Language Identification Based on String Kernels

Canasai Kruengkrai, Prapass Srichaivattana, 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,prapass,virach}@tcllab.org, isahara@nict.go.jp

***Abstract*— In this paper, we propose a novel approach for automatically identifying the language of a given text based on the concept of string kernels. Our approach can identify the language from the text directly, regardless of its coding system. In particular, we view the text in a more fine-grained encoding as the string of bytes. The similarity between two strings can be implicitly computed through an efficient dynamic alignment using suffix trees. We provide empirical evidence that applying the string kernels to the language identification problem yields an impressive performance using two different kernel classifiers: the kernelized version of the centroid-based method and the support vector machines. Our experiments are based on a reasonable scale of the data set in terms of the number of languages to be identified, including 17 different languages.**

***Keywords*— language identification, string kernels, n-gram models, suffix trees**

I. Introduction

Language identification has become an increasingly crucial technique for many applications such as search engines and language translation systems. However, for many web pages, HTML-tag information like charset is not always useful. For example, the standard coding system ISO-8859-1 is used for 19 European languages. Thus, the mechanism of identifying the languages from text body itself is extremely desired.

Without loss of generality, we can consider language identification as a classification problem. Given a set of texts and their associated language labels, the idea is to first learn characteristics of languages from the training data, and then classify an unknown text to the most probable language based on the learning model. The problem is how to recognize the characteristic of each language. One simple solution is to use some linguistic resources such as monolingual dictionaries. Unfortunately, obtaining such dictionaries for all considered languages is very expensive. Another economical solution is to obtain features of each language extracted from available written texts such as common words [6]. However, classification based on the word level comparison does not tolerate to spelling errors.

Consequently, many statistical learning algorithms for language identification are based on n-gram language models. One competitive advantage of n-gram language models is the language-independent property. For example, a set of four-grams for the string 'language model' can be expressed as {lang, angu, ngua, guag, uage, age␣, ge␣m, e␣mo, ␣mod, mode, odel}. However, extracting all the unique n-grams for a text given corpus is not a trivial task. It needs some methods to parse and count n-grams efficiently.

In this paper, we propose a novel approach to language identification for written texts based on *string kernels*. Our approach can identify the language from a given text directly, regardless of its coding system. This is in contrast to work by [5] that divides the language identification task into two steps by first determining the coding system and then identifying the language. In particular, we view the text in a more fine-grained encoding as the string of bytes. The similarity between two strings can be computed through an efficient dynamic alignment using suffix trees. We provide empirical evidence that applying the string kernels to the language identification problem yields an impressive performance using two different kernel classifiers: the kernelized version of the centroid-based method and the support vector machines.

The remainder of the paper is organized as follows. Section II describe important concepts of string kernel computation, while Section III describes how to combine string kernels with two different kernel classifiers, including the kernelized version of the centroid-based method and the support vector machines. In section IV, we provide experimental results. Finally, we conclude in Section V with some directions of future work.

II. String Kernel Computation

We first introduce some notation. Let Σ be a finite collection of characters (or symbols). A string is a list of characters $\boldsymbol{u} = (u_1 \ldots u_{|\boldsymbol{u}|})$ where $|\boldsymbol{u}|$ is the length of the string, and $\boldsymbol{uv} = (u_1 \ldots u_{|\boldsymbol{u}|} v_1 \ldots v_{|\boldsymbol{v}|})$ 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 [4]:

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

which is the set of all contiguous substrings of length $|\boldsymbol{v}|$ in $|\boldsymbol{u}|$. Let $\mathbf{i} = (i_1 \ldots i_r)$ with $1 \leq i_1 < \ldots < i_r \leq |\boldsymbol{u}|$ be an index vector, and $\boldsymbol{u}[\mathbf{i}] = (u_{i_1} \ldots u_{i_r})$ 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 $\boldsymbol{v} = \boldsymbol{u}[\mathbf{i}]$.

A. Brute-Force Matching

One can represent strings as feature vectors of all possible substrings of (up to) length r, and computes their inner products directly. The substrings are also known as r-spectrums of the strings, which are equivalent to n-grams in the language modeling where $r = n$. Thus, we have to extract all possible substrings from the data to construct the features. The computational time of the native extraction may be prohibitive. 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 [4]:

$$K_r(\boldsymbol{u},\boldsymbol{v}) = \sum_{i=1}^{r} \lambda^{2i} \sum_{\boldsymbol{s}\in\Sigma^i} \sum_{\mathbf{i}\in\mathcal{I}_{\boldsymbol{s},\boldsymbol{u}}} \sum_{\mathbf{j}\in\mathcal{I}_{\boldsymbol{s},\boldsymbol{v}}} I(\boldsymbol{s} = \boldsymbol{u}[\mathbf{i}] = \boldsymbol{v}[\mathbf{j}]) \ , \quad (3)$$

which can be implicitly computed by using the following recursion:

$$K_r(u_1\boldsymbol{u},\boldsymbol{v}) = \begin{cases} 0, & \text{if } |u_1\boldsymbol{u}| = 0; \\ K_r(\boldsymbol{u},\boldsymbol{v}) + \sum_{j=1}^{|\boldsymbol{v}|} \lambda^2 \cdot K'_r(u_1\boldsymbol{u},\boldsymbol{v}), & \\ \quad \text{otherwise}, & \end{cases} \quad (4)$$

$$K'_r(u_1\boldsymbol{u}, v_1\boldsymbol{v}) = \begin{cases} 0, & \text{if } r = 0; \\ 0, & \text{if } |u_1\boldsymbol{u}| = 0; \\ 0, & \text{if } |v_1\boldsymbol{v}| = 0; \\ 0, & \text{if } u_1 \neq v_1; \\ \left(1 + \lambda^2 \cdot K'_{r-1}(\boldsymbol{u},\boldsymbol{v})\right), & \\ \quad \text{otherwise}. & \end{cases} \quad (5)$$

The interpretation of this recursion is straightforward. Equations (4) and (5) 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 $O(r|\boldsymbol{u}||\boldsymbol{v}|)$.

B. 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 [3]. 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 [3]. 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 [8], we can build the suffix tree in the linear time corresponding to the size of a given string. Our approach is similar to work by [10] that also uses the suffix tree but applies the McCreight's algorithm [7] in the construction process. The advantages of the Ukkonen's algorithm are that it is easier to understand and implement, and it has an on-line property that can construct the suffix tree incrementally without scanning the entire string.

Algorithm 1 Suffix tree matching string kernel computation

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

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
```

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.

III. Kernel Classifiers

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

A. 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 [1] 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:

$$\mathbf{m}_j = \frac{1}{|\mathcal{C}_j|} \sum_{\boldsymbol{v} \in \mathcal{C}_j} \Phi(\boldsymbol{v}) \ . \tag{6}$$

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

$$\mathcal{C}^* = \text{argmin}_j \parallel \Phi(\boldsymbol{u}) - \mathbf{m}_j \parallel^2 \ . \tag{7}$$

By replacing (6) in (7), we obtain:

$$\parallel \Phi(\boldsymbol{u}) - \frac{1}{|\mathcal{C}_j|} \sum_{\boldsymbol{v} \in \mathcal{C}_j} \Phi(\boldsymbol{v}) \parallel^2 \ . \tag{8}$$

If we further expand (8) and write $K_r(\cdot,\cdot)$ for inner product terms, we can derive:

$$K_r(\boldsymbol{u},\boldsymbol{u}) - \frac{2}{|\mathcal{C}_j|} \sum_{\boldsymbol{v} \in \mathcal{C}_j} K_r(\boldsymbol{u},\boldsymbol{v}) + \frac{1}{|\mathcal{C}_j|^2} \sum_{\boldsymbol{v},\boldsymbol{w} \in \mathcal{C}_j} K_r(\boldsymbol{v},\boldsymbol{w}) \ . \tag{9}$$

However, it is unnecessary to compute the first term in (9), 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:

$$\mathcal{C}^* = \text{argmin}_j \ \frac{-2}{|\mathcal{C}_j|} \sum_{\boldsymbol{v} \in \mathcal{C}_j} K_r(\boldsymbol{u},\boldsymbol{v}) + \gamma_j \ , \tag{10}$$

where

$$\gamma_j = \frac{1}{|\mathcal{C}_j|^2} \sum_{\boldsymbol{v},\boldsymbol{w} \in \mathcal{C}_j} K_r(\boldsymbol{v},\boldsymbol{w}) \ . \tag{11}$$

B. Support Vector Machines

Based on the structural risk minimization principle from the statistical learning theory [9], 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.

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

$$\textit{minimize:} \quad \boldsymbol{V}(\mathbf{w}, b, \xi) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{l} \xi_i \tag{12}$$

$$\textit{subject to:} \quad y_i(\mathbf{w} \cdot \Phi(\boldsymbol{v}_i) + b) \geq 1 - \xi_i, \quad \xi_i \geq 0 \ , \tag{13}$$

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

TABLE I

17 LANGUAGES USED IN OUR EXPERIMENTS

Language	#Sent.	Language	#Sent.
albanian	415	romanian	251
bulgarian	561	russian	980
croatian	207	serbian	275
czech	763	slovak	355
greek	278	slovene	809
hungarian	726	spanish	925
macedonian	208	turkish	751
polish	655	ukrainian	801
portuguese	874		

In practice, we work with its dual form instead of directly solving the primal problem by 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 $\mathbf{w}$ 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:

$$f(\boldsymbol{u}) = sgn\Big(\sum_i \alpha_i K_r(\boldsymbol{v}_i, \boldsymbol{u}) + b\Big) \ . \tag{14}$$

IV. EXPERIMENTS

A. Data Set and Experimental Setting

We collected a set of news articles taken from BBC World Service.[1] In preprocessing, we extracted only bodies of pages by removing all html tags, headers and footers. We then parsed them into sentences in order to form training and test data. Table I shows 17 considered languages and numbers of text samples (sentences), covering Western, Central, and Eastern European languages.

For the purpose of comparison, we used `TextCat` that is an implementation of the algorithm presented in [1] as the baseline. We used `LIBSVM` [2] as the core engine for running SVMs, and adapted an ANSI C implementation of a suffix tree[2] that can perform the Ukkonen's algorithm as the routine for kernel computation. In our experiments, we fixed the maximum substring length $r = 5$, and the weight $\lambda = 1$. During processing, we ignored the coding systems and decoded the samples as one-byte streams. To reduce the bias of the string length, we normalized the kernel function by:

$$\tilde{K}_r(\boldsymbol{u},\boldsymbol{v}) = \frac{\langle \Phi(\boldsymbol{u}), \Phi(\boldsymbol{v}) \rangle}{\|\Phi(\boldsymbol{u})\|\|\Phi(\boldsymbol{v})\|} \tag{15}$$

$$= \frac{K_r(\boldsymbol{u},\boldsymbol{v})}{\sqrt{K_r(\boldsymbol{u},\boldsymbol{u})K_r(\boldsymbol{v},\boldsymbol{v})}} \ . \tag{16}$$

We used 5-fold cross validation method to evaluate algorithms. We conducted five trials in which we trained classifiers using data from four parts, and test the classifiers using data from the remaining. All experimental results were averaged over five trials.

[1]All news articles were downloaded from `http://www.bcc.co.uk/worldservice/us/languages.shtml`.

[2]This software is publicly available at `http://cs.haifa.ac.il/~shlomo/suffix_tree`.

TABLE II

ACCURACY ON THE 17-LANGUAGES DATA SET

	`TextCat`	K. Centroid	SVM
0	0.898	0.955	0.995
1	0.899	0.960	0.999
2	0.919	0.966	0.998
3	0.894	0.953	0.997
4	0.898	0.961	0.996
Avg.	**0.902**	**0.959**	**0.997**

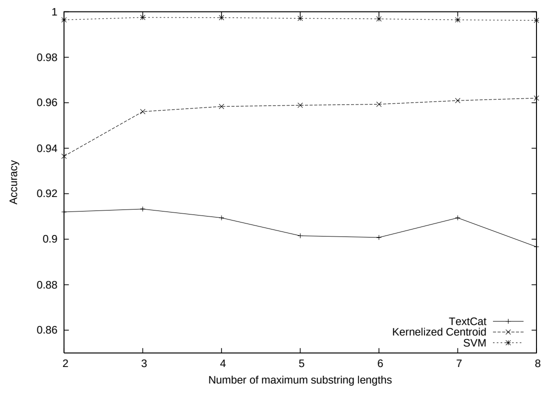

Fig. 1. Effect of the maximum length of the substrings.

B. Preliminary Results

In Table II, we present the accuracy results on the 17-languages data set, comparing with `TextCat` and our kernel classifiers. We can see that both kernel classifiers perform relatively better than simple n-grams comparison performed by `TextCat`. The average accuracy for `TextCat` is 0.902, while the kernelized centroid-based classifier gives an improved result at 0.959. The SVM classifier, which is based on a more powerful statistical background, achieves the best result at 0.997.

We also investigate the effect of the maximum length of all possible substrings. The objective is to observe how the length of substrings can improve or hurt the performance. We plot results in Figure 1, where the maximum length of substrings is varied in the range [2, 8]. From the curves, we can see that the accuracy results of the SVM classifier do not significantly change over the entire range. This indicates that the SVM classifier is more stable than other approaches. As the maximum length of substrings increases, the kernelized centroid-based classifier tends to slightly improve the performance of classification, while `TextCat` begins to decrease the performance after the maximum substring length 3.

V. Conclusion and Future Work

We have presented an alternative method for automatic language identification of written texts based on the idea of the string kernel. First, we introduce the idea of the string kernel for contiguous substrings that theoretically corresponds to the concept of n-gram models for sequence classification. Then, two kernel classifiers are described. The classifiers can combine the string kernel as the core module for computing the similarity between two texts. Finally, we provide experimental results, which are very encouraging.

In future work, we plan to conduct experiments at a larger scale in terms of the number of languages. Also, introducing weights to matched substrings may help to improve discriminating among samples. We will study the effect of some weighting techniques for scoring the matched substrings in the kernel computation.

References

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

[2] C.-C. Chang and C.-J. Lin. LIBSVM: a library for support vector machines (version 2.71). The software is publicly available at: `http://www.csie.ntu.edu.tw/~cjlin/libsvm`. 2004.

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

[4] R. Herbrich. *Learning Kernel Classifiers: Theory and Algorithms*. The MIT Press, 2002.

[5] G. Kikui. Identifying the coding system and language of on-line documents on the internet. In *Proceedings of the 16th conference on Computational linguistics*, pages 652–657, 1996.

[6] R. D. Lins and P. Gonçalves Jr. Automatic language identification of written texts. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1128–1133, 2004.

[7] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.

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

[9] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, Berlin, 1995.

[10] S. V. N. Vishwanathan and A. J. Smola. Fast kernels on strings and trees. In *Proceedings of Neural Information Processing Systems*, 2002.