ADAC ADAC manuscript No.
(will be inserted by the editor)

Lambda Pruning: An Approximation of the String Subsequence Kernel for Practical SVM Classification and Redundancy Clustering

Alexander K. Seewald · Florian Kleedorfer

Abstract The Support Vector Machine (SVM) is a powerful learning algorithm, e.g. for classification and clustering tasks, that works even for complex data structures such as strings, trees, lists and general graphs. It is based on the usage of a kernel function for measuring scalar products between data units. For analyzing string data Lodhi et al. (2002) have introduced a String Subsequence Kernel (SSK). It this paper we propose an approximation to SSK based on dropping higher orders terms (i.e. subsequences which are spread out more) that reduces the computational burden of SSK. As we are also concerned with practical application of complex kernels with high computational complexity and memory consumption, we provide an empirical model to predict runtime and memory of the approximation as well as the original SSK, based on easily measurable properties of input data. We provide extensive results on the properties of the proposed approximation, SSK-LP, with respect to prediction accuracy, runtime and memory consumption. Using some real-life datasets of text mining tasks, we show that models based on SSK and SSK-LP perform similarily for a set of real-life learning tasks, and that the empirical runtime model is also useful in roughly determining total learning time for a SVM using either kernel.

Keywords Machine Learning · Kernel Methods · String Kernels · Runtime estimation · Memory consumption estimation

Mathematics Subject Classification (2000) 68T05 · 90C59 · 93A30

A.K. Seewald
Seewald Solutions, Leitermayergasse 33/24, A-1180 Vienna, Austria
E-mail: alex@seewald.at

F. Kleedorfer
Research Studios Austria, Smart Agent Techn., Hasnerstraße 123, A-1160 Vienna, Austria
E-mail: florian.kleedorfer@{researchstudio.at,austria.fm}

1 Introduction

Historically, the perceptron classifier was one of the first linear learning machines, and although the famous paper Minsky & Papert (1969), which showed that it cannot learn non-linear discriminant models (such as the XOR problem), had an adverse effect on its development, the perceptron classifier can be seen as ancestor of both Neural networks and Support Vector machines.

In both cases, a solution to solve non-linear problems was introduced: for Neural networks, by combining layers of perceptrons with the back-propagation learning rule; for SVMs, by regularizing the linear problem so that it has a unique solution (i.e. the maximum margin hyperplane) and by blowing up the dimensionality of the original space to improve linear separability, plus using slack variables to find a unique solution even when the data is not linear separable.

Given a linear separable dataset, there will normally be infinitely many hyperplanes that separate the two classes. The perceptron classifier will randomly choose one of them. SVM instead finds the maximum margin hyperplane, which is defined as the separating hyperplane with the maximum distance (margin) from the convex hull of both classes. This is uniquely defined for each linear separable dataset, and thus guarantees an unique solution. If the classes are not linear separable, SVM allows some examples on the wrong side of the hyperplane (determined by complexity parameter C), which again guarantees an unique solution.

Support Vector machines improve upon the perceptron by not only having one well-defined global optimum in concept space, but also proven convergence towards this optimum. Neural networks have many local optima, and as for the perceptron convergence is not assured.

Usually, drastically increasing input dimensionality also makes a learning algorithm much slower. However, the algorithms for solving the SVM can be reformulated to use not the high-dimensional transformed data samples themselves, but just the dot-product of two high-dimensional data samples. The high-dimensional samples themselves are then never needed directly. This is where the kernel function comes into play. A kernel function is a computational short-cut to efficiently compute this necessary dot-product without needing to expand the data samples first. Thus, for example, a SVM which has as input all possible products of nine numeric input variables (i.e. a 9-degree polynomial kernel) can compute the dot-product as $K(x_1, x_2) = \langle x_1, x_2 \rangle^9$ (i.e. the dot-product of two samples to the ninth power) instead of as $K(x_1, x_2) = \langle \phi(x_1), \phi(x_2) \rangle$ with

$$\phi(x) = \left(\prod_{i,j,k,l,m,n,o,p,q} x[i] * x[j] * x[k] * x[l] * x[m] * x[n] * x[o] * x[p] * x[q] \right)$$

(i.e. the dot-product of two samples in the high-dimensional space). Both ways to compute the dot-product are equivalent, but it is instantly obvious that the first formulation can be computed much faster.

This kernel trick, as it is also called, allows to uncouple the training method from the data – since only dot-products are needed by the SVM algorithm – and thus allows the SVM to process not only numeric data, or data which has been appropriately transformed into numeric format, but also character sequences, trees, lists, graphs and other complex data structures directly (see Haussler (1999) and Cortes et al. (2004) for two general approaches to build such kernels). In each case, we only need to define a valid kernel, i.e. a computational shortcut to efficiently compute a dot-product of any two samples. This is a unique capability which few other classifiers share.[1] However, efficiency is an issue with some non-standard kernels and experiments can take a long time and much memory.

We tackled these issues from two directions, beginning with one well-developed kernel, the Subsequence String Kernel (SSK, Lodhi et al. (2002)).

- By introducing an approximation to the SSK, SSK-LP, which takes far less memory than SSK and is usually several orders of magnitude faster.
- By creating average runtime and maximum memory consumption models for both variants. These models can be applied prior to experimentation, and compute expected runtime and memory consumption for a single kernel evaluation. As kernel evaluation runtime is one major determinant of total runtime of the SVM, especially for complex kernels, this gives a rough estimate of total runtime and the minimum amount of memory needed to successfully run the kernel on a given dataset.

It is our hope that both approaches will prove sufficient to apply these SSK variants more widely, and that the underlying approach will inspire others as well.

2 Related work

Leslie et al. (2002) propose a different kind of string kernel, the so-called spectrum or mismatch kernel. The restriction to n-grams (i.e., contiguous subsequences) as features allows a more efficient implementation. They do not show that their kernel satisfies Mercer's theorem, but Cortes et al. (2004) demonstrate this. As SSK-LP with $\theta = 2 * n$ is equivalent to using n-gram features, this specific variant must therefore satisfy Mercer's theorem as well.

Collins & Duffy (2002) introduce convolution kernels for natural language processing, i.e. a tree kernel based on the ideas of Haussler (1999). They quote (unproven) an average runtime that is linear in the tree size, but worst case performance is on the order of square of tree size – similar to our approach. This kernel is shown to satisfy Mercer's theorem. Features are common subtrees, so again no gaps are considered.

[1] E.g. Instance-based learning algorithms. Even in that case an appropriate distance measure needs to be defined, and kernels can be viewed as distance measures – see for example Section 5.3.

Lodhi et al. (2002) is another work that can be viewed as based on Haussler (1999), and gives the original implementation of SSK as well as some experimental results. This work will be amply discussed throughout this paper where appropriate.

A recent work is Cortes et al. (2004) on Rational Kernels, which unifies several approaches to build kernels on complex data structures, and also shows that edit distance with a alphabet of size greater than 1 does not satisfy Mercer's theorem. They show that a single algorithm is sufficient to implement all rational kernels, albeit via a finite automaton approach – a programming language by any other name – and thus cannot claim to significantly simplify the implementation of new kernels.

Rousu & Shawe-Taylor (2005) offer a comprehensive evaluation of several approaches to compute string kernels, namely trie-based and full dynamic programming (similar to SSK), and propose a sparse dynamic programming approach. While runtime results are given, no analytical model is obtained. Additionally, their proposed new method depends on alphabet size and the number of gaps that are permitted as well as string length, so the results are not directly comparable to our work. Their runtime complexity is $O(n|M|log|s|)$, where $|M|$ is the size of the alphabet. Worst-case memory consumption is not given. They have focussed only on single kernel evaluations rather than running a SVM based on their proposed kernel.

3 Background

This section aims to introduce the reader to the general concept of Support Vector Machines, the *kernel trick* and more specifically to the String Subsequence Kernel (SSK). After this, we explain our approximation method, Lambda Pruning (SSK-LP), in detail.

3.1 Kernels and support vector machines

A Support Vector Machine (SVM) is a classification algorithm which learns a hyperplane that separates two sets of points that belong to different classes. The algorithm is based on viewing the linear separation of these sets as an optimization problem, the goal of which is to maximize the margin of the hyperplane, i.e. the distance from the hyperplane to the nearest point of each class, which is a measure of the robustness of the separation. In case of non-separability, the sum of the distances of all points which are on the wrong side of the hyperplane is bounded by a complexity parameter, usually called C.[2]

Several methods exist that solve this problem efficiently, for instance *Sequential Minimal Optimization* (SMO, Platt (1998)) or *SVM-light* (Joachims (2002)). A nonlinear mapping from the attribute space to a higher-dimensional

[2] The original formulation of SVMs calls this complexity parameter λ. Here, λ is a decay factor that penalizes non-continguous subsequence matches.

feature space is usually applied prior to training, since a higher number of dimensions increases the probability that the data becomes linear separable within the high-dimensional feature space. This space may be very high-dimensional, which would normally incur massive computational issues, as runtime and memory consumption would have to increase at least linear with the number of dimensions used.

This is the point where one central idea of SVMs comes into play, the so-called *kernel trick*: A closer look into the mathematical formulation of SVMs shows that the data points (vectors) are never used directly, they are only introduced into the calculations via the dot-product between two such points (see e.g. Cristianini & Taylor (2000)). This fact gives rise to the idea of a function combining the mapping and the dot product, so that it is not necessary to compute the high-dimensional vectors explicitly. This kind of function is called a *kernel function*, or short *kernel*.

More formally, for any mapping $\phi : D \to F$ the function $K : K(x_1, x_2) = \langle \phi(x_1), \phi(x_2) \rangle$ is a kernel function. ($\langle ., . \rangle$ denotes the dot product). An interesting consequence is that the attribute space D from which the kernel projects data into F need not necessarily be an Euclidean space, but may have any form and dimension, even infinite dimensionality.

3.2 String subsequence kernel (SSK)

The basic idea behind this type of kernel is to define the dot product of two sequences by means of the subsequences they contain. The high-dimensional feature space implied by this kernel contains all possible subsequences as features. In order to understand the subsequent discussion better, we will now cite the corresponding part (Def. 1 ff.) from Lodhi et al. (2002) in an abbreviated form here. Please note that our numbering scheme is slightly different.

Definitions Let Σ be a finite alphabet. A string is a finite sequence of characters from Σ, including the empty sequence. For strings s,t, we denote by $|s|$ the length of the string $s = s_1 \dots s_{|s|}$, and by st the string obtained by concatenating the strings s and t. The string $s[i : j]$ is the substring $s_i \dots s_j$ of s. We say that u is a subsequence of s, if there exist indices $\mathbf{i} = (i_1, \dots, i_{|u|})$ with $1 \leq i_1 \leq i_2 \leq i_{|u|} \leq |s|$, such that $u_j = s_{i_j}$, for $j = 1, \dots, |u|$, or $u = s[\mathbf{i}]$ for short. The length $l(\mathbf{i})$ of the subsequence in s is $i_{|u|} - i_1 + 1$. We denote by Σ^n the set of all finite strings of length n, and by Σ^* the set of all strings

$$\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$$

We now define feature spaces $F_n = \mathbb{R}^{\Sigma^n}$. The feature mapping ϕ for a string s is given by defining the u coordinate $\phi_u(s)$ for each $u \in \Sigma^n$. We define

$$\phi_u(s) = \sum_{\mathbf{i}:u=s[\mathbf{i}]} \lambda^{l(\mathbf{i})}$$

for some $\lambda \in (0, 1)$. These features measure the number of occurrences of subsequences in the string s weighting them according to their lengths. Hence, the inner product of the feature vectors for two strings s and t give a sum over all common subsequences weighted according to their frequency of occurrence and lengths.

As can be seen by above definition, the method proposed in Lodhi et al. (2002) considers contiguous as well as non-contiguous subsequences; the degree of contiguity determines the weight of the subsequence match in the comparison. SSK is parametrized by two values, n and λ. The length of the common subsequences to look for is n. λ is a real value $\in (0, 1)$, which is used as a decay factor to penalize non-contiguous substring matches[3]. The kernel implicitly maps the input strings to a feature space F that has one dimension for each possible combination u of n characters. The u-coordinate $\phi_u(s)$ of a given string s is calculated by summing over all occurrences of u in s. Each occurrence of u yields a value of λ^l, where l denotes the length of that occurrence of u in s, that is, the length of u plus all the interior gaps of the occurrence. The result of the kernel function for two input strings s,t is the dot product of their feature mappings. More formally:

$$\begin{aligned} K_n(s,t) &= \sum_{u \in \Sigma^n} \langle \phi_u(s) \cdot \phi_u(t) \rangle = \sum_{u \in \Sigma^n} \sum_{\mathbf{i}: u = s[\mathbf{i}]} \lambda^{l(\mathbf{i})} \sum_{\mathbf{j}: u = t[\mathbf{j}]} \lambda^{l(\mathbf{j})} \\ &= \sum_{u \in \Sigma^n} \sum_{\mathbf{i}: u = s[\mathbf{i}]} \sum_{\mathbf{j}: u = t[\mathbf{j}]} \lambda^{l(\mathbf{i}) + l(\mathbf{j})} \end{aligned} \tag{1}$$

At first glance this may seem like an exponential algorithm, but Lodhi et al. (2002) also propose an efficient recursive formulation with dynamic programming that uses only $O(n|s||t|)$ time. However, an essential feature to obtain the reported runtime performance is using a cache for all intermediate results, which uses $O(n|s||t|)$ space as well.[4]

3.3 Lambda pruning

Equation 1 shows that the result of the kernel evaluation is a sum over different powers of λ. It is obvious that the contribution of subsequences u to the overall result is the smaller the more the subsequence match is stretched in both strings. When analyzing the recursion tree that is processed for each kernel evaluation, it becomes clear, on the other hand, that these stretched matches neccessitate a vast amount of computational effort. These observations together motivate a change to the algorithm in a way that the recursion is stopped as soon as it is certain that the result of the current branch is very

[3] Note that all contiguous substring matches have the same value of $\lambda^{|u|}$.

[4] Initially, we implemented SSK without such a cache and it distinctly showed exponential runtime behaviour.

small. This is a trade-off between result accuracy and consumed computation time.

This behaviour can be achieved by introducing a bound to the parameters in the formula that account for the addends in the kernel computation $\lambda^{l(\mathbf{i})+l(\mathbf{j})}$. By bounding the sum of $l(\mathbf{i})+l(\mathbf{j})$, we can stop the computation once it reaches the bound, which we have called *Maximum Lambda Exponent*, or θ.

The new kernel parameter θ is used as the upper bound for exponents of λ that may occur in the computation. This means that a subsequence match for a string u is only counted in the final result if the sum of the length of u in s and the length of u in t (both lengths including the gaps) is smaller than θ. The definition of our kernel for Lambda Pruning is:

$$K_{n,\theta}(s,t) = \sum_{u\in\Sigma^n} \sum_{\mathbf{i}:u=s[\mathbf{i}]} \sum_{\mathbf{j}:u=t[\mathbf{j}]} f(l(\mathbf{i}),l(\mathbf{j})) \tag{2}$$

$$f(x,y) = \begin{cases} 0 & \text{if } x+y>\theta \\ \lambda^{x+y} & \text{otherwise} \end{cases}$$

The influence that θ has on the result is, qualitatively speaking, control of subsequence match relaxation. The definition set of θ is $x \in \mathbb{N}, 2n \leq x < \infty$. $\overline{\theta} = \theta - 2n$ specifies a slack value for match relaxation defining how many interior gaps each subsequence match may have in s and t together.

Using $\theta = 2n$ ($\overline{\theta} = 0$) makes SSK-LP identical to an n-grams kernel, as only contiguous substring matches are allowed. Values slightly above $2n$ ($\overline{\theta} > 0$) create a 'fuzzy' n-grams kernel, while using $\theta \in [4n, 8n]$ ($\overline{\theta} \in [2n, 6n]$) usually gives quite a good approximation of SSK, depending on the type of data used. We have chosen to set a default value of $\theta = 3n$ ($\overline{\theta} = n$), which seems a good compromise for a variety of learning tasks.

We should point out that the Gram matrix approximation proposed in (Lodhi et al., 2002), section 5, is concerned with reducing the number of kernel evaluations for a given learning task by decomposing each kernel evaluation into a product of two kernel evaluations on a smaller input set, and does not speed up the kernel evaluations themselves. As such, it is a different kind of approximation than SSK-LP and cannot be directly compared.

First, it is necessary to reformulate the original exponential time computation formula for $K_{n,\theta}$ into a recursive computation with time complexity $O(n|s||t|)$ using the helper functions K' and K''. K' counts the length from the beginning of the particular sequence through to the end of strings s and t instead of just $l(i)$ and $l(j)$. An efficient recursive definition of K' reduces the complexity to $O(n|s||t|^2)$. The second helper function K'' efficiently computes K' by reusing the results of K' for shorter s and t strings in the computation of longer strings.

This recursive computation of SSK has to be adapted in order to conform to this definition of lambda pruning. An additional parameter m is added to the kernel function. Each time a value of $\lambda^{|u|}$ is multiplied with the recursion result m is decremented by $|u|$ and passed to the next recursion level; if the

condition $m < 2i$ is true, the recursion ends prematurely as this means that i characters would need to be matched in each of string s and t, incurring a total weight of λ^{2i}. Thus the recursion depth is limited by m.

Definition 1 *Recursive computation of SSK with Lambda Pruning (SSK-LP)*

$$
\begin{aligned}
K'_{0,m}(s,t) &= 1, \quad \text{for all } s,t \\
K'_{i,m}(s,t) &= 0, \quad \text{if } \min(|s|,|t|) < i \\
K_{i,m}(s,t) &= 0, \quad \text{if } \min(|s|,|t|) < i \\
K'_{i,m}(s,t) &= 0, \quad \text{if } m < 2i \\
K'_{i,m}(sx,t) &= \lambda K'_{i,m-1}(s,t) + K''_{i,m}(sx,t), \quad i = 1,\ldots,n-1 \\
K''_{i,m}(sx,tu) &= \lambda^{|u|} K''_{i,m-|u|}(sx,t), \quad \nexists k : u_k = x \\
K''_{i,m}(sx,tx) &= \lambda(K''_{i,m-1}(sx,t) + \lambda K'_{i-1,m-2}(s,t)) \\
K_{n,m}(sx,t) &= K_{n,m}(s,t) + \sum_{j:t_j=x} K'_{n-1,m-2}(s,t[1:j-1])\lambda^2
\end{aligned}
$$

Contrary to the standard SSK implementation, where all return values of K' and K'' were cached, in the case of Lambda pruning we did not use a cache of intermediate results. Although this would have improved runtime slightly, it would also have drastically increased memory consumption beyond that needed for the original SSK.

This is because there are usually several ways to enter the same recursion tree, but at different entry points. While in the non-depth-limited case (for SSK) caching is simple since all these entry points can use the same value for that subtree, in the depth-limited case (SSK-LP) we would need more than one cached value per subtree. This is because, depending on how deep the entry point is, the subtree must be followed for a variable depth so that the total depth does not exceed the depth-limit. One reasonable way to solve this would be to add the current depth to the cache index. This however would increase memory consumption for the full cache by a factor of $n + 1$ over SSK, which would have exceeded the available memory on our machine for the largest datasets.

For the Support Vector Machine algorithm to work provably it is sufficient to show that the kernel function satisfies Mercer's Theorem, although simpler definitions exist for the finite-dimensional case. It is also sufficient to show that a function ϕ exists such that K can be written as a dot product of $\phi(x_1)$ and $\phi(x_2)$, or that the kernel matrix is always positive semi-definite. Some kernels have been introduced that are not known to satisfy Mercer's theorem – which means that they either do not satisfy the theorem, or that they do but no proof has yet been found – but which still work well in practice. SSK-LP falls into this category, as do some of the kernels used within BioInformatics, and kernels based on edit distance, see Cortes et al. (2004). Theoretically, using a non-valid kernel can lead to non-convergence of the SVM algorithm. However, both SSK and SSK-LP (with SSK-LP being the default setting) have been

available within WEKA since August 2005, and up to date we have not heard of a single case where SSK-LP did not converge, which may indicate that SSK-LP is either a avalid kernel or that non-convergence is not of practical relevance. Note also that SSK-LP with $\theta = 2 * n$ clearly satisfies Mercer's theorem, as it is equivalent to the kernel from Leslie et al. (2002) which is know to satisfy Mercer's theorem.

Remark 1 An alternative would be to threshold both x and y with the same value θ, rather than the sum. This has the advantage of guaranteeing a valid kernel, since we are mapping each sequence to a feature vector only depending on λ and θ. The same argument does not hold for SSK-LP, as there the feature vectors of x and y depend on each other. We ran similar extensive experiments and found that this variant runs 1.57 ± 0.57 times slower than SSK-LP. As we search deeper, this is to be expected. Memory consumption is unchanged. What really surprised us is that the approximation accuracy is so much worse[5] – the values are on average 3.50 ± 3.48 times too high, while SSK-LP has a much better factor of 0.80 ± 0.33. While both converge towards the true value of the original SSK, this variant converges much slower than SSK-LP. We speculate that the combined threshold for x and y yields some nontrivial advantage which we are unable to explain theoretically. In any case, the faster convergence of SSK-LP makes it far more suited for practical applications, so we focussed our investigation on SSK-LP.

4 Modeling time, space and error

This section describes the average case runtime model, the worst case memory consumption model and an investigation into SSK-LP approximation error.

As we shall see later, these very accurate models give useful estimates on the runtime of large learning tasks, enabling an intelligent choice of parameter settings and subsampling sizes for a given runtime and memory budget, both for SSK and SSK-LP.

When CPU time must be measured in months and years and memory consumption in gigabytes, as for SSK, it is of utmost importance to know in advance how much memory will be needed and how long we must wait for a result. If such models become more widespread, these questions can be answered a priori from statistical measurements of datasets, which would constitute a big step forward in practical application of complex machine learning systems.

4.1 Experimental setup

There are several parameters of the SSK which directly influence runtime and memory consumption. Most prominent among these is the string length

[5] This was also what we found in less systematic initial experiments and what prompted us to develop SSK-LP.

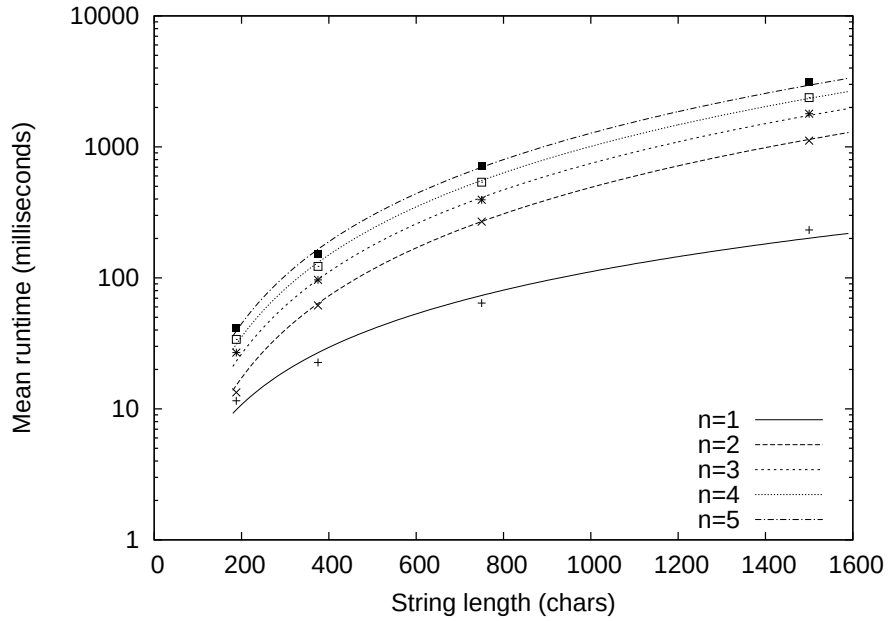

Fig. 1 Runtime for SSK, averaged over λ.

strLen of the input strings (assuming strLen $\approx |s| \approx |t|$) and the common subsequence length n. λ has no effect on runtime and memory consumption, but features prominently in the approximation error versus SSK-LP and was therefore also included. For SSK-LP, we also have θ as an essential parameter for the trade-off between runtime and approximation accuracy.

Five values for n were considered (1,2,3,4,5), as well as for λ (0.1, 0.25, 0.5, 0.75, 0.9). θ defaults to $3 * n$ which yields a good trade-off between speed and accuracy, but we used $6 * n$, $9 * n$ and $12 * n$ as well. As n influences the runtime quite drastically, the maximum value of n was chosen so as to keep the runtime for the whole experiment on the order of several CPU-months.

Rather than doing a theoretical worst-case analysis on runtime, we opted for a systematic experiment with real-life input data in an average-case runtime analysis. Random strings are somewhat of a best case for SSK, while the worst case would be when both strings consist only of the same character, and both are unlikely to appear in practice.

As real-life corpus, we used an earlier text-mining dataset from Seewald (2003), consisting of several tens of thousand samples of MEDLINE abstracts, as string corpus. Each MEDLINE entry corresponds to a single string in this corpus. From this corpus, we sampled about one hundred strings of specified lengths, namely strLen = 1500, 750, 375, and 188. We first sampled strings with exactly the desired length, then those one character longer than the desired length, then those one character shorter than the desired length and so on until enough samples were collected, so the lengths of the collected strings

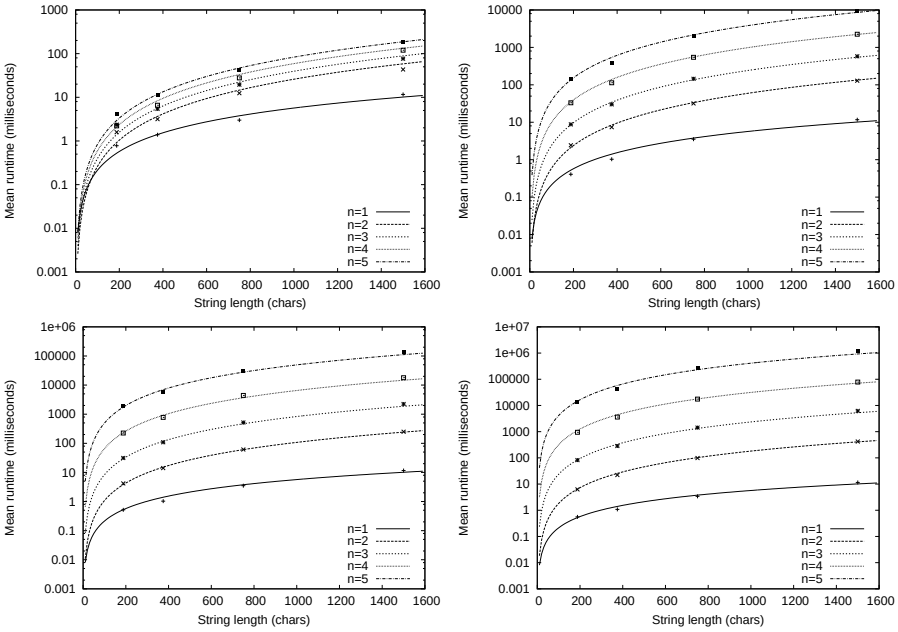

Fig. 2 Runtime for SSK-LP, averaged over λ. Top left $\theta = 3{*}n$, Top right $\theta = 6{*}n$. Bottom left: $\theta = 9 * n$, Bottom right: $\theta = 12 * n$.

are as close to the target as possible. Each set of string samples was then paired with a randomly shuffled version of the same set. Thus we obtained pairs of strings with approximately the desired length.[6]

We tested all combinations of these parameter values in extensive experiments on a single machine[7], and measured time in two different ways to make sure that the overall CPU load was accounted for, as the experiment did not use a dedicated machine. The reported execution times are relative to this platform, and therefore will need to be calibrated for other platforms, but should give a rough overview of the speed of string kernel evaluation.

4.2 Average case runtime

Figure 1 shows the runtime for SSK, while Figure 2 shows the runtime for SSK-LP and the four different values for θ. All runtimes are averaged over different

[6] The actual string lengths were 1499.60±0.49, 749.18±3.13, 373.91±7.34, and 187.58±3.09, respectively.

[7] Athlon64 4000+ with 4GB of main memory, which was sufficient to keep even the largest working set in memory, running the pure64 version of Debian, kernel 2.6.11., with Sun Java 1.5. (64bit version) and a CVS version of WEKA from May 2005 with SMO.java V1.12.

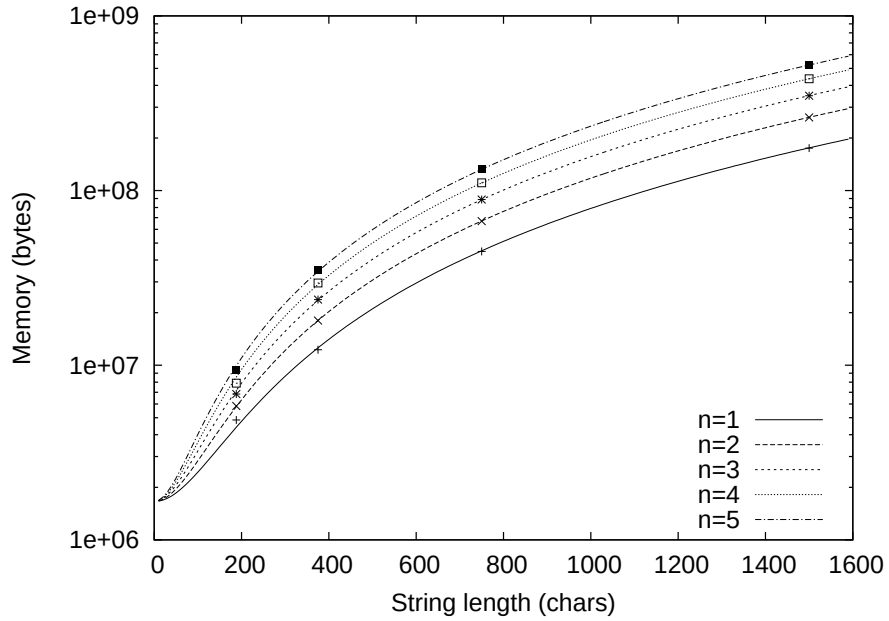

Fig. 3 Memory consumption for SSK, averaged over λ.

values of λ to reduce variance.[8] In each case we fitted double logarithmic ridge-regression models which are shown below, and as lines in each graph.

The coefficient of a parameter in the double log linear model can be interpreted as exponent to this parameter, while the whole model is the product of all parameters with their respective exponents. We have chosen to rearrange the obtained model in the interests of comprehensibility, and also aimed for simplicity.

Each symbol in the graph corresponds to a specific average runtime measurement, averaged both over the approximately 100 string sample pairs as well as over all different values of λ.

For SSK, we obtained

$$\text{runTime}_{\text{SSK}} = \text{strLen}^{1.456} * e^{-5.3165}$$

(for $n = 1$) and

$$\text{runTime}_{\text{SSK}} = (\text{strLen}^2 * n)^{1.0402} * e^{-8.897}$$

(for $n > 1$).[9] The output is the estimated time in milliseconds per kernel evaluation. Some small optimizations of the first inner loop are responsible

[8] Average relative standard deviation is 0.44% for SSK and 1.03% for SSK-LP, the latter for execution times of more than 100ms. Values below this had high variance as we approach the accuracy limits of runtime measurement (1ms per sample).

[9] We used $\text{strLen}^2 * n$ as one of the input features because of theoretical considerations concerning worst-case runtime. Similar but not identical results were obtained when using strLen and n as separate features.

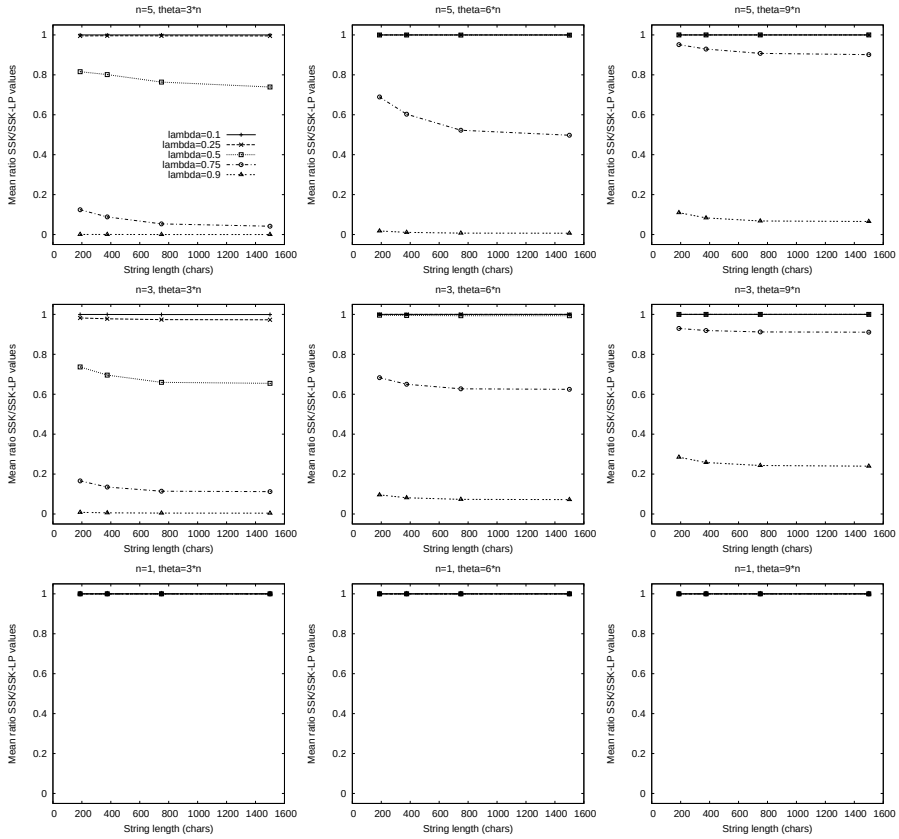

Fig. 4 Ratio of SSK-LP values to SSK values (1.0 = perfect agreement).

for the better runtime of $n = 1$, and for $n > 1$ we see that the average case runtime is very close to the worst case runtime of $O(\text{strLen}^2 * n)$ which was given in Lodhi et al. (2002).

For SSK-LP, we similarily obtained

$$\text{runTime}_{\text{SSK-LP}} = \text{strLen}^{1.425} * e^{-8.1063}$$

(for $n = 1$) and

$$\text{runTime}_{\text{SSK-LP}} = \frac{\theta!^{1.0549}}{n!^{1.8839} * (\theta - n)!^{1.039} * \theta^{1.3331}} * \text{strLen}^{1.9978} * e^{-10.4875}$$

(for $n > 1$). The left part is similar to a binominal coefficent of θ and n (except for the additional factors of $\frac{1}{n!^{0.8839}}$ and $\frac{1}{\theta^{1.3331}}$), which we have reason to believe to feature prominently in the runtime of SSK-LP due to an earlier analysis. The correlation coefficient of this model is 0.9986, relative mean squared error is 0.1662 (5.2037%), which reinforces our belief that this model, albeit being rather more complex than the one for SSK, is valid. Also, Figure 2 shows graphically that this function fits the observations very well.

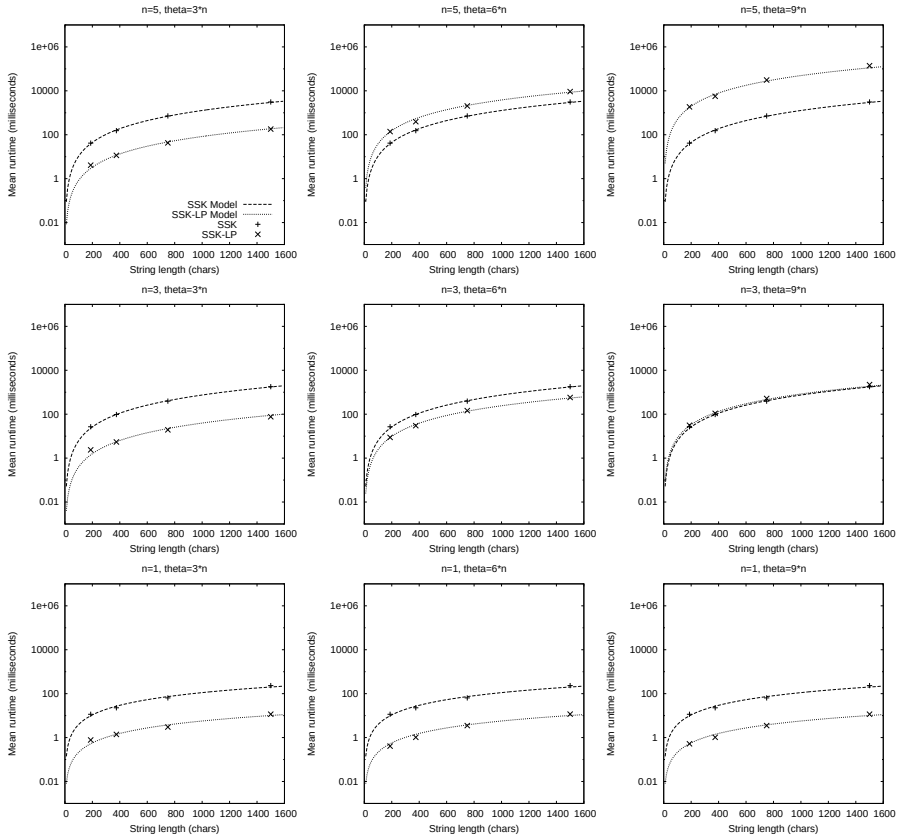

Fig. 5 Runtime of SSK-LP and SSK (in milliseconds) according to the runtime models, also showing real runtime as + and ×.

4.3 Worst case memory consumption

Figure 3 shows the memory consumption for SSK. Memory consumption was averaged over all values of λ, which reduces variance.[10] For each set of string pair samples, only the highest measured memory consumption was used. A regression model has been fitted to the data. Memory consumption is dominated by the cache for intermediate results, and can be computed as $\text{Mem}_{\text{SSK}} = 38.5888 * (\text{strLen} + 1)^2 * (n + 1) + 1,655,638.1147$ (in bytes, round up final result).

The worst case memory consumption of SSK-LP, $\text{Mem}_{\text{SSK-LP}}$, is 4,259,840 bytes. The same value was consistently measured in all experimental runs.

[10] As expected, values were practically identical for all five different λ values, given any set of other parameter values.

Table 1 This table shows all real-life performance results w.r.t. estimated and real runtime (RT) at one glance. RT/ev is the average runtime for one kernel evaluation in milliseconds. All variants were run with default settings $n = 3$, $\lambda = 0.5$, and $\theta = 3 * n = 9$. Additionally, 5.2. used feature space normalization.

Sec.	SSK				SSK-LP			
	Estimated		Real		Estimated		Real	
	RT/ev	RT	RT	res.	RT/ev	RT	RT	res.
5.1	1731	399.5d	n/a	n/a	89	20.5d	21.6d	97.5%
5.2	0.36	2h 17m	16h 50m	84.32%	0.03	11.4m	3h 19m	83.98%
5.3	17.76	17.7m	13.95m	see text	1.09	1.09m	1.03m	see text

4.4 Approximation error of SSK-LP vs. SSK

Figure 4 shows the approximation error of SSK-LP vs. SSK in dependence on θ (X axis), n (Y axis) and λ (lines within each subgraph).[11] As can be seen, for λ of 0.1 and 0.25, agreement is very good over all parameter settings. In some cases agreement is also good for $\lambda = 0.5$. For λ of 0.75 and 0.9, the approximation is much worse, since in this case non-contiguous subsequence matches have a much higher effect on the final value. Figure 5 shows the runtime of SSK vs. SSK-LP in the same layout as Figure 4. It shows that SSK-LP is faster in the lower triangle (for slightly less than half of all parameter settings tested). A more precise estimate of the applicability of SSK-LP to a given problem can always be computed directly, via the models of memory consumption and runtime we presented earlier. The observed constantly small memory footprint of SSK-LP enables its application within embedded systems, which would not be feasible for SSK.

5 Real-life performance

In this section we report on experiments with real-life data. We have focussed on three different problems: domain recognition from biological research papers, email spam recognition from sender email address, and the use of SSK as a similarity measure for redundancy clustering – a linguistic task related to sentence entailment.

We used the default settings of $n = 3$ and $\lambda = 0.5$ for SSK, and $\theta = 9$ for SSK-LP, unless otherwise noted. The SVM cost parameter was set to $C = 1$. Table 1 gives a short overview of all results.

5.1 Text mining

For this experiment, we wanted to check the suitability of SSK for standard text mining. We chose the domain dataset from Seewald (2003).[12] The learn-

[11] Because of space restrictions, only $n = 1, 3, 5$ and $\theta = 3 * n, 6 * n, 9 * n$ are shown.

[12] Dataset available at `http://alex.seewald.at/projects.html#biomint`

ing task was text classification, i.e. classifying bibliographic entries from MEDLINE as belonging to one of four biological domains: Archaea, Bacteria, Eukaryota or Virus. We used the 5% sample and a two-fold CV for accuracy estimation. The results are thus comparable to those reported in Seewald (2003), Table 1, *Acc.CV*. The dataset contains 5156 examples. Average length of the input strings is 1497.21±486.26.

According to our runtime model, SSK is expected to take around 1731ms for one kernel evaluation. To generate the full kernel matrix for a dataset with n examples, $\frac{n(n-1)}{2} \approx \frac{n^2}{2}$ kernel evaluations are needed.[13] Additionally, a two-fold CV must compute a half-sized kernel matrix twice, so the total number of kernel evaluations is about $\frac{3n^2}{4}$. Running SSK on this dataset would have taken more than a year, and was therefore not considered.

SSK-LP, on the other hand, should take around 89ms for one kernel evaluation, which reduces the expected runtime to 20.5 days. We ran SSK-LP on this data, and the actual runtime was 518h (21.6d), which agrees well with our estimate.

The performance of SSK-LP at 97.5% accuracy is competitive to a linear SVM with the word vector as input (i.e. one attribute for each word that appears in training data) at 97.7%. Concerning runtime, it is not competitive: the linear SVM takes around two minutes for the same task. This agrees well with Joachims (2002), who argued that for most text classification tasks the relatively simple word vector representation combined with a linear kernel is already sufficient.

5.2 Spam filtering

For the second task, we chose a string classification task. Rather than a classic text mining task, where it is feasible to segment the string into smaller constituents by tokenization, this is no longer easily possible for string classification. Seewald (2007) describes extensive experiments on a large corpus consisting of about 90,000 ham and spam mails. Based on the anecdotal observation that it is often possible to recognize spam mails via a small number of string features (namely, sender address, sender name and the subject), we were interested to find out whether SSK could learn to do the same.

We randomly chose a small number of 3,902 samples of sender email addresses from this corpus (roughly half from spam and half from ham mails).[14] Sender email addresses were chosen since these are the hardest to tokenize; sender name and subject would have been amenable to a simple tokenization approach and were thus considered unsuitable. The string length in this case is only 25.56±11.08, so both SSK and SSK-LP can be run on this data. Our runtime models give 0.36ms for SSK, and 0.03ms for SSK-LP, which both are below the applicability of the runtime model – the shortest runtime that could

[13] This is always incurred, as WEKA needs to compute training set accuracy as well.

[14] Corpus available at `http://alex.seewald.at/spam/index.html#datasets`

be measured was 1ms. This explains why the agreement in that case is quite bad. SSK took 16h 50min to run (est. 2h 17m), while SSK-LP took 3h 19min (est. 11.4m). The speedup factor of about an order of magnitude for SSK-LP over SSK can still be approximately deduced from the differences in estimated runtimes. We used feature space normalization[15] here, which approximately doubles runtime for kernel evaluation in the given implementation.[16]

The accuracy was again estimated via two-fold cross-validation. The accuracies of SSK at 84.32% and SSK-LP at 83.98% were very similar. Systematic parameter optimization of c, n and λ for SSK, including switching off feature space normalization, had only negligible influence on the result. The default parameter settings already gave the second highest accuracy of 83.39%. SSK-LP with $\theta = 2{*}n = 6$ (equivalent to 3-grams) using feature space normalization offers an accuracy of 82.4% and took 57min to run.

5.3 Redundancy clustering

Here, we describe previously unpublished work on redundancy clustering within the BioMinT project (for context see Pillet et al. (2005), Seewald (2003), Seewald (2004)). The main focus of BioMinT was the extraction of new knowledge from biological research papers via innovative text mining approaches. One subtask was to automatically determine and remove redundant sentences from a result set to give a succint presentation of the found results to the user. On the one hand, this is somewhat related to summarization approaches; on the other hand, to the known linguistic concept of sentence entailment. Entailment is considered a very hard task, and there are few approaches which perform better than a simple baseline approach.

Within BioMinT, a corpus of redundant sentences was provided by one of our partners.[17] We received this information as a set of sentence groups. Within each sentence group there was a primary sentence that provided all of the information, associated with a set of secondary sentences that provided only information that was already present in the primary sentence. All secondary sentences were considered redundant. To make the task more feasible, we mapped this dataset to pairwise similarity, without distinguishing between primary sentence and secondary sentences. The similarity between two sentences was determined via several methods, and all pairs of sentences with similarity values above a certain threshold were considered redundant while all other pairs were considered to be not redundant. Thus we obtained 59,810 sentence pairs, of which 2,500 were marked as redundant. Instead of fixing the threshold arbitrarily and reporting a single result, we have chosen to visualize the results as a ROC curve, where the performance at arbitrary thresholds is observable at one glance. The string length for this learning task was

[15] Using kernel $K'(x,y) = \frac{K(x,y)}{K(x,x)K(y,y)}$, which ensures $K'(x,x)$ is equal to 1.

[16] It would have been possible to implement this in a way as to incur additional cost linear to the training set size by precomputing all $K(x,x)$ values.

[17] Dataset available at `http://alex.seewald.at/projects.html#biomint`

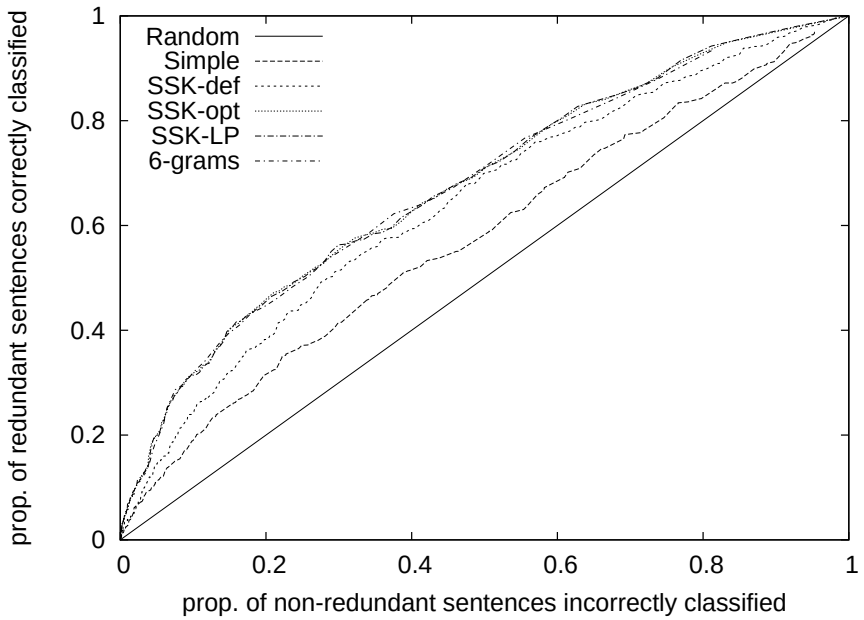

Fig. 6 This figure shows a ROC curve of several similarity measures (see text). The X axis corresponds to the proportion of non-redundant sentences which were incorrectly classified, and the Y axis corresponds to the proportion of redundant sentences which were correctly classified by the system. The unbroken line from (0,0) to (1,1) corresponds to the average performance of a random similarity measure.

165.68±61.27, giving single evaluation times of 17.76ms for SSK and 1.09ms for SSK-LP.

We have selected five ways to determine similarity between arbitrary sentences (S1,S2):

- *Simple*, the number of common words between S1 and S2, divided by the length (in words) of the shorter sentence. This is a simple baseline approach.
- *SSK-def*, which determines the similarity between S1 and S2 by computing an evaluation of the unnormalized string kernel (i.e. SSK), which is normally used as a kernel for the learning algorithm family of Support Vector Machines. We chose the default settings of $\lambda = 0.5$ and $n = 3$ here.
- *SSK-opt* is the same as *SSK-def* but with $n = 6$ which was found to be the optimal setting for n here.
- *SSK-LP*, which determines the similarity between S1 and S2 via SSK-LP. In this case, we used $\lambda = 0.5$, $n = 6$ and $\theta = 20$.
- *6-grams*, which determines the similarity between S1 and S2 via SSK-LP. In this case, we used $n = 6$ and $\theta = 2 * n = 12$, so this is equivalent to a simpler 6-grams approach.

Figure 6 shows the results. The unbroken line visualizes the average performance of a random similarity measure (i.e. where the values of sim(S1,S2) are chosen randomly). The higher above this line a measure is, the better it performs. As can be seen, *SSK-def* already performs much better than the simpler baseline approach. Parameter optimization for n improves on this result, but *SSK-opt*, *SSK-LP* and *6-grams* are very similar. The equivalent *6-grams* approach, which is simpler and could be implemented more efficiently, performs competitively. Optimizing λ does not improve performance for SSK beyond *SSK-opt*. λ has of course no effect on *6-grams*.

6 Conclusion

The work at hand presents *Lambda Pruning*, a novel approach to approximating the String Subsequence Kernel (SSK) by Lodhi et al. (2002). The resulting kernel, SSK-LP for short, is implemented in Java and available as a part of the WEKA data mining platform.[18] Our runtime and space complexity models allow to compute the expected runtime and memory consumption for both variants, and would even allow an automatic choice between the variants based on input data and given parameter settings.

We have investigated both variants on several learning tasks and noted that the string kernels are most useful for string classification tasks where a tokenization is not easily apparent, and as similarity measure for redundancy clustering. The given runtime models usually agree well with actual runtime and have proven useful to determine the relative speedup of SSK-LP versus SSK as well as to exclude some parameter settings because of excessive runtime. Our work is therefore an important step towards the practical applicability of string kernels for real-life learning tasks, and this approximation approach may prove useful for other complex kernels as well. Also, the presented runtime and memory consumption models allow intelligent choice of parameter values and subsampling sizes for a given time and memory budget, which will help to apply both SSK and SSK-LP in practical applications.

We should note that in our real-life experiments, n-gram approaches were found to be competitive, and their implementations can be made much more efficient. More work is needed to improve SSK to take adequate advantage of the capability to consider non-contiguous subsequences as well as towards improving its efficiency.

Being the first algorithm for learning from sequences in WEKA, this opens a new perspective for its users; moreover, the reduced complexity of SSK-LP makes the Support Vector Machine approach a viable alternative for text mining even on small devices with little main memory as it has a small constant memory footprint. Last but not least the availability of an open source implementation will hopefully motivate others to implement further kernels of strings and other complex data structures for WEKA.

[18] `http://www.cs.waikato.ac.nz/~ml/weka`. Use weka.classifiers.functions.SMO with kernel weka.classifiers.functions.supportVector.StringKernel. -P 0 = SSK, -P 1 = SSK-LP.

Acknowledgements This research was in small part supported by the European Commission as project no. QLRI-CT-2002-02770 (*BioMinT*) under the RTD programme *Quality of Life and Management of Living Resources*. The Studio Smart Agent Technologies is supported by the Austrian Federal Ministry of Economics and Labour. We gratefully acknowledge the support of the University of Manchester in providing the redundancy clustering dataset.

References

Collins M., Duffy N. (2002) *Convolution kernels for natural language.* In: Dietterich T.G., Becker S., Ghahramani Z. (eds), Advances in Neural Information Processing Systems 14 (2001) pp. 625–632, Cambridge, MA, MIT Press.

Cortes C., Haffner P., Mohri M. (2004) *Rational Kernels: Theory and Algorithms*, Journal of Machine Learning Research 5 (2004) 1035–1062.

Cristianini N., Shawe-Taylor J. (2000) *An Introduction to Support Vector Machines*, Cambridge University Press, 2000.

Frank E., Witten I.H. (2005) *Data Mining - Practical Machine Learning Tools and Techniques (2nd ed)*. Morgan Kaufmann / Elsevier, 2005. `http://www.cs.waikato.ac.nz/~ml/weka/`

Gartner T., Flach P.A., Wrobel S. (2003) *On Graph Kernels: Hardness Results and Efficient Alternatives*, Sixteenth Annual Conference on Computational Learning Theory and The Seventh Kernel Workshop (COLT-2003).

Haussler D. (1999) *Convolution kernels on discrete structures*. Technical Report UCSCCRL-99-10, Baskin School of Engineering, University of California, Santa Cruz.

Joachims T. (2002) *Learning to Classify Text Using Support Vector Machines*, Kluwer Academic Publishers, May 2002.

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

Lodhi H., Saunders C., Shawe-Taylor J., Christianini N., Watkins C. (2002) *Text Classification using String Kernels*, Journal of Machine Learning Research 2002 (2), pp. 419–444.

Minsky M., Papert S.A. (1969) *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, expanded edition, reprinted 1988.

Pillet V., Zehnder M., Seewald A.K., Veuthey A-L, and Petrak J. (2005) *GPSDB: a new database for synonyms expansion of gene and protein names.* Bioinformatics 2005 (21), pp. 1743-1744.

Platt J. (1998) *Fast Training of Support Vector Machines using Sequential Minimal Optimization*. In: Schölkopf B., Burges C., Smola A. (eds), Advances in Kernel Methods - Support Vector Learning, MIT Press.

Rousu J., Shawe-Taylor J. (2005) *Efficient Computation of Gapped Substring Kernels on Large Alphabets*, Journal of Machine Learning Research 6 (2005), pp.1323–1344.

Seewald A.K. (2003): *Recognizing Domain and Species from MEDLINE Proteomics Publications*. Workshop on Data Mining and Text Mining for Bioinformatics, 14th European Conference on Machine Learning (ECML-2003), Dubrovnik-Cavtat, Croatia.

Seewald A.K. (2004) *Ranking for Medical Annotation: Investigating Performance, Local Search and Homonymy Recognition*. Proceedings of the Symposium on Knowledge Exploration in Life Science Informatics (KELSI 2004), Milano, Italy.

Seewald A.K. (2007) *An Evaluation of Naive Bayes Variants in Content-Based Learning for Spam Filtering*. Intelligent Data Analysis 11(5).