CIRRELT

CIRRELT-2023-08

Lookup Table String Similarity Algorithm

Banafsheh Mehri
Yves Goussard
Martin Trépanier

January 2023

Bureau de Montréal
Université de Montréal
C.P. 6128, succ. Centre-Ville
Montréal (Québec) H3C 3J7
Tél : 1-514-343-7575
Télécopie : 1-514-343-7121

Bureau de Québec
Université Laval,
2325, rue de la Terrasse
Pavillon Palasis-Prince, local 2415
Québec (Québec) G1V 0A6
Tél : 1-418-656-2073
Télécopie : 1-418-656-2624

Lookup Table String Similarity Algorithm

Banafsheh Mehri[1,2,*], Yves Goussard[2], Martin Trépanier[1,3]

[1] Interuniversity Research Centre on Enterprise Networks, Logistics and Transportation (CIRRELT)

[2] Department of Electrical Engineering, Polytechnique Montréal

[3] Department of Mathematics and Industrial Enginnering, Polytechnique Montréal

Abstract. String similarity algorithms are mainly used for measuring similarity/dissimilarity between words for comparison and approximate matching in many domains such as document clustering, fraud detection, word-sense disambiguation for information retrieval and many more applications. Although this field of research is well explored, most of the existing works rely on optimizing result accuracy of such metrics whereas the trade-off between speed of the measurement process (especially when operating on large data), sensitivity of metrics with respect to the thresholds and discriminatory power of similarity metrics are mostly ignored. The main objective of this paper is to introduce a new algorithm: Lookup Table String Similarity (LTSS), which provides an interesting trade-off between speeding up the process, reducing the sensitivity to thresholds and increasing the ability to put similar words close to each other when performing clustering as well as avoiding being too discriminative to separate highly similar words. We also perform an evaluation of a set of string similarity metrics to show the performance of our algorithm. Our experiments indicate that the proposed method significantly outperforms the existing metrics in terms of computational efficiency while exhibiting adequate performance with respect to sensitivity, inter- and intra-bucket clustering. These characteristics make LTSS algorithm an attractive candidate for computing similarity distance when performing text classifications/clustering specially on very large datasets.

Keywords: String similarity, misspelling, approximate matching, document clustering.

Acknowledgements. The authors gratefully acknowledge the financial support provided by the Natural Sciences and Engineering Council of Canada (NSERC), and the Société de transport de Montréal (STM) for data provisioning.

Results and views expressed in this publication are the sole responsibility of the authors and do not necessarily reflect those of CIRRELT.

Les résultats et opinions contenus dans cette publication ne reflètent pas nécessairement la position du CIRRELT et n'engagent pas sa responsabilité.

---

* Corresponding author: banafsheh.mehri@polymtl.ca

Dépôt légal – Bibliothèque et Archives nationales du Québec
Bibliothèque et Archives Canada, 2023

© Mehri, Goussard, Trépanier and CIRRELT, 2023

1 Introduction

In textual data, measuring the similarity between the texts is done by calculating the distance between their strings based on a similarity metric. Similarity metric algorithms are widely used in Information Retrieval (IR) systems in order to enable searching for information in databases, documents or linked data and they are used in various specific applications such as clustering or matching entity names [13] [7] [12], spell error detection and normalization of micro texts [44], data cleaning and duplicate detection [6] [28] and many more.

To showcase the importance of similarity measures on the performance of similarity metric algorithms when dealing with real-life data sets,, Lopez et al. [26] have shown that choosing an efficient similarity measure can help producing an ontology-based question answering system that supports query disambiguation, knowledge fusion, and ranking mechanisms, to identify the most accurate answers. In another effort, Ngonga Ngomo and Auer [33] have revealed the impact of similarity measures in semantic web. They developed an approach for the large-scale matching of instances in metric spaces for the discovery of links between knowledge bases on the linked data.
With regard to the performance evaluation, string similarity measures used in many applications are usually evaluated in terms of their ability to retrieve relevant and accurate information regarding a text string query. Papadimitriou et al. [35] discussed the possible effects of sensitivity of similarity functions on web graphs that would associate to the quality of search results in search engines. They have shown that by using a similarity function that is more sensitive to changes in high-quality vertices in web graphs, results of a search query can be significantly improved.

Although string similarity metrics are amongst well studied fields of research, most of these studies are focused on the precision of such algorithms while other characteristics such as swiftness, sensitivity to thresholds and desired discriminatory power are in most cases are overlooked. This research proposes a new string similarity algorithm that adapts to the situations where those above-mentioned characteristics of an algorithm are unavoidably required in real-world applications. Our proposed algorithm has been evaluated through a comparison with several existing algorithms. The experimental results depict that it outperforms the existing algorithms with regard to the time efficiency while obtaining comparable results in terms of accuracy, which makes it of great interest to computational intensive applications when working with massive amount of data.

The rest of this paper is organized as follows. Section 2 points out to the commonly used string similarity metrics. Section 3 introduces our proposed string similarity algorithm, the *Lookup Table String Similarity* algorithm. Sections 4 describes the experimental setup for the implementations and the performance comparisons. Section 4 also includes the description about the data used in this study, the methodology to conduct the performance comparisons and the metrics used to evaluate the experiments. Section 5 discusses the results of such experiments and Section 6 includes the conclusion.

2 Related Work

Finding similarity between words is the fundamental part of similarity between sentences, paragraphs and documents. String-based similarity algorithms operate on string sequences and character composition. These measures usually map a pair of strings s and t to a real number r, where a smaller value of r indicates greater similarity between s and t. Gomaa and Fahmy [16] grouped string similarity measures in two categories: *character-based similarity measures*, which consider distance as the difference between characters of strings (thus useful in the case of typographical errors) and *term-based similarity measures*, which take into the account the distance between the two terms.
These types of categorization are mostly addressing the problem of document clustering. We can also group string similarity measures in *edit-based similarity measures*, *token-based similarity measures* and *hybrid similarity measures*. Edit-based similarity measures are characterized to process and evaluate the contrast between strings as a weighted aggregate of the quantity of additions, eliminations, substitutions and additionally transpositions needed to obtain the second string from the first one. The distance is then the cost of best sequence of edit operations that convert s to t. *Levenshtein*, *Smith-Waterman* and *Jaro–Winkler* are examples of edit-based similarity measures.

Token-based similarity measures essentially first try to decompose texts into token sets[1] sets to use tokens rather than complete texts, and then compute the similarity based on the token sets. Usually, two similar strings end up having a large overlap in their token sets. Nonetheless, Token-based similarity measures are not quite efficient to calculate similarity when typos and misspelling words are introduced [12]. In general, such measures suffer from the limitation that they only consider exact match of two tokens in bag of words, hence ignoring string fuzzy matches. *Jaccard*, *Sorensen-Dice* and *Cosine* are popular examples of token-based similarity measures.

There also exist *hybrid similarity measures* which combine the benefits of edit-based and token-based methods. When more control is needed over the similarity measure, hybrid algorithms can be effective. Unlike edit-based measures [48] [42], hybrid measures can be used for matching an attribute value to its abbreviation or acronym. *Monge-Elkan* belongs to this group of similarity measures.

Here, for a better understanding of the sequence, we will briefly describe the main algorithms of each class:

Levenshtein [24] is a commonly used similarity measure that describes the distance between two strings by checking the base number of operations expected to change one string into the other, where an operation is defined as an addition, cancellation, or substitution of a character, or a transposition of two nearby characters [24]. In other words, The Levenshtein distance between two strings is the minimum number of edits needed to transform one string into the other, with the permissible operations being: insertion,

[1] A token or q-gram is a character string of length q

deletion, or substitution of a single character. The Levenshtein distance between two strings s and t (of length $|s|$ and $|t|$ respectively) is given by $lev_{a,b}(|a|, |b|)$ where

$$lev_{s,t}(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0 \\ min & \begin{cases} lev_{s,t}(i-1,j)+1 \\ lev_{s,t}(i,j-1)+1 \\ lev_{s,t}(i-1,j-1)+1_{(s_i \neq \mathrm{t}_j)} \end{cases} \quad \text{Otherwise} \end{cases}$$

where $1_{(s_i \neq t_j)}$ is the indicator function equal to 0 when $s_i = t_j$ and equal to 1 otherwise, and $lev_{s,t}(i,j)$ is the distance between the first i characters of s and the first j characters of t. i and j are 1-based indices.
[4], [40], [39] and [25] are some of many applications of the Levenshtein similarity metric such as comparing movement patterns detection, plagiarism detection, measuring errors in text entry tasks and comparing sequence information in animals' vocalizations.

Smith-Waterman [36] is a famous algorithm for acting nearby sequence alignment to find the best alignment over the conserved domain of two sequences [36]. The Smith-Waterman algorithm compares segments of all possible lengths and optimizes the similarity degree. More specifically, this algorithm determines the sequence of operations needed to transform one string to another, but attributes lower weights to transformations among similar-sounding characters and employs specialized logic for handling alignment gaps such as a "gap start" penalty corresponding to the beginning of a string of unmatched characters, and a separate "gap continuation" penalty for its continuation. Instead of looking at an entire sequence at once, the Smith-Waterman algorithm compares multi-lengthed segments, looking for whichever segment maximizes the scoring measure. The algorithm itself is recursive in nature:

$$H_{ij} = maxH_{i-1,j-1} + s(a_i, b_j); H_{i-k}, j - W_k; H_{i,j-1} - W_1; 0$$

Where s = $a_1 \ldots a_K$ and t = $b_1 \ldots b_L$ are the sequences to be aligned, K and L are the lengths of s and t respectively. $s(s,t)$ is the similarity score of the elements that constituted the two sequences and W_k as the penalty of a gap that has length k. $H_{i-1,j-1}$ + $s(a_i, b_j)$ is the score of aligning a_i and b_j. $H_{i-k,j}$ - W_k is the score if a_i is at the end of a gap of length k. $H_{i,j-1}$ - W_1 is the score if b_j is at the end of a gap of length l and 0 means means there is no similarity up to a_i and b_j.
[20], [14], [18] and [47] are some examples of applications of this similarity metric on plagiarism and collusion detection, automatic traffic signature extraction, early illness recognition and its usage in genomic applications.

Jaro–Winkler [43] is, basically, an extension of Jaro distance [22]. In theory, Jaro distance is identifiable as the minimum number of single-character transpositions required to change one string into the other, whereas Jaro-Winkler distance utilizes a prefix that

establishes more favorable weights to strings that match from the beginning for a set of prefix length. Given strings $s = a_1 \ldots a_K$ and $t = b_1 \ldots b_L$, define a character a_i in s to be common with t, suppose that $H = \frac{min(|s|,|t|}{2}$, there is a $b_j = a_i$ in string t such that $i - H \leq$j$\leq$i+H.
Now suppose that the characters in s common with t and vice versa be $s' = a'_1 \ldots a'_K$ and $t' = b'_1 \ldots b'_L$. Now define a transposition for s' and t' The Jaro similarity for s and t is:

$$Jaro(s,t) = \frac{1}{3}.(\frac{|s'|}{|s|} + \frac{|t'|}{|t|} + \frac{|s' - T_{s',t'}|}{|s'|})$$

Given P the length of the longest common prefix of s and t, let $P' = max(P, 4)$, then the Jaro-Winkler distance between s and t is:

$$Jaro - Winkler(s,t) = Jaro(s,t) + \frac{P'}{10}.(1 - Jaro(s,t))$$

Some applications of Jaro-Winkler similarity metric are such as duplicate detection in health related records [1] and entity linking in Twitter data[8].

Jaccard [21] is mostly used in document similarity applications. Jaccard index refers to the ratio of the size of the intersection of two strings to the size of their union [21]. In order to use this algorithm, a document, typically, must be presented as a bag of words which is the list of unique words in it, then we can compute Jaccard index between them. The Jaccard similarity between the word sets s and t is simply:

$$Jaccard(s,t) = \frac{|s \bigcap t|}{|s \bigcup t|}$$

Jaccard similarity coefficient has been used in many real word problems such as in recommender system [46], distributed genome comparisons [5] and application level traffic classification [10].

Sorensen-Dice [38] works by comparing the number of identical character pairs between the two strings [37]. It is often called *Sørensen index* or *Dice's Coefficient*. For two sets of strings s and t, the Sorensen-Dice distance is defined as:

$$d(s,t) = 1 - \frac{2 * |s| \bigcap |t|}{|s| + |t|}$$

Livestock emission characterization [19], biogeographic classifications [32] and anomaly-based intrusion detection in system activities [34] can be mentioned as some real world applications of Sorensen-Dice similarity.

Cosine is a very famous string similarity measure, extensively used in document similarity in information retrieval domain [2] and clustering [23]. Cosine similarity measure determines the cosine of the angle between two vectors. Once the strings are transformed in vectors of occurrences of sequences of k characters, the similarity between them will be the cosine of their respective vectors. Given two string sets of s and t, the cosine similarity between the two sets is defined as:

$$\cos\theta = \frac{\overrightarrow{s}.\overrightarrow{t}}{\|s\|\|t\|}$$

Cosine similarity is the same concept as TFIDF which is widely used in the information retrieval problems.

Monge-Elkan [30] computes the average of the similarity values between the more similar token pairs in two strings s and t. This algorithm was introduced by Monge and Elkan [31] and it has been used in many name-matching and record linkage comparative studies [6] [7]. This hybrid method maintains the properties of the internal character-based measure, the ability to deal with misspellings, typos, OCR errors, and deals successfully with missing or disordered tokens. Given string sets of s, t and their substrings $s = a_1 \ldots a_K$ and $t = b_1 \ldots b_L$, the Monge-Elkan algorithm measures the average of the similarity values between pairs of more similar tokens within string sets s and t. The Monge-Elkan similarity is defined as:

$$sim(s,t) = \sum_{i=1}^{K} \max_{j=1}^{L} sim'(A_i, B_j)$$

Where sim' is some secondary distance function.

The main advantage of this algorithm is being recursive, which gives an ability to handle sub-fields or sub-sub-fields, meaning that the algorithm is more likely to find a match between a string and its corresponding incomplete string in several formats. Ontology alignment [41], biomedical abbreviation clustering [45] and title matching [15] can be mentioned as some examples of application of this similarity metric.

3 Lookup Table String Similarity Algorithm

We introduce Lookup Table String Similarity (LTSS) algorithm which is a method based on pairwise comparison of the components of two strings s and t to find the similarity. Our newly proposed algorithm splits the original problem into smaller sub problems to compute the similarity of two strings converted to integer arrays corresponding to the indices of the components of each string by yielding scores to matches and mismatches.

The LTSS algorithm is mainly based on constructing the similarity matrix of the components of the strings, which is the calculated cost of changing one letter to another.

For certain letters, the matrix values are initialized by predefined weight values. The scoring metric used in this algorithm is sensitive to detect changes and swaps in the components of strings and it penalizes the comparison. Besides the penalty on changing and swapping the components of strings, additional penalty will be added based on the first letters of the two strings s and t. For instance, it adds up a weight for a change from the letter "K" in "Katerine" when compared with the letter "C" of "Catherine", meaning that instead of putting zero for the similarity of such two letters, it detects the change and adds a non-zero value. It also considers that the shift might have been occurred in more than *N* letters, equal to the defined radius. The algorithm then iterates over all the indices in a radius and add up a penalty based on the number of shifts and compares the two letters within the radius in both strings. Finally, it finds the minimum distance within the radius and returns it. Algorithm 1 illustrates the pseudo-code of LTSS algorithm.

The steps of our proposed method can be summarized as:

1. Initializing parameters:
   - *radius*: radius of the filter for the comparison of pairwise letters
   - *costShift*: cost of swapping two letters
   - *costDiff*: cost of the difference of two letters
   - *costFirst*: cost of the difference of the two first letters
2. Compute the difference between size of the two strings and return zero if it is greater than the radius.
3. Convert the two strings to indices of letters.
4. Construct similarity matrix of letters, which contains the cost of changing one letter to another. For certain letters, we initialize the matrix values by specific weight values.
5. Compute a penalty related to the size difference of the two strings, replacing a letter by another, the penalty related to the shift between letters, and an additional penalty on the first letter.
6. Iterate through the letters of the shortest string, and generate a vector storing all the distances between pair letters within a radius.
7. Compute the distance between letters of the first index.
8. Iterate over all the indices in a radius and add up a penalty based on the number of shifts.
9. Perform the comparison of two letters within the radius in both strings.
10. Find the minimum distance within the radius and return it.

4 Experimental Setup

In this section, first we explain the steps to prepare the data and the gold standard that are used in this experiments. Then, we discuss our methodology regarding the implementation of an indirect comparison between selected similarity algorithms and LTSS and different parts of the evaluation process.

Algorithm 1 Lookup Table String Similarity algorithm

procedure SIMILARITY($string1$, $string2$)
  $SETradius$
  $SETcostShift$
  $SETcostDiff$
  $SETcostFirst$
  $difference\_Sizes = |string1.length - string2.length|$
  if $(differencesizes > this.maxDiff)$ **then**
    return 0
  end if
  $ind1 \leftarrow getIndices(string1)$
  $ind2 \leftarrow getIndices(string2)$
  $WTB \leftarrow loadthetableofweights$
  $penalty = \frac{costDiff \times max(string1.length, string2.length)}{min(string1.length, string2.length)}$
  $dist \leftarrow 0$
  for $i = 0 \rightarrow min(string1.length, string2.length)$ **do**
    $distRadius = vector[4 \times radius + 1]$
    if $(i == 0)$ **then**
      $distRadius.add(WTB[ind2[i]][ind1[i]] \times costFirst)$
      $ElsedistRadius.add(WTB[ind2[i]][ind1[i]])$
    end if
    for $j = 0 \rightarrow radius$ **do**
      $penaltyShift = costShift \times |j - i|$
      $distRadius.add(WTB[ind2[i]][ind1[i - j]] + penaltyShift)$
      $distRadius.add(WTB[ind2[i]][ind1[i + j]] + penaltyShift)$
      $distRadius.add(WTB[ind1[i]][ind2[i - j]] + penaltyShift)$
      $distRadius.add(WTB[ind1[i]][ind2[i + j]] + penaltyShift)$
    end for
    $dist+ = min(distRadius)$
  end for
  return $dist$
end procedure

4.1 Data Preparation

To conduct this study, we used WikEd Error Corpus [17]. This dataset contains large and diversified records extracted from the Wikipedia revision history using data mining techniques. It is a freely available corpus including 12,130,508 pairs of edited sentences from the English version of Wikipedia and a total of 14 million edits of various types. The edits include:

- grammatical error corrections
- stylistic changes
- sentence rewordings and paraphrases:
- spelling error corrections
- encyclopaedic style adjustments
- time reference changes
- information supplements

– numeric information updates
– item additions/deletions to/from bulleted lists
– etc ...

Although there are many types of edits in this corpus, the scope of our research has to be limited to spelling error corrections. Since the data is too noisy and the variability of texts is high, we perform an error selection process including two steps of extraction and cleaning. First, for the extraction steps, we implemented a program that crawls the raw data and extracts only the spelling error corrections amongst all the error correction types. The spelling error corrections are presented inside the logs with the format: *[-donload-] [+download+]*. Basically, this format indicates that the word between the two minuses was misspelled and replaced with the word between the two pluses.

Second, in the cleaning step, we filtered out texts consisting of stop words such as ”and”, ”then”, ”when” etc, and those including numbers, digits and special characters. Even though this refinement process reduced the size of the dataset, the remainder still contained a very large amount of records. Since the average length of a word in most English documents is over 5 characters [2], we eliminated non-English words and those with less than five characters. Finally, a subset of 10,000 misspelled words was assembled to perform the study. A gold standard was brought together by merging the results of forming 777 groups of similar words by performing several manual adjustment and verification processes. Each group was carefully edited by hand and repeatedly verified to ensure having a reliable solution file. The biggest group contained 40 words and smallest one had only 4 words.

4.2 Methodology

To compare the performance of LTSS algorithm to the selected similarity measures, we implemented a program that generates clusters, which we call *Buckets*, and then estimated the correctness of the obtained results by comparing them to the gold standard. In this setting the evaluation is mainly provided by casting the task as a clustering problem under the hypothesis that similar strings will end up in the same cluster using each string similarity algorithm. If the overall performance of an string similarity algorithm is good in terms of putting very similar words in one cluster and putting very dissimilar words in different clusters, then it would be the same case for any 1:1 matching strings. The operational methods are demonstrated in Figure Figure 1, and can be summarized as follows:

– Creating the similarity matrix: This corresponds to a similarity graph (based on each string similarity algorithm) with data points for nodes and edges whose weights are the closeness between data points represented by a value between 0 and 1. We used a Java library called SimMetrics [9] that contains the implementations of string similarity algorithms in order to develop an application that performs pairwise comparisons to compute such matrix for all of the algorithms.

[2] http://www.wolframalpha.com/input/?i=average+english+word+length

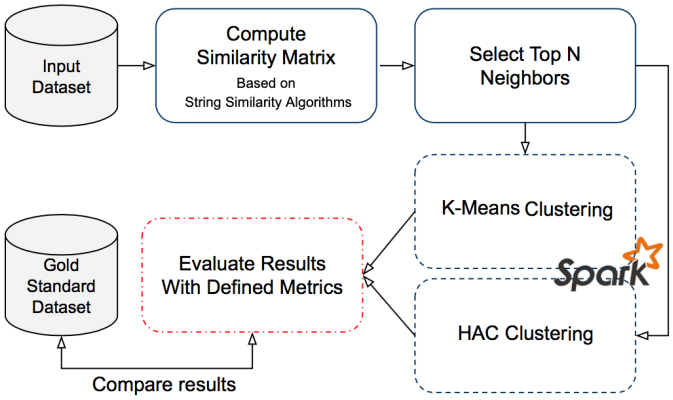


Fig. 1: Methodology steps

– Detecting outliers in data points: We keep only the most similar neighbors of each data point using a nearest neighbor based outlier detection technique [3]. For this purpose, we set the pruning threshold to the weakest outlier, then we calculate the nearest neighbors for a data instance, and we set the outlier threshold for any data instance to the score of the weakest outlier found. This removes instances that are close to the outliers, and hence not interesting to be included in the flowing steps.

  By plotting the histograms of similarities we found out the best value of the threshold that covers the larger proportion of data and cuts out the lower, in order to filter out weak similarities and reduce the size of the matrix. In other words, we keep the strongest edges of the similarity graph. We repeated the procedure with many iterations to figure out the best value of N based on obtained results.

  It is important to mention that it was not necessary to normalize the matrices because the values are within a fixed range of zero to one. Hence, re-normalization would shrink matrices, but there would have no effect in regard to the final results. However, we considered the similarity distance matrix as a graph with connected nodes (distance values) and we decided to keep only strong edges that correspond to higher similarity distances i.e. $distance = (1 - similarity)$ between data points in our matrices. This process not only reduced the size of matrices and calculation time consequently, but also helped to cut out most of the outliers, hence resulting in a better accuracy. The top N neighbors were chosen based on a careful analysis of the proportion of data that seemed to be outliers. We plotted the frequency of similarity values and filtered out only the segment containing the data that was associated with less occurrence and higher similarity distance (between 0.6 and 1 in most cases).

– Implementing K-Means clustering [27] to build *buckets* of similar words by using each string similarity algorithm. K-Means requires setting the parameter for num-

ber of buckets, and in our experiments we did set this number to number of buckets in our gold standard in order to compare our results with the solution file.

– Implementing Hierarchical Agglomerative Clustering (HAC) [11] to create buckets with no a priori information about the number of clusters required. Each data point is allocated to its own bucket and after that the calculation proceeds iteratively, joining the two most comparative buckets at each progression step, and continues until there is only one bucket.

– Some metrics were used to evaluate the obtained results. In the following section, we explain in details such metrics and the purpose of choosing them.

4.3 Evaluation Metrics

The evaluation is mainly provided by casting the task as a clustering problem under the hypothesis that similar strings will end up in the same cluster using each string similarity algorithm. If the overall performance of an string similarity algorithm is good in terms of putting very similar words in one cluster and putting very dissimilar words in different clusters, then it would be the same case for any 1:1 matching strings.
To evaluate the performance of the LTSS algorithm through the clustering task and to assess the obtained results we developed several quantitative metrics, hence realizing indirect comparison tests between the similarity algorithms and the LTSS algorithm:

– *Intra-buckets distance*:
– *Intra-buckets distance*: Intra-bucket distance is a measure of compactness that calculates within set sum of squared errors. This measure is the sum of squares of distances between the points of each bucket and the corresponding bucket center. Smaller value for the within set sum of squared Errors is more desirable as it shows that the string similarity algorithm considers words with minimum dissimilarity between them as similar words. Since the dataset contains large volume of data, we used the Apache Spark machine learning library [29] to rapidly compute this metric.
– *Inter-buckets distance*: Inter-buchet distance is the measure of separation. It measures the distances between buckets' centers by calculating the sum of distances between the centroids and the total sample mean multiplied by the number of points within each cluster. Inter-bucket distances represent the dissimilarities between the buckets, therefore, an efficient string similarity algorithm tends to maximize this value in order to come up with buckets that their centers are distant from each other. This metric and the previous metric attribute to the compromise sought between minimizing intra-bucket distance and maximizing inter-bucket distance that corresponds to the discriminatory power of the algorithm used as the string similarity measure. Ideally, the selective force of the algorithm must be sufficiently strong to result in minimal internal and maximal external distances.

– *Number of buckets per threshold*: Using this metric helps understand the sensitivity of each algorithm with respect to changes in thresholds. An efficient metric is expected to avoid being too sensitive to threshold, meaning that it should not generate considerable different number of buckets when the threshold is changed insignificantly. This feature of the similarity algorithm has a direct effect on the end results when applied to an automated machine learning system, since it renders the learning process overly reliant on the threshold.

– *Total execution time*: The overall time that it takes to generate buckets is calculated in milliseconds. This metric shows the efficiency of string similarity algorithm in terms of computational cost. Lower execution time makes the algorithm more adoptable and applicable in variety of domains with high interest in less computational cost. All experiments has been conducted 30 times and the average amount is reported.

5 Results

Our results enabled us to draw conclusions on the efficiency of our novel string similarity algorithm in comparison with broadly used ones. We discuss the outcome of experimental analyses in the direction of specified characteristics that make an algorithm effective while operating on expansive data sets such as bibliographic databases. Our experiments were performed using an Intel Xeon 2.6 GHz computer with 64 Gig of RAM.

Figure 2 shows the different values of within set sum of squared error resulting from K-Means clustering on each similarity matrix that is generated by the selected string similarity algorithms.
The results of computing within set sum of squared error showed that the performance of these methods covers a wide range. Smith-Waterman and Monge-Elkan showed higher errors in terms of within set sum of squared error. Our proposed algorithm outperforms Cosine, Sorensen-Dice and Jaccard, surpasses the hybrid algorithm of Monge-Elkan, and Smith-Waterman. the Levenshtein algorithm, the Jaro-Winkler algorithm and the LTSS algorithm have the least within set sum of squared error compared to all other algorithms.

We examined all the buckets created by each algorithm and found out that for instance some very similar misspelled words like: ”achievedto”, ”chievment” and ”unachieved” are put together in one bucket when we applied the LTSS algorithm, whereas all other algorithms distinguished them by putting into separate buckets.

We observed the buckets created by each algorithm and found out that for instance some very similar misspelled words like: ”achievedto”, ”chievment” and ”unachieved” are put together in one bucket when we applied the LTSS algorithm, whereas all other algorithms distinguished them by putting into separate buckets.

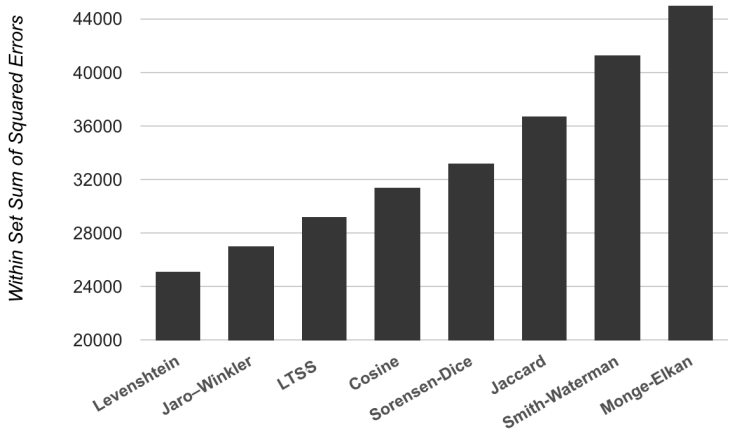


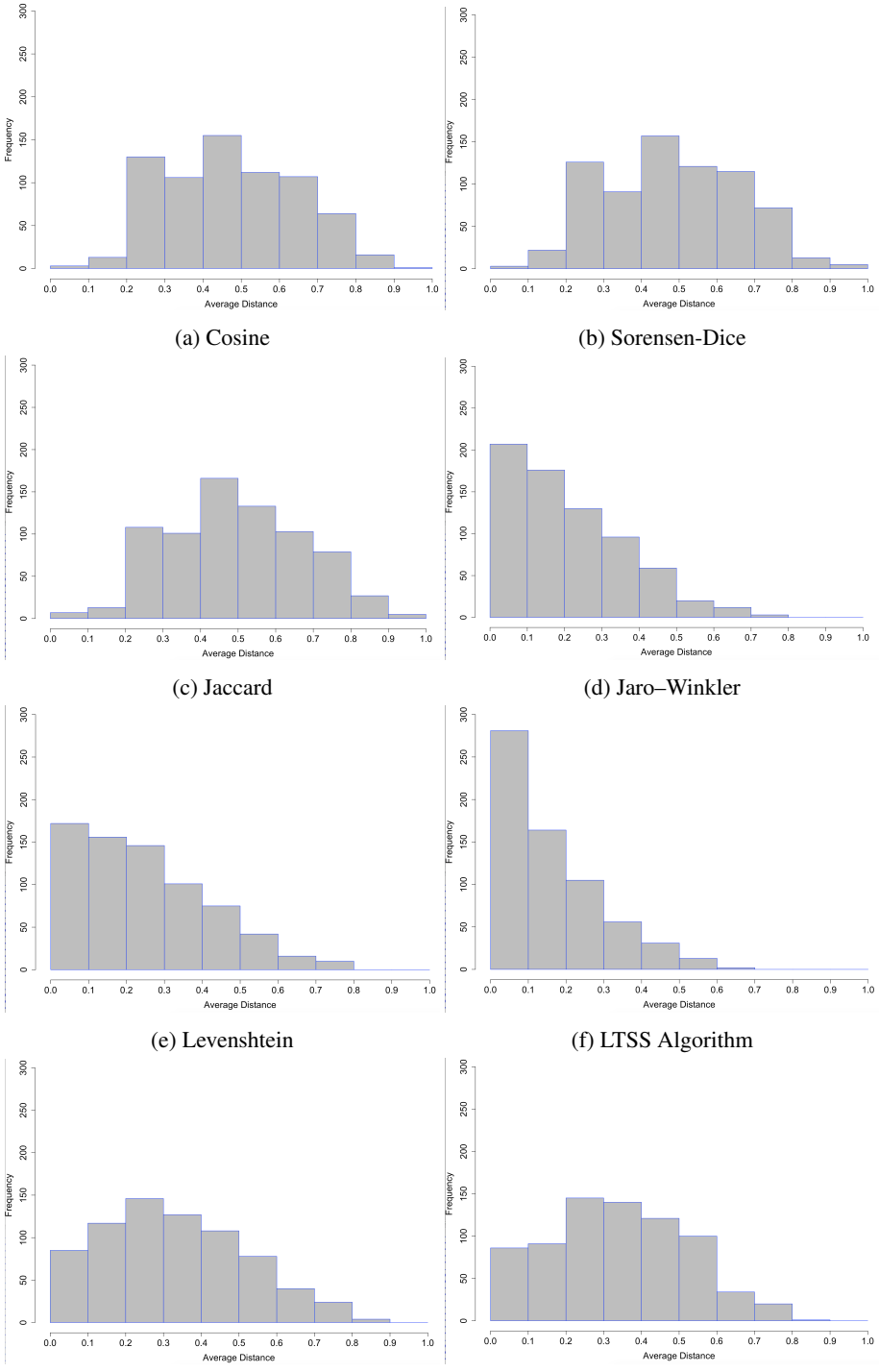
Fig. 2: Intra-buckets distances

Although the Levenshtein algorithm produced lower error than the LTSS algorithm, it considers the above-mentioned words not similar because of not containing the first and the last exact similar characters, which makes the discriminatory power of this algorithm questionable. When the first letters of the two compared strings are completely different, the Levenshtein algorithm distinguishes them with reporting less similarity value, whereas our method detects the shift in letters and returns more precise value for the similarity score.

As an example, the LTSS algorithm computes the similarity between the words ”chmith” and ”shmith” as: ”0.9325”, while the Levenshtein algorithm return the value of ”0.8333” for the similarity score; because of containing two different letters of ”c” and ”s” at the beginning of each word. Additionally, the LTSS algorithm considers words such as ”atlanica”, ”atlantic” and ”antlatic” as similar words, however, the Levenshtein algorithm returns words such as ”arbania”, ”astanga”, as similar words for ”atlanica”.

Figure 3 includes the histograms of all average distance values of elements inside buckets produced by each algorithm. Our results indicates that the Jaro-Winkler algorithm, the Levenshtein algorithm and the LTSS algorithm show better results compared to others since the distribution of inter-bucket average distance values are biased toward the minimum values. On the contrary, the rest of algorithms show tendency toward an average distance of 0.4 to 0.6 between words in buckets. In contrast, the LTSS algorithm results in creating buckets with almost less than 0.6 average distance between words in them.

Looking inside the buckets we can see the difference between the two algorithms’ results. For an example, the Sorensen-Dice algorithm considers the two words ”substration” and ”altrations” as similar words, whilst the LTSS algorithm separated them and



(a) Cosine

(b) Sorensen-Dice

(c) Jaccard

(d) Jaro–Winkler

(e) Levenshtein

(f) LTSS Algorithm

(g) Monge-Elkan

(h) Smith-Waterman

Fig. 3: Intra-buckets average distances

grouped ”substration” with words such as ”subtraction”, ”sebtraction”, etc., and ”altrations” was located inside a bucket with ”altraction”, ”aterations”, etc . as similar words.

Moreover, we consider an algorithm as successful if it also shows higher between clusters distance. The results of inter-bucket distances are illustrated in Figure 4. The Monge-Elkan algorithm shows the least between cluster sums of squares value. The highest value of between cluster sums of squares belongs to the Levenshtien algorithm followed by the LTSS algorithm which seems to outperform the Jaro-Winkler and the rest of algorithms considering the discriminatory power with regard to inter-bucket distances.

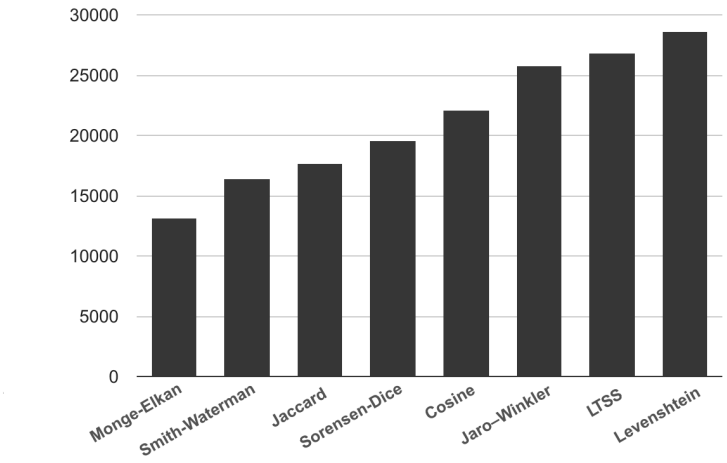


Fig. 4: Inter-buckets distances

While k-means tries to advance optimization to reach a global optimal, agglomerative hierarchical clustering aims at searching for the best stride at each group combination, running insatiable calculation, which is done precisely yet bringing a sub-optimal solution. Nevertheless, K-means clustering produces a single partitioning, but hierarchical clustering can give distinctive partitioning depending on the level-of-determination that we bring into consideration.

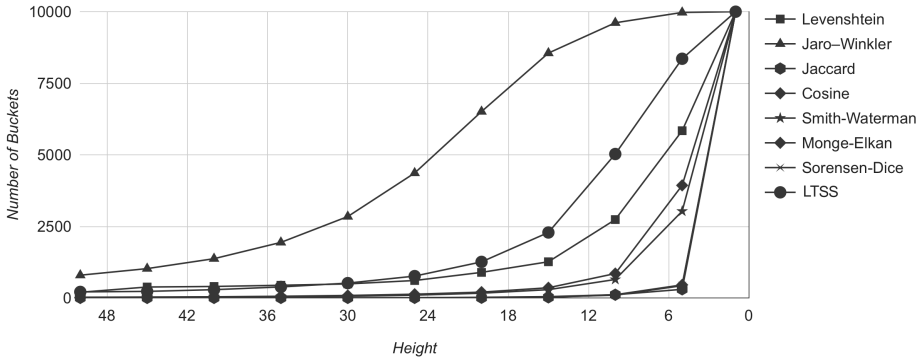


Fig. 5: Number of buckets per threshold

Figure 5 shows that the LTSS algorithm is not too sensitive to the changes in the level of threshold because it produces continuous good results through the shift in threshold during the hierarchical clustering process. The best algorithm is the Jaro-Winkler, which can be seen as a smooth curve with exponential gradual increase versus decreasing the height. The rest of the algorithms show similar behavior by generating neighboring curved lines.

When comparing string similarity measures, many inexpensive algorithms in terms of computational cost can surpass quality limitations and open up new application fields such as the task of disambiguation. In cascaded classification used to obtain disambiguation results on large amount of data, each classifier should produce a decision as fast as possible to speed up the overall process. Results of comparing algorithms in terms of computational cost in Figure 6 shows that the LTSS algorithm is the fastest algorithm which makes it a better choice for big-data and real-time applications.

The closest competitor of the LTSS algorithm is the Levenshtein algorithm, with the significant difference of practically twice the amount of total time to produce results. The slowest algorithm is Smith-Waterman with showing around thrice completion time compared to the LTSS algorithm.

6 Conclusion

We studied the string similarity search problem and introduced a new method: the Lookup Table String Similarity (LTSS) algorithm. We performed a quantitative analysis between the new algorithm and several extensively used string similarity algorithms to evaluate the performance of the LTSS algorithm. Additionally, our research introduced a comprehensive comparison study between some of the existing widely used similarity measures. On the one, depending on the type of the application, one can use our results in order to understand the trade-offs between different performance indicators when choosing a suitable similarity algorithm. On the other hand, our novel algorithm has a lower computational cost with an acceptable rate of accuracy. When operating on large datasets, our proposed algorithm is time-efficient, which can be of an interest

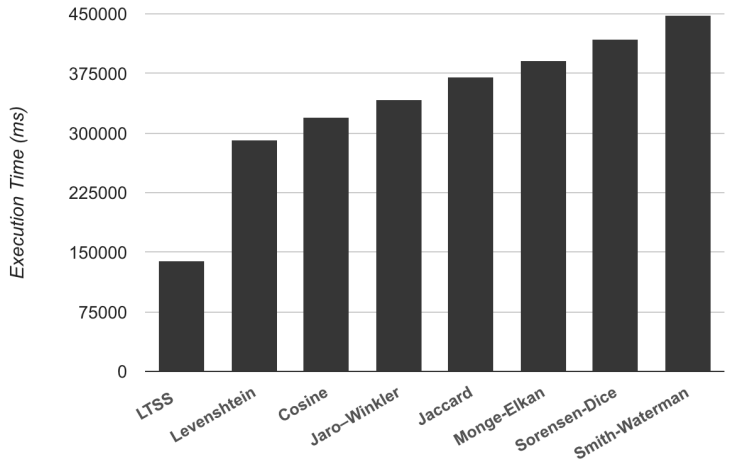


Fig. 6: Total execution time

in applications with indispensable need of high processing speed such as disambiguation. Future work will include testing our method against different databases, to better understand its drawbacks and improve its performance.

References

1. Agbehadji, I.E., Yang, H., Fong, S., Millham, R.: The comparative analysis of smith-waterman algorithm with jaro-winkler algorithm for the detection of duplicate health related records. In: 2018 International Conference on Advances in Big Data, Computing and Data Communication Systems (icABCD). pp. 1–10. IEEE (2018)
2. Baeza-Yates, R., Ribeiro-Neto, B., et al.: Modern information retrieval, vol. 463. ACM press New York (1999)
3. Bay, S.D., Schwabacher, M.: Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 29–38. ACM (2003)
4. Beernaerts, J., Debever, E., Lenoir, M., De Baets, B., Van de Weghe, N.: A method based on the levenshtein distance metric for the comparison of multiple movement patterns described by matrix sequences of different length. Expert Systems with Applications 115, 373–385 (2019)
5. Besta, M., Kanakagiri, R., Mustafa, H., Karasikov, M., Rätsch, G., Hoefler, T., Solomonik, E.: Communication-efficient jaccard similarity for high-performance distributed genome comparisons. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 1122–1132. IEEE (2020)
6. Bilenko, M., Mooney, R.J.: Adaptive duplicate detection using learnable string similarity measures. In: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 39–48. ACM (2003)
7. Branting, L.K.: A comparative evaluation of name-matching algorithms. In: Proceedings of the 9th international conference on Artificial intelligence and law. pp. 224–232. ACM (2003)

8. Caliano, D., Fersini, E., Manchanda, P., Palmonari, M., Messina, E.: Unimib: Entity linking in tweets using jaro-winkler distance, popularity and coherence. In: # Microposts. pp. 70–72 (2016)
9. Chapman, S.: Simmetrics. URL http://sourceforge. net/projects/simmetrics/. SimMetrics is a Similarity Metric Library, eg from edit distance's (Levenshtein, Gotoh, Jaro etc) to other metrics,(eg Soundex, Chapman). Work provided by UK Sheffield University funded by (AKT) an IRC sponsored by EPSRC, grant number GR N 15764 (2009)
10. Chung, J.Y., Park, B., Won, Y.J., Strassner, J., Hong, J.W.: An effective similarity metric for application traffic classification. In: 2010 IEEE Network Operations and Management Symposium-NOMS 2010. pp. 286–292. IEEE (2010)
11. Cios, K.J., Pedrycz, W., Swiniarski, R.W.: Data mining methods for knowledge discovery, vol. 458. Springer Science & Business Media (2012)
12. Cohen, W., Ravikumar, P., Fienberg, S.: A comparison of string metrics for matching names and records. In: Kdd workshop on data cleaning and object consolidation. vol. 3, pp. 73–78 (2003)
13. Cohen, W.W., Richman, J.: Learning to match and cluster large high-dimensional data sets for data integration. In: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 475–480. ACM (2002)
14. Feng, X., Huang, X., Tian, X., Ma, Y.: Automatic traffic signature extraction based on smith-waterman algorithm for traffic classification. In: 2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT). pp. 154–158. IEEE (2010)
15. Gali, N., Mariescu-Istodor, R., Fränti, P.: Similarity measures for title matching. In: 2016 23rd International Conference on Pattern Recognition (ICPR). pp. 1548–1553. IEEE (2016)
16. Gomaa, W.H., Fahmy, A.A.: A survey of text similarity approaches. International Journal of Computer Applications 68(13) (2013)
17. Grundkiewicz, R., Junczys-Dowmunt, M.: The wiked error corpus: A corpus of corrective wikipedia edits and its application to grammatical error correction. In: International Conference on Natural Language Processing. pp. 478–490. Springer (2014)
18. Hajihashemi, Z., Popescu, M.: An early illness recognition framework using a temporal smith waterman algorithm and nlp. In: AMIA Annual Symposium Proceedings. vol. 2013, p. 548. American Medical Informatics Association (2013)
19. Heynderickx, P.M., Van Huffel, K., Dewulf, J., Van Langenhove, H.: Application of similarity coefficients to sift-ms data for livestock emission characterization. Biosystems engineering 114(1), 44–54 (2013)
20. Irving, R.W.: Plagiarism and collusion detection using the smith-waterman algorithm. University of Glasgow 9 (2004)
21. Jaccard, P.: The distribution of the flora in the alpine zone. New phytologist 11(2), 37–50 (1912)
22. Jaro, M.A.: Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. Journal of the American Statistical Association 84(406), 414–420 (1989)
23. Larsen, B., Aone, C.: Fast and effective text mining using linear-time document clustering. In: Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 16–22. ACM (1999)
24. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics doklady. vol. 10, pp. 707–710 (1966)
25. Lilley, M.S., Garland, E.C., Rekdahl, M.L., Noad, M.J., Goldizen, A.W., Garrigue, C.: Improved versions of the levenshtein distance method for comparing sequence information in animals' vocalisations: tests using humpback whale song. Behaviour 149(13-14), 1413–1441 (2012)
26. Lopez, V., Fernández, M., Motta, E., Stieler, N.: Poweraqua: Supporting users in querying and exploring the semantic web. Semantic Web 3(3), 249–265 (2012)

27. MacQueen, J., et al.: Some methods for classification and analysis of multivariate observations. In: Proceedings of the fifth Berkeley symposium on mathematical statistics and probability. vol. 1, pp. 281–297. Oakland, CA, USA. (1967)
28. Martins, B.: A supervised machine learning approach for duplicate detection over gazetteer records. In: International Conference on GeoSpatial Sematics. pp. 34–51. Springer (2011)
29. Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al.: Mllib: Machine learning in apache spark. Journal of Machine Learning Research 17(34), 1–7 (2016)
30. Monge, A., Elkan, C.: An efficient domain-independent algorithm for detecting approximately duplicate database records (1997)
31. Monge, A.E., Elkan, C., et al.: The field matching problem: Algorithms and applications. In: KDD. pp. 267–270 (1996)
32. Murguía, M., Villaseñor, J.L.: Estimating the effect of the similarity coefficient and the cluster algorithm on biogeographic classifications. In: Annales Botanici Fennici. pp. 415–421. JSTOR (2003)
33. Ngomo, A.C.N., Auer, S.: Limes-a time-efficient approach for large-scale link discovery on the web of data. integration 15(3) (2011)
34. Nikolova, E., Jecheva, V.: Some similarity coefficients and application of data mining techniques to the anomaly-based ids. Telecommunication Systems 50(2), 127–135 (2012)
35. Papadimitriou, P., Dasdan, A., Garcia-Molina, H.: Web graph similarity for anomaly detection. Journal of Internet Services and Applications 1(1), 19–30 (2010)
36. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. Journal of molecular biology 147(1), 195–197 (1981)
37. Sørensen, T.: A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on danish commons. Biol. Skr. 5, 1–34 (1948)
38. Sørenson, T.: A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on Danish commons (1948)
39. Soukoreff, R.W., MacKenzie, I.S.: Measuring errors in text entry tasks: An application of the levenshtein string distance statistic. In: CHI'01 extended abstracts on Human factors in computing systems. pp. 319–320 (2001)
40. Su, Z., Ahn, B.R., Eom, K.Y., Kang, M.K., Kim, J.P., Kim, M.K.: Plagiarism detection using the levenshtein distance and smith-waterman algorithm. In: 2008 3rd International Conference on Innovative Computing Information and Control. pp. 569–569. IEEE (2008)
41. Sun, Y., Ma, L., Wang, S.: A comparative evaluation of string similarity metrics for ontology alignment. Journal of Information &Computational Science 12(3), 957–964 (2015)
42. Tao, W., et al.: Approximate string joins with abbreviations. Ph.D. thesis, Massachusetts Institute of Technology (2018)
43. Winkler, W.E.: String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. (1990)
44. Xue, Z., Yin, D., Davison, B.D., Davison, B.: Normalizing microtext. Analyzing Microtext 11, 05 (2011)
45. Yamaguchi, A., Yamamoto, Y., Kim, J.D., Takagi, T., Yonezawa, A.: Discriminative application of string similarity methods to chemical and non-chemical names for biomedical abbreviation clustering. In: BMC genomics. vol. 13, p. S8. Springer (2012)
46. Zhang, X.L., Fu, Y.Z., Chu, P.X.: Application of jaccard similarity coefficient in recommender system. Computer Technology and Development 24(4), 158–165 (2015)
47. Zhao, M., Lee, W.P., Garrison, E.P., Marth, G.T.: Ssw library: an simd smith-waterman c/c++ library for use in genomic applications. PloS one 8(12), e82138 (2013)

48. Zhu, M., Shen, D., Nie, T., Kou, Y.: An adjusted-edit distance algorithm applying to web environment. In: 2009 Sixth Web Information Systems and Applications Conference. pp. 71–75. IEEE (2009)