A Simple Fuzzy Search Algorithm

Abdullah Saquib (Independent Researcher)

Introduction

The algorithm compares two strings to generate a number between 0 and 1 based on how well the two strings matches. This algorithm consists of two functions `fuzzy_compare_words` and `compare_strings`.

`fuzzy_compare_words(word1, word2)` take two strings as input. The second argument is the correct word and the string to be compared is passed as the first argument. It returns a real number between 0 and 1, which is a measure of how well the two words matches with each other. The number 0 represents no match at all, and the number 1 represents complete matching. A number between 0 and 1 represents that the two words matches partially.

For example:

`fuzzy_compare_words("John","John")` will return 1

`fuzzy_compare_words("Doe","John")` will return 0

`fuzzy_compare_words("Jones","John")` will return 0.625

The number 0.625 signifies that about 62.5% of first word matches with the second word.

`compare_strings(arg1, arg2)` compare two strings to return a number between 0 and 1, measuring how much of the first string *is present* in the second string. The convention is that the second argument is the string in which the string as the first argument is to be searched. The number 0 represents that the second string does not contain the first string at all. The number 1 represents that the first string is present in the second string. A number between 0 and 1 represents that the first string is partially present in the second string.

For example:

`compare_strings("The string", "This is the string")` returns 1

`compare_strings("My book", " This is the string")` returns 0

`compare_strings("Strong man", " This is the string")` returns 0.41667

The third example shows partial matching because “strong” and “string” have almost the same letters in the same sequence.

How do the function `fuzzy_compare_words` work ?

Consider we have two arguments `arg1` and `arg2`. `marks` is a temporary variable whose value is incremented based on element-wise matching between the two arguments. The initial value of `marks` is 0.

`arg1[i]` is checked first with `arg2[i]`, then with `arg2[i+1]` and finally with `arg2[i-1]`. `marks` is incremented by 1 or 0.5 based on whether `arg1[i]` matches with `arg2[i]` or `arg2[i±1]`. `marks` is not incremented if there is no match.

The function first checks if `arg1[0]` and `arg2[0]` are the same characters. If they match, `marks` is incremented by 1. If they do not match, the function checks if the next element of `arg2` (that is, `arg2[1])` matches with `arg[0]`. If they do, `marks` is incremented by 0.5 else marks is not incremented and the function moves onto the next element of `arg1` (that is, `arg1[1]`).

The function checks if `arg1[1]` stores the same character as `arg2[1]`, if yes `marks` is incremented by 1. If they are not the same characters the function checks if `arg2[2]` stores the same character as `arg1[1]`, if they are the same character `marks` is increased by 0.5. If they are not the same characters, the function checks if `arg2[0]` stores the same character as `arg1[1]`, if yes `marks` is incremented by 0.5 else the function moves onto the next element.

Similar checks are made for next elements of `arg1` with elements of `arg2`.

After running the above checks over all the elements of `arg1`, `marks` is divided by the length of the smallest argument. For example if `arg1` have 4 elements, and `arg2` have 6 elements, `marks` will be divided by 4 and the value computed will be returned by the function. The reason is we want a complete match (i.e. returned value 1) for words like “call”, “calling”, “called”.

Suppose

First argument (arg1)

Index	0	1	2	3	4	
Element	J	O	N	E	S	

Second argument (arg2)

Index	0	1	2	3
Element	J	O	H	N

The algorithm work as follows:

i	**0**	**1**	**2**	**3**	**4**
arg1[i]	J	O	N	E	S
arg2[i]	J	O	H	N	
if arg1[i]==arg2[i]	Yes	Yes	No	No	N/C
if arg1[i]==arg2[i+1]	N/C	N/C	Yes	N/C	N/C
if arg1[i]==arg2[i-1]	N/C	N/C	N/C	N/C	N/C
marks	1	2	2.5	2.5	

where N/C stands for Not Checked.

The value returned by the function will be 2.5/4 = 0.625

How do the function `compare_strings` works ?

Consider we have two strings passed into parameters `search_string` and `db_string` of the function. First, the function splits these strings into corresponding arrays of the words making the two strings. Then it calls `fuzzy_compare_words` with each pair of the words in the two strings, with the words of the `search_string` passed as the first argument and the words of the `db_string` passed as the second argument.

A score variable is incremented by the *maximum* `marks` value returned for each word of `search_string`. Initially, the value of `score` is 0.

Finally, the `score` is divided by the number of words in the `search_string` and the value is returned.

Consider the following scenario in which `search_string = "The strong man"` and `db_string = "This is the string"`

search_string	**Word 1**	**Word 2**	**Word 3**
Strong man	The	strong	man

db_string	**Word 1**	**Word 2**	**Word 3**	**Word 4**
This is the string	This	is	the	string

In the following table the method behind the `compare_strings` is illustrated. The grey shaded cells of the table are the values returned by the `fuzzy_compare_words(arg1, arg2)`.

arg2 ↓	**arg1→**	The	strong	man
This		0.66667	0.25	0
is		0	0.25	0
the		1.0	0.16667	0
string		0.16667	0.83333	0
Maximum marks		1.0	0.83333	0
score		1.0	1.83333	1.83333

The value returned by the `compare_strings` function is

$$value\ returned = \frac{score}{number\ of\ words\ in\ search\_string}$$

$$value\ returned = \frac{1.83333}{3} = 0.61111$$

This signifies that about 61% of the searched string (“The strong man”) is contained in the string “This is the string”.

Another example: Searching “Adventure President” and “Alice Wonderland” in the book titles in the following table:

Book Titles ↓	**search_string →**	“Adventure President”	“Alice Wonderland”
`The Absolutely True Diary`		0.3264	0.1625
`The Adventures of Captain Hatteras`		0.6667	0.3062
`After the Wreck`		0.4	0.25
`Alcatraz Versus the Evil Librarians`		0.2604	0.4
`Alice Adventures in Wonderland`		0.5278	1.0
`Broken Piano for President`		0.5833	0.2667
`Charlie and the Chocolate Factory`		0.4167	0.3333
`God Bless You`		0.2833	0.35
`Twenty Thousands Leagues Under the Sea`		0.3167	0.3
`The Very Hungry Caterpillar`		0.25	0.225

Application

We can sort an array of strings based on how well the strings match with the searched string. We can use `compare_strings` to compare each string in the array with the searched string. And then use the values returned by the function to rank the strings of the array. The algorithm can also be used in auto-correction.