Regular Paper

Efficient Classification with Conjunctive Features

Naoki Yoshinaga[1,a) Masaru Kitsuregawa[1]

Received: April 11, 2011, Accepted: July 8, 2011

Abstract: This paper proposes a method that speeds up a classifier trained with many conjunctive features: combinations of (primitive) features. The key idea is to precompute as partial results the weights of primitive feature vectors that represent fundamental classification problems and appear frequently in the target task. A prefix tree (trie) compactly stores the primitive feature vectors with their weights, and it enables the classifier to find for a given feature vector its longest prefix feature vector whose weight has already been computed. Experimental results on base phrase chunking and dependency parsing demonstrated that our method speeded up the svm and llm classifiers by a factor of 1.8 to 10.6.

Keywords: conjunctive feature, log-linear model, support vector machine, base phrase chunking, dependency parsing

1. Introduction

In the information-explosion era, researchers in the field of natural language processing (nlp) and data mining (dm) have demonstrated that processing more texts promotes better science [16], [23], [43], [47]. Deep and accurate text analysis based on discriminative models is, however, not yet efficient enough to process Web-scale corpora for knowledge acquisition [5], [38], [39], [40] or semi-supervised learning [1], [3], [8], [29], [31], [44] even with distributed computing environments [38], [41]; typically, syntactic parsing such as dependency parsing and supertagging is orders of magnitudes slower than the front-end part-of-speech (pos) tagging [15], [20], [39], which forces us to limit the size of web texts to make processing feasible.

One of the main reasons for this inefficiency is attributed to the inefficiency of core classifiers trained with many feature combinations (e.g., word *n*-grams). Hereafter, we refer to features that explicitly represent combinations of features as *conjunctive features* and the other atomic features as *primitive features*. The feature combinations play an essential role in obtaining a classifier with state-of-the-art accuracy for several nlp tasks; recent examples include morphological analysis [33], dependency parsing [24], parse re-ranking [31], named-entity recognition [29], pronoun resolution [34], and semantic role labeling [30]. However, 'explicit' feature combinations significantly increase the feature space, which slows down not only training but also testing of the classifier.

ℓ_1-regularized log-linear models (ℓ_1-llms) provide sparse solutions, in which weights of irrelevant features are exactly zero as a result of assuming a Laplacian prior on the weights [46], [49]. However, Kazama and Tsujii [22] have reported in a text categorization task that most features regarded as irrelevant during the training of ℓ_1-llms appeared rarely in the task. In such a case, ℓ_1-regularization cannot greatly reduce the number of active features in each classification, while retaining the classification accuracy. We later confirm this in a dependency parsing task.

Kernel-based methods such as support vector machines (svms), on the other hand, consider feature combinations space-efficiently by using a polynomial kernel function [7]. The kernel-based classification is, however, known to be very slow in nlp tasks, so efficient classifiers should sum up the weights of the explicit conjunctive features as in llms [14], [17], [26]. In the end, when efficiency is a major concern, we must use exhaustive feature selection [26], [36], [50] or even restrict the order of conjunctive features at the expense of accuracy.

In this study, we provide a simple, effective solution to the inefficiency of classifiers trained with higher-order conjunctive features (or polynomial kernel), by exploiting the Zipfian nature of language data. The key idea is to precompute the weights of primitive feature vectors, which represent fundamental classification problems in the task, and use them as partial results to compute the weight of a given feature vector. We maintain primitive feature vectors and their pre-calculated weights in a trie called the *feature sequence trie* to quickly find for a given feature vector its longest prefix feature vector whose weight has been computed. The trie is built from feature vectors generated by applying the classifier to actual data in the task. The time complexity of the classifier approaches time that is linear with respect to the number of active primitive features when the retrieved feature vector covers most of the features in the input feature vector.

We implemented our algorithm for svm and llm classifiers and evaluated the performance of the resulting classifiers in a base phrase chunking task and a dependency parsing task. Experimental results show that it successfully speeded up classifiers trained with conjunctive features by a factor of up to 10.

The rest of this paper is organized as follows. Section 2 introduces llms and svms. Section 3 proposes our classification algorithm. Section 4 presents experimental results. Section 5

[1] Institute of Industrial Science, the University of Tokyo
[a)] ynaga@tkl.iis.u-tokyo.ac.jp

concludes with a summary and addresses future directions.

2. Preliminaries

In this paper, we focus on linear classifiers that calculate the probability (or margin) by summing up weights of individual features. Examples include not only LLMs but also SVMs with kernel expansion [14], [17], [26]. Below, we introduce these two classifiers and the ways that they treat feature combinations.

In classification-based NLP, the target task is modeled as one or more classification steps. For example in POS tagging, each classification decides whether to assign a particular *label* (POS tag) to a given *example* (each word in a given sentence). Each example is then represented by a *feature vector* $\boldsymbol{x}$, whose element x_i is a value of a feature function $f_i \in \mathcal{F}$.

Here, we assume a binary feature function $f_i(\boldsymbol{x}) \in \{0, 1\}$, in which a non-zero value means that particular context data appears in the example. We say that a feature f_i is *active* in example $\boldsymbol{x}$ when $x_i = f_i(\boldsymbol{x}) = 1$ and $|\boldsymbol{x}|$ represents the number of active features in $\boldsymbol{x}$ ($|\boldsymbol{x}| = |\{f_i \mid f_i(\boldsymbol{x}) = 1\}|$).

2.1 Log-Linear Models

The log-linear model (LLM), or also known as maximum-entropy model [4], is a linear classifier widely used in the NLP literature. Let the training data $\mathcal{D}$ of LLMs be $\{\langle \boldsymbol{x}_i, y_i\rangle\}_{i=1}^{|\mathcal{D}|}$, where $\boldsymbol{x}_i \in \{0, 1\}^n$ is a feature vector and y_i is a class label associated with $\boldsymbol{x}_i$.

The classifier provides conditional probability $p(y \mid \boldsymbol{x})$ for a given feature vector $\boldsymbol{x}$ and label y:

$$p(y \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \exp \sum_i w_{i,y} f_{i,y}(\boldsymbol{x}, y), \qquad (1)$$

where $f_{i,y}(\boldsymbol{x}, y)$ is a feature function that returns a non-zero value when $f_i(\boldsymbol{x}) = 1$ ($f_i \in \mathcal{F}$) and the label is y, $w_{i,y} \in \mathbb{R}$ is a weight associated with $f_{i,y}$, and $Z(\boldsymbol{x})$ is the partition function defined as:

$$Z(\boldsymbol{x}) = \sum_y \exp \sum_i w_{i,y} f_{i,y}(\boldsymbol{x}, y).$$

We can consider feature combinations in LLMs by explicitly introducing a new conjunctive feature $f_{\mathcal{F}',y}(\boldsymbol{x}, y)$ that is activated when a particular set of features $\mathcal{F}' \subseteq \mathcal{F}$ to be combined is activated (namely, $f_{\mathcal{F}',y}(\boldsymbol{x}, y) = \bigwedge_{f_i \in \mathcal{F}'} f_{i,y}(\boldsymbol{x}, y)$) [*1].

We then introduce an ℓ_1-regularized LLM (ℓ_1-LLM), in which the weight vector $\boldsymbol{w}$ is tuned so as to maximize the logarithm of the a posteriori probability of the training data:

$$\begin{aligned} \boldsymbol{w}^\star &= \operatorname*{argmax}_{\boldsymbol{w}} \mathcal{L}_{\boldsymbol{w}} \\ \text{where} \quad \mathcal{L}_w &= \sum_{i=1}^{|\mathcal{D}|} \log p(y_i \mid \boldsymbol{x}_i) - C\|\boldsymbol{w}\|_1 \end{aligned} \qquad (2)$$

Hyper-parameter C thereby controls the degree of over-fitting (solution sparseness). Interested readers may refer to the cited literature [48] for the optimization procedures.

To the best of our knowledge, there are no previous reports of an *exact* weight calculation faster than linear summation for an LLM (Eq. (1)).

2.2 Support Vector Machines

A support vector machine (SVM) is a binary classifier [7]. Training with examples $\{\langle \boldsymbol{x}_i, y_i\rangle\}_{i=1}^{|\mathcal{D}|}$ where $\boldsymbol{x}_i \in \{0, 1\}^n$ and $y_i \in \{\pm 1\}$ yields the following decision function:

$$\begin{aligned} & y(\boldsymbol{x}) = \mathrm{sgn}(m(\boldsymbol{x}) + b) \\ & \text{where} \quad m(\boldsymbol{x}) = \sum_{\boldsymbol{x}_j \in \mathcal{S}} \alpha_j \boldsymbol{\phi}(\boldsymbol{x}_j)^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}), \end{aligned} \qquad (3)$$

where $b \in \mathbb{R}$, $\boldsymbol{\phi} : \mathbb{R}^n \mapsto \mathbb{R}^H$ and *support vectors* $\boldsymbol{x}_j \in \mathcal{S}$ (*support set*, a subset of training examples), each of which is associated with weight $\alpha_j \in \mathbb{R}$. We hereafter call $m(\boldsymbol{x})$ the *weight function*. The nonlinear mapping function $\boldsymbol{\phi}$ is chosen to make the training examples linearly separable in $\mathbb{R}^H$ space. Kernel function $k(\boldsymbol{x}_j, \boldsymbol{x}) = \boldsymbol{\phi}(\boldsymbol{x}_j)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x})$ is then introduced to compute the dot product in $\mathbb{R}^H$ space without mapping $\boldsymbol{x}$ to $\boldsymbol{\phi}(\boldsymbol{x})$.

To consider combinations of primitive features $f_j \in \mathcal{F}$, we use a *polynomial kernel* $k_d(\boldsymbol{x}_j, \boldsymbol{x}) = (\boldsymbol{x}_j^{\mathrm{T}}\boldsymbol{x} + 1)^d$. From Eq. (3), we obtain the weight function for the polynomial kernel as:

$$m(\boldsymbol{x}) = \sum_{\boldsymbol{x}_j \in \mathcal{S}} \alpha_j k_d(\boldsymbol{x}_j, \boldsymbol{x}) = \sum_{\boldsymbol{x}_j \in \mathcal{S}} \alpha_j (\boldsymbol{x}_j^{\mathrm{T}}\boldsymbol{x} + 1)^d. \qquad (4)$$

Since we assumed that x_i is a binary value representing whether a (primitive) feature f_i is active in the example, the polynomial kernel k_d implies a mapping $\boldsymbol{\phi}_d$ from $\boldsymbol{x}$ to $\boldsymbol{\phi}_d(\boldsymbol{x})$ that has $H = \sum_{k=0}^{d} \binom{n}{k}$ dimensions. Each dimension represents a (weighted) conjunction of d features in the original example $\boldsymbol{x}$ [*2].

The time complexity of Eq. (4) is $O(|\mathcal{S}||\boldsymbol{x}|)$. This cost is usually high for classifiers used in NLP tasks because they often have many support vectors ($|\mathcal{S}| > 10{,}000$).

2.2.1 Kernel Inverted

Kudo and Matsumoto [26] proposed *polynomial kernel inverted* (PKI), which builds inverted indices $h_{\mathcal{S}}(f_i) = \{\boldsymbol{x}_j \mid \boldsymbol{x}_j \in \mathcal{S}, f_i \in \boldsymbol{x}_j\}$ from each feature f_i to support vector $\boldsymbol{x}_j \in \mathcal{S}$ to only consider support vector $\boldsymbol{x}_j$ relevant to a given feature vector $\boldsymbol{x}$ such that $\boldsymbol{x}_j^{\mathrm{T}}\boldsymbol{x} \neq 0$. The time complexity of PKI is $O(B \cdot |\boldsymbol{x}| + |\mathcal{S}|)$ where $B \equiv \frac{1}{|\boldsymbol{x}|}\sum_{f_j \in \boldsymbol{x}} |h_{\mathcal{S}}(f_j)|$, which is smaller than $O(|\mathcal{S}||\boldsymbol{x}|)$ if $\boldsymbol{x}$ has many rare features f_i such that $|h_{\mathcal{S}}(f_i)| \ll |\mathcal{S}|$.

2.2.2 Kernel Expansion

Isozaki and Kazawa [17] and Kudo and Matsumoto [26] proposed *polynomial kernel expanded* (PKE) to convert Eq. (4) into the linear sum of the weights in the mapped feature space as in LLM ($p(y \mid \boldsymbol{x})$ in Eq. (1)):

$$m(\boldsymbol{x}) = \left(\sum_{\boldsymbol{x}_j \in \mathcal{S}} \alpha_j \boldsymbol{\phi}_d(\boldsymbol{x}_j)\right)^{\mathrm{T}} \boldsymbol{\phi}_d(\boldsymbol{x}) = \sum_{i : x_i^d = 1} w_i, \qquad (5)$$

where $\boldsymbol{x}^d \in \{0, 1\}^H$ is a binary feature vector whose element x_i^d has a non-zero value when $(\boldsymbol{\phi}_d(\boldsymbol{x}))_i \neq 0$, $\boldsymbol{w}$ is the weight vector for $\boldsymbol{x}^d$ in the expanded feature space $\mathcal{F}^d$ and is precalculated from the support vectors $\boldsymbol{x}_j$ and their weights α_j:

[*1] Precisely speaking, we can kernelize an LLM (with Gaussian prior) to implicitly consider conjunctive features by using the polynomial kernel function (as in Section 2.2).

[*2] For example, given an input vector $\boldsymbol{x} = (x_1, x_2)^{\mathrm{T}}$ and a support vector $\boldsymbol{x}' = (x_1', x_2')^{\mathrm{T}}$, the 2nd-order polynomial kernel returns $k_2(\boldsymbol{x}', \boldsymbol{x}) = (x_1'x_1 + x_2'x_2 + 1)^2 = 3x_1'x_1 + 3x_2'x_2 + 2x_1'x_1x_2'x_2 + 1$ ($\because x_i', x_i \in \{0, 1\}$). This function thus implies a mapping $\boldsymbol{\phi}_2(\boldsymbol{x}) = (1, \sqrt{3}x_1, \sqrt{3}x_2, \sqrt{2}x_1x_2)^{\mathrm{T}}$. In the following argument, we ignore the dimension of the constant in the mapped space and assume constant b is set to include it.

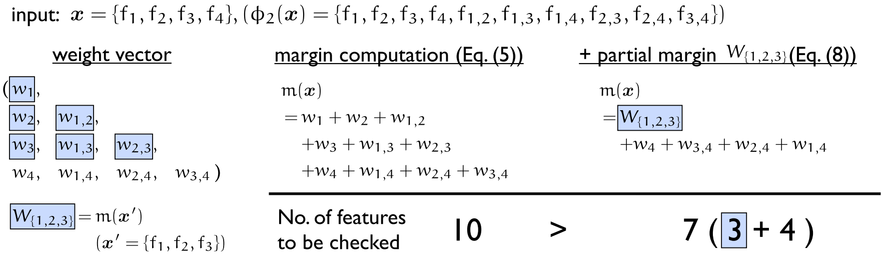
input: $\boldsymbol{x} = \{f_1, f_2, f_3, f_4\}, (\phi_2(\boldsymbol{x}) = \{f_1, f_2, f_3, f_4, f_{1,2}, f_{1,3}, f_{1,4}, f_{2,3}, f_{2,4}, f_{3,4}\})$
weight vector
$(\boxed{w_1},$
$\boxed{w_2}, \boxed{w_{1,2}},$
$\boxed{w_3}, \boxed{w_{1,3}}, \boxed{w_{2,3}},$
$w_4, w_{1,4}, w_{2,4}, w_{3,4})$
$\boxed{W_{\{1,2,3\}}} = m(\boldsymbol{x}')$
$(\boldsymbol{x}' = \{f_1, f_2, f_3\})$
margin computation (Eq. (5))
$m(\boldsymbol{x}) = w_1 + w_2 + w_{1,2} + w_3 + w_{1,3} + w_{2,3} + w_4 + w_{1,4} + w_{2,4} + w_{3,4}$
+ partial margin $W_{\{1,2,3\}}$ (Eq. (8))
$m(\boldsymbol{x}) = \boxed{W_{\{1,2,3\}}} + w_4 + w_{3,4} + w_{2,4} + w_{1,4}$
No. of features to be checked
10
>
7 ($\boxed{3}$ + 4)

Fig. 1 Efficient computation of $m(\boldsymbol{x})$.

$$\boldsymbol{w} = \sum_{\boldsymbol{x}_j \in \mathcal{S}} \alpha_j \sum_{k=0}^{d} c_d^k I_k(\boldsymbol{x}_j^d). \tag{6}$$

where c_d^k is a squared coefficient of k-th order conjunctive features for d-th order polynomial kernel (e.g., $c_2^0 = 1$, $c_2^1 = 3$, and $c_2^2 = 2$)[*3] and $I_k(\boldsymbol{x}_j^d)$ is $\boldsymbol{x}_j^d \in \{0,1\}^H$ whose dimensions other than those of k-th order conjunctive features are set to zero. The time complexity of Eq. (5) (and Eq. (1)) is $O(|\boldsymbol{x}|^d)$, which is linear with respect to the number of active features in $\boldsymbol{x}^d$ within the expanded feature space $\mathcal{F}^d$.

Since kernel expansion demands a huge memory volume to store the weight vector, $\boldsymbol{w}$, in $\mathbb{R}^H$ ($H = \sum_{k=0}^{d} \binom{|\mathcal{F}|}{k}$), it is unrealistic to maintain explicit conjunctive features for higher-order conjunctive features. To make the weight vector sparse, Kudo and Matsumoto [26] proposed an approximation method that filters out less useful features whose absolute weight values are less than a pre-defined threshold σ[*4]. They reported that increased threshold value σ resulted in a dramatically sparse feature space $\mathcal{F}^d$, which had the side-effects of accuracy degradation and classifier speed-up.

2.2.3 Kernel Splitting

To cope with memory explosion, Goldberg and Elhadad [14] only explicitly considered conjunctions among features $f_C \in \mathcal{F}_C$ that commonly appear in support set $\mathcal{S}$, and they handled the other conjunctive features relevant to rare features $f_R \in \mathcal{F} \setminus \mathcal{F}_C$ by using the polynomial kernel:

$$\begin{aligned} m(\boldsymbol{x}) &= m(\tilde{\boldsymbol{x}}) + m(\boldsymbol{x}) - m(\tilde{\boldsymbol{x}}) \\ &= \sum_{f_i \in \tilde{\boldsymbol{x}}^d} \tilde{w}_i + \sum_{\boldsymbol{x}_j \in \mathcal{S}_R} \alpha_j k'_d(\boldsymbol{x}_j, \boldsymbol{x}, \tilde{\boldsymbol{x}}), \end{aligned} \tag{7}$$

where $\tilde{\boldsymbol{x}}$ is $\boldsymbol{x}$ whose dimensions of rare features are set to zero, $\tilde{\boldsymbol{w}}$ is a weight vector computed with Eq. (6) for $\mathcal{F}_C^d$, and $k'_d(\boldsymbol{x}_j, \boldsymbol{x}, \tilde{\boldsymbol{x}})$ is defined as:

$$\begin{aligned} k'_d(\boldsymbol{x}_j, \boldsymbol{x}, \tilde{\boldsymbol{x}}) &\equiv k_d(\boldsymbol{x}_j, \boldsymbol{x}) - k_d(\boldsymbol{x}_j, \tilde{\boldsymbol{x}}) \\ &= (\boldsymbol{x}_j^{\mathrm{T}} \boldsymbol{x} + 1)^d - (\boldsymbol{x}_j^{\mathrm{T}} \tilde{\boldsymbol{x}} + 1)^d. \end{aligned}$$

We can space-efficiently compute the first term of Eq. (7) since $|\tilde{\boldsymbol{w}}| \ll |\boldsymbol{w}|$, whereas we can quickly compute the second term of Eq. (7) since $k'_d(\boldsymbol{x}_j, \boldsymbol{x}, \tilde{\boldsymbol{x}}) = 0$ when $\boldsymbol{x}_j^{\mathrm{T}} \boldsymbol{x} = \boldsymbol{x}_j^{\mathrm{T}} \tilde{\boldsymbol{x}}$; we only need to consider a small subset of the support set, $\mathcal{S}_R = \bigcup_{f_R \in \boldsymbol{x} \setminus \tilde{\boldsymbol{x}}} h_{\mathcal{S}}(f_R)$, that has at least one of the rare features f_R appearing in $\boldsymbol{x} \setminus \tilde{\boldsymbol{x}}$ ($|\mathcal{S}_R| \ll |\mathcal{S}|$). Parameter r is set to determine $\mathcal{F}_C = \{f_i \,|\, |h_{\mathcal{S}}(f_i)| \geq r\}$.

Although they referred to this computation as splitsvm, we hereafter refer to this computation as *polynomial kernel splitting* (pks) since it does not depend on svms.

3. Proposed Method

In this section, we propose a method that speeds up a classifier trained with many conjunctive features. Below, we focus on a kernel-based classifier trained with a polynomial kernel of degree d (here, svms), but an analogous argument is possible for linear classifiers (e.g., llms)[*5].

We hereafter represent a binary feature vector $\boldsymbol{x}$ as a set of active features $\{f_i \,|\, f_i(\boldsymbol{x}) = 1\}$. $\boldsymbol{x}$ can thereby be represented as an element of the power set $2^{\mathcal{F}}$ of the set of features $\mathcal{F}$.

3.1 Idea

Let us remember that weight function $m(\boldsymbol{x})$ in Eq. (5) maps $\boldsymbol{x} \in 2^{\mathcal{F}}$ to $W \in \mathbb{R}$. If we could calculate $W_{\boldsymbol{x}} = m(\boldsymbol{x})$ for all possible $\boldsymbol{x}$ in advance, we could obtain $m(\boldsymbol{x})$ by simply checking $|\boldsymbol{x}|$ elements, namely, in $O(|\boldsymbol{x}|)$ time. However, because $|\{\boldsymbol{x} \,|\, \boldsymbol{x} \in 2^{\mathcal{F}}\}| = 2^{|\mathcal{F}|}$ and $|\mathcal{F}|$ is likely to be very large (often $|\mathcal{F}| > 10{,}000$) in nlp tasks, this calculation is impractical.

We then compute and store weight $W_{\boldsymbol{x}'} = m(\boldsymbol{x}')$ for $\boldsymbol{x}' \in \mathcal{X}_c (\subset 2^{\mathcal{F}})$, a certain subset of the possible value space, and compute $m(\boldsymbol{x})$ for $\boldsymbol{x} \notin \mathcal{X}_c$ by using precalculated weight $W_{\boldsymbol{x}_c}$ for $\boldsymbol{x}_c \subseteq$[*6] $\boldsymbol{x}$ in the following way:

$$m(\boldsymbol{x}) = W_{\boldsymbol{x}_c} + \sum_{f_i \in \boldsymbol{x}^d \setminus \boldsymbol{x}_c^d} w_i. \tag{8}$$

Intuitively speaking, starting from partial weight $W_{\boldsymbol{x}_c}$, we add up remaining weights of primitive features $f \in \mathcal{F}$ that are not active in $\boldsymbol{x}_c$ but are active in $\boldsymbol{x}$ and conjunctive features that combine f and the other active features in $\boldsymbol{x}$.

An example of this computation ($d = 2$) is depicted in **Fig. 1**. We can efficiently compute $m(\boldsymbol{x})$ for a vector $\boldsymbol{x}$ that has four active features f_1, f_2, f_3, and f_4 (and $\boldsymbol{x}^2$ has their six conjunctive features) using precalculated weight $W_{\{1,2,3\}}$; we should first check the three features f_1, f_2, and f_3 to retrieve $W_{\{1,2,3\}}$ and next check the remaining four features related to f_4, namely f_4, $f_{1,4}$, $f_{2,4}$, and

[*3] Following Lemma 1 in Ref. [26],

$$c_d^k = \sum_{l=k}^{d} \binom{d}{l} \left(\sum_{m=0}^{k} (-1)^{k-m} \cdot m^l \binom{k}{m} \right).$$

[*4] Precisely speaking, they set different thresholds for positive ($\alpha_j > 0$) and negative ($\alpha_j < 0$) support vectors, considering the proportion of positive and negative support vectors.

[*5] When a feature vector $\boldsymbol{x}$ includes (explicit) conjunctive features $f \in \mathcal{F}^d$, we assume weight function $m'(\boldsymbol{x}') = m(\boldsymbol{x})$, where $\boldsymbol{x}'$ is a projection of $\boldsymbol{x}$ (by $\boldsymbol{\phi}_d^{-1} : \mathcal{F}^d \mapsto \mathcal{F}$).

[*6] This means that all active features in $\boldsymbol{x}_c$ are active in $\boldsymbol{x}$.

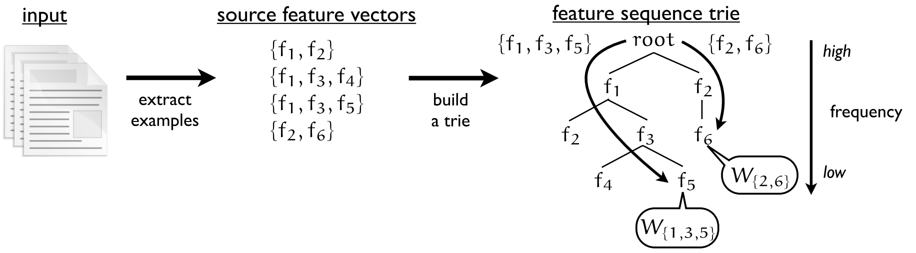

Fig. 2 Feature sequence trie and completion of prefix feature vector weights.

$f_{3,4}$, in order to add up the remaining weights, while the normal computation in Eq. (5) should check the four primitive and six conjunctive features to get the individual weights.

Counting the number of features $f(\boldsymbol{x}, \boldsymbol{x}_c, d)$ to be checked in the computation, we obtain the time complexity of Eq. (8) as:

$$f(\boldsymbol{x}, \boldsymbol{x}_c, d) = |\boldsymbol{x}_c| + |\boldsymbol{x}^d| - |\boldsymbol{x}_c^d| \in O(|\boldsymbol{x}|^d - |\boldsymbol{x}_c|^d), \quad \text{where} \quad |\boldsymbol{x}^d| = \sum_{k=1}^{d} \binom{|\boldsymbol{x}|}{k} \qquad (9)$$

(e.g., $|\boldsymbol{x}^2| = \frac{|\boldsymbol{x}|^2+|\boldsymbol{x}|}{2}$ and $|\boldsymbol{x}^3| = \frac{|\boldsymbol{x}|^3+5|\boldsymbol{x}|}{6}$)[*7]. Note that when $|\boldsymbol{x}_c|$ becomes close to $|\boldsymbol{x}|$, this time complexity actually approaches $\Omega(|\boldsymbol{x}|)$.

Thus, to minimize this computational cost, $\boldsymbol{x}_c$ is to be chosen from $\mathcal{X}_c$ as follows:

$$\boldsymbol{x}_c = \operatorname*{argmin}_{\boldsymbol{x}' \in \mathcal{X}_c, \boldsymbol{x}' \subseteq \boldsymbol{x}} (|\boldsymbol{x}'| + |\boldsymbol{x}^d| - |\boldsymbol{x}'^d|). \qquad (10)$$

3.2 Construction of Feature Sequence Trie

There are two issues with speeding up the classifier by the computation shown in Eq. (8). First, since we can store weights for only a small fraction of possible feature vectors (namely, $|\mathcal{X}_c| \ll 2^{|\mathcal{F}|}$), we should choose $\mathcal{X}_c$ so as to maximize its impact on the speed-up. Second, we should quickly find an optimal $\boldsymbol{x}_c$ from $\mathcal{X}_c$ for a given feature vector $\boldsymbol{x}$.

The solution to the first problem is to enumerate partial feature vectors that frequently appear in the target task. Note that typical linguistic features used in NLP tasks usually consist of disjunctive sets of features (e.g., word surface and POS), in which the sets are likely to follow Zipf's law [52] and correlate with each other. We can expect the distribution of feature vectors, the mixture of Zipf distributions, to be Zipfian. This has been confirmed for word n-grams [12] and itemset support distribution [6]. We can thus expect that a small set of partial feature vectors will commonly appear in the task.

To solve the second problem, we introduce a *feature sequence trie* (*fstrie*), which represents a hierarchy of feature vectors, to enable the classifier to efficiently retrieve (sub-)optimal $\boldsymbol{x}_c$ (in Eq. (10)) for a given feature vector $\boldsymbol{x}$.

We build an fstrie in the following steps:

Step 1: Apply the target classifier to actual (raw) data in the task to enumerate possible feature vectors (hereafter, *source feature vectors*).

Step 2: Sort the features in each source feature vector according to their frequency in the training data (in descending order).

Step 3: Build a trie from the source feature vectors by regarding feature indices as characters and store weights of all prefix feature vectors.

An fstrie built from four source feature vectors is shown in **Fig. 2**. In fstries, a path from the root to another node represents a feature vector. An important point here is that the fstrie stores the weights of all prefix feature vectors of the source feature vectors, and the trie structure enables us to retrieve for a given feature vector $\boldsymbol{x}$ the weight of its longest prefix vector $\boldsymbol{x}_c \subseteq \boldsymbol{x}$ in $O(|\boldsymbol{x}_c|)$ time. To handle feature functions in LLMS (Eq. (1)), we store partial weight $W_{\boldsymbol{x}_c,y} = \sum_i w_{i,y} f_{i,y}(\boldsymbol{x}_c, y)$ for each label y on the node that expresses $\boldsymbol{x}_c$.

Since we sort the features in the source feature vectors according to their frequency, the prefix feature vectors exclude less frequent features in the source feature vectors. Lexical features or finer-grained features (e.g., POS-subcategory) are usually less frequent than coarse-grained features (e.g., POS), so they lie in the latter part of the feature vectors. This sorting helps us to retrieve longer feature vector $\boldsymbol{x}_c$ for input feature vector $\boldsymbol{x}$ that will have diverse infrequent features. It also minimizes the size of fstrie by sharing the common frequent prefix (e.g., $\{f_1, f_3\}$ in Fig. 2).

Pruning nodes from fstrie We have so far described the way to construct an fstrie from the source feature vectors. However, a naive enumeration of source feature vectors will result in the explosion of the fstrie size, and we want to have a principled way to control the fstrie size rather than reducing the processed data size. Below, we present a method that prunes useless prefix feature vectors (nodes) from the constructed fstrie to maximize its impact on the classifier efficiency.

We adopt a greedy strategy that iteratively prunes a leaf node (one prefix feature vector and its weight) from the fstrie built from all the source feature vectors, according to a certain utility score calculated for each node. In this study, we consider two metrics for each prefix feature vector $\boldsymbol{x}_c$ to calculate its utility score.

Probability $p(\boldsymbol{x}_c)$, which denotes how often the stored weight $W_{\boldsymbol{x}_c}$ will be used in the target task. The maximum-likelihood estimation provides probability:

$$p(\boldsymbol{x}_c) = \frac{\sum_{\boldsymbol{x}' \supseteq \boldsymbol{x}_c} n_{\boldsymbol{x}'}}{\sum_{\boldsymbol{x}} n_{\boldsymbol{x}}},$$

where $n_{\boldsymbol{x}} \in \mathbb{N}$ is the frequency count of a source feature vector $\boldsymbol{x}$ in the processed data.

Computation reduction $\Delta_d(\boldsymbol{x}_c)$, which denotes how much computation is reduced by $W_{\boldsymbol{x}_c}$ to calculate a weight of $\boldsymbol{x} \supseteq \boldsymbol{x}_c$. This can be estimated by counting the number of conjunctive features we additionally have to check when we

[*7] This is the maximum number of conjunctive features.

```
Input: fstrie $\mathcal{W}$, node_limit $N \in \mathbb{N}$
Output: fstrie $\mathcal{W}$
1: while no. of nodes in $\mathcal{W} > N$ do
2:     $x_c \leftarrow \mathrm{argmin}_{x' \in leaf(\mathcal{W})} u(x')$
3:     remove $x_c$, $\mathcal{W}$
4: end while
5: return $W$
```

Fig. 3 Pruning nodes from fstrie.

```
Input: weight vector $\boldsymbol{w} \in \mathbb{R}^{|\mathcal{F}^d|}$, fstrie $\mathcal{W}$,
    feature vector $\boldsymbol{x} \in 2^{\mathcal{F}}$
Output: weight $W = m(\boldsymbol{x}) \in \mathbb{R}$
1: $\boldsymbol{x} \leftarrow \mathrm{sort}(\boldsymbol{x})$
2: $\langle \boldsymbol{x}_c, W_{\boldsymbol{x}_c} \rangle \leftarrow$ prefix_search($\mathcal{W}$, $\boldsymbol{x}$)
3: $W \leftarrow W_{\boldsymbol{x}_c}$
4: for all features $f_i \in \boldsymbol{x}^d \setminus \boldsymbol{x}_c^d$ do
5:     $W \leftarrow W + w_i$
6: end for
7: return $W$
```

Fig. 4 Computing weight with fstrie.

remove $\boldsymbol{x}_c$. Since the fstrie stores the weight of a prefix feature vector $\boldsymbol{x}_{c^-} \subset \boldsymbol{x}_c$ such that $|\boldsymbol{x}_{c^-}| = |\boldsymbol{x}_c| - 1$ (e.g., in Fig. 2, $\boldsymbol{x}_{c^-} = \{f_1, f_3\}$ for $\boldsymbol{x}_c = \{f_1, f_3, f_4\}$), we can define the computation reduction as:

$$\Delta_d(\boldsymbol{x}_c) = (|\boldsymbol{x}_c^d| - |\boldsymbol{x}_{c^-}^d|) - (|\boldsymbol{x}_c| - |\boldsymbol{x}_{c^-}|)$$
$$= \sum_{k=2}^{d} \binom{|\boldsymbol{x}_c|}{k} - \sum_{k=2}^{d} \binom{|\boldsymbol{x}_c| - 1}{k} \ (\because \text{ Eq. (9)}).$$

$\Delta_2(\boldsymbol{x}_c) = |\boldsymbol{x}_c| - 1$ and $\Delta_3(\boldsymbol{x}_c) = \frac{|\boldsymbol{x}_c|^2 - |\boldsymbol{x}_c|}{2}$.

We calculate the utility score of each node $\boldsymbol{x}_c$ in the fstrie as $u(\boldsymbol{x}_c) = p(\boldsymbol{x}_c) \cdot \Delta_d(\boldsymbol{x}_c)$, which means the expected computation reduction by $\boldsymbol{x}_c$ in the task, and prune the lowest-utility-score leaf nodes from the fstrie one by one (**Fig. 3**). If several prefix vectors have the same utility score, we eliminate them in numerical descending order.

3.3 Classification Algorithm

Our classification algorithm is shown in detail in **Fig. 4**. The classifier first sorts the active features in input feature vector $\boldsymbol{x}$ according to their frequency in the training data. Then, for $\boldsymbol{x}$, it retrieves the longest common prefix vector $\boldsymbol{x}_c$ from the fstrie (line 2 in Fig. 4). It then adds the weights of the remaining features to partial weight $W_{\boldsymbol{x}_c}$ (line 5 in Fig. 4).

Note that the remaining features whose weights we sum up (line 4 in Fig. 4) are primitive and conjunctive features that relate to $f \in \boldsymbol{x} \setminus \boldsymbol{x}_c$, which appear less frequently than $f' \in \boldsymbol{x}_c$ in the training data. Thus, when we apply our algorithm to classifiers with the sparse solution (*e.g.*, ℓ_1-LLMS or SVMS that filter out less useful features), $|\boldsymbol{x}^d| - |\boldsymbol{x}_c^d|$ can be much smaller than the theoretical expectation (Eq. (9)). We confirmed this in the following experiments.

Figure 5 depicts a classification algorithm that accommodates the kernel splitting described in Section 2.2. The algorithm changes its behavior depending on whether or not the prefix feature vector $\boldsymbol{x}_c$ covers all the common features $\tilde{\boldsymbol{x}} = \boldsymbol{x} \cap \mathcal{F}_C$: if $\boldsymbol{x}_c$ does not cover all the common features $\tilde{\boldsymbol{x}}$, it uses $\tilde{\boldsymbol{w}}$ to compute the partial weight $W_{\tilde{\boldsymbol{x}}}$. We then compute the remaining weights regarding features that are not covered by either $\boldsymbol{x}_c$ or $\tilde{\boldsymbol{x}}$ (line 13 in Fig. 5). Note that the line 8 in Fig. 5 implicitly assumes $\tilde{\boldsymbol{x}} \supset \boldsymbol{x}_c$ (namely, $\tilde{\boldsymbol{x}}^d \supset \boldsymbol{x}_c^d$). To satisfy this assumption, we choose the common features $\mathcal{F}_C$ according to their frequencies in the training examples, while Goldberg and Elhadad [14] chose $\mathcal{F}_C$ according to their frequencies in the subset of training examples (support set $\mathcal{S}$).

```
Input: common features $\mathcal{F}_C \subseteq \mathcal{F}$, weight vector $\tilde{\boldsymbol{w}} \in \mathbb{R}^{|\mathcal{F}_C^d|}$,
    fstrie $\mathcal{W}$, feature vector $\boldsymbol{x} \in 2^{\mathcal{F}}$
Output: weight $W = m(\boldsymbol{x}) \in \mathbb{R}$
 1: $\boldsymbol{x} \leftarrow \mathrm{sort}(\boldsymbol{x})$
 2: $\langle \boldsymbol{x}_c, W_{\boldsymbol{x}_c} \rangle \leftarrow$ prefix_search($\mathcal{W}$, $\boldsymbol{x}$)
 3: $W \leftarrow W_{\boldsymbol{x}_c}$
 4: $\tilde{\boldsymbol{x}} \leftarrow \boldsymbol{x} \cap \mathcal{F}_C$
 5: if $\tilde{\boldsymbol{x}} \subseteq \boldsymbol{x}_c$ then
 6:     $\tilde{\boldsymbol{x}} \leftarrow \boldsymbol{x}_c$
 7: else // remaining common features
 8:     for all features $f_i \in \tilde{\boldsymbol{x}}^d \setminus \boldsymbol{x}_c^d$ do
 9:         $W \leftarrow W + \tilde{w}_i$
10:     end for
11: end if
12: for all support vectors $\boldsymbol{x}_j \in \mathcal{S} = \bigcup_{f_R \in \boldsymbol{x} \setminus \tilde{\boldsymbol{x}}} h_{\mathcal{S}}(f_R)$ do
13:     $W \leftarrow W + \alpha_j k'_d(\boldsymbol{x}_j, \boldsymbol{x}, \tilde{\boldsymbol{x}})$
14: end for
15: return $W$
```

Fig. 5 Computing weight with fstrie and kernel splitting.

4. Evaluation

We applied our algorithm to SVM and ℓ_1-LLM classifiers and evaluated the resulting classifiers in two NLP tasks: a base phrase chunking task and a dependency parsing task. We compared our SVM classifier with PKI [26] (described in Section 2.2.1) and PKS [14] (described in Section 2.2.3), both of which perform exact weight computation, and PKE [26] (described in Section 2.2.2) for SVMS that approximated the original SVMS by setting the threshold σ for the conjunctive feature weights (referred to as SVM*). Analogously, we compared our LLM classifier with the linear summation described in Eq. (1).

4.1 Task descriptions

4.1.1 Japanese Base Phrase Chunking

A Japanese '*bunsetsu*' base phrase chunker inputs a morphologically analyzed sentence and outputs its correct bunsetsu segmentation; here, a bunsetsu is a grammatical unit in Japanese consisting of one or more content words followed by zero or more function words. A chunker generates a feature vector for each morpheme. The classifier then outputs label y = '+1' (the beginning of the bunsetsu) or '−1' (not the beginning of the bunsetsu).

4.1.2 Japanese Dependency Parsing

A Japanese dependency parser inputs a bunsetsu-segmented sentence and outputs the correct head (bunsetsu) for each bunsetsu. It generates a feature vector for a particular pair of bunsetsus (modifier and modifiee candidates) by exploiting the head-final and projective [35] nature of dependency relations in Japanese. The classifier then outputs label y = '+1' (dependent) or '−1' (independent).

Since our classifier is independent of individual parsing algorithms, we targeted speeding up (a classifier in) the shift-reduce parser proposed by Sassano [42], which has been reported to be the most efficient for this task, with almost state-of-the-art accuracy [18]. This parser decreases the number of classification steps

Table 2 Chunking results for test corpus: SVM classifiers.

model	PKI classify (total) [ms/sent.]	baseline $\boldsymbol{w}$ [MiB]	 classify (total) [ms/sent.]	w/ fstrie$_{\text{S}}$ $\mathcal{W}$ [MiB]	 classify (total) [ms/sent.]	w/ fstrie$_{\text{M}}$ $\mathcal{W}$ [MiB]	 classify (total) [ms/sent.]	w/ fstrie$_{\text{L}}$ $\mathcal{W}$ [MiB]	 classify (total) [ms/sent.]	speed up
SVM_1	1.428 (1.453)	0.2	0.003 (0.027)	0.1	**0.007** (0.031)	4.3	0.007 (0.031)	134.7	0.016 (0.041)	×0.5
SVM_2	1.626 (1.652)	1.8	0.022 (0.047)	0.1	0.013 (0.037)	4.7	**0.012** (0.037)	156.7	0.018 (0.043)	×1.8
SVM_3	2.135 (2.160)	9.0	0.072 (0.098)	0.2	0.033 (0.059)	5.6	0.023 (0.049)	183.7	**0.022** (0.047)	×3.3

by using the fact that a bunsetsu is likely to modify a bunsetsu close to itself.

4.2 Settings

For evaluation, we used the following standard split of Kyoto University Text Corpus (version 4.0) [27], Mainichi news articles in 1995 that have been manually annotated with bunsetsu segmentation and dependency relations [*8]:

Training: Articles of January 1st and 3rd through 11th and editorial articles of January through August (24,283 sentences and 234,685 bunsetsus).

Development: Articles of January 12th and 13th and editorial articles of September (4,833 sentences, 47,571 bunsetsus).

Test: Articles of January 14th through 17th and editorial articles of October through December (9,284 sentences, 89,874 bunsetsus).

The following experiments were performed on a server with an Intel® Xeon™ 3.20-GHz CPU. We used LIBSVM (version 3.1) [*9] and a simple C++ library for maximum entropy classification [*10] to train SVMs and ℓ_1-LLMs, respectively. We used darts-clone (version 0.32f rc2) [*11], a double-array trie [2], [51], as a compact trie implementation. All these libraries and algorithms are implemented in C++. The code for building fstries occupies around 100 lines, while the code for the classifier occupies around 20 lines (except those for kernel expansion).

We should mention that the base phrase chunking is less complex than the dependency parsing, since i) the number of active (conjunctive) features is small and ii) the features originate from local trigrams (as we will later explain in Section 4.3). To keep the following discussion simple, we trained a classifier only with SVMs for base phrase chunking, since we do not need to perform an approximation such as ℓ_1-regularization for LLM or techniques to reduce the size of weight vectors for SVM (described in Sections 2.2.2 and 2.2.3) to reduce the number of active conjunctive features.

We varied SVM soft margin parameter c from 1.0 to 0.0001 and width factor parameter ω [*12], which controls the impact of the prior, from 0.1 to 5.0, and adjusted the values to maximize chunking/dependency accuracy for the development set: for base phrase chunking, we used $(d, c) = (1, 0.5), (2, 0.05), (3, 0.005)$ for SVMs, while for dependency parsing, we used $(d, c) = (1, 0.1)$ $(2, 0.005)$, $(3, 0.0001)$, for SVMs and $(d, \omega) = (1, 1.5), (2, 1.0), (3, 1.0)$ for ℓ_1-LLMS [*13]. For LLM training, we designed explicit conjunctive features for all the d or lower-order feature combinations to make the results comparable with those of SVMs. We hereafter refer to SVM and LLM classifiers trained with d or lower-order conjunctive features as SVM_d and LLM_d, respectively. The superscripts attached to classifiers express their hyper-parameters (e.g., $\text{LLM}_2^{\omega=1.0}$); we omit the hyper-parameters when clear from contexts.

Table 1 Specifications of SVMs for the base phrase chunking task. The accuracies marked with '≫' or '>' were significantly better than the $d = 1$ counterpart ($p < 0.01$ or $0.01 \leq p < 0.05$ by McNemar's test).

model	statistics $\lvert\mathcal{S}\rvert$	 $\lvert\mathcal{F}^d\rvert$	 $\lvert\boldsymbol{x}^d\rvert$	accuracy (%) partial	 complete
SVM_1	13,365	12,261	11.0	99.58	90.80
SVM_2	15,612	215,744	62.3	99.72	$93.90^{\gg}$
SVM_3	20,619	1,704,940	201.7	$99.70^{>}$	$93.55^{\gg}$

4.3 Results for Base Phrase Chunking

We used surface-form, POS, POS-subcategory and inflection form of previous, current, and next words as features to train a classifier. Specifications of SVMs used here are shown in **Table 1**; $|\mathcal{F}^d|$ is the number of active features, while $|\boldsymbol{x}^d|$ is the average number of active features in each classification for the test corpus. Partial accuracy is the ratio of morphemes whose labels are correctly identified by the chunker, while complete accuracy is the exact match accuracy of complete bunsetsu segmentation in a sentence.

We obtained source feature vectors to build fstries for this task by applying the chunkers with the target classifiers to a raw corpus in the target domain, 3,261,638 sentences of 1991–94 Mainichi news articles that were morphologically analyzed by MeCab 0.98 (with a JUMAN dictionary) [*14]. We first built fstrie$_{\text{L}}$ using all the source feature vectors. We then attempted to reduce the number of prefix feature vectors in fstrie$_{\text{L}}$ to $1/2^n$ the size by the algorithm shown in Fig. 3. We refer to fstries built from 1/32 and 1/1024 of the prefix feature vectors in fstrie$_{\text{L}}$ as fstrie$_{\text{M}}$ and fstrie$_{\text{S}}$ in the following experiments.

The performances of parsers having SVM classifiers with and without the fstrie are given in **Table 2**. The column titled $\boldsymbol{w}$ shows the size of weight vectors for SVM classifiers, while the columns $\mathcal{W}$ show the size of fstries$_{\text{S}}$, fstries$_{\text{M}}$, and fstries$_{\text{L}}$, respectively. The 'speed-up' column shows the speed-up factor of the most efficient classifier with an fstrie (bold) versus the baseline classifier without fstries.

[*8] http://nlp.ist.i.kyoto-u.ac.jp/EN/index.php?Kyoto%20University%20 Text%20Corpus

[*9] http://www.csie.ntu.edu.tw/~cjlin/libsvm/

[*10] http://www-tsujii.is.s.u-tokyo.ac.jp/~tsuruoka/maxent/. We used a preliminary version that implements stochastic gradient descent training of the ℓ_1-LLM [48].

[*11] http://code.google.com/p/darts-clone/

[*12] The parameter C of ℓ_1-LLM in Eq. (2) was set to $\omega/|\mathcal{D}|$ (referred to as '*single width*' in Ref. [22]).

[*13] The estimator used to learn ℓ_1-LLMS [48] needed two other hyper-parameters, α and η_0, both of which were tuned to maximize the dependency accuracy for development set: (d, α, η_0) = (1, 0.8, 1.0), (2, 0.8, 0.01), (3, 0.8, 0.0005).

[*14] http:://mecab.sourceforge.net/

Table 3 Feature set used for the dependency parsing task.

Modifier, modifiee bunsetsu	head word (surface-form, POS, POS-subcategory, inflection form), functional word (surface-form, POS, POS-subcategory, inflection form), brackets, quotation marks, punctuation marks, position in sentence (beginning, end)
Between bunsetsus	distance (1, 2–5, 6–), case-particles, brackets, quotation marks, punctuation marks

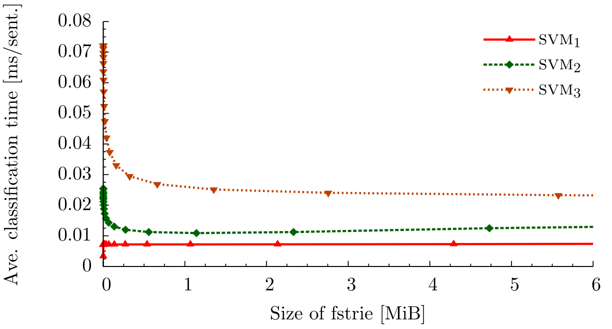

Fig. 6 Average classification time per sentence plotted against size of fstrie in base phrase chunking.

The fstries successfully speeded up the baseline PKE classifiers with conjunctive features ($d \geq 2$). The inefficiency of the classifier ($d = 1$) results from the cost of the additional sort function (line 1 in Fig. 4) and CPU cache failure due to random accesses to the huge fstries. Considering these costs, the performance improvement is almost saturated for fstrie$_{\text{M}}$. The average classification time of our classifiers plotted against fstrie size is shown in **Fig. 6**; the rightmost plots refer to fstrie$_{\text{M}}$ ($n = 5$). With a tiny fstrie ($n = 7$, 1.1 MiB [*15]), the most accurate chunker ($d = 2$) is almost as fast as $d = 1$ counterpart.

4.4 Results for Dependency Parsing

We used the standard feature set tailored for this task [18], [25], [42] (**Table 3**) to train SVM and LLM classifiers. Note that features listed in the 'Between bunsetsus' row represent contexts between the target pair of bunsetsus and appear independently of other features, which will become an obstacle to finding the longest prefix vector. This task is therefore a better measure of our method than the base phrase chunking or simple sequential labeling such as POS tagging or named-entity recognition that uses features originating from a local context.

Specifications of SVMs and LLMs used here are shown in **Table 4** and **Table 5**; SVMs* refer to SVMs that filter out less useful features by setting threshold σ for weights of conjunctive features. r is the common feature threshold in PKS (feature whose frequency is more than $r \cdot |\mathcal{D}|$ was regarded as a common feature), $|\mathcal{F}^d_C|$ is the number of active common features in PKS, while $|\tilde{\boldsymbol{x}}^d|$ is the average number of active common features in PKS in each classification for the test corpus. Partial accuracy is the ratio of dependency relations correctly identified by the parser, while sentence accuracy is the exact match accuracy of complete dependency relations in a sentence.

Table 4 Specifications of SVMs for the dependency parsing task. The accuracies marked with '≫' or '>' were significantly better than the $d = 2$ counterpart ($p < 0.01$ or $0.01 \leq p < 0.05$ by McNemar's test).

model	statistics			accuracy (%)	
	$\lvert S\rvert$	$\lvert\mathcal{F}^d\rvert$ ($\lvert\mathcal{F}^d_C\rvert$)	$\lvert\boldsymbol{x}^d\rvert$ ($\lvert\tilde{\boldsymbol{x}}^d\rvert$)	partial	complete
SVM$_1$	78,327	39,719	27.3	88.27	46.42
SVM$_2$	65,104	1,478,077	380.6	90.76	53.82
SVM$_3$	68,499	26,198,606	3,286.7	90.93	54.44
SVM$_3^{r=0.0001}$	"	(8,175,643)	(3,156.3)	"	"
SVM$_3^{r=0.001}$	"	(1,657,900)	(2,872.4)	"	"
SVM$_3^{r=0.01}$	"	(329,780)	(2,573.4)	"	"
SVM$^{\star\,\sigma=0.001}_3$	"	13,249,989	2,725.9	90.92$^{\gg}$	54.38$^{\gg}$
SVM$^{\star\,\sigma=0.002}_3$	"	2,515,058	2,238.3	90.91$^{\gg}$	54.31$^{>}$
SVM$^{\star\,\sigma=0.003}_3$	"	793,300	1,856.0	90.83	54.21

Table 5 Specifications of LLMs for the dependency parsing task. The accuracies marked with '≫' or '>' were significantly better than the $d = 2$ counterpart ($p < 0.01$ or $0.01 \leq p < 0.05$ by McNemar's test).

model	statistics		accuracy (%)	
	$\lvert\mathcal{F}^d\rvert$	$\lvert\boldsymbol{x}^d\rvert$	partial	complete
ℓ_1-LLM$_1^{\omega=1.5}$	4,874	18.2	88.22	45.86
ℓ_1-LLM$_2^{\omega=1.0}$	171,493	238.0	90.55	53.12
ℓ_1-LLM$_3^{\omega=1.0}$	2,379,326	2,265.0	90.76$^{\gg}$	54.17$^{\gg}$
ℓ_1-LLM$_3^{\omega=2.0}$	703,968	1,774.2	90.75$^{\gg}$	54.15$^{\gg}$
ℓ_1-LLM$_3^{\omega=3.0}$	351,870	1,469.4	90.68$^{>}$	53.78$^{>}$

The accuracy of around 90.9% (SVM$_3$) is close to the performance of state-of-the-art parsers [18], and the model statistics are considered to be complex (or realistic) enough to evaluate our classifier's utility. We could clearly observe that the number of active features $|\boldsymbol{x}^d|$ increased dramatically for this task according to the order d of feature combinations. The density of $|\boldsymbol{x}^d|$ for SVMs was very high (e.g., $|\boldsymbol{x}^3| = 3{,}287.6$, close to the maximum shown in Eq. (9): $(27.3^3 + 5\times 27.3)/6 \simeq 3{,}414$). Comparing specifications of SVMs for the two tasks (Table 1 and Table 4), we can understand why dependency parsing is slower than base phrase chunking.

For $d = 3$ models, we attempted to control the size of the feature space $|\mathcal{F}^d|$ by changing the model's hyper-parameters: threshold σ for the SVM* and width factor ω for the ℓ_1-LLM. Although we successfully reduced the size of the feature space $|\mathcal{F}^d|$, we could not dramatically reduce the average number of active features $|\boldsymbol{x}^d|$ in each classification while keeping the accuracy advantage. This confirms that the solution sparseness does not suffice to obtain an efficient classifier.

We obtained source feature vectors to build fstries by applying parsers with the target classifiers to the data used in Section 4.3 that were bunsetsu-segmented by the chunker with SVM$_2$ shown in Table 1. We used the algorithm in Fig. 3 to build fstrie$_{\text{L}}$, fstrie$_{\text{M}}$ and fstrie$_{\text{S}}$ in the analogous way described in Section 4.3.

Because we exploited the algorithm shown in Fig. 4 to calculate the weights of the prefix feature vectors, it took less than one hour on the 3.20-GHz server to build fstrie$_{\text{L}}$ (and calculate the utility score for all the nodes in it) for the slowest SVM$_3$ from the 40,368,771 source feature vectors (63,365,958 prefix feature vectors) generated by parsing the 3,261,638 sentences.

4.4.1 Results for SVMs

The performances of parsers having SVM classifiers with and

[*15] 1 MiB = 2^{20} bytes = 1,048,576 bytes.

Table 6 Parsing results for test corpus: SVM classifiers.

model	PKI	baseline		w/ fstrie$_S$		w/ fstrie$_M$		w/ fstrie$_L$		speed
	classify (total)	$\boldsymbol{w}$	classify (total)	$\mathcal{W}$	classify (total)	$\mathcal{W}$	classify (total)	$\mathcal{W}$	classify (total)	up
	[ms/sent.]	[MiB]	[ms/sent.]	[MiB]	[ms/sent.]	[MiB]	[ms/sent.]	[MiB]	[ms/sent.]	
SVM$_1$	16.708 (16.758)	0.3	0.004 (0.015)	1.0	**0.006** (0.018)	33.1	0.007 (0.019)	1,074.3	0.017 (0.030)	×0.5
SVM$_2$	12.293 (12.333)	14.9	0.048 (0.061)	1.0	0.023 (0.035)	31.9	**0.023** (0.036)	1,097.3	0.026 (0.039)	×2.1
SVM$_3$	13.271 (13.312)	150.0	0.425 (0.442)	1.0	0.199 (0.214)	32.1	0.127 (0.142)	1,104.4	**0.091** (0.105)	×4.7
SVM$_3^{r=0.0001}$	n/a	59.8	0.401 (0.417)	"	0.177 (0.192)	"	0.108 (0.123)	"	**0.076** (0.090)	×5.3
SVM$_3^{r=0.001}$	n/a	15.0	0.387 (0.404)	"	0.163 (0.178)	"	0.097 (0.112)	"	**0.069** (0.083)	×5.6
SVM$_3^{r=0.01}$	n/a	3.9	0.687 (0.708)	"	0.376 (0.395)	"	0.201 (0.218)	"	**0.125** (0.140)	×5.5

Table 7 Parsing results for test corpus: SVM* and ℓ_1-LLM classifiers.

model	baseline		w/ fstrie$_S$		w/ fstrie$_M$		w/ fstrie$_L$		speed
	$\boldsymbol{w}$	classify (total)	$\mathcal{W}$	classify (total)	$\mathcal{W}$	classify (total)	$\mathcal{W}$	classify (total)	up
	[MiB]	[ms/sent.]	[MiB]	[ms/sent.]	[MiB]	[ms/sent.]	[MiB]	[ms/sent.]	
SVM$^{\star}{}_3^{\sigma=0.001}$	69.7	0.415 (0.431)	1.0	0.179 (0.194)	32.0	0.111 (0.125)	1083.2	**0.078** (0.093)	×5.3
SVM$^{\star}{}_3^{\sigma=0.002}$	17.2	0.386 (0.402)	1.0	0.142 (0.157)	31.7	0.083 (0.097)	1054.7	**0.059** (0.073)	×6.5
SVM$^{\star}{}_3^{\sigma=0.003}$	6.3	0.359 (0.373)	1.0	0.113 (0.126)	31.1	0.063 (0.076)	982.9	**0.046** (0.059)	×7.8
ℓ_1-LLM$_1^{\omega=1.5}$	0.1	0.003 (0.016)	0.8	**0.005** (0.018)	24.4	0.005 (0.018)	629.2	0.012 (0.026)	×0.5
ℓ_1-LLM$_2^{\omega=1.0}$	2.1	0.051 (0.065)	1.1	0.020 (0.033)	37.3	**0.018** (0.032)	1,130.9	0.021 (0.036)	×2.8
ℓ_1-LLM$_3^{\omega=1.0}$	12.6	0.432 (0.448)	1.2	0.165 (0.180)	37.9	0.091 (0.107)	1,233.7	**0.061** (0.076)	×7.1
ℓ_1-LLM$_3^{\omega=2.0}$	4.0	0.387 (0.402)	1.0	0.130 (0.144)	30.9	0.066 (0.080)	987.7	**0.043** (0.057)	×9.1
ℓ_1-LLM$_3^{\omega=3.0}$	2.2	0.358 (0.372)	0.8	0.114 (0.128)	26.8	0.054 (0.068)	848.7	**0.034** (0.048)	×10.6

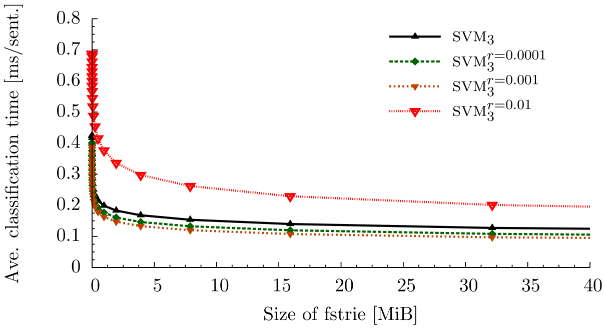

Fig. 7 Average classification time per sentence plotted against size of fstrie in dependency parsing: SVM$_3$.

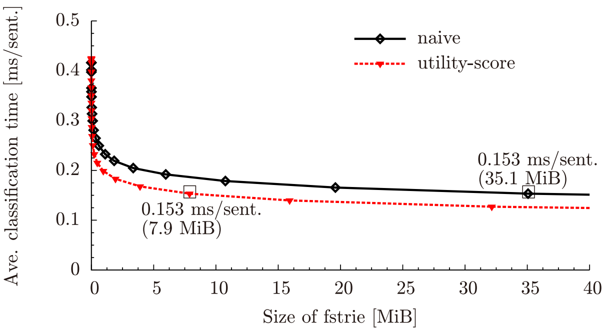

Fig. 8 Fstrie reduction: utility score vs. processed sentence reduction for SVM$_3$.

without the fstrie are given in **Table 6**. Since each classifier with a different order of conjunctive features solved a slightly different number of classification steps, we show the (average) cumulative classification time for a sentence.

The fstries successfully speeded up SVM classifiers with conjunctive features ($d \geq 2$). Although the baseline PKE/PKS classifiers without fstries were still faster than PKI, as expected from a large $|\boldsymbol{x}^d|$ value, the classifiers with higher conjunctive features were much slower than the classifier with only primitive features ($d = 1$) by factors of 14 ($d = 2$), and 121 ($d = 3$) and the classification time accounted for most of the parsing time.

We should note that the slowest PKS classifier ($d = 3$, $r = 0.01$) can be faster than the PKE classifier when we combine the classifier with the fstrie$_M$. The combination of PKS with fstrie (the algorithm shown in Fig. 5) yielded a space-efficient SVM classifier while keeping the accuracy and the classification speed.

The average classification time of our classifiers plotted against fstrie size is shown in **Fig. 7**. Surprisingly, we obtained a significant speed-up even with tiny fstrie sizes of < 1 MiB. Furthermore, we naively controlled the fstrie size by simply reducing the number of sentences processed to $1/2^n$. The impact on the speed-up of the resulting fstries (*naive*) and the fstries constructed by our utility score (*utility-score*) on SVM$_3$ is shown in **Fig. 8**. The Zipfian nature of language data let us obtain a substantial speed-up even when we naively reduced the fstrie size, and the utility score further decreased the fstrie size required to obtain the same speed-up. We needed less than 1/4 size fstries to achieve the same speed-up: 0.425 → 0.153 ms/sent. with 7.9 MiB (*utility-score*) vs. 35.1 MiB (*naive*).

4.4.2 Results for SVMs* and ℓ_1-LLMs

The performances of parsers having SVM* and ℓ_1-LLM classifiers with and without the fstrie are given in **Table 7**. The fstries successfully speeded up the SVM* and ℓ_1-LLM classifiers by factors of 7.8 ($d = 3$, $\sigma = 0.003$) and 10.6 ($d = 3$, $\omega = 3.0$), respectively. We obtained more speed-up when we used fstries for classifiers with more sparse feature space $\mathcal{F}^d$ (**Fig. 9** and **Fig. 10**). The parsing speeds with $d = 3$ models are now comparable to those with $d = 2$ models.

Without fstries, little speed-up of SVM classifiers versus the SVM* classifiers (in Table 6) was obtained owing to the mild reduction in the average number of active features $|\boldsymbol{x}^d|$ in the classification. This result agrees with the results reported in Ref. [26].

The parsing speed reached 13,653 sentences per second with accuracy of 90.91% (SVM$^{\star}{}_3^{\sigma=0.002}$). We used this parser to process 22,540,994 sentences (140,633,895 bunsetsus) from Japanese weblog feeds updated in December 2010, to see how much the impact of fstries lessens when the test data and the data processed to build fstries mismatch. The parsing time was 4,472.1 seconds

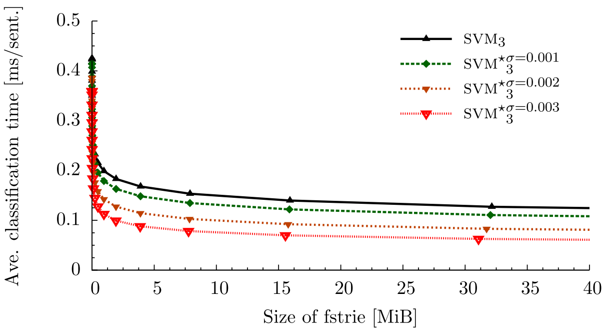

Fig. 9 Average classification time per sentence plotted against size of fstrie: $\textsc{svm}^{\star}{}_3$.

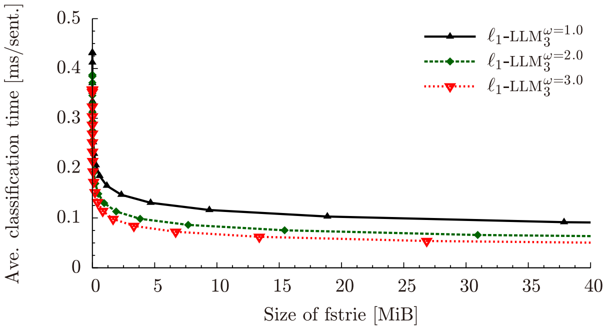

Fig. 10 Average classification time per sentence plotted against size of fstrie: ℓ_1-$\textsc{llm}_3$.

without fstrie$_{\text{L}}$, while it was only 1,032.4 seconds with fstrie$_{\text{L}}$. The speed-up factor of 4.3 for weblog feeds was slightly worse than that for news articles (0.402/0.073 $\simeq$ 5.5) but still evident. This implies that sorting features in building fstries yielded prefix features vectors that commonly appear in a Japanese dependency parsing task by excluding domain-specific features such as lexical features.

In summary, our algorithm successfully minimized the efficiency gap among classifiers with different degrees of feature combinations and made accurate classifiers trained with higher-order feature combinations practical.

5. Conclusions

Our simple method speeds up a classifier trained with many conjunctive features by using precalculated weights of (partial) feature vectors stored in a feature sequence trie (fstrie). We experimentally demonstrated that it speeded up SVM and LLM classifiers for a base phrase chunking task and a dependency parsing task by a factor of 1.8 to 10.6. We also confirmed that the sparse feature space provided by ℓ_1-LLMS and SVMS contributed much to size reduction of the fstrie required to achieve the same speed-up. The implementations of the proposed algorithm for LLMS and SVMS (with a polynomial kernel) and the bunsetsu chunker and dependency parser for Japanese are publicly available at http://www.tkl.iis.u-tokyo.ac.jp/˜ynaga/pecco/ and http://www.tkl.iis.u-tokyo.ac.jp/˜ynaga/jdepp/, respectively.

We plan to apply our method to wider range of classifiers used in various NLP tasks. To speed up classifiers used in a real-time application, we can build fstries incrementally by using feature vectors generated from user inputs. When we run our classifiers on resource-tight environments such as cell-phones, we can use a random feature mixing technique [13] or a memory-efficient trie implementation based on a succinct data structure [11], [19] to reduce required memory usage.

We will combine our method with other techniques that provide sparse solutions, for example, kernel methods on a budget [9], [10], [37] or kernel approximation (surveyed in Ref. [21]). And in the future, we will consider the issue of speeding up decoding with structured models [28], [32], [45].

Acknowledgments This work was supported by the Multimedia Web Analysis Framework towards Development of Social Analysis Software program of the Ministry of Education, Culture, Sports, Science and Technology, Japan. The authors wish to thank Susumu Yata and Yoshimasa Tsuruoka for letting them use their libraries. They also thank Nobuhiro Kaji and the anonymous reviewers for their valuable comments.

Reference

[1] Ando, R.K. and Zhang, T.: A Framework for Learning Predictive Structures from Multiple Tasks and Unlabeled Data, *Journal of Machine Learning Research*, Vol.6, pp.1817–1853 (2005).

[2] Aoe, J.: An Efficient Digital Search Algorithm by Using a Double-Array Structure, *IEEE Trans. Softw. Eng.*, Vol.15, No.9, pp.1066–1077 (1989).

[3] Bellare, K., Talukdar, P.P., Kumaran, G., Pereira, F., Liberman, M., McCallum, A. and Dredze, M.: Lightly-Supervised Attribute Extraction, *Proc. NIPS 2007 Workshop on Machine Learning for Web Search* (2007).

[4] Berger, A., Pietra, S.D. and Pietra, V.D.: A Maximum Entropy Approach to Natural Language Processing, *Computational Linguistics*, Vol.22, No.1, pp.39–71 (1996).

[5] Brants, T., Popat, A.C., Xu, P., Och, F.J. and Dean, J.: Large Language Models in Machine Translation, *Proc. EMNLP-CoNLL 2007*, pp.858–867 (2007).

[6] Chuang, K.-T., Huang, J.-L. and Chen, M.-S.: Power-law Relationship and Self-similarity in the Itemset Support Distribution: Analysis and Applications, *The VLDB Journal*, Vol.17, No.5, pp.1121–1141 (2008).

[7] Cortes, C. and Vapnik, V.: Support-Vector Networks, *Machine Learning*, Vol.20, No.3, pp.273–297 (1995).

[8] Daumé III, H.: Cross-task Knowledge-constrained Self Training, *Proc. EMNLP 2008*, pp.680–688 (2008).

[9] Dekel, O., Shalev-Shwartz, S. and Singer, Y.: The Forgetron: A Kernel-Based Perceptron on a Budget, *SIAM J. Comput.*, Vol.37, No.5, pp.1342–1372 (2008).

[10] Dekel, O. and Singer, Y.: Support Vector Machines on a Budget, *Advances in Neural Information Processing Systems 19*, Schölkopf, B., Platt, J. and Hofmann, T. (Eds.), The MIT Press, pp.345–352 (2007).

[11] Delpratt, O., Rahman, N. and Raman, R.: Engineering the LOUDS Succinct Tree Representation, *Proc. WEA 2006*, pp.134–145 (2006).

[12] Egghe, L.: The distribution of N-grams, *Scientometrics*, Vol.47, No.2, pp.237–252 (2000).

[13] Ganchev, K. and Dredze, M.: Small Statistical Models by Random Feature Mixing, *Proc. ACL 2008 Workshop on Mobile Language Processing*, pp.19–20 (2008).

[14] Goldberg, Y. and Elhadad, M.: splitSVM: Fast, Space-Efficient, non-Heuristic, Polynomial Kernel Computation for NLP Applications, *Proc. ACL 2008, Short Papers*, pp.237–240 (2008).

[15] Huang, L. and Sagae, K.: Dynamic Programming for Linear-Time Incremental Parsing, *Proc. ACL 2010*, pp.1077–1086 (2010).

[16] Inui, K., Abe, S., Hara, K., Morita, H., Sao, C., Eguchi, M., Sumida, A., Murakami, K. and Matsuyoshi, S.: Experience Mining: Building a Large-Scale Database of Personal Experiences and Opinions from Web Documents, *Proc. WI-IAT 2008*, pp.313–321 (2008).

[17] Isozaki, H. and Kazawa, H.: Efficient Support Vector Classifiers for Named Entity Recognition, *Proc. COLING 2002*, pp.1–7 (2002).

[18] Iwatate, M., Asahara, M. and Matsumoto, Y.: Japanese Dependency Parsing Using a Tournament Model, *Proc. COLING 2008*, pp.361–368 (2008).

[19] Jacobson, G.: Space-efficient Static Trees and Graphs, *Proc. FOCS 1989*, pp.549–554 (1989).

[20] Kaji, N., Fujiwara, Y., Yoshinaga, N. and Kitsuregawa, M.: Efficient

Staggered Decoding for Sequence Labeling, *Proc. ACL 2010*, pp.485–494 (2010).

[21] Kashima, H., Idé, T., Kato, T. and Sugiyama, M.: Recent Advances and Trends in Large-scale Kernel Methods, *IEICE Trans. Inf. Syst.*, Vol.E92-D (2009). to appear.

[22] Kazama, J. and Tsujii, J.: Maximum Entropy Models with Inequality Constraints: A Case Study on Text Categorization, *Machine Learning*, Vol.60, No.1-3, pp.159–194 (2005).

[23] Kitsuregawa, M., Tamura, T., Toyoda, M. and Kaji, N.: Socio-Sense: A System for Analysing the Societal Behavior from Long Term Web Archive, *Proc. APWeb 2008*, pp.1–8 (2008).

[24] Koo, T., Carreras, X. and Collins, M.: Simple Semi-supervised Dependency Parsing, *Proc. ACL 2008*, pp.595–603 (2008).

[25] Kudo, T. and Matsumoto, Y.: Japanese Dependency Analysis Using Cascaded Chunking, *Proc. CoNLL 2002*, pp.1–7 (2002).

[26] Kudo, T. and Matsumoto, Y.: Fast Methods for Kernel-based Text Analysis, *Proc. ACL 2003*, pp.24–31 (2003).

[27] Kurohashi, S. and Nagao, M.: Building a Japanese Parsed Corpus, *Treebank: Building and Using Parsed Corpora*, Abeillé, A. (Ed.), Kluwer Academic Publishers, pp.249–260 (2003).

[28] Lafferty, J.D., McCallum, A. and Pereira, F.C.N.: Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data, *Proc. ICML 2001*, pp.282–289 (2001).

[29] Liang, P., Daumé III, H. and Klein, D.: Structure Compilation: Trading Structure for Features, *Proc. ICML 2008*, pp.592–599 (2008).

[30] Liu, Y. and Sarkar, A.: Experimental Evaluation of LTAG-based Features for Semantic Role Labeling, *Proc. EMNLP-CoNLL 2007*, pp.590–599 (2007).

[31] McClosky, D., Charniak, E. and Johnson, M.: Effective Self-training for Parsing, *Proc. HLT-NAACL 2006*, pp.152–159 (2006).

[32] Miyao, Y. and Tsujii, J.: Maximum Entropy Estimation for Feature Forests, *Proc. HLT 2002*, pp.292–297 (2002).

[33] Neubig, G., Nakata, Y. and Mori, S.: Pointwise Prediction for Robust, Adaptable Japanese Morphological Analysis, *Proc. ACL 2011, Short Papers*, pp.529–533 (2011).

[34] Nguyen, N.L. and Kim, J.-D.: Exploring Domain Differences for the Design of a Pronoun Resolution System for Biomedical Texts, *Proc. COLING 2008*, pp.625–632 (2008).

[35] Nivre, J.: An Efficient Algorithm for Projective Dependency Parsing, *Proc. IWPT 2003*, pp.149–160 (2003).

[36] Okanohara, D. and Tsujii, J.: Learning Combination Features with L_1 Regularization, *Proc. HLT-NAACL 2009, Short Papers*, pp.97–100 (2009).

[37] Orabona, F., Keshet, J. and Caputo, B.: Bounded Kernel-Based Online Learning, *Journal of Machine Learning Research*, Vol.10, pp.2643–2666 (2009).

[38] Pantel, P.: Data Catalysis: Facilitating Large-Scale Natural Language Data Processing, *Proc. ISUC 2007*, pp.201–204 (2007).

[39] Pantel, P., Ravichandran, D. and Hovy, E.: Towards Terascale Knowledge Acquisition, *Proc. COLING 2004*, pp.771–777 (2004).

[40] Saeger, S.D., Torisawa, K. and Kazama, J.: Mining Web-scale Treebanks, *Proc. NLP 2009*, pp.837–840 (2009).

[41] Saito, H., Kamoshida, Y., Sawai, S., Hironaka, K., Takahashi, K., Sekiya, T., Dun, N., Shibata, T., Yokoyama, D. and Taura, K.: InTrigger: A Multi-Site Distributed Computing Environment Supporting Flexible Configuration Changes, *Proc. SWOPP 2007*, pp.237–242 (2007).

[42] Sassano, M.: Linear-Time Dependency Analysis for Japanese, *Proc. COLING 2004*, pp.8–14 (2004).

[43] Shinzato, K., Shibata, T., Kawahara, D., Hashimoto, C. and Kurohashi, S.: TSUBAKI: An Open Search Engine Infrastructure for Developing New Information Access Methodology, *Proc. IJCNLP 2008*, pp.189–196 (2008).

[44] Spoustová, D.J., Hajič, J., Raab, J. and Spousta, M.: Semi-supervised Training for the Averaged Perceptron POS Tagger, *Proc. EACL 2009*, pp.763–771 (2009).

[45] Sutton, C., Rohanimanesh, K. and McCallum, A.: Dynamic Conditional Random Fields: Factorized Probabilistic Models for Labeling and Segmenting Sequence Data, *Proc. ICML 2004*, pp.783–790 (2004).

[46] Tibshirani, R.: Regression Shrinkage and Selection via the Lasso, Technical report, Department of Statistics, University of Toronto (1994).

[47] Torisawa, K., Saeger, S.D., Kakizawa, Y., Kazama, J., Murata, M., Noguchi, D. and Sumida, A.: TORISHIKI-KAI, An Autogenerated Web Search Directory, *Proc. ISUC 2008*, pp.179–186 (2008).

[48] Tsuruoka, Y., Tsujii, J. and Ananiadou, S.: Stochastic Gradient Descent Training for L1-regularized Log-linear Models with Cumulative Penalty, *Proc. ACL-IJCNLP 2009*, pp.477–485 (2009).

[49] Williams, P.M.: Bayesian Regularization and Pruning using a Laplace Prior, *Neural Computation*, Vol.7, No.1, pp.117–143 (1995).

[50] Wu, Y.-C., Yang, J.-C. and Lee, Y.-S.: An Approximate Approach for Training Polynomial Kernel SVMs in Linear Time, *Proc. ACL 2007 Poster and Demo Sessions*, pp.65–68 (2007).

[51] Yata, S., Morita, K., Fuketa, M. and Aoe, J.: Fast String Matching with Space-efficient Word Graphs, *Proc. Innovations in Information Technology 2008*, pp.79–83 (2008).

[52] Zipf, G.K.: *Human Behavior and the Principle of Least-Effort*, Addison-Wesley (1949).

Naoki Yoshinaga received his B.Sc. and M.Sc. from the University of Tokyo in 2000 and in 2002, and Ph.D. from the University of Tokyo in 2005. He had been a JSPS research fellow (DC1, PD) from 2002 to 2008. Since 2008, he has been a research associate at the Institute of Industrial Science at the University of Tokyo in Japan. His current research interests include computational linguistics and machine learning.

Masaru Kitsuregawa is a professor, and the director of the Center for Information Fusion at the Institute of Industrial Science, and executive director for Earth Observation Data Integration and Fusion Research Initiative (EDITORIA), at the University of Tokyo in Japan. He received his Ph.D. degree in information engineering in 1983 from the University of Tokyo. His research interests include database engineering and advanced storage system. He serves as the chair of the steering committee of IEEE ICDE (Int. Conf. on Data Engineering) and has been a trustee of the VLDB Endowment. He was the first recipient in Asia of the ACM SIGMOD E. F. Codd Innovation Award. He is serving as a science advisor to the Ministry of Education, Culture, Sports, Science and Technology.