IIAI Open Conference Publication Series
IIAI Letters on Institutional Research
Vol. 005, LIR414
DOI: https://doi.org/10.52731/lir.v005.414

Japanese Tokenization using Simple Associative Arrays

Michiko Yasukawa [*] , Koichi Yamazaki [†]

Abstract

This study addresses a technical challenge in text analysis for the practice of Institutional Research (IR). In the research field of Natural Language Processing (NLP), development focuses on accuracy and speed. Therefore, suitable data for evaluation and efficient programming languages are used. On the other hand, tasks outside the scope of NLP are also important in IR. For example, not only textual data, but also numerical data must be analyzed, and in some cases, the analysis of data stored in web servers is required. Since NLP and IR involve different tasks, the tools required are also different. One serious problem for text analysis in IR tasks is the limited development of Japanese tokenization. In this study, we implement a Japanese tokenizer, focusing on its applicability in web servers. Since function libraries in scripting languages for web environments are limited, our main technical challenge is to achieve efficiency in tokenization by combining the available functions. To deal with this problem, our method uses associative arrays to reduce unnecessary steps in tokenization. Evaluation experiments were conducted using Japanese text data, and the results showed that our tokenizer is viable in web server environments.

Keywords: bibliometrics, management of research projects, open science, text mining

1 Introduction

Recently, generative AI has been actively developed and applied in various fields. Developing generative AI requires data to train the models. Amid this trend, it has been decided that scientific papers and data resulting from publicly funded research in Japan will be made available through open access[1]. Additionally, Japanese models of generative AI [1] have been developed and made available by The National Institute of Informatics (NII).

The use of generative AI is expected to make Institutional Research (IR) more efficient and meaningful. However, as AI that can make human-equivalent decisions has not yet been realized, human experts with specialized knowledge will still be required to carry out the work in IR. It is widely known that the output of generative AI may contain errors, oversights, contradictions, and misleading expressions [2]. IR specialists must use human intelligence, common sense, and ethics to perform their tasks, without being misled by

[*] Gunma University, Gunma, Japan
[†] Tokyo Denki University, Tokyo, Japan
[1] `https://current.ndl.go.jp/car/222611`

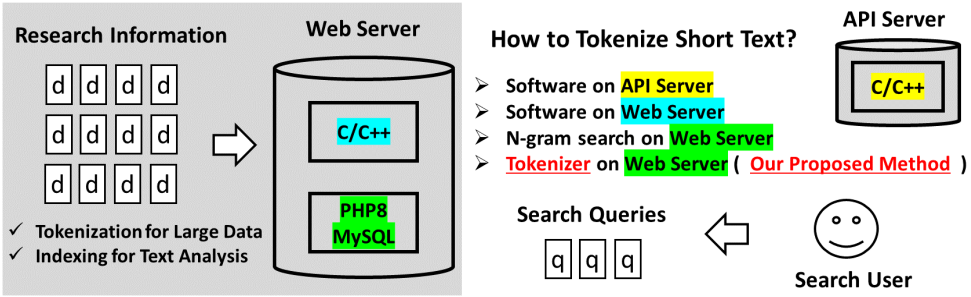


Figure 1: Tokenization for empowering institutional research

the errors of generative AI. For such purposes, IR specialists must identify reliable data so that they can verify fact and truth. And, computer assistance should be helpful in this verification.

At universities in Japan, text data almost always includes the Japanese language. For example, the analysis of research and educational information [3] [4] [5] involves Japanese text data. One serious problem here is that the currently available Japanese tokenization tools are not sufficiently developed from a practical standpoint of IR tasks. Unlike English, Japanese text data do not have spaces between words. Hence, in analyzing text data, tokenization for segmenting sentences into words is essential. When searching text data on a web server (for sharing data among multiple people, or searching from any location), the text data are tokenized, uploaded to the server's database, and indexed using a database function. However, search queries cannot be tokenized in advance. The tokenization process is performed each time a search user enters a search query. Currently, there are three options for tokenizing search queries, as shown in Figure 1.

1. Use an API server (e.g. WebAPI by Yahoo[2]).
2. Install a tokenizer (e.g. MeCab[3]) on the web server.
3. Tokenize queries by character N-grams.

Among these three options, the character N-gram is the easiest to implement because it can be generated by a short piece of code in the scripting language on the web server. There is no need to create an account on an external API server or install C/C++ software with an administrator account on web servers. However, character N-gram search has a fatal problem with poor search performance. It is known that character N-gram has very low accuracy in Japanese when the text is separated into single characters (Uni-gram). And, when the text is separated in two characters (Bi-gram), it becomes impossible to search by one character query.

In this study, we will develop a tokenization tool that can be easily deploy, like character N-gram. To be more specific, when searching Japanese text data, the only thing to do is upload the files, which include the code for tokenization, onto the web server. Then, IR specialists start quick search without worrying about installation issues related to C/C++ software, or without reading instruction guides for an API server. Now that highly advanced AI tools are easily accessible through just a web browser, improving the usability of basic

[2] `https://developer.yahoo.co.jp/webapi/jlp/ma/v2/parse.html`
[3] `https://taku910.github.io/mecab/`

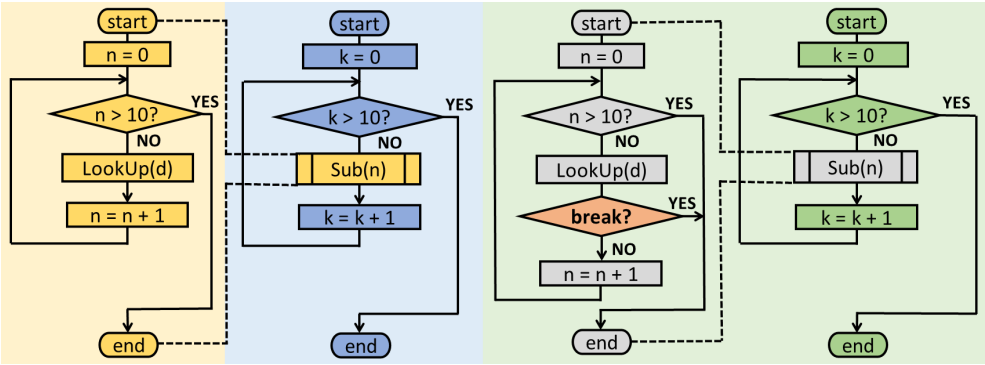


Figure 2: Nested loops and break from the loop

NLP tools for IR tasks is indispensable. While AI tools can be helpful, reliable work will not be possible without human verification of the actual data for IR tasks.

The detailed technical issues to be resolved in this study are explained as follows. In this study, we specifically devise a method for efficient dictionary lookup using PHP's standard libraries. In text analysis on a local PC, rather than a web server, Python is commonly used for coding. For such purposes, MeCab's Python interface[4] is provided, and the Python-native morphological analyzer Janome[5] is also provided. On the other hand, in web development, PHP is more widely used than Python. According to w3tech,[6] PHP is used by 74.5% of all the websites. However, PHP-native Japanese tokenization tools have not been developed so far. There is "php-mecab"[7] that executes MeCab from PHP. As it is a PHP interface and not a native implementation, MeCab must be installed on the server when using this interface. MeCab is implemented in C/C++. And, PHP has a similar syntax with C/C++. Thanks to this similarity, necessary algorithms in MeCab, such as Viterbi algorithm [6], can be easily reimplemented in PHP. However, PHP does not have function libraries for efficient dictionary lookup. To be more specific, MeCab uses a particular data structure, called "double array" [7] for dictionary lookup. Double array enables scalability, allowing for high-speed processing even when the input data becomes large. PHP is a scripting language, and memory access, etc. are fundamentally different from C/C++, so such data structures written in C/C++ cannot be reimplemented by simple code porting.

2 Method

Japanese tokenization is performed through two main processes: (1) looking up tokens in a token dictionary, and (2) selecting the most optimal token from the obtained tokens. The problem to be solved in this study is to reduce unnecessary lookup steps and improve the efficiency of the tokenization. To be more specific, the break from loop (as shown in Figure 2) is inserted in the repeated lookup steps. This strategy is explained in detail below.

[4] `https://taku910.github.io/mecab/bindings.html`
[5] `https://janome.mocobeta.dev/en/`
[6] `https://w3techs.com/technologies/details/pl-php`
[7] `https://github.com/ranvis/php-mecab`

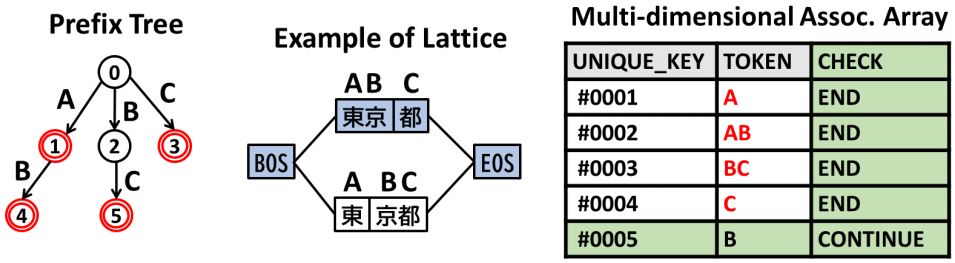


Figure 3: How to implement Japanese tokenization using associative arrays.

2.1 Dictionary Look-up

First, let us examine the yellow flowchart on the left of Figure 2. In this process, LookUp(d), which is a dictionary lookup function, is executed 10 times. This corresponds to the case, for example, where there are 10 tokens in one piece of text data, and the dictionary lookup (checking whether the token is in the dictionary) is executed 10 times. If we assume that LookUp(d) takes 1 ms per execution, it takes at least 10 ms from the start to the end of the yellow flowchart. Next, let us examine the blue flowchart. Here, the yellow flowchart is used as the predefined process, Sub(n). The repeated process of the blue flowchart executes Sub(n) 10 times. Therefore, LookUp(d) is executed 100 times in total. If each execution takes 1 ms, it takes at least 100 ms from the start to the end of the blue flowchart. This situation corresponds to the case, for example, where there are 10 tokens in one piece of text data, and there are 10 such pieces of text data. The green flowchart is basically the same as the blue flowchart, but it includes a break in the process. For this reason, the processing time for the green flowchart varies depending on the break condition. In the worst case, Sub(n) is executed 100 times, whereas in the best case, Sub(n) is executed 10 times. In Figure 2, k corresponds to the number of characters in the input text. If the value of k is large, the number of dictionary lookups will be gigantic. Therefore, it is essential to eliminate unnecessary dictionary lookups.

2.2 Lattice and Prefix Tree

Japanese tokenization itself is a very simple process. It segments an input string (such as a user's search query) and converts it into a sequence of tokens. For example, given the input string ABC, possible tokenization candidates include AB/C and A/BC, and the most optimal sequence of the tokens is obtained. In Figure 3, the probabilistic calculation indicates that AB/C is more optimal than A/BC. For this calculation, a dictionary of tokens that has been created in advance is used.[8]

To select the best sequence of tokens from the candidates, a graph structure called a lattice [8], which contains nodes and edges, is used. The lattice structure is illustrated at the center of Figure 3, where the squares represent nodes, and the lines between the nodes are edges. The lattice for the example input string ABC is constructed as follows. Take a look at the first character of the input string ABC. It is A. What are the tokens whose first character starts with A? Examining the red letters in Figure 3, we can see that there

[8]Parenthetically, the dictionary includes the Part of Speech (POS) and Katakana reading of each token.

are A and AB. The nodes for A and AB are connected after the special node BOS, which represents the beginning of a sentence. Then, move on to the next. What is the token that starts with the second character of the input, B? We can see that it is BC. Connect the node for BC after the node ending with A. What is the token that starts with the third character of the input, C? We can see that it is C. Add a node for C after the node ending with B. As there are no more characters, connect EOS, which indicates the end of the sentence. In the token dictionary, the probability of token occurrence (emission cost) and the probability of connection between tokens (connection cost) are defined for each token. Using this token information, the most likely path between BOS and EOS is obtained by calculation. This cost calculation simply involves adding the numerical values defined in the dictionary.[9]

Thus, tokenization can be viewed as a combination of the following processes: (a) dictionary lookup, (b) creating a lattice, (c) cost calculation, and (d) optimal path selection. These processes are simple enough, but since there are a huge number of node and path candidates, a brute-force method will instantly fail. Hence, efficiency is necessary. This is the difficult point in implementation of tokenization. To understand more about a simple brute-force method, let us look at some examples explained as follows. How many dictionary lookups are performed in a nested loop for the substrings of ABC? The answer is: six dictionary lookups are performed for A, AB, ABC, B, BC, and C. In the case of ABCDE, the dictionary lookups are performed for A, AB, ABC, ABCD, ABCDE, B, BC, BCD, BCDE, C, CD, CDE, D, DE, E. So, there are 15 dictionary lookups. As observed, when the number of characters is k, and k is one or more, there are $(k*(k+1))/2$ dictionary lookups. Note that, if the number of characters is small enough, efficiency is not an issue. For example, if the number of character is one (e.g., lookup for the string, "A"), dictionary lookup is conducted only once. However, if the number of input characters is even slightly higher, the number of substrings becomes non-negligible, and the unnecessary dictionary lookups must be avoided.

One solution to this problem of the time-consuming lookup is to use a tree structure, as shown on the left in Figure 3, called a Prefix Tree [8]. In this tree structure, the circles represent vertices. The arrows represent branches, and the branches are associated with the characters. The red double circle indicates the end of the token. The number 0 represents the root of the tree. By tracing the vertices from the root, we identify four tokens: A, AB, BC, and C. As can be seen, this tree represents the four tokens shown in the red letters in Figure 3. The use of this tree structure makes the dictionary lookups efficient. Let us consider an example of the tree search, where the input string is ABC. What is the token that starts with A? The answer is: A, AB. (The number 0, 1, and 4. This is the end of search.) What is the token that starts with B? (The number 0, 2, and 5. There is no red double circle for the number 2, so B is skipped.) The answer is BC. (This is the end of search.) What is the token that starts with C? The answer is C. (The number 0 and 3. This is the end of search.) And, there is nothing even if we try to search further.

Meanwhile, let us recall the previous example, where the input is the string ABCDE and the dictionary lookup is conducted in a brute-force search. We can conduct a string search on the tokens shown in the red letters in Figure 3 and check whether A is there, whether AB is there, whether ABC is there, whether ABCD is there, whether ABCDE is there. It should be noted that in a tree structure, the search enumerates the tokens that start with A, and we obtained A and AB. No further unnecessary searches are performed on

[9]Note that a high cost means that it is unlikely to be Japanese, so the path with the lowest cost is output. For the efficiency of the cost calculation in large data, Viterbi algorithm is utilized in MeCab.

the tree. Efficient tokenizers use this kind of ingenious data structures (fast lookup, and concise in memory), such as double arrays for implementation. Note that building such a complex dictionary takes time. Therefore, dictionaries are built for the tokenizers (e.g., MeCab) during installation. The binary-format dictionary is opened when tokenization is executed and is ready for searching. For script languages in web environments (e.g., PHP), however, an alternative method is necessary since accessing arbitrary binary files is not recommended.

2.3 Associative Array for Efficiency

When sharing data over a web interface safely, standard text formats, such as JSON, are recommended. Following this recommendation, our proposed method uses dictionary data expressed as text data rather than tree-structured binary data. These dictionary files in the text format are opened at runtime and loaded into the memory without harming the server computers. A detailed explanation of this process is provided below.

The proposed method uses simple associative arrays. Specifically, the green part in Figure 3 is added to determine if a string is (1) the end of a token, or (2) a part of a token. This condition allows the break from the loop (as shown in Figure 2) and the search to be terminated for strings that are neither the end nor a part of a token.

A concrete example is as follows. For the string ABCDE, if we search for A and AB, we will find them. If we look up ABC, it is not in the token dictionary. Since ABC is neither the end nor a part of a token, the loop of the dictionary lookup exits (this corresponds to the break in the green flowchart in Figure 2). Then, ABCD and ABCDE are not be searched.

For the implementation of this dictionary lookup, the proposed method uses a multidimensional associative array, instead of a tree structure. The associative relationships between the columns for KEY and TOKEN, and the columns for KEY and CHECK in Figure 3 achieve this efficiency. When considering actual coding, a large multidimensional array is inefficient; therefore, a single multidimensional associative array is decomposed into multiple one-dimensional associative arrays. This table decomposition contributes to the (1) faster dictionary opening, (2) faster token search, and (3) smaller dictionary size. In actual coding, multiple duplicated tokens are grouped together and assigned a unique KEY. Subsequently, an associative array is created to associate the unique KEY with the representative token. Finally, a one-dimensional array of sequential numbers is created and associated with the array of unique KEYs. Because the serial numbers are equivalent to the array index numbers, the decomposed one-dimensional arrays can be represented in CSV format instead of JSON format, greatly speeding up file opening.[10]

3 Experiment

We implemented the proposed method in PHP8 to develop a tokenizer.[11] Since our tokenizer is a reimplementation of MeCab, the same dictionary (IPADIC) [9] as MeCab was used. For the experiment, a desktop PC (Ubuntu 20.04, Intel Core i9-9820X CPU @ 3.30GHz, 256 GB RAM) was used.

[10] Note that JSON decoding is a much heavier operation than loading CSV files onto the memory.
[11] Specifically, we used PHP 8.1.32.

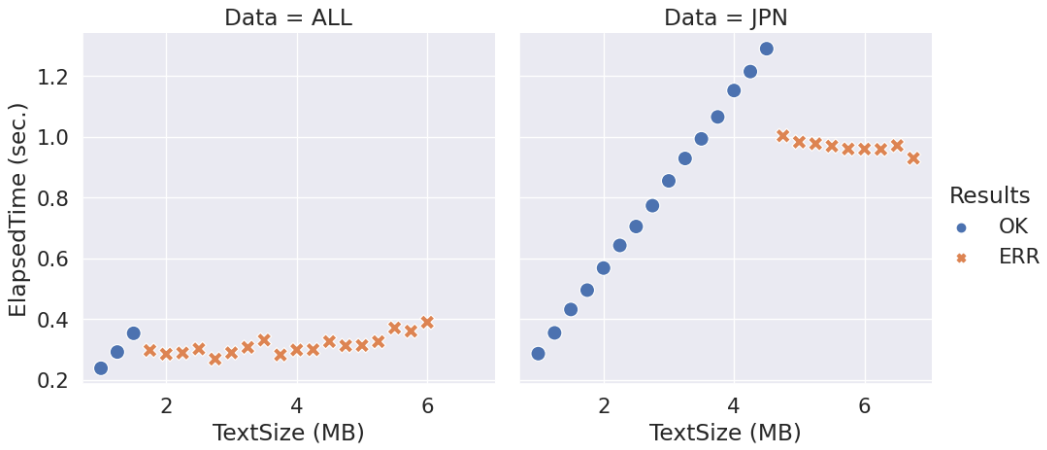


Figure 4: With large text data, what happens in tokenization?

3.1 Dataset and Comparison Method

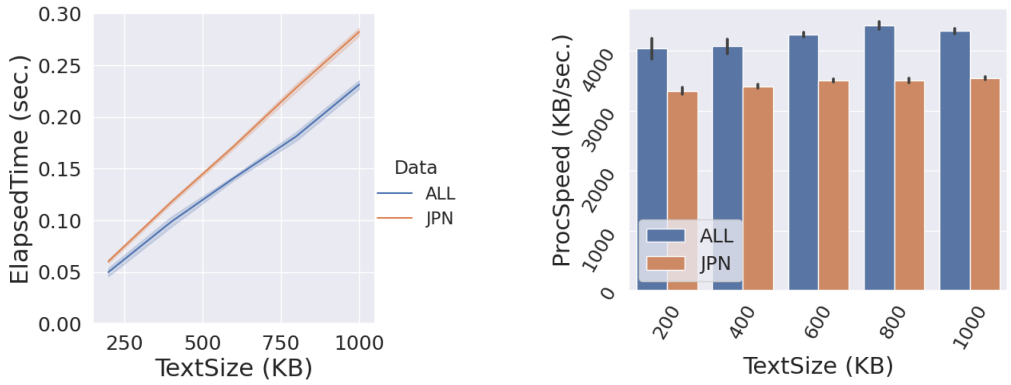
To prepare experimental text data, we extracted text content from the PDF files of research papers at annual meetings of NLP.[12] The extracted text content was mixture of multilingual text data including Japanese, English, etc. We refer this text data as ALL. Then, we created another text data, where all alphanumeric characters and symbols were deleted, and only characters for Japanese text were retained. We refer this text data as JPN.

As a comparison method, php-mecab was used. It is an interface for calling MeCab from PHP scripts. By using the comparison method, the normality of the experiments is verified. To be more specific, if the output of MeCab and the output of proposed method are the same, then the proposed method works properly. Since our tokenizer is a reimplementation of MeCab, it cannot do what MeCab cannot do. In addition, the proposed method is less scalable than MeCab, so the experimental data must be smaller than what MeCab can achieve. To correctly set the experimental conditions, we confirmed the limitation of MeCab. There are two upper limits for input text:

1. Buffer overflow warning.
   This is a soft limit, and the limit can be modified with the -b #SIZE option when executing. As the default setting was only 8,192 bytes, the SIZE was set larger size than the size of each dataset in the experiment.
2. Errors of overflow in the INTEGER type variable for the cost calculation.
   This is a hard limit, and if MeCab reaches this limit, the process terminates, and no execution results are obtained. Instead, an error message is displayed. Note that this error can be resolved by splitting the input text into shorter pieces of text.

Using the aforementioned experimental data, we observed the behavior of MeCab when the input text size was very large. Specifically, we confirmed the MB scale of the input text. The results are presented in Figure 4. The X-axis in the figure represents the size of the data. The Y-axis in the figure represents the elapsed time. The orange crosses indicate cases where no results were obtained due to the fatal overflow error. The blue circles indicate

[12] https://anlp.jp/guide/nenji.html



(a) Processing time (b) Processing speed

Figure 5: Performance analysis of the comparison method (MeCab)

normal completion. At this data scale, errors occurred due to the overflow in the cost calculation for both ALL (multiple languages) and JPN (Japanese language). The left-hand figure shows the results for ALL, while the right-hand figure shows the results for JPN. As can be seen, the fatal errors started occurring at 1–2MB for ALL, and 4–5MB for JPN.

Since we found that errors occurred at the MB scale, we performed experiments on smaller text sizes at the KB scale. We created 100 pieces of text data for each of the five data sizes (200 KB, 400 KB, 600 KB, 800 KB, 1000 KB). Then, the elapsed time and processing speed of the tokenization were measured. The results are shown in Figure 5. At the KB scale, no overflow errors were observed. The processing time of MeCab was sufficiently short. In addition, the processing speed remained stable with increasing text size. To ensure the validity of the evaluation experiments, we used the text data that were confirmed to work properly with MeCab.

3.2 Baseline vs. Proposed

Our tokenization tool is designed to process search queries on a web server. Thus, we need to consider the size of data that can be handled between the web server and client. When search users conduct quick web searches, they do not manually input tens of thousands of characters as a search query on the web client side. Moreover, there is an upper limit to the amount of text data that each web browser can send to the web servers. For example, Internet Explorer, which is no longer used, had the maximum character length of 2,083 for URLs,[13] and it was referenced as one of the standards in web environments. Currently, different types of web browsers are used, and the upper limits are varied. Meanwhile, web servers have upper limits, too. They do not accept unlimited input to prevent excessive workload and DoS attacks. For example, Google's Apps Script[14] limits the number of characters in a URL to 2,082 bytes.

Using this numerical value as a reference, we conducted experiments with input character counts ranging from 1 KB to 2.5 KB. In the experiments, the baseline method is the simplest and least efficient method introduced as basic knowledge by Kudo [8]. This

[13] https://support.microsoft.com/en-us/topic/maximum-url-length-is-2-083-characters-in-internet-explorer-174e7c8a-6666-f4e0-6fd6-908b53c12246

[14] https://developers.google.com/apps-script/reference/url-fetch/url-fetch-app

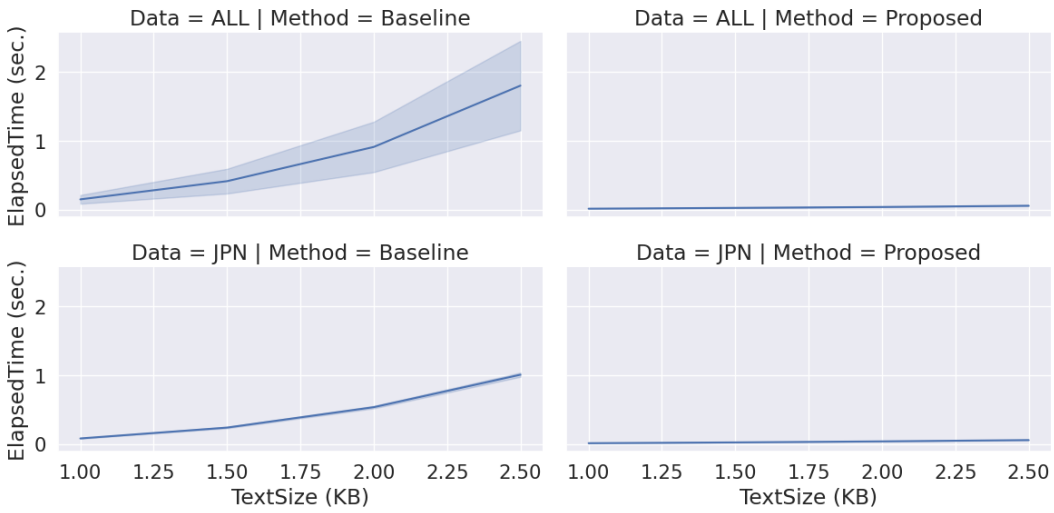
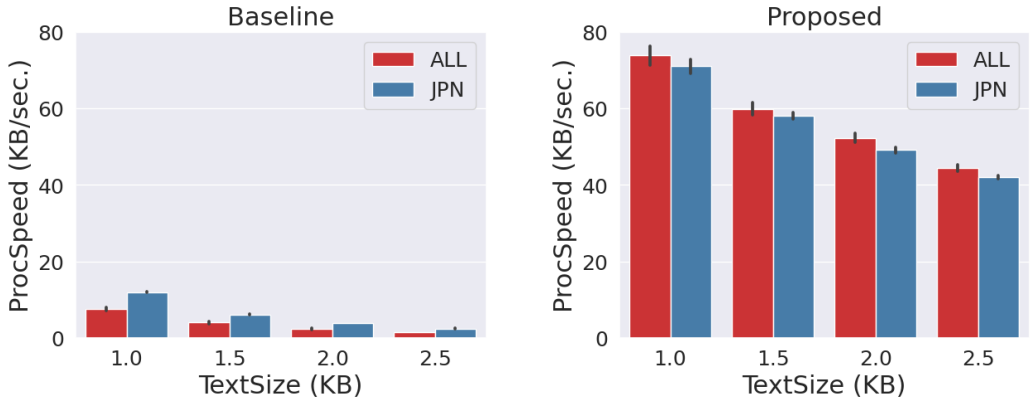


Figure 6: Processing time of the baseline and proposed methods



(a) Baseline (b) Proposed

Figure 7: Processing speed by text size

method is implemented only with a token dictionary (hash map), as shown in the white cells in Figure 3. The proposed method is implemented with a token dictionary that is extended by associative arrays, as shown in the white and green cells in Figure 3. The obtained results are shown in Figure 6. The figure shows the elapsed time for the data size in KB. The processing time of the baseline method increases explosively as the data size increases. This is because the loop in the baseline method does not include a break to terminate the dictionary lookup. On the other hand, the proposed method does not experience a sudden increase in processing time. Next, we compare the speeds of the baseline and proposed methods. The results are shown in Figure 7, which presents the processing capacity (the data size in KB) per second. Figure 7 (a) and (b) show that the proposed method has a higher tokenization processing capacity than the baseline method. Both the baseline and the proposed methods perform less well than the comparison method (MeCab), as shown in Figure 5(b) and Figure 7. As observed in Figure 7, the speed gradually decreases as the

Table 1: Average processing time in seconds for tokenization (2.5 KB datasets)

Data	Comparison			Baseline			Proposed		
	INIT	MAIN	TOTAL	INIT	MAIN	TOTAL	INIT	MAIN	TOTAL
ALL	0.0002	0.0021	0.0023	0.3229	1.8031	2.1259	0.3699	0.0572	0.4271
JPN	0.0002	0.0026	0.0028	0.3229	1.0066	1.3295	0.3699	0.0600	0.4299

number of input characters increases, for both the baseline and proposed methods. Next, we compare the processing times of the comparison, baseline, and proposed methods in detail when the number of input characters is large.

Table 1 shows the average processing time for 2.5 KB datasets. In the table, INIT indicates the initialization time, which is the time between the launch of the program and opening of the dictionary. MAIN indicates the processing time for tokenization after the dictionary opening. TOTAL indicates the total time from the program launch and the end of tokenization. The results are shown for the two experimental datasets, ALL (multiple languages) and JPN (Japanese language). The comparison method is php-mecab (MeCab using the PHP interface). The dictionary sizes for the comparison, baseline, and proposed methods are 51MB, 66MB, and 67MB, respectively. The comparison method (MeCab) had the smallest dictionary size and the fastest tokenization process. The proposed method had an expanded dictionary to speed up the tokenization process, so the dictionary size was larger than that of the baseline method. In addition, the proposed method took more time for the initialization process than the baseline method. However, the tokenization process of the proposed method was faster than the baseline method. The reason for this was the additional association arrays for efficient token search. MeCab only opens a dictionary that has been built in advance. Thus, its initialization takes almost no time. Note that the time MeCab took to build the dictionary at installation was 3.567 seconds, which was longer than the baseline and shorter than the proposed method.

From the perspective of website usability,[15] there are three main time limits [10]. If the response time is 0.1 second, the user feel that the system is reacting instantaneously. MeCab meets this standard. If the response time is more than 0.1 second but less than 1.0 second, the user will sense the delay, but no special feedback from websites is needed. From a response time of 1–10 seconds, users should be given some feedback, such as a progress indicator. The baseline method is too slow to operate on websites without the progress indicator. In contrast, the proposed method can be operated with a response time less than 1.0 second, and is more viable on websites than the baseline method.[16]

Now, our question is: why was the proposed method faster than the baseline method, but slower than the comparison method? This argument can be rationalized as follows. For this investigation, we conducted an analysis on tokens in the tokenizer dictionary. Number of characters and frequency of the IPADIC dictionary are shown in Figure 8. Regarding the character number, the average and median were 3.5 and 3, respectively. Although some tokens had a large number of characters, most were short. While the baseline method does not include break in the loop, the proposed method quickly terminates when the token is short because of the break in the lookup dictionary steps. This strategy contributes to making the

[15] `https://www.nngroup.com/articles/website-response-times/`

[16] In addition to program execution time, web server operations generally take several to tens of milliseconds, depending on the network speed.

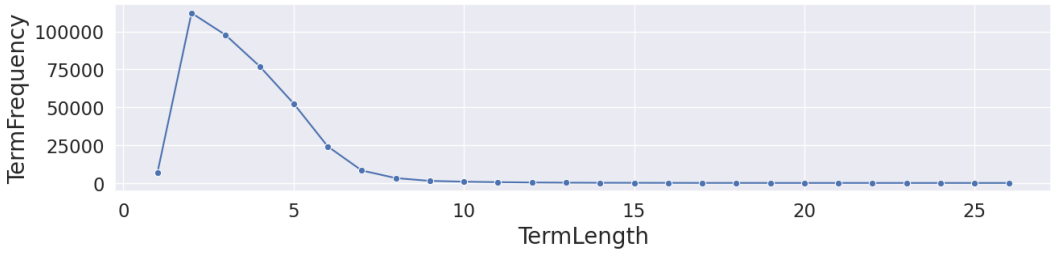


Figure 8: Length and frequency of the terms in mecab-ipadic

tokenizer efficient. However, the proposed method was not as efficient as MeCab. MeCab utilizes double arrays, which is sufficiently efficient, enabling $O(N)$. The proposed method uses associative arrays in the loop, which enables $O(KN)$. The baseline method includes a nested loop, which results in $O(N^2)$. Owing to these differences in computational costs, the three methods differ in terms of speed.[17] It should be noted that the development of MeCab started more than 20 years ago.[18] Nowadays the hardware and software of computers have advanced dramatically. Thanks to this advancement, our proposed method, which uses a scripting language and is not as super efficient as MeCab, has been able to realize a practical level of processing speed in the web environment.[19] This reality is noteworthy for IR practitioners and web application developers.

4 Concluding Remarks

We have developed a tokenization tool that is implemented natively in PHP. It is easy to use by simply uploading a set of files to the web server. Thus, there are no compilation errors or complicated initial settings before using it. Our tool can be used to analyze text data for research and education at universities.

Previous research on the proposed method includes the hash-coding and tree-search algorithm, which was proposed by Burkhard in 1976 [13]. Our method, which uses simple associative arrays, has limited scalability; however, it has the advantage of being able to be implemented with a combination of standard functions in the scripting language. Our target problem is tokenization that can be applied to small-scale data, such as user search queries. Therefore, the proposed method is suitable for solving our target problem. In the field of NLP research, efficiency for tokenization of large-scale Japanese text data is realized using efficient programming languages [14] [15]. In addition, in the development of Japanese tokenization, evaluation experiments are conducted using formally written Japanese data [16], which is suitable for comparing the accuracy in text segmentation. In IR practice, multilingual data may be analyzed. In this study, actual research papers, which include other languages than the Japanese language, were used for evaluation. Thus, the experimental results of this study are expected to be used as a realistic benchmark in IR tasks such as

[17] The Appendix provides pseudocode for explaining the "algorithm analysis" [11] for this study.

[18] MeCab is explained in detail in the developer's book [8].

[19] Although PHP8 has achieved significant performance improvements over PHP7 and earlier versions, extensive trial and error was required for "performance tuning" [12] of the implementation in this study to ensure its feasibility in the web environment. The proposed method in this study was devised because the baseline method that applied all possible optimization was not fast enough.

educational text analysis and research information management.

As IR practitioners need to demonstrate diverse expertise and deal with interdisciplinary tasks [17], NLP tools should be easily installed, and the time and effort must be spend for more important tasks. From the perspective of contributing to society, this research proposed a technology that has not yet been addressed by the NLP research community. Therefore, our tokenizer can be useful not only for IR practitioners, but also for general users. In future research, we aim to develop other practical tools that can be applied to text analysis in IR tasks.

Acknowledgments

This research was supported by JSPS KAKENHI Grant Number JP23K11764. We thank the reviewers for their careful reading and detailed comments.

References

[1] S. Kurohashi, "From Data Platforms to Knowledge Infrastructure," in *Proceedings of the 2024 Annual International ACM SIGIR Conference on Research and Development in Information Retrieval in the Asia Pacific Region*, 2024, pp. 114–114.

[2] K. Wach, C. D. Duong, J. Ejdys, R. Kazlauskaitė, P. Korzynski, G. Mazurek, J. Paliszkiewicz, and E. Ziemba, "The dark side of generative artificial intelligence: A critical analysis of controversies and risks of ChatGPT," *Entrepreneurial Business and Economics Review*, vol. 11, no. 2, pp. 7–30, 2023.

[3] N. Kai and T. Shimbaru, "Characteristic Analysis of Data Description in Highly Cited Research Data," *IIAI Letters on Institutional Research*, vol. 001, no. LIR010, pp. 1–9, 2022.

[4] H. Phan, S. Hasegawa, and W. Gu, "Implementation of Automated Feedback System for Japanese Essays in Intermediate Education," *IIAI Letters on Informatics and Interdisciplinary Research*, vol. 003, no. LIIR057, pp. 1–11, 2023.

[5] T. Tsumagari, Y. Nakazato, and T. Tsumagari, "A Study on the Influence of Community-Based Education on Post-University Workers and Its Time Dependence," *IIAI Letters on Institutional Research*, vol. 004, no. LIR284, pp. 1–9, 2024.

[6] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, 1967.

[7] J. Aoe and K. Morimoto, "Implementation of trie search strategies by double-array structures," *IPSJ SIG Technical Reports*, vol. 1991, no. 80 (1991-NL-085), pp. 9–16, 1991.

[8] T. Kudo, *Theory and implementation of morphological analysis (Keitaiso kaiseki no riron to jisso)*. Kindai kagaku sha Co.,Ltd., 2018.

[9] M. Asahara and Y. Matsumoto, "ipadic version 2.7. 0 User's Manual," 2003.

[10] R. B. Miller, "Response time in man-computer conversational transactions," in *Proceedings of AFIPS*, 1968, pp. 267–277.

[11] R. Sedgewick, *Algorithms in C, PARTS 1–4: Fundamentals, Data Structures, Sorting, Searching (3rd Edition)*. Pearson Education, 1998.

[12] E. A. Mann, *PHP Cookbook: Modern Code Solutions for Professional Developers*. O'Reilly Media, 2023.

[13] W. A. Burkhard, "Hashing and trie algorithms for partial match retrieval," *ACM Transactions on Database Systems (TODS)*, vol. 1, no. 2, pp. 175–187, 1976.

[14] K. Akabe, S. Kanda, Y. Oda, and S. Mori, "Vaporetto: Efficient Japanese Tokenization Based on Improved Pointwise Linear Classification," *arXiv preprint arXiv:2406.17185*, 2024.

[15] N. Yoshinaga, "Back to Patterns: Efficient Japanese Morphological Analysis with Feature-Sequence Trie," *arXiv preprint arXiv:2305.19045*, 2023.

[16] K. Maekawa, "Compilation of the Balanced Corpus of Contemporary Written Japanese," *Journal of the Japanese Society for Artificial Intelligence*, vol. 24, no. 5, pp. 616–622, 2009.

[17] T. Oishi and T. Nishide, "A Survey on Self-Perception of Institutional Research Skills and Knowledge by Focusing on the Gap with the Participants Needs of Training Courses," *IIAI Letters on Institutional Research*, vol. 004, no. LIR239, pp. 1–12, 2024.

Appendix A Commentary on Time Complexity

In Figure 2, flowchart examples are provided to explain the difference between nested loops with and without a break. Some readers of this paper may be more familiar with pseudocode than with flowcharts. Below, we provide examples of pseudocode and explain the time complexity of non-nested loops, and nested loops with and without a break. Note that the explanation is limited to the scope of this study. For details of the algorithm analysis and O-notation, please refer to books on algorithms, such as Sedgewick's book [11].

Appendix B Non-nested Loops

Non-nested loops are used, for example, performing an operation on the elements of an array in order. This operation is categorized as "linear time," in which the execution time increases in proportion to the input size. This kind of algorithm is denoted as $O(N)$. Here, O-notation is a notation used in the analysis of algorithms to evaluate how execution time increases with respect to the input size. An example pseudocode for this operation is shown below:

Listing 1: Pseudocode for Non-nested Loops

```
for(i = 0; i < N; i++){
  do_some_operation_on_array(i);
}
```

Appendix C Nested Loops Without Break

Nested "for" loops without breaks are used, for example, to look up a dictionary for the extracted substrings from a string. The computation time for this operation is proportional to N^2, where N is the length of the input string. This kind of algorithm is denoted as $O(N^2)$. An example pseudocode for this operation is shown below:

Listing 2: Pseudocode for Nested Loops

```
for(i = 0; i < N - 1; i++){
  for(j = i + 1; j < N; j++){
    look_up_dictionary_for_the_substring(i, j);
  }
}
```

Appendix D Nested Loops With Break

Nested "for" loops with breaks are used, for example, to look up a dictionary for the extracted substrings from a string. If the inner loop is expected to finish in K iterations on average, the computation time for this operation is proportional to $K \times N$. Here, K is not a constant but an independent variable that does not depend on N. This kind of algorithm is denoted as $O(KN)$. An example pseudocode for this operation is shown below:

Listing 3: Pseudocode for Nested Loops with Break

```
for(i = 0; i < N - 1; i++){
  for(j = i + 1; j < N; j++){
    if(some_condition(i, j, N) == true){
      break;
    }
    look_up_dictionary_for_the_substring(i, j);
  }
}
```

In theory, the above operation is $O(N)$ in the best case and $O(N^2)$ in the worst case depending on the value for K. In reality, however, operation time is data-dependent. Specifically, our evaluation experiments confirmed that the proposed method, which implements an $O(KN)$ algorithm, is slower than the comparison method MeCab, which implements an $O(N)$ algorithm, but faster than the baseline method, which implements an $O(N^2)$ algorithm (see Figure 5(b) and Figures 7(a) and (b)). The reason for this evaluation result is that the tokenizer's dictionary has a bias in the length of the words, and many words have a small number of characters (see Figure 8).