

INTRODUCTION TO NLP

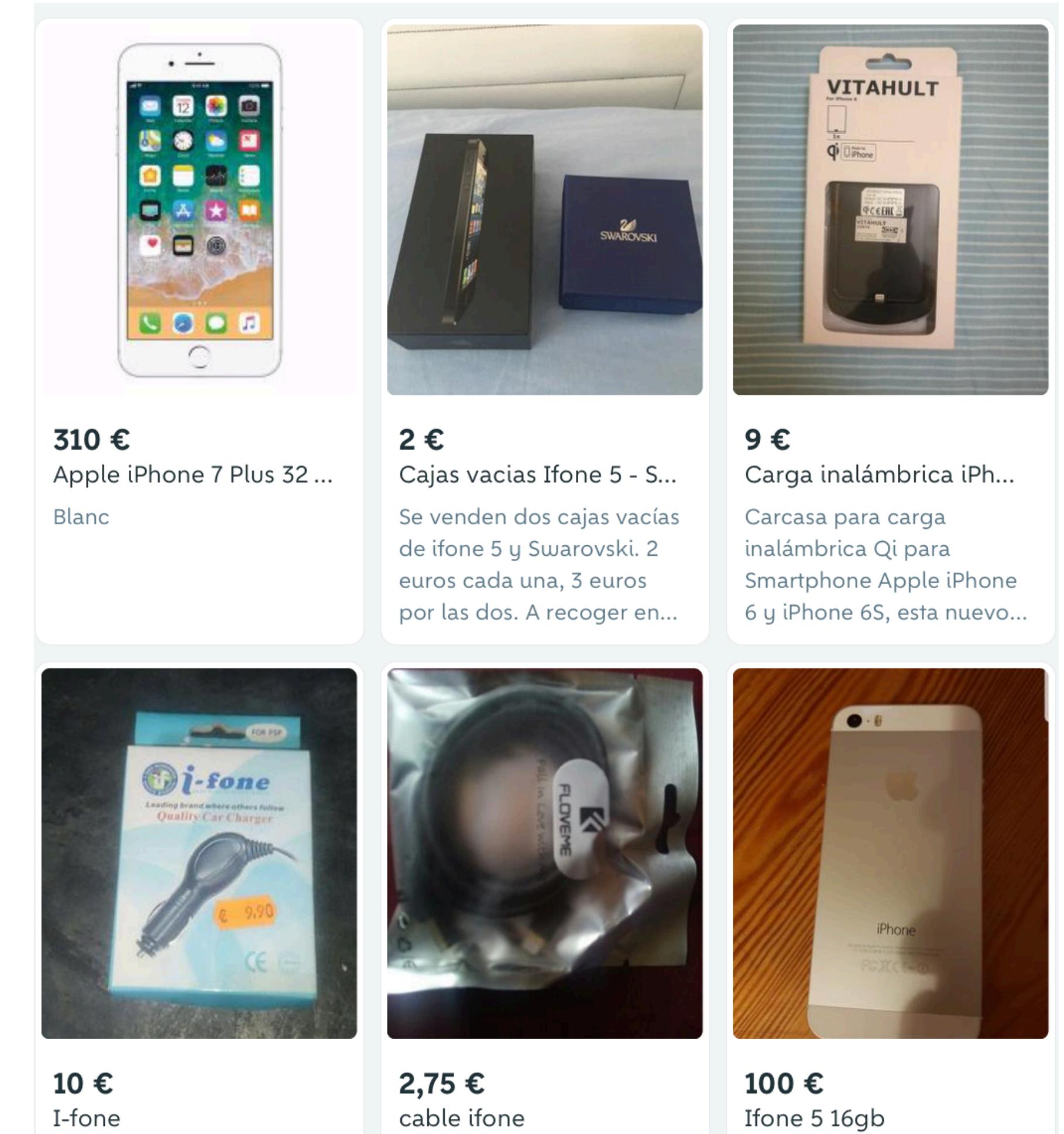
*Session 3
David Buchaca Prats
2022*

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.

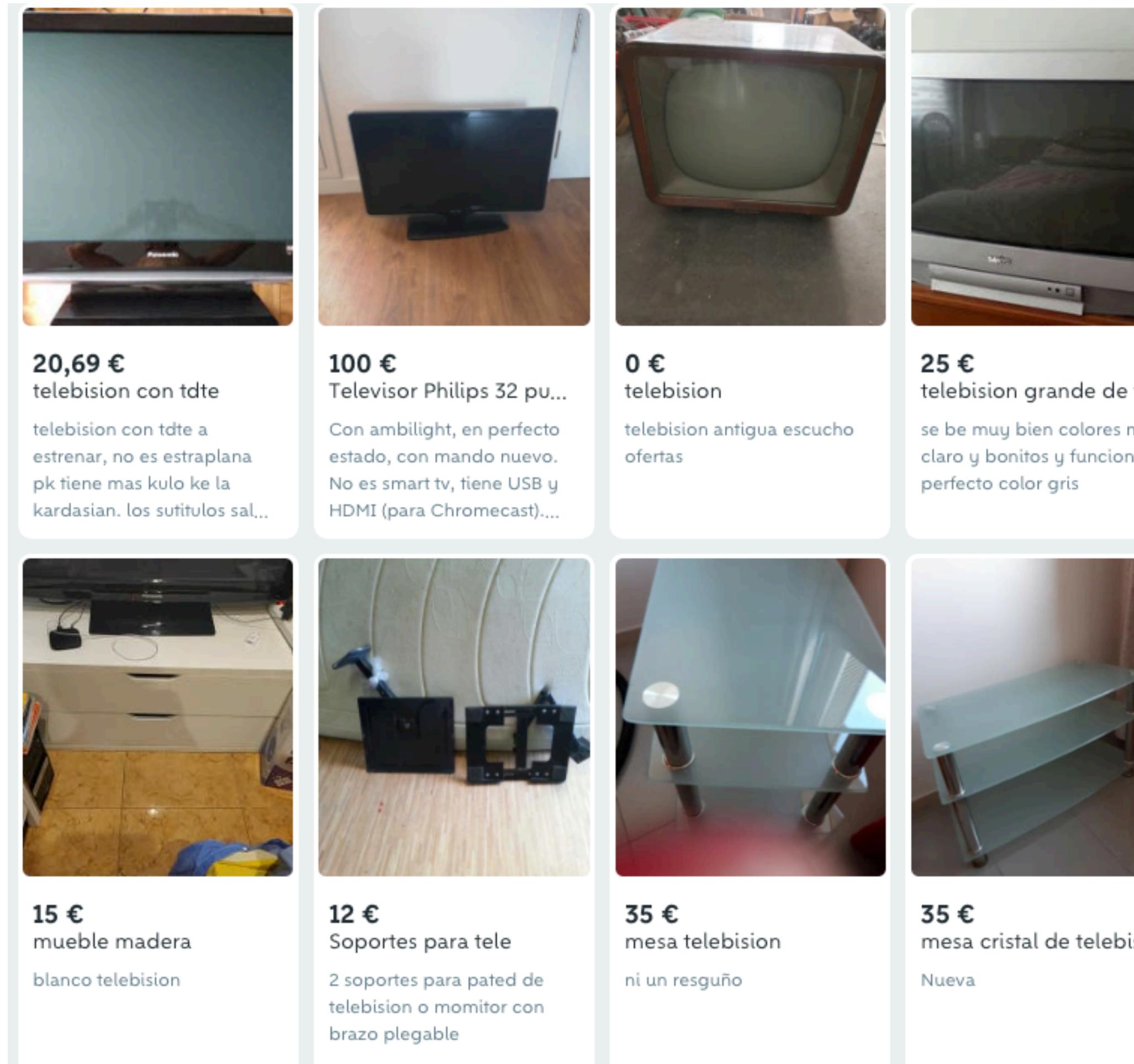
EXAMPLE: 'IFONE'

- Consider users writing 'ifone'
- Users will get information from documents containing 'ifone' but not 'iphone' (unless both 'ifone' and 'iphone' appear in the text).



EXAMPLE: 'TELEVISION'

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

0.888888888888888

```
▼ 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']
```

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

```
: 1  n = 0
2  def edit_distance_recursive(x,y):
3      global n
4      if len(x) == 0:
5          return len(y)
6      if len(y) == 0:
7          return len(x)
8
9      if x == "super" and y == "sup":
10         n += 1
11
12     delta = 0 if x[-1] == y[-1] else 1
13     return min(edit_distance_recursive(x[:-1],y[:-1]) + delta,
14                edit_distance_recursive(x[:-1],y) + 1,
15                edit_distance_recursive(x,y[:-1]) + 1)
16
```

```
: 1  edit_distance_recursive("superman", "supermaniac")
2  n
```

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]

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

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

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

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

OVERVIEW OF THE LEVENSHTEIN DISTANCE

- Given \mathbf{x} , \mathbf{y} strings we want to compute $\mathbf{d}(\mathbf{x}, \mathbf{y})$.
- This edit distance consist on finding the cost associated to the minimum number of edits needed to go from \mathbf{x} to \mathbf{y}
- In order to make the computation efficiently we will reuse pre-computed substring distances.
 - We will compute $\mathbf{d}(\mathbf{x}, \mathbf{y}) = \mathbf{d}(\mathbf{x}[:-1], \mathbf{y}[:-1]) + \text{something}$
 - We define $\mathbf{x}[1:i]$ as the first i characters from \mathbf{x}
 - We define $\mathbf{D}[i,j]$ as the distance between $\mathbf{x}[:i]$ and $\mathbf{y}[:j]$
 - We define \mathbf{D} a matrix of shape $(\mathbf{len}(\mathbf{x}), \mathbf{len}(\mathbf{y}))$ containing $\mathbf{D}[i,j]$ 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 s_1 and s_2
 - We start creating an empty array X
 - $X.shape = (\text{len}(s_1), \text{len}(s_2))$

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

EXAMPLE: INITIALIZATION

- Given two strings s_1 and s_2
 - We start creating an empty array X
 - $X.shape = (\text{len}(s_1), \text{len}(s_2))$

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

EXAMPLE: C COMPUTATION

- Let us compute $\mathbf{C}[1,1]$

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

$$\text{del}_{1,1} = C(0,1) + c_{\text{del}} = ? + 1$$

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

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

*	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			

The table shows the state of the computation grid. The first row contains labels: *, k, n, i, t, t, i, n, g. The second row contains indices: 0, 1, 2, 3, 4, 5, 6, 7, 8. The third row contains values: 1, followed by empty cells. The fourth row contains value 2, followed by empty cells. The fifth row contains value 3, followed by empty cells. The sixth row contains value 4, followed by empty cells. The seventh row contains value 5, followed by empty cells. The eighth row contains value 6, followed by empty cells.

EXAMPLE: C COMPUTATION

- Let us compute $\mathbf{C}[1,1]$

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

$$\text{del}_{1,1} = C(0,1) + c_{\text{del}} = ! + 1$$

$$\text{sub}_{1,1} = C(0,0) + c_{\text{sub}} = ! + 1$$

$$\text{ins}_{1,1} = C(1,0) + c_{\text{ins}} = ! + 1$$

*	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 $\mathbf{C}[1,2]$

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

$$\text{del}_{1,2} = C(0,2) + c_{\text{del}} = ? + 1$$

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

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

*	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								

The diagram shows a grid for dynamic programming. The columns are labeled with symbols: *, k, n, i, t, t, i, n, g. The rows are labeled with numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8. A cell at row 1, column 1 (labeled 'k') contains the value 0. An arrow points from this cell to the cell at row 2, column 1 (labeled 'i'), which is also shaded dark gray.

EXAMPLE: C COMPUTATION

- Let us compute $\mathbf{C}[1,2]$

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

$$\text{del}_{1,2} = C(0,2) + c_{\text{del}} = 2 + 1$$

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

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

*	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 $\mathbf{C}[2,1]$

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

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

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

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

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

A 9x9 grid representing a dynamic programming table for sequence comparison. The columns are labeled with symbols * (0), k (1), n (2), i (3), t (4), t (5), i (6), n (7), and g (8). The rows are labeled with symbols * (0), k (1), i (2), t (3), t (4), e (5), and n (6). The cell at row 1, column 1 (k=1, i=2) contains the value 0, which is highlighted with a dark gray background. An arrow points from the label 'k' to this cell.

EXAMPLE: C COMPUTATION

- Let us compute $\mathbf{C}[2,1]$

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

$$\text{del}_{2,1} = C(1,1) + c_{\text{del}} = 0 + 1$$

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

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

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

A 9x9 grid representing a dynamic programming table for sequence comparison. The columns are labeled with symbols * (0), k (1), n (2), i (3), t (4), t (5), i (6), n (7), and g (8). The rows are labeled with symbols *, k (1), i (2), t (3), t (4), e (5), and n (6). The cell at row 2, column 1 (k=1) contains the value 0, which is highlighted with a dark gray background. An arrow points from the label 'i' to the row index 2, and another arrow points from the label 'k' to the column index 1.

EXAMPLE: C COMPUTATION

- Let us compute $\mathbf{C}[2,2]$

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

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

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

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

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

EXAMPLE: C COMPUTATION

- Let us compute $\mathbf{C}[2,2]$

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

$$\text{del}_{2,2} = C(1,2) + c_{\text{del}} = 1 + 1$$

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

$$\text{ins}_{2,2} = C(2,1) + c_{\text{ins}} = 1 + 1$$

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

A 9x9 grid for dynamic programming. The columns are labeled with symbols: *, k, n, i, t, t, i, n, g. The rows are labeled with numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8. The cell at (k, 1) contains 0, which is highlighted with a dark gray background. An arrow points from the value 1 in row 2, column 1 to the value 0 in row 1, column 2.

EXAMPLE: C COMPUTATION

- Let us compute $\mathbf{C}[2,3]$

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

$$\text{del}_{2,3} = C(1,3) + c_{\text{del}} = ? + 1$$

$$\text{sub}_{2,3} = C(1,2) + c_{\text{sub}} = ? + 1$$

$$\text{ins}_{2,3} = C(2,2) + c_{\text{ins}} = ? + 1$$

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

The diagram shows a grid for dynamic programming. The columns are labeled with symbols: *, k, n, i, t, t, i, n, g. The rows are labeled with numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8. A cell at row 1, column 1 (labeled *) contains a value of 0. A cell at row 1, column 2 (labeled k) contains a value of 1. A cell at row 1, column 3 (labeled n) contains a value of 2. A cell at row 1, column 4 (labeled i) contains a value of 3. A cell at row 1, column 5 (labeled t) contains a value of 4. A cell at row 1, column 6 (labeled t) contains a value of 5. A cell at row 1, column 7 (labeled i) contains a value of 6. A cell at row 1, column 8 (labeled n) contains a value of 7. A cell at row 1, column 9 (labeled g) contains a value of 8. Arrows point from the labels to their corresponding grid positions. An arrow points from the 'i' label to the 'i' column. Another arrow points from the '2' in the 'i' row to the '2' in the 'i' column.

EXAMPLE: C COMPUTATION

- Let us compute $\mathbf{C}[2,3]$

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

$$\text{del}_{2,3} = C(1,3) + c_{\text{del}} = ! + 1$$

$$\text{sub}_{2,3} = C(1,2) + c_{\text{sub}} = ! + 1$$

$$\text{ins}_{2,3} = C(2,2) + c_{\text{ins}} = ! + 1$$

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

The table shows the computation of the edit distance matrix. The columns are labeled with k (0 to 6), n (0 to 7), i (0 to 2), t (0 to 4), and g (0 to 8). The rows are labeled with * (0 to 6), k (1 to 6), n (0 to 7), i (0 to 2), t (0 to 4), and g (0 to 8). The matrix values are as follows:

- Row 0 (k=1): [0, 1, 2, 3, 4, 5, 6, 7, 8]
- Row 1 (k=2): [1, 0, 1, 2, 3, 4, 5, 6, 7]
- Row 2 (k=3): [2, 1, 1 → 1, null, null, null, null, null, null]
- Row 3 (k=4): [3, null, null, null, null, null, null, null, null]
- Row 4 (k=5): [4, null, null, null, null, null, null, null, null]
- Row 5 (k=6): [5, null, null, null, null, null, null, null, null]
- Row 6 (k=7): [6, null, null, null, null, null, null, null, null]

An arrow points from the value 2 in the (k=2, i=1) cell to the value 1 in the (k=3, i=1) cell, indicating the transition from insertion to match.

EXAMPLE: C COMPUTATION

- Let us compute the rest of the table....

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

	*	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

EXAMPLE: C COMPUTATION

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

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

*	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