Nicole M. Ramirez Mulero

801-19-3940

Prof. Edusmildo Orozco

March 20, 2025

Final Project 6050 – Second Delivery

I. Statement of the Problem

Levenshtein Distance is an algorithm that measures the similarities between two strings. In essence it's a number that tells you how different 2 strings are. The higher this number is, the more different are the 2 strings. This calculation takes account insertion (what does need to be added to approach both strings to be equal), deletion (what does need to be removed to approach both strings to be equal), and replacement (what does need to be changed to approach both strings to be equal). This algorithm has application for text processing purposes, specifically data cleaning, searching and autocorrection. This project will be focusing on the spelling autocorrection aspect, and how this algorithm can predict what the user is intending to write and correct it automatically. Therefore, the problem that this project seeks to solve is "**Spelling Autocorrection method utilizing Leveshtein Distance Algorithm**".

II. Background and methodologies

The Levenshtein algorithm, also known as the Levenshtein distance or edit distance, is a method for measuring the similarity between two strings. This algorithm calculates the difference between two strings by calculating how many edits are needed for them to be identical. These edits are based on adding, replacing, or deleting a character. A point is added for each edit, and the higher the score, the more distinct the two strings are [1]. This algorithm was designed by

Vladimir Levenshtein, a Soviet mathematician, in 1965. Since its creation, this technique has been widely used in various fields.

The main function of this algorithm is to identify the differences and similarities between two lines of text. It does this by using a dynamic programming approach using an edit matrix. Each cell of the matrix contains the minimum edit distance between the two strings up to that point[3]. The matrix can be built as follows: If both characters are the same, only the number of the previous line diagonally is taken. If they are different, the smallest number above, to the left or diagonally left is taken, and if there are any extra characters, the number of extra characters that exist is added. An example follows.

	.	g	a	t	o
.	0	1	2	3	4
g	1	0	1	2	4
t	2	3	1	3	4
a	3	4	3	2	3
o	4	4	4	3	2

The levenstein distance between “gato” and “gtao” is 2.

The levenstein distance can seem has a simple and arbitrary algorithm, but it has important applications in industry. Spell checking and Autocorrection can be done with this algorithm has its helps identify the closest match to a misspelled word. Data cleansing can be achieve with this algorithm by identifying similar records and storing them in one group. Search engine results can be improved by detecting misspellings and looks for alternatives. Fraud and plagiarism detection can be done by identifying similarities and potential fraud and plagiarism.

All these applications are available thanks to the simplicity of looking the quantity of changes a text need to go through to approach the other.

In the case of this work, the Levenstein distance algorithm is used has a spelling autocorrection method. The idea is a program that can suggest an approximated word to fix the misspelling. The Levenstein algorithm is performed using dynamic programming on a matrix built with both text strings (This function is called levenshteinFullMatrix). The main idea of this application is that the user can enter a word, and the program can verify if it is spelled correctly or not and can give an approximate suggestion of what the user probably meant. For the scope of this project, the only words that the program can identify are a bank of 700 animal names[5]. LevenshteinFullMatrix is used by the CheckBestMatch function, which compares each word in the CSV file with the word entered by the user, looking for the word with the lowest Levenshtein distance. If it finds the same word, it tells the user that his spelling is correct, otherwise the program suggests a word to correct the spelling.

III. The Algorithms

A. Recursive pseudocode for Levenstein distance algorithm [4]:

```
def levenshteinFullMatrix(str1, str2):
[1]m = len(str1)
[2]n = len(str2)

#Create matrix
[3]dp = [[0 for _ in range(n + 1)] for _ in range(m + 1)]

#Initialize the first row and column with values form 0 to m and 0 to n
[4]for i in range(m + 1):
[5]   dp[i][0] = i
```

```
[6]for j in range(n + 1):
[7]    dp[0][j] = j

# Fill the matrix using dynamic programming to compute edit distances
[8]for i in range(1, m + 1):
[9]    for j in range(1, n + 1):
[10]        if str1[i - 1] == str2[j - 1]:
                # Characters match, no operation needed
[11]            dp[i][j] = dp[i - 1][j - 1]
[12]        else:
                # Characters don't match, choose minimum cost among
                #insertion, deletion, or substitution
[13]            dp[i][j] = 1 + min(dp[i][j - 1], dp[i - 1][j], dp[i - 1][j
                        - 1])
# Return the edit distance between the strings
[14]return dp[m][n]
```

B. Pseudocode for CheckBestMatch:

```
CheckBestMatch(data, word):
[1]bestMatchLD = levenshteinFullMatrix(data[0], word)
[2]bestMatchWord = data[0]
[3]data.pop(0)
[4]for i in range(len(data)):
[5]    LD = levenshteinFullMatrix(data[0], word)
[6]    if (LD <= bestMatchLD):
[7]        then bestMatchLD = LD
[8]        bestMatchWord = data[i]
[9]if(bestMatchLD == 0 and bestMatchWord != ""):
[10]   then print("Your spelling is correct")
[11]elif(bestMatchWord != ""):
[12]   then print("Word fix suggestion:",bestMatchWord)
[13]else:
[14]   print("No matches found")
```

IV. The Complexity Analysis

A. Time Complexity for Levenstein distance algorithm:

[1] O(1)

[2] O(1)

[3] O(m n)

[4] O (m)

[5] O(1)

[6] O(n)

[7] O(1)

[8] [9] O(m , n)

[10] O(1)

[11] O(1)

[12] O(1)

[13] O(1)

[14] O(1)

The overall complexity of this algorithm is T(m, n) = O(m * n)

B. Time Complexity for CheckBestMatch:

[1] O(1)

[2] O(1)

[3] O(k)

[4] O (m * n)

[5] O (1)

[6] O (1)

[7] O(1)

[8] O(1)

[9] O(1)

[10] O(1)

[11] O(1)

[12] O(1)

[13] O(1)

[14] O(1)

The overall complexity of this algorithm is T(m, n) = O(k * (m * n))

V. The computational tools

The implementation of this program was done with Python 3.0. It is done using a Jupiter Notebook for better organization. The csv library was used. The implementation of the levenshteinFullMatrix function was taken from the GeeksForGeeks website [4]. The csv file of animal names was taken from the EyeOfMidas github repository [5]. The rest of the code is from

the author of this Project. The implementation of the code in Jupiter Notebook can be found in the following github repository: https://github.com/Nicole-M-Ramirez/LD-Spelling-Checking

VI. Bibliography

[1] Liju, P. (2022). Algorithm to derive shortest edit script using Levenshtein distance algorithm.

[2] Mehta, A., Salgond, V., Satra, D., & Sharma, N. (2021). Spell correction and suggestion using Levenshtein distance. *Int Res J Eng Technol*, *8*(8), 1977-1981.

[3] Nadhia Nurin Syarafina, Jozua Ferjanus Palandi, & Nira Radita. (2021). Designing a word recommendation application using the Levenshtein Distance algorithm. *Matrix: Jurnal Manajemen Teknologi Dan Informatika*, *11*(2), 63–70. https://doiorg.uprrp.idm.oclc.org/10.31940/matrix.v11i2.2419

[4] GeeksforGeeks, "Introduction to Levenshtein distance," *GeeksforGeeks*. https://www.geeksforgeeks.org/introduction-to-levenshtein-distance

[5] "A simple csv of animal names - scraped from https://a-z-animals.com/animals/," *Gist*. https://gist.github.com/EyeOfMidas/311e77b8b8c2f334fc8bdaf652c1f47f