1 Fast Kernels for String and Tree Matching

S.V.N. Vishwanathan
Machine Learning Program
National ICT Australia
Canberra, ACT 0200, Australia
vishy@axiom.anu.edu.au

Alexander Johannes Smola
Machine Learning Group, RSISE
The Australian National University
Canberra, ACT 0200, Australia
Alex.Smola@anu.edu.au

In this chapter we present a new algorithm suitable for matching discrete objects such as strings and trees in linear time, thus obviating dynamic programming with quadratic time complexity.

This algorithm can be extended in various ways to provide linear time prediction cost in the length of the sequence to be classified. We demonstrate extensions in the case of position dependent weights, sliding window classifiers for a long sequence, and efficient algorithms for dealing with weights given in the form of dictionaries. This improvement on the currently available algorithms makes string kernels a viable alternative for the practitioner.

1.1 Introduction

Many problems in machine learning require a data classification algorithm to work with a set of discrete objects. Common examples include biological sequence analysis where data is represented as strings (Durbin et al., 1998) and Natural Language Processing (NLP) where the data is given in the form of a string combined with a parse tree (Collins and Duffy, 2001) or an annotated sequence (Altun et al., 2003).

In order to apply kernel methods one defines a measure of similarity between discrete structures via a feature map $\phi : \mathcal{X} \to \mathcal{F}$. Here $\mathcal{X}$ is the set of discrete structures (eg. the set of all parse trees of a language) and $\mathcal{F}$ is a Hilbert space. Since $\phi(x) \in \mathcal{F}$ we can define a kernel by evaluating the scalar products

$$k(x, x') = \langle \phi(x), \phi(x') \rangle \tag{1.1}$$

where $x, x' \in \mathcal{X}$. The success of a kernel method employing k depends both on the *faithful representation* of discrete data and an *efficient means of computing k*.

Recent research effort has focussed on defining meaningful kernels on strings. Many ideas based on use of substrings (Herbrich, 2002), gapped substrings (Lodhi et al., 2002), k-length substrings (Leslie et al., 2002a) and mismatch penalties (Leslie et al., 2002b) have been proposed. This chapter presents a means of computing substring kernels on strings (Herbrich, 2002Leslie et al., 2002aJoachims, 2002) and trees in *linear time* in the size of the arguments, independent of the weights associated with any of the matching subwords. We also present a linear time algorithm for prediction which is independent of the number of support vectors. This is a significant improvement, since the so-far fastest methods rely on dynamic programming which incurs a quadratic cost in the length of the argument (Herbrich, 2002) or are additionally linear in the length of the matching substring (Leslie et al., 2002b). Further extensions to finite state machines, formal languages, automata, etc. can be found in (Vishwanathan, 2002Smola and Vishwanathan, 2003Cortes et al., 2002). Other means of generating kernels via the underlying correlation structure can be found in (Smola and Hofmann, 2003Takimoto and Warmuth, 1999).

In a nutshell our idea works as follows: assume we have a kernel $k(x, x') := \sum_{i \in I} \phi_i(x)\phi_i(x')$, where the index set I may be large, yet the number of nonzero entries is small in comparison to $|I|$ or the terms $\phi_i(x)$ have special structure. Then an efficient way of computing k is to sort the set of nonzero entries of $\phi(x)$ and $\phi(x')$ beforehand and count only matching nonzeros.

Sparse vectors

This is similar to the dot-product of sparse vectors in numerical analysis. As long as the sorting is done in an intelligent manner, the cost of computing k is linear in the sum of nonzeros entries combined. In order to use this idea for matching strings (which have a quadratically increasing number of substrings) and trees (which can be transformed into strings) efficient sorting is realized by the compression of the set of all substrings into a suffix tree. Moreover, dictionary keeping allows us to use (almost) arbitrary weightings for each of the substrings and still compute the kernels in linear time.

Our results improve on the algorithm proposed by Leslie et al. (2002b) in the case of *exact* matches, as the algorithm is now independent of the length of the matches and furthermore we have complete freedom in choosing the weight parameters. For inexact matches, unfortunately, such modifications are (still) not possible and we suggest the online-construction of Leslie et al. (2002b) for an efficient implementation.

Outline of the Chapter In section 1.2 we give the basic definition for the string and tree kernels used in this chapter. Section 1.3 contains the main result of the paper, namely how suffix trees can be used compute string kernels efficiently. The following two sections deal with the issue of computing weights efficiently and how to use the newly established algorithms for prediction purposes while keeping the linear-time property. Experimental results in Section 1.6 and a discussion (Section 1.7) conclude.

1.2 Kernels

1.2.1 String Kernels

We begin by introducing some notation. Let $\mathcal{A}$ be a finite set which we call the *alphabet*, e.g. $\mathcal{A} = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$. The elements of $\mathcal{A}$ are *characters*. Let \$ be a sentinel character such that $\$ \notin \mathcal{A}$. Any $x \in \mathcal{A}^k$ for $k = 0, 1, 2 \ldots$ is called a *string*. The empty string is denoted by ϵ and $\mathcal{A}^*$ represents the set of all non empty strings — the Kleene closure (Hopcroft and Ullman, 1979) — defined over the alphabet $\mathcal{A}$.

In the following we will use $s, t, u, v, w, x, y, z \in \mathcal{A}^*$ to denote strings and $a, b, c \in \mathcal{A}$ to denote characters. $|x|$ denotes the length of x, $uv \in \mathcal{A}^*$ the concatenation of two strings u and v, $au \in \mathcal{A}^*$ the concatenation of a character and a string. We use $x[i:j]$ with $1 \leq i \leq j \leq |x|$ to denote the substring of x between locations i and j (both inclusive). If $x = uvw$ for some (possibly empty) u, v, w, then u is called a *prefix* of x while v is called a substring (also denoted by $v \sqsubseteq x$) and w is called a *suffix* of x. Finally, $\mathtt{num}_y(x)$ denotes the number of occurrences of y in x (that is the number of times y occurs as a substring of x). The type of kernels we will be studying are defined by

String kernel definition

$$k(x, x') := \sum_{s \sqsubseteq x, s' \sqsubseteq x'} w_s \delta_{s,s'} = \sum_{s \in \mathcal{A}^*} \mathtt{num}_s(x)\, \mathtt{num}_s(x') w_s. \tag{1.2}$$

That is, we count the number of occurrences of every substring s in both x and x' and weight it by w_s, where the latter may be a weight chosen *a priori* or after seeing data, e.g., inverse document frequency counting (Leopold and Kindermann, 2002). This includes a large number of special cases:

- Setting $w_s = 0$ for all $|s| > 1$ yields the bag-of-characters kernel, counting simply single characters (Joachims, 2002).
- The bag-of-words kernel is generated by requiring s to be bounded by whitespace, i.e. $w_s \neq 0$ if and only if s is bounded by a whitespace character on either side (Joachims, 2002).
- Setting $w_s = 0$ for all $|s| > n$ yields limited range correlations of length n that is we only consider the contributions due to substrings of length n or less.
- The *k-spectrum* kernel takes into account substrings of length k (Leslie et al.,

2002a). It is achieved by setting $w_s = 0$ for all $|s| \neq k$.

- TFIDF weights (Salton, 1989) are achieved by first creating a (compressed) list of all s including frequencies of occurrence, and subsequently rescaling w_s accordingly.

All these kernels can be computed efficiently via the construction of suffix-trees, as we will see in the following sections.

However, before we go about computing $k(x, x')$ let us turn to trees. The latter are important for two reasons: first since the *suffix tree* representation of a string will be used to compute kernels efficiently, and secondly, since we may wish to compute kernels on trees, which will be carried out by reducing trees to strings and then applying a string kernel.

1.2.2 Tree Kernels

A tree is defined as a simple, directed, connected graph with no cycles. A *rooted tree* has a single special node called the root. An *internal node* has one or more child nodes and is called the *parent* of its children. The *root* is a node with no parent. A node with no children is referred to as a *leaf.* A sequence of nodes $n_1, n_2, \ldots, n_k$, such that n_i is the parent of n_{i+1} for $i = 1, 2, \ldots, k-1$ is called a *path.* Given two nodes a and d, if there is a path from node a to d then a is called an *ancestor* of d and d is called a *descendent* of a. We define a *subtree* as a node in the tree together with all its descendents. A subtree rooted at node n is denoted as T_n and $t \models T$ is used to indicate that t is a subtree of T. An *ordered tree* is a rooted tree in which the order of the subtrees (hanging from every node) is significant. From this point on all trees (and subtrees) we consider are ordered trees. If a set of nodes in the tree along with the corresponding edges forms a tree then we define it to be a *subset tree.*

If every node n of the tree contains a label, denoted by `label`(n), then the tree is called an *labeled* tree. If only the leaf nodes contain labels then the tree is called an *leaf-labeled* tree. Kernels on trees can be defined by defining kernels on matching subset trees as proposed by Collins and Duffy (2001) or (more restrictively) by defining kernels on matching subtrees. In the latter case we have

Tree kernel definition

$$k(T, T') = \sum_{t \models T, t' \models T'} w_t \delta_{t,t'}. \tag{1.3}$$

where $\delta_{t,t'} = 1$ iff the ordered subtrees t and t' are isomorphic and have the same order of child nodes at every level. We can compute our (more restrictive) tree kernels in linear time but if we consider subset tree as in Collins and Duffy (2001) quadratic-time algorithms are required.

If a tree $\tilde{T}$ can be obtained from a tree T by swapping the order of child nodes then they are said to be *equivalent.* Alternatively, equivalent trees are obtained by permuting the order of the leaf nodes without disturbing any parent child relationship in the tree (see Figure 1.1). The tree kernel as defined in (1.3) does not take this equivalence into account. While this may be desirable in

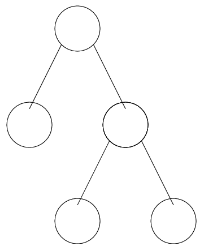
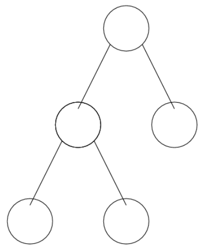

Figure 1.1 Two equivalent trees, which can be transformed into each other by swapping the children of the root node.

many applications (where ordered trees are naturally used), there are domains, for instance phylogenetic trees in bioinformatics, where it is desirable to reduce trees to a canonical form before computing kernels. We achieve that by implicitly sorting (or ordering) the trees.

1.2.3 Ordering Trees

To order trees we assume that a lexicographic order is associated with the labels if they exist. Furthermore, we assume that the additional symbols '$['$, '$]'$ satisfy '$['<$ '$]'$, and that '$]'$, '$['<$ $\texttt{label}(n)$ for all labels. We will use these symbols to define tags for each node as follows:

- For an unlabeled leaf n define $\texttt{tag}(n) := \texttt{[]}$.
- For a labeled leaf n define $\texttt{tag}(n) := \texttt{[}\,\texttt{label}(n)\texttt{]}$.
- For an unlabeled node n with children $n_1, \ldots, n_c$ define a lexicographically sorted permutation π of the child nodes such that $\texttt{tag}(n_{\pi(i)}) \leq \texttt{tag}(n_{\pi(j)})$ if $\pi(i) < \pi(j)$ and define

$$\texttt{tag}(n) = \texttt{[}\,\texttt{tag}(n_{\pi(1)})\,\texttt{tag}(n_{\pi(2)})\ldots\texttt{tag}(n_{\pi(c)})\texttt{]}.$$

Tree to string conversion

- For a labeled node perform the same operations as above and set

$$\texttt{tag}(n) = \texttt{[}\,\texttt{label}(n)\,\texttt{tag}(n_{\pi(1)})\,\texttt{tag}(n_{\pi(2)})\ldots\texttt{tag}(n_{\pi(c)})\texttt{]}.$$

For instance, the root nodes of both trees depicted in Figure 1.1 would be encoded as [[[][]][]]. We now prove that the tag of the root node, indeed, is a unique identifier and that it can be constructed in log linear time.

Theorem 1.1
Denote by T a binary tree with l nodes and let λ be the maximum length of a label. Then the following properties hold for the tag of the root node:

1. $\texttt{tag}(\texttt{root})$ can be computed in $(\lambda + 2)(l \log_2 l)$ time and linear storage in l.

2. Every substring s of $\texttt{tag}(\texttt{root})$ starting with $'['$ and ending with a balanced $']'$ has a one-to-one mapping with a subtree T_n of T where s is the tag on node n.

3. If trees T and $\tilde{T}$ are equivalent then their $\texttt{tag}(\texttt{root})$ is the same. Furthermore $\texttt{tag}(\texttt{root})$ allows the reconstruction of a unique element of the equivalence class.

Proof **Claim 1** We use induction. The tag of a leaf can be constructed in constant time by storing [,], and a pointer to the label of the leaf (if it exists), that is in 3 operations. Next assume that we are at node n, with children n_1, n_2. Let T_n contain l_n nodes and T_{n_1} and T_{n_2} contain l_1, l_2 nodes respectively. By our induction assumption we can construct the tag for n_1 and n_2 in $(\lambda+2)(l_1 \log_2 l_1)$ and $(\lambda+2)(l_2 \log_2 l_2)$ time respectively. Comparing the tags of n_1 and n_2 costs at most $(\lambda+2)\min(l_1, l_2)$ operations and the tag itself can be constructed in constant time and linear space by manipulating pointers. Without loss of generality we assume that $l_1 \leq l_2$. Thus, the time required to construct $\texttt{tag}(n)$ (normalized by $\lambda+2$) is

$$l_1(\log_2 l_1 + 1) + l_2 \log_2(l_2) = l_1 \log_2(2l_1) + l_2 \log_2(l_2) \leq l_n \log_2(l_n). \tag{1.4}$$

Claim 2 Without loss of generality we consider a labeled tree and use induction to show that every subtree of T is associated with a balanced substring of $\texttt{tag}(\texttt{root})$. For a leaf node n we assign $\texttt{tag}(n) = [\texttt{label}(n)]$ and it is clear that it corresponds to a balanced substring of $\texttt{tag}(\texttt{root})$. Suppose we are at an internal node n with two child nodes n_1 and n_2. Furthermore assume that the subtrees T_{n_1} and T_{n_2} correspond to balanced substrings $\texttt{tag}(n_1)$ and $\texttt{tag}(n_2)$ respectively. Since we assign $\texttt{tag}(n) = [\texttt{label}(n)\,\texttt{tag}(n_1)\,\texttt{tag}(n_2)]$ it is clear that the substring corresponding to T_n is balanced.

Conversely, by our recursive definition, every balanced substring of $\texttt{tag}(\texttt{root})$ must be of the form $\texttt{[}\,\texttt{label}(n)\,\texttt{tag}(n_1)\,\texttt{tag}(n_2)\texttt{]}$ for some node n with children n_1 and n_2. But this precisely corresponds to the tag on node n and hence to the subtree T_n. This proves claim 2.

Another way of visualizing our ordering is by imagining that we perform a DFS (depth first search) on the tree T and emit a $'['$ followed by $\texttt{label}(n)$, when we visit a node n for the first time and a $']'$ when we leave a node for the last time. It is clear that a *balanced* substring s of $\texttt{tag}(\texttt{root})$ is emitted only when the corresponding DFS on T_n is completed.

Claim 3 Since leaf nodes do not have children their tag is clearly invariant under permutation. For an internal node we perform lexicographic sorting on the tags of its children. This sorting maps all the trees of the equivalence class into a canonical representative. We then define the $\texttt{tag}(\texttt{root})$ of this tree as the string corresponding to our given tree. This directly proves the first part of our claim.

Concerning the reconstruction, we proceed as follows: each tag of a subtree starts with '$['$ and ends in a balanced '$]'$, hence we can strip the first [] pair from the tag, take whatever is left outside brackets as the label of the root node, and repeat the procedure with the balanced $[\ldots]$ entries for the children of the root node. This will construct a tree with the same tag as $\texttt{tag}(\texttt{root})$, thus proving our claim. ■

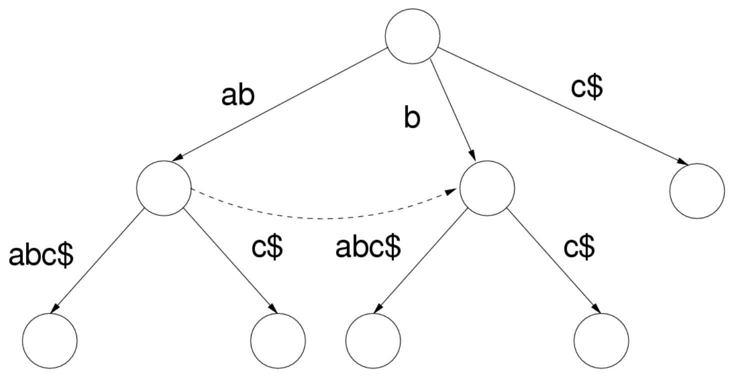

Figure 1.2 Suffix Tree of `ababc`. The leaf containing $ hanging off the root node has been omitted for readability.

An extension to trees with d nodes is straightforward (the cost increases to $d \log_2 d$ of the original cost due to additional comparisons required in the sorting of the leaves), yet the proof, in particular (1.4) becomes more technical without providing additional insight, hence we omit this generalization for brevity.

The above tree to string conversion algorithm along with Claim 2 of Theorem 1.1 yields the following straightforward corollary, whose proof is left as a simple exercise for the reader.

Corollary 1.2
Given trees T, T' the subtree matching kernel defined in (1.3) can be computed via string kernels, if we use the strings $\texttt{tag}(T)$ and $\texttt{tag}(T')$ and require that only balanced substrings s of the form $[\ldots]$ have nonzero weight w_s.

This reduces the problem of tree kernels to string kernels and all we need to show in the following is how the latter can be computed efficiently. For this purpose we need to introduce suffix trees.

1.3 Suffix Trees and Matching Statistics

1.3.1 Definition

The suffix tree is a compacted trie (Knuth, 1998) that stores all suffixes of a given text string (Weiner, 1973). We denote the suffix tree of the string x by $S(x)$. Moreover, let $\texttt{nodes}(S(x))$ be the set of all nodes of $S(x)$ and let $\texttt{root}(S(x))$ be the root of $S(x)$. If w denotes the path from the root to a node we label the node as $\overline{w}$. For a node $\overline{w}$, $T_{\overline{w}}$ denotes the subtree tree rooted at the node, $\texttt{lvs}(\overline{w})$ denotes the number of leaves in $T_{\overline{w}}$ and $\texttt{parent}(\overline{w})$ denotes it parent node.

We denote by $\texttt{words}(S(x))$ the set of all non-empty strings w such that $\overline{wu} \in \texttt{nodes}(S(x))$ for some (possibly empty) string u, which means that $\texttt{words}(S(x))$

is the set of all possible substrings of x (Giegerich and Kurtz, 1997). For every $t \in \texttt{words}(S(x))$ we define $\texttt{ceil}(t)$ as the node $\overline{w}$ such that $w = tu$ and u is the shortest (possibly empty) substring such that $\overline{w} \in \texttt{nodes}(S(x))$. That is, it is the immediate next node on the path leading up to t in $S(x)$. Finally, for every $t \in \texttt{words}(S(x))$ we define $\texttt{floor}(t)$ as the node $\overline{w}$ such that $t = wu$ and u is the shortest non-empty substring such that $\overline{w} \in \texttt{nodes}(S(x))$. That is, it is the last node encountered on the path leading up to t in $S(x)$. For an internal node $\overline{w}$ it is clear that $\texttt{ceil}(w) = \overline{w}$ and $\texttt{floor}(w)$ is the parent of $\overline{w}$. Given a string t and a suffix tree $S(x)$, we can decide if $t \in \texttt{words}(S(x))$ in $O(|t|)$ time by just walking down the corresponding edges of $S(x)$ (Weiner, 1973).

If the sentinel character \$ is added to the string x then it can be shown that for any $t \in \texttt{words}(S(x))$, $\texttt{lvs}(\texttt{ceil}(t))$ gives us the number of occurrence of t in x (Giegerich and Kurtz, 1997). The idea works as follows: all suffixes of x starting with t have to pass through $\texttt{ceil}(t)$, hence we simply have to count the occurrences of the sentinel character, which can be found only in the leaves. Note that a simple DFS on $S(x)$ will enable us to calculate $\texttt{lvs}(\overline{w})$ for each node $\overline{w}$ of $S(x)$ in $O(|x|)$ time and space.

Let $\overline{aw}$ be a node in $S(x)$, and v be the longest suffix of w such that $\overline{v} \in \texttt{nodes}(S(x))$. An unlabeled edge $\overline{aw} \to \overline{v}$ is called a suffix link in $S(x)$. A suffix link of the form $\overline{aw} \to \overline{w}$ is called *atomic*. It can be shown that all the suffix links in a suffix tree are atomic (Giegerich and Kurtz, 1997, Proposition 2.9). We add suffix links to $S(x)$, to allow us to perform efficient string matching: suppose we found that aw is a substring of x by parsing the suffix tree $S(x)$. It is clear that w is also a substring of x. If we want to locate the node corresponding to w, it would be wasteful to parse the tree again. Suffix links can help us locate this node in constant time. The suffix tree building algorithms (for instance Ukkonen (1995)) make use of this property of suffix links and construct the suffix tree and all its suffix links in *linear time*.

Note that suffix links are just a special case of so-called *failure functions* from automata theory: there they allow one to backtrack gracefully and re-use some of the information obtained in checking whether a string is accepted by an automaton (Hopcroft and Ullman, 1979Cortes et al., 2002).

1.3.2 Matching Statistics

Given strings x, y with $|x| = n$ and $|y| = m$, the matching statistics of x with respect to y are given by vectors v, c and c' with $v_i \in \mathbb{N}$ and $c_i, c'_i \in \texttt{nodes}(S(y))$ for $i = 1, 2, \ldots, n$. We define v_i as the length of the longest substring of y matching a prefix of $x[i : n]$, $\overline{v_i} := i + v_i - 1$, while c_i is the node corresponding to $\texttt{ceil}(x[i : \overline{v_i}])$ and c'_i is the node corresponding to $\texttt{floor}(x[i : \overline{v_i}])$ in $S(y)$. For an example see Table 1.1.

For a given x one can construct the matching statistics corresponding to y in linear time. The key observation is that $v_{i+1} \geq v_i - 1$, since if $x[i : \overline{v_i}]$ is a substring of y then definitely $x[i + 1 : \overline{v_i}]$ is also a substring of y. Besides this, the matching

String	b	c	b	a	b
v_i	2	1	3	2	1
$x[i : \overline{v_i}]$	bc	c	bab	ab	b
c_i	bc$	c$	babc$	ab	b
c'_i	b	root	b	root	root

Table 1.1 Matching statistic of `bcbab` with respect to $S($`ababc`$)$ are shown here along with the matching substrings.

substring in y that we find, *must* have $x[i+1 : \overline{v_i}]$ as a prefix. The Matching Statistics algorithm of Chang and Lawler (1994) exploits this observation and uses it to cleverly walk down the suffix links of $S(y)$ in order to compute the matching statistics in $O(|x|)$ time.

More specifically, given c'_i the algorithm finds the intermediate node $p'_{i+1} :=$ `floor`$(x[i+1 : \overline{v_i}])$ by first walking down the suffix link of c'_i and then walking down the edges corresponding to the remaining portion of $x[i+1 : \overline{v_i}]$ until it reaches p'_{i+1}. Now c_{i+1}, c'_{i+1} and v_{i+1} can be found easily by walking from p'_{i+1} along the edges of $S(y)$ that match $x[\overline{v_i}+1 : n]$, until we can go no further. The value of v_1 is found by simply walking down $S(y)$ to find the longest prefix of x which matches a substring of y.

1.3.3 Matching Substrings

Using the matching statistics we can read off the number of matching substrings in x and y. The useful observation here is that the only substrings which occur in both x and y are those which are prefixes of $x[i : \overline{v_i}]$. The number of occurrences of a substring in y can be found by `lvs`(`ceil`(w)), as discussed in Section 1.3.1. The two lemmas below formalize this.

Lemma 1.3
w is a substring of x iff there is an i such that w is a prefix of $x[i : n]$. The number of occurrences of w in x can be calculated by finding all such i.

Proof Let w be a substring of x. For each occurrence of w in x we can write $w = x[i : j]$ for some unique indices i and j. Clearly $x[i : j]$ is a prefix of $x[i : n]$. Conversely, for every index i, every prefix of $x[i : n]$ is a substring of x. ■

Lemma 1.4
The set of matching substrings of x and y is the set of all prefixes of $x[i : \overline{v_i}]$.

Proof Let w be a substring of both x and y. By above lemma there is an i such that w is a prefix of $x[i : n]$. Since v_i is the length of the maximal prefix of $x[i : n]$ which is a substring in y, it follows that $v_i \geq |w|$. Hence w must be a prefix of $x[i : \overline{v_i}]$.

Conversely, at any position i the longest prefix of $x[i:n]$ which matches some substring of y is $x[i:\overline{v_i}]$. Hence it follows that every prefix of $x[i:\overline{v_i}]$ is a substring of both x and y. ■

1.3.4 Efficient Kernel Computation

From the previous discussion we know how to determine the set of all longest prefixes $x[i:\overline{v_i}]$ of $x[i:n]$ in y in linear time. The following theorem uses this information to compute kernels efficiently.

Theorem 1.5
Let x and y be strings such that v, c and c' be the matching statistics of x with respect to y. Assume that

$$W(y,t) = \sum_{p \in \texttt{prefix}(v)} w_{up} - w_u \text{ where } \overline{u} = \texttt{floor}(t) \text{ in } S(y) \text{ and } t = uv. \tag{1.5}$$

Fast kernel computation

can be computed in constant time for any t. Then $k(x,y)$ defined in (1.2) can be computed in $O(|x|+|y|)$ time as

$$k(x,y) = \sum_{i=1}^{|x|} [\texttt{val}(c_i') + \texttt{lvs}(c_i) \cdot W(y, x[i:\overline{v_i}])] \tag{1.6}$$

where $\texttt{val}(\overline{t}) := \texttt{val}(\texttt{parent}(t)) + \texttt{lvs}(\overline{t}) \cdot W(y,t)$ and $\texttt{val}(\texttt{root}) := 0$.

Proof We first show that (1.6) can indeed be computed in linear time. We know that for $S(y)$ the number of leaves per node can be computed in $O(|y|)$ time by performing a DFS. Also, v, c and c' can be computed in $O(|x|)$ time by a invocation of the matching statistics algorithm. By assumption on $W(y,t)$ and by exploiting the recursive nature of $\texttt{val}(\overline{t})$ we can pre-compute $W(y,w)$ for all $\overline{w} \in \texttt{nodes}(S(y))$ by using simple top down procedure in $O(|y|)$ time.

Now we may compute each term in constant time by a simple lookup for $\texttt{val}(c_i')$ and $\texttt{lvs}(c_i)$ and by computing $W(y, x[i:\overline{v_i}])$ in constant time. Since we have $|x|$ terms, the whole procedure takes $O(|x|)$ time, which proves the $O(|x|+|y|)$ time complexity.

Now we prove that (1.6) really computes the kernel. We know from Lemma 1.4 that the sum in (1.2) can be decomposed into the sum over matches between y and each of the prefixes of $x[i:\overline{v_i}]$ (this takes care of all the substrings in x matching with y). This reduces the problem to showing that each term in the sum of (1.6) corresponds to the contribution of all prefixes of $x[i:\overline{v_i}]$.

The key observation here is that all substrings of y which share the same `ceil` node occur the same number of times in y. This allows us to bracket the contribution due to each of the prefixes of $x[i:\overline{v_i}]$ efficiently. Consider our previous example with x = `bcbab` and y = `ababc`. To compute `val(bab)` we need to consider the contributions due to `bab`, `ba` as well as `b`. Looking at $S(y)$ immediately tells us that `b` occurs twice in y (because $\texttt{lvs}(\texttt{ceil}(\texttt{b})) = 2$) and hence its contribution must be counted twice, while ba and bab occur only once in y and hence their contributions must be counted only once.

Because of its recursive definition and by exploiting the above observation it is clear that $\texttt{val}(\overline{w})$ for each $\overline{w} \in \texttt{nodes}(S(y))$ computes the contribution to the kernel due to w and *all* its prefixes. Given an arbitrary substring t such that $\overline{u} = \texttt{floor}(t)$ and $t = uv$ there are two components of t which contribute to the kernel. One is the contribution due to u and all the prefixes and the other is the contribution due to all strings of the form us where $s \in \texttt{prefix}(v)$. Since each such substring occurs exactly $\texttt{lvs}(\texttt{ceil}(t))$ times in y we can perform efficient bracketing and use $W(y,t)$ to compute the kernel. ■

Clearly once $S(y)$ and the annotations of $S(y)$ have been computed, evaluating another $k(x,y)$ costs only $O(|x|)$ rather than $O(|x|+|y|)$ time. This suggests that for prediction we can gain extra efficiency by pre-computing a large part of the kernel expansion. This idea is made more explicit in Section 1.5.1. Before we do so, however, we need to address the issue under which conditions $W(y,t)$ can really be computed in constant time.

1.4 Weights and Kernels

1.4.1 Length Dependent Weights

If the weights w_s depend only on $|s|$ we have $w_s = w_{|s|}$. Define $\omega_j := \sum_{i=1}^{j} w_i$ and compute its values beforehand up to ω_J where $J \geq |x|$ for all x. Then the sum in $W(y,t)$ telescopes and it follows that

$$W(y,t) = \left[\sum_{j=|\texttt{floor}(t)|}^{|t|} w_j\right] - w_{|\texttt{floor}(t)|} = \sum_{j=1}^{|t|} w_j - \sum_{j=1}^{|\texttt{floor}(t)|} w_j = \omega_{|t|} - \omega_{|\texttt{floor}(t)|} \tag{1.7}$$

which can be computed in constant time by looking up the pre-computed values. Examples of such weighting schemes include decay factors $w_i = \lambda^{-i}$, or indicator functions $w_i = 1$, bounded range interactions $w_i = 1$ if $i \leq n$ and $w_i = 0$ otherwise, and character counts $w_i = \delta_{1i}$ (Joachims, 2002). Denote by $\tau := |t|$ and $\gamma := |\texttt{floor}(t)|$ the string boundaries. In this case we can compute $W(y,t)$ as follows

$$W(y,t) = \frac{(\lambda^{-\gamma} - \lambda^{-\tau})}{\lambda - 1} \qquad \text{Exponential decay} \tag{1.8}$$

$$W(y,t) = \tau - \gamma \qquad \text{Constant weight} \tag{1.9}$$

$$W(y,t) = \max(0, \min(\tau, n) - \gamma) \qquad \text{Bounded range} \tag{1.10}$$

$$W(y,t) = \begin{cases} 1 & \text{if } \gamma = 0 \\ 0 & \text{otherwise} \end{cases} \qquad \text{Bag of characters} \tag{1.11}$$

1.4.2 Position Dependent Weights

Next we study the case where the weights w_s depend on $|s|$ and the position of s relative to the beginning of the strings x, y. That is, $w_s = w_{|s|}\phi(|u_x|+1)\phi(|u_y|+1)$, where $x = u_x s v_x$ and $y = u_y s v_y$.

Note that all words of y beginning at the same position i, share the same weight, namely $\phi(i)$. They also share the same terminal leaf in $S(y)$ (recall that there is exactly one leaf per suffix). Hence it is sufficient to enumerate the leaves and weigh them based on the starting position of the suffix they correspond to. Recall that in section 1.3.1 we defined $\texttt{lvs}(\overline{w})$ as the number of leaves in $T_{\overline{w}}$. If we assume a unit weight placed on each of the leaves then $\texttt{lvs}(\overline{w})$ can be thought of as the sum of the weights of the leaves in $T_{\overline{w}}$. In the case of position dependent weights each leaf contains a possibly different weight. Hence we generalize the above notion to define $\texttt{lvs}(\overline{w})$ as the sum of the weights of the leaves in $T_{\overline{w}}$. Furthermore, to compute the kernel, (1.6) needs to be rewritten as follows:

$$k(x,y) = \sum_{i=1}^{|x|} \phi(i)\left[\texttt{val}(c_i') + \texttt{lvs}(c_i) \cdot W(y, x[i:\overline{v_i}])\right] \tag{1.12}$$

where $\texttt{val}(c_i)$ is as defined in theorem 1.5.

1.4.3 Dictionary Weights

Next assume that we have a set of strings $D = \{x_1, x_2, \ldots, x_k\}$ with corresponding weights w_{x_i} and that all other string-weights vanish, i.e. $w_d = 0$ for $d \notin D$. Such a method was recently put to good practical use by Ben-Hur and Brutlag (2003), who use a set of motifs as the features used in classification of remote homologies.

We now provide a fast version of the motif-kernel algorithm, which does not depend on the number of matching motifs or the number of support vectors any more, unlike (Ben-Hur and Brutlag, 2003, Figure 2), however with the assumption that the motifs *do not contain any wildcards.*

For the dictionary D we define $|D| = \sum_{i=1}^{k} |x_i|$. The suffix tree of D is denoted by $S(D)$ and is defined as the compacted trie obtained by inserting all the suffixes of all $x_i \in D$. By a clever modification of the construction algorithm of McCreight (1976) we can construct $S(D)$ in $O(|D|)$ time (Amir et al., 1994). Given a new string x_{k+1} we can insert it into $S(D)$ in $O(|x_{k+1}|)$ time. The suffix tree $S(D)$ contains a node corresponding to each of the strings in D. If none of the strings in D are substrings of each other then $S(D)$ contains a unique leaf node corresponding to each x_i. To ensure that none of the strings in D are substrings of each other we might need to append a new symbol $\$_i$ to each string x_i in the dictionary.

Given a text and a static or dynamic dictionary of patterns the problem of reporting all occurrence of any pattern of the dictionary in the text has been well studied (Aho and Corasick, 1975Amir et al., 1994). Here we concentrate on the static dictionary matching problem and sketch an algorithm which is close in spirit to our string kernel algorithm.

The basic idea of our algorithm is as follows: Given strings x and y we first construct $S(\{x, y\})$. We then show that to compute the kernel $k(x, y)$ it is sufficient to annotate the nodes of this joint suffix tree. We then show how the annotation can be performed by (conceptually) constructing the suffix tree $S(\{x, y\} \bigcup D)$.

We first prove the following technical lemma:

Lemma 1.6
Let x and y be strings such that v, c and c' be the matching statistics of x with respect to y. The suffix tree $S(\{x, y\})$ contains a node corresponding to $x[i : \overline{v_i}]$ for $i = 1 \ldots |x|$.

Proof By definition of matching statistics, $x[i : \overline{v_i}]$ occurs as a substring of y while $x[i : \overline{v_i} + 1]$ does not. This implies that in $S(\{x, y\})$ we can share the path corresponding to $x[i : \overline{v_i}]$ but to represent the string $x[i : \overline{v_i}+1]$ and its superstrings we will need to introduce a node. ■

Let $\overline{w} \in \texttt{nodes}(S(\{x, y\}))$, we define $\texttt{lvs}_x(\overline{w})$ and $\texttt{lvs}_y(\overline{w})$ as the number of times w occurs as a substring of x and y respectively. The following lemma provides an alternate view of our string kernel by characterizing it in terms of the nodes of $S(\{x, y\})$.

Lemma 1.7
Given the strings x and y the string kernel defined in (1.2) can be computed as

$$k(x, y) = \sum_{\overline{t} \in \texttt{nodes}(S(\{x,y\}))} \texttt{lvs}_x(\overline{t}) \cdot \texttt{lvs}_y(\overline{t}) \cdot W(\{x, y\}, t) \tag{1.13}$$

where $W(\{x, y\}, t)$ is defined as in Theorem 1.5

Proof We sketch the outline of the proof. We first observe that all strings which share a common `ceil` (say $\overline{t}$) in $S(\{x, y\})$ occur $\texttt{lvs}_x(\overline{t})$, $\texttt{lvs}_y(\overline{t})$ number of times in x and y respectively. From the previous lemma we know that $x[i : \overline{v_i}]$ occurs as a node in this joint suffix tree. These two observations allow us to bracket the terms efficiently to compute the kernel. ■

In order to compute the kernel $k(x, y)$ all that remains is to compute $W(\{x, y\}, t)$ efficiently for all nodes $\overline{t} \in \texttt{nodes}(S(\{x, y\}))$. Let $\overline{t}$ be a node in $S(\{x, y\})$ and $\overline{w}$ be its parent. In order to compute $W(\{x, y\}, t)$ we just need to sum up the weights of all strings from the dictionary D which end on the edge connecting $\overline{w}$ to $\overline{t}$. This can be easily computed by keeping track of the weights on the path leading from string w to string t in the suffix tree $S(D)$. In fact this computation can be performed in constant time if we annotate each node in $S(D)$ with the sum of the weights on the path from the root to the node.

A conceptually easy way to think of the above procedure is as follows: Assume that we construct $S(\{x, y\} \bigcup D)$ (this can be done in $O(|x| + |y|)$ time if $S(D)$ is already given (Amir et al., 1994)). Each $\overline{t} \in \texttt{nodes}(S(\{x, y\}))$ is also a node in

$S(\{x,y\} \bigcup D)$. For each such node t let $\overline{w}$ be its parent in $S(\{x,y\})$. We compute the weights on the path from $\overline{w}$ to $\bar{t}$ in $S(\{x,y\} \bigcup D)$. As before, this computation can be performed in constant time by annotating the nodes of $S(D)$ beforehand.

Wildcards Mismatches can be taken into account if we allow for a rather simple modification of the above algorithm (albeit at a somewhat higher runtime): treat wildcards as an additional (special) symbol and build the suffix tree based on the set of dictionary terms. When computing the matching statistics algorithm and parsing the suffix tree, allow for multiple paths, i.e. if a vertex contains contains a wildcard `'*'` and the symbol `'A'` in its children, then when observing `'A'` both paths would need to be followed. This means that the algorithm now is more expensive, exactly by the number of concurrent paths that need to be taken into account simultaneously.

Another approach to account for wildcards is to build a compact Finite State Automata (FSA) to represent the dictionary patterns (Navarro and Raffinot, 1999). This can be viewed as a generalization of suffix trees with failure functions.

1.4.4 Generic Weights

In case of generic weights, where individual subwords are assigned weights independent of each other, we proceed as in the dictionary approach. The main difference is that here the dictionary is only given implicitly. Hence denote by $X = \{x_1, \ldots, x_m\}$ the set of all arguments to be used for kernel evaluation purposes.

Compute w_s for all substrings of x_i and treat the result as a dictionary. This reduces the present case to the situation above, which may result in superlinear time to annotate $S(D)$, due to the significantly larger number of nonzero weights. However, this should not be surprising, since we need to read each of the weights used in computing the kernel at least once.

Our approach offers the option to assign weights according to the frequency of occurrence of strings in a text. For this purpose, build a suffix tree first, use the latter to obtain string frequency counts, then compute the according weights, and proceed by annotating the suffix tree. This allows one to implement kernels such as the ones proposed by Leopold and Kindermann (2002) efficiently.

1.5 Optimization and Prediction

Beyond the actual evaluation of scalar products, we can extend our techniques to compute *linear combinations* of kernel functions efficiently. Crucial in this context is the fact that the computation of $k(x,y)$ is *asymmetric* in x and y insofar as a suffix tree for y is computed, whereas x is merely compared to $S(y)$.

In the following we will extend this idea by manipulating the suffix tree of the set of support vectors directly, which will allow us to obtain $O(|x|)$ time algorithms even for combinations of kernels.

1.5.1 Linear Time Prediction

Let $\mathcal{X} = \{x_1, x_2, \ldots, x_m\}$ be the set of support vectors. Recall that for prediction in a Support Vector Machine we need to compute

$$f(x) = \sum_{i=1}^{m} \alpha_i y_i k(x_i, x), \tag{1.14}$$

which implies that we need to combine the contribution due to matching substrings from each one of the support vectors. Using the definition of the string kernel we can expand f as

$$f(x) = \sum_{i=1}^{m} \sum_{s \sqsubseteq x_i} \sum_{s' \sqsubseteq x} \alpha_i y_i w_s \delta_{s,s'}. \tag{1.15}$$

Previously each substring match contributed a weight of w_s to the kernel. Now substring s from support vector x_i contributes a weight of $\alpha_i y_i w_s$. This suggest a simple strategy akin to the one we used for position dependent weights.

We first construct $S(\mathcal{X})$ in linear time by using the algorithm of Amir et al. (1994). Next, each leaf in $S(\mathcal{X})$ arising from x_i, is assigned the weight $\alpha_i y_i$ rather than 1. All we need to do now is define at each node $\overline{w}$ an updated weight, given by

Weighted leaf nodes

$$\texttt{weight}(\overline{w}) = \sum_{i=1}^{m} \alpha_i y_i \, \texttt{lvs}_{x_i}(\overline{w}) \tag{1.16}$$

where $\texttt{lvs}_{x_i}(\overline{w})$ denotes the number of leaves of $S(x_i)$ at node $\texttt{ceil}(\overline{w})$.

This allows us to take the contribution of all SVs into account simultaneously. A straightforward application of the matching statistics algorithm of Chang and Lawler (1994) shows that we can find the matching statistics of x with respect to all strings in $\mathcal{X}$ in $O(|x|)$ time. Now (1.6) can be applied verbatim to compute $f(x)$.[1]

In summary, we can classify texts in linear time regardless of the size of the training set. This makes SVM for large-scale text categorization practically feasible. However, note that this ability comes at a cost: we have to store $S(\mathcal{X})$, which has memory requirements linear in the size of the support vectors.

1.5.2 Sliding Windows

Now assume that we want to classify subsequences of a long string, e.g. a DNA sequence d. The typical approach in this case is to take $d[i : i+n]$, where n specifies the window size, and compute $f(d[i : i+n])$ for all $i \leq |d| - n$ *individually* without taking the correlation between the function values $f(d[i : i+n])$ into account.

This is highly wasteful, since the difference between two adjoint strings, that is

1. See (Vishwanathan, 2002) for further details and a proof.

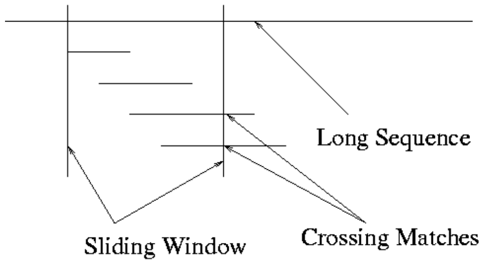

Figure 1.3 In a sliding window classifier we extend only those matches which cross the current window boundary.

$d[i : i + n]$ and $d[i + 1 : i + n + 1]$ is just given by $d[i]$ and $d[i + n + 1]$. All other symbols and their order are *identical*. This means that a large part of what has been computed for $d[i : n]$ can be reused for its successor $d[i + 1 : n + 1]$.

As before we merge the suffix trees of all support vectors into a master suffix tree denoted by $S(\mathcal{X})$. Next we compute the matching statistics of the entire long sequence x with respect to $S(\mathcal{X})$. Let c_i^j and v_i^j denote the matching statistics of the j^{th} character in the i^{th} window, $x[i : i + n]$. Since a match cannot extend beyond the boundary of the window it is clear that $v_i^j = \min(n - j, v_{i+j})$ while $c_i^j = \texttt{ceil}(x[i : \overline{v}_i^j]$. This means that we need not parse $S(\mathcal{X})$ again and again for computing the matching statistics of each window, which in turn will lead to implementation speedups.

An even more efficient strategy is to keep track of those strings whose matches end on the current window boundary. In the worst case there can be n of them, but, in general the number will be typically smaller than n. When we perform estimation for the next window we need to extend only those matches which ended on the previous window boundary. If the number of matches that we need to extend is less than n, then we can perform estimation for the next window in sub-linear time (that is less than $O(n)$ time). Figure 1.3 depicts this strategy.

In general, v_i denotes the length of the longest match at location i of string x. This implies that the match corresponding to $x[i : \overline{v_i}]$ will cross at most v_i window boundaries and hence will need to be extended exactly v_i times. Thus the total time required to compute the sliding window kernel is proportional to $\sum_i \min(v_i, n)$ which is typically much smaller than $n|x|$.

1.6 Experimental Results

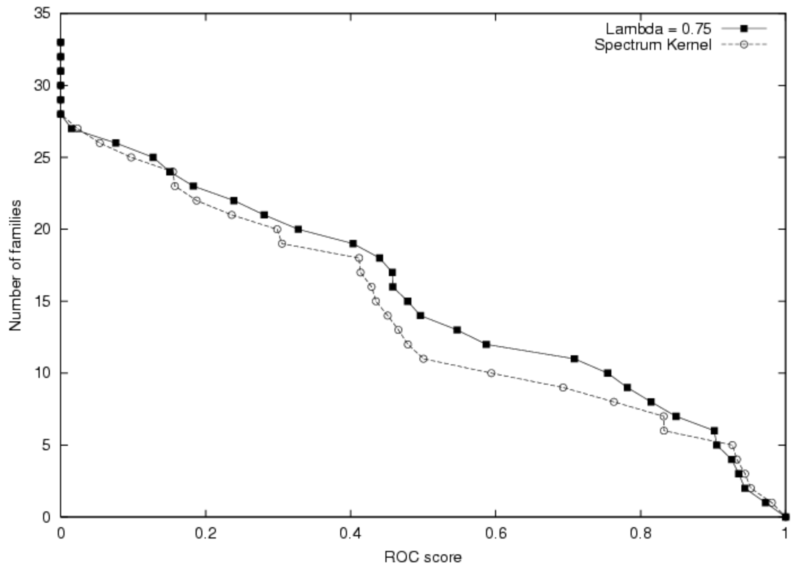

Figure 1.4 Total number of families for which an SVM classifier exceeds a ROC_{50} score threshold.

For a proof of concept we tested our approach on a remote homology detection problem[2] (Jaakkola et al., 2000) using Stafford Noble's SVM package[3] as the training algorithm. A length weighted kernel was used and we assigned weights $w_s = \lambda^{|s|}$ for all substring matches of length greater than 3 regardless of triplet boundaries. To evaluate performance we computed the ROC_{50} scores.[4]

Being a proof of concept, we did not try to tune the soft margin SVM parameters (the main point of the chapter being the introduction of a novel means of evaluating string kernels efficiently rather than applications - a separate paper focusing on applications is in preparation).

Figure 1.4 contains the ROC_{50} scores for the spectrum kernel with $k = 3$ (Leslie et al., 2002a) and our string kernel with $\lambda = 0.75$. We tested with $\lambda \in \{0.25, 0.5, 0.75, 0.9\}$ and report the best results here. As can be seen our kernel outperforms the spectrum kernel based on the total number of families for which a classifier based on our kernel exceeds a ROC_{50} score threshold.

It should be noted that this is the first method to allow users to specify weights

2. Details and data available at www.cse.ucsc.edu/research/compbio/discriminative.
3. Available at www.cs.columbia.edu/compbio/svm.
4. The ROC_{50} score (Gribskov and Robinson, 1996Leslie et al., 2002a) is the area under the receiver operating characteristic curve (the plot of true positives as a function of false positives) up to the first 50 false positives. A score of 1 indicates perfect separation of positives from negatives, whereas a score of 0 indicates that none of the top 50 sequences selected by the algorithm were positives.

rather arbitrarily for all possible lengths of matching sequences and still be able to compute kernels at $O(|x| + |x'|)$ time, plus, to predict on new sequences at $O(|x|)$ time, once the set of support vectors is established.[5]

1.7 Conclusion

We have shown that string kernels need not come at a super-linear cost in SVMs and that prediction can be carried out at cost linear only in the length of the argument, thus providing optimal run-time behavior. Furthermore the same algorithm can be applied to trees.

The methodology pointed out in this chapter has several immediate extensions: for instance, we may consider coarsening levels for trees by removing some of the leaves. For not too-unbalanced trees (we assume that the tree shrinks at least by a constant factor at each coarsening) computation of the kernel over all coarsening levels can then be carried out at cost still linear in the overall size of the tree. The idea of coarsening can be extended to approximate string matching. If we remove characters, this amounts to the use of wildcards.

Likewise, we can consider the strings generated by finite state machines and thereby compare the finite state machines themselves. This leads to kernels on automata and other dynamical systems. More details and extensions can be found in Vishwanathan (2002).

Acknowledgments

We would like to thank Asa Ben-Hur, Eleazar Eskin, Patrick Haffner, and Bob Williamson for comments and suggestions. We would like to thank Jean-Philippe Vert and Koji Tsuda for carefully going the draft and pointing out many errors and helping us improve the presentation significantly. This research was supported by a grant of the Australian Research Council. SVNV thanks Trivium India Software and Netscaler Inc. for their support.

5. Leslie et al. (2002a) obtain an $O(k|x|)$ algorithm in the (somewhat more restrictive) case of $w_s = \delta_k(|s|)$.

References

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

Y. Altun, I. Tsochantaridis, and T. Hofmann. Hidden markov support vector machines. In *International Conference of Machine Learning*, 2003. forthcoming.

A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. Dynamic dictionary matching. *Journal of Computer and System Science*, 49(2):208–222, October 1994.

A. Ben-Hur and D. Brutlag. Remote homology detection: A motif based approach. In *ISMB 2003*, 2003.

W. I. Chang and E. L. Lawler. Sublinear approximate sting matching and biological applications. *Algorithmica*, 12(4/5):327–344, 1994.

M. Collins and N. Duffy. Convolution kernels for natural language. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, Cambridge, MA, 2001. MIT Press.

C. Cortes, P. Haffner, and M. Mohri. Rational kernels. In *Proceedings of Neural Information Processing Systems 2002*, 2002. in press.

R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic models of proteins and nucleic acids.* Cambridge University Press, 1998.

R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

M. Gribskov and N. L. Robinson. Use of receiver operating characteristic (ROC) analysis to evaluate sequence matching. *Computers and Chemistry*, 20(1):25–33, 1996.

R. Herbrich. *Learning Kernel Classifiers: Theory and Algorithms.* MIT Press, 2002.

J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, Massachusetts, first edition, 1979.

T. S. Jaakkola, M. Diekhans, and D. Haussler. A discriminative framework for detecting remote protein homologies. *Journal of Computational Biology*, 7:95–114, 2000.

T. Joachims. *Learning to Classify Text Using Support Vector Machines: Methods, Theory, and Algorithms.* The Kluwer International Series In Engineering And

Computer Science. Kluwer Academic Publishers, Boston, May 2002. ISBN 0-7923-7679-X.

D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, Massachusetts, second edition, 1998.

E. Leopold and J. Kindermann. Text categorization with support vector machines: How to represent text in input space? *Machine Learning*, 46(3):423–444, March 2002.

C. Leslie, E. Eskin, and W. S. Noble. The spectrum kernel: A string kernel for SVM protein classification. In *Proceedings of the Pacific Symposium on Biocomputing*, pages 564–575, 2002a.

C. Leslie, E. Eskin, J. Weston, and W. S. Noble. Mismatch string kernels for SVM protein classification. In *Proceedings of Neural Information Processing Systems 2002*, 2002b. in press.

H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins. Text classification using string kernels. *Journal of Machine Learning Research*, 2:419–444, February 2002.

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

G. Navarro and M. Raffinot. Fast regular expression search. In *Proceedings of WAE*, number 1668 in LNCS, pages 198–212. Springer, 1999.

Gerard Salton. *Automatic Text Processing*. Addison-Wesley, Massachusetts, 1989.

A.J. Smola and T. Hofmann. Exponential families in feature space. In *Advances in Neural Information Processing Systems*, 2003. submitted.

A.J. Smola and S.V.N. Vishwanathan. Hilbert space embeddings in dynamical systems. In *Proceedings of the* 13th *IFAC symposium on system identification*. IFAC, August 2003. In press.

Eiji Takimoto and Manfred K. Warmuth. Predicting nearly as well as the best pruning of a planar decision graph. In *Proc. 10th International Conference on Algorithmic Learning Theory - ALT '99*, volume 1720 of *Lecture Notes in Artificial Intelligence*, pages 335–346. Springer-Verlag, 1999.

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

S. V. N. Vishwanathan. *Kernel Methods: Fast Algorithms and Real Life Applications*. PhD thesis, Indian Institute of Science, Bangalore, India, November 2002.

P. Weiner. Linear pattern matching algorithms. In *Proceedings of the IEEE* 14^{th} *Annual Symposium on Switching and Automata Theory*, pages 1–11, The University of Iowa, 1973. IEEE.