Comparative Evaluation of String Metrics for Context Ontology Database

Farhanah Atiqah Norki, Radziah Mohamad and Noraini Ibrahim
Department of Software Engineering, Faculty of Computing,
Universiti Teknologi Malaysia, 81310 Johor Bahru, Johor, Malaysia.
farhanahatiqah@ymail.com

***Abstract*—Static Context Code Coverage Program (SCCCP) is a program developed to calculate the coverage of context code in a Java file of an Android application. The database built for SCCCP includes records on location and speech context, exclusive to Android. There is a huge need for string matching algorithm since strings from the source codes and database have to be checked for any similarity first before moving on to the calculation of context coverage. Therefore, three different string metrics were analyzed prior to choosing the most suitable one for SCCCP. In this paper, the results obtained from using Jaro-Winkler, Levenshtein, and Strike a Match string distance metrics are analyzed based on the task of matching the source codes with database records and other pair of strings. Some issues related during our experiment on source code matching are discussed in this paper. The findings conclude that Strike a Match algorithm is the best option since it gives the highest accuracy among others.**

***Index Terms*—Comparative Evaluation; Context Ontology; String Matching; String Similarity.**

I. INTRODUCTION

Context aware mobile application is highly capable in adapting itself according to its surrounding such as change in location, tracking user's movement, geofencing, adapting to situation with voice over command, sudden change in battery or power level, and adapting or interacting with other accessible devices. Since mobile application is a growing technology, some consideration should be taken during development and testing so that all aspects related to context-awareness can be developed and tested efficiently. Therefore, a context code coverage focusing on location and speech to help novice developer understand and easily identify the location and speech-based codes in the applications, as well as guiding them in writing better codes that could optimize context-awareness in mobile apps, is proposed. To achieve this goal, a similarity algorithm is needed to scan for necessary context codes inside a Java file. Then, the similarity between the codes and the records from database is calculated. When the similarity reaches a certain point, the codes are concluded to be context codes. Next, the context coverage present in mobile apps will be calculated. Three string metrics are compared in terms of precision, recall, and accuracy before the best is chosen to be implemented in SCCCP. Levenstein and Jaro Winkler are based on edit distances while Strike a Match is based on dice's coefficient. These string metrics are chosen because they measure the operation on string sequences and character composition. Figure 1 shows the design of the SCCCP.

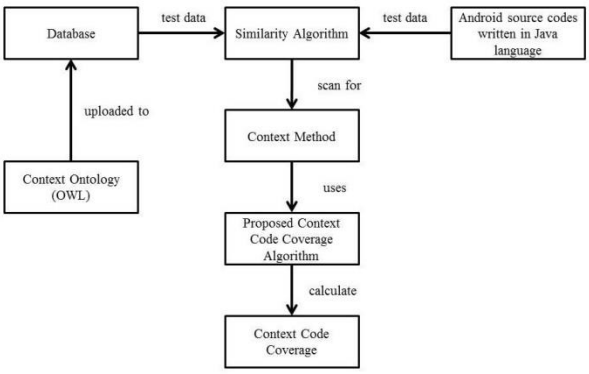

Figure 1: Static Context Code Coverage Program design

Similarity measurement between texts is highly important in research related to text and other similar works such as retrieving information, classification of text, document gathering, topic tracking and detection, machine translation, text summarization and others [1]. Two words are considered similar if they share the same pattern of alphabet in words, whether opposite of each other, or an inversion of each other. Experts from different fields like Mathematics and Computer Science propose different string metrics to calculate similarity between different strings such as approximate string matching, Bitap algorithm, Damerau–Levenshtein distance, edit distance, hamming distance, Jaro–Winkler distance, Lee distance, and Levenshtein distance. This paper nevertheless, will focus on Levenshtein, Jaro-Winkler, and Strike a Match.

The concept behind edit distance is to calculate the minimum number of operations taken to convert one string into another. It heavily concentrates on frequent typing errors, character insertion, omissions (deletion), substitutions, and reversals.

Levenshtein distance is used to measure the difference between two strings. The Levenshtein distance between two strings is the minimum number of change in single-character such as insertion, deletion, and substitutions [2, 3]. It works by changing one character into the other.

Jaro-Winkler, a variant of Jaro distance, focuses on duplication detection in two different strings [4]. The prefix scale used by Jaro-Winkler allows more appropriate assessment to strings that match from the start for a set of prefix length [4].

Simon White proposes a new algorithm based on lexical similarity, alteration in word's order, and language independence [5]. Lexical similarity is the degree of similarity measurement of sets of two given languages based on words.

The rest of the paper is organized as follows. Section II, III, and IV discuss the formula of the string metrics in details. Section V presents the evaluation results of the mentioned string metrics. Section VI discusses the experimentation results, and finally Section VII concludes the overall findings of this study.

II. LEVENSHTEIN

Levenstein distance counts the number of operations needed to match two strings [2]. The operations involved during transformation are insertion, deletion, or substitution. Figure 2 depicts the Levenshtein algorithm.

```
Set n as length of a.
Set m as length of b.
If n = 0, return m and exit.
If m = 0, return n and exit.
Construct a matrix; [0..m] as the row and [0..n] as the column.
Set the first row to 0..n.
Set the first column to 0..m.
Check one by one character in a (i from 1 to n).
Check one by one character in b (j from 1 to m).
If a[i] equals to b[j], the cost is 0.
If a[i] does not equal to b[j], the cost is 1.
Matrix d[i,j] is equal to the minimum of:
    a) Deletion: d[i-1,j] + 1.
    b) Insertion: d[i,j-1] + 1.
    c) Substitution: d[i-1,j-1] + cost.
Return d[n,m]
```

Figure 2: Levenshtein algorithm

From Figure 2, the difference in length between two strings is used to calculate the number of operation that take place to transform String 1 into String 2. If the length of String 1 is smaller than String 2, insertion and substitution will be performed. If the length of String 1 is bigger than String 2, deletion and substitution will be performed.

For characters' substitution, the number is calculated according to the following formula:

$$Min\ (String\ 1, String\ 2) - \sum \begin{cases} \exists x, \exists y, t\ if\ x = y & 0 \\ otherwise & 1 \end{cases} \qquad (1)$$

Levenshtein distance can be computed through dynamic programming using Wagner-Fischer algorithm for editdistance by initializing (n+1) x (m+1) matrix in a (m, n) cell where m and n are the lengths of both string. The matrix needs to be filled from upper left to the lower right corner.

Transition from one cell to another is parallel to insertion, deletion, or substitution. For each insertion, deletion, or substitution that occur, the cost is set to 1. If each character from two strings matches each other in respective sequence, it will return 0. Table 1 shows an example of comparison of two strings; 'abcdef' on X axis and 'agced' on Y axis, using dynamic programming.

Table 1
Two Pairs of Strings Used

		a	b	c	d	e	f
	0	1	2	3	4	5	6
a	1	0	1	2	3	4	5
g	2	1	1	2	3	4	5
c	3	2	2	1	2	3	4
e	4	3	3	2	2	2	3
d	5	4	4	3	2	3	3

The Levenshtein distance for turning 'agced' to 'abcdef' is 3. Replace g with b at position 2, insert d at position 3, and replace d with f at position 5.

1. a**g**ced ⟶ a**b**ced (g is replaced with b)
2. abced ⟶ abc**d**ed (insert d)
3. abcde**d** ⟶ abcde**f** (replace d with f)

Besides dynamic programming, Levenhstein distance can also be calculated using similarity measurement. The formula for similarity measurement in Levenshtein is presented below.

$$\mathrm{sim}_{\mathrm{ld}}(\mathrm{String\ 1}, \mathrm{String\ 2}) = 1.0 - \frac{\mathrm{dist}_{\mathrm{ld}}(\mathrm{String\ 1}, \mathrm{String2})}{\max(|\mathrm{String\ 1}|, |\mathrm{String\ 2}|)} \qquad (2)$$

By implementing the formula, the similarity measurement of string 'agced' and 'abcdef' is calculated to be 0.50. In second example, as shown in Figure 3, the similarity from one line of Android source codes with the records from database is calculated. String 1 is a line from Android source code and string 2 is a record from the context ontology database.

```
String 1 = locationManager.requestLocationUpdates(provider,MIN_TIME_
           FOR_UPDATE, MIN_DISTANCE_FOR_UPDATE, this);

String 2 = http://www.semanticweb.org/dell/ontologies/2016/4/
           untitled-ontology-24#requestLocationUpdates
```

Figure 3: Example of String 1 and String 2

Both strings have "requestLocationUpdates" as a part of them but because Levenshtein's operation only includes insertion, deletion, and substitution, these two are deemed to be only 0.19 similar. 81 operations are needed to transform String 1 into String 2.

Being an edit distance, Levenshtein is much more practical in detecting plagiarism in texts. Su et al. combine Levenshtein distance and Smith-Waterman algorithm for plagiarism detection [6]. They explore the use of diagonal line from Levenshtein distance and a simplified version of Smith-Waterman algorithm to identify and quantify local similarities in biological sequences [6]. Mihov et al. solve the problem of computing a suitable set of correction candidates in text correction by using Levenshtein automata, dynamic web dictionaries, and optimized correction models [7]. Hall et al. make an extension of Levenshtein that allows the calculation of different edit costs that is based on characters [8].

III. JARO-WINKLER

Jaro–Winkler, an extension of Jaro distance, utilizes the beginning of a scale, which allows better ratings to strings that match from the beginning [4]. It counts the usual character between two strings even though both of them are misplaced by a small distance.

A high Jaro score constitutes a substantial similarity between the strings. The formula for calculating Jaro score is depicted below.

$$d_j = \begin{cases} 0 & if\ m = 0 \\ \frac{1}{3}\left(\frac{m}{|s1|} + \frac{m}{|s2|} + \frac{m-t}{m}\right) & otherwise \end{cases} \qquad (3)$$

where: s1 = String 1
s2 = String 2
m = no. of matching character
t = half the no. of transposition

String 'agced' and 'abcdef' are used as example.

String 1= agced
String 2= abcdef

Jaro score:

$$\frac{1}{3}\left(\frac{4}{5}+\frac{4}{6}+\frac{4-1}{4}\right)=0.738$$

By applying the Jaro score into Jaro-Winkler formula as shown below,

$$d_w=\begin{cases} d_j & if\ d_j < b_t \\ d_{j+}\left(lp(1-d_j)\right) & otherwise \end{cases} \quad (4)$$

where: d_j = jaro distance of string s1 and s2
l = length of prefix at the start of the string up to maximum of 4 characters
p = is a continuous scaling factor for how much the score is adjusted upwards for having usual prefixes

$$0.738+\left(1\times 0.1\ (1-0.738)\right)=0.765$$

The similarity between s1 and s2 is calculated to be 0.765. This number is higher than Levenshtein calculation by 0.265 because Jaro-Winkler has consideration toward transposition of character. Figure 4 shows the algorithm for calculating Jaro-Winkler distance.

```
Set a as the length of s1
Set b as the length of s2
Set m as the number of matching characters
Set t as number of transposition divided by half
Construct a matrix; [0..a] as the row and [0..b] as the column
Set the first row to 0..a
Set the second column to 0..b
Check one by one character in s1 (i from 1 to a).
Check one by one character in s2 (j from 1 to b).
If a[i] equals to b[j], m is 1.
If a[i] does not equal to b[j], m is 0.
Find the number of transposition and divide it by half
Calculate the jaro score according to the given formula
Calculate the jaro distance according to the formula
```

Figure 4: Jaro-Winkler algorithm

Similar strings as in Figure 3 are calculated using Jaro-Winkler formula. The similarity is computed to be 0.449. At the same time, if both S1 and S2 are modified by moving *requestLocationUpdates* at the front, the similarity in calculation produced a slightly higher number which is 0.542, showing that Jaro-Winkler supports the suggestion that the similarity at the beginning of the string is more important than near the end of the string [9].

IV. STRIKE A MATCH

Strike a Match splits string into two character pairs. For example, string 'agced' is split into 4 parts; ag, gc, ce, ed whereas string 'abcdef' is split into 5 parts; ab, bc, cd, de, and ef.

Then, it will search for the same pair in the string. Below is the Strike a Match formula to calculate similarity. The formula focuses on lexical similarity, in other words, the overlapped between vocabularies.

$$similarity\ (s1,s2)=\frac{2\ x\ |pairs(s1)\cap pairs(s2)|}{|pairs(s1)|+|pairs(s2)|} \quad (5)$$

String 1= agced
String 2= abcdef

{ag, gc, ce, ed}
{ab, bc, cd, de, ef}

Number of overlapping string = 0

$$similarity\ (s1,s2)=\frac{2\ x\ |0|}{|4|+|5|}=0$$

The formula concludes that String 'agced' and 'abcdef' have zero similarity since no similar pair that overlaps each other is found, producing 0 lexical similarity.

In Strike a Match, if one of the strings is a random arrangement of the other string (anagram), it is usually considered as dissimilar. Besides lexical similarity, Strike a Match also gives fair similarity value for a string of different languages. For example, Republic of Ingushetia and Respublika Ingushetiya, both refer to the same republic but in different language. Republic of Ingushetia is in English whereas Respublika Ingushetiya is a direct translation from Russian Cyrillic. Computation of these two strings in Strike a Match concludes that these strings have 0.722 similarities.

It should be noted that Jaro-Winkler also computes a high similarity for 'Republic of Ingushetia' and 'Respublika Ingushetiya'. It is not uncommon since they refer to the same republic. However, 'Republic of Ingushetiya' and 'Republic of France' are two different nations yet the percentage of similarities calculated by Jaro-Winkler is 0.907 indicating that they are very similar, unlike Strike a Match which computes that both string is only 0.533 similar, hence give a better precision in accordance to words meaningfulness.

V. RESULT

In this section, Levenshtein, Jaro-Winkler and Strike a Match are evaluated in order to check their degree of suitability to be adapted in SCCCP. The pair of strings included are; (1) strings with similarity at the beginning, (2) strings with similarity at the end, (3) strings that contain the same character but have different arrangement, (4) random strings that have little to no similarity, and (5) sample strings from Android source codes and context ontology database. In string matching, one needs to be represented with approximate agreement's value. The maximum value for the approximate agreement is 1, indicating full agreement (fully similar), whereas the value between 0 and 1 indicates partial agreement (less similar). The precision, recall, and accuracy (F-measure) of five pairs are calculated and presented in Table 12.

The experiment was conducted on Eclipse since the program is written in Java language. Protégé is used to create the context ontology file. Then, by using XAMPP, the context ontology is imported into phpMyAdmin database, producing a large amount of entry records. Through programming, each line of codes in the Android java file and database's records are calculated one by one.

Table 2 shows pair of strings with similarity at the beginning. Five (5) pairs of strings is chosen. P1 and P2 have the same length. P3, P4, and P5 are strings with different length. The results of similarity calculation are presented in Table 3.

Table 2
String with Similarity at the Beginning

No.	String 1	String 2
P1	Sons	Sold
P2	Book	Boss
P3	Roast	Road
P4	The Fox Jumps	The Cow Lazes
P5	William	Wilhelmina

Table 3
Similarity Result (Min=0, Max=1) of String with Similarity at the Beginning

No.	Jaro-Winkler	Levenshtein	Strike a Match
P1	0.667	0.500	0.333
P2	0.667	0.500	0.333
P3	0.848	0.600	0.571
P4	0.692	0.538	0.250
P5	0.897	0.500	0.266

Table 4 shows the pair of strings with similarity at the end. The similar five (5) pairs of strings are also chosen for the experiment and the results of similarity calculation are presented in Table 5. Table 5 shows that Jaro-Winkler algorithm gives a lower similarity result than in Table 3 due to the facts that strings in Table 5 is dissimilar at the beginning.

Table 4
String with Similarity at the End

No.	String 1	String 2
P6	Bass	Toss
P7	External	Internal
P8	Mary Sew A Dress	Lisa Cut A Dress
P9	Augustus	Drautus
P10	Daughter	Grandmother

Table 5
Similarity Result (Min=0, Max=1) of String with Similarity at the End

No.	Jaro-Winkler	Levenshtein	Strike a Match
P6	0.667	0.500	0.333
P7	0.833	0.750	0.714
P8	0.650	0.563	0.444
P9	0.607	0.500	0.461
P10	0.645	0.364	0.125

Table 6, on the other hand, shows three (3) pairs of strings which contain the same character with different arrangement, while Table 7 presents the similarity calculation for these strings.

Table 8 shows three (3) pairs of random strings that have little to no similarity at all. P14 contains incomprehensible strings with no meaning. The similarity calculations for these strings are presented in Table 9.

Table 10 shows 10 pairs of sample strings from Android java file and context ontology database. String 1 is from source code whereas String 2 is from the database. Table 11, in contrast, depicts the similarity results from Jaro-Winkler, Levenshtein, and Strike a Match.

Table 6
Strings that Contains Same Character but Have Different Arrangement

No.	String 1	String 2
P11	The Brown Fox Jumped Over The Red Cow	The Red Cow Jumped Over The Brown Fox
P12	The Brown Fox Jumped Over The Red Cow	The Red Fox Jumped Over The Brown Cow
P13	Marry Had A Little Lamb	Little Marry Had A Lamb

Table 7
Similarity Result (Min=0, Max=1) of String that Contains Same Character but Have Different Arrangement

No.	Jaro-Winkler	Levenshtein	Strike a Match
P11	0.760	0.676	1.000
P12	0.800	0.784	1.000
P13	0.708	0.391	1.000

Table 8
Random String with Little to No Similarity

No.	String 1	String 2
P14	httpsabcdebdhdhlkkbbjj	klbdhjdhbnbbnbnbddhhhq
P15	keyboard keyboard	mouse mouse
P16	to be or not to be	that is the problem

Table 9
Similarity Result (Min=0, Max=1) of Random String with Little to No Similarity

No.	Jaro-Winkler	Levenshtein	Strike a Match
P14	0.470	0.045	0.190
P15	0.450	0.176	0.000
P16	0.640	0.263	0.000

Table 10
Sample String from Android Source Codes and Ontology Database

No.	String 1	String 2
P17	locationManager.requestLocationUpdates(provider,MIN_TIME_FOR_UPDATE, MIN_DISTANCE_FOR_UPDATE, this);	Uv:http://www.semanticweb.org/dell/ontologies/2016/4/untitled-ontology-24#requestLocationUpdates
P18	Intent i = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);	Uv:http://www.semanticweb.org/dell/ontologies/2016/4/untitled-ontology-24#Speech_recognition
P19	locationManager.requestLocationUpdates(provider,MIN_TIME_FOR_UPDATE, MIN_DISTANCE_FOR_UPDATE, this);	Uv:http://www.semanticweb.org/dell/ontologies/2016/4/untitled-ontology-24#Speech_recognition
P20	List<Address> addressList = geocoder.getFromLocation(latitude, longitude, 1);	Uv:http://www.semanticweb.org/dell/ontologies/2016/4/untitled-ontology-24#getFromLocation
P21	List<Address> addressList = geocoder.getFromLocation(latitude, longitude, 1);	Uv:http://www.semanticweb.org/dell/ontologies/2016/4/untitled-ontology-24#requestLocationUpdates
P22	location = locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);	Uv:http://www.semanticweb.org/dell/ontologies/2016/4/untitled-ontology-24#getLastKnownLocation
P23	List<Address> addressList = geocoder.getFromLocation(latitude, longitude, 1);	Uv:: http://www.semanticweb.org/dell/ontologies/2016/4/untitled-ontology-24#getLongitude
P24	public void onProviderEnabled(String provider) {	Uv::http://www.semanticweb.org/dell/ontologies/2016/4/untitled-ontology-24#getBestProvider
P25	isNetworkEnabled = locationManager.isProviderEnabled(LocationManager.NETWORK_PROVIDER);	Uv:http://www.semanticweb.org/dell/ontologies/2016/4/untitled-ontology-24#setAltitudeRequired
P26	for (int i = 0; i < address.getMaxAddressLineIndex(); i++) {	Uv:http://www.semanticweb.org/dell/ontologies/2016/4/untitled-ontology-24#setAddressLine

Table 11
Similarity Results (Min=0, Max=1) of Sample String from Android Source Codes and Context Ontology Database

No.	Jaro-Winkler	Levenshtein	Strike a Match
P17	0.444	0.160	0.304
P18	0.421	0.226	0.272
P19	0.462	0.120	0.160
P20	0.567	0.132	0.272
P21	0.557	0.155	0.222
P22	0.556	0.167	0.337
P23	0.562	0.170	0.225
P24	0.456	0.222	0.203
P25	0.530	0.147	0.195
P26	0.464	0.111	0.250

VI. DISCUSSION

Results from P01 to P10, and from P17 to P26, shows that Jaro-Winkler is found to be a better option at detecting similarity between strings compared to Levenshtein and Strike a Match. Whenwords in a string swap places as in Table 6, Levenshtein, that relies on the number of edits necessary to transform one string to another is too pessimistic. At the same time, since Jaro-Winkler considers transposition between letters, it is not precise enough in calculating similarity of two long strings. For example, in P15, both strings do not have any matching keywords but Jaro-Winkler calculates word's order in both strings even if they are arranged differently, concluding that both strings are almost half similar.

By using 10 samples from Android codes and content from database, the precision, recall and F-measure (accuracy) of the edit distances are calculated as shown in Table 12. Precision is the number of context correctly found whereas recall is the number of context and non-context correctly determined by the string metrics. The threshold value is set to 0.25, which means a pair of string with the value of 0.25 and above is regarded as a context string. Since some of the lines in the source codes contain longer string, a similarity of 0.25 or above is deemed as feasible. In all 10 strings, only P17, P20, P22, and P26 carry context information. The other pair of strings carries no context information. Interestingly, Jaro-Winkler deems all pair of strings as context, totally opposite to Levenshtein which deems all pair of strings as non-context. Strike a Match on the other hand, get all the context right except one in which it deems a non-context string as context. The result from P14 totally supports the conclusion that Jaro-Winkler, albeit being able to calculate even the slightest similarity, is not precise.

Table 12
Precision, recall, and F-value of Jaro-Winkler, Levenshtein, and Strike a Match based on Table 10

Threshold value	Jaro-Winkler			Levenshtein			Strike a Match		
	P	*R*	*F*	*P*	*R*	*F*	*P*	*R*	*F*
0.25	1	0.40	0.57	0	0.60	0	1	0.90	0.95

VII. CONCLUSION

Overall, it is found that Strike a Match holds the highest recall value since it can correctly identify between context and non-context string. Along with Jaro-Winkler, it has perfect precision in concluding whether the identified string is a context or non-context. However, Jaro-Winkler has the lowest recall percentage. At 0.25 threshold value, it identifies all pair of string as context. This could be contributed to the fact that Jaro-Winkler favours transposition between different characters in string, making a totally different string with almost similar characters (even though they are arranged differently), strikes a high percentage of similarity. String P14, P15, and P16 support this point. Jaro-Winkler also does not fare when it comes to accurately finding similar strings with meaningful information, unlike Strike a Match. With the highest streak of accuracy, Strike a Match is the best option for SCCCP.

Threat to validity is considered as limitation to this experiment. If another type of string criteria is added, it will not disrupt the current knowledge and results of the string metrics since the result obtained from the strings comparison is unique to their grouped criteria. In fact, adding another criteria may be a help itself since it could improve the knowledge on the string metric itself.

Further work will focus on exploring other string metrics such as Damerau-Levenshtein, in order to improve the performance of SCCCP.

ACKNOWLEDGMENT

We would like to thank Ministry of Higher Education Malaysia (MOE) for sponsoring the research through the FGRS grant with vote number 4F857 and Universiti Teknologi Malaysia for providing the facilities and support for the research.

REFERENCES

[1] W. H. Gomaa, and A. A. Fahmy, "A survey of text similarity Approaches," *International Journal of Computer Applications*, vol. 68, no. 13, pp. 13-18, Apr. 2013.

[2] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, *v*ol.10, no. 8, pp. 707-710, Feb. 1966.

[3] J. L. Peterson, "Computer programs for detecting and correcting spelling errors," *Communications of the ACM*, vol. 23, no. 12, pp.676-687, Dec. 1980.

[4] W. E. Winkler, "String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage, " in *Proc. of the Section on Survey Research Methods, American Statistical Association*, 1990, pp. 354-359.

[5] S. White, "How to strike a match," 2014, Retrieved October 11, 2016, from http://www.catalysoft.com/articles/StrikeAMatch.html.

[6] Z. Su, B. R. Ahn, K. Y. Eom, M. K. Kang, J. P. Kim, and M.K. Kim, "Plagiarism detection using the levenshtein distance and smith-waterman algorithm", in *3rd Int. Conf. on Innovative Computing Information and Control*, 2008, pp.569-569.

[7] S. Mihov, S. Koeva, C. Ringlstetter, K. U. Schulz and C. Strohmaier, "Precise and efficient text correction using levenshtein automata, dynamic web dictionaries and optimized correction models," in *Proc. of Workshop on International Proofing Tools and Language Technologies*, Patras, 2004, pp. 1-10 .

[8] P. A. V. Hall, and G. R. Dowling, "Approximate string matching," *ACM Computing Surveys*, vol. 12, no. 4, pp.381-402, Dec. 1980.

[9] J. J. Pollock, and A. Zamora, "Automatic spelling correction in scientific and scholarly text," *Communications of the ACM*, vol. 27, no. 4, pp.358-368, Apr. 1984.