Levenshtein distance

Abstract

Clustering algorithms rely on the concept of edit distance to compare strings. Edit distance quantifies the number of insertions, deletions, and substitutions required to transform one sequence into another. The most widely used variant is the Levenshtein distance, which plays a significant role in applications such as spell checking, speech recognition, detecting dialect variations or plagiarism, and molecular biology. The following topics are discussed: i) Introduction ii) Levenshtein distance, including its algorithm and examples iii) Python implementations, both manual and library-based iv) Optimization techniques, including space optimization, bit-parallelism, and upper bound optimization v) Damerau-Levenshtein distance vi) Hamming distance vii) Jaro-Winkler distance viii) Application areas

1. Introduction

First, in 1965, V. I. Levenshtein published 'Binary Codes Capable of Correcting Deletions, Insertions, and Reversals,' building on an earlier paper by R. W. Hamming that stemmed from research in information transmission and coding theory. Consequently, the Levenshtein distance is named after the Soviet scientist Vladimir Levenshtein, who was of Jewish-Russian origin.

Picture 1. *Vladimir Levenshtein*

Vladimir Iosifovich Levenshtein(20 May 1935 – 6 September 2017) was a Russian scientist who did research in information theory, error-correcting codes, and combinatorial design. In addition, he is known for the Levenshtein distance and a Levenshtein algorithm. He received the IEEE Richard W. Hamming Medal in 2006, for "contributions to the theory of error-correcting codes and information theory, including the Levenshtein distance"

1.1 Levenshtein Distance

In information theory, linguistics, and computer science, the Levenshtein distance (LD) is a string metric used to measure the difference between two sequences: the source (s) and the target (t). Thus, the Levenshtein distance (LD) between two strings or words is defined as the minimum number of changes required to transform one into the other. These changes are categorized into three types: 1) insertion (adding a character), 2) deletion (removing a character), and 3) substitution (replacing a character). Essentially, a greater Levenshtein distance indicates that the strings or words are more dissimilar. In this context, the source represents the input, while the target is one of the entries.

Let's exemplify it,

The Levenshtein distance between **"Bayoh" and "Boyhe"** is 3;

- By substituting "o" for "a"
- Adding "e"
- Removing "o"

As a result, 1 substitution + 1 insertion + 1 deletion = 3 changes.

1.2 The LD Algorithm

1) As a 1st step, we set m to be the length of target, and n to be the length of source. Then, we construct a matrix with m rows and n columns, initializing the first row and the first column to with values from 0 to m, and from 0 to n, respectively.

2) As a 2nd step, we will analyze both source and target using indexes:
    a. Source, i from 1 to n.
    b. Target, j from 1 to m.

   Then, there is a condition, if s[i] equals to t[j], the cost is 0. Otherwise, the cost is 0. Moreover, set cell k[i, j] of the matrix equal to the minimum of:

   a) The cell immediately above plus 1: *k[i-1,j] + 1*.

   b) The cell immediately to the left plus 1: *d[i,j-1] + 1.*

   c) The cell diagonally above and to the left plus the cost: *d[i-1,j-1] + cost.*

Eventually, this process is repeated until the k[m, n] value is found.

1.2.1 Example 1:

Finding Levenshtein distance between “Bayoh” and “Boyhe” manually:

1st Step: iteration = 0

		B	A	Y	O	H
	0	1	2	3	4	5
B	1					
O	2					
Y	3					
H	4					
E	5					

2nd Step: iteration = 1

		B	A	Y	O	H
	0	1	2	3	4	5
B	1	0	1	2	3	4
O	2	1				
Y	3	2				
H	4	3				
E	5	4				

3rd Step: iteration = 2

		B	A	Y	O	H
	0	1	2	3	4	5
B	1	0	1	2	3	4
O	2	1	1	2	2	3
Y	3	2	2			
H	4	3	3			
E	5	4	4			

4th Step: iteration = 3

		B	A	Y	O	H
	0	1	2	3	4	5
B	1	0	1	2	3	4
O	2	1	1	2	2	3
Y	3	2	2	1	2	3
H	4	3	3	2		
E	5	4	4	3		

5th Step: iteration = 4

		B	A	Y	O	H
	0	1	2	3	4	5
B	1	0	1	2	3	4
O	2	1	1	2	2	3
Y	3	2	2	1	2	3
H	4	3	3	2	2	2
E	5	4	4	3	3	

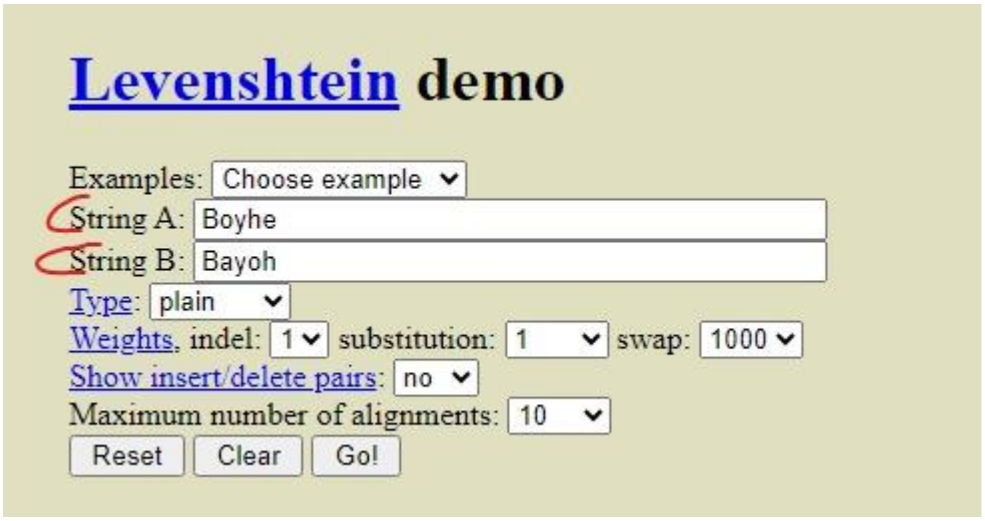
Final Step, iteration -> {i = n, j = m}

		B	A	Y	O	H
	0	1	2	3	4	5
B	1	0	1	2	3	4
O	2	1	1	2	2	3
Y	3	2	2	1	2	3
H	4	3	3	2	2	2
E	5	4	4	3	3	3

Finally, as we can see, result is 3. On the other hand, we can use calculator for finding it.

1.2.2 Example 2:

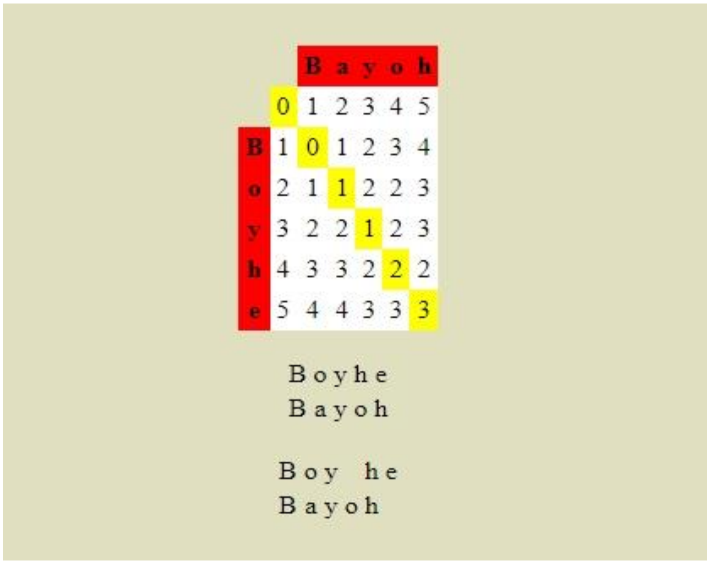
Finding Levenshtein distance between “Bayoh” and “Boyhe” using calculator:



Levenshtein demo

Examples: Choose example
String A: Boyhe
String B: Bayoh
Type: plain
Weights, indel: 1 substitution: 1 swap: 1000
Show insert/delete pairs: no
Maximum number of alignments: 10

Picture 2*. Interface of calculator*



		B	a	y	o	h
	0	1	2	3	4	5
B	1	0	1	2	3	4
o	2	1	1	2	2	3
y	3	2	2	1	2	3
h	4	3	3	2	2	2
e	5	4	4	3	3	3

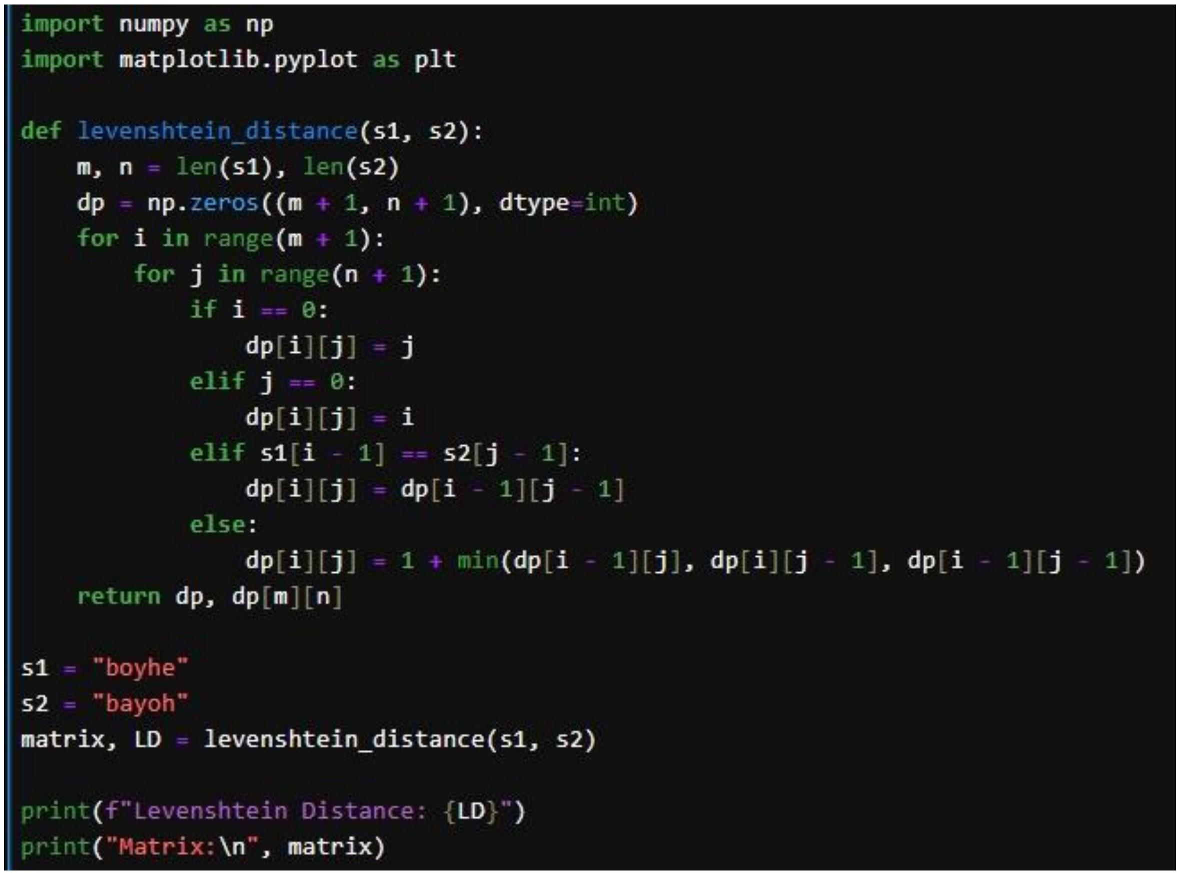
```
B o y h e
B a y o h

B o y   h e
B a y o h
```

Picture 3. *Output (result is 3 too)*

1.2.3 Example 3:

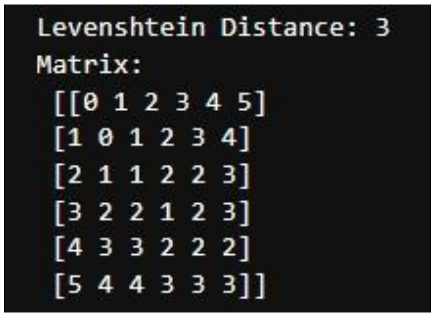
Finding Levenshtein distance between “Bayoh” and “Boyhe” in Python with manual and library-based implementation:

a. *Manual implementation:*



```python
import numpy as np
import matplotlib.pyplot as plt

def levenshtein_distance(s1, s2):
    m, n = len(s1), len(s2)
    dp = np.zeros((m + 1, n + 1), dtype=int)
    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0:
                dp[i][j] = j
            elif j == 0:
                dp[i][j] = i
            elif s1[i - 1] == s2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1])
    return dp, dp[m][n]

s1 = "boyhe"
s2 = "bayoh"
matrix, LD = levenshtein_distance(s1, s2)

print(f"Levenshtein Distance: {LD}")
print("Matrix:\n", matrix)
```

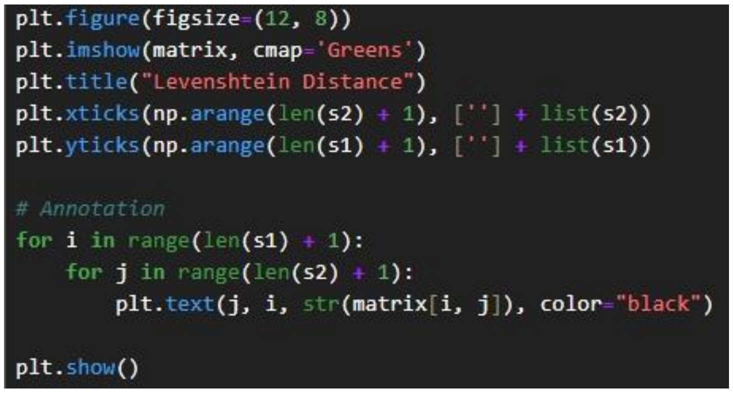
Picture 4. *Manual implementation in python*



```
Levenshtein Distance: 3
Matrix:
 [[0 1 2 3 4 5]
 [1 0 1 2 3 4]
 [2 1 1 2 2 3]
 [3 2 2 1 2 3]
 [4 3 3 2 2 2]
 [5 4 4 3 3 3]]
```

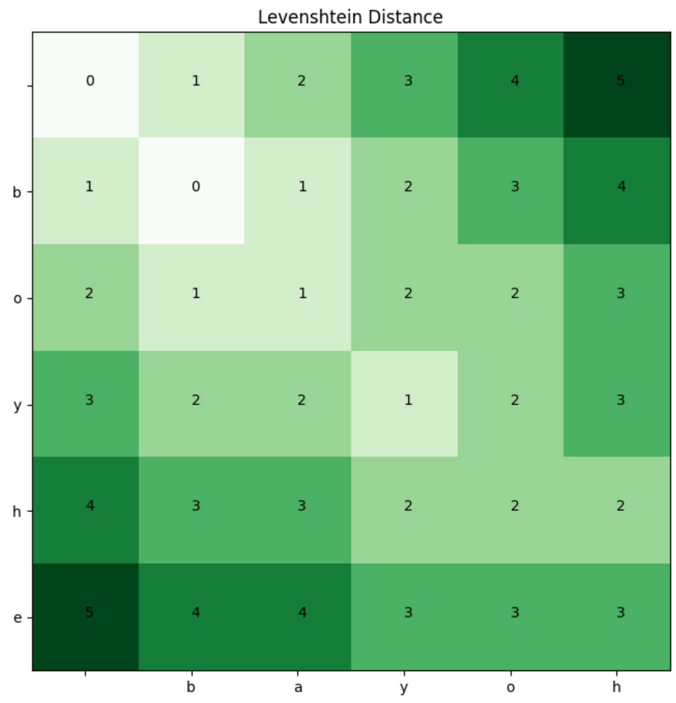
Picture 5. *Output of *picture 4**

Visualization of manual implementation



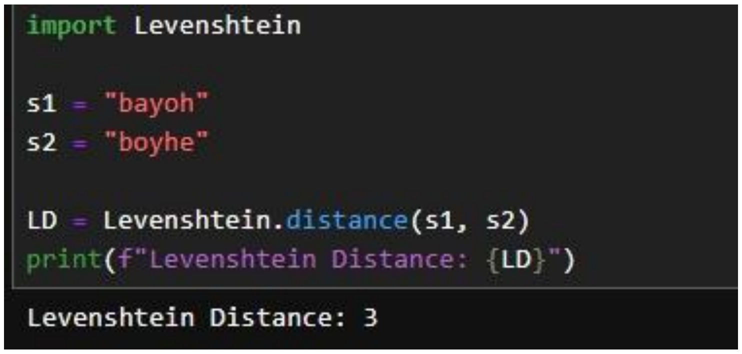
```
plt.figure(figsize=(12, 8))
plt.imshow(matrix, cmap='Greens')
plt.title("Levenshtein Distance")
plt.xticks(np.arange(len(s2) + 1), [''] + list(s2))
plt.yticks(np.arange(len(s1) + 1), [''] + list(s1))

# Annotation
for i in range(len(s1) + 1):
    for j in range(len(s2) + 1):
        plt.text(j, i, str(matrix[i, j]), color="black")

plt.show()
```

Picture 6. *Visualization of manual implementation in python with code and output*



		b	a	y	o	h
	0	1	2	3	4	5
b	1	0	1	2	3	4
o	2	1	1	2	2	3
y	3	2	2	1	2	3
h	4	3	3	2	2	2
e	5	4	4	3	3	3

b. Library-based implementation:



```
import Levenshtein

s1 = "bayoh"
s2 = "boyhe"

LD = Levenshtein.distance(s1, s2)
print(f"Levenshtein Distance: {LD}")
```

```
Levenshtein Distance: 3
```

Picture 7. *LD with Levenshtein library*

1.3 Optimization

Optimizing the Levenshtein distance calculation can significantly reduce computational time and memory usage. In this case, we will talk about 3 common techniques:

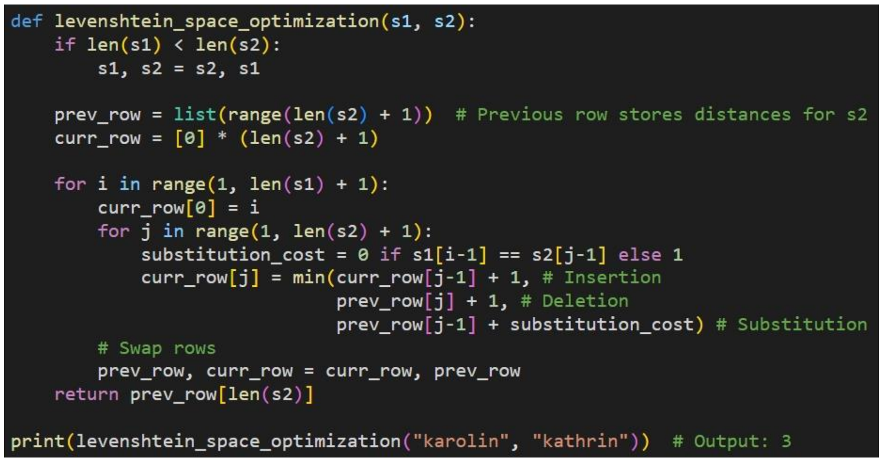
1. *Space Optimization*
2. *Bit-Parallelism*
3. *Upper Bound Optimization*

1.3.1 Space Optimization

Space optimization is essential in various scenarios due to its numerous advantages. In environments with limited memory, such as on older hardware, every byte matters, and efficient memory usage can significantly improve performance, particularly in large-scale applications. It also ensures scalability, allowing your application to handle larger inputs without crashing. Effective resource management reflects good programming practices, while in real-time applications, such as chatbots, quick response times are crucial, and memory optimization plays a key role in achieving that. Furthermore, in cloud computing, where you pay for resources consumed, reducing memory usage directly translates to cost-effectiveness. Optimized code is often more concise, making it easier to maintain, and can give you a competitive edge in the tech industry. Additionally, less complexity often leads to fewer bugs, making debugging simpler, and future-proofing your application for potential technological advancements. In addition, some techniques to optimize space in the **Levenshtein distance** calculation. They are:

1. **Reduce Matrix Size**: Instead of using a full 2D matrix, opt for a 1D array since only the current and previous rows are needed.
2. **Rolling Array Technique**: By maintaining only two rows at any time, you can reduce space complexity from *O(m * n) to O(min(m, n)).*
3. **Use a Hash Map**: For sparse matrices, a hash map can be more memory-efficient than storing a full matrix.
4. **Early Termination**: If the computed distance exceeds a predefined threshold, terminate early to save both space and computation time.
5. **Character Frequency Count**: When strings differ greatly in length, counting character frequencies can allow you to skip unnecessary calculations.
6. **Optimize Character Comparisons**: Speed up character comparisons by using a lookup table for common characters.
7. **Parallel Processing**: For systems with multiple cores, consider parallelizing the computation to enhance speed.
8. **Bit Manipulation**: In certain cases, bit manipulation techniques can help reduce space usage significantly.
9. **String Preprocessing**: Preprocess strings by removing unnecessary characters to minimize the workload before computing the distance.

10. **Lazy Evaluation**: Calculate distances only when required rather than precomputing the entire matrix, saving space.



```python
def levenshtein_space_optimization(s1, s2):
    if len(s1) < len(s2):
        s1, s2 = s2, s1

    prev_row = list(range(len(s2) + 1))  # Previous row stores distances for s2
    curr_row = [0] * (len(s2) + 1)

    for i in range(1, len(s1) + 1):
        curr_row[0] = i
        for j in range(1, len(s2) + 1):
            substitution_cost = 0 if s1[i-1] == s2[j-1] else 1
            curr_row[j] = min(curr_row[j-1] + 1, # Insertion
                              prev_row[j] + 1, # Deletion
                              prev_row[j-1] + substitution_cost) # Substitution
        # Swap rows
        prev_row, curr_row = curr_row, prev_row
    return prev_row[len(s2)]

print(levenshtein_space_optimization("karolin", "kathrin"))  # Output: 3
```

Picture 8. *Space optimization in Python for Rolling Array Technique (from O(m * n) to O(min(m, n)).*

1.3.2 Bit-Parallelism (Myer's Algorithm)

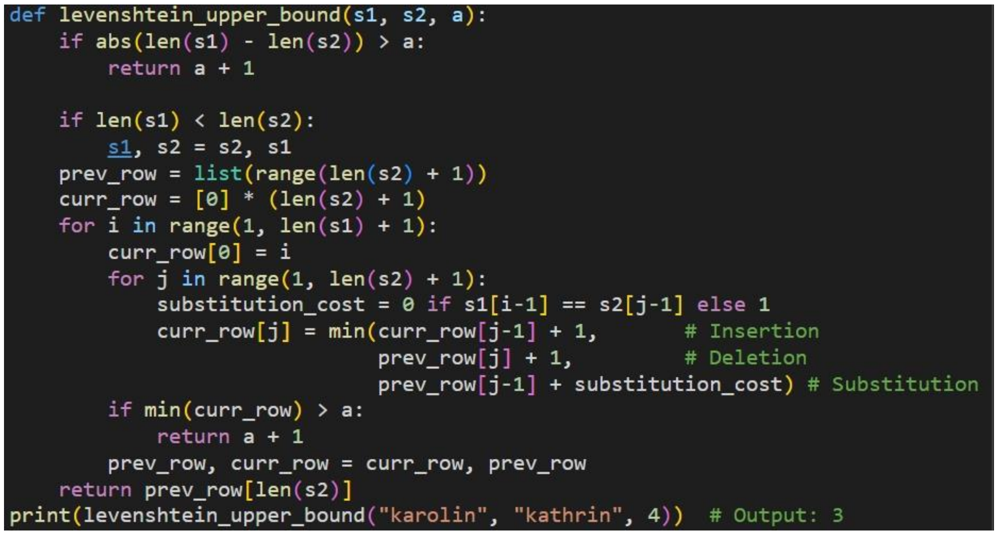
The simple idea is to 'parallelize' another algorithm using bits. The results are noteworthy from a practical perspective, particularly when short patterns are involved. Thus, this approach can be effective for any error level. Furthermore, this technique commonly used in string matching. It involves leveraging the intrinsic parallelism of bit operations within a computer word. In fact, the number of operations that an algorithm performs can be reduced by a factor of at most *w*, where *w* is the number of bits in a computer word. Since in current architectures *w* is 32 or 64, the speedup is very significant in practice and improves with technological progress. To relate the behavior of bit-parallel algorithms to other work, it is generally assumed that *w = log(n)*, as dictated by the RAM model of computation. Let's briefly explain the algorithm:The length of a computer word (in bits) is *w*.

- We denote as $b_k \dots b1$ the bits of a mask of length *k*. This mask is stored somewhere inside the computer word.
- We use exponentiation to denote bit repetition (e.g. $0^3 1$ = 0001).
- We use bitwise operations to handle multiple comparisons at once (i.e. bitwise-and, bitwise-or, bitwise-xor)
- We can perform arithmetic operations on the bits, such as addition and subtraction, which operate the bits as if they formed a number. For example, *0b10100 - 1 = 0b10011*

Eventually, it has the advantage of being much simpler, in many cases faster, and easier to extend in handling complex patterns than its classical counterparts. Its main disadvantage is the limitation it imposes on the size of the computer word.

1.3.3 Upper Bound Optimization

Initially, the idea is to pass an upper bound to lower levels so that further processing can be avoided once that bound is reached. In this case, only a single alpha (or beta) value is required. This method significantly prunes the solution tree—cutting off branches of the computation tree unlikely to yield a valid solution—while greatly enhancing performance. In this case, the algorithm stops early if the distance exceeds a specified maximum value and reduces the computation space by focusing only on parts of the problem where the distance could plausibly remain below the threshold.



```python
def levenshtein_upper_bound(s1, s2, a):
    if abs(len(s1) - len(s2)) > a:
        return a + 1

    if len(s1) < len(s2):
        s1, s2 = s2, s1
    prev_row = list(range(len(s2) + 1))
    curr_row = [0] * (len(s2) + 1)
    for i in range(1, len(s1) + 1):
        curr_row[0] = i
        for j in range(1, len(s2) + 1):
            substitution_cost = 0 if s1[i-1] == s2[j-1] else 1
            curr_row[j] = min(curr_row[j-1] + 1,       # Insertion
                              prev_row[j] + 1,         # Deletion
                              prev_row[j-1] + substitution_cost) # Substitution
        if min(curr_row) > a:
            return a + 1
        prev_row, curr_row = curr_row, prev_row
    return prev_row[len(s2)]
print(levenshtein_upper_bound("karolin", "kathrin", 4))  # Output: 3
```

Picture 9. *Upper Bound Optimization in Python*

2. Related metrics

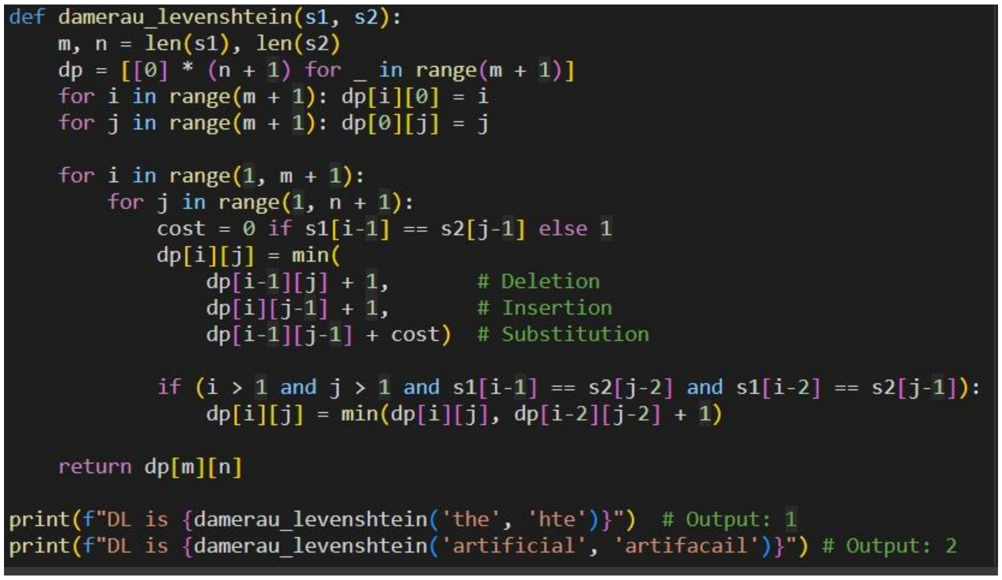
Related metrics include Damerau-Levenshtein, which also considers adjacent transpositions, and   Hamming, which counts substitutions between strings of equal length. Jaro-Winkler is another variant that emphasizes matching prefixes and is often used for short string comparisons. Let’s explain each of them briefly.

2.1 Damerau–Levenshtein distance

In information theory and computer science, the Damerau–Levenshtein distance is a string metric for measuring the edit distance between two sequences a and b. It can be defined using a function f{a, b}(i, j) where i and j represent the prefix length of string a and b respectively. It named after Frederick Damerau and Vladimir Levenshtein, the Damerau-Levenshtein distance extends the Levenshtein distance by allowing transpositions along with insertions, deletions, and substitutions.

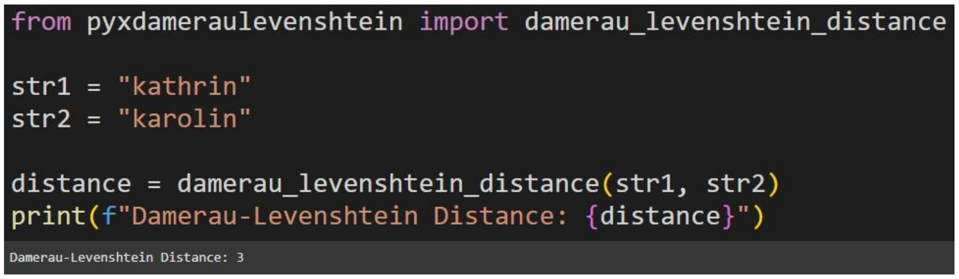
$$f_{a,b}(i,j) = \min \begin{cases} 0 & \text{if } i = j = 0 \\ f_{a,b}(i-1, j) + 1 & \text{if } i > 0 \\ f_{a,b}(i-1, j) + 1 & \text{if } j > 0 \\ f_{a,b}(i-1, j-1) + 1 \ (a_i \text{ and } b_j \text{ are not equal}) & \text{if } i,j > 0 \\ f_{a,b}(i-2, j-2) + 1 & \text{if } i,j > 1 \text{ and } a_{i-1} = b_j \text{ and } a_i = b_{j-1} \end{cases}$$

Picture 10. *Formula of Damerau-Levenshtein*



```python
def damerau_levenshtein(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(m + 1): dp[i][0] = i
    for j in range(m + 1): dp[0][j] = j

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            cost = 0 if s1[i-1] == s2[j-1] else 1
            dp[i][j] = min(
                dp[i-1][j] + 1,       # Deletion
                dp[i][j-1] + 1,       # Insertion
                dp[i-1][j-1] + cost)  # Substitution

            if (i > 1 and j > 1 and s1[i-1] == s2[j-2] and s1[i-2] == s2[j-1]):
                dp[i][j] = min(dp[i][j], dp[i-2][j-2] + 1)

    return dp[m][n]

print(f"DL is {damerau_levenshtein('the', 'hte')}")  # Output: 1
print(f"DL is {damerau_levenshtein('artificial', 'artifacail')}") # Output: 2
```

Picture 11. *The Damerau-Levenshtein distance in Python (manual)*



```python
from pyxdameraulevenshtein import damerau_levenshtein_distance

str1 = "kathrin"
str2 = "karolin"

distance = damerau_levenshtein_distance(str1, str2)
print(f"Damerau-Levenshtein Distance: {distance}")
```

```
Damerau-Levenshtein Distance: 3
```

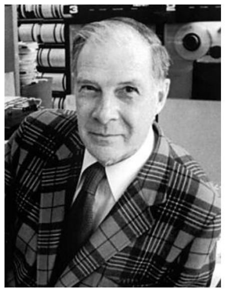
Picture 12. *The Damerau-Levenshtein distance in Python (library-based)*

2.2 Hamming Distance

In information theory, the Hamming distance between two strings or vectors of equal length is the number of positions at which their corresponding symbols are different. In other words, it allows only substitutions, which cost 1 in the simplified definition. In the literature the search problem is often referred to as "string matching with k mismatches". Consequently, the distance is symmetric, and it is finite whenever $|x| = |y|$.

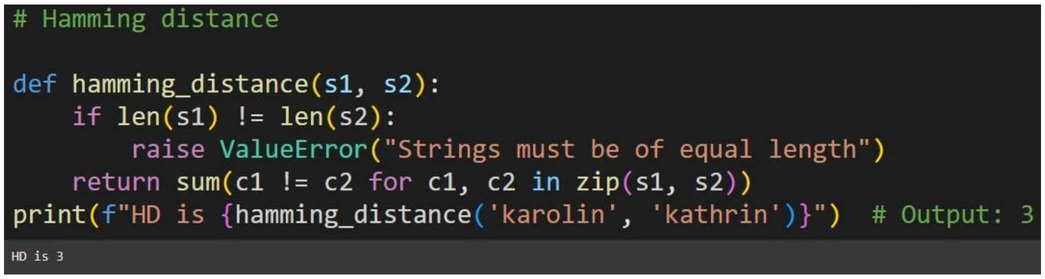
$$0 \leq d(x, y) \leq |x|$$

Moreover, it is named after the American mathematician Richard Hamming. Also, a major application is in coding theory, more specifically to block codes, in which the equal-length strings are vectors over a finite field. For example; "karolin" and "kathrin" is 3.



***Picture 13**. Richard Hamming*

He(February 11, 1915 – January 7, 1998) was an American mathematician whose work had many implications for computer engineering and telecommunications. His contributions include the Hamming code, the Hamming window, Hamming numbers, sphere-packing (or Hamming bound), Hamming graph concepts, and the Hamming distance.



```
# Hamming distance

def hamming_distance(s1, s2):
    if len(s1) != len(s2):
        raise ValueError("Strings must be of equal length")
    return sum(c1 != c2 for c1, c2 in zip(s1, s2))
print(f"HD is {hamming_distance('karolin', 'kathrin')}")  # Output: 3
```

```
HD is 3
```

***Picture 14 .** Hamming distance in Python*

2.3 Jaro–Winkler distance

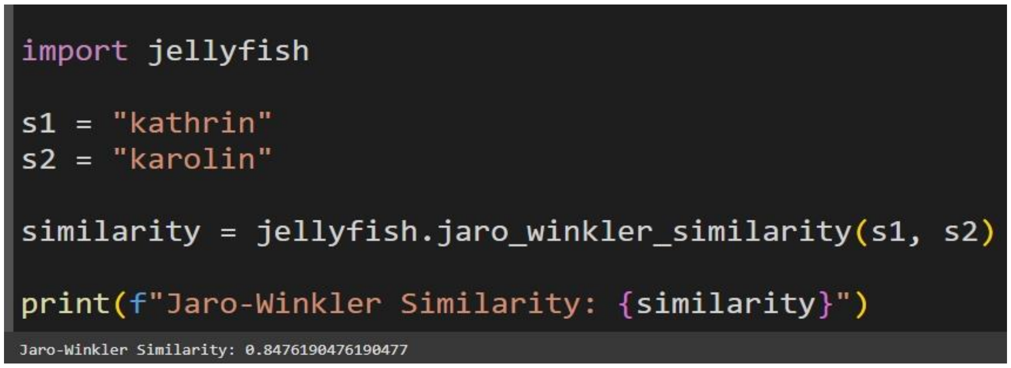
The Jaro–Winkler similarity measures the edit distance between two strings, favoring matches with common prefixes. It extends the Jaro distance and was proposed by William E. Winkler in 1990. The score ranges from 0 (exact match) to 1 (no similarity) but is not a true metric as it violates the triangle inequality.

Jaro, and Jaro–Winkler similarity formula are:

$$\textit{Jaro}(s1, s2) = \frac{1}{3}\left(\frac{m}{|s1|} + \frac{m}{|s2|} + \frac{m-t}{m}\right) \text{ if } m > 0\text{, else } 0.$$

$$JW = \textit{Jaro}(s1, s2) + p * l * (1 - \textit{Jaro}(s1, s2))\text{, where:}$$

- *m:* the number of matching characters
- *t:* half the number of transpositions among matching characters,
- *l:* the length of the common prefix
- *p:* a scaling factor for the prefix (mostly 0.1)



```python
import jellyfish

s1 = "kathrin"
s2 = "karolin"

similarity = jellyfish.jaro_winkler_similarity(s1, s2)

print(f"Jaro-Winkler Similarity: {similarity}")
```

```
Jaro-Winkler Similarity: 0.8476190476190477
```

Picture 15. *Jaro-Winkler similarity in Python*

3. Application Areas

1) Computational biology:

DNA and protein sequences can be seen as long texts over specific alphabets, representing the genetic code of living beings. Searching for specific sequences within these texts appeared as a fundamental operation for problems such as assembling the DNA chain from the pieces obtained by the experiments, looking for given features in DNA chains, or determining how different two genetic sequences are. As a result, searching DNA and protein sequences often involves allowing for errors, as biological mutations are common. The distance between sequences is measured by the minimum number of operations required to transform one into another.

2) Signal processing:

The physical transmission of signals is prone to errors. To ensure accurate communication over a physical channel, it is essential to recover the correct message despite potential modifications (errors) introduced during transmission. Signal processing theory determines the likelihood of such errors and assigns them a cost. In some cases, the exact message being sought is unknown; instead, the goal is to identify the most accurate text based on the applied error-correcting code and its closest match to the received message. Although this field has not seen significant advancements in approximate searching, it has contributed to the development of one of the most critical similarity measures: the Levenshtein distance (also known as "edit distance").

3) Text retrieval:

The problem of correcting misspelled words in written text is quite old—perhaps one of the earliest applications of approximate string matching. One of the most demanding fields where this challenge arises is Information Retrieval (IR), which focuses on finding relevant information within large text collections. String matching is a fundamental tool in this process. However, classical string matching is often inadequate due to the rapid growth of text collections, which are becoming larger, more diverse, and more error-prone. Nowadays, nearly every text retrieval system incorporates some form of error-tolerant search to address mistakes in both the database and the user's query. Beyond IR, approximate string matching is widely used in spelling checkers, natural language processing (NLP), command-line interfaces, computer-aided tutoring, and language learning applications.

4) Others:

The number of applications for string matching continues to grow daily. Problems relying on approximate string matching include, for instance, handwriting recognition, virus and intrusion detection, image compression, data mining, pattern recognition, optical character recognition, file comparison, and more.

---

4. Conclusion

As data continues to grow, it is essential to develop more sophisticated algorithms for data processing. Levenshtein published his landmark paper over half a century ago, laying the foundation for sequence comparison search. Nevertheless, the Levenshtein edit distance has played a central role both in the past and present. Additionally, his formulation of string metric distance remains relevant to this day and continues to inspire active research.

References

[1] Vladimir Levenshtein - wikipedia

[2] Levenshtein distance - wikipedia

[3] Haldar, R., & Mukhopadhyay, D. (n.d.). Levenshtein Distance Technique in Dictionary Lookup Methods: An Improved Approach. Web Intelligence & Distributed Computing Research Lab.

[4] Measuring Text Similarity Using the Levenshtein Distance - Ahmed Fawzy Gad

[5] Levenshtein VI, "Binary codes capable of correcting deletions, insertions, and reversals," (in Russian), Doklady Akademii Nauk, vol. 163, no. 4, pp. 845–848, 1965.

[6] Berger, B., Waterman, M. S., & Yu, Y. W. (n.d.). Levenshtein Distance, Sequence Comparison and Biological Database Search.

[7] Hamming distance - wikipedia

[8] Richard Hamming - wikipedia

[9] Damerau-Levenshtein distance - wikipedia

[10] Damerau-Levenshtein distance - geeksforgeeks

[11] Jaro-Winkler distance - wikipedia

[12] Jaro winkler vs Levenshtein Distance by Srinivas Kulkarni

[13] Navarro, G. A Guided Tour to Approximate String Matching.

[14] Winkler, W. E. (1990). String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. U.S. Bureau of the Census, Statistical Research Division.

[15] Gusfield, D. (1997). Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press.

[16] Efficient Recursive Levenshtein (Edit) Distance Algorithm

[17] Levenshtein distance - Rosetta Code

[18] Space Optimization in Levenshtein Distance Calculation