Similar String algorithm

Asked 15 years, 11 months ago Modified 8 years, 7 months ago

Viewed 17k times

21

I'm looking for an algorithm, or at least theory of operation on how you would find similar text in two or more different strings...

Much like the question posed here: Algorithm to find articles with similar text, the difference being that my text strings will only ever be a handful of words.

Like say I have a string: "Into the clear blue sky" and I'm doing a compare with the following two strings: "The color is sky blue" and "In the blue clear sky"

I'm looking for an algorithm that can be used to match the text in the two, and decide on how close they match. In my case, spelling, and punctuation are going to be important. I don't want them to affect the ability to discover the real text. In the above example, if the color reference is stored as "'sky-blue'", I want it to still be able to match. However, the 3rd string listed should be a BETTER match over the second, etc.

I'm sure places like Google probably use something similar with the "Did you mean:" feature...

*** EDIT ***

In talking with a friend, he worked with a guy who wrote a

paper on this topic. I thought I might share it with everyone reading this, as there are some really good methods and processes described in it...

Here's the link to his paper, I hope it is helpful to those reading this question, and on the topic of similar string algorithms.

c++ c algorithm string

Share

Improve this question

Follow

edited May 23, 2017 at 12:17

Community Bot

1 ● 1

asked Jan 16, 2009 at 20:34

LarryF

5,073 ● 4 ● 34 ● 40

9 Answers

Sorted by: Highest score (default)

16

Levenshtein distance will not completely work, because you want to allow rearrangements. I think your best bet is going to be to find best rearrangement with levenstein distance as cost for each word.

To find the cost of rearrangement, kinda like the pancake sorting problem. So, you can permute every combination of words (filtering out exact matches), with every combination of other string, trying to minimize a

combination of permute distance and Levenshtein distance on each word pair.

edit: Now that I have a second I can post a quick example (all 'best' guesses are on inspection and not actually running the algorithms):

```
original strings              | best rearrangement w/ l
Into the clear blue sky       |    Into the c_lear blue
The color is sky blue         |    is__ the colo_r blue

R_dist = dist( 3 1 2 5 4 ) --> 3 1 2 *4 5* --> *2 1 3*
L_dist = (2D+S) + (I+D+S) (Total Subsitutions: 2, dele
```

(notice all the flips include all elements in the range, and I use ranges where Xi - Xj = +/- 1)

Other example

```
original strings              | best rearrangement w/ l
Into the clear blue sky       |   Into the clear blue s
In the blue clear sky         |   In__ the clear blue s

R_dist = dist( 1 2 4 3 5 ) -->  1 2 *3 4* 5  = 1
L_dist = (2D) (Total Subsitutions: 0, deletions: 2, in
```

And to show all possible combinations of the three...

```
The color is sky blue          |    The colo_r is sky b
In the blue clear sky          |    the c_lear in sky b

R_dist = dist( 2 4 1 3 5 ) --> *2 3 1 4* 5 --> *1 3 2*
L_dist = (D+I+S) + (S) (Total Subsitutions: 2, deletio
```

Anyway you make the cost function the second choice will be lowest cost, which is what you expected!

Share Improve this answer Follow

edited Jan 19, 2009 at 23:36

community wiki
7 revs
nlucaroni

Ha -- I answered Levenstein distance on that one, too :P I'm not sure I'm smart enough to grok the paper you referenced in that one though 0.o – Dana Jan 16, 2009 at 20:53

I'm not yet sure what Levenstein distance does, but to make results "without respect to order" it seems like you could normalize the order before running the algorithm. perhaps alphabetizing the words. "blue color is sky the" | "blue clear in sky the". There are probably cases where it would not help, just a thought. – SketchBookGames Jul 20, 2016 at 17:03

lev distance counts the numbers of insertions and deletions to transform one piece of text into the other. By ordering the sentence, you aren't adding a penalty for that ordering or adding the complexity of that ordering. So, that could work if you don't care about that. – nlucaroni Nov 1, 2016 at 16:15

15

One way to determine a measure of "overall similarity without respect to order" is to use some kind of **compression-based distance**. Basically, the way most compression algorithms (e.g. `gzip`) work is to scan along a string looking for string segments that have appeared earlier -- any time such a segment is found, it is replaced

with an (offset, length) pair identifying the earlier segment to use. You can use measures of how well two strings compress to detect similarities between them.

Suppose you have a function `string comp(string s)` that returns a compressed version of `s`. You can then use the following expression as a "similarity score" between two strings `s` and `t`:

```
len(comp(s)) + len(comp(t)) - len(comp(s . t))
```

where `.` is taken to be concatenation. The idea is that you are measuring how much *further* you can compress `t` by looking at `s` first. If `s == t`, then `len(comp(s . t))` will be barely any larger than `len(comp(s))` and you'll get a high score, while if they are completely different, `len(comp(s . t))` will be very near `len(comp(s) + comp(t))` and you'll get a score near zero. Intermediate levels of similarity produce intermediate scores.

Actually the following formula is even better as it is symmetric (i.e. the score doesn't change depending on which string is `s` and which is `t`):

```
2 * (len(comp(s)) + len(comp(t))) - len(comp(s . t)) -
```

This technique has its roots in information theory.

Advantages: good compression algorithms are already available, so you don't need to do much coding, and they

run in linear time (or nearly so) so they're fast. By contrast, solutions involving all permutations of words grow super-exponentially in the number of words (although admittedly that may not be a problem in your case as you say you know there will only be a handful of words).

Share Improve this answer

Follow

answered Jan 17, 2009 at 7:52

j_random_hacker

51.2k ● 10 ● 108 ● 173

1 I like this method too! Definitly going to put this sea in my tool kit!! – nlucaroni Jan 17, 2009 at 19:13

1 VERY interesting idea... I will need to decide on a compression algorithm to use. Do I go for something tried and true, like deflate, or LZ77 vs. UDA? I guess I would want to use whichever is best at doing the raw compression, throwing out all the dictionary data, etc. Or is that part of len()? – LarryF Jan 19, 2009 at 22:33

Deflate prepends a Huffman table, so for short inputs it will give you scores that are distorted in an "absolute" sense, but still it retains the property that score(X, Y) < score(X, Z) implies X is more similar to Y than to Z. Some quick experiments with echo -n "..."|gzip -c|wc -c confirm this. – j_random_hacker Jan 20, 2009 at 12:40

Most practical compression algos output some initial header info, so distorted scores for short inputs will always be a problem, but if the important thing is being able to detect the best match of a string X to several possible strings Y, that's not important -- only relative scores matter. – j_random_hacker Jan 20, 2009 at 12:58

1 You may also want to consider normalising the score by dividing by e.g. len(comp(s))+len(comp(t)).

– j_random_hacker Jan 20, 2009 at 13:02

5

One way (although this is perhaps better suited a spellcheck-type algorithm) is the "edit distance", ie., calculate how many edits it takes to transform one string to another. A common technique is found here:

http://en.wikipedia.org/wiki/Levenshtein_distance

Share Improve this answer

Follow

answered Jan 16, 2009 at 20:40

Dana

32.9k ● 17 ● 64 ● 73

Thanks. I'm gonna read up on this. It was mentioned in the other question I referenced, but I wasn't sure that's what I'm looking for. I thought I was looking more for an algorithm that looked at the words, and did a match-seach-match type of approach. – LarryF Jan 16, 2009 at 20:48

5

You might want to look into the algorithms used by biologists to compare DNA sequences, since they have to cope with many of the same things (chunks may be missing, or have been inserted, or just moved to a different position in the string.

The Smith-Waterman algorithm would be one example that'd probably work fairly well, although it might be too slow for your uses. Might give you a starting point, though.

Share Improve this answer

Follow

answered Jan 16, 2009 at 21:22

Stack Overflow is garbage

247k ● 52 ● 353 ● 555

2

i had a similar problem, i needed to get the percentage of characters in a string that were similar. it needed exact sequences, so for example "hello sir" and "sir hello" when compared needed to give me five characters that are the same, in this case they would be the two "hello"'s. it would then take the length of the longest of the two strings and give me a percentage of how similar they were. this is the code that i came up with

```
int compare(string a, string b){
   return(a.size() > b.size() ? bigger(a,b) : bigger(b
}



int bigger(string a, string b){



int maxcount = 0, currentcount = 0;//used to see which
characters were biggest

for(int i = 0; i < a.size(); ++i){

    for(int j = 0; j < b.size(); ++j){

        if(a[i+j] == b[j]){

         ++currentcount;

         }
```

```
        else{

            if(currentcount > maxcount){

             maxcount = currentcount;

             }//end if

             currentcount = 0;

            }//end else

        }//end inner for loop

     }//end outer for loop


    return ((int)(((float)maxcount/((float)a.size()))*1
}
```

Share Improve this answer

Follow

edited Nov 21, 2011 at 6:44

Jesse Beder

34k ● 22 ● 110 ● 146

answered Nov 21, 2011 at 6:40

mckinnley

21 ● 1

2

I can't mark two answers here, so I'm going to answer and mark my own. The Levenshtein distance appears to be the correct method in most cases for this. But, it is worth mentioning `j_random_hackers` answer as well. I have used an implementation of LZMA to test his theory, and it proves to be a sound solution. In my original question I was looking for a method for short strings (2 to 200 chars), where the Levenshtein Distance algorithm will

work. But, not mentioned in the question was the need to compare two (larger) strings (in this case, text files of moderate size) and to perform a quick check to see how similar the two are. I believe that this compression technique will work well but I have yet to study it to find at which point one becomes better than the other, in terms of the size of the sample data and the speed/cost of the operation in question. I think a lot of the answers given to this question are valuable, and worth mentioning, for anyone looking to solve a similar string ordeal like I'm doing here. Thank you all for your great answers, and I hope they can be used to serve others well too.

Share Improve this answer

Follow

edited May 23, 2017 at 12:10

Community Bot

1 ● 1

answered Feb 5, 2009 at 2:34

LarryF

5,073 ● 4 ● 34 ● 40

1

There's another way. Pattern recognition using convolution. Image A is run thru a Fourier transform. Image B also. Now superimposing F(A) over F(B) then transforming this back gives you a black image with a few white spots. Those spots indicate where A matches B strongly. Total sum of spots would indicate an overall similarity. Not sure how you'd run an FFT on strings but I'm pretty sure it would work.

Share Improve this answer

Follow

answered May 26, 2016 at 20:27

Wood

65 ● 4

0

The difficulty would be to match the strings semantically.

You could generate some kind of value based on the lexical properties of the string. e.g. They bot have blue, and sky, and they're in the same sentence, etc etc... But it won't handle cases where "Sky's jean is blue", or some other odd ball English construction that uses same words, but you'd need to parse the English grammar...

To do anything beyond lexical similarity, you'd need to look at natural language processing, and there isn't going to be one single algorith that would solve your problem.

Share Improve this answer

Follow

answered Jan 16, 2009 at 21:37

Calyth

1,673 ● 4 ● 16 ● 26

-2

Possible approach:

Construct a Dictionary with a string key of "word1|word2" for all combinations of words in the *reference* string. A single combination may happen multiple times, so the value of the Dictionary should be a *list* of numbers, each representing the *distance* between the words in the reference string.

When you do this, there will be duplication here: for every "word1|word2" dictionary entry, there will be a "word2|word1" entry with the same list of distance values, but negated.

For each combination of words in the *comparison* string (words 1 and 2, words 1 and 3, words 2 and 3, etc.), check the two keys (word1|word2 and word2|word1) in the reference string and find the *closest* value to the distance in the current string. Add the absolute value of the difference between the current distance and the closest distance to a counter.

If the closest reference distance between the words is in the opposite direction (word2|word1) as the comparison string, you may want to weight it smaller than if the closest value was in the same direction in both strings.

When you are finished, divide the sum by the square of the number of words in the comparison string.

This should provide some decimal value representing how closely each word/phrase matches some word/phrase in the original string.

Of course, if the original string is longer, it won't account for that, so it may be necessary to compute this both directions (using one as the reference, then the other) and average them.

I have absolutely no code for this, and I probably just re-invented a very crude wheel. YMMV.

Share Improve this answer

Follow

edited Jan 16, 2009 at 21:25

answered Jan 16, 2009 at 21:11

richardtallent

35.3k ● 14 ● 86 ● 125