Getting the closest string match

Asked 13 years, 7 months ago Modified 1 month ago

Viewed 199k times

441

I need a way to compare multiple strings to a test string and return the string that closely resembles it:

```
TEST STRING: THE BROWN FOX JUMPED OVER THE RED COW

CHOICE A   : THE RED COW JUMPED OVER THE GREEN
CHICKEN
CHOICE B   : THE RED COW JUMPED OVER THE RED COW
CHOICE C   : THE RED FOX JUMPED OVER THE BROWN COW
```

(If I did this correctly) The closest string to the "TEST STRING" should be "CHOICE C". What is the easiest way to do this?

I plan on implementing this into multiple languages including VB.net, Lua, and JavaScript. At this point, pseudo code is acceptable. If you can provide an example for a specific language, this is appreciated too!

algorithm language-agnostic string-comparison levenshtein-distance

Share

Improve this question

edited Nov 14, 2014 at 1:23

Yu Hao

122k ● 48 ● 246 ● 302

Follow

asked May 2, 2011 at 16:20

Freesnöw

32k ● 31 ● 93 ● 139

3 Algorithms that typically do this type of stuff work on determining how many changes it takes to turn an examined string into the target string. Those types of algorithms don't work well at all in a situation like this. I think getting a computer to pull this off will be very tough. – Matt Greer May 2, 2011 at 16:28

4 Levenshtein distance source code in many languages: Java, Ruby, Python, PHP, etc. en.wikibooks.org/wiki/Algorithm_Implementation/Strings/… – joelparkerhenderson May 4, 2012 at 1:26

11 In general, what counts as "closest string" will depend on the similarity measure used, and the penalties used for introducing gaps in the alignment. For example, do you consider "cow" and "chicken" more similar than "cow" and "red" (because they are related concepts), or is it the other way around (because "chicken" has more letters than "cow")? But given a similarity measure and gap penalty, it can be shown that the Levenshtein algorithm below is guaranteed to find you the closest string. Same is true of Needleman-Wunsch and Smith-Waterman (further below). – Sten L May 4, 2012 at 9:56

Do character grouping, or word grouping. Give it score. – Casey ScriptFu Pharr Feb 8, 2019 at 19:12

14 Answers

Sorted by: Highest score (default)

1024

+100

I was presented with this problem about a year ago when it came to looking up user entered information about a oil rig in a database of miscellaneous information. The goal was to do some sort of fuzzy string search that could identify the database entry with the most common elements.

Part of the research involved implementing the Levenshtein distance algorithm, which determines how many changes must be made to a string or phrase to turn it into another string or phrase.

The implementation I came up with was relatively simple, and involved a weighted comparison of the length of the two phrases, the number of changes between each phrase, and whether each word could be found in the target entry.

The article is on a private site so I'll do my best to append the relevant contents here:

Fuzzy String Matching is the process of performing a human-like estimation of the similarity of two words or phrases. In many cases, it involves identifying words or phrases which are most similar to each other. This article describes an in-house solution to the fuzzy string matching problem and its usefulness in solving a variety of problems which can allow us to automate tasks which previously required tedious user involvement.

Introduction

The need to do fuzzy string matching originally came about while developing the Gulf of Mexico Validator tool. What existed was a database of known gulf of Mexico oil rigs and platforms, and people buying insurance would give us some badly typed out information about their assets and we had to match it to the database of known platforms. When there was very little information given, the best we could do is rely on an underwriter to "recognize" the one they were referring to and call up the proper information. This is where this automated solution comes in handy.

I spent a day researching methods of fuzzy string matching, and eventually stumbled upon the very useful Levenshtein distance algorithm on Wikipedia.

Implementation

After reading about the theory behind it, I implemented and found ways to optimize it. This is how my code looks like in VBA:

```
'Calculate the Levenshtein Distance between two string
insertions,
'deletions, and substitutions needed to transform the
second)
Public Function LevenshteinDistance(ByRef S1 As String
Long
    Dim L1 As Long, L2 As Long, D() As Long 'Length of
distance matrix
    Dim i As Long, j As Long, cost As Long 'loop count
substitution for current letter
    Dim cI As Long, cD As Long, cS As Long 'cost of ne
```

```
and Substitution
    L1 = Len(S1): L2 = Len(S2)
    ReDim D(0 To L1, 0 To L2)
    For i = 0 To L1: D(i, 0) = i: Next i
    For j = 0 To L2: D(0, j) = j: Next j

    For j = 1 To L2
        For i = 1 To L1
            cost = Abs(StrComp(Mid$(S1, i, 1), Mid$(S2
            cI = D(i - 1, j) + 1
            cD = D(i, j - 1) + 1
            cS = D(i - 1, j - 1) + cost
            If cI <= cD Then 'Insertion or Substitutio
                If cI <= cS Then D(i, j) = cI Else D(i
            Else 'Deletion or Substitution
                If cD <= cS Then D(i, j) = cD Else D(i
            End If
        Next i
    Next j
    LevenshteinDistance = D(L1, L2)
End Function
```

Simple, speedy, and a very useful metric. Using this, I created two separate metrics for evaluating the similarity of two strings. One I call "valuePhrase" and one I call "valueWords". valuePhrase is just the Levenshtein distance between the two phrases, and valueWords splits the string into individual words, based on delimiters such as spaces, dashes, and anything else you'd like, and compares each word to each other word, summing up the shortest Levenshtein distance connecting any two words. Essentially, it measures whether the information in one 'phrase' is really contained in another, just as a word-wise permutation. I spent a few days as a side project coming up with the most efficient way possible of splitting a string based on delimiters.

valueWords, valuePhrase, and Split function:

```
Public Function valuePhrase#(ByRef S1$, ByRef S2$)
    valuePhrase = LevenshteinDistance(S1, S2)
End Function

Public Function valueWords#(ByRef S1$, ByRef S2$)
    Dim wordsS1$(), wordsS2$()
    wordsS1 = SplitMultiDelims(S1, " _-")
    wordsS2 = SplitMultiDelims(S2, " _-")
    Dim word1%, word2%, thisD#, wordbest#
    Dim wordsTotal#
    For word1 = LBound(wordsS1) To UBound(wordsS1)
        wordbest = Len(S2)
        For word2 = LBound(wordsS2) To UBound(wordsS2)
            thisD = LevenshteinDistance(wordsS1(word1)
            If thisD < wordbest Then wordbest = thisD
            If thisD = 0 Then GoTo foundbest
        Next word2
foundbest:
        wordsTotal = wordsTotal + wordbest
    Next word1
    valueWords = wordsTotal
End Function

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' SplitMultiDelims
' This function splits Text into an array of substring
' delimited by any character in DelimChars. Only a sin
' may be a delimiter between two substrings, but Delim
' contain any number of delimiter characters. It retur
' array containing all of text if DelimChars is empty,
' element array if the Text is successfully split into
' If IgnoreConsecutiveDelimiters is true, empty array
' If Limit greater than 0, the function will only spli
' array elements or less. The last element will contai
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Function SplitMultiDelims(ByRef Text As String, ByRef
        Optional ByVal IgnoreConsecutiveDelimiters As
        Optional ByVal Limit As Long = -1) As String()
    Dim ElemStart As Long, N As Long, M As Long, Eleme
    Dim lDelims As Long, lText As Long
    Dim Arr() As String
```

```
    lText = Len(Text)
    lDelims = Len(DelimChars)
    If lDelims = 0 Or lText = 0 Or Limit = 1 Then
        ReDim Arr(0 To 0)
        Arr(0) = Text
        SplitMultiDelims = Arr
        Exit Function
    End If
    ReDim Arr(0 To IIf(Limit = -1, lText - 1, Limit))

    Elements = 0: ElemStart = 1
    For N = 1 To lText
        If InStr(DelimChars, Mid(Text, N, 1)) Then
            Arr(Elements) = Mid(Text, ElemStart, N - E
            If IgnoreConsecutiveDelimiters Then
                If Len(Arr(Elements)) > 0 Then Element
            Else
                Elements = Elements + 1
            End If
            ElemStart = N + 1
            If Elements + 1 = Limit Then Exit For
        End If
    Next N
    'Get the last token terminated by the end of the s
    If ElemStart <= lText Then Arr(Elements) = Mid(Tex
    'Since the end of string counts as the terminating
character
    'was also a delimiter, we treat the two as consecu
last elemnent
    If IgnoreConsecutiveDelimiters Then If Len(Arr(Ele
= Elements - 1

    ReDim Preserve Arr(0 To Elements) 'Chop off unused
    SplitMultiDelims = Arr
End Function
```

Measures of Similarity

Using these two metrics, and a third which simply computes the distance between two strings, I have a

series of variables which I can run an optimization algorithm to achieve the greatest number of matches. Fuzzy string matching is, itself, a fuzzy science, and so by creating linearly independent metrics for measuring string similarity, and having a known set of strings we wish to match to each other, we can find the parameters that, for our specific styles of strings, give the best fuzzy match results.

Initially, the goal of the metric was to have a low search value for for an exact match, and increasing search values for increasingly permuted measures. In an impractical case, this was fairly easy to define using a set of well defined permutations, and engineering the final formula such that they had increasing search values results as desired.

Search for	Inside		Value Phrase	Value Words	SearchVal
abcd efgh	abcd efgh	(exact match)	0	0	0.00
abcd efgh	AbcD EFgH	(exact match, case insensitive)	0	0	0.00
abcd efgh	abcd efgh	(match, but with extra space)	0.6	0	0.12
abcd efgh	abcd zz efgh	(match, but with extra words)	0.6	0	0.12
abcd efgh	abcd,efgh	(match, but as one word)	1	0	0.20
abcd efgh	qqq abcd rrr sss efgh ttt	(words are found within phrase)	3.2	0	0.64
abcd efgh	abcds efghs	(words match, but with suffixes)	0.4	2	0.72
abcd efgh	efgh abcd	(words match, but phrase is reversed)	8	0	1.60
abcd efgh	abcdefgh	(words match, but concatinated)	0.2	8	1.76
abcd efgh	acbd efgh	(words almost match, but are permuted)	2	2	2.00
abcd efgh	zzz abcd zzzz	(one word match in the phrase)	4.8	4	4.16
abcd efgh	zzz efgh zzzz	(other word match in the phrase)	5.8	4	4.36
abcd efgh	aqbrcsdte	(some letters found, but not as a word)	7	14	8.40
abcd efgh	grasdcbue	(some permuted letters match)	9	16	10.40
abcd efgh	zzzzzzzzz	(no match)	9	18	10.80

In the above screenshot, I tweaked my heuristic to come up with something that I felt scaled nicely to my perceived difference between the search term and result. The heuristic I used for `Value Phrase` in the above spreadsheet was `=valuePhrase(A2,B2)-0.8*ABS(LEN(B2)-LEN(A2))`. I was effectively reducing the penalty of the Levenstein distance by 80% of the difference in the

length of the two "phrases". This way, "phrases" that have the same length suffer the full penalty, but "phrases" which contain 'additional information' (longer) but aside from that still mostly share the same characters suffer a reduced penalty. I used the `Value Words` function as is, and then my final `SearchVal` heuristic was defined as `=MIN(D2,E2)*0.8+MAX(D2,E2)*0.2` - a weighted average. Whichever of the two scores was lower got weighted 80%, and 20% of the higher score. This was just a heuristic that suited my use case to get a good match rate. These weights are something that one could then tweak to get the best match rate with their test data.

Value Phrase	Worldwide ALL Perils	Worldwide WIND	Worldwide QUAKE	North America ALL Pe	North America WIND	North America QUAKE	International ALL Peril	International WIND	International QUAKE	California QUAKE	Hawaii WIND	Florida WIND	Gulf WIND	Texas WIND	North East WIND	Mid Atlantic WIND	SouthEast WIND	Europe1 (UK & EIRE)	Europe3 (Scandinavia	Europe2 (Continental	Japan WIND	Europe4 QUAKE	Japan QUAKE	Texas and Gulf WIND	North America x FLA	Europe Wind	New Madrid QUAKE	Pacific Northwest QU	MidWest TORNADO
WORLDWIDE_ALL_PERIL	3	10	9	13	16	16	14	17	17	17	17	15	17	17	15	17	16	21	22	22	18	16	17	18	30	17	16	22	17
WORLDWIDE_WIND	10	1	6	20	11	16	21	12	17	15	9	7	9	10	9	12	9	17	19	19	10	13	14	14	30	8	12	20	14
WORLDWIDE_EARTHQUAKE	11	11	6	20	17	12	21	19	14	13	18	16	18	18	15	18	16	21	23	23	19	13	14	20	31	18	13	16	15
NA_ALL_PERIL	12	12	14	14	14	15	14	15	16	15	10	12	11	11	13	14	13	21	23	24	10	13	10	16	31	11	14	21	15
NA_WIND	18	10	13	21	12	16	21	12	16	14	6	7	5	5	9	12	9	20	20	20	5	13	10	14	32	7	13	20	13
NA_EARTHQUAKE	18	14	10	19	15	11	20	15	12	9	11	13	13	12	12	14	12	22	23	23	11	8	6	16	32	12	8	14	13
INTL_ALL_PERIL	11	13	14	15	15	17	12	13	14	16	13	13	13	12	14	13	13	21	23	23	13	14	13	17	32	13	15	20	15
INTL_WIND	17	9	12	21	13	17	19	10	14	14	7	8	5	6	10	10	9	19	20	18	6	13	11	13	32	7	15	20	12
INTL_EARTHQUAKE	17	14	9	20	16	12	18	13	10	10	14	14	14	13	12	14	12	22	23	23	14	9	9	17	33	14	9	14	13
CALIFORNIA_EARTHQUAKE	20	19	14	22	18	14	21	18	14	6	19	16	18	20	16	18	17	22	24	24	19	14	14	20	32	19	15	13	18
HAWAII_WIND	17	9	14	20	11	16	21	12	17	14	1	7	7	6	9	10	9	19	19	21	5	13	9	13	31	7	13	20	13
FLORIDA_WIND	15	7	12	20	11	16	20	11	16	12	7	1	8	8	9	12	9	18	19	19	8	12	12	14	31	7	13	20	14
GULF_WIND	18	9	13	22	14	18	22	14	17	13	7	8	1	6	11	12	9	19	21	20	6	12	11	11	33	6	15	21	14
TEXAS_WIND	17	10	14	20	12	16	20	11	16	15	6	8	6	1	8	12	7	19	20	19	5	13	10	10	31	7	14	21	13
NORTH_EAST_WIND	15	9	13	16	8	13	19	11	15	16	9	9	11	8	2	11	4	18	20	18	10	13	14	15	27	9	15	16	14
MID_ATLANTIC_WIND	17	12	17	21	12	17	19	12	16	16	10	12	12	12	11	2	11	19	19	19	10	17	14	13	31	13	16	21	13
SOUTHEAST_WIND	16	9	14	18	10	15	21	13	17	16	9	9	9	7	4	11	1	17	20	18	9	13	13	14	29	9	16	18	13
EUROPE1_WIND	17	9	13	21	12	17	21	12	17	15	8	8	7	8	9	13	9	13	16	16	7	7	12	14	32	2	15	21	14
EUROPE3_WIND	17	9	13	21	12	17	21	12	17	15	8	8	7	8	9	13	9	14	15	16	7	7	12	14	32	2	15	21	14
EUROPE2_WIND	17	9	13	21	12	17	21	12	17	15	8	8	7	8	9	13	9	14	16	15	7	7	12	14	32	2	15	21	14
JAPAN_WIND	18	10	14	21	13	17	21	12	16	14	5	8	6	5	10	10	9	19	19	20	1	12	6	12	32	7	15	20	14
EUROPE4_EARTHQUAKE	17	16	11	20	16	12	21	18	13	12	17	16	17	16	14	17	14	16	18	18	16	6	11	18	33	12	12	15	15
JAPAN_EARTHQUAKE	18	16	11	21	17	13	21	17	13	10	14	15	16	15	13	15	13	21	22	23	11	10	6	16	33	15	11	14	14
TEXAS_AND_GULF_WIND	18	14	18	20	14	18	18	13	17	18	13	14	11	10	15	13	14	19	17	19	12	17	15	3	30	14	16	22	19

Value Words	Worldwide ALL Perils	Worldwide WIND	Worldwide QUAKE	North America ALL Pe	North America WIND	North America QUAKE	International ALL Peril	International WIND	International QUAKE	California QUAKE	Hawaii WIND	Florida WIND	Gulf WIND	Texas WIND	North East WIND	Mid Atlantic WIND	SouthEast WIND	Europe1 (UK & EIRE)	Europe3 (Scandinavia	Europe2 (Continental	Japan WIND	Europe4 QUAKE	Japan QUAKE	Texas and Gulf WIND	North America x FLA	Europe Wind	New Madrid QUAKE	Pacific Northwest QU	MidWest TORNADO
WORLDWIDE_ALL_PERIL	1	9	9	8	15	15	8	16	17	17	16	15	15	15	14	14	16	14	16	16	16	17	17	13	13	16	13	15	14
WORLDWIDE_WIND	4	0	5	11	7	12	11	7	13	13	7	6	7	7	7	7	7	7	7	7	7	13	13	7	7	7	10	11	10
WORLDWIDE_EARTHQUAKE	8	8	5	14	14	12	16	17	13	13	15	15	16	15	14	15	14	15	15	15	15	13	13	15	14	14	11	11	14
NA_ALL_PERIL	4	12	13	4	11	12	4	12	13	13	12	12	11	11	10	10	12	9	12	12	12	13	13	8	8	12	9	13	10
NA_WIND	7	3	9	7	3	9	7	3	9	9	3	3	3	3	3	3	3	2	3	3	3	9	9	2	2	3	6	9	6
NA_EARTHQUAKE	11	11	9	10	10	9	12	13	9	9	11	12	12	11	10	11	10	10	11	11	11	9	9	10	9	10	7	9	10
INTL_ALL_PERIL	4	12	14	4	11	12	4	12	14	14	12	12	11	11	10	10	12	10	12	12	12	14	14	9	9	12	10	14	12
INTL_WIND	7	3	10	7	3	9	7	3	10	10	3	3	3	3	3	3	3	3	3	3	3	10	10	3	3	3	7	10	8
INTL_EARTHQUAKE	11	11	10	10	10	9	12	13	10	10	11	12	12	11	10	11	10	11	11	11	11	10	10	11	10	10	8	10	12
CALIFORNIA_EARTHQUAKE	16	16	15	14	14	12	17	18	15	5	16	15	17	16	15	16	15	16	16	16	16	14	13	16	14	15	12	11	16
HAWAII_WIND	9	5	10	9	5	10	9	5	10	10	0	5	5	5	5	5	5	5	5	5	4	10	9	5	5	5	8	10	10
FLORIDA_WIND	9	6	11	8	4	9	9	6	12	11	6	0	6	6	5	5	6	6	6	6	6	12	12	6	4	6	8	12	9
GULF_WIND	7	4	9	7	4	9	7	4	9	9	4	4	0	4	4	4	4	3	4	4	4	9	9	0	3	4	8	9	8
TEXAS_WIND	8	5	10	8	5	10	8	5	10	10	5	5	5	0	3	5	5	5	5	5	4	10	9	0	4	5	8	10	9
NORTH_EAST_WIND	12	9	14	7	4	9	12	9	14	14	9	9	9	8	0	9	9	8	9	9	9	14	14	8	4	9	12	13	13
MID_ATLANTIC_WIND	13	9	17	13	8	16	13	9	17	17	8	9	9	9	8	0	9	9	9	9	8	17	16	8	8	9	13	16	13
SOUTHEAST_WIND	12	8	12	10	6	11	12	9	12	12	8	8	8	6	5	8	0	8	8	8	8	12	12	6	6	8	11	9	10
EUROPE1_WIND	10	7	10	10	6	10	10	7	10	10	7	7	6	7	6	7	7	0	1	1	6	6	10	6	6	1	9	10	10
EUROPE3_WIND	10	7	10	10	6	10	10	7	10	10	7	7	6	7	6	7	7	1	0	1	6	6	10	6	6	1	9	10	10
EUROPE2_WIND	10	7	10	10	6	10	10	7	10	10	7	7	6	7	6	7	7	1	1	0	6	6	10	6	6	1	9	10	10
JAPAN_WIND	8	5	10	8	5	10	8	5	10	10	4	5	5	4	4	5	5	5	5	5	0	10	5	4	4	5	9	10	9
EUROPE4_EARTHQUAKE	14	15	10	13	13	10	15	17	10	10	15	16	15	15	13	15	14	9	9	9	14	5	10	14	13	8	10	10	14
JAPAN_EARTHQUAKE	12	13	10	11	12	10	13	15	10	10	12	14	14	12	11	13	12	13	13	13	8	10	5	12	11	12	10	10	13
TEXAS_AND_GULF_WIND	13	11	18	13	11	18	13	11	18	18	11	11	7	6	9	11	11	10	11	11	10	18	17	0	9	11	15	18	16

As you can see, the last two metrics, which are fuzzy string matching metrics, already have a natural tendency to give low scores to strings that are meant to match (down the diagonal). This is very good.

Application To allow the optimization of fuzzy matching, I weight each metric. As such, every application of fuzzy string match can weight the parameters differently. The formula that defines the final score is a simply combination of the metrics and their weights:

```
value = Min(phraseWeight*phraseValue,
wordsWeight*wordsValue)*minWeight
      + Max(phraseWeight*phraseValue,
wordsWeight*wordsValue)*maxWeight
      + lengthWeight*lengthValue
```

Using an optimization algorithm (neural network is best here because it is a discrete, multi-dimentional problem), the goal is now to maximize the number of matches. I created a function that detects the number of correct

matches of each set to each other, as can be seen in this final screenshot. A column or row gets a point if the lowest score is assigned the the string that was meant to be matched, and partial points are given if there is a tie for the lowest score, and the correct match is among the tied matched strings. I then optimized it. You can see that a green cell is the column that best matches the current row, and a blue square around the cell is the row that best matches the current column. The score in the bottom corner is roughly the number of successful matches and this is what we tell our optimization problem to maximize.

Phrase Weight	0.5		Min Weight		10																									
Word Weight	1.0		Max Weight		1																									
Length Weight	-0.3																													
Match Value	Worldwide ALL Perils	Worldwide WIND	Worldwide QUAKE	North America ALL Perils	North America WIND	North America QUAKE	International ALL Perils	International WIND	International QUAKE	California QUAKE	Hawaii WIND	Florida WIND	Gulf WIND	Texas WIND	North East WIND	Mid Atlantic WIND	SouthEast WIND	Europe1 (UK & EIRE) WIND	Europe3 (Scandinavia) WII	Europe2 (Continental) WII	Japan WIND	Europe4 QUAKE	Japan QUAKE	Texas and Gulf WIND	North America x FLA WIND	Europe Wind	New Madrid QUAKE	Pacific Northwest QUAKE	MidWest TORNADO	
WORLDWIDE_ALL_PERIL	11	58	53	72	95	95	77	101	102	101	99	88	97	97	88	98	95	118	124	124	103	95	100	103	139	99	92	124	98	1
WORLDWIDE_WIND	43	0.5	35	108	61	91	113	66	97	87	51	40	51	56	52	66	52	76	76	76	56	78	82	76	78	46	69	108	79	1
WORLDWIDE_EARTHQUAKE	63	61	34	113	98	72	120	111	83	77	102	93	103	102	88	104	92	119	128	128	107	76	80	115	150	101	75	90	88	1
NA_ALL_PERIL	44	71	82	43	79	85	43	85	91	87	62	72	65	65	74	79	76	97	123	128	61	78	63	86	88	67	78	115	84	0.5
NA_WIND	75	33	72	75	33	85	75	33	85	76	32	32	27	27	32	33	32	25	34	34	27	72	58	23	27	32	64	95	64	0
NA_EARTHQUAKE	99	81	58	102	84	62	109	87	67	53	65	77	76	70	69	80	70	108	118	118	65	49	38	88	99	69	46	76	74	0
INTL_ALL_PERIL	44	77	84	45	85	96	43	76	83	93	76	76	75	70	80	74	77	108	123	123	76	84	78	93	99	76	84	111	86	1
INTL_WIND	75	33	68	76	34	91	75	32	77	78	33	33	28	33	33	33	33	35	35	34	33	74	64	34	37	33	75	106	66	0
INTL_EARTHQUAKE	95	81	55	107	89	68	99	77	59	60	80	81	80	75	70	80	70	118	118	118	80	54	54	95	110	79	53	78	77	0
CALIFORNIA_EARTHQUAKE	116	109	83	123	103	81	121	107	84	34	108	92	103	113	93	105	98	125	135	135	108	82	80	115	151	107	86	75	105	1
HAWAII_WIND	91	49	79	96	53	88	97	54	93	79	0.5	40	39	35	49	53	49	56	55	56	29	74	54	54	57	40	72	106	74	1
FLORIDA_WIND	82	40	70	86	44	87	96	59	90	70	41	0.5	45	45	49	55	50	65	65	65	45	72	72	65	48	41	72	109	78	1
GULF_WIND	76	43	72	77	44	96	77	44	91	72	38	43	0.5	34	44	44	43	35	45	45	34	68	63	2.5	38	33	81	96	76	1
TEXAS_WIND	86	54	79	86	54	87	86	53	87	83	35	44	35	0.5	33	54	39	55	55	55	29	74	59	2.3	47	40	76	107	72	1
NORTH_EAST_WIND	86	54	79	75	43	73	104	63	88	94	53	53	62	47	1	63	29	86	97	96	58	78	83	82	47	53	87	91	83	1
MID_ATLANTIC_WIND	97	68	101	116	68	100	106	69	96	97	56	68	67	67	62	1	63	97	97	97	56	101	84	72	89	72	93	119	78	1
SOUTHEAST_WIND	90	53	82	97	55	85	114	73	96	91	52	52	52	40	25	62	0.5	86	86	85	52	77	76	66	67	52	90	96	74	1
EUROPE1_WIND	93	51	74	107	64	93	107	65	93	84	47	47	40	46	50	71	51	2.9	14	14	40	41	70	65	68	11	83	107	79	1
EUROPE3_WIND	93	51	74	107	64	93	107	65	93	84	47	47	40	46	50	71	51	13	3.3	14	40	41	70	65	68	11	83	107	79	1
EUROPE2_WIND	93	51	74	107	64	93	107	65	93	84	47	47	40	46	50	71	51	13	14	3.3	40	41	70	65	68	11	83	107	79	1
JAPAN_WIND	86	54	79	86	54	92	86	54	87	78	29	44	35	29	44	53	49	55	55	55	0.5	69	35	43	48	40	82	106	77	1
EUROPE4_EARTHQUAKE	98	94	64	111	93	70	118	107	75	69	98	94	97	93	82	100	83	87	97	97	92	34	63	104	141	66	69	84	88	1
JAPAN_EARTHQUAKE	101	92	65	114	96	74	116	99	74	60	81	88	92	85	76	88	76	116	120	125	61	59	34	91	120	86	65	78	83	1
TEXAS_AND_GULF_WIND	103	80	107	112	81	108	102	76	103	107	74	79	59	53	83	75	80	104	94	104	67	101	90	1.5	99	79	94	127	110	1
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	43.5

The algorithm was a wonderful success, and the solution parameters say a lot about this type of problem. You'll notice the optimized score was 44, and the best possible score is 48. The 5 columns at the end are decoys, and do not have any match at all to the row values. The more

decoys there are, the harder it will naturally be to find the best match.

In this particular matching case, the length of the strings are irrelevant, because we are expecting abbreviations that represent longer words, so the optimal weight for length is -0.3, which means we do not penalize strings which vary in length. We reduce the score in anticipation of these abbreviations, giving more room for partial word matches to supersede non-word matches that simply require less substitutions because the string is shorter.

The word weight is 1.0 while the phrase weight is only 0.5, which means that we penalize whole words missing from one string and value more the entire phrase being intact. This is useful because a lot of these strings have one word in common (the peril) where what really matters is whether or not the combination (region and peril) are maintained.

Finally, the min weight is optimized at 10 and the max weight at 1. What this means is that if the best of the two scores (value phrase and value words) isn't very good, the match is greatly penalized, but we don't greatly penalize the worst of the two scores. Essentially, this puts emphasis on requiring *either* the valueWord or valuePhrase to have a good score, but not both. A sort of "take what we can get" mentality.

It's really fascinating what the optimized value of these 5 weights say about the sort of fuzzy string matching taking place. For completely different practical cases of fuzzy

string matching, these parameters are very different. I've used it for 3 separate applications so far.

While unused in the final optimization, a benchmarking sheet was established which matches columns to themselves for all perfect results down the diagonal, and lets the user change parameters to control the rate at which scores diverge from 0, and note innate similarities between search phrases (which could in theory be used to offset false positives in the results)

Match Value	Worldw	Worldw	Worldw	North A	North A	North A	Internat	Internat	Internat	Californi	Hawaii V	Florida V	Gulf Wil	Texas W	North Ea	Mid Atla	SouthEa	Europe1	Europe3	Europe2	Japan W	Europe4	Japan Q	Texas an	North A	Europe V	New Ma	Pacific N	MidWes	
Worldwide ALL Perils	0	63	65	64	105	110	69	112	121	121	112	100	115	110	100	115	107	130	142	137	117	121	126	111	159	112	108	140	110	1
Worldwide WIND	49	0	35	117	64	99	122	69	106	101	54	42	54	59	54	69	54	86	88	88	59	91	96	79	100	49	80	117	85	1
Worldwide QUAKE	49	35	0	117	94	64	122	104	71	66	89	77	87	94	87	104	89	121	139	139	94	56	61	107	141	82	47	74	92	1
North America ALL Perils	68	135	139	0	61	61	78	130	139	144	135	131	143	133	98	131	125	141	150	145	140	144	149	129	79	140	124	151	147	1
North America WIND	101	72	104	49	0	35	121	82	119	114	72	68	87	77	42	75	67	100	107	107	82	114	114	82	20	77	96	128	117	1
North America QUAKE	106	107	74	49	35	0	126	117	84	79	107	103	115	112	75	110	102	131	142	142	117	79	84	115	63	110	63	93	124	1
International ALL Perils	75	140	142	78	129	134	0	63	65	142	140	133	139	131	126	129	140	148	150	143	138	142	142	117	175	140	138	148	134	1
International WIND	108	77	112	116	78	113	49	0	35	112	77	70	83	70	70	73	82	112	107	95	75	112	112	80	119	77	110	125	99	1
International QUAKE	113	112	77	121	113	78	49	35	0	77	112	105	111	105	103	108	117	143	142	130	110	77	77	113	161	110	77	90	106	1
California QUAKE	109	101	70	122	104	69	124	106	75	0	91	77	89	101	99	101	101	132	136	141	91	63	56	109	141	94	59	72	101	1
Hawaii WIND	98	50	85	109	60	95	110	61	100	85	0	40	40	35	50	55	50	68	68	70	28	80	63	62	80	40	76	115	80	1
Florida WIND	88	42	77	99	50	93	109	62	99	77	42	0	47	47	50	61	52	77	78	78	47	84	84	73	70	42	81	119	83	1
Gulf WIND	87	48	83	91	53	103	91	53	103	83	38	43	0	33	50	51	48	48	60	59	33	73	68	10	62	33	91	110	81	1
Texas WIND	96	55	90	99	61	100	99	60	95	95	35	45	35	0	37	61	40	68	69	68	28	80	63	9	70	40	81	120	78	1
North East WIND	94	58	93	84	38	73	114	68	103	103	58	58	68	51	0	68	33	96	108	103	63	93	98	81	65	58	94	101	91	1
Mid Atlantic WIND	111	73	114	121	71	112	121	73	114	109	61	73	73	73	66	0	68	103	103	103	61	119	102	76	109	78	96	132	91	1
SouthEast WIND	99	56	89	105	57	92	124	78	109	99	56	56	56	42	25	66	0	96	99	97	56	89	89	73	88	56	97	103	80	1
Europe1 (UK & EIRE) WIND	142	118	151	145	116	149	152	128	161	156	128	123	124	128	116	119	118	0	86	81	126	111	146	117	170	91	143	156	138	1
Europe3 (Scandinavia) WIND	152	126	157	152	119	152	152	121	152	148	124	126	130	129	124	119	131	84	0	56	122	126	152	112	182	99	143	155	157	1
Europe2 (Continental) WIND	147	126	159	147	119	152	145	109	140	155	136	124	125	124	119	119	117	77	56	0	127	126	157	122	171	99	150	143	150	1
Japan WIND	97	55	90	100	62	105	100	61	100	85	28	45	35	28	49	55	50	68	68	69	0	75	35	51	71	40	88	120	83	1
Europe4 QUAKE	105	89	50	120	102	61	120	104	61	55	84	84	75	84	85	109	89	55	78	78	77	0	40	100	134	40	55	65	92	1
Japan QUAKE	98	90	55	101	100	62	100	100	61	50	63	80	68	63	86	90	85	100	122	123	35	40	0	86	113	73	55	65	90	1
Texas and Gulf WIND	111	87	121	121	82	121	111	82	121	121	82	87	64	57	83	82	87	105	102	112	75	116	104	0	118	87	110	141	122	1
North America x FLA WIND (Provisional)	201	210	229	133	138	165	199	201	225	229	210	204	214	208	175	199	200	200	208	209	213	244	239	196	0	215	212	221	215	1
Europe Wind	98	47	78	110	61	98	110	67	98	88	42	42	35	42	50	72	52	23	25	25	42	42	73	63	81	0	86	110	85	1
New Madrid QUAKE	104	88	57	114	98	63	124	108	77	67	86	91	101	93	96	92	103	127	133	138	98	67	65	102	150	96	0	100	101	1
Pacific Northwest QUAKE	140	129	96	141	128	93	142	133	100	88	131	133	136	136	103	138	115	142	151	141	138	103	103	139	161	134	104	0	129	1
MidWest TORNADO	110	95	104	133	115	122	130	105	114	109	90	95	95	90	93	89	90	122	145	140	95	104	104	118	151	95	103	127	0	1
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	58

Further Applications

This solution has potential to be used anywhere where the user wishes to have a computer system identify a string in a set of strings where there is no perfect match. (Like an approximate match vlookup for strings).

So what you should take from this, is that you probably want to use a combination of high level heuristics (finding words from one phrase in the other phrase, length of both

phrases, etc) along with the implementation of the Levenshtein distance algorithm. Because deciding which is the "best" match is a heuristic (fuzzy) determination - you'll have to come up with a set of weights for any metrics you come up with to determine similarity.

With the appropriate set of heuristics and weights, you'll have your comparison program quickly making the decisions that you would have made.

Share Improve this answer Follow

edited Apr 12, 2017 at 13:38

answered May 2, 2011 at 16:40

Alain

27.2k ● 21 ● 119 ● 197

16 Bonus: If anyone wants to include additional metrics into their weighted heuristic, (since I only provided 3 which weren't all that linearly independent) - here is a whole list on wikipedia: en.wikipedia.org/wiki/String_metric – Alain Apr 9, 2012 at 20:22

1 If S2 has a lot of words (and creating many small objects is not prohibitively slow in your language of choice) a trie can speed things up. Fast and Easy Levenshtein distance using a Trie is a great article about tries. – JanX2 May 4, 2012 at 19:20

1 @Alain This is an interesting approach! I am just playing a bit with your idea (in C++) but do not understand one point, the value of `valuePhrase`. If I see right in your code, its the return value of the Levenshtein distance function. How come it is a double/float value in the 'abcd efgh' search table? Levenshtein distance is an integer value and I cannot

see further calculations in your code that makes it a float. What do I miss? – Andreas W. Wylach Mar 31, 2017 at 4:23

1 @AndreasW.Wylach Great observation. The VBA I showed was just to compute the Levenshtein distance, but the heuristic I used in my spreadsheet was `=valuePhrase(A2,B2)-0.8*ABS(LEN(B2)-LEN(A2))` So I was reducing the penalty of the levenstein distance by 80% of the difference in the length of the two "phrases". This way, "phrases" that have the same length suffer the full penalty, but "phrases" which contain 'additional information' (longer) but aside from that still mostly share the same characters suffer a reduced penalty. – Alain Apr 12, 2017 at 13:30

1 @Alain Thanks for getting back to my question, I appreciate that. Your explanation makes things clearer now. Meanwhile I implemented a value_phrase method that gets a little deeper into analysing the tokens of a phrase a bit more, that is the order/positions of the phrase tokens, non-query token sequences and it accepts a bit more fuzziness when it comes to something like "acbd" compared to "abcd". Tendency of the phrase_value scores equals yours, but get a bit lower here and there. Once again, great workout and it gave me inspiration for the fuzzy search algorithm!
– Andreas W. Wylach Apr 14, 2017 at 10:21

94

This problem turns up all the time in bioinformatics. The accepted answer above (which was great by the way) is known in bioinformatics as the Needleman-Wunsch (compare two strings) and Smith-Waterman (find an approximate substring in a longer string) algorithms. They work great and have been workhorses for decades.

But what if you have a million strings to compare? That's a trillion pairwise comparisons, each of which is O(n*m)! Modern DNA sequencers easily generate a *billion* short DNA sequences, each about 200 DNA "letters" long. Typically, we want to find, for each such string, the best match against the human genome (3 billion letters). Clearly, the Needleman-Wunsch algorithm and its relatives will not do.

This so-called "alignment problem" is a field of active research. The most popular algorithms are currently able to find inexact matches between 1 billion short strings and the human genome in a matter of hours on reasonable hardware (say, eight cores and 32 GB RAM).

Most of these algorithms work by quickly finding short exact matches (seeds) and then extending these to the full string using a slower algorithm (for example, the Smith-Waterman). The reason this works is that we are really only interested in a few close matches, so it pays off to get rid of the 99.9...% of pairs that have nothing in common.

How does finding exact matches help finding *inexact* matches? Well, say we allow only a single difference between the query and the target. It is easy to see that this difference must occur in either the right or left half of the query, and so the other half must match exactly. This idea can be extended to multiple mismatches and is the basis for the ELAND algorithm commonly used with Illumina DNA sequencers.

There are many very good algorithms for doing exact string matching. Given a query string of length 200, and a target string of length 3 billion (the human genome), we want to find any place in the target where there is a substring of length k that matches a substring of the query exactly. A simple approach is to begin by indexing the target: take all k-long substrings, put them in an array and sort them. Then take each k-long substring of the query and search the sorted index. ~~Sort and~~ search can be done in O(log n) time.

But storage can be a problem. An index of the 3 billion letter target would need to hold 3 billion pointers and 3 billion k-long words. It would seem hard to fit this in less than several tens of gigabytes of RAM. But amazingly we can greatly compress the index, using the Burrows-Wheeler transform, and it will still be efficiently queryable. An index of the human genome can fit in less than 4 GB RAM. This idea is the basis of popular sequence aligners such as Bowtie and BWA.

Alternatively, we can use a suffix array, which stores only the pointers, yet represents a simultaneous index of all suffixes in the target string (essentially, a simultaneous index for all possible values of k; the same is true of the Burrows-Wheeler transform). A suffix array index of the human genome will take 12 GB of RAM if we use 32-bit pointers.

The links above contain a wealth of information and links to primary research papers. The ELAND link goes to a

PDF with useful figures illustrating the concepts involved, and shows how to deal with insertions and deletions.

Finally, while these algorithms have basically solved the problem of (re)sequencing single human genomes (a billion short strings), DNA sequencing technology improves even faster than Moore's law, and we are fast approaching trillion-letter datasets. For example, there are currently projects underway to sequence the genomes of 10,000 vertebrate species, each a billion letters long or so. Naturally, we will want to do pairwise inexact string matching on the data...

Share Improve this answer Follow

edited May 4, 2012 at 9:20

answered May 4, 2012 at 8:07

Sten L

1,782 ● 1 ● 12 ● 14

3 Really good run-down. A couple of corrections: Sorting the infixes takes O(n) at least, not O(log n). And since O(log n) search is actually too slow in practice, you'd normally build an additional table to get O(1) lookup (q-gram index). Furthermore, I'm not sure why you treat this differently from the suffix array – it's just an optimisation of the latter, no (sorting fixed-length infixes instead of suffixes since we don't actually need more than a fixed length). – Konrad Rudolph May 4, 2012 at 8:52

1 Furthermore, these algorithms are still impractical for *de novo* sequencing. They've solved the sequencing of human genomes only insofar as we have a reference sequence that can be used to map against. But for de novo assembly other

algorithms are needed (well, there are some aligners which are based on mapping but stitching the contigs together is a whole 'nother problem). Finally, shameless plug: my bachelor thesis contains a detailed description of the ELAND algorithm. – Konrad Rudolph May 4, 2012 at 8:55

1 Thanks. I edited out the error. The reason I started by describing the fixed-length array was because it's easy to understand. Suffix arrays and BWT are a bit harder to grasp, but actually we do sometimes want to use an index with different values of k. For example, STAR uses suffix arrays to efficiently find *spliced* alignments. This is of course useful for aligning RNA to the genome. – Sten L May 4, 2012 at 9:16

30

I contest that choice B is closer to the test string, as it's only 4 characters(and 2 deletes) from being the original string. Whereas you see C as closer because it includes both brown and red. It would, however, have a greater edit distance.

There is an algorithm called Levenshtein Distance which measures the edit distance between two inputs.

Here is a tool for that algorithm.

1. Rates choice A as a distance of 15.
2. Rates choice B as a distance of 6.
3. Rates choice C as a distance of 9.

EDIT: Sorry, I keep mixing strings in the levenshtein tool. Updated to correct answers.

Share Improve this answer Follow

edited May 2, 2011 at 16:46

answered May 2, 2011 at 16:29

adorablepuppy
1,077 ● 1 ● 7 ● 13

2 Ok, I guess that is true. I'll take a look at this. I personally don't care *how* close it is to the target as long as it is pretty dang close. No need for perfection ;) Points for you until I can verify the results of your answer :) – Freesnöw May 2, 2011 at 16:31

20

Lua implementation, for posterity:

```
function levenshtein_distance(str1, str2)
    local len1, len2 = #str1, #str2
    local char1, char2, distance = {}, {}, {}
    str1:gsub('.', function (c)
table.insert(char1, c) end)
    str2:gsub('.', function (c)
table.insert(char2, c) end)
    for i = 0, len1 do distance[i] = {} end
    for i = 0, len1 do distance[i][0] = i end
    for i = 0, len2 do distance[0][i] = i end
    for i = 1, len1 do
        for j = 1, len2 do
            distance[i][j] = math.min(
                distance[i-1][j  ] + 1,
                distance[i  ][j-1] + 1,
                distance[i-1][j-1] + (char1[i] ==
char2[j] and 0 or 1)
                )
        end
    end
```

```
    return distance[len1][len2]
end
```

Share Improve this answer

Follow

answered Apr 27, 2012 at 19:32

Mud

28.9k ● 11 ● 61 ● 97

17

You might find this library helpful!
http://code.google.com/p/google-diff-match-patch/

It is currently available in Java, JavaScript, Dart, C++, C#, Objective C, Lua and Python

It works pretty well too. I use it in a couple of my Lua projects.

And I don't think it would be too difficult to port it to other languages!

Share Improve this answer

Follow

answered May 21, 2012 at 13:21

SatheeshJM

3,634 ● 8 ● 38 ● 62

14

You might be interested in this blog post.

http://seatgeek.com/blog/dev/fuzzywuzzy-fuzzy-string-matching-in-python

Fuzzywuzzy is a Python library that provides easy distance measures such as Levenshtein distance for string matching. It is built on top of difflib in the standard

library and will make use of the C implementation Python-levenshtein if available.

http://pypi.python.org/pypi/python-Levenshtein/

Share Improve this answer

Follow

answered May 4, 2012 at 3:32

jseabold

8,283 ● 2 ● 44 ● 53

> For others reading this, Fuzzywuzzy actually implements a lot of the ideas in Alain's wonderful post. If you're actually looking to use some of those ideas its a great place to start. – Gregory Arenius Sep 10, 2018 at 18:59

2

If you're doing this in the context of a search engine or frontend against a database, you might consider using a tool like Apache Solr, with the ComplexPhraseQueryParser plugin. This combination allows you to search against an index of strings with the results sorted by relevance, as determined by Levenshtein distance.

We've been using it against a large collection of artists and song titles when the incoming query may have one or more typos, and it's worked pretty well (and remarkably fast considering the collections are in the millions of strings).

Additionally, with Solr, you can search against the index on demand via JSON, so you won't have to reinvent the

solution between the different languages you're looking at.

Share Improve this answer Follow

answered May 4, 2012 at 18:21

Spoom
168 • 9

2

The problem is hard to implement if the input data is too large (say millions of strings). I used elastic search to solve this.

Quick start : https://www.elastic.co/guide/en/elasticsearch/client/net-api/6.x/elasticsearch-net.html

Just insert all the input data into DB and you can search any string based on any edit distance quickly. Here is a C# snippet which will give you a list of results sorted by edit distance (smaller to higher)

```
var res = client.Search<ClassName>(s => s
    .Query(q => q
    .Match(m => m
        .Field(f => f.VariableName)
        .Query("SAMPLE QUERY")
        .Fuzziness(Fuzziness.EditDistance(5))
    )
));
```

Share Improve this answer Follow

edited Mar 11, 2019 at 22:32

answered May 12, 2017 at 14:13

cegprakash

3,127 ● 1 ● 36 ● 62

What library are you using? Some more information is needed for this to be helpful. – bets Mar 11, 2019 at 11:41

1

A very, very good resource for these kinds of algorithms is Simmetrics: http://sourceforge.net/projects/simmetrics/

Unfortunately the awesome website containing a lot of the documentation is gone :(In case it comes back up again, its previous address was this: http://www.dcs.shef.ac.uk/~sam/simmetrics.html

Voila (courtesy of "Wayback Machine"): http://web.archive.org/web/20081230184321/http://www.dcs.shef.ac.uk/~sam/simmetrics.html

You can study the code source, there are dozens of algorithms for these kinds of comparisons, each with a different trade-off. The implementations are in Java.

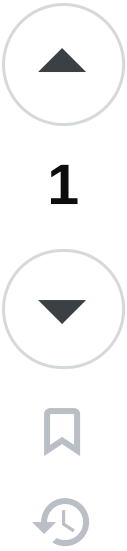
Share Improve this answer Follow

edited May 4, 2012 at 14:48

answered May 4, 2012 at 14:39

oblio

1,623 ● 1 ● 18 ● 42



To query a large set of text in efficient manner you can use the concept of Edit Distance/ Prefix Edit Distance.

> Edit Distance ED(x,y): minimal number of transfroms to get from term x to term y

But computing ED between each term and query text is resource and time intensive. Therefore instead of calculating ED for each term first we can extract possible matching terms using a technique called Qgram Index. and then apply ED calculation on those selected terms.

An advantage of Qgram index technique is it supports for Fuzzy Search.

One possible approach to adapt QGram index is build an Inverted Index using Qgrams. In there we store all the words which consists with particular Qgram, under that Qgram.(Instead of storing full string you can use unique ID for each string). You can use Tree Map data structure in Java for this. Following is a small example on storing of terms

> col : **col**mbia, **col**ombo, gan**col**a, ta**col**ama

Then when querying, we calculate the number of common Qgrams between query text and available terms.

```
Example: x = HILLARY, y = HILARI(query term)
Qgrams
$$HILLARY$$ -> $$H, $HI, HIL, ILL, LLA, LAR, ARY,
RY$, Y$$
$$HILARI$$ -> $$H, $HI, HIL, ILA, LAR, ARI, RI$,
I$$
number of q-grams in common = 4
```

number of q-grams in common = 4.

For the terms with high number of common Qgrams, we calculate the ED/PED against the query term and then suggest the term to the end user.

you can find an implementation of this theory in following project(See "QGramIndex.java"). Feel free to ask any questions. https://github.com/Bhashitha-Gamage/City_Search

To study more about Edit Distance, Prefix Edit Distance Qgram index please watch the following video of Prof. Dr Hannah Bast
https://www.youtube.com/embed/6pUg2wmGJRo
(Lesson starts from 20:06)

Share Improve this answer

Follow

answered Apr 3, 2017 at 6:30

Baxter
233 ● 2 ● 10

0

Here you can have a golang POC for calculate the distances between the given words. You can tune the `minDistance` and `difference` for other scopes.

Playground: https://play.golang.org/p/NtrBzLdC3rE

```
package main

import (
    "errors"
    "fmt"
    "log"
    "math"
    "strings"
)

var data string = `THE RED COW JUMPED OVER THE GREEN C
OVER THE RED COW-THE RED FOX JUMPED OVER THE BROWN COW

const minDistance float64 = 2
const difference float64 = 1

type word struct {
    data    string
    letters map[rune]int
}

type words struct {
    words []word
}

// Print prettify the data present in word
func (w word) Print() {
    var (
        lenght int
        c      int
        i      int
        key    rune
    )
    fmt.Printf("Data: %s\n", w.data)
    lenght = len(w.letters) - 1
    c = 0
    for key, i = range w.letters {
        fmt.Printf("%s:%d", string(key), i)
        if c != lenght {
            fmt.Printf(" | ")
        }
```

```
        c++
    }
    fmt.Printf("\n")
}

func (ws words) fuzzySearch(data string) ([]word, erro
    var (
        w      word
        err    error
        founds []word
    )
    w, err = initWord(data)
    if err != nil {
        log.Printf("Errors: %s\n", err.Error())
        return nil, err
    }
    // Iterating all the words
    for i := range ws.words {
        letters := ws.words[i].letters
        //
        var similar float64 = 0
        // Iterating the letters of the input data
        for key := range w.letters {
            if val, ok := letters[key]; ok {
                if math.Abs(float64(val-w.letters[key]
                    similar += float64(val)
                }
            }
        }

        lenSimilarity := math.Abs(similar - float64(le
strings.Count(data, " ")))
        log.Printf("Comparing %s with %s i've found %f
weight %f", data, ws.words[i].data, similar, lenSimila
        if lenSimilarity <= difference {
            founds = append(founds, ws.words[i])
        }
    }

    if len(founds) == 0 {
        return nil, errors.New("no similar found for d
    }

    return founds, nil
```

```
}

func initWords(data []string) []word {
    var (
        err   error
        words []word
        word  word
    )
    for i := range data {
        word, err = initWord(data[i])
        if err != nil {
            log.Printf("Error in index [%d] for data:
        } else {
            words = append(words, word)
        }
    }
    return words

}

func initWord(data string) (word, error) {
    var word word

    word.data = data
    word.letters = make(map[rune]int)
    for _, r := range data {
        if r != 32 { // avoid to save the whitespace
            word.letters[r]++
        }

    }
    return word, nil
}
func main() {
    var ws words
    words := initWords(strings.Split(data, "-"))
    for i := range words {
        words[i].Print()
    }
    ws.words = words

    solution, _ := ws.fuzzySearch("THE BROWN FOX JUMPE
    fmt.Println("Possible solutions: ", solution)
```

```
}
```

Share Improve this answer Follow

answered Feb 9, 2020 at 20:32

alessiosavi

3,027 ● 2 ● 20 ● 41

0

There is one more similarity measure which I once implemented in our system and was giving satisfactory results :-

Use Case

There is a user query which needs to be matched against a set of documents.

Algorithm

1. Extract keywords from the user query (relevant POS TAGS - Noun, Proper noun).
2. Now calculate score based on below formula for measuring similarity between user query and given document.

For every keyword extracted from user query :-

- Start searching the document for given word and for every subsequent occurrence of that word in the document decrease the rewarded points.

In essence, if first keyword appears 4 times in the document, the score will be calculated as :-

- first occurrence will fetch '1' point.
- Second occurrence will add 1/2 to calculated score
- Third occurrence would add 1/3 to total
- Fourth occurrence gets 1/4

Total similarity score = 1 + 1/2 + 1/3 + 1/4 = 2.083

Similarly, we calculate it for other keywords in user query.

Finally, the total score will represent the extent of similarity between user query and given document.

Share Improve this answer

Follow

answered Oct 3, 2020 at 15:04

ravi

10.7k ● 1 ● 20 ● 36

0

Here is a quick solution that doesn't depend on any libraries, and works well enough for things like autocomplete forms:

```
function compare_strings(str1, str2) {
    arr1 = str1.split("");
    arr2 = str2.split("");
    res = arr1.reduce((a, c) => a +
arr2.includes(c), 0);
    return(res)
}
```

Can use in an autocomplete input like this:

HTML:

```
<div id="wrapper">
    <input id="tag_input" placeholder="add
tags..."></input>
    <div id="hold_tags"></div>
</div>
```

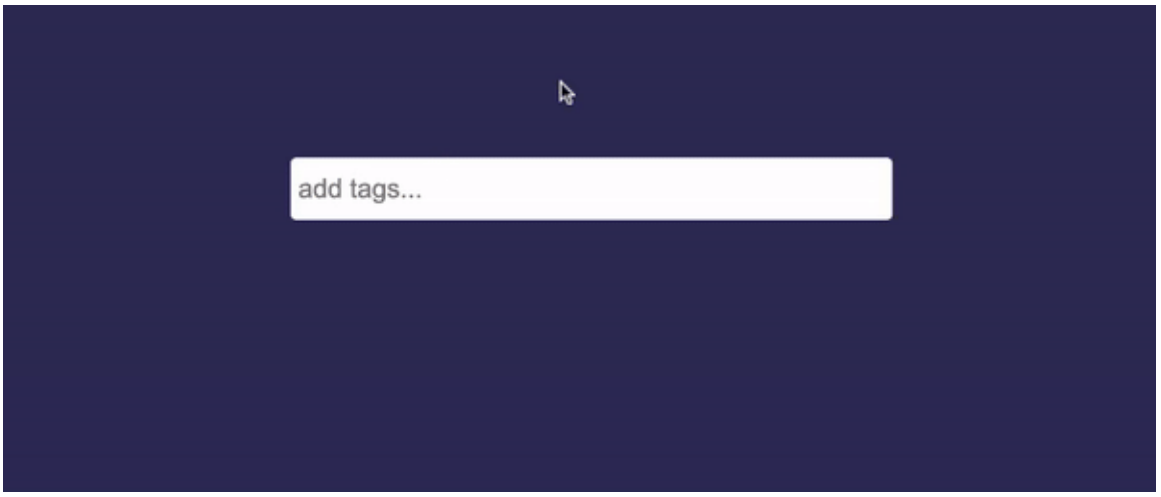
CSS:

```
body {
  background: #2c2c54;
  display: flex;
  justify-content: center;
  align-items: center;
}

input {
  height: 40px;
  width: 400px;
  border-radius: 4px;
  outline: 0;
  border: none;
  padding-left: 5px;
  font-size: 18px;
}

#wrapper {
  height: auto;
  background: #40407a;
}

.tag {
  background: #ffda79;
  margin: 4px;
  padding: 5px;
  border-radius: 4px;
  box-shadow: 2px 2px 2px black;
  font-size: 18px;
  font-family: arial;
  cursor: pointer;
}
```

JS:

```
const input =
document.getElementById("tag_input");
const wrapper =
document.getElementById("wrapper");
const hold_tags =
document.getElementById("hold_tags");
const words = [
  "machine",
  "data",
  "platform",
  "garbage",
  "twitter",
  "knowledge"
];
input.addEventListener("input", function (e) {
  const value =
document.getElementById(e.target.id).value;
  hold_tags.replaceChildren();
  if (value !== "") {
    words.forEach(function (word) {
      if (compare_strings(word, value) >
value.length - 1) {
        const tag = document.createElement("div");
        tag.className = "tag";
        tag.innerText = word;
        hold_tags.append(tag);
      }
    });
  }
});

function compare_strings(str1, str2) {
  arr1 = str1.split("");
  arr2 = str2.split("");
  res = arr1.reduce((a, c) => a +
arr2.includes(c), 0);
  return res;
}
```

Result:



Share Improve this answer

Follow

answered Nov 3, 2022 at 3:56

Cybernetic

13.3k ● 16 ● 103 ● 148

Operator '+' cannot be applied to types 'number' and 'boolean'.ts(2365) – Dimas Lanjaka Jan 22 at 22:17

0

A sample using C# is here.

```
public static void Main()
{
    Console.WriteLine("Hello World " +
        LevenshteinDistance("Hello", "World"));

    Console.WriteLine("Choice A " +
        LevenshteinDistance("THE BROWN FOX JUMPED OVER
                            "THE RED COW JUMPED OVER T

    Console.WriteLine("Choice B " +
        LevenshteinDistance("THE BROWN FOX JUMPED OVER
                            "THE RED COW JUMPED OVER T
    Console.WriteLine("Choice C " +
        LevenshteinDistance("THE BROWN FOX JUMPED OVER
                            "THE RED FOX JUMPED OVER T
}

public static float LevenshteinDistance(string a, stri
```

```
{
    var rowLen = a.Length;
    var colLen = b.Length;
    var maxLen = Math.Max(rowLen, colLen);

    // Step 1
    if (rowLen == 0 || colLen == 0)
    {
        return maxLen;
    }

    /// Create the two vectors
    var v0 = new int[rowLen + 1];
    var v1 = new int[rowLen + 1];

    /// Step 2
    /// Initialize the first vector
    for (var i = 1; i <= rowLen; i++)
    {
        v0[i] = i;
    }

    // Step 3
    /// For each column
    for (var j = 1; j <= colLen; j++)
    {
        /// Set the 0'th element to the column number
        v1[0] = j;

        // Step 4
        /// For each row
        for (var i = 1; i <= rowLen; i++)
        {
            // Step 5
            var cost = (a[i - 1] == b[j - 1]) ? 0 : 1;

            // Step 6
            /// Find minimum
            v1[i] = Math.Min(v0[i] + 1, Math.Min(v1[i
cost));
        }

        /// Swap the vectors
        var vTmp = v0;
```

```
            v0 = v1;
            v1 = vTmp;
        }

        // Step 7
        /// The vectors were swapped one last time at the
        /// that is why the result is now in v0 rather tha
        return v0[rowLen];
}
```

The output is:

```
Hello World 4
Choice A 15
Choice B 6
Choice C 8
```

Share Improve this answer

Follow

edited Nov 12 at 7:31

CarenRose

1,316 ● 1 ● 14 ● 25

answered Sep 10, 2020 at 20:19

John Henckel

11.3k ● 4 ● 89 ● 90