Text Categorization with All Substring Features

Daisuke Okanohara* Jun'ichi Tsujii†

Abstract

This paper presents a novel document classification method using all substrings as features. Although tokenized words are not enough for determining a class of a document, learning by using all substrings has a prohibitive computational cost because the number of all candidate substrings can be very large. We show that the idea of equivalent classes of substrings can help determine all effective substrings exhaustively in linear time. Moreover, by applying L_1 regularization to our model, we obtain a compact result, which makes an inference extremely efficient in time and space, and robust even if we use substrings of all lengths. In experiments, we show that our method can extract effective substrings efficiently, and achieved more accurate results and the its inference was faster than the results using previous methods.

Keywords: Document Classification, Logistic Regression, L_1 regularization, Enhanced Suffix Arrays

1 Introduction

Document classification is a fundamental task for many applications; given a document, we assign a label such as a category (*sports*, *money*), or polarity (*positive* or *negative* opinion) according to the meaning of the document. Rule-based methods were first applied in this task, but recently, machine learning methods have been applied using support vector machines (SVM) or logistic regressions (LR) because they are robust, easy to adapt to a new domain, and achieve more accurate results.

Generally, a document d is represented as a feature vector $f(d) \in R^m$ where each dimension corresponds to the occurrence of a word in a document. Since this representation ignores the order or the position of the words, this representation is called a *bag of words* (BOW).

Although a BOW representation loses much of the document information, this often achieves high performance because the occurrence of a few keywords can often determine the label of the document.

However, this BOW representation still suffers from the following three problems.

The first is the error in the conversion from a document to a set of words. For example, several languages, such as Japanese and Chinese, do not represent word boundary information explicitly. The word identification task itself is not easy, and the result includes many errors. What is worse is that the keywords for document classification are often unknown words such as a person's name, (e.g. *Shaquille O'Neal*), and BOW representation loses this information due to errors occurring in the analysis.

The second is that, in some data, it is difficult to define what the word is, such as, log data and bio-informatics data.

The third is the most important problem. Words units are often inappropriate for document classification, although N-gram words are effective. For example an occurrence of a movie title is effective for determining the label of the document to be *movie*. , many movie titles consist of several common words, which are lost in a BOW representation. The spam mail detection task is another example; signature and template information is important but this is not word information.

In this paper, we propose a logistic regression model with all substring features for a document classification. We consider all substrings as features, and can assign different weights to them. Although the number of substrings, and the features are prohibitively large to optimize, we can find the optimal classifier in liner time in the total length of documents by summarizing substring information in the equivalent classes.

This paper shows that we can find effective features exhaustively by checking the features corresponding to maximal substring only. The number of maximal substrings is not quadratic but linear in the document length, and we can therefore efficiently train the weight vector.

Moreover, since we apply L_1 regularization on weights, we obtain a very compact model; the number of non-zero weights is very few, the model is easy to interpret, and the inference is extremely efficient in time and working space. By combining Grafting algorithm [1], a weight vector can be optimized in time proportional to the number of non-zero weights

Many previous works studied to use all substrings information for a document classification task. Among them, string kernels [2] is most popular, which defines a kernel for two documents d_1, and d_2 as follows,

$$K(d_1, d_2) = \sum_{s \in \Sigma^*} r_s s(d_1) s(d_2), \tag{1.1}$$

where Σ is the alphabet set, and Σ^* is the set of all substrings

*Department of Computer Science, University of Tokyo

†Department of Computer Science, University of Tokyo, School of Informatics, University of Manchester, National Center for Text Mining

on Σ, and r_s is a weight parameter for s (which is not decided by learning), and $s(d)$ is the frequency of a substring s in a document d.

By incorporating this string kernel into SVM learning, we can classify a document according to all the substring information in the document. Teo [3] shows that by using suffix arrays and auxiliary data structures, we can calculate a kernel value in in $O(|d_1| + |d_2|)$ time, and an inference for a test document d can be done in $O(|d|)$ time where $|d|$ is the length of a document.

However, string kernels require a large amount of working space not only at training time, but also at inference time. As an example, it requires 19 times for an original training examples. Therefore, such a method cannot be applied for a large document set.

Moreover, kernel methods cannot control each weight independently, and they can only control a weight for each training examples. In general, very few features contribute to the label decision, and a string kernel cannot capture these features efficiently.

Also, string kernels tend to be affected by noise, so that we may need to cut off a long substring. Therefore, it is very difficult to consider all substrings in a string kernel.

Very recently, Ifrim et. al [4] proposed a logistic regression model with variable-length N-gram features (structured logistic regression: SLR). In their model, different weights can be assigned to each features. They showed that N-gram information is important for document classification, and more accurate than BOW representation. However, because effective N-grams are searched greedily, important N-gram phrases can be lost. Another problem is that Ifrim's method suffers from over-fitting due to the lack of regularization, and difficult to decide when the search process stops at training.

In experiments, we compared our method with previous methods and showed that our method achieved the highest performance in terms accuracy and speed.

2 Preliminaries

2.1 L_1 Regularized Logistic Regression Model In this paper, we consider a multi-class logistic regression model (LR). For an input vector $\boldsymbol{x} \in R^{m'}$, and an output label $y \in \mathscr{Y}$, and $k = |\mathscr{Y}|$, we define a feature vector $\phi(x, y) \in R^m$. Then, the probability for a label y given an input $\boldsymbol{x}$ is defined as follows,

$$p(y|x; \boldsymbol{w}) = \frac{1}{Z(x)} \exp(\boldsymbol{w}^T \phi(x, y)) \tag{2.2}$$

$$Z(x) = \sum_{y'} \exp(\boldsymbol{w}^T \phi(x, y')),$$

where $\boldsymbol{w} \in R^m$ is the weight vector. The most probable label is the one that maximizes the margin,

$$y^* = \arg\max_y p(y|x; \boldsymbol{w}) = \arg\max_y \boldsymbol{w}^T \phi(x, y). \tag{2.3}$$

The parameter $\boldsymbol{w}$ is estimated by maximum likelihood estimation (MLE) using training examples $\{(x_i, y_i)\}$ $(i = 1, \ldots, n)$,

$$\boldsymbol{w}^* = \arg\max_{\boldsymbol{w}} L(\boldsymbol{w}) \tag{2.4}$$

$$L(\boldsymbol{w}) = \sum_i \log p(y_i|x_i; \boldsymbol{w}) \tag{2.5}$$

However, this MLE tends to over-fit the training data when the amount of training examples is insufficient for the number of parameters.

To avoid this problem, a regularization term $r(\boldsymbol{w}) : R^m \mapsto R$ is added to the likelihood term shown in (2.5). By applying L_1 regularization (which is also called Lasso regularization), the weight vector is estimated as follows:

$$\boldsymbol{w}^*_{MAP} = \arg\max_{\boldsymbol{w}} L(\boldsymbol{w}) - C|\boldsymbol{w}|_1, \tag{2.6}$$

where $|\boldsymbol{w}|_1 = |w_1| + |w_2| + \ldots + |w_m|$, and $C > 0$ is the trade-off parameter between the likelihood term and the regularization term; a small C emphasizes the training data, and a large C emphasizes the regularization. This L_1 regularization corresponds to the maximum a posteriori (MAP) estimation with the Laplace prior on $\boldsymbol{w}$. We call this estimation L_1-LR.

It is known that the result of L_1-LR is a sparse parameter vector, in which many of the parameters are exactly zero. In other words, learning with L_1 regularization naturally has an effect on the feature selection, which results in an efficient and interpretable inference. For example, Gao et. al [5] compared L_1-LR with other learning methods including L_2 regularized LR. Even though the performances for these methods are almost identical, the number of non-zero weights is approximately $1/10$ of that of L_2.

To optimize (2.6), a gradient based optimization cannot be used directly, since the objective function is not differentiable where $w_i = 0$. Therefore several specialized methods have been proposed for the L_1-LR optimization. In this study, we will use OWL-QN [6] where the orthant of parameter is fixed at the time of updating, which was recently generalized in [7].

2.2 Grafting To maximize the training efficiency by employing the characteristics of L_1 regularization, we use *grafting* [1]. Algorithm 1 shows the pseudo code for this algorithm.

In the algorithm, we keep the current weight vector $\boldsymbol{w}$ and active features H (features that have non-zero weights). At the beginning, we initialize the parameters as $\boldsymbol{w} = \boldsymbol{0}$, and $H = \{\}$.

Next, we let $\boldsymbol{v}$ be the gradient of the likelihood term with

Algorithm 1 The training of L_1-LR using grafting

Input: Training data (x_i, y_i) $(i = 1, \ldots, n)$, Parameter C
$H = \{\}, \boldsymbol{w} = \mathbf{0}$
loop
  $\boldsymbol{v} = \frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}}$
  $k^* = \arg\max_k |v_k|$
  ($\boldsymbol{v}$ is the gradient of log likelihood term)
  if $|\boldsymbol{v}_{k^*}| < C$ **then**
    break
  end if
  $H = H \cup k^*$
  Optimize $\boldsymbol{w}$ with regards to H
end loop
Output $\boldsymbol{w}$ and H

regard to parameters $\boldsymbol{w}$;

$$\boldsymbol{v} = \frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}} \tag{2.7}$$

$$= \sum_{i,y} \left(I(y = y_i) - p(y|x_i; \boldsymbol{w})\right) \phi(x_i, y), \tag{2.8}$$

where $I(a)$ is 1 if a is true and 0 otherwise. Let k^* be the feature such that $|v_{k^*}|$ is the largest. Then we add k^* to H, and optimize $\boldsymbol{w}$ with H only by using a solver for L_1-LR such as OWL-QN [6], in that w_k such that $k \notin$ are fixed to be 0).

We continue this process repeatedly until $|v_{k^*}| < C$. Then the obtained weight vector is identical to the optimal weight vector $\boldsymbol{w}_{MAP}$ [1].

Consequently, the training time is almost proportional to the number of active features if we can efficiently compute $\arg\max_k |v_k|$, even if the number of features is very large

2.3 Data Structures In this section, we explain several data structures, which are used to calculate substring information efficiently.

Let $T[1, s]$ be an input text drawn from an alphabet set Σ of length s. We assume that T is terminated by a special character \$ ($T[s] = \$$), which is lexicographically smaller than all other characters, and appears nowhere else in T.

Let $S_i = T[i, s]$[1] $(i = 1, \ldots, s)$ be a suffix of T. A suffix tree is a powerful tool for many string processing; a trie data structure consisting of all suffixes of T, and a node with only one child is removed [8]. The number of internal nodes in a suffix tree for text T is less than $s - 1$. However, the working space is very large even if we manipulate the implementation of the suffix trees (e.g. $20s$ bytes).

Enhanced suffix arrays (ESA) [9] support many string operations as efficiently as suffix trees. ESA consists of suffix array, and the height array, and the working space of both is $9s$ bytes in total. See [9] for more detail.

A suffix array [10, 11] of T is a permutation of all suffixes of input text T so that the suffixes are lexicographically sorted. Formally, the suffix array of T is an array $SA[1 \ldots s]$ containing a permutation of the interval $[1 \ldots s]$, such that $T_{SA[i]} < T_{SA[i+1]}$, for all $1 \leq i < s$, where "<" between strings is the lexicographical order. Figure 1 shows an example of the suffix array for text $T = abracadabra\$$.

The suffix array can be built in $O(s)$ time for an input text of length s using $O(s \log s)$ bits of working space [12].

The height array $H[1, s]$ for T is defined as $H[i] = lcp(T_{SA[i]}, T_{SA[i+1]})$, where $lcp(T_{SA[i]}, T_{SA[i+1]})$ is the length of the longest common prefix between $T_{SA[i]}$ and $T_{SA[i+1]}$. That is, H contains the lengths of the longest common prefixes of the suffixes of T that are consecutive in lexicographic order.

T=abracadabra$

i	SA	H	B	suffix
1	12	0	a	$
2	11	1	r	a$
3	8	4	d	abra$
4	1	1	$	abracadabra$
5	4	1	r	acadabra$
6	6	0	c	adabra$
7	9	3	a	bra$
8	2	0	a	bracadabra$
9	5	0	a	cadabra$
10	7	0	a	dabra$
11	10	2	b	ra$
12	3	0	b	racadabra$

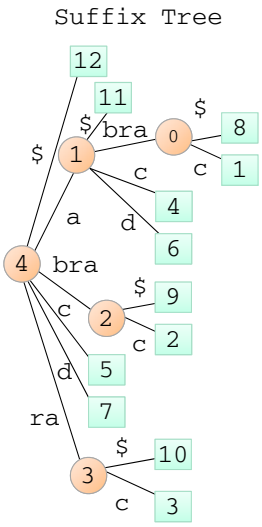

Figure 1: The column *SA*, show the suffix array for $T = abracadabra$, the column H shows the height array, the column B shows the Burrows-Wheeler's transformed text, and the column *suffix* shows the suffixes. On the right, the suffix trees for T is shown.

3 Document Classification with All Substring

3.1 Maximal Substring We propose a novel document classifier using all substrings as features. This can be considered as a bag of N-grams with $N = 1 \ldots \infty$. Although the number of features (substrings) are the quadratic of the document length, we can find the optimal solution in linear time of a length of a document by reducing equivalent substrings.

[1] We denote $T[i, j]$ as the substring of T from i-th character to j-th character

Formally, we represent a document as a bag of all substring representations where all substrings correspond to each dimension a feature vector. We call this representation *all-BOW*.

The cost of an all-BOW representation is prohibitive to directly train the L_1-LR model. However, we show that effective substrings can be found exhaustively by enumerating all maximal substring information. Note that this is not an approximation, but an exact solution.

To achieve this solution, we summarize substrings into classes. The substrings int the same class which have equivalent statistical information. The same idea was proposed in [13], which calculates term frequencies and document frequencies for all substrings efficiently. In this paper, we extend and simplify this concept to find effective substrings efficiently. The differences will be discussed.

First, let us explain the idea of equivalent classes of substrings by using suffix trees, which are not discussed in the Yamamoto's original paper [13]. Recall that a suffix tree for T store all suffixes of T in a trie data structure (Fig. 1). Let $\mathbf{occ}(T, q)$ be the number of occurrences of a substring q in T, and $P_{T,q}[1, \ldots, \mathbf{occ}(T, q)]$ be the list of all occurrence positions of q in T. We omit T if there is no confusion. For example $P_a = \{1, 4, 6, 8, 11\}$ in Figure 1. Then, P_q can be examined by traversing the suffix tree from the root to the edge along the edge characters. Note that suffix trees stores all suffix of T and any substring occurred in T correspond to some position in a suffix tree.

Let we call $t(q)$ be the position of a node q in the suffix tree. Then, all descendant leaves from $t(q)$ denotes the occurrence positions of q. For example, in figure 1, when q = ab, $t(q)$ is at the edge between the internal node 1 and the internal node 0. Therefore, P_q = $\{8, 1\}$. Similarly "abr" and "$abra$", are again at the edge between 1 and 0 and, they also occur at $\{8, 1\}$. From this observation, it is easy to show that when two substrings q_1 and q_2 are on the same edge of the suffix trees, the occurrence positions for q_1, and q_2 are the same.

Let q_1 and q_2 be in the same class if $t(q_1)$ and $t(q_2)$ are on the same edge.

The number of edges between internal nodes is at most $s - 1$, and between an internal node and an leaf is s, where s is the length of an input text. Hence, the number of different classes is at most $s - 1 + s = 2s - 1$. Since all substrings appearing in T at least once are mapped into some position in the suffix tree for T, we can factorize all substring into $2s - 1$ classes.

We can easily extend this idea into a set of documents. Given a document set (x_i, y_i) $(i = 1, \ldots, n)$, let T be the concatenated test of documents, $x_1\$_1x_2\$_2 \ldots x_n\$_n$ where $\$_i$ are special characters that do not appear in the original text. Let s be the length of T. We then build a suffix trees for T. All the issue discussed above also hold.

Let $\mathbf{tf}(q, x)$ be the term frequency, or the number of occurrence substring q in a document x, and $\mathbf{df}(q)$ bet the document frequency, or the number of documents that include x. Then we summarize the idea of equivalent classes of substrings as follows [13].

LEMMA 3.1. *If two substrings q_1 and q_2 are in the same class, then $\mathbf{tf}(q_1, x_i) = \mathbf{tf}(q_2, x_i)$ for all $1 \leq i \leq n$, and $\mathbf{df}(q_1) = \mathbf{df}(q_2)$. All substrings belong to one class, and the number of different classes is at most $2s - 1$*

Proof. Let q_1 and q_2 be in the same class. Then the occurrence positions of them in T are all same. Since documents are delimited by special characters, the occurrence of q_i $(i = 1, 2)$ does not overlap the document boundaries. Therefore, the number of occurrence in a document, is always same.

We can further summarize the substring information by considering a left expansion, which was not discussed in [13]. We again see in the example in Figure 1, that the occurrence positions of bra are $\{7, 1\}$, which seem to be different from $abra$ whose positions are $\{8, 2\}$. However, these positions are the same with the constant move ($8 = 7 + 1, 2 = 7 + 1$).

Figure 2 shows some examples for $T = abracadabra\$$. The occurrence positions of "ab", "abr", "b", "br", "bra", "ra", "$abra$" appears in the same positions with constant move, and all these substrings are substrings of "$abra$". Another class that appears more than once is only for "a". All other substrings appears once, and their longest maximal substring are suffixes, such as "$abracadabra\$$", and "$bracadabra$".

Therefore, if we find the longest substring in the class, it can enumerate all substring in the same class efficiently. We call such a substring *maximal substring*.

Formally, we define the maximal strings as follows. Given input T and substring q, let $\mathbf{occ}(T, q)$ be the number of occurrences of q in T, and $P_{T,q}[1, \ldots, \mathbf{occ}(T, q)]$ be the list of all occurrence positions of q in T. We omit T as $P_q[1, \ldots, \mathbf{occ}(q)]$ if there is no confusion. We first define P-relation ($=_P$) with substrings.

DEFINITION 3.1. *Given two substring q_1 and q_2, $q_1 <_P q_2$ if and only if (1) q_1 is a substring of q_2 (2) $\mathbf{occ}(q_1) = \mathbf{occ}(q_2)$, (3) there exists $c \in N$ such that $P_{q_1}[i] + c = P_{q_2}[i]$ for all $1 \leq i \leq \mathbf{occ}(q_1)$.*

For example, in $T = abracadabra$, $ab <_P abr$, and $bra <_P abra$, but a and ab are not in this relation.

The relation $<_P$ satisfies the transitivity: if $q_1 <_P q_2$ and $q_2 <_P q_3$, then $q_1 <_P q_3$. Therefore, by $<_p$ relation, all substrings in T are divided into several groups.

We finally, define the maximal substring,

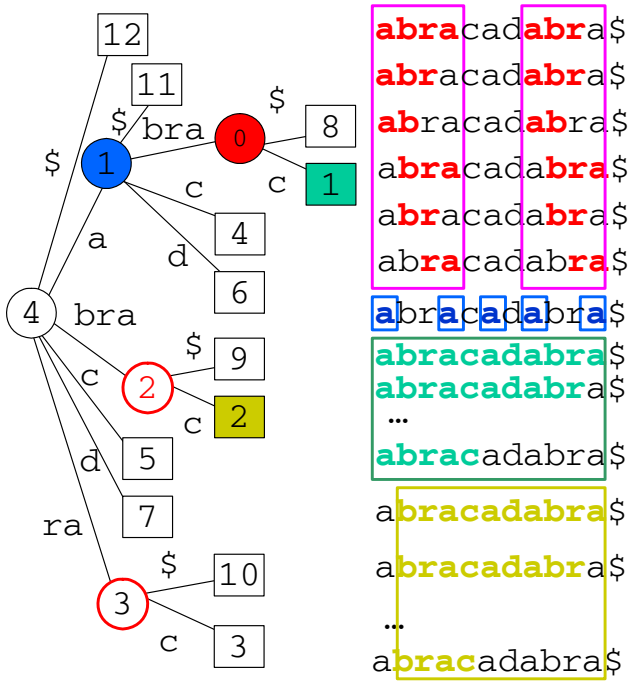

Figure 2: The substrings and its classes for a text "$T = abracadabra\$$". The suffix tree for T is shown in the left of the figure, and all substring appeared in T is shown in the right. The substrings in the same color belong to the same class, and the longest one (the most above one in the class), is the maximal substring. In this example, maximal substrings are "a", "$abra$", and all suffixes of T.

DEFINITION 3.2. *The maximal substring p is a substring such that there is no q such that $p <_P q$.*

We can easily prove that there is exactly one maximal substring in each equivalent class.

This $<_P$ relation satisfies all properties in the lemma 3.1, but the number of classes are much smaller than the original number of equivalent classes.

These maximal substring can be enumerated efficiently by using enhanced suffix arrays (ESA) (Sec. 4).

First, all maximal substrings corresponds to internal nodes or leaves in suffix trees. Especially a maximal substring that occurs more than once correspond to internal nodes only.

An internal node and leaves can be expressed as a pair of index $[l, r]$, thus indicating that the corresponding substring appears in $T[p, p + d]$ in $p \in SA[l, \ldots, r]$.

The enumeration of all leaves, and internal nodes can be done in linear time of a document length [14]. The working space for this enumeration is $10|T| + \mathrm{O}(n)$bits if a pointer is represented in 4 bytes ($|T| < 2^{32}$) .

However, not all internal nodes correspond to a maximal substring. For example, in Figure 1, although the substring ra corresponds to the internal node, it is not maximal one because $abra$ is the maximal substring.

To filter out these redundant internal nodes, we use an array $B[1, \ldots, s]$, defined as $B[i] = T[SA[i] - 1]$, except when $A[i] = 1$, then $B[i] = T[s] = \$$. This array is known as Burrows Wheeler Transform [15].

The following lemma holds,

LEMMA 3.2. *The sufficient and necessarily condition of a substring $q = [l, r, d]$ being a maximal substring is that a q corresponds to a internal node or a leaf, and $B[l, r]$ has more than one character type.*

We can check weather $B[l, r]$ has more than one character type in constant time using $n + \mathrm{o}(n)$ bits of working space (Appendix 7).

The pseudo code of this algorithm is show in the algorithm 2. Therefore, we can obtain all maximal substrings in linear time in the total length of documents.

3.2 Training with the Maximal Substring In this section, we show that optimal weights for substrings can be determined by considering the maximal substrings only.

First, we assume that the feature types is **tf** (term frequency) but this is not the only case. The feature values for substrings x and y are such that $x <_P y$ is always the same for all documents.

In the L_1 regularization, if features have equal values in all training examples, then the set of optimal weights for these features belong to the simplex distributions. Let $f_1, f_2, \ldots, f_k$ be a feature set that has the same values in all examples. We set these features belonging to the same class. Note that this class is more general than that in the substring discussed in 3.1.

LEMMA 3.3. *In the L_1 regularization, let $E = \{f_i\}$ be a set of feature indexes that belongs to the same class, and $\boldsymbol{w}^*$ be the weight vector that maximize (2.6). Then a weight vector $\boldsymbol{w}'$ such that $\sum_{i \in E} w'_i = \sum_{i \in E} w^*_i$, and $w^*_i w'_i > 0$ for all $i \in E$, and $w'_i = w^*_i$ for all $i \notin E$ also maximize (2.6).*

Proof. From the definition, we have,

$$
\begin{aligned}
|\boldsymbol{w}'|_1 &= \sum_{i \notin E} |w'_i| + \sum_{i \in E} |w'_i| \\
&= \sum_{i \notin E} |w^*_i| + \sum_{i \in E} |w^*_i|. \qquad (3.9)
\end{aligned}
$$

And, since ${\boldsymbol{w}'}^T \phi(x_i, y) = {\boldsymbol{w}^*}^T \phi(x_i, y)$ for all $1 \leq i \leq n$ and $y \in \mathcal{Y}$, for the likelihood term, $L(\boldsymbol{w}') = L(\boldsymbol{w}^*$ holds.

Therefore, when there are feature sets belong to the equivalent class, it is adequate to keep the sum of the weights for these weights. In summary, for training we deal with

the features that correspond to maximal substrings. And the obtained weight correspond to the sum of weights in the equivalent class

4 Learning L_1-LR using maximal substrings

We can generate a feature vector corresponding maximal substrings only. However, its computational cost is still large; the number of maximal substrings are linear in the total length of documents. In this section, we show that how to deal with these maximal substrings without generating feature vectors explicitly.

Recall that, the grafting algorithm (Algorithm 1) only requires finding a feature such that the absolute value of the gradient of the likelifood is the maximum ($k^* = \arg\max_k v_k$). We show that we can estimate k^* efficiently by using auxiliary data structures.

In this paper, we consider the following feature types, but our method is not limited to these feature types only.

- $\mathbf{tf}(q, d)$: the frequency of q in a document d.
- $\mathbf{bin}(q, d)$: 1 if q appears in d and 0 otherwise.
- $\mathbf{idf}(q)$: $\log(n/\mathbf{df}(q))$ where n is the number of documents, and $\mathbf{df}(q)$ is the number of documents that include q.
- $\mathbf{len}(q)$: the length of q.

In generally, we can efficiently compute the gradient value if feature functions depends on the position information. If the feature function depends on the different information, such as orthographic feature, then we cannot summarize substring information, and we require different techniques for efficient computation.

Let us consider the calculation of the gradient value $g(l, r, y)$ for the substring q that appears in $q = T[p, p + d]$ in $p \in SA[l, r]$ with label y. Remember that any substrings in T are store in the consecutive region in SA.

Let $D[i]$ be a document index that includes $SA[i]$. Let $\alpha[1, k][1, s]$ be the two dimensional array defined as,

$$\alpha[y][i] = \sum_{j=1}^{i-1} \left(I(y_{D[j]} = y) - P(y|x_{D[j]}; \boldsymbol{w}) \right). \tag{4.10}$$

Then we can calculate $g(l, r, y)$ as

$$g(l, r, y) = \alpha[y][r] - \alpha[y][l]. \tag{4.11}$$

This is because,

$$\begin{aligned} & \alpha[y][r] - \alpha[y][l] \\ = & \sum_{j=l}^{r} \left(I(y_{D[j]} = y) - P(y|x_{D[j]}; \boldsymbol{w}) \right) \\ = & \sum_{i=1}^{n} \left(I(y_{D[j]} = y) - P(y|x_{D[j]}; \boldsymbol{w}) \right) \mathbf{tf}(q, x_i). \end{aligned}$$

Therefore, the gradient for any substring can be calculated in constant time by using table lookup, where table size is $\mathrm{O}(s)$ bits.

For feature types $\mathbf{idf}(d)$ and $\mathbf{bin}(q, d)$, we can compute the gradient of any substrings in constant time using auxiliary data structures [16]. In this case, we need to remove duplicated documents at the enumarating the occurence of q in $[l, r]$. In practice, the auxiliary data structures requires much working space, so we adapt a simpler strategy; first enumerate all positions including q, and then remove duplicated documents in the positions.

When we use $\mathbf{len}(q)$ features, the gradient values for substrings in the same class is different. In this case we enumerate substring from the longest ones in the class

In summary, we state the following theorem;

THEOREM 4.1. *Given training documents whose total length is s, we can train a L_1 regularized logistic regression model using all substring features in $\mathrm{O}(s)$ time using $\mathrm{O}(s \log s)$ bits of space.*

The algorithm 2 shows the overall framework to compute the $\arg\max k$. This is same as the bottom-up traversing of all nodes in suffix tree using the height array [14] except that we compute the gradient value of each features by using $grad(l, r, d)$ as discussed above.

4.1 Extension of Substring

Finally, let us consider the case when we can use external information such as the word/phrase boundaries.

We replace an input T with an input T' such that the special characters # is inserted at the boundaries of words, and then we apply the algorithm as above. Then, we only deal with the maximal substring with # being the first characters.

This conversion does not increase the computational cost since the new input size is at most 2 times the original input size and the number of maximal strings is much smaller.

5 Inference

We explain how to classify a test document by using the result of our algorithm.

After the training, we have a set of substrings H, and their weights. We first build a trie data structure for H and we assign a weight at each leaf or internal node. Then, we find all matching for H by using the Aho-Corasick method [17]. This is done in linear time in a length of a test document .

Note that, unlike string kernels in which we have to keep all of the document set, we only keep a few substrings due to L_1 regularization. Therefore the working space is very small.

Algorithm 2 The calculation of gradients of all maximal substring

Input : $H[0,s], SA[0,s], B[0,s], D[0,s]$
S: A stack storing (pos: the beginning position in SA, len: the length of substring)
$v_k^* = 0$: Store the maximal substring that have the largest gradient value
for $i = 0$ **to** $n+1$ **do**
  $cur = (i, L[i])$
  $cand = top(S)$
  while $cand.len > cur.len$ **do**
    if $B[cand.pos, \ldots, i]$ have more than 2 characters **then**
      v_k = grad(cand.pos, i, cand.len)
      // Estimate a gradient value of feature . See section 4.
      if $v_k > v_k^*$ **then**
        $v_k^* = v_k$ // Also stores k
      end if
    end if
  end while
  if $cand.len < cur.len$ **then**
    push(S, v) // Internal node
  end if
  push(S,(i, n-SA[i] + 1)) // Leaf
end for
output v_k.

6 Experiments

We conducted a series of document classification experiments for two data sets MOVIE and Tech-TC300.

MOVIE is a sentiment classification task, given a review information we classify it into positive or negative ones. There are two types of data set, the one provided by Bo Pang [2] (MOVIE-A), and the other provided by Ifrim[3] (MOVIE-B) which was used in [4]. TechTC-300 consists of 300 binary classification task. An original category comes from Open Directory Project. Among 300 tasks, we used two tasks for which where SVM classification achieve only 70% accuracies. [4].

The details of each data set are described in Table 1.

We examined the performance using 5 cross validations. We determined the hyper parameters by using the development set.

We compared our method (Proposed in Table 2) with L_1-LR with BOW (BOW+L_1), LR with variable length N-gram [4] (SLR), and BOW with SVM (SVM). We used a polynomial kernel because it achieved highest accuracy compared other kernels (including string kernels).

For feature types, we compared the result using **bin**, **tf**, **idf**, **len** and all these combinations. For word-based BOW, **tf** achieved the best performance, and for the proposed method, **idf** achieved the best performance[5]. We used these feature types in the following experiments.

In practice, the most time consuming part in our method was the calculation of $\arg\max_k v_k$ because we need to access whole data sequentially. In an original grafting algorithm, only one feature is added from the feature candidates. We instead chose pre-defined number of largest features and added these into H. Note that even if we include these features toghater, it converges to the global optimum.

All experiments were conducted on a 3.0 GHz Xeon processor with 32GB main memory. The operation system was the Linux version 2.6.9. The compiler was g++ (gcc version 4.0.3) executed with the -O3 option.

Table 2 shows the accuracy results. The proposed method achieved the highest or the second-highest accuracy in all data sets. SVM achieved the highest performance in the MOVIE-a corpus, but very low performance in other corpora because SVM was suffered from noise words. The methods with L_1 regularization could filter out ineffective substrings, and achieved high performance in TC300B. The results for SLR were always equal to or worse than our method, because SLR searches effective-substrings in a greedy manner, and in some cases, they cannot find the effective substring. Our proposed method could successfully find the effective substrings from all substrings.

Finally we examined the scalability of the proposed method. We changed the length of an input text, and examined the time to enumerate all maximal substrings. Note that this part is the most time-consuming, and dominant part in our algorithm. Figure 3 shows the results.

This result indicate that our method can process in the proportional to the text size even if the text is very large such as 1 GB.

7 Conclusion

We proposed a novel classifier with all substrings as features and showed that we can train the document classification model with all substrings without approximation in the liner time at training.

The experimental results showed that our method achieves the highest performance in several tasks compared to other document classification methods; word-based BOW, and very recent variable-length N-gram logistic regression model [4]. Our training results are represented as a very

[2] http://www.cs.cornell.edu/People/pabo/movie-review-data/, Polarity dataset v2.0

[3] http://www.mpi-inf.mpg.de/ ifrim/data/kdd08-datasets.zip, KDD08-datasets

[4] http://techtc.cs.technion.ac.il/techtc300/techtc300.html, A: 10341-14271, B: 10539-194915

[5] There are no significant difference between idf, len, tf-idf, idf-len, tf-len and tf-idf-len

Table 1: Detail of the data set

CORPUS	NUM. DOCS	TOTAL LEN. (BYTE)	NUM. TYPE OF WORDS	NUM. MAXIMAL STRING
MOVIE-A	2000	7786004	38187	1685037
MOVE-B	7440	213970	55764	713229
TC300-A	200	1953894	16655	378673
TC300-B	200	1424566	14430	236220

Table 2: Result of the document classification task

CORPUS	PROPOSED	BOW+L_1	SLR	SVM
MOVIE-A	86.5%	83.0%	81.6%	87.2%
MOVIE-B	75.1%	71.0%	74.0%	69.1%
TC300A	80.0%	66.7%	80.0%	73.1%
TC300B	86.7%	86.7%	73.3%	71.9%

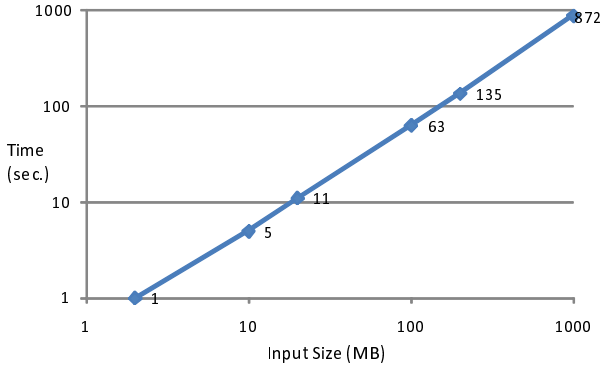

Figure 3: The time for finding all maximal substrings. The x-axis shows the input size, and the y-axis shows the time for reporting all maximal substrings.

compact set of substrings, and the inference time is very fast in theory and practice.

As a next step, we will consider an application of our method to unsupervised learning, such as clustering. We will also extend our method to find effective combination of substring information

References

[1] S. Perkins, K. Lacker, and J. Theiler. Grafting: Fast, incremental feature selection by gradient descent in function space. *JMLR*, 3:1333–1356, 2003.

[2] S. V. N Vishwanathan and A. J. Smola. Fast kernels for string and tree matching. *Kernels and Bioinformatics*, 2004.

[3] C. H. Teo and S. V. N. Vishwanathan. Fast and space efficient string kernels using suffix arrays. In *Proc. of ICML*, pages 929–936, 2006.

[4] G. Ifrim, G. Bakir, and G. Weikum. Fast logistic regression for text categorization with variable-length n-grams. In *Proc. of SIGKDD*, 2008.

[5] J. Gao, G. Andrew, M. Johnson, and K. Toutanova. A comparative study of parameter estimation methods for statistical natural language processing. In *Proc. of ACL*, pages 824–831, 2007.

[6] G. Andrew and J. Gao. Scalable training of l1-regularized log-linear models. In *Proc. of ICML*, 2007.

[7] J. Yu, S. V. N. Vishwanathan, S. Guenter, and N. Schraudolph. A quasi-Newton approach to nonsmooth convex optimization. In *Proc. of ICML*, 2008.

[8] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[9] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algs*, 2:53–86, 2004.

[10] U. Manber and E. W. Myers. Suffix arrays: A new method for online string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[11] G. H. Gonnet, R. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. *Information Retrieval: Algorithms and Data Structures*, pages 66–82, 1992.

[12] G. Nong, S. Zhang, and W. H. Chan. Two efficient aglrotihms for linear suffix array constraction, 2008.

[13] M. Yamamoto and K. W. Church. Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *Comput. Linguist.*, 27(1):1–30, 2001.

[14] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. of CPM*, pages 181–192, 2001.

[15] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[16] K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.

[17] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

A Rank Dictionary

Let $R[1,\ldots,s-1]$ be a bit vector such that $R[i]=1$ if $B[i]=B[i+1]$ and $R[i]=0$ otherwise. Then, the $B[l,r]$ consists of only one character type if and only if $R[l,r-1]$ contains only 0s. This can be checked in constant time using $n+\mathrm{o}(n)$ bits as follows. Given a bit array $R[1,\ldots,n]$, $R[i]\in\{0,1\}$, we can check whether $R[l,r]$ contains 1 or not in constant time using $n+\mathrm{o}(n)$ bits using rank dictionaries as follows. We assume a RAM-model in which all $\log n$ sized operations can be done in constant time. Rank dictionaries supports $\mathbf{rank}(R,c,p)$ that return the number of $c\in\{0,1\}$ in $R[1,\ldots,p]$. It is easy to check by $\mathbf{rank}(R,1,r)-\mathbf{rank}(R,1,l)>0$.

First, we conceptually divide an array R into large blocks of $l=\log^2$ bits, and again divide each large block into small blocks of $s=\log^n/2$ bits. We keep the results for $\mathbf{rank}(B,1,i\times l)$ in $L[n/l]$ and the number of 1's from the beginning of the large block to each small block in $S[n/s]$. We also calculate all results for the array of $\log^n/2$ bits in table using $2^{\{}\log^n/2\}=\sqrt{(}n)$ bits of space. Then $\mathbf{rank}(R,1,i)=L[\lfloor i/l\rfloor]+S[\lfloor i/s\rfloor]+popcount(R,\lfloor i/s\rfloor,i)$ where $popcount(B,i,j)$ returns the number of 1's in $B[i,j]$ in constant time by table lookup. The size of an auxiliary data is $\log nn/\log^2 n+\log\log n/(\log^n/2)+\sqrt{(}n)=\mathrm{o}(n)$