STRING DISTANCES AND APPLICATIONS

Introduction to NLP
Session 4
2023

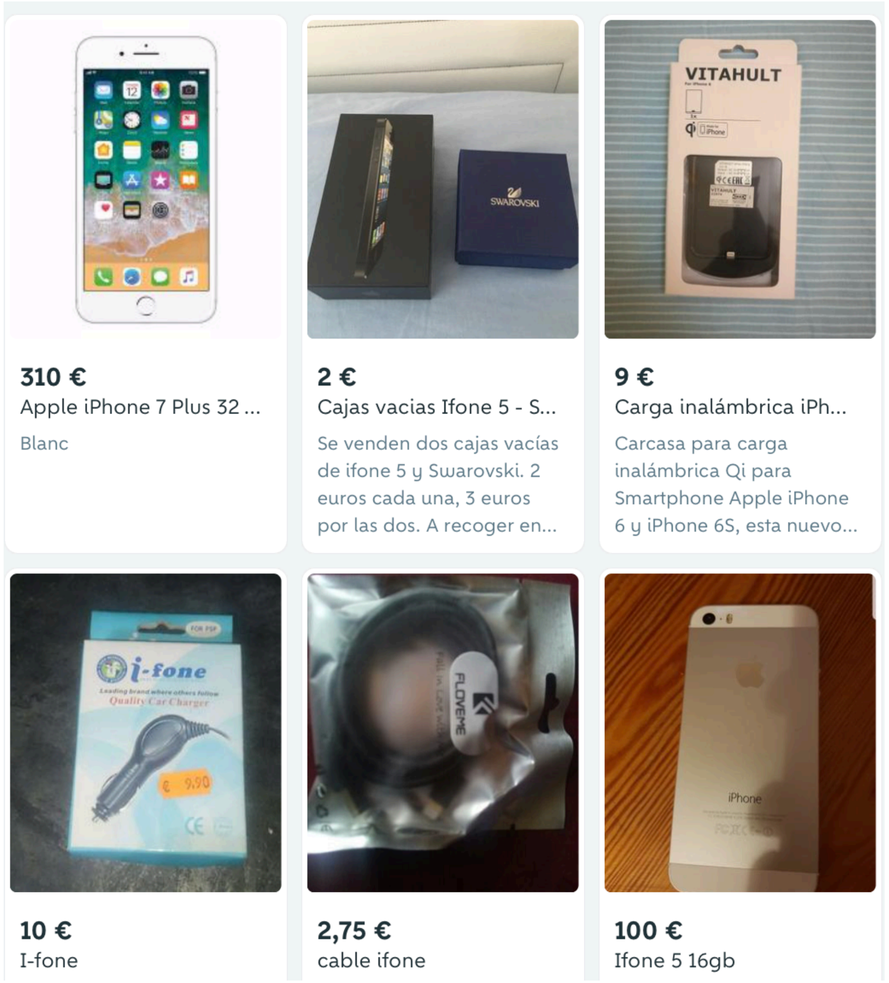
David Buchaca Prats

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 be in the Vocabulary.
  - 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.

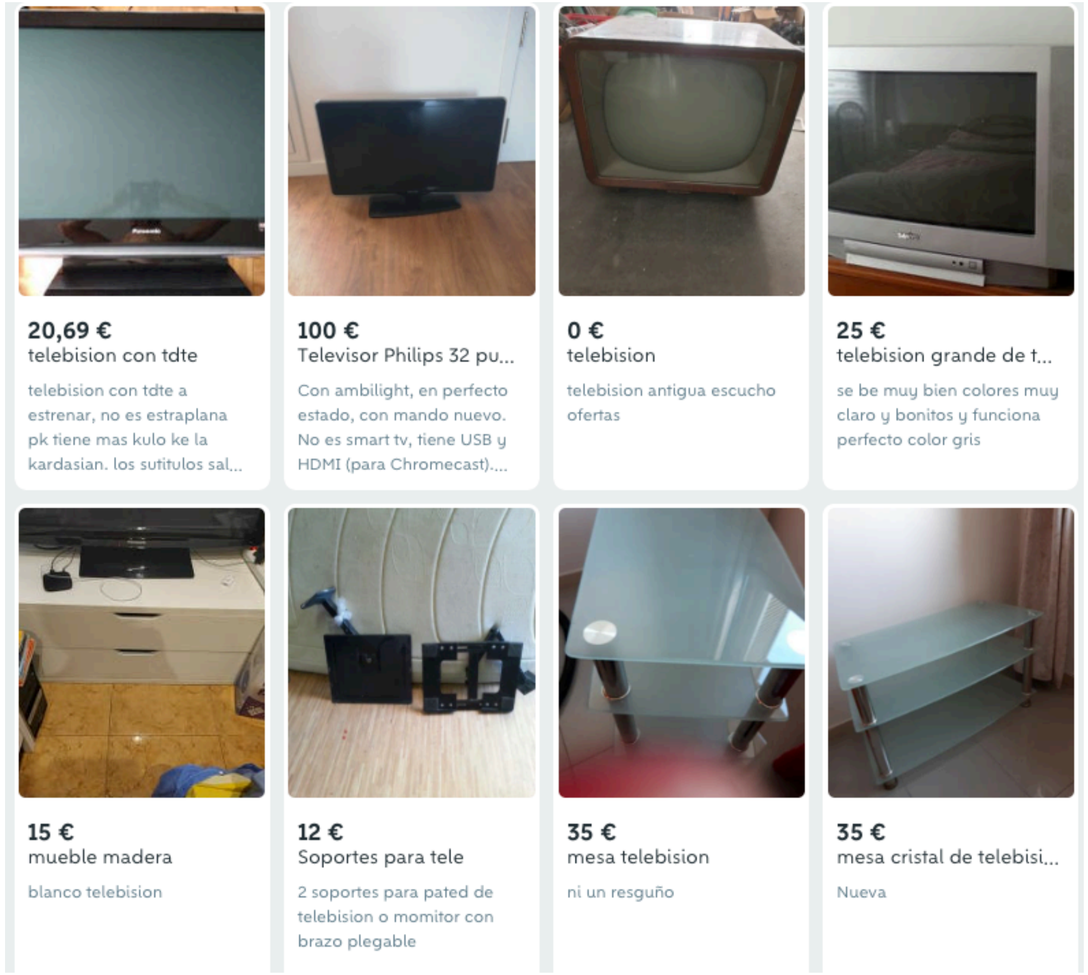
EXAMPLE: ‘IFONE’

- Consider users writting **‘ifone’**
- Users will get information from documents containing **‘ifone’** but not **‘iphone’** (unless both **‘ifone’** an **‘iphone’** appear in the text).



EXAMPLE: ‘TELEBISION’

No string distances used



String distances are used

String distances allow searching for similar strings

STRING DISTANCE INTUITION

- The objective of a string distance is to measure how different two strings are.
  - If two strings are exactly the same the edit distance has to be zero.
  - If two strings differ from a single character the edit distance has to be the distance provided by the different character.
  - If two strings differ from two characters the edit distance has to be the distance provided by the different characters.
- In other words, we want to know the minimum number of edit operations between two strings, where operations are:
  - Insert: Add a character in a given position.
  - Delete: Delete a character in a given position.
  - Substitute: Substitute a character in a given position by another character.

JACCARD DISTANCE

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

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

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

executed in 3ms, finished 16:36:48 2020-03-01

```
0.8888888888888888
```

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

```
jaccard_similarity(set("exponential"),set("polynomial"))
```

executed in 2ms, finished 16:36:30 2020-03-01

```
0.5454545454545454
```

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

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)
```

executed in 3ms, finished 16:27:53 2021-03-07

```
0.1666666666666663
```

panmpi is similar to **mapping** ?

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

executed in 3ms, finished 16:27:57 2021-03-07

```
0.1666666666666663
```

mapping is similar to **mappin** ?

JACCARD DISTANCE

- Not taking into account order in strings makes the jaccard distance misleading:

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

executed in 219ms, finished 16:17:26 2021-03-08

```
the closest word to query={'u', 'r', 'a', 'd', 'i', 'n', 'g'} is guardian
```

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

executed in 66ms, finished 16:17:27 2021-03-08

```
['unniggard',
 'gurniad',
 'guarding',
 'undaring',
 'guardian',
 'indiguria',
 'ungrained',
 'antidrug',
 'unarraigned',
 'underguardian']
```

STANDARD EDIT DISTANCE IMPLEMENTATION (WIKIPEDIA)

Computation [edit]

The first algorithm for computing minimum edit distance between a pair of strings was published by Damerau in 1964.[6]

Common algorithm [edit]

Main article: Wagner–Fischer algorithm

Using Levenshtein's original operations, the edit distance between $a = a_1 \dots a_n$ and $b = b_1 \dots b_m$ is given by d_{mn}, defined by the recurrence[2]

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

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

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

This algorithm can be generalized to handle transpositions by adding another term in the recursive clause's minimization.[3]

SIMPLE CASE: CONSIDER ALL OPERATIONS HAVE THE SAME COST

```
def edit_distance_recursive(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(edit_distance_recursive(x[:-1],y[:-1]) + delta,
               edit_distance_recursive(x[:-1],y)      + 1,
               edit_distance_recursive(x,y[:-1])      + 1)
```

RECURSIVE EDIT DISTANCE

- The previous recursive implementation is correct but slow

```
n = 0
def edit_distance_recursive(x,y):
    global n
    if len(x) ==0:
        return len(y)
    if len(y) == 0:
        return len(x)

    if x =="super" and y=="sup":
        n += 1

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

```

```
edit_distance_recursive("superman", "supermaniac")
n
```

```
833
```

OVERVIEW OF THE LEVENSHTEIN DISTANCE

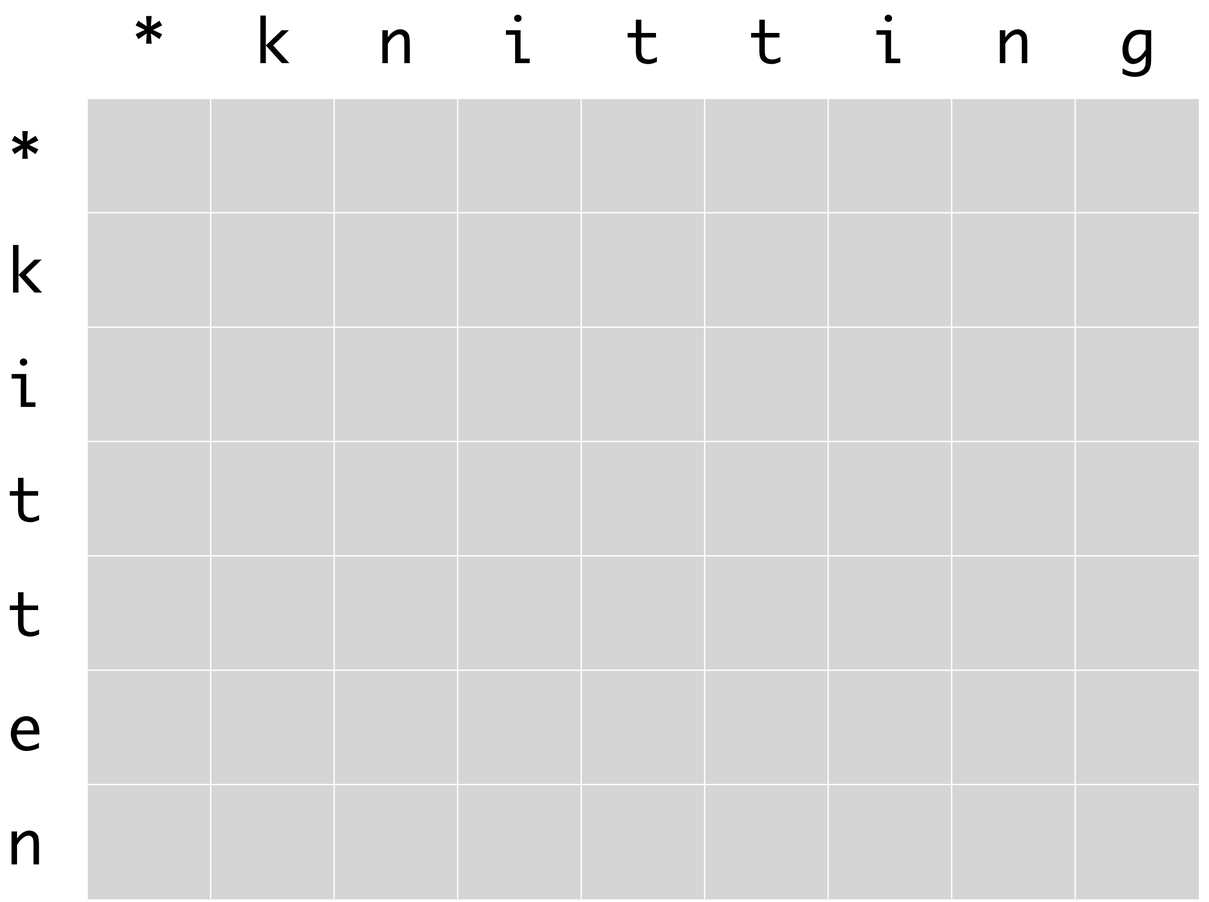
- Given **x**, **y** strings we want to compute **d(x,y)**.
- This edit distance consist on finding the cost associated to the minimum number of edits needed to go from **x** to **y**
- In order to make the computation efficiently we will reuse pre-computed substring distances.
  - We will compute **d(x,y) = d(x[:-1], y[:-1]) + something**
- We define **x[1:i]** as the first i characters from **x**
- We define **D[i,j]** as the distance between **x[:i]** and **y[:j]**
- We define **D** a matrix of shape **(len(x), len(y))** containing **D[i,j]** at position **i,j**

OVERVIEW OF THE LEVENSHTEIN DISTANCE

- Initialization
  - We start from an empty string **‘*’**
  - Since **d(‘*’, ‘c’) = 1** for any character **‘c’** we know that the cost of going from an empty string to any character is 1.

EXAMPLE: INITIALIZATION

- Given two strings s1 and s2
  - We start creating an empty array X
  - X.shape = (len(s1), len(s2))



EXAMPLE: INITIALIZATION

- Given two strings s1 and s2
  - We start creating an empty array X
  - X.shape = (len(s1), len(s2))

	*	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								

EXAMPLE: C COMPUTATION

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

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{1,1} = C(0,1) + c_{del} = ? + 1$$

$$sub_{1,1} = C(0,0) + c_{sub} = ? + 1$$

$$ins_{1,1} = C(1,0) + c_{ins} = ? + 1$$

	*	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								

EXAMPLE: C COMPUTATION

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

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{1,1} = C(0,1) + c_{del} = ! + 1$$

$$sub_{1,1} = C(0,0) + c_{sub} = ! + 1$$

$$ins_{1,1} = C(1,0) + c_{ins} = ! + 1$$

	*	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								

EXAMPLE: C COMPUTATION

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

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{1,2} = C(0,2) + c_{del} = ? + 1$$

$$sub_{1,2} = C(0,1) + c_{sub} = ? + 1$$

$$ins_{1,2} = C(1,1) + c_{ins} = ? + 1$$

	*	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								

EXAMPLE: C COMPUTATION

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

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{1,2} = C(0,2) + c_{del} = 2 + 1$$

$$sub_{1,2} = C(0,1) + c_{sub} = 1 + 1$$

$$ins_{1,2} = C(1,1) + c_{ins} = 0 + 1$$

	*	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								

EXAMPLE: C COMPUTATION

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

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{2,1} = C(1,1) + c_{del} = ? + 1$$
$$sub_{2,1} = C(1,0) + c_{sub} = ? + 1$$
$$ins_{2,1} = C(2,0) + c_{ins} = ? + 1$$

	*	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								

EXAMPLE: C COMPUTATION

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

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{2,1} = C(1,1) + c_{del} = 0 + 1$$
$$sub_{2,1} = C(1,0) + c_{sub} = 1 + 1$$
$$ins_{2,1} = C(2,0) + c_{ins} = 2 + 1$$

	*	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								

EXAMPLE: C COMPUTATION

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

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{2,2} = C(1,2) + c_{del} = ? + 1$$
$$sub_{2,2} = C(1,1) + c_{sub} = ? + 1$$
$$ins_{2,2} = C(2,1) + c_{ins} = ? + 1$$

	*	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								

EXAMPLE: C COMPUTATION

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

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{2,2} = C(1,2) + c_{del} = 1 + 1$$
$$sub_{2,2} = C(1,1) + c_{sub} = 0 + 1$$
$$ins_{2,2} = C(2,1) + c_{ins} = 1 + 1$$

	*	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								

EXAMPLE: C COMPUTATION

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

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{2,3} = C(1,3) + c_{del} = ? + 1$$
$$sub_{2,3} = C(1,2) + c_{sub} = ? + 1$$
$$ins_{2,3} = C(2,2) + c_{ins} = ? + 1$$

	*	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								

EXAMPLE: C COMPUTATION

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

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{2,3} = C(1,3) + c_{del} = ! + 1$$

$$sub_{2,3} = C(1,2) + c_{sub} = ! + 1$$

$$ins_{2,3} = C(2,2) + c_{ins} = ! + 1$$

	*	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								

EXAMPLE: C COMPUTATION

- Let us compute the rest of the table….

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

	*	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

EXAMPLE: C COMPUTATION

- The result of **d('knitting','kitten') = C[-1,-1] = 3**

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

	*	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

PYTHON IMPLEMENTATION OF STRING DISTANCE TABLE

```
def create_memoization_table(X,Y):

    len_x = len(X)
    len_y = len(Y)
    D = np.zeros((len_x+1,len_y+1))

    for i in range(len(X)+1):
        for j in range(len(Y)+1):

            if i == 0:
                D[i][j] = j

            elif j == 0:
                D[i][j] = i

            elif X[i-1] == Y[j-1]:
                D[i][j] = D[i-1][j-1]

            else:
                D[i][j] = 1+min(D[i][j-1],      # Insert
                                D[i-1][j],      # Remove
                                D[i-1][j-1])    # Replace
    return D
```

```
x = "EXPONENTIAL"
y = "POLYNOMIAL"
D = create_memoization_table(x,y)
memoization_table(x,y,D)
D[-1,-1]
```

	empty	P	O	L	Y	N	O	M	I	A	L
empty	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0
E	1.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0
X	2.0	2.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0
P	3.0	2.0	3.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0
O	4.0	3.0	2.0	3.0	4.0	5.0	5.0	6.0	7.0	8.0	9.0
N	5.0	4.0	3.0	3.0	4.0	4.0	5.0	6.0	7.0	8.0	9.0
E	6.0	5.0	4.0	4.0	4.0	5.0	5.0	6.0	7.0	8.0	9.0
N	7.0	6.0	5.0	5.0	5.0	4.0	5.0	6.0	7.0	8.0	9.0
T	8.0	7.0	6.0	6.0	6.0	5.0	5.0	6.0	7.0	8.0	9.0
I	9.0	8.0	7.0	7.0	7.0	6.0	6.0	6.0	6.0	7.0	8.0
A	10.0	9.0	8.0	8.0	8.0	7.0	7.0	7.0	7.0	6.0	7.0
L	11.0	10.0	9.0	8.0	9.0	8.0	8.0	8.0	8.0	7.0	6.0

PYTHON IS SLOW

- The previous implementation, even if choosing a fast algorithm, is quite slow.
- Almost all "famous" Python packages (Pandas, Scikit-learn, Numpy) are build upon Cython.
  - Cython is a programming language (superset of Python) that can compile any python code with the goal of improving python speed while keeping Python like syntax.

```
def fib(n):
    a = 0.
    b = 1.
    for i in range(n):
        a, b = a + b, a
    return a
```

```
%load_ext cython
```

```
#%%cython --a
cpdef 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
```

WRITTING PYTHON CODE IN CYTHON CAN PROVIDE GOOD SPEEDUPS

```
import timeit
```

```
n_times = 100000
t_fib = timeit.timeit("fib(10)", setup="from __main__ import fib",number=n_times)
t_cyfib = timeit.timeit("cy_fib(10)", setup="from __main__ import cy_fib",number=n_times)
t_fib_unit = t_fib/n_times

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

print(" Python version took: {} sec\n Cython version took: {} sec\n Cython is {:.0f}x faster"\
      .format(t_fib, t_cyfib, t_fib/t_cyfib))

print("\n Python version 1 run took: {} sec\n Cython version took: {} sec\n Cython is {:.0f}x faster"\
      .format(t_fib_unit, t_cyfib_unit, t_fib_unit/t_cyfib_unit))
```

```
 Python version took: 0.06224320799998395 sec
 Cython version took: 0.0037031669999976202 sec
 Cython is 17x faster

 Python version 1 run took: 6.224320799998396e-07 sec
 Cython version took: 3.70316699999762e-08 sec
 Cython is 17x faster
```

EXAMPLE: CYTHONIZING A METHOD

```python
def create_memoization_table(X,Y):

    len_x = len(X)
    len_y = len(Y)
    D = np.zeros((len_x+1,len_y+1))

    for i in range(len(X)+1):
        for j in range(len(Y)+1):

            if i == 0:
                D[i][j] = j

            elif j == 0:
                D[i][j] = i

            elif X[i-1] == Y[j-1]:
                D[i][j] = D[i-1][j-1]

            else:
                D[i][j] = 1+min(D[i][j-1],      # Insert
                                D[i-1][j],      # Remove
                                D[i-1][j-1])    # Replace
    return D
```

```
%%cython --a

cimport numpy as np
import numpy as np
cimport cython

@cython.boundscheck(False) # turn off bounds-checking for entire function
@cython.wraparound(False)  # turn off negative index wrapping for entire function
cpdef cy_create_memoization_table(str X, str Y):

    cdef int i, j, del_char, ins_char, sub_char, Z
    cdef int len_x = len(X)
    cdef int len_y = len(Y)
    cdef int [:, :] D =  np.zeros((len_x + 1, len_y + 1), dtype=np.int32)

    for i in range(len_x+1):
        D[i,0] = i

    for j in range(len_y+1):
        D[0,j] = j

    for i in range(1, len_x + 1):
        for j in range(1, len_y + 1):
            del_char = D[i-1,j] + 1
            ins_char = D[i,j-1] + 1

            if X[i-1] == Y[j-1]:
                Z = 0
            else:
                Z = 1
            sub_char = D[i-1,j-1] + Z

            D[i,j] = min(del_char, ins_char, sub_char)

    return D
```

CUSTOM BUILD EDIT DISTANCE WITH CYTHON

- In the exercise notebook for day 3 you are asked to write a fast version with cython that should provide results similar to:

```
Python version took: 1.3459972919999927 sec
Cython version took: 0.006109708999929353 sec
nltk   version took: 0.4242933330000369 sec
Cython is 220x faster than python
Cython is 69x faster than nltk
```

NOTE: DO WE NEED THE FULL MATRIX TO GET THE DISTANCE?

- Note that the matrix D is build using
  - For a given (i,j) information from
    - D[i-1,j-1], D[i,j-1], D[i-1,j-1]
- To fill in a row we only use the previous row. Therefore, we might only need two rows.

```
def create_memoization_table(X,Y):

    len_x = len(X)
    len_y = len(Y)
    D = np.zeros((len_x+1,len_y+1))

    for i in range(len(X)+1):
        for j in range(len(Y)+1):

            if i == 0:
                D[i][j] = j

            elif j == 0:
                D[i][j] = i

            elif X[i-1] == Y[j-1]:
                D[i][j] = D[i-1][j-1]

            else:
                D[i][j] = 1+min(D[i][j-1],      # Insert
                                D[i-1][j],      # Remove
                                D[i-1][j-1])    # Replace
    return D
```

OPTIMIZATION TRICKS: USING ONLY TWO ROWS FOR THE TABLE

- We can have a single 2 row array that stores the partial computations which are updated at every step.
- The i%2 operation allows us to iteratively select row 0/row 1/row 0 /row 1 and update the table.

```
x = "EXPONENTIAL"
y = "POLYNOMIAL"
D = create_memoization_table(x,y)
memoization_table(x,y,D)
D[-1,-1]
```

	empty	P	O	L	Y	N	O	M	I	A	L
empty	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0
E	1.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0
X	2.0	2.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0
P	3.0	2.0	3.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0
O	4.0	3.0	2.0	3.0	4.0	5.0	5.0	6.0	7.0	8.0	9.0
N	5.0	4.0	3.0	3.0	4.0	4.0	5.0	6.0	7.0	8.0	9.0
E	6.0	5.0	4.0	4.0	4.0	5.0	5.0	6.0	7.0	8.0	9.0
N	7.0	6.0	5.0	5.0	5.0	4.0	5.0	6.0	7.0	8.0	9.0
T	8.0	7.0	6.0	6.0	6.0	5.0	5.0	6.0	7.0	8.0	9.0
I	9.0	8.0	7.0	7.0	7.0	6.0	6.0	6.0	6.0	7.0	8.0
A	10.0	9.0	8.0	8.0	8.0	7.0	7.0	7.0	7.0	6.0	7.0
L	11.0	10.0	9.0	8.0	9.0	8.0	8.0	8.0	8.0	7.0	6.0

```
def __init__(self, int  c_ins=1,int c_del=1, int c_rep=1):
    self.c_ins = c_ins
    self.c_del = c_del
    self.c_rep = c_rep


cpdef int evaluate(self,str str1, str str2):
    cdef int m = len(str1)
    cdef int n = len(str2)
    cdef int i,j,del_char, ins_char, rep_char

    cdef np.ndarray[int,ndim=2] dist = np.zeros([2,m+1], dtype=np.int32)

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

    for i in range(1,n+1):
        for j in range(m+1):
            if j==0:
                dist[i%2,j] = i

            elif str1[j-1] == str2[i-1]:
                dist[i%2,j] = dist[(i-1)%2,j-1]

            else:
                del_char = dist[(i-1)%2,j] + self.c_del #Delete
                ins_char = dist[i%2,j-1] + self.c_ins #Insert
                rep_char = dist[(i-1)%2,j-1] + self.c_rep #Replace

                dist[i%2,j] = min(del_char, ins_char, rep_char)

    return dist[n%2,m]
```