	3
6	definately	3	3	3	3
7	diarea	0—2	0—2	2	0—2
8	diseace	2	2	3	3
9	emberasment	0	3	1—2	3
10	enhansment	2	1—3	3	2—3
11	intire	0	1—3	0	3
12	equaterial	3	3	3	3
13	exagurate	0	1	3	3
14	fittiest	0*	0*	3	3
15	formely	3	3	3	3
16	fourty	3	3	3	3
17	garantee	3	3	3	3
18	happend	0*	0*	3	3
19	happilly	3	3	3	3
20	harrased	0*	0*	3	3
21	kenedy	0*	0*	3	3
22	lapyop	0*	0*	2—3	3
23	lisence	3	0	3	0
24	lollypop	3	3	3	3
25	menkind	2**	2**	3	3
26	milenium	3	3	3	3
27	misundrestanding	3	3	3	3
28	mosow	3	3	3	3
29	narow	3	3	3	3
30	nostalia	3	3	3	3
31	occured	0*	0*	3	3
32	passtime	3	3	3	3
33	percieve	3	3	3	3
34	persistant	3	3	3	3
35	poetty	0—1	0—1	0—1	1
36	polititian	3	3	3	3
37	portugese	3	3	0*	0*
38	propoganda	0	3	1	3
39	publically	3	3	3	3
40	quizz	2	2	3	3
41	raiting	0	3	0	3
42	reinessance	0	1—2	2	2—3

43	rythm	0	1	1	1
44	sence	0**	0**	0	0
45	silouhetted	0*	0*	3	3
46	souverein	1	2—3	2	3
47	spounge	3	3	3	3
48	squirel	2—3	2—3	3	3
49	thoroly	2	1	3	3
50	tounge	3	0—2	3	2—3
51	triology	1	1	3	3
52	truely	3	3	3	3
53	whith	0	2—3	0	3
	TOTAL	**92—96**	**100—113**	**131—134**	**140—145**

Evaluation and conclusion

Overall, the total result for the Jaro-Winkler distance algorithm within FreeBSD database is 92—96 scores, for Ratcliff/Obsershelp is 100—113 scores. In case of Mieliestroke's database, Jaro-Winkler received 131—134 scores, while Ratcliff/Obershelp got 140—145.

The Ratcliff/Obershelp algorithm completed the task more accurately within both dictionaries.

The percentage by which the Ratcliff/Obershelp algorithm was more efficient comparing to the Jaro-Winkler distance algorithm is:

a) Within FreeBSD dictionary:

$$E_{FreeBSD} = \left(1 - \frac{96}{100}\right) * 100 \; to \; \left(1 - \frac{92}{113}\right) * 100 = 4.0\% \; to \; 18.6\%$$

b) Within Mieliestronk’s dictionary:

$$E_{FreeBSD} = \left(1 - \frac{134}{140}\right) * 100 \; to \; \left(1 - \frac{131}{145}\right) * 100 = 4.3\% \; to \; 9.7\%$$

Thus, according to the test, the Ratcliff/Obershelp pattern matching algorithm is at minimum 4% more efficient than Jaro-Winkler distance algorithm.

Impressively, in Mieliestroke's dictionary 39 out of 53 tested words (which equal 73.6%) received the highest score of 3 from both Jaro-Winkler and Ratcliff/Obershelp algorithms. That reflects the efficiency of either algorithms and confirms that even without further improvements the algorithms can find its application in modern software.

However, the Ratcliff/Obershelp algorithm gave the same similarity score within the two dictionaries to a significantly bigger number of words than the Jaro-Winkler distance algorithm. It is expressed through the difference between the lower and the upper total score for the same dictionary. For the FreeBSD dictionary the difference makes 13 points and for Mieliestroke's it is 5, when for the Jaro-Winkler distance algorithm it is equal to 4 and 4 respectively.

Partially, this can be explained by simpler mathematical operations that produce a narrower range of possible similarity scores. In real life software, it might create more confusion, producing a bigger short-list of words with the same high similarity score. To deal with that, additional filters and improvements might be used.

For instance, the preference may be given to a word with which a misspelled word has the longest common substring at the beginning, as it is implemented in Winkler's improvement. Additionally, such tools as frequency lists—lists which indicate how popular the words are based on their frequency of appearance in literature—might be used, when the preference will be given to a word with the highest index of frequency within the short-list.

Paradoxically, a bigger in size FreeBSD dictionary did not contain 7 out of 53 tested words, while a 4-times smaller Mieliestroke's dictionary contained all the words except one. It demonstrates that the number of words in a dictionary does not necessarily reflect its quality, and a wisely chosen selection of words in a dictionary is the main condition for carrying out an effective spell check.

Analyzing the resulting table, I noticed that the Ratcliff/Obershelp algorithm gives a similarity score more precisely when a word contains one mistype, such as a wrong or missed letter, or a sequence of wrong characters following one another. The evidence of this are the word #11 (*intire* instead of *entire*), #38 (*propoganda* instead of *propaganda*), as well as #41 (*raiting* instead of *rating*) and #53 (*whith* instead of *with*). In this case, the two comparing strings have longer common substrings (one big substring if the mistake is located closer to the beginning or to the end of the string, or two smaller parts if a mistake is located closed to the middle of the word), and it is what is needed for the two strings to get a higher similarity score in the Ratcliff/Obershelp pattern matching algorithm.

Theoretically, the Ratcliff/Obershelp algorithm works especially well when a mistyped character (or a few) is the first or the last character in the string. In this scenario, in the very first loop Ratcliff/Obershelp algorithm selects the common substring of the comparing strings and marks it as matching, awarding the similarity score a high value.

Due to the Winkler's improvement, in Jaro-Winkler the strings starting with the longer equal sequence of characters receive a higher similarity score. However, the improvement does have a diminishing impact on the Jaro distance of the two strings, thus even when strings begin with different characters, their similarity score will not be decreased.

The Ratcliff/Obershelp algorithm showed a 4% —18.6% better result while processing the list of 53 misspelled words. Nevertheless, I don't rule out of the possibility that the set of words favored such conclusion. For a more precise investigation, the list containing hundreds or thousands of misspelled words must be used. And then, it might be that the two algorithms will show a very similar level of accuracy.

Bibliography

[1] P. Black, "Ratcliff/Obershelp pattern recognition," in *Dictionary of Algorithms and Data Structures* [online], Dec. 2004. Available: http://www.nist.gov/dads/HTML/ratcliffObershelp.html. [Accessed: Nov. 07, 2013].

[2] W. Cohen, P. Ravikumar, and S. Fienberg, "A Comparison of String Distance Metrics for Name-Matching Tasks," *Proceedings of IJCAI-03 Workshop on Information Integration, pp. 73—78, Aug. 2003.*

[3] S. Gries, "What is Corpus Linguistics?" Language and Linguistics Compass, vol. 3, pp. 1225—1241, Sept. 2009.

[4] P. Hall and G. Dowling, "Approximate String Matching," Computing Surveys, vol. 12, no. 4, pp. 381—402, Dec. 1980.

[5] A. Kostarev and D. Koterov, *PHP 5 in the original,* ed. 2. St. Petersburg, Russia: BHV, 2012.

[6] J. Peterson, "Computer Programs for Detecting and Correcting Spelling Errors," in Communications of the ACM, vol. 23, no. 12, Dec. 1980.

[7] J. Ratcliff, and D. Metzener, "*Pattern Matching: The Gestalt Approach,*" in Dr. Dobb's Journal, Jul. 1988.

[8] G. Recchia and M. Louwerse, "A Comparison of String Similarity Measures for Toponym Matching," Proc. 21st ACM SIGSPATIAL Int. Conf. Advances in Geographic Information Systems, Nov. 05—08, 2013, Orlando, Florida, USA. Available: http://stko.geog.ucsb.edu/comp2013/comp2013_submission_2.pdf. [Accessed: Nov. 09, 2013]

[9] J. Repici (2002). "Understanding Classic SoundEx Algorithms". Creativylist [Online]. Available: http://creativyst.com/Doc/Articles/SoundEx1/SoundEx1.htm. [Accessed: Nov. 10, 2013].

[10] E. Ristad and P. Yianilos, "Learning String Edit Distance," Dept. Comput. Sci, Princeton University, Rep. CS-TR-532-96, Oct. 1996. Available: http://pnylab.com/pny/papers/sed/sed.pdf. [Accessed: Nov. 10, 2013].

[11] A. Rodichevski, “Approximate string-matching algorithms, part 2,” Morfoedro, a portal of arts and culture. Available: http://www.morfoedro.it/doc.php?n=223&lang=en. [Accessed: Nov. 08, 2013].

[12] J. Sorrentino (Jan. 11, 2008). “Is Spell Check Creating a Generation of Dummies?” Education.com [Online]. Available: http://www.education.com/magazine/article/spell_check/. [Accessed: Nov. 09, 2013].

[13] W. Winkler (1999). "The state of record linkage and current research problems". Statistics of Income Division, Internal Revenue Service Publication R99/04 [Online]. Available: http://www.census.gov/srd/papers/pdf/rr99-04.pdf. [Accessed: Nov. 08, 2013].

Databases of English words:

[14] List FreeBSD [Online]. Available: http://svnweb.freebsd.org/base/head/share/dict/web2?view=co. [Accessed: Nov. 07, 2013].

[15] Mieliestronk’s list of more than 58 000 English words [Online]. Available: http://www.mieliestronk.com/corncob_lowercase.txt. [Accessed: Nov. 07, 2013].

Programs and source codes:

[16] C. Greenbacker. The algorithm which compares two strings and returns longest common substring. [Online] Source available: http://web.archive.org/web/20090823033402/http://artificialminds.net/labs/strlcs.txt. [Accessed: Nov. 09, 2013]

[17] D. Koterov (2013). Denwer—local server. Available: http://www.denwer.ru/about.html. [Accessed: Nov. 08, 2013].

[18] I. Ugrina (2005). A PHP library implementing Jaro and Jaro-Winkler distance, measuring similarity between strings, ver. 1.3. [Online] Source available: http://www.iugrina.com/files/JaroWinkler/JaroWinkler.phps [Accessed: Nov. 09, 2013].

Appendix A: Resulting table

Original word	Dictionary FreeBSD						Dictionary Mieliestronk					
	Jaro-Winkler			Ratcliff/Obershelp			Jaro-Winkler			Ratcliff/Obershelp		
1. acommodation	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
	1	accommodation	0.979	1	accommodation	0.96	1	accommodation	0.979	1	accommodation	0.96
	2	commodation	0.972	2	commodation	0.957	2	accommodations	0.962	2	accommodations	0.923
	3	accommodational	0.947	3	accommodational	0.889	3	accommodating	0.937	3	accommodating	0.88
	Raccommodation		0.979	Raccommodation		0.96	Raccommodation		0.979	Raccommodation		0.96
2. bandadge	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
	1	bandage	0.975	1	bandage	0.933	1	bandage	0.975	1	bandage	0.933
	2	bandager	0.95	2	bandager	0.875	2	bandaged	0.963	2	bandaged	0.875
	3	banda	0.925	3	rebandage	0.824	3	bandages	0.95	3	bandages	0.875
	Rbandage		0.975	Rbandage		0.933	Rbandage		0.975	Rbandage		0.933
3. cathegory	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
	1	category	0.974	1	category	0.941	1	category	0.974	1	category	0.941
	2	cathro	0.9	2	subcategory	0.8	2	catcher	0.889	2	theory	0.8
	3	cathography	0.898	3	theory	0.8	3	cattery	0.889	3	catcher	0.75
	Rcategory		0.974	Rcategory		0.941	Rcategory		0.974	Rcategory		0.941
4. collegue	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
	1	colleague	0.978	1	colleague	0.941	1	colleague	0.978	1	colleague	0.941
	2	college	0.975	2	college	0.933	2	college	0.975	2	college	0.933
	3	colleger	0.95	3	colleger	0.875	3	colleagues	0.96	3	colleagues	0.889
	Rcolleague		0.978	Rcolleague		0.941	Rcolleague		0.978	Rcolleague		0.941
5. coatia	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
	1	coati	0.967	1	coati	0.909	1	croatia	0.957	1	croatia	0.923
	2	coaita	0.961	2	coaita	0.833	2	croatian	0.933	2	croatian	0.857
	3	coat	0.933	3	coatie	0.833	3	coat	0.925	3	coat	0.8
	Rcroatia		0.957	Rcroatia		0.923	Rcroatia		0.957	Rcroatia		0.923
6. definately	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
	1	definitely	0.96	1	definitely	0.9	1	definitely	0.96	1	definitely	0.9
	2	defiantly	0.958	2	defiantly	0.842	2	defiantly	0.958	2	defiantly	0.842
	3	definably	0.938	3	definably	0.842	3	definably	0.938	3	definably	0.842
	Rdefinitely		0.96	Rdefinitely		0.9	Rdefinitely		0.96	Rdefinitely		0.9
7. diarea	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
	1	diarrhea	0.95	1	diarrhea	0.857	1	diarrhea	0.95	1	diarrhea	0.857
	2	diarhemia	0.933	2	area	0.8	2	diarrhoea	0.933	2	area	0.8
	3	diarrheal	0.933	3	dare	0.8	3	diarrhoeal	0.92	3	dare	0.8
	Rdiarrhoea		0.933	Rdiarrhoea		0.8	Rdiarrhoea		0.933	Rdiarrhoea		0.8
8. diseace	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
	1	dispeace	0.971	1	dispeace	0.933	1	disease	0.943	1	disease	0.857
	2	disease	0.943	2	disease	0.857	2	diseased	0.921	2	diseased	0.8
	3	diseased	0.921	3	diseased	0.8	3	diseases	0.921	3	diseases	0.8
	Rdisease		0.943	Rdisease		0.857	Rdisease		0.943	Rdisease		0.857
9. emberasment	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
	1	embedment	0.923	1	embarrassment	0.833	1	embers	0.909	1	embarrassment	0.833
	2	embracement	0.923	2	embowerment	0.818	2	embellishment	0.902	2	temperament	0.818
	3	embowerment	0.915	3	embracement	0.818	3	embarrassment	0.902	3	embarrassments	0.8
	Rembarrassment		0.902	Rembarrassment		0.833	Rembarrassment		0.902	Rembarrassment		0.833
10. enhansment	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
	1	enchainment	0.925	1	enchainment	0.857	1	enhancement	0.921	1	enchantment	0.857
	2	enhancement	0.921	2	enchantment	0.857	2	enhancements	0.908	2	enhancement	0.857
	3	enhance	0.891	3	enhancement	0.857	3	enhance	0.891	3	enchantments	0.818
	Renhancement		0.921	Renhancement		0.857	Renhancement		0.921	Renhancement		0.857
11. intire	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score

	1	intine	0.933	1	entire	0.833	1	interim	0.928	1	entire	0.833
	2	interim	0.928	2	intine	0.833	2	inter	0.914	2	tire	0.8
	3	intrine	0.928	3	lintie	0.833	3	interims	0.903	3	entires	0.769
	R	entire	0.789	R	entire	0.833	R	entire	0.789	R	entire	0.833
12. equaterial	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	equatorial	0.96	1	equatorial	0.9	1	equatorial	0.96	1	equatorial	0.9
	2	equatorially	0.93	2	equilateral	0.857	2	equate	0.92	2	equilateral	0.857
	3	equate	0.92	3	quaternal	0.842	3	equilateral	0.897	3	arterial	0.778
	R	equatorial	0.96	R	equatorial	0.9	R	equatorial	0.96	R	equatorial	0.9
13. exagurate	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	exaugurate	0.951	1	exaugurate	0.947	1	exaggerate	0.888	1	exaggerate	0.842
	2	exarate	0.948	2	exarate	0.875	2	expurgate	0.874	2	exaggerated	0.8
	3	exarchate	0.896	3	exaggerate	0.842	3	exaggerated	0.873	3	exaggerates	0.8
	R	exaggerate	0.888	R	exaggerate	0.842	R	exaggerate	0.888	R	exaggerate	0.842
14. fittiest	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	fittiness	0.931	1	fittiness	0.824	1	fittest	0.975	1	fittest	0.933
	2	fitters	0.921	2	fitters	0.8	2	fitters	0.921	2	fattiest	0.875
	3	fittingness	0.902	3	fiftieth	0.75	3	wittiest	0.917	3	wittiest	0.875
	R	fittest	0.975	R	fittest	0.933	R	fittest	0.975	R	fittest	0.933
15. formely	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	formerly	0.975	1	formerly	0.933	1	formerly	0.975	1	formerly	0.933
	2	formel	0.971	2	formel	0.923	2	formally	0.921	2	foreplay	0.8
	3	forme	0.943	3	forelay	0.857	3	form	0.914	3	formally	0.8
	R	formerly	0.975	R	formerly	0.933	R	formerly	0.975	R	formerly	0.933
16. fourty	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	forty	0.956	1	forty	0.909	1	forty	0.956	1	forty	0.909
	2	fourthly	0.95	2	fourthly	0.857	2	fourthly	0.95	2	fourthly	0.857
	3	four	0.933	3	floury	0.833	3	four	0.933	3	floury	0.833
	R	forty	0.956	R	forty	0.909	R	forty	0.956	R	forty	0.909
17. garantee	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	guarantee	0.967	1	guarantee	0.941	1	guarantee	0.967	1	guarantee	0.941
	2	garance	0.921	2	grantee	0.933	2	guaranteed	0.94	2	grantee	0.933
	3	grantee	0.92	3	reguarantee	0.842	3	guarantees	0.94	3	guaranteed	0.889
	R	guarantee	0.967	R	guarantee	0.941	R	guarantee	0.967	R	guarantee	0.941
18. happend	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	happen	0.971	1	append	0.923	1	happened	0.975	1	happened	0.933
	2	append	0.952	2	happen	0.923	2	happen	0.971	2	append	0.923
	3	apprend	0.905	3	apprend	0.857	3	append	0.952	3	happen	0.923
	R	happened	0.975	R	happened	0.933	R	happened	0.975	R	happened	0.933
19. happilly	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	happily	0.975	1	happily	0.933	1	happily	0.975	1	happily	0.933
	2	happify	0.921	2	unhappily	0.824	2	unhappily	0.884	2	unhappily	0.824
	3	unhappily	0.884	3	happify	0.8	3	happier	0.868	3	apply	0.769
	R	happily	0.975	R	happily	0.933	R	happily	0.975	R	happily	0.933
20. harrased	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	arrased	0.958	1	arrased	0.933	1	harassed	0.942	1	harassed	0.875
	2	harassedly	0.901	2	harassedly	0.778	2	harried	0.921	2	arrases	0.8
	3	harr	0.9	3	unharassed	0.778	3	harare	0.903	3	arrayed	0.8
	R	harassed	0.942	R	harassed	0.875	R	harassed	0.942	R	harassed	0.875
21. kenedy	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	kendyr	0.922	1	keened	0.833	1	kennedy	0.967	1	kennedy	0.923
	2	kend	0.922	2	kendyr	0.833	2	kerned	0.911	2	kerned	0.833
	3	kneed	0.89	3	kend	0.8	3	kneed	0.89	3	likened	0.769
	R	kennedy	0.967	R	kennedy	0.923	R	kennedy	0.967	R	kennedy	0.923
22. lapyop	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	lap	0.883	1	apoop	0.727	1	laptop	0.922	1	laptop	0.833

	2	lapon	0.876	2	lapon	0.727	2	lapp	0.922	2	lapp	0.8
	3	lay	0.867	3	malaprop	0.714	3	laptops	0.894	3	laptops	0.769
	R	laptop	0.922	R	laptop	0.833	R	laptop	0.922	R	laptop	0.833

	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
23. lisence	1	license	0.962	1	sence	0.833	1	license	0.962	1	licence	0.857
	2	silence	0.952	2	lenience	0.8	2	silence	0.952	2	licences	0.8
	3	licensed	0.929	3	ligeance	0.8	3	licensed	0.929	3	listened	0.8
	R	license	0.962	R	license	0.571	R	license	0.962	R	license	0.571

	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
24. lollypop	1	lollipop	0.95	1	lollipop	0.875	1	lollipop	0.95	1	lollipop	0.875
	2	lollopy	0.946	2	lollop	0.857	2	lollipops	0.931	2	lollipops	0.824
	3	lolly	0.925	3	lollopy	0.8	3	lolly	0.925	3	lolly	0.769
	R	lollipop	0.95	R	lollipop	0.875	R	lollipop	0.95	R	lollipop	0.875

	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
25. menkind	1	menkind	1	1	menkind	1	1	mankind	0.914	1	mankind	0.857
	2	womenkind	0.926	2	womenkind	0.875	2	mentioned	0.889	2	unkind	0.769
	3	mankind	0.914	3	mankind	0.857	3	mending	0.875	3	humankind	0.75
	R	mankind	0.914	R	mankind	0.857	R	mankind	0.914	R	mankind	0.857

	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
26. milenium	1	millennium	0.953	1	millennium	0.889	1	millennium	0.953	1	millennium	0.889
	2	milium	0.942	2	milium	0.857	2	milieu	0.903	2	minimum	0.8
	3	minium	0.933	3	minium	0.857	3	mile	0.9	3	ileum	0.769
	R	millennium	0.953	R	millennium	0.889	R	millennium	0.953	R	millennium	0.889

	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
27. misundrestanding	1	misunderstanding	0.988	1	misunderstanding	0.938	1	misunderstanding	0.988	1	misunderstanding	0.938
	2	misunderstandingly	0.965	2	misunderstandingly	0.882	2	misunderstandings	0.976	2	misunderstandings	0.909
	3	misunderstand	0.947	3	unmisunderstanding	0.882	3	misunderstand	0.947	3	misunderstand	0.828
	R	misunderstanding	0.988	R	misunderstanding	0.938	R	misunderstanding	0.988	R	misunderstanding	0.938

	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
28. mosow	1	moscow	0.961	1	moscow	0.909	1	moscow	0.961	1	moscow	0.909
	2	moo	0.893	2	moo	0.75	2	moo	0.893	2	moo	0.75
	3	mosswort	0.866	3	mow	0.75	3	moos	0.88	3	mow	0.75
	R	moscow	0.961	R	moscow	0.909	R	moscow	0.961	R	moscow	0.909

	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
29. narow	1	narrow	0.961	1	narrow	0.909	1	narrow	0.961	1	narrow	0.909
	2	narrowy	0.933	2	arow	0.889	2	narrows	0.933	2	narrows	0.833
	3	arow	0.933	3	narrowy	0.833	3	narrowed	0.913	3	arrow	0.8
	R	narrow	0.961	R	narrow	0.909	R	narrow	0.961	R	narrow	0.909

	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
30. nostalia	1	nostalgia	0.978	1	nostalgia	0.941	1	nostalgia	0.978	1	nostalgia	0.941
	2	notalgia	0.933	2	notalgia	0.875	2	nostalgic	0.931	2	nostalgic	0.824
	3	nostalgic	0.931	3	ostalgia	0.875	3	nostalgically	0.923	3	nostalgically	0.762
	R	nostalgia	0.978	R	nostalgia	0.941	R	nostalgia	0.978	R	nostalgia	0.941

	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
31. occured	1	occur	0.943	1	occur	0.833	1	occurred	0.975	1	occurred	0.933
	2	occurrent	0.905	2	accursed	0.8	2	occur	0.943	2	cured	0.833
	3	occursive	0.905	3	curled	0.769	3	occupied	0.921	3	occur	0.833
	R	occurred	0.975	R	occurred	0.933	R	occurred	0.975	R	occurred	0.933

	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
32. passtime	1	pastime	0.971	1	pastime	0.933	1	pastime	0.971	1	pastime	0.933
	2	pastimer	0.942	2	pastimer	0.875	2	passim	0.95	2	pastimes	0.875
	3	passive	0.921	3	passive	0.8	3	pastimes	0.927	3	passim	0.857
	R	pastime	0.971	R	pastime	0.933	R	pastime	0.971	R	pastime	0.933

	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
33. percieve	1	perceive	0.975	1	perceive	0.875	1	perceive	0.975	1	perceive	0.875
	2	perceiver	0.953	2	perceiver	0.824	2	perceived	0.953	2	perceived	0.824

	3	perceptive	0.935	3	perigee	0.8	3	perceives	0.953	3	perceives	0.824
	R	perceive	0.975	R	perceive	0.875	R	perceive	0.975	R	perceive	0.875
	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
34. persistant	1	persistent	0.96	1	persistent	0.9	1	persistent	0.96	1	persistent	0.9
	2	persist	0.94	2	resistant	0.842	2	persian	0.94	2	resistant	0.842
	3	persistently	0.93	3	persist	0.824	3	persist	0.94	3	persian	0.824
	R	persistent	0.96	R	persistent	0.9	R	persistent	0.96	R	persistent	0.9
	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
35. poetty	1	potty	0.956	1	petty	0.909	1	potty	0.956	1	petty	0.909
	2	petty	0.95	2	potty	0.909	2	petty	0.95	2	potty	0.909
	3	poet	0.933	3	piotty	0.833	3	poet	0.933	3	poetry	0.833
	R	poetry	0.933	R	poetry	0.833	R	poetry	0.933	R	poetry	0.833
	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
36. polititian	1	politician	0.96	1	politician	0.9	1	politician	0.96	1	politician	0.9
	2	politist	0.915	2	geopolitician	0.783	2	politicians	0.944	2	politicians	0.857
	3	politzerization	0.903	3	politist	0.778	3	politicisation	0.913	3	politicking	0.762
	R	politician	0.96	R	politician	0.9	R	politician	0.96	R	politician	0.9
	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
37. portugese	1	portuguese	0.98	1	portuguese	0.947	1	portage	0.905	1	portage	0.75
	2	portugee	0.978	2	portugee	0.941	2	porters	0.905	2	porters	0.75
	3	portagais	0.911	3	porthouse	0.778	3	port	0.889	3	pores	0.714
	R	portuguese	0.98	R	portuguese	0.947	R	portuguese	0.98	R	portuguese	0.947
	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
38. propoganda	1	propagand	0.924	1	propaganda	0.9	1	propound	0.915	1	propaganda	0.9
	2	propound	0.915	2	propagand	0.842	2	propagation	0.91	2	propound	0.778
	3	propago	0.911	3	propound	0.778	3	propaganda	0.904	3	propagandist	0.727
	R	propaganda	0.904	R	propaganda	0.9	R	propaganda	0.904	R	propaganda	0.9
	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
39. publically	1	publicly	0.96	1	publicly	0.889	1	publicly	0.96	1	publicly	0.889
	2	public	0.92	2	umbilically	0.857	2	public	0.92	2	cubically	0.842
	3	publican	0.915	3	cubically	0.842	3	publican	0.915	3	biblically	0.8
	R	publicly	0.96	R	publicly	0.889	R	publicly	0.96	R	publicly	0.889
	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
40. quizz	1	quizzy	0.967	1	quizzy	0.909	1	quiz	0.96	1	quiz	0.889
	2	quiz	0.96	2	quiz	0.889	2	quizzed	0.943	2	quizzed	0.833
	3	quizzee	0.943	3	quizzee	0.833	3	quizzes	0.943	3	quizzes	0.833
	R	quiz	0.96	R	quiz	0.889	R	quiz	0.96	R	quiz	0.889
	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
41. raiting	1	railing	0.933	1	rating	0.923	1	rabbiting	0.941	1	rating	0.923
	2	raising	0.933	2	gaiting	0.857	2	radiating	0.941	2	rabbiting	0.875
	3	radicating	0.92	3	grating	0.857	3	raiding	0.933	3	radiating	0.875
	R	rating	0.917	R	rating	0.923	R	rating	0.917	R	rating	0.923
	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
42. reinessance	1	reinsurance	0.927	1	reinsane	0.842	1	reinsurance	0.927	1	reinsurance	0.818
	2	reincrease	0.91	2	preissuance	0.818	2	renaissance	0.898	2	renaissance	0.818
	3	reinless	0.902	3	reinsurance	0.818	3	reins	0.891	3	refinance	0.8
	R	renaissance	0.898	R	renaissance	0.818	R	renaissance	0.898	R	renaissance	0.818
	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
43. rythm	1	erythema	0.875	1	rhythm	0.909	1	rhythm	0.86	1	rhythm	0.909
	2	eurythmy	0.875	2	dryth	0.8	2	rhythms	0.824	2	rhythms	0.833
	3	dryth	0.867	3	erythema	0.769	3	rhyme	0.805	3	rhythmic	0.769
	R	rhythm	0.86	R	rhythm	0.909	R	rhythm	0.86	R	rhythm	0.909
	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
44. sence	1	sence	1	1	sence	1	1	seance	0.956	1	seance	0.909
	2	seance	0.956	2	seance	0.909	2	seances	0.924	2	absence	0.833
	3	spence	0.95	3	spence	0.909	3	silence	0.914	3	essence	0.833

	#	Word	Score	#	Word	Score	#	Word	Score	#	Word	Score
	R	sense	0.907	R	sense	0.8	R	sense	0.907	R	sense	0.8
45. silouhetted	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	silhouette	0.944	1	silhouette	0.857	1	silhouetted	0.968	1	silhouetted	0.909
	2	siliciuretted	0.886	2	slotted	0.778	2	silhouette	0.944	2	silhouette	0.857
	3	silo	0.873	3	louchettes	0.762	3	silhouettes	0.923	3	silhouettes	0.818
	R	silhouetted	0.968	R	silhouetted	0.909	R	silhouetted	0.968	R	silhouetted	0.909
46. souverein	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	souverain	0.956	1	souverain	0.889	1	souvenir	0.953	1	sovereign	0.889
	2	souvenir	0.953	2	sovereign	0.889	2	sovereign	0.941	2	sovereigns	0.842
	3	sovereign	0.941	3	cosovereign	0.8	3	souvenirs	0.931	3	sovereignty	0.8
	R	sovereign	0.941	R	sovereign	0.889	R	sovereign	0.941	R	sovereign	0.889
47. spounge	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	sponge	0.967	1	sponge	0.923	1	sponge	0.967	1	sponge	0.923
	2	spong	0.933	2	splunge	0.857	2	sponged	0.933	2	sponged	0.857
	3	sponged	0.933	3	sponged	0.857	3	sponger	0.933	3	sponger	0.857
	R	sponge	0.967	R	sponge	0.923	R	sponge	0.967	R	sponge	0.923
48. squirel	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	squirely	0.975	1	squirely	0.933	1	squirrel	0.975	1	squirrel	0.933
	2	squirrel	0.975	2	squirrel	0.933	2	squire	0.971	2	squire	0.923
	3	squire	0.971	3	squire	0.923	3	squirrels	0.956	3	squirrels	0.875
	R	squirrel	0.975	R	squirrel	0.933	R	squirrel	0.975	R	squirrel	0.933
49. thoroly	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	thoro	0.943	1	hooly	0.833	1	thoroughly	0.94	1	thoroughly	0.824
	2	thoroughly	0.94	2	thoro	0.833	2	thor	0.914	2	throroughly	0.778
	3	thornily	0.921	3	thoroughly	0.824	3	thorny	0.91	3	hourly	0.769
	R	thoroughly	0.94	R	thoroughly	0.824	R	thoroughly	0.94	R	thoroughly	0.824
50. tounge	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	tongue	0.933	1	strounge	0.857	1	tongue	0.933	1	lounge	0.833
	2	toug	0.922	2	lounge	0.833	2	tone	0.911	2	tongue	0.833
	3	strounge	0.917	3	thunge	0.833	3	toughen	0.908	3	tone	0.8
	R	tongue	0.933	R	tongue	0.833	R	tongue	0.933	R	tongue	0.833
51. triology	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	triology	1	1	triology	1	1	trilogy	0.938	1	trilogy	0.933
	2	trichology	0.953	2	trilogy	0.933	2	terminology	0.918	2	terminology	0.842
	3	trilogy	0.938	3	storiology	0.889	3	trio	0.9	3	astrology	0.824
	R	trilogy	0.938	R	trilogy	0.933	R	trilogy	0.938	R	trilogy	0.933
52. truely	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	truly	0.961	1	truly	0.909	1	truly	0.961	1	truly	0.909
	2	true	0.933	2	rudely	0.833	2	true	0.933	2	rudely	0.833
	3	trebly	0.911	3	trebly	0.833	3	truer	0.893	3	rely	0.8
	R	truly	0.961	R	truly	0.909	R	truly	0.961	R	truly	0.909
53. whith	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**	**#**	**Word**	**Score**
	1	whit	0.96	1	whit	0.889	1	whither	0.943	1	with	0.889
	2	whither	0.943	2	with	0.889	2	whitish	0.943	2	whither	0.833
	3	whitish	0.943	3	whither	0.833	3	white	0.92	3	whitish	0.833
	R	with	0.828	R	with	0.889	R	with	0.828	R	with	0.889

Appendix B: Source codes

Listing of "main.php"

```php
<?php
ini_set('memory_limit', '-1');

include "jaroWinkler.php";
include "ratcliffObershelp.php";


$dictionaryFreeBSD = fopen("dictionary-freebsd.txt", "r");
$dictionaryMieliestronk = fopen("dictionary-mieliestronk.txt", "r");
$dictionaryMisspelled = file("dictionary-misspelled.txt");
$dictionaryCorrect = file("dictionary-correct.txt");

echo <<<HTML
<html>

<head>
    <title>The comparison of Jaro-Winkler and Ratcliff/Obershelp algorithms in spell-checking</title>
</head>

<style>

table { border: 1px solid gray; width: 100%; text-align:center;}
table tr td{border-bottom: 1px solid gray;}
table tr.trGray td {background:#888888; text-align:center; font-weight:bold;}
table tr.trCorrect td {background: #C0C0C0; color:white;}


</style>

<body>
<table cellspacing=0>

<tr>
<th rowspan=2 style="margin-bottom:5px;"> Original word </th> <th colspan=6>Dictionary FreeBSD</th> <th colspan=6>Dictionary Mieliestronk </th>
</tr>

<tr>
<td colspan=3>Jaro-Winkler</td> <td colspan=3>Ratcliff/Obershelp</td>
<td colspan=3>Jaro-Winkler</td> <td colspan=3>Ratcliff/Obershelp</td>
</tr>

HTML;

$r = NULL;
$r = array();

for ($i=0; $i<count($dictionaryMisspelled); $i++) {
    $misspelledWord = trim ( $dictionaryMisspelled[$i] );
    $correctWord = trim  ( $dictionaryCorrect[$i] );

    $r[$i]["misspelledWord"] = $misspelledWord;
    $r[$i]["correctWord"] = $correctWord;

    //Calculating Jaro-Winkler and Ratcliff/Obershelp scores for correct answers
    $r[$i]["correctWordJW"] = JaroWinkler($misspelledWord, $correctWord);
    $r[$i]["correctWordRO"] = Ratcliff($misspelledWord, $correctWord);

    $r[$i]["freebsd"]["jw"][1]["score"] = 0.0;
    $r[$i]["freebsd"]["jw"][2]["score"] = 0.0;
    $r[$i]["freebsd"]["jw"][3]["score"] = 0.0;
    $r[$i]["freebsd"]["ro"][1]["score"] = 0.0;
    $r[$i]["freebsd"]["ro"][2]["score"] = 0.0;
    $r[$i]["freebsd"]["ro"][3]["score"] = 0.0;
    $r[$i]["mielie"]["jw"][1]["score"] = 0.0;
    $r[$i]["mielie"]["jw"][2]["score"] = 0.0;
    $r[$i]["mielie"]["jw"][3]["score"] = 0.0;
    $r[$i]["mielie"]["ro"][1]["score"] = 0.0;
    $r[$i]["mielie"]["ro"][2]["score"] = 0.0;
    $r[$i]["mielie"]["ro"][3]["score"] = 0.0;

    while(!feof($dictionaryFreeBSD)) {
        $dictionaryWord = trim( fgets($dictionaryFreeBSD) );

        $jw = JaroWinkler($misspelledWord, $dictionaryWord);
        $ro = Ratcliff($misspelledWord, $dictionaryWord);

        // Jaro-Winkler
        if ($jw > $r[$i]["freebsd"]["jw"][1]["score"]) {
                $r[$i]["freebsd"]["jw"][3]["score"] = $r[$i]["freebsd"]["jw"][2]["score"];
                $r[$i]["freebsd"]["jw"][3]["word"] = $r[$i]["freebsd"]["jw"][2]["word"];

                $r[$i]["freebsd"]["jw"][2]["score"] = $r[$i]["freebsd"]["jw"][1]["score"];
                $r[$i]["freebsd"]["jw"][2]["word"] = $r[$i]["freebsd"]["jw"][1]["word"];

                $r[$i]["freebsd"]["jw"][1]["score"] = $jw;
                $r[$i]["freebsd"]["jw"][1]["word"] = $dictionaryWord;
        }
        elseif ($jw > $r[$i]["freebsd"]["jw"][2]["score"]) {
                $r[$i]["freebsd"]["jw"][3]["score"] = $r[$i]["freebsd"]["jw"][2]["score"];
                $r[$i]["freebsd"]["jw"][3]["word"] = $r[$i]["freebsd"]["jw"][2]["word"];

                $r[$i]["freebsd"]["jw"][2]["score"] = $jw;
                $r[$i]["freebsd"]["jw"][2]["word"] = $dictionaryWord;
        }
        elseif ($jw > $r[$i]["freebsd"]["jw"][3]["score"]) {
                $r[$i]["freebsd"]["jw"][3]["score"] = $jw;
                $r[$i]["freebsd"]["jw"][3]["word"] = $dictionaryWord;
        }

        // Ratcliff/Obershelp
        if ($ro > $r[$i]["freebsd"]["ro"][1]["score"]) {
                $r[$i]["freebsd"]["ro"][3]["score"] = $r[$i]["freebsd"]["ro"][2]["score"];
                $r[$i]["freebsd"]["ro"][3]["word"] = $r[$i]["freebsd"]["ro"][2]["word"];

                $r[$i]["freebsd"]["ro"][2]["score"] = $r[$i]["freebsd"]["ro"][1]["score"];
                $r[$i]["freebsd"]["ro"][2]["word"] = $r[$i]["freebsd"]["ro"][1]["word"];

                $r[$i]["freebsd"]["ro"][1]["score"] = $ro;
                $r[$i]["freebsd"]["ro"][1]["word"] = $dictionaryWord;
        }
        elseif ($ro > $r[$i]["freebsd"]["ro"][2]["score"]) {
                $r[$i]["freebsd"]["ro"][3]["score"] = $r[$i]["freebsd"]["ro"][2]["score"];
                $r[$i]["freebsd"]["ro"][3]["word"] = $r[$i]["freebsd"]["ro"][2]["word"];

                $r[$i]["freebsd"]["ro"][2]["score"] = $ro;
                $r[$i]["freebsd"]["ro"][2]["word"] = $dictionaryWord;
        }
        elseif ($ro > $r[$i]["freebsd"]["ro"][3]["score"]) {
                $r[$i]["freebsd"]["ro"][3]["score"] = $ro;
                $r[$i]["freebsd"]["ro"][3]["word"] = $dictionaryWord;
        }
    }

    while(!feof($dictionaryMieliestronk)) {
        $dictionaryWord = trim( fgets($dictionaryMieliestronk) );

        $jw = JaroWinkler($misspelledWord, $dictionaryWord);
        $ro = Ratcliff($misspelledWord, $dictionaryWord);

        // Jaro-Winkler
        if ($jw > $r[$i]["mielie"]["jw"][1]["score"]) {
                $r[$i]["mielie"]["jw"][3]["score"] = $r[$i]["mielie"]["jw"][2]["score"];
                $r[$i]["mielie"]["jw"][3]["word"] = $r[$i]["mielie"]["jw"][2]["word"];
                $r[$i]["mielie"]["jw"][2]["score"] = $r[$i]["mielie"]["jw"][1]["score"];
                $r[$i]["mielie"]["jw"][2]["word"] = $r[$i]["mielie"]["jw"][1]["word"];

                $r[$i]["mielie"]["jw"][1]["score"] = $jw;
                $r[$i]["mielie"]["jw"][1]["word"] = $dictionaryWord;
        }
        elseif ($jw > $r[$i]["mielie"]["jw"][2]["score"]) {
                $r[$i]["mielie"]["jw"][3]["score"] = $r[$i]["mielie"]["jw"][2]["score"];
                $r[$i]["mielie"]["jw"][3]["word"] = $r[$i]["mielie"]["jw"][2]["word"];

                $r[$i]["mielie"]["jw"][2]["score"] = $jw;
                $r[$i]["mielie"]["jw"][2]["word"] = $dictionaryWord;
        }
        elseif ($jw > $r[$i]["mielie"]["jw"][3]["score"]) {
                $r[$i]["mielie"]["jw"][3]["score"] = $jw;
                $r[$i]["mielie"]["jw"][3]["word"] = $dictionaryWord;
        }

        // Ratcliff/Obershelp
        if ($ro > $r[$i]["mielie"]["ro"][1]["score"]) {
                $r[$i]["mielie"]["ro"][3]["score"] = $r[$i]["mielie"]["ro"][2]["score"];
                $r[$i]["mielie"]["ro"][3]["word"] = $r[$i]["mielie"]["ro"][2]["word"];

                $r[$i]["mielie"]["ro"][2]["score"] = $r[$i]["mielie"]["ro"][1]["score"];
                $r[$i]["mielie"]["ro"][2]["word"] = $r[$i]["mielie"]["ro"][1]["word"];

                $r[$i]["mielie"]["ro"][1]["score"] = $ro;
                $r[$i]["mielie"]["ro"][1]["word"] = $dictionaryWord;
        }
        elseif ($ro > $r[$i]["mielie"]["ro"][2]["score"]) {
                $r[$i]["mielie"]["ro"][3]["score"] = $r[$i]["mielie"]["ro"][2]["score"];
                $r[$i]["mielie"]["ro"][3]["word"] = $r[$i]["mielie"]["ro"][2]["word"];

                $r[$i]["mielie"]["ro"][2]["score"] = $ro;
                $r[$i]["mielie"]["ro"][2]["word"] = $dictionaryWord;
        }
        elseif ($ro > $r[$i]["mielie"]["ro"][3]["score"]) {
                $r[$i]["mielie"]["ro"][3]["score"] = $ro;
                $r[$i]["mielie"]["ro"][3]["word"] = $dictionaryWord;
        }
    }

    fseek($dictionaryFreeBSD, 0);
    fseek($dictionaryMieliestronk, 0);

echo <<<HTML
<tr class="trGray">
<td rowspan=5> $misspelledWord </td>
<td>#</td> <td>Word</td> <td>Score</td>
<td>#</td> <td>Word</td> <td>Score</td>
<td>#</td> <td>Word</td> <td>Score</td>
<td>#</td> <td>Word</td> <td>Score</td>
</tr>

<tr>
HTML;

    for ($t=1; $t<=3; $t++) {
echo "
            <td>$t</td>
            <td>".$r[$i]['freebsd']['jw'][$t]['word']."</td>
            <td>".round($r[$i]['freebsd']['jw'][$t]['score'], 3)."</td>

            <td>$t</td>
            <td>".$r[$i]['freebsd']['ro'][$t]['word']."</td>
            <td>".round($r[$i]['freebsd']['ro'][$t]['score'], 3)."</td>

            <td>$t</td>
            <td>".$r[$i]['mielie']['jw'][$t]['word']."</td>
            <td>".round($r[$i]['mielie']['jw'][$t]['score'], 3)."</td>

            <td>$t</td>
            <td>".$r[$i]['mielie']['ro'][$t]['word']."</td>
            <td>".round($r[$i]['mielie']['ro'][$t]['score'], 3)."</td>
            </tr>";
    }

    echo "<tr class='trCorrect'>";
    for ($t=1; $t<=2; $t++) {
echo "
            <td>R</td>
            <td>$correctWord</td>
            <td>".round($r[$i]['correctWordJW'], 3)."</td>

            <td>R</td>
            <td>$correctWord</td>
            <td>".round($r[$i]['correctWordRO'], 3)."</td> ";
    }
    echo "</tr><tr><td colspan=13 style='font-size:3px'>&nbsp;</td></tr>";
}

echo "</table></body></html>";

?>
```

Listing of “jaroWinkler.php” [18]

```
<?php

function getCommonCharacters( $string1, $string2, $allowedDistance ){

  $str1_len = strlen($string1);
  $str2_len = strlen($string2);
  $temp_string2 = $string2;

  $commonCharacters='';

  for( $i=0; $i < $str1_len; $i++){

    $noMatch = True;

    // compare if char does match inside given allowedDistance
    // and if it does add it to commonCharacters
    for( $j= max( 0, $i-$allowedDistance ); $noMatch && $j < min( $i + $allowedDistance + 1, $str2_len ); $j++){
      if( $temp_string2[$j] == $string1[$i] ){
        $noMatch = False;

    $commonCharacters .= $string1[$i];

    $temp_string2[$j] = '';
      }
    }
  }

  return $commonCharacters;
}

function Jaro( $string1, $string2 ){

  $str1_len = strlen( $string1 );
  $str2_len = strlen( $string2 );

  // theoretical distance
  $distance = (int) floor(min( $str1_len, $str2_len ) / 2.0);

  // get common characters
  $commons1 = getCommonCharacters( $string1, $string2, $distance );
  $commons2 = getCommonCharacters( $string2, $string1, $distance );

  if( ($commons1_len = strlen( $commons1 )) == 0) return 0;
  if( ($commons2_len = strlen( $commons2 )) == 0) return 0;

  // calculate transpositions
  $transpositions = 0;
```

```php
  $upperBound = min( $commons1_len, $commons2_len );
  for( $i = 0; $i < $upperBound; $i++){
    if( $commons1[$i] != $commons2[$i] ) $transpositions++;
  }
  $transpositions /= 2.0;


  // return the Jaro distance
    return ($commons1_len/($str1_len) + $commons2_len/($str2_len) + ($commons1_len - $transpositions)/($commons1_len)) / 3.0;


}

function getPrefixLength( $string1, $string2, $MINPREFIXLENGTH = 4 ){

  $n = min( array( $MINPREFIXLENGTH, strlen($string1), strlen($string2) ) );

  for($i = 0; $i < $n; $i++){
    if( $string1[$i] != $string2[$i] ){
      // return index of first occurrence of different characters
      return $i;
    }
  }

  // first n characters are the same
  return $n;
}

function JaroWinkler($string1, $string2, $PREFIXSCALE = 0.1 ){

  $JaroDistance = Jaro( $string1, $string2 );

  $prefixLength = getPrefixLength( $string1, $string2 );

  return $JaroDistance + $prefixLength * $PREFIXSCALE * (1.0 - $JaroDistance);
}

?>
```

Listing of “ratcliffObershelp.php”, with embedded [16]

```php
<?php
/**
     *  compares two strings and returns longest common substring
     *
     *  Compares the two source strings character by character, captures every common substring
     *  between them, and returns the longest common substring found. Substrings of less than
     *  two characters long are ignored, and if there are multiple longest common substrings,
     *  the one that appears first in the first source string is returned.
     *
     *  @author Charlie Greenbacker charlie@artificialminds.net
     *
     *  @param $str1 - String - first source string for comparison
     *  @param $str2 - String - second source string for comparison
     *
     *  @return String - longest common substring of the two source strings
     */
    function longest_common_substring($str1, $str2)
    {
        $arySubstrings = array(); //stores all common substrings
        //iterate one-by-one through every character in both strings
        for ($i = 0; $i < strlen($str1); $i++) {
            for ($j = 0; $j < strlen($str2); $j++) {
                if (substr($str1, $i, 1) == substr($str2, $j, 1)) { //initial match found
                    $substring = substr($str1, $i, 1); //start with first 2 matching characters
                    /* $i_temp is used to move character-by-character in $str1 while keeping track
                     * of the starting position of the substring with $i
                     */
                    $i_temp = $i + 1;
                    $j = $j + 1; //move to the next character after the initial match in $str2
                    /* continue while subsequent character pairs match and the ends of both strings
                     * have not been reached
                     */
                    while (($str1{$i_temp} == $str2{$j}) && ($i_temp < strlen($str1)) && ($j < strlen($str2))) {
                        //append this matched character to the end of the substring
                        $substring .= $str1{$i_temp};
                        $i_temp++; //move to the next character pair
                        $j++;
                    }
                    $arySubstrings[] = trim($substring);
                }
            }
        }
        $arySubstrings = array_unique($arySubstrings); //remove duplicate common substrings
        /* return the longest substring in the array; if more than one are longest,
         * the first of them is returned
         */
        $strLCS = $arySubstrings[0];
        foreach ($arySubstrings as $strCurrent) {
            if (strlen($strCurrent) > strlen($strLCS)) {
                $strLCS = $strCurrent;
            }
        }
        return $strLCS;
    }



function Ratcliff($string1, $string2) {

    $blocks[0][0] = $string1;
    $blocks[0][1] = $string2;
    $m = 0;

    do {
        $words = array_pop($blocks);
        $common = longest_common_substring($words[0], $words[1]);

        if (!$common) {continue;}

        $m += strlen($common);

        $leftWord1 = trim(strstr($words[0], $common, true));
        $rightWord1 = trim(strstr($words[0], $common));

        $leftWord2 = trim(strstr($words[1], $common, true));
        $rightWord2 = trim(strstr($words[1], $common));

        for ($i=0; $i<strlen($common); $i++) {$rightWord1[$i]=""; $rightWord2[$i]="";}

        if ($leftWord1 && $leftWord2) {array_push( $blocks, array($leftWord1, $leftWord2) );}
        if ($rightWord1 && $rightWord2) {array_push( $blocks, array($rightWord1, $rightWord2) );}
    }
    while (count($blocks));

    $score = (2*$m) / ( strlen($string1) + strlen($string2) );
    return $score;

}

?>
```