Language, Script, and Encoding Identification with String Kernel Classifiers

Canasai Kruengkrai, Virach Sornlertlamvanich, Hitoshi Isahara
Thai Computational Linguistics Laboratory
National Institute of Information and Communications Technology
112 Paholyothin Road, Klong 1, Klong Luang, Pathumthani 12120, Thailand
{canasai,virach}@tcllab.org, isahara@nict.go.jp

Abstract

This paper discusses the problem of language, script, and encoding (LSE) identification for written text based on string kernel classifiers. We describe three increasingly efficient methods of string kernel computation, including explicit mapping, brute-force matching, and suffix tree matching. To perform LSE identification, the string kernel is incorporated with two different kernel classifiers: the kernelized centroid-based method and the support vector machine classifier. We present experimental results based on subsets of UDHR collection, consisting of 10 LSE schemes used in India and 24 LSE schemes used in Africa.

1 Introduction

Language, script, and encoding (LSE) identification is one of major challenges in Web Language Engineering (WLE) research. WLE is established to study about techniques in processing the languages using in the publishing web documents. Although more than 6,900 languages reported by the Ethnologue (Gordon, 2005) are currently used all over the world, only a small number of them has been used in the cyberspace. Many efforts have been making to prevent the fall-off in using minority languages in the online community and less-computerized languages. The objective of WLE is to analyze statistics of existing web documents in the aspects of LSE schemes to raise awareness about the minority languages in the Internet and stimulate the development of web documents in the less-computerized languages.

The mechanism of automatically identifying LSE information from text body itself is extremely desired for WLE research. Unlike speech input, text does not demonstrate the variability associated with speech, e.g., speech habits, speaker emotion, etc (Muthusamy et al., 1996). One may think that language identification for written text is a simple problem. This is probably true when identification is performed on a set of languages written with different encodings. For example, in our preliminary study we experimented on a textual data set, containing Chinese and Japanese languages. Each language was composed of text samples written in 3 encodings: BIG5, GB2312 and UTF-8 for Chinese; and EUC-JP, SJIS, and UTF-8 for Japanese. Thus, we finally had 6 LSE combination classes to be identified. By testing with several classification methods, we achieved the accuracy up to 99%, since the range of encodings is well separated. However, the situation may be more difficult, if the data set is composed of languages written in the same script and encoding. For example, both Hindi and Magahi languages are written with *Devanagari* script and encoded with UTF-8. One possible solution to deal with this case is to use some linguistic resources such as monolingual dictionaries. Unfortunately, obtaining such dictionaries for all considered languages is very expensive. An alternative approach is to obtain features of each language extracted from available documents such as unique words frequently used in particular languages (Lins and Gonçalves, 2004). However, identification based on the word level comparison does not tolerate to spelling errors.

Another direction of the LSE identification techniques is based on n-gram language models. In (Dunning, 1994), n-gram models are applied with Markov assumptions in order to capture the dependency among grams. The bigrams and trigrams are computed using the maximum likelihood estimate (MLE) and associated with Bayesian decision theory for classification. However, a crucial problem of n-gram models is the data sparseness in training set. Some smoothing techniques have to be applied to deal with bias probabilities for potentially missing grams (Peng et al., 2003). In (Cavnar and Trenkle, 1994), a simple counting is used to select most frequently occurred n-grams in training data.

Based on the concept of Zipf's Law, the resulting n-grams are ranked according to their frequencies and kept as a profile. To identify a new text, the sum of the out-of-place measure is calculated from the changes of gram's positions between the profile and the new text to find the closest profile.

In this paper, we propose a statistical learning approach to LSE identification based on string kernel classifiers. Our approach does not require linguistic presuppositions about the data such as word boundaries for unsegmented text. In particular, we consider the text in a more fine-grained encoding as the string of bytes. The similarity between two strings can be computed by using a string kernel function. The basic idea of string kernel computation is to count the number of substrings in common between two strings. We discuss three increasingly efficient methods of string kernel computation. We first explain a simple method based on explicit mapping, and further describe two more sophisticated methods that attempt to compute the string kernel using the concept of pattern matching. Here we mainly focus on string kernel computation for contiguous substrings. Comprehensive evaluations conducted by (Lodhi et al., 2002) indicate that the string kernel with contiguous substrings (also referred as the n-gram kernel) works surprisingly well compared to non-contiguous substrings and the standard word kernel on the text classification task. We combine the string kernel with two kernel classifiers, including the kernelized centroid-based method and the support vector machine (SVM) classifier. We provide empirical evidence that applying the proposed approaches to LSE identification yields an impressive performance.

The remainder of the paper is organized as follows. Section 2 discusses important concepts of string kernel computation, while Section 3 describes how to combine string kernels with two different kernel classifiers. In section 4, we provide experimental results. Finally, we conclude in Section 5 with some directions of future work.

2 String Kernel Computation for Contiguous Substrings

We first introduce some notation. Let Σ be a finite collection of characters (or symbols). A string is a list of characters $\boldsymbol{u} = (u_1 \dots u_{|\boldsymbol{u}|})$ where $|\boldsymbol{u}|$ is the length of the string, and $\boldsymbol{uv} = (u_1 \dots u_{|\boldsymbol{u}|} v_1 \dots v_{|\boldsymbol{v}|})$ is the concatenation of two strings. Let Σ^i be the set of all finite substrings of length i. Thus, the set of all substrings of any length can be expressed as:

$$\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i \; . \tag{1}$$

Given two strings $\boldsymbol{u}, \boldsymbol{v} \in \Sigma^*$ where $|\boldsymbol{u}| \geq |\boldsymbol{v}|$, we define the index set as (Herbrich, 2002):

$$\mathcal{I}_{\boldsymbol{v},\boldsymbol{u}} = \{ i : (i + |\boldsymbol{v}| - 1) \mid i \in \{1, \dots, |\boldsymbol{u}| - |\boldsymbol{v}| + 1\} \} \; , \tag{2}$$

which is the set of all contiguous substrings of length $|\boldsymbol{v}|$ in $|\boldsymbol{u}|$. Let $\mathbf{i} = (i_1 \dots i_r)$ with $1 \leq i_1 < \dots < i_r \leq |\boldsymbol{u}|$ be an index vector, and $\boldsymbol{u}[\mathbf{i}] = (u_{i_1} \dots u_{i_r})$ be a substring. We say that $\boldsymbol{v}$ is a substring of $\boldsymbol{u}$ if there exists an index vector $\mathbf{i} \in \mathcal{I}_{\boldsymbol{v},\boldsymbol{u}}$ such that $\boldsymbol{v} = \boldsymbol{u}[\mathbf{i}]$.

2.1 Explicit Mapping

The idea of explicit mapping is to represent a string as a feature vector of all possible substrings of (up to) length r, and compute the inner product of any two strings directly. Let $I(\alpha)$ be the indicator function on a predicate α. Given a fixed lexicon $\mathcal{L} \subset \Sigma^*$, the mapping function $\Phi : \Sigma^* \rightarrow \mathcal{H}$ that maps a string from Σ^* to a feature space $\mathcal{H}$ can be written as:

$$\Phi(\boldsymbol{v}) = \lambda^{|\boldsymbol{s}|} \sum_{\mathbf{i} \in \mathcal{I}_{\boldsymbol{s},\boldsymbol{v}}} I(\boldsymbol{s} = \boldsymbol{v}[\mathbf{i}]) \; , \tag{3}$$

where λ is an arbitrary weight of the matched substring, and $\boldsymbol{s}$ is a word in $\mathcal{L}$.

Let us describe how the string kernel can be computed with explicit mapping through an example. Consider two strings $\boldsymbol{u} = (yzxxz)$ and $\boldsymbol{v} = (xyzxxxy)$, where $1 \leq r \leq 2$ and $\Sigma = \{x, y, z\}$. Consequently, the set of finite substrings becomes $\Sigma^1 \cup \Sigma^2$, which is equal to the set of tokens in $\mathcal{L}$. We then perform mapping the two strings to their corresponding feature vectors. By organizing all possible substrings in the lexicographic order, for substrings in Σ^1, we get

Σ^1	$\Phi_1(\boldsymbol{u})$	$\Phi_1(\boldsymbol{v})$	$\Phi_1(\boldsymbol{u}) \cdot \Phi_1(\boldsymbol{v})$
x	$2\lambda^1$	$4\lambda^1$	$8\lambda^2$
y	λ^1	$2\lambda^1$	$2\lambda^2$
z	$2\lambda^1$	λ^1	$2\lambda^2$

Thus, we can simply compute the value for K_1 by

$$K_1(\boldsymbol{u}, \boldsymbol{v}) = 8\lambda^2 + 2\lambda^2 + 2\lambda^2 = 12\lambda^2.$$

For substrings in Σ^2, we obtain

Σ^2	$\Phi_2(\boldsymbol{u})$	$\Phi_2(\boldsymbol{v})$	$\Phi_2(\boldsymbol{u})\cdot\Phi_2(\boldsymbol{v})$
xx	λ^2	$2\lambda^2$	$2\lambda^4$
xy	0	$2\lambda^2$	0
xz	λ^2	0	0
yx	0	0	0
yy	0	0	0
yz	λ^2	λ^2	λ^4
zx	λ^2	λ^2	λ^4
zy	0	0	0
zz	0	0	0

Similarly, we can compute the value for K_2 by

$$K_2(\boldsymbol{u},\boldsymbol{v}) = 2\lambda^4 + \lambda^4 + \lambda^4 = 4\lambda^4.$$

Finally, the overall value of the kernel between the two strings $\boldsymbol{u}$ and $\boldsymbol{v}$ is

$$K_r(\boldsymbol{u},\boldsymbol{v}) = K_1(\boldsymbol{u},\boldsymbol{v}) + K_2(\boldsymbol{u},\boldsymbol{v}) = 12\lambda^2 + 4\lambda^4.$$

2.2 Brute-Force Matching

The explicit mapping method that we just described needs to define the lexicon $\mathcal{L}$, so we have to extract all possible substrings from the data to construct $\mathcal{L}$. The computational time of the native extraction may be prohibitive (see (Yamamoto and Church, 2001) for more details on this issue). This motivates a more general method for computing the string kernel based on a dynamic alignment. The idea is to count the number of substrings in common between two strings using a simple pattern matching technique. Focusing on all substrings of the maximum length r, the string kernel for the dynamic alignment can be expressed as (Herbrich, 2002):

$$K_r(\boldsymbol{u},\boldsymbol{v}) = \sum_{i=1}^{r} \lambda^{2i} \sum_{\boldsymbol{s}\in\Sigma^i} \sum_{\mathbf{i}\in\mathcal{I}_{s,u}} \sum_{\mathbf{j}\in\mathcal{I}_{s,v}} I(\boldsymbol{s} = \boldsymbol{u}[\mathbf{i}] = \boldsymbol{v}[\mathbf{j}]) \,, \tag{4}$$

which can be implicitly computed by using the following recursion:

$$K_r(u_1\boldsymbol{u},\boldsymbol{v}) = \begin{cases} 0, & \text{if } |u_1\boldsymbol{u}| = 0; \\ K_r(\boldsymbol{u},\boldsymbol{v}) + \sum_{j=1}^{|\boldsymbol{v}|} \lambda^2 \cdot K'_r(u_1\boldsymbol{u},\boldsymbol{v}), & \text{otherwise}, \end{cases} \tag{5}$$

$$K'_r(u_1\boldsymbol{u}, v_1\boldsymbol{v}) = \begin{cases} 0, & \text{if } r = 0; \\ 0, & \text{if } |u_1\boldsymbol{u}| = 0; \\ 0, & \text{if } |v_1\boldsymbol{v}| = 0; \\ 0, & \text{if } u_1 \neq v_1; \\ \left(1 + \lambda^2 \cdot K'_{r-1}(\boldsymbol{u},\boldsymbol{v})\right), & \text{otherwise}. \end{cases} \tag{6}$$

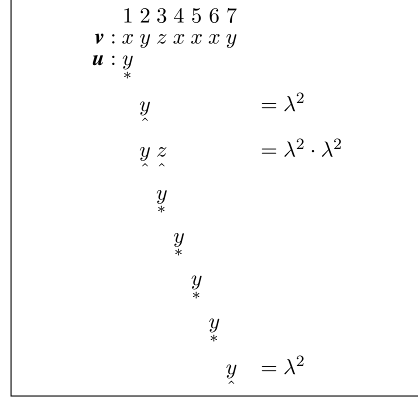

Figure 1: Substring matching of an inner iteration starting at the leftmost character of $\boldsymbol{u}$.

The interpretation of this recursion is straightforward. Equations (5) and (6) can be considered as the outer and the inner iterations, respectively. Each outer iteration begins with the current leftmost character and proceeds through the rightmost character of $\boldsymbol{u}$, whereas each inner iteration performs comparison of substrings in $\boldsymbol{u}$ and $\boldsymbol{v}$ up to the length r. As a result, we can think of this process as brute-force matching that searches all substrings of $\boldsymbol{u}$ in $\boldsymbol{v}$, which its computational complexity is $O(r|\boldsymbol{u}||\boldsymbol{v}|)$.

Figure 1 shows an example (taken from the previous example described in Section 2.1) of substring matching of an inner iteration starting at the leftmost character of $\boldsymbol{u}$. A caret beneath a character indicates a match and a star indicates a mismatch. When a substring is found, a score of the kernel is computed and stored. The following table lists all the matches between $\boldsymbol{u}$ and $\boldsymbol{v}$, where the first column shows the starting characters of $\boldsymbol{u}$, the second column shows the matched substrings in $\boldsymbol{v}$ (the subscript in each character indicates the matched position), and the last column shows their scores.

u	$\boldsymbol{v}[\mathbf{i}]$	score
y	y_2, y_2z_3, y_7	$2\lambda^2 + \lambda^4$
z	z_3, z_3x_4	$\lambda^2 + \lambda^4$
x	$x_1, x_4, x_4x_5, x_5, x_5x_6, x_6$	$4\lambda^2 + 2\lambda^4$
x	x_1, x_4, x_5, x_6	$4\lambda^2$
z	z_7	λ^2

By summing all the scores, the value of the kernel between the two strings $\boldsymbol{u}$ and $\boldsymbol{v}$ becomes $K_r(\boldsymbol{u},\boldsymbol{v}) = 12\lambda^2 + 4\lambda^4$, which is exactly equal to the value of explicit mapping computation.

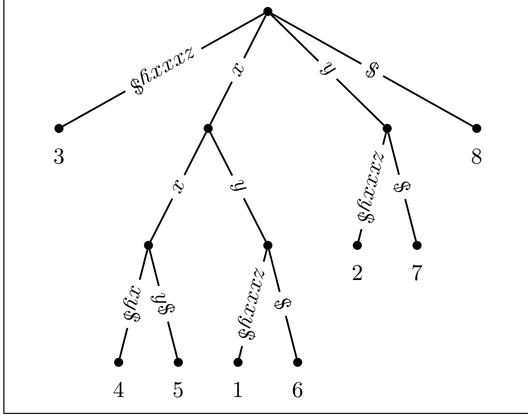

Figure 2: Suffix tree for the string $\boldsymbol{v} = xyzxxxy\$$.

2.3 Faster Matching with Suffix Trees

As mentioned earlier, the computational complexity of brute-force matching is quadratic in the lengths $|\boldsymbol{u}|$ and $|\boldsymbol{v}|$, since it performs iterative searches in $\boldsymbol{v}$ for every substring in $\boldsymbol{u}$. However, it is possible to accelerate the kernel computation that has the complexity on the order of $|\boldsymbol{u}|+|\boldsymbol{v}|$ by applying a data structure called suffix trees (Gusfield, 1997). The suffix trees have been widely used in computational sequence analysis, e.g., checking DNA sequences against a database and searching queries in a large text corpus. The idea is to first preprocess the target string by representing it with a suffix tree, and then use this suffix tree to answer the queries. This leads to a very efficient method in which the search time is proportional to the size of the queries rather than the size of the target string.

In our context, given a string $\boldsymbol{v}$, we need to build a suffix tree for $\boldsymbol{v}$ in $O(|\boldsymbol{v}|)$ time. Then, given a query substring $\boldsymbol{u}[\mathbf{i}]$ with the length r, we can determine whether $\boldsymbol{u}[\mathbf{i}]$ is a substring of $\boldsymbol{v}$ in $O(r)$ time. If $\boldsymbol{u}[\mathbf{i}]$ occurs in $\boldsymbol{v}$ precisely k times, then we can find all occurrences in $O(r+k)$ time (Gusfield, 1997). We perform lookup at most $|\boldsymbol{u}|$ times according to the length of $\boldsymbol{u}$. Thus, the overall time complexity for computing the kernel is $O(c|\boldsymbol{u}|+|\boldsymbol{v}|)$, where $c = r+k$. By applying the Ukkonen's algorithm (Ukkonen, 1995), we can build the suffix tree in the linear time corresponding to the size of a given string. The Ukkonen's algorithm has an on-line property that can construct the suffix tree incrementally without scanning the entire string.

Figure 2 shows the suffix tree for the string $\boldsymbol{v} = (xyzxxxy\$)$ taken from the previous example in Section 2.1 (added with the termination character $). Based on the data structure of the suffix tree, we derive two important properties, including (a) the unique path from the root to a leaf node corresponds to a suffix (denoted by the number below that leaf node), and (b) the number of all leaf nodes in the subtree below the point of the last match indicates all occurrences of a substring.

For example, in order to determine whether a substring $\boldsymbol{u}[\mathbf{i}] = yz$ occurs in $\boldsymbol{v}$ and find its occurrences, we match the characters of $\boldsymbol{u}[\mathbf{i}]$ along the unique path from the root until $\boldsymbol{u}[\mathbf{i}]$ is exhausted. Thus, from the property (b), we can conclude that y occurs 2 times and yz occurs 1 time in $\boldsymbol{v}$. The kernel score can be computed using Equation (4), where the inner sum can be efficiently obtained in a constant time. The following table lists all the matches between $\boldsymbol{u}$ and $\boldsymbol{v}$, where the number of all occurrences of substrings in common is given in parentheses.

u	$\boldsymbol{v}[\mathbf{i}]$	score
y	$y(2), yz(1)$	$2 \cdot \lambda^{2 \cdot 1} + 1 \cdot \lambda^{2 \cdot 2} = 2\lambda^2 + \lambda^4$
z	$z(1), zx(1)$	$1 \cdot \lambda^{2 \cdot 1} + 1 \cdot \lambda^{2 \cdot 2} = \lambda^2 + \lambda^4$
x	$x(4), xx(2)$	$4 \cdot \lambda^{2 \cdot 1} + 2 \cdot \lambda^{2 \cdot 2} = 4\lambda^2 + 2\lambda^4$
x	$x(4), xz(0)$	$4 \cdot \lambda^{2 \cdot 1} = 4\lambda^2$
z	$z(1)$	$1 \cdot \lambda^{2 \cdot 1} = \lambda^2$

Again, by summing all the scores, we get the same value of the kernel as in the explicit mapping computation and the brute-force matching method, which is $K_r(\boldsymbol{u}, \boldsymbol{v}) = 12\lambda^2 + 4\lambda^4$. Algorithm 1 provides an outline of the string kernel computation with the suffix tree. The algorithm starts by building the suffix tree for $\boldsymbol{v}$ based on the Ukkonen's algorithm. It then iterates over all substrings of $\boldsymbol{u}$ to find their matches with the maximum length r. Note that, in the implementation, it is unnecessary to start from the first character of the substring in the inner loop every time. Since the search always walks through the same unique path for each substring, the value of the kernel can be computed accumulatively.

3 Kernel Classifiers

So far we have discussed several algorithms for string kernel computation. To perform classification, we can combine such algorithms with arbitrary kernel classifiers. Here we focus on two kernel classifiers: the kernelized version of the centroid-based method and the SVM classifier.

Algorithm 1 Suffix tree matching string kernel computation

Input: Strings $\boldsymbol{u}$ and $\boldsymbol{v}$, the maximum substring length r, and the weight λ.
Output: The kernel score $K_r(\boldsymbol{u}, \boldsymbol{v})$.

1: // Build a suffix tree for $\boldsymbol{v}$ using Ukkonen's algorithm
2: $S(\boldsymbol{v}) \longleftarrow UkkonenBuild(\boldsymbol{v})$
3: $K_r(\boldsymbol{u}, \boldsymbol{v}) \longleftarrow 0$
4: **for** $i = 1, 2, \ldots, |\boldsymbol{u}|$ **do**
5: **initialize** a substring $\boldsymbol{u}[\mathbf{i}]$ with $\mathbf{i} = (i_1, \ldots, i_r)$
6: **for** $j = 1, 2, \ldots, r$ **do**
7: **match** $\boldsymbol{u}[i_1, \ldots, i_j]$ along the unique path of $S(\boldsymbol{v})$
8: **if** $\boldsymbol{u}[i_1, \ldots, i_j]$ is exhausted **then**
9: $n \longleftarrow$ *number of all leaf nodes in the subtree below the point of the last match*
10: $K_r(\boldsymbol{u}, \boldsymbol{v}) \longleftarrow K_r(\boldsymbol{u}, \boldsymbol{v}) + n \cdot \lambda^{2j}$
11: **end if**
12: **end for**
13: **end for**

3.1 Kernelized Centroid-Based Method

The centroid-based method is a simple but effective classifier. The idea of the algorithm is closely related to the Cavnar and Trenkle's algorithm (Cavnar and Trenkle, 1994) that tries to construct the language profile from training data, but we use the centroid (or mean) vector as the profile and apply the squared Euclidean distance instead of the out-of-place measure. Training samples are pre-categorized according to their languages, forming the disjoint subsets $\mathcal{C}_1, \ldots, \mathcal{C}_m$. Using the mapping function Φ, the centroid vector for each $\mathcal{C}_j$ can be calculated by:

$$\mathbf{m}_j = \frac{1}{|\mathcal{C}_j|} \sum_{\boldsymbol{v} \in \mathcal{C}_j} \Phi(\boldsymbol{v}) \ . \tag{7}$$

We can classify a new string $\boldsymbol{u}$ as a member of the centroid $\mathcal{C}^*$ with the minimum squared Euclidean distance:

$$\mathcal{C}^* = \mathrm{argmin}_j \parallel \Phi(\boldsymbol{u}) - \mathbf{m}_j \parallel^2 \ . \tag{8}$$

By replacing (7) in (8), we obtain:

$$\parallel \Phi(\boldsymbol{u}) - \frac{1}{|\mathcal{C}_j|} \sum_{\boldsymbol{v} \in \mathcal{C}_j} \Phi(\boldsymbol{v}) \parallel^2 \ . \tag{9}$$

If we further expand (9) and write $K_r(\cdot, \cdot)$ for inner product terms, we can derive:

$$K_r(\boldsymbol{u}, \boldsymbol{u}) - \frac{2}{|\mathcal{C}_j|} \sum_{\boldsymbol{v} \in \mathcal{C}_j} K_r(\boldsymbol{u}, \boldsymbol{v}) + \frac{1}{|\mathcal{C}_j|^2} \sum_{\boldsymbol{v}, \boldsymbol{w} \in \mathcal{C}_j} K_r(\boldsymbol{v}, \boldsymbol{w}) \ . \tag{10}$$

However, it is unnecessary to compute the first term in (10), since it is a constant and does not affect classification of $\boldsymbol{u}$ to the centroid vector. Furthermore, one can notice that the third term is fixed for each $\mathcal{C}_j$, so it can be pre-computed and stored. There remains only the second term to be computed during the classification process. Thus, the discriminant function can be eventually written as:

$$\mathcal{C}^* = \mathrm{argmin}_j \frac{-2}{|\mathcal{C}_j|} \sum_{\boldsymbol{v} \in \mathcal{C}_j} K_r(\boldsymbol{u}, \boldsymbol{v}) + \gamma_j \ , \tag{11}$$

where

$$\gamma_j = \frac{1}{|\mathcal{C}_j|^2} \sum_{\boldsymbol{v}, \boldsymbol{w} \in \mathcal{C}_j} K_r(\boldsymbol{v}, \boldsymbol{w}) \ . \tag{12}$$

3.2 SVM Classifier

Based on the structural risk minimization principle from the statistical learning theory (Vapnik, 1995), support vector machines have been applied to many real world applications with remarkable performance. A simple formulation of the SVM classifier can be explained through the binary classification problem, which attempts to separate the data into two categories corresponding to $y_i = 1$ and $y_i = -1$. We note that it can be extended to the multi-class classification using some techniques such as the pairwise binary classification (Hsu and Lin, 2002).

Given training samples $(\boldsymbol{v}_1, y_1), \ldots, (\boldsymbol{v}_l, y_l)$, in order to obtain the hyperplane having the maximum margin with the closest training samples, we consider the following primal optimization problem:

$$\textit{minimize:} \quad \boldsymbol{V}(\mathbf{w}, b, \xi) = \frac{1}{2}\|\mathbf{w}\|^2 + C \sum_{i=1}^{l} \xi_i \tag{13}$$

$$\textit{subject to:} \quad y_i(\mathbf{w} \cdot \Phi(\boldsymbol{v}_i) + b) \geq 1 - \xi_i, \quad \xi_i \geq 0 \ , \tag{14}$$

where ξ_i is a slack variable, and C is a parameter for controlling the tradeoff between the training error and the maximum margin.

In practice, we work with its dual form instead of directly solving the primal problem by

Bengali.Bengali.UTF-8
Gujarati.Gujarati.UTF-8
Hindi.Devanagari.UFT-8
Kannada.Kannada.UTF-8
Magahi.Devanagari.UTF-8
Marathi.Devanagari.UTF-8
Punjabi.Arabic.UTF-8
Sanskrit.Devanagari.UTF-8
Saraiki.Arabic.UTF-8
Tamil.Tamil.UTF-8

Table 1: LSE schemes of India10 data set.

Afrikaans.Latin.ISO-8859-1	Malagasy.Latin.ISO-8859-1
Arabic.Arabic.UTF-8	Nyanja-Chechew.Latin.ISO-8859-1
Bambara.Latin*.UTF-8	Nyanja-Chinyanja.Latin.ISO-8859-1
Bemba.Latin.ISO-8859-1	Oromiffa.Latin.ISO-8859-1
English.Latin.ISO-8859-1	Portuguese.Latin.ISO-8859-1
Ewe.Latin*.UTF-8	Somali.Latin.ISO-8859-1
Fon.Latin*.UTF-8	Spanish.Latin.ISO-8859-1
French.Latin.ISO-8859-1	Swahili.Latin.ISO-8859-1
Hausa.Latin.ISO-8859-1	Themne.Latin*.UTF-8
Ibibio.Latin.ISO-8859-1	Tonga.Latin.ISO-8859-1
Italian.Latin.ISO-8859-1	Yoruba.Latin.UTF-8
Lingala.Latin.ISO-8859-1	Zulu.Latin.ISO-8859-1

Latin* = Latin alphabet using some additional phonetic characters

Table 2: LSE schemes of Africa24 data set.

introducing Lagrange multipliers α_i and using the Kuhn-Tucker condition. When we successfully solve the dual problem, we can get the values of variables **w** and b. Only samples $\boldsymbol{v}_i$ that lie closest to the hyperplane have nonzero values α_i, which are called *support vectors*. Given a new string $\boldsymbol{u}$, we classify it by using the following decision function:

$$f(\boldsymbol{u}) = sgn\Big(\sum_i \alpha_i K_r(\boldsymbol{v}_i, \boldsymbol{u}) + b\Big) . \qquad (15)$$

4 Experimental Results

4.1 Data Sets

The text collection of Universal Declaration of Human Rights (UDHR)[1] was used in our experiments. The UDHR can avoid the domain dependence problem (Dunning, 1994) in LSE identification problem, since it consists of 332 translated texts. We derived 2 subsets of UDHR according to languages used in 2 different regions of the world (Nakanishi, 1998). Tables 1 and 2 show the considered languages (labeled with language.script.encoding format) forming India10 and Africa24 data sets, respectively. The text sizes range between 18-38 Kbytes in India10, and 9-17 Kbytes in Africa24.

During preprocessing, we skipped headers but stored the content lines. In India10, we randomly extracted 150 samples from each text to form training and test sets. As a result, we obtained 1,500 samples, and the length of each sample was fixed at 100 bytes. In Africa24, we also generated 150 samples, but the length of each sample was reduced to 50 bytes due to its smaller text size. We obtained 3,600 samples on this data set. The results reported in the following section were averaged over 5 trials of random extractions. In each trial, we performed stratified 3-fold cross validation.

In experiments, we also considered 2 conditions that affect the performance of LSE identification, including (1) the amount of training samples and (2) the length of test strings. These conditions were varied from the initial setting. In India10, we gradually decreased training data with sizes of 10, 5, and 2 Kbytes per language. In Africa24, training data were varied by 5, 2.5, and 1 Kbytes per language.

For the purpose of comparison, we used TextCat software[2] that is an implementation based on (Cavnar and Trenkle, 1994) as the baseline. We implemented our LSE classifiers, the string kernel SVM classifier (SKSVM) and the kernelized centroid based method (KCB), based on LIBSVM (Chang and Lin, 2004). We adapted an implementation of a suffix tree[3] that can perform the Ukkonen's algorithm. We fixed the maximum substring length $r = 5$ for all the classifiers. For kernel computations, we fixed the weight of matched substrings at $\lambda = 1.5$.

4.2 Results

Figure 3 shows identification accuracy on India10 data set by varying the size of training data and the length of test strings. SKSVM outperforms both KCB and TextCat on all settings. We can see that the performance of all classifiers drops as the size of training samples decreases. SKSVM produces a reasonable gap of accuracy when all the training samples are available. This indicates the generalization capability of SKSVM over other classifiers. KCB and

[1]The collection of translations is available at `http://www.unhchr.ch/udhr`. For Indian translations, see `http://gii.nagaokaut.ac.jp`.

[2]`http://www.let.rug.nl/~vannoord/TextCat`.

[3]`http://cs.haifa.ac.il/~shlomo/suffix_tree`.

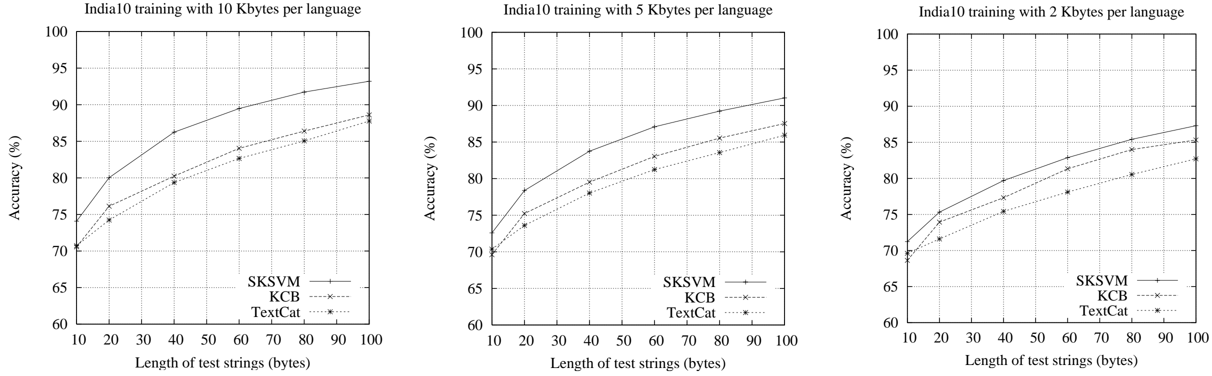

Figure 3: Identification accuracy on India10 with 10, 5, and 2 Kbtyes training data per language.

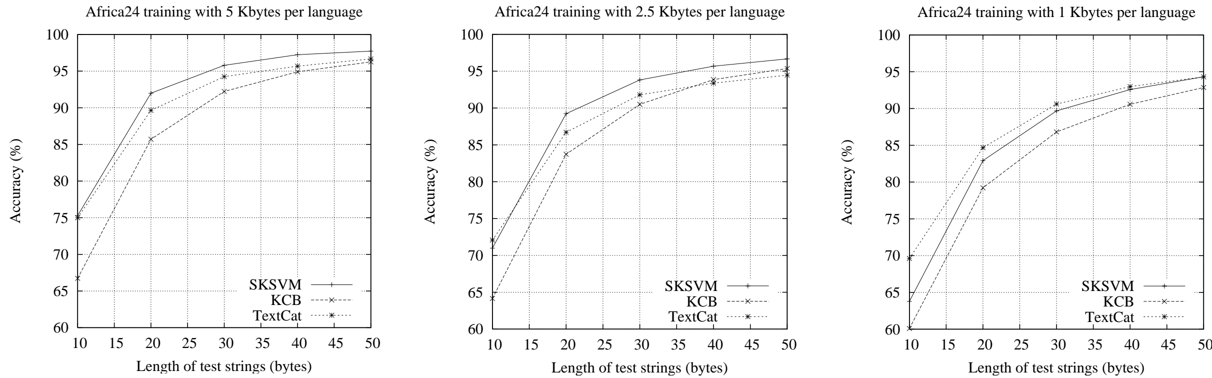

Figure 4: Identification accuracy on Africa24 with 5, 2.5, and 1 Kbtyes training data per language.

TextCat demonstrate the similar trend of accuracy curves on 10 and 5 Kbytes subsets.

Figure 4 shows identification accuracy on Africa24 data set. With the initial setting (5 Kbytes per language), SKSVM yields the best result, following by TextCat and KCB. TextCat shows a good performance on this data set. It outperforms SKSVM and KCB on some settings, especially when we reduced the size of training data to just 1 Kbytes per language. Note that Africa24 data set is mostly composed of ISO-8859-1, which is a single-byte character encoding. This is in contrast to India10 data set where contains UTF-8, which is a variable-length character encoding. The different properties of data sets indicate the discrimination power of the candidate classifiers. From the accuracy curves, we observe that identifying LSE on India10 is harder than Africa24, since the performance does not significantly increase as the training size increases.

To present the identification results more clearly, Figures 5 and 6 show confusion matrices generated by SKSVM and TextCat on India10 with 5 Kbytes training data per language. Rows indicate the actual class labels, where only the language names are shown. Columns indicate the identification results. We can see that it is easy to identify languages written in different scripts, such as Tamil, Kannada, Gujarati, and Bengali. However, both SKSVM and TextCat suffer from the ambiguity between Saraiki and Punjabi, since these two languages are written in the same *Arabic* script and UTF-8 encoding. Another ambiguity occurs in *Devanagari* script, including Sanskrit, Marathi, Magahi, and Hindi. These case examples issue a challenge to further studies to identify languages that use a very similar writing system.

5 Conclusion and Future Work

We have discussed LSE identification based on string kernel classifiers. First, we introduce several ideas of the string kernel computation for contiguous substrings. Then, two kernel classifiers are described. The classifiers can combine the string kernel as the core module for computing the similarities among text samples. Fi-

Language		0	1	2	3	4	5	6	7	8	9
Tamil	0	50	.	.	.	.	.	.	.	.	.
Saraiki	1	.	46	.	4	.	.	.	.	.	.
Sanskrit	2	.	.	47	.	1	1	.	1	.	.
Punjabi	3	.	9	.	41	.	.	.	.	.	.
Marathi	4	.	.	3	.	43	.	.	4	.	.
Magahi	5	.	.	.	.	5	38	.	7	.	.
Kannada	6	.	.	.	.	.	.	50	.	.	.
Hindi	7	.	.	1	.	8	9	.	32	.	.
Gujarati	8	.	.	.	.	.	.	.	.	50	.
Bengali	9	.	.	.	.	.	.	.	.	.	50

Figure 5: Confusion matrix on India10 with 5 Kbytes training data generated by SKSVM.

Language		0	1	2	3	4	5	6	7	8	9
Tamil	0	50	.	.	.	.	.	.	.	.	.
Saraiki	1	.	42	.	8	.	.	.	.	.	.
Sanskrit	2	.	.	43	.	3	.	.	4	.	.
Punjabi	3	.	20	.	30	.	.	.	.	.	.
Marathi	4	.	.	9	.	34	2	.	5	.	.
Magahi	5	.	.	.	.	2	35	.	13	.	.
Kannada	6	.	.	.	.	.	.	50	.	.	.
Hindi	7	.	.	5	.	9	12	.	24	.	.
Gujarati	8	.	.	.	.	.	.	.	.	50	.
Bengali	9	.	.	.	.	.	.	.	.	.	50

Figure 6: Confusion matrix on India10 with 5 Kbytes training data generated by TextCat.

nally, we report empirical results based on two different subsets of UDHR collection, including India10 and Africa24 data sets. SKSVM yields the best performance on almost all experimental settings conducted in this paper. In practice, if the amount of text samples for a considered language is available enough (e.g., 5-10 Kbytes), SKSVM is a good choice for automatic identification, since it is more robust to a variety of languages, scripts, and encodings.

In future work, we plan to conduct experiments at a larger scale. The problem is how to deal with a extremely large number of classes (languages), e.g., all the 332 available languages in UDHR collection. Can we maintain the identification performance at 90-95% at that class size? Another issue is how to detect an unknown language that does not occur in the training data. These are interesting open problems that need to be explored.

Acknowledgment

We thank Yoshiki Mikami and Language Observatory team for providing us the collection of UDHR translations with converting to plain text format.

References

William B. Cavnar and John M. Trenkle. 1994. N-gram-based text categorization. In *Proceedings of the Third Annual Symposium on Document Analysis and Information Retrieval*, pages 161–169.

Chih-Chung Chang and Chih-Jen Lin. 2004. LIBSVM: a library for support vector machines (version 2.71). The software is available at: `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

Ted Dunning. 1994. Statistical identification of language. *Technical Report MCCS 94-273, New Mexico State University*.

Raymond G. Gordon, Jr, editor. 2005. *Ethnologue: Languages of the World, Fifteenth Edition*. SIL International.

Dan Gusfield. 1997. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York.

Ralf Herbrich. 2002. *Learning Kernel Classifiers: Theory and Algorithms*. The MIT Press.

Chih-Wei Hsu and Chih-Jen Lin. 2002. A comparison of methods for multi-class support vector machines. *IEEE Transactions on Neural Networks*, 13:415–425.

Rafael Dueire Lins and Paulo Gonçalves. 2004. Automatic language identification of written texts. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1128–1133.

Huma Lodhi, Craig Saunders, John Shawe-Taylor, Nello Cristianini, and Chris Watkins. 2002. Text classification using string kernels. *The Journal of Machine Learning Research*, 2:419–444.

Yeshwant K. Muthusamy, Etienne Barnard, and Ronald A. Cole. 1996. Automatic Language Identification. In *State of the Art in Human Language Technology*, pages 273–285. Cambridge University Press.

Akira Nakanishi. 1998. *Writing Systems of the World*. Charles E. Tuttle Co., Inc.

Fuchun Peng, Dale Schuurmans, and Shaojun Wang. 2003. Language and task independent text categorization with simple language models. In *HLT-NAACL 2003*.

Esko Ukkonen. 1995. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260.

Vladimir Vapnik. 1995. *The Nature of Statistical Learning Theory*. Springer-Verlag, Berlin.

Mikio Yamamoto and Kenneth W. Church. 2001. Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *Computational Linguistics*.