Journal of Machine Learning Research 1 (2006) 1-18 Submitted 10/05; Published 2006; To appear

Large Scale Multiple Kernel Learning

Sören Sonnenburg Soeren.Sonnenburg@first.fraunhofer.de
Fraunhofer FIRST.IDA, Kekuléstr. 7, 12489 Berlin, Germany

Gunnar Rätsch Gunnar.Raetsch@tuebingen.mpg.de
Friedrich Miescher Laboratory of the Max Planck Society, Spemannstr. 39, Tübingen, Germany

Christin Schäfer Christin.Schaefer@first.fraunhofer.de
Fraunhofer FIRST.IDA, Kekuléstr. 7, 12489 Berlin, Germany

Bernhard Schölkopf Bernhard Schoelkopf@tuebingen.mpg.de
Max Planck Institute for Biological Cybernetics, Spemannstr. 38, 72076, Tübingen, Germany

Editor: Emilio Parrado-Hernández, Kristin P. Bennett

Abstract

While classical kernel-based learning algorithms are based on a single kernel, in practice it is often desirable to use multiple kernels. Lankriet et al. (2004) considered conic combinations of kernel matrices for classification, leading to a convex quadratically constrained quadratic program. We show that it can be rewritten as a semi-infinite linear program that can be efficiently solved by recycling the standard SVM implementations. Moreover, we generalize the formulation and our method to a larger class of problems, including regression and one-class classification. Experimental results show that the proposed algorithm works for hundred thousands of examples or hundreds of kernels to be combined, and helps for automatic model selection, improving the interpretability of the learning result. In a second part we discuss general speed up mechanism for SVMs, especially when used with *sparse* feature maps as appear for string kernels, allowing us to train a string kernel SVM on a 10 million real-world splice dataset from computational biology. We integrated Multiple Kernel Learning in our Machine Learning toolbox `SHOGUN` for which the source code is publicly available at `http://www.fml.tuebingen.mpg.de/raetsch/projects/shogun`.

Keywords: Multiple Kernel Learning, String Kernels, Large Scale Optimization, Support Vector Machines, Support Vector Regression, Column Generation, Semi-Infinite Linear Programming

1. Introduction

Kernel based methods such as Support Vector Machines (SVMs) have proven to be powerful for a wide range of different data analysis problems. They employ a so-called kernel function $\mathbf{k}(\mathbf{x}_i, \mathbf{x}_j)$ which intuitively computes the similarity between two examples $\mathbf{x}_i$ and $\mathbf{x}_j$. The result of SVM learning is an $\boldsymbol{\alpha}$-weighted linear combination of kernels with a bias b

$$f(\mathbf{x}) = \operatorname{sign}\left(\sum_{i=1}^{N} \alpha_i y_i \mathbf{k}(\mathbf{x}_i, \mathbf{x}) + b\right), \tag{1}$$

where the $\mathbf{x}_i$, $i = 1, \ldots, N$ are labeled training examples ($y_i \in \{\pm 1\}$).

Recent developments in the literature on SVMs and other kernel methods have shown the need to consider multiple kernels. This provides flexibility and reflects the fact that typical learning problems often involve multiple, heterogeneous data sources. Furthermore, as we shall see below, it leads to an elegant method to interpret the results, which can lead to a deeper understanding of the application.

While this so-called "multiple kernel learning" (MKL) problem can in principle be solved via cross-validation, several recent papers have focused on more efficient methods for multiple kernel learning (Chapelle et al., 2002, Bennett et al., 2002, Grandvalet and Canu, 2003, Ong et al., 2003, Bach et al., 2004, Lanckriet et al., 2004, Bi et al., 2004).

One of the problems with kernel methods compared to other techniques is that the resulting decision function (1) is hard to interpret and, hence, is difficult to use in order to extract relevant knowledge about the problem at hand. One can approach this problem by considering convex combinations of K kernels, i.e.

$$\mathbf{k}(\mathbf{x}_i, \mathbf{x}_j) = \sum_{k=1}^{K} \beta_k \mathbf{k}_k(\mathbf{x}_i, \mathbf{x}_j) \tag{2}$$

with $\beta_k \geq 0$ and $\sum_{k=1}^{K} \beta_k = 1$, where each kernel $\mathbf{k}_k$ uses only a distinct set of features. For appropriately designed sub-kernels $\mathbf{k}_k$, the optimized combination coefficients can then be used to understand which features of the examples are of importance for discrimination: if one is able to obtain an accurate classification by a *sparse* weighting β_k, then one can quite easily interpret the resulting decision function. This is an important property missing in current kernel based algorithms. Note that this is in contrast to the kernel mixture framework of Bennett et al. (2002) and Bi et al. (2004) where each kernel *and* each example are assigned an independent weight and therefore do not offer an easy way to interpret the decision function. We will illustrate that the considered MKL formulation provides useful insights and at the same time is very efficient.

We consider the framework proposed by Lanckriet et al. (2004), which results in a convex optimization problem - a quadratically-constrained quadratic program (QCQP). This problem is more challenging than the standard SVM QP, but it can in principle be solved by general-purpose optimization toolboxes. Since the use of such algorithms will only be feasible for small problems with few data points and kernels, Bach et al. (2004) suggested an algorithm based on sequential minimization optimization (SMO Platt, 1999). While the kernel learning problem is convex, it is also non-smooth, making the direct application of simple local descent algorithms such as SMO infeasible. Bach et al. (2004) therefore considered a smoothed version of the problem to which SMO can be applied.

In the first part of the paper we follow a different direction: We reformulate the binary classification MKL problem (Lanckriet et al., 2004) as a *Semi-Infinite Linear Program*, which can be efficiently solved using an off-the-shelf LP solver and a standard SVM implementation (cf. Section 2.1 for details). In a second step, we show how easily the MKL formulation and the algorithm is generalized to a much larger class of convex loss functions (cf. Section 2.2). Our proposed *wrapper method* works for any kernel and many loss functions: In order to obtain an efficient MKL algorithm for a new loss function, it now suffices to have an LP solver and the corresponding single kernel algorithm (which is assumed to be efficient). Using this

general algorithm we were able to solve MKL problems with up to 30,000 examples and 20 kernels within reasonable time.[1]

We also consider a *Chunking* algorithm that can be considerably more efficient, since it optimizes the SVM $\boldsymbol{\alpha}$ multipliers and the kernel coefficients $\boldsymbol{\beta}$ at the same time. However, for large scale problems it needs to compute and cache the K kernels separately, instead of only one kernel as in the single kernel algorithm. This becomes particularly important when the sample size N is large. If, on the other hand, the number of kernels K is large, then the amount of memory available for caching is drastically reduced and, hence, kernel caching is not effective anymore. (The same statements also apply to the SMO-like MKL algorithm proposed in Bach et al. (2004).)

Since kernel caching cannot help to solve large scale MKL problems, we sought for ways to avoid kernel caching. This is of course not always possible, but it certainly is for the class of kernels where the feature map $\Phi(\mathbf{x})$ can be explicitly computed and computations with $\Phi(\mathbf{x})$ can be implemented efficiently. In Section 3.1.1 we describe several string kernels that are frequently used in biological sequence analysis and exhibit this property. Here, the feature space can be very high dimensional, but $\Phi(\mathbf{x})$ is typically very sparse. In Section 3.1.2 we discuss several methods for efficiently dealing with high dimensional sparse vectors, which not only is of interest for MKL but also for speeding up ordinary SVM classifiers. Finally, we suggest a modification of the previously proposed Chunking algorithm that exploits these properties (Section 3.1.3). In the experimental part we show that the resulting algorithm is more than 70 times faster than the plain Chunking algorithm (for 50,000 examples), even though large kernel caches were used. Also, we were able to solve MKL problems with up to one million examples and 20 kernels and a 10 million real-world splice site classification problem from computational biology. We conclude the paper by illustrating the usefulness of our algorithms in several examples relating to the interpretation of results and to automatic model selection. Moreover, we provide an extensive benchmark study comparing the effect of different improvements on the running time of the algorithms.

We have implemented all algorithms discussed in this work in C++ with interfaces to *Matlab*, *Octave*, *R* and *Python*. The source code is freely available at `http://www.fml.tuebingen.mpg.de/raetsch/projects/shogun`. The examples used to generate the figures are implemented in *Matlab* using the *Matlab* interface of the `SHOGUN` toolbox. They can be found together with the datasets used in this paper at `http://www.fml.tuebingen.mpg.de/raetsch/projects/lsmkl`.

2. A General and Efficient Multiple Kernel Learning Algorithm

In this section we first derive our MKL formulation for the binary classification case and then show how it can be extended to general cost functions. In the last subsection we will propose algorithms for solving the resulting Semi-Infinite Linear Programs (SILPs).

2.1 Multiple Kernel Learning for Classification using SILP

In the Multiple Kernel Learning problem for binary classification one is given N data points $(\mathbf{x}_i, y_i)$ $(y_i \in \{\pm 1\})$, where $\mathbf{x}_i$ is translated via K mappings $\Phi_k(\mathbf{x}) \mapsto \mathbb{R}^{D_k},\ k = 1, \ldots, K$,

1. Results not shown.

from the input into K feature spaces $(\Phi_1(\mathbf{x}_i), \ldots, \Phi_K(\mathbf{x}_i))$ where D_k denotes the dimensionality of the k-th feature space. Then one solves the following optimization problem (Bach et al., 2004), which is equivalent to the linear SVM for $K = 1$:[2]

MKL Primal for Classification

$$\begin{aligned} \min \quad & \frac{1}{2}\left(\sum_{k=1}^{K}\|\mathbf{w}_k\|_2\right)^2 + C\sum_{i=1}^{N}\xi_i \\ \text{w.r.t.} \quad & \mathbf{w}_k \in \mathbb{R}^{D_k}, \boldsymbol{\xi} \in \mathbb{R}^N, b \in \mathbb{R}, \\ \text{s.t.} \quad & \xi_i \geq 0 \text{ and } y_i\left(\sum_{k=1}^{K}\langle \mathbf{w}_k, \Phi_k(\mathbf{x}_i)\rangle + b\right) \geq 1 - \xi_i, \ \ \forall i = 1, \ldots, N \end{aligned} \tag{3}$$

Note that the problem's solution can be written as $\mathbf{w}_k = \beta_k \mathbf{w}'_k$ with $\beta_k \geq 0$, $\forall k = 1, \ldots, K$ and $\sum_{k=1}^{K}\beta_k = 1$ (Bach et al., 2004). Note that therefore the ℓ_1-norm of $\boldsymbol{\beta}$ is constrained to one, while one is penalizing the ℓ_2-norm of $\mathbf{w}_k$ in each block k separately. The idea is that ℓ_1-norm constrained or penalized variables tend to have sparse optimal solutions, while ℓ_2-norm penalized variables do not (e.g. Rätsch, 2001, Chapter 5.2). Thus the above optimization problem offers the possibility to find sparse solutions on the block level with non-sparse solutions within the blocks.

Bach et al. (2004) derived the dual for problem (3). Taking their problem (D_K), squaring the constraints on gamma, multiplying the constraints by $\frac{1}{2}$ and finally substituting $\frac{1}{2}\gamma^2 \mapsto \gamma$ leads to the to the following *equivalent* multiple kernel learning dual:

MKL Dual for Classification

$$\begin{aligned} \min \quad & \gamma - \sum_{i=1}^{N}\alpha_i \\ \text{w.r.t.} \quad & \gamma \in \mathbb{R}, \boldsymbol{\alpha} \in \mathbb{R}^N \\ \text{s.t.} \quad & \mathbf{0} \leq \boldsymbol{\alpha} \leq \mathbf{1}C, \ \ \sum_{i=1}^{N}\alpha_i y_i = 0 \\ & \frac{1}{2}\sum_{i,j=1}^{N}\alpha_i\alpha_j y_i y_j \mathbf{k}_k(\mathbf{x}_i, \mathbf{x}_j) \leq \gamma, \ \ \forall k = 1, \ldots, K \end{aligned}$$

where $\mathbf{k}_k(\mathbf{x}_i, \mathbf{x}_j) = \langle \Phi_k(\mathbf{x}_i), \Phi_k(\mathbf{x}_j)\rangle$. Note that we have one quadratic constraint per kernel $(S_k(\boldsymbol{\alpha}) \leq \gamma)$. In the case of $K = 1$, the above problem reduces to the original SVM dual. We will now move the term $-\sum_{i=1}^{N}\alpha_i$, into the constraints on γ. This can be equivalently done by adding $-\sum_{i=1}^{N}\alpha_i$ to both sides of the constraints and substituting $\gamma - \sum_{i=1}^{N}\alpha_i \mapsto \gamma$:

2. We assume $\mathrm{tr}(K_k) = 1$, $k = 1, \ldots, K$ and set d_j in Bach et al. (2004) to one.

MKL Dual* for Classification

$$\begin{aligned} \min \quad & \gamma \\ \text{w.r.t.} \quad & \gamma \in \mathbb{R}, \boldsymbol{\alpha} \in \mathbb{R}^N \\ \text{s.t.} \quad & \mathbf{0} \leq \boldsymbol{\alpha} \leq \mathbf{1}C, \ \sum_{i=1}^{N} \alpha_i y_i = 0 \\ & \underbrace{\frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j y_i y_j \mathbf{k}_k(\mathbf{x}_i, \mathbf{x}_j) - \sum_{i=1}^{N} \alpha_i}_{=:S_k(\boldsymbol{\alpha})} \leq \gamma, \ \ \forall k = 1, \ldots, K \end{aligned} \tag{4}$$

In order to solve (4), one may solve the following saddle point problem: minimize

$$\mathcal{L} := \gamma + \sum_{k=1}^{K} \beta_k (S_k(\boldsymbol{\alpha}) - \gamma) \tag{5}$$

w.r.t. $\boldsymbol{\alpha} \in \mathbb{R}^N, \gamma \in \mathbb{R}$ (with $\mathbf{0} \leq \boldsymbol{\alpha} \leq C\mathbf{1}$ and $\sum_i \alpha_i y_i = 0$), and maximize it w.r.t. $\boldsymbol{\beta} \in \mathbb{R}^K$, where $\mathbf{0} \leq \boldsymbol{\beta}$. Setting the derivative w.r.t. to γ to zero, one obtains the constraint $\sum_{k=1}^{K} \beta_k = 1$ and (5) simplifies to: $\mathcal{L} = S(\boldsymbol{\alpha}, \boldsymbol{\beta}) := \sum_{k=1}^{K} \beta_k S_k(\boldsymbol{\alpha})$. While one *minimizes* the objective w.r.t. $\boldsymbol{\alpha}$, at the same time one *maximizes* w.r.t. the kernel weighting $\boldsymbol{\beta}$. This leads to a

Min-Max Problem

$$\begin{aligned} \max_{\boldsymbol{\beta}} \min_{\boldsymbol{\alpha}} \quad & \sum_{k=1}^{K} \beta_k S_k(\boldsymbol{\alpha}) \\ \text{w.r.t.} \quad & \boldsymbol{\alpha} \in \mathbb{R}^N, \ \boldsymbol{\beta} \in \mathbb{R}^K \\ \text{s.t.} \quad & 0 \leq \boldsymbol{\alpha} \leq C \ , \ 0 \leq \boldsymbol{\beta}, \sum_{i=1}^{N} \alpha_i y_i = 0 \text{ and } \sum_{k=1}^{K} \beta_k = 1. \end{aligned} \tag{6}$$

This problem is very similar to Equation (9) in Bi et al. (2004) when "composite kernels," i.e. linear combinations of kernels are considered. There the first term of $S_k(\boldsymbol{\alpha})$ has been moved into the constraint, still $\boldsymbol{\beta}$ including the $\sum_{k=1}^{K} \beta_k = 1$ is missing.[3]

Assume $\boldsymbol{\alpha}^*$ were the optimal solution, then $\theta^* := S(\boldsymbol{\alpha}^*, \boldsymbol{\beta})$ would be minimal and, hence, $S(\boldsymbol{\alpha}, \boldsymbol{\beta}) \geq \theta^*$ for all $\boldsymbol{\alpha}$ (subject to the above constraints). Hence, finding a saddle-point of (5) is equivalent to solving the following semi-infinite linear program:

3. In Bi et al. (2004) it is argued that the approximation quality of composite kernels is inferior to mixtures of kernels where a weight is assigned per example *and* kernel as in Bennett et al. (2002). For that reason and as no efficient methods were available to solve the composite kernel problem, they only considered mixtures of kernels and in the experimental validation used a uniform weighting in the composite kernel experiment. Also they did not consider to use composite kernels as a method to interpret the resulting classifier but looked at classification accuracy instead.

Semi-Infinite Linear Program (SILP)

$$\begin{aligned} \max \quad & \theta && (7)\\ \text{w.r.t.} \quad & \theta \in \mathbb{R}, \boldsymbol{\beta} \in \mathbb{R}^K \\ \text{s.t.} \quad & \mathbf{0} \leq \boldsymbol{\beta}, \ \sum_k \beta_k = 1 \text{ and } \sum_{k=1}^K \beta_k S_k(\boldsymbol{\alpha}) \geq \theta && (8)\\ & \text{for all } \boldsymbol{\alpha} \in \mathbb{R}^N \text{ with } \mathbf{0} \leq \boldsymbol{\alpha} \leq C\mathbf{1} \text{ and } \sum_i y_i \alpha_i = 0 \end{aligned}$$

Note that this is a linear program, as θ and $\boldsymbol{\beta}$ are only linearly constrained. However there are infinitely many constraints: one for each $\boldsymbol{\alpha} \in \mathbb{R}^N$ satisfying $0 \leq \boldsymbol{\alpha} \leq C$ and $\sum_{i=1}^N \alpha_i y_i = 0$. Both problems (6) and (7) have the same solution. To illustrate that, consider $\boldsymbol{\beta}$ is fixed and we minimize $\boldsymbol{\alpha}$ in (6). Let $\boldsymbol{\alpha}^*$ be the solution that minimizes (6). Then we can increase the value of θ in (7) as long as none of the infinitely many $\boldsymbol{\alpha}$-constraints (8) is violated, i.e. up to $\theta = \sum_{k=1}^K \beta_k S_k(\boldsymbol{\alpha}^*)$. On the other hand as we increase θ for a fixed $\boldsymbol{\alpha}$ the maximizing $\boldsymbol{\beta}$ is found. We will discuss in Section 2.3 how to solve such semi-infinite linear programs.

2.2 Multiple Kernel Learning with General Cost Functions

In this section we consider a more general class of MKL problems, where one is given an *arbitrary* strictly convex and differentiable loss function, for which we derive its MKL SILP formulation. We will then investigate in this general MKL SILP using different loss functions, in particular the soft-margin loss, the ϵ-insensitive loss and the quadratic loss.

We define the MKL primal formulation for a strictly convex and differentiable loss function $L(f(\mathbf{x}), y)$ as:

MKL Primal for Generic Loss Functions

$$\begin{aligned} \min \quad & \frac{1}{2}\left(\sum_{k=1}^K \|\mathbf{w}_k\|\right)^2 + \sum_{i=1}^N L(f(\mathbf{x}_i), y_i) && (9)\\ \text{w.r.t.} \quad & \mathbf{w} = (\mathbf{w}_1, \ldots, \mathbf{w}_K) \in \mathbb{R}^{D_1} \times \cdots \times \mathbb{R}^{D_K} \\ \text{s.t.} \quad & f(\mathbf{x}_i) = \sum_{k=1}^K \langle \Phi_k(\mathbf{x}_i), \mathbf{w}_k \rangle + b, \ \ \forall i = 1, \ldots, N \end{aligned}$$

In analogy to Bach et al. (2004) we treat problem (9) as a second order cone program (SOCP) leading to the following dual (see Appendix A for the derivation):

MKL Dual* for Generic Loss Functions

$$\begin{aligned} \min \quad & \gamma \\ \text{w.r.t.} \quad & \gamma \in \mathbb{R},\ \boldsymbol{\alpha} \in R^N \\ \text{s.t.} \quad & \sum_{i=1}^N \alpha_i = 0 \quad \text{and} \\ & \frac{1}{2}\left\|\sum_{i=1}^N \alpha_i \Phi_k(\mathbf{x}_i)\right\|_2^2 - \sum_{i=1}^N L(L'^{-1}(\alpha_i, y_i), y_i) + \sum_{i=1}^N \alpha_i L'^{-1}(\alpha_i, y_i) \le \gamma,\ \forall k = 1, \ldots, K \end{aligned} \tag{10}$$

Here L'^{-1} denotes the inverse of the derivative of $L(f(\mathbf{x}), y)$ w.r.t. the prediction $f(\mathbf{x})$. To derive the SILP formulation we follow the same recipe as in Section 2.1: deriving the Lagrangian leads to a max-min problem formulation to be eventually reformulated as a SILP:

SILP for Generic Loss Functions

$$\begin{aligned} \max \quad & \theta \\ \text{w.r.t.} \quad & \theta \in \mathbb{R}, \boldsymbol{\beta} \in \mathbb{R}^K \\ \text{s.t.} \quad & \mathbf{0} \le \boldsymbol{\beta}, \quad \sum_{k=1}^K \beta_k = 1 \quad \text{and} \quad \sum_{k=1}^K \beta_k S_k(\boldsymbol{\alpha}) \ge \theta,\ \forall \boldsymbol{\alpha} \in \mathbb{R}^N,\ \sum_{i=1}^N \alpha_i = 0, \end{aligned} \tag{11}$$

where

$$S_k(\boldsymbol{\alpha}) = -\sum_{i=1}^N L(L'^{-1}(\alpha_i, y_i), y_i) + \sum_{i=1}^N \alpha_i L'^{-1}(\alpha_i, y_i) + \frac{1}{2}\left\|\sum_{i=1}^N \alpha_i \Phi_k(\mathbf{x}_i)\right\|_2^2.$$

We assumed that $L(x, y)$ is strictly convex and differentiable in x. Unfortunately, the soft margin and ϵ-insensitive loss do not have these properties. We therefore consider them separately in the sequel.

Soft Margin Loss We use the following loss in order to approximate the soft margin loss:

$$L_\sigma(x, y) = \frac{C}{\sigma}\log(1 + \exp(\sigma(1 - xy))).$$

It is easy to verify that

$$\lim_{\sigma\to\infty} L_\sigma(x, y) = C(1 - xy)_+.$$

Moreover, L_σ is strictly convex and differentiable for $\sigma < \infty$. Using this loss and assuming $y_i \in \{\pm 1\}$, we obtain (cf. Appendix B.3):

$$S_k(\boldsymbol{\alpha}) = -\sum_{i=1}^N \frac{C}{\sigma}\left(\log\left(\frac{Cy_i}{\alpha_i + Cy_i}\right) + \log\left(-\frac{\alpha_i}{\alpha_i + Cy_i}\right)\right) + \sum_{i=1}^N \alpha_i y_i + \frac{1}{2}\left\|\sum_{i=1}^N \alpha_i \Phi_k(\mathbf{x}_i)\right\|_2^2.$$

If $\sigma \to \infty$, then the first two terms vanish provided that $-C \leq \alpha_i \leq 0$ if $y_i = 1$ and $0 \leq \alpha_i \leq C$ if $y_i = -1$. Substituting $\alpha_i = -\tilde{\alpha}_i y_i$, we obtain

$$S_k(\tilde{\boldsymbol{\alpha}}) = -\sum_{i=1}^{N} \tilde{\alpha}_i + \frac{1}{2} \left\| \sum_{i=1}^{N} \tilde{\alpha}_i y_i \Phi_k(\mathbf{x}_i) \right\|_2^2 \text{ and } \sum_{i=1}^{N} \tilde{\alpha}_i y_i = 0,$$

with $0 \leq \tilde{\alpha}_i \leq C$ $(i = 1, \ldots, N)$ which is the same as (7).

One-Class Soft Margin Loss The one-class SVM soft margin (e.g. Schölkopf and Smola, 2002) is very similar to the two-class case and leads to

$$S_k(\boldsymbol{\alpha}) = \frac{1}{2} \left\| \sum_{i=1}^{N} \alpha_i \Phi_k(\mathbf{x}_i) \right\|_2^2$$

subject to $\mathbf{0} \leq \boldsymbol{\alpha} \leq \frac{1}{\nu N}\mathbf{1}$ and $\sum_{i=1}^{N} \alpha_i = 1$.

ϵ-insensitive Loss Using the same technique for the epsilon insensitive loss $L(x, y) = C(1 - |x - y|)_+$, we obtain

$$\begin{aligned} S_k(\boldsymbol{\alpha}, \boldsymbol{\alpha}^*) \quad &= \quad \frac{1}{2} \left\| \sum_{i=1}^{N} (\alpha_i - \alpha_i^*) \Phi_k(\mathbf{x}_i) \right\|_2^2 - \sum_{i=1}^{N} (\alpha_i + \alpha_i^*)\epsilon - \sum_{i=1}^{N} (\alpha_i - \alpha_i^*) y_i \\ &\text{and} \quad \sum_{i=1}^{N} (\alpha_i - \alpha_i^*) y_i = 0, \text{ with } \mathbf{0} \leq \boldsymbol{\alpha}, \boldsymbol{\alpha}^* \leq C\mathbf{1}. \end{aligned}$$

It is easy to derive the dual problem for other loss functions such as the quadratic loss or logistic loss (see Appendix B.1 & B.2). Note that the dual SILP's only differ in the definition of S_k and the domains of the $\boldsymbol{\alpha}$'s.

2.3 Algorithms to solve SILPs

All *Semi-Infinite Linear Programs* considered in this work have the following structure:

$$\begin{aligned} \max \quad & \theta \\ \text{w.r.t.} \quad & \theta \in \mathbb{R}, \boldsymbol{\beta} \in \mathbb{R}^K \\ \text{s.t.} \quad & \mathbf{0} \leq \boldsymbol{\beta}, \quad \sum_{k=1}^{K} \beta_k = 1 \quad \text{and} \quad \sum_{k=1}^{K} \beta_k S_k(\boldsymbol{\alpha}) \geq \theta \text{ for all } \boldsymbol{\alpha} \in \mathcal{C}. \end{aligned} \tag{12}$$

They have to be optimized with respect to $\boldsymbol{\beta}$ and θ. The constraints depend on definition of S_k and therefore on the choice of the cost function. Using Theorem 5 in Rätsch et al. (2002) one can show that the above SILP has a solution if the corresponding primal is feasible and bounded (see also Hettich and Kortanek, 1993). Moreover, there is no duality gap, if $\mathcal{M} = \text{co}\{[S_1(\boldsymbol{\alpha}), \ldots, S_K(\boldsymbol{\alpha})]^\top \mid \boldsymbol{\alpha} \in \mathcal{C}\}$ is a closed set. For all loss functions considered in this paper this condition is satisfied.

We propose to use a technique called *Column Generation* to solve (12). The basic idea is to compute the optimal $(\boldsymbol{\beta}, \theta)$ in (12) for a restricted subset of constraints. It is called

the *restricted master problem*. Then a second algorithm generates a new, yet unsatisfied constraint determined by $\boldsymbol{\alpha}$. In the best case the other algorithm finds the constraint that maximizes the constraint violation for the given intermediate solution $(\boldsymbol{\beta}, \theta)$, i.e.

$$\boldsymbol{\alpha}_{\boldsymbol{\beta}} := \operatorname*{argmin}_{\boldsymbol{\alpha} \in \mathcal{C}} \sum_k \beta_k S_k(\boldsymbol{\alpha}). \tag{13}$$

If $\boldsymbol{\alpha}_{\boldsymbol{\beta}}$ satisfies the constraint $\sum_{k=1}^K \beta_k S_k(\boldsymbol{\alpha}_{\boldsymbol{\beta}}) \geq \theta$, then the solution $(\theta, \boldsymbol{\beta})$ is optimal. Otherwise, the constraint is added to the set of constraints and the iterations continue.

Algorithm 1 is a special case of a set of SILP algorithms known as *exchange methods*. These methods are known to converge (cf. Theorem 7.2 in Hettich and Kortanek, 1993). However, no convergence rates for such algorithm are known.[4]

Since it is often sufficient to obtain an approximate solution, we have to define a suitable convergence criterion. Note that the problem is solved when all constraints are satisfied. Hence, it is a natural choice to use the normalized maximal constraint violation as a convergence criterion, i.e. the algorithm stops if $\epsilon_{MKL} \geq \epsilon_{MKL}^t := \left| 1 - \frac{\sum_{k=1}^K \beta_k^t S_k(\boldsymbol{\alpha}^t)}{\theta^t} \right|$, where ϵ_{MKL} is an accuracy parameter, $(\boldsymbol{\beta}^t, \theta^t)$ is the optimal solution at iteration $t-1$ and $\boldsymbol{\alpha}^t$ corresponds to the newly found maximally violating constraint of the next iteration.

In the following we will formulate algorithms that alternately optimize the parameters $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$.

2.3.1 A Wrapper Algorithm

The wrapper algorithm (see Algorithm 1) divides the problem into an inner and an outer subproblem. The solution is obtained by alternatively solving the outer problem using the results of the inner problem as input and vice versa until convergence. The outer loop constitutes the *restricted master problem* which determines the optimal $\boldsymbol{\beta}$ for a fixed $\boldsymbol{\alpha}$ using an of-the-shelf linear optimizer. In the inner loop one has to identify unsatisfied constraints, which, fortunately, turns out to be particularly simple. Note that (13) is for all considered cases exactly the dual optimization problem of the single kernel case for fixed $\boldsymbol{\beta}$. For instance for binary classification with soft-margin loss, (13) reduces to the standard SVM dual using the kernel $\mathbf{k}(\mathbf{x}_i, \mathbf{x}_j) = \sum_k \beta_k \mathbf{k}_k(\mathbf{x}_i, \mathbf{x}_j)$:

$$\begin{aligned} v = \quad & \min_{\boldsymbol{\alpha} \in \mathbb{R}^N} \sum_{i,j=1}^N \alpha_i \alpha_j y_i y_j \mathbf{k}(\mathbf{x}_i, \mathbf{x}_j) - \sum_{i=1}^N \alpha_i \\ & \text{s.t.} \quad \mathbf{0} \leq \boldsymbol{\alpha} \leq C\mathbf{1} \text{ and } \sum_{i=1}^N \alpha_i y_i = 0. \end{aligned}$$

Hence, we can use a standard SVM implementation with a single kernel in order to identify the most violated constraint $v \leq \theta$. Since there exists a large number of efficient algorithms to

4. It has been shown that solving semi-infinite problems like (7), using a method related to boosting (e.g. Meir and Rätsch, 2003) one requires at most $T = \mathcal{O}(\log(M)/\hat{\epsilon}^2)$ iterations, where $\hat{\epsilon}$ is the remaining constraint violation and the constants may depend on the kernels and the number of examples N (Rätsch, 2001, Rätsch and Warmuth, 2005, Warmuth et al., 2006). At least for not too small values of $\hat{\epsilon}$ this technique produces reasonably fast good approximate solutions.

Algorithm 1 The MKL-wrapper algorithm optimizes a convex combination of K kernels and employs a linear programming solver to iteratively solve the semi-infinite linear optimization problem (12). The accuracy parameter ϵ_{MKL} is a parameter of the algorithm. $S_k(\boldsymbol{\alpha})$ and $\mathcal{C}$ are determined by the cost function.

$S^0 = 1$, $\theta^1 = -\infty$, $\beta_k^1 = \frac{1}{K}$ for $k = 1, \ldots, K$
for $t = 1, 2, \ldots$ **do**
  Compute $\boldsymbol{\alpha}^t = \operatorname{argmin}_{\boldsymbol{\alpha} \in \mathcal{C}} \sum_{k=1}^{K} \beta_k^t S_k(\boldsymbol{\alpha})$ by single kernel algorithm with $\mathbf{k} = \sum_{k=1}^{K} \beta_k^t \mathbf{k}_k$
  $S^t = \sum_{k=1}^{K} \beta_k^t S_k^t$, where $S_k^t = S_k(\boldsymbol{\alpha}^t)$
  if $\left| 1 - \frac{S^t}{\theta^t} \right| \leq \epsilon_{MKL}$ **then break**
  $(\boldsymbol{\beta}^{t+1}, \theta^{t+1}) = \operatorname{argmax}\ \theta$
    w.r.t. $\boldsymbol{\beta} \in \mathbb{R}^K, \theta \in \mathbb{R}$
    s.t. $\mathbf{0} \leq \boldsymbol{\beta}$, $\sum_{k=1}^{K} \beta_k = 1$ and $\sum_{k=1}^{K} \beta_k S_k^r \geq \theta$ for $r = 1, \ldots, t$
end for

solve the single kernel problems for all sorts of cost functions, we have therefore found an easy way to extend their applicability to the problem of Multiple Kernel Learning. Also, if the kernels are computed on-the-fly within the SVM still only a single kernel cache is required. The wrapper algorithm is very easy to implement, very generic and already reasonably fast for small to medium size problems. However, determining $\boldsymbol{\alpha}$ up to a fixed high precision even for intermediate solutions, while $\boldsymbol{\beta}$ is still far away from the global optimal is unnecessarily costly. Thus there is room for improvement motivating the next section.

2.3.2 A Chunking Algorithm for Simultaneous Optimization of α and β

The goal is to simultaneously optimize $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ in SVM training. Usually it is infeasible to use standard optimization tools (e.g. MINOS, CPLEX, LOQO) for solving even the *SVM training* problems on data sets containing more than a few thousand examples. So-called decomposition techniques as chunking (e.g. used in Joachims, 1998) overcome this limitation by exploiting the special structure of the SVM problem. The key idea of decomposition is to freeze all but a small number of optimization variables (*working set*) and to solve a sequence of constant-size problems (subproblems of the SVM dual).

Here we would like to propose an extension of the chunking algorithm to optimize the kernel weights $\boldsymbol{\beta}$ and the example weights $\boldsymbol{\alpha}$ at the same time. The algorithm is motivated from an insufficiency of the wrapper algorithm described in the previous section: If the $\boldsymbol{\beta}$'s are not optimal yet, then the optimization of the $\boldsymbol{\alpha}$'s until optimality is not necessary and therefore inefficient. It would be considerably faster if for any newly obtained $\boldsymbol{\alpha}$ in the chunking iterations, we could efficiently recompute the optimal $\boldsymbol{\beta}$ and then continue optimizing the $\boldsymbol{\alpha}$'s using the new kernel weighting.

Intermediate Recomputation of $\boldsymbol{\beta}$ Recomputing $\boldsymbol{\beta}$ involves solving a linear program and the problem grows with each additional $\boldsymbol{\alpha}$-induced constraint. Hence, after many it-

Algorithm 2 Outline of the MKL-Chunking algorithm for the classification case (extension to SVM$^{\text{light}}$) that optimizes $\boldsymbol{\alpha}$ and the kernel weighting $\boldsymbol{\beta}$ simultaneously. The accuracy parameter ϵ_{MKL} and the subproblem size Q are assumed to be given to the algorithm. For simplicity we omit the removal of inactive constraints. Also note that from one iteration to the next the LP only differs by one additional constraint. This can usually be exploited to save computing time for solving the LP.

$g_{k,i} = 0$, $\hat{g}_i = 0$, $\alpha_i = 0$, $\beta_k^1 = \frac{1}{K}$ for $k = 1, \ldots, K$ and $i = 1, \ldots, N$
for $t = 1, 2, \ldots$ **do**
  Check optimality conditions and stop if optimal
  select Q suboptimal variables $i_1, \ldots, i_Q$ based on $\hat{\mathbf{g}}$ and $\boldsymbol{\alpha}$
  $\boldsymbol{\alpha}^{old} = \boldsymbol{\alpha}$
  solve SVM dual with respect to the selected variables and update $\boldsymbol{\alpha}$
  $g_{k,i} = g_{k,i} + \sum_{q=1}^{Q} (\alpha_{i_q} - \alpha_{i_q}^{old}) y_{i_q} \mathbf{k}_k(\mathbf{x}_{i_q}, \mathbf{x}_i)$ for all $k = 1, \ldots, M$ and $i = 1, \ldots, N$
  for $k = 1, \ldots, K$ **do**
    $S_k^t = \frac{1}{2} \sum_r g_{k,r} \alpha_r^t y_r - \sum_r \alpha_r^t$
  end for
  $S^t = \sum_{k=1}^{K} \beta_k^t S_k^t$
  if $\left| 1 - \frac{S^t}{\theta^t} \right| \geq \epsilon_{MKL}$
    $(\boldsymbol{\beta}^{t+1}, \theta^{t+1}) = \text{argmax } \theta$
      w.r.t. $\boldsymbol{\beta} \in \mathbb{R}^K, \theta \in \mathbb{R}$
      s.t. $\mathbf{0} \leq \boldsymbol{\beta}, \quad \sum_k \beta_k = 1$ and $\sum_{k=1}^{M} \beta_k S_k^r \geq \theta$ for $r = 1, \ldots, t$
  else
    $\theta^{t+1} = \theta^t$
  end if
  $\hat{g}_i = \sum_k \beta_k^{t+1} g_{k,i}$ for all $i = 1, \ldots, N$
end for

erations solving the LP may become infeasible. Fortunately, there are two facts making it still possible: (a) only a small number of the added constraints remain active and one may as well remove inactive ones — this prevents the LP from growing arbitrarily and (b) for Simplex-based LP optimizers such as `CPLEX` there exists the so-called *hot-start feature* which allows one to efficiently recompute the new solution, if for instance only a few additional constraints are added.

The SVM$^{\text{light}}$ optimizer which we are going to modify, internally needs the output $\hat{g}_i = \sum_{j=1}^{N} \alpha_j y_j \mathbf{k}(\mathbf{x}_i, \mathbf{x}_j)$ for all training examples $i = 1, \ldots, N$ in order to select the next variables for optimization (Joachims, 1999). However, if one changes the kernel weights, then the stored $\hat{g}_i$ values become invalid and need to be recomputed. In order to avoid the full recomputation one has to additionally store a $K \times N$ matrix $g_{k,i} = \sum_{j=1}^{N} \alpha_j y_j \mathbf{k}_k(\mathbf{x}_i, \mathbf{x}_j)$, i.e. the outputs for each kernel separately. If the $\boldsymbol{\beta}$'s change, then $\hat{g}_i$ can be quite efficiently recomputed by $\hat{g}_i = \sum_k \beta_k g_{k,i}$. We implemented the final chunking algorithm for the MKL regression and classification case and display the latter in Algorithm 2.

2.3.3 Discussion

The Wrapper as well as the Chunking algorithm have both their merits: The Wrapper algorithm only relies on the repeated efficient computation of the single kernel solution,

for which typically large scale algorithms exist. The Chunking algorithm is faster, since it exploits the intermediate $\boldsymbol{\alpha}$'s – however, it needs to compute and cache the K kernels separately (particularly important when N is large). If, on the other hand, K is large, then the amount of memory available for caching is drastically reduced and, hence, kernel caching is not effective anymore. The same statements also apply to the SMO-like MKL algorithm proposed in Bach et al. (2004). In this case one is left with the Wrapper algorithm, unless one is able to exploit properties of the particular problem or the sub-kernels (see next section).

3. Sparse Feature Maps and Parallel Computations

In this section we discuss two strategies to accelerate SVM training. First we consider the case where the explicit mapping Φ into the kernel feature space is known as well as sparse. For this case we show that MKL training (and also SVM training in general) can be made drastically faster, in particular, when N and K are large. In the second part we discuss a simple, yet efficient way to parallelize MKL as well as SVM training.

3.1 Explicit computations with Sparse Feature Maps

We assume that all K sub-kernels are given as

$$\mathbf{k}_k(\mathbf{x}, \mathbf{x}') = \langle \Phi_k(\mathbf{x}), \Phi_k(\mathbf{x}') \rangle$$

and the mappings Φ_k are given explicitly ($k = 1, \ldots, K$). Moreover, we suppose that the mapped examples $\Phi_k(\mathbf{x})$ are very sparse. We start by giving examples of such kernels and discuss two kernels that are often used in biological sequence analysis (Section 3.1.1). In Section 3.1.2 we discuss several strategies for efficiently storing and computing with high dimensional sparse vectors (in particular for these two kernels). Finally in Section 3.1.3 we discuss how we can exploit these properties to accelerate chunking algorithms, such as SVM$^{\text{light}}$, by a factor of up to Q (the chunking subproblem size).

3.1.1 String Kernels

The Spectrum Kernel The spectrum kernel (Leslie et al., 2002) implements the n-gram or bag-of-words kernel (Joachims, 1998) as originally defined for text classification in the context of biological sequence analysis. The idea is to count how often a d-mer (a contiguous string of length d) is contained in the sequences $\mathbf{x}$ and $\mathbf{x}'$. Summing up the product of these counts for every possible d-mer (note that there are exponentially many) gives rise to the kernel value which formally is defined as follows: Let Σ be an alphabet and $\boldsymbol{u} \in \Sigma^d$ a d-mer and $\#\boldsymbol{u}(\mathbf{x})$ the number of occurrences of $\boldsymbol{u}$ in $\mathbf{x}$. Then the spectrum kernel is defined as the inner product of $\mathbf{k}(\mathbf{x}, \mathbf{x}') = \langle \Phi(\mathbf{x}), \Phi(\mathbf{x}') \rangle$, where $\Phi(\mathbf{x}) = (\#\boldsymbol{u}(\mathbf{x}))_{\boldsymbol{u} \in \Sigma^d}$. Note that spectrum-like kernels cannot extract any positional information from the sequence which goes beyond the d-mer length. It is well suited for describing the content of a sequence but is less suitable for instance for analyzing signals where motifs may appear in a certain order or at specific positions. Also note that spectrum-like kernels are capable of dealing with sequences with varying length.

The spectrum kernel can be efficiently computed in $\mathcal{O}(d(|\mathbf{x}| + |\mathbf{x}'|))$ using tries (Leslie et al., 2002), where $|\mathbf{x}|$ denotes the length of sequence $\mathbf{x}$. An easier way to compute the kernel for two sequences $\mathbf{x}$ and $\mathbf{x}'$ is to separately extract and sort the N d-mers in each sequence, which can be done in a preprocessing step. Note that for instance DNA d-mers of length $d \leq 16$ can be efficiently represented as a 32-bit integer value. Then one iterates over all d-mers of sequences $\mathbf{x}$ and $\mathbf{x}'$ simultaneously and counts which d-mers appear in both sequences and sums up the product of their counts. The computational complexity of the kernel computation is $\mathcal{O}(\log(|\Sigma|)d(|\mathbf{x}| + |\mathbf{x}'|))$.

The Weighted Degree Kernel The so-called *weighted degree* (WD) kernel (Rätsch and Sonnenburg, 2004) efficiently computes similarities between sequences while taking positional information of k-mers into account. The main idea of the WD kernel is to count the (exact) co-occurrences of k-mers at corresponding positions in the two sequences to be compared. The *WD kernel of order d* compares two sequences $\mathbf{x}_i$ and $\mathbf{x}_j$ of length L by summing all contributions of k-mer matches of lengths $k \in \{1, \ldots, d\}$, weighted by coefficients β_k:

$$\mathbf{k}(\mathbf{x}_i, \mathbf{x}_j) = \sum_{k=1}^{d} \beta_k \sum_{l=1}^{L-k+1} \mathbf{I}(\boldsymbol{u}_{k,l}(\mathbf{x}_i) = \boldsymbol{u}_{k,l}(\mathbf{x}_j)). \tag{14}$$

Here, $\boldsymbol{u}_{k,l}(\mathbf{x})$ is the string of length k starting at position l of the sequence $\mathbf{x}$ and $\mathbf{I}(\cdot)$ is the indicator function which evaluates to 1 when its argument is *true* and to 0 otherwise. For the weighting coefficients, Rätsch and Sonnenburg (2004) proposed to use $\beta_k = 2\frac{d-k+1}{d(d+1)}$. Matching substrings are thus rewarded with a score depending on the length of the substring.[5]

Note that the WD kernel can be understood as a Spectrum kernel where the k-mers starting at different positions are treated independently of each other.[6] Moreover, it does not only consider substrings of length exactly d, but also all shorter matches. Hence, the feature space for each position has $\sum_{k=1}^{d} |\Sigma|^k = \frac{|\Sigma|^{d+1}-1}{|\Sigma|-1} - 1$ dimensions and is additionally duplicated L times (leading to $\mathcal{O}(L|\Sigma|^d)$ dimensions). However, the computational complexity of the WD kernel is in the worst case $\mathcal{O}(dL)$ as can be directly seen from (14).

3.1.2 Efficient Storage of Sparse Weights

The considered string kernels correspond to a feature space that can be huge. For instance in the case of the WD kernel on DNA sequences of length 100 with $K = 20$, the corresponding feature space is 10^{14} dimensional. However, most dimensions in the feature space are not used since only a few of the many different k-mers actually appear in the sequences. In this section we briefly discuss three methods to efficiently deal with sparse vectors $\boldsymbol{v}$. We assume that the elements of the vector $\boldsymbol{v}$ are indexed by some index set $\mathcal{U}$ (for sequences, e.g. $\mathcal{U} = \Sigma^d$) and that we only need three operations: `clear`, `add` and `lookup`. The first

5. Note that although in our case $\beta_{k+1} < \beta_k$, longer matches nevertheless contribute more strongly than shorter ones: this is due to the fact that each long match also implies several short matches, adding to the value of (14). Exploiting this knowledge allows for a $\mathcal{O}(L)$ reformulation of the kernel using "block-weights" as has been done in Sonnenburg et al. (2005b).
6. It therefore is very position dependent and does not tolerate any positional "shift". For that reason we proposed in Rätsch et al. (2005) a WD kernel *with shifts*, which tolerates a small number of shifts, that lies in between the WD and the Spectrum kernel.

operation sets the vector $\boldsymbol{v}$ to zero, the `add` operation increases the weight of a dimension for an element $\boldsymbol{u} \in \mathcal{U}$ by some amount α, i.e. $v_{\boldsymbol{u}} = v_{\boldsymbol{u}} + \alpha$ and `lookup` requests the value $v_{\boldsymbol{u}}$. The latter two operations need to be performed as quickly as possible (whereas the performance of the `lookup` operation is of higher importance).

Explicit Map If the dimensionality of the feature space is small enough, then one might consider keeping the whole vector $\boldsymbol{v}$ in memory and to perform direct operations on its elements. Then each read or write operation is $\mathcal{O}(1)$.[7] This approach has expensive memory requirements ($\mathcal{O}(|\Sigma|^d)$), but is very fast and best suited for instance for the Spectrum kernel on DNA sequences with $d \leq 14$ and on protein sequences with $d \leq 6$.

Sorted Arrays More memory efficient but computationally more expensive are sorted arrays of index-value pairs $(\boldsymbol{u}, v_{\boldsymbol{u}})$. Assuming the L indexes are given and sorted in advance, one can efficiently change or look up a single $v_{\boldsymbol{u}}$ for a corresponding $\boldsymbol{u}$ by employing a binary search procedure ($\mathcal{O}(\log(L))$). When given L' look up indexes at once, one may sort them in advance and then simultaneously traverse the two arrays in order to determine which elements appear in the first array (i.e. $\mathcal{O}(L + L')$ operations – omitting the sorting of the second array – instead of $\mathcal{O}(\log(L)L')$). This method is well suited for cases where L and L' are of comparable size, as for instance for computations of single Spectrum kernel elements (as proposed in Leslie et al., 2004). If, $L \gg L'$, then the binary search procedure should be preferred.

Tries Another way of organizing the non-zero elements are *tries* (Fredkin, 1960): The idea is to use a tree with at most $|\Sigma|$ siblings of depth d. The leaves store a single value: the element $v_{\boldsymbol{u}}$, where $\boldsymbol{u} \in \Sigma^d$ is a d-mer and the path to the leaf corresponds to $\boldsymbol{u}$.

To `add` or `lookup` an element one only needs d operations to reach a leaf of the tree (and to create necessary nodes on the way in an `add` operation). Note that the worst-case computational complexity of the operations is independent of the number of d-mers/elements stored in the tree.

While tries are not faster than *Sorted Arrays* in `lookup` and need considerably more storage (e.g. for pointers to its parent and siblings), they are useful for the previously discussed WD kernel. Here we not only have to lookup one substring $\boldsymbol{u} \in \Sigma^d$, but also all prefixes of $\boldsymbol{u}$. For *Sorted Arrays* this amounts to d separate `lookup` operations, while for tries all prefixes of $\boldsymbol{u}$ are already known when the bottom of the tree is reached. In this case the trie has to store weights also on the internal nodes. This is illustrated for the WD kernel in Figure 1.

3.1.3 Speeding up SVM Training

As it is not feasible to use standard optimization toolboxes for solving large scale SVM training problem, decomposition techniques are used in practice. Most chunking algorithms work by first selecting Q variables (working set $W \subseteq \{1, \ldots, N\}$, $Q := |W|$) based on the current solution and then solve the reduced problem with respect to the working set variables. These two steps are repeated until some optimality conditions are satisfied (see e.g. Joachims (1998)). For selecting the working set and checking the termination criteria in each iteration,

7. More precisely, it is $\log d$, but for small enough d (which we have to assume anyway) the computational effort is exactly one memory access.

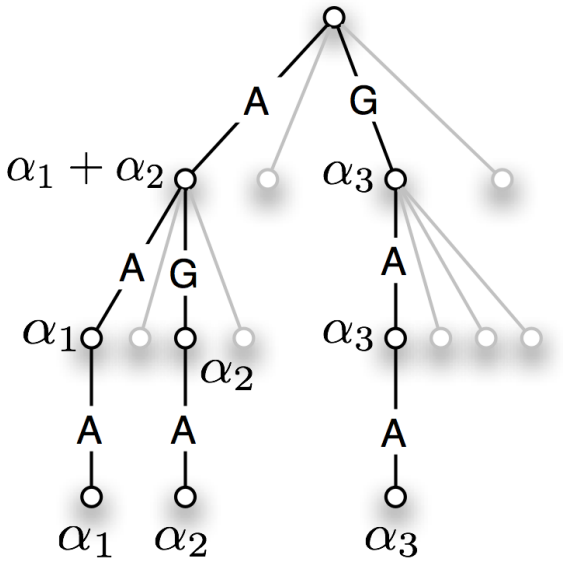

Figure 1: Three sequences AAA, AGA, GAA with weights α_1, α_2 & α_3 are added to the trie. The figure displays the resulting weights at the nodes.

the vector $\mathbf{g}$ with $g_i = \sum_{j=1}^{N} \alpha_j y_j \mathbf{k}(x_i, x_j), \quad i = 1, \ldots, N$ is usually needed. Computing $\mathbf{g}$ from scratch in every iteration which would require $\mathcal{O}(N^2)$ kernel computations. To avoid recomputation of $\mathbf{g}$ one typically starts with $\mathbf{g} = \mathbf{0}$ and only computes updates of $\mathbf{g}$ on the working set W

$$g_i \leftarrow g_i^{old} + \sum_{j \in W} (\alpha_j - \alpha_j^{old}) y_j \mathbf{k}(x_i, x_j), \quad \forall i = 1, \ldots, N.$$

As a result the effort decreases to $\mathcal{O}(QN)$ kernel computations, which can be further speed up by using kernel caching (e.g. Joachims, 1998). However kernel caching is not efficient enough for large scale problems[8] and thus most time is spend computing kernel rows for the updates of $\mathbf{g}$ on the working set W. Note however that this update as well as computing the Q kernel rows can be easily parallelized; cf. Section 4.2.1.

Exploiting $\mathbf{k}(\mathbf{x}_i, \mathbf{x}_j) = \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle$ and $\mathbf{w} = \sum_{i=1}^{N} \alpha_i y_i \Phi(\mathbf{x}_i)$ we can rewrite the update rule as

$$g_i \leftarrow g_i^{old} + \sum_{j \in W} (\alpha_j - \alpha_j^{old}) y_j \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle = g_i^{old} + \langle \mathbf{w}^W, \Phi(\mathbf{x}_i) \rangle, \tag{15}$$

where $\mathbf{w}^W = \sum_{j \in W} (\alpha_j - \alpha_j^{old}) y_j \Phi(\mathbf{x}_j)$ is the normal (update) vector on the working set.

If the kernel feature map can be computed explicitly and is sparse (as discussed before), then computing the update in (15) can be accelerated. One only needs to compute and store

8. For instance when using a million examples one can only fit 268 rows into 1 GB. Moreover, caching 268 rows is insufficient when for instance having many thousands of active variables.

$\mathbf{w}^W$ (using the `clear` and $\sum_{q\in W}|\{\Phi_j(\mathbf{x}_q)\neq 0\}|$ `add` operations) and performing the scalar product $\langle \mathbf{w}^W, \Phi(\mathbf{x}_i)\rangle$ (using $|\{\Phi_j(\mathbf{x}_i)\neq 0\}|$ `lookup` operations).

Depending on the kernel, the way the sparse vectors are stored Section 3.1.2 and on the sparseness of the feature vectors, the speedup can be quite drastic. For instance for the WD kernel one kernel computation requires $\mathcal{O}(Ld)$ operations (L is the length of the sequence). Hence, computing (15) N times requires $\mathrm{O}(NQLd)$ operations. When using tries, then one needs QL `add` operations (each $\mathcal{O}(d)$) and NL `lookup` operations (each $\mathcal{O}(d)$). Therefore only $\mathcal{O}(QLd+NLd)$ basic operations are needed in total. When N is large enough it leads to a speedup by a factor of Q. Finally note that kernel caching is no longer required and as Q is small in practice (e.g. $Q=42$) the resulting trie has rather few leaves and thus only needs little storage.

The pseudo-code of our `linadd` SVM chunking algorithm is given in Algorithm 3.

Algorithm 3 Outline of the chunking algorithm that exploits the fast computations of linear combinations of kernels (e.g. by tries).

{INITIALIZATION}
$g_i=0$, $\alpha_i=0$ for $i=1,\ldots,N$
{LOOP UNTIL CONVERGENCE}
for $t=1,2,\ldots$ **do**
  Check optimality conditions and stop if optimal
  select working set W based on $\mathbf{g}$ and $\boldsymbol{\alpha}$, store $\boldsymbol{\alpha}^{old}=\boldsymbol{\alpha}$
  solve reduced problem W and update $\boldsymbol{\alpha}$

  `clear` $\mathbf{w}$
  $\mathbf{w}\leftarrow \mathbf{w}+(\alpha_j-\alpha_j^{old})y_j\Phi(\mathbf{x}_j)$ for all $j\in W$ (using `add`)
  update $g_i=g_i+\langle \mathbf{w},\Phi(\mathbf{x}_i)\rangle$ for all $i=1,\ldots,N$ (using `lookup`)
end for

MKL Case As elaborated in Section 2.3.2 and Algorithm 2, for MKL one stores K vectors $\mathbf{g}_k,\quad k=1,\ldots,K$: one for each kernel in order to avoid full recomputation of $\hat{\mathbf{g}}$ if a kernel weight β_k is updated. Thus to use the idea above in Algorithm 2 all one has to do is to store K normal vectors (e.g. tries)

$$\mathbf{w}_k^W=\sum_{j\in W}(\alpha_j-\alpha_j^{old})y_j\Phi_k(\mathbf{x}_j),\quad k=1,\ldots,K$$

which are then used to update the $K\times N$ matrix $g_{k,i}=g_{k,i}^{old}+\langle \mathbf{w}_k^W,\Phi_k(\mathbf{x}_i)\rangle$ (for all $k=1\ldots K$ and $i=1\ldots N$) by which $\hat{g}_i=\sum_k\beta_k g_{k,i}$, (for all $i=1\ldots N$) is computed.

3.2 A Simple Parallel Chunking Algorithm

As still most time is spent in evaluating $g(\mathbf{x})$ for all training examples further speedups are gained when parallelizing the evaluation of $g(\mathbf{x})$. When using the `linadd` algorithm, one first constructs the trie (or any of the other possible more appropriate data structures) and then performs parallel `lookup` operations using several CPUs (e.g. using shared memory or several copies of the data structure on separate computing nodes). We have implemented this

algorithm based on multiple *threads* (using shared memory) and gain reasonable speedups (see next section).

Note that this part of the computations is almost ideal to distribute to many CPUs, as only the updated $\boldsymbol{\alpha}$ (or $\mathbf{w}$ depending on the communication costs and size) have to be transfered before each CPU computes a large chunk $I_k \subset \{1,\dots,N\}$ of

$$h_i^{(k)} = \langle \mathbf{w}, \Phi(\mathbf{x}_i) \rangle, \quad \forall i \in I_k, \quad \forall k = 1,\dots,N, \text{ where } (I_1 \cup \cdots \cup I_n) = (1,\dots,N)$$

which is transfered to a master node that finally computes $\mathbf{g} \leftarrow \mathbf{g} + \mathbf{h}$, as illustrated in Algorithm 4.

Algorithm 4 Outline of the parallel chunking algorithm that exploits the fast computations of linear combinations of kernels.

{ **Master node** }
{INITIALIZATION}
$g_i = 0$, $\alpha_i = 0$ for $i = 1,\dots,N$
{LOOP UNTIL CONVERGENCE}
for $t = 1, 2, \dots$ **do**
  Check optimality conditions and stop if optimal
  select working set W based on $\mathbf{g}$ and $\boldsymbol{\alpha}$, store $\boldsymbol{\alpha}^{old} = \boldsymbol{\alpha}$
  solve reduced problem W and update $\boldsymbol{\alpha}$
  transfer to Slave nodes: $\alpha_j - \alpha_j^{old}$ for all $j \in W$
  fetch from n Slave nodes: $\mathbf{h} = (\mathbf{h}^{(1)}, \dots, \mathbf{h}^{(n)})$
  update $g_i = g_i + h_i$ for all $i = 1,\dots,N$
end for
signal convergence to slave nodes

{ **Slave nodes** }
{LOOP UNTIL CONVERGENCE}
while not converged **do**
  fetch from Master node $\alpha_j - \alpha_j^{old}$ for all $j \in W$
  `clear` **w**
  $\mathbf{w} \leftarrow \mathbf{w} + (\alpha_j - \alpha_j^{old}) y_j \Phi(\mathbf{x}_j)$ for all $j \in W$ (using `add`)
  node k computes $h_i^{(k)} = \langle \mathbf{w}, \Phi(\mathbf{x}_i) \rangle$
    for all $i = (k-1)\frac{N}{n}, \dots, k\frac{N}{n} - 1$ (using `lookup`)
  transfer to master: $\mathbf{h}^{(k)}$
end while

4. Results and Discussion

4.1 MKL for Knowledge Discovery

In this section we will discuss toy examples for binary classification and regression, demonstrating that MKL can recover information about the problem at hand, followed by a brief review on problems for which MKL has been successfully used.

4.1.1 Classification

The first example we deal with is a binary classification problem. The task is to separate two concentric classes shaped like the outline of stars. By varying the distance between the boundary of the stars we can control the separability of the problem. Starting with a non-separable scenario with zero distance, the data quickly becomes separable as the distance between the stars increases, and the boundary needed for separation will gradually tend towards a circle. In Figure 2 three scatter plots of data sets with varied separation distances are displayed.

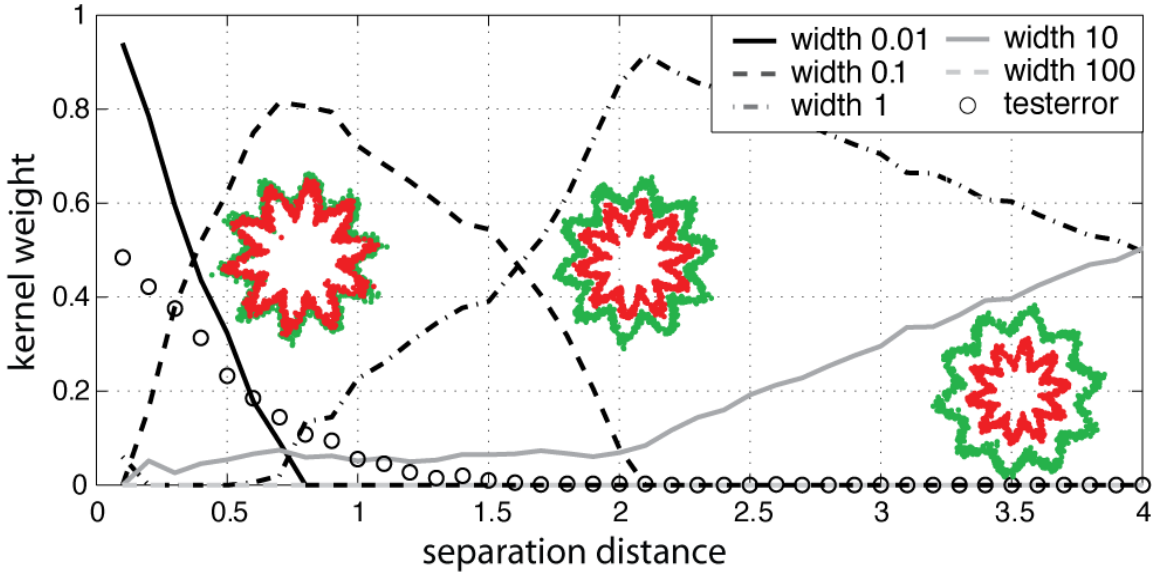

Figure 2: A 2-class toy problem where the dark gray star-like shape is to be distinguished from the light gray star inside of the dark gray star. The distance between the dark star-like shape and the light star increases from the left to the right.

We generate several training and test sets for a wide range of distances (the radius of the inner star is fixed at 4.0, the outer stars radius is varied from $4.1 \ldots 9.9$). Each dataset contains 2,000 observations (1,000 positive and 1,000 negative) using a moderate noise level (Gaussian noise with zero mean and standard deviation 0.3). The MKL-SVM was trained for different values of the regularization parameter C, where we set $\epsilon_{MKL} = 10^{-3}$. For every value of C we averaged the test errors of all setups and choose the value of C that led to the smallest overall error ($C = 0.5$).[9]

The choice of the kernel width of the Gaussian RBF (below, denoted by RBF) kernel used for classification is expected to depend on the separation distance of the learning problem: An increased distance between the stars will correspond to a larger optimal kernel width. This effect should be visible in the results of the MKL, where we used MKL-SVMs with five RBF kernels with different widths ($2\sigma^2 \in \{0.01, 0.1, 1, 10, 100\}$). In Figure 2 we

9. Note that we are aware of the fact that the test error might be slightly underestimated.

show the obtained kernel weightings for the five kernels and the test error (circled line) which quickly drops to zero as the problem becomes separable. Every column shows one MKL-SVM weighting. The courses of the kernel weightings reflect the development of the learning problem: as long as the problem is difficult the best separation can be obtained when using the kernel with smallest width. The low width kernel looses importance when the distance between the stars increases and larger kernel widths obtain a larger weight in MKL. Increasing the distance between the stars, kernels with greater widths are used. Note that the RBF kernel with largest width was not appropriate and thus never chosen. This illustrates that MKL can indeed recover information about the structure of the learning problem.

4.1.2 Regression

We applied the newly derived MKL support vector regression formulation to the task of learning a sine function using three RBF-kernels with different widths ($2\sigma^2 \in \{0.005, 0.05, 0.5, 1, 10\}$). To this end, we generated several data sets with increasing frequency of the sine wave. The sample size was chosen to be 1,000. Analogous to the procedure described above we choose the value of $C = 10$, minimizing the overall test error. In Figure 3 exemplarily three sine waves are depicted, where the frequency increases from left to right. For every frequency the computed weights for each kernel width are shown. One can see that MKL-SV regression switches to the width of the RBF-kernel fitting the regression problem best.

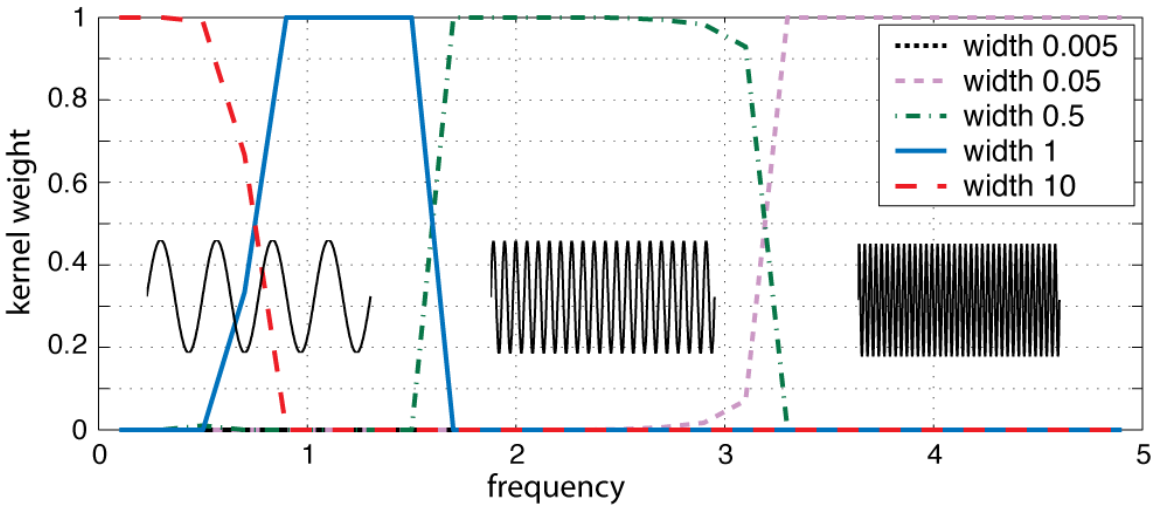

Figure 3: MKL-Support Vector Regression for the task of learning a sine wave (please see text for details).

In another regression experiment, we combined a linear function with two sine waves, one of lower frequency and one of high frequency, i.e. $f(x) = \sin(ax) + \sin(bx) + cx$. Furthermore we increase the frequency of the higher frequency sine wave, i.e. we varied a leaving b and c unchanged. The MKL weighting should show a combination of different kernels. Using

ten RBF-kernels of different width (see Figure 4) we trained a MKL-SVR and display the learned weights (a column in the figure). Again the sample size is 1,000 and one value for $C = 5$ is chosen via a previous experiment ($\epsilon_{MKL} = 10^{-5}$). The largest selected width (100) models the linear component (since RBF kernels with large widths are effectively linear) and the medium width (1) corresponds to the lower frequency sine. We varied the frequency of the high frequency sine wave from low to high (left to right in the figure). One observes that MKL determines an appropriate combination of kernels of low and high widths, while decreasing the RBF kernel width with increased frequency. Additionally one can observe that MKL leads to sparse solutions since most of the kernel weights in Figure 4 are depicted in blue, that is they are zero.[10]

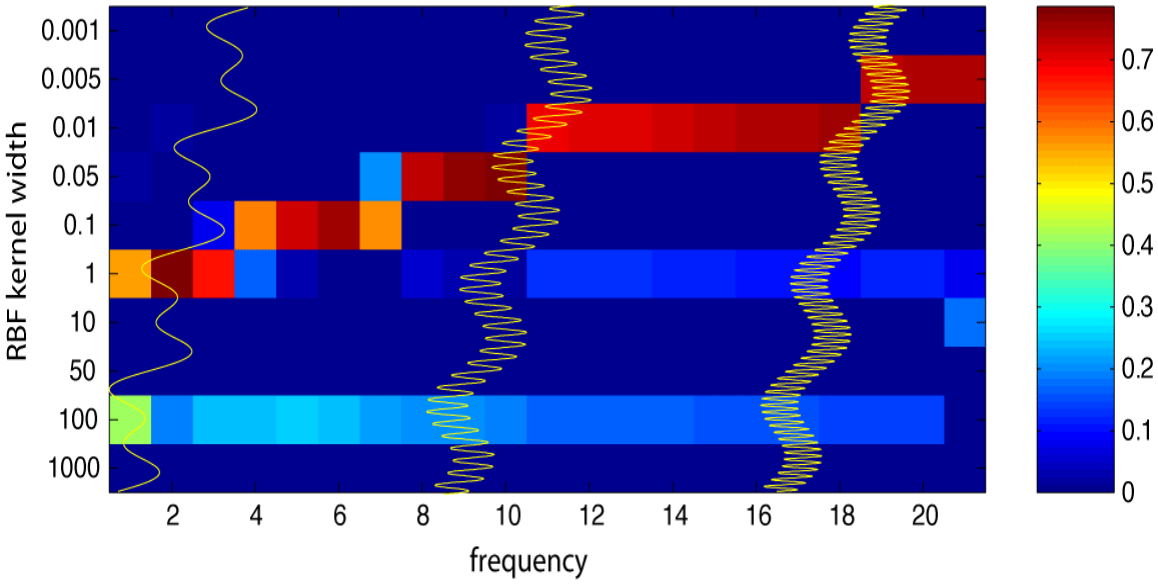

Figure 4: MKL support vector regression on a linear combination of three functions: $f(x) = \sin(ax) + \sin(bx) + cx$. MKL recovers that the original function is a combination of functions of low and high complexity. For more details see text.

4.1.3 Real World Applications in Bioinformatics

MKL has been successfully used on real-world datasets in the field of computational biology (Lanckriet et al., 2004, Sonnenburg et al., 2005a). It was shown to improve classification performance on the task of ribosomal and membrane protein prediction (Lanckriet et al., 2004), where a weighting over different kernels each corresponding to a different feature set was learned. In their result, the included random channels obtained low kernel weights. However, as the data sets was rather small ($\approx 1,000$ examples) the kernel matrices could be precomputed and simultaneously kept in memory, which was not possible in Sonnenburg

10. The training time for MKL-SVR in this setup but with 10,000 examples was about 40 minutes, when kernel caches of size 100MB are used

et al. (2005a), where a splice site recognition task for the worm *C. elegans* was considered. Here data is available in abundance (up to one million examples) and larger amounts are indeed needed to obtain state of the art results (Sonnenburg et al., 2005b).[11] On that dataset we were able to solve the classification MKL SILP for $N = 1,000,000$ examples and $K = 20$ kernels, as well as for $N = 10,000$ examples and $K = 550$ kernels, using the `linadd` optimizations with the weighted degree kernel. As a result we a) were able to learn the weighting $\boldsymbol{\beta}$ instead of choosing a heuristic and b) were able to use MKL as a tool for interpreting the SVM classifier as in Sonnenburg et al. (2005a), Rätsch et al. (2005).

As an example we learned the weighting of a WD kernel of degree 20, which consist of a weighted sum of 20 sub-kernels each counting matching d-mers, for $d = 1, \ldots, 20$. The

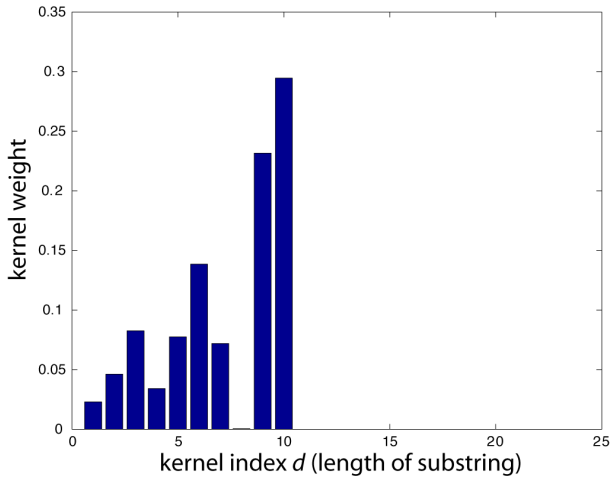

Figure 5: The learned WD kernel weighting on a million of examples.

learned weighting is displayed in Figure 5 and shows a peak for 6-mers and 9&10-mers. It should be noted that the obtained weighting in this experiment is only partially useful for interpretation. In the case of splice site detection, it is unlikely that k-mers of length 9 or 10 are playing the most important role. More likely to be important are substrings of length up to six. We believe that the large weights for the longest k-mers are an artifact which comes from the fact that we are combining kernels with quite different properties, i.e. the 9th and 10th kernel leads to a combined kernel matrix that is most diagonally dominant (since the sequences are only similar to themselves but not to other sequences), which we believe is the reason for having a large weight.[12]

In the following example we consider one weight per position. In this case the combined kernels are more similar to each other and we expect more interpretable results. Figure 6 shows an importance weighting for each position in a DNA sequence (around a so called acceptor splice site, the start of an exon). We used MKL on 65,000 examples to compute

11. In Section 4.2 we will use a *human* splice dataset containing 15 million examples, and train WD kernel based SVM classifiers on up to 10 million examples using the parallelized `linadd` algorithm.
12. This problem might be partially alleviated by including the identity matrix in the convex combination. However as 2-norm soft margin SVMs can be implemented by adding a constant to the diagonal of the kernel (Cortes and Vapnik, 1995), this leads to an additional 2-norm penalization.

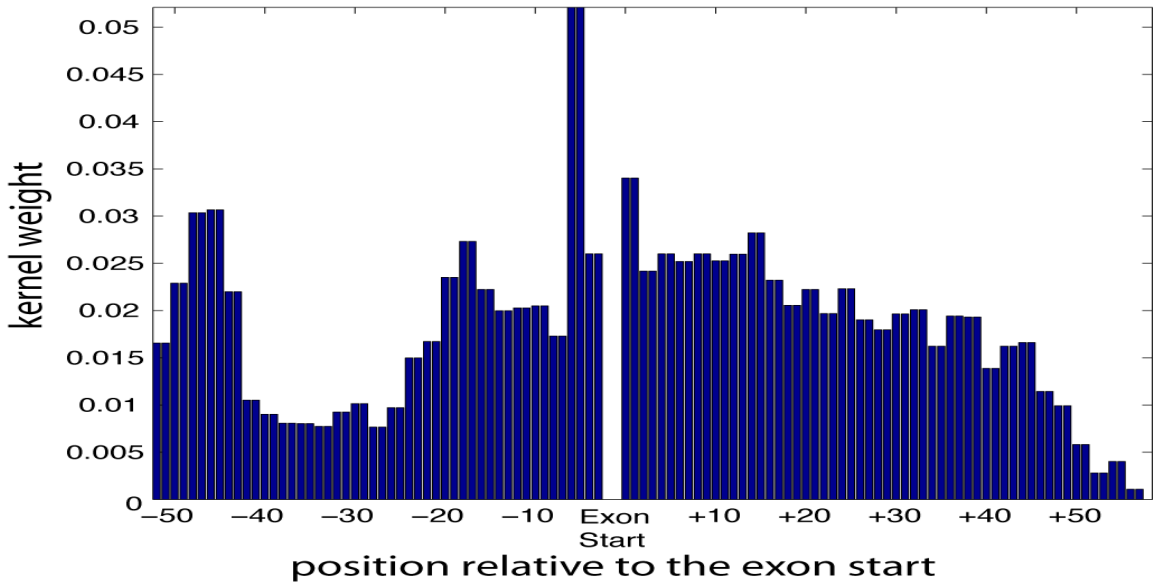

Figure 6: The figure shows an importance weighting for each position in a DNA sequence (around a so called splice site). MKL was used to determine these weights, each corresponding to a sub-kernel which uses information at that position to discriminate splice sites from non-splice sites. Different peaks correspond to different biologically known signals (see text for details). We used 65,000 examples for training with 54 sub-kernels.

these 54 weights, each corresponding to a sub-kernel which uses information at that position to discriminate true splice sites from fake ones. We repeated that experiment on ten bootstrap runs of the dataset. We can identify several interesting regions that we can match to current biological knowledge about splice site recognition: a) The region −50 nucleotides (nt) to −40nt, which corresponds to the donor splice site of the previous exon (many introns in *C. elegans* are very short, often only 50nt), b) the region −25nt to −15nt that coincides with the location of the branch point, c) the intronic region closest to the splice site with greatest weight (−8nt to −1nt; the weights for the `AG` dimer are zero, since it appears in splice sites and decoys) and d) the exonic region (0nt to +50nt). Slightly surprising are the high weights in the exonic region, which we suspect only model triplet frequencies. The decay of the weights seen from +15nt to +45nt might be explained by the fact that not all exons are actually long enough. Furthermore, since the sequence ends in our case at +60nt, the decay after +45nt is an edge effect as longer substrings cannot be matched.

4.2 Benchmarking the Algorithms

Experimental Setup To demonstrate the effect of the several proposed algorithmic optimizations, namely the `linadd` SVM training (Algorithm 3) and for MKL the SILP formulation with and without the `linadd` extension for single, four and eight CPUs, we applied

each of the algorithms to a *human* splice site data set[13], comparing it to the original WD formulation and the case where the weighting coefficients were learned using Multiple Kernel Learning. The splice data set contains 159,771 true acceptor splice site sequences and 14,868,555 decoys, leading to a total of 15,028,326 sequences each 141 base pairs in length. It was generated following a procedure similar to the one in Sonnenburg et al. (2005a) for *C. elegans* which however contained "only" 1,026,036 examples. Note that the dataset is very unbalanced as 98.94% of the examples are negatively labeled. We are using this data set in all benchmark experiments and trained (MKL-)SVMs using the `SHOGUN` machine learning toolbox which contains a modified version of SVM$^{\text{light}}$ (Joachims, 1999) on 500, $1,000$, $5,000$, $10,000$, $30,000$, $50,000$, $100,000$, $200,000$, $500,000$, $1,000,000$, $2,000,000$, $5,000,000$ and $10,000,000$ randomly sub-sampled examples and measured the time needed in SVM training. For classification performance evaluation we always use the same remaining 5,028,326 examples as a test data set. We set the degree parameter to $d = 20$ for the WD kernel and to $d = 8$ for the spectrum kernel fixing the SVMs regularization parameter to $C = 5$. Thus in the MKL case also $K = 20$ sub-kernels were used. SVM$^{\text{light}}$'s subproblem size (parameter `qpsize`), convergence criterion (parameter `epsilon`) and MKL convergence criterion were set to $Q = 112$, $\epsilon_{SVM} = 10^{-5}$ and $\epsilon_{MKL} = 10^{-5}$, respectively. A kernel cache of 1GB was used for all kernels except the precomputed kernel and algorithms using the `linadd`-SMO extension for which the kernel-cache was disabled. Later on we measure whether changing the quadratic subproblem size Q influences SVM training time. Experiments were performed on a PC powered by *eight* 2.4GHz AMD Opteron(tm) processors running Linux. We measured the training time for each of the algorithms (single, quad or eight CPU version) and data set sizes.

4.2.1 Benchmarking SVM

The obtained training times for the different SVM algorithms are displayed in Table 1 and in Figure 7. First, SVMs were trained using standard SVM$^{\text{light}}$ with the Weighted Degree Kernel precomputed (*WDPre*), the standard WD kernel (*WD1*) and the precomputed (*SpecPre*) and standard spectrum kernel (*Spec*). Then SVMs utilizing the `linadd` extension[14] were trained using the WD (*LinWD*) and spectrum (*LinSpec*) kernel. Finally SVMs were trained on four and eight CPUs using the parallel version of the `linadd` algorithm (*LinWD4, LinWD8*). *WD4* and *WD8* demonstrate the effect of a simple parallelization strategy where the computation of kernel rows and updates on the working set are parallelized, which works with *any* kernel.

The training times obtained when precomputing the kernel matrix (which includes the time needed to precompute the full kernel matrix) is lower when no more than $1,000$ examples are used. Note that this is a direct cause of the relatively large subproblem size $Q = 112$. The picture is different for, say, $Q = 42$ (data not shown) where the *WDPre* training time is in all cases larger than the times obtained using the original WD kernel demonstrating the effectiveness of SVM$^{\text{light}}$'s kernel cache. The overhead of constructing a trie on $Q = 112$ examples becomes even more visible: only starting from 50,000 examples

13. The splice dataset can be downloaded from `http://www.fml.tuebingen.mpg.de/raetsch/projects/lsmkl`
14. More precisely the `linadd` and $\mathcal{O}(L)$ block formulation of the WD kernel as proposed in Sonnenburg et al. (2005b) was used.

`linadd` optimization becomes more efficient than the original WD kernel algorithm as the kernel cache cannot hold all kernel elements anymore.[15] Thus it would be appropriate to lower the chunking size Q as can be seen in Table 3.

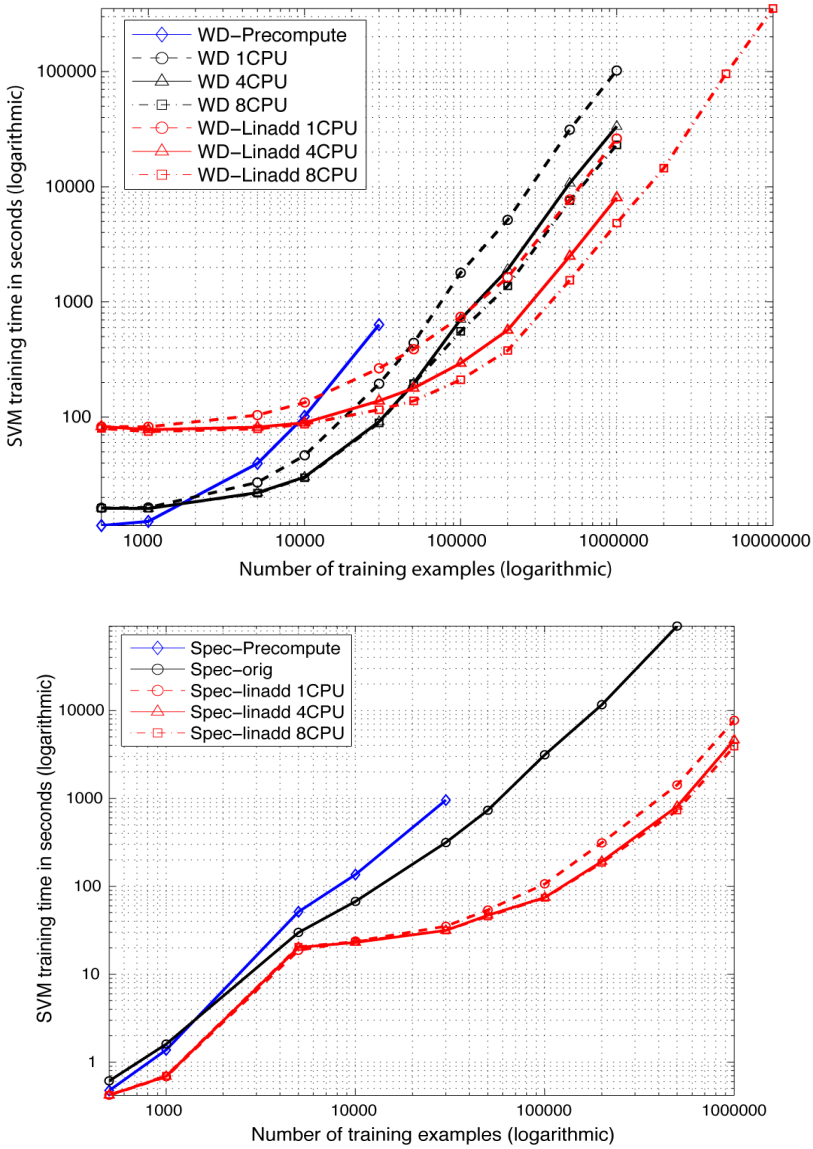

Figure 7: Comparison of the running time of the different SVM training algorithms using the weighted degree kernel. Note that as this is a log-log plot small appearing distances are large for larger N and that each slope corresponds to a different exponent. In the upper figure the Weighted Degree kernel training times are measured, the lower figure displays Spectrum kernel training times.

15. When single precision 4-byte floating point numbers are used, caching all kernel elements is possible when training with up to 16384 examples.

N	*WDPre*	*WD1*	*WD4*	*WD8*	*LinWD1*	*LinWD4*	*LinWD8*
500	12	17	17	17	83	83	80
1,000	13	17	17	17	83	78	75
5,000	40	28	23	22	105	82	80
10,000	102	47	31	30	134	90	87
30,000	636	195	92	90	266	139	116
50,000	-	441	197	196	389	179	139
100,000	-	1,794	708	557	740	294	212
200,000	-	5,153	1,915	1,380	1,631	569	379
500,000	-	31,320	10,749	7,588	7,757	2,498	1,544
1,000,000	-	102,384	33,432	23,127	26,190	8,053	4,835
2,000,000	-	-	-	-	-	-	14,493
5,000,000	-	-	-	-	-	-	95,518
10,000,000	-	-	-	-	-	-	353,227

N	*SpecPre*	*Spec*	*LinSpec1*	*LinSpec4*	*LinSpec8*
500	1	1	1	1	1
1,000	2	2	1	1	1
5,000	52	30	19	21	21
10,000	136	68	24	23	24
30,000	957	315	36	32	32
50,000	-	733	54	47	46
100,000	-	3,127	107	75	74
200,000	-	11,564	312	192	185
500,000	-	91,075	1,420	809	728
1,000,000	-	-	7,676	4,607	3,894

Table 1: **(top)** Speed Comparison of the original single CPU Weighted Degree Kernel algorithm (*WD1*) in SVM$^{\text{light}}$ training, compared to the four (*WD4*)and eight (*WD8*) CPUs parallelized version, the precomputed version (Pre) and the `linadd` extension used in conjunction with the original WD kernel for 1,4 and 8 CPUs (*LinWD1, LinWD4, LinWD8*). **(bottom)** Speed Comparison of the spectrum kernel without (*Spec*) and with `linadd` (*LinSpec1, LinSpec4, LinSpec8* using 1,4 and 8 processors). *SpecPre* denotes the precomputed version. The first column shows the sample size N of the data set used in SVM training while the following columns display the time (measured in seconds) needed in the training phase.

The `linadd` formulation outperforms the original WD kernel by a factor of 3.9 on a million examples. The picture is similar for the spectrum kernel, here speedups of factor 64 on $500,000$ examples are reached which stems from the fact that explicit maps (and not tries as in the WD kernel case) as discussed in Section 3.1.2 could be used leading to a `lookup` cost of $\mathcal{O}(1)$ and a dramatically reduced map construction time. For that reason the parallelization effort benefits the WD kernel more than the Spectrum kernel: on one million

examples the parallelization using 4 CPUs (8 CPUs) leads to a speedup of factor 3.25 (5.42) for the WD kernel, but only 1.67 (1.97) for the Spectrum kernel. Thus parallelization will help more if the kernel computation is slow. Training with the original WD kernel with a sample size of $1,000,000$ takes about 28 hours, the `linadd` version still requires 7 hours while with the 8 CPU parallel implementation only about 6 hours and in conjunction with the `linadd` optimization a single hour and 20 minutes are needed. Finally, training on 10 million examples takes about 4 days. Note that this data set is already $2.1GB$ in size.

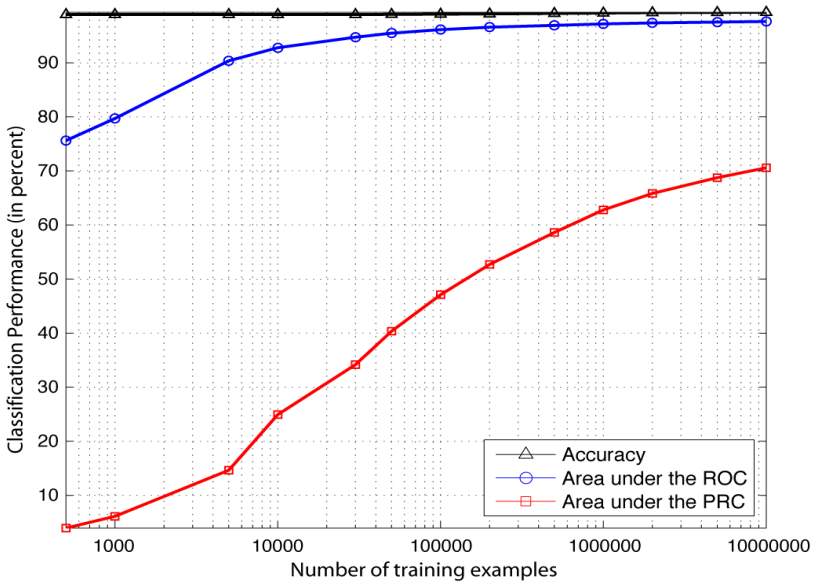

Figure 8: Comparison of the classification performance of the Weighted Degree kernel based SVM classifier for different training set sizes. The area under the Receiver Operator Characteristic (ROC) Curve, the area under the Precision Recall Curve (PRC) as well as the classification accuracy are displayed (in percent). Note that as this is a very unbalanced dataset, the accuracy and the area under the ROC curve are less meaningful than the area under the PRC.

Classification Performance Figure 8 and Table 2 show the classification performance in terms of classification accuracy, area under the Receiver Operator Characteristic (ROC) Curve (Metz, 1978, Fawcett, 2003) and the area under the Precision Recall Curve (PRC) (see e.g. Davis and Goadrich (2006)) of SVMs on the human splice data set for different data set sizes using the WD kernel.

Recall the definition of the ROC and PRC curves: The sensitivity (or recall) is defined as the fraction of correctly classified positive examples among the total number of positive examples, i.e. it equals the true positive rate $TPR = TP/(TP + FN)$. Analogously, the fraction $FPR = FP/(TN + FP)$ of negative examples wrongly classified positive is called the false positive rate. Plotting FPR against TPR results in the Receiver Operator Characteristic Curve (ROC) Metz (1978), Fawcett (2003). Plotting the true positive rate against the positive predictive value (also precision) $PPV = TP/(FP + TP)$, i.e. the fraction of

correct positive predictions among all positively predicted examples, one obtains the Precision Recall Curve (PRC) (see e.g. Davis and Goadrich (2006)). Note that as this is a very unbalanced dataset the accuracy and the area under the ROC curve are almost meaningless, since both measures are independent of class ratios. The more sensible auPRC, however, steadily increases as more training examples are used for learning. Thus one should train using all available data to obtain state-of-the-art results.

N	*Accuracy*	*auROC*	*auPRC*
500	98.93	75.61	3.97
1,000	98.93	79.70	6.12
5,000	98.93	90.38	14.66
10,000	98.93	92.79	24.95
30,000	98.93	94.73	34.17
50,000	98.94	95.48	40.35
100,000	98.98	96.13	47.11
200,000	99.05	96.58	52.70
500,000	99.14	96.93	58.62
1,000,000	99.21	97.20	62.80
2,000,000	99.26	97.36	65.83
5,000,000	99.31	97.52	68.76
10,000,000	99.35	97.64	70.57
10,000,000	-	96.03*	44.64*

Table 2: Comparison of the classification performance of the Weighted Degree kernel based SVM classifier for different training set sizes. The area under the ROC curve (*auROC*), the area under the Precision Recall Curve (*auPRC*) as well as the classification accuracy (*Accuracy*) are displayed (in percent). Larger values are better. A optimal classifier would achieve 100% Note that as this is a very unbalanced dataset the accuracy and the area under the ROC curve are almost meaningless. For comparison, the classification performance achieved using a 4th order Markov chain on 10 million examples (order 4 was chosen based on model selection, where order 1 to 8 using several pseudo-counts were tried) is displayed in the last row (marked *).

Varying SVM$^{\text{light}}$'s `qpsize` parameter As discussed in Section 3.1.3 and Algorithm 3, using the `linadd` algorithm for computing the output for all training examples w.r.t. to some working set can be speed up by a factor of Q (i.e. the size of the quadratic subproblems, termed `qpsize` in SVM$^{\text{light}}$). However, there is a trade-off in choosing Q as solving larger quadratic subproblems is expensive (quadratic to cubic effort). Table 3 shows the dependence of the computing time from Q and N. For example the gain in speed between choosing $Q = 12$ and $Q = 42$ for 1 million of examples is 54%. Sticking with a mid-range Q (here $Q = 42$) seems to be a good idea for this task. However, a large variance can be observed, as the SVM training time depends to a large extend on which Q variables are

		Q				
N	112	12	32	42	52	72
500	83	**4**	**1**	22	68	67
1,000	83	**7**	**7**	**11**	34	60
5,000	105	**15**	**21**	33	31	68
10,000	134	**32**	**38**	54	67	97
30,000	266	**128**	**128**	**127**	160	187
50,000	389	258	**217**	242	252	309
100,000	740	696	**494**	585	573	643
200,000	1,631	1,875	1,361	**1,320**	1,417	1,610
500,000	7,757	9,411	6,558	**6,203**	6,583	7,883
1,000,000	26,190	31,145	20,831	**20,136**	21,591	24,043

Table 3: Influence on training time when varying the size of the quadratic program Q in SVM$^{\text{light}}$, when using the `linadd` formulation of the WD kernel. While training times do not vary dramatically one still observes the tendency that with larger sample size a larger Q becomes optimal. The $Q = 112$ column displays the same result as column *LinWD1* in Table 1.

selected in each optimization step. For example on the related *C. elegans* splice data set $Q = 141$ was optimal for large sample sizes while a midrange $Q = 71$ lead to the overall best performance. Nevertheless, one observes the trend that for larger training set sizes slightly larger subproblems sizes decrease the SVM training time.

4.2.2 Benchmarking MKL

The WD kernel of degree 20 consist of a weighted sum of 20 sub-kernels each counting matching d-mers, for $d = 1, \ldots, 20$. Using MKL we learned the weighting on the splice site recognition task for one million examples as displayed in Figure 5 and discussed in Section 4.1.3. Focusing on a speed comparison we now show the obtained training times for the different MKL algorithms applied to learning weightings of the WD kernel on the splice site classification task. To do so, several MKL-SVMs were trained using precomputed kernel matrices (*PreMKL*), kernel matrices which are computed on the fly employing kernel caching (*MKL*[16]), MKL using the `linadd` extension (*LinMKL1*) and `linadd` with its parallel implementation[17] (*LinMKL4* and *LinMKL8* - on 4 and 8 CPUs). The results are displayed in Table 4 and in Figure 9. While precomputing kernel matrices seems beneficial, it cannot be applied to large scale cases (e.g. $> 10,000$ examples) due to the $\mathcal{O}(KN^2)$ memory constraints of storing the kernel matrices.[18] On-the-fly-computation of the kernel matrices is computationally extremely demanding, but since kernel caching[19] is used, it is still possible

16. Algorithm 2
17. Algorithm 2 with the `linadd` extensions including parallelization of Algorithm 4
18. Using 20 kernels on 10,000 examples requires already 7.5GB, on 30,000 examples 67GB would be required (both using single precision floats)
19. Each kernel has a cache of 1GB

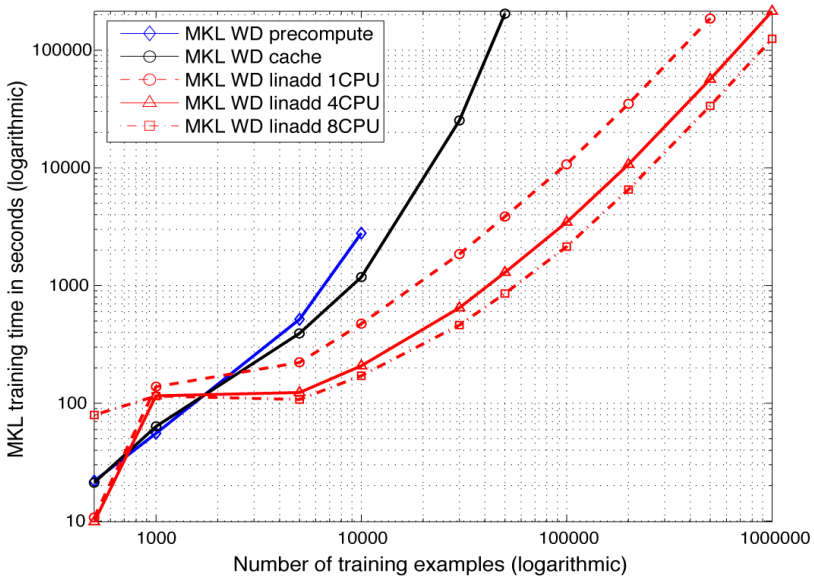

Figure 9: Comparison of the running time of the different MKL algorithms when used with the weighted degree kernel. Note that as this is a log-log plot, small appearing distances are large for larger N and that each slope corresponds to a different exponent.

on 50,000 examples in about 57 hours. Note that no WD-kernel specific optimizations are involved here, so one expects a similar result for arbitrary kernels.

The `linadd` variants outperform the other algorithms by far (speedup factor 53 on 50,000 examples) and are still applicable to datasets of size up to one million. Note that without parallelization MKL on one million examples would take more than a week, compared with 2.5 (2) days in the quad-CPU (eight-CPU) version. The parallel versions outperform the single processor version from the start achieving a speedup for 10,000 examples of 2.27 (2.75), quickly reaching a plateau at a speedup factor of 2.98 (4.49) at a level of $50,000$ examples and approaching a speedup factor of 3.28 (5.53) on 500,000 examples (efficiency: 82% (69%)). Note that the performance gain using 8 CPUs is relatively small as e.g. solving the QP and constructing the tree is not parallelized.

5. Conclusion

In the first part of the paper we have proposed a simple, yet efficient algorithm to solve the multiple kernel learning problem for a large class of loss functions. The proposed method is able to exploit the existing single kernel algorithms, thereby extending their applicability. In experiments we have illustrated that MKL for classification and regression can be useful for automatic model selection and for obtaining comprehensible information about the learning problem at hand. It would be of interest to develop and evaluate MKL algorithms for unsupervised learning such as Kernel PCA and one-class classification and to try different losses on the kernel weighting $\boldsymbol{\beta}$ (such as L_2). In the second part we proposed performance

N	*PreMKL*	*MKL*	*LinMKL1*	*LinMKL4*	*LinMKL8*
500	22	22	11	10	80
1,000	56	64	139	116	116
5,000	518	393	223	124	108
10,000	2,786	1,181	474	209	172
30,000	-	25,227	1,853	648	462
50,000	-	204,492	3,849	1292	857
100,000	-	-	10,745	3,456	2,145
200,000	-	-	34,933	10,677	6,540
500,000	-	-	185,886	56,614	33,625
1,000,000	-	-	-	214,021	124,691

Table 4: Speed Comparison when determining the WD kernel weight by Multiple Kernel Learning using the chunking algorithm (MKL) and MKL in conjunction with the (parallelized) `linadd` algorithm using 1, 4, and 8 processors (*LinMKL1, LinMKL4, LinMKL8*). The first column shows the sample size N of the data set used in SVM training while the following columns display the time (measured in seconds) needed in the training phase.

enhancements to make large scale MKL practical: the SILP wrapper, SILP chunking and (for the special case of kernels that can be written as an inner product of sparse feature vectors, e.g., string kernels) the `linadd` algorithm, which also speeds up standalone SVM training. For the standalone SVM using the spectrum kernel we achieved speedups of factor 64 (for the weighted degree kernel, about 4). For MKL we gained a speedup of factor 53. Finally we proposed a parallel version of the `linadd` algorithm running on a 8 CPU multiprocessor system which lead to *additional* speedups of factor up to 5.5 for MKL, and 5.4 for vanilla SVM training.

Acknowledgments

The authors gratefully acknowledge partial support from the PASCAL Network of Excellence (EU #506778), DFG grants JA 379 / 13-2 and MU 987/2-1. We thank Guido Dornhege, Olivier Chapelle, Cheng Soon Ong, Joaquin Quiñoñero Candela, Sebastian Mika, Jason Weston, Manfred Warmuth and K.-R. Müller for great discussions.

References

F. R. Bach, G. R. G. Lanckriet, and M. I. Jordan. Multiple kernel learning, conic duality, and the SMO algorithm. In C. E. Brodley, editor, *Twenty-first international conference on Machine learning*. ACM, 2004.

K. P. Bennett, M. Momma, and M. J. Embrechts. MARK: a boosting algorithm for heterogeneous kernel models. In *Proceedings of the Eighth ACM SIGKDD International*

Conference on Knowledge Discovery and Data Mining, pages 24–31. ACM, 2002.

J. Bi, T. Zhang, and K. P. Bennett. Column-generation boosting methods for mixture of kernels. In W. Kim, R. Kohavi, J. Gehrke, and W. DuMouchel, editors, *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 521–526. ACM, 2004.

S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, UK, 2004.

O. Chapelle, V. Vapnik, O. Bousquet, and S. Mukherjee. Choosing multiple parameters for support vector machines. *Machine Learning*, 46(1–3):131–159, 2002.

C. Cortes and V.N. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

J. Davis and M. Goadrich. The relationship between precision-recall and roc curves. Technical report #1551, University of Wisconsin Madison, January 2006.

T. Fawcett. Roc graphs: Notes and practical considerations for data mining researchers. Technical report hpl-2003-4, HP Laboratories, Palo Alto, CA, USA, January 2003.

E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

Y. Grandvalet and S. Canu. Adaptive scaling for feature selection in SVMs. In S. Thrun S. Becker and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 553–560, Cambridge, MA, 2003. MIT Press.

R. Hettich and K. O. Kortanek. Semi-infinite programming: Theory, methods and applications. *SIAM Review*, 3:380–429, 1993.

T. Joachims. Text categorization with support vector machines: Learning with many relevant features. In C. Nédellec and C. Rouveirol, editors, *ECML '98: Proceedings of the 10th European Conference on Machine Learning*, Lecture Notes in Computer Science, pages 137–142, Berlin / Heidelberg, 1998. Springer-Verlag.

T. Joachims. Making large–scale SVM learning practical. In B. Schölkopf, C.J.C. Burges, and A.J. Smola, editors, *Advances in Kernel Methods — Support Vector Learning*, pages 169–184, Cambridge, MA, USA, 1999. MIT Press.

G.R.G. Lanckriet, T. De Bie, N. Cristianini, M.I. Jordan, and W.S. Noble. A statistical framework for genomic data fusion. *Bioinformatics*, 20:2626–2635, 2004.

C. Leslie, E. Eskin, and W.S. Noble. The spectrum kernel: A string kernel for SVM protein classification. In R. B. Altman, A. K. Dunker, L. Hunter, K. Lauderdale, and T. E. Klein, editors, *Proceedings of the Pacific Symposium on Biocomputing*, pages 564–575, Kaua'i, Hawaii, 2002.

C. Leslie, R. Kuang, and E. Eskin. Inexact matching string kernels for protein classification. In *Kernel Methods in Computational Biology*, MIT Press series on Computational Molecular Biology, pages 95–112. MIT Press, 2004.

R. Meir and G. Rätsch. An introduction to boosting and leveraging. In S. Mendelson and A. Smola, editors, *Proc. of the first Machine Learning Summer School in Canberra*, LNCS, pages 119–184. Springer, 2003.

C.E. Metz. Basic principles of ROC analysis. *Seminars in Nuclear Medicine*, VIII(4), October 1978.

C. S. Ong, A. J. Smola, and R. C. Williamson. Hyperkernels. In S. Thrun S. Becker and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, volume 15, pages 478–485, Cambridge, MA, 2003. MIT Press.

J. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schölkopf, C.J.C. Burges, and A.J. Smola, editors, *Advances in Kernel Methods — Support Vector Learning*, pages 185–208, Cambridge, MA, USA, 1999. MIT Press.

G. Rätsch. *Robust Boosting via Convex Optimization*. PhD thesis, University of Potsdam, Potsdam, Germany, 2001.

G. Rätsch and S. Sonnenburg. *Accurate Splice Site Prediction for Caenorhabditis Elegans*, pages 277–298. MIT Press series on Computational Molecular Biology. MIT Press, 2004.

G. Rätsch and M.K. Warmuth. Efficient margin maximization with boosting. *Journal of Machine Learning Research*, 6:2131–2152, 2005.

G. Rätsch, A. Demiriz, and K. Bennett. Sparse regression ensembles in infinite and finite hypothesis spaces. *Machine Learning*, 48(1–3):193–221, 2002. Special Issue on New Methods for Model Selection and Model Combination. Also NeuroCOLT2 Technical Report NC-TR-2000-085.

G. Rätsch, S. Sonnenburg, and B. Schölkopf. RASE: Recognition of alternatively spliced exons in C. elegans. *Bioinformatics*, 21:i369–i377, 2005.

B. Schölkopf and A. J. Smola. *Learning with Kernels*. MIT Press, Cambridge, MA, 2002.

S. Sonnenburg, G. Rätsch, and C. Schäfer. Learning interpretable SVMs for biological sequence classification. In S. Miyano, J. P. Mesirov, S. Kasif, S. Istrail, P. A. Pevzner, and M. Waterman, editors, *Research in Computational Molecular Biology, 9th Annual International Conference, RECOMB 2005*, volume 3500, pages 389–407. Springer-Verlag Berlin Heidelberg, 2005a.

S. Sonnenburg, G. Rätsch, and B. Schölkopf. Large scale genomic sequence SVM classifiers. In L. D. Raedt and S. Wrobel, editors, *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pages 849–856, New York, NY, USA, 2005b. ACM Press.

M.K. Warmuth, J. Liao, and G. Rätsch. Totally corrective boosting algorithms that maximize the margin. In *ICML '06: Proceedings of the 23nd international conference on Machine learning*. ACM Press, 2006.

Appendix A. Derivation of the MKL Dual for Generic Loss Functions

We start from the MKL primal problem Equation (9):

$$\begin{aligned}
\min \quad & \frac{1}{2}\left(\sum_{k=1}^{K}\|\mathbf{w}_k\|\right)^2 + \sum_{i=1}^{N} L(f(\mathbf{x}_i), y_i) \\
\text{w.r.t.} \quad & \mathbf{w} = (\mathbf{w}_1, \ldots, \mathbf{w}_K) \in \mathbb{R}^{D_1} \times \cdots \times \mathbb{R}^{D_K} \\
\text{s.t.} \quad & f(\mathbf{x}_i) = \sum_{k=1}^{K} \langle \Phi_k(\mathbf{x}_i), \mathbf{w}_k \rangle + b, \;\; \forall i = 1, \ldots, N
\end{aligned}$$

Introducing $u \in \mathbb{R}$ allows us to move $\sum_{k=1}^{K}\|\mathbf{w}_k\|$ into the constraints and leads to the following equivalent problem

$$\begin{aligned}
\min \quad & \frac{1}{2}u^2 + \sum_{i=1}^{N} L(f(\mathbf{x}_i), y_i) \\
\text{w.r.t.} \quad & u \in \mathbb{R}, \; (\mathbf{w}_1, \ldots, \mathbf{w}_K) \in \mathbb{R}^{D_1} \times \cdots \times \mathbb{R}^{D_K} \\
\text{s.t.} \quad & f(\mathbf{x}_i) = \sum_{k=1}^{K} \langle \Phi_k(\mathbf{x}_i), \mathbf{w}_k \rangle + b, \; \forall i = 1, \ldots, N \\
& \sum_{k=1}^{K}\|\mathbf{w}_k\| \leq u
\end{aligned}$$

Using $t_k \in \mathbb{R}, \; k = 1, \ldots, K$, it can be equivalently transformed into

$$\begin{aligned}
\min \quad & \frac{1}{2}u^2 + \sum_{i=1}^{N} L(f(\mathbf{x}_i), y_i) \\
\text{w.r.t.} \quad & u \in \mathbb{R}, \; t_k \in \mathbb{R}, \mathbf{w}_k \in \mathbb{R}^{D_k}, \; \forall k = 1, \ldots, K \\
\text{s.t.} \quad & f(\mathbf{x}_i) = \sum_{k=1}^{K} \langle \Phi_k(\mathbf{x}_i), \mathbf{w}_k \rangle + b, \; \forall i = 1, \ldots, N \\
& \|\mathbf{w}_k\| \leq t_k, \;\; \sum_{k=1}^{K} t_k \leq u.
\end{aligned}$$

Recall that the second-order cone of dimensionality D is defined as

$$\mathcal{K}_D = \{(\mathbf{x}, c) \in \mathbb{R}^D \times \mathbb{R}, \; \|\mathbf{x}\|_2 \leq c\}.$$

We can thus reformulate the original MKL primal problem (Equation (9)) using the following *equivalent* second-order cone program, as the norm constraint on $\mathbf{w}_k$ is implicitly taken care of:

Conic Primal

$$\begin{aligned}
\min \quad & \frac{1}{2}u^2 + \sum_{i=1}^{N} L(f(\mathbf{x}_i), y_i) \\
\text{w.r.t.} \quad & u \in \mathbb{R},\ t_k \in \mathbb{R}, (\mathbf{w}_k, t_k) \in \mathcal{K}_{D_k},\ \forall k = 1, \ldots, K \\
\text{s.t.} \quad & f(\mathbf{x}_i) = \sum_{k=1}^{K} \langle \Phi_k(\mathbf{x}_i), \mathbf{w}_k \rangle + b,\ \forall i = 1, \ldots, N \\
& \sum_{k=1}^{K} t_k \leq u
\end{aligned}$$

We are now going to derive the conic dual following the recipe of Boyd and Vandenberghe (2004) (see p. 266). First we derive the conic Lagrangian and then using the infimum w.r.t. the primal variables in order to obtain the conic dual. We therefore introduce Lagrange multipliers $\boldsymbol{\alpha} \in \mathbb{R}^K$, $\gamma \in \mathbb{R}$, $\gamma \geq 0$ and $(\boldsymbol{\lambda}_k, \mu_k) \in {\mathcal{K}^*}_D$ living on the self dual cone ${\mathcal{K}^*}_D = \mathcal{K}_D$. Then the conic Lagrangian is given as

$$\begin{aligned}
\mathcal{L}(\mathbf{w}, b, \mathbf{t}, u, \boldsymbol{\alpha}, \gamma, \boldsymbol{\lambda}, \boldsymbol{\mu}) \quad = \quad & \frac{1}{2}u^2 + \sum_{i=1}^{N} L(f(\mathbf{x}_i), y_i) - \sum_{i=1}^{N} \alpha_i f(\mathbf{x}_i) + \\
& + \sum_{i=1}^{N} \alpha_i \sum_{k=1}^{K} \left(\langle \Phi_k(\mathbf{x}_i), \mathbf{w}_k \rangle + b \right) + \gamma \left(\sum_{k=1}^{K} t_k - u \right) - \sum_{k=1}^{K} \left(\langle \boldsymbol{\lambda}_k, \mathbf{w}_k \rangle + \mu_k t_k \right).
\end{aligned}$$

To obtain the dual, the derivatives of the Lagrangian w.r.t. the primal variables, $\mathbf{w}, b, \mathbf{t}, u$ have to vanish which leads to the following constraints

$$\begin{aligned}
\partial_{\mathbf{w}_k} \mathcal{L} \quad &= \quad \sum_{i=1}^{N} \alpha_i \Phi_k(\mathbf{x}_i) - \boldsymbol{\lambda}_k \ \Rightarrow\ \boldsymbol{\lambda}_k = \sum_{i=1}^{N} \alpha_i \Phi_k(\mathbf{x}_i) \\
\partial_b \mathcal{L} \quad &= \quad \sum_{i=1}^{N} \alpha_i \ \Rightarrow\ \sum_{i=1}^{N} \alpha_i = 0 \\
\partial_{t_k} \mathcal{L} \quad &= \quad \gamma - \mu_k \ \Rightarrow\ \gamma = \mu_k \\
\partial_u \mathcal{L} \quad &= \quad u - \gamma \Rightarrow \gamma = u \\
\partial_{f(\mathbf{x}_i)} \mathcal{L} \quad &= \quad L'(f(\mathbf{x}_i), y_i) - \alpha_i \Rightarrow f(\mathbf{x}_i) = {L'}^{-1}(\alpha_i, y_i).
\end{aligned}$$

In the equation L' is the derivative of the loss function w.r.t. $f(x)$ and ${L'}^{-1}$ is the inverse of L' (w.r.t. $f(x)$) for which to exist L is required to be strictly convex and differentiable. We now plug in what we have obtained above, which makes $\boldsymbol{\lambda}_k$, μ_k and all of the primal

variables vanish. Thus the dual function is

$$\begin{aligned} D(\boldsymbol{\alpha},\gamma) &= -\frac{1}{2}\gamma^2 + \sum_{i=1}^{N} L(L'^{-1}(\alpha_i,y_i),y_i) - \sum_{i=1}^{N}\alpha_i L'^{-1}(\alpha_i,y_i) + \\ &\quad + \sum_{i=1}^{N}\alpha_i \sum_{k=1}^{K}\langle \Phi_k(\mathbf{x}_i),\mathbf{w}_k\rangle - \sum_{k=1}^{K}\sum_{i=1}^{N}\alpha_i\langle\Phi_k(\mathbf{x}_i),\mathbf{w}_k\rangle \\ &= -\frac{1}{2}\gamma^2 + \sum_{i=1}^{N} L(L'^{-1}(\alpha_i,y_i),y_i) - \sum_{i=1}^{N}\alpha_i L'^{-1}(\alpha_i,y_i). \end{aligned}$$

As constraints remain $\gamma \geq 0$, due to the bias $\sum_{i=1}^{N}\alpha_i = 0$ and the second-order cone constraints

$$\|\boldsymbol{\lambda}_k\| = \left\| \sum_{i=1}^{N}\alpha_i\Phi_k(\mathbf{x}_i)\right\|_2 \leq \gamma, \quad \forall k = 1,\ldots,K.$$

This leads to:

$$\begin{aligned} \max \quad & -\frac{1}{2}\gamma^2 + \sum_{i=1}^{N} L(L'^{-1}(\alpha_i,y_i),y_i) - \sum_{i=1}^{N}\alpha_i L'^{-1}(\alpha_i,y_i) \\ \text{w.r.t.} \quad & \gamma\in\mathbb{R},\ \boldsymbol{\alpha}\in R^N \\ \text{s.t.} \quad & \gamma \geq 0,\ \sum_{i=1}^{N}\alpha_i = 0 \\ & \left\|\sum_{i=1}^{N}\alpha_i\Phi_k(\mathbf{x}_i)\right\|_2 \leq \gamma,\ \forall k = 1,\ldots,K \end{aligned}$$

Squaring the latter constraint, multiplying by $\frac{1}{2}$, relabeling $\frac{1}{2}\gamma^2 \mapsto \gamma$ and dropping the $\gamma \geq 0$ constraint as it is fulfilled implicitly, we obtain the MKL dual for arbitrary strictly convex loss functions.

Conic Dual

$$\begin{aligned} \min \quad & \underbrace{\gamma - \sum_{i=1}^{N} L(L'^{-1}(\alpha_i,y_i),y_i) + \sum_{i=1}^{N}\alpha_i L'^{-1}(\alpha_i,y_i)}_{:=T} \\ \text{w.r.t.} \quad & \gamma\in\mathbb{R},\ \boldsymbol{\alpha}\in R^N \\ \text{s.t.} \quad & \sum_{i=1}^{N}\alpha_i = 0 \\ & \frac{1}{2}\left\|\sum_{i=1}^{N}\alpha_i\Phi_k(\mathbf{x}_i)\right\|_2^2 \leq \gamma,\ \forall k = 1,\ldots,K. \end{aligned}$$

Finally adding the second term in the objective (T) to the constraint on γ and relabeling $\gamma + T \mapsto \gamma$ leads to the reformulated dual Equation (10), the starting point from which one can derive the SILP formulation in analogy to the classification case.

Appendix B. Loss functions

B.1 Quadratic Loss

For the quadratic loss case $L(x,y) = C(x-y)^2$ we obtain as the derivative $L'(x,y) = 2C(x-y) =: z$ and $L'^{-1}(z,y) = \frac{1}{2C}z + y$ for the inverse of the derivative. Recall the definition of

$$S_k(\boldsymbol{\alpha}) = -\sum_{i=1}^{N} L(L'^{-1}(\alpha_i, y_i), y_i) + \sum_{i=1}^{N} \alpha_i L'^{-1}(\alpha_i, y_i) + \frac{1}{2}\left\|\sum_{i=1}^{N} \alpha_i \Phi_k(\mathbf{x}_i)\right\|_2^2.$$

Plugging in L, L'^{-1} leads to

$$\begin{aligned} S_k(\boldsymbol{\alpha}) &= -\sum_{i=1}^{N}(\frac{1}{2C}\alpha_i + y_i - y_i)^2 + \sum_{i=1}^{N}\alpha_i(\frac{1}{2C}\alpha_i + y_i) + \frac{1}{2}\left\|\sum_{i=1}^{N} \alpha_i \Phi_k(\mathbf{x}_i)\right\|_2^2 \\ &= \frac{1}{4C}\sum_{i=1}^{N}\alpha_i^2 + \sum_{i=1}^{N}\alpha_i y_i + \frac{1}{2}\left\|\sum_{i=1}^{N} \alpha_i \Phi_k(\mathbf{x}_i)\right\|_2^2. \end{aligned}$$

B.2 Logistic Loss

Very similar to the Hinge loss the derivation for the logistic loss $L(x,y) = \log(1+e^{-xy})$ will be given for completeness.

$$L'(x,y) = \frac{-ye^{-xy}}{1+e^{-xy}} = -\frac{ye^{(1-xy)}}{1+e^{(1-xy)}} =: z.$$

The inverse function for $y \neq 0$ and $y + z \neq 0$ is given by

$$L'^{-1}(z,y) = -\frac{1}{y}\log\left(-\frac{z}{y+z}\right)$$

and finally we obtain

$$S_k(\boldsymbol{\alpha}) = \sum_{i=1}^{N}\log\left(1 - \frac{\alpha_i}{y_i+\alpha_i}\right) - \sum_{i=1}^{N}\frac{\alpha_i}{y_i}\log\left(-\frac{\alpha_i}{y_i+\alpha_i}\right) + \frac{1}{2}\left\|\sum_{i=1}^{N} \alpha_i \Phi_k(\mathbf{x}_i)\right\|_2^2.$$

B.3 Smooth Hinge Loss

Using the Hinge Loss $L(x,y) = \frac{C}{\sigma}\log(1+e^{\sigma(1-xy)})$ with $\sigma > 0$, $y \in \mathbb{R}$ fixed, $x \in \mathbb{R}$ one obtains as derivative

$$L'(x,y) = \frac{-\sigma C y e^{\sigma(1-xy)}}{\sigma(1+e^{\sigma(1-xy)})} = -\frac{Cye^{\sigma(1-xy)}}{1+e^{\sigma(1-xy)}} =: z.$$

Note that with y fixed, z is bounded: $0 \leq \mathrm{abs}(z) \leq \mathrm{abs}(Cy)$ and $\mathrm{sign}(y) = -\mathrm{sign}(z)$ and therefore $-\frac{z}{Cy+z} > 0$ for $Cy + z \neq 0$. The inverse function is derived as

$$\begin{aligned}
z + ze^{\sigma(1-xy)} &= -Cye^{\sigma(1-xy)} \\
(Cy+z)e^{\sigma(1-xy)} &= -z \\
e^{\sigma(1-xy)} &= -\frac{z}{Cy+z} \\
\sigma(1-xy) &= \log(-\frac{z}{Cy+z}) \\
1-xy &= \frac{1}{\sigma}\log(-\frac{z}{Cy+z}) \\
x &= \frac{1}{y}(1-\frac{1}{\sigma}\log(-\frac{z}{Cy+z})),\ y \neq 0 \\
L'^{-1}(z,y) &= \frac{1}{y}(1-\frac{1}{\sigma}\log(-\frac{z}{Cy+z}))
\end{aligned}$$

Define $C_1 = \frac{1}{2}\left\|\sum_{i=1}^N \alpha_i \Phi_k(\mathbf{x}_i)\right\|_2^2$ and $C_2 = \sum_{i=1}^N \alpha_i \frac{1}{y_i}\left(1 - \frac{1}{\sigma}\log(-\frac{\alpha_i}{Cy_i+\alpha_i})\right)$

Using these ingredients it follows for $S_k(\boldsymbol{\alpha})$

$$\begin{aligned}
S_k(\boldsymbol{\alpha}) &= -\sum_{i=1}^N L\left(\frac{1}{y_i}\left(1-\frac{1}{\sigma}\log(-\frac{\alpha_i}{Cy_i+\alpha_i})\right), y_i\right) + C_2 + C_1 \\
&= -\sum_{i=1}^N \frac{1}{\sigma}\log\left(1+e^{\sigma\left(1-\left(\frac{y_i}{y_i}\left(1-\frac{1}{\sigma}\log(-\frac{\alpha_i}{Cy_i+\alpha_i})\right)\right)\right)}\right) + C_2 + C_1 \\
&= -\sum_{i=1}^N \frac{1}{\sigma}\log\left(1-\frac{\alpha_i}{Cy_i+\alpha_i}\right) + \sum_{i=1}^N \frac{\alpha_i}{y_i}\left(1-\frac{1}{\sigma}\log(-\frac{\alpha_i}{Cy_i+\alpha_i})\right) + C_1.
\end{aligned}$$