Some thoughts about normalizing an edit distance

Xavier Dupré
http://www.xavierdupre.fr/

3 mai 2013

Résumé

An edit distance is usually used to compute a distance between two words. The basic version gives the same weight to every mistake or operation which means every comparison, every insertion or deletion between two words weight the same. The sum of all these errors gives a integer score and most of time, it is better to keep that score within an interval such as [0,1]. Most of the time, we divide the edit distance by the length of the longest word involved in that distance. This score is not a distance anymore in a sense it does not verify the triangular inequality. There are many option and the following document studies some aspects of normalizations. Based on that, the maximum length is not the best option, the minimum length is not very good either. A third option studied in that document seems to be the best, specially if this distance is used by a machine learned model as a feature.

Table des matières

1 Normalized edit distance

Considering two strings or two sequences $(a_1, ..., a_I)$, $(b_1, ..., b_J)$, Levenstein's edit distance is defined as follows :

$$d(0,i) = i \;\forall i \leqslant I \quad (1)$$

$$d(j,0) = j \;\forall j \leqslant J \quad (2)$$

$$d(i,j) = \min \left\{ \begin{array}{ll} d(i-1,j)+1 & \text{if } i>0 \\ d(i,j-1)+1 & \text{if } j>0 \\ d(i-1,j-1)+ \mathbb{1}_{\{a_i \neq b_j\}} & \text{if } i,j>0 \end{array} \right. \quad (3)$$

$d(i,j)$ defines the distance between subsequences $(a_1, ..., a_i)$ and $(b_1, ..., b_j)$. The final distance is $d(a,b) = d(I,J)$. A optimal path can be built by following every couple (i,j) which was chosen to minimize the distance. It can be used to align both sequences and find the shortest srt of operations which transform the first sequence into the second one.

This edit distance is proven to be distance, it verifies :

1. $d(s_1, s_2) = 0 \Longleftrightarrow s_1 = s_2$
2. $d(s_1, s_2) = d(s_2, s_1)$
3. $d(s_1, s_3) \leqslant d(s_1, s_2) + d(s_2, s_3)$

The two first condition are easy to prove. The last one can be proven by considering the optimal path p_a between (s_1, s_2) and p_b between (s_2, s_3). We could describe a path by sequence of four letters :

i	insertion of a letter from sequence 2
d	deletion of a letter from sequence 1
n	comparison of two different letters
e	comparison of two equivalent letters

A path can be eeeienndeeen for example. We can arrange a third path by merging the first ones. Because a path can be seen as function which transform s_1 into s_2, we can compose two functions and get a third path $p = p_b \otimes p_a$. This path is not optimal. So $d(s_1, s_3) \leqslant d(s_1, s_2) + d(s_2, s_3)$.

This distance is an integer and is bounded by the sum of the two lengths of both sequences. Many people prefer to build a distance which stays in interval $[0, 1]$ and they usually choose a formula like this one :

$$d'(a, b) = \frac{d(a, b)}{\max(a, b)} \tag{4}$$

But this function (or the other one obtained by replace max by min) do not verify the triangular inequality. As examples, we can consider strings $s_1 = ab$, $s_2 = aba$, $s_3 = ba$. If max is replaced by min, we can choose $s_1 = a$, $s_2 = ab$, $s_3 = cb$. But there is another way to build such a normalized pseudo-distance :

$$d'(0, i) = \frac{i}{I} \ \forall i \leqslant I \tag{5}$$

$$d'(j, 0) = \frac{j}{J} \ \forall j \leqslant J \tag{6}$$

$$d'(i, j) = \min \begin{cases} d(i-1, j) + \frac{1}{I} & \text{if } i > 0 \\ d(i, j-1) + \frac{1}{J} & \text{if } j > 0 \\ d(i-1, j-1) + \frac{1}{2}\left(\frac{1}{I} + \frac{1}{J}\right) 1\!\!1_{\{a_i \neq b_j\}} & \text{if } i, j > 0 \end{cases} \tag{7}$$

If we assume $0 < I < J$, we can prove that :

$$d(a, b) \leqslant \frac{I}{2}\left(\frac{1}{I} + \frac{1}{J}\right) + (J - I)\frac{1}{J} \tag{8}$$

$$\leqslant \frac{1}{2} + \frac{I + 2(J - I)}{2J} = \frac{3}{2} - \frac{I}{2J} < \frac{3}{2} \tag{9}$$

But it is still not a distance, the same counter examples previously introduced prove it. It comes that fact that comparing two characters depend on the length of the sequence they belong too. To give a sense of how many time the inequality will not be fulfilled, we draw N times random strings. The length is randomly chosen between 1 and l, the characters are then randomly chosen among C characters. Next table gives the proportion of cases for which the inequality fails ($N = 100.000$) :

l	C	$d_{\min}$	$d_{\max}$	d'	d
3	3	5.16	0.03	2.68	0
3	5	3.35	0.00	1.33	0
2	3	2.48	0.00	2.48	0
2	5	8.54	0.02	1.71	0

$d_{\max}$ seems to be the best way to normalize among the three choices introduced here. I also considered to replicate the same experiment using words extracted from an English text (*The Tailor of Gloucester* by Beatrix Potter) which contains a little bit more than 900 distinct words.

text	distinct words	$d_{\min}$	$d_{\max}$	d'	d
The Tailor of Gloucester	915	2.10	0.00	0.00	0
The portrait of Dorian Gray	8867	1.25	0.00	0.00	0

But why should we consider to normalize the edit distance ? The first answer is obviously to get distance which can stay within some boundaries. For that purpose, the normalization using max function is probably the best one. The others reasons is to get distinct values. The edit distance returns integers. Knowing the longest word in a set of words, you can easily deduce what will all the possible values the distance can take. And for that purpose, the distance d' is probably the best one as shown Tables (1) and Figure 2.

	text	$d_{\min}$	$d_{\max}$	d'	d
Number of disinct values	The Tailor of Gloucester	165	73	702	26
	The portrait of Dorian Gray	128	77	721	21
	text	$d_{\min}$	$d_{\max}$	d'	d
Highest value	The Tailor of Gloucester	18.0	1.0	1.5	25.0
	The portrait of Dorian Gray	17.0	1.0	1.5	20.0

Table 1 : *Figures obtained for 10000 triplet of random words. Number of distinct values and highest values. Figure 2 shows the distribution of values.*

Correction

2 Code for the edit distance

```
#coding:latin-1
import copy, random
#from importme import *

def enumerate_none (p) :
    yield -1, None
    for i,x in enumerate(p) : yield i,x

def distance (p1, p2, noWeight) :

    dist = { (-1,-1) : (0,None,None) }

    w1 = 1.0 / len(p1) if not noWeight else 1.
```

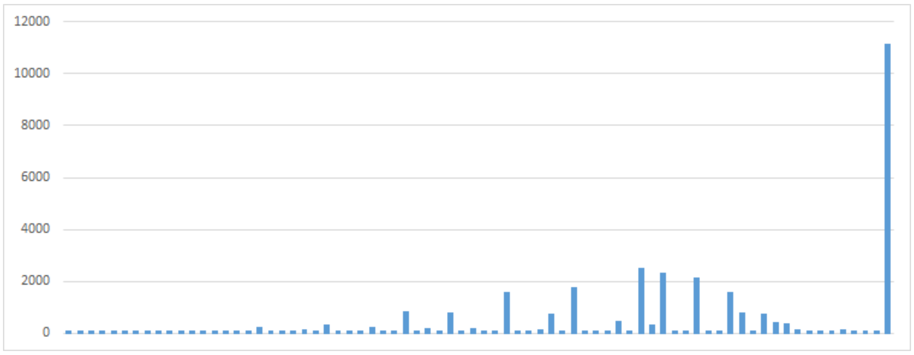
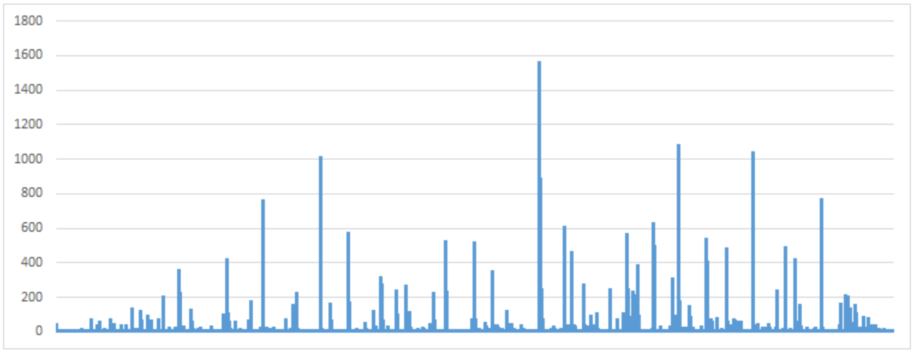
distance $d_{\max}$	
distance d'	

Table 2 : *Distribution of distinct values for distance distance $d_{\max}$ and d' measured on The Tailor of Gloucester.*

```
    w2 = 1.0 / len(p2) if not noWeight else 1.

    for i1,eorv1 in enumerate_none (p1) :
        for i2,eorv2 in enumerate_none (p2) :
            np = i1,i2
            posit = [ ((i1-1,i2),  (eorv1, None)),
                      ((i1,i2-1),  (None, eorv2)),
                      ((i1-1,i2-1),(eorv1,eorv2)), ]

            func = lambda x,y : \
                        0 if x == y else (\
                        0.5*(w1 + w2) if x != None and y != None \
                        else (w1 if y == None else w2))

            for p,co in posit :
                if p in dist :
                    c0 = dist [p][0]
                    c1 = func (co[0], co[1])
                    c  = c0 + c1
                    if np not in dist :
                        dist[np] = (c, p, co, (c0, c1))
                    elif c < dist[np][0] :
                        dist[np] = (c, p, co, (c0, c1))

    last = dist [len(p1)-1, len(p2)-1]
    path = [ ]
    while last[1] != None :
        path.append (last)
        last = dist [ last[1] ]
    path.reverse()

    d = dist [len(p1)-1, len(p2)-1][0]

    return d, path
```