A Self-adaptive Classifier for Efficient Text-stream Processing

Naoki Yoshinaga
Institute of Industrial Science,
the University of Tokyo,
Meguro-ku, Tokyo 153-8505, Japan
ynaga@tkl.iis.u-tokyo.ac.jp

Masaru Kitsuregawa
Institute of Industrial Science,
the University of Tokyo,
Meguro-ku, Tokyo 153-8505, Japan
and
National Institute of Informatics,
Chiyoda-ku, Tokyo 101-8430, Japan
kitsure@tkl.iis.u-tokyo.ac.jp

Abstract

A self-adaptive classifier for efficient text-stream processing is proposed. The proposed classifier adaptively speeds up its classification while processing a given text stream for various NLP tasks. The key idea behind the classifier is to reuse results for past classification problems to solve forthcoming classification problems. A set of classification problems commonly seen in a text stream is stored to reuse the classification results, while the set size is controlled by removing the least-frequently-used or least-recently-used classification problems. Experimental results with Twitter streams confirmed that the proposed classifier applied to a state-of-the-art base-phrase chunker and dependency parser speeds up its classification by factors of 3.2 and 5.7, respectively.

1 Introduction

The rapid growth in popularity of microblogs (*e.g.*, Twitter) is enabling more and more people to instantly publish their experiences or thoughts any time they want from mobile devices. Since information in text posted by hundreds of millions of those people covers every space and time in the real world, analyzing such a text stream tells us what is going on in the real world and is therefore beneficial for reducing damage caused by natural disasters (Sakaki et al., 2010; Neubig et al., 2011a; Varga et al., 2013), monitoring political sentiment (Tumasjan et al., 2010) and disease epidemics (Aramaki et al., 2011), and predicting stock market (Gilbert and Karahalios, 2010) and criminal incident (Wang et al., 2012).

Text-stream processing, however, faces a new challenge; namely, the quality (content) and quantity (volume of flow) changes dramatically, reflecting a change in the real world. Current studies on processing microblogs have focused mainly on the difference between the quality of microblogs (or spoken languages) and news articles (or written languages) (Gimpel et al., 2011; Foster et al., 2011; Ritter et al., 2011; Han and Baldwin, 2011), and they have not addressed the issue of so-called "bursts" that increase the volume of text. Although it is desirable to use NLP analyzers with the highest possible accuracy for processing a text stream, high accuracy is generally attained by costly structured classification or classification with rich features, typically conjunctive features (Liang et al., 2008). It is therefore inevitable to trade accuracy for speed by using only a small fraction of features to assure real-time processing.

In this study, the aforementioned text-quantity issue concerning processing a text stream is addressed, and a self-adaptive algorithm that speeds up an NLP classifier trained with many conjunctive features (or with a polynomial kernel) for a given text stream is proposed and validated. Since globally-observable events such as natural disasters or sports events incur a rapid growth in the number of posts (Twitter, Inc., 2011), a text stream is expected to contain similar contents concerning these events when the volume of flow in a text stream increases. To adaptively speed up the NLP classifier, the proposed algorithm thus enumerates common classification problems from seen classification problems and keeps their classification results as partial results for use in solving forthcoming classification problems.

The proposed classifier was evaluated by applying it to streams of classification problems generated during the processing of the Twitter streams on the day of the 2011 Great East Japan Earthquake and on another day in March 2012 using a state-of-the-art base-phrase chunker (Sassano, 2008) and dependency parser (Sassano, 2004), and the obtained results confirm the effectiveness of the proposed algorithm.

2 Related work

A sentence is the processing unit used for fundamental NLP tasks such as word segmentation, part-of-speech tagging, phrase chunking, syntactic parsing, and semantic role labeling. Most efficient algorithms solving these tasks thus aim at speeding up the processing based on this unit (Kaji et al., 2010; Koo et al., 2010; Rush and Petrov, 2012), and few studies have attempted to speed up the processing of a given text (a set of sentences) as a whole. In the following, reported algorithms that adaptively speed up NLP analyzers for a given text are introduced.

A method of speeding up a classifier trained with many conjunctive features by using precomputed results for common classification problems was proposed by Yoshinaga and Kitsuregawa (2009; 2012). It solves classification problems that commonly appear in the processing of a large amount of text in advance and stores the results in a trie, so that they can be reused as partial results for solving new classification problems. This method was reported to achieve speed-up factors of 3.3 and 10.6 for base-phrase chunking and dependency parsing, respectively. An analogous algorithm for integer linear program (ILP) used to solve structured classification was proposed by Srikumar, Kundu, and Roth (2012; 2013). The algorithm was reported to achieve speed-up factors of 2.6 and 1.6 for semantic role labeling and entity-relation extraction, respectively. Although these two algorithms can be applied to various NLP tasks that can be solved by using a linear classifier or an ILP solver, how effective they are for processing a text stream is not clear.

A method of feature sharing for beam-search incremental parsers was proposed by Goldberg et al. (2013). Motivated by the observation that beam parsers solve similar classification problems in different parts of the beam, this method reuses partial results computed in the previous beam items. It reportedly achieved a speed-up factor of 1.2 for arc-standard and arc-eager dependency parsers. The key differences between the method proposed in this study and their feature-sharing method are twofold. First, the feature sharing in Goldberg et al. (2013) is performed in a token-wise manner in the sense that a key to retrieve a cached result is represented by a bag of tokens that invoke features, which manner prevents fine-grained caching. Second, the feature sharing is dynamically performed during parsing, but the cached results are cleared after processing each sentence.

An adaptive pruning method for fast HPSG parsing was proposed by van Noord (2009). This method preprocesses a large amount of text by using a target parser to collect derivation steps that are unlikely to contribute to the best parse, and it speeds up the parser by filtering out those unpromising derivation steps. Although this method was reported to attain a speed-up factor of four while keeping parsing accuracy, it needs to be tuned to trade parsing accuracy and speed for each domain. It is difficult to derive the true potential of their method in regard to processing a text stream whose domain shifts from time to time.

It has been demonstrated by Wachsmuth et al. (2011) that tuning a pipeline schedule of an information extraction (IE) system improves the efficiency of the system. Furthermore, the self-supervised learning algorithm devised by Wachsmuth et al. (2013) predicts the processing time for each possible pipeline schedule of an IE system, and the prediction is used to adaptively change the pipeline schedule for a given text stream. This method and the proposed method for speeding up an NLP classifier are complementary, and a combination of both methods is expected to synergistically speed up various NLP-systems.

In this study, based on the classifier proposed by Yoshinaga and Kitsuregawa (2009), a self-adaptive classifier that enumerates common classification problems from a given text stream and reuses their results is proposed. As a result, the proposed classifier adaptively speeds up the classification of forthcoming classification problems.

3 Preliminaries

As the basis of the proposed classifier, the previously-presented classifier that uses results of common classification problems (Yoshinaga and Kitsuregawa, 2009) is described as follows. This base classifier targets a linear classifier trained with many conjunctive features (including one converted from a classifier trained with polynomial kernel (Isozaki and Kazawa, 2002)) that are widely used for many NLP tasks. Although this classifier (and also the one proposed in this paper) can handle a multi-class classification problem, a binary classification problem is assumed here for brevity.

A binary classifier such as a perceptron and a support vector machine determines label $y \in \{+1, -1\}$ of input classification problem $\boldsymbol{x}$ by using the following equation (from which the bias term is omitted for brevity):

$$m(\boldsymbol{x}; \boldsymbol{\phi}, \boldsymbol{w}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}) = \sum_i w_i \phi_i(\boldsymbol{x}) \tag{1}$$

$$y = \begin{cases} +1 & (m(\boldsymbol{x}; \boldsymbol{\phi}, \boldsymbol{w}) \geq 0) \\ -1 & (m(\boldsymbol{x}; \boldsymbol{\phi}, \boldsymbol{w}) < 0). \end{cases} \tag{2}$$

Here, ϕ_i is a feature function, w_i is a weight for ϕ_i obtained as a result of training, and $m(\boldsymbol{x}; \boldsymbol{\phi}, \boldsymbol{w})$ is a margin between $\boldsymbol{x}$ and the separating hyperplane.

In most NLP tasks, feature functions are mostly indicator (or binary) functions that typically represent particular linguistic constraints. Here, feature functions are assumed to be indicator functions that return $\{0, 1\}$, and margin $m(\boldsymbol{x}; \boldsymbol{\phi}, \boldsymbol{w})$ is represented by the following equation:

$$m(\boldsymbol{x}; \boldsymbol{\phi}, \boldsymbol{w}) = \sum_i w_i \phi_i(\boldsymbol{x}) = \sum_{i, \phi_i(\boldsymbol{x})=1} w_i. \tag{3}$$

Feature function ϕ_i is hereafter referred to as feature ϕ_i; when $\phi_i(\boldsymbol{x}) = 1$ for given $\boldsymbol{x}$, $\boldsymbol{x}$ is said to "include" feature ϕ_i or feature ϕ_i is "active" in $\boldsymbol{x}$ (denoted as $\phi_i \in \boldsymbol{x}$). Having the number of active features, $|\boldsymbol{\phi}(\boldsymbol{x})| \equiv |\{\phi_i \mid \phi_i \in \boldsymbol{x}\}|$, Eq. 3 requires $\mathcal{O}(|\boldsymbol{\phi}(\boldsymbol{x})|)$ when the weights for the active features are summed up.

To speed up the summation in Eq. 3, classification results for some classification problems $\boldsymbol{x}_c$ are precomputed as $M_{\boldsymbol{x}_c} \equiv m(\boldsymbol{x}_c; \boldsymbol{\phi}, \boldsymbol{w})$ in advance, and then these precomputed results are reused as partial results for solving an input classification problem, $\boldsymbol{x}$:

$$m(\boldsymbol{x}; \boldsymbol{x}_c, \boldsymbol{\phi}, \boldsymbol{w}) = M_{\boldsymbol{x}_c} + \sum_{i, \phi_i \in \boldsymbol{x}, \phi_i \notin \boldsymbol{x}_c} w_i \tag{4}$$
$$\textit{where} \quad \forall \phi_j \in \boldsymbol{x}_c, \phi_j \in \boldsymbol{x}.$$

Note that for Eq. 4 to be computed faster than Eq. 3, $M_{\boldsymbol{x}_c}$ must be retrieved in time less than $\mathcal{O}(|\boldsymbol{\phi}(\boldsymbol{x}_c)|)$. It is actually possible when $\boldsymbol{x}_c$ includes conjunctive feature $\phi_{i,j}(\boldsymbol{x}_c) = \phi_i(\boldsymbol{x}_c)\phi_j(\boldsymbol{x}_c)$. If it is necessary to retrieve margin $M_{\boldsymbol{x}_c}$ precomputed for $\boldsymbol{x}_c$ including ϕ_i, ϕ_j, and $\phi_{i,j}$, it is necessary to check only non-conjunctive (or *primitive*) features ϕ_i and ϕ_j, since $\phi_{i,j}$ is active whenever ϕ_i and ϕ_j are active (so checking $\phi_{i,j}$ can be skipped). The second term of Eq. 4 sums up the weights of the remaining features that are not included in $\boldsymbol{x}_c$ but are included in $\boldsymbol{x}$. For example, under the assumption that $\boldsymbol{x}$ includes features ϕ_i, ϕ_j, ϕ_k, $\phi_{i,j}$, $\phi_{i,k}$, and $\phi_{j,k}$ and that margin $M_{\boldsymbol{x}_c}$ has been obtained for $\boldsymbol{x}_c$ (including ϕ_i, ϕ_j, and $\phi_{i,j}$), five features must be checked (two to retrieve $M_{\boldsymbol{x}_c}$ and three to sum up the weights of the remaining features ϕ_k, $\phi_{i,k}$ and $\phi_{j,k}$) by using Eq. 4. On the other hand, to compute $m(\boldsymbol{x}; \boldsymbol{\phi}, \boldsymbol{w})$ by Eq. 3, the weights for the six features must be checked.

To maximize the speed-up obtained by Eq. 4, reuse of margin $M_{\boldsymbol{x}_c}$ of common classification problem $\boldsymbol{x}_c$ should minimize the number of remaining features included only in $\boldsymbol{x}$. In other words, $\boldsymbol{x}_c$ should be as similar to $\boldsymbol{x}$ as possible (ideally, $\boldsymbol{x}_c = \boldsymbol{x}$). It is not, however, realistic to precompute margin $M_{\boldsymbol{x}} \equiv m(\boldsymbol{x}; \boldsymbol{\phi}, \boldsymbol{w})$ for every possible classification problem $\boldsymbol{x}$ since it requires $\mathcal{O}(2^{|\boldsymbol{\phi}'|})$ space where $|\boldsymbol{\phi}'|$ is the number of primitive features ($\boldsymbol{\phi}' \subset \boldsymbol{\phi}$) and $|\boldsymbol{\phi}'|$ is usually more than 10,000 in NLP tasks due to lexical features. Yoshinaga and Kitsuregawa (2009) therefore preprocess a large amount of text to enumerate possible classification problems, and select common classification problems, $\mathcal{X}_c \subset 2^{\boldsymbol{\phi}'}$, according to their probability and the reduction in the number of features to be checked by Eq 4.

Yoshinaga and Kitsuregawa (2009) then represent the common classification problems $\boldsymbol{x}_c \in \mathcal{X}_c$ by sequences of active primitive feature indices, and store those feature (index) sequences as keys in a prefix trie with precomputed margin $M(\boldsymbol{x}_c)$ as their values. To reuse a margin of common classification problem that is similar to input $\boldsymbol{x}$ in Eq. 4, features are ordered according to their frequency to form a feature sequence of $\boldsymbol{x}_c$. A longest-prefix search for the trie thereby retrieves a common classification problem similar to the input classification problem in linear time with respect to the number of primitive features in $\boldsymbol{x}_c$, $\mathcal{O}(|\boldsymbol{\phi}'(\boldsymbol{x}_c)|)$.

Algorithm 1 A self-adaptive classifier for enumerating common classification problems

```
INPUT: $\boldsymbol{x}, \boldsymbol{\phi}, \boldsymbol{\phi}' \subset \boldsymbol{\phi}, \boldsymbol{w} \in \mathbb{R}^{|\boldsymbol{\phi}|}, \mathcal{X}_c \subset 2^{\boldsymbol{\phi}'}, k > 0$
OUTPUT: $m(\boldsymbol{x}; \boldsymbol{\phi}, \boldsymbol{w}) \in \mathbb{R}, \mathcal{X}_c$
1: INITIALIZE: $\boldsymbol{x}_c$ s.t. $\boldsymbol{\phi}'(\boldsymbol{x}_c) = \boldsymbol{0}$, $M_{\boldsymbol{x}_c} \leftarrow 0$
2: repeat
3:   $\boldsymbol{x}_c^{old} \leftarrow \boldsymbol{x}_c$
4:   $\phi'_i = \underset{\phi'_i \in \boldsymbol{x}, \phi'_i \notin \boldsymbol{x}_c}{\text{argmax}} \ \text{FREQ}(\phi'_i)$        (extract a primitive feature according to its frequency)
5:   $\phi'_i(\boldsymbol{x}_c) \leftarrow 1$        (construct a new common-classification problem)
6:   if $\boldsymbol{x}_c \notin \mathcal{X}_c$ then
7:     $M_{\boldsymbol{x}_c} \leftarrow m(\boldsymbol{x}_c; \boldsymbol{x}_c^{old}, \boldsymbol{\phi}, \boldsymbol{w})$        (compute margin by using Eq. 4)
8:     if $|\mathcal{X}_c| = k$ then
9:       $\mathcal{X}_c \leftarrow \mathcal{X}_c - \{\text{USELESS}(\mathcal{X}_c)\}$
10:    $\mathcal{X}_c \leftarrow \mathcal{X}_c \cup \{\boldsymbol{x}_c\}$
11: until $\boldsymbol{\phi}'(\boldsymbol{x}_c) \neq \boldsymbol{\phi}'(\boldsymbol{x})$
12: return $m(\boldsymbol{x}; \boldsymbol{\phi}, \boldsymbol{w}) = M_{\boldsymbol{x}_c}, \mathcal{X}_c$
```

4 Proposed method

The classifier described in Section 3 is extended so that it dynamically enumerates common classification problems from a given text stream[1] to adaptively speed up the classification. This classification "speed up" faces two challenges: which (partial) classification problems should be chosen to reuse their results from a given stream of classification problems, and how to efficiently maintain the extracted common classification problems. These two challenges are addressed in Sections 4.1 and 4.2, respectively.

4.1 Enumerating common classification problems dynamically from a text stream

Although Yoshinaga and Kitsuregawa (2009) select common classification problems according to their probability, such statistics cannot be known before a target text stream is entirely seen. A set of common classification problems was thus kept updated adaptively while processing a text stream; that is, classification problems are added when they will be useful, while they are removed when they will be useless, so that the number of common classification problems, $|\mathcal{X}_c|$, does not exceed a pre-defined threshold, k.

Algorithm 1 depicts the proposed self-adaptive classifier for enumerating common classification problems from an input classification problem, $\boldsymbol{x}$. To incrementally construct common classification problem $\boldsymbol{x}_c$ (Line 4-5), the algorithm extracts the primitive features (ϕ'_i) included in $\boldsymbol{x}$ one by one according to their probability of appearing in the training data of the classifier. When the resulting $\boldsymbol{x}_c$ is included in the current set of common classification problems, $\mathcal{X}_c$, stored margin $M_{\boldsymbol{x}_c}$ is reused. Otherwise, margin $M_{\boldsymbol{x}_c} = m(\boldsymbol{x}_c; \boldsymbol{x}_c^{old}, \boldsymbol{\phi}, \boldsymbol{w})$ is computed by using Eq. 4 (Line 7), and $\boldsymbol{x}_c$ is registered in $\mathcal{X}_c$ as a new common classification problem (Line 10).

An important issue is how to define function USELESS, which selects a common classification problem that will not contribute to speeding up the forthcoming classification, when the number of common classification problems, $|\mathcal{X}_c|$, reaches the pre-defined threshold k. To address this issue, the following two policies (designed originally for CPU caching) are proposed and compared in terms of the efficiency of the classifier in experiments:

Least Frequently Used (LFU) This policy counts frequency of common classification problems in a seen text stream, and it maintains only the top-k common classification problems by removing the least-common classification problem from $\mathcal{X}_c$:

$$\text{USELESS}_{\text{LFU}}(\mathcal{X}_c) = \underset{\boldsymbol{x}_c \in \mathcal{X}_c}{\text{argmin}} \, \text{FREQ}(\boldsymbol{x}_c) \tag{5}$$

[1]More precisely, a stream of classification problems generated during the analysis of a text stream.

A space-saving algorithm, (Metwally et al., 2005), is used to efficiently count the approximated frequency of k classification problems at most and to remove the common classification problem rejected by the space-saving algorithm.

Least Recently Used (LRU) When the volume of flow in a text stream rapidly increases, it is likely to relate to a burst of a certain topic. To exploit this characteristics, this policy preserves k common classification problems whose results are most recently reused:

$$\text{USELESS}_{\text{LRU}}(\mathcal{X}_c) = \operatorname*{argmin}_{\boldsymbol{x}_c \in \mathcal{X}_c} \text{TIME}(\boldsymbol{x}_c) \tag{6}$$

Common classification problems are associated with the last timing when their results are reused, and the least-recently-reused common classification problem is removed when $|\mathcal{X}_c| = k$. To realize this policy, a circular linked-list of size k is used to maintain precomputed results, and the oldest element is just overwritten while the corresponding classification problem is removed.

Fixed threshold k is used throughout the processing of a text stream, and its impact on classification speed was evaluated by experiments.

Since Algorithm 1 naively constructs common classification problems using all the active primitive features in input classification problem $\boldsymbol{x}$, it might repeatedly add and remove classification problems that include rare primitive features such as lexical features. This will incur serious overhead costs. To avoid this situation, the margin computation is terminated as soon as it is determined that the remaining computation does not change the sign of margin (namely, classification label y) of $\boldsymbol{x}$.

When $\boldsymbol{x}$ and $\boldsymbol{x}_c$ are given, lower- and upper-bounds of $m(\boldsymbol{x}; \boldsymbol{\phi}, \boldsymbol{w})$ can be computed by accumulating bounds of a partial margin computed by adding remaining active primitive features, $\{\phi'_j \in \boldsymbol{x} \mid \phi'_j \notin \boldsymbol{x}_c\}$, one by one to $\boldsymbol{x}_c$. It is assumed that primitive feature ϕ'_i is newly activated in $\boldsymbol{x}_c$ and $\boldsymbol{x}_c^{old}$ refers to $\boldsymbol{x}_c$ without ϕ'_i being activated. The partial margin, $m(\boldsymbol{x}_c; \boldsymbol{\phi}, \boldsymbol{w}) - m(\boldsymbol{x}_c^{old}; \boldsymbol{\phi}, \boldsymbol{w})$, is computed by summing up the weights of primitive feature ϕ'_i and conjunctive features that are composed of ϕ'_i and one or more primitive features $\phi'_j \in \boldsymbol{x}_c^{old}$. This partial margin is upper- and lower-bounded as follows:

$$m(\boldsymbol{x}_c; \boldsymbol{\phi}, \boldsymbol{w}) - m(\boldsymbol{x}_c^{old}; \boldsymbol{\phi}, \boldsymbol{w}) \geq \max(w_i^{min} |\{\phi_j \in \boldsymbol{x}_c \mid \phi_j \notin \boldsymbol{x}_c^{old}\}|, W_i^-) \tag{7}$$

$$m(\boldsymbol{x}_c; \boldsymbol{\phi}, \boldsymbol{w}) - m(\boldsymbol{x}_c^{old}; \boldsymbol{\phi}, \boldsymbol{w}) \leq \min(w_i^{max} |\{\phi_j \in \boldsymbol{x}_c \mid \phi_j \notin \boldsymbol{x}_c^{old}\}|, W_i^+), \tag{8}$$

where w_i^{min} and w_i^{max} refer to minimum and maximum weights among all the features regarding ϕ'_i, while W_i^+ and W_i^- refer to summations of all the features regarding ϕ'_i with positive and negative weights, respectively; that is, this upper or lower-bound is computed by assuming all the features regarding ϕ'_i to have a maximum or minimum weight (each bounded by W_i^+ or W_i^-). Accumulating these bounds for each remaining primitive feature makes it possible to obtain the bounds of $m(\boldsymbol{x}; \boldsymbol{\phi}, \boldsymbol{w})$ and thereby judge whether the sign of the margin can be changed by processing the remaining features.

4.2 Maintaining common classification problems with dynamic double-array trie

To maintain the enumerated common classification problems, a double-array trie (Aoe, 1989) is applied. The trie associates common classification problem $\boldsymbol{x}_c$ with unique index $i (1 \leq i \leq k)$, which is further associated with computed margin $M_{\boldsymbol{x}_c}$ and frequency or access time as described in Section 4.1. Although a double-array trie provides an extremely fast look-up, it had been considered that update operation (adding a new key to a double-array trie) is slow. However, in a recent study (Yata et al., 2009), the update speed of a double-array trie approaches that of a hash table. In the following section, a double-array trie similar to that of Yata et al. (2009) is used to maintain common classification problems.

Efficient dynamic double-array trie with deletion

A double-array trie (Aoe, 1989) and an algorithm that allows a fast update (Yata et al., 2009) are briefly introduced in the following. A double array is a data structure for a compact trie, which consists of two one-dimensional arrays called BASE and CHECK. In a double-array trie, each trie node occupies one element in BASE and CHECK, respectively.[2] For each node, p, BASE stores the offset address of its child

[2]Although the original double-array (Aoe, 1989) realizes a minimal-prefix trie by using another array (called TAIL) to store suffix nodes with only one child, TAIL is not adopted here since it is difficult to support space-efficient deletion with TAIL.

nodes, so a child node takes the address $c = \text{BASE}[p]\ \text{XOR}^3\ l$ when the node is traversed from p by label l. For each node, c, CHECK stores the address of its parent node, p, and is used to confirm the validity of the traversal by checking whether $\text{CHECK}[c] = p$ is held after the node is reached by $c = \text{BASE}[p]\ \text{XOR}\ l$.

Adding a new node to a trie could cause a conflict, meaning that the newly added node could be assigned to the address taken by an existing node in the trie. In such a case, it is necessary to collect all the sibling nodes of either the newly added node or the existing node that took the conflicting address, and then relocate either branching (with a lower number of child nodes) to empty addresses that are not taken by other nodes in the trie. This relocation is time-consuming, and is the reason for the slow update.

To perform this relocation quickly, Yata et al. (2009) introduced two additional one-dimensional arrays, called NLINK (node link) and BLOCK. For each node, NLINK stores the label needed to reach its first child and the label needed to reach from its parent the sibling node next to the node. It thereby makes it possible to quickly enumerate the sibling nodes for relocation. BLOCK stores information on empty addresses within each 256 consecutive addresses called a block[4] in BASE and CHECK. Each block is classified into three types, called "full," "closed" and "open." Full blocks have no empty addresses and are excluded from the target of relocation. Closed blocks have only one empty address or have failed to be relocated more times than a pre-specified threshold. Open blocks are other blocks, which have more than one empty address. The efficient update of the trie is enabled by choosing appropriate blocks to relocate a branching; a branching with one child node is relocated to a closed block, while a branching with multiple child nodes is relocated to an open block.

The above-described double-array trie was modified to support a deletion operation, which simply registers to each block empty addresses resulting from deletion. In consideration that a new key (common classification problem) will be stored immediately after the deletion (Line 10 in Algorithm 1), the double-array trie is not packed as in Yata et al. (2007) after a key is deleted.

Engineering a double-array trie to reduce trie size

To effectively maintain common classification problems in a trie, it is critical to reduce the number of trie nodes accessed in look-up, update, and deletion operations. The number of trie nodes was therefore reduced as much as possible by adopting a more compact representation of keys (common classification problems) and by elaborating the way to store values for the keys in the double-array trie.

Gap-based key representation To compress representations of common classification problems (feature sequences) in the trie, frequency-based indices are allocated to primitive features (Yoshinaga and Kitsuregawa, 2010). A gap representation (used to compress posting lists in information retrieval (Manning et al., 2008, Chapter 5)) is used to encode feature sequences. Each feature index is replaced with a gap from the preceding feature index (the first feature index is used as is). Each gap is then encoded by variable-byte coding (Williams and Zobel, 1999) to obtain shorter representations of feature sequences.

A reduced double-array trie The standard implementation of a double-array trie stores an (integer) index with a key at a child node (value node) traversed by a terminal symbol `'\0'` (or an alphabet not included in a key, *e.g.*, `'#'`) from the node reached after reading the entire key (Yoshinaga and Kitsuregawa, 2009; Yasuhara et al., 2013). However, when a key is not a prefix to the other keys, the value node has no sibling node, so a value can be directly embedded on the BASE of the node reached after reading the entire key instead of the offset address of the child (value) node. All the value nodes for the longest prefixes are thereby eliminated from the trie. The resulting double-array trie is referred to as a *reduced double-array trie*.

These two tricks reduce the number of trie nodes (memory usage), and make the trie operations faster. A reduced double-array trie is also used to compactly store the weights of conjunctive features, as described in Yoshinaga and Kitsuregawa (2010). Interested readers may refer to **cedar**,[5] open-source software of a dynamic double-array trie, for further implementation details of the reduced double-array trie.

[3]The original double-array (Aoe, 1989) uses addition instead of XOR operation to obtain a child address.

[4]Note that the XOR operation guarantees that all the child nodes are located within a certain block i (assuming 1 byte (0-255) for each label, l, child nodes of a node, p, are all located in addresses ($256i \leq c = \text{BASE}[p]\ \text{XOR}\ l < 256(i+1)$).

[5]`http://www.tkl.iis.u-tokyo.ac.jp/~ynaga/cedar/`

	base-phrase chunking	dependency parsing
Number of features $\|\phi\|$	645,951	2,084,127
Number of primitive features $\|\phi'\|$	11,509	27,063
Accuracy (partial)	99.01%	92.23%
Accuracy (complete)	94.16%	58.38%

Table 1: Model statistics for base-phrase chunking and dependency parsing.

5 Experiments

The proposed self-adaptive classifier was experimentally evaluated by applying it to streams of classification problems. The streams of classification problems were generated by processing Twitter streams using a state-of-the-art base-phrase chunker and dependency parser. All experiments were conducted with an Intel® Core™ i7-3720QM 2.6-GHz CPU server with 16-GB main memory.

5.1 Setup

Since March 11, 2011 (the day of the Great East Japan Earthquake; "3.11 earthquake" hereafter), Twitter streams were crawled by using Twitter API.[6] Tweets from famous Japanese users were crawled first. Next, timelines of those users were obtained. Then, the set of users were repeatedly expanded by tracing retweets and mentions in their timelines to collect as many tweets as possible. In the following experiments, two sets of 24-hour Twitter streams from the crawled tweets were used. The first Twitter stream was taken from the day of 3.11 earthquake (12:00 on Friday, March 11, 2011 to 12:00 on Saturday, March 12, 2011), and the second one was taken from the second weekend in March, 2012 (12:00 on Friday, March 9, 2012 to 12:00 on Saturday, March 10, 2012). The first Twitter stream is intended to evaluate the classifier performance on days with a significant, continuous burst, while the second one is to evaluate the performance on days without such a burst. No special events, other than a small earthquake (02:25 on March 10), occurred from March 9 to 10, 2012. Because the input to base-phrase chunking and dependency parsing is a sentence, each post was split by using punctuations as clues.

Although it might be better to evaluate the chunking and parsing speed with the proposed classifier for a text stream, the classification speed was evaluated for streams of classification problems generated in processing the Twitter streams by a deterministic base-phrase chunker (Sassano, 2008) and a shift-reduce dependency parser (Sassano, 2004), which are implemented in J.DepP.[12] Note that the chunker and parser are known to spend most of the time for classification (Yoshinaga and Kitsuregawa, 2012), and reducing the classification time leads to efficient processing of Twitter streams.

The base-phrase chunker processes each token in a sentence identified by a morphological analyzer, MeCab,[7] and judges whether the token is the beginning of a base-phrase chunk in Japanese (called a *bunsetsu*[8]) or not. The shift-reduce dependency parser processes each chunk in the chunked sentences and determines whether the head candidate chosen by the parser is correct head or not.

The classifiers for base-phrase chunking and dependency parsing were trained by using a variant of a passive-aggressive algorithm (PA-I) (Crammer et al., 2006) with a standard split[9] of the Kyoto-University Text Corpus (Kawahara et al., 2002) Version 4.0.[10] A third-order polynomial kernel was used to consider combinations of up-to three primitive features. The features used for training the classifiers were identical to those implemented in J.DepP. The polynomial kernel expanded (Kudo and Matsumoto, 2003) was used to make the number of resulting conjunctive features tractable without harming the accuracy.

Table 1 lists the statistics of the models trained for chunking and parsing. In Table 1, "accuracy (partial)" is the ratio of chunks (or dependency arcs) correctly identified by the chunker (or the parser), while "accuracy (complete)" is the exact-match accuracy of complete chunks (or dependency arcs) in a sentence. The accuracy of the resulting parser on the standard split was better than any published results

[6]`https://dev.twitter.com/docs/api`

[7]`http:://mecab.sourceforge.net/`

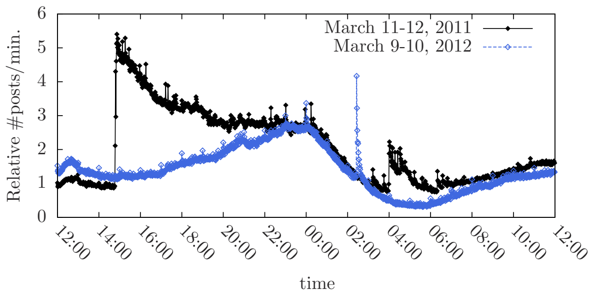
[8]A *bunsetsu* is a linguistic unit consisting of one or more content words followed by zero or more function words.

[9]24,263, 4,833 and 9,284 sentences (234,685, 47,571 and 89,874 base phrases) for training, development, and testing.

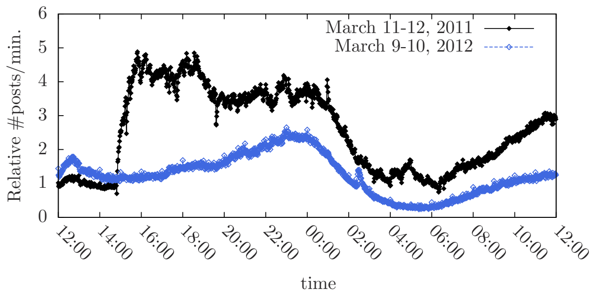
[10]`http://nlp.ist.i.kyoto-u.ac.jp/EN/index.php?Kyoto%20University%20Text%20Corpus`

	March 11-12, 2011	March 9-10, 2012
Number of posts	9,291,767	6,360,392
Number of posts/s	108	74
Number of sentences	24,722,596	13,521,196
Number of classification problems (chunking)	220,490,401	109,452,133
Number of classification problems (parsing)	70,096,105	34,380,385

Table 2: Twitter stream used for evaluation.



(a) Change in number of posts per min.



(b) Change in number of classifications per min. (parsing)

Figure 1: Volume of flow of Twitter streams from March 11 to 12, 2011 and from March 9 to 10, 2012.

method	March 11-12, 2011 space [MiB]	speed [ms/sent.]	ratio	March 9-10, 2012 space [MiB]	speed [ms/sent.]	ratio
baseline	12.01	0.0221	1.00	12.01	0.0188	1.00
Y&K '09	30.46	0.0118	1.87	30.46	0.0112	1.69
proposed $k = 2^{16}$	18.05	0.0092	2.40	17.93	0.0098	1.93
(LFU) $k = 2^{20}$	90.70	0.0088	2.51	90.78	0.0089	2.12
$k = 2^{24}$	463.04	0.0081	2.73	473.60	0.0076	2.48
proposed $k = 2^{16}$	17.32	0.0086	2.57	17.32	0.0093	2.02
(LRU) $k = 2^{20}$	85.89	0.0077	2.88	86.09	0.0085	2.22
$k = 2^{24}$	399.17	**0.0070**	**3.16**	409.59	**0.0068**	**2.76**

Table 3: Experimental results obtained with the reduced double array trie: base phrase chunking.

for this dataset other than those reported for a parser based on “stacking” (Iwatate, 2012).[11]

Table 2 lists the detail of the Twitter streams used for evaluating the proposed classifier. Figures 1(a) and 1(b) show the change in the number of posts and classifications for parsing per minute, when the average number of posts and classifications per minute before the 3.11 earthquake is counted as one, respectively. The dataset shows a rapid growth in the number of posts after the 3.11 earthquake occurred (14:46:18). This event also incurs a rapid growth in the number of classifications for parsing. Although space limitations precluded the number of classifications for chunking, it had the same tendency as for parsing. It should be noted that the official retweets (reposts) occupied 25.8% (2,394,025) and 8.5% (542,726) of the entire posts from March 11 to 12, 2011 and from March 9 to 10, 2012, respectively.

5.2 Results

Tables 3 and 4 list the timings needed to solve the classification problems generated for each sentence by processing the Twitter streams listed in Table 2 using the base-phrase chunker and the dependency parser, respectively. In Table 3, “baseline” refers to the classifier using Eq. 3, while “Y&K ’09” refers to Yoshinaga and Kitsuregawa (2009) who used Eq. 4, and enumerates common classification problems from the training corpus[9] of the classifier in advance. To highlight the performance gap caused by the algorithmic differences and make the memory consumptions comparable, the experiments were conducted using the same implementation of the reduced double-array trie, described in Section 4.2, for all the methods. The proposed classifier (with 65,536 ($k = 2^{16}$) common classification problems) achieved higher classifica-

[11]The best reported accuracy of a non-stacking parser is 91.96% (partial) and 57.44% (complete) for Kyoto-University Text Corpus Version 3.0 (Iwatate et al., 2008), and is better than that achieved by the MST algorithm (McDonald et al., 2005).

method	March 11-12, 2011 space [MiB]	speed [ms/sent.]	ratio	March 9-10, 2012 space [MiB]	speed [ms/sent.]	ratio
baseline	31.50	0.1187	1.00	31.50	0.0979	1.00
Y&K '09	99.91	0.0738	1.61	99.91	0.0651	1.51
proposed $k = 2^{16}$	43.21	0.0469	2.53	43.01	0.0542	1.81
(LFU) $k = 2^{20}$	113.40	0.0293	4.06	113.27	0.0399	2.45
$k = 2^{24}$	904.32	0.0222	5.35	905.62	0.0285	3.44
proposed $k = 2^{16}$	42.68	0.0497	2.39	42.66	0.0546	1.79
(LRU) $k = 2^{20}$	108.88	0.0283	4.20	108.94	0.0421	2.32
$k = 2^{24}$	840.85	**0.0208**	**5.71**	840.93	**0.0280**	**3.50**

Table 4: Experimental results obtained with the reduced double array trie: dependency parsing.

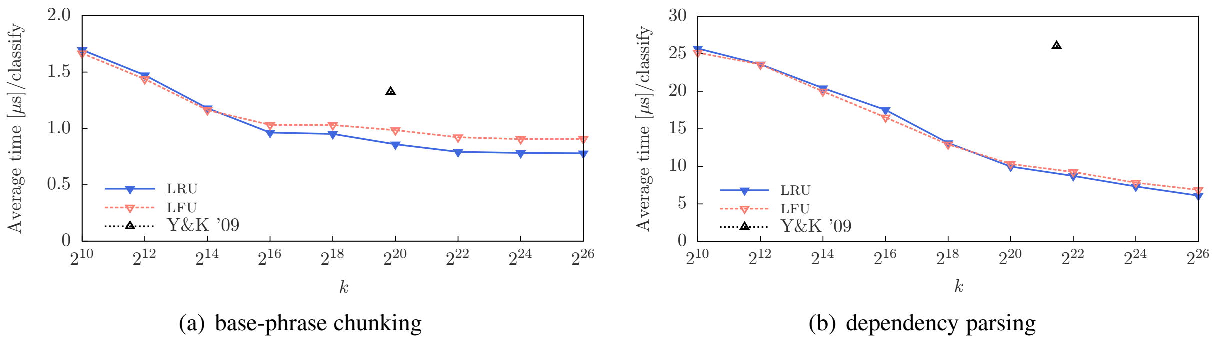


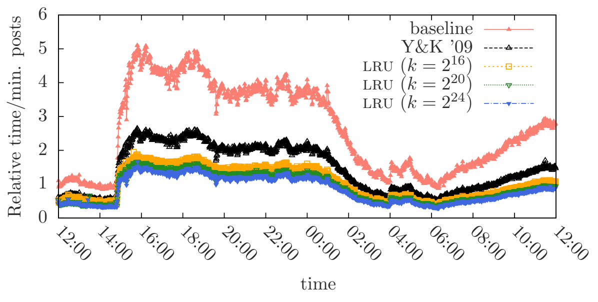
Figure 2: Average classification time per classification problem as a function of number of common classification problems k (2011 tweet stream).

tion speed than that achieved by Y&K '09 (with 943,864 (chunking) and 2,902,679 (parsing) common classification problems). Although the speed up is evident for both tweet datasets, the speed-up is more obvious in the case of the 2011 tweet stream. In the following experiments, in view of space limitations and redundancy, the 2011 tweet stream was used; however, note that the same conclusions are drawn from the results with the 2012 twitter stream.

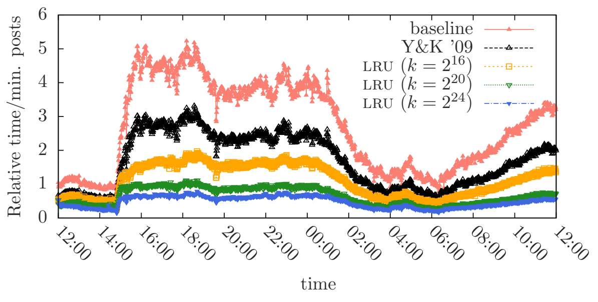
Figure 2 shows the time needed for solving each classification problem for chunking and parsing of the 2011 tweet stream when the threshold to the number of common classification problems k is varied from 2^{10} to 2^{26}, respectively. In both tasks, the proposed classifier with the LRU policy outperforms the proposed classifier with the LFU policy when k was increased. This is not only because the LFU policy has higher overheads than the LRU policy but also because the LFU policy selects useless classification problems that include lexical features related to a burst in the past. The speed-up is saturated in the case of base-phrase chunking at $k = 2^{22}$ (Figure 2(a)). This is because the proposed classifier often terminates margin computation without seeing lexical features for base phrase chunking, so it rarely reuses results of common classification problems including lexical features that are preserved when k is increased. On the other hand in dependency parsing, the classifier relies on lexical features to resolve semantic ambiguities, so it cannot terminate margin computation without seeing lexical features and thus exploits common classification problems including lexical features.

Figure 3 shows the change in the time needed for solving classification problems generated from a one-minute text stream for chunking and parsing of the 2011 tweet stream. The y-axis shows the relative classification time, when the average classification time of the baseline method before the 3.11 earthquake is counted as one. The classification time of the baseline method and Yoshinaga and Kitsuregawa (2009)'s method rapidly increased in response to the increase of the number of classification problems, while the proposed classifier suppressed the increase in classification time. It is thus concluded that the proposed classifier is more robust in terms of real-time processing for a text stream.

Finally, the contributions of the three tricks of the proposed classifier to the classification performance for dependency parsing were evaluated. The three tricks are a gap-based key representation and a reduced double-array trie (Section 4.1), as well as the early termination of margin computation (Section 4.1).



(a) base-phrase chunking



(b) dependency parsing

Figure 3: Change in classification time of one-minute posts (2011 tweet stream).

method		plain (no tricks)			+ gap-based key			+ reduced double array			+ early termination		
		space [MiB]	speed [ms/sent.]	ratio	space [MiB]	speed [ms/sent.]	ratio	space [MiB]	speed [ms/sent.]	ratio	space [MiB]	speed [ms/sent.]	ratio
baseline		39.88	0.1413	0.84	n/a	n/a	n/a	31.50	0.1187	1.00	38.21	0.0745	1.59
Y&K '09		117.66	0.0922	1.29	110.52	0.0904	1.31	99.91	0.0738	1.61	106.61	0.0406	2.93
proposed	$k = 2^{16}$	44.64	0.1037	1.14	44.50	0.1009	1.18	36.09	0.0845	1.41	42.68	0.0497	2.39
(LRU)	$k = 2^{20}$	117.21	0.0590	2.01	114.57	0.0572	2.07	105.21	0.0492	2.41	108.88	0.0283	4.20
	$k = 2^{24}$	969.48	0.0412	2.88	923.96	0.0398	2.98	897.70	0.0350	3.39	840.85	**0.0208**	**5.71**

Table 5: Contribution of each trick to classification performance; 2011 tweet dataset (underlined numbers are quoted from Table 4).

Table 5 lists the classification times per sentence in the case of dependency parsing, when each trick is cumulatively applied to plain classifiers without all the tricks. Classification is significantly speeded up by early termination of the margin computation and the reduced double-array trie. These tricks also contribute to speeding up the baseline method and Yoshinaga and Kitsuregawa (2009)'s method.

6 Conclusion

Aiming to efficiently process a real-world text stream (such as a Twitter stream) in real-time, a self-adaptive classifier that becomes faster for a given text stream is proposed. It enumerates common classification problems that are generated during the processing of a text stream, and reuses the results of those classification problems as partial results for solving forthcoming classification problems.

The proposed classifier was evaluated by applying it to the streams of classification problems generated by processing two sets of Twitter streams on the day of the 2011 Great East Japan Earthquake and the second weekend in March 2012 using a state-of-the-art base-phrase chunker and dependency parser. The proposed classifier speeds up the classification by factors of 3.2 (chunking) and 5.7 (parsing), which are significant factors in regard to processing a massive text stream.

It is planned to evaluate the classifier on other NLP tasks. A linear classifier with conjunctive features is widely used for NLP tasks such as word segmentation, part-of-speech tagging (Neubig et al., 2011b), and dependency parsing (Nivre and McDonald, 2008). Even for NLP tasks in which structured classification is effective (*e.g.*, named entity recognition), structure compilation (Liang et al., 2008) (or "uptraining" (Petrov et al., 2010)) gives state-of-the-art accuracy when a linear classifier with many conjunctive features is used. The proposed classifier is expected to be applied to a range of NLP tasks.

All the codes have been available for the research community as open-source software, including **pecco** (a self-adaptive classifier)[12] and **J.DepP** (a base-phrase chunker and dependency parser).[13]

Acknowledgments

This work was supported by the Research and Development on Real World Big Data Integration and Analysis program of the Ministry of Education, Culture, Sports, Science, and Technology, JAPAN.

[12]`http://www.tkl.iis.u-tokyo.ac.jp/~ynaga/pecco/`
[13]`http://www.tkl.iis.u-tokyo.ac.jp/~ynaga/jdepp/`

References

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

Eiji Aramaki, Sachiko Maskawa, and Mizuki Morita. 2011. Twitter catches the flu: Detecting influenza epidemics using Twitter. In *Proceedings of EMNLP*, pages 1568–1576.

Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. 2006. Online passive-aggressive algorithms. *JMLR*, 7:551–585, March.

Jennifer Foster, Özlem Çetinoglu, Joachim Wagner, Joseph Le Roux, Stephen Hogan, Joakim Nivre, Deirdre Hogan, and Josef van Genabith. 2011. #hardtoparse: POS tagging and parsing the Twitterverse. In *Proceedings of the AAAI-11 Workshop on Analyzing Microtext*.

Eric Gilbert and Karrie Karahalios. 2010. Widespread worry and the stock market. In *Proceedings of ICWSM*, pages 58–65.

Kevin Gimpel, Nathan Schneider, Brendan O'Connor, Dipanjan Das, Daniel Mills, Jacob Eisenstein, Michael Heilman, Dani Yogatama, Jeffrey Flanigan, and Noah A. Smith. 2011. Part-of-speech tagging for Twitter: Annotation, features, and experiments. In *Proceedings of ACL-HLT*, pages 42–47.

Yoav Goldberg, Kai Zhao, and Liang Huang. 2013. Efficient implementation of beam-search incremental parsers. In *Proceedings of ACL, Short Papers*, pages 628–633.

Bo Han and Timothy Baldwin. 2011. Lexical normalisation of short text messages: Makn sens a #twitter. In *Proceedings of ACL-HLT*, pages 368–378.

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

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

Masakazu Iwatate. 2012. *Development of Pairwise Comparison-based Japanese Dependency Parsers and Application to Corpus Annotation*. Ph.D. thesis, Graduate School of Information Science, Nara Institute of Science and Technology.

Nobuhiro Kaji, Yasuhiro Fujiwara, Naoki Yoshinaga, and Masaru Kitsuregawa. 2010. Efficient staggered decoding for sequence labeling. In *Proceedings of ACL*, pages 485–494.

Daisuke Kawahara, Sadao Kurohashi, and Kôiti Hasida. 2002. Construction of a Japanese relevance-tagged corpus. In *Proceedings of LREC*, pages 2008–2013.

Terry Koo, Alexander M. Rush, Michael Collins, Tommi Jaakkola, and David Sontag. 2010. Dual decomposition for parsing with non-projective head automata. In *Proceedings of EMNLP*, pages 1288–1298.

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

Gourab Kundu, Vivek Srikumar, and Dan Roth. 2013. Margin-based decomposed amortized inference. In *Proceedings of EMNLP*, pages 905–913.

Percy Liang, Hal Daumé III, and Dan Klein. 2008. Structure compilation: trading structure for features. In *Proceedings of ICML*, pages 592–599.

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

Twitter, Inc. 2011. Twitter's 2011 year in review. `http://yearinreview.twitter.com/en/tps.html`.

Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005. Online large-margin training of dependency parsers. In *Proceedings of ACL*, pages 523–530.

Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of ICDT*, pages 398–412.

Graham Neubig, Yuichiroh Matsubayashi, Masato Hagiwara, and Koji Murakami. 2011a. Safety information mining - what can NLP do in a disaster -. In *Proceedings of IJCNLP*, pages 965–973.

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

Joakim Nivre and Ryan McDonald. 2008. Integrating graph-based and transition-based dependency parsers. In *Proceedings of ACL-HLT*, pages 950–958.

Slav Petrov, Pi-Chuan Chang, Michael Ringgaard, and Hiyan Alshawi. 2010. Uptraining for accurate deterministic question parsing. In *Proceedings of EMNLP*, pages 705–713.

Alan Ritter, Sam Clark, Mausam, and Oren Etzioni. 2011. Named entity recognition in tweets: An experimental study. In *Proceedings of EMNLP*, pages 1524–1534.

Alexander Rush and Slav Petrov. 2012. Vine pruning for efficient multi-pass dependency parsing. In *Proceedings of NAACL-HLT*, pages 498–507.

Takeshi Sakaki, Makoto Okazaki, and Yutaka Matsuo. 2010. Earthquake shakes Twitter users: Real-time event detection by social sensors. In *Proceedings of WWW*, pages 851–860.

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

Manabu Sassano. 2008. An experimental comparison of the voted perceptron and support vector machines in Japanese analysis tasks. In *Proceedings of IJCNLP*, pages 829–834.

Vivek Srikumar, Gourab Kundu, and Dan Roth. 2012. On amortizing inference cost for structured prediction. In *Proceedings of EMNLP-CoNLL*, pages 1114–1124.

Andranik Tumasjan, Timm O. Sprenger, Philipp G. Sandner, and Isabell M. Welpe. 2010. Predicting elections with Twitter: What 140 characters reveal about political sentiment. In *Proceedings of ICWSM*, pages 178–185.

Gertjan van Noord. 2009. Learning efficient parsing. In *Proceeding of EACL*, pages 817–825.

István Varga, Motoki Sano, Kentaro Torisawa, Chikara Hashimoto, Kiyonori Ohtake, Takao Kawai, Jong-Hoon Oh, and Stijn De Saeger. 2013. Aid is out there: Looking for help from tweets during a large scale disaster. In *Proceedings of ACL*, pages 1619–1629.

Henning Wachsmuth, Benno Stein, and Gregor Engels. 2011. Constructing efficient information extraction pipelines. In *Proceedings of CIKM*, pages 2237–2240.

Henning Wachsmuth, Benno Stein, and Gregor Engels. 2013. Learning efficient information extraction on heterogeneous texts. In *Proceedings of IJCNLP*, pages 534–542.

Xiaofeng Wang, Matthew S. Gerber, and Donald E. Brown. 2012. Automatic crime prediction using events extracted from Twitter posts. In *Proceedings of SBP*, pages 231–238.

Hugh E. Williams and Justin Zobel. 1999. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201.

Makoto Yasuhara, Toru Tanaka, Jun ya Norimatsu, and Mikio Yamamoto. 2013. An efficient language model using double-array structures. In *Proceedings of EMNLP*, pages 222–232.

Susumu Yata, Masaki Oono, Kazuhiro Morita, Masao Fuketa, and Jun ichi Aoe. 2007. An efficient deletion method for a minimal prefix double array. *Journal of Software: Practice and Experience*, 37(5):523–534.

Susumu Yata, Masahiro Tamura, Kazuhiro Morita, Masao Fuketa, and Jun'ichi Aoe. 2009. Sequential insertions and performance evaluations for double-arrays. In *Proceedings of the 71st National Convention of IPSJ*, pages 1263–1264. (In Japanese).

Naoki Yoshinaga and Masaru Kitsuregawa. 2009. Polynomial to linear: Efficient classification with conjunctive features. In *Proceedings of EMNLP*, pages 1542–1551.

Naoki Yoshinaga and Masaru Kitsuregawa. 2010. Kernel slicing: Scalable online training with conjunctive features. In *Proceedings of COLING*, pages 1245–1253.

Naoki Yoshinaga and Masaru Kitsuregawa. 2012. Efficient classification with conjunctive features. *Journal of Information Processing*, 20(1):228–227.