Juman++ v2: A Practical and Modern Morphological Analyzer

Arseny Tolmachev Sadao Kurohashi
Kyoto University

arseny@nlp.ist.i.kyoto-u.ac.jp, kuro@i.kyoto-u.ac.jp

1 Introduction

Japanese language has no natural delimiters between words and the first step in Japanese NLP is usually a morphological analysis, which consists of several subtasks. The most representative ones are segmentation (finding morpheme boundaries), and part of speech (POS) tagging. A morphological analyzer is useful from a practical point of view only if it is fast as well as highly accurate in its analysis.

Juman++ morphological analyzer [1] (referred also as V1), which uses a combination of a linear model and a neural network-based language model to compute a semantic plausibility of a segmentation. Juman++ has achieved state-of-the-art analysis accuracy on Jumandic (JUMAN dictionary and segmentation standard) based corpora, and drastically reduced the number of intolerable analysis errors. Unfortunately, its execution speed was extremely slow. The execution speed has limited the practical usage of Juman++.

We have reimpelmented the core idea of Juman++ and have released it as Juman++ v2[1] (referred also as V2). Our implementation is **more than 250 times faster** than V1, and achieves **better accuracy** than V1.

V2 is implemented with modern C++, with intention to be used not only as a program, but also as an embeddable library, usable in a multi-threaded environment. Additionally, V2 is not hardwired to a particular dictionary and can use partially annotated data for training. This makes Juman++ v2 a practical analyzer suitable for large-scale usage.

The reasons for the speedup come from two directions: algorithmic improvements and low level optimizations which enabled V2 to use CPUs more effectively. Both of them are equally important to the resulting speed improvement. The main low level optimizations are:

- Use struct-of-arrays object layout for frequently used objects to improve data locality and cache usage.
- Generate specialized C++ code to perform feature extraction based on a dictionary and feature templates description. The generated specialized feature extraction code drastically reduces the number of branches in the feature extraction code path and enables the use of other compiler optimizations like inlining and identical subexpression folding.
- Prefetch the linear model weights to work around the DRAM latency.
- Vectorize and batch the RNNLM.

Algorithmic improvements are discussed in more detail in the following section.

[1] https://github.com/ku-nlp/jumanpp

2 Algorithmic Improvements

Juman++ is a lattice-based analyzer. It works by assigning a score s to each path through the lattice. The path with the highest score is considered to be the analysis result. The score s consists of two components: a linear model score s^l and RNNLM score s^{RNN} which are combined as $s = s^l + \alpha(s^{\text{RNN}} + \beta)$, where α and β are scale and bias hyperparameters respectively. The RNNLM score is a log-probability score outputted by the language model.

The current version of V2 does not contain direct RNNLM-related algorithmic improvements. There are two algorithmic improvements for the linear model part and one which is shared between the both models.

2.1 Linear Model

The linear score is defined as

$$s^l = \sum_{t0 \in p} \mathbf{f}(t0, p)\mathbf{w},$$

where $\mathbf{f}(t0, p)$ is an indicator vector which contains features, extracted for a node $t0$ in a path p and $\mathbf{w}$ is a model weight vector. Weights are learned using the Soft Confidence Weighted [4] algorithm. We denote a node $t0$ if it starts on the character *boundary* we are considering at the moment. We also call $t0$ nodes *right* because they are to the right of the boundary. Nodes, which end at the boundary are referred to as *left* or $t1$. The previous nodes of $t1$s are called $t2$. See Figure 1 for the illustration.

Contents of the feature vector $\phi(t0, p)$ consist of concrete features, which are computed using the node $t0$ itself and up to two previous nodes ($t1$ and

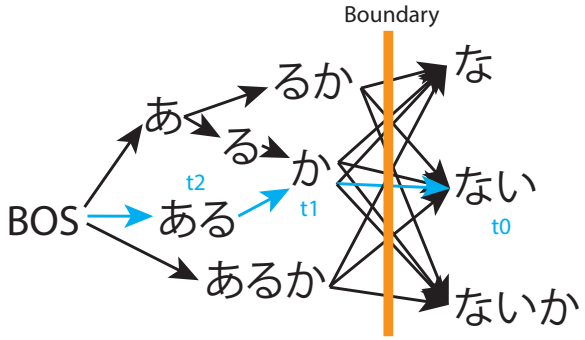


Figure 1: A lattice construction step at a character boundary.

Table 1: Dictionary and model sizes

Analyzer	Dict (MB)	Model (MB)
Raw Dictionary	256	-
MeCab	311	7.7
KyTea	-	200
V1	445	135
V2	158	16

$t2$) on the path p. These concrete linear model features are computed using *ngram feature* templates. A feature template consists of a *pattern* for each applicable node. A pattern combines several *primitive* features inside a lattice node. Those primitive features could be:

1. Dictionary fields like surface form or POS tag;
2. Surface characters;
3. Surface character types;
4. Information from unknown word extractors.

For example, a trigram template **(POS, SUB) (POS) (BASEFORM)** has combination of POS and sub-POS fields (or a **POS, SUB** pattern) for $t2$, only POS for $t1$ and a dictionary form of the node $t0$.

Patterns and primitives are combined using *hashing*, both by V1 and V2. V1 does not explicitly consider patterns and computes the whole ngram feature from the primitives each time. However, different ngram features often share patterns, and because of this the V2 computes patterns only *once for each node*.

In addition to this, paths traverse the same lattice nodes. Trigram features, for instance, would contain a different combination of ($t2$, $t1$, $t0$) for each possible path. Bigram and unigram features will be shared between some paths. V1 redundantly computes those features, but V2 does not. This removal of redundancy gave speed increase around four times.

2.2 Dictionary

Most of the current Japanese morphological analyzers store the dictionary as a row database with a trie-based index for the fast lookup. When trying to use the dictionary fields as features the analyzer needs to either perform string manipulations at runtime, which is slow, or to convert entries to unique identifiers by hashing or other means.

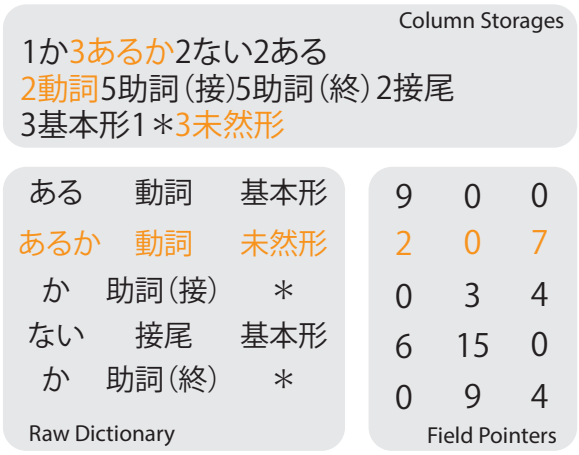


Figure 2: Dictionary as a column database. Strings in the column storage are prefixed by their lengths.

V2 instead stores the dictionary as a *column-based database*, compactly representing the dictionary field data as integers. An example is shown on Figure 2. A raw (or row-based) dictionary has its column data deduplicated and stored separately as column storage. Field entries themselves are turned into pointers into the column storage. Pointers uniquely identify field values and are used as identifiers for the entries.

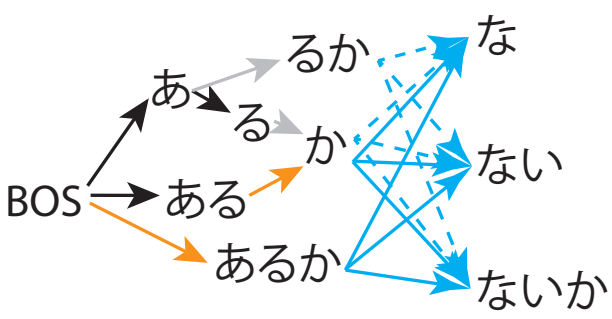
The actual dictionary implementation uses LEB128 encoding for integers. It allows to encode small numbers into small number of bytes. We exploit this fact further by sorting the column values using their frequency inside the dictionary, making the pointers to frequent entries to have smaller values and require fewer bytes to encode. This combined with the deduplication made by the column database format, causes the V2 dictionary to use significantly less space than other morphological analyzers.

The compiled dictionary and model sizes for the different morphological analyzers are shown in the table 1. All dictionaries except KyTea's [2] were built using the same conjugation-expanded Jumandic. For KyTea we used only deduplicated entries consisting of surface, POS and sub-POS fields. Actually, about 45% (71MB) of the V2 dictionary is used by the trie index, so the compression ratio of dictionary entries is very high: the compiled version uses about three times less space than the raw dictionary.

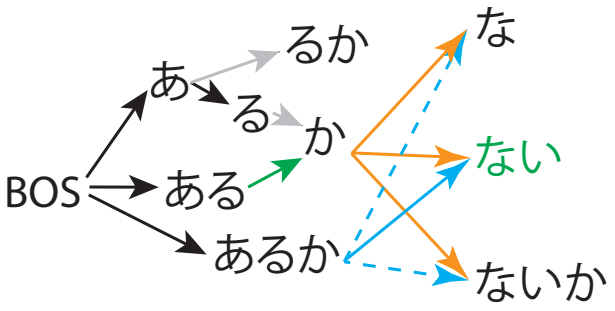
Finally, the dictionary format is memory mappable and does not require any post-processing after the dictionary load, which makes V2 startup time non-noticeable. The dictionary implementation reduced the analysis time approximately three times.

2.3 Search Space Trimming

Because Juman++ uses trigram features, the beam searching procedure has to deal with combinatorial



(a) Left global beam. $k = 2$. Top left paths are displayed in orange. Remaining paths are displayed in solid gray. Non-considered paths are displayed as dashed blue arrows. Considered paths are solid blue.



(b) Right global beam. $l = m = 1$. A top left path and a top right node is displayed in green. Orange paths via the boundary were used for scoring right nodes. Solid blue path via the boundary connects the remaining $k - l$ top left paths the top right nodes. Dashed blue paths are not considered.

Figure 3: Trimming the search space using the global beams.

explosion caused by higher order ngram features. V1 uses *local beams* of width j, meaning that only j incoming paths with top scores are kept. The rest of the paths are discarded. The problem is that those paths are discarded after evaluating their scores and the number of evaluations can still be rather large. Most of the sentences have several boundaries which have 20-30 both left and right nodes. Almost all those paths are useless and we don't want to consider them in the analysis at all.

The first improvement is to use only top k paths ending on $t1$ nodes instead of using all $t1$ paths. Connections for the remaining paths are not considered. We refer to this process as *left global beam*. The application of the left global beam is shown on Figure 3a.

The second improvement is the *right global beam*, displayed on Figure 3b. As the first substep, we use top $l(\leq k)$ $t1$ paths to compute scores of right nodes. After that, we evaluate the connections from the remaining $k - l$ left paths to the top-scored m right nodes. The rest of connections are not considered.

Figure 4 shows the F1 score on cross-validation over Kyoto University (KU) corpus when varying the global beam parameters. It can be seen that the accuracy of the models does not fall even if using very small global beam sizes. When the beam size at training is very small, the model can not correctly learn the weights for trigram parameters, lowering the accuracy on larger beam sizes. On the other hand, when the training beam size is too large, then the model losses accuracy on small beam sizes. Models which are trained with a sufficient beam size have a similar test accuracy if the test beam sizes were equal or greater than the training ones. Search space trimming gave speedup around four times with the beam parameters described in the evaluation section.

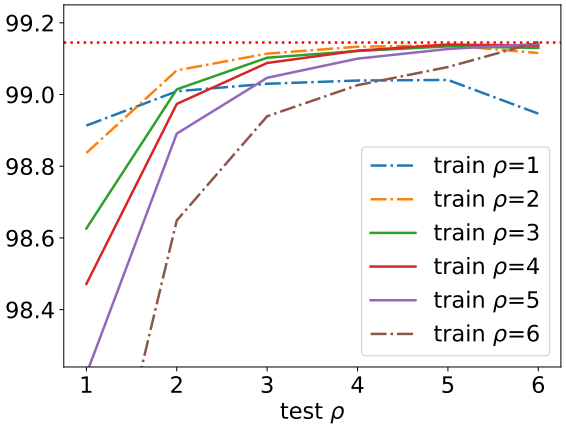


Figure 4: Cross-validation test F1 score average on KU corpus for POS tags when using different global beam parameters $k = l = \rho$, $m = 1$. Red dotted line at the top is F1 score of a model without global beams.

3 Performance Comparison

We compare both the speed and the accuracy of five different morphological analyzers: JUMAN[2], MeCab[3], KyTea, Juman++ (V1) and Juman++ (V2). For both versions of Juman++ we report results both using and not using neural network language model (noRNN).

Analysis speed We used a desktop computer with Intel i7-6850K CPU, 64GB of RAM and Ubuntu 16.04 Linux for analysis speed comparison. The models were trained from scratch using the same Juman++ dictionary, Kyoto University[4]and KWDLC[5] corpora for all morphological analyzers except JUMAN, which is not trainable. For KyTea we also

[2] http://nlp.ist.i.kyoto-u.ac.jp/index.php?JUMAN
[3] http://taku910.github.io/mecab/
[4] http://nlp.ist.i.kyoto-u.ac.jp/index.php?京都大学テキストコーパス
[5] http://nlp.ist.i.kyoto-u.ac.jp/index.php?KWDLC

Table 2: F1 scores of morphological analyzers on Jumandic-based corpora. Seg is segmentation; +Pos is correctly guessing the POS-tags after segmentation; +Sub is correctly guessing sub-POS tags (細分類) as well.

	Kyoto University			KWDLC		
Analyzer	Seg	+Pos	+Sub	Seg	+Pos	+Sub
JUMAN	98.41	97.20	95.48	98.10	97.01	95.76
KyTea	99.12	98.16	97.28	98.00	96.75	95.46
MeCab	99.14	98.58	97.62	98.28	97.61	96.23
V1 noRNN	98.94	98.42	97.06	97.66	96.95	95.51
V1 RNN	99.38	98.95	97.53	98.41	97.87	96.45
V2 noRNN	99.44	98.98	97.80	98.44	97.79	96.42
V2 RNN	**99.51**	**99.05**	**97.83**	**98.67**	**98.02**	**96.62**

Table 3: Analysis speed of morphological analyzers.

Analyzer	Speed (sents/sec)	Ratio
JUMAN	8,802	1.00
MeCab	52,410	0.17
KyTea (Jumandic)	4,892	1.79
KyTea (Unidic)	1,995	4.41
V1 noRNN	27	328.82
V1 RNN	16	535.72
V2 noRNN	7,422	1.18
V2 RNN	4,803	1.83

report the throughput using the Unidic-based models, which are available for download from the KyTea website. Jumandic-based model for KyTea was learned using default parameters. We used a concatenation of POS and sub-POS tags as the only tag for the Jumandic-based KyTea model. V1 uses local beam width $j = 5$. V2 uses $j = 5$, $k = 6$, $l = 1$, $m = 5$ beam parameters.

Table 3 shows the analysis speed of the considered morphological analyzers and speed ratio as compared to the JUMAN. The speed was measured by analyzing 50k sentences from a web corpus. Each analyzer was launched five times and the median time was used for computing the analysis speed. V2 noRNN is only 20% slower than the JUMAN, while having considerably complex model. V2 RNN has 1.8 times the execution speed of JUMAN and is more than 250 times faster than V1.

Accuracy Table 2 shows the F1 score for both KU and KWDLC corpora. A concatenation of training sections of both corpora was used to train a combined model; the reported scores are for the test sections. MeCab and V2 have hyperparameters optimized using Spearmint [3]. The V2 POS score here is lower than those on figure 4, because that experiment uses the cross-validation splitting instead of standard train/test one.

V2 RNN achieves higher F1 score than the previous SOTA of V1 RNN. Even the scores of V2 noRNN are higher in some cases than those of V1 RNN. Note that V1 noRNN score is lower than of even JUMAN, so we hypothesize that the number of training iterations of the V1 linear model was not sufficient. However it was difficult to increase it because of extremely slow analysis speed.

With the V2, we could find an optimal number of iterations for learning the linear model with the best accuracy. The other reason for improved accuracy for V2, is that compared to V1, it uses surface character and character type features. Finally, V2 treats compatible unknown words as gold nodes during the training procedure. V1 was always creating virtual nodes for words which did not have dictionary entries, ultimately not learning how to treat unknown words.

4 Conclusion and Future Work

We present Juman++ V2: a modern and practical morphological analyzer for Japanese. It achieves a new state-of-the-art accuracy for both Kyoto University and KWDLC corpora while drastically reducing the analysis time. Juman++ V2 can be used as a library, can use both fully and partially annotated data for training and is not hardwired to a specific dictionary.

We plan to create a Unidic version of Juman++ V2 and use it to annotate the readings of Jumandic-based corpora.

References

[1] Hajime Morita, Daisuke Kawahara, and Sadao Kurohashi. Morphological analysis for unsegmented languages using recurrent neural network language model. In *EMNLP*, pages 2292–2297, 2015.

[2] Graham Neubig, Yosuke Nakata, and Shinsuke Mori. Pointwise prediction for robust, adaptable japanese morphological analysis. In *ACL*, pages 529–533, 2011.

[3] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *NIPS*, pages 2951–2959, 2012.

[4] Jialei Wang, Peilin Zhao, and Steven CH Hoi. Soft confidence-weighted learning. *ACM TIST*, 8(1):15, 2016.