International Journal of Artificial Intelligence Research
Vol. 8, No. 2, December 20224, pp. 231-247
ISSN 2579-7298

Optimizing Text Correction For Voice Based IoT Smart Building Virtual Assistants

Maulana Ahmad As Shidiqi [a,1], Mokh. Sholihul Hadi [a,2,*], Aji Prasetya Wibawa [a,3] , Mhd. Irvan [b.4]

[a] *Department of Electrical Engineering and Informatics, Universitas Negeri Malang, Indonesia*
[b] *Graduate School of Information Science and Technology, The University of Tokyo Tokyo, Japan*

[1] Maulana.ahmad.2105348@students.um.ac.id; [2 *]mokh.sholihul.ft@um.ac.id; [3] aji.prasetya.ft@um.ac.id, [4]irvan@yamagula.ic.i.u-tokyo.ac.jp
* corresponding author

ARTICLE INFO

Article history
Received 21 Dec 2023
Revised 31 May 2024
Accepted 21 Oct 2024

Keywords
Text Correction
Virtual Assistant
Internet of Things

ABSTRACT

The integration of Virtual Assistants (VAs) within Smart Building Internet of Things (IoT) ecosystems is increasingly critical, particularly for interpreting user commands via Automatic Speech Recognition (ASR). This paper presents an in-depth performance analysis of text correction algorithms on a Raspberry Pi 4—a cost-effective and widely used computing solution in smart building applications. Due to the absence of GPU acceleration for Python on ARM architecture, a specialized dataset was developed to benchmark algorithmic performance, focusing on correction times and accuracy. Our study utilized a near-real-world experimental setup, deploying Docker containers to simulate IoT MQTT brokers, a Smart Building Platform, and Rasa for dialogue management. Among the algorithms tested—Edit distance, Jaccard, FuzzPartialRatio, FuzzSortRatio, MLE, and Norvig Spell—the Edit distance and Norvig Spell emerged as leaders in accuracy, achieving an 84% success rate in text correction. Notably, the Edit distance algorithm demonstrated superior speed, vital for real-time processing demands. The Fuzz Sort Ratio algorithm distinguished itself with the fastest correction time at 31.6 milliseconds, albeit with a slight compromise on accuracy, attaining a 79% success rate. Consequently, the Edit distance algorithm is recommended for applications where accuracy and response time are paramount, while the Fuzz Sort Ratio is preferable for scenarios where speed is the overriding priority. This research paves the way for future exploration into the computational impacts of these algorithms and the exploration of neural network-based methods to further enhance text correction capabilities in smart building automation systems.

1. Introduction

The building has been an essential element in human life since ancient times. Several decades ago, humans needed security from damage and earthquake destruction. However, in today's world, users seek more than just security[1]; they desire a quality-of-life experience[2]. This includes the ability for systems to automatically control HVAC (Heating, Ventilation, Air Conditioning) systems, lighting, electricity, energy, and access control[3]. Researchers have coined the terms "Smart Building" or "Intelligence Building" to describe the research and products related to these innovations. This concept emerged in the 1980s in the United States, where the Intelligent Building Institution explained the need for a system that integrates various systems to coordinate with each other, aiming to maximize operating cost savings, improve return on investment (ROI), and provide flexibility in management[4], [5]. The realization of Smart Buildings is made possible by the

Internet of Things (IoT), which is a technology development from machine-to-machine (M2M) communications, enabling machines such as HVAC equipment, lighting, and electricity to communicate via the internet[6]. The demand for such technology has been growing steadily and is predicted to have a compound annual growth rate (CAGR) of around 13% until 2030, with the number of devices expected to increase from 8.6 billion in 2022 to 29 billion in 2030[7].

Currently, Smart Buildings not only utilize Graphical User Interfaces (GUI) as Human Machine Interfaces (HMI), but they also leverage Voice User Interfaces (VUI) [8], [9] , which have been gaining popularity year after year. Unfortunately, ASR is reliable because running on server which is has a lot of resource to do that with highly accurate. The journey of ASR began in the 1950s with the Three Bell Labs Researchers who designed an Automatic Speech Recognition (ASR) system called Audrey, which could recognize digits from 0 to 9. In 1962, IBM demonstrated Shoebox, a system capable of understanding up to 16 spoken words in English. In 1971, IBM created the Automatic Call Identification system [10] In the 1970s, DARPA conducted Speech Understanding Research with a vocabulary size of one thousand words. Additionally, they developed the Hidden Markov Model (HMM)[11] as an advancement of the Markov Chain applied to ASR. In the 1980s, an IBM team developed the Tangora voice-activated typewriter, utilizing HMM technology. In short, In the development of ASR, in 1990, the Sphinx-II ASR system developed by CMU became the first to recognize continuous speech with a wide vocabulary[12]. The application of Deep Neural Networks (DNN) in the early 2000s [13] gained significant attention in the ASR field, as DNNs have high learning capacity and can model complex nonlinear relationships in speech data. The "end-to-end" approach and Attention-based ASR models began to be used in the mid-2010s, allowing for simultaneous learning of all components of speech recognition and yielding better overall speech recognition performance[14]. Even though performance accuracy and error rates are getting better and better every year, a super high-powered computational engine should be available[15], [16]. Despite user acquisitions each year grow rapidly. It still needs a privacy concern especially Europe standard GDPR became harder to entry[17], [18]. One solution that handy is running all of that inside the machine, not go to outside server [19]. This is why speech recognition on our devices needs a lot of storage and fast processing, or the users must make up for the fact that the lowering the quality of recognition. The alternative that recognizes inside the machine but increasing accuracy with fast correction algorithms[18].

The evolution of Smart Buildings, supported by the Internet of Things (IoT), has brought a significant shift from traditional structures to ones that offer a comprehensive quality-of-life experience. This research highlights the strength of Smart Buildings in their ability to integrate various systems—such as HVAC, lighting, and access control—to optimize operational costs, improve ROI, and offer management flexibility. The deployment of Voice User Interfaces (VUI) through Automatic Speech Recognition (ASR) systems presents a modern approach to Human-Machine Interfaces, marking a notable advancement from the early days of digit recognition to today's sophisticated end-to-end models with attention mechanisms.

However, the research also reveals certain weaknesses. While ASR systems have improved, their reliance on powerful server-side resources for high accuracy poses challenges, especially considering the privacy concerns under regulations like Europe's GDPR. This necessitates a move towards on-device processing to safeguard user privacy, which in turn demands substantial storage and computational speed on local machines. Herein lies a trade-off: ensuring user privacy and data security may result in compromised recognition quality unless offset by rapid and precise text correction algorithms[20]–[24].

2. Method

In this study, we aimed to evaluate the performance of text correction for smart building virtual assistants. Raspberry Pi 4 is used for this study to measure how fast the algorithms perform in low-cost computing devices Despite the SoC (single-on-chip) there is GPU available which theoretically can use GPU acceleration, none of the python itself or python libraries the including Tensorflow doesn't support GPU acceleration for ARM architecture [25] which is cannot utilizing parallelism. That's why this research calculates the time as performance.

We collected a dataset of utterances related to smart building commands, where each record contained the original command, an incorrect version of the command, and the corrected output from different correction algorithms. These included methods[26], [27] such as Edit distance (Levenshtein) [28], [29],Jaccard[30], [31], FuzzPartialRatio[32], [33], FuzzSortRatio, MLE[34], and Norvig Spell algorithm[35]. For each correction algorithm, we recorded the correction time, the corrected command, and whether the correction was correct or not.

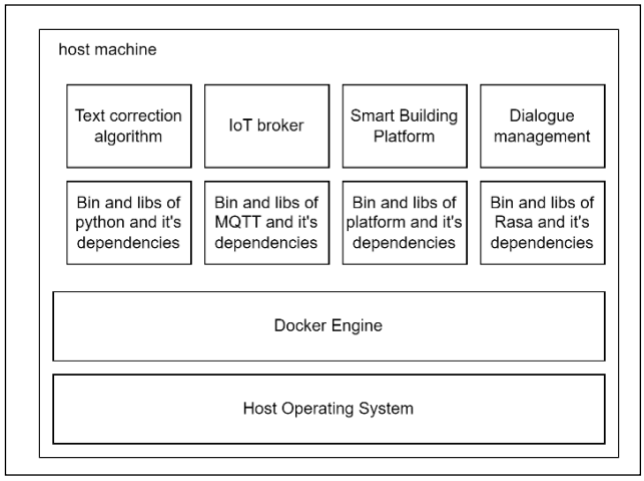

Fig. 1. Edge Computer architecture

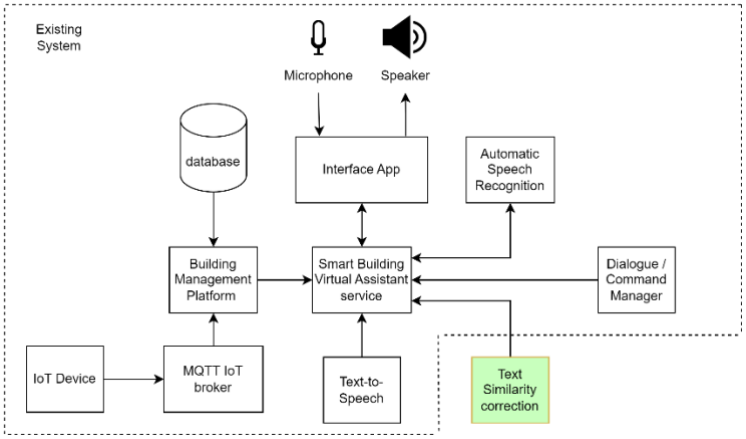

Fig. 2. Voice-based Virtual Assistant for Smart Building

2.1. Experiment Scenario

This experiment contains a text correction application using Docker containers[36] on a Raspberry Pi 4 (see figure 1), along with other dockerized applications is running to make sure such as an IoT MQTT broker which function as broker between IoT device and the platform[37] , a Smart Building Platform as IoT devices manager[38], and Rasa dialogue management [39] as interface between user and smart building platform. Docker containers offer numerous benefits, including the ability to isolate each application, thereby preventing conflicts between libraries and binary dependencies [40]. This ensures the smooth operation and compatibility of the system's various components. If look at the whole system (see figure 2), there are a lot of components in a voice-based virtual assistant for smart building in an existing system; such as a building management platform that has a database, IoT broker service and its devices, automatic speech recognition, dialogue management, and text-to-speech; and an additional system for this research, which is text similarity correction system.

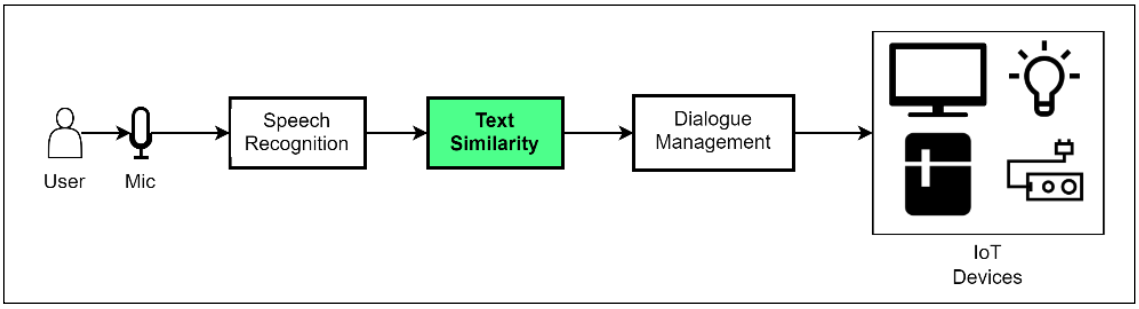

Fig. 3. Text Similarity Process Location

Text Similarity occurs after the user's voice has been converted to string text by speech recognition (see figure 3). The output of the Text Similarity function is the corrected string text which is required for dialogue management to ensure the string is matched. After a match has been made, the dialogue management will instruct the IoT device via Smart Building Platform to turn on or off depending on the command. The analysis phase of the text similarity process involved evaluating the accuracy and speed of each correction algorithm. We measured accuracy by comparing the corrected command against the original command and determining the percentage of successful corrections. Speed was evaluated based on the recorded correction time. In order to improve the efficiency of the entire system's process (see figure 4), the system only performs text similarity correction if the query didn't match after being fed to the dialogue manager.

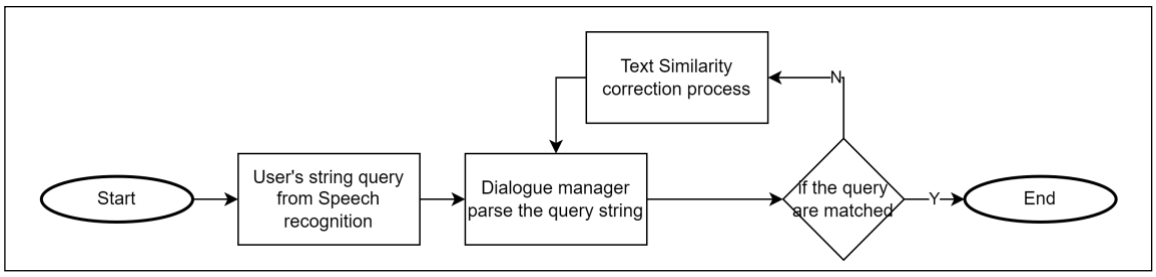

Fig. 4. Text Similarity execution Algorithm

2.2. Text correction method

The first step is the registered devices, and their corresponding commands are recorded and stored in a dictionary within the text correction system. This dictionary serves as a reference for the system to recognize and correct any input that corresponds to a device and it's synonym and command. By maintaining this dictionary, the text correction system can accurately identify and correct user queries related to specific devices or commands. And then when a user enters a string query command, it is passed through each algorithm within the text correction system. Before applying the algorithms, a preprocessing step is performed on the query. This preprocessing includes converting the query to lowercase using the `lower()` function and tokenizing it by whitespace by using `tokens = nltk.word_tokenize(user_query)`. These steps help standardize the input and ensure consistency in the text correction process. and then the next step is correcting token by token and match it within the dictionary, if there is a closed word between every token user_query and the dictionary, the algorithm will return the correction instead. In every user query, before and after the correction method the time will be recorded to measure how fast the algorithm is. This process is based on pseudocode below.

```
dictionary = ["on", "off", "fan", "tv", "television", "PC", "lamp", ...]
user_query = # String var
user_query = user_query.lower() # Convert query to lowercase
tokens = nltk.word_tokenize(user_query) # Tokenize the user query string
start_time = time.time() # Start measuring time
corrected_tokens = []  #create a new array of text
for token in tokens:
    if token in [".", ",", "!", "?", ":", ";", "the"]:
        corrected_tokens.append(token)
        continue
    result = text_correction(token)
    corrected_tokens.append(result)
elapsed_time = time.time() - start_time
corrected_sentence = " ".join(corrected_tokens)
```

Table 1. Pseudocode text correction process

2.2.1. Edit Distance

The edit distance[41]–[43] is a metric that measures the dissimilarity or similarity of two strings[44]. It quantifies the minimal number of operations required to transform one string into another, where each operation may involve the insertion, deletion, or substitution of a single character. The edit distance algorithm evaluates each character in both strings and computes the cost of transforming one into the other. Typically, 1 is assigned to the cost of each operation. The algorithm seeks to identify the sequence of operations with the lowest total cost. A common method for calculating the edit distance is based on dynamic programming. It requires constructing a matrix called the edit distance matrix, where each cell represents the cost of transforming a prefix of one string into a prefix of the other string. The matrix is initialized with the base cases and then iteratively filled in using the following recurrence relation:

a. If the current characters in both strings are identical, the current cell's cost is identical to the previous cell's cost.
b. If the characters are different, the cost in the current cell is the minimum of the cost in the left cell (corresponding to deletion), the cost in the upper cell

(corresponding to insertion), or the cost in the diagonal cell (corresponding to substitution).

c. Once the edit distance matrix is constructed, the minimum cost, which is in the bottom-right cell of the matrix, represents the edit distance between the two strings. This value indicates the minimum number of operations required to transform one string into the other.

For the pseudocode, see table 2.

Table 2. Pseudocode Edit distance

```
def calculate_edit_distance(string1, string2):
    m = len(string1)
    n = len(string2)

    # Create a matrix to store the edit distances
    distances = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialize the first row and column of the matrix
    for i in range(m + 1):
        distances[i][0] = i
    for j in range(n + 1):
        distances[0][j] = j

    # Calculate the edit distances
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if string1[i - 1] == string2[j - 1]:
                distances[i][j] = distances[i - 1][j - 1]
            else:
                substitute_cost = distances[i - 1][j - 1] + 1
                delete_cost = distances[i - 1][j] + 1
                insert_cost = distances[i][j - 1] + 1
                distances[i][j] = min(substitute_cost, delete_cost,
insert_cost)
    return distances[m][n]

def editdistance_function(text):
    outcomes = []
    distances = [(edit_distance(lowercase_text, word), word) for word
in spellings_series]
    closest = sorted(distances)[:3]
    outcomes.append(closest)
    return outcomes[0][0][1], outcomes[0][0][0]
```

2.2.2. Jaccard Index

The Jaccard similarity coefficient[45], also known as the Jaccard index, measures the overlap between two sets by computing the ratio between their intersection and union sizes. Given two sets A and B, the Jaccard index (J(A, B)) is calculated as the intersection cardinality of A and B divided by the union cardinality of A and B ($J(A, B) = |A \cap B| / |A \cup B|$). The resultant coefficient ranges from 0 to 1, with 0 indicating no overlap between the sets and 1 indicating perfect similarity. In situations when the presence or absence of elements in sets is of vital importance, the Jaccard index

is especially useful. It is notably useful for tasks involving set-based representations, such as document similarity, recommendation systems, and clustering. The Jaccard index, unlike other similarity indices, only evaluates set membership and ignores the frequency or order of elements within sets. This makes it resistant to fluctuations in set sizes and excellent for processing sparse and high-dimensional data. In addition, the Jaccard index is computationally efficient because it only requires the calculation of set intersections and unions. For pseudocode see table 3.

Table 3. Jaccard Index Pseudocode

```
def jaccard(string, gram_number):
    outcomes = []

    for entry in entries:
        startwith = spellings_series.str.startswith(entry[0])
        spellings = spellings_series[startwith]
        distances = []

        for word in spellings:
            wordgram = nltk.ngrams(word, gram_number)
            entrygram = set(nltk.ngrams(entry, gram_number))
            union = entrygram.union(set(wordgram))

            if len(union) == 0:
                continue

            distance = jaccard_distance(entrygram, set(wordgram))
            distances.append((distance, word))

        closest = sorted(distances)[:1]
        outcomes.append([match for match in closest])

    if len(outcomes) == 0 or len(outcomes[0]) == 0:
        return entry, 0
    else:
        return outcomes[0][0][1], outcomes[0][0][0]
```

2.2.3. TheFuzz (FuzzyWuzzy)

The popular Python module "thefuzz"[32], [33] (formerly known as "fuzzywuzzy") provides fuzzy string matching functionality. It provides a variety of string similarity calculation methods, enabling the approximate matching and correction of text data. Methods such as Partial Ratio, and Token Sort Ratio are supported and used in this research.

a. Partial Ratio
   This method considers the best matching substring between the two strings. Instead of comparing the entire strings, it identifies the most similar substring in both strings and calculates the ratio of similarity for that substring. Like the simple ratio, this ratio is also scaled to a number between 0 and 100. For pseudocode, see table 4.

Table 4. Jaccard Index Pseudocode

```
def partial_ratio(s1, s2):
  s1, s2 = utils.make_type_consistent(s1, s2)
  if len(s1) <= len(s2):
    shorter = s1
    longer = s2
  else:
    shorter = s2
    longer = s1
  m = SequenceMatcher(None, shorter, longer)
  blocks = m.get_matching_blocks()
  scores = []
  for block in blocks:
    long_start = block[1] - block[0] if (block[1] - block[0]) > 0
else 0
    long_end = long_start + len(shorter)
    long_substr = longer[long_start:long_end]
    m2 = SequenceMatcher(None, shorter, long_substr)
    r = m2.ratio()
    if r > .995:
      return 100
    else:
      scores.append(r)

  return utils.intr(100 * max(scores))
```

b. Sort Ratio

This method tokenizes the strings into words, sorts the words alphabetically, and then joins them back into a string. The ratio between these processed strings is then calculated. This method is useful when comparing strings where the word order might not be the same but the same words are present. For pseudocode, see table 5.

Table 5. Sort Ratio Pseudocode

```
def _process_and_sort(s, force_ascii, full_process=True):
  ts = utils.full_process(s, force_ascii=force_ascii) if full_process
else s
  tokens = ts.split()
  sorted_string = " ".join(sorted(tokens))
  return sorted_string.strip()

def _token_sort(s1, s2, partial=True, force_ascii=True,
full_process=True):
  sorted1 = _process_and_sort(s1, force_ascii,
full_process=full_process)

def token_sort_ratio(s1, s2, force_ascii=True, full_process=True):
  return _token_sort(s1, s2, partial=False, force_ascii=force_ascii,
full_process=full_process)
```

2.2.4. Maximum Likelihood Estimation

MLE [46]in short is a statistical method widely used for estimating the parameters of a probability distribution based on observed data. In the context of natural language processing and text correction[47]. The main objective of MLE is to find the parameter values that maximize the likelihood of observing the given data. It starts by compiling a list of correct spellings from user inputs and commands. This list is then converted into a string representation. The training data is generated by tokenizing the string and creating n-grams, which are contiguous sequences of n tokens. An MLE model is then initialized, specifying the desired order of the model. The model is trained on the generated training data, using the MLE algorithm to estimate the probability distribution of word sequences.

The `predict_next_chars` function takes an input string and aims to predict the most likely next words based on the trained MLE model. It utilizes a context-based approach, where the input string is used to determine the initial context for prediction. The function iterates a specified number of times, generating words using the MLE model's generate method. Each generated word is appended to the list of predicted characters until a period ('.') is encountered, indicating the end of a sentence or a significant break in the predicted sequence. The function returns the input string concatenated with the predicted characters, forming a corrected and extended version of the original input. For the pseudocode, see table 6.

Table 6. MLE Pseudocode

```
# Training phase
correct_spellings_mle = [...]
correct_spellings_string = '.'.join(correct_spellings_mle)
tokens = list(correct_spellings_string)
n = 3
train_data = list(ngrams(tokens, n))
model = MLE(n)
model.fit([train_data], vocabulary_text=tokens)
# Text correction phase
def predict_next_chars(input_str, num_word=10, n=2):
    original_input_str = input_str
    # Prepare the context
    if len(input_str) >= n:
        input_str = input_str[:n-1]
    context = tuple(input_str)
    context = ('.',) + context

    predicted_chars = []
    for i in range(num_word, 0, -1):
        try:
            next_chars = model.generate(num_words=num_word-i,
text_seed=context)
            if '.' in next_chars:
                idx = next_chars.index('.')
                predicted_chars.extend(next_chars[:idx])
                break
            predicted_chars.extend(next_chars)
            context = (context + tuple(next_chars))[-(n-1):]
        except:
            continue
    if not predicted_chars:
        return input_str
```

```
    return input_str + ''.join(predicted_chars)
```

2.2.5. Norvig Algorithm

The Norvig algorithm[48], [49], developed by Peter Norvig, is a widely used approach for text correction and spell checking. It employs statistical analysis and probabilistic models to suggest the most likely spelling corrections for misspelled words. The algorithm is based on the principle that the correct spelling of a word is the one that has the highest probability given the context in which it appears.

The algorithm prepares training data. This corpus extracts words, normalizes them to lowercase, then calculates their frequencies using a Counter object. Probabilistic models are based on frequency distributions. The system uses potentially misspelled words to fix text. It suggests corrections within one or two edit distances from the original word. The algorithm generates candidates using known(), edits1(), and edits2(). The known() function filters candidate adjustments by checking the training corpus frequency distribution. Only valid words are considered for repairs. The edits1() function generates corrections one edit from the original word, considering deletions, transpositions, replacements, and insertions. Edits2() adds two-edit-distance corrections to edits1(). The algorithm uses the P(word) function to determine the most likely spelling fix. This function assesses word likelihood based on training corpus frequency. The most likely spelling correction for a term is chosen. For pseudocode, see table 7.

Table 7. Norvig Pseudocode

```
function correct(word):
  candidates = (known(edits0(word)) or known(edits1(word)) or
known(edits2(word)) or [word])
  return max(candidates, key=language_model_probability)

function known(words):
  return set(w for w in words if w in language_model)

function edits0(word):
  return {word}

function edits1(word):
  letters    = 'abcdefghijklmnopqrstuvwxyz'
  splits     = [(word[:i], word[i:]) for i in range(len(word) + 1)]
  deletes    = [L + R[1:] for L, R in splits if R]
  transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R)>1]
  replaces   = [L + c + R[1:] for L, R in splits if R for c in letters]
  inserts    = [L + c + R for L, R in splits for c in letters]
  return set(deletes + transposes + replaces + inserts)

function edits2(word):
  return {e2 for e1 in edits1(word) for e2 in edits1(e1)}
```

3. Results and Discussion

Within the course of this study, we analyzed and compared a number of text correction techniques, including Edit Levenshtein, Jaccard, FuzzPartialRatio, FuzzSortRatio, Maximum Likelihood Estimation (MLE), and Norvig. Each of these approaches was evaluated under identical settings,

and their performance was evaluated based on the accuracy and speed of their corrections. To test the algorithms' ability to recognize and rectify faults, they were exposed to a variety of texts with errors that were inserted on purpose. The accuracy of each algorithm's adjustments were then compared to establish the most trustworthy algorithm, with a focus on those that consistently produced correct answers across all sentences. Alongside the accuracy of each algorithm, its speed was also tested. In real-time applications where response speed is critical, this is a crucial factor. The execution duration of each algorithm was meticulously documented, enabling us to determine which of these methods would provide the ideal mix of speed and accuracy.

3.1. Correctness

Every algorithm's accuracy is determined by comparing the ground truth of the sentences with the modified sentences. Upon closer inspection of the data in table 8. except MLE, it becomes clear that the algorithm's performance varies considerably based on the type of text and the nature of the errors inside it. Each strategy appears to handle the issue of vocabulary similarity, particularly when words in incorrect sentences exhibit a similar appearance to their correct counterparts despite their meanings deviating substantially.

Table 8. Correctness result

Edit Distances	Jaccard Index	Fuzz Partial Ratio	fuzz sort ratio	MLE	Norvig result
84%	11%	37%	79%	0%	84%

Taking the Edit Levenshtein algorithm as a case in point, it's clear that it has difficulties dealing with scenarios where the edit distance is high. The 'edit distance' is a measure of the dissimilarity between two strings, computed as the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one word into the other. For instance, in the case of "disassemble the back door" and "disable the back door", the algorithm mistakenly corrects "disassemble" to "disable the lock", reflecting its struggle with the multiple transformations needed to go from the incorrect to the correct sentence. The Jaccard algorithm, FuzzSimpleRatio, FuzzPartialRatio, and FuzzSortRatio display similar challenges. They are each slightly confused by the difficulty of lexically related but semantically dissimilar terms. In the wrong statement "oven the gate," for instance, the algorithms mistakenly adjust "oven" to a range of other words, none of which is the proper "open." This exemplifies the possible dangers of an excessive dependence on lexical similarity, which can lead to problems when the correct and erroneous words have similar spellings but separate meanings. Meanwhile, the MLE (Maximum Likelihood Estimation) algorithm seems to encounter unique challenges. Although it is a powerful method used widely in statistical estimation, its limitations are brought to light in this analysis. In all the tested sentences, it didn't provide a single correct correction, suggesting that it might not be as suited to text correction as the other methods, or it might need additional fine-tuning or adjustments to deal with the unique challenges of this task. Norvig's approach, which is commonly considered as a sophisticated spell correction algorithm, does not emerge as the clear victor either. Although "unlock the back door" is changed appropriately, "turn on the tv" and "open the gate" are not. This demonstrates that even advanced algorithms may be misled by words with similar spelling but distinct meanings, such as "oven" and "open.", for sample result, see table 9.

Table 9. Sample correction result

Ground Truth	User's Query	Corrected by Algorithm					
		Edit	**Jaccard**	**FuzzPartial Ratio**	**FuzzSort Ratio**	**MLE**	**Norvig**
turn on the tv.	turn on the tea.	turn on the tv.	television off the television.	turn television the gate.	turn on the gate.	turn off the tv.	turn on the the.
turn off the light.	turn off the flight.	turn off the light.	television off the fan.	turn off the light.	turn off the light.	ter off the fan.	turn off the light.
lock the door.	look the door.	lock the door.	lamp the desktop.	unlock the door.	lock the door.	laptop the desktopen.	lock the door.
unlock the door.	unluck the door.	unlock the door.	unlock the desktop.	unlock the door.	unlock the door.	unlock the dision.	unlock the door.
turn on the lamp.	turn on the lump.	turn on the lamp.	television off the lamp.	turn television the lamp.	turn on the lamp.	ter open the lock.	turn on the lamp.
lock the windows.	look the windows.	lock the windows.	lamp the windows.	unlock the windows.	lock the windows.	laptop the windoor.	lock the windows.

3.2. Speed

Due to the emphasis on efficiency in computing, speed should be examined to determine whether algorithms are viable for low-cost computers in real-time to achieve a greater user experience while yet obtaining more precision.

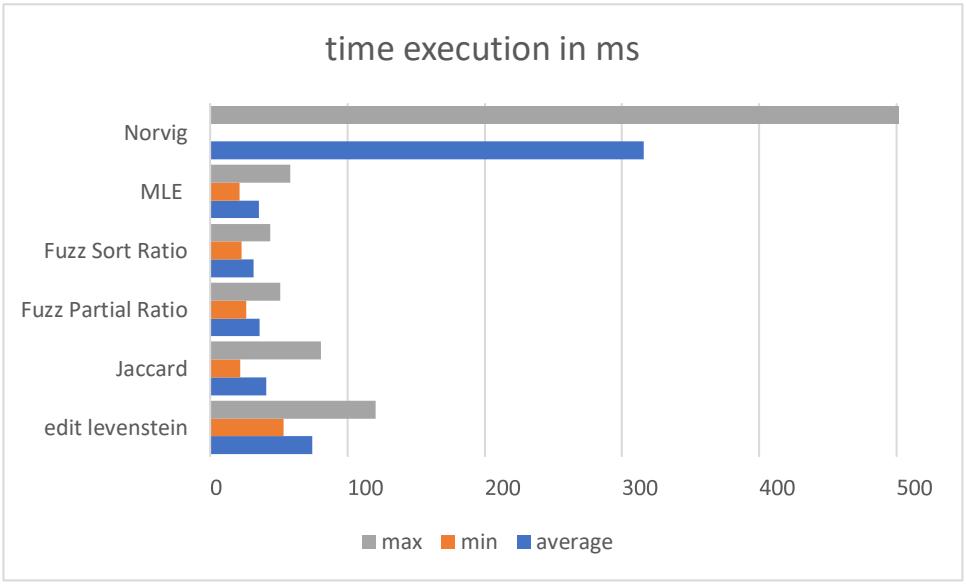

Fig. 5. Algorithm Time Execution

Based on figure 4, we disregard the MLE result as it contains no accurate phrases. However, if we compare this speed test with the accuracy based on table 5, edit Levenshtein emerges as the winner for selecting the fastest and most accurate algorithm. If the goal is to have a quicker method than Edit Levenshtein, but with a lesser expectation of accuracy, the Fuzz Sort Ratio algorithm appears as a possible option. This method has an average execution time of 31.6 milliseconds, making it the quickest of all evaluated algorithms. However, the precision of the adjustments suffers because of this increased speed. Based on Table 5, the Fuzz Sort Ratio algorithm has an accuracy percentage of 79%, which, while high, is somewhat lower than the Edit

Levenshtein algorithm's correctness rate of 84%. This shows that while the Fuzz Sort Ratio approach may provide results more quickly than the Edit Levenshtein algorithm, it may also produce more mistakes.

4. Conclusion

In conclusion, each method for text correction evaluated in this study has its own benefits and drawbacks. The selection of a suitable algorithm is significantly influenced by the unique needs and limits of a given work, with a focus on the relevance of speed and accuracy. The Edit Levenshtein method has the greatest percentage of accuracy (84%), tied with the Norvig algorithm. This makes these two algorithms ideally suited for jobs requiring a high level of precision. However, the average execution time of the Norvig method (315.9 ms) was much greater than that of Edit Levenshtein (74.5 ms), making it less suitable for real-time applications. The method with the quickest average execution time (31.6 ms) was the Fuzz Sort Ratio algorithm. Its accuracy rate (79 %) was somewhat lower than that of Edit Levenshtein but is still substantial. This makes Fuzz Sort Ratio a desirable option in cases when speed is critical, and a small accuracy sacrifice is acceptable.

Examine the following ideas considering the study's limitations and future research. First, this study focused on execution speed and accuracy, but future research may incorporate processor and memory consumption to further assess algorithm performance. Neural network-based text correction techniques might benefit from rapid improvements in machine learning and natural language processing. These methods may use artificial neural networks to enhance speed and accuracy.

Acknowledgment

This research is funded by the Ministry of Education, Republic of Indonesia with funding program Graduate research scheme - research master thesis programme with registration number 140/E5/PG.02.00.PL/2023.

References

[1] A. H. Buckman, M. Mayfield, and S. B. M. Beck, “What is a Smart Building?”, doi: 10.1108/SASBE-01-2014-0003.

[2] F. Siu, D. Chan, D. Clements-Croome, and T. Yang, “Editorial: smart buildings,” *https://doi.org/10.1080/17508975.2021.1867335*, vol. 13, no. 1, pp. 1–3, 2021, doi: 10.1080/17508975.2021.1867335.

[3] J. King and C. Perry, “Smart Buildings: Using Smart Technology to Save Energy in Existing Buildings,” 2017.

[4] A. Ghaffarianhoseini *et al.*, “What is an intelligent building? Analysis of recent interpretations from an international perspective,” *https://doi.org/10.1080/00038628.2015.1079164*, vol. 59, no. 5, pp. 338–357, Sep. 2015, doi: 10.1080/00038628.2015.1079164.

[5] J. King and C. Perry, “Smart Buildings: Using Smart Technology to Save Energy in Existing Buildings,” 2017.

[6] R. Minerva, A. Biru, and D. Rotondi, *Towards a definition of the Internet of Things (IoT)*, Revision 1. IEEE Internet Initiative, 2015. Accessed: Apr. 05, 2023. [Online]. Available: https://iot.ieee.org/images/files/pdf/IEEE_IoT_Towards_Definition_Internet_of_Things_Revision1_27MAY15.pdf

[7] L. S. Vailshery, "IoT connected devices worldwide 2019-2030," Statista. Accessed: Apr. 05, 2023. [Online]. Available: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/

[8] F. Goossens, "Designing a VUI – Voice User Interface." Accessed: Jun. 19, 2020. [Online]. Available: https://www.toptal.com/designers/ui/designing-a-vui

[9] Mokh. S. Hadi, M. A. A. Shidiqi, I. A. E. Zaeni, M. A. Mizar, and M. Irvan, "Voice-Based Monitoring and Control System of Electronic Appliance Using Dialog Flow API Via Google Assistant," in *2019 International Conference on Electrical, Electronics and Information Engineering (ICEEIE)*, IEEE, Oct. 2019, pp. 106–110. doi: 10.1109/ICEEIE47180.2019.8981415.

[10] B. H. Juang and L. R. Rabiner, "Automatic Speech Recognition - A Brief History of the Technology Development," 2004. Accessed: May 03, 2023. [Online]. Available: https://web.ece.ucsb.edu/Faculty/Rabiner/ece259/Reprints/354_LALI-ASRHistory-final-10-8.pdf

[11] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989, doi: 10.1109/5.18626.

[12] H. A. Bourlard and N. Morgan, "Connectionist Speech Recognition," *Connectionist Speech Recognition*, 1994, doi: 10.1007/978-1-4615-3210-1.

[13] X. Huang, J. Baker, and R. Reddy, "A historical perspective of speech recognition," *Commun ACM*, vol. 57, no. 1, pp. 94–103, Jan. 2014, doi: 10.1145/2500887.

[14] B. Shillingford *et al.*, "Large-Scale Visual Speech Recognition," *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, vol. 2019-September, pp. 4135–4139, Jul. 2018, doi: 10.21437/Interspeech.2019-1669.

[15] M. Pleva, J. Juhar, and A. S. Thiessen, "Automatic Acoustic Speech segmentation in Praat using cloud based ASR," *Proceedings of 25th International Conference Radioelektronika, RADIOELEKTRONIKA 2015*, pp. 172–175, Jun. 2015, doi: 10.1109/RADIOELEK.2015.7129000.

[16] S. Ghosh and O. Kristensson, "Neural Networks for Text Correction and Completion in Keyboard Decoding", Accessed: Oct. 20, 2023. [Online]. Available: http://luululu.com/tweet/

[17] T. Ke and K. Sudhir, "Privacy Rights and Data Security: GDPR and Personal Data Driven Markets," *SSRN Electronic Journal*, 2020, doi: 10.2139/ssrn.3643979.

[18] A. Hard *et al.*, "FEDERATED LEARNING FOR MOBILE KEYBOARD PREDICTION", Accessed: Oct. 20, 2023. [Online]. Available: https://www.tensorflow.org/lite/

[19] Y. Sun, J. Zhang, Y. Xiong, and G. Zhu, "Data Security and Privacy in Cloud Computing," *Int J Distrib Sens Netw*, vol. 10, no. 7, p. 190903, Jul. 2014, doi: 10.1155/2014/190903.

[20] J. Jo, J. Kung, S. Lee, and Y. Lee, "Similarity-Based LSTM Architecture for Energy-Efficient Edge-Level Speech Recognition," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, IEEE, Jul. 2019, pp. 1–6. doi: 10.1109/ISLPED.2019.8824862.

[21] E. Kalbaliyev and S. Rustamov, "Text Similarity Detection Using Machine Learning Algorithms with Character-Based Similarity Measures," 2021, pp. 11–19. doi: 10.1007/978-3-030-74728-2_2.

[22] N. Gahman and V. Elangovan, "A Comparison of Document Similarity Algorithms," Apr. 2023.

[23] D. D. Prasetya, A. Prasetya Wibawa, and T. Hirashima, "The performance of text similarity algorithms," *International Journal of Advances in Intelligent Informatics*, vol. 4, no. 1, p. 63, Mar. 2018, doi: 10.26555/ijain.v4i1.152.

[24] W. H.Gomaa and A. A. Fahmy, "A Survey of Text Similarity Approaches," *Int J Comput Appl*, vol. 68, no. 13, pp. 13–18, Apr. 2013, doi: 10.5120/11638-7118.

[25] Q. Li, J. Song, J. Ning, and J. Yuan, "The Detailed Data on the Neural Compute Stick Acceleration Performance," *Proceedings - 2019 Chinese Automation Congress, CAC 2019*, pp. 4959–4962, Nov. 2019, doi: 10.1109/CAC48633.2019.8996841.

[26] J. Lu, C. Lin, W. Wang, C. Li, and H. Wang, "String similarity measures and joins with synonyms," in *Proceedings of the 2013 international conference on Management of data - SIGMOD '13*, New York, New York, USA: ACM Press, 2013, p. 373. doi: 10.1145/2463676.2465313.

[27] M. Benard Magara, S. O. Ojo, and T. Zuva, "A comparative analysis of text similarity measures and algorithms in research paper recommender systems," in *2018 Conference on Information Communications Technology and Society (ICTAS)*, IEEE, Mar. 2018, pp. 1–5. doi: 10.1109/ICTAS.2018.8368766.

[28] Y. S. Han, S. K. Ko, T. Ng, and K. Salomaa, "Closest substring problems for regular languages," *Theor Comput Sci*, vol. 862, pp. 144–154, Mar. 2021, doi: 10.1016/J.TCS.2020.09.005.

[29] S. Konstantinidis, "Computing the edit distance of a regular language," *Inf Comput*, vol. 205, no. 9, pp. 1307–1316, Sep. 2007, doi: 10.1016/J.IC.2007.06.001.

[30] "(PDF) Modifying Jaccard Coefficient for Texts Similarity." Accessed: Nov. 06, 2023. [Online]. Available: https://www.researchgate.net/publication/340267006_Modifying_Jaccard_Coefficient_for_Texts_Similarity

[31] İ. Kabasakal and H. Soyuer, "A Jaccard Similarity-Based Model to Match Stakeholders for Collaboration in an Industry-Driven Portal," *Proceedings 2021, Vol. 74, Page 15*, vol. 74, no. 1, p. 15, Mar. 2021, doi: 10.3390/PROCEEDINGS2021074015.

[32] P. J. Rao, K. N. Rao, and S. Gokuruboyina, "An Experimental Study with Fuzzy-Wuzzy (Partial Ratio) for Identifying the Similarity between English and French Languages for Plagiarism Detection," *International Journal of Advanced Computer Science and Applications*, vol. 13, no. 10, pp. 393–401, 2022, doi: 10.14569/IJACSA.2022.0131047.

[33] G. A. Rao, G. Srinivas, K. V. Rao, and P. V. G. D. P. Reddy, "G APPA RAO, et al.: A PARTIAL RATIO AND RATIO BASED FUZZY-WUZZY PROCEDURE FOR CHARACTERISTIC MINING OF MATHEMATICAL FORMULAS FROM DOCUMENTS A PARTIAL RATIO AND RATIO BASED FUZZY-WUZZY

PROCEDURE FOR CHARACTERISTIC MINING OF MATHEMATICAL FORMULAS FROM DOCUMENTS", doi: 10.21917/ijsc.2018.0242.

[34] Y. Song *et al.*, "Improving Maximum Likelihood Training for Text Generation with Density Ratio Estimation," 2020.

[35] Y. Kantor *et al.*, "Learning to combine Grammatical Error Corrections," *ACL 2019 - Innovative Use of NLP for Building Educational Applications, BEA 2019 - Proceedings of the 14th Workshop*, pp. 139–148, Jun. 2019, doi: 10.18653/v1/w19-4414.

[36] N. Zhao *et al.*, "Large-Scale Analysis of Docker Images and Performance Implications for Container Storage Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 4, pp. 918–930, Apr. 2021, doi: 10.1109/TPDS.2020.3034517.

[37] C. S. Park and H. M. Nam, "Security Architecture and Protocols for Secure MQTT-SN," *IEEE Access*, vol. 8, pp. 226422–226436, 2020, doi: 10.1109/ACCESS.2020.3045441.

[38] M. Wang, S. Qiu, H. Dong, and Y. Wang, "Design an IoT-based building management cloud platform for green buildings," *Proceedings - 2017 Chinese Automation Congress, CAC 2017*, vol. 2017-January, pp. 5663–5667, Dec. 2017, doi: 10.1109/CAC.2017.8243793.

[39] T. Bocklisch, J. Faulkner, N. Pawlowski, and A. Nichol, "Rasa: Open Source Language Understanding and Dialogue Management," Dec. 2017, [Online]. Available: http://arxiv.org/abs/1712.05181

[40] A. Krasnov, R. R. Maiti, and D. M. Wilborne, "Data Storage Security in Docker," *Conference Proceedings - IEEE SOUTHEASTCON*, vol. 2020-March, Mar. 2020, doi: 10.1109/SOUTHEASTCON44009.2020.9249757.

[41] S. Aouragh, H. Gueddah, and A. Yousfi, "Adaptating the Levenshtein Distance to Contextual Spelling Correction," *International Journal of Computer Science & Applications*, vol. 12, pp. 127–133, May 2015.

[42] T. Anjali, T. R. Krishnaprasad, and P. Jayakumar, "A Novel Sentiment Classification of Product Reviews using Levenshtein Distance," in *2020 International Conference on Communication and Signal Processing (ICCSP)*, IEEE, Jul. 2020, pp. 0507–0511. doi: 10.1109/ICCSP48568.2020.9182198.

[43] L. S. Riza, F. Syaiful Anwar, E. F. Rahman, C. U. Abdullah, and S. Nazir, "Natural Language Processing and Levenshtein Distance for Generating Error Identification Typed Questions on TOEFL Journal of Computers for Society," 2020.

[44] V. ~I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, Feb. 1966.

[45] K. Rinartha and W. Suryasa, "Comparative study for better result on query suggestion of article searching with MySQL pattern matching and Jaccard similarity," *2017 5th International Conference on Cyber and IT Service Management, CITSM 2017*, Oct. 2017, doi: 10.1109/CITSM.2017.8089237.

[46] A. Carlson and I. Fette, "Memory-Based Context-Sensitive Spelling Correction at Web Scale."

[47] C. Telvis, "Using text analysis techniques to build a predictive text model." Accessed: May 22, 2023. [Online]. Available: https://rpubs.com/telvis/capstone_report_1

[48] M. Näther, "An In-Depth Comparison of 14 Spelling Correction Tools on a Common Benchmark," in *Proceedings of the Twelfth Language Resources and Evaluation Conference*, N. Calzolari, F. Béchet, P. Blache, K. Choukri, C. Cieri, T. Declerck, S. Goggi, H. Isahara, B. Maegaard, J. Mariani, H. Mazo, A. Moreno, J. Odijk, and S. Piperidis, Eds., Marseille, France: European Language Resources Association, May 2020, pp. 1849–1857. [Online]. Available: https://aclanthology.org/2020.lrec-1.228

[49] M. N. Samsuri, A. Yuliawati, and I. Alfina, "A Comparison of Distributed, PAM, and Trie Data Structure Dictionaries in Automatic Spelling Correction for Indonesian Formal Text," in *2022 5th International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*, IEEE, Dec. 2022, pp. 525–530. doi: 10.1109/ISRITI56927.2022.10053062.