Polynomial to Linear: Efficient Classification with Conjunctive Features

Naoki Yoshinaga
Institute of Industrial Science
University of Tokyo
4-6-1 Komaba, Meguro-ku, Tokyo
ynaga@tkl.iis.u-tokyo.ac.jp

Masaru Kitsuregawa
Institute of Industrial Science
University of Tokyo
4-6-1 Komaba, Meguro-ku, Tokyo
kitsure@tkl.iis.u-tokyo.ac.jp

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 appear frequently in the target NLP task. A 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 for a Japanese dependency parsing task show that our method speeded up the SVM and LLM classifiers of the parsers, which achieved accuracy of 90.84/90.71%, by a factor of 10.7/11.6.

1 Introduction

Deep and accurate text analysis based on discriminative models is not yet efficient enough as a component of real-time applications, and it is inadequate to process Web-scale corpora for knowledge acquisition (Pantel, 2007; Saeger et al., 2009) or semi-supervised learning (McClosky et al., 2006; Spoustová et al., 2009). 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 dependency parsing (Koo et al., 2008), parse re-ranking (McClosky et al., 2006), pronoun resolution (Nguyen and Kim, 2008), and semantic role labeling (Liu and Sarkar, 2007). However, 'explicit' feature combinations significantly increase the feature space, which slows down not only training but also testing of the classifier.

Kernel-based methods such as support vector machines (SVMs) consider feature combinations space-efficiently by using a polynomial kernel function (Cortes and Vapnik, 1995). 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 (Isozaki and Kazawa, 2002; Kudo and Matsumoto, 2003; Goldberg and Elhadad, 2008).

ℓ_1-regularized log-linear models (ℓ_1-LLMs), on the other hand, provide sparse solutions, in which weights of irrelevant features are exactly zero, by assuming a Laplacian prior on the weights (Tibshirani, 1996; Kazama and Tsujii, 2003; Goodman, 2004; Gao et al., 2007). However, as Kazama and Tsujii (2005) have reported in a text categorization task and we later confirm in a dependency parsing task, when most features regarded as irrelevant during training ℓ_1-LLMs appear rarely in the task, we cannot greatly reduce the number of active features in each classification. In the end, when efficiency is a major concern, we must use exhaustive feature selection (Wu et al., 2007; Okanohara and Tsujii, 2009) or even restrict the order of conjunctive features at the expense of accuracy.

In this study, we provide a simple, but 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 and use them as partial results to compute the weight of a given feature vector. We use a trie called the *feature sequence trie* to efficiently 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 classification task. The time complexity of the classifier approaches time that

is linear with respect to the number of 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 Japanese dependency parsing task. Experimental results show that it successfully speeded up classifiers trained with higher-order conjunctive features by a factor of 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 concludes with a summary and addresses future directions.

2 Preliminaries

In this paper, we focus on linear classifiers that calculate the probability (or score) by summing up weights of individual features. Examples include not only log-linear models but also support vector machines with kernel expansion (Isozaki and Kazawa, 2002; Kudo and Matsumoto, 2003). Below, we introduce these two classifiers and their ways to consider feature combinations.

In classification-based NLP, the target task is modeled as one or more classification steps. For example in part-of-speech (POS) tagging, each classification decides whether to assign a particular *label* (POS tag) to a given *sample* (each word in a given sentence). Each sample 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 sample. We say that a feature f_i is *active* in sample $\boldsymbol{x}$ when $x_i = f_i(\boldsymbol{x}) = 1$ and $|\boldsymbol{x}|$ represents the number of active features in $\boldsymbol{x}$ ($|\boldsymbol{x}| = |\{f_i | f_i(\boldsymbol{x}) = 1\}|$).

2.1 Log-Linear Models

The log-linear model (LLM), or also known as maximum-entropy model (Berger et al., 1996), is a linear classifier widely used in the NLP literature. Let the training data of LLMs be $\{\langle \boldsymbol{x}_i, y_i \rangle\}_{i=1}^{L}$, where $\boldsymbol{x}_i \in \{0, 1\}^n$ is a feature vector and y_i is a class label associated with $\boldsymbol{x}_i$. We assume a binary label $y_i \in \{\pm 1\}$ here to simplify the argument.

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

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

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

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:

$$\mathcal{L}(\boldsymbol{w}) = \sum_{i=1}^{L} \log p(y_i|\boldsymbol{x}_i) - C\|\boldsymbol{w}\|_1. \quad (2)$$

Hyper-parameter C thereby controls the degree of over-fitting (solution sparseness). Interested readers may refer to the cited literature (Andrew and Gao, 2007) for the optimization procedures.

2.2 Support Vector Machines

A support vector machine (SVM) is a binary classifier (Cortes and Vapnik, 1995). Training with samples $\{\langle \boldsymbol{x}_i, y_i \rangle\}_{i=1}^{L}$ where $\boldsymbol{x}_i \in \{0, 1\}^n$ and $y_i \in \{\pm 1\}$ yields the following decision function:

$$y(\boldsymbol{x}) = \mathrm{sgn}(g(\boldsymbol{x}) + b)$$
$$g(\boldsymbol{x}) = \sum_{\boldsymbol{x}_j \in \mathcal{SV}} y_j \alpha_j \phi(\boldsymbol{x}_j)^{\mathrm{T}} \phi(\boldsymbol{x}), \quad (3)$$

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

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:

$$g(\boldsymbol{x}) = \sum_{\boldsymbol{x}_j \in \mathcal{SV}} y_j \alpha_j (\boldsymbol{x}_j^{\mathrm{T}} \boldsymbol{x} + 1)^d. \quad (4)$$

Since we assumed that x_i is a binary value representing whether a (primitive) feature f_i is active in the sample, the polynomial kernel of degree d implies a mapping ϕ_d from $\boldsymbol{x}$ to $\phi_d(\boldsymbol{x})$ that has

$H = \sum_{k=0}^{d} \binom{n}{k}$ dimensions. Each dimension represents a (weighted) conjunction of d features in the original sample $\boldsymbol{x}$.[1]

Kernel Expansion (SVM-KE) The time complexity of Eq. 4 is $O(|\boldsymbol{x}| \cdot |\mathcal{SV}|)$. This cost is usually high for classifiers used in NLP tasks because they often have many support vectors ($|\mathcal{SV}| > 10,000$). *Kernel expansion* (KE) was proposed by Isozaki and Kazawa (2002) to convert Eq. 4 into the linear sum of the weights in the mapped feature space as in LLM ($p(y|\boldsymbol{x})$ in Eq. 1):

$$g(\boldsymbol{x}) = \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}^d = \sum_i w_i x_i^d, \tag{5}$$

where $\boldsymbol{x}^d$ is a binary feature vector whose element x_i^d has a non-zero value when $(\phi_d(\boldsymbol{x}))_i > 0$, $\boldsymbol{w}$ 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. Interested readers may refer to Kudo and Matsumoto (2003) for the detailed computation for obtaining $\boldsymbol{w}$.

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 $\mathcal{F}^d$.

Heuristic Kernel Expansion (SVM-HKE) To make the weight vector sparse, Kudo and Matsumoto (2003) proposed a heuristic method that filters out less useful features whose absolute weight values are less than a pre-defined threshold σ.[2] 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.

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,

[1]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 $\phi_2(\boldsymbol{x}) = (1, \sqrt{3}x_1, \sqrt{3}x_2, \sqrt{2}x_1x_2)^{\mathrm{T}}$. In the following argument, we ignore the dimension of the constant in the mapped space and assume constant b is set to include it.

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

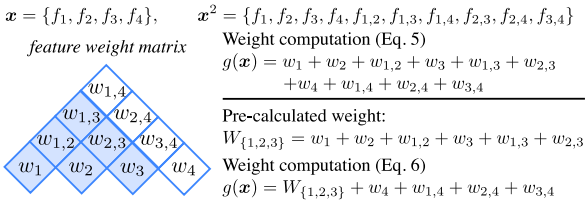

Figure 1: Efficient computation of $g(\boldsymbol{x})$.

SVMs), but an analogous argument is possible for linear classifiers (*e.g.*, LLMs).[3]

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 $g(\boldsymbol{x})$ in Eq. 5 maps $\boldsymbol{x} \in 2^{\mathcal{F}}$ to $W \in \mathbb{R}$. If we could calculate $W_{\boldsymbol{x}} = g(\boldsymbol{x})$ for all possible $\boldsymbol{x}$ in advance, we could obtain $g(\boldsymbol{x})$ 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 $W_{\boldsymbol{x}'} = g(\boldsymbol{x}')$ for $\boldsymbol{x}' \in \mathcal{V}_c (\subset 2^{\mathcal{F}})$, a certain subset of the possible value space, and compute $g(\boldsymbol{x})$ for $\boldsymbol{x} \notin \mathcal{V}_c$ by using precalculated weight $W_{\boldsymbol{x}_c}$ for $\boldsymbol{x}_c \subseteq$[4] $\boldsymbol{x}$ in the following way:

$$g(\boldsymbol{x}) = W_{\boldsymbol{x}_c} + \sum_{f_i \in \boldsymbol{x}^d - \boldsymbol{x}_c^d} w_i. \tag{6}$$

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 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 Figure 1. We can efficiently compute $g(\boldsymbol{x})$ 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 $f_{3,4}$, in order to add up the remaining

[3]When a feature vector $\boldsymbol{x}$ includes (explicit) conjunctive features $f \in \mathcal{F}^d$, we assume weight function $g'(y|\boldsymbol{x}') = g(y|\boldsymbol{x})$, where $\boldsymbol{x}'$ is a projection of $\boldsymbol{x}$ (by $\phi_d^{-1} : \mathcal{F}^d \to \mathcal{F}$).

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

weights, while the normal computation in Eq. 5 should check the four primitive and six conjunctive features to get the individual weights.

Expected time complexity Counting the number of features to be checked in the computation, we obtain the time complexity $f(\boldsymbol{x}, d)$ of Eq. 6 as:

$$f(\boldsymbol{x}, d) = O(|\boldsymbol{x}_c| + |\boldsymbol{x}^d| - |\boldsymbol{x}_c^d|), \quad (7)$$

$$\textit{where} \quad |\boldsymbol{x}^d| = \sum_{k=1}^{d} \binom{|\boldsymbol{x}|}{k} \quad (8)$$

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

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

$$\boldsymbol{x}_c = \underset{\boldsymbol{x}' \in \mathcal{V}_c, \boldsymbol{x}' \subseteq \boldsymbol{x}}{\operatorname{argmin}} (|\boldsymbol{x}'| + |\boldsymbol{x}^d| - |\boldsymbol{x}'^d|). \quad (9)$$

3.2 Construction of Feature Sequence Trie

There are two issues with speeding up the classifier by the computation shown in Eq. 6. First, since we can store weights for only a small fraction of possible feature vectors (namely, $|\mathcal{V}_c| \ll 2^{|\mathcal{F}|}$), we should choose $\mathcal{V}_c$ so as to maximize its impact on the speed-up. Second, we should quickly find an optimal $\boldsymbol{x}_c$ from $\mathcal{V}_c$ 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 each set is likely to follow Zipf's law (Zipf, 1949) 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 (Egghe, 2000) and itemset support distribution (Chuang et al., 2008). We can thereby expect that a small set of partial feature vectors 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. 9) 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*).

[5] This is the maximum number of conjunctive features.

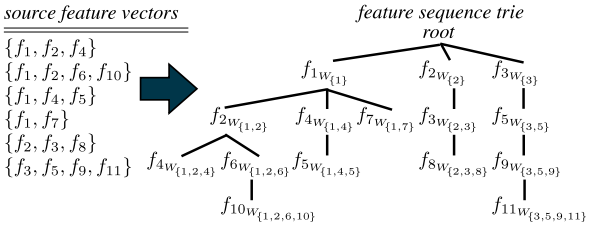

Figure 2: Feature sequence trie and completion of prefix feature vector weights.

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 six source feature vectors is shown in Figure 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.*, $\{f_1, f_2\}$ in Figure 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.

Algorithm 1 PRUNE NODES FROM FSTRIE

Input: fstrie T, node_limit $N \in \mathbb{N}$
Output: fstrie T

1: **while** # of nodes in $T > N$ **do**
2: $\quad \boldsymbol{x}_c \leftarrow \underset{\boldsymbol{x}' \in \mathit{leaf}(T)}{\operatorname{argmin}} u(\boldsymbol{x}')$
3: $\quad$ remove $\boldsymbol{x}_c$, T
4: **end while**
5: **return** T

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:

$$p(\boldsymbol{x}_c) = \frac{\sum_{\boldsymbol{x}' \supseteq \boldsymbol{x}_c} n_{\boldsymbol{x}'}}{\sum_{\boldsymbol{x}} n_{\boldsymbol{x}}}, \qquad (10)$$

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 remove $\boldsymbol{x}_c$. Since the fstrie stores the weight of a prefix feature vector $\boldsymbol{x}_{c\text{-}} \subset \boldsymbol{x}_c$ such that $|\boldsymbol{x}_{c\text{-}}| = |\boldsymbol{x}_c| - 1$ (*e.g.*, in Figure 2, $\boldsymbol{x}_{c\text{-}} = \{f_1, f_2\}$ for $\boldsymbol{x}_c = \{f_1, f_2, f_4\}$), we can define the computation reduction as:

$$\begin{aligned}\Delta_d(\boldsymbol{x}_c) &= (|\boldsymbol{x}_c^d| - |\boldsymbol{x}_{c\text{-}}^d|) - (|\boldsymbol{x}_c| - |\boldsymbol{x}_{c\text{-}}|) \\ &= \sum_{k=2}^{d} \binom{|\boldsymbol{x}_c|}{k} - \sum_{k=2}^{d} \binom{|\boldsymbol{x}_c| - 1}{k} \\ &\quad (\because \text{Eq. 8}).\end{aligned}$$

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

We calculate 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 target task, and prune the lowest-utility-score leaf nodes from the fstrie one by one (Algorithm 1). If several prefix vectors have the same utility score, we eliminate them in numerical descending order.

Algorithm 2 COMPUTE WEIGHT WITH FSTRIE

Input: fstrie T, weight vector $\boldsymbol{w} \in \mathbb{R}^{|\mathcal{F}^d|}$
feature vector $\boldsymbol{x} \in 2^{\mathcal{F}}$
Output: weight $W = g(\boldsymbol{x}) \in \mathbb{R}$

1: $\boldsymbol{x} \leftarrow \text{sort}(\boldsymbol{x})$
2: $\langle \boldsymbol{x}_c, W_{\boldsymbol{x}_c} \rangle \leftarrow$ prefix_search(T, $\boldsymbol{x}$)
3: $W \leftarrow W_{\boldsymbol{x}_c}$
4: **for all** feature $f_j \in \boldsymbol{x}^d - \boldsymbol{x}_c^d$ **do**
5: $\quad W \leftarrow W + w_j$
6: **end for**
7: **return** W

3.3 Classification Algorithm

Our classification algorithm is shown in detail in Algorithm 2. 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 Algorithm 2). It then adds the weights of the remaining features to partial weight $W_{\boldsymbol{x}_c}$ (line 5 in Algorithm 2).

Note that the remaining features whose weights we sum up (line 4 in Algorithm 2) are primitive and conjunctive features that relate to $f \in \boldsymbol{x} - \boldsymbol{x}_c$, 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.*, SVM-HKEs or ℓ_1-LLMs), $|\boldsymbol{x}^d| - |\boldsymbol{x}_c^d|$ can be much smaller than the theoretical expectation (Eq. 8). We confirmed this in the following experiments.

4 Evaluation

We applied our algorithm to SVM-KE, SVM-HKE, and ℓ_1-LLM classifiers and evaluated the resulting classifiers in a Japanese dependency parsing task. To the best of our knowledge, there are no previous reports of an *exact* weight calculation faster than linear summation (Eqs. 1 and 5). We also compared our SVM classifier with a classifier called polynomial kernel inverted (PKI: Kudo and Matsumoto (2003)), which uses the polynomial kernel (Eq. 4) and inverted indexing to support vectors.

4.1 Experimental Settings

A Japanese dependency parser inputs *bunsetsu*-segmented sentences and outputs the correct head (bunsetsu) for each bunsetsu; here, a bunsetsu is a grammatical unit in Japanese consisting of one or more content words followed by zero or more function words. A parser generates a feature vec-

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

Table 1: Feature set used for experiments.

tor for a particular pair of bunsetsus (modifier and modifiee candidates) by exploiting the head-final and projective (Nivre, 2003) 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 (2004), which has been reported to be the most efficient for this task, with almost state-of-the-art accuracy (Iwatate et al., 2008). This parser decreases the number of classification steps by using the fact that a bunsetsu is likely to modify a bunsetsu close to itself. Due to space limitations, we omit the details of the parsing algorithm.

We used the standard feature set tailored for this task (Kudo and Matsumoto, 2002; Sassano, 2004; Iwatate et al., 2008) (Table 1). Note that features listed in the 'Between bunsetsus' row represent contexts between the target pair of bunsetsus and appear independently from other features, which will become an obstacle to finding the longest prefix vector. This task is therefore a better measure of our method than simple sequential labeling such as POS tagging or named-entity recognition.

For evaluation, we used Kyoto Text Corpus Version 4.0 (Kurohashi and Nagao, 2003), Mainichi news articles in 1995 that have been manually annotated with dependency relations.[6] The training, development, and test sets included 24,283, 4833, and 9284 sentences, and 234,685, 47,571, and 89,874 bunsetsus, respectively. The training samples generated from the training set included 150,064 positive and 146,712 negative samples.

The following experiments were performed on a server with an Intel® Xeon™ 3.20-GHz CPU. We used TinySVM[7] and a simple C++ library for maximum entropy classification[8] to train SVMs and ℓ_1-LLMs, respectively. We used Darts-Clone,[9] a double-array trie (Aoe, 1989; Yata et al., 2008), as a compact trie implementation. All these libraries and algorithms are implemented in C++. The code for building fstries occupies 100 lines, while the code for the classifier occupies 20 lines (except those for kernel expansion).

[6] http://nlp.kuee.kyoto-u.ac.jp/nl-resource/corpus-e.html
[7] http://chasen.org/~taku/software/TinySVM/
[8] http://www-tsujii.is.s.u-tokyo.ac.jp/~tsuruoka/maxent/
[9] http://code.google.com/p/darts-clone/

Model type			Model statistics		Dep.	Sent.
Model	d	ω / σ	$\lvert\mathcal{F}^d\rvert$	$\lvert\boldsymbol{x}^d\rvert$	acc.	acc.
SVM-KE	1	0	39712	27.3	88.29	46.49
SVM-KE	2	0	1478109	380.6	90.76	53.83
SVM-KE	3	0	26194354	3286.7	90.93$^{\gg}$	54.43$^{\gg}$
SVM-HKE	3	0.001	13247675	2725.9	90.92$^{\gg}$	54.39$^{\gg}$
SVM-HKE	3	0.002	2514385	2238.1	90.91$^{\gg}$	54.32$^{>}$
SVM-HKE	3	0.003	793195	1855.4	90.83	54.21
SVM-KE	4	0	293416102	20395.4	90.91$^{\gg}$	**54.69**$^{\gg}$
SVM-HKE	4	0.0002	96522236	15282.1	90.93$^{\gg}$	54.53$^{>}$
SVM-HKE	4	0.0004	19245076	11565.0	**90.96**$^{\gg}$	54.64$^{\gg}$
SVM-HKE	4	0.0006	7277592	8958.2	90.84	54.48$^{>}$
ℓ_1-LLM	1	1.0	9268	26.5	88.22	46.06
ℓ_1-LLM	2	2.0	32575	309.8	90.62	53.46
ℓ_1-LLM	3	3.0	129503	2088.3	90.71	54.09$^{>}$
ℓ_1-LLM	3	4.0	85419	1803.0	90.61	53.79
ℓ_1-LLM	3	5.0	63046	1699.5	90.59	53.55

Table 2: Specifications of LLMs and SVMs. The accuracy marked with '$\gg$' or '$>$' was significantly better than the $d = 2$ counterpart ($p < 0.01$ or $0.01 \leq p < 0.05$ by McNemar's test).

4.2 Results

Specifications of SVMs and LLMs used here are shown in Table 2; $|\mathcal{F}^d|$ is the number of active features, while $|\boldsymbol{x}^d|$ is the average number of active features in each classification for the test corpus. Dependency 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.

For LLM training, we designed explicit conjunctive features for all the d or lower-order feature combinations to make the results comparable to those of SVMs. We could not train $d = 4$ LLMs due to parameter explosion. We varied SVM soft margin parameter c from 0.1 to 0.000001 and LLM width factor parameter ω,[10] which controls the impact of the prior, from 1.0 to 5.0, and adjusted the values to maximize dependency accuracy for the development set: $(d, c) = (1, 0.1), (2, 0.005), (3, 0.0001), (4, 0.000005)$ for SVMs and $(d, \omega) = (1, 1.0), (2, 2.0), (3, 4.0)$ for ℓ_1-LLMs.

The accuracy of around 90.9% (SVM-KE, $d = 3, 4$) is close to the performance of state-of-the-

[10] The parameter C of ℓ_1-LLM in Eq. 2 was set to ω/L (referred to in Kazama and Tsujii (2003) as '*single width*').

Model type	d	PKI classify [ms/sent.]	Baseline Mem. (MB)	Baseline Time [ms/sent.] classify (total)	Proposed w/ fstrie$_S$ Mem. (MB)	Proposed w/ fstrie$_S$ Time [ms/sent.] classify (total)	Proposed w/ fstrie$_M$ Mem. (MB)	Proposed w/ fstrie$_M$ Time [ms/sent.] classify (total)	Proposed w/ fstrie$_L$ Mem. (MB)	Proposed w/ fstrie$_L$ Time [ms/sent.] classify (total)	Speed up
SVM-KE	1	13.480	0.2	**0.003** (0.015)	+0.6	0.006 (0.018)	+20.2	0.007 (0.018)	+662.9	0.016 (0.029)	NA
SVM-KE	2	10.313	13.5	0.041 (0.054)	+0.5	**0.020** (0.032)	+18.0	0.021 (0.034)	+662.4	0.023 (0.036)	2.1
SVM-KE	3	10.945	142.2	0.345 (0.361)	+0.5	0.163 (0.178)	+18.2	0.108 (0.123)	+667.0	**0.079** (0.093)	4.4
SVM-KE	4	12.603	648.0	2.338 (2.363)	+0.5	1.156 (1.178)	+18.6	0.671 (0.690)	+675.9	**0.415** (0.432)	5.6

Table 3: Parsing results for test corpus: SVM-KE classifiers with dense feature space.

art parsers (Iwatate et al., 2008), and the model statistics are considered to be complex (or realistic) enough to evaluate our classifier's utility. The number of support vectors of SVMs was $71,766 \pm 9.2\%$, which is twice as many as those used by Kudo and Matsumoto (2003) (34,996) in their experiments on the same task.

We could clearly observe that the number of active features $|\boldsymbol{x}^d|$ increased dramatically according to the order d of feature combinations. The density of $|\boldsymbol{x}^d|$ for SVMs was very high (*e.g.*, $|\boldsymbol{x}^3| = 3286.7$, close to the maximum shown in Eq. 8: $(27.3^3 + 5 \times 27.3)/6 \simeq 3414$.

For $d \geq 3$ 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-HKE 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 a raw corpus in the target domain, 3,258,313 sentences of 1991–94 Mainichi news articles that were morphologically analyzed by JUMAN[6] and segmented into bunsetsus by KNP.[6] We first built fstrie$_L$ using all the source feature vectors. We then attempted to reduce the number of prefix feature vectors in fstrie$_L$ to $1/2^n$ the size by Algorithm 1. We refer to fstries built from 1/32 and 1/1024 of the prefix feature vectors in fstrie$_L$ as fstrie$_M$ and fstrie$_S$ in the following experiments.

Because we exploited Algorithm 2 to calculate the weights of the prefix feature vectors, it took less than one hour (59 min. 29 sec.) on the 3.20-GHz server to build fstrie$_L$ (and calculate the utility score for all the nodes in it) for the slowest SVM-KE ($d = 4$) from the 40,409,190 source feature vectors (62,654,549 prefix feature vectors) generated by parsing the 3,258,313 sentences.

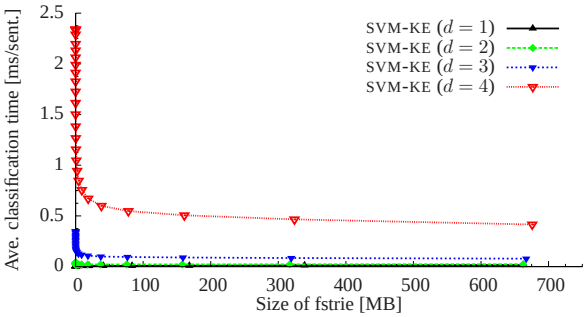

Figure 3: Average classification time per sentence plotted against size of fstrie: SVM-KE.

Results for SVM-KE with dense feature space The performances of parsers having SVM-KE classifiers with and without the fstrie are given in Table 3. The 'speed-up' column shows the speed-up factor of the most efficient classifier (bold) versus the baseline classifier without fstries. Since each classifier solved a slightly different number of classification steps ($112,853 \pm 0.15\%$), we show the (average) cumulative classification time for a sentence. The Mem. columns show the size of weight vectors for SVM-KE classifiers and the size of fstries$_S$, fstries$_M$, and fstries$_L$, respectively.

The fstries successfully speeded up SVM-KE classifiers with the dense feature space.[11] The SVM-KE classifiers without fstries were still faster than PKI, but 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 by factors of 13 ($d = 2$), 109 ($d = 3$) and 738 ($d = 4$) and the classification time accounted for most of the parsing time.

The average classification time of our classifiers plotted against fstrie size is shown in Figure 3. Surprisingly, we obtained a significant speed-up even with tiny fstrie sizes of < 1 MB. Furthermore, we naively controlled the fstrie size by sim-

[11] The inefficiency of the classifier ($d = 1$) results from the cost of the additional sort function (line 1 in Algorithm 2) and CPU cache failure due to random accesses to the huge fstries.

Model type	d	σ / ω	Baseline Mem. (MB)	Baseline Time [ms/sent.] classify (total)	Proposed w/ fstrie$_S$ Mem. (MB)	Proposed w/ fstrie$_S$ Time [ms/sent.] classify (total)	Proposed w/ fstrie$_M$ Mem. (MB)	Proposed w/ fstrie$_M$ Time [ms/sent.] classify (total)	Proposed w/ fstrie$_L$ Mem. (MB)	Proposed w/ fstrie$_L$ Time [ms/sent.] classify (total)	Speed up
SVM-HKE	3	0.001	64.6	0.348 (0.363)	+0.5	0.151 (0.166)	+17.6	0.097 (0.111)	+638.0	**0.070** (0.084)	5.0
SVM-HKE	3	0.002	13.9	0.332 (0.346)	+0.5	0.123 (0.137)	+17.0	0.074 (0.088)	+612.2	**0.053** (0.067)	6.2
SVM-HKE	3	0.003	4.2	0.314 (0.328)	+0.4	0.102 (0.115)	+14.7	0.057 (0.070)	+526.2	**0.041** (0.054)	7.8
SVM-HKE	4	0.0002	235.0	2.258 (2.280)	+0.5	1.022 (1.042)	+17.7	0.558 (0.575)	+637.1	**0.330** (0.346)	6.8
SVM-HKE	4	0.0004	82.8	2.038 (2.058)	+0.5	0.816 (0.835)	+16.8	0.414 (0.430)	+601.7	**0.234** (0.249)	8.7
SVM-HKE	4	0.0006	32.2	1.802 (1.820)	+0.4	0.646 (0.662)	+15.7	0.311 (0.326)	+558.9	**0.168** (0.183)	10.7
ℓ_1-LLM	1	1.0	0.1	**0.004** (0.016)	+0.8	0.006 (0.018)	+25.0	0.007 (0.019)	+787.7	0.016 (0.029)	NA
ℓ_1-LLM	2	2.0	0.4	0.043 (0.055)	+0.6	0.016 (0.028)	+20.5	**0.015** (0.027)	+698.0	0.018 (0.030)	2.9
ℓ_1-LLM	3	3.0	1.0	0.314 (0.326)	+0.5	0.091 (0.103)	+17.8	0.041 (0.054)	+601.0	**0.027** (0.040)	11.6
ℓ_1-LLM	3	4.0	0.7	0.300 (0.313)	+0.5	0.082 (0.094)	+16.3	0.036 (0.049)	+550.1	**0.024** (0.037)	12.4
ℓ_1-LLM	3	5.0	0.5	0.290 (0.302)	+0.5	0.076 (0.088)	+15.1	0.032 (0.045)	+510.7	**0.022** (0.035)	13.3

Table 4: Parsing results for test corpus: SVM-HKE and ℓ_1-LLM classifiers with sparse feature space.

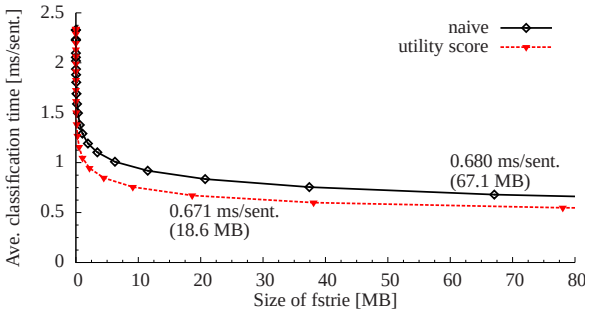

Figure 4: Fstrie reduction: utility score vs. processed sentence reduction for SVM-KE ($d = 4$).

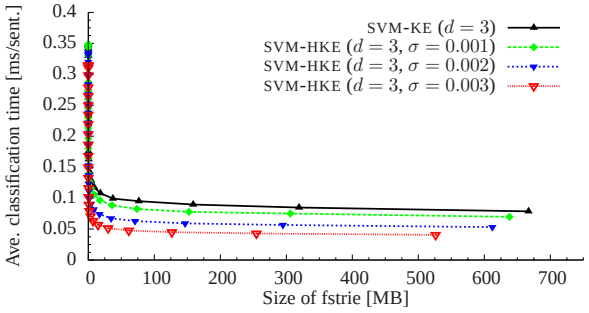

Figure 5: Average classification time per sentence plotted against size of fstrie: SVM-HKE ($d = 3$).

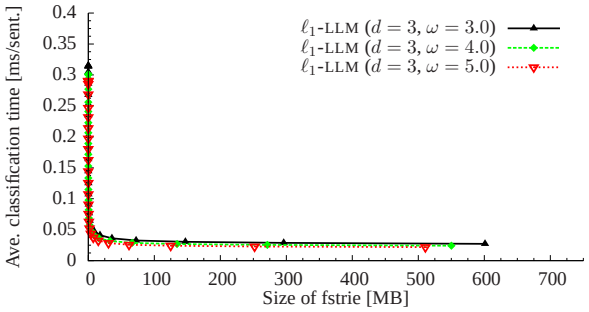

Figure 6: Average classification time per sentence plotted against size of fstrie: ℓ_1-LLM ($d = 3$).

ply 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-KE ($d = 4$) is shown in Figure 4. 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 1/3 size fstries to achieve the same speed-up: 0.671 ms./sent. (18.6 MB) (*utility-score*) vs. 0.680 ms./sent. (67.1 MB) (*naive*).

Results for SVM-HKE and ℓ_1-LLM classifiers with sparse feature space The performances of parsers having SVM-HKE and ℓ_1-LLM classifiers with and without the fstrie are given in Table 4. The fstries successfully speeded up the SVM-HKE and ℓ_1-LLM classifiers by factors of 10.7 (SVM-HKE, $d = 4$, $\sigma = 0.0006$) and 11.6 (ℓ_1-LLM, $d = 3$, $\omega = 3.0$). We obtained more speed-up when we used fstries for classifiers with more sparse feature space $\mathcal{F}^d$ (Figures 5 and 6). The parsing speed with $d = 3$ models are now comparable to the parsing speed with $d = 2$ models.

Without fstries, little speed-up of SVM-HKE classifiers versus the SVM-KE classifiers (in Table 3) was obtained due to the mild reduction in the average number of active features $|\boldsymbol{x}^d|$ in the classification. This result conforms to the results reported in (Kudo and Matsumoto, 2003).

The parsing speed reached 14,937 sentences per second with accuracy of 90.91% (SVM-HKE, $d = 3$, $\sigma = 0.002$). We used this parser to process 1,005,918 sentences (5,934,184 bunsetsus) randomly extracted from Japanese weblog feeds

updated in November 2008, 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 156.4 sec. without fstrie$_{\mathrm{L}}$, while it was just 35.9 sec. with fstrie$_{\mathrm{L}}$. The speed-up factor of 4.4 on weblog feeds was slightly worse than that on news articles ($0.346/0.067 = 5.2$) but still evident. This implies that sorting features in building fstries yielded prefix features vectors that commonly appear in this 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 Conclusion and Future Work

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 Japanese dependency parsing task by a factor of 10. We also confirmed that the sparse feature space provided by ℓ_1-LLMs and SVM-HKEs 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 Japanese dependency parser will be available at http://www.tkl.iis.u-tokyo.ac.jp/~ynaga/.

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 (Ganchev and Dredze, 2008) or a memory-efficient trie implementation based on a succinct data structure (Jacobson, 1989; Delpratt et al., 2006) to reduce required memory usage.

We will combine our method with other techniques that provide sparse solutions, for example, kernel methods on a budget (Dekel and Singer, 2007; Dekel et al., 2008; Orabona et al., 2008) or kernel approximation (surveyed in Kashima et al. (2009)). It is also easy to combine our method with SVMs with partial kernel expansion (Goldberg and Elhadad, 2008), which will yield slower but more space-efficient classifiers. We will in the future consider an issue of speeding up decoding with structured models (Lafferty et al., 2001; Miyao and Tsujii, 2002; Sutton et al., 2004).

Acknowledgment The authors wish to thank Susumu Yata and Yoshimasa Tsuruoka for letting the authors to use their pre-release libraries. The authors also thank Nobuhiro Kaji and the anonymous reviewers for their valuable comments.

References

Galen Andrew and Jianfeng Gao. 2007. Scalable training of L_1-regularized log-linear models. In *Proc. ICML 2007*, pages 33–40.

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

Adam Berger, Stephen Della Pietra, and Vincent Della Pietra. 1996. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71, March.

Kun-Ta Chuang, Jiun-Long Huang, and Ming-Syan Chen. 2008. Power-law relationship and self-similarity in the itemset support distribution: analysis and applications. *The VLDB Journal*, 17(5):1121–1141, August.

Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine Learning*, 20(3):273–297, September.

Ofer Dekel and Yoram Singer. 2007. Support vector machines on a budget. In Bernhard Schölkopf, John Platt, and Thomas Hofmann, editors, *Advances in Neural Information Processing Systems 19*, pages 345–352. The MIT Press.

Ofer Dekel, Shai Shalev-Shwartz, and Yoram Singer. 2008. The forgetron: A kernel-based perceptron on a budget. *SIAM Journal on Computing*, 37(5):1342–1372, January.

O'Neil Delpratt, Naila Rahman, and Rajeev Raman. 2006. Engineering the LOUDS succinct tree representation. In *Proc. WEA 2006*, pages 134–145.

Leo Egghe. 2000. The distribution of n-grams. *Scientometrics*, 47(2):237–252, February.

Kuzman Ganchev and Mark Dredze. 2008. Small statistical models by random feature mixing. In *Proc. ACL 2008 Workshop on Mobile Language Processing*, pages 19–20.

Jianfeng Gao, Galen Andrew, Mark Johnson, and Kristina Toutanova. 2007. A comparative study

of parameter estimation methods for statistical natural language processing. In *Proc. ACL 2007*, pages 824–831.

Yoav Goldberg and Michael Elhadad. 2008. splitSVM: Fast, space-efficient, non-heuristic, polynomial kernel computation for NLP applications. In *Proc. ACL 2008, Short Papers*, pages 237–240.

Joshua Goodman. 2004. Exponential priors for maximum entropy models. In *Proc. HLT-NAACL 2004*, pages 305–311.

Hideki Isozaki and Hideto Kazawa. 2002. Efficient support vector classifiers for named entity recognition. In *Proc. COLING 2002*, pages 1–7.

Masakazu Iwatate, Masayuki Asahara, and Yuji Matsumoto. 2008. Japanese dependency parsing using a tournament model. In *Proc. COLING 2008*, pages 361–368.

Guy Jacobson. 1989. Space-efficient static trees and graphs. In *Proc. FOCS 1989*, pages 549–554.

Hisashi Kashima, Tsuyoshi Idé, Tsuyoshi Kato, and Masashi Sugiyama. 2009. Recent advances and trends in large-scale kernel methods. *IEICE Transactions on on Information and Systems*, E92-D. to appear.

Jun'ichi Kazama and Jun'ichi Tsujii. 2003. Evaluation and extension of maximum entropy models with inequality constraints. In *Proc. EMNLP 2003*, pages 137–144.

Jun'ichi Kazama and Jun'ichi Tsujii. 2005. Maximum entropy models with inequality constraints: A case study on text categorization. *Machine Learning*, 60(1-3):159–194.

Terry Koo, Xavier Carreras, and Michael Collins. 2008. Simple semi-supervised dependency parsing. In *Proc. ACL 2008*, pages 595–603.

Taku Kudo and Yuji Matsumoto. 2002. Japanese dependency analysis using cascaded chunking. In *Proc. CoNLL 2002*, pages 1–7.

Taku Kudo and Yuji Matsumoto. 2003. Fast methods for kernel-based text analysis. In *Proc. ACL 2003*, pages 24–31.

Sadao Kurohashi and Makoto Nagao. 2003. Building a Japanese parsed corpus. In Anne Abeillé, editor, *Treebank: Building and Using Parsed Corpora*, pages 249–260. Kluwer Academic Publishers.

John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. ICML 2001*, pages 282–289.

Yudong Liu and Anoop Sarkar. 2007. Experimental evaluation of LTAG-based features for semantic role labeling. In *Proc. EMNLP 2007*, pages 590–599.

David McClosky, Eugene Charniak, and Mark Johnson. 2006. Effective self-training for parsing. In *Proc. HLT-NAACL 2006*, pages 152–159.

Yusuke Miyao and Jun'ichi Tsujii. 2002. Maximum entropy estimation for feature forests. In *Proc. HLT 2002*, pages 292–297.

Ngan L.T. Nguyen and Jin-Dong Kim. 2008. Exploring domain differences for the design of a pronoun resolution system for biomedical texts. In *Proc. COLING 2008*, pages 625–632.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proc. IWPT 2003*, pages 149–160.

Daisuke Okanohara and Jun'ichi Tsujii. 2009. Learning combination features with L_1 regularization. In *Proc. HLT-NAACL 2009, Short Papers*, pages 97–100.

Francesco Orabona, Joseph Keshet, and Barbara Caputo. 2008. The projectron: a bounded kernel-based perceptron. In *Proc. ICML 2008*, pages 720–727.

Patrick Pantel. 2007. Data catalysis: Facilitating large-scale natural language data processing. In *Proc. ISUC*, pages 201–204.

Stijn De Saeger, Kentaro Torisawa, and Jun'ichi Kazama. 2009. Mining web-scale treebanks. In *Proc. NLP 2009*, pages 837–840.

Manabu Sassano. 2004. Linear-time dependency analysis for Japanese. In *Proc. COLING 2004*, pages 8–14.

Drahomíra "Johanka" Spoustová, Jan Hajič, Jan Raab, and Miroslav Spousta. 2009. Semi-supervised training for the averaged perceptron POS tagger. In *Proc. EACL 2009*, pages 763–771.

Charles Sutton, Khashayar Rohanimanesh, and Andrew McCallum. 2004. Dynamic conditional random fields: factorized probabilistic models for labeling and segmenting sequence data. In *Proc. ICML 2004*, pages 783–790.

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

Yu-Chieh Wu, Jie-Chi Yang, and Yue-Shi Lee. 2007. An approximate approach for training polynomial kernel SVMs in linear time. In *Proc. ACL 2007 Poster and Demo Sessions*, pages 65–68.

Susumu Yata, Kazuhiro Morita, Masao Fuketa, and Jun'ichi Aoe. 2008. Fast string matching with space-efficient word graphs. In *Proc. Innovations in Information Technology 2008*, pages 79–83.

George K. Zipf. 1949. *Human Behavior and the Principle of Least-Effort*. Addison-Wesley.