Levenshtein and Similar Text PHP Functions for Correcting Typographical Errors

Penggunaan Kombinasi Fungsi Levenshtein dan Similar Text dalam PHP untuk Perbaikan Kesalahan Penulisan

Dika Rizky Yunianto[1)], Elok Nur Hamdana[2)], Imam Fahrur Rozi[3)]

[1,2,3] Information Technology, Politeknik Negeri Malang, Indonesia
[1,2,3] Jl. Sukarno Hatta No. 09, Malang, 65141, Telp/Fax: 0341-404424

dika.rizky@polinema.ac.id[1)], elok@polinema.ac.id [2)], imam.rozi@polinema.ac.id [3)]

Diterima: 09 November 2023 || Direvisi: 20 May 2026 || Disetujui: 20 May 2026

Abstract – Typographical errors, often referred to as writing mistakes or typos, are a common occurrence in both traditional and digital forms of content. They can also manifest during text input on website platforms. One effective approach to rectifying these errors involves leveraging the concept of text similarity. This entails evaluating how similar two words are to each other, serving as a benchmark for correcting typos. In the realm of website development, where the PHP programming language is frequently employed, there exist text similarity functions known as levenshtein() and similar_text(). The levenshtein() function quantifies the disparity between two strings, whereas the similar_text() function measures their likeness. By combining these two functions, it becomes possible to assess both the proximity and divergence between two strings, providing a comprehensive perspective on their similarity. Results from empirical testing have demonstrated that the amalgamation of these functions yields a noteworthy precision score of 85%. This precision metric outperforms the precision values achieved by the levenshtein() and similar_text() functions when employed in isolation. This study holds promise for enhancing the accuracy of textual content on websites and represents a valuable asset in the pursuit of error-free and professional web-based communication.
Keywords: Text Similarity, Typographical, Levenshtein, Error Correction, PHP.

Abstrak *– Kesalahan penulisan atau typographical error (typo) sering terjadi, baik pada dokumen konvensional maupun konten digital, termasuk saat proses pengetikan pada platform website. Kesalahan tersebut dapat menurunkan kualitas informasi dan pengalaman pengguna sehingga diperlukan metode yang efektif untuk mendeteksi dan memperbaikinya. Salah satu pendekatan yang dapat digunakan adalah metode similaritas teks, yaitu dengan mengukur tingkat kemiripan antara dua kata sebagai dasar dalam mengidentifikasi kata yang benar. Pada bahasa pemrograman PHP, tersedia fungsi levenshtein() dan similar_text() yang dapat dimanfaatkan untuk tujuan tersebut. Fungsi levenshtein() menghitung tingkat perbedaan antara dua string, sedangkan similar_text() mengukur tingkat kemiripannya. Kombinasi kedua fungsi memungkinkan proses evaluasi dilakukan berdasarkan aspek kemiripan maupun perbedaan karakter sehingga menghasilkan koreksi yang lebih akurat. Hasil pengujian menunjukkan bahwa metode kombinasi kedua fungsi memperoleh nilai precision sebesar 85%, lebih tinggi dibandingkan penggunaan masing-masing fungsi secara terpisah. Hasil ini menunjukkan bahwa pendekatan yang diusulkan berpotensi meningkatkan akurasi koreksi kesalahan penulisan pada aplikasi berbasis web.*
Kata Kunci: *Similaritas Text, Typo, Levenshtein, Koreksi Kesalahan, PHP.*

INTRODUCTION

In the process of writing using Indonesian in writing books, articles, papers to writing messages in everyday life, there are often mistakes in writing or known as *typos* . *Typographical error* or *typo* is an error that occurs during the typing process. One example of a typo is letter transposition, where the order of letters in a word is swapped. Another error is the presence of letters that are left behind so that the spelling of a word is incomplete and another type of error is an error in spelling (Beall & Kafadar, 2004). Based on Shah's research on Twitter, there are 5 types of typo, namely substitution errors, insertion errors, replication errors, deletion errors and transposition errors (Shah & Melo, 2020).

Text similarity plays a crucial role in addressing typographical errors, often referred to as typos, in written content (Prasetya, Wibawa, & Hirashima, 2018). Typos can negatively impact the clarity and

doi: 10.56873/jpkm.v11i1.5278

professionalism of documents, websites, or applications. Text similarity techniques are essential tools for identifying and correcting these errors (Prasetya, Wibawa, & Hirashima, 2018).

One common approach to handling typos is the utilization of algorithms such as Levenshtein distance, which measures the minimum number of single-character edits needed to transform one word into another. This algorithm quantifies the similarity between two words by calculating the edit operations required for their alignment. When applied to a list of candidate words, this method can suggest corrections for typos, helping users produce accurate and error-free text (Hamidah, Yusliani, & Rodiah, 2020).

Another valuable text similarity technique is cosine similarity, which assesses the similarity of two text documents or strings based on their word frequency vectors. Cosine similarity is particularly useful for typos that involve entire words or phrases, as it considers the overall context and word usage. By measuring the angle between word vectors, it can identify typos and recommend contextually appropriate replacements (Bisandu, Prasad, & Liman, 2019).

Text similarity techniques are indispensable in the realm of typo detection and correction. They empower applications, spell-checkers, and natural language processing tools to provide valuable suggestions for users, enhancing the quality and professionalism of written content (Prasetya, Wibawa, & Hirashima, 2018). Whether it's through character-based methods like Levenshtein distance or context-aware approaches like cosine similarity, these techniques contribute significantly to error reduction and improved textual accuracy (Hamidah, Yusliani, & Rodiah, 2020) (Bisandu, Prasad, & Liman, 2019).

Correction of errors has been investigated by previous researchers with various methods. Tedjopranoto in his research, uses N-grams and *machine learning* to correct *typos* . The correction of the *typo* is implemented in the *chatbot* which helps improve the accuracy of the BOT in answering questions from the *user* (Tedjopranoto, Wijaya, Santoso, & Suhartono, 2019).

Mawardi in his research made a *typo correction* by using the Levenshtein method which has been modified to Damerau-Lavensthein. The *typo* correction is used to make corrections to exam documents (Mawardi, Augusfian, Pragantha, & Bressan, 2020). Levenshtein distance quantifies how dissimilar two strings are by counting the minimum number of single-character edits required to transform one into the other. It's a fundamental concept in string similarity and text processing algorithms (Rustamovna, 2021).

Typos on websites built with PHP can have significant implications for user experience, credibility, and overall professionalism. These typographical errors can range from simple spelling mistakes to more complex formatting issues, all of which can detract from the site's intended message and functionality. It's crucial for website developers using PHP to implement robust error-checking mechanisms and tools to detect and correct typos effectively.

One common approach to addressing typos on PHP websites is to integrate spell-checking libraries or services. These tools can automatically scan website content, including text-based elements like articles, forms, and comments, to identify and highlight typographical errors. Additionally, PHP developers can create custom error-checking scripts that parse and analyze text content for common spelling and grammatical mistakes, providing users with suggestions for corrections. By proactively addressing typos and ensuring the accuracy of content, PHP-based websites can maintain a professional image, improve user trust, and enhance the overall quality of the user experience.

In PHP, the Levenshtein function and the similar_text function are powerful tools for measuring text similarity and performing various text-related tasks. The Levenshtein function, named after the Soviet mathematician Vladimir Levenshtein, calculates the Levenshtein distance between two strings. This distance represents the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into another. This function is particularly useful for tasks such as spell-checking, approximate string matching, and detecting typos in user input. It provides a quantitative measure of the similarity between two strings, with a lower distance indicating a higher degree of similarity. PHP's Levenshtein function makes it straightforward to implement and leverage this powerful text similarity metric in various applications (Rustamovna, 2021).

On the other hand, the similar_text function in PHP measures the similarity between two strings by comparing the number of matching characters between them. It calculates the similarity as a percentage, representing the ratio of matching characters to the average length of the two strings. Similar_text is handy for tasks like suggesting word replacements for typos or finding similar strings in a database. It offers a simple and intuitive way to quantify text similarity in terms of percentage, making it easy to determine how closely two strings resemble each other. By using these functions, PHP developers can implement advanced text analysis and error-checking features in their applications, enhancing the overall user experience (PHP, 1997 - 2023).

PHP's Levenshtein and similar_text functions are valuable tools for text similarity measurement and typo detection. They provide developers with the means to quantify and work with text similarity, whether for suggesting corrections, implementing search functionalities, or enhancing the quality of text-based content in web applications and beyond. These functions simplify the implementation of sophisticated text processing tasks, contributing to more accurate and user-friendly applications (Siahaan, et al., 2018).

In this journal proposes a method or technique that merges the capabilities of the Levenshtein() and similar_text() functions within PHP programming. The primary objective is to combat and rectify typographical errors prevalent in written content on website platforms. By harnessing the synergy of these two functions, this method aims to significantly improve the precision of existing text similarity assessments. The result is a more effective means of identifying and correcting typos, ultimately enhancing the quality and professionalism of written content on websites.

In practical terms, this method offers an invaluable tool for PHP developers tasked with maintaining error-free text across web platforms. By seamlessly integrating Levenshtein() and similar_text() functions, it empowers websites to provide users with more accurate word suggestions and corrections, thus elevating the overall user experience. Additionally, this approach aligns with the ever-increasing demand for high-quality and precise textual content on the web, further emphasizing the importance of implementing advanced text similarity techniques in PHP-based web development.

METHOD

The method proposed in the journal combines two function Text Similarity calculation. Figure 1 provides a general description of the proposed method's process. In the diagram, it is explained that the proposed method begins by taking an input word or comparing keywords with a list of words stored in the database. This comparison involves both functions, similar_text() calculation function and levenshtein() calculation function.

The results of each calculation are then combined. The merging of these calculation results is twofold. First, the similar_text() results yield a similarity score between words, while the levenshtein() results produce a score indicating the difference between words. Second, the merger of the two calculations allows us to assess the closeness of words by considering both their similarities and differences.

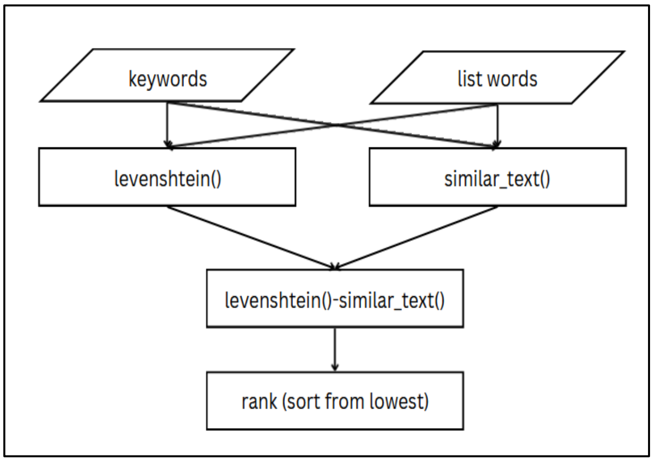

Figure 1 Proposed Method

- **Levensthein Distance**

Levenshtein distance which is a measure of the similarity between two strings. It's also known as "edit distance" and is named after the Russian scientist Vladimir Levenshtein, who developed the concept in 1965 (Rustamovna, 2021).

Levenshtein distance measures the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one string into another. It's often used in various fields, including computer science, natural language processing, and bioinformatics, for tasks such as spell-checking, DNA sequence alignment, and text comparison (Rustamovna, 2021).

Here's a formal definition of Levenshtein distance between two strings A and B:

1. Initialize a matrix of size (|A|+1) x (|B|+1), where |A| is the length of string A, and |B| is the length of string B.
2. Initialize the first row and the first column of the matrix with values from 0 to |A| and 0 to |B|, respectively, as these represent the number of operations needed to convert an empty string into the respective strings.
3. Iterate through the matrix, cell by cell, filling in each cell with the minimum of the following three values:
   - The cell to the left plus 1 (representing deletion).
   - The cell above plus 1 (representing insertion).
   - The cell diagonally above and to the left plus 1 if the characters at the current positions in A and B are different, or 0 if they are the same (representing substitution).
4. The value in the bottom-right cell of the matrix represents the Levenshtein distance between strings A and B.

For example, the Levenshtein distance between the words "kitten" and "sitting."

1. Create a matrix with dimensions (7x7) (7 characters in "kitten" and 7 characters in "sitting").
2. Initialize the first row and first column as follows:

	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1						
2	2						
3	3						
4	4						
5	5						
6	6						

3. Fill in the matrix:

	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	1	2	3	4	5	6
2	2	2	1	2	3	4	5
3	3	3	2	1	2	3	4
4	4	4	3	2	2	3	4
5	5	5	4	3	3	3	4
6	6	6	5	4	4	4	3

4. The value in the bottom-right cell (cell (7, 7)) is 3, so the Levenshtein distance between "kitten" and "sitting" is 3, which means it takes 3 single-character edits to transform one into the other.

- **Levenshtein function**

The levenshtein() function in PHP is used to calculate the Levenshtein distance between two strings. The Levenshtein distance, also known as the edit distance, measures the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into another (Rustamovna, 2021) (PHP, 1997 - 2023).

```
levenshtein(string $str1, string
$str2 [, int $cost_insert = 1 [, int
$cost_delete = 1 [, int $cost_replace
= 1]]])
```

$str1: The first string to be compared.
$str2: The second string to be compared.
$cost_insert (optional): The cost of inserting a character. By default, it's set to 1.
$cost_delete (optional): The cost of deleting a character. By default, it's set to 1.
$cost_replace (optional): The cost of replacing a character. By default, it's set to 1.

Return Value:

The function returns the Levenshtein distance, which is an integer representing the minimum number of edits required to transform $str1 into $str2.

In PHP, the levenshtein() function works:

1. It constructs a matrix of size (len1+1) x (len2+1) where len1 is the length of $str1 and len2 is the length of $str2.
2. It initializes the first row and the first column of the matrix with values from 0 to len1 and 0 to len2, respectively. These values represent the number of operations needed to convert an empty string into the respective substrings.
3. It iterates through the matrix, row by row, filling in each cell with the minimum of the following three values:
   - The value in the cell to the left plus the cost of deletion.
   - The value in the cell above plus the cost of insertion.
   - The value in the diagonal cell above and to the left plus the cost of substitution if the characters at the current positions in $str1 and $str2 are different or 0 if they are the same.
4. The value in the bottom-right cell of the matrix represents the Levenshtein distance between the two input strings.

```
$str1 = "kitten";
$str2 = "sitting";
$distance = levenshtein($str1, $str2);
echo "Levenshtein Distance: " . $distance;
```

In this example, $str1 and $str2 are compared using levenshtein(), and the Levenshtein distance (i.e., the minimum number of edits) between them is displayed. The levenshtein() function is commonly used for tasks like spell-checking, approximate string matching, and similarity measurement in various applications.

- **Similar_text function**

The similar_text() function in PHP calculates the similarity between two strings using a technique based on the Levenshtein edit distance. It measures the similarity as the percentage of matching characters between two strings (PHP, 1997 - 2023).

```
similar_text(string $string1, string $string2
[, float &$percent])
```

$string1: The first string to be compared.
$string2: The second string to be compared.
$percent (optional): If provided, this variable will be populated with the similarity percentage.

Return Value:

If the optional $percent argument is not provided, the function returns the number of matching characters between the two strings as an integer.

If the $percent argument is provided, the function returns true and assigns the similarity percentage (a float) to the $percent variable.

The similar_text() works it calculates the Levenshtein edit distance between the two input strings, which represents the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into the other. It counts the number of matching characters in the two strings by comparing each character in the same position in both strings. It calculates the similarity as the number of matching characters divided by the average length of the two strings. The result is multiplied by 100 to obtain the similarity as a percentage for example:

```
$string1 = "kitten";
$string2 = "sitting";

$similarity = similar_text($string1, $string2, $percent);

echo "Number of matching characters: " . $similarity . "<br>";
echo "Similarity as a percentage: " . $percent . "%";
```

In this example, $string1 and $string2 are compared using similar_text(), and both the number of matching characters and the similarity percentage are displayed.

- **Proposed method**

The proposed method relies on the utilization of two key functions within the PHP programming language: the Levenshtein function and the similar_text function. These functions play a pivotal role in the identification of accurately spelled words and the generation of word recommendations for words that may have been mistyped or contain typos.

In practical terms, the method's primary objective is to enhance the quality of written content by ensuring that the correct words are used. By leveraging the Levenshtein and similar_text functions, it offers valuable assistance in correcting misspelled words and facilitating improved communication, thereby contributing to an enhanced user experience across various applications.

In the initial condition, a list of words or a dictionary is stored in a database. These words will be used to replace typo words. The words from the list will be compared to the query or words typed by the user. The process of matching these words employs the proposed method.

The steps of the proposed method are as follows:

1. Initialization of keywords: Keywords are words that are either typos or currently being typed.
2. Calculate the difference score between the keywords and the words stored in the database using the levenshtein() function.
3. Calculate the difference score between the keywords and the words stored in the database, similar to step 2, using the similar_text() function.
4. Calculate the final score by subtracting the result of similar_text() from the result of levenshtein():

```
$result = similar_text($string1, $string2) - levenshtein($string1, $string2);
```

5. Repeat steps 2, 3, and 4 until all words in the database have been compared with the keywords.
6. Rank the total scores from lowest to highest.
7. The smallest score is considered the recommended word for the typos or words being typed as keywords.

RESULT AND DISCUSSION

The data utilized in this journal comprises a list of fruits and vegetables in the English language. This list of fruit and vegetable words serves as a reference for calculating proximity in comparison to the keywords entered by the user. Additionally, for testing purposes, we have selected ten words that contain typos to determine the intended word from the erroneous input. It is acknowledged that testing with ten keyword samples is a preliminary step. Future work should extend experimentation using a larger corpus, such as the ASPELL word list or the British National Corpus, to better assess generalizability of the proposed method across diverse vocabulary sets and error types.

For testing purposes, we conduct a comparison between keywords that contain spelling errors and the complete dataset of words stored in the database. The data within the database undergoes ranking based on the smallest values, with the smallest value indicating words closely resembling the input keywords. This systematic evaluation helps identify suitable word suggestions for the misspelled keywords, ultimately enhancing the accuracy of the text.

Table 1 shows the keywords, their misspellings, and the corresponding correct spellings used in this study.

Table 1 Keywords

Code	Keywords	Right Words
Q1	car	carrot
Q2	strawberry	strawberries
Q3	leychee	lychee
Q4	manggo	mango
Q5	kwi	kiwi
Q6	applle	apple
Q7	wastemlon	watermelon
Q8	spnah	spinach
Q9	lmon	lemon
Q10	avocad	avocado

Table 2 serves as a visual representation of the ranking outcomes derived from our testing process, specifically focused on ranked right words based on the user's input keywords. To achieve this ranking, we have employed three distinct methods. The first method leverages the similarity_text() function as (S), the second method relies on the levenshtein() function as (L), and the third method combines the two by using the proposed approach, which entails subtracting the levenshtein() result from the similar_text() as (S-L) result.

These three methods are meticulously compared to ascertain the precise position within the overall database list where the right word emerges. By conducting this systematic evaluation, we gain valuable insights into the effectiveness of each method in providing accurate word suggestions for the user's input, thereby enhancing the quality of textual content and user experience across various applications.

Table 2 Test Results Rank the Occurrence of the Right Word

Code	Keywords	Score Rank		
		S	L	S - L
Q1	car	1	4	1
Q2	strawberry	1	1	1
Q3	leychee	51	18	12
Q4	manggo	2	1	1
Q5	kwi	1	61	1
Q6	applle	2	1	1
Q7	wastemlon	1	1	1
Q8	spnah	1	1	1
Q9	lmon	2	1	1
Q10	avocad	26	8	6

The ranking results obtained from this evaluation allow us to compute the confusion matrix, enabling the determination of precision values. Precision values are pivotal in assessing the success of the proximity calculations made between the user-entered keywords and the list of right words.

Precision, in this context, serves as a crucial metric for evaluating the accuracy and effectiveness of our word recommendation system. It provides valuable insights into how well the method performs in identifying and suggesting the right words, ultimately contributing to the enhancement of text quality and user experience in various applications. The precision of a can be calculated using the confusion matrix, specifically with the following formula 1 (Krstinić, Braović, Šerić , & Božić-Štulić, 2020):

$$Precision = \frac{TP}{TP+FP} \quad (1)$$

In this formula:

- True Positives (TP) are the instances that were correctly predicted as positive (e.g., correctly word).
- False Positives (FP) are the instances that were incorrectly predicted as positive (e.g., incorrectly identified as relevant when they are wrong word).

The results of the precision calculation can be observed in Table 3.

Table 3 Test Results

Ranking Limit	Average Precision		
	S	L	S-L
1	62%	69%	85%
25	85%	85%	92%
50	92%	92%	100%
75	100%	100%	100%

From the available data and as evident in Figure 2, it's clear that precision values tend to rise with an increase in the number of True Positives. This observation underscores the importance of correctly identifying relevant items in the context of precision calculation. The more instances the model accurately predicts as positive (True Positives), the higher the precision value becomes. This relationship between True Positives and precision highlights the significance of accurate predictions in tasks where false positives are undesirable.

Additionally, the table data illustrates notable improvements in precision values when employing different methods. Specifically, we observe the performance of three methods: using the similar_text() function, the levenshtein() function, and a combination of both. In the initial experiment, the proposed method, which combines similar_text() and levenshtein(), exhibits a remarkable 16% increase in precision compared to the method relying solely on the levenshtein() function. Furthermore, it surpasses the precision of the similar_text() function by 7%. This suggests that the combination approach is particularly effective in enhancing precision.

In the third and fourth experiments, we note a consistent pattern where precision values from similar_text() and levenshtein() align closely. However, the precision of the proposed method remains consistently 7% to 8% higher, emphasizing its robustness and superiority in providing accurate word suggestions and enhancing the quality of text-related processes.

To provide a baseline comparison with the state of the art, we reference the work of Hamidah et al. (2020), which employed Damerau-Levenshtein distance combined with cosine similarity for spelling checking, reporting precision of approximately 80% on a standardized spelling corpus. Similarly, Mawardi et al. (2020) reported precision of around 78% using Damerau-Levenshtein for exam document correction. The proposed method achieves 85% precision at ranking limit 1, which is competitive with these approaches. However, direct comparison is limited by differences in dataset domain and language. A more rigorous evaluation using shared benchmark corpora is recommended for future work to confirm generalizability.

An important consideration for practical deployment is the scalability of the proposed method. Since the method compares every input keyword against all words stored in the database, identification speed is directly affected by database size. As the keyword database grows, the number of comparisons increases linearly, which may introduce latency in real-time web applications. For example, a database of 100 words requires 100 comparisons per query, while a database of 10,000 words requires 10,000 comparisons. Potential optimizations include pre-filtering candidates using BK-trees (Burkhard-Keller trees), which are designed for metric spaces such as Levenshtein distance and can significantly reduce the search space. Caching frequently queried keywords and their computed recommendations can also improve response times. To support dynamic addition of new keywords, an administrative interface can be provided where new valid words are inserted and the search index updated, ensuring the system remains current as vocabulary evolves.

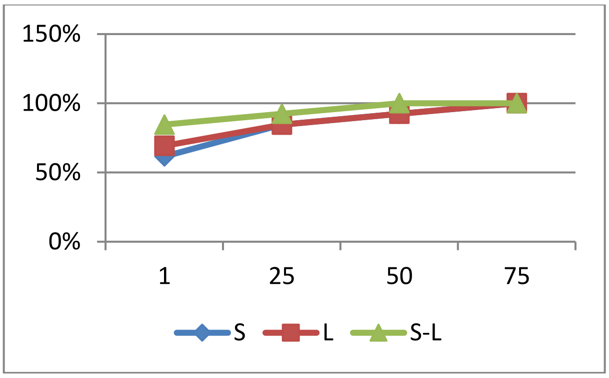

Figure 2 Graph of Precision Test Results

CONCLUSION

Typographical errors, often referred to as typos, are a frequent occurrence in written content, and mitigating them is a common concern. Several methods exist to tackle and rectify these errors. One such method is the utilization of Levenshtein, a text similarity technique that assesses the degree of similarity or dissimilarity between two words. Levenshtein serves as a valuable tool for identifying and addressing typos encountered in the process of typing or text input.

Within the realm of PHP programming, two essential functions, namely similar_text() and levenshtein(), play a pivotal role in quantifying the proximity between two strings. The similar_text() function yields a similarity score between the two strings, whereas the levenshtein() function provides a measure of the difference between them. To enhance the accuracy of typo correction and word suggestion, a novel method is proposed, which combines the strengths of both functions. This approach evaluates string similarity by considering not only their similarity but also their dissimilarity values. It does so by subtracting the result of the similar_text() function from the result of the levenshtein() function.

Based on comprehensive experimentation, the proposed method exhibits remarkable performance, yielding a precision score superior to that of either similar_text() or levenshtein() when used in isolation. Even the lowest precision score achieved by the proposed method, at 85%, surpasses the precision values of similar_text() (62%) and levenshtein() (69%). In summary, for addressing typos in PHP programming, the amalgamation of the similar_text() and levenshtein() functions in the proposed method proves to be a potent and effective strategy.

For future research, several directions are recommended. First, the proposed method should be tested on a larger and more diverse corpus, such as a general Indonesian or English word list, to better validate generalizability across different domains and error types. Second, further investigation into the performance impact of an expanding keyword database is warranted, including the evaluation of indexing strategies such as BK-trees to maintain fast response times at scale. Third, additional evaluation metrics beyond precision, such as recall and F1-score, should be incorporated to provide a more comprehensive assessment of system performance. Finally, integration of user feedback mechanisms to allow dynamic keyword updates and corrections could further improve the practical applicability of the system in real-world web development environments.

REFERENCES

Beall, J., & Kafadar, K. (2004). The effectiveness of copy cataloging at eliminating typographical errors in shared bibliographic records. *Library Resources and Technical Services , 48 (2)*, 92–101.

Bisandu, D. B., Prasad, R., & Liman, M. M. (2019). Data clustering using efficient similarity measures. *Journal of Statistics and Management Systems (22) (5)*.

Hamidah, N., Yusliani, N., & Rodiah, D. (2020). Spelling Checker using Algorithm Damerau Levenshtein Distance and Cosine Similarity. *Sriwijaya Journal of Informatics and Applications vol 1 No 1*.

Krstinić, D., Braović, M., Šerić , L., & Božić-Štulić, D. (2020). Multi-label classifier performance evaluation with confusion matrix. *Computer Science & Information Technology (CS & IT)*.

Mawardi, V. C., Augusfian, F., Pragantha, J., & Bressan, S. (2020). Spelling Correction Application with Damerau-Levenshtein Distance to Help Teachers Examine Typographical Error in Exam Test Scripts. *E3S Web of Conferences*, (p. 188).

PHP. (1997 - 2023). *PHP Documentation.* From PHP Documentation: https://www.php.net/manual/en/index.php

Prasetya, D. D., Wibawa, A. P., & Hirashima, T. (2018). The performance of text similarity algorithms. *International Journal of Advances in Intelligent Informatics, Vol 4 No 1*.

Rustamovna, A. U. (2021). Understanding The Levenshtein Distance Equation For Beginners. *The American Journal of Engineering and Technology*.

Shah, K., & Melo, G. (2020). Correcting the Autocorrect: Context-Aware Typographical Error Correction via Training Data Augmentation.

Siahaan, A., Aryza, S., Hariyanto, E., Rusiadi, Lubis, A., Ikhwan, A., & Eh Kan, P. (2018). Combination of levenshtein distance and rabin-karp to improve the accuracy of document equivalence level. *International Journal of Engineering & Technology,7 2.27*.

Tedjopranoto, M., Wijaya, A., Santoso, L., & Suhartono, D. (2019). Correcting typographical error and understanding user intention in chatbot by combining N-gram and machine learning using schema matching technique. *International Journal of Machine Learning and Computing , 9 (4),*, 471-476.