Efficient Japanese Tokenization Based on Improved Pointwise Linear Classification

Anonymous ACL submission

Abstract

This paper proposes multiple techniques to improve runtime efficiency of Japanese tokenization based on the Pointwise Linear Classification (PLC) framework, which formulates the whole tokenization process as a sequence of linear classification problems. Our techniques are optimized by leveraging the characteristics of the PLC framework and the task definition itself. Specifically, we introduce (1) composing multiple classifications into array-based operations, (2) efficient feature lookup with memory-optimized automata, and (3) three orthogonal preprocessing to reduce actual score calculation. Combining these techniques, our implementation works 5.7 times faster than the existing tokenizer based on the same model without any loss of tokenization accuracy.[1]

1 Introduction

In languages without explicit word boundaries such as Japanese and Chinese, natural language processing systems are required to determine these boundaries from unsegmented texts before any types of word-based analyses. In Japanese processing, two major linguistic[2] tokenization methods have been proposed: *lattice* methods and *pointwise* methods. The lattice methods (Hisamitsu and Nitta, 1990) generate a lattice of possible tokenizations over the input text and determines the path on the lattice that minimizes a given cost function. In contrast, pointwise methods (Shinnou, 2000; Sassano, 2002; Neubig et al., 2011; Kitagawa and Komachi, 2018; Tolmachev et al., 2019) utilize a binary classifier to predict whether a particular character boundary becomes a word boundary or not, as in Figure 1. A thorough analysis revealed that both methods are comparable in terms of tokenization accuracy (Mori et al., 2011).

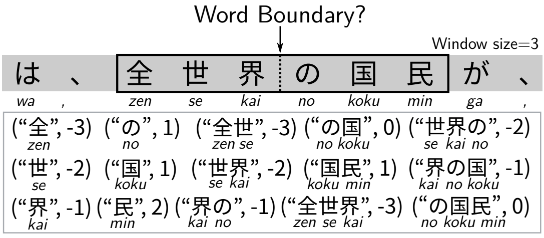

Figure 1: An example of Japanese tokenization with the pointwise method. The bottom box contains character n-gram features described in Section 2.1.

Although the pointwise methods could work in a linear time of input lengths as described in Section 2, it requires careful designs of the overall algorithm to reduce unnecessary bottlenecks which cause a main concern of runtime efficiency. Particularly, comparing well-known tools of both methods, we can see that KyTea (Neubig et al., 2011) (pointwise) is almost 2.1 times slower than MeCab (Kudo et al., 2004) (lattice), meaning that there are still rooms for improvements.

In this paper, we focus on designing an efficient algorithm of the pointwise methods with linear classifiers, which we call *Pointwise Linear Classification* (PLC), without changing the model structure. Our algorithm works as the same as KyTea, but is extremely faster than the original implementation. To this end, we (1) formulate the whole PLC algorithm as a set of array manipulations (Section 2.3), (2) introduce an efficient pattern matching algorithm to look up features from the text (Section 3.1), and (3) propose three preprocessing methods to reduce runtime score calculation (Sections 3.2 to 3.4). Experiments show that all of our approaches improve the overall tokenization speed and eventually runs 5.7 times faster than KyTea in a controlled environment. We also provide thorough analyses of the proposed methods to capture the tendency of their behavior from different perspectives.

[1] Our implementation is available as an open source software at https://example.com/ (MIT or Apache-2.0)

[2] There are also unsupervised tokenization methods such as SentencePiece (Kudo and Richardson, 2018) but we focused on linguistic tokenization which is still important for word sensitive methods such as information retrieval.

2 Pointwise Linear Classification

2.1 Algorithm Overview

Pointwise methods predict whether a particular character boundary becomes a word boundary or not using a binary classifier with *context features* (Shinnou, 2000). Sassano (2002) and Neubig et al. (2011) introduce an SVM with three types of context features: *character n-grams*, *type n-grams*, and *dictionary features*. These features are generated within a sliding window of width W which contains a sequence of surrounding characters around the boundary. Since the classifier is defined independently from the context features, we can employ off-the-shelf binary classification models for this purpose. KyTea introduced a linear SVM as the classification model and utilized LIBLINEAR (Fan et al., 2008) for training the model parameters. According to this characteristics, KyTea models can be considered as a variant of the PLC. As we discussed later, PLC models are capable of many optimization techniques.

A single classification in a PLC model is formulated as follows:

$$y_i(\boldsymbol{w}, x_i) := \boldsymbol{w}^\mathsf{T}\boldsymbol{\phi}(x_i) + b, \qquad (1)$$

where x_i is the i-th input data corresponding to the i-th character boundary, $\boldsymbol{\phi}(x_i)$ is a binary vector representing a set of available features extracted from x_i , $\boldsymbol{w}$ is a weight vector corresponding to all features, and b is a scalar bias. The resulting value y_i suggests how likely the character boundary becomes a word boundary: the higher the more likely it becomes a word boundary. $y_i = 0$ is the decision boundary of this formulation: the classifier determines the word boundary where y_i is positive.

2.2 Context Features

In Figure 1, we input a piece of Japanese text, and the model is predicting whether the character boundary “界-の” is a word boundary or not. Figure 1 shows available character n-gram features in this prediction. Character n-gram features are defined as a pair of character n-grams and their relative positions from the boundary.

Similarly to character n-grams, the type n-gram features are defined as a pair of *character type* n-grams and their relative positions. Character types are defined as a function $t(a)$ which assigns a character a to one of 6 categories: H (Hiragana), T (Katakana), K (Kanji), D (Digit), R (Roman), and O (Other). For example, the character

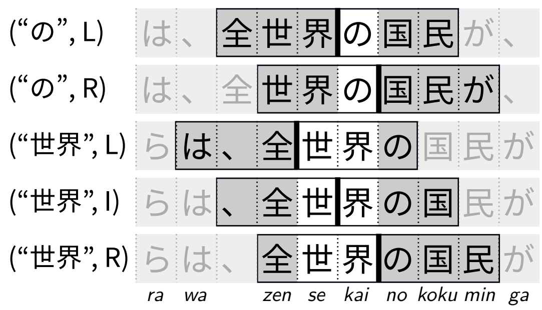

Figure 2: Examples of dictionary features of two words “の” and “世界” with different positions. The I feature is repeated for all intermediate character boundaries.

n-gram “世界の” is mapped to the type n-gram $[t(‘世’), t(‘界’), t(‘の’)] = [‘K’, ‘K’, ‘H’]$.

If a character boundary is overlapped by some dictionary words, dictionary features corresponding to the word are additionally introduced to enhance the confidence of the prediction.[3] Each dictionary word has at most 3 types of features: L (the leftmost side of the word), R (the rightmost side of the word), and I (any boundaries inside the word). Figure 2 shows an example of dictionary features.

2.3 PLC with Pattern Matching

In Equation (1), we need to extract the features $\boldsymbol{\phi}(x_i)$ and calculate an inner product for every character boundary. Since consecutive classifications in PLC requires similar features, this process can be composed into a unified routine in Algorithm 1.

Figure 3 shows an example of this algorithm with character n-gram features. First, the input text of N characters is decomposed into a character sequence $\boldsymbol{a} = (a_0, a_1, \ldots, a_{N-1})$ and analyzed by the pattern matching function $\mathrm{Match}(\boldsymbol{a})$ to look up all character n-grams with nonzero weights. The *score array* $\boldsymbol{w}_{\text{pattern}}(\boldsymbol{q})$ is defined for each n-gram pattern $\boldsymbol{q}$ and formulated as follows:

$$\boldsymbol{w}_{\text{pattern}}(\boldsymbol{q}) := \left(w_{(\boldsymbol{q}, W-n)}, w_{(\boldsymbol{q}, W-n-1)}, \ldots, w_{(\boldsymbol{q}, -W)}\right), \qquad (2)$$

where $w_{(\boldsymbol{q},\cdot)}$ is a weight corresponding to the n-gram feature $(\boldsymbol{q}, \cdot)$ described in Section 2.2 and n is the length of the n-gram. Each score array contains $2W - n + 1$ elements corresponding to all relative positions. For each $\boldsymbol{q}$, a corresponding score array is integrated to the appropriate span of

[3] According to Neubig et al. (2011), word features are shared among all words with the same number of characters. Our definition is more general as well as covering the original definition.

Algorithm 1 Pointwise linear classification using n-gram pattern matching.

Input: Text: $\boldsymbol{a}$, Weights: $\boldsymbol{w}$
Output: Result array: $(y_1, y_2, \ldots, y_{N-1})$

$(y_1, y_2, \ldots, y_{N-1}) \longleftarrow \mathbf{0}$
for $[j, k) \in$ Match$(\boldsymbol{a})$ **do**
  $\boldsymbol{q} \longleftarrow (a_j, a_{j+1}, \ldots, a_{k-1})$
  $\boldsymbol{w}' \longleftarrow \boldsymbol{w}_{\text{pattern}}(\boldsymbol{q})$
  $p \longleftarrow k - W$
  for $i = 0$ to $2W - n$ **do**
    if $p + i \in [1, N-1]$ **then**
      $y_{p+i} \longleftarrow y_{p+i} + w'_i$
    end if
  end for
end for
return $(y_1, y_2, \ldots, y_{N-1})$

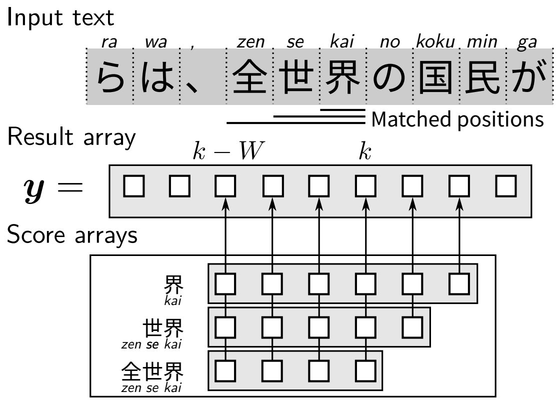

Figure 3: Integrating character n-gram scores to the result array $\boldsymbol{y}$. $W = 3$. $\boldsymbol{w}_{\text{pattern}}$(“界”) has 6 weights, $\boldsymbol{w}_{\text{pattern}}$(“世界”) has 5 weights, and $\boldsymbol{w}_{\text{pattern}}$(“全世界”) has 4 weights as formulated in the Equation (2). Each score array is integrated to the position $k - W$ on $\boldsymbol{y}$, where k is the rightmost position of the pattern.

the *result array* $(y_1, y_2, ..., y_{N-1})$, representing a sequence of classification results.

Although Algorithm 1 does not show any improvements in terms of complexity, the calculation of y_i is decomposed into elementwise summing between multiple arrays that has ability to bring high hardware-level throughput.[4] Algorithm 1 also involves several bottlenecks that can be improved as discussed in Section 3.

Note that a similar approach to Algorithm 1 has also been adopted in KyTea, but there are several rooms for improvement about the entire efficiency of the algorithm. In our implementation, we focused on adjusting the whole algorithm to maximize the actual utilization in modern processors.

3 Improving Efficiency of PLC

3.1 Efficient Pattern Matching

The Match function runs over the whole input text $\boldsymbol{a}$, and discovers all available substrings registered in the pattern set of the function. We need three Match functions to achieve the whole feature lookup: character n-grams, type n-grams, and dictionary. The character n-grams and dictionary need to match over the raw characters $\boldsymbol{a}$, while the type n-grams need to match over the sequence of character types $t(\boldsymbol{a}) := [t(a_0), t(a_1,), ..., t(a_{N-1})]$. This pattern matching is solved efficiently by so-called *Aho-Corasick (AC)* algorithm (Aho and Corasick, 1975; Maruyama, 1994) which is also used in KyTea. The AC algorithm employs an automaton called *Pattern Matching Automaton* (PMA) that performs in $O(N + \text{occ})$ time in the most efficient cases, where occ is the number of pattern occurrences in the text.

The complexity of the AC algorithm additionally depends on the data structure of the inner PMA (Nieminen and Kilpeläinen, 2007). KyTea employs a binary search to discover state transitions of the PMA due to the large alphabet size of Japanese characters, which actually requires $O(N \log \sigma + \text{occ})$ where σ is the expected number of possible transitions. To mitigate this problem, we employ *compacted double-arrays* (CDAs) (Aoe, 1989; Yata et al., 2007). CDAs adjust the assignment of state IDs to share the memory space of their transition mappings as much as possible, which enables direct lookup of state transitions by character IDs so that the whole performance of the PMA becomes back $O(N + \text{occ})$.

3.2 Merging Character n-gram Scores

PLC models usually introduce character n-grams with different n-s to capture a variety of contexts around the boundary. In this case, a longer n-gram overlaps several substring n-grams and the PMA may report all possible n-grams sharing the same suffix as well as the longest n-gram.

Figure 3 shows an example of summing three score arrays of “界” (domain), “世界” (the world) and “全世界” (overall the world). Since these patterns are suffixes of the longest one, summing the

[4] Specifically, arithmetic with sequential access is beneficial for (i) accurate branch prediction, (ii) high availability of hardware caches, and (iii) high availability of SIMD optimizations.

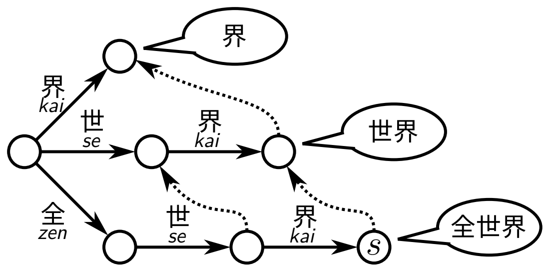

Figure 4: A PMA built from three patterns (“界”, “世界”, “全世界”). Balloons indicate patterns reported at corresponding states. Dotted lines indicate failure edges to non-root states.

score array $\boldsymbol{w}_{\text{pattern}}$(“全世界”) always involves summing $\boldsymbol{w}_{\text{pattern}}$(“世界”) and $\boldsymbol{w}_{\text{pattern}}$(“界”).

Figure 4 shows a PMA built from “界”, “世界”, and “全世界”. When the PMA reaches the state s, it yields a list of all possible suffixes collected by tracing a chain of failure edges from s, and the Algorithm 1 eventually aggregates all score arrays corresponding to the yielded suffixes. Since each state in the PMA yields always the same list of possible suffixes, we can calculate a *partial sum* of the score arrays $\boldsymbol{w}_{\text{state}}(s)$ in advance by summing over all possible $\boldsymbol{w}_{\text{pattern}}(\cdot)$:

$$\boldsymbol{w}_{\text{state}}(s) := \sum_{\boldsymbol{q} \in \mathcal{S}(s)} \boldsymbol{w}_{\text{pattern}}(\boldsymbol{q}), \tag{3}$$

where $\mathcal{S}(s)$ is the set of possible suffixes at s. Using $\boldsymbol{w}_{\text{state}}(\cdot)$ instead of $\boldsymbol{w}_{\text{pattern}}(\cdot)$ improves the runtime efficiency by allowing the Algorithm 1 to aggregate score arrays only once for each character boundary.

3.3 Integrating Dictionary Words

As we discussed in Section 3.1, PMAs of the character n-grams and the dictionary run over the same sequence $\boldsymbol{a}$ and some of their patterns may be overlapped each other. This suggests that we can further integrate score arrays of dictionary words into the partial sum of the character n-gram scores to reduce certain amount of burdens caused by matching dictionary words. To this end, we propose two methods to integrate dictionary features as discussed in the following sections.

3.3.1 Integrating Short Dictionary Words

As we can see in Figure 3, the score arrays of character n-grams at the position k start with always the position $k - W$, and dictionary features of any words of lengths $l \leq W$ can be integrated into the partial sums of character n-grams without introducing any extra treatment. Based on this observation, it may be reasonable to integrate only words with short lengths while long words (of lengths $l > W$) are remained in the separate PMA. We call this method Short.

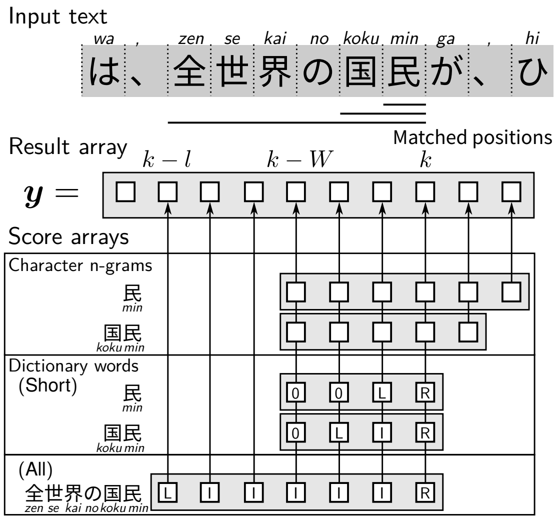

Figure 5: The difference of adding positions of character n-gram scores and dictionary word scores. $W = 3$. L, I, and R indicate types of the dictionary feature. 0 indicates a padding.

3.3.2 Integrating All Dictionary Words

We can further consider to integrate every dictionary word into the partial sums to get rid of the PMA of dictionary words completely. Since score arrays of long words cover beyond the range of character n-gram scores (as in the last case of Figure 5), we need to prepare additional arrays to store partial sums for all long words. We also need to determine the correct starting positions of the score summation, which may cause an additional cost of the calculation. We call this method All.

3.4 Caching Type n-gram Scores

Type n-gram scores are calculated similarly to that of character n-grams, but its alphabet size is limited. As described in Section 2.1, KyTea models distinguish only 6 character types. Since the sliding window of size W contains a sequence of character types of length $2W$, the number of possible type n-gram sequences is only 6^{2W}. This is small enough to store all resulting scores of possible sequences at initialization for a reasonable value of W, which is typically $W = 3$.[5] This approach allows to achieve the score calculation of type n-grams by looking

[5]According to the analysis in Appendix A.4.

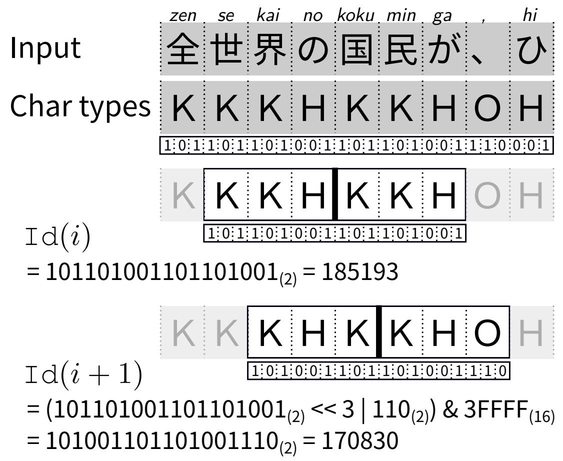

Figure 6: Calculating sequence IDs of character types with $W = 3$. Binary sequences under character types indicate codes related to each character type: for example, $\texttt{Cd}(\mathrm{H}) = 001_{(2)}$. Sequence ID of the next sliding window can be calculated by the current sequence ID and the incoming character type.

up only one integrated score and avoid both the PMA of type n-grams and summing corresponding score arrays.

Specifically, we assign a binary code of 3 bits $\texttt{Cd}(t) \in [1, 6]$ for each character type t, and defines the *sequence ID* $\texttt{Id}(i)$ as follows:

$$\texttt{Id}(i) := \sum_{k=0}^{2W-1} 2^{3(2W-1-k)} c(i - W + k), \quad (4)$$

$$c(i) := \begin{cases} \texttt{Cd}(t_i) & \text{if } i \in [0, N-1], \\ 0 & \text{otherwise,} \end{cases} \quad (5)$$

for each sliding window at position i, where $t_i := t(a_i)$. $\texttt{Id}(i)$ is a $6W$ bit integer which is used as the address of the integrated type n-gram score. Since $\texttt{Id}(i)$ shares most of its subsequence with the neighboring window $\texttt{Id}(i + 1)$ as shown in Figure 6, we can induce $\texttt{Id}(\cdot)$ values recurrently as follows:

$$\texttt{Id}(-W) := 0, \quad (6)$$

$$\texttt{Id}(i + 1) := (2^3 \texttt{Id}(i) + c(i + W))\%2^{6W}, \quad (7)$$

where $\%$ indicates the modulo operation. This calculation can be performed by only a few bit operations and requires a constant time complexity for each character boundary.

4 Experiments

4.1 Setup

We evaluate tokenization speed of our methods with multiple combinations and compared them with conventional tokenization tools. We use short unit words (SUW) in BCCWJ 1.1 corpus (Maekawa et al., 2014)[6] to train PLC models. The corpus consists of 60k Japanese sentences with manually annotated SUW boundaries. We also use 667k words for dictionary features extracted from UniDic 3.1.0 (Den et al., 2007)[7] with manual filtering.[8] Our methods are implemented in Rust and compiled by rustc 1.60.0 with optimization flag `opt-level=3`. Other tools are compiled by GCC 11.2.0 or the same rustc with their recommended configuration. For each experiment, we performed 10-fold cross validation: 9 fractions are used in training, and the remaining one is used for test. The PLC model is trained by the LIBLINEAR with L1 regularization (Tibshirani, 1996). The same PLC model is used in both KyTea and our methods. We fixed several hyperparameters to obtain representative metrics of each method: the penalty parameter C to 1, the window size W to 3, and the maximum length of n-grams to 3, according to the analysis in Appendix A.2 and Mori et al. (2011).

In our experiments, we are especially targeting to compare the performance of the systems on a long-lived server service. To this end, we aim to measure only the computing overhead imposed by the actual tokenization processes. Specifically, model initialization (loading the parameters and preprocessing integrated scores) are omitted from the resulting measure because these are performed only once during the runtime. In addition, all test sentences are loaded onto the memory in advance to avoid constant disk access.

We perform all experiments in a single thread process on a dedicated machine with Intel Core i7-8086K CPU (4GHz, 6 cores, 32KiB L1, 256KiB L2, 12MiB L3) and 64GiB DDR4 RAM.

4.2 Baseline systems

We introduced following tools as our baselines:

KyTea[9] The original implementation of the PLC method. Since we use the same model in both KyTea and our methods, tokenization accuracy of both systems are exactly the same.

[6] https://clrd.ninjal.ac.jp/bccwj/en/

[7] https://clrd.ninjal.ac.jp/unidic/en/ (GPL-2 or LGPL-2.1 or BSD-3)

[8] We removed all words including whitespaces.

[9] https://github.com/neubig/kytea (Apache-2.0)

Table 1: Average time elapsed to tokenize the test data. [ms]

Tokenizer	Time	Time − (a)	SD
KyTea (2020-04-03)	136.6		5.6
Our methods			
(a): §3.1 Efficient Pattern Matching	33.2	(0)	0.5
(b): (a) + §3.2 Merging Character n-gram Scores	31.2	-2.0	0.6
(c): (a) + §3.3 Integrating Dictionary Scores (All)	29.4	-3.8	0.5
(d): (a) + §3.4 Caching Type n-gram Scores	30.2	-3.0	0.5
(e): (a) + §3.2 + §3.3 (All) + §3.4	**23.8**	**-9.4**	0.4
MeCab (2020-09-14)			
IPADic	65.4		1.9
UniDic	161.5		7.3
sudachi.rs (0.6.4-a1)	169.5		3.4

Table 2: Average number of score arrays aggregated during tokenizing the test data. [$\times 10^3$]

Subroutine	(a)	(b)	(a)+Short	(a)+All	(b)+Short	(b)+All
Character n-grams	360.	203.	386.	392	204.	204.
Dictionary words	307.	307.	6.	–	6.	–
Total	667.	509.	392.	392	210.	204.

MeCab[10] A major Japanese tokenizer based on the lattice method. We used IPADic dictionary (Asahara and Matsumoto, 2003) as well as UniDic.[11]

sudachi.rs[12] An efficient implementation of Sudachi (Takaoka et al., 2018) which is also based on the lattice method. We picked up Sudachi because it is widely used as an internal tokenizer of larger systems such as spaCy.[13] We used *SudachiDict-core 20210802* model and disabled all postprocessing.

4.3 Overall Speed Comparison

Table 1 shows average time elapsed to tokenize the test data. The results in KyTea and our methods include only time elapsed by the boundary classification, while MeCab and sudachi.rs involves the whole analysis which is hard to be separated due to the model formulation.[14]

First, we focus on 5 different settings (a) to (e) of our methods described in Section 3. The settings (b), (c), and (d) run faster than (a), demonstrating that these preprocessing approaches are effective to suppress computation time. We can also see that applying the all preprocessing (e) achieves the fastest result and the overall improvement is comparable with the sum of (b), (c), and (d). This suggests that these techniques are orthogonal and improves different part of computation in the whole algorithm.

Comparing (e) and other tools, our method achieves 5.7 times faster performance than KyTea, 6.8 times faster than MeCab with the same dictionary,[15] and 7.1 times faster than sudachi.rs.

4.4 Performance of Subroutines

Table 2 shows the average number of score arrays aggregated during score calculation of all test examples. We focused on the number of arrays rather than the number of actual scores because most score arrays can be aggregated by at most a few SIMD instructions. We do not show corresponding metrics of type n-grams because its calculation can be removed completely by the method described in Section 3.4.

[10] `https://github.com/taku910/mecab` (GPL-2 or LGPL-2.1 or BSD-3)

[11] We provide a reference performance of MeCab with a typical configuration (IPADic) for a fair comparison of speeds.

[12] `https://github.com/WorksApplications/sudachi.rs` (Apache-2.0)

[13] `https://spacy.io/`

[14] Lattice methods are designed to consider a joint distribution of tokenization and morphology.

[15] PLC models rely on word boundary labels annotated in the training corpus (BCCWJ) and the word unit of the dictionary must be compatible with the corpus' standard to avoid unnecessary lacking of tokenization accuracy. Since the word unit of IPADic is not compatible with BCCWJ, we did not prepare the results of PLC models with IPADic.

Table 3: Average time elapsed to process character n-grams and dictionary. [ms]

Subroutine	(a)	(b)	(a)+Short	(a)+All	(b)+Short	(b)+All
Character n-grams						
Pattern matching	6.15	5.99	7.68	9.65	7.35	9.09
All	7.46	6.75	10.96	17.75	9.60	15.88
Dictionary words						
Pattern matching	8.28	8.26	5.82	–	5.74	–
All	13.63	13.56	6.27	–	6.36	–
All subroutines	23.67	22.82	20.77	17.75	19.40	15.88

We can see that merging character n-gram scores (b) reduces 44% of array summations involved by character n-gram features, and integrating short dictionary words (a)+Short reduces 98% of array summations involved by dictionary features. (a)+Short also increases the number of array summations in character n-grams slightly due to introducing unseen character n-grams derived from the dictionary.

On the other hand, combining both methods (b)+Short successfully reduced the calculation of character n-gram scores to a comparable range of (b). This tendency shows that even if the dictionary contains unseen character n-grams, they can be integrated to the partial sums of other patterns in most cases. Comparing methods Short and All in both (a) and (b), there was no significant difference of the number of score summations by integrating long words.

Table 3 shows the average time elapsed during each subroutine. We measured each metric by simply disabling other subroutines from the whole process. The last row shows the time with all subroutines, which should be longer than the sum of all individual metrics due to the lack of caching efficiency. We can see that the time of character n-grams in (b)+All is longer than that in (b)+Short nevertheless the numbers of score summations in Table 2 are almost the same. This is expected because the method All introduces an additional complexity as discussed in Section 3.3.2. However, we can also see that the whole process of (b)+All achieved faster performance than (b)+Short because (b)+All eventually removes the whole process of dictionary features completely.

4.5 Effect of Model Size

We analysed the impact of model sizes against tokenization efficiency by varying the penalty parameter C of L1 regularization (Tibshirani, 1996) during training SVMs.[16] L1 regularization squashes certain number of weights into 0 and such weights can be removed from the model (Gao et al., 2007).

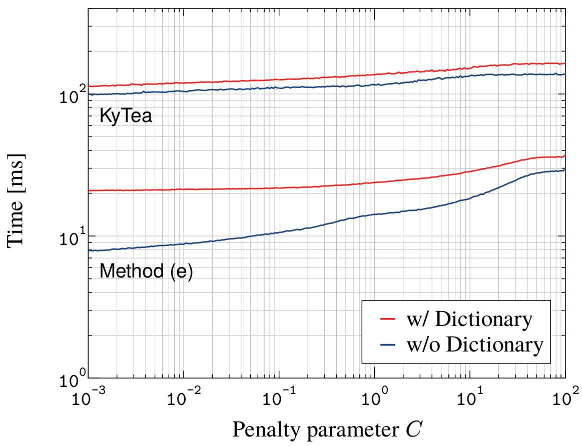

Figure 7: Effect of the penalty parameter C against the elapsed time of each method.

Figure 7 shows the whole processing time of the method (e) and KyTea with varying C. Intuitively, both models achieve faster processing speed with small C (strong regularization), and our method achieves better processing speed against KyTea for every values of C.

Considering the gradient of each plot in Figure 7, we can also see that C brings larger exponential effect against processing speed of the method (e) than KyTea. This tendency is reasonable because our methods heavily rely on the CPU architecture: large models may cause many disruption of efficient computing. Here we provide several observations of this tendency in detail.

We hypothesized first that this tendency is simply caused by increasing the number of array summations because a larger model may discover more patterns on the same input. As Figure 8 shows, this

[16] We followed the same definition of C in Fan et al. (2008): the smaller the stronger regularization. The actual relation between model sizes and C is shown in Appendix A.3.

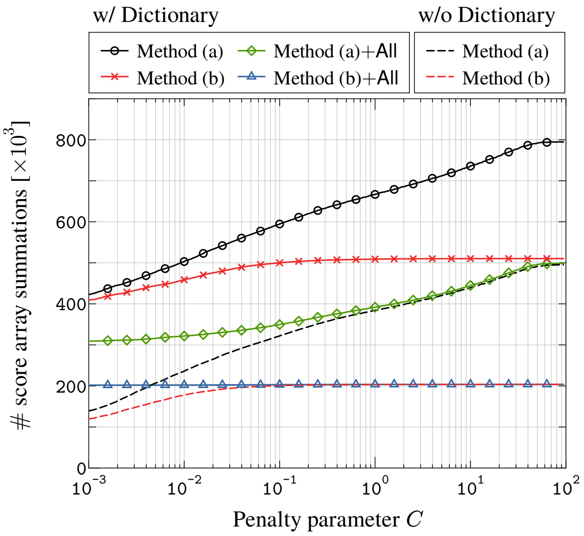

Figure 8: Effect of the penalty parameter C against the number of score array summations.

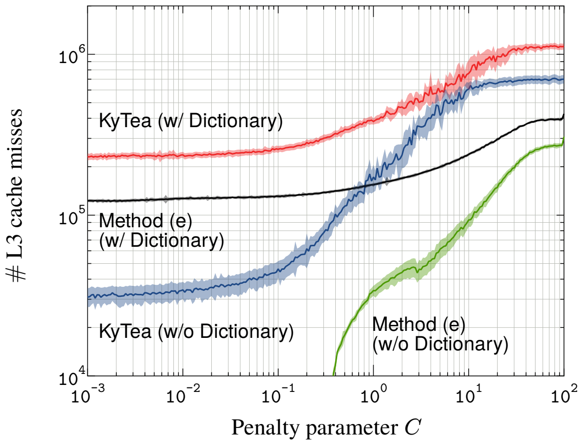

Figure 9: Effect of the penalty parameter C against the number of L3 cache misses of each method. Lines and areas indicate means and standard deviations, respectively.

hypothesis is not correct because merging character n-grams ((b) and (b)+All) effectively suppressed the number of array summations under a certain upper bound even when C was large (weak regularization). This means that the speed reduction on large C is not caused by the complexity of the algorithm itself.

We also investigated hardware level efficiency of each method. As clearly shown in Figure 9, the number of CPU cache misses[17] grows significantly by increasing C despite maintaining the number of summation operations. This phenomenon is explained by observing access frequency of score

[17]Counted by the `perf_event_open` system call on Linux.

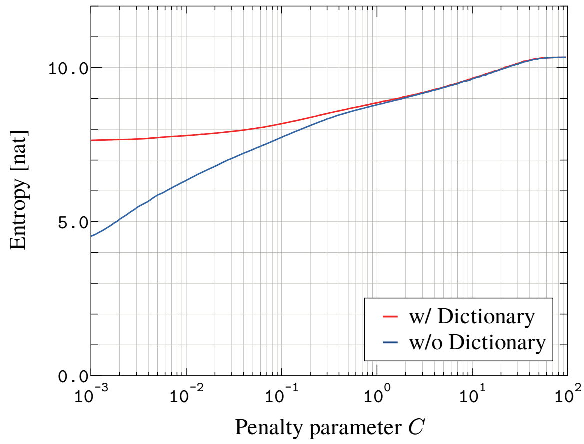

Figure 10: Entropy of access frequency distribution of character n-gram score arrays by the method (e).

arrays. Figure 10 summarizes the access frequency distribution of score arrays as an entropy, and shows that large models tend to require more varied score arrays than small models to calculate the final scores. Since the CPU cache can remember only the neighboring contents around the memory accessed recently, requiring varied parts of memory lacks utilization of the cache, resulting in overall speed reduction of the algorithm.

5 Conclusion and Future Work

We introduced multiple techniques to improve efficiency of the Japanese tokenizer based on Pointwise Linear Classification (PLC) models. Experiments clearly showed substantial improvements brought by our methods compared with the baseline implementation (KyTea) and other tokenization tools in terms of tokenization speed. Although we focused especially on the tokenization task, some of the techniques presented in this paper is generic and can also be applied to other tasks if it can be decomposed into a sequence of multiple problems in the same manner.

We improved only the tokenization speed in this paper because it is the most essential part of practical use-cases of tokenizers. Improving the whole efficiency of the PLC based lexical analysis is also challenging; it is one of the main focus of our future work. Especially, PLC based part-of-speech tagging requires much larger alphabet size (set of *words* rather than *characters*) and further improvement of the PMA architecture is required.

References

Alfred V. Aho and Margaret J. Corasick. 1975. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340.

Jun-ichi Aoe. 1989. An efficient digital search algorithm by using a double-array structure. *IEEE Transactions on Software Engineering*, 15(9):1066–1077.

Masayuki Asahara and Yuji Matsumoto. 2003. *ipadic version 2.7.0 User's Manual*. Nara Institute of Science and Technology.

Yasuharu Den, Toshinobu Ogiso, Hideki Ogura, Atsushi Yamada, Nobuaki Minematsu, Kiyotaka Uchimoto, and Hanae Koiso. 2007. The development of an electronic dictionary for morphological analysis and its application to Japanese corpus linguistics (in japanese). *Japanese linguistics*, 22:101–123.

Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. Liblinear: A library for large linear classification. *J. Mach. Learn. Res.*, 9:1871–1874.

Jianfeng Gao, Galen Andrew, Mark Johnson, and Kristina Toutanova. 2007. A comparative study of parameter estimation methods for statistical natural language processing. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 824–831, Prague, Czech Republic. Association for Computational Linguistics.

Toru Hisamitsu and Yoshihiko Nitta. 1990. Morphological analysis of Japanese sentences by minimum connective-cost method (in Japanese). *SIGNLC*, pages 17–24.

Yoshiaki Kitagawa and Mamoru Komachi. 2018. Long short-term memory for Japanese word segmentation. In *Proceedings of the 32nd Pacific Asia Conference on Language, Information and Computation*, Hong Kong. Association for Computational Linguistics.

Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium. Association for Computational Linguistics.

Taku Kudo, Kaoru Yamamoto, and Yuji Matsumoto. 2004. Applying conditional random fields to Japanese morphological analysis. In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, pages 230–237, Barcelona, Spain. Association for Computational Linguistics.

Kikuo Maekawa, Makoto Yamazaki, Toshinobu Ogiso, Takehiko Maruyama, Hideki Ogura, Wakako Kashino, Hanae Koiso, Masaya Yamaguchi, Makiro Tanaka, and Yasuharu Den. 2014. Balanced corpus of contemporary written Japanese. *Lang. Resour. Eval.*, 48(2):345–371.

Hiroshi Maruyama. 1994. Backtracking-Free Dictionary Access Method for Japanese Morphological Analysis. In *Proceedings of the 15th Conference on Computational Linguistics - Volume 1*, COLING '94, page 208–213, USA. Association for Computational Linguistics.

Shinsuke Mori, Yosuke Nakata, Graham Neubig, and Tatsuya Kawahara. 2011. Morphological Analysis with Pointwise Predictors. *Journal of Natural Language Processing (in Japanese)*, 18(4):367–381.

Masaaki Nagata. 1994. A stochastic Japanese morphological analyzer using a forward-DP backward-A* n-best search algorithm. In *COLING 1994 Volume 1: The 15th International Conference on Computational Linguistics*.

Graham Neubig, Yosuke Nakata, and Shinsuke Mori. 2011. Pointwise prediction for robust, adaptable Japanese morphological analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 529–533, Portland, Oregon, USA. Association for Computational Linguistics.

Janne Nieminen and Pekka Kilpeläinen. 2007. Efficient implementation of aho–corasick pattern matching automata using unicode. *Software: Practice and Experience*, 37(6):669–690.

Manabu Sassano. 2002. An empirical study of active learning with support vector machines for Japanese word segmentation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 505–512, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Hiroyuki Shinnou. 2000. Deterministic Japanese word segmentation by decision list method. In *Proceedings of the 6th Pacific Rim International Conference on Artificial Intelligence*, PRICAI'00, page 822, Berlin, Heidelberg. Springer-Verlag.

Kazuma Takaoka, Sorami Hisamoto, Noriko Kawahara, Miho Sakamoto, Yoshitaka Uchida, and Yuji Matsumoto. 2018. Sudachi: a Japanese tokenizer for business. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan. European Language Resources Association (ELRA).

Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288.

Arseny Tolmachev, Daisuke Kawahara, and Sadao Kurohashi. 2019. Shrinking Japanese morphological analyzers with neural networks and semi-supervised learning. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2744–2755, Minneapolis, Minnesota. Association for Computational Linguistics.

Susumu Yata, Masaki Oono, Kazuhiro Morita, Masao Fuketa, Toru Sumitomo, and Jun-ichi Aoe. 2007. A compact static double-array keeping character codes. *Inf. Process. Manage.*, 43(1):237–247.

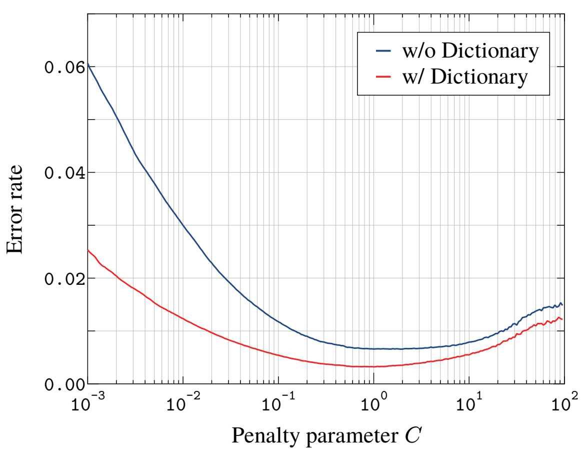

Figure 11: Effect of penalty parameter C against mean error rate.

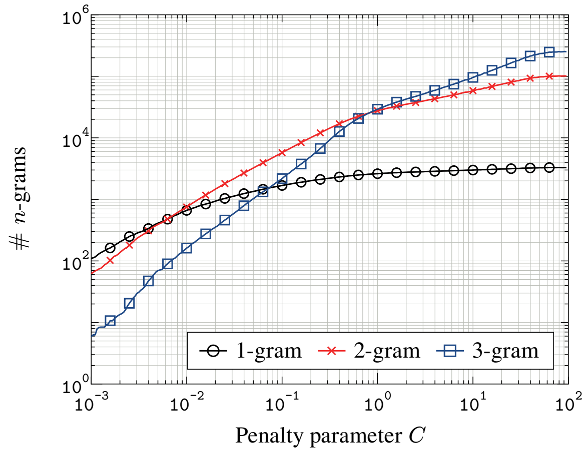

Figure 12: Effect of penalty parameter C against the number of n-grams in the model. (w/o Dictionary)

A Appendix

A.1 Accuracy Metrics

To measure the tokenization accuracy, we choose two metrics: *boundary error rate* which is the ratio of false classifications for all character boundaries, and the *word-wise* F_1 *measure* introduced by Nagata (1994).

A.2 Relation Between Penalty Parameter and Accuracy

We investigated the effect of the strength of the L1 regularization against the tokenization accuracy. Figure 11 shows the tendency of the boundary error rate with varying the penalty parameter C. As we discussed in the section 4.2, all of our methods and KyTea shares the same accuracy. We can see that the error rate becomes the minimum around $C = 1$ with and without employing a dictionary.

A.3 Relation Between Penalty Parameter and Model Size

We investigated the effect of the strength of the L1 regularization on the model size. Figure 12 shows the tendency of the number of n-grams in the model, and Figure 13 shows the number of states in the PMA with varying C. We can see that lager C (weak regularization) yields more n-grams with nonzero weights, and hence it requires more PMA states. We can also see that C has different effects for n-grams with different n-s.

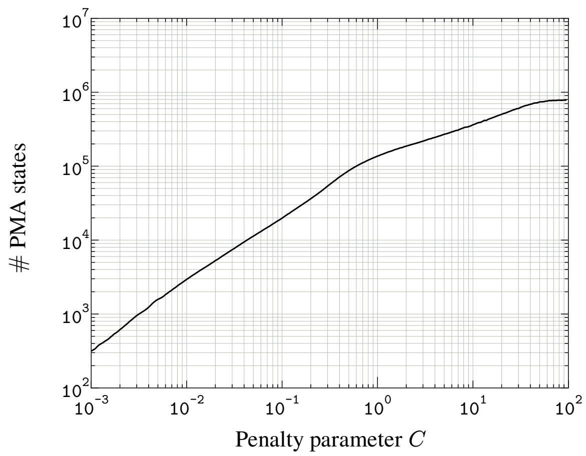

Figure 13: Effect of penalty parameter C against the number of PMA states in the model. (w/o Dictionary)

A.4 Relation Between Window Size and Accuracy

We investigated the effect of the window size W and the maximum length of n-grams against tokenization speed and accuracy. We did not use any dictionary for this experiment because the dictionary feature is independent from both W and n. For models with $W \geq 4$, we did not introduce caching type n-gram scores discussed in Section 3.4 because it requires a large amount of memory.

Table 4 shows the comparison of the tokenization accuracy and the speed. We can see that the tokenization accuracy is saturated with a certain value of W and n (3 for SUW and 5 for LUW), although the tokenization speed drops when we selected larger W and n.

Table 4: Effect of window size W and n-gram size n on elapsed time [ms] and accuracy.

W, n	SUW			LUW		
	Time	F_1	Error rate	Time	F_1	Error rate
1,1	7.7	0.8265	0.0887	7.6	0.8207	0.0749
2,2	11.3	0.9810	0.0093	11.1	0.9608	0.0155
3,3	13.3	0.9867	0.0066	13.3	0.9779	0.0088
4,4	15.7	0.9867	0.0066	15.9	0.9805	0.0078
5,5	20.1	0.9862	0.0069	20.5	0.9807	0.0077
6,6	23.3	0.9857	0.0071	24.2	0.9804	0.0078
7,7	25.6	0.9850	0.0074	27.1	0.9799	0.0080