Back to Patterns:
Efficient Japanese Morphological Analysis with Feature-Sequence Trie

Naoki Yoshinaga
Institute of Industrial Science, The University of Tokyo
ynaga@iis.u-tokyo.ac.jp

Abstract

Accurate neural models are much less efficient than non-neural models and are useless for processing billions of social media posts or handling user queries in real time with a limited budget. This study revisits the fastest pattern-based NLP methods to make them as accurate as possible, thus yielding a strikingly simple yet surprisingly accurate morphological analyzer for Japanese. The proposed method induces reliable patterns from a morphological dictionary and annotated data. Experimental results on two standard datasets confirm that the method exhibits comparable accuracy to learning-based baselines, while boasting a remarkable throughput of **over 1,000,000 sentences per second** on a single modern CPU. The source code is available at https://www.tkl.iis.u-tokyo.ac.jp/~ynaga/jagger/.

1 Introduction

The amount of text data being processed has greatly increased since the advent of communication platforms such as Twitter, Zoom, and Slack, and NLP services such as DeepL and Grammarly have millions of users. Some users analyze textual big data for marketing, linguistics, or sociology, while others deploy NLP services on their own devices because of privacy concerns. It is therefore becoming important to develop highly efficient methods to process massive text data and user queries with limited computational resources.

However, the recent campaign for efficient NLP does not focus on literally efficient methods that scale to increasing data sizes and run on resource-constrained devices. Instead, most "efficient" NLP studies (Treviso et al., 2022) focus on neural methods, which are too slow to handle billions of social media posts and too large to deploy on edge devices. Those studies seek to make model training or inference *relatively* efficient within the deep learning framework. Thus, the large efficiency gap with respect to classical methods has never been filled.

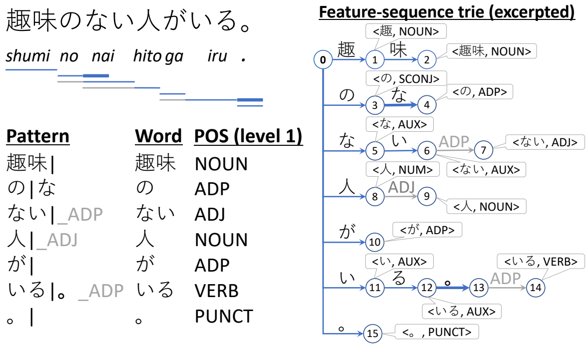

Figure 1: Pattern-based morphological analysis via a feature-sequence trie. The blue and gray lines below the input indicate pattern matches (trailing characters and previous POS tags) to determine where to split (indicated by '|' in the patterns) and what to tag.

In this study, I take an orthogonal approach toward *absolutely* efficient NLP by seeking to boost the accuracy of the fastest methods. Specifically, I have developed a remarkably simple yet accurate method for Japanese morphological analysis, which is a joint task of word segmentation, part-of-speech (POS) tagging, and lemmatization. This method revisits the classical longest matching method; it greedily applies patterns that determine the next position to segment and then identifies the POS tag for the segmented word, as illustrated in Figure 1. To obtain reliable patterns, starting from words in a morphological dictionary and training data, patterns are extended with posterior surface contexts and previous POS tags, and the patterns' segmentation offsets and tags are determined by frequency. The extracted patterns are then stored in an efficient double-array trie (Aoe, 1989).

The proposed method was evaluated on two standard corpora (Kurohashi and Nagao, 2003; Hangyo et al., 2012). The experimental results confirmed that this simple method can process 1,000,000 sentences per second on an M2 MacBook Air, with comparable accuracy to learning-based baselines (Kudo et al., 2004; Neubig et al., 2011).

Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics
Volume 2: Short Papers, pages 13–23

Algorithm 1 Pattern-based morphological analysis

INPUT: sequence of characters, $\boldsymbol{c}$; set of patterns stored in trie, $\mathcal{P} = \{(\mathsf{p}, \mathsf{shift}, t)\}$
OUTPUT: sequence of words with tags $\boldsymbol{s} = \{(w_j, t_j)\}$

$i \leftarrow 0$
while $i < \texttt{len}(\boldsymbol{c})$ **do**
  $(\hat{\mathsf{shift}}, \hat{t}) = \texttt{longest\_prefix\_search}(\boldsymbol{c}_{\geq i}, \mathcal{P})$
  $\mathsf{append}(\boldsymbol{s}, (\boldsymbol{c}_i^{i+\hat{\mathsf{shift}}}, \hat{t}))$
  $i \leftarrow i + \hat{\mathsf{shift}}$
return $\boldsymbol{s}$

Algorithm 2 Pattern extraction from training data

INPUT: training data $\mathcal{D}$ and dictionary $\mathcal{V}$
OUTPUT: set of patterns, $\mathcal{P} = \{(p, \mathsf{shift}, t)\}$

1: $\hat{\mathcal{P}} \leftarrow \phi$
2: $L_{\text{max}} = \max_{(w,t)\in\mathcal{V}} \texttt{len}(w)$
3: **for all** training examples $(\boldsymbol{c}, \boldsymbol{s} = \{(w_l, t_l)\}_{l=1}^{L}) \in \mathcal{D}$ **do**
4:   $i \leftarrow 0$
5:   **for** $j = 0$ to L **do**
6:     $\mathsf{shift} = \texttt{len}(w_j)$
7:     **for** $k = \mathsf{shift}$ to L_{max} **do**
8:       $\hat{\mathcal{P}}[\boldsymbol{c}_i^{i+k}][(\mathsf{shift}, t_j)] \mathrel{+}= 1$
9:       $\hat{\mathcal{P}}[\boldsymbol{c}_i^{i+k}; t_{j-1}][(\mathsf{shift}, t_j)] \mathrel{+}= 1$
10:     $i \leftarrow i + \mathsf{shift}$
11: $\mathcal{P} \leftarrow \{(w, \texttt{len}(w), \hat{t})\}$ where $(w, *) \in \mathcal{V}, w \notin \hat{\mathcal{P}}$,
12:   $\hat{t} = \text{argmax}_{\{t|(w,t)\in\mathcal{V}\}} \sum_{w'} \hat{\mathcal{P}}[w'][(\texttt{len}(w'), t)]$
13: **for all** pattern candidates $p \in \hat{\mathcal{P}}$ from shortest one **do**
14:   $\mathsf{shift} = \text{argmax}_{\mathsf{shift}} \sum_t \hat{\mathcal{P}}[p][(\mathsf{shift}, t)]$
15:   $t = \text{argmax}_t \hat{\mathcal{P}}[p][\mathsf{shift}, t)]$
16:   $(\mathsf{shift}', t') = \texttt{longest\_prefix\_search}(p, \mathcal{P})$
17:   **if** $(\mathsf{shift}, t) = (\mathsf{shift}', t')$ **then**
18:     $\mathcal{P} \leftarrow \mathcal{P} \cup \{(p, \mathsf{shift}, t)\}$
19: **return** $\mathcal{P}$

2 Pattern-based Morphological Analysis

This section describes the method of Japanese morphological analysis used here, which performs word segmentation, POS tagging, and lemmatization. To maximize the tagging efficiency, I return to a pattern-based algorithm that is similar to the longest matching algorithm (Nagata, 1994).

The longest matching algorithm performs deterministic word segmentation by using a dictionary. Starting from the beginning of the input, it greedily finds the longest dictionary words to segment the input. Although this simple algorithm exhibits moderate accuracy in Chinese and Japanese with transformation rules (Palmer, 1997; Hockenmaier and Brew, 1998; Sassano, 2014), there is a gap in accuracy from search- and classification-based approaches (Kudo et al., 2004; Neubig et al., 2011). To make search-based morphological analysis partially deterministic, Morita and Iwakura (2019) extracted surface patterns from tagging results; however, the speed-up factor was at most 1.5.

2.1 Basic algorithm

Algorithm 1 is a simple, deterministic algorithm for joint word segmentation, POS tagging, and lemmatization. It repeatedly applies the longest-matching patterns in a trie $\mathcal{P}$ to a given sequence of characters, $\boldsymbol{c}$, and a start position i to segment and tag the next word ($w_j = \boldsymbol{c}_i^{i+\hat{\mathsf{shift}}}$ and $\hat{t}_j$). As will be shown later in § 3, this simple algorithm *works* as well as learning-based approaches.

This algorithm is inspired by the longest matching algorithm but differs in that the segmentation offset shift can be smaller than the surface length matched with patterns, k (see Line 7 in Algorithm 2). A running example is shown in Figure 1.

The algorithm is also inspired by the precomputation of feature weights in sequence labeling (Kaji et al., 2010) and classification with conjunctive features (Yoshinaga and Kitsuregawa, 2009, 2010, 2014). Those methods accumulate certain feature weights in advance and retrieve those partial results by using simple keys such as word unigrams, POS bigrams, and primitive feature sequences to compute the final results (labels) by an argmax operation on the weights. The proposed method regards word segmentation and tagging as a joint, multiclass classification problem and directly obtains the label (i.e., where to segment and what to tag) by using the feature sequence as a pattern, thus skipping the expensive argmax operation over a number of labels. The longest matching thus implies classification with as many features as possible.

2.2 Pattern extraction from data

Following the feature templates of learning-based methods (Kudo et al., 2004; Neubig et al., 2011), the algorithm's pattern template was designed as a sequence of characters, $\boldsymbol{c}$, followed by the previous word's POS tag t_{j-1}, thus giving $\boldsymbol{c}; t_{j-1}$, where ';' represents string concatenation.

Algorithm 2 is the procedure to extract patterns for word segmentation and POS tagging from the annotated data and a dictionary. Given training data $\mathcal{D}$ with annotation of (word) segmentations and (POS) tags and a dictionary $\mathcal{V}$ compiling words and their possible tags, the algorithm iteratively extracts possible patterns from $\mathcal{D}$. It first enumerates surface patterns $\boldsymbol{c}_i^{i+k}$ from all starting positions of words in $\mathcal{D}$, and it then concatenates them with tag t_{j-1} for the preceding words to form pattern candidates (Lines 3-10 in Algorithm 2). Patterns are added for dictionary words that are unseen in the

	KYOTO			KWDLC		
	train	dev	test	train	dev	test
# sentences	35,478	1145	1783	12,271	1585	2195
ave. # words	25.37	26.24	25.83	15.85	14.27	16.34

Table 1: Statistics of the evaluation datasets.

	# words	# tags (four levels)				
		1	2	3	4	all (1-4)
JUMAN 5.1	475,716	14	35	34	60	980
JUMAN 7.0	702,358	14	35	33	77	1,188

Table 2: Statistics of the morphological dictionaries.

training data (Lines 11-12). The segmentation offset (shift) and tag t for a pattern are determined by the frequency (Lines 14-15). To avoid extra matching to the posterior contexts and previous tag, we only keep patterns whose segmentation offsets and tags differ from those of the longest *prefix* patterns that share prefixes of posterior contexts (Lines 16-18). This not only reduces the number and length of patterns but also minimizes the longest matching method's overhead for word segmentation.[1]

3 Experiments

This section describes an experimental evaluation of the pattern-based morphological analyzer on two annotated corpora in different domains (Kurohashi and Nagao, 2003; Hangyo et al., 2012). The method was compared with two learning-based baselines (Kudo et al., 2004; Neubig et al., 2011) in terms of efficiency and accuracy. Note that all language resources and software used in the experiments are publicly available and free for academic use.

3.1 Setup

Data The experiments used the Kyoto-University Text Corpus[2] (KYOTO) (Kurohashi and Nagao, 2003), compiled from newspaper articles, and the Kyoto-University Web Document Leads Corpus[3] (KWDLC) (Hangyo et al., 2012), compiled from the first three sentences of various Web pages. I adopted the split of development and test sets given in the corpora's `github` repositories and used the remaining portions as training sets. The datasets' statistics are listed in Table 1.

Methods The three methods below were compared. To prevent overfitting, the hyperparameter C in the underlying model was tuned for the two learning-based baseline methods[4] by using the development set to maximize the F_1 of the POS tags.

MeCab (ver. 0.996) is a C++ implementation of a search-based method (Kudo et al., 2004).[5] It enumerates possible segmentations and tags as word lattices by using a dictionary and performs Viterbi search by using unigram and bigram scores factorized from feature weights.

Vaporetto (ver. 0.6.2) is a Rust[6] implementation of a classification-based method (Neubig et al., 2011).[7] It first performs word segmentation by classifying whether to segment after each character in the input, and it then identifies the resulting words' POS tags. It also trains classifiers for the possible POS tag sets of individual words, and it assigns the POSs of its first dictionary entries for words that are unseen in the training data.[8] A morphological dictionary was used to extract word features.

Jagger is a C++ implementation of the proposed algorithm. It greedily applies patterns extracted from the training data and a dictionary to jointly segment words and assign tags. Appendices A and B respectively describe the method to handle unknown words and the implementation details. Jagger is more similar to Vaporetto than to MeCab but differs in that it jointly performs segmentation and tagging instead of using a two-step cascaded pipeline, and it uses patterns instead of classifiers to find labels (i.e., where to segment and what to tag). Appendix C compares Jagger with the other implementations.

Dictionaries As listed in Table 2, the experiments used two morphological dictionaries imported to MeCab from a manually tailored morphological analyzer, JUMAN.[9] Specifically, mecab-jumandic-5.1-20070304 and mecab-jumandic-7.0-20130310 were compared to examine the impact of the dictionary's quality and size. The jumandic-

[1] In preliminary experiments, a variant of backtracking-free search (Maruyama, 1994) did not improve the throughput.

[2] `https://github.com/ku-nlp/KyotoCorpus`

[3] `https://github.com/ku-nlp/KWDLC`

[4] $C = \{0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0\}$.

[5] `https://taku910.github.io/mecab/`

[6] Rust exhibits comparable efficiency to C++ on program benchmarks: `https://github.com/kostya/benchmarks/`.

[7] `https://github.com/daac-tools/vaporetto`

[8] Words that did not appear in the dictionary were assigned "SAHEN noun," following Kudo et al. (2004). The efficiency results below do not include this postprocessing.

[9] `https://nlp.ist.i.kyoto-u.ac.jp/?JUMAN`

KYOTO	time [s] ↓	speed [sent./s] ↑	space [MiB] ↓	seg	top (level 1)	all (levels 1-4)
			w/ jumandic-5.1			
MeCab	26.83	66,455	55.81	98.68 (98.47/98.89)	97.32 (97.12/97.53)	95.97 (95.76/96.17)
Vaporetto	15.14	117,767	658.80	98.94 (98.97/98.92)	98.30 (98.32/98.27)	96.92 (96.95/96.90)
Jagger (proposed)	1.77	1,007,344	26.39	98.73 (98.62/98.83)	97.62 (97.52/97.72)	96.55 (96.45/96.65)
			w/ jumandic-7.0			
MeCab	29.99	59,453	77.98	98.37 (98.02/98.72)	97.19 (96.84/97.54)	96.10 (95.75/96.44)
Vaporetto	16.93	105,316	828.85	99.08 (99.08/99.08)	98.42 (98.42/98.43)	97.05 (97.04/97.05)
Jagger (proposed)	1.83	974,316	35.09	98.68 (98.51/98.86)	97.63 (97.46/97.80)	96.57 (96.74/96.40)

Table 3: F_1 (precision/recall) results on KYOTO.

KWDLC	time [s] ↓	speed [sent./s] ↑	space [MiB] ↓	seg	top (level 1)	all (levels 1-4)
			w/ jumandic-5.1			
MeCab	23.83	92,110	53.88	97.13 (96.82/97.44)	95.62 (95.32/95.93)	94.30 (94.00/94.60)
Vaporetto	10.93	200,823	642.63	97.35 (97.39/97.32)	96.16 (96.20/96.13)	94.08 (94.11/94.04)
Jagger (proposed)	1.44	1,524,305	28.89	97.17 (96.94/97.40)	95.71 (95.49/95.94)	94.20 (93.98/94.42)
			w/ jumandic-7.0			
MeCab	26.90	81,598	76.38	97.99 (97.82/98.16)	96.66 (96.49/96.83)	95.62 (95.45/95.78)
Vaporetto	12.55	174,900	842.40	97.53 (97.58/97.49)	96.39 (96.43/96.34)	94.68 (94.72/94.63)
Jagger (proposed)	1.46	1,503,424	40.22	97.60 (97.49/97.71)	96.14 (96.04/96.25)	94.63 (94.52/94.73)

Table 4: F_1 (precision/recall) results on KWDLC.

7.0 dictionary contains words extracted automatically from the Web (Murawaki and Kurohashi, 2008), comprising a larger number (702,358) than in jumandic-5.0 (475,716). The POS tags include four levels of hierarchical morphosyntactic information: (1) major POS (*e.g.*, *noun* and *verb*); (2) minor POS (*e.g.*, *common noun*); (3) conjugation type (*e.g.*, *ichidan verb*); and (4) conjugation form (*e.g.*, *irrealis*). For example, the POS tags of *shumi* and *iru* in Figure 1 are *noun-common_noun-*-** and *verb-*-ichidan_verb-terminal*, respectively.

Evaluation procedure The precision, recall, and F_1 of the segmentation with various levels of POS tags (Kudo et al., 2004) were used as metrics. As Vaporetto does not output lemmas, lemmatization was evaluated via the tagging results of the full POS tag set ("all (levels 1-4)" in Tables 3 and 4), which included conjugation types and forms, given that Japanese words can be mapped to their lemmas according to their conjugation types and forms. I processed 1000 copies of the test data and measured the time, speed, and maximum memory consumption three times with the `/usr/bin/time -l` command. The median values are reported here. All experiments were done on an M2 MacBook Air with a 3.5-GHz CPU and 24-GB main memory.

3.2 Results

Tables 3 and 4 summarize the morphological analysis results on the KYOTO and KWDLC datasets. The pattern-based method here, Jagger, was 16 and 7 times faster than MeCab and Vaporetto with 1/2 and 1/20 as much memory consumption, respectively, while achieving comparable accuracy. Jagger is efficient because it does not have massive floating-point parameters, unlike other methods, and because it minimizes the number and length of patterns by pruning (Lines 16-18 in Algorithm 2). As a result, the training took less than six seconds. MeCab's accuracy depends on the dictionary: with jumandic-7.0, it worked best on KWDLC and worst on KYOTO. In contrast, Vaporetto's accuracy depends on the training data size. It worked best on KYOTO but was just as good as Jagger on KWDLC.

Below are the detailed results for Jagger with the jumandic-7.0 dictionary.

Comparison to neural methods Jagger was compared to a state-of-the-art neural method (Tolmachev et al., 2018), JUMAN++-V2,[10] which was trained on the same data with the official script and hyperparameters.[11] Note that this comparison was **unfair** to Jagger in terms of accuracy and to JUMAN++-V2 in terms of efficiency, because JUMAN++-V2 uses 0.8 million additional dictionary entries from Wikipedia and a neural language model trained on 10 million sentences from the Web.

[10] https://github.com/ku-nlp/jumanpp
[11] https://github.com/ku-nlp/jumanpp-jumandic

	time [s] ↓	speed [sent./s] ↑	space [MiB] ↓	seg	top (level 1)	all (levels 1-4)
			KYOTO			
JUMAN++-V2	331.14	5384	300.80	99.37 (99.30/99.45)	98.72 (98.65/98.80)	97.74 (97.66/97.82)
Jagger (proposed)	1.83	974,316	35.09	98.68 (98.51/98.86)	97.63 (97.46/97.80)	96.57 (96.74/96.40)
			KWDLC			
JUMAN++-V2	283.11	7753	290.05	98.37 (98.25/98.50)	97.61 (97.49/97.73)	96.42 (96.30/96.55)
Jagger (proposed)	1.46	1,503,424	40.22	97.60 (97.49/97.71)	96.14 (96.04/96.25)	94.63 (94.52/94.73)

Table 5: F_1 (precision/recall) comparison with JUMAN++.

	time [s] ↓	speed [sent./s] ↑	space [MiB] ↓
		KYOTO	
MeCab	28.53	62,495	40.52
Vaporetto	4.87	366,119	283.49
Jagger (proposed)	1.41	1,264,539	21.05
		KWDLC	
MeCab	25.70	85,408	39.59
Vaporreto	4.87	366,119	283.49
Jagger (proposed)	1.13	1,942,477	20.16

Table 6: Word segmentation efficiency.

	seg	top (level 1)	all (levels 1-4)
	training: KWDLC → test: KYOTO		
MeCab	97.90	96.56	94.82
Vaporetto	95.76	93.81	91.31
Jagger (proposed)	97.25	95.42	93.30
	training: KYOTO → test: KWDLC		
MeCab	97.78	96.02	94.48
Vaporetto	97.05	95.15	92.72
Jagger (proposed)	97.22	95.01	93.12

Table 7: F_1 results for cross-domain evaluation.

Table 5 summarizes the comparison between Jagger and JUMAN++-V2. Although JUMAN++-V2 was reported to speed up JUMAN++ (Morita et al., 2015) by a factor of 250, Jagger was faster than JUMAN++-V2 by a factor of 180 with 1/7 as much of a memory footprint. JUMAN++-V2 was more accurate than Jagger, but the gain was less than 1% for word segmentation. If external text could be used, this gap could be reduced with a technique called structure compilation (Liang et al., 2008), which runs JUMAN++-V2 on external text to extract patterns. That idea is beyond this paper's scope but important for future work.

Word segmentation efficiency Because of different approaches to handling unknown words and supporting lemmatization, it is difficult to compare Vaporetto with Jagger and MeCab as a morphological analyzer in a strictly fair manner. Instead, the word segmentation efficiency was compared, as summarized in Table 6. Here, Vaporetto was trained to perform only word segmentation by using the dictionary and the training data without POS tags. Jagger was faster and more space-efficient than Vaporetto, even taking the overhead of loading large models (1.7 seconds) into account.

Cross-domain evaluation Lastly, Table 7 lists the results for cross-domain evaluation. Vaporetto's accuracy became much worse, indicating that the classification-based method was prone to overfitting to the training domain. The proposed method enjoys the benefits of the dictionary and training data: it can change its behavior by adding not only dictionary entries but also patterns.

4 Conclusions

This study sought to improve the accuracy of speed-oriented, pattern-based methods for Japanese morphological analysis, rather than improving the speed of accuracy-oriented neural models. The proposed method extracts POS-augmented patterns from a morphological dictionary and annotated data. Experimental results on two standard datasets confirmed that this method achieves accuracy comparable to that of learning-based methods, with a very fast throughput of over 1,000,000 sentences per second on a laptop.

I plan to apply this approach to other languages and even to other NLP tasks by discretizing the continuous representations induced by neural models to obtain patterns. The source code is released with GPL, LGPL, and 2-clause BSD licenses.

Message to researchers Because the accuracies on NLP benchmark datasets are becoming saturated with a larger foundation model, researchers may want to set diverse goals based on underrepresented metrics besides accuracy (*e.g.*, efficiency). I hope that this study will initiate *serious* research on speed-intensive approaches to NLP that can meet industry demands and enable researchers with limited computational resources to exert their ability.

5 Limitations

This evaluation had two limitations. First, although the method is not language-dependent, it was evaluated on a single language, Japanese. It would be worthwhile to evaluate the method on other languages to examine the approach's versatility. Second, the method uses dictionaries to obtain patterns. Although Japanese morphological analysis commonly uses dictionaries to perform lemmatization, it would be worthwhile to evaluate the method with only training data or dictionaries derived from text.

Below, I discuss the current limitations for word segmentation, POS tagging, and lemmatization in detail.

Word segmentation The proposed method's accuracy of word segmentation will depend on the target language's typological factors (Shao et al., 2018), such as the character set size, lexicon size, and average word length. Among those factors, the character set size will especially matter because the current patterns mostly comprise surface strings and are likely to suffer from data sparseness. It will thus be valuable to evaluate the method on Chinese, which has a larger character set than Japanese. It will also be important to evaluate the method on languages with different typological factors from Japanese, such as Hebrew and Finnish. The training data size will not matter if the method is used to approximate some existing resource-efficient method via structure compilation (Liang et al., 2008).

POS tagging Compared to word segmentation, POS tagging requires more complex and abstract feature sets that are tailored for the target language and POS tag set (Spoustová et al., 2009), which poses a challenge for the proposed method. The current pattern template is tailored for Japanese and the JUMAN POS tag set; hence, for other languages and POS tag sets, a pattern template will need to be designed by referring to the feature templates of existing learning-based methods for the target language and POS tag set. Because the method jointly solves word segmentation and POS tagging in a left-to-right manner, patterns cannot leverage certain abstract features from posterior contexts of the target word (*e.g.*, the next word's suffix). For application to other languages, it would be worthwhile to explore not only left-to-right processing but also right-to-left processing and a cascaded pipeline approach.

Lemmatization The approach here currently requires a morphological dictionary with lemmas or a fine-grained POS tag set that includes conjugation types and forms to perform lemmatization. Because lemma generation rules for other languages can be induced from lemma-annotated datasets (Straka, 2018), the method could be applied to other languages by using such lemma generation rules as the target labels for classification. Challenging target languages include morphologically rich languages such as Arabic and Czech.

6 Ethics Statement

I am not aware of any specific social risks that this work directly creates or exacerbates. However, because morphological analysis is a core text processing function used in various NLP applications, those who attempt to abuse NLP applications may benefit from the proposed method's efficiency.

Acknowledgements

This work was partially supported by JSPS KAKENHI Grant Number JP21H03494 and by JST, CREST Grant Number JPMJCR19A4, Japan. I thank Koichi Akabe for showing implementations of assigning POSs to unknown words in Vaporetto, Keiji Shinzato for his comments on an early draft of this paper, and Manabu Sassano for useful discussions on the future of speed-intensive NLP. Finally, I thank the anonymous reviewers for their encouraging comments on the paper's goal.

References

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

Masatsugu Hangyo, Daisuke Kawahara, and Sadao Kurohashi. 2012. Building a diverse document leads corpus annotated with semantic relations. In *Proceedings of the 26th Pacific Asia Conference on Language, Information, and Computation*, pages 535–544, Bali, Indonesia.

Julia Hockenmaier and Chris Brew. 1998. Error-driven learning of Chinese word segmentation. In *Proceedings of the 12th Pacific Asia Conference on Language, Information and Computation*, pages 218–229, Singapore.

Nobuhiro Kaji, Yasuhiro Fujiwara, Naoki Yoshinaga, and Masaru Kitsuregawa. 2010. Efficient staggered decoding for sequence labeling. In *Proceedings of* *the 48th Annual Meeting of the Association for Computational Linguistics*, pages 485–494, Uppsala, Sweden.

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.

Sadao Kurohashi and Makoto Nagao. 2003. Building a japanese parsed corpus. In Anne Abeillé, editor, *Treebanks: Building and Using Parsed Corpora*, pages 249–260. Springer Netherlands, Dordrecht.

Percy Liang, Hal Daumé, and Dan Klein. 2008. Structure compilation: Trading structure for features. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, page 592–599, New York, NY, USA. Association for Computing Machinery.

Hiroshi Maruyama. 1994. Backtracking-free dictionary access method for Japanese morphological analysis. In *COLING 1994 Volume 1: The 15th International Conference on Computational Linguistics*, Kyoto, Japan.

Hajime Morita and Tomoya Iwakura. 2019. A fast and accurate partially deterministic morphological analysis. In *Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP 2019)*, pages 804–809, Varna, Bulgaria. INCOMA Ltd.

Hajime Morita, Daisuke Kawahara, and Sadao Kurohashi. 2015. Morphological analysis for unsegmented languages using recurrent neural network language model. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 2292–2297, Lisbon, Portugal.

Yugo Murawaki and Sadao Kurohashi. 2008. Online acquisition of Japanese unknown morphemes using morphological constraints. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 429–437, Honolulu, Hawaii.

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*, Kyoto, Japan.

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.

David D. Palmer. 1997. A trainable rule-based algorithm for word segmentation. In *35th Annual Meeting of the Association for Computational Linguistics and 8th Conference of the European Chapter of the Association for Computational Linguistics*, pages 321–328, Madrid, Spain.

Manabu Sassano. 2014. Deterministic word segmentation using maximum matching with fully lexicalized rules. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics, volume 2: Short Papers*, pages 79–83, Gothenburg, Sweden.

Yan Shao, Christian Hardmeier, and Joakim Nivre. 2018. Universal word segmentation: Implementation and interpretation. *Transactions of the Association for Computational Linguistics*, 6:421–435.

Drahomíra "johanka" Spoustová, Jan Hajič, Jan Raab, and Miroslav Spousta. 2009. Semi-supervised training for the averaged perceptron POS tagger. In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*, pages 763–771, Athens, Greece. Association for Computational Linguistics.

Milan Straka. 2018. UDPipe 2.0 prototype at CoNLL 2018 UD shared task. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 197–207, Brussels, Belgium. Association for Computational Linguistics.

Arseny Tolmachev, Daisuke Kawahara, and Sadao Kurohashi. 2018. Juman++: A morphological analysis toolkit for scriptio continua. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 54–59, Brussels, Belgium.

Marcos Treviso, Tianchu Ji, Ji-Ung Lee, Betty van Aken, Qingqing Cao, Manuel R. Ciosici, Michael Hassid, Kenneth Heafield, Sara Hooker, Pedro H. Martins, André F. T. Martins, Peter Milder, Colin Raffel, Edwin Simpson, Noam Slonim, Niranjan Balasubramanian, Leon Derczynski, and Roy Schwartz. 2022. Efficient methods for natural language processing: A survey. *CoRR*, arXiv:2209.00099.

Naoki Yoshinaga and Masaru Kitsuregawa. 2009. Polynomial to linear: Efficient classification with conjunctive features. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 1542–1551, Singapore.

Naoki Yoshinaga and Masaru Kitsuregawa. 2010. Kernel slicing: Scalable online training with conjunctive features. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 1245–1253, Beijing, China. Coling 2010 Organizing Committee.

Naoki Yoshinaga and Masaru Kitsuregawa. 2014. A self-adaptive classifier for efficient text-stream processing. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pages 1091–1102, Dublin, Ireland. Dublin City University and Association for Computational Linguistics.

KYOTO	time [s] ↓	speed [sent./s] ↑	space [MiB] ↓
	w/ jumandic-5.1		
MeCab	26.83	66,455	55.81
Vibrato	12.47	142,983	97.75
Vaporetto	15.14	117,767	658.80
Jagger (proposed)	1.77	1,007,344	26.39
	w/ jumandic-7.0		
MeCab	29.99	59,453	77.98
Vibrato	16.01	111,367	164.20
Vaporetto	16.93	105,316	828.85
Jagger (proposed)	1.83	974,316	35.09

Table 8: Efficiency of morphological analysis on KYOTO; results other than for Vibrato are from Table 3.

KWDLC	time [s] ↓	speed [sent./s] ↑	space [MiB] ↓
	w/ jumandic-5.1		
MeCab	23.83	92,110	53.88
Vibrato	11.51	190,703	97.92
Vaporetto	10.93	200,823	642.63
Jagger (proposed)	1.44	1,524,305	28.89
	w/ jumandic-7.0		
MeCab	26.90	81,598	76.38
Vibrato	15.01	146,235	163.99
Vaporetto	12.55	174,900	842.40
Jagger (proposed)	1.46	1,503,424	40.22

Table 9: Efficiency of morphological analysis on KWDLC; results other than for Vibrato are from Table 4.

	time [s] ↓	speed [sent./s] ↑	space [MiB] ↓
	KYOTO		
MeCab	28.53	62,495	40.52
Vibrato	14.69	121,375	163.92
Vaporetto	4.87	366,119	283.49
Jagger (proposed)	1.41	1,264,539	21.05
SentencePiece	16.63	107,215	9.02
UTF-8 split	0.31	5,751,612	1.55
	KWDLC		
MeCab	25.70	85,408	39.59
Vibrato	13.94	157,460	164.30
Vaporreto	4.87	366,119	283.49
Jagger (proposed)	1.13	1,942,477	20.16
SentencePiece	14.54	150,962	9.05
UTF-8 split	0.27	8,129,629	1.55

Table 10: Efficiency of word segmentation (tokenization); some results are from Table 6.

A Handling of Unknown Words

Words that appear in neither the dictionary nor the training data matter in both the proposed method and search-based morphological analysis. Here, a common method (Kudo et al., 2004) was used to segment unknown words. Specifically, characters (and words) with the same character types, numbers, letters, or katakana were concatenated, with the concatenation restricted for katakana words when the total length of two katakana words exceeded a specific length (here, 18 bytes). The POS tags of concatenated unknown words were determined from a pattern based on the previous POS tag and the last concatenated word.

B Implementation Details

Implementation techniques used in the existing efficient implementations of Japanese morphological analyzers were leveraged to implement Jagger. As in MeCab, memory-mapped I/O was adopted to reduce the memory footprint, and outputs are generated by referring to strings in the in-memory dictionary while avoiding dynamic memory allocation. To maintain patterns, I used a character-wise, double-array trie that was adopted in Vaporetto and Vibrato.[12] To implement it, I modified an implementation of a byte-wise, double-array trie (Yoshinaga and Kitsuregawa, 2014), cedar.[13] The character-wise, double-array trie uses UTF-8 characters as atomic transition labels instead of UTF-8 bytes, which reduces the number of random accesses in traversing Japanese multi-byte characters. For the trie transition, UTF-8 characters in the training data are counted to obtain cache-friendly, frequency-based IDs for the UTF-8 characters. These implementation tricks provided a total speed-up factor of at most two.

Note that block I/O, which outputs results with a fixed large size (256 KiB in these experiments), is crucial to maintain the method's very fast throughput when lengthy POS tags and lemmas are output. The use of `strcpy` and `strlen` should be strictly avoided in formatting the output because they incur extra search for the terminal symbol `\0`.

C Comparison to Other Implementations

I also compared Jagger with Vibrato (ver. 0.5.0),[12] which is a recent Rust reimplementation of MeCab by the developer of Vaporetto, and SentencePiece (ver. 0.1.99),[14] which is an unsupervised text tokenizer for neural generation. SentencePiece was trained with the default options (vocabulary size of 8K) on the same training data.

Tables 8 and 9 summarize the efficiency of morphological analysis and Table 10 summarizes the ef-

[12] https://github.com/daac-tools/vibrato
[13] https://www.tkl.iis.u-tokyo.ac.jp/~ynaga/cedar/
[14] https://github.com/google/sentencepiece

ficiency of word segmentation (tokenization) with the jumandic-7.0 dictionary. Although Vibrato is twice as fast as MeCab and shows comparable speed to Vaporetto for morphological analysis, Jagger is even faster and is more space-efficient than Vibrato. Jagger's throughput is on the same order as that of UTF-8 split, which simply looks at the first bytes (byte lengths) of UTF-8 characters to segment inputs into characters. Note that SentencePiece's small memory consumption is due to its small vocabulary size of 8K: it requires more memory for a larger vocabulary.

Finally, it is noteworthy that the degree to which the processing speed is affected by the morphological dictionary's size varies from one implementation to another (Tables 8 and 9). Vibrato is the most affected by the dictionary size, whereas Jagger is the least affected.

ACL 2023 Responsible NLP Checklist

A For every submission:

☑ A1. Did you describe the limitations of your work?
5

☑ A2. Did you discuss any potential risks of your work?
6

☑ A3. Do the abstract and introduction summarize the paper's main claims?
1

☒ A4. Have you used AI writing assistants when working on this paper?
Left blank.

B ☑ Did you use or create scientific artifacts?

3

☑ B1. Did you cite the creators of artifacts you used?
3

☑ B2. Did you discuss the license or terms for use and / or distribution of any artifacts?
4

☑ B3. Did you discuss if your use of existing artifact(s) was consistent with their intended use, provided that it was specified? For the artifacts you create, do you specify intended use and whether that is compatible with the original access conditions (in particular, derivatives of data accessed for research purposes should not be used outside of research contexts)?
3

☐ B4. Did you discuss the steps taken to check whether the data that was collected / used contains any information that names or uniquely identifies individual people or offensive content, and the steps taken to protect / anonymize it?
Not applicable. Left blank.

☑ B5. Did you provide documentation of the artifacts, e.g., coverage of domains, languages, and linguistic phenomena, demographic groups represented, etc.?
3

☑ B6. Did you report relevant statistics like the number of examples, details of train / test / dev splits, etc. for the data that you used / created? Even for commonly-used benchmark datasets, include the number of examples in train / validation / test splits, as these provide necessary context for a reader to understand experimental results. For example, small differences in accuracy on large test sets may be significant, while on small test sets they may not be.
The statistics of the evaluation datasets (the number of sentences and average number of words per sentence in train/test/dev splits)

C ☑ Did you run computational experiments?

3

☑ C1. Did you report the number of parameters in the models used, the total computational budget (e.g., GPU hours), and computing infrastructure used?
3

The Responsible NLP Checklist used at ACL 2023 is adopted from NAACL 2022, with the addition of a question on AI writing assistance.

☑ C2. Did you discuss the experimental setup, including hyperparameter search and best-found hyperparameter values?
3

☐ C3. Did you report descriptive statistics about your results (e.g., error bars around results, summary statistics from sets of experiments), and is it transparent whether you are reporting the max, mean, etc. or just a single run?
Not applicable. Our pattern-based method has no fluctuation in results. The other non-neural methods compared in the main paper use convex optimization.

☑ C4. If you used existing packages (e.g., for preprocessing, for normalization, or for evaluation), did you report the implementation, model, and parameter settings used (e.g., NLTK, Spacy, ROUGE, etc.)?
3

D ☒ Did you use human annotators (e.g., crowdworkers) or research with human participants?

Left blank.

☐ D1. Did you report the full text of instructions given to participants, including e.g., screenshots, disclaimers of any risks to participants or annotators, etc.?
Not applicable. Left blank.

☐ D2. Did you report information about how you recruited (e.g., crowdsourcing platform, students) and paid participants, and discuss if such payment is adequate given the participants' demographic (e.g., country of residence)?
Not applicable. Left blank.

☐ D3. Did you discuss whether and how consent was obtained from people whose data you're using/curating? For example, if you collected data via crowdsourcing, did your instructions to crowdworkers explain how the data would be used?
Not applicable. Left blank.

☐ D4. Was the data collection protocol approved (or determined exempt) by an ethics review board?
Not applicable. Left blank.

☐ D5. Did you report the basic demographic and geographic characteristics of the annotator population that is the source of the data?
Not applicable. Left blank.