Csorba-Kurzer (CK) Similarity

US Patent 9,269,028

Brett Csorba Jacob Kurzer

July 10, 2016

1 Overview

While numeric data is easy to compare and correlate, the same cannot be said for character strings. What determines how similar *cat* is to *bat* and how similar both are to *chat*? The traditional approach used to determine this similarity can be defined by the edit distance or the number of character insertions, deletions, and transpositions to transform one character string to another. This edit distance is also known as Damerau–Levenshtein distance. While this metric produces accurate and useful measures of similarity, algorithms traditionally used to compute this distance have a $\mathrm{O}(mn)$ running time, where m and n are the length of the input strings. Faster running algorithms do exist that run in linear time, but these algorithms are limited by aspects such as machine word size or limit the range of the input strings they can operate on.

Our algorithm calculates string similarity resulting in orders that strongly approximate traditional edit distance calculations while running in linear $\mathrm{O}(m+n)$ time. This is achieved through linear preprocessing steps of input strings that can be memoized for future string comparisons and linear character comparison steps. Our approach is general purpose and has been shown to be effective in computing similarity of a variety of string types including machine produced hashed strings. Applications range from finding patterns in phishing emails, evaluating network topology, and determining attack patterns on large scale data encompassing tens of thousands of string of variable lengths and hierarchies.

Our algorithm will return a floating point value in the range of [0 1], 0 indicating perfect similarity and 1 indicating maximum dissimilarity. A notable point is that we believe that this algorithm returns values which will satisfy the triangle inequality, allowing for pruning approaches when dealing with large data sets. A Java implementation of this code is currently being used for server side computations. The original implementation of the algorithm was in JavaScript, allowing this code to be highly portable and run client side on web applications, having the potential to reduce server computation time and costs.

2 Preprocess Strings

Our algorithm will first preprocess each of the input strings into a data structure which will allow the algorithm to find indexes of characters within each string. These structures can be memorized in a hashtable structure preventing this step from being repeated for future comparisons against the input strings.

Each input string s contains the character set C. Create a hashtable h for this string. For each character s_i in s, append the index i of s_i to the end of the List l at key s_i. As i is always increasing, l will be in sorted order. Assuming $\mathrm{O}(1)$ insertions, accesses, and edits for h and $\mathrm{O}(1)$ inserts into l, a string of length m can be preprocessed in $\mathrm{O}(m)$ time.

3 Comparing Strings

Preprocess strings A and B with lengths m and n into hashes a and b taking $\mathrm{O}(m+n)$ time. These strings are compared through both a measurement of dissimilarity in the character sets C_a and C_b and the different indexes of intersecting characters in these sets. Three measures are kept track of, character similarity in string A (S_A), character similarity in string B, (S_B), and total disordering (D).

First, each key k_a in a is evaluated.

Key	Vals		
a	0	1	3
b	2	4	
c	5		

Figure 1: Hash for string $A =$ aababc

Key	Vals		
a	2		
b	0	3	
d	1	4	5

Figure 2: Hash for string $B =$ bdabdd

If $k_a \notin b$, then add 1 to S_A.

If $k_a \in b$, we need to compute how disordered the indexes of l_{k_a} are compared to l_{k_b}. Compare the lengths of l_{k_a} and l_{k_b}. Assign the shorter list to c and the longer list to d. Remember c and d are ordered.

For each index c_i in c, find the first element in d that is greater than c_i by iterating through d. For each element in d that is less than or equal to c_i, push this element onto a stack s.

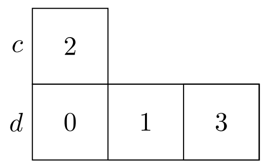

Figure 3: For k_a = a

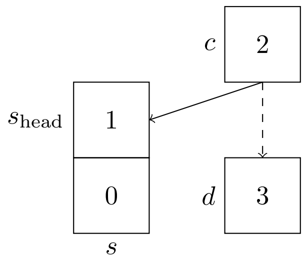

Figure 4: State after alignment, k_a = a

If the total number of remaining indexes in c and d are equal and the stack is empty, this means that the smallest possible disordering would be to pair $c_1 \to d_1, c_2 \to d_2, ..., c_n \to d_n$. Add to D the absolute value of $c_1 - d_1, c_2 - d_2, ..., c_n d_n$. There are no more letters then in c to process.

If not, compare $d_j - c_i$ to $c_i - s_{\text{head}}$. The smaller value is the index that is the closest to c_i. Add this value to D. If this is $d_j - c_i$, remove d_j from d so it is no longer evaluated. If this is $c_i - s_{\text{head}}$, pop s_{head} from s. Continue to process the remaining c_i in c in this way.

After all remaining c_i have been processed, d and s may contain still contain indexes. If d was assigned from a, add 1 to S_A for each letter remaining in d and in s. If d was assigned from b, add 1 to S_B for each letter remaining in d and in s.

Add the count of $k_b \notin a$ to S_B.

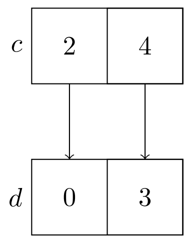

Figure 5: State after alignment, k_b = b

In this approach, each character of A is touched once for a total of m times and each character of B is touched at most $2n$. This means this step will run in $\mathrm{O}(m+2n)$ or $\mathrm{O}(m+n)$.

At the conclusion of this step we have the two measures of similarity, S_A and S_B. The greater value is equal to the least possible number of character insertions/deletions/replacements to transform A into B. Set S to this greater value. We also have the measure of reordering necessary to transform A into B as D.

4 Calculating CK Similarity

Given the above measure of S and D, this gives end users two important measures in which unique similarity algorithms can be developed. Our developed metric takes equally into account S and D.

$$1 - (\%\mathrm{Ordered})(\%\mathrm{SimilarChars})$$

Where n equals the length of the longer string:

$$1 - (\frac{n^2 - D}{n^2})(\frac{n - S}{n})$$

$$\frac{S}{n} + \frac{D}{n^2} + \frac{DS}{n^3}$$

5 Pseudocode

```
Data: String $s$ of length $m$
Result: CKHash $h$ of $s$: key: character, value: queue
for $i \leftarrow 0$ to $m-1$ do
    $c \leftarrow s.charAt(i)$;
    $l \leftarrow h.get(c)$;
    $l.enqueue(i)$;
end
```

Algorithm 1: Generate a CKHash

```
**Data:** String $X$ of length $m$, String $Y$ of length $n$, where $m \leq n$
**Result:** CKSimilarity of $X$ and $Y$
$x \leftarrow CKHash(X)$;
$y \leftarrow CKHash(Y)$;
$S_a \leftarrow 0$;
$S_b \leftarrow 0$;
$D \leftarrow 0$;
for $c \in x.keys \cap y.keys$ do
    $qx \leftarrow x.get(c)$;
    $qy \leftarrow y.get(c)$;
    if $qx$ is longer than $qy$ then
        swap $qx$ and $qy$;
    end
    while $qx$ is not empty do
        $xhead \leftarrow qx.dequeue$;
        while $qy.head < xhead$ do
            Stack $s.push(qy.dequeue)$;
        end
        if $xhead - s.peek < qy.head - head$ then
            $D$ += $s.pop$;
        end
        else
            $D$ += $qy.dequeue$;
        end
    end
    if $qx \in x$ then
        $S_a$ += $s.size$;
        $S_a$ += $qy.size$;
    end
    else
        $S_b$ += $s.size$;
        $S_b$ += $qy.size$;
    end
end
for $c \in x.keys \setminus y.keys$ do
    $S_a$ += $x.get(c).size$;
end
for $c \in y.keys \setminus x.keys$ do
    $S_b$ += $y.get(c).size$;
end
$S \leftarrow max(S_a, S_b)$;
Calculate CKSimilarity;
```

Algorithm 2: CKSimilarity algorithm