Edit Distance and Levenshtein

Jari Koister

SLIDE 0: Intuition Levenshtein

In computer science, **edit distance** is a way of quantifying how dissimilar two strings are to one another by counting the minimum number of operations required to transform one string into the other.

sanfransicso-> san francisco

[insert] san fransicso-> [delete] san fransicso -> [insert] san francicso -> [delete] san francicso -> [insert] san francisco *{Distance = 5, not minimal}*

[insert] san fransicso-> [substitute] san francicso -> [substitute] san francisso -> [substitute] san francisco *{Distance = 4}*

SLIDE 1: Why Details About Edit Distance

Our explanation is based on a dynamic programming algorithm originating from Wagner-Fisher, 1974.

https://en.wikipedia.org/wiki/Wagner%E2%80%93Fischer_algorithm

Why interesting in a course about computation:

1. It is important to understand complexity in order to implement computation.
2.
3. A very practical easy to understand application.

SLIDE 2: Edit Distance w/ Dynamic Programming

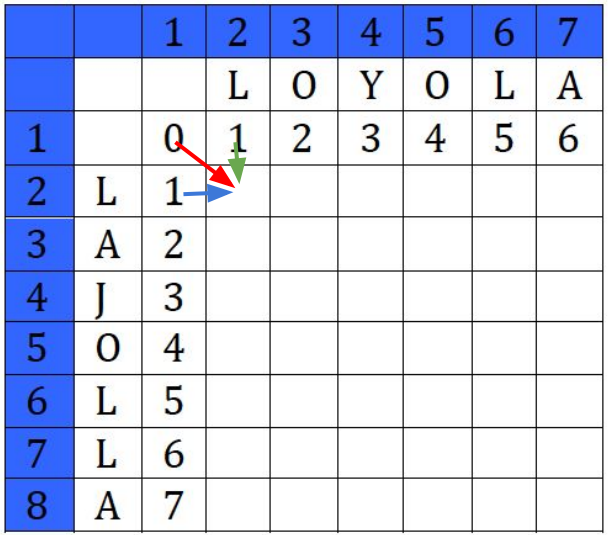
(Wagner-Fisher et al)

Denote the row by r and column by c.
We have n rows and m columns.
d[i,j] denotes the value on row i and columns j.

$cost\ [i,j]=1\ if\ c[i]\ !=r[j]$
$cost\ [i,j]=0\ if\ c[i]==r[j]$

d[i,j] is to be set to the minimum of:
d[i-1,j]+1 or
d[i,j-1]+1 or
d[i-1, j-1]+cost[i,j]

Distance is found in the resulting value d[n,m]



		1	2	3	4	5	6	7
			L	O	Y	O	L	A
1		0	1	2	3	4	5	6
2	L	1						
3	A	2						
4	J	3						
5	O	4						
6	L	5						
7	L	6						
8	A	7						

SLIDE 3: Column d[?,2]

```
d[2,2], cost is 0, minimum is d[1,1]+0=>0
d[3,2], cost is 1, minimum is d[2,2]+1=>1
d[4,2], cost is 1, minimum is d[3,2]+1=>2
d[5,2], cost is 1, minimum is d[4,2]+1=>3
d[6,2], cost is 0, minimum is d[5,1]+0=>4, or d[5,2]+1
d[7,2], cost is 0, minimum is d[6,2]+0=>5, or d[6,2]+1
d[8,2], cost is 1, minimum is d[7,2]+1=>6
```

		1	2	3	4	5	6	7
			L	O	Y	O	L	A
1		0	1	2	3	4	5	6
2	L	1	0					
3	A	2	1					
4	J	3	2					
5	O	4	3					
6	L	5	4					
7	L	6	5					
8	A	7	6					

SLIDE 4: Column d[?,3]

```
d[2,3], cost is 1, minimum is d[2,2]+1=>1
d[3,3], cost is 1, minimum is d[2,2]+1=>1
d[4,3], cost is 1, minimum is d[3,2]+1=>2, or d[3,3]+1
d[5,3], cost is 0, minimum is d[4,2]+0=>2
d[6,3], cost is 1, minimum is d[5,3]+1=>3
d[7,3], cost is 1, minimum is d[6,2]+1=>4, or d[6,3]+1
d[8,3], cost is 1, minimum is d[7,2]+1=>4
```

		1	2	3	4	5	6	7
			L	O	Y	O	L	A
1		0	1	2	3	4	5	6
2	L	1	0	1				
3	A	2	1	1				
4	J	3	2	2				
5	O	4	3	2				
6	L	5	3	3				
7	L	6	3	4				
8	A	7	4	4				

SLIDE 5: Final Result

		1	2	3	4	5	6	7
			L	O	Y	O	L	A
1		0	1	2	3	4	5	6
2	L	1	0	1	2	3	4	5
3	A	2	1	1	2	3	4	4
4	J	3	2	2	2	3	4	5
5	O	4	3	2	3	2	3	4
6	L	5	3	3	3	3	2	3
7	L	6	3	4	4	4	3	3
8	A	7	4	4	5	5	4	**3**

SLIDE 6: Complexity

For each i,j:

compare c[i] to r[j]

pick min of **[i-1,j]+1** or **d[i,j-1]+1** or **d[i-1, j-1]+cost[i,j]**

Complexity is *O*(*mn*)

Scale with Block Char: Perform a first pass over the sequence of strings to evaluate and obtain 'blocks' in which all strings share a substring of a given 'blocking size' (defaults to 6 chars in OpenRefine).