

Technical Report of *Japanese Morphological
Analyzer*

Jun Jiang, Vernkin Chen

September 18, 2009

Abstract

In this TR, we first present a hybrid method for Japanese word segmentation. Word-level information is useful for analysis of known words, while character-level information is useful for analysis of unknown words, and the method utilizes both these two types of information in order to effectively handle known and unknown words. Then we discuss lemma identification in Japanese morphological analysis, which is crucial for a proper formulation of morphological analysis. Since Japanese words often have variation in orthography and the vocabulary of Japanese consists of words of several different origins, it sometimes happens that more than one writing form corresponds to the same lemma and that a single writing form corresponds to two or more lemmas with different readings and/or meanings. The current study focuses on disambiguation of heteronyms, words with the same writing form but with different word forms. To resolve heteronym ambiguity, we make use of goshu information, the classification of words based on their origin. Founded on the fact that words of some goshu classes are more likely to combine into compound words than words of other classes, we employ a statistical model based on CRFs (with hybrid method) using goshu information. The hybrid method and lemma identification are employed together in this TR.

Changes

Date	Author	Notes
2009-08-28	Jun, Vernkin	Initial version
2009-08-31	Vernkin	Add Change Log Section
2009-08-31	Jun	Update project schedule section with the final refinement stage.
2009-09-01	Vernkin	Update the "Compile System Dictionary" and "Run the Demo" Section in User Guide Chapter and "User Dictionary" section in Dictionary Chapter.
2009-09-01	Jun	Add section 6.3 "Manual Comparison Result"
2009-09-11	Jun	Update section 5.2 "How to use the interface" on OPTION_TYPE_POS_FORMAT_ALPHABET in JMA API to configure POS result in alphabet format.

Contents

1	Introduction	4
2	Related Works	6
2.1	The Character Tagging Method	6
2.2	UniDic	7
2.3	Word Segmentation Using Word-Level and Character-Level Information	7
2.4	Goshu Information	8
2.5	Comparison of ChaSen and Mecab	8
2.6	Conclusion	9
3	Algorithm	10
3.1	Problem Description	10
3.2	Conditional Random Fields	11
4	Dictionary	13
4.1	Source Files of System Dictionary	13
4.2	Binary Files of System Dictionary	16
4.3	User Dictionary File	16
4.3.1	CSV User Dictionary	16
4.3.2	TXT User Dictionary	17
4.4	Dictionary Usage	17
5	User Guide	19
5.1	Interface description	19
5.2	How to use the interface	20
5.3	How to build the library	22
5.4	How to use the library	22
5.5	How to compile System Dictionary	23
5.6	How to run the demos	23
6	Experiment	25
6.1	Dictionary Statictics	25
6.2	Automatic Comparison Result	26

<i>CONTENTS</i>	3
6.3 Manual Comparison Result	27
6.4 Analysis Speed Evaluation	28
A Appendix - Project schedules and milestones	30

Chapter 1

Introduction

Automatic morphological analysis is a widely-used technique for the development of NLP systems and linguistically-annotated corpora. Morphological analysis in Chinese and Japanese is an important and difficult task. In these languages, words are not separated by explicit delimiters, and word segmentation must be conducted first in most natural language processing applications. One of the problems which makes word segmentation more difficult is existence of unknown (out-of-vocabulary) words. Unknown words are defined as words that do not exist in a system's dictionary. The word segmentation system has no knowledge about these unknown words, and determining word boundaries for such words is difficult. Accuracy of word segmentation for unknown words is usually much lower than that for known words.

In this TR, we first propose a hybrid method for Chinese and Japanese word segmentation, which utilizes both word-level and character-level information. Word-level information is useful for analysis of known words, and character-level information is useful for analysis of unknown words. We use these two types of information at the same time to obtain high overall performance.

A great amount of studies have attempted to develop software for performing automatic morphological analysis with high accuracy, and several systems for Japanese morphological analysis have been freely available (Asahara and Matsumoto, 2000[1]; Kudo et al., 2004[2]).

In traditional morphological analysis on computer, the task is divided into two sub-tasks: i) segmentation of an input string into a sequence of units, and ii) assignment of a part of speech (POS) tag to each segmented unit. Linguistic analysis of corpora, however, requires more information than one provided by these sub-tasks (see e.g., Mizutani, 1983[4]). For instance,

Japanese words often have variation in orthography; verb *arawasu* (to express) is written either as ”表わす,” ”表す,” or ”あらわす,” and noun *sakura* (a cherry blossom) either as ”桜,” ”サクラ,” or ”さくら.” In examining the frequency of words occurring in a text, linguists usually want to collapse these variants. Japanese also has lots of heteronyms¹, two or more words that have the identical writing form but different word forms. For instance, nouns *namamono* (raw food) and *seibutu* (a living thing) are both written as ”生物.” These issues are crucial for linguistic analysis of corpora, but are not handled by traditional Japanese morphological analyzers on computer. These tasks, though operating in opposite directions—one mapping two or more different writing forms onto a single word form and the other mapping one writing form onto more than one word form, can be regarded as the same problem; that is identification of lemmas, i.e., entry words in a dictionary. The task of identifying a lemma corresponding to each segmented unit in an input has been totally ignored in the study of automatic morphological analysis of Japanese.

In this paper, we also address a proper approach to Japanese morphological analysis, taking lemma identification into account. We first propose an electronic dictionary for Japanese morphological analysis, UniDic, which employs hierarchical definition of word indexes to represent orthographic variants as well as allomorphs. In this hierarchical structure, heteronyms are represented as different nodes with the same index but with different super-nodes. We then propose a statistical model for resolving heteronym ambiguity, making use of *goshu* information, the classification of words based on their origin. Founded on the fact that words of some *goshu* classes are more likely to combine into compound words than words of other classes, we employ a statistical model based on CRFs using *goshu* information.

¹An example of heteronym in English is “bow,” which has two different meanings with different sounds, /bou/ and /bau/. In this paper, writing forms refer to representation in orthography, which corresponds to spelling in English. Word forms, on the other hand, are based on kana-reading and roughly correspond to sounds, although in a few cases, e.g., particles *wa* and *e*, there is dissociation between kana-reading and sound.

Chapter 2

Related Works

2.1 The Character Tagging Method

This method carries out word segmentation by tagging each character in a given sentence, and in this method, the tags indicate word-internal positions of the characters. We call such tags position-of-character (POC) tags (Xue, 2003[9]) in this paper. Several POC-tag sets have been studied (Sang and Veenstra, 1999[6]; Sekine et al., 1998[7]), and we use the '*B, I, E, S*' tag set shown in Table 2.1¹.

Table 2.1: The '*B, I, E, S*' Tag Set

<i>B</i>	The character is in the beginning of a word.
<i>I</i>	The character is in the middle of a word.
<i>E</i>	The character is in the end of a word.
<i>S</i>	The character is itself a word.

An example of POC-Tagging:

Sentence: 窓の近くに大きな木があります。

POC Tag: 窓/B の/E 近/B く/E に/S 大/B き/I な/S 木/S が/S あ/B
り/E ま/B す/E 。/S

ThePOC-tags can represent word boundaries for any sentences, and the word segmentation task can be reformulated as the POC-tagging task. The tagging task can be solved by using general machine learning techniques

¹ The '*B, I, E, S*' tags are also called '*OP-CN, CN-CN, CNCL, OP-CL*' tags (Sekine et al., 1998[7]) or '*LL, MM, RR, LR*' tags (Xue, 2003[9]).

such as maximum entropy (ME) models (Xue, 2003[9]) and support vector machines (Yoshida et al., 2003[8]).

This character tagging method can easily handle unknown words, because known words and unknown words are treated equally and no other exceptional processing is necessary. This approach is also used in base-NP chunking (Ramshaw and Marcus, 1995[5]) and named entity recognition (Sekine et al., 1998[7]) as well as word segmentation.

2.2 UniDic

UniDic² is with the aim of providing a proper tool for Japanese morphological analysis (Den et al., 2007[10])³. The dictionary has the following features:

1. The unit for identifying a word is based on the short unit word (Maekawa, in press), which provides word segmentation in uniform size, without being harmed by too long words.
2. The indexes for words are defined at several levels, including *lemma*, *form*, and *orthography*, which enables us to represent orthographic variants as well as allomorphs.
3. An extensive amount of phonological information, such as lexical accent and sandhi, is also described and can be utilized in speech research.

2.3 Word Segmentation Using Word-Level and Character-Level Information

For the Word-Level segmentation, taking Markov model as example (And we will use CRF). It is observed that the Markov model-based method has high overall accuracy, however, the accuracy drops for unknown words, and the character tagging method has high accuracy for unknown words but lower accuracy for known words (Yoshida et al., 2003[8]; Xue, 2003[9]). This seems natural because words are used as a processing unit in the Markov model-based method, and therefore much information about known words (e.g.,

²Freely available at <http://download.unidic.org/>

³Throughout the paper, writing forms, i.e., indexes at the orthography level, are written in Japanese characters, and word forms, i.e., indexes at the form level, are written in Romaji. A lemma is expressed by a triple consisting of a standardized word form, a standardized writing form, and a meaning articulated in English.

POS or word bigram probability) can be used. However, unknown words cannot be handled directly by this method itself. On the other hand, characters are used as a unit in the character tagging method. In general, the number of characters is finite and far fewer than that of words which continuously increases. Thus the character tagging method may be robust for unknown words, but cannot use more detailed information than character-level information.

Then, we propose a hybrid method which combines the Word-Level method and the character tagging method to make the most of wordlevel and character-level information, in order to achieve high overall accuracy. The hybrid method is mainly based on word-level Markov models, but both POC-tags and POS-tags are used in the same time and word segmentation for known words and unknown words are conducted simultaneously. (More detail see Hybrid Method chapter. *TODO*)

2.4 Goshu Information

Japanese words often have variation in orthography and the vocabulary of Japanese consists of words of several different origins, it sometimes happens that more than one writing form corresponds to the same lemma and that a single writing form corresponds to two or more lemmas with different readings and/or meanings.

Hence, the mapping from a writing form onto a lemma is important in linguistic analysis of corpora. The current study focuses on disambiguation of heteronyms, words with the same writing form but with different word forms. The paper shows that for resolving heteronyms in Japanese is to make use of goshu information. Goshu is the classification of words based on their origin. And it employed Mecab using goshu-related features and Mecab (with goshu) outperform Mecab (without goshu) in the heteronym disambiguation (see section-2.5). And It seems the Mecab is a preferable referenced JMA system while developing our JMA system.

2.5 Comparison of ChaSen and Mecab

ChaSen and Mecab are popular JMA and are with goshu information (see section-2.4).

ChaSen (Asahara and Matsumoto, 2000[1]) run with the published version of UniDic (a dictionary with the aim of providing a proper tool for Japanese morphological analysis) , which employs (an extension of) a hidden Markov

model (HMM) to determine the optimal segmentation and POS assignment but can only bring a poor modeling for lemma identification, i.e., the unigram probability of lemmas given a writing form. Incorporating statistical information of goshu classes into an HMM-based analyzer, however, is problematic since in HMMs the only way to utilize goshu information is to introduce new tags that consist of combinations of POS tags and goshu classes and this will easily lead us to the data sparseness.

Mecab (Kudo et al., 2004[2]) is more recent and is based on a novel statistical method, conditional random fields (CRFs) (Lafferty et al., 2001[3]), which overcome several problems of HMMs including label bias, length bias, and difficulty in using features that can co-occur at the same position such as the POS tag and the goshu class. With the lexicon contained in ChaSen's standard dictionary and the RWCP Text Corpus as the training data, Mecab is shown to outperform ChaSen in the segmentation and POS assignment tasks.

2.6 Conclusion

From the survey, CRF are better choice than HMM and ME. Also we perform a small experiment on the CRF (from the CRF module of Jun's CMA) and the precision is 0.964 (Corpus are generated from the Mecab). And we think it is good enough for our JMA to develop based on the CRF. Also, JMA will use hybrid method and the goshu information which are discussed above to gain a higher precision than 0.964.

Chapter 3

Algorithm

3.1 Problem Description

In Japanese morphological analysis, we assume that a lexicon (or dictionary) is available, which lists a pair of a word and its corresponding part-of-speech. The lexicon gives a tractable way to build a lattice from an input sentence. A lattice represents all candidate paths or all candidate sequences of tokens, where each token denotes a word with its part-of-speech.

In the building of the lattice, if no matching word could be found in the lexicon, unknown word processing would be invoked. That is, the characters in the same type would be grouped together as a candidate token. The character types include *hiragana*, *katakana*, Chinese characters, alphabets, numbers and punctuations.

Formally, the task of Japanese morphological analysis can be defined as follows. Let \mathbf{x} be an input, unsegmented sentence. Let \mathbf{y} be a path consisting of a sequence of tokens where each token is a pair of word w_i and its part-of-speech t_i . Then their relation could be described below, where $\#\mathbf{y}$ is the number of tokens in the path \mathbf{y} :

$$\mathbf{y} = (\langle w_1, t_1 \rangle, \dots, \langle w_{\#\mathbf{y}}, t_{\#\mathbf{y}} \rangle) \quad (3.1)$$

Let $\mathcal{Y}(\mathbf{x})$ be a set of candidate paths in a lattice built from the input sentence \mathbf{x} and a lexicon. The goal is to select a correct path $\hat{\mathbf{y}}$ from all candidate paths in the $\mathcal{Y}(\mathbf{x})$.

The distinct property of Japanese morphological analysis is that the number of tokens \mathbf{y} varies, since the set of labels and the set of states are not the same.

3.2 Conditional Random Fields

To achieve the high accuracy and performance, our system uses the statistical analysis approach based on Conditional Random Fields (CRF)(Lafferty et al., 2001[3]).

CRFs are discriminative models and can thus capture many correlated features of the inputs. This allows flexible feature designs for hierarchical tagsets. CRFs have a single exponential model for the joint probability of the entire paths given the input sentence, while MEMMs consist of a sequential combination of exponential models, each of which estimates a conditional probability of next tokens given the current state. This minimizes the influences of the label and length bias.

As explained in 3.1, we use the lattice to represent the word boundary ambiguity in Japanese. This implies that the set of labels and the set of states are different, and the number of tokens $\#y$ varies according to a path.

In order to accomodate this, we define CRFs for Japanese morphological analysis as the conditional probability of an output path y given an input sequence x :

$$P(y|x) = \frac{1}{Z_x} \exp \left(\sum_{i=1}^{\#y} \sum_k \lambda_k f_k(\langle w_{i-1}, t_{i-1} \rangle, \langle w_i, t_i \rangle) \right) \quad (3.2)$$

where Z_x is a normalization factor over all candidate paths, i.e.,

$$Z_x = \sum_{y' \in \mathcal{Y}(x)} \exp \left(\sum_{i=1}^{\#y'} \sum_k \lambda_k f_k(\langle w'_{i-1}, t'_{i-1} \rangle, \langle w'_i, t'_i \rangle) \right) \quad (3.3)$$

$f_k(\langle w_{i-1}, t_{i-1} \rangle, \langle w_i, t_i \rangle)$ is an arbitrary feature function over i -th token $\langle w_i, t_i \rangle$ and its previous token $\langle w_{i-1}, t_{i-1} \rangle$. $\lambda_k (\in \Lambda = \{\lambda_1, \dots, \lambda_K\} \in \mathbb{R}^K)$ is a learned weight or parameter associated with feature function f_k .

Note that the formulation 3.2 is different from the widely-used formulation 3.4 below. The previous applications of CRFs assign a conditional probability for a label sequence $y = y_1, \dots, y_T$ given an input sequence $x = x_1, \dots, x_T$ as:

$$P(y|x) = \frac{1}{Z_x} \exp \left(\sum_{i=1}^T \sum_k \lambda_k f_k(y_{i-1}, y_i, x) \right) \quad (3.4)$$

In formulation 3.2, CRFs deal with word boundary ambiguity. Thus, the size of output sequence T is not fixed through all candidates $\mathbf{y} \in \mathcal{Y}(\mathbf{x})$. The index i is not tied with the input \mathbf{x} as in the original CRFs, but unique to the output $\mathbf{y} \in \mathcal{Y}(\mathbf{x})$.

Here, we introduce the *global feature vector* $\mathbf{F}(\mathbf{y}, \mathbf{x})$ below:

$$\mathbf{F}(\mathbf{y}, \mathbf{x}) = \{F_1(\mathbf{y}, \mathbf{x}), \dots, F_K(\mathbf{y}, \mathbf{x})\} \quad (3.5)$$

$$F_k(\mathbf{y}, \mathbf{x}) = \sum_{i=1}^{\#\mathbf{y}} f_k(\langle w_{i-1}, t_{i-1} \rangle, \langle w_i, t_i \rangle) \quad (3.6)$$

Using the above global feature vector, $P(\mathbf{y}|\mathbf{x})$ in formulation 3.2 can also be represented as:

$$P(\mathbf{y}|\mathbf{x}) = \frac{1}{Z_{\mathbf{x}}} \exp(\mathbf{\Lambda} \cdot \mathbf{F}(\mathbf{y}, \mathbf{x})) \quad (3.7)$$

The most probable path $\hat{\mathbf{y}}$ for the input sentence \mathbf{x} is then given by

$$\hat{\mathbf{y}} = \operatorname{argmax}_{\mathbf{y} \in \mathcal{Y}(\mathbf{x})} P(\mathbf{y}|\mathbf{x}) = \operatorname{argmax}_{\mathbf{y} \in \mathcal{Y}(\mathbf{x})} \mathbf{\Lambda} \cdot \mathbf{F}(\mathbf{y}, \mathbf{x}) \quad (3.8)$$

which can be found with the Viterbi algorithm.

Chapter 4

Dictionary

As explained in 3.1, we need a lexicon (or dictionary) to build the lattice in Japanese morphological analysis. So dictionary plays an important role in our JMA. That's because the character tagging method (in 2.1) cannot directly reflect lexicons which contain prior knowledge about word segmentation. We cannot ignore a lexicon since over 90% accuracy can be achieved even using the longest prefix matching with the lexicon.

Basing on the approach used in MeCab, we introduce the dictionary usage in our JMA.

4.1 Source Files of System Dictionary

The source files include *dicrc*, **.csv*, *char.def*, *unk.def*, *matrix.def*, *rewrite.def*, *left-id.def*, *right-id.def*, *pos-id.def*, which are described below respectively.

- *dicrc*

The MeCab configuration file. Table 4.1 is an example.

Table 4.1: Example of *dicrc*

config-charset = EUC-JP
dictionary-charset = EUC-JP
cost-factor = 800
bos-feature = BOS/EOS,*,*,*,*,*,*
eval-size = 8
unk-eval-size = 4

The meaning of each entry in Table 4.1 is described below.

config-charset character encoding of files *dicrc*, *char.def*, *unk.def*, *rewrite.def*, *left-id.def*, *right-id.def* and *pos-id.def*.

dictionary-charset character encoding of CSV files, used in compiling CSV files into binary file.

cost-factor scaling factor used in cost calculation.

bos-feature feature definition for sentence beginning and ending.

eval-size the number of features used in training evaluation on known words.

unk-eval-size the number of features used in training evaluation on unknown words.

- *.csv

The CSV file contains the dictionary words in text format. Multiple CSV files could be defined if only its file extension is *csv*. Each line in the file is an entry of one word. Table 4.2 is an example of CSV file from IPA dictionary.

Table 4.2: Example of CSV

行く,994,994,8852,動詞,非自立,*,*,五段	カ行促音便,基本形,行く,イク,イク
行か,1002,1002,7754,動詞,非自立,*,*,五段	カ行促音便,未然形,行く,イカ,イカ
行こ,998,998,8417,動詞,非自立,*,*,五段	カ行促音便,未然ウ接続,行く,イコ,イコ
行き,1014,1014,8036,動詞,非自立,*,*,五段	カ行促音便,連用形,行く,イキ,イキ
行け,986,986,7878,動詞,非自立,*,*,五段	カ行促音便,仮定形,行く,イケ,イケ

The meaning of each column in Table 4.2 is described below in Table 4.3.

Table 4.3: Format of CSV

column id	content example	meaning in Japanese	meaning in English
1	行く	表層形	string of the word entry
2	994	左連接状態番号	index as a left token in context state
3	994	右連接状態番号	index as a right token in context state
4	8852	コスト	word cost used in run time analysis
5	動詞	品詞	POS category in 1st level
6	非自立	品詞細分類1	POS category in 2nd level
7	*	品詞細分類2	POS category in 3rd level
8	*	品詞細分類3	POS category in 4th level
9	五段 カ行促音便	活用型	conjugation type
10	基本形	活用形	conjugation form
11	行く	基本形	base form
12	イク	読み	reading form
13	イク	発音	pronunciation form

In Table 4.3, the first four columns are mandatory for analysis. The succeeding columns are called *feature*, including those Japanese language characteristics such as POS, conjugation, etc.

The value in column 2 and 3 is the index in a context state, which index value is selected from *left-id.def* and *right-id.def* respectively. This value could also be set to -1, so that in dictionary compiling, MeCab would automatically select an appropriate index basing on those features in succeeding columns. Column 4 is a cost value of word. The smaller this value, the higher frequency this word is assumed.

- char.def

The definition file for unknown word processing, which groups characters into unknown words if they are in the same category. The characters are categorized into symbol, numeric, Kanji, Hiragana, Katakana, etc.

- unk.def

The feature definition file for the character categories defined in *char.def*. The format of this file is similar with that of CSV file.

- matrix.def

The definition file for the connection matrix. Each value in the matrix represents a cost value for connecting adjacent tokens, which cost value would be used in CRF analysis.

- rewrite.def

The definition file for feature selection. In this definition file, the features are selected or rewritten for the CRF analysis. For example, columns from 5 to 12 in Table 4.3 could be selected as features in the definition. Besides, as Japanese words often have variation in orthography, such as verb *kuru* (to come) is written either as “来る” or “くる”, by means of this definition file, the unified form “来る” could be selected as the base form in the features.

- left-id.def & right-id.def

The definition file for feature index, which index value is used as column 2 and 3 in Table 4.3.

- pos-id.def

The definition file for POS index. In this file, index number is defined for each POS string. If this file is not defined, the POS index of each word would be set to 1 defaultly.

4.2 Binary Files of System Dictionary

Also it is called the Compiled System Dictionary. After compiling, some of the dictionary source files are converted to binary type, which are used for run time analysis. These binary files are listed below.

- sys.dic

The binary file compiled from those CSV files **.csv*.

- char.bin

The binary file compiled from *char.def* and *unk.def*.

- unk.dic

The binary file compiled from *unk.def*.

- matrix.bin

The binary file compiled from *matrix.def*.

- pos-feature.def

The Simple POS maps to feature. This file are with "simplePOS = feature" format. The Simple POS used in the User Dictionary are only allowed appeared in this file. If give wrong Simple POS, the Program only print out warning and reset the Simple POS to default POS.

4.3 User Dictionary File

User dictionary are CSV files or TXT (text) files.

4.3.1 CSV User Dictionary

If user dictionary are CSV files, its format is the same as that in Table 4.3. Unlike the CSV files in system dictionary, which need to be compiled into a binary file *sys.dic* for run time using, multiple user CSV files could be loaded in run time directly. Table 4.4 is an example of user CSV file.

In Table 4.4, we could see that column 2 and 3 are set to -1. In this case, please confirm that the features in succeeding columns has a corresponding index value in file *left-id.def* and *right-id.def*, otherwise MeCab would not be able to select an appropriate index value for this entry. User could also adjust the cost value in column 4 with a small integer for those high frequency words.

Table 4.4: Example of User Dictionary

真人,-1,-1,0,名詞,固有名詞,人名,名,*,*,真人,マサト,マサト
藤樹,-1,-1,0,名詞,固有名詞,人名,名,*,*,藤樹,トウジュ,トージュ
香与子,-1,-1,0,名詞,固有名詞,人名,名,*,*,香与子,カヨコ,カヨコ
千紘,-1,-1,0,名詞,固有名詞,人名,名,*,*,千紘,チヒロ,チヒロ

4.3.2 TXT User Dictionary

The TXT User Dictionary are more flexible than CSV User Dictionary. Each entry is in one line. The simple POS see pos-feature.def in each Compiled System Dictionary (see 4.2).

Each entry can be:

- the word itself. Like "真人". The word would be given the default Simple POS, which is set by attribute "default-pos" in the file dicrc under Compiled System Dictionary.
- the word and the one or more Simple POSs, these tokens are separated by spaces, like "真人 N V".
- The same as entry in the CSV file, like "真人,-1,-1,0,名詞,固有名詞,人名,名,*,*,真人,マサト,マサト".
- the word and the completed feature, like "真人 -1,-1,0,名詞,固有名詞,人名,名,*,*,真人,マサト,マサト" (The first comma becomes space).

4.4 Dictionary Usage

The CSV files in the source directory of system dictionary are editable so that user could revise words and add new words as well. When CSV files are modified or new CSV file is added, the source files of system dictionary need to be compiled into binary type for run time loading. The CSV files as user dictionary could be loaded at run time directly without compiling. The details of these files have been described in previous sections.

JMA supports different character encodings such as *EUC-JP* and *SHIFT-JIS*. The run time encoding type should be the same with that of system dictionary in binary type. A specific encoding type could be configured and used like below steps.

1. Set character encoding of system CSV files.

That is, set the entry **dictionary-charset** with value *EUC-JP* or *SHIFT-JIS* in *dicrc* file, which file is under source directory of system dictionary. The example of *dicrc* file is shown in Table 4.1, which file in IPA dictionary could be available as */jma/db/ipadic/src/dicrc*.

2. Compile system dictionary source files to binary type in run time encoding.

Below is an example of compiling system dictionary files into run time encoding of *EUC-JP*.

```
$ cd bin
$ mkdir ../db/ipadic/bin_eucjp
$ ./jma_encode_sysdict --encode eucjp ../db/ipadic/src ../db/ipadic/bin_eucjp
```

In the above example, the source files of system dictionary are in directory *../db/ipadic/src*. They are compiled to binary files in directory *../db/ipadic/bin_eucjp*.

Before executing program *jma_encode_sysdict*, please ensure that the directory for binary files *../db/ipadic/bin_eucjp* already exists. The value of command option *--encode* in this program could be *eucjp* or *sjis*, which would be the character encoding type on JMA run time.

3. Set character encoding of user CSV file.

This step is similar with step 1, except with a different *dicrc* file.

That is, set the entry **dictionary-charset** with value *EUC-JP* or *SHIFT-JIS* in *dicrc* file, which file is under the directory of system dictionary in binary type, instead of the directory in source type. That is because at JMA run time, only the binary directory of system dictionary and user CSV file would be loaded.

The directory of IPA dictionary in binary type could be available as */jma/db/ipadic/bin_eucjp*, and the example file of user CSV file could be */jma/db/userdic/ipa_eucjp.csv*. Both of them are in *EUC-JP* encoding type.

4. Load the dictionaries at JMA run time analysis.

At run time, before loading dictionary files, call JMA library interface *Knowledge::setEncodeType()* to set the character encoding type. The parameter¹ of this interface should be the same with the encoding type used in step 2.

Then call interfaces *Knowledge::setSystemDict()* and *Knowledge::addUserDict()* to set dictionary paths, and call interface *Knowledge::loadDict()* to load those dictionary files. The interface calling example could be found in section 5.2.

¹ENCODE.TYPE.EUCJP, ENCODE.TYPE.SJIS, or ENCODE.TYPE.UTF8

Chapter 5

User Guide

5.1 Interface description

The C++ API of the library is described below:

Class JMA_Factory creates instances for CMA, its methods below create instances of JMA_Factory, Analyzer and Knowledge, which are used in the morphological analysis.

```
static CMA_Factory* instance();  
virtual Analyzer* createAnalyzer() = 0;  
virtual Knowledge* createKnowledge() = 0;
```

Class Knowledge manages the linguistic information, its methods below load files of system dictionary, user dictionary, stop-word dictionary and configuration files.

```
virtual int setSystemDict(const char* dirPath) = 0;  
virtual int addUserDict(const char* fileName) = 0;  
virtual int loadDict() = 0;  
virtual int loadStopWordDict(const char* fileName) = 0;  
virtual int loadSentenceSeparatorConfig(const char* fileName) = 0;  
virtual int loadConfig(const char* fileName) = 0;  
  
void setEncodeType(EncodeType type);  
virtual int encodeSystemDict(const char* txtFileName, const char* binFileName) = 0;
```

By means of Knowledge::setEncodeType(), we could set the character encode as EUC-JP, SHIFT-JIS or UTF-8. If this function is not called, the default encode is EUC-JP. Knowledge::encodeSystemDict() is the interface of encoding system dictionary from text type to the binary one.

Class Analyzer executes the morphological analysis, its methods below execute the morphological analysis based on a sentence, a paragraph or a file separately. Analyzer::splitSentence() splits a paragraph into sentences.

```
virtual void setKnowledge(Knowledge* pKnowledge) = 0;
virtual void setOption(OptionType nOption, double nValue);

virtual int runWithSentence(Sentence& sentence) = 0;
virtual const char* runWithString(const char* inStr) = 0;
virtual int runWithStream(const char* inFileName, const char* outFileName) = 0;

virtual void splitSentence(const char* paragraph, std::vector<Sentence>& sentences) = 0;
```

Class Sentence saves the analysis results of Analyzer::runWithSentence(), so that the n-best or one-best results could be accessed.

```
void setString(const char* pString);
const char* getString(void) const;
int getListSize(void) const;
int getCount(int nPos) const;
const char* getLexicon(int nPos, int nIdx) const;
const char* getBaseForm() const;
int getPOS(int nPos, int nIdx) const;
const char* getStrPOS(int nPos, int nIdx) const;
double getScore(int nPos) const;
int getOneBestIndex(void) const;
```

5.2 How to use the interface

The interface could be used like the following steps:

1. Include the header files in directory include.

```
#include "jma_factory.h"
#include "analyzer.h"
#include "knowledge.h"
#include "sentence.h"
```

2. Use the library name space.

```
using namespace jma;
```

3. Call the interface and handle the result.

In the example below, the return value of some functiona calls are not handled for simplicity, please properly handle those return values denoting failures in your using.

```

// create instances
CMA_Factory* factory = CMA_Factory::instance();
Analyzer* analyzer = factory->createAnalyzer();
Knowledge* knowledge = factory->createKnowledge();

// (optional) set character encode type as SJIS,
// if this function is not called, the default encode is EUC-JP.
knowledge->setEncodeType(Knowledge::ENCODE_TYPE_SJIS);

// set the path of solely system dictionary
knowledge->setSystemDict("..."); // must be and only invoke once
// (optional) add the paths of several user dictionaries
knowledge->addUserDict("..."); // for user dictionary No.1
knowledge->addUserDict("..."); // for user dictionary No.2
// load system and user dictionaries
knowledge->loadDict();

// (optional) load stop word dictionary
knowledge->loadStopWordDict("...");

// (optional) load iJMA general configuration file
knowledge->loadConfig("...");
// (optional) load the sentence separator configuration file
knowledge->loadSentenceSeparatorConfig("...");

// (optional) if POS tagging is not needed, call the function below
analyzer->setOption(Analyzer::OPTION_TYPE_POS_TAGGING, 0);

// (optional) output POS result in alphabet format such like "NP-S", instead of in Japanese format
analyzer->setOption(Analyzer::OPTION_TYPE_POS_FORMAT_ALPHABET, 1);

// (optional) set the number of N-best results,
// if this function is not called, one-best analysis is
// performed defaultly on Analyzer::runWithSentence().
analyzer->setOption(Analyzer::OPTION_TYPE_NBEST, 5);

// set knowledge
analyzer->setKnowledge(knowledge);

// 0. split paragraphs into sentences
string line;
vector<Sentence> sentVec;
while(getline(cin, line)) // get paragraph string from standard input
{
    sentVec.clear(); // remove previous sentences
    analyzer->splitSentence(line.c_str(), sentVec);
    for(size_t i=0; i<sentVec.size(); ++i)
    {
        analyzer->runWithSentence(sentVec[i]); // analyze each sentence
        ...
    }
}

// 1. analyze a sentence
Sentence s;
s.setString("...");
analyzer->runWithSentence(s);

// get one-best result
int i= s.getOneBestIndex();
...

```

```

// get n-best results
for(int i=0; i<s.getListSize(); ++i)
{
    for(int j=0; j<s.getCount(i); ++j)
    {
        const char* pLexicon = s.getLexicon(i, j);
        // get the base form of the specific Japanese word
        const char* baseForm = s.getBaseForm(i, j);
        const char* strPOS = s.getStrPOS(i, j);
        ...
    }
    double score = s.getScore(i);
    ...
}

// 2. analyze a paragraph
const char* result = analyzer->runWithString("...");
...

// 3. analyze a file
analyzer->runWithStream("...", "...");

// destroy instances
delete knowledge;
delete analyzer;

```

5.3 How to build the library

The project uses CMake as the build system, it could be built using script `build.sh` in directory `build` like below.

```

$ cd build
$ ./build.sh [release | debug | clean]

```

In the above build, the release or debug version could be built with command option `release` or `debug`. Without command option, the default build version is `release`. The built files could be removed with command option `clean`.

After the project is built, the library targets `libjma.a` and `libmecab.so` are created in directory `lib`.

Those executables in directory `bin` are used for demo, test, and also training statistical model.

5.4 How to use the library

To use the library, the user application needs to include header files in directory `include`, link the library files `libjma.a`, `libmecab.so` in directory `lib`.

An example of compiling user application `test.cpp` looks like:


```
$ export JMA_PATH=path_of_jma_project
$ g++ -I$JMA_PATH/include -o test test.cpp $JMA_PATH/lib/libjma.a \
    $JMA_PATH/lib/libmecab.so -Wl,-rpath,$JMA_PATH/lib
```

5.5 How to compile System Dictionary

The System Dictionary Source (in plain text format, with csv or def as suffix name) must be compiled first before JMA can use it.

There are three existence dictionaries: ipa, juman and uni. Under the directory "db" in the Project's root directory, the three directories ipadic, jumandic and unidic are named for those three dictionaries separately. We call three directories Dictionary Directory .

Under each Dictionary Directory, "src" directory is the source of the associated dictionary, and "bin_****" is the Compiled System Dictionary . "****" here indicates the encoding of the compiled dictionary. There are three encodings now: euc-jp, shift-jis and utf-8.

The executable file "bin/jma_encode_sysdict" is used to compile System Dictionary Source.

Take the ipadic for example, the working directory is "bin" directory. Compile the ipadic for three encoding separately:

```
./jma_encode_sysdict --encode eucjp ../db/ipadic/src ../db/ipadic/bin_eucjp
./jma_encode_sysdict --encode sjis ../db/ipadic/src ../db/ipadic/bin_sjis
./jma_encode_sysdict --encode utf8 ../db/ipadic/src ../db/ipadic/bin_utf8
```

5.6 How to run the demos

To run the demos, please make sure the project is built, and also at least one system dictionary are compiled by the guidance above.

The demo file are "bin/jma_run". And usage for this file are:

The working directory for all the code in this section are "bin" directory.

```
./jma_run --sentence N-best [--dict DICT_PATH]
./jma_run --string [--dict DICT_PATH]
./jma_run --stream INPUT OUTPUT [--dict DICT_PATH]
```

The DICT_PATH is the path compiled System Dictionary, like "../db/ipadic/bin_eucjp" (also this is the default one).

Demo of analyzing 5-best results of a sentence string from standard input.

```
$ ./jma_run --sentence 5
```

Demo of analyzing a paragraph string from standard input.

```
$ ./jma_run --string
```

To exit the loop in the above demos, please press CTRL-C.

Demo of analyzing file from SOURCE to DEST (the example of SOURCE file could be available as `../db/test/test_raw_1.txt`).

```
$ ./jma_run --stream SOURCE DEST
```

Chapter 6

Experiment

We compared the result of our JMA with two other JMAs. One is Basis¹, the other is ChaSen². We also used three dictionaries separately in our JMA, which are introduced below firstly.

6.1 Dictionary Staticstics

Two of the dictionaries are called IPA and Juman, which are in the source package of MeCab³. The other dictionary is called UniDic⁴.

These dictionaries are built from different corpora, where each corpus has a different POS tagset and word segmentation standard. For example, the Juman dictionary is built from Kyoto University Corpus⁵. The staticstics of these dictionaries are listed in Table 6.1.

Table 6.1: Dictionary Statistics

Dictionary	#words	POS structure	#POS categories
IPA	392,033	4-levels	69
Juman	516,003	2-levels	43
UniDic	559,239	4-levels	53

The column of POS structure in Table 6.1 means the number of POS levels.

¹<http://www.basistech.com>

²<http://chasen.naist.jp>

³<http://mecab.sourceforge.net>

⁴<http://www.tokuteicorpus.jp/dist>

⁵<http://www-lab25.kuee.kyoto-u.ac.jp/nl-resource/corpus.html>

For example, in Table 4.3, there are four levels in POS category. And the last column in Table 6.1 gives the number of POS categories at the bottom level.

Regarding the dictionary used in Basis, from its online document⁶, it is said that there are about 600,000 words in its Japanese dictionary. And the POS structure of Basis is 1-level containing 29 POS categories.

By comparing the POS structure, we could see that the POS category number in our dictionaries is much more than that of Basis. So that the morpheme results in our JMA is more rigorous than Basis on Japanese grammar, which comparison result could also be seen in the section below.

6.2 Automatic Comparison Result

In the automatic comparison, we used Basis and ChaSen as the base JMA to compare with, which means that the result of the base JMA would be assumed as the standard answer.

Then we use precision, recall and balanced F-score as the comparison measure. Precision (P) is defined as the number of correctly segmented words divided by the total number of words in our segmentation result. Recall (R) is defined as the number of correctly segmented words divided by the total number of words in the standard answer. F-score (F) is defined as below:

$$F = \frac{P \times R \times 2}{P + R} \quad (6.1)$$

The test data is the raw text extracted from the news website⁷, which data size is 806K bytes. By using this test data, we got the F-score result in Table 6.2, in which table the numbers are those F-scores between the base JMA and our JMA using those three dictionaries respectively.

Table 6.2: Comparison F-score

Base JMA	IPA	Juman	UniDic
Basis	0.830	0.858	0.806
ChaSen	0.948	0.811	0.896

From the comparison result, the highest F-score is 0.948, which is between

⁶<http://www.basistech.com/data-sheets/basis-asian-text-analysis.pdf>

⁷<http://www.asahi.com>

ChaSen and our JMA with IPA dictionary. This is because ChaSen also uses the same source from IPA dictionary. As each dictionary gives its own definition on what should be a word, we could see that the comparison result relies on the dictionary largely.

Compared with Basis, the F-score is above 0.8, although Basis dictionary is very different from ours. By comparing the analysis result manually, we found that our result is more rigorous than Basis on Japanese grammar.

For example, from the raw input text “もらえます”, the analysis result of Basis is one word “もらえます/Verb 動詞”, while our result on IPA dictionary contains two separated words “もらえ/Verb 動詞,自立” and “ます/Particle 助動詞”, which result is more rigorous on Japanese grammar, and giving more details in word boundary and POS information.

By comparing the total number of segmented words in the analysis results, the number of Basis result is about ten percentage less than ours. So we could see that our JMA could give more details on morphological information.

6.3 Manual Comparison Result

We also compared the precisions of our JMA (iJMA) with Basis by manual check. That is, the incorrect analysis results are picked out by users who could read Japanese.

We used IPA dictionary on iJMA in this manual check. The test data is the raw text extracted from web sites⁸ by users randomly. The comparison result is summarized in Table 6.3.

Table 6.3: Precision Comparison by Manual Check

	Token Number	Same Percentage	Precision
Basis	14,970	84.85%	95.99%
iJMA	16,498	76.99%	94.18%

In above table, the 1st column is the total number of tokens in each JMA’s output. We could see that the token number of iJMA is ten percentage more than that of Basis.

The 2nd column means how many tokens in one JMA’s output are the same with that of the other JMA. By using the formulation 6.1, the comparison F-score between Basis and iJMA is 0.807.

⁸www.zakzak.co.jp, www.yahoo.co.jp, www.goo.ne.jp, etc

The last column gives the precision value, meaning how many tokens are correct in user’s manual check. In the review of manual check results, we found that those incorrect results could be mainly categorized into two situations.

One situation is caused by lacking of the word in dictionary. For example, as the word “有元” (used as Japanese surname) does not exist in IPA dictionary, it is segmented into two words “有” and “元” incorrectly.

The other situation is caused by different segmentation standards between IPA dictionary and user’s manual check. For example, the evaluator assumes “高さ” (meaning highness in English) is a whole word. While in iJMA result, it is separated into two words “高” (meaning high) and “さ” (meaning ness). So we could see that both results could be correct on its own segmentation standard. Regarding iJMA with IPA dictionary, its segmentation standard is more rigorous than that of Basis.

Both of these situations could be improved by adding those words into iJMA dictionary. For example, after words “有元” and “高さ” are added into a user dictionary loaded in iJMA, they are output as whole words.

So we could see that iJMA could be improved by expanding or adjusting the dictionary. The dictionary usage in iJMA could be found in section 4.4.

6.4 Analysis Speed Evaluation

The test environment for analysis speed evaluation is described in Table 6.4.

Table 6.4: Test environment

Platform	Ubuntu x86_32 Kernel Linux 2.6.28-14
Memory	3.7GB
CPU	Intel Core 2 Duo 2.33GHz

In case of 1 Megabyte raw input file, the execution time of word segmentation is evaluated on Basis, ChaSen, and our JMA with those three dictionaries respectively. The statistics is listed in Table 6.5.

Table 6.5: Time statistics (seconds) on 1 Megabyte file

Basis	ChaSen	IPA	Juman	UniDic
6.83	0.68	0.76	1.43	1.2

In the time statistics result, ChaSen is the fastest one. The analysis speed of our JMA with IPA dictionary is very close to that of ChaSen, which uses the same dictionary source. The speed of our JMA relies on the dictionary size and unknown word processing rules.

From the above statistics, we could see that our JMA is faster than Basis about 5 to 9 times.

Appendix A

Appendix - Project schedules and milestones

APPENDIX A. APPENDIX - PROJECT SCHEDULES AND MILESTONES31

Milestone			Start	Finish	In Charge	Description	Status
1	Survey on the project		2008-05-04	2008-05-15	Vernkin	1. Have a general overview of current Japanese Segmentation techniques and their differences. 2. Select a suitable one and do more research on it.	Finished
2	Experiment using CMA approach		2009-05-11	2009-05-15	Jun	Experiment CMA approach using only character-level information, which result reached 0.96 on ASahi text. To improve the result further for JMA, we would adopt the hybrid approach, which utilizes both word-level and character-level information.	Finish
2	Analyze requirement specification, and study the potential solutions		2009-05-18	2009-05-22	Vernkin	Analyze requirements, and study the hybrid approach.	Finish
3	Implement the Japanese morphological analyzer		2009-05-25	2009-06-26	Jun	Implement an initial version of this project.	Finish
4	Experiment and debugging		2009-06-29	2009-07-31	Vernkin	Experiment the system on training data and test data, and debugging the system.	Finish
5	Compare iJMA with basis		2009-08-03	2009-08-31	Vernkin	Build a web platform for JMAs comparison, so that user could compare and manual check the results.	Finish
6	Improve iJMA basing on comparison feedback, finish TR and doxygen document		2009-09-01	2009-09-25	Jun	Improve iJMA by means of the user feedback from Wisenut. It could be done by enhance iJMA dictionary and optimize code. Then integrate the documents available into TR and generate doxygen documents.	On going

Bibliography

- [1] Masayuki Asahara and Yuji Matsumoto. Extended models and tools for high-performance part-of-speech tagger. *Proceedings of the 18th International Conference on Computational Linguistics*, pages 21–27, 2000.
- [2] Taku Kudo, Kaoru Yamamoto, and Yuji Matsumoto. Applying conditional random fields to japanese morphological analysis. *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, pages 230–237, 2004.
- [3] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.
- [4] Shizuo Mizutani. Vocabulary (in japanese). *Asakura New Series on Japanese*, 2, 1983.
- [5] Lance Ramshaw and Mitch Marcus. Text chunking using transformation-based learning. *Proceedings of the 3rd Workshop on Very Large Corpora*, pages 88–94, 1995.
- [6] Erik F. Tjong Kim Sang and Jorn Veenstra. Representing text chunks. *Proceedings of 9th Conference of the European Chapter of the Association for Computational Linguistics*, pages 173–179, 1999.
- [7] Ralph Grishman Satoshi Sekine and Hiroyuki Shinnou. A decision tree method for finding and classifying names in japanese texts. *Proceedings of the 6th Workshop on Very Large Corpora*, pages 171–177, 1998.
- [8] Kiyonori Ohtake Tatsumi Yoshida and Kazuhide Yamamoto. Performance evaluation of chinese analyzers with support vector machines. *Natural Language Processing*, 10(1):109–131, 2003.
- [9] Nianwen Xue. Chinese word segmentation as character tagging. *International Journal of Computational Linguistics and Chinese*, 8(1):29–48, 2003.

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