String Distance

Natural Language Processing

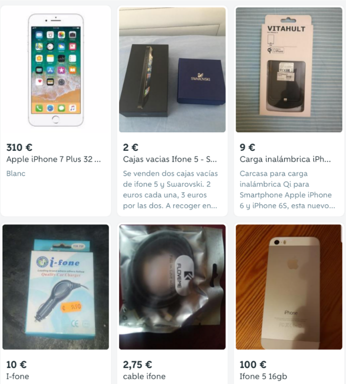
Daniel Ortiz Martínez, David Buchaca Prats

Problems Working with Strings

- Strings treated as elements within a vocabulary are problematic
- A string might be in the vocabulary, but the same string with a single character change might not
- Small changes in the input string might have dramatic consequences:
  - **motorbike** in vocabulary
  - **motorvike** not in vocabulary
- To deal with this type of issues many NLP applications process the data using edit distances

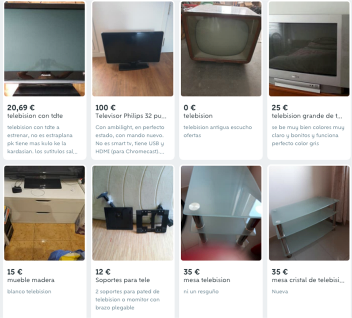
Example: “Ifone”

- Consider users writing “ifone”
- Users will get information from documents containing “ifone” but not “iphone” (unless both words appear in the text)

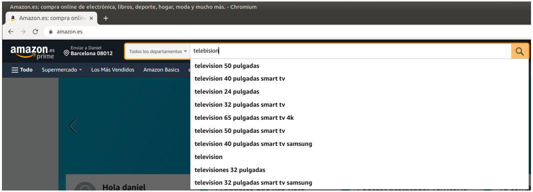


Example: “Telebision”

Without String Distances



With String Distances



String Distance Intuition

- String distances measure how different two strings are
  - If two strings are exactly the same, the string distance should be zero
  - If two strings differ from a single character the edit distance should be the distance caused by the different character
  - If two strings differ from two characters the edit distance should be the distance caused by the different characters
- One possibility is to obtain the minimum number of *edit operations*:
  - Insertion: insert a character in a given position
  - Deletion: delete a character in a given position
  - Substitution: substitute a character in a given position by another one

Jaccard Distance

- Jaccard similarity: $s_{\text{jaccard}}(x, y) = \frac{|x \cap y|}{|x \cup y|}$

```
def jaccard_similarity(s1, s2):
    return len(s1.intersection(s2)) / len(s1.union(s2))

jaccard_similarity(set("exponential"), set("exponentia"))
0.8888888888888888
```

- Jaccard distance: $d_{\text{jaccard}}(x, y) = 1 - \frac{|x \cap y|}{|x \cup y|}$

```
def jaccard_distance(s1, s2):
    return 1 - jaccard_similarity(s1, s2)

jaccard_distance(set("exponential"), set("polynomial"))
0.4545454545454546
```

Jaccard Distance

- The Jaccard distance does not fit very well with the sequential nature of strings

```
set1 = set('panmpi')
set2 = set('mapping')
jaccard_distance(set1, set2)
0.16666666666666663
```

```
set1 = set('mapping')
set2 = set('mappin')
jaccard_distance(set1, set2)
0.16666666666666663
```

Jaccard Distance

- Not taking into account order makes the jaccard distance less useful:

```
query = set('guardin')
words = words.words()
distances = compute_distances(query, words)
print("The closest word to query=", query, "is", words[np.argmin(distances)])

closest_words = [words[d] for d in np.argsort(distances)]
print(closest_words[0:10])
```

- Output:

```
The closest word to query= {'u', 'i', 'r', 'n', 'd', 'a', 'g'} is guardian
['unniggard', 'gurniad', 'guarding', 'undaring', 'guardian', 'indiguria', '
     ungrained', 'antidrug', 'unarraigned', 'underguardian']
```

Edit Distance: Overview

- Given the strings x and y, we want to compute their distance
- Edit distance finds the minimum number of edits to go from x to y
- The edit operations considered are:
  - Insertions
  - Substitutions
  - Deletions
- Edit distance is typically defined as a recurrence

Edit Distance

- The edit distance from $a = a_1 \ldots a_m$ to $b = b_1 \ldots b_n$ is given by d_{mn}, defined by the recurrence:

$$d_{i0} = \sum_{k=1}^{i} w_{\mathrm{del}}(b_k), \qquad \textit{for } 1 \leq i \leq m$$

$$d_{0j} = \sum_{k=1}^{j} w_{\mathrm{ins}}(a_k), \qquad \textit{for } 1 \leq j \leq n$$

$$d_{ij} = \begin{cases} d_{i-1,j-1} & \textit{for } a_j = b_i \\ min \begin{cases} d_{i-1,j} + w_{\mathrm{del}(b_i)} \\ d_{i,j-1} + w_{\mathrm{ins}(a_j)} \\ d_{i-1,j-1} + w_{\mathrm{sub}(a_j,b_i)} \end{cases} & \textit{for } a_j \neq b_i \end{cases} \qquad \textit{for } 1 \leq i \leq m, 1 \leq j \leq n$$

Edit Distance: Recursive Version

```
def recursive_edit_dist(x, y):
    if len(x) == 0:
        return len(y)
    if len(y) == 0:
        return len(x)

    delta = 0 if x[-1] == y[-1] else 1
    return min(recursive_edit_dist(x[:-1], y[:-1]) + delta,
               recursive_edit_dist(x[:-1], y) + 1,
               recursive_edit_dist(x, y[:-1]) + 1)
```

Edit Distance

- Recursive implementation is slow

```
n = 0
def recursive_edit_dist(x, y):
    global n
    n += 1
    if len(x) == 0:
        return len(y)
    if len(y) == 0:
        return len(x)

    delta = 0 if x[-1] == y[-1] else 1
    return min(recursive_edit_dist(x[:-1], y[:-1]) + delta,
               recursive_edit_dist(x[:-1], y) + 1,
               recursive_edit_dist(x, y[:-1]) + 1)

recursive_edit_dist("exponential", "polynomial")
print(n)
```

- Output

```
27711949
```

Edit Distance: Efficient Computation

- Efficient computations reuse pre-computed substring distances
  - We will compute $d(x, y) = d(x[: -1], y[: -1]) + \mathrm{cost}$
- We define $x[1 : i]$ as the first i characters from x
- We define $C[i, j]$ as the distance between $x[: i]$ and $y[: j]$
- We define C, a matrix of shape $(\mathrm{len}(x), \mathrm{len}(y))$ containing $C[i, j]$ at position i, j

Edit Distance: Efficient Computation

- Initialization
  - We start from an empty string '*'
  - Since $d(\text{'*'}, \text{'c'}) = 1$ for any character 'c' we know that the cost of going from an empty string to any character is 1

Edit Distance: Iterative Version

```
def iterative_edit_dist(s, t):
    rows = len(s)+1
    cols = len(t)+1
    dist = [[0 for x in range(cols)] for x in range(rows)]

    # source prefixes can be transformed into empty strings
    # by deletions:
    for i in range(1, rows):
        dist[i][0] = i

    # target prefixes can be created from an empty source string
    # by inserting the characters
    for i in range(1, cols):
        dist[0][i] = i

    for row in range(1, rows):
        for col in range(1, cols):
            if s[row-1] == t[col-1]:
                cost = 0
            else:
                cost = 1
            dist[row][col] = min(dist[row-1][col] + 1,      # deletion
                                 dist[row][col-1] + 1,      # insertion
                                 dist[row-1][col-1] + cost) # substitution

    return dist[row][col]
```

Edit Distance: Iterative Version

- What would be the computational complexity of the iterative version?
- And that of the recursive version?

Edit Distance Example: Initialization

- Given two strings s_1 and s_2
  - an empty array C is created
  - $C.shape = (len(s_1), len(s_2))$

	*	k	n	i	t	t	i	n	g
*									
k									
i									
t									
t									
e									
n									

Edit Distance Example: Initialization

- Given two strings s_1 and s_2
  - an empty array C is created
  - $C.shape = (len(s_1), len(s_2))$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1								
i	2								
t	3								
t	4								
e	5								
n	6								

Edit Distance Example: C Computation

- Let us compute $C[1,1]$

$$d_{ij} = \begin{cases} C(i-1, j-1) & \text{if } a_j = b_i \\ min(\text{ins}_{i,j}, \text{del}_{i,j}, \text{sub}_{i,j}) & \text{if } a_j \neq b_i \end{cases}$$

$$\begin{aligned} \text{del}_{1,1} &= C(0,1) + c_{\text{del}} && =?+? \\ \text{sub}_{1,1} &= C(0,0) + c_{\text{sub}} && =?+? \\ \text{ins}_{1,1} &= C(1,0) + c_{\text{ins}} && =?+? \end{aligned}$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	?							
i	2								
t	3								
t	4								
e	5								
n	6								

Edit Distance Example: C Computation

- Let us compute $C[1,1]$

$$d_{ij} = \begin{cases} C(i-1, j-1) & \text{if } a_j = b_i \\ min(\mathrm{ins}_{i,j}, \mathrm{del}_{i,j}, \mathrm{sub}_{i,j}) & \text{if } a_j \neq b_i \end{cases}$$

$$\begin{aligned} \mathrm{del}_{1,1} &= C(0,1) + c_{\mathrm{del}} &&= 1 + 1 \\ \mathrm{sub}_{1,1} &= C(0,0) + c_{\mathrm{sub}} &&= 0 + 0 \\ \mathrm{ins}_{1,1} &= C(1,0) + c_{\mathrm{ins}} &&= 1 + 1 \end{aligned}$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0							
i	2								
t	3								
t	4								
e	5								
n	6								

Edit Distance Example: C Computation

- Let us compute $C[1,2]$

$$d_{ij} = \begin{cases} C(i-1, j-1) & \text{if } a_j = b_i \\ min(\mathrm{ins}_{i,j}, \mathrm{del}_{i,j}, \mathrm{sub}_{i,j}) & \text{if } a_j \neq b_i \end{cases}$$

$$\begin{aligned} \mathrm{del}_{1,2} &= C(0,2) + c_{\mathrm{del}} & &= ? + ? \\ \mathrm{sub}_{1,2} &= C(0,1) + c_{\mathrm{sub}} & &= ? + ? \\ \mathrm{ins}_{1,2} &= C(1,1) + c_{\mathrm{ins}} & &= ? + ? \end{aligned}$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	?						
i	2								
t	3								
t	4								
e	5								
n	6								

Edit Distance Example: C Computation

- Let us compute $C[1,2]$

$$d_{ij} = \begin{cases} C(i-1, j-1) & \text{if } a_j = b_i \\ min(\text{ins}_{i,j}, \text{del}_{i,j}, \text{sub}_{i,j}) & \text{if } a_j \neq b_i \end{cases}$$

$$\begin{aligned} \text{del}_{1,2} &= C(0,2) + c_{\text{del}} &&= 2+1 \\ \text{sub}_{1,2} &= C(0,1) + c_{\text{sub}} &&= 1+1 \\ \text{ins}_{1,2} &= C(1,1) + c_{\text{ins}} &&= 0+1 \end{aligned}$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1						
i	2								
t	3								
t	4								
e	5								
n	6								

Edit Distance Example: C Computation

- Let us compute $C[2, 1]$

$$d_{ij} = \begin{cases} C(i-1, j-1) & \text{if } a_j = b_i \\ min(\mathrm{ins}_{i,j}, \mathrm{del}_{i,j}, \mathrm{sub}_{i,j}) & \text{if } a_j \neq b_i \end{cases}$$

$$\mathrm{del}_{2,1} = C(1,1) + c_{\mathrm{del}} \qquad = ? + ?$$

$$\mathrm{sub}_{2,1} = C(1,0) + c_{\mathrm{sub}} \qquad = ? + ?$$

$$\mathrm{ins}_{2,1} = C(2,0) + c_{\mathrm{ins}} \qquad = ? + ?$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1	2	3	4	5	6	7
i	2	?							
t	3								
t	4								
e	5								
n	6								

Edit Distance Example: C Computation

- Let us compute $C[2,1]$

$$d_{ij} = \begin{cases} C(i-1, j-1) & \text{if } a_j = b_i \\ min(\mathrm{ins}_{i,j}, \mathrm{del}_{i,j}, \mathrm{sub}_{i,j}) & \text{if } a_j \neq b_i \end{cases}$$

$$\begin{aligned} \mathrm{del}_{2,1} &= C(1,1) + c_{\mathrm{del}} &&= 0 + 1 \\ \mathrm{sub}_{2,1} &= C(1,0) + c_{\mathrm{sub}} &&= 1 + 1 \\ \mathrm{ins}_{2,1} &= C(2,0) + c_{\mathrm{ins}} &&= 2 + 1 \end{aligned}$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1	2	3	4	5	6	7
i	2	1							
t	3								
t	4								
e	5								
n	6								

Edit Distance Example: C Computation

- Let us compute $C[2,2]$

$$d_{ij} = \begin{cases} C(i-1, j-1) & \text{if } a_j = b_i \\ min(\text{ins}_{i,j}, \text{del}_{i,j}, \text{sub}_{i,j}) & \text{if } a_j \neq b_i \end{cases}$$

$$\begin{aligned} \text{del}_{2,2} &= C(1,2) + c_{\text{del}} &&=?+? \\ \text{sub}_{2,2} &= C(1,1) + c_{\text{sub}} &&=?+? \\ \text{ins}_{2,2} &= C(2,1) + c_{\text{ins}} &&=?+? \end{aligned}$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1	2	3	4	5	6	7
i	2	1	?						
t	3								
t	4								
e	5								
n	6								

Edit Distance Example: C Computation

- Let us compute $C[2,2]$

$$d_{ij} = \begin{cases} C(i-1, j-1) & \text{if } a_j = b_i \\ min(\text{ins}_{i,j}, \text{del}_{i,j}, \text{sub}_{i,j}) & \text{if } a_j \neq b_i \end{cases}$$

$$\begin{aligned} \text{del}_{2,2} &= C(1,2) + c_{\text{del}} &&= 1+1 \\ \text{sub}_{2,2} &= C(1,1) + c_{\text{sub}} &&= 0+1 \\ \text{ins}_{2,2} &= C(2,1) + c_{\text{ins}} &&= 1+1 \end{aligned}$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1	2	3	4	5	6	7
i	2	1	1						
t	3								
t	4								
e	5								
n	6								

Edit Distance Example: C Computation

- Let us compute $C[2,3]$

$$d_{ij} = \begin{cases} C(i-1, j-1) & \text{if } a_j = b_i \\ min(\mathrm{ins}_{i,j}, \mathrm{del}_{i,j}, \mathrm{sub}_{i,j}) & \text{if } a_j \neq b_i \end{cases}$$

$$\begin{aligned} \mathrm{del}_{2,3} &= C(1,3) + c_{\mathrm{del}} & &= ? + ? \\ \mathrm{sub}_{2,3} &= C(1,2) + c_{\mathrm{sub}} & &= ? + ? \\ \mathrm{ins}_{2,3} &= C(2,2) + c_{\mathrm{ins}} & &= ? + ? \end{aligned}$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1	2	3	4	5	6	7
i	2	1	1	?					
t	3								
t	4								
e	5								
n	6								

Edit Distance Example: C Computation

- Let us compute $C[2,3]$

$$d_{ij} = \begin{cases} C(i-1, j-1) & \text{if } a_j = b_i \\ min(\mathrm{ins}_{i,j}, \mathrm{del}_{i,j}, \mathrm{sub}_{i,j}) & \text{if } a_j \neq b_i \end{cases}$$

$$\begin{aligned} \mathrm{del}_{2,3} &= C(1,3) + c_{\mathrm{del}} &&= 2+1 \\ \mathrm{sub}_{2,3} &= C(1,2) + c_{\mathrm{sub}} &&= 1+0 \\ \mathrm{ins}_{2,3} &= C(2,2) + c_{\mathrm{ins}} &&= 1+1 \end{aligned}$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1	2	3	4	5	6	7
i	2	1	1	1					
t	3								
t	4								
e	5								
n	6								

Edit Distance Example: C Computation

- Let us compute the rest of the table

$$d_{ij} = \begin{cases} C(i-1, j-1) & \text{if } a_j = b_i \\ min(\text{ins}_{i,j}, \text{del}_{i,j}, \text{sub}_{i,j}) & \text{if } a_j \neq b_i \end{cases}$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1	2	3	4	5	6	7
i	2	1	1	1	2	3	4	5	6
t	3	2	2	2	1	2	3	4	5
t	4	3	3	3	2	1	2	3	4
e	5	4	4	4	3	2	2	3	4
n	6	5	4	5	4	3	3	2	3

Edit Distance Example: C Computation

- $d(\text{'knitting'}, \text{'kitten'}) = C[-1][-1] = 3$

$$d_{ij} = \begin{cases} C(i-1, j-1) & \text{if } a_j = b_i \\ min(\text{ins}_{i,j}, \text{del}_{i,j}, \text{sub}_{i,j}) & \text{if } a_j \neq b_i \end{cases}$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1	2	3	4	5	6	7
i	2	1	1	1	2	3	4	5	6
t	3	2	2	2	1	2	3	4	5
t	4	3	3	3	2	1	2	3	4
e	5	4	4	4	3	2	2	3	4
n	6	5	4	5	4	3	3	2	3

Obtaining the String Distance Table

```python
def iterative_edit_dist(s, t):
    rows = len(s)+1
    cols = len(t)+1
    dist = [[0 for x in range(cols)] for x in range(rows)]

    # source prefixes can be transformed into empty strings
    # by deletions:
    for i in range(1, rows):
        dist[i][0] = i

    # target prefixes can be created from an empty source string
    # by inserting the characters
    for i in range(1, cols):
        dist[0][i] = i

    for row in range(1, rows):
        for col in range(1, cols):
            if s[row-1] == t[col-1]:
                cost = 0
            else:
                cost = 1
            dist[row][col] = min(dist[row-1][col] + 1,      # deletion
                                 dist[row][col-1] + 1,      # insertion
                                 dist[row-1][col-1] + cost) # substitution

    return dist
```

Obtaining the String Distance Table

- Usage example:

```
dist = iterative_edit_dist("exponential", "polynomial")
for row in dist:
    print(row)
```

- Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[2, 2, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[3, 2, 3, 3, 4, 5, 6, 7, 8, 9, 10]
[4, 3, 2, 3, 4, 5, 5, 6, 7, 8, 9]
[5, 4, 3, 3, 4, 4, 5, 6, 7, 8, 9]
[6, 5, 4, 4, 4, 5, 5, 6, 7, 8, 9]
[7, 6, 5, 5, 5, 4, 5, 6, 7, 8, 9]
[8, 7, 6, 6, 6, 5, 5, 6, 7, 8, 9]
[9, 8, 7, 7, 7, 6, 6, 6, 6, 7, 8]
[10, 9, 8, 8, 8, 7, 7, 7, 7, 6, 7]
[11, 10, 9, 8, 9, 8, 8, 8, 8, 7, 6]
```

Python is Slow

- The previous implementation, even if choosing a fast algorithm, is quite slow
- The majority of standard Python packages (Pandas, Scikit-learn, Numpy) are built upon Cython
- Cython is a programming language (superset of Python) to compile python code with the goal of improving speed

```
def fib(n):
    a = 0.
    b = 0.
    for i in range(n):
        a, b = a + b, a
    return a
```

```
%load_ext cython
%%cython --annotate
def cy_fib(int n):
    cdef int i
    cdef double a=0.0, b=1.0
    for i in range(n):
        a, b = a + b, a
    return a
```

Writing Python Code in Cython can Provide Good Speedups

```
import timeit

n_times = 10000000
tfib = timeit.timeit("fib(10)", setup="from __main__ import fib", number=n_times)
tfib_unit = tfib/n_times

tcyfib = timeit.timeit("cy_fib(10)", setup="from __main__ import cy_fib", number=n_times)
tcyfib_unit = tcyfib/n_times

print("Python implementation took:", tfib, ", time per call:", tfib_unit)
print("Cython implementation took:", tcyfib, ", time per call:", tcyfib_unit)
print("Cython speedup:", tfib/tcyfib)
```

```
Python implementation took: 4.510144141000183 , time per call: 4.510144141000827e-07
Cython implementation took: 0.2599191509998491 , time per call: 2.599191509998491e-08
Cython speedup: 17.35210400484419
```

Cythonizing a Method

```
%%cython --annotate
import numpy as np

def cy_iterative_edit_dist(s, t):
    cdef int rows = len(s)+1
    cdef int cols = len(t)+1
    cdef int [:, :] dist = np.zeros((rows, cols), dtype=np.int32)
    cdef int i

    # source prefixes can be transformed into empty strings
    # by deletions:
    for i in range(1, rows):
        dist[i][0] = i

    # target prefixes can be created from an empty source string
    # by inserting the characters
    for i in range(1, cols):
        dist[0][i] = i

    cdef int row
    cdef int col
    for row in range(1, rows):
        for col in range(1, cols):
            if s[row-1] == t[col-1]:
                cost = 0
            else:
                cost = 1
            dist[row][col] = min(dist[row-1][col] + 1,      # deletion
                                 dist[row][col-1] + 1,      # insertion
                                 dist[row-1][col-1] + cost) # substitution

    return dist
```

Time Cost Experiments

```python
n_times = 100000
ted = timeit.timeit("iterative_edit_dist('exponential', 'polynomial')", setup="from
      __main__ import iterative_edit_dist", number=n_times)
ted_unit = ted/n_times

tcyed = timeit.timeit("cy_iterative_edit_dist('exponential', 'polynomial')", setup="from
      __main__ import cy_iterative_edit_dist", number=n_times)
tcyed_unit = tcyed/n_times

print("Python implementation took:", ted, ", time per call:", ted_unit)
print("Cython implementation took:", tcyed, ", time per call:", tcyed_unit)
print("Cython speedup:", ted/tcyed)
```

```
Python implementation took: 4.63264147100017 , time per call: 4.63264147100017e-05
Cython implementation took: 1.5121306539995203 , time per call: 1.5121306539995203e-05
Cython speedup: 3.0636515824522395
```

Do we Need the Full Matrix to Get the Distance?

```
%%cython --annotate
def cy_iterative_edit_dist(s, t):
    cdef int rows = len(s)+1
    cdef int cols = len(t)+1
    cdef int [:, :] dist = np.zeros((rows, cols), dtype=np.int32)
    cdef int i

    # source prefixes can be transformed into empty strings
    # by deletions:
    for i in range(1, rows):
        dist[i][0] = i

    # target prefixes can be created from an empty source string
    # by inserting the characters
    for i in range(1, cols):
        dist[0][i] = i

    cdef int row
    cdef int col
    for row in range(1, rows):
        for col in range(1, cols):
            if s[row-1] == t[col-1]:
                cost = 0
            else:
                cost = 1
            dist[row][col] = min(dist[row-1][col] + 1,      # deletion
                                 dist[row][col-1] + 1,      # insertion
                                 dist[row-1][col-1] + cost) # substitution
    return dist[(rows-1), cols-1]
```

- To obtain the values of each new matrix row, we only pay attention to the immediately previous row
- Only two rows need to be stored at any given time

Using Only Two Rows for the Table

```
%%cython --annotate
def cy_iterative_edit_dist_2rows(s, t):
    cdef int rows = len(s)+1
    cdef int cols = len(t)+1
    cdef int [:, :] dist = np.zeros([2, cols], dtype=np.int32)
    cdef int i

    for i in range(1, cols):
        dist[0][i] = i

    cdef int row
    cdef int col
    for row in range(1, rows):
        dist[row%2][0] = row
        for col in range(1, cols):
            if s[row-1] == t[col-1]:
                cost = 0
            else:
                cost = 1
            dist[row%2][col] = min(dist[(row-1)%2][col] + 1,       # deletion
                                   dist[row%2][col-1] + 1,         # insertion
                                   dist[(row-1)%2][col-1] + cost) # substitution
    return dist[(rows-1)%2, cols-1]
```